Eric A. Nyamsi

IT-Lösungen auf Basis von SysML und UML

Anwendungsentwicklung mit Eclipse UML Designer und Eclipse Papyrus

Springer Vieweg

IT-Lösungen auf Basis von SysML und UML

Eric A. Nyamsi

IT-Lösungen auf Basis von SysML und UML

Anwendungsentwicklung mit Eclipse UML Designer und Eclipse Papyrus

Springer Vieweg

Eric A. Nyamsi
Karlsruhe, Deutschland

ISBN 978-3-658-29056-6 ISBN 978-3-658-29057-3 (eBook)
https://doi.org/10.1007/978-3-658-29057-3

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Planung/Lektorat: Reinhard Dapper
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Inhaltsverzeichnis

Anwendungen der Modellierung in der Programmierung: Modeling4Programming

1

Eine Software zu modellieren bedeutet, dass die Entwickler Teile der Software mithilfe standardisierter Tools, damit sie effizient ist, beschreiben. Die Tools eines Systems oder einer Software zu beschreiben, stellen moderne Modellierungssprachen dar: SysML und UML sind berühmte Modellierungssprache geworden. Der Titel *„IT-Lösungen auf Basis von SysML und UML“* stellt eine wichtig vernünftige Beschreibung von Software durch Modelle dar und gehört zu großen Anwendungen ohne Frage dazu. SysML ist ein *„Erbe“* von der Modellierungssprache UML und erbt die Modellelemente und Diagrammtypen. Allerdings verbessert das SysML-Profil der UML die Semantik einiger Elementen, wobei die Verbesserung keinen Schaden führt. Das Kernelement von SysML ist der *„Block“*, welche die Struktur zur objektorientierten Programmierung darstellt. Ein Block repräsentiert Komponente wie z. B. Softwareartefakte, Hardware, Funktionen, Prozesse oder auch Personen. Mithilfe der objektorientierten Modellierung werden Klassen oder Kompositionsstrukturdiagramme zu Blockdiagrammen bzw. zu internen Blockdiagrammen umgewandelt. Die aus der UML bekannten Ports wurden als Flowports für den Transport von Stoffen oder physikalischen Eigenschaften geerbt. Das Aktivitätsdiagramm wird um physikalische Objektflüsse und Aktivitätsparameter ergänzt. Andere Diagrammformen wie Use-Case-Diagramm, Sequenzdiagramm oder Zustandsdiagramm sind auch geerbt. SysML stellt sowohl einige, aus UML bekannten Sprachelemente als auch Sprachkonstrukte in Bezug auf das Requirements Engineering. Damit lassen sich nicht nur funktionale Anforderungen mithilfe von Use Cases oder Interaktionsszenarien modellieren, sondern auch nichtfunktionale Anforderungen darstellen. Anforderungen (Requirements) sind in SysML ein wichtiges Sprachkonstrukt. Sie stellen Klassensymbol mit dem Stereotyp *„Requirement“* dar und können wie Klassensymbole eigene Strukturen (Elemente) haben, in denen zum Beispiel eine ID oder der Text der Anforderung selbst stehen kann.

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML,*
https://doi.org/10.1007/978-3-658-29057-3_1

SysML und UML ermöglichen die Anwendungen der Modellierung in der Programmierung und stellen damit den neuen Begriff *„Modeling4Programming“* dar. Zum einen ermöglichen diese Tools den Entwicklern die Beschreibung der Modellierungsdiagramme mithilfe der standardisierten Annotationen in eine Programmiersprache wie z. B. Java. Zum anderen ermöglichen die Modellierungsdiagramme mit SysML/UML die Spezifizierung der Systeme und der Software.

Das Buch fokussiert auf die grafische Spezifizierung der Systeme und Software mithilfe der Modellierungssprachen SysML und UML und deren Anwendung in der Programmierung. Die Modellierung soll das Programmieren der Komponente der Systeme ermöglichen. Z. B. wollen die Entwickler eines Transistors wie z. B. IGBT die Verluste mithilfe der Modellierung der Komponente bestimmen und damit reduzieren, werden die Teile des Systems mit SysML/UML modelliert, anschließen werden mithilfe einer Programmiersprache wie z. B. Java die Verluste mit der Objektorientierten Programmierung erfasst. Die Modellierung hat einen Sinn, wenn ihre Anwendung in der objektorientierten Programmierung das Verstehen der Funktionalität des Systems oder der Software erfolgt. Mit der Systems Modeling Language (SysML) stellte die Object Management Group (OMG) [1] eine formale Modellierungssprache für die Entwicklung eingebetteter Systeme mit Softwareanteilen zur Verfügung.

Das Buch gibt Überblicke über Entwicklung von Tools auf Basis von Open Source Eclipse-Papyrus und UML-Designer. Diese Modellierung-Frameworks ermöglichen die Beschreibung der Systeme oder Software mit SysML/UML bzw. UML und damit deren Anwendungen in der Programmierung. Wobei das Buch fokussiert auf die Programmierungssprache Java. Hierbei werden Diagrammebeschreibungen der Modellierung mithilfe der Java-Klassen in der Programmierung angewendet.

Ziel der Modellierung mit beiden Frameworks ist es, die modellbasierte Entwicklung der Systeme mithilfe der Modellierungsdiagramme von SysML/UML zu realisieren. Das Buch kombiniert die Vorteile von SysML und UML mithilfe der Frameworks Eclipse-Papyrus und UML-Designer.

Der Begriff *„Modeling4Programming“* enthält zwei Elemente: Modellierung mit SysML/UML und Programmierung mit objektorientierter Programmierungssprache wie z. B. Java. Das Kernthema des Buches ist die Kombination SysML/UML und Java/C++. Deshalb ist die Modellierung mittels Diagramme die Beschreibung der Programmierung. Modellierung soll den Inhalt der Programmierung darstellen. Die Frameworks Papyrus und UML-Designer ermöglichen mithilfe der Diagramme die Realisierung der modellbasierten Entwicklung. Das Buch stellt mithilfe der Frameworks Papyrus und Obeo-UML-Designer Abhängigkeitsbeziehungen wie z. B. Klassendiagrammen, Kontextdiagrammen, ggf. Datenflussdiagramme dar.

1.1 Papyrus Framework zur Modellierung mit SysML/UML

Papyrus ist ein erweiterbares Framework, das sowohl bestehende Modellierungssprachen à la UML 2.5 und SysML 1.4 unterstützt, als auch die Erstellung eigener domänenspezifischer Modellierungssprachen erlaubt. Durch diese Anpassbarkeit deckt Papyrus potenziell verschiedenste Entwicklungsaspekte ab, darunter z. B. modellbasierte Simulationen, modellbasierte Tests und Sicherheitsanalysen. Damals war Papyrus eine Komponente, die zu bestehenden Eclipse-Installationen hinzugefügt werden konnte. Papyrus ist vieles: Im Bereich Open Source wird das Framework als UML-Modellierungswerkzeug angewendet, wobei man es herunterladen und sofort mit der Modellierung beginnen kann. Von Eclipse-Papyrus 2.0.1-, 2.0.2-, 2.0.3- Neon-release bis 3.0.0- und 3.0.1-Oxygen-release wurden Installation und allgemeine User Experience verbessert, um den Einstieg zu erleichtern. Es ist zu bemerken, dass bei dem Schritt von Eclipse-Kepler zu Luna sich bei Papyrus einiges verbessert hat und es auch mit Eclipse-Mars viel positives Feedback von den Entwicklern und Kunden gibt.

Die größte Stärke von Papyrus liegt in seiner Anpassbarkeit. Das Papyrus-Framework ermöglicht Anwendern viele Spielräume bei der Modifizierung, um die Voraussetzungen einer bestimmten Domäne, Industrie oder Sprache zu erfüllen. Das macht Papyrus zu einem starken Framework, um UML-basierte, domänenspezifische Modellierungssprachen (DSML) zu entwickeln.

Papyrus stellt ein Eclipse-Projekt für die SysML- und UML-Modellierung dar. Von allen UML-Werkzeugen, die viele Entwickler bisher angewendet haben, hat das Framework Papyrus die zur Spezifikation konformste Umsetzung von UML 2.5. In dieser Hinsicht ist Papyrus der *„Boss der UML-SysML-Modellierungs-Liga“*. Sowohl Kunden als auch Entwickler sind von dem Framework begeistert. Im Bereich sicherheitskritischer Systeme ermöglicht Papyrus den Entwicklern das Herstellen eines guten Produkts.

1.1.1 UML-Aktivitätsdiagramme mit Eclipse-Papyrus

Ziel des Ansatzes der Aktivitätsdiagramme ist es, Verhalten der Systeme bezüglich der Kontrolle- und des Datenflussmodells zu modellieren. Hierbei beschreiben die Prozesse aus der Sicht der Anwender, wobei die Aktivitätsdiagramme eine Kommunikationsgrundlage zwischen dem Software-Architekt und den Anwendern darstellen. Abb. 1.1 zeigt ein Aktivitätsdiagramm von Tasks zum Analysieren von Schwingkreiswechselrichter. Gemäß Abb. 1.1 verfügt das Aktivitätsdiagramm über Notationselemente wie z. B. Knoten, Aktionen oder Kontrollflüsse. Abb. 1.1 zeigt eine parallele Modellierung mithilfe von Fork-Phase, Work-Phase und Join-Phase. Mithilfe der Aktivitätsdiagramme wird die Parallelität die Aktivitäten ausführen. Gemäß Abb. 1.1 ist es zu erkennen, dass Tasks sich parallel ausführen lassen, da hierbei die Tasks keine Abhängigkeiten bezüglich der Reihenfolge aufweisen. Gemäß Abb. 1.1 lassen sich Tasks-Ausführungen sowohl sequenziell verknüpfen als auch mit *„und-“* beziehungsweise *„oder-“* Operationen zusammenführen.

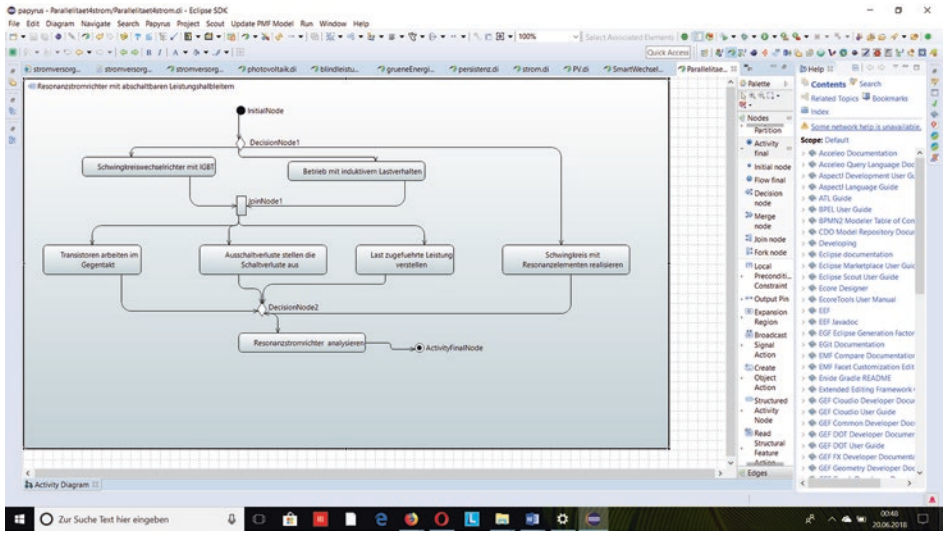


Abb. 1.1 Aktivitätsdiagramm von Tasks zum Analysieren von Schwingkreiswechelrichter

1.1.2 Zustandsdiagramme mit Eclipse-Papyrus

Zustandsdiagramme beschreiben das Verhalten der Komponenten des Lebenszyklusses mithilfe der Zustände und Zustandsübergänge. Hierbei werden *„Classifier“* wie z. B. Klasse oder Systeme beschrieben. Ziel des Einsatzes der Zustandsdiagramme ist es, Reaktionen von Systemen zu beschreiben [2]. Zum Modellieren des Verhaltens der Schnittstellen mit UML haben die Zustandsdiagramme einen Vorteil gegenüber Aktivitätsdiagramme. Beispielsweise werden Zustandsdiagramme zum Beschreiben der Kommunikation zwischen zwei Komponenten in Bezug auf Schnittstellendesign angewendet.

Abb. 1.2 zeigt das Zustandsdiagramm zum Modellieren des Verhaltens von Schnittstellen mit UML in Bezug auf die Beschreibung der Kommunikation zwischen zwei Komponenten wie z. B. Navigation und Human Machine Interface abgekürzt HMI (Display) in einem automobilen Infotainment-System. Gemäß Abb. 1.2 stehen Modellierungselemente u. a. Region, Zustand, Transition oder Initialzustand zur Verfügung. Transitionen verfügen über Auslöser, genannt *„Trigger“*, in Form des erwarteten, logischen Ereignisses und der Angabe über sendende und empfangende Komponenten [3]. Abb. 1.2 stellt eine Beschreibung des dynamischen Schnittstellenverhaltens mithilfe einer domänenspezifischen Sprache (DSL) wie z. B. Obeo Designer im automobilen Infotainment dar.

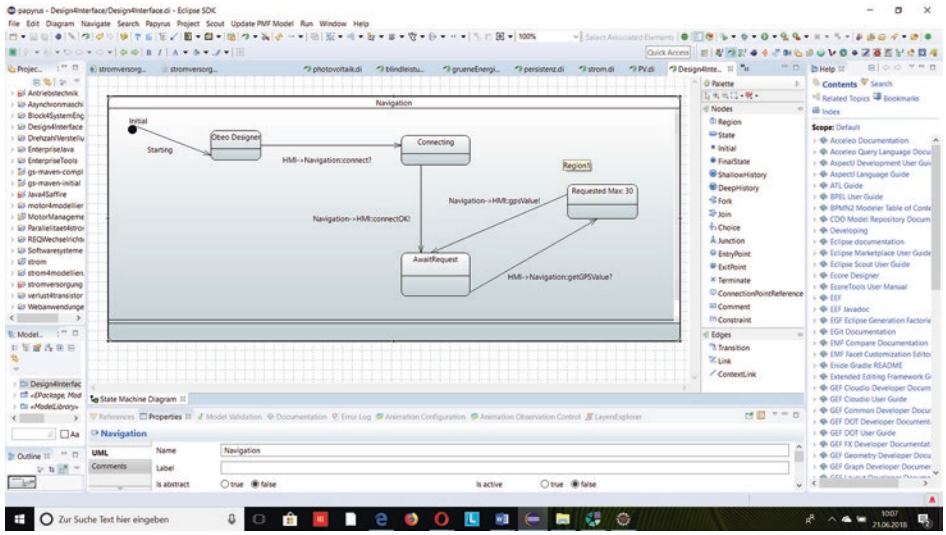


Abb. 1.2 Zustandsdiagramm zum Modellieren des Verhaltens von Schnittstellen mit UML

1.1.3 Erstelllen von Sequenzdiagrammen mit Eclipse-Papyrus

Sequenzdiagramme stellen die Interaktionen zwischen verschiedenen Objekten in Bezug auf Ereignisse dar. Diese Diagramme beschreiben die dynamische Modellierung des Systems. Sequenzdiagramme werden zum Beschreiben der Funktionalitäten der Lebenslinien der Objekte und deren Interaktion mit anderen Objekten verwendet. Abb. 1.3 zeigt ein Sequenzdiagramm zum Beschreiben der dynamischen Schnittstellenverhalten in Form einer Nachrichtensequenz. Gemäß Abb. 1.3 gibt einen Überblick über sowohl synchrone Nachricht, dargestellt durch eine durchgezogene Linie mit einem ausgefüllten Spitz, als auch Anwortnachricht, dargestellt durch eine gestrichelte Linie mit einem gestrichelten Spitz. Die zweite Nachricht stellt Anwort auf Aufruf dar. Mithilfe des Sequenzdiagrammes wird den Ablauf der Kommunikation über die Schnittstelle spezifiziert.

1.1.4 Erstellen von Anwendungsfalldiagrammen mit Eclipse-Papyrus

Anwendungsfalldiagramme beschreiben die Aktionen der Akteure zur Steuerung der Anwendungsfälle. Es geht darum, sowohl die Beziehungen als auch die Interaktionen zwischen den Akteuren und den Anwendungsfällen grafisch darzustellen [4]. Akteure und Anwendungsfälle sind in Verbindung, weil jemand etwas tun würde. Abb. 1.4 zeigt ein Anwendungsfalldiagramm zum Beschreiben der Temperatureregelung mit

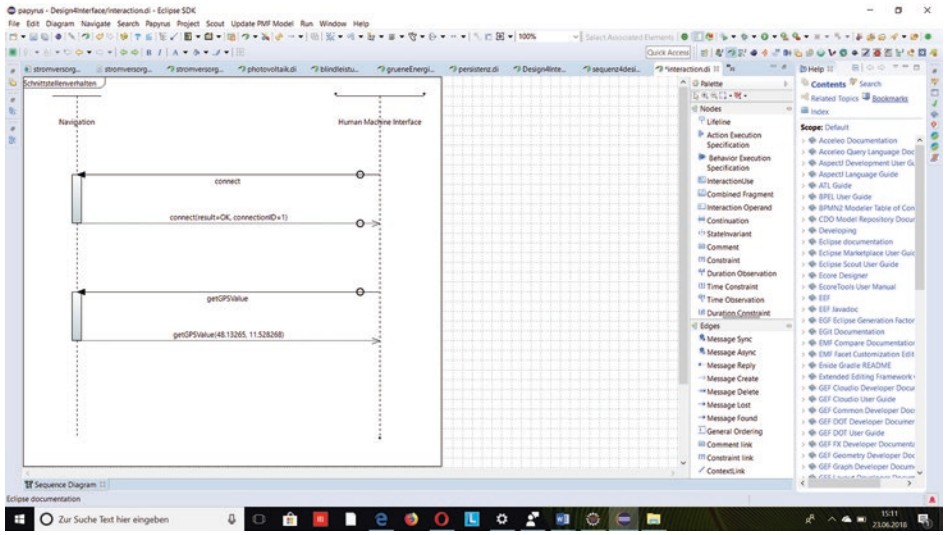


Abb. 1.3 Sequenzdiagramme zum Beschreiben der Funktionalitäten der Lebenslinien der Objekte und deren Interaktion mit anderen Objekten

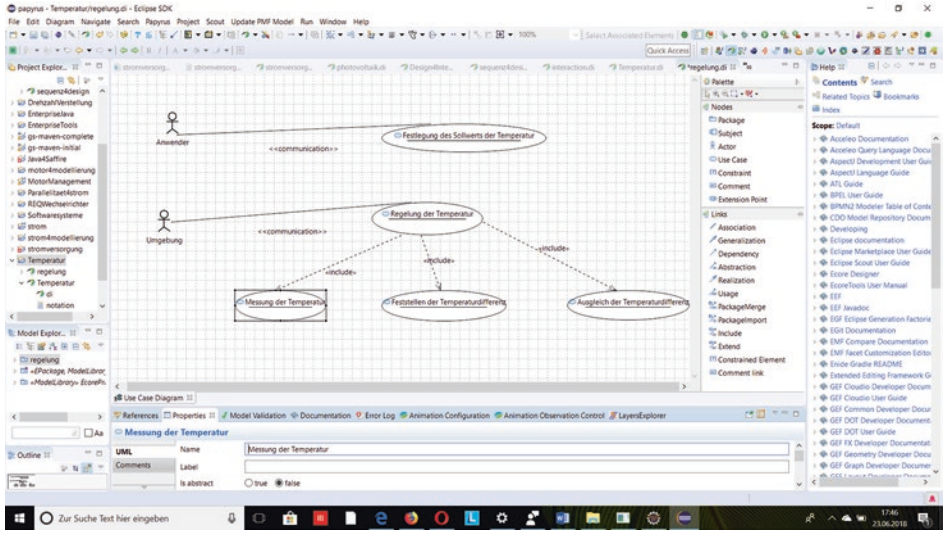


Abb. 1.4 grafische Darstellung der Beziehungen als auch die Interaktionen zwischen den Akteuren und den Anwendungsfällen

dem Open Source Eclipse-Papyrus. Gemäß Abb. 1.4 verfügt das Anwendungsfalldiagramm über zum einen Akteure wie z. B. Anwender oder Umgebung und zum anderen Anwendsfälle wie z. B. *„Regelung der Temperatur“* oder *„Festlegung des Sollwerts der Temperatur“*. Es gibt eine Kommunikation zwischen Anwendern und Anwendungsfällen.

Die Anaylse der Beschreibung des Anwendungsfalldiagramm der Abb. 1.4 ermöglicht zum einen die Festlegung der Solltemperatur mithilfe des Akteurs *„Anwender“* und zum anderen die Regelung der Temperatur mithilfe des Akteurs *„Umwelt“*. Hierbei ändert sich die Außentemperatur und das Programm versucht die Temperatur durch Regelung auszugleichen. Die Beziehungen „communication“ und *„include“* beschreiben eine Assoziation zwischen Akteuren und Anwendungsfällen bzw. einen Aufruf eines Anwendungsfalles.

1.2 Obeo-UML-Designer

UML-Designer ist ein Open-Source-Tool der Firma Obeo zum Erstellen und Visualisieren der Diagramme auf Basis von UML 2.5. Das Framework wurde auf Basis von der grafischen domänenspezifischen Sprache (DSL) Obeo Sirius entwickelt. Das Framework UML-Designer ermöglicht den Entwicklern die Kombination von UML- und DSL-Modellierung. Der Fokus zum Verbessern der User Experience lag sowohl für Endnutzer auf Sirius basierenden Tools wie z. B. UML-Designer als auch für deren Entwickler.

Für Sirius 5 wurden viele Issues unter anderem *„Decorator“*-Management und High-Resolution-Export gefixt, und die Nutzerfreundlichkeit wurde verbessert. Ziel war es, für die Endnutzer einen besseren Workflow zu erschaffen. Das Resultat dieser Entwicklung ist ein neuer Editor, der es Nutzern erlaubt, alle Konzepte ihrer Modeling-Projekte an einer Stelle zu bearbeiten. Das schließt das Bearbeiten der semantischen Modelle, der nutzbaren *„Viewpoints“* und aller Darstellungen (Diagramme, Bäume, Tabellen und Matrizen) ein.

UML-Designer verfügt über zehn Diagramme: Package Hierachy, Class Diagram, Component Diagram, Composite Structure Diagram, Deployment Diagram, Use Case Diagram, Activity Diagram, State Machine, Sequence Diagram, Profile Diagram.

Mithilfe dieser Diagramme können die Entwickler die Transformation von UML zu DSL ermöglichen. Eclipse Sirius Version 5.0 wurde mithilfe von Eclipse-Oxygen veröffentlicht und enthält viele neue Features.

1.2.1 Visualisieren der Diagramme auf Basis von UML 2.5

UML ermöglicht das Visualisieren der Struktur eines Diagramms bezüglich der grafischen Erklärung. Hierbei wird die Semantik der Diagramme mithilfe der Modellierungstools erklärt. Die Unified Modeling Language abgekürzt UML stellt eine visuelle Modellierungssprache dar, die sich über die Bereiche Architektur, Design und die Abläufe von Herstellungsprozessen erstreckt. Das Konzept von UML fokussiert auf eine grafische Standard-Notation bezüglich des Aufzeichnungssystems mithilfe von Zeichen und Symbolen. UML wird als Modellierungssprache zur Beschreibung von Softwaresystemen. Wobei die grafische Standard-Notation mithilfe von Zeichen und Symbolen dargestellt ist. Die Modellierungstools von UML ermöglicht einen Einsatz in

den großen Anwendungsbereichen. Dank dieser Modellierungstools gilt UML momentan als eine der wichtigen Modellierungssprachen für die Softwareentwicklung. Wobei die Sprache für die Spezifikation, Konstruktion und Visualisierung von erklärenden Modellen eingesetzt ist. Sowohl für strukturelle als auch verhaltensbasierte Modellierung wird UML als visuelle Modellierungssprache für Architektur und Design eingesetzt. Tab. 1.1 zeigt die Modellierungstools zum Visualisieren der Diagramme mithilfe von Open Source Eclipse UML-Designer.

Gemäß Tab. 1.1 sind zwei wichtige Fragen zum Modellieren der Anwendungen essenziell: Was soll die Anwendung sein? Was soll die Anwendung tun? Mithilfe dieser Fragen werden sowohl die strukturelle als auch die verhaltensbasierte Modellierung die Antwort zu diesen beiden Fragen. UML-Designer verfügt einerseits über Modellierungstools wie z. B. *„activity tools“* und andererseits über Diagrammtyps zum Modellieren der Anwendungen in Bezug auf Objektorientierung. Abb. 1.5 zeigt die Anwendung zum Realisieren der Effizienz des Asynchronmotors. Hierbei ermöglicht die parallele Modellierung mithilfe des Aktivitätsdiagrammes vom Open Source Ecipse UML-Designer die Erstellungen sowohl der Interfaces der Kenngrößen wie z. B. Drehzahl oder Wirkungsgrad als auch der Klassen der Kenngrößen Drehmoment oder Schlupf. Gemäß Abb. 1.5 stellt die Anwendung die Berechnungen der Kenngrößen des Asynchronmotors mithilfe der Interfaces und Klassen dar. Diese Anwendung soll die Analyse der Effizienz des Motors mithilfe der Kenngrößen ermöglichen. Das Aktivitätsdiagramm, wie es auf der Abb. 1.5 zu sehen ist, zeigt das Verhalten des Asynchronmotors. Hierbei ist das Verhalten des Asynchronmotors bezüglich der Berechnungen der Kenngrößen dynamisch. Abb. 1.5 zeigt die Visualisierung der Analyse der Effizienz des Motors anhand des Aktivitätsdiagramms. Das Aktivitätsdiagramm zeigt den ablaufbasierten Kontrollfluss, genannt Prozesse zwischen Klassenobjekten zusammen mit geplanten Prozessen wie Tools-Workflows. Mithilfe der Modellierungstools

Tab. 1.1 Tools zur Visualisierung der UML-Diagramme mit Obeo-UML-Designer

Anwendungs-modellierung	Was soll die Anwendung sein?	Was soll die Anwendung tun?	Beschreibung des Systems	Erläuterung der internen Struktur des Systems	Definieren der System-umgebung
Diagrammtyp	*„Activity diagram“*	*„Use case diagram“*	*„Component diagram“*	*„Composite structure diagram“*	*„Deployment diagram“*
Modellierungstools	*„Activity tools“*	*„Use Case“*	Node/Edge	Node/Edge	Types/ Relationships
Elemente	Node, Opaque Action, Control Flow …	Actor, Use Case, Subject	Component Class …/ Dependency, Connector …	Component, Class, Part/ Connector, General-ization	Node, Device …/Dependency, Deployment …

Modellierungstool von Open Source Eclipse UML-Designer

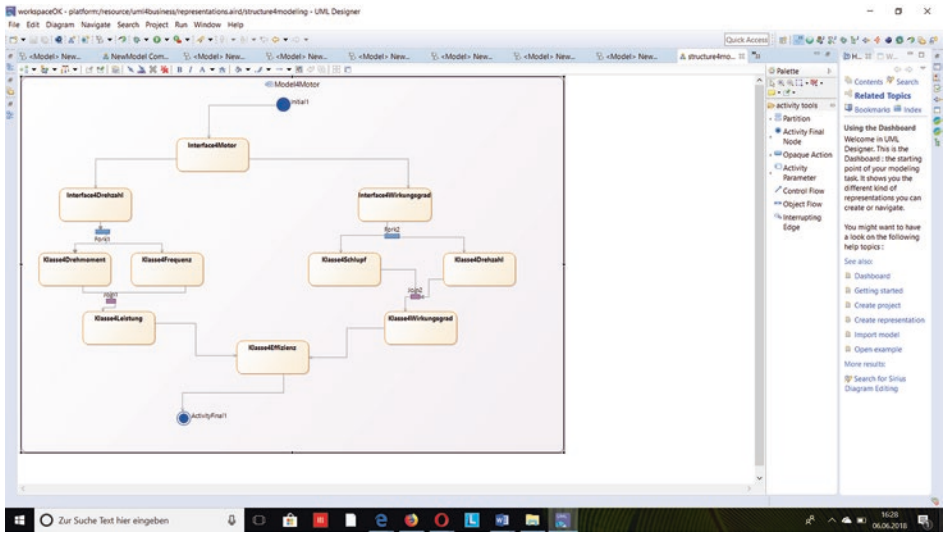


Abb. 1.5 Visualisierung der Analyse der Effizienz des Motors anhand des Aktivitätsdiagrammes

wie z. B. *„Opaque Action“* oder *„Control Flows“* werden Diagramme erstellt. Die Anwendung gemäß Abb. 1.5 ermöglicht die Darstellungen der Modellierung sowohl auf der vertikalen als auch auf der horizontalen Schicht. Die vertikale Schicht, wie es auf beiden Beispielen Example 1–2

zu sehen ist, zeigt die Orientierung der Aktionen vom oben nach unten. Es gibt zwei Säulen oder vertikale Schichten 1 und 2. In jeder Säule gibt es sowohl Interface als auch Klassen.

1. Vertikale Schicht:
   Interface4Drehzahl
   Klasse4Drehmoment
   Klasse4Frequenz + Klasse4Leistung

2. Vertikale Schicht
   Interface4Wirkungsgrad
   Klasse4Schlupf + Klasse4Drehzahl
   Klasse4Wirkungsgrad

Gemäß Example 1–2 zeigt die Anwendung eine Parallelisierungsstruktur bezüglich der Position der vertikalen Schichten 1 und 2. Hierbei sind die vertikalen Schichten 1 und 2 zueinander parallel. Die Anwendung zeigt mithilfe der parallelen Modellierung zwei parallel vertikale Schichten.

Das Modellieren der Anwendung zur Effizienzanalyse des Asynchronmotors beginnt mit einem ausgefüllten Kreis, genannt *„Initial1“* zum Zeichen des Aktivitätsdiagramms.

Hierbei wird *„Initial1“* mit der ersten Aktion genannt *„Interface4Motor“* verbunden, welche durch ein Rechteck mit abgerundeten Ecken dargestellt wird, wie es auf der Abb. 1.5 zu sehen ist.

Das Verbinden jeder Aktion mit anderen Aktionen ermöglicht das Modellieren des Verhaltens der Asynchronotors mithilfe der Modellierungstools wie z. B. Linien zum Darstellen schrittweise den Fluss des gesamten Prozesses.

Anwendungsfalldiagramme ermöglicht den Entwicklern das Suchen einer Lösung zum Ermitteln, was eine Anwendung tun soll. Hierbei werden Entwickler mithilfe des Anwendungsfalldiagrammes eine Antwort auf die Frage *„Was soll die Anwendung zur Analyse der Motoreffizienz tun?“* finden. Zur Analyse des Verhaltens des Systems wird das Anwendungsfalldiagramm, genannt *„Use Case Diagram“*, verwendet, welches eine Beziehung zwischen Benutzern, genannt Akteur, und Anwendungen des Systemes darstellt. Abb. 1.6 zeigt die Funktionalität eines Anwendungsfalldiagramms mithilfe von Eclipse UML-Designer bezüglich der Analyse der Effizienz eines Asynchronmotors. Gemäß Abb. 1.6 verfügt das Diagramm über sechs Anwendungsfälle, an denen sowohl Computer als auch Entwickler interessiert sind. Abb. 1.6 zeigt, dass die Beziehungen zwischen den Anwendungsfällen untereinanderstehen, wie z. B. zwischen dem Anwendungsfall *„Klasse4Hauptprogramm“* und den Anwendungsfällen *„Klasse4Effizienz implementieren“* und *„Klasse4Verluste implementieren“*. Gemäß Abb. 1.6 ist der Akteur *„Computer“* in Beziehung mit dem Akteur *„Entwickler“* in Beziehung zur Modellierung der Motoreffizienz, wobei der Computer von Software wie z. B. Eclipse UML Designer oder Eclipse Papyrus dargestellt ist. Der Entwickler benutzt den Computer zum Erstellen des UML-Anwendungsfalldiagramms mit dem

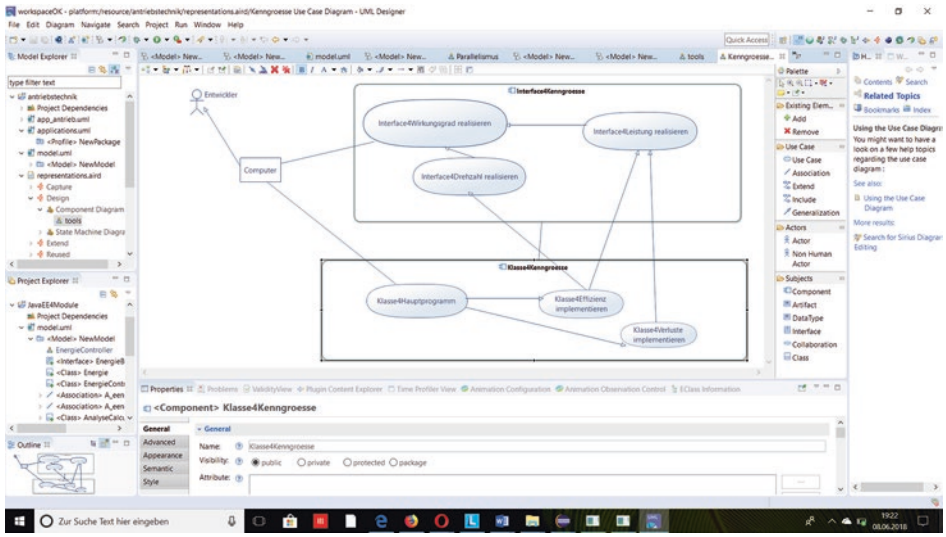


Abb. 1.6 Analyse der Effizienz eines Asynchronmotors mit dem Anwendungsfalldiagramm

Ziel, die Motoreffizienz zu analysieren. Gemäß Abb. 1.6 zeigt das Diagramm zwei Arte der Beziehungen: Generalisierung und Assoziationen. Beispielerweise zeigt die Abb. 1.6 Generalisierung zwischen den Anwendungsfällen *„Interface4Leistung realisieren“* und *„Interface4Wirkungsgrad realisieren“*. Außerdem gibt es eine Assoziation zwischen den Komponenten *„Interface4Kenngroesse“* und *„Klasse4Kenngroesse“*. Der Akteur *„Computer“* ist mit den vorherigen Komponenten mithilfe der Anwendungsfälle *„Interface4Wirkungsgrad realisieren“* und *„Klasse4Hauptprogramm“* assoziiert. Die Anwendungsfallspezifikationen stellen Informationen zu den Anwendungsfällen in Textform dar, wie es im Container *„Important“* zu sehen ist. Eine Beispieldarstellung von Anwendungsfallspezifikationen zeigt der Container *„Important“* [5, 6]. Ein Anwendungsfall eines Systems stellt eine Folge von Aktionen dar, welche ein System ausführt und die ein erkennbares Ergebnis von Wert für einen bestimmten Akteur liefert, während ein Anwendungsfalldiagramm mehrere Anwendungsfälle sowie die Beziehungen zwischen den Anwendungsfällen und den Personen, Gruppen oder Systemen enthalten, die miteinander interagieren, um den Anwendungsfall ausführen zu können [7].

▶ **Wichtig**

Ziel ist es, möglichst einfach zu zeigen, was man mit dem zu bauenden Softwaresystem machen will, welche Fälle der Anwendung es also gibt.

Akteure werden als „Strichmännchen“ dargestellt, welche sowohl Personen wie Kunden oder Administratoren als auch ein System darstellen können (bei Systemen wird manchmal auch ein Bandsymbol verwendet).

Anwendungsfälle werden in Ellipsen dargestellt. Sie müssen beschrieben werden (z. B. in einem Kommentar oder einer eigenen Datei).

Assoziationen zwischen Akteuren und Anwendungsfällen müssen durch Linien gekennzeichnet werden.

Systemgrenzen werden durch Rechtecke gekennzeichnet.

include-Beziehungen werden mittels (mit *<<include>>* gekennzeichneter) gestrichelter Linie und einem Pfeil zum inkludierten Anwendungsfall gekennzeichnet, wobei dieser für den aufrufenden Anwendungsfall notwendig ist.

extend-Beziehungen werden mittels (mit *<<extend>>* gekennzeichneter) gestrichelter Linie und einem Pfeil vom erweiternden Anwendungsfall gekennzeichnet, wobei dieser von dem aufrufenden Anwendungsfall aktiviert werden *„kann“*, aber nicht muss.

Tab. 1.2 zeigt Informationen über Darstellungsspezifikationen von Anwendungsfalldiagrammen bezüglich des Verhaltens des Systems. Hierbei können diese Anwendungsfallspezifikationen nach Bedarf mithilfe eines Texteditors dokumentiert werden. Tab. 1.2 beschreibt eine Darstellung einer Anwendungsspezifikation [7].

Tab. 1.2 Beispieldarstellung einer Anwendungsfallspezifikation

Spezifikationstyp	Beschreibung
Anwendungsfallname	Gibt den Namen des Anwendungsfalls an. Normalerweise drückt der Name die Zielsetzung oder das erkennbare Ergebnis des Anwendungsfalls aus, beispielsweise für einen Bankautomaten *„Bargeld abheben“*.
Kurzbeschreibung	Beschreibt die Rolle und den Zweck des Anwendungsfalls.
Ereignisablauf	Stellt den Basisablauf (Fluss) und die alternativen Abläufe dar. Der Ereignisablauf (Ereignisfluss) beschreibt das Verhalten des Systems. Wie das System funktioniert, die Details der Darstellung oder die Details der Benutzerschnittstelle werden jedoch vom Ereignisablauf nicht beschrieben. Wenn Informationen ausgetauscht werden, muss der Anwendungsfall die ausgetauschten Informationen explizit angeben. Beispielsweise muss anstelle der folgenden Beschreibung *„der Akteur gibt Kundendaten ein“* für eine Aktion die explizite Angabe *„der Akteur gibt den Kundennamen und die Kundenadresse ein“* erfolgen.
Basisablauf	Beschreibt das ideale, primäre Verhalten des Systems.
Alternative Ereignisabläufe	Beschreibt Ausnahmen oder Abweichungen vom Basisablauf, z. B. die Art und Weise, wie sich das System verhält, wenn der Akteur eine falsche Benutzer-ID eingibt und die Benutzerauthentifizierung scheitert.
Spezielle Anforderungen	Nicht funktionale Anforderungen, die spezifisch für einen Anwendungsfall sind, aber im Text des Ereignisablaufs für den Anwendungsfall nicht angegeben sind. Beispiele für spezielle Anforderungen sind gesetzliche Bestimmungen, Anwendungsstandards, Qualitätsattribute des Systems, einschließlich Benutzerfreundlichkeit, Zuverlässigkeit, Leistung und Servicefreundlichkeit, Betriebssysteme und Umgebungen, Anforderungen bezüglich der Kompatibilität und Designeinschränkungen.
Vorbedingungen	Ein Status des Systems, der gegeben sein muss, bevor ein Anwendungsfall gestartet wird.
Nachträgliche Bedingungen	Eine Liste möglicher Zustände des Systems nach Beendigung des Anwendungsfalls.
Erweiterungspunkte	Ein Punkt im Ereignisablauf des Anwendungsfalls, an dem ein anderer Anwendungsfall referenziert wird.

Darstellung einer Anwendungsspezifikation

1.2.2 Überblicke über UML-Diagrammme mit Eclipse-UML-Designer

Die OMG definiert den Zweck der UML wie folgt [8]:

Definition

I. Systemarchitekten und Softwareentwickler erhalten ein Tool für die Analyse, das Design und die Implementierung von softwarebasierten Systemen sowie für die Modellierung von Geschäfts- und ähnlichen Prozessen.

II. Die Branchensituation soll optimiert werden, indem dafür gesorgt wird, dass Tools zur visuellen Modellierung von Objekten miteinander kompatibel sind. Allerdings ist es für einen sinnvollen Austausch von Modelldaten zwischen Tools notwendig, dass eine einheitliche Notation und Semantik verwendet wird.

UML erfüllt die folgenden Anforderungen [8]:

▶ **Wichtig**

- Festlegung einer formalen Definition für ein gemeinsames Metamodell, das auf Meta-Object Facility (MOF) basiert und die abstrakte Syntax der UML spezifiziert. Die abstrakte Syntax definiert die UML-Modellierungskonzepte, ihre Attribute und ihre Beziehungen sowie die Regeln für die Kombination dieser Konzepte zur Entwicklung partieller oder kompletter UML-Modelle.
- Eine detaillierte Erklärung der Semantik für jedes einzelne UML-Modellierungskonzept. Die Semantiken definieren – in einer von der Technologie unabhängigen Art und Weise –, wie die UML-Konzepte von Computern realisiert werden.
- Bestimmung der von Menschen lesbaren Notationselemente für die Darstellung individueller UML-Modellierungskonzepte sowie die Festlegung von Regeln für die Kombination solcher Konzepte, um eine Vielzahl von Diagrammarten für verschiedene Aspekte des modellierten Systems erstellen zu können.
- Festlegung von Methoden, mit denen gewährleistet werden kann, dass UML-Tools mit dieser Spezifizierung übereinstimmen. Dies wird (in einer separaten Spezifizierung) durch eine XML-basierte Spezifizierung von zugehörigen Modell-Austauschformaten (XMI) unterstützt – konforme Tools müssen diesen Prozess dann durchlaufen.

Eclipse-UML-Designer ist ein Open-Source-Projekt der Firma Obeo, welches auf Sirius 5 basiert. Eclipse Sirius stellt ein Werkzeug zum Erstellen der domänenspezifischen Modellierungswerkzeuge dar. UML-Designer stellt ein Tool sowohl zum Erstellen als auch zum Visualisieren von UML-2.5-Modellen. Eclipse-UML-Designer verfügt über zehn generische UML-Diagramme: Package Hierarchy, Class Diagram, Component Diagram, Composite Structure Diagram, Deployment Diagram, Use Case Diagram,

Activity Diagram, State Machine, Sequence Diagram und Profile Diagram. UML nutzt Modellierungstools zum Erstellen verschiedener Arten von Diagrammen: einerseits statische Diagramme zum Darstellen struktureller Aspekte eines Systems und andererseits verhaltensbasierte Diagramme zum Erfassen dynamischer Aspekte eines Systems. Der Container zur Background Information stellt zusätzliche Information über die Definition von UML.

Hintergrundinformation
Ziel der Entwicklung der Unified Modeling Language (UML) ist es, sowohl Semantik als auch Syntax visueller Modellierungssprachen in den Bereichen Architektur, Design und Implementierung komplexer Softwaresysteme bezüglich struktureller als auch verhaltensbasierter Modelle zu realisieren.

Die UML besteht aus verschiedenen Diagrammarten. UML-Diagramme stellen die Grenzen, die Struktur und das Verhalten von Systemen und deren Objekten dar.

UML verfügt über Tools, die UML-Diagramme anwenden, zum Generieren der Code in verschiedenen Programmierungssprachen u. a. Java, C++ oder C#. UML bezieht sich sowohl auf objektorientierter Analyseals auch auf Design.

1.2.2.1 Strukturelle Diagramme

Strukturelle Diagramme beschreiben die Modellierung zeitinvarianter oder inveränderlicher Komponenten von Systemen, wobei diese Komponenten statisch sind [8].

1. **Klassendiagramm:** Das ist das am häufigsten verwendete UML-Diagramm und die wichtigste Grundlage für jede objektorientierte Lösung. Klassen in einem System, Attribute und Vorgänge sowie die Beziehung zwischen den einzelnen Klassen. Klassen werden gruppiert, um Klassendiagramme zu erstellen, wenn große Systeme als Diagramm dargestellt werden sollen.
2. **Komponentendiagramm:** Stellt die strukturelle Beziehung von Softwaresystemelementen dar. Wird am häufigsten für komplexe Systeme mit mehreren Komponenten eingesetzt. Komponenten kommunizieren über Schnittstellen.
3. **Kompositionsstrukturdiagramme:** Kompositionsstrukturdiagramme werden verwendet, um die interne Struktur einer Klasse darzustellen.
4. **Implementierungsdiagramme:** Illustriert die Systemhardware und die zugehörige Software. Nützlich, wenn eine Softwarelösung auf mehreren Maschinen mit individuellen Konfigurationen implementiert wird.
5. **Paketdiagramme:** Es gibt zwei spezielle Arten von Abhängigkeiten, die zwischen Paketen definiert werden: Paketimporte und Paketverschmelzungen. Pakete können die unterschiedlichen Ebenen eines Systems darstellen, um die Architektur zu visualisieren. Paketabhängigkeiten können so dargestellt werden, dass die Kommunikationsmechanismen zwischen verschiedenen Schichten erkennbar sind.

1.2.2.2 Verhaltenbasierte Diagramme

Verhaltenbasierte Diagramme beschreiben die Modellierung dynamischer Komponenten von Systemen [8].

1. **Aktivitätsdiagramm:** Grafisch dargestellte Geschäfts- oder Betriebsabläufe, um die Aktivität eines Teils oder einer Komponente in einem System zu visualisieren. Aktivitätsdiagramme werden alternativ zu Zustandsdiagrammen verwendet.
2. **Sequenzdiagramm** zeigt, wie und in welcher Reihenfolge Objekte miteinander interagieren. Solche Diagramme repräsentieren Interaktionen für ein bestimmtes Szenario.
3. **Zustandsdiagramm:** Ähnlich wie Aktivitätsdiagramme beschreibt diese Art von Diagramm das Verhalten von Objekten, die in ihrem aktuellen Zustand unterschiedliche Verhaltensweisen an den Tag legen.
4. **Anwendungsfalldiagramm** stellt eine bestimmte Funktionalität eines Systems dar und wurde entwickelt, um zu illustrieren, wie Funktionen zueinander in Beziehung stehen und welche internen/externen Akteure es gibt.
5. **Profildiagramm** erweitert UML-Standard zum Erstellen neuer Konzepte. Hierbei wirkt das Profildiagramm in dem Metamodellbereich zum Erstellen neue Stereotype und Profile.

1.2.3 Beispiele von UML-Diagrammen

1.2.3.1 Klassendiagramme für Stromversorgungstester

Klassendiagramme sind der wichtige Teil von UML. Sie erfüllen die wichtigsten Ziele der UML, weil sie Designelemente von der Kodierung des Systems trennen. Ziel der Anwendung der UML in der Softwareentwicklung ist es, ein normiertes Modell zur Beschreibung eines objektorientierten Kodierungsansatzes zu entwickeln. Weil Klassen die Struktur von Objekten darstellen, können Klassendiagramm als das Herzstück der UML verstanden werden. Die Diagrammkomponenten eines Klassendiagramms entsprechen den zu programmierenden Klassen, Hauptobjekten oder Interaktion zwischen Klasse und Objekt.

Die Klassenstruktur verfüg über ein Rechteck mit drei Schichten. Die oberste Schicht enthält den Namen der Klasse, die mittlere Schicht gibt ihre Attribute an und die unterste Schicht gibt Information über die Methoden oder Abläufe innerhalb der Klasse. In einem Klassendiagramm werden Klassen und Unterklassen gruppiert, um den strukturellen Zusammenhang zwischen den einzelnen Objekten zu verdeutlichen. Abb. 1.7 zeigt die Modellierung eines Prüfprotokolls für die Stromversorgung mithilfe des Klassendiagramms. Hierbei verfügen die Klassen TestA–E über Merkmale wie z. B. Attribute und Operationen zum Durchführen des Tests für die Stromversorgung. Mithilfe der Klasse *„ErgebnisA“* werden die Ergebnisse anderer Klassen geprüft. Die Klasse *„ErgebnisA“* enthält zwei Attribute *„fail“* und *„pass“* sowie die Operation *„ergebnisPruefen“* zum Bestätigen des Ergebnises jedes Testes, wobei die Ergebnisse der Unterklassen von der Oberklasse *„ErgebnisA“* abhängen.

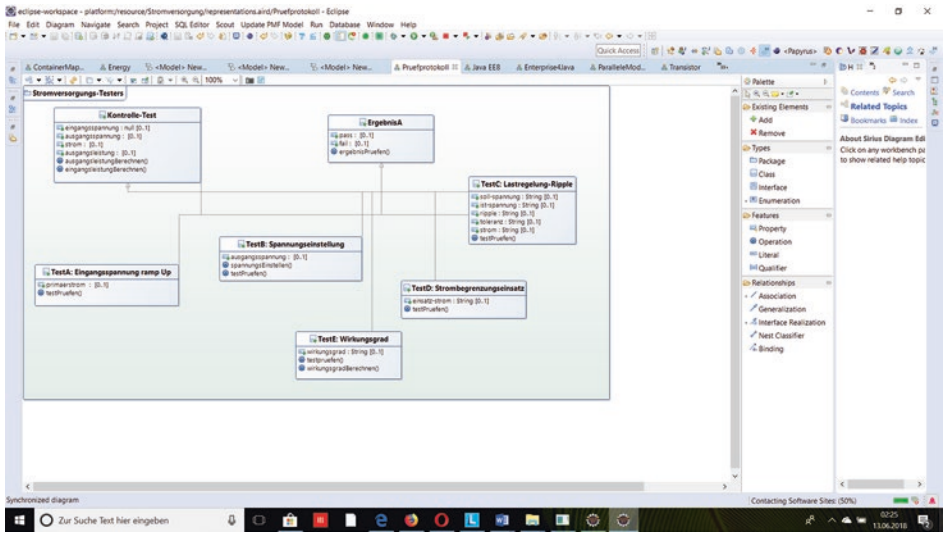


Abb. 1.7 Modellierung eines Prüfprotokolls für die Stromversorgung mithilfe des Klassendiagramms

1.2.3.2 Komponentendiagramme für die Websystemqualität

Komponentendiagramme zeigen sowohl die interne als auch die externe Sicht einer Software oder eines Systems. Hierbei werden verschiedene Komponenten dargestellt. Sie sind als Rechteck dargestellt und tragen in der linken, oberen Ecke das Komponentensymbol. Beziehungen können im Komponentendiagramm als gestrichelte Linien angegeben werden. Komponentendiagramme ermöglichen die Visualisierung der physikalischen Struktur des Systems. Außerdem werden die Komponentendiagramme hilfreich bei sowohl der Analyse der Komponenten des Systems als auch bei deren Beziehungen. Abb. 1.8 stellt die generische Architektur eines Websystems mithilfe des Komponentendiagrammes. Gemäß Abb. 1.8 verfügt ein Websystem über vier Schichten: Frontend, Backend, Integration und Service. Die Komponente Benutzerschnittstelle, genannt *„Frontend“*, befindet sich im Browser. Die Komponente *„Frontend“* ist von der Komponente *„Backend“* geladen und bezieht ihre Daten von dieser. Die Komponente *„Integration“* stellt die Vermittlungsschicht zur Anbindung der Daten- und Geschäftsdienste dar. Die Komponente *„Service“*, entkoppelt von der Komponente *„Integration“*, stellt REST- oder SOAP-Anwendungen für die Websysteme dar. Diese Entkopplung zwischen Komponenten *„Integration“* und *„Service“* ermöglicht eine bessere Websystemqualität bezüglich der Wartbarkeit, Verfügbarkeit, Sicherheit oder Performance [9].

1.2.3.3 Zustandsdiagramme für die Websystemqualität

Zustandsdiagramme, genannt Maschinenzustandsdiagramme, stellen Abbildungen der Übergänge zwischen verschiedenen Objekten dar. Essenzielle Elemente der Zustandsdiagramme sind u. a. Zustände und Übergänge. Zustände werden mit Rechtecken mit abgerundeten Ecken dargestellt, die mit dem Namen des Zustands bezeichnet werden.

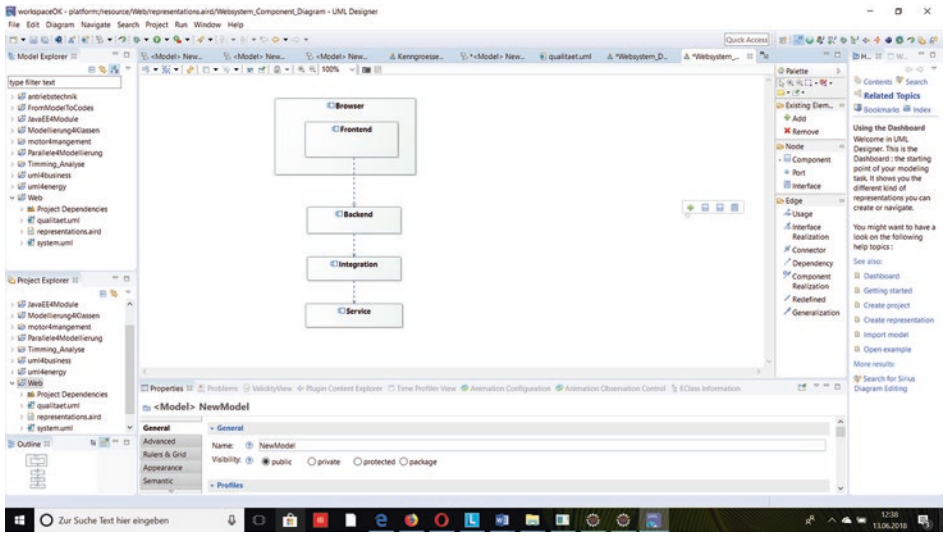


Abb. 1.8 Darstellung der generischen Architektur eines Websystems mihilfe des Komponentendiagramms

Übergänge werden mit Pfeilen, die von einem Zustand zu einem anderen verlaufen, gekennzeichnet und geben die Zustandsveränderung an. Zustandsdiagramme verfügen über verschiedene Elemente wie z. B. Anfangzustand, Endezustand, Ereignis, Zustand, Übergangsverhalten, Austrittpunkt, Übergang und Auslöser.

Zustandsdiagramme finden viele Anwendungen wie z. B. Abbildung ereignisgesteuerter Objekte in einem reaktiven System oder Aufzeigen des Gesamtverhaltens einer Zustandsmaschine oder des Verhaltens mehrerer, miteinander in Beziehung stehender Zustandsmaschinen. Abb. 1.9 zeigt die Veranschaulichung eines Qualitätsszenarios in der Websystemqualität. Hierbei ist zu bemerken, dass die Messbarkeit eines essenziellen Faktors zum Erfüllen der Anforderungen der Websystemqualität angewendet wird. Abb. 1.9 zeigt das Zustandsdiagramm eines Qualitätsszenarios bezüglich des Verhaltens des Websystems. Zwischen Anfang- und Endzustand des Zustandsdiagramms gibt es Übergänge wie z. B. *„Stimulus“* oder *„Antwort“* und Zustände wie z. B. *„Quelle“* oder *„Messkriterium“*. Gemäß Abb. 1.9 stellt die Quelle des Stimulus den Ursprung des Reizes u. a. User oder Administrator dar. Der Übergang *„Stimulus“* beschreibt eine spezifische Kooperation des Zustandes *„Quelle“* mit dem Websystem. Der Zustand *„Artefakt“* liegt in der Umgebung, wobei das Artefakt ein Funktionsblock darstellt. Der Übergang *„Antwort“* wird mithilfe des Zustandes *„Messkriterium“* geprüft.

1.2.3.4 Profildiagramme für Parallelisierungsprozesse für den Asynchronmotor

Profilediagramme ermöglichen die Modellierungen der Metaebene. Außerdem spielen die Profilediagramme eine Nebenrolle zur Modellierung der Softwaresysteme.

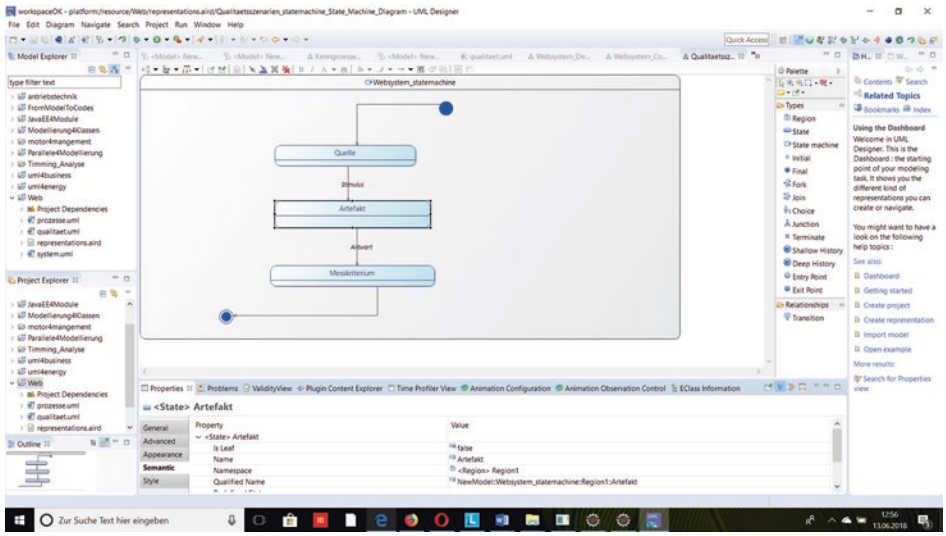


Abb. 1.9 Zustandsdiagramm eines Qualitätsszenarios bezüglich des Verhaltens des Websystems

Profildiagramme werden in der Metamodellebene verwendet, um **Klassen** mit zusätzlichen **Stereotypen** auszuzeichnen, welche die Bezeichnung <<stereotype>> oder bei Profilen erhalten. Profile stellen die Erweiterung des UML-Modells zum Erstellen von benutzerdefinierter Stereotypen, Eigenschaftswerte und Randbedingungen für Plattformen oder Domänen. Mithilfe der Profildiagramme lassen sich die UML-Diagramme an besondere Einsatzgebiete anpassen [10]. Abb. 1.10 zeigt das Profildiagramm eines Parallelisierungsprozesses mit Hinblick auf mathematische Beziehungen zwischen Kenngrößen. Hierbei sind mithilfe des Profilediagrammes vom UML-Designer sowohl horizontale als auch vertikale Schichten zu erkennen.

1.2.3.5 Verteilungsdiagramme oder Deployment-Diagramm

Mit UML werden Verteilungsdiagramme zur Modellierung der physischen Architektur eines Systems beschrieben. Verteilungsdiagramme stellen die Beziehungen zwischen den Software- und Hardwarekomponenten im System und die physische Verteilung der Verarbeitung dar [11]. Abb. 1.11 gibt einen Überblick über die Modellierung der physischen Architektur des Systems Energie für das Web bestehend aus Artefakten, Einheiten oder *„Device“* und Ausführungsumgebung genannt *„Execution Environment“*. Gemäß Abb. 1.11 stellen Device oder Einheiten Knotentype zum Darstellen von physischen Rechenresourcen wie Red Hat Enterprise Linus Advance Server einerseits und Application-Server Tomcat 8 anderseits im System Energie dar. Außerdem zeigt Abb. 1.11 zum einen zwei Artefakte als ausführbare Dateien *„energie.war“* und *„rechnungen.war“* und zum anderen einen Catalina Servlet Container, der eine Ausführungsumgebung darstellt. Diese Ausführungsumgebung liegt in einem Applikationsserver Tomcat 8.

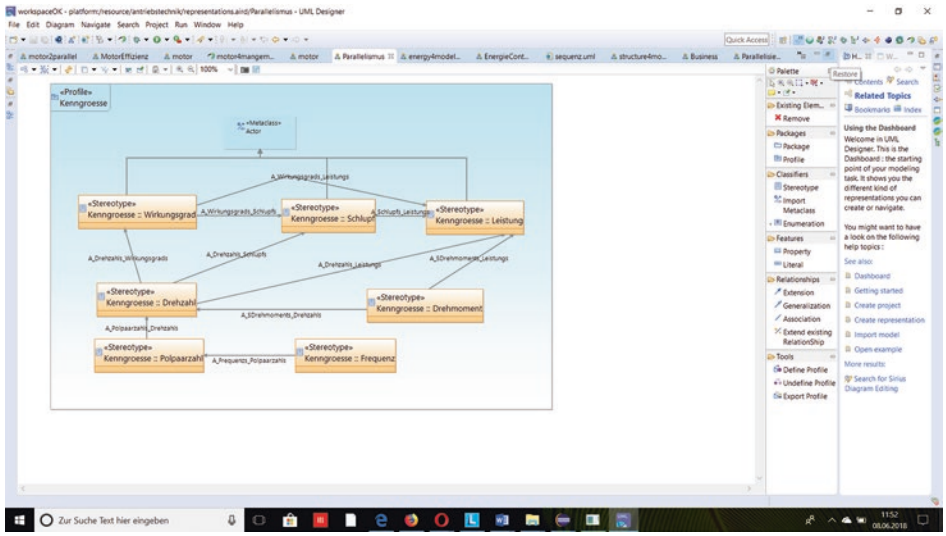


Abb. 1.10 Profildiagramm eines Parallelisierungsprozesses mit Hinblick auf mathematische Beziehungen zwischen Kenngrößen

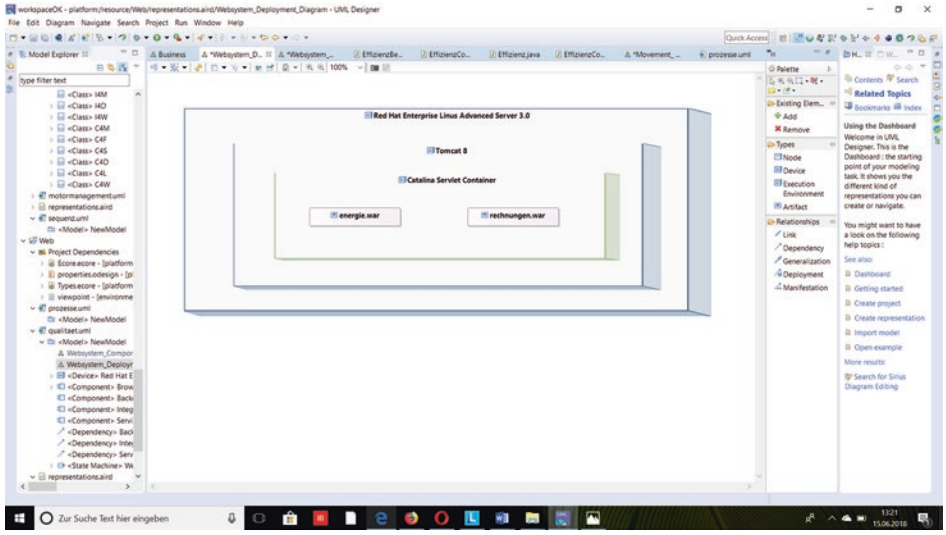


Abb. 1.11 Modellierung der physischen Architektur des Systems Energie für das Web

1.3 UML-SysML-Struktur

Eclipse Framework ermöglicht die Entwicklung von Software mit der Modellierungssprache UML, wobei verschiedene Systemschwerpunkte mithilfe von drei Diagrammarten modelliert werden: Datenstruktur, d. h. statische Strukturen, Systemverhalten, d. h. Dynamik und Systementwurf, d. h. statische Strukturen. Die erste Diagrammart stellt die Klassen- und Objektdiagramme (genannt Class and Object Diagram) dar, die zweite stellt Anwendungsfalldiagramm (Use Case Diagram), Interaktionsdiagramm (Interaction Diagram, genannt Sequence Diagram), Kollaborationsdiagramm (Collaboration Diagram), Zustandsdiagramm (State Diagram) Aktivitätsdiagramm (Activity Diagram) dar und die dritte stellt Komponentendiagramm (Component Diagram) und Einsatzdiagramm (Deployment Diagram) dar.

SysML ist eine Erweiterung der Modellierungssprache UML mit den Anforderungen für Spezifikation, Design und Verifikation von komplexen Systemen, wobei SysML ein Teil von UML dient. Diagramme von SysML stellen Struktur-, Verhaltens-, Anforderungs- und Parametermodellierung dar. Die erste Modellierungsart enthält Blockdefinitionsdiagramm (BDD) und internes Blockdiagramm (IBD), die zweite beinhaltet Diagramme für Aktivitäten (Activities), Interaktionen (Interactions), Zustandsmaschinen (StateMachines) und Anwendungsfälle (UseCases) und die dritte stellt die Zuteilung dar.

1.3.1 Blockdefinitionsdiagramme

Blockdiagramme, auch in Englisch *„block definition diagrams“* genannt, stellen sowohl Funktionsabläufe als auch funktionale Beziehungen zum Beschreiben der Funktionalitäten der Systeme wie z. B. Modulen, Geräte, Systeme, Software und Netzwerke dar. Mithilfe der Linien und Pfeile sind die Interaktionen zwischen Blöcken gekennzeichnet. Sowohl Baumhierarchien als auch Verbindungen zwischen einzelnen Komponenten beschreiben das Systemdesign in Blöcken. Beziehungen zwischen Blöcken wie z. B. Komposition, Assoziation und Spezialisierung beschreiben die Funktionalität der Blockdefinitionsdiagramme. Hierbei definieren die Blockdefinitionsdiagramme die grafische Visualisierung der Informationen. Blockdefinitionsdiagramme schildern die Merkmale und strukturellen beziehungsweise hierarchischen Beziehungen von Blöcken. Ein Block (block) modelliert eine Gruppe respektive Klasse struktureller Elemente eines Systems. Blockdefinitionsdiagramme sind als Rechtecke dargestellt und in eine Folge von Abschnitten wie z. B. *„Part“* oder *„Association“* unterteilt [12].

Der Editor vom Open Source Eclipse-Papyrus zeigt die Palette, genannt Modellierungstools des Blockdefinitionsdiagramms mit sechs Bereichen: *„General Annotation“*, *„General Structure“*, *„Viewpoint“*, *„Blocks“*, *„Ports and Flows“*, und *„Constraints“*, wie es auf der Abb. 1.10 zu sehen ist. Gemäß Abb. 1.12 verfügt das Blockdefinitionsdiagramm, genannt *„smartWechselrichter“*, drei Packages *„Energieuebertragung“*,

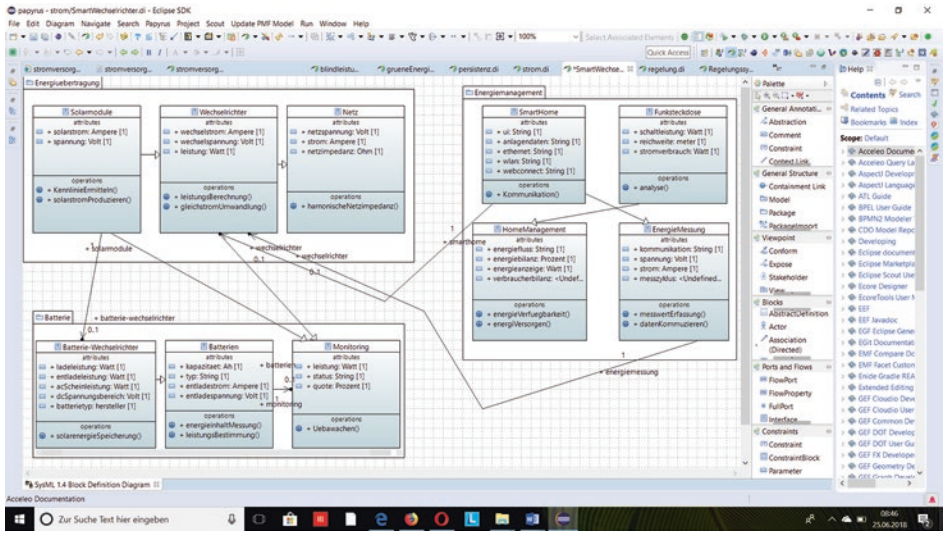


Abb. 1.12 Blocksdefinitionsdiagramm zum Darstellen der Beziehungen sowohl zwischen Blöcken des gleichen Pakets als auch zwischen Blöcken verschiedener Packages

„*Batterie*“ und „*Energiemanagement*“ und deren Blöcke in zwei Eigenschaften unterteilt sind: „*attributes*“ und „*operations*“. Das Blockdefinitionsdiagramm „*smartWechselrichter*“ stellt die Funktionalität der Solarwechselrichter in Bezug auf das Auslegen von Solaranlagen und Energiesystemen dar. Abb. 1.12 zeigt die Beziehungen sowohl zwischen Blöcken des gleichen Pakets als auch zwischen Blöcken verschiedener Packages. Bespielweise zeigt einerseits das Paket „*Energieuebertragung*“ eine lineare Verbindung von dem Block „*Solarmodule*“ über den Block „*Wechselrichter*“ bis zum Block „*Netz*“, wie es auf der Abb. 1.12 zu sehen ist, und andererseits eine direkte Assoziation mit dem Package „Energiemanagement“ in Bezug auf die Nutzung des Wechselrichters. Hierbei sind die Blöcke „*SmartHome*“ und „*EnergieMessung*“ mithilfe der Operationen „*Kommunikation()*“ bzw. „*datenKommunizieren()*“ mit dem Block „*Wecherichter*“ vom Package „*Energieuebertragung*“ assoziert. Mithilfe des Pakets „*Energiemanagement*“ werden sowohl die Analyse des Eigenverbrauchspotenzials oder die wirtschaftlichen Auswertungen als auch die ganzheitliche Simulation von Energiesystemen realisiert. Das Package „*Batterie*“ verfügt über drei Blöcke, die untereinander verbunden sind: „Batterie-Wechselrichter“, „*Batterien*“ und „*Monitoring*“. Der letzte Block stellt mithilfe sowohl einer Beziehung zwischen den Blöcken „*Batterie-Wechselrichter*“ und „*Batterien*“ als auch einer direkt Assoziation zwischen den Blöcken „Monitoring“ und „Batterien“ eine Überwachung der Funktionalität des Batterie-Wechselrichters dar. Die letzten beiden Blöcke haben eine gerichtete Beziehung, genannt Generalisierung, und sie ist durch einen durchgezogenen Pfeil mit geschlossener, nicht ausgefüllter Pfeilspitze gekennzeichnet.

1.3.2 Interne Blockdiagramme

Interne Blockdiagramme stellen eine miniature Ebene eines Blocks bezüglich der Detaillierung der Funktionalität des Systems dar. Hierbei zeigen die internen Blockdiagramme mithilfe von *„Ports“* oder *„Connectors“* verschiedene Verbindungen zwischen den Parts. Ein Einzelteil, genannt Part, stellt ein strukturelles Merkmal eines Blocks dar. Es besitzt einen optionalen Namen, einen durch einen Doppelpunkt eingeleiteten obligatorischen typisierenden Block und eine Multiplizität in eckigen Klammern. Parts werden in einem separaten Abschnitt eines Blocks modelliert [12]. Abb. 1.13 zeigt ein internes Blockdiagramm vom Open Source Eclipse-Papyrus zum Modellieren der Funktionalität eines Wechselrichters bezüglich der Beschreibung verschiedener leistungselektronischen Elemente oder Parts wie z. B. „Gleichstrom“, „Wechselspannung“ oder *„Leistung“*. Gemäß Abb. 1.13 zeigt das interne Blockdiagramm den Block namens *„Wechselrichter“*, dessen Inhalte aus verschiedenen *„Parts“* und deren Konnektoren genannt *„Connectors“* zusammengesetzt ist. Jeder Part verfügt über ein Attribut und dessen Typ. Beispielsweise hat den Part *„Leistung“* das Attribut *„leistung“* und den Typ *„watt“*. Wobei, das Diagramm gemäß Abb. 1.13 zeigt das interne Blockdiagramm die Details des Blocks *„Wechselrichter“*. Die Funktion eines Wechselrichters ist die Umwandlung einerseits des Gleichstromes in den *„Wechselstrom“* und andererseits der *„Gleichspannung“* zum *„Wechselspannung“*, wie es auf der Abb. 1.13 mithilfe der Parts zu sehen ist. Mithilfe der Parts gemäß Abb. 1.13 wird sowohl die Leistung als auch die Verlustleistung berechnet. Die Konnektoren oder *„Connectors“* stellen Verbindungen zwischen Parts dar. Z. B. sind die Parts *„gleichstrom“* und *„wechselstrom“* mithilfe

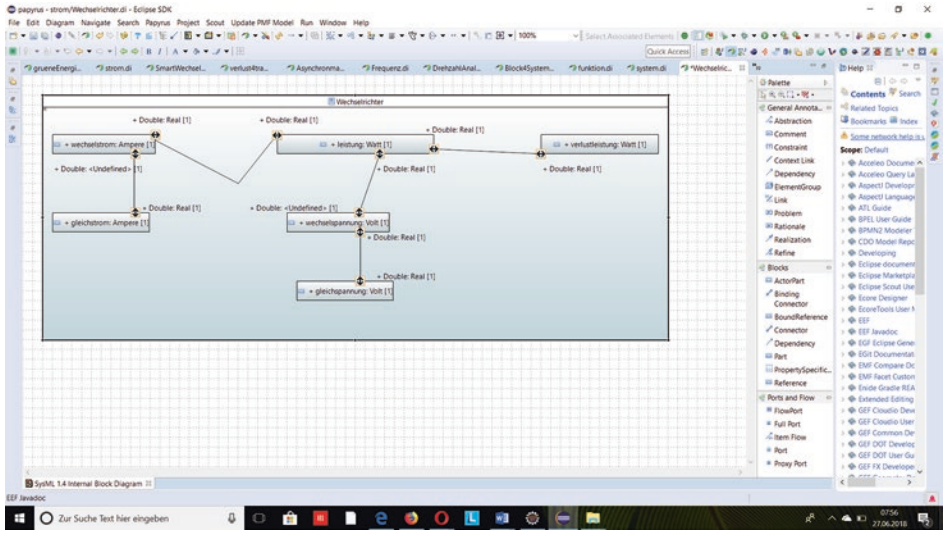


Abb. 1.13 Darstellung des internen Bockdiagramms zum Visualisieren des internen Aufbaus des Blocks „Wechselrichter“

von *„FlowPort“* und *„Connector“* verbunden. Gemäß Abb. 1.13 ermöglicht das interne Blockdiagramm das Visualisieren des internen Aufbaus des Blocks *„Wechselrichter“* mit seinen Parts, Schnittstellen genannt *„Ports“* und Relationen genannt *„Flows“*. Ports stellen Anschlüsse zum Verbinden von Parts dar. Hierbei werden die *„Connectors“* die Verbindungen zwischen Ports ermöglichen. Z. B. verbindet ein Connector die Zugangspunkte an den Grenzen von den Parts „leistung:Watt“ und *„wechselspannung:Volt“*, wie auf der Abb. 1.13 zu sehen ist. Gemäß Abb. 1.13 ist zu sehen, wie die Flussrichtungen der Datenströme in den Ports aufgezeichnet sind.

1.3.3 Anforderungsdiagramme

Ziel der Anwendungen der Anforderungsdiagramme in der Systementwicklung ist es, das Beschreiben der Funktionalitäten der Systeme mithilfe einer Architektur zu ermöglichen. Wobei es sowohl funktionale als auch nichtfunktionale Anforderungen zum Darstellen der Architekturen gibt. Anforderungsdiagramme ermöglichen das Dokumentieren der Anforderungen eines Modells.

Anwendungen der Anforderungsdiagramme stellen das Beschreiben sowohl der Eigenschaften als auch des Verhaltens der Systeme dar. Hierbei stehen Anforderungen mit anderen in Beziehungen mithilfe von Modellierungstools wie z. B. *„DeriveReqt“*, *„Refine“*, *„Satisfy“*, *„Trace“*, *„Verify“*, *„Containment link“*, *„Dependency“*, *„Copy“*. Der Editor von Eclipse-Papyrus für SysML verfügt über vier Bereiche zum Nutzen der Modellierungstools: *„General Annotations“*, *„General Structure“*, *„Viewpoint“*, *„Requirement“*. Verschiedene Modellierungselemente der Systeme werden mithilfe der Beziehungsytype erkannt und damit nachvollziehbar bei den Projektbeteiligten zugänglich gemacht [13]. Abb. 1.14 zeigt ein Anforderungsdiagramm mit sechs Anforderungen und eine Test-Aktivität zum Darstellen der modellbasierten Systementwicklung für sowohl die Regelung als auch die Überwachung der Temperatur im Bereich Mechatronik. Gemäß Abb. 1.14 werden Anforderungen zum Messen der Temperatur einerseits mithilfe der Regelung und andererseits mithilfe des Sensors beschrieben. Anforderungen werden vom dem Regelungssystem mithilfe der Beziehung *„DeriveReqt“* einerseits für das An- und andererseits für das Ausschalten des Lüfters abgeleitet. Die Nachverfolgbarkeit, genannt *„Traceability“*, des Systems gemäß Abb. 1.14 stellt mithilfe der Beziehungen *„Verify“*, *„DeriveReqt“* und *„Satisfy“* das Dokumentieren der Anforderungen dar. Die beziehung *„Verify“* ermöglicht das Erkennen der vordefinierten Testfälle für Anforderungen.

1.3.4 Zusicherungsdiagramme

Zusicherungsdiagramme stellen das Visualisieren der mathematischen Formel mithilfe der Grafik zum Beschreiben von Contraints-Blöcken dar. Wobei der Begriff *„Contraints“*, auch *„Zusicherung“* genannt, sich auf das Definieren der formalen Bedingung oder

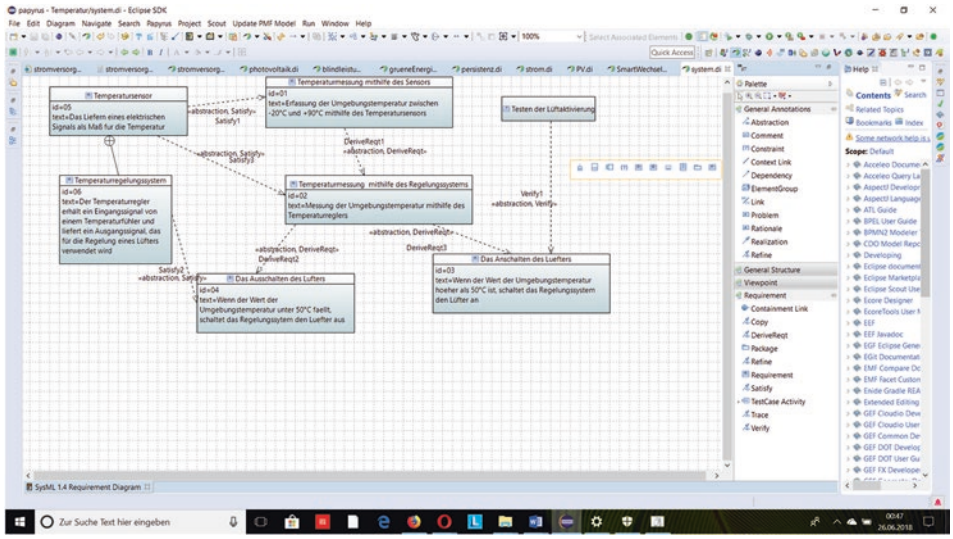


Abb. 1.14 Anforderungsdiagramm zum Messen der Temperatur einerseits mithilfe der Regelung und andererseits mithilfe des Sensors

Gleichung bezieht. Sowohl *„Parts“* als auch *„Contraints-Property“* aus dem Block ermöglichen das Erstellen der Zusicherungsdiagramme zum Integrieren der Analyse der Systeme mithilfe der Designmodellierung. Hierbei spielen die Beziehungen u. a. *„Connectors“* oder *„Dependency“* eine wichtige Rolle beim Erstellen des Zusicherungsdiagramms. Der Editor von Eclipse-Papyrus verfügt über vier Bereiche zum Anwenden der Modellierungselemente: *„General Annotations“*, *„Blocks“*, *„Ports and Flows“* und *„Constraints“*. Abb. 1.15 zeigt ein Zusicherungsdiagramm zum Analysieren der Funktionalitäten eines Schwingkreiswechselrichters mithilfe von *„Parts“*, *„Connectors“* und *„Comment“*. Schwingkreiswechselrichter sind Stromrichter zum Kennzeichen des Lastkreises mithilfe eines Schwingkreises bestehend aus Lastwiderstand und Kondensator. Hierbei wird in der Regel der eigentliche ohm-induktive Lastwiderstand durch Hinzufügen eines Kondensators entweder zu einem Parallelschwingkreis oder zu einem Reihenschwingkreis ergänzt [14]. Das Beschreiben des zusicherungsdiagrammes anhand der Abb. 1.15 fokussiert auf die Anwendungen der parametrischen Formel oder Gleichungen zum Nachvollziehen von Zusammenhängen zwischen verschiedenen Kenngrößen dieses Reihenschwingkreis-Wechselrichters in der Design-Modellierung. Beispielsweise ist der Abb. 1.15 den Resonanzfall bezüglich der Gleicheit zwischen induktivem und kapazitivem Blindwiderstand bei diesem Resonanzwechselrichter zu entnehmen. Gemäß Abb. 1.15 sind die Parts von induktiven und kapazitiven Blindwiderstand mithilfe der Beziehung „Connector“ verbunden. Hierbei stellen Parts bezüglich von Annotation „Comment“ mathematische Formel zum Beschreiben der Funktionsweise dieses Resonanzwechselrichters dar. Z. B. definieren die Parts „verlustleistung“ und *„frequenz“* die Gleichungen zum Bestimmen der Verlustleistung bzw. Frequenz. Beide Parts bzw. Gleichungen sind in Verbindung mithilfe der Beziehung

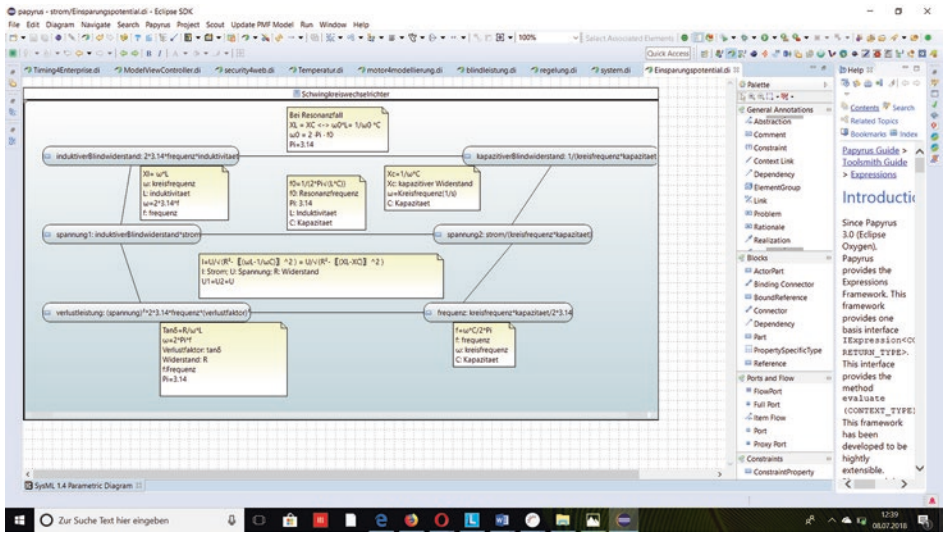


Abb. 1.15 Zusicherungsdiagramm zum Analysieren der Funktionalitäten eines Schwingkreiswechselrichters mithilfe von Open source Eclipse-Papyrus

„*Connectors*“. Gemäß Abb. 1.15 stellt das Design des Zusicherungsdiagrammes eine Modellierung der Parallelität sowohl auf horizontalen als auch vertikalen Ebene dar. Beispielsweise zeigt das Zusicherungsdiagramm zwei horizontale Säulen zum Modellieren der Parallelität. Die erste Säule erfolgt vom Part „*induktiverBlindwiderstand*“ den Part „*spannung1*“ bis zum Part „*verlustleistung*“. Die zweite Säule beginnt mit dem Part „*kapazitiverBindwiderstand*“ über den Part „*spannung2*“ bis zum dem Part „*frequenz*“. Die beiden Säulen stellen mithilfe der Design-Modellierung der „*Parts*“ einerseits ein kapazitives und andererseits ein induktives Verhalten des angeschlossenen Lastkreises einer Betriebsfrequenz bei dem Schwingkreiswechselrichter dar. Außerdem ermöglicht das Zusicherungsdiagramme eine horizontale Modellierung der „*Parts*“. Gemäß Abb. 1.15 verfügt das Zusicherungsdiagramm drei Schichte zum Modellieren der horizontalen Ebene. Beispielsweise ist die erste Schicht das Modellieren vom Part „*induktiverBlindwiderstand*“ bis zum Part „*kapazitiverBlindwiderstand*“. Anschließend modelliert die zweite Schicht den Part „*spannung1*“ bis zum Part „*spannung2*“. Am Schluss modelliert die dritte Schicht den Part „verlustleistung“ bis zum Part „*frequenz*“.

1.4 IT-Lösungen mit „*Modeling4Programming*“

Das Konzept von „*Modeling4Programming*“ stellt die Anwendung der Modellierung in der Programmierung dar. Hierbei werden Anwendungen mithilfe der Modellierung programmiert. Das Buch fokussiert auf mächtige objektorientierte Programmierungssprachen: Java und C++. Diese Sprachen ermöglichen die Realisierungen von IT-Lösungen z. B. für die Leistungselektronik, Java Enterprise Architektur und Design Patterns.

1.4.1 Modellierung von IT-Lösungen mit C++

C++ ist eine starke Programmierungssprache zur Darstellung von Modellierungstools bezüglich der Übersetzung von Code in Model-Elemente. Das Konzept fokussiert auf die Nutzung der Codes zur Modellierung von IT-Lösungen. Jeder Code ist Teil eines Modellierungsdiagramms wie z. B. Klassendiagramm oder Blockdiagramm entsprechend den Sprachen UML bzw. SysML. Ein Code aus C++ kann in einem UML-Klassendiagramm angewendet werden. Ein Klassendiagramm verfügt über Funktionalitäten zur Modellierung von Anwendungen, die mithilfe von objektorientierter Programmierung das Programmieren sowohl erleichtert als auch erklären.

IT-Lösungen zu modellieren ermöglicht das Programmieren der Anwendungen mithilfe sowohl der objektorientierten als auch prozeduralen oder modularen Programmierung. Hierbei spielt der Begriff Funktion eine wichtige Rolle beim Darstellen der funktionalen Programmierung mithilfe von C++. Programmieren der Funktionen in einer Sprache ermöglicht das Darstellen der Funktionalitäten der Anwendungen mithilfe von einer Programmierungssprache wie z. B. C++. Der Einsatz von Funktionen dient der Zerlegung der Aufgaben in mehrere Teilaufgaben. Dies erfordert die Deklarationen von Funktionen. Hierbei werden die wichtigen Daten der Funktionen mitgegeben, wobei der Aufruf der Funktionen mithilfe der Namensnennung erfolgt [15]. Die Syntax eines Funktionsprototyps besteht aus drei Elementen: Rückgabetyp oder Funktionstyp, Funktionsname und Parameterliste. Die Definition von Funktionen erfordert die Angabe der Variablen, deren Verwendung im Funktionskörper benötigt wird. Die Funktionalität der Programmierung mithilfe der Funktionen dient der Strukturierung der Programme.

Objektorientierungsprogrammierung mit C++ ermöglicht die Verwendung von Klassen einerseits bezüglich ihrer Eigenschafte d. h. Attribute und Methoden andererseits bezüglich der Vererbung oder Polymorphie. Mithilfe des Begriffes *„Class“* ist zu verstehen, dass Objekte Variablen oder Instanzen der Klassen darstellen. Das Erstellen der Objekte oder Instanzen stellt die Definition der Eigenschafte dieser Instanzen dar.

1.4.1.1 Anwendungen der Funktionalitäten der Elektronik in der Modellierung

Funktionalitäten der Leistungselektronik sind mit Modellierungen sowohl funktional oder modular als auch objektorientiert verbunden. Systeme im Bereich Leistungselektronik sind komplex und benötigen IT-Lösungen einerseits zum Beschreiben der Zusammenhänge und andererseits zum Ausführen der Programme. Von Model zu Code stellt die Verwendungen der Modellierungssprachen zum Beschreiben der Funktionalitäten der Programme mithilfe von Programmierungssprache wie z. B. C++ dar. Listing 1.1 zeigt den Programmcode, genannt *„Drehzahl“*, aus der Programmierungssprache C++ mit Microsoft Visual studio 2017 zum Ausführen der Funktionalitäten der Charakterisierung des Asynchronmotors bezüglich der Drehzahlsteuerung. Gemäß Listing 1.1 aus Microsoft Visual Studio 2017 soll das Programm *„Drehzahl“* die Kenngrößen Schlupf und Wirkungsgrad des Asynchronmotors mithilfe der Funktionen *„schlupfBerechnen()“*

und *„WirkungsgradBerechnen()“* berechnen. Die Funktion *„anzeige()“* verfügt über Information zum Visualisieren des Charakterisierens der Motors mithilfe von *„std::cout“*, wobei der Bildschirm die Informationen zum einen über die berechnete Kenngrößen und zum anderen über die Hilfe oder Detail zum Berechnen der Kenngrößen ausdruckt. Mithilfe der Funktion *„get_eingabe()“* werden die Werte der Kenngrößen wie z. B. Läuferdrehzahl und synchrone Drehzahl eingegeben, damit die Berechnungen der Schlupf und Wirkungsgrad mithilfe der Funktionen *„schlupfBerechnen()“* bzw. *„wirkungsgradBerechnen()“* erfolgen. Ergebnisse der Berechnungen aus den Funktionen *„schlupfBerechnen()“* bzw. *„wirkungsgradBerechnen()“* werden mithilfe der Funktion *„berichten()“*. Gemäß Abb. 1.16 sind die Definition aller Funktionen des Programms *„Drehzahl“* von der Quelldatei *„wirkungsgrad.cpp“* in der main-Funktion aufgerufen. Einerseits verfügt das Programm „Drehzahl“ über Funktionen mit dem Datentyp *„void“* ohne Rückgabe von Werten wie z. B. *„berichten()“* oder *„schlupfBerichten()“*, und andererseits verfügt es über die Funktion *„anzeige()“* mit der Wertrückgabe des Datengtyps *„char“*. Gemäß Abb. 1.16 stellt die Funktionsdefinition vier Elemente: Namen, Type, Parameter und Body dar. Das letzte Element, auch Hauptteil oder Körper der Funktion genannt, enthält Codes, die während ihres Aufrufes ausgeführt werden. Die Anwendung des Konzeptes *„Modeling4Programming“* ermöglicht das Erstellen eines UML-Diagramms zum Beschreiben die Funktionalität der Charakterisierung des Asynchronmotors aus dem C++-Codes. Hierbei wird das Aktivitätsdiagramm mithilfe des Open Source Frameworks Eclipse-Papyrus erstellt. Das Aktivitätsdiagramm ermöglicht die Darstellung des Verhaltens eines Softwaresystems, wobei die verschiedenen Prozesse, genannt Aktionen, in Reihenfolgen als Steuerungsflüsse ausgeführt werden.

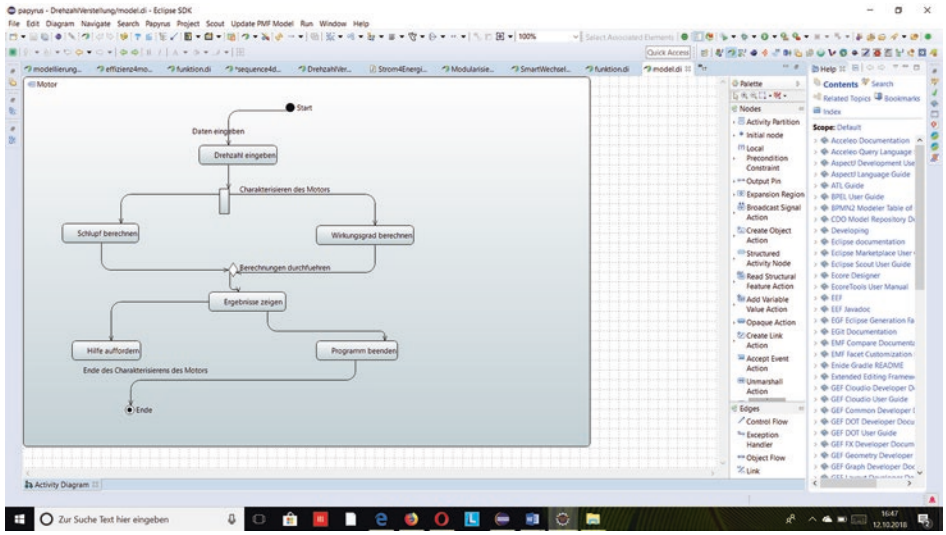


Abb. 1.16 Modellierung der Charakterisierung des Asynchronmotors mit dem Aktivitätsdiagramm vom Eclipse-Papyrus

Mithilfe vom Listing 1.1 wird ein Aktivititätsdiagramm zum Beschreiben des Verhaltens und zum Charakterisieren des Asynchromotors erstellt, wie es auf der Abb. 1.16 zu sehen ist. Gemäß Abb. 1.16 werden die Berechnungen von Schlupf und Wirkungsgrad mithilfe eines Parallelisierungsprozesses durchgeführt. Anschließend werden die Ergebnisse zum Charakterisieren des Motors gezeigt. Dies führt mithilfe eines Parallelisierungsprozesses entweder zum Beenden des Programms oder zur Aufforderung der Hilfe. Abb. 1.17 zeigt das kompilierte Programm hinsichtlich der Berechnungen des Schlupfes und des Wirkungsgrades, wobei zuerst die Werte dieser Kenngrößen angegeben werden. Die main-Funktion, wie im Listing 1.1 zu sehen ist, verfügt über Aufrufe von verschiedenen Funktionen, u. a. *„anzeige()“*, *„berichten()“*, *„hilfe_bildschirm()“*, *„schlupfBerechnen()“* oder *„wirkungsgradBerechnen()“*, welche mithilfe der switch-Auswahlanweisung aufgerufen werden. Hierbei ist zu bemerken, dass alles bis zum nächsten Break ausgeführt wird. In den runden Klammern wird eine Variable als ganze Zahl übergeben wie z. B. char oder int. Anschließend wird der case-Bereich äquivalent zu dieser Variablen angesprungen. Switch-Auswahlanweisung vom Listing 1.1 enthält keinen default-Zweig, wobei diese Abwesenheit in Ordnung ist und die switch-Anweisung sowieso beendet ist. Wichtig ist die Anwesenheit der break-Anweisung zum Ausführen der Zeile für switch-Auswahlanweisungen. Ebenfalls zeigt Listing 1.2 die Struktur der Kundenverwaltung für die Stromlieferung mithilfe von C++. Mithilfe der Switch-Anweisungen in der main-Funktion werden verschiedene Funktionen der Klasse Stromlieferung u. a. *„add_konto()“*, *„ausgabe_konto()“* und *„sort()“* zum Verwalten der Kundenrechnungen für die Stromlieferung aufgerufen. Die Klasse Stromlieferung ermöglicht dem Energieverkäufer das Konto der Kunden sowohl zu sortieren als auch zu zeigen oder zu registrieren. Mit den Vektoren gibt es eine Möglichkeit die Funktionen *„ausgabe_konto()“* und *„sort()“* zu

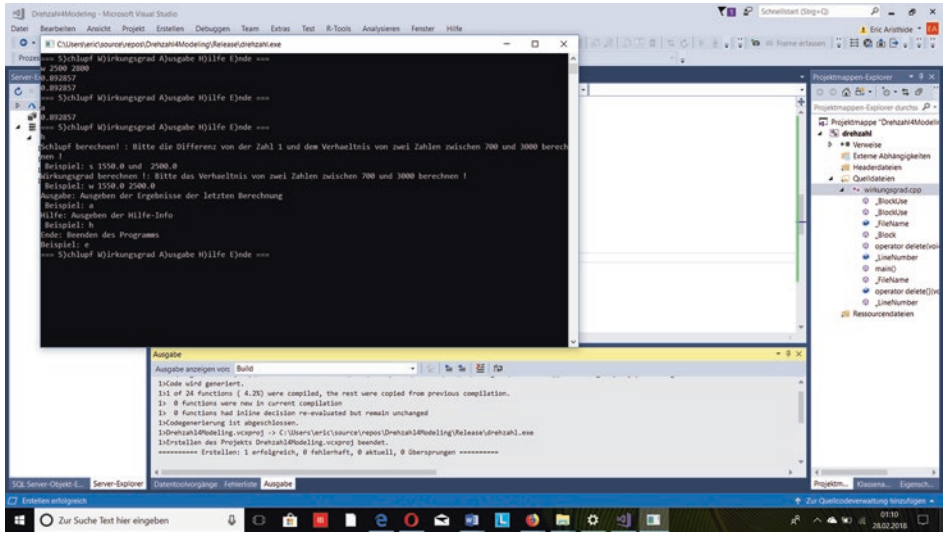


Abb. 1.17 Kompilieren des Programms zum Charakterisieren des Motors

vektorisieren. Dies ermöglicht die Erleichterung bei der Verwaltung der Daten der Kunden. Die Klasse Stromlieferung verfügt über die Attribute *„id“*, *„rechnung“* und *„namen“*, welche als *„public“* deklariert wurde. Abb. 1.18 stellt die Modellierung der Anwendung zur Kundenverwaltung bezüglich der Programmierung aus dem Listing 1.2 mithilfe des Aktivitätsdiagramms dar. Hierbei sind sowohl synchrone als auch parallele Prozesse in Zusammenhang mit dem Stromverbrauch. Das Aktivitätsdiagramm zeigt die Funktionalität der Programmierung, wobei die Eingaben über den Kunden zum Berechnen des Stromverbrauchs die Modellierung der Kundenverwaltung darstellen. Gemäß Abb. 1.18 benötigt dieser Prozess der Berechnung des Stromverbrauchs das Bearbeiten, Anzeigen und Analysieren der Information und verläuft parallel. Die Modellierung liegt in einer vertikalen Schicht, d. h., von dem Start der Modellierung bis zum Beenden des Programms werden verschiedene Informationen in einer vertikalen Richtung verlaufen. Abb. 1.19 ist das kompilierte Programm aus dem Listing 1.2 und gibt einen Überblick über die Berechnung des Stromverbrauchs mithilfe der Information über die Kunden.

Listing 1.1 Funktionalität der Charakterisierung des Asynchonmotors in Bezug auf die Drehzahlsteuerung zur Berechnung von Schlupf und Wirkungsgrad mit C++ (Quelldatei ist *„wirkungsgrad.cpp“*)

```
#include <iostream>
#include <cmath>
void hilfe_bildschirm()
{
std::cout << "Schlupf berechnen! : Bitte die Differenz von der Zahl 1
und "
```

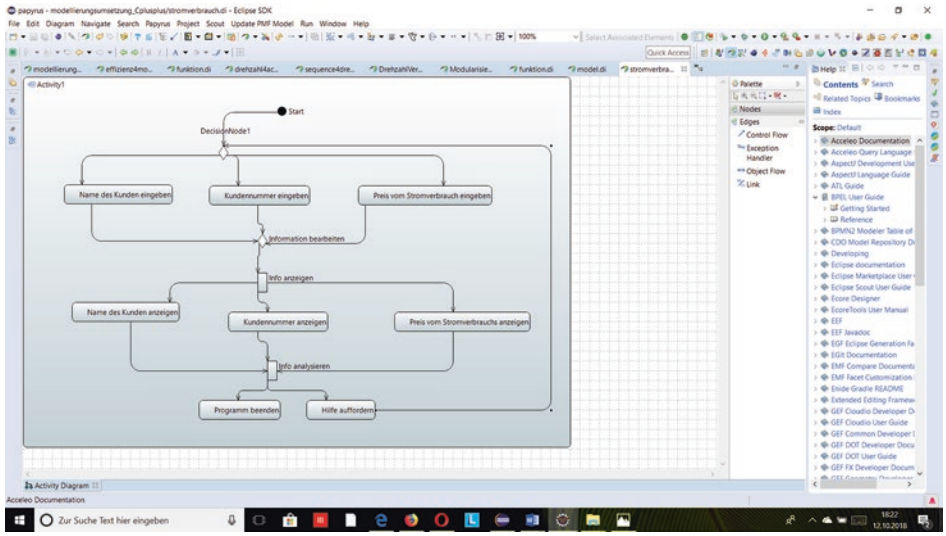


Abb. 1.18 Modellierung der Berechnung des Stromverbrauchs

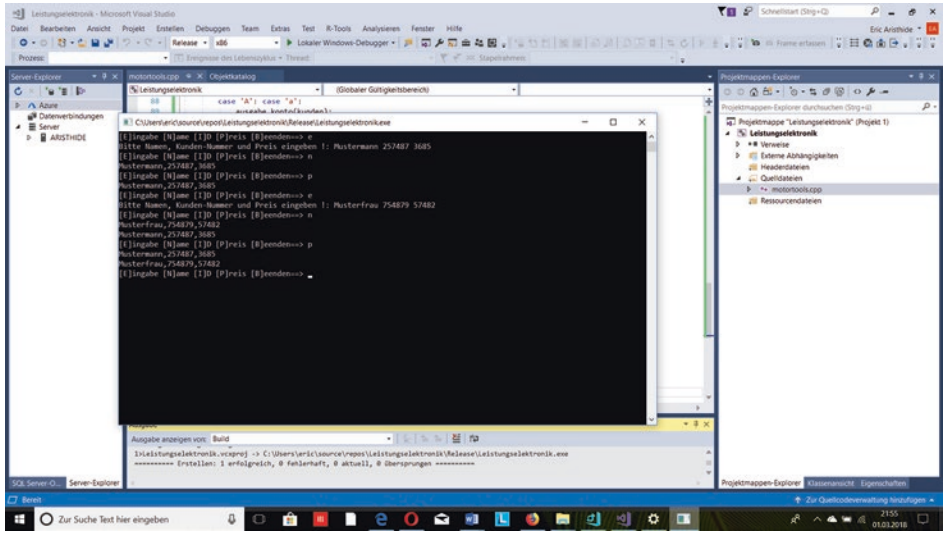


Abb. 1.19 Das Kompilieren des Programms zum Berechnen des Stromverbrauchs

```
<< "dem Verhaeltnis von zwei Zahlen zwischen 700 und 3000 berechnen
!\n";
std::cout << " Beispiel: s 1550.0 2500.0 \n";
std::cout << "Wirkungsgrad berechnen !: "
<<"Bitte das Verhaeltnis von zwei Zahlen zwischen 700 und 3000
berechnen !\n";
std::cout << " Beispiel: w 1550.0 2500.0 \n";
std::cout << "Ausgabe: Ausgeben der Ergebnisse der letzten Berechnung\n";
std::cout << " Beispiel: a\n";
std::cout << "Hilfe: Ausgeben der Hilfe-Info\n";
std::cout << " Beispiel: h\n";
std::cout << "Ende: Beenden des Programms\n";
std::cout << "Beispiel: e\n";
}
char anzeige()
{
 std::cout << "=== S)chlupf W)irkungsgrad A)usgabe H)ilfe E)nde
===\n";
 char ch;
 std::cin >> ch;
 return ch;
}
double ergebnis = 0.0, laeuferdrehzahl, synchronedrehzahl;
void get_eingabe()
{
 std::cin >>laeuferdrehzahl >> synchronedrehzahl;
```

```
}
void berichten()
{
 std::cout << ergebnis << '\n';
}
void schlupfBerechnen()
{
 ergebnis = 1 - (laeuferdrehzahl/synchronedrehzahl);
}
void wirkungsgradBerechnen()
{
 ergebnis = laeuferdrehzahl/synchronedrehzahl;
}
int main()
{
 bool getan = false;
 do
 { switch (anzeige())
 { case 'S':
 case 's':
 get_eingabe();
 schlupfBerechnen();
 berichten();
 break;
 case 'W':
 case 'w':
 get_eingabe();
 wirkungsgradBerechnen();
 berichten();
 case 'A':
 case 'a':
 berichten();
 break;
 case 'H':
 case 'h':
 hilfe_bildschirm();
 break;
 case 'E':
 case 'e':
 getan = true;
 break;
 }
 }
 while (!getan);
}
```

Listing 1.2 Funktionalität der Datenverwaltung mit der Klasse Stromlieferung bezüglich der Sortierung, der Eingabe und Ausgabe der Informationen

```
#include <iostream>
#include <string>
#include <vector>
class Stromlieferung
{
public:
 std::string namen;

 int id;

 double rechnung;
};
void add_konto(std::vector<Stromlieferung>& strom)
{
std::string namen;
int nummer;
double preis;
std::cout << "Bitte Namen, Kunden-Nummer und Preis eingeben !: ";
std::cin >> namen >> nummer >> preis;
Stromlieferung liefern;
liefern.namen = namen;
liefern.id = nummer;
liefern.rechnung = preis;
strom.push_back(liefern);
}
void ausgabe_konto(const std::vector<Stromlieferung>& strom)
{
 int n = strom.size();
 for (int i = 0; i < n; i++)
 std::cout << strom[i].namen << "," << strom[i].id << "," << strom[i].
rechnung << '\n';
}
void swap(Stromlieferung& er1, Stromlieferung& er2)
{
 Stromlieferung temp = er1;
 er1 = er2;
 er2 = temp;
}
bool weniger_als_namen(const Stromlieferung& e1, const Stromlieferung&
e2)
{
```

```
 return e1.namen < e2.namen;
}
bool weniger_als_id(const Stromlieferung& e1, const Stromlieferung&
e2)
{
 return e1.id < e2.id;
}
bool weniger_als_preis(const Stromlieferung& e1, const Stromlieferung&
e2)
{
 return e1.rechnung < e2.rechnung;
}
void sort(std::vector<Stromlieferung>& db, bool (*comp)(const Strom-
lieferung&, const Stromlieferung&))
{
 int size = db.size(); for (int i = 0; i < size - 1; i++)
{
  int kleinste = i;
  for (int j = i + 1; j < size; j++)
   if (comp(db[j], db[kleinste])) kleinste = j;
  if (kleinste != i)
   swap(db[i], db[kleinste]);
 }
}

int main() {
 std::vector<Stromlieferung> kunden;
 char cmd;
 bool done = false;
 do {
  std::cout << "[E]ingabe [N]ame [I]D [P]reis [B]eenden==> ";
  std::cin >> cmd;
  switch (cmd)
  {
  case 'E':
  case 'e':
   add_konto(kunden);
   break;
  case 'A': case 'a':
   ausgabe_konto(kunden);
   break;
  case 'N': case 'n':
    sort(kunden, weniger_als_namen);
    ausgabe_konto(kunden);
    break; case 'I':
```

```
    case 'i':
     sort(kunden, weniger_als_id);
     ausgabe_konto(kunden);
     break;
    case 'P':
    case 'p':
      sort(kunden, weniger_als_preis);
      ausgabe_konto(kunden);
      break;
     case 'B':
     case 'b':
      done = true; break;
  }
 } while (!done);
}
```

1.4.1.2 Modellierungsaspekte mithilfe der Programmierung

Dieser Abschnitt fokussiert auf die Modularisierung durch Klassen und Funktionen mit der Umwandlung der Codes in Models. Hierbei sind die Klassen mithilfe der objektorientierten Programmerstellung definiert. Die Modellierung der Klassen ermöglicht die Darstellung der Objekte der Methoden. Mithilfe der Modellierung wird die Funktionalität eines Programms in Bezug auf die Darstellung der Diagramme beschrieben und vereinfacht.

Mit den Diagrammen vom Eclipse-Papyrus wird die Darstellung der Programmierung in die Modellierung umgewandelt. Die Codes des Programms werden in die Model-Elemente umgewandelt. Listing 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11 und 1.12 zeigen die Programme in C++ zum Charakterisieren der Kenngrößen des Asynchronmotors. Abb. 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25 und 1.26 stellen die Umwandlungen der C++-Programme in die UML-Diagramme. Aktivitätsdiagramme sind von den Abb. 1.18 und 1.19 dargestellt worden. Abb. 1.21 und 1.22 stellen ein Sequenzdiagramm bzw. ein Kommunikationsdiagramm dar. Zustandsdiagramme sind von den Abb. 1.23 und 1.24 dargestellt worden. Innere Klassendiagramme werden von den Abb. 1.25 und 1.26 dargestellt. Mithilfe des Konzepts *„Codes in Models“* werden verschiedene C++-Programme in die UML-Diagramme umgewandelt, um die Funktionalitäten dieser Codes sowohl zu beschreiben als auch zu vereinfachen. In diesem Abschnitt werden verschiedene UML-Diagramme zum Beschreiben der Zusammenhänge zwischen Klassenelementen dargestellt. Von daher wird es mithilfe der UML-Digramme eine Stufe der Abstraktion zum Beschreiben der C++-Programme erreichen.

1.4.1.2.1 Aktivitätsdiagramm

Mithilfe von den Listings 1.3 und 1.4 und der Abb. 1.20 ist das Konzept *„Codes in Models“* leicht verständlich, weil die Abb. 1.20 mithilfe von Eclipse-Papyrus das Verhalten des Programms bezüglich des Drehfeldes des Asynchronmotors ermöglicht. Hierbei sind die Kenngrößen Ständerfrequenz und synchrone Drehzahl parallel

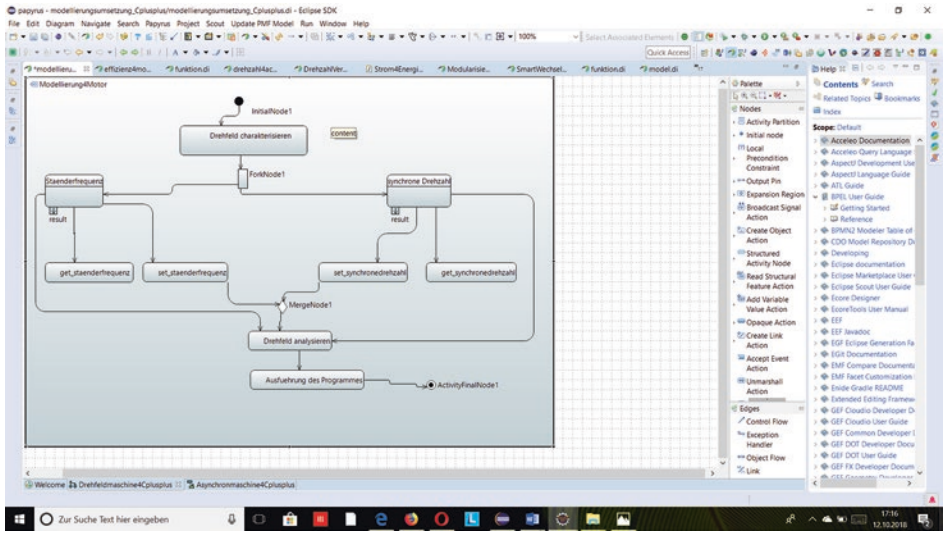


Abb. 1.20 Aktivitätsdiagramm zur Modellierung der Anwendung zur Kundenverwaltung bezüglich der Programmierung aus dem Listing 1.2

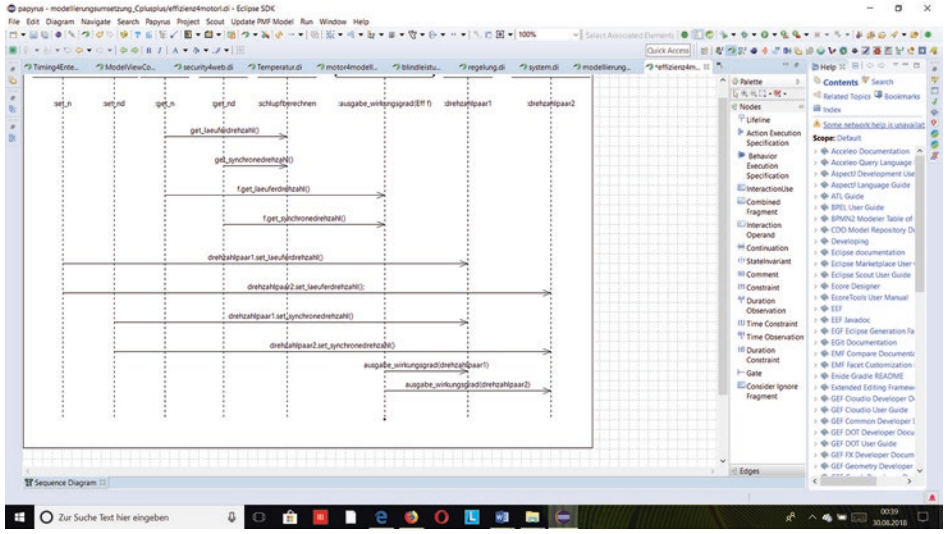


Abb. 1.21 Anwendung des Sequenzdiagramms in der Modellierung des Verhaltens des Asynchronmotors

zum Ausführen des Programms modelliert. Listing 1.3 stellt die Header-Datei, genannt „*Drehfeld.h*“ zur Deklaration der Methoden dar und zeigt zum einen die Konstruktion von Klassenobjekten und zum anderen die Deklaration der Methoden mit dem Spezifizierer „*const*“. Dies soll sicherstellen, dass die Methoden wie z. B. „*get_*

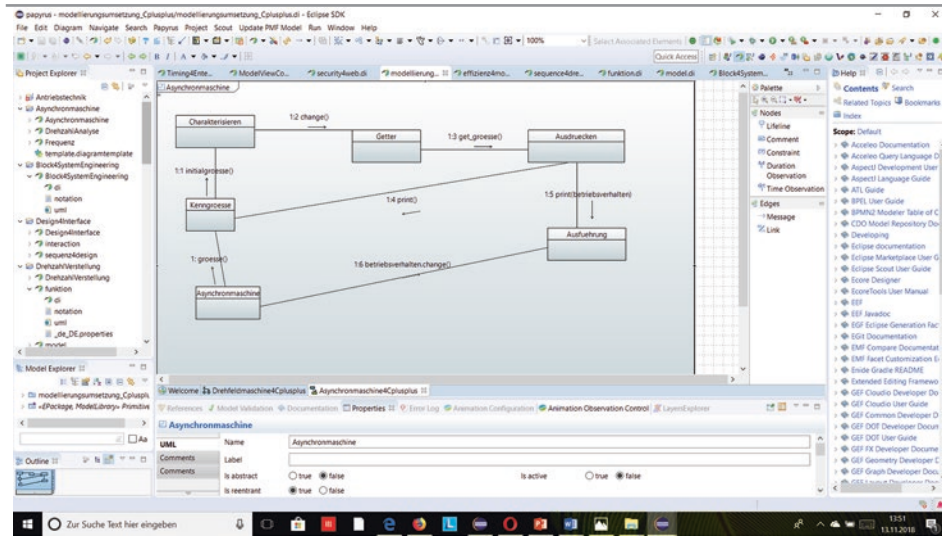


Abb. 1.22 Kommunikation zur Darstellung des Verhaltens des Klassenobjektes *„initial_groesse“* von *„Kenngroesse“* in der Klasse Asynchronmaschine

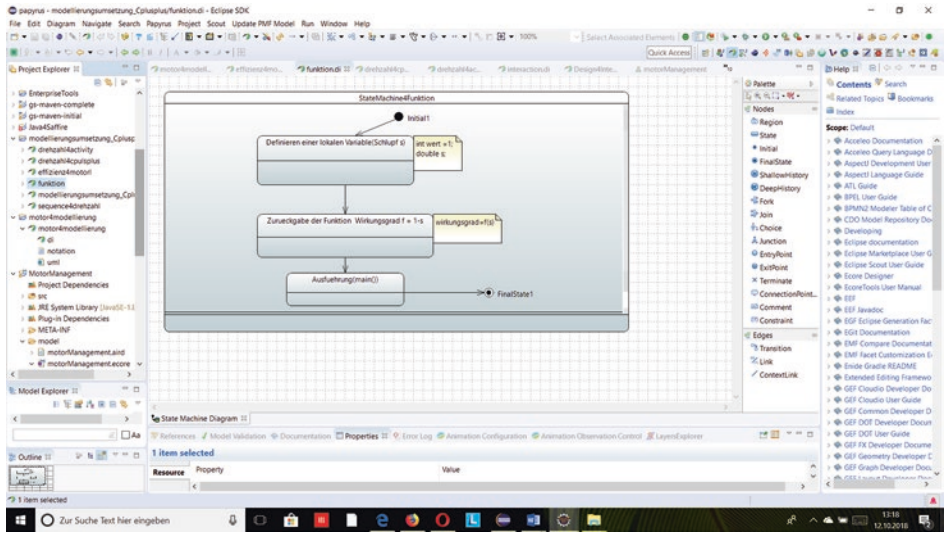


Abb. 1.23 Modellierung der Funktionalität der Berechnung des Wirkungsgrades mithilfe der Lamda-Funktion

staenderfrequenz()“ oder *„get_synchronedrehzahl()“* nicht modifiziert werden. Gemäß Listing 1.3 verfügt die Klasse *„Drehfeld“* eine Methode mit dem demselben Namen wie sie, genannt Konstruktor. Der letzte wird zum Erzeugen und Initialisieren der Klasse *„Drehfeld“* aufgerufen. Gemäß Listing 1.3 ist der Konstruktor *„Drehfeld(double*

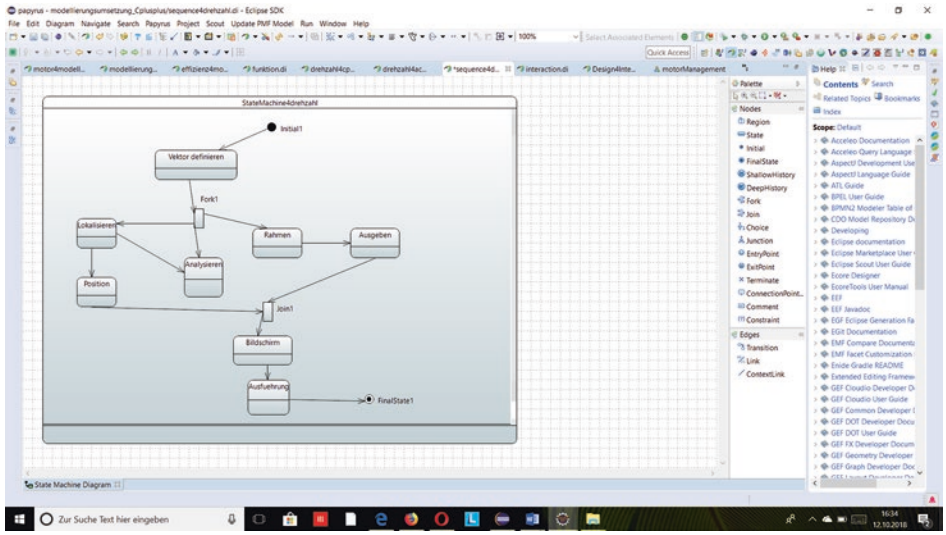


Abb. 1.24 Modellierung der linearen Suche mithilfe des Zustandsdiagramms

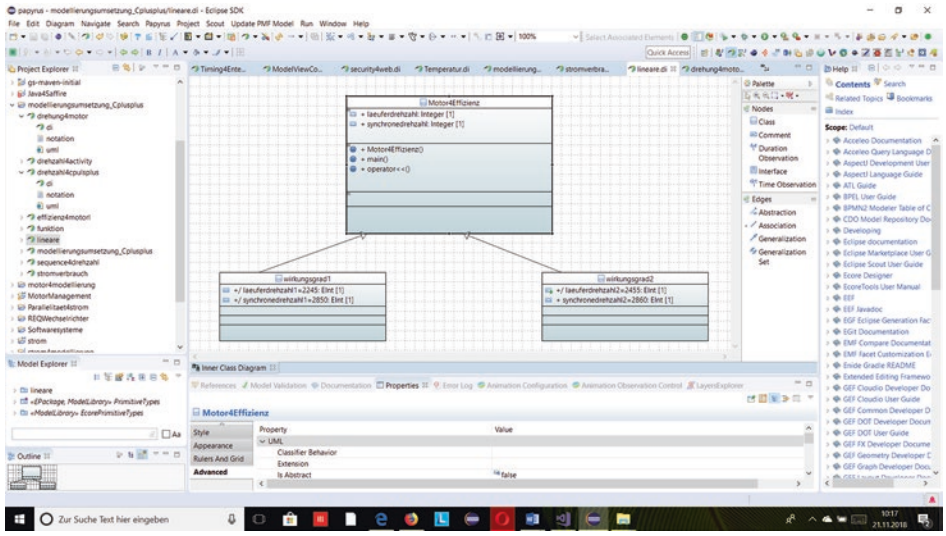


Abb. 1.25 Modellierung der Struktur der Klasse „*Motor4Effizienz*“ mit Hinblick auf den Aufruf der Methode „*operator<<()*“

staenderfrequenz, double synchronedrehzahl)“ zur Erzeugung der Objekte der Klasse „*Drehfeld*“ mit dem Zugriffsmodifizierer „*public*“ deklariert ist. Sowohl in der Header-Datei als auch in Quelltextdatei ist ein Standardkonstruktor durch Standardargumente definiert, wie es in Listing 1.3 und 1.4 zu sehen ist. Die Aufrufe der Methoden

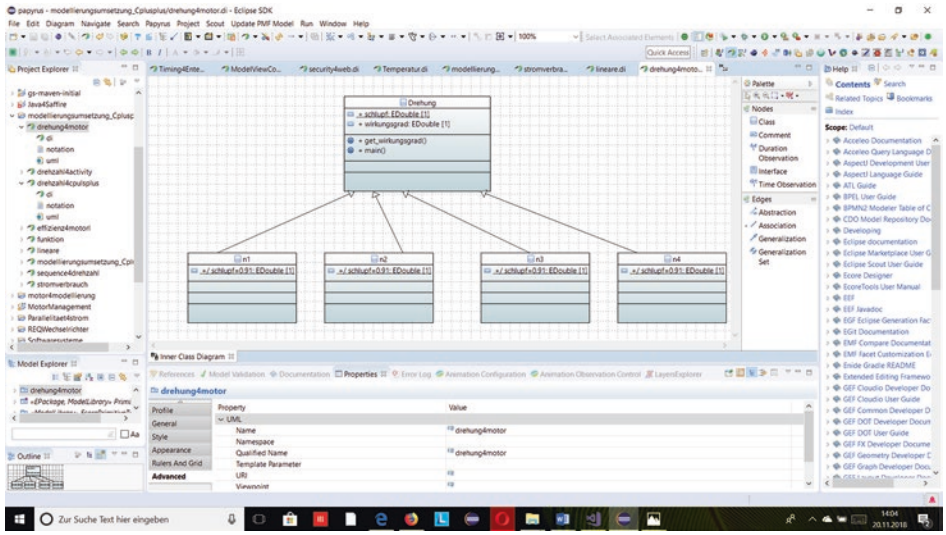


Abb. 1.26 Modellierung der Struktur der Klasse *„Drehung“* mit Hinblick auf den Aufruf eines *„const“* deklarierten Methode, genannt *„get_wirkungsgrad()“*

der Klasse *„Drehfeld“* ermöglichen die Ausführungen der Anweisungen, d. h., die Codes der Methoden werden ausgeführt. Die Methoden *„get_staenderfrequenz()“* und *„get_synchronedrehzahl()“* definieren *„double“*-Parameter, während die Methoden *„set_staenderfrequenz()“* und *„set_synchronedrehzahl()“* mithilfe von dem Zeiger *„this“* und von *„void“* als Typ des Methodenwertes keine Ergebniswerte zurückliefern. Wobei der Zeiger *„this“* ein Schlüsselwort zum Zugreifen in diesen Methoden und zum Zuweisen an die Klassenelemente darstellt. Listing 1.4 stellt die Quelltextdatei genannt *„Drehfeld.cpp“* zum Implementieren der Methoden der Klasse *„Drehfeld“* dar. Die Methode *„analyse()“* in der cpp.datei vom Listing 1.4 enthält zwei Argumente d_1 und d_2 als Referenzen von Drehfeld-Objekten. Wobei d_1 und d_2 bezüglich der Parameter des Konstruktors „Drehfeld“ sowohl andere Namen für *„staenderfrequenz“* bzw. *„synchronedrehzahl“* darstellen, als auch im Konstruktor deklariert wurden. Hierbei sind die Referenzen zum einen Klassenelemente von *„Drehfeld“*, zum anderen immer konstant. Die Methode *„analyse()“* wirkt wie eine Funktion mit den Argumente d_1 und d_2. Gemäß Listing 1.4 erfolgt der Aufruf der Methode *„analyse()“* in der main-Methode mithilfe der Übergabe der Referenzen d_1 und d_2. Wobei die main-Methode die Referenzen aufrufen. Die Funktionalität dieses Programms wirkt auf die Referenzierung der Objekte mithilfe der Methode *„analyse()“* wie einer Lamda-Funktion. Hierbei werden nur die Argumente dieser Funktion übergeben. Die Abb. 1.20 vereinfacht die Beschreibungen von Listings 1.3 und 1.4, indem das Diagramm Elemente zum Modellieren in Bezug auf Staenderfrequenz und synchrone Drehzahl darstellt. Dies ermöglicht das Verständnis der Programmierung mit C++, weil es eine parallele Modellierung der Elemente zum Ausführen des Programms gibt. Abb. 1.20 zeigt die Setter- und Getter-Methoden zum

Beschreiben der Funktionalität der Klassenobjekte von *„Drehfeld“*. Gemäß Abb. 1.20 ermöglicht die parallele Modellierung das Beschreiben der Methode *„analyse()“* vom Listing 1.4 mithilfe der Referenzierung auf Ständerfrequenz und synchrone Drehzahl.

1.4.1.2.2 Sequenzdiagramm

Sequenzdiagramme gemäß Abb. 1.21 beschreiben das Verhalten des Systems in Bezug auf Interaktionen zwischen Objekten zum einen und auf Darstellungen der Reihenfolge dieser Objekte zum anderen. Hierbei werden Interaktionen zwischen Funktionen oder Methoden des Programmes modelliert. Gemäß Listing 1.5 werden verschiedene Funktionen der Klasse *„Effizienz“* mit C++ zum Charakterisieren des Asynchronmotors definiert, wobei der Konstruktor *„Effizienz“* einen Standardkonstruktor darstellt. Die Beziehungen zwischen Klassenobjekten ermöglicht die Anwendung des Sequenzdiagramms in der Modellierung des Verhaltens des Asynchronmotors. Die Getter- und Setter-Methoden werden zum Belegen und zur Zugänglichkeit der Attribute „laeuferdrehzahl“ und „synchronedrehzahl“ definiert. Gemäß Listing 1.5 sind die Funktionen *„schlupfberechen()“* und *„ausgabe_wirkungsgrad()“* zum einen Elementefunktionen der Klasse *„Effizienz“* und zum anderen, Teil ihres Geltungsbereichs. Die erste Funktion verfügt über zwei Elemente des Objektes, genannt f1 und f2, welche vom Typ *„Effizienz“* sind. Diese Elemente des Objekts werden für ihre Funktion aufgerufen. Außerdem ruft die Funktion *„drehzahlpaar()“* aus der main-Funktion die Elemente f1 und f2 zum Berechnen des Schlupfes auf. Die Funktion *„ausgabe_wirkungsgrad()“* stellt die Berechnung des Wirkungsgrades mithilfe des Elementes des Objektes, genannt *„f!“*, und Getter-Funktionen, genannt *„get_laeuferdrehzahl()“* und *„get_synchronedrehzahl()“*, dar. In der *„main“*-Funktion aus dem Listing 1.5 ist es zu bemerken, dass der Aufruf der Funktion *„drehzahlpaar()“* zum Aufruf der Funktionen *„schlupfberechen()“* und *„ausgabe_wirkungsgrad()“* durchführt. Hierbei werden diese Aufrufe zum Berechnen einerseits des Schlupfes und andererseits des Wirkungsgrades mithilfe der Kenngrößen synchrone Drehzahl und Läuferdrehzahl. Die Funktionalität der *main*-Funktion zeigt einen Call-Value-Effekt beim Aufruf der Funktion *„drehzahlpaar()“*. Die Logik stimmt auch, weil die Berechnungen des Schlupfes und des Wirkungsgrades von den Drehzahlen, d. h. der Funktion *„drehzahlpaar()“* abhängen, welche die Funktionalität der Attribute *„laeuferdrehzahl“* und *„synchronedrehzahl“* darstellt. Es ist zu bemerken, dass die Lambda-Ausdrücke zum Übergeben der Funktionsobjekte *„drehzahlpaar1“* und *„drehzahlpaar2“* in den Funktionen *„schlupfberechen()“* und *„ausgabe_wirkungsgrad()“* angewendet sind.

1.4.1.2.3 Kommunikationsdiagramm

In UML zeigt ein Kommunikationsdiagramm die Interaktionen zwischen den Objekten oder Rollen, die Lebenslinien zugeordnet sind, sowie die Nachrichten an, die zwischen Lebenslinien übergeben werden [16]. Mit Kommunikationsdiagrammen werden das Zusammenwirken der Objekte in einem System oder einer Anwendung analysiert. Gemäß Listing 1.6, 1.7, 1.8 und 1.9 und Abb. 1.22 wird das Verhalten des Klassenobjektes

„initial_groesse“ von *„Kenngroesse“* in der Klasse *„Asynchronmaschine“* analysiert. Listing 1.6 zeigt, dass das Objekt *„groesse“* von *„KennGroesse“* privat deklariert wurde und deshalb im anderen Teil des Programms wie im Listing 1.7 zugegriffen wird. Gemäß Listing 1.6 und 1.7 wird *„initial_groesse“* in der Klasse *„Asynchronmaschine“* mithilfe derer Konstruktoraufrufes initialisiert. D. h. das letzte Klassenobjekt verhält sich als Teilobjekt, welches ein Klassenelement von Asynchronmaschine darstellt. Wenn die Objekte einer Klasse B nur in einer Klasse A, in der sie als Klassenelemente auftreten, benötigt werden, kann man auch die Definition von B lexikalisch innerhalb von A vornehmen [17]. Gemäß Listing 1.6 und 1.7 bedeutet dies, dass sich die Klasse *„Kenngroesse“* als eine eingebettete Klasse in der Klasse Asynchronmaschine verhält. Die Methode *„get_groesse“* von *„KennGroesse“* ist in der Klasse *„Asynchronmaschine“* definiert und ermöglicht mittels ihrer Implementierung den Zugriff auf das Objekt *„groesse“* von *„KennGroesse“*. Aber mithilfe des Konstruktors *„Asynchronmaschine“* werden die Objekte *„initial_groesse“* und *„groesse“* gleich. Die Kommunikation zwischen der eingebetteten Klasse *„KennGroesse“* und der umgebenden Klasse *„Asynchronmaschine“* wird mithilfe von Verwendungen der Enumeratoren „Drehzahl“, *„Schlupf“* und *„Wirkungsgrad“* ermöglicht, wobei gemäß Listing 1.7 die Instanzen von *„KennGroesse“* als Datenelemente von *„Asynchronmaschine“* vereinbart sind. Dies ermöglicht den Zugriff auf die Elemente von *„Asynchronmaschine“*. Die Kommunikation zwischen *„KennGroesse“* und *„Asynchronmaschine“* basiert auf dem Transfer der Objekte der ersten Klasse in die zweite. Z. B. sind die Klassenobjekte *„initial_groesse“* und *„groesse“* von *„KennGroesse“* in der Klasse *„Asynchronmschine“* als Teilobjekt definiert. Außerdem ermöglichen die Implementierungen der Methoden *„change()“* und *„print()“* von *„Asynchronmaschine“* den Zugriff auf die Objekte *„initial_groesse“* und *„groesse“*. Listing 1.8 zeigt, wie der Aufruf *„betriebsverhalten.change()“* für den Zugriff mittels des Enumerators *„Schlupf“* von *„KennGroesse“* sinnvoll und erlaubt ist. Dies ist möglich, weil betriebsverhalten ein in *„main()“* definiertes Objekt der Klasse *„Asynchronmaschine“* darstellt.

1.4.1.2.4 Zustandsdiagramm

Listing 1.9 und Abb. 1.23 zeigen die Funktionalität der Anwendung von Lambda-Ausdrücken für C++ bzw. UML-Diagramm. Hierbei wird die Nutzung der Funktion zum Zurückgeben einer Lambda-Funktion mithilfe der C++-Programmierung vorteilhaft. Das Listing 1.9 fokussiert auf Closure zum Transportieren von zugegriffen lokalen Variablen außerhalb einer Funktion.

Die Funktion *wirkungsgrad()* gibt eine Lambda-Funktion zum ihrem Aufruf zurück. Wobei diese Lambda-Funktion die lokale Variable *„wert“* in der Berechnung nutzt. Dies bedeutet, dass diese Lambada-Funktion eine Closure zum Zugriffen auf die lokale Variable *wert* der Funktion *wirkungsgrad()* darstellt. Das Zustandsdiagramm mit Eclipse-Papyrus aus der Abb. 1.23 ermöglicht das Umsetzen der Programmierung in die Modellierung. Hierbei werden Code aus dem Listing 1.9 in die Modelle umgewandelt. Das Zustandsdiagramm zeigt das Verhalten des Systems bezüglichder Modellierung der Funktionalität der Berechnung des Wirkungsgrades mithilfe der Lamda-Funktion.

Zuerst wird die lokale variable *wert* definiert und mithilfe einer Lambda-Funktion außerhalb von der Funktion *wirkungsgrad()* übertragen. Anschließend wird die Funktion *wirkungsgrad()* die Lambda-Funktion mit der lokalen Variable wert die Berechnung des Wirkungsgrad mithilfe der Variable *schlupf* oder *s* zurückgeben. Gemäß Listing 1.9 und Abb. 1.23 berechnet die Funktion f den Wirkungsgrad aus *1* und *s*. Das Ausführen des Programms in der *main*-Funktion wird mithilfe von dem Schlüsselwort *auto* zum einen und mithilfe des Überladens der Funktion wirkungsgrad() ermöglicht. Danach wird die Berechnung dieser linearen Funktion *f* mit den Werten von der Variable *s* erfolgen.

Listing 1.10 und Abb. 1.24 zeigen die Programmierung der Funktionalität zum linearen Suchen eines Elements in einem Vektor bzw. die Modellierung dieser Suche mithilfe des Zustandsdiagrammes. Gemäß Listing 1.10 fokussiert die Suche eines Elements mithilfe einer Funktion genannt *lokalisieren(),* die beim Treffen des gesuchten Elements eine Markierung darstellt. Falls es kein Treffer gibt, gibt die Funktion *lokaliseren()* den Wert −1 zurück. Die Funktion *lokalisieren()* koordiniert die Aufgabe anderer Funktionen wie z.: *bildschirm(),* welche über zwei Parameter verfügt und einen Raumabstand zwischen gesuchten Elementen des Vektors berechnet. Hierbei drückt die Funktion *bildschirm()* die Inhalte des Vektors aus. Zum Darstellen der Grafik auf dem Bildschirm ist die Nutzung der Dienste anderer Funktionen wichtig, wie es auf dem Listing 1.10 und auf der Abb. 1.24 zu sehen ist. Die Modellierung der linearen Suche mit dem Zustandsdiagramm ermöglicht das Darstellen linearer Suche eines Elements in einem Vektor nach einer Ordnung. Gemäß Listing 1.10 und Abb. 1.24 hängt das Anzeigen der Informationen auf dem Bildschirm von dem Positionieren und dem Ausgeben der gefundenen Elemente im Vektor ab. Abb. 1.24 zeigen wie die Zustände Rahmen und Ausgeben einerseits und Lokalisieren und Analysieren parallele Prozesse darstellen. Von daher gibt es eine Beziehung zwischen den Zuständen zum Darstellen der Parallelisierung der linearen Suche.

1.4.1.2.5 Inneres Klassendiagramm

Dieser Abschnitt fokussiert auf die Struktur einer Klasse und derer Darstellung mithilfe der Ausgabe eines Klassenobjekts. Gemäß Listing 1.11 und Abb. 1.25 wird eine Ausgabeanweisung der Form *„cout << f“* ermöglicht, wenn der Ausgabeoperator *„<<“* überladen wird. Im Listing 1.11 ist die Ausgabe des Klassenobjekts *„f“* dargestellt, Die Methode *„operator<<()“* wird gemäß Listing 1.11 folgendend aufgerufen:

```
os << f.laeuferdrehzahl << '/' << f.synchronedrehzahl;
```

Gemäß Listing 1.11 verfügt die Methode *„operator<<()“* über zwei Parameter. Der erste ist vom Typ *„ostream&“* und der zweite vom Typ *„const Motor4Effizienz“*. Hierbei sind *„os“* bzw. f Referenz von dem ersten bzw. von dem zweiten Parameter. Der Rückgabewert der Methode *„operator<<()“* ist als Referenz auf das als erstes Argument übergebene *„ostream“*-Objekt. Der Ausdruck *„cout.operator<<(wirkungsgrad1)“* ist die

Abkürzung des Ausdruckes *„cout.operator<<(wirkungsgrad1)<<'\' oder std::cout<<wirkungsgrad1<<'\n'“*. Gemäß Listing 1.11 gibt der Ausdruck *„std::cout<<wirkungsgrad1<<'\n'“* den Wert von *„wirkungsgrad“* (Man könnte statt *„Wirkungsgrad“* auch*„f“* schreiben.) auf dem Bildschirm aus und gibt *„std::cout“*-Objekte selbst zurück.

Es ist zu bemerken, dass der *„return“*-Wert (d. h. *„std::cout“*) gemäß Listing 1.11 das Überladen der Methode *„operator<<()“* ermöglicht. Die Umsetzung der Programmierung in die Modellierung wird mithilfe der Abb. 1.25 realisiert. Die Abb. 1.25 stellt die Modellierung der Struktur der Klasse *„Motor4Effizienz“* mit Hinblick auf die Ausgabe eines Klassenobjekts dar. Das Klassendiagramm zeigt die Attribute *„laeuferdrehzahl“* und *„sycnhronedrehzahl“* sowie die Methoden *„operator<<()“* und *„main()“*. Zwei Klassenobjekte *„wirkungsgrad1“* und *„wirkungsrad2“* stellen die Anwendungen der Ausgabe von Klassenobjekten in der Modellierung dar. Ziel ist es, die Berechnung des Wirkungsgrads mithilfe der Modellierung darzustellen. Wobei die *main*-Methode gemäß Listing 1.11 die Deklaration der Objekte *„wirkunsgrad1“* und *„wirkungsrad2“* zum Überladen der Methode *„operator<<()“* zeigt. Eigentlich ermöglicht die Methode *„operator<<()“* die Berechnung des Wirkungsgrades des Motors.

Listing 1.12 und Abb. 1.26 zeigen die Struktur der Klasse *„Drehung“* mit ihrer Eigenschaft und ihrer Klassenobjekten. Die Struktur der Code gemäß Listing 1.12 stellt den Aufruf eines *„const“* deklarierten Methode, genannt *„get_wirkungsgrad()“*, bezüglich der Deklaration von den nicht *„const“* Klassenobjekten n1, n2, n3 und n4 dar. Gemäß Listing 1.12 ist der Aufruf der Methode *„get_wirkungsgrad()“* mithilfe der Klassenobjekte n1, n2, n3 und n4 erlaubt und sinnvoll, weil der Spezifizierer *const* in der Deklaration der Methode *get_wirkungsgrad()* angegeben ist. Es ist zu bemerken, dass die Klassenobjekte n1, n2, n3 und n4 als *„static“* deklariert sind. Dies bedeutet, dass der Wert des Attributes *„schlupf“ nicht* geändert wird. Abb. 1.25 stellt das Klassendiagramm von *„Drehung“* und zeigt die Attribute *„schlupf“* und *„wirkungsgrad“* sowie die Methoden *„get_wirkungsgrad ()“* und *„main()“*.

Listing 1.3 Überblick über die Klasse *„Drehfeld“* zum Charakterisieren des Asynchronmotors in Hinblick auf die Getter- und Setter-Methoden

```
#pragma once
class Drehfeld
{
 double staenderfrequenz;
 double synchronedrehzahl;
public:
 Drehfeld(double staenderfrequenz, double synchronedrehzahl) :
  staenderfrequenz(staenderfrequenz), synchronedrehzahl(synchronedrehz
  ahl)
 {
```

```
 }
 double get_staenderfrequenz() const
 {
  return staenderfrequenz;
 }
 double get_synchronedrehzahl() const
 {
  return synchronedrehzahl;
 }

 void set_staenderfrequenz(double staenderfrequenz)
 {
  this->staenderfrequenz = staenderfrequenz;
 }
 void set_synchronedrehzahl(double synchronedrehzahl)
 {
  this->synchronedrehzahl = synchronedrehzahl;
 }
```

Listing 1.4 Darstellung sowohl der Methoden der Klasse „*Drehfeld*“ in der Quelltextdatei als auch der Erzeugung des Objekts „*Drehfeld*“ in der main()-Funktion der Anwendung

```
#include "Drehfeld.h"
#include <iostream>
Drehfeld::Drehfeld(double staenderfrequenz, double synchronedrehzahl) :
 staenderfrequenz(staenderfrequenz), synchronedrehzahl(synchronedrehzahl)
{
}
double Drehfeld::get_staenderfrequenz()
const
{
 return staenderfrequenz;
}
double Drehfeld::get_synchronedrehzahl()
const
{
 return synchronedrehzahl;
}
void Drehfeld::set_staenderfrequenz(double staenderfrequenz)
{
 this->staenderfrequenz = staenderfrequenz;
}
void Drehfeld::set_synchronedrehzahl(double synchronedrehzahl)
```

```
{
 this->synchronedrehzahl= synchronedrehzahl;
}
double analyse(const Drehfeld& d1, const Drehfeld& d2)
{
 return 0.0;
}
int main()
{
 Drehfeld d1(3000.0, 200.0), d2(0.0, 0.0);
 std::cout << analyse(d1, d2) << '\n';
}
```

Listing 1.5 Beschreibung der Funktionalität der verschiedenen Klassenobjekten zum Berechnen des Wirkungsgrades des Asynchronmotors

```
#include <iostream>
#include <cstdlib>
class Effizienz
{
 int laeuferdrehzahl; int synchronedrehzahl;
public:
 Effizienz(int n, int nd) : laeuferdrehzahl(n), synchronedrehzahl(nd)
 {
  if (nd == 0)
  {
   std::cout << "Eine nulle synchrone Drehzahl ist nicht in
   Ordnung!\n"; exit(1);
  }
 }
 Effizienz() : laeuferdrehzahl(0), synchronedrehzahl(1)
 {
 }
 void set_laeuferdrehzahl(int n)
 {
  laeuferdrehzahl = n;

 }
 void set_synchronedrehzahl(int nd) {
  if (nd != 0) synchronedrehzahl = nd; else
  {
   std::cout << "Eine nulle synchrone Drehzahl ist nicht in
   Ordnung!\n";
   exit(1);
```

```
  }
 }
 int get_laeuferdrehzahl()
 {
  return laeuferdrehzahl;
 }
 int get_synchronedrehzahl()
 {
  return synchronedrehzahl;
 }
};
Effizienz schlupfberechnen(Effizienz f1, Effizienz f2)
{

 return  {  1-(f1.get_laeuferdrehzahl()  *  f2.get_laeuferdrehzahl()),
 1-(f1.get_synchronedrehzahl() * f2.get_synchronedrehzahl())};
}
void ausgabe_wirkungsgrad(Effizienz f)
{
 std::cout << f.get_laeuferdrehzahl() << "/" << f.get_synchronedrehzahl();
}
int main() {
 Effizienz drehzahlpaar(2500, 2800);
 std::cout << "Der Wirkungsgrad ist";
 ausgabe_wirkungsgrad(drehzahlpaar); std::cout << '\n';
 drehzahlpaar.set_laeuferdrehzahl(2900);
 drehzahlpaar.set_synchronedrehzahl(3000);
 std::cout << "Der neuer Wirkungsgrad ist ";
 ausgabe_wirkungsgrad(drehzahlpaar);
 std::cout << '\n';
 Effizienz drehzahlpaar1{ 2500, 2800 },
  drehzahlpaar2{ 2750, 3000 };
 auto prod = schlupfberechnen(drehzahlpaar1, drehzahlpaar2);
 std::cout << "Die Drehzahldifferenz ";
 ausgabe_wirkungsgrad(drehzahlpaar1);
 std::cout << " und ";
 ausgabe_wirkungsgrad(drehzahlpaar2);
 std::cout << " ist ";
 ausgabe_wirkungsgrad(prod);
 std::cout << '\n';
}
```

Listing 1.6 Beschreibung des Verhaltens des Asynchronmotors mithilfe der Kenngrößen Drehzahl, Schlupf und Wirkungsgrad

```
"asynchronmaschine.h"
#pragma once
enum class KennGroesse
{

 Drehzahl, Schlupf, Wirkungsgrad
};
class Asynchronmaschine
{
private: KennGroesse groesse;

public:

 Asynchronmaschine(KennGroesse initial_groesse);
 void change();
 KennGroesse get_groesse() const;
};
```

Listing 1.7 Header-Datei *„asynchronmaschine.h"* zum Darstellen der Klassenobjekte von *Kenngroesse* als Klassenelemente von *Asynchronmaschine* mit Hinblick auf Erzeugung und Initialisierung der Klassenobjekte im Konstruktor

```
#include "asynchronmaschine.h"
Asynchronmaschine::Asynchronmaschine(KennGroesse initial_groesse)
{
 switch (initial_groesse)
 {
 case KennGroesse::Drehzahl:
 case KennGroesse::Schlupf:
 case KennGroesse::Wirkungsgrad:
  groesse = initial_groesse;
  break;
 default:
  groesse = KennGroesse::Drehzahl;
 }
}
void Asynchronmaschine::change()
{
 if (groesse ==KennGroesse::Drehzahl)
  groesse =KennGroesse::Schlupf;
```

```
 else if (groesse == KennGroesse::Schlupf)
  groesse = KennGroesse::Wirkungsgrad;
 else if (groesse ==KennGroesse::Wirkungsgrad)
  groesse = KennGroesse::Drehzahl;
}
KennGroesse Asynchronmaschine::get_groesse() const {
 return groesse;
}
```

Listing 1.8 Quelltextdatei *„asynchronmaschine.cpp“* zum Darstellen der Funktionalität der Aufrufe der Funktion *print()* in *main()* mithilfe von *betriebsverhalten()* und *change()*

```
#include <iostream>
#include "asynchronmaschine.h"
void print(Asynchronmaschine asyn)
{
KennGroesse groesse = asyn.get_groesse();
std::cout << "+-----+\n";
std::cout << "| |\n";
if (groesse == KennGroesse::Drehzahl) std::cout << "| (Drehzahl)
|\n";
else std::cout << "| ( ) |\n";
std::cout << "| |\n";
if (groesse == KennGroesse::Wirkungsgrad) std::cout << "| (Wirkungs-
grad) |\n";
else std::cout << "| ( ) |\n";
std::cout << "| |\n";
if (groesse == KennGroesse::Schlupf) std::cout << "| (Schlupf) |\n";
else std::cout << "| ( ) |\n"; std::cout << "| |\n";
std::cout << "+-----+\n";
}
int main()
{
Asynchronmaschine betriebsverhalten(KennGroesse::Schlupf);
while (true)
{
 print(betriebsverhalten);
 betriebsverhalten.change();
std::cin.get();
}
}
```

Listing 1.9 Funktionalität der Berechnung des Wirkungsgrades mithilfe der Lamda-Funktion

```
#include <iostream>
#include <functional>
std::function<double(double)> wirkungsgrad()
{
 int wert = 1; // Definieren einer lokalen Variable
 return [wert](double schlupf)
 {
  return wert - schlupf;
 }; // Zurueckgabe einer Funktion
}
int main()
{ auto f = wirkungsgrad();
std::cout << f(0.875) << '\n';
std::cout << f(0.795) << '\n';
}
```

Listing 1.10 Lineare Suche mithilfe der Funktion zum Ermitteln der Position eines Elements in einem Vektor

```
#include <iostream>
#include <vector>
#include <iomanip>
int lokalisieren(const std::vector<int>& x, int analysieren)
{
 int n = x.size();
 for (int i = 0; i < n; i++)
  if (x[i] == analysieren)
   return i;
 return -1;
}
void rahmen(int i)
{
 if (i > 3000) std::cout << "Drehzahl Drehzahl Drehzahl" << '\n';
 else
  std::cout << std::setw(7) << i;
}
void ausgeben(const std::vector<int>& v)
{
for (int i : v)
rahmen(i);
}
void bildschirm(const std::vector<int>& a, int wert)
```

```
{
 int position = lokalisieren(a, wert);
if (position >= 0)
 {
  ausgeben(a);
std::cout << '\n';
position = 7*position + 5;
  std::cout << std::setw(position);
std::cout << " ^ " << '\n';
  std::cout << std::setw(position);
std::cout << " | " << '\n';
  std::cout << std::setw(position);
std::cout << " +-- " << wert << '\n';
 }
else
{ std::cout << wert << " not in ";
 ausgeben(a);
 std::cout << '\n';
}
 std::cout << "……….." << '\n';
}
int main()
{
 std::vector<int> list{ 700, 1500, 1550, 1700, 1800, 2550,2800,
2800, 2900};
bildschirm(list, 5); bildschirm(list, 700); bildschirm(list, 1500);
bildschirm(list,2550); bildschirm(list, 2800);
}
```

Listing 1.11 Darstellung der Struktur der Klasse *Motor4Effizienz* bezüglich des Überladenes der Methode *operator<<()*

```
#include <iostream>
#include <cstdlib>
// Modellierung einer rationalen Zahl
class Motor4Effizienz
{
 int laeuferdrehzahl; int synchronedrehzahl;
public:

 // Initializierung der Komponenten eines Objektes der Klasse
 Motor4Effizienz
 Motor4Effizienz(int n, int nd): laeuferdrehzahl(n), synchronedrehzahl(nd)
 {
  if (nd == 0)
```

```
 { // Der Bildschirm zeigt einen Fehler an.
  std::cout << "Synchrone Drehzahl ist null ! Das ist ein Fehler !\n";
  exit(1);
  //Beenden des Programmes

 }
 }
 // Der Default-Konstruktor erzeugt eine rationale Null-Zahl
 // 0/1000
 Motor4Effizienz(): Motor4Effizienz(0, 1000)
 {

 }
 // Der Operator hat den Zugang zum Inneren des Objektes der Klasse.

 friend std::ostream& operator<<(std::ostream& os, const Motor4Effizienz&
 f);
};
std::ostream& operator<<(std::ostream& os, const Motor4Effizienz& f)
{
 os << f.laeuferdrehzahl << '/' << f.synchronedrehzahl;
 return os;
}
int main()
{
 Motor4Effizienz wirkungsgrad1{2445, 2850};
 // Der Wirkungsgrad als Verhaeltnis zwischen 2445/2850
 std::cout << "Der Wirkungsgrad ist " << wirkungsgrad << '\n';
 Motor4Effizienz wirkungsgrad2{2455, 2860};
 // Der Wirkungsgrad als Verhaeltnis zwischen 2445/2850
 std::cout << "Der Wirkungsgrad ist " << wirkungsgrad2 << '\n';
}
```

Listing 1.12 Struktur der Klasse *Drehung* mit dem Hinblick auf Aufruf einer *const* deklarierten Methode

```
#include <iostream>
class Drehung
{
 static double schlupf;
 double wirkungsgrad;
public:
 Drehung() : wirkungsgrad(1-schlupf)
 {
```

```
 }
 double get_wirkungsgrad() const
 {
  return wirkungsgrad;
 }
};
double Drehung::schlupf = 0.91;
int main()
{
Drehung n1,n2, n3, n4;
std::cout << "n1 Wirkungdgrad = " << n1.get_wirkungsgrad() << '\n';
std::cout << "n2 Wirkungdgrad = " << n2.get_wirkungsgrad() << '\n';
std::cout << "n3 Wirkungdgrad = " << n3.get_wirkungsgrad() << '\n';
std::cout << "n4 Wirkungdgrad = " << n4.get_wirkungsgrad() << '\n';
}
```

1.4.2 Modellierungen von Java-Anwendungen mithilfe von UML

Dieser Abschnitt beschäftigt sich mit dem Thema Modellierung von Java-Anwendungen mithilfe einerseit von UML-Designer und andererseits von Eclipse-Papyrus zur Entwicklung von IT-Lösungen auf Basis von Java. Es geht darum, IT-Lösungen mithilfe der Modellierung bezüglich der Diagramme zu entwickeln.

1.4.2.1 Modellierung von Anwendungen für den Einsatz von Resonanzelementen Kondensator und Spule mit Eclipse-UML-Designer

Vom Modell zu den Codes ermöglicht die Beschreibung der Architektur des Diagramms mithilfe der Programmierung. Hierbei wird die Programmiersprache Java zum Beschreiben des Diagramms angewendet. Das Klassendiagramm gemäß Abb. 1.27 und Listing 1.13, 1.14, 1.15 und 1.16 zeigt ein objektorientiertes Modellierungsdesign mithilfe des Design-Musters *„Abstract“*. Hierbei stellt die abstrakte Klasse Resonanz eine Hilfeklasse zum Implementieren der Funktionalitäten der Klasse Kondensator und Spule. Wobei die abstrakte Basisklasse *„Resonanz“* aus abstrakten Methoden *„frequenzBerechnen()“* und *„spannungsBerechnen()“* besteht. Gemäß Listing 1.14 und 1.15 und Abb. 1.27 ist die Vererbungshierarchie mithilfe des Schlüsselwortes *„extend“* und der Beziehung genannt *„Generalization“* zu erkennen. Das Klassendiagramm auf der Abb. 1.27 von Open Source Eclipse-UML-Designer zeigt deutlich die Beziehungen zwischen den Klassen mithilfe der Pfeile für *„Generalisation“* und *„Assoziation“*. Die untergeordneten Klassen *„Kondensator“* und *„Spule“* erben die Methoden *„spannungsBerechnen()“* und *„frequenzBerechnen()“* der abstrakten Klasse *„Resonanz“*. Die Programmierung des Charakterisierens der Bauelemente Kondensator und Spule fokussiert auf die Anwendungen der abstrakten Methoden der obergeordnenten Klasse *„Resonanz“* in den untergeordnenten Klassen zum Berechnen der Kenngrößen Frequenz

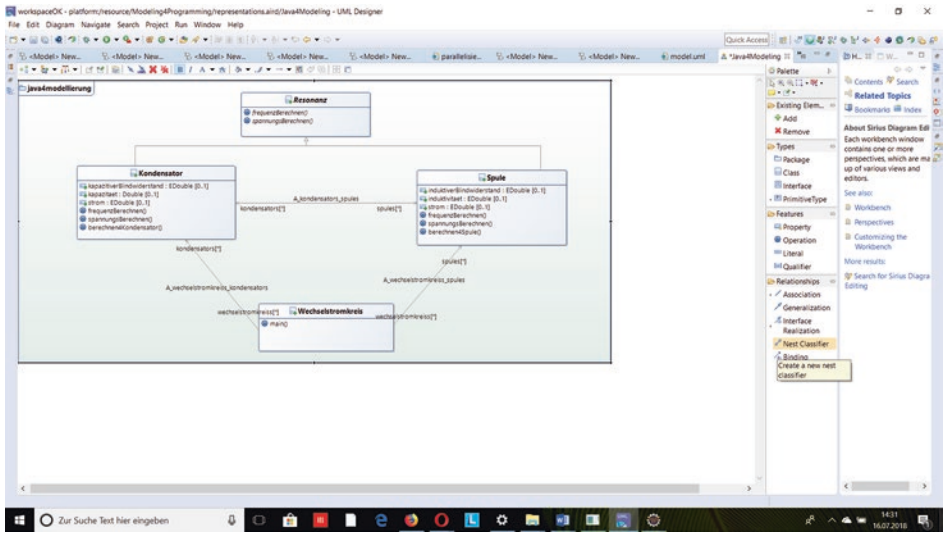


Abb. 1.27 Modellierung des Charakterisierens der Bauelemente Kondensator und Spule mithilfe des Klassendiagramms

und Spannung. Dies erfolgt mithilfe der Getter- und Setter-Methoden in den untergeordneten Klassen. Anschließend gemäß Listing 1.14 und 1.15 werden die Methoden *„Berechnen4kondensator()“* und *„berechnen4Spule()“* zum Berechnen der Kenngrößen implementiert und gemäß Listing 1.16 in der *„main“*-Methode der Hauptklasse *„Wechselstromkreis“* aufgerufen. Die Modellierung der Strukturen der Vererbungshierarchie wird mithilfe des Klassendiagramms gemäß Abb. 1.27 ermöglicht. Das Klassendiagramm von Open Source Eclipse-UML-Designer zeigt sowohl die Attribute und die Methoden der untergeordneten Klassen *„Kondensator“* und *„Spule“* als auch deren Beziehungen mit der abstrakten Klasse *„Resonanz“*.

Listing 1.13 Struktur der abstrakten Basisklasse *Resonanz* mit Hinblick auf abstrakten Methoden *frequenzBerechnen()* und *spannungsBerechnen()*

```
package java4modellierung;
abstract class Resonanz
{

public abstract double frequenzBerechnen();
public abstract double spannungsBerechnen();
}
```

Listing 1.14 Implementierung der Klasse *Kondensator* mithilfe der Eigenschaften der Kenngrößen

```
package java4modellierung;
public class Kondensator extends Resonanz
{
 static double kapazitiverBlindwiderstand=18.45;
 static double kapazitaet=0.000022;
 static double strom=12.46;

 public Kondensator(double kapazitiverBlindwiderstand, double kapazitaet,
 double strom)
 {
  Kondensator. kapazitiverBlindwiderstand =kapazitiverBlindwiderstand;
  Kondensator.kapazitaet=kapazitaet;
  Kondensator.strom=strom;

 }

 public double getKapaziverBlindwiderstand()
 {
  return kapazitiverBlindwiderstand;
 }

 public double getKapazitaet()
 {
  return kapazitaet;

 }

 public double getStrom()
 {
  return strom;

 }

 public void setKapazitiverBlindwiderstand(double kb)
 {
  if(kb > 0)
   Kondensator.kapazitiverBlindwiderstand=kb;
 }
```

```
 public void setKapazitaet(double k)
 {
  if (k > 0)
   Kondensator.kapazitaet=k;
 }

 public void setStrom(double s)
 {
  if(s > 0)
   Kondensator.strom=s;
 }

 public double frequenzBerechnen()
 {
  return 1/(2*Math.PI*kapazitaet*kapazitiverBlindwiderstand);

 }

 public double spannungsBerechnen()
 {
  return strom*kapazitaet;

 }

 public void berechnen4Kondensator()
 {
  System.out.println("Frequenz  für  den  kapazitiven  Blindwiderstand
  ist:" +
  frequenzBerechnen() + " 1/s");
  System.out.println("Spannung für den Kondensator ist:" +
  spannungsBerechnen() + " Volt");
 }
}
```

Listing 1.15 Implementierung der Klasse *Spule* mithilfe der Eigenschaften der Kenngrößen

```
package java4modellierung;
public class Spule extends Resonanz
{
 static double induktiverBlindwiderstand=775;
 static double induktivitaet=0.02;
 static double strom=0.008;
```

```
 public Spule(double induktiverBlindwiderstand, double induktivitaet,
double strom)
 {
  Spule.induktiverBlindwiderstand=induktiverBlindwiderstand;
  Spule.induktivitaet=induktivitaet;
  Spule.strom=strom;

 }

 public double getInduktiverBlindwiderstand()
 {
  return induktiverBlindwiderstand;
 }

 public double getInduktivitaet()
 {
  return induktivitaet;

 }

 public double getStrom()
 {
  return strom;

 }

 public void setInduktiverBlindwiderstand(double ib)
 {
  if(ib > 0)
   Spule.induktiverBlindwiderstand=ib;
 }

 public void setInduktivitaet(double i)
 {
  if (i > 0)
   Spule.induktivitaet=i;
 }

 public void setStrom(double s)
 {
  if(s > 0)
   Spule.strom=s;
 }
```

```
 public double frequenzBerechnen()
 {
  return induktiverBlindwiderstand/(2*Math.PI*induktivitaet);

 }

 public double spannungsBerechnen()
 {
  return induktiverBlindwiderstand*strom;

 }

 public void berechnen4Spule()
 {
  System.out.println("Frequenz für die Spule ist: " + frequenzBerechnen()
  + " 1/s");
  System.out.println("Spannung für die Spule ist: " + spannungsBerechnen()
  + " Volt");
 }
}
```

Listing 1.16 Charakterisieren der Bauelemente *Kondensator* und *Spule* mithilfe des Aufrufes der Methoden *berechnen4spule()* und *berechnen4kondensator()* in der main-Methode der Hauptklasse *Wechselstromkreis*

```
package java4modellierung;
public class Wechselstromkreis
{
 public static void main(String[] args)
 {

  Spule sp= new Spule(775,0.02,0.008);
  sp.berechnen4Spule();
{

  Kondensator ko = new Kondensator(18.45, 0.000022, 12.46);
  ko.berechnen4Kondensator();

 }
}
```

1.4.2.2 Anwendung der Programmierungsperformance in der Berechnung der Kenngröße Wirkungsgrad des Motors

Dieser Abschnitt fokussiert gemäß Listing 1.17, 1.18, 1.19 und 1.20 und Abb. 1.28 auf die Umsetzung der Programmierung von dem Charakterisieren des Motors in die Modellierung der Vererbungshierarchie zwischen verschiedenen Klassen. Hierbei werden die Tools als Hotspot zum Beschreiben der Performance der JVM mittels Implementierungen der Methoden des Interfaces *„IEffizienz4Motor“* zum einen in der abstrakten Klasse *„AEffizienz4Motor“* und zum anderen in der untergeordneten Klasse *„Effizienz4Motor“*. Gemäß Listing 1.17, 1.18 und 1.19 werden vererbte Methoden der abstrakten Basisklasse *„AEffizienz4Motor“* in der Klasse „Effizienz4Motor“ implementiert. Aber die abstrakte Basisklasse delegiert den Aufruf der Methode *„wirkungsgradAnalyse()“* an die Klasse *„Effizienz4Motor“*, die mithilfe der geerbeten Methode *„analysieren()“* aus dem Interface *„IEffizienz4Motor“* die Methode *„wirkungsgradAnalyse“* aufruft und diese zum Berechnen des Wirkungsgrad anwendet. Abb. 1.29 stellt die Modellierung mittels des Ecore-Diagramms dieser Vererbungshierarchie von dem Interface *„IEffizienz4Motor“* über die abstrakte Klasse *„AEffizienz4Motor“* bis zur untergeordnete Klasse *„Effizienz4Motor“* dar. Auf der Abb. 1.28 basiert auf dem Ecore-Diagramm sind die Beziehungen zwischen dem Interface und der Basisklasse zu einen und zwischen der *abstrakten* Klasse und ihrer „Tochter“-Klasse zum anderen zu sehen. Die Beziehung zwischen *„Anwendung4Motor“* und *„Effizienz4Motor“* gemäß Abb. 1.28 stellt eine Referenz über den Aufruf der Methode *„analysieren()“* aus der Klasse Effizienz4Motor in der *„main“*-Methode der Hauptklasse *„Anwendung4motor2“* dar.

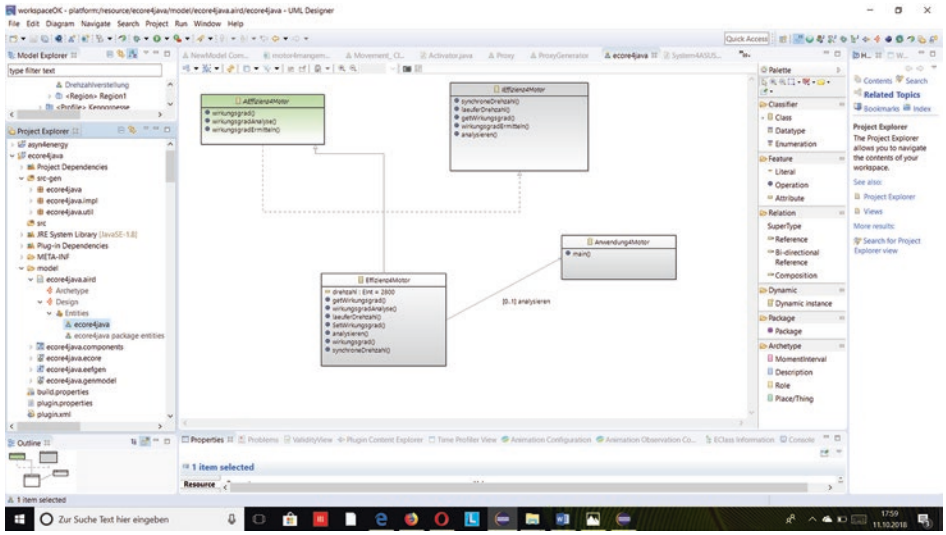


Abb. 1.28 Modellierung der Vererbungshierarchie zwischen verschiedenen Klassen mit dem Ecore-Diagramm

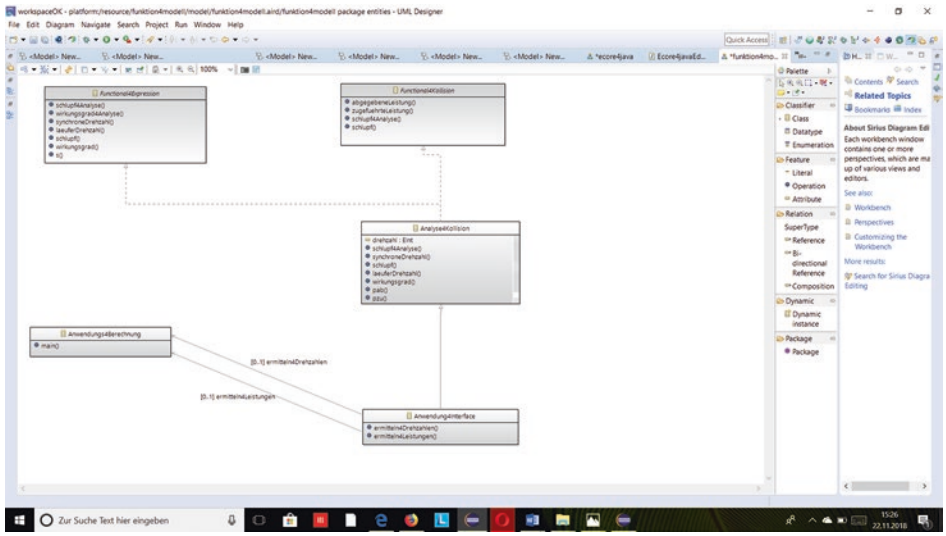


Abb. 1.29 Modellierung der Vererbungshierarchie zwischen Interface und Klassen zur Vermeidung der Kollision zwischen gleichnamigen Methoden

Die grafische Darstellung des Ecore-Diagrammes von Eclipse Modeling Framework (EMF) ähnelt diese des UML-Klassendiagramms von UML-Designer. Die Nutzung vom Ecore-Klassendiagramm ermöglicht die Generierung von Code aus dem Modell.

Listing 1.17 Überblick über das Interface *IEffizienz4Motor* zum Darstellen der implementierten Methode *wirkungsgradErmitteln()*

```
package modeling.motor.performance;
public interface IEffizienz4Motor {

 int synchroneDrehzahl();
 int laeuferDrehzahl ();
 double getWirkungsgrad();

 default void wirkungsgradErmitteln(double wirkungsgrad)
 {

  wirkungsgrad = synchroneDrehzahl()/laeuferDrehzahl();
 }
 void analysieren();
```

Listing 1.18 Implementierung der Methode wirkungsgradErmitteln() in der Abstrakten Klasse *AEffizienz4Motor*

```
package modeling.motor.performance;
public abstract class AEffizienz4Motor implements IEffizienz4Motor{
 abstract double wirkungsgrad();

 abstract void wirkungsgradAnalyse(double wirkungsgrad);
 public void wirkungsgradErmitteln(double wirkungsgrad)
 {
  wirkungsgrad=laeuferDrehzahl()/synchroneDrehzahl();
 }
}
```

Listing 1.19 Überblick über die Vererbungsstruktur in der Klasse *Effizienz4Motor* mithilfe der Implementierungen der Methoden aus dem Interface *IEffzienz4Motor* und der abstrakten Klasse *AEffizienz4Motor*

```
package modeling.motor.performance;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
public class Effizienz4Motor extends AEffizienz4Motor {
 final public int drehzahl=2800;

 @Override
 public double getWirkungsgrad() {

  // TODO Auto-generated method stub
  return wirkungsgrad();
 }

 void wirkungsgradAnalyse(double wirkungsgrad)
 {

  wirkungsgrad= synchroneDrehzahl()/laeuferDrehzahl();
 }

 @Override
 public int laeuferDrehzahl() {
  // TODO Auto-generated method stub
```

```
  return laeuferDrehzahl();
 }
 double SetWirkungsgrad(double wirkungsgrad) {

  // TODO Auto-generated method stub
  return wirkungsgrad();
 }

 public void analysieren() {
  // TODO Auto-generated method stub
  List <Integer> laeuferDrehzahl= Arrays.asList(500, 800, 1100, 1400,
  1700, 2000, 2300, 2600, 2750);
  Consumer<Integer>    wirkungsgrad4analyse    =i    ->    System.out.
  println("Läuferdrehzahl: " + i + " Umdrehungen/min");
  Function<Integer, Double> drehzahl4division = i -> (double)i/2800;

  for(Integer i: laeuferDrehzahl)
  {
   wirkungsgrad4analyse.accept(i);

   System.out.println("Wirkungsgrad des Asynchronmotors ist zu der
   Laeuferdrehzahl proportional. "
     + "Wirkungsgrad:"
     + " " + drehzahl4division.apply(i));
  }

 }
 @Override
 double wirkungsgrad() {
  // TODO Auto-generated method stub
  return synchroneDrehzahl()/laeuferDrehzahl();
 }
 @Override
 public int synchroneDrehzahl() {
  // TODO Auto-generated method stub
  return drehzahl;
 }
}
```

Listing 1.20 Darstellung der Hauptklasse *Anwendung4Motor* zum Ausführen des Programms mithilfe des Aufrufes der Methode *analysieren()* in der *main*-Methode

```
package modeling.motor.performance;
public class Anwendung4Motor {

 public static void main(String[] args) {
  // TODO Auto-generated method stub

  IEffizienz4Motor effizienz= new Effizienz4Motor();

  effizienz.analysieren();

 }

}
```

1.4.2.3 Modellierung der Vermeidung der Kollision zwischen gleichnamigen Methoden aus zwei unterschiedlichen Interfaces

Mit Java 8 sind die Implementierungen der Methoden im Interface mithilfe des Schlüsselwortes „default“ erlaubt. Dies ermöglicht auch die Darstellungen der Code in zwei gleichnamigen Methoden aus zwei unterschiedlichen Interfaces zum Vermeiden eines Konflikts für den Compiler. Es ist auch möglich, zwei gleichnamige Methoden in zwei unterschiedliche Interfaces zu implementieren. Aber dies soll keine Kollision zwischen den Interfaces verursachen.

Dieser Abschnitt stellt mithilfe der Programmierung gemäß Listing 1.21, 1.22, 1.23, 1.24 und 1.25 und der Modellierung gemäß Abb. 1.29 eine Vermeidung der Kollision zwischen gleichnamigen Methoden aus zwei unterschiedlichen Interfaces dar. Mithilfe der Modellierung mithilfe des Ecore-Diagrammes von EMF gemäß Abb. 1.29 sind die Vererbungen zum einen zwischen *„Functional4Expression“* und *„Analyse4Kollision“* und zum anderen zwischen *„Functional4Kollision“* und *„Analyse4Kollision“* mit der Methode *„schlupf4Analyse()“* dargestellt. Die Klasse 2 *„Analyse4Kollision“* dient als Basisklasse für die untergeordnete Klasse *„Anwendung4Interface“*, welche über die Methoden *„ermitteln4Drehzahle()“* und *„ermitteln4Leistungen()“* zum Implementieren die Berechnungen sowohl der Drehzahl als auch die der Leistungen verfügt. Listing 1.24 zeigt die Nutzung der Lambda-Ausdrücke in den Methoden *„ermitteln4Drehzahlen()“* und *„ermitteln4Leistungen()“* zum Ermitteln der Berechnungen der Kenngrößen Drehzahl und Leistung. Die Implementierungen der beiden Methoden in der untergeordneten Klasse *„Anwendung4Interface“* stellt die Vermeidung der Kollision zwischen den beiden Interfaces dar. Dies bedeutet, dass die Kollision nicht stattgefunden hat. Die Beziehung zwischen den Klassen *„Anwendung4Interface“* und

„Anwendungs4Berechnung“ stellt eine Referenz auf die Aufrufe der Methoden *„ermitteln4Drehzahlen()“* und *„ermitteln4Leistungen()“* in der *„main“*-Methode der Hauptklasse *„Anwendungs4Berechnung“* dar.

Listing 1.21 Struktur des Interface *Funtional4Expression* zur Darstellung der Implementierungen der Methode *schlupf4Analyse()*

```
package design.funtional.lamba;
public interface Functional4Expression {
int synchroneDrehzahl();
int laeuferDrehzahl();
double schlupf(double s);
double wirkungsgrad(double N);
 public default void schlupf4Analyse()
 {
  @SuppressWarnings("unused")
  double s = 1- (synchroneDrehzahl()/laeuferDrehzahl());

  }

 @SuppressWarnings("unused")
 default void wirkungsgrad4Analyse()
 {
  double N= 1-s();
 }
 double s();
 double Setschlupf(double s);
 double schlupf();
 int getLaeuferDrehzahl();
 double getWirkungsgrad();
 double pab();

}
```

Listing 1.22 Struktur des Interface Functional4Kollision zur Darstellung der Implementierungen der Methoden

```
package design.funtional.lamba;
public interface Functional4Kollision {

 double schlupf(double s);
 double pab();
 double pzu();
 double getPzu();
 double setPzu();
```

```
 double getPab();
 double setPab();
 double Setschlupf();

default double abgegebeneLeistung(double pab)
{
 return pab;
}
default double zugefuehrteLeistung(double pzu)
{
 return pzu;
}

@SuppressWarnings("unused")
 public default void schlupf4Analyse() {
double s=1-(pab()/pzu());
 }

}
```

Listing 1.23 Darstellung der Klasse *Analyse4Kollision* zur Vermeidung der Kollision zwischen zwei gleichnamigen Methoden aus zwei unterschiedlichen Interface

```
package design.funtional.lamba;
public class Analyse4Kollision implements Functional4Kollision,
Functional4Expression{

public double N() {
 return wirkungsgrad();
}
 @Override
 public double pab() {
  // TODO Auto-generated method stub
  return getPab();
 }
 @Override
 public double getPzu() {
  // TODO Auto-generated method stub
  return pzu();
 }

  public void schlupf4Analyse()
 {
  Functional4Expression.super.schlupf4Analyse();
```

```
 Functional4Kollision.super.schlupf4Analyse();
}
public int synchroneDrehzahl(int drehzahl) {
 // TODO Auto-generated method stub
 return drehzahl;
}
@Override
public int getLaeuferDrehzahl() { // TODO Auto-generated method stub
 return laeuferDrehzahl();
}

@Override
public int synchroneDrehzahl() {
 // TODO Auto-generated method stub
 return getDrehzahl();
}
public int getDrehzahl() {
 // TODO Auto-generated method stub
 return synchroneDrehzahl();
}
@Override
public double pzu() {
 // TODO Auto-generated method stub
 return getPzu();
}

public double schlupf() {
 // TODO Auto-generated method stub
 return Setschlupf();
}
@Override
public double setPzu() {
 // TODO Auto-generated method stub
 return pzu();
} @Override
public double getPab() {
 // TODO Auto-generated method stub
 return pab();
}
@Override
public double setPab() {
 // TODO Auto-generated method stub
 return pab();
}
```

```
 @Override
 public double Setschlupf() {
  // TODO Auto-generated method stub
  return s();
 }

 @Override
 public double Setschlupf(double s) {
  // TODO Auto-generated method stub
  return s();
 }
 @Override
 public double schlupf(double s) {
  // TODO Auto-generated method stub
  return s;
 } @Override
 public double s() {
  // TODO Auto-generated method stub
  return schlupf();
 }
 @Override
 public int laeuferDrehzahl() {
  // TODO Auto-generated method stub
  return getLaeuferDrehzahl();
 }
 public double wirkungsgrad() {
  // TODO Auto-generated method stub
  return getWirkungsgrad();
 }
 @Override
 public double getWirkungsgrad() {
  // TODO Auto-generated method stub
  return N();
 }
 @Override
 public double wirkungsgrad(double N) {
  // TODO Auto-generated method stub
  return N;
 }

}
```

Listing 1.24 Darstellung der Klasse *Anwendung4Interface* zur Implementierung der Vererbung der Vermeidung der Kollision zwischen zwei gleichnamigen Methoden

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
public class Anwendung4Interface extends Analyse4Kollision {

 public void ermitteln4Drehzahlen()
 {
  List <Integer> laeuferDrehzahl= Arrays.asList(500, 800, 1100, 1400,
  1700, 2000, 2300, 2600, 2750);
  Consumer<Integer>     schlupf4Analyse     =i     ->     System.out.
  println("Läuferdrehzahl: " + i + " Umdrehungen/min");
  Function<Integer, Double> s = i -> 1- ((double)i/2800);

  for(Integer i: laeuferDrehzahl)
  {
   schlupf4Analyse.accept(i);

   System.out.println("Je niedriger der Schlupf des Asynchronmotors
   ist, um desto höher ist die Laeuferdrehzahl. "
     + "Schlupf:"
     + " " + s.apply(i));
  }

  System.out.println("-----------------------------------------------
  -------------------"
    );
 }

  public void ermitteln4Leistungen()
 {
  List  <Double>  abLeistung=  Arrays.asList(7000.0,  7200.0,  7300.0,
  7400.0, 7800.0, 7500.0,
    7600.0, 7700.0, 7800.0, 7990.0);
  Consumer<Double> wirkungsgrad4Analyse=i -> System.out.println("Abgegebene
  Leistung: " + i + " Watt");
  Function<Double, Double> wirkungsgrad = i -> ((double)i/7999.95);

  for(Double i: abLeistung)
  {
   wirkungsgrad4Analyse.accept(i);
```

```
    System.out.println("Wirkungsgrad des Asynchronmotors ist zu der
    abgegebenen Leistung proportional. "
      + "Wirkungsgrad:"
      + " " + wirkungsgrad.apply(i));
  }
 }
}
```

Listing 1.25 Darstellung der Hauptklasse *Anwendungs4Berechnung* zum Aufrufen der Methoden *ermitteln4Drehzahlen()* und *ermitteln4Leistungen()*

```
package design.funtional.lamba;
public class Anwendungs4Berechnung {
 public static void main(String[] args)
 {
  // TODO Auto-generated method stub
Anwendung4Interface and = new Anwendung4Interface();
and.ermitteln4Drehzahlen();
and.ermitteln4Leistungen();
 }
}
```

1.4.3 Umsetzung der funktionellen Modellierung in der Programmierung

Dieser Abschnitt befasst sich mit dem Thema Anwendung der Funktionen zum Beschreiben von UML-Diagrammen. Hierbei werden die Funktionen Informationen über sowohl die Strukture als auch das Verhalten eines Systems. Die Funktionalität eines Systems hängt von den Algorithmen zum Laufen der Programme ab. Eine Modellierung eines Systems erzielt die Beschreibung der Elemente dieses Systems mithilfe von Funktionen.

Programmen-Codes aus Basis von C++ oder Java werden zum Beschreiben der Funktionalität der Systeme in der funktionellen Modellierung angewendet. Funktionen werden zum anwenden der Algorithmen in C++ und Java benutzt.

1.4.3.1 Java für die funktionelle Modellierung

Die Programmiersprache Java mit der Version 8 nutzt die Lambda-Ausdrücke zum Darstellen der funktionelle Programmierung. Das Ziel ist es, die Lambda-Funktionen zum Beschreiben der mathematischen Aspekte der Programme mithilfe der Algorithmen anzuwenden. Hierbei werden Programme mithilfe von mathematischen Zielen ohne objektorientierte Gedanken entworfen. Diese Programme sind einfach und verfügt über einfache Kodierung. Das heißt, die Funktionalität der Programme dient der Ersparnis der

Programmierzeit. Mit wenigen Codes werden Funktionen zum Entwickeln funktioneller Einsätze implementiert.

Mithilfe funktioneller Programmierung werden Funktionsobjekte anstatt Funktionstypen in Fokus der Programmierung mit dem Ziel starke Programme zu entwickeln dargestellt.

1.4.3.1.1 Lambda-Ausdrücke zur Modellierung der Berechnungen

Listing 1.26, 1.27, 1.28 und 1.29 geben Überblicke über die Nutzungen der Lambda-Ausdrücke zur Berechnung der Kenngrößen des Asynchronmotors, u. a. Wirkungsgrad und Schlupf. Listing 1.26 zeigt deutlich die Nutzung der Lambda-Ausdrücke in der *„main“*-Funktion zum Berechnen der Wirkungsgrades mithilfe zum einen einer *„Liste“* und zum anderen von zwei parametrisierten Funktion *„Consumer()“* und *„Function()2“*. In der *„main“*-Funktion bestätigt die zweite Funktion die Berechnung der Formel aus der ersten Funktion mithilfe einer *„for“*-Schleife. Hierbei werden den Wirkungsgrad nach der Formel mithilfe der abgegebenen Leistung aus dem Array berechnet. Der Fokus liegt in der Nutzung der Funktionen als Objekte zum Darstellen der mathematischen Operationen. Gemäß Listing 1.27 wird die Funktion *„consumer()“* mithilfe ihres Objektes genannt *„wirkungsgrad4Analyse“* den Zugriff auf die Elemente i der Liste *„abLeistung“* implementieren. Hierbei stellt das Element *„i“* die abgegebene Leistung dar. Anschließend wird die Funktion *„Function()“* mithilfe ihres Objektes *„wirkungsgrad“* und des Elements i der Liste durch die Formel zum Berechnen des Wirkungsgrads implementiert. Die Funktionalität zur Berechnung des Wirkungsgrades findet sich in der *„for“*-Schleife der *„main“*-Funktion zum Aufrufen der Methoden *„accept()“* und *„apply()“* der funktionalen Interfaces *„Consumer“* bzw. *„Function“* mithilfe der Zugriffe auf die Funktionsobjekte *„wirkungsgrad4Analyse“* und *„wirkungsgrad“* statt. Die Zugriffe auf diese Funktionsobjekte erfolgen mithilfe der Verwendung des *„Punkt-Operator“*. Die Aufrufe der Methoden *„accept()“* und *„apply()“* sind dank der Importe von den Packages *„**import** java.util.function.Consumer“* bzw. *„**import** java.util.function.Function“* zu den Definitionen der funktionalen Interfaces *„Consumer“* bzw. *„Function“* sinnvoll und erlaubt. Gemäß Listing 1.26 sind die Zugriffe auf die Elemente i der Liste *„abLeistung“* mithilfe der Zugriffe auf die Funktionsobjekte *„wirkungsgrad4Analyse“* und *„wirkungsgrad“* ermöglicht.

Die Modellierung der Berechnung des Wirkungsgrades mithilfe der *„main“*-Funktion stellt die Modellierung der Funktionen mithilfe der Funktionsargumente einerseits und mithilfe der Funkltionswerte andererseits dar. Die *„main“*-Funktion fokussiert auf das Eingabe-Verarbeitungs-Ausgabe-(EVA-)Prozess zum Modellieren der Berechnung des Wirkungsgrades. Hierbei sind die Eingabe und Ausgabe des Prozesses die Funktionsargumente bzw. -werte zum Modellieren der Relationen zwischen Definitions- und Wertemengen.

Listing 1.27 zeigt die Ausgabe des Prozesses zum Modellieren der Berechnung des Wirkungsgrades. Gemäß Listing 1.27 sind mithilfe der Eingaben *„Leistung“*

die Berechnung des Wertes der Funktion zu den Ausgaben ermöglicht. Die Logik stellt die Nutzung der Lambda-Ausdrücke mithilfe der parametrisierten Funktionen „*Consumer()*“ und „*Function()*“ in der „*main*“-Funktion zum Aufrufen der funktionalen Methode „*accept()*“ bzw. „*apply()*“ dar. Von daher ist zu bemerken, das die main-Funktion ihre Funktionalität an die funktionale Methoden „*accept()*“ und „*apply()*“ mittels Aufrufes und Argumentübergabe delegiert. „*main()*“ ermöglicht den funktionalen Methoden „*accept()*“ und „*apply()*“ das Zugriffrecht auf die Funktionsobjekte „*wirkungsgrad4Analyse*“ bzw. „*Wirkungsgrad*“.

Listing 1.27 stellt die Proportionalität zwischen der abgegebenen Leistung und dem Wirkungsgrad dar. Hierbei ist die Ausgabe zu der Eingabe proportional, d. h., je höher die abgegebene Leistung ist, um desto höher ist der Wirkungsgrad. Wenn das Funktionsargument steigt, wird der Funktionswert auch steigen. Das Listing 1.27 zeigt auch die Modellierung der Proportionalität zwischen der abgegebenen Leistung und dem Wirkungsgrad.

Listing 1.28 zeigt auch die Modellierung der Berechnung mithilfe der Lambda-Ausdrücke bezüglich des Aufrufes der Funktion „*analyse4Motor()*“ in der „*main*“-Funktion. Hierbei werden die funktionale Methoden „*accept()*“ und „*apply()*“ in der privat-deklarierten Funktion „*schlupf4Analyse()*“ aufgerufen. Die Logik der Anwendung der Lambda-Ausdrücke findet in „*schlupf4analyse()*“ mithilfe der Implementierung der funktionellen Interfaces „*Consumer*“ und „*Function*“ wie im Listing 1.26 zum Ermöglichen der Berechnungen statt. Der Unterschied zwischen dem Listing 1.26 und 1.27 liegt in dem Bereich des Aufrufes der funktionalen Methoden „*accept()*“ und „*apply()*“. Im ersten Listing werden diese funktionalen Methoden direkt in „*main()*“ aufgerufen, während sie im zweiten in der Funktion „*schlupf4Analyse()*“ aufgerufen sind. Der Aufruf von „*schlupf4Analyse()*“ erfolgt, nachdem ein Klassenobjekt von „*Effizienz*“, genannt „*eff*“, mithilfe des „*new*“-Operators und ein Zugriff auf dieses Objekt erzeugt wurden. Das Zugriffrecht der Funktion „*schlupf4Analyse()*“ auf das Klassenobjekt „*eff*“ ist erlaubt. Wie im Listing 1.26 werden Berechnungen mithilfe einer Formel realisiert. Die Formel gemäß Listing 1.28 ermöglich die Berechnung des Motorschlupfes mithilfe der Läuferdrehzahlen als Eingaben. Die Funktionalität, zur Berechnung des Schlupfes gemäß Listing 1.28, fokussiert wie im Listing 1.26 auf das Eingabe-Verarbeitung-Ausgabe-Prinzip. Hierbei stellen einerseits die Eingaben die Liste aus Drehzahlen und andererseits die Ausgabe die Anzeige der Berechnung des Schlupfes dar. Die Verarbeitung ist der Prozess zum Berechnen des Funktionswertes. Im Vergleich zu der Funktionalität der Proportionalität im Listing 1.26 wird die Berechnung des Schlupfes im Listing 1.28 nicht proportionell. D. h., je niedriger die Läuferdrehzahl ist, desto höher ist der Schlupf. Oder je niedriger der Schlupf ist, desto höher ist die Läuferdrehzahl. Listing 1.29 zeigt die Ausgabe zur Berechnung des Schlupfes aus den Eingaben an.

Das Ziel der Modellierung der Berechnung des Schlupfes gemäß Listing 1.28 und 1.29 ist es, die Funktionalität zu Anti-Proportionalität zwischen den Eingaben und den Ausgaben darzustellen.

Listing 1.26 Darstellung der Lambda-Funktion zur Modellierung der Berechnung des Wirkungsgrades

```
package funktion4modell;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
public class Motor4Funktion {
 public static void main(String[] args)
   {

   List <Double> abLeistung= Arrays.asList(7000.0, 7200.0, 7300.0,
   7400.0, 7800.0, 7500.0,
     7600.0, 7700.0, 7800.0, 7990.0);
   Consumer<Double>     wirkungsgrad4Analyse=i     ->     System.out.
   println("Abgegebene Leistung: " + i + " Watt");
   Function<Double, Double> wirkungsgrad = i -> ((double)i/7999.95);

   for(Double i: abLeistung)
   {
    wirkungsgrad4Analyse.accept(i);

    System.out.println("Wirkungsgrad des Asynchronmotors ist zu der
    abgegebenen Leistung proportional. "
      + "Wirkungsgrad:"
      + " " + wirkungsgrad.apply(i));
   }

  // TODO Auto-generated method stub
 }
}
```

Listing 1.27 Konsole-Anzeige von der Java-Anwendung zum Listing 1.26

```
Abgegebene Leistung: 7000.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.87500546878418
Abgegebene Leistung: 7200.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9000056250351565
Abgegebene Leistung: 7300.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9125057031606447
```

```
Abgegebene Leistung: 7400.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9250057812861331
Abgegebene Leistung: 7500.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9375058594116213
Abgegebene Leistung: 7600.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9500059375371096
Abgegebene Leistung: 7700.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9625060156625979
Abgegebene Leistung: 7800.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9750060937880862
Abgegebene Leistung: 7990.0 Watt
Wirkungsgrad des Asynchronmotors ist zu der abgegebenen Leistung
proportional. Wirkungsgrad: 0.9987562422265139
```

Listing 1.28 Darstellung der Lambda-Funktion zur Analyse der Effizienz des Asynchronmotors

```
package funktion4modell;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
public class Effizienz {

 private void analyse4Motor()
 {
  List <Integer> laeuferDrehzahl= Arrays.asList(500, 800, 1100, 1400,
  1700, 2000, 2300, 2600, 2750);
  Consumer<Integer>     schlupf4Analyse     =i     ->     System.out.
  println("Läuferdrehzahl: " + i + " Umdrehungen/min");
  Function<Integer, Double> schlupf = i -> 1- ((double)i/2800);

  for(Integer i: laeuferDrehzahl)
  {
   schlupf4Analyse.accept(i);

   System.out.println("Je niedriger der Schlupf des Asynchronmotors
   ist, um desto höher ist die Laeuferdrehzahl. "
     + "Schlupf:"
     + " " + schlupf.apply(i));
  }
```

```
  System.out.println("-----------------------------------------------
  --------------------"
    );

 }
 public static void main(String[] args) {
  // TODO Auto-generated method stub
Effizienz eff = new Effizienz();
eff.analyse4Motor();

 }
}
```

Listing 1.29 Konsole-Anzeige von der Java-Anwendung zum Listing 1.28

```
Läuferdrehzahl: 500 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.8214285714285714
Läuferdrehzahl: 800 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.7142857142857143
Läuferdrehzahl: 1100 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.6071428571428572
Läuferdrehzahl: 1400 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.5
Läuferdrehzahl: 1700 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.3928571428571429
Läuferdrehzahl: 2000 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.2857142857142857
Läuferdrehzahl: 2300 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.1785714285714286
Läuferdrehzahl: 2600 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.0714285714285714
Läuferdrehzahl: 2750 Umdrehungen/min
Je niedriger der Schlupf des Asynchronmotors ist, um desto höher ist
die Laeuferdrehzahl. Schlupf: 0.017857142857142905
-----------------------------------------------------------------
```

1.4.3.1.2 Modellierung der funktionellen Berechnung mithilfe von konstanten Eingaben

Modellierung der Funktionalität einer Berechnung stellt einerseits die Analyse der Parameter der Funktion bezüglich des Datentyps und andererseits die Bedeutung des Ausdrucks zum Funktionsaufruf dar. Dieser Abschnitt befasst sich mit der Anwendung der konstanten Variable mithilfe des Schlüsselwort „*final*“. Hierbei werden die Berechnungen der Kenngrößen der Bauelemente Spule und Kondensator anhand Implementierungen der Funktionen dargestellt. Die Eingaben sind final deklariert, damit sie nicht mehr in der Verarbeitung verändert werden. Listing 1.30 zeigt die Funktionalität zur Berechnung der Kenngrößen der Bauelemente Spule und Kondensator mithilfe der Implementierungen der Funktionen. Listing 1.30 verfügt einerseits über vier Funktionen mit den Rückgabenwerten „*frequenzBerechnen4Spule()*“, „*spannungsBerechnen4Spule()*“, „*frequenzBerechnen4Kondensator()*“ oder „*spannungsBerechnen4kondensator()*“ und andererseits über zwei Funktionen ohne Rückgabewerten „*berechnen4Spule()*“ und „*berechnen4Kondensator()*“ zum Aufruf der Funktionen mit Rückgabenwerten. Die Hauptfunktion „*main()*“ ermöglicht die Aufrufe der Funktionen „*berechnen4Spule()*“ und „*berechnen4Kondensator()*“ mithilfe der Zugriffe auf Objekte „*spule*“ bzw. „*kondensator*“ der Klasse „*Elemente4Funktion*“. Die Aufrufe der beiden letzten Funktionen führen zu den Aufrufen der Funktionen „*frequenzBerechnen4Spule()*“, „*spannungsBerechnen4Spule()*“, „*frequenzBerechnen4Kondensator()*“ und „*spannungsBerechnen4kondensator()*“.

Die Logik zur Berechnung der Kenngrößen basiert auf dem Aufruf einer Funktion mit Rückgabenwerten in einer Funktion ohne Rückgabenwerte. Dies ermöglicht die Nutzung der konstanten Eingaben in der Funktion mit Rückgabenwerten mithilfe einer mathematischen Formel. Wobei die Rückgabenwerte dieser Funktionen die final deklarierten Eingaben wie z. B. „*kapazitaet*“und „*induktivitaet*“ darstellen.

Die Anzeige der Berechnung mithilfe der Aufrufe der Funktionen „*berechnen4Spule()*“ und „*berechnen4kondensator()*“ in „*main()*“ stellt das Listing 1.31 dar. Gemäß Listing 1.31 werden die Bauelemente mithilfe der Berechnungen der Kenngrößen „*Frequenz*“ und „*Spannung*“ charakterisiert.

Das Konzept Eingabe-Verarbeitung-Anzeige wird mithilfe der Aufrufe der Funktionen zum Berechnen der Kenngrößen aus den konstanten Variablen in „*main()*“ ermöglicht.

Listing 1.30 Darstellung der Funktionen zur Berechnung der Kenngrösse der Bauelemente Spule und Kondensator

```
package funktion4modell;
public class Elemente4Funktion {

  final double induktiverBlindwiderstand = 775;
  final double induktivitaet = 0.02;
  final double kapazitiverBlindwiderstand = 18.45;
```

```
 final double kapazitaet=0.000022;
 final double strom4spule = 0.008;
 final double strom4kondensator = 12.46;
public double frequenzBerechnen4Spule()
{

 return induktiverBlindwiderstand/(2*Math.PI*induktivitaet);
}

public double spannungsBerechnen4Spule()
{

 return induktiverBlindwiderstand*strom4spule;
}

public double frequenzBerechnen4Kondensator()
{

 return 1/(2*Math.PI*kapazitaet*kapazitiverBlindwiderstand);
}

public double spannungsBerechnen4Kondensator()
{

 return strom4kondensator*kapazitaet;
}
public void berechnen4Spule()

{
 System.out.println("Charakterisieren der Spule:" + "\n");
 System.out.println("Frequenz    für    die    Spule    ist:    "    +
 frequenzBerechnen4Spule() + " 1/s");
 System.out.println("Spannung    für    die    Spule    ist:    "    +
 spannungsBerechnen4Spule() + " Volt");
}

public void berechnen4Kondensator()

{
 System.out.println("-----------------------------------------------
 ----------");
```

```
  System.out.println("Charakterisieren des Kondensators:" + "\n");
  System.out.println("Frequenz für den kapazitiven Blindwiderstand
  ist:" +
  frequenzBerechnen4Kondensator() + " 1/s");
  System.out.println("Spannung für den Kondensator ist:" +
  spannungsBerechnen4Kondensator() + " Volt");
 }
 public static void main(String[] args) {
  // TODO Auto-generated method stub

  Elemente4Funktion spule= new Elemente4Funktion();
  spule.berechnen4Spule();

  Elemente4Funktion kondensator = new
    Elemente4Funktion();
  kondensator.berechnen4Kondensator();

 }
}
```

Listing 1.31 Konsole-Anzeige von der Java-Anwendung zum Listing 1.30

```
Charakterisieren der Spule:
Frequenz für die Spule ist: 6167.254044810944 1/s
Spannung für die Spule ist: 6.2 Volt
-------------------------------------------------------
Charakterisieren des Kondensators:
Frequenz für den kapazitiven Blindwiderstand ist:392.10382629193236
1/s
Spannung für den Kondensator ist:2.7412000000000004E-4 Volt
```

1.4.3.2 C++ für die funktionelle Modellierung

Funktionelle Modellierung wird mithilfe der Analyse der Implementierung der Werkzeuge der C++-Standardbibliothek bezüglich der Themen wie z. B. Ein-/Ausgabe-, String-, Container- oder Iterator-Bibliothek realisiert.

Container-Bibliothek wie z. B. *„array“*, *„vector“*, *„set“* oder *„map“* ermöglichen das Speichern gleichartiger Elemente. Iteratoren verweisen auf Elemente von Containern. Mithilfe eines Iterators wird es auf ein Element zugegriffen. Das Sortieren, oder Finden von Elementen im Container wird mithilfe der Algorithmen-Bibliothek realisiert.

1.4.3.2.1 Modellierung der Entfernung eines gezielten Elements im Vektor

Ziel der Modellierung der Funktionalität der C++-Standard-bibliothek ist es, die Funktionen mithilfe der Iteratoren und Algorithmen zu implementieren. Hierbei werden

gemäß Listing 1.32 die gewählten Elemente eines ersten Vektors mithilfe von der Funktion *„copy()“* in einen zweiten Vektor kopiert. Außerdem wird ein ungewünschtes Element in dem zweiten Vektor mit der Funktion *„is_event“* entfernt. Die parametrisierte Funktion *„vector()“* ermöglicht das Kopieren eines Vektors ohne ein ungewünschtes Element. Die Funktion *„output“* wird mithilfe des Typs *„ostream-iterator“* Zeichen darstellen. Die Funktion *„event_count“* ermöglicht gemäß Listing 1.32 mithilfe des Überladens der Funktionen *„if_count“* und *„is_event“* das Prüfen des Datentyps der Größe einer Zahl im Vektor. Die Nutzung des Schlüsselwortes *„auto“* ermöglicht die Funktion *ist event* und die Entfernung eines ungewünschten Elements in dem zweiten Vektor mithilfe einer Formel als Rückgabewert. Diese Funktion gibt eine Prüfung als Wert zurück. Hierbei wird geprüft, welche Zahl in der Formel nicht im zweiten Vektor angezeigt wird. Gemäß Listing 1.33 ist 2850 die ungewünschte Zahl und stellt die Läuferdrehzahl dar. Die Modellierung der Anzeige eines Vektors aus Drehzahlen mithilfe der C++-Standardbibliothek ermöglicht das Erstellen von Vektoren mit verschiedenen Elementen.

Listing 1.32 Modellierung der Bewegung eines Elements im Vektor

```
// Effizienz4Wirkungsgrad.cpp: Diese Datei enthält die Funktion
"main". Hier beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
int main()
{
// Einen Vektor aus den Drehzahlen d.h. Läuferdrehzahl und
synchronedrehzahl eines Motors mit der Integer-Größe erstellen
 std::vector<int> seq1 { 700, 900, 1500, 1800, 2000,2200, 2400, 2500,
 2650, 2750, 2850 };
 auto output = std::ostream_iterator<int>(std::cout, " ");

 // Die Sequenz seq1 aus den Drehzahlen anzeigen
 std::cout << "Laeufer- und Drehfelddrehzahl in 1/min: " << '\n';
 std::copy(std::begin(seq1), std::end(seq1), output);
 std::cout << '\n';
 // Diese Funktion testet die Selbsverständlichkeit der synchronen
Drehzahl
 auto is_even = [](int n)
 {
return (n/2850 !=1);
```

```
 };
 // Größe der Zahlen von Integer prüfen
 int even_count = count_if(std::begin(seq1),
  std::end(seq1), is_even);
 // Erstellen einer Kopie vom Vektor ohne synchrone Drehzahl
 std::vector<double> seq2(even_count);
 std::copy_if(std::begin(seq1), std::end(seq1), std::begin(seq2), is_
even);
 // Die Sequenz seq2 aus Läuferdrehzahlen anzeigen
 std::cout << " Laeuferdrehzahl in 1/min: ";
 std::cout << '\n';
 copy(std::begin(seq2), std::end(seq2), output);
 std::cout << '\n';

}
```

Listing 1.33 Anzeige der Kopie von einem Vektor aus einem vorhandenen Vektor

```
Laeufer- und Drehfelddrehzahl in 1/min:
700 900 1500 1800 2000 2200 2400 2500 2650 2750 2850
Laeuferdrehzahlen in 1/min:
700 900 1500 1800 2000 2200 2400 2500 2650 2750
C/Users/eric/source/repos/Effizienz4Wirkungsgrad/Debug/
Effizienz4Wirkungsgrad.exe (Prozess "28552") wurde mit Code "0"
beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.4.3.2.2 C++-Standardbibliothek mit Funktions-Template zum Modellieren der Funktionalität des Wirkungsgrades

Ziel der Modellierung mit Funktions-Template ist es, die Anwendung zur Berechnung des Wirkungsgrades mithilfe des Platzhalters bezüglich der generischen Programmierung zu ermöglichen. Hierbei werden Datentype als Parameter eingesetzt. Dies definiert einen Platzhalter für den Datentyp. Listing 1.34 zeigt die Anwendung des Funktions-Templates zur Berechnung des Wirkungsgrades. Die Funktion *„wirkungsgrad(laeuferdrehzahl“*, *„synchronedrehzahl)“* gibt die Formel zur Berechnung des Wirkungsgrades als Verhältnis zwischen der Läuferdrehzahl und der synchronen Drehzahl zurück. Gemäß Listing 1.34 ist *„T“* der Platzhalter für den Datentyp. Die Argumente *„laeuferdrehzahl“* und *„sychronedrehzahl“* werden als Referenz auf *„const“* zum Vermeiden jeglicher Veränderung in der Funktion *„wirkungsgrad()“* übergeben. Diese Funktion funktioniert

nicht nur für ganze Zahlen, sondern auch für Gleitkommazahlen (*„float“* und *„double“*) und ermöglicht die Nutzung des Templates zum Gestalten des Datentyps. Der Aufruf der Funktion *„wirkungsgrad(laeuferdrehzahl, synchronedrehzahl)“* in der *„main()“* führt zum Berechnen des Wirkungsgrades mithilfe der Parameter. Listing 1.35 zeigt die Konsole zur Darstellung der Ausgabe der Berechnung mithilfe der Eingaben an.

Das Prinzip Eingabe-Verarbeitung-Ausgabe wird mithilfe des Template erleichtert, weil die Eingaben als Parameter der Funktion in *„main()“* erscheinen. Die Ausgabe der Funktion stellt der Aufruf von *„main()“* dar. Die Verarbeitung der Funktion stellt die Rückgabe von *„wirkungsgrad(laeuferddrehzahl, synchronedrehzahl)“* als Formel zur Berechnung des Wirkungsgrades dar.

Listing 1.34 Anwendung des Funktions-Template zur Berechnung des Wirkungsgrades

```
include "pch.h"
#include<iostream>
template <typename T>
T wirkungsgrad(const T& laeuferdrehzahl, const T& synchronedrehzahl)
{
 return laeuferdrehzahl / synchronedrehzahl;
}
int main()
{
std::cout << "Berechnung des Wirkungsgrades mithilfe der
Laeuferdrehzahl und der synchronen Drehzahl" << '\n';
 std::cout << '\n';
 std::cout <<"Laeuferdrehzahl:2850 1/min; synchrone Drehzahl: 2900 1/
min" << '\n';
 std::cout << "Wirkungsrad als Verhaltnis zwischen Laeuferdrehzahl und
synchronen Drehzahl ist:" << '\n';
 std::cout << wirkungsgrad<double>(2850, 2900) << '\n';
 std::cout << "-----------------------------------------------------
---------------" << '\n';
 std::cout << "Laeuferdrehzahl:2750 1/min; synchrone Drehzahl: 2800 1/
min" << '\n';
 std::cout << "Wirkungsrad als Verhaltnis zwischen Laeuferdrehzahl und
synchronen Drehzahl ist:" << '\n';
 std::cout << wirkungsgrad<double>(2750, 2800) << '\n';
```

Listing 1.35 Darstellung der Konsole zur Anzeige der Berechnung des Wirkungsgrades

```
Berechnung des Wirkungsgrades mithilfe der Laeuferdrehzahl und der
synchronen Drehzahl
```

```
Laeuferdrehzahl:2850 1/min; synchrone Drehzahl: 2900 1/min
Wirkungsgrad als Verhaltnis zwischen Laeuferdrehzahl und synchronen
Drehzahl ist:
0.982759
---------------------------------------------------------------------
Laeuferdrehzahl:2750 1/min; synchrone Drehzahl: 2800 1/min
Wirkungsgrad als Verhaltnis zwischen Laeuferdrehzahl und synchronen
Drehzahl ist:
0.982143
C:\Users\eric\source\repos\ConsoleApplication4Wirkungsgrad\Debug\
ConsoleApplication4Wirkungsgrad.exe (Prozess "10664") wurde mit Code
"0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.4.3.2.3 „*C++-Standardbibliokthek*“ mit Funktions-„*Template*“, Iteratoren und Überladen der „*Operatoren*“ zum Modellieren der Orthogonalität zwischen zwei Vektoren

Die Parametrisierung einer Funktion ermöglicht ihre Spezifizierung mit dergleichen Namen für verschiedene Argumenttype. Dies führt zum Überladen der Funktionsnamen. Mithilfe des Funktionsaufrufes einer parametrisierten Funktion werden die explizite Spezifizierung der Argumente für die „*template*“-Parameter dargestellt. Gemäß Listing 1.36 wird die Funktion „*operator<<()*“ mithilfe der Spezifizierung der Argumente „*os*“ und „*s*“ zur „*template*“-Funktion umgewandelt. „*Operator<<()*“ verfügt über zwei Argumente. Hierbei stellt dieser „*template*“-Name eine Liste von Typnamen mithilfe von eingeschlossenen „*template*“-Argumenten in <> qualifizierte Elementname wie z. B. „*set*“-Container dar.

Ziel der Anwendung des Stream-Konzeptes mithilfe des Objektes der Klasse „*ostream*“ in der Parametrisierung der Funktion ist es, das Überladen des Ausgabeoperators „<<“ für benutzerdefinierte Datentype zu ermöglichen. Gemäß Listing 1.36 ist der Rückgabewert „*os*“ als Referenz auf das als erstes Argument übergebene „*ostream*“-Objekt deklariert. Dies ermöglicht das Zusammenfassen mehrerer Ausgaben in einer Anweisung. Der zweite Parameter der Funktion „*operator<<()*“ ist vom Typ „*const set<> &*“. Hierbei wird das auszugebende Objekt der „*template*“-Klasse „*Set*“ bei der Ausgabe nicht verändert. „*Set*“ verfügt über zwei parametrisierte Funktionen „*begin()*“ und „*end()*“ einerseits zum Beginnen andererseits zum Beenden der Iteratoren. Gemäß Listing 1.36 ist der Funktionsaufruf.

„*os << *iter++*“ ist ausführbar, weil der Ausgabeoperator „<<“ für den Typ der Variable „**iter++*“ bereits deklariert ist. Es ist zu bemerken, dass die Funktion „*operator<<()*“ nicht als Elementfunktion von der template-Klasse *set<>* definierte

ist. Gemäß des C++-Sprachstandards ist die Stream-Bibliothek wie z. B. die Klasse *„ostream“* mithilfe von *„std::ostream“* in den Namensbereich std aufgenommen.

Listing 1.37 stellt die cpp-Datei zum Darstellen der Modellierung der Orthogonalität zwischen zwei Vektoren mithilfe von Funktions-*„Template“* und Überladungen der *„Operator“*-Funktionen dar. Hierbei besteht die Funktion *„operator &()“* aus zwei template-Parametern von dem Typ set<>. *„Operator &()“* stellt die Implementierung von *„set“*-Container zum Modellieren der Überschneidung zwischen den Sequenzen s_1 und s_2 dar. Gemäß Listing 1.37 wird jeder Vektor zuerst sortiert und anschließend wird er zur Standard-Funktion für die Überschneidung gesendet. Gemäß Listing 1.37 bestehen *„s1“* und *„s2“* aus Läuferdrehzahlen bzw. synchronen Drehzahlen. Die Überschneidung ermöglicht die Modellierung der Orthogonalität zwischen zwei Vektoren mithilfe der Anwendung der Funktion *„operator&()“* in der Parametrisierung von *„set“*-Containern. Hierbei wird die Funktion *„operator&()“* mithilfe einerseits der Funktion *„set_intersection()“* und andererseits der Funktion *„set_inserter()“* überladen. Der Zugriff auf das set-Objekt stellt die Überladung der Funktion *„operator &()“* dar. Gemäß Listing 1.37 ermöglicht der Zugriff auf das *„set“*-Objekt *„result“* das Hinzufügen der Elemente nach dessen Sortieren in den *„set“*-Vektor.

Aufrufe der *„set“*-Objekte in der *„main“*-Funktion gemäß Listing 1.37 rufen die Funktion *„operator&()“* mithilfe der Funktion *„operator<<()“* auf. Hierbei gibt operator&() *„result“* zum Ermöglichen einer Überschneidung zwischen den Set-Elementen zurück. Der Aufruf der Funktion zur Überschneidung zwischen zwei set-Containern erfolgt zuerst mithilfe des Aufrufes der Funktion *„operator<<()“* und anschließend mithilfe des Aufrufes von *„operator&()“*. Diese Überschneidung wird mithilfe der Funktion *„std::inserter()“* realisiert. *„Std::inserter()“* verfügt über zwei Parameter. Der erste ist das *„set“*-Objekt *result* und der Zweite ist die *„set“*-Funktion *„std::end()“*. Der Zugriff auf das set-Objekt result wird mithilfe der set-Funktion *„std::end()“* ermöglicht.

Listing 1.36 Header-Datei zum Darstellen der Parametrisierung mit der Funktion *operator<<()*

```
#pragma once
#ifndef DREHZAHL4SET_H_DEFINED
#define DREHZAHL4SET_H_DEFINED
#include <iostream>
#include <set>
// Eine Gruppe mit Elementen anzeigen
template <typename T>
std::ostream& operator<<(std::ostream& os, const std::set<T>& s)
{
 os << '{';
auto iter = std::begin(s);
```

```
 auto done = std::end(s);
 if (iter != done)
{
 os << *iter++;
 while (iter != done)
os << ", " << *iter++;
}
 os << '}';
 return os;
}
#endif
```

Listing 1.37 Cpp-Datei zum Darstellen der Modellierung der Orthogonalität zwischen zwei Vektoren mithilfe von Funktions-*Template* und *Operator*-Funktionen

```
#include <iostream>
#include <set>
#include <algorithm>
#include <iterator>
# include "Drehzahl4Set.h"
// Die Überschneidung zwischen den Sequenzen s1 und berechnen
template<typename T>
std::set<T> operator&(const std::set<T>& s1, const std::set<T>& s2)
{
std::set<T> result;
std::set_intersection(std::begin(s1),  std::end(s1),  std::begin(s2),
std::end(s2), std::inserter(result, std::end(result)));
return result;
}
int main()
{
 std::set<int> s1{ 700, 800, 850, 900, 1500, 1750, 1800, 1900, 2500,
 2750, 2800,2850 };
 std::cout << " Eine Gruppe s1 bestehend aus Laeuferdrehzahlen : ";
 std::cout << '\n';
 std::set<int> s2{ 1600, 1750, 1800, 1850, 1950, 2000, 2500, 2650,
 2750, 2800, 2880};
 std::cout << " Eine Gruppe s2 bestehend aus synchroner Drehzahlen : ";
 std::cout << '\n';
 std::cout << s1 << " & " << s2 << " = " << (s1 & s2) << '\n';
 std::cout << " Der Schnitt zwischen den Gruppen s1 und s2 bestehend
aus Laeuferdrehzahlen und synchronen Drehzahlen : ";
 std::cout << '\n';
```

1.4.3.2.4 Lambda-Funktion zum Darstellen der Derivation einer Funktion

Im Bereich funktionale Modellierung ist die Anwendungen der Lambda-Ausdrücke zur Berechnung der mathematischen Formel in Java sehr hilfsreich, weil diese Berechnungen mithilfe von Funktionen durchgeführt werden. Listing 1.38 zeigt die Anwendung der Lambda-Ausdrücke zum Modellieren der Berechnungen des Derivates mithilfe von zwei Parametern: der generischen Funktion *f* und dem Wert *h*. Diese generische Funktion *f* ist ein Pointer und besteht aus einem Parameter von dem Typ *double.*

Die Funktion f ist im Folgenden dargestellt:

$f(x) = 1-x$ mit x: Wirkungsgrad und f(x): Schlupf
$f'(x) = -1\, f'$ ist das Derivativ von f

Ziel dieser Modellierung ist es, die Funktionalität des Schlupfes mithilfe des Wirkungsgrades zu analysieren. Die Anwendungen der Derivation in der Modellierung der Kenngröße *Schlupf* stellen die Annäherung der Werte dar. Gemäß Listing 1.38 ist die generische Funktion std::function<> berechnet die Derivation einer anderen Funktion in C++. Wobei die derivative Funktion eine Lambda-Funktion zurückgibt. Die Funktion, die die derivative Funktion zurück, stellt eine Closure dar. Hierbei greift diese Closure-Funktion auf die Parameter *f* und *h* zu.

Die Funktion *„derivative()“* besteht aus zwei Parametern: der generischen Funktion *f* und dem Wert *h*. Diese Parameter ermöglichen die Darstellung der Lambda-Funktion, weil *„derivative()“* Zugriff auf die Funktion *f* und den Wert *h* und anschließend die Bildung einer Lambda-Ausdrüe mithilfe von x mit dem Typ double ermöglicht. Die Funktion *„schlupf4Analyse()“* bestehend aus dem Parameter x vom Typ *„double“ und* stellt einerseits die Formel des Schlupfes und andererseits die gewünschte Funktion zum Differenzieren dar. Der Wert *„ans* $= -1.0$*“* stellt das bekannte Derivativ der Funktion *„schlupf4Analyse()“* dar. Hierbei ist zu bemerken, dass die Pointer-Funktion *„derivative()“* der Variable x zugewiesen ist und der Funktion *f* als Parameter übergibt oder von *f* als Ergebnis zur Bestimmung der Derivation-Formel zurückgibt.

Die *„main“*-Funktion stellt die Funktionalität der Darstellung der Lambda-Ausdrücke mithilfe der Aufrufe zum einen der Funktion *„schlupf4Analyse()“* und zum anderen der Lambda-Funktion *„der(x)“* dar. Gemäß Listing 1.38 stellt die Funktion *„der(x)“* mithilfe des Ausdrückes *„auto = der derivative(schluf4Analyse, h)“* eine Lambda-Funktion zum Zugriff auf die Parameter *„schlupf4Analyse“* und *„h“* dar. Hierbei gibt die Funktion *„der(x)“* den Wert des Derivates der Funktion *„schlupf4analyse()“ zurück.* Von daher hat die Funktion *„der(x)“* den Zugang zur Funktion *„schlupf4Analyse()“*. In *„main()“* gibt es eine Kaskade von Anweisungen mithilfe der *„while“*-Schleife, welche aus dem Anfangswert $x = 0.97$ mit dem Typ double die Bedingung sowohl auf *„true“* als auch auf *„falsch“* geprüft wird. Die Anweisungen in der while-Schleife stellen sowohl die Aufrufe der Funktionen *„schlupf4Analyse()“* und *„der(x)“* als auch den Zugriff auf die Werte *„x“*, *„h“* und *„ans“* dar. Das Ausführen der Anweisungen in *„main()“* stellt auch die Nutzung der Lambda-Funktion *„der(x)“* zur Bildung der

Derivation mithilfe der Funktion „*schlupf4Analyse()*“ und dem Annäherungswert „*h*“ dar. Mithilfe der Parameter „*schlupf4Analyse()*“ und „*h*“ ruft die Funktion „*der(x)*“ die Funktion „*derivation()*“ auf. Listing 1.39 zeigt die Konsole der Ausgabe des Programms aus dem Listing 1.38 zum Bestimmen der Derivation der Funktion „$f(x) = 1-x$“. Wobei *f(x)* und *x* stellen den Schupf bzw. den Wirkungsgrad dar des Asynchronmotors dar. Mithilfe der Konsole wird die Ausgabe der funktionalen Modellierung dargestellt.

Listing 1.38 Anwendung der Lambda-Funktion zum Darstellen der derivaten Funktion

```
// Effizienz4Funktion.cpp : Diese Datei enthält die Funktion "main".
Hier beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <iomanip>
#include <functional>
// Annäherung des Derivates der Funktion f mithilfe des Wertes h.
// h nährt null an zum Erhalten einer besseren Abschätzung
std::function<double(double)> derivative(std::function<double(double)>
f, double h)
{

// Verwendung der Funktion f und des Wertes h
 return [f, h] (double x)
 {
  return (f(x + h) - f(x)) / h;
 };
}
double schlupf4Analyse(double x)
{

// Die gewünschte Funktion zum Differenzieren
 return 1-x;
}

// Das bekannte Derivat der Funktion schlupf4Analyse()
double ans= -1.0

  int main()
  {

// Die Differenz f(x+h) - f(x) ist die Annäherung wenn h -> 0
   double h = 0.001;
```

```
// Berechnung der Funktion der Annährung zur Derivation der Funktion
// schlupf4analyse
   auto der = derivative(schlupf4Analyse, h);

// Vergleich zwischen dem berechneten Derivat und dem exakten
// abgeleiteten Derivat
   double x = 0.97;
std::cout << "-------------------------------------------------\n";
  std::cout << " Annaeherung Aktuell \n";
   std::cout << " x f(x) h f\'(x) f\'(x) \n";
   std::cout << "-------------------------------------------------\n";
while (x < 0.98)
{
std::cout << std::fixed << std::showpoint << std::setprecision(4);
std::cout << x << " " << schlupf4Analyse(x) << " " << h << " " <<
der(x) << " " << ans << '\n';
x += 0.01;
}

}
```

Listing 1.39 Konsole zum Anzeigen des Kompilieren des Programms vom Listing 1.38

```
-------------------------------------------------------
Annaeherung Aktuell
x f(x) h f'(x) f'(x)
-------------------------------------------------------
0.97000 0.03000 0.00100 -1.00000 -1.00000
C:\Users\eric\source\repos\Effizienz4Funktion\Debug\Effizienz4Funktion.
exe (Prozess "11568") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.4.3.2.5 Anwendungen der C++-Standardbibliothek in der Modellierung der Parallelisierung

Die Modellierung der Parallelisierungsprozesse mithilfe der C++-Standardbibliothek ermöglicht die Anwendungen der verschiedenen Tools wie Algorithmen, Vektor oder Lambda-Ausdrücke in der parallelen Darstellung der Sequenzen. Hierbei stellt dieser Abschnitt das Modellieren der Werkzeuge der C++-Standardbibliothek zum Parallelisieren von Sequenzen. Listing 1.40 zeigt die Anwendungen der verschiedenen Tools in der Parallelisierung von Sequenzen. Mithilfe des Algorithmus *„std::for_each“* wird die Funktion *„for_each()“* einen Iterator zum Treiben einer Schleife in

der Sequenz dargestellt. Hierbei ermöglicht die Funktion *„for_each()“* die Anwendung einer einstelligen Funktion für jedes Element im Container. Die folgenden Ausdrücke aus dem Listing 1.40 geben mithilfe der Funktion *for_each()* jedes Element in dem *std::vector*-Objekt genannt *seq1* bestehend aus dem Integer-Typ.

```
std::for_each(std::begin(seq1), std::end(seq1), [](int a)
 {
  std::cout << a << ' ';
 });
```

Die Anwendung der Lambda-Funktion mithilfe des Parameter *„a“* vom Typ *„Integer“* in dem oben genannten Ausdruck ermöglicht die Funktion *„std::for_each()“* zum Verändern jedes Elementes in der Sequenz *„seq1“*. Hierbei passiert die Lambda-Funktion den Parameter *„a“* mithilfe der Referenzierung.

Gemäß Listing 1.40 ermöglicht die Funktion *„std::iota()“* das Ausfüllen des Vektors seq1 mit aufsteigenden Zahlen, d. h. von 0, 1, 2, 3, 4, 5, 6 bis 41. Die Funktion *„std::copy()“* ermöglicht das Kopieren der Elemente von einem Container auf einen anderen Container z. B. von seq1 auf seq2. Aber die Funktion *„std::transform()“* ermöglicht auch die Veränderung der kopierten Elemente in der anderen Sequenz mithilfe einer Anwendung der Lambda-Funktion in der Implementierung der Funktionalität zur Parallelisierung. *„std::copy()“* und *„std::transform()“* benötigen einerseits den *„beginn“*- und *„end“*-Iterator aus dem Quelle-Container, z. B. *„seq1“*, und andererseits den beginn-Iterator für dem Empfänger-Container, z. B. *„seq2“* bzw. *„seq3“*. Gemäß Listing 1.41 zeigt die Konsole die Ausgabe des Programms aus dem Listing 1.40 zum Darstellen der Parallelisierungsprozesse zum Berechnen der synchronen Drehzahl aus dem Polpaarzahl mithilfe der Implementierungen der Funktionen *„std::for_each()“*, *„std::iota()“*, *„std::copy()“* und *„std::transform()“*. Listing 1.41 zeigt drei Sequenzen mit Ganzzahlen. Die erste Sequenz besteht aus den Polpaarzahlen des Asynchronmotors von 1 bis 41 und die zweite ist mithilfe der Funktion *„std::copy()“* aus der ersten erstellt worden. Die dritte Sequenz wurde mithilfe der Funktion *„std::transform()“* nach der Transformation der ersten erstellt.

Listing 1.40 Anwendung der Algorithmen in der Implementierung der Funktionen zum Parallelisieren

```
// Funktion4Vektor.cpp : Diese Datei enthält die Funktion "main". Hier
beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
int main() {
```

```
 const int SIZE = 41;
 // Einen Vektor aus Polpaarzahlen des Asynchronmotors mit Integer-
Groessen erstellen
 std::vector<double> seq1(SIZE);

 // Den Vektor mit den Polpaarzahlen 1, 2, 3, 4,….10,…, SIZE - 1
ausfuellen
 std::iota(std::begin(seq1), std::end(seq1), 1);

 // Den Vektor anzeigen
 std::cout << " Die Polpaarzahl p des Asynchronmotors ist :" << '\n';
 std::cout << '\n';
 std::for_each(std::begin(seq1), std::end(seq1), [](int a)
 {
  std::cout << a << ' ';
 });
std::cout << '\n';

 // Den Vektor vergroessern damit die kopierten Werte erhalten werden
 std::vector<int> seq2(SIZE);

 // seq1 auf seq2 kopieren
 std::copy(std::begin(seq1), std::end(seq1), std::begin(seq2));
 std::cout << '\n';
 // seq2 anzeigen
 std::cout << " Die Polpaarzahl p des Asynchronmotors ist :" << '\n';
 std::cout << '\n';
 std::for_each(std::begin(seq2), std::end(seq2), [](int a)
 { std::cout << a << ' ';
 });
 std::cout << '\n';
 // Den Vektor vergroessern damit die tranformierten Werte erhalten
werden
 std::vector<int> seq3(SIZE);
 // seq1 auf seq3 kopieren
 std::transform(std::begin(seq1), std::end(seq1), std::begin(seq3), []
(int p)
 {
  return 50 * 60 * 1/(int)p;
 });
 std::cout << '\n';
 // seq3 anzeigen
 std::cout << " Die Drehfelddrehzahl in 1/min mit der Polpaarzahl p
bei Frequenz f= 50 1/s ist :" << '\n';
 std::cout << '\n';
 std::for_each(std::begin(seq3), std::end(seq3), [](int a)
```

```
 {

  std::cout << a << ' ';
 });

 std::cout << '\n';
}
```

Listing 1.41 Konsole der Ausgabe des Programmes aus dem Listing 1.40 zum Darstellen der Parallelisierung

```
Die Polpaarzahl p des Asynchronmotors ist :
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
Die Polpaarzahl p des Asynchronmotors ist :
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
Die Drehfelddrehzahl in 1/min mit der Polpaarzahl p bei Frequenz f=
50 1/s ist :
3000 1500 1000 750 600 500 428 375 333 300 272 250 230 214 200 187 176
166 157 150 142 136 130 125 120 115 111 107 103 100 96 93 90 88 85 83
81 78 76 75 73
C:Users\eric\source\repos\Funktion4Vektor\Debug\Funktion4Vektor.exe
(Prozess "24568") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.

// Funktion4Drehzahlen.cpp : Diese Datei enthält die Funktion "main".
Hier beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <iostream>
#include <string>
#include <tuple>
int main()
{
//Einige lokal variablen deklarieren
 std::string bezeichnung;
 int strassenpreis;
 double leistung;
 // Tuple-Funktion mithilfe von algebraischen Werten erstellen
 std::tuple<std::string, int, double> t1 {"Lyra", 400, 20.8};
```

```
 // Die Elemente von Tuple mithilfe der Funktion tie() auspacken
 std::tie(bezeichnung, strassenpreis, leistung) = t1;
 //Die Ergebnissse anzeigen
 std::cout << "Bezeichnung vom Mesh-WLAN-System = " << bezeichnung <<
 ", Strassenpreis in Euro fuer WLAN 3er-System = "
  << strassenpreis << ", Leistungsaufnahme in Watt fuer 3er-System =
  " << leistung << '\n';

 //Die Funktion make-tuple() anwenden
 auto t2 = std::make_tuple("Fritz!WLAN Repearter 1750E", 390, 15.8);
 // Die Elemente von Tuple mithilfe der Funktion tie() auspacken
 std::tie(bezeichnung, strassenpreis, leistung) = t2;

 // Die Ergebnisse anzeigen
 std::cout << "Bezeichnung vom Mesh-WLAN-System= " << bezeichnung <<
 ", Strassenpreis in Euro fuer WLAN 3er-System = " << strassenpreis
  << ", Leistungsaufnahme in Watt fuer 3er-System = " << leistung <<
  '\n';
 // Einige Variable deklarieren
 std::string marke = "dLAN 1200+ Wifi ac";
 int preis = 200;
 double leistungsaufnahme = 9.4;

 // Das Tuple für die allgemeine Anwendung erstellen
 auto t3 = std::make_tuple(marke, preis * 2, leistungsaufnahme * 2);

 // Die Elemente von Tuple mithilfe der Funktion get() auspacken einmal
 bezeichnung = std::get<0>(t3);

 // Das erste Element liegt bei Index =0
 strassenpreis = std::get<1>(t3);
 // Das zweite Element liegt bei Index =1
 leistung = std::get<2>(t3);
 // Das dritte Element liegt bei Index =2
 // Die Ergebnisse anzeigen
 std::cout << "Bezeichnung vom Mesh-WLAN-System = " << bezeichnung <<
 ", Strassenpreis = " << strassenpreis << " Leistung = " << leistung
<< '\n';
}
```

```
Bezeichnung vom Mesh-WLAN-System= Fritz!WLAN Repearter 1750E,
Strassenpreis in Euro fuer WLAN 3er-System = 390, Leistungsaufnahme
in Watt fuer 3er-System = 15.8
Bezeichnung vom Mesh-WLAN-System = dLAN 1200+ Wifi ac, Strassenpreis
= 400 Leistung = 18.8
```

```
C:\Users\eric\source\repos\Funktion4Drehzahlen\Debug\
Funktion4Drehzahlen.exe (Prozess "3504") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.4.3.2.6 Anwendungen der C++-Standardbibliothek in der Implementierung der Funktion *„std::tuple()“*

Tuple-Funtionen ermöglichen die Rückgabe verschiedener Datentype mithilfe eines Vektors. Dies wird mithilfe der Rückgabe des gewünschten Vektors realisiert. Beim Aufruf einer Funktion werden die Elemente des Vektors mithilfe der Rückgabe der Funktion ausgezogen.

Listing 1.42 zeigt die Anwendungen der Funktion *„std::tie()“* in *„std::tuple()“* zum Auspacken der Elemente eines Vektors, genannt *„t1“*, und zum Übertragen der Elemente von *„std::stuple()“* an einzelnen Variablen mithilfe eines Ausdruckes. *„std::tie()“* ist eine generische Funktion zum Wenden an jedem gespeicherten Typen in der Funktion *„std:tuple()“*. Die Anwendung der generischen Funktion *„std::get()“* in *„std::stuple()“* ermöglicht nur das Auspacken von einem Element mithilfe einer Parametrisierung mit Ganzzahlen als Index. Die *„main“*-Funktion gemäß Listing 1.42 ermöglicht die Aufrufe der Funktionsobjekte von *„std::get<>()“* bezüglich der Zugriffe auf die Elemente der Funktion *„std::tie()*: bezeichnung, strassenpreis, leistung“, die mithilfe der generischen Funktion *„get<>()“* ausgepackt werden. Die Funktionsobjekte *„t2“* und *„t3“* ermöglichen das Erstellen der Tuple-Elemente mithilfe der Funktion *„std::make_tuple(); t1“* ist sie Funktionsobjekt von *„std::stuple()“* zum Erstellen von algebraischen Werten. Der Aufruf des Funktionsobjekts *„t3“* wird mithilfe des Aufrufes des Funktionsobjektes *„t2“* realisiert. *„std::tie()“* hat für Objekt *„t2“* und besteht aus den Parametern: *„Bezeichnung“*, *„strassenpreis“* und *„Leistung“*, welche Funktionsobjekte der generischen Funktion *„std::get<>()“* darstellen. Bei Aufruf dieser Funktionsobjekte werden *„t3“* und *„t2“* aufgerufen. Von daher werden *„std::make_tuple()“* bzw. *„std::tie()“* aufgerufen. Listing 1.43 zeigt die Konsole zur Ausgabe der Eigenschaften der verschiedenen WLAN-Systeme bezüglich der Bezeichnung, des Preises, der Leistung.

Listing 1.42 Implementierung der Funktionalität von *std::tuple()* zum Darstellen der Rückgabe verschiedener Datentype aus einem Vektor

```
// Funktion4Drehzahlen.cpp : Diese Datei enthält die Funktion "main".
Hier beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <string>
#include <tuple>
```

```
int main()
{
//Einige lokal variablen deklarieren
 std::string bezeichnung;
 int strassenpreis;
 double leistung;
 // Tuple-Funktion mithilfe von algebraischen Werten erstellen
 std::tuple<std::string, int, double> t1 {"Lyra", 400, 20.8};

 // Die Elemente von Tuple mithilfe der Funktion tie() auspacken
 std::tie(bezeichnung, strassenpreis, leistung) = t1;
 //Die Ergebnisse anzeigen
 std::cout << "Bezeichnung vom Mesh-WLAN-System = " << bezeichnung <<
 ", Strassenpreis in Euro fuer WLAN 3er-System = "
  << strassenpreis << ", Leistungsaufnahme in Watt fuer 3er-System =
  " << leistung << '\n';
 //Die Funktion make-tuple() anwenden
 auto t2 = std::make_tuple("Fritz!WLAN Repearter 1750E", 390, 15.8);
 // Die Elemente von Tuple mithilfe der Funktion tie() auspacken
 std::tie(bezeichnung, strassenpreis, leistung) = t2;

 // Die Ergebnisse anzeigen
 std::cout << "Bezeichnung vom Mesh-WLAN-System= " << bezeichnung <<
 ", Strassenpreis in Euro fuer WLAN 3er-System = " << strassenpreis
  << ", Leistungsaufnahme in Watt fuer 3er-System = " << leistung <<
  '\n';
 // Einige Variable deklarieren
 std::string marke = "dLAN 1200+ Wifi ac";
 int preis = 200;
 double leistungsaufnahme = 9.4;

 // Das Tuple für die allgemeine Anwendung erstellen
 auto t3 = std::make_tuple(marke, preis * 2, leistungsaufnahme * 2);

 // Die Elemente von Tuple mithilfe der Funktion get() auspacken ein-
mal
 bezeichnung = std::get<0>(t3);

 // Das erste Element liegt bei Index =0
 strassenpreis = std::get<1>(t3);
 // Das zweite Element liegt bei Index =1
 leistung = std::get<2>(t3);
 // Das dritte Element liegt bei Index =2
 // Die Ergebnisse anzeigen
std::cout << "Bezeichnung vom Mesh-WLAN-System = " << bezeichnung << ",
Strassenpreis = " << strassenpreis << " Leistung = " << leistung << '\n';
}
```

```
Listing 1.43 Konsole zur Ausgabe der Eigenschaften von drei WLAN-
Systeme bezüglich der Bezeichnung, des Preises und der Leistung

Bezeichnung vom Mesh-WLAN-System : Lyra, Strassenpreis in Euro fuer
WLAN 3er-System = 400, Leistungsaufnahme in Watt fuer 3er-System =
20.8
Bezeichnung vom Mesh-WLAN-System : Fritz!WLAN Repearter 1750E,
Strassenpreis in Euro fuer WLAN 3er-System = 390, Leistungsaufnahme
in Watt fuer 3er-System = 15.8
Bezeichnung vom Mesh-WLAN-System : dLAN 1200+ Wifi ac, Strassenpreis
= 400, Leistung = 18.8
C:\Users\eric\source\repos\Funktion4Drehzahlen\Debug\
Funktion4Drehzahlen.exe (Prozess "13408") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.4.3.2.7 Anwendung der „*std::map*“-Objekte in der Analyse der Elemente einer Sammlung

Der Zugang auf Elemente einer Sammlung wird effizienter mithilfe der „*std::map*“-Objekte ermöglicht.

Mithilfe der Klasse „*map<Key, t>*“ werden „*Paare*“ von Schlüsseln, genannt „*Key*“, und deren Daten gespeichert, wobei jedem Datensatz demselben Schlüssel entspricht. Hierbei stellt „*map*“ einen assoziativen Container zum Finden der Daten mithilfe der direkten Angabe des Schlüssels [15]. Die Abbildung, genannt „*map*“, ermöglicht den Zugriff auf Daten mithilfe eines Schlüssels. Gemäß Listing 1.44 stellt das „*map*“-Objekt „*std::map<std::string, double> wlan*“ ein Paar mithilfe eines konstanten Schlüssels zum Ermöglichen einer Sortierung der Position im Container dar.

Die Implementierung der Erstellung einer Liste von WLAN-Systeme bezüglich der „*Eingaben*“, der „*Suche*“ und des „*Beendens*“ der Elemente von „*std::map()*“ mithilfe der Überladung Funktionen „*operator[]()*“, „*operator*<<()“ und „*operator*>>()“ zeigt Listing 1.44. Die Funktion „*std::map()*“ ermöglicht das Zuordnen der Elemente nach ihren Bezeichnungen und Leistungen. Hierbei wird eine Liste aus WLAN-Bezeichnern und deren Leistungen mithilfe eines Schlüsselmachers, genannt „*Map*“. Listing 1.44 verfügt über eine geheime Depot-Anweisung zum Anzeigen des Inhaltes des Map-Objekts. Hierbei führt diese Anweisung die folgende Schleife:

```
for (auto& elem : wlan) std::cout << elem.first << " " << elem.second
<< ' n';
```

Die Schleife zeigt die Reihenfolge des Aufrufes des Elements des Paars im Container, wobei das erste Element der WLAN-Bezeichnung wie z. B. AmplifiHD und das zweite der WLAN-Leistung wie z. B. 13.5 entsprechen. Die Modellierung der Funktionalität

der Sortierung mithilfe vom *„map“*-Container stellt die Analyse der *„main“*-Funktion dar. Die Hauptfunktion main() besteht aus zwei Type von Kontrollstrukturen zum Implementieren der Sortierung von Paaren aus WLAN-Bezeichnern und -Leistung: *„while“* und *„switch“*. Die Funktionalität zur Wiederholung der Informationen mithilfe der Schleife *„while“* ermöglicht die Anwendung der Operatoren „<<“ und „>>“ in der Erstellung einer Liste aus WLAN-Systemen. Die Ausdrücke zum Darstellen der o. g. Operatoren werden in der Schleife switch zum Implementieren die Erstellung der WLAN-Liste angewendet.

Die Verwendung des Index-Operators [] ermöglicht die Sortierung der existierten Einträge. Gemäß Listing 1.44 werden sowohl Eingabe als auch Ausgabe mithilfe der Operatoren >> bzw. << ermöglicht. Gemäß Listing 1.44 und 1.45 werden jedes Paar aus Bezeichnung und Leistung angezeigt, wenn der Schlüssel dem richtigen Wert entspricht. Hierbei entspricht ein WLAN-Bezeichner nur einer Leistung.

Listing 1.44 Erzeugung einer Liste aus WLAN-Bezeichnern und deren Leistungen mithilfe von Map-Objekte

```
// MeshWLANSysteme.cpp : Diese Datei enthält die Funktion "main". Hier
beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
int main() {
 std::map<std::string, double> wlan;
 // Liste von verschiedenen Marken fuer Mesh-WLAN-Systeme
 bool running = true;
 while (running)
 {
  std::string bezeichnung;
  double leistung;
  char anweisung;
  std::cout << "E)ingeben S)uchen B)eenden: ";
  std::cin >> anweisung;
  switch (anweisung)
  {
  case 'E':
  case 'e':
    std::cin >> bezeichnung;
   std::transform(std::begin(bezeichnung),      std::end(bezeichnung),
   std::begin(bezeichnung), toupper);
```

```
    std::cin >> leistung;
   wlan[bezeichnung] = leistung;
  break;
  case 'S':
  case 's':
  std::cin >> bezeichnung;
transform(std::begin(bezeichnung),   std::end(bezeichnung),   std::be-
gin(bezeichnung), ::toupper);
  std::cout << bezeichnung << " " << wlan[bezeichnung] << '\n';
  break;
  case 'B':
  case 'b':
  running = false;
  break;
  case 'G': // Das ist eine geheime Anweisung
  case 'g': for (auto& elem : wlan) std::cout << elem.first << " " <<
  elem.second << '\n';
  break;
  default: std::cout << anweisung << "ist nicht eine ordentliche
  Anweisung" << '\n';
  }
 }
}
```

Listing 1.45 Konsole zum Darstellen die Sortierung der WLAN-Liste mithilfe des map-container

```
E)ingeben S)uchen B)eenden: E
AmplifiHD
13.5
E)ingeben S)uchen B)eenden: E
DecoM5
11.7
E)ingeben S)uchen B)eenden: E
OrbiRBK43
18.8
E)ingeben S)uchen B)eenden: E
Velop
15.1
E)ingeben S)uchen B)eenden: E
Wifi
9.9
E)ingeben S)uchen B)eenden: E
dLAN1200+Wifiac
17.5
```

```
E)ingeben S)uchen B)eenden: E
Fritz!Powerline1260E
23.5
E)ingeben S)uchen B)eenden: E
Fritz!WLANRepeater1750E
15.8
E)ingeben S)uchen B)eenden: E
Lyra
20.8
E)ingeben S)uchen B)eenden: S
Velop
VELOP 15.1
E)ingeben S)uchen B)eenden: S
DecoM5
DECOM5 11.7
E)ingeben S)uchen B)eenden: S
Lyra
LYRA 20.8
E)ingeben S)uchen B)eenden: S
Wifi
WIFI 9.9
E)ingeben S)uchen B)eenden: G
AMPLIFIHD 13.5
DECOM5 11.7
DLAN1200+WIFIAC 17.5
FRITZ!POWERLINE1260E 23.5
FRITZ!WLANREPEATER1750E 15.8
LYRA 20.8
ORBIRBK43 18.8
VELOP 15.1
WIFI 9.9
E)ingeben S)uchen B)eenden:
```

1.4.3.2.8 Modellierung der Ausnahmebehandlung mithilfe der Ein- und Ausgabemöglichkeit

Ziel der Anwendung des Stream-Konzeptes ist es, den Fluss der abstrakten Datenströme von einer Quelle zu einer Senke zu ermöglichen. Mithilfe der Header-Datei wie z. B. <*iostream*> oder <fstream> werden Standardein-und ausgabe bzw. Ein- und Ausgabe für Dateien ermöglicht.

Die Modellierung der Ausnahmebehandlung mithilfe des Stream-Konzeptes erzielt sowohl das Erkennen als auch das Signalisieren der Fehler mithilfe der Fehler-Meldung *„try“* und *„catch“*. Hierbei wird die Ausnahme, genannt *„exception“*, mit der *„throw“*-Anweisung geworfen. Die Funktionalität zu modellieren bedeutet, dass eine Funktion eine bestimmte Aufgabe mithilfe des C++-Schlüsselworts *„try“* zu terminieren versucht. Wobei eine Ausnahme beim Feststellen eines Fehlers, genannt

exception, gestellt wird. Mithilfe des C++-Schlüsselworts *„catch“* wird die Ausnahme von einem Fehlerbehandlungsprozess zum Bearbeiten und zum Transportieren aufgefangen. Fall es keine Möglichkeit in der Hauptfunktion *„main()“* den Fehler nicht zu bearbeiten, wird das Programm abgebrochen. Listing 1.46 zeigt die Modellierung einer Kaskade von Fehlerbehandlungen mithilfe von catch-Klauseln. Hierbei zeigt Listing 1.46 verschiedene Arte von Fehlern und deren Behandlungen mithilfe der *„try“*- und *„catch“*-Anweisungen. Gemäß Listing 1.46 wird mithilfe einem try-Block und drei catch-Blöcken das Auswerfen der Ausnahme ermöglicht. Die Funktionalität der Modellierung fokussiert zum einen auf die Anwendungen des Operators << im Auswerfen der Ausnahme und zum anderen auf die Überladung der Funktion *„std::stoi()“*. Hierbei wird „std::stoi“ einerseits beim Eingeben einer falschen Ganzzahlen eine ungültige Ausnahme mithilfe der Funktion *„std::invalid_argument()“* auswerfen und andererseits beim Entsprechen des Strings einer gültigen Ganzzahl außerhalb des C++-Ganzzahlbereiches die Funktion *„std::out_of_range()“* auswerfen. Die Funktion *„get_int()“* stellt zum einen die Überladung des *„Operators >>“* und zum anderen den Zugriff auf die Variable oder Senke, genannt *„eingabe“*, mithilfe der Funktion *„std::stoi()“* dar. Gemäß Listing 1.46 zeigt die Eingabeanweisung *„std::cin >> eingabe“*, dass der Datenstrom in Richtung auf das Ziel des Eingabeoperators verläuft >>. Im try-Block wird die Funktion „get_int()“ zum einen mithilfe des Ausdrucks *„int i= get_int()“* überladen und zum anderen mithilfe der Anweisung *„std::cout<< i“* aufgerufen. Anschließend wird mithilfe der Anweisung *„std::cout << x. at(i)“* sowohl die Funktion „at()“ mit ihrer Variable *i* aufgerufen. Hierbei gibt der Benutzer eine gültige Ganzzahl im Vektor *x*, dessen index *i* mithilfe der Funktion „at()“ festgestellt ist. Die Fehlerbehandlung wird mithilfe der drei catch-Blöcke zum Feststellen des Typs der Ausnahme realisiert. Der erste catch-Block wird ausgeführt, wenn der eingegebene Index ein Treffer ist. Ansonsten wird mithilfe des zweiten catch-Blocks der Typ der Ausnahme mit der Funktion *„std::out_of_range()“* verglichen. Falls es keinen Treffer gibt, wird der catch-Block mithilfe der Anweisung *„std::cout<< „Unbekannter Fehler!\n““* ausgeführt.

Listing 1.46 Modellierung des Auswerfens der Ausnahmen mithilfe der Stream-Bibliothek

```
// Exception4Funktion.cpp : Diese Datei enthält die Funktion "main".
Hier beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
int get_int()
{
```

```
std::string eingabe;
std::cin >> eingabe;
int result = stoi(eingabe);
return result;
}
int main() {
 std::vector<int> x{ 1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12, 13, 14,
15,16, 17, 18, 19, 20 };
 try {
  // Ziel ist es, den Fehler zu analysieren
  int n = x.size();
  std::cout << "Bitte geben Sie eine Ganzzahl als Polpaarzahl fuer den
  Asynchronmotor zwischen 1 und 20 ein. D.h. bis …" << n - 1 << ": ";

  // Der Benutzer soll eine gueltige Ganzzahl im Vektor x angeben
  int i = get_int();
  std::cout << "Eingebener Index: " << i << '\n';

  // Der eingegebene Index soll dem Vektor x gehoeren
  std::cout << "a[" << i << "] = " << x.at(i) << '\n';
 }
 catch (std::out_of_range&)
 {
  std::cout << "Der eingegebene Index gehort nicht zur Reihe.\n";
 }
 catch (std::invalid_argument&)
 {
  std::cout << "Der eingegebene Index ist nicht eine Ganzzahl!\n";
 }
 catch (…)
 {
  // Fehler pruefen
  std::cout << "Unbekannter Fehler!\n";
 }
}
```

1.4.3.2.9 Modellierung des Überladens der Operatoren < und << mithilfe des *„set“*-Containers

Anwendung des Set-Containers zum Modellieren des Überladens der Operatoren ermöglichen die Darstellung von Datentypen und Funktionen einer Klasse. Hierbei wird ein Set mit Daten zum Darstellen einer Menge initialisiert. Gemäß Listing 1.47 sind zwei *„set“*-Container, genannt *„Laeuferdrehzahl“* und *„Drehmoment“*, mit Integer-Elementen dargestellt. Mithilfe des Operators < werden die Objekte der Klasse *„MotorBetrieb“* verglichen. Hierbei zeigt die main-Funktion die Funktionalität des Klassenobjektes *„synchronedrehzahl“* mithilfe des Vergleichs zwischen Elementen von

„synchronedrehzahl“ zum Überladen des Operators <. Wobei die Funktion operator<() aus zwei referenzierten konstanten Parametern *x* und *y* den Vergleich zwischen x und y zurückgibt.

In der main-Funktion wird der Operator << zum Ausgeben der Elemente des Klassenobjektes *„synchronedrehzahl“* überladen. Mithilfe des Stream-Konzeptes wird das Überladen Ausgabeoperator << für *„data“* mit dem Datentyp Integer ermöglicht wie die folgende Anweisung es bestätig:

```
os << obj.data
```

„os“ ist als Referenz für das als erste Argument übergebene *„ostream“*-Objekt deklariert. Der Parameter *„obj“* in der zweiten Position ist vom Typ *„const MotorBetrieb&“*. Die o. g. Anweisung zeigt den Zugriff auf *„data“* mithilfe des zweiten Argumentes der Funktion *„operator<<()“*. Hierbei ist diese Anweisung in Ordnung, weil *„data“* im Konstruktor deklariert ist.

Der Konstruktor der Klasse *„MotorBetrieb“* ist explizit definiert, damit der Compiler keinen *„default“*-Konstruktor erstellt.

Gemäß Listing 1.47 Überladen des Operators << ermöglicht das Anzeigen der Elemente der set-Container „Laeuferdrehzahl“ und *„Drehmoment“* mithilfe der Anwendungen der for-Schleife in der Sortierung der Elemente der *„set“*-Container „Laeuferdrehzahl“ und *„Drehmoment“*.

Listing 1.47 Überladen der Operatoren < und << zum Ausgeben der Klassenobjekte und zum Sortieren der Elemente der *set*-Container

```
#include <iostream>
#include <set>
struct MotorBetrieb
{
 int data;
MotorBetrieb(int d): data(d)
{
}
};
bool operator<(const MotorBetrieb& x, const MotorBetrieb& y)
{
 return x.data < y.data;
}
std::ostream& operator<<(std::ostream& os, const MotorBetrieb& obj)
{
 os << obj.data;
 return os;
}
```

```
int main()
{
 std::set<int> Laeuferdrehzahl {0, 500, 1000, 1500, 2000};
 for (auto elem: Laeuferdrehzahl)
std::cout << "Laeuferdrehzahl in 1/min" <<elem << ' ';
 std::cout << '\n';
 std::set<int> Drehmoment{ 0, 50, 100, 150, 200 };
 for (auto elem: Drehmoment)
std::cout << "Drehmoment in Nm"<< elem << ' ';
 std::cout << '\n';

 std::set<MotorBetrieb> synchroneDrehzahl
 {
  MotorBetrieb(3000),MotorBetrieb(2000),MotorBetrieb(1500),
MotorBetrieb(500)
};
 for (auto& elem: synchroneDrehzahl)
std::cout << "Sychron Drehzahl in 1/min"<< elem << ' ';
 std::cout << '\n';
}
```

1.4.3.2.10 Modellierung der Anwendung der Funktion *„evaluate() in der Analyse der Zeiger auf Funktionen“*

Dieser Abschnitt modelliert die Berechnungen der Kenngrößen Schlupf und Wirkungsgrad mithilfe der Implementierung der Funktionen *„schlupf4berechnen()“* und *„wirkungsgrad4berechnen()“* durch einerseits der Substraktion zwischen 1 und des Ergebnisses der Division zwischen *„Laeuferdrehzahl“* und *„synchronen Drehzahl“* und andererseits der Division zwischen *„Laeuferdrehzahl“* und *„synchronen Drehzahl“*. Gemaß Listing 1.48 sind die beide Funktionen zum Berechnen erstellt. Mithilfe der Zeigerskonzeptes wird eine neue Funktion, genannt *„evaluate()“* zum Kopieren der Berechnungen auf Basis der Zeiger auf Funktionen. Die Funktion *„evaluate()“* besteht aus drei Argumenten: die Funktion f mit dem Typ *„double“* als Zeiger von *„evaluate()“*, *„laeuferdrehzahl“* und *„synchronedrehzahl“* mit dem Datentyp *„int“* jeweils. Die Funktion *„evaluate()“* gibt die Funktion f mit den Argumenten *„laeuferdrehzahl“* und *„synchronedrehzahl“* zurück. Hierbei stellt das erste Argument der Funktion *„evaluate()“* eine Zuweisung an f() dar. Die Analyse der Funktion „evaluate()“ zeigt, dass ihre Implementierung auf die Speicherung der Adresse der Funktion mithilfe einer Pointer-Funktion, genannt f, fokussiert. Die Zuweisung ermöglich die Anwendung der funktionalen Modellierung in der Parallelisierung der Berechnungen. Es ist zu bemerken, dass evaluate() die Funktion f() aufruft. Die main-Funktion ruft die Funktion *„evaluate()“* durch *„schlupf4berechnen()“* zum einen und *„wirkungsgrad4berechnen()“* zum anderen auf. Listing 1.49 zeigt das Ergebnis der Berechnung und derer Zeiger.

Listing 1.48 Modellierung von Berechnungen und deren Zeiger

```
// Modellierung_evaluate_Funktion.cpp : Diese Datei enthält die
Funktion "main". Hier beginnt und endet die Ausführung des Programms.
//
#include "pch.h"
#include <iostream>
double schlupf4berechnen(int laeuferdrehzahl, int synchronedrehzahl)
{
 int a = 1;
 return a - (double)laeuferdrehzahl / synchronedrehzahl;
}
double wirkungsgrad4berechnen(int laeuferdrehzahl, int synchronedrehzahl)
{
 return (double)laeuferdrehzahl / synchronedrehzahl;
}
double evaluate(double(*f)(int, int), int laeuferdrehzahl, int
synchronedrehzahl)
{
 return f(laeuferdrehzahl, synchronedrehzahl);
}
int main()
{
 std::cout << "Laeuferdrehzahl:2550 Umdrehungen/min" << '\n';
 std::cout << "synchrone Drehzahl:2750 Umdrehungen/min" << '\n';
 std::cout << "Berechnung vom Schlupf mithilfe von der Laeuferdrehzahl
 und der synchronen Drehzahl:" << schlupf4berechnen(2550, 2750) << '\n';
 std::cout << "Berechnung vom Wirkungsgrad mithilfe von der
 Laeuferdrehzahl und der synchronen Drehzahl:" << wirkungsgrad-
 4berechnen(2550, 2750) << '\n';
 std::cout << "Berechnung des Zeigers auf den Schlupf: " <<
 evaluate(&schlupf4berechnen, 2550, 2750) << '\n';
 std::cout << "Berechnung des Zeigers auf den Wirkungsgrad: " << evalu
 ate(&wirkungsgrad4berechnen, 2550, 2750) << '\n';
}
```

Listing 1.49 Konsole zum Anzeigen der Berechnungen

```
Laeuferdrehzahl:2550 Umdrehungen/min
synchrone Drehzahl:2750 Umdrehungen/min
Berechnung vom Schlupf mithilfe von der Laeuferdrehzahl und der
synchronen Drehzahl:0.0727273
Berechnung vom Wirkungsgrad mithilfe von der Laeuferdrehzahl und der
synchronen Drehzahl:0.927273
Berechnung des Zeigers auf den Schlupf: 0.0727273
Berechnung des Zeigers auf den Wirkungsgrad: 0.927273
```

```
C:\Users\eric\source\repos\Modelierung_evaluate_Funktion\Debug\
Modelierung_evaluate_Funktion.exe (Prozess "12848") wurde mit Code "0"
beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.4.3.2.11 Modellierung der Funktionalität von *„std::for_each()“* zum Darstellen der Lambda-Ausdrücke-Rolle

Die Anwendung der Algorithmen der C++-Standardbibliothek in der Veränderung der Elemente eines Containers wird mithilfe der Funktion *„for_each()“* ermöglicht. Hierbei wirkt die Lambda-Funktion zum Ermöglichen des Durchganges der Element durch Referenzierung. Gemäß Listing 1.50 wird zuerst eine Liste, genannt *„seq“*, aus Ganzzahlen zum Darstellen der Paarpolzahlen eines Asynchronmotors erstellt. Anschließend wird *„seq“* mithilfe der Funktion *„for_each()“*, der Iterator-Funktionen *„begin()“* und *„end()“*; und der Lambda-Funktion durch den referenzierten Parameter *x* kopiert. Die o. g. Iteratoren verweisen auf Elemente von der Liste *„seq“*. Nach dem Kopieren der Liste seq wird mithilfe der drei o. g. Funktionen eine Liste aus Ganzzahlen zum Darstellen der Berechnung der synchronen Drehzahlen eines Asynchronmotors aus der Liste seq erstellt. Die zweite *„for_each“*-Funktion besteht aus einer Formel zum Berechnen der Drehfelddrehzahl eines Asynchronmotors mithilfe einerseits des referenzierten Parameter x durch die Wirkung der Lambda-Funktion und andererseits der Konstante a. Wobei a eine Berechnung 60*50 für die Konvertierung der Frequenz 50 Hz in 1/s darstellt. Mithilfe der dritten *„for_each“*-Funktion wird das Ergebnis der Berechnung der synchronen Drehzahl, genannt Drehfelddrehzahl, auf dem Bildschirm angezeigt. Listing 1.51 zeigt das Ergebnis der Anwendungen der drei *„for_each“*-Funktionen mithilfe einer Liste *„seq“* und der Wirkung der Lambda-Funktion an.

Listing 1.52 ist eine Wiederholung des Konzeptes der Wirkung der Lambda-Funktion zum Implementieren der *„for_each“*-Funktion. Hierbei besteht die Liste seq auch aus Ganzzahlen zum Darstellen der Drehzahlen eines Asynchronmotors. Listing 1.52 zeigt die Funktionalität zum Erhöhen jedes Elementes der Liste *„seq“* mit der Zahl 1. Dies ermöglicht die Erstellung einer neuen Liste aus neuen Drehzahlen dank der Wirkung der Lambda-Funktion mithilfe der Iterator-Funktionen *„begin()“* und *„ende()“* in der Function *„for_each()“*.

Listing 1.50 Modellierung der Verwendungen der *for_each*-Funktion zum Darstellen der Elemente einer Liste mit der Wirkung der Lambda-Funktion

```
#include <iostream>
#include <list>
#include <algorithm>
```

```
int main()
{
 std::list<int> seq{ 1,2,3,4 };
 std::for_each(std::begin(seq), std::end(seq), [](int x)
 {
  std::cout << "Polpaarzahl des Asynchronmotors ist:" << ' ';
  std::cout << x << ' ';
  std::cout << '\n';
 });
 std::cout << '\n';
 std::for_each(std::begin(seq), std::end(seq), [](int& x)
 { int a = 3000;
   a/x;
 });
 std::for_each(std::begin(seq), std::end(seq), [](int x)
 {
  int a = 3000;
  std::cout << "Drehfelddrehzahl des Asynchronmotors in 1/min ist:" <<
  a/x << ' ';
  std::cout << '\n';
 });
 std::cout << '\n';
}
```

Listing 1.51 Konsole zum Anzeigen des Programms aus dem Listing 1.50

```
Polpaarzahl des Asynchronmotors ist: 1
Polpaarzahl des Asynchronmotors ist: 2
Polpaarzahl des Asynchronmotors ist: 3
Polpaarzahl des Asynchronmotors ist: 4
Drehfelddrehzahl des Asynchronmotors in 1/min ist:3000
Drehfelddrehzahl des Asynchronmotors in 1/min ist:1500
Drehfelddrehzahl des Asynchronmotors in 1/min ist:1000
Drehfelddrehzahl des Asynchronmotors in 1/min ist:750
C:\Users\eric\source\repos\Funktion4each_for\Debug\Funktion4each_for.
exe (Prozess "17208") wurde mit Code "0" beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

Listing 1.52 Anwendung der *for_each*-Funktion in der Darstellung der Rolle der Lambda-Funktion

```
#include <iostream>
#include <list>
#include <algorithm>
int main()
{
 std::list<int> seq{749,999,1499,2999};
std::for_each(std::begin(seq), std::end(seq), [](int x)
{
 std::cout << x << ' ';
});
std::cout << '\n';
std::for_each(std::begin(seq), std::end(seq), [](int& x)
{
 x++;
});
std::for_each(std::begin(seq), std::end(seq), [](int x)
{
 std::cout << "Drehfelddrehzahl in 1/min ist:"<< x << ' ';
});
std::cout << '\n';
}
```

1.4.3.2.12 Anwendungen der Funktionen *„std::copy()“* und *„std::transform“* zum Darstellen der Parallelisierung

Ein System zu parallelisieren ist dank der Anwendungen der Algorithmen der C++-Standardbibliothek in dem Manipulieren der Elemente eines Vektors oder einer Liste vorteilhaft. Die C++-Standardbibliothek ermöglicht die Darstellung der Parallelisierung dank der Implementierungen der Funktionen wie *„std::copy()“* oder *„std::transform“*.

Listing 1.53 zeigt die Anwendungen der Funktionen *„std::copy()“* und *„std::transform“* zum Darstellen der Rolle der Lambda-Funktion währen des Parallelisierungsprozesses. Beide Funktionen erfüllen den selben Zweck im Lauf der Parallelisierung, aber während die zweite Funktion das Kopieren der veränderten Elemente in eine neue Sequenz ermöglicht, kopiert die erste die Elemente von einem Container zum einem anderen. Gemäß Listing 1.53 zeigt die Wirkung der Iterator-Funktionen *„begin()“* und *„end()“* in der Erstellung der Parallelisierung mithilfe des Einsatzes des Lambda-Ausdrucks in den Funktionen *„copy()“* und *„transform()“*. Gemäß Listing 1.53 nach dem Erstellen eines Vektors werden mehrere Aktionen mithilfe der Anwendungen der Funktionen *„iota()“*, *„for_each()“*, *„copy()“*, *„transform()“* durchgeführt. Dies ermöglicht zuerst das Anzeigen der Elemente des erstellten Vektor aus Ganzzahlen zum Darstellen der Werte der Polpaarzahlen eines Asynchronmotors.

Anschließend wird der Vektor zum Ermöglichen des Manipulierens seiner Elemente vergrößert. Danach werden die Funktionen *§copy()§* und *§transform()§* das Kopieren der Elemente des Vektor bzw. das Veränderung der Elemente mithilfe der Berechnung zum Darstellen der Drehfelddrehzahlen, genannt synchronen Drehzahlen, unter der Wirkung der Lambda-Funktion und der o. g. Iterator-Funktionen implementiert. Listing 1.54 zeigt die Konsole zum Anzeigen der Funktionalität von *„copy()“* und *„transform()“* mithilfe der Iteratoren und Lambda-Ausdrücke. Gemäß Listing 1.54 werden die Polpaarzahlen erstellt. Anschließend werden sie kopiert. Mithilfe dieser Drehzahlen werden synchrone Drehzahlen, genannt Drehfeldrehzahlen, berechnet und danach mithilfe einer Mechanismus der Parallelisierung auf dem Bildschirm angezeigt.

Listing 1.53 Anwendungen der Funktionen *std::copy()* und *std::transform* zum Darstellen der Rolle der Lambda-Funktion

```
/ motor4function.cpp: Definiert den Einstiegspunkt für die Konsolen-
anwendung.
//
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include "stdafx.h"
int main()
{

 const int SIZE = 4;
 std::vector<int> seq(SIZE);
 std::iota(std::begin(seq), std::end(seq), 0);
 std::for_each(std::begin(seq), std::end(seq), [](int x)
 {

  //std::cout << x << ' ';
  std::cout << "Polpaarzahl des Asynchronmotors : " << x << '\n' << ' ';
  std::cout << '\n';

 });
 std::cout << '\n';
 std::vector<int> seq2(SIZE);
 std::copy(std::begin(seq), std::end(seq), std::begin(seq2));
 std::for_each(std::begin(seq2), std::end(seq2), [](int x)
 {

  std::cout << "Kopie der Polpaarzahl des Asynchronmotors : " << x <<
  '\n' << ' ';
```

```
 std::cout << '\n';
});
std::cout << '\n';
std::vector<int> seq3(SIZE);
std::transform(std::begin(seq), std::end(seq), std::begin(seq3), []
(int n)
{
 return 3000 * n;
});
std::for_each(std::begin(seq3), std::end(seq3), [](int x)
{
 std::cout << " Drehfelddrehzahl des Asynchronmotors mithilfe der
 Polpaarzahl und der Frequenz: " << x << " 1/min " << '\n' << ' ';
 std::cout << '\n';
});
std::cout << '\n';
return 0;}
```

Listing 1.54 Konsole zum Anzeigen der Funktionalität von *copy()* und *transform()*

```
Polpaarzahl des Asynchronmotors : 0
Polpaarzahl des Asynchronmotors : 1
Polpaarzahl des Asynchronmotors : 2
Polpaarzahl des Asynchronmotors : 3
Kopie der Polpaarzahl des Asynchronmotors : 0
Kopie der Polpaarzahl des Asynchronmotors : 1
Kopie der Polpaarzahl des Asynchronmotors : 2
Kopie der Polpaarzahl des Asynchronmotors : 3
Drehfelddrehzahl des Asynchronmotors mithilfe der Polpaarzahl und der
Frequenz: 0 Umdrehungen/min
Drehfelddrehzahl des Asynchronmotors mithilfe der Polpaarzahl und der
Frequenz: 3000 Umdrehungen/min
Drehfelddrehzahl des Asynchronmotors mithilfe der Polpaarzahl und der
Frequenz: 6000 Umdrehungen/min
Drehfelddrehzahl des Asynchronmotors mithilfe der Polpaarzahl und der
Frequenz: 9000 Umdrehungen/min
C:\Users\eric\source\repos\Parallellisierung4Application\Debug\
Parallellisierung4Application.exe (Prozess "11448") wurde mit Code "0"
beendet.
Um die Konsole beim Beenden des Debuggens automatisch zu schließen,
aktivieren Sie "Extras" > "Optionen" > "Debuggen" > "Konsole beim
Beenden des Debuggings automatisch schließen".
Drücken Sie eine beliebige Taste, um dieses Fenster zu schließen.
```

1.5 Software-Architektur mit Papyrus und UML-Designer

Die Architektur eines Softwaresystems definiert dieses als Teile zusammen mit den Relationen, die zwischen den Teilen bestehen. Eine Software-Architektur gibt einen Überblick über die Zusammenhänge zwischen den Anforderungen und dem beschriebenen System, wobei die Modellierung der Architektur den ersten Schritt zum detaillierten Systementwurf darstellt. Das Buch fokussiert auf Design Pattern wie z. B. Delegate, Template und Interface zur Modellierung von Systemen u. a. Java, Modularisierung, Schichte-Architektur und Funktionen der Anwendungen.

1.5.1 Modellierung eines Klassendiagramms mithilfe vom Open Source Eclipse-Papyrus zum Analysieren eines Testprogramms mit Junit

1.5.1.1 Modellierung einesTestsystems für die Energietools

Modellierung eines Klassen-Diagramms erzielt die Darstellung der Eigenschaften dieser Klassen bezüglich des Attributs und der Operationen. Dieser Abschnitt fokussiert auf das Konzept *„Codes in Models“* zum Anwenden der Kodierungsinformationen in einem Model des Diagramms.

Das Erstellen des Klassendiagramms aus der Abb. 1.30 basiert auf der Programmierung des Testprogramms aus Listing 1.55, 1.56, 1.57 und 1.58. Abb. 1.30 zeigt die Modellierung des Testsystems mithilfe der Beschreibung des UML-Klassendiagramms mit vier Klassen:

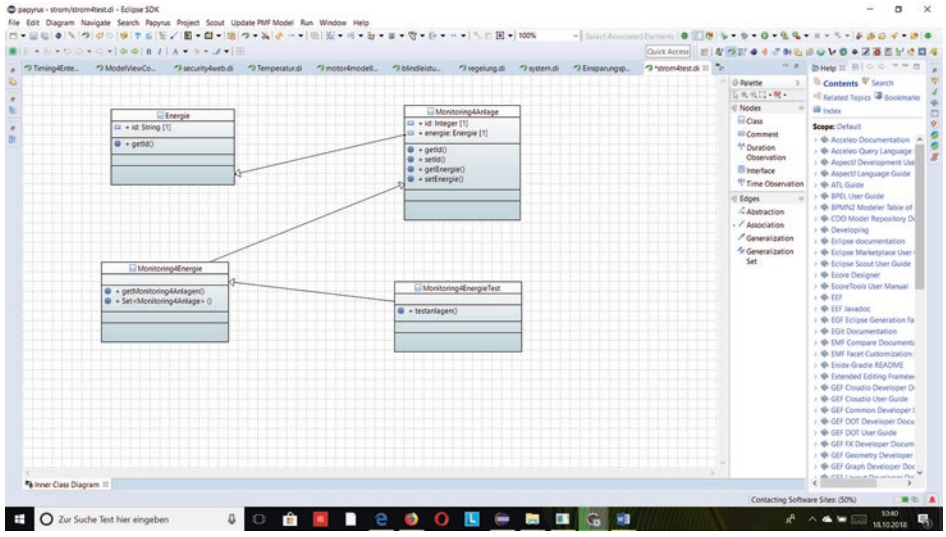


Abb. 1.30 Klassendiagramm auf Basis von Eclipse-Papyrus zum Beschreiben der Zusammenhänge zwischen Klassen des Testprogramms

„Energie“, *„Monitoring4Anlage“*, *„Monitoring4Energie“* und *„Monitoring4EnergieTest“*. Hierbei werden die Verbunde nach der Funktionalität des Systems dargestellt, weil die erste Klasse ihre Funktionalität an die zweite übergibt, die ihre Funktionalität an die dritte Klasse weitergibt. Am Schluss wird die Funktionalität zum Testen des Systems an die vierte Klasse weitergeleitet. Mithilfe der Methode *„getMonitoring4Anlage()“* werden gemäß Listing 1.57 einerseits drei Objekte der Klasse *„Monitoring4Anlage“* instanziiert und andererseits Energie-Tools wie z. B. Solarmodule und Wechselrichter zugeordnet und als unsortierte Gruppe zurückgegeben. Der Systemtest ermöglicht gemäß Listing 1.58 die Analyse des Monitorings der Anlagen für Energie mittels Vergleiches von Inhalten der String-Vertretungen. Hierbei werden mithilfe der Methode *„testanlagen()“* der Klasse *„Monitoring4EnergieTest“* die Elemente der Klasse *„Monitoring4Anlage“* mit dem Aufruf der Methode *„getMonitoring4Anlagen()“* abgefragt. Danach werden diese Elemente zum einen in String-Vertretungen transformiert und zum anderen sortiert zeilenweise in einem String abgespeichert.

Das Testen fokussiert auf den Vergleich des Strings mittels des Schlüsselwort *„assert“* mit Erwartungshaltung zum Illustrieren der Abweichungen zwischen Erwartung und realem Ergebnis während eines negativen Tests mit Junit5. Die Logik des Testsystems mit JUnit5 in der Methode *„testanlagen()“* stellt einerseits die Anwendungen der Lambda-Ausdrücke im Aufruf der Methode *„map()“* und andererseits die Darstellung der funktionalen Programmierung durch die Parallelisierung mithilfe der Aufrufe der Methoden *„stream()“*, *„sorted()“* und *„joining()“* dar. Der Vergleich nach *„Strings“* wird mithilfe des Objektes *„expected“* der Klasse *„StringBuilder“* zum Testen mit dem Aufruf der Methode *„assertNotEqual()“* realisiert. Es ist zu bemerken, dass der Aufruf der Methode *„apppend()“* auf das Objekt *„expected“* der deklarierten Klasse *„StringBuilder“* das Testen darstellt. Gemäß Listing 1.58 wird der Test mit dem Aufruf der Methode *„assertEqual()“* fehlgeschlagen.

Abb. 1.30 Klassendiagramm auf Basis von Eclipse-Papyrus zum Beschreiben der Zusammenhänge zwischen Klassen des Testprogramms.

Listing 1.55 Implementierung der Getter- und Setter-Methoden in der Klasse *Energie*

```
package energie.solar.anwendungen4test;
public class Energie {
 private final String id;
 public Energie(final String id)
 {
  this.id = id;
 }

 public String getId()
 {
  return id;
 }
}
```

Listing 1.56 Implementierung der Klasse *Monitoring4Anlage* mithilfe der Vererbung der Eigenschaft der Klasse *Energie*

```
package energie.solar.anwendungen4test;
public class Monitoring4Anlage {

 private String id;
 private Energie energie;

 public String getId()
 {
  return id;

 }

 public void setId(String id)
 {
  this.id = id;
 }

 public Energie getEnergie()
 {
  return energie;

 }

 public void setEnergie(Energie energie)
 {
  this.energie=energie;
 }
}
```

Listing 1.57 Implementierung der Methode getMonitoring4Anlagen() in der Klasse Monitoring4Energie mithilfe eienrseits der Instanziierung der *Monitoring4Energie*-Objekte und andererseits des Zuordnens der Energien

```
package energie.solar.anwendungen4test;
import java.util.HashSet;
import java.util.Set;
public class Monitoring4Energie {

 public Set<Monitoring4Anlage> getMonitoring4Anlagen()
 {
  Energie solarModule4Energie = new Energie("Solarmodule");
  Energie Wechselrichter4Energie= new Energie("Wechselrichter");
```

```
  Monitoring4Anlage batterie = new Monitoring4Anlage();
  batterie.setId("Batterie");
  batterie.setEnergie(solarModule4Energie);

  Monitoring4Anlage steuerungssystem = new Monitoring4Anlage();
  steuerungssystem.setId("Steuerungssystem");
  steuerungssystem.setEnergie(Wechselrichter4Energie);

  Monitoring4Anlage anwendungssoftware = new Monitoring4Anlage();
  anwendungssoftware.setId("Anwendungssoftware");
  anwendungssoftware.setEnergie(Wechselrichter4Energie);

  Set<Monitoring4Anlage> analysieren = new HashSet<>();
  analysieren.add(batterie);
  analysieren.add(steuerungssystem);
  analysieren.add(anwendungssoftware);
  return analysieren;

 }
}
```

Listing 1.58 Implementierung der Testklasse *Monitoring4EnergieTest* auf Basis von Junit zum Abfragen der Elemente der Klasse *Monitoring4Anlagen*

```
package energie.solar.anwendungen4test;
import java.util.Set;
import java.util.stream.Collectors;
import org.junit.Assert;
import org.junit.jupiter.api.Test;
public class Monitoring4EnergieTest {

 @Test
 public void testanlagen()
 {
  Monitoring4Energie ene = new Monitoring4Energie();
  Set<Monitoring4Anlage> anlagen = ene.getMonitoring4Anlagen();
  String programmieren = anlagen.stream()
    .map(i ->
    i.getId() + "-" + i.getEnergie().getId() + "\n"
    )
    .sorted()
    .collect(Collectors.joining());
  StringBuilder expected = new StringBuilder();
  expected.append("Batterie-Solarmodule\n");
```

```
  expected.append("Steuerungssystem-Wechselrichter4Energie\n");
  expected.append("Anwendungssoftware-Wechselrichter4Energie\n");

  Assert.assertNotEquals(expected, programmieren);

  //Assert.assertEquals(expected, programmieren);

 }
}
```

1.5.1.2 Modellierung einesTestsystems für WLAN-Systeme

Wie in dem Abschn. 1.5.1.1 fokussiert dieser auf die Analyses eines Testsytem auf Basis von Junit5. Die Funktionalität zum Anwenden des Tests in der Sortierung der Elemente der Liste stellt den Vergleich des ersten Elements der Liste vom Datentyp *String* mit dem zweiten vom Datentyp *„Integer“* mithilfe des Aufrufes der Methode *sort* von dem funktionalen Interface *„Comparator“* dar. Mithilfe der Methode *„testCompareByBezeicherThenStromkosten()“* werden zum einen die Liste für WLANSysteme aus den Bezeichern und deren Stromkosten erstellt. Der Aufruf von *„sort()“* auf das Objekt *„wlansystems“* von *„WLANSystem“* zeigt die Anwendungen der *„Extension Methods“* in dem Vergleich zwischen Elementen einer Liste. Diese Methode der Superklasse *„Comparator“* ermöglicht die Aufrufe der Methoden *„comparing()“* und *„thenComparing()“* zum Weiteraufrufen von *„getBezeichner()“* bzw. *„getStromnkosten()“* der Klasse *„WLANSystem“*. Das Testen stellt den Aufruf der Methode *„assertNotEqual()“* mit zwei Parametern dar. Dar erste Parameter ist vom Datentyp String während das zweite Aufrufen von den Methoden *„get(0)“* und *„getBezeichner()“* auf dem Objekt *„wlansystems“* der Klasse *„WLANSystem* darstellt. Die Beziehung oder *Ganzes-Teile-Beziehung* zwischen den Klassen *WLANSystem* und *WLANSystemTest“* gemäß Abb. 1.31 und Listing 1.59 und 1.60 stellt eine spezielle Assoziation genannt Aggregation zum Beschreiben einer *„Hat-Beziehung“* dar. Der Ausdruck zum Deklarieren des Objekts *„wlansystems“* der Klasse *„WLANSystem“* in der Methode *„testCompareByBezeicherThenStromkosten()“* der Test-Klasse mithilfe einer Liste stellt die Beschreibung der Aggregation zwischen dem Teil genannt *„WLANSystemTest“* und dem Ganzes genannt *„WLANSystem“* dar: „List<WLANSystem> wlansystems = **new** ArrayList<>()“.

Es ist zu bemerken, dass sowohl *„WLANSystem*(Ganzes)“ als auch *„WLANSystemTest*(Teil)“ unabhängig voneinander existieren können, wobei das Entfernen der zweiten Klasse nicht die Existenz der ersten Klasse gefährdet. Das Ganze verfügt über die Getter-Methoden *„getBezeichner“* und *„getStromosten“*, deren Aufrufe in der Methode *„testCompareByBezeicherThenStromkosten()“* des Teils mithilfe der Aufrufe von *„comparing()“* und „thenComparing()“ von dem funktionalen Interface *„Comparator“* die Funktionalität der Aggregation zwischen der Ganze-Klasse und der Teil-Klasse darstellen.

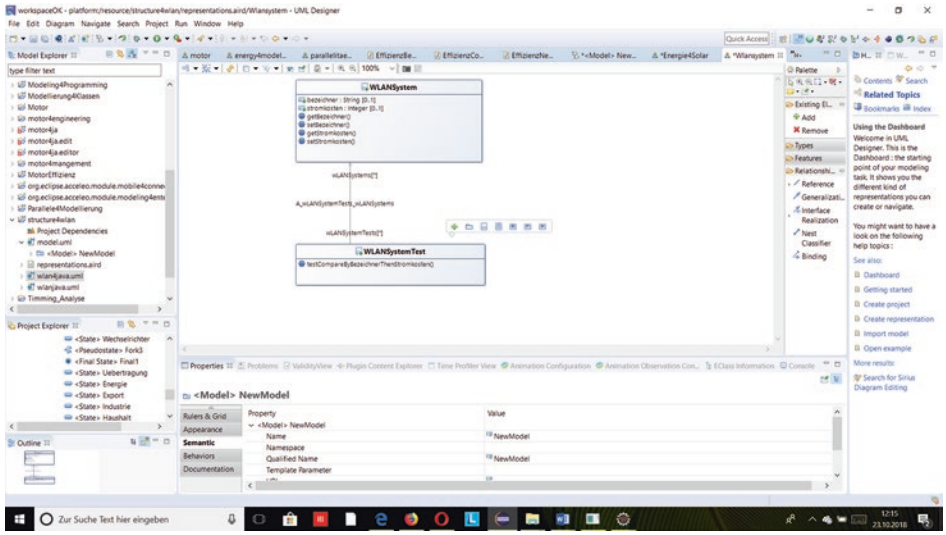


Abb. 1.31 Modellierung eines Klassendiagramms eines Testsystems mit Junit für WLAN-Systeme

Listing 1.59 Darstellung der Komponente der Klasse *WLANSystem* mithilfe der *Getter*- und *Setter*-Methoden

```
package wlan4java.application;
public class WLANSystem {
private String bezeichner;
private int stromkosten;
public WLANSystem() {
super();
}
public WLANSystem(String bezeichner, int stromkosten) {
super();
this.bezeichner=bezeichner;
this.stromkosten = stromkosten;
}
 public int getStromkosten() {
  return stromkosten;
 }
 public void setStromkosten(int stromkosten) {
  this.stromkosten = stromkosten;
 }
 public String getBezeichner() {
  return bezeichner;
 }
```

```
 public void setBezeichner(String bezeichner) {
  this.bezeichner = bezeichner;
 }

 }
```

Listing 1.60 Darstellung der Testklasse *WLANSystemTest* zum Implementieren der Funktionalitäten des funktionalen Interfaces *Comparator* mithilfe des Aufrufes der Methode *sort()*

```
package wlan4java.application;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotEquals;
import java.util.ArrayList;
import java.util.List;
import java.util.Comparator;
import org.junit.Assert;
import org.junit.jupiter.api.Test;
@SuppressWarnings("unused")
class WLANSystemTest {

 @Test
 public void testCompareByBezeichnerThenStromkosten() {
  //fail("Not yet implemented");

  List<WLANSystem> wlansystems = new ArrayList<>();
   wlansystems.add(new WLANSystem("Devolo dLAN 1200+ Wifi ac", 46));
   wlansystems.add(new WLANSystem("Google Wifi", 26));
   wlansystems.add(new WLANSystem("Netgear orbi", 49));
   wlansystems.add(new WLANSystem("TP-Link Deco M5", 31));
   wlansystems.add(new WLANSystem("Ubiquiti Amplifi HD", 36));
    wlansystems.add(new WLANSystem("AWM Fritz!WLAN Repeater 1750E",
  42));

 wlansystems.sort(Comparator.comparing(WLANSystem::getBezeichner).then
Comparing(WLANSystem::getStromkosten));
assertNotEquals("Netgear orbi", wlansystems.get(0).getBezeichner());
//Comparator<WLANSystem> comparator = (wla, wlb) -> wla.
getBezeichner().compareTo(wlb.getBezeichner());
//assertNotEquals("Netgear orbi", wlansystems.get(0).getBezeichner());

    }
}
```

1.5.2 Modellierung des Klassendiagramms zum Beschreiben der parametrisierten Systeme mit Eclipse-Ecore-Framework

Parametrisierte Systeme ermöglichen das Erstellen von Template oder Schablone zum Wiederverwenden von Platzhaltern als Typparameter. Wobei parametrisierte Klassen die Klassen mit Typargumenten darstellen. Beschreibungen der modellierten parametrisiert Klassen gemäß Abb. 1.32 und Listing 1.61, 1.62, 1.63, 1.64, 1.65, 1.66, 1.67, 1.68, 1.69, 1.70, 1.71 und 1.72 stellen die Zusammenhänge zwischen Klassen hinsichtlich der Aufrufe der parametrisierten Methoden aus einer parametrisierten Tochterklasse zum Anzeigen der Eigenschaften der Basisklasse dar. Das Symbol T gemäß Listing 1.67 beschreibt die Struktur der implementierten parametrisierten Klasse zum Darstellen eines Typplatzhalters. T gemäß Listing 1.69 ist der Parametertyp der generischen Klasse *„WLANBox<T>“* und ist als Typplatzhalter einerseits für die Deklaration des Datentyps des Attributs *„daten“* und andererseits für die Deklaration des Typparameters *„Funktion“* in der Methode *„analyse4performance()“* oder für den Ersatz des Datentyps der Methode *„analyse4effizienz()“* wiederverwendet. Wobei die Methode *„analyse-4effizienz()“* über den selben Datentyp wie derer Rückgabe *„tmp“* verfügt: *„T“*. Dieser Typplatzhalter ermöglicht auch den Zugriff auf das privat deklarierte Attribut *„daten“* mithilfe der der Implementierung der Methode *„analyse4effizienz()“*. Wobei das Attribut *„daten“* in der Methode *„analyse4effizienz()“* initialisiert wird. Die Funktionalität der Anwendungen der parametrisierten Klassen in der Darstellung der Beziehungen zwischen Klassen zum Verwenden des Typplatzhalters zeigt Listing 1.70. Hierbei verfügt die Klasse *„Box4Mesh“* gemäß Listing 1.70 über zwei Typparameter: Das erste Typparameter ist T und von der Klasse *„MeshWLANSysteme“* abgeleitet. Wobei diese Basisklasse auch als zwei Typparameter zulässig ist. Gemäß Listing 1.70 ist die Klasse *„MeshWLANSysteme“* die Basisklasse der generischen Klasse „Box4WLAN“. Hierbei verfügt die letzte Klasse über das Typparameter T, das als Typplatzhalter dient. Die Klasse nach dem Schlüsselwort *„extends“* ist nach *„Bound-Klausel“* genannt. Es gibt einen Vorteil bezüglich der Wiederverwendung des Platzhalters T gemäß Listing 1.72 beim Implementieren von Methoden einer generische Klasse; beispielsweise dient *„analyse4effizient()“* und *„analyse4performance()“* aus der generischen Klasse *„Box4WLAN.T“* gemäß Listing 1.68 sowohl als Datentyp der ersten Methode als auch als Datentyp des Parameters, genannt *„Funktion“*, der zweiten Methode der generischen Klasse „*Box4WLAN*.“

Die Vererbungshierarchie gemäß Abb. 1.32, Listing 1.62, 1.63, 1.64, 1.65, 1.66, 1.67, 1.68, 1.69 und 1.70 ermöglicht die Darstellungen der Beziehungen einerseits zwischen den beiden generischen Klasse *„Box4WLAN“* und *„Box4Mesh“* hinsichtlich der Assoziation mithilfe des Aufrufes der Methode *„analyse4performace()“* auf Objekte *„mesh1-7“* der generischen Klasse *„Box4Mesh“* und andererseits zwischen *„MeshWLANSysteme“* und ihren sieben abgeleiteten Klassen hinsichtlich

der Generalisierung mithilfe des Aufrufes des Konstruktors der Mutterklasse *„MeshWLANSysteme“* mit Methode „super()“.

In den sieben Methoden der generischen Klasse *„Box4Mesh“* werden einerseits Objekte *„mesh1-7“* von sieben abgeleiteten Klassen der Oberklasse *„MeshWLANSysteme“* zum Aufrufen der Methode *„analyse4performance()“* der generischen Klasse *„Box4Mesh“* erzeugt, und andererseits werden Objekte *„tool“* der sieben abgeleiteten zum Aufrufen der Methode *„analyse4effizienz()“* von *„Box4Mesh“* auf *„mesh1-7“* erzeugt: *„analyse4lyra()“*, *„anaylse4avm()“*, *„analyse4devolo()“*, *„analyse4google()“*, *„analyse4linksys()“*, *„analyse4netgear()“* und *„analyse4tplink()“*. Die Logik zum Erstellen einer Liste aus technischen Daten von Mesh-WLAN-Systeme zum Darstellen der Bezeichnungen, der Konfigurationen, der Leistungen und der Preise implementieren diese sieben Methoden mithilfe der Schleife *if/else* gemäß Listing 1.68. Mithilfe vom Listing 1.71 werden die Methoden *„analyse4lyra()“*, *„anaylse4avm()“*, *„analyse4devolo()“*, *„analyse4google()“*, *„analyse4linksys()“*, *„analyse4netgear()“* und *„analyse4tplink()“* nochmals in der Hauptklasse *„Anwendung4Analyse“* aufgerufen. Dies ermöglicht das Visualisieren der Inhalte der Liste für WLAN-Systeme mithilfe vom Listing 1.72.

Gemäß Abb. 1.32 und Listing 1.70 und 1.71 werden Beziehungen, genannt Assoziationen, zwischen der generischen Klasse „Box4Mesh“ und den sieben abgeleiteten Klassen der Oberklasse *„MeshWLANSysteme“* dargestellt. Ebenfalls gibt es eine Assoziation zwischen der Hauptklasse *„Anwendung4Analyse“* und der generischen Klasse *„Box4Mesh“*.

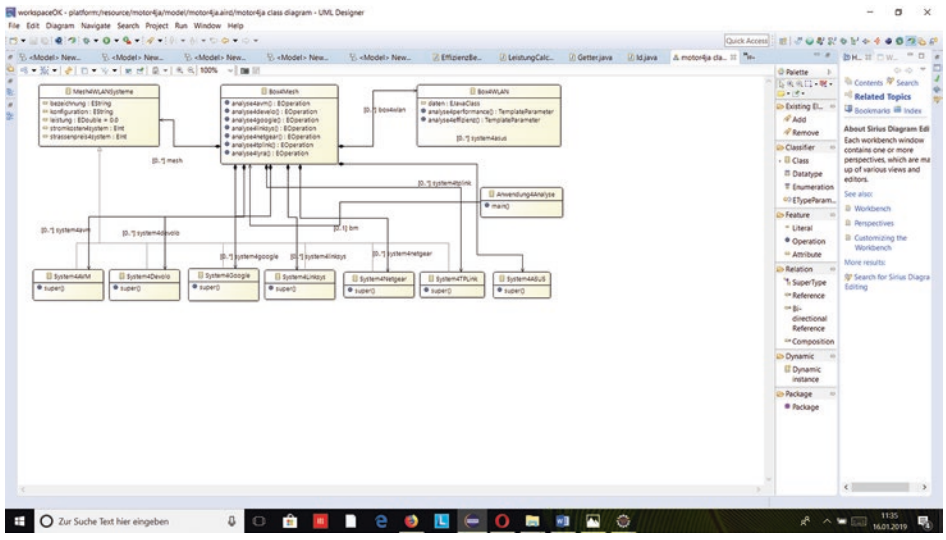


Abb. 1.32 Modellierung des Klassendiagrammes zum Beschreiben der parametrisierten Systeme mit Eclipse-Ecore-Framework

Listing 1.61 Darstellung der Basis-Klasse *MeshWLANSysteme* zum Deklarieren des Konstruktors

```
package wlan4java.parametric.structure.typ;
public class Mesh4WLANSysteme {
 String bezeichnung;
 String konfiguration;
 double leistung;
 int stromkosten4system;
 int strassenpreis4system;

  Mesh4WLANSysteme(String bezeichnung, String konfiguration, double
  leistung, int stromkosten4system,
  int strassenpreis4system) {
   this.bezeichnung=bezeichnung;
   this.konfiguration=konfiguration;
   this.leistung=leistung;
   this.stromkosten4system=stromkosten4system;
   this.strassenpreis4system=strassenpreis4system;
  }
}
```

Listing 1.62 Darstellung der Generalisierung zwischen *System4ASUS* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4ASUS extends Mesh4WLANSysteme{
 System4ASUS(String bezeichnung, String konfiguration, double
 leistung, int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.63 Darstellung der Generalisierung zwischen *System4AVM* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4AVM extends Mesh4WLANSysteme{
 System4AVM(String bezeichnung, String konfiguration, double leistung,
int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
```

```
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.64 Darstellung der Generalisierung zwischen *System4Devolo* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4Devolo extends Mesh4WLANSysteme{
 System4Devolo(String bezeichnung, String konfiguration, double
 leistung, int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.65 Darstellung der Generalisierung zwischen *System4Google* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4Google extends Mesh4WLANSysteme{
 System4Google(String bezeichnung, String konfiguration, double
leistung, int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.66Darstellung der Generalisierung zwischen *System4Linksys* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4Linksys extends Mesh4WLANSysteme{
 System4Linksys(String bezeichnung, String konfiguration, double
leistung, int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.67 Darstellung der Generalisierung zwischen *System4Netgear* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4Netgear extends Mesh4WLANSysteme{
 System4Netgear(String bezeichnung, String konfiguration, double
 leistung, int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.68 Darstellung der Generalisierung zwischen *System4TPLink* und der Basis-Klasse *MeshWLANSysteme*

```
package wlan4java.parametric.structure.typ;
public class System4TPLink extends Mesh4WLANSysteme{
 System4TPLink(String bezeichnung, String konfiguration, double
 leistung, int stromkosten4system,
   int strassenpreis4system) {
  super(bezeichnung, konfiguration, leistung, stromkosten4system,
  strassenpreis4system);
  // TODO Auto-generated constructor stub
 }
}
```

Listing 1.69 Darstellung der Parametrisierung der Klasse *Box4WLAN* mithilfe der Implementierung der Methoden *analyse4Performance()* und *analyse4effizienz()*

```
package wlan4java.parametric.structure.typ;
public class Box4WLAN<T>{
 private T daten;

 void analyse4performance(T funktion) {
  daten=funktion;
 }

 T analyse4effizienz() {
  T tmp = daten;
  daten=null;
  return tmp;
 }
}
```

Listing 1.70 Darstellung der parametrisierten Vererbung mit der Implementierung der parametrisierten Tochterklasse *Box4Mesh<T>* mithilfe der Methoden zum Aufrufen der parametrisierten Methode der Klasse *Box4WLAN<T>*

```
package wlan4java.parametric.structure.typ;
public class Box4Mesh<T extends Mesh4WLANSysteme> extends Box4WLAN<T>
{

 public void analyse4lyra() {

  Box4Mesh<System4ASUS> mesh1= new Box4Mesh<System4ASUS>();
  mesh1.analyse4performance(new System4ASUS("Lyra", "per App/Browser,
  Telnet, SS", 20.8, 55, 400));
  System4ASUS tool =mesh1.analyse4effizienz();
  if(tool == null)
   System.out.println(" Box ist noch nicht konfiguriert");
  else
   System.out.println(" Box von WLAN ist: \n" + " Bezeichnung " +
   tool.bezeichnung + "\n" + " Konfiguration: "
   + tool.konfiguration + "\n" + " Leistung: "
     + tool.leistung + " Watt \n" + " Stromkosten: " + tool.strom-
     kosten4system + " Euro\n"
     + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");

  // TODO Auto-generated method stub
 }

public void analyse4avm() {

  Box4Mesh<System4AVM> mesh2= new Box4Mesh<System4AVM>();
  mesh2.analyse4performance(new System4AVM(" Fritz!WLAN Repeater
  1750E",
    " per Browser, HTTPS ", 15.8, 42, 390));
  System4AVM tool =mesh2.analyse4effizienz();
  if(tool == null)
   System.out.println(" Box ist noch nicht konfiguriert");
  else
   System.out.println(" Box von WLAN ist: \n" + " Bezeichnung: " +
   tool.bezeichnung + "\n"+ " Konfiguration: "
   + tool.konfiguration + "\n" + " Leistung: "
     + tool.leistung + " Watt \n" + " Stromkosten: " + tool.strom-
     kosten4system + " Euro\n"
     + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");
```

```
  // TODO Auto-generated method stub
 }

public void analyse4devolo() {

 Box4Mesh<System4Devolo> mesh3= new Box4Mesh<System4Devolo>();
 mesh3.analyse4performance(new System4Devolo(" dLAN 1200+Wifi ac", "
 per Browser", 17.5, 46, 348));
 System4Devolo tool =mesh3.analyse4effizienz();
 if(tool == null)
  System.out.println(" Box ist noch nicht konfiguriert");
 else
  System.out.println(" Box von WLAN ist : \n" + " Bezeichnung: " +
  tool.bezeichnung + "\n" + " Konfiguration: "
  + tool.konfiguration + "\n" + " Leistung: "
    + tool.leistung + " Watt \n" + " Stromkosten: " + tool.strom-
    kosten4system + " Euro\n"
    + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");

 // TODO Auto-generated method stub
}
public void analyse4google() {

 Box4Mesh<System4Google> mesh4= new Box4Mesh<System4Google>();
 mesh4.analyse4performance(new System4Google("Wifi", "per App", 9.9,
26, 360));
 System4Google tool =mesh4.analyse4effizienz();
 if(tool == null)
  System.out.println(" Box ist noch nicht konfiguriert");
 else
  System.out.println(" Box von WLAN ist : \n" + " Bezeichnung: " +
  tool.bezeichnung + "\n" + " Konfiguration: "
  + tool.konfiguration + "\n"+ " Leistung: "
    + tool.leistung + " Watt\n" + " Stromkosten: " + tool.stromkosten-
    4system + " Euro\n"
  + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");
}
public void analyse4linksys() {

 Box4Mesh<System4Linksys> mesh5= new Box4Mesh<System4Linksys>();
 mesh5.analyse4performance(new   System4Linksys("Velop",   "per   App/
 Browser, HTTPS", 15.1, 40, 486));
 System4Linksys tool =mesh5.analyse4effizienz();
 if(tool == null)
  System.out.println(" Box ist noch nicht konfiguriert");
 else
```

```
  System.out.println(" Box von WLAN ist: \n" + " Bezeichnung: " +
  tool.bezeichnung + "\n" + " Konfiguration: "
  + tool.konfiguration + "\n"+ " Leistung: "
    + tool.leistung + " Watt\n" + " Stromkosten: " + tool.stromkosten-
    4system + " Euro\n"
    + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");
}
public void analyse4netgear() {

 Box4Mesh<System4Netgear> mesh6= new Box4Mesh<System4Netgear>();
 mesh6.analyse4performance(new System4Netgear("Orbi RBK43", "per App/
 Browser, HTTPS", 18.8, 49, 400));
 System4Netgear tool =mesh6.analyse4effizienz();
 if(tool == null)
  System.out.println(" Box ist noch nicht konfiguriert");
 else
  System.out.println(" Box von WLAN ist: \n" + " Bezeichnung: " +
  tool.bezeichnung + "\n" + " Konfiguration: "
  + tool.konfiguration + "\n"+ " Leistung: "
    + tool.leistung + " Watt\n" + " Stromkosten: " + tool.stromkosten-
    4system + " Euro\n"
    + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");
}
public void analyse4tplink() {

 Box4Mesh<System4TPLink> mesh7= new Box4Mesh<System4TPLink>();
 mesh7.analyse4performance(new System4TPLink("Deco M5", "per App",
11.7, 31, 242));
 System4TPLink tool =mesh7.analyse4effizienz();
 if(tool == null)
  System.out.println(" Box ist noch nicht konfiguriert");
 else
  System.out.println(" Box von WLAN ist: \n" + " Bezeichnung: " +
  tool.bezeichnung + "\n" + " Konfiguration: "
  + tool.konfiguration +"\n" +" Leistung: "
    + tool.leistung + " Watt\n" + " Stromkosten: " + tool.stromkosten-
    4system + " Euro\n"
    + " Strassenpreis: " + tool.strassenpreis4system + " Euro\n");
}
}
```

Listing 1.71 Darstellung der Hauptklasse *Anwendungs4Analyse* zum Aufrufen der Methoden der parametrisierten Klasse *Box4Mesh<T>*

```
package wlan4java.parametric.structure.typ;
public class Anwendung4Analyse {
```

```
public static void main(String[] args) {
 // TODO Auto-generated method stub
 Box4Mesh<?> bm = new Box4Mesh<>();
 bm.analyse4lyra();
 bm.analyse4avm();
 bm.analyse4devolo();
 bm.analyse4google();
 bm.analyse4linksys();
 bm.analyse4netgear();
 bm.analyse4tplink();
}}
```

Listing 1.72 Darstellung der Konsole zum Erstellen einer Liste von *WLANSystemen*

```
Box von WLAN ist:
Bezeichnung Lyra
Konfiguration: per App/Browser, Telnet, SS
Leistung: 20.8 Watt
Stromkosten: 55 Euro
Strassenpreis: 400 Euro
Box von WLAN ist:
Bezeichnung: Fritz!WLAN Repeater 1750E
Konfiguration: per Browser, HTTPS
Leistung: 15.8 Watt
Stromkosten: 42 Euro
Strassenpreis: 390 Euro
Box von WLAN ist :
Bezeichnung: dLAN 1200+Wifi ac
Konfiguration: per Browser
Leistung: 17.5 Watt
Stromkosten: 46 Euro
Strassenpreis: 348 Euro
Box von WLAN ist :
Bezeichnung: Wifi
Konfiguration: per App
Leistung: 9.9 Watt
Stromkosten: 26 Euro
Strassenpreis: 360 Euro
Box von WLAN ist:
Bezeichnung: Velop
Konfiguration: per App/Browser, HTTPS
Leistung: 15.1 Watt
Stromkosten: 40 Euro
Strassenpreis: 486 Euro
Box von WLAN ist:
```

```
Bezeichnung: Orbi RBK43
Konfiguration: per App/Browser, HTTPS
Leistung: 18.8 Watt
Stromkosten: 49 Euro
Strassenpreis: 400 Euro
Box von WLAN ist:
Bezeichnung: Deco M5
Konfiguration: per App
Leistung: 11.7 Watt
Stromkosten: 31 Euro
Strassenpreis: 242 Euro
```

1.5.3 Modellierungen der parallelen Implementierungen von Interface

Parallelisierung von Systemen mithilfe der Modellierung des Aktivitätsdiagramms (Abb. 1.33) ermöglicht die Darstellung des Verhaltens des Systems mithilfe sowohl der verschiedenen Aktionen als auch deren Abläufe durch Steuerungsflüsse. Hierbei werden die Funktionalitäten des Systems mithilfe von Aktionen dargestellt. Die Richtungen der Steuerungsflüsse werden in den Aktivitäten mithilfe einerseits der Verzweigungen und andererseits der Gabelungen und Vereinigungen dargestellt. Abb. 1.33 stellt die Modellierung der parallelen Implementierungen der Interfaces

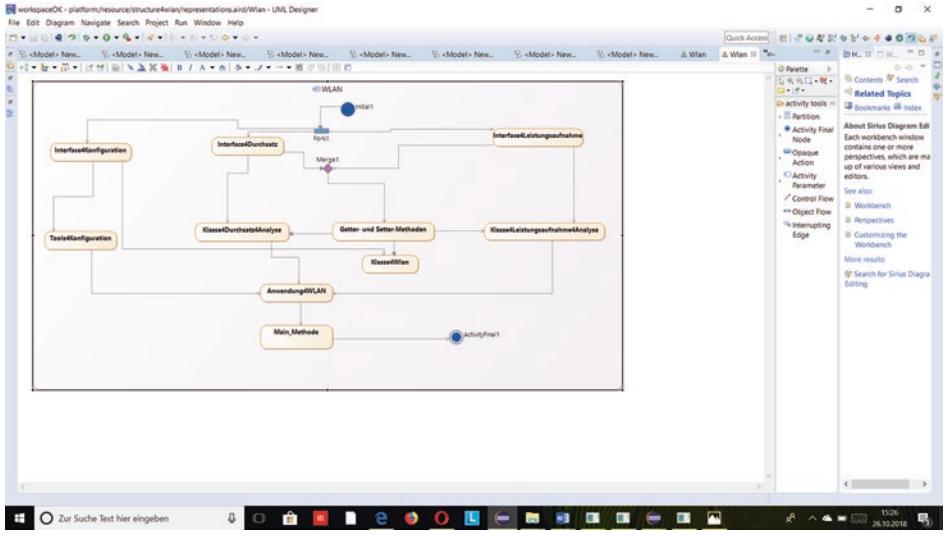


Abb. 1.33 Modellierungen der parallelen Implementierungen von Interface mit dem Aktivitätsdiagram von Eclipse-UML-Designer

„Durchsatz“, *„Leistungsaufnahme“* und *„Konfiguration“* zum Ausdrücken der Steuerungsflüsse mittels Gabelungen in der Aktivität, genannt *„WLAN“*, dar. Gemäß Abb. 1.33 sind die Steuerungsflüsse parallel einerseits mittels „Gabelungen(*Fork*)“ von *„Interface4Konfiguration“* bis *„Tools4Konfiguration“*, von *„Interface4Durchsatz“* bis *„Klasse4Durchsatz4Analyse“* oder von *„Interface4Leistungsaufnahme“* bis *„Klasse4Leistungsaufnahme4Analyse“* und andererseits mittels Vereinigung(*Join*) von *„Tools4Konfiguration“* bis *„Anwendung4WLAN“*, von *„Klasse4Durchsatz4Analyse“* bis *„Anwendung4WLAN“* oder von *„Klasse4Leistungsaufnahme4Analyse“* bis *„Anwendung4WLAN“* ausgeführt. Hierbei stellen die Aktionen sowohl Interface als Klassen gemäß Listing 1.71, 1.72, 1.73, 1.74, 1.75, 1.76, 1.77 und 1.78 und Abb. 1.33 dar. Dies bedeutet, dass der Parallelisierungsprozess gemäß Abb. 1.33 auf das Konzept *„Fork-and-Join“* zum Modellieren der parallelen Steuerungsflüsse mittels des Ausführens der Aktionen durch Gabelungen und Vereinigungen basiert.

Umsetzungen des Konzepts *„Codes in Models“* zeigen Listing 1.73, 1.74, 1.75, 1.76, 1.77, 1.78 und 1.79 zum Darstellen einerseits des Implementierens der Methoden der Interfaces in den abgeleiteten Klassen *„Durchsatz4Analyse“*, *„Leistungsaufnahme4Analyse“* und *„Tools4konfiguration“* und andererseits der Aufrufe der implementierten Methoden der abgeleiteten Klassen in der Hauptklasse genannt *„Anwendung4WLAN“*. Die Klasse *WLAN* gemäß Listing 1.75 und Abb. 1.33 dient zum Erstellen sowohl von Getter- als auch von Setter-Methoden der Interfaces *„Durchsatz“*, *„Leistungsaufnahme“* und *„Konfiguration“*. Dies ermöglicht die Erzeugungen von Standardkonstruktoren mithilfe der Klasse *„WLAN“*.

Die Funktionalitäten zum Implementieren der Parallelisierungen mithilfe der Programmierung gemäß Listing 1.76, 1.77 und 1.79 zeigen die Anwendungen einerseits der Lambda-Ausdrücke in der Erstellung der Liste und andererseits der Aufrufe der Methoden *„forEach()“* in dem Anzeigen der Elemente der *„Liste“*-Objekte und *„sort()“* in dem Einordnen der Elemente der Liste *„list“*. Gemäß Listing 1.76, 1.77 und 1.79 stellt *list* ein Objekt der generischen Klasse „Array*List*“, welche aus einem Typparameter, genannt *„String“*, besteht. Die Funktionalität der Methode *„forEach()“* wird mithilfe der Implementierung der Methoden der abgeleiteten Klassen *„Durchsatz4Analyse“* und *„Leistungsaufnahme4Analyse“* zum Erzeugen der Objekte wie z. B. *„durchsatzclient1-3“* oder *„durchsatz4backbone“* der generischen Klasse *„List“* dargestellt. Aufrufe der Methode „forEach()“ auf diese Objekte ermöglichen die Wirkung der Parallelisierung; beispielsweise des Kopierens der Elemente der Objekte wie z. B. *„durchsat4client1-3“* oder *„durchsatz4backbone“*. Das Einwirken der Lambda-Ausdrücke auf den Parallelisierungsprozess in den Methoden der abgeleiteten Klassen *„Durchsatz4Analyse“* und *„Leistungsaufnahme4Analyse“* z. B. *„durchsatz-4client1()“* bzw. *„Leistung4root()“* wird mithilfe der Erzeugungen von *„Objekten“* des generisch funktionalen Interfaces *„Consumer“*, das aus dem Typparameter *„Double“* besteht realisiert.

Die Parallelität zwischen den Aktionen *„Interface4Durchsatz“* und *„Interface4Leistungsaufnahme“* gemäß Abb. 1.33 und Listing 1.75 stellt die Vereinigung beider Aktionen

mittels des Symbols *„merge“* zum Kapseln der Eigenschaften bezüglich Getter- und Setter-Methoden beider Interfaces in der Klasse *WLAN* dar. Hierbei laufen die parallelen Flüsse aus den Aktionen *„Interface4Durchsatz“* und *„Interface4Leistungsaufnahme“* gemäß Abb. 1.33 zum Bilden der Aktion *„WLAN“* zusammen. Außerdem werden die Steuerungsflüsse mittel vertikalen Parallelitäten orientiert. Beispielsweise ist der vertikale Steuerungsfluss zwischen den Aktionen *„Interface4Konfiguration“* und *„Tools4Konfiguration“* gemäß Abb. 1.33 ist diesem zwischen *„Interface4Durchsatz“* und *„Klasse4Durchsatz“* zueinander parallel. Danach gibt es die vertikale Parallelität zwischen den Steuerungsflüssen von *„Interface4Durchsatz“* bis *„Klasse4Durchsatz“* und von *„Interface4Leistungsaufnahme“* bis *„Klasse4Interface4Leistungsaufnahme“*. Dank der Theorie der Transitivität gibt es die vertikale Parallelität zwischen den Steuerungsflüssen von *„Interface4Konfiguration“* bis *„Tools4konfiguration“* und von *„Interface4Leistungsaufnahme“* bis *„Klasse4Interface4Leistungsaufnahme“*.

Das aufsteigende Einordnen der Elemente in einer Liste gemäß Listing 1.79 wird mithilfe des Aufrufens der Methode *„sort()“* auf die importierte Klasse *„Collections“* ermöglicht. Die Methode *„sort()“* verfügt über das Argument *„list“*, welches ein Objekt der importierten Klasse *„ArrayList“* darstellt. Die Klasse *„ArrayList“* ist eine generische Klasse gemäß Listing 1.79 und besteht aus einem Typparameter *„String“*. Mithilfe der Methode *„add()“* werden Elemente in die Liste eingefügt. Wobei der Zugriff auf das Objekt *„list“* mithilfe des Aufrufes der Methode *„add()“* erfolgt. Gemäß Listing 1.79 sind die Funktionalitäten zum Aufrufen der Methoden *„add()“* und *„sort()“* in der Methode *„analyse4konfiguration()“* der Klasse *„Tools4Konfiguration“* dargestellt.

Funktionalitäten der parallelen Implementierungen der Interfaces *„Durchsatz“*, *„Leistungsaufnahme“* und *„Konfiguration“* in den abgeleiteten Klassen sind mithilfe der Konsole gemäß Listing 1.79 zum Visualisieren sowohl der technischen Daten der WLAN-Systeme als auch der Daten aus acht Marken der Mesh-WLAN-Systeme dargestellt. Wobei die Hauptklasse *„Anwendung4WLAN“* über Aufrufe der Methoden der abgeleiteten Klassen verfügt und das Visualisieren der Parallelisierungsprozesse ermöglicht.

Zustandsdiagramme (Abb. 1.33) werden ähnlich wie Aktivitätsdiagramme zum Darstellen der Parallelisierungsprozesse eingesetzt. Hierbei werden Folgen von Zuständen zum Modellieren des Verhaltens der Objekte mithilfe der Zustandsübergänge, genannt *„Transitions“*, gemäß Abb. 1.34 dargestellt. Abb. 1.34 ist eine Teilmodellierung der Parallelisierung aus Abb. 1.33 und Listing 1.76 und 1.77 bezüglich der Visualisierung der Parallelisierungsprozesse mithilfe des Verhaltens der Zustände zum Beschreiben der Rollen der *„Interfaces“* dargestellt. Bezüglich der Parallelisierung werden aus dem Zustand *„Interface4Konfiguration“* zwei Zustände nämlich *„Analyse4Konfiguration“* und *„Implementierung4Konfiguration“* mittels einer Verzweigung ausgeführt. Die beiden Zustände werden parallel zum Ausführen des Zustandes *„Tools4Konfiguration“* zusammenlaufen. Anschließend führt der letzte Zustand zu dem Zustand *„Array4Produkt“* aus. Am Schluß werden durch Gabelungen und Vereinigungen die Parallelisierungsprozesse mit den Zuständen *„Produkt4Lyra“*, *„Produkt4Fritz!WLAN Repeater 1750E“*, *„Fritz!Powerline 1260E“*, *„dLAN 1200+Wifi ac“*, *„Wifi“*, *„Velop“*, *„Orbi RBK43“*, *„Deco M5“*, und

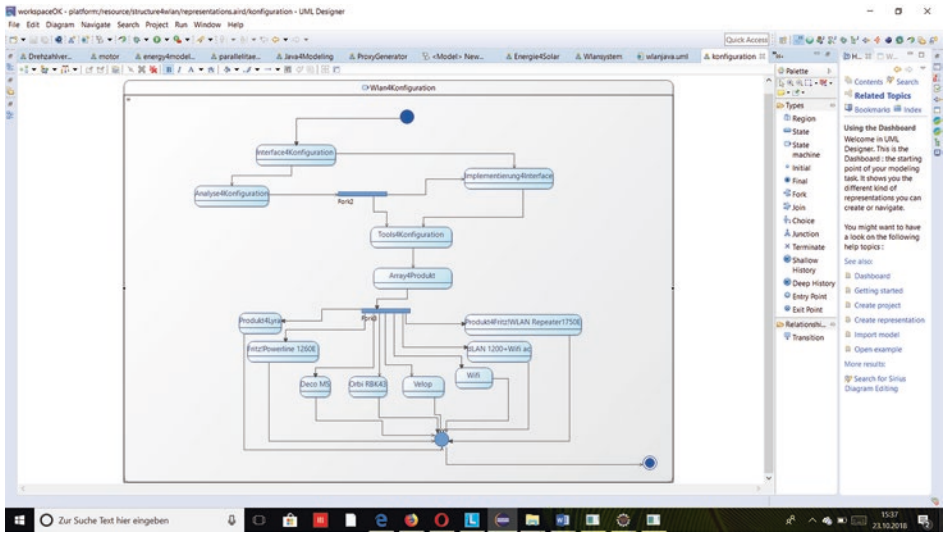


Abb. 1.34 Modellierung der Parallelisierung mit dem Zustandsdiagramm von Eclipse -UML-Designer

„*Amplifi HD*“ dargestellt. Dies ist mithilfe der Parallelisierung der Zustandsübergänge zum Einordnen der Elemente einer bestimmenten Liste mithilfe der Aufrufe der methode *add()* und *sort()* in der Methode *anaylse4konfiguration()* der Klasse *Tools4konfiguration* aus dem Listing 1.77 ermöglicht. Ähnlich wie auf der Abb. 1.33 sind die *Zustände* gemäß Abb. 1.34 auch vertikal zueinander parallel. Zum Veranschaulichen der Parallelisierung der Zustandsübergänge gemä Abb. 1.34 ist der Aufruf der Methode *analyse4konfiguration()* auf das Objekt *t4k* Klasse *Toos4Konfiguration* in der Hauptklasse *Anwendung4WLAN* aus dem Listing 1.80 dargestellt. Hierbei werden Elemente von Arrays mittels der Parallelisierung-Methode *sort()* angezeigt. Im Listing 1.81 ist das Sortieren der Elemente der Liste aus dem den Zustand „*Produkt4Array*“ dargestellt.

Listing 1.73 Implementierung des Interfaces *Durchsatz* zum Darstellen der Getter- du Setter-methode

```
package wlan4java.structure;
public interface Durchsatz {
 static double getDurchsatz4client1(double durchsatz1)
 {
  return durchsatz1;
 }
 static double getDurchsatz4client2(double durchsatz2)
 {
  return durchsatz2;
 }
```

```
 static double getDurchsatz4client3(double durchsatz3)
 {
  return durchsatz3;
 }

 static double getDurchsatz4backbone(double backbone)
 {
  return backbone;
 }

 default void setDurchsatz4client1(double durchsatz4client1)
 {
  this.durchsatz4client1();

 }

 void durchsatz4client1();
 default void setDurchsatz4client2(double durchsatz4client2)
 {
  this.durchsatz4client2();

 }
 void durchsatz4client2();
 default String getHersteller(String hersteller)
 {
  return hersteller;
 }

 default void setDurchsatz4client3(double durchsatz4client3)
 {
  this.durchsatz4client3();

 }

 void durchsatz4client3();
 default void setDurchsatz4backbone(double durchsatz4backbone)
 {
  this.durchsatz4backbone();

 }
 void durchsatz4backbone();

}
```

Listing 1.74 Implementierung des Interface *Leistungsaufnahme* zum Darstellen der Parallelisierung

```
package wlan4java.structure;
public interface Leistungsaufnahme {

 static double getLeistung1(int leistung1)
 {
  return leistung1;

 }

 static double getLeistung2(int leistung2)
 {
  return leistung2;

 }

 static double getLeistung3(int leistung3)
 {
  return leistung3;

 }

 default void setLeistung1(double leistung1)
 {
  this.leistung1();
 }

 public double leistung1();
 default void setLeistung2(double leistung2)
 {
  this.leistung2();
 }

 public double leistung2();
 default void setLeistung3(double leistung3)
 {
  this.leistung3();
 }
 public double leistung3();
}
```

Listing 1.75 Implementierung der Klasse *WLAN* zum Darstellen der technischen Daten eines *MeshWLAN*-Systems mittels Erstellungen der Getter- und Setter-Methoden von den Interfaces *Durchsatz, Leistungsaufnahme* und *Konfiguration*

```
package wlan4java.structure;
import java.util.Optional;
public class WLAN {
private Durchsatz durchsatz;
private Leistungsaufnahme leistungsaufnahme;
private Konfiguration konfiguration;
// standard getters/setters/constructors
public Optional< Konfiguration> getKonfiguration() {
return Optional.ofNullable(konfiguration);
}
 public Durchsatz getDurchsatz() {
  return durchsatz;
 }
 public void setDurchsatz(Durchsatz durchsatz) {
  this.durchsatz = durchsatz;
 }
 public Leistungsaufnahme getLeistungsaufnahme() {
  return leistungsaufnahme;
 }
 public void setLeistungsaufnahme(Leistungsaufnahme leistungsaufnahme)
{
  this.leistungsaufnahme = leistungsaufnahme;
 }
}
```

Listing 1.76 Implementierung der Methoden des Interfaces *Durchsatz* in der Klasse *DurchsatzAnalyse*

```
package wlan4java.structure;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
public class Durchsatz4Analyse implements Durchsatz{
 @Override
 public void durchsatz4client1() {
  // TODO Auto-generated method stub
  List<Double>durchsatz4client1  =Arrays.asList(180.0,  176.0,  100.0,
  173.0, 181.0, 170.0, 101.0);
  Consumer<Double> analysieren1 = x ->
  System.out.println("Client-Durchsatz am Root-Node 2.4 GHz : "
```

```
    + x);
  durchsatz4client1.forEach(analysieren1);
 }
 @Override
 public void durchsatz4client2() {
  // TODO Auto-generated method stub

  List<Double>durchsatz4client2  =Arrays.asList(250.0,  307.0,  182.0,
  336.0, 300.0, 266.0,261.0);
  Consumer<Double> analysieren2 = x ->
  System.out.println("Client-Durchsatz am Root-Node 5 GHz : "
    + x);
  durchsatz4client2.forEach(analysieren2);

 }
 @Override
 public void durchsatz4client3() {
  // TODO Auto-generated method stub
  List<Double>durchsatz4client3  =Arrays.asList(120.0,  122.0,  72.0,
  84.0, 59.0, 100.0, 92.0);
  Consumer<Double> analysieren3 = x ->
  System.out.println("Client-Durchsatz am Root-Node 2.4 GHz : "
    + x);
  durchsatz4client3.forEach(analysieren3);
 }
 @Override
 public void durchsatz4backbone() {
  // TODO Auto-generated method stub
  List<Double>durchsatz4backbone  =Arrays.asList(165.0,  232.0,  178.0,
  195.0, 108.0, 257.0, 194.0);
  Consumer<Double> analysieren4 = x ->
  System.out.println("Backbone-Durchsatz für die Entfernung 20m: "
    + x);
  durchsatz4backbone.forEach(analysieren4);

 }
}
```

Listing 1.77 Implementierung der Methoden des Interfaces *Leistungsaufnahme* in der Klasse *LeistungsaufnahmeAnalyse*

```
ackage wlan4java.structure;
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;
```

```
public class Leistungsaufnahme4Analyse implements Leistungsaufnahme{
 public double leistung1() {
  return getLeistung1();

 }
 private double getLeistung1() {

  return leistung1();
 }
 public void leistung4root()
 {
  List<Double>leistung1 =Arrays.asList(7.2, 9.4, 13.3, 3.5, 5.3, 6.4,
  4.1);
  Consumer<Double> analysieren4leistung1 = x ->
  System.out.println("Leistungsaufnahme für Root in Watt : "
    + x);
  leistung1.forEach (analysieren4leistung1);
 }

 public double leistung2() {
  return getLeistung2();

 }
 private double getLeistung2() {

  return leistung2();
 }
 public void leistung4node()
 {
  List<Double>leistung2 =Arrays.asList(6.8, 3.2, 5.6, 5.7, 3.2, 4.9,
  6.2);
  Consumer<Double> analysieren4leistung2 = x ->
  System.out.println("Leistungsaufnahme für Node in Watt : "
    + x);
  leistung2.forEach (analysieren4leistung2);
 }

 public double leistung3() {
  return getLeistung3();

 }
 private double getLeistung3() {

  return leistung3();
 }
```

```
 public void leistung4system()
 {
  List<Double>leistung3 =Arrays.asList(7.2, 9.4, 13.3, 3.5, 5.3, 6.4,
  4.1);
  Consumer<Double> analysieren4leistung3 = x ->
  System.out.println("Leistungsaufnahme Root in Watt : "
    + x);
  leistung3.forEach (analysieren4leistung3);
 }

}
```

Listing 1.78 Darstellung der Methode *analyse4konfiguration()* im Interface *Konfiguration*

```
package wlan4java.structure;
public interface Konfiguration {
default public void analyse4konfiguration()
{

}
}
```

Listing 1.79 Darstellung der Implementierung der Methode *analyse-4konfiguration()* in der Klasse *Tools4Konfiguration*

```
package wlan4java.structure;
import java.util.ArrayList;
import java.util.Collections;
public class Tools4Konfiguration implements Konfiguration{

 public void analyse4konfiguration()
 {
   System.out.println();

   ArrayList<String> list = new ArrayList<String>();
    list.add(" Bezeichnung: Lyra; Konfiguration: per App/Browser, "
      + " Telnet, SS; Leistung: 20.8 Watt; "
      + " Stromkosten: 55.0 Euro; "
      + " Strassenpreis: 400.0 Euro" + "\n");

    list.add(" Bezeichnung: Fritz!WLAN Repeater 1750E; Konfiguration:
    per Browser, "
      + " HTTPS; Leistung: 15.8 Watt; "
      + " Stromkosten: 42.0 Euro; Strassenpreis: 390.0 Euro " + "\n");
```

```
    list.add(" Bezeichnung: Fritz!Powerline 1260E Konfiguration: per
    Browser, "
      + " HTTPS; Leistung: 23.5 Watt; "
      + " Stromkosten: 62.0 Euro; "
      + " Strassenpreis: 513.0 Euro " + "\n");

    list.add(" Bezeichnung: dLAN 1200+Wifi ac; Konfiguration: per
    Browser;"
      + " Leistung: 17.5 Watt; "
      + " Stromkosten: 46.0 Euro; "
      + " Strassenpreis: 348.0 Euro " + "\n");

    list.add(" Bezeichnung: Wifi, Konfiguration: per App;"
      + " Leistung: 9.9 Watt; "
      + " Stromkosten: 26.0 Euro;"
      + " Strassenpreis: 360.0 Euro " + "\n");

    list.add(" Bezeichnung: Velop; Konfiguration: per App/Browser, "
      + " HTTPS, "
      + " Leistung: 15.1 Watt;"
      + " Stromkosten: 40.0 Watt;"
      + " Strassenpreis: 486.0 Euro " + "\n");

    list.add(" Bezeichnung: Orbi RBK43; Konfiguration: per App/
    Browser, "
      + " HTTPS; Leistung: 18.8 Watt;"
      + " Stromkosten: 49.0 Euro;"
      + " Strassenpreis: 400.0 Euro " + "\n");

    list.add(" Bezeichnung: Deco M5; Konfiguration: per App;"
      + " Leistung: 11.7 Watt; "
      + " Stromkosten: 31.0 Euro;"
      + " Strassenpreis: 242.0 Euro " + "\n");

    Collections.sort(list);
    System.out.println("" + list);

  }
}
```

Listing 1.80 Darstellung der Hauptklasse *Anwendung4WLAN* zum Aufrufen der Methoden der drei Klassen *Durchsatz4Analyse, Leistungsaufnahme4Analyse* und *Tools4konfiguration* mittels der Erzeugungen der entsprechenden Objekte *d4a. la4a* und *t4k*

```
package wlan4java.structure;
public class Anwendung4WLAN
{

 public static void main(String[] args) {
  Durchsatz4Analyse d4a = new Durchsatz4Analyse();
  d4a.durchsatz4client1();
  d4a.durchsatz4client2();
  d4a.durchsatz4client3();
  d4a.durchsatz4backbone();

  System.out.println("-----------------------------------------------
  ------------------");

  Leistungsaufnahme4Analyse la4a = new Leistungsaufnahme4Analyse();
  la4a.leistung4node();
  la4a.leistung4root();
  la4a.leistung4system();

  System.out.println("-----------------------------------------------
  ------------------");

  Tools4Konfiguration t4k = new Tools4Konfiguration ();
  t4k.analyse4konfiguration();

 }
}
```

Listing 1.81 Darstellung der Konsole zum Anzeigen sowohl der technischen Daten der WLAN-Systeme als auch der Daten aus acht Marken der Mesh-WLAN-Systeme

```
Client-Durchsatz am Root-Node 2.4 GHz : 180.0
Client-Durchsatz am Root-Node 2.4 GHz : 176.0
Client-Durchsatz am Root-Node 2.4 GHz : 100.0
Client-Durchsatz am Root-Node 2.4 GHz : 173.0
Client-Durchsatz am Root-Node 2.4 GHz : 181.0
Client-Durchsatz am Root-Node 2.4 GHz : 170.0
Client-Durchsatz am Root-Node 2.4 GHz : 101.0
```

```
Client-Durchsatz am Root-Node 5 GHz : 250.0
Client-Durchsatz am Root-Node 5 GHz : 307.0
Client-Durchsatz am Root-Node 5 GHz : 182.0
Client-Durchsatz am Root-Node 5 GHz : 336.0
Client-Durchsatz am Root-Node 5 GHz : 300.0
Client-Durchsatz am Root-Node 5 GHz : 266.0
Client-Durchsatz am Root-Node 5 GHz : 261.0
Client-Durchsatz am Root-Node 2.4 GHz : 120.0
Client-Durchsatz am Root-Node 2.4 GHz : 122.0
Client-Durchsatz am Root-Node 2.4 GHz : 72.0
Client-Durchsatz am Root-Node 2.4 GHz : 84.0
Client-Durchsatz am Root-Node 2.4 GHz : 59.0
Client-Durchsatz am Root-Node 2.4 GHz : 100.0
Client-Durchsatz am Root-Node 2.4 GHz : 92.0
Backbone-Durchsatz für die Entfernung 20m: 165.0
Backbone-Durchsatz für die Entfernung 20m: 232.0
Backbone-Durchsatz für die Entfernung 20m: 178.0
Backbone-Durchsatz für die Entfernung 20m: 195.0
Backbone-Durchsatz für die Entfernung 20m: 108.0
Backbone-Durchsatz für die Entfernung 20m: 257.0
Backbone-Durchsatz für die Entfernung 20m: 194.0
----------------------------------------------------------------
Leistungsaufnahme für Node in Watt : 6.8
Leistungsaufnahme für Node in Watt : 3.2
Leistungsaufnahme für Node in Watt : 5.6
Leistungsaufnahme für Node in Watt : 5.7
Leistungsaufnahme für Node in Watt : 3.2
Leistungsaufnahme für Node in Watt : 4.9
Leistungsaufnahme für Node in Watt : 6.2
Leistungsaufnahme für Root in Watt : 7.2
Leistungsaufnahme für Root in Watt : 9.4
Leistungsaufnahme für Root in Watt : 13.3
Leistungsaufnahme für Root in Watt : 3.5
Leistungsaufnahme für Root in Watt : 5.3
Leistungsaufnahme für Root in Watt : 6.4
Leistungsaufnahme für Root in Watt : 4.1
Leistungsaufnahme Root in Watt : 7.2
Leistungsaufnahme Root in Watt : 9.4
Leistungsaufnahme Root in Watt : 13.3
Leistungsaufnahme Root in Watt : 3.5
Leistungsaufnahme Root in Watt : 5.3
Leistungsaufnahme Root in Watt : 6.4
Leistungsaufnahme Root in Watt : 4.1
----------------------------------------------------------------
```

```
[ Bezeichnung: Lyra; Konfiguration: per App/Browser, Telnet, SS;
Leistung: 20.8 Watt; Stromkosten: 55.0 Euro; Strassenpreis: 400.0 Euro
, Bezeichnung: Deco M5; Konfiguration: per App; Leistung: 11.7 Watt;
Stromkosten: 31.0 Euro; Strassenpreis: 242.0 Euro
, Bezeichnung: Fritz!Powerline 1260E Konfiguration: per Browser,
HTTPS; Leistung: 23.5 Watt; Stromkosten: 62.0 Euro; Strassenpreis:
513.0 Euro
, Bezeichnung: Fritz!WLAN Repeater 1750E; Konfiguration: per Browser,
HTTPS; Leistung: 15.8 Watt; Stromkosten: 42.0 Euro; Strassenpreis:
390.0 Euro
, Bezeichnung: Orbi RBK43; Konfiguration: per App/Browser, HTTPS;
Leistung: 18.8 Watt; Stromkosten: 49.0 Euro; Strassenpreis: 400.0 Euro
, Bezeichnung: Velop; Konfiguration: per App/Browser, HTTPS, Leistung:
15.1 Watt; Stromkosten: 40.0 Watt; Strassenpreis: 486.0 Euro
, Bezeichnung: Wifi, Konfiguration: per App; Leistung: 9.9 Watt;
Stromkosten: 26.0 Euro; Strassenpreis: 360.0 Euro
, Bezeichnung: dLAN 1200+Wifi ac; Konfiguration: per Browser;
Leistung: 17.5 Watt; Stromkosten: 46.0 Euro; Strassenpreis: 348.0 Euro
]
```

1.5.4 Modellierung der Funktionalitäten der Pattern-Methoden mithilfe des Klassendiagramms von Eclipse-UML-Designer

Template-Pattern ermöglichen die Wiederverwendung von deklarierten Methoden einer abstrakten Klasse zum Implementieren bestimmten Funktionalitäten in einer abgeleiteten Klasse. Die Logik der Implementierung der abstrakten Methoden in der abgeleiteten Klasse stellt auch das Konzept von Schablonen, genannt *„Template“*, sowohl zum Wiederverwenden als auch zum Trennen der Code in implementierten Methoden bezüglich einer Vererbungshierarchie dar.

Das Modellieren dieses Entwurfmusters ist beim Implementieren komplexer Algorithmen zum Verkapseln der Logik in einer einzelnen Methode vorteilhaft.

Abb. 1.35 und Listing 1.82 und 1.83 zeigen die Modellierung der Funktionalitäten der Pattern-Methoden anhand einerseits des Klassendiagramms zum Beschreiben der Vererbungsbeziehungen zwischen den Klassen in der Vererbungshierarchie und andererseits der implementierten Klassen zum Veranschaulichen der Wiederverwendungen der Methoden bezüglich implementierter Funktionalitäten für die Methoden der abgeleiteten Klassen.

Die Logik zum Implementieren der Wiederverwendungen der Template-Methode beruht gemäß Listing 1.83 auf den Implementierungen einerseits des Interfaces *„Liste“* durch die Klasse *„ArrayListe“*, deren Objekt das Aufrufen der Methode *„forEach()“* ermöglicht und andererseits des generisch funktionalen Interfaces *„Consumer“*, dessen Objekt die Wirkung der Lambda-Funktion ermöglicht. Es ist zu bemerken gemäß Listing 1.83, dass sowohl die generische Klasse *„ArrayList“* aus dem Typparameter

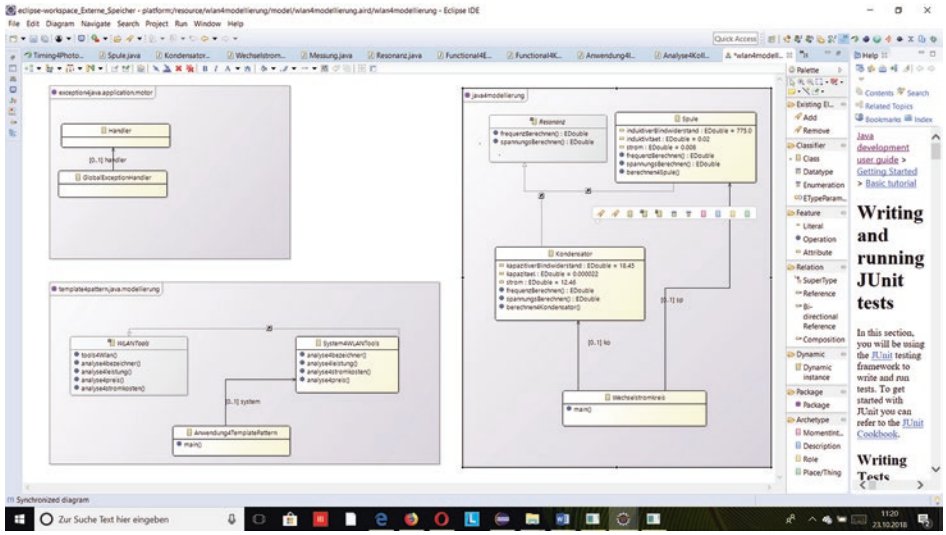


Abb. 1.35 Modellierung der Funktionalitäten der Pattern-Methoden mit dem Klassendiagramm von Eclipse-UML-Designer

„String“ oder *„Double“* als auch das generisch funktionale Interface *„Consumer“* aus dem Typparameter *„String“*, *„Interger“* oder *„Double“* bestehen. Wobei das Funktionsobjekt von *„Consumer“* den Parameter der Methode *„forEach()“* darstellt. Gemäß Listing 1.81 stellt die Methode *„forEach()“* eine wichtige Methode zum Anzeigen der Elemente der WLANSystem-Liste mittels des Konzeptes der Parallelisierung dar.

Gemäß Listing 1.84 werden die abgeleiteten Methoden *„analyse4preis()“*, *„analyse4leistung()“*, *„anylyse4stromkosten()“* und *„analyse4bezeichner()“* in der Klasse *„Anwendung4TemplatePattern“* auf das Objekt „system“ der Klasse *„System4WLANTools“* aufgerufen. Die Konsole aus dem Listing 1.85 stellt das Anzeigen der Elemente der Liste bezüglich der Daten eines WLANSystems dar.

Listing 1.82 Darstellung der abstrakten Basis-Klasse *WLANTools* mit deklariert abstrakten Methoden

```
package template4pattern.java.modelierung;
public abstract class WLANTools {

 public final void tools4Wlan ()
 {
  analyse4bezeichner();
  analyse4leistung();
  analyse4preis();
  analyse4stromkosten();
```

```
 }
 public abstract void analyse4bezeichner();
 public abstract void analyse4leistung();

 public abstract void analyse4preis();
 public abstract void analyse4stromkosten();
}
```

Listing 1.83 Darstellung der abgeleiteten Klasse *System4WLANTools* mit wiederverwendeten und implementierten Methoden aus der abstrakten Klasse *WLANTools*

```
package template4pattern.java.modelierung;
import java.util.Arrays;
import java.util.function.Consumer;
public class System4WLANTools extends WLANTools{

 @Override
 public void analyse4bezeichner() {
  // TODO Auto-generated method stub
  java.util.List<String>    bezeichnung    =    Arrays.asList("Lyra",
  "Fritz!WLAN Repeater 1750E", "Fritz!Powerline 1260E",
    "dLAN 1200+ Wifi ac", "Wifi", "Velop", "Orbi RBK43");
  Consumer<? super String> analysieren4bezeichnung = x ->
  System.out.println("Bezeichnung für Mesh-WLAN-Systeme : "
    + x);
  bezeichnung.forEach (analysieren4bezeichnung);
  System.out.println("-----------------------------------------------
  ---------------");

 }
 @Override
 public void analyse4leistung() {
  // TODO Auto-generated method stub
  java.util.List<Double> leistung = Arrays.asList(6.8, 3.2, 5.6, 5.7,
  3.2, 4.9, 6.2);
  Consumer<Double> analysieren4leistung = x ->
  System.out.println("Leistungsaufnahme für Node : "
    + x + " Watt ");
  leistung.forEach (analysieren4leistung);
  System.out.println("-----------------------------------------------
  -----------------");
```

```
 }
 @Override
 public void analyse4stromkosten() {
  // TODO Auto-generated method stub
  java.util.List<Integer> stromkosten = Arrays.asList(55, 42, 62, 46,
  26, 40, 49);
  Consumer<Integer> analysieren4stromkosten = x ->
  System.out.println("jaehrliche Stromkosten fürs System : "
    + x + " Euro ");
  stromkosten.forEach (analysieren4stromkosten);
  System.out.println("-----------------------------------------------
  ------------------");
 }
 @Override
 public void analyse4preis() {
  // TODO Auto-generated method stub
  java.util.List<Integer> strassenpreis = Arrays.asList(400, 390,
  513, 348, 360, 486, 400);
  Consumer<Integer> analysieren4strassenpreis = x ->
  System.out.println("jaehrlicher Strassenpreis fürs System : "
    + x + " Euro ");
  strassenpreis.forEach (analysieren4strassenpreis);
 }
}
```

Listing 1.84 Darstellung der Hauptklasse *Anwendung4TemplatePattern* bezüglich der Aufrufe der implementierten Methoden aus der abgeleiteten Klasse *System4WLANTools*

```
package template4pattern.java.modelierung;
public class Anwendung4TemplatePattern {
 public static void main(String[] args) {
  // TODO Auto-generated method stub
  System4WLANTools system = new System4WLANTools();
  system.analyse4bezeichner();
  system.analyse4leistung();
  system.analyse4stromkosten();
  system.analyse4preis();
 }
}
```

Listing 1.85 Konsole um Darstellen der Anwendungen des Template-Patterns

```
Bezeichnung für Mesh-WLAN-Systeme : Lyra
Bezeichnung für Mesh-WLAN-Systeme : Fritz!WLAN Repeater 1750E
Bezeichnung für Mesh-WLAN-Systeme : Fritz!Powerline 1260E
Bezeichnung für Mesh-WLAN-Systeme : dLAN 1200+ Wifi ac
Bezeichnung für Mesh-WLAN-Systeme : Wifi
Bezeichnung für Mesh-WLAN-Systeme : Velop
Bezeichnung für Mesh-WLAN-Systeme : Orbi RBK43
------------------------------------------------------------
Leistungsaufnahme für Node : 6.8 Watt
Leistungsaufnahme für Node : 3.2 Watt
Leistungsaufnahme für Node : 5.6 Watt
Leistungsaufnahme für Node : 5.7 Watt
Leistungsaufnahme für Node : 3.2 Watt
Leistungsaufnahme für Node : 4.9 Watt
Leistungsaufnahme für Node : 6.2 Watt
------------------------------------------------------------
jaehrliche Stromkosten fürs System : 55 Euro
jaehrliche Stromkosten fürs System : 42 Euro
jaehrliche Stromkosten fürs System : 62 Euro
jaehrliche Stromkosten fürs System : 46 Euro
jaehrliche Stromkosten fürs System : 26 Euro
jaehrliche Stromkosten fürs System : 40 Euro
jaehrliche Stromkosten fürs System : 49 Euro
------------------------------------------------------------
jaehrlicher Strassenpreis fürs System : 400 Euro
jaehrlicher Strassenpreis fürs System : 390 Euro
jaehrlicher Strassenpreis fürs System : 513 Euro
jaehrlicher Strassenpreis fürs System : 348 Euro
jaehrlicher Strassenpreis fürs System : 360 Euro
jaehrlicher Strassenpreis fürs System : 486 Euro
jaehrlicher Strassenpreis fürs System : 400 Euro
```

1.5.5 Modellierung der Anwendungen des Interfaces *„Collection“* mit dem Klassendiagrammvon Eclipse-UML-Designer

„Collection“ stellt ein wichtiges Interface zum Darstellen sowohl der Speicherung als auch der Verwendung der Objekte dar. Hierbei verfügt das Interface über drei abgeleiteten interface: *„List“*, *„Queue“* und *„Set“*. Gemäß Listing 1.87 sind sowohl die Klassen „HashMap“ und *„LinkedList“* als auch deren Interfaces *„Map“* bzw. *„List“* zum Ermöglichen derer Implementierungen importiert.

Abb. 1.36 stellt die Modellierung der Vererbungsbeziehungen bezüglich der Assoziation zwischen zum einen der Klasse *„WLAN“* und der Klasse *„WLAN4analyse“* und zum anderen der Klasse *„WLAN“* und dem Interface *„WLANFilter“* dar. Der Abb. 1.36 ist die Struktur der Klasse *„WLAN4analyse“* zum Modellieren der Funktionalitäten der Aufrufe der Methoden *„keySet()“*, *„get()“* und *„put()“* mithilfe des Zugriffs auf das Objekt *„system4wlan“* der *„HashMap“*-Klasse zu entnehmen. Die Methode *„put()“* von der Klasse *„HashMap“* ist gemäß Listing 1.87 zum Darstellen der Typparameter der generischen Oberklasse *„Map“*, nämlich *„WLAN“ und „String“*, mithilfe ihrer Aufrufes auf das Objekt *„system4wlan“* in der Methode *„add4WLAN()“* verwendet. Die letzte Methode verfügt über zwei Parameter WLAN und String, deren Zeiger *„w“* und *„funktion“* Parameter von *„put()“* darstellen. Wobei gemäß Listing 1.87 einerseits mithilfe eines Objektes von String, genannt *„aktiveFunktion“*, der Aufruf der Methode „get()“ in der Methode *„add4WLAN()“* ermöglicht wird. Hierbei besteht die Methode „get()“ aus einem Parameter, genannt *„w“*, der ein Zeiger auf WLAN darstellt. In der Methode *„getFunktion()“* mit dem Datentyp *„String“* und dem Parameter *„WLAN“* wird die Methode „get()“ auch auf das Objekt *„system4wlan“* aufgerufen. In der Methode *„getSystem4WLAN()“* von dem Datentyp *„Map<WLAN, String>“* wird der Zugriff auf die vorher deklarierten Klasse ermöglicht. Diese Methode gibt die generische Klasse *„HashMap“* mit dem Parameter das Objekt *„system4wlan“* zurück. Die Methode *„getFunktion4WLAN()“* ist vom Datentyp *„List<WLAN>“* und gibt die Klasse *„LinkedList“* mit dem Parameter der Aufruf der Methode *„keySet()“* auf das Objekt *„wlan4system“* zurück.

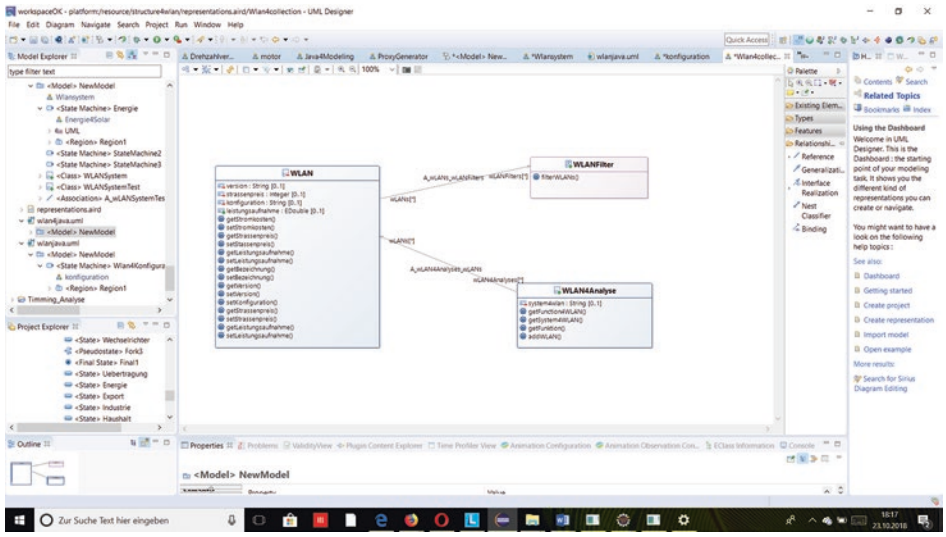


Abb. 1.36 Modellierung der Anwendungen des Interfaces Collection mit dem Klassendiagrammvon Eclipse-UML-Designer

Listing 1.86 Darstellung der Klasse *WLAN* zum Charakterisieren der Funktionalitäten von *WLAN*-Systemen

```
package wlan4java.collection.application;
public class WLAN {
 private String version;
 private int strassenpreis;
 private String konfiguration;
 private double leistungsaufnahme;
 private Hersteller bezeichnung;

 public WLAN(String version) {
  if(version == null) {
   throw new IllegalArgumentException("Invalid Firmware-Version");
  }
  this.setVersion(version);
 }

 public int getStromkosten(int stromkosten) {
  return stromkosten;
 }

 public void setStromkosten(int stromkosten) {
 }

 public int getStrassenpreis(int strassenpreis) {
  return strassenpreis;
 }

 public void setStassenpreis(int strassenpreis) {
  this.setStrassenpreis(strassenpreis);
 }

 public Hersteller getBezeichnung() {
  return bezeichnung;
 }

 public void setBezeichnung(Hersteller bezeichnung) {
  this.bezeichnung=bezeichnung;
 }
 public String getVersion() {
  return version;
 }
 public void setVersion(String version) {
```

```
  this.version = version;
 }
 public String getKonfiguration(String string) {
  return konfiguration;
 }
 public void setKonfiguration(String konfiguration) {
  this.konfiguration = konfiguration;
 }
 public int getStrassenpreis() {
  return strassenpreis;
 }
 public void setStrassenpreis(int strassenpreis) {
  this.strassenpreis = strassenpreis;
 }
 public double getLeistungsaufnahme(double d) {
  return leistungsaufnahme;
 }
 public void setLeistungsaufnahme(double leistungsaufnahme) {
  this.leistungsaufnahme = leistungsaufnahme;
 }

 }
```

Listing 1.87 Darstellung der Klasse *WLAN4Analyse* zum Erstellen der Funktionalitäten von Liste

```
package wlan4java.collection.application;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
public class WLAN4Analyse {
private Map<WLAN, String> system4wlan = new HashMap<>();
public List<WLAN> getFunction4WLAN(){
 return new LinkedList<>(system4wlan.keySet());

}
public Map<WLAN, String> getSystem4WLAN(){
 return new HashMap<>(system4wlan);

}
public String getFunktion(WLAN w) {
 return system4wlan.get(w);

}
```

```
public void addWLAN (WLAN w, String funktion) {
 String aktiveFunktion = system4wlan.get(w);
 if(aktiveFunktion == null) {
  aktiveFunktion = "";
 }

 {
system4wlan.put(w, funktion + aktiveFunktion);
}

}
}
```

Listing 1.88 Darstellung des Interfaces WLANFilter zum Kapseln der Elemente des generischen Interfaces *List*

```
package wlan4java.collection.application;
import java.util.List;
public interface WLANFilter {
 public List<WLAN> filterWLANs(List<WLAN> wlans);

}
```

1.6 Zusammenfassung

SysML und UML ermöglichen die Anwendungen der Modellierung in der Programmierung und stellen damit den neuen Begriff Modeling4Programming dar. Zum einen ermöglichen diese Tools den Entwicklern die Beschreibung der Modellierungsdiagramme mithilfe der standardisierten Annotationen in eine Programmierungssprache wie z. B. Java oder C++. SysML ist eine Erweiterung der Modellierungssprache UML mit den Anforderungen für Spezifikation, Design und Verifikation von komplexen Systemen, wobei SysML ein Teil von UML dient. Diagramme von SysML stellen Struktur-, Verhaltens-, Anforderungs- und Parametermodellierung dar. Die erste Modellierungsart enthält Blockdefinitionsdiagramm (BDD) und internes Blockdiagramm (IBD), die zweite beinhaltet Diagramme für Aktivitäten (Activities), Interaktionen (Interactions), Zustandsmaschinen (StateMachines) und Anwendungsfällen (UseCases) und die dritte stellt die Zuteilung dar.

Eclipse-Papyrus-Framework ermöglicht die Entwicklung von Software mit der Modellierungssprache UML, wobei verschiedene Systemschwerpunkte mithilfe von drei Diagrammarten modelliert werden: Datenstruktur, d. h. statische Strukturen, Systemverhalten, d. h. Dynamik und Systementwurf, d. h. statische Strukturen.

UML-Designer ist ein Open-Source-Tool der Firma Obeo zum Erstellen und Visualisieren der Diagramme auf Basis von UML 2.5. Das Framework wurde auf Basis von der grafischen domänenspezifischen Sprache (DSL) Obeo Sirius entwickelt.

Strukturelle Diagramme beschreiben die Modellierung zeitinvarianter oder unveränderlicher Komponenten von Systemen, wobei diese Komponenten statisch sind. Verhaltenbasierte Diagramme beschreiben die Modellierung dynamischer Komponenten von Systemen.

Das Konzept von *„Modeling4Programming“* stellt die Anwendung der Modellierung in der Programmierung in Beug auf Java und C++ dar. Hierbei werden Anwendungen mithilfe der Modellierung programmiert. C++ ist eine starke Programmierungssprache zur Darstellung von Modellierungstools bezüglich der Übersetzung von Code in Model-Elemente. Das Konzept fokussiert auf der Nutzung der Codes zur Modellierung von IT-Lösungen. Jeder Code ist Teil eines Modellierungsdiagramms wie z. B. Klassendiagramm oder Blockdiagramm entsprechend den Sprachen UML bzw. SysML. IT-Lösungen zu modellieren, ermöglicht das Programmieren der Anwendungen mithilfe sowohl der objektorientierten als auch prozeduralen oder modularen Programmierung. Hierbei spielt der Begriff Funktion eine wichtige Rolle beim Darstellen der funktionalen Programmierung mithilfe von C++. Das Konzept *„Vom Modell zu den Codes“* mithilfe von Java ermöglicht die Beschreibung der Architektur des Diagramms mithilfe der Programmierung. Hierbei wird die Programmiersprache Java zum Beschreiben des Diagrammes angewendet. Mit Java 8 sind die Implementierungen der Methoden im Interface mithilfe des Schlüsselwortes „default“ erlaubt. Dies ermöglicht auch die Darstellungen der Codes in zwei gleichnamigen Methoden aus zwei unterschiedlichen Interfaces zum Vermeiden eines Konfliktes für den Compiler. Es ist auch möglich, zwei gleichnamige Methoden in zwei unterschiedliche Interfaces zu implementieren. Aber dies soll nicht eine Kollision zwischen den Interfaces verursachen. Die Programmiersprache Java mit der Version 8 nutzt die Lambda-Ausdrücke zum Darstellen der funktionellen Programmierung. Das Ziel ist es, die Lambda-Funktionen zum Beschreiben der mathematischen Aspekte der Programme mithilfe der Algorithmen anzuwenden. Hierbei werden Programme mithilfe von mathematischen Zielen ohne objektorientierte Gedanken entworfen. Mithilfe funktioneller Programmierung werden Funktionsobjekte anstatt Funktionstypen in Fokus der Programmierung mit dem Ziel, starker Programme zu entwickeln, verwendet.

Modellierung der Funktionalität einer Berechnung stellt einerseits die Analyse der Parameter der Funktion bezüglich des Datentyps und andererseits die Bedeutung des Ausdrucks zum Funktionsaufruf dar. Funktionelle Modellierung wird mithilfe der Analyse der Implementierung der Werkzeuge der C++-Standardbibliothek bezüglich der Themen wie z. B. Ein-/Ausgabe-, String-, Container- oder Iterator-Bibliothek realisiert. Im Bereich funktionale Modellierung ist die Anwendungen der Lambda-Ausdrücke zur Berechnung der mathematischen Formel in Java sehr hilfreich, weil diese Berechnungen mithilfe von Funktionen durchgeführt werden. Die Modellierung der Parallelisierungsprozesse mithilfe der C++-Standardbibliothek ermöglicht die Anwendungen der verschiedenen Tools wie Algorithmen, Vektor oder Lambda-Ausdrücke in der parallelen Darstellung der Sequenzen.

Ein System zu parallelisieren, ist dank der Anwendungen der Algorithmen der C++-Standardbibliothek in dem Manipulieren der Elemente eines Vektors oder einer Liste vorteilhaft. Die C++-Standardbibliothek ermöglicht die Darstellung der Parallelisierung dank der Implementierungen der Funktionen wie *„std::copy()“* oder *„std::transform“*.

Parallelisierung von Systemen mithilfe der Modellierung des Aktivitätsdiagramms ermöglicht die Darstellung des Verhaltens des Systems mithilfe sowohl der verschiedenen Aktionen als auch deren Abläufe durch Steuerungsflüsse. Hierbei werden die Funktionalitäten des Systems mithilfe von Aktionen dargestellt.

Literatur

1. Object Management Group System Modeling Language (OMG SysML), www.omgsysml.org, (2019).
2. Kecher, C., Salvanos, A.: UML 2.5: Das umfassende Handbuch, Rheinwerk Verlag GmbH, Bonn, (2015).
3. Zeller, M., Weiß, G.: „Modellgetrieben Absicherung von Softwareschnittstellen für vernetze eingebettete Systeme“, in: „Embedded Software“ iX Developer, Heise Zeitschriften Verlag GmbH, Februar (2014).
4. Louis, D., Müller, P., Java: der umfassende Programmierkurs, O'Reilly Verlag, Köln (2014).
5. Oestereich, B.: Analyse und Design mit UML 2, Oldenbourg Wissenschaftsverlag, ISBN 3-486-57926-6, (2006).
6. Kecher, C.: UML 2.0: Das umfassende Handbuch, Galileo Computing, ISBN 3-89842-738-2 (2006).
7. Darstellung der Anwendungsfallspezifikation mit IBM Rational in: Anwendungsfalldiagramme erstellen, IBM Knowledge Center, https://www.ibm.com/support/knowledgecenter/de/SSYMRC_6.0.2/com.ibm.rational.rrm.help.doc/topics/r_uc_spec_outline.html (2019).
8. Lucichart: Was ist ein UML-Diagramm, Web Portal, Lucid Software Inc, https://www.lucidchart.com/pages/de/was-ist-ein-uml-diagramm (2019).
9. Takai, D.: Qualitätsmerkmale von Websystemen, Javamagazin, Verlag Software & Support Media GmbH, Juli (2015).
10. Scheibls Portalseite: HTW Berlin, http://user-old.f1.htw-berlin.de/Scheibl/UML/index.htm?./Profildiagramm/Grundlagen.htm (2015).
11. Verteilungsdiagramme mit IBM Rational in: Verteilungsdiagramme erstellen, IBM Knowledge Center, Web Portal https://www.ibm.com/support/knowledgecenter/de/SS8PJ7_9.6.0/com.ibm.xtools.modeler.doc/topics/cdepd.html (2019).
12. Herbrechtsmeier, S.: „Modell eines agilen Leiterplattenentwurfsprozesses basierend auf der interdisziplinären Entwicklung eines modularen autonomen Miniroboters“, Dissertation, Technische Fakultät der Universität Bielefeld, 21. Dezember (2016).
13. Alt, O.: System- und Softwareentwicklung mit SysML und UML für eingebettete Systeme, „Embedded Software“, Developer, Heise Zeitschriften Verlag GmbH & Co. KG, Februar (2014).
14. Hagmann, G.: Leistungselektronik: Gundlagen und Anwendungen, AULA-Verlag, Wiesbaden (1993).
15. Breymann, U.: Der C++-Programmierer, Carl Hanser Verlag München (2015).
16. Kommunikationsdiagramm mit IBM Rational, in: Kommunikationsdiagramm erstellen, IBM Knowledge Center, Web Portal https://www.ibm.com/support/knowledgecenter/de/SS8PJ7_9.6.1/com.ibm.xtools.sequence.doc/topics/ccommndiag.html (2019).
17. Schader, M., Kuhlins, S.: Programmieren in C++: Einführung in den Sprachstandard, 5.Auflage, Springer-Verlag (1998).

UML-Modellierung mit der Eclipse-Umgebung

2

Unified Modeling Language abgekürzt UML stellt die Standardmodellierungssprache für Softwaresysteme dar, wobei diese Sprache, von der ObjectManagement Group entwickelt wurde. Das Buch gibt Überblicke in Bezug auf den Eclipse-Editor über Modellierung der Systemaspekte und objektorientierte Modellierung. Hierbei werden z. B. für das Klassendiagram einige Objektstrukturen modelliert, da es einen Unterschied zwischen Klassen von Objektdiagrammen gibt. Das Kapitel fokussiert auf die Thematik *„Modeling4Programming“* und gibt einen Überblick über das Tool *„Codes in Models“* bezüglich der Beziehung zwischen UML-Modellen und Java-Klassen. Editoren von Obeo-UML-Designer und Papyrus werden mithilfe des Eclipse-Frameworks heruntergeladen. Eclipse-Framework ermöglicht den Entwicklern den Zugang zu den Modellierungsplattformen UML-Designer und Papyrus.

2.1 Modellierung des Klassendiagramms mit Obeo-UML-Designer

Ein Klassendiagramm stellt die statische Struktur eines Systems oder einer Software dar und beinhaltet den Kern des Analysemodells. Dieses Diagramm verfügt über Teile wie z. B. Paket, Objektklasse, Attribut, Operation zur Modellierung [1].

Mithilfe des Frameworks Obeo-UML-Designer ist die Modellierung einfach und effizient. Der Editor dieses Frameworks wurde auf Basis der DSL Sirius 5 entwickelt (www.obeo.designer). Abb. 2.1, 2.2, 2.3, 2.4, 2.5, 2.6 und 2.7 geben Überblicke über den Mechanismus zur Erstellung eines Klassendiagramms. Hierbei erfolgt der Prozess mit der Erstellung eines UML-Projektes mithilfe des Eclipse-Editors (z. B. Version Oxygen für diesen Abschnitt). Abb. 2.1, 2.2, 2.3 und 2.4 zeigen die Erstellung der UML-Model-Projekte, wobei Abb. 2.1 und 2.2 den Inhalt des Obeo-Designer-UML bzw. die Eingabe über den Projektnamen und den Speicherort des UML-Projektes

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML,*
https://doi.org/10.1007/978-3-658-29057-3_2

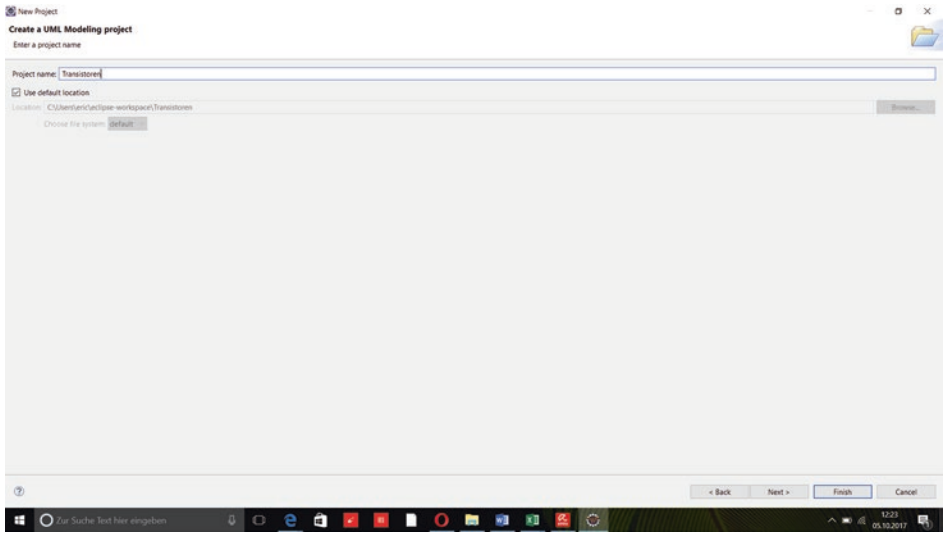


Abb. 2.1 Erstellung der UML-Model-Projekt

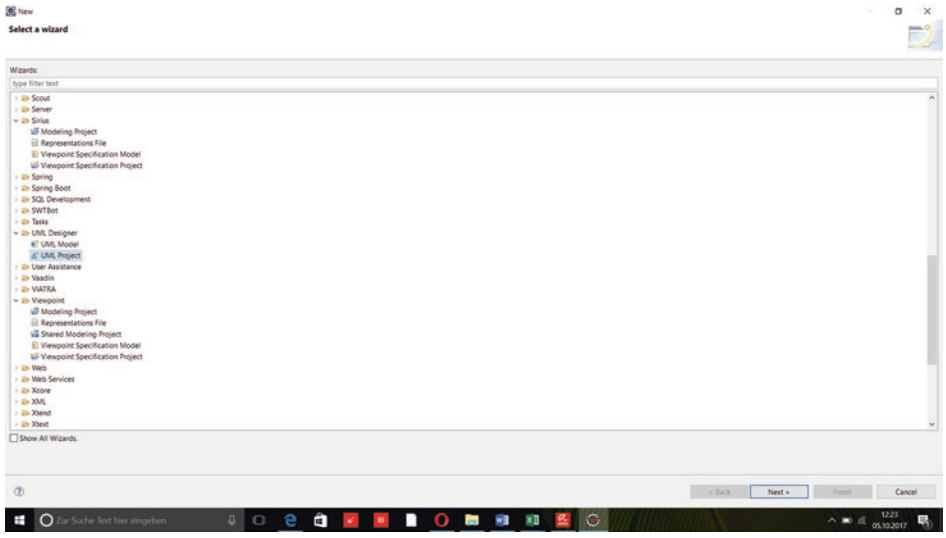


Abb. 2.2 Inhalt des Obeo-Designer-UML

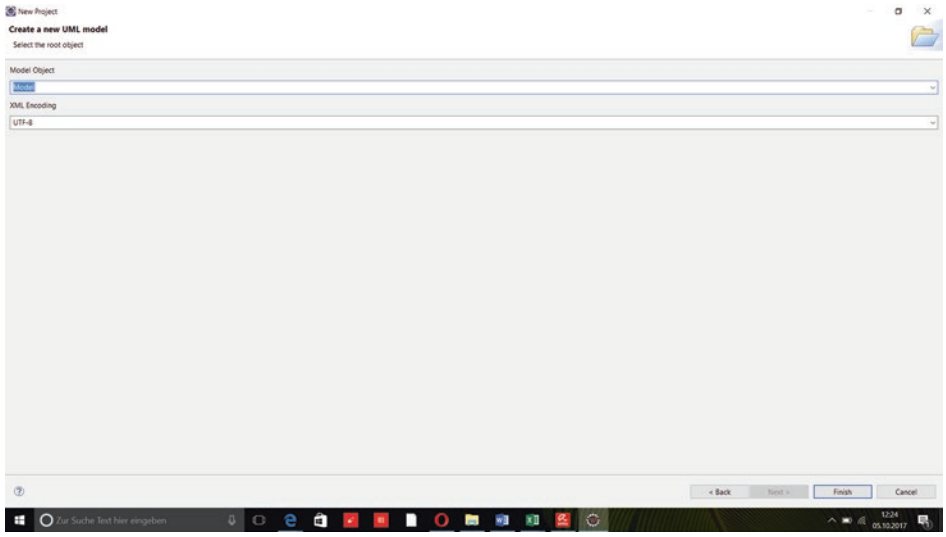


Abb. 2.3 Einsatz von *„root object“* auf *„model“* während der Erstellung von *„uml model“*

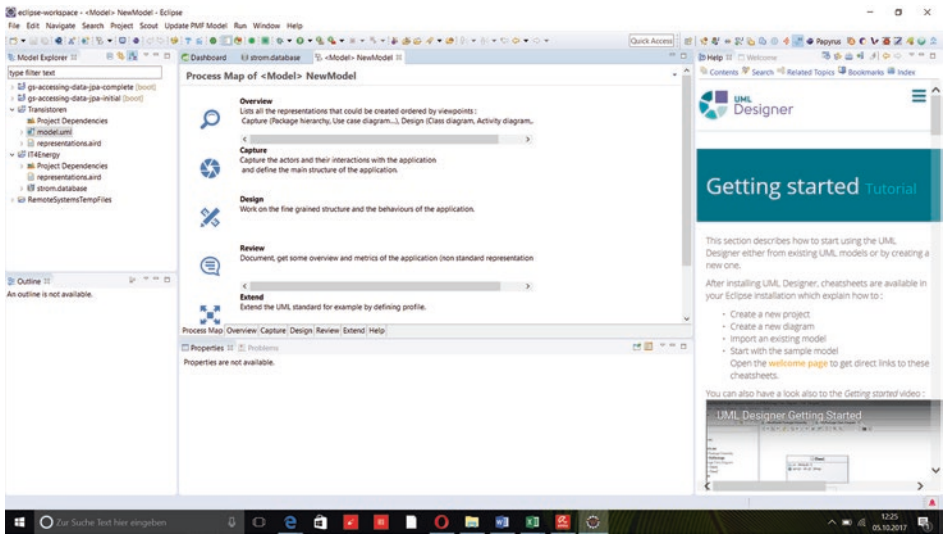


Abb. 2.4 Überblick über Projekt-Explorer in Bezug auf den Projektname *„Transistoren“*

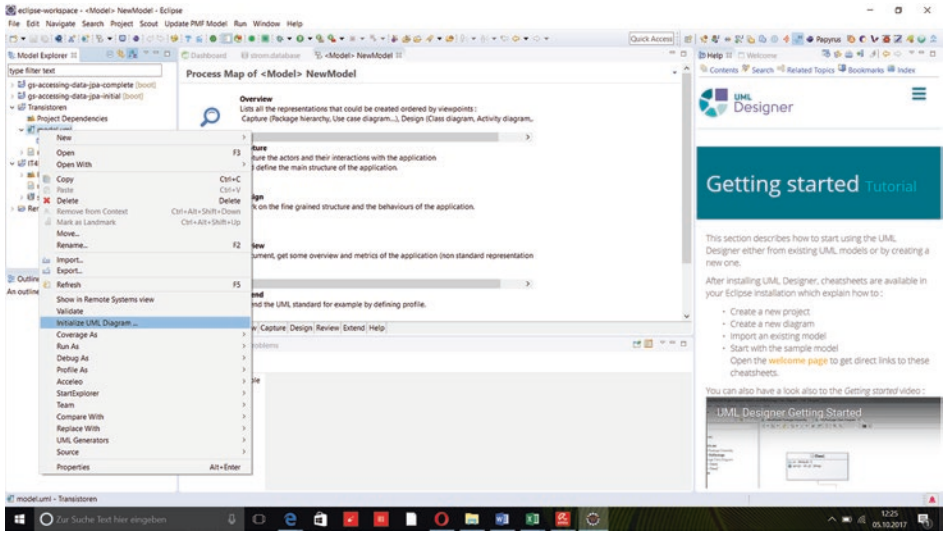


Abb. 2.5 Darstellung der Initialisierung der Klassen-Diagramme im „*Model-Explorer*“

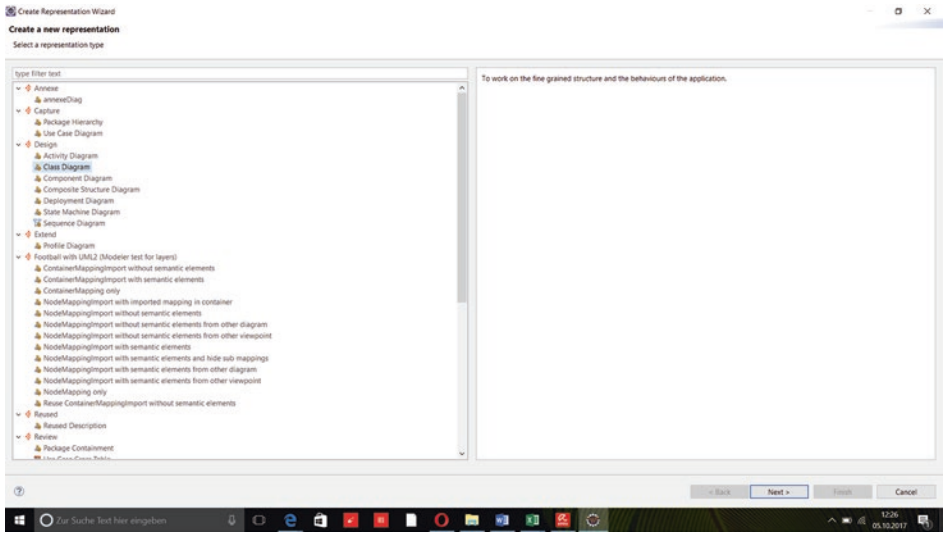


Abb. 2.6 Überblick über Komponente der UML-Diagramme

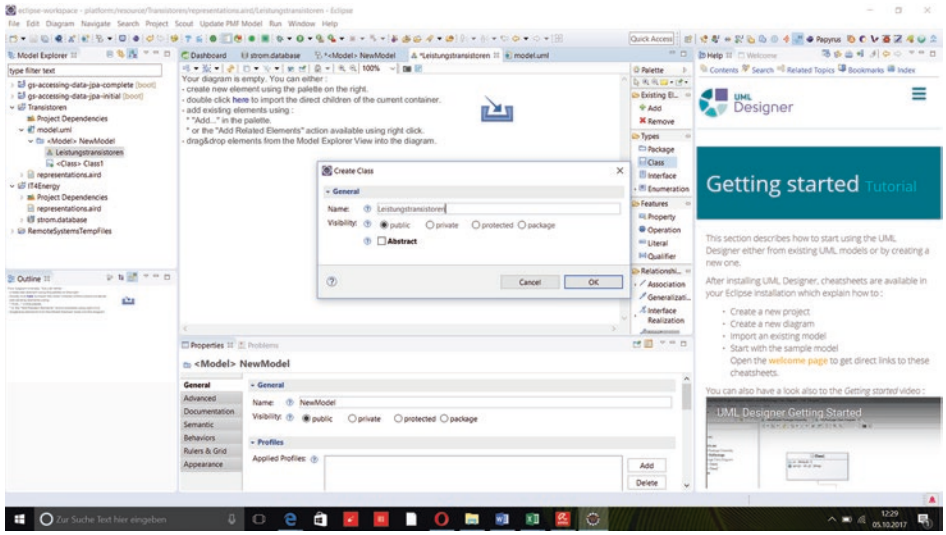


Abb. 2.7 Überblick über Inhalte von Paletten

zeigen. Mit Abb. 2.3 ist das *„root object“* auf *„model“* während der Erstellung von *„uml model“* gesetzt, damit *„Model Diagramme“* erstellt werden. Mit der Abb. 2.4 wird eine Mappe zur Modellierungskomponente erstellt, wobei die Mappe verschieden Elemente wie z. B. *„Design“*, *„Capture“*, *„Overview“*, *„Review“* und *„Extend“* beinhaltet. Die UML-Diagramme befinden sich in der Komponente *„Design“*. Auf der Abb. 2.4 ist in Projekt-Explorer der Projektname *„Transistoren“* mit seinen Inhalten wie z. B. *„representations.air“*, *„model.uml“* und *„Project Dependencies“* zu sehen. Abb. 2.5 stellt die Initialisierung der Klassen-Diagramme im *„Model-Explorer“* dar. Abb. 2.6 gibt Überblicke über Inhalte von *„Mappe Process“* mit Hinblick auf *„representations.air“*. Hierbei sind Inhalte von Verschiedenen Komponenten bei *„representations.air“* wie z. B. *„Design“*, mit UML-Diagrammen zu sehen. Auf der Abb. 2.6 sind sieben UML-Diagramme der Komponente *„Design“* zu sehen: *„Activity Diagram“*, *„Class Diagram“*, *„Component Diagram“*, *„Composite Structure Diagram“*, *„Deployment Diagram“*, *„State Machine Diagram“* und *„Interaction Diagrams“*. Abb. 2.7 gibt einen Überblick über die Erstellung der Kasse *„Leistungstransistoren“*. Auf der Abb. 2.7 ist rechts einen Werkzeugkasten, genannt *„Palette“*, mit 4 Gruppen von Komponenten wie z. B. *„Existing Elements“*, *„Type“*, *„Features“* und *„Relationships“*. Auf der Abb. 2.7 enthält die erste Komponente *„Package“*, *„Class“*, *„Interface“* und *„Enumeration“* während Komponenten über *„Features“* Elemente wie z. B. *„Property“*, *„Operation“*, *„Litteral“* und *„Qualifier“* verfügen. Hierbei ist anzumerken, dass *„Property“* und *„Operation“* Attribute bzw. Methode einer Klasse oder Interface darstellen. Abb. 2.8 und 2.9 geben einen Überblick über die Klasse und ihre Attribute. Abb. 2.8 zeigt das Editieren der Attribute *„Typ“* in der

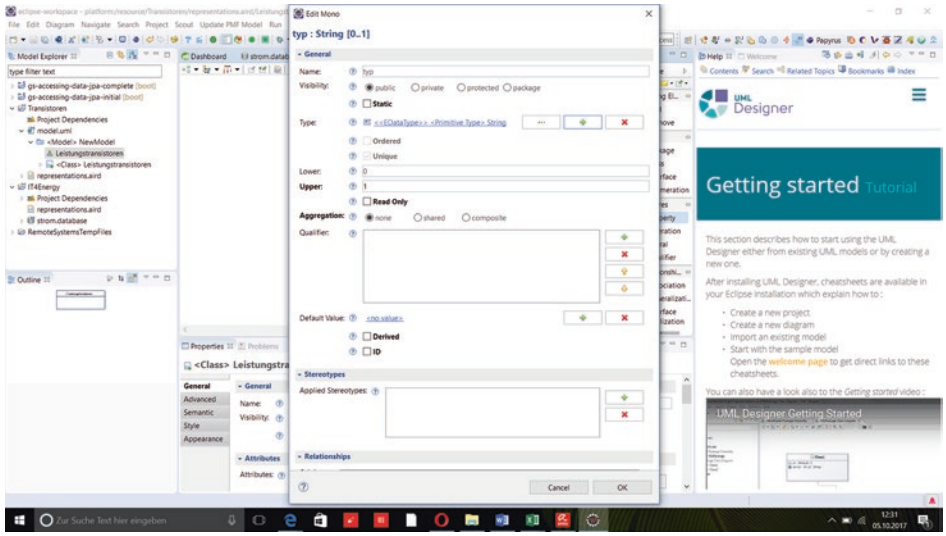


Abb. 2.8 Das Editieren der Attribute „Typ“ in der Klasse „Leistungstransistor“

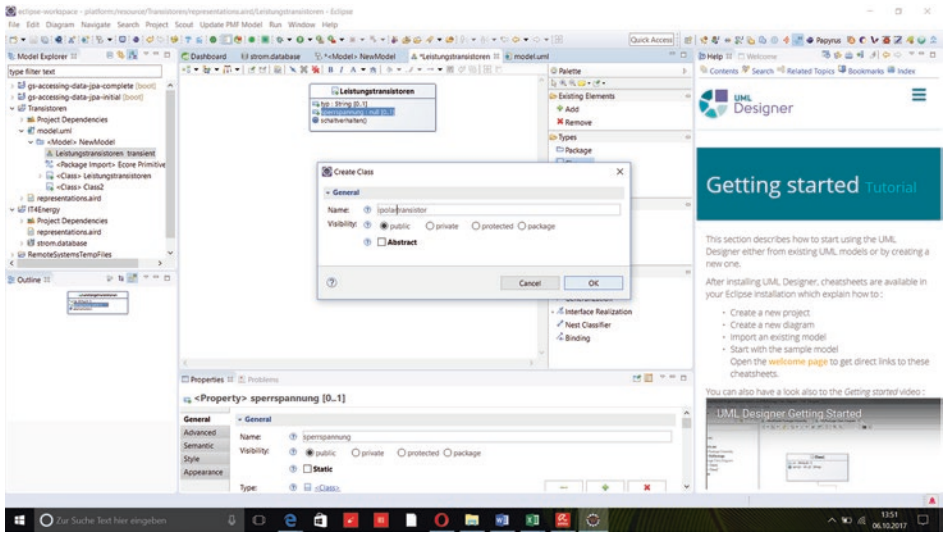


Abb. 2.9 Attributen und Methode der Klasse „*Leistungstransistor*“

Klasse *„Leistungstransistor“* in Bezug auf Namen, Sichtbarkeit, Typ und Assoziationen. Zum einem ist auf der Abb. 2.8 die Struktur dieser Attribute zu sehen und zum anderen sind auf der Abb. 2.9 Attributen und Methode der Klasse *„Leistungstransistor“* editiert worden. Eine Klasse wird mit einem Rechteck dargestellt, wobei der Name der Klasse angezeigt wird. Das Klassensymbol kann mit einem Bereich zur Darstellung der Attribute und der Methoden erweitert werden. Für parametrierbare Klassen wird das Klassensymbol mit einem Rechteck erweitert, das den Namen des Parameters enthält. Die Funktionalität einer Klasse ist in Methoden abgelegt. Eine Methode besteht aus einer Signatur und einem Rumpf, der die Implementierung beschreibt [2].

2.1.1 Vererbung

Klassendiagramme können auch Vererbungsbeziehungen darstellen [3]. Abb. 2.10, 2.11, 2.12, 2.13 und 2.14 zeigen die Beziehungen zwischen der Mutterklasse *„Leistungstransistor“* und den Tochterklassen *„Bipolartransistor“*, *„MOSFET“* und *„IGBT“* (Insulated Gate Bipolar Transistor) in Bezug auf die Generalisierung, welche eine Beziehung zwischen einer generellen und einer besonderen Klasse in Bezug auf Attribute und Operation darstellt. Auf den Abb. 2.12 und 2.13 ist einen hierarchischen Vererbungsbaum zu sehen. Es ist anzumerken, dass der Leistungstransistor sowohl Bipolartransistor als auch *MOSFET* und *IGBT* sein kann. Die Eigenschaften der Mutterklasse werden an die entsprechenden Tochterklasse weitergegeben, d. h. vererbt. Eine Tochterklasse verfügt

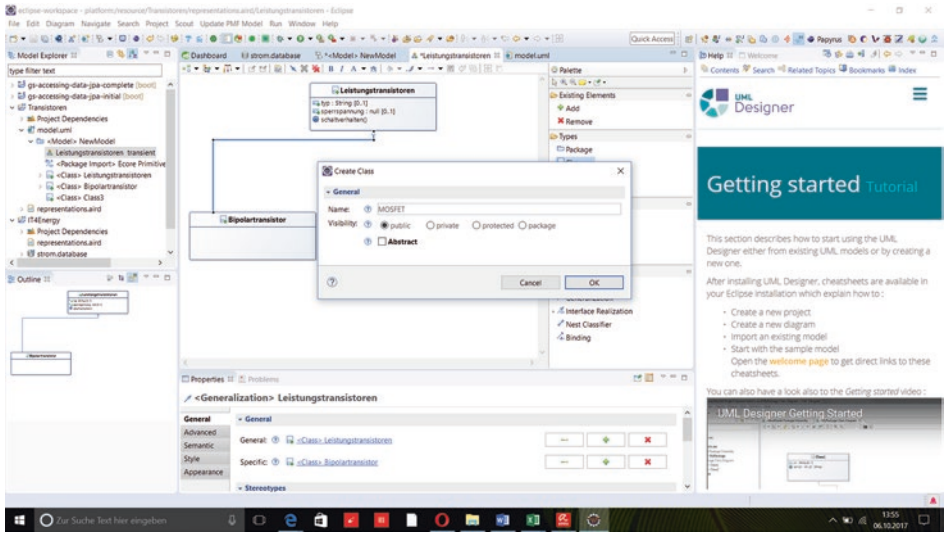


Abb. 2.10 Überblick über die Mutterklasse und ihre Tochter in Bezug auf Generalisierung

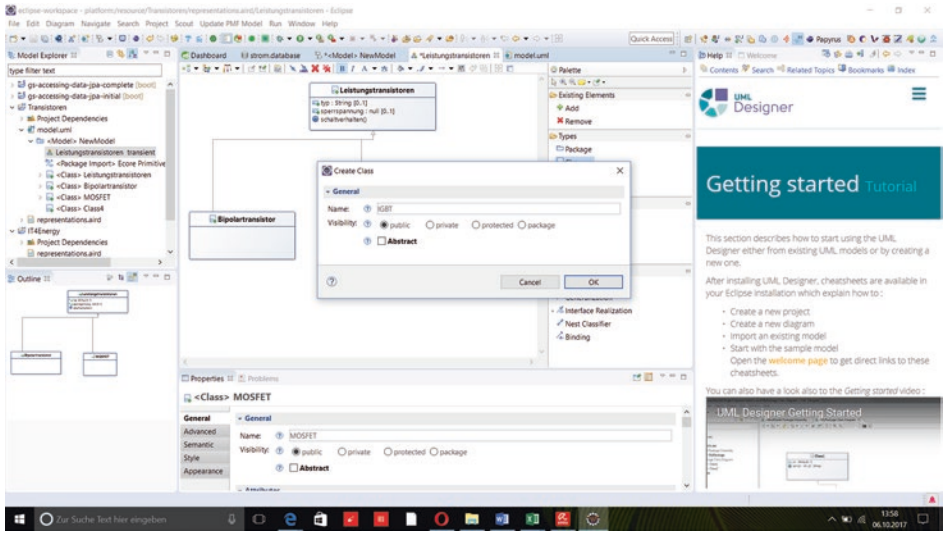


Abb. 2.11 Baumstruktur zur Vererbungsbeziehung

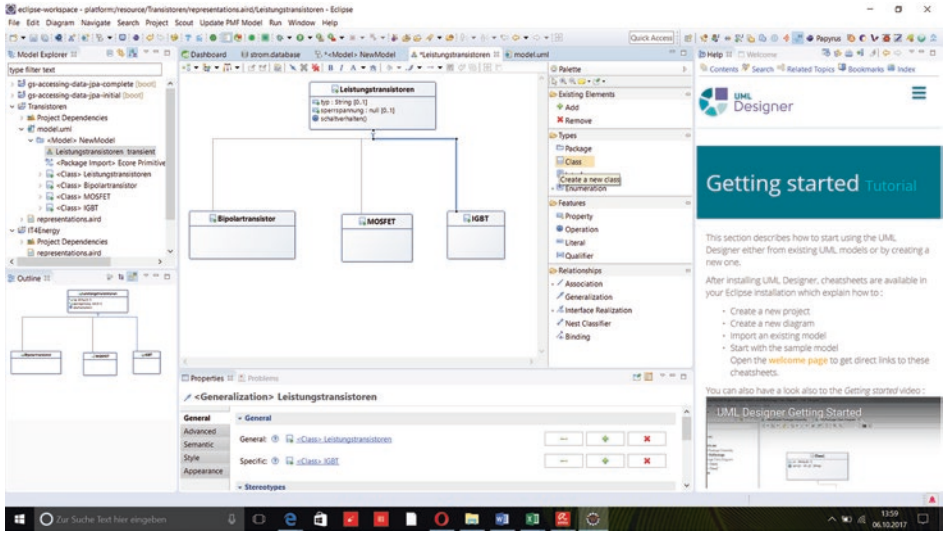


Abb. 2.12 Hierarchischer Vererbungsbaum im Hinblick auf durchgezogene Linien

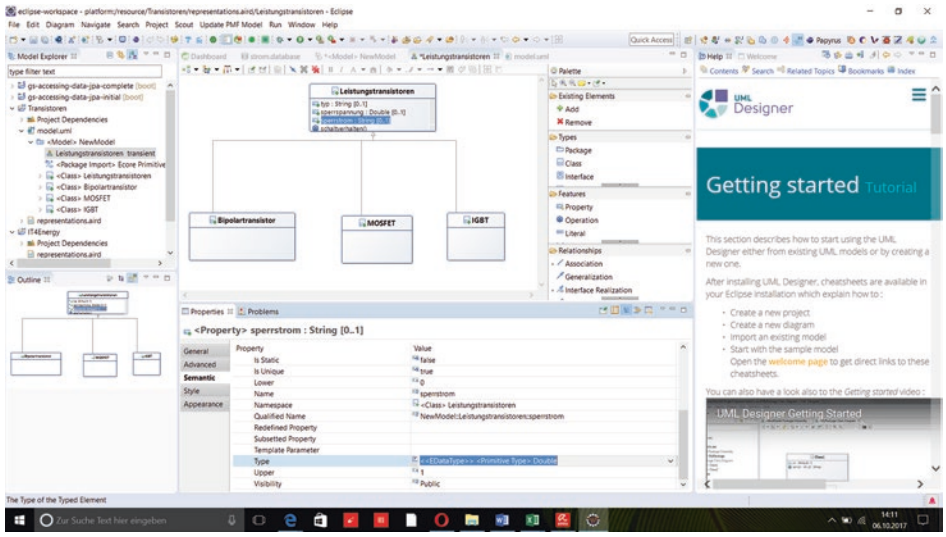


Abb. 2.13 Hierarchischer Vererbungsbaum im Hinblick auf *„Properties“*

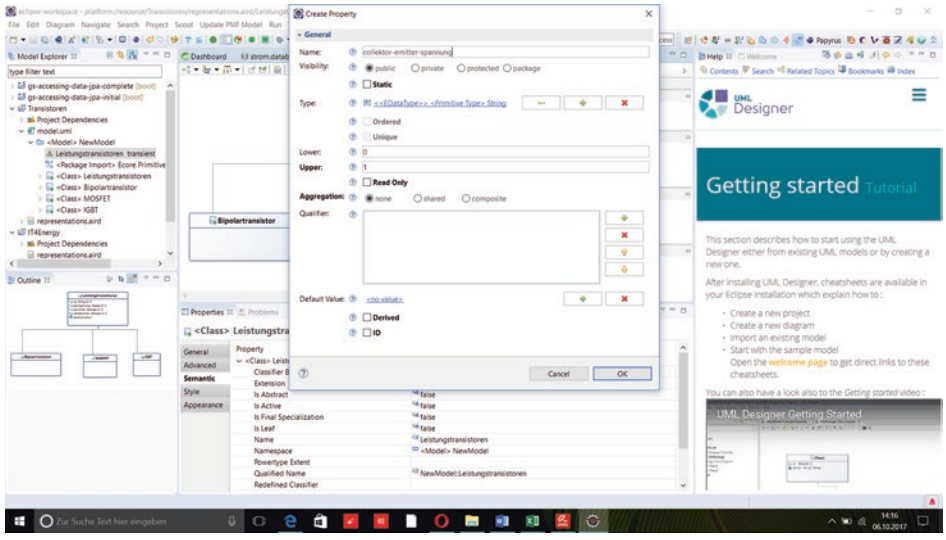


Abb. 2.14 Struktur des Attributs (oder Property) *„collector-emitter-spannung“*

über die in ihm besondere Eigenschaften sowie über die Eigenschaft ihrer Mutterklasse. Tochterklassen erben alle Eigenschaften ihrer Mutterklasse. Abb. 2.11 zeigt zum einen die Baumstruktur zur Vererbung und zum anderen die Erstellung einer Tochterklasse. Im Project Explorer auf den Abb. 2.10, 2.11, 2.12, 2.13, 2.14 und 2.15 ist eine Gliederung der Klassen des Klassendiagramms zu sehen. Auf der Abb. 2.10 unten ist bei Properties die Information über die Mutterklasse und ihre Tochter in Bezug auf Generalisierung. Hierbei wird der hierarchischen Vererbungsbaum mit den Klassen modelliert (Abb. 2.10, 2.11, 2.12 und 2.13). Abb. 2.12 und 2.13 zeigen mithilfe der durchgezogenen Linie die Beziehung zwischen der Mutterklasse *„Leistungstransistor“* und ihren Tochterklassen *Bipolartransistor, MOSFET* und *IGBT.*

2.1.2 Eigenschaften der Klassen

Eine Klasse besteht aus einer Sammlung von Attributen und Methoden: Sie stellen die Eigenschaften der Klassen dar. Attribute (oder Property beim Obeo-UML-Designer) definieren mithilfe von Bezeichnern *„Typ“* und *„Name“* den Zustand einer Klasse.

Auf den Abb. 2.15 und 2.16 sind bei allen fünf Attributen die Datentypen angegeben. Bezüglich der Programmierungssprache Java wurde die in der UML standardmäßig übliche Form *„attribut: Typ“* durch die Java-konforme Fassung *„Typ attribut“* ersetzt.

Für Attribute stehen mehrere Modifikatoren zur Verfügung, die die Attributeigenschaften genauer festlegen. UML definiert Formen „+“ für public, „#“ für protected und „−“ für private, um die Sichtbarkeit des Attributs für unbekannte Klassen zu beschreiben. „+“ ermöglicht einen generellen Zugriff, „#“ für Tochterklassen und „−“ erlaubt Zugriff nur innerhalb der definierenden Klasse [2]. Ein so markiertes Attribut ist frei lesbar, darf aber nur in Tochterklassen und der Mutterklasse selbst modifiziert werden. Diese Sichtbarkeitsangabe wirkt also beim Lesen wie public und bei der Modifikation wie protected. Sie erweist sich bei der Modellierung als hilfreich, um die Zugriffsrechte noch feiner zu beschreiben. Obeo-UML-Designer stellt ein Attribut als Dreieck mit einem Plus-Zeichen dar, wobei es vor dem Klassennamen ein Dreieck mit einem Plus-Zeichen zusehen ist. Dies bedeutet, dass diese Klasse *„public“* deklariert ist (Abb. 2.15).

Mithilfe der Programmiersprache Java stehen Modifikatoren zur Verfügung, wie beispielsweise static und final zur Beschreibung statischer und nichtmodifizierbarer Attribute, wobei sie im Klassendiagramm auch angewendet werden können. Da die Modifikatoren zur Definition von Konstanten dienen, werden Konstanten in Klassendiagrammen absichtlich nicht hinzugefügt.

Mit dem Framework Obeo-UML-Designer werden Methoden, genannt *„Operationen“*, als Zahnrad zur Klassenrepräsentation mit Namen, Sichtbarkeit und Modifikatoren wie z. B. *Static, Abstract* und *Query* dargestellt. Mit dem UML-Designer wird die offizielle UML-Schreibweise *„Operation“* verwendet (Abb. 2.15, 2.16 und 2.17). Während Attribute den Zustand eines Objektes speichern, dienen Methoden dazu,

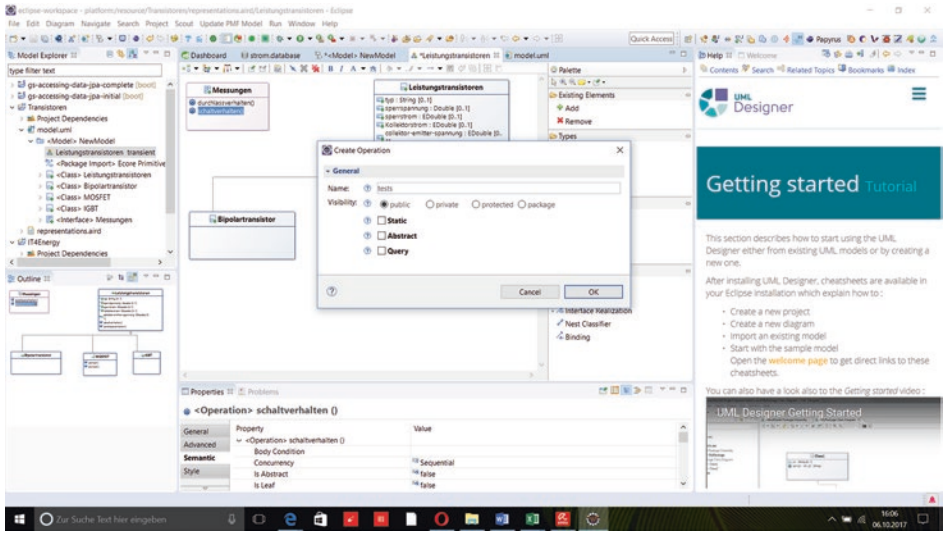


Abb. 2.15 Darstellung der Eigenschaften einer Klasse mit Hinblick auf Property und Operation

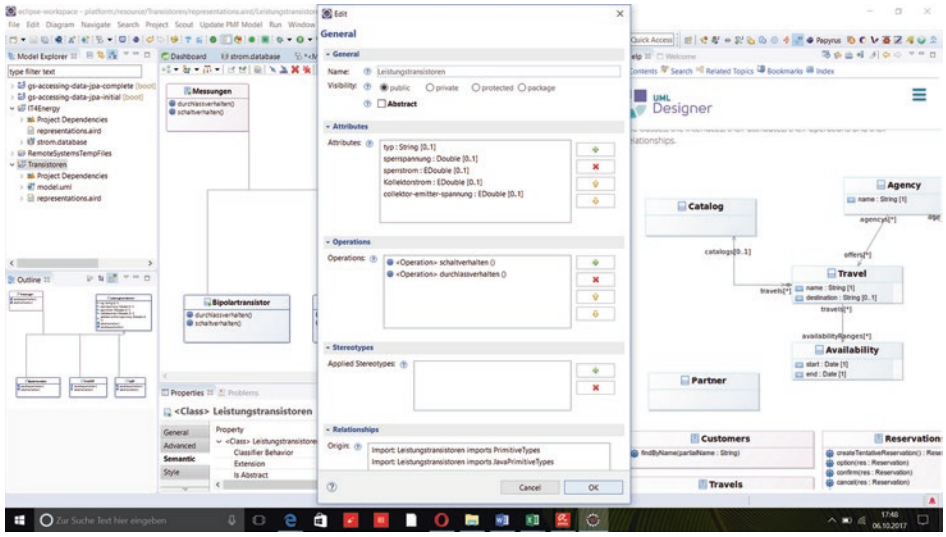


Abb. 2.16 Struktur der Mutterklasse *„Leistungstransistor“* mit Hinblick auf *„Attributes“* und *„Operations“* und *„Relationships“*

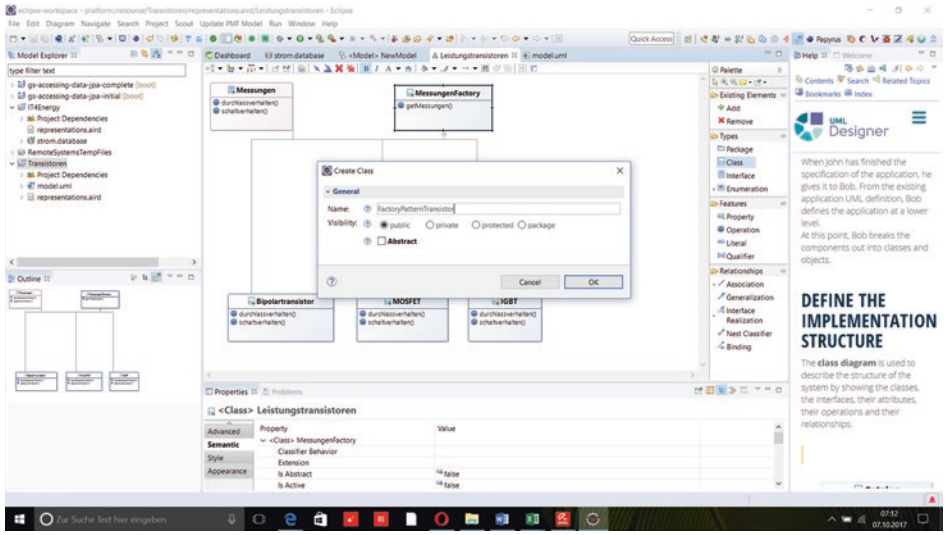


Abb. 2.17 Das Erstellen der neuen Klasse *„FactoryPatternTransistor“* in der Vererbungsbeziehung

bestimmte Aufgaben zu erledigen und Daten zu berechnen. Bei UML-Designer, wie es auf der Abb. 2.15 zu sehen ist, sind Modifikatoren für Methoden *Static* für den Zugang zur Methode ohne instanziiertes Objekt; *Query* für die Methode beim Suchen mit Datenbank und *Abstract* für die nicht implementierte Methode einer Klasse.

2.1.3 Modellierung des Klassendiagramm mithilfe der Operationen

Wie schon in dem Abschn. 2.1.2 erwähnt wurde, ist die Struktur einer Operation in dem erzeugten Klassendiagramm beim Framework Obeo-UML-Designer mit einem Zahnrad-Symbol dargestellt.

Da Klassen durch Rechtecke mit Klassennamen und Merkmalen wie z. B. Attribute und Operationen dargestellt sind, enthält die Struktur des Klassendiagramms beim UML-Designer keine horizontale Linie zur Trennung der Attribute von der Operation. Wie schon erwähnt wurde, steht vor dem Klassennamen ein Dreieck mit einem Plus-Zeichen zur Sichtbarkeit der Klasse. Abb. 2.18, 2.19 und 2.20 zeigen das Design Pattern *„Factory-Methode“* des Klassendiagramms mit Operationen. Wie auf den Abb. 2.18, 2.19 und 2.20 zu sehen ist, gibt es ein Interface als Superklasse, genannt *Messungen;* drei Subklassen, genannt *Bipolartransistor, MOSFET,* eine Factory-Klasse, genannt *MessungenFactory und IGBT,* eine Hauptprogramm-Klasse, genannt *FactoryPatternTransistor.* Ein Interface beim Framework UML-Designer wird wie eine Klasse durch ein Rechteck dargestellt und mit dem Buchstaben *„I“*, einem Dreieck gekennzeichnet. Es gibt ein Plus-Zeichen unter

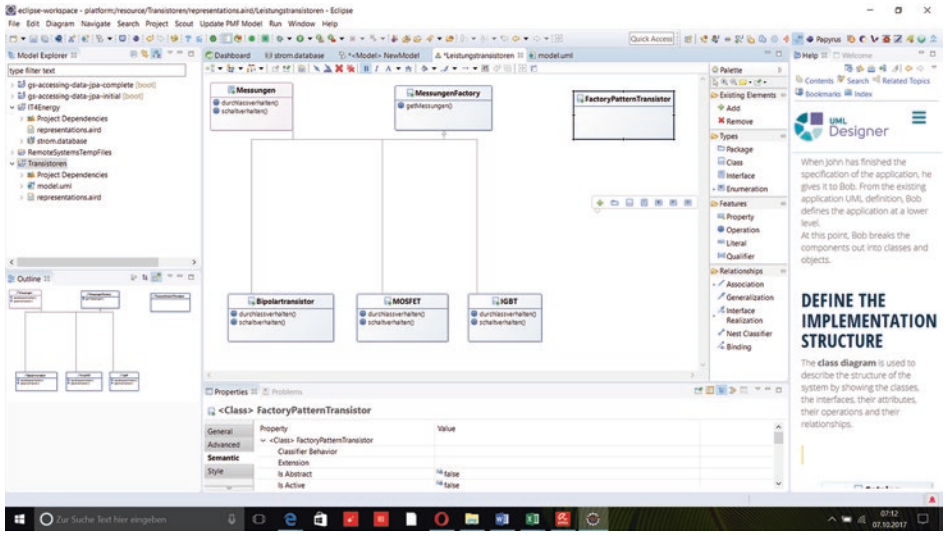


Abb. 2.18 Überblick über Methoden des Design Patterns *FactoryMethod*

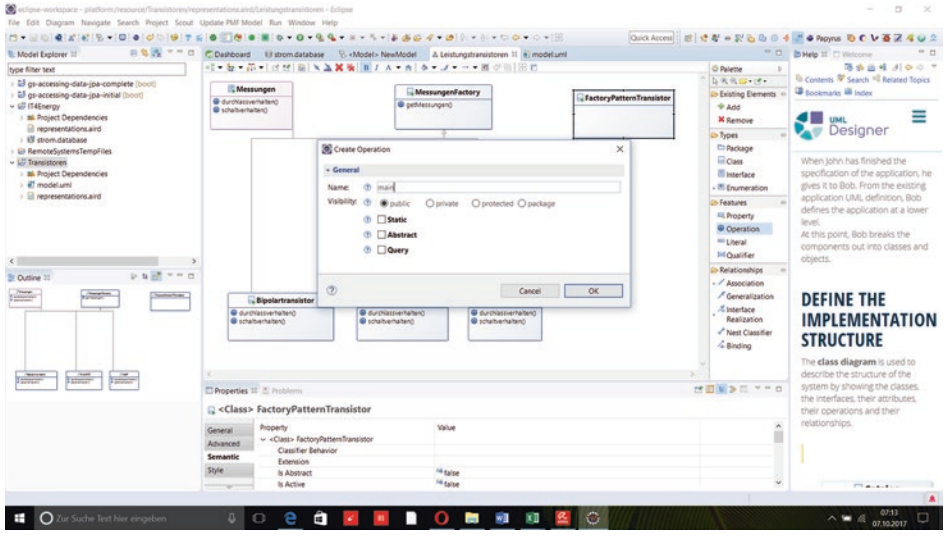


Abb. 2.19 Das Erstellen der Hauptklasse *FactoryPatternTransistor* im Hinblick auf Sichtbarkeit

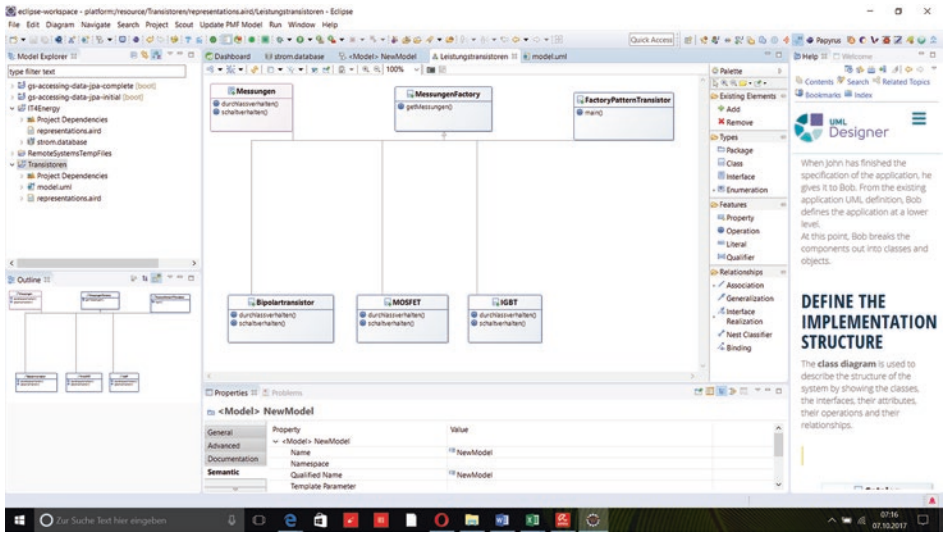


Abb. 2.20 Vererbungshierarchie der Klassen im Design *Pattern FactoryMethod*

dem Buchstaben *„I“*, wobei wie bei abstrakten Klassen nicht direkt Objekte von den Interfaces instanziiert werden können. Deswegen müssen die angegebenen Methodensignaturen in Klassen dargestellt werden, die ein Interface implementieren. Abb. 2.18 zeigt die zwei Operationen *durchlassverhalten* und *schaltverhalten* von der Superklasse und den Subklassen. Allerdings implementieren die Subklassen die zwei Operationen mithilfe der Factory-Klasse, wobei diese eine Methode, genannt *getMessungen,* enthält. Es ist auf den Abb. 2.18, 2.19 und 2.20 zu sehen wie die Instanziierung der Klasse *FactoryMessungen* mithilfe der Methode *getMessungen* das Implementieren der Operationen *durchlassverhalten* und *schaltverhalten* der Superklasse ermöglicht. Auf der Abb. 2.19 ist die Erstellung der Klasse *FactoryPatternTransistor* mit der Operation *main,* welche das Hauptprogramm enthält. Alle Operationen sind in einem hierarchischen Vererbungsbaum zu sehen. Es ist zu bemerken, dass beim Framework UML-Designer vor jeder Operation ein Zahnrad-Zeichen steht.

2.1.4 Praxis-Beispiel: Anwendung der Klassendiagramme in der Modellierung des Durchlassverhaltens des Transistors

Obeo-UML-Designer ist ein interessantes Framework zur Modellierung des Klassendiagramms. Hierbei nutzt das Framework die Stärke der DSL Sirius 5. Im realen Fall ist Eclipse Obeo-UML-Designer zur Modellierung von Systemen in Bezug auf Leistungselektronik effizient. Die folgende Modellierung fokussiert auf das Durchlassverhalten des Transistors, genannt *IGBT*. Mit dem Editor des Frameworks Obeo-UML-Designer sind

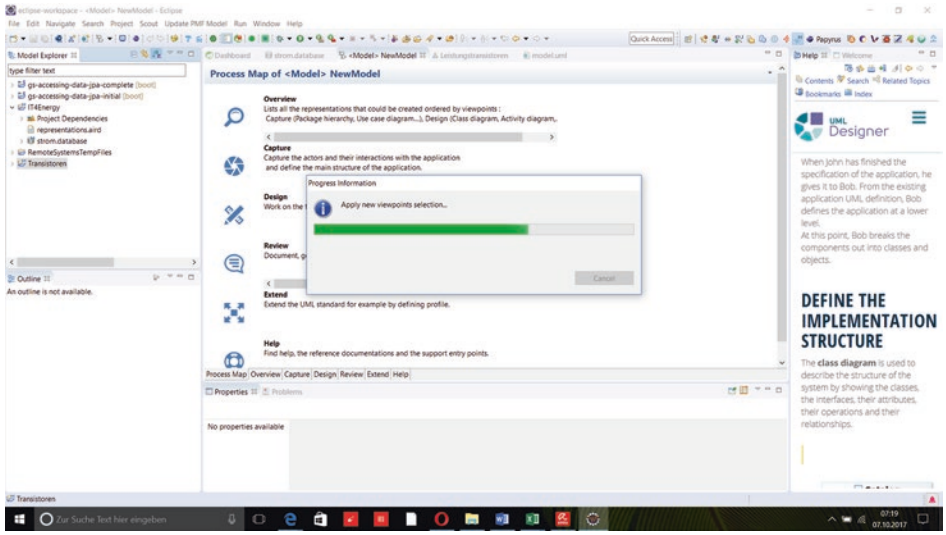


Abb. 2.21 Erstellen einer Mappe des Models, genannt *„Process Map of Model“*

verschiedene Klassen mithilfe des Design Patterns *„Model View Controller“* (MVC) modelliert. Dieses Praxis-Beispiel ermöglicht die Realisierung der Grenzwerte des Transistors wie z. B. Gate-Emitter-Spannung, Kollektor-Emitter-Spannung und Kollektorstrom für das Datenausblatt. Ziel dieser Modellierung mit Obeo-UML-Designer ist es, die Spezifizierung der Komponenten des IGBT in Bezug auf Grenz- und Kennwerte zu ermöglichen. Abb. 2.21, 2.22, 2.23, 2.24, 2.25, 2.26, 2.27, 2.28, 2.29, 2.30, 2.31, 2.32, 2.33, 2.34, 2.35, 2.36, 2.37, 2.38, 2.39 und 2.40 zeigen die Modellierung des Durchlassverhaltens des IGBT mithilfe von Kollektor-Emitter-Spannung, Gate-Emitter-Spannung und Kollektorstrom, wobei die Modellierung die Analyse der Durchlassverluste des IGBT ermöglicht. Ein IGBT ist ein Halbleiterbauelement, welches zunehmend in der Leistungselektronik verwendet wird, da es die Vorteile des Bipolartransistors (gutes Durchlassverhalten, hohe Sperrspannung) und die Vorteile eines Feldeffekttransistors (nahezu leistungslose Ansteuerung) vereinigt. Vorteilhaft ist auch eine gewisse Robustheit gegenüber Kurzschlüssen, da der IGBT den Laststrom begrenzt [4, 5].

Das Klassendiagramm beinhaltet die Struktur des Design Patterns MVC, wobei es ein Architektur- oder Entwurfsmuster zur Trennung von Software in die Komponenten Datenmodell, Präsentation und Programmsteuerung darstellt. Dieses Praxis-Beispiel nutzt das Design Pattern zum Modellieren der Wiederverwendung der Klassenelemente, wobei die Grenz- und Kennwerte des IGBT Ziele der Modellierung mit Klassendiagrammen auf Basis von MVC sind.

Wir beginnen dieses Beispiel mit der Erstellung eines neuen Modells mithilfe des Frameworks Obeo-Designer-UML, wie es auf der Abb. 2.21 zu sehen ist, wobei dies das Erstellen einer Mappe des Models, genannt *„Process Map of Model“*, mit verschieden

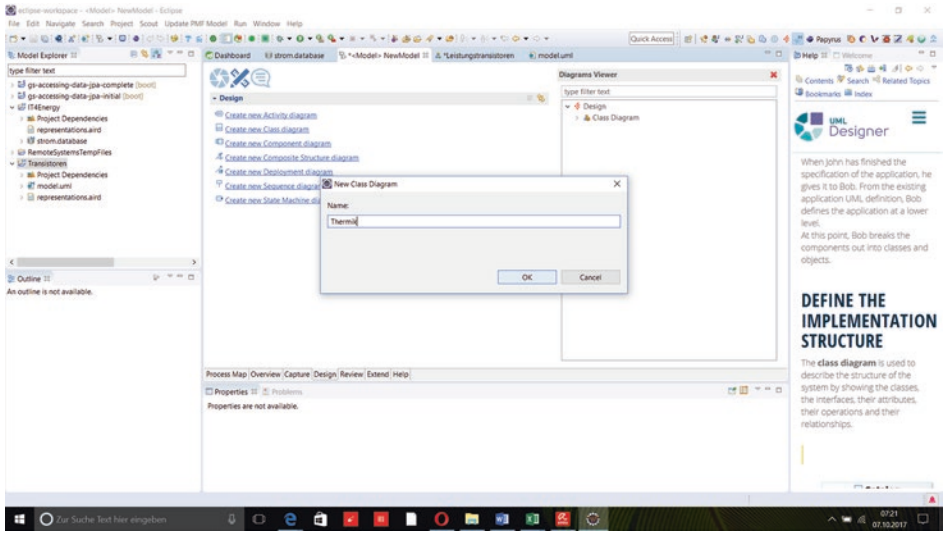


Abb. 2.22 Überblick über den Inhalt des Ordners *„Design“*

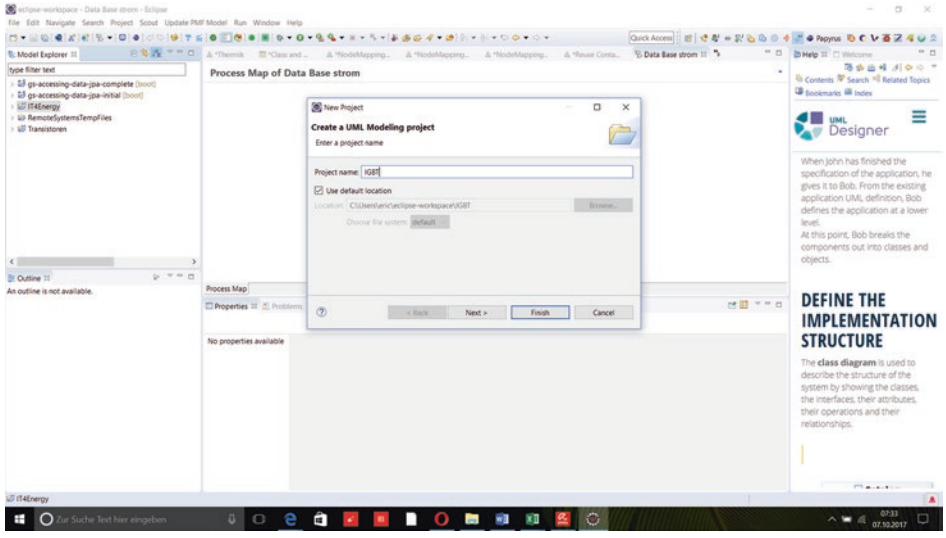


Abb. 2.23 Speicherort des UML-Projektes

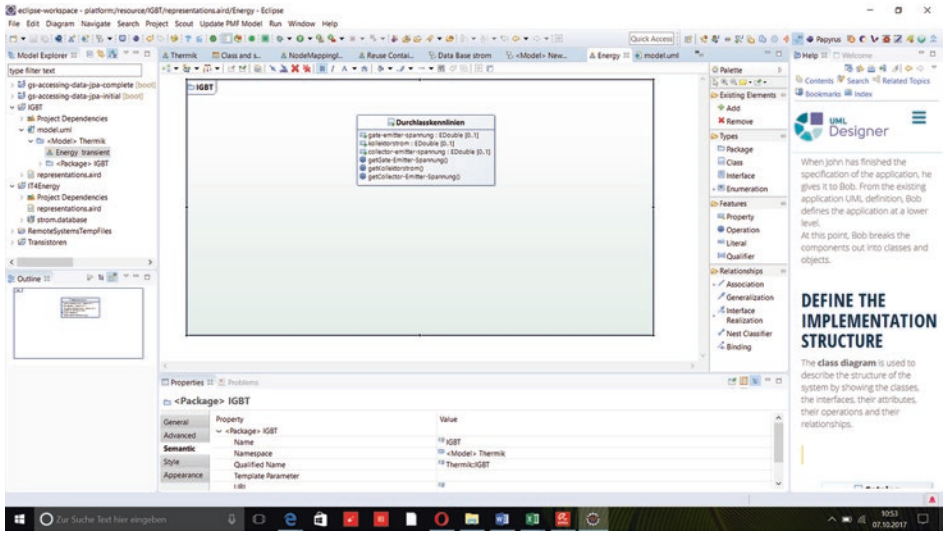


Abb. 2.24 Überblick über das Erstellen der Operation *DurchlasskennlinienDaten*

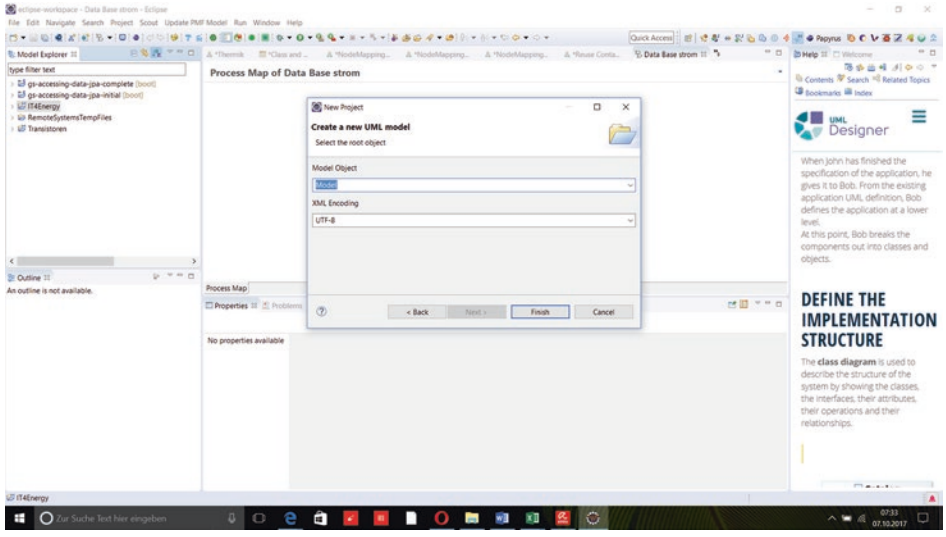


Abb. 2.25 Überblick über XML-Enconding und Model Object

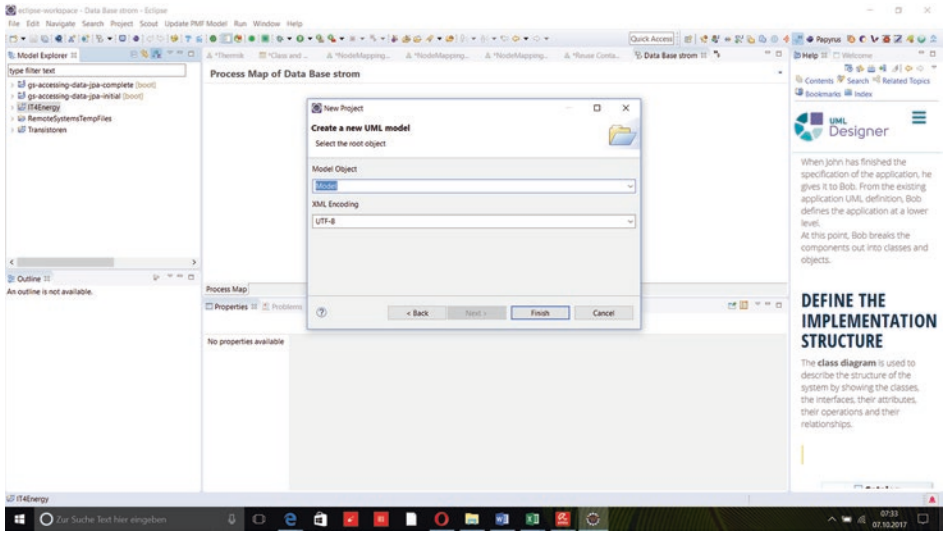


Abb. 2.26 Project Explorer mit dem UML-Projekt „*IGBT*“

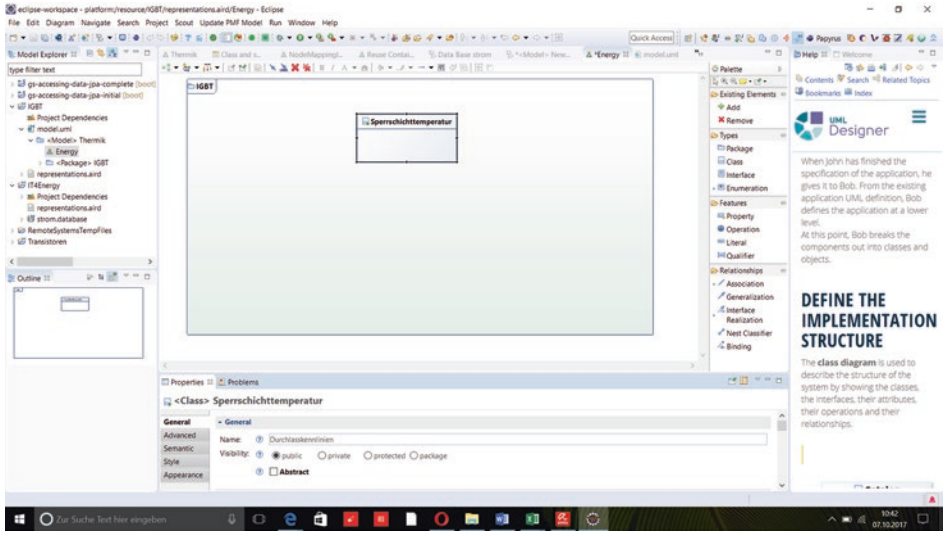


Abb. 2.27 Erstellen eines Property mit dem Framework Obeo-UML-Designer

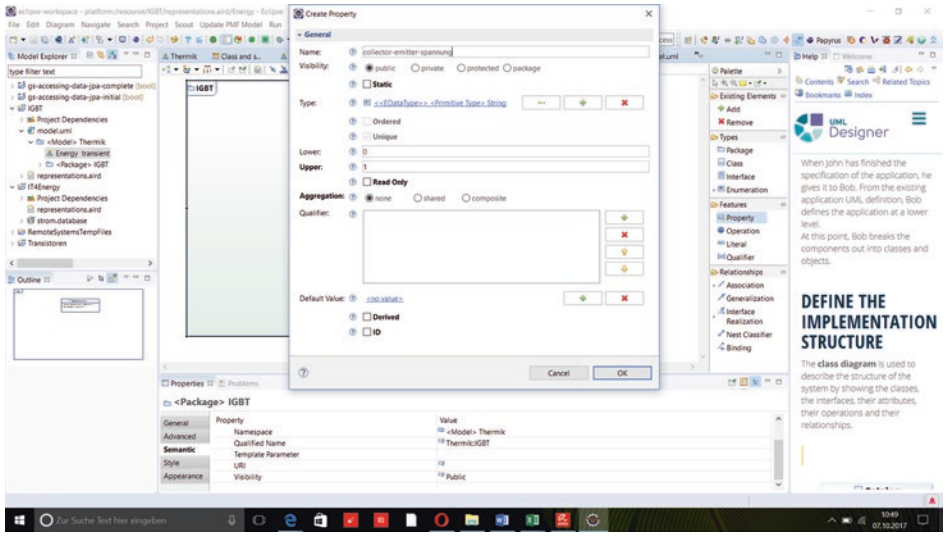


Abb. 2.28 Überblick über die Klasse Durchlasskennlinien

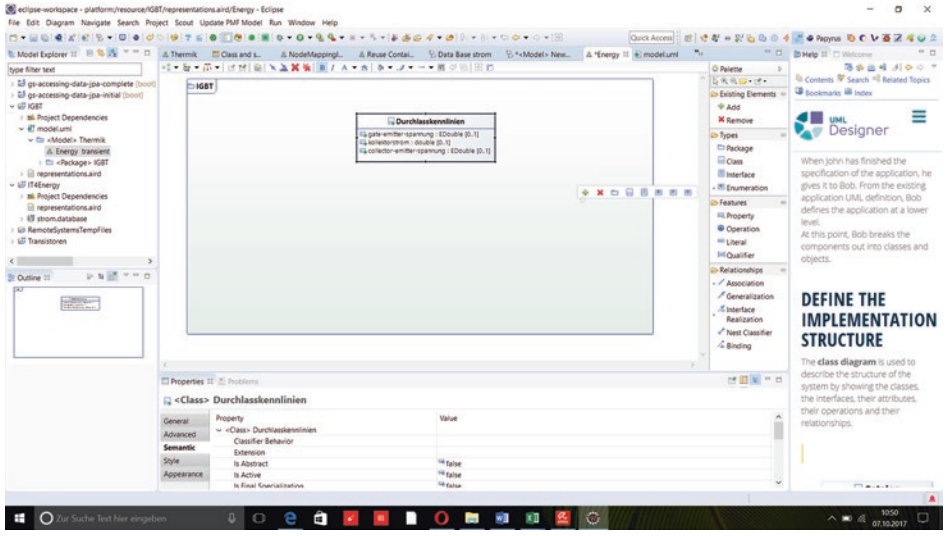


Abb. 2.29 Überblick über das Erstellen der Operation (oder Methode), genannt *„getKollektorstrom“*

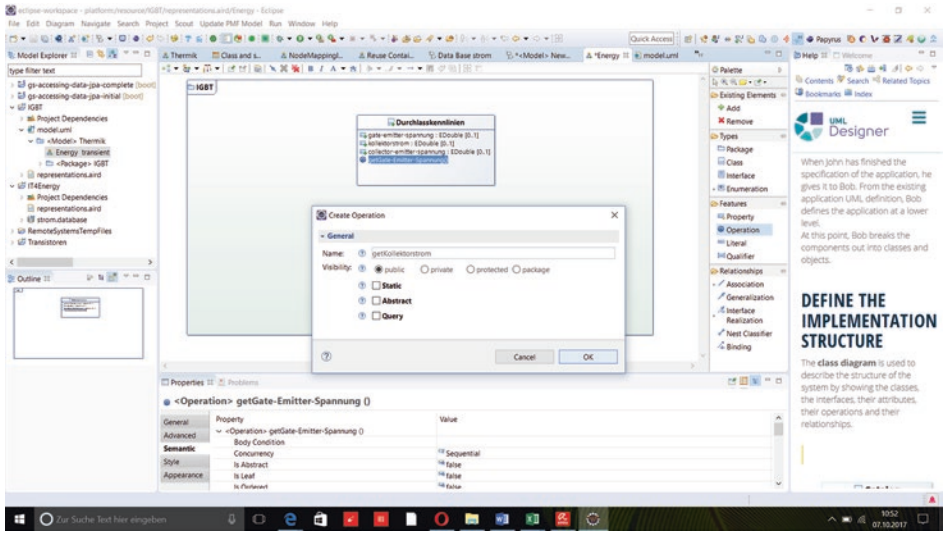


Abb. 2.30 Klasse *Durchlasskennlinien* mit Property und Operationen

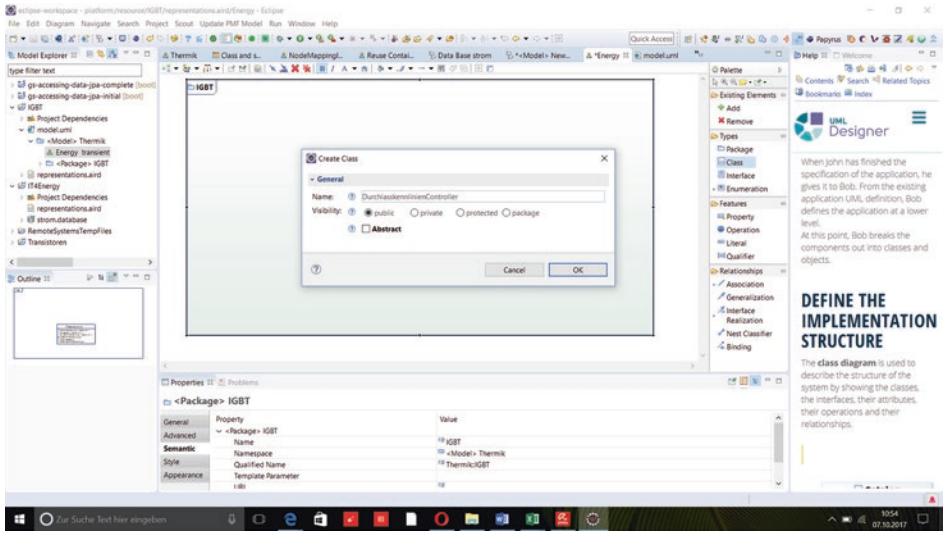


Abb. 2.31 Das Editieren der Klasse *DurchlasskennlinienController* mit beiden Attributen *Durchlasskennlinien* und *DurchlasskennlinienView*

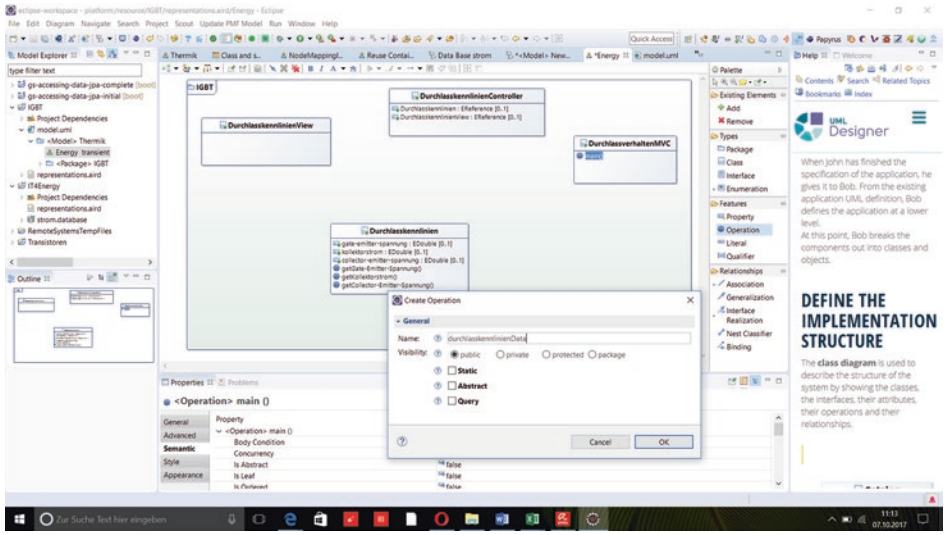


Abb. 2.32 Hauptklasse mit einer neuen Operation *DurchlasskennlinienData*

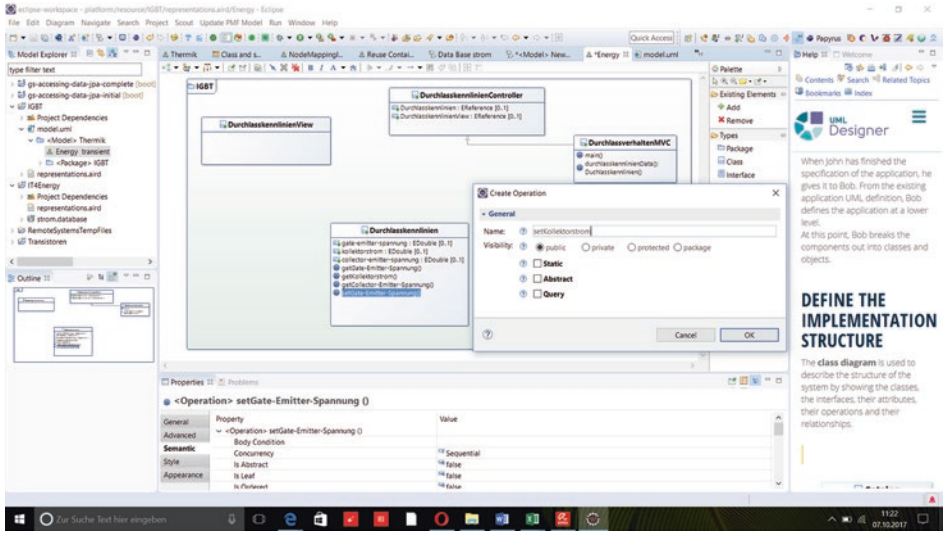


Abb. 2.33 Erstellen neuer „set-Operationen“ in der Klasse *Durchlasskennlinien*

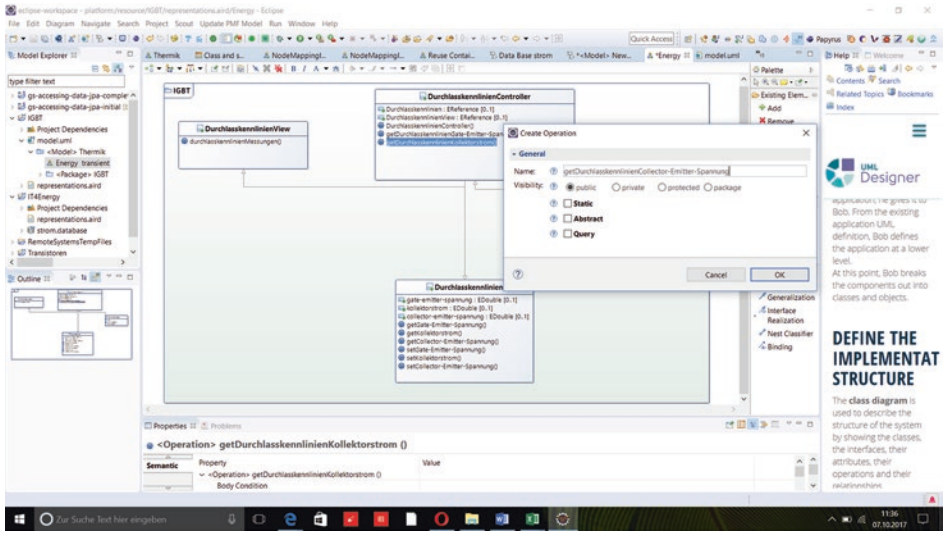


Abb. 2.34 Erstellen einer neuen *„get- Operationen“* in der Klassen *DurchlasskennlinienControl ler*

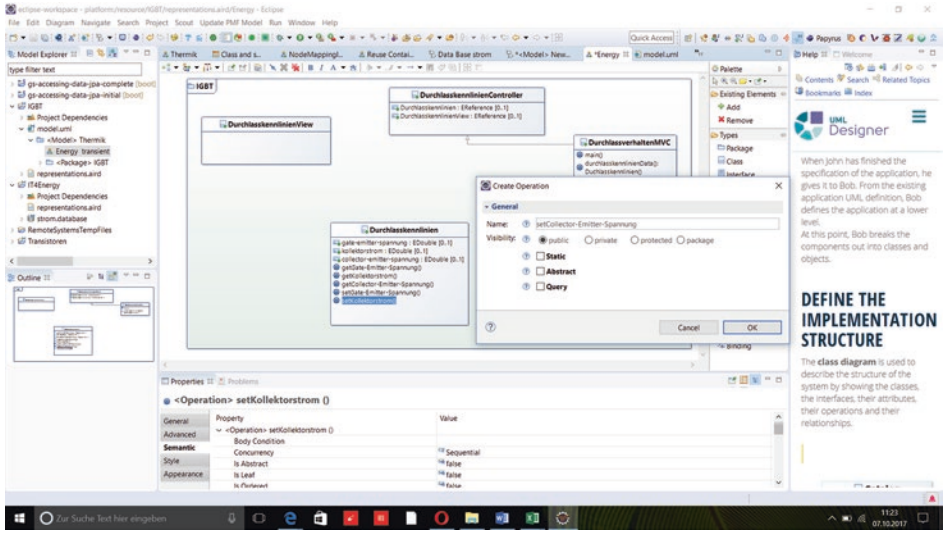


Abb. 2.35 Erstellen einer neuen *„set-Operationen“* in der Klasse *Durchlasskennlinien*

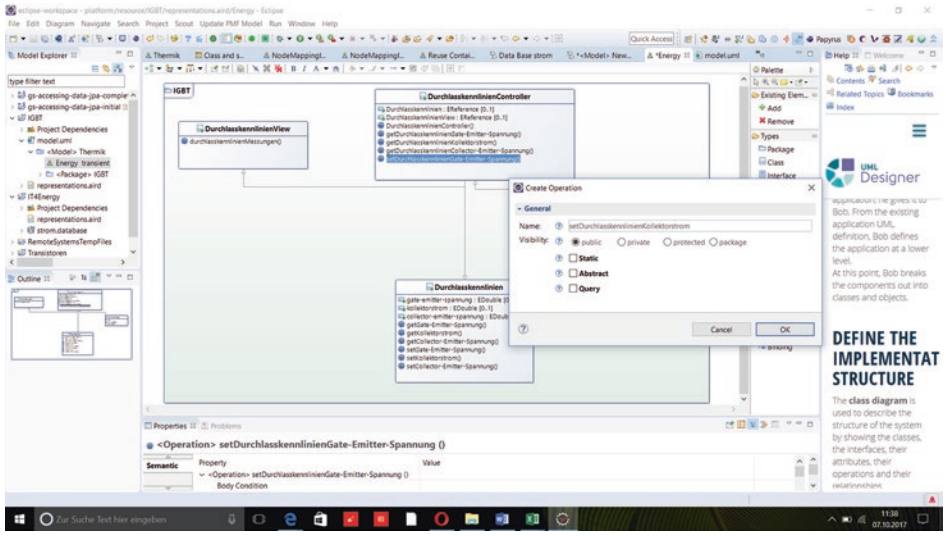


Abb. 2.36 Erstellen der Operation *setDurchlasskennlinienKollectorstrom* in der Klasse *DurchlasskennlinienController*

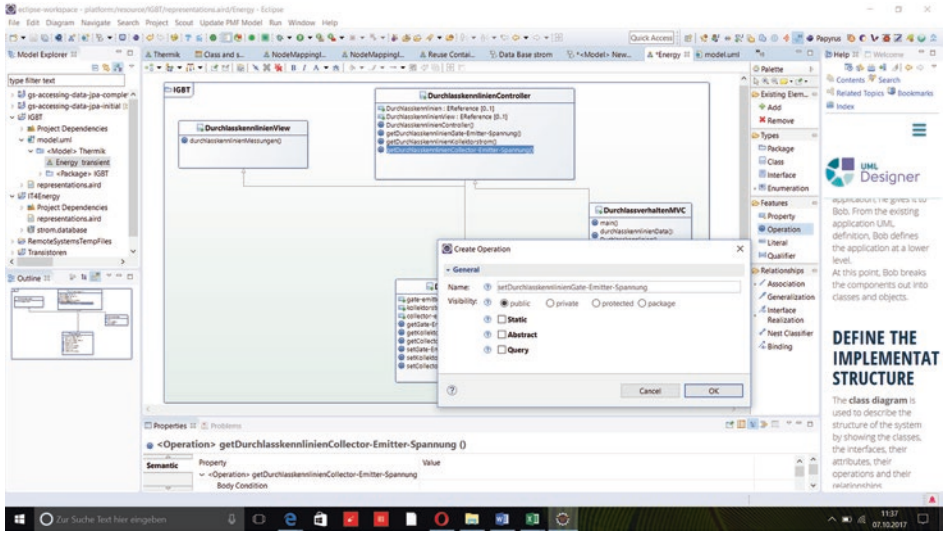


Abb. 2.37 Erstellen der Operationen *setDurchlasskennlinienGate-Emitter-Spannung*

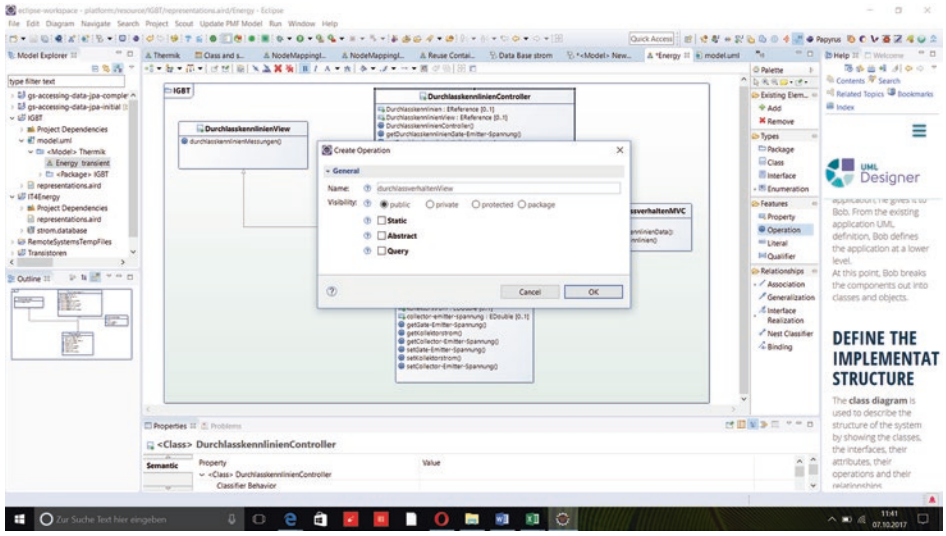


Abb. 2.38 Struktur der Operation *„durchlassverhaltenView“* in der Klasse *Durchlasskennlinien-Controller* in Bezug auf die Eigenschaften (Properties)

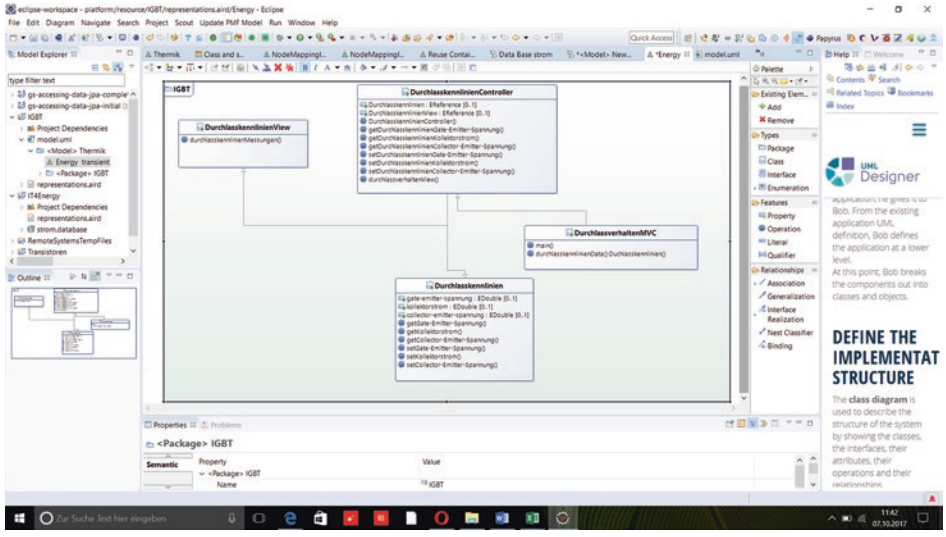


Abb. 2.39 Überblick über Design Pattern MVC mithilfe des hierarchischen Beziehungsbaumes

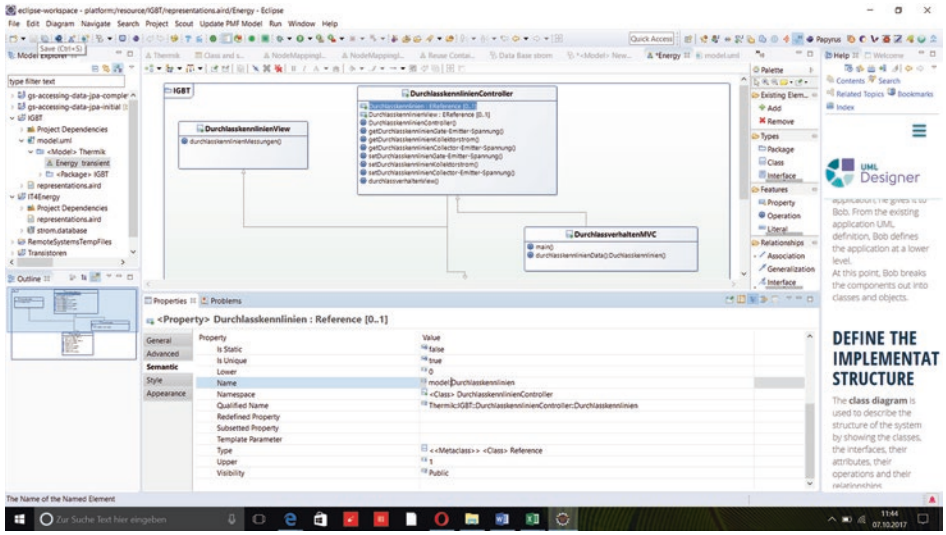


Abb. 2.40 Referenz der Model-Schicht Durchlasskennlinien in der Controller-Schicht mithilfe des Property (Attributs)

Ordnern darstellt, wie z. B. Design, dessen Inhalt UML-Diagramme u. a. *„Activity, Class, Component, Composite Structure, Depolyment, Sequence diagram“*, wie auf der Abb. 2.22 zu sehen ist. Die Abb. 2.22 zeigt zum einen den Inhalt des Ordners *„Design“* und zum anderen das Erstellen eines neuen Klassendiagramms, genannt *Thermik*. Anschließend wird ein neues UML-Modellierungsprojekt, genannt *IGBT,* wobei dieser Schritt das Erstellen eines neuen UML-Models, genannt *„model“*, ermöglicht. Abb. 2.23 zeigt den Speicherort des UML-Projektes. Abb. 2.24 gibt einen Überblick zum einen über das XML-Encoding *„UTF-8“* und zum anderen über den Namen des *„Model Objects“ (model)*. Abb. 2.25 zeigt im Project Explorer die Struktur des UML-Projektes, genannt *IGBT,* und zum anderen das Erstellen der Klasse, genannt *Sperrschichttemperatur* (ein Beispiel), die im *Package,* genannt IGB, erstellt wurde. Vor dem Klassennamen ist ein Rechteck mit einem Plus-Symbol zu sehen: Dies bedeutet, dass diese Klasse *„public“* definiert wurde. Unten im Ordner *„Properties“* sind die Informationen über die erstellte Klasse, genannt *Durchlasskennlinien* wie z. B. Sichtbarkeit. Auf der Abb. 2.26 ist das Erstellen eines Property (beim Obeo-UML-Designer) von dem Bezeichner *„Property“*, genannt *gate-emitter-spannung,* in der Klasse Durchlasskennlinien zu sehen. Die Struktur eines Property gibt Überblicke z. B. über seinen Namen, seine Sichtbarkeit und seinen Modifikator. Die Klasse Durchlasskennlinien besteht, wie auf der Abb. 2.27 zu sehen ist, schon aus drei Properties, deren Datentype festgelegt wurden. Es ist anzumerken, dass vor jedem Property ein Dreieck mit einem Plus-Symbol steht. Dies bedeutet, dass die Sichtbarkeit auf *public* gesetzt wurde. Abb. 2.28 gibt einen Überblick über das Erstellen der Operation (oder Methode), genannt *„getKollektorstrom“,* wobei vor der Operation *„Namen“* ein Zahnrad-Symbol steht. Die erweitete Klasse *Durchlasskennlinien* mit

Property und *Operationen* wie *getGate-Emitter-Spannung(), getKollektorstrom()* und *getCollector-Emitter-Spannung()* zeigt die Abb. 2.29, welche unten im Ordner *Properties* mit Informationen über Property und Value der Package *„IGBT“* darstellt. Das Design Pattern MVC ist schon, wie es auf der Abb. 2.30 zu sehen ist, realisiert. Die neuen Klassen sind *DurchlasskennlinienController, DurchlasskennlinienView* und *DurchlassverhaltenMVC.* Die letzte Klasse enthält das Hauptprogramm der Klassendiagramme in Bezug auf das Konzept MVC. Es ist auf der Abb. 2.30 einen Überblick über das Erstellen der Operation *DurchlasskennlinienDaten* zu sehen ist. Das Editieren der Klasse *DurchlasskennlinienController* mit beiden Attributen *Durchlasskennlinien* und *DurchlasskennlinienView* zeigt die Abb. 2.31. Auf der Abb. 2.32 ist die Hauptklasse mit einer neuen Operation *DurchlasskennlinienData* erweitert. Außerdem ist auf den Abb. 2.32, 2.33, 2.34, 2.35, 2.36 und 2.37 das Erstellen neuer *„set/get- Operationen“* in den Klassen *Durchlasskennlinien* und *DurchlasskennlinienController* zu sehen. Klassenbeziehungen mithilfe des Design Pattern MVC stellen einen hierarchischen Baum, wie es auf den Abb. 2.38, 2.39 und 2.40 zu sehen ist, in Bezug auf Abstraktion, Kommunikation und Verwaltung der Komponente dar. Abb. 2.38 zeigt die allgemeine Struktur in Bezug auf die Eigenschaften*(Properties)* der erstellten Operation genannt *durchlassverhaltenView* in der Klasse *DurchlasskennlinienController.* Mit der Abb. 2.39 ist das Design Pattern MVC mithilfe des Beziehungsbaumes zu analysieren und die Modellierungsstruktur zu verstehen. Wobei die Modellierung des Design Pattern MVC die Abstraktion der Komponente darstellt. Hierbei ist mit der Abb. 2.39 die Abstraktion der Modellierung mithilfe der Struktur der *Controller-Schicht,* genannt *DurchlasskennlinienController* bezüglich der Referenz der *Model-* und *View-Schicht* zu sehen, wobei die Merkmale der Model-Schicht über die Views zur Verfügung gestellt sind. Die Referenz der Model-Schicht *Durchlasskennlinien* in der Controller-Schicht mithilfe der Property (Attributs) in Bezug auf die Semantik der Modellierung im Ordner *Properties* zeigt Abb. 2.40.

2.2 Kompositionsstrukturdiagramm von Obeo-Designer-UML

Das Kompositionsstrukturdiagramm dient der Dekomposition und Modellierung von Classifiern, die in Kompositionsbeziehungen zueinander stehen. Es zeigt, wie die Teile eines Classifiers den Classifier selbst bilden. Da es mit dem Kompositionsstrukturdiagramm möglich ist, die Teile (Architekturelemente) eines Systems und ihre Beziehungen darzustellen, wird es auch Architekturdiagramm genannt. Das Kompositionsstrukturdiagramm eignet sich ebenfalls zur Modellierung von Entwurfsmustern [6].

Das praktische Beispiel fokussiert auf die Kompositionsstruktur von Wechselrichter und Netz in Bezug auf Stromproduktion. Die Modellierung der Zusammensetzung von Wechselrichter und Netz ermöglicht das Analysieren der Funktionsweise der PV-Wechselrichter, wobei das Gerät verschiedene Komponente wie z. B. Leistungstransistoren, Dioden, Resonanzelemente enthält. Praktisch alle modernen Energieversorgungsnetze

arbeiten mit Wechselstrom, und das ist der Trend. Man kann ihn mithilfe von Synchrongeneratoren einfach erzeugen. Man kann ihn über große Strecken übertragen, denn Wechselspannung lässt sich sehr einfach auf andere Niveaus transformieren. Außerdem können elektrische Verbraucher einfach angeschlossen werden. Doch auch Gleichstrom spielt nach wie vor eine wichtige Rolle: Nicht nur Straßenbahnen und Akkuladegeräte, so gut wie alle elektronischen Schaltungen benötigen ihn zum Betrieb. Batterien, Brennstoff- und Solarzellen liefern ausschließlich Gleichstrom. Normalerweise sind Gleich- und Wechselrichter als Bindeglied zwischen den beiden Stromsystemen von großer praktischer Bedeutung. Gleichrichter finden sich in beinahe jedem Steckernetzteil, doch auch Wechselrichter sind weit verbreitet: Ob zum Betrieb von handelsüblichen 230-V-Geräten im Auto, in unterbrechungsfreien Stromversorgungen oder für die Nutzung erneuerbarer Energien – Anwendungen für Wechselrichter gibt es jede Menge. Die Einspeisung von umweltverträglich erzeugtem Strom in das Versorgungsnetz hat sich dabei zu einem Schwerpunkt der Wechselrichtertechnik entwickelt. Bei den meisten Windkraftwerken [7] und ausnahmslos allen Photovoltaik-Anlagen ist der Wechselrichter die Schnittstelle zum Versorgungsnetz – bei PV-Anlagen das zentrale Element. Er ist nicht nur verantwortlich für die möglichst vollständige Umwandlung des Gleichstroms in Wechselstrom, sondern sorgt erst für den Betrieb der Solarzelle im optimalen Betriebspunkt, überwacht das Netz und die Leistung der PV-Anlage. Zur Einspeisung von Strom aus Photovoltaik-Generatoren in das Wechselspannungsnetz der öffentlichen Versorgung ist dem gegenüber grundsätzlich ein Wechselrichter erforderlich [8].

Die Solarzellen werden zu etwa 1,3 × 1,9 m großen Modulen zusammengefasst und liefern bei entsprechender Bestrahlung eine Gleichspannung, die in einem angeschlossenen Stromkreis einen Gleichstrom treibt und so genutzt werden kann. Der Wechselrichter wandelt diesen Gleichstrom in netzüblichen Wechselstrom (z. B. 50 Hz/230 V für das deutsche Niederspannungsnetz) und speist ihn gegen Bezahlung in das Versorgungsnetz ein. Bei den Wechselrichtern stehen je nach Anlagengröße mehrere Konzepte zur Wahl: Vom Modulwechselrichter (Leistung entsprechend dem einzelnen Solarmodul) über den klassischen Stringwechselrichter (Leistung 2,5–11 kW) bis hin zu Zentralwechselrichtern (Leistung 100–1200 kW) mit denen fußballfeldgroße PV-Kraftwerke betrieben werden. Sie speisen die Energie zum Teil direkt in das Mittelspannungsnetz mit einer Spannung von 20.000 V ein.

Abb. 2.41, 2.42, 2.43, 2.44, 2.45, 2.46, 2.47, 2.48, 2.49, 2.50 und 2.51 geben Überblicke über die Modellierung der Zusammensetzung von Wechselrichter-Komponenten und Gleichspannung aus dem Solarmodul. In Bezug auf die Schaltung von Resonanz-Wechselrichtern ist auf den Abb. 2.45, 2.46, 2.47, 2.48, 2.49, 2.50 und 2.51 die Funktionsweise der IGBT zu analysieren, wobei diese Transistoren synchron arbeiten, d. h., IGBT 1 und 4 werden gleichzeitig ein- und ausgeschaltet, ebenso mit IGBT 2 und 3. Die Modellierung der Komponente des Resonanz-Wechselrichters fokussiert zum einen auf die Beziehung zwischen Gleichspannung (Teil der Solarmodule) und Wechselspannung oder Wechselstrom (Teil des Wechselrichters) und zum anderen auf die Beziehung zwischen der Last bestehend aus der Spule und dem Wider-

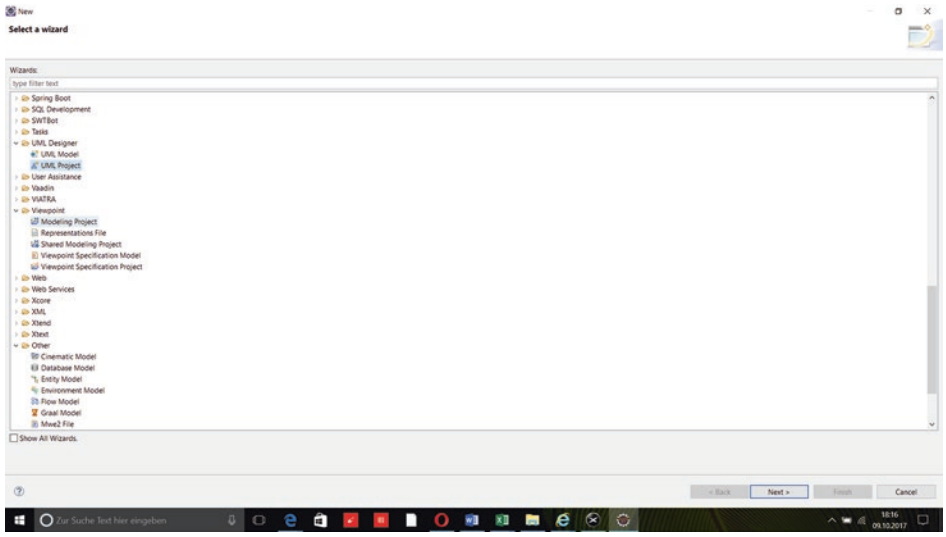


Abb. 2.41 Das Erstellen des Projekts genannt *Solar* mit Hinblick auf den Ordner *„UML-Designer“*

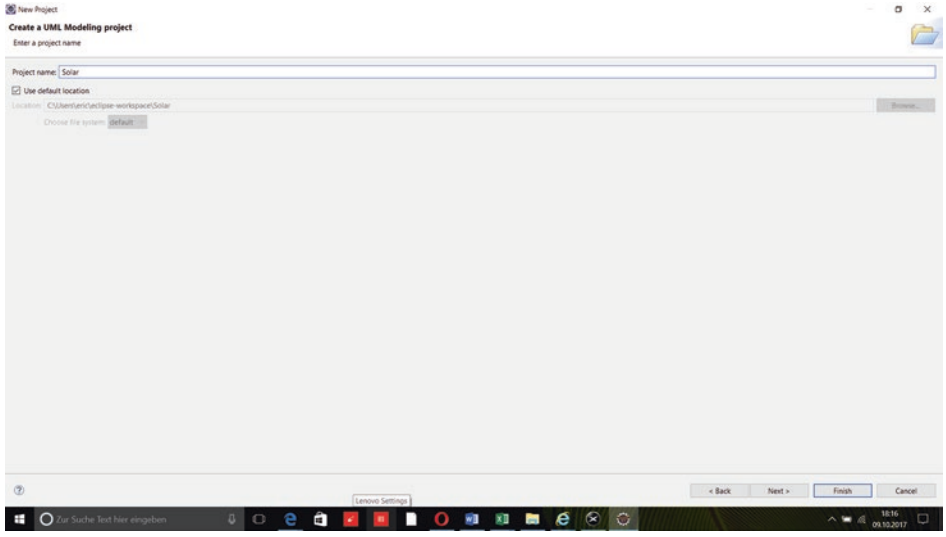


Abb. 2.42 Überblick über das Fenster *„New Project“* zum Erstellen des Projekts *Solar*

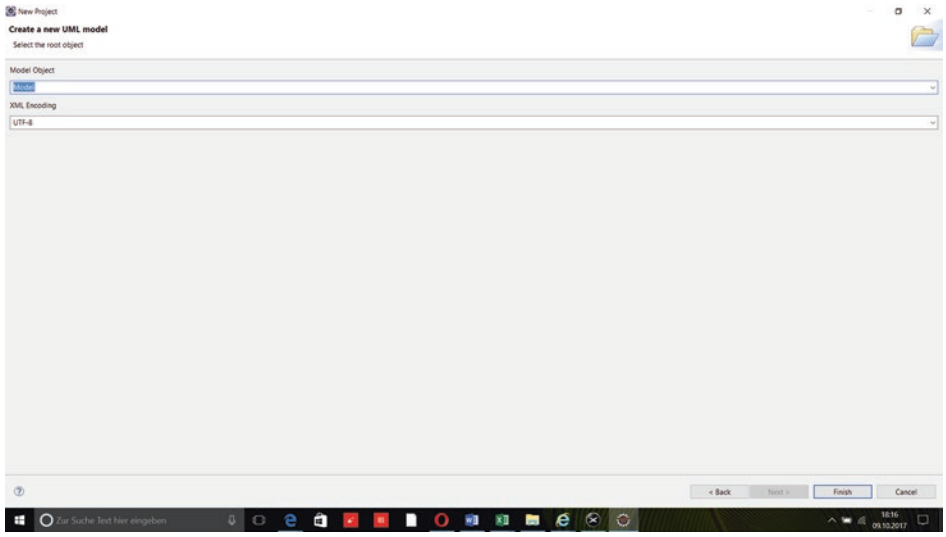


Abb. 2.43 Erstellen des neuen UML-Models in Bezug auf *„Model Object“* und *„XML-Encoding“*

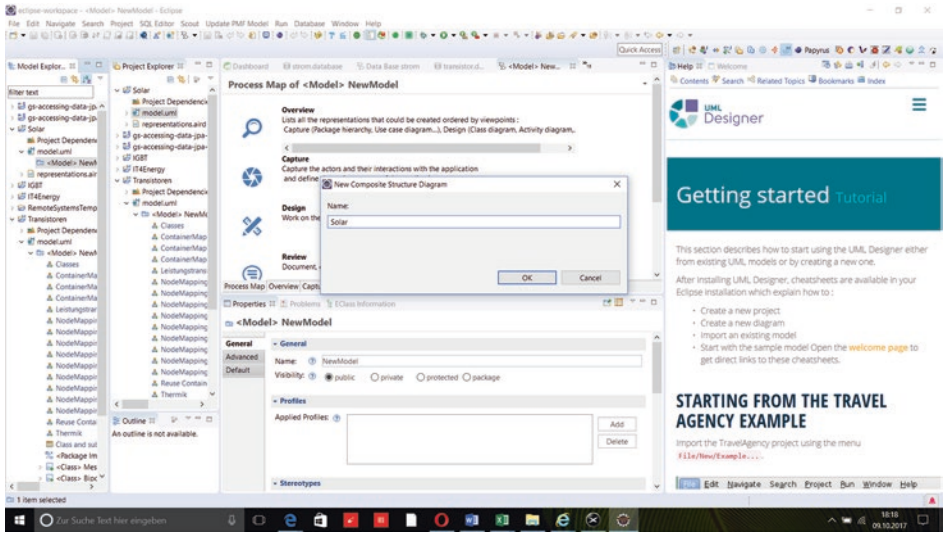


Abb. 2.44 Überblick über das neue erstellte Kompositionstrukturdiagramm *Solar*

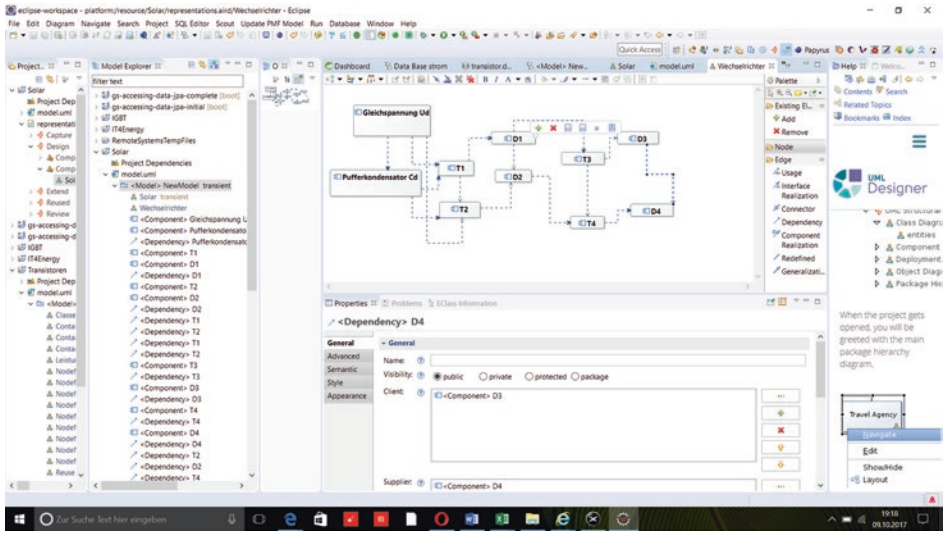


Abb. 2.45 Darstellung der Abhängigkeit zwischen PV-Komponenten mit Hinblick auf Abhängigkeit zwischen Dioden 3 und 4

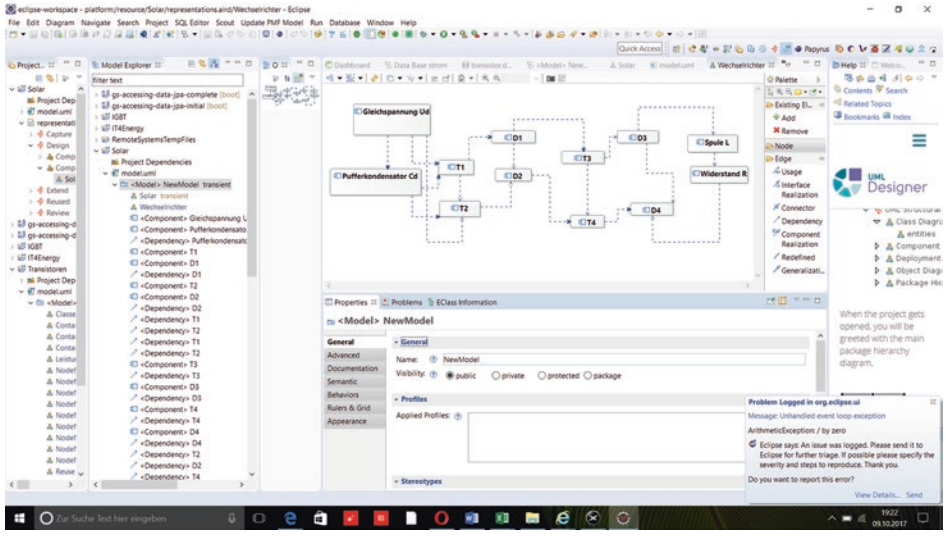


Abb. 2.46 Darstellung der Abhängigkeit zwischen Gleichspannung und Komponenten des Wechselrichters mit Hinblick auf Abhängigkeit zwischen Dioden und Last

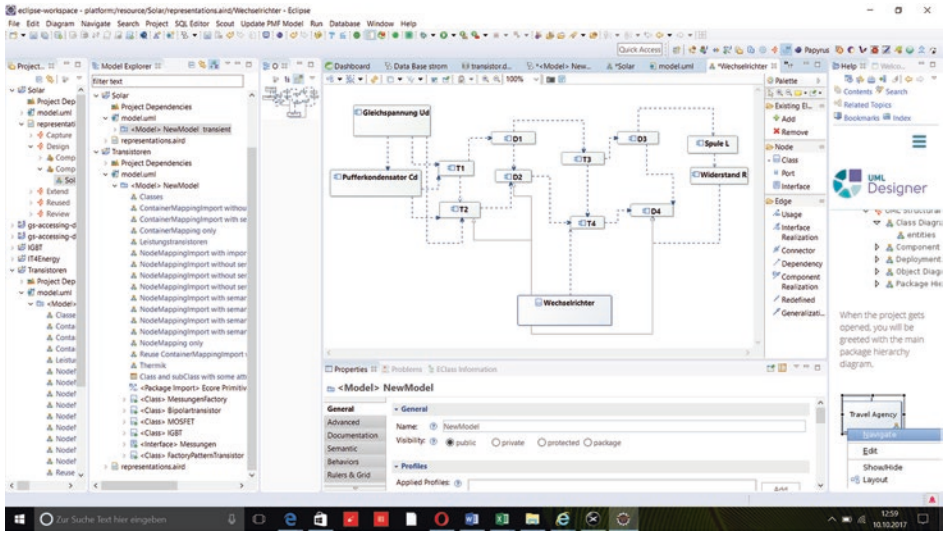


Abb. 2.47 Überblick über Beziehungen zwischen der Klasse *Wechselrichter* und ihren Komponenten

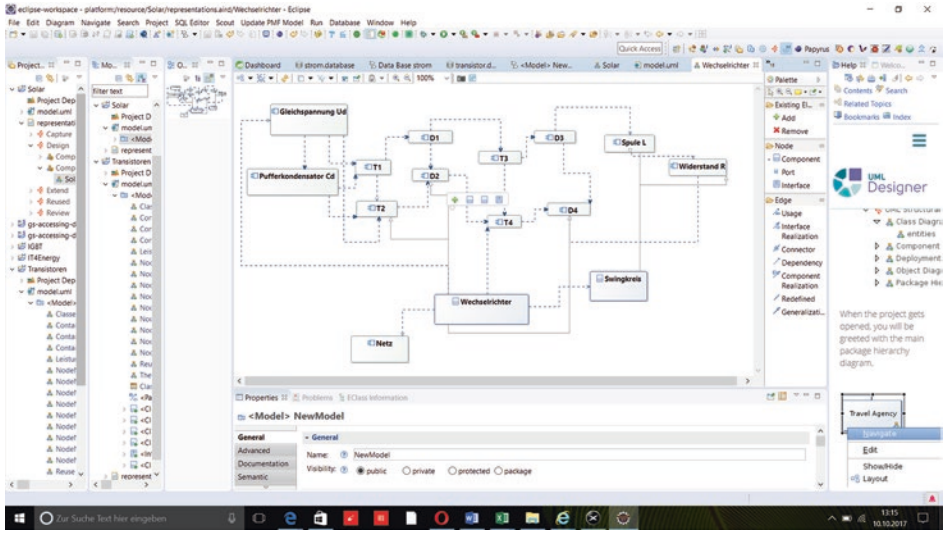


Abb. 2.48 Darstellung der Stromeinspeisung von dem Solargenerator bis zum Netz über Komponente des Wechselrichters mithilfe von *Ports* und *Connectors*

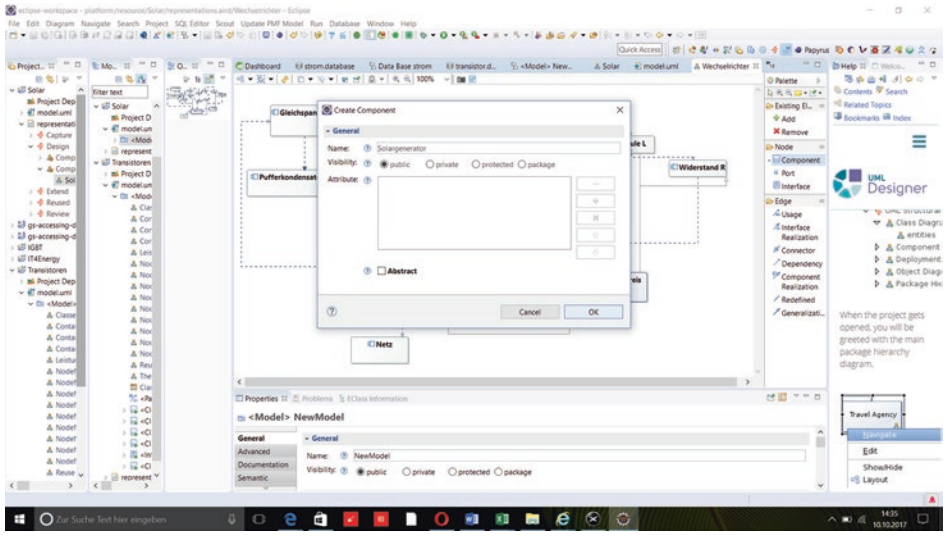


Abb. 2.49 Überblick über das Erstellen der Komponente *„Solargenerator“*

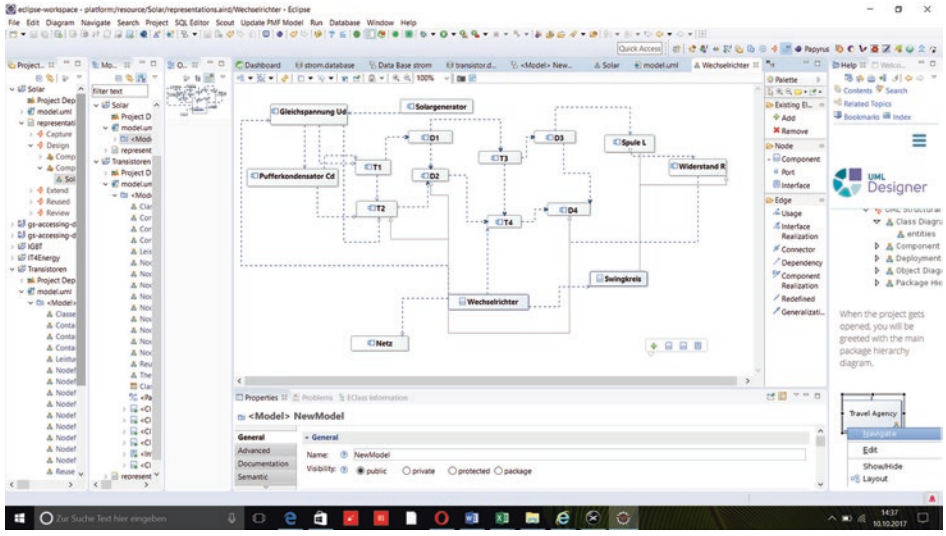


Abb. 2.50 Erstellen der neuen Komponente *„Solargenerator“*

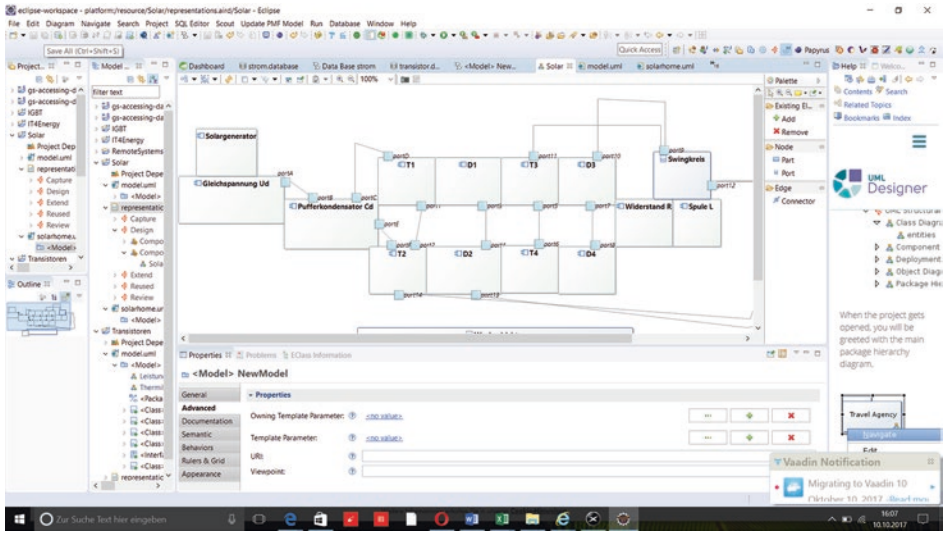


Abb. 2.51 Darstellung des selbstgeführten Wechselrichters mit Hinblick auf *Ports* und *Connectors*

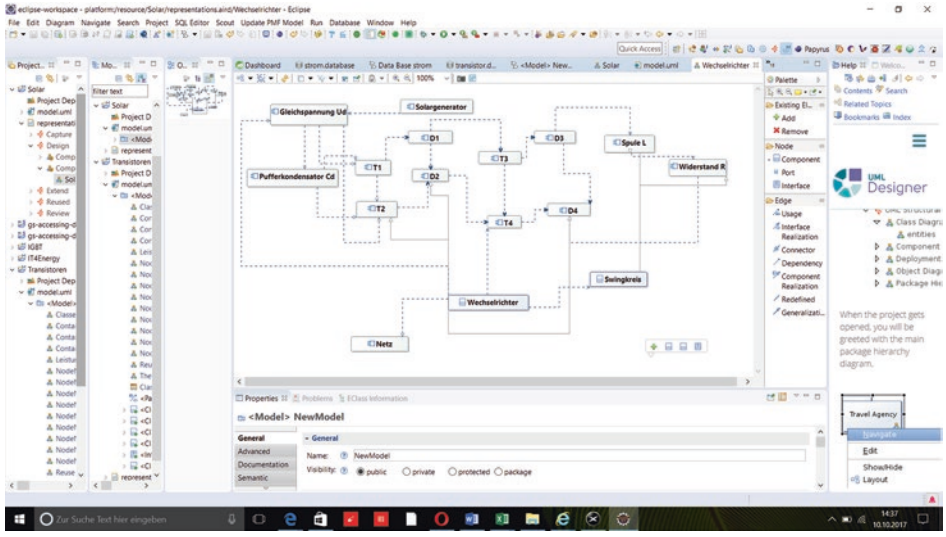


Abb. 2.52 Überblick über Abhängigkeit zwischen dem Wechselrichter und dem Netz

stand und den anderen Komponente des Wechselrichters. Dies erzeugt einen Schwingkreis. Mit den Abb. 2.49, 2.50 und 2.51 ist die Einspeisung des Stroms in das Netz zu visualisieren. Ziel der Modellierung mit dem Framework Obeo-UML-Designer ist es, die leistungselektronischen Komponenten mithilfe des Kompositionsdiagramms darzustellen. Bevor das Kompositionsdiagramm erstellt wird, ist das Projekt, genannt *Solar,* zu erstellen, wie es auf den Abb. 2.41, 2.42, 2.43 und 2.44 zu sehen ist. Wie schon in den vorherigen Abschnitten erwähnt ist, beginnt das Erstellen des UML-Projektes mit dem Editor von *Eclipse-Oxygen* beim Klicken des Register *New,* wobei verschiedene Wizards (Ordner) wie z. B. UML-Designer zur Verfügung stehen. Die Abb. 2.41 zeigt den Inhalt des Ordners UML-Designer. Nach der Auswahl von *wizard „UML Projekt“* öffnet sich ein Fenster, genannt *New Project,* zum Erstellen eines neuen Projektes, genannt S*olar*, wie es auf der Abb. 2.42 zu sehen ist, wobei die Abb. 2.43 das Erstellen des neuen UML-Models in Bezug auf *„Model Object“* und *„XML-Encoding“* zeigt. Sowohl im Model Explorer als auch im Project Explorer ist die Struktur des Projektes *Solar* zu sehen, wie die Abb. 2.44 es darstellt. Sowohl vor dem Projektnamen als auch vor dem Modulnamen steht ein M-Symbol. Dies bedeutet, dass sich das Projekt auf die Modellierung bezieht. Auf der Abb. 2.44 ist das neue erstellte Kompositionstrukturdiagramm, genannt *Composite Structure Diagramm,* Solar zu sehen, wobei die Mappe *„Process Map of <Model> NewModel“* zur Erstellung verschiedener Diagramm-Komponenten wie z. B. Design enthält. Design beinhaltet die UML-Diagramme wie Klassen- oder Kompositionsstrukturdiagramme. Das Analysieren der Trennung zwischen DC- und AC-Seite, wobei ein Pufferkondensator, genannt Cd, zwischen beiden Seiten steht. Auf den Abb. 2.45, 2.46 und 2.47 sind wichtige Komponente des Wechselrichters, die bestehend aus Transistoren T1-4 und Dioden D1-4, Spule L und Widerstand R wobei T1-4 und D1-4, Spule und Widerstand die AC-Seite darstellt. Es ist anzumerken, dass die Last aus L und R besteht. Dies ermöglicht eine Realisierung eines Resonanzwechselrichters. Außerdem zeigen die Abb. 2.45, 2.46 und 2.47 die Kopplung zwischen Gleichspannung und Transistoren. Es ist erkennbar, dass der Kondensator Cd ein Puffer zwischen beiden Seiten darstellt. Verschiedene Abhängigkeitslinien, genannt *Dependency,* sind sowohl im Project Explorer als auch im Project Explorer zu sehen, wie Abb. 2.44 und 2.45 es darstellt.

Auf der Abb. 2.48 ist der Wechselrichter als Klasse mit verschiedenen Beziehungen dargestellt. Die Klasse Wechselrichter ist Teil des Kompositionsstrukturdiagramms, weil ihre Komponenten wie z. B. T1–4, D1–4, R und L dargestellt sind. Abb. 2.49 zeigt die Abhängigkeitsbeziehung zwischen dem Wechselrichter und dem Netz während der Stromeinspeisung. Es ist klar, dass es ohne Netz keine Stromeinspeisung gibt. Außerdem zeigt die Abb. 2.49 die Funktionalität des Wechselrichters in Bezug auf den Schwingkreis, der ein Lastkreis bestehend aus der Spule L und dem Widerstand R enthält. Die Analyse des Kompositionsstrukturdiagramms zeigt, dass die Abb. 2.49 das Puzzle zur Stromerzeugung aus Photovoltaik darstellt, wobei ein Solargenerator, bestehend aus mehreren Solarmodulen, einen Gleichspannung erzeugt. Diese ist mithilfe des Resonanzwechselrichters, bestehend aus Leitungstransistoren (wie z. B. IGBT) und Resonanzelemente, zu der Wechselspannung umgewandelt. Anschließend ist die Stromeinspeisung

mithilfe des Netzes ermöglicht. Die Abb. 2.49 zeigt Interaktionspunkte, genannt *„Ports“*, zwischen den Komponenten des Diagramms. Vom Solargenerator zum Netz gibt es verschiedene Komponente, die untereinander verbunden sind. Abb. 2.50 gibt einen Überblick über das Erstellen der Komponente, genannt *„Solargenerator“*, in Bezug auf *Namen, Sichtbarkeit* und *Merkmal.* Auf der Abb. 2.49 ist der Schwingkreis mithilfe der Interaktionen zwischen der Last, bestehend aus R und L, und den anderen Komponenten des Resonanzwechselrichters zu analysieren. Abb. 2.51 verdeutlicht die Verbindungen mittels *Connectors* und *Ports* zwischen verschiedenen Komponenten zur Stromeinspeisung von dem Solargenerator bis zum Netz über Resonanzwechselrichter, wobei der letzte aus Transistoren Dioden und Last besteht. Mit der Abb. 2.52 ist die Verbindung zwischen dem Wechselrichter und dem Netz zu visualisieren: Dies bedeutet, dass der Wechselrichter mit dem Netz gekoppelt ist.

Leistungstransistoren als elektronische Schalter ermöglichen die Konstruktion von deutlich effizienteren Geräten. Die dabei benutzte H-Brückenschaltung bildet die Grundlage jedes Wechselrichters: Vier Halbleiterschalter öffnen und schließen dabei abwechselnd paarweise über Kreuz, sodass sich die Polarität der mittleren *„Brücke“* jedes Mal umkehrt. Die zeitliche Steuerung der Halbleiter bestimmt dabei die Frequenz der Umpolung und damit der ausgangsseitigen Wechselspannung: Im einfachsten Fall wird 100-mal pro Sekunde zwischen den Schaltzuständen *„T1+T4 offen“* und *„T2+T3 offen“* umgeschaltet – es ergibt sich eine Rechteckwechselspannung von 50 Hz.

Die Modellierung der Wechselrichterschaltung mit Obeo-UML-Designer ermöglicht das Verständnis der leistungselektronischen Funktionsweise der PV-Wechselrichter.

Auf diese Weise funktionierten die ersten Halbleiter-Wechselrichter, die Thyristoren als Schaltelemente einsetzten und sich schnell als robust und zuverlässig erwiesen. Durch die Weiterentwicklung der Halbleitertechnik ist inzwischen aber wesentlich mehr möglich. Moderne Leistungstransistoren haben maximale Schaltfrequenzen von mehreren 10.000 Hz, können also erheblich schneller schalten, als für die Ausgangsfrequenz von 50 Hz erforderlich wäre. Genau das macht man sich mit der Technik der Pulsweitenmodulation zunutze: Eine wesentlich schneller getaktete Brückenschaltung erzeugt viele kurze Spannungspulse von unterschiedlicher Dauer (Pulsweite), die im zeitlichen Mittel das gewünschte Ausgangssignal ergeben. Der gepulsten Spannung lässt sich somit jede beliebige Signalform aufmodulieren – selbstverständlich auch der gewünschte Sinusverlauf. Eine Drosselspule glättet das aus kurzen Pulsen zusammengesetzte Signal (Tiefpassfilter): Das Ergebnis ist eine saubere, sinusförmige Wechselspannung. Um die geforderte Spannungshöhe (230, 400 oder 20.000 V) zu erreichen, ist der Wechselrichterbrücke in der Regel ein Transformator nachgeschaltet. Dieser sorgt zusätzlich für eine galvanische Trennung von DC- und AC-Netz. Es gibt aber auch transformatorlose Wechselrichter: Die Geräte sind kleiner, entsprechend leichter und erreichen etwas bessere Wirkungsgrade. Die gewünschte Höhe der Ausgangspannung wird hier über einen Hochsetzsteller erreicht, der der Wechselrichterbrücke vorgeschaltet ist.

Abb. 2.41, 2.42, 2.43, 2.44, 2.45, 2.46, 2.47, 2.48, 2.49, 2.50 und 2.51 geben Überblicke über die Modellierung der Zusammensetzung von Wechselrichter-Komponenten

und Gleichspannung aus dem Solarmodul. In Bezug auf die Schaltung von Wechselrichtern ist auf den Abb. 2.45, 2.46, 2.47, 2.48, 2.49, 2.50 und 2.51 die Funktionsweise der IGBT zu analysieren, wobei diese Transistoren synchron arbeiten, d. h., IGBT 1 und 4 werden gleichzeitig ein- und ausgeschaltet, ebenso mit IGBT 2 und 3. Abb. 2.49, 2.50, 2.51 und 2.52 stellt die Visualisierung der Interaktionen zwischen Komponenten des netzgekoppelten Wechselrichters dar. Umwandlung des Solar-Gleichstroms in Wechselstrom, Einspeisung des umweltverträglich erzeugten Stroms ins Versorgungsnetz und Überblick über die Resonanzelemente in Stromrichterschaltung stellen die Modellierung des Kompositionsstrukturdiagramms dar, wie es auf den Abb. 2.49, 2.50, 2.51 und 2.52 zu sehen ist.

2.3 Zustandsdiagramm von Obeo-UML-Designer

Zustandsautomaten (state machines) sind Diagramme zur Spezifikation des Verhaltens von Classifiern (Elementen). Cassifier können sein: Klassen, Komponenten, Systeme u. a. Zustandsautomaten beschreiben das Verhalten der Elemente während ihres Lebenszyklus durch Darstellung der möglichen Zustände und Zustandsübergänge [6].

Dieser Abschnitt fokussiert auf die Erstellung von Zustandsdiagrammen mithilfe des Frameworks Obeo-UML-Designer auf Basis von Java-Framework RapidClipse. RapidClipse ist eine freie Eclipse-Distribution für Rapid-Cross-Platform-Development mit Java. RapidClipse ist ein Erbe vom Java-Framework Eclipse, aber vorkonfiguriert, mit zusätzlichen Tools erweitert und an vielen Stellen optimiert, um die professionelle Anwendungsentwicklung mit Java und Eclipse zu ermöglichen. Das neue Java-Framework RapidClipse vereinfacht den Start mit Eclipse und die Anwendungsentwicklung mit Java erheblich – aber dabei darf man nicht vergessen – RapidClipse ist Eclipse. Durch die Vielzahl an Eclipse-Plugins, -Tools und -APIs, die in RapidClipse zum Einsatz kommen, sowie durch die mächtige Eclipse-IDE selbst, ergeben sich zahlreiche Einstellungs- und Konfigurationsmöglichkeiten [9].

Mithilfe des Zustandsdiagramms sind verschiedene Zustände und Übergänge der Systeme auf dem Diagramm-Editor von Obeo-UMl-Designer mit dem *„Java-Framework RapidClipse“* zu veranschaulichen. Wie mit Editor von Eclipse-Oxygen ist das Zustandsdiagramm-Beispiel mit RapidClipse 3.1 auf Basis von Eclipse-Neon, wobei die Neon-Version die Vorgängerin der Version Oxygen wurde.

Das Erstellen eines Modellierungsdiagramms mit RapidClipse ist wie mit dem Java Framework Eclipse ähnlich. Abb. 2.53 zeigt ein erstelltes Zustandsdiagramm mit dem Framework-Obeo UML-Designer. Es ist zu bemerken, dass Abb. 2.53 sowohl das Logo des Frameworks RapidClipse als auch verschiedenen Tools wie bei Eclipse darstellt. Das Diagramm, genannt *„NewModel State Machine Diagram“*, ist in der Modellierungsstruktur vom Obeo-UML-Designer, genannt *„representations.air“*, erstellt, wie auf der Abb. 2.53 zu sehen ist, wobei das Modellierungsprojekt nach *„modellierungs4strom“* genannt ist. Außerdem ist oben auf der Abb. 2.53 den Namen *„RapidClipse“* zu sehen.

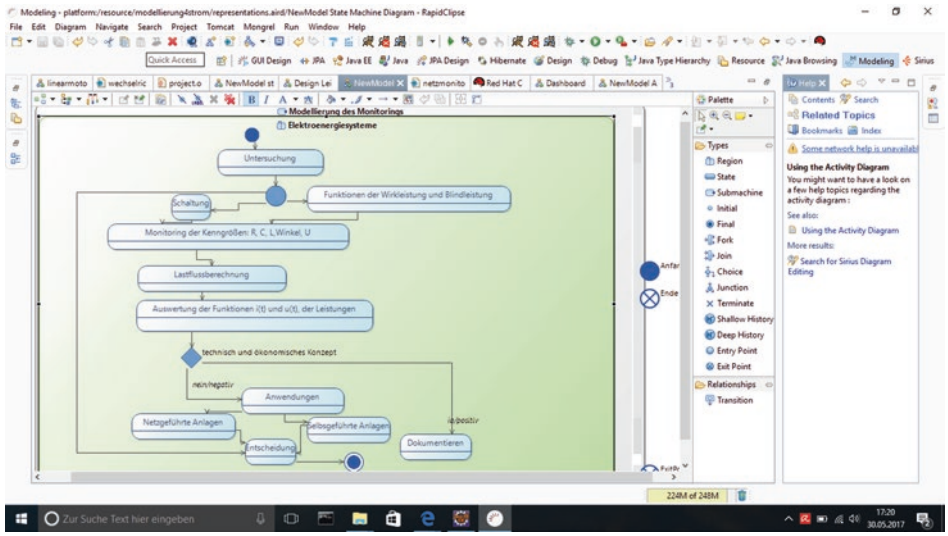


Abb. 2.53 Aktivitätsdiagramm zum Analysieren eines Systems für Elektroenergie

Abb. 2.53 stellt eine Anwendung zum Analysieren eines Systems für Elektroenergie. Die erste Beim Analysieren der Abb. 2.53 ist es anzumerken, dass sie zwei folgende Fragen : *„Was soll die Anwendung tun?“* und *„Was könnte die Anwendung sein?“* versteckt. Die Fragen beziehen sich auf ein Aktivitätsdiagramm, das ein Verhalten des Systems darstellt. Hierbei ist das Verhalten dieses Diagramms zu analysieren. Gemäß Abb. 2.53 stellt ein Zustandsdiagramm eine Synchronisierung der Elemente des Diagramms, die in einer Aktivität des UML-Modells existieren. Sie sind in einem Aktivitätsdiagramm sichtbar.

2.3.1 Überblick über Erstellungstools des Zustandsdiagramms

Das Erstellen der Zustandselemente ist mithilfe der Palette, wie auf der Abb. 2.53 zu sehen ist, realisiert. Auf der Abb. 2.53 sind die Werkzeuge als Elemente zum Erstellen des Zustandsdiagramms dargestellt, wobei ein Element nach dem Anklicken auf das Arbeitsblatt des Diagramms entsprechend dem Package im UML-Model erstellt wird. Die Palette verfügt über *„Tooltip tools“* zur Erstellung von Werkzeugelementen (Tab. 2.1).

2.3.2 Notationselemente

Abb. 2.53 zeigt verschiedene Notationselemente für das Beispiel. Initial State und Final State stellen den Beginn und Ende des Lebenszyklus eines modellierten Elementes dar. Erzeugen und Löschen eines Objektes hängen von Anfangs- bzw. Endzustand ab. Wie

Tab. 2.1 Überblick über Erstellungstools des Zustandsdiagramms mit Obeo-UML-Designer

Elemente	Beschreibung
Region	Gruppe von logischen Zuständen
State	Elemente eines spezifischen Zeitpunktes
Submachine	Unterzustand
Initial	Beginn einer Transition
Final	Ende einer Transition
Fork (Gabelung)	Aufteilung einer eingehenden Transition in mehrere ausgehende parallele Transitionen
Join (Vereinigung)	Zusammenführen von aufgeteilten parallelen Zweigen
Choice	Knoten zum Darstellen von komplexen Zusammenhängen zwischen Zuständen
Junction	Verbindung mehrerer Transitionen
Terminate	Ausdruck des Endes der *„State Machine“*
Shallow history	Darstellung des letzten aktiven Unterzustandes in seinem Zustand-Container
Deep history	Darstellung des letzten aktiven Unterzustandes eines in einem zusammengesetzten Zustand enthaltenen Zustandes.
Entry point	Darstellung des Beginns der State Machine
Exit point	Darstellung des Endes der State Machine

Werkzeuge des Zustandsdiagrammes vom Obeo-UML-Designer

auf der Abb. 2.53 zu sehen ist, wird der Zustand mit einem Namen als Rechteck mit abgerundeten Ecken dargestellt. Die Zustände sind durch eine waagrechte Linie unterteilt. Auf der Abb. 2.53 ist eine Region, genannt *„Elektroenergiesysteme“*, mit verschiedenen Zuständen und Transitionen zu sehen, wobei sie parallele Abläufe darstellt und Teil von *„Submachine“* ist. Transition gehört zu dem Ordner *„Relationship“* oder Verbindung und definiert z. B., wie auf der Abb. 2.53 zu sehen ist, einen Zustandsübergang vom Zustand *„Lastflussberechnung“* zum Zustand *„Auswertung der Funktionen“*. Transitionenverwaltung mit Obeo-UML-Designer ermöglicht sowohl das Erstellen oder Löschen als auch die Verbindung einer Transition.

2.3.3 Anwendung des Zustandsdiagramms in der Energietechnik

Wie auf der Abb. 2.53 zu sehen ist, modelliert das Zustandsdiagramm Objekte wie z. B. Kenngröße zum Monitoring der Elektroenergiesysteme [10]. Vom Anfangs- (Initial) bis zum Endzustand (Final) stellt das Diagramm eine *„Submachine State“*, d. h. Unterzustandsautomatenzustand mit verschiedenen Notationselemente wie z. B.

Zuständen, Transitionen, Kreuzungen, Entscheidung, Terminator, Eintritt- und Austrittpunkt dar. Wie auf der Abb. 2.53 zu sehen ist, beginnt die Modellierung mit dem Analyse-Zustand. Es folgt eine Kreuzung, die drei Transitionen verbindet. Einer der Transition beeinflusst den Endzustand d. h. die Entscheidung. Das Monitoring der Kenngrößen wie z. B. Widerstand R, Kapazität C, Drosselspule L, Phasenwinkel und Spannung U hängt, wie auf der Abb. 2.53 zu sehen ist, sowohl von dem Zustand *„Schaltung“* als auch von dem Zustand *„Funktionen der Wirk- und Blindleistung“* ab. Gemäß der Abb. 2.53 stellt der Zustand *„Schaltung“* die Struktur des Elektroenergiesystems mit Blindleistungskompensationsanlagen während des Zustandes *„Funktionen der Wirkleistung und Blindleistung“* Art und Belastungsverhalten der zeitlichen Verläufe der Wirk- und Blindleistung mit linearer und nichtlinearer Spannungscharakteristik dar. Diese beiden Zustände ermöglichen mithilfe der parallelen Transition das Monitoring der Kenngrößen Widerstand R, Kapazität C, Drosselspule L, Phasenwinkel und Spannung U, wobei es ein Auslöser für die Transition gibt, welche zum Zustand *„Lastflussberechnung“* führt. Dieser Zustand stellt eine Methode der numerischen Analyse bezüglich der Ermittlung des Wirk- und Blindleistungsflusses mithilfe von Spannung und Phasenwinkel dar. Anschließend führt eine Transition zum Ergebnis der Lastflussberechnung in Bezug auf die zeitlichen Verläufe des Stroms und der Spannung. Dieser Zustand definiert Faktoren der Elektroenergiequalität, Leistungen und Leistungsfaktoren [10]. Das Ergebnis der Lastflussberechnung führt zu einer technischen und ökonomischen Wahl, die entweder positiv oder negativ ist. Diese Wahl bezieht sich auf die Analyse des Ergebnisses zur Einhaltung der Qualitätsfaktoren. Wenn die Wahl zur Einhaltung des technisch wirtschaftlichen Konzeptes mit einer positiven Transition, werden die Ergebnisse dokumentiert. Aber wenn das nicht der Fall ist, gibt es eine Kaskade der Transitionen von dem Zustand *„Anwendungen“* bis zum Endzustand *„Entscheidung“*, wobei der Zustand *„Anwendung“* mithilfe von zwei parallelen Transitionen zu den Zuständen *„Netzgeführte-“* und *„Selbstgeführte“* Anlagen führt. Beide Zustände führen parallel zum Endzustand *„Entscheidung“*. Dies beendet das Monitoring.

2.4 Komponentendiagramm

Ein Komponentendiagramm stellt ein Strukturdiagramm zum Implementieren der Funktionalitäten und zum Abwickeln der Kommunikation über fest definierte Schnittstellen dar [11]. Im Bereich Softwaresysteme ist das Ziel eines Komponentendiagramms, das Modulieren kommunizierender Softwarekomponente zu ermöglichen, wobei die Entwicklung auf die Erstellung von Design Patterns fokussiert.

Komponentendiagramme geben einen Überblick über eine bestimmte Sicht auf die Struktur des modellierten Systems. Das Komponentendiagramm vom Obeo-UML-Designer, wie auf der Abb. 2.54 zu sehen ist, stellt eine Diagramm-Palette zum einen mit Komponenten, Klasse, Interface und Ports und zum anderen mit

Abhängigkeitsbeziehungen wie z. B. *„Dependency“* Konnektoren. Notationselemente aus der Palette ermöglichen das Darstellen des Komponenteninneren wie bei Klassen- oder Kompositionsstrukturdiagrammen [6]. Modellierung von komponentenbasierten Softwaresystemen wie z. B. Java EE ist mithilfe von Komponentendiagrammen ermöglicht.

2.4.1 Komponentenmodell von Java EE

Komponentenmodelle stellen die Struktur und das Verhalten von Komponenten in den Systemen dar. Zur Trennung der Komponenten sind komplexe webbasierte Softwaresysteme wie z. B. Java Enterprise Edition (Java EE) in einer Multi-Schichten-Architektur [11].

Java Enterprise Edition Version 8 von Oracle stellt eine vierschichtige Standard-Architektur für Java-basierte Business-Anwendungen [12]. Die Anwendungslogik ist mit Komponenten, welche abhängig von der Funktionalität auf unterschiedlich Schichte verteil sind, realisiert.

Abb. 2.54 zeigt die Java-EE-Architektur Version 8, genannt Java EE 8, in Bezug auf das Komponentendiagramm, welches vier Schichte*(Tier)* darstellt.

Gemäß Abb. 2.54 verfügt die Java EE über vier Schichte: Client-, Web-, Business- und Enterprise-Information-(EIS-)Tier. Die Architektur stellt auch drei Anwendungsschichten: Client Machine, Java-EE-Server und Database-Server dar, wobei die Abwendungen auf drei unterschiedlichen Lokationen verteilt sind.

Java-EE-Spezifikation definiert folgende Java-EE-Komponenten:

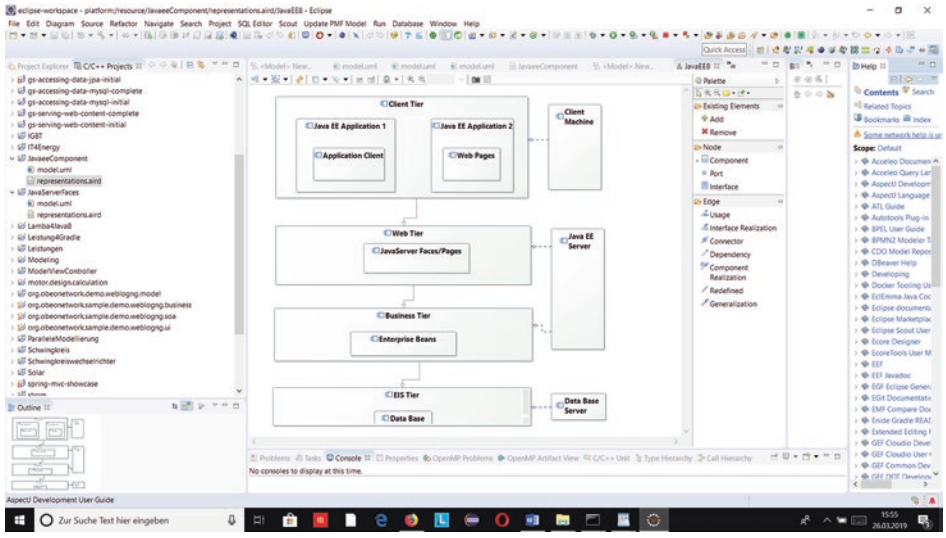


Abb. 2.54 Komponentendiagramm mit vier Schichten genannt *„Tier“* für die Java-EE-Anwendungen

- Komponenten für Clients- und Apples-Anwendungen werden auf dem Client ausgeführt.
- Komponenten der Technologie aus Java Servlet, JavaServer Pages (JSP) und JavaServer Faces (JSF) stellen Web-Komponenten zum Ausführen auf dem Server dar.
- EJB-Komponenten, genannt Enterprise Beans, sind Business-Komponenten zum Ausführen auf dem Server.
  Java-EE-Komponenten sind auf Basis der Programmierungssprache Java geschrieben, wobei sie wie jede Sprache kompiliert [12].

2.4.2 Komponenten für Java EE

Wie auf der Abb. 2.54 ist Java-EE-Client sowohl für Web als auch für Anwendungen definiert. Ein Web Client besteht aus zwei Teilen: *„Dynamic Web Page“* und *„Web Browser“*. Zum einen sind HTML und XML von dem ersten Teil mithilfe von Web-Komponenten der Web-Schichte erzeugt und zum anderen bearbeitet der zweite Teil die gesendeten Seiten des Servers. Manchmal ist *„Web Client“* nach *„Thin Client“* genannt, wobei Thin Client oft kein Zutun mit den Datenbanken hat.

Gemäß der Abb. 2.54 ist der Anwendungs-Client auf dem Client-Rechner ausgeführt und ermöglicht dem Benutzer die Realisierung von Aufgaben auf Basis von Graphical User Interface (GUI) mithilfe von HTML oder XML. Hierbei ist GUI mithilfe von Swing API oder abstract Window Toolkit (AWT) erstellt. Anwendungs-Clients habe einen direkten Zugang zu den Enterprise Beans, welche auf der Business-Schicht ausgeführt sind.

Gemäß Java-EE-Spezifikation zählen Java-Beans-Komponenten nicht zu den Java-EE-Komponenten.

Java-EE-Komponenten sind sowohl erzeugte Servlet als auch Web Page, die JSF- und/oder JSP-Technologie ermöglichen. Servlets sind Java-Klassen zum dynamischen Verarbeiten der Anforderungen und der entsprechenden *„Requests“* eines Konstruktes. JSP Pages stellen Textdokumente zum Ausführen von Servlets dar, wobei sie statische Inhalte erzeugen. Zum einen ist JSF-Technologie auf Basis von Servlets- und JSP-Technologie aufgebaut und zum anderen stellt Web-Anwendungen eine Benutzerinterface-Komponente zur Verfügung. Laut der Java-EE-Spezifikation sind sowohl statische HTML-Seiten und Applets als auch Klassen von dem Server nicht Teil der Web-Komponenten.

Abb. 2.54 zeigt die Struktur der Geschäftsschicht, genannt Business-Tier, in Bezug auf Enterprise Bean, welche sowohl auf der Business- als auch auf der Web-Schicht ausgeführt sind. Auf der Abb. 2.54 ist zu sehen, dass zum einen *„Enterprise Beans“* Daten von Clients-Programmen erhalten und zum anderen diese verarbeiten und sie anschließend der Enterprise-Information-System-(EIS-)Schicht zum Speichern senden. Das heißt, ein Enterprise Bean holt Daten aus dem Speicher, verarbeitet sie bei Bedarf und sendet sie zum Client-Programm zurück [12].

Abb. 2.54 gibt einen Überblick über die Datenbankschicht, genannt Enterprise Information System (EIS), welche Software und Enterprise Infrastructure Systems stellen wie z. B. Enterprise Resource Planning (ERP), Mainframe Transaction Processing und Datenbank, wobei eine Java-EE-Anwendungskomponente Zugänge zu EIS für die Datenbankverbindung benötigen.

2.4.3 Komponenten für JSF, JPA und CDI

Die JavaServer-Faces-Technologie baut auf den Java EE bestehenden Standards für Web-Applikationen auf und bietet ein serverseitiges Komponentenmodell für Oberflächenobjekte [13]. Java EE stellt mithilfe von JFS, JPA und CDI Anwendungen von Weboberflächen dar. Die Komponenten von JSF, JPA und CDI ermöglichen die Entwicklung von Enterprise-Anwendungen auf Basis von Java-Enterprise-Standards [14].

Abb. 2.55 gibt einen Überblick über die Strukturen der Komponenten JSF, JPA und CDI in Bezug auf Webanwendungen mit Java-EE-Web-Profile. Gemäß Abb. 2.55 stellen die drei Komponenten ein Zusammenspiel der entsprechenden Frameworks JSF, JPA und CDI. Abb. 2.55 zeigt die Komponenten zur administrativen Kundenverwaltung in Bezug auf das Erstellen, Lesen, Bearbeiten und Löschen von Kundendaten [14].

2.4.3.1 JavaServer Faces (JSF)

JSF stellt Implementierungen der Anwendungen in Java EE für die webbasierte Oberfläche dar.

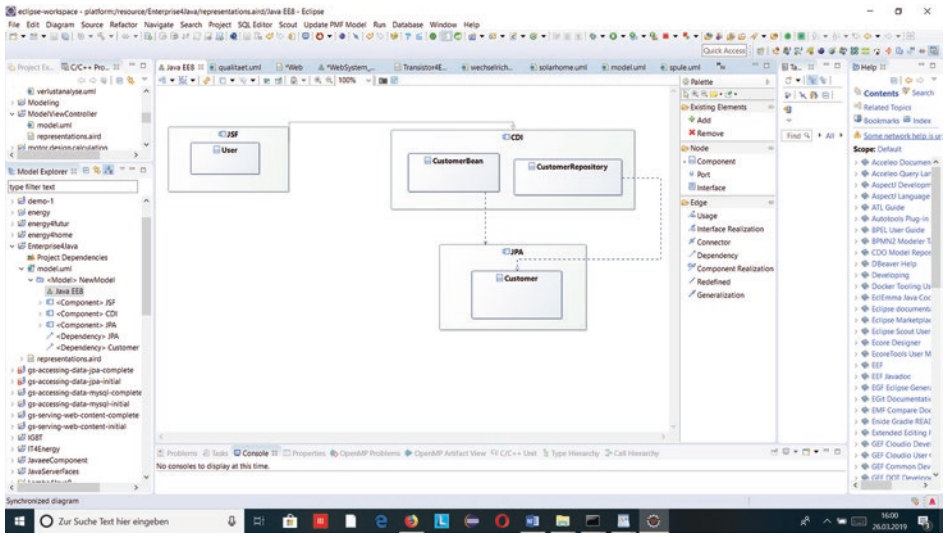


Abb. 2.55 Überblick über Java EE 8 mit JSF, JPA und CDI

Laut Java EE 8 Spezifikation ist die JSF die Spezifikation für ein komponentenbasiertes Model View Controller (MVC). Die aktuelle Version ist JSF 2.3 und verfügt über mehrere Features wie z. B. <f:importConstants/>, DataModel-Implementierungen, CDI-Ersetzung für @ManagedProperty, Nutzung von CDI und Unterstützung von @Inject für JSF [15]. Für JSF-Spezifikation gibt es verschiedene Implementierungen, wobei Mojarra als Referenzimplementierungen dient und *MyFaces* als ein wichtiges Open-Source-Projekt, das JSF-Implementierung für Oracle-Spezifikation darstellt.

Abb. 2.55 zeigt wie die drei Komponenten JSF. CDI und JPA Abhängigkeitsbeziehungen darstellen. Zum einen gemäß der Abb. 2.55 unterstützt JSF CDI-Bestandteile mittels Annotationen oder Injektionsmöglichkeit und zum anderen hat die JSF-Komponente Zugriff auf Bean-Klassen in der CDI-Komponente. Mit der Abb. 2.55 erfolgt die Sichtbarkeit der Klasse *User* mithilfe der *@Named*-Annotationen, wobei das Erzeugen und Entfernen der Objekte der Klasse *User* vom CDI-Container übernommen wird [13]. Laut Java-EE-Spezifikation können JSF-Seiten über die Expression Language (EL) auf Objekte von *@Named*-annotierten Klassen wie z. B. *User* zugreifen, sofern der CDI-Container sie erstellt hat.

2.4.3.2 Java-Persistence-API (JPA)

JPA-Spezifikation mit JPA 2.3 stellt das Speichern und Lesen der Daten mithilfe objektrelationaler Abbildungen in relationalen Datenbanken dar. Mit Java EE 8 werden Abfrageergebnisse als Stream API, *Repeatable Annotationen* und die Nutzung von *CDI-Injection* in *Attribute Convertern* ermöglicht. JPA mit *EntityManager* verbindet Datenbank- und Java-Komponenten, d. h., Java-Objekte werden in die Datenbanken sowohl geschrieben als auch gelesen.

Mit Abb. 2.55 sind Abhängigkeitsbeziehungen zwischen die Komponenten JPA und CDI in Bezug auf *Injections.* Gemäß Abb. 2.55 lässt sich der *EntityManager* über die Annotation *@PersistenceContext* in das *Repository* injizieren, wobei die Klasse *CustomerRepository* sowohl das Laden der Daten der Kunden für die Übersichtseite als auch das Suchen der Kunden über die Kundennummer ermöglicht [14]. Konkret erfolgt das Laden der Daten über den Primärschlüssel. Außerdem ermöglicht JPA mithilfe der objektorientierten Abfragesprache, genannt JPQL, das Setzen der Abfrage-Parameter wie z. B. *setParameter* über die Query-Abstraktion.

2.4.3.3 Contexts and Dependency Injection (CDI)

Mit Java EE 8 stellt CDI Komponententechnologie zur Abhängigkeitsmanagement dar. Der Einfluss von CDI ist mithilfe der Annotationen wie z. B. *@Named-, @RequestScoped-,* oder *@PostScoped-Annotation* erkennbar. Abb. 2.55 zeigt, wie die *CDI-* die Abhängigkeitsbeziehungen sowohl mit JSF- als auch mit *JPA-*Komponente darstellt. Hierbei ermöglicht die Struktur der CDI-verwaltete Klasse, genannt *CustommerBean,* ihre Sichtbarkeit in JSF-Komponente mithilfe der *@Named-Annotation* [14]. Außerdem ist *@RequestScoped-Annotation* für die Lebensdauer der Klasse *CustommerBean* der CDI-Komponente, wobei während der

Aktion des *Requests* der Zugriff auf eine Instanz der Klasse *CustommerBean* von JSF-View-Dateien erfolgt. Die Abhängigkeitsbeziehung zwischen CDI- und JPA-Komponenten gemäß Abb. 2.55 stellt das Einführen eines *„Daten-Repositorys“* mithilfe der Annotationen dar. Dies ermöglicht das Heben einer Repository-Klasse wie z. B. *CustomerRepository* in die CDI-Komponente. Abb. 2.55 zeigt den Aufbau der Abhängigkeitsbeziehungen zwischen CDI- und JPA-Komponenten in Bezug auf die Kapselung des Datenbankhandlings mithilfe der *Repository-Schicht*. Hierbei holt die Klasse *CustomerRepository* die Kundendaten (Klasse *Customer*) aus der Datenbank.

Wie in der CDI-Komponente der Abb. 2.55 zu sehen ist, erfolgt die Injektion der Klasse *CustomerRepository* in die Klasse *CustomerBean* durch die *@Inject-Annotation.*

2.5 Verteilungsdiagramm (Deployment-Diagramm)

Das Verteilungsdiagramm stellt ein Strukturdiagramm dar und gibt einen konkreten Überblick über die Struktur der Systeme, wie es auf der Abb. 2.56 zu sehen ist. Das funktioniert wie ein asynchrones Diagramm, in dem der Benutzer vorhandene Elemente ins Diagramm einfügen muss. Die Palette des Editors des Obeo-UML-Designers umfasst typischerweise zum einen Typen (oder „Types“) wie z. B. Rechnerknoten(*„Nodes“*), Artefakte(*„Artifacts“*), *„Device“*, *„Execution Environment“* und zum anderen Verbindungen (*„Relationships“*) wie z. B. Verteilungsbeziehungen (*„Deployment“*), Link, Abhängigkeitsbeziehungen (*„Dependency“*). Abb. 2.56 zeigt die Struktur der Palette des Verteilungsdiagramms mit dem Obeo-UML-Designer. Diese Struktur zeigt die Symbole der Werkzeugelemente zum Erstellen von Tooltips.

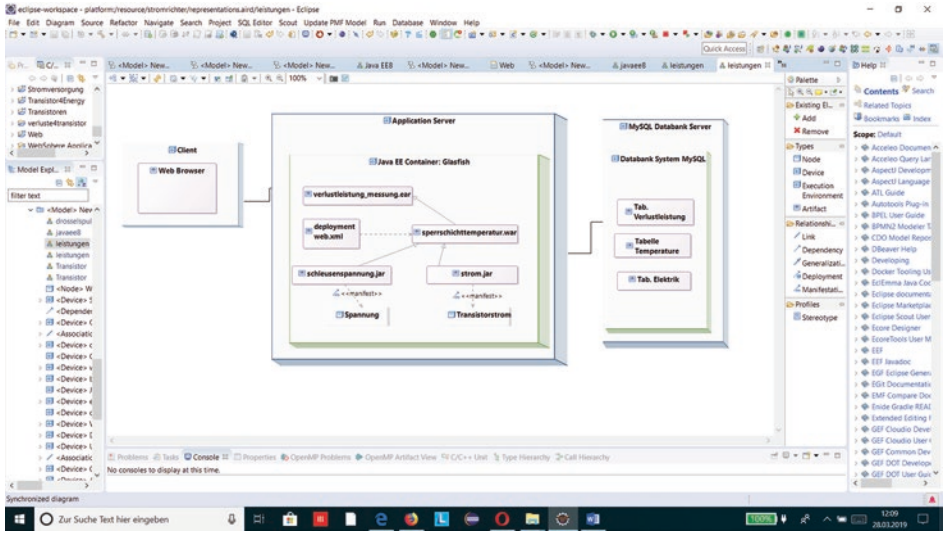


Abb. 2.56 Kommunikation zwischen Client, Application Server und MySQL -Datenbank-Server

Tab. 2.2 Überblick über die Rolle der Erstellungstools zum Erstellen des Verteilungsdiagramms

Elemente	Beschreibung
Node	Es ist ein physikalisches Element, welches ein Verteilungssystem enthält
Device	Das ist eine physikalisch elektronische Resource mit verarbeitender Fähigkeit
Execution Environment	Darstellung eines Knotens, in dem die Artefakte verteilt sind
Artifact	Ein physikalisches Stück der Information, welches von einem System verwendet oder hergestellt ist

Werkzeuge zum Erstellen von Verteilungsdiagrammen

Tab. 2.2 beschreibt die Rolle dieser Werkzeugelemente in Bezug auf das Erstellen des Verteilungsdiagramms. Dies ermöglicht dem Benutzer zum Erstellen von Verteilungsdiagrammen mithilfe von *„Device“*, *„Execution Environment“*, *„Node“* und *„Links“*.

2.5.1 Device für *„Application Server Java EE“*

Java-EE-Anwendungen sind während der Ausführung in einem Applicationsserver wie z. B. Glasfish in standardisierte Archive verpackt. Abb. 2.56 gibt einen Überblick über die Struktur des Java-EE-Application-Servers in Bezug auf Java Archive (JAR), Web Archive (WAR) und Enterprise Archive (EAR). Hierbei stellt der Java-EE-Application-Server ein *„Device“* dar und besteht aus einem *„Execution Environment“*, genannt *„Glasfish-Java-EE-Container“*. Dieser Container besteht aus mehreren Artefakten wie z. B. WAR-Datei, EAR-Datei, JAR-Datei oder XML-Datei. EAR-Dateien verfügt über Java-EE-Modules und optional einen Deployment-Deskriptor, genannt XML-Datei. Laut Java-EE-Spezifikation stellt Java EE Server-Web-Container und EJBs dar.

Mit der Abb. 2.56 sind im Java EE Container verschiedene Archive wie z. B. verlust_messung.ear, sperrschichttemperatur.war, oder strom.jar. Die WAR-Dateien sind für den Web-Teil von Java EE-Anwendungen wie z. B. JavaServerPages(JSP) oder Servlets zusammengefasst. Web-Komponenten verwalten die Ausführung von Web Pages, Servelets oder einigen EJB für Java EE Applikation. Wie auf der Abb. 2.56 zu sehen ist, legt der Deployment-Deskriptor web.xml die Installationsinformation vom Web-Archiv wie z. B. sperrschichttemperatur.war fest. EAR-Datei steht für Enterprise Application Archiv und verfügt über mehere Dateien wie z. B. WAR-Datei. Abb. 2.56 gibt einen Überblick über die Beziehungen zwischen verschiedenen Archivdateien. Zum einen stehen die Java Archiv-Dateien *„schleusenspannung.jar“* und *„strom.jar“* mit der WebArchiv-Datei *„sperrschichttemperatur“* mittels Generalisierung in Vererbungsbeziehung und zum anderen steht die letzte mit der EAR-Datei *„verlust_messung.ear“* in Vererbungsbeziehung. Die Nodes *„Spannung“* bzw *„Transistorstrom“* sind die Anwendungen für die *schleusenspannung.jar* bzw *strom.jar*.

2.5.2 Device *„Client“*

Laut Java-EE-Spezifikation ist ein Java-EE-Client entweder ein Web-Client oder ein Applikation-Client. Abb. 2.56 zeigt die Struktur von *„Device“* Client, wobei ein Artefakt, genannt *„Web Browser“*, ein Element des Clients ist. Hierbei gibt es eine Verbindung zwischen *„Client“* und *„Application Server“*, weil der Web Browser die Seiten des Servers ausführt. Diese Verbindung stellt eine indirekte Kommunikation zwischen dem Client und dem Business-Tier ausgeführt auf dem Java-EE-Server mithilfe von *„Web Pages“* dar.

2.5.3 Device MySQL-Datenbankserver

Device (Symbol D) Datenbank gehört zu der EIS-Schicht, welche über die Ausfürungsumgebung (oder Execution Environment mit dem Symbol E), und mehreren Artefakten wie z. B. *„Tabelle Verlustleistung“*, *„Tabelle Temperature“* und *„Tabelle Elektrik“* verfügt. Mithilfe von Java-EE-Connector-Architecture hat der *Application Server* mittels Resource-Adapters den Zugang zum MySQL-Datenbank-Server, wobei es eine Verbindung zwischen den beiden Devices Application-Server und *MySQL-Datenbank-Server* gibt, wie auf der Abb. 2.56 zu sehen ist. Abb. 2.56 enthält die EAR-Datei, genannt *„verlust_messung.ear“*, welche eine *Resource-Adapter-Archive*-(RAR-)Datei, genannt *Resource Adapter,* beinhaltet.

2.6 Zusammenfassung

Das Kapitel fokussierte auf die Thematik *„Modeling4Programming“* und gibt einen Überblick über das Tool *„Codes in Models“* bezüglich der Beziehung zwischen UML-Modellen und Java-Klassen. Editoren von Obeo-UML-Designer und Papyrus werden mithilfe des Eclipse-Frameworks heruntergeladen.

Ein Klassendiagramm stellt die statische Struktur eines Systems oder einer Software dar und beinhaltet den Kern des Analysemodells. Dieses Diagramm verfügt über Teile wie z. B. Paket, Objektklasse, Attribut, Operation zur Modellierung. Eine Klasse wird mit einem Rechteck dargestellt, wobei der Name der Klasse angezeigt wird. Das Klassensymbol kann mit einem Bereich zur Darstellung der Attribute und der Methoden erweitert werden. Für parametrierbare Klassen wird das Klassensymbol mit einem Rechteck erweitert, das den Namen des Parameters enthält. Eine Klasse besteht aus einer Sammlung von Attributen und Methoden: Sie stellen die Eigenschaften der Klassen dar. Attribute (oder Property beim Obeo-UML-Designer) definieren mithilfe von Bezeichnern *„Typ“* und *„Name“* den Zustand einer Klasse. Obeo-UML-Designer ist ein interessantes Framework zur Modellierung des Klassendiagramms. Hierbei nutzt das Framework die Stärke der DSL Sirius 5.

Das Kompositionsstrukturdiagramm dient der Dekomposition und Modellierung von Classifiern, die in Kompositionsbeziehungen zueinander stehen. Es zeigt, wie die Teile eines Classifiers den Classifier selbst bilden. Da es mit dem Kompositionsstrukturdiagramm möglich ist, die Teile (Architekturelemente) eines Systems und ihre Beziehungen darzustellen, wird es auch Architekturdiagramm genannt. Das Kompositionsstrukturdiagramm eignet sich ebenfalls zur Modellierung von Entwurfsmustern.

Zustandsautomaten (state machines) sind Diagramme zur Spezifikation des Verhaltens von Classifiern (Elementen). Classifier können Klassen, Komponenten oder Systeme sein.

Zustandsautomaten beschreiben das Verhalten der Elemente während ihres Lebenszyklus durch Darstellung der möglichen Zustände und Zustandsübergänge. Mithilfe des Zustandsdiagramms sind verschiedene Zustände und Übergänge der Systeme auf dem Diagramm-Editor von Obeo-UMl-Designer mit dem *„Java Frameworks Rapidclipse“* zu veranschaulichen. Wie mit Editor von Eclipse-Oxygen ist das Zustandsdiagramm-Beispiel mit RapidClipse 3.1 auf Basis von Eclipse Neon, wobei die Neon-Version die Vorgängerin der Version Oxygen ist.

Ein Komponentendiagramm stellt ein Strukturdiagramm zum Implementieren der Funktionalitäten und zum Abwickeln der Kommunikation über fest definierte Schnittstellen dar. Komponentendiagramme geben einen Überblick über eine bestimmte Sicht auf die Struktur des modellierten Systems. Das Komponentendiagramm vom Obeo UML-Designer stellt eine Diagramm-Palette zum einen mit Komponenten, Klasse, Interface und Ports und zum anderen mit Abhängigkeitsbeziehungen wie z. B. *„Dependency“*, Konnektoren. Notationselemente aus der Palette ermöglichen das Darstellen des Komponenteninneren wie bei Klassen- oder Kompositionsstrukturdiagrammen. Komponentenmodelle stellen die Struktur und das Verhalten von Komponenten in den Systemen dar. Zur Trennung der Komponenten sind komplexe webbasierte Softwaresysteme wie z. B. Java-Enterprise-Edition (Java EE) in einer Multi-Schicht-Architektur. Java Enterprise Edition Version 8 (Java EE 8) von Oracle stellt eine vierschichtige Standard-Architektur für Java-basierte Business-Anwendungen. Die Anwendungslogik ist mit Komponenten, welche abhängig von der Funktionalität auf unterschiedliche Schichten verteil sind, realisiert. Java-EE-Komponenten sind sowohl erzeugte Servlet als auch Web Page, die JSF- und/oder JSP-Technologie. Servlets sind Java-Klassen zum dynamischen Verarbeiten der Anforderungen und der entsprechenden *„Requests“* eines Konstruktes. JSP Pages stellen Textdokumente zum Ausführen von Servlets dar, wobei sie statische Inhalte erzeugen. Zum einen ist JSF-Technologie auf Basis von Servlets- und JSP-Technologie aufgebaut und zum anderen stellen Web-Anwendungen eine Benutzerinterface-Komponente zur Verfügung. Laut der Java-EE-Spezifikation sind sowohl statische HTML-Seiten und Applets als auch Klassen von dem Server nicht Teil der Web-Komponenten.

Das Verteilungsdiagramm stellt ein Strukturdiagramm dar und gibt einen Überblick über einen konkreten Überblick über die Struktur der Systeme. Das funktioniert

wie ein asynchrones Diagramm, in dem der Benutzer vorhandene Elemente ins Diagramm einfügen muss. Die Palette des Editors des Obeo-UML-Designers umfasst typischerweise zum einen Typen (oder *„types“*) wie z. B. Rechnerknoten(*„Nodes“*), Artefakte(*„Artifacts“*), *„Device“*, *„Execution Environment“* und zum anderen Verbindungen (*„Relationships“*) wie z. B. Verteilungsbeziehungen (*„Deployment“*), Link, Abhängigkeitsbeziehungen (*„Dependency“*). Java-EE-Anwendungen sind während der Ausführung in einem Applicationsserver wie z. B. Glasfish in standardisierte Archive verpackt. Die WAR-Dateien sind für den Web-Teil von Java-EE-Anwendungen wie z. B. JavaServer Pages (JSP) oder Servlets zusammengefasst. Web-Komponenten verwalten die Ausführung von Web Pages, Servelets oder einigen EJB für Java-EE-Applikation.

Literatur

1. Taentzer, G.: Vorlesung für UML in: Softwaretechnik, Universität Marburg (2015)
2. Rumpe, B.: Agile Modellierung mit UML. Springer Verlag (2012)
3. Fowler, M.: UML konzentriert, Addison Wesley Verlag (2004)
4. Virtual University: Wikipedia, MediaWiki, Web Portal, www.v-u.ch (2017)
5. Böhmer, E.: Elemente der angewandten Elektronik, Vieweg Verlag (1994)
6. Informatik Labor: Web Portal, Fachbereich Informatik, Uni. Darmstadt, https://www.fbi.h-da.de/labore/case/uml/kompositionsstrukturdiagramm.html (2010)
7. SMA Web Portal: Wechselrichter in: Leistungselektronik für eine saubere Energieversorgung, Web Portal, SMA Solar Technology, https://www.sma.de/partner/expertenwissen/wechselrichter-leistungselektronik-fuer-eine-saubere-energieversorgung.html (2017)
8. Hackstein, D.: Lehrmaterial für die Elektrische Energietechnik, Fakultät für Mathematik und Informatik, Fernuni in Hagen (2012)
9. RapidClipse Dokumentation, Web Portal, XDEV Software Corp, San Francisco, USA; https://rapidclipse.atlassian.net/wiki/spaces/DOK/pages/43614250/Overview (2019)
10. Lappe, R. et al.: Handbuch Leistungselektronik: Grundlagen, Stromversorgung und Antriebe, Verlag Technik GmbH, 5. Auflage (1994)
11. Matevska, J.: Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit, Vieweg+Teubner Verlag, 1. Auflage (2010)
12. Oracle Web Portal: Distributed Multitiered Applications in:The Java EE Tutorial, Web Portal, Java Platform, Enterprise Edition (Java EE) 8, https://javaee.github.io/tutorial/overview004.html, Oracle (2019)
13. Koschel, A., Fischer, S., Wagner, G.: J2EE/Java EE kompakt, 2. Auflage, Spektrum Akademischer Verlag (2006)
14. Röwenkamp, L., Limburg, A. und Kölpin, S.: Java EE Tutorial-Teil 1: Dreigestirn aus JSF, CDI und JPA In: *Java 2017*, Zeitschrift IX Developer, Sommer (2017)
15. Cowen, L.: JPA 2.2 and JSF 2.3 updates in the october liberty beta, IBM Developer, Blog, https://developer.ibm.com/wasdev/blog/2017/10/26/jpa-2-2-jsf-2-3-oct-liberty-beta/#jsf23 (2017)

Eclipse-Papyrus-Framework

3

Eclipse-Papyrus stellt ein Open-Source-Plugin für die Eclipse IDE [1] der Eclipse Foundation zur Modellierung und Unterstützung modellgetriebener Entwicklungsansätze dar. Papyrus nutzt als Basis das Eclipse-Modeling-Framework (https://eclipse.org/modeling/emf/). Neben UML und SysML werden auch andere Modellierungssprachen wie z. B. BMM unterstützt.

Papyrus stellt mit UML 2 als Familie zur Modellierung zwei Arten von Diagrammen dar: 6 Diagramme zur Strukturmodellierung und 7 Diagramme zur Verhaltensmodellierung. Die erste Gruppe enthält folgende Diagramme: Klassendiagramm, Komponentendiagramm, Objektdiagramm, Package-Diagramm, Verteilungsdiagramm und Profile-Diagramm. Die zweite Gruppe verfügt über folgende Diagramme: Zustandsdiagramm, Aktivitätsdiagramm, Use Case-Diagramm, Interaktionsdiagramme (Sequenzdiagramm, Kommunikationsdiagramm, Interaktion Überblicksdiagramm, Timing Diagramm).

3.1 Erstellung eines UML-Klassendiagramms

Der Diagrammtyp „Klassendiagramm“ dient dazu, eine oder mehrere Klassen, unabhängig von der Sprache in der sie implementiert wurden/werden sollen, abzubilden. Es kann/können sowohl die Klasse(n) selbst als auch ihre Beziehungen zu anderen Klassen dargestellt werden. Je nach Menge der Codezeilen kann der Quellcode einer bzw. mehrerer Klassen sehr unübersichtlich sein. Die Idee hinter diesem Diagrammtyp ist, eine übersichtliche Form zu finden, die, unabhängig von Kenntnissen über Details der Programmiersprache oder des Programms, zu verstehen ist. Klassendiagramme können sowohl zur Beschreibung eines schon in Code umgesetzten Programms als auch zur Modellierung eines Sachverhaltes vor der konkreten Umsetzung in einer Programmiersprache verwendet werden. Klassendiagramme zählen zu den UML-Strukturdiagrammen [2].

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML,*
https://doi.org/10.1007/978-3-658-29057-3_3

Das Erstellen eines neuen Papyrus Project erfolgt nach dem Installieren und Starten des Frameworks, wobei die Papyrus-Perspective geöffnet ist. Abb. 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7 und 3.8 geben Überblicke über die Erstellung eines UML-Klassendiagrammms mit dem Eclipse Framework Papyrus. Mit einem Rechtsklick auf den *„Project Explorer -> New -> Papyrus Project“* zeigen Abb. 3.1 und 3.2 wie das Projekt erstellt ist. Wie auf der Abb. 3.3 zu sehen ist, wird im Wizard zunächst *UML* als *Software Engineering* für *Architecture Contexts* ausgewählt. Anschließend zeigt Abb. 3.4 die Angabe über die Namen des neuen Projektes und der Modell-Datei. Abb. 3.5 und 3.6 zeigen verschiedene Diagrammtypen der UML-Architektur vom Framework Papyrus. Mit einem Hacken auf den Kasten des Diagrammtyps ist eine Auswahl zu realisieren. Das Erstellen eines Klassendiagramms ist mithilfe des Auswählens von *„Class Diagram“* ermöglicht. Nach dem Erstellen des Projektes *„stromversorgung“* ist in dem Model Explorer die Struktur des Diagrammtyps zu sehen, wobei der Name des Diagramms erkennbar ist. Abb. 3.7 verfügt über *„ClassDiagram-Element“* wie z. B. *„Package Windkraft“*, das mithilfe der Diagramm-Palette erstellt wird. Abb. 3.8 zeigt die verschiedenen Elemente des erstellten UML-Klassendiagramms. Hierbei sind in dem Package *„Oberfläche-Schicht“*, *„Datenschicht“* und *„Tabellenschicht“* Klassen mit ihren Attributen und Operationen ähnlich wie mit dem Obeo-UML-Designer zu sehen. User Guide des Frameworks Eclipse-Papyrus gibt Information über Tutorial zum Erstellen von Diagrammtypen mit UML-Architektur *[Eclipse Foundation, Papyrus User Guide, Papyrus User Guide, Retrieved from]*.

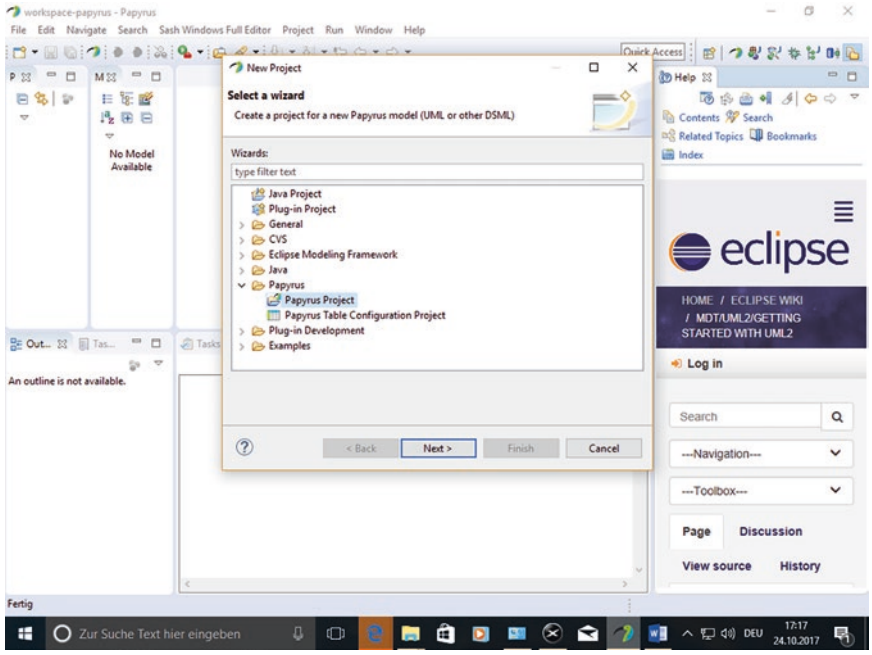


Abb. 3.1 Auswahl von Wizard zum Erstellen eines Papyrus-Projektes

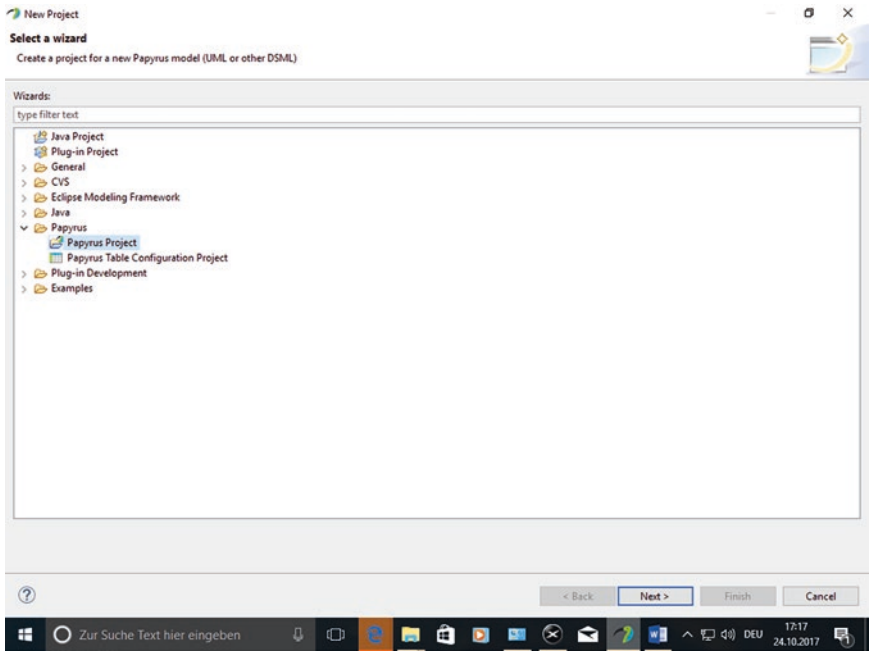


Abb. 3.2 Überblick über Papyrus-Projekt

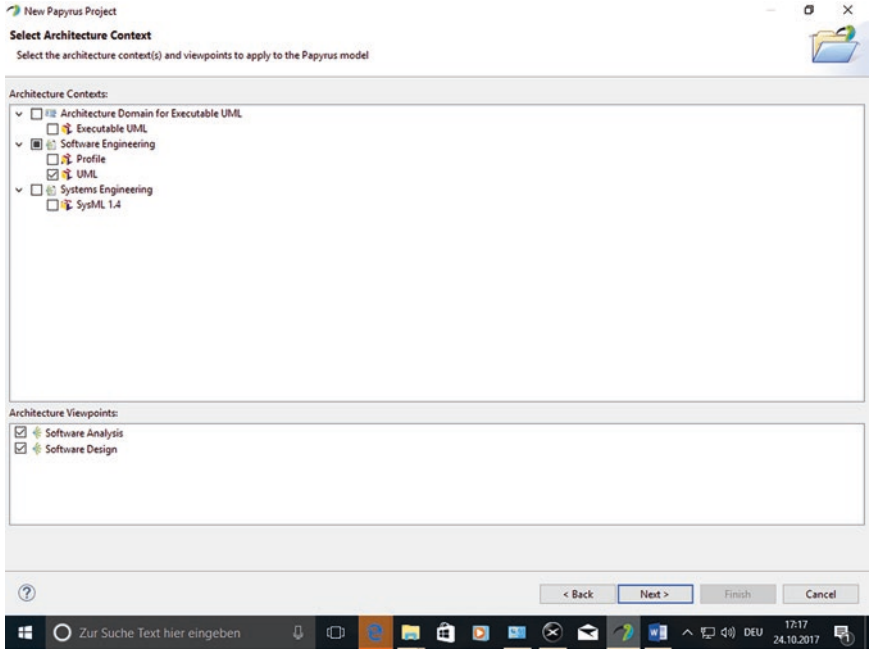


Abb. 3.3 Auswahl der UML-Architektur

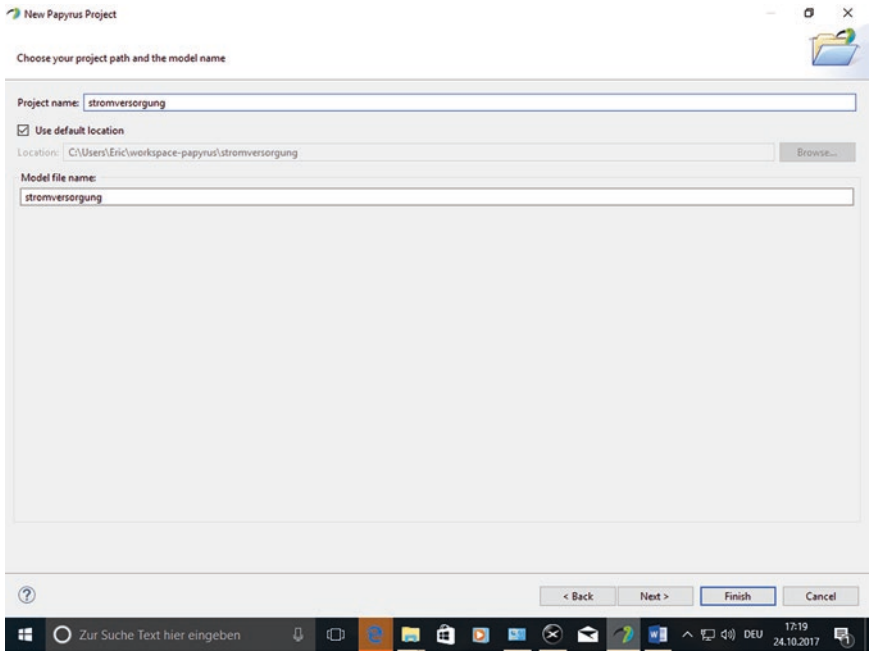


Abb. 3.4 Überblick über Projekt- und Modell-Datei-Name

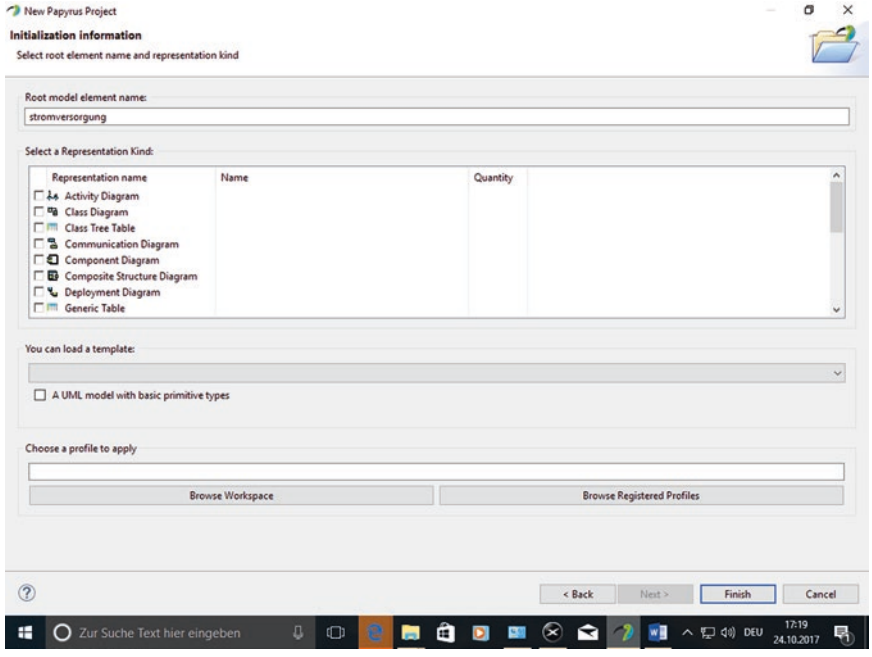


Abb. 3.5 Überblick über verschiedene Diagrammtype

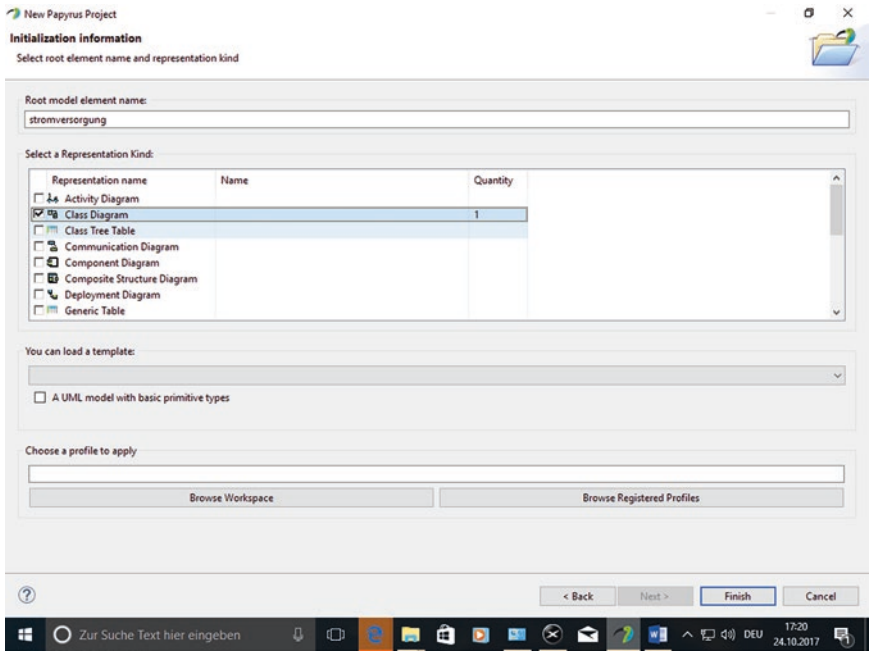


Abb. 3.6 Auswahl des Klassendiagramms

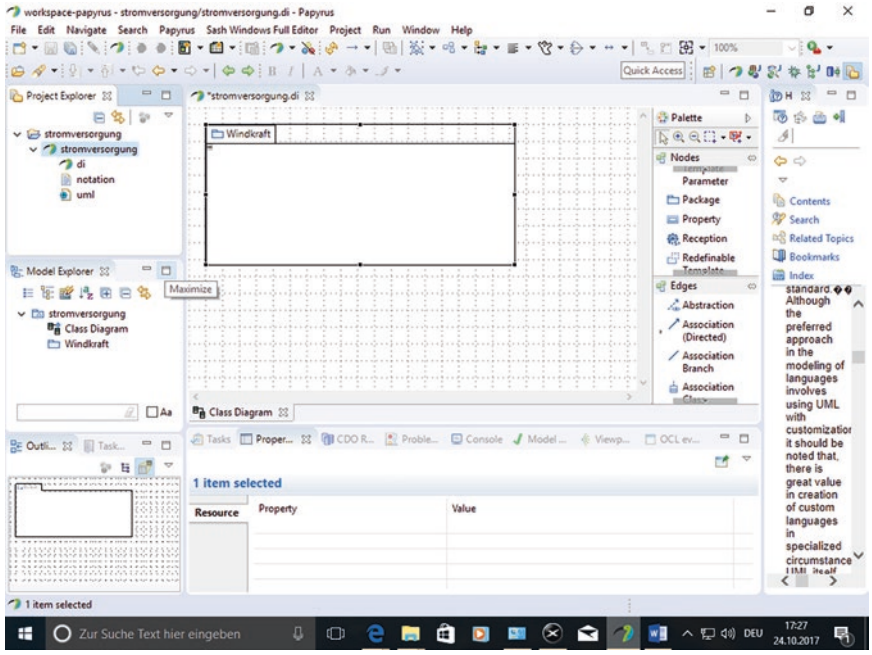


Abb. 3.7 Darstellung der Klassendiagramm-Elemente mithilfe von Model Explorer und Diagramm-Palette

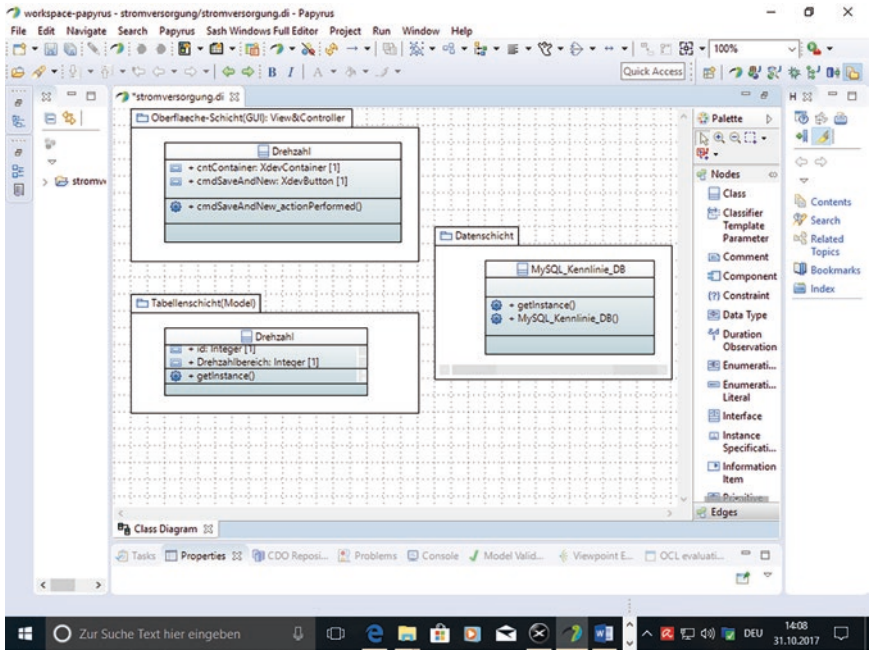


Abb. 3.8 Struktur des Klassendiagramms mit Package, Klassen, Attributen und Operationen

3.1.1 Struktur des UML-Klassendiagramms

Nach dem Erstellen der Klasse genannt *resonanzfrequenz* wurden Merkmale wie z. B. Attribute hinzugefügt. Abb. 3.9, 3.10, 3.11, 3.12, 3.13, 3.14 und 3.15 zeigen die Schritte vom Erstellen der Klasse bis zum Erstellen von Attributen. Mit einem Rechtsklick auf die erstellte Klasse *resonanzfrequenz* im Model Explorer und das Auswählen des gesuchten Attributes, genannt Property, mithilfe von *„New Child -> Property“* wurden die Properties der Attribute *induktivitaet* und *kapazitaet* erstellt. Vor dem Property-Namen steht ein Doppelrechteck mit einem Plus-Symbol: Dies bedeutet, dass diese Attribute *„public“* definiert wurden. Im Container der Klassenstruktur ist ein hierarchisches Design mit den Namen *„frequenz“* und *„resonanzfrequenz“* zu sehen. Der erste Name ist derjenige des Models und der zweite ist derjenige der Klasse. Abb. 3.12 zeigt den Inhalt der Modellierungswerkzeugkasten, genannt Palette, in Bezug auf *„Nodes“* und *„Edges“*, die über Elemente wie z. B. *„Class“*, *„Property“* oder *„Attribute“* bzw. Beziehungen wie z. B. *„Dependency“*, *„Assoziation“* oder *„Link“* verfügen. Abb. 3.12 und 3.13 stellt einen Wizard zum Auswählen eines neuen Vererbungselements für das neue Objekt im Container *„frequenz“* dar. Mit Abb. 3.12 und 3.13 ist die Struktur des Containers *„Frequenz“* und seine Klasse *„resonanzfrequenz“* zu sehen.

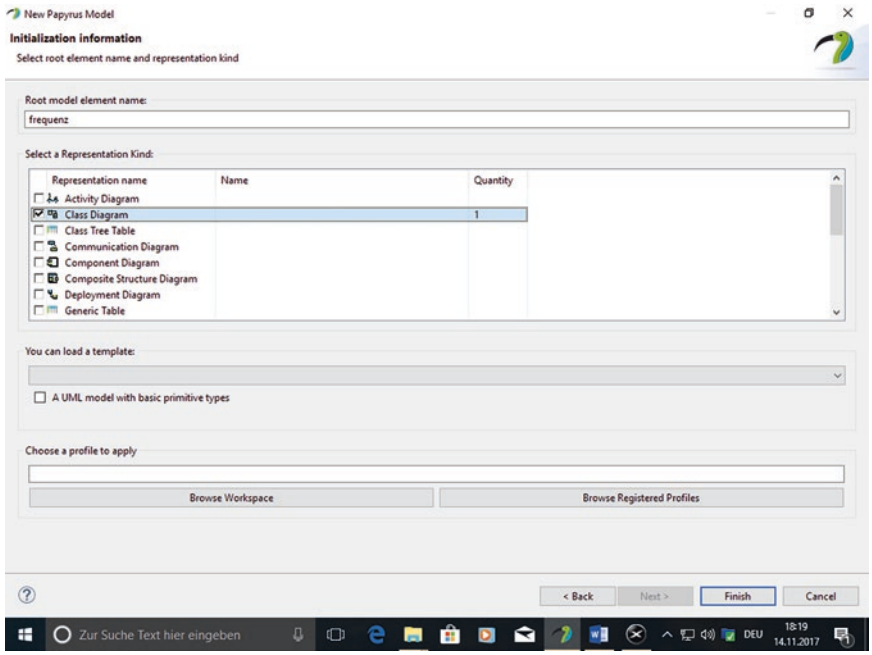


Abb. 3.9 Wizard zum Auswählen eines Diagrammtyps

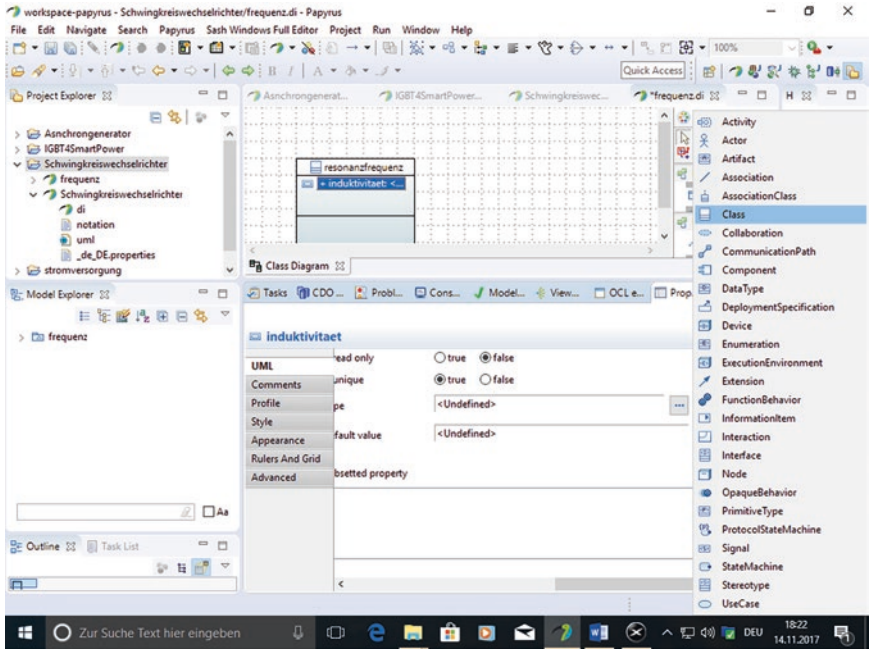


Abb. 3.10 Überblick über die Struktur der Klasse *„resonanzfrequenz“*

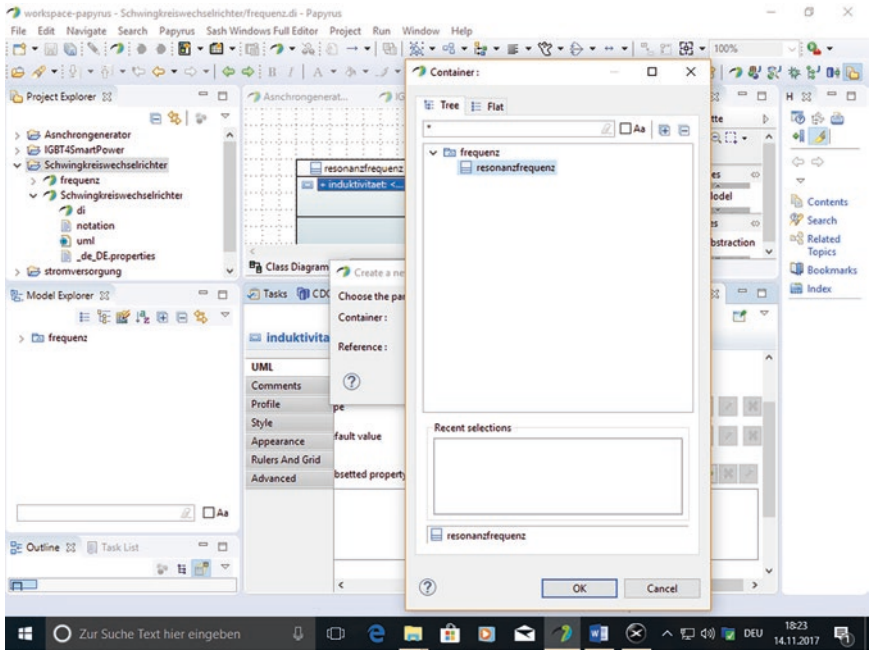


Abb. 3.11 Hierarchische Struktur des Containers *„frequenz“*

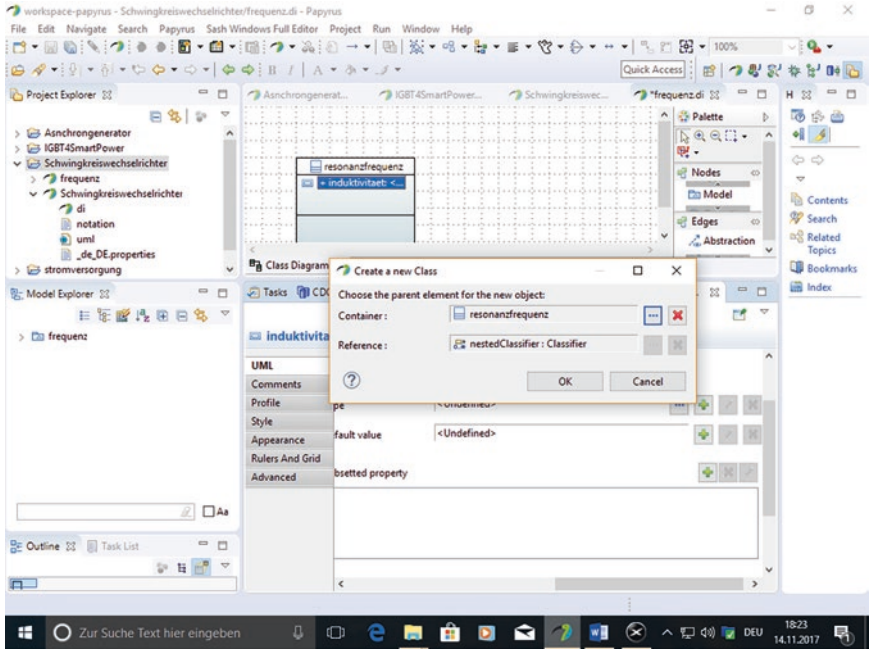


Abb. 3.12 Überblick über das Erstellen des Attributs *„induktivitaet“* im Container *„resonanzfrequenz“*

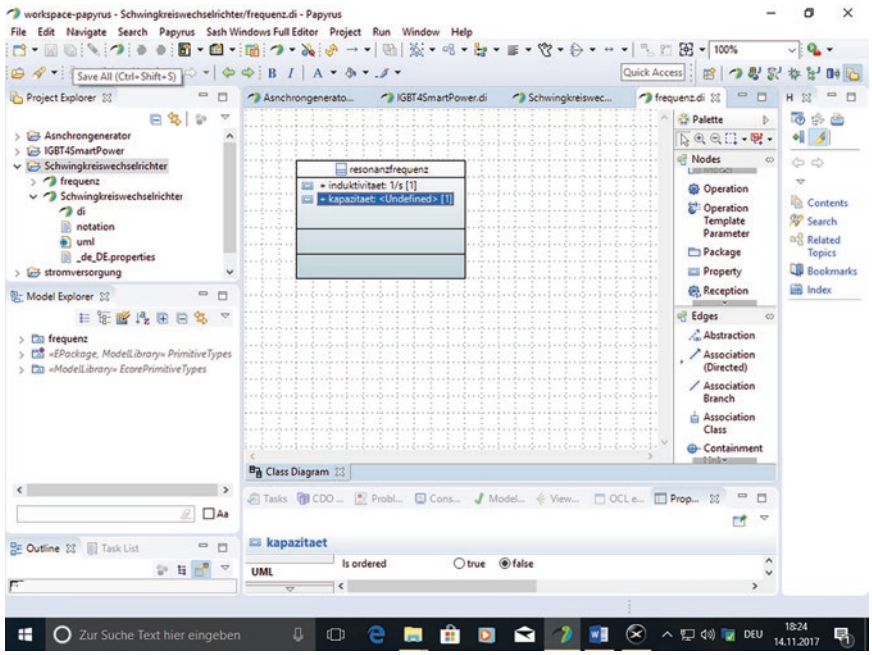


Abb. 3.13 Inhalt der Modellierungswerkzeugkasten, genannt Palette, in Bezug auf *„Nodes“* und *„Edges“*

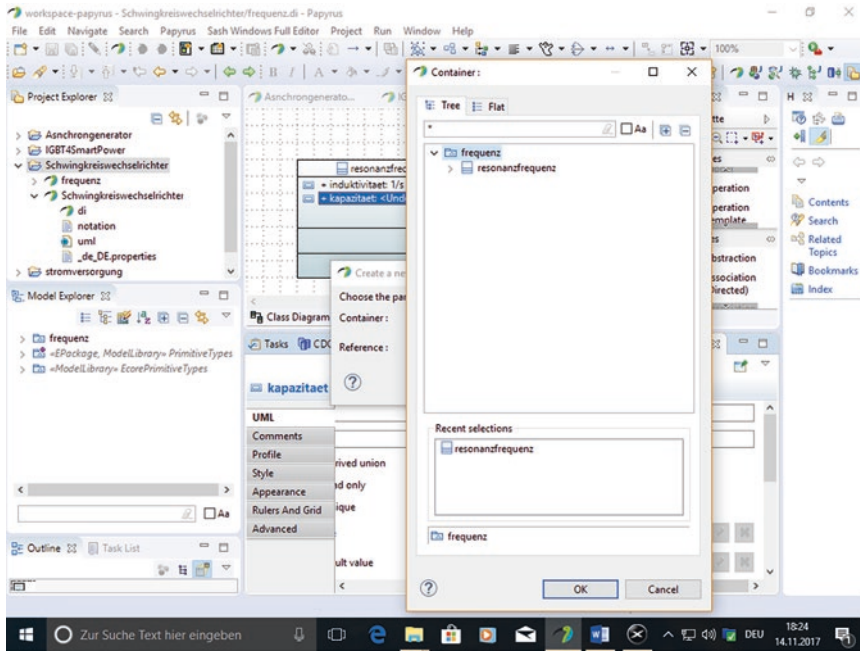


Abb. 3.14 Auswahl eines Elements im Container

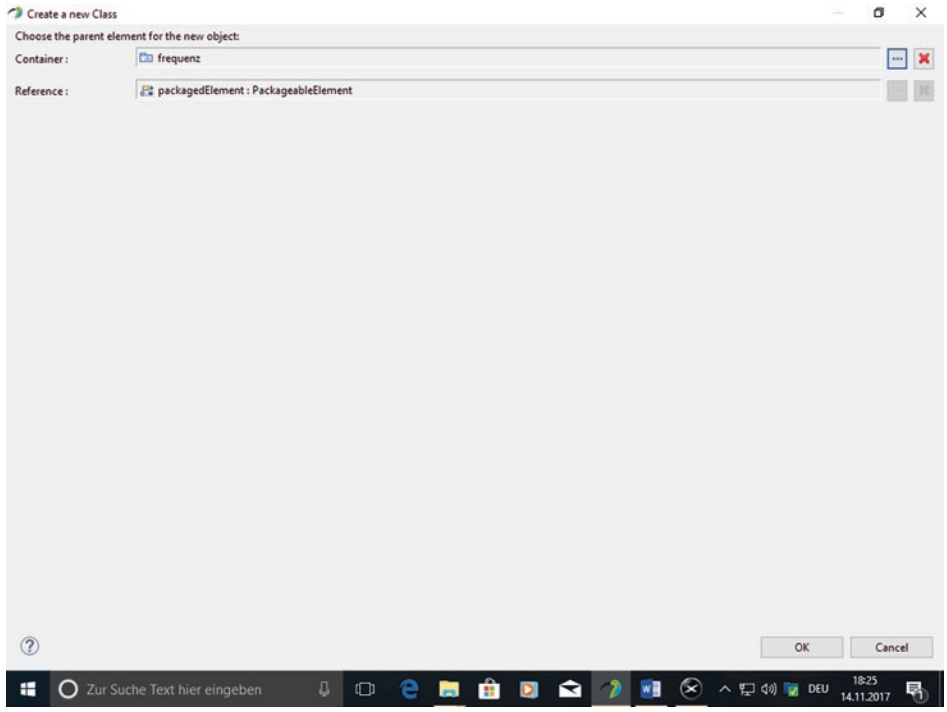


Abb. 3.15 Überblick über Container und seine Referenz

3.1.2 Beispiel: Inneres Klassendiagramm

Da Klassendiagramme den Baustein jeder objektorientierten Struktur darstellen, beruht die Analyse des Klassendiagramms auf Merkmalen der Klassen und deren Beziehungen zwischen ihnen. Eclipse-Papyrus verfügt über ein inneres Klassendiagramm, welches die konkrete Struktur zur Modellierung darstellt. Dieses Beispiel fokussiert auf Modellierung der Last-Elemente im Schwingkreiswechselrichter in Bezug auf die Beziehungen zwischen verschiedenen Spannungselemente. Hierbei ist anzumerken, dass die Resonanzelemente Induktivität und Kapazität eine wichtige Rolle zur Reduzierung der Verluste bei Schwingkreiswechselrichter, genannt Resonanzwechselrichter, spielen. Mithilfe der objektorientierten Modellierung ist das Analysieren der Funktionalität des Schwingkreiswechselrichters einfach.

Abb. 3.16 und 3.17 zeigt das Erstellen des Klassendiagramms mit dem Framework Eclipse-Papyrus.

Gemäß Abb. 3.19 und 3.20 sind Klassen als Rechteck mit Klassennamen dargestellt. Normalerweise bestehen Klassen aus drei Bereichen: oberer, mittlerer und unterer Bereich. Der Erste verfügt über den Stereotyp, genannt Paket, zu dem die

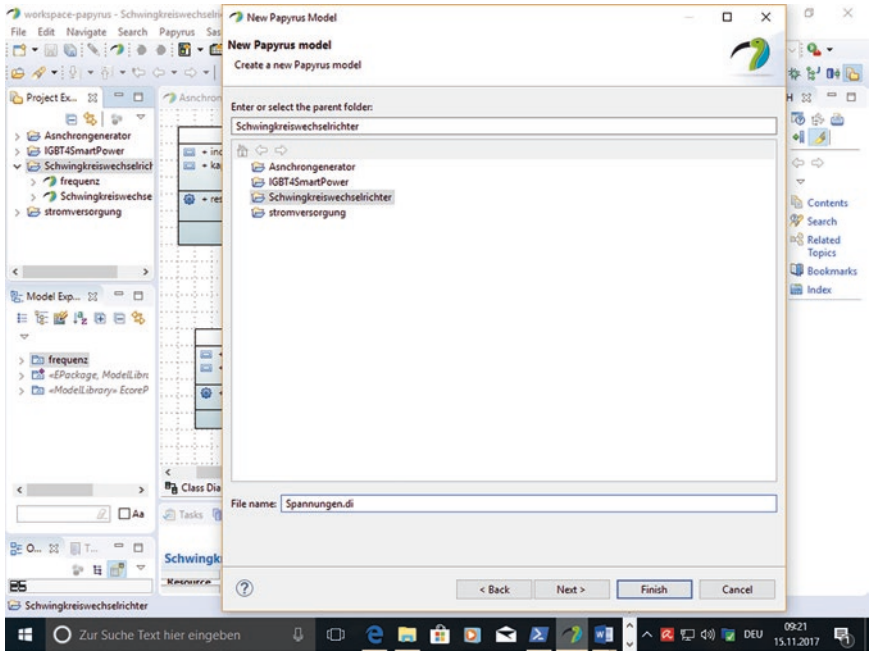


Abb. 3.16 Erstellen eines neuen Papyrus-Modells im Projekt *„Schwingkreiswechselrichter“*

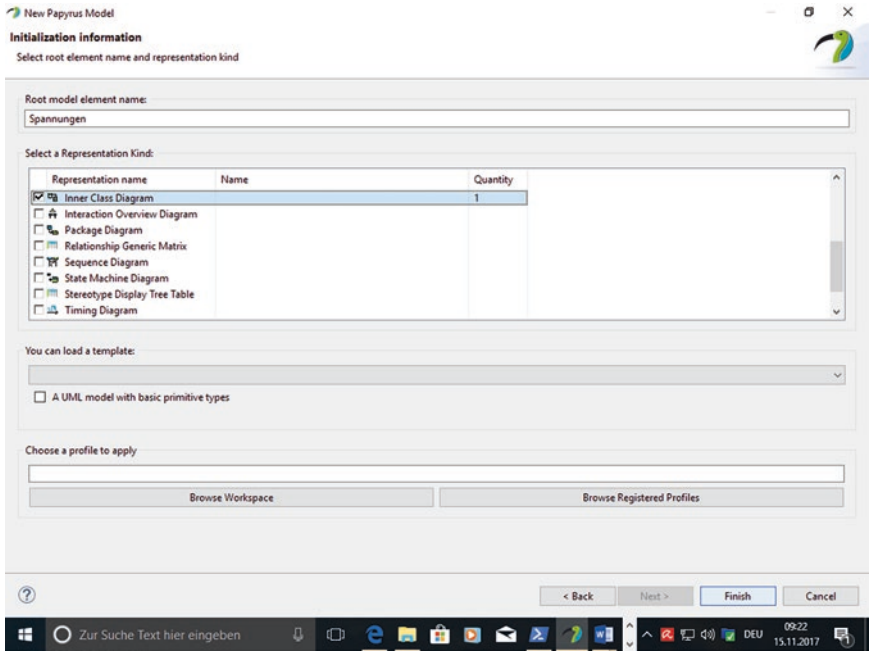


Abb. 3.17 Auswählen des Diagrammtyps, genannt Inner Class Diagram, im Modell *„Spannungen“*

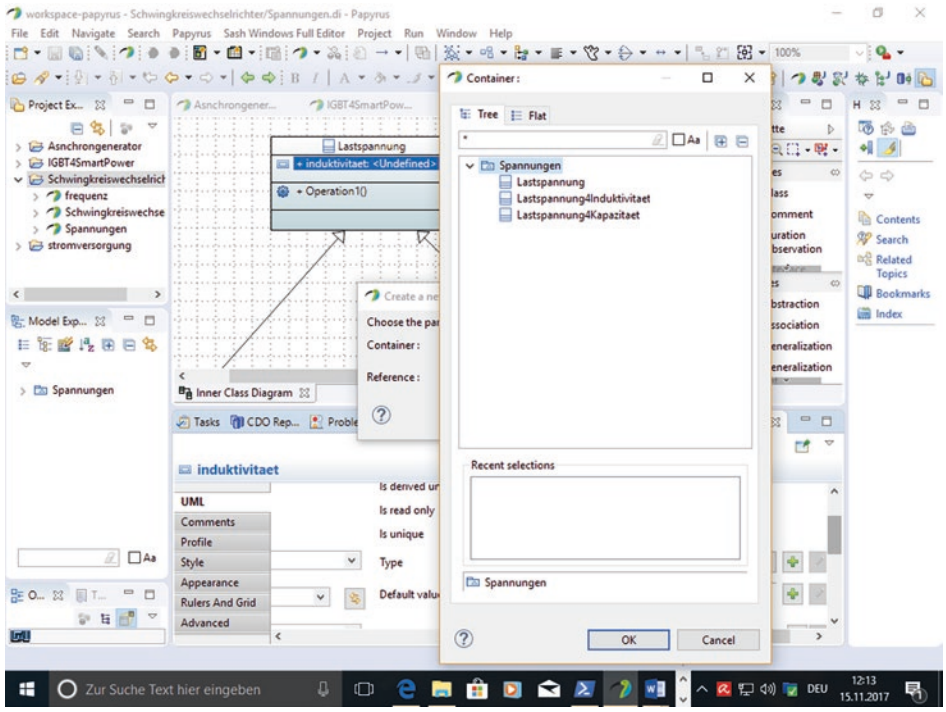


Abb. 3.18 Überblick über den Container „*Spannungen*“ mit seiner Referenz

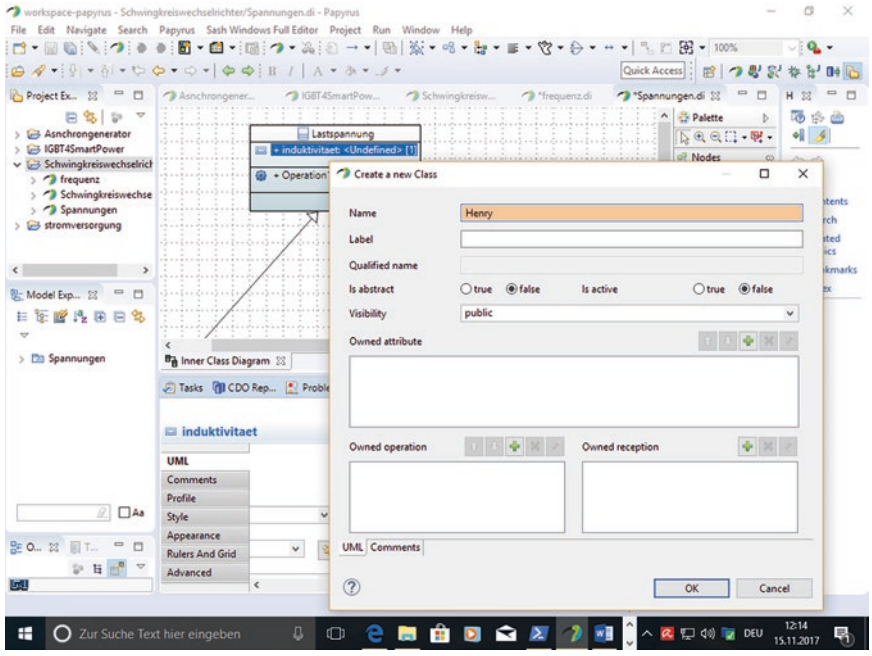


Abb. 3.19 Inneres Klassendiagramm mit Klassen und deren Beziehungen

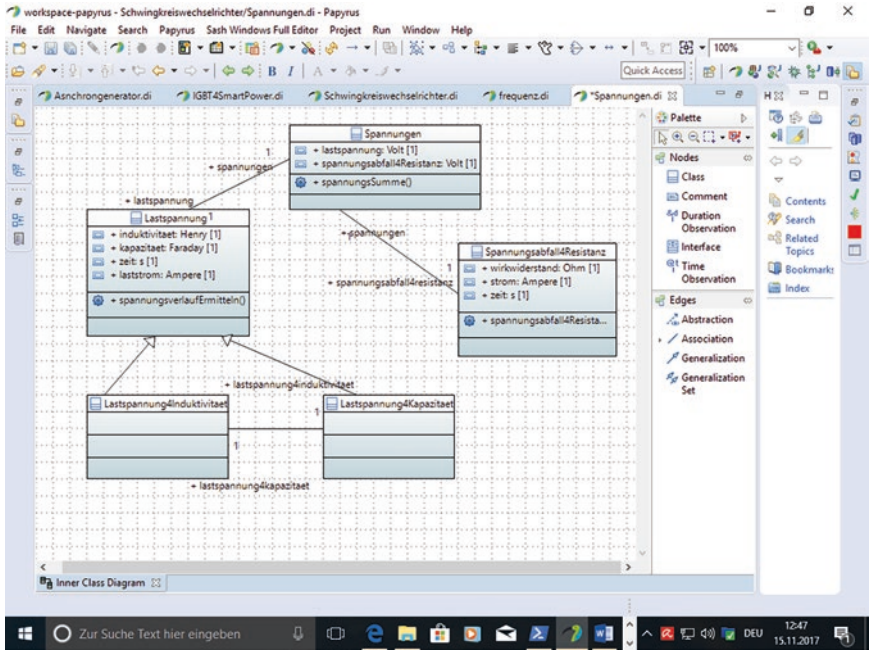


Abb. 3.20 Überblick über Assoziationen

Klasse gehört und den Namen. Im Zweiten sind die Attribute, genannt *„Property“*, angegeben und im Unteren gibt es die Methoden, genannt Operationen, der Klasse. Laut UML-Spezifikation kann die Darstellung einer Klasse zusätzliche Bereiche enthalten [3].

Mit der Abb. 3.16 ist das Erstellen des Modells *„Spannungen“* für das Projekt, genannt *„Schwingkreiswechselrichter“,* zu sehen. Anschließend zeigt Abb. 3.17 das Auswählen des Diagrammtyps, genannt *Inner Class Diagram,* im Modell *„Spannungen“*. Mit der Abb. 3.18 ist der Container *„Spannungen“* mit seiner Referenz zu sehen. Klassen und deren Beziehungen zeigen Abb. 3.19 und 3.20 in Bezug auf objektorientierte Modellierungen.

3.1.2.1 Überblick über Assoziationen

Eine Assoziation stellt eine Beziehung mittels Linien zwischen Klassen dar. Eine Linie zwischen den Klassen stellt eine Assoziation dar. Wie auf Abb. 3.19 und 3.20 zu sehen ist, kommunizieren die Objekte der Klassen über die Assoziationen miteinander. Hierbei verfügen die Assoziation zwischen den Klassen *„Lastspannung“* und *„Spannungen“* über Namen-Rolle mit Plus-Symbol an jedem Enden der Verbindung, Kardinalität und Richtung wie z. B. *„+lastspannung“* und *„+spannungen“* haben. Die Zahl 1 an dem Ende des Assoziationsnamens gibt die Leserichtung des Namens an, wobei an den Assoziationsenden die Rollen der beteiligten Klassen und die Multiplizität gezeigt sind.

3.1.2.2 Überblick über Generalisierung

Die Generalisierung zeigt eine Vererbung zwischen Klassen. Mit Abb. 3.19 und 3.20 ist eine hierarchische Vererbung zwischen der Mutterklasse, genannt *„Lastspannung“*, und den Tochterklassen *„Lastspannung4Induktivitaet“* und *„Lastspannung4Kapazitaet“* mittels Generalisierung dargestellt. Hierbei ist die Vererbung zum Veranschaulichen von Beziehungen zwischen einer Ober- und Unterklasse und sind Attribute und Operationen der Oberklasse in den Unterklassen zugänglich gemacht. Die Vererbung ist das klassische Programmierkonzept, welches in den Objektorientierten Programmiersprachen verwendet wird [4]. Abb. 3.19 und 3.20 stellen die Vererbungsbeziehungen mittels eines Pfeils dar. Hierbei zeigt die Pfeilspitze auf die Mutterklasse *„Lastspannung, welche ihre Eigenschaften“* an die Tochterklassen *„Lastspannung4Induktivitaet“* und *„Lastspannung4Kapazitaet“* vererbt.

3.1.2.3 *„Vererbungskaskade“*

Eine Reaktionskette der Vererbung mittels hierarchischer Vererbung stellt den Begriff *„Vererbungskaskade“* dar. Hierbei sind mehrere Vererbungsschichten im Klassendiagrammen zu sehen. Abb. 3.21 zeigt, wie diese Kette der Vererbung aussieht. Abb. 3.21 gibt einen Überblick über die leistungselektronische Funktionalität der Leistungshalb-

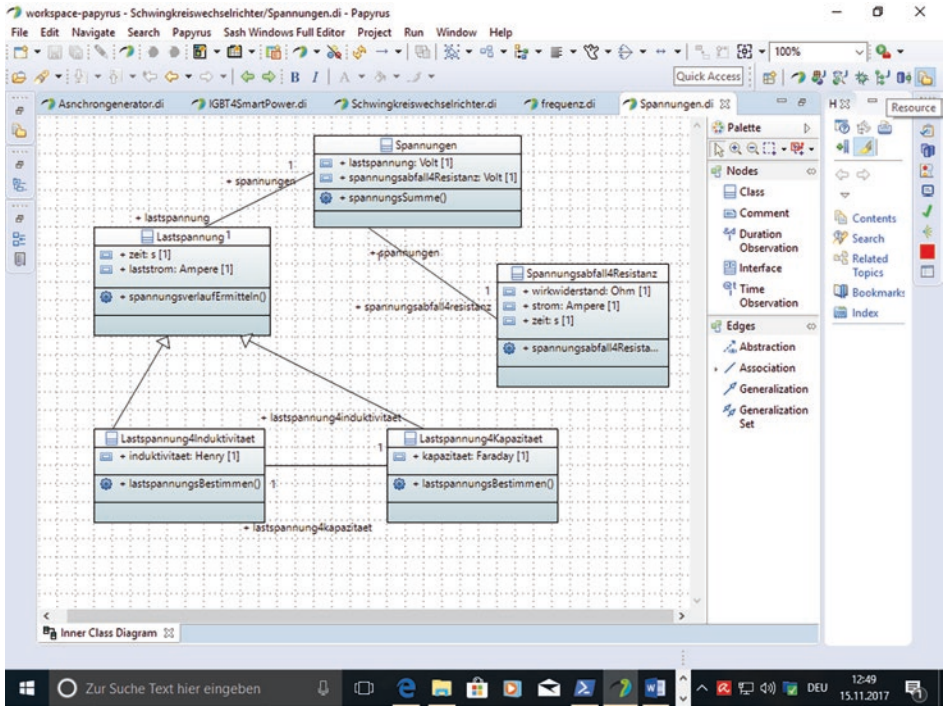


Abb. 3.21 Vererbungsschichten in Klassendiagrammen

leiter. Hierbei sind sowohl Ober- als auch Unterklassen während dieser Vererbungskaskade dargestellt. Mit Abb. 3.21 verfügt diese Vererbungskaskade drei Schichte: Zum einen stellt die Erste eine Vererbung zwischen den Tochterklassen *„AbschaltbareLeistungshalbleiter“* und *„NichtabschaltbareLeistungshalbleiter“,* genannt *„Leistungshalbleiter“,* dar. Zum anderen sind *„AbschaltbareLeistungshalbleiter“* und *„NichtabschaltbareLeistungshalbleiter“* (oder *„Leistungshalbleiter“*) Mutterklassen der Tochterklassen *Transistoren, GTO* und *IGTC* bzw. *Thyristoren* und *Dioden* aus der zweiten Schicht. Die dritte Schicht stellt die Vererbung zwischen der Mutterklasse *„Transistoren“* und den Tochterklassen *„LTR“*, *„MOSFET“* und *„IGBT“* dar. Die letzte Schichte verfügt über Klassen mit sichtbaren Merkmalen, d. h. *Property* und *Operation.* Diese Tochterklassen definieren gemeinsame Attribute und Methoden (Operationen) wie z. B. *„einschaltzeit“* und *„ausschaltzeit“* einerseits und andererseits *„einschaltzeitBerechnen“, „ausschaltzeitBerechnen“* und *„verlustleistungsBerechnen“*. Wie in dem Abschn. 2.3.1.2 zeigen die Pfeilspitze auf die Mutterklassen. In dieser Vererbungskaskade ist die Generalisierung mittels Pfeilspitzen dargestellt. Die Analyse der Abb. 3.21 fokussiert auf die Tatsache, dass LTR, MOSFET und IGBT zu den Transistoren, abschaltbaren Leistungshalbleitern und Leistungshalbleitern gehören. GTO und IGTC sind sowohl abschaltbare Leistungshalbleitern als auch Leistungshalbleitern, aber Thyristoren und Dioden sind keine abschaltbaren Leistungshalbleiter und nur Leistungshalbleiter. Mithilfe der objektorientierten Modellierung ist die Analyse dieser Vererbungskaskade einfach.

3.2 Paketdiagramm

Das Paketdiagramm zum Beschreiben des Strukturdiagramms mit Framework Eclipse-Papyrus stellt die Zusammenfassung von Modellelementen dar. Hierbei werden Elemente wie z. B. Klassen strukturiert modelliert, damit man einen Überblick über die Komponente der Software oder des Systems hat. Da Pakete Modellelemente zu Gruppen zusammenfassen, können sie für verschiedene Zwecke eingesetzt werden. Pakete können verwendet werden, um Subsysteme zu modellieren, in die ein großes System zerlegt wird. Pakete können auch verwendet werden, um die Ergebnisse von Entwicklungsphasen festzulegen und die entsprechenden Diagramme, die einer Phase zugeordnet werden, zusammenzufassen. Darüber hinaus können Pakete für jede sinnvolle Gruppierung von Modellelementen in einem Projekt eingesetzt werden. Die Systemarchitektur kann z. B. mit Paketen abgebildet werden [3]. Abb. 3.22, 3.23, 3.24, 3.25, 3.26, 3.27, 3.28, 3.29, 3.30, 3.31, 3.32, 3.33 und 3.34 stellt die Entwicklung eines Paketdiagramms vom Projekt-Erstellen bis zum Modellieren dar. Abb. 3.22 zeigt den Wizard zum Erstellen eines Projektes, genannt *„Asynchrongenerator“,* anschließend zeigt Abb. 3.23 das Auswählen

eines Paketdiagramms oder *„Package Diagram“* im Modell *„Asynchrongenerator“*. Den Container, *genannt „Asynchrongenerator“,* mit seiner Referenz zeigt Abb. 3.24. Das Paketdiagramm bezüglich des Design-Patterns *„Model View Controller“* (MVC) stellen Abb. 3.25, 3.26, 3.27, 3.28, 3.29, 3.30 und 3.31 dar. Das Paketdiagramm besteht aus drei Paketen: *„Fenster-Schicht“* (GUI), *„VirtuellenTabellen“* und *„Datenquellen“* (Datenbank MySQL). Jede Klasse verfügt über *Attribute* und *Operation.* Abb. 3.32 verdeutlich die Syntax der Klassen im Paketdiagramm mithilfe der Kommentare. Abb. 3.33 und 3.34 stellt die Anwendung der Modellierung in der Programmierung bezüglich des Konzeptes *„Code in Model“*, d. h., Mithilfe der Programmierung ist die Modellierungsstruktur sowohl zum Analysieren als auch zum Beschreiben.

3.2.1 Paketdiagramm mit dem Design Pattern Model View Controller

Model View Controller ist ein Design Pattern zum Darstellen von Anwendungen in getrennten Klassen.

Hierbei ist das Paketdiagramm eine Darstellung des Design Patterns MVC, welches die strickte Teilung von Präsentation (View), der Programmlogik (Controller) und der Datenschicht (Model) ermöglicht. Dies vereinfacht die Lesbarkeit und Wartbarkeit des Codes. Mithilfe des Design Patterns MVC kann man ein Projekt viel einfacher um weitere Funktionen ergänzen, ohne den halben Sourcecode durchzuschauen [5]. Abb. 3.33 und 3.34 zeigt die Struktur einer MVC-Anwendung mit dem Java-Framework XDEV4. Das Designen der Oberflächen mit Java XDEV 5 basiert auf Java Swing, das seit 1997 als Standard-Bibliothek für die Entwicklung grafischer Oberflächen in Java gilt und auch heute noch mit gutem Grund von vielen Java-Entwicklern für die Anwendungsentwicklung eingesetzt wird. Swing ist beliebt, ausgereift und stark. In XDEV 5 stehen den Entwicklern viele GUI-Controls zur Verfügung, die Swing bietet, u. a. Buttons, Formular-Komponenten, Fenster und Dialoge, Tabs, Menüleisten und Kontextmenüs sowie eine leistungsfähige Tabellen- und Tree-Komponente. Java Swing stellt einen Umfang an Controls für die Entwicklung grafischer Oberflächen (GUI = Graphical User Interface) zur Verfügung wie z. B. Buttons, Labels, Tabs, sämtliche Formular-Controls, Table, Tree sowie Fenstertechniken. Viele Standardanwendungen lassen sich damit umsetzen [6]. Abb. 3.33 und 3.34 zeigen die Struktur der verschiedenen Pakete bezüglich des Paketdiagramms von Eclipse-Papyrus. Hierbei sind in MVC-Komponenten verschiedene Java-Klassen zu sehen. Das Fenster-Paket vom Paketdiagramm entspricht

dem Paket *„Fenster“* vom dem Projektmanagement mit dem Java-Framework XDEV4 mit verschieden Klassen u. a. *„Drehzahlen“, „AufgenommeLeistungen“, „InneresMoment“*. Gemäßt Abb. 3.33 und 3.34 sind in den Paketen *„Datenquelle“, „Fenster“* und *„VirtuelleTabelle“*. Die drei Pakete des Java-Frameworks XDEV4 enthalten Java-Klassen. Abb. 3.33 zeigt die Klasse *„Drehzahlen“* mit einigen ihrer Methoden u. a. *cmdReset_ actionPerformed(), cmdNew_actionPerformed()* und this_windowClosing(). Die Klasse *„Drehzahlen“* vererbt Eigenschaften der Oberklasse, genannt *XdevWindow.* Abb. 3.33 gibt einen Überblick über den Aufbau der Tabelle *Drehzahl.*

3.2.2 Überblick über Java-Code in dem Modell

Java-Framework XDEV stellt sowohl eine visuelle Entwicklung als auch Java-Coding dar. Dafür verfügt das Framework XDEV über einen Java-Code-Editor mit Funktionen wie z. B. Code-Vervollständigung, Refactoring, Debugger und Versionsverwaltung. Bei XDEV setzt sich das Rapid-Application-Development-Konzept auch beim Code-Editor fort. Damit ist die Java-Programmierung mit dem Framework Java XDEV 5 einfach. Zum einen fokussieren Abb. 3.25, 3.26, 3.27, 3.28, 3.29, 3.30 und 3.31 auf das Java-Coding mithilfe der Modellierung und zum anderen verfügen sie über Struktur zur objektorientierten Modellierung bezüglich der Eigenschaften der Klassen wie z. B. Attribute und Operation. Abb. 3.25, 3.26, 3.27, 3.28, 3.29, 3.30, 3.31 und 3.32 zeigen wie die Struktur einer Java-Klasse aussieht, wobei es anzumerken ist, dass eine Modellierungsklasse, wie es auf den Abb. 3.25, 3.26, 3.27, 3.28, 3.29, 3.30, 3.31 und 3.32 zu sehen ist, verfügt gleiche Merkmale wie eine Java-Klasse. Abb. 3.33 zeigt die Strukturähnlichkeit zwischen Modellierungsklasse *„Drehzahl“* des Pakets *„Fenster“* für das Model View Controller und Java-Klasse *„Drehzahlen“*. Zum Beispiel zeigt die Java-Klasse *„Drehzahlen“* Codes in Bezug auf *Rapid Application Development.* Diese Codes entsprechen denjenigen der Modellierungsklasse. Abb. 3.25, 3.26, 3.27, 3.28, 3.29, 3.30, 3.31 und 3.32 geben Überblicke zum einen über das Hinzufügen der Attribute und Operation und zum anderen über das Modellierungswerkzeug, genannt *„Palette“,* welche über *„Nodes“*, *„Edges“* und *„Relationships“* verfügt. Von Abb. 3.25, 3.26, 3.27, 3.28, 3.29, 3.30, 3.31 und 3.32 bis zur Abb. 3.33 ist der Weg von der UML-Modellierung bis zur Java-Programmierung realisierbar. D. h., UML-Klassen enthalten Java-Klassen. Dieser Weg stellt das Konzept *„Codes in Models“* dar. Die Java-Codes verbergen sich im UML-Modell.

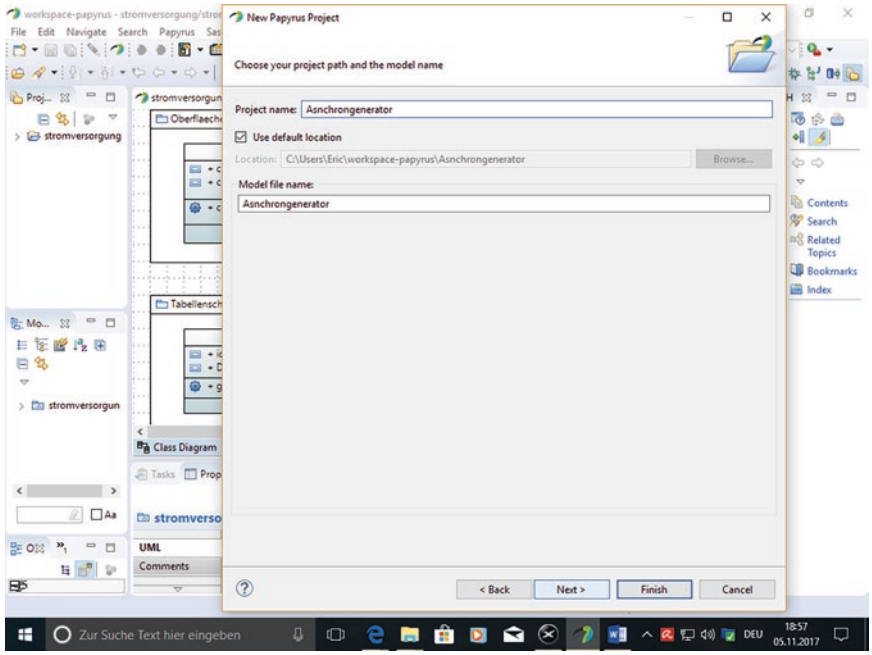


Abb. 3.22 Erstellen des Projektes *„Asnchrongenerator“* mit dem Framwork Eclipse-Papyrus

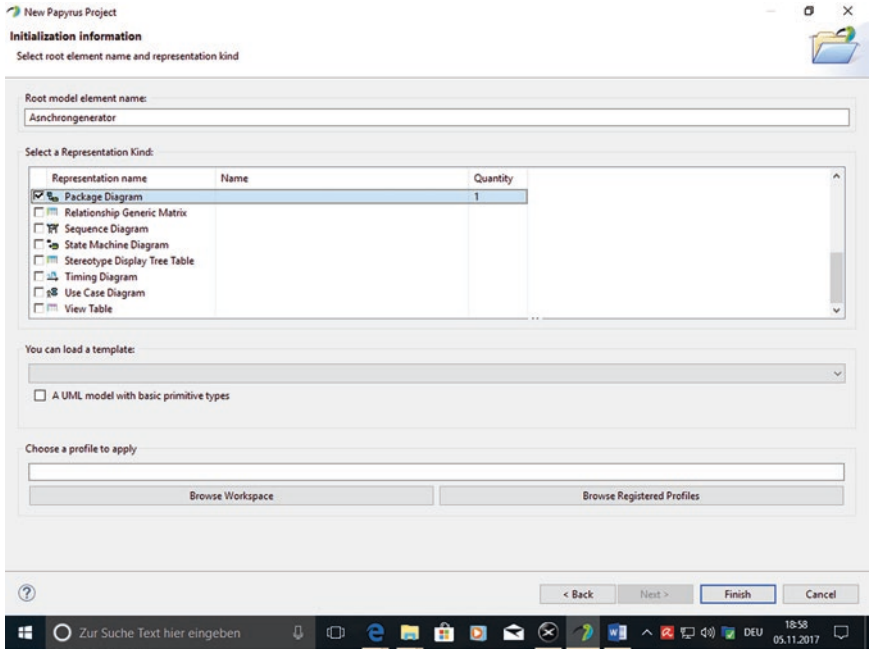


Abb. 3.23 Auswählen des Diagrammtyps *„Package Diagramm“* mit dem Framework UML-Papyrus

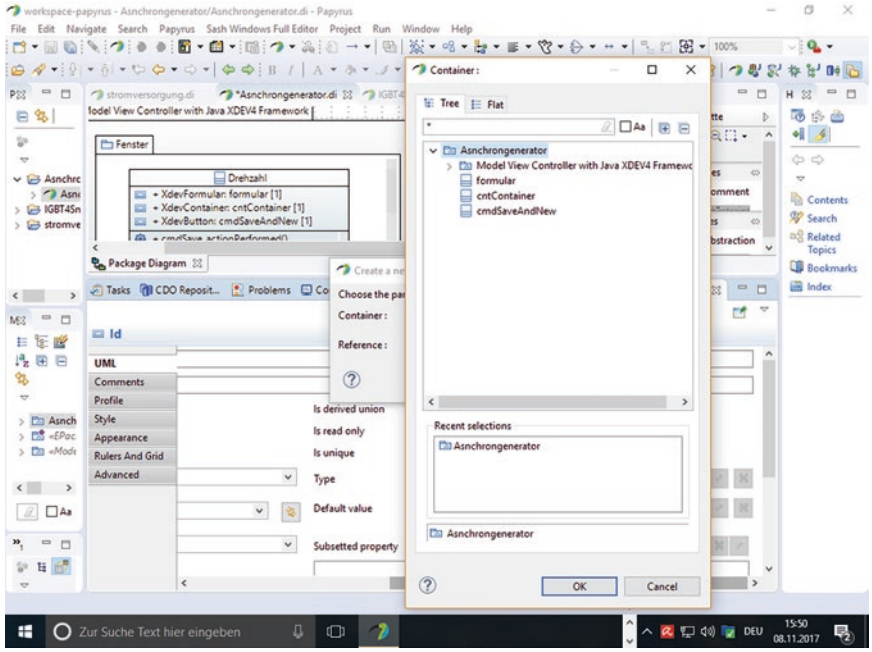


Abb. 3.24 Überblick über den Container *„Asynchrongenerator“*

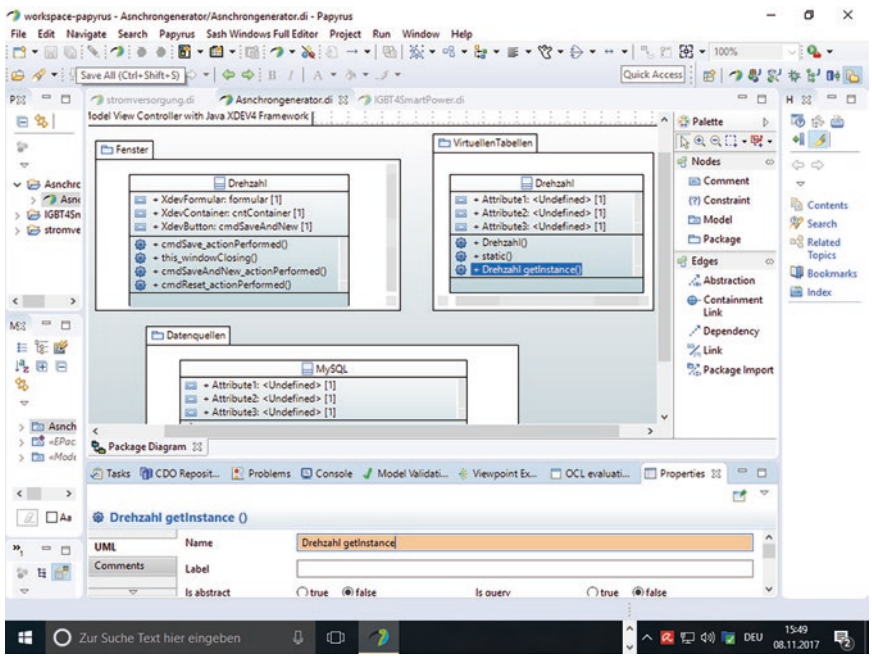


Abb. 3.25 Überblick über Paketdiagramm für Model View Controller mit dem Klassendiagramm und Palette

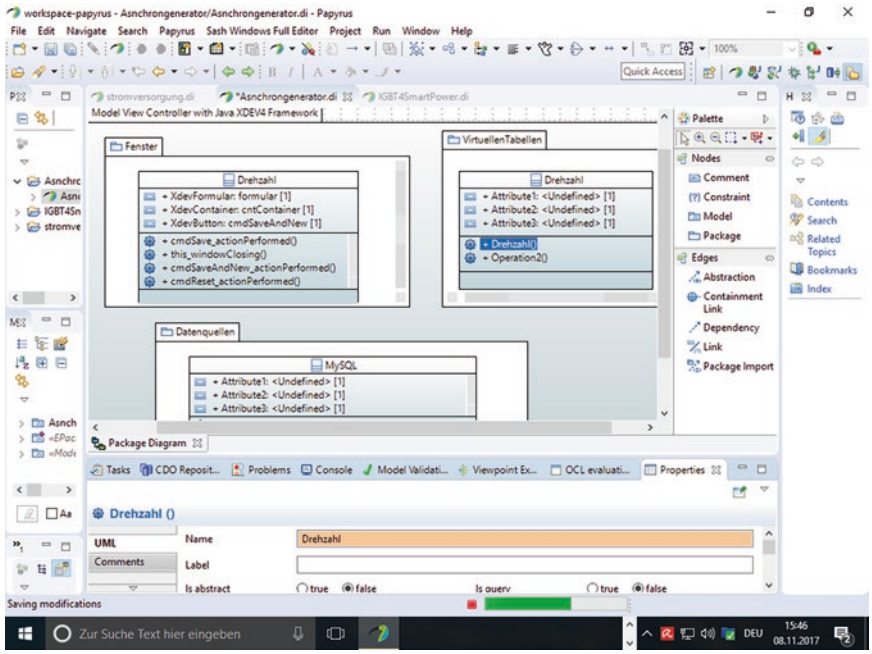


Abb. 3.26 Hinzufügen eines Attributs und einer Operation in die Klasse *„Drehzahl“*

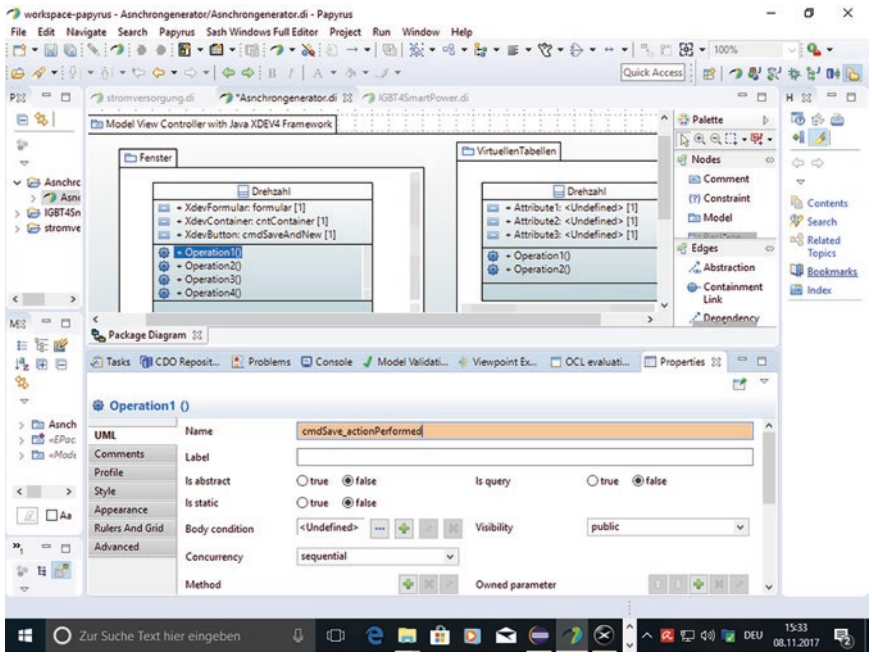


Abb. 3.27 Überblick die Struktur der Operation *„cmdSave_actionPerformed“*

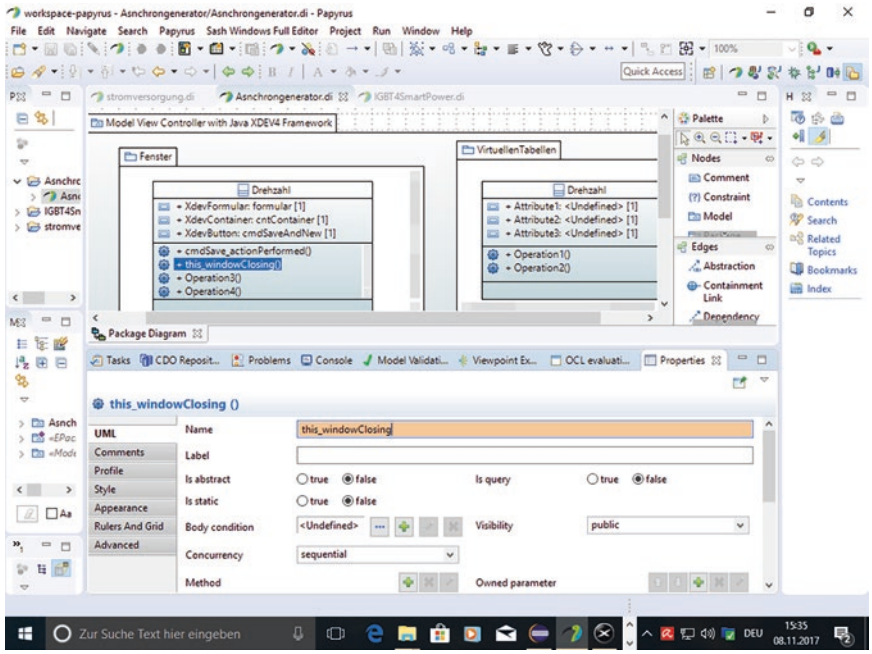


Abb. 3.28 Überblick über die Operation *„this_windowClosing“* der Klasse *„Drehzahl“* des Pakets *„Fenster“*

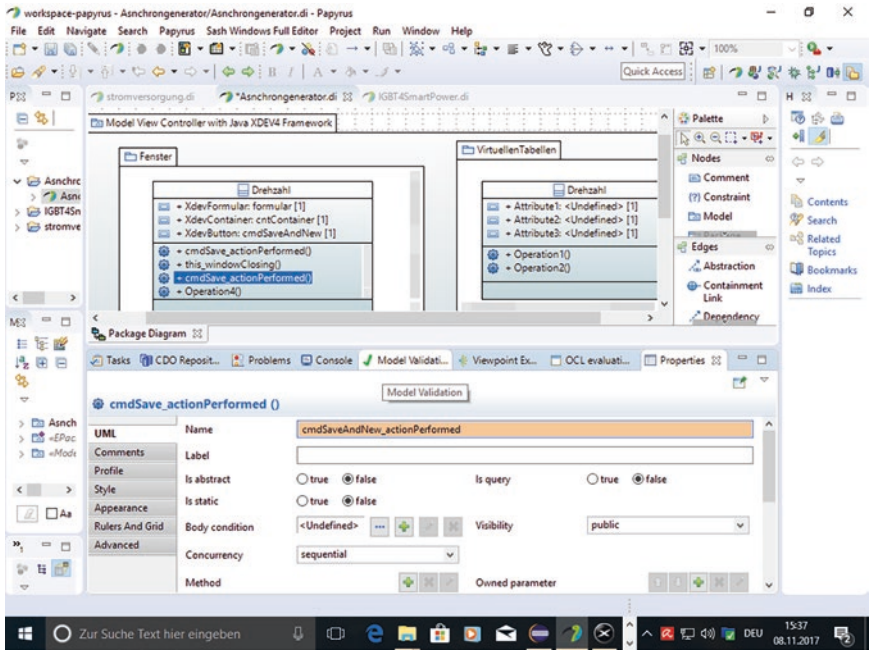


Abb. 3.29 Validierung der Operation *„cmdSave_actionPerformed„ der Klasse „Drehzahl“*

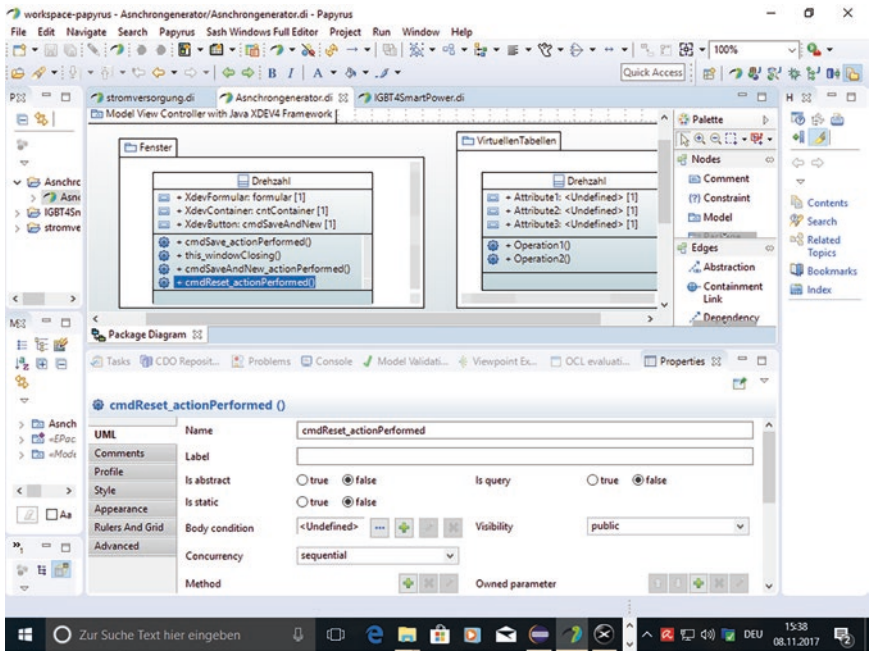


Abb. 3.30 Hinzufügen der Operation *„cmdReset_actionPerformed“*

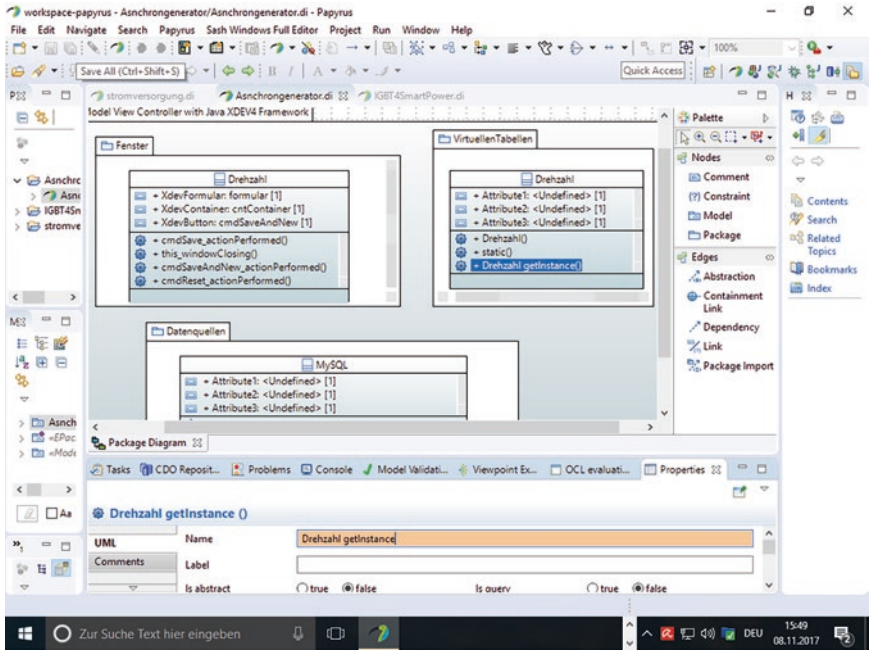


Abb. 3.31 Validierung der Operation *„Drehzahl“* der virtuellen Tabelle

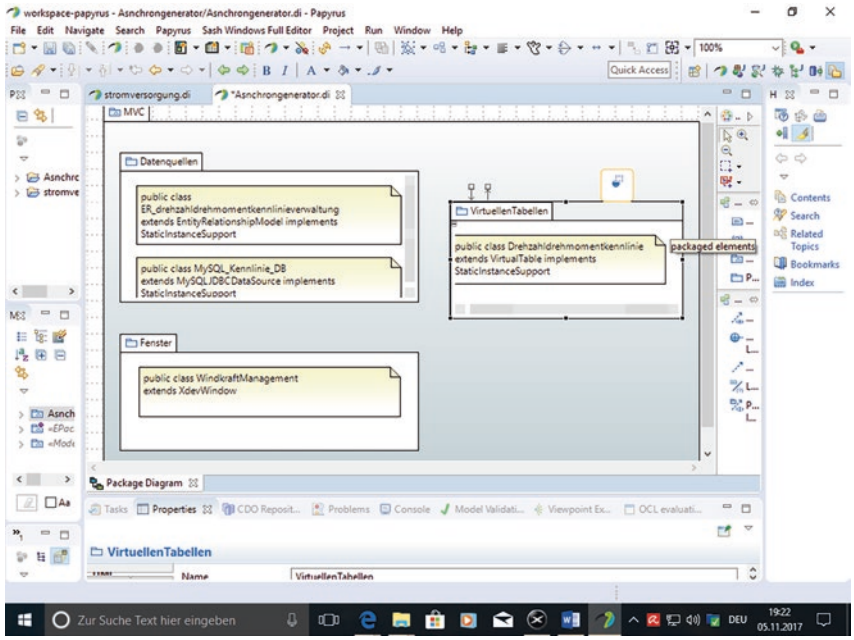


Abb. 3.32 Überblick über das Paketdiagramm in Bezug auf Beschreibung der Vererbung zwischen den Klassen mithilfe der Kommentare

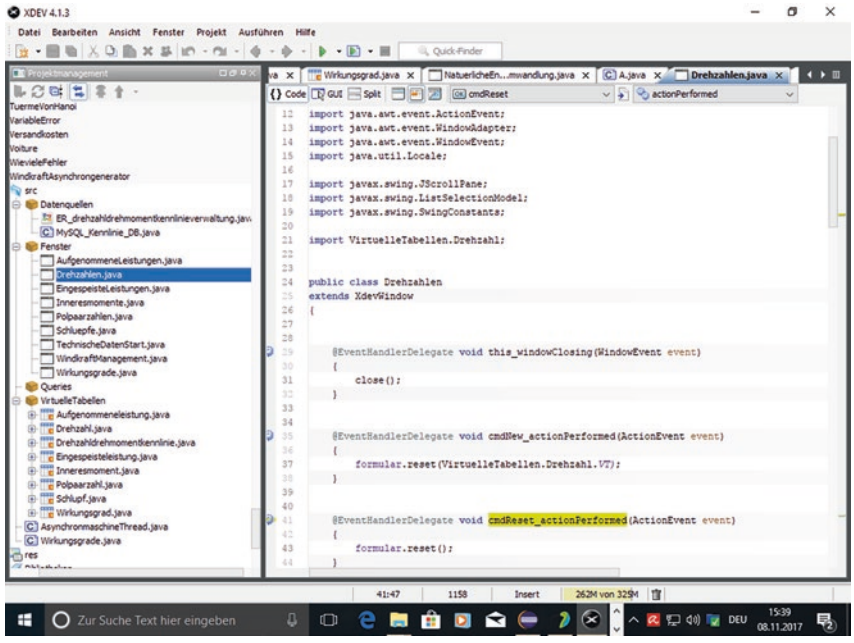


Abb. 3.33 Darstellung des Java-Codings bei der Klasse *„Drehzahlen“* und des Design Patterns MVC mithilfe des Editors vom Java XDEV

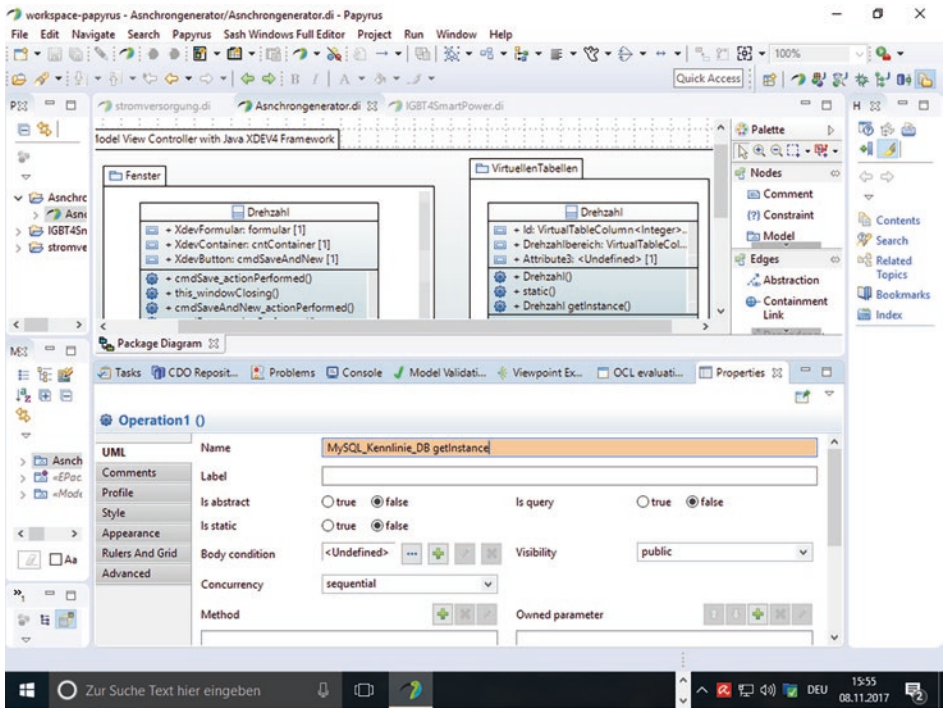


Abb. 3.34 Überblick über die Klasse *Drehzahl* des Modell-Pakets *„VirtuellenTabellen“*

3.3 *„Class Tree Table“*

Klassen-Bäume-Tabelle genannt *Class Tree Table* ist eine UML-Tabelle vom Famework Eclipse-Papyrus. Diese Tabelle fokussiert auf die Darstellung der Klassen und ihren Attributen und Operationen mithilfe von Spalten und Zellen. Die Struktur dieser Tabelle vom Eclipse-Papyrus ähnelt dieser von Excel-Tabellen. Mithilfe dieser Tabelle sind die Klassen als Bäume-Diagramm dargestellt. Wer mit Kalkulationstabelle Spalten und Zeilen ausgefüllt hat, wird keine Schwierigkeiten beim Modellieren der Klassen mit dem *„Class Tree Table“* haben. Diese Tabelle stellt eine Erleichterung zum Organisieren der Eigenschaften der Klassen dar. Hierbei sind Klassen, Attribute und Operation nach Nummerierung gelistet. Die Tabelle gibt einen Überblick über das gesamte Klassensystem des Modells. Diese Tabelle zeigt die Struktur der Klassen hinsichtlich ihrer vertikalen und horizontalen Position.

3.3.1 Struktur der Tabelle

Klassen-Baum-Tabelle geben Informationen mithilfe Tabellen über die vertikale Position. Das Erstellen dieser Tabelle mit dem Framework Eclipse-Papyrus beginnt mit dem Projekt-Erstellen, wie auf Abb. 3.35 und 3.36 zu sehen ist. Der Projekt- und Modellname sind nach *„Schwingkreiswecherichter“* genannt. Das Auswählen der *„Class Tree Table“* folgt der gleichen Prozedur wie bei anderen UML-Diagrammen mit Eclipse-Papyrus. Abb. 3.36 verdeutlich die Position der *„Klassen-Baum-Tabelle“* nach dem Klassendiagramm. Auf Abb. 3.37 ist die Struktur einer einfachen Tabelle mit Klassen dargestellt. Hierbei sind fünf Spalten mit Titeln zu sehen. Die erste dient der Nummerierung der Klassen in der Tabelle. Die zweite zeigt alle Klassen der Tabelle, wobei das Zeichen „−“ die Sichtbarkeit der Klassen dargestellt. Die dritte Spalte listet alle Klassen der Tabelle nach ihren Positionsnummern in der Tabelle. Die vorletzte Spalte zeigt die Eingabe über die Klassen-Namen, wobei die Spalte über den Typ *„String“* verfügt. Abb. 3.37 zeigt auch im Properties-Bereich das Hinzufügen einer neuen Klasse, genannt *„IGBT“*. Außerdem gibt Abb. 3.37 im Bereich Properties Überblicke über die Struktur der hinzugefügten Klasse bezüglich des Namens, Sichtbarkeit und der Eigenschaften. Abb. 3.37, 3.38, 3.39 und 3.40 zeigt die Baum-Struktur im Model Explorer auf der linken Seite des Editors. Hierbei sind die Klassen nach ihren Positionsnummern eingeordnet.

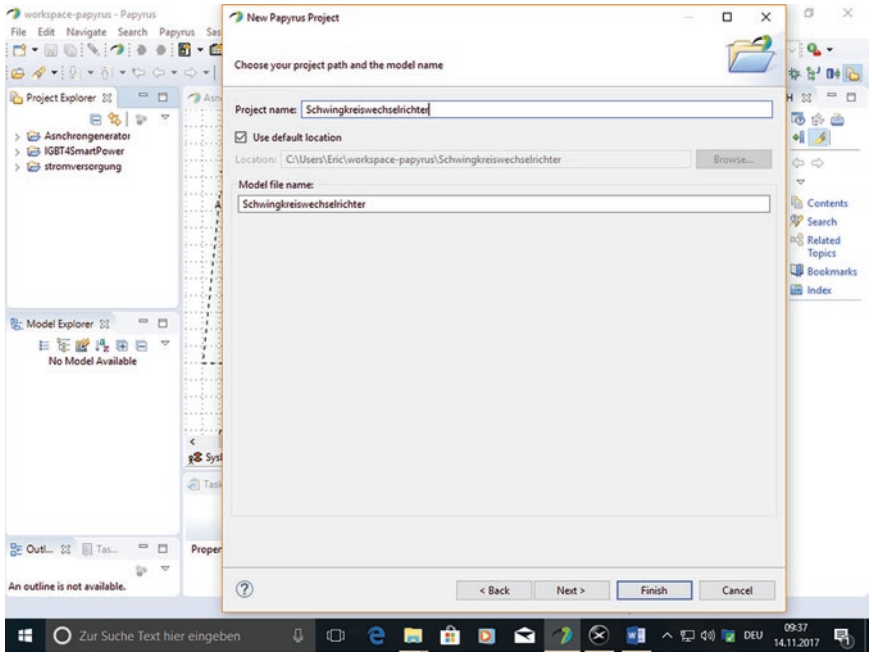


Abb. 3.35 Erstellung des Projekts *„Schwingkreiswechelrichter“*

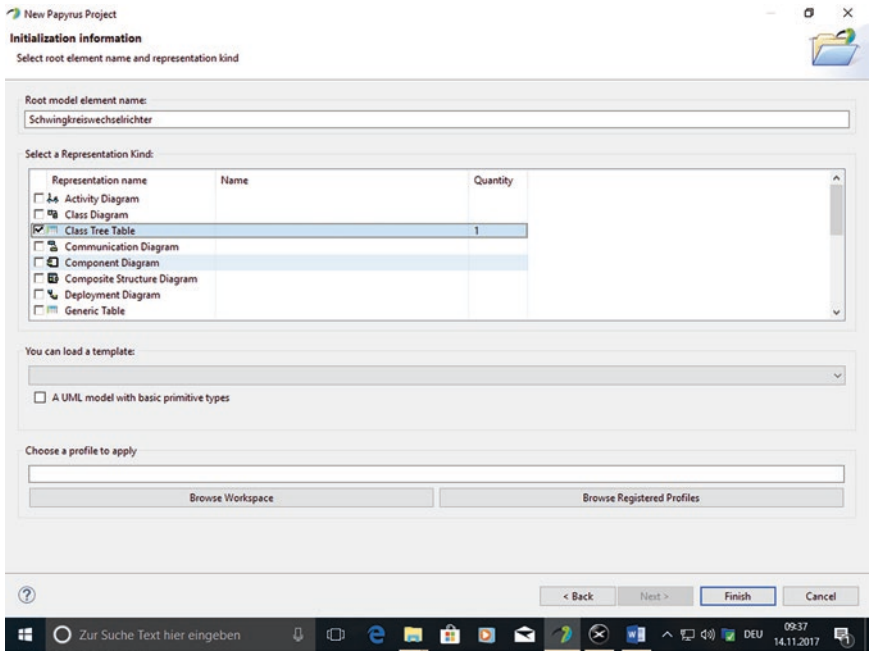


Abb. 3.36 Auswählen des Diagrammtyps *„Class Tree Table“* vom Framework Papyrus

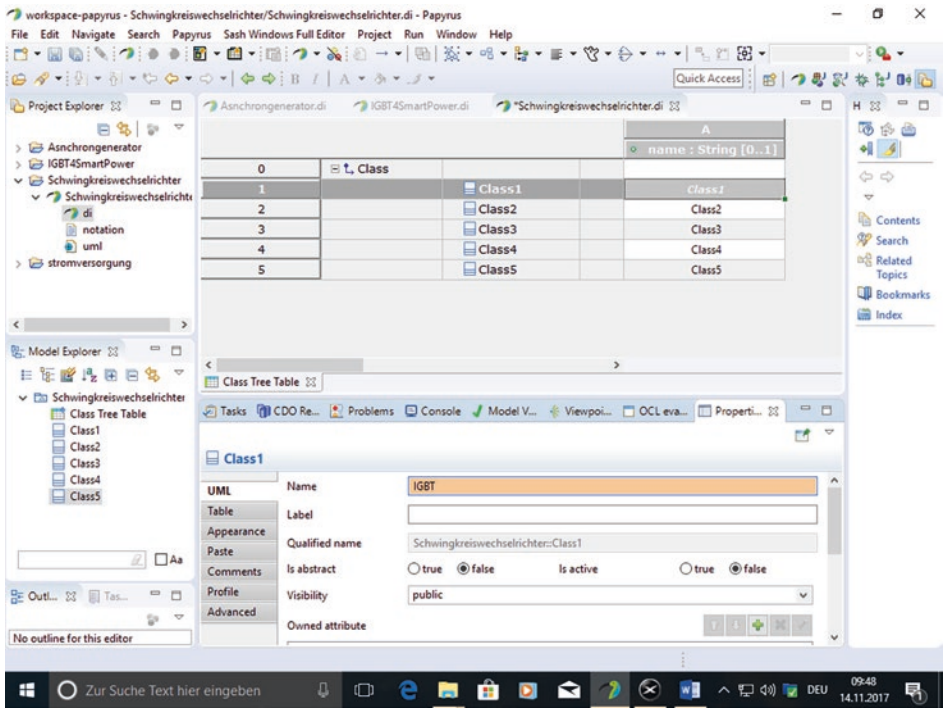


Abb. 3.37 Überblick über die Struktur der *„Klassen-Baum-Tabelle“*

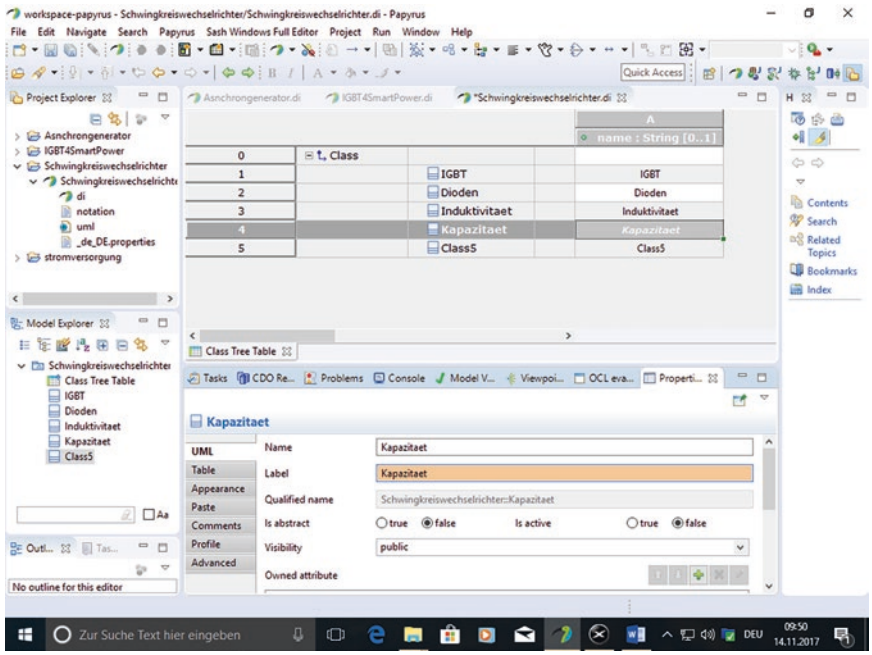


Abb. 3.38 Überblick über die Struktur der Klasse *„Kapazitaet“*

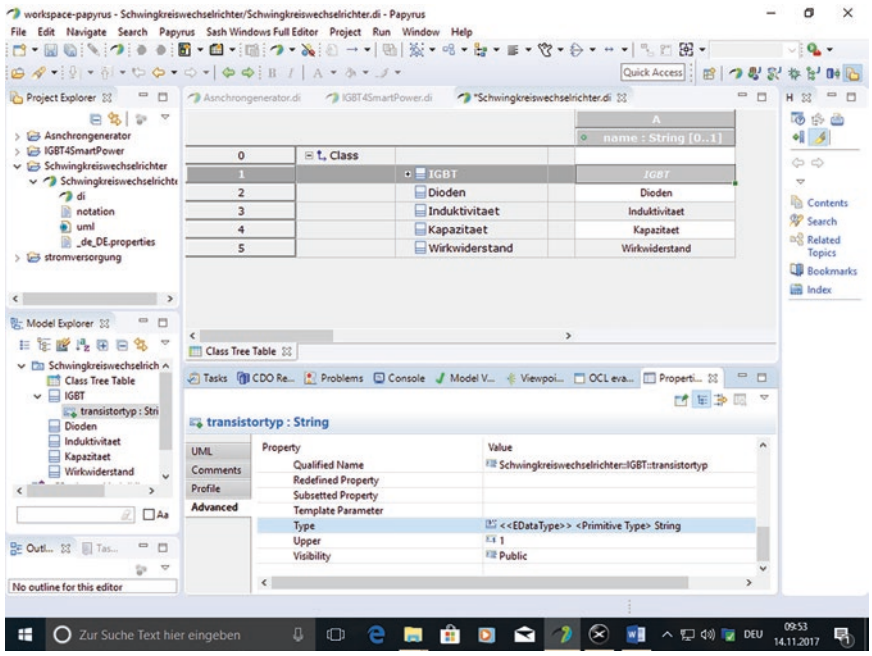


Abb. 3.39 Darstellung der Klassen-Baum-Tabelle im Hinblick auf die Eigenschaft des Attributs *„transistortyp“*

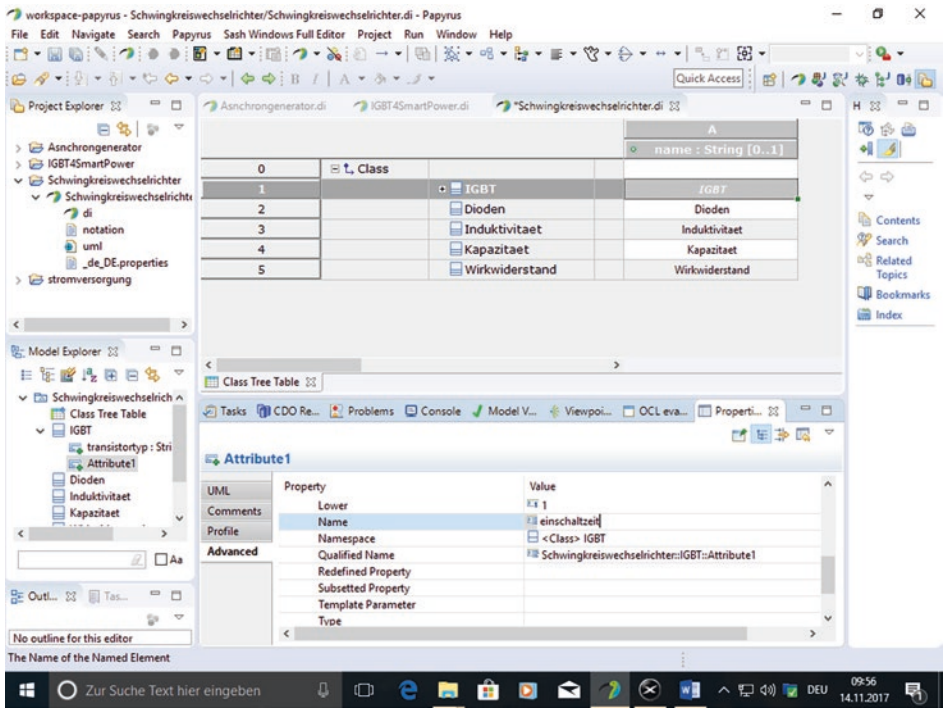


Abb. 3.40 Das Hinzufügen des Attributs *„einschaltzeit“* in die Klasse *„IGBT“*

3.3.2 Vertikale Position

Die Tabelle stellt eine geometrische Position in Bezug auf Spalten und Zeilen. Die Klassen sind gelistet nach ihren Positionsnummern in der Tabelle. Dies entspricht einer vertikalen Orientierung in der Tabelle. Diese Orientierung folgt einer vertikalen Achse. Die Klassen in der vertikalen Position sind von oben nach unten eingerichtet. Abb. 3.37, 3.38, 3.39, 3.40, 3.41 und 3.42. zeigt hinsichtlich der vertikalen Position der Klassen in der Tabelle eine Darstellung der Funktionsweise eines *„Schwingkreiswechselrichters“*. Die vertikale Anordnung gibt einen Überblick über die Komponenten eines *„Schwingkreiswechselrichters“*. In der Tabelle entspricht jede Klasse einer Komponente des Wechselrichters. Zum einen sind IGBT und Dioden Bauelemente und zum anderen stellen *„Kapazitaet“*, *„Induktivitaet“* und *„Wirkwiderstand“* den Schwingkreis dar. In der vertikalen Position sind IGBT, Dioden, Induktivitaet, Kapazitaet und Wirkwiderstand in der Spalte nach der Reihenfolge eingeordnet. Die Klassen in der vertikalen Position sind als Rechte. Die Zahlen *„0“*, *„1“*, *„1“*, *„3“*, *„4“* und *„5“* entsprechen *„Class“*, *„IGBT“*, *„Dioden“*, *„Induktivitaet“*, *„Kapazitaet“* und *„Wirkwiderstand“* (Abb. 3.41).

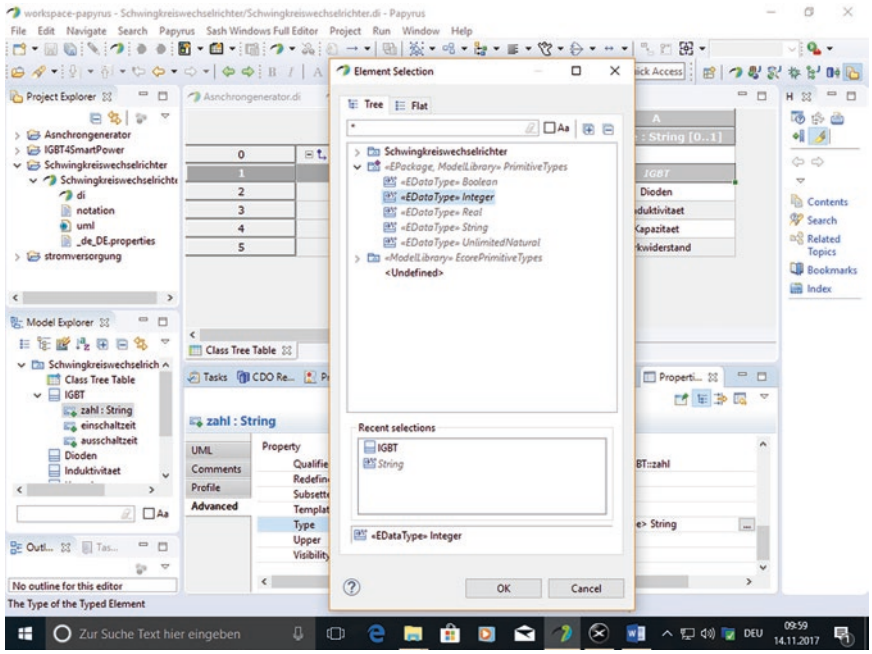


Abb. 3.41 Das Hinzufügen des Datentyps „*EDataType Integer*“ in die Klasse IGBT

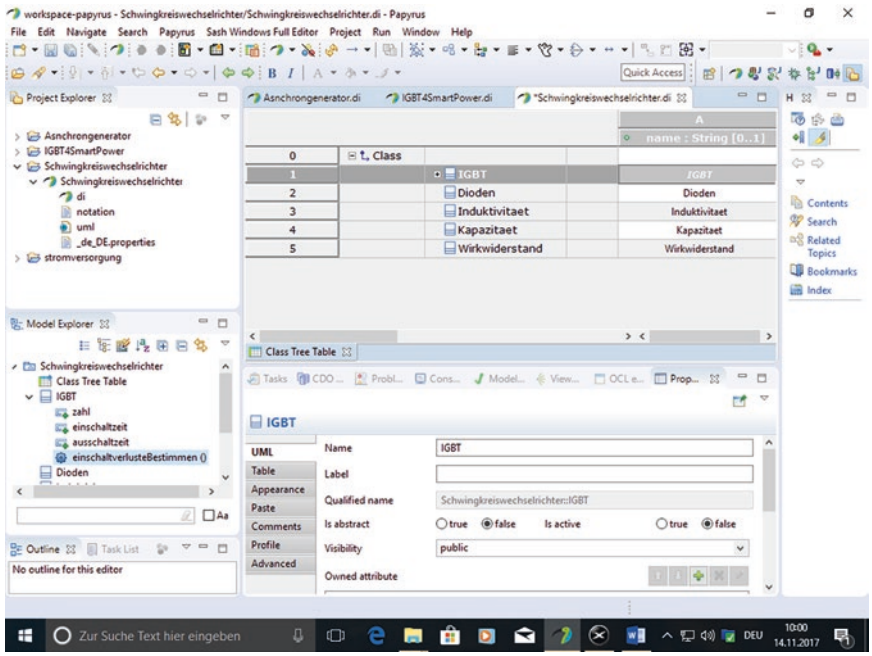


Abb. 3.42 Überblick über Attribut und Operation der Klasse „*IGBT*“ im Model Explorer

3.3.3 Horizontale Position

Die horizontale Position in der *„Class Tree Table“* ermöglicht die Darstellung der Eigenschaften der Klassen hinsichtlich ihrer Attribute und Operationen. Diese geometrische Orientierung stellt die horizontale Achse eines Cartesischen Systems dar. Abb. 3.44, 3.45, 3.46, 3.47, 3.48, 3.49 und 3.50 geben Überblicke über die horizontale Position der Klassen in Bezug auf die Verschiebung der Klassen. Dies ist mithilfe des Hinzufügens der Attribute und Operationen möglich. Es entsteht eine Steigerung der Zahl der Nummerierung. Abb. 3.43 und 3.44 zeigt das Hinzufügen einer neuen Operation, genannt *„Operation1“*, in der Klasse *„IGBT“*. Abb. 3.44 zeigt im *„Model Explorer“* die Struktur des Modells *„Schwingkreiswechselrichter“*, das die Klassen-Baum-Tabelle enthält. Diese enthält die Klasse *„IGBT“*, die über Attribute und Operationen verfügt. Die Klassen im Model Explorer sind nach dem Prinzip der objektorientierten Modellierung aufgeführt. Auf der Abb. 3.45 sind zum einen die Attribute *„zahl“*, *„einschaltzeit“* und *„ausschaltzeit“* und zum anderen die Operationen *„einschaltzeitBestimmen()“* und *„ausschaltzeitBestimmen()“* zu sehen. Wie auf der Abb. 3.45 sind weitere Klassen wie z. B. *Dioden* mit Attributen auch auf den Abb. 3.46 und 3.47 zu sehen. Im unteren Teil des Editors ist im *Property-Bereich* für die Klasse Dioden die Syntax des Attributs, genannt *„sperrverzoegerungsstrom“*, zu sehen. Mit der Abb. 3.48 ist die Klasse *„IGBT“*

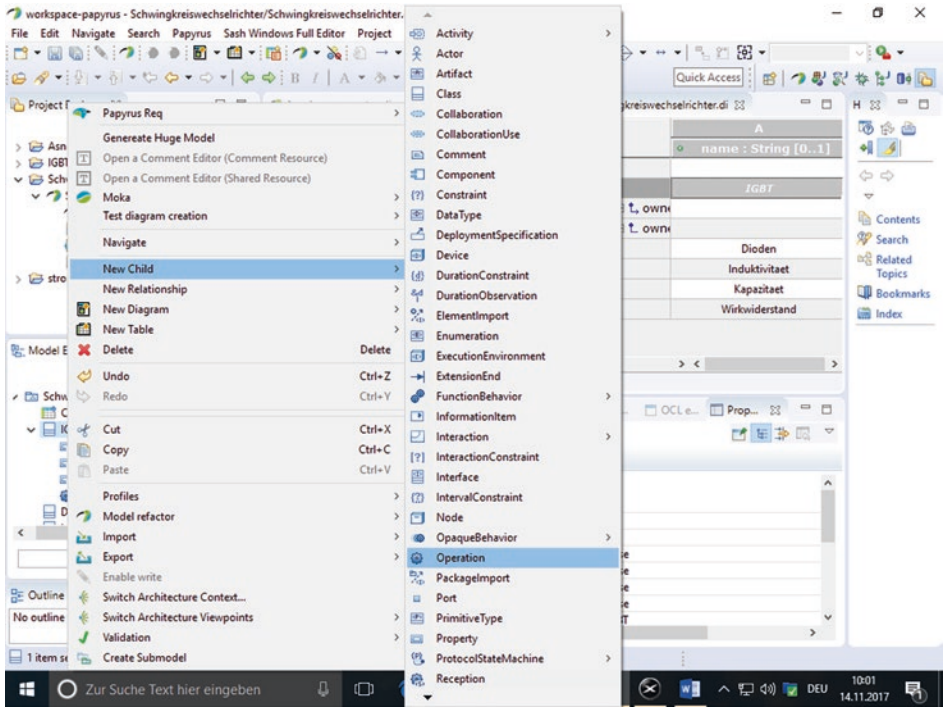


Abb. 3.43 Das Hinzufügen einer Operation mithilfe von *„New->New Child->Operation“*

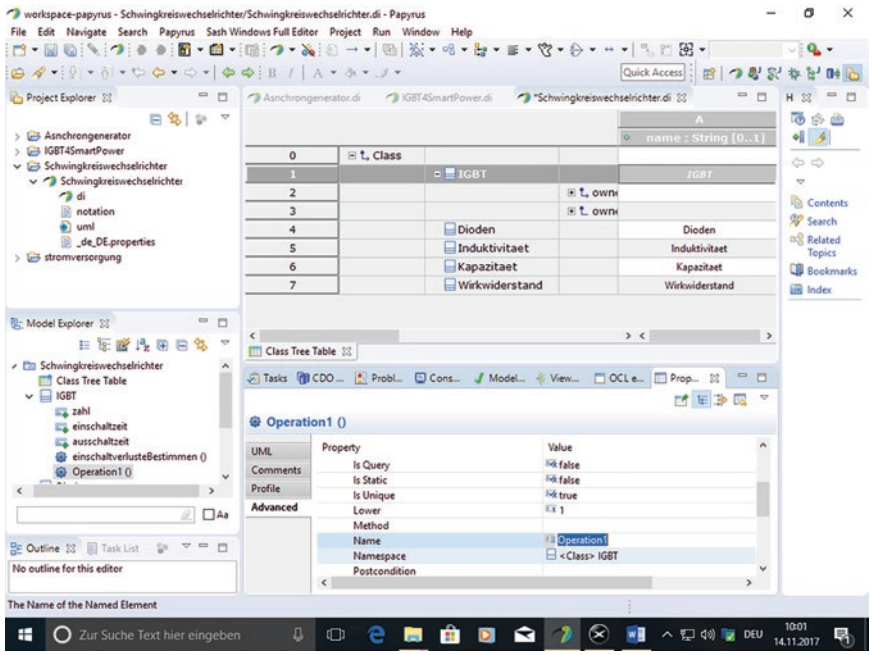


Abb. 3.44 Darstellung der horizontalen Position mit Hinblick auf die Operation „*Operation1*“

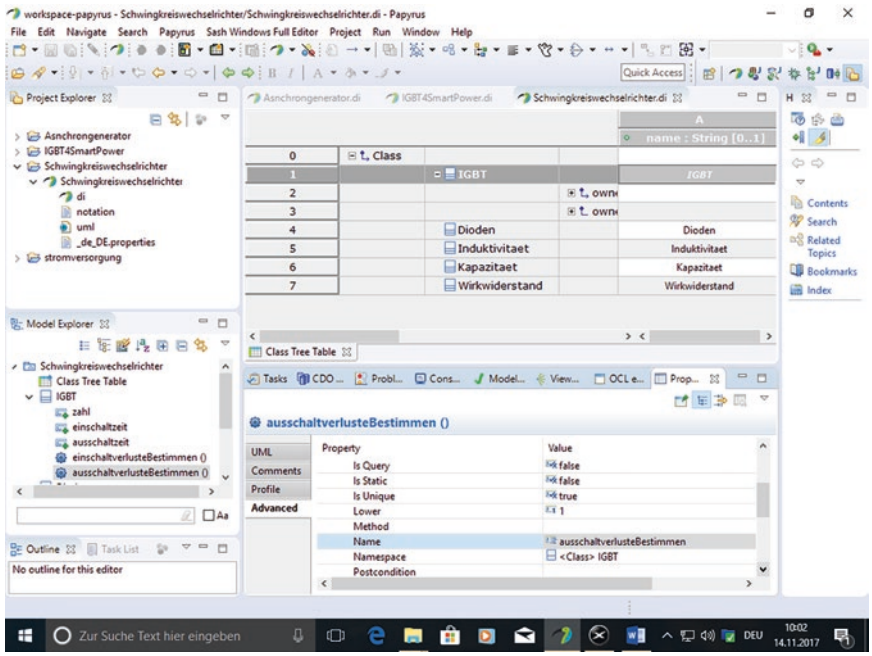


Abb. 3.45 Überblick über die Operation „*Operation1*“ im Model Exporer

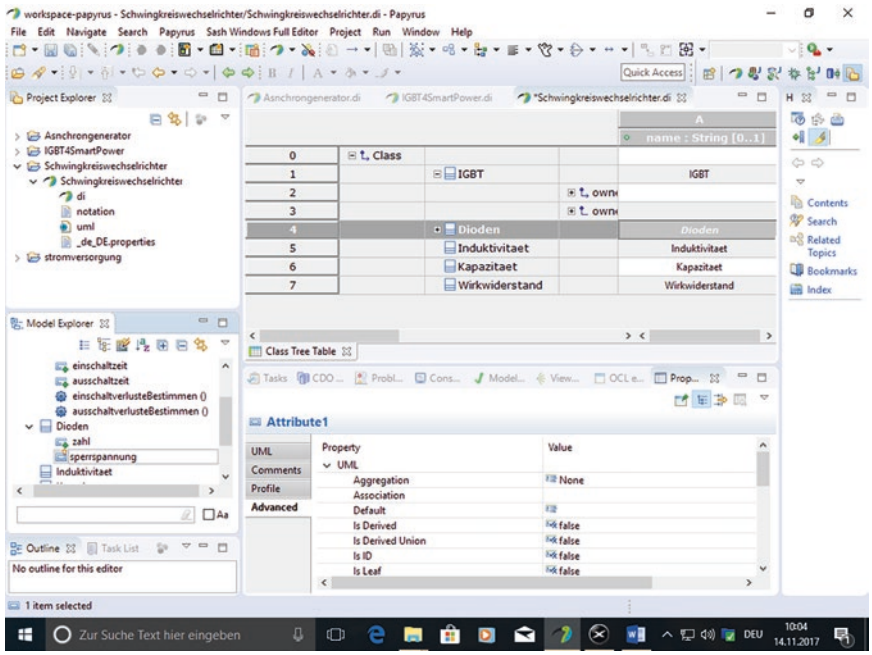


Abb. 3.46 Struktur der horizontalen Position im Hinblick auf Attribute und Operationen im Model Explorer

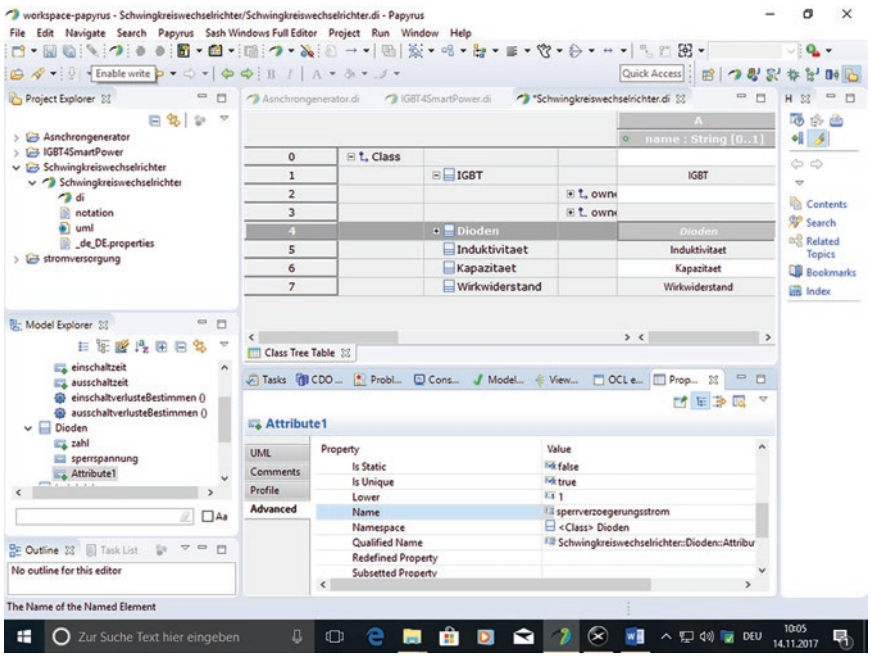


Abb. 3.47 Darstellung der Syntax des Attributs *„sperrverzoegerungsstrom“* der Klasse *„Diode“* im Properties-Bereich

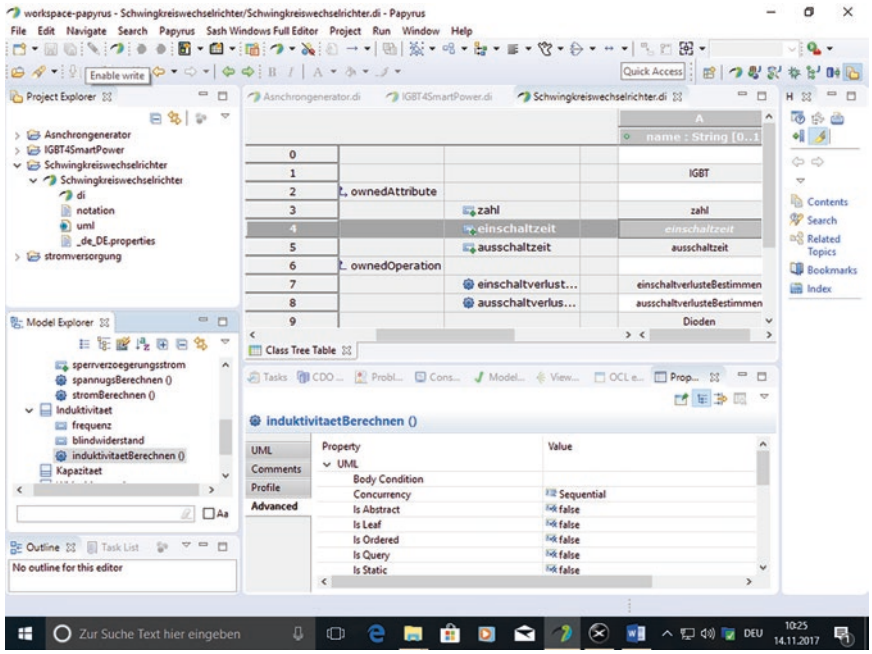


Abb. 3.48 Darstellung von Attributen und Operationen der Klassen-Baum-Tabelle

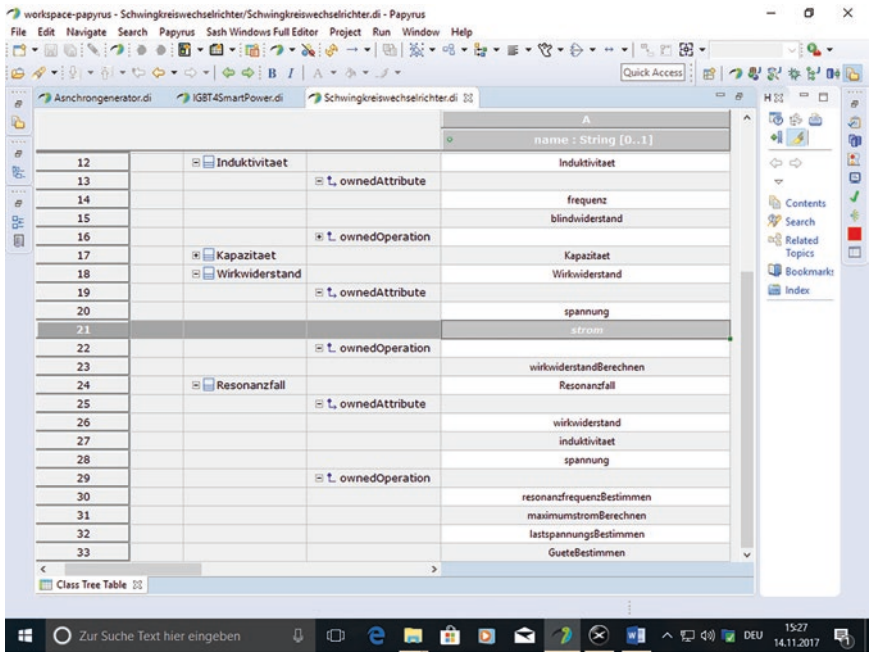


Abb. 3.49 Überblick über horizontale Position mithilfe von Attributen und Operationen der Klasse IGBT in der Klassen-Baum-Tabelle

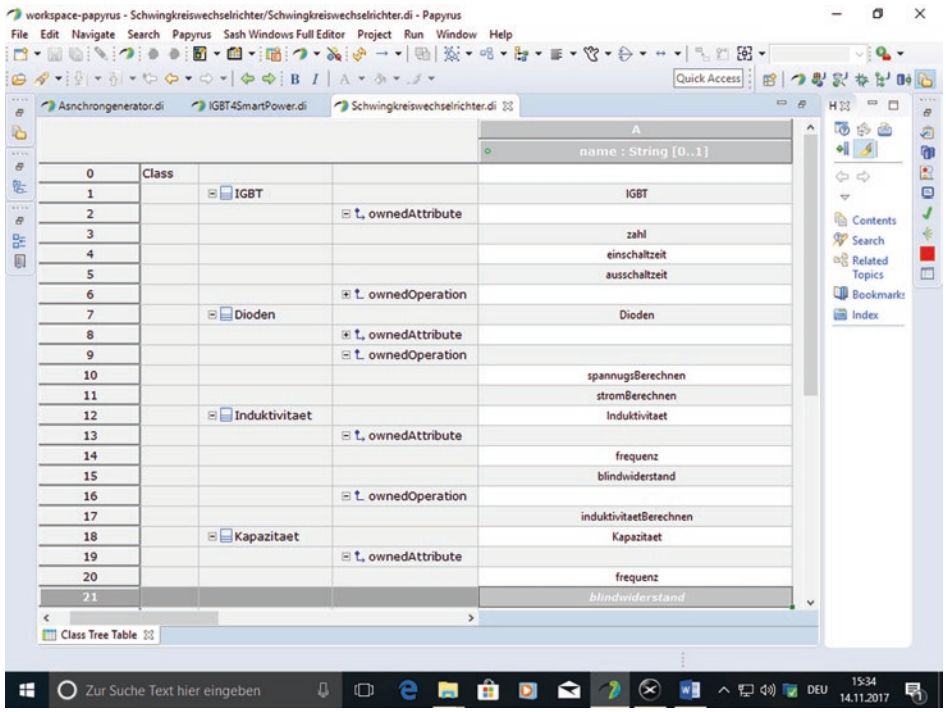


Abb. 3.50 Darstellung der horizontalen Position mit Attributen und Operation der Klasse *„Kapazitaet“* im Model Explorer

mit ihren Attributen und Operationen horizontale zu sehen. Die Klasse *„IGBT“* erstreckt sich von Zeile 1 bis 8. Auf jeder Zeile befindet sich eine Eigenschaft der Klasse. Die Begriffe *„ownedAttribute“* und *„ownedOperation“* belegen die Zeile für Attribute bzw. Operation. Zum Beispiel nach *„ownedAttribute“* entsprechen die Attribute *„zahl“*, *„einschaltzeit“* und *„ausschaltzeit“* die Zeile 3, 4 und 5. Anschließend belegen die Operation *„einschaltverlustBestimmen()“* und *„ausschaltverlustBestimmen()“* der Klasse *„IGBT“* die Zeilen 7 bzw. 8. Abb. 3.49 und 3.50 geben einen Überblick über das Ausdehnen der verschiedenen Klassen in der Klassen-Baum-Tabelle nach der horizontalen Orientierung. Die horizontale Position ermöglicht die Darstellung der Klassen nach der Position von *„ownedAttribute“* und *„ownedOperation“*. Mit den Abb. 3.49 und 3.50 sind die Verteilung der Klassen nach der Ausdehnung der Zeilen dargestellt.

3.4 Sequenzdiagramme mit Eclipse-Papyrus

Sequenzdiagramme ermöglichen die dynamische Modellierung bezüglich der Interaktionen zwischen Teilen des Systems. Mit dem Framework Eclipse-Papyrus fokussieren die Sequenzdiagramme auf die Lebenslinien der Objekte und deren Interaktionen mit

anderen Objekten. Bei Sequenzdiagrammen handelt es sich um eine Art von Interaktionsdiagramm, denn sie beschreiben wie und in welcher Reihenfolge eine Gruppe von Objekten zusammenarbeitet. Software-Entwickler und Unternehmen greifen auf diese Diagramme zurück, um zu verstehen, welche Anforderungen ein neues System stellt oder um bereits bestehende Prozesse grafisch festzuhalten. Sequenzdiagramme werden manchmal auch als Ereignisdiagramme oder Ereignisszenarien bezeichnet [7]. Im Sequenzdiagramm werden Objekte in einer Reihe nebeneinander dargestellt und mit einer nach unten gerichteten Zeitlinie versehen. Objekte tragen optional einen Namen und einen Typ. Es werden aber im Gegensatz zum Objektdiagramm keine Attribute und keine Links dargestellt. An einem Objekt ankommende und abgehende Interaktionen sind in zeitlicher Reihenfolge durch das Auftreffen auf der Zeitlinie dargestellt und könnten in dieser Form von einem Beobachter protokolliert worden sein [8]. Sequenzdiagramme beschreiben die Kommunikation zwischen Objekten in einer bestimmten Szene. Es wird beschrieben, welche Objekte an der Szene beteiligt sind, welche Informationen (Nachrichten) sie austauschen und in welcher zeitlichen Reihenfolge der Informationsaustausch stattfindet. Sequenzdiagramme enthalten eine implizite Zeitachse. Die Zeit schreitet in einem Diagramm von oben nach unten fort. Die Reihenfolge der Pfeile in einem Sequenzdiagramm gibt die zeitliche Reihenfolge der Nachrichten an [3].

Abb. 3.51, 3.52, 3.53, 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.61 zeigen das Erstellen eines Sequenzdiagramms mit dem Framework Eclipse-Papyrus. Die Abb. 3.52, 3.53, 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.61 geben Überblicke über die Funktionalität der Lebenslinienmithilfe der Modellierungstools, genannt *„Palette“*. Die Modellierung mit dem Framework Eclipse-Papyrus beginnt mit dem Auswählen des Diagrammtyps *„Sequence Diagram“* im Projekt *„DrehzahlVerstellung“*, wie auf der Abb. 3.51 zu sehen ist. Auf den Abb. 3.52 und 3.53 ist das Hinzufügen der Lebenslinien mithilfe des Modellierungswerkzeuges, genannt *„Palette“,* wobei Lebenslinien Überblicke über die aufeinanderfolgenden Ereignisse geben, die sich mit Bezug auf ein Objekt und im Verlauf des dargestellten Prozesses ergeben. Abb. 3.53, 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.61 stellen Lebenslinien als gestrichelte, vertikale Linien dar, welche sich analog zum Ereignisverlauf nach unten hin ausdehnen. Abb. 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.61 zeigen die Position der Aktivitätsbalken und deren Hinzufügen. Sie sind als rechteckige Form dargestellt und geben die Zeit an, die ein Objekt zum Abschließen einer Aufgabe braucht. Abb. 3.53, 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.61 stellen Objekte und deren Klassen mithilfe der Objekt-Symbole zum Beobachten der Verhalten der Objekte dar. Z. B. sind *„drehzahl“* das Objekt der Klasse *„Interface4Drehzahl“* und *„moment“* dieses der Klasse *„1Interface4Moment“*.

Wie auf den Abb. 3.53, 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.61 zu sehen ist, werden der Objektname und der Klassenname in dem Rechteck oberhalb der gestrichelten Linie hinzugefügt. Die senkrechte, gestrichelte Linie stellt die Lebenslinie *(lifeline)* eines Objektes dar. In diesem zeitlichen Bereich befindet sich das Objekt. Das schmale Rechteck auf der gestrichelten Linie stellt eine Aktivierung dar. Eine Aktivierung ist der Bereich zum Ausführen einer Methode des Objektes. Abb. 3.54 zeigt das Hinzufügen der Aktivitätsbalken. Auf Abb. 3.58, 3.59, 3.60 und 3.61 ist das Kommunizieren der Objekte über

Nachrichten zu sehen. Nachrichten, genannt *„Messages“*, werden als Pfeile zwischen den Aktivierungen dargestellt. Der Name der Nachricht (Message) steht an dem Pfeil. Es gibt Start-Message, Sandwich-Message und End-Message wie auf den Abb. 3.58, 3.59, 3.60 und 3.61 zu sehen ist, wobei eine Nachricht zwischen einem sendenden und einem empfangenden Objekt liegt. Beispielsweise liegt die Nachricht *„Message1“* zwischen Objekt 1 *„drehzahl“* und Objekt2 *„drehmoment“*. *„Message2“* stellt eine Sandwich-Nachricht dar. *„Message2“* befindet sich zwischen den Objekten *„drehmoment“* und *„leistung“*. Die End-Nachricht *„Message3“* liegt zwischen den Objekten *„leistung“* und *„wirkungsgrad“*. Mithilfe der Nachrichten werden Objekte erzeugt. Abb. 3.58, 3.59, 3.60 und 3.61 stellen die synchrone Struktur der Nachricht in Bezug auf die ausgefüllte Spitze des Pfeils dar. Der Aufruf der Nachricht erfolgt von der Quelle-Klasse wie z. B. *„Interface4Drehzahl“* zum Ziel-Klasse wie z. B. *„Interface4Drehmoment“*, wobei die Ziel-Klasse eine entsprechende Methode implementiert. Die Quelle-Klasse wartet mit der weiteren Verarbeitung bis die Ziel-Klasse ihre Verarbeitung beendet hat und setzt die Verarbeitung dann fort. Ein Aufruf muss einen Namen haben; in runden Klammern können Aufrufparameter angegeben werden [3]. Asynchrone Nachrichte werden im Gegenteil zur synchronen Nachricht mit einer offenen Pfeilspitze dargestellt. Von der Quelle-Klasse bis zur Ziel-Klasse erfolgt der Anruf. Modellierungswerkzeuge, genannt Palette, auf den Abb. 3.52, 3.53, 3.54, 3.55, 3.56, 3.57, 3.58, 3.59, 3.60 und 3.60 sind in zwei Ordner eingegliedert: *„Nodes“* oder Knoten und *„Edges“* für Verbindungen. *„Nodes“* verfügt über Tools und Symbole wie z. B. *„Lebenslinien“*, *„Aktion-Spezifikation“*, *„Verhalten-Spezifikation“*, Objekt, *„Interaktionen-Anwendung“*, *„Zeit-Beobachtung“* und *„Kommentare“* während *„Edges“* aus Verbindungen wie z. B. *„ContextLink“* und Nachrichten wie z. B. *„Message Async“* oder *„Message Sync“* besteht.

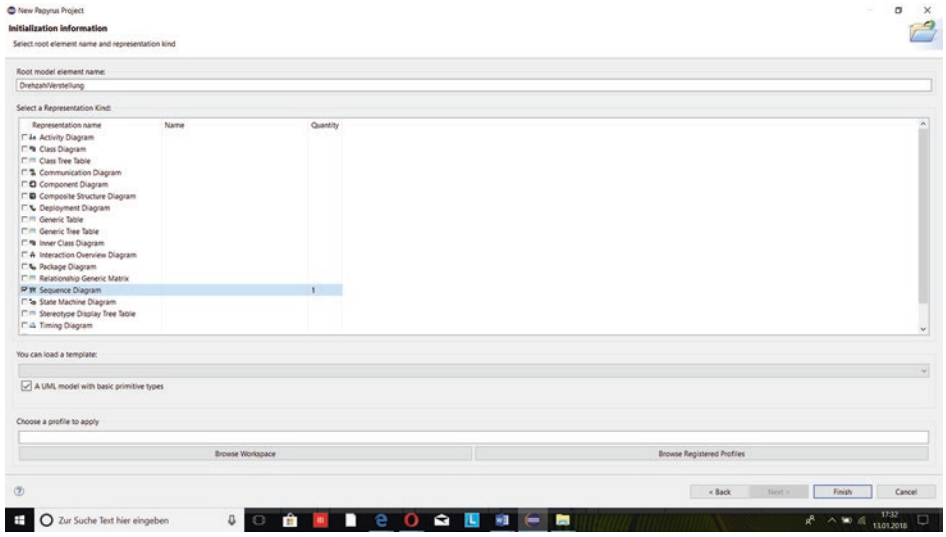


Abb. 3.51 Auswählen des Diagrammtyps *„Sequence Diagram“*

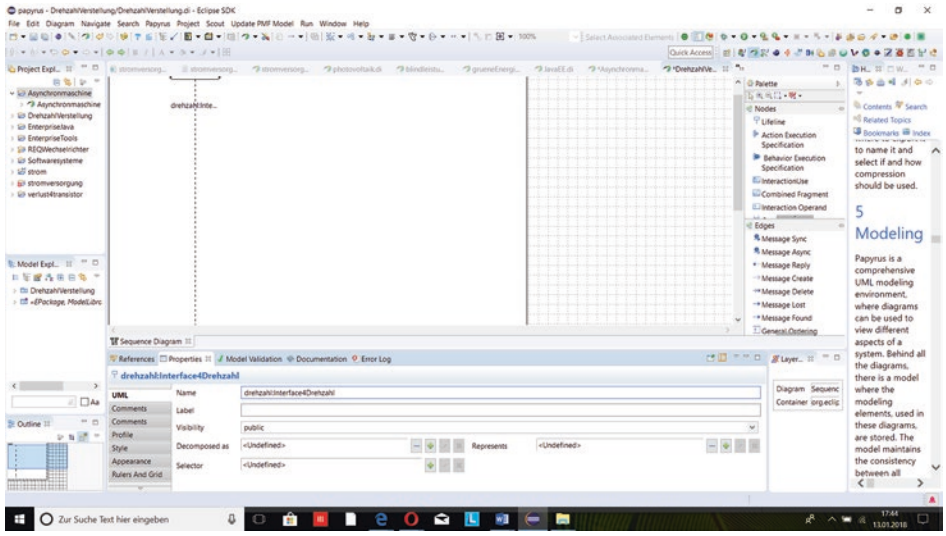


Abb. 3.52 Überblick über das Erstellen einer Lebenslinie für das entsprechende Objekt *„drehzahl“*

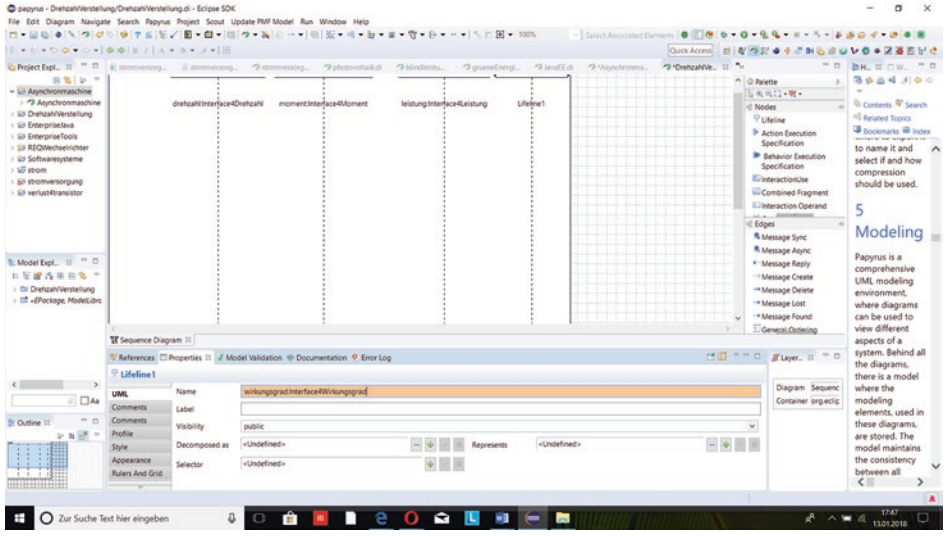


Abb. 3.53 Horizontale Position der Objekte und Klassen mit deren Lebenslinien und Hinzufügen des Objektes *„wirkungsgrad“*

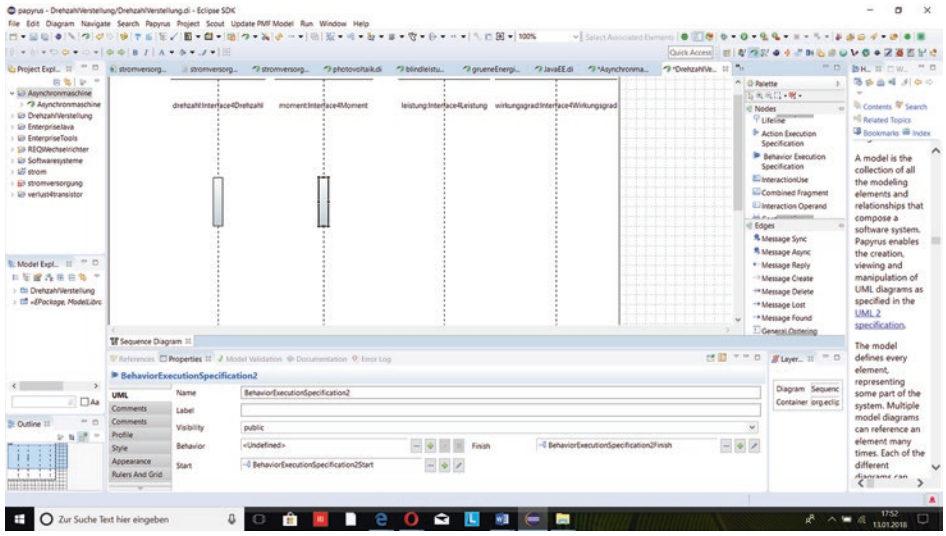


Abb. 3.54 Das Hinzufügen der Aktivitätsbalken

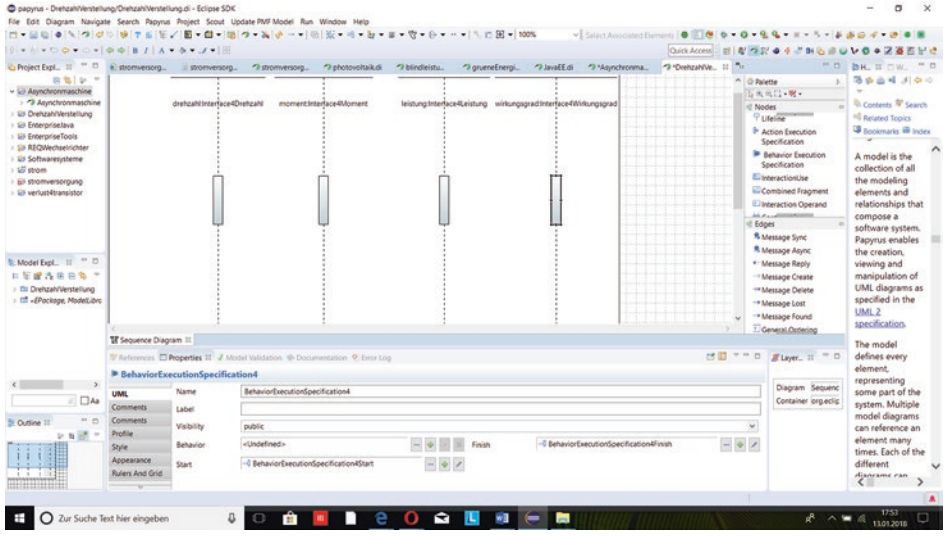


Abb. 3.55 Horizontale Position der Aktivitätsbalken der Objekte mit der Zeit

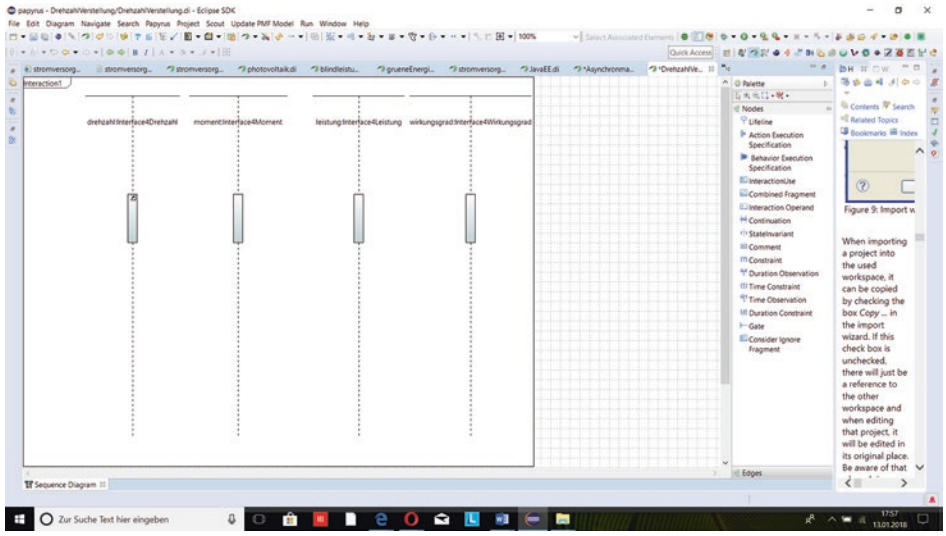


Abb. 3.56 Vertikale Position der Lebenslinien mit deren Aktivitäsbalken im Laufe der Zeit

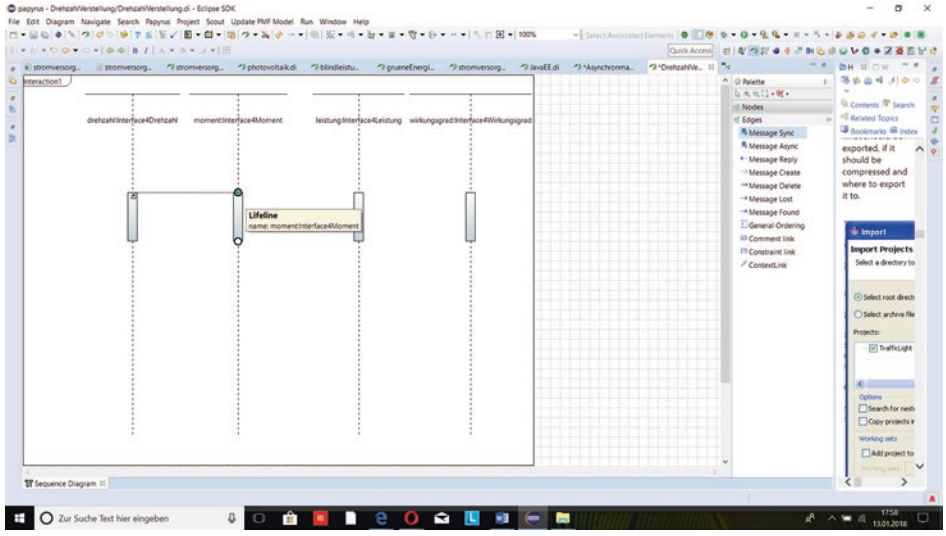


Abb. 3.57 Überblick über die Interaktion zwischen Objekten *„drehzahl“* und *„moment“* mithilfe deren Aktivitätsbalken

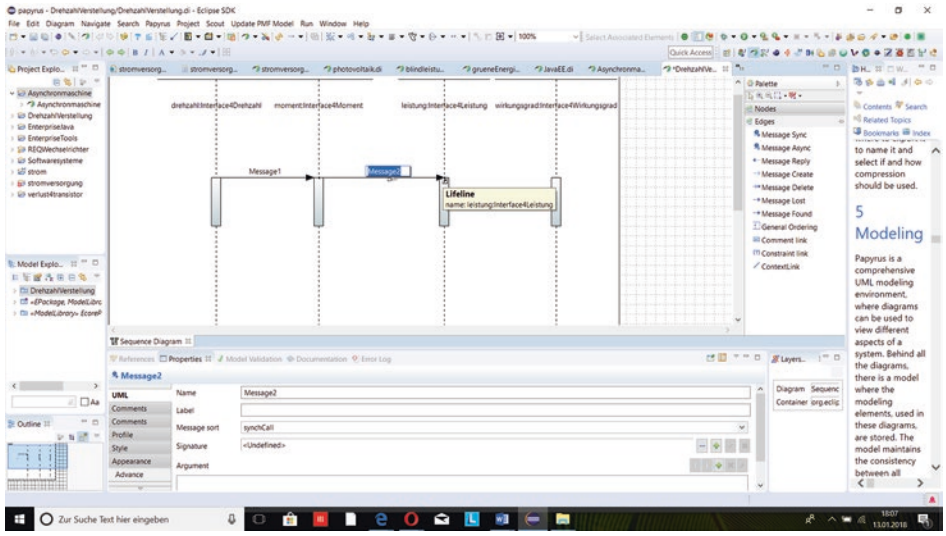


Abb. 3.58 *„Message2“* am Pfeil während der Kommunikation zwischen 2 Objekten

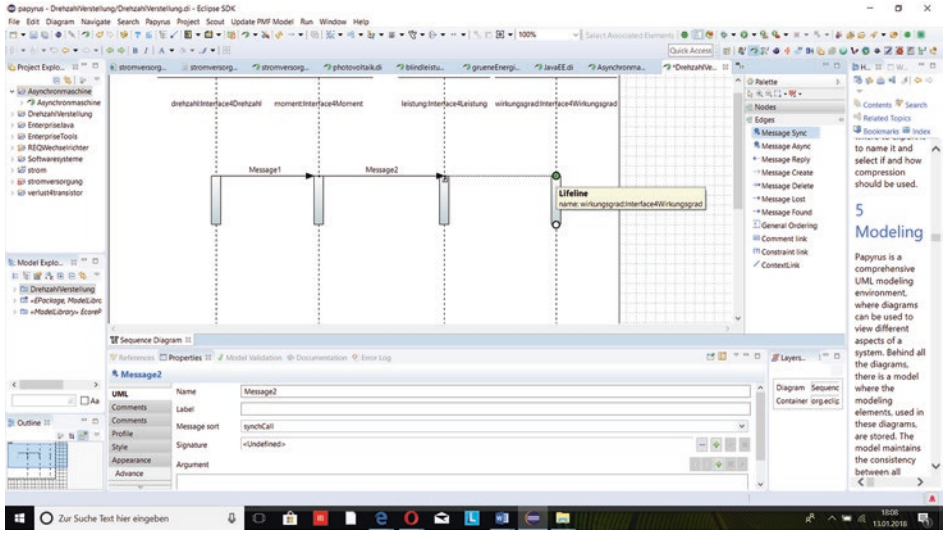


Abb. 3.59 Kaskade Aktivierung mithilfe von synchronen Nachrichten

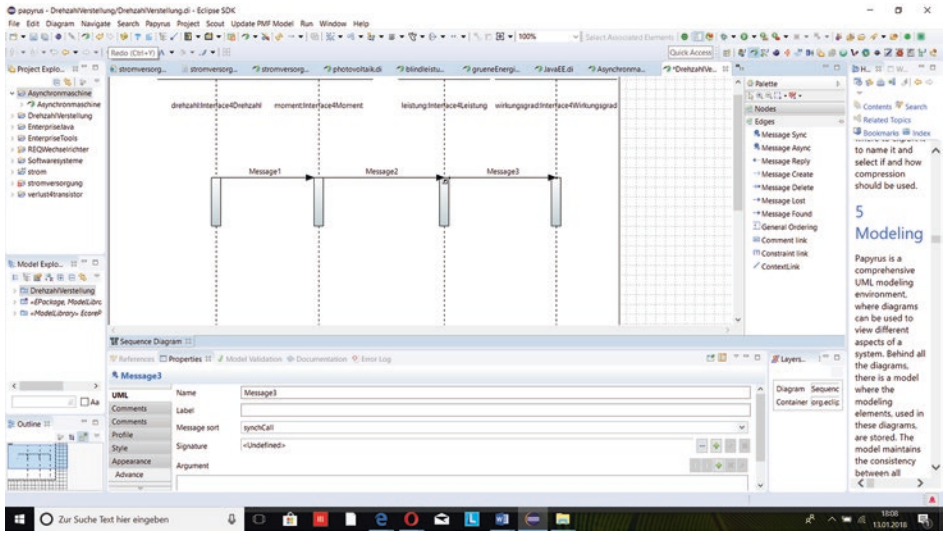


Abb. 3.60 Überblick über die Orientierung der Nachricht mit den entsprechendenen Objekten Eric Nyamsi

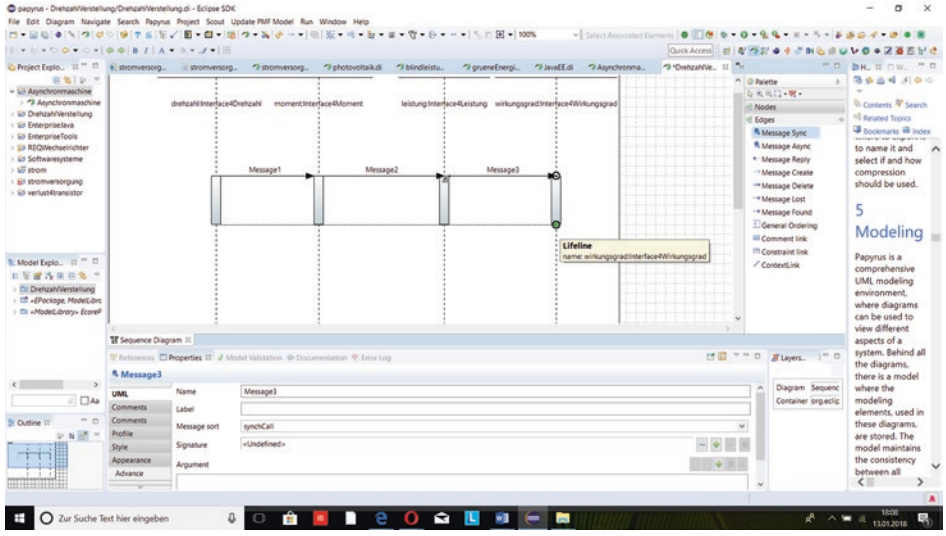


Abb. 3.61 Darstellung der physikalischen Struktur der Nachricht in Bezug auf die ausgefüllte Spitze des Pfeils

3.5 Kommunikationsdiagramm mit Eclipse-Papyrus

Kommunikationsdiagramme stellen die Interaktionen zwischen den Objekten oder Rollen dar, die Lebenslinien zugeordnet sind, sowie die Nachrichten, die zwischen Lebenslinien übergeben werden. Kommunikationsdiagramme sind ein Interaktionsdiagrammtyp, mit dem das dynamische Verhalten eines Systems oder einer Softwareanwendung untersucht werden können. Sie zeigen eine alternative Sicht derselben Informationen wie in Sequenzdiagrammen bereit. In Sequenzdiagrammen liegt der Fokus auf der zeitlichen Reihenfolge der Nachrichten, in Kommunikationsdiagrammen dagegen auf der Struktur der Nachrichten, die zwischen den Objekten in der Interaktion übergeben werden. Diese Diagramme veranschaulichen den Nachrichtenfluss zwischen Objekten und die implizierten Beziehungen zwischen Klassen [9].

Kommunikationsdiagrammen ermöglichen die Analyse der Zusammenwirkung der Objekte in einem System oder einer Anwendung. Kommunikationsdiagramme stellen Aspekte einer Interaktion für Objekte, Schnittstellen der beteiligten Klassen, Strukturänderungen und Übergabe der Daten der Objekte dar. Kommunikationsdiagramme sehen ähnlich aus wie Objektdiagramme, bei denen eine Lebenslinie die Objekte in der Interaktion darstellt und Pfeile für die Nachrichten stehen, die zwischen den Lebenslinien übergeben werden. Pfeilspitzen geben die Richtung der Nachrichten an (vorwärts oder rückwärts), Folgenummern die Reihenfolge, in der die Nachrichten übergeben werden [9].

Abb. 3.61, 3.62, 3.63, 3.64, 6.65, 3.66, 3.67, 3.68 und 3.69 geben Überblicke über das Erstellen des Kommunikationsdiagramms zur Web-Sicherheit mit dem Framework Eclipse-Papyrus. Das Beispiel fokussiert auf sicherheitsrelevante Header wie z. B. X-Frame-Options, X-Content-Type-Options und http Strict Transport Security (HSTS) oder Content Security Policy (CSP). Das Kommunikationsdiagramm ermöglicht das Modellieren der Kommunikation zwischen Rollen des Sicherheitssystemes: *„deny“*, *„nosniff“* und *„default-src ‚self‘“*. Hierbei handelt es sich um Response-Header zum Verhindern des Web-Angriffes. Das modellieren mithilfe des Kommunikationsdiagramms fokussiert auf die Anwendung der Header in der Entwicklung der Webanwendung. Der erste Header *„X-Frame-Option“* verhindert das klingende *„Clickjacking“* des Angriffes und verfügt über den Wert *„DENY“*, welche verhindert, dass die Seite nicht in einem Frame angezeigt werden. Die zweite Header-Response X-Content-Type-Options enthält ausschließlich den Wert *„nosniff“*, der verhindert, dass die Browser wie z. B. Internet Explorer oder Google Chrome den MIME-Type der Response selbst zu ermitteln versuchen. Der letzte Header, genannt HTSHFilter, schützt mithilfe von der Content Security Policy (CSP) moderne Browser vor Cross-Site Scripting (XSS). CSP verfügt über den Wert *„default-src ‚self‘“* [10].

Das Kommunikationsdiagramm ist mithilfe des Frameworks Papyrus über *„New->New Project“*, wie auf der Abb. 3.62 zu sehen ist, erstellt. Der Projektname ist nach *„Web-Anwendung“* genannt. Auf der Abb. 3.63 ist das Auswählen des Diagrammtyps *„Communication Diagram“* zu ermöglichen. Abb. 3.64 zeigt ein leeres Arbeitsblatt und den Werkzeugkasten, genannt Palette, mit Nodes und Edges. Das Erste verfügt

über Lebenslinien, Kommentare oder Zeitbeobachtung, während Edges über Nachrichte und Link verfügen. Der Name des Diagramms ist *„Security4login“*. Das Hinzufügen von Lebenslinien zeigt Abb. 3.65. Abb. 3.66, 3.67, 3.68 und 3.69 geben Überblicke über Rollen der Lebenslinien und deren Interaktionen. Rollen bei den Lebenslinien verdeutlichen den Einsatz gegen Web-Angriffe, wobei diese Rollen in dem Sicherheitssystem zusammenwirken. Nachrichten *„doFilter“*, *„init“* und *„destroy“* sind zwischen Lebenslinien übergeben. Auf den Abb. 3.66, 3.67, 3.68 und 3.69 liegt der Fokus auf der Struktur der Nachrichten *„doFilter“*, *„init“* und *„destroy“*, die zwischen Rollen in der Interaktion übergeben werden. doFilter-Methoden stellen eine Implementierung des Design Patterns Delegate dar, wobei das Interface *Filter* ein Teil des Pakets *„javax.servlet“* ist. Gemäß Abb. 3.66, 3.67, 3.68 und 3.69 spielt das Interface *Filter* eine wichtige Rolle als Sieb in der *doFilter*-Methode. Das Interface Filter ist in dem Verteilung-Deskriptor der Web-Anwendung konfiguriert. Die *init*-Methode stellt einen Zugang des Interfaces *Filter* zu *„FilterConfig“*-Objekten wie z. B. X-Frame-Option oder HTSHFilter zum Initialisieren des Prozesses der Filterung mithilfe von *ServletContext* dar. Die letzte Methode *destroy()* ist vom Web-Container aufgerufen, um dem Interface *Filter* das Ende seines Dienstes während des Prozesses der Web-Sicherheitsanwendungen zu benachrichtigen. Diese drei Methoden stellen Schutzmaßnahmen gegen Angriffe auf die Webanwendungen dar.

Lebenslinien mit derer Dreiecken auf den Abb. 3.66, 3.67, 3.68 und 3.69 zeigen Rollen oder Objekte zur Gestaltung der Web_Sicherheitsanwendungen. Die Linien zwischen Klasseninstanzen stellen die Verbindungen zwischen verschiedenen Komponenten der Websicherheitsanwendungen dar. Die Pfeile stehen für gesendete Nachrichten zwischen Objekten oder Rollen.

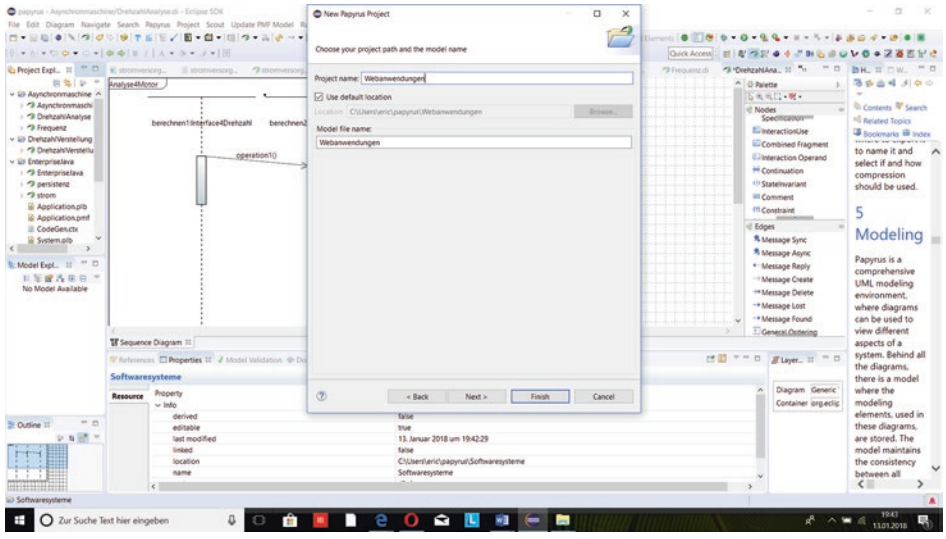


Abb. 3.62 Das Erstellen des Projektes *„Web-Anwendungen“* mit dem Framework Eclipse-Papyrus

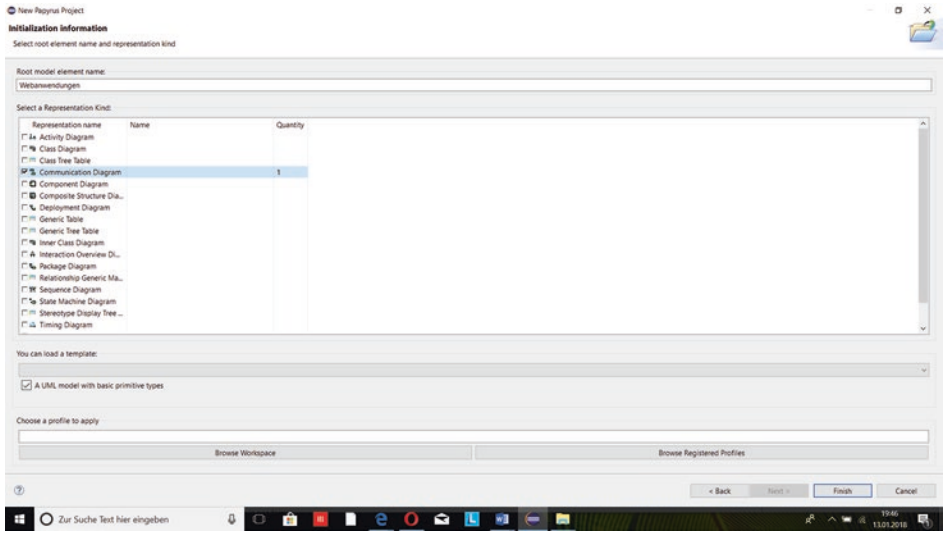


Abb. 3.63 Das Auswählen des Diagrammtyps *„Communication Diagram“*

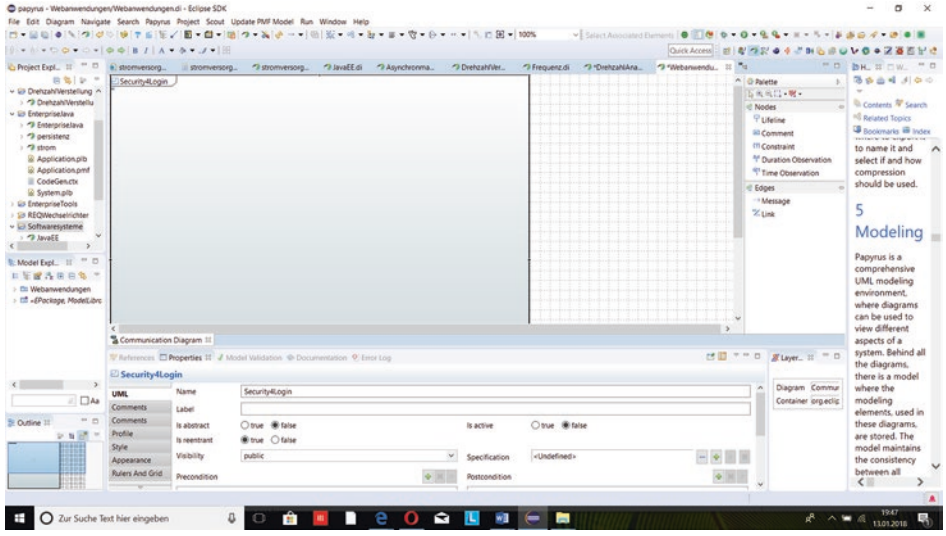


Abb. 3.64 Editor des Kommunikationsdiagramms vom Framework Eclipse-Papyrus

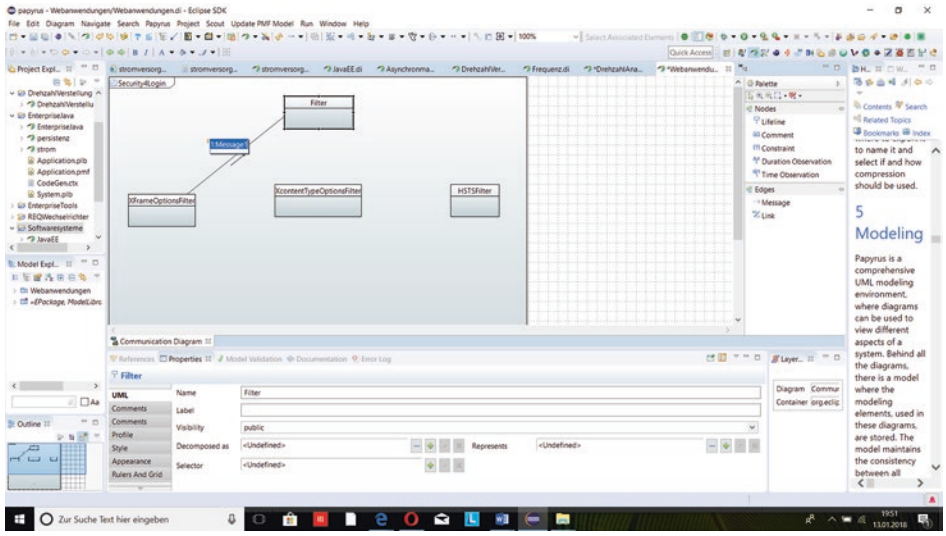


Abb. 3.65 Überblick über Lebenslinien mit Objekten (Rollen) und Klassen

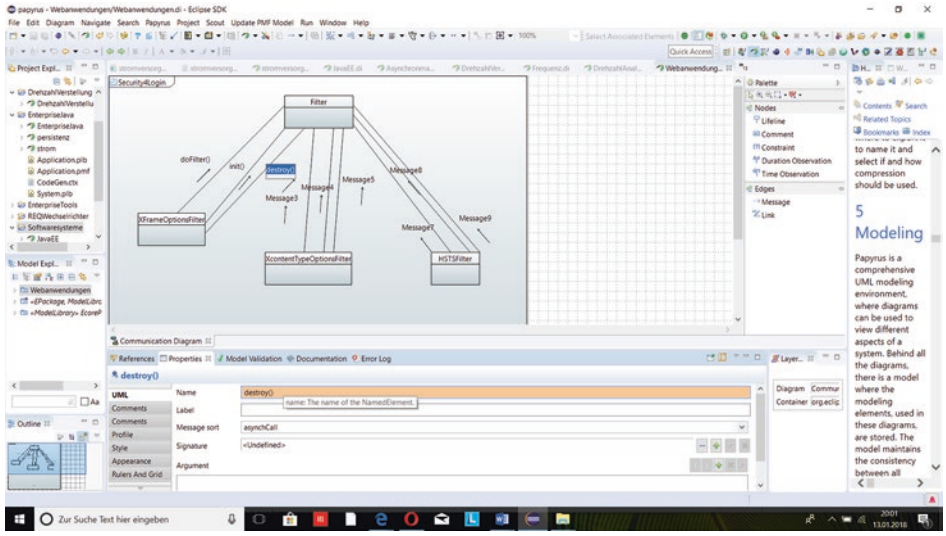


Abb. 3.66 Hinzufügen der Nachrichte als Methode für die Interaktionen zwischen Objekten oder Rollen

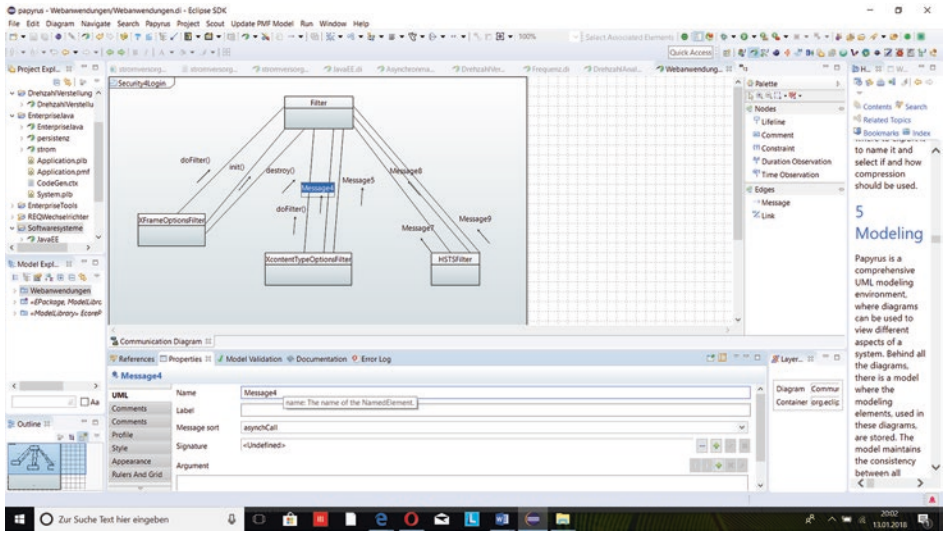


Abb. 3.67 Überblick über die Nachricht *„Message4“* zwischen Lebenslinien *„Filter“* und *„XContentTypeOptionsFilter“*

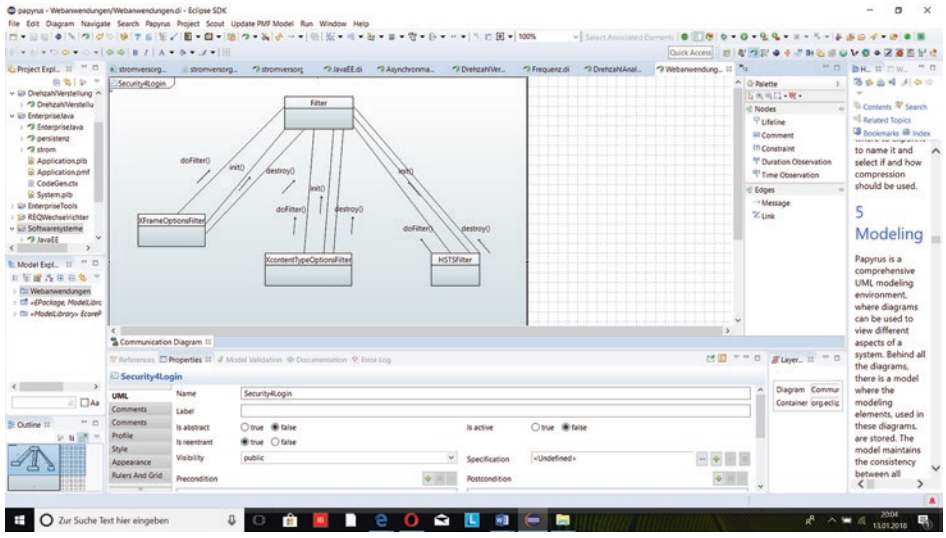


Abb. 3.68 Darstellung des Kommunikationsdiagramms mit Hinblick auf Interaktionen zwischen dem Interface *„Filter“* und seinen Vererbungsklassen

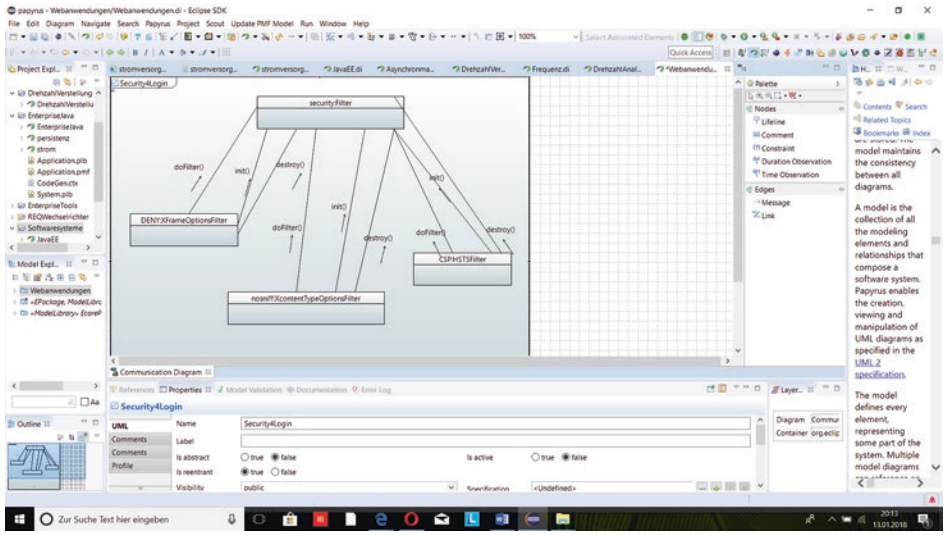


Abb. 3.69 Darstellung der Interaktionen zwischen dem Objekt *„security“* und den drei Objekten *„DENY“*, *„nosniff“* und *„CSP“* mithilfe der Methoden

3.6 Objektdiagramme mit Eclipse-Papyrus

Objektdiagramme ermöglichen die Modellierung der Objektstruktur. Ein Objektdiagramm ist ein Klassendiagramm mit dem Ziel, die Darstellung der Objekte bezüglich der Struktur der Datentype zu realisieren. Hierbei werden Objekte nach ihren Datentypen spezifiziert. Das Objektdiagramm gibt Überblicke über die Werte der Objekte in Bezug auf die statische Struktur des Diagramms. Mithilfe eines Objektdiagramms wird das Analysemodell konkret, präzise und einfach. Ein Objektdiagramm stellt eine Klassifikation der Objekte mithilfe von Attribute, Operationen und Beziehungen dar [11].

Ein Objektdiagramm beschreibt eine Sammlung von Objekten und deren Beziehungen zu einem Zeitpunkt im Leben eines Systems. Objektdiagramme sind daher auf Instanzen fokussiert und haben einen exemplarischen Charakter. Ein Objekt ist Instanz einer Klasse und enthält die Attribute, welche mit einem Wert initialisiert sind. Im Objektdiagramm werden prototypische Objekte verwendet, um exemplarische Situationen zu illustrieren. Zwischen den im Diagramm sichtbaren prototypischen Objekten und den echten Objekten des Systems besteht normalerweise keine 1:1-Beziehung. Ein Attribut beschreibt eine Zustandskomponente eines Objektes. Ein Attribut im Objektdiagramm ist charakterisiert durch den Attributnamen, den Typ und einen konkreten Wert. Weitere Charakteristika, wie Sichtbarkeit, können dem Attribut angeheftet werden. In abstrakten Objektdiagrammen können statt konkreten Werten auch Variablennamen oder Ausdrücke eingesetzt werden, deren Inhalt im Diagramm *„unspezifiziert“* bleibt. Attributtyp oder -wert können auch ausgelassen werden [8].

Ein Objektdiagramm kann eine Spezialisierung eines Klassendiagramms bezüglich der Konkretisierung des Inhaltes des Datentyps zum einen Zeitpunkt darstellen, wobei Objektediagramme die Untergruppen der Elemente der Klassendiagramme zum Verdeutlichen die Beziehungen zwischen Instanzen von Klassen verwenden. Obwohl die Objektediagramme nicht neue architekturelle Konzepte als Klassendiagramme zeigen, sind sie beim Verstehen der Funktionalitäten der Klassendiagramme hilfreich [12].

3.6.1 Erstellen eines Klassendiagrammes mit Eclipse-Papyrus

Klassen bestehen aus Attributen und Methoden, die den Zustand und das Verhalten ihrer Instanzen (Objekte) festlegen. Klassen sind durch Assoziationen und Vererbungsbeziehungen miteinander verknüpft. Klassennamen erlauben es, die Klassen zu identifizieren [8]. Hierbei stellen Attribute mithilfe der Namen und Typen die Zustandskomponente der Klassen dar. Die Klassenmethoden implementieren mithilfe der Signaturen und Rumpf die Funktionalitäten der Klassen.

Abb. 3.70, 3.71, 3.72, 3.73 und 3.74 zeigen das Erstellen des Klassendiagramms, genannt *„Object“*, im Projekt *„Verlust4Transistor“*. Abb. 3.70 gibt einen Überblick über die Auswahl der Architektur zur Modellierung, wobei entweder eine Architektur für UML oder für SysML zur Auswahl steht. Anschließen wird mithilfe von Abb. 3.71, 3.72, 3.73 und 3.74 der Diagrammtyp ausgewählt. Vorher soll das UML-Modell, wie auf den Abb. 3.71 und 3.72 zu sehen ist, genannt werden. Abb. 3.74 zeigt die Vielfalt an UML-Modellierungsdiagrammen. Das Klassendiagramm ist, wie auf der Abb. 3.74, ausgewählt worden. Abb. 3.75, 3.76 und 3.77 stellt die Struktur des Klassendiagrammes mithilfe des Design-Patterns *„Interface“* dar. Hierbei sind zum einen die gesamten Klassen in einer Transistor-Komponente, genannt IGBT *(Insulate Gate Bipolar Transistor),* zusammengefasst und zum anderen Vererbungsbeziehungen zwischen verschieden Klassen mithilfe des Design-Patterns modelliert. Abb. 3.75 stellt das Klassendiagramm mit der Implementierung des Interface, genannt *„Schaltverlust“*. Die Vererbungsbeziehungen stehen zwischen dem Interface *„Schaltverlust“* und den Klassen *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“*. Die Haupt-Klasse *Berechnung* verfügt über zum einen die *main*-Methode und zum andren über Aufrufe der Klassen *Einschaltverlustleistung* und *Ausschaltverlustleistung,* welche das Interface *Schaltverlust* implementieren. Das Interface *Schaltverlust* ist wie die beiden Klassen *Ausschaltverlustleistung* und *Einschaltverlustleistung* durch ein Rechteck dargestellt und mit dem Stereotyp ≪interface≫ bezeichnet.

Abb. 3.74 und 3.75 zeigen die Festlegung der Sichtbarkeit, Instanziieren und Veränderung des modifizierten Elemente mit der Anwendung des Modifikators *„public“* sowohl auf Eigenschaften des Interfaces *„Schaltverlust“* als auch auf diese der Klassen *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“*. Auf der Abb. 3.75 ist ein

Rechteck vor dem grafischen Modifikator „+“ zu sehen. Als Test-Datentype Double bzw. Integer sind Zahlen wie z. B. *0.00125, 0.03125* oder *0.0008* bzw. *25 oder 20* angegeben worden.

Der Editor des Frameworks Papyrus zeigt wie auf den Abb. 3.75, 3.76 und 3.77 im dritten Teil der Klassenschicht Interface- bzw. Klassenmethoden wie z. B. *„ein_verlustarbeit()“* oder *„aus_verlustarbeit()“* bzw. *„ein_verlustErmitteln()“* oder *„aus_verlustErmitteln()“* mit Namen, Signaturen und Plus-Zeichen für einen öffentlichen Modifikator dargestellt ist. Im Vergleich zu den Attributen, welche den Zustand eines Objektes speichern, ermöglichen Methoden wie z. B. *„ein_verlustErmitteln()“* oder *„aus_verlustErmitteln()“* dazu, bestimmte Funktionen wie z. B. Analysieren zu realisieren und Daten zu berechnen. Außerdem verwenden die Klassen-Methoden die gespeicherten Daten der Attribute und rufen andere Methoden des Objektes wie z. B. *„ein_verlustleistung()“* oder *„aus_verlustleistung()“* vom Interface *„Schaltverlust“* auf. Die Zugriffsrechte für o. g. Methoden, wie es auf den Abb. 3.75, 3.76 und 3.77 zu sehen ist, sind analog zu den Sichtbarkeiten für Attribute mit „+“ gesteuert worden.

Das Klassendiagramm auf den Abb. 3.75, 3.76 und 3.77 soll mithilfe des Design-Patterns Interface die Verluste des Transistors IGBT zum einen mithilfe der objektorientierten Modellierung bezüglich der Vererbung zwischen dem Interface „Schaltverluste“ und den implementierten Klassen *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* ermitteln und zum anderen mithilfe des Hauptprogramms *„Berechnung“* berechnen.

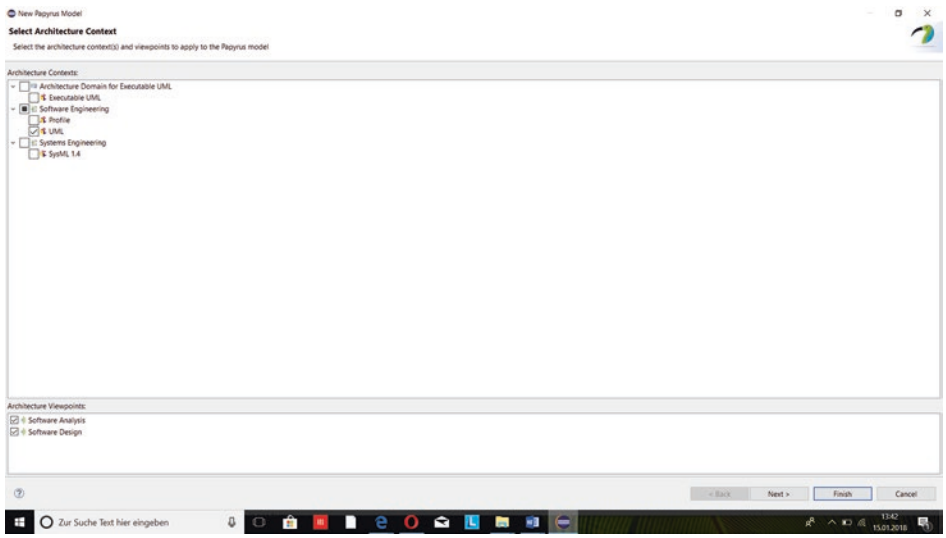


Abb. 3.70 Auswählen der UML-Architektur zur Modellierung

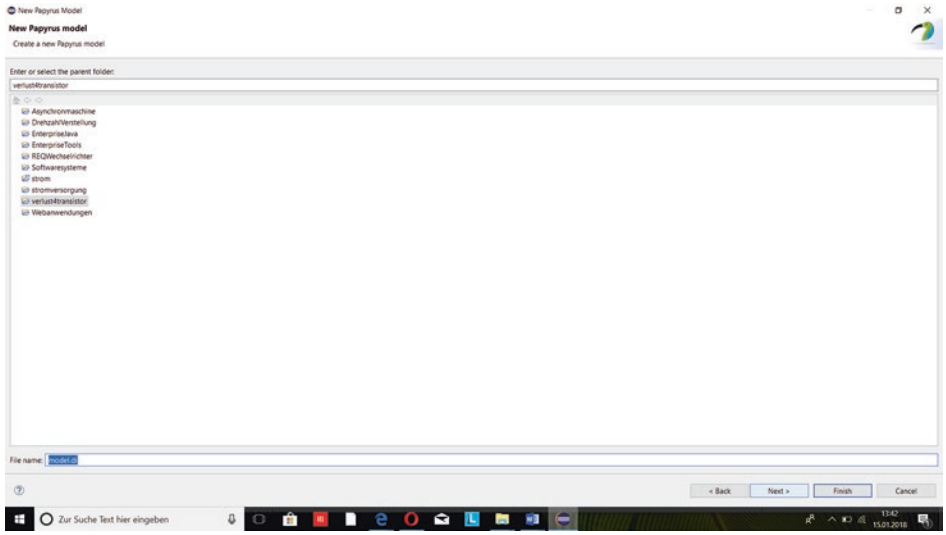


Abb. 3.71 Das Erstellen eines Modells mit Framework Papyrus im Projekt *„verlust4transistor“*

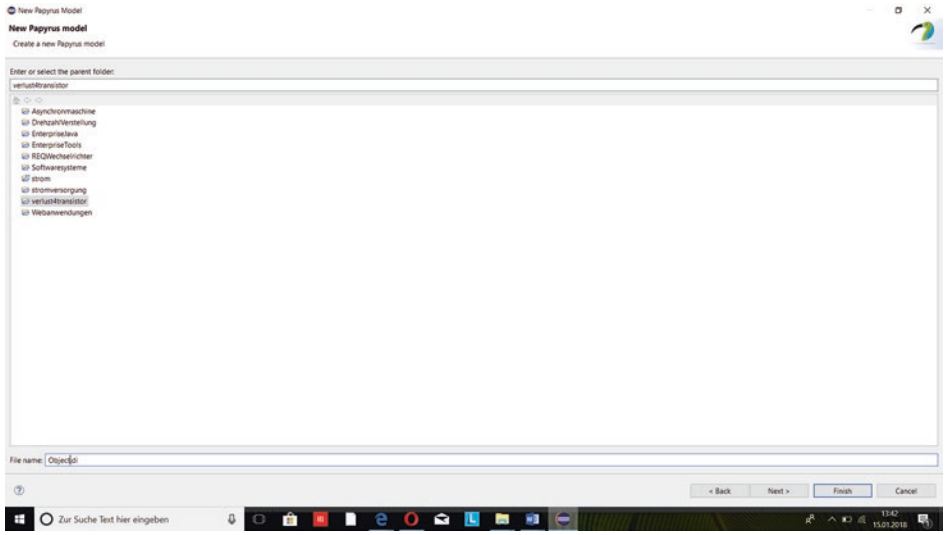


Abb. 3.72 Das Erstellen des Modells genannt *„Object“* im Projekt *„verlust4transistor“*

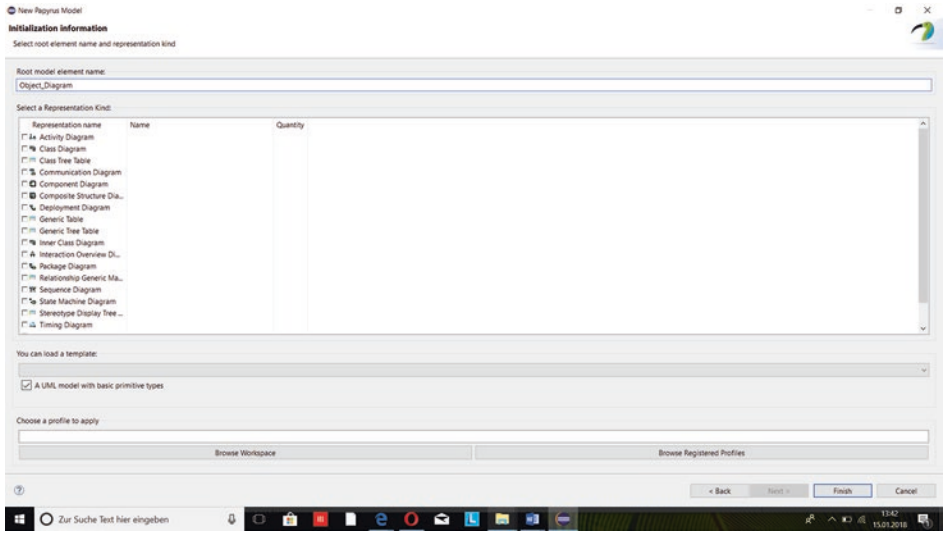


Abb. 3.73 Überblick über verschiedene Diagrammtypen für die UML-Modellierung mit Eclipse-Papyrus

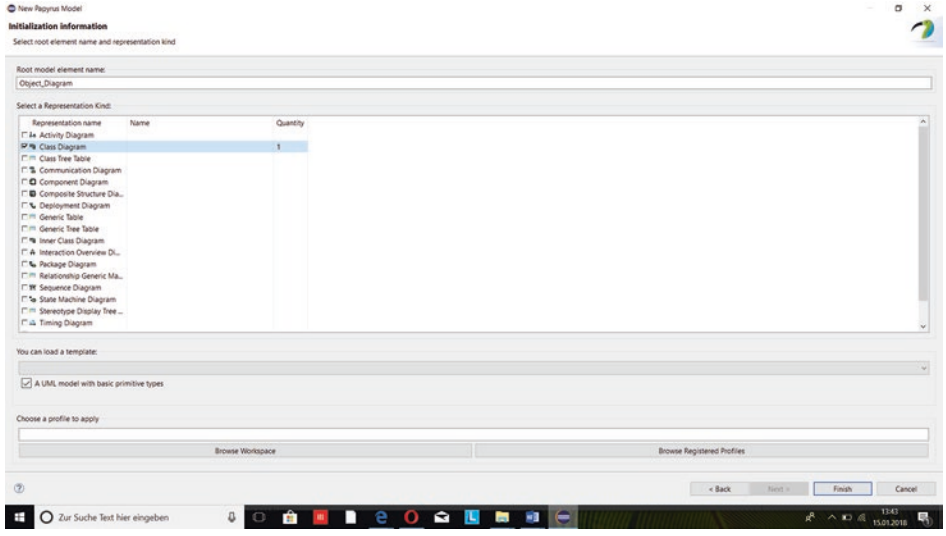


Abb. 3.74 Auswahl des Diagrammtyps *„Class Diagramm“* zur UML-Modellierung mit Eclipse-Papyrus

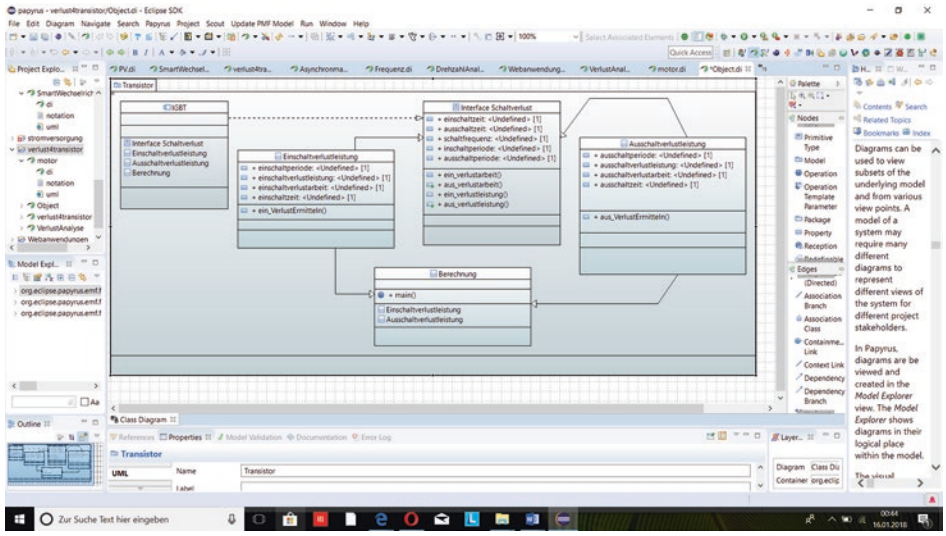


Abb. 3.75 Darstellungen der Komponente, des Interfaces und der Klassen für das Klassendiagramm im Modell *„Transistor“*

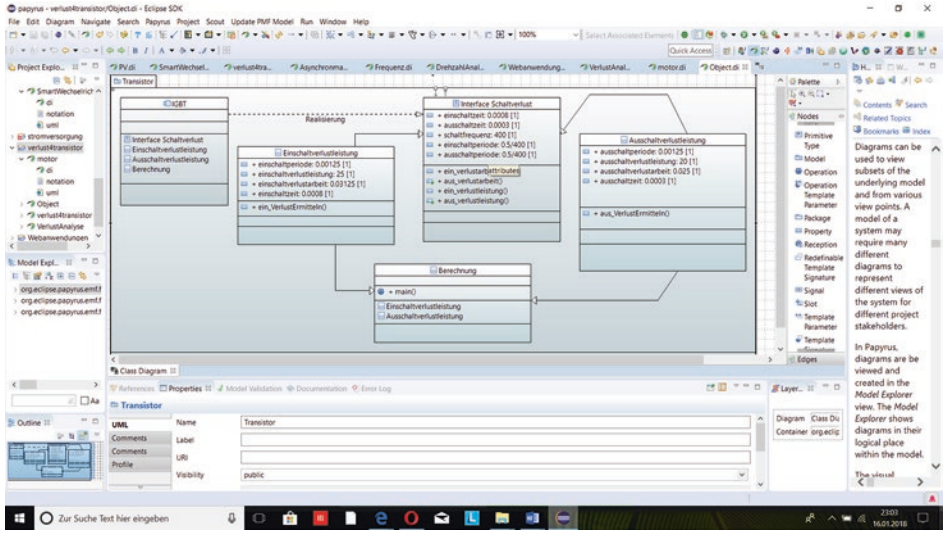


Abb. 3.76 Das Testen der Funktionalität des Klassendiagramms mithilfe von Werten

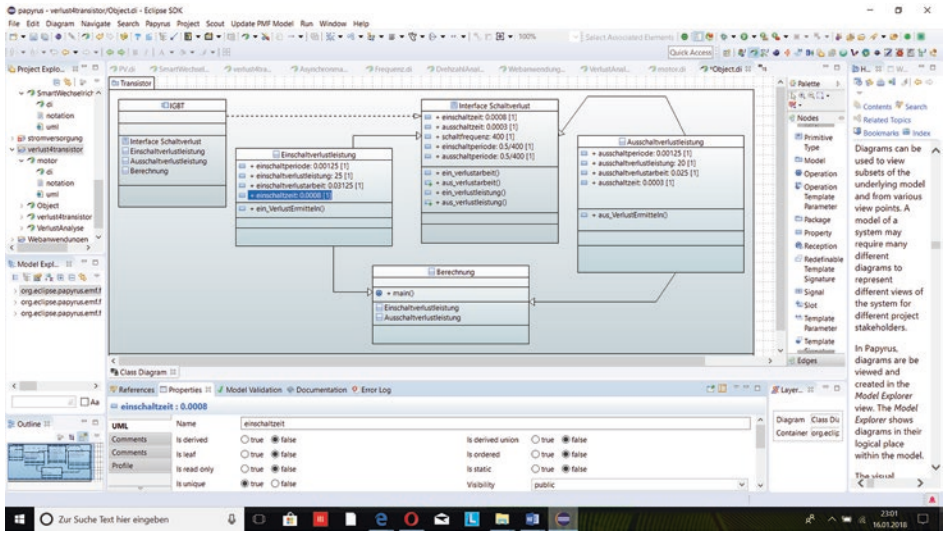


Abb. 3.77 Überblick über das Klassendiagramm im Hinblick auf hinzugefügte Werte in die Klassen

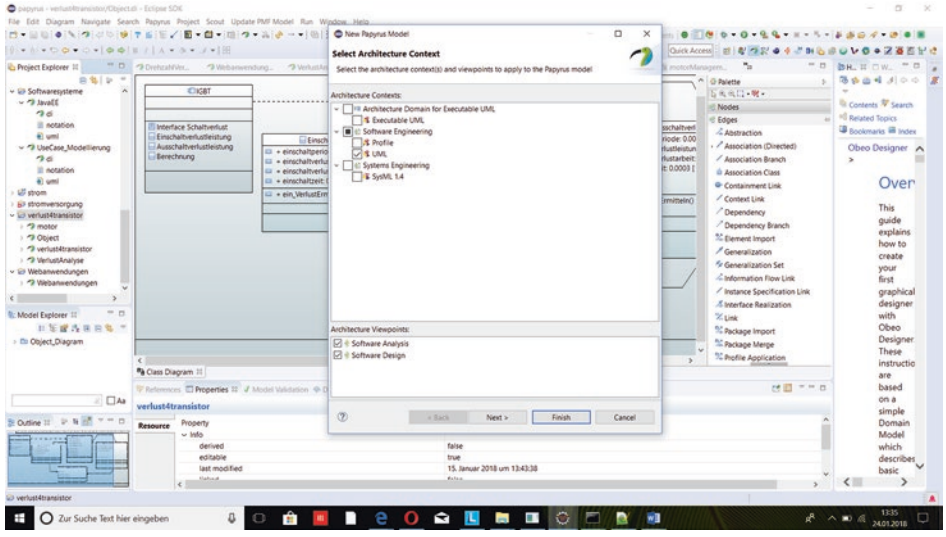


Abb. 3.78 Auswahl der UML-Architektur für Software-Engineering

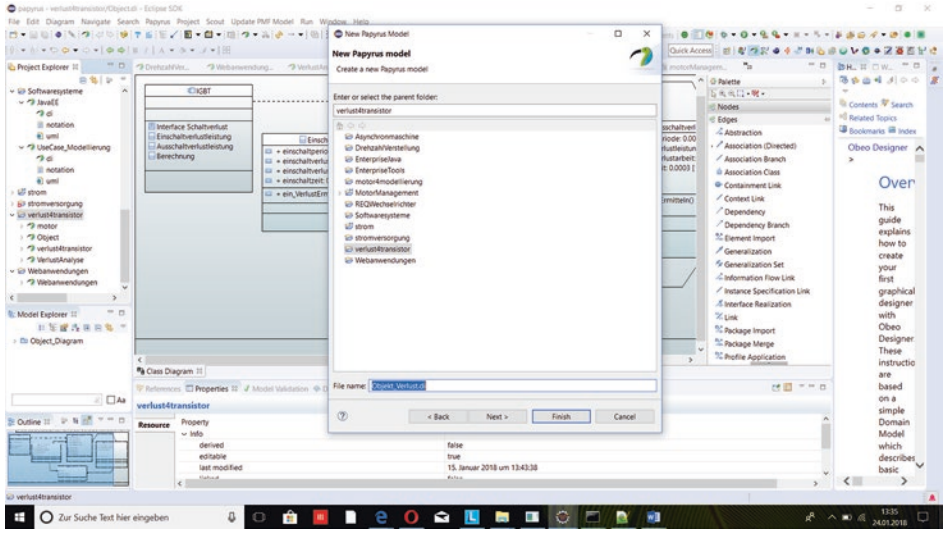


Abb. 3.79 Das Erstellen des Modells *„Objekt_Verlust“* im Projekt *„verlust4transistor2“* mit Eclipse-Papyrus

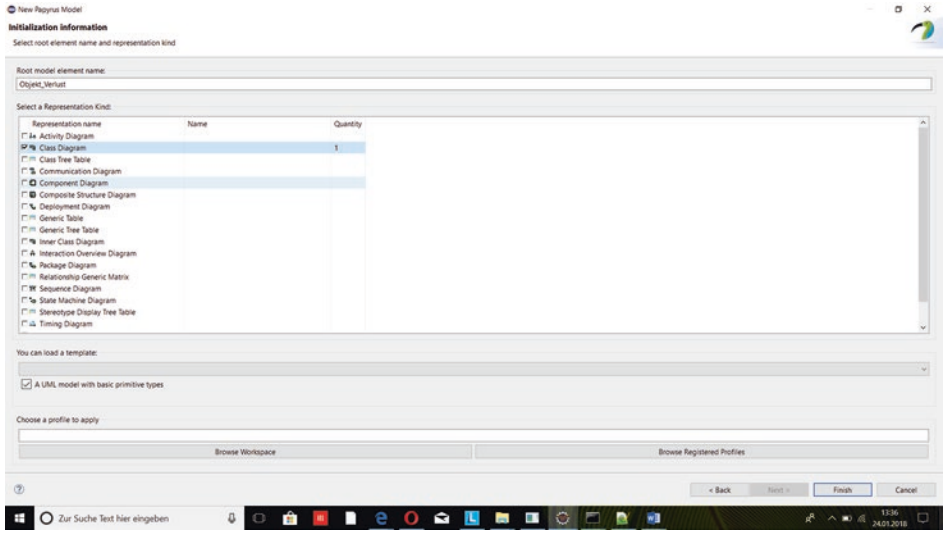


Abb. 3.80 Auswahl des Diagrammtyps *„Class Diagram“* zur UML-Modellierung mit Eclipse-Papyrus

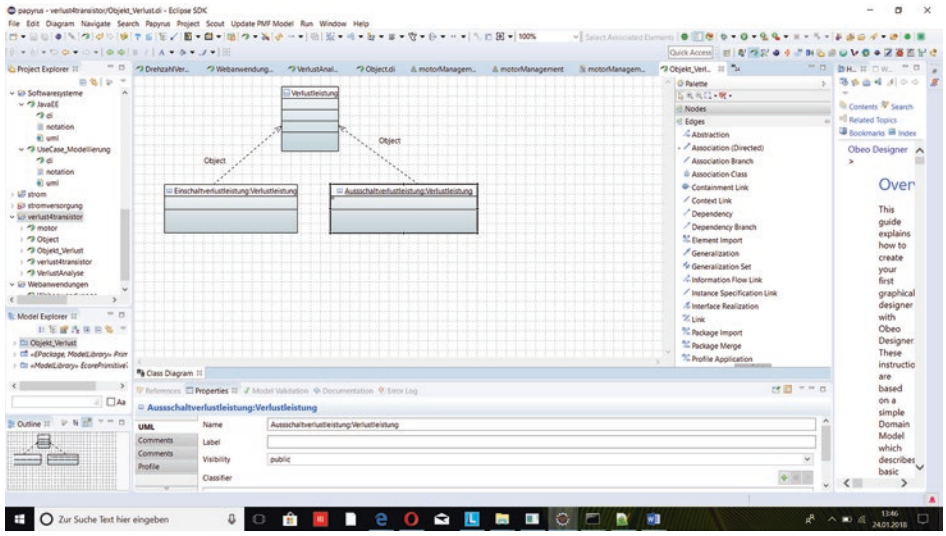


Abb. 3.81 Das Erstellen eines Objektdiagramms mithilfe der Instanzbeschreibung

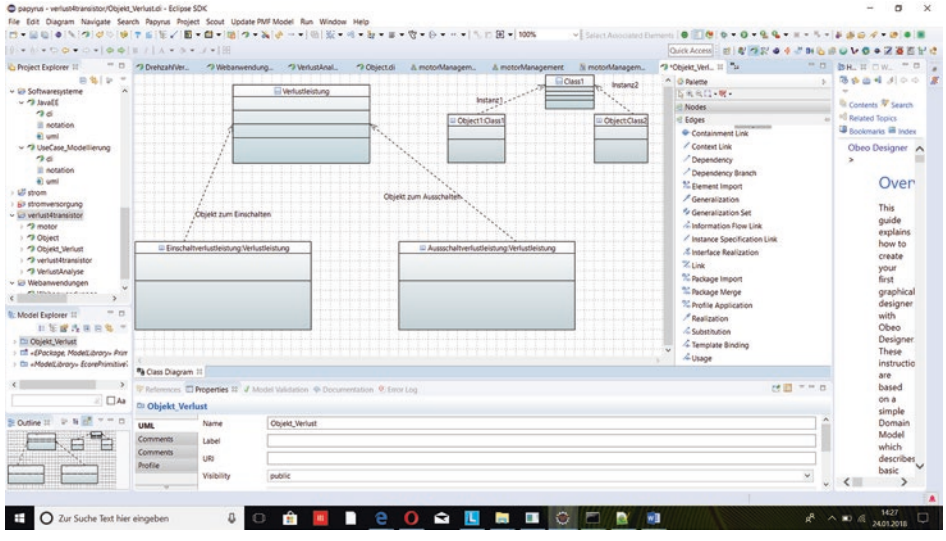


Abb. 3.82 Überblick über Abhängigkeitsbeziehungen zwischen einer Klasse und ihren Objekten

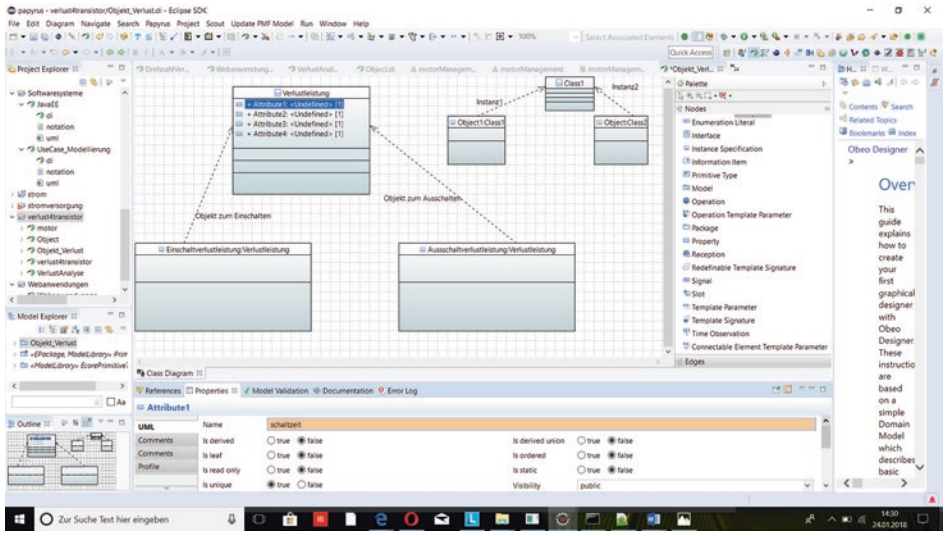


Abb. 3.83 Interaktionen zwischen der Klasse *Class1* und zwei Instanzbeschreibungen

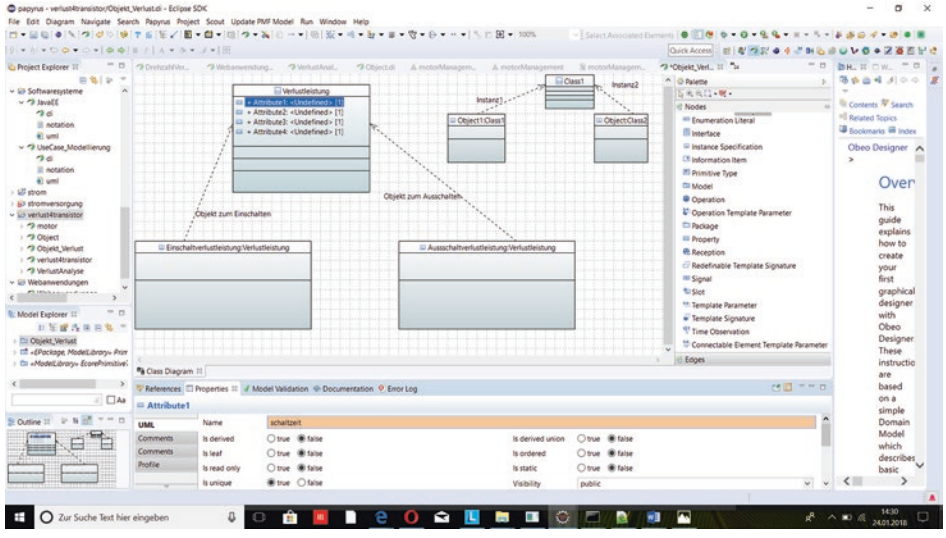


Abb. 3.84 Struktur von UML-konformer Notation der Instanzbeschreibungen

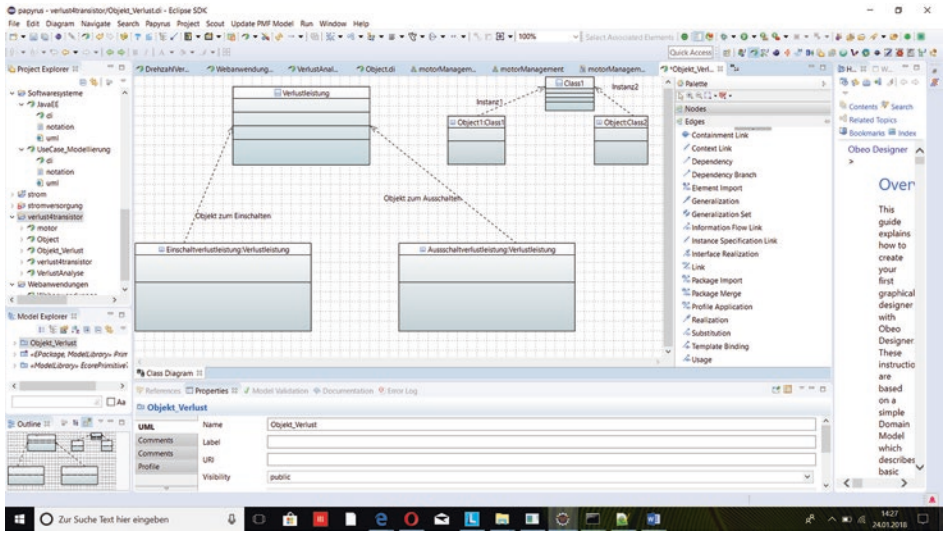


Abb. 3.85 Aufbau der Eigenschaften der Klasse *„Verlustleistung“* im Hinblick auf Attribute

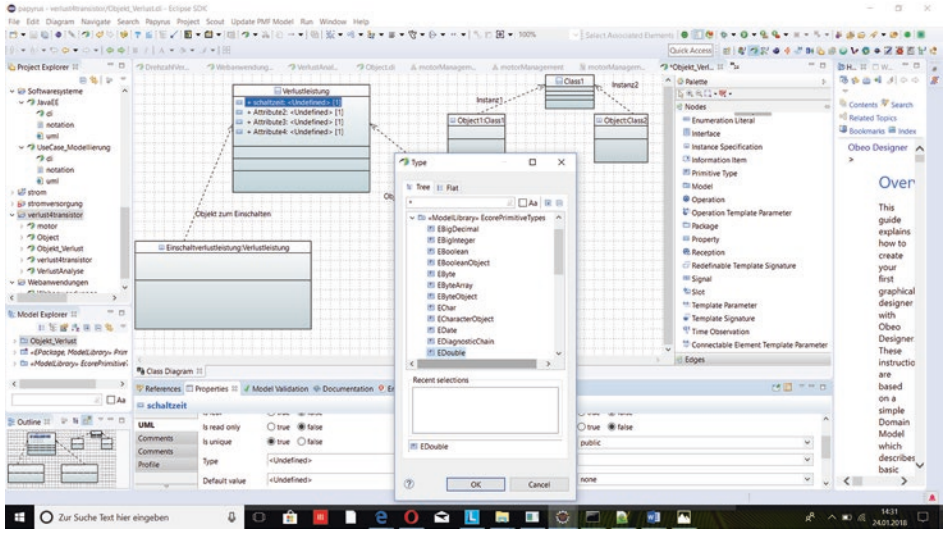


Abb. 3.86 Das Auswählen des Datentyps *„Edouble“* für das Attribut *„schaltzeit“* mithilfe von *„ModelLibrary“ EcorePrimitiveType*

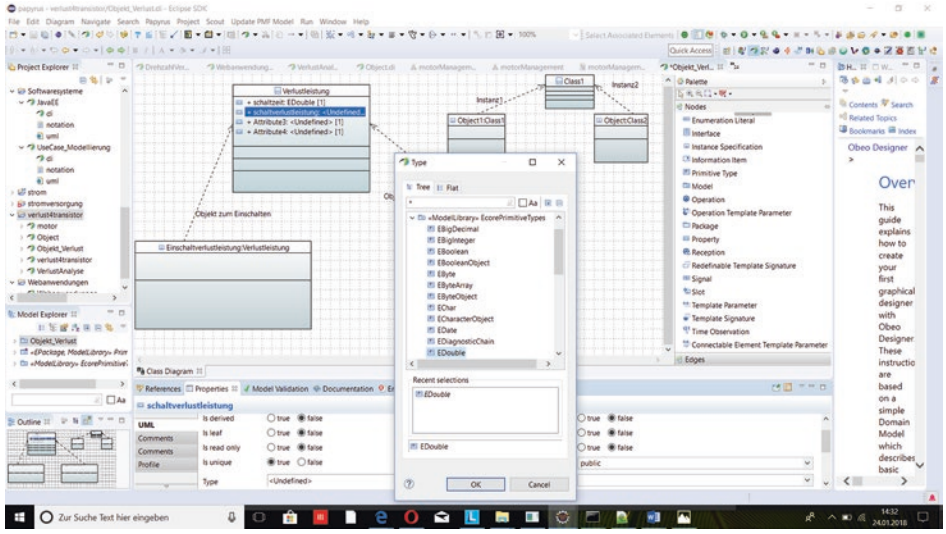


Abb. 3.87 Das Auswählen des Datentyps *„Edouble“* für das Attribut *„schaltverlustleistung“* mithilfe von *„ModelLibrary“ EcorePrimitiveType*

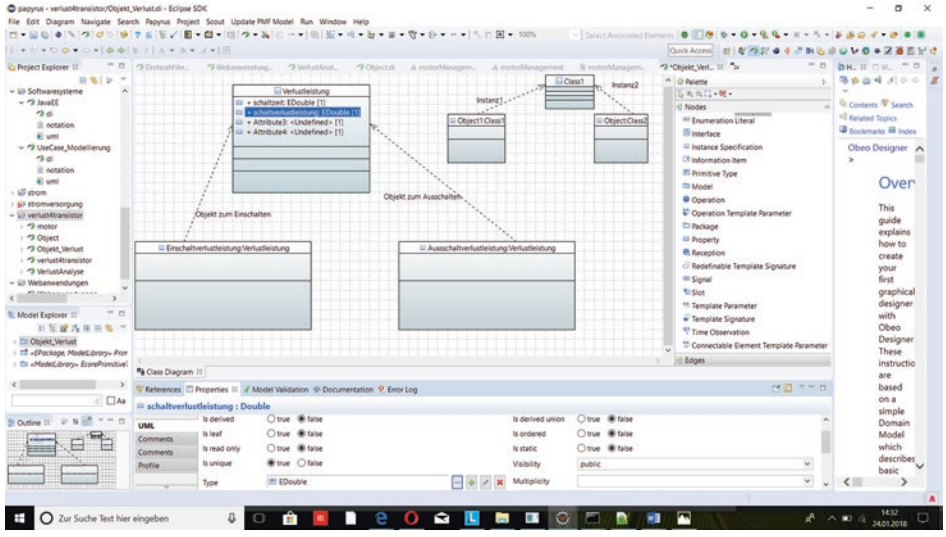


Abb. 3.88 Überblick über das Attribut *„schaltverlustleistung“* im Hinblick auf seinen Datentyp

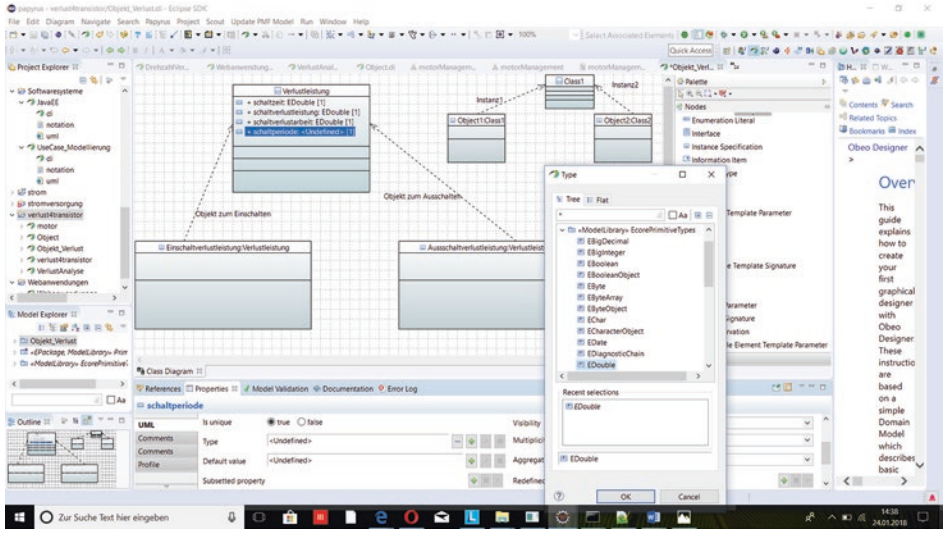


Abb. 3.89 Das Auswählen des Datentypes *„Edouble“* für das Attribut *„schaltperiode“* mithilfe von *„ModelLibrary“* EcorePrimitiveType

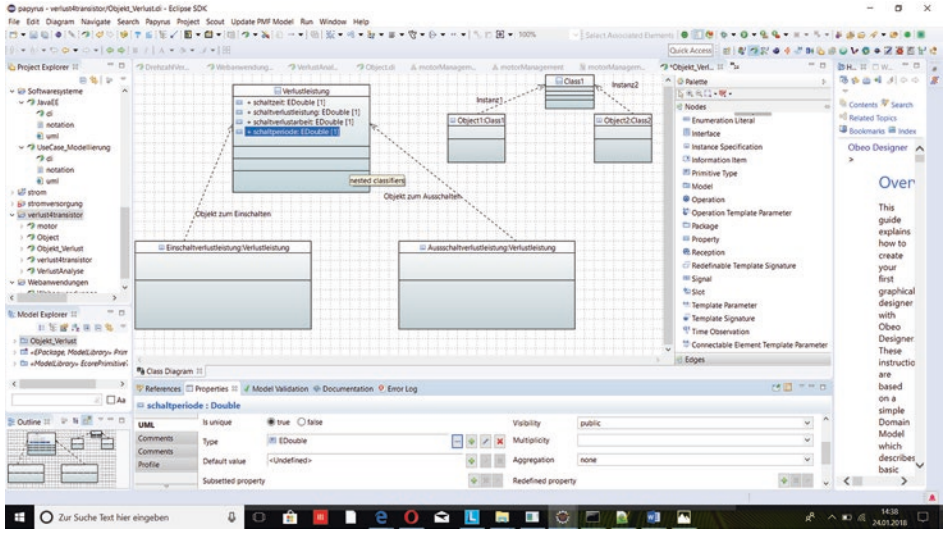


Abb. 3.90 Überblick über das Attribut *„schaltperiode“* im Hinblick auf seinen Datentyp

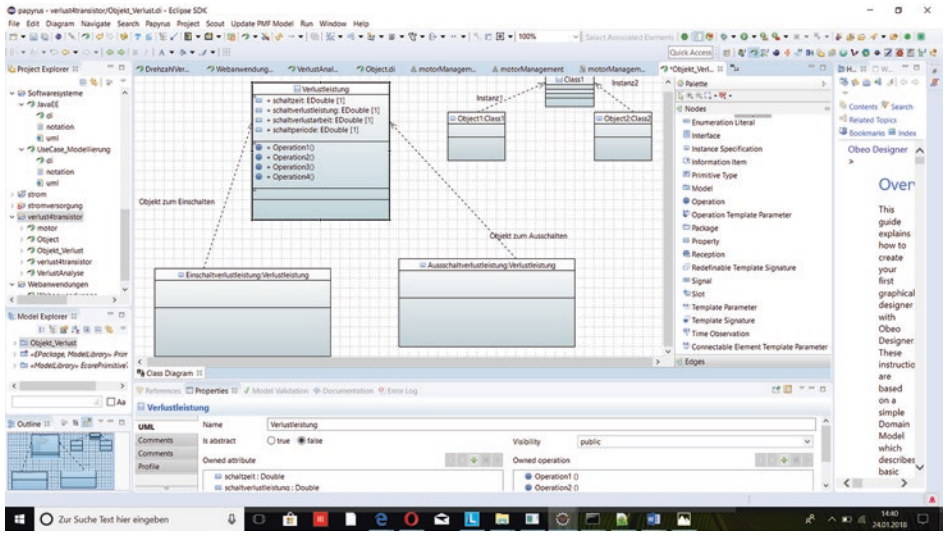


Abb. 3.91 Das Hinzufügen von vier Operationen in die Klasse *„Verlustleistung“*

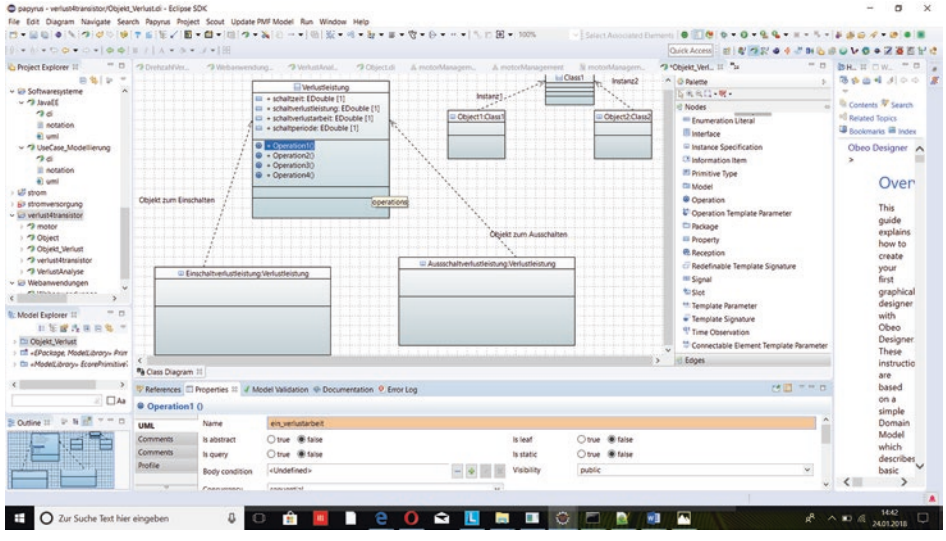


Abb. 3.92 Aufbau der Eigenschaften der Klasse *„Verlustleistung“* im Hinblick auf die Operation *„ein_verlustarbeit()“*

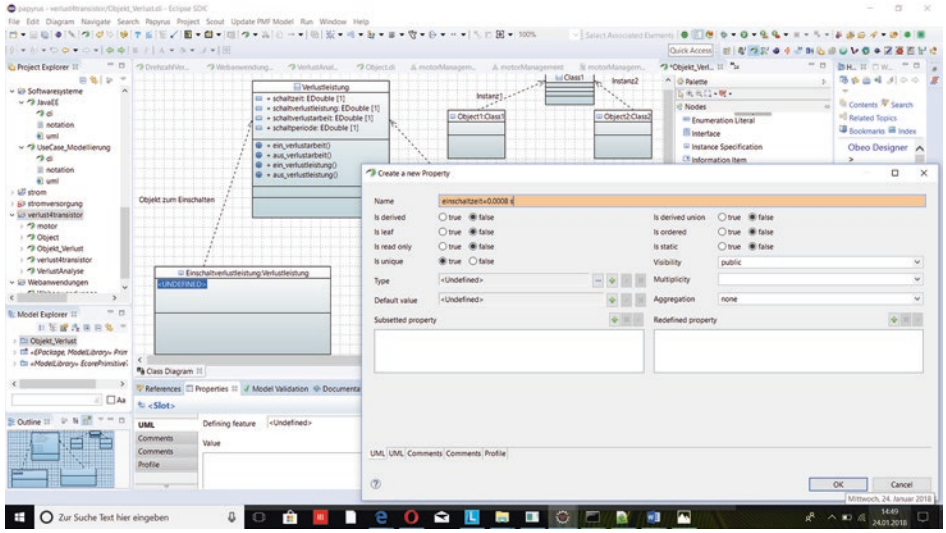


Abb. 3.93 Eingabe des Wertes des Slots *„einschaltzeit“* vom Objekt *„Einschaltverlustleistung“*

3.6.2 Erstellen eines Objektdiagramms mit Eclipse-Papyrus

Bei Objektdiagrammen geht es in erster Linie um die Attribute einer Reihe von Objekten und wie diese zueinander in Beziehung stehen [7]. Abb. 3.78, 3.79, 3.80, 3.81, 3.82, 3.83, 3.84, 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91, 3.92, 3.93, 3.94, 3.95, 3.96, 3.97, 3.98, 3.99, 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 sind mit dem Framework Eclipse-Papyrus erstellt worden und geben Überblicke über das Erstellen des Objektdiagramms, genannt *„Object_Verlust“*, für das Projekt *„verlust4transistor“*. Ziel des Erstellens dieses Objektdiagramms ist es, die Verluste des Transistors mithilfe der Attribute *„einschaltperiode“*, *„einschaltzeit“*, *„einschaltverlustarbeit“*, *„einschaltverlustleistung“* bzw. *„ausschaltperiode“*, *„ausschaltzeit“*, *„ausschaltverlustarbeit“*, *„ausschaltverlustleistung“* der Objekte *„Einschaltverlustleistung“* bzw. *„Ausschaltverlustleistung“* zu bestimmen. Die Kalkulation der Verluste hängt von der Zeit ab. Verlustleistung hängt von der Verlustarbeit ab und diese hängt von der Periode ab. Abb. 3.78, 3.79 und 3.80 zeigen die Schritte zum Erstellen des Papyrus-Modells *„Object_Verlust“* aus einem vorhandenen Projekt namens *„verlust4transistor“*. Anschließend ist der Diagrammtyp ausgewählt. Den Aufbau eines Objektdiagramms zeigen Abb. 3.81, 3.82, 3.83 und 3.84 in Bezug zum einen auf die Instanzen der Klassen und zum anderen auf die Beziehungen zwischen der *„Klasse“* und den *„Objekten“*. Der Editor zeigt zwei Diagramme zum Vergleichen zwischen einem abstrakten und einem konkreten Beispiel. Das Objektdiagramm ist mithilfe des Dreieck-Symbols, genannt *„Instance Specification“*, aus den *Knoten* (Nodes) des Modellierungswerkzeuges oder Palette erstellt. *„Slot“* aus der Knoten der Palette dient als Attribut des Objektes. Instanzenbeschreibung

oder *„Instance Specification“* ist ein Element des Modells zum Darstellen einer Instanz in einem modellierten System. Abb. 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91 und 3.92 geben Überblicke über das Hinzufügen von Attributen mit dem Datentyp *„Double“* u. a. *schaltzeit, schaltperiode, schaltverlustleistung, schaltverlustarbeit* bzw. Operationen wie z. B. *ein_verlustarbeit(), aus_verlustarbeit(), ein_verlustleistung*(), *aus_verlustleistung()* in der Klasse *„Verlustleistung“*. Aus dem Klassendiagramm wird das Objektdiagramm mithilfe des Modellierungstools *„Instance Specification“* erstellt, wie es auf den Abb. 3.82, 3.83, 3.84, 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91, 3.92, 3.93, 3.94, 3.95, 3.96, 3.97, 3.98, 3.99, 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 zu sehen ist. Mit den Abb. 3.82, 3.83, 3.84, 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91, 3.92, 3.93, 3.94, 3.95, 3.96, 3.97, 3.98, 3.99, 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 ist das Testen des Klassendiagramms mithilfe des Objektdiagramms möglich. Nachdem das Klassendiagramm erstellt ist (Abb. 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91 und 3.92), stellt das Erstellen des Objektdiagramms die Attribute wie z. B. *„einschaltzeit“* bzw *„ausschaltzeit“* der Objekte *„Einschaltverlustleistung“* bzw. *„Ausschaltverlustleistung“,* wie es auf den Abb. 3.93 und 3.94.

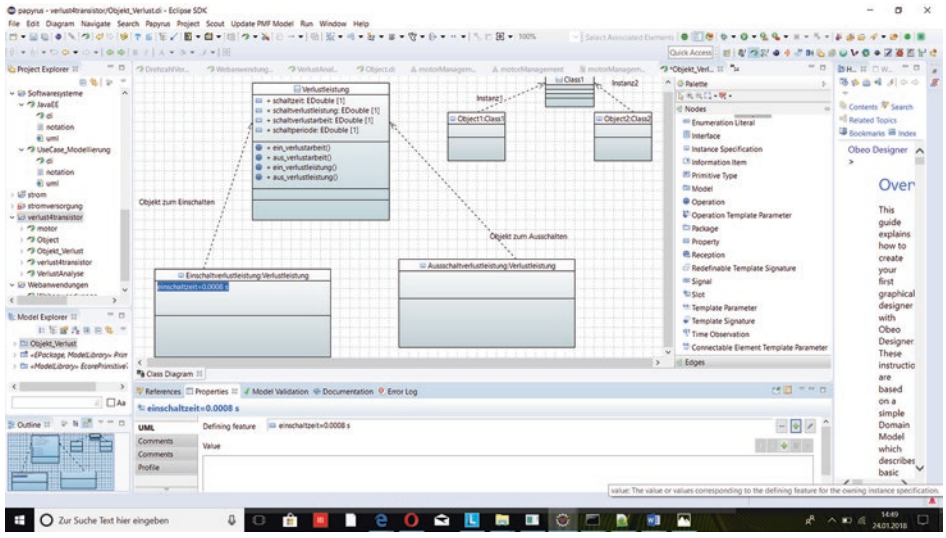


Abb. 3.94 Überblick über den Wert des Slots *„einschaltzeit“* des Objektes *„Einschaltverlustleistung“* bezüglich der Typkonformität

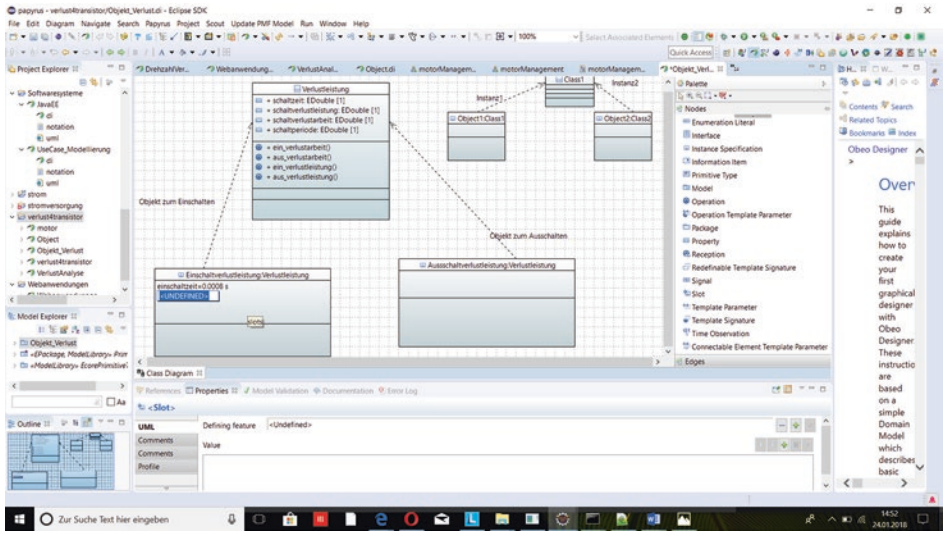


Abb. 3.95 Darstellung eines leeren *Slots* (Wertangabe) der Instanzenbeschreibung

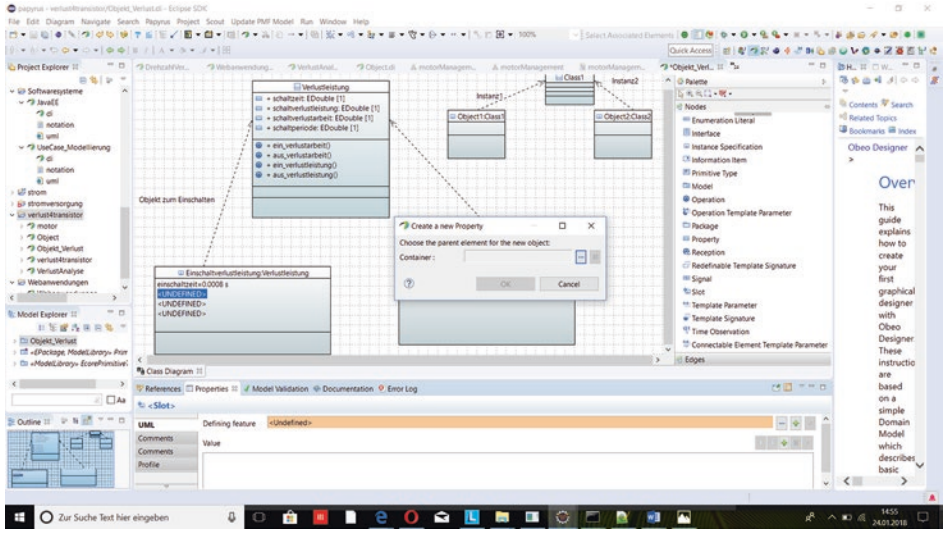


Abb. 3.96 Überblick über das Hinzufügen von Werteangaben (Slots) in das Objekt mit Hinblick auf das Auswählen der Eltern-Klasse *„Verlustleistung“* für das entsprechende Objekt *„Einschaltverlustleistung“*

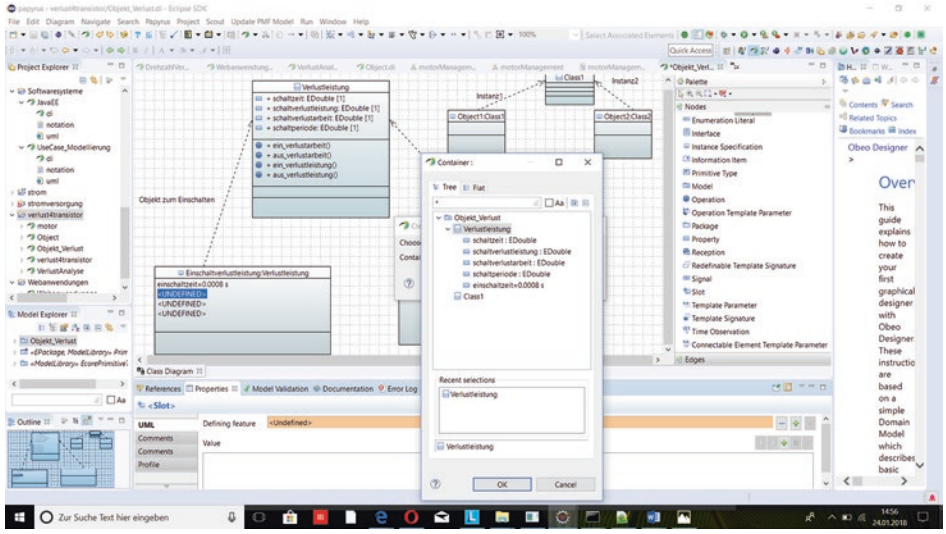


Abb. 3.97 Container zur Darstellung der Instanzen der Klasse „*Verlustleistung*“

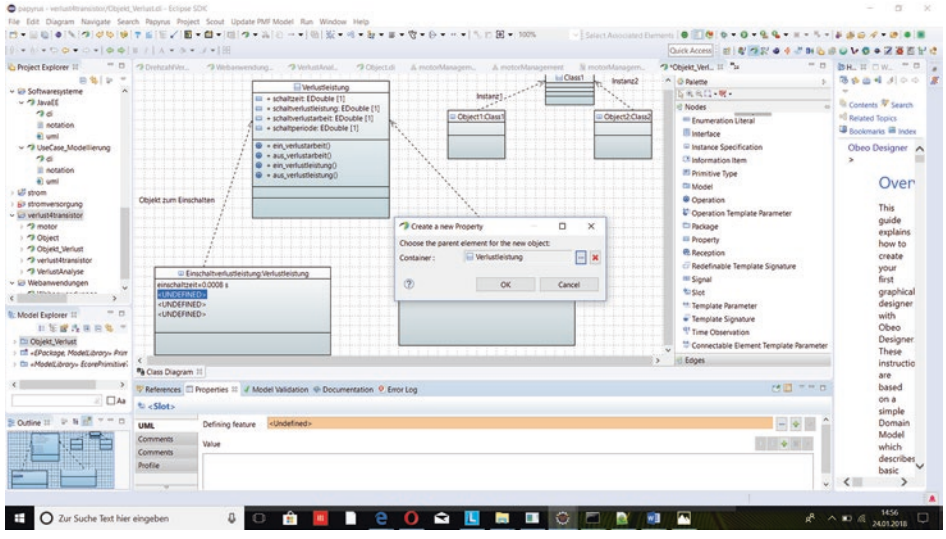


Abb. 3.98 Das Erstellen eines Slots im Container „*Verlustleistung*“

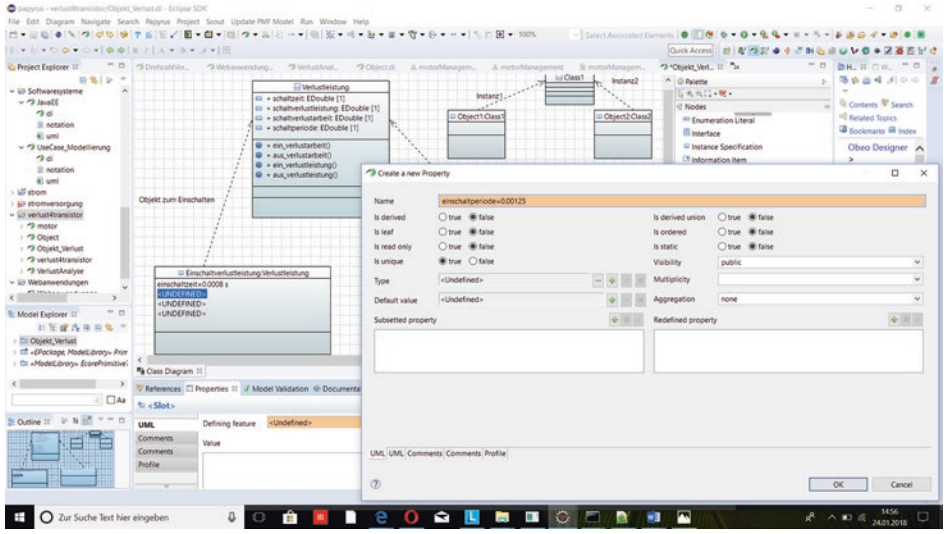


Abb. 3.99 Das Eingaben des Werts *„0.00125“* für den erstellten Slot *„einschaltperiode“*

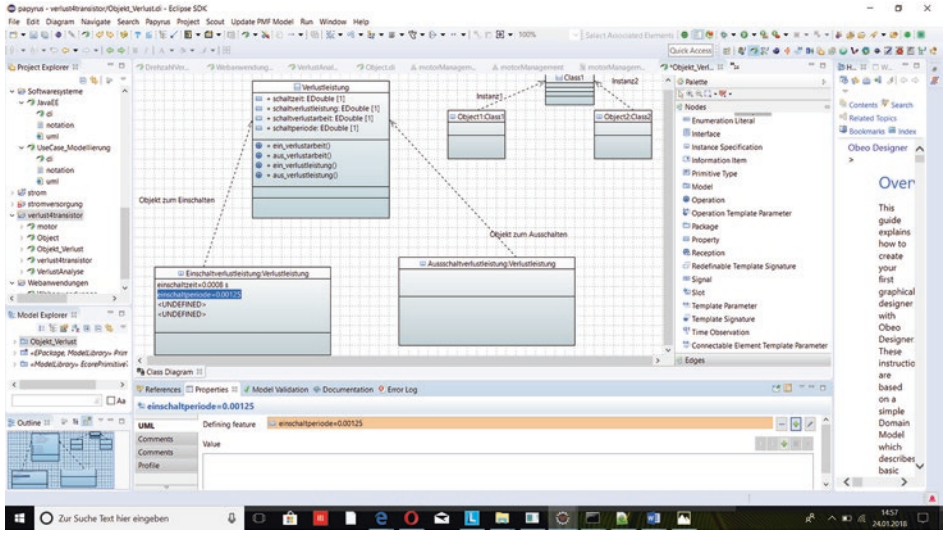


Abb. 3.100 Überblick über den Wert des erstellten Slots *„einschaltperiode“*

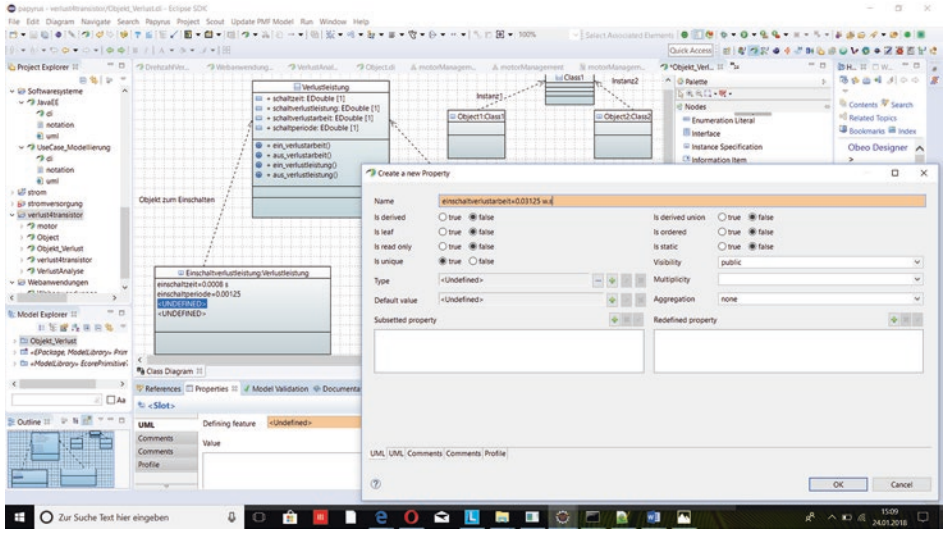


Abb. 3.101 Das Eingeben des Werts *„0.03125 W.s“* für den erstellten Slot *„einschaltverlustarbeit“*

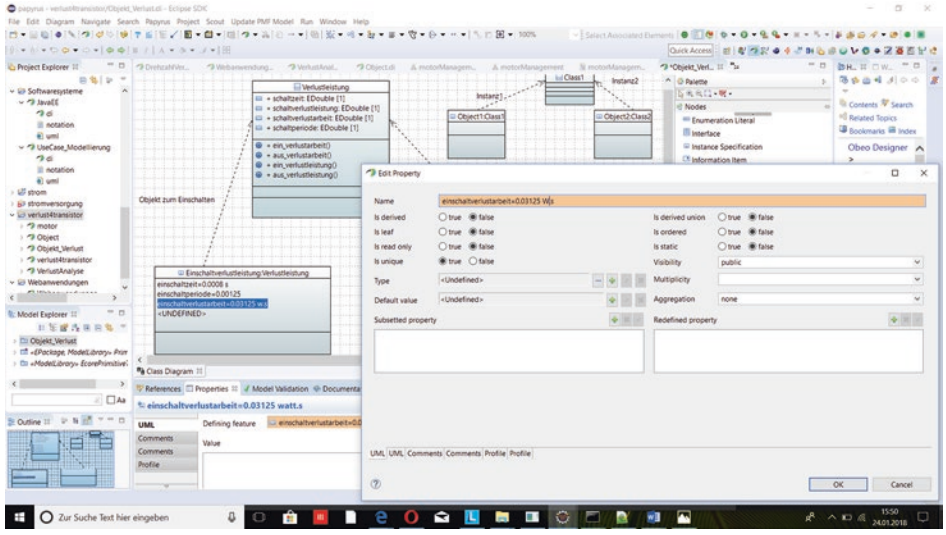


Abb. 3.102 Eingabe des korrigierten Wertes *„0.03125 W.s“* für den erstellten Slot *„einschaltverlustarbeit“*

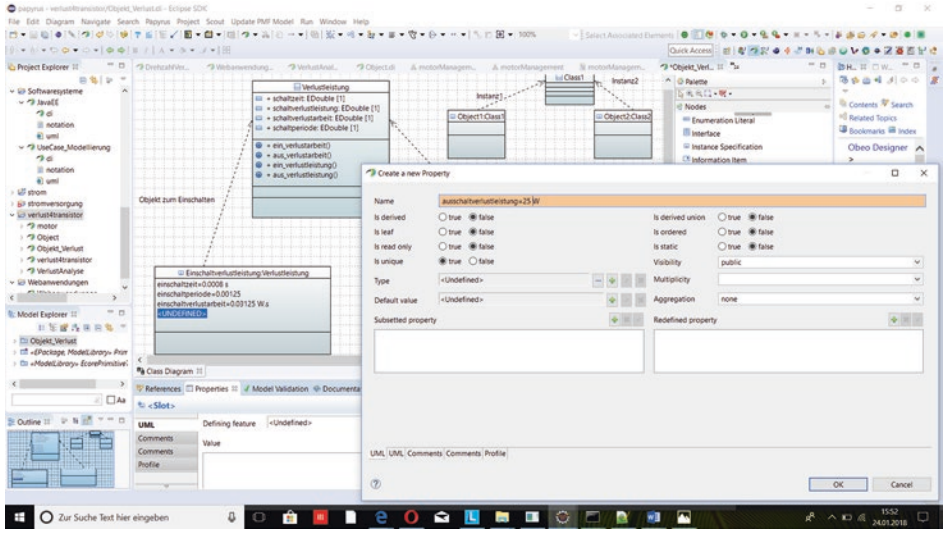


Abb. 3.103 Das Eingeben des *Wertes „25 W“* für den falschen erstellten Slot *„ausschaltverlustleistung“*

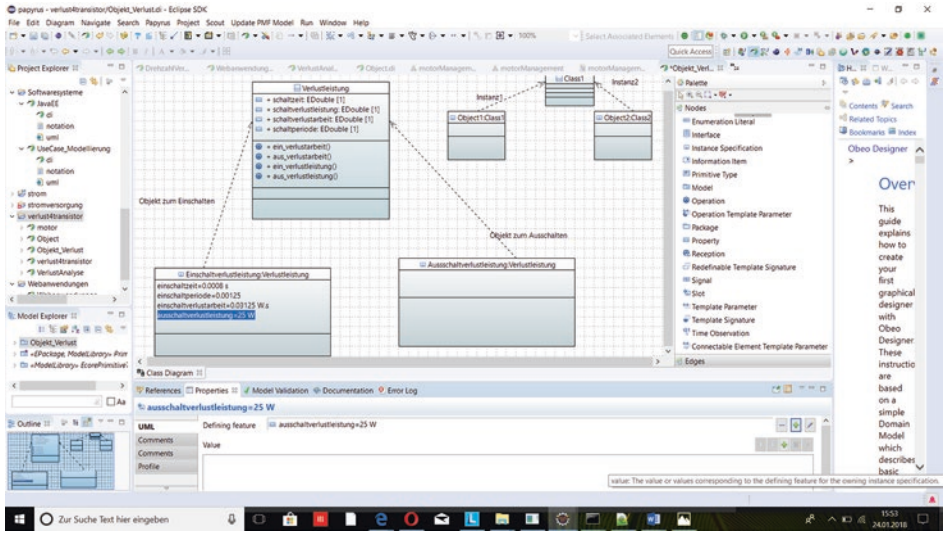


Abb. 3.104 Überblick über den Wert *„25 W“* des falschen erstellten Slots *„ausschaltverlustleistung“*

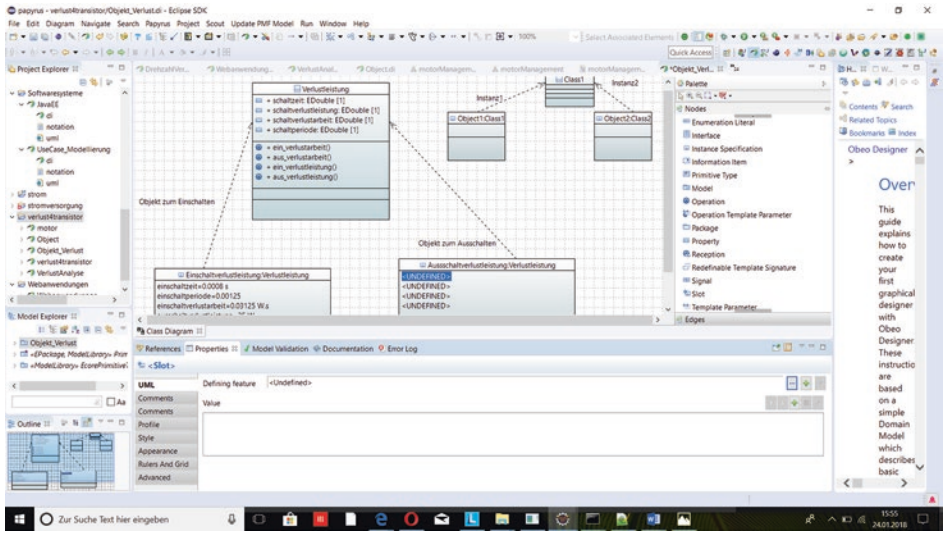


Abb. 3.105 Leere Slots vom Objekt *„Ausschaltverlustleistung“*

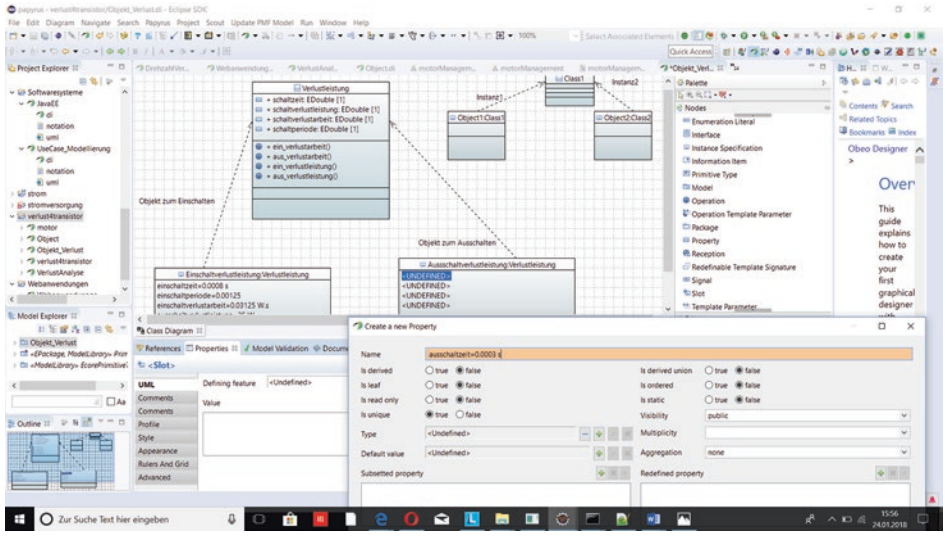


Abb. 3.106 Das Eingeben des Wertes *„0.0003 s“* für den erstellten Slot *„ausschaltzeit“*

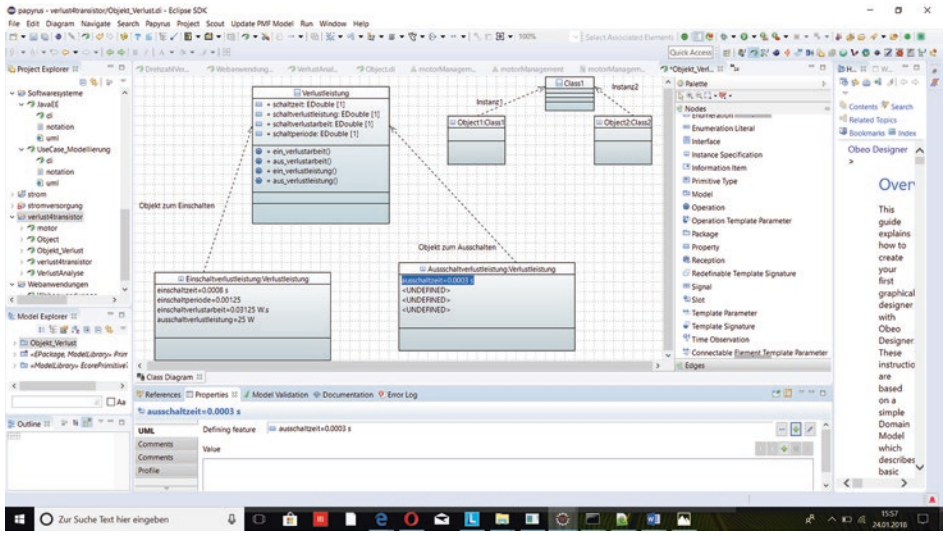


Abb. 3.107 Das Eingeben des Werts *„0.00125 s“* für den erstellten Slot *„ausschaltperiode“*

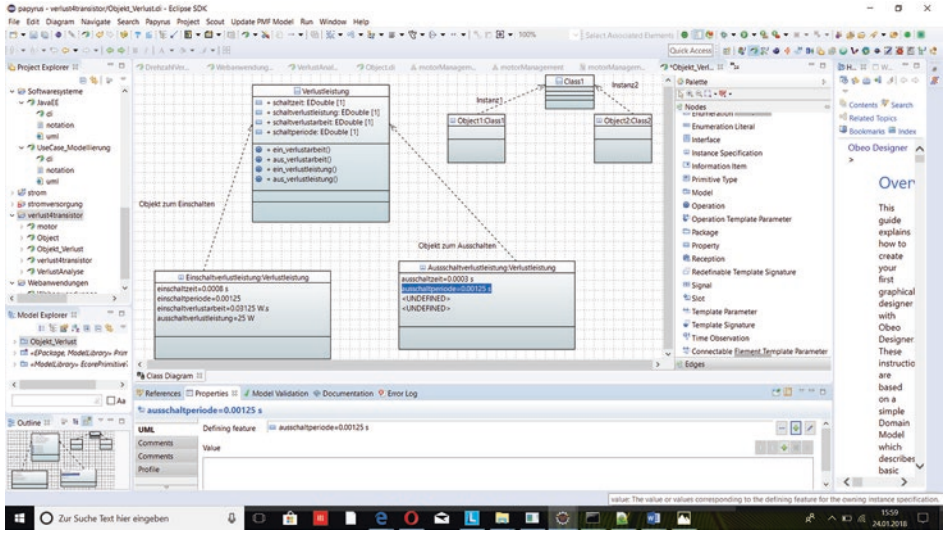


Abb. 3.108 Überblick über den Wert des erstellten Slots *„ausschaltperiode“*

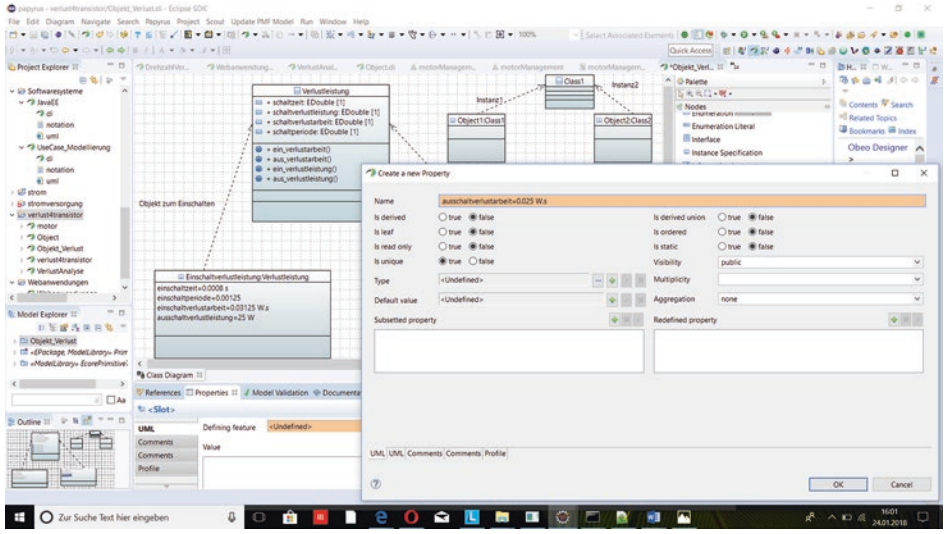


Abb. 3.109 Das Eingeben des Werts *„0.025 W.s“* für den erstellten Slot *„ausschaltverlustarbeit“*

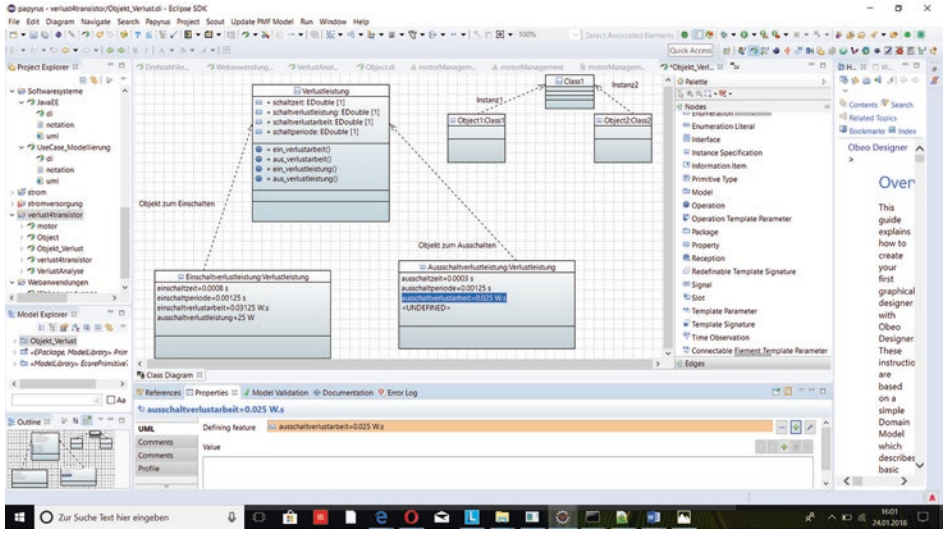


Abb. 3.110 Überblick über den Wert des erstellten Slots *„ausschaltverlustarbeit“*

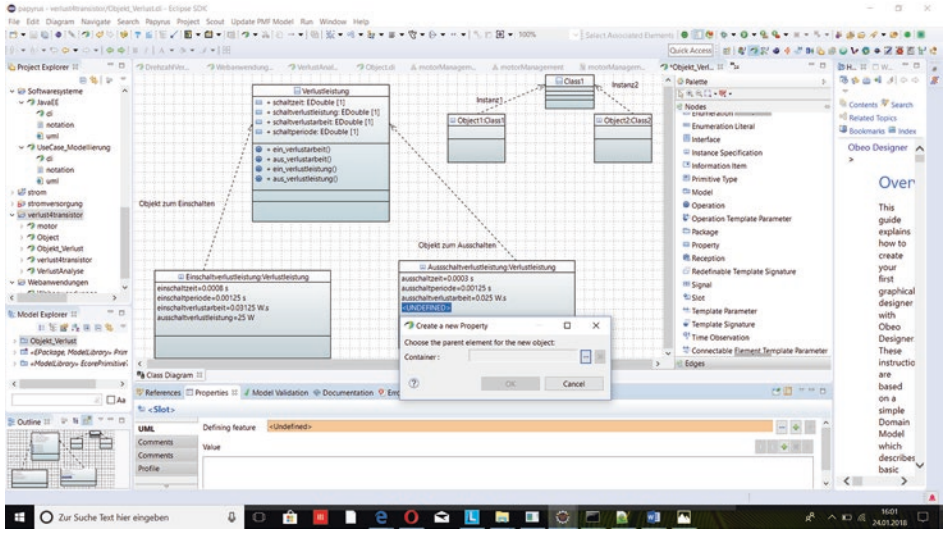


Abb. 3.111 Überblick über das Hinzufügen von Werteangaben (Slots) in das Objekt mit Hinblick auf das Auswählen der Eltern-Klasse *„Verlustleistung“* für das entsprechende Objekt *„Ausschaltverlustleistung“*

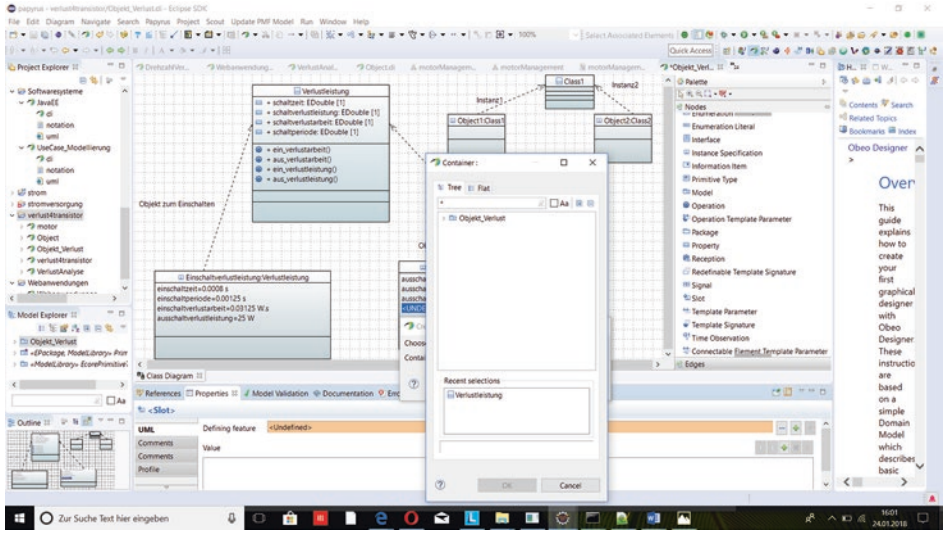


Abb. 3.112 Das Auswählen des Containers *„Verlustleistung“* im Modell *„Object_Verlust“*

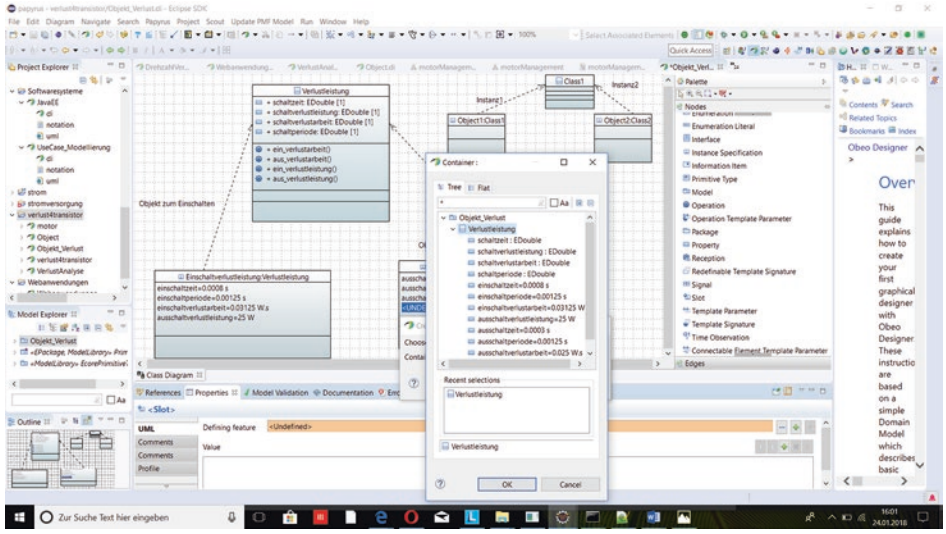


Abb. 3.113 Das Erstellen eines neuen Slots mithilfe des Containers *„Verlustleistung“*

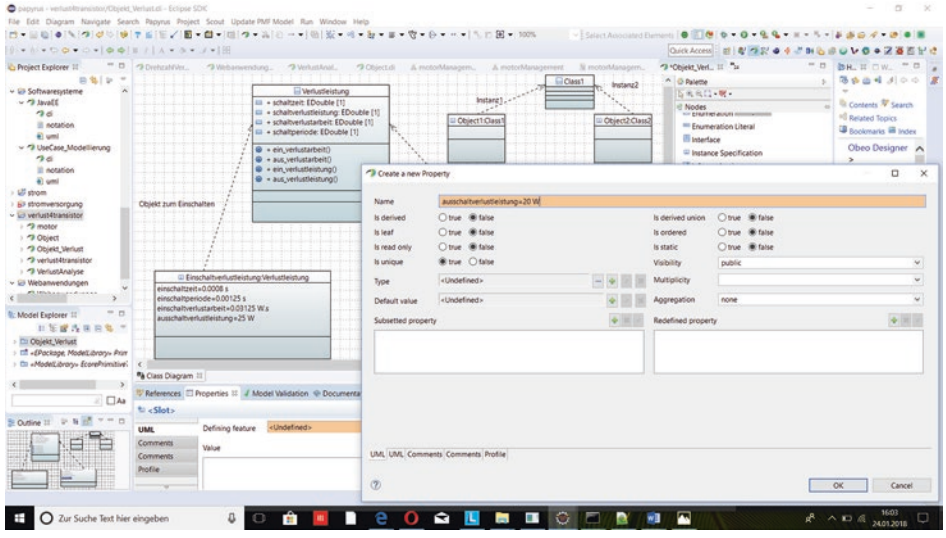


Abb. 3.114 Das Eingeben des Werts *„20 W“* für den erstellten Slot *„ausschaltverlustleistung“*

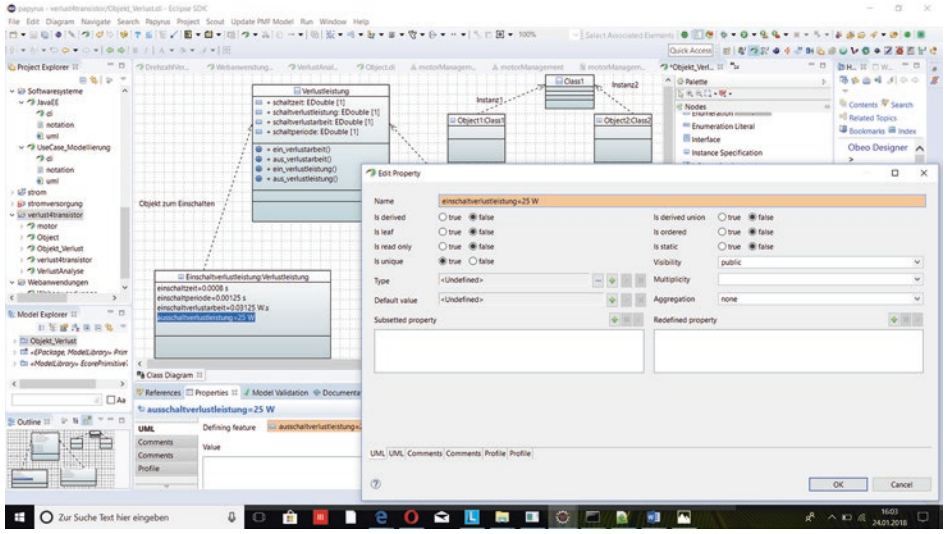


Abb. 3.115 Das Ersetzen des falschen Slots „*Ausschaltverlustleistung*“ durch den richtigen Slot „*Einschaltverlustleistung*“

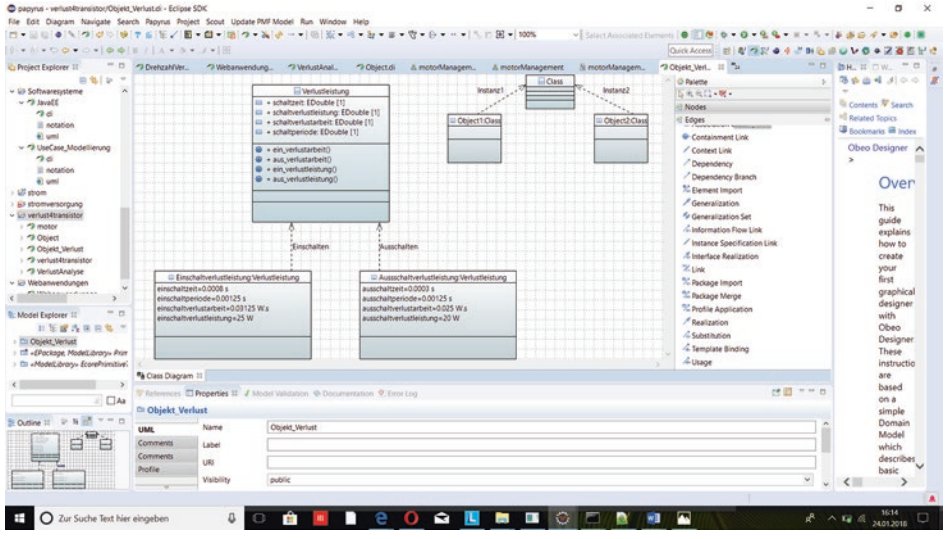


Abb. 3.116 Darstellung des modellierten Objektdiagramms mit Hinblick auf Objekte „*Einschaltverlustleistung*“und „*Ausschaltverlustleistung*“

3.6.2.1 Elemente des Objektdiagramms

Objektdiagramme sind mithilfe eines Dreieck-Symbols, genannt *„Instance Specification“,* aus der Palette des Editors des Frameworks Eclipse-Papyrus erstellt, wie es auf den Abb. 3.82, 3.83, 3.84, 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91, 3.92, 3.93, 3.94, 3.95, 3.96, 3.97, 3.98, 3.99, 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 zu sehen ist. Sie bestehen aus Objekten wie z. B. *Einschaltverlustleistung oder Ausschaltverlustleistung,* die durch Rechtecke dargestellt und mit Linien verbunden werden, wobei Objektdiagramme von UML die Beschreibung der Instanzen darstellen. Ein Objektdiagramm verfügt über folgende Elemente:

a) Objekte z. B. *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* – sind Instanzen einer Klasse. In der Klasse *„Verlustleistung“* können sich z. B. die Objekte *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* befinden. Die Objekte in der Klasse „Transistoren“ sind die jeweiligen Transistorteile, z. B. *„Insulate Gate Bipolar Transistor“, „MOSFET“, „Bipolare Leistungstransistor“* und *„Darlington-Transistor“*.
b) Klassentitel stellen die konkreten Attribute bestimmter Klassen dar. Im Objektdiagramm der Verlustleistungsanalyse sind die *„Schaltzeit“,* die *„Schaltperiode“,* die *„Schaltverlustarbeit“* und die *„Schaltverlustleistung“* des entsprechenden Transistorzustandes d. h. Einschaltung und Ausschaltung. Diese können in den Eigenschaften des Objektes (z. B. Schaltungsposition) aufgeführt werden.
c) Klassenattribute stellt ein Rechteck, genannt *„Slot“,* oder Werteangabe, die über mehrere Wertspezifikationen verfügt, während eine Instanzenbeschreibung über mehrere Slots verfügt. Gemäß UML-Spezifikation müssen die Werte eines Slots die Typkonformität darstellen.
d) Modellierungstools wie z. B. „Link“ sind Bestandteile von *„Nodes“* und *„Edges“* aus der Palette.

3.6.2.2 Grafische Darstellung vom Objektdiagramm

Objektdiagramme ermöglichen die Darstellung der Modellierung eines Systemausschnittes aus dem Systemdiagramm. Es gibt Ähnlichkeiten zwischen dem Objekt- und dem Klassendiagrammen. Interaktionen zwischen Objekten und Klassen gehören zum Konzept von Klassendiagrammen. Objekte stellen die Instanzen der Klassen dar. Abb. 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 erläutern die grafische Darstellung des Objektdiagrammes.

Auf den Abb. 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 sind zwei Zustandsverlustleistungen *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* zur Darstellung des Objektdiagramms mit der Transistorverlustleistung verbunden. Die Klassenbezeichnungen geben an, welchen Typ von Verlusten (Einschaltverlust und Ausschaltverlustleistung) die

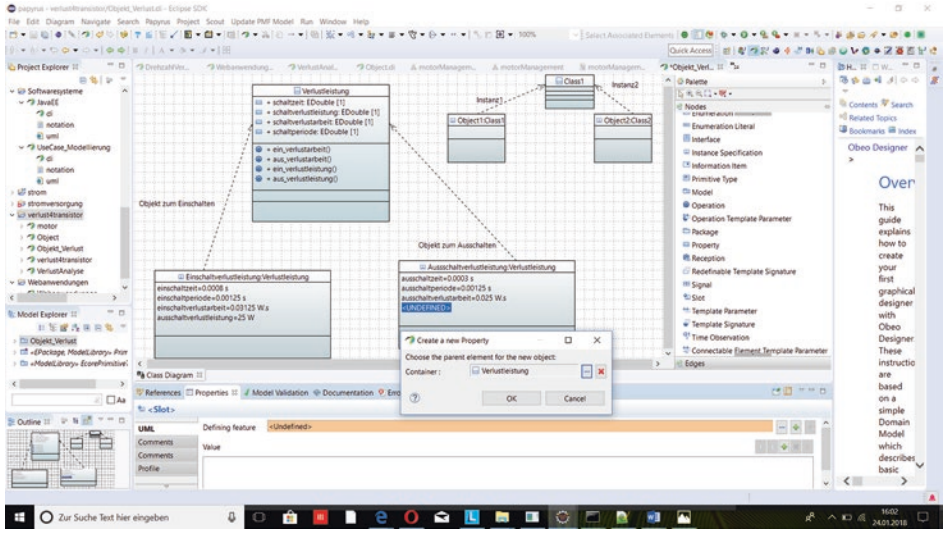


Abb. 3.117 Überblick über die Struktur des Containers *„Verlustleistung“* bezüglich des Objektes *„Ausschaltverlustleistung“*

Entwickler bei den Transistorverlusten berechnen. Die Klassenattribute wie z. B. *„einschaltzeit“* und *„ausschaltzeit“* sind von Verlust zu Verlust unterschiedlich. Dies wird dadurch dargestellt, dass das erste Objekt „Einschaltverlustleistung“ nur die Berechnung des Einschaltzustandes realisiert, während das zweite, genannt *„Ausschaltverlustleistung“*, die Kalkulation der Verluste beim Ausschalten ermöglicht.

3.7 Kompositionsstrukturdiagramm

Das Kompositionsstrukturdiagramm, genannt *„Composite Structure Diagram“*, ermöglicht die Darstellung der internen Struktur von Komponenten sowie anderen Modellelementen wie z. B. Klassen. Ziel der Modellierung mit Kompositionsstrukturdiagrammen ist es Beziehungen zwischen den Bestandteilen einer Struktur zu modellieren. Dieser Diagrammtyp ist deshalb zur Erstellung einer Softwarearchitektur geeignet. Die Bestandteile einer Komponente bzw. Klasse heißen Parts. Diese werden zusammen mit der Komponenteninstanz bzw. Objekten erzeugt und auch zerstört.

Das Kompositionsstrukturdiagramm dient der Dekomposition und Modellierung von *„Klassifikatoren“*, die in Kompositionsbeziehungen zueinanderstehen. Es zeigt, wie die Teile eines Klassifikators den Klassifikator selbst bilden. Kompositionsstrukturdiagramme ermöglichen die Darstellung der Teile (Architekturelemente) eines Systems und derer Beziehungen. Deshalb werden sie auch Architekturdiagramme genannt und eignen sich ebenfalls zur Modellierung von Entwurfsmustern [3]. Ein Part ist eine Laufzeitinstanz einer

Klasse oder Schnittstelle. Parts werden verwendet, um Kompositionsstrukturen oder Entwurfsmuster darzustellen. Parts können Eigenschaften von Klassen sein, d. h., sie können in der Klasse enthalten sein.

Das Kompositionsstrukturdiagramm stellt auch einen strukturierten Klassifikator in Bezug auf die Zusammensetzung des Ganzen aus den richtigen Teilen dar. Hierbei sind Objekte als *„Parts“* dargestellt und zur Initialisierung der Klasse erzeugt.

3.7.1 Komposition

Klassen bestehen aus verschieden Elementen, u. a. Attribute und Methoden. Der Begriff „Komposition“ stellt die Struktur der Teile des Ganzen (z. B. Klasse) dar. Abb. 3.118, 3.119, 3.120, 3.121, 3.122, 3.123, 3.124, 3.125, 3.126, 3.127, 3.128, 3.129, 3.130, 3.131, 3.132, 3.133, 3.134, 3.135, 3.136, 3.137, 3.138, 3.139 und 3.140 geben Überblicke über das Erstellen des Kompositionsstrukturdiagramms bezüglich der Laufinstanzen von Klassen. Abb. 3.118 zeigt das Auswählen des Diagrammtyps *„Composite Structure Diagram“* für das Projekt *„verlust4transistor“*. Anschließend zeigt Abb. 3.119 den Aufbau einer Komposition in Bezug auf zwei Parts und das *„Interface Schaltverlust“*. Abb. 3.120, 3.121, 3.122 und 3.123 zeigen die Funktionalität des Klassifikators *„Berechnung“* mit den Objekten *„ein_VerlustErmitteln“* und *„aus_VerlustErmitteln“*. Das Erstellen der Laufinstanzen mithilfe der Klasse *„Berechnung“* zeigt Abb. 3.120, 3.121, 3.122 und 3.123. Abb. 3.124, 3.125, 3.126, 3.127, 3.128, 3.129, 3.130, 3.131, 3.132, 3.133, 3.134 und 3.135 geben Überblicke über Schritte zum Hinzufügen von Parts mithilfe des Containers *„Berechnung“*. Hierbei sind die hinzugefügten Objekte *ein_VerlustErmitteln* und *aus_VerlustErmitteln* Teile des Klassifikators *„Berechnung“*. Sie stellen die Instanzen der Klassen *Einschaltverlustleistung* und *Ausschaltverlustleistung* dar.

Gemäß Abb. 135–140 sind die Klassen *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* in der Hauptklasse *(Main-Programm)* zum Realisieren der Kalkulation für die Verlustanalyse des Transistors instanziiert. Die Klasse *„Berechnung“* verfügt über die main-Methode, deren Instanzen das Berechnen ermöglichen. Die beiden Objekte (*„Parts“*) *ein_VerlustErmitteln* und *aus_VerlustErmitteln,* die im Strukturabschnitt der Klasse dargestellt sind, werden beim Initialisieren der Klasse kreiert und mit ihr gelöscht, wobei *ein_VerlustErmitteln aus_VerlustErmitteln* Eigenschaften der Klassen Einschaltverlustleistung bzw. Ausschaltverlustleistung sind. Abb. 3.135, 3.136, 3.137, 3.138, 3.139 und 3.140 stellen sicher, dass *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* mithilfe der Ermittlung der *Einschaltverlustleistung* bzw. *Ausschaltverlustleistung* zu der gleichen *„Kalkulation“* gehören.

3.7.2 Klassifikator

Weil Klassen Klassifikationen der Objekte mithilfe von Attributen, Operationen und Beziehungen mit anderen Objekten darstellen, verfügen Klassifikatoren über Instanzen dieser Klassen. Wobei Klassifikatoren Instanzen mit gemeinsamen Eigenschaften beschreiben. Spezielle Klassifikatoren sind sowohl Interface als auch Klassen. In Bezug auf die strukturelle Einordnung ist der Begriff *„Klassifikator“* für die Gestaltung der Kompositionsstrukturdiagramme wichtig. Hierbei enthält der Klassifikator Instanzen der anderen Klassen. Das bedeutet, dass Klassen nach ihren strukturellen Eigenschaften eingeordnet werden. Abb. 3.138, 3.139 und 3.140 zeigen deutlich die Struktur des Klassifikators *„Berechnung“* hinsichtlich der Darstellung der Instanzen. Der Klassifikator *„Berechnung“* ist eine Klasse mit der *main*-Methode zum Instanziieren der beiden Klassen *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“*, welche eine Verbindung, genannt *„Konnektor“* oder *Connector,* zum Transistor haben. Ein Konnektor verbindet zwei Parts wie z. B. *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* oder *„Objekte“* mithilfe von Eingängen, genannt *„Ports“*. Abb. 3.135, 3.136, 3.137, 3.138, 3.139 und 3.140 zeigen das Erstellen von sowohl einem Konnektor zwischen den Parts *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* als auch von *„Port1 und Port2“*. Ports verfügen über Datentypen wie z. B. *„EDouble“*.

Abb. 3.140 zeigt die Modellierung der Analyse der Verluste bei einem Transistor mithilfe des Kompositionsstrukturdiagramms bezüglich der Struktur des Klassifikators *„Berechnung“*.

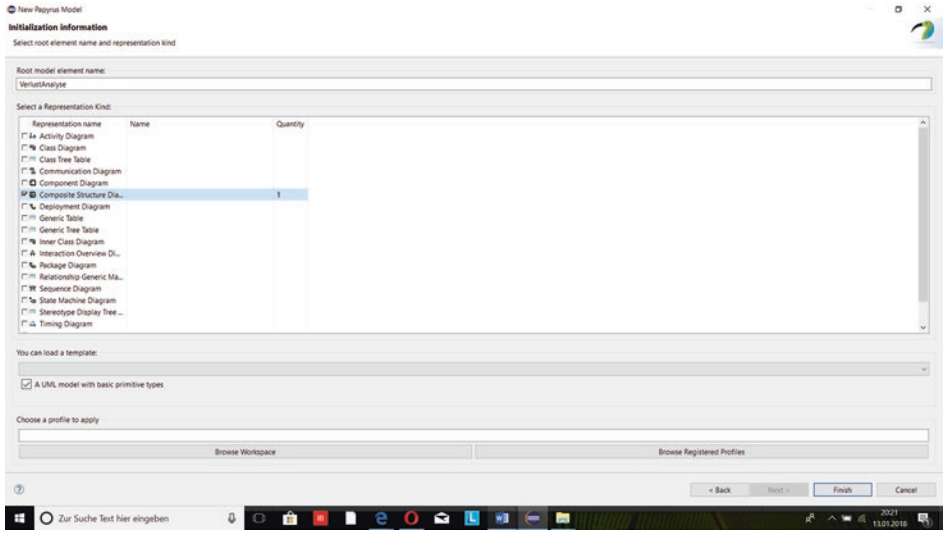


Abb. 3.118 Das Auswählen des Diagrammtyps *„Composite Structure Diagram“* für das Modell *„VerlustAnalyse“*

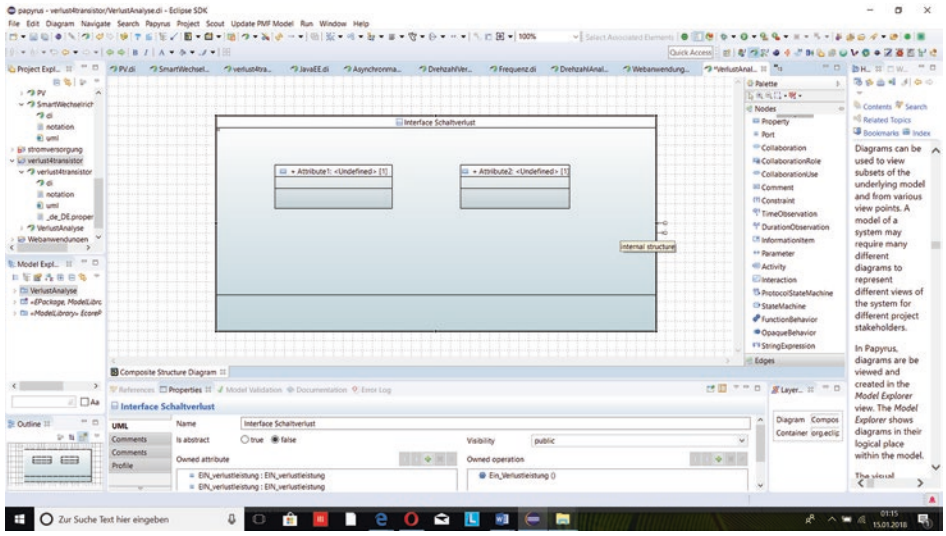


Abb. 3.119 Überblick über die Struktur des Klassifikators *„Interface Berechnung“*

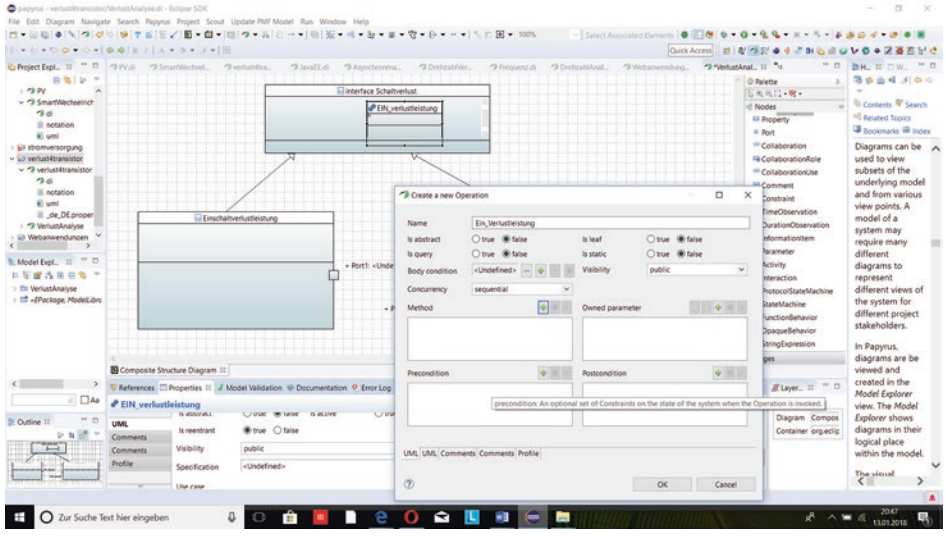


Abb. 3.120 Hinzufügen von Objekten in eine Klasse

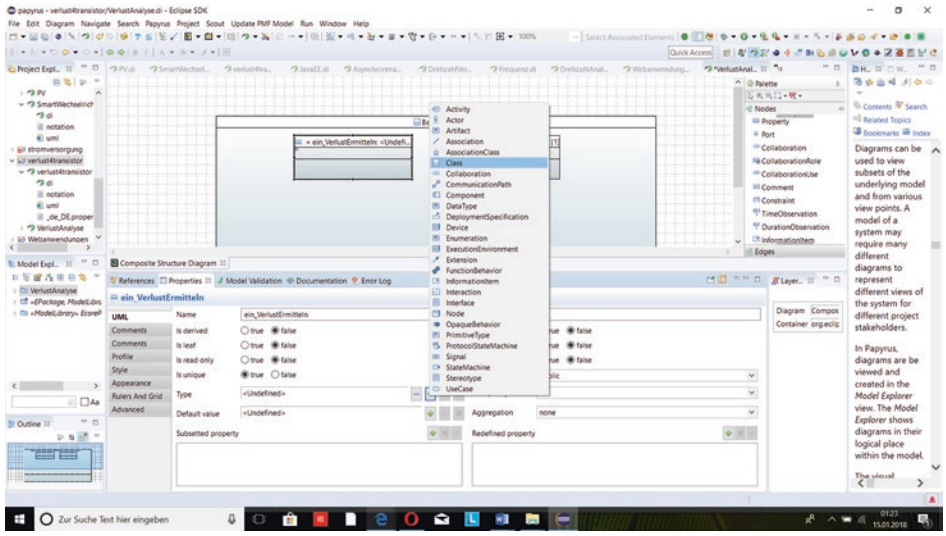


Abb. 3.121 Das Auswählen des Klassifikator in einem Container

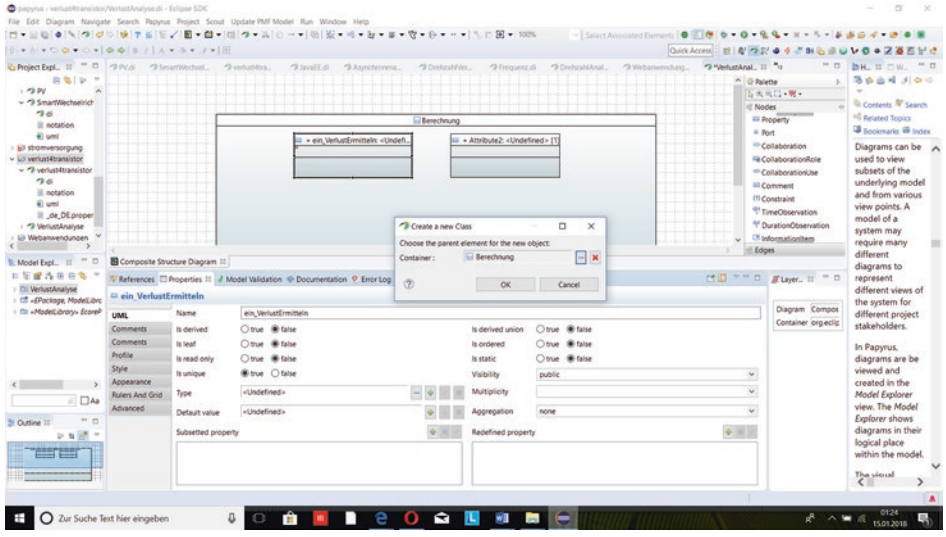


Abb. 3.122 Das Erstellen des Objektes *„ein_verlustErmitteln“* in dem Container *„Berechnung“*

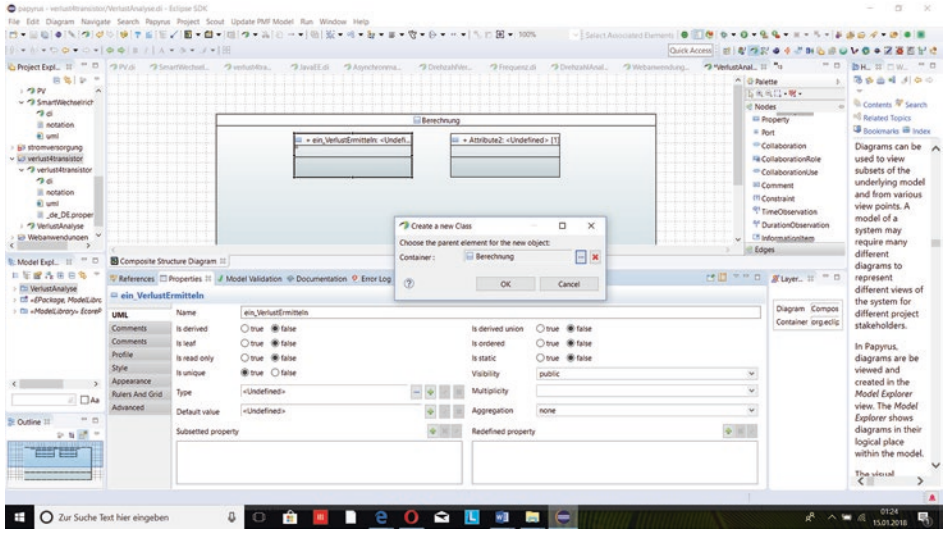


Abb. 3.123 Das Erstellen einer Klasse im Container *„Berechnung“*

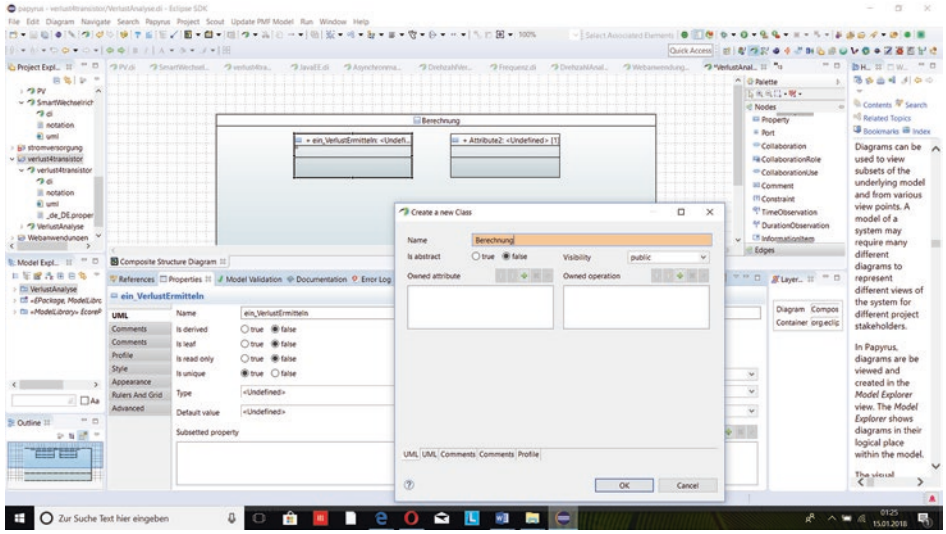


Abb. 3.124 Das Hinzufügen des Klassifikators *„Berechnung“* zum Erstellen einer Klasse

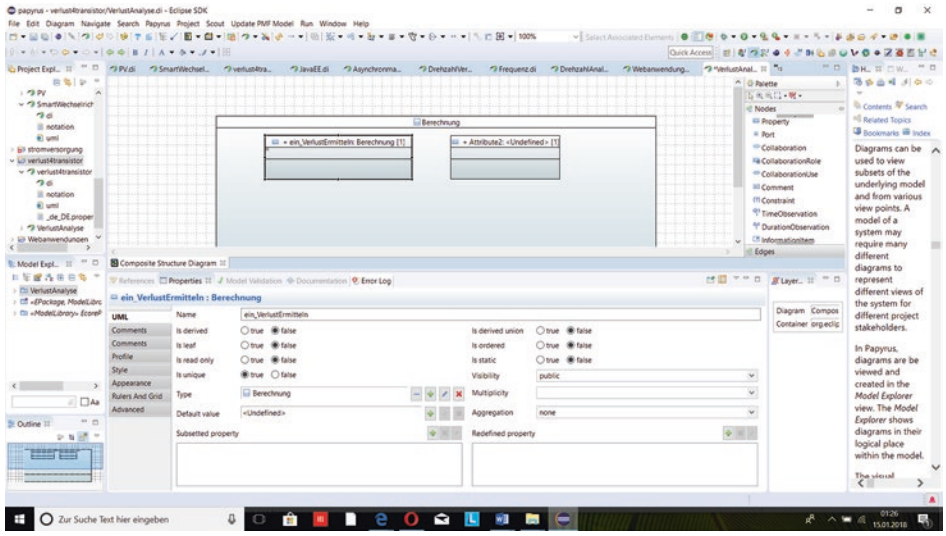


Abb. 3.125 Darstellung des *Objekts „ein_VerlustErmitteln“* als *Part* des Klassifikators *Berechnung*

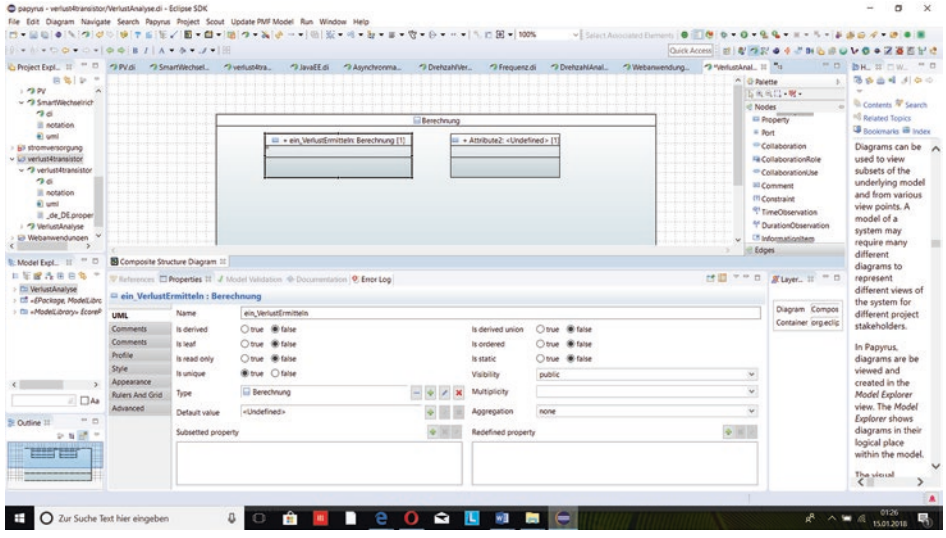


Abb. 3.126 Überblick über Struktur des Parts *„aus_VerlustErmitteln“*

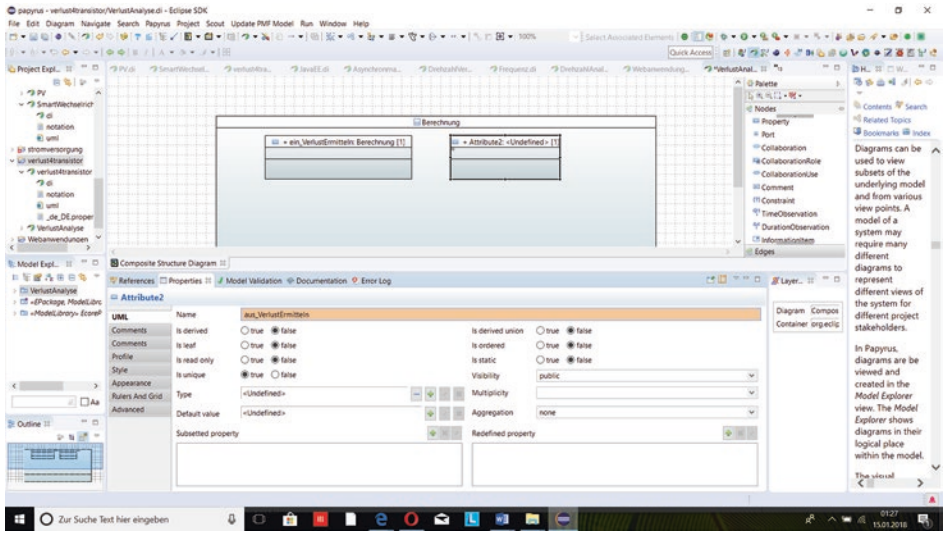


Abb. 3.127 Das Hinzufügen einer Klasse für das Objekt *„aus-VerlustErmitteln“*

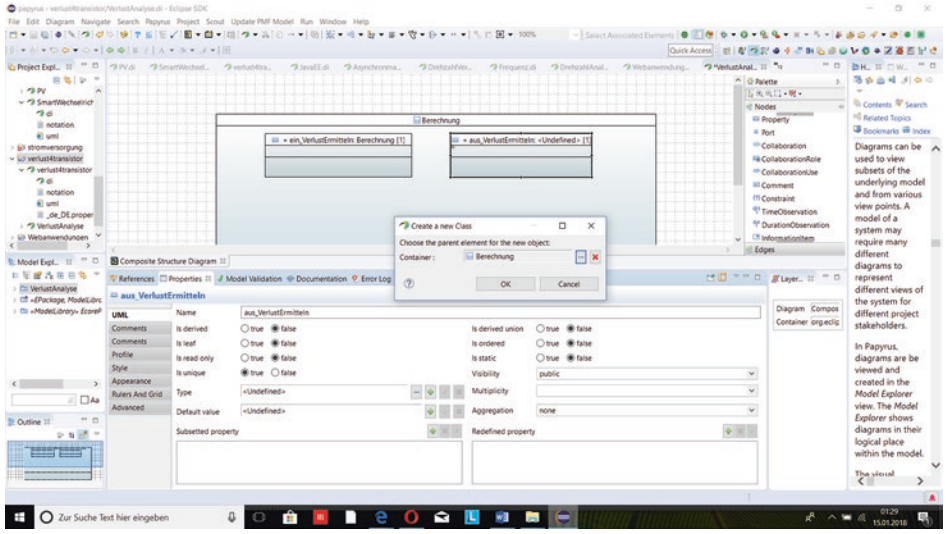


Abb. 3.128 Das Auswählen des Containers *„Berechnung“* zum Erstellen einer Klasse für das Objekt *„aus_VerlustErmitteln“*

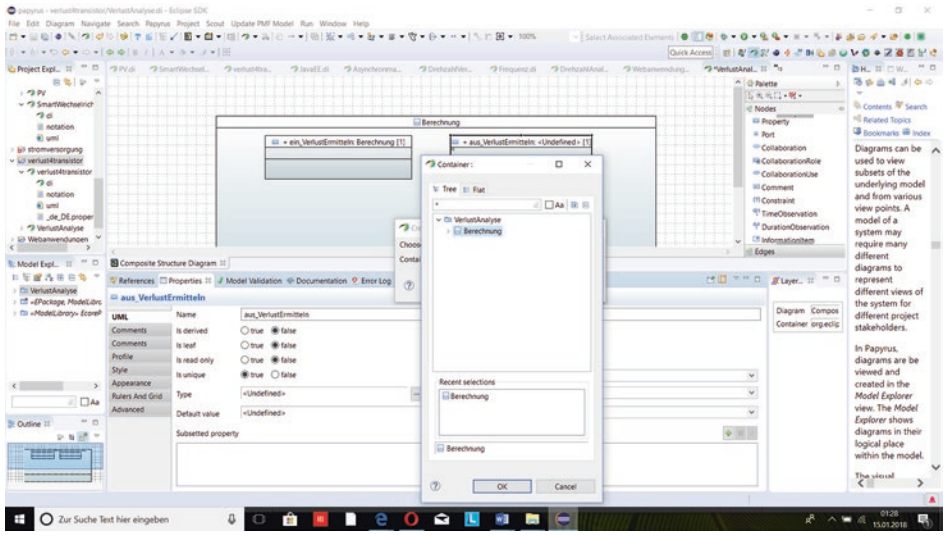


Abb. 3.129 Überblick über Container *„Berechnung“*

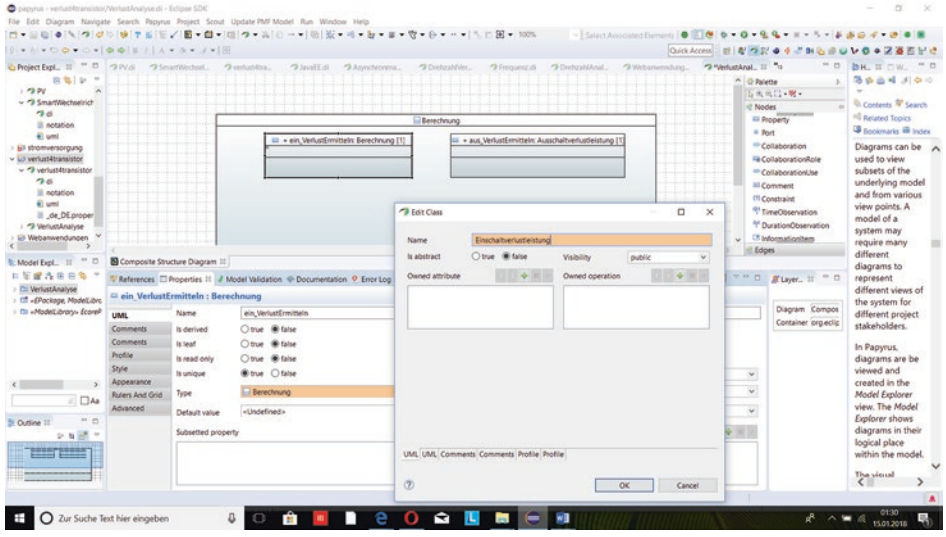


Abb. 3.130 Ersetzen der nicht passenden Klasse *„Berechnung“* durch die richtige Klasse, genannt *„Einschaltverlustleistung“*, für das Objekt *„ein_VerlustErmitteln“*

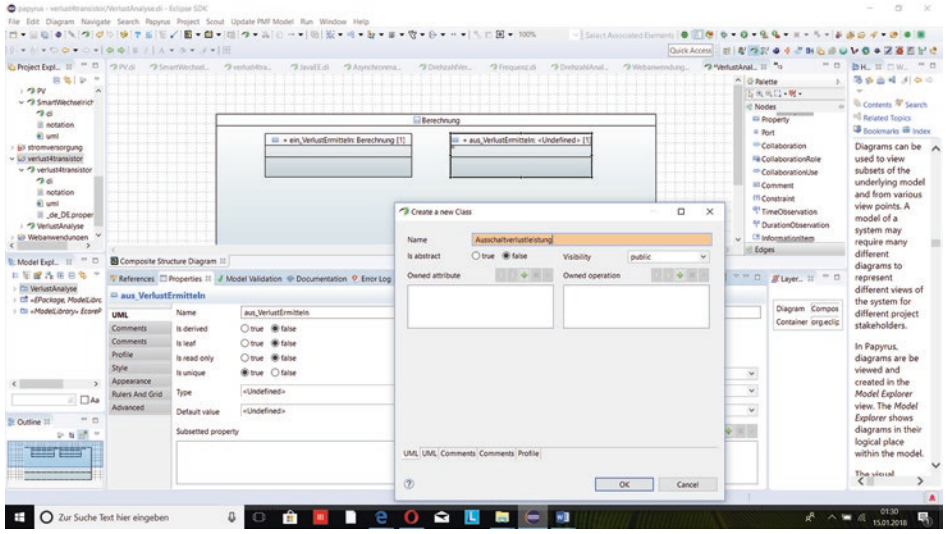


Abb. 3.131 Ersetzen der nicht passenden Klasse *„Berechnung“* durch die richtige Klasse, genannt *„Ausschaltverlustleistung“*, für das Objekt *„aus_VerlustErmitteln“*

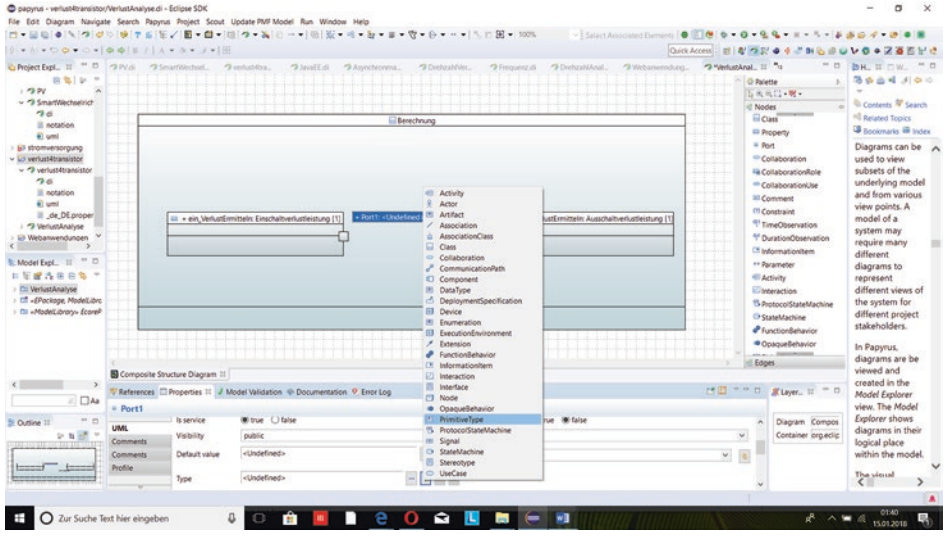


Abb. 3.132 Das Hinzufügen des Datentyps des *„Port1“* aus dem Part *„ein_VerlustErmitteln“* mithilfe vom Symbol *„PrimitiveType“*

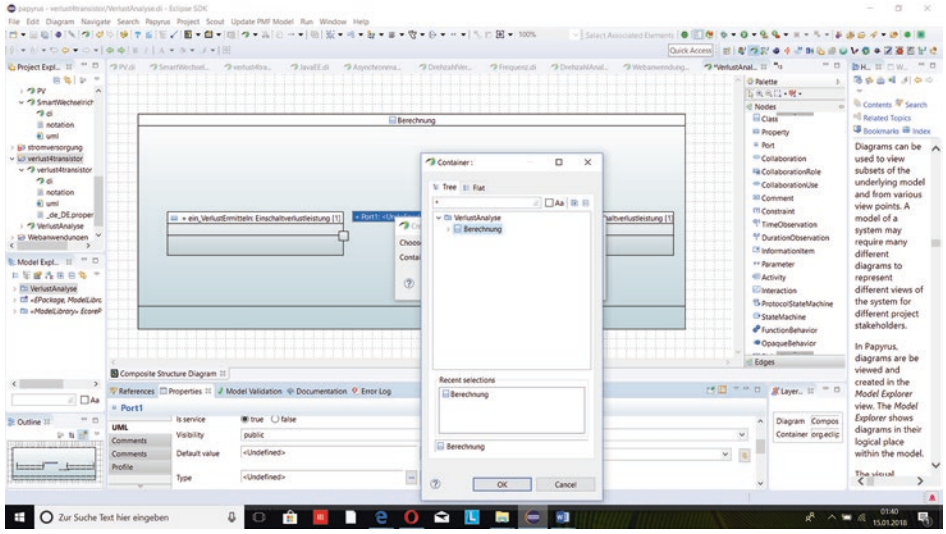


Abb. 3.133 Das Auswählen des Containers *„Berechnung“* zum Erstellen des *„Port1“* aus dem Part *„ein_VerlustErmitteln“*

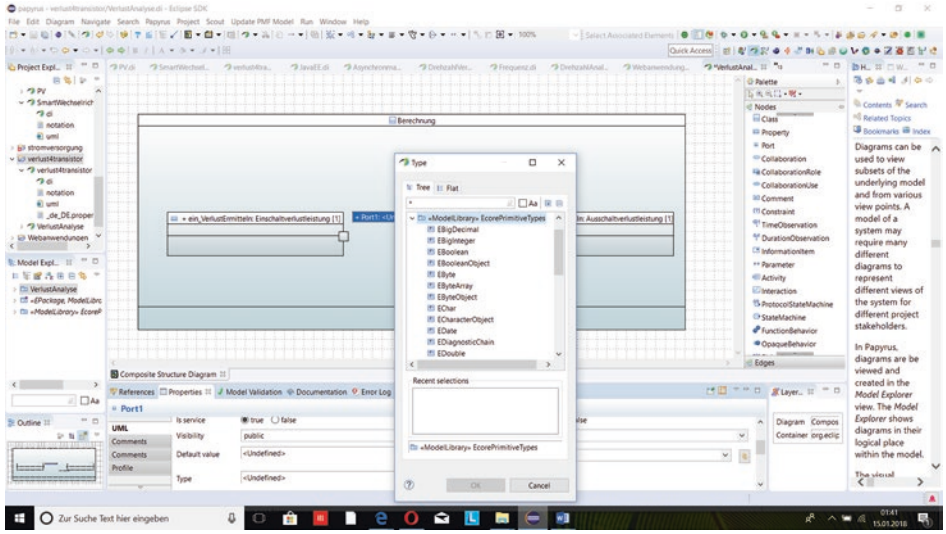


Abb. 3.134 Das Auswählen des Ecore Datentyps des *„Port1“* aus dem Part *„ein_VerlustErmitteln“* mithilfe von *„ModelLibrary“ EcorePrimitiveTypes*

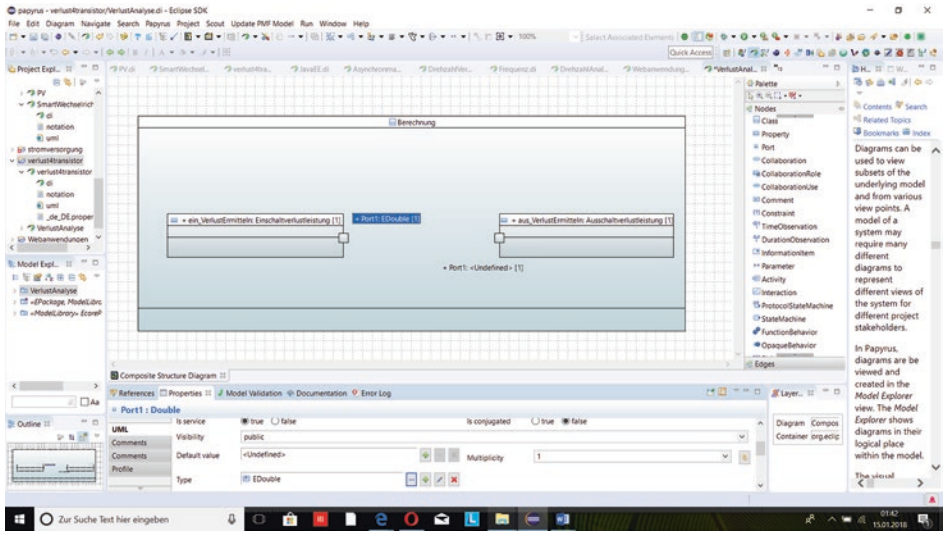


Abb. 3.135 Überblick über den Datentyp *„EDouble“* vom *„Port1“* aus dem Part *„ein_VerlustErmitteln“*

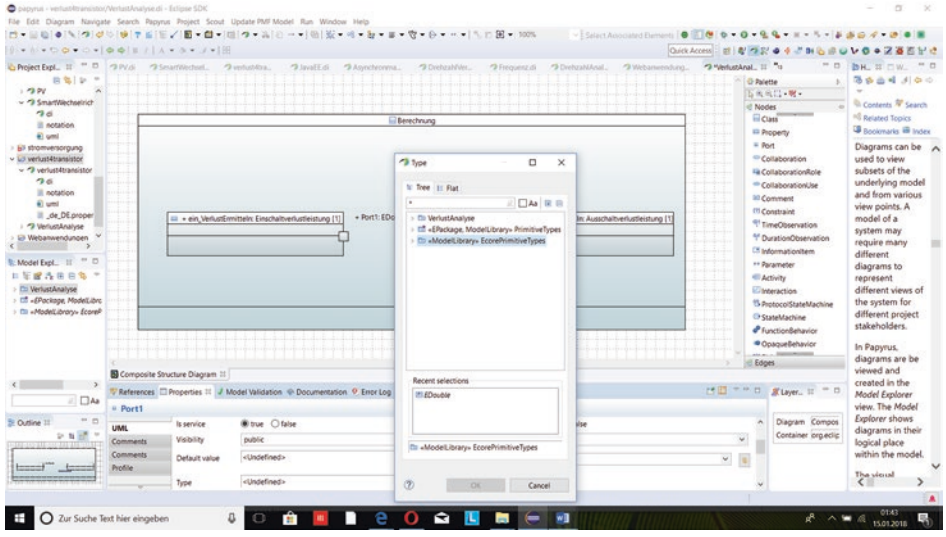


Abb. 3.136 Darstellung der Pakete *„EPackage ModelLibrary“* PrimitiveTypes und *„ModelLibrary“* EcorePrimitiveTypes

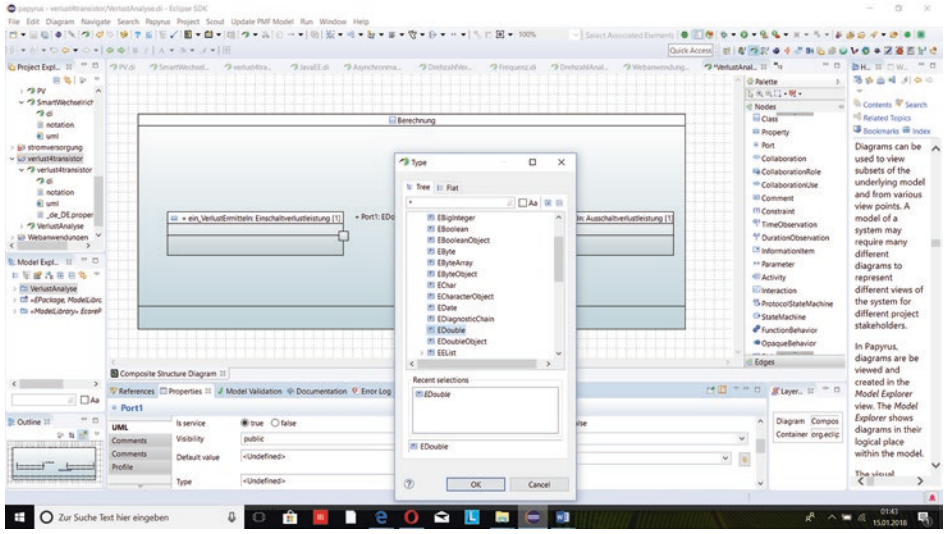


Abb. 3.137 Das Auswählen des Datentyps *„EDouble“* für den Port vom Part *„aus_VerlustErmitteln“*

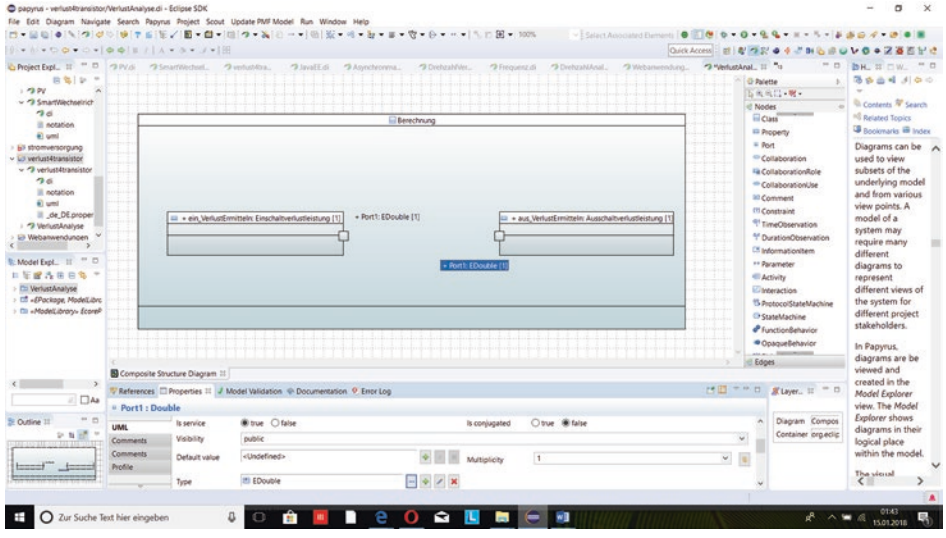


Abb. 3.138 Überblick über den Port und dessen Datentyps *„Edouble“* aus dem Part *„aus_VerlustErmitteln“*

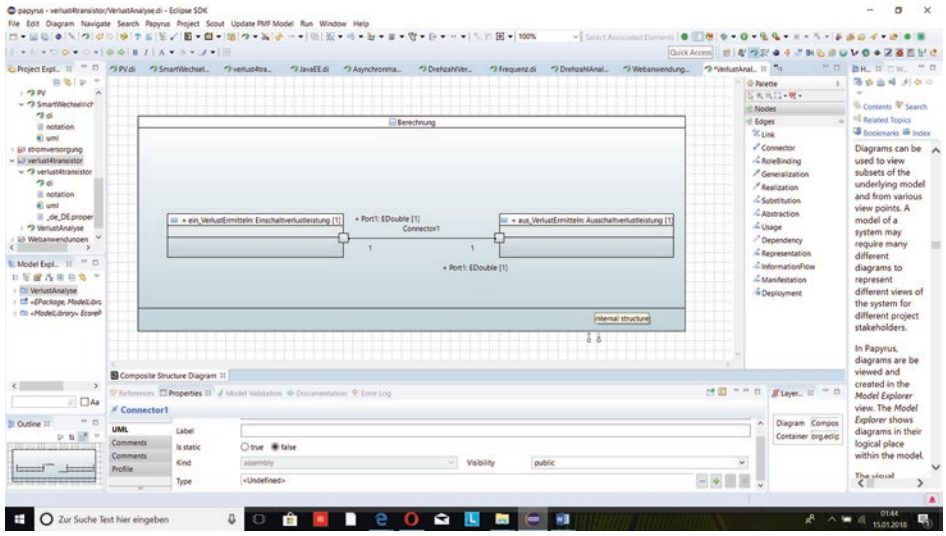


Abb. 3.139 Darstellung der Verbindung zwischen den Parts *„ein_VerlustErmitteln“* und *„aus_VerlustErmitteln“* mithilfe von zwei Ports

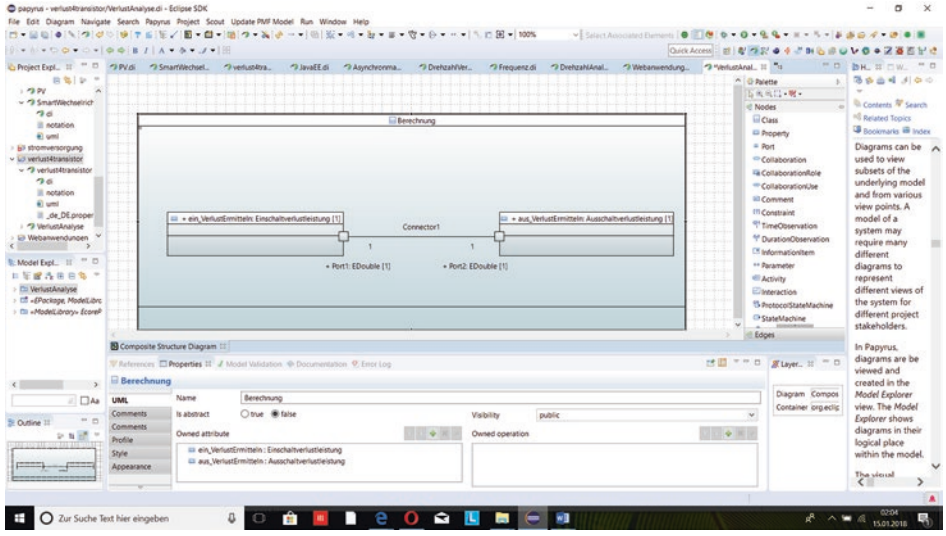


Abb. 3.140 Modellierung der Analyse der Verluste bei einem Transistor mithilfe des Kompositionsstrukturdiagramm bezüglich der Struktur des Klassifikators *„Berechnung“*

3.8 Komponentendiagramm mit Eclipse-Papyrus

Komponentendiagramme stellen die Verbindung zwischen den Komponenten zum Analysieren der Komponenten- oder Softwaresysteme dar. Das Ziel der UML-Modellierung mit den Komponentendiagrammen ist es, die Abhängigkeiten jeder Komponente im System zu repräsentieren. Komponenten sind erforderlich, damit Stereotyp-Funktionen ausgeführt werden können. Wobei Komponenten-Stereotypen mithilfe der ausführbaren Programme, Dokumente, Datenbanktabelle, Dateien oder Bibliotheksdateien erzeugt werden können [7].

Komponenten stellen sowohl die Kapselung des Zustandes und des Verhaltens derer Elemente als auch die Spezifizierung einer Verbindung mit der Umgebung dar. Wie Klassen verfügen Komponente über Instanzen, wobei sie aus Klassen bestehen. Komponentendiagramme verfügen über zum einen exportierte Schnittstellen und zum anderen importierte Schnittstellen, welche Verbindungen zwischen Komponenten und Umgebungen ermöglichen. Dies bedeutet auch, dass Komponenten eine ausführbare und austauschbare Softwareeinheit mithilfe von Schnittstellen darstellen [13]. In Lauf der Zeit weisen verbundene Komponente auf austauschbare Einheiten mithilfe von Interface hin [14]. Dies bedeutet, dass zum einen Komponentendiagramme die Struktur eines Systems zur Laufzeit modellieren und zum anderen Werkzeugkasten die Bestandteile dieses Systems beschreiben. Beispielsweise Dot.NET-Assembly oder Java-war-Dateien stellen Artefakte zur Implementierung der Komponente dar.

Beispielsweise stellen Servlets-, JavaServer-Pages-, JavaServer-Faces-Anwendungen die Web-Komponenten, genannt *„Web-tier components“*, dar. Diese Komponenten sind in Web-Container wie z. B. Tomcat ausgeführt und können auf http-Anfragen mithilfe des Web-Clients antworten [15].

3.8.1 Praxisbeispiel: Abhängigkeit zwischen Komponenten und Interface

Verbindungen zwischen Komponenten und Interface ermöglichen das Analysieren der Funktionalitäten der Systeme zum einen und zum anderen der objektorientierten Modellierung mithilfe des UML-Diagramms. Hierbei sind Interface wichtige Verbindungen zu den Komponenten zum Darstellen von Anwendungen in Bezug auf Technik und Informatik. Das Praxisbeispiel fokussiert auf die Analyse der Verluste im Transistor *„Insulated Gate Bipolar Transistor“* mithilfe der Informatik. Das Beispiel stellt die Verbindung zwischen dem Transistor und dem Interface *„Schaltverlust“* dar. Abb. 3.141, 3.142, 3.143, 3.144, 3.145, 3.146, 3.147, 3.148, 3.149, 3.150, 3.151, 3.152, 3.153 und 3.154 zeigen mithilfe des Erstellens des Komponentendiagramms die Funktionalität der Verbindungen zwischen Komponenten und Interface. Mithilfe der Interfaces sind Verluste anhand objektorientiere Modellierung zu analysieren. Abb. 3.141 zeigt das Auswählen des Diagrammtyps, genannt *„Component Diagram“,* für das Root

Model „motor“. Anschließend zeigt Abb. 3.142 zeigt den Editor von Eclipse-Papyrus mit dem *„Komponentendiagramm“*, welches über zum einen den genannt „Palette“ und zum anderen das Paket *„Verlust4Modeling“*. Die Verbindung zwischen der Komponente IGBT und dem Interface IGBT zeigt Abb. 3.143. Hierbei ist das Hinzufügen des Attributs *„ein_verlustleistung“* und dessen Datentyps mithilfe von *„ModelLibrary“ EcorePrimitiveTypes dargestellt.* Abb. 3.144 zeigt das Hinzufügen eines zweiten Attributs, genannt *„aus_verlustleistung“*. Abb. 3.144 gibt einen Überblick über die Struktur des Komponentendiagramms mit der Komponente *„IGBT“* und drei Interfaces. Abb. 3.144 zeigt das Hinzufügen einer Operation ins Interface. Abb. 3.145, 3.146, 3.147, 3.148, 3.149 und 3.150 zeigen das Komponentendiagramm mit der Komponente und das Interface *„Schaltverlust“* und dessen Eigenschaften wie z. B. Attribute *„ein_verlustleistung“* und *Operation „ein_VerlustErmitteln“*. Abb. 3.148, 3.149 und 3.150 stellt die objektorientierte Modellierung mithilfe der Vererbungsbeziehung zwischen Interface „Schaltverlust“ und „Einschaltverlustleistung“ auf einer Seite und auf anderer Seite Interface *„Schaltverlust“* und *„Ausschaltverlustleistung“* mithilfe der *„Links“ „Generalization“* dar. Hierbei ist anzumerken, dass zum einen *Schaltverlust* das *Ober-Interface* darstellt und zum anderen *Einschalt-* und *Ausschaltverlustleistung Unter-Interfaces* darstellen. Eine Generalisierung ist eine Beziehung zwischen einer allgemeinen und einer speziellen Klasse, wobei die speziellere weitere Merkmale hinzufügen kann und sich kompatibel zur allgemeinen verhält. Bei der Generalisierung bzw. Spezialisierung werden Merkmale hierarchisch gegliedert, d. h., Merkmale allgemeinerer Bedeutung werden allgemeineren Elementen zugeordnet und speziellere Merkmale werden Elementen zugeordnet, die den allgemeineren untergeordnet sind. Die Merkmale der Ober-Elemente werden an die entsprechenden Unter-Elemente weitergegeben, d. h. vererbt. Ein Unter-Element verfügt demnach über die in ihm spezifizierten Merkmale sowie über die Merkmale seiner Ober-Elemente. Unter-Elemente erben alle Merkmale ihrer Ober-Elemente und können diese um weitere erweitern oder überschreiben [12].

Gemäß Abb. 3.145, 3.146, 3.147, 3.148, 3.149 und 3.150 sind hierarchische Vererbungsbäume modelliert. Vererbungsbäume stellen, wie es auf den Abb. 3.147, 3.148, 3.149 und 3.150 zu sehen ist, wichtige Gestaltungselemente bei der Modellierung von Softwarearchitekturen dar.

Abb. 3.149 und 3.150 zeigen sowohl das Hinzufügen von weiteren Attribute in das Interface *„Schaltverlust“* als auch dieses von Operationen *„ein_VerlustErmitteln()“* und *„aus_VerlustErmitteln()“* in die Interfaces *„Einschaltverlustleistung“* bzw. *„Ausschaltverlustleistung“*. Gemäß Abb. 3.150 verfügt das Ober-Interface *„Schaltverlust“* über vier *Attribute,* u. a. *ein_verlustarbeit, aus_verlustarbeit, einschaltperiode, ausschaltperiode* und zwei *Operationen ein_verlustleistung()* und *aus_verlustleistung().*

Der Werkzeugkasten, genannt *„Palette“*, besteht aus zwei Teilen: Nodes und Links. Zum einen geben Abb. 3.146 oder Abb. 3.149 und 3.150 einen Überblick über die Elemente von Nodes, u. a. *„Constraint“, „Component“, „Interface“, „Package“, „Property“, „Operation“*

oder *„Port“* und zum anderen zeigen Abb. 3.147 und 3.148 verschiedene Symbole von Links wie z. B. *„Dependency“, „Connector“, „Generalization“, „Link“, „Interface Realisation“* oder *„Component Realization“*.

3.8.2 Kapselung von Zustand und Verhalten

Kapselung stellt die Abhängigkeit von Objekten in einer Komponente mithilfe der objektorientierten Analyse dar. Die Analyse der Funktionalität einer Kapselung wird in diesem Abschnitt als Darstellung der Beziehungen zwischen Elemente bezeichnet. Transistor IGBT ist mit der Diode verbunden und beide sind mit dem Lastkreis des Wechselrichters verbunden. Das Ganze ist indirekt mit dem Interface *„Schaltverlust“* verbunden. Dieses Ober-Interface stellt Vererbungsbeziehungen mit den Unter-Interfaces *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* dar. Die Kapselung von Zustand und Verhalten fokussiert sowohl auf *„Property“* als auch auf *„Operation“*. Das Ober-Interface *„Schaltverlust“* wird mithilfe von dem Transistor IGBT dargestellt, aber die Unter-Interfaces *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* haben keine direkte Beziehung mit dem Transistor. Sie wurden nicht vom demTransistor realisiert. Sie sind Erbe des Interfaces *„Schaltverlust“*. Die Funktionalitäten zum Berechnen der Verlustleistung während des Einschaltens bzw. Ausschaltens des Transistors werden mithilfe der Instanziierung in den Unter-Interfaces *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* realisiert. Die Kapselung ermöglicht mithilfe von *Objekten* die Berechnung der Verluste sowohl im *Einschalt-* als auch im *Ausschaltzustand.*

Abb. 3.151, 3.152, 3.153 und 3.154 zeigen, dass die Unter-Interfaces *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* keinen Zugriff auf den Transistor IGBT haben und deshalb nur seine Verluste über die Methoden des Ober-Interfaces ermitteln. Abb. 3.151 und 3.152 stellen zum einen die Realisierung des Interfaces *„Schaltverluste“* mithilfe des Transistors IGBT und zum anderen das Hinzufügen eines neuen Attributs, genannt *„ausschaltperiode“,* ins Interface *„Schaltverlust“*, wobei die Vererbungsbeziehungen zwischen dem Ober-Interface *„Schaltverlust“* und den Unter-Interfaces verdeutlicht sind. Aus den Abb. 3.151, 3.152 und 3.153 ist anzumerken, dass die Komponente IGBT die Realisierung des Interfaces *„Schaltverlust“* ermöglicht. Abhängigkeitsbeziehung, genannt *„Dependency“,* zwischen den Komponenten IGBT und Diode in Bezug auf die Funktionalität des Transistors zeigt Abb. 3.154. Hierbei ist anzumerken, dass die Freilaufdiode zwischen dem Emittler und dem Kollektor eingebaut ist und die Sperrfähigkeit des IGBT in Rückwärtsrichtung leitet. Das heißt, dass beide Komponenten IGBT und Diode eine Abhängigkeit darstellen. Abb. 3.150 gibt einen Überblick über eine Dreieck-Komponentenmodellierung mit IGBT, Diode und Lastkreis des Wechselrichters. Gemäß Abb. 3.150 hängt Diode sowohl von IGBT als auch von dem Lastkreis des Wechselrichters ab, aber IGBT hängt nur vom Lastkreis des *Wechselrichter* ab.

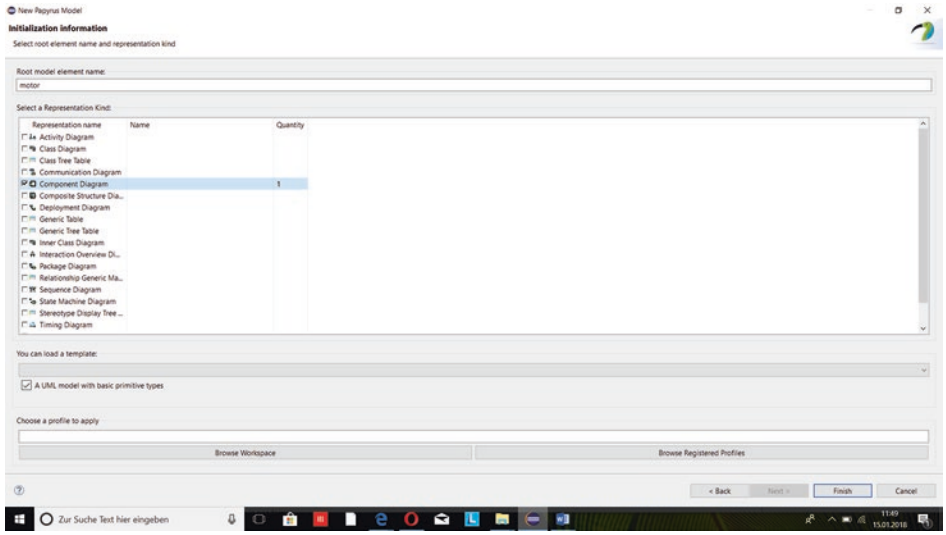


Abb. 3.141 Das Auswählen des Diagrammtyps *„Component Diagram“* für das Papyrus-Modell

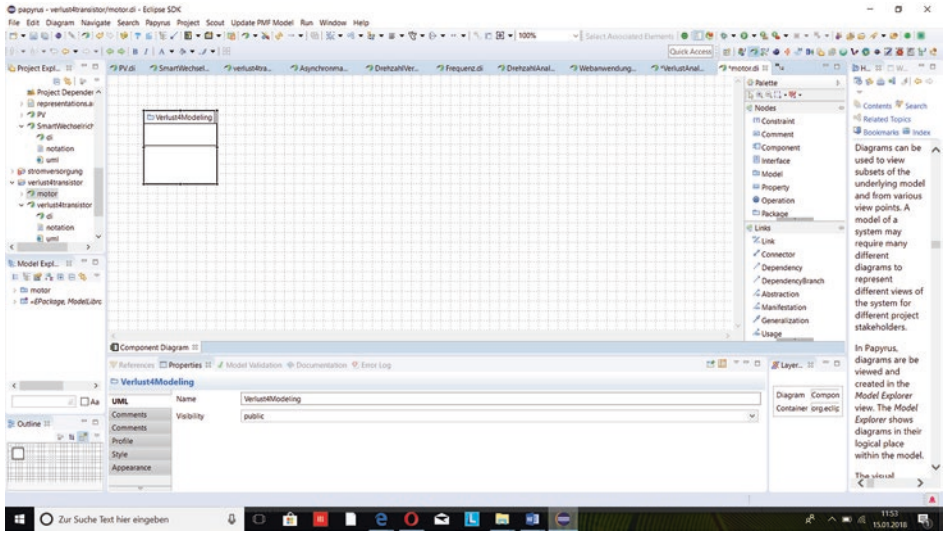


Abb. 3.142 Überblick über den Editor von Komponentendiagramm mit Hinblick auf das erstellte Package *“Verlust4Modeling”* und den Werkzeugkasten, genannt *„Palette“*

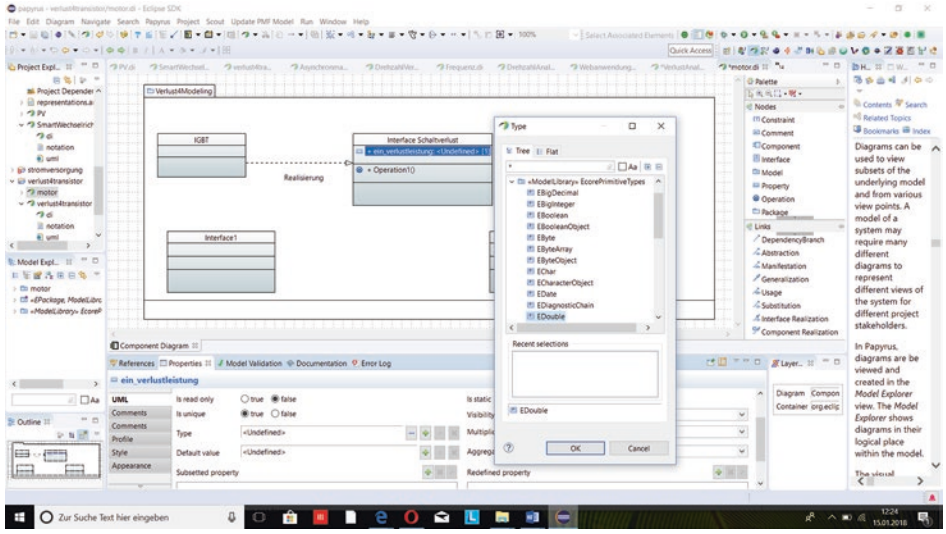


Abb. 3.143 Darstellung der Verbindung zwischen der Komponente *„IGBT“* und dem Interface *„Schaltverlust“* mit dem Hinzufügen der Eigenschaften des Interfaces

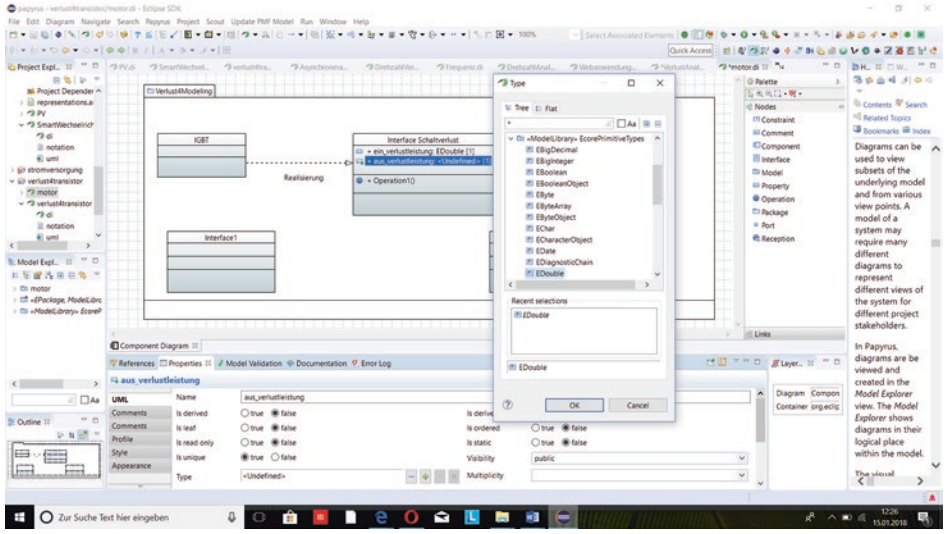


Abb. 3.144 Überblick über das Hinzufügen des Datentyps des Attributes *aus_verlustleistung* mithilfe von *„ModelLibrary“* EcorePrimitiveTypes

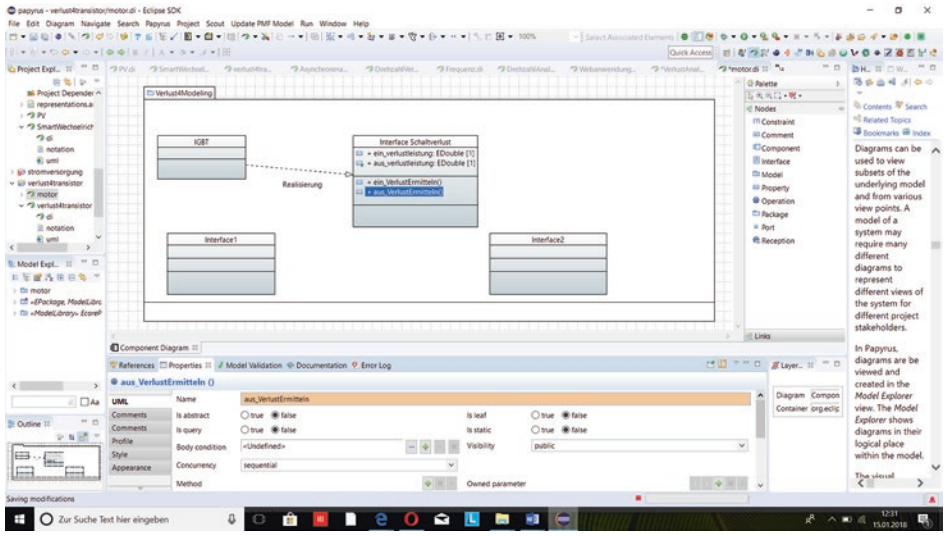


Abb. 3.145 Das Erstellen zweier neuer *Unter-Interfaces* und das Hinzufügen von zwei Operationen ins *Ober-Interface „Schaltverlust“*

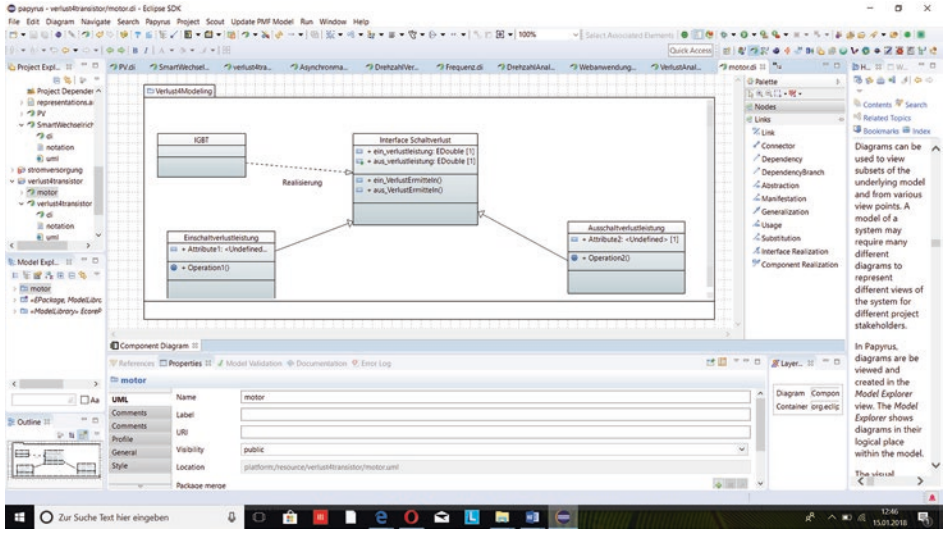


Abb. 3.146 Das Umbenennen neuer *Unter-Interfaces* nach *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“*

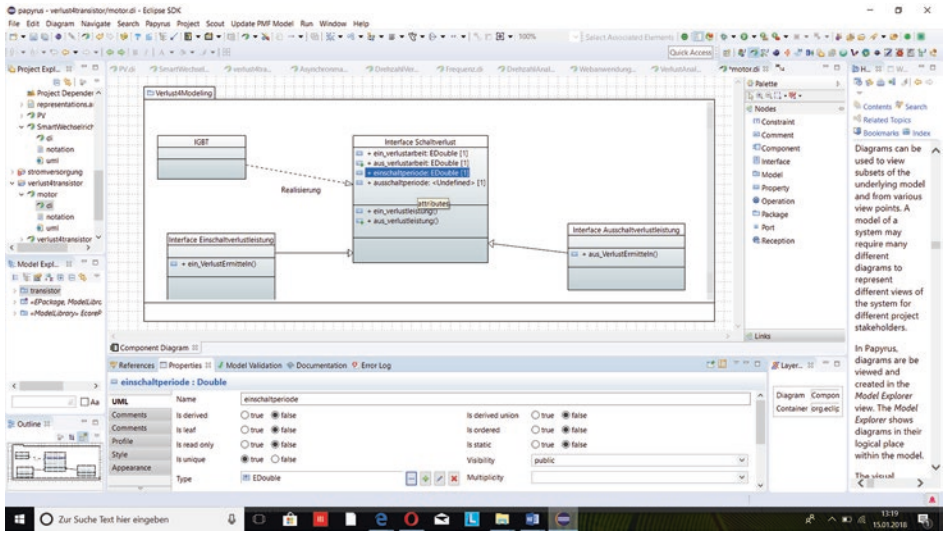


Abb. 3.147 Darstellung der Vererbungsbeziehungen zwischen dem Ober-Interface *„Schaltverlust“* und den Unter-Interfaces *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“*

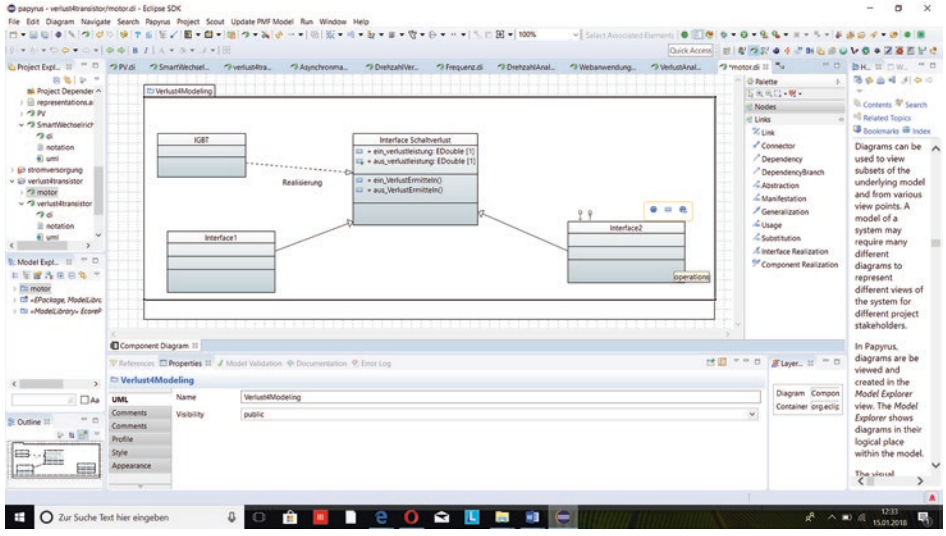


Abb. 3.148 Überblick über Kapselung von Zustand und Verhalten mithilfe von Beziehungen zwischen leeren Interfaces und Eigenschaften des Interfaces *„Schaltverlust“*

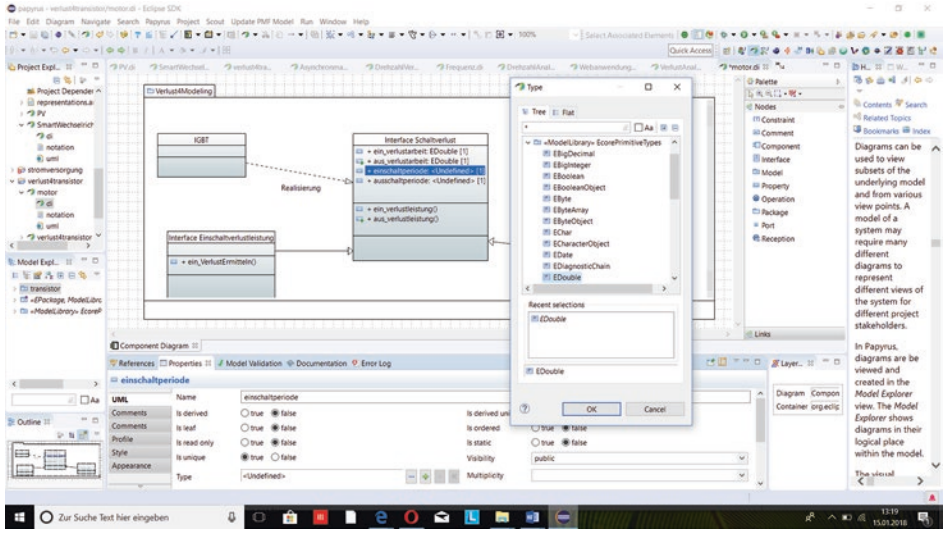


Abb. 3.149 Überblick über das Hinzufügen des Attributs *„einschaltperiode“* und dessen Datentyps mithilfe von *„ModelLibrary“ EcorePrimitiveTypes* und über das Umbenennen der Eigenschaft des Interfaces *„Einschaltverlustleistung“* nach *„ein_VerlustErmitteln()“*

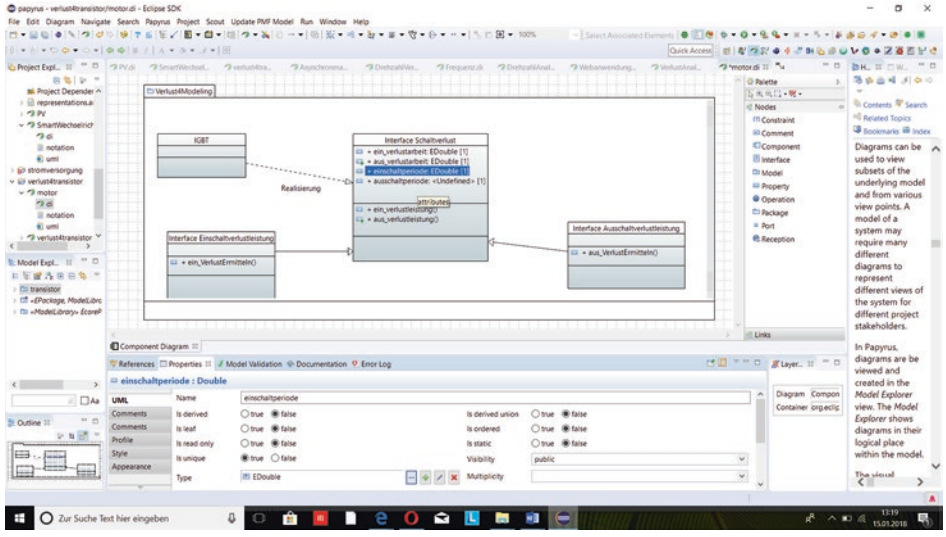


Abb. 3.150 Kompatibilität zwischen den Merkmalen von den Unter-Interfaces und dem Ober-Interface *„Schaltverlust“*

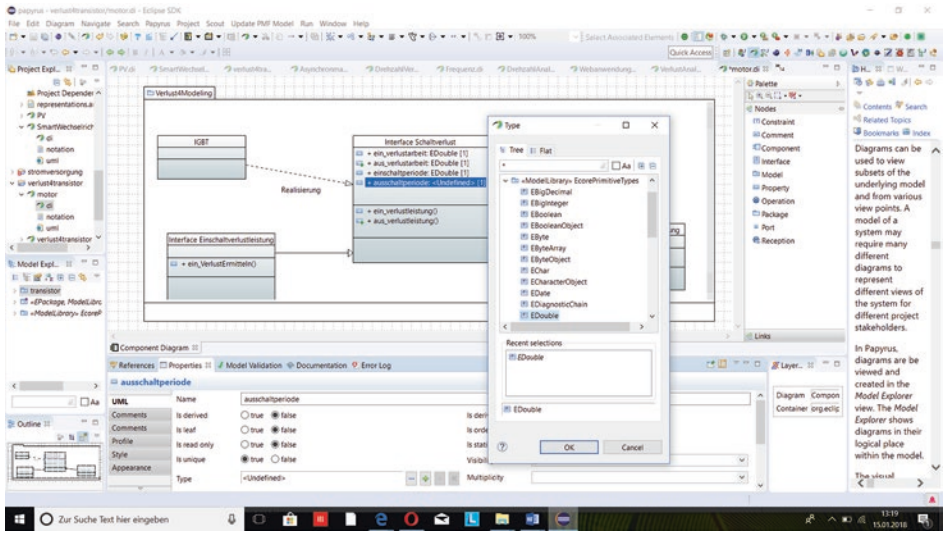


Abb. 3.151 Überblick über das Hinzufügen des Attributs *„ausschaltperiode“* und dessen Datentyps mithilfe von *„ModelLibrary“* EcorePrimitiveTypes und über das Merkmal des Interfaces *„Einschaltverlustleistung“* in Bezug auf die *Operation „ein_VerlustErmitteln()“*

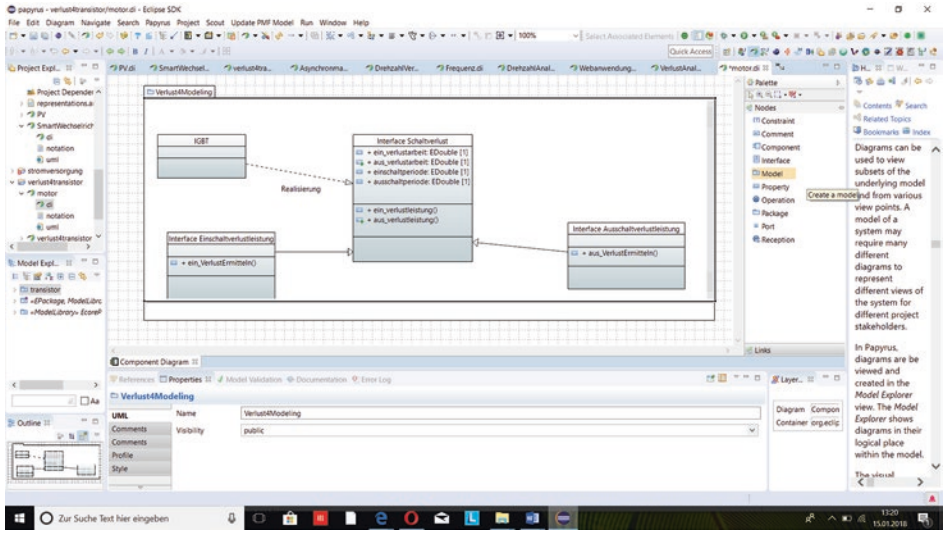


Abb. 3.152 Darstellung der Datentype der Attribute des Ober-Interfaces *„Schaltverlust” und der Kompatibilität zwischen Operationen „ein_VerlustErmitteln()“* oder *„aus_VerlustErmitteln()“* und *Eigenschaften* vom Interface *„Schaltverlust“*

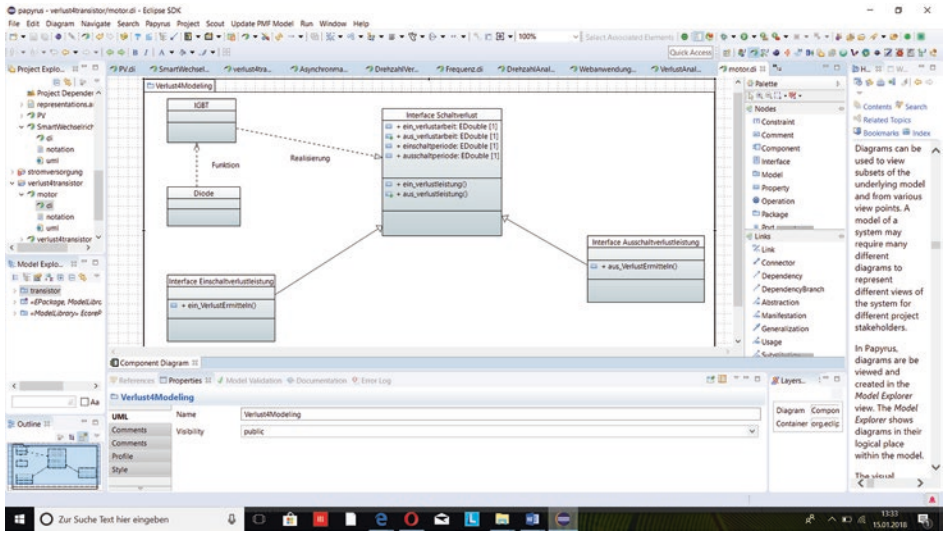


Abb. 3.153 Darstellung der Interaktion zwischen den Komponenten *IGTBT-Diode* und dem Vererbungsbaum *Interface-Interface-Interface*

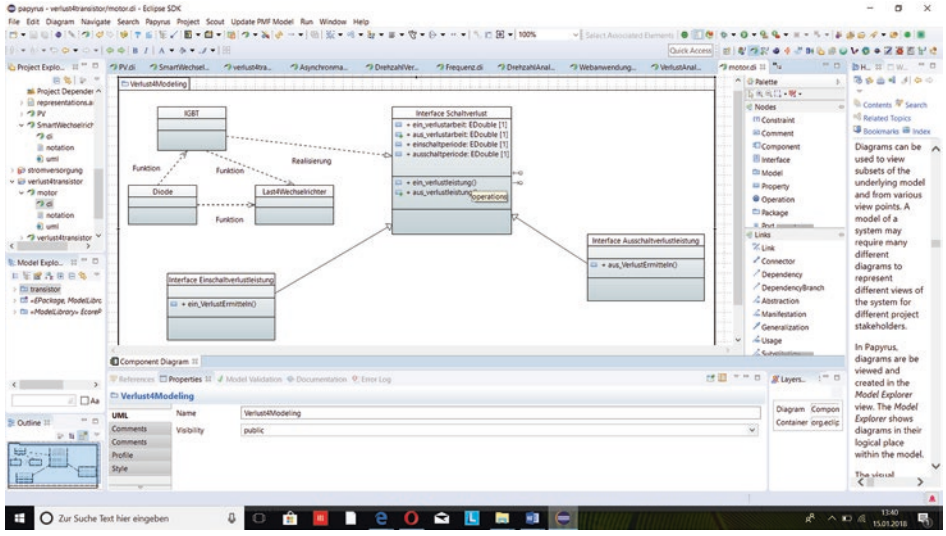


Abb. 3.154 Darstellung der Interaktion zwischen den Komponenten *IGTBT-Diode-Last4Wechselrichter* und dem Vererbungsbaum Interface-Interface-Interface

3.9 Zusammenfassung

Eclipse-Papyrus stellt ein Open-Source-Plugin für die Eclipse IDE der Eclipse Foundation zur Modellierung und Unterstützung modellgetriebener Entwicklungsansätze dar. Papyrus nutzt als Basis das Eclipse-Modeling-Framework. Neben UML und SysML werden auch andere Modellierungssprachen wie z. B. BMM unterstützt. Eclipse-Papyrus stellt mit UML 2 als Familie zur Modellierung zwei Arten von Diagrammen dar: 6 Diagramme zur Strukturmodellierung und 7 Diagramme zur Verhaltensmodellierung. Die erste Gruppe enthält folgende Diagramme: Klassendiagramm, Komponentendiagramm, Objektdiagramm, Package-Diagramm, Verteilungsdiagramm und Profile-Diagramm. Die zweite Gruppe verfügt über folgende Diagramme: Zustandsdiagramm, Aktivitätsdiagramm, Use-Case-Diagramm, Interaktionsdiagramme (Sequenzdiagramm, Kommunikationsdiagramm, Interaktion-Überblicksdiagramm, Timing-Diagramm). Klassendiagramme können sowohl zur Beschreibung eines schon in Code umgesetzten Programms als auch zur Modellierung eines Sachverhaltes vor der konkreten Umsetzung in einer Programmiersprache verwendet werden. Das Erstellen eines neuen Papyrus-Projektes erfolgt nach dem Installieren und Starten des Frameworks, wobei die Papyrus Perspective geöffnet ist. Mit einem Rechtsklick auf den *„Project Explorer -> New -> Papyrus Project“* wird im Wizard zunächst *UML* als *Software Engineering* für *Architecture Contexts* ausgewählt. Danach werden verschiedene Diagrammtypen der UML-Architektur vom Framework Papyrus angezeigt. Das Erstellen eines Klassendiagramms ist mithilfe des Auswählens *„Class Diagram“* ermöglicht. User Guide des Frameworks Eclipse-Papyrus gibt Information über Tutorial zum Erstellen von Diagrammtypen mit UML-Architektur.

Da Klassendiagramme den Baustein jeder objektorientierten Struktur darstellen, beruht die Analyse des Klassendiagramms auf Merkmalen der Klassen und deren Beziehungen zwischen ihnen. Eclipse-Papyrus verfügt über ein inneres Klassendiagramm, welches die konkrete Struktur zur Modellierung darstellt. Normalerweise bestehen Klassen aus drei Bereichen: oberer, mittlerer und unterer Bereich. Der Erste verfügt über den Stereotyp, genannt Paket, zu dem die Klasse gehört und den Namen. Im Zweiten sind die Attribute, genannt *„Properties“*, angegeben und im Unteren gibt es die Methoden, genannt Operationen, der Klasse. Laut UML-Spezifikation kann die Darstellung einer Klasse zusätzliche Bereiche enthalten. Eine Assoziation stellt eine Beziehung mittels Linien zwischen Klassen dar. Eine Linie zwischen den Klassen stellt eine Assoziation dar. Die Generalisierung zeigt eine Vererbung zwischen Klassen. Die Vererbungen ermöglichen das Veranschaulichen von Beziehungen zwischen einer Ober- und Unterklasse und sind Attribute und Operationen der Oberklasse in den Unterklassen zugänglich gemacht. Die Vererbung ist das klassische Programmierkonzept, welches in den objektorientierten Programmiersprachen verwendet wird. Eine Reaktionskette der

Vererbung mittels hierarchischer Vererbung stellt den Begriff *„Vererbungskaskade“* dar. Hierbei sind mehrere Vererbungsschichten im Klassendiagrammen zu sehen.

Das Paketdiagramm zum Beschreiben des Strukturdiagramms mit dem Framework Eclipse-Papyrus stellt die Zusammenfassung von Modellelementen dar. Hierbei werden Elemente wie z. B. Klassen strukturiert modelliert, damit man einen Überblick über die Komponente der Software oder des Systems hat. Da Pakete Modellelemente zu Gruppen zusammenfassen, können sie für verschiedene Zwecke eingesetzt werden. Pakete können verwendet werden, um Subsysteme zu modellieren, in die ein großes System zerlegt wird. Pakete können auch verwendet werden, um die Ergebnisse von Entwicklungsphasen festzulegen und die entsprechenden Diagramme, die einer Phase zugeordnet werden, zusammenzufassen. Darüber hinaus können Pakete für jede sinnvolle Gruppierung von Modellelementen in einem Projekt eingesetzt werden. Model View Controller ist ein Design Pattern zum Darstellen von Anwendungen in getrennten Klassen. Hierbei ist das Paketdiagramm eine Darstellung des Design Patterns MVC, welches die strickte Teilung von Präsentation (View), der Programmlogik (Controller) und der Datenschicht (Model) ermöglicht. Diese vereinfacht die Lesbarkeit und Wartbarkeit des Codes. Mithilfe des Design Patterns MVC kann man ein Projekt viel einfacher um weitere Funktionen ergänzen, ohne den halben Sourcecode durchzuschauen. Das Designen der Oberflächen mit Java XDEV 5 basiert auf Java Swing, das seit 1997 als Standard-Bibliothek für die Entwicklung grafischer Oberflächen in Java gilt und auch heute noch mit gutem Grund von vielen Java-Entwicklern für die Anwendungsentwicklung eingesetzt wird. Swing ist beliebt, ausgereift und stark. In XDEV 5 stehen den Entwicklern viele GUI-Controls zur Verfügung, die Swing bietet, u. a. Buttons, Formular-Komponenten, Fenster und Dialoge, Tabs, Menüleisten und Kontextmenüs sowie eine leistungsfähige Tabellen- und Tree-Komponente. Java Swing stellt einen Umfang an Controls für die Entwicklung grafischer Oberflächen (GUI = Graphical User Interface) zur Verfügung wie z. B. Buttons, Labels, Tabs, sämtliche Formular-Controls, Table, Tree sowie Fenstertechniken. Viele Standard-anwendungen lassen sich damit umsetzen.

Java-Framework XDEV stellt sowohl eine visuelle Entwicklung als auch Java-Coding dar. Dafür verfügt das Framework XDEV über einen Java-Code-Editor mit Funktionen wie z. B. Code-Vervollständigung, Refactoring, Debugger und Versionsverwaltung. Bei XDEV setzt sich das Rapid-Application-Development-Konzept auch beim Code-Editor fort. Damit ist die Java-Programmierung mit dem Framework Java XDEV 5 einfach.

Klassen-Bäume-Tabelle, genannt *Class Tree Table,* ist eine UML-Tabelle vom Famework Eclipse-Papyrus. Diese Tabelle fokussiert auf die Darstellung der Klassen und ihren Attributen und Operationen mithilfe von Spalten und Zeilen. Die Struktur dieser Tabelle vom Eclipse-Papyrus ähnelt dieser von Excel-Tabellen. Mithilfe dieser Tabelle sind die Klassen als Bäume-Diagramm dargestellt. Wer mit Kalkulationstabelle Spalten und Zeilen ausgefüllt hat, wird keine Schwierigkeiten beim Modellieren der Klassen mit dem „Class Tree Table“ haben. Diese Tabelle stellt eine Erleichterung zum Organisieren der Eigenschaften der Klassen. Hierbei sind Klassen, Attribute und Operation nach Nummerierung gelistet. Die Tabelle gibt einen Überblick über das gesamte Klassensystem

des Modells. Diese Tabelle zeigt die Struktur der Klassen hinsichtlich ihrer vertikalen und horizontalen Position. Klassen-Baum-Tabelle geben Informationen mithilfe Tabellen über die vertikale Position. Das Erstellen dieser Tabelle mit dem Framework Eclipse-Papyrus beginnt mit dem Projekt-Erstellen. Die Tabelle für die vertikale Position stellt eine geometrische Position in Bezug auf Spalten und Zeilen. Die Klassen sind gelistet nach ihren Positionsnummern in der Tabelle. Dies entspricht einer vertikalen Orientierung in der Tabelle. Diese Orientierung folgt einer vertikalen Achse. Die Klassen in der vertikalen Position sind von oben nach unten eingerichtet. Die Klassen in der vertikalen Position sind als Rechtecke mit einer Bezeichnung dargestellt. Die horizontale Position in der *„Class Tree Table“* ermöglicht die Darstellung der Eigenschaften der Klassen hinsichtlich ihrer Attribute und Operationen. Diese geometrische Orientierung stellt die horizontale Achse eines Cartesischen Systems dar. Die horizontale Position ermöglicht die Darstellung der Klassen nach der Position von *„ownedAttribute“* und *„ownedOperation“*. Wobei die Verteilung der Klassen nach der Ausdehnung der Zeilen dargestellt sind.

Sequenzdiagramme ermöglichen die dynamische Modellierung bezüglich der Interaktionen zwischen Teilen des Systems. Mit dem Framework Eclipse-Papyrus fokussieren die Sequenzdiagramme auf die Lebenslinien der Objekte und deren Interaktionen mit anderen Objekten. Bei Sequenzdiagrammen handelt es sich um eine Art von Interaktionsdiagramm, denn sie beschreiben, wie und in welcher Reihenfolge eine Gruppe von Objekten zusammenarbeitet. Software-Entwickler und Unternehmen greifen auf diese Diagramme zurück, um zu verstehen, welche Anforderungen ein neues System stellt oder um bereits bestehende Prozesse grafisch festzuhalten. Sequenzdiagramme werden manchmal auch als Ereignisdiagramme oder Ereignisszenarien bezeichnet. Sequenzdiagramme beschreiben die Kommunikation zwischen Objekten in einer bestimmten Szene. Es wird beschrieben, welche Objekte an der Szene beteiligt sind, welche Informationen (Nachrichten) sie austauschen und in welcher zeitlichen Reihenfolge der Informationsaustausch stattfindet. Sequenzdiagramme enthalten eine implizite Zeitachse. Die Zeit schreitet in einem Diagramm von oben nach unten fort. Die Reihenfolge der Pfeile in einem Sequenzdiagramm gibt die zeitliche Reihenfolge der Nachrichten an.

Kommunikationsdiagramme stellen die Interaktionen zwischen den Objekten oder Rollen dar, die Lebenslinien zugeordnet sind, sowie die Nachrichten, die zwischen Lebenslinien übergeben werden. Kommunikationsdiagramme sind ein Interaktionsdiagrammtyp, mit dem das dynamische Verhalten eines Systems oder einer Softwareanwendung untersucht werden können. Sie zeigen eine alternative Sicht derselben Informationen wie in Sequenzdiagrammen bereit. In Sequenzdiagrammen liegt der Fokus auf der zeitlichen Reihenfolge der Nachrichten, in Kommunikationsdiagrammen dagegen auf der Struktur der Nachrichten, die zwischen den Objekten in der Interaktion übergeben werden. Diese Diagramme veranschaulichen den Nachrichtenfluss zwischen Objekten und die implizierten Beziehungen zwischen Klassen. Kommunikationsdiagramme stellen Aspekte einer Interaktion für Objekte, Schnittstellen der beteiligten Klassen, Strukturänderungen und Übergabe der Daten der Objekte dar. Kommunikationsdiagramme sehen ähnlich aus wie Objektdiagramme, bei denen

eine Lebenslinie die Objekte in der Interaktion darstellt und Pfeile für die Nachrichten stehen, die zwischen den Lebenslinien übergeben werden. Das Kommunikationsdiagramm ist mithilfe des Frameworks Papyrus über „New->New Project“ erstellt. Die Linien zwischen Klasseninstanzen stellen die Verbindungen zwischen verschiedenen Komponenten der Websicherheitsanwendungen dar. Die Pfeile stehen für gesendete Nachrichte zwischen Objekten oder Rollen.

Objektdiagramme ermöglichen die Modellierung der Objektstruktur. Ein Objektdiagramm ist ein Klassendiagramm mit dem Ziel, die Darstellung der Objekte bezüglich der Struktur der Datentype zu realisieren. Hierbei werden Objekte nach ihren Datentypen spezifiziert. Das Objektdiagramm gibt Überblicke über die Werte der Objekte in Bezug auf die statische Struktur des Diagramms. Mithilfe eines Objektdiagramms wird das Analysemodell konkret, präzise und einfach. Ein Objektdiagramm stellt eine Klassifikation der Objekte mithilfe von Attributen, Operationen und Beziehungen dar. Ein Objektdiagramm beschreibt eine Sammlung von Objekten und deren Beziehungen zu einem Zeitpunkt im Leben eines Systems. Objektdiagramme sind daher auf Instanzen fokussiert und haben einen exemplarischen Charakter. Ein Objekt ist Instanz einer Klasse und enthält die von ihr festgelegten Attribute. Diese Attribute sind mit einem Wert initialisiert. Im Objektdiagramm werden prototypische Objekte verwendet, um exemplarische Situationen zu illustrieren. Zwischen den im Diagramm sichtbaren prototypischen Objekten und den echten Objekten des Systems besteht normalerweise keine 1:1-Beziehung. Ein Attribut beschreibt eine Zustandskomponente eines Objektes. Ein Attribut im Objektdiagramm ist charakterisiert durch den Attributnamen, den Typ und einen konkreten Wert. Weitere Charakteristika wie Sichtbarkeit können dem Attribut angeheftet werden. In abstrakten Objektdiagrammen können statt konkreten Werten auch Variablennamen oder Ausdrücke eingesetzt werden, deren Inhalt im Diagramm „unspezifiziert“ bleibt. Attributtyp oder -wert können auch ausgelassen werden. Klassen bestehen aus Attributen und Methoden, die den Zustand und das Verhalten ihrer Instanzen (Objekte) festlegen. Klassen sind durch Assoziationen und Vererbungsbeziehungen miteinander verknüpft. Klassennamen erlauben es, die Klassen zu identifizieren. Hierbei stellen Attribute mithilfe der Namen und Typen die Zustandskomponente der Klassen dar. Die Klassenmethoden mithilfe der Signaturen und Rumpf implementieren die Funktionalitäten der Klassen. Bei Objektdiagrammen geht es in erster Linie um die Attribute einer Reihe von Objekten und wie diese zueinander in Beziehung stehen. Objektdiagramme sind mithilfe eines Dreieck-Symbols, genannt *„Instance Specification“,* aus der Palette des Editors des Frameworks Eclipse-Papyrus erstellt, wie es auf den Abb. 3.82, 3.83, 3.84, 3.85, 3.86, 3.87, 3.88, 3.89, 3.90, 3.91, 3.92, 3.93, 3.94, 3.95, 3.96, 3.97, 3.98, 3.99, 3.100, 3.101, 3.102, 3.103, 3.104, 3.105, 3.106, 3.107, 3.108, 3.109, 3.110, 3.111, 3.112, 3.113, 3.114, 3.115, 3.116 und 3.117 zu sehen ist. Sie bestehen aus Objekten wie z. B. *Einschaltverlustleistung oder Ausschaltverlustleistung,* die durch Rechtecke dargestellt und mit Linien verbunden werden, wobei Objektdiagramme von UML die Beschreibung der Instanzen darstellen. Ein Objektdiagramm verfügt über folgende Elemente: Objekte, Klassentitel, Klassenattribute und

Modellierungtools. Objektdiagramme ermöglichen die Darstellung der Modellierung eines Systemausschnittes aus dem Systemdiagramm. Es gibt Ähnlichkeiten zwischen dem Objekt- und dem Klassendiagrammen. Interaktionen zwischen Objekten und Klassen gehören zum Konzept von Klassendiagrammen. Objekte stellen die Instanzen der Klassen dar.

Das Kompositionsstrukturdiagramm, genannt *„Composite Structure Diagram“,* ermöglicht die Darstellung der internen Struktur von Komponenten sowie anderen Modellelementen wie z. B. Klassen. Ziel der Modellierung mit Kompositionsstrukturdiagrammen ist es, Beziehungen zwischen den Bestandteilen einer Struktur zu modellieren. Dieser Diagrammtyp ist deshalb zur Erstellung einer Softwarearchitektur geeignet. Die Bestandteile einer Komponente bzw. Klasse heißen Parts. Das Kompositionsstrukturdiagramm dient der Dekomposition und Modellierung von „Klassifikatoren“, die in Kompositionsbeziehungen zueinanderstehen. Es zeigt, wie die Teile eines Klassifikators den Klassifikator selbst bilden. Kompositionsstrukturdiagramme ermöglichen die Darstellung der Teile (Architekturelemente) eines Systems und deren Beziehungen. Deshalb werden sie auch Architekturdiagramme genannt und eignen sich ebenfalls zur Modellierung von Entwurfsmustern. Weil Klassen Klassifikationen der Objekte mithilfe von Attributen, Operationen und Beziehungen mit anderen Objekten darstellen, verfügen Klassifikatoren über Instanzen dieser Klassen, wobei Klassifikatoren Instanzen mit gemeinsamen Eigenschaften beschreiben. Spezielle Klassifikatoren sind sowohl Interface als auch Klassen. In Bezug auf die strukturelle Einordnung ist der Begriff „Klassifikator“ für die Gestaltung der Kompositionsstrukturdiagramme wichtig. Hierbei enthält der Klassifikator Instanzen der anderen Klassen. Das bedeutet, dass Klassen nach ihren strukturellen Eigenschaften eingeordnet werden. Ein Konnektor verbindet zwei Parts wie z. B. *„Einschaltverlustleistung“* und *„Ausschaltverlustleistung“* oder *„Objekte“* mithilfe von Eingängen, genannt *„Ports“*.

Komponentendiagramme stellen die Verbindung zwischen den Komponenten zum Analysieren der Komponenten- oder Softwaresysteme dar. Das Ziel der UML-Modellierung mit den Komponentendiagrammen ist es, die Abhängigkeiten jeder Komponente im System zu repräsentieren. Komponenten sind erforderlich, damit Stereotyp-Funktionen ausgeführt werden können. Wobei Komponenten-Stereotypen mithilfe der ausführbaren Programme, Dokumente, Datenbanktabelle, Dateien oder Bibliotheksdateien erzeugt werden können. Komponenten stellen sowohl die Kapselung von Zustand und Verhalten derer Elemente als auch die Spezifizierung einer Verbindung mit der Umgebung dar. Wie Klassen verfügen Komponenten über Instanzen, wobei sie aus Klassen bestehen. Komponentendiagramme verfügen über zum einen exportierte Schnittstellen und zum anderen importierte Schnittstellen, welche Verbindungen zwischen Komponenten und Umgebungen ermöglichen. Dies bedeutet auch, dass Komponenten ausführbare und austauschbare Softwareeinheit mithilfe von Schnittstellen darstellen.

Verbindungen zwischen Komponenten und Interface ermöglichen das Analysieren der Funktionalitäten der Systeme zum einen und zum anderen der objektorientierten

Modellierung mithilfe des UML-Diagramms. Hierbei sind Interfaces wichtige Verbindungen zu den Komponenten zum Darstellen von Anwendungen in Bezug auf Technik und Informatik. Das Praxisbeispiel fokussiert auf die Analyse der Verluste im Transistor *„Insulated Gate Bipolar Transistor“* mithilfe der Informatik. Das Beispiel stellt die Verbindung zwischen dem Transistor und dem Interface *„Schaltverlust“* dar.

Kapselung stellt die Abhängigkeit von Objekten in einer Komponente mithilfe der objektorientierten Analyse dar. Die Analyse der Funktionalität einer Kapselung wird in diesem Abschnitt als Darstellung der Beziehungen zwischen Elemente bezeichnet.

Literatur

1. Eclipse Web Portal: Papyrus Modeling einvironment, Eclipse Foundation, Eclipse, https://eclipse.org/papyrus
2. Igler,B., Krümmel, N., Ried, M., Renz, B.: Darstellungen von Konzepten für die Softwareentwicklung, Institut für SoftwareArchitektur, Technische Hochschule Mittelhessen, Campus Hessen (2017)
3. Informatik Labor: Web Portal, Fachbereich Informatik, Uni. Darmstadt, https://www.fbi.h-da.de/labore/case/uml/kompositionsstrukturdiagramm.html (2010)
4. Heinrich, M., G., Vogler, I.: Klassendiagramm in: Objekt-orientierte Analyse und Design, Teil G Arbeitserzeugnisse, Version N.3, Web Portal, Atlassian, https://wiki.marisma.net/display/OAOD/Klassendiagramm (2012)
5. Gross, S.: Tutorial Model View Controller MVC: Struktur in Java Projekten nutzen, In: BigBasti Blog, Web Portal, http://blog.bigbasti.com/tutorial-model-view-controller-mvc-struktur-in-java-projekten-nutzen/ (2010)
6. XDEV 5 Web Portal, XDEV Software Corporation, San Francisco, USA, http://www.xdev-software.de/xdevide/tools/guibuilder.html (2019)
7. Lucichart, Was ist ein UML-Diagramm, Web Portal, Lucid Software Inc, https://www.lucidchart.com/pages/de/was-ist-ein-uml-diagramm (2019)
8. Rumpe, B.: Agile Modellierung mit UML., 2. Auflage, Springer Verlag (2011)
9. Kommunikationsdiagramm mit IBM Rational, in: Kommunikationsdiagramm erstellen, IBM Knowledge Center, Web Portal https://www.ibm.com/support/knowledgecenter/de/SS8PJ7_9.6.1/com.ibm.xtools.sequence.doc/topics/ccommndiag.html (2019)
10. Schadow, D.: Kopfsache für mehr Sicherheit in Webanwendungen, in: Java Magazin, Ausgabe 11.14, Software & support Media GmbH Verlag, Frankfurt a. M. (2014)
11. Taentzer, G.: Vorlesung für UML in: Softwaretechnik, Universität Marburg (2015)
12. Sparx Systems Web Portal: Object Diagrams In: Tutorials for the Unified Modeling Language (UML), https://sparxsystems.com/resources/tutorials/uml2/object-diagram.html, Sparx Systems Pty Ltd, (2019)
13. Oestereich, B.: Analyse und Design mit UML 2, Oldenbourg Wissenschaftsverlag, ISBN 3–486-57926-6 (2006)
14. Rost, D., Naab, M.: Softwarearchitekturen einfacher designen und verständlicher dokumentieren mit dem Fraunhofer ADF, Fraunhofer IESE Web Portal, https://blog.iese.fraunhofer.de/ (2018)
15. Matevska, J.: Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit, Vieweg+Teubner Verlag, 1. Auflage (2010)

SysML-Modellierung mit Eclipse-Papyrus

4

Die Sprache SysML ist eine von der Object Management Group (OMG) standardisierte Modellierungssprache für die Spezifikation, die Analyse, das Design und die Verifikation und Validierung von Systemen und deren Systemelemente wie beispielsweise Software, Hardware, Informationen, Prozesse, Personen und Gegenstände [1]. Dabei dient ein Teil der UML als Sprachkern, welcher um zusätzliche Sprachelemente erweitert wurde. Eine Systemanalyse ist für einige Systeme bereits auf Grundlage der Modelle durchführbar. Viele Systeme lassen sich allerdings erst durch eine Simulation hinsichtlich ihrer Untersuchungskriterien bewerten. Ausgehend von einem Systemmodell wird mithilfe von Konfigurationseinstellungen und der Übergabe von Werten an die äußere Schnittstelle des Systems die Veränderung des Systemzustandes im Verlauf der Zeit beobachtet. Die Modellgrößen in einem System, die sich als Zustandsgrößen identifizieren lassen, können sich dabei grundsätzlich zeitkontinuierlich oder zeitdiskret verändern [2].

SysML zur grafischen Modellierungssprache der Systemmodellierung stellt Unterstützungen bei der Analyse, Spezifikation, Design, Verifikation und Validation von Systemen dar, die aus Hardware, Software, Daten, Personal, Prozessen und technischen Hilfsmitteln (Anlagen) bestehen. Es gibt verschiedene Perspektiven zur Modellierung mit SysML unter anderem Anforderungen, statische Architektur, dynamische Architektur und Interaktionen. Anforderungen, Struktur, Verhalten und Zusicherung stellen die vier Bausteine der SysML-Modellierung dar.

Mithilfe des Frameworks Papyrus werden Struktur-, Verhaltens-, Anforderungs- und Parametermodellierung können verschiedene Aspekte eines Systems mit SysML modelliert.

Die Strukturmodellierung mithilfe von Blockdefinitionsdiagramm (BDD) und internem Blockdiagramm (IBD) ermöglicht die Modellierung eines Systems bezüglich seiner strukturellen Eigenschaften. Die möglichen Systemelemente in einem realen System werden beschrieben und der strukturelle Aufbau eines Elementes und Verbindungen und Vererbungsbäume zwischen ihnen werden festgelegt.

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML,*
https://doi.org/10.1007/978-3-658-29057-3_4

Tab. 4.1, 4.2 und 4.3 zeigen die Funktionalitäten der Systemmodellierung mit SysML bezüglich des Aufbaus der Diagramme. Tab. 4.1 gibt einen Überblick über die verschiedenen Modellebenen. und deren Anwendungen zur Systemmodellierung mit SysML. Zum einen ermöglichen Modellebene zur funktionelle Modellierung die Entwicklungen von Schnittstellen von IT und Technik mithilfe objektorientierter Programmierungen wie z. B. Java oder C++ und zum anderen stellen sie für Systeme bzw. Komponenten Anwendungen für Design bzw. Software-Komponenten wie z. B. Java EE 8. Tab. 4.2 zeigt die Struktur von SysML in Bezug auf die Taxonomie. Hierbei gibt es vier Diagrammtype für Struktur-, Verhalten-, Anforderungs- und Zusicherungsmodellierungen. Die anderen UML-Diagramme stellen die Schnittstellenpunkte zwischen UML und SysML dar. Tab. 4.3 fokussiert auf die Funktionalität des Systems Engineering bezüglich der Anforderungen, des Designs, der Integration und der Komponenten. Diese Funktionalität ermöglicht die Modellierungen von System Spezifikation und Design; System Integration und Tests; Komponenten Design, Integration und Tests.

Tab. 4.1 Modellebene zur Systemmodellierung mit SysML

Modellebenen	Anwendungen
Funktionelle Modellen	Schnittstellen von IT und Technik
Systemmodelle	System Design
Komponentenmodelle	Java EE 8

Modellierungen bei mehreren Ebenen der Systeme

Tab. 4.2 Taxonomie von SysML-Diagramme

Strukturelle Diagramme	Verhaltensdiagramme	Anforderungsdiagramm	Zusicherungsdiagramm
Block Definition Diagramm	Activity Diagram	Requirement Diagram	Parametric Diagram
Internes Block Diagramm	UML-Diagramm		

Diagramme von SysML nach UML-Modifikation

Tab. 4.3 System Engineering

System Spezifikation und Design	System Integration und Tests	Komponenten Design, Integration und Tests
Anforderungssysteme	Integration	Geprüfte Komponenten
Anforderungskomponenten	Test Feedback	Design Feedback

Systemlösungen für die Modellierung mit SysML

4.1 Blockdefinitionsdiagramm

Blöcke definieren Mengen von Eigenschaften und stellen Systemelementtyp-Definitionen dar. Die Eigenschaften können sowohl Struktur (Property) als auch Verhalten (Operation) sein. Blöcke können auch Blackboxes darstellen, deren Eigenschaften nicht verfügbar sind, sondern nur die Schnittstellen des Blocks als Interaktionspunkte mit dem restlichen System bekannt sind [2].

Blockdefinitionsdiagramme zeigen die Definition von Systembausteinen und ihre Beziehungen. Sie sind wichtige Diagramme aus der Systemmodellierung und haben ihre Ursprünge in der UML, wobei sie die *„Klassendiagramme“* darstellen. Bei der SysML-Entstehung wurden Klassendiagramme in Blockdefinitionsdiagramme transformiert.

4.1.1 Aufbau von Blockdefinitionsdiagrammen

Blockdefinitionsdiagramme stellen zwei dimensionale geometrische symbolische Darstellungen von Informationen dar. Dreidimensionale Visualisierungen dienen der Projizierung zweidimensionaler Fläche. SysMlL-Modellierung ermöglicht die Darstellung von Hierarchien und Verbindungen zwischen einzelnen Komponenten in einem Systemdesign mithilfe von Blöcken. In einem Blockdefinitionsdiagramm können Beziehungen zwischen Blöcken wie z. B. Komposition, Assoziation und Spezialisierung beschrieben werden [3].

Die Interaktionen zwischen Blöcken werden durch Linien und Pfeile realisiert. Entsprechende Diagramme können bis auf einzelne Komponenten oder elektronische Schaltungen hinunter gebrochen werden.

Blockdefinitionsdiagramme bestehen auch aus Operationen, welche Verhalten von Blöcken beschreiben. Verhaltensdefinitionen bei SysML sind wie bei UML modelliert.

Tab. 4.4 gibt einen Überblick über die Funktionalität der BDD im Hinblick auf Konzept und Beschreibung [4]. Hierbei sind die Funktionen der Werkzeugtools der BDD beschrieben für die Systemmodellierung.

4.1.2 Erstellung von Blockdefinitionsdiagrammen (BDD) mit Eclipse-Papyrus

Mithilfe von Eclipse-Papyrus sind BDD erstellt worden. Diese Praxisbeispiele fokussieren auf den Bereich Energietechnik-Informatik. Blockdefinitionsdiagramme stammen aus dem UML-Klassendiagramm und stellen zum einen Block- und Wertetyp-Definitionen und zum anderen die Assoziationen zwischen Blöcken und Propertys dar. Ein Property beschreibt eine strukturelle Eigenschaft eines Blocks. Ein Property verfügt über mehrere Elemente wie z. B. Part-Property, Reference-Property, Value-Property, Constraint-Property und Port-Property.

Tab. 4.4 Funktionalität der BDD im Hinblick auf Konzept und Beschreibung

Concept	Description
BLOCK DEFINITION DIAGRAM	The diagram as a whole, it consist of 1 or more BLOCKS.
BLOCK	The main modular unit in a block definition diagram, a BLOCK has several specializations. Zero too many BLOCK have one to many ASSOCIATION. The only properties it has is Name.
UNIT BLOCK	An UNIT BLOCK is a specialization of the BLOCK, it gives an identification of a physical quantity. It has the property Quantity Kind.
VALUETYPE BLOCK	A VALUETYPE BLOCK is a specialization of the BLOCK, provides a uniform definition of quantity. It has the properties Values, Operations and ValueType
BASIC BLOCK	A BASIC BLOCK is a specialization of the BLOCK, it is the most basic version of the BLOCK with no special meaning. It has the properties Parts, References and Values
ENUMERATION BLOCK	An ENUMERATION BLOCK is a specialization of the BLOCK, it gives a definition for a set of values. It has the property EnumerationLitera
ACTOR BLOCK	An ACTOR BLOCK is a specialization of the BLOCK, it indicates this block, is an actor and has no special properties
ASSOCIATION	Set of lines used to create different kinds of relations between BLOCKS and the other concepts. Multiplicities at both ends can be used. Also an association can be given a name.
COMPOSITE	A COMPOSITE ASSOCIATION is a whole-parts relation
GENERALIZATION	A GENERALIZATION ASSOCIATION is a way to indicate a specialization of a certain concept. The arrow is pointing to the parent.

Beschreibungen der Blockdefinitionsdiagramme [4]

Ein Property bezeichnet man als Part-Property, wenn es eine Kompositionsbeziehung zu einem anderen Classifier definiert. Besteht zwischen dem Property und dem Block jedoch eine Aggregationsbeziehung, spricht man von einem Reference-Property. Wenn es sich bei dem Property um die Verwendung eines Wertetyps handelt, nennt man es Value-Property. Ein Property kann auch die Verwendung eines Constraint-Blocks sein und ist dann ein Constraint-Property. Ein Port ist ein Interaktionspunkt für einen Block (UML-Metaklasse Port). SysML definiert die Erweiterung *„FlowPort“*. Ein Flow-Port dient dem Austausch von Elementen und kann atomar sein und nur einen Interaktionstyp und -richtung festlegen oder mit einer Flussspezifikation („FlowSpezification“) verschiedene Interaktionstypen und Richtungen definieren. Die Angabe des Typs des Flow-Propertys spezifiziert den Interaktionstyp und die -richtung. Falls der Typ eine Interface-Definition ist, können mit dem Stereotyp (*„FlowSpecification“*) für die UML-Metaklasse Interface mehrere Interaktionstypen

mit -richtung spezifiziert werden. Eine Flussspezifikation besitzt hierfür besondere Propertys, die Flow-Propertys (*„FlowProper- ty“*). Ein Flow-Property hat das Property direction, welches die Richtung (in, out oder inout) festlegt. Da *„FlowProperty“* ein Stereotyp der UML-Metaklasse Property ist, kann der Typ des Propertys an dem Interaktionspunkt ausgetauscht werden.

4.1.2.1 Praxisbeispiel: Modellierung der Schaltung vom Schwingkreiswechselrichter

Modellierungen der Schaltung des Schwingkreiswechselrichters beschreiben die Funktionalitäten der abschaltbaren Leistungshalbleiter wie z. B. Transistoren (IGBT) bezüglich der Resonanzelemente. Hierbei besteht die Schaltung des Schwingkreiswechselrichters aus Gleichspannung, Stützkondensator, IGBT mit Dioden und Lastkreis mit Spule, Widerstand und Kondensator. Die Modellierung der Schaltung ist mithilfe des Frameworks Eclipse-Papyrus realisiert worden. Diese Schaltung stellt diese eines Resonanzstromrichters mithilfe des Lastkreises und der Verwendung der abschaltbaren Leistungshalbleiter wie z. B. IGBT dar. Ziel der Modellierung der Schaltung des Schwingkreiswechselrichters ist es, den Auftritt der geringen Schaltverluste bei hohen Betriebsfrequenz zu ermöglichen.

Abb. 4.1, 4.2, 4.3, 4.4 und 4.5 einerseits und Abb. 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22 und 4.23 andererseits zeigen die Modellierungen der Schaltungen für die Schwingkreiswechselrichter mit Papyrus-SysML bezüglich der Erstellung von BDD. Die Erstellung von BDD zur Modellierung mit Papyrus-SysML benötig wie mit UML das Erstellen eines Projektes mit einem Namen und einem Modell, wie auf der Abb. 4.1 zu sehen ist. Anschließend ist eine Modellierungssprache zwischen UML-Core und DSML zu wählen, wobei dieser Abschnitt auf SysML als Teil von DSML (Domain Specific Modeling Language) fokussiert. Abb. 4.2 gibt einen Überblick über die beiden Modellierungssprachen von dem Framework Eclipse-Papyrus: UML und SysML. Nachdem SysML 1.4 gewählt wurde, erscheint ein Fenster auf der Abb. 4.3 mit verschiedenen SysML1.4-Komponenten u. a. *„Block Definition Diagram“*, *„Internal Block“*, *„Parametric“* oder *„Requirement“*. Abb. 4.4 zeigt die Auswahl von *„Block Definition Diagram“* zur Modellierung der Schaltung. Das Blockdefinitionsdiagramm zur Modellierung der Funktionalität des Reihenschwingkreises zeigt Abb. 4.5. Hierbei sind Beziehungen zwischen Blöcken zu sehen. Zum einen ist der Block *„Schwingkreis“* in Verbindung mit dem Block *„Wechselrichter“*, wobei der Lastkreis ein Teil vom Wechselrichter ist. Zum anderen gibt es zwei Beziehungen zwischen den Blöcken *„Kapazitives Verhalten“* und *„Schwingkreis“* oder *„Induktives Verhalten“* und *„Schwingkreis“*. Hierbei ist anzunehmen, dass bei kapazitivem Lastverhalten nur Einschaltverluste auftreten und bei induktivem Lastverhalten nur Ausschaltverluste.

Abb 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22 und 4.23 fokussieren auf die Erstellung von Blockdefinitionsdiagrammen zur Modellierung der Schaltung von einem Schwingkreiswechselrichter mithilfe von

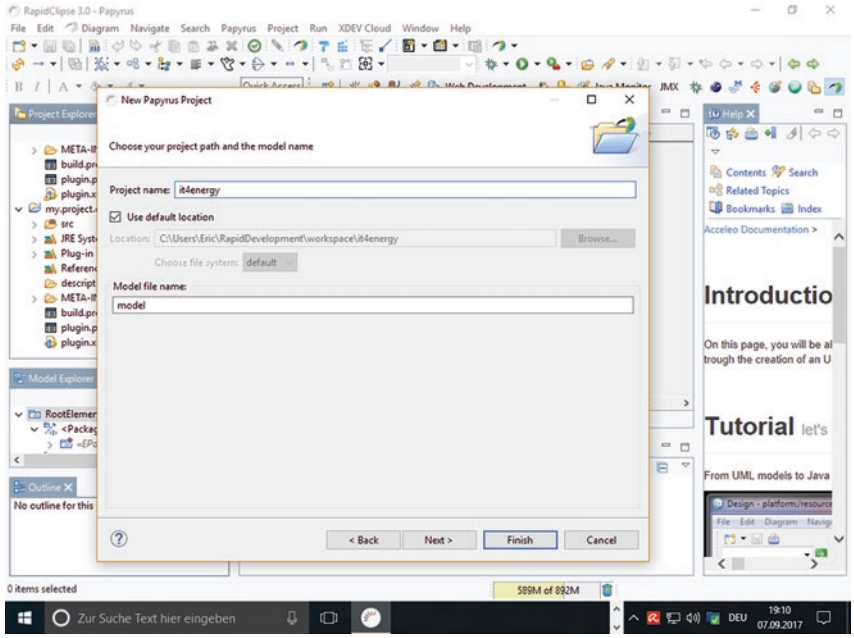


Abb. 4.1 Erstellung eines Projektes genannt *„it4energy“*

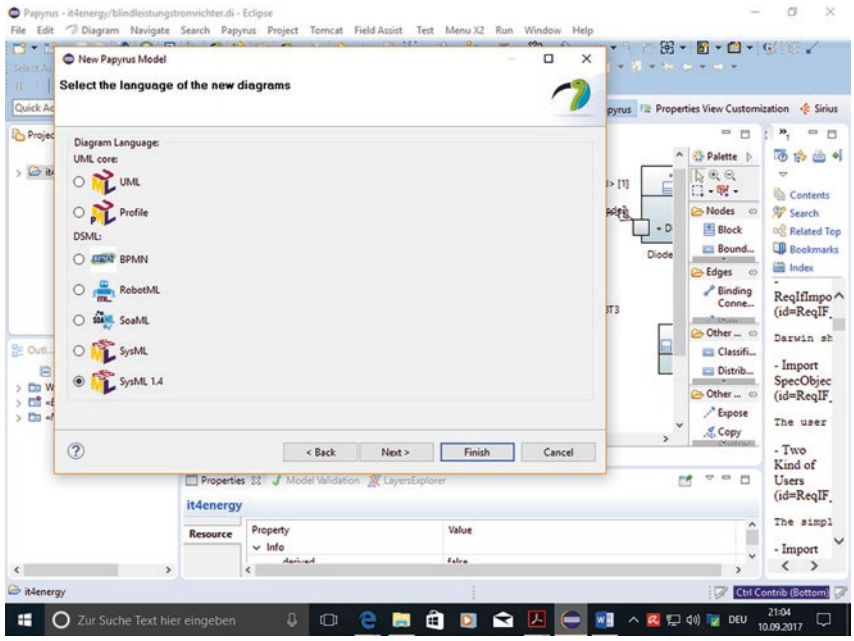


Abb. 4.2 Überblick über UML und SysML vom Framework Eclipse-Papyrus

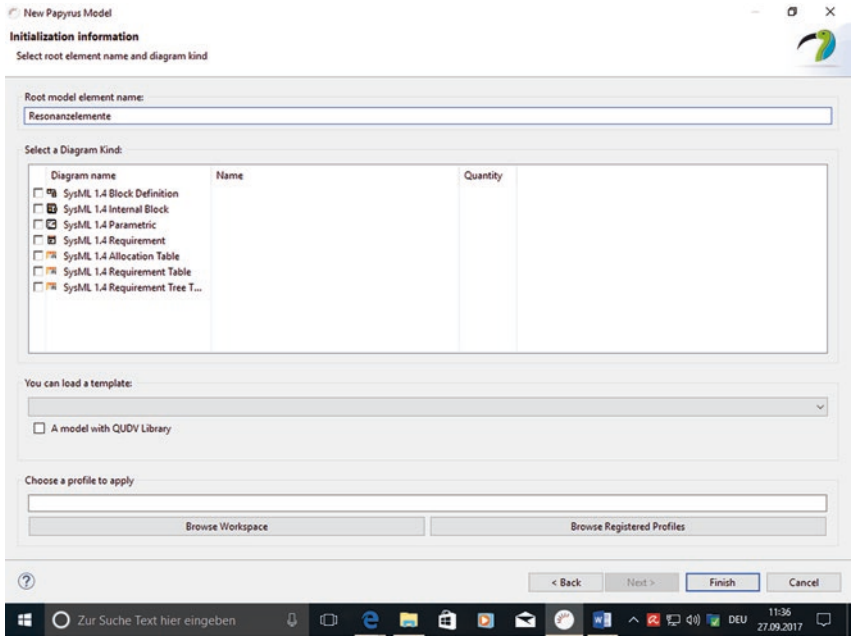


Abb. 4.3 Das Erscheinen eines Fensters mit verschiedenen SysML-Komponenten

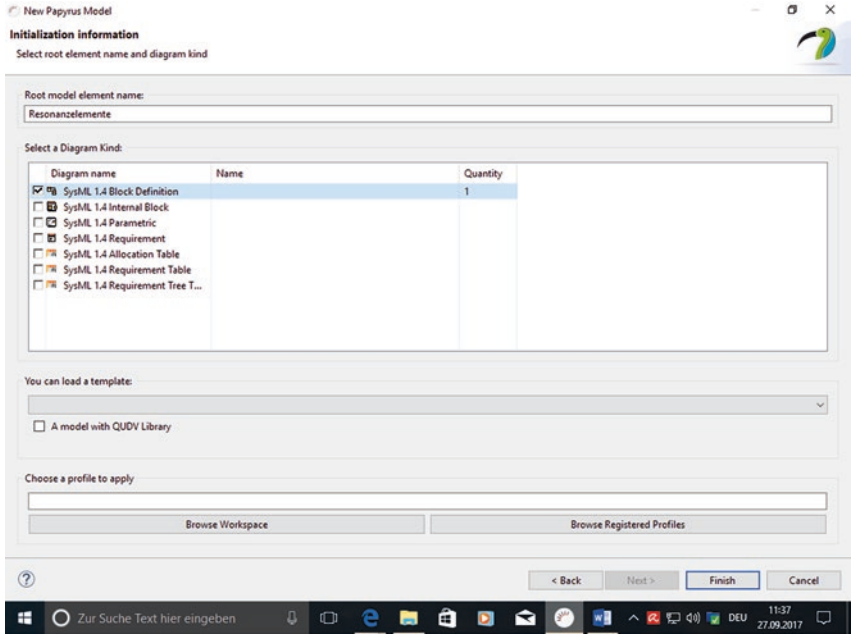


Abb. 4.4 Auswahl von Block Definition Diagram (BDD) zur Modellierung der Schaltung

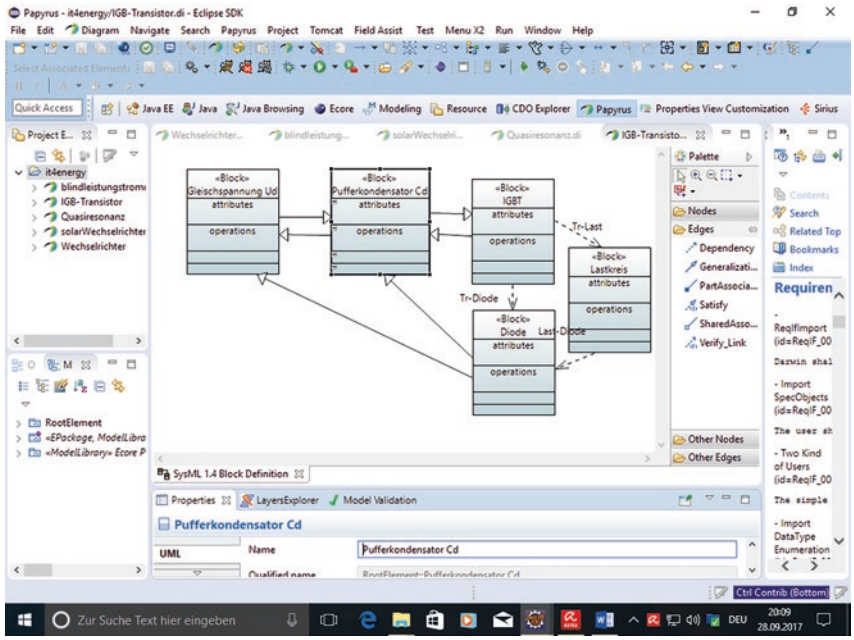


Abb. 4.5 Blockdefinitionsdiagramm (BDD) zur Modellierung der Funktionalität des Reihenschwingkreises

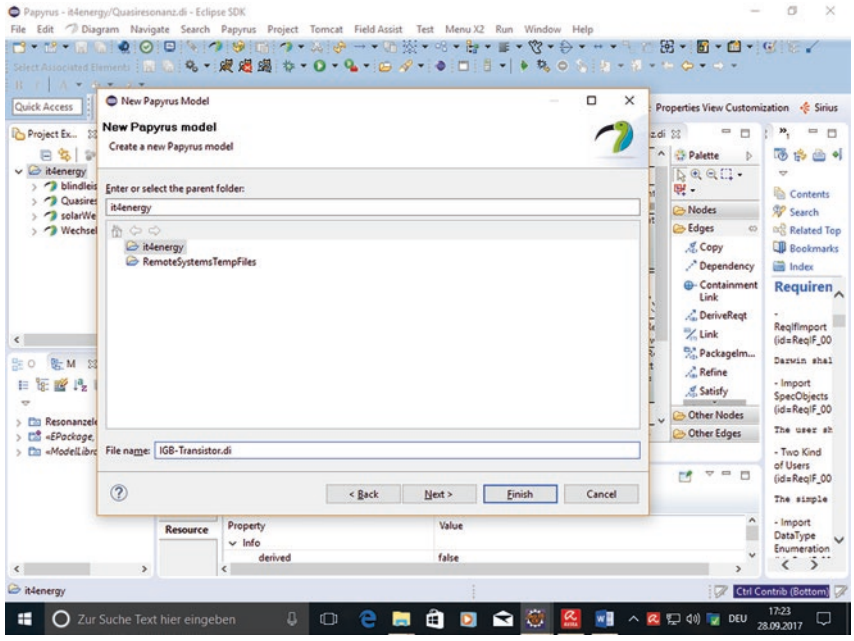


Abb. 4.6 Erstellung des Papyrus-Modells im Projekt *„it4energy“*

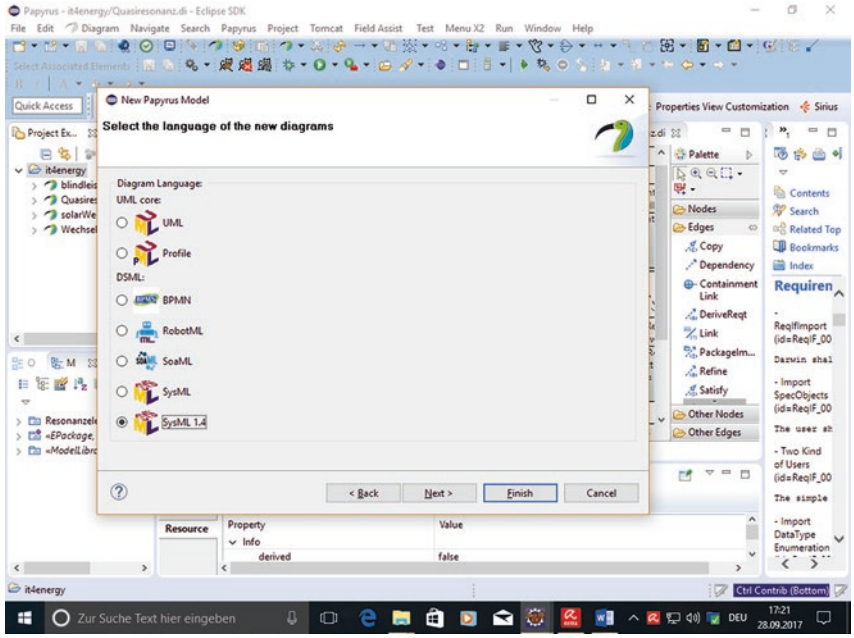


Abb. 4.7 Auswahl der Modellierungssprache SysMl version 1.4 beim Erstellen des Modells genannt *„IGB-Transistor“*

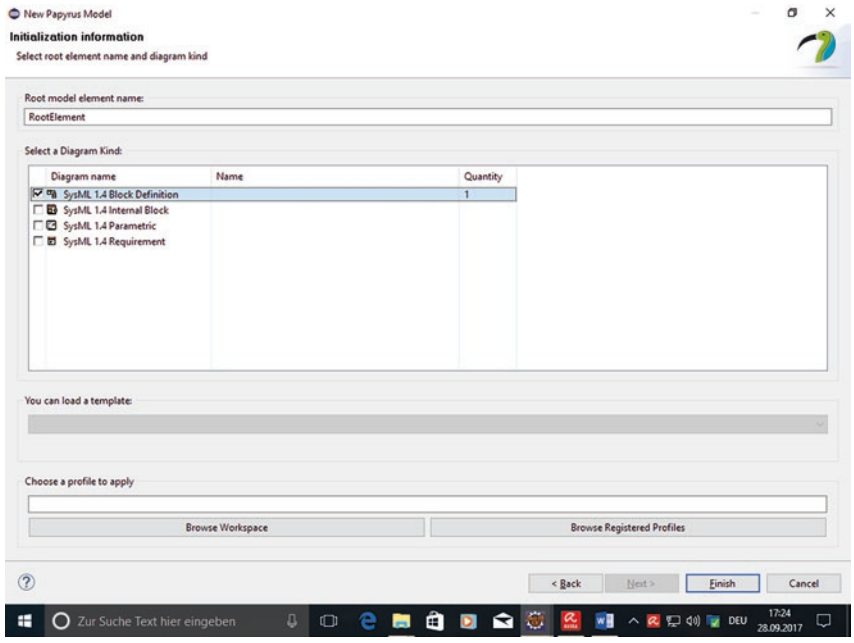


Abb. 4.8 Auswahl des Blockdefinitionsdiagramms (BDD) für SysML 1.4

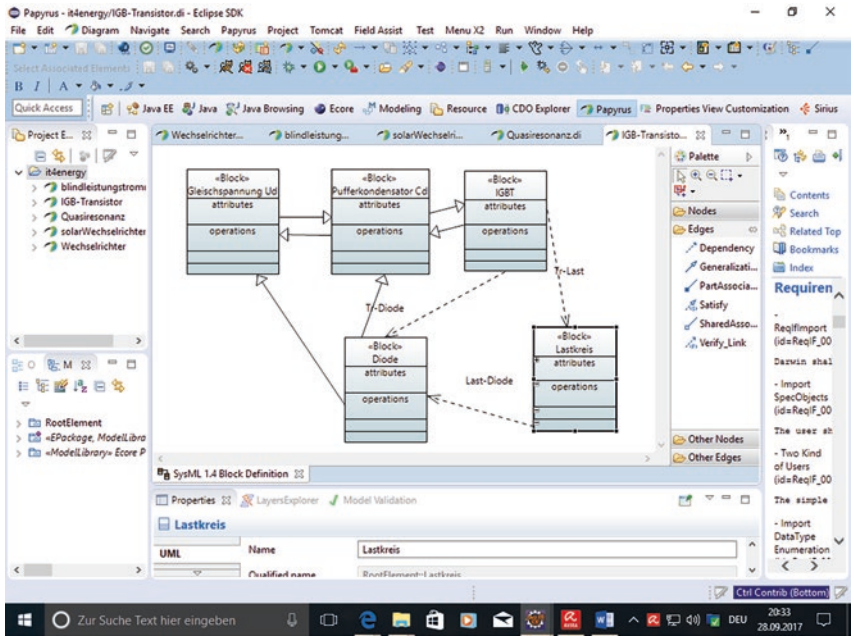


Abb. 4.9 Überblick über das Blockdefinitionsdiagramm (BDD) *„IGB-Transistor“* im Hinblick auf Beziehungen zwischen IGBT, Diode und Lastkreis

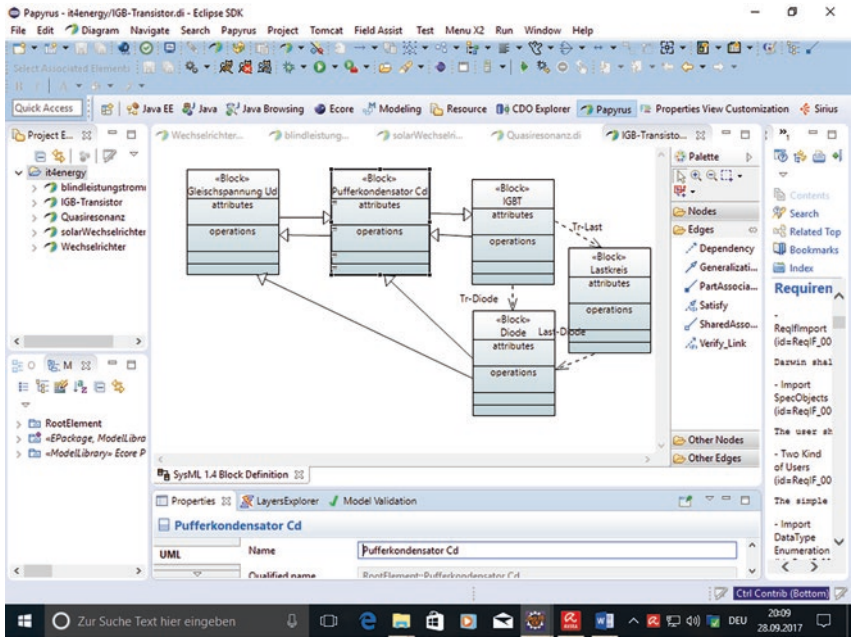


Abb. 4.10 Überblick über das Blockdefinitionsdiagramm (BDD) *„IGB-Transistor“* im Hinblick auf die Dreieck-Beziehungen zwischen Pufferkondensator, IGBT und Diode

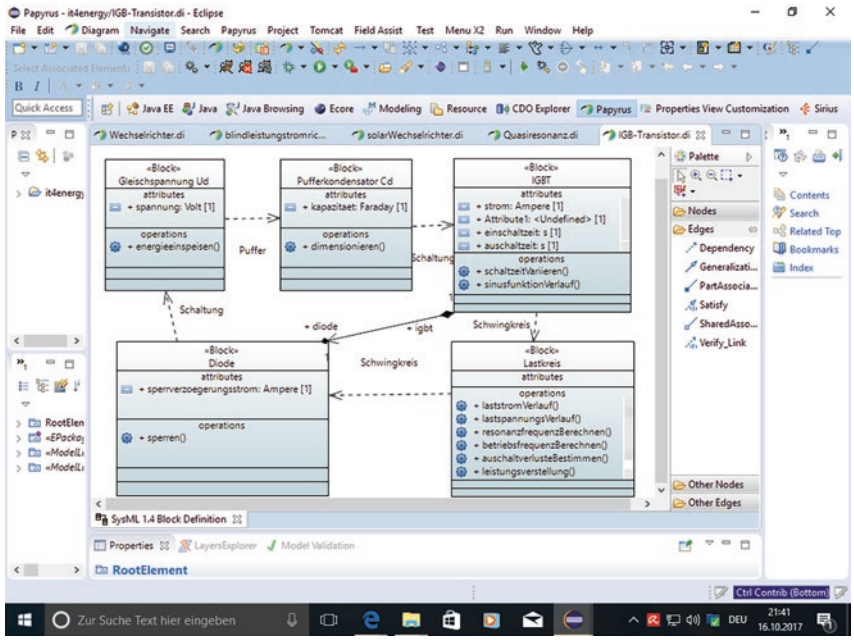


Abb. 4.11 Modellierungskreis von Blöcken im Hinblick auf Abhängigkeitsbeziehungen

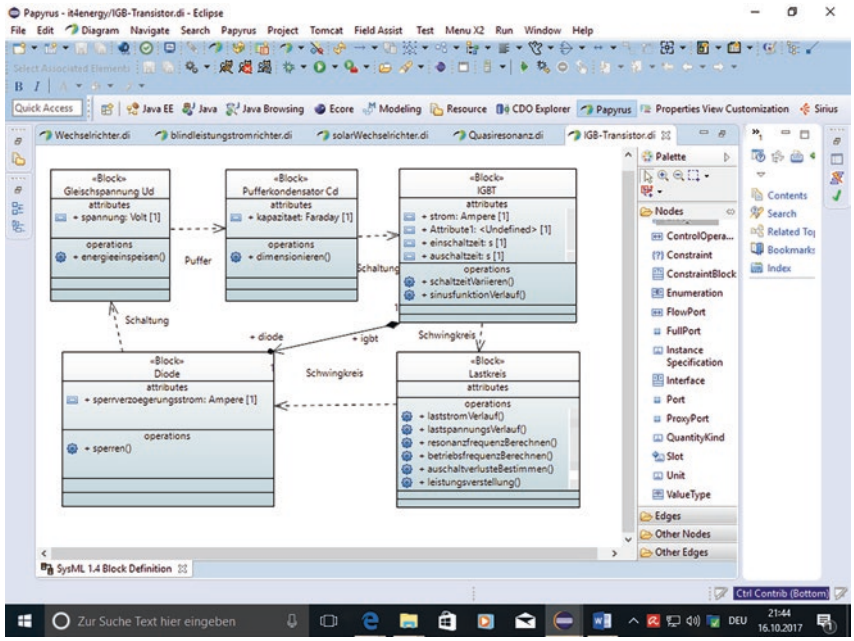


Abb. 4.12 Modellierungskreis von Blöcken im Hinblick auf Eigenschaften *„attributes“*, *„operations“* und *„Nodes“*

Modellierungswerkzeugen, genannt Palette. Hierbei sind die Schritte zur Modellierung mit Papyrus-SysML detailliert. Die Erstellung des BDD beginnt mit dem Erstellen des Projektes genannt *„it4energy“*, und dessen Modells mit dem Namen *„IGB-Transistor“*, wie auf der Abb. 4.6 zu sehen ist. Die Auswahl der Modellierungssprache SysML Version 1.4 zeigt Abb. 4.7. Anschließend erfolgt die Auswahl des Blockdefinitionsdiagrammes auf der Abb. 4.8. Abb. 4.9, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22 und 4.23 und zeigen die Schritte zur Erstellung des Blockdefinitionsdiagrammes und Modellierung der Funktionalität des Reihenschwingkreiswechselrichters. Abb. 4.9 und 4.10 fokussieren auf Beziehungen zwischen Blöcken bezüglich der Funktionalität des Wechselrichters.

Erstens zeigen Abb. 4.9 und 4.10 Dreieck-Abhängigkeitsbeziehungen zwischen *„IGBT“*, *„Diode“* und *„Lastkreis“* bezüglich der Einschaltverluste und Ausschaltverluste. Während des kapazitiven Verhaltens kommutiert der Laststrom von Dioden einerseits nach Transistoren. Dies ergibt Einschaltverluste, wobei die auftretenden Sperrverzögerungsströme der Dioden relative hohe Einschaltverluste verursachen. Während des induktiven Verhaltens verursacht die Ausschaltung der Transistoren andererseits Ausschaltverluste.

Zweitens zeigen Abb. 4.9 und 4.10 Beziehungen sowohl zwischen Pufferkondensator und IGBT-Dioden als auch Pufferkondensator und Gleichspannung. Es ist anzunehmen, dass der in der Schaltung enthaltene Kondensator mit der Kapazität Cd zur Stützung der Versorgungsspannung Ud dient. Mithilfe von dem Block *„Pufferkondensator“* entstehen zwei Typen von Modellierung: horizontale Modellierung und Dreieck-Modellierung. Die horizontale Modellierung ermöglicht die Beziehung von Gleichspannung über den Kondensator bis zum Transistor. Die umgekehrte Richtung von Transistor über Stützkondensator bis zur Gleichspannung erfolgt mithilfe des periodisch auftretenden Energierücktransports. Dieser dient als Speicher für den Kondensator bei induktiver Last. Die Dreieck-Modellierung zwischen Pufferkondensator, Diode und Gleichspannung zeigt eine Schaltung mit eingeprägter Gleichspannung. Der Pufferkondensator ist parallel zur Spannungsquelle geschaltet und verhindert, dass die gelieferte Gleichspannung bei den auftretenden Schaltvorgängen möglichst wenig einbricht. Abb. 4.11 und 4.12 geben Überblicke über die Merkmale von verschieden Blöcken bezüglich der Attribute und Operationen. Außerdem sind die Beziehungen mithilfe des Werkzeuges *„Abhängigkeit“* versehen. Zwischen Blöcken entstehen Abhängigkeiten. Abb. 4.11 und 4.12 zeigen Blöcke in einer Kreis-Modellierung von Gleichspannung->Pufferkondensator ->Lastkreis ->Diode->Gleichspannung. Diese Kreis-Modellierung fokussiert auf Modellierungen von Eigenschaften der Blöcke wie z. B. Attribut oder Operation. Hierbei enthält der Block *„Gleichspannung“* zum einen das Attribut *„spannung“* mit dem Wert *„volt“* und zum anderen die Operation *„energieeinspeisen()“*. Die Einheit der Spannung ist Volt und die Gleichspannung Ud versorgt die Wechselrichterschaltung. Anschließend stütz der Kondensator mit der Kapazität Cd die Versorgungsspannung damit sie nicht einbricht, wobei zum Erzielen eines hohes Wirkungsgrades Cd dimensioniert werden kann. Der Block *„IGBT“* ist ein wichtiger

Bestandteil dieser Schaltung des Schwingkreiswechselrichters, weil die Kommutierung des Stroms von einem Ventil des IGBT auf das andere durch Einwirkung der Spannung erfolgt. Der Block *„IGBT“* enthält Attribute wie z. B. *„strom“*, *„einschaltzeit“* und *„ausschaltzeit“* mit den entsprechenden Werten *„Ampere“*, *„s“* und *„s“*. Die Operationen *„schaltzeitVariieren()“* und *„sinusfunktionVerlauf()“* ermöglichen auf einer Seite die Einstellung der Einschaltzeit- und Ausschaltzeitpunkte der Transistoren und auf der anderen Seite die Arbeit der Transistoren im Gegentakt, d. h., ein paar Transistoren z. B. T1 und T4 bzw. T2 und T3 sind gleichzeitig ein- und ausgeschaltet. Dies ergibt einen zeitlichen Verlauf der Lastspannung und des Laststroms sowohl bei kapazitivem Lastverhalten als auch bei induktivem Lastverhalten. Der Block *„Lastkreis“* besteht aus Operationen zur Darstellung einerseits des zeitlichen Verlaufs der Lastspannung und des Laststroms bei kapazitivem und induktivem Lastverhalten und andererseits der Darstellung der Verluste, der Frequenzen und der Leistungsverstellung. Der Block *„Diode“* beinhaltet das Attribut *„sperrverzoegerungsstrom“* mit dem Wert *„Ampere“* sowie die Operation *„sperren()“*, welche beim Einschalten von dem zweiten Paar der Transistoren z. B. T2 und T3 die Sperrung des ersten Paars der Dioden z. B. D1 und D4 mit negativem Sperrspannung ermöglicht. Der Block „Diode“ ist mit dem Block *„Gleichspannung“* verbunden, weil der Letztere den Ersten versorgt. Der Modellierungskreis schließt sich mit dem Block *„Gleichspannung“*. Dieser stellt den Anfang sowie das Ende der Modellierung mit dem Blockdefinitionsdiagramm, wie auf den Abb. 4.11 und 4.12 zu sehen ist.

Abb. 4.11 und 4.12 zeigen Blockdefinitionsdiagramme über eine Dreieck-Modellierung zwischen den Blöcken *„IGBT“*, *„Lastkreis“* und *„Diode“* zur Darstellung der Beziehungen zwischen Dioden und Lastkreis einerseits und IGBT und Lastkreis andererseits. Es gibt eine Assoziation zwischen den Dioden und Transistoren, wobei IGBT mit Dioden kombiniert sind. Das heißt, die abschaltbaren Transistoren, nämlich IGBT, arbeiten mit den Dioden. Die Beziehungen zwischen den Blöcken in dieser Dreieck-Modellierung sind die Abhängigkeiten sowie Assoziation.

4.1.2.2 Überblicke über Merkmale der Blöcke zur SysML-Modellierung

Der Editor von Eclipse-Papyrus verfügt über die Modellierungswerkzeuge, genannt Palette, u. a. *„Associations“*, *„ModelElements“*, *„DataTypes“* oder *„PortsAndFlows“*. Mithilfe der objektorientierten Modellierung stellen Modelle eine Instanziierung von Objekten anhand von Typdefinitionen dar. Typdefinitionen enthalten Attribute sowie Verhaltensbeschreibungen. Es gibt Wertetyp- und Blockdefinitionen. Typdefinition eines Blocks stellt eine Verbindung mit Verhalten dar. Werte von Modellbeschreibungsgrößen sind veränderbar, weil Blöcke Stereotype der UML-Metaklasse *„Class“* sind und *„Class“* von *„BehavioredClassifier“* erbt.

Abb. 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.21, 4.22 und 4.23 fokussieren auf die Erstellung von Blockdefinitionsdiagrammen mithilfe von Eclipse-Papyrus. Das Erstellen eines Modells, genannt *„Wechselrichter.di“*, erfolgt im Projekt *„it4energy“*.

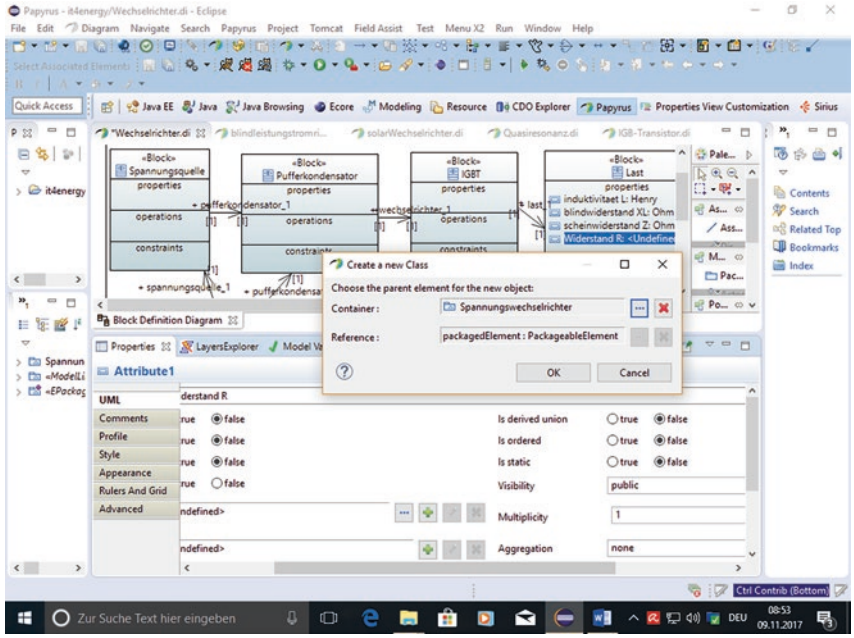


Abb. 4.13 Überblick über das Blockdefinitionsdiagramm (BDD) *„Wechselrichter“* im Hinblick auf das Hinzufügen des Wertes des Attributs *„widerstand“* mithilfe des Containers *„Spannungswechselrichter“*

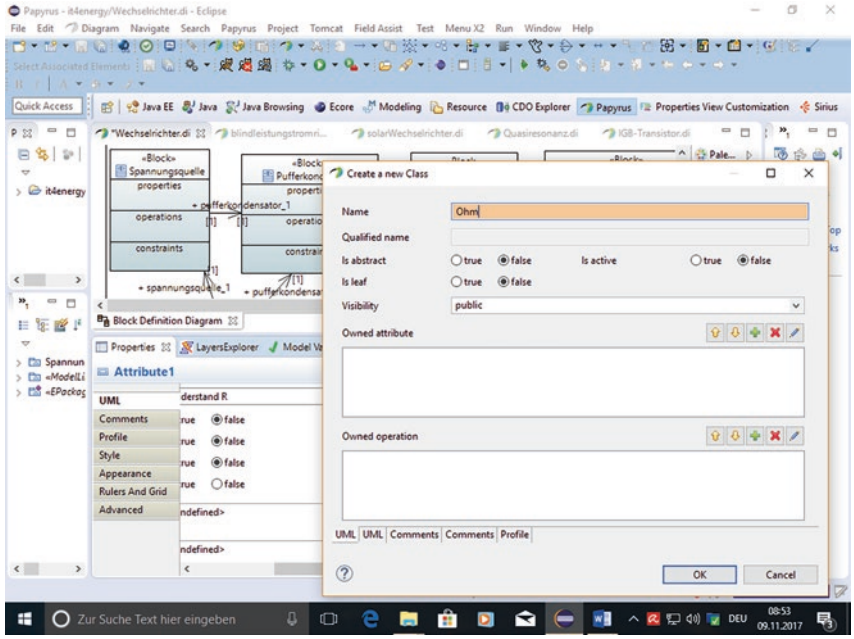


Abb. 4.14 Das Erstellen des Attributs *„Widerstand R“* mit dem Wert *„Ohm“* im Block *„Last“*

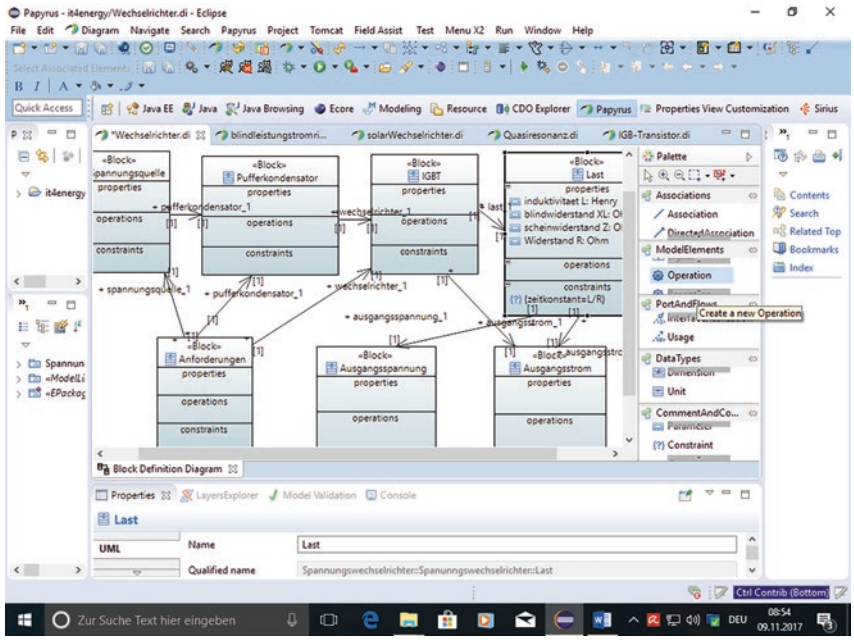


Abb. 4.15 Beziehungen zwischen dem Block *„Last“* und den anderen Blöcken im Hinblick auf die Eigenschaften properties und operations

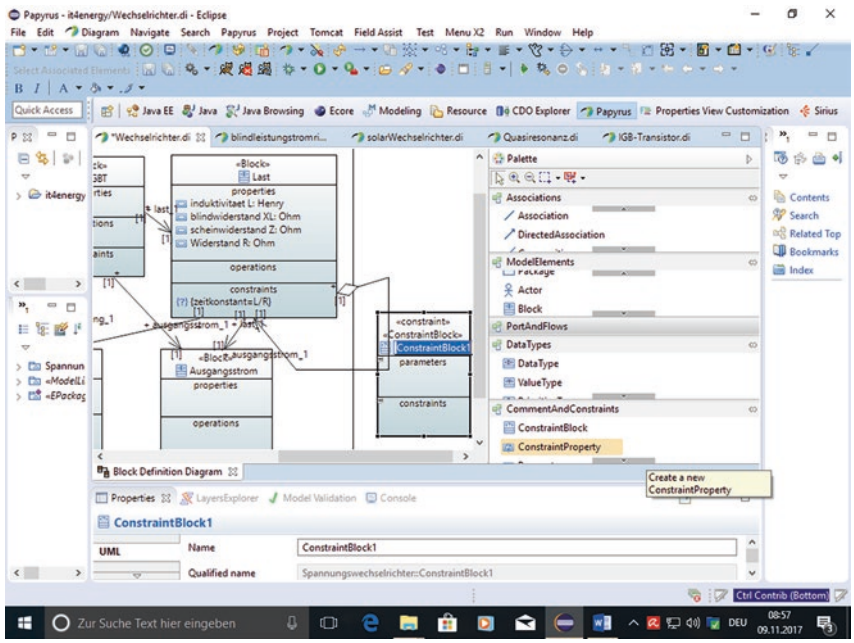


Abb. 4.16 Überblick über den Block *„Last“* im Hinblick auf die Attribute genannt *„properties“* aus den Palette-Komponenten

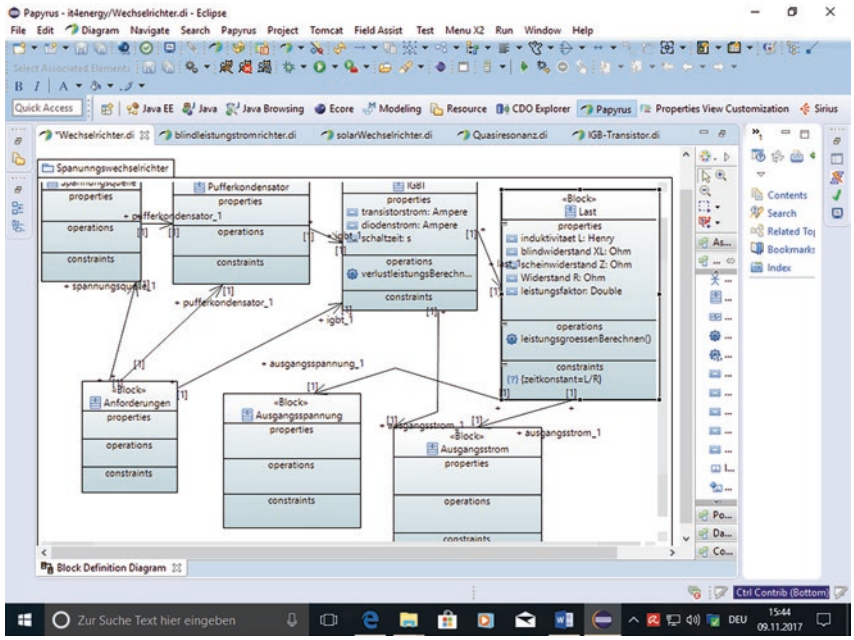


Abb. 4.17 Strukturen der Blöcke *„IGBT“* und *„Last“* im Hinblick auf *„properties“*, *„operations“* und *„constraints“*

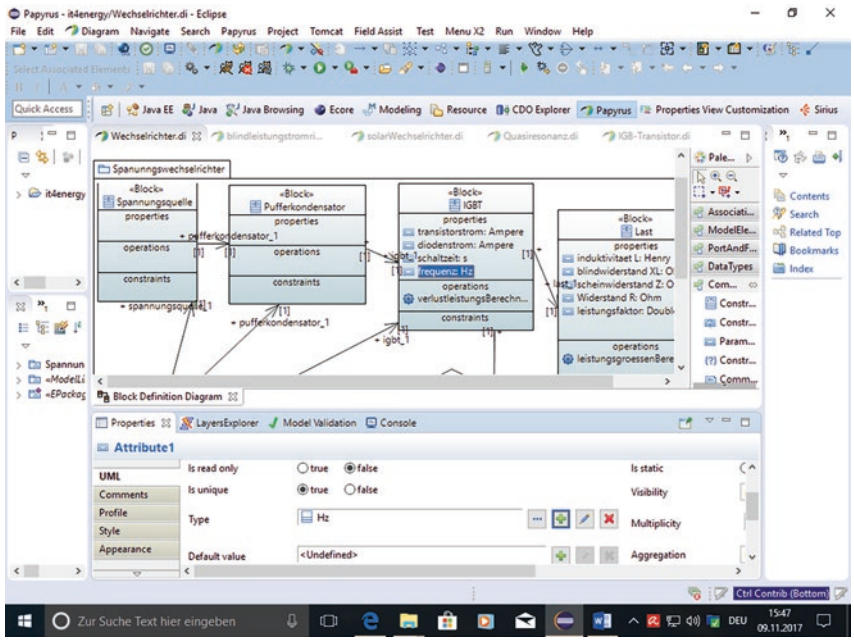


Abb. 4.18 Überblick über das Attribut *„frequenz“* mit seinem Wert *„Hz“*

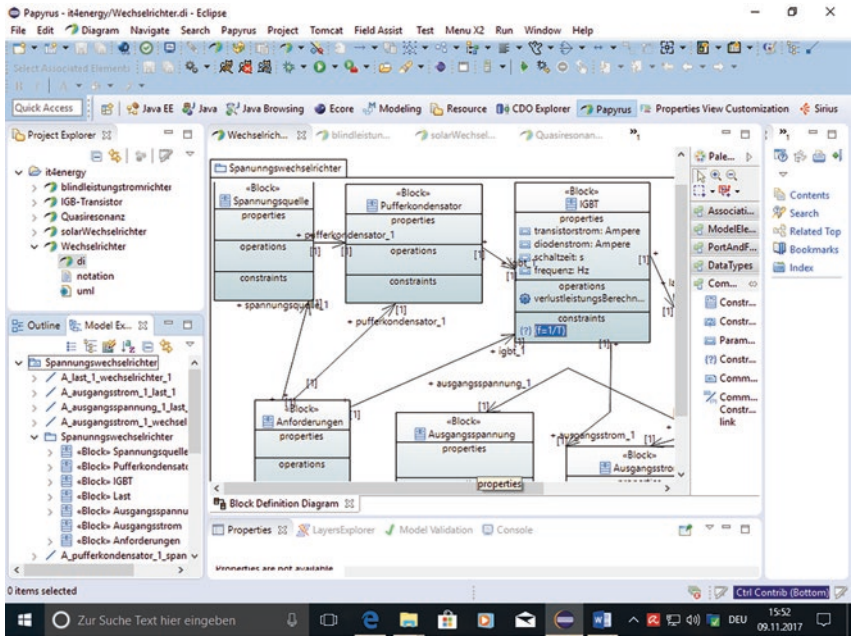


Abb. 4.19 Überblick über den Block *„IGBT“* im Hinblick auf *„properties“*, *„operations“* und *„constraints“*

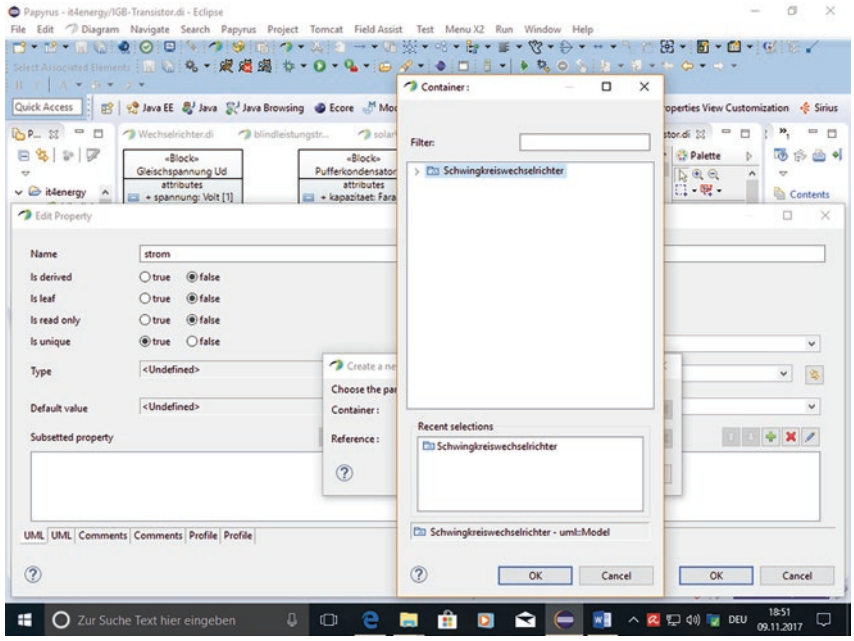


Abb. 4.20 Das Erstellen des Attributs *„strom“* des Blocks *„IGBT“* mithilfe von *„Edit Properties“* im Hinblick auf den Container *„Schwingkreiswechselrichter“*

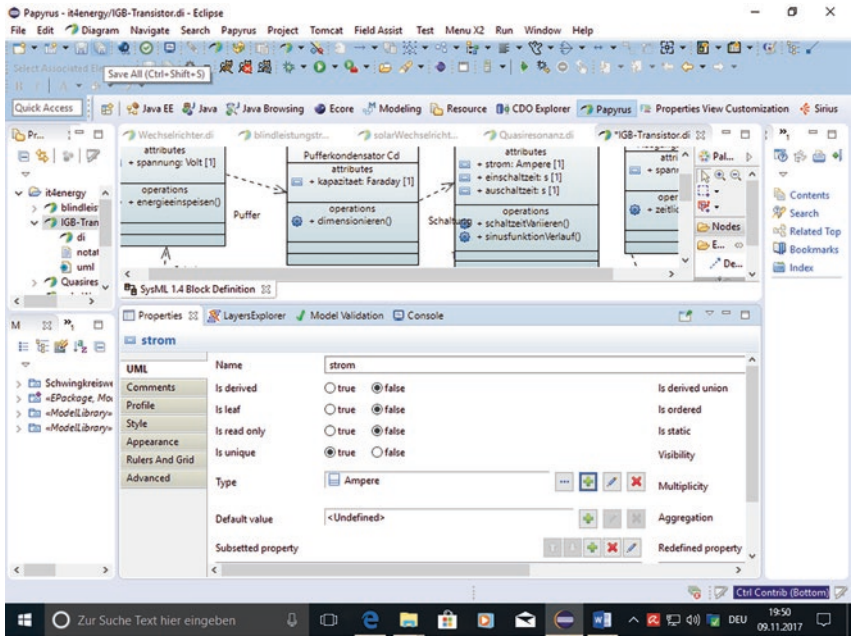


Abb. 4.21 Überblick über das Attribut *„strom“* und dessen Typ *„Ampere“* im Hinblick auf *„Edit Properties“* für den Bock *„IGBT“*

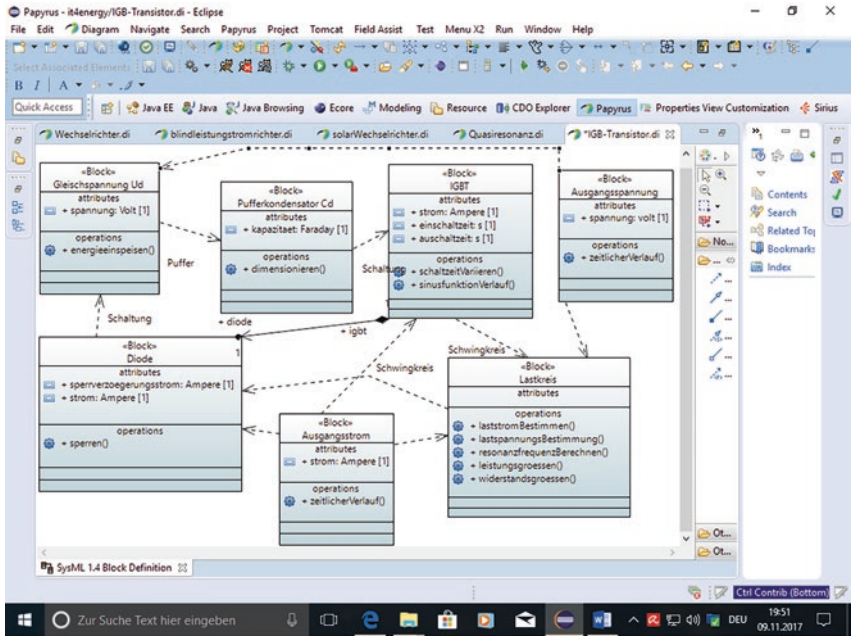


Abb. 4.22 Funktionsweise der Schaltung des Schwingkreiswechselrichter mit Hinblick auf die Beziehungen zwischen den Blöcken dargestellten von Gleichspannung, Pufferkondensator, IGBT, Dioden, Ausgangs- und Ausgangsstrom

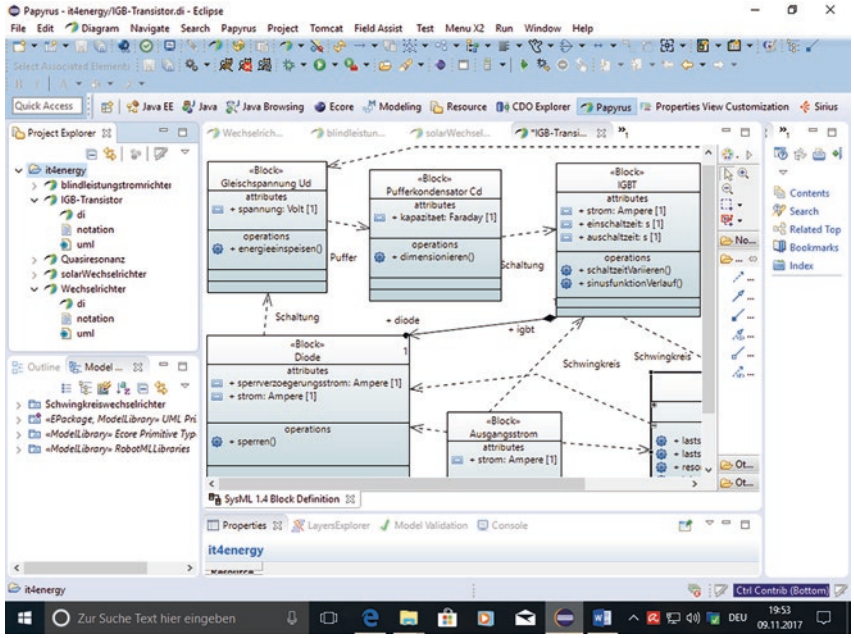


Abb. 4.23 Überblick über die Funktionsweise des Wechselrichters im Hinblick auf die Beziehung zwischen den Blöcken *„IGBT“* und *„Diode“*

Das Modell-Paket, genannt *„Spannungswechselrichter“,* beinhaltet 7 Blöcke, die miteinander verbunden sind. Diese Blöcke stellen die Modellierung der Schaltung des Schwingkreiswechselrichters bezüglich der Verbindung *„DirectedAssociation“* dar. Hierbei sind zum einen auf Abb. 4.13, 4.14, 4.15, 4.16, 4.17, 4.18 und 4.19 direkte Assoziationen zwischen Blöcken zu sehen und zum anderen auf Abb. 4.21, 4.22 und 4.23 Abhängigkeitsbeziehungen. Ziel dieser Modellierung ist es, die Merkmale eines Blocks anhand Erstellens von Attributen und Operationen zu erläutern. Von dem Block *„Spannungsquelle“* über die Blöcke *„Pufferkondensator“*, *„IGBT“* bis zum dem Block *„Last“* folgt die Modellierung einer horizontalen Richtung, weil die Schaltung des Schwingkreiswechselrichters diese Orientierung mit Beziehungen zwischen den Blöcken *„Spannungsquelle“*, *„Pufferkondensator“* und *„Last“* darstellt.

Abb. 4.13 zeigt das BDD *„Wechselrichter“* im Hinblick auf das Hinzufügen des Wertes des Attributs *„widerstand“* mithilfe des Containers *„Spannungswechselrichter“*. Abb. 2.16, 2.17, 2.18 und 2.19 zeigen die Inhalte der Blöcke *„IGBT“* und *„Last“* bezüglich der *„properties“*, *„operations“* und *„constraints“*, wobei diese Eigenschaften diese der UML-Klassendiagramme ähneln. Der Block „Last“ verfügt über die Attribute *„induktivitaet L“*, „blindwiderstand X_L“, *„scheinwiderstand Z“*, *„Widerstand R“* und *„leistungsfaktor“* mit den entsprechenden Werten *„Henry“*, *„Ohm“*, *„Ohm“*, *„Ohm“* und *„Double“*. Die Operation des Blocks „Last“ ist nach *„leistungsgroessenBerechnen“*

bezüglich der Bestimmungen der Leistungen bei Schwingkreiswechselrichtern mit abschaltbaren Leistungshalbleitern wie z. B. IGBT genannt. Hierbei kann die Leistungssteuerung durch drei Verfahren erfolgen: Verstellung der Versorgungsgleichspannung, Verstellung der Betriebsfrequenz und Schwenkverfahren. Der Constraint-Parameter, genannt *„zeitkonstant“* mit dem Wert *„L/R“* ermöglicht die Darstellungen der Properties-Elemente des Blocks *„Last“* bezüglich der Modellierung des zeitlichen Verlaufes des Laststromes sowohl bei kapazitivem als auch bei induktivem Lastverhalten. Der Constraint-Paramerter *„zeitkonstant = L/R“* stellt die Wirkung der Gleichspannung(Spannungsquelle) gegen den Laststrom dar, wobei der Laststrom nach einer Exponentiell-Funktion mit der Zeitkonstanten = L/R kleiner ist. Abb. 4.17, 4.18 und 4.19 zeigt die Struktur des Blocks *„IGBT“* bezüglich der *„properties“*, *„operations“* und *„constraints“* zu den Transistoren, Dioden, Verlustleistungen und Frequenz. Gemäß Abb. 4.17, 4.18 und 4.19 beziehen sich die Properties des Blocks *„IGBT“* auf das Schaltverhalten der Transistoren und Rolle der Dioden. Zwei Paar Transistoren werden im Gegentakt zueinander geschaltet, d. h., T1 und T4 bzw. T2 und T3 werden synchron ein- und ausgeschaltet. Lastströme fließen über die Dioden D1 und D4 bzw. D2 und D3. Das Attribut *„schaltzeit“* mit dem Wert *„s“* stellt die Ein- und Ausschaltzeit der Transistoren. Mithilfe der Attribute *„transistorstrom“* und *„diodenstrom“* mit dem Wert *„Ampere“* fließt der Laststrom entweder über Transistoren oder Dioden. Die Operation *„verlustleistungsBerechnen()“* bezieht sich auf die Bestimmung der Verluste und der Constraint-Parameter *„f = 1/T“* ermöglicht das Berechnen der Frequenz, wie auf der Abb. 4.17 zu sehen ist. Abb. 4.20 und 4.21 zeigen das Erstellen des Attributs *„strom“* des Blocks *„IGBT“* mithilfe von *„Edit Properties“* des Frameworks Eclipse-Papyrus. Auf den Abb. 4.20 und 4.21 sind den Namen des Attributs und den Typ *„strom“* bzw. *„Ampere“* gewählt. Dies entspricht der Eigenschaft des Stromes des Transitors *„IGBT“*. Abb. 4.22 und 4.23 geben Überblicke über die Funktionsweis des Schwingkreiswechselrichters in Bezug auf die elektronischen Blöcke für Arbeit der Transistoren im Gegentakt, Betriebsfrequenz des Wechselrichters mit sowohl kapazitivem als auch induktivem Verhalten des Lastkreises, Leistungssteuerung, zeitliche Verläufe der Lastspannung(Ausgangsspannung) sowie des Laststromes(Ausgangsstromes), Rolle des Pufferkondensators zum Stützen der Gleichspannung sowie der Dioden zum Sperren des Stromes. Abb. 4.22 verdeutlicht die Funktionsweise der Schaltung des Schwingkreiswechselrichter mit Hinblick auf die Beziehungen zwischen den Blöcken. Diese Beziehungen stellen eine Kreis-Modellierung mithilfe der Abhängigkeitsbeziehungen dar. In dem Kreis stellen die Abhängigkeitsbeziehungen zwischen Blöcken Interaktionen dar. Der Kreis ist eine Darstellung der elektronischen Schaltung des Schwingkreiswechselrichters mithilfe der Gleichspannung über Pufferkondensator, IGBT, Dioden bis zu Ausgangs- und Ausgangsstrom. Gemäß Abb. 4.22 verfügt der Block *„Last“* über Operationen zu Bestimmungen von Laststrom, Lastspannung, zugeführter Leistung, Resonanzfrequenz und Widerstand. Mithilfe des Blocks „Last“ stellt der Schwingwechselrichter ein Resonanzwechselrichter dar, weil die Last sowohl das kapazitive als auch das induktive Verhalten ermöglicht. Das Berechnen der Resonanzfrequenz aus dem Block „Last“ ermöglicht das Verstellen der

Leistung und damit der Betriebsfrequenz, wenn beide Frequenzen gleich sind. Aus der Abb. 4.22 ist zu bemerken, dass der Block „Last“ mit den Blöcken „Ausgangsspannung“ und „Ausgangsstrom“ Abhängigkeitsbeziehungen enthält, weil zum einen die Verläufe des Laststromes und der Lastspannung durch den Lastkreis bestimmt werden und zum anderen der Einsatz der abschaltbaren Leistungshalbleitern das Arbeiten der Resonanzwechselrichter zu induktivem sowie kapazitivem Lastverhalten. ermöglicht. Außerdem ist der Block „Last“ in Verbindung mit den Blöcken „IGBT“ und „Diode“, weil die Schaltung des Schwingkreiswechselrichters elektronische Verbindungen zwischen IGBT-Diode und Last darstellt. Gleichspannung und Ausgangspannung sind in Verbindung, weil der Effektivwert der Ausgangspannung der Gleichspannung ist. Die Diode ist in den Transistor IGBT integriert und die Abb. 4.23 zeigt eine Beziehung zwischen den beiden Blöcken „IGBT“ und „Diode“.

4.2 Internes Blockdiagramm (IBD)

Ein Blockdiagramm ist eine grafische Darstellung eines Systems, mithilfe derer eine funktionale Sicht auf dieses System eingenommen werden kann. Blockdiagramme dienen dazu, ein besseres Verständnis für Funktion des und Zusammenhänge innerhalb des Systems zu erzeugen. Blockdiagramme haben ihren Namen von den meist rechteckig gezeichneten Elementen in diesem Diagrammtyp. Blockdiagramme werden zur Beschreibung von Hardware- oder Softwaresystemen sowie zur Darstellung von Abläufen und Prozessen verwendet. In Blockdiagrammen werden Blöcke hinsichtlich ihrer Funktion und Struktur sowie ihrer Beziehung zu anderen Blöcken beschrieben und definiert [5].

Blöcke definieren die modulare Struktureinheit innerhalb der SysML (Systems Modeling Language) und stellen statische Konzepte und Gegenstände des zugrunde liegenden Systems dar. Im Bereich Softwareentwicklung beschreibt ein Block z. B. ein Datenelement, einen Operator oder ein Kontrollflusselement. Ein Block in einem Blockdiagramm beschreibt eine Menge an eindeutig identifizierbaren Eigenschaften, deren Gemeinsamkeit die Definition des jeweiligen Blocks ist. Blöcke beschreiben ein System als eine Ansammlung von Teilen, die im spezifischen Kontext eine bestimmte Rolle spielen.

Interne Blockdiagramme werden zur Darstellung des inneren eines Systems im Bereich System-Engineering.

In *internen Blockdiagrammen (IBD) werden* die Struktur und die Flüsse innerhalb eines Systembausteins (Block) mit den Mitteln der OMG SysML™ *(Systems Modeling Language)* beschrieben. Interne Blockdiagramme bieten eine einfache Übersicht darüber, wie die Teile (Parts) eines Blocks interagieren und welche Art von Daten, Informationen, Signalen oder Materialien in welche Richtungen fließen. Dabei lassen sich auch beliebige Schachtelungstiefen darstellen. IBD verfügen über sowohl Elemente als auch Beziehungen zur Darstellung der Modellierungen. Zum einen verfügen die interne

BD über verschiedene Elemente u. a. *„Block“*, *„Part“*, *„Interface“*, *„Referenz“* oder *„Port“* und zum anderen enthalten IBD mehrere Beziehungen wie z. B. *„Verbindender Konnektor“*, „Konnektor“ oder *„Abhängigkeit“*.

Interne Blockdiagramme von dem Framework Eclipse-Papyrus für SysML-1.4 verfügen über Modellierungswerkzeuge, genannt Palette, die aus drei Bereichen bestehen: *„General Annotation“*, *„Blocks“* und *„Ports and Flow“*. Der erste Bereich beinhaltet verschiedene Elemente wie z. B. *„Realization“*, *„Dependency“*, *„Constraint“*, *„Abstraction“* oder *„Context Link“*. Der zweite Bereich besteht aus Elementen wie z. B. *„ActorPart“*, *„Part“*, *„Binding Connector“*, *„Connector“*, *„BoundReference“*, *„Reference“*, „*Dependency*“. Der dritte Bereich setzt sich aus Elementen wie z. B. *„Port“*, „*FlowPort*“, *„Item Flow“*, *„ProxyPort“* oder *„Full Port“* zusammen.

4.2.1 Modellierung der Schaltung eines Blindleistungsstromrichters mit IBD

Die Schaltung eines Blindleistungsstromrichters dient der Kompensation sowohl von induktiver als auch von kapazitiver Blindleistung. Das Besondere in der Schaltung ist, dass auf der Gleichspannungsseite keine besondere Spannungsquelle, sondern ein Kondensator vorgesehen ist [6]. Ziel ist es, dass bei einem sinusförmigen Verlauf eine Phasenverschiebung von 90°gegenüber der Netzspannung besteht. Die Modellierung der Schaltung stellt den Einsatz von selbstgeführten Stromrichtern dar, wobei der Netzstrom eine sinusförmige Kurvenform annimmt. Hierbei ermöglicht die Modellierung der Schaltung eines Blindstromrichters mit den internen Blockdiagrammen sowohl eine Energieentnahme aus dem Netz als auch eine Energieeinspeisung in das Netz.

4.2.1.1 Modellierung der Funktionalität des Blindleistungsstromrichters mithilfe von SysML-Informationsobjektflüssen

Informationsobjektflüsse stellen die Modellierung eines Kanals zwischen zwei Elementen des Modells dar, über den die beiden Elemente Informationseinheiten,ge nannt *„Item Flow“* austauschen. Informationseinheiten werden eingesetzt, wenn der strukturierte Austausch von Informationen zwischen Elementen eines Systems modelliert werden soll, ohne dass das Modell bereits die Struktur- und Repräsentationsdetails der Information festlegt.

Abb. 4.24, 4.25, 4.26, 4.27, 4.28, 4.29, 4.30, 4.31, 4.32, 4.33, 4.34 und 4.35 zeigen die Modellierung der Schaltung eines Blindleistungsstromrichters mit dem Framework Eclipse-Papyrus bestehend aus einem internen Blockdiagramm mit verschiedenen Blöcken wie z. B. Netz, Spule, IGBT, Dioden, und Kondensator. Abb. 4.26, 4.27, 4.28 und 4.29 geben Überblicke über Interaktionen zwischen Blöcken des internen Blockdiagrammes mithilfe von Informationsobjektflüssen, genannt *„Item Flow“*. Diese Informationsobjektflüsse stellen sowohl eine Energieentnahme aus dem Netz als auch

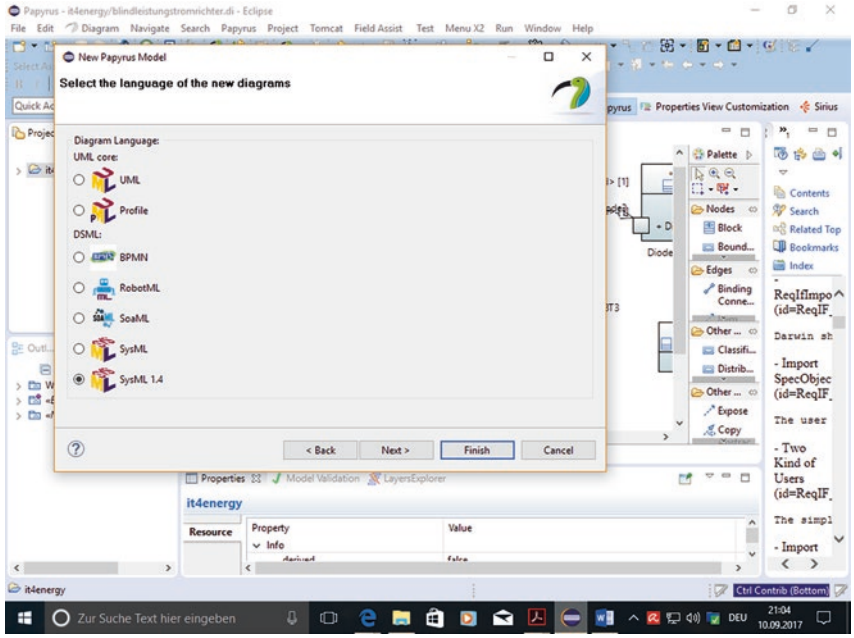


Abb. 4.24 Auswahl der Modellierungssprache SysML zum Erstellen des Modells *„blindleistungsstromrichter“* für das Projekt *„it4energy“* mit Eclipse-Papyrus

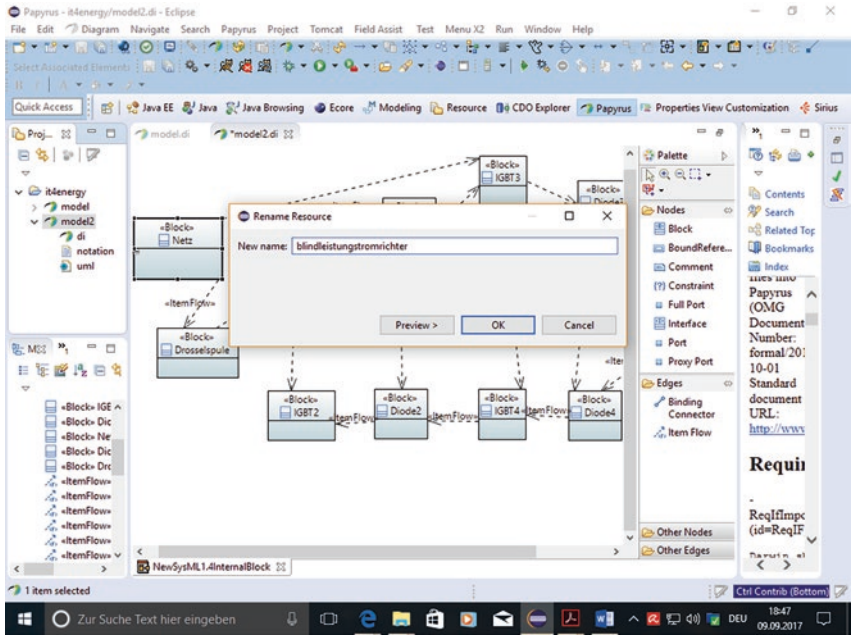


Abb. 4.25 Überblick über das Modell *„blindleistungsstromrichter“*

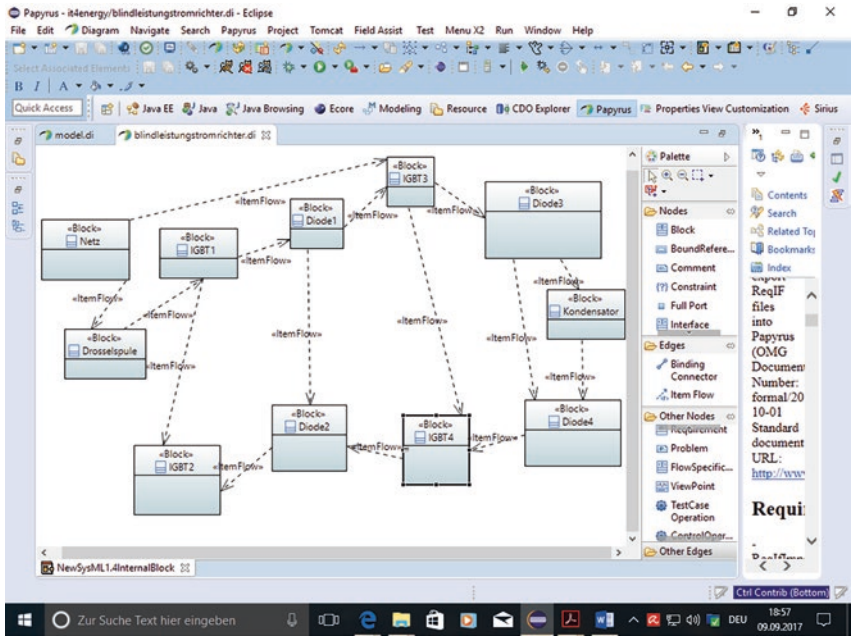


Abb. 4.26 Überblick über das interne Blockdiagramm (IBD), genannt *„blindleistungsstromrichter“*, mit verschiedenen Blöcken wie z. B. Netz, IGBT, Dioden und Kondensator

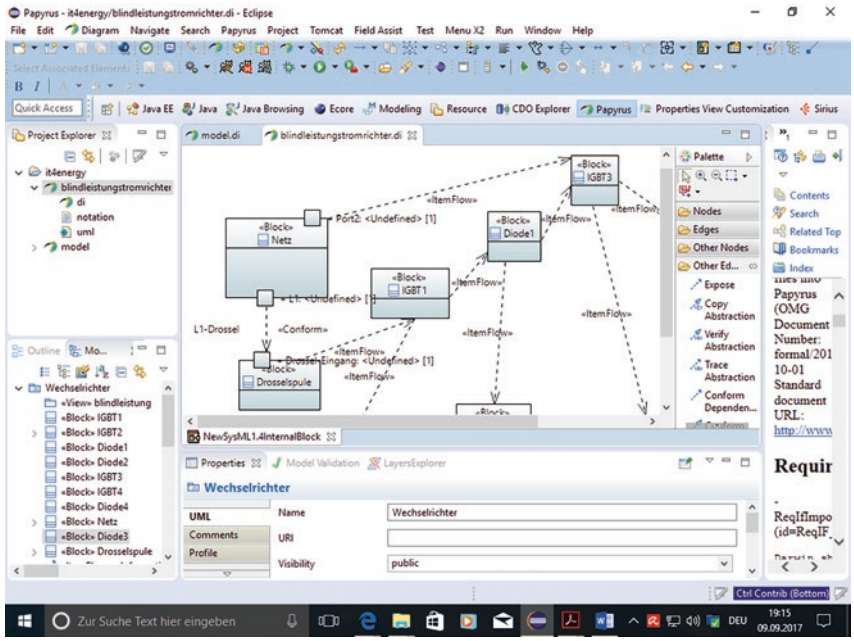


Abb. 4.27 Darstellung der Informationsflüsse mithilfe einer Dreieck-Modellierung

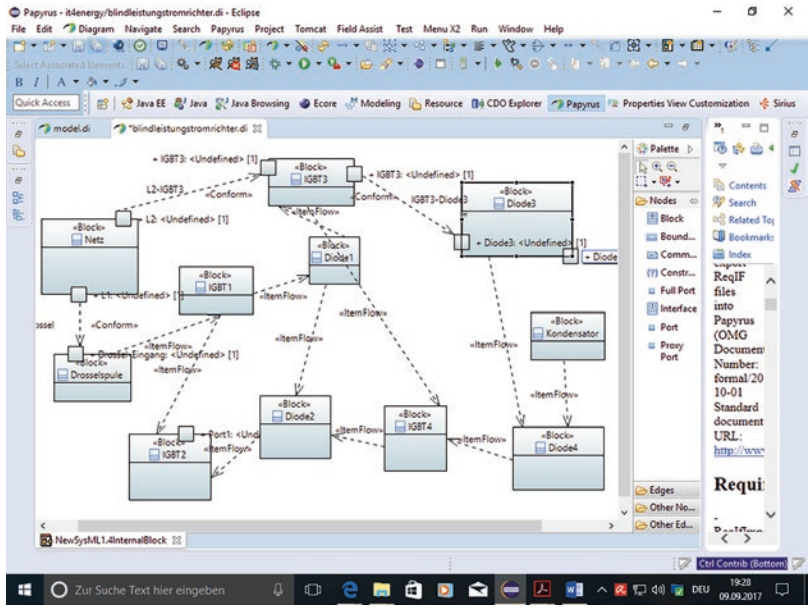


Abb. 4.28 Darstellung der Kompensation der Blindleistung im Hinblick auf die Informationsflüsse von dem Block *„Netz“* über die Blöcke *„Drosselspule“*, *„IGBT1-4“*, *„Diode1-4“* bis zum Block *„Kondensator“*

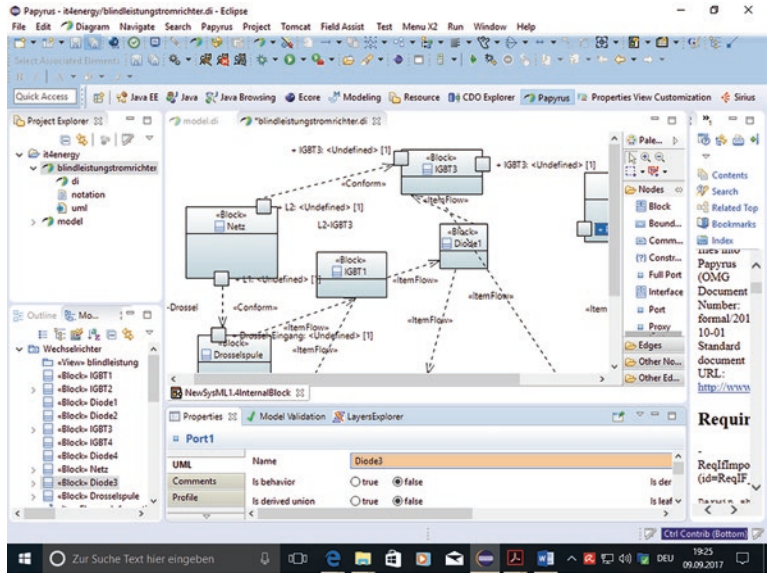


Abb. 4.29 Darstellung der Funktionalität der Blindleistungskompensation mithilfe der Trapez-Modellierung

eine Umkehrung der Energierichtung dar. Im Fall einer Einspeisung der Leistung in das Netz zeigen die Abb. 4.26, 4.27, 4.28 und 4.29 wie die Informationsobjektflüsse verschiedener Blöcke in der Schaltung miteinander zur Darstellung der Funktionalität der Blindleistungskompensation kommunizieren. Hierbei werden Transistoren IGBT1 und IGBT4 einerseits und andererseits die Transistoren IGBT2 und IGBT3 in Gegentakt ein- und ausgeschaltet. Der Strom steigt, wenn der Netzspannungsaugenblickswert positiv ist. Dies führt zu Einschaltung von IGBT1 und IGBT4, wobei der Strom über diese Transistoren fließt. Falls es eine Umschaltung der beiden Zweipaare d. h. IGBT1-IGBT4 und IGBT2-IGBT3 mithilfe von Dioden-Paaren D1-D4 und D2-D3 erfolgt, werden IGBT1 ausgeschaltet und IGBT2 wieder eingeschaltet oder aber IGBT4 ausgeschaltet und IGBT3 wieder eingeschaltet. Von daher wird der Strom durch die Drosselspule getrieben und über einen Freilaufkreis geführt, je nachdem, wie das Transistor-Zweigpaar eine Umschaltung vornimmt. Je nach dem Schaltzustand der Transistoren IGBT1, IGBT2, IGTB3 und IGBT4 lässt sich erreichen, dass der Strom entweder zu- oder abnimmt [6]. Durch das Verstellen von Ein- und Ausschaltzeiten der Transistoren ist das Erzielen eines nahezu sinusförmigen Verlaufs möglich.

Die Informationsflüsse von dem Block *„Netz“* über Blöcke *„Drosselspule“*, *„IGBT1-4“*, *„Diode1-4“* bis zum Block *„Kondensator“* beschreiben den Prozess zur Kompensation von Blindleistung, wie es auf den Abb. 4.26, 4.27, 4.28 und 4.29 zu sehen ist. Der Kondensator ermöglicht einerseits der Speicherung der mit der Blindleistungslieferung verbundenen periodisch auftretenden Energieaustausch. Andererseits stellt der Kondensator die wichtige Gleichspannung der Schaltungsfunktion zur Verfügung.

4.2.1.2 Modellierung der Schaltung des Blindleistungsstromrichters mithilfe der SysML-Objektflussports

Der SysML-Objektflussport auch *„flow port“* genannt ist der Stereotyp des UML-Ports. Objektflussports beschreiben Interaktionspunkte eines Systembausteins mit seiner Umgebung, über welche die Objekte in oder aus dem Baustein fließen können [7]. Ein Objektflussport ist Teil des Systembausteins. Er besitzt einen klein geschrieben Namen, einen Typ sowie eine Multiplizität und wird als kleines Quadrat am Systembaustein dargestellt.

Mittels der Objektflussspezifikation, genannt *flow specification,* die simultan der UML-Schnittstelle ist, wird die Kommunikationsmethode des Objektflussport erläutert, welche die ausgetauschten Daten beschreibt.

Der Informationsobjektfluss, auch *item flow* genannt, stellt ursprünglich aus dem UML-Informationsfluss einen speziellen Informationsfluss dar, der im internen Blockdiagramm an einem Konnektor beschreibt, dass konkrete Objekte transportiert werden. Der Objektflussport beschreibt die fließenden Objekte.

Abb. 4.30, 4.31, 4.32, 4.33, 4.34 und 4.35 fokussieren auf die Modellierungen der Schaltung eines Blindleistungsstromrichters in Bezug auf die Verbindungen zwischen Objektflussports mithilfe von Informationsobjektflüssen. Die Abb. 4.30, 4.31, 4.32, 4.33, 4.34 und 4.35 zeigen auch die Funktionalität der Blindleistungskompensation mit

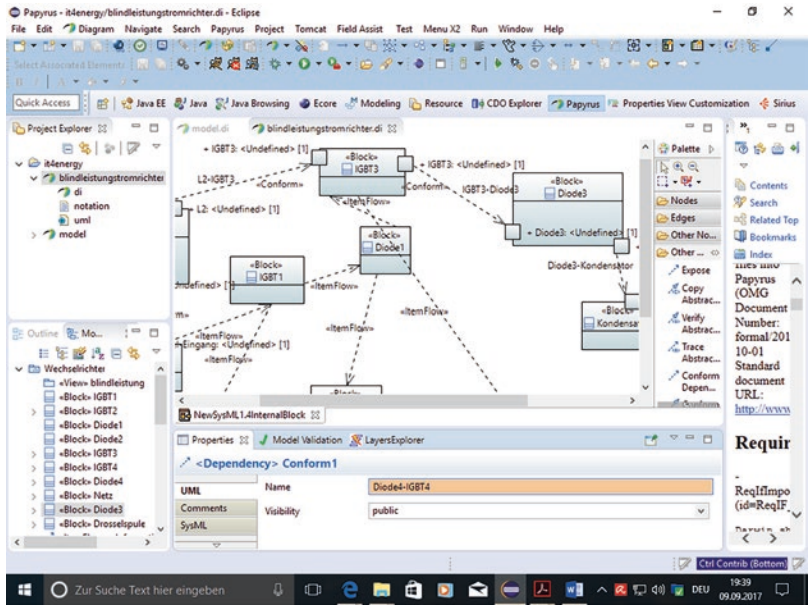


Abb. 4.30 Darstellung der Verbindungen zwischen Objektflussports mithilfe von Informationsobjektflüssen im Hinblick auf die Blöcke *„IGBT3“* und *„Diode3“*

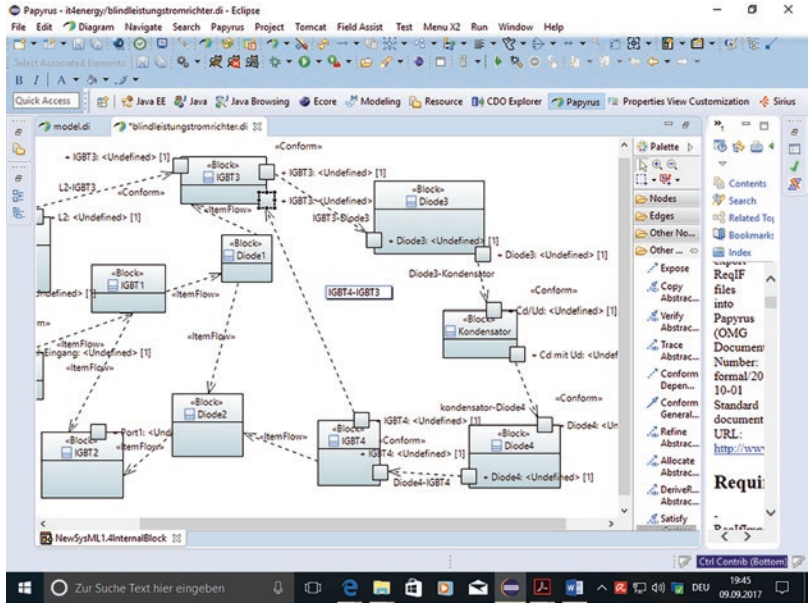


Abb. 4.31 Darstellung eines Informationsobjektflusses zwischen den Blöcken *„IGBT4“* und *„IGBT3“*

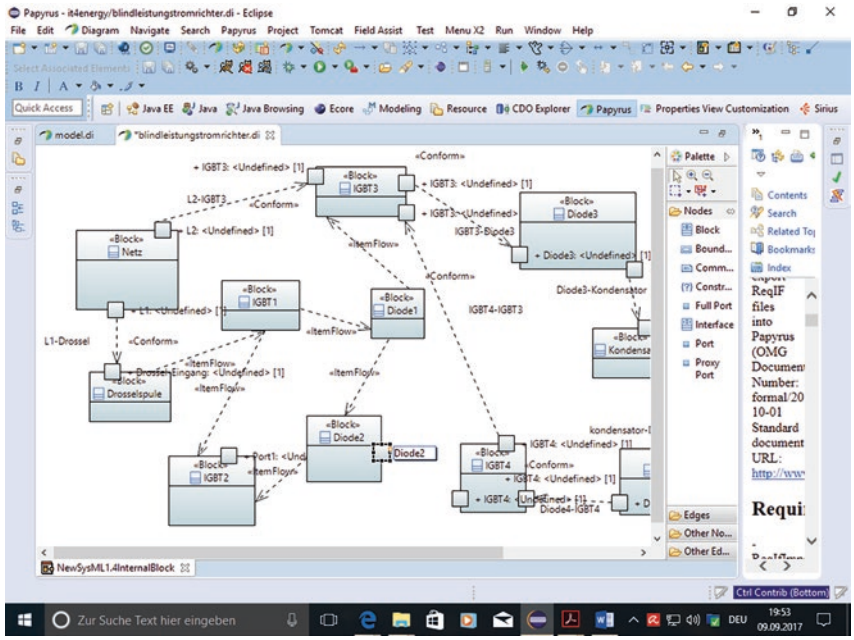


Abb. 4.32 Überblick über die Position eines Ports des Blocks *„Diode2“*

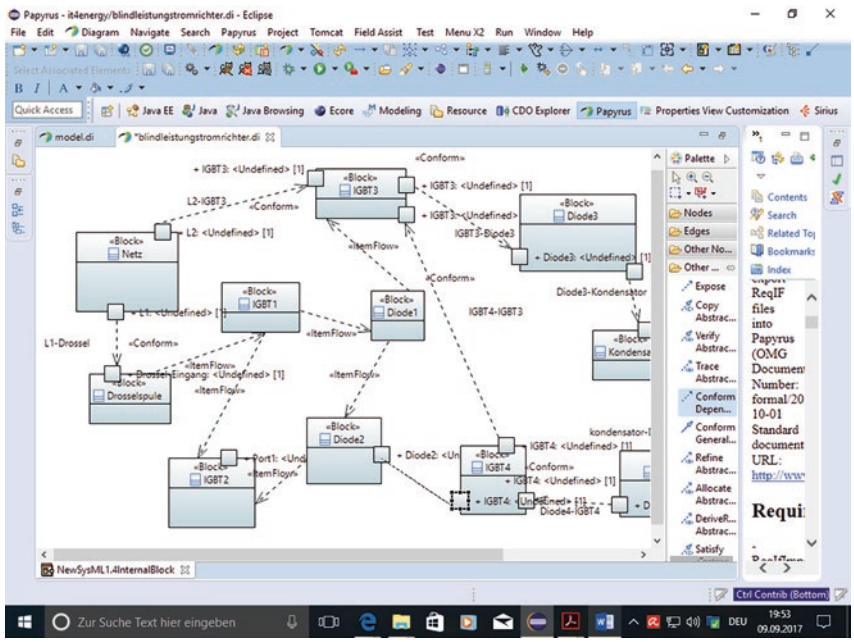


Abb. 4.33 Überblick über verschiedene Informationsobjektflüsse zwischen Blöcken im Hinblick auf die Verbindung zwischen zwei Ports

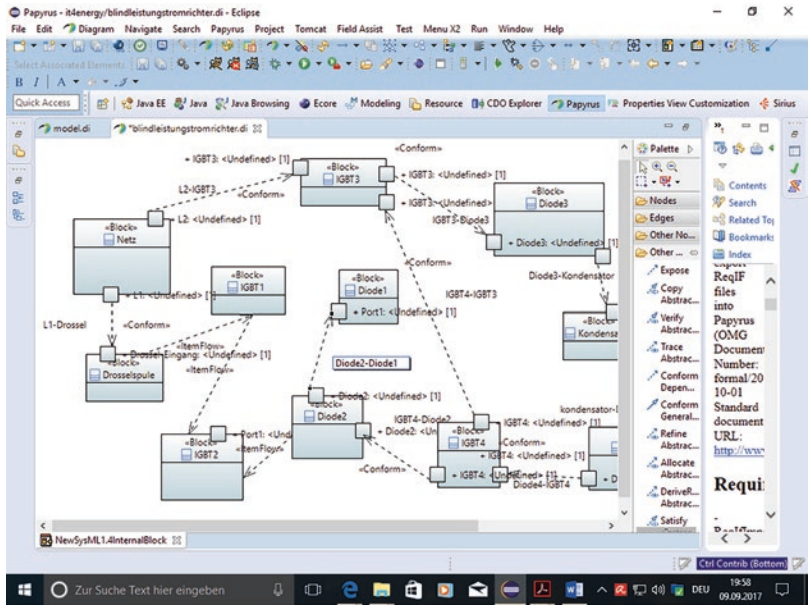


Abb. 4.34 Darstellung der Informationsobjektflüsse zur Darstellung der Energierichtung der Blindleistungskompensation mithilfe von Kommunikationen zwischen Blöcken im Hinblick auf die Blöcke *„Diode2“* und *„Diode1“*, *„IGBT2“* und *„IGBT1“* oder *„IGBT4“* und *„IGBT3“*

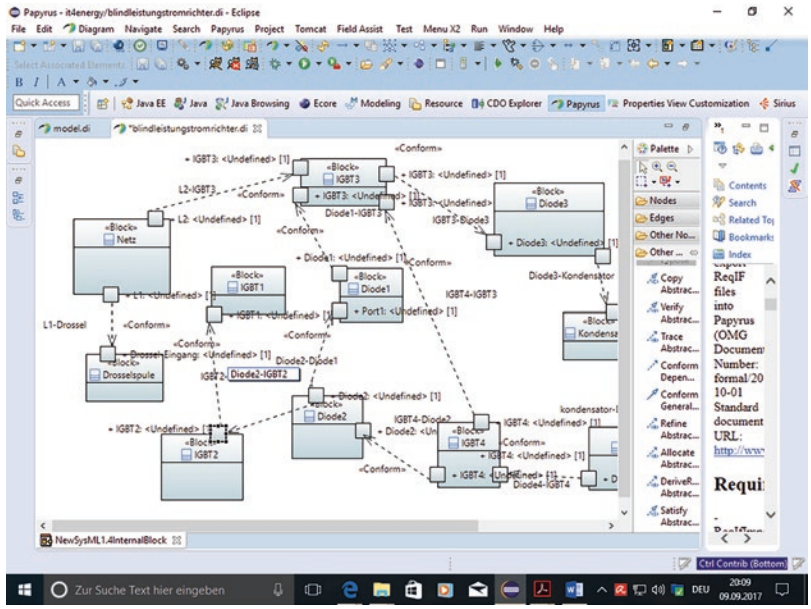


Abb. 4.35 Überblick über den Block *„IGBT3“* mit vier Ports

Hinblick auf Abhängigkeitsbeziehungen, genannt *„Dependency“*, und Informationsobjektflüsse. Gemäß Abb. 4.30, 4.31, 4.32, 4.33, 4.34 und 4.35 sind zwei Blöcke in der Verbindung, wenn sie mithilfe von *„Ports“* und *„Item Flow“* einen Informationsobjektfluss erzeugen können. Zum Beispiel auf der Abb. 4.33 sind verschiedene Informationsobjektflüsse mittels *„Ports“* und *„Item Flow“* zu sehen, u. a. zwischen Blöcken *„Netz“* und *„Drosselspule“* einerseits und andererseits *„Netz“* und *„IGBT3“*. Informationsobjektflüsse auf den Abb. 4.30, 4.31, 4.32, 4.33, 4.34 und 4.35 stellen die Energierichtung der Blindleistungskompensation mithilfe von Kommunikationen zwischen Blöcken des internen Blockdiagramms dar. Beispielsweise verhindert der Block *„Drosselspule“* den Anstieg des Kollektorstromes während des Einschaltens der Transistoren *„IGBT1“* und *„IGBT4“*, wie es auf den Abb. 4.30, 4.31, 4.32, 4.33, 4.34 und 4.35 zu sehen ist, wobei es zu bemerken ist, dass die Modellierung mit Informationsobjektflüssen die Kompensation sowohl von induktiver als auch von kapazitiver Blindleistung darstellt.

4.2.2 Modellierung der Leistungssteuerung durch die Spannungsverstellung bei Schwingkreiswechselrichtern mit IBD

Die Modellierung der Leistungssteuerung bei Schwingkreiswechselrichter mit dem internen Blockdiagramm von dem Framework Eclipse-Papyrus stellt die Kommunikation zwischen verschiedenen *„Parts“* mithilfe von Konnektoren dar. Beispielsweise ist der Part1 mit dem Part2 in der Verbindung mithilfe von Konnektoren, genannt *„connector“*. Der Part stellt eine Struktur in Relation eines Systembausteins dar. Dargestellt wird der Part als Rechteck innerhalb des Systembausteins. Der Unterschied zu einem Systembaustein besteht darin, dass der Name des Parts nicht unterstrichen ist. Die Zahl in der oberen rechten Ecke zeigt die Multiplizität. Das Rechteck wird gestrichelt dargestellt, wenn der Part nicht von dem Systembaustein besessen wird. Als Konnektoren werden die Beziehungen zwischen Parts bezeichnet, über welche die Kommunikation untereinander stattfindet.

Abb. 4.36, 4.37, 4.38, 4.39, 4.40, 4.41, 4.42, 4.43, 4.44 und 4.45 geben Überblicke über die Modellierung der Leistungssteuerung bei Schwingkreiswechselrichtern mit dem internen Blockdiagramm (IBD) vom Eclipse-Papyrus bezüglich der Verbindungen zwischen verschiedenen Parts. Abb. 4.36, 4.37, 4.38, 4.39, 4.40, 4.41, 4.42, 4.43, 4.44 und 4.45 stellt die Modellierung der Schaltung des Schwingkreiswechselrichters bezüglich der Leistungssteuerung durch das Verstellen der Versorgungsgleichspannung (Gleichspannung). Dieses Verfahren hat den Vorteil, dass die Betriebsfrequenz sehr nahe der Resonanzfrequenz gewählt werden kann und somit nur geringe Schaltverluste auftreten. Das Verfahren zum Verstellen der Gleichspannung versursacht zusätzliche Schaltungsaufwand [6].

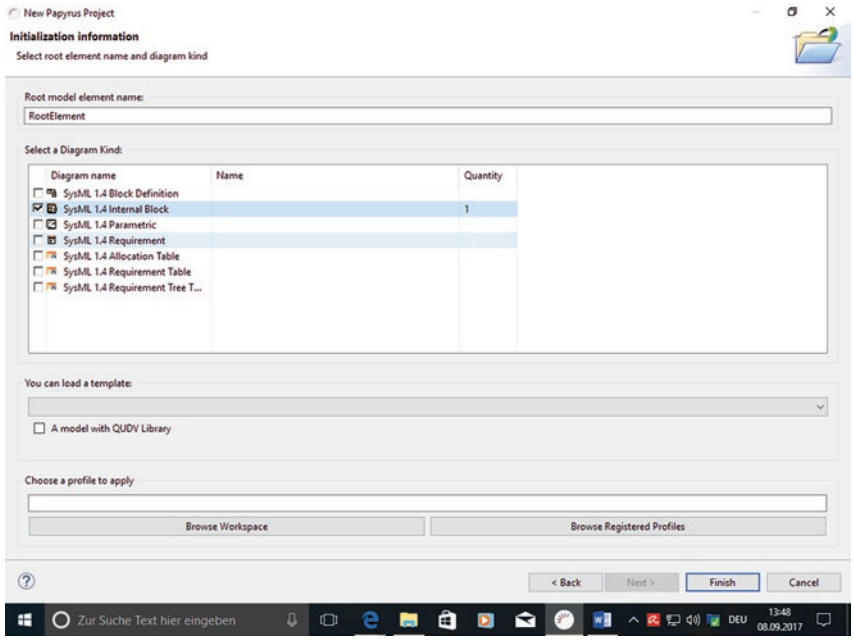


Abb. 4.36 Auswahl des internen Blockdiagramms *„internal Block“* von SysML 1.4 mit Eclipse-Papyrus zur Modellierung der Schaltung des Schwingkreiswechselrichters

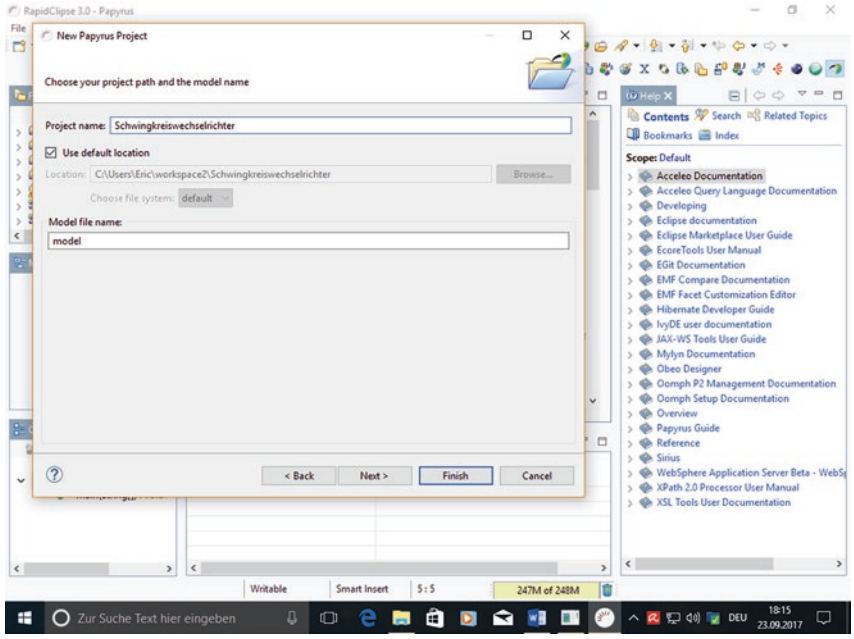


Abb. 4.37 Überblick über den Namen des Projektes und dessen Pfad

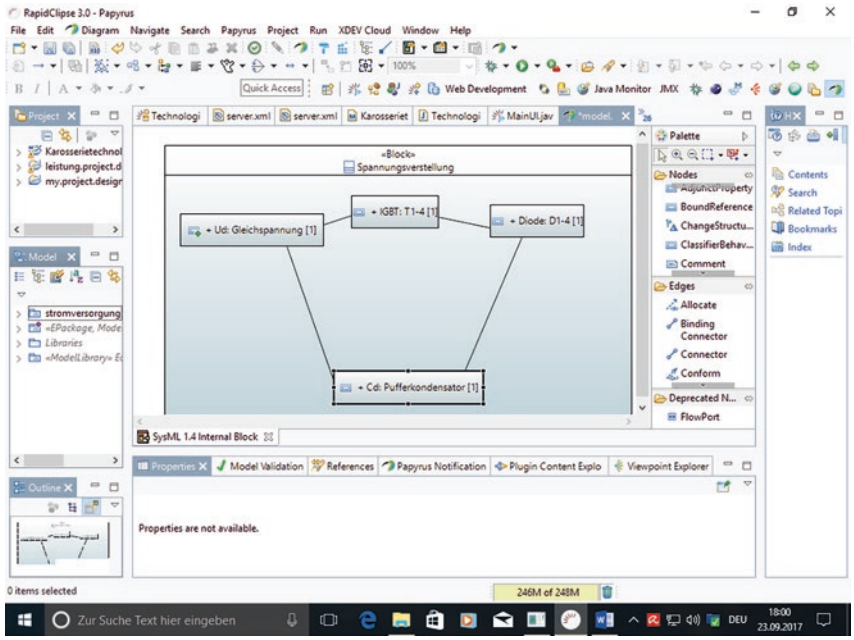


Abb. 4.38 Darstellung der Verbindungen zwischen Gleichspannung, Kondensator, Transistoren IGBT (T1-4) und Dioden (D1-4) zur Modellierung der Funktionalität des Lastgeführten Wechselrichters mit den Parts des internen Blockdiagramms (IBD)

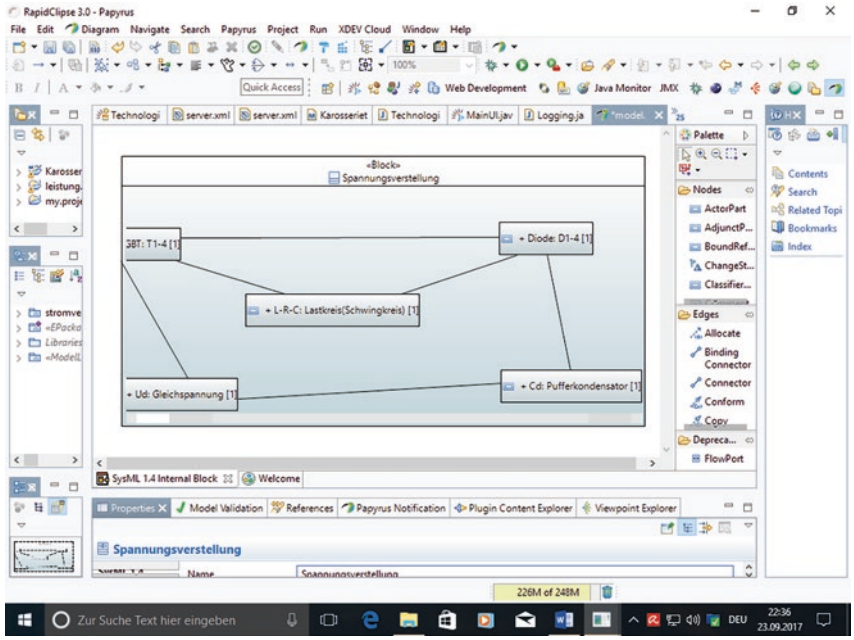


Abb. 4.39 Überblick über die Parts des internen Blockdiagrammes zur Darstellung der Funktionalität des Wechselrichters im Hinblick auf die Elemente des Lastkreises

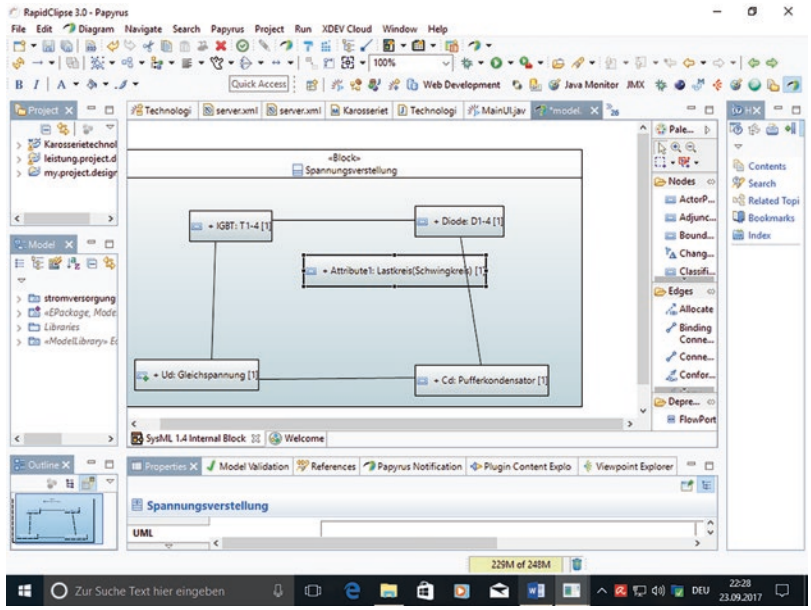


Abb. 4.40 Darstellung der Viereck-Modellierung im Hinblick auf die Verbindungen zwischen Gleichspannung-IGBT, IGBT-Diode, Diode-Pufferkondensator und Pufferkondensator-Diode

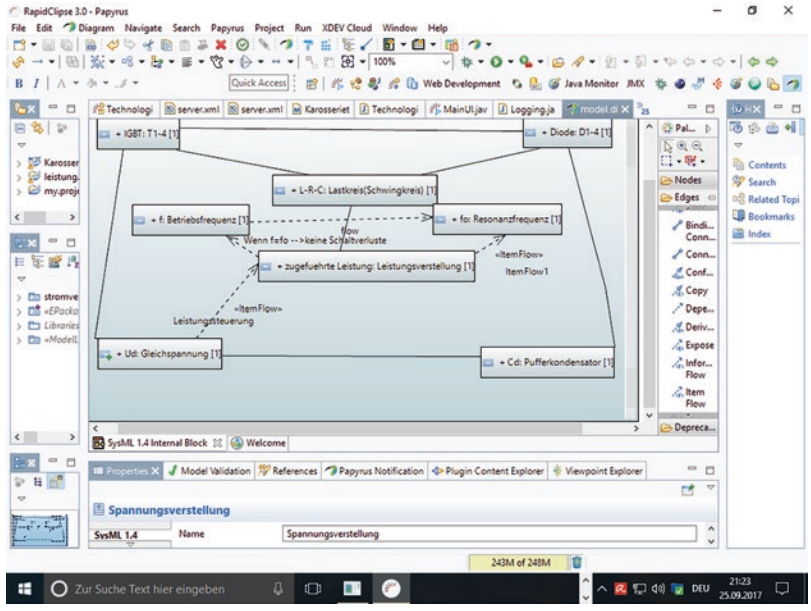


Abb. 4.41 Die erste Dreieck-Modellierung aus den Parts *„IGBT“*, *„Diode“* und *„L-R-C“* und die zweite Dreieck-Modellierung aus den Parts *„Betriebsfrequenz“*, *„Resonanzfrequenz“* und *„zugeführte Leistung“*

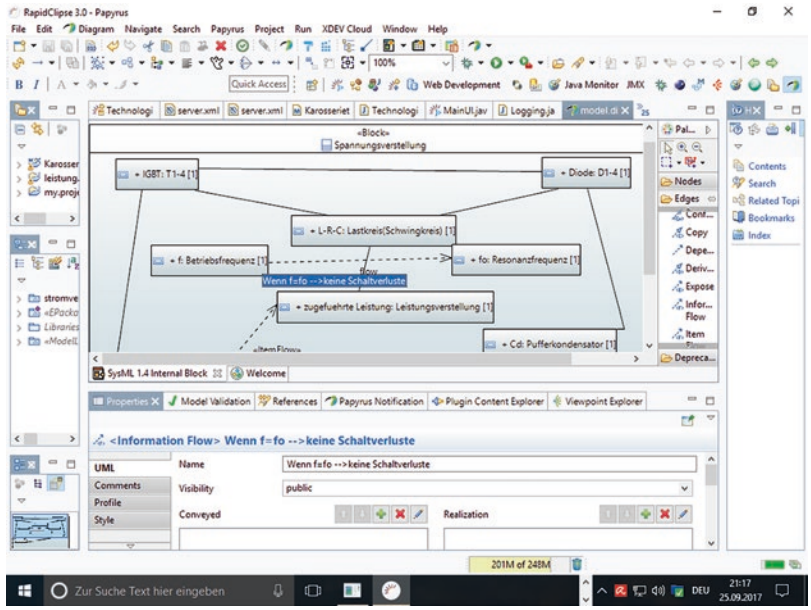


Abb. 4.42 Überblick über die Verbindung zwischen den Parts „Lastkreis“ und „Leistungsverstellung“ mithilfe eines Konnektors

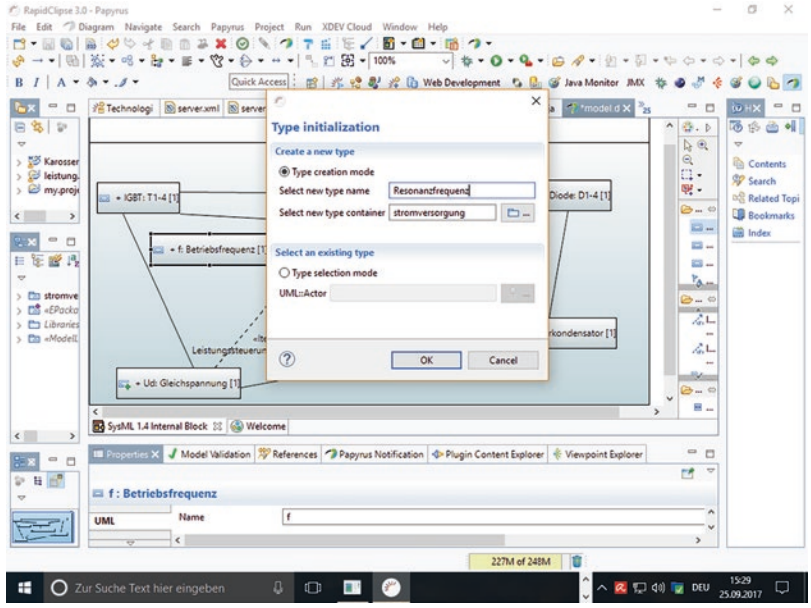


Abb. 4.43 Überblick über das Erstellen des Parts *„Resonanzfrequenz“* im Container *„stromversorgung“*

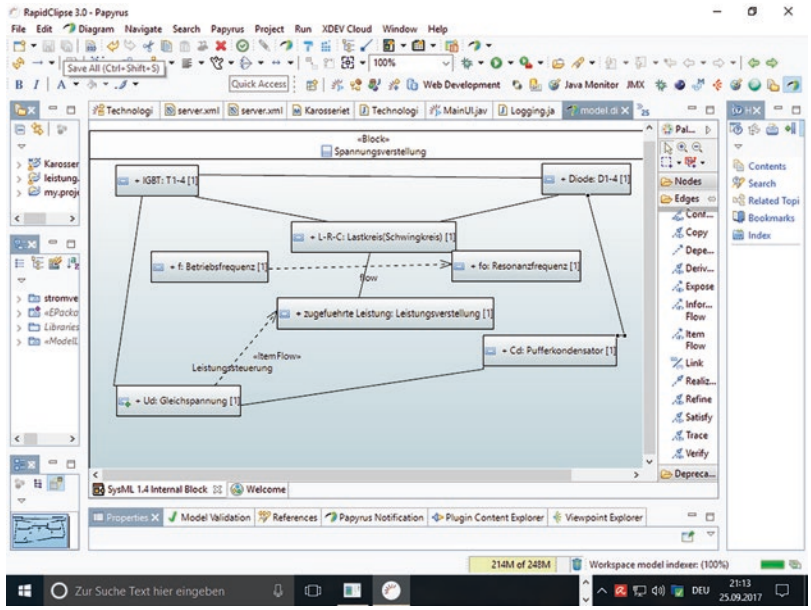


Abb. 4.44 Überblick über Verbindungen zwischen den Parts *„Betriebsfrequenz“*- *„Resonanzfrequenz“* und *„Gleichspannung“* –*„Leistungsverstellung“*

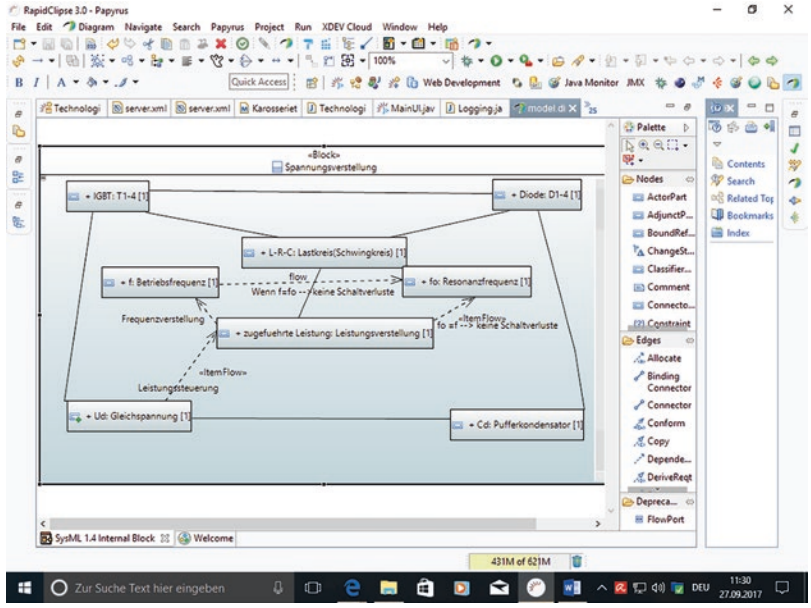


Abb. 4.45 Darstellung der Struktur der Schaltung eines Wechselrichters mit einem Viereck-Modellierung angeschlossen an eine Dreieck-Modellierung zum Ermöglichen eines kapazitiven und induktiven Verhaltens

Abb. 4.38 zeigt die Verbindungen zwischen Gleichspannung, Kondensator, Transistoren IGBT (T1-4) und Dioden (D1-4) zur Modellierung der Funktionalität des Lastgeführten Wechselrichters wie z. B. Schwingkreiswechselrichters. Hierbei dient der Kondensator mit der Kapazität C_d zur Pufferung (Stützung) der Versorgungsgleichspannung, auch genannt Gleichspannung U_d. Gemäß Abb. 4.38, 4.39, 4.40, 4.41, 4.42 und 4.43 arbeiten die vier Transistoren in Gegentakt, d. h., T1 und T4 werden gleichzeitig ein- und ausgeschaltet. Das gilt ebenso für T2 und T3. Dioden D1 und D4 bzw. D2 und D3 gehen in Sperrung mit negativer Sperrspannung während des Einschaltens von T1 und T4 bzw. des Ausschaltens von T2 und T3 über. Abb. 4.38, 4.39, 4.40, 4.41, 4.42, 4.44 und 4.45 zeigen die Verbindungen zwischen Komponenten des Schwingkreiswechselrichters.

Die Funktionalität des Schwingkreiswechselrichters ist mithilfe der Abb. 4.39, 4.40, 4.41, 4.42, 4.43, 4.44 und 4.45 bezüglich der Elemente des Lastkreises, d. h. Widerstand R, Induktivität L und Kapazität C, verdeutlicht. Die drei Parts *„IGBT“*, *„Diode“* und *„L-R-C“* stellen Verbindungen mithilfe von Konnektoren zur Modellierung der Funktionen der Transistoren, Dioden und Resonanzelemente im Schwingkreis dar. Wobei *„IGBT“*, *„Diode“* und *„L-R-C“* Attribute mit entsprechenden Blöcke *„T1-4“*, *„D1-4“* und *„Lastkreis(Schwingkreis)“* in einer Dreieck-Modellierung darstellen. Außerdem bilden die Parts *„f:Betriebsfrequenz“*, *„f_o:Resonanzfrequenz“* und *„zugefuehrte Leistung: Leistungsverstellung“* mithilfe von Informationsobjektflüssen genannt *„Item flow“* eine zweite Dreieck-Modellierung zum einen zur Analyse der Schaltverluste und zum anderen zur Verstellung der Leistung durch Verstellung der Frequenz, wie es auf den Abb. 4.44, 4.40, 4.41, 4.42, 4.43, 4.44 und 4.45 zu sehen ist. Auf den Abb. 4.44, 4.40, 4.41, 4.42, 4.43, 4.44 und 4.45 gibt es Informationsobjektflüsse von dem Part *„$U_{d:}$Gleichspannung“* über dem Part *„zugefuehrte Leistung: Leistungsverstellung“* bis zum Part *„f:Betriebsfrequenz“* zum Darstellen der Analyse der Leistungssteuerungen sowohl durch Verstellung der Versorgungsgleichspannung als auch durch Verstellung der Betriebsfrequenz. Eine Viereck-Modellierung mit den Parts *„IGBT“*, *„Diode“*, *„C_d“* und *„U_d“* stellen die Struktur der Schaltung eines Wechselrichters dar. Der angeschlossene Lastkreis im Viereck gibt einen Überblick sowohl über ein kapazitives Verhalten als auch über ein induktives Verhalten mithilfe des Parts *„L-R-C“*. Gemäß Abb. 4.41, 4.40, 4.41, 4.42, 4.43, 4.44 und 4.45 arbeitet der Wechselrichter mit einer Betriebsfrequenz, dass der angeschlossene Lastkreis sowohl ein kapazitives als auch ein induktives Verhalten hat.

4.3 Anforderungsdiagramm

Ein Anforderungsdiagramm, genannt *„Requirement Diagram“*, mit der Abkürzung REQ stellt ein benutzerdefiniertes Diagramm der nichtfunktionellen Anforderungen eines Systems mithilfe eines visuellen Modells dar.

Eclipse-Papyrus unterstützt die SysML-Modellierung für eingebettete Systeme und andere komplexe Geräte.

In einem eingebetteten System werden sowohl Input als auch Output erzeugt. Außerdem sind interne Operationen und Zustände des Systems nicht durchsichtig. Dies führt zu Problemen bei der Steuerung eingebetteter Systeme während des Entwickelns, Testens und Debuggens von Software. Bei der Modellierung von Software mit SysML-Papyrus werden rückverfolgbare Vorlagen erstellt, mithilfe von Anforderungsdiagrammen des Frameworks Eclipse-Papyrus werden das Planen, Entwerfen und Dokumentieren des Systems ermöglicht.

Der Bedarf vom Markt an Entwicklungen von komplexeren Systemen ist enorm, aber der hohe Anspruch an GUI und Qualität ist konstant und stellt für viele Projekte eine zunehmende Herausforderung für die Beschreibungen der Anforderungen der Systeme dar. Anforderungsmodellierungen des Systems mit SysML bezüglich des Einsatzes des Anforderungsdiagramms, genannt *Requirement Diagram,* ermöglicht das Annehmen dieser Herausforderung. Ein Anforderungsdiagramm stellt den Diagrammtyp zur Beschreibung von Funktion, Leistung und Schnittstelle mit dem Ziel des Ersetzens des „Use Case Diagrams“ von UML dar [3].

Das REQ ermöglicht das präzise Beschreiben des Systems zum Entwickeln nach den Anforderungen der Kunden. Das Ziel des Modellierens der Anforderungen mithilfe der SysML ist es, das Beschreiben der nichtfunktionalen Anforderungen zu ermöglichen. Dank der Standardisierung von SysML und deren Unterstützung mithilfe der Werkzeuge wie z. B. Framework Eclipse-Papyrus, SysML-Designer oder SysML-Modelio sind die Modellierungen der Anforderungen auf Basis von SysML anspruchsvoll.

4.3.1 Anforderungsdiagramm des Schwingkreiswechselrichters mit Papyrus

Schwingkreiswechselrichter stellen Resonanzstromrichter bezüglich der Anwesenheit der Resonanzelemente des Lastkreises in der Schaltung dar. Hierbei besteht der Lastkreis aus einem Schwingkreis, welcher sowohl ein induktives als auch ein kapazitives Verhalten mit Drosselspule bzw. Kondensator zeigt. Anforderungen der Schaltungen mit dem Schwingkreiswechselrichter fokussieren zum einen auf Reduzieren der Verluste dank sowohl des induktiven als auch des kapazitiven Verhaltens der Last. Zum anderen fokussieren sie auf den Erhalt eines sinusförmigen Verlaufs des Laststroms oder der Lastspannung. Außerdem zeigt es sich, dass, bei hoher Frequenz, ein Betrieb mit induktivem Lastverhalten oder mit einer Frequenz, die sich sehr nahe bei der Resonanzfrequenz liegt, Vorteile hat [6]. Eine wichtige Anforderung für den Resonanzwechselrichter ist die Verwendung der abschaltbaren Leistungshalbleiter wie z. B. IGBT.

Abb. 4.46 zeigt das Erstellen des Anforderungsdiagramms mithilfe des Frameworks SysML-Papyrus. Abb. 4.47, 4.48, 4.49, 4.50, 4.51 und 4.52 beschreiben die Anforderungen zur Funktionalität eines Resonanzwechselrichters. Das *Anforderungsd*iagramm *(Requirement Diagram)* mit der Kennung *„Requirement“* im Kontext stellt eine Anforderung dar. Auf den Abb. 4.46, 4.47, 4.48, 4.49, 4.50, 4.51 und 4.52 sind

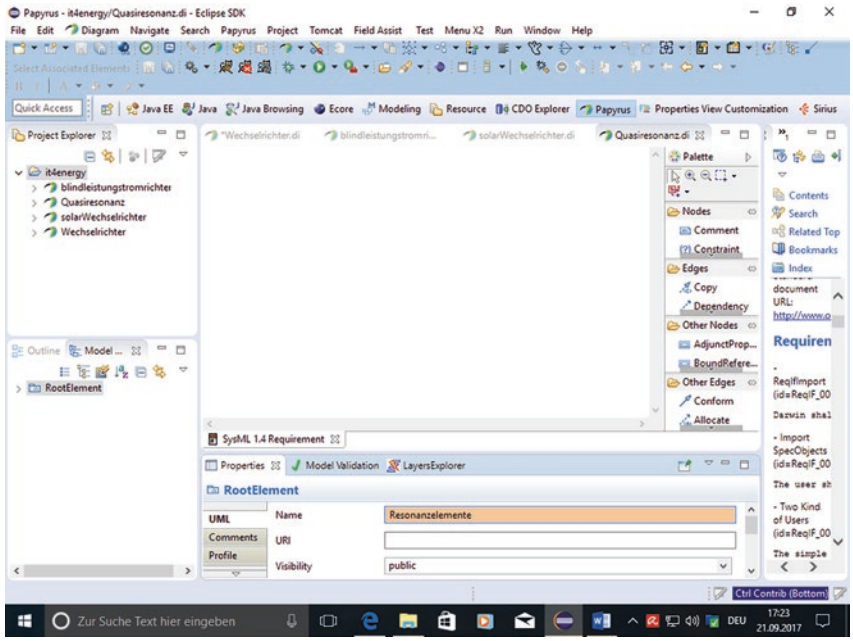


Abb. 4.46 Das Erstellen des Anforderungsdiagramms *„Quasiresonanz“* im Projekt *„it4energy“* mithilfe des Frameworks SysML-Papyrus

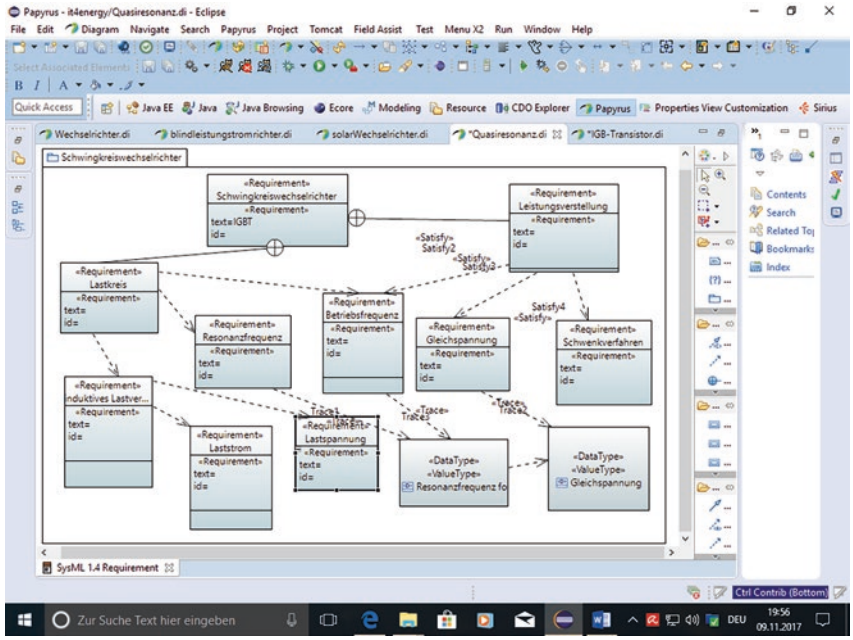


Abb. 4.47 Überblick über die Anforderungen zum Beschreiben der Funktionalität des Resonanzwechselrichters im Hinblick auf die Beziehungen zwischen den Anforderungen

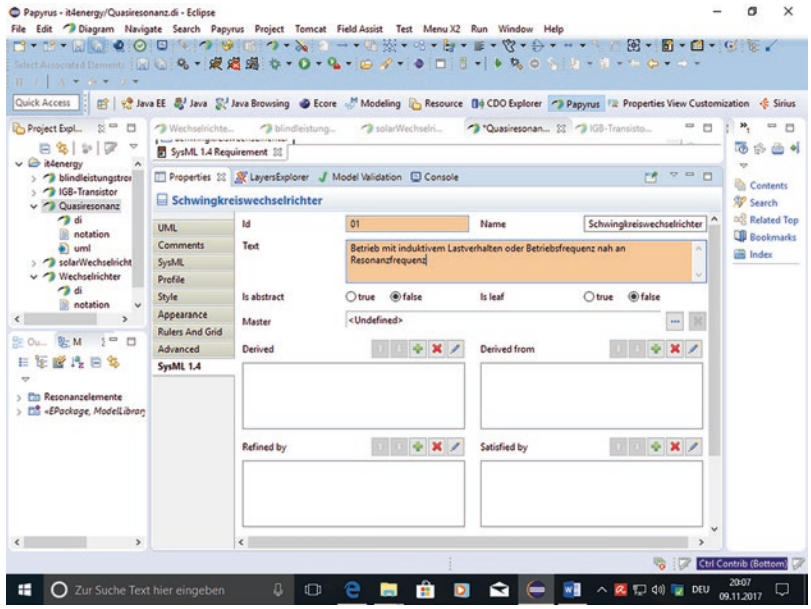


Abb. 4.48 Darstellung der Struktur einer Anforderung im Hinblick auf *„id“*, *„Name“* und *„Text“*

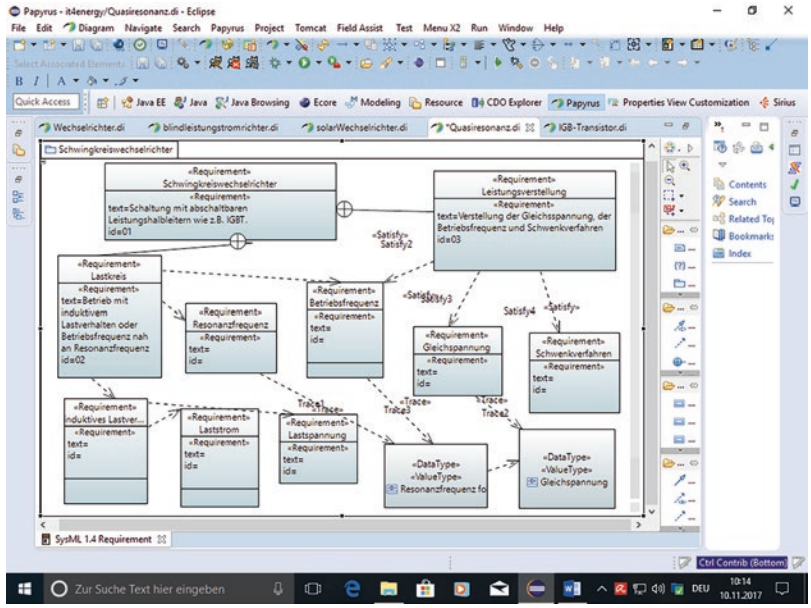


Abb. 4.49 Überblick über die Beschreibungen der Anforderungen *„Schwingkreiswechselrichter“*, *„Lastkreis“* und *„Leistungsverstellung“* mithilfe von der Identifikationssnummer *„id“* und dem Text *„text“*

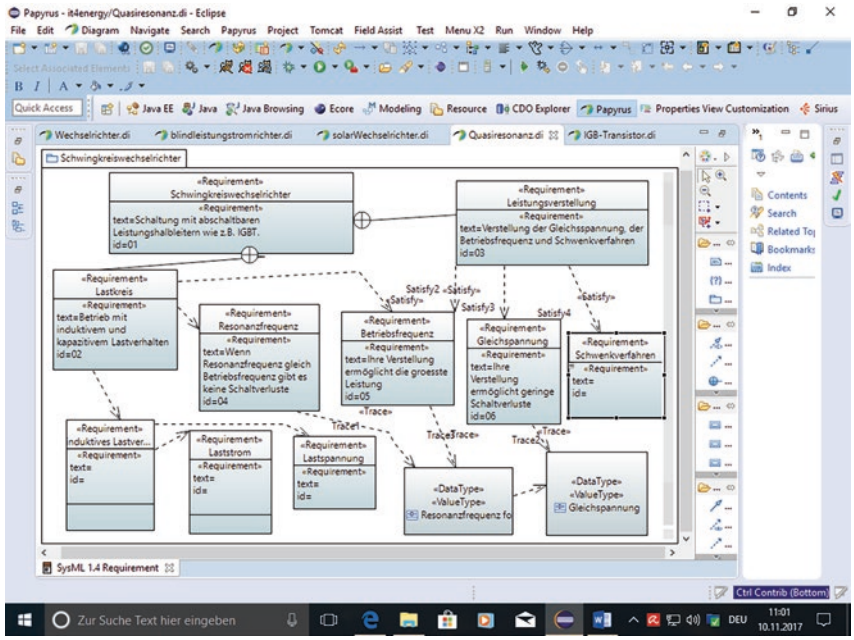


Abb. 4.50 Überblick über die vertikale Modellierung zum Beschreiben der Ableitung der Anforderung im Hinblick auf die Beziehungen zwischen den abgeleiteten Anforderungen und untergeordneten Anforderungen

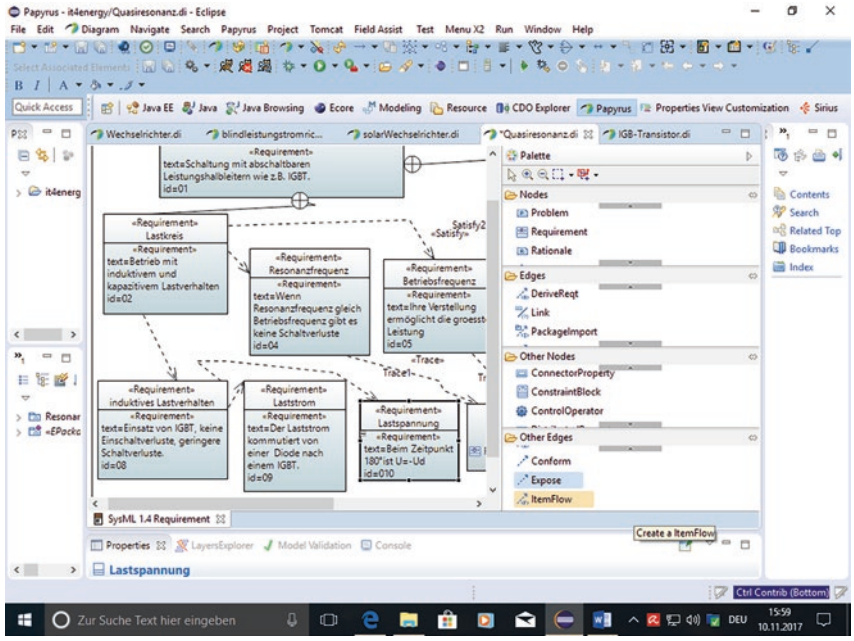


Abb. 4.51 Überblick über die Kaskade-Modellierung zum Beschreiben der Beziehungen zwischen den Anforderungen im Hinblick auf die Verfolgung, genannt *„Trace“*

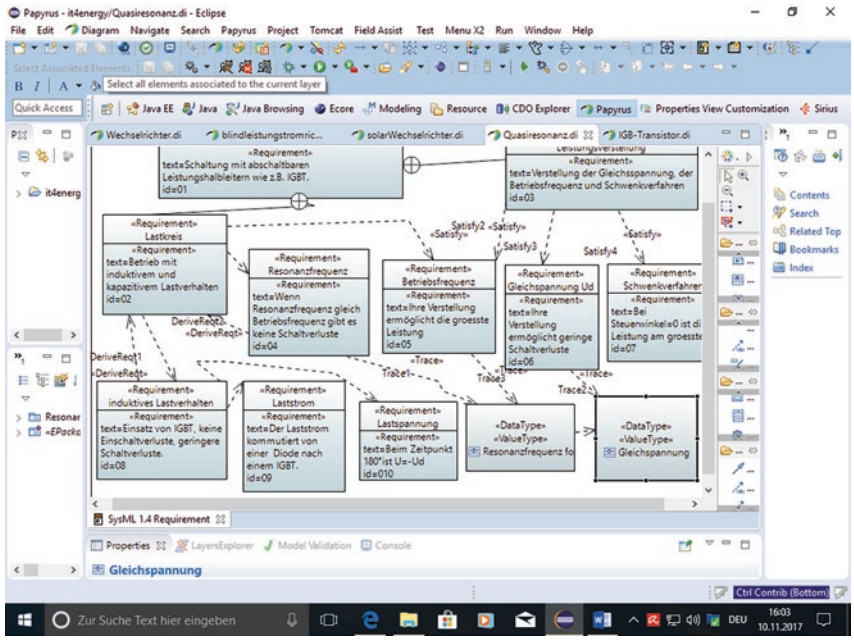


Abb. 4.52 Darstellung der ausgefüllten Anforderungen und deren Beziehungen *„Satisfy“*, *„DeriveReq“* und *„Trace“* zum Beschreiben sowohl einer vertikalen Modellierung als auch einer Kaskadenmodellierung

die Anforderungen an das System genannt *„Schwingkreiswechselrichter“*, mithilfe der Verbindungen zwischen diesen und deren Bezüge zu anderen Modellelementen aufgezeigt. Gemäß Abb. 4.46, 4.47, 4.48, 4.49, 4.50, 4.51 und 4.52 verfügt jede Anforderung über einen Namen, eine Identifikationsnummer, genannt *„id“*, und einen Text, genannt *„text“*. Die verschiedenen *Verbinder (Connector)* koppeln die einzelnen Elemente und beschreiben dabei den Typ der Verbindung. Die *Ableitung,* genannt *„Derive“,* zeigt, aus welchem Kontext eine Anforderung abgeleitet wird. Lässt sich eine Anforderung von einer anderen Anforderung ableiten, wird ihre Relation als „Derive Requirement Relationship“ bezeichnet. Sie ist von der abgeleiteten Anforderung auf die ursprüngliche Anforderung gerichtet und wird mit „DeriveReqt“ gekennzeichnet. Die Verfeinerung *„Refine“* zerlegt einen groben Block in mehrere Teilblöcke. Die Verfolgbarkeit, genannt *„Trace“,* stellt eine allgemeine Beziehung zwischen zwei Anforderungen dar. Besteht eine Beziehung zwischen einer Anforderung und einem beliebigen Modellelement, handelt es sich um eine *„Trace Relationship“*, vorausgesetzt, ein Fall von „Traceability“ liegt vor. Sie wird mit *„Trace“* gekennzeichnet. Sie ist beispielsweise von einer funktionalen Anforderung auf eine nichtfunktionale Anforderung gerichtet. Wird eine Anforderung von einem Designelement erfüllt, handelt es sich um eine *„Satisfy Relationship“*. Sie ist vom Designelement zur Anforderung gerichtet und wird mit *„Satisfy“* gekennzeichnet [7].

Abb. 4.47 und 4.48 geben Überblicke über verschiedene Anforderungen zum Spezifizieren des Resonanzwechselrichters bezüglich der Funktionalität des Resonanzfalls beim Lastkreis sowohl mit dem induktiven als auch dem kapazitiven Verhalten. Dies ist mit Einsätzen der abschaltbaren Leistungshalbleiter wie z. B. IGBT oder GTO erforderlich. Die erste Anforderung mit der Identifikationsnummer *„id = 01“* ist der Einsatz von IGBT, wie es auf den Abb. 4.48, 4.49, 4.50, 4.51 und 4.52 zu sehen ist. Die Modellierung der Schaltung des Schwingkreiswechselrichters mithilfe des Anforderungsdiagrammes des Frameworks SysML-Papyrus beginnt mit der Anforderung „Schwingkreiswechselrichter“ (id = 01), die mithilfe des Symbols *„Containment Link“* in zwei Anforderungen *„Lastkreis“* und *„Leistungsverstellung“* entsprechenden den Identifikationsnummern *„id = 02“* bzw. *„id = 03“* zerlegt ist. Gemäß Abb. 4.46 und 4.47 sind die Anforderungen 2 und 3 diejenigen, die Funktionalität der Resonanzelemente bzw. der Verstellung der zugeführten Leistung der Last ermöglichen. Req2 zeigt sowohl das induktive als auch das kapazitive Verhalten des Lastkreises. Dies stellt die Eigenschaft dieses Resonanzwechselrichters bezüglich der Wirkung der Resonanzelemente Drosselspule und Kondensator dar. Req3 verfügt über drei Verfahren zum Verstellen der zugeführten Leistung der Last: Verstellung der Versorgungsgleichspannung, Verstellung der Betriebsfrequenz und Schwenkverfahren. Von Requirement2 hängen drei Anforderungen *„Requirement4-6“* mit den entsprechenden Namen *„Resonanzfrequenz“*, *„Betriebsfrequenz“* und *„induktives Verhalten“*. Requirement3, genannt *„Leistungsverstellung“*, ist in Beziehungsverbindungen mit den Requirement5-7, mit den entsprechenden Namen *„Betriebsfrequenz“*, *„Gleichspannung“* und *„Schwenkverfahren“*. Gemäß Abb. 4.46 und 4.47 bezüglich des Requirement5 tritt die größte Leistung auf, wenn die Betriebsfrequenz gleich der Resonanzfrequenz gewählt wird. In Bezug auf die Leistungssteuerung durch Verstellung der Versorgungsspannung hat Requirement6 den Vorteil, dass die Betriebsfrequenz sehr nahe der Resonanzfrequenz gewählt wird. Requirement8 hat Abhängigkeitsbeziehungen zu Requirement9 und 10. Dies bedeutet, dass diese Beziehungen, genannt „Dependency“, den zeitlichen Verlauf des Laststroms und der Lastspannung bei sowohl kapazitivem als auch induktivem Lastverhalten, wobei das „Requirement“ für das kapazitive Lastverhalten wegen Platzmangel nicht dargestellt ist. Auf der Abb. 4.52 sind die Beziehungen *„Satisfy“, „DeriveReq“* und *„Trace“* zwischen Anforderungen *„Requirement“* aufgezeigt. Mit *„Satisfy“* werden die Anforderungen „Requirement4-7“ die sowohl die Funktionalität des Resonanzwechselrichters als auch die Leistungsverstellung erfüllen. Hierbei erfüllen einerseits Requirement4 und 5 die Funktion sowohl des induktiven als auch des kapazitiven Lastverhaltens. Andererseits erfüllen Requirement5, 6 und 7 die Möglichkeit zur Leistungsverstellung durch drei Verfahren: Verstellung durch Betriebsfrequenz mit Req5, Verstellung durch Gleichspannung mit Req6 und Schwenkverfahren mit Req7. Gemäß Abb. 4.49, 4.50 und 4.52 lassen sich Anforderungen Req 8 und 4 von der Anforderung Req2 mithilfe der Beziehungen *„DeriveReq1“* bzw. *„DeriveReq2“* ableiten. Drei Beziehungen, genannt Verfolgbarkeit, oder Traceabilty mithilfe des Symbols *„Trace“* bestehen zwischen drei Anforderungen

Req4, 5, 6 und den Datentypen „Gleichspannung“ und „Resonanzfrequenz“. D. h. „Trace1“, „Trace2“ und „Trace3“ stellen entsprechende Verfolgbarkeit zwischen Req4 und DataType „Resonanzfrequenz“, Req5 und DataType „Resonanzfrequenz“ bzw. Req6 und DataType „Gleichspannung“ dar.

4.3.2 Anforderungstabelle des Solar-Schwingkreiswechselrichters mit Papyrus

Einspeisung vom Strom aus Photovoltaik-Generatoren in das Wechselspannungsnetz der öffentlichen Versorgung ist die Hauptfunktion des Solar-Wechselrichters. Wechselrichter für PV-Anlagen mit Netzeinspeisung sind durch folgende Funktionen gekennzeichnet:

a) Funktionsprinzip: Pulsweitenmodulation (PWD),
b) Synchronisation der erzeugten PV-Spannung mit der Netzspannung,
c) Sofort, Selbst Abschalten bei Netzausfall oder Netzstörungen,
d) Hoher Wirkungsgrad in Teillastbereichen,
e) Erreichen einer maximalen Leistungsteuerung mithilfe von MPP,
f) Keine abgestrahlte HF-Spannungen auf der Gleich- und Wechselstromseite,
g) Möglichst niedrige Ein- und Ausschaltverluste,
h) Vorschreiben einer Einrichtung zur Netzüberwachung mit Schaltorgan in Reihe sowie
i) (ENS) zur selbsttätigen Freischaltstelle.

Wechselrichter verfügen über wichtige Aufgaben wie z. B. Umwandlung des gelieferten Stroms in den Wechselstrom, Übernehmen der Betriebsführung der gesamten Anlage, Durchführung der Netzüberwachung (wichtig!), Überprüfung des Netzes bei Einspeisungsbedingungen und Lösen der Verbindung Solaranlage-Netz bei einem Fehler.

Anforderungen des PV-Wechselrichters sind:

1. Sinusförmiger Verlauf des eingespeisten Stromes.
2. Kein Überschreiten der Grenzwerte (VDE088, EN60555) der Verzerrungen (Oberwellen) bei Vielfachem der Netzfrequenz.
3. Kein Gleichanteil beim Ausgangsstrom. (Weil der Trafo den Fehlerstrom-Schutzschalter beeinträchtigen kann.)
4. Keine Phasenverschiebung (cos Phi = 1 d. h. Phi = 0) bei eingespeistem Netzstrom und eingespeister Netzspannung.
5. Kein Pendeln der Blindleistung zwischen Netz und Wechselrichter.
6. Automatische Trennung des Wechselrichters vom Netz bei anormalen Betriebszuständen (fehlende oder zu hohe Netzspannung, starke Frequenzabweichung von der Sollfrequenz, Kurzschlüsse oder Isolationsfehler).

Vorschriften einer *E*inrichtung zur *N*etzüberwachung mit *S*chaltorgan in Reihe (ENS) zur selbsttätigen Freischaltstelle für PV-Anlage bis 5 kW sind:

- DC-Freischaltstelle zur Trennung des Wechselrichters bei Wartungs- oder Reparaturarbeit problemlos vom Generator.
- Vermeiden der Ausbildung vom Inselnetz bei abgeschaltetem Versorgungsnetz.
- ENS (für PV-Anlage bis 5 kW) als Verfahren zur Messung der Netzimpedanz, der dynamischen Änderung der Netzimpedanz sowie der Netzspannung und der Frequenz.
- Erkennung des Ausfalls des Netzes und Trennung des Wechselrichters vom Netz über zwei Schaltorgane.
- Rundsteuersignale (Relais) dürfen nicht den Betrieb des Wechselrichters stören.
- Vorliegen einer guten Anpassung an maximaler Solargeneratorleistung auf der Eingangsseite des Wechselrichters.
- Kleine Schwankung (<3 %) der Eingangsspannung aufgrund der bei einphasigen Wechselrichtern mit 50 Hz ins Netz eingespeisten Energie.
- Anwesenheit eines genügend großen Puffers Kondensators am Eingang des Wechselrichters.
- Vorbeugen der Überspannung (Überspannungen dürfen nicht zu Defekten führen!!!).
- Dimensionierung: Nennleistung des WR < Nennleistung des Solargenerators, d. h. Nennleistung des Solargenerators = 0,9 bis 0,9*Nennleistung des WR.
- Abschalten bei Überlast und periodisches Wiederanlaufen.
- Mögliche Selbstversorgung des Netzwechselrichters aus dem Solargenerat.
- Höher Wirkungsgrad des WR (>90 % bei 100 % der Nennleistung) bereits bei kleinen Leistungen.
- Integrierte Selbstüberwachung zur ständigen Überwachung und Ferndiagnose.

Kriterien der ENS bezüglich der Netzüberwachung und Fehlerstromüberwachung sind:

- 195 V < Netzspannung < 260 V,
- Frequenzabweichung > 0,5 Hz,
- Impedanz-Sprung > 0,2 O.

Das Praxisbeispiel zeigt die Modellierung der Anforderungen zur Funktionalität des Solar-Wechselrichters bezüglich der Netzsicherheit und Netzüberwachung durch den Wechselrichter. Hierbei werden die Anforderungen für die Funktionalität des Wechselrichters mithilfe einer Anforderungstabelle mit SysML1.4 spezifiziert. Abb. 4.53, 4.54, 4.55, 4.56, 4.57, 4.58, 4.59, 4.60, 4.61, 4.62, 4.63 und 4.64 zeigen das Erstellen der Anforderungstabelle, genannt *„SysML1.4 Requirement Table“*. Die Modellierung mit *„SysML1.4 Requirement Table“* erfordert das Erstellen sowohl von dem Projektnamen, genannt *„REQWechselrichter“*, als auch vom Modellnamen, genannt *„REQ“*,

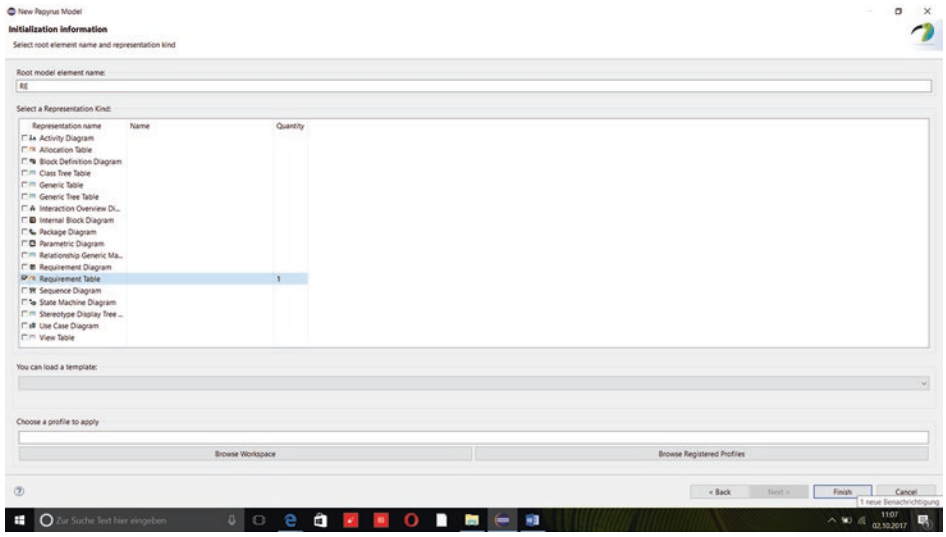


Abb. 4.53 Überblick über die Auswahl eines Anforderungstabelle mit Eclipse-Papyrus

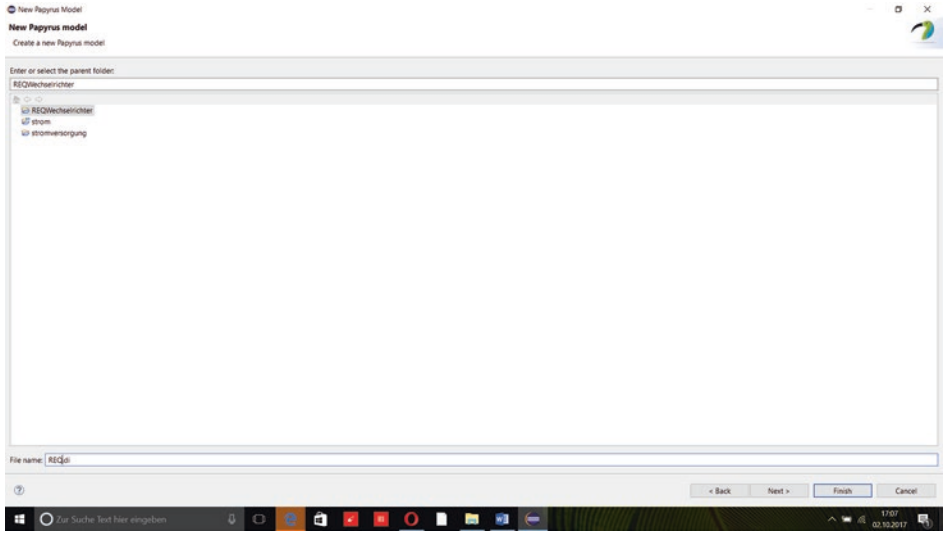


Abb. 4.54 Das Erstellen des Modells genannt *„REQ“* im Projekt *„REQWechselrichter“*

wie es auf den Abb. 4.53 und 4.54 zu sehen ist. Mit der Abb. 4.55 wird geprüft, ob die *„Traceabilty Tools“* von *„Papyrus Req“* zum Modellieren von Anforderungen wie z. B. *„Satisfy“*, *„Refines“*, *„Realizes“* oder *„Verifies“* vorhanden sind. Anschließend wird mithilfe der Umgebung Papyrus im Untermenü *„Preference“* von Menü *„Window“*

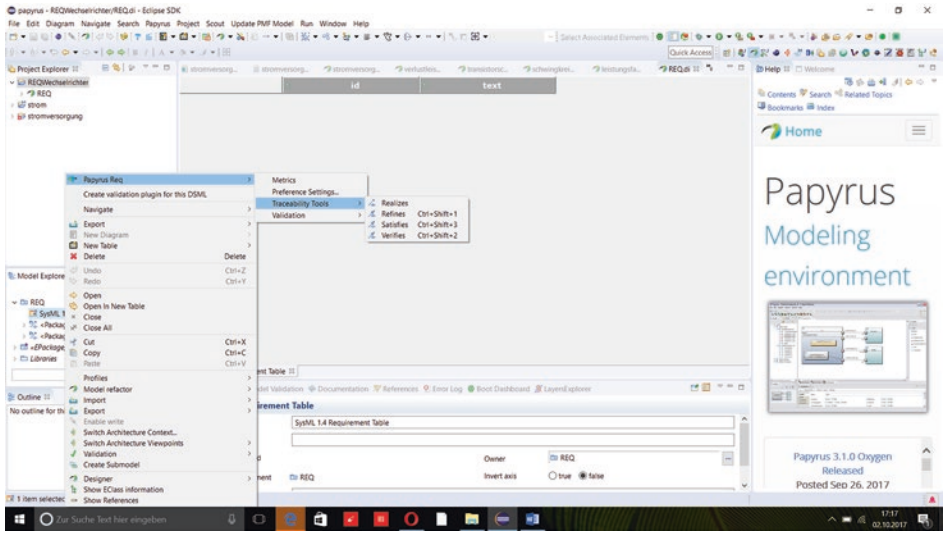


Abb. 4.55 Das Prüfen des Vorhandenseins der *„Traceabilty Tools“* von *„Papyrus Req“* zum Modellieren von Anforderungen

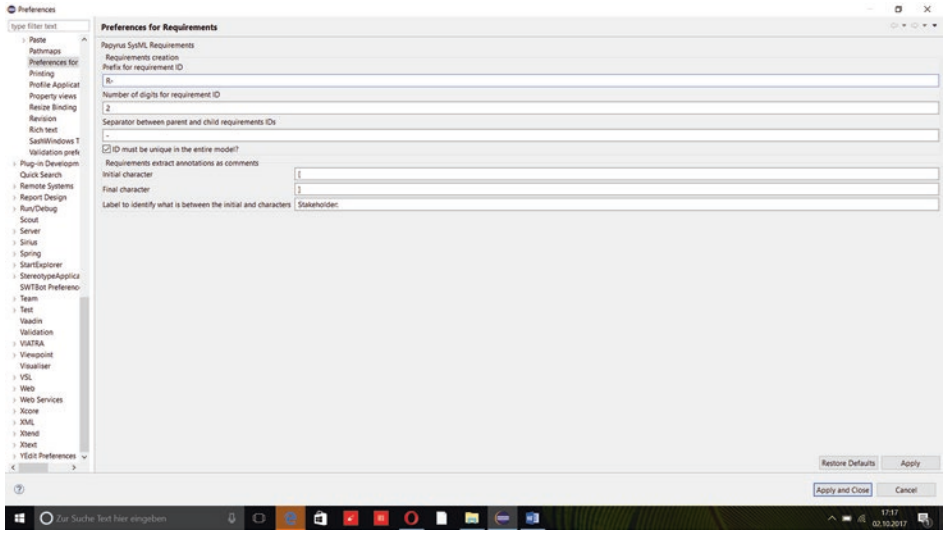


Abb. 4.56 Einstellungen für das Erstellen von Identifikationsnummer wie z. B. *„REQ“* mit Papyrus-SysML Requirement

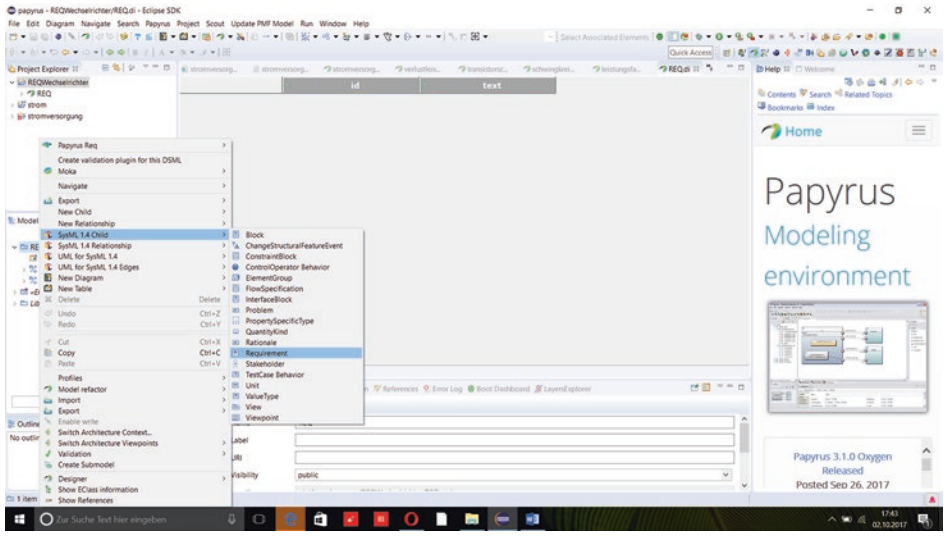


Abb. 4.57 Das Hinzufügen von Anforderungen, genannt *„Requirement“*, in die Tabelle mithilfe von *„SysML-Child“*

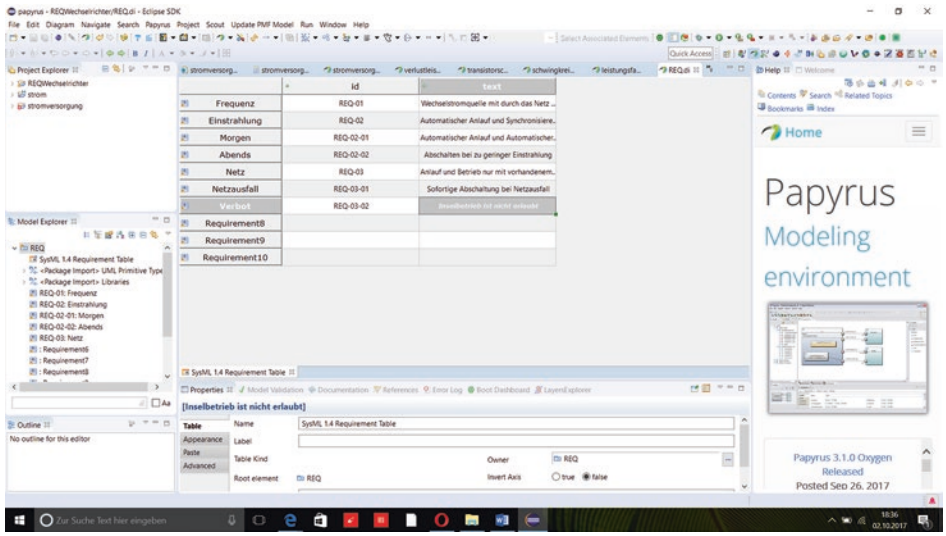


Abb. 4.58 Überblick über die Spalten der Anforderungen im Hinblick auf *„name“*, *„id“* und *„text“*

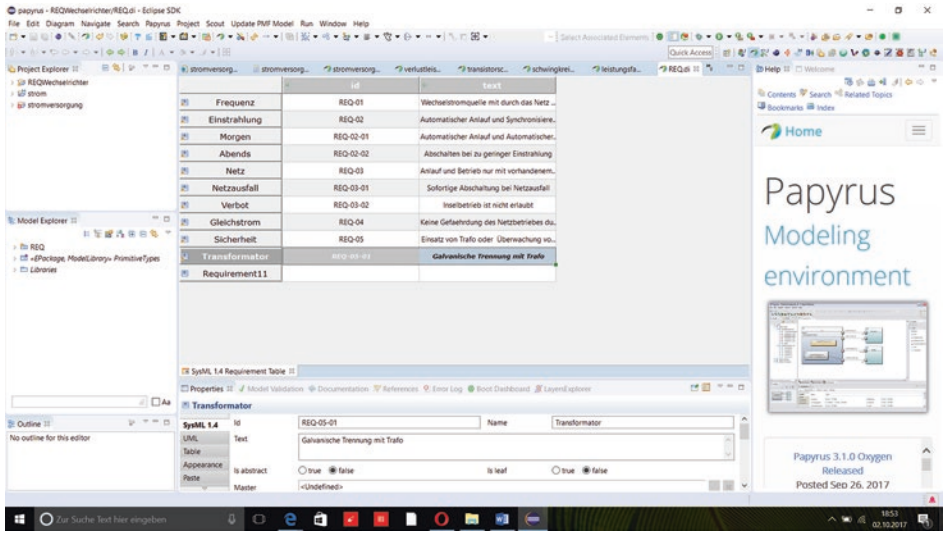


Abb. 4.59 Darstellung der Anforderungstabelle nach vertikaler und horizontaler Ausrichtung im Hinblick auf die Struktur der Anforderung *„REQ-05–01“*

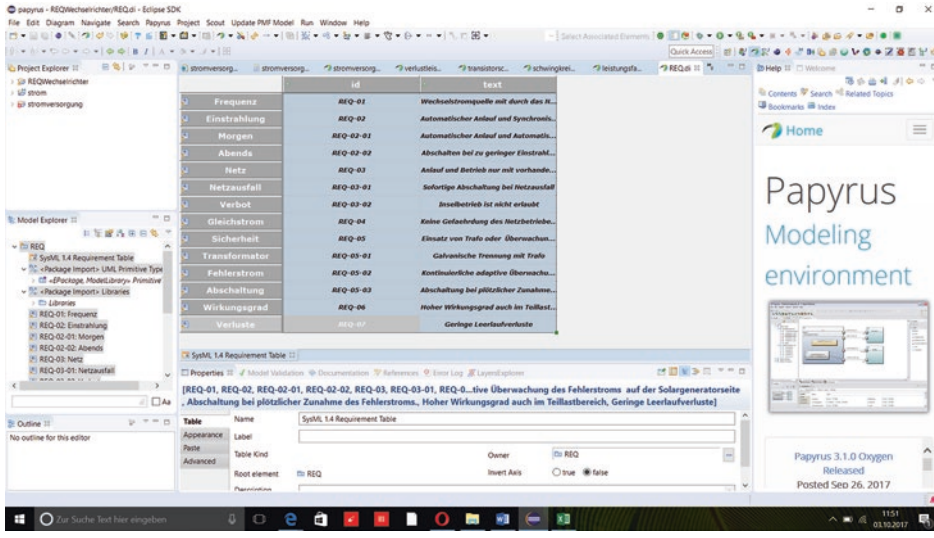


Abb. 4.60 Überblick über die Markierung der Spalten und Zeilen der Anforderungstabelle

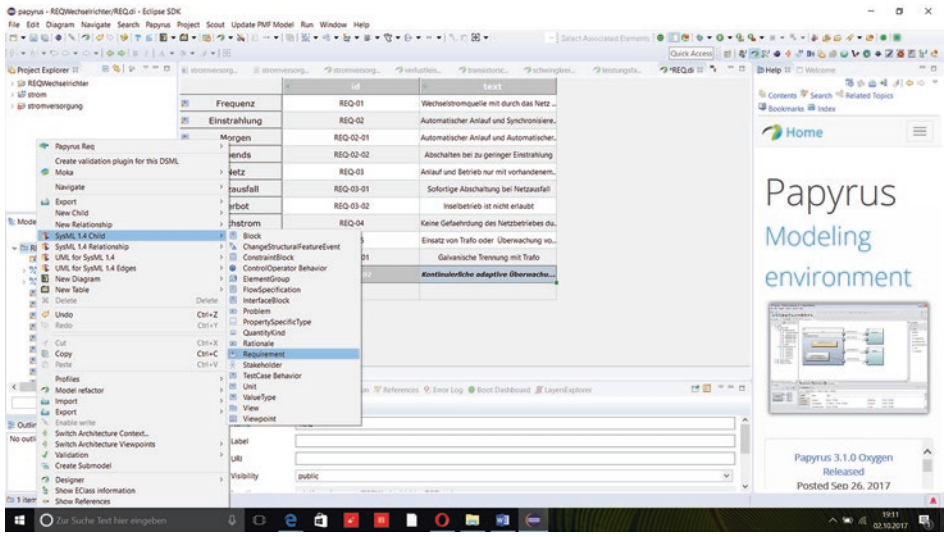
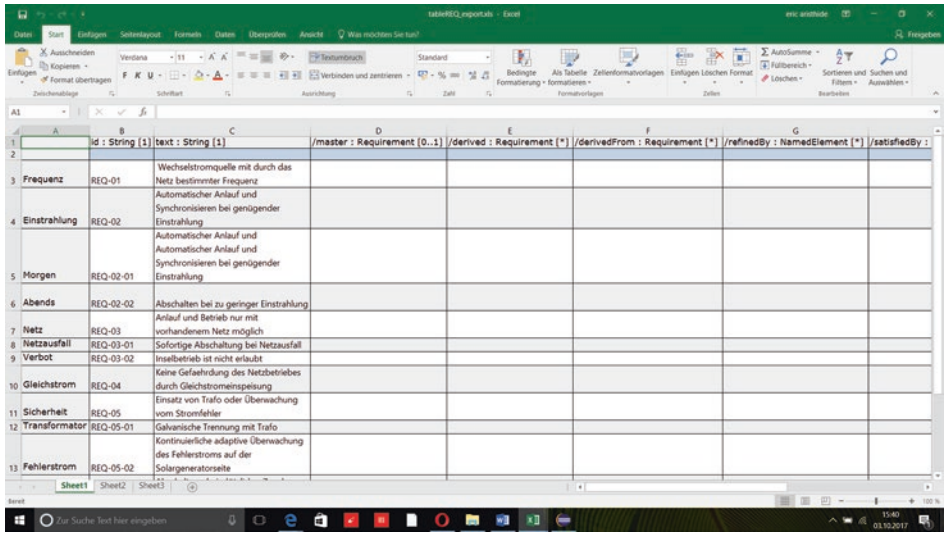


Abb. 4.61 Das Hinzufügen von Anforderungen, genannt *„Requirement“*, in die Tabelle mithilfe von *„SysML-Child“*



	A	B	C	D	E	F	G	
1		id : String [1]	text : String [1]	/master : Requirement [0..1]	/derived : Requirement [*]	/derivedFrom : Requirement [*]	/refinedBy : NamedElement [*]	/satisfiedBy :
2								
3	Frequenz	REQ-01	Wechselstromquelle mit durch das Netz bestimmter Frequenz					
4	Einstrahlung	REQ-02	Automatischer Anlauf und Synchronisieren bei genügender Einstrahlung					
5	Morgen	REQ-02-01	Automatischer Anlauf und Automatischer Anlauf und Synchronisieren bei genügender Einstrahlung					
6	Abends	REQ-02-02	Abschalten bei zu geringer Einstrahlung					
7	Netz	REQ-03	Anlauf und Betrieb nur mit vorhandenem Netz möglich					
8	Netzausfall	REQ-03-01	Sofortige Abschaltung bei Netzausfall					
9	Verbot	REQ-03-02	Inselbetrieb ist nicht erlaubt					
10	Gleichstrom	REQ-04	Keine Gefaehrdung des Netzbetriebes durch Gleichstromeinspeisung					
11	Sicherheit	REQ-05	Einsatz von Trafo oder Überwachung vom Stromfehler					
12	Transformator	REQ-05-01	Galvanische Trennung mit Trafo					
13	Fehlerstrom	REQ-05-02	Kontinuierliche adaptive Überwachung des Fehlerstroms auf der Solargeneratorseite					

Abb. 4.62 Der Export der SysML-Anforderungstabelle von Eclipse-Papyrus in Microsoft Office-Excel-Editor für die entsprechende csv-Datei wie zum Beispiel *„REQ.csv“*

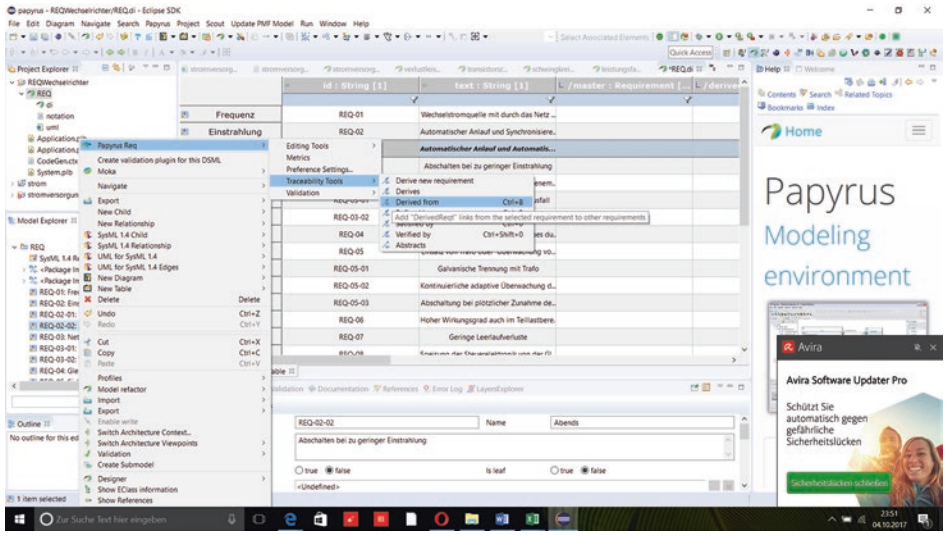


Abb. 4.63 Das Hinzufügen von Verfolgbarkeit, genannt *„Traceability“*, mit *„Papyrus Req“* und *„Traceability Tools“*

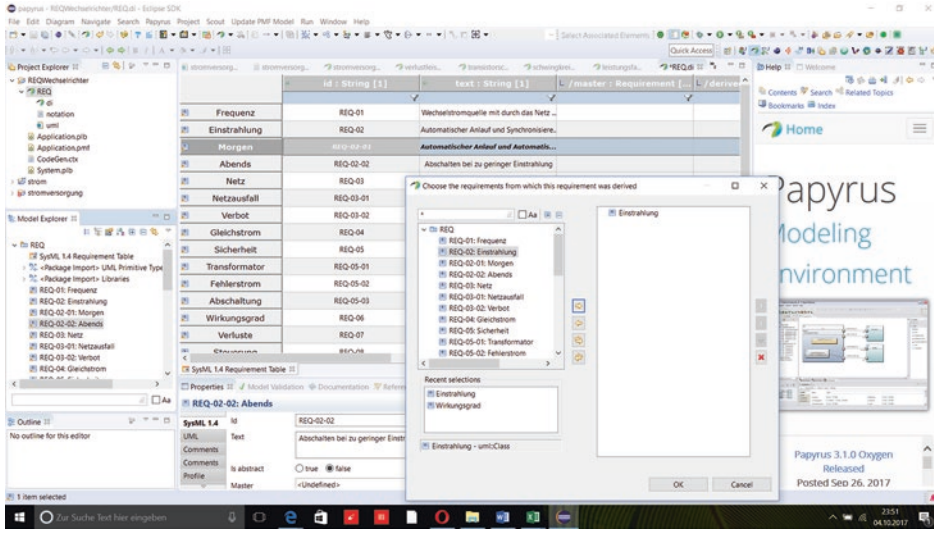


Abb. 4.64 Beschreibungen der Anforderungen nach der Auswahl von Verbindungen oder Relationen mithilfe von Traceability Tools

die Einstellungen für das Erstellen von Identifikationsnummer wie z. B. *„REQ“* mit Papyrus-SysML Requirement realisiert, wie es auf der Abb. 4.56 zu sehen ist. Abb. 4.57 zeigt das Hinzufügen von Anforderungen, genannt *Requirement*, in die Tabelle mithilfe von „SysML-Child“. Die Requirement-Tabelle besteht aus mehreren Spalten nach dem Alphabet für Requirement-Nummer, Text, ID usw. Abb. 4.58, 4.59, 4.60, 4.61, 4.62, 4.63 und 4.64 geben Überblicke über die Beschreibungen der Anforderungen der Funktionalität des Solar-Wechselrichters bezüglich zum einem der Einspeisung von Strom ins öffentlichen Wechselspannungsnetz und zum anderen der Sicherheitsmaßnahmen während des Betreibens des Wechselrichters ohne Transformator. Das Hinzufügen von Verfolgbarkeit, genannt *Traceability,* ist über *„Papyrus Req“* nach der Auswahl von Verbindungen oder Relationen mithilfe von Traceability Tools ermöglicht, wie es auf der Abb. 4.63 zu sehen ist. Die Anforderungen sind mithilfe von id-, text- und Relation-Spalten beschrieben, wobei vor der id-Spalte die Spalte für die entsprechende Anforderung steht, wie es auf den Abb. 4.60, 4.61, 4.62, 4.63 und 4.64 zu sehen ist. Beispielsweise entsprechen REQ-01, REQ-02, REQ-02–01, REQ-02–02 die Anforderungen *„Frequenz“*, *„Einstrahlung“*, *„Morgen“* und *„Abends“* und damit die folgenden Texte *„Wechselstromquelle durch das Netz bestimmter Frequenz“*, *„Automatischer Anlauf und Synchronisieren bei genügender Einstrahlung“*, und *„Abschalten bei zu geringer Einstrahlung“*. Der Export der SysML-Anforderungstabelle von Eclipse-Papyrus kann in Microsoft Office-Excel-Editor für die entsprechende csv-Datei wie zum Beispiel *„REQ.csv“* realisiert werden. Detailinformation über die Struktur einer Anforderung wie z. B. *„id“*, *„Name“* oder *„Text“* ist mithilfe von *„Properties“* von SysML Requirement Table unter der Tabelle zu entnehmen, wie es auf der Abb. 4.62 zu sehen ist. Gemäß Abb. 4.62 sieht man in der Struktur von der Tabelle folgende Detailinformation: id = REQ-05–02, Name = Fehlerstrom, Text = Kontinuierliche adaptive Überwachung des Fehlerstroms auf der Solargeneratorseite. Diese Information entspricht der Beschreibung der Anforderung REQ-05-02.

4.4 Zusicherungsdiagramm (Parametrisierdiagramm)

Zusicherungsdiagramme stellen den Zusammenhang der Eigenschaften der Blöcke bezüglich der Formulierungen der Constraints mithilfe der Werten der Eigenschaften dar. Systeme verfügen über Systemelemente, deren veränderbaren Werte sich gegeneinander beeinflussen [2].

Mithilfe von Zusicherungsdiagrammen werden parametrische Diagramme zwischen Eigenschaften von Systembausteinen ermöglicht. Ziel des Zusicherungsdiagramms ist es, Relationen von einzelnen Systembausteinen mithilfe der Formel darzustellen. Zusicherungsdiagramme sind Strukturdiagramme, die Formulierungen von Constraints zum Verbinden von Parametern mithilfe der Blöcke darstellen. Contraints sind durch geschweiften Klammern gekennzeichnet.

Mithilfe von Eclipse-Papyrus wurde das Zusicherungsdiagramm erstellt. Der Editor zum Erstellen von SysML-Zusicherungsdiagrammen verfügt über Modellierungswerkzeuge, genannt Palette, welche aus vier Teilen bestehen: *„General Annotations“*, *„Blocks“*, *„Ports and Flow“* und *„Constraints“*. *„General* Annotations*“* enthalten Elemente u. a. Constraints, Abstraction, Comment, Context Link, Dependency, Link, Rationale, Problem oder Refine während Elemente wie z. B. Part, ActorPart, Binding Connector, Connector, BoundReference oder Dependency zu *„Blocks“* gehören. Anschließend verfügt *„Port and Flow“* über Elemente wie z. B. FlowPort, Full Port, Port, Item Port und Proxy Port. Der vierte Teil, genannt *„Constraints“*, verfügt über ein einziges Element: Constraints.

4.4.1 Modellierung von Verlusten in „Insulate Gate Bipolar Transistor“ (IGBT) mithilfe von Sicherungsdiagrammen auf Basis von Eclipse-Papyrus-SysML

Dieses Praxisbeispiel ermöglicht die Modellierungen der Verluste in Transistoren, genannt IGBT, mittels Parametrisierdiagramme bezüglich der Analyse der Bestimmungen der Verluste. Zum einen werden Ein- und Ausschaltverlustarbeiten mithilfe der Ein- bzw. Ausschaltverluste und Ein- bzw. Ausschaltperioden bestimmt. Zum anderen wird die durchschnittliche Verlustleistung mithilfe des Verhältnisses der Summe von Ein- und Ausschaltverlustarbeit mit der Summe von Ein- und Ausschaltperiode bestimmt. Wobei die Schaltfrequenz die Berechnung der Periode ermöglicht. Abb. 4.65 und 4.66 zeigen die Modellierung der Verluste beim IGBT mithilfe des Zusicherungsdiagramms von Eclipse-Papyrus. Gemäß Abb. 4.65 und 4.66 gibt es Beziehungsabhängigkeiten zwischen „Constraint“ von *„General Annotation“* und *„ConstraintProperty“* von Constraint. Gemäß Abb. 4.65 und 4.66 sind die ConstraintProperty als Attribute von Blocks dargestellt und mit Zusicherungsparameter verbunden. Das Rechteck der Zusicherung kann mit abgerundeten Ecken, oder als normale Part dargestellt werden. Abb. 4.65 und 4.66 zeigen, dass das Symbol *„Constraints“* von *„General Annotations“* mit einem Fragezeichen gekennzeichnet ist und mithilfe der geschweiften Klammern die Zusicherung darstellt. Wie es auf den Abb. 4.65 und 4.66 zu sehen ist, sind die Abhängigkeitsbeziehungen zwischen Attributen, Blocks und Constraints zum Beschreiben des Zusicherungszustandes wie z. B. zwischen ConstraintProperty *„schaltfrequenz f“* und verschiedene Contraints u. a. *„verlustleistung P_v“*, *„EIN_Verlustarbeit“*, *„AUS_Verlustarbeit“* und *„schaltperiode T“* dargestellt. Gemäß Abb. 4.65 und 4.66 sind die Contraints mithilfe der Formel gekennzeichnet und beschrieben. Dieser Formel entsprechen die Werte der ContraintProperty.

Mithilfe der Zusicherungsdiagramme werden die unterschiedlichen Eigenschaften der Systembausteine und deren parametrische Beziehungen definiert [3]. Das Praxisbeispiel in dem Zusammenhang mit den Abb. 4.65 und 4.66 stellt das Bestimmen der Verlustleistung P_v für IGBT mithilfe der Formel $P_v = 1/2\ f * W_v$(Verlustleistung = 1/2 *

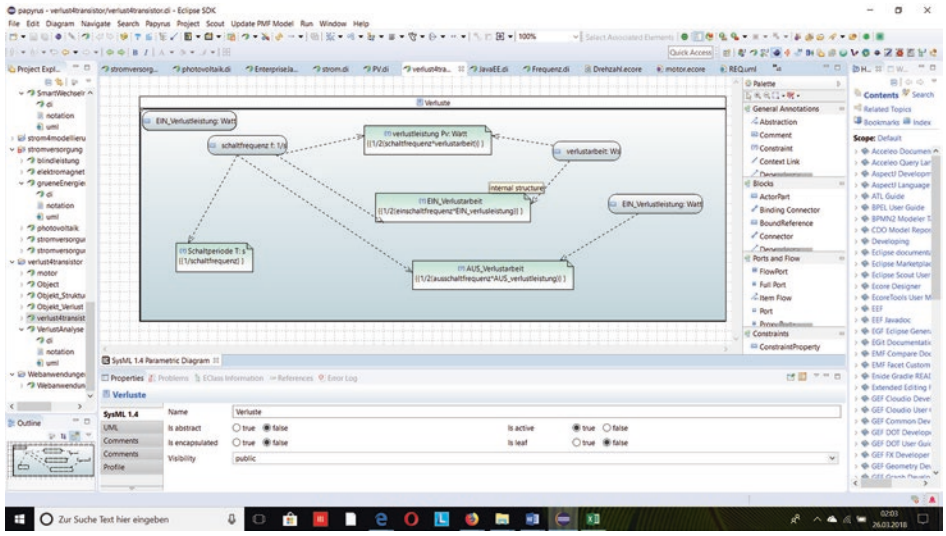


Abb. 4.65 Überblick über das Zusicherungsdiagramm genannt *„verlust4transistor“* im Hinblick auf Beziehungsabhängigkeit zwischen Zusicherungsparameter und *„ConstraintProperty“*

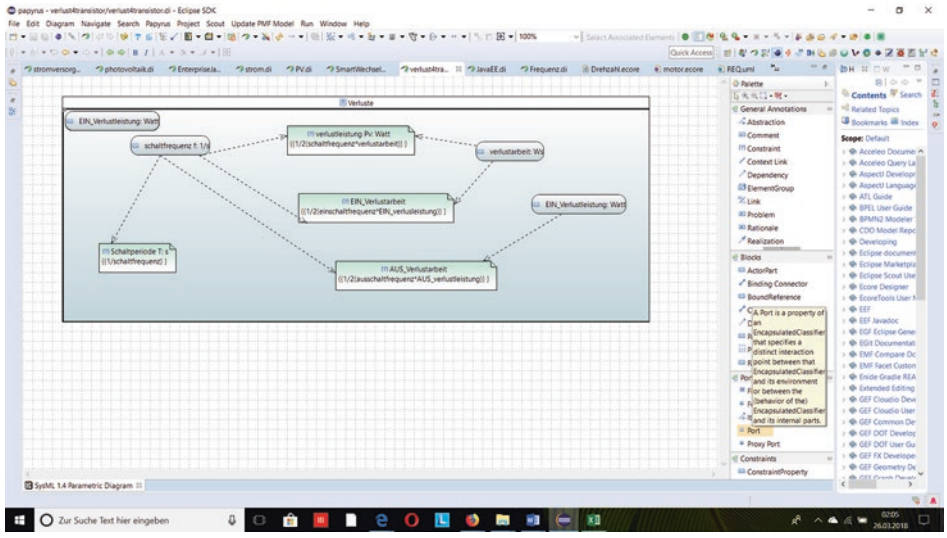


Abb. 4.66 Modellierung der Verluste beim IGBT mithilfe des Zusicherungsdiagramms von Eclipse-Papyrus zum Beschreiben des Zusicherungszustandes mithilfe der Formel im Hinblick auf die Ausrichtung der Abhängigkeitsbeziehungen

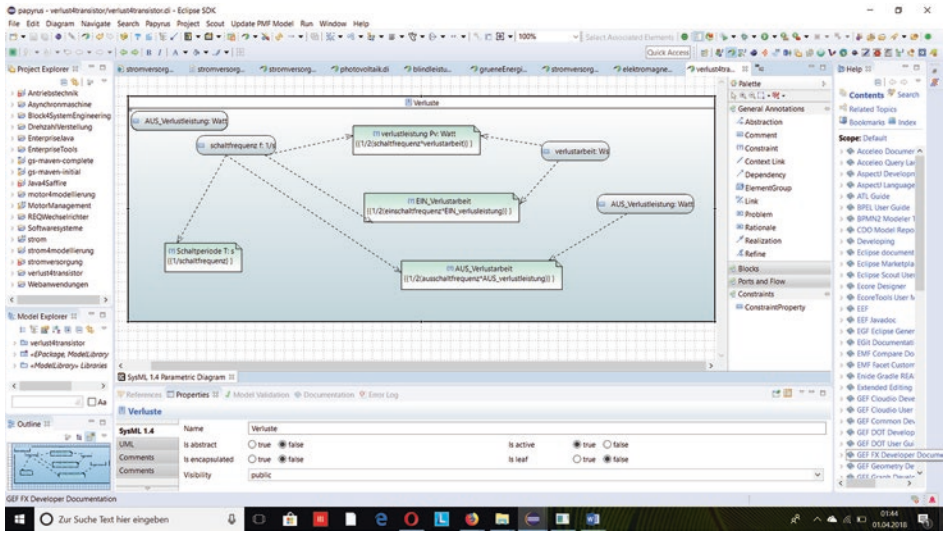


Abb. 4.67 Überblick über die Zusicherungsformeln gekennzeichnet mit Fragezeichen und geschweiften Klammern

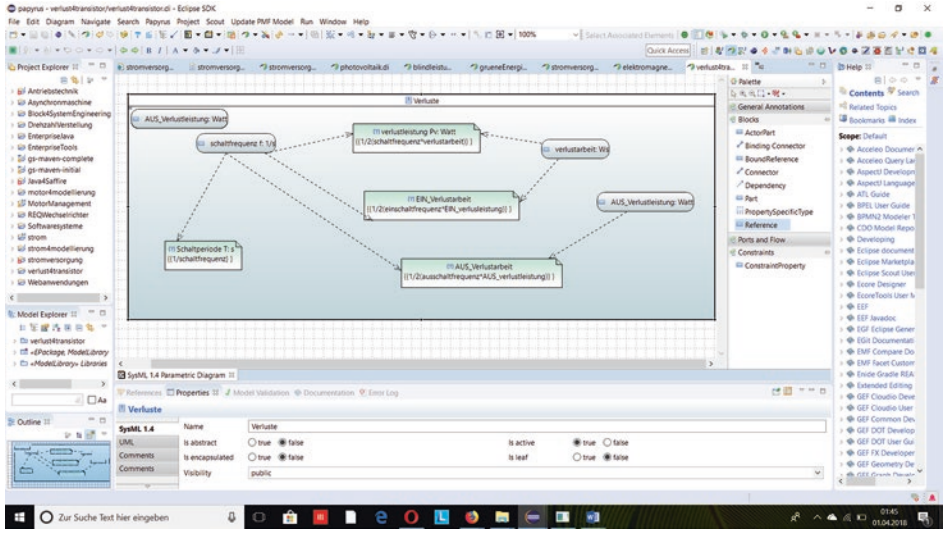


Abb. 4.68 Überblick über den Zusammenhang der Eigenschaften der Blöcke bezüglich der Formulierungen der Constraints im Hinblick auf die Ausrichtung der Elemente von Blocks

Schaltfrequenz/Verlustarbeit) dar. Diese Verlustleistungsformel ist auf das System Transistor bezogen. Die Schaltfrequenz ist die Inverse der Periode. Die Verlustarbeit ist den Parametern des Transistors zu entnehmen. So kann, nach der Formel zum Berechnen der Verlustleistung, auch die Schaltfrequenz bestimmt werden.

4.4.2 Modellierung von Blindleistungen mit Zusicherungsdiagrammen

Oberschwingungen des Stroms mit der sinusförmigen Netzspannung erzeugen nur Blindleistung. Strom-Oberschwingungen tragen nicht zur Lieferung von Wirkleistung bei. Die Formel zum Berechnen der Wirkleitung ist:

$$P = U * I_1 * Cos\ Phi_1 \tag{4.1}$$

Wobei U der Effektivwert der Netzspannung und I_1 der Effektivwert der Strom-Grundschwingung sind. Die Größe $CosPhi_1$ stellt den Leistungsfaktor der Grundschwingung dar und wird als Verschiebungsfaktor bezeichnet [6]. Die Scheinleistung S der Schaltung mit I als der Effektivwert des Laststroms beträgt.

$$S = U * I \tag{4.2}$$

Die benötige Blindleistung Q ist mithilfe der folgenden Formel berechnet:

$$Q = \sqrt{S^2 - P^2} \tag{4.3}$$

Die Grundschwingung erzeugt aufgrund ihrer Phasenverschiebung gegenüber der Spannung auch eine Blindleistung, genannt Grundschwingungsblindleistung Q_1 oder Steuerblindleistung. Daher gilt für Q_1:

$$Q_1 = U * I_1 * Sin\ Phi \tag{4.4}$$

Verzerrungsleistung D stellt die Blindleistung dar, die durch die Abweichung des Stroms von der Sinusform zustande kommt. Wenn die Oberschwingung nach einer Fourier-Analyse bekannt sind, lässt sich D für jede Oberschwingung getrennt berechnet [6].

Für alle Oberschwingungen zusammen lässt D sich mit der folgenden Formel berechnen:

$$D = \sqrt{Q - Q_1^2} \tag{4.5}$$

Abb. 4.67 und 4.68 zeigen die Modellierung der Blindleistung mithilfe des Zusicherungsdiagramms. Mit dem Zusicherungsdiagramm anhand der Abb. 4.67 und 4.68 sind die Zusammenhänge von verschiedenen Eigenschaften des Systems Blindleistung dargestellt worden. Durch das Definieren der Beziehungen einerseits zwischen

Wirkleistung und Leistungsfaktor und andererseits zwischen Wirkleistung und Scheinleistung entsteht ein so genanntes Netzwerk von Beziehungen zur Modellierung von Blindleistungen. Innerhalb dieses genannten Netzwerkes können Beziehungen, wie es auf den Abb. 4.67 und 4.68 zu sehen ist, oder Zusammenhänge mit den Systemen interagieren und damit sie beeinflussen. Z. B. auf den Abb. 4.67 und 4.68 stellen Beziehungen einerseits zwischen Wirkleistung und Leistungsfaktor mithilfe vom Konnektor3(Connector3) genannt *„BindingConnector“*, zwischen Wirkleistung und Grundschwingungsstrom mithilfe vom Konnektor11(Connector9) und andererseits zwischen Grundschwingungsblindleistung und Spannung mithilfe vom Konnektor10, zwischen Grundschwingungsblindleistung und Grundschwingungsstrom mithilfe vom Konnektor9 Netzwerke zum Modellieren des Einsatzes der Konnektoren. Hierbei sind die Werte der Eigenschaften an beiden Enden gleich.

4.5 Zusammenfassung

Die Modellgrößen in einem System, das sich als Zustandsgrößen identifizieren lässt, können sich dabei grundsätzlich zeitkontinuierlich oder zeitdiskret verändern.

SysML zur grafischen Modellierungssprache der Systemmodellierung stellt Unterstützungen bei der Analyse, Spezifikation, Design, Verifikation und Validation von Systemen dar, die aus Hardware, Software, Daten, Personal, Prozessen und technischen Hilfsmitteln (Anlagen) bestehen. Es gibt verschiedene Perspektiven zur Modellierung mit SysML unter anderem Anforderungen, statische Architektur, dynamische Architektur und Interaktionen. Anforderungen, Struktur, Verhalten und Zusicherung stellen die vier Bausteine der SysML-Modellierung dar. Mithilfe des Frameworks Papyrus werden Struktur-, Verhaltens-, Anforderungs- und Parametermodellierung verschiedene Aspekte eines Systems mit SysML modelliert.

Die Strukturmodellierung mithilfe von Blockdefinitionsdiagramm (BDD) und internem Blockdiagramm (IBD) ermöglicht die Modellierung eines Systems bezüglich seiner strukturellen Eigenschaften. Die möglichen Systemelemente in einem realen System werden beschrieben und der strukturelle Aufbau eines Elementes und Verbindungen und Vererbungsbäume zwischen ihnen werden festgelegt.

Blockdefinitionsdiagramme zeigen die Definition von Systembausteinen und ihre Beziehungen. Sie sind wichtige Diagramme aus der Systemmodellierung und haben ihre Ursprünge in der UML, wobei sie die „Klassendiagramme“ darstellen. Bei der SysML-Entstehung wurden Klassendiagramme in Blockdefinitionsdiagramme transformiert. Die Interaktionen zwischen Blöcken werden durch Linien und Pfeile realisiert. Entsprechende Diagramme können bis auf einzelne Komponenten oder elektronische Schaltungen hinunter gebrochen werden. Blockdefinitionsdiagramme bestehen auch aus Operationen, welche Verhalten von Blöcken beschreiben. Verhaltensdefinitionen bei SysML sind wie bei UML modelliert.

Modellierungen der Schaltung des Schwingkreiswechselrichters beschreiben die Funktionaltäten der abschaltbaren Leistungshalbleiter wie z. B. Transistoren (IGBT) bezüglich der Resonanzelemente. Hierbei besteht die Schaltung des Schwingkreiswechselrichters aus Gleichspannung, Stützkondensator, IGBT mit Dioden und Lastkreis mit Spule, Widerstand und Kondensator. Die Modellierung der Schaltung ist mithilfe des Frameworks Eclipse-Papyrus realisiert worden. Diese Schaltung stellt diese eines Resonanzstromrichters mithilfe des Lastkreises und der Verwendung der abschaltbaren Leistungshalbleiter wie z. B. IGBT dar. Ziel der Modellierung der Schaltung des Schwingkreiswechselrichters ist es, den Auftritt der geringen Schaltverluste bei hohen Betriebsfrequenz zu ermöglichen. Der Editor von Eclipse-Papyrus verfügt über die Modellierungswerkzeuge, genannt Palette, u. a. *„Associations“*, *„ModelElements“*, *„DataTypes“* oder *„PortsAndFlows“*. Mithilfe der objektorientierten Modellierung stellen Modelle eine Instanziierung von Objekten anhand von Typdefinitionen dar. Typdefinitionen enthalten Attribute sowie Verhaltensbeschreibungen. Es gibt Wertetyp- und Block-Definitionen. Typdefinition eines Blocks stellt eine Verbindung mit Verhalten dar. Werte von Modellbeschreibungsgrößen sind veränderbar, weil Blöcke Stereotype der UML-Metaklasse *„Class“* sind und *„Class“* von *„BehavioredClassifier“* erbt.

Blöcke definieren die modulare Struktureinheit innerhalb der SysML (Systems Modeling Language) und stellen statische Konzepte und Gegenstände des zugrunde liegenden Systems dar. Im Bereich Softwareentwicklung beschreibt ein Block z. B. ein Datenelement, einen Operator oder ein Kontrollflusselement. Ein Block in einem Blockdiagramm beschreibt eine Menge an eindeutig identifizierbaren Eigenschaften, deren Gemeinsamkeit die Definition des jeweiligen Blocks ist. Blöcke beschreiben ein System als eine Ansammlung von Teilen, die im spezifischen Kontext eine bestimmte Rolle spielen.

Interne Blockdiagramme werden zur Darstellung des inneren eines Systems im Bereich System-Engineering eingesetzt. In *internen Blockdiagrammen (IBD) werden* die Struktur und die Flüsse innerhalb eines Systembausteins (Block) mit den Mitteln der OMG SysML™ *(Systems Modeling Language)* beschrieben. Interne Blockdiagramme bieten eine einfache Übersicht darüber, wie die Teile (Parts) eines Blocks interagieren und welche Art von Daten, Informationen, Signalen oder Materialien in welche Richtungen fließen. Dabei lassen sich auch beliebige Schachtelungstiefen darstellen. IBD verfügen über sowohl Elemente als auch Beziehungen zur Darstellung der Modellierungen. Zum einen verfügen die IBD verschiedene Elemente u. a. *„Block“*, *„Part“*, *„Interface“*, *„Referenz“* oder *„Port“* und zum anderen enthalten IBD mehrere Beziehungen wie z. B. *„Verbindender Konnektor“*, *„Konnektor“* oder *„Abhängigkeit“*.

Interne Blockdiagramme von dem Framework Eclipse-Papyrus für SysML-1.4 verfügen über Modellierungswerkzeuge, genannt Palette, die aus drei Bereichen bestehen: *„General Annotation“*, *„Blocks“* und *„Ports and Flow“*. Der erste Bereich beinhaltet verschiedene Elemente wie z. B. *„Realization“*, *„Dependency“*, *„Constraint“*, *„Abstraction“* oder *„Context Link“*. Der zweite Bereich besteht aus Elementen wie z. B. *„ActorPart“*, *„Part“*, *„Binding Connector“*, *„Connector“*, *„BoundReference“*,

„Reference“, *„Dependency“*. Der dritte Bereich setzt sich aus Elementen wie z. B. *„Port“*, *„FlowPort“*, *„Item Flow“*, *„ProxyPort“* oder *„Full Port“* zusammen.

Die Schaltung eines Blindleistungsstromrichters dient der Kompensation sowohl von induktiver als auch von kapazitiver Blindleistung. Das Besondere in der Schaltung ist, dass auf der Gleichspannungsseite keine besondere Spannungsquelle, sondern ein Kondensator vorgesehen ist. Hierbei ermöglicht die Modellierung der Schaltung eines Blindstromrichters mit den internen Blockdiagrammen sowohl eine Energieentnahme aus dem Netz als auch eine Energieeinspeisung in das Netz.

Informationsobjektflüssen stellen die Modellierung eines Kanals zwischen zwei Elementen des Modells dar, über den die beiden Elemente Informationseinheiten, genannt *„Item Flow“*, austauschen. Informationseinheiten werden eingesetzt, wenn der strukturierte Austausch von Informationen zwischen Elementen eines Systems modelliert werden soll, ohne dass das Modell bereits die Struktur- und Repräsentationsdetails der Information festlegt.

Der SysML-Objektflussport, auch *„flow port“* genannt, ist der Stereotyp des UML-Ports. Objektflussports beschreiben Interaktionspunkte eines Systembausteins mit seiner Umgebung, über welche die Objekte in oder aus dem Baustein fließen können. Ein Objektflussport ist Teil des Systembausteins. Er besitzt einen klein geschrieben Namen, einen Typ sowie eine Multiplizität und wird als kleines Quadrat am Systembaustein dargestellt. Mittels der Objektflussspezifikation, genannt *flow specification,* die simultan der UML-Schnittstelle ist, wird die Kommunikationsmethode des Objektflussports erläutert, welche die ausgetauschten Daten beschreibt. Der Informationsobjektfluss, auch *item flow* genannt, stellt ursprünglich aus dem UML-Informationsfluss einen speziellen Informationsfluss dar, der im internen Blockdiagramm an einem Konnektor beschreibt, dass konkrete Objekte transportiert werden. Der Objektflussport beschreibt die fließenden Objekte.

Anforderungsdiagramm, genannt *„Requirement Diagram“*, mit der Abkürzung REQ stellt ein benutzerdefiniertes Diagramm der nichtfunktionellen Anforderungen eines Systems mithilfe eines visuellen Modells dar.

Die Umgebung Eclipse-Papyrus unterstützt die SysML-Modellierung für eingebettete Systeme und andere komplexe Geräte.

In einem eingebetteten System werden sowohl Input erhält als auch Output erzeugt. Außerdem sind interne Operationen und Zustände des Systems nicht durchsichtig. Dies führt zu Problemen bei der Steuerung eingebetteter Systeme während des Entwickelns, Testens und Debuggens von Software. Bei der Modellierung von Software mit SysML-Papyrus werden rückverfolgbare Vorlagen erstellt, mithilfe von Anforderungsdiagrammen des Frameworks Eclipse-Papyrus werden das Planen, Entwerfen und Dokumentieren des Systems ermöglicht.

Der Bedarf vom Markt an Entwicklungen von komplexeren Systemen ist enorm, aber der hohe Anspruch an GUI und Qualität ist konstant, stellt für viele Projekte eine zunehmende Herausforderung für die Beschreibungen der Anforderungen der Systeme dar. Anforderungsmodellierungen des Systems mit SysML bezüglich des Einsatzes des

Anforderungsdiagramms, genannt *Requirement Diagram,* ermöglicht das Annehmen dieser Herausforderung. Das Anforderungsdiagramm stellt den Diagrammtyp zur Beschreibung von Funktion, Leistung und Schnittstelle mit dem Ziel das Ersetzen des „Use Case Diagrams“ von UML zu ermöglichen.

Schwingkreiswechselrichter stellen Resonanzstromrichter bezüglich der Anwesenheit der Resonanzelemente des Lastkreises in der Schaltung dar. Hierbei besteht der Lastkreis aus einem Schwingkreis, welches sowohl ein induktives als auch ein kapazitives Verhalten mit Drosselspule bzw. Kondensator zeigt. Anforderungen der Schaltungen mit dem Schwingkreiswechselrichter fokussieren zum einen auf Reduzieren der Verluste dank sowohl der induktiven als auch kapazitiven Verhalten der Last. Zum anderen fokussieren sie auf den Erhalt eines sinusförmigen Verlaufs des Laststroms oder der Lastspannung. Außerdem zeigt es sich, dass, bei hoher Frequenz, ein Betrieb mit induktivem Lastverhalten oder mit einer Frequenz, die sich sehr nahe bei der Resonanzfrequenz liegt, Vorteile hat.

Zusicherungsdiagramme stellen den Zusammenhang der Eigenschaften der Blöcke bezüglich der Formulierungen der Constraints mithilfe der Werte der Eigenschaften dar. Systeme verfügen über Systemelemente, deren veränderbare Werte sich gegeneinander beeinflussen. Mithilfe von Zusicherungsdiagrammen werden parametrischen Diagramme zwischen Eigenschaften von Systembausteinen ermöglicht. Ziel des Zusicherungsdiagramms ist es, Relationen von einzelnen Systembausteinen mithilfe der Formel darzustellen. Zusicherungsdiagramme sind Strukturdiagramme, die Formulierungen von Constraints zum Verbinden von Parameter mithilfe der Blöcke darstellen. Contraints sind mithilfe von geschweiften Klammern gekennzeichnet.

Mithilfe von Eclipse-Papyrus wurden die Zusicherungsdiagramme erstellt. Der Editor zum Erstellen von SysML-Zusicherungsdiagrammen verfügt über Modellierungswerkzeuge, genannt Palette, welche aus vier Teilen bestehen: „General Annotations“, *„Blocks“*, *„Ports and Flow“* und *„Constraints“*. *„General Annotations“* enthalten Elemente, u. a. Constraints, Abstraction, Comment, Context Link, Dependency, Link, Rationale, Problem oder Refine während Elemente wie z. B. Part, ActorPart, Binding Connector, Connector, BoundReference oder Dependency, die zu *„Blocks“* gehören. Anschließend verfügt „Port and Flow“ über Elemente wie z. B. FlowPort, Full Port, Port, Item Port und Proxy Port. Der vierte Teil, genannt *„Constraints“*, verfügt über ein einziges Element: Constraints.

Literatur

1. OMG Web Portal: Tutorial for Model-Based Systems Engineering (MBSE), In: What ist SysML, http://www.omgsysml.org, Object management group (2019).
2. Hausding, P.: Systemmodellierung mit SysML, Studienarbeit, Institut für Informatik Lehr- und Forschungseinheit Systemanalyse, Humboldt-Uni zu Berlin, https://www.informatik.hu-berlin.de/de/forschung/gebiete/sam/Lehre/proseminar-sysml/material/studienarbeit-systemmodellierung-mit-sysml-von-peer-hausding (2010).

3. Igler, B., Krümmel, N., Ried, M., Renz, B.: Darstellungen von Konzepten für die Softwareentwicklung, Institut für Software Architektur, Technische Hochschule Mittelhessen, Campus Hessen, https://wiki.thm.de/ (2012).
4. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML:The System Modeling Language, 1. Auflage, Verlag Morgan Kaufmann, ISBN 10: 012378607X, ISBN 13: 9780123786074, USA (2009).
5. objectiF Web Portal: Diagrammtype für den Systementwurf, In: Design und Implementierung, objectiF Software, https://www.microtool.de/was-ist-ein-blockdiagramm, microTOOL, Berlin (2019).
6. Hagmann, G.: Leistungselektronik: Gundlagen und Anwendungen, AULA-Verlag, Wiesbaden, ISBN 10: 389104626X ISBN 13: 9783891046265 (1998).
7. Weilkiens, T.: Systems Engineering mit SysML/UML: Anforderungen, Analyse, Architektur, dpunkt. Verlag GmbH, ISBN-10: 3864900913, ISBN-13: 978-3864900913 (2014).

Parallele Modellierung mit Obeo-UML-Designer

5

Die *Unified Modeling Language (UML)* ermöglicht die Modellierung, Visualisierung, Dokumentation, Verbesserung, Simulation komplexer Softwaresysteme. Grafische Modellierungen von parallelen Systemen wie z. B. Asynchronmaschine werden mithilfe von UML-Diagrammen realisiert. Multi-View-Sicht von UML ermöglicht die Darstellung unterschiedlicher Diagrammtypen wie z. B. Zustandsdiagramm, Sequenzdiagramm oder Aktivitätsdiagramm. UML stellt die Modellierung nebenläufiger Systeme mithilfe von Diagrammen wie z. B. Sequenz-, Aktivitäts- und Zustandsdiagrammen dar.

5.1 Modellierung mit Zustandsdiagrammen

Zustandsdiagramme stellen die kontrollierten Zustände von Zustandsautomaten wie z. B. Objekten oder Systemen während bestimmter Laufzeit dar, wobei sie Ereignisse zum Auslösen der Zustandsübergänge angeben. Hierbei ermöglichen Zustandsdiagramme die Synchronisation der Funktionen der Systeme mithilfe der Zustände. Das Open-Source-Framework Eclipse-UML-Designer von der Firma Obeo Designer ermöglicht die parallele Modellierung mit Zustandsdiagrammen. Der Editor von UML-Designer verfügt über mehrere UML-Diagramme u. a. Klassen-, Aktivitäts-, Sequenz-, oder Zustandsdiagramme zum Modellieren von Softwaresystemen. Das Zustandsdiagramm verfügt über Modellierungstools, genannt Palette, welche in zwei Teile eingegliedert ist: *„Types“* und *„Relationships“*, wie es auf der Abb. 5.1 zu sehen ist.

Gemäß Abb. 5.1 werden die Zustände in einem Zustandsdiagramm von Eclipse-UML-Designer durch Rechtecke mit abgerundeten Ecken dargestellt. Mithilfe der Pfeile zwischen den Zuständen werden Zustandsübergänge dargestellt. Sie sind mit den Ereignissen zum Durchführen des jeweiligen Zustandsüberganges beschriftet [1].

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML*,
https://doi.org/10.1007/978-3-658-29057-3_5

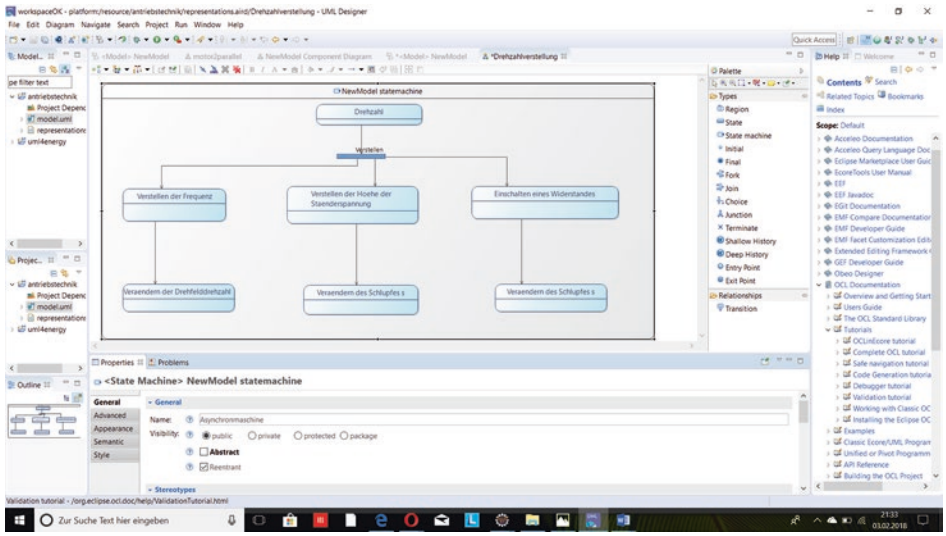


Abb. 5.1 Modellierung der Drehzahlverstellung des Asynchronmotors

Gemäß Abb. 5.1 verfügt die Palette des Zustandsdiagramms über Werkzeuge einerseits zum Synchronisieren mit *„join node“* und andererseits zum Parallelisieren mit *„fork node“*.

Anhand der Abb. 5.1 werden die Möglichkeiten zur Verstellung der Drehzahl des Asynchronmotors mithilfe der Folgen von Zuständen mit dem Zustandsdiagramm von Eclipse-UMlL-Designer beschrieben.

Abb. 5.1 zeigt zum einen die Modellierung der Zustände mithilfe von Transitionen und deren Ereignisse zur kontinuierlichen Verstellung der Drehzahl und zum anderen das Bestimmen des Wirkungsgrades des Asynchronmotors.

5.1.1 Horizontale Modellierung

Modellierung horizontaler Schichte stellt die Modellierung von Zuständen auf einer Linie oder einer Ebene dar. Hierbei werden Aktivitäten der Zustände berücksichtigt. Abb. 5.1 zeigt die verschiedenen Schichten und deren Zustände, die auf einer horizontalen Schicht positioniert sind. Gemäß Abb. 5.1 verfügt das Zustandsdiagramm über 4 Schichten, wobei jede Schicht eine horizontale Ebene darstellt. Zwischen Start- und Endzustand werden die Zustände von der ersten Schichte bis zu letzten Schichten positioniert. Gemäß Abb. 5.1 enthält die erste Schichte sowohl den Startzustand als auch den Zustand genannt *„Drehzahl n“*. Anschließend wird die Transition *„Verstellen“* mithilfe von Parallelisierungsknoten, genannt „fork node“, den Übergang zur zweiten Schichte darstellen. Die zweite Schichte enthält drei Zustände: Verstellen der Frequenz f, Verstellen der Höhe der Ständerspannung U und Einschalten eines Widerstandes R.

Die dritte Schicht wird mithilfe einer Transition erstellt, wobei jeder Zustand der zweiten Schichte in Verbindung mit dem entsprechenden Zustand der dritten Schicht ist. Eine Transition liegt zwischen einer Quelle wie z. B. *„Verstellen der Frequenz f“* und einem Ziel oder Empfänger wie z. B. *„Veraendern der Drehfelddrehzahl nd“*. Der dritte Schicht besteht aus drei Zuständen: *„Veraendern der Drehfelddrehzahl n_d“*, *„Veraendern des Schlupfes s“* und *„Veraendern des Schlupfes s“*. Mithilfe der letzten Schicht oder der vierten Schicht wird die Drehzahlungssteuerung zum Bestimmen des Wirkungsgrades des Motors ermöglicht. Zwischen der dritten Schicht und der vierten Schichte liegen sowohl Parallelisierungsknoten, genannt „Fork Node“, als auch Synchronisierungsknoten, genannt *„Join Node“*. Die vierte Schichte stellt mit der ersten Schichte eine Synchronisation einerseits zwischen dem Parallelisierungsknoten von Zuständen *„Drehzahl n“* und *„Veraendern der Drehfelddrehzahl n_d“* und andererseits zwischen Zuständen *„Veraendern des Schlupfes s“* zum Bestimmen des Wirkungsgrades dar. Nebenläufigkeitsmodellierungen werden zwischen dem Zustand der ersten Schichte und den Zuständen der vierten Schichte zur Drehzahlsteuerung des Asynchronmotors realisiert. Das Bestimmen des Wirkungsgrades wird mithilfe der beiden Formel aus den Parallelisierungsprozessen und Synchronisierungsprozessen modelliert. Zum einen stellt der Wirkungsgrad ein Verhältnis von der Drehzahl n und von der Drehfelddrehzahl n_d dar und zum anderen stellt er eine Differenz zwischen der Zahl 1 und dem Motorschlupf s dar. Beide folgende Formel sind auf der Abb. 5.1 zum Bestimmen des Wirkungsgrades dargestellt.

$$\eta = n/n_d \text{ mit n: Drehzahl, } n_d = \text{Drehfelddrehzahl} \tag{5.1}$$

$$\eta = 1 - s \text{ mit s = Schlupf des Motors} \tag{5.2}$$

5.1.2 Vertikale Modellierung

Die Informationsmodellierung stellt die Strukturierung der Information hinsichtlich der Integration der Modelle dar. Hierbei werden die Informationen nach Orientierungen modelliert. Mithilfe der UML-Diagramme wie z. B. Zustand-, Sequenz-, Aktivitäts- oder Klassendiagramme werden sich die Informationen der Modelle nach geometrischen Orientierungen bewegen, wobei es möglich ist, dass die Informationen sich entweder vertikal oder horizontal bewegen. Sowohl Synchronisierungen als auch Parallelisierung der Informationen ermöglichen die dynamischen Modellierungen der Objekte, wobei die Prozesse der Parallelisierung oder der Synchronisierung die Struktur der Informationen darstellen. Mithilfe des Zustandsdiagramms wie dieses der Abb. 5.1 werden die Modellierungen der Informationen bezüglich der Drehzahlsteuerung in einer vertikalen Orientierung realisiert. Vom Start aus bis zum Schluss der Modellierung zeigt die Abb. 5.1 einen Nebenläufigkeitsprozess mit einer vertikalen Integration. D. h., zwischen Start und Ende der Modellierung liegen drei vertikale und parallele Prozesse zum Bestimmen des Wirkungsgrades des Asynchronmotors. Die erste vertikale Integration enthält die Zustände

„Drehzahl n“, *„Verstellen der Frequenz f“* und *„Veraendern der Drehfelddrehzahl n_d“*. Diese Integration führt zum Bestimmen des Wirkungsgrades mit der Formel 5.1. Die erste vertikale Integration besteht aus zwei Transitionen, welche Relationen zwischen Zuständen darstellen. Die erste vertikale Integration bedeutet, dass das Verstellen der Frequenz f zum Verstellen der Drehfelddrehzahl n_d führt. D. h., das Verstellen der Drehzahl des Asynchronmotors ist mithilfe des Veränderns der Drehfelddrehzahl durch Verstellen der Frequenz der Ständerspannung möglich. Weil die Ständerspannung zu Ständerfrequenz proportional ist, gilt die folgende Formel zum Verstellen der Ständerspannung:

$$\frac{U_s}{U_N} = f_S / f_N \tag{5.3}$$

Mit f_S: Ständerfrequenz, f_N: Nennfrequenz, U_n: Nennspannung, U_S: Ständerspannung

$$f_R = s * fS = \left(1 - \frac{n}{n_d}\right) \tag{5.4}$$

Mit f_R: Rotorfrequenz, s: Motorschlupf, n: Läuferdrehzahl, n_d: Drehfelddrehzahl

Nachdem die Drehfelddrehzahl verstellt wird, ist das Bestimmen des Wirkungsgrades mithilfe des Verstellens der Läuferdrehzahl n ermöglicht, wie es auf der Abb. 5.3 zu sehen ist. Die zweite vertikale Integration stellt einen parallelen Prozess zu der ersten Integration dar. Ziel des zweiten Prozesses zum Bestimmen des Wirkungsgrades ist es, die Verstellung der Drehzahl n mit dem Verändern des Schlipfes des Motors durch Verstellen der Höhe der Ständerspannung U_S zu ermöglichen. Die zweite vertikale Integration ist zu der ersten parallel, weil es um das Bestimmen des Wirkungsrades nach den Formel 5.3 und 5.4 mithilfe des Verstellens der Drehzahl n des Asynchronmotors geht. Mithilfe von Tab. 5.1 wird die Modellierung der Drehzahlverstellung bezüglich der vertikalen Integration verständlich. Hierbei ist es zu bemerken, dass jede Spalte eine vertikale Modellierung darstellt. Ziel ist es, das Verstellen der Motordrehzahl mithilfe der drei Spalten zu beschreiben. Tab. 5.1 ergänzt die Beschreibung der Abb. 5.1 bezüglich der vertikalen Integration. Bezüglich zum einen der Drehzahlverstellung n und zum anderen der Bestimmung des Wirkungsgrades η sind die drei Spalten zueinander parallel.

Tab. 5.1 Vertikale Integration zur Modellierung der Drehzahlverstellung des Asynchronmotors

Drehzahl n		
Verstellen der Frequenz f	Verstellen der Höhe der Staenderspannung U	Einschalten eines Widerstandes R
Veraendern der Drehfelddrehzahl n_d	Veraendern des Schlupfes s	Veraendern des Schlupfes s
Wirkungsgrad η		

Vertikale Modellierung der Drehzahlverstellung

5.2 Modellierung mit Aktivitätsdiagrammen

Aktivitätsdiagramme stellen die Modellierungen zum einen der Vorgänge, genannt Workflow, und zum anderen der Prozesse zum Aneinanderreihen sowohl der Aktivitäten als auch der Funktionen dar. Ziel der Anwendung der Aktivitätsdiagramme ist es, die Anwendungsfalldiagramme, genannt Use-Case-Diagramme, zu ergänzen und beschreiben. Modellierung mit Aktivitätsdiagrammen ermöglicht das Beschreiben des Verhaltens des Systems mithilfe von Prozessen oder Workflows.

Dieser Abschnitt setzt das Aktivitätsdiagramm zum Beschreiben von Nebenläufigkeitsprozessen bezüglich sowohl der parallelen Modellierung als auch der Synchronisierungsmodellierung ein. Mithilfe der Abstraktion wird das Aktivitätsdiagramm Prozesse oder Workflows beschrieben.

Der Modellierungskern mit dem Aktivitätsdiagram ist das Beschreiben der Aktivitäten, genannt *„Activity“*, die über Aktionen zum Beschreiben der Prozesse verfügen. Aktionen stellen wesentliche Vorgänge zur Modellierung des Systems. Das Aktivitätsdiagramm vom Open Source UML-Designer verfügt über Modellierungstools, genannt „activity tools“, die aus Elementen wie z. B. *„Partition“*, *„Nodes“*, *„Actions“*, *„Activity Parameter“* oder *„Flows“* bestehen.

Die Modellierung mit dem Aktivitätsdiagramm von UML-Designer ermöglicht die Beschreibungen der IT-Lösungen für den Drehstromantrieb. Die Asynchronmaschine ist der meist benutzte Maschinentyp und deren Einsatz sowohl in Geräten des Haushaltes als auch in der Industrie die Erklärung der Anwendungen in Energietechnik darstellt [2].

Abb. 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 und 5.12 zeigen das Erstellen vom Aktivitätsdiagramm mit UML-Designer bezüglich der Bestimmung des Wirkungsgrades des Asynchronmotors. Das Erstellen des Projektes mit dem Namen *„uml4energy“* zeigt Abb. 5.2. Anschließend werden die Schritte zum Erstellen des Diagramms mit dem Editor von UML-Designer, wie es auf den Abb. 5.3, 5.4, 5.5, 5.6 und 5.7 zu sehen ist, dargestellt. Mit der Abb. 5.3 wird das UML-Diagramm mithilfe von *„model uml“* von dem Project Explorer initialisiert. Abb. 5.4 und 5.5 stellen die Abgangsmappe vom Modell mit verschiedenen Wizards, u. a. Design, Extend, Review, Capture oder Overview, dar. Abb. 5.6 zeigt den Inhalt der Mappe *„Design“* zum Erstellen von UML-Diagrammen mit UML-Designer, u. a. „Class diagram“, *„Activity diagram“* oder *„Component diagram“*. Abb. 5.7 und 5.8 zeigen das Erstellen des Aktivitätsdiagramms, dessen Modelle den Namen *„motor2paralle“* der Funktionalität des Systems *„Asynchronmotors“* darstellt. Abb. 5.9 stellt einen Parallelisierungsprozess zum Bestimmen des Wirkungsgrades einerseits von Drehzahl über Schlupf bis zum Wirkungsgrad und andererseits von Drehmoment über Leistung bis zum Wirkungsgrad dar. Abb. 5.10 und 5.11 zeigen die Funktionalitäten des Aktivitätsdiagramms eines Asynchronmotors zum Bestimmen der Effizienz des Motors mit dem Ziel einen effizienten Motor zu charakterisieren. Abb. 5.11 stellt die Prozesse zum Bestimmen der Wirkungsgrades des Asynchronmotors dar [2].

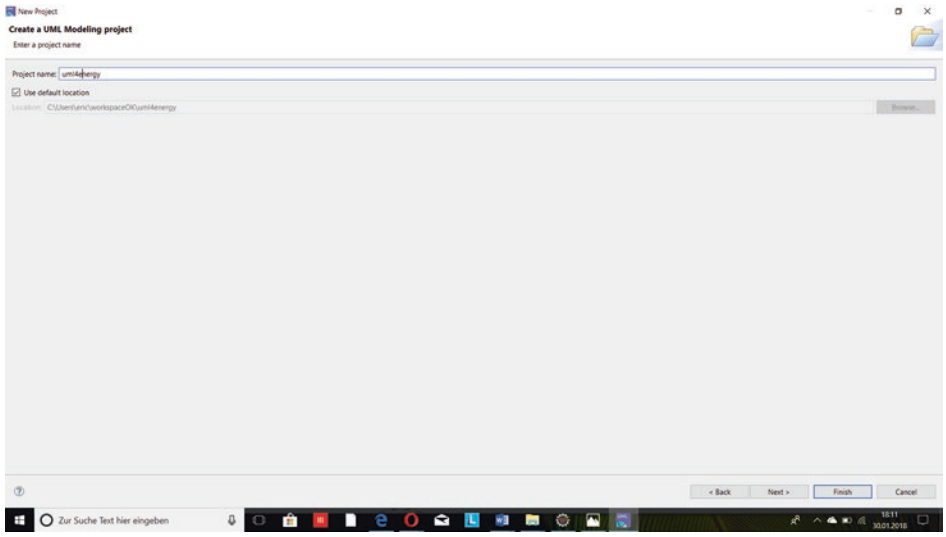


Abb. 5.2 Das Erstellen des UML-Modellierungsprojektes *„uml4energy“* mit UML-Designer

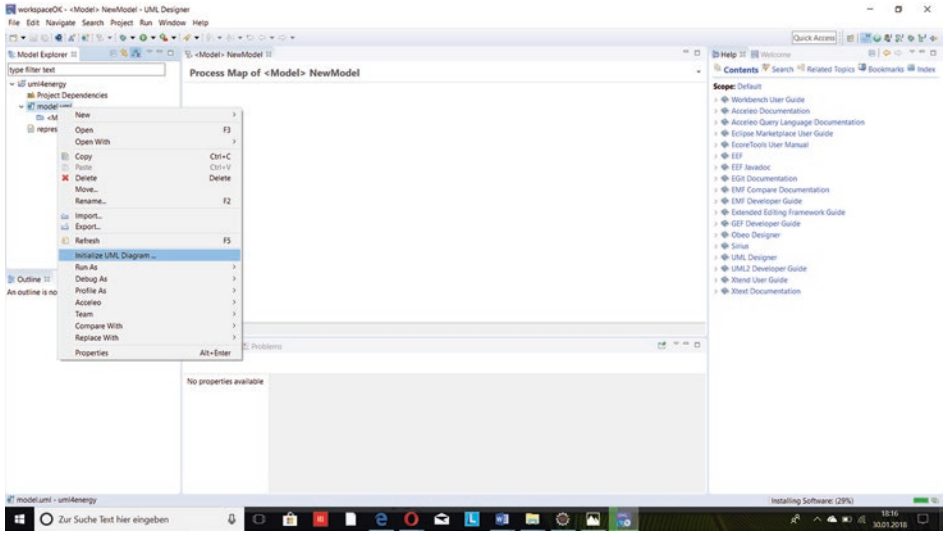


Abb. 5.3 Das Initialisieren des UML-Diagramms mithilfe von *„uml model“* aus dem *„Project Explorer“*

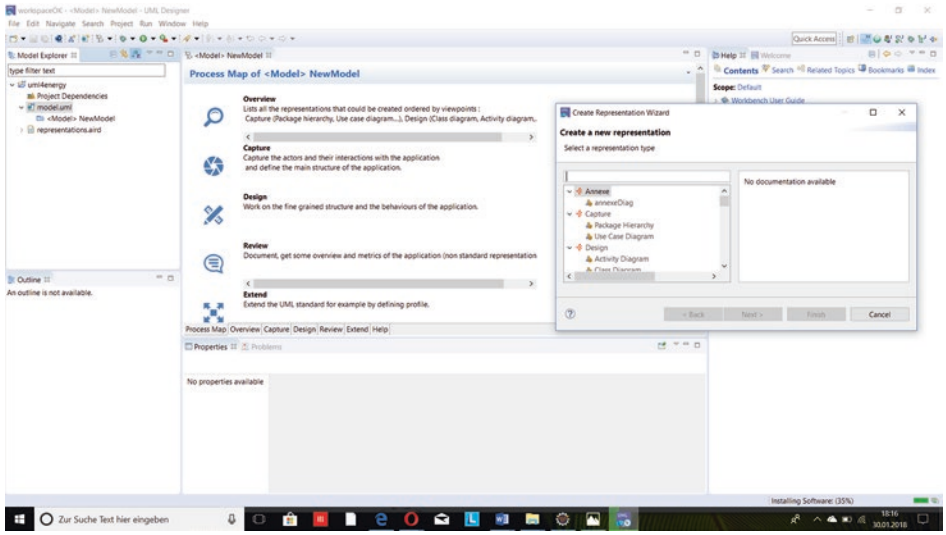


Abb. 5.4 Überblick über verschiedene Wizards, u. a. *„Design“*, *„Overview“*, oder *„Capture“* vom *„Process Map of Model“*

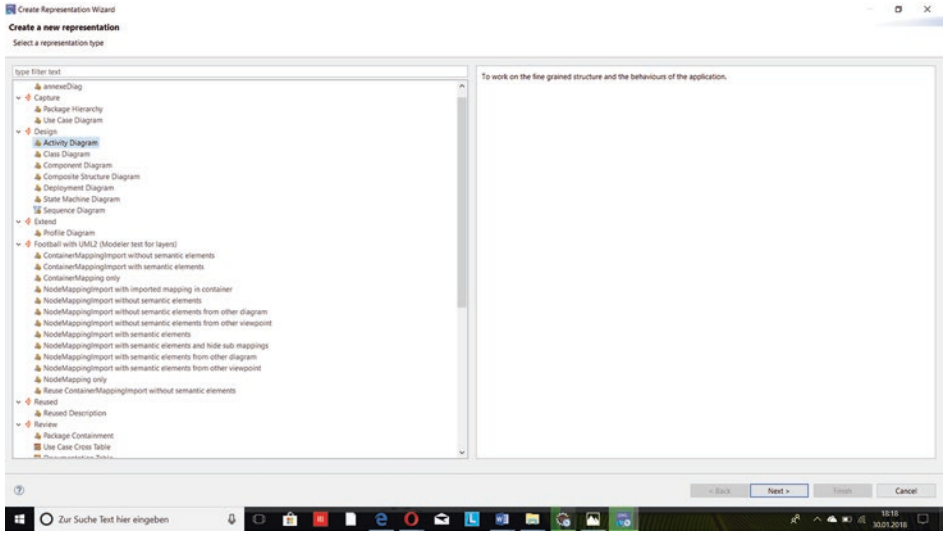


Abb. 5.5 Das Auswählen des Diagrammtyps, genannt *„Activity Diagram“*, im Wizard *„Design“*

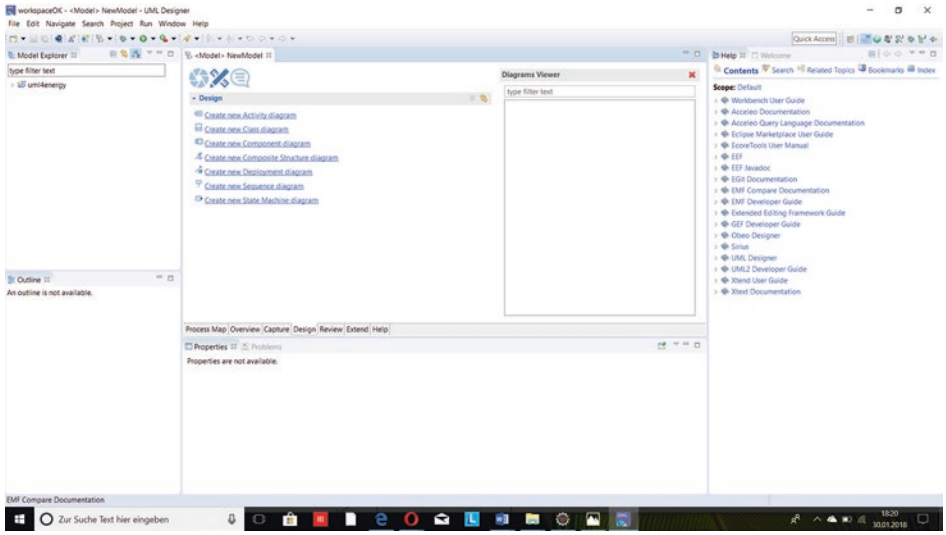


Abb. 5.6 Darstellung von UML-Diagrammen aus der Mappe *„Design“*, u. a. *„Class diagram“* *„Activity diagram“* oder *„Component diagram“*

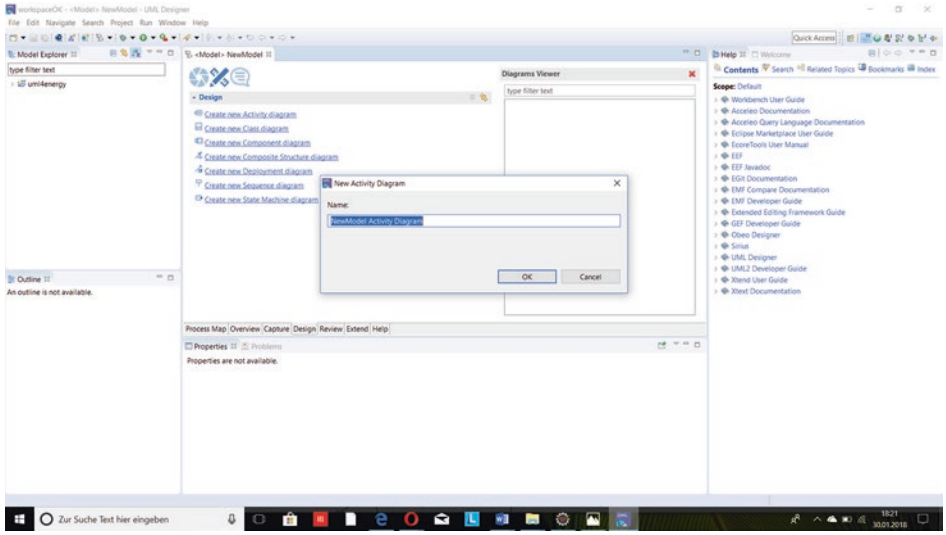


Abb. 5.7 Das Erstellen des Aktivitätsdiagramms

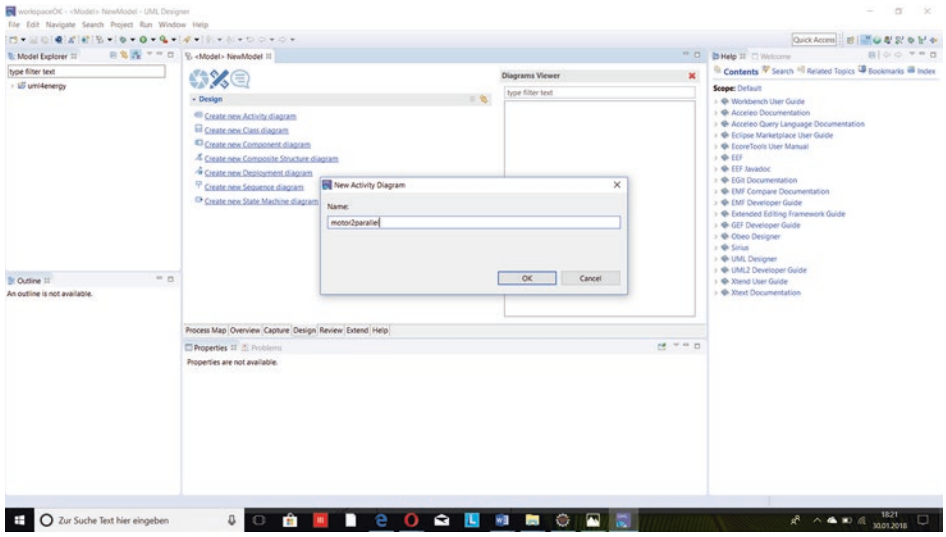


Abb. 5.8 Aktivitätsdiagramm *„motor2parallel“*

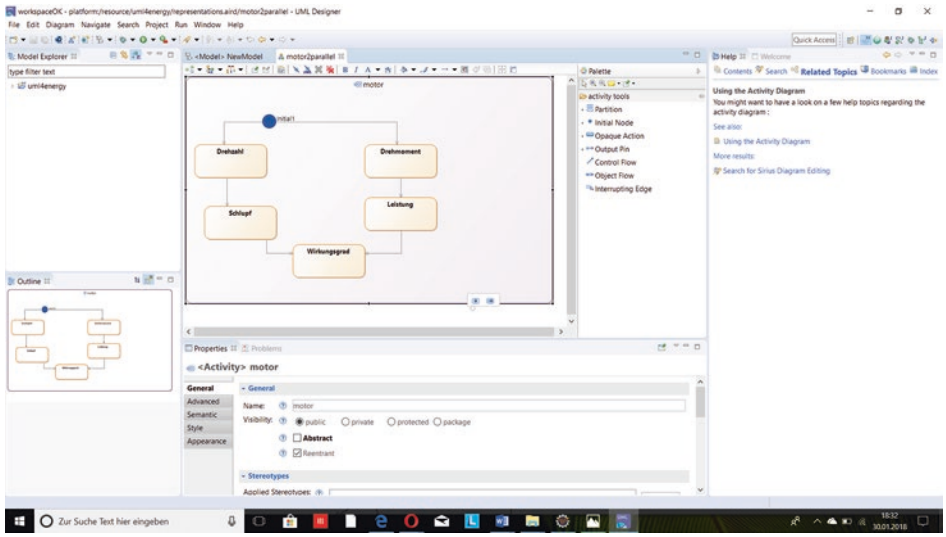


Abb. 5.9 Überblick über parallele Modellierung mithilfe von *„Control Flow“*

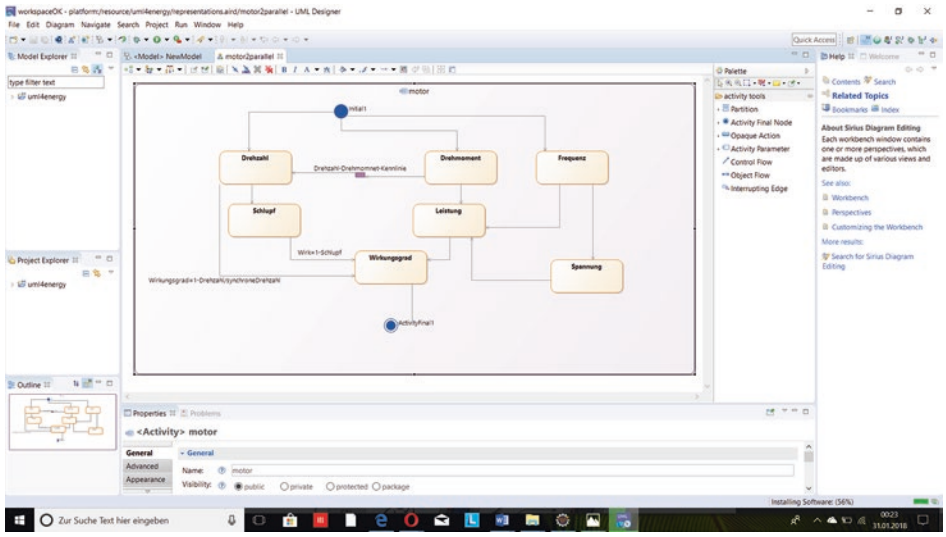


Abb. 5.10 Überblick über drei Parallelisierungsprozesse mithilfe von *„Control Flow“* zum Bestimmen des Wirkungsgrades

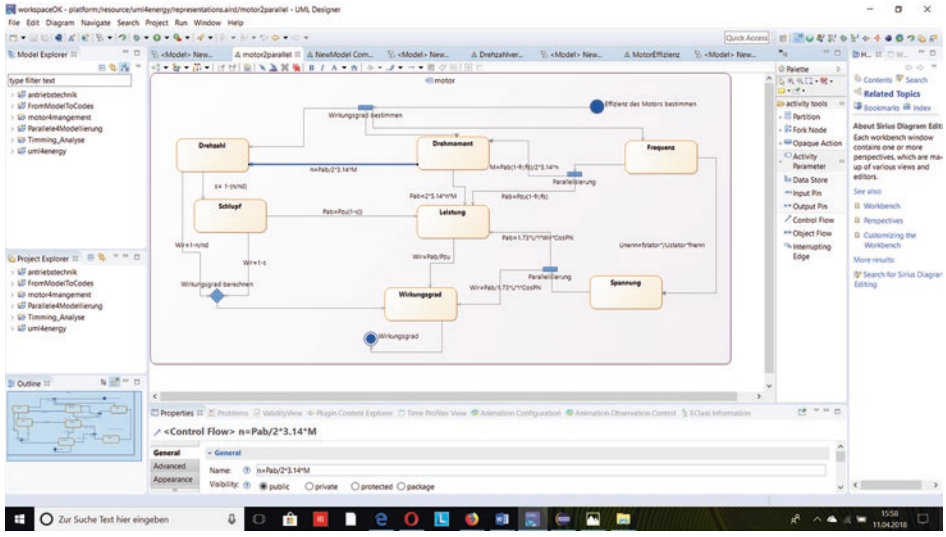


Abb. 5.11 Darstellung der Parallelisierungsprozesse mithilfe von *„Fork Node“* und *„Decision Node“* zum Bestimmen des Wirkungsgrades des Motors

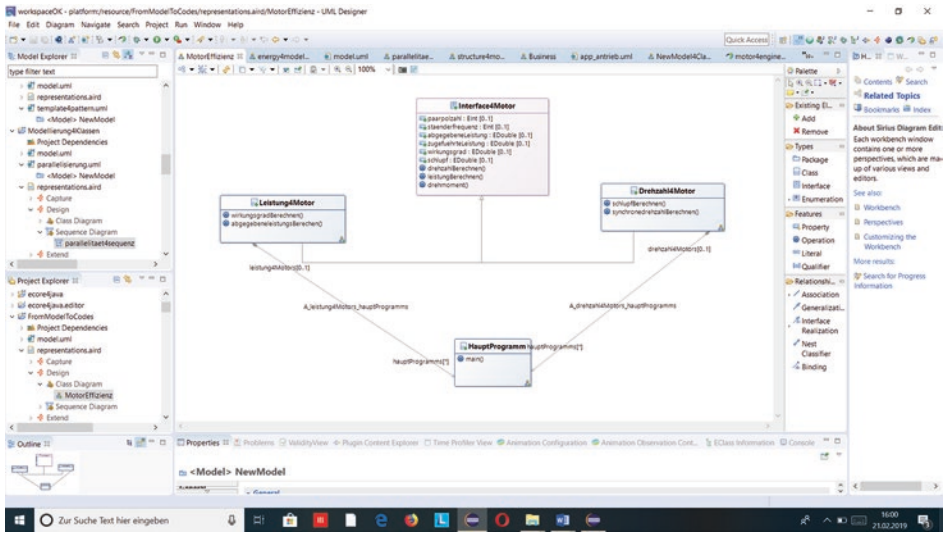


Abb. 5.12 Vererbungshierarchie bezüglich der Parallelisierungsprozesse zum Charakterisieren des Asynchronmotors

Abb. 5.11 zeigt das Aktivitätsdiagramm zum Darstellen der Effizienz des Asynchronmotors bezüglich der Bestimmung des Wirkungsgrades mithilfe von verschiedenen Kenngrößen, u. a. Drehzahl, Leistung, Schlupf, Spannung oder Frequenz. Zwischen Startknoten, genannt *„Initial Node“*, und Endknoten, genannt *„Activity Final Node“*, befinden sich Aktionen als Rechteck mit abgerundeten Ecken dargestellt. Diese Knoten, genannt *„Nodes“*, stellen einerseits Kenngrößen zum Modellieren der Bestimmung des Wirkungsgrades des Asynchronmotors dar und andererseits die Workflows der Aktivität des Diagramms, genannt *„motor“*. Im Aktivitätsdiagramm der Abb. 5.11 befinden sich acht Aktionen, d. h. Kenngrößen sowohl in der horizontalen als auch in der vertikalen Position. Diese Aktionen sind mithilfe der Pfeile miteinander verbunden, wobei auf jedem Pfeil eine Gleichung oder Funktion zum Bestimmen der entsprechenden Kenngröße steht. Abb. 5.11 zeigt die Verwendung sowohl von Splitting-Knoten, genannt *„Fork Node“*, als auch Entscheidungsknoten, genannt *„Decision Node“*, zum Parallelisieren der Prozessflüsse bzw. zum Zusammenführen der Prozessflüsse. Mit dem Zusammenführen eines Prozesses wird eine Alternative Option zu dem ersten Prozess gewählt. Dies ermöglicht die Modellierungen der Nebenläufigkeitsprozesse auf sowohl horizontalem als auch vertikalen Ebene. Gemäß Abb. 5.11 fasst die Aktivität *„motor“* die sieben Aktionen oder Kenngrößen und deren Steuerungsflüss, genannt *„Control Flows“*, zusammen. Der Startknoten oder Initial Node der Aktivität *„motor“* ist nach *„Effizienz ders Motors bestimmen“* genannt und wird durch einen ausgefüllten Kreis dargestellt während der Finalknoten oder *"Activity Final Node"* genannt *„Wirkungsgrad“* durch einen Kreis mit einem kleineren und ausgefüllten Kreis.

5.2.1 Vertikale Integration

Modellierungen des Verhaltens eines Systems wie eines Asynchronmotors mithilfe des Aktivitätsdiagramms ermöglicht das Analysieren des Diagramms zum Beschreiben der Funktionalität des Systems mit einerseits den Nebenläufigkeitsprozessen und andererseits den Abläufen der Aktionen durch die Steuerungsflüsse. Die Aktivität verfügt über mehre Aktionen, deren Steuerungsflüsse Aktionen verbinden. Die Richtungen der Steuerungsflüsse stellen die geometrische Modellierung des Diagramms entweder vertikal oder horizontal dar.

Vertikale Modellierung mit dem Aktivitätsdiagramm ermöglicht das Analysieren der Modellierung in die vertikale Position, wobei die Aktionen mithilfe der Pfeile der vertikalen Richtung folgen werden. Hierbei werden die Aktionen und deren Control Flow in eine Spalte integriert. Die vertikale Integration mit dem Aktivitätsdiagramm stellt eine geometrische Richtung der Modellierung mithilfe der Spalten dar. Die Spalten orientiert sich automatisch vertikal und von oben nach unten, wobei Startknoten und Endknoten die Eingliederungen der Spalten in einer geometrischen Integration d. h. vertikalen Integration ermöglichen.

Gemäß Abb. 5.10 und 5.11 verfügt die Aktivität „motor“ drei Spalten oder vertikale Integrationen aus Aktionen oder Kenngrößen. Zum Beispiel enthält die erste Spalte (erste vertikale Integration) die Aktionen *„Drehzahl“* und *„Schlupf“* Anschließend gehören die Aktionen *„Drehmoment“*, *„Leistung“* und *„Wirkungsgrad“* zu der zweiten Spalte(zweite vertikale Integration). Am Schluss verfügt die dritte Spalte (dritte vertikale Integration) die Aktionen *„Frequenz“* und *„Spannung“*. Die drei Spalten können in eine Tabelle integriert werden, wobei die Aktionen vertikale und horizontal dargestellt werden. Die erste Gabelung nach dem Startpunkt führt zum Parallelisierungsprozess mithilfe von drei Spalten. Hierbei ist die Gabelung oder Fork Node durch einen waagerechten blauen Balken dargestellt. Die parallelen Steuerungsflüsse der Spalten laufen zum Bestimmen des Wirkungsgrades wieder zusammen. Ziel aller drei Spalten ist es, den Wirkungsgrad zu bestimmen.

1. Erste vertikale Integration mit Drehzahl und Schlupf,
2. zweite vertikale Integration mit Drehmoment, Leistung und Wirkungsgrad sowie
3. dritte vertikale Integration mit Frequenz und Spannung.

Die erste vertikale Integration ermöglicht von Drehzahl über Schlupf das Bestimmen des Wirkungsgrades des Asynchronmotors. Dies hängt von drei Gleichungen ab.

$$s = 1 - \frac{n}{n_d} \text{ mit s: schlupf; n: L ä uferdrehzahl; } n_d : \text{ synchrone Drehzahl} \tag{5.5}$$

$$\text{Wir} = 1 - \frac{n}{n_d} \text{ mit Wir: Wirkungsgrad; n: l ä uferdrehzahl; } n_d : \text{ synchrone Drehzahl} \tag{5.6}$$

$$\text{Wir} = 1 - s \text{ mit Wir: Wirkungsgrad; s: Schlupf} \tag{5.7}$$

Gemäß Abb. 5.10 und 5.11 ist die erste vertikale Integration mit einem Parallelisierungsprozess zwischen den Aktionen Drehzahl und Schlupf dargestellt. Dies führt mithilfe eines Zusammenführens von zwei Aktionen, genannt *„Decision Node“*, zum Bestimmen des Wirkungsgrades. Hierbei werden zwei nebenläufige Prozesse bis zum Bestimmen des Wirkungsgrades zusammengeführt. Der Parallelisierungsprozess ermöglicht das Bestimmen des Wirkungsgrades entweder mit der Kenngröße Drehzahl oder mit der Kenngröße Schlupf.

Die zweite vertikale Integration erstreckt sich von der Aktion *„Drehmoment“* über die Aktion *„Leistung“* bis zur Aktion *„Wirkungsgrad“*. Die drei Kenngrößen gehören zu den wichtigen Kenngrößen zum Charakterisieren des Asynchronmotors. Die zweite vertikale Integration stellt zum einen die Proportionalität der Leistung P_{ab} zu dem Drehmoment M und zum anderen die Proportionalität des Wirkungsgrades Wir zu P_{ab} dar.

Gemäß Abb. 5.10 und 5.11 sind die drei Aktionen miteinander mithilfe der Steuerungsflüsse, genannt „*Control Flows*“, verbunden. Die vertikale Integration läuft direkt zum Bestimmen des Wirkungsgrades mithilfe von zwei Gleichungen.

$$P_{ab} = 2 * 3{,}14 * n * M \text{ mit } P_{ab}: \text{ mechanische Leistung; n: Drehzahl; M: Drehmoment} \tag{5.8}$$

$$\text{Wir} = \frac{P_{ab}}{P_{zu}} \text{ mit Wir: Wirkungsgrad; } P_{ab}: \text{ mechanische Leistung; } P_{zu}: \text{ elektrische Leistung} \tag{5.9}$$

Die dritte vertikale Integration verfügt über zwei Aktionen oder Kenngrößen: Frequenz und Spannung.

Die Aktionen sind miteinander mithilfe von Steuerungsflüssen zum Bestimmen des Wirkungsgrades verbunden. Gemäß Abb. 5.10 und 5.11 ist die dritte vertikale Integration sowohl zu der zweiten als auch zu der ersten vertikalen Integration parallel. Die dritte vertikale Integration stellt die dritte Spalte dieses Diagramms zum Modellieren der Energieeffizienz des Motors dar. Die dritte vertikale Integration wie es auf den Abb. 5.10 und 5.11 zu sehen ist, zeigt, dass die Frequenz und die Motorspannung bei Frequenzumrichter zueinander proportional verstellt werden. Diese Proportionalität ermöglicht die Realisierung der Kennlinien-Steuerung, genannt Spannungs-Frequenz-Kennlinien, für Frequenzumrichter. Diese Kennlinien ermöglicht die Erzielung eines guten Wirkungsgrades. Die dritte vertikale Integration zeigt eine Abhängigkeit zwischen der Motorspannung und der Frequenz mithilfe des Spannungsverhältnisses U_{nenn}/U_{stator} und Frequenzverhältnisses f_{stator}/f_{nenn}. Gemäß Abb. 5.10 und 5.11 sind die beiden Verhältnisse gleich. Die Gl. 5.10 stellt die Spannungs-Frequenz-Kennlinien bezüglich der Proportionalität von der Spannung zu der Frequenz.

$$\frac{U_{nenn}}{U_{stator}} = \frac{f_{stator}}{f_{nenn}} \text{ mit } U_{nenn}: \text{Nennspannung; } U_{stator}: \text{Ständerspannung; } f_{stator}: \text{Ständerfrequenz; } f_{nenn}: \text{ Nennfrequenz} \tag{5.10}$$

Die Gl. 5.10 ermöglicht das Bestimmen der Nennspannung mithilfe der Gl. 5.11

$$U_{\text{nenn}} = U_{\text{stator}} * \frac{f_{\text{stator}}}{f_{\text{nenn}}} \tag{5.11}$$

5.2.2 Horizontale Integration

Horizontale Integration mit dem Aktivitätsdiagramm stellt die Visualisierung der Modellierungsprozesse auf einer horizontalen Schicht mithilfe einerseits der Aktionen und andererseits der Steuerungsflüsse dar. Die horizontale Integration ist mit der Linearität der Modellierung verbunden. Die Modellierung einer horizontalen Integration mit dem Aktivitätsdiagramm ermöglicht die Darstellung der Schichten im Diagramm. Jede Schicht verfügt über Aktionen und deren Verbindungen. Gemäß Abb. 5.10 und 5.11 verfügt die Aktivität drei Schichte mit insgesamt sieben Aktionen. Die erste Schicht enthält drei Aktionen: Frequenz, Drehmoment und Drehzahl. Die zweite Schicht hat zwei Aktionen: Leistung und Schlupf. Die dritte Schicht hat auch zwei Aktionen: Spannung und Wirkungsgrad.

Die erste Schicht, genannt erste horizontale Integration, beginnt mit der Parallelisierung der Aktion oder Kenngröße *„Frequenz“* entweder zur Aktion *„Drehmoment“* oder zur Aktion *„Leistung“*. Dieser Parallelisierungsprozess ermöglicht z. B. die Darstellung der zeitlich unabhängigen Nebenläufigkeitsprozesse *„Bestimmung des Drehmoments“* und *„Bestimmung der Leistung“* mithilfe der Gl. 5.11 bzw. 5.12, wie es auf den Abb. 5.10 und 5.11 zu sehen ist, Dieser Parallelisierungsprozess wird mithilfe des Splitting-Knotens oder *„Fork Node“* realisiert, wobei *„Bestimmung des Drehmoments“* und *„Bestimmung der Leistung“* nicht gleichzeitig durchgeführt werden müssen.

$$M = \frac{P_{\text{ab}}\left(1 - \frac{f_r}{f_s}\right)}{2} * 3{,}14 * n \text{ mit M: Drehmoment; } P_{\text{ab}}\text{: abgegebene Leistung; } f_r\text{: Rotorfrequenz; } f_s\text{: Ständerfrequenz; n: Läuferdrehzahl} \tag{5.11}$$

$$P_{\text{ab}} = P_{\text{zu}}\left(1 - \frac{f_r}{f_s}\right) \text{ mit } P_{\text{zu}}\text{: zugeführte Leistung (elektrische Leistung)} \tag{5.12}$$

Die erste Schicht geht linear von der Aktion *„Frequenz“* über die Aktion *„Drehmoment“* bis zur Aktion *„Drehzahl“*. Von der Kenngröße *„Drehmoment“* geht es linear zu der Kenngröße *„Drehzahl“* mithilfe einer nicht proportionellen Gleichung genannt Gl. 5.13, wobei diese Gleichung stellt auch die Kennlinie Drehzahl-Drehmoment für den Asynchronmotor dar. Hierbei ist die Drehzahl n nicht proportionell zu dem Drehmoment M, wie es auf den Abb. 5.10 und 5.11 zu sehen ist.

$$n = \frac{P_{ab}}{2 * 3{,}14 * M} \text{ mit } P_{ab}\text{: abgegebene Leistung; M: Drehmoment; n: Läuferdrehzahl} \tag{5.13}$$

Die zweite Schicht, genannt zweite horizontale Integration, beginnt mit der Aktion *„Schlupf“* und endet mit der Aktion *„Leistung“*. Mithilfe dieser Schicht wird der Motor bezüglich der Bestimmung der Leistung aus der Kenngröße Schlupf charakterisiert. Der Schlupf ist eine wichtige Betriebskenngröße zum Charakterisieren des energieeffizienten Asynchronmotors [2]

Mit der dritten Schicht von der Aktion *„Spannung“* zu der Aktion *„Wirkungsgrad“* gibt es einen Parallelisierungsprozess, welcher mit einem Splitter-Knoten von der Aktion *„Spannung“* zu den Aktionen *„Leistung“* und *„Wirkungsgrad“* führt. Hierbei müssen *„Bestimmung der Leistung“* und *„Bestimmung des Wirkungsgrades“* nicht gleichzeitig durchgeführt werden. Die Nebenläufigkeitsprozesse *„Bestimmung der Leistung“* und *„Bestimmung des Wirkungsgrades“* werden mit den Gl. 5.14 bzw. 5.15 realisiert.

Gemäß Gl. 5.14 ist die abgegebene Leistung P_{ab} zu der Spannung U proportional.

$$P_{ab} = 1{,}73 * \text{U} * \text{I} * \text{Wir} * \text{CosPhi} \text{ mit U: Spannung; I: Strom; Wir: Wirkungsgrad; CosPhi: Cosinus } \varphi \tag{5.14}$$

$$\text{Wir} = \frac{P_{ab}}{1{,}73 * U * I * \text{CosPhi}} \tag{5.15}$$

Tab. 5.2 gibt einen Überblick über die Darstellung der Aktionen oder Kenngrößen zum Bestimmen des Asynchronmotors. Die erste Zeile stellt die erste Schicht der Aktivitätsmodellierung, genannt erste horizontale Integration oder horizontale Integration1, dar. Die erste Zeile besteht aus n, M und f_r f_s.

Die horizontale Intergation2 besteht aus s und P_{ab}, während die dritte Schicht aus Wir und U besteht.

Erste vertikale Integration besteht aus n und s, zweite vertikale Integration besteht aus M, P_{ab}, und die dritte vertikale Integration enthält f_r oder f_s, und U.

Die Tab. 5.2 ermöglicht das Analysieren der Aktionen des UML-Aktivitätsdiagramms bezüglich der geometrischen Position der Modellierung: horizontale und vertikale Modellierung.

Tab. 5.2 Darstellung der Aktionen für die Schichten der horizontalen und vertikalen Integration

Integration	Vertikale integration1	Vertikale Integration2	Vertikale Intergation3
Horizontale Integration1	Drehzahl n	Drehmoment M	Frequenz f_r oder f_s
Horizontale Integration2	Schlupf s	Leistung P_{ab}	
Horizontale Integration3		Wirkungsgrad Wir	Spannung U

Geometrische Modellierung mithilfe des UML-Aktivitätsdiagramms

5.3 Modellierung mit Klassendiagrammen

Parallele Modellierung mit dem Klassendiagramm stellt die Anwendung eines statischen Diagramms in der Modellierung der Nebenläufigkeitsprozesse dar. Darstellungen der Klassen mithilfe vom UML ermöglichen das Analysieren der Merkmale der Klassen während der Modellierung. Hierbei ist der Parallelisierungsprozess mithilfe der Beziehungen zwischen den Klassen zu analysieren.

Modellierung mit dem Klassendiagramm erzielt die Darstellungen der Klassen und deren Unterklassen mithilfe der objektorientierten Modellierung. Zum einen werden den Zusammenhang zwischen den Objekten der Klassen dargestellt. Zum anderen werden die Klassen-Objekte während des Parallelisierungsprozesses gekapselt.

Parallelisierungsprozesse mit dem Klassendiagramm beschreiben die Vererbungshierarchie zwischen Oberklassen und Unterklassen. Die Parallelität stellt das Modellieren von Strukturen der Klassen mithilfe des Objektorientierungskonzepts dar.

5.3.1 Vererbungshierarchie

Modellierung der Vererbung ermöglicht das Analysieren der Merkmale von Oberklassen bis zu Unterklassen hinsichtlich Attribute und Operationen um die Objekte zu Orientieren. Mithilfe der Modellierung des Klassendiagramms werden Charakteristika der Tochterklassen beschrieben, damit die Vererbungslinie mithilfe einer durchgezogenen Linie mit einem geschlossenen hohlen Pfeil dargestellt werden können.

Abb. 5.12 gibt einen Überblick über die Vererbungshierarchie bezüglich der Parallelisierungsprozesse zum Charakterisieren des Asynchronmotors. Hierbei sind die Kenngrößen Drehzahl, Drehmoment, Leistung, Frequenz, Schlupf und Wirkungsgrad zum Realisieren der Einsparpotenziale bei elektrischen Energieverbrauchern [2]. Abb. 5.12 beschreibt die Vererbungshierarchie einerseits zwischen der Oberklasse *„Interface4Motor“* und den Unterklassen *„Drehzahl4Motor“* und *„Leistung4Motor“* und andererseits zwischen dem Ganzen *„Hauptprogramm“* und seinen Teilen *„Drehzahl4Motor“* und *„Leistung4Motor“*. In dem Klassendiagramm der Abb. 5.12 wird die Vererbung mithilfe einer durchgezogenen Linie mit einem geschlossenen, hohlen Pfeil dargestellt.

Gemäß 5.12 handelt es sich um Vererbung, weil die Tochter-Objekte *„Drehzahl4Motor“* und *„Leistung4Motor“* ähnliche Merkmale ihres Mutter-Objektes *„Interface4Motor“* erhalten. Beispielsweise erbt das Objekt *Drehzahl4Motor* von dem Interface *„Interface4Motor“* Merkmale wie z. B.. die Operation *„DrehzahlBerechnen“*. Es ist zu bemerken, dass die Operation zum Berechnen von Schlupf zum einen von der Drehzahl und zum anderen vom Wirkungsgrad abhängt. Parallel sind die Operationen für die Berechnungen des Wirkungsgrades und der abgegebenen Leistung vom Objekt *„Leistung4Motor“* einerseits von Attributen *„Wirkungsgrad“* bzw. *„abgegebeneLeistung“* und andererseits von Operationen *„drehzahlBerechnen()“* bzw. *„leistungsBerechnen()“*

abhängig. Allerdings kann die Operation *„drehmoment()“* vom Interface *„Interface4-Motor“* zum Berechnen der abgegebenen Leistung implementiert werden. Die Parallelisierungsprozesse ermöglichen das nebenläufige Berechnen der Kenngrößen mithilfe der Unterklassen *„Drehzahl4Motor“* und *„Leistung4Motor“*.

Die Vererbungsbeziehung mithilfe der Interaktion *„Komposition“* stellt eine Beziehung zwischen dem Ganzen *„Hauptklasse“* und den Teilen *„Drehzahl4Motor“* und *„Leistung4Motor“* dar. Gemäß Abb. 5.12 gibt es starke Beziehungen zwischen der Hauptklasse *„Hauptprogramm“* und den Teilen *„Drehzahl4Motor“* und *„Leistung4Motor“*, sodass die beiden Teilklassen nicht ohne die Ganzklasse existieren können. Die Beziehungsart ermöglicht den Aufruf der Methoden der Teilklassen in der Ganzklasse. Beispielsweise werden die Methoden *„wirkungsgradBerechnen()“* und *„schlupfBerechnen()“* der Klassen „Leistung4Motor“ bzw. „Drehzahl4Motor“ in der Methode main() der Ganzklasse *„Hauptprogramme“* mit der *„new-Schlüsselwörter“* dargestellt. Die Existenz der Teilklassen „Leistung4Motor“ und *„Drehzahl4Motor“* hängt von dem Aufruf der Methoden *„wirkungsgradBerechnen()“* oder *„abgegebeneLeistungsBerechnen()“* bzw. *„schlupfBerechnen()“* oder *„synchronedrehzahlBerechnen()“* in der Methode main() der Ganzklasse *„Hauptptogramm“* ab. Wenn das Ganze gelöscht wird, so werden auch automatisch alle Teile gelöscht. Teile können auch gelöscht werden, bevor das Ganze seine Gültigkeit verliert [3].

5.3.2 Modellieren der Strukturen der Klassen

Strukturen der Klassen ermöglichen die Anwendungen der Objekte bzw. Klassen in der objektorientierten Modellierung. Hierbei verfügen die Klassen die Objekte, deren Struktur zwei Eigenschaften beinhalten: Attribut und Operation.

Hinsichtlich der Parallelisierung bei der Modellierung der Klassen sind die Eigenschaften der Klassen zur Analyse der Struktur der Klasse wichtig. Die Eigenschaften der Klassen stellen die strukturelle Modellierung bezüglich der Beschreibung des Datentyps. Hierbei werden mithilfe sowohl der Attribute als auch der Operationen die Datentypen beschrieben. Parallelisierungsmodellierung ermöglicht die Interaktionen zwischen Klassen bezüglich der Implementierung der beschriebenen Struktur. Beispielsweise verfügen die Klassen *„Leistung4Motor“* und *„Drehzahl4Motor“* über Operationen wie z. B. *wirkungsgradBerechnen()* bzw. *schlupfBerechnen()* zur Bestimmung des Wirkungsgrades bzw. des Schlupfes mithilfe der Attribute *„wirkungsgrad“* bzw. *„schlupf“* aus dem Interface *„Interface4Motor“*, wie es auf der Abb. 5.12 zu sehen ist. Nebenläufige Prozesse in der Klasse *„Leistung4Motor“* ermöglichen beispielsweise einerseits Berechnungen des Wirkungsgrades aus zwei Attributen *„abgegebeneLeistung“* und *“zugefuehrteLeistung“* und andererseits aus dem Attribut *„schlupf“*. Aus dem Interface *„Interface4Motor“* kann der Wirkungsgrad mithilfe sowohl der Methode *„drehzahlBerechnen()“* als auch der Methode *„leistungsBerechnen()“* berechnet werden. Dies bedeutet, dass der Wirkungsgrad eine Funktion sowohl von Drehzahl als auch von Leistung ist und damit zu letzten

genannten Kenngrößen proportional ist, wie die Gl. 5.6 und 5.15–5.16 dies darstellen. Gl. 5.16 zeigt, dass die Implementierung der Bestimmung des Wirkungsgrades von denen von Drehzahl und Leistung abhängt.

$$\eta = f(n) = f(\mathrm{P}) \text{ mit } \eta\text{: wirkungsgrad; } n\text{: Laeuferdrehzahl; } P\text{: abgegebene Leistung} \tag{5.16}$$

Parallel zu den Bestimmungen der Kenngrößen Wirkungsgrad und Leistung aus der Unterklasse *„Leistung4Motor“* werden die Bestimmungen der Kenngrößen Schlupf und Drehzahl der Unterklasse *„Drehzahl4Motor“* mithilfe der Eigenschaften, d. h. Attribute und Operationen, dem parallelen Prozess der Implementierung folgen. Hierbei wird der Begriff Parallelisierungsprozess die nebenläufige Implementierung der Unterklasse „Drehzahl4Motor“ mithilfe von Attribute und Operationen darstellen.

Die Implementierung der Klasse „*Drehzahl4Motor*“ kann einerseits mithilfe der Operation *„schlupfBerechnen()“* und andererseits mit *„synchronedrehzahlBerechnen()“* erfolgen. Hierbei kann der Schlupf einerseits mithilfe der Attribute *„zugefuehrteleistung“* und *„abgegebeneLeistung“* und andererseits mithilfe des Attributes *„wirkungsgrad“*. Dies bedeutet, dass der Schlupf sowohl von der abgegebenen Leistung als auch vom Wirkungsgrad abhängt und damit zu diesen Kenngrößen proportional wie die Gl. 5.17–5.19 dies darstellen.

$$s = f(P) = f(\eta) \text{ mit s: Schlupf des Motors; P: Leistung; } \eta\text{: Wirkungsgrad} \tag{5.17}$$

$$s = 1 - \eta \text{ mit s: Schlupf des Motors; } \eta\text{: Wirkungsgrad} \tag{5.18}$$

$$s = 1 - \frac{\mathrm{P_{ab}}}{\mathrm{P_{zu}}} \text{ mit } \mathrm{P_{ab}}\text{: abgegebene Leistung; } \mathrm{P_{zu}}\text{ : zugeführte Leistung} \tag{5.19}$$

Gemäß Abb. 5.12 kann die synchrone Drehzahl aus der Klasse *„Drehzahl4Motor“* einerseits mithilfe der Attribute Ständerfrequenz und Polpaarzahl und andererseits mithilfe der Operation *„drehzahlBerechnen()“*, deren Implementierung im Interface „Interface4Motor“ erfolgt, berechnet werden. Die synchrone Drehzahl auch Drehfelddrehzahl ist eine wichtige Kenngröße zum Charakterisieren des Asynchronmotors. Diese Kenngröße ist mithilfe der Gl. 5.20 bestimmt, wobei die synchrone Drehzahl und die Ständerfrequenz zueinander proportional sind.

$$n_d = \frac{f}{p} \text{ mit f: Ständerfrequenz; p: Polpaarzahl} \tag{5.20}$$

5.3.3 Parallelisierung der objektorientierten Modellierung

Parallelisierung von Klassen fokussiert auf die objektorientierten Modellierungen bezüglich der Beschreibung der Strukturen der Objekte zu modellieren. Zwei Klassen sind zueinander parallel, wenn ihre Eigenschaften zueinander parallel sind. Hierbei werden die Eigenschaften der parallelen Klassen bezüglich der Äquivalenzrelation dargestellt.

Die modellierten Klassen *„Leistungs4Motor“* und *„Drehzahl4Motor“* sind parallel, weil es eine Äquivalenzrelation zwischen beiden gibt. Außerdem stellt die Parallelität mithilfe der Äquivalenzrelation das Betrachten der Symmetrie, Transitivität und Reflexivität dar [4].

Das Beispiel aus dem Asynchronmotor stellt die Äquivalenzrelation bezüglich des UML-Klassendiagrammes gemäß Abb. 5.12 dar. Eine Menge von Kenngrößen, genannt M, für den Asynchronmotor definiert die zweistellige Relation ~. Für je zwei Kenngrößen Wirkungsgrad und Schlupf aus M soll Wirkungsgrad ~ Schlupf gelten, wenn Wirkungsgrad und Schlupf Kenngrößen derselben Art sind. Für die Klassen *„Leistung4Motor“* und *„Drehzahl4Motor“* abgekürzt *„L“* bzw. *„D“* gilt L ~ D, wobei es eine Symmetrie zwischen L und D gibt:

L ~ D -> D~L oder Leistung4Motor ~ Drehzahl4Motor-> Drehzahl4Motr ~ Leistung4 Motor.

Die Transitivität bedeutet, dass die dritte Klasse *„Hauptprogramm“* abgekürzt H eine Rolle für die drei Forderungen nach Äquivalenzrelationen oder Parallelenrelationen spielt. Beispielsweise sind *„Leistung4Motor“* und *„Drehzahl4Motor“* Klassen derselben Art und ebenso sind *„Drehzahl4Motor“* und *„Hauptprogramm“* derselben Art. Dann sind auch *„Leistung4Motor“* und *„Hauptprogramm“* derselben Art. Hierbei sind *„Leistung4Motor“* und *„Drehzahl4Motor“* Teilklassen der Ganzklasse *„Hauptprogramm“*, wobei zum einen *„Leistung4Motor“* und *„Drehzahl4Motor“* Parallelenklassen bezüglich der Äquivalenzrelationen zwischen beiden Klassen darstellen und zum anderen beide Klassen gemäß Abb. 5.12 über gleiche Anzahl an Methoden verfügen.

Anschließend besagt die Reflexivität bezüglich der Individualität, dass jede Klasse derselben Art wie sie selbst ist. D. h., dass jede der drei Klassen ihre Partikularität enthält. Es ist zu bemerken, dass die Ganzklasse *„Hauptklasse“* nur die main-Methode enthält.

5.4 Modellierung mit Sequenzdiagrammen

Parallelisierungsmodellierungen mit UML stellen das Analysieren der Interaktionen zwischen Objekten dar, wobei die Objekte mithilfe der Nachrichten miteinander kommunizieren. Sequenzdiagramme geben Überblicke über das Beschreiben der Zusammenarbeit zwischen Objekten. Sowohl synchrone als auch asynchrone Kommunikationen definieren den Verlauf der Interaktionen zwischen Objekten. Bezüglich der Parallelisierungsmodellierung mit UML spielen die Interaktionen zwischen Objekten eine wesentliche Rolle zum horizontalen Parallelisieren der Objekte. Hierbei werden Interaktion zwischen zwei Objekten als horizontale Gerade dargestellt, wobei die geometrische Parallelität zwischen zwei Geraden die Darstellungen der Elemente dieser Geraden betrachtet. Im Bereich Software-Entwicklung ist der Begriff Parallelität mit Äquivalenzklassen verbunden, d. h. zwei Interaktionen sind parallel, wenn ihre entsprechende Nachrichten zueinander parallel sind. Modellierung mit Sequenzdiagrammen

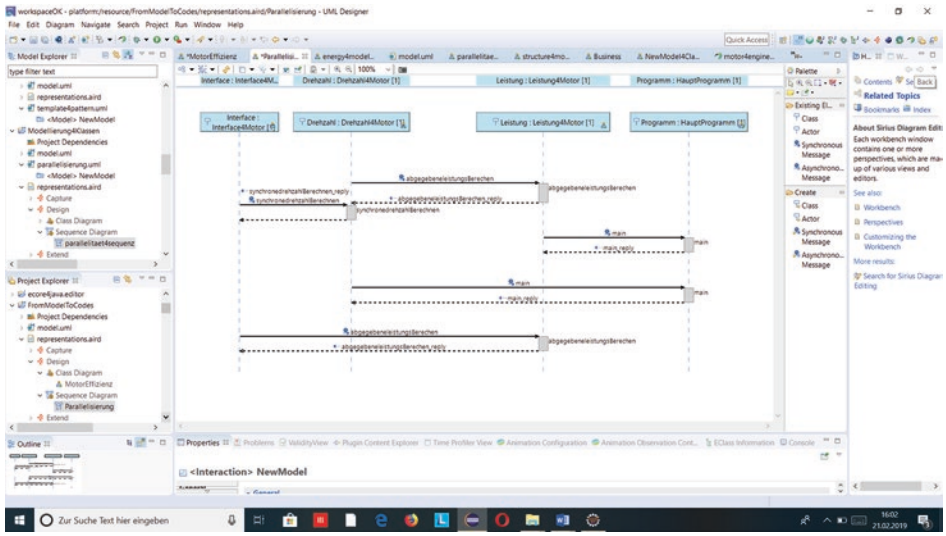


Abb. 5.13 Sequenzdiagramm zum Modellieren der Charakterisierung des Asynchronmotors

ermöglichen das Modellieren der Funktionen eines Systems mithilfe von Nebenläufigkeitsprozessen. Die Parallelität der Funktionen wird mithilfe sowohl der Objekte als auch deren Interaktionen ermöglicht. Modellieren mit Sequenzdiagrammen bedeutet, dass das System sowohl auf horizontalen als auch vertikalen Ebene modelliert wird. Hierbei wird sowohl die horizontale als auch die vertikale Integration modelliert.

Abb. 5.13 stellt das Modellieren der Funktionalität des Asynchronmotors bezüglich der Charakterisierung der Betriebskenngrößen, u. a. Drehzahl, Leistung, Schlupf und Wirkungsgrad mit dem Sequenzdiagramm, dar. Gemäß Abb. 5.13 verfügt das Sequenzdiagramm über sechs Objekte und ihre entsprechende Klasse: Umdrehung, Bewegungskraft, Drehzahldifferenz, Energie, Effizienz und Anwendung bzw. Drehzahl, Drehmoment, Schlupf, Leistung, Wirkungsgrad und Hauptprogramm.

Gemäß Abb. 5.13 stellen die Objekte *„Drehzahldifferenz“*, *„Energie“* und *„Effizienz“* die Berechnungen der entsprechenden Kenngrößen des Asynchronmotors *„Schlupf“*, *„Leistung“*, *„Wirkungsgrad“* mithilfe der entsprechenden Methoden *„schlupf4drehzahl()“*, *„leistung4drehmoment()“* und *„“wirkunsgrad4drehzahl“* dar. Die Klasse *„Wirkungsgrad“* ermöglicht mithilfe des Objektes *„Effizienz“* die Darstellung der Effizienz des Asynchronmotors zum Berechnen des Wirkungsgrades des Motors. Das Objekt *„Anwendung“* der Klasse *„Hauptprogramm“* enthält die Funktionalität des Programmes zum Charakterisieren des Asynchronmotors.

Parallelisierungsprozesse mit Sequenzdiagrammen können mithilfe sowohl von Lebenslinien oder Objekten für die vertikale Modellierung als auch von Interaktionen zwischen Objekten für die horizontale Modellierung dargestellt werden.

5.4.1 Darstellung der Parallelisierungsprozesse mit Objekten oder Lebenslinien

Gemäß Abb. 5.13 stellen die Objekte diejenige der Klassen in der objektorientierten Modellierung dar. Hierbei verfügen die Lebenslinien über Objekte, wobei jedes Objekt nur eine Lebenslinie enthält. Abb. 5.13 verfügt über sechs Lebenslinien bzw. Objekte. Die Existenzlaufzeit der Objekte wird mithilfe der Lebenslinien dargestellt. Gemäß 5.13 zeigen die Lebenslinien eine vertikale Orientierung oder Integration. Lebenslinien zeigen zum einen zeitlichen und zum anderen aufeinanderfolgende Prozesse zum Charakterisieren der Objekte. Dies ermöglicht eine Beschreibung der Parallelisierung bezüglich der Position der Objekte.

Die Parallelisierungsprozesse fokussieren auf die geometrische Parallelisierung sowohl zwischen Objekten als auch zwischen Lebenslinien. Zwei Lebenslinien oder Objekte sind zueinander parallel, wenn es eine Relation zwischen beiden gibt. Beispielsweise gibt es eine Verbindung zwischen den Objekten *„Umdrehung“* und *„Bewegungskraft“* bzw. den Interfaces *„Drehzahl“* und *„Drehmoment“*. Beide Interfaces gehören zu der gleichen Äquivalenzklasse und stellen eine Drehzahl-Drehmoment-Kennlinie für das Charakterisieren des Asynchronmotors dar. Drehzahl und Drehmoment sind zueinander parallel, weil ihre Objekte *„Umdrehung“* und *„Bewegungskraft“* sowohl in einer Beziehung sind als auch in der gleichen Äquivalenzklasse liegen. Anschließend gibt es Parallelitäten zum einen zwischen den Objekten *„Drehzahldifferenz“* und *„Energie“* und zum anderen zwischen den Objekten *„Effizienz“* und *„Anwendung“*. Die Drehzahldifferenz ermöglicht das Bestimmen des Schlupfes, dessen Wert wichtig zum Charakterisieren des Asynchronmotors ist. Die Energie wird mithilfe der Leistung dargestellt. Es gibt eine Relation zwischen den Objekten *„Drehzahldifferenz“* und *„Energie“* bzw. den Klassen *„Schlupf“* und *„Leistung“*. Die Leistung P ist zu dem Schlupf proportional, weil P = f(s) ist. Die Gl. 5.21 abgeleitet aus der Gl. 5.19 stellt die Parallelität zwischen den beiden Objekten *„Drehzahldifferenz“* und *„Energie“* dar:

$$P_{\text{ab}} = \frac{1-s}{P_{\text{zu}}} \text{ mit } P_{\text{ab}}\text{: abgegebene Leistung; } P_{\text{zu}}\text{: zugeführte Leistung; s: Schlupf des Motors} \tag{5.21}$$

Mit den Objekten *„Effizienz“* und *„Energie“* wird die parallele Modellierung die Möglichkeit zum Analysieren der Komponenten der Charakterisierung des Motors sowohl mithilfe der Objekte *„Energie“* als auch *„Effizienz“* dargestellt. Parallelisieren bedeutet, dass diese Objekte zu einander parallel sind. Ziel ist es den Wirkungsgrad des Motors mithilfe der Parallelisierung zu bestimmen. Die Parallelisierung zwischen Wirkungsgrad und Leistung ist mithilfe der Gl. 5.22 dargestellt:

$$\eta = \frac{P_{\text{ab}}}{P_{\text{zu}}} \text{ mit } P_{\text{ab}}\text{: abgegebene Leistung; } P_{\text{zu:}}\text{zugeführte Leistung; } \eta\text{: Wirkungsgrad} \tag{5.22}$$

Gemäß Gl. 5.22 sind die Objekte Effizienz und Energie zueinander parallel, weil zum einen der Wirkungsgrad gleich dem Verhältnis von Leistungen ist und zum anderen beide Objekte zu den Parallelenklassen gehören. Hierbei werden Objekte oder Lebenslinien parallel bezeichnet, wenn sie in derselben Äquivalenzklasse liegen. Für zwei parallele Objekte *„Energie“* und *„Effizienz“* gilt, dass sie entweder identisch oder disjunkt sind, wie die Gl. 5.23 es darstellt.

$$\text{Energie} \| \text{Effizienz} \rightarrow (\text{Energie}) = (\text{Effizienz}) \text{ oder } (\text{Energie}) \cap (\text{Effizienz}) = \emptyset \tag{5.23}$$

Gemäß Abb. 5.13 ist die Äquivalenzklasse eines Objektes *„Drehzahldifferenz“* die Klasse der Objekte, die äquivalent oder isomorph zu *„Drehzahldifferenz“* sind.

- **Symmetrie:** *„Drehzahldifferenz“ ~ „Energie“ -> „Energie“ ~ „Drehzahldifferenz“*;
- **Transitivität:** *„Drehzahldifferenz“ ~ „Energie“* und *„Energie“ ~ „Effizienz“ -> „Drehzahldifferenz“ ~ „Effizienz“*;
- **Reflexivität:** *„Drehzahldifferenz“ ~ „Drehzahldifferenz“*, *„Energie“ ~ „Energie“*, *„Effizienz ~ „Effizienz“*.

Mithilfe der funktionellen Modellierung wird die Parallelität zwischen Objekten oder Lebenslinien bezüglich der Abhängigkeiten zwischen ihnen dargestellt. Die Objekte *„Umdrehung“*, genannt Drehzahl, und *„Drehzahldifferenz“* sind zueinander parallel, weil beide Objekte in der Abhängigkeitsbeziehung sind. Das heißt, der Schlupf s, genannt Drehzahldifferenz, ist eine Funktion der Läuferdrehzahl n mithilfe der Gl. 5.5 und 5.24. Anschließend sind die Objekte Drehzahldifferenz und Energie auch in der Abhängigkeitsbeziehung, weil die Energie oder Leistung eine Funktion des Schlupfs ist, wie die Gl. 5.19 und 5.25 dies zeigen. Gemäß Gl. 5.22 und 5.26 sind die Objekte Energie und Effizienz, genannt Wirkungsgrad, in der Abhängigkeitsbeziehung mithilfe der funktionellen Modellierung. Hierbei gilt:

$$n \parallel s \rightarrow s = f(n) \text{ mit n: Läuferdrehzahl; s: Schlupf des Motors} \tag{5.24}$$

$$P \parallel s \rightarrow s = f(P) \text{ mit P: abgegebene Leistung; s: Schlupf des Motors} \tag{5.25}$$

$$P \parallel \eta \rightarrow \eta = f(P) \text{ mit P: abgegebene Leistung; } \eta\text{: Wirkungsgrad} \tag{5.26}$$

5.4.2 Darstellung der Parallelisierungsprozesse mit Interaktionen

Darstellungen der horizontalen Modellierungen werden mithilfe der Interaktionen sowohl zwischen Objekten oder deren Lebenslinien als auch zwischen Aktivitätsbalken sichtbar, welche zeitliche Aufgabe zum Beenden eines Prozesses zeigt. Zwischen zwei Aktivitätsbalken oder Lebenslinien gibt es eine Operation.

Zwei Interaktionen sind parallel, wenn die Operationen zwischen den Lebenslinien parallel sind. Gemäß Abb. 5.13 sind die Äquivalenzrelationen zwischen den Interaktionen bzw. Operationen bezüglich der Darstellungen der Parallelisierungsprozesse zu betrachten. Hierbei werden die Interaktionen zum einen zwischen Objekten oder Lebenslinien zum anderen zwischen den Aktivitätsbalken die Abhängigkeitsbeziehungen zur Modellierung der Parallelitäten dargestellt. Mithilfe der Interaktionen sind die horizontalen Modellierungen bezüglich der Parallelisierungsprozesse zu beschreiben. Horizontale Modellierungen mithilfe des UML-Sequenzdiagramms ermöglichen das Analysieren der Parallelitäten zwischen Operationen zum Symbolisieren der Beziehungen, genannt Äquivalenzrelationen. Gemäß Abb. 5.13 stellen die Operationen *„schlupf4drehzahl()“* *„leistung4drehmoment()“* und *„wirkungsgrad4drehzahl()“* Interaktionen zwischen den Objekten *„Umdrehung“* und *„Drehzahldifferenz“*, *„Bewegungskraft“* und *„Energie“* bzw. *„Umdrehung“* und *„Effizienz“* dar. Gemäß Abb. 5.13 sind die Operationen zueinander parallel.

Symmetrie:

schlupf4drehzahl||leistung4drehmoment => leistung4drehmoment||schlupf4drehzahl

Transitivität:

schlupf4drehzahl||leistung4drehmoment und leistung4drehmoment||wirkungsgrad4drehzahl
⇒ schlupf4drehzahl||wirkungsgrad4drehzahl

Reflexivität:

schlupf4drehzahl||schlupf4drehzahl oder leistung4drehmoment
||leistung4drehmoment oder wirkungsgrad4drehzahl||wirkungsgrad4drehzahl

Ebenfalls sind die Interaktionen *„wirkungsgrad4schlupf()“*,*„leistung4schlupf()“* und *„wirkungsgrad4leistung()“* zueinander parallel, d. h., dass sie in gleicher Äquivalenzklasse liegen. Dies ist mithilfe der Abhängigkeitsbeziehung symbolisiert mit „~“ bezüglich der Symmetrie, Transitivität und der Reflexivität dargestellt. Der Begriff *„äquivalent“* oder *„parallel“* zeigt Beziehungen zwischen zwei Interaktionen oder Operationen. Wenn zwei Interaktionen wie z. B. Operation1 und Operation2 äquivalent sind, ist diese Beziehung mit Operation1 ~ Operation2 symbolisiert. Gemäß Abb. 5.13 hat der Begriff *„äquivalent“* die drei folgenden Eigenschaften:

1. Symmetrie: wirkungsgrad4schlupf() ~ leistung4schlupf() => leistung4schlupf() ~ wirkungsgrad4schlupf()
2. Transitivität: wirkungsgrad4schlupf()“ ~ leistung4schlupf() und leistung4schlupf() ~ wirkungsgrad4leistung() => wirkungsrad4schlupf() ~ wirkungsgrad4leistung()
3. Reflexivität: wirkungsgrad4schlupf() ~ wirkungsgrad4schlupf(); leistung4schlupf() ~ leistung4schlupf(); wirkungsgrad4schlupf() ~ wirkungsgrad4schlupf().

Die Symmetrie zeigt, dass wenn ***"Operation1"*** zu ***"Operation2"*** äquivalent ist, dann ist auch ***"Operation2"*** zu ***"Operation1"*** äquivalent. Anschließend zeigt die Transitivität, dass wenn ***"Operation1"*** zu ***"Operation2"*** äquivalent und ***"Operation2"*** zu ***"Operation3"*** äquivalent ist, dann ist ***"Operation1"*** äquivalent zu ***"Operation3"***. Bei der Reflexivität ist jede Interaktion oder Operation zu sich selbst äquivalent ist.

5.5 Zusammenfassung

UML stellt die Modellierung nebenläufiger Systeme mithilfe von Diagrammen wie z. B. Sequenzdiagrammen, Aktivitäts- und Zustandsautomaten dar.

Zustandsdiagramme stellen die kontrollierten Zustände von Zustandsautomaten wie z. B. Objekten oder Systemen während bestimmter Laufzeit dar, wobei sie Ereignisse zum Auslösen der Zustandsübergänge angeben. Hierbei ermöglichen Zustandsdiagramme die Synchronisation der Funktionen der Systeme mithilfe der Zustände. Das Open-Source-Framework Eclipse-UML-Designer von der Firma Obeo Designer ermöglicht die Parallele Modellierung mit Zustandsdiagrammen. Der Editor von UML-Designer verfügt über mehrere UML-Diagramme, u. a. Klassen-, Aktivitäts-, Sequenz-, oder Zustandsdiagramme zum Modellieren von Softwaresystemen. Das Zustandsdiagramm verfügt über Modellierungstools, genannt Palette, welche in zwei Teile eingegliedert ist: *„Types"* und *„Relationships"*.

Modellierung horizontaler Schichten stellt die Modellierung von Zuständen auf einer Linie oder einer Ebene dar. Hierbei werden Aktivitäten der Zustände berücksichtigt. Die Informationsmodellierung stellt die Strukturierung der Information hinsichtlich der Integration der Modelle dar. Hierbei werden die Informationen nach Orientierungen modelliert. Mithilfe der UML-Diagramme wie z. B. Zustand-, Sequenz-, Aktivitäts-, oder Klassendiagramme werden sich die Informationen der Modelle nach geometrischen Orientierungen bewegen, wobei es möglich ist, dass die Informationen sich entweder vertikal oder horizontal bewegen. Sowohl Synchronisierungen als auch Parallelisierung der Informationen ermöglichen die dynamischen Modellierungen der Objekte, wobei die Prozesse der Parallelisierung oder der Synchronisierung die Struktur der Informationen darstellen.

Aktivitätsdiagramme stellen die Modellierungen zum einen der Vorgänge, genannt Workflow, und zum anderen der Prozesse zum Aneinanderreihen sowohl der Aktivitäten als auch der Funktionen dar. Ziel der Anwendung der Aktivitätsdiagramme ist es, die Anwendungsfalldiagramme, genannt Use-Case-Diagramme, zu ergänzen und beschreiben. Modellierung mit Aktivitätsdiagrammen ermöglicht das Beschreiben des Verhaltens des Systems mithilfe von Prozessen oder Workflows.

Dieser Abschnitt setzt das Aktivitätsdiagramm zum Beschreiben von Nebenläufigkeitsprozessen bezüglich sowohl der parallelen Modellierung als auch der Synchronisierungsmodellierung ein. Mithilfe der Abstraktion wird das Aktivitätsdiagramm Prozesse oder Workflows beschrieben.

Der Modellierungskern mit dem Aktivitätsdiagramm ist das Beschreiben der Aktivitäten, genannt *„Activity“*, die über Aktionen zum Beschreiben der Prozesse verfügen. Aktionen stellen wesentliche Vorgänge zur Modellierung des Systems dar. Das Aktivitätsdiagramm vom Open Source UML-Designer verfügt über Modellierungstools, genannt *„activity tools“*, die aus Elementen wie z. B. *„Partition“*, *„Nodes“*, *„Actions“*, *„Activity Parameter“* oder *„Flows“* bestehen.

Die Modellierung mit dem Aktivitätsdiagramm von UML-Designer ermöglicht die Beschreibungen der IT-Lösungen für den Drehstromantrieb. Die Asynchronmaschine ist der meist benutzte Maschinentyp und deren Einsatz sowohl in Geräten des Haushaltes als auch in der Industrie die Erklärung der Anwendungen in Energietechnik dargestellt. Vertikale Modellierung mit dem Aktivitätsdiagramm ermöglicht das Analysieren der Modellierung in die vertikale Position, wobei die Aktionen mithilfe der Pfeile der vertikalen Richtung folgen werden. Hierbei werden die Aktionen und deren Control Flow in eine Spalte integriert. Die vertikale Integration mit dem Aktivitätsdiagramm stellt eine geometrische Richtung der Modellierung mithilfe der Spalten dar. Die Spalten orientiert sich automatisch vertikal und von oben nach unten, wobei Startknoten und Endknoten die Eingliederungen der Spalten in einer geometrischen Integration ermöglichen, d. h. vertikalen Integration. Horizontale Integration mit dem Aktivitätsdiagramm stellt die Visualisierung der Modellierungsprozesse auf einer horizontalen Schicht mithilfe einerseits der Aktionen und andererseits der Steuerungsflüsse dar. Die horizontale Integration ist mit der Linearität der Modellierung verbunden. Die Modellierung einer horizontalen Integration mit dem Aktivitätsdiagramm ermöglicht die Darstellung der Schichten im Diagramm. Jede Schicht verfügt über Aktionen und deren Verbindungen.

Parallele Modellierung mit dem Klassendiagramm stellt die Anwendung eines statischen Diagramms in der Modellierung der Nebenläufigkeitsprozesse dar. Darstellungen der Klassen mithilfe von UML ermöglichen das Analysieren der Merkmale der Klassen während der Modellierung. Hierbei ist der Parallelisierungsprozess mithilfe der Beziehungen zwischen den Klassen zu analysieren.

Modellierung mit dem Klassendiagramm erzielt die Darstellungen der Klassen und deren Unterklassen mithilfe der objektorientierten Modellierung. Zum einen wird der Zusammenhang zwischen den Objekten der Klassen dargestellt. Zum anderen werden die Klassen-Objekte während des Parallelisierungsprozesses gekapselt.

Parallelisierungsprozesse mit dem Klassendiagramm beschreiben die Vererbungshierarchie zwischen Oberklassen und Unterklassen. Die Parallelität stellt das Modellieren von Strukturen der Klassen mithilfe des Objektorientierungskonzeptes dar. Hinsichtlich der Parallelisierung bei der Modellierung der Klassen sind die Eigenschaften der Klassen zur Analyse der Struktur der Klasse wichtig. Die Eigenschaften der Klassen stellen die strukturelle Modellierung bezüglich der Beschreibung des Datentyps dar. Hierbei werden mithilfe sowohl der Attribute als auch der Operationen die Datentypen beschrieben. Parallelisierungsmodellierung ermöglicht die Interaktionen zwischen Klassen bezüglich der Implementierung der beschriebenen Struktur. Parallelisierung von Klassen fokussiert auf die objektorientierten Modellierungen bezüglich der

Beschreibung der Strukturen der Objekte zu modellieren. Zwei Klassen sind zueinander parallel, wenn ihre Eigenschaften zueinander parallel sind. Hierbei werden die Eigenschaften der parallelen Klassen bezüglich der Äquivalenzrelation dargestellt.

Parallelisierungsmodellierungen mit UML stellen das Analysieren der Interaktionen zwischen Objekten dar, wobei die Objekte mithilfe der Nachrichten miteinander kommunizieren. Sequenzdiagramme geben Überblicke über das Beschreiben der Zusammenarbeit zwischen Objekten. Sowohl synchrone als auch asynchrone Kommunikationen definieren den Verlauf der Interaktionen zwischen Objekten. Im Bereich Software-Entwicklung ist der Begriff Parallelität mit Äquivalenzklassen verbunden, d. h. zwei Interaktionen sind parallel, wenn ihre entsprechende Nachrichten zueinander parallel sind. Modellierung mit Sequenzdiagrammen ermöglichen das Modellieren der Funktionen eines Systems mithilfe von Nebenläufigkeitsprozessen. Die Parallelität der Funktionen wird mithilfe sowohl der Objekte als auch deren Interaktionen ermöglicht. Modellieren mit Sequenzdiagrammen bedeutet, dass das System sowohl auf horizontalem als auch vertikalem Ebene modelliert wird. Hierbei wird sowohl die horizontale als auch die vertikale Integration modelliert. Die Parallelisierungsprozesse fokussieren auf die geometrische Parallelisierung sowohl zwischen Objekten als auch zwischen Lebenslinien. Zwei Lebenslinien oder Objekte sind zueinander parallel, wenn es eine Relation zwischen beiden gibt. Beispielsweise gibt es eine Verbindung zwischen den Objekten „Umdrehung“ und *„Bewegungskraft“* bzw. den Interfaces *„Drehzahl“* und *„Drehmoment“*. Beide Interfaces gehören zu der gleichen Äquivalenzklasse und stellen eine Drehzahl-Drehmoment-Kennlinie für das Charakterisieren des Asynchronmotors dar. Drehzahl und Drehmoment sind zueinander parallel, weil ihre Objekte *„Umdrehung“* und *„Bewegungskraft“* sowohl in einer Beziehung sind als auch in der gleichen Äquivalenzklasse liegen.

Literatur

1. Wikipedia Web Portal: Zustandsdiagramme, In: UML**, WIKIMEDIA project, MediaWiki,** https://de.wikipedia.org/wiki/Zustandsdiagramm_(UML) (2018).
2. Nyamsi, E., A.: Realisierung der Einsparpotentiale bei Energieverbrauchern, Springer Vieweg (2018).
3. Hardy, D.: UML für IT-Berufe, Verlag Europa-Lehrmittel (2011).
4. Beutelspacher, A., Danckwerts, R., Nickel, G., Spies, S., Wickel, G.:. Mathematik neu denken, Vieweg + Teubner, ISBN: 978–3-83 48-1648-1 (2011).

Vom Modellieren zum Programmieren

6

6.1 Anwendung von Java Swing in der Entwicklung der grafischen Oberfläche [1]

Das Model-View-Controller (MVC Design Pattern) ist ein Entwurfmuster bezüglich der Strukturierung von Software zum Trennen von Daten-Modell und dessen grafischer Darstellung [2]. Das Muster besteht aus drei Komponenten: Das Modell genannt model, die Darstellung genannt view und die Steuerung genannt *controller.*

Java-Swing basiert auf dem MVC-Konzept, das die Unterteilung einer GUI-Komponente in *„model“*, *„view“* und *„controller“* vorsieht [3]. Hierbei enthält das Modell die Daten, während das View seine Daten auf der Benutzeroberfläche anzeigt (z. B. in einer XDEV-Tabelle) und der Controller Ereignisse auf der GUI-Komponente registriert [1].

XDEV ist eine visuelle Java -Entwicklungsumgebung für Rapid Application Development (RAD). Mit der Entwicklungsumgebung XDEV lassen sich professionelle Web- und Desktopapplikationen auf Basis von Java entwickeln.

Listing 6.1 stellt die Implementierung der Oberklasse *XdevWindow* in der Klasse *WindkraftManagement* zum Entwickeln der Benutzeroberfläche dar. Mithilfe der Autovervollständigung wurden Imports für die verwendeten Klassen automatisch erzeugt. Listing 6.1 zeigt die Implementierung der GUI-Komponenten von Java XDEV 5 wie z. B. Button, Formular oder Container in der Klasse *WindkraftManagement.*

Die Implementierung von XdevWindow fokussiert auch auf verschiedene Tools u. a. ActionEvent, EventHandlerDelegate, Event Model, Event Sources und EventListeners, Schnittstellen wie z. B. FormularAdapter und WindowAdapter. Das erzeugen eines Fenster erfoglt mithilfe von GUI-Dialogelementen wie z. B. LayoutManager(Nyamsi 2018). Gemäß Listing 6.1 wird die Verwendung von *FlowLayout, GridLayout* und *GridLayout* bei Containern und Formularen mithilfe der Methode *setLayout()* realisiert.

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML,*
https://doi.org/10.1007/978-3-658-29057-3_6

Listing 6.1 zeigt wie XDEV -GUI-Komponenten wie z. B. Jbutton ode JTable zum Implementieren der Schnistellen durch lokale Klassen eingesetzt werden.

Listing 6.1 Erzeungung von Codes für die Implementierung der Klasse *Windkraftmanagement* aus mithilfe des XDEV 5 Java Frameworks

```
package Fenster;
import xdev.lang.EventHandlerDelegate;
import xdev.ui.*;
import xdev.ui.text.TextFormat;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Locale;
import javax.swing.SwingConstants;
public class WindkraftManagement
extends XdevWindow // ${GENERATED-CODE-LINE:BEAN_SUPERCLASS}
{

 @EventHandlerDelegate void this_windowClosing(WindowEvent
 arg0)// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
 {// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
  close();
 }// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
 @EventHandlerDelegate  void  cmdNew_actionPerformed(ActionEvent  arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
 {// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
  formular.reset(VirtuelleTabellen.Aufgenommeneleistung.VT);
 }// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
 @EventHandlerDelegate void cmdReset_actionPerformed(ActionEvent arg0)
 // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
 {// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
  formular.reset();
 }// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
 @EventHandlerDelegate void cmdSave_actionPerformed(ActionEvent arg0)
 // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
 {// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
  if(formular.verifyFormularComponents())
  {
   try
```

```
   {
    formular.save();
   }
   catch(Exception e)
   {
    e.printStackTrace();
   }
  }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSaveAndNew_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular.verifyFormularComponents())
 {
  try
  {
   formular.save();
   formular.reset(VirtuelleTabellen.Aufgenommeneleistung.VT);
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdNew2_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular2.reset(VirtuelleTabellen.Drehzahl.VT);
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdReset2_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular2.reset();
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSave2_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular2.verifyFormularComponents())
 {
  try
  {
   formular2.save();
  }
  catch(Exception e)
```

```
   {
    e.printStackTrace();
   }
  }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSaveAndNew2_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular2.verifyFormularComponents())
 {
  try
  {
   formular2.save();
   formular2.reset(VirtuelleTabellen.Drehzahl.VT);
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdNew3_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular3.reset(VirtuelleTabellen.Drehzahldrehmomentkennlinie.VT);
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdReset3_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular3.reset();
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSave3_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular3.verifyFormularComponents())
 {
  try
  {
   formular3.save();
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
```

```
@EventHandlerDelegate void cmdSaveAndNew3_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular3.verifyFormularComponents())
 {
  try
  {
   formular3.save();
   formular3.reset(VirtuelleTabellen.Drehzahldrehmomentkennlinie.VT);
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdNew4_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular4.reset(VirtuelleTabellen.Polpaarzahl.VT);
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate   void   cmdReset4_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular4.reset();
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSave4_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular4.verifyFormularComponents())
 {
  try
  {
   formular4.save();
  }
  catch(Exception e)
  {
  e.printStackTrace();
   }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSaveAndNew4_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular4.verifyFormularComponents())
```

```
 {
  try
  {
   formular4.save();
   formular4.reset(VirtuelleTabellen.Polpaarzahl.VT);
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdNew5_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular5.reset(VirtuelleTabellen.Schlupf.VT);
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate   void   cmdReset5_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular5.reset();
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSave5_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular5.verifyFormularComponents())
 {
  try
  {
   formular5.save();
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSaveAndNew5_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular5.verifyFormularComponents())
 {
  try
  {
   formular5.save();
   formular5.reset(VirtuelleTabellen.Schlupf.VT);
```

```
   }
   catch(Exception e)
   {
    e.printStackTrace();
   }
  }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdNew6_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular6.reset(VirtuelleTabellen.Wirkungsgrad.VT);
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate    void    cmdReset6_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular6.reset();
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSave6_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular6.verifyFormularComponents())
 {
  try
  {
   formular6.save();
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSaveAndNew6_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular6.verifyFormularComponents())
 {
  try
  {
   formular6.save();
   formular6.reset(VirtuelleTabellen.Wirkungsgrad.VT);
  }
  catch(Exception e)
  {
   e.printStackTrace();
```

```
   }
  }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdNew7_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular7.reset(VirtuelleTabellen.Inneresmoment.VT);
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate    void    cmdReset7_actionPerformed(ActionEvent
arg0)// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 formular7.reset();
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSave7_actionPerformed(ActionEvent arg0)
// ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular7.verifyFormularComponents())
 {
  try
  {
   formular7.save();
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
@EventHandlerDelegate void cmdSaveAndNew7_actionPerformed(ActionEvent
arg0) // ${GENERATED-CODE-BLOCK-START:EVENT_HANDLER_DELEGATE}
{// ${GENERATED-CODE-BLOCK-END:EVENT_HANDLER_DELEGATE}
 if(formular7.verifyFormularComponents())
 {
  try
  {
   formular7.save();
   formular7.reset(VirtuelleTabellen.Inneresmoment.VT);
  }
  catch(Exception e)
  {
   e.printStackTrace();
  }
 }
}// ${GENERATED-CODE-LINE:EVENT_HANDLER_DELEGATE}
```

```
//Generated definitions, do not edit! ${GENERATED-CODE-BLOCK-START:D
EFINITIONS}
XdevContainer container, container2, container3, container4,
container5, container6,
  container7;
XdevFormattedTextField formattedTextField2, formattedTextField,
formattedTextField4,
  formattedTextField3, formattedTextField6, formattedTextField7,
  formattedTextField8,
  formattedTextField9, formattedTextField10, formattedTextField11,
  formattedTextField12,
  formattedTextField13, formattedTextField14, formattedTextField15,
  formattedTextField16,
  formattedTextField17, formattedTextField18, formattedTextField19,
  formattedTextField20,
  formattedTextField21, formattedTextField5, formattedTextField23,
  formattedTextField22,
  formattedTextField25, formattedTextField24, formattedTextField27,
  formattedTextField26,
  formattedTextField29, formattedTextField28;
XdevTab LeistungAus, Drezahl, EinLeistung, Polparzahl, Schlupf,
Wirkungsgrad,
  Moment;
XdevButton cmdNew, cmdReset, cmdSave, cmdSaveAndNew, cmdNew2,
cmdReset2, cmdSave2,
  cmdSaveAndNew2, cmdNew3, cmdReset3, cmdSave3, cmdSaveAndNew3,
  cmdNew4, cmdReset4,
  cmdSave4, cmdSaveAndNew4, cmdNew5, cmdReset5, cmdSave5,
  cmdSaveAndNew5, cmdNew6,
  cmdReset6, cmdSave6, cmdSaveAndNew6, cmdNew7, cmdReset7, cmdSave7,
  cmdSaveAndNew7;
XdevTextField textField;
XdevFormular formular, formular2, formular3, formular4, formular5,
formular6,
  formular7;
XdevTabbedPane AufLeistung;
XdevLabel label2, label, label4, label3, label6, label7, label8,
label9, label10,
  label11, label12, label13, label14, label15, label16, label17,
  label18, label19,
  label20, label21, label22, label5, label24, label23, label26,
  label25, label28,
  label27, label30, label29;
//End generated definitions ${GENERATED-CODE-BLOCK-END:DEFINITIONS}
```

```
{//Generated initializers, do not edit! ${GENERATED-CODE-BLOCK-START:
INITIALIZERS}
 AufLeistung = new XdevTabbedPane();
 LeistungAus = new XdevTab();
 formular = new XdevFormular();
 label2 = new XdevLabel();
 formattedTextField2 = new XdevFormattedTextField();
 label = new XdevLabel();
 formattedTextField = new XdevFormattedTextField();
 container = new XdevContainer();
 cmdNew = new XdevButton();
 cmdReset = new XdevButton();
 cmdSave = new XdevButton();
 cmdSaveAndNew = new XdevButton();
 Drezahl = new XdevTab();
 formular2 = new XdevFormular();
 label4 = new XdevLabel();
 formattedTextField4 = new XdevFormattedTextField();
 label3 = new XdevLabel();
 formattedTextField3 = new XdevFormattedTextField();
 container2 = new XdevContainer();
 cmdNew2 = new XdevButton();
 cmdReset2 = new XdevButton();
 cmdSave2 = new XdevButton();
 cmdSaveAndNew2 = new XdevButton();
 EinLeistung = new XdevTab();
 formular3 = new XdevFormular();
 label6 = new XdevLabel();
 formattedTextField6 = new XdevFormattedTextField();
 label7 = new XdevLabel();
 formattedTextField7 = new XdevFormattedTextField();
 label8 = new XdevLabel();
 formattedTextField8 = new XdevFormattedTextField();
 label9 = new XdevLabel();
 formattedTextField9 = new XdevFormattedTextField();
 label10 = new XdevLabel();
 formattedTextField10 = new XdevFormattedTextField();
 label11 = new XdevLabel();
 textField = new XdevTextField();
 label12 = new XdevLabel();
 formattedTextField11 = new XdevFormattedTextField();
 label13 = new XdevLabel();
 formattedTextField12 = new XdevFormattedTextField();
 label14 = new XdevLabel();
 formattedTextField13 = new XdevFormattedTextField();
```

```
label15 = new XdevLabel();
formattedTextField14 = new XdevFormattedTextField();
label16 = new XdevLabel();
formattedTextField15 = new XdevFormattedTextField();
label17 = new XdevLabel();
formattedTextField16 = new XdevFormattedTextField();
label18 = new XdevLabel();
formattedTextField17 = new XdevFormattedTextField();
label19 = new XdevLabel();
formattedTextField18 = new XdevFormattedTextField();
label20 = new XdevLabel();
formattedTextField19 = new XdevFormattedTextField();
label21 = new XdevLabel();
formattedTextField20 = new XdevFormattedTextField();
label22 = new XdevLabel();
formattedTextField21 = new XdevFormattedTextField();
label5 = new XdevLabel();
formattedTextField5 = new XdevFormattedTextField();
container3 = new XdevContainer();
cmdNew3 = new XdevButton();
cmdReset3 = new XdevButton();
cmdSave3 = new XdevButton();
cmdSaveAndNew3 = new XdevButton();
Polparzahl = new XdevTab();
formular4 = new XdevFormular();
label24 = new XdevLabel();
formattedTextField23 = new XdevFormattedTextField();
label23 = new XdevLabel();
formattedTextField22 = new XdevFormattedTextField();
container4 = new XdevContainer();
cmdNew4 = new XdevButton();
cmdReset4 = new XdevButton();
cmdSave4 = new XdevButton();
cmdSaveAndNew4 = new XdevButton();
Schlupf = new XdevTab();
formular5 = new XdevFormular();
label26 = new XdevLabel();
formattedTextField25 = new XdevFormattedTextField();
label25 = new XdevLabel();
formattedTextField24 = new XdevFormattedTextField();
container5 = new XdevContainer();
cmdNew5 = new XdevButton();
cmdReset5 = new XdevButton();
cmdSave5 = new XdevButton();
cmdSaveAndNew5 = new XdevButton();
```

```
  Wirkungsgrad = new XdevTab();
  formular6 = new XdevFormular();
  label28 = new XdevLabel();
  formattedTextField27 = new XdevFormattedTextField();
  label27 = new XdevLabel();
  formattedTextField26 = new XdevFormattedTextField();
  container6 = new XdevContainer();
  cmdNew6 = new XdevButton();
  cmdReset6 = new XdevButton();
  cmdSave6 = new XdevButton();
  cmdSaveAndNew6 = new XdevButton();
  Moment = new XdevTab();
  formular7 = new XdevFormular();
  label30 = new XdevLabel();
  formattedTextField29 = new XdevFormattedTextField();
  label29 = new XdevLabel();
  formattedTextField28 = new XdevFormattedTextField();
  container7 = new XdevContainer();
  cmdNew7 = new XdevButton();
  cmdReset7 = new XdevButton();
  cmdSave7 = new XdevButton();
  cmdSaveAndNew7 = new XdevButton();

  this.setPreferredSize(new Dimension(600,400));
  LeistungAus.setTitle("AufLeistung");
  LeistungAus.setIndex(0);
  label2.setText("Id");
  label2.setName("label2");
  formattedTextField2.setDataField("VirtuelleTabellen.
  Aufgenommeneleistung.Id");
  formattedTextField2.setTabIndex(1);
  formattedTextField2.setName("formattedTextField2");

 formattedTextField2.setTextFormat(TextFormat.getNumberInstance(Locale.
 getDefault(),null,0,0,false,false));
  formattedTextField2.setHorizontalAlignment(SwingConstants.LEFT);
  label.setText("MechanischeLeistung");

 formattedTextField.setDataField("VirtuelleTabellen.
 Aufgenommeneleistung.MechanischeLeistung");
  formattedTextField.setTabIndex(2);

 formattedTextField.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
```

```
formattedTextField.setHorizontalAlignment(SwingConstants.LEFT);
 cmdNew.setTabIndex(3);
 cmdNew.setText("Neu");
 cmdReset.setTabIndex(4);
 cmdReset.setText("Zurücksetzen");
 cmdSave.setTabIndex(5);
 cmdSave.setText("Speichern");
 cmdSaveAndNew.setTabIndex(6);
 cmdSaveAndNew.setText("Speichern + Neu");
 Drezahl.setTitle("Drehzahl");
 Drezahl.setIndex(1);
 formular2.setName("formular2");
 label4.setText("Id");
 label4.setName("label4");
 formattedTextField4.setDataField("VirtuelleTabellen.Drehzahl.Id");
 formattedTextField4.setTabIndex(7);
 formattedTextField4.setName("formattedTextField4");

formattedTextField4.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
 formattedTextField4.setHorizontalAlignment(SwingConstants.LEFT);
 label3.setText("Drehzahlbereich");
 formattedTextField3.setDataField("VirtuelleTabellen.Drehzahl.Dreh-
 zahlbereich");
 formattedTextField3.setTabIndex(8);

formattedTextField3.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
 formattedTextField3.setHorizontalAlignment(SwingConstants.LEFT);
 cmdNew2.setTabIndex(9);
 cmdNew2.setText("Neu");
 cmdNew2.setName("cmdNew2");
 cmdReset2.setTabIndex(10);
 cmdReset2.setText("Zurücksetzen");
 cmdReset2.setName("cmdReset2");
 cmdSave2.setTabIndex(11);
 cmdSave2.setText("Speichern");
 cmdSave2.setName("cmdSave2");
 cmdSaveAndNew2.setTabIndex(12);
 cmdSaveAndNew2.setText("Speichern + Neu");
 cmdSaveAndNew2.setName("cmdSaveAndNew2");
 EinLeistung.setTitle("EinLeistung");
 EinLeistung.setIndex(2);
 formular3.setName("formular3");
 label6.setText("Id");
```

```
 label6.setName("label6");
 formattedTextField6.setDataField("VirtuelleTabellen.Drehzahldreh-
 momentkennlinie.Id");
 formattedTextField6.setTabIndex(13);
 formattedTextField6.setName("formattedTextField6");

formattedTextField6.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
 formattedTextField6.setHorizontalAlignment(SwingConstants.LEFT);
 label7.setText("Nennleistung");
 label7.setName("label7");

formattedTextField7.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Nennleistung");
 formattedTextField7.setTabIndex(14);
 formattedTextField7.setName("formattedTextField7");

formattedTextField7.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
 formattedTextField7.setHorizontalAlignment(SwingConstants.LEFT);
 label8.setText("Nennspannung");
 label8.setName("label8");
 formattedTextField8.setDataField("VirtuelleTabellen.Drehzahldreh-
 momentkennlinie.Nennspannung");
 formattedTextField8.setTabIndex(15);
 formattedTextField8.setName("formattedTextField8");

formattedTextField8.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
 formattedTextField8.setHorizontalAlignment(SwingConstants.LEFT);
 label9.setText("Netzfrequenz");
 label9.setName("label9");

formattedTextField9.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Netzfrequenz");
 formattedTextField9.setTabIndex(16);
 formattedTextField9.setName("formattedTextField9");

formattedTextField9.setTextFormat(TextFormat.getNumberInstance(Locale.
getDefault(),null,0,0,false,false));
 formattedTextField9.setHorizontalAlignment(SwingConstants.LEFT);
 label10.setText("Nennstrom");
 label10.setName("label10");
```

```
formattedTextField10.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Nennstrom");
 formattedTextField10.setTabIndex(17);
 formattedTextField10.setName("formattedTextField10");

formattedTextField10.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField10.setHorizontalAlignment(SwingConstants.LEFT);
 label11.setText("Wicklungsschaltung");
 label11.setName("label11");

textField.setDataField("VirtuelleTabellen.Drehzahldrehmomentkenn-
linie.Wicklungsschaltung");
 textField.setTabIndex(18);
 textField.setMaxSignCount(100);
 textField.setHorizontalAlignment(SwingConstants.LEFT);
 label12.setText("Volllastblindleistung");
 label12.setName("label12");

formattedTextField11.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Volllastblindleistung");
 formattedTextField11.setTabIndex(19);
 formattedTextField11.setName("formattedTextField11");

formattedTextField11.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField11.setHorizontalAlignment(SwingConstants.LEFT);
 label13.setText("VolllastCosPhie");
 label13.setName("label13");

formattedTextField12.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.VolllastCosPhie");
 formattedTextField12.setTabIndex(20);
 formattedTextField12.setName("formattedTextField12");

formattedTextField12.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,2,2,true,false));
 formattedTextField12.setHorizontalAlignment(SwingConstants.LEFT);
 label14.setText("Nenndrehzahl");
 label14.setName("label14");

formattedTextField13.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Nenndrehzahl");
 formattedTextField13.setTabIndex(21);
 formattedTextField13.setName("formattedTextField13");
```

```
formattedTextField13.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField13.setHorizontalAlignment(SwingConstants.LEFT);
 label15.setText("Nennschlupf");
 label15.setName("label15");

formattedTextField14.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Nennschlupf");
 formattedTextField14.setTabIndex(22);
 formattedTextField14.setName("formattedTextField14");

formattedTextField14.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,2,2,true,false));
 formattedTextField14.setHorizontalAlignment(SwingConstants.LEFT);
 label16.setText("Nennwirkungsgrad");
 label16.setName("label16");

formattedTextField15.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Nennwirkungsgrad");
 formattedTextField15.setTabIndex(23);
 formattedTextField15.setName("formattedTextField15");

formattedTextField15.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,2,2,true,false));
 formattedTextField15.setHorizontalAlignment(SwingConstants.LEFT);
 label17.setText("EingespeisteLeistung_id");
 label17.setName("label17");

formattedTextField16.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.EingespeisteLeistung_id");
 formattedTextField16.setTabIndex(24);
 formattedTextField16.setName("formattedTextField16");

formattedTextField16.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField16.setHorizontalAlignment(SwingConstants.LEFT);
 label18.setText("Polpaarzahl_id");
 label18.setName("label18");

formattedTextField17.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Polpaarzahl_id");
 formattedTextField17.setTabIndex(25);
 formattedTextField17.setName("formattedTextField17");
```

```
formattedTextField17.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField17.setHorizontalAlignment(SwingConstants.LEFT);
 label19.setText("AufgenommeneLeistung_id");
 label19.setName("label19");

formattedTextField18.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.AufgenommeneLeistung_id");
 formattedTextField18.setTabIndex(26);
 formattedTextField18.setName("formattedTextField18");

formattedTextField18.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField18.setHorizontalAlignment(SwingConstants.LEFT);
 label20.setText("Drehzahl_id");
 label20.setName("label20");

formattedTextField19.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Drehzahl_id");
 formattedTextField19.setTabIndex(27);
 formattedTextField19.setName("formattedTextField19");

formattedTextField19.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField19.setHorizontalAlignment(SwingConstants.LEFT);
 label21.setText("Schlupf_id");
 label21.setName("label21");

formattedTextField20.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Schlupf_id");
 formattedTextField20.setTabIndex(28);
 formattedTextField20.setName("formattedTextField20");

formattedTextField20.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField20.setHorizontalAlignment(SwingConstants.LEFT);
 label22.setText("Wirkungsgrad_id");
 label22.setName("label22");

formattedTextField21.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.Wirkungsgrad_id");
 formattedTextField21.setTabIndex(29);
 formattedTextField21.setName("formattedTextField21");
```

```
formattedTextField21.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField21.setHorizontalAlignment(SwingConstants.LEFT);
 label5.setText("InneresMoment_id");

formattedTextField5.setDataField("VirtuelleTabellen.Drehzahldreh-
momentkennlinie.InneresMoment_id");
 formattedTextField5.setTabIndex(30);

formattedTextField5.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField5.setHorizontalAlignment(SwingConstants.LEFT);
 cmdNew3.setTabIndex(31);
 cmdNew3.setText("Neu");
 cmdNew3.setName("cmdNew3");
 cmdReset3.setTabIndex(32);
 cmdReset3.setText("Zurücksetzen");
 cmdReset3.setName("cmdReset3");
 cmdSave3.setTabIndex(33);
 cmdSave3.setText("Speichern");
 cmdSave3.setName("cmdSave3");
 cmdSaveAndNew3.setTabIndex(34);
 cmdSaveAndNew3.setText("Speichern + Neu");
 cmdSaveAndNew3.setName("cmdSaveAndNew3");
 Polparzahl.setTitle("Polpaarzahl");
 Polparzahl.setIndex(3);
 formular4.setName("formular4");
 label24.setText("Id");
 label24.setName("label24");
 formattedTextField23.setDataField("VirtuelleTabellen.Polpaarzahl.
 Id");
 formattedTextField23.setTabIndex(35);
 formattedTextField23.setName("formattedTextField23");

formattedTextField23.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField23.setHorizontalAlignment(SwingConstants.LEFT);
 label23.setText("Zahl");
 formattedTextField22.setDataField("VirtuelleTabellen.Polpaarzahl.
 Zahl");
 formattedTextField22.setTabIndex(36);

formattedTextField22.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField22.setHorizontalAlignment(SwingConstants.LEFT);
```

```
 cmdNew4.setTabIndex(37);
 cmdNew4.setText("Neu");
 cmdNew4.setName("cmdNew4");
 cmdReset4.setTabIndex(38);
 cmdReset4.setText("Zurücksetzen");
 cmdReset4.setName("cmdReset4");
 cmdSave4.setTabIndex(39);
 cmdSave4.setText("Speichern");
 cmdSave4.setName("cmdSave4");
 cmdSaveAndNew4.setTabIndex(40);
 cmdSaveAndNew4.setText("Speichern + Neu");
 cmdSaveAndNew4.setName("cmdSaveAndNew4");
 Schlupf.setTitle("Schlupf");
 Schlupf.setIndex(4);
 formular5.setName("formular5");
 label26.setText("Schlupfbereich");
 label26.setName("label26");
 formattedTextField25.setDataField("VirtuelleTabellen.Schlupf.
 Schlupfbereich");
 formattedTextField25.setTabIndex(41);
 formattedTextField25.setName("formattedTextField25");

formattedTextField25.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,4,6,false,false));
 formattedTextField25.setHorizontalAlignment(SwingConstants.LEFT);
 label25.setText("Id");
 formattedTextField24.setDataField("VirtuelleTabellen.Schlupf.Id");
 formattedTextField24.setTabIndex(42);

formattedTextField24.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField24.setHorizontalAlignment(SwingConstants.LEFT);
 cmdNew5.setTabIndex(43);
 cmdNew5.setText("Neu");
 cmdNew5.setName("cmdNew5");
 cmdReset5.setTabIndex(44);
 cmdReset5.setText("Zurücksetzen");
 cmdReset5.setName("cmdReset5");
 cmdSave5.setTabIndex(45);
 cmdSave5.setText("Speichern");
 cmdSave5.setName("cmdSave5");
 cmdSaveAndNew5.setTabIndex(46);
 cmdSaveAndNew5.setText("Speichern + Neu");
 cmdSaveAndNew5.setName("cmdSaveAndNew5");
 Wirkungsgrad.setTitle("Wirkungsgrad");
```

```
 Wirkungsgrad.setIndex(5);
 formular6.setName("formular6");
 label28.setText("Id");
 label28.setName("label28");
 formattedTextField27.setDataField("VirtuelleTabellen.Wirkungsgrad.
 Id");
 formattedTextField27.setTabIndex(47);
 formattedTextField27.setName("formattedTextField27");

formattedTextField27.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField27.setHorizontalAlignment(SwingConstants.LEFT);
 label27.setText("GeneratorWirkungsgrad");

formattedTextField26.setDataField("VirtuelleTabellen.Wirkungsgrad.
GeneratorWirkungsgrad");
 formattedTextField26.setTabIndex(48);

formattedTextField26.setTextFormat(TextFormat.getPercentInstance(Local
e.getDefault(),null,4,5,false,false));
 formattedTextField26.setHorizontalAlignment(SwingConstants.LEFT);
 cmdNew6.setTabIndex(49);
 cmdNew6.setText("Neu");
 cmdNew6.setName("cmdNew6");
 cmdReset6.setTabIndex(50);
 cmdReset6.setText("Zurücksetzen");
 cmdReset6.setName("cmdReset6");
 cmdSave6.setTabIndex(51);
 cmdSave6.setText("Speichern");
 cmdSave6.setName("cmdSave6");
 cmdSaveAndNew6.setTabIndex(52);
 cmdSaveAndNew6.setText("Speichern + Neu");
 cmdSaveAndNew6.setName("cmdSaveAndNew6");
 Moment.setTitle("Moment");
 Moment.setIndex(6);
 formular7.setName("formular7");
 label30.setText("Id");
 label30.setName("label30");
 formattedTextField29.setDataField("VirtuelleTabellen.Inneresmoment.
 Id");
 formattedTextField29.setTabIndex(53);
 formattedTextField29.setName("formattedTextField29");

formattedTextField29.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,0,0,false,false));
 formattedTextField29.setHorizontalAlignment(SwingConstants.LEFT);
```

```
 label29.setText("Drehmoment");
 formattedTextField28.setDataField("VirtuelleTabellen.Inneresmoment.
 Drehmoment");
 formattedTextField28.setTabIndex(54);

formattedTextField28.setTextFormat(TextFormat.getNumberInstance(Local
e.getDefault(),null,5,5,false,false));
 formattedTextField28.setHorizontalAlignment(SwingConstants.LEFT);
 cmdNew7.setTabIndex(55);
 cmdNew7.setText("Neu");
 cmdNew7.setName("cmdNew7");
 cmdReset7.setTabIndex(56);
 cmdReset7.setText("Zurücksetzen");
 cmdReset7.setName("cmdReset7");
 cmdSave7.setTabIndex(57);
 cmdSave7.setText("Speichern");
 cmdSave7.setName("cmdSave7");
 cmdSaveAndNew7.setTabIndex(58);
 cmdSaveAndNew7.setText("Speichern + Neu");
 cmdSaveAndNew7.setName("cmdSaveAndNew7");

label2.saveState();
 formattedTextField2.saveState();
 label.saveState();
 formattedTextField.saveState();
 label4.saveState();
 formattedTextField4.saveState();
 label3.saveState();
 formattedTextField3.saveState();
 label6.saveState();
 formattedTextField6.saveState();
 label7.saveState();
 formattedTextField7.saveState();
 label8.saveState();
 formattedTextField8.saveState();
 label9.saveState();
 formattedTextField9.saveState();
 label10.saveState();
 formattedTextField10.saveState();
 label11.saveState();
 textField.saveState();
 label12.saveState();
 formattedTextField11.saveState();
 label13.saveState();
 formattedTextField12.saveState();
```

```
label14.saveState();
formattedTextField13.saveState();
label15.saveState();
formattedTextField14.saveState();
label16.saveState();
formattedTextField15.saveState();
label17.saveState();
formattedTextField16.saveState();
label18.saveState();
formattedTextField17.saveState();
label19.saveState();
formattedTextField18.saveState();
label20.saveState();
formattedTextField19.saveState();
label21.saveState();
formattedTextField20.saveState();
label22.saveState();
formattedTextField21.saveState();
label5.saveState();
formattedTextField5.saveState();
label24.saveState();
formattedTextField23.saveState();
label23.saveState();
formattedTextField22.saveState();
label26.saveState();
formattedTextField25.saveState();
label25.saveState();
formattedTextField24.saveState();
label28.saveState();
formattedTextField27.saveState();
label27.saveState();
formattedTextField26.saveState();
label30.saveState();
formattedTextField29.saveState();
label29.saveState();
formattedTextField28.saveState();

container.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container.add(cmdNew);
container.add(cmdReset);
container.add(cmdSave);
container.add(cmdSaveAndNew);
formular.setLayout(new GridBagLayout());
formular.add(label2,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
```

```
formular.add(formattedTextField2,new  GBC(2,1,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular.add(label,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular.add(formattedTextField,new   GBC(2,2,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular.add(container,new GBC(1,3,2,1,1.0,0.0,GBC.CENTER,GBC.HORIZO
NTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular,true,true);
LeistungAus.setLayout(new BorderLayout());
LeistungAus.add(formular,BorderLayout.CENTER);
container2.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container2.add(cmdNew2);
container2.add(cmdReset2);
container2.add(cmdSave2);
container2.add(cmdSaveAndNew2);
formular2.setLayout(new GridBagLayout());
formular2.add(label4,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular2.add(formattedTextField4,new GBC(2,1,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular2.add(label3,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular2.add(formattedTextField3,new GBC(2,2,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular2.add(container2,new GBC(1,3,2,1,1.0,0.0,GBC.CENTER,GBC.HORI
ZONTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular2,true,true);
Drezahl.setLayout(new BorderLayout());
Drezahl.add(formular2,BorderLayout.CENTER);
container3.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container3.add(cmdNew3);
container3.add(cmdReset3);
container3.add(cmdSave3);
container3.add(cmdSaveAndNew3);
formular3.setLayout(new GridBagLayout());
formular3.add(label6,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField6,new GBC(2,1,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label7,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField7,new GBC(2,2,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
```

```
formular3.add(label8,new GBC(1,3,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField8,new GBC(2,3,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label9,new GBC(1,4,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField9,new GBC(2,4,1,1,1.0,0.0,GBC.BASELI
NE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label10,new GBC(1,5,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField10,new GBC(2,5,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label11,new GBC(1,6,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(textField,new GBC(2,6,1,1,1.0,0.0,GBC.BASELINE_
LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label12,new GBC(1,7,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField11,new GBC(2,7,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label13,new GBC(1,8,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField12,new GBC(2,8,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label14,new GBC(1,9,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField13,new GBC(2,9,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label15,new GBC(1,10,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField14,new GBC(2,10,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label16,new GBC(1,11,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField15,new GBC(2,11,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label17,new GBC(1,12,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField16,new GBC(2,12,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label18,new GBC(1,13,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField17,new GBC(2,13,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label19,new GBC(1,14,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
```

```
formular3.add(formattedTextField18,new GBC(2,14,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label20,new GBC(1,15,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField19,new GBC(2,15,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label21,new GBC(1,16,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField20,new GBC(2,16,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label22,new GBC(1,17,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField21,new GBC(2,17,1,1,1.0,0.0,GBC.BASE
LINE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(label5,new GBC(1,18,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular3.add(formattedTextField5,new GBC(2,18,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular3.add(container3,new GBC(1,19,2,1,1.0,0.0,GBC.CENTER,GBC.HOR
IZONTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular3,true,true);
EinLeistung.setLayout(new BorderLayout());
EinLeistung.add(formular3,BorderLayout.CENTER);
container4.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container4.add(cmdNew4);
container4.add(cmdReset4);
container4.add(cmdSave4);
container4.add(cmdSaveAndNew4);
formular4.setLayout(new GridBagLayout());
formular4.add(label24,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular4.add(formattedTextField23,new GBC(2,1,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular4.add(label23,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular4.add(formattedTextField22,new GBC(2,2,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular4.add(container4,new GBC(1,3,2,1,1.0,0.0,GBC.CENTER,GBC.HORI
ZONTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular4,true,true);
Polparzahl.setLayout(new BorderLayout());
Polparzahl.add(formular4,BorderLayout.CENTER);
container5.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container5.add(cmdNew5);
container5.add(cmdReset5);
container5.add(cmdSave5);
```

```
container5.add(cmdSaveAndNew5);
formular5.setLayout(new GridBagLayout());
formular5.add(label26,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular5.add(formattedTextField25,new GBC(2,1,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular5.add(label25,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular5.add(formattedTextField24,new GBC(2,2,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular5.add(container5,new GBC(1,3,2,1,1.0,0.0,GBC.CENTER,GBC.HORI
ZONTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular5,true,true);
Schlupf.setLayout(new BorderLayout());
Schlupf.add(formular5,BorderLayout.CENTER);
container6.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container6.add(cmdNew6);
container6.add(cmdReset6);
container6.add(cmdSave6);
container6.add(cmdSaveAndNew6);
formular6.setLayout(new GridBagLayout());
formular6.add(label28,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular6.add(formattedTextField27,new GBC(2,1,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular6.add(label27,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular6.add(formattedTextField26,new GBC(2,2,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular6.add(container6,new GBC(1,3,2,1,1.0,0.0,GBC.CENTER,GBC.HORI
ZONTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular6,true,true);
Wirkungsgrad.setLayout(new BorderLayout());
Wirkungsgrad.add(formular6,BorderLayout.CENTER);
container7.setLayout(new FlowLayout(FlowLayout.TRAILING,3,3));
container7.add(cmdNew7);
container7.add(cmdReset7);
container7.add(cmdSave7);
container7.add(cmdSaveAndNew7);
formular7.setLayout(new GridBagLayout());
formular7.add(label30,new GBC(1,1,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
formular7.add(formattedTextField29,new GBC(2,1,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular7.add(label29,new GBC(1,2,1,1,0.0,0.0,GBC.BASELINE_
LEADING,GBC.NONE,new Insets(3,3,3,3),0,0));
```

```
formular7.add(formattedTextField28,new GBC(2,2,1,1,1.0,0.0,GBC.BASEL
INE_LEADING,GBC.HORIZONTAL,new Insets(3,3,3,3),0,0));
formular7.add(container7,new GBC(1,3,2,1,1.0,0.0,GBC.CENTER,GBC.HORI
ZONTAL,new Insets(3,3,3,3),0,0));
GBC.addSpacer(formular7,true,true);
Moment.setLayout(new BorderLayout());
Moment.add(formular7,BorderLayout.CENTER);
AufLeistung.addTab(LeistungAus);
AufLeistung.addTab(Drezahl);
AufLeistung.addTab(EinLeistung);
AufLeistung.addTab(Polparzahl);
AufLeistung.addTab(Schlupf);
AufLeistung.addTab(Wirkungsgrad);
AufLeistung.addTab(Moment);
AufLeistung.setSelectedIndex(5);
this.setLayout(new BorderLayout());
AufLeistung.setPreferredSize(new Dimension(200,200));
this.add(AufLeistung,BorderLayout.NORTH);

this.addWindowListener(new WindowAdapter()
{
 @Override
 public void windowClosing(WindowEvent arg0)
 {
  this_windowClosing(arg0);
 }
});
cmdNew.addActionListener(arg0 -> cmdNew_actionPerformed(arg0));
cmdReset.addActionListener(arg0 -> cmdReset_actionPerformed(arg0));
cmdSave.addActionListener(arg0 -> cmdSave_actionPerformed(arg0));
cmdSaveAndNew.addActionListener(arg0 -> cmdSaveAndNew_
actionPerformed(arg0));
cmdNew2.addActionListener(arg0 -> cmdNew2_actionPerformed(arg0));
cmdReset2.addActionListener(arg0 -> cmdReset2_
actionPerformed(arg0));
cmdSave2.addActionListener(arg0 -> cmdSave2_actionPerformed(arg0));
cmdSaveAndNew2.addActionListener(arg0 -> cmdSaveAndNew2_
actionPerformed(arg0));
cmdNew3.addActionListener(arg0 -> cmdNew3_actionPerformed(arg0));
cmdReset3.addActionListener(arg0 -> cmdReset3_
actionPerformed(arg0));
cmdSave3.addActionListener(arg0 -> cmdSave3_actionPerformed(arg0));
cmdSaveAndNew3.addActionListener(arg0 -> cmdSaveAndNew3_
actionPerformed(arg0));
cmdNew4.addActionListener(arg0 -> cmdNew4_actionPerformed(arg0));
```

```
  cmdReset4.addActionListener(arg0 -> cmdReset4_
  actionPerformed(arg0));
  cmdSave4.addActionListener(arg0 -> cmdSave4_actionPerformed(arg0));
  cmdSaveAndNew4.addActionListener(arg0 -> cmdSaveAndNew4_
  actionPerformed(arg0));
  cmdNew5.addActionListener(arg0 -> cmdNew5_actionPerformed(arg0));
  cmdReset5.addActionListener(arg0 -> cmdReset5_
  actionPerformed(arg0));
  cmdSave5.addActionListener(arg0 -> cmdSave5_actionPerformed(arg0));
  cmdSaveAndNew5.addActionListener(arg0 -> cmdSaveAndNew5_
  actionPerformed(arg0));
  cmdNew6.addActionListener(arg0 -> cmdNew6_actionPerformed(arg0));
  cmdReset6.addActionListener(arg0 -> cmdReset6_
  actionPerformed(arg0));
  cmdSave6.addActionListener(arg0 -> cmdSave6_actionPerformed(arg0));
  cmdSaveAndNew6.addActionListener(arg0 -> cmdSaveAndNew6_
  actionPerformed(arg0));
  cmdNew7.addActionListener(arg0 -> cmdNew7_actionPerformed(arg0));
  cmdReset7.addActionListener(arg0 -> cmdReset7_
  actionPerformed(arg0));
  cmdSave7.addActionListener(arg0 -> cmdSave7_actionPerformed(arg0));
  cmdSaveAndNew7.addActionListener(arg0 -> cmdSaveAndNew7_
  actionPerformed(arg0));
 }// ${GENERATED-CODE-BLOCK-END:INITIALIZERS}
}
```

Listing 6.2 zeigt die Implementierung der Entwicklung eines Entity-Relation-Diagramms, genannt ER-Diagramm, in der Klasse ER_drehzahldrehmomentkennlinieverwaltung, welche die Funktionalität der Verbindungen zwischen Tabellen, genannt *VirtuelleTabellen,* der Klasse EntityRelationshipModel implementiert. Die importierte EntityRelationshipModel gemäß Listing 6.2 implementiert das Interface *StaticInstanceSupport.* Diese Implementierung stellt die Relationen zwischen den Tabellen dar. Wobei die Funktionalitäten des Er-Diagramms sowohl das Dokumentieren des Datenmodells als auch das Auslesen der Verbindungen zum automatischen Generieren der Abfragen ermöglichen. Gemäß Listing 6.2 verfügt die Klasse ER_drehzahldrehmomentkennlinieverwaltung über sieben importierte Tabellen oder virtuelle Tabellen, welche mithilfe von der Datenbank MySQL entwickelt wurden: Aufgenommeneleistung, Drehzahl, Drehzahldrehmomentkennlinie, Eingespeisteleistung, Inneresmoment, Polpaarzahl, Schlupf und Wirkungsgrad. Mithilfe des ER-Diagramms werden die Datenmodelle gemäß Listing 6.2 visualisiert.

Gemäß Listing 6.2 liegt die Funktionalität der Verbindungen zwischen den Tabellen in der Methode *add(),* die aus sechs Parametern zum Aufrufen der Methode *getName()* besteht. Hierbei wird der Zugriff gemäß Listing 6.2 auf den Aufzählungstyp *ONE* mit ihrem Enumerator, genannt *Cardinalty,* ermöglicht. Beispielsweise stellt der Zugriff

auf die Konstante *ONE* ein Parameter der Methode *add()* dar. Wobei der Enumerator Cardinality über Aufzählungstype wie z. B. *ONE* oder *MANY verfügt. Cardinality* spezifiziert die eventuelle Kardinalität von Entitäten in der Klasse EntityRelationship.

Aufrufe der Methode *getName()* in der Methode *add()* zeigen, dass die Tabellen miteinander verknüpft wurden. Beispielsweise gemäß Listing 6.2 wurden die Tabellen Aufgenommeneleistung und Drehzahldrehmomentkennlinie miteinander per Drag-und-Drop verknüpft. Hierbei wurde die Kardinalität 1:n bestätigt [1].

Listing 6.2 Implementierung der Entwicklung eines Entity Relation-Diagrammes in der Klasse ER_drehzahldrehmomentkennlinieverwaltung

```
package Datenquellen;
import xdev.lang.StaticInstanceSupport;
import xdev.vt.Cardinality;
import xdev.vt.EntityRelationshipModel;
import VirtuelleTabellen.Aufgenommeneleistung;
import VirtuelleTabellen.Drehzahl;
import VirtuelleTabellen.Drehzahldrehmomentkennlinie;
import VirtuelleTabellen.Eingespeisteleistung;
import VirtuelleTabellen.Inneresmoment;
import VirtuelleTabellen.Polpaarzahl;
import VirtuelleTabellen.Schlupf;
import VirtuelleTabellen.Wirkungsgrad;
public class ER_drehzahldrehmomentkennlinieverwaltung
extends EntityRelationshipModel implements StaticInstanceSupport
// ${GENERATED-CODE-LINE:ER_SUPERCLASS}
{

 // Generated code, do not edit! ${GENERATED-CODE-BLOCK-START:ER_
 MODEL}
 private static ER_drehzahldrehmomentkennlinieverwaltung
 instance = null;

 public static ER_drehzahldrehmomentkennlinieverwaltung getInstance()
 {
  if(instance == null)
  {
   instance = new ER_drehzahldrehmomentkennlinieverwaltung();
  }
  return instance;
 }
```

```
 {
  add(Aufgenommeneleistung.class.getName(),new
String[]{Aufgenommeneleistung.Id.getName()},
    Cardinality.ONE,Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.AufgenommeneLeistung_
    id.getName()},
    Cardinality.ONE);
  add(Polpaarzahl.class.getName(),new String[]{Polpaarzahl.Id.getName(
  )},Cardinality.ONE,
    Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.Polpaarzahl_id.getName()}
    ,Cardinality.ONE);
  add(Drehzahl.class.getName(),new String[]{Drehzahl.Id.getName()},Car
  dinality.ONE,
    Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.Drehzahl_id.getName()},Ca
    rdinality.ONE);
  add(Inneresmoment.class.getName(),new String[]{Inneresmoment.Id.getN
  ame()},Cardinality.ONE,
    Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.InneresMoment_
    id.getName()},
    Cardinality.ONE);
  add(Wirkungsgrad.class.getName(),new String[]{Wirkungsgrad.Id.getNam
  e()},Cardinality.ONE,
    Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.Wirkungsgrad_id.getName()
    },Cardinality.ONE);
  add(Eingespeisteleistung.class.getName(),new String[]{Eingespeistele
  istung.Id.getName()},
    Cardinality.ONE,Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.EingespeisteLeistung_
    id.getName()},
    Cardinality.ONE);
  add(Schlupf.class.getName(),new String[]{Schlupf.Id.getName()},Cardi
  nality.ONE,
    Drehzahldrehmomentkennlinie.class.getName(),
    new String[]{Drehzahldrehmomentkennlinie.Schlupf_id.getName()},Car
    dinality.ONE);
 }
 // End generated code ${GENERATED-CODE-BLOCK-END:ER_MODEL}
}
```

6.2 DesignPattern Interface

Mithilfe des Design Patterns Interface werden die Funktionalitäten der Klassen zuerst in einem Interface definiert und implementiert. Anschließend werden mithilfe der verschiedenen Aufrufe der Methoden der implementierten Klassen diese implementierten Funktionalitäten dargestellt. Wobei die Anwendungen des Design Patterns Interface zum Trennen von Code aus dem Interface und anderen Klassen in den Funktionalitäten der objektorientierten Programmierung mithilfe der Verwendungen der Objekte das Ziel dieses Entwurfmusters ermöglichen. Hierbei werden mithilfe der Verwendung des Interfaces die Trennung zwischen den Funktionalitäten und deren Implementierungen ermöglicht.

Außerdem werden die Funktionalitäten des Interfaces zum Darstellen der Verbindungen zwischen Interface und deren implementierten Klassen in der Kapselung von Objekten programmiert. Hierbei entstehen Austauche der Nachrichten zwischen dem Interface und den Klassen zum Implementieren der Funktionalitäten des Interfaces.

Mithilfe der Tools von Java 8 werden die Methoden der Interfaces mithilfe von Schlüsselwörter *default* oder *static* zum Implementieren der Funktionalitäten dieser Interfaces deklariert. Listing 6.3, 6.4 und 6.5 zeigen die Anwendung des Interfaces in der Analyse der Berechnungen der Verluste von Insulate Gate Bipolar Transistor (IGBT). Mithilfe der Implementierung der Funktionalitäten in dem Interface zum Analysieren des Schaltverhaltens des Transistors IGBT werden verschiedenen Berechnungen direkt in dem Interface Schaltverlust durchgeführt. Sowohl das Ein- als auch das Ausschalten werden mithilfe der Programmierung mit Java 8 im Interface zum Berechnen der Verluste aus den Verlustarbeiten analysiert. Die Klasse *Analyse4Verlust* gemäß Listing 6.4 implementiert die Funktionalitäten zum Berechnen der Verluste des Interfaces *Schaltverlust* mithilfe der Überladungen der implementierten Methoden des Interfaces in den Methoden berechnung4Einschalten() und berechnung4Ausschalten(). Hierbei wird der Zugang zu den Objekten des Interfaces Schaltverlust nur mithilfe der Implementierung dieses Interfaces möglicht.

Die Aufrufe der Methoden berechnung4Einschalten() und berechnung4Ausschalten() in der Hauptklasse *Analyse4Transistor* mithilfe des Zugriffes auf das Objekt *a4v* der Klasse *Analyse4Verlust* stellen gemäß Listing 6.5 die Kapselung der Funktionalitäten des Interfaces *Schaltverlust* mithilfe deren Implementierungen in dieser Klasse dar. Dies bedeutet, dass es eine Kommunikation zwischen dem Interface *Schaltverlust* und der Klasse *Analyse4Verlust* zum Berechnen der Verluste des Transistors IGBT während sowohl des Ein- als auch des Ausschaltvorganges gibt. Listing 6.6 zeigt die Konsole zum Darstellen der Berechnungen der Verluste des Transistors IGBT. Gemäß Listing 6.6 werden die Berechnungen während des Ein- und Ausschaltvorganges des Transistors IGBT dargestellt.

Ziel ist es, die Verluste des Transistors IGBT während des Ein- und Ausschaltvorganges mithilfe der Anwendungen der Funktionalitäten des Interfaces *Schaltverlust* in dessen implementierten Klasse Analyse4Verlust zu analysieren.

Listing 6.3 Darstellung des Interfaces *Schaltverlust* Zum Charakterisieren der Analyse der Verluste des Transistors IGBT

```
package verluste4motor;
public interface Schaltverlust
{
 double ausschaltverlustleistung = 1200.0;
 double einschaltverlustleistung = 800.0;
 double EIN_verluste = 25.0;
 double AUS_verluste = 20.0;
 double schaltfrequenz = 400.0;
 double einschaltverhaeltnis = 0.5;
 double ausschaltverhaeltnis = 0.5;
 double einschaltzeit = 0.8/1000;
 double ausschaltzeit = 0.3/1000;

 default public double einschaltperiode() {
  return einschaltverhaeltnis/schaltfrequenz;
 }

 default public double ausschaltperiode() {
  return ausschaltverhaeltnis / schaltfrequenz;
 }

 default public double EIN_verlustleistung() {
  //TODO Auto-generated method stub
  return EIN_verlustarbeit()/einschaltperiode();
 }

 default public double EIN_verlustarbeit() {
  //TODO Auto-generated method stub
  return EIN_verluste*einschaltperiode();
 }

 default public double einschaltverlustarbeit() {
  //TODO Auto-generated method stub
  return einschaltverlustleistung * einschaltzeit;
 }

 default public double ausschaltverlustarbeit() {
  return ausschaltverlustleistung * ausschaltzeit;
 }
```

```
 default public double AUS_verlustarbeit() {
  //TODO Auto-generated method stub
  return AUS_verluste*ausschaltperiode();
 }

 default public double AUS_verlustleistung() {
  //TODO Auto-generated method stub
  return AUS_verlustarbeit()/ausschaltperiode();
 }

 }
```

Listing 6.4 Implementierung des Interfaces *Schaltverlust* in der Klasse *Analyse4Verlust*

```
package verluste4motor;
public class Analyse4Verlust implements Schaltverlust{
 public void Eingabe4berechnung() {

  System.out.println("Eingaben für den Transistor:");

  System.out.println(("Einschaltverlustleistung ist: " + einschaltver-
  lustleistung + " Watt"));
  System.out.println(("Ausschaltverlustleistung ist: " + ausschaltver-
  lustleistung + " Watt"));
  System.out.println(("Verluste beim Einschalten des Transistors ist :
  " + EIN_verluste + " Watt"));
  System.out.println(("Verluste beim Ein des Transistors ist: " + AUS_
  verluste+ " Watt"));

  System.out.println(("Schaltfrequenz des Transistors ist: " + schalt-
  frequenz+ " 1/s"));
  System.out.println(("Einschaltverhaeltnis   des   Transistors   ist:
  " + einschaltverhaeltnis));
  System.out.println(("Ausschaltverhaeltnis   des   Transistors   ist:
  " + ausschaltverhaeltnis));

  System.out.println("-----------------------------------------
  -------------------------------");

 }

 public void berechnung4Einschalten()
```

```
 {

  System.out.println("Berechnungen beim Einschalten");
  System.out.println(("Einschaltzeit ist: " + einschaltzeit+ " s"));
  System.out.println(("Einschaltperiode ist: " + einschaltperiode() +
  " s"));

  System.out.println("Verlustarbeit beim Einschalten ist:"+ einschalt-
  verlustarbeit() + " W.s");
  System.out.println("Verlustarbeit beim Ein ist: "+ EIN_verlust-
  arbeit() + " W.s");

  System.out.println("Verlustleistung beim Einschalten ist:"+ EIN_ver-
  lustleistung() + " Watt");

  System.out.println("-----------------------------------------
  ----------------------------");

 }

 public void berechnung4Ausschalten()

 { System.out.println("Berechnungen beim Ausschalten");
  System.out.println(("Ausschaltzeit ist: " + ausschaltzeit+ " s"));

  System.out.println(("Ausschaltperiode ist: " + ausschaltperiode() +
  " s"));
  System.out.println("Verlustearbeit beim Ausschalten "+ ausschaltver-
  lustarbeit());
  System.out.println("Verlustarbeit beim Aus ist:"+ AUS_verlust-
  arbeit() + " W.s");
  System.out.println(("Verluste beim Ausschalten des Transistors ist:
  " + AUS_verluste + " Watt"));

  System.out.println("Verlustleistung beim Ausschalten ist:”+ AUS_ver-
  lustleistung() + " Watt");

 }

}
```

Listing 6.5 Darstellung der Hauptklasse *Analyse4Transistor* zum Aufrufen der Methoden der implemetierten Klasse *Analyse4Verlust*

```
package verluste4motor;
public class Analyse4Transistor {
 public static void main(String[] args) {

  //TODO Auto-generated method stub´

  Analyse4Verlust a4v = new Analyse4Verlust();
  a4v.Eingabe4berechnung();
  a4v. berechnung4Einschalten();
  a4v.berechnung4Ausschalten();
 }
}
```

Listing 6.6 Darstellung der Konsole zum Implementieren der Funktionalitäten des Interfaces *Schaltverlust*

```
Eingaben für den Transistor:
Einschaltverlustleistung ist: 800.0 W
Ausschaltverlustleistung ist: 1200.0 W
Verluste beim Einschalten des Transistors ist: 25.0 W
Verluste beim Ein des Transistors ist: 20.0 W
Schaltfrequenz des Transistors ist: 400.0 1/s
Einschaltverhaeltnis des Transistors ist: 0.5
Ausschaltverhaeltnis des Transistors ist: 0.5
------------------------------------------------------------------------
Berechnungen beim Einschalten
Einschaltzeit ist: 8.0E-4 s
Einschaltperiode ist: 0.00125 s
Verlustarbeit beim Einschalten ist:0.64 W.s
Verlustarbeit beim Ein ist: 0.03125 W.s
Verlustleistung beim Einschalten ist:25.0 W
------------------------------------------------------------------------
Berechnungen beim Ausschalten
Ausschaltzeit ist: 3.0E-4 s
Ausschaltperiode ist: 0.00125 s
Verlustearbeit beim Ausschalten 0.36
Verlustarbeit beim Aus ist:0.025 W.s
Verluste beim Ausschalten des Transistors ist: 20.0 W
Verlustleistung beim Ausschalten ist:20.0 W
```

6.3 Codegenerierung aus Ecore-Modellen

Aus einem Ecore-Modell lassen sich EMF-Codefragmente generieren [4, 5]. Hierbei werden für jedes EClass-Element ein Java-Interface sowie eine implementierende Klasse und für jedes Attribut eine set- und eine get-Methode erzeugt. Das für die Erzeugung zuständige enthält Zusatzinformationen für den Generator, die im Ecore-Modell nicht vorliegen, zum Beispiel in welches Verzeichnis der Code zu stellen ist. Der Codegenerator ist aus dem Kontextmenü des Generatormodells aufzurufen.

Das Plugin *„org.eclipse.emf.ecore“* ermöglicht die Manipulation von Modellen und Instanzen. Beispielsweise verfügt die Klasse *EObject* über reflektierende Methoden u. a.: *eClass(),eContents(), eGet(EStructuralFeature feature), eSet(EStructuralFeature feature, Object value).* Die Methode *eclass()* gibt die Klasse zurück, *eContents()* besteht aus der Liste der enthaltenen Objekte, *eGet(EStructuralFeature feature)* gibt den Wert des angegebenen Feature zurück und *eSet(EStructuralFeature feature, Object value)* setzt den Wert des angegebenen Features. Die Methode *getEStructuralFeatures()* gehört zu der Klasse EClass und verfügt über eine Liste aller enthaltenen Attribute und Referenzen. Die Klasse *EReference* besteht aus den Methoden: *getEOpposite()* und *getUpperBound().* Die erste stellt eine entgegengesetzte Referenz dar, und die zweite gibt die obere Schranke zurück [5].

Listing 6.7 zeigt die Codegenerierung mit Java aus einem Ecore-Modell der Implementierung des Interfaces *Anwendung4Interface* in der Klasse *Anwendung4InterfaceImp*l, welche Obereklasse von *Analyse4KollisionImpl* ist. Wobei *Analyse4KollisionImp*l die Funktionalitäten des Interfaces *Anwendung4Interface* implementiert. Gemäß Listing 6.7 stellt die Klasse *Anwendung4InterfaceImp*l die konkrete Implementierung der im Ecore-Modell definierten Interfaces dar. Hierbei verfügt die Klasse *Anwendung4InterfaceImpl* getter- und setter-Methoden wie z. B. *getErmitteln4Drehzahlen()* und setErmitteln4Drehzahlen*(Anwendungs4Berechnung newErmitteln4Drehzahlen).* Einerseits gibt die erste Methode mithilfe der Kapselung in Java den Wert vorher als protected deklariert *ermitteln4Drehzahlen* zurück. Andererseits ermöglicht die zweite Methode den Zugriff auf Objekte der Klasse *Anwendung4Berechnung.* Wobei diese setter-Methode generierte Anzeigen über die Beobachtungen des Modells darstellt. Die Methoden setErmitteln4Drehzahlen *(Anwendungs4Berechnung newErmitteln4Drehzahlen)* und setErmitteln4Leistungen *(Anwendungs4Berechnung newErmitteln4Leistungen)* setzen Objekt-Zeiger, damit die Notifizierung über Änderungen mithilfe des Aufrufes der Methode eNotify() jedem Beobachter der Objekte *gesendet wird.* Hierbei wird mithilfe der eNotificationRequired() geprüft, ob es einen Beobachter des Objektes gibt. Bei den Methoden getErmitteln4Drehzahlen() und getErmitteln4Leistungen() werden zuerst die Methode eIsProxy() zum Prüfen, ob die Referenz ein Stellvertretter ist, aufgerufen. Anschließend wird es mithilfe des Aufrufes der Methode eResolveProxy() bestätigt, ob es um einen Stellvertreter handelt. Die letzte Methode ermöglicht den Aufruf der Methode EcoreUtil.resolve() zum Zugriff auf Objekte des Stellvertreters. Mithilfe

der *if*-Schleife werden die Funktionalitäten der aufgelösten Objekte in den Methoden eNotificationRequired() und *eNotiify()* analysiert und schließlich zum Rückgeben der Objekte nämlich *ermitteln4Drehzahlen* oder *ermitteln4Leistungen* ermöglicht.

Gemäß Listing 6.7 werden mithilfe der Methoden *eSet()* und *eGet()* die Methoden *setErmitteln4Drehzahlen()* und *setErmitteln4Leistungen()* bzw. *getErmitteln4Drehzahlen()* und *getErmitteln4Leistungen()* überladen. Die Methode *eGet()* besteht aus drei Parametern: featureID, resolver und coreType. Zum einen ist der erste von dem Datentyp int und zum anderen sind der zweite und der dritte vom Datentyp boolean. Während die Methode *eSet()* aus zwei Parametern besteht: featureId und *newValue* von den Datentypen int bzw. *object.* Die Methode *eUnset()* mit dem Parameter f*eatureID* stellt wie *eSet()* die Überladungen der Methoden setErmitteln4Drehzahlen() und setErmitteln4Leistungen() dar. Dies ermöglicht den Zugriff auf Elemente von *Anwendungs4Berechnung.* Aufrufe von *eSet()* und *eUnset()* erfolgen auf *super()* zum Setzen von den Parametern *newValu*e bzw. *null.* Wobei *eSet()* und *eUnset()* gemäß Listing 6.7 die Werte der angegebenen *Features* setzen. Gemäß Listing 6.7 ermöglicht die Methode *eStaticClass()* vom Datentyp *EClass* den Zugriff auf Elemente des Interfaces *Anwendung4Interface.*

Gemäß Listing 6.8 ermöglicht die Implementierung der Funktionalität der main-Methode mithilfe des Aufrufes der Methode *eInvoke()* auf den Konstruktor *super().* Ebenfalls im Listing 6.7 werden die Überladungen der Methode *ermitteln4Drehzahlen()* und ermitteln4Leistungen() mithilfe der Methode *eInvoke()* ermöglicht. Die Methode *eInvoke()* verfügt über zwei Parameter, nämlich *operationID* und *argument*s von Datentypen *int* bzw. *EList*<?>.

Listing 6.7 Darstellung der generierten Klasse *Anwendung4InterfaceImpl* aus dem Ecore-Modell

```
import funktion4modell.Anwendung4Interface;
import funktion4modell.Anwendungs4Berechnung;
import funktion4modell.Funktion4modellPackage;
import java.lang.reflect.InvocationTargetException;
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.InternalEObject;
import org.eclipse.emf.ecore.impl.ENotificationImpl;
public class Anwendung4InterfaceImpl extends Analyse4KollisionImpl
implements Anwendung4Interface {
 protected Anwendungs4Berechnung ermitteln4Drehzahlen;

 protected Anwendungs4Berechnung ermitteln4Leistungen;

 protected Anwendung4InterfaceImpl() {
  super();
 }
```

```
 @Override
 protected EClass eStaticClass() {
  return Funktion4modellPackage.Literals.ANWENDUNG4_INTERFACE;
}
 public Anwendungs4Berechnung getErmitteln4Drehzahlen() {
  if (ermitteln4Drehzahlen != null && ermitteln4Drehzahlen.
  eIsProxy()) {
   InternalEObject    oldErmitteln4Drehzahlen    =    (InternalEObject)
   ermitteln4Drehzahlen;
   ermitteln4Drehzahlen = (Anwendungs4Berechnung)
eResolveProxy(oldErmitteln4Drehzahlen);
   if (ermitteln4Drehzahlen != oldErmitteln4Drehzahlen) {
    if (eNotificationRequired())
     eNotify(new ENotificationImpl(this, Notification.RESOLVE,

       Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_DREH-
       ZAHLEN, oldErmitteln4Drehzahlen,
       ermitteln4Drehzahlen));
   }
  }
  return ermitteln4Drehzahlen;
 }

 public Anwendungs4Berechnung basicGetErmitteln4Drehzahlen() {
  return ermitteln4Drehzahlen;
 }
 public void setErmitteln4Drehzahlen(Anwendungs4Berechnung
 newErmitteln4Drehzahlen) {
  Anwendungs4Berechnung oldErmitteln4Drehzahlen = ermitteln4Drehzahlen;
  ermitteln4Drehzahlen = newErmitteln4Drehzahlen;
  if (eNotificationRequired())
   eNotify(new ENotificationImpl(this, Notification.SET,
     Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_DREH-
     ZAHLEN, oldErmitteln4Drehzahlen,
     ermitteln4Drehzahlen));
 }
 public Anwendungs4Berechnung getErmitteln4Leistungen() {
  if (ermitteln4Leistungen != null && ermitteln4Leistungen.
  eIsProxy()) {
   InternalEObject oldErmitteln4Leistungen = (InternalEObject)
   ermitteln4Leistungen;
   ermitteln4Leistungen = (Anwendungs4Berechnung)
eResolveProxy(oldErmitteln4Leistungen);
   if (ermitteln4Leistungen != oldErmitteln4Leistungen) {
    if (eNotificationRequired())
     eNotify(new ENotificationImpl(this, Notification.RESOLVE,
```

```
        Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_
        LEISTUNGEN, oldErmitteln4Leistungen,
        ermitteln4Leistungen));
    }
   }
   return ermitteln4Leistungen;
  }
  public Anwendungs4Berechnung basicGetErmitteln4Leistungen() {
   return ermitteln4Leistungen;
  }

  public void setErmitteln4Leistungen(Anwendungs4Berechnung
  newErmitteln4Leistungen) {
   Anwendungs4Berechnung oldErmitteln4Leistungen = ermitteln4Leistungen;
   ermitteln4Leistungen = newErmitteln4Leistungen;
   if (eNotificationRequired())
    eNotify(new ENotificationImpl(this, Notification.SET,

      Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_
      LEISTUNGEN, oldErmitteln4Leistungen,
      ermitteln4Leistungen));
  }

  public void ermitteln4Drehzahlen() {
   //TODO: implement this method
   //Ensure that you remove @generated or mark it @generated NOT
   throw new UnsupportedOperationException();
  }

  public void ermitteln4Leistungen() {
   //TODO: implement this method
   //Ensure that you remove @generated or mark it @generated NOT
   throw new UnsupportedOperationException();
  }

  @Override
  public Object eGet(int featureID, boolean resolve, boolean coreType)
{
   switch (featureID) {
   case   Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_DREH-
   ZAHLEN:
    if (resolve)
     return getErmitteln4Drehzahlen();
    return basicGetErmitteln4Drehzahlen();
```

```
 case Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_
 LEISTUNGEN:
  if (resolve)
   return getErmitteln4Leistungen();
  return basicGetErmitteln4Leistungen();
 }
 return super.eGet(featureID, resolve, coreType);
}

@Override
public void eSet(int featureID, Object newValue) {
 switch (featureID) {
 case   Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_DREH-
 ZAHLEN:
  setErmitteln4Drehzahlen((Anwendungs4Berechnung) newValue);
  return;
 case Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_
 LEISTUNGEN:
  setErmitteln4Leistungen((Anwendungs4Berechnung) newValue);
  return;
 }
 super.eSet(featureID, newValue);
}

@Override
public void eUnset(int featureID) {
 switch (featureID) {
 case Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_DREH-
 ZAHLEN:
  setErmitteln4Drehzahlen((Anwendungs4Berechnung) null);
  return;
 case Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_
 LEISTUNGEN:
  setErmitteln4Leistungen((Anwendungs4Berechnung) null);
  return;
 }
 super.eUnset(featureID);
}

@Override
public boolean eIsSet(int featureID) {
 switch (featureID) {
 case   Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_DREH-
 ZAHLEN:
  return ermitteln4Drehzahlen != null;
```

```
  case Funktion4modellPackage.ANWENDUNG4_INTERFACE__ERMITTELN4_
  LEISTUNGEN:
   return ermitteln4Leistungen != null;
  }
  return super.eIsSet(featureID);
 }
 @Override
 public Object eInvoke(int operationID, EList<?> arguments) throws
 InvocationTargetException {
  switch (operationID) {
  case  Funktion4modellPackage.ANWENDUNG4_INTERFACE___ERMITTELN4_DREH-
  ZAHLEN:
   ermitteln4Drehzahlen();
   return null;
  case Funktion4modellPackage.ANWENDUNG4_INTERFACE___ERMITTELN4_
  LEISTUNGEN:
   ermitteln4Leistungen();
   return null;
  }
  return super.eInvoke(operationID, arguments);
  }
} //Anwendung4InterfaceImpl
```

Listing 6.8 Darstellung der generierten Klasse Anwendungs4BerechnungImpl aus dem Ecore-Modell

```
package funktion4modell.impl;
import funktion4modell.Anwendungs4Berechnung;
import funktion4modell.Funktion4modellPackage;
import java.lang.reflect.InvocationTargetException;
import org.eclipse.emf.common.util.EList;
import org.eclipse.emf.ecore.EClass;
import org.eclipse.emf.ecore.impl.MinimalEObjectImpl;
public class Anwendungs4BerechnungImpl extends MinimalEObjectImpl.
Container implements Anwendungs4Berechnung {
 protected Anwendungs4BerechnungImpl() {
  super();
 }

 @Override
 protected EClass eStaticClass() {
  return Funktion4modellPackage.Literals.ANWENDUNGS4_BERECHNUNG;
 }
```

```
 public void main() {
  //TODO: implement this method
  //Ensure that you remove @generated or mark it @generated NOT
  throw new UnsupportedOperationException();
 }

 @Override
 public Object eInvoke(int operationID, EList<?> arguments) throws
 InvocationTargetException {
  switch (operationID) {
  case Funktion4modellPackage.ANWENDUNGS4_BERECHNUNG___MAIN:
   main();
   return null;
  }
  return super.eInvoke(operationID, arguments);
 }
} //Anwendungs4BerechnungImpl
```

6.4 Zusammenfassung

XDEV ist eine visuelle Java-Entwicklungsumgebung für Rapid Application Development (RAD). Mit der Entwicklungsumgebung XDEV lassen sich professionelle Web- und Desktopapplikationen auf Basis von Java entwickeln. Die Implementierung von XdevWindow bezüglich des MVC-Patterns fokussiert auch auf verschiedene Tools u. a. ActionEvent, EventHandlerDelegate, Event Model, Event Sources und EventListeners, Schnittstellen wie z. B. FormularAdapter und WindowAdapter. Das Erzeugen eines Fenster erfolgt mithilfe von GUI-Dialogelementen wie z. B. LayoutManager. Eine Implementierung der Entwicklung eines Entity-Relation-Diagramms, genannt ER-Diagramm, in einer Klasse ermöglicht die Relationen zwischen verschiedenen Tabellen.

Anwendungen des Design Patterns Interface zum Trennen von Code aus dem Interface und anderen Klassen in den Funktionalitäten der objektorientierten Programmierung stellen mithilfe der Verwendungen der Objekte das Ziel dieses Entwurfmusters dar. Hierbei werden mithilfe der Verwendung des Interfaces die Trennung zwischen den Funktionalitäten und deren Implementierungen ermöglicht. Außerdem werden die Funktionalitäten des Interfaces zum Darstellen der Verbindungen zwischen Interface und deren implementierten Klassen in der Kapselung von Objekten programmiert. Hierbei entstehen Austausche der Nachrichten zwischen dem Interface und den Klassen zum Implementieren der Funktionalitäten des Interfaces. Aufrufe implementierter Methoden in einer Hauptklasse mithilfe des Zugriffes auf Objekte dieser Klasse stellen die Kapselung der Funktionalitäten eines Interfaces mithilfe deren Implementierungen in dieser Klasse dar.

Aus einem Ecore-Modell lassen sich EMF-Codefragmente generieren. Hierbei werden für jedes EClass-Element ein Java-Interface sowie eine implementierende Klasse und für jedes Attribut eine set- und eine get-Methode erzeugt. Das für die Erzeugung Zuständige enthält Zusatzinformationen für den Generator, die im Ecore-Modell nicht vorliegen, zum Beispiel in welches Verzeichnis der Code zu stellen ist. Der Codegenerator ist aus dem Kontextmenü des Generatormodells aufzurufen.

Das Plugin „org.eclipse.emf.ecore“ ermöglicht die Manipulation von Modellen und Instanzen. Beispielsweise verfügt die Klasse *EObject* über reflektierende Methoden u. a.: *eClass(), eContents(), eGet(EStructuralFeature feature), eSet(EStructuralFeature feature, Object value).*

Literatur

1. Nyamsi, E., A.: Realisierung der Einsparpotentiale bei elektrischen Energieverbrauchern, Springer Vieweg Verlag (2018).
2. Czeschla, J.: Was versteht man unter Model View Controller(MVC)? In: Design Patterns, javabeginners Web Portal, https://javabeginners.de/Design_Patterns/Model-View-Controller.php (2016).
3. Krüger, G., Hansen, H.: Java Programmierung, das Handbuch zu Java 8, S. 1–1079, Addison-Wesley Verlag, Boston, USAN (2014).
4. Pohl, T.: Codegenerierung von EMF-Fragmenten, In: Modellgetriebene Softwareentwicklung mit EMF und Xtext, heise online, heise Developer, https://www.heise.de/developer/artikel/Codegenerierung-von-EMF-Fragmenten-912495.html (2010).
5. Taentzer, G.: Dynamisches EMF und codegenerierung mit JET, In: Modellgetriebene Software Entwicklung, Online, http://www.mathematik.uni-marburg.de/~swt/ws12/mdd/files/Folien121114.pdf, Uni.Marburg (2012).

Stichwortverzeichnis

E. A. Nyamsi, *IT-Lösungen auf Basis von SysML und UML,*
https://doi.org/10.1007/978-3-658-29057-3