Pattern-Oriented Transformations between Analysis and Design Models (POTAD)

DISSERTATIONSSCHRIFT

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technische Universität Ilmenau

von Diplom-Wirtschaftsinformatiker Gabriel Schwefer (geb. Vögler)
geboren am 18. September 1976

Gutachter: 1. Prof. Dr.-Ing. habil. Ilka Philippow

2. Prof. Dr.-Ing. Ina Schieferdecker

3. Prof. Dr.-Ing. habil. Wolfgang Fengler

Eingereicht: 10. Oktober 2007
Datum der Promotion: 19. Februar 2008

urn:nbn:de:gbv:ilm1-2008000021

Vorwort

Diese Arbeit entstand im Rahmen meiner Doktorandentätigkeit in der Abteilung *Software-Architekturen* (mittlerweile *Software-Strukturen*) der DaimlerChrysler-Forschung in Ulm, die von Dr. Bernhard Hohlfeld geleitet wird. Das bearbeitete Thema war in das vom BMBF geförderten Verbundprojekt *Integrative Pattern- und UML-orientierte Lern- und System-Entwicklungsumgebung* (InPULSE) eingebettet. An diesem Projekt war neben der DaimlerChrysler AG unter anderem auch das Fachgebiet *Prozessinformatik* der TU Ilmenau beteiligt, das von Frau Prof. Dr.-Ing. habil. Ilka Philippow geleitet wird. Sie übernahm die wissenschaftliche Betreuung dieser Arbeit und ergänzte somit Dr. Thomas Flor, der die Arbeit seitens DaimlerChrysler begleitete.

Für die Betreuung möchte ich Ilka Philippow und Thomas Flor herzlich danken. Sehr hilfreich waren unter anderem die Ratschläge zum methodischen Aufbau und die kritischen Fragen zur Eingrenzung des Themas. Persönlich und auch stellvertretend für die DaimlerChrysler AG möchte ich Herrn Hohlfeld für die Unterstützung dieser Arbeit und meiner beruflichen Entwicklung danken. Die Finanzierung von Konferenzbesuchen und die Genehmigung von Diplomandenstellen im Umfeld dieser Arbeit haben zu dem Gelingen entscheidend beigetragen. Ein Kompliment auch an die Diplomanden, die sich mit viel Interesse in ein umfangreiches Gebiet eingearbeitet haben und sich mit Engagement an der Fallstudie und der Werkzeugimplementierung beteiligten. Ein persönlicher Dank an diejenigen, die mich trotz meiner wissenschaftlichen Tätigkeiten nicht vergessen haben und nun insgesamt fünf Jahre Geduld und Rücksicht aufgebracht haben. Bei der wichtigsten Person in meinem Leben möchte ich mich dafür bedanken, dass sie da ist und ihr, Stella, diese Arbeit widmen.

Kurzfassung

Eine Antwort auf viele aktuelle Anforderungen im Elektrik/Elektronik-Bereich der Fahrzeugentwicklung ist ein durchgängig modellbasierter Entwicklungsprozess, der Modelle der System- und Softwareentwicklung integriert. Ein Systemmodell beschreibt mit der logischen Systemarchitektur die Funktionen eines Fahrzeugs und mit der technischen Systemarchitektur die realisierende Elektrik/Elektronik, wie z. B. Steuergeräte, Sensoren/Aktoren und Bussysteme. Im Rahmen der Softwareentwicklung muss für einzelne Funktionen aus der logischen Systemarchitektur unter Berücksichtigung der technischen Systemarchitektur und weiterer Anforderungen ein Softwaredesignmodell erstellt werden. Aktuelle modellbasierte Entwicklungsansätze versprechen mit Hilfe des Konzepts der Modelltransformation den Übergang zwischen Modellen unterschiedlicher Entwicklungsphasen automatisieren zu können. Dieses Konzept bietet sich dazu an, aus einem Systemarchitekturmodell ein Grundgerüst eines Softwaredesignmodells zu erzeugen und damit einen Teil der Softwareentwicklungsaktivitäten zu automatisieren.

Die Analyse dieser Arbeit zeigt, dass die erarbeiteten domänenspezifischen Anforderungen, die für solch ein Szenario an einen Modelltransformationsmechanismus gestellt werden müssen, durch aktuelle Ansätze nicht vollständig erfüllt werden. Der eigene Ansatz *Pattern-Oriented Transformations between Analysis and Designmodels* (POTAD) verwendet die logische Systemarchitektur im Rahmen der Softwareentwicklung als Analysemodell und systematisiert dessen Zusammenhang mit dem Designmodell auf der Basis von Analyse- und Designmustern. Für ein im Analysemodell gefundenes Analysemuster instanziiert eine POTAD-Transformationsregel mit Hilfe dieser Systematik in Abhängigkeit nichtfunktionaler Anforderungen und der technischen Systemarchitektur unterschiedliche Designmuster im Designmodell. Gleichzeitig werden Verknüpfungen zwischen den Analyse- und Designmustern angelegt, die zur späteren Verfolgung von Designentscheidungen genutzt werden. Anhand eines dem POTAD-Entwicklungsprozess folgenden Prototyps, der die in der POTAD-Transformationssprache formulierten Regeln ausführen kann und die Verfolgbarkeit werkzeugseitig unterstützt, wird die Realisierbarkeit des Lösungsansatzes gezeigt.

POTAD wurde durch studentische Arbeiten anhand einer Fallstudie überprüft, die typische Eigenschaften der betrachteten Domäne abdeckt. Die Ergebnisse dieser Arbeiten haben die Tauglichkeit von POTAD gezeigt, die Methodik und die Transformationssprache verbessert und Grenzen aufgezeigt.

Abstract

One answer to many current challenges in the electronic domain of automotive development, is a continuous model-based engineering process that integrates models of system and software development. A system model describes by the use of the logical system architecture the functions of a vehicle and through the technical system architecture the realising electronics, such as control units, sensors/actuators and data busses. During software development, a software design model for selected functions of the logical system architecture must be constructed with consideration of the technical architecture and further requirements. Current model-based development approaches claim to automate the transition between different development phases by the concept of model transformations. This concept lends itself to generate a skeleton of the software design model from the system architecture model, thereby automating a part of the software engineering activities.

The analysis of this work shows that the collected domain specific requirements, which must be made on a model transformation mechanism for such a scenario, are not fulfilled by current approaches. The approach taken in this work, the *Pattern-Oriented Transformations between Analysis and Designmodels* (POTAD) uses the system architecture as an analysis model within software development and systemizes the connection with the design model on the basis of analysis and design patterns. By means of this systematisation, a POTAD transformation rule instantiates for an analysis pattern different design patterns under consideration of non-functional requirements and the technical system architecture. At the same time, links between an analysis and design pattern are created, which are used to trace design decision later. The feasibility of the solution is shown by a prototype, which follows the POTAD development process and executes the transformation rules formulated in the POTAD transformation language.

POTAD was verified by several student works based on a case study, which covers typical characteristics of the examined domain. The results of these works showed the suitability and improved the methodology as well as the transformation language and pointed out the limits of the approach taken.

Inhaltsverzeichnis

1 Einleitung und Motivation

Die Problemstellung dieser Arbeit entstand vor dem Hintergrund neuer Anforderungen und Trends im Bereich *Automotive Software*. Aktuelle Herausforderungen liegen hier in der steigenden Komplexität durch die zunehmende Vernetzung von Funktionen, immer höheren Optimierungsanforderungen bzgl. nichtfunktionaler Kriterien und dem zunehmenden Bedarf, eine Softwareimplementierung in Bezug auf Änderungen in der Systemarchitektur schnell und flexibel anzupassen. Die Systemarchitektur wiederum wird im Rahmen neuerer Systementwicklungsansätze zunehmend in formalisierter Weise erfasst und beinhaltet üblicherweise eine systematische Beschreibung der Fahrzeugfunktionen und der realisierenden Elektrik/Elektronik (E/E). Einige dieser Inhalte werden traditionell auch in der Softwareentwicklung formalisiert erfasst, so dass es hier nun zu Überlappungen kommen kann. Vor diesem Hintergrund sind Bestrebungen zu beobachten, die Prozesse der System- und Softwareentwicklung stärker zu integrieren und die an den Schnittstellen ausgetauschten Informationen so zu formalisieren, dass sie durchgängig in möglichst vielen Phasen des Entwicklungszyklus nutzbar sind. Die Ausgangssituation dieser Versuche erscheint dabei günstig, da die Standardisierungsinitiativen im Bereich der Systementwicklung in letzter Zeit Formalisierungstechniken und Beschreibungsmittel nutzen, die im Umfeld der Softwareentwicklung entstanden und etabliert sind.

Die stärkere prozessuale Verzahnung beider Entwicklungsdisziplinen einerseits und die zunehmende Formalisierung der Entwicklungsinformationen auf Basis gemeinsam genutzter Beschreibungsmittel andererseits, wirft die Frage auf, ob sich Software in diesem neuen Umfeld nicht systematischer entwickeln lässt als bisher. Konkret stellt sich folgende Frage: Ist es für einzelne Funktionen einer formalisiert beschriebenen Systemarchitektur möglich, automatisiert einen Teil des Softwaredesigns abzuleiten?

Moderne modellbasierte Ansätze aus dem Umfeld der Softwareentwicklung scheinen in Bezug auf diese Frage eine Lösung anbieten zu können. Für die Integration und Wiederverwendung von formalisierten Informationen benachbarter Fachdisziplinen werden *domänenspezifische* und *plattformunabhängige Modelle* vorgeschlagen, die ein Softwaresystem problemorientiert und designunabhängig beschreiben. Mit dem Konzept der *Modelltransformation* scheint auch die angesprochene Automatisierung bei der Ableitung eines Softwaredesigns realisierbar zu ein. Sie versprechen die *plattformunabhängigen Modelle* weitgehend maschinell in *plattformspezifische Modelle* (beschreiben eine Designlösung) überführen zu können. Die hierbei anzuwendende Systematik spiegelt sich in Form von Transformationsregeln wider.

Das folgende Unterkapitel erläutert das Umfeld der Automotive-Softwareentwicklung etwas genauer und wirft einige Fragen auf, die sich stellen, wenn man diese modellbasierten Ansätze im Kontext von *Automotive Software* für die Integration von System- und Softwareentwicklung nutzen möchte. Diese Betrachtungen führen schließlich im darauf folgenden Unterkapitel zu der Formulierung einer Problemstellung, für die im Rahmen dieser Arbeit ein Lösungsansatz erarbeitet wird.

1.1 Umfeld

Die hier betrachtete *Automotive Software* läuft auf dem Mikrocontroller eines Steuergeräts, das mit Hilfe von angeschlossenen Sensoren, Sollwertgebern und Aktuatoren Steuerungs- und Regelungsfunktionen realisiert. In heutigen Fahrzeugen gibt es bis zu 70 solcher Steuergeräte, von denen die meisten mehrere Funktionen ausüben. Da sich mittlerweile viele Funktionen aus Teilfunktionen unterschiedlicher Steuergeräte zusammensetzen, sind die Steuergeräte zur Deckung des Kommunikationsbedarfs über bis zu fünf Bussysteme miteinander vernetzt, auf denen wiederum ca. 8000 Signale ausgetauscht werden. Ein Beispiel für solch eine verteilte Funktion ist der Tempomat. Dieser versucht das Fahrzeug so zu beschleunigen, dass eine eingestellte Geschwindigkeit erreicht bzw. gehalten wird. Dazu wird der Fahrerwunsch über ein Bedienelement des Mantelrohrschaltermoduls in der Nähe des Lenkrads eingelesen und die aktuelle Geschwindigkeit über das ESP-Steuergerät ermittelt. Sollte sich eine Abweichung zwischen Soll- und Ist-Geschwindigkeit ergeben, kann das Motormoment über das Motorsteuergerät, das Getriebe über das Getriebesteuergerät und das Bremsmoment über das ESP-Steuergerät beeinflusst werden. Abbildung 1.1 zeigt skizzenhaft die Steuergeräte und ihre Vernetzung eines Fahrzeugs.

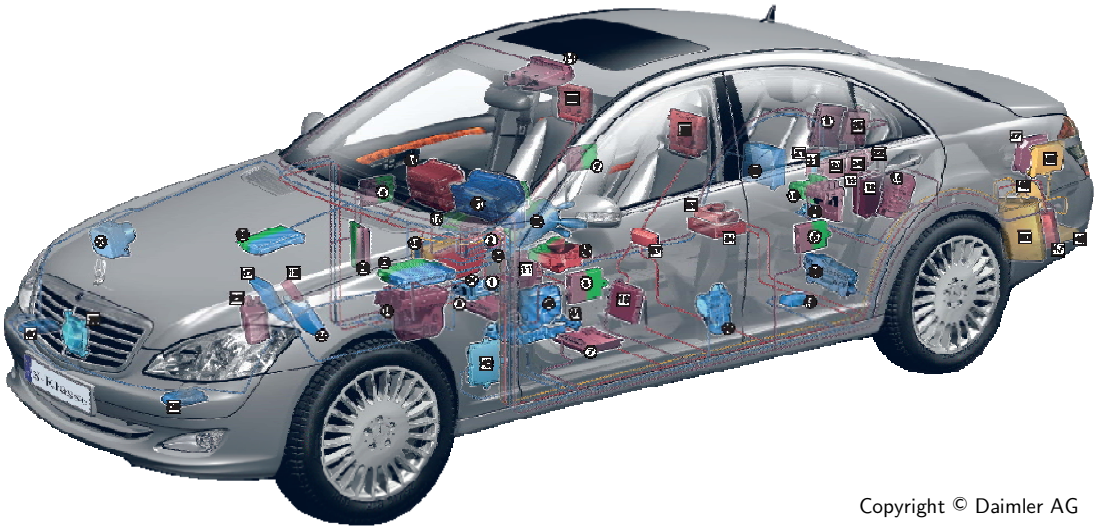


Abbildung 1.1: Vernetzte Steuergeräte in einem Oberklassenfahrzeug

In diesem Umfeld beschäftigt sich ein Softwareprojekt typischerweise mit der Software eines Steuergeräts. Die in der Systemarchitektur erfassten Informationen beantworten für die Softwareentwicklung viele entscheidende Fragen: Welche Funktionen müssen implementiert werden, zu welchen Funktionen gibt es Schnittstellen, wie ist der Aufbau des verwendeten Steuergeräts und welche Kommunikationssysteme stehen zur Deckung des Kommunikationsbedarfs zur Verfügung? Die für diese Fragen relevanten Entscheidungen werden üblicherweise im Rahmen der E/E-Systementwicklung gefällt und von der Softwareentwicklung als Anforderungen bzw. Randbedingungen übernommen. Durch verschiedene Trends hat der hierbei entstehende Abstimmungsbedarf in letzter Zeit zugenommen. Dazu gehört z. B. die steigende Zahl von verteilten Funktionen, die sich inhaltlich oft querschnittlich zu der an Steuergeräten orientierten

Projektorganisation verhalten. Ein weiteres hier zu nennendes Themenfeld sind die neuen Assistenzsysteme. Da diese teilweise aktiv Einfluss auf das Fahrzeug nehmen, unterliegen sie oft Sicherheits- und Verfügbarkeitsanforderungen, die eng koordinierte Hardware- und Softwaremaßnahmen für die Fehlererkennung und -behandlung erfordern. Auf der Ebene der Hardware kann dies z. B. die redundante Auslegung der Kommunikationskanäle und der Energieversorgung notwendig machen. Auf der Ebene der Software muss dies z. B. durch eine *Voting*-Funktion (bei Abweichung der redundanten Werte) oder ein Energiemanagement berücksichtigt werden.

Die oben geschilderten Tendenzen, wie z. B. die zunehmende Vernetzung der Funktionen oder die eng zu koordinierenden Maßnahmen bei sicherheitsrelevanten Systemen, führen zu einer komplexen Abhängigkeit zwischen System- und Softwareentwicklung, die mit den herkömmlichen Methoden immer schwieriger zu beherrschen ist. Es zeigt sich jedoch ein Trend bei den Entwicklungsmethoden, bei dem sich der Fokus weg von der isolierten Entwicklung von Einzelmodulen hin zu einer funktionsorientierten Entwicklung des Gesamtsystems bzw. dessen Subsystemen bewegt. In diesem Kontext wird die Softwareentwicklung immer stärker als eine Teilaktivität einer übergeordneten Systementwicklung betrachtet, in der viele Fachdisziplinen eng miteinander verzahnt sind. Um die Abhängigkeiten der Entwicklungsartefakte aus unterschiedlichen Bereichen zu beherrschen, werden diese auf möglichst feingranularer Ebene miteinander verknüpft. Auf diese Weise soll eine durchgängige Verfolgbarkeit (engl. *Traceability*) hergestellt werden, um z. B. leicht ermitteln zu können, welche Softwarekomponenten welche Funktionen aus der Systemarchitektur realisieren. Bei der Formalisierung und Verknüpfung der Informationen verfolgen die meisten Vorschläge aus dem Bereich der Systementwicklung modellbasierte Ansätze, die eng mit den in der Softwareentwicklung etablierten Ansätzen verwandt sind. Diese Modelle werden für die Softwareentwicklung durch die angesprochene Verzahnung zu Berührungspunkten mit der Systementwicklung.

Zu den für die Softwareentwicklung relevanten Artefakten, die aus Systementwicklung hervorgehen, gehören die *logische* und *technische Systemarchitektur*. Die logische Systemarchitektur beschreibt die Funktionen und Eigenschaften des Gesamtsystems (auch der Nicht-Softwareanteile) unabhängig von einer möglichen technischen Realisierung. Der Fokus liegt hierbei auf der abstrakten Beschreibung der Systemstruktur und des Zusammenwirkens der einzelnen Subsysteme mit den vorhandenen logischen Sensoren, Sollwertgebern und Aktuatoren. Die logische Systemarchitektur wird im Laufe der Systementwicklung auf die technische Systemarchitektur abgebildet, die eine mögliche technische Realisierung für die Funktionen der logischen Systemarchitektur darstellt. Bei dieser *Partitionierung* wird eine Funktion in Einzelteile zerlegt, die unterschiedlichen Steuergeräten zugeordnet werden können. Die technische Systemarchitektur beschreibt z. B. den Hardware-Aufbau von Sensoren, Aktuatoren und Steuergeräten sowie die Vernetzung der Hardwareeinheiten.

Abbildung 1.2 zeigt schematisch die Inhalte der logischen und technischen Systemarchitektur. Die logische Systemarchitektur umfasst ein Funktionsnetz aus drei logischen Sensoren/Sollwertgebern, zwei Steuerungs-/Regelungseinheiten sowie zwei logischen

Aktuatoren. Zwischen diesen logischen Einheiten sind die abstrakten Signale S1-S6 definiert. In der technischen Systemarchitektur ist im unteren Teil ein Steuergerätenetzwerk mit einem Bus und drei Steuergeräten beschrieben, die zwei Busnachrichten austauschen. Darüber ist beispielhaft für Steuergerät 2 der Aufbau eines Steuergeräts mit seinen angeschlossenen Sensoren/Sollwertgebern und Aktuatoren dargestellt. Die gestichelten Linien repräsentieren die Abbildung zwischen logischer und technischer Systemarchitektur. So werden beispielsweise Steuerungs- und Regelungseinheiten CPUs zugeordnet (sofern sie durch Software implementiert werden), logische Sensoren und Aktuatoren werden den entsprechenden Sensoren und Aktuatoren auf der Hardwareebene zugeordnet und die abstrakten Signale aus der logischen Systemarchitektur werden Busnachrichten zugeordnet (sofern die kommunizierenden Funktionen nicht demselben Steuergerät zugeordnet sind).

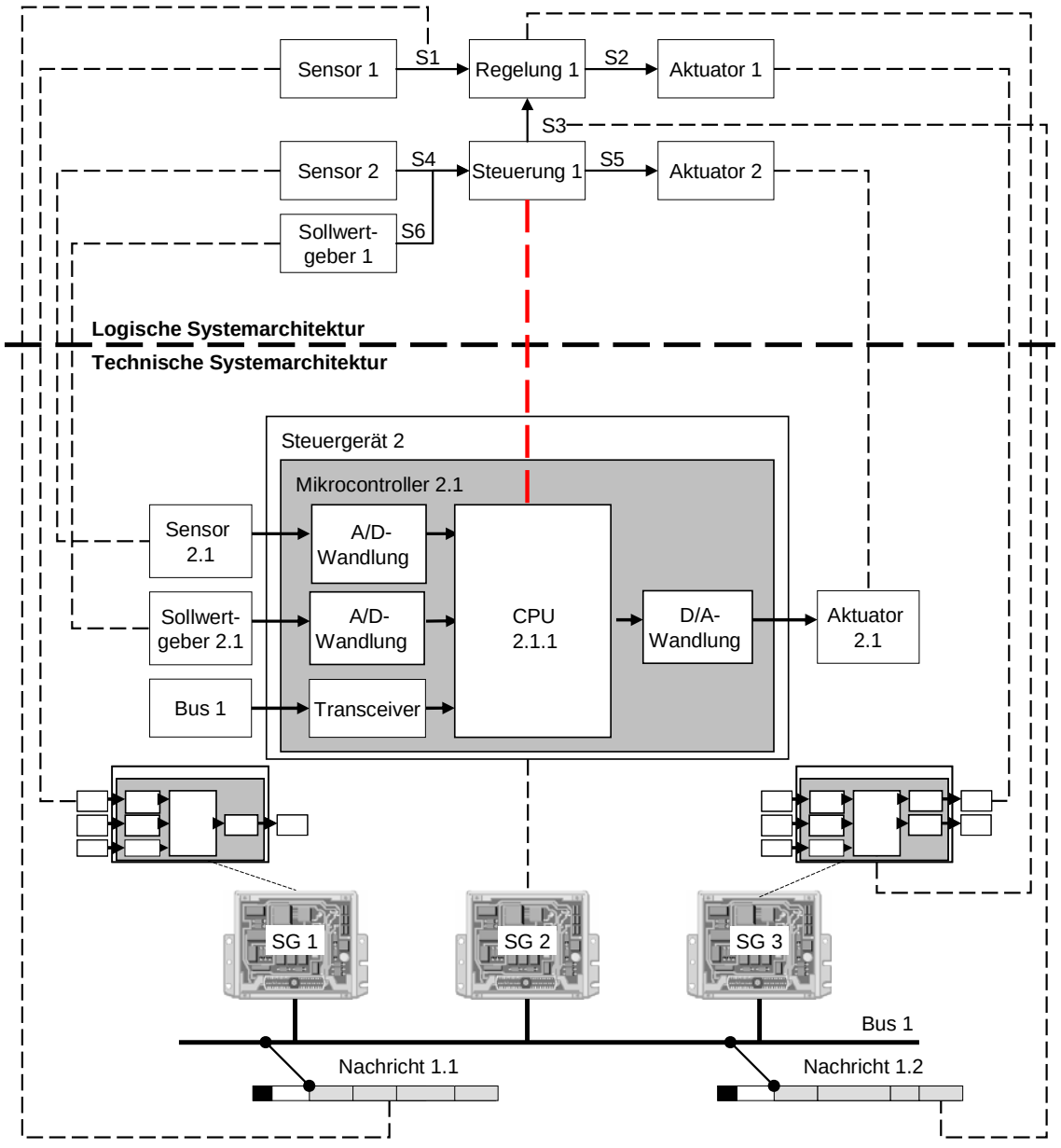


Abbildung 1.2: Logische und technische Systemarchitektur

Im Kontext dieser Sichtweise ergibt sich das Entwurfsproblem der Softwareentwicklung durch Verknüpfungen zwischen Funktionen und Steuergerät bzw. CPU (rot markiert). Für einen Mikrocontroller muss ein Programmcode bereitgestellt werden, der einerseits alle dem Steuergerät zugewiesenen Funktionen der logischen Systemarchitektur realisiert und andererseits auf die durch die technische Systemarchitektur beschriebene Plattform abgestimmt ist. Darüber hinaus müssen weitere Anforderungen berücksichtigt werden, insbesondere auch nichtfunktionale. Dazu gehören z. B. gesetzliche Bestimmungen, Industriestandards und klassische Anforderungen wie Wiederverwendung, Wartbarkeit oder Variabilität [Schäuffele et al. '03].

Die logische und technische Systemarchitektur der Systementwicklung sind somit Analysegegenstand der Softwareentwicklung. Stark vereinfacht könnte dieser Zusammenhang auch wie folgt formuliert werden: Das Designmodell der Systementwicklung ist das Analysemodell der Softwareentwicklung.

Eine entscheidende Herausforderung bei der Erstellung des Softwaredesigns, die dem Softwareentwickler sein ingenieurmäßiges Können abverlangt, ist die Tatsache, dass bei einer Funktion die nichtfunktionalen Anforderungen und die Plattformen variieren können. Die nichtfunktionalen Anforderungen können außerdem im gegenseitigen Konflikt stehen und eine Gewichtung notwendig machen. Diese Gewichtung sieht beispielsweise bei einer sicherheitsrelevanten Regelung im Fahrwerksbereich anders aus, als bei einer Regelung, die eine Komfortfunktion im Innenraum realisiert. Außerdem kann die aus Sicht der logischen Systemarchitektur gleiche Steuerung oder Regelung bei verschiedenen Fahrzeugmodellen zum Einsatz kommen, die mit unterschiedlichen technischen Plattformen ausgestattet sind. So können z. B. die benötigten Sensoren in dem einen Fall direkt am Steuergerät angeschlossen sein, im anderen Fall aber nur indirekt über den Bus zugreifbar sein. In einem Fahrzeug muss die Software als Teil eines fahrzeugweiten Energiemanagements fungieren, in einem anderen Fahrzeug besteht diese Anforderung nicht. In Abhängigkeit dieser wechselnden Anforderung müssen Designentscheidungen getroffen werden, die letztendlich zu unterschiedlichen Designlösungen für dieselbe logische Systemarchitektur führen können. Diese Variabilität und die damit verbundene Komplexität beim Design müssen mit Hilfe entsprechender Methoden effizient beherrscht werden.

Der Prozess der Softwaredesignentscheidungsfindung stützt sich bei aktuellen Vorgehensweisen oft auf informelle Regeln oder Erfahrungswerte. Einer der systematischsten Ansätze bei der Softwaredesignerstellung ist die Verwendung von Designmustern. Ein Designmuster liefert eine in der Praxis bewährte Lösungsschablone für ein wiederkehrendes Entwurfsproblem. Diese Entwurfsprobleme decken sich in großen Teilen mit oben angesprochenen nichtfunktionalen Anforderungen. So gibt es beispielsweise eine Gruppe von Designmustern, die die Wiederverwendbarkeit eines Designs fördern und eine andere Gruppe, die Sicherheits- und Verfügbarkeitsanforderungen adressieren. Für Muster gibt es einheitliche Beschreibungsschemata, die z. B. das Entwurfsproblem, die Lösungsschablone, Vor- und Nachteile des Musters sowie Anwendungsbeispiele beinhalten. Muster, die entsprechend eines solchen Schemas beschrieben sind, werden üblicherweise in Katalogen organisiert. Der Entwickler kann dann anhand von Schlüsselwörtern

potentielle Muster für ein Entwurfsproblem finden und anhand der Beschreibung prüfen, ob es zu dem konkreten Anwendungskontext passt. Moderne Entwicklungsumgebungen können Lösungsschablonen automatisiert in das Designmodell übernehmen und unterstützen den Entwickler bei der Anpassung an das restliche Design.

Insgesamt ergibt sich folgende Situation: Neue Ansätze aus dem Bereich der Systementwicklung formalisieren und standardisieren Informationen auf der Basis von Modellen stärker als bisher und streben eine enge Verzahnung mit der Softwareentwicklung an. In der Softwareentwicklung haben sich seit einiger Zeit Modelle etabliert, die technisch große Ähnlichkeiten mit den Modellen der Systementwicklung haben.

Ansätze im Umfeld modellgetriebener Softwareentwicklung versprechen Modelle aus unterschiedlichen Bereichen integrieren zu können. Für den Übergang zwischen solchen Modellen wird das Konzept der Modelltransformationen angeboten, das diesen Vorgang weitgehend automatisieren soll. Da sowohl in der System- als auch in der Softwareentwicklung Modelle genutzt werden, erscheinen die Ansätze der modellgetriebenen Entwicklung als eine Möglichkeit, die angestrebte Verzahnung von System- und Softwareentwicklung zu erreichen.

Eine aktuell sehr bekannte Ausprägung von modellgetriebener Entwicklung ist die *Model Driven Architecture* (MDA). Unter diesem Oberbegriff bündelt die Object Management Group (OMG) einige Standardisierungsaktivitäten, die z. B. die Formalisierung von Modellen (Metamodellierung), den Austausch von Modellen und die Transformationsregeln umfassen. Ein Ziel der MDA ist es, mit diesen Technologien plattformunabhängige Modelle (*Platform Independent Models – PIMs*) automatisch in plattformspezifische Modelle (*Platform Specific Models – PSMs*) überführen zu können.

In der Praxis hat sich die modellgetriebene Entwicklung noch nicht umfassend durchgesetzt. Ein Grund hierfür ist der noch relativ niedrige Reifegrad der beteiligten Technologien und die z.T. noch fehlende Standardisierung. Speziell im Bereich der Modelltransformationen gibt es momentan noch sehr viele unterschiedliche Ansätze, die i. d. R. noch experimentellen Status besitzen. Auch gibt es nur wenig Arbeiten, die sich umfassend mit der konkreten inhaltlichen Ausgestaltung von Modelltransformationen in einer bestimmten Domäne beschäftigen. Entsprechend gibt es auch kaum fundierte Daten, um welchen Umfang sich der Automatisierungsgrad tatsächlich erhöhen lässt.

1.2 Problemstellung

Die modellgetriebene Entwicklung verspricht Informationen unterschiedlicher Entwicklungsphasen durchgängig in formalisierten Modellen zu erfassen und Übergänge zwischen Modellen teilweise automatisieren zu können. Aufgrund dieser Eigenschaften scheint sich dieser Ansatz auch für die Integration von System- und Softwareentwicklung bei der Entwicklung von *Automotive Software* anzubieten. Zum einen sind die geforderten Modelle für beide Ebenen vorhanden, zum anderen werden viele aktuelle Herausforderungen im angesprochenen Bereich der Komplexitätsbeherrschung adressiert.

Im Vergleich zu vielen anderen IT-Anwendungsdomänen existieren in der Form der Systemmodelle zu Beginn der Softwareentwicklung umfangreiche Informationen zu den Anforderungen mit hohem Formalisierungsgrad. Hier lässt sich vermuten, dass sich diese systematisch in der Softwareanalyse wiederverwenden lassen und dass mit Hilfe des Konzepts der Modelltransformationen automatisiert Softwaredesignmodelle abgeleitet werden können.

Im Rahmen dieser Arbeit werden Modelltransformationen für diesen Kontext untersucht. Mängel existierender Ansätze, die sich in diesem konkreten Anwendungsumfeld ergeben, werden identifiziert. Folgende Fragestellungen liegen den Untersuchungen bestehender Methoden und Technologien im nächsten Kapitel zugrunde:

- Was sind die zentralen Anforderungen und Trends bei softwareintensiven Systemen im Kraftfahrzeug, die durch Modelltransformationen in diesem Bereich adressiert werden sollten?
- Inwiefern unterstützten herkömmliche Methoden die integrierte System- und Softwareentwicklung durch den Austausch von Modellen? Wie sieht der Entwicklungsprozess aus und welche Modelle werden konkret an den Schnittstellen zwischen System- und Softwareentwicklung ausgetauscht?
- Welche Anforderungen an Modelltransformationsmechanismen müssen gestellt werden, wenn diese dazu genutzt werden sollen, Modelle der Systementwicklung automatisiert in Modelle der Softwareentwicklung zu überführen?
- Wie lässt sich der Zusammenhang zwischen Modellen der System- und Softwareentwicklung derart systematisieren, dass dieser formalisiert beschrieben werden kann? Wie lassen sich auf dieser Grundlage Abbildungsregeln für einen Modelltransformationsmechanismus formulieren?
- Welche Ansätze zu Modelltransformationen gibt es und inwiefern erfüllen diese die zuvor gesammelten domänenspezifischen Anforderungen?

Die Ergebnisse der Untersuchungen obiger Fragen fließen in eine neue Methode für den systematisierten Übergang zwischen System- und Softwaredesignmodellen auf der Basis von Modelltransformationen ein. Die Methode ist Gegenstand dieser Arbeit und besteht aus der Erweiterung eines bestehenden Softwareentwicklungsprozesses, einem erweiterten Template-Mechanismus für Analyse- und Designmuster, einer neuen Modelltransformationssprache, die den Kernteil dieser Arbeit darstellt und einem Verfolgbarkeitsmodell.

1.3 Abgrenzung

Die Problemstellung dieser Arbeit umfasst die Entwicklung einer Modelltransformationslösung, die auf Anforderungen einer integrierten System- und Softwareentwicklung im Bereich der Automobilindustrie abgestimmt ist. Um das Thema im Rahmen einer Dissertation bearbeiten zu können, mussten bereits zu Beginn Einschränkungen getroffen werden, von denen im Folgenden einige erläutert sind.

Sowohl in der System- als auch in der Softwareentwicklung gibt es Ansätze, neben der Struktur auch das Verhalten eines Systems zu modellieren. Der in dieser Arbeit entwickelte Modelltransformationsmechanismus ist für den Umgang mit Strukturmodellen optimiert – Modelltransformationen von Verhaltensmodellen werden nicht betrachtet. Da sich der POTAD-Ansatz im Wesentlichen auf die Instanziierung von Mustern stützt, werden die Möglichkeiten POTAD in diese Richtung zu erweiterten als günstig eingeschätzt (siehe Kapitel 7.1). Hier gibt es Ansätze, die bei der Instanziierung eines Musters neben einem Struktur- auch ein Verhaltensmodell instanziieren.

Im Rahmen der Arbeit wird das in Kapitel 3.1.2 erläuterte V-Modell zu Grunde gelegt. Der besondere Fokus liegt auf dem Übergang zwischen System- und Softwareentwicklung im linken Ast des V-Modells. Die Verwendung der erläuterten Modelle und Modelltransformationen hat auch Implikationen auf den rechten Ast des V-Modells. Diese werden aber nicht weiter untersucht.

Die in der Arbeit gezeigten Beispiele gehen davon aus, dass die Software für eine Funktion vollständig neu nach *top-down*-Vorgehensweise entwickelt wird. In der Praxis wird aber oft auf bestehenden Implementierungen aufgesetzt, die mit neuentwickelten Anteilen integriert werden. Da es Muster für diese Problematik gibt, wird davon ausgegangen, dass sich auch Transformationsregeln formulieren lassen, die statt der Erzeugung eines neuen Designs existierende Implementierungen integrieren.

Im Fokus der Arbeit steht die Entwicklung der Modelltransformationssprache. Die gezeigten Beispiele für Modelltransformationsregeln erzeugen ein Design, das von einzelnen Experten als sinnvoll für einen bestimmten Kontext erachtet wird. Die Regeln wurden aber nicht systematisch anhand vieler Systeme evaluiert. Eine solche Evaluierung ist ein mögliches Thema für zukünftige Arbeiten (siehe Kapitel 7.1).

Ein klassisches Thema im Softwaredesign ist die Behandlung von Nebenläufigkeit. Die in der Arbeit gezeigten Designbeispiele verwenden ein in der Automobilindustrie verbreitetes *Scheduling*-Verfahren, bei dem es keine nebenläufigen Tasks gibt. Die Behandlung von Nebenläufigkeit wird somit durch die Transformationsregeln nicht adressiert. Da es aber auch hier zahlreiche Muster zu dem Thema gibt, wird davon ausgegangen, dass sich die Transformationsregeln in diese Richtung erweitern lassen.

1.4 Organisation der Arbeit

In Kapitel 2 wird ein durchgängig für diese Arbeit genutztes Fallbeispiel vorgestellt, das typische Anforderungen an *Automotive Software* enthält. Es dient im weiteren Verlauf der Arbeit als Basis für die Untersuchung der existierenden Ansätze und der Diskussion der eigenen Lösung.

In Kapitel 3 wird der relevante Stand der Technik umrissen und anhand des Fallbeispiels erläutert. Nach grundlegenden Betrachtungen zu den Eigenschaften der Domäne und den

Prozessen und Modellen der System- und Softwareentwicklung wird mit Untersuchungen zu Modelltransformationen und Mustern auf den eigentlichen Themenkomplex dieser Arbeit fokussiert. Die Ergebnisse resultieren schließlich in einer präzisierten Problemstellung.

Kapitel 4 stellt zu dieser Problemstellung den eigenen Lösungsansatz *Pattern-Oriented Transformations between Analysis and Designmodels* (POTAD) vor. Nach einigen Festlegungen bzgl. der verwendeten Entwicklungsmethode sowie den verwendeten Modellen und Mustern wird ein Formalismus für die Beschreibung musterbasierter Transformationsregeln vorgestellt, der durch die Einbeziehung von Zusatzinformationen bzgl. unterschiedlicher nichtfunktionaler Anforderungen und der technischen Systemarchitektur das Zielmodell variieren kann.

Eine prototypische Implementierung des Ansatzes auf Basis des Werkzeugs *Rational XDE* wird in Kapitel 5 vorgestellt. Hier wurde ein Plug-in geschrieben, das die entwickelten Transformationsregeln ausführen kann. Abschließend wird eine Beispielregel detailliert erläutert.

Eine kritische Reflexion des eigenen Ansatzes erfolgt in Kapitel 6. Die Ergebnisse werden bewertet und den eingangs formulierten Anforderungen gegenübergestellt. Im anschließenden Kapitel 7 wird die Arbeit zusammengefasst, die Anwendbarkeit in anderen Domänen diskutiert und ein Ausblick auf mögliche zukünftige Themen gegeben.

2 Fallbeispiel: Kombiinstrument- und Fahrfunktionen

Am Beispiel eines kleineren Systems aus Kombiinstrument- und Fahrfunktionen wurde die Problemstellung dieser Arbeit entwickelt, der Lösungsansatz (siehe Kapitel 4) überprüft, weiterentwickelt und bewertet. Dieses Kapitel stellt das System vor und erläutert dessen wichtigsten Eigenschaften. Für beispielhafte Erläuterungen im weiteren Verlauf dieser Arbeit wird das Kombiinstrument- und Fahrfunktionen-System (KFS) zugrunde gelegt.

2.1 Übersicht

Um mit praxistypischen Anforderungen zu arbeiten, wurden für die Fallstudie eine Menge von zusammenhängenden Funktionen zugrunde gelegt, die sich an die Funktionalität der Mercedes-Benz C-Klasse (Baureihe 203) anlehnen. Zu den Fahrfunktionen des KFS gehören ein klassischer Tempomat und die sog. *Speedtronic*. Der Tempomat bringt und hält die Geschwindigkeit des Fahrzeugs automatisch auf einen vom Fahrer eingestellten Wert. Die *Speedtronic* begrenzt dagegen die Höchstgeschwindigkeit des Fahrzeugs auf einen vom Fahrer eingestellten Wert. Die Kombiinstrumentfunktionen umfassen die Berechnung, Anzeige und Speicherung von Fahrdaten wie der aktuellen Geschwindigkeit, Tages- bzw. Gesamtkilometer, Kilometer bis zum nächsten Service-Intervall und die aufgetretenen Fehler. Außerdem zeigt das Kombiinstrument auch den Status der Fahrfunktionen an. Eine ausführliche Beschreibung der Benutzeranforderungen findet sich in Anhang C. In Abbildung 2.1 ist die logische Systemarchitektur des KFS analog zu der Notation aus [Schäuffele et al. '03] skizziert.

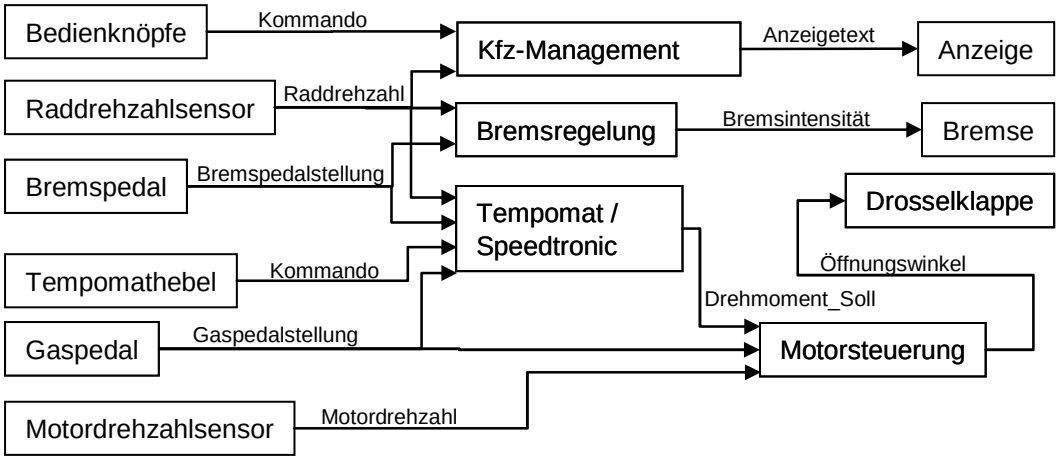


Abbildung 2.1: Logische Systemarchitektur des KFS

Das KFS deckt die Fahrzeugdomänen Antriebstrang, Fahrwerk (Fahrfunktionen) und Telematik (Kombiinstrumentfunktionen) ab (siehe Kapitel 3.1.1.f). Damit sind zwei Subdomänen mit recht unterschiedlichen Eigenschaften vertreten. So handelt es sich bei den Fahrfunktionen

überwiegend um kontinuierliche Funktionen, die Kfz-Managementfunktionen und das Kombiinstrument haben reaktiven Charakter.

Im Rahmen der Fallstudie wurde von der in Abbildung 2.2 grob und informell skizzierten technischen Systemarchitektur ausgegangen. Es gibt zwei Steuergeräte, die über einen Feldbus (wie z. B. CAN) verbunden sind:

- **Mensch-Maschine-Interaktion (MMI) und Kfz-Management**: An dieses Steuergerät sind alle für die Benutzerinteraktion notwendigen Bedien- und Anzeigeelemente angeschlossen. Im Steuergerät werden die Eingaben ausgewertet, verarbeitet und ggf. die Anzeigeelemente aktualisiert. Einige Eingaben (Brems- und Gaspedalstellung, eingestelltes Fahrprogramm, Stellung des Zündschlüssels) werden über den Bus an das zweite Steuergerät weitergeleitet. Zu den auf dem Steuergerät lokal ausgeführten Softwarefunktionen gehören, neben der Berechnung der Anzeigewerte, die Kfz-Managementfunktionen (Kapitel 4 in Anhang C). Um diese Funktionen ausführen zu können, müssen über den Bus der Wert des Raddrehzahlsensors und auftretende Fehlermeldungen des Steuergeräts *Antriebstrang und Fahrwerk* empfangen werden.
- **Antriebstrang und Fahrwerk**: An diesem Steuergerät sind die für die Fahrfunktion notwendigen Aktuatoren und Sensoren angeschlossen. Zu den lokal ausgeführten Softwarefunktionen gehören die Tempomat- und *Speedtronic*-Funktion, die Motorsteuerung und das Ansteuern der Bremsen. Um diese Funktionen ausführen zu können, ist das Steuergerät darauf angewiesen, dass die Eingabewerte des Brems- und Gaspedals, das eingestellte Fahrprogramm und die Stellung des Zündschlüssels durch das Steuergerät *MMI und Kfz-Management* über den Bus gesendet werden.

Es ist zu beachten, dass diese Architektur nicht der in aktuellen Serienfahrzeugen entspricht. So sind kritische Eingabelelemente wie z. B. das Brems- oder Gaspedal in der Regel direkt an das verarbeitende Steuergerät angebunden und nicht wie hier, über einen Bus. Außerdem verteilt sich die gezeigte Funktionalität bei heutigen Fahrzeugen auf mehr als die hier gezeigten zwei Steuergeräte. So unterteilt sich beispielsweise die Funktionalität des Steuergeräts *Antriebstrang und Fahrwerk* oft auf mindestens drei Steuergeräte, z. B.: Bremsfunktionen, Motorsteuerung und Geschwindigkeitsfunktionen (oft kombiniert mit Getriebesteuerung). Bremsfunktionen und Motorsteuerung sind in der hier gezeigten Architektur zudem stark vereinfacht und wurden nur mit aufgenommen, um den Fahrfunktionen die benötigten Schnittstellen zur Verfügung zu stellen. Eine Motorsteuerung verfügt z. B. über weitaus mehr Sensoren und Aktuatoren als der hier gezeigte Motordrehzahlsensor und die Drosselklappe (siehe z. B. [BOSCH '03]). Die gezeigte technische Systemarchitektur passt zu einem in der Automobilindustrie angestrebten Zukunftsszenario, indem die Reduzierung der Steuergeräte durch die Integration von Softwarefunktionen weiter fortgeschritten ist.

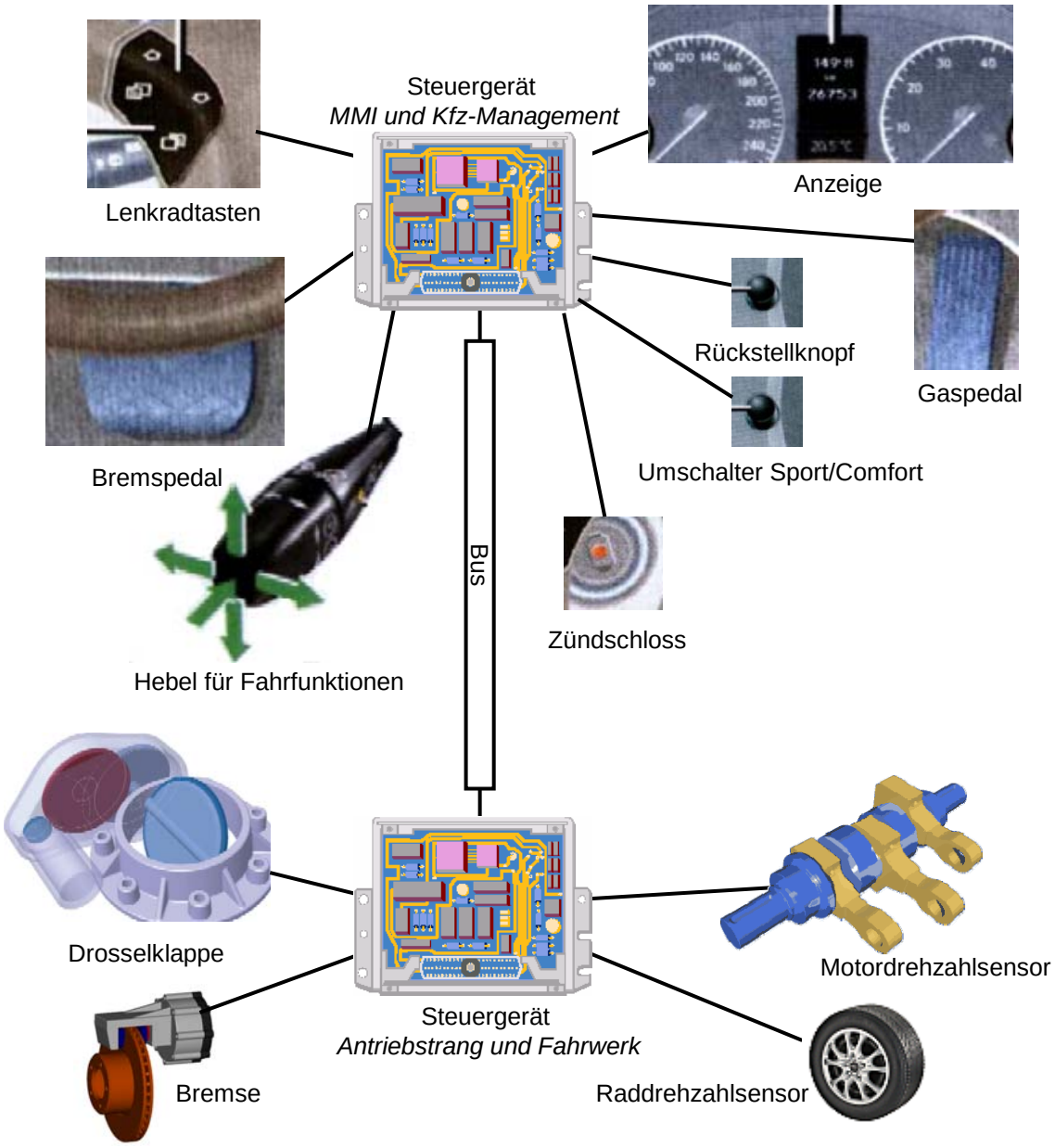


Abbildung 2.2: Skizze der technischen Systemarchitektur der Fallstudie (informell)

Der für den Lösungsansatz dieser Arbeit relevante Aufgabenbereich bei der Entwicklung des KFS umfasst die Abbildung der Funktionen (Kfz-Management, Fahrfunktionen etc.) aus der logischen Systemarchitektur auf ein Softwaredesign, das zu der in Abbildung 2.2 gezeigten technischen Systemarchitektur passt. Um sich auf diese Aufgabe konzentrieren zu können, wurde für die technische Systemarchitektur und die Umgebung (Regelstrecke) ein Softwaresimulator entwickelt, der auf einem PC ausgeführt werden kann (Abbildung 2.3).

Zu dem Funktionsumfang des Softwaresimulators gehört die Bereitstellung der Ein- und Ausgabeelemente in einem GUI, mit dem der Benutzer das KFS am PC bedienen kann. Durch die Bereitstellung eines virtuellen Busses lässt sich die technische Systemarchitektur mit zwei Rechenknoten abbilden. Ferner enthält die Simulation für den Geschwindigkeitssensor ein Um-

gebungsmodell (Regelstreckenmodell), so dass die Fahrfunktionen mit plausiblen Geschwindigkeitswerten versorgt werden können. Letztendlich stellt der Simulator eine Umgebung zur Verfügung, in der die Hardware der technischen Systemarchitektur über eine Programmierschnittstelle virtuell nutzbar gemacht wird. In diesen Rahmen hinein können die eigentlichen Softwarefunktionen des KFS implementiert werden, deren Entwurfsmethodik im Fokus dieser Arbeit ist.

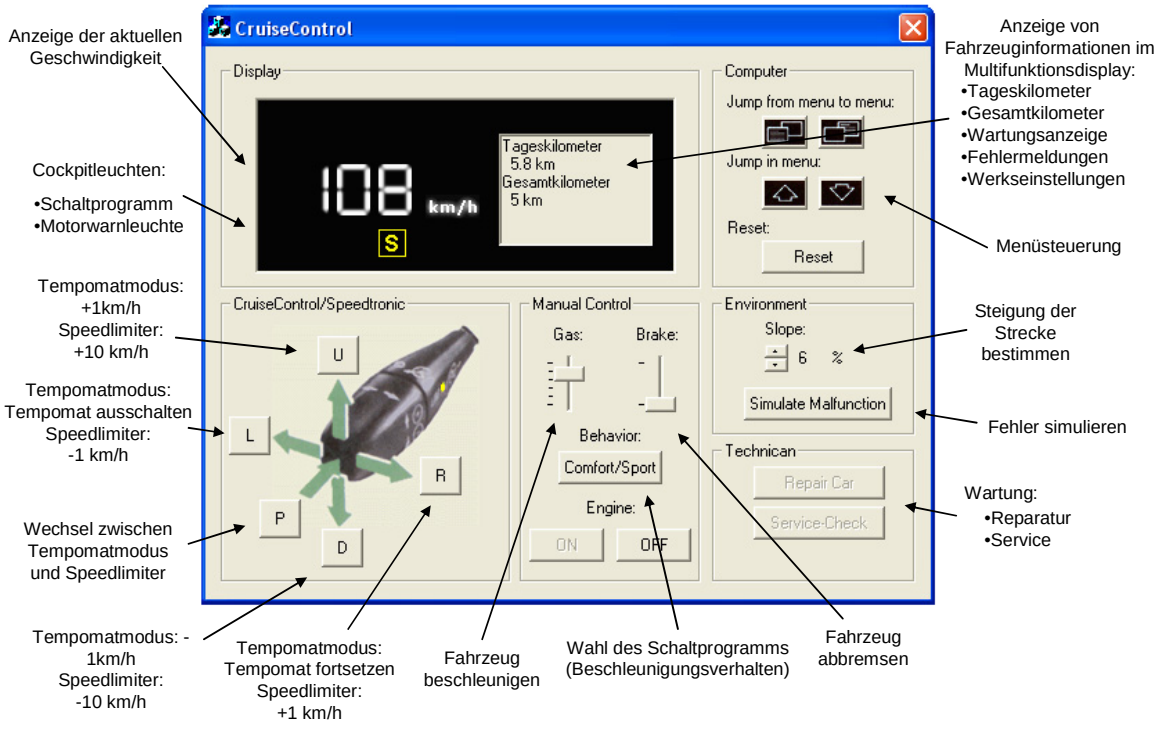


Abbildung 2.3: Softwaresimulator für das KFS

2.2 Design der Softwarefunktionen

In einem ersten Schritt wurde im Rahmen einer Diplomarbeit (siehe Kapitel 6.2) ein Softwaredesign für das KFS mit einer herkömmlichen modellbasierten Entwicklungsmethode (ROPES, siehe Kapitel 3) entwickelt. Dazu wurde die logische und technische Systemarchitektur in den entsprechenden Modellen der ROPES-Methode nachmodelliert und anschließend der vorgeschriebene Prozess durchlaufen.

Viele Funktionen des KFS haben Steuerungs-, Regelungs- oder Überwachungsaufgaben, die nach einem ähnlichen logischen Schema aufgebaut sind: zyklisch werden aktuelle Sensor- und Sollwertgeberdaten eingelesen, mit denen eine Berechnung durchgeführt wird, deren Ergebnis an Aktuatoren gesendet wird, die dieses in physikalische Arbeit umsetzen. Aufgrund dieser Ähnlichkeit lässt sich vermuten, dass sich auch ein einheitliches Design für Softwarefunktionen mit Steuerungs-, Regelungs- oder Überwachungsaufgaben finden lässt. Am Beispiel der ersten Umsetzung des KFS wurde jedoch deutlich, dass das Design dieser Funktionen stark variieren kann. Der Grund liegt zum einen in der Platzierung der Softwarefunktion in der technischen

Systemarchitektur und zum anderen in den unterschiedlichen nicht-funktionalen Anforderungen an die Funktion. Im Folgenden sind ein paar Beispiele aus dem KFS genannt die zeigen, wie sich unterschiedliche Rahmenbedingungen auf das Design auswirken können.

- Wenn die Tempomatfunktion als sicherheitsrelevant eingestuft wird, müssen die Ein- und Ausgangswerte plausibilisiert werden und evtl. redundant mit unterschiedlichen Verfahren berechnet werden. Bei der Berechnung der aktuellen Geschwindigkeit für die Anzeige mit Hilfe des Raddrehzahlsensors ist dies nicht notwendig.
- Das Steuergerät *MMI und Kfz-Management* ist mit relativ umfangreichen Ressourcen bzgl. Rechenleistung und Speicher ausgestattet. Da die auf diesem Steuergerät laufenden Funktionen in jeder Baureihe vorkommen, soll das Softwaredesign möglichst wiederverwendbar sein. Die Optimierung des Designs in Richtung Wiederverwendung darf bei diesen Rahmenbedingungen auch etwas zu Lasten des Ressourcenverbrauchs gehen. Bei dem Steuergerät Antriebstrang und Fahrwerk ist dagegen der Ressourcenverbrauch kritischer. Die Motorsteuerung muss in sehr kurzen Zeitabständen die Werte für die Aktuatorik berechnen können (Echtzeitanforderungen). Da die Algorithmen außerdem stark auf einen bestimmten Motor optimiert sind und sich dadurch nicht viele Wiederverwendungsmöglichkeiten ergeben, liegt in diesem Fall der Schwerpunkt des Designs eher auf der Performance-Optimierung.
- Die Motorsteuerung empfängt die aktuelle Gaspedalstellung als digitalisierten Wert über den Bus in zyklischen Abständen von 10 ms, die Motordrehzahl wird über eine diskrete Leitung bezogen. Während die Gaspedalstellung direkt verwendet werden kann, ist bei der Motordrehzahl eine Vorverarbeitung des Wertes notwendig.

Am Beispiel der ersten Umsetzung des KFS wird deutlich, dass Funktionen, die sich auf einer logischen Ebene mit einem einheitlichen Schema beschreiben lassen, auf der Designebene stark variieren können. Bei der Erstellung des Designs müssen Anforderungen berücksichtigt werden, von denen vorher abstrahiert wurde. Das KFS-Design zeigt aber auch, dass es auch für die Umsetzung dieser Anforderungen Designschemata gibt (vgl. Kapitel 3.2.4), die wiederum unabhängig von der logischen Funktion sind. So gibt es beispielsweise für die Anforderungen im Bereich Wiederverwendung Designtechniken, die bei eingebetteter Regelungssoftware genauso eingesetzt werden, wie bei betriebswirtschaftlicher Software für Großrechner.

Letztendlich sucht der Entwickler bei der Designerstellung in der logischen Systemarchitektur nach gewissen Mustern. Für die Designumsetzung dieser Muster nutzt er in Abhängigkeit von den Randbedingungen (technische Systemarchitektur, nichtfunktionale Anforderungen) ebenfalls wieder gewisse Muster. Bei der Erstellung des ersten KFS-Designs gab es zahlreiche Anhaltspunkte, die vermuten ließen, dass sich dieser Zusammenhang weit aus mehr systematisieren lässt, als in den herkömmlichen Methoden vorgesehen. Die Problemstellung der Arbeit leitet sich aus der Frage ab, ob diese Systematisierung so weitgehend möglich ist, dass sie im Rahmen einer Modelltransformation für einen automatisierten Übergang von Analyse- zu Designmodellen genutzt werden kann.

3 Stand der Technik

Im Folgenden sollen die für diese Arbeit relevanten Technologien und Methoden vorgestellt, erläutert und in einen Gesamtzusammenhang gebracht werden. Zunächst werden die Methoden der System- und Softwareentwicklung vorgestellt, um damit einen Rahmen für die Untersuchung der Modelltransformationen zu schaffen.

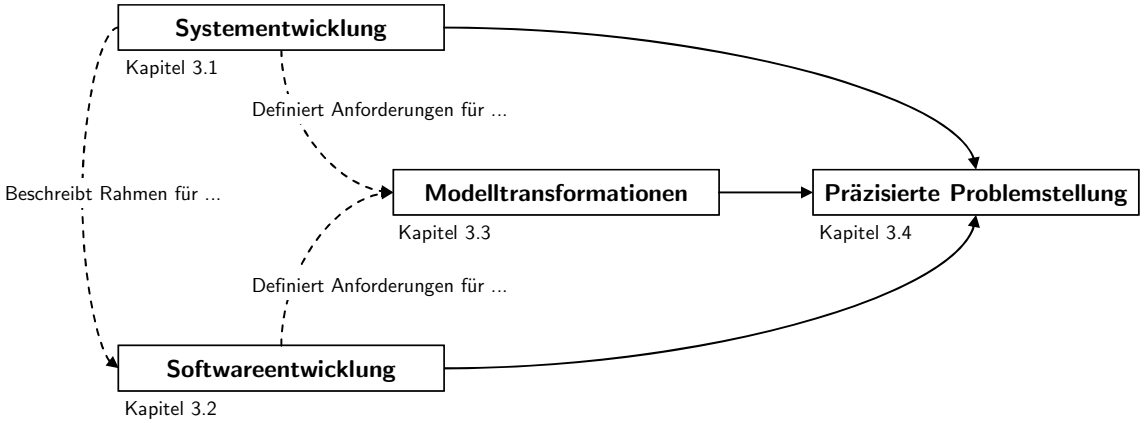


Abbildung 3.1: Überblick der Themen

Abbildung 3.1 zeigt den Aufbau der folgenden Unterkapitel im Überblick. Wegen der großen Wechselwirkung mit der Softwareentwicklung werden zunächst die grundlegenden Merkmale, der Kernprozess und die zu berücksichtigenden Methoden der Systementwicklung vorgestellt. Aus diesen Betrachtungen leiten sich Rahmenbedingungen für die in den Kernprozess eingebettete Softwareentwicklung ab. In dem darauf folgenden Unterkapitel 3.2 wird untersucht, inwiefern aktuelle modellbasierte Softwareentwicklungsmethoden diesen Rahmenbedingungen gerecht werden und ob den neuen Herausforderungen, die sich aus aktuellen Trends der Domäne ableiten (siehe Kapitel 1.2), geeignet begegnet werden kann. Mit den in Kapitel 3.3 besprochenen Modelltransformationen wird ein erster Schritt in Richtung eines Ansatzes vollzogen, der System- und Softwareentwicklung durch automatisiertes Überführen der Modelle stärker integriert. Dazu werden verfügbare Techniken zum Thema Modelltransformationen zusammengetragen und speziell hinsichtlich der in den vorherigen Unterkapiteln gesammelten Anforderungen untersucht. Probleme und Mängel der existierenden Ansätze werden aufgezeigt und am Ende des Kapitels zu einer Problemstellung präzisiert.

3.1 Systementwicklung

Anforderungen an *Automotive Software* folgen oft einer bestimmten Charakteristik. Die Entwicklung der Software ist eine Teilaktivität einer übergeordneten Systementwicklung. Aus diesen Tatsachen leiten sich einige Rahmenbedingungen für die Softwareentwicklungsmethode ab, die im Rahmen der Gesamtentwicklung von elektronischen Systemen im Fahrzeug zum

Einsatz kommt. Dies umfasst zum einen grundlegende Anforderungen an die zu entwickelnde Software selber, zum anderen aber auch an den Prozess, in den die Softwareentwicklung eingebettet ist.

Dieses Kapitel erläutert die Domäne in Hinblick auf diese Merkmale. Die Ausführungen stützen sich dabei im Wesentlichen auf [Schäuffele et al. '03]. Nach den derzeitigen Recherchen ist dies einige der wenigen Arbeiten, die *Automotive Software Engineering* für die Verwertung im Kontext einer wissenschaftlichen Arbeit ausreichend systematisch und umfassend beschreibt.

3.1.1 Grundlegende Anforderungen der Domäne

Die folgenden Unterkapitel gehen auf grundlegende Anforderungen, die im Automobilbereich an softwareintensive Systeme gestellt werden, ein.

3.1.1.a Eingebettete Regelungs- und Steuerungssysteme

Die meisten Softwarefunktionen im Fahrzeug haben steuerungs- oder regelungstechnischen Charakter. Regelungs- und Steuerungssysteme habe die Aufgabe, mit Hilfe von Sensoren, Sollwertgebern und Aktuatoren eine Strecke zu beeinflussen oder zu überwachen. Abbildung 3.2 zeigt, wie sich der grundlegende logische Aufbau solcher Systeme bei einem Fahrzeug darstellt.

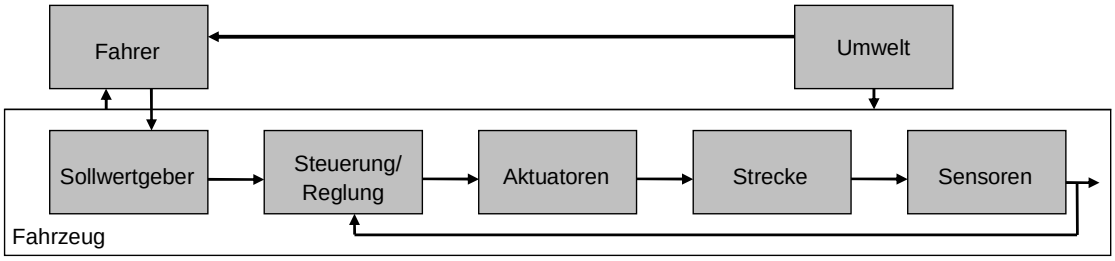


Abbildung 3.2: Das System Fahrer-Fahrzeug-Umwelt [Schäuffele et al. '03]

Im Kontext von *Automotive Software Engineering* werden solche Systeme betrachtet, bei denen die Steuerungs- bzw. Regelungseinheit durch ein sog. elektronisches Steuergerät (engl. *Electronic Control Unit*) realisiert ist, das über einen Mikrocontroller verfügt. Die CPU führt eine Software aus, die das Steuerungs- und Regelungsverhalten implementiert. Ein solches Steuergerät ist für den Fahrer oft unsichtbar und verfügt über keine direkte Benutzerschnittstelle. Es bildet mit den Sensoren und Aktuatoren oft eine integrierte elektronische Komponente, die eng mit der Software verzahnt ist. Ein Steuergerät wird in diesem Kontext daher auch als eine funktionale Einheit aus Hard- und Software betrachtet. Aufgrund dieser Merkmale spricht man bei *Automotive Software* auch von eingebetteten Systemen.

Viele dieser Funktionen müssen zyklisch in festen Abständen aufgerufen werden, um ihre Regelungs- oder Steuerungsaufgabe zuverlässig und stabil zu erfüllen. Außerdem muss bei manchen Funktionen aus Sicherheitsgründen sichergestellt werden, dass auf das Auftreten eines Ereignisses innerhalb einer vordefinierten Zeitschranke reagiert wird. Aus diesen Gründen unterliegt *Automotive Software* oft auch Echtzeitanforderungen.

3.1.1.b Zuverlässige und sichere Systeme

Bei vielen Softwarefunktionen im Fahrzeug kann ein Ausfall im Extremfall für den Fahrer zu einem tödlichen Risiko werden. Aus diesem Grund haben die Analyse der Funktionssicherheit und die Spezifikation geeigneter Konzepte, die dieses Risiko mindert, einen hohen Stellenwert. Für ein fehlertolerantes Verhalten sind z. B. geeignete Maßnahmen für die Fehlererkennung und die Fehlerbehandlung zu implementieren. Viele Sicherheitsmechanismen werden dabei durch eine Kombination aus Hardware- und Softwaremaßnahmen realisiert, wodurch sich besondere Anforderungen an das Softwareengineering ableiten. Eine weitere Voraussetzung für zuverlässige Systeme, die Auswirkung auf die Softwareentwicklung hat, ist die Anforderung Funktionen und Schnittstellen für die Diagnose bereitzustellen.

3.1.1.c Verteilte Systeme

Viele neue Funktionen basieren auf dem Zusammenspiel mehrerer Teilfunktionen, die auf unterschiedlichen Steuergeräten realisiert sind. Ein modernes Oberklassenfahrzeug verfügt mittlerweile über ca. 70 Steuergeräte, die über ca. fünf Bussysteme mehrere tausend Signale austauschen. Die sich daraus ergebende Komplexität durch Abhängigkeiten verschiedener Entwicklungsprojekte muss in der Entwicklungsmethode berücksichtigt werden.

3.1.1.d Wechselnde Plattformen

Der Markt bietet vergleichsweise viele Mikrocontrollerarchitekturen, die für ein Steuergerät genutzt werden können. Die Entscheidung für eine bestimmte Plattform ist stark durch den Einkauf und die Systementwicklung bestimmt. Für die Softwareentwicklung relevante Plattformaspekte der Hardware sind z. B. der verwendete Prozessor (bestimmt die möglichen Compiler und Programmiersprachen) oder die angeschlossenen Bussysteme (haben Auswirkung auf das Kommunikationsmodell und das Zeitverhalten). Auch auf der Ebene der Software spielen Plattformaspekte eine Rolle. So gibt es Standardkomponenten (*Basissoftware*) für z. B. Kommunikation, Energiemanagement oder Diagnose. Außerdem kann für das *Scheduling* z. B. ein Echtzeitbetriebssystem zum Einsatz kommen. Obwohl es mit *OSEK* [OSEK/VDX '05] Bemühungen für eine entsprechende Standardisierung gibt, finden sich in der Praxis immer noch viele herstellerspezifische Basiskomponenten. Es ist angestrebt, diesen Zustand mit der Initiative *Autosar* [AUTOSAR] zu beenden. Trotz der jüngsten Standardisierungsbemühungen hat es die Softwareentwicklung in der Automobilindustrie immer noch mit einer vergleichsweise hohen Anzahl von Plattformen zu tun.

3.1.1.e Weitere nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen sind in der Automobilindustrie von hoher Bedeutung. Aus den oben genannten Merkmalen leiten sich bereits einige solcher Randbedingungen für die Softwareentwicklung ab (z. B. Sicherheit und Zuverlässigkeit). Darüber hinaus gibt es noch einige weitere nichtfunktionale Anforderungen, die im Vergleich zu anderen Domänen besonders hervorstechen und sich hauptsächlich durch Kostenaspekte motivieren:

- **Ressourcenverbrauch**: Aufgrund der hohen Stückzahlen dominieren in der Automobilindustrie die proportionalen Herstellkosten den Stückpreis. Für elektronische Komponenten bedeutet dies, dass möglichst günstige Hardware eingesetzt werden muss, was mit vergleichsweise geringen Hardwareressourcen (Speicher, Rechenleistung, Kommunikationsmittel etc.) verbunden ist. Dies hat sehr hohe Optimierungsanforderungen in der Softwareentwicklung zur Folge.
- **Wiederverwendung und Portabilität**: Ein Hersteller produziert in der Regel mehrere Baureihen, die viele identische Funktionen aufweisen. Entsprechend sollte die Software-Implementierung für die verschiedenen Baureihen wiederverwendbar sein. Dabei muss auch berücksichtigt werden, dass in den unterschiedlichen Baureihen verschiedene Steuergeräteplattformen zum Einsatz kommen können (sowohl Prozessor als auch Betriebssystem).
- **Variabilität**: Viele Softwarefunktionen können in Abhängigkeit mit der vom Kunden gewählten Ausstattungsvariante variieren. Da Software-Varianten einfacher zu handhaben sind als Hardware-Varianten, besteht außerdem häufig die Anforderung, variantenspezifische Anteile eines elektronischen Systems möglichst ausschließlich durch Software zu realisieren.

Darüber hinaus gibt es zahlreiche gesetzliche Bestimmungen und Industriestandards, die zu berücksichtigen sind. Dazu gehören z. B. Datenübertragungsprotokolle, Stecker-Layouts, Diagnoseprotokolle sowie Standards für das Flashen und Codieren von Steuergeräten.

3.1.1.f Bildung von Subsystemen

Einige der bisher genannten Anforderungen stehen im Konflikt, so dass das System- oder Softwaredesign nicht gleichermaßen in alle Richtungen optimiert werden kann und eine entsprechende Gewichtung vorgenommen werden muss.

In der Automobilindustrie hat es sich bewährt, diese Gewichtung in Bezug auf die sog. Subsysteme (Automobilindustrie-intern auch als *Domänen* bezeichnet) festzulegen. Eine mögliche Aufteilung des Gesamtsystems Fahrzeug auf Subsysteme, ist die folgende:

- **Antriebstrang** (engl. *Powertrain*): z. B. Motorsteuerung, Getriebesteuerung.
- **Fahrwerk** (engl. *Chassis*): z. B. Elektronisches Stabilitätsprogramm (ESP), Fahrwerksregelung.
- **Karosserie** (engl. *Body*): z. B. Schließsysteme, Licht, Einparkhilfen.
- **Telematik/Multimedia**: z. B. Kombiinstrument, Audiosysteme, Navigation.

Die Zuordnung der Funktionen zu diesen Domänen orientiert sich in erster Linie über den inhaltlichen Zusammenhang, in zweiter Linie aber auch an den nichtfunktionalen Anforderungen. So gelten beispielsweise in der Domäne Fahrwerk hohe Sicherheits- und Verfügbarkeitsanforderungen. Demgegenüber ist die Karosserie-Domäne eher durch Kostendruck und Ausstattungsvarianten geprägt. Hier müssen die einzelnen Funktionen möglichst wiederverwendbar und variabel implementiert werden. Eine ausführlichere Beschreibung der einzelnen Subsysteme und ihrer Anforderungen findet sich bei [Schäuffele et al. '03].

3.1.2 Der Kernprozess der Systementwicklung

In Kapitel 1.1 wurde bereits auf die Einordnung der Softwareentwicklung in eine übergeordnete Systementwicklung eingegangen. In diesem Kapitel soll detaillierter ein ausgewählter *Kernprozess* der Systementwicklung vorgestellt werden.

Systementwicklung (engl. *System Engineering*) ist ein interdisziplinärer Ansatz für die Analyse und den Entwurf des Systems als Ganzes [Schäuffele et al. '03]. Ziel ist unter anderem ein durchgängiger und strukturierter Entwicklungsprozess, der alle beteiligten Disziplinen in einen einheitlichen Entwicklungsansatz integriert. Für eine ausführlichere Definition sei auf [INCOSE] verwiesen.

Für die Entwicklung von elektronischen Systemen in Kraftfahrzeugen müssen nach dem Kernprozess Fachgebiete wie z. B. Regelungstechnik, Hardwareentwicklung und Netzwerktechnik integriert werden. Die Softwareentwicklung repräsentiert somit nur eine von vielen Fachdisziplinen innerhalb der Systementwicklung. Die Prozessschritte und Artefakte eines solchen Kernprozesses sind in Abbildung 3.3 gezeigt. Es handelt sich bei diesem Beispiel um eine Präzisierung des V-Modells [IABG '97] für die Automobilindustrie, die bei [Schäuffele et al. '03] beschrieben ist.

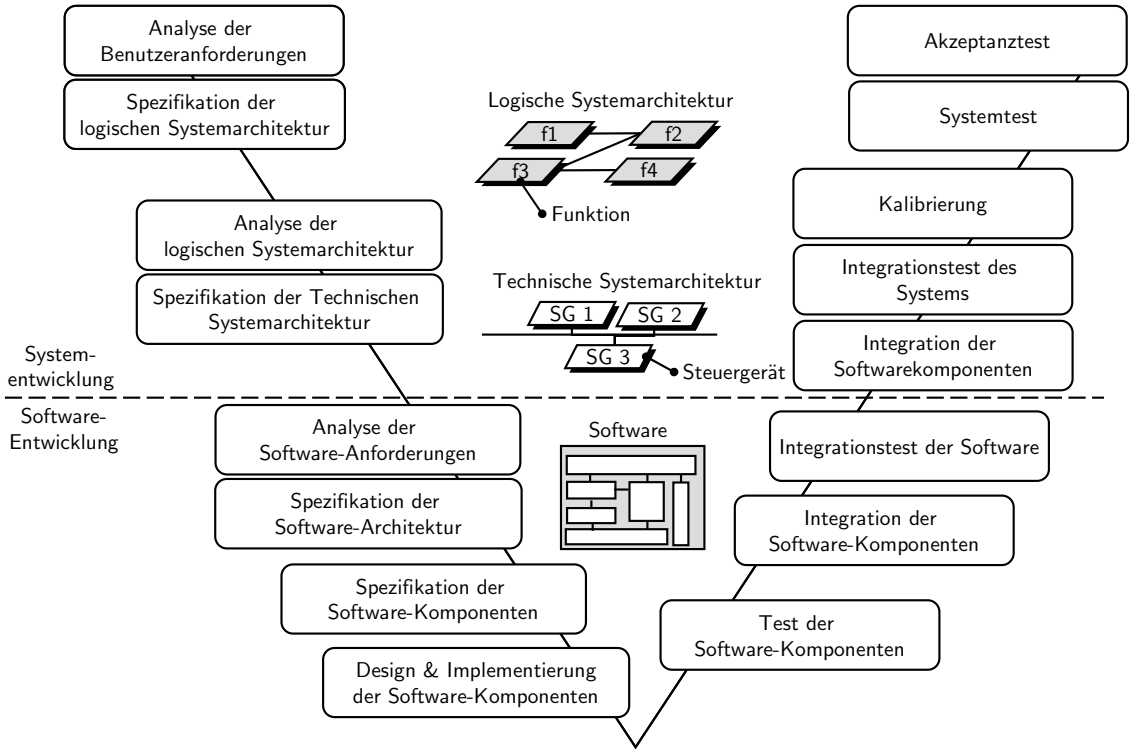


Abbildung 3.3: Ein Kernprozess für System- und Softwareentwicklung [Schäuffele et al. '03]

Durch den Kernprozess werden die Spezifikations- und Integrationsschnittstellen zwischen System- und Softwareentwicklung aufgezeigt. Diese Schnittstellen müssen wiederum von der verwendeten Softwareentwicklungsmethode berücksichtigt werden, um das Ziel eines durchgängigen Entwicklungsprozesses verwirklichen zu können. In den folgenden Kapiteln werden die

einzelnen Prozessschritte und Artefakte der System- und Softwareentwicklung etwas detaillierter vorgestellt. Dem Schwerpunkt dieser Arbeit entsprechend beschränkt sich diese Betrachtung auf den linken Zweig des V-Modells.

3.1.2.a Prozessschritte und Artefakte der Systementwicklung

Analyse der Benutzeranforderungen und Spezifikation der logischen Systemarchitektur

Ausgangspunkt sind die akzeptierten Benutzeranforderungen, die die Anforderungen der unterschiedlichen Benutzergruppen (z. B. Fahrer, Servicemitarbeiter oder Gesetzgeber) an das Fahrzeug in natürlicher Sprache und i. d. R. nicht technisch beschreiben. Hierbei kann zwischen funktionalen und nichtfunktionalen Anforderungen unterschieden werden. Während funktionale Anforderungen die Normal- und Fehlfunktionen des Systems beschreiben, enthalten nichtfunktionale Anforderungen Randbedingungen wie Varianten- und Skalierbarkeitsanforderungen, gesetzliche Bestimmungen (z. B. Sicherheitsanforderungen) oder auch Anforderungen bzgl. Wartbarkeit (siehe auch Kapitel 3.1.1.).

Im Rahmen dieses Prozessschrittes werden diese unstrukturierten Informationen in ein erstes formalisiertes und funktionales Modell überführt, in der die Anforderungen eindeutig, strukturiert und vollständig erfasst sind. Das resultierende Artefakt ist die logische Systemarchitektur. Diese hat sich als Zwischenschritt bei der Abbildung der Benutzeranforderungen auf eine technische Systemarchitektur insbesondere bei komplexen Systemen bewährt [Stevens et al. '98].

Bei der *logischen Systemarchitektur* handelt es sich um ein abstraktes logisches Modell des Systems und seiner Funktionen, das unabhängig von einer möglichen technischen Realisierung ist. Es dient als Bindeglied zwischen Benutzeranforderungen und der technischen Systemarchitektur und ist in der Sprache der Systementwicklung verfasst. Es handelt sich bei diesem Prozessschritt somit um einen kreativen Entwurfs- und Strukturierungsprozess, in dem das System schrittweise in logische Komponenten und Schnittstellen zerlegt wird. Über die Ausführungen von [Schäuffele et al. '03] hinausgehend, werden die möglichen Inhalte der logischen Systemarchitektur für diese Arbeit in Tabelle 3.1 weiter detailliert.

Zeitangaben werden in dem Kernprozess von [Schäuffele et al. '03] nicht für die logische Systemarchitektur erwähnt. Andere Ansätze erachten die Spezifikation von Abtastraten aber bereits in dieser frühen Phase als sinnvoll (siehe Kapitel 3.1.3.b und 3.2.2).

Funktions-netzwerk (FNetz)	Ein Netzwerk aus Funktionen (insb. Steuerungs- und Regelungs-funktionen), logischen Sensoren (bzw. Sollwertgeber) und Aktuatoren. Die Kanten zwischen diesen Elementen symbolisieren logische Kommunikationsbeziehungen. Die Elemente des Netzwerks können in einer Hierarchie organisiert sein (Kompositionen) und auf diese Weise das System in Subsysteme aufteilen.
Abstrakte Datenelemente (DElem)	Die über Kommunikationsbeziehungen zwischen Elementen des Funktions-netzwerks ausgetauschten Datenelemente werden implementierungs-unabhängig beschrieben.
Schnittstellen (Schnitts)	Datenelemente die Funktionen, logische Sensoren oder Aktuatoren empfangen oder senden können, werden zu einer Schnittstelle zusammen-gefasst.
Zeitangaben (ZAng)	Elemente des Funktionsnetzwerks werden mit Zeitangaben versehen, die für die Realisierung als Ober- oder Unterschranken zu verstehen sind. Beispielsweise kann für einen logischen Sensor eine Abtastrate spezifiziert werden.

Tabelle 3.1: Mögliche Inhalte der logischen Systemarchitektur

Analyse der logischen Systemarchitektur und Spezifikation der technischen Systemarchitektur

In diesem Prozessschritt wird die logische Systemarchitektur durch Treffen konkreter Realisierungsentscheidungen in eine *technische Systemarchitektur* überführt. Hierzu müssen die Funktionen und logischen Sensoren/Aktuatoren aus der logischen Systemarchitektur Komponenten aus der technischen Systemarchitektur zugeordnet werden. Dieser Vorgang wird auch *Partitionierung* oder *Mapping* genannt. Damit verbunden ist auch die Identifikation und Bewertung unterschiedlicher Realisierungsalternativen für die logische Systemarchitektur. Dabei müssen auch die erwähnten Randbedingungen (siehe Kapitel 3.1.1) berücksichtigt werden. Da es zwischen diesen Randbedingungen zu Zielkonflikten kommen kann, müssen diese evtl. durch das Treffen von Designentscheidungen aufgelöst werden (siehe Kapitel 3.1.1.). Im Einzelnen werden folgende Aktivitäten durchgeführt:

- **Analyse und Spezifikation der steuerungs- und regelungstechnischen Systeme**: Die Sensoren, Sollwertgeber, Aktuatoren und Steuerungs- bzw. Regelungseinheiten der logischen Systemarchitektur werden auf die realisierende Hardware abgebildet.
- **Analyse und Spezifikation von Echtzeitsystemen**: Für die unterschiedlichen Steuerungs- und Regelungsfunktionen werden die Abtastraten festgelegt. Aus diesen Abtastraten leiten sich sowohl die Echtzeitforderungen für die Software, als auch für das Kommunikations-system ab.
- **Analyse und Spezifikation verteilter und vernetzter Systeme**: Die logischen Software-funktionen werden Mikrocontrollern zugeordnet und die Signale werden zu Nachrichten

eines Kommunikationssystems zusammengefasst. Hierbei müssen Randbedingungen wie Rechen- oder Kommunikationsleistung berücksichtigt werden.

- **Analyse und Spezifikation zuverlässiger und sicherer Systeme**: In der Analysephase werden gefährliche Situationen analysiert und entsprechende Zuverlässigkeits- und Sicherheitsanforderungen spezifiziert. Daraus leiten sich auch bereits grundlegende Sicherheitsstrategien ab (z. B. notwendige Überwachungsfunktionen oder redundante Auslegung von Hard- und Software).

Mit dem Ende dieses Prozessschrittes sind somit die Hardware, die Abtastraten, die Kommunikationssysteme inkl. der wichtigsten Nachrichten sowie die grundlegenden Anforderungen und Maßnahmen für Sicherheit und Zuverlässigkeit definiert. Über die Ausführungen von [Schäuffele et al. '03] hinausgehend, werden die möglichen Inhalte der technischen Systemarchitektur für diese Arbeit wie in Tabelle 3.2 weiter detailliert.

E/E-Komponenten (EEKomp)	Liste der vorhandenen Sensoren, Sollwertgebern, Aktuatoren und Steuergeräte sowie einfacher Elektrik-Elektroniken.
Interner Steuergeräteaufbau (IntSG)	Die in einem Steuergerät eingesetzten Rechen- und Logikbausteine wie z. B. Mikroprozessoren, ASICs und FPGAs. Des Weiteren wird der verbaute Speicher wie z. B. RAM, ROM oder EEPROM modelliert. Außerdem können die internen Verbindungen zwischen diesen Bausteinen erfasst sein.
Kommunikationsverbindungen (KVerb)	Bussysteme und konventionelle Verbindungen (diskrete Verbindungen durch elektrische Leitungen) eines Fahrzeugs, die E/E-Komponenten verbinden. Für die beteiligten E/E-Komponenten werden die entsprechenden Anbindungen (Busanbindung, konventionelle Anbindung) beschrieben.
Buskommunikation (BKomm)	Botschaften und die enthaltenen Signale auf einem Bussystem inkl. der Sender- und Empfängersteuergeräte (K-Matrix). Verfügt das Netzwerk über Gateways, können auch die Routing-Tabellen erfasst sein.
Leistungsversorgung (LeistVers)	Batterien, Generatoren, Massestellen und Leitungsverteiler eines Fahrzeugs. Diese Elemente verfügen ebenso wie die E/E-Komponenten über Leistungsversorgungsanbindungen (Ein- und Ausgänge), die durch Leistungsversorgungsverbindungen miteinander verbunden werden.
Leitungen (Leit)	Pins und Stecker der Anbindungen sowie die Leitungen, die diese in einem Fahrzeug verbinden.
Fahrzeugtopologie (Topo)	Einbauorte eines Fahrzeugs in denen E/E-Komponenten verbaut werden können sowie die verbindenden Segmente, durch die Leitungen verlegt werden können.

Tabelle 3.2: Mögliche Inhalte der technischen Systemarchitektur

Eine mögliche Auswirkung der Leistungsversorgung, der Leitungen und der Fahrzeugtopologie auf die Softwareentwicklung wird im weiteren Verlauf der Arbeit nicht betrachtet. Daher entfallen diese Punkte bei der weiteren Diskussion.

Ein weiteres Ergebnis, das aus der Systementwicklung hervorgeht, ist die Liste der *Mappings*. Diese verknüpfen die logische und technische Systemarchitektur und dokumentieren so einerseits Realisierungsentscheidungen, fungieren aber andererseits auch als Navigationsstruktur für die Verfolgbarkeit (siehe Kapitel 1.1). In dieser Arbeit werden die in Tabelle 3.3 gezeigten *Mapping*-Kategorien als für die Softwareentwicklung relevant betrachtet.

Funktion ⇔ Mikroprozessor (F2Proz)	Funktionen werden mit einem Mikroprozessor verknüpft. Semantik: Zur Laufzeit führt dieser Mikroprozessor eine Software aus, die das Verhalten dieser Funktion implementiert.
Logischer Sensor/Aktuator ⇔ E/E-Komponente (SA2Komp)	Logische Sensoren/Aktuatoren werden mit ihren Hardware-Pendants verknüpft. Semantik: Diese E/E-Komponente realisiert den logischen Sensor/Aktuator.
Abstraktes Datenelement ⇔ Bussignal bzw. Botschaft (DE2Sig)	Ein abstraktes Datenelement wird mit einem Signal in einer Bus-Botschaft verknüpft. Semantik: Dieses Bussignal ist an der Realisierung der durch das abstrakte Datenelement definierten logischen Kommunikationsbeziehung beteiligt.

Tabelle 3.3: Für die Softwarteentwicklung relevante *Mappings* der Systementwicklung

3.1.2.b Prozessschritte und Artefakte der Softwareentwicklung

Die Softwareentwicklung beginnt zu einem Zeitpunkt, zu dem im Rahmen der Systementwicklung mit der logischen und technischen Systemarchitektur bereits umfangreiche Informationen bzgl. der Softwareanforderungen vorliegen. Die Analyse der Softwareentwicklung ist entsprechend eng mit den Modellen und Methoden der Systementwicklung verzahnt. Die genaue Ausprägung der Softwareentwicklung ist stark von der eingesetzten Entwicklungsmethode abhängig. In Kapitel 3.2.2 wird die Methode ROPES besprochen, die im Rahmen dieser Arbeit als Referenzmethode dient. Im Folgenden werden übersichtsartig die groben Prozessschritte und Artefakte erläutert, die für die Softwareentwicklung im Kernprozess vorgesehen sind und von der eingesetzten Softwareentwicklungsmethode entsprechend unterstützt werden müssen.

Analyse der Softwareanforderungen und Spezifikation der Softwarearchitektur

Die Analyse der Softwareanforderungen sammelt primär Festlegungen aus der technischen Systemarchitektur, die Auswirkung auf das Design und die Implementierung der Software haben. Dazu gehört die Analyse des zugewiesenen Steuergeräts (z. B. Prozessor, Busanschlüsse), die technische Vernetzung mit interagierenden Funktionen und Zeitanforderungen, die sich aus Abtastraten ableiten. Auf dieser Basis wird die Softwarearchitektur eines Steuer-

geräts spezifiziert. Zu den Architekturentscheidungen gehören die Wahl des Schichtenmodells, die Festlegung von Komponenten und die Bereitstellung von Tasks. Hier kann auf Standardarchitekturen wie z. B. OSEK [OSEK/VDX '05] oder in Zukunft Autosar [AUTOSAR] zurückgegriffen werden. Eine weitere Designentscheidung, die unabhängig von den einzelnen Softwarefunktionen in dieser Phase getroffen wird, sind die möglichen Betriebszustände des Steuergeräts. So werden z. B. für die Parametrierung oder das Software-Update besondere Betriebszustände benötigt, bei denen die Überwachungsfunktionen abgeschaltet sind. On- und Off-Board-Schnittstellen können ebenfalls Gegenstand von Architekturentscheidungen sein.

Spezifikation der Softwarekomponenten

Mit diesem Prozessschritt wechselt die Entwicklungsperspektive von der Architektur zu der Spezifikation der Interna einer einzelnen Komponente. Hierbei lassen sich folgende Aktivitäten identifizieren:

- **Spezifikation des Datenmodells**: Die zu verarbeitenden Daten werden abstrakt (Skalar, Vektor oder Matrix) und unabhängig von konkreten Implementierungstypen spezifiziert. Beispielsweise werden die grundlegenden Datenstrukturen für Kennlinien oder Kennfelder festgelegt.
- **Spezifikation des Verhaltensmodells**: Der interne Daten- und Kontrollfluss einer Komponente wird spezifiziert. Auf diese Weise ergibt sich der Weg der Dateninformation zwischen einzelnen Anweisungen und Kontrollstrukturen für die Ausführung dieser Anweisungen.
- **Spezifikation des Echtzeitmodells**: Die Anweisungen einer Softwarekomponente werden Prozessen bzw. Tasks zugeordnet. Hier wird entschieden, welche Anweisungen in einer möglichen Initialisierungstask ausgeführt werden und wie das Ausführungsmodell in den einzelnen Betriebszuständen aussieht. Hierbei kann unterschieden werden, ob die Prozesse periodisch oder episodisch ausgeführt werden.

Design und Implementierung der Softwarekomponenten

In den bisherigen Prozessschritten wurde soweit wie möglich von der Implementierungsplattform abstrahiert. Beim Design und der Implementierung müssen diese Abstraktionen durch entsprechende Designentscheidungen aufgelöst werden. Eine besondere Rolle spielen hier die geforderten nichtfunktionalen Produkteigenschaften. Aus Anforderungen nach Variabilität leitet sich z. B. ein Design ab, bei dem Programm- und Datenstand deutlich getrennt werden. Der Forderung nach möglichst minimalen Hardwareressourcen muss durch eine entsprechende Optimierung bei den Datenstrukturen und Algorithmen begegnet werden.

Entsprechend muss beim Design und der Implementierung des Datenmodells festgelegt werden, ob die einzelnen Daten Variablen und unveränderliche Parameter sind. Des Weiteren muss eine Abbildung der physikalischen Spezifikation auf einen Implementierungsdatentyp erfolgen. Bei Kennlinien und Kennfeldern muss z. B. festgelegt werden, welches tabellarisches Ablageschema, und welche Interpolationsmethode verwendet wird. Entsprechend dem zu erwartenden Zugriff

muss entschieden werden, in welchem Speichersegment die Daten abgelegt werden. Beim Design und der Implementierung des Verhaltensmodells müssen u. a. Genauigkeitsfragen (z. B. in Bezug auf Rundungsfehler) geklärt werden.

3.1.3 Modellbasierte Entwicklung

Eine eng verzahnte System- und Softwareentwicklung, in der z. B. verteilte Funktionen mit Sicherheits- und Zuverlässigkeitsanforderungen interdisziplinär realisiert werden, muss mit einem hohen Maß an Komplexität durch Abhängigkeiten umgehen. Die Informatik beschäftigt sich seit ihrer Entstehung mit der Fragestellung, wie Komplexität beherrschbar gemacht werden kann. Bekannte Prinzipien sind z. B. *Formalisierung* (engl. *rigor and formality*), *Trennung unterschiedlicher Belange* (engl. *separation of concerns*) und *Abstraktion* (engl. *abstraction*) [Ghezzi et al. '91].

Ein Ansatz, der diese Prinzipien adressiert und auch von neueren Systementwicklungsmethoden propagiert wird [Schäuffele et al. '03], ist die *Modellbildung*. Hierbei handelt es sich um einen Abstraktionsprozess, mit dem man versucht, die Komplexität eines Systems auf ein „bewältigbares“ Maß zu reduzieren. Grundlage ist eine Modellierungssprache, die für modellierungsrelevante Begriffe feste Bezeichner definiert, deren Semantik möglichst genau beschreibt und mögliche Beziehungen zwischen diesen Begriffen festlegt. Ergebnis ist ein *Modell* des Systems, in dem die entscheidenden Eigenschaften/Funktionalitäten des Systems enthalten sind [IESE b].

Darüber hinaus enthält eine Modellierungssprache oft eine graphische Notation, die festlegt, wie die Begriffe und Beziehungen eines Modells graphisch in einem *Diagramm* zu visualisieren sind. Dabei wird versucht, die graphische Symbolik so zu wählen, dass sie von der Zielgruppe möglichst intuitiv verwendbar ist. Ein Diagramm kann bei einigen Modellierungssprachen auch nur Ausschnitte des Modells enthalten, was dazu genutzt wird, für unterschiedliche Fragestellungen unterschiedliche Diagramme zu erstellen, die jeweils nur die relevanten Modellelemente enthalten.

Modelle und ihre Diagramme können auf unterschiedliche Fragestellungen fokussieren und dabei von anderen Details abstrahieren und somit zu einem gemeinsamen Problem- und Lösungsverständnis zwischen verschiedenen Fachdisziplinen beitragen [Schäuffele et al. '03]. Weitere Mehrwerte von formalisierten Modellen liegen z. B. in den Möglichkeiten der Simulation, Code-Generierung und *Rapid Prototyping* [Bertram et al. '01].

Die konsequente Weiterentwicklung der modellbasierten Entwicklung führte schließlich zu dem Ansatz von *modellgetriebener Entwicklung*. Hier wird in jeder Entwicklungsphase ein hochformalisiertes Modell als das zentrale Entwicklungsartefakt verwendet. Aufgrund des hohen Formalisierungsgrades ist es teilweise möglich, die Beziehungen zwischen den Modellen zu beschreiben und dies für die automatische Gewinnung von Systembestandteilen bzw. weiteren Modellsichten auf das System auszunutzen. Dieser Vorgang der automatisierten Konstruktion eines Zielmodells auf der Basis eines Quellmodells wird allgemein als *Modelltransformation* bezeichnet (siehe Kapitel 3.3). Wenn in allen Entwicklungsphasen Modelle verwendet werden

und diese automatisiert Informationen austauschen bzw. wiederverwenden, spricht man auch von einem *durchgängig modellbasierten Entwicklungsprozess*.

Modellbasierte Ansätze gibt es sowohl in der System- als auch in der Softwareentwicklung (siehe Kapitel 3.1.3.b und 3.1.3.c). Wenn es gelingt, die Modelle dieser beiden Bereiche zusammenzuführen, kann von einem durchgängigen modellbasierten Entwicklungsprozess nach dem oben beschriebenen Verständnis gesprochen werden.

3.1.3.a Metamodelle

Eine wichtige Rolle bei der Herstellung eines durchgängig modellbasierten Entwicklungsprozesses spielen *Metamodelle*. Ein Metamodell ist eine Definition einer Modellierungssprache, also ein Modell, das ein Modell beschreibt. Um festzulegen, wie solche Metamodelle aufgebaut sein dürfen, kann für sie wiederum ein Metamodell entwickelt werden, das dann als *Metametamodell* bezeichnet wird. Im Allgemeinen wird gefordert, dass ein solches Metametamodell sich selbst definiert und nicht wieder auf ein Metamodell zurückgreift. Prinzipiell kann der Vorgang der „Metaisierung“ jedoch beliebig oft wiederholt werden. Die hier besprochenen Konzepte und Standards stammen aus dem Umfeld modellbasierter Softwareentwicklung, werden aber auch für Metamodelle der Systementwicklung verwendet.

In Analogie zur Definition von Programmiersprachen wird oftmals davon gesprochen, dass die durch ein Metametamodell definierbaren Metamodelle eine *abstrakte Syntax* für Modelle beschreiben, die eine notationsunabhängige Grammatik der verwendbaren Konzepte darstellt. Ein Metamodell stellt somit eine Sprache dar. Um eine Einschränkung vorzunehmen, welche mit der Grammatik erzeugbaren Ausdrücke inhaltlich sinnvoll sind – im Allgemeinen als *statische Semantik* bezeichnet – werden sog. *Wohlgeformtheitsregeln* eingeführt, die beispielsweise Bedingungen wie Zyklenfreiheit beschreiben. Die *dynamische* oder *operationale Semantik*, die die Bedeutung von wohlgeformten Ausdrücken festlegt, wird üblicherweise nur als *Semantik* bezeichnet und meist in natürlicher Sprache ausgedrückt. Die abstrakte Syntax der Sprache kann durch verschiedene *konkrete Syntaxen* realisiert werden: zum einen durch graphische Notationen (meist verschiedene Arten von Diagrammen), zum anderen durch textuelle Darstellungen. Details der Realisierung werden dann durch ein *Mapping* definiert, das Elemente aus der abstrakten Syntax mit den zugehörigen Notationselementen aus der konkreten Syntax verknüpft. Die gesamte Beschreibung eines Metamodells beinhaltet somit die abstrakte Syntax, statische und dynamische Semantik sowie eventuelle konkrete Notationen mit *Mappings*. Mit dem Begriff „Metamodell“ selbst ist allerdings überwiegend nur die abstrakte Syntax gemeint.

Mit Hilfe der konkreten Notationen, die sich aus einem Metamodell ableiten, können Modelle erstellt werden. Im Falle einer Modellierungssprache für Softwareentwicklung beschreibt ein solches Modell z. B. ein bestimmtes Softwaresystem. Die zur Laufzeit existierenden Daten oder Objekte dieses Systems können als Instanz dieses Modells betrachtet werden.

Die in Kapitel 3.2.1 beschriebene Modellierungssprache UML ist in die in Tabelle 3.4 gezeigte Vier-Schichten-Metamodellarchitektur eingebettet, die ein Beispiel für das Konzept der sich

selbst definierender Metametamodelle liefert. Die einzelnen Ebenen werden hier mit M0 bis M3 abgekürzt. Im Rahmen von Kapitel 3.1.3.a wird noch auf diverse OMG-Standards zu diesen Ebenen eingegangen.

Schicht	**Kürzel**	**Beschreibung**	**Beispielelement**
Metametamodell	M3	Definiert die Sprache für die Spezifikation von Metamodellen und sich selbst.	`MetaClass`, `MetaAttribute`, `MetaOperation`
Metamodell	M2	Instanz eines Metametamodells, definiert die Sprache für die Spezifikation von Modellen.	`Class`, `Attribute`, `Operation`, `Component`
Modell	M1	Instanz eines Metamodells, definiert die Sprache für die Beschreibung eines Systems oder Domäne.	`Class:Fahrzeug`, `Attribute:geschwindigkeit`
Objekte	M0	Instanzen des Modells. Beschreiben Systeme zur Laufzeit oder Ausprägungen einer bestimmten Domäne.	`geschwindigkeit=80`

Tabelle 3.4: Vier-Schichten-Metamodellarchitektur (Darstellung angelehnt an [IESE c])

3.1.3.b Modellbildung in der Systementwicklung

Die ersten elektronischen Systeme im Fahrzeug waren relativ autonom und konnten daher auch weitgehend isoliert entwickelt werden. Die E/E-Systementwicklung beschränkte sich daher auf Fragen wie Bauraum oder Verkabelung. Mit der Einführung von Bussystemen, wie z. B. CAN, erweiterte sich das Aufgabengebiet auf die Festlegung der Netzwerktopologie und die Definition der Kommunikationsmatrix (beschreibt für jeden Netzknoten die ein- und ausgehenden Nachrichten).

Aufgrund der noch relativ geringen Bedeutung der Systementwicklung war die Modellbildung im Bereich der Automobilindustrie daher wenig standardisiert (abgesehen von einigen wenigen Dateiformaten für die Kommunikationsmatrix). Mit dem in der Einleitung aufgezeigten Trend, die einzelnen Funktionen auch funktional über mehrere Steuergeräte hinweg zu vernetzen, hat die Notwendigkeit einer formalisierten und standardisierten Modellbildung jedoch zugenommen. Es gibt daher einige Initiativen die versuchen, die Artefakte der unterschiedlichen Fachdisziplinen aus dem Kernprozess zu formalisieren und deren Zusammenhang zu beschreiben. Ziel ist ein Metamodell für den gesamten Kernprozess, das für jedes Artefakt die zulässigen Modellierungselemente der Fachdisziplin beschreibt und festlegt, wie diese mit den Modellierungselementen anderer Fachdisziplinen in Beziehung gebracht werden können. Diese als *Mappings* bezeichneten Verknüpfungen zwischen den unterschiedlichen Modellen sind ein

zentrales Element, um die für die Komplexitätsbeherrschung so wichtige Verfolgbarkeit herzustellen. In den folgenden Unterkapiteln werden Modellierungssprachen für die Systementwicklung ausführlicher vorgestellt.

EAST-ADL

Die *EAST-EEA Architecture Description Language* (EAST-ADL) [Lönn et al. '04] ist eine Modellierungssprache für die Beschreibung elektronischer Architekturen im Automobilbereich. Sie wurde im Rahmen des von der Europäischen Union geförderten Forschungsprojekts *EAST-EEA* von bedeutenden Automobilherstellern und Zulieferern entwickelt. Die in diesem Projekt ebenfalls begonnen Überlegungen zu einer offenen Standardarchitektur für Software in Kraftfahrzeugen mündete in der Entwicklung des Industriestandards *Autosar*. Das im Rahmen der Autosar-Aktivitäten entwickelte Metamodell stellt für manche Teile der hier vorgestellten EAST-ADL eine Weiterentwicklung dar (siehe nächstes Unterkapitel), lässt dabei jedoch insbesondere die Teile aus, die der logischen Systemarchitektur hinzugerechnet werden können.

Die EAST-ADL betrachtet die Systemarchitektur mit Hilfe unterschiedlicher Artefakte, die miteinander verlinkt sind. Abbildung 3.4 zeigt diese Artefakte im Überblick.

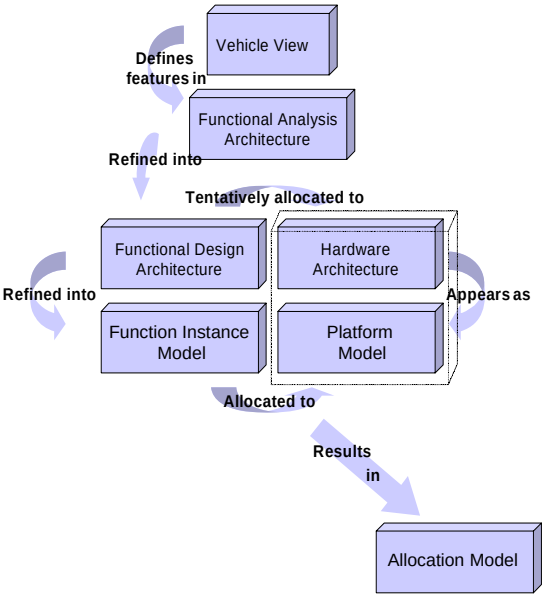


Abbildung 3.4: Artefakte und deren Beziehung in der EAST-ADL [Lönn et al. '04]

Die Artefakte *Vehicle View*, *Functional Analysis Architecture*, *Functional Design Architecture* und *Function Instance Model* stehen in der Vertikalen in einer Verfeinerungsbeziehung. Auf dieser Achse werden Funktionen ausgehend von Benutzeranforderungen in Richtung realisierender Softwarekomponenten verfeinert, wobei die Hardware weitgehend unberücksichtigt bleibt. Auf der Horizontalen stehen die einzelnen Artefakte in einer Allokations- oder Mapping-Beziehung. Die realisierungsunabhängig definierten Funktionen werden hier der ausführenden Hardware eines Fahrzeugs zugeordnet. Die Verlinkung zwischen *Function Instance*

Model und *Platform Model* ist sehr umfangreich und wird in einem eigenen Modell, dem *Allocation Model*, beschrieben. Die Artefakte im Einzelnen:

- **Vehicle View***:* Ein *Requirements*-Modell, das unter anderem die kundenerlebbaren Funktionen in Form von *Features* (Merkmalen) beschreibt. Für die detaillierte Beschreibung der Anforderungen an ein *Feature* wird auf Modellierungselemente der *SysML* [OMG a] zurückgegriffen. Außerdem können die möglichen Varianten bei der Kombination dieser *Features* für ein Fahrzeug modelliert werden.
- **Funktional Analysis Architecture**: Beschreibt ein Netz aus logischen Funktionen (*AnalysisFunctions*) und logischen Sensoren/Aktuatoren (*FunctionalDevice*s). Diese Elemente können über *Ports*, die wiederum über *Connector*s verbunden sind, kommunizieren. Ein Port ist über ein Interface getypt, das wiederum Signale (*ConnectorSignal*) spezifiziert, die einen abstrakten Typ (*DesignDataType*) haben. In der *FunctionalDesignArchitecture* wird ein Implementierungstyp (*ImplementationDataType*) hinzugefügt. Mit einem Objekt vom Typ *TypeAssoziation* wird die Abbildung zwischen beiden Typen beschrieben (z. B. Skalierung und Offset). Für *Port*s können Zeitangaben (z. B. Zykluszeit) gemacht werden, die als Anforderung zu verstehen sind. Eine *AnalysisFunction* kann eine Komposition aus weiteren *AnalysisFunctions* sein. Auf diese Weise kann das Funktionsnetz in einer Hierarchie organisiert werden.
- **Functional Design Architecture**: Beschreibt Softwarekomponenten und ihre Subfunktionen. Es wird zwischen Kompositionen (*CompositeSoftwareFunction*) und elementaren Softwarefunktionen (*ElementarySoftwareFunction*) differenziert. *CompositeSoftwareFunction*s können weitere *CompositeSoftwareFunction*s oder *ElementarySoftwareFunction*s enthalten. *ElementarySoftwareFunctions* können nicht weiter unterteilt werden und lassen sich somit später einem Steuergerät zuordnen. Bei der Kommunikationsmodellierung greift die *Functional Design Architecture* auf die aus der *Functional Analysis Architecture* bekannten *Port*s, *Connector*s und *Interfaces* zurück. Ein *ConnectorSignal* kann hier aber mit dem Implementierungstyp (*ImplementationDataType*) getypt sein. Zwischen *DesignDataType* und *ImplementationDataType* kann eine Abbildung beschrieben werden. Ports und Funktionen verfügen über Attribute, mit denen das zeitliche Verhalten spezifiziert werden kann (z. B. Zykluszeit). Über einen *LocalDeviceManager* kann mit der Hardware kommuniziert werden.
- **Function Instance Model:** Beschreibt die Instanziierung von Funktionen für ein konkretes Fahrzeug. Dazu werden *ElementarySoftwareFunction*s mit gleichen zeitlichen Anforderungen (für die spätere Zuordnung zu Tasks) zu Clustern zusammengefasst und somit die logische Hierarchie und die Variabilität aufgelöst. Für Funktionen, die über Cluster-Grenzen hinweg kommunizieren müssen, werden Signalinstanzen zwischen den entsprechenden Clustern angelegt.
- **Hardware Architecture**: Beschreibt die Hardware für Steuergeräte (*ECU*), Sensoren und Aktuatoren sowie die Kommunikationskanäle (*Channel*), die diese verbinden. Ein *Channel* kann vom Typ „electrical“, „CAN“ oder weiteren Bustechnologien sein. Der interne Aufbau

eines Steuergeräts kann mit Elementen wie *Processor*, *Memory* und *Peripheral* (Peripherie, z. B. auch Bus-Transceiver) beschrieben werden. Außerdem ist es möglich, für Steuergeräte Pins anzulegen und diese mit Leitungen (*Wire*s) zu verbinden. Hier kann aber offenbar kein Bezug zu *Channel*-Elementen hergestellt werden, so dass es z. B. nicht möglich ist, einen Bus später durch Pins und Leitungen detaillierter zu beschreiben. Leitungen haben zudem keine Attribute.

- **Platform Model**: Beschreibt das Betriebssystem (*OS*), Funktionen für die Abstraktion der Hardware (*HAF*) und Funktionen der Middleware. Diese Ebene ist stark durch die EAST-Laufzeitumgebung geprägt (siehe unten).
- **Allocation Model:** Beschreibt die letztendliche Laufzeitarchitektur, indem die bisherigen Artefakte verlinkt werden. Für Prozessoren aus der *Hardware Architecture* werden *OSTask*s gebildet, denen *Cluster* aus dem *Function Instance Model* zugewiesen werden. Für die Kommunikation zwischen Tasks werden Botschaften definiert, die Signalinstanzen aus dem *Function Instance Model* enthalten. Die Botschaften sind wiederum *Channels* (z. B. vom Typ „CAN“) aus der *Hardware Architecture* zugeordnet.

Zu den besonderen Merkmalen der EAST-ADL gehört ein Variantenmanagement, mit dem es möglich ist, durch das Setzen von Variationspunkten die Variabilität eines Modellelements in Hinblick auf Variabilitätsdimensionen wie z. B. Baureihe, Aufbau (z. B. Limousine versus Kombi), Ausstattungsumfang oder Markt (z. B. USA versus Japan) zu beschreiben. Mit Hilfe weiterer Mechanismen, die im Metamodell definiert sind, soll es möglich sein, die mit diesen Dimensionen verbundene Variabilität ausgehend von den Anforderungen (*Vehicle View*) durchgängig über alle Artefakte hinweg konsistent zu beschreiben.

Für viele Elemente gibt es Typen, was die Wiederverwendung von Modellelementen über mehrere Projekte hinweg fördert. So werden beispielsweise Kommunikationsports von Funktionen/Komponenten durch mehrfach referenzierbare Interfaces typisiert. Softwarekomponenten in der *Functional Design Architecture* werden verwendungsunabhängig beschrieben und erst in dem *Function Instance Model* für ein konkretes Fahrzeugprojekt instanziiert.

Ein weiteres Merkmal ist das Konzept, den Aufbau und die Kommunikationsbeziehungen von Softwarekomponenten weitgehend unabhängig von der technischen Plattform zu definieren. Diese Vorgehensweise soll durch eine besondere Laufzeitumgebung (engl. *runtime environement*) möglich werden, die auf jedem Steuergerät verfügbar ist. Diese stellt umfangreiche standardisierte Basisdienste für eine Softwarekomponente zur Verfügung, z. B. für den Zugriff auf Hardware und die Kommunikation mit anderen Softwarekomponenten. Aus Sicht einer Softwarekomponente ist die Realisierung der Kommunikation mit einer anderen Softwarekomponente dann weitgehend transparent. Es ist nicht direkt ersichtlich, ob der Kommunikationspartner auf einem anderen Steuergerät platziert ist und der Datenaustausch über einen Bus läuft, oder ob der Kommunikationspartner auf demselben Steuergerät platziert ist und die Kommunikation intern über einen gemeinsam benutzten Speicherbereich realisiert wird. Mit diesem Konzept sollen Komponenten wiederverwendbar werden und flexibel im Fahr-

zeug auf unterschiedliche Steuergeräte verteilt werden können. Die Spezifikation der dazu nötigen Laufzeitumgebung wurde ebenfalls im Projekt EAST-EEA begonnen und wird im Projekt Autosar (siehe unten) fortgeführt.

Die Modelle der EAST-ADL haben die Entwicklung der E/E-Architektur des Gesamtfahrzeugs im Blick und passen somit zum Betrachtungsrahmen der Systementwicklung. Tabelle 3.5 ordnet einige Elemente aus dem Metamodell der EAST-ADL den in Tabelle 3.1 - Tabelle 3.3 definierten Inhalten der Systementwicklung zu. Die Elemente für die Inhalte der logischen und technischen Systemarchitektur stammen überwiegend aus den Artefakten *Functional Analysis Architecture* und *Hardware Architecture*.

Inhalt nach Kernprozess (siehe Tabelle 3.1 - Tabelle 3.3)			**Mögliche Elemente der EAST-ADL**
Logische Systemarchitektur	1	Funktionsnetzwerk	*AnalysisFunction, FunctionalDevice, FunctionPort*
	2	Abstrakte Datenelemente	*DesignDataType, ConnectorSignal*
	3	Schnittstellen	*SignalFunctionPort*, *SignalFunctionPortInterface*
	4	Zeitangaben	Attribute *Period* und *TransferTime* von *FunctionPort*
Technische Systemarchitektur	1	E/E-Komponenten	*ECU*, *Sensor*, *Actuator*
	2	Interner Steuergeräteaufbau	*Processor*, *Memory*, *Peripheral*
	3	Kommunikationsverbindungen	*Channel*
	4	Buskommunikation	*SignalInstance*, *Frame*
	5	Leistungsversorgung	-
	6	Leitungen	*Pin*, *Wire*
	7	Fahrzeugtopologie	-
Mappings	1	Funktion ⇔ Mikroprozessor	Kette: *AnalysisFunction – ElementarySoftwareFunction – FunctionInstance – LogicalCluster – OSTask – Processor*
	2	Logischer Sensor/Aktuator ⇔ E/E-Komponente	Kette: *FunctionalDevice – LocalDeviceManager –* (*Sensor*/*Actuator*/*Peripheral*)
	3	Abstraktes Datenelement ⇔ Bussignal bzw. Botschaft	Kette: *ConnectorSignal – SignalInstance – Frame – Channel* Zusätzlich: D*esignDataType – TypeAssociation – ImplementationDataType*

Tabelle 3.5: Mögliche Elemente der EAST-ADL für die logische und technische Systemarchitektur sowie für die *Mappings*

Die logische Systemarchitektur kann auf der Seite der EAST-ADL durch die *Functional Analysis Architecture* abgebildet werden. In Abbildung 3.5 ist ein kleiner Ausschnitt einer möglichen *Functional Analysis Architecture* für das KFS aus Kapitel 2 dargestellt, der inhaltlich aus Abbildung 2.1 abgleitet ist. Ein *Functional Device* mit Namen `Tempomathebel` verfügt über einen Sender-Port mit dem Namen `Benutzerkommando`. Der Port ist durch das Interface `IF_Benutzerkommando` getypt, das die Beschreibung eines abstrakten Signals für die Übermittlung von Benutzerkommandos enthält (die Interface-Definition ist nicht abgebildet). Die *Analysis Function* mit dem Namen `Tempomat/Speedtronic` verfügt über einen entsprechenden Empfänger-Port, der über einen *Connector* mit dem Sender-Port verbunden ist. Auf diese Weise wird der abstrakte Signalfluss zwischen den beiden logischen Einheiten `Tempomathebel` und `Tempomat/Speedtronic` ausgedrückt.

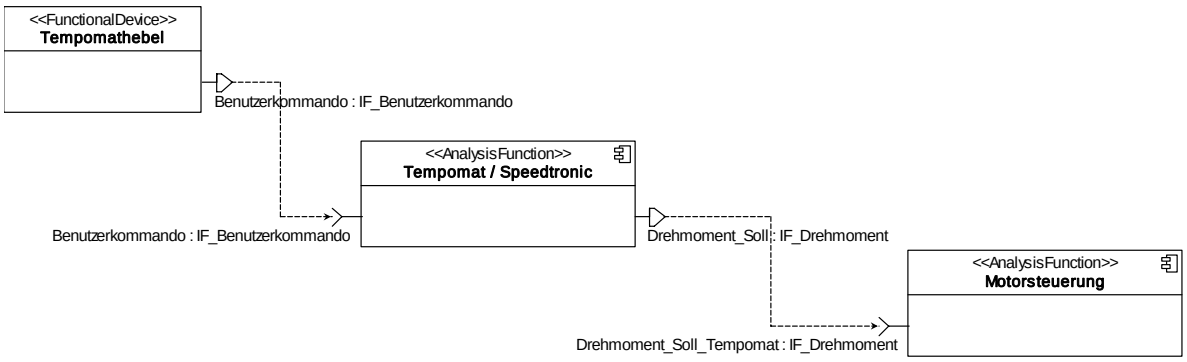


Abbildung 3.5: Ausschnitt einer möglichen *Funktional Analysis Architecture* für das KFS

Für die Darstellung der Inhalte aus der technischen Systemarchitektur bietet sich vor allem die *Hardware Architecture* an. Abbildung 3.6 zeigt einen Ausschnitt einer möglichen *Hardware Architecture* für das KFS, der aus Abbildung 2.2 abgleitet ist.

Berücksichtigt man, dass die technische Systemarchitektur auch die Auslegung der K-Matrix beschreibt, so lassen sich z. B. auch in den Artefakten *Allocation Model* und *Function Instance Model* viele Inhalte finden, die der technischen Systemarchitektur hinzuzurechnen sind. Die unterschiedlichen Artefakte der EAST-ADL folgen somit der im Kernprozess verankerten Idee, zwischen einer logischen und technischen Perspektive zu trennen, unterteilen diese aber weiter. Auf eine Prozessbeschreibung für die Nutzung der EAST-ADL wird in der Spezifikation verzichtet. Die Artefakte wurden mit dem Ziel entwickelt, kompatibel mit den in der Branche üblichen Entwicklungsprozessen zu sein.

Die EAST-ADL zeigt, in welch großem Umfang formalisierte Informationen bei einer intensiven Systementwicklung zur Verfügung stehen können, die für die Softwareentwicklung relevant sind. Im Bereich der mit der technischen Systemarchitektur verwandten Artefakte wird die EAST-ADL im Projekt *Autosar* [AUTOSAR] bereits weiterentwickelt und standardisiert. Bei den Artefakten, die der logischen Systemarchitektur hinzugerechnet werden können (*Functional Analysis Architecture*), verharrte das Metamodell auf dem Stand des Forschungsprojekts EAST-EEA. Im Rahmen des Projekts ATESST [ATESST] wird aktuell eine Version 2 der EAST-ADL

in der Form eines UML2-Profils (siehe Kapitel 3.2.1.b) entwickelt. In dieser Version sollen die durch Autosar standardisierten Inhalte übernommen und mit den übrigen Artefakten harmonisiert werden. Dennoch liefert die EAST-ADL eine präzise und umfassende Beschreibung der Konzepte in der Systementwicklung der Automobilindustrie und kann als Referenz für aufbauende Arbeiten, wie der hier vorliegenden, herangezogen werden. In Kapitel 3.2.2.g werden die einzelnen Modelle der EAST-ADL zusammen mit Modellen der Softwareentwicklungsmethode ROPES den Artefakten des Kernprozesses aus Kapitel 3.1.2 zugeordnet.

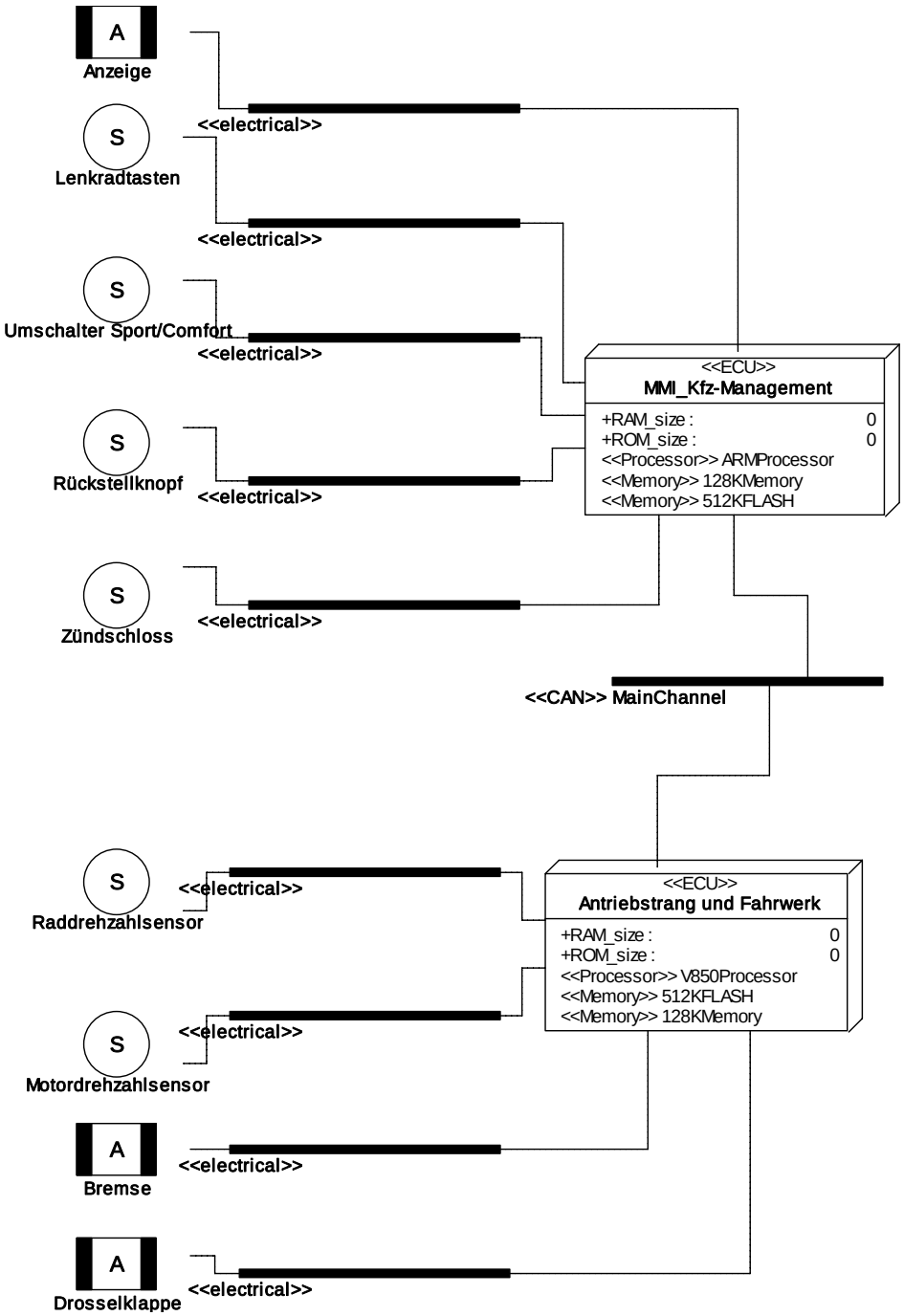


Abbildung 3.6: Ausschnitt einer möglichen *Hardware Architecture* für das KFS

E/E-Konzepttool

Im Umfeld der DaimlerChrysler AG und des Forschungszentrums Informatik (FZI) in Karlsruhe entstand das *E/E-Konzept-Tool* [Belschner et al. '05], das mittlerweile unter dem Namen *PREEvision* von der Firma *Aquintos* kommerziell vertrieben wird [Aquintos]. Dabei handelt es sich um ein modellbasiertes Werkzeug für Entwurf, Dokumentation und Bewertung von E/E-Architekturen. Einsatzschwerpunkte sind die frühen Entwicklungsphasen (insb. die sogenannte „Konzeptphase") in der der E/E-Architekt das Gesamtfahrzeug betrachtet und die grundlegenden Architektur-Entscheidungen zu fällen hat. Abbildung 3.7 zeigt die Modellinhalte des E/E-Konzept-Tools in Form einer Schichtenarchitektur.

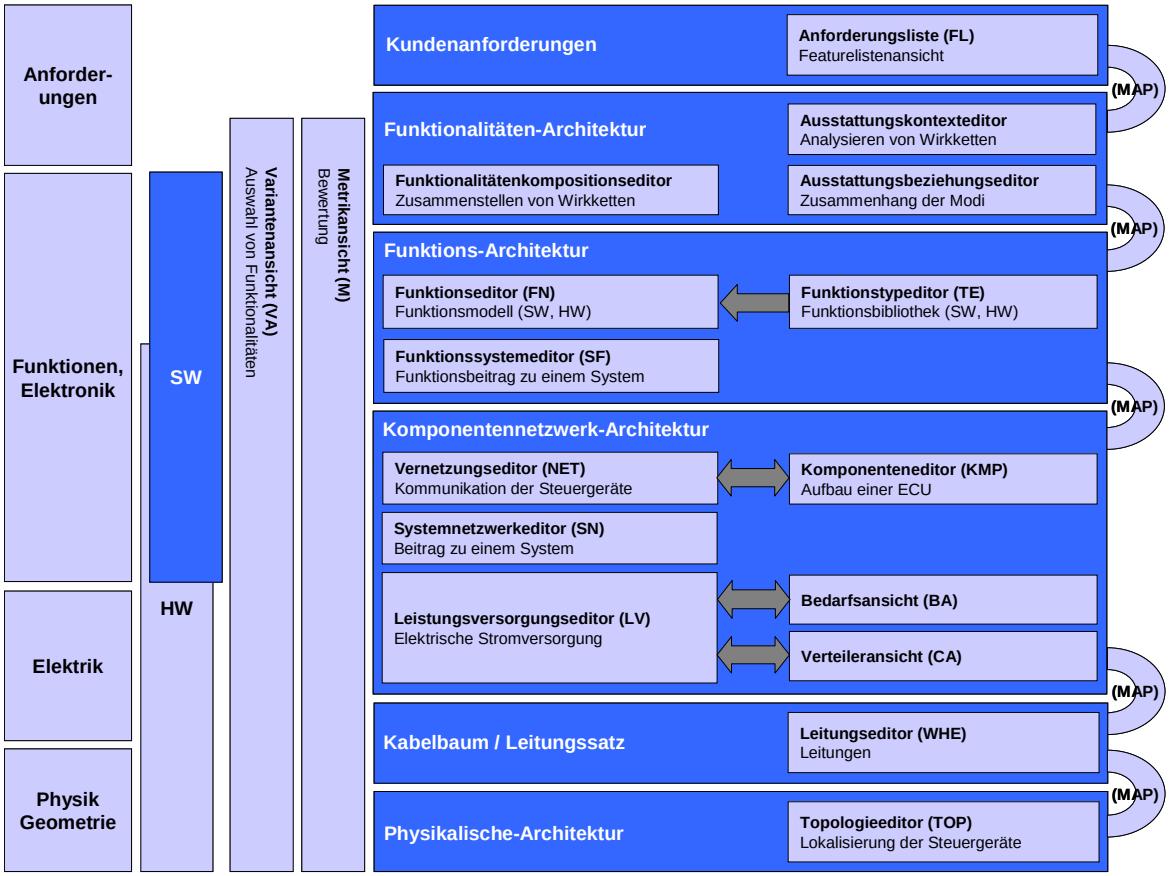


Abbildung 3.7: Modellinhalte des E/E-Konzept-Tools [Aquintos '06]

Die Inhalte der einzelnen Modelle sind auf der Vertikalen durch *Mappings* verknüpft. Da es auch *Mappings* gibt, die einzelne Schichten überspringen, handelt es sich jedoch nicht um eine strikte Schichtenarchitektur. Ein solches *Mapping* existiert z. B. zwischen den Kundenanforderungen und der Komponentennetzwerkarchitektur und trägt der Tatsache Rechnung, dass nicht in jedem Projekt die Funktionalitäten- und Funktionsarchitektur modelliert werden. Im Folgenden sind die Inhalte der einzelnen Modelle kurz erläutert:

- **Kundenanforderungen**: Die Kundenanforderungen werden in Form von *Features* aus einem Anforderungsmanagementwerkzeug importiert. Sie fungieren als Ausgangspunkt für

die *Mappings,* um alle weiteren Modellelemente bis auf die Anforderungen zurückverfolgen zu können.

- **Funktionalitäten-Architektur**: Ein Modell um Wirkketten von *Features* zu analysieren. Die dahinter stehende Methode wurde insbesondere mit der Zielstellung entwickelt, unterschiedliche Architekturen, die ähnliche Funktionen realisieren, systematisch vergleichen zu können.
- **Funktions-Architektur**: Beschreibt ein Netz aus Funktionen, logischen Sensoren und Aktuatoren, in dem Datenelemente und Operationsaufrufe ausgetauscht werden. Das Metamodell orientiert sich an dem *Virtual Function Bus* von Autosar.
- **Komponentennetzwerkarchitektur**: Beschreibt die Hardware von E/E-Komponenten und Verbindungen (Bussystem und konventionelle Verbindungen) die diese verbinden. In diese Architektur integriert ist ein Modell, in dem die Busbotschaften und die darin enthaltenen Signale und Routingtabellen von Gateways modelliert werden. Bzgl. des internen Aufbaus eines Steuergeräts können z. B. Prozessoren, verschiedene Speicherarten und Verbindungen zwischen diesen modelliert werden. Eine E/E-Komponente kann mit Steckern und Pins versehen werden, die mit unterschiedlichen Leitungstypen verbunden werden können. Die Leistungsversorgung (Stromversorgung) ist ebenfalls Teil des Modells und umfasst z. B. die Beschreibung von Batterien, Generatoren, Leistungsverteilern und den entsprechenden Leistungsversorgungsleitungen.
- **Physikalische Architektur**: Beschreibt die Bauräume und ihre Einbauorte eines Fahrzeugs, in denen E/E-Komponenten verbaut werden können sowie die verbindenden Segmente, durch die Leitungen verlegt werden können.

Tabelle 3.6 ordnet einige Elemente aus dem Metamodell des E/E-Konzept-Tools den in Tabelle 3.1 - Tabelle 3.3 definierten Inhalten der Systementwicklung zu. Die Elemente für die Inhalte der logischen und technischen Systemarchitektur stammen überwiegend aus der *Funktions-* und *Komponentennetzwerk Architektur.*

Die logische Systemarchitektur kann im E/E-Konzept-Tool durch das *Funktionsnetzwerk* aus der *Funktions-Architektur* abgebildet werden. In Abbildung 3.8 ist ein kleiner Ausschnitt eines möglichen *Funktionsnetzwerks* für das KFS aus Kapitel 2 dargestellt, der inhaltlich aus Abbildung 2.1 abgeleitet ist. Ein *Logischer Sensor* mit Namen `Tempomathebel` verfügt über einen Sender-Port mit dem Namen `Benutzerkommando`. Der Port ist durch das Interface `IF_Benutzerkommando` getypt, das die Beschreibung eines abstrakten Signals für die Übermittlung von Benutzerkommandos enthält (die Interface-Definition ist nicht abgebildet). Der *Function Block* mit dem Namen `Tempomat/Speedtronic` verfügt über einen entsprechenden Empfänger-Port, der über einen *Assembly Connector* mit dem Sender-Port verbunden ist. Auf diese Weise wird der abstrakte Signalfluss zwischen den beiden logischen Einheiten `Tempomathebel` und `Tempomat/Speedtronic` ausgedrückt.

Für die Darstellung der Inhalte aus der technischen Systemarchitektur bietet sich vor allem die *Komponentennetzwerk-Architektur* an. Abbildung 3.9 zeigt einen Ausschnitt eines möglichen Modells für das KFS, die aus Abbildung 2.2 abgleitet ist.

Inhalt nach Kernprozess (siehe Tabelle 3.1 - Tabelle 3.3)			**Mögliche Elemente aus dem Metamodell des E/E-Konzepttools**
Logische Systemarchitektur	1	Funktionsnetzwerk	*FunctionBlock, ActuatorBlock, SensorBlock, Composition*
	2	Abstrakte Datenelemente	*DataElement, NaturalType*
	3	Schnittstellen	*SenderReceiverPort, SenderReceiverInterface*
	4	Zeitangaben	*PortCommunicationRequirement*
Technische Systemarchitektur	1	E/E-Komponenten	*EEKomponente, ECU, Aktuator, Sensor*
	2	Interner Steuergeräteaufbau	*CPU, ASIC, FPGA, RAM, ROM, EEPROM*
	3	Kommunikationsverbindungen	*BusSystem, KonventionelleVerbindung, BusAnbindung, KonventionelleAnbindung*
	4	Buskommunikation	*Signal, SignalGroup, BusTransmission, Frame*
	5	Leistungsversorgung	*Batterie, Generator, LVVerbindung, LVAnbindung*
	6	Leitungen	*Leitung, Ader, Pin, Stuetzpunkt, Trennstelle, Pin*
	7	Fahrzeugtopologie	*Bauraum, EinbauOrt, TopologieSegment, Ausbindung, Trennstelle*
Mappings	1	Funktion ⇔ Mikroprozessor	*FunctionBlockMapping*
	2	Logischer Sensor/Aktuator ⇔ E/E-Komponente	*SensorBlockMapping, ActuatorBlockMapping*
	3	Abstraktes Datenelement ⇔ Bussignal bzw. Botschaft	*InformationUnitSignalMapping*

Tabelle 3.6: Mögliche Elemente aus dem Metamodell des E/E-Konzept-Tools für die logische und technische Systemarchitektur sowie für die *Mappings*

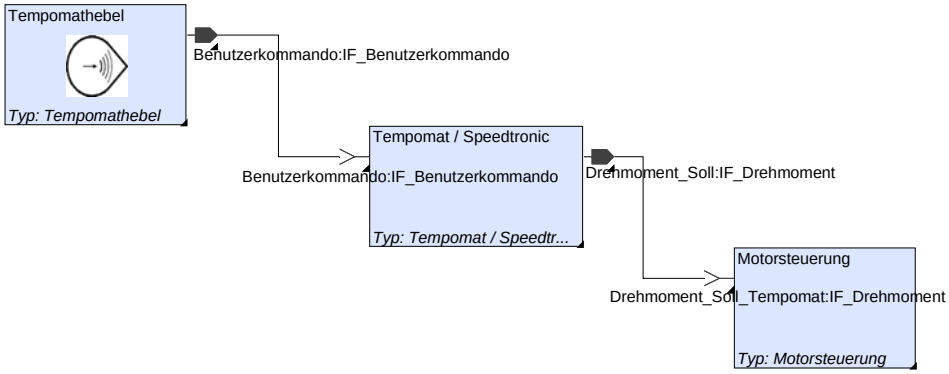


Abbildung 3.8: Ausschnitt eines möglichen *Funktionsnetzwerks* für das KFS

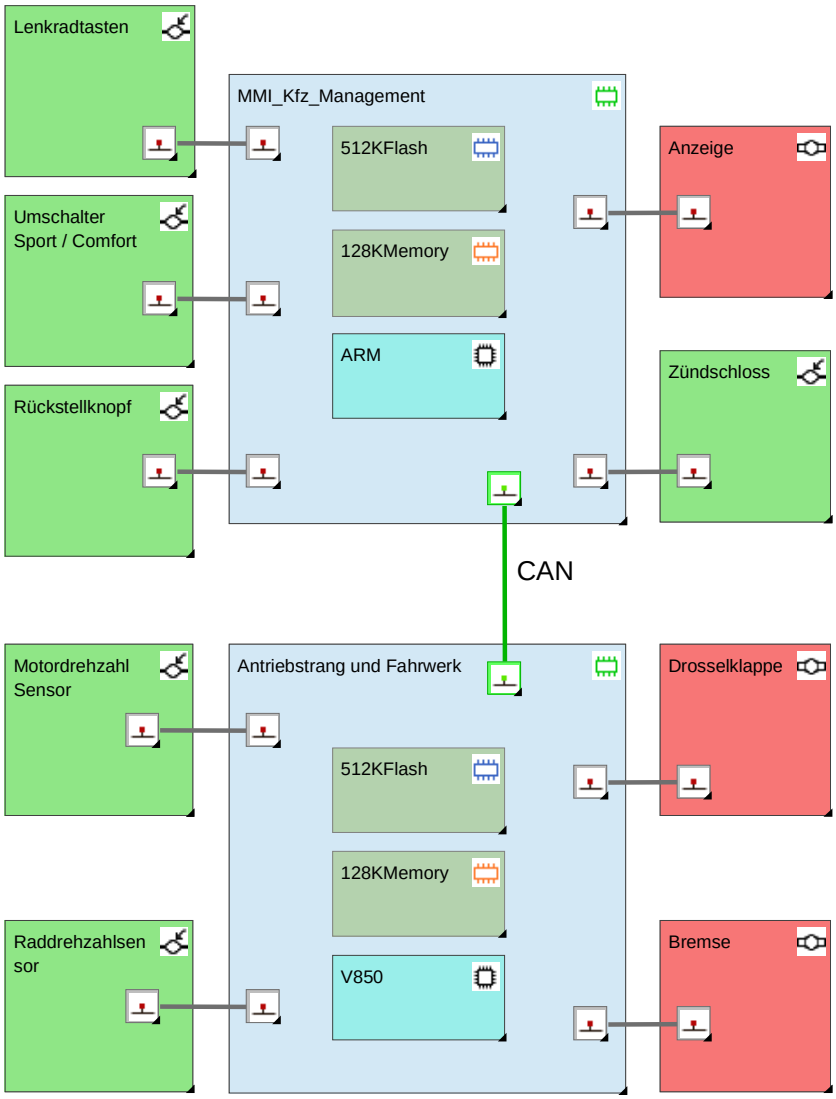


Abbildung 3.9: Ausschnitt einer möglichen *Komponentennetzwerk-Architektur* für das KFS

Vergleich

In Tabelle 3.7 werden die beiden betrachteten Modellierungsansätze in Bezug auf die Inhalte des Kernprozesses (siehe Tabelle 3.1 - Tabelle 3.3) verglichen. Die Modellierung der logischen Systemarchitektur ist demnach mit beiden Ansätzen möglich. Allerdings gibt es bei dem Funktionsnetzwerk des E/E-Konzept-Tools die Einschränkung, dass dieses nicht explizit zwischen einem rein logischen Funktionsnetzwerk und einem Netzwerk aus Softwarekomponenten differenziert. Beide Ebenen werden durch dasselbe Modell adressiert. Da aber von zahlreichen Realisierungsaspekten abstrahiert werden kann, lässt sich das Funktionsnetzwerk für die Modellierung der logischen Systemarchitektur weitgehend nutzen.

Inhalt nach Kernprozess (siehe Tabelle 3.1 - Tabelle 3.3)			**Abdeckung durch Modellierungsansatz** EAST-ADL	E/E-Konzept-Tool
Logische Systemarchitektur	1	Funktionsnetzwerk	●	●
	2	Abstrakte Datenelemente	●	●
	3	Schnittstellen	●	●
	4	Zeitangaben	●	●
Technische Systemarchitektur	1	E/E-Komponenten	●	●
	2	Interner Steuergeräteaufbau	●	●
	3	Kommunikationsverbindungen	●	●
	4	Buskommunikation	●	●
	5	Leistungsversorgung	○	●
	6	Leitungen	◉	●
	7	Fahrzeugtopologie	○	●
Mappings	1	Funktion ⇔ Mikroprozessor	●	●
	2	Logischer Sensor/Aktuator ⇔ E/E-Komponente	●	●
	3	Abstraktes Datenelement ⇔ Bussignal bzw. Botschaft	●	●

● Wird im vollen Umfang unterstützt.
◉ Wird eingeschränkt unterstützt.
○ Wird nicht unterstützt.

Tabelle 3.7: Vergleich der unterschiedlichen Modellierungsansätze in Bezug auf die Inhalte des Kernprozesses

Bei der technischen Systemarchitektur zeigt sich, dass die Definition von E/E-Komponenten und die Beschreibung ihrer Kommunikationsverbindungen (Busse, konventionelle Verbindungen) von allen Ansätzen unterstützt werden. Unterschiede gibt es bei der Beschreibung des internen Steuergeräteaufbaus. Die Themen Leistungsversorgung, Leitungen und Fahrzeugtopologie werden nur vom E/E-Konzept-Tool ausführlich adressiert. Die *Mappings* lassen sich bei allen

Ansätzen pflegen, teilweise ergeben sie sich jedoch erst indirekt durch eine Kette von verlinkten Objekten.

Die Mängel der EAST-ADL sind in Themenbereichen angesiedelt, deren Relevanz für die weitere Diskussion bereits ausgeschlossen wurde. Da das Metamodell des E/E-Konzept-Tools nur eingeschränkt öffentlich verfügbar ist, wird im Folgenden die EAST-ADL als Referenz-Modellierungssprache für die Systementwicklung verwendet.

3.1.3.c Modellbildung in der Softwareentwicklung

Im Bereich der Softwareentwicklung ist die Standardisierung der Modellbildung im Vergleich zur Systementwicklung relativ weit fortgeschritten. Mit der *Unified Modeling Language* (UML) gibt es eine standardisierte und weithin akzeptierte Modellierungssprache, mit der sich Modelle für ein breites Spektrum von Softwareentwicklungsaufgaben erstellen lassen. Hier gibt es auch einen relativ gut entwickelten Markt von Modellierungswerkzeugen und der Austausch von Modellen ist prinzipiell möglich. Die UML wird ausführlicher in Kapitel 3.2.1 behandelt.

3.1.4 Zwischenergebnis

Das zurückliegende Kapitel adressiert aus der Sicht der Systementwicklung die in der Problemstellung (Kapitel 1.2) formulierten Fragen nach den grundlegenden Anforderungen und Trends der Domäne und die Integration von System- und Softwareentwicklung durch einen Gesamtprozess sowie den Austausch von Modellen.

Kapitel 3.1.1 hat gezeigt, dass viele Softwarefunktionen im Fahrzeug dadurch charakterisiert werden, dass sie in ein elektronisches System mit Sensoren und Aktuatoren eingebettet sind, in dem sie Steuerungs- und Regelungsaufgaben wahrnehmen. Einfluss auf das Design einer solchen Softwarefunktion hat neben der logischen Systemarchitektur, die die funktionalen Steuerungs- und Regelungsaufgaben definiert, insbesondere auch die technische Systemarchitektur. Sie beschreibt die relevanten Hardwarekomponenten, mit denen eine Funktion realisiert wird und legt z. B. fest, ob ein für die Funktion relevanter Wert über einen Bus empfangen werden muss oder lokal über einen Sensor erfasst wird, der direkt am Steuergerät der Funktion angeschlossen ist. Darüber hinaus können in dieser Domäne Anforderungen bzgl. Zuverlässigkeit, Sicherheit, Ressourcenverbrauch, Wiederverwendung und Portabilität sowie Variabilität von besonderer Bedeutung sein. Die Gewichtung dieser Anforderungen kann jedoch von Funktion zu Funktion bzw. von Subsystem zu Subsystem variieren.

Mit dem in Kapitel 3.1.2 vorgestellten Kernprozess steht eine grobe Beschreibung von Prozessschritten und Artefakten der Systementwicklung zur Verfügung. Es wurden Listen mit möglichen Inhalten der logischen und technischen Systemarchitektur sowie der *Mappings* zusammengestellt, die im weiteren Verlauf der Diskussion als Referenz für Artefakte dienen, die von der Systementwicklung an die Softwareentwicklung übergeben werden (siehe Tabelle 3.1 - Tabelle 3.3). In Kapitel 3.1.3 wurden modellbasierte Ansätze vorgestellt, die diese Inhalte in formalisierten Modellen erfassen und abschließend in Tabelle 3.7 in Bezug auf ihre Inhalte anhand der erwähnten Listen verglichen.

Ein zentrales Modell bzgl. der funktionalen Anforderungen, das von der System- an die Softwareentwicklung übergeben wird, ist die logische Systemarchitektur (in dieser Arbeit auch Funktionsnetzwerk genannt). Sie beschreibt die hierarchische Struktur der Funktionen, die vorhandenen logischen Sensoren und Aktuatoren und die Kommunikationsbeziehungen in Form von abstrakten Signalen. Alle untersuchten Modellierungssprachen für Systemarchitekturen aus Kapitel 3.1.3 enthalten ein solches Modell der logischen Systemarchitektur.

Die Brücke von der logischen zur technischen Systemarchitektur wird von den untersuchten Modellierungssprachen durch die sog. *Mappings* geschlagen. Diese verbinden z. B. die Funktionen mit den Steuergeräten, auf denen sie implementiert sind oder die abstrakten Signale mit den konkreten Signalen eines realisierenden Kommunikationssystems. Die über diese Brücke verlinkten Modelle der technischen Systemarchitektur enthalten einige Informationen, die für die Softwareentwicklung ebenfalls relevant sind. Schwerpunkte sind der Aufbau von Hardwarekomponenten sowie deren Vernetzung. Die in Kapitel 3.1.3 untersuchten Modellierungssprachen behandeln diese Ebenen, grenzen die einzelnen Modelle jedoch unterschiedlich voneinander ab.

Auf Basis dieser Erkenntnisse können auch erste Feststellungen bzgl. der in der Problemstellung (Kapitel 1.2) gestellten Frage nach domänenspezifischen Anforderungen an einen möglichen Modelltransformationsmechanismus zwischen Modellen der System- und Softwareentwicklung gemacht werden. So müssen als Eingabemodelle die besprochenen Modelle der logischen und technischen Systemarchitektur verarbeitet werden können. Welche Informationen im Detail für die Softwareentwicklung relevant und somit enthalten sein müssen, wird noch im weiteren Verlauf der Untersuchungen geklärt. Aus den gesammelten grundlegenden Anforderungen und Trends der Domäne lässt sich weiterhin ablesen, dass der Modelltransformationsmechanismus das Ausgabemodell in Hinblick auf die variierende Gewichtung der oben genannten Anforderungen optimieren muss. Darüber hinaus müssen die erzeugten Ausgabemodelle mit den Eingabemodellen verknüpft werden, um so der allgemeinen Forderung in der Domäne nach Verfolgbarkeitsmechanismen gerecht zu werden.

Zusammenfassend kann festgehalten werden, dass seitens der Systementwicklung wesentliche Voraussetzungen erfüllt sind, um die Softwareentwicklung durch den Austausch von Modellen stärker mit der Systementwicklung zu integrieren. Durch die gestiegene Verbreitung modellbasierter Methoden in der Systementwicklung kann zu Beginn der Softwareentwicklung in dieser Domäne auf vergleichsweise stark formalisierte Informationen zurückgegriffen werden. Bei der Untersuchung aktueller Softwareentwicklungsmethoden in dem folgenden Kapitel soll über die Fragen aus der Problemstellung hinaus überprüft werden, inwiefern diesen methodischen Veränderungen im Bereich der Systementwicklung, auf der Seite der Softwareentwicklung Rechnung getragen wird. Ein Schwerpunkt bildet dabei die Frage, wie sich die Modelle der Systementwicklung im Rahmen der Softwareentwicklung wiederverwenden lassen.

3.2 Softwareentwicklung

Dieses Kapitel erläutert den Stand der Technik zu den für die Problemstellung relevanten Themen aus dem Bereich Softwareentwicklung. Die bereits in Kapitel 3.1.3 begonnene Schwerpunktbildung auf modellbasierter Entwicklung wird an dieser Stelle fortgeführt.

In dem folgenden Unterkapitel wird mit der UML zunächst eine etablierte Modellierungssprache für die modellbasierte Softwareentwicklung vorgestellt. Die in dem darauf folgenden Unterkapitel zusammengefasste *ROPES*-Methode präzisiert deren Einsatz für die Entwicklung von eingebetteten Echtzeitsystemen und beschreibt darüber hinaus auch eine konkrete Vorgehensweise. In Kapitel 3.2.4 wird mit den *Mustern* ein verbreitetes Konzept für eine systematisierte Erstellung von Modellen vorgestellt.

3.2.1 Die Unified Modeling Language

Die *Unified Modeling Language* (UML) ist eine Sprache für die Modellierung von softwareintensiven Systemen im Sinne der in Kapitel 3.1.3 referenzierten Definition. Die verfügbaren Modellelemente können sowohl statische als auch dynamische Aspekte eines Systems beschreiben und wurden mit dem Ziel konzipiert, unabhängig von dem Fach- und Realisierungsgebiet zu sein.

Die UML selbst ist keine Methode, unterstützt an vielen Stellen aber in besonderem Maße eine objektorientierte Vorgehensweise, wie z. B. die in Kapitel 3.2.2 zusammengefasste ROPES-Methode. Die bereitgestellten Modellelemente und Diagrammarten decken viele Phasen des Softwarelebenszyklus ab, wie z. B. Anforderungserfassung, Analyse, Design und Dokumentation.

Entwickelt und spezifiziert wird die UML unter dem Dach der *Object Management Group* (OMG), einer Organisation, in der namhafte Firmen aus dem Softwareumfeld vertreten sind. Die UML-Spezifikation der OMG hat somit den Charakter eines Industriestandards. Aktuell ist die Version 2.0 (kurz „UML2“) [OMG '05c], die ältere Version 1.4 ist ein ISO/IEC-Standard [ISO/IEC '05].

Die UML-Spezifikation setzt sich aus einem formalisierten Metamodell, einer textuellen Beschreibung der Semantik für jedes dort spezifizierte Modellelement und der Festlegung der graphischen Darstellung zusammen. Diese Beschreibung der Semantik enthält bei manchen Modellelementen sog. *Semantic Variation Points*. Dabei handelt es sich um Bereiche der Spezifikation, die bewusst nicht vollständig spezifiziert wurden, um domänenspezifischen Erweiterungen mehr Freiheiten zu lassen. Die Semantik dieser Elemente muss also projektspezifisch festgelegt werden. Die UML besitzt schon aufgrund dieser Tatsache keine formale Semantik und verhindert per se auch nicht widersprüchliche Modelle [Jeckle et al. '03].

Das folgende Unterkapitel gibt einen Überblick über die vorhandenen Diagramme. Aufgrund des hohen Bekanntheitsgrades der UML wird auf eine ausführliche Darstellung der Modellierungssprache verzichtet und an dieser Stelle auf den Standard [OMG '05c] und die umfangreich

vorhandene Sekundärliteratur verwiesen (z. B. [Jeckle et al. '03] und [Born et al. '05]). Stattdessen konzentrieren sich die beiden weiteren Unterkapitel auf Aspekte, die im Rahmen dieser Arbeit von besonderer Bedeutung sind. Dazu gehören das UML-Metamodell und der Profilmechanismus sowie der Einsatz von UML-Modellen für die Darstellung von Ergebnissen der Systementwicklung.

3.2.1.a Diagramme

Die Übersicht in Abbildung 3.10 zeigt alle in der UML2 vorhandenen Diagrammarten und charakterisiert deren Inhalt mit Stichworten.

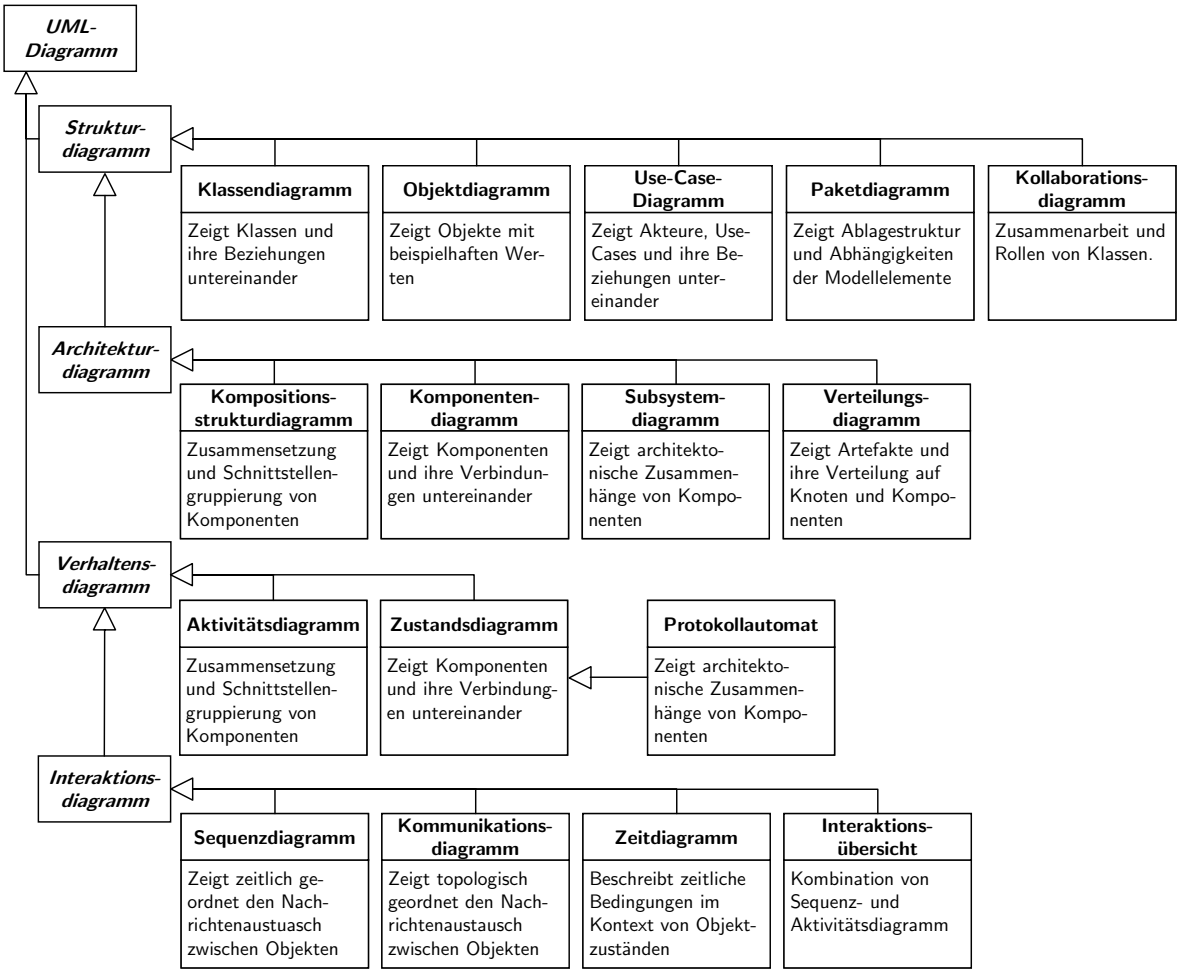


Abbildung 3.10: Die UML-Diagramme im Überblick (angepasst nach [Oestereich '04])

Obwohl die UML keine Methode beinhaltet, die den Einsatz dieser Diagramme in einem einheitlichen Prozess beschreibt, lassen sich den einzelnen Diagrammformen doch bestimmte Entwicklungsaktivitäten zuordnen. Abbildung 3.11 zeigt die Einordnung einiger Diagramme in die Entwicklungsaktivitäten „Analyse“, „Design“ und „Implementierung“ sowie für welche der Sichten „Struktur“, „Interaktion“ und „Verhalten“ sie sich besonders eignen. Konzeptuelle Über-

lappungen verschiedener Diagrammarten erlauben die alternative Darstellung bestimmter Sachverhalte mit Hilfe verschiedener Diagramme [IESE a].

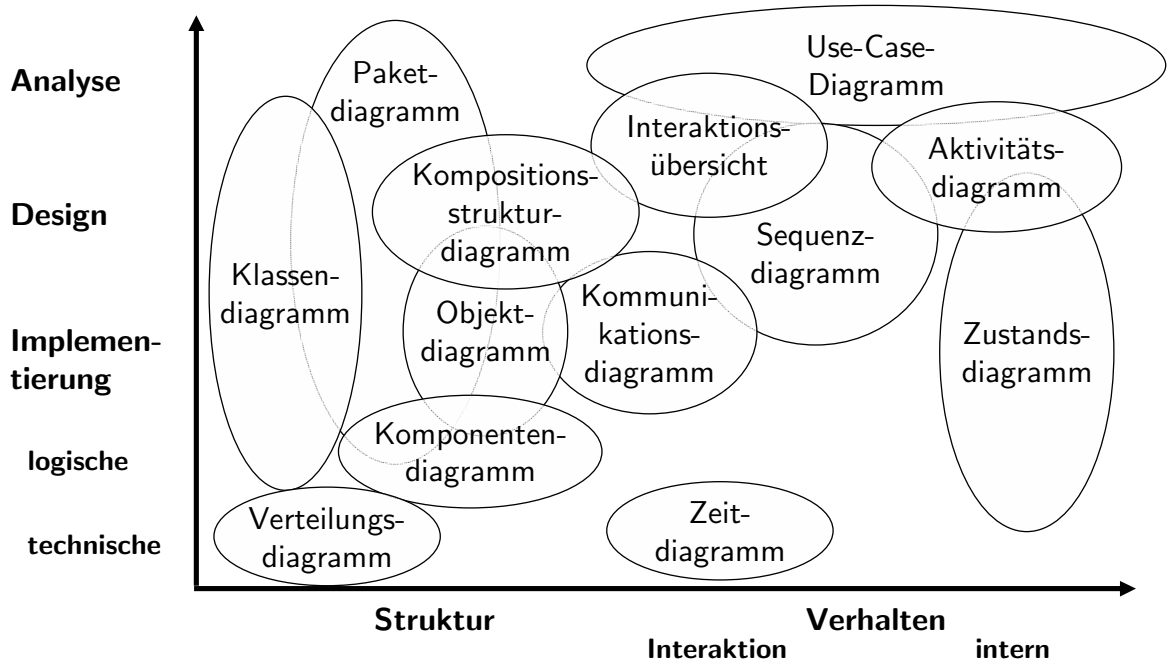


Abbildung 3.11: Methodische Einordnung der UML-Diagramme (angepasst nach [IESE a])

Welche Diagramme in einem Entwicklungsprojekt zum Einsatz kommen, hängt stark von der verwendeten Methode und weiteren Randbedingungen des Projekts ab. In der Praxis hat sich gezeigt, dass besonders die Klassendiagramme zur Strukturdarstellung, Use-Case-Diagramme zur Darstellung von Anwendungsfällen, Zustandsdiagramme zur Verhaltensbeschreibung, und Sequenzdiagramme zur Darstellung von Interaktion zum Einsatz kommen [IESE a]. Im Rahmen dieser Arbeit kommen ausschließlich Klassendiagramme (entsprechend der in Kapitel 1.3 beschriebenen Fokussierung auf Strukturmodelle) zum Einsatz, die in Kapitel 3.2.1.c näher erläutert werden.

3.2.1.b UML-Metamodell und -Profile

Die Definition der UML und die damit zusammenhängenden weiteren Standards der OMG richten sich nach der in Kapitel 3.1.3.a vorgestellten Vier-Schichten-Metamodellarchitektur. In den Versionen 1.x der UML wurde für Metamodellierung die *Meta-Object Facility* (MOF) [OMG '02a] verwendet, die selbst wiederum auf Grundlage eines Kerns von UML beschrieben war. Im Normierungsprozess für die Versionen 2.0 der beiden Standards wurde die Zielsetzung verfolgt, MOF und den notwendigen Kern der UML zu vereinigen und als einheitliche Menge von Basiskonzepten darzustellen. Diese Basiskonzepte sind mittlerweile verabschiedet und in der sog. *Infrastructure Library* zusammengefasst, die den ersten Teil der Definition von UML darstellt und eine Sammlung grundlegender Metametaklassen enthält. In dieser Bibliothek ist auch bereits ein Metamodellpaket für Erweiterungsmechanismen für Metamodelle durch Profile

enthalten. Die *UML 2 Infrastructure* [OMG '05b] liefert die Grundlage für den zweiten Teil der Spezifikation, die *UML 2 Superstructure* [OMG '05c], in der die eigentlichen Modellelemente, Notationen und Semantik der UML unter Benutzung der Konzepte aus der *Infrastructure* beschrieben sind. Für die statische Semantik wird auf die textuelle Spezifikationssprache *Object Constraint Language* (OCL) [OMG '06] zurückgegriffen, die dynamische Semantik wird für jedes Modellelement in einem eigenen Abschnitt in englischer Sprache beschrieben.

Die verschiedenen Diagramme der UML erlauben zwar bereits eine relativ umfangreiche Beschreibung eines Softwaresystems, können allerdings durch verschiedene Technologien noch erweitert werden. Ein in der UML integrierter Mechanismus für Erweiterungen mittels Spezialisierungen ist durch die Möglichkeit der Definition von *UML-Profilen* gegeben, die als Zusammenfassung von Adaptionen von Konstrukten mit einer bestimmten Zielstellung beschrieben werden können. In *UML-Profilen* können für jeden Typ von Elementen in einem Modell spezialisierte Versionen definiert werden, die durch *Stereotypen* gekennzeichnet sind. Die Kennzeichnung ist entweder mit Text (in Guillemets gefasste Namen, also etwa `<<stereotyp>>`) oder mit vom Benutzer vorzugebenden Icons möglich. In der Definition eines solchen Stereotyps können spezifische Name-Wert-Paare (sog. *Tagged Values* in der Form `{Eigenschaft = Wert}` als Attribute des Stereotyps) und zusätzliche Bedingungen formuliert werden, um ein spezielles Konzept auszudrücken. Ein Profil bündelt dann eine Sammlung von Stereotypen zu einem Paket. Bei der Erstellung von Profilen ist zu beachten, dass grundsätzlich nur die Spezialisierung existierender Elementtypen gestattet ist und die beschriebenen Konzepte nicht in Widerspruch mit den sonstigen Konzepten der UML stehen dürfen. Für gängige Einsatzgebiete gibt es auch bereits vordefinierte bzw. standardisierte Profile [OMG c].

Anknüpfungspunkte zum UML-Metamodell und der UML-Metamodellierungssprache gibt es im Rahmen dieser Arbeit bei der Untersuchung von Modelltransformationssprachen (Kapitel 3.3), der Untersuchung der UML-Templates (Kapitel 3.2.4.e) sowie der Spezifikation der eigenen Transformationssprache in Kapitel 4. Der UML-Profilmechanismus wird im Rahmen dieser Arbeit genutzt, um neu definierte Metaelemente, die die Semantik der UML erweitern, in herkömmlichen UML-Diagrammen darstellen zu können (siehe Kapitel 4.4). Es gibt zudem modellbasierte Ansätze aus dem Bereich der Systementwicklung, die Inhalte der Systementwicklung mit Hilfe eines Profils aus Basis der UML modellieren (siehe Kapitel 3.1.3.b).

3.2.1.c Modellierung der logischen Systemarchitektur mit Klassendiagrammen

Ein Klassendiagramm ist eine graphische Darstellung von Klassen (Objekttypen) und ihren Beziehungen zu anderen Klassen. Ein durch solch ein Diagramm visualisiertes Klassenmodell wird in objektorientierten Softwareentwicklungsmethoden üblicherweise sowohl in der Analyse- als auch in der Designphase verwendet (siehe Kapitel 3.2.2). Im Folgenden soll die in der Problemstellung aufgeworfene Frage diskutiert werden, wie sich die Modelle der Systementwicklung in der UML darstellen lassen. Dazu gehört insbesondere auch die Modellierung der logischen Systemarchitektur in der Analysephase der Softwareentwicklung. Die hier zu modellierenden Inhalte sind in Tabelle 3.1 zusammengefasst.

Das im Rahmen dieser Arbeit betrachtete Funktionsnetzwerk besteht im Wesentlichen aus Blöcken, die über gerichtete Kanten verbunden sind und über diese abstrakte Datenelemente austauschen. Abbildung 3.12 zeigt die vereinfachte Darstellung, die in Anlehnung an [Schäuffele et al. '03] z. B. in Abbildung 2.1 verwendet wurde, am Beispiel eines Signals `SensorValue`, das von Block `Sensor` zu Block `ControlFunction` versendet wird.

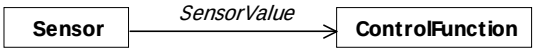


Abbildung 3.12: Beispiel eines Blockdiagramms

Bei der Konzeption der Modellierungssprachen für die Systementwicklung, die die Modellierung eines solchen Funktionsnetzwerks unterstützten (siehe Kapitel 3.1.3.b), wurde teilweise auf eine möglichst weitgehende UML-Konformität geachtet. Bei der EAST-ADL bezieht sich diese auf die UML2. Das Beispiel in Abbildung 3.13 zeigt für einige Elemente der *Functional Analysis Architecture,* wie diese mit Hilfe eines *UML-Profils* dargestellt werden könnte. Die Blöcke werden als stereotypisierte Klassen dargestellt, die über Interfaces verfügen. Im Gegensatz zur offiziellen Notation der EAST-ADL, haben die Interfaces hier nicht die Dreieck-Symbolik, da diese nicht zur Standardnotation der UML gehört. Die Zeitangaben (im Beispiel `period=10`) lassen sich als *tagged value* realisieren, deren Darstellung jedoch werkzeugabhängig sein kann.

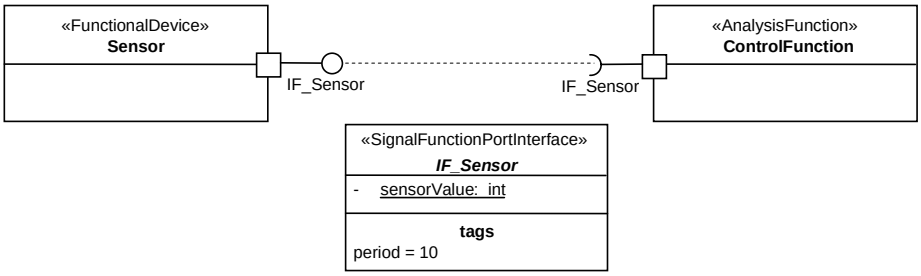


Abbildung 3.13: Beispiel eines möglichen UML2-Profils für die Modellierung der *Functional Analysis Architecture* der EAST-ADL

Sofern bei der Softwareentwicklung die UML2 zum Einsatz kommt, lässt sich somit im Falle der EAST-ADL das Funktionsnetzwerk direkt für die Analyse übernehmen. Im Rahmen der Arbeit wurde jedoch auch nach einer sehr einfachen Darstellung gesucht, die sich möglichst auch mit UML 1.4-Werkzeugen modellieren lässt. Eine in Abbildung 3.14 gezeigte Möglichkeit das Beispiel mit diesen Mitteln zu modellieren, ist die Verwendung von zwei Klassen `Sensor` und `ControlFunction`, die über eine Assoziation verbunden sind und zur Laufzeit ein Signal austauschen können. Mit „Signal" ist dabei der im UML-Metamodell definierte Typ gemeint, der als Klasse mit dem Stereotyp `<<signal>>` deklariert wird. Die Empfangbarkeit des Signals wird bei `ControlFunction` in einem zusätzlichen *Compartment* durch den Eintrag des Signals modelliert. Durch die private Methode `send_sensorValue()` wird angezeigt, welche Klasse ein Signal sendet (für größere Modelle wichtig). In Anlehnung an [Douglass '99] können mit dem Stereotyp `<<periodic>>` und dem *tagged value* `period` Angaben zum Zeitverhalten beim Senden des

Signals gemacht werden. Der Wert von `period` gibt die Zykluszeit in Millisekunden an. Ist kein Stereotyp angegeben, wird das Signal episodisch versendet. Über die gerichtete Assoziation wird entsprechend der UML-Semantik modelliert, dass `Sensor` zu `ControlFunction` zur Laufzeit eine Verbindung hat und somit zu jeder Zeit Nachrichten an `ControlFunction` senden kann.

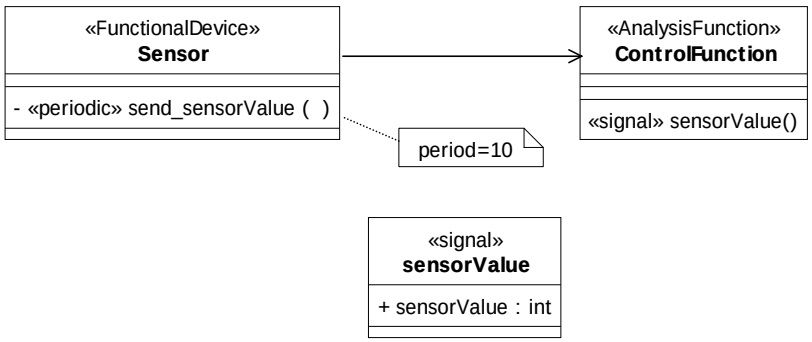


Abbildung 3.14: Funktionsnetzwerk als Klassendiagramm

Um die Lesbarkeit zu erhöhen, wird im Rahmen dieser Arbeit für diese Modellierung eine alternative Darstellung vereinbart, die in Abbildung 3.15 gezeigt ist. Hierbei wird der Signalfluss mit Hilfe der in der UML2 neu eingeführten *Information Flows* (siehe [OMG '05c]) modelliert. *Information Flows* werden durch Abhängigkeitsbeziehungen mit dem Stereotyp `<<flow>>` modelliert.

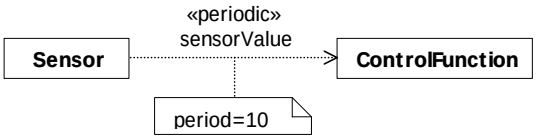


Abbildung 3.15: Modellierung von Signalen mit Hilfe von *Information Flows*

Die Semantik dieser Abhängigkeitsbeziehung ist im UML-Standard wie folgt definiert: „eine oder mehrere Informationseinheiten fließen von der Quelle zum Ziel“. Als Name des *Information Flows* wird in dieser Arbeit der Name des entsprechenden Signals verwendet. Da bei den hier modellierten Funktionsnetzen aber jede Abhängigkeitsbeziehung diese Semantik hat, wird der Stereotyp in den Diagrammen nicht angezeigt. Wenn im Folgenden Diagramme mit *Information Flows* wie in Abbildung 3.15 gezeigt werden, wird implizit angenommen, dass entsprechend der beschriebenen Konvention die Signalbeziehung auch nach dem Muster aus Abbildung 3.14 modelliert wurde.

3.2.1.d Modellierung der technischen Systemarchitektur mit Verteilungsdiagrammen

Für die Modellierung der Hardware-Topologie und der Verteilung der Software des Systems auf die Hardware ist in der UML das Verteilungsdiagramm (engl. *Deployment Diagram*) vorgesehen. Hardware-Einheiten werden als *Knoten* (engl. *Nodes*) modelliert, die über Kommunikationsverbindungen und Abhängigkeiten miteinander verknüpft werden können. Eine auf einen Knoten verteilbare Einheit wird als *Artefakt* (engl. *Artifact*) bezeichnet und kann für

den Code einer Softwarekomponente stehen oder eine andere Form von verteilbarer Information (z. B. Dateien).

Abbildung 3.16 zeigt eine Modellierung der technischen Systemarchitektur des KFS mit einem Verteilungsdiagramm, das inhaltlich mit Abbildung 3.6 und Abbildung 3.9 vergleichbar ist. Um eine ähnliche Aussagekraft wie die in Kapitel 3.1.3.b erläuterten Modelle der Systementwicklung zu erreichen, wurde die UML in diesem Beispiel um ein „ad hoc-Profil“ mit Stereotypen für die Beschreibung von Steuergeräten, Sensoren/Aktuatoren und Kommunikationsverbindungen erweitert.

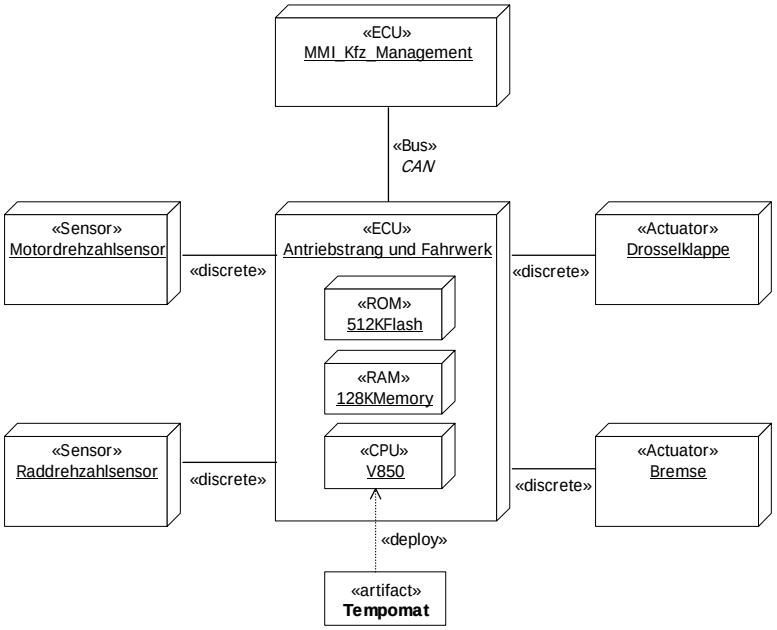


Abbildung 3.16: Ausschnitt eines möglichen Verteilungsdiagramms für das KFS

Bei der Modellierung weiterer Aspekte der technischen Systemarchitektur, wie z. B. Buskommunikation (insbesondere Nachrichten), Leistungsversorgung, Leitungen oder die Fahrzeugtopologie, stößt die Stereotypisierung jedoch an ihre Grenzen, da eine in der Domäne übliche Darstellungsweise auf diesem Weg kaum zu erreichen ist.

Die reine UML eignet sich aufgrund der wenigen Modellelemente nur sehr begrenzt zur Spezifikation von Hardware [Jeckle et al. '03]. Sie stellt zur Beschreibung der Hardware nur die Mittel zur Verfügung, um die Verteilung von Software spezifizieren zu können. Ein standardisiertes und etabliertes *UML-Profil* würde die Situation verbessern, ist aber zurzeit nicht verfügbar. Wie bereits in Kapitel 3.1.3.b erwähnt, wird aktuell im Rahmen des Projekts ATESST [ATESST] an einer neuen Version der EAST-ADL gearbeitet, die als UML2-Profil veröffentlicht werden soll. Sollte es den Werkzeugherstellern nach der Verabschiedung gelingen, eine in der Domäne akzeptierte Darstellung zu finden, ist in Zukunft die Nutzung der UML für die Modellierung der technischen Systemarchitektur denkbar.

3.2.2 Die ROPES-Methode

Rapid Object-Oriented Process for Embedded Systems (ROPES) [Douglass '99] [Douglass '04] ist eine objektorientierte Entwicklungsmethode, die speziell für eingebettete Systeme mit Echtzeitanforderungen konzipiert ist. Sie soll sich auch für Projekte mit hohem Systementwicklungsanteil eignen, bei denen neben der Softwareentwicklung weitere Ingenieurdisziplinen in einen Gesamtprozess integriert werden müssen. Als Beispiele werden von dem Autor der Flugzeug- und Automobilbau genannt.

3.2.2.a Prozessmodell

Als Methode umfasst ROPES die Festlegung einer Modellierungssprache und die Beschreibung eines Entwicklungsprozesses, der die Verwendung der in der Modellierungssprache enthaltenen Modellelemente beschreibt. Als Modellierungssprache kommt die UML zum Einsatz (siehe Kapitel 3.2.1), bei dem Entwicklungsprozess handelt es sich im Kern um ein Spiralmodell.

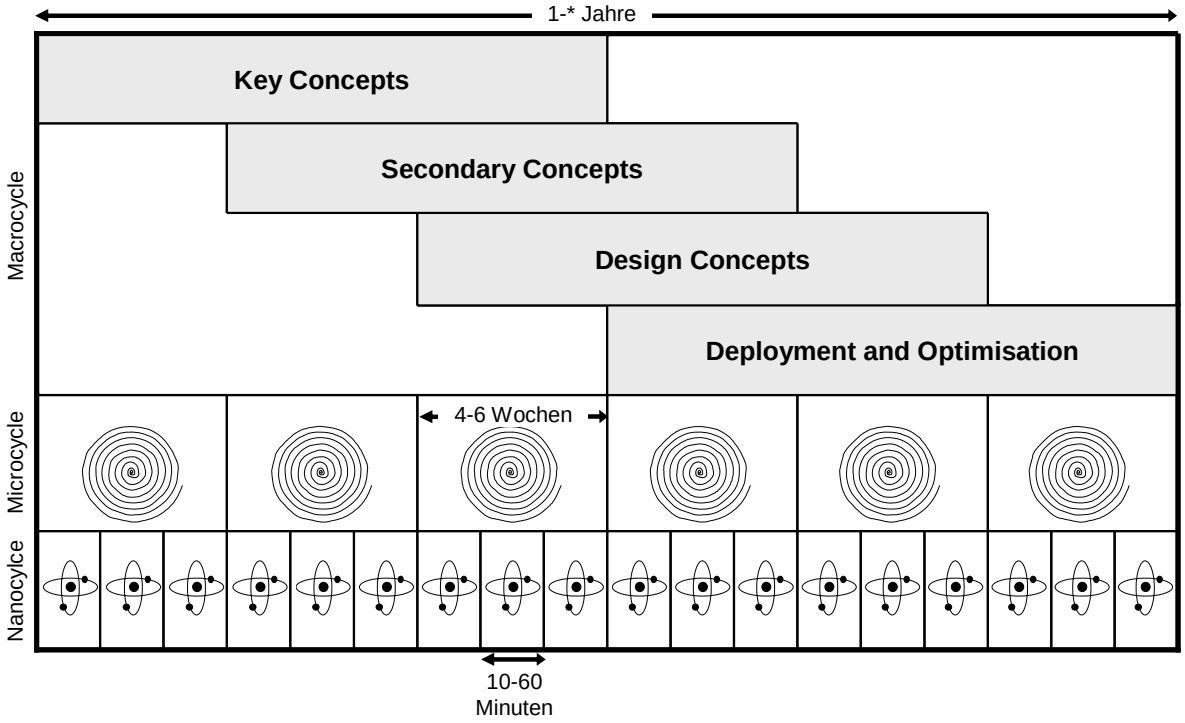


Abbildung 3.17: Entwicklungszyklen nach ROPES [Douglass '04]

Abbildung 3.17 zeigt die drei Zeitachsen des ROPES-Standardprozesses. Die Makroebene läuft über die gesamte Projektlaufzeit (bis mehrere Jahre) und unterteilt sich in die überlappenden Phasen *key concepts* (Schlüsselkonzepte), *refinement of concepts* (Konzeptverfeinerung), *design and implementation* (Design und Implementierung) und *optimisation & deployment* (Optimierung und Verteilung). In jeder Phase der Makroebene werden 3-4 Mikrozyklen nach dem Spiralmodell durchlaufen, die immer mit der Bereitstellung eines Prototyps beendet werden. Jeder neue Prototyp erweitert den letzten Prototyp, wobei sich der Erweiterungsschwerpunkt nach der aktuellen Phase der Makroebene richtet. Ein Nanozyklus umfasst im

ROPES-Prozess eine abgeschlossene Abfolge aus Design/Codierung, Übersetzung und Debugging.

Für Projekte, bei denen der Softwareentwicklung eine länger andauernde Entwicklung von Hardwarekomponenten vorausgeht, sieht ROPES ein alternatives Lebenszyklusmodell vor. Abbildung 3.18 zeigt die Variante *SemiSpiral*, bei der der Mikrozyklus in Entwicklungsphasen eingebettet ist, die das Gesamtsystem betreffen und nach dem Wasserfallprinzip durchlaufen werden. Da diese Variante konform mit dem in Kapitel 3.1.2 vorgestellten Kernprozess der Systementwicklung ist (eine genauere Einordnung erfolgt in Kapitel 3.2.2.g), wird nur diese im weiteren Verlauf der Arbeit betrachtet.

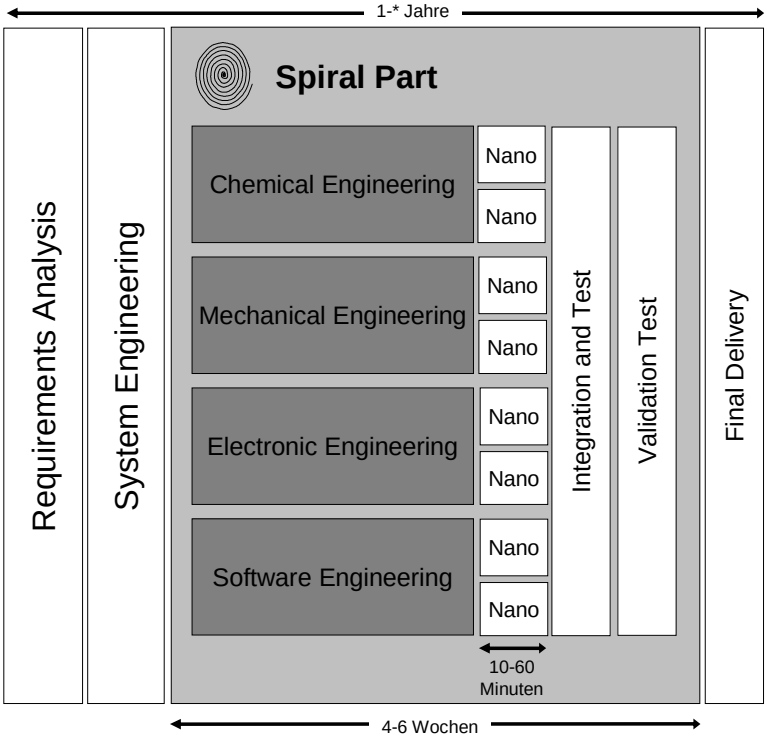


Abbildung 3.18: Entwicklungszyklen nach ROPES (Variante *SemiSpiral*)[Douglass '04]

In der Phase *Requirements Analysis* (Anforderungsanalyse) werden alle Anforderungen des Systems erfasst, auf Basis derer im folgenden *System Engineering* (Systementwicklung) eine Systemarchitektur festgelegt wird. Auf der Grundlage dieser Informationen wird die Entwicklung einzelner Hardwarekomponenten begonnen. An dieser Stelle geht der Prozess in eine Phase über, bei der verschiedene Disziplinen parallel den bereits oben erwähnten Mikrozyklus nach dem Spiralmodell durchlaufen.

Abbildung 3.19 zeigt den Mikrozyklus aus Sicht der Softwareentwicklung. Er unterteilt sich in fünf primäre *Mikrophasen*, die wiederum in *Subphasen* aufgeteilt werden:

- **Party**: Projektplanung und -assessment sowie Prozessoptimierung.
- **Analyse**: Identifikation der wesentlichen Eigenschaften, die der Prototyp unabhängig von jeder Realisierung erfüllen muss.

- **Design**: Identifikation einer spezifischen optimalen Lösung, die konsistent mit dem Analysemodel ist.
- **Translation**: Erzeugung lauffähiger Einzelkomponenten des Prototyps, die durch *Unit Tests* abgesichert sind.
- **Testing**: Absicherung, dass die Einzelkomponenten zur Architektur passen (*Integration Testing*) und dass der Prototyp sein Ziel erreicht (*Validation Testing*), z. B. die geforderte Performance.

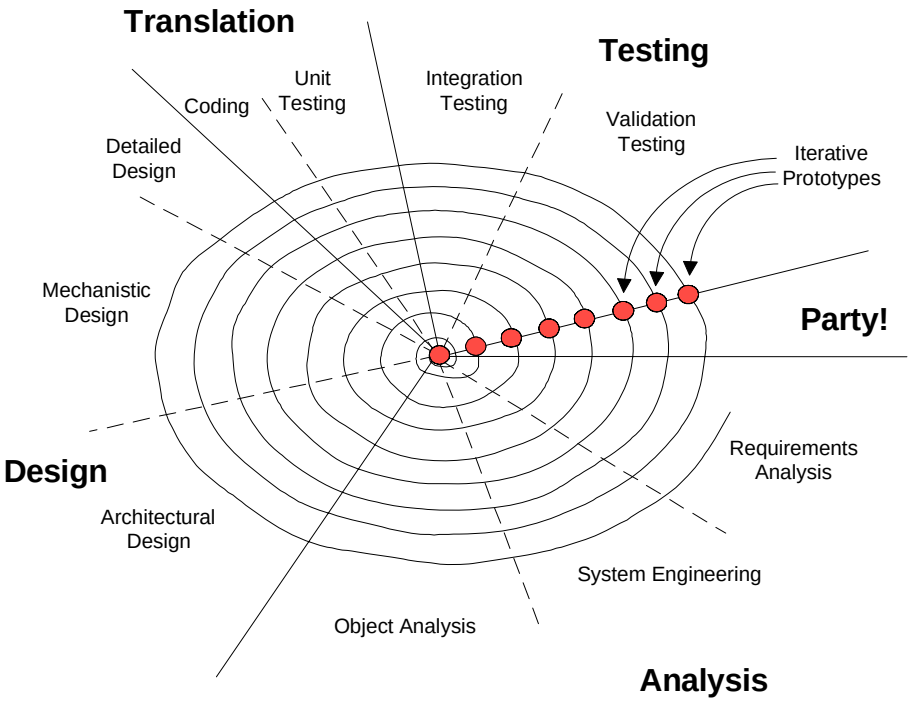


Abbildung 3.19: Mikrozyklus der Softwareentwicklung [Douglass '04]

Eine Mikrophase setzt sich aus *Aktivitäten* zusammen (müssen durch den Entwickler erledigt werden) und erzeugt am Ende *Artefakte* (Ergebnisse von Aktivitäten, die Einfluss auf nachgelagerte Phasen haben). Artefakte stellen weitgehend verschiedene UML-Konstrukte dar, wie beispielsweise Szenarien oder Use-Cases. Abbildung 3.20 zeigt die primären Mikrophasen (Ellipsen) und die ein- und ausgehenden Artefakte (durch horizontale Striche eingerahmt) im Überblick.

Die folgenden Unterkapitel erläutern detaillierter die Aktivitäten der einzelnen Phasen und den Inhalt der Artefakte. Zunächst wird jedoch das in ROPES verwendete Architekturmodell vorgestellt, da dessen Bestandteile im Rahmen der Prozessbeschreibung referenziert werden.

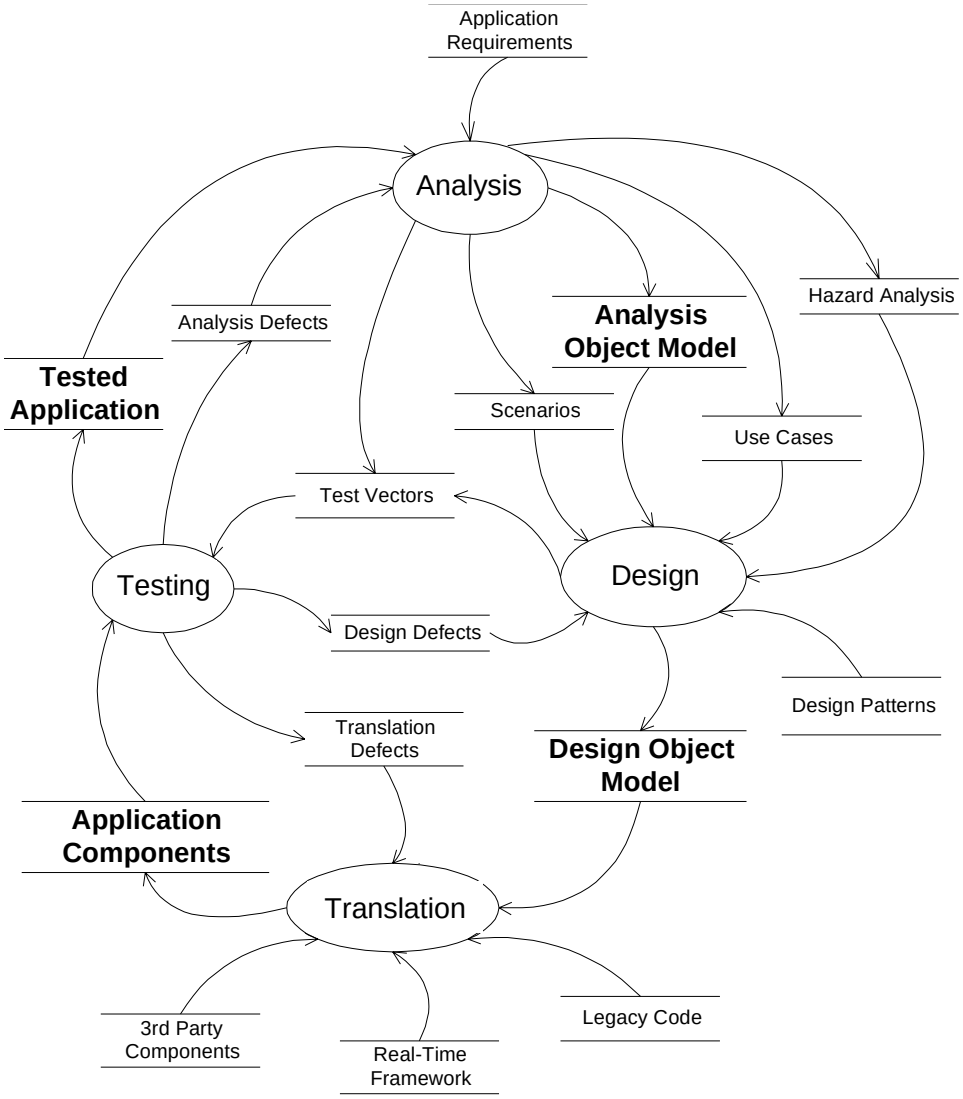


Abbildung 3.20: Phasen und Artefakte des Gesamtprozesses [Douglass '99]

3.2.2.b Architekturmodell

Im Kontext von Architekturen wird bei ROPES von einer *Logical* und einer *Physical Architecture* gesprochen. Die *Logical Architecture* beschreibt, wie Elemente, die zur Entwicklungszeit existieren (z. B. Klassen oder einfache Datentypen), in Modellen organisiert werden. Einfluss auf diese Struktur kann z. B. die Zusammensetzung der Entwicklungsorganisation haben. Die *Physical Architecture* beschreibt dagegen die Struktur der Elemente, die zur Laufzeit existieren (z. B. Subsysteme, Tasks und Objekte). Mit einer *logischen* Sicht wird bei ROPES also nicht eine realisierungsabhängige funktionale Beschreibung des Systems verbunden, so wie es bei dem Kernprozess der Systementwicklung der Fall ist (siehe Kapitel 3.1).

Als zentrales Strukturierungsmittel für die logische Architektur wird das Konzept der *Domäne* empfohlen. Unter einer Domäne wird ein abgrenzbarer Themenbereich verstanden, der über ein

eigenes Fachvokabular verfügt. Beispiele für Domänen sind Fahrregelung, Benutzerschnittstelle, oder Datenmanagement. Für die Repräsentation von Domänen auf UML-Ebene werden Pakete verwendet. Um zu kennzeichnen, dass in dem Paket nur Elemente der Domäne platziert werden sollen, wird der Stereotyp <<domain>> verwendet.

Die *Physical Architecture* bezieht sich auf die Organisation der Elemente des Systems, die zur Laufzeit existieren. Typische Elemente sind Subsysteme, Komponenten und aktive Objekte. ROPES sieht für die Betrachtung dieser Architektur unterschiedliche Abstraktionsebenen vor, die in Abbildung 3.21 gezeigt werden.

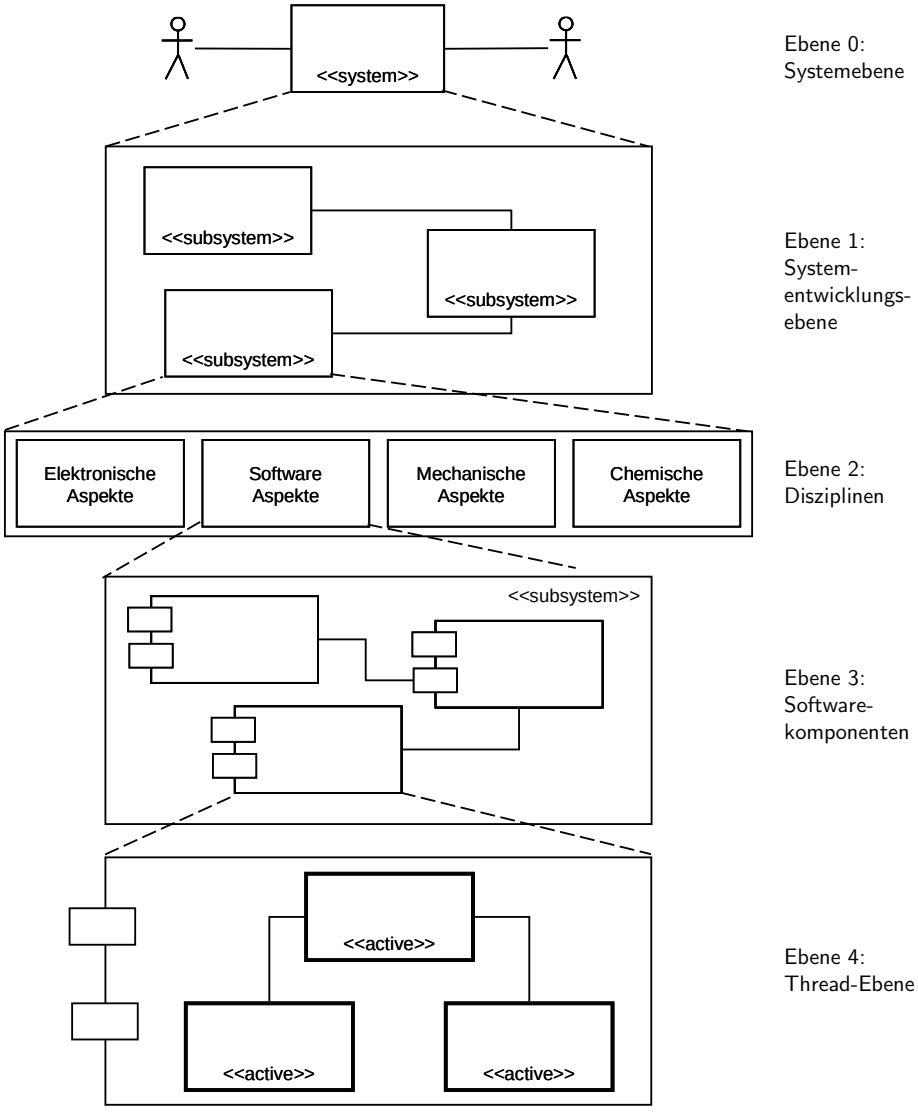


Abbildung 3.21: Abstraktionsebenen der Architektur [Douglass '02]

Die oberste Ebene (Ebene 0) umfasst das gesamte System, wie z. B. ein Fahrzeug oder Flugzeug. Die darunter liegende Ebene (Ebene 1) betrachtet Subsysteme und Ihre Schnittstellen. Mögliche Subsysteme im Falle eines Fahrzeugs wären z. B. „Antriebstrang“ oder „Fahrwerk“. Subsysteme können beliebig verschachtelt sein (z. B. „Antriebstrang“ in „Motorsteuerung“ und

„Getriebe“) und trennen i. d. R. noch nicht nach Hard- und Softwareaspekten. Diese Auftrennung in unterschiedliche Disziplinen erfolgt auf Ebene 2. In der Beschreibung zu ROPES werden hier die Disziplinen Elektrik/Elektronik, Mechanik, Software und Chemie genannt. Die oben erwähnte Motorsteuerung kann z. B. in Elektrik/Elektronik-Aspekte (Sensoren für z. B. Luftmasse oder -druck, Steuergerät, Aktuatoren wie z. B. Zündkerze oder Einspritzpumpe), Mechanik-Aspekte (Kolben, Nockenwelle, Ventil etc.), Chemie-Aspekte (Mischverhältnisse, katalytische Abgasreinigung) und Software (Steuerungs- und Regelungsprogramm, das mit Hilfe der beteiligten Elektrik/Elektronik Einfluss auf die Mechanik nimmt, die wiederum Wechselwirkung mit den chemischen Prozessen hat) aufgeteilt werden.

Die Software eines Subsystems kann auf der darunter liegenden Ebene 3 in Software-Komponenten untergliedert werden. Dabei handelt es sich um austauschbare Softwareteile auf hoher Ebene, die das Subsystem ausmachen. Im Falle des Motorsteuergeräts könnten das die unterschiedlichen Steuerungs- und Regelungsfunktionen (Regelung der Aufladung, Leerlaufdrehzahlregelung, Lambdaregelung), eine Komponente für den Umgang mit dem CAN-Bus und eine Diagnosekomponente sein.

Auf der untersten Ebene werden Prozesse bzw. Threads und ihre Nebenläufigkeit betrachtet. Mindestens eine Komponente im System erzeugt einen Thread (in der UML durch den Stereotyp <<acitve>> gekennzeichnet), andere können sich in dem Sinne passiv verhalten, dass sie nur auf Aufrufe von aktiven Komponenten reagieren. Threads enthalten die Laufzeitobjekte, die in dieser Abstraktionshierarchie die kleinste Einheit darstellen.

Nicht in allen Projekten müssen alle Abstraktionsebenen betrachtet und modelliert werden. In reinen Softwareprojekten, die existierende Standardhardware verwenden, können beispielsweise die Ebenen für Softwarekomponenten und Threads ausreichen.

Parallel zu der in Abbildung 3.21 dargestellten Zerlegung der Architektur, existieren in ROPES fest definierte Sichten (*Views*), die die Architektur in Hinblick auf eine ganz bestimmte Fragestellung hin betrachten. Sichten basieren in einem Projekt immer auf demselben Modell, filtern jedoch nur die Anteile heraus die für die Fragestellung relevant sind. ROPES verwendet die folgenden fünf Sichten:

- **Subsystem and Component View**: Zeigt, wie das System auf oberste Ebene in Subsysteme und Komponenten unterteilt wird. Diese Unterteilung erfolgt vor der Trennung in Disziplinen, so dass z. B. ein Subsystem sowohl Hard- als auch Softwareaspekte umfassen kann. Bei reinen Softwareprojekten kann sich diese Unterteilung auf Softwarekomponenten beziehen. Diese Sicht korrespondiert mit den gleichnamigen Abstraktionsebenen aus Abbildung 3.21 und ist die Grundlage für alle weiteren Sichten. Die Darstellung erfolgt in UML-Klassen- und Komponentendiagramme.
- **Concurrency and Resource View**: identifiziert nebenläufige Tasks, zeigt den grundsätzlichen Ansatz für das *Scheduling* und Regeln für die Synchronisierung und das Ressourcenmanagement. Diese Themen werden primär im Rahmen der Softwareentwicklung gesehen, können jedoch Auswirkung auf die Wahl des Prozessortyps und der Busvernetzung haben.

Die Darstellung dieser Sicht erfolgt mit UML-Klassendiagrammen, in denen für Klassen die Stereotypen <<active>>, <<task>> und <<resource>> verwendet werden.

- **Distribution View**: Identifiziert Objekte, die in unterschiedlichen Adressräumen erzeugt werden und zeigt, wie diese über Adressraumgrenzen hinweg kommunizieren. Zu den dargestellten Sachverhalten gehören die verwendeten Protokoll-Stacks und eingesetzte Kommunikations-Designmuster (siehe Kapitel 3.2.4.b). Die Darstellung erfolgt durch Klassendiagramme, in denen die relevanten Klassen und ihre Beziehungen hervorgehoben werden. In Abhängigkeit der verwendeten Kommunikationstechnologie können Stereotypen zum Einsatz kommen.
- **Safety and Reliability View**: Zeigt, wie Fehler im System zur Laufzeit identifiziert, isoliert und behandelt werden. Zu den Inhalten dieser Sicht gehören das Aufzeigen von redundanten Strukturen und Überwachungsmechanismen sowie die Beschreibung des Verhaltens im Fehlerfall. Dies erfolgt mit UML-Klassendiagrammen, in denen manche Elemente mit Stereotypen ausgezeichnet sind (z. B. werden redundante Subsysteme mit <<channel>> markiert).
- **Deployment View**: Zeigt, wie sich die Elemente der Softwarearchitektur auf die Hardwareeinheiten verteilen. Dies erfolgt mit Hilfe von UML-Verteilungsdiagrammen (engl. *deployment diagram*), bei denen Hardwareeinheiten als *Node*s dargestellt werden, die miteinander verknüpft sind. Durch die Platzierung zuvor modellierter Komponenten innerhalb der *Nodes* wird die Software der Hardware zugeordnet.

Das Architekturmodell und die erwähnten Sichten werden in unterschiedlichen Phasen verwendet. Im Folgenden werden die Mikrophasen nach ROPES genauer erläutert und der Bezug zu den Architekturaspekten hergestellt.

3.2.2.c Analyse

Die Analysephase identifiziert die essentiellen Eigenschaften, die jede technische Umsetzung des zu entwickelnden Systems erfüllen muss. Das Analysemodell soll keine Angaben zu einer technischen Lösung enthalten und keine Optimierungsentscheidungen treffen. Die Analyse wird nochmals in drei Subphasen unterteilt: *Requirements Analysis*, *System Engineering* und *Object Analysis* (siehe Abbildung 3.22).

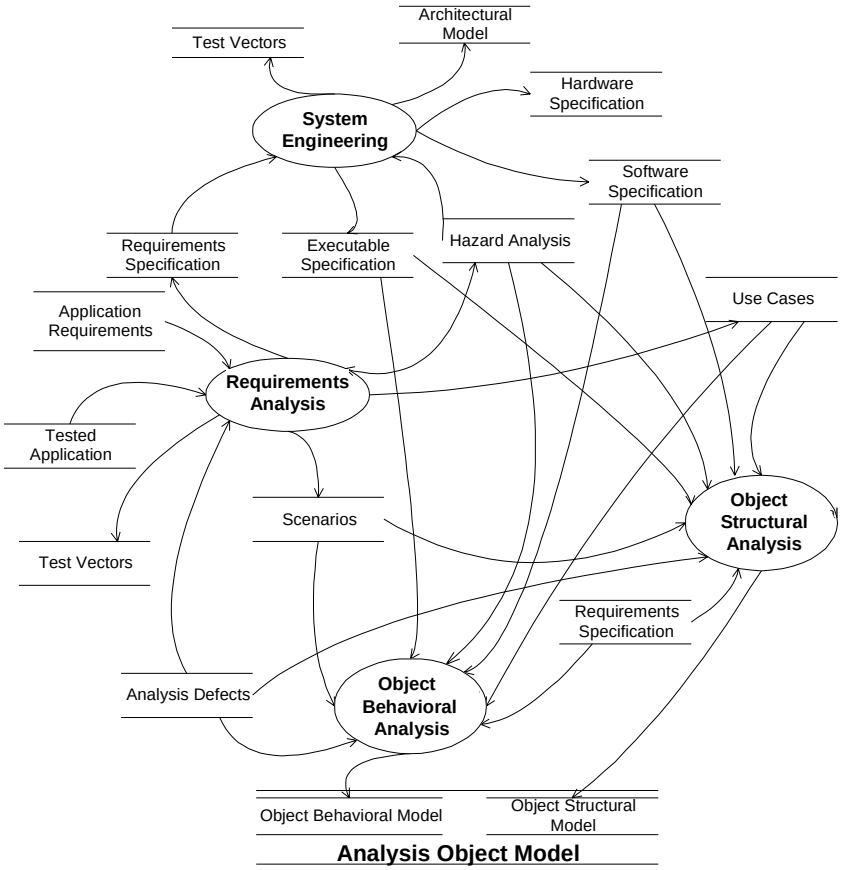


Abbildung 3.22: Artefakte der Phase *Analysis* [Douglass '99]

Requirements Analysis

In der Anforderungsanalyse werden alle funktionalen und nichtfunktionalen Kundenanforderungen strukturiert und in eine möglichst konsistente Form gebracht. Zu den *Aktivitäten* gehören das Identifizieren und Beschreiben von Use-Cases, die sich aus den Gesprächen mit dem Kunden ergeben. Dazu müssen unter anderem die externen Ereignisse, die das System beeinflussen, identifiziert werden. Mit der Aufstellung von Szenarien kann das Verhalten der identifizierten Use-Cases genauer beschrieben werden. Bei sicherheitsrelevanten Anwendungen oder bei Systemen, bei denen ein Ausfall mit hohen Kosten verbunden ist, müssen Gefährdungssituationen und Risiken analysiert werden (engl. *hazard analysis*). Schließlich müssen Randbedingungen des zu erstellenden Systems erfasst werden, wie z. B. vorgegebene Schnittstellen oder Performance-Anforderungen.

Zu den resultierenden *Artefakten* dieser Phase sind an erster Stelle die Use-Cases zu nennen. Eine Use-Case-Beschreibung umfasst ein Use-Case-Diagramm, Zustandsdiagramme (bei reaktivem Verhalten), eine Liste mit externen Ereignissen und ein Kontextdiagramm. Bei dem Kontextdiagramm handelt es sich um ein UML-Objektdiagramm mit einem „System"-Objekt und Akteuren, die mit dem System durch den Austausch von Nachrichten interagieren. Weiteres Artefakt ist die Sammlung von *Scenarios* (Szenarien), die in Form von Sequenz- und Timing-Diagrammen beschrieben werden. Das Ergebnis der *Hazard Analysis* wird als Tabelle

dargestellt, in der die Gefährdungen zeilenweise aufgelistet sind und in den Spalten Attribute wie z. B. Fehlertoleranzzeiten vermerkt sind. Mit *Test Vectors* (Testvektoren) werden in ROPES Testbeschreibungen in Textform verstanden, mit denen das zu entwickelnde System gegen die Anforderungen getestet werden kann. Schließlich sieht ROPES mit der *Requirements Specification* (Anforderungsspezifikation) die Erstellung eines klassischen Textdokuments vor, das unter anderem den Inhalt des Systems, die Schnittstellen, die externen Akteure, das von außen sichtbare Verhalten und zu erfüllenden Randbedingungen beschreibt.

In der *Semi-Spiral*-Variante wird die *Requirements Analysis* nur einmal durchlaufen, in der Standardvariante können die Anforderungen mit jeder Iteration erweitert werden.

System Engineering

Die Phase *System Engineering* (in früheren Publikationen auch als „System Analysis“ bezeichnet [Douglass '99]) ist eine optionale Phase für komplexere Projekte wie z. B. in der Luftfahrt- oder Automobilindustrie. Sie partitioniert das System in Subsysteme und identifiziert für die beteiligten Disziplinen (Software, Elektronik, Mechanik und Chemie) die relevanten Anteile (siehe Abbildung 3.21). Ein Hauptziel ist die Klärung von Verantwortlichkeiten zwischen den beteiligen Entwicklerteams. Es handelt sich aber weiterhin um eine funktionale Sicht, in der keine Objektorientierung verwendet wird. Subsysteme werden als Black Box betrachtet, ihr interner Aufbau ist somit unbekannt.

Zu den weiteren *Aktivitäten* gehört:

- Die Definition der Subsystem-Architektur
- Die Definition der Interfaces and Interaktionsprotokolle zwischen den Subsystemen
- Die Zusammenarbeit zwischen den Subsystemen für unterschiedlichen Use-Cases beschreiben
- Use-Cases in Anforderungen an Subsysteme zu zerlegen
- Analyse und Festlegung der Schlüsselalgorithmen (insb. kontinuierliches Verhalten)
- Zerlegung der Subsysteme in ihre technischen Disziplinen

Zu den möglichen resultierenden *Artefakten* gehört das A*rchitectural Model* (Architekturmodel), das in Form von Klassen- bzw. Komponentendiagrammen vorgelegt wird. Die Inhalte dieser Diagramme entsprechen den Erläuterungen zu den Sichten *Subsystem and Component* und *Deployment* (siehe vorheriges Unterkapitel). Die *Executable Specification* (ausführbare Spezifikation) beschreibt das Verhalten auf Subsystem- bzw. Komponentenebene und kann in der Form von Zustandsdiagrammen (für reaktives Verhalten), Aktivitätsdiagrammen (Skizzierung nebenläufiger Algorithmen), Sequenzdiagrammen (für ausgesuchte Pfade innerhalb der Zustandsdiagramme) und mathematischen Modellen (für kontinuierliches Verhalten, z. B. PID-Regler) vorliegen. Bei der *Software* und *Hardware Specification* (Software- und Hardwarespezifikation) handelt es sich um Textdokumente, die detailliert die Anforderungen der zugeordneten Verantwortlichkeiten zu diesen Bereichen beschreiben. Die *Test Vecotrs* (Test-

vektoren) umfassen im Rahmen des *System Engineerings* Sequenzdiagramme (modellieren Szenarien, mit denen das System getestet werden soll) und einen Testplan, der die Testaktivitäten für das Gesamtsystem zeitlich und inhaltlich plant.

Es wird darauf hingewiesen, dass die UML und objektorientierte Modellierung generell in der Phase *System Engineering* weniger akzeptiert ist als in anderen Phasen. Daher sind auch Modelle, die nicht UML-konform sind, ausdrücklich erlaubt. Beispielhaft werden die Werkzeuge *Statemate* und *Simulink* [The Mathworks '04] genannt.

Object Analysis

In dieser Phase werden die zuvor gesammelten Use-Cases, die für die Softwareentwicklung relevant sind, in ein objektorientiertes Modell überführt. Die wesentlichen Konzepte und Objekte, die für das korrekte Verhalten notwendig sind, werden identifiziert und ihr essentielles Verhalten beschrieben. Im Rahmen der Objekt-Strukturanalyse (engl. *object structural analysis*) werden diese Objekte zu Klassen abstrahiert und ihre möglichen Beziehungen modelliert. In dem *Domain Model* werden die Schlüsselkonzepte und das Vokabular einer beteiligten „Domäne" (z. B. User Interface, Hardwareanbindung) in eine projektweit geltenden Klassen-Vererbungshierarchie eingeordnet und vereinheitlicht. Für einzelne Use-Cases werden Objektkollaborationen gebildet. Im weiteren Schritt können an dieser Stelle bereits Schlüsselmethoden und -attribute der Klassen definiert werden. Im Rahmen der parallel durchgeführten Objekt-Verhaltensanalyse (*Object Behavioral Analysis*) wird das Verhalten modelliert, das für eine korrekte Funktion des Systems essentiell ist. Idealisierweise erfolgt dies in einer Umgebung, in der die Modelle ausführbar sind und somit gegen die Use-Case-Beschreibungen validiert werden können.

Das *Object Structural Model* ist das resultierende *Artefakt* der Strukturanalyse. Es besteht aus Klassen- und Objektdiagrammen, die unter anderem die Vererbungshierarchie und die Kollaborationen der Schlüsselklassen und -Objekte zeigen. Das *Object Behavioral Model* besteht als resultierendes Artefakt der Objekt-Vehaltensanalyse aus einer Auswahl aus Zustands-, Aktivitäts-, Sequenz-, Kollaborations- und Zeitdiagrammen.

3.2.2.d Design

Die Designphase definiert eine spezielle auf die Projektziele hin optimierte Lösung, die konsistent mit dem Analysemodell ist. Die Optimierung ist durch nichtfunktionale Anforderungen getrieben, z. B. in den Bereichen Vorhersagbarkeit, Datendurchsatz, Zuverlässigkeit, Sicherheit, Wiederverwendbarkeit, Portierbarkeit, Wartbarkeit, Skalierbarkeit, Komplexität, Ressourcenverbrauch, Energieverbrauch und Entwicklungsaufwand. Die Herausforderung liegt in der Tatsache, dass einige dieser Anforderungen im Konflikt stehen und Designentscheidungen verlangen. Optimierungsmöglichkeiten ergeben sich z. B. bei der Behandlung folgender Fragestellungen:

- Festlegung aktiver Objekte
- Scheduling-Strategien

- Fehlerbehandlungsstrategien
- Kommunikationsmedien und Protokolle
- Speicherverwaltungsstrategien
- Implementierungsstrategien (Pointer, Referenzen, TCP/IP sockets)

Wie in Abbildung 3.23 zu sehen, wird die Designphase in drei Subphasen unterteilt: *Architectural Design* (betrachtet das gesamte Softwaresystem), *Mechanistic Design* (betrachtet Kollaborationen) und *Detailed Design* (betrachtet Klassen).

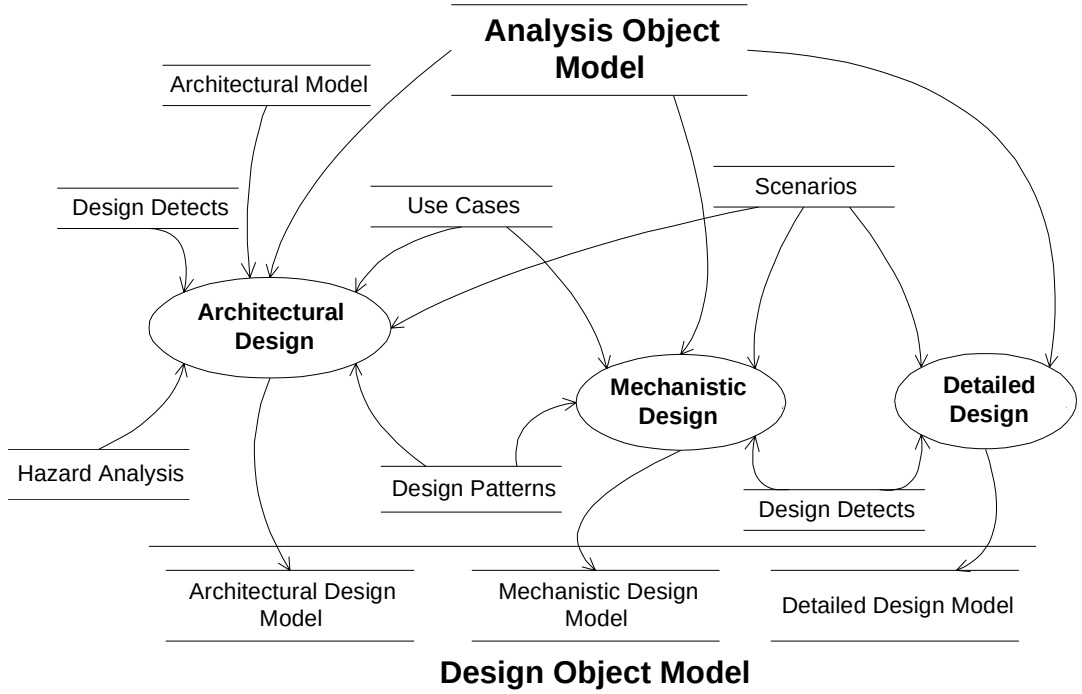


Abbildung 3.23: Artefakte der Phase *Design* [Douglass '99]

Architectural Design

Das *Architectural Design* trifft strategische Designentscheidungen, die die meisten oder alle Softwarekomponenten im System betreffen, wie beispielsweise die Verteilung der Komponenten auf Prozessoren, die Bestimmung von Threads und die Behandlung von Nebenläufigkeitsfragen. Zentrales Designinstrument ist die Anwendung von *Architectural Patterns* (siehe Kapitel 3.2.4). Das resultierende *Artefakt* ist das *Architectural Design Model*, das im Wesentlichen die Diagramme aus dem *System Engineering* und der *Object Analysis* um architekturspezifische Inhalte ergänzt. Dies umfasst z. B. das Hinzufügen von Protokollklassen für die Kommunikation in Multiprozessorumgebungen und das Markieren aktiver Klassen (durch den Stereotyp <<active>>) bei der Festlegung von Threads.

Mechanistic Design

Das *Mechanistic Design* dient der Verfeinerung der Zusammenarbeit zwischen den Objekten. Die *Aktivität* ist die Erarbeitung von einzelnen Kollaborationen durch Hinzufügen von *Glue-*

Objekten wie z. B. Container oder Iteratoren. *Glue-Objekte* sorgen für einen Zusammenhalt von bestimmten, im System existierenden Objekten. Falls z. B. ein Controller viele eintreffende Nachrichten verwalten soll, so ist es notwendig, dass ein Container (FIFO) angelegt wird und zusätzlich ein Iterator, der es ermöglicht, die Einträge des *Glue-Objekts* zu manipulieren. Durch die intensive Nutzung von *Mechanistic Design Patterns* (siehe Kapitel 3.2.4) werden auch in dieser Phase Muster als zentrales Hilfsmittel genutzt. Als *Artefakt* entsteht das *Mechanistic Design Model*, das eine Weiterentwicklung der Klassen-, Sequenz-, Zustands- und Aktivitätsdiagramme aus der vorgelagerten Phase ist.

Detailed Design

Das *Detailed Design* optimiert die interne Struktur und das Verhalten einzelner Klassen. Zu den *Aktivitäten* gehört die Realisierung von Assoziationen, Aggregationen und Kompositionen. Assoziationen können hier beispielsweise als Zeiger, Referenzen oder TCP/IP Sockets verwirklicht werden. Außerdem werden die Sichtbarkeit und der Datentyp von Attributen bestimmt, sowie ihr Gültigkeitsbereich und bestimmte Ausnahmebehandlungen. Der Entwickler kann sich in dieser Phase auf einfache Faustregeln oder Idiome (siehe Kapitel 3.2.4) stützen. Als *Artefakt* entsteht das *Detailed Design Model*, das eine Weiterentwicklung der schon existierenden Struktur- und Verhaltensdiagramme darstellt.

3.2.2.e Translation

In dieser Phase wird aus dem UML-Modell Quellcode erzeugt, der (i. d. R. unter Hinzunahme weiterer Codeteile) kompiliert und gelinkt wird. Nach einem Test der so erzeugten Einheiten können schließlich ausführbare Komponenten zur Verfügung gestellt werden.

Die Erzeugung des Quellcodes kann manuell (*Manuel Code Generation*), automatisiert (*Automatic Code Generation*) oder aus einer Kombination von beiden erfolgen (siehe Abbildung 3.24). Im ersten Fall wird der Code manuell durch Verfeinerung und Ausarbeitung des Analyse- bzw. Designmodells entwickelt. Da hier die Gefahr besteht, dass die Modelle und der Code nicht mehr synchronisiert sind, empfiehlt ROPES grundsätzlich eine automatische Codegenerierung.

Zu den *Aktivitäten* des Entwicklers gehören beim manuellen Ansatz die Erzeugung des Quellcodes für eine objektorientierte Programmiersprache oder eine prozedurale Sprache. Der letzte Fall ist aufwendiger, da die objektorientierten Mechanismen der UML in der Programmiersprache nachgebildet werden müssen und hierfür ein entsprechender *Translation Style-Guide* erarbeitet bzw. vereinbart werden muss.

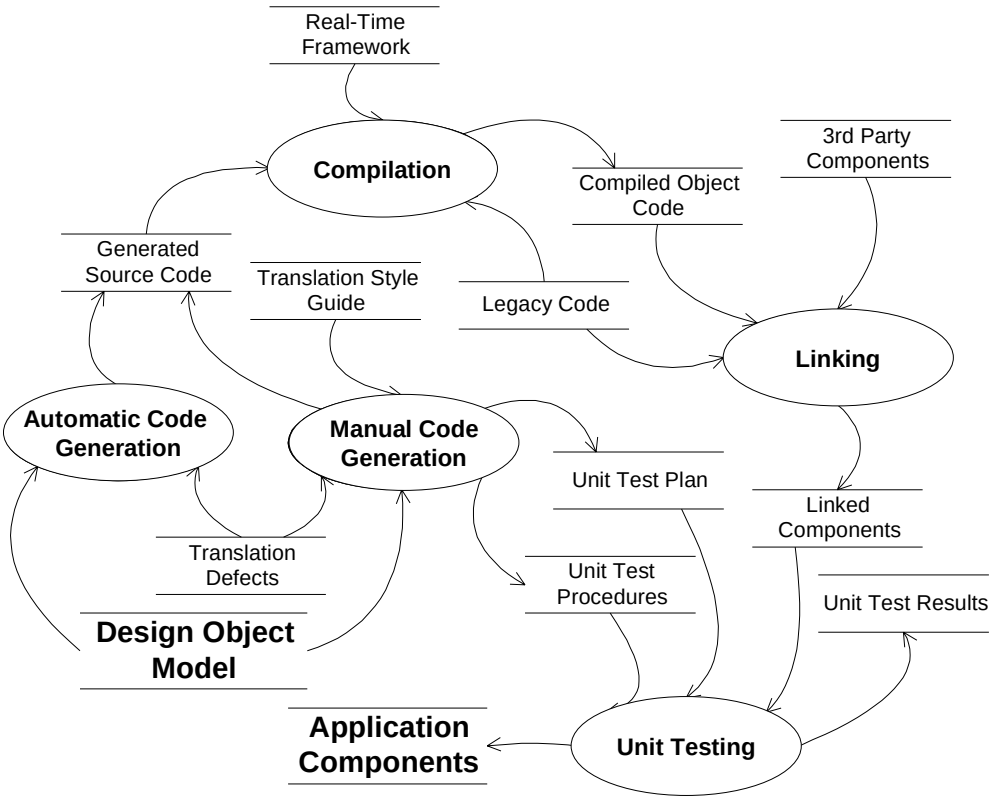


Abbildung 3.24: Artefakte der Phase *Translation* [Douglass '99]

ROPES sieht bereits in der dieser Phase *Unit Testing* vor. Hierbei werden Teile des Codes mit Hilfe einer Anzahl von White-Box-Tests durchlaufen, die sicherstellen, dass die *Units* (Einheiten) intern korrekt sind und dem Design entsprechen. Dies soll eine höhere Qualität und die Verringerung der Entwicklungszeit durch frühes Erkennen von Fehlern bewirken. Zu den resultierenden *Artefakten* dieser Phase gehören neben dem *Generated Source Code* (erzeugter Quellcode) und den üblichen Ergebnissen eines Compiler/Linker-Laufs, die Ergebnisse, die aus dem *Unit Testing* resultieren. Dazu gehören der *Unit Test Plan*, der Testtiefe und -breite jeder *Unit* dokumentiert, die *Unit Test Procedures*, die Schritt für Schritt die Durchführung eines Tests inkl. der „bestanden"-Kriterien dokumentieren und die *Unit Test Results*, ein Dokument über die Testergebnisse mit ausführlichen Informationen für jeden Test (Datum, Name und Version der beteiligten Komponenten, Erfolg/Misserfolg). Die *Application Components* sind ausführbare und getestete Softwarekomponenten, die das Endergebnis dieser Phase darstellen.

3.2.2.f Test

In dieser Phase werden die Entwicklungsergebnisse zu einem Prototyp zusammengeführt, getestet und entdeckte Fehler für die nächste Iteration des Mikrozykluses gesammelt. Das *Testing* unterteilt sich in die Subphasen *Integration Test Design / Integration Testing* und *Validation Test Design / Validation Testing* (siehe Abbildung 3.25).

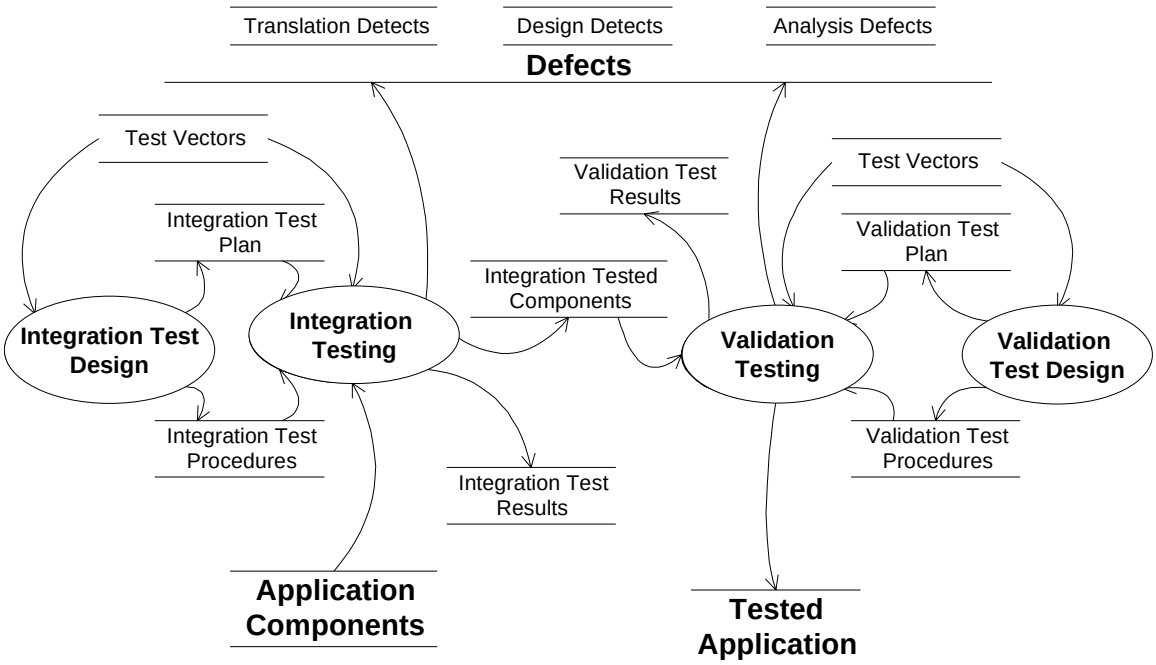


Abbildung 3.25: Artefakte der Phase *Testing* [Douglass '99]

Zu den *Aktivitäten* im Rahmen des *Integration Testing* gehört die Sicherstellung, dass die Einzelteile zusammenpassen. Dazu wird z. B. überprüft, ob Interfaces richtig genutzt werden und keine gesetzten Constraints verletzt werden. Hierfür wird der Prototyp stufenweise „zusammengebaut" und die Software mit der Ziel-Hardware integriert. Im Rahmen des nachgelagerten *Validation Testing* wird nach dem Black-Box-Verfahren überprüft, ob der Prototyp sein Ziel erreicht. Das Ziel eines Prototypen kann z. B. die Erfüllung bestimmter Use-Cases sein oder die Erfüllung bestimmter nichtfunktionaler Anforderungen. Diese Use-Cases und die dazugehörigen *Test Vectors* wurden bereits in der Phase *Requirements Analysis* definiert.

Zu den resultierenden Artefakten gehören die Testpläne (*Integration/Validation Test Plan*), die Beschreibung der Testfälle (*Integration/Validation Test Procedures)* und die Dokumentation der Testergebnisse (*Integration/Validation Test Results)*. Diese Artefakte sind ähnlich wie beim oben erwähnten *Unit Testing* aufgebaut. Während des Tests identifizierte Probleme werden in der Form von *Defects* erfasst, wobei nach *Translation*, *Design* und *Analysis* kategorisiert wird.

3.2.2.g Vergleich mit dem Kernprozess und der Modellbildung in der Systementwicklung

In den Phasen *Requirements Analysis* und *System Engineering* adressiert ROPES die Entwicklung des Gesamtsystems, also auch der Nicht-Softwareanteile wie z. B. Elektronik. Da somit auch Inhalte der Systementwicklung beschrieben werden, stellt sich die Frage, wie der von ROPES beschriebene Prozess zu dem automobilspezifischen Kernprozess der Systementwicklung aus Kapitel 3.1.2 passt. Außerdem definiert ROPES Modelle, die sich der logischen und technischen Systemarchitektur zuordnen lassen. Damit wird die Frage aufgeworfen, welche

Überschneidungen es zwischen den ROPES-Modellen und den in Kapitel 3.1.3.b vorgestellten Architekturbeschreibungssprachen gibt.

Abbildung 3.26 zeigt auf der linken Seite die Prozesse und Artefakte aus dem Kernprozess, auf der rechten Seite Phasen und resultierende Artefakte der ROPES-Methode und in der Mitte die Artefakte der EAST-ADL. Dem Schwerpunkt der Arbeit entsprechend (siehe Kapitel 1.3), beschränkt sich die Gegenüberstellung auf den linken Teil des V-Modells.

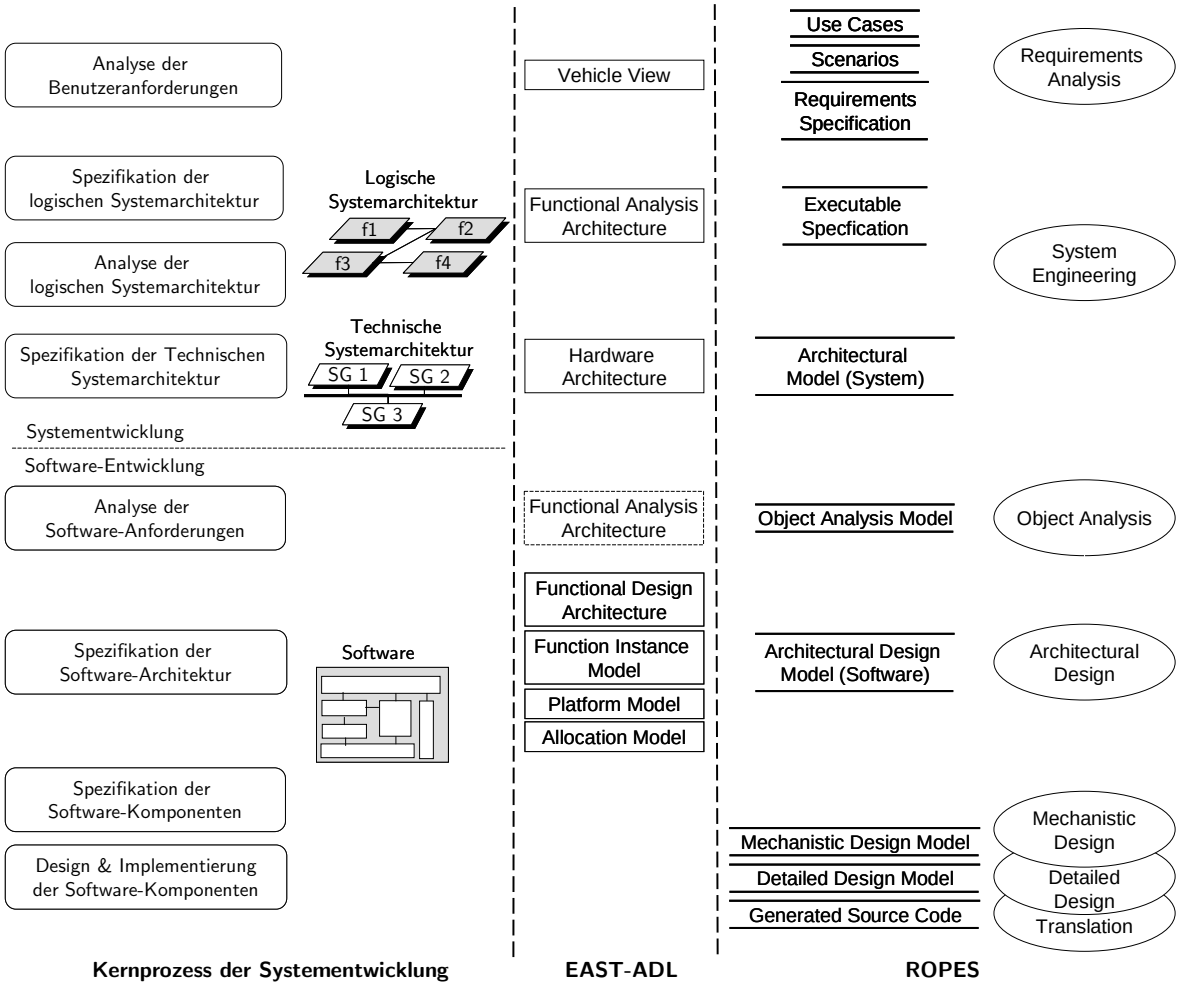


Abbildung 3.26: Gegenüberstellung der Artefakte aus Kernprozess, EAST-ADL und ROPES (linke Seite des V-Modells)

Auf der Prozessebene passt seitens ROPES die Variante *SemiSpiral* (siehe Abbildung 3.9) zu dem Kernprozess. Wie oben erläutert, erfolgen hier die Phasen *Requirements Analysis* und *System Engineering* nach dem Wasserfallprinzip für das Gesamtsystem, bevor die Softwareentwicklung nach dem Spiralmodell den ersten Mikrozyklus durchläuft.

Bei der Beschreibung der Benutzeranforderungen ist bei ROPES die *Requirements Specification* das zentrale Dokument, zu dessen Aufbau und Inhalt jedoch wenig formale Vorgaben gemacht werden. Es wird jedoch empfohlen, das Verhalten des Systems möglich umfassend mit Use-

Cases und Szenarien zu beschreiben. Das Artefakt *Vehicle View* aus der EAST-ADL ist dagegen relativ formalisiert. Es beschreibt die Funktionen eines Fahrzeugs als *Features*, klassifiziert die damit verbundenen Anforderungen nach funktionalen und nichtfunktionalen Anforderungen und stellt umfangreiche Mechanismen zur Verfügung, um die Variabilität eines *Features* zu modellieren.

Auch bei der Beschreibung der logischen Systemarchitektur sind die Vorgaben der EAST-ADL formaler. Das Metamodell der *Functional Analysis Architecture* macht sehr konkrete Angaben, wie Funktionen und ihre Kommunikationsbeziehungen modelliert werden. Bei der Verhaltensbeschreibung verweist die EAST-ADL auf externe Werkzeuge wie z. B. *Simulink*. ROPES schlägt für die Strukturbeschreibung (insb. Festlegung von Subsystemen) Klassen- und Komponentendiagramme vor, ohne jedoch konkretere Angaben zu machen. Die *Executable Specification* kann als Verhaltensbeschreibung der logischen Systemarchitektur interpretiert werden und wird ebenfalls nicht konkret festgelegt. Ebenso wie bei der EAST-ADL wird auch bei ROPES auf Werkzeuge wie *Simulink* oder *Statemate* verwiesen.

Bei der Beschreibung der technischen Systemarchitektur stützt sich ROPES vor allem auf UML-Verteilungsdiagramme, bei denen Hardware-Einheiten als *Nodes* beschrieben werden. Kapitel 3.2.1.d kam bereits zu dem Ergebnis, dass sich mit UML-Verteilungsdiagrammen nur sehr begrenzt die E/E-Architektur eines Fahrzeugs modellieren lässt. Die *Hardware Architecture* der EAST-ADL ist hier wesentlich ausdrucksstärker, da sie eigene Metaklassen für Sensoren, Aktuatoren, Bussysteme, einfache elektrische Verbindungen und vieles mehr definiert.

Im Rahmen der Analyse der Softwareanforderungen sieht der Kernprozess kein besonderes Artefakt vor. Im Falle der EAST-ADL können die zu betrachtenden Inhalte der bereits in der Systementwicklung modellierten *Functional Analysis Architecture* entnommen werden. Bei ROPES wird für diese Phase mit dem *Object Analysis Model* ein eigenes Modell erstellt. Bei der Spezifikation der Softwarearchitektur ist der modellierte Umfang ähnlich, wobei die Modelle der EAST-ADL von der Existenz der EAST-Laufzeitumgebung ausgehen und sich auf deren Basisdienste stützen (insbesondere Kommunikationsdienst und Zugriff auf I/O der Hardware). Einen solch festen Bezug zur Laufzeitumgebung gibt es bei ROPES nicht.

Für die Modellierung der Struktur des Innenlebens einer Softwarekomponente stellt die EAST-ADL keine Artefakte zur Verfügung. Die Designoptimierung und Implementierung einer Softwarekomponente ist somit auf Basis der Modelle der EAST-ADL nicht möglich. Die im Rahmen von ROPES verwendeten Modelle legen hier wiederum einen Schwerpunkt.

Insgesamt lässt sich festhalten, dass ROPES für die systementwicklungsrelevanten Artefakte keine bestimmten Modelle vorschreibt und auch für Architekturbeschreibungssprachen wie die EAST-ADL offen ist. Bei der Softwareentwicklung sind die zu verwendenden Modelle dagegen enger gefasst und genauer beschrieben. System- und Softwareentwicklung sind nur lose gekoppelt. Da bei den Modellarten in der Systementwicklung weitgehend Wahlfreiheit herrscht, ist es mit den im Rahmen der Methode zu Verfügung stehenden Mitteln auch nicht möglich, den Zusammenhang zwischen Modellen der System- und Softwareentwicklung systematisch zu

beschreiben. Die Softwareentwicklung beginnt (konsequenterweise) mit der Erstellung eines eigenen Analysemodells (dem *Object Analysis Model*). Hier herrscht ein gewisser Bruch in der Durchgängigkeit, da der Bezug dieses Modells zu den Modellen der Systementwicklungen ungeklärt ist.

Die EAST-ADL liefert für alle systementwicklungsrelevanten Artefakte detaillierte Metamodelle. Die Modellierung geht bis zur Spezifikation der Softwarearchitektur und der darin enthaltenen Softwarekomponenten mit ihren Schnittstellen. Modelle für das interne Design der Komponenten stehen dagegen nicht zur Verfügung.

Zusammenfassend kann festgestellt werden, dass die EAST-ADL konkrete Modelle für die Artefakte (bezogen auf den Kernprozess) der Systementwicklung liefert. ROPES liefert dies für die Softwareentwicklung. Im Bereich der Softwarearchitektur gibt es inhaltliche Überlappungen. Die Formalisierungslücke, die bei ROPES in der Systementwicklung durch die Wahlfreiheit bei den Modellen herrscht, kann durch die EAST-ADL geschlossen werden. Bezogen auf die Problemstellung dieser Arbeit ergibt sich damit Folgendes: Durch die Kombination aus EAST-ADL und ROPES stehen sowohl für System- und Softwareentwicklung detaillierte Modelle zur Verfügung, so dass der Kernprozess weitergehend modellbasiert gestaltet werden kann. Wie die Modelle methodisch integriert werden können, ist noch ungeklärt. Insbesondere bei der Frage des Zusammenhangs der Modelle zwischen System und Softwareentwicklung ergeben sich keine neuen Erkenntnisse.

Die weiteren Untersuchungen dieser Arbeit gehen von einer aus EAST-ADL und ROPES (UML) kombinierten Modelllandschaft aus und fokussieren sich auf die Entwicklungsaktivitäten, die zwischen einer vollendeten *Functional Analysis Architecture* und der Fertigstellung des *Design Object Models* anzusiedeln sind. Das *Object Analysis Model* soll in einer angepassten Form die Inhalte der *Functional Analysis Architecture* weitgehend übernehmen. Die in der Problemstellung angedachte Modelltransformation soll ROPES dahingehend erweitern, dass das *Design Object Model* automatisiert aus dem *Object Analysis Model* abgeleitet werden kann. Dies soll unter einer möglichst breiten Ausnutzung der in den EAST-ADL-Modellen festgehaltenen Informationen bzgl. der technischen Systemarchitektur geschehen. Eine Konvertierung der technischen Systemarchitektur in UML-Verteilungsdiagramme wird nicht für sinnvoll erachtet, da die hierzu notwendige Festlegung eines Profils den Rahmen dieser Arbeit sprengen würde, zumal es hier das erwähnte ATESST-Projekt mit dieser Zielstellung gibt. Es wird vielmehr davon ausgegangen, dass die EAST-ADL-Modelle im Rahmen der Softwareentwicklung in ihrer ursprünglichen Form verwendet werden.

3.2.3 Weitere UML-basierte Methoden

Neben ROPES gibt es zahlreiche weitere Softwareentwicklungsmethoden, die UML-Modelle verwenden. Sehr bekannt ist der *Rational Unified Process* (RUP) [Kruchten '04]. Er ist eine konkrete Umsetzung des *Unified Process* [Jacobson et al. '99], der parallel zur UML entwickelt und veröffentlicht wurde und dessen Autoren Ivar Jacobson, Grady Booch und James Rumbaugh auch zu den Protagonisten der UML-Standardisierung gehörten. RUP wurde durch

die Firma *Rational*, die heute zu der Firma *IBM* gehört, kommerzialisiert und wird bis heute mit speziellen Werkzeugen unterstützt.

Zu den Kernprinzipien des *Unified Process* gehören eine iterative und inkrementelle Vorgehensweise, die Beschreibung und Ausrichtung vieler Aktivitäten auf Anwendungsfälle und objektorientierte Analyse und Design. Hier gibt es somit eine große Ähnlichkeit zu ROPES. Ein Unterschied ergibt sich aber durch die Tatsache, dass ROPES speziell für die Entwicklung eingebetteter Software mit Echtzeitanforderungen konzipiert wurde und dadurch verstärkt z. B. Hardware-, Sicherheits- und Verfügbarkeitsaspekte betrachtet. Bei den Systemen, die in Publikationen zu RUP beschrieben werden, handelt es sich überwiegend um PC- oder Server-Anwendungen. Neben dieser unterschiedlichen Schwerpunktbildung bzgl. der Domäne sind aber auch in der Prozessgestaltung im Detail Unterschiede zu finden.

Eine weitere UML-basierte Methode, die speziell für die Entwicklung verteilter Echtzeitanwendungen mit Nebenläufigkeit optimiert wurde, ist COMET [Gomaa '01]. Wie bei ROPES und RUP ist der Prozess iterativ und durch Use-Cases getrieben, die im Rahmen der Anforderungsphase modelliert werden. In der Analysephase werden statische und dynamische Modelle entwickelt. Das statische Modell definiert strukturelle Beziehungen zwischen Klassen, die mit Hilfe durch die Methode bereitgestellter Strukturierungskriterien gebildet werden. Die Modellierung der Dynamik erfolgt mit Objektmodellen die zeigen, wie Objekte an jedem Use-Case partizipieren und wie sie dabei interagieren. Zustandsbehaftete Objekte werden durch Zustandsdiagramme beschrieben. In der Design-Phase wird zunächst ein Architekturmodell erstellt. Bei der Bildung von Subsystemen kommen wieder spezielle Strukturierungskriterien zum Einsatz. Ein besonderes Gewicht der Methode liegt auf der Bildung von Komponenten bei verteilten Systemen. Hierzu gehören Regeln für die Aufteilung von Verantwortlichkeiten zwischen Clients und Servern und die Frage, ob Daten und Steuerungsaufgaben zentralisiert oder verteilt werden. Ein weiteres ausführlich behandeltes Thema ist die Auslegung von Kommunikationsschnittstellen hinsichtlich unterschiedlicher Kommunikationsmodelle. Die Festlegung von aktiven Objekten, die Bildung von Tasks und Spezifikation der Inter-Task-Kommunikation und -Synchronisation wird ebenfalls methodisch adressiert. Eine Besonderheit ist die Festlegung eines Verfahrens für die Laufzeitabschätzung, dass sich insbesondere für Systeme mit „harten“ Echtzeitanforderungen (eine vorgegebene Zeitschranke muss unbedingt eingehalten werden) eignet.

COMET adressiert in besonderem Maße Fragen, die bei hochverteilten nebenläufigen Systemen behandelt werden müssen, hat aber ansonsten mit ROPES und RUP viele Ähnlichkeiten. ROPES differenziert sich in diesem Vergleich etwas durch den ausführlich beschriebenen Einsatz von Mustern, die als ein zentrales Hilfsmittel der Methode verwendet werden.

3.2.4 Muster

Zu den bereits in Kapitel 1.1 angesprochenen Prinzipien des Software Engineerings gehört auch die Wiederverwendung [Boehm '83]. Durch Wiederverwendung sollen Zeitaufwand, Kosten und Qualitätsrisiken reduziert werden. Auf der Code-Ebene konnte sich dieses Prinzip z. B. in der

Form von Programmbibliotheken relativ früh etablieren. Die seit Anfang der 1990er Jahre publizierten *Muster*-Ansätze versuchen Wiederverwendung auch in früheren Phasen zu realisieren, insbesondere im Design. Der Schwerpunkt bei Mustern liegt nicht in der Lieferung von Lösungen, die 1:1 übernommen werden können, sondern eher in der Wiederverwendung von Lösungskonzepten bzw. Lösungsideen, die zu wiederkehrenden Problemstellungen der jeweiligen Entwicklungsphase passen.

Eine Definition, die zu allen Musterarten (siehe folgendes Unterkapitel) passt, kann wie folgt formuliert werden:

Ein Muster ist eine wiederverwendbare Lösungsvorlage für ein wiederkehrendes Entwicklungsproblem.

Eine Lösungsvorlage ist in diesem Zusammenhang *wiederverwendbar*, wenn sie ihre Tauglichkeit bei einem Projekt in der Vergangenheit nachgewiesen hat und außerdem gezeigt werden konnte, dass sie sich auch bei anderen Systemen anwenden lässt. *Wiederkehrend* heißt, dass das Problem bei der Entwicklung mehrerer Systeme auftreten muss, da sonst niemand von der Musterbeschreibung profitieren würde. Ein *Entwicklungsproblem* kann in jeder Phase eines Entwicklungsprozesses auftreten, ist *kontextabhängig* und muss durch einen Entwickler bearbeitet werden.

Die durch die Muster beschriebenen Lösungen sind somit keine Innovationen, sondern verallgemeinern lediglich Lösungen, die sich in der Vergangenheit bewährt haben. Aus diesem Grund kann auch von der Systematisierung von *best practices* (in der Praxis bewährte Vorgehensweisen) gesprochen werden.

Neben der Wiederverwendung von Lösungskonzepten verspricht man sich von Mustern auch Vorteile in der Kommunikation zwischen Entwicklern. Da Muster einen Namen haben und im optimalen Fall allen Beteiligten bekannt sind, können Muster bei der Diskussion von Lösungsalternativen wie ein Glossar fungieren, der die Erklärung von Lösungsideen vereinfacht. Muster können demnach auch als wichtige Elemente einer Fachsprache für Entwickler verstanden werden.

Zu Mustern gibt es seit einiger Zeit eine Community mit eigenen Konferenzen und Veröffentlichungsreihen. Eine zentrale Anlaufstelle ist die Webseite der *Hillside Group* [Hillside Group]. Nach den Regeln der Community unterliegt die Entstehung und Verwendung von Mustern einem bestimmten Lebenszyklusmodell. Bei [Yacoub et al. '04] wird dieses mit drei Phasen beschrieben:

- **Mining** (Entdeckung): In der ersten Phase wird ein Muster zum ersten Mal dokumentiert. Wenn in der Praxis wiederholt ein bestimmtes Designproblem auftritt und eine Lösung bekannt ist, die für dieses Problem in anderen Projekten erfolgreich war, ist ein Musterkandidat gefunden. Eine Herausforderung in dieser Phase ist es, bei der Beschreibung die richtige Verallgemeinerung zu finden.

- **Polishing** (Polierung): In der zweiten Phase wird der Entwurf der Musterbeschreibung evaluiert und verbessert. Dies geschieht durch Einreichung zur Begutachtung bei einer PLoP-Konferenz [Hillside Group], bei dem das Muster nach einem für die Muster-Community spezifischen Prozess gereviewed wird. In einem ersten Schritt unterstützt ein erfahrener Musterexperte (s*hepperd*) den Autor bei der Verbesserung der Musterbeschreibung. Wird am Ende dieses Schrittes der Beitrag angenommen, erfolgen im zweiten Schritt die Diskussion und die Einarbeitung weitere Verbesserungsvorschläge auf der Konferenz. Mit der Veröffentlichung in den PLoP-Proceedings endet die Phase.
- **Reuse** (Wiederverwendung): In der dritten Phase ist das Muster in den PLoP-Proceedings veröffentlicht. Bei der Recherche nach Lösungen für ihr Problem stoßen Entwickler auf das Muster und wenden es an. Ergeben sich bei der Verwendung in ihrem Projekt Verbesserungsvorschläge, werden diese idealerweise an den Autor zurückgemeldet.

3.2.4.a Historie und Musterarten

Der Musteransatz innerhalb des Software Engineering wurde durch Arbeiten des Architekten Christopher Alexander aus den 1970er Jahren inspiriert [Alexander '77]. Er versuchte mit seinen Ansätzen, Architekturen von Bauwerken in elementare Bestandteile und ihre Beziehungen aufzulösen und das Ergebnis zu katalogisieren. Der durch Alexander geprägte Musterbegriff wurde 1987 von Kent Beck und Ward Cunningham [Beck et al. '87] aufgegriffen, um Muster für die Erstellung von graphischen Benutzungsschnittstellen in Smalltalk zu beschreiben. Mit dem Buch *Advanced C++ Programming Styles and Idioms* veröffentlichte James Coplien 1991 einen sehr musterähnlichen Ansatz für die Entwicklung in C++ [Coplien '91]. Der Durchbruch für Muster in der Softwareentwicklung gelang aber 1995 mit dem Buch *Design Patterns – Elements of Reusable Object-Oriented Software* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides [Gamma et al. '95] – oft als *Gang of Four* (GoF) bezeichnet. Die 23 Designmuster (auch als „Entwurfsmuster" bezeichnet), die in diesem Buch beschrieben sind, besitzen heute noch einen hohen Bekanntheitsgrad und sind Grundlage zahlreicher weiterer Arbeiten.

Der Erfolg der Designmuster hat viele Autoren zur Entwicklung weiterer Musterarten inspiriert. Häufig werden die verfügbaren Musterarten in folgende Hauptkategorien unterteilt (z. B. [Yacoub et al. '04], [Douglass '02]):

- **Analysemuster** beschreiben für einen wiederholt anzutreffenden Sachverhalt einer Domäne eine Lösungsvorlage, um diesen in einem Analysemodell zu erfassen. Fowler beschreibt Analysemuster als eine „Idee, die in einem praktischen Kontext hilfreich war und dies wahrscheinlich auch in einem anderen sein könnte" [Fowler '97]. Beispielmuster sind *Account*, *Accounting Transaction* oder *Measurement*. In der ROPES-Methode (siehe Kapitel 3.2.2) sind Analysemuster nicht explizit Teil der Methode, passen aber zur Phase *Object Analysis*.
- **Architekturmuster** beschreiben wiederkehrende grundsätzliche Organisationsprinzipien für die Architektur einer Softwareanwendung. Nach Buschmann et al. drücken sie „fundamentale strukturelle Organisationsschemata für Softwaresysteme aus. Sie liefern eine

Menge vordefinierter Subsysteme, spezifizieren deren Verantwortlichkeiten und beinhalten Regeln und Richtlinien für deren Beziehungen untereinander“ [Buschmann '98]. Beispielmuster sind *Layers*, *Broker* und *Mikrokernel*. In der ROPES-Methode (siehe Kapitel 3.2.2) werden Architekturmuster in der Phase *Architectural Design* verwendet.

- **Designmuster** liefern für wiederkehrende Designprobleme eine Lösungsvorlage für das Designmodell in der Form von Klassen- bzw. Objektkollaborationen. Für die GoF beschreiben Designmuster „eine allgemein wiederkehrende Struktur aus kommunizierenden Komponenten, die ein generelles Designproblem für einen bestimmten Kontext lösen“ [Gamma et al. '95]. Buschmann et al. definieren ein Designmuster als ein „Schema für die Verfeinerung von Subsystemen oder Komponenten eines Softwaresystems oder die Beziehungen zwischen ihnen“ [Buschmann '98]. Beispielmuster sind *Strategy*, *State* und *Proxy*. In der ROPES-Methode (siehe Kapitel 3.2.2) werden Designmuster in der Phase *Mechanistic Design* verwendet, weswegen sie hier auch als *Mechanistic Design Patterns* bezeichnet werden.
- **Idioms** liefern für wiederkehrende Designprobleme ein Codegerüst in einer bestimmten Programmiersprache. Es wird beschrieben, wie bestimmte Aspekte von Komponenten oder Beziehungen zwischen ihnen unter Ausnutzung besondere Eigenschaften der Programmiersprache, implementiert werden können. Beispiele sind *Singleton* in C++ [Buschmann '98] und *Counted Pointer* [Coplien '91]. In der ROPES-Methode (siehe Kapitel 3.2.2) passen Idioms zur Phase *Translation*, in einem gewissen Umfang auch zum *Detailed Design*.

Eine weitere Differenzierung in Musterarten ist nach der Domäne möglich. Tabelle 3.8 listet für die Hauptkategorien Musterkataloge verschiedener Autoren auf und ordnet sie einer Domäne zu. Die Auswahl ist ungeordnet und beispielhaft. Entsprechend dem Schwerpunkt dieser Arbeit sind aber vor allem Muster aus dem Bereich der eingebetteten Steuerungs- und Regelungssysteme gewählt worden. Weitere hier nicht aufgeführte Muster aus diesem Bereich finden sich z. B. bei [Kircher et al. '02, Kircher et al. '01].

Tendenziell nehmen die Designmuster in ihrer Beschreibung wenig Bezug auf eine Domäne. Eine Domäne bedient sich zwar oft einer bestimmten Auswahl von Designmustern, diese können jedoch in mehreren Domänen verwendet werden und sind entsprechend auch domänenneutral beschrieben. So spielen Muster, die z. B. Zuverlässigkeitsaspekte adressieren, im Bereich der Steuerungs- und Regelungssysteme genauso eine Rolle, wie bei Systemen aus dem Bankbereich.

Kategorie	Musterkatalog	Domäne
Analysemuster	Fowler: Analysis Patterns [Fowler '97]	Unternehmenssoftware
	Balzert: Analysemuster [Balzert '99]	allgemein
	Konrad et al.: Object Analysis Patterns for Embedded Systems [Konrad et al. '04b]	Eingebettete Steuerungs- und Regelungssysteme
	Coad et al.: Patterns [Coad et al. '97]	allgemein
	Fernandez/Yuan: Semantic Analysis Patterns [Fernandez et al. '00]	Verwaltung / allgemein
	Geyer-Schulz/Hahsler: Analysis Patterns [Geyer-Schulz et al. '01]	Groupware-Systeme
Architekturmuster	Buschmann et al.: Architectural Patterns [Buschmann '98]	allgemein
	Douglass: Real Time Design Patterns [Douglass '02]	Eingebettete Steuerungs- und Regelungssysteme
	Welch/Marinucci: Dynamic Resource Management Architecture Patterns [Welch et al. '02]	Eingebettete Steuerungs- und Regelungssysteme
	Cross: Proactive and Reactive Resource Allocation Patterns [Cross '02]	Eingebettete Steuerungs- und Regelungssysteme
Designmuster	Gamma et al.: Design Patterns [Gamma et al. '95]	allgemein
	Buschmann et al: Design Patterns [Buschmann '98]	allgemein
	Douglass: Mechanistic Design Patterns [Douglass '99]	Eingebettete Steuerungs- und Regelungssysteme
Idioms	Buschmann et al.: Idioms [Buschmann '98]	allgemein (C++)
	Pont: Patterns for Time-Triggered Embedded Systems [Pont '01]	Eingebettete Steuerungs- und Regelungssysteme (C/8051-Prozessor)
	Homann: Muster für OSEK [Homann '04]	Eingebettete Steuerungs- und Regelungssysteme (C/OSEK-Betriebssystem)

Tabelle 3.8: Musterkategorien

Die Beschreibungen von Analysemustern verwenden dagegen oft die Sprache eines Fachgebiets (z. B. Lagerverwaltung oder die hier betrachteten eingebetteten Steuerungs- und Regelungssysteme) und sind somit tendenziell domänenspezifisch. Idiome beschreiben die Lösung immer für eine spezifische Programmiersprache. Die Muster von [Homann '04] beschränken sich darüber hinaus auf ein bestimmtes Betriebssystem und bei [Pont '01] wird der Problembereich weiter auf zeitgesteuerte Systeme eingegrenzt und der Lösungsraum umfasst nur einen bestimmten Mikrocontroller.

Typisch für alle Musterarten ist, dass sie nach einem einheitlichen Schema katalogisiert werden. Der Aufbau dieses Schemas variiert nach Musterart und Autor. Zu den oft verwendeten Kernelementen gehören ein eindeutiger Mustername, Motivation und Ziel des Musters, die Beschreibung der Lösung und eine Diskussion von Vor- und Nachteilen (siehe Kapitel 3.2.4.d). Im Folgenden werden die Analyse- und Designmuster vertieft, da sie im Laufe dieser Arbeit eine besondere Rolle spielen. Die Betrachtung beginnt mit den Designmustern, da in ihrem Umfeld viele Konzepte entstanden sind, die für die Analysemuster übernommen wurden.

3.2.4.b Designmuster

Die hier betrachteten Designmuster werden im ROPES-Prozess (siehe Kapitel 3.2.2) während der Phase *Mechanistic Design* eingesetzt und liefern als Lösung ein Fragment eines Klassenmodells. Zu den Aufgaben dieser Phase gehört die Optimierung einzelner Objekt-Kollaborationen in Hinblick auf nichtfunktionale Anforderungen. Designmuster unterstützen den Entwickler entsprechend dabei, eine ausgewählte Objektkollaboration in Richtung einer einzelnen oder einer Gruppe von nichtfunktionalen Anforderungen zu optimieren. Diese Fokussierung auf bestimmte nichtfunktionale Anforderungen wird notwendig, weil viele nichtfunktionale Anforderungen in einem konfliktären Verhältnis stehen (siehe Kapitel 3.2.2.d). Es ist die Aufgabe des Entwicklers bei der Auswahl von Mustern einen Kompromiss zu finden, der alle nichtfunktionalen Anforderungen des Systems im richtigen Verhältnis adressiert.

Bei diesem Auswahlprozess und der Anwendung des Musters stützt sich der Entwickler auf folgende wichtige Kernelemente einer Musterbeschreibung. Das *Problem* beschreibt die Designaufgabe, zu der das Muster passt – bei Designmustern i. d. R. die Optimierung einer Kollaboration in Richtung einer nichtfunktionalen Anforderung. Im *Kontext* wird die Situation beschrieben, in dem das Muster anwendbar ist. Die *Lösung* beschreibt die Struktur des Musters in der Form eines Klassendiagramms und evtl. das Verhalten mit Hilfe von Sequenz- oder Zustandsdiagrammen. Da die Optimierung des Designs für eine nichtfunktionale Anforderung oft zu Lasten anderer nichtfunktionaler Anforderungen erfolgt, wird mit den *Konsequenzen* die Auswirkung der Musteranwendung auf andere nichtfunktionale Anforderungen erläutert. Diese Kernelemente finden sich direkt oder indirekt auch in den bekannten Schemata für die Beschreibung von Mustern wieder.

Im Rahmen dieser Arbeit ging es unter anderem darum, aus dem vielfältigen Angebot von Designmustern eine geeignete Auswahl für die Domäne *Automotive Software* zu finden. Dazu wurde zunächst eine Übersicht der Arbeiten aus [Gamma et al. '95], [Buschmann '98],

[Douglass '99], [Douglass '02] sowie weiterer Einzelpublikationen aus diversen Musterkonferenzen zusammengestellt. In einem ersten Schritt wurde diese Liste dahingehend verdichtet, dass ähnliche bzw. konzeptgleiche Muster von unterschiedlichen Autoren verknüpft wurden. In einem zweiten Schritt wurde jedes einzelne Muster auf die Verwendbarkeit in der Domäne und insbesondere in der Fallstudie geprüft. Die in Frage kommenden Kandidaten wurden auf einer Skala von 1 (sehr geeignet) bis 4 (nur sehr eingeschränkt geeignet) bewertet und in 4 Kategorien eingeteilt (siehe Abbildung 3.27).

Ein wichtiges Selektionskriterium war dabei die Granularitätsebene des Musters. Muster für das *Mechanistic Design* (siehe Kapitel 4.2.3) beschäftigen sich mit Kollaborationen zwischen Objekten. Architekturmuster oder Idiome wurden daher aussortiert. Insbesondere bei Architekturmustern ist die Abgrenzung zu Mustern des *Mechanistic Design* aber oft fließend. Einige der Muster aus Abbildung 3.27 haben Architekturcharakter, können aber auch für Objekt-Kollaborationen verwendet werden. Die Bewertung erfolgt jedoch in Hinblick auf die Verwendung für das *Mechanistic Design*, da dies die von den Transformationsregeln adressierte Abstraktionsebene ist.

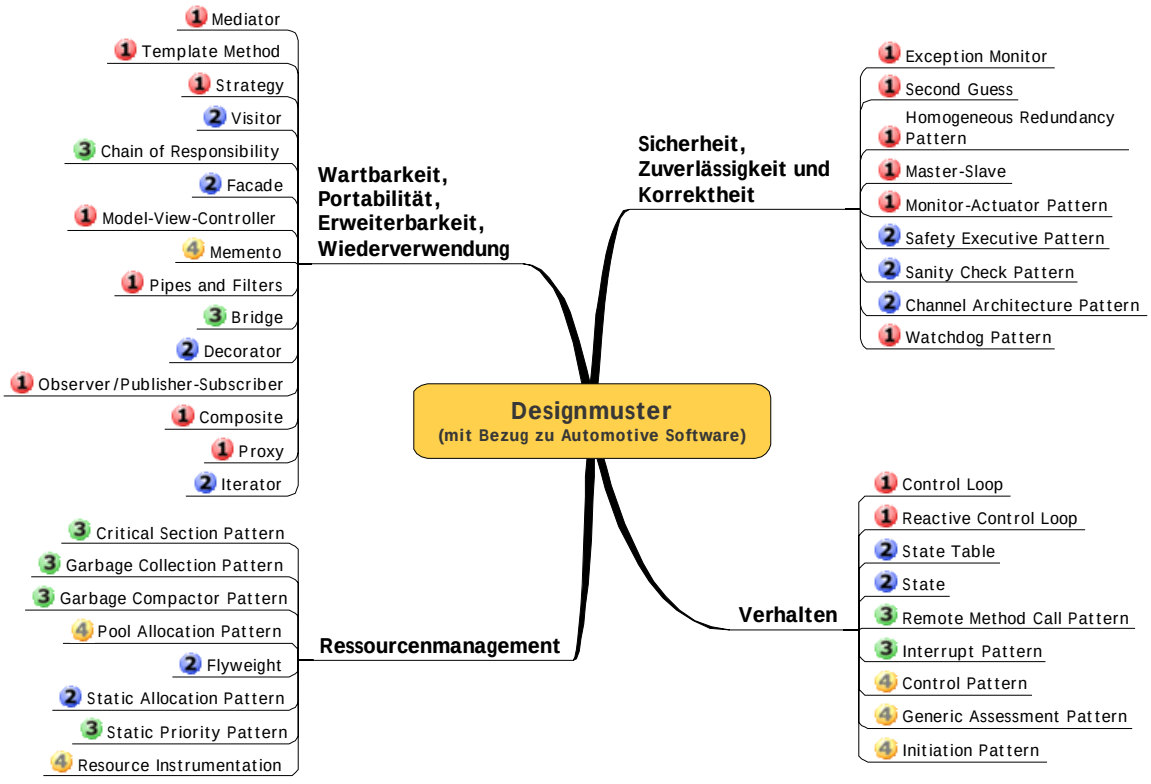


Abbildung 3.27: kategorisierte und bewertete Designmuster

Die Zusammenstellung und Bewertung der Muster ist nicht das Ergebnis einer systematischen Untersuchung. Ein Muster wurde in die Auswahl mit aufgenommen, wenn eine Anwendungsmöglichkeit im Rahmen der Fallstudie oder einem anderen dem Autor bekannten System als

realistisch erschien. Mit der Zusammenstellung soll lediglich gezeigt werden, dass es für die hier betrachtete Domäne ausreichend anwendbare Muster gibt.

Auf Basis der Zusammenstellung kann als Ergebnis festgehalten werden, dass es durchaus zahlreiche Designmuster mit Anwendungsmöglichkeit im Bereich *Automotive Software* gibt. Es gibt allerdings auch viele publizierte Designmuster, für die sich kein Anwendungsfall finden ließ oder die in einigen Projekten geltenden Richtlinien widersprechen. Beispielsweise gibt es Systeme, bei denen eine Speicherallokation nach der Initialisierungsphase aus Sicherheits- und/oder Verfügbarkeitsgründen verboten ist. Bei Designmustern, die dynamisch Objekte anlegen, muss entsprechend darauf geachtet werden, dass dies nur zur Initialisierungsphase geschieht oder die Verwendung ganz ausgeschlossen wird. Insgesamt muss bei dieser Diskussion auch nach den in Kapitel 3.1.1.f erläuterten Subdomänen differenziert werden, da deren Charakteristika mitentscheidend bei der Frage nach der Verwendungsmöglichkeit einzelner Muster sind.

Durch die vielen Publikationen gibt es viele Muster, die sich inhaltlich überlappen oder sogar das gleiche Konzept beschreiben, aber unterschiedlich benannt sind. Es gibt leider keine zentrale Instanz, die alle Muster in einem Namensraum nach einer festen Systematik verwaltet und nach einem einheitlichen Schema beschreibt.

Die Lösung eines Musters wird bei den meisten Veröffentlichungen relativ grob und beispielhaft beschrieben. Oft ist z. B. nicht dezidiert ausgewiesen, welche Teile des Musters durch eigene Elemente ausgetauscht werden können (also Parameter des Musters darstellen) und welche Bedingungen dabei eingehalten werden müssen. Dies lässt sich damit begründen, dass Muster nach Definition nur eine Lösungsidee bzw. ein abstraktes Lösungskonzept beschreiben, das individuell ausgestaltet werden kann (siehe oben). In Bezug auf die klassischen Ziele von Designmustern lässt sich hier daher auch kein Mangel ableiten. Wenn Muster aber darüber hinaus im Kontext von Modelltransformationen zur automatisierten Erstellung von Modellen genutzt werden sollen, reicht diese Art der Lösungsbeschreibung nicht aus. Im weiteren Verlauf dieses Kapitels werden formalisierte Musterbeschreibungen noch adressiert (insb. in Kapitel 3.2.4.e). Bei einer automatisierten Verarbeitung von Mustern zeigen die überwiegend textuellen Muster-Beschreibungen weitere Schwächen. So sind die (nichtfunktionalen) Eigenschaften des Musters (bzw. die „Kräfte“) bei den Standardwerken in Prosaform beschrieben. Eine systematische Auswertung, die z. B. die Frage beantwortet, ob ein Muster die Laufzeit oder den Speicherverbrauch positiv, neutral oder negativ beeinflusst, ist daher schwierig. Vorgreifend sei auf die erarbeitete Tabelle 4.2 verwiesen, in der die Eigenschaften eines Designmusters einheitlich bewertet werden.

3.2.4.c Analysemuster

Bezogen auf die ROPES-Methode, passen Analysemuster zu der Phase *Object Analysis*. In der Beschreibung der ROPES-Analysephase werden die Analysemuster zwar nicht als Artefakte genannt, im Grundlagenteil zu den Mustern findet sich jedoch ein Hinweis, dass Analysemuster Kandidaten für das *Object Analysis Model* sind [Douglass '99]. Zu den Aufgaben bei der Er-

stellung dieses Modells gehört die Modellierung der wesentlichen Konzepte und Objekte, die für das korrekte Verhalten des Systems notwendig sind. Entsprechend unterstützen Analysemuster den Entwickler dabei, Konzepte, die immer wieder beschrieben werden müssen, in geeigneter Weise zu modellieren.

Der bekannteste Beitrag zu Analysemustern ist das Buch *Analysis Patterns: reusable Object Models* von Martin Fowler [Fowler '97]. Er definiert ein Analysemuster als eine „Idee, die in einem praktischen Kontext hilfreich war und dies wahrscheinlich auch in einem anderen sein könnte". Ein entscheidendes Abgrenzungskriterium zu den Designmustern sieht Fowler in der Eigenschaft, dass Analysemuster keine Softwareimplementierung beschreiben.

Die von Fowler beschriebenen Muster beschreiben Sachverhalte auf der Analyseebene aus den Bereichen Buchhaltung, Handel und Beziehungen innerhalb von Organisationen. Abbildung 3.28 zeigt das Muster *Accounting Transaction* aus dem Bereich Buchhaltung.

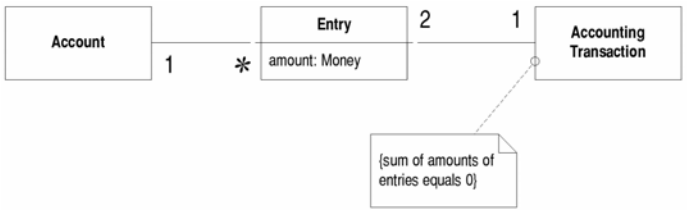


Abbildung 3.28: Analysemuster *Accounting Transaction* nach [Fowler '97]

Das Muster beschreibt einen buchhaltungstechnischen Geschäftsvorgang nach dem Prinzip der doppelten Buchführung. Wie den Kardinalitäten in dem UML-Diagramm zu entnehmen ist, kann ein Objekt der Klasse `Account` (Konto) mehreren Objekten der Klasse `Entry` (Buchung) zugeordnet werden. Aus diesen müssen genau zwei `Entry`-Objekte genau einem Objekt der Klasse `Accounting Transaction` zugeordnet werden. Entsprechend des Prinzips der doppelten Buchführung, muss die Summe beider Buchungseinträge Null ergeben.

Das dargestellte Muster zeigt einen wiederholt anzutreffenden Sachverhalt, der bei der Entwicklung von Buchhaltungssystemen in der Analysephase modelliert werden muss. Es bildet einen komplexen Zusammenhang aus der Realwelt modellhaft nach und verwendet die Sprache des Domänenexperten. Das gezeigte UML-Modell ist rein konzeptuell und enthält keine Designaspekte.

Zu den ersten Arbeiten zu Analysemustern gehört auch [Coad et al. '97]. Der Beitrag von [Fernandez et al. '00] führt *Semantic Analysis Pattern* ein, die auf der Problemseite Use-Cases oder eine Sammlung von Anforderungen enthalten und auf der Lösungsseite Verhaltensbeschreibungen in Form von Zustands- und Sequenzdiagrammen liefern. Im Gegensatz zu Fowler, der seine Muster informell beschreibt, wird hier ein festes Beschreibungsschema verwendet. [Geyer-Schulz et al. '01] benutzen für die Beschreibung ihrer Analysemuster für Groupware-Produkte ebenfalls eine strukturierte Form.

In [Konrad et al. '04b] werden unter dem Titel *Object Analysis Pattern* Analysemuster für die Domäne der eingebetteten Systeme vorgestellt. Die Muster sind ihrem Namen entsprechend für die Phase *Object Analysis* im ROPES-Prozess (siehe Kapitel 3.2.2) vorgesehen und erweitern die Methode an dieser Stelle. Der Beitrag umfasst die Definition eines Beschreibungsschemas und die Katalogisierung von 7 Mustern (detailliert beschrieben in [Konrad et al. '04a]), die in Tabelle 3.9 zusammengefasst sind.

Mustername	**Beschreibung**
Actuator-Sensor (Strukturmuster)	Das Muster *Actuator-Sensor* spezifiziert Basistypen für Sensoren und Aktuatoren in einem eingebetteten System. Zu den Hauptverantwortlichkeiten eines eingebetteten Systems gehört die Interaktion mit der Umgebung über Sensoren und Aktuatoren. Aus diesem Grund beschreibt das Muster wie Beziehungen zwischen Aktuatoren und Sensoren und anderen Komponenten im System erfasst werden können.
Communication Link (Verhaltensmuster)	Durch die steigende Nachfrage nach verteilten eingebetteten Echtzeitsystemen werden Kommunikationsfähigkeiten immer wichtiger. Das Muster *Communication Link* beschreibt, wie auf einer abstrakten Ebene die angebotenen Kommunikationsfähigkeiten eines eingebetteten Systems erfasst werden können, wie z. B. periodische Nachrichten zu anderen Systemen.
Computing Component (Verhaltensmuster)	Eingebettete Systeme müssen häufig unterschiedliche Betriebszustände unterstützten, da ein vollständiger Ausfall des Systems einen hohen Verlust bedeutet. In dem Muster *Computing Component* werden verschiedene Betriebszustände eines eingebetteten Systems definiert, wie z. B. *Fail-Safe*-Zustände, in die das System nach dem Auftreten eines Fehlers übergeht.
Controller Decompose (Strukturmuster)	Das Muster *Controller Decompose* beschreibt, wie ein eingebettetes System in unterschiedliche Komponenten aufgeteilt werden kann. Das Muster ist die Basis für alle weiteren Muster. Es führt eine High-Level-Sicht eines eingebetteten Systems ein, die zur Verfeinerung auf andere Analysemuster verweist.

Tabelle 3.9 (Teil 1): Analysemuster von [Konrad et al. '04b] (übersetzt aus dem Englischen)

Mustername	Beschreibung
Detector-Corrector (Verhaltensmuster)	Eingebettete Systeme unterliegen typischerweise Zeit- und Betriebsbedingungen. Das Ziel des Musters *Detector-Corrector* ist es, einen Mechanismus zu beschreiben, der sicherstellt, dass eine Komponente in Betrieb ist oder spezifische Bedingungen nicht verletzt sind. In dem Muster stellen *Detector*s Fehlererkennungsfähigkeiten und *Corrector*s Fehlerbehebungsfähigkeiten zur Verfügung, wobei die Interaktion zwischen beiden durch einen *Local Fault Handler* gesteuert wird.
Fault Handler (Verhaltensmuster)	Die Behandlung von Fehlern hat bei eingebetteten Systemen eine besondere Bedeutung. In diesem Muster wird ein globaler und mehrere lokale *Fault Handler*s definiert. Der *Global Fault Handler* sammelt Nachrichten der *Local Fault Handler* und agiert als zentraler Koordinator für Systemwiederherstellungs- und Sicherheitsmaßnahmen.
User Interface (Strukturmuster)	Benutzerinteraktion ist ein wichtiger Aspekt eingebetteter Systeme. In dem Muster *User Interface* (Benutzerschnittstelle) interagiert das System mit dem Benutzer über *Control*s (Bedienelement) und *Indicator*s (Anzeigeelement). Im Gegensatz zu Aktuatoren und Sensoren unterliegt diese Interaktion üblicherweise weniger harten Anforderungen (z. B. Zeitanforderungen). Das Muster *User Interface* beschreibt, wie ein Objektmodell für eine Benutzerschnittstelle spezifiziert werden kann, das erweiterbar und wiederverwendbar ist.

Tabelle 3.9 (Teil 2): Analysemuster von [Konrad et al. '04b] (übersetzt aus dem Englischen)

Das Beschreibungsschema orientiert sich an [Gamma et al. '95]. Hinzugefügt wurden die Felder „Constraints“, „Verhalten“ und „Anwendbare Designmuster“. Die Felder „Implementierung“ und „Beispielcode“ wurden dagegen entfernt und das Feld „Verwandte Designmuster” in „Verwandte Analysemuster“ umbenannt.

Das Feld „Constraints“ stellt im Vergleich zu anderen Arbeiten eine Besonderheit dar. Hier können in formaler Weise Bedingungen in temporaler Logik spezifiziert werden, die für jede Instanz des Musters gelten müssen. Hierzu wurde eine eigene Syntax entwickelt, mit der z. B. die Beziehung zwischen Ereignissen und erwarteten Reaktionen beschrieben werden kann. Die Autoren verfügen über eine Werkzeugkette, innerhalb derer UML-Analysemodelle mit einer formalen Semantik erstellt werden können. Wurden in ein Modell die Analysemuster eingebracht, kann mit Hilfe eines Model-Checkers überprüft werden, ob die für das Muster definierten Constraints in diesem Modell erfüllt werden. Im Anhang A.16 ist das Muster *Detector-Corrector* in Form eines Originalauszugs aus dem in [Konrad et al. '04a] vorgestellten Musterkatalog beschrieben.

Die hier besprochenen Analysemuster sind noch relativ jung und nicht breit etabliert. Sie leisten für diese Arbeit vor allem einen konzeptionellen Beitrag. Nach eigener Einschätzung gibt es auf dieser Ebene noch zahlreiche Konzepte, die sich durch Muster beschreiben lassen. Der in Anhang A.1 zusammengestellte Analysemusterkatalog enthält einige selbst erstellte Muster, die im weiteren Verlauf dieser Arbeit verwendet werden.

Die bei den Designmustern angebrachte Kritik gilt eingeschränkt auch für die Analysemuster. Ein Muster wird bei [Konrad et al. '04b] zwar aufgrund der Constraints wesentlich formaler beschrieben, bei der Beschreibung der Lösungsstruktur werden aber auch hier die austauschbaren Elemente und deren Anforderungen nicht dezidiert ausgewiesen. Da bei Analysemustern nichtfunktionale Eigenschaften nicht betrachtet werden, gilt die Forderung nach einer einheitlichen Beschreibung derselbigen nur für Designmuster.

3.2.4.d Weiterführende Themen

Neben der Konzeption des Musterbegriffs, der Beschreibung von Mustern und der Abgrenzung unterschiedlicher Musterarten, gibt es noch zahlreiche weitere Arbeiten, die sich im Umfeld von Mustern bewegen. Im Folgenden wird auf einige dieser Themen eingegangen.

Musterkataloge und –systeme

Unter einem *Musterkatalog* versteht man eine Sammlung mehr oder weniger zusammenhängender Muster, die diese in eine Reihe von Kategorien einteilt und Beziehungen zwischen den Mustern dokumentiert. Der Katalog stellt eine bloße Aufzählung von Muster dar, die mit minimalen Mitteln strukturiert wurde.

Ein *Mustersystem* [Buschmann '98] dagegen ist weitaus zusammenhängender, hierarchisch gegliedert und spricht ein definiertes Problemumfeld an. Es beschreibt, wie übergeordnete Muster mit Hilfe von untergeordneten Mustern realisiert werden können und stellt Abhängigkeiten und Zusammenhänge zwischen den Mustern, bezogen auf das Zielgebiet, umfassend dar. Von der *Pattern Language* unterscheidet es sich darin, dass die Abdeckung des Umfeldes weitaus geringer ist und Mustersysteme somit als Teil einer umfassenderen Mustersprache eingesetzt werden können.

Um den Anwender bei der Auswahl eines passenden Musters zu unterstützen, können Kataloge oder Systeme eine Suchfunktion bzw. Suchmethoden anbieten, die es erlauben, ausgehend von der Fachdomäne und konkreten Problemstellung passende Muster zu finden.

Formalisierung der Lösungsbeschreibung

Viele Autoren (z. B. [Gamma et al. '95] und [Buschmann '98]) verstehen die in den Mustern enthaltenen Diagramme als eine reine Konzeptillustration, die mit dem Modell eines angewendeten Musters in keiner formalisierten Beziehung steht. Manche Autoren (z. B. [Douglass '02]) sehen jedoch die Möglichkeit, die Beziehung zwischen der Strukturbeschreibung eines Musters und dem Modell, in der dieses Muster angewendet wird, formaler zu fassen. Ein häufig anzutreffender Ansatz ist die Beschreibung eines Musters als *parametrisierbare Kollaboration*

(siehe Kapitel 3.2.4.e). Eine Kollaboration besteht in diesem Kontext aus Rollen (modelliert als *Classifier*) die zusammenarbeiten (z. B. durch Assoziationen). Viele dieser Rollen können als parametrisierbar ausgewiesen werden und somit bei der Anwendung des Musters mit konkreten Modellelementen belegt werden (Prozess der *Bindung*). Auf diese Weise wird festgelegt, welche Teile eines Musters statisch (kommen bei jeder Anwendung vor) sind und welche Teile durch eigene Elemente ausgetauscht werden können. Über die bei der Anwendung des Musters mitgepflegten Rollen bleibt der Bezug zu der Musterbeschreibung erhalten.

Anti-Muster

Während Muster eine vorbildliche Lösung einer Problemstellung beschreiben, sollen *Anti-Muster* als negatives Beispiel vor einer Problemlösung, die in einer nachteiligen Situation resultiert, abschrecken. Der Begriff Anti-Muster wurde ursprünglich 1995 von [Koenig '95] vorgeschlagen und ist eventuell etwas missverständlich, da es abgesehen von den Anti-Mustern mit negativer Vorbildfunktion, auf die das Präfix *Anti* sicherlich passt, noch eine weitere Gruppe von Anti-Muster gibt, die zusätzlich beschreiben, wie man ausgehend von einer unvorteilhaften Ausgangssituation den Sprung zu einer rettenden Lösung schaffen kann. Zu Anti-Mustern sind in den vergangenen Jahren zahlreiche Werke entstanden, z. B. [Brown et al. '98].

Qualitative Eigenschaften eines Designmuster

In Kapitel 3.2.4.b wurden einige grundsätzliche Anforderungen an Designmuster genannt, wie z. B. Wiederverwendbarkeit der Lösung oder das wiederholte Auftreten eines Problems. In [Lea et al. '94] werden weitere Eigenschaften beschrieben, anhand dessen sich die Qualität eines Musters beurteilen lässt:

- **Kapselung** und **Abstraktion**: Ein Muster kapselt ein wohldefiniertes Problem in einem festgelegten Anwendungsfeld und stellt eine Abstraktion von empirisch erarbeitetem Wissen und praktischer Erfahrung (*best practice*) dar.
- **Offenheit** und **Variabilität**: Jedes Muster sollte Erweiterungen durch andere Muster erlauben und eine Abhängigkeit seiner Ausprägung bzw. Implementierung von bestimmten Gegebenheiten ermöglichen.
- **Generativität** und **Zusammensetzbarkeit**: Der Vorgang der Erstellung einer Musterausprägung soll an eigene Bedürfnisse anpassbar sein. Muster sollten geeignet kombinierbar sein.
- **Ausgeglichenheit**: Ein Muster soll eine möglichst gute Balance zwischen inneren Erfordernissen und äußeren Sachzwängen darstellen. Es sollen alle Invarianten der Problemstellung erfüllt und Inkonsistenzen in der Lösung vermieden werden.

Einheitliches Beschreibungsschema für Muster

Bisher konnte sich kein einheitliches Schema für die Beschreibung eines Musters herausbilden. Neben der ursprünglichen Notation von Alexander [Alexander '77] ist vor allem das Schema der GoF [Gamma et al. '95] bekannt. Die *Hillside Group* sieht folgende Inhalte einer Musterbeschreibung als wichtig an [Hillside Group]:

- Einen **Bezeichner**, um das Muster eindeutig zu identifizieren und zusätzlich Verweise auf weitere Bezeichnungen (*Also Known As*), unter denen das Muster bekannt ist.
- Eine exakte **Beschreibung der Problemstellung**, die klar die Ziele des Musters innerhalb eines bestimmten Kontextes definiert.
- Eine **Beschreibung des Kontextes** und der Bedingungen, unter denen das Muster angewendet werden kann.
- **Auflistung aller Einflussfaktoren und Invarianten**: Dadurch wird das Lösungsumfeld des Musters genau definiert, wodurch inkonsistente bzw. ungeeignete Lösungen vermieden werden.
- Die eigentliche **Lösungsbeschreibung** für das Problem, die oftmals in einer Anleitung zur Erstellung des gewünschten Ergebnisses besteht und teilweise auch Varianten abhängig von bestimmten Einflussfaktoren beinhaltet.
- **Beispiele**, die exemplarisch Kontext, Problemstellung und Lösung illustrieren und so dem Leser ein besseres Verständnis des Musters ermöglichen.
- Eine **Beschreibung des Ergebnisses** der Musteranwendung, also der positiven als auch negativen Konsequenzen, der Folgen für den Zustand des Systems, der gelösten sowie noch ausstehenden Probleme und der Möglichkeiten des weiteren Vorgehens.
- Eine **Begründung**, die die einzelnen Vorgehensschritte und das Muster als Ganzes erläutert und erklärt, warum das Muster auf die gegebene Problemstellung anwendbar ist und wie es das Problem löst.
- Eine **Auflistung weiterer Muster**, die alternativ oder in Ergänzung zu dem gegebenen Muster eingesetzt werden können.

Muster als Komponenten

Mit dem Ansatz *Pattern oriented Analysis and Design* (POAD) präsentiert [Yacoub et al. '04] eine Methode zur Erstellung von Designmodellen durch die Komposition von Mustern. Muster werden als Komponenten mit Interfaces betrachtet, für die es eine Außen- (die Komponente und ihre Interfaces) und Innensicht (Struktur des Musters) gibt. Es wird ein Entwicklungsprozess beschrieben, bei dem ausgehend von Use-Cases zunächst lose Kompositionen von Mustern auf Basis der Außenansicht zusammengestellt wird. Diese im ersten Schritt nur durch Abhängigkeiten verbundenen Komponenten werden im nächsten Schritt durch die Verbindung der Interfaces stärker verwoben. Im nächsten Schritt erfolgt die Auflösung der Komponenten, in dem sie durch die Struktur des Musters (Innenansicht der Komponente) ersetzt werden. Ergebnis ist ein Klassenmodell mit verwobenen Musterinstanzen. Im letzten Schritt wird dieses Klassenmodell durch die Entfernung redundanter Strukturen und der Anwendung weiterer Optimierungstechniken verfeinert und führt schließlich zum letztendlichen Designmodell.

3.2.4.e Muster in der UML

Im letzten Unterkapitel wurde auf Arbeiten hingewiesen, die sich mit der formalisierten Beschreibung der Struktur- und Verhaltensmodelle eines Musters befassen. Eine Zielsetzung hierbei ist, den Zusammenhang zwischen dem Modell der Vorlage und dem Anwendungsmodell, in dem das Muster verwendet ist, so zu beschreiben, dass auch im Nachhinein erkennbar ist, an welcher Stelle im Modell ein Muster eingesetzt ist und welche Elemente die in der Vorlage definierten Rollen ausüben. Der UML2-Standard [OMG '05c] liefert hierzu Lösungen, die im Folgenden erläutert werden.

Der Mustergedanke wird innerhalb des UML-Standards durch zwei unterschiedliche Konstrukte adressiert: *Collaboration* (Kollaboration) und *Template* (Vorlage). Die Beschreibungen sind in Kapitel 9.3.3, bzw. 17.5 in [OMG '05c] zu finden.

Eine *Collaboration* ist in der UML2 ein Subtyp des neu eingeführten *Structured Classifier*. Sie beschreibt in Form von Rollen, wie Elemente für die Erfüllung einer bestimmten Aufgabe zusammenarbeiten. In dem Modell eines konkreten Systems kann mit dem Element *CollaborationUse* markiert werden, wo diese Struktur verwendet wird und mit Hilfe von *RoleBindings* gezeigt werden, welche Elemente die definierten Rollen ausüben. Laut UML-Standard dienen *Collaboration* und *CollaborationUse* hauptsächlich Verständnis- und Dokumentationszwecken. Ein Element, das eine in einer *Collaboration* definierte Rolle ausübt, muss manuell angelegt werden und dann mit einem *RoleBinding* versehen werden. Eine automatische Instanziierung, die die Struktur der *Collaboration* automatisch übernimmt, wird im UML-Standard nicht erwähnt. Ein weiterer Nachteil ist, dass nur sehr wenige UML-Typen in einer *Collaboration* als Rolle definiert werden können. Außerdem gibt es keine Möglichkeit, Teile einer *Collaboration* als statisch auszuweisen. Alle Elemente einer *Collaboration* werden als Rolle interpretiert. Es ist somit nicht möglich festzulegen, dass z. B. eine Klasse genau in dieser Form (gleicher Name und gleiche *Features*) in jeder Anwendung der *Collaboration* vorkommen muss.

Die UML-Templates realisieren dagegen ein weiter formalisiertes Vorlagenkonzept, dessen Grundprinzip von den schon länger bekannten C++-Templates übernommen wurde. Grundlegende Idee dieses Konzepts ist es, den Typ eines Elements zur Modellierungszeit offen zu lassen und als sog. *formalen Template-Parameter* (engl. *formal template parameter*) auszuweisen. Zu einem späteren Zeitpunkt kann dieser Parameter an einen konkreten Typ *gebunden* werden. Dies geschieht im Rahmen der *Template-Instanziierung*, bei der allen formalen Template-Parametern *tatsächliche Template-Parameter* (engl. *actual template parameter*) zugewiesen werden und die im Template definierte Struktur ins Modell übernommen wird.

Besondere Eigenschaften

Die UML-Templates unterstützen über die einfache Template-Definition und Template-Instanziierung hinaus einige weitere Merkmale. So ist es möglich, Einschränkung bzgl. der tatsächlichen Parameter zu formulieren (Beispiel: „Der tatsächliche Parameter muss ein Subtyp von Typ X sein“). Um Namen für Modellelemente automatisch aus anderen Modellteilen oder

Parametern abzuleiten, stehen String-Ausdrücke zur Verfügung, die einfache String-Operationen wie z. B. Konkatenation unterstützen.

Auch für die systematische Verfeinerung von Templates bietet der UML-Standard Ansätze. So sind einerseits prinzipiell Vererbungen zwischen Templates möglich. Andererseits gibt es auch grundsätzlich verschachtelte Templates (Templates, die andere Templates instanziieren). Leider ist der UML-Standard bei diesen letzten beiden Punkten in Bezug auf die genaue Semantik unpräzise.

Das Template-Metamodell im Detail

Die relevanten Teile des Metamodells befinden sich im Paket `Templates` des UML-Standards [OMG '05c]. Abbildung 3.29 zeigt einige Paket-Beziehungen, die für das Verständnis der Inhalte von `Templates` wichtig sind.

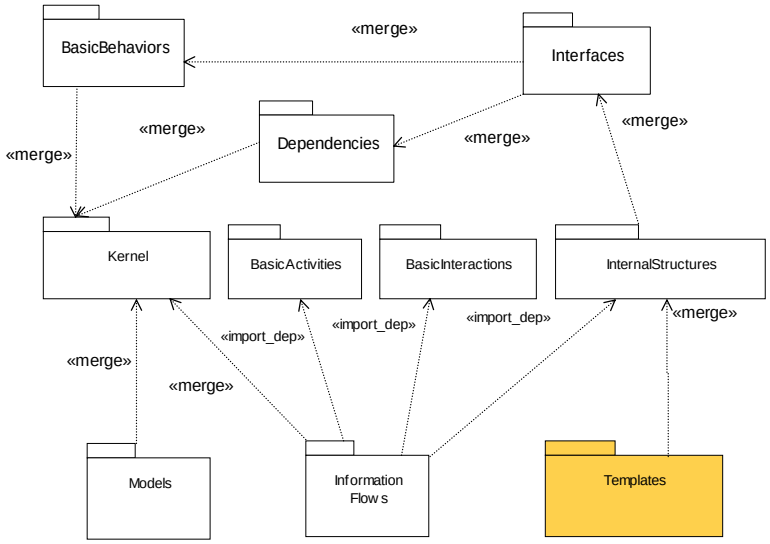


Abbildung 3.29: Paketbeziehungen im Umfeld von `Templates`

Zunächst kann grob zwischen den Metaelementen differenziert werden, die eine Template-Definition spezifizieren und solchen, die die Bindung eines Templates beschreiben. Abbildung 3.30 zeigt zunächst den Ausschnitt für die Template-Definition.

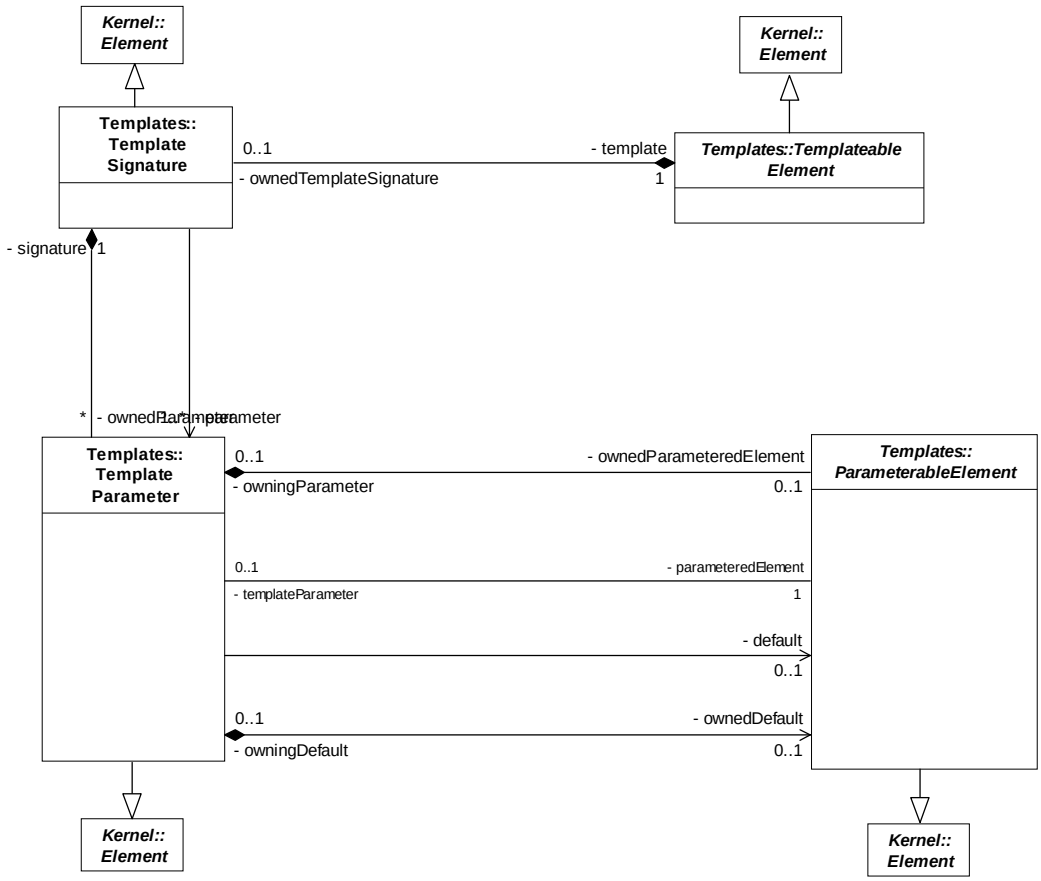


Abbildung 3.30: Metamodell für die Template-Definition

Das Wurzelelement des Metamodells für die Template-Definition ist die abstrakte Klasse `TemplateableElement`. Alle Spezialisierungen dieser Metaklasse können als Template definiert werden. Wie Abbildung 3.31 zu entnehmen ist, handelt es sich dabei neben `Classifier` (damit auch jede Spezialisierung wie z. B. Komponenten, Datentypen, Schnittstellen oder Signale) unter anderem auch um Operationen oder Attribute (bzw. Properties).

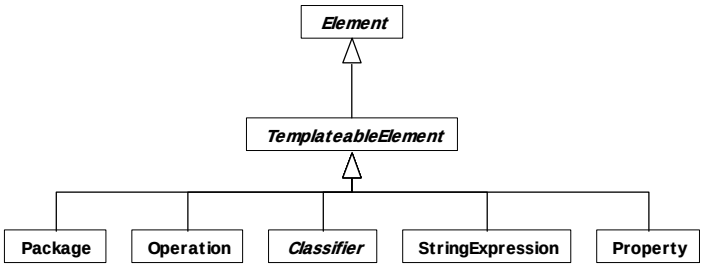


Abbildung 3.31: Elemente die als Template deklariert werden können

Wie Abbildung 3.30 weiter zu entnehmen ist, enthält ein Template eine Signatur (`TemplateSignature`), die wiederum Parameter (`TemplateParameter`) enthält. Ein Parameter verweist auf eine abstrakte Klasse `ParameterableElement`. Über diese Beziehung wird unter

anderem der Typ des Parameters festgelegt. Um welchen Typ es sich dabei konkret handeln kann, ist in Abbildung 3.32 zu sehen.

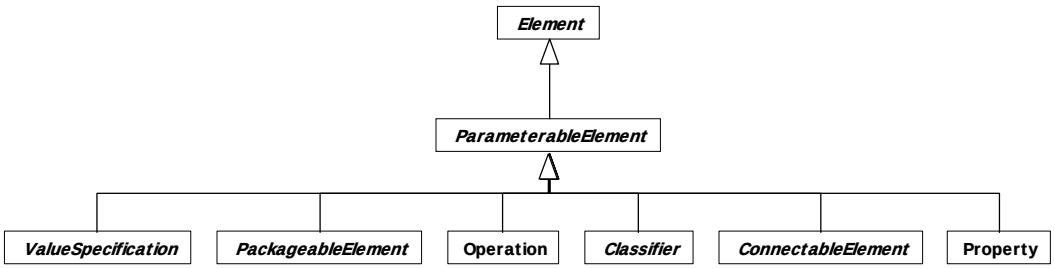


Abbildung 3.32: Elemente die als Parameter deklariert werden dürfen

Abbildung 3.33 zeigt die Elemente, die für eine Template-Bindung benötigt werden. Wurzelelement für eine Bindung ist die Klasse `TemplateBinding`. Hierbei handelt es sich um eine gerichtete Beziehung (erbt von `DirectedRelationship`) zwischen einer Template-Instanz (`TemplateableElement` mit Aggregation `boundElement`) und dem Template-Typ (`TemplateSignature`). Über die enthaltenen Elemente `TemplateParameterSubstituion` verfügt eine Bindung ferner über Informationen bzgl. der Zuordnung von tatsächlichen zu formalen Template-Parametern. Diese Zuordnung wird dadurch realisiert, dass `TemplateParameterSubstitution` jeweils eine gerichtete Assoziation zu einem formalen Template-Parameter (Rolle `formal`) und zu mindestens einem konkreten Modellelement, das von `ParamterableElement` erbt, besitzt (Rolle `actual`). Die Bedingung, dass der tatsächliche Template-Parameter und der formale Template-Parameter typkompatibel sein müssen, ist im Standard durch ein entsprechendes OCL-Constraint spezifiziert.

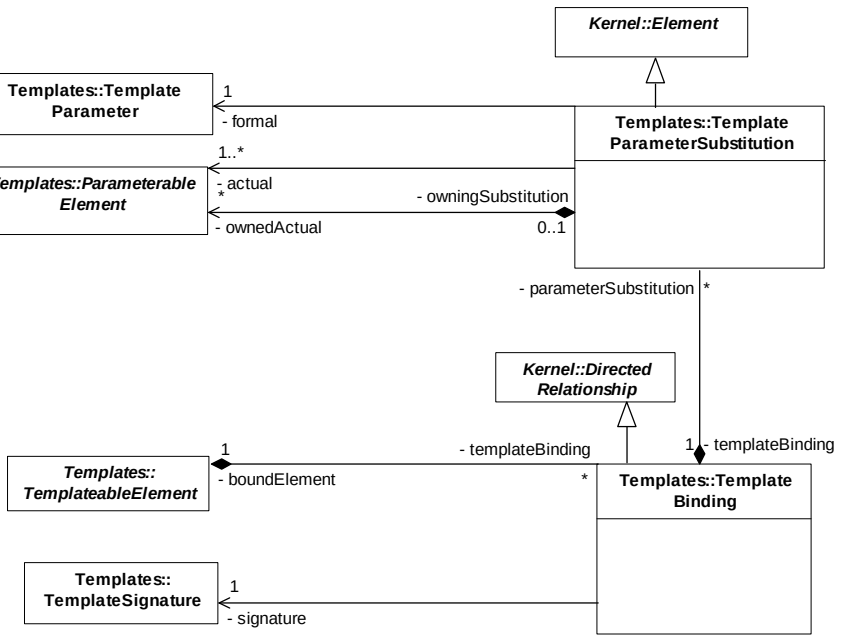


Abbildung 3.33: Template-Bindung

Die Subtypen von TemplateableElement sind in der Template-Spezifikation ausführlicher beschrieben. Hier finden sich für jede Template-Art spezifische Angaben zur Instanziierung des Templates. Schwerpunktmäßig wird dabei beschrieben, wie Template-Elemente mit bereits existierenden Elmenten *verschmolzen* (engl. *merging*) werden. Der UML-Standard [OMG '05c] definiert den Begriff der Verschmlezung z. B. bei der Erläuterung der mit <<merge>> stereotypisierte gerichtete Beziehung zwischen Paketen. Durch solch eine Beziehung wird angezeigt, dass die Inhalte zweier Pakete „kombiniert" werden. Dies ist laut UML-Standard in dem Sinne mit einer Generalisierung vergleichbar, dass einem Element konzeptionell die Charakterisitk eines anderen Elements hinzugefügt wird. Das Ergebnis ist ein neues Element, dass die Charakteristika dieser beiden Elemente kombiniert. Bei einer Template-Instanziierung werden entsprechend die Template-Elemente mit den Elementen des Zielmodells verschmolzen. Nach der Instanziierung enthält das Zielmodell eine Kombination der zuvor existierten Elemente und der Template-Elemente.

Modellierung von Mustern

Eine Möglichkeit, die Klassenstruktur eines Musters zu modellieren und die parametrisierbaren Anteile auszuweisen, bieten die Paket-Templates. Abbildung 3.34 zeigt die Definition eines Musters Messung in dieser Form.

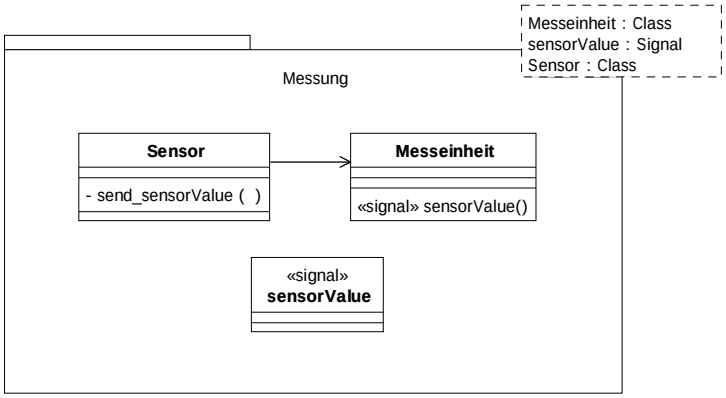


Abbildung 3.34: Das Template Messung

Abbildung 3.35 zeigt das Paket Wetterstation, bevor es mit dem Template Messung gebunden wird. Es enthält zwei leere Klassen und ein Signal, die nicht über Assoziationen miteinander verbunden sind.

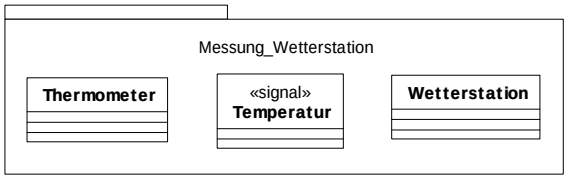


Abbildung 3.35: Das Paket Messung_Wetterstation vor der Template-Instanziierung

In Abbildung 3.36 ist das Paket nach der Bindung und Instanziierung zu sehen. Die beiden Klassen Thermometer und Wetterstation fungieren als tatsächliche Parameter für die formalen Parameter Sensor und Messeinheit, das Signal Temperatur ist sensorWert zugewiesen. Die im Template definierte Struktur wurde mit dem Zielpaket verschmolzen. Entsprechend wird die Assoziation zwischen Thermometer und Wetterstation, die Operation send_sensorValue bei Thermometer und der Signalempfang von Temperatur bei Wetterstation angelegt.

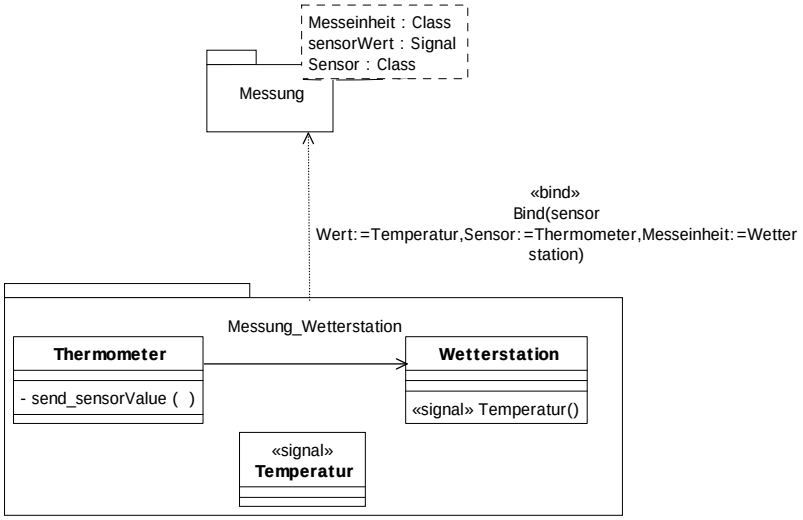


Abbildung 3.36: Instanz des Templates Messung

Der Verschmelzungsprozess ist jedoch komplizierter, als das gezeigte Beispiel es vermuten lässt. Wenn die an Parameter gebundenen Elemente z. B. bereits über Attribute, Operationen oder Assoziationen verfügen, können sich Konflikte mit der Template-Struktur ergeben. Bei der Instanziierung aus Abbildung 3.36 stellen sich z. B. folgende Fragen:

- Angenommen zwischen Thermometer und Wetterstation existiert bereits vor der Instanziierung des Musters eine gerichtete Assoziation – wird durch die Musterinstanziierung eine weitere Assoziation angelegt oder wird die existierende „wiederverwendet“?
- Was passiert, wenn die Klasse Wetterstation bereits vor der Musterinstanziierung ein Signal mit dem Namen Temperatur empfangen kann, dieses aber einen anderen Typ hat, als der tatsächliche Parameter aus der Musterbindung. Wird der existierende Signalempfang überschrieben?
- Was passiert, wenn die Klasse Thermometer bereits vor der Musterinstanziierung über eine Operation mit dem Namen send_sensorValue verfügt, diese aber eine andere Signatur hat? Wird diese gelöscht und mit der Signatur aus der Musterdeklaration ersetzt?

Für die Auflösung dieser Konflikte sind für die unterschiedlichen Template-Arten Verschmelzungsregeln definiert. Bei Paket-Templates gelten hierbei dieselben Verschmelzungsregeln wie bei einer Paket-Verschmelzung (Abhängigkeitsbeziehung zwischen Paketen mit Stereotyp <<merge>>).

Kritik

Im Rahmen der Modellierung der Fallstudie fiel auf, dass sich ein in der Domäne häufig auftretender Sachverhalt mit UML-Templates nicht adäquat abbilden lässt. Bezogen auf das Beispiel wird dies an der Tatsache deutlich, dass eine Messeinheit häufig über mehrere Sensoren verfügt. In diesem Fall bietet es sich an, den Parametern Sensor und sensorValue entsprechend mehrere Objekte zuzuweisen. Dies ist laut UML-Metamodell erlaubt (die Eigenschaft actual von TemplateParameterSubstitution hat die Multiplizität „*“), wird aber im Standard nicht näher beschrieben. Es ist auch nicht möglich, einem Parameter bei der Definition eines Templates eine Multiplizität zuzuweisen (TemplateParameter ist kein Subtyp von MultiplicityElement), womit eine Festlegung getroffen werden könnte, wie viele Elemente bei der Instanziierung zugewiesen werden dürfen bzw. müssen.

Abbildung 3.37 zeigt eine Instanz des Musters, in der einzelnen Parametern mehrere Elemente zugewiesen wurden. Dem Parameter Sensor sind die Klassen Thermometer und Hygrometer und dem Parameter sensorWert die Signale Temperatur und Luftfeuchtigkeit zugewiesen.

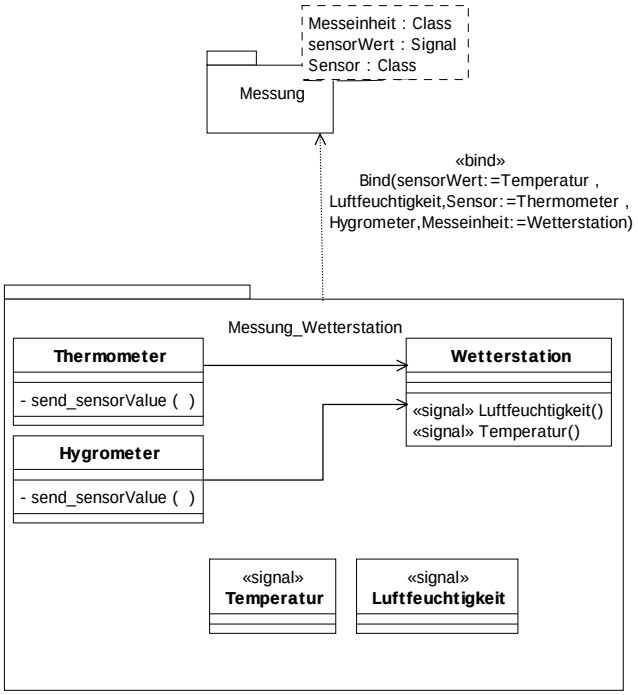


Abbildung 3.37: Instanz des Musters Messung mit mehreren Sensoren

Die Zuweisung mehrerer Elemente an einen Parameter führt jedoch zu einem Problem, wenn es Abhängigkeiten zu anderen Parametern gibt. Im Beispiel gibt es eine solche Abhängigkeit zwischen Sensor und sensorValue: Nach der Instanziierung ist nicht mehr klar, welches Signal von welchem Sensor gesendet wird. Der Zusammenhang von Thermometer und Temperatur sowie Hygrometer und Luftfeuchtigkeit ist nicht modelliert. Ein entsprechender Gruppierungsmechanismus oder ein Konstrukt, mit dem die Abhängigkeit zwischen den Parametern Sensor und sensorWert bei der Modellierung des Templates ausgedrückt werden kann, steht in der

UML nicht zur Verfügung. In dem in Kapitel 4.2.1 vorgestellten selbst modifizierten Metamodell wird für diese Lücke ein entsprechendes Konstrukt vorgeschlagen.

Im Rahmen der Recherchen konnte nicht geklärt werden was passiert, wenn bei der Bindung eines Paket-Templates die tatsächlichen Parameter aus einem anderen Paket kommen. Im Beispiel aus Abbildung 3.37 könnte z. B. die Klassen `Thermometer` und `Hygrometer` aus einem Paket `Sensorik` stammen. Bei der Behandlung dieses sehr realistischen Falls wäre es aus Sicht der Musterinstanziierung wünschenswert, wenn zwischen den Paketen `Messung_Wetterstation` und `Sensorik` eine Import-Beziehung angelegt wird und die Klassen `Thermometer` und `Hygrometer` innerhalb von `Messung_Wetterstation` verwendet werden. Vermutlich dürfen die tatsächlichen Parameter aber nur aus dem gebundenen Paket kommen, was dann ein Nachteil der Paket-Templates wäre. Eine Überprüfung anhand von Werkzeugumsetzungen war nicht möglich, da im Rahmen der Untersuchung kein UML2-Werkzeug gefunden werden konnte, das bei der Instanziierung von Templates eine Modellverschmelzung nach den Regeln der UML durchführt.

Ein weiterer Mangel kann in den Paket-Verschmelzungsregeln gesehen werden, die bei der Instanziierung von Paket-Templates zum Einsatz kommen. Sie sind für die Modellierung von Metamodellen optimiert und passen bei einigen Details nicht zu den Anforderungen, die man bei der Instanziierung von Mustern haben kann. Würde z. B. die Klassen `Thermometer` und `Wetterstation` bereits vor der Instanziierung (Abbildung 3.35) über eine gerichtete Assoziation mit Namen `Temperaturübertragung` verfügen, so würde das Paket nach der Instanziierung (Abbildung 3.36) entsprechend der Verschmelzungsregeln von Paketen über eine zusätzliche gerichtete Assoziation ohne Namen verfügen, weil diese so im Template definiert wurde und die Verschmelzung nur auf der Basis von Namensgleichheit durchgeführt wird. Im Kontext eines Musters könnte aber der Name der Assoziation bewusst offen gelassen werden, da es aus Sicht des Musters nur wichtig ist, dass irgendeine Assoziation zwischen den Klassen besteht und bereits existierende Assoziationen demnach wiederverwendet werden sollen.

Ein weiteres Beispiel sind Operationen. Nach den Regeln für Pakete werden diese nur verschmolzen, wenn sie namensgleich sind und über dieselbe Parameterliste verfügen. Unterscheiden sich die Parameterlisten, sind im Verschmelzungsergebnis zwei namensgleiche Operationen mit unterschiedlichen Parameterlisten vorhanden. Dieses Verhalten kann gewollt sein, wenn ein Überladen der Operation bewirkt werden soll. Im Fall eines Musters kann es aber auch sinnvoll sein, im Template eine Operation mit Parameter anzulegen und diese mit einer namensgleichen Operation im Modell zu verschmelzen, die über keine Parameter verfügt. Dies ist z. B. der Fall, wenn im Modell eine „leere“ Operation als Platzhalter angelegt wurde, in der Erwartung, dass sie durch die Anwendung des Musters in Bezug auf ihre Parameter verfeinert wird. Eine im Modell mit Parametern versehene Operation mit einer parameterlosen Operation aus dem Template zu verschmelzen, ist ebenso denkbar. Ein Beispiel wäre ein parameterloser Konstruktor im Template, mit dem deutlich gemacht wird, dass mindestens ein Konstruktor im Verschmelzungsergebnis vorhanden sein muss, dieser aber über beliebige Parameter verfügen kann. Sowohl das gezielte Überladen als auch das Verschmelzen von Operationen kann bei einer Musterinstanziierung Sinn machen. Daher wäre es wünschenswert, wenn für jedes Element im

Template das Verschmelzungsverhalten feiner spezifiziert werden könnte. Das im folgenden Unterkapitel vorgestellte Werkzeug verfügt über entsprechende Einstellungsmöglichkeiten, wohingegen die UML immer dieselben Standardregeln vorsieht.

3.2.4.f Werkzeugunterstützung für Muster

In [Fischer '05] wurden als Vorarbeit zu der Entwicklung von POTAD die Musterunterstützung in den Werkzeugen *Rational XDE* [IBM b], *IBM Software Architect* [IBM a] und *Compuware OptimalJ* [Compuware] vergleichend evaluiert. Die Musterunterstützung dieser Werkzeuge unterscheidet sich von anderen UML-Werkzeugen dahingehend, dass zwischen Mustertypen und -instanzen differenziert wird und die von der UML beschriebene Parameter-Bindung vom Grundsatz her unterstützt wird. Bezogen auf die Umsetzung der Templates aus dem UML-Metamodell (Kapitel 3.2.4.e) erwies sich das Produkt XDE als das mächtigste Werkzeug. Als stellvertretendes Beispiel dafür, was an Werkzeugunterstützung für Muster zurzeit verfügbar ist, wird die Musterunterstützung von XDE im Folgenden ausführlicher erläutert.

XDE [IBM b] integriert eine klassische Entwicklungsumgebung (C++ und Java) und ein UML-Modellierungswerkzeug auf der Basis von Eclipse. Eine Besonderheit dieses Werkzeugs ist ein verhältnismäßig mächtiger Mustermechanismus, der unter anderem das Erstellen eigener UML-Muster erlaubt, die um Code-Templates ergänzt werden können. Bibliotheken der bekannten Gamma-Muster [Gamma et al. '95] für Java und C++ werden mitgeliefert. Das betrachtete *Rational XDE* lag in Version 2003.06.12 vor.

Definition von Mustern

Im Folgenden soll der Mustermechanismus anhand des Beispiel-Musters *ClosedLoopControl* (siehe Anhang A.1) erläutert werden. Abbildung 3.38 zeigt die Definition dieses Musters im Modellbaum. Wurzelelement ist ein sog. *Asset*. Dabei handelt es sich um einen XDE-eigenen Mechanismus zur Bildung wiederverwendbarer und austauschbarer Modelleinheiten. Das eigentliche Muster wird in der Notation einer *Collaboration* (Ellipse) mit Parametern (rote Punkte) angelegt. Für den Parameter `ClosedLoopController` wurde außerdem ein Defaultwert angegeben (Symbol „<->“).

Das UML-Element unterhalb des Parameters (im Beispiel Klasse oder Signal) spezifiziert den Typ des Parameters. Die Kind-Elemente dieser Elemente (im Fall von `ClosedLoopController` z. B. der Signal-Empfang `<<signal>> FeedbackValue()`) werden bei einer Muster-Anwendung von dem tatsächlichen Parameter entsprechend der Verschmelzungsregeln (siehe folgendes Unterkapitel) übernommen. Die hier ebenfalls zu sehende Abhängigkeit (gestrichelter Pfeil) zeigt ein Beispiel für die integrierte Scriptsprache. Mit der Anweisung `<%=ControlOutputValue%>` wird der Name des Elements als String zurückgegeben, das dem Parameter `ControlOutputValue` zugewiesen wurde.

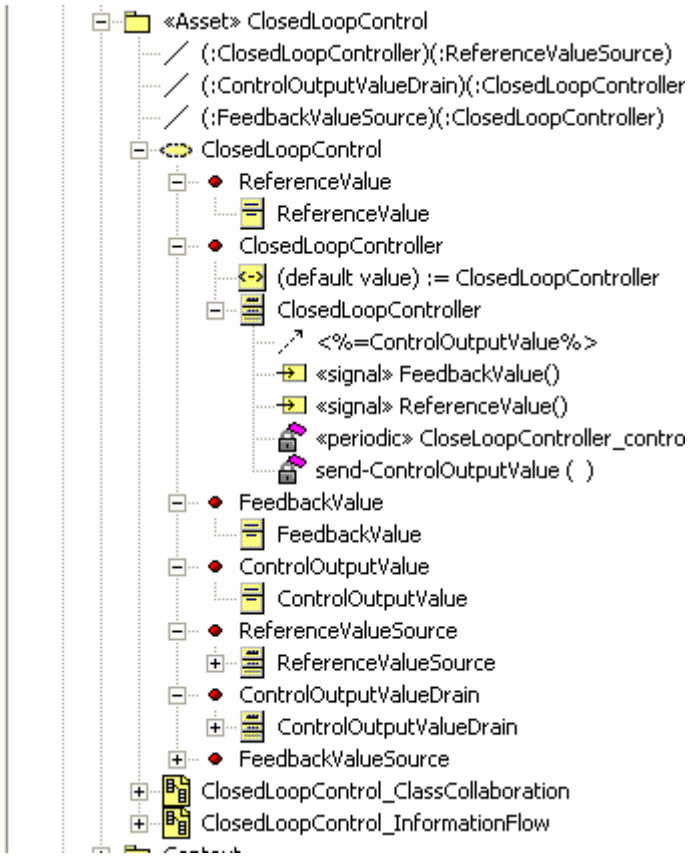


Abbildung 3.38: Musterdefinition im Modellbaum

Teil der Musterdefinition sind im Beispiel zwei Diagramme zu dem Muster, entsprechend der Vereinbarung aus Kapitel 3.2.1.c. Abbildung 3.39 zeigt das Diagramm `ClosedLoopControL_InformationFlow`, welches das Muster in der *Information Flow*-Darstellung zeigt.

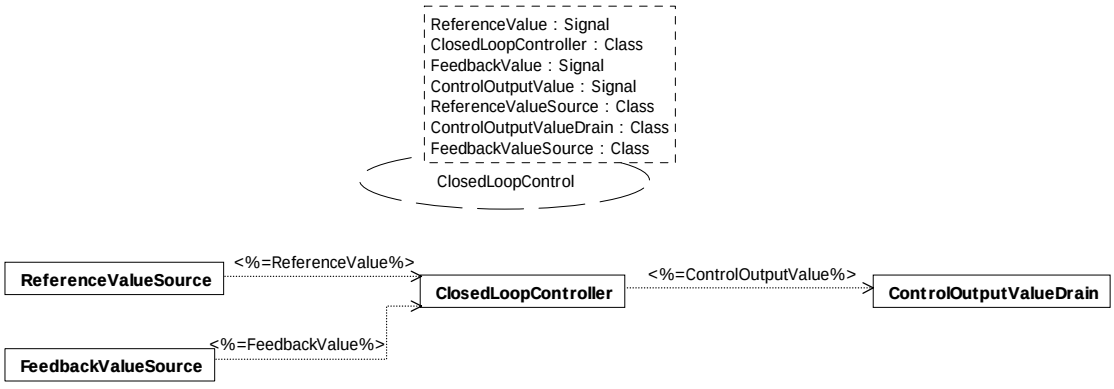


Abbildung 3.39: Darstellung des Musters im Diagramm

Über diese grundlegende Musterdefinition hinaus können in dem *Pattern Explorer* noch detailliertere Einstellungen vorgenommen werden (siehe Abbildung 3.40). Dazu gehört ein Feld für jedes Element des Musters, in dem das Verhalten beim Verschmelzen eingestellt werden

kann (siehe folgendes Unterkapitel). Des Weiteren können im *Pattern Explorer* spezielle Dialoge für die Anwendung des Musters, Constraints für die tatsächlichen Musterparameter, zusätzlichen Script-Code für die Anwendung des Musters und vieles mehr spezifiziert werden. Zu den definierbaren Constraints gehört auch die Multiplizität des Parameters, womit die bei der Erläuterung der UML bemängelte Lücke an dieser Stelle geschlossen wird.

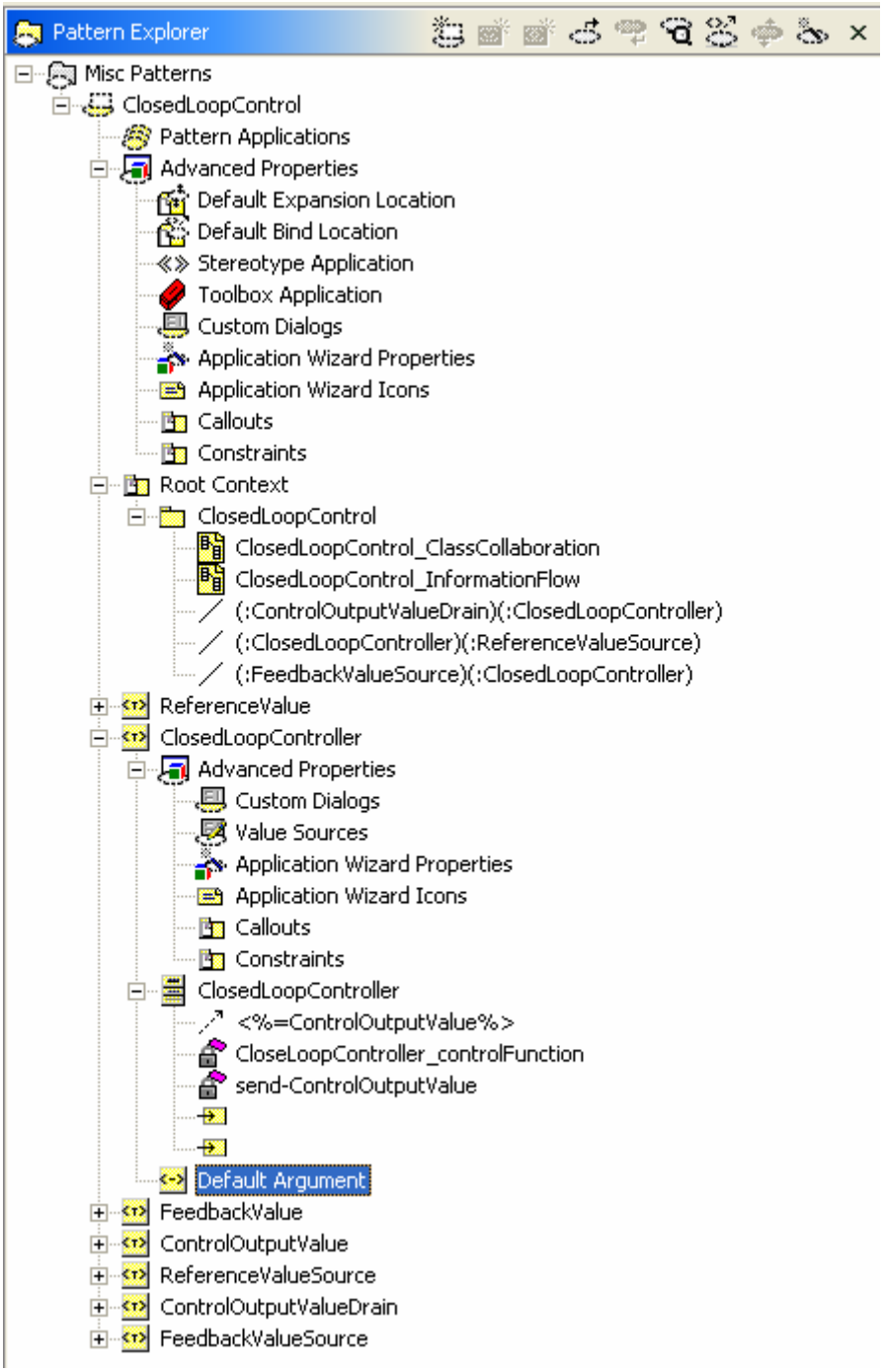


Abbildung 3.40: Erweiterte Einstellungen für ein Muster im Pattern-Explorer

Der *Root Context* nimmt alle nicht-parametrisierten Elemente des Musters auf. Er ist im Beispiel vom Typ Paket und kann generell als ein spezieller Parameter aufgefasst werden, dem bei der Instanziierung ein tatsächlicher Parameter zugewiesen werden kann.

Verhaltensmodelle in Form von Zustands- und Sequenzdiagrammen können in XDE ebenfalls Teil einer Musterdefinition sein und parametrisierbare Elemente ausweisen. Bei der Instanziierung werden in diesen Modellen ebenso wie bei Klassenmodellen die formalen Parameter durch tatsächliche Parameter ersetzt. Da die Modellierung von dynamischen Aspekten aber nicht zu dem Schwerpunkt dieser Arbeit gehört (siehe Kapitel 1.3), wird dieser Teil nicht näher erläutert.

Instanziierung von Mustern

Abbildung 3.41 zeigt den Dialog, mit dem eine Bindung des Musters angelegt wird. Zu sehen sind unter anderem alle formalen Parameter, denen nun tatsächliche Parameter zugewiesen werden müssen. Darüber hinaus müssen der *Root Context* (*Expansion Location*) und der Zielort für die Abspeicherung der Bindung (*Bind Location*) angegeben werden.

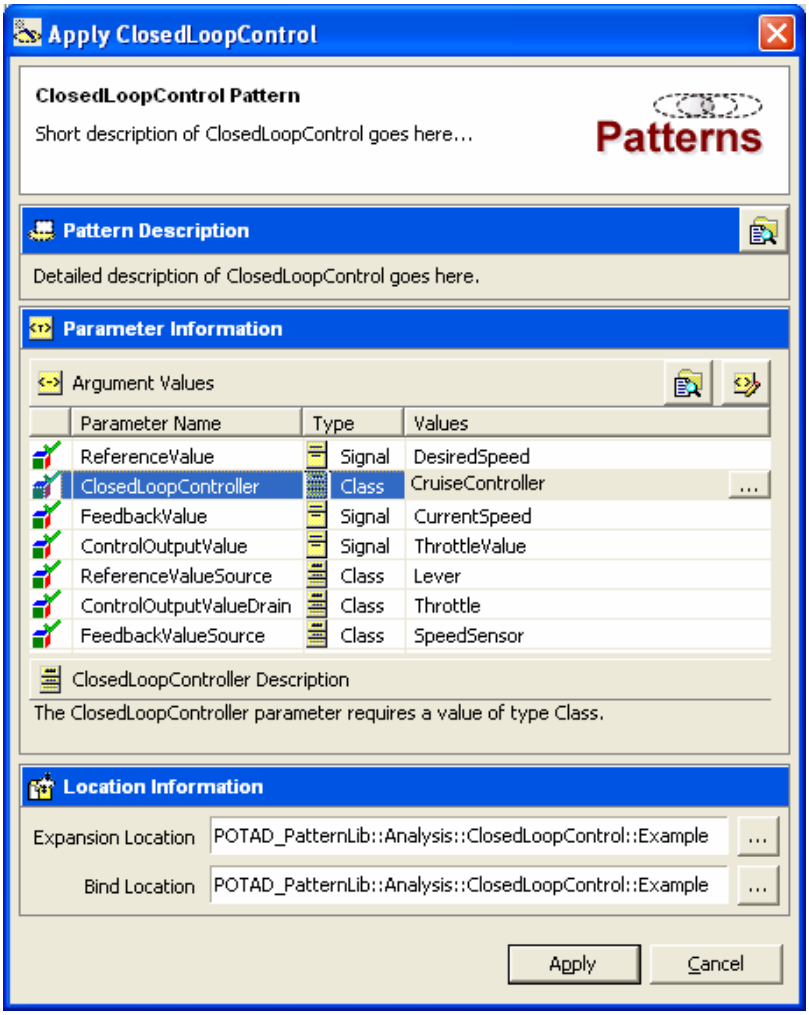


Abbildung 3.41: Instanziierung eines Musters

Für die Zuweisung der Musterparameter bietet XDE zwei Möglichkeiten. Zum einen können bereits existierende Elemente aus dem Modellbaum ausgewählt werden (linker Dialog in Abbildung 3.42). Sollten für die Parameterbelegung des Musters neue Elemente angelegt werden, so kann dies zum anderen direkt in dem Dialog geschehen. Dazu muss dem Element ein Name gegeben werden. Hierbei kann der Benutzer alternativ direkt einen Namen angeben oder den Namen per String-Ausdruck in der XDE-eigenen Script-Sprache generieren lassen (rechter Dialog in Abbildung 3.42). Letzteres ist vor allem hilfreich, wenn das Muster verschachtelt in einem weiteren Muster verwendet wird oder wenn das Muster wiederholt angewendet wird.

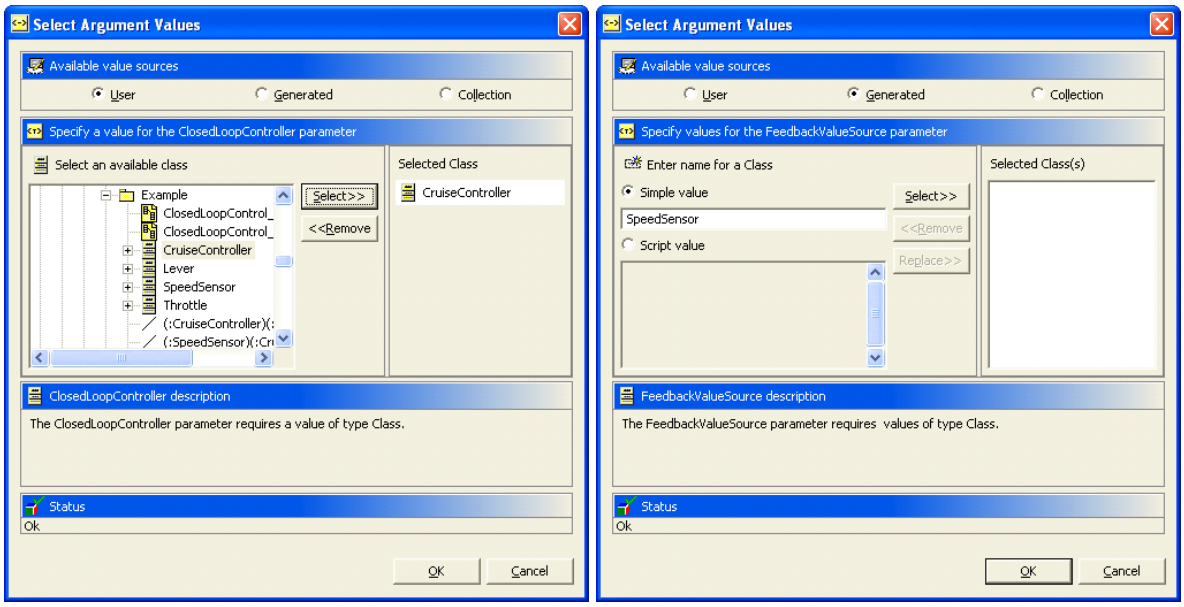


Abbildung 3.42: Alternativen für die Zuweisung der Musterparameter

Mit der Bestätigung des Dialogs werden die gemachten Zuordnungen als Musterbindung im Modell abgelegt und der Vorgang der *Expansion* gestartet. Darunter wird das Ersetzen der formalen Parameter durch die tatsächlichen Parameter in der Musterstruktur und die anschließende Verschmelzung mit dem Zielmodell verstanden. Abbildung 3.43 zeigt das angewendete Muster in der Diagramm-Darstellung. Die Musterinstanz ist als Ellipse dargestellt (`ClosedLoopControl_CruiseControl`). Diese ist über eine spezielle Abhängigkeitsbeziehung (Stereotyp `<<bind>>`) mit dem Mustertyp verbunden (`ClosedLoopControl`). Oberhalb des Pfeils ist die Parameterzuordnung zu sehen. Unterhalb der Ellipsen befindet sich die *expandierte* Struktur des Musters, in der die formalen Parameter durch die tatsächlichen Parameter ersetzt sind.

Über das Kontextmenü der Musterinstanz `ClosedLoopControl_CruiseControl` kann das Muster jederzeit neu expandiert werden. Die Dialoge aus Abbildung 3.42 und Abbildung 3.43 sind in diesem Fall mit der alten Bindung vorbelegt und müssen nicht noch mal neu ausgefüllt werden. Dies kann z. B. notwendig werden, wenn Elemente aus der Musterinstanz gelöscht oder geändert wurden, oder sich die Musterdefinition selbst geändert hat. Über den Verschmelzungsmechanismus (siehe nächstes Unterkapitel) kann dabei sichergestellt werden, dass durch die

wiederholte Expandierung Elemente des Musters nicht doppelt im Zielmodell angelegt werden. Beispielsweise führt eine Neuexpansion direkt im Anschluss auf die erste Instanziierung zu keinen Änderungen im Modell. In dem einer Musterbindung über einen längeren Zeitraum hin Schritt für Schritt immer neue tatsächliche Parameter zugewiesen werden, kann die Musterinstanz auf diese Weise sukzessive „ausgebaut“ werden.

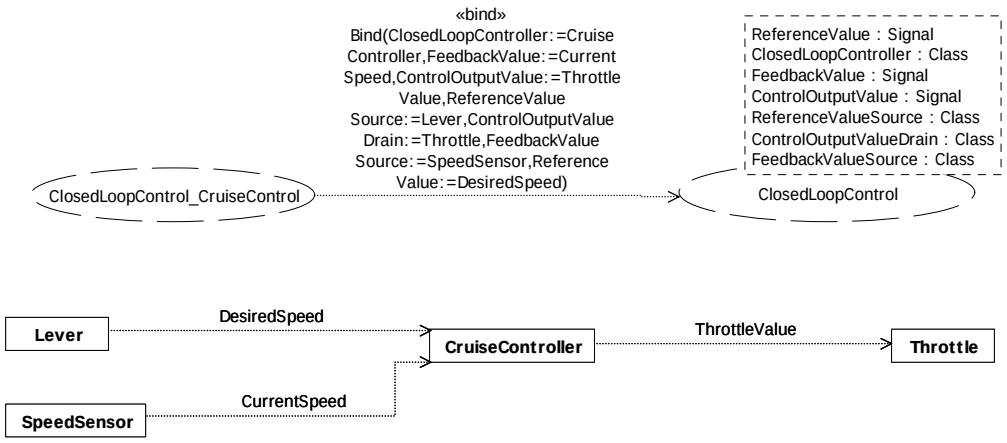


Abbildung 3.43: Das angewendete Muster in der Diagramm-Darstellung

Verschmelzungsregeln

Über die sog. *merge hints* (Verschmelzungshinweise) kann für jedes Element des Musters die Verschmelzung mit dem existierenden Modell individuell gesteuert werden. Im Rahmen der Expansion werden Elemente des Musters zu den tatsächlichen Parametern und dem *Root Context* hinzufügt. Bezüglich des Verschmelzungsverhaltens kann für jedes Musterelement bei der Musterdefinition eine der folgenden Optionen gewählt werden: *Merge* (verschmelzen), *Preserve* (bewahren), *Replace* (ersetzen) und *No Copy* (keine Kopie). Standardmäßig wird *Merge* verwendet.

Bis auf die Option *No Copy* haben diese Einstellungen nur Auswirkungen, wenn bei der Verschmelzung eine Kollision auftritt. Eine Kollision liegt in diesem Kontext vor, wenn ein Element aus dem Muster denselben Namen, denselben UML-Typ und denselben Kontext (dieselbe Vater- und Kindelementstruktur) wie das Zielelement hat. Eine Ausnahme bilden Kollisionen zwischen formalen und tatsächlichen Parametern. Diese Elemente werden bei der Option *Merge* auch verschmolzen, wenn keine Namensgleichheit gegeben ist. In Tabelle 3.10 werden die Verschmelzungshinweise erläutert.

Option	Erläuterung
Merge	Bei einer Kollision werden alle Merkmale des Musterelements, die nicht den Default-Wert haben, auf das Zielelement übertragen. Dabei wird für jedes Merkmalpaar nach folgender Fallunterscheidung vorgegangen: 1. Das Merkmal des Musterelements hat den Default-Wert: Das Zielelement wird nicht geändert. 2. Das Merkmal des Musterelements hat nicht den Default-Wert: Das Merkmal des Zielelements übernimmt den Wert des Musterelements. Demnach können mit der Option *Merge* die Merkmale im Zielelement nicht auf ihre Default-Werte zurückgesetzt werden. Ein solcher Effekt kann mit der Option *Replace* erreicht werden. Beispiel: Klasse mit `f(int x)` im Muster und `f(float x)` im Zielmodell. Im Verschmelzungsergebnis hat die Klasse die Operationen `f(int x)`.
Preserve	Bei einer Kollision wird das existierende Merkmal im Zielelement nicht geändert. Stattdessen wird ein neues Merkmal im Zielelement angelegt. Beispiel: Klasse mit `f(int x)` im Muster und `f(float x)` im Zielmodell. Im Verschmelzungsergebnis verfügt die Klasse über beide Operationen (Überladung). Bei Attributen mit unterschiedlichen Typen wird wegen des entstehenden Namenkonflikts an das Attribut aus dem Muster ein Postfix angehängt („_1“).
Replace	Bei einer Kollision wird das Zielelement gelöscht und durch das Musterelement ersetzt. Diese Einstellung kann dazu verwendet werden, um das Zielelement auf Default-Werte zurückzusetzen. Beispiel: Klasse mit `f()` im Muster und `f(float x)` im Zielmodell. Im Verschmelzungsergebnis hat die Klasse die Operationen `f()`.
No Copy	Das Musterelement wird nicht in das Zielmodell übernommen. Es werden alle im Muster definierten Referenzen zu diesem Element normal weiterverarbeitet. Beispiel: Eine als Parameter ausgewiesene Klasse in einem Muster verfügt über Attribute und hat eine Assoziation zu einer anderen Klasse. Beim Verschmelzen werden die Attribute aus dem Muster nicht in die Klasse übernommen, die im Modell als tatsächlicher Parameter fungiert, wohl aber die Assoziation.

Tabelle 3.10: Verschmelzungshinweise in *Rational XDE*

Für jedes Element im Muster kann der Verschmelzungshinweis individuell gesetzt werden. Allerdings muss das Vaterelement in der Kompositionshierarchie auf *Merge* gesetzt sein, damit die Kindelemente mit Elementen im Zielmodell kollidieren können und eine Verschmelzung stattfindet. Ein Beispiel ist eine auf *Preserve* oder *No Copy* gesetzte Klasse, die in einem Muster als Parameter ausgewiesen ist und über eine auf *Merge* gesetzt Operation verfügt. Die Einstellung *Merge* bei der Operation bleibt dann ohne Effekt, da durch die Einstellung *Preserve* oder *No Copy* der übergeordneten Klasse nie eine Kollision auftreten kann, die eine Verschmelzung der Operation auslösen würde.

Kollisionen bei Beziehungen können auch erkannt werden und eine entsprechende Verschmelzung auslösen. Die hierbei verwendeten Regeln vergleichen unter anderem die an der Beziehung teilnehmenden Partner, die Namen der Beziehungs-Endpunkte und deren Eigenschaften. Auf detaillierte Erläuterungen der dabei angewendeten Regeln wird hier verzichtet. Gleiches gilt für den Mechanismus, mit dem Code-Templates für die Muster-Elemente hinterlegt werden, die dann bei der Musterinstanziierung ins Zielmodell mitübertragen werden.

Zusammenfassend kann festgestellt werden, dass der Verschmelzungsmechanismus die im letzten Unterkapitel beschriebenen Mängel der Paket-Templates aus der UML beheben kann. Es können auch Elemente verschmolzen werden, die sich strukturell stärker unterscheiden (z. B. Operationen mit unterschiedlichen Parameterlisten) und das Verschmelzungsverhalten kann für jedes Musterelement individuell gesteuert werden. Es können mehr Beziehungen als bei den Paket-Templates „wiederverwendet“ werden und bei kollidierenden Operationen ist sowohl eine Verschmelzung als auch eine Überladung in unterschiedlichen Ausprägungen möglich.

Einordnung und Vergleich mit den UML-Templates

Wesentliche Konzepte der UML-Templates, wie z. B. Template-Parameter und eine Template-Bindung mit Parameterersetzung, sind in *Rational XDE* mit dem als *Pattern* bezeichneten Konstrukt umgesetzt. Im Folgenden werden die entdeckten Unterschiede erläutert.

In der UML wird ein Template deklariert, indem ein herkömmlicher UML-Typ, wie z. B. `Class`, `Package` oder `Component` (muss Subtyp von `TemplateableElement` sein), um eine Signatur mit einer Parameterliste ergänzt wird. Auf diese Weise können unterschiedliche „Template-Arten“ entstehen (z. B. Klassen-, Paket- oder Komponenten-Templates). Bei *Rational XDE* gibt es für das Template nur einen Metatyp, der graphisch als gestichelte Ellipse dargestellt wird. Mit dem nicht in den UML-Templates vorhandenen *Root Context* wird ein UML-Typ festgelegt, der alle nicht parametrisierten Elemente des Templates aufnehmen kann. Bei der Instanziierung eines Musters muss dem `Root Context` wie bei einem Template-Parameter ein konkretes Element zugewiesen werden. Die Zuweisung unterschiedlicher Typen zu dem *Root Context* (z. B. `Class`, `Package` oder `Component`) kann als Festlegung einer „Template-Art“ in dem oben beschriebenen Sinne interpretiert werden. Bei den mitgelieferten GoF-Mustern ist der *Root Context* beispielsweise als Paket definiert.

Ein XDE-Muster mit *Root Context* `Package` hat jedoch vermutlich eine leicht andere Semantik als das Paket-Template der UML. Wie in Kapitel 3.2.4.e erläutert, konnte im Rahmen der

Arbeit nicht geklärt werden, ob bei der Bindung eines Paket-Templates die tatsächlichen Parameter aus einem anderen Paket kommen können als aus dem bindenden Paket. Die vermutete Einschränkung, dass dies nicht möglich ist, gilt bei XDE nicht. Hier müssen die tatsächlichen Parameter nicht aus dem Paket kommen, das dem *Root Context* zugewiesen ist.

Die Ellipsen-Notation wird in XDE bei jedem *Root Context*-Typ beibehalten. Dies gilt z. B. auch für ein Template, das einen *Root Context* vom Typ `Class` und einen Parameter vom Typ `Operation` deklariert (siehe linke Hälfte von Abbildung 3.44). Die mit dieser Notation assoziierte Kollaboration ist an dieser Stelle etwas verwirrend, da der *Root Context* graphisch nicht angezeigt wird und es sich bei solch einem Template um eine parametrisierbare Klasse handelt. Der UML-Notation für Klassen-Templates würde die rechte Darstellung in Abbildung 3.44 entsprechen.

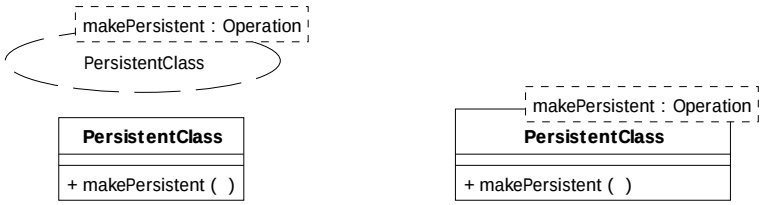


Abbildung 3.44: Ein Klassen-Template in XDE (links) und in UML-Notation (rechts)

Einen weiteren Unterschied zwischen den Mustern in XDE und den UML-Templates betrifft das Verschmelzungskonzept bei der Instanziierung von Mustern. Wie im vorherigen Unterkapitel bereits erläutert, liefert XDE in diesem Bereich für die in Kapitel 3.2.4.e identifizierten Mängel der UML-Templates entsprechende Lösungskonzepte. Der ebenfalls in Kapitel 3.2.4.e als sinnvoll erachtete Gruppierungsmechanismus für Template-Parameter fehlt aber auch in XDE.

XDE wird im Laufe der weiteren Diskussion als Referenzwerkzeug verwendet. Dementsprechend werden auch alle Konzepte dieses Werkzeugs rund um Templates bzw. Muster übernommen, inkl. der Anteile, die nicht konform mit den in Kapitel 3.2.4.e erläuterten UML-Templates sind. Dies bedeutet z. B. auch, dass die Ellipsen-Notation für Templates beibehalten wird, auch wenn es sich hier nicht um eine Kollaboration handelt. Die in XDE verwendete Bezeichnung „Pattern“ wird jedoch nicht übernommen und stattdessen der in der UML verwendete Begriff „Template“ benutzt. Unter dem Begriff „Pattern“ bzw. „Muster“ wird in dieser Arbeit ein Problem-Lösungspaar entsprechend der eingangs zu diesem Kapitel gemachten Definitionen verstanden, wobei die Lösung als „abstrakte Idee“ beschrieben wird. Die hier betrachteten Templates werden als eine Möglichkeit betrachtet, eine konkrete Lösung eines Musters in formalisierter Weise zu beschreiben. Für Templates gibt es über die Muster hinaus noch weitere Einsatzgebiete – die Schlussfolgerung, dass jedes Template eine Lösung eines Musters beschreibt, ist daher nicht zulässig.

3.2.4.g Beobachtungen zum Zusammenhang von Analyse- und Designmustern

Bei der Anwendung von Analyse- und Designmustern im Rahmen der Fallstudie wurden hinsichtlich der in der Problemstellung formulierten Frage nach der Systematisierbarkeit des Zusammenhangs zwischen System- und Softwaremodellen einige Beobachtungen gemacht.

In einem ersten Schritt wurde für verschiedene Instanzen eines Analysemusters untersucht, wie dieses auf der Designebene realisiert werden kann. Dabei konnte festgestellt werden, dass für unterschiedliche Instanzen eines Analysemusters dieselbe Designmusterkonstellation auf der Designebene vorkommen kann.

Beispiele

Diese Beobachtung ist anhand von Abbildung 3.45 und Abbildung 3.46 verdeutlicht. Abbildung 3.45 zeigt im oberen Bereich eine Instanz des Analysemusters `ClosedLoopControl` (siehe Anhang A.1) für die Funktion eines Tempomaten (vereinfachter Ausschnitt aus der Fallstudie, siehe Kapitel 2). Über einen Sollwertgeber (`SpeedDirector`) kann die gewünschte Geschwindigkeit eingestellt werden. Der Geschwindigkeitsregler (`CruiseController`) vergleicht diese fortlaufend mit der über den Geschwindigkeitssensor (`SpeedSensor`) erfassten Ist-Geschwindigkeit und berechnet bei einer Abweichung einen neuen Wert für die Drosselklappe (`Throttle`). Abbildung 3.46 zeigt dasselbe Muster am Beispiel einer Klimaanlage. Hier gibt es analog einen Sollwertgeber (`TemperatureDirector`) und einen Sensor (`TemperatureSensor`) für die Temperatur. Der Regler (`ClimateController`) berechnet in diesem Fall die Öffnung eines Ventils für einen Kompressor (`CompressorControlValve`).

Im unteren Bereich von Abbildung 3.45 ist ein mögliches Design unter Verwendung von Designmustern (siehe Anhang A.2) zu sehen, das in Richtung Wartbarkeit und Portabilität optimiert ist. Die Sensor- und Sollwertgeberdaten werden unter Verwendung des Musters `PublisherSubscriber` zur Verfügung gestellt. Das erlaubt das einfache Hinzufügen von weiteren Konsumenten zu einem späteren Zeitpunkt und umgeht eine ressourcenintensive Kommunikation durch die Verwendung des Push-Prinzips. Mit Hilfe des Musters `Repository` werden alle Eingangsdaten zentral gekapselt. Dadurch wird es möglich, die Reglung unabhängig vom Datenübertragungsmechanismus zu realisieren. Das Muster `ControlLoop` stellt ein typisches Methodengerüst für eine Regelung zur Verfügung und separiert die eigentliche Regelschleife vom Kontext. Mit Hilfe des Musters `Strategy` wird der Algorithmus für die Berechnung der Drosselklappenstellung leicht austauschbar. In dem hier gezeigten Beispiel kann alternativ eine sportliche (`Sport`) oder eine komfortable Geschwindigkeitsregelung (`Comfort`) zum Einsatz kommen.

Der untere Bereich von Abbildung 3.46 zeigt ein Design für die Klimaanlage, das für dieselben Randbedingungen wie bei dem Design des Tempomaten entwickelt wurde. Bis auf die Elementnamen sind beide Diagramme identisch.

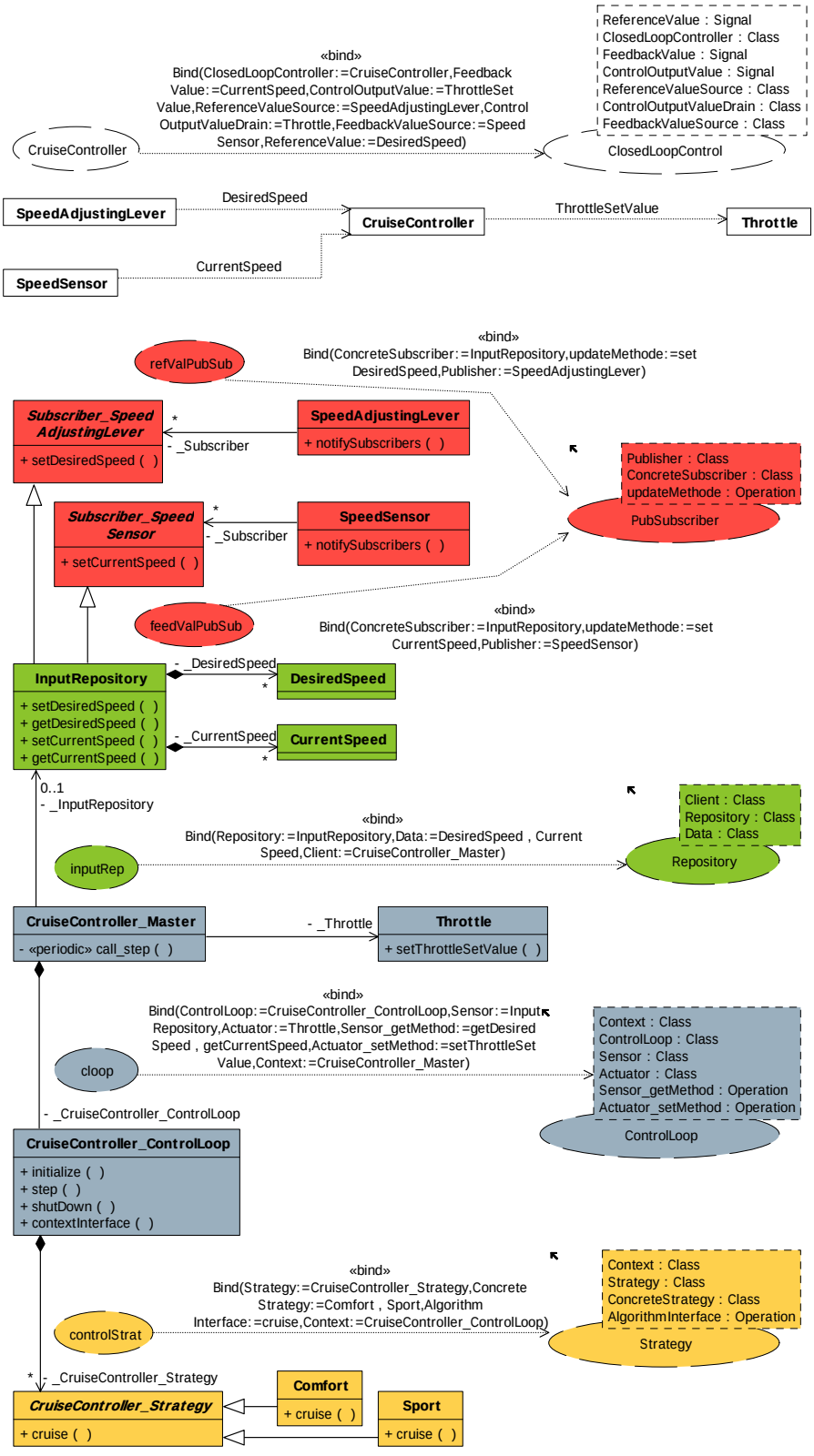


Abbildung 3.45: Analysemuster `ClosedLoopControl` am Beispiel eines Tempomaten und einem auf Wartbarkeit und Wiederverwendbarkeit optimierten Design

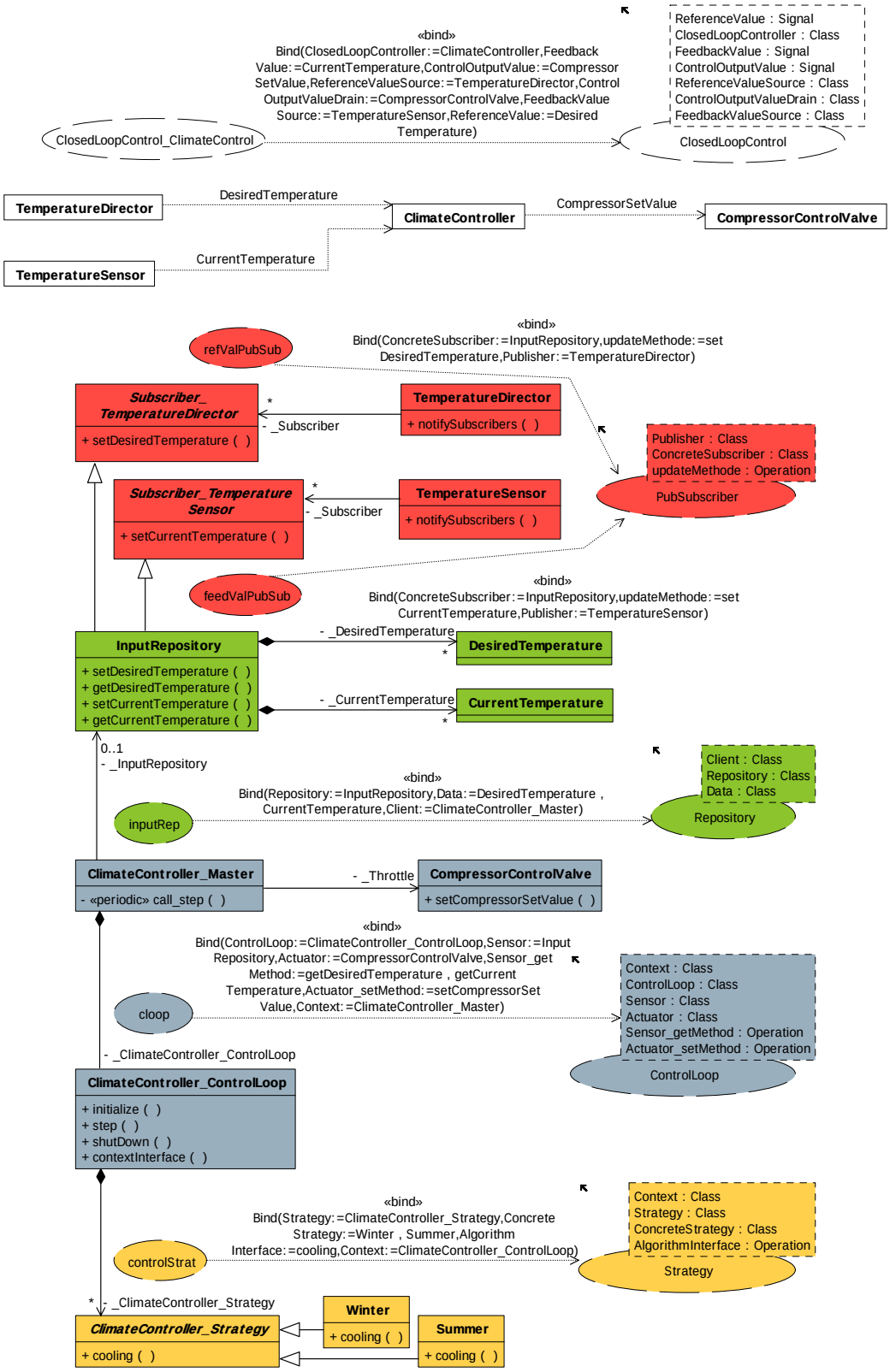


Abbildung 3.46: Analysemuster `ClosedLoopControl` am Beispiel einer Klimaanlage und einem auf Wartbarkeit und Wiederverwendbarkeit optimierten Design

Im Rahmen der Analysen konnte der beobachtete Zusammenhang zwischen Analyse- und Designmuster für weitere Beispiele bestätigt werden. Die feste Zuordnung von Design- zu Analysemustern gilt aber offenbar nur, wenn die nichtfunktionalen Anforderungen und die Plattform gleich bleiben. Ändern sich diese Randbedingungen, können auf der Designebene andere Muster zum Einsatz kommen. Abbildung 3.47 zeigt ein Design für den Tempomaten, das in Richtung Sicherheit und Robustheit optimiert wurde.

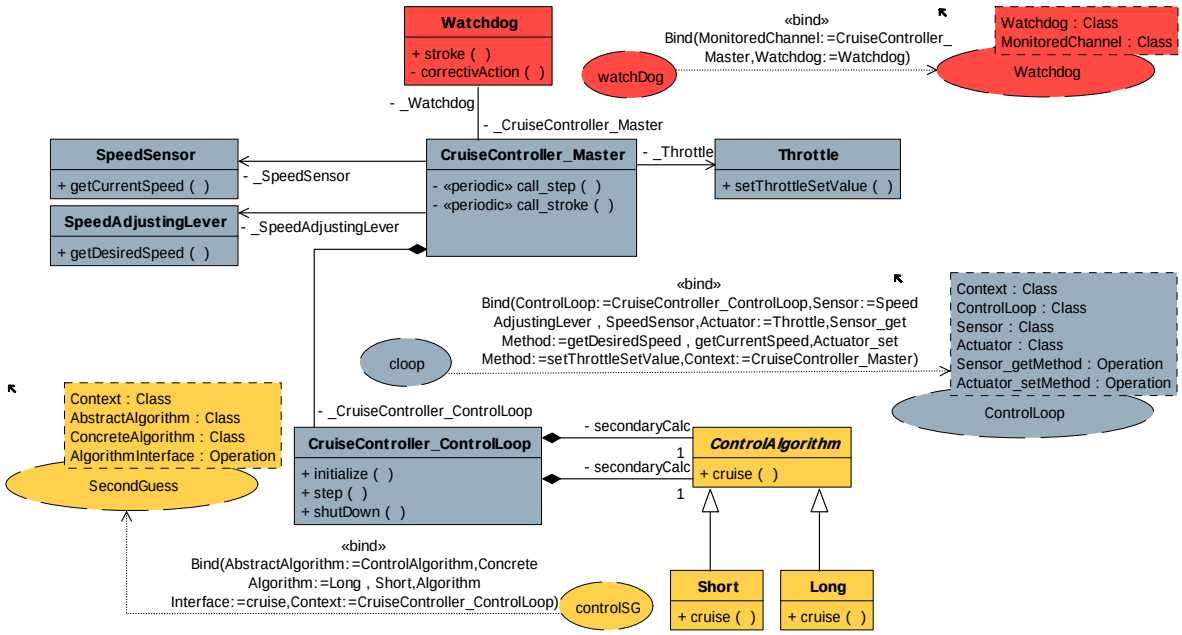


Abbildung 3.47: Ein auf Sicherheit und Robustheit optimiertes Design des Tempomaten

Die Muster zur Entkopplung der Sensorik wurden in diesem Fall weggelassen. Stattdessen wurde durch den Einsatz des Musters SecondGuess bei der die Berechnung der Drosselklappenstellung eine Diversifikation auf der Softwareebene erreicht. Die kritische Berechnung wird durch zwei unterschiedliche Algorithmen vorgenommen. In einem ersten Schritt wird mit einem genauen und laufzeitintensiven Algorithmus ein Ergebnis berechnet. Dieses wird dann mit Hilfe eines weniger laufzeitintensiven Algorithmus, dessen Ergebnis ungenauer sein kann, verifiziert. Eine Maßnahme zur Erhöhung der Robustheit ist der Einsatz des Watchdog-Musters. Mit Hilfe des *Watchdog*s wird überprüft, ob der Regler entsprechend der vorgegebenen Zykluszeit aktiv ist. Dazu sendet der Regler periodisch eine Nachricht an den Watchdog. Bleibt diese für eine bestimmte Zeitspanne aus, werden vom Watchdog korrigierende Maßnahmen ergriffen. Dabei kann es sich z. B. um den Reset des Mikrocontrollers handeln.

Variabilität

Das Beispiel der beiden Designmodelle für den Tempomaten zeigt, dass die Zuordnung von Designmustern zu einem Analysemuster mit unterschiedlichen nichtfunktionalen Anforderungen variieren kann. Dies ist angesichts der Tatsache, dass Designmuster explizit für die Umsetzung einer bestimmten nichtfunktionalen Eigenschaft entwickelt werden, auch nicht überraschend. Die Untersuchung weiterer Beispiele hat gezeigt, dass eine solche Variabilität auch von der

technischen Systemarchitektur ausgeht. So können z. B. in Abhängigkeit von der Frage, ob die Sensoren und Sollwertgeber über ein Bussystem mit dem Regler kommunizieren oder direkt an den Mikrokontroller des Reglers angebunden sind, unterschiedliche Muster für die Datenbereitstellung zum Einsatz kommen.

Die nichtfunktionalen Anforderungen und die technische Systemarchitektur spannen somit einen zweidimensionalen Lösungsraum für Designmusterkonstellationen eines Analysemusters auf. Da die Erfassung aller möglichen Kombinationen zwischen einer nichtfunktionalen Anforderung und einer technischen Systemarchitektur nicht praktikabel ist, wurde die erste Dimension auf eine Auswahl (exklusives ODER) der folgenden vier häufig vorkommenden Anforderungen beschränkt:

1. Laufzeit
2. Speicherverbrauch
3. Sicherheit und Verfügbarkeit
4. Wartbarkeit, Portabilität, Erweiterbarkeit, Wiederverwendung

Für verschiedene Analysemuster wurde zu jeweils einem dieser Optimierungskriterien eine entsprechende Designmusterkonstellation gesucht. Für jede dieser Designmusterkonstellationen wurde darüber hinaus geprüft, inwiefern sich unterschiedliche Plattformen variierend auf das Design auswirken können.

Systematik

Auf Basis dieser Grundlage konnte im Rahmen der Untersuchungen der Zusammenhang zwischen Analyse- und Designmustern für mehrere Beispiele systematisiert beschrieben werden. Im Kern beruht diese Systematik darauf, die Parameter eines Analysemusters auf Parameter eines Designmusters abzubilden. Abbildung 3.48 zeigt die Grundidee anhand einer informellen und inhaltlich unvollständigen Illustration.

Eine Abbildung zwischen zwei Parametern wird durch einen Pfeil symbolisiert. Hat ein Parameter mehr als einen eingehenden Pfeil, so wird das dazugehörige Muster auch entsprechend oft instanziiert. Dabei gilt für die übrigen Parameter, dass die eingehenden Pfeile bei jeder Instanz anliegen. Das `PublisherSubscriber`-Muster wird im Beispiel daher einmal mit `ReferenceValueSorce` und einmal mit `FeedbackValueSource` als `Publisher` angelegt, wobei der Parameter `ConcreteSubscriber` jedes Mal mit dem Inhalt des Parameters `Blackboard` aus dem `Blackboard`-Muster belegt wird. An dieser Stelle der Abbildung wird deutlich, dass die Designmuster auch untereinander *verwoben* sind. Eine solche Verwebung von zwei Designmusterinstanzen liegt in dem hier betrachteten Kontext vor, wenn bei der Parameterbelegung der einen Designmusterinstanz die Parameterbelegung der anderen Designmusterinstanz ausgelesen wird. Durch diese Verwebung wird implizit eine Instanziierungsreihenfolge der Muster festgelegt, da die Muster, deren Parameterbelegungen ausgelesen werden, zuerst instanziiert werden müssen. Der Fall, dass ein Muster mehr als einen Parameter mit mehreren eingehenden Pfeilen hat, kommt hier nicht vor.

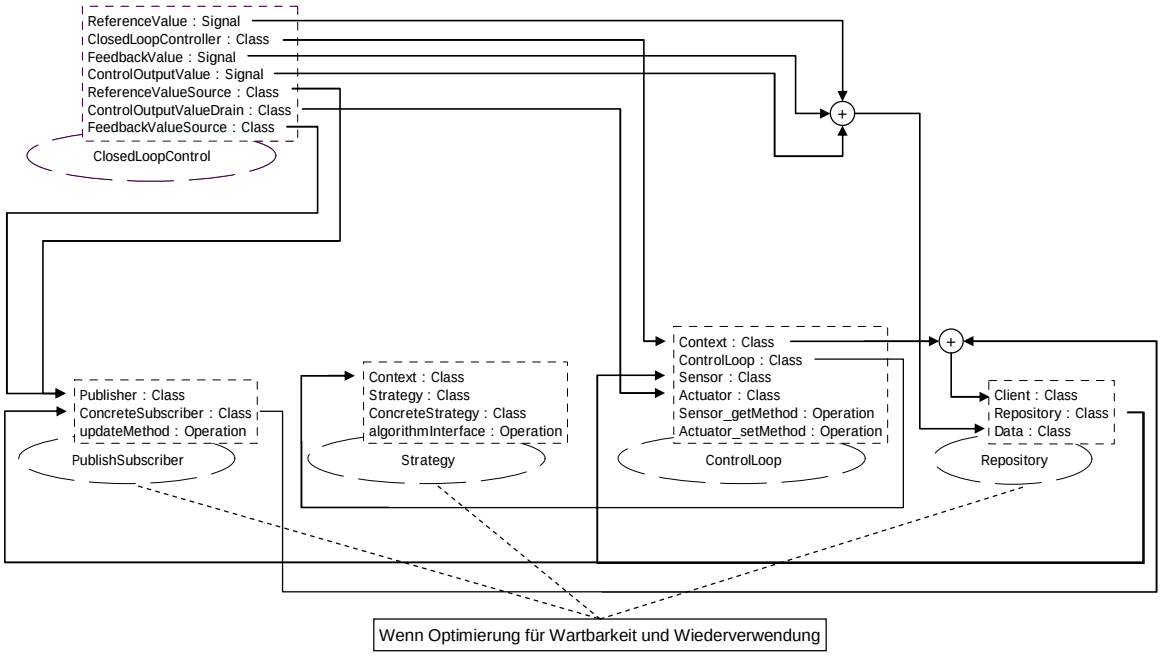


Abbildung 3.48: Parameterabhängigkeiten zwischen Analyse- und Designmustern

Mit dem +-Operator werden einzelne Elemente einer Parameterbelegung zu einer Menge vereinigt. Dadurch wird beispielsweise das Muster `Blackboard` nur einmal instanziiert, der Parameter `ConcreteDataHolder` jedoch mit drei Elementen für diese Instanz belegt. Vorraussetzung dafür ist, dass der Zielparameter eine Multiplizität größer als Eins hat. Parameter die keine eingehenden Pfeile haben, müssen bei der Instanziierung entweder mit Standardwerten belegt, oder vom Nutzer spezifiziert werden. Das Beispiel des `Strategy`-Musters zeigt, dass die Instanziierung eines Musters an Bedingungen geknüpft werden kann. Eine solche Bedingung kann sich auf eine nichtfunktionale Anforderung oder die Implementierungsplattform (Abfrage der technischen Systemarchitektur) beziehen, womit der oben angesprochenen Variabilität Rechnung getragen wird.

Verfolgbarkeit

Über die beschriebene Unterstützung bei der Erstellung von Designmodellen hinaus, konnte ein weiterer Mehrwert der Systematik beobachtet werden. Werden für alle Muster die Bindungen und zusätzlich für die Designmusterinstanzen die Abhängigkeit zur Analysemusterinstanz gespeichert, so ergibt sich eine Grundlage für einen Verfolgbarkeitsmechanismus. Dieser erlaubt es, entlang der verknüpften Musterinstanzen und deren Parameter durch die Modelle zu navigieren. Abbildung 3.49 zeigt einen solchen Pfad für das Designmodell aus Abbildung 3.47.

Exemplarisch ist die Situation geschildert, in der ein Entwickler ermitteln möchte, welche Funktion aus dem Analysemodell die Klasse `CruiseAlgorithmLong` erfüllt. Zu sehen ist, wie über Musterbindungen und deren Verknüpfung in das Analysemodell navigiert wird und auf diese Weise festgestellt werden kann, dass `CruiseAlgorithmLong` zur Implementierung des Reglers `CruiseController` beiträgt.

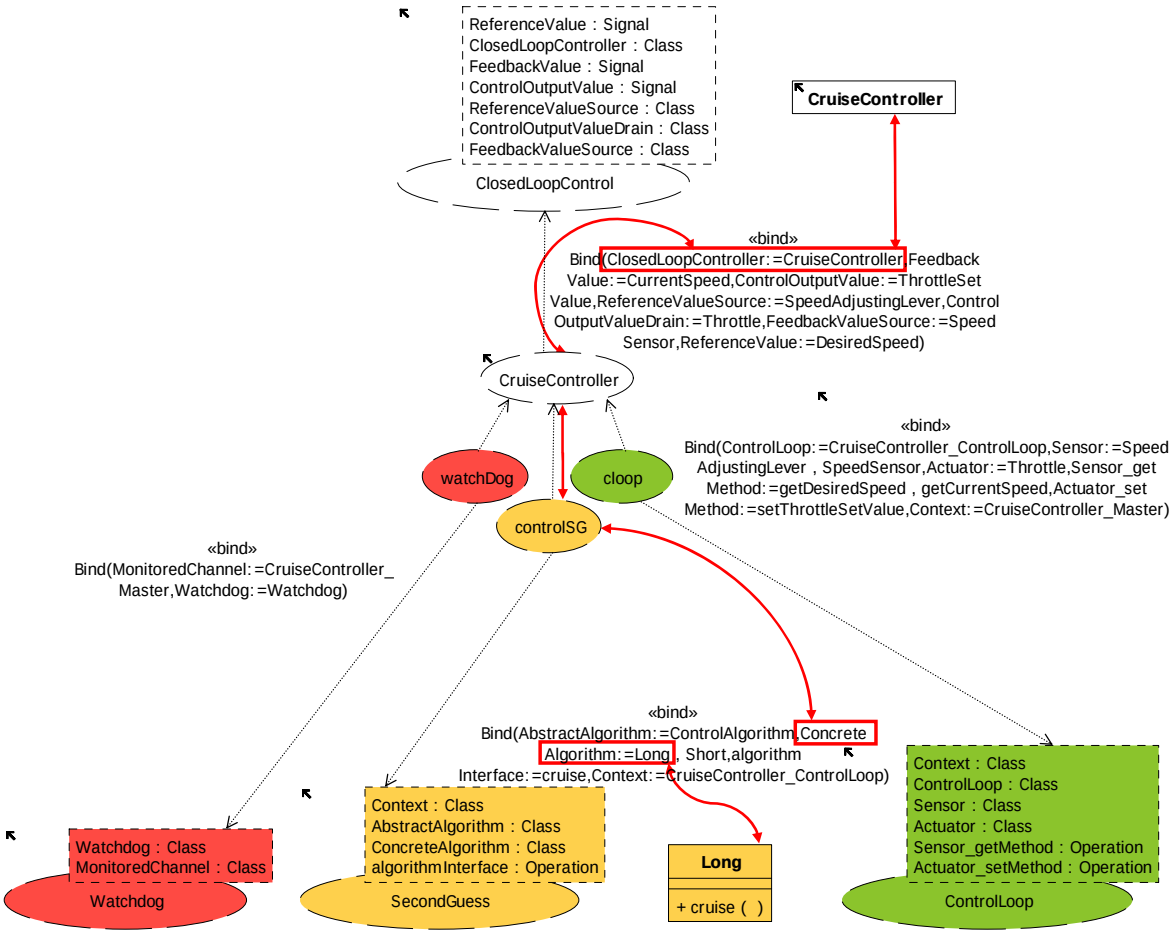


Abbildung 3.49: Verfolgbarkeit durch Verknüpfung von Mustern

3.2.5 Zwischenergebnis

Das zurückliegende Kapitel beschäftigt sich mit aktuellen Methoden und Modellierungssprachen der Softwareentwicklung. Ein Schwerpunkt bildet die ROPES-Methode, die im Laufe dieser Arbeit als Referenz bei der Diskussion der Grenzen und möglichen Erweiterungen von Softwareentwicklungsmethoden verwendet wird. Die im Rahmen von ROPES vorgesehenen Designmuster werden in einem gesonderten Kapitel vertieft, da bei ihnen in Hinblick auf Modelltransformationen ein besonderes Potential gesehen wird.

Die Frage aus der Problemstellung (siehe Kapitel 1.2) nach der Wiederverwendbarkeit von Modellen der Systementwicklung in der Softwareentwicklung, kann aus Sicht der im Rahmen von ROPES verwendeten Modellierungssprache UML eingeschränkt positiv beantwortet werden. Wie in Kapitel 3.2.1 erläutert wird, verfügt die UML über einen Profilmechanismus, der die Anpassung der Modellierungssprache an unterschiedliche Domänen vorsieht. Für die Beschreibung der logischen Systemarchitektur konnte für einige Standardbeispiele aufgezeigt werden, wie die Funktionsnetzwerke der Systementwicklungsansätze aus Kapitel 3.1.3.b mit stereotypisierten Klassendiagrammen modelliert werden können. Für diese Arbeit wurde eine eigene einfache Modellierungsweise festgelegt, die für die weitere Diskussion ausreichend ist. Bei

der Modellierung der technischen Systemarchitektur weist die UML noch erhebliche Mängel auf. Neben einem fehlenden Standardprofil für typische Hardwareeinheiten ist vor allem die Umsetzung domänentypischer Darstellungsweisen ein Problem. Für die grobe Modellierung der Verteilung von Softwarekomponenten auf Hardwareeinheiten reicht das Verteilungsdiagramm aber aus. Ein Modell der logischen Systemarchitektur lässt sich durch ein Profil demnach relativ einfach in die Softwareentwicklung übernehmen und wiederverwenden, bei der technischen Systemarchitektur funktioniert dies nur eingeschränkt. Für die weitere Diskussion wird angenommen, dass die logische Systemarchitektur im Rahmen der Softwareentwicklung entsprechend der in Kapitel 3.2.1.c vereinbarten Konvention mit den Mitteln der UML modelliert wird. Die technische Systemarchitektur wird dagegen nicht in ein UML-Modell überführt, sondern es werden die in Kapitel 3.1.3.b erläuterten Modelle der Systementwicklung weiterverwendet.

Im Falle der ROPES-Methode (siehe Kapitel 3.2.2) muss die Frage nach der Wiederverwendbarkeit von Modellen differenziert beantwortet werden. So wird zwar im Rahmen der Phase *System Engineering* die Systementwicklung grundsätzlich berücksichtigt, bei den verwendbaren Modellen herrscht jedoch weitgehend Wahlfreiheit (insb. für Inhalte der logischen Systemarchitektur) oder die vorgeschlagenen Modelle haben nur einen geringen Detaillierungsgrad. Im weiteren Verlauf des Entwicklungsprozesses ist mit dem *Object Analysis Model* ein Analysemodell für die Softwareentwicklung vorgesehen, das inhaltlich starke Überschneidungen mit der logischen Systemarchitektur hat. Der Zusammenhang mit dem entsprechenden Modell in der Systementwicklung ist nicht systematisch beschrieben. Hier herrscht ein gewisser Bruch in der Durchgängigkeit der Modelle. In Kapitel 3.2.2.g wurde eine mögliche Modelllandschaft aufgezeigt, in der die inhaltlichen Lücken von ROPES bzgl. formalisierter Modelle der Systementwicklung durch Modelle der EAST-ADL geschlossen werden. Die Erstellung des *Analysis Object Models* reduziert sich nach dieser Festlegung auf eine einfache Konvertierung aus dem Modell der Systementwicklung. Ab der Designphase gibt es in dieser Modelllandschaft keinen Modellaustausch mit der Systementwicklung mehr, so dass ab hier die ROPES-Methode in ihrer ursprünglichen Form angewendet wird. Da im Folgenden immer dieselbe Standardarchitektur angenommen wird (beschrieben in Kapitel 4.1.2), bleibt die Phase *Architectural Design* in den weiteren Ausführungen unberücksichtigt. Auch die Phase *Detailed Design* und alle nachgelagerten Phasen werden nicht vertieft. Die Diskussion konzentriert sich somit auf die Überführung des von der Systementwicklung übernommenen *Analysis Object Model* in das *Mechanistic Design Model*.

Bei dem Übergang von Analyse zu Designmodellen stützt sich die ROPES-Methode schwerpunktmäßig auf textuelle Regeln und Designmuster. Wann welches Designmuster angewendet werden kann, muss ein Entwickler nach der ROPES-Methode auf der Grundlage der textuellen Beschreibung des Musters entscheiden. Für die Nutzung im Kontext der angestrebten Modelltransformation zeigt sich die Methode an dieser Stelle zu wenig formalisiert.

Die in ROPES verwendeten Muster liefern jedoch einen ersten Anhaltspunkt für die systematische Erstellung von Modellen. Kapitel 3.2.4 beschreibt zunächst einige Grundlagen

und Kategorien von Mustern. Analyse- und Designmuster werden für die Fragestellung dieser Arbeit als besonders relevant erachtet und die in diesen Bereichen publizierten Muster daher eingehender untersucht. Im Bereich der weniger verbreiteten Analysemuster wurde eine Sammlung von Mustern gefunden, deren Abstraktionsgrad zur logischen Systemarchitektur passt und die sich inhaltlich auf die Domäne der eingebetteten Steuerungs- und Regelungssysteme ausrichten. Im Bereich der sehr umfangreich publizierten Designmuster wurde eine Auswahl von Mustern zusammengestellt, die auf typische Designfragestellungen der hier betrachteten Domäne abzielen.

Im Rahmen der Untersuchung der UML-Templates (siehe Kapitel 3.2.4.e) in Hinblick auf formalisierte Beschreibungsmöglichkeiten von Mustern, wurden einige kleine Mängel identifiziert. Dazu gehören die in der UML definierten Verschmelzungsregeln, die bei der Instanziierung eines Templates zum Einsatz kommen. Das Werkzeug *Rational XDE*, das in Kapitel 3.2.4.f untersucht wurde, enthält aber Konzepte, die diese Mängel beheben können. Es kann mit einem verhältnismäßig mächtigen Mustermechanismus aufwarten, der sowohl eine formalisierte Beschreibung eigener Muster erlaubt, als auch die Verschmelzung der Musterstruktur mit einem Zielmodell während der Instanziierung unterstützt. Der Mustermechanismus von XDE ist allerdings nicht in jedem Detail konform mit den UML-Templates. Das Werkzeug XDE und sein Musterkonzept werden für die weitere Diskussion als Referenz betrachtet und dienen in dieser Arbeit als Basisinfrastruktur für neue Mechanismen.

Bei dem intensiven Einsatz von Mustern im Rahmen der Fallstudie und weiteren Evaluierungsmodellen konnte festgestellt werden, dass der Entscheidungsprozess, welches Designmuster zum Einsatz kommt, bei der vorherigen Verwendung von Analysemustern systematisiert werden kann. So wurde beobachtet, dass Instanzen eines Analysemusters auf der Designebene in Abhängigkeit von nichtfunktionalen Anforderungen und der technischen Systemarchitektur durch dieselben Designmuster realisiert werden können. Eine erste in Kapitel 3.2.4.e beschriebene informelle Systematik beschreibt den Zusammenhang zwischen Analyse- und Designmustern mit Hilfe von Parameterabbildungen zwischen Mustern, Operatoren für den Umgang mit Mengen und Bedingungen.

Diese Systematik ist eine Antwort auf die in der Problemstellung (siehe Kapitel 1.2) gestellte Frage nach der Systematisierbarkeit des Zusammenhangs zwischen Modellen der System- und Softwareentwicklung und liefert somit auch eine Grundlage für den gesuchten Modelltransformationsmechanismus.

Auf Basis der bisherigen Erkenntnisse und Festlegungen erscheint nun das in Abbildung 3.50 illustrativ gezeigte Grobkonzept sinnvoll. Die Systementwicklung übergibt die logische und technische Systemarchitektur in Form eines wie Kapitel 3.1.3.b beschriebenen Modells an die Softwareentwicklung. Die Softwareentwicklung überführt die logische Systemarchitektur in das *Analysis Object Strutural Model* und reichert es auf der Basis von Templates mit Analysemustern an. Eine Modelltransformation überführt dieses Modell in ein neues Modell, dass ein Gerüst für das *Mechanistic Design Object Model* darstellt und auf eine Standardarchitektur abgestimmt ist. Die ebenfalls erzeugten Verfolgbarkeitslinks verknüpfen Elemente des Quell-

modells mit resultierenden Elementen im Zielmodell. Das Transformationsergebnis ist abhängig von der technischen Systemarchitektur und Designparametern, die während der Transformation abgefragt werden. Entsprechend der erläuterten Systematik sucht die Transformation nach Analysemustern im Quellmodell (in Form von gebundenen Templates) und instanziiert mit Hilfe von Regeln Designmuster im Zielmodell (durch die Bindung von Templates).

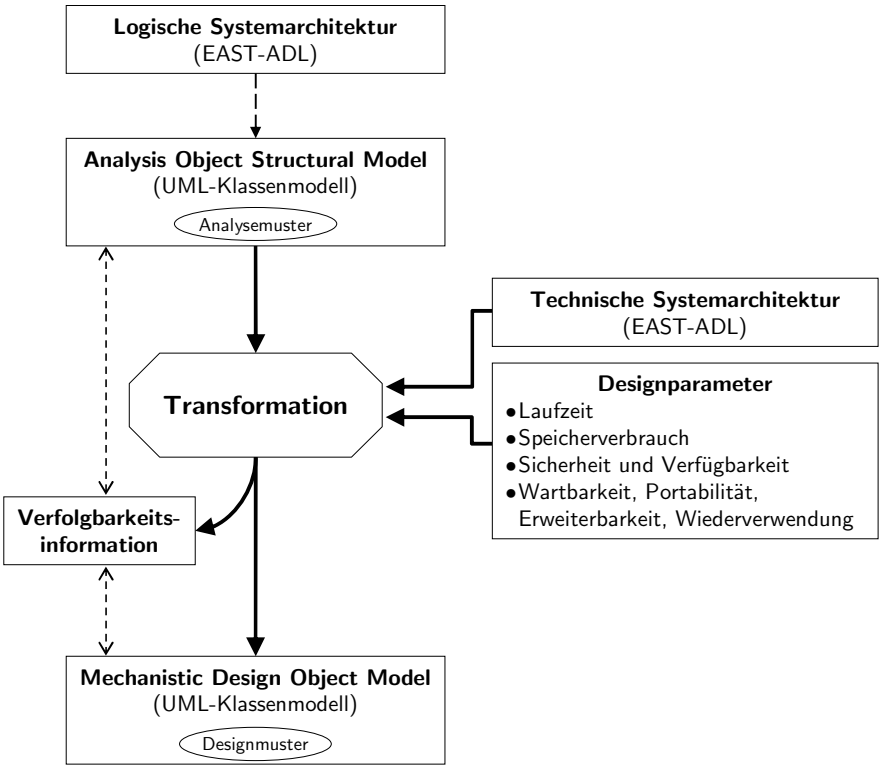


Abbildung 3.50: Illustration der angestrebten Transformation

Auf der Basis dieses Grobkonzepts kann nun die bereits in Kapitel 3.1.4 begonnene Sammlung von Anforderungen an den Modelltransformationsmechanismus verfeinert werden. Mit in den Klammern angegebenen Kürzeln werden sie für die weitere Diskussion referenzierbar gemacht.

- **UML-Klassenmodelle (UML)**: Ein- und Ausgabemodell der Transformation sind UML-Klassenmodelle. Die in Kapitel 3.2.1.c beschriebene Modellierungsweise für Funktionsnetze muss unterstützt werden.
- **Template-Definition und -Bindung (TEM)**: Ein wie in Kapitel 3.2.4.f erläutertes Template-Konzept muss unterstützt werden. Zu den Kernanforderungen gehören:
    - Ein Template kann definiert werden. Zu der Definition gehören ein Name und ein Klassenmodell, aus dem einzelne Elemente als formale Template-Parameter deklariert sind.
    - Ein Template kann gebunden werden. Bei der Bindung werden den formalen Template-Parametern die tatsächlichen Parameter aus dem Zielmodell zugewiesen.

- **Template-Expansion (EXP)**: Die Klassenstruktur eines Templates muss nach der Bindung, wie in Kapitel 3.2.4.f erläutert, expandiert und mit dem Zielmodell verschmolzen werden können.
- **Template-Instanz als Ausführungskriterium einer Regel (AKR)**: Template-Instanzen im Quellmodell müssen auffindbar und ihre Parameterbelegung auflösbar sein. Das Vorkommen einer Template-Instanz eines bestimmten Templates muss als Kriterium für die Ausführung einer bestimmten Transformationsregel definiert werden können.
- **Zusatzinformation (ZI)**: Neben dem eigentlichen Quellmodell muss eine Transformation auch Zusatzinformationen entsprechend Kapitel 3.3.1.b verarbeiten können, in Abhängigkeit derer die Transformation das Zielmodell variiert. Konkret sollen Modelle der technischen Systemarchitektur und Designparameter abgefragt werden können. Hierbei ist es akzeptabel, wenn diese Informationen für die Transformation in eine bestimmte Form gebracht werden müssen.
- **Bedingungen (BED)**: Das Anlegen von Template-Instanzen muss mit Bedingungen verknüpft werden können. Eine Bedingung kann die an einer Transformation beteiligten Modelle und die spezifizierten Zusatzinformationen (ZI) abfragen.
- **Verfolgbarkeitsinformationen (TRC)**: Es muss möglich sein, im Rahmen einer Transformation spezielle Abhängigkeitsbeziehungen zwischen Template-Instanzen aus dem Zielmodell und solchen aus dem Quellmodell anzulegen. Diese können entweder als direkte Verknüpfungen zwischen den Modellen oder in einem separaten Verfolgbarkeitsmodell realisiert werden.
- **Zugriff auf bereits erzeugte Elemente im Zielmodell (ZBE)**: Es kommt vor, dass innerhalb einer Transformation auf bereits erzeugte Elemente im Zielmodell zugegriffen werden muss, um diese z. B. für eine Parameterbelegung bei einer weiteren Template-Instanziierung zu verwenden (verweben von Templates). Dies impliziert die weitere Anforderung, dass die Reihenfolge der Template-Instanziierung festgelegt werden kann.
- **Iterationsmechanismus für Mengen von Modellelementen (IT)**: Für Mengen von Modellelementen mit bestimmter Charakterisierung (z. B. mengenwertige Parameter von Template-Instanzen) ist die Möglichkeit zur Durchführung einer bestimmten Aktion für alle Elemente der Menge erforderlich (z. B. die Instanziierung eines weiteren Templates für jedes Element aus einer Menge). Dies setzt einen impliziten oder expliziten Iterationsmechanismus für Mengen von Modellelementen voraus.

Bei der folgenden Untersuchung von Ansätzen zu Modelltransformationen soll entsprechend der oben beschriebenen Systematik geprüft werden, ob sich Musterinstanziierungen als wesentliches Element der Modellkonstruktion nutzen lassen und die beschriebene Variabilität im Designmodell möglich ist. Die oben gesammelten konkreten Anforderungen sollen im Rahmen der Bewertung Verwendung finden.

3.3 Modelltransformationen

Nach dem in den vorherigen Kapiteln Anforderungen der Domäne gesammelt wurden, beschäftigt sich dieses Kapitel nun mit dem Stand der Technik des eigentlichen Schwerpunktthemas dieser Arbeit: den Modelltransformationen. Durch die im ersten Unterkapitel behandelte *Model Driven Architecture* (MDA) wird diesem Thema zunächst in Bezug auf Konzepte und Technologien ein Rahmen gesetzt. Anschließend wird das in der MDA eingeführte Konzept der *Modelltransformation* vertieft. Zunächst werden einige Grundlagen und Grundbegriffe zu dieser Idee erläutert und eine Reihe konzeptioneller und technischer Unterscheidungsmerkmale aufgeführt. Anschließend werden einige konkrete Ansätze vergleichend vorgestellt. Die abschließende Bewertung erfolgt auf der Basis ausgewählter Anforderungen, die sich aus Untersuchungen der vorangegangenen Kapitel ergeben.

3.3.1 Model Driven Architecture / Engineering

Die *Model Driven Architecture* (MDA) der OMG [OMG b] ist ein Ansatz für modellgetriebene Softwareentwicklung (siehe Kapitel 3.1.3). Die im *MDA Guide* [Miller et al. '03] beschriebene Vision umfasst die Ziele Portierbarkeit, Interoperabilität und Wiederverwendbarkeit, die vor allem durch „architektonische Trennung unterschiedlicher Belange“ (*architectural separation of concerns*) erreicht werden sollen. Da der Ansatz im Umfeld von modellgetriebener Softwareentwicklung viele Begriffe prägt und häufig zitiert wird, werden im Folgenden einige für diese Arbeit wichtigen Begriffe und Konzepte der MDA auf der Grundlage von [Miller et al. '03] näher erläutert. Für eine detaillierte Beschreibung des Ansatzes sei auf die offiziellen OMG-Dokumente [OMG b] und entsprechende Sekundärliteratur (z. B. [Kleppe et al. '03], [IESE b] oder [Kempa et al. '05]) verwiesen.

3.3.1.a Grundlegende Konzepte und Artefakte

Der Betrachtungsrahmen aller MDA-Konzepte umfasst ein geplantes oder existierendes *System*. Dies kann nach Definition „alles“ enthalten: Ein Programm, ein Computersystem, aber auch komplexe organisatorische Gebilde, wie ganze Unternehmen. Dem Fokus der OMG entsprechend liegt der Schwerpunkt jedoch auf softwareintensiven Systemen. Ein *Modell* ist eine Beschreibung oder Spezifikation eines solchen Systems für einen bestimmten Zweck und kann sowohl graphisch als auch in Textform vorliegen.

Bei der technischen Realisierung eines solchen Systems spielt der Begriff *Plattform* eine zentrale Rolle. Eine Plattform wird zunächst als „Menge von Subsystemen und Technologien“ beschrieben, die „zusammenhängende Funktionen durch Schnittstellen und spezifizierte Nutzungsmuster anbietet, die jede von dieser Plattform unterstützte Anwendung ... benutzen kann“ [Miller et al. '03]. Solche Plattformen können generisch über architektonische Besonderheiten, auf eine bestimmte Technologie bezogen (z. B. CORBA, J2EE) oder auch hersteller- oder produktspezifisch definiert werden. Nach den Vorstellungen der OMG sollte dies in einem

einheitlichen *Plattform-Modell* geschehen, das eine Sammlung aller technischen Konzepte sowie Anforderungen an die Verbindung und Benutzung einzelner Teile einer Plattform enthält.

Ein entscheidender Mehrwert von Modellen wird in der unter dem Begriff „Plattformunabhängigkeit“ (*Platform Independence*) beschriebenen Möglichkeit gesehen, ein System unabhängig von der realisierenden Plattform zu beschreiben. Die Perspektive einer solchen Abstraktion wird mit sog. *Viewpoints* beschrieben. In der MDA werden drei solche *Viewpoints* definiert:

- Ein EDV-unabhängiger *Viewpoint* (*computation independent viewpoint*), der sich auf die Umgebung und Anforderungen des Systems konzentriert.
- Ein plattformunabhängiger *Viewpoint* (*platform independent viewpoint*), der die grobe Funktionalität des Systems beschreibt, ohne Besonderheiten einzelner Plattformen einzubeziehen.
- Ein plattformspezifischer *Viewpoint* (*platform specific viewpoint*), der schließlich auch Details der Benutzung einer bestimmten Plattform miteinbezieht.

Die dazu korrespondierenden Modelle werden analog CIM (*Computation Independent Model*), PIM (*Platform Independent Model*) und *PSM* (*Platform Specific Model*) genannt.

3.3.1.b Methodik und Modelltransformationen

Neben der Beschreibung der *Viewpoints* und deren Modelle, beschäftigt sich der MDA-Ansatz ausführlicher damit, die Beziehungen zwischen den Modellen CIM, PIM und PSM zu formalisieren und deren Entstehung und Verwendung im Kontext eines Entwicklungsprozesses zu beschreiben.

Das CIM wird manuell erstellt und dient primär dazu, Anforderungen zu beschreiben und durch die Etablierung eines einheitlichen Vokabulars ein gemeinsames Problemverständnis zu erreichen. Die im CIM gesammelten Anforderungen sollen in Bezug auf die nachgelagerten PIM- und PSM-Konstrukte nachverfolgbar (*traceable*) sein. D. h., es soll durch geeignete Mechanismen möglich sein, zu ermitteln, welcher Teil des Systems welche Anforderung realisiert. Das PIM stellt eine konkrete Beschreibung des Systems ohne Details der Plattform dar und wird laut Vorgabe „erstellt“, d. h., es geht nicht unmittelbar aus dem CIM hervor.

Anders verhält es sich bei dem PSM, das durch ein festgelegtes *Mapping* unter Berücksichtigung des Plattform-Modells über eine *Modelltransformation* aus dem PIM entsteht. Dabei handelt es sich um den Vorgang, bei dem zwei Modelle desselben Systems ineinander überführt werden. Für den Übergang von PIM zu PSM beschreibt die OMG einige Konzepte, wie dieser Vorgang möglichst formalisiert und automatisiert durchgeführt werden könnte. Abbildung 3.51 zeigt diese in einem schematischen Überblick.

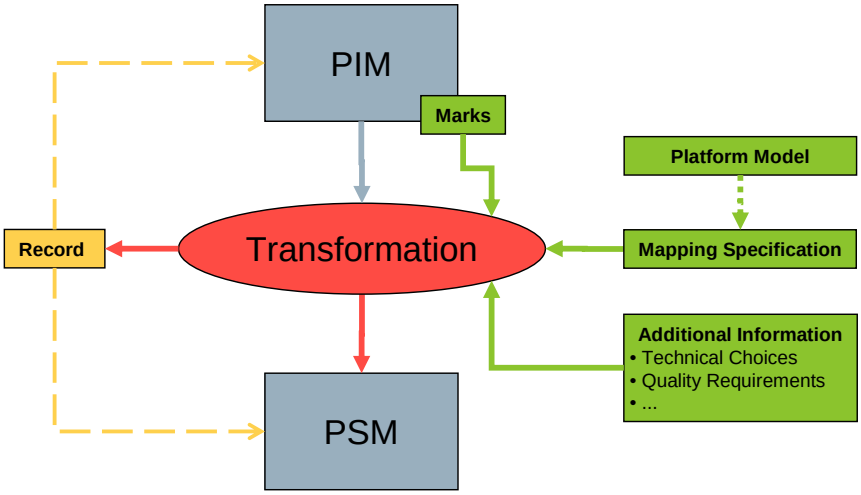


Abbildung 3.51: Schematischer Überblick zu MDA-Modelltransformationen

Modelltransformationen sind in der MDA das Schlüsselkonzept, um das oben beschriebene Konzept von *architectural separation of concerns* effizient realisieren zu können. Für die *Mapping*-Spezifikation, die die Abbildungsregeln zwischen PIM- und PSM-Elementen beschreibt, schlägt die OMG unterschiedliche Kategorien vor:

- **Modeltypen-Mapping** (*Model Type Mappings*): Es wird beschrieben, wie Typen aus dem PIM-Modell auf Typen des PSM-Modells abgebildet werden. Handelt es sich um Typen, die in einem MOF-Metamodell spezifiziert sind, wird auch von *Metamodel Mapping* gesprochen.
- **Modelinstanz-Mapping** (*Model Instance Mappings*): Durch das zusätzliche Anbringen von Markierungen (*Marks)* an Elemente aus dem PIM-Modell wird für jedes Element unabhängig vom Typ entschieden, wie es auf die PSM-Ebene abgebildet wird. *Marks* repräsentieren nach Vorstellung der OMG z. B. plattformspezifische Realisierungskonzepte oder Quality-of-Service-Anforderungen und können unter anderem in Form von Stereotypen eines *UML-Profils* oder Rollen eines Musters realisiert sein.
- **Muster und Templates**: Ein *Mapping* kann auch mit Hilfe von Mustern und Templates beschrieben werden. Unter einem Template wird in diesem Kontext ein parametrisierbares Modell verstanden, das mit einem Designmuster (vgl. Kapitel 3.2.4) vergleichbar ist, jedoch einen höheren Spezifikationsgrad besitzt. Die Abbildungsregel beschreibt hierbei, wie für die Anwendung eines Templates in der PSM-Ebene die Template-Parameter mit Elementen aus PIM belegt werden. Welche Elemente aus dem PIM für die Parameterbelegung verwendet werden, kann wieder durch *Marks* angezeigt werden. Diese können wiederum die Anwendung eines Musters auf der PIM-Ebene markieren, so dass eine *Mapping*-Regel in diesem Fall die Abbildung zwischen zwei Mustern beschreibt.

Eine *Mapping*-Spezifikation wird mit Hilfe einer *Mapping*-Sprache (*Mapping Language*) spezifiziert. Der MDA Guide beschreibt keine konkrete *Mapping* Language, verweist allerdings auf die oben genannten Konzepte und den lancierten RFP *MOF Query/View/Transformation*

[OMG '02b] (siehe Kapitel 3.3.3), der eine entsprechende Realisierungstechnologie zum Ziel hat.

Neben den verschiedenen *Mapping*-Ansätzen werden noch einige Merkmale konzeptionell beschrieben, die Transformationsmechanismen grundsätzlich erfüllen sollen. Ein solches Merkmal ist die Aufzeichnung einer Transformation (*Record of Transformation*). Diese Aufzeichnung soll festhalten, welche PSM-Elemente aus welchen PIM-Elementen hervorgegangen sind und auf diese Weise eine Verfolgbarkeit der PIM-Elemente in nachgelagerte Modelle unterstützten. Weitere Angaben werden zu möglichen Eingabedaten einer Transformation gemacht. Neben dem PIM (evtl. versehen mit *Marks*) und dem Plattformmodell werden noch Zusatzinformationen (*Additional Information)* als mögliche Eingabe einer Transformation vorgeschlagen. Diese können nach der Vorstellung der OMG z. B. Designentscheidungen (*Technical Choices*) des Architekten oder die Angabe von Qualitätsanforderungen (*Quality Requriements*) umfassen.

Die genannten Konzepte werden in dem *MDA Guide* überwiegend anhand von Modelltransformationen zwischen PIM und PSM erläutert. Es wird jedoch darauf hingewiesen, dass sowohl für das PIM, als auch für das PSM Zwischenstufen möglich sind. So kann ein PIM zunächst in ein verfeinertes PIM transformiert werden, genauso wie das entstehende PSM durch weitere Modelltransformationen verfeinert werden kann.

3.3.1.c Die MDA-Artefakte im Kontext der Systementwicklung

Die in den MDA-Dokumenten enthaltenen Beispiele stammen aus Bereichen wie Verwaltung, Betriebswirtschaft oder Finanzwesen. Dies wirft die Frage auf, welche Bedeutung die MDA-Artefakte in einem Systementwicklungsprozess wie dem in dieser Arbeit betrachteten Bereich der softwareintensiven Steuerungs- und Regelungssysteme im Kraftfahrzeug haben. Im Folgenden wird untersucht, wie sich die MDA-Artefakte mit dem bereits beschriebenen Systementwicklungsprozess abgleichen lassen.

Der Betrachtungsrahmen von MDA umfasst explizit ein System (inkl. der Nicht-Softwareanteile), was sich mit der Perspektive des Kernprozesses der Systementwicklung deckt. Ein prägendes Merkmal des MDA-Ansatzes ist die Abstraktion von der Plattform, die sich in den beschriebenen *Viewpoints* mit ihren zugehörigen Modellen *CIM*, *PIM* und *PSM* äußert. Diese Unterteilung passt auch zu den durch den Kernprozess propagierten Abstraktionsebenen. So lässt sich das CIM als ein Modell der Anforderungsphase auffassen, das alle Wirkungsketten des Gesamtsystems „Fahrzeug-Fahrer-Umwelt“ modelliert. Das PIM kann als ein Funktionsnetz der logischen Systemarchitektur interpretiert werden, das die durch Software zu realisierenden Funktionen enthält. Das PSM kann wiederum ein Softwaredesignmodell repräsentieren, das auf die technische Systemarchitektur abgestimmt ist.

3.3.2 Grundlagen

3.3.2.a Begriffsklärung

Das Konzept der *Modelltransformation* ist als wesentlicher Baustein des MDA-Ansatzes, in Bezug auf Anforderungen und Grundidee von der OMG relativ ausführlich beschrieben (siehe Kapitel 3.3.1). Der Begriff ist jedoch schon älter und kann allgemeiner gefasst werden. In der Literatur (z. B. [Czarnecki et al. '03]) wird zunächst zwischen *model-to-model* und *model-to-text* unterschieden. Letztere beschränken sich dabei auf die Generierung von Text (z. B. für Dokumentationszwecke oder Datenaustausch) oder Programmcode, was im Deutschen meist mit *Codegenerierung* bezeichnet wird. Genau genommen stellen diese Textformen jedoch auch ein Modell eines Systems dar, so dass keine klare Trennung vorgenommen werden kann.

Im Rahmen dieser Arbeit wird der Fokus auf Transformationen zwischen solchen Modellen gelegt, die über ein MOF-basiertes Metamodell und eine graphische Repräsentation verfügen. Bei der Literaturrecherche wurden deshalb generische Ansätze, die allgemeine Modelle zulassen, inkludiert, während Besonderheiten der Codegenerierung explizit ausgeklammert wurden.

Für die Betrachtungen im Rahmen dieses Kapitels wird für den Begriff „Modelltransformation" die folgende an [Sendall et al. '03a] angelehnte Definition verwendet:

Eine Modelltransformation ist ein automatisierter Prozess mit festen Regeln, der eine Menge von Quellmodellen als Eingabe erhält und eine Menge von Zielmodellen als Ausgabe produziert.

Diese allgemeingültige Definition ist allerdings nur als kleinster gemeinsamer Nenner aller Ansätze aufzufassen und beinhaltet noch keine der in Kapitel 3.1.4 bzw. 3.2.5 genannten Anforderungen, wie beispielsweise die Variabilität der Transformation in Abhängigkeit von bestimmten Parametern oder die Rückverfolgbarkeit der Transformationsergebnisse. Diese und weitere Aspekte werden vielmehr als konkrete Unterscheidungsmerkmale einzelner Ansätze betrachtet, die im folgenden Abschnitt aufgeführt und erläutert werden.

3.3.2.b Unterscheidungsmerkmale

In der Literatur werden Modelltransformationsansätze anhand bestimmter Merkmale verglichen. Die folgenden Merkmale wurden aus verschiedenen Quellen zusammengetragen und ausgewählt, wobei die Hauptquelle ein Merkmalmodell von [Czarnecki et al. '03] ist. Die Merkmale beziehen sich auf die Anforderungen an einen Modelltransformationsmechanismus und legen somit noch nicht die Art der Realisierung fest. Unterschieden werden kann nach:

- **Ziel oder Einsatzzweck**: Modelltransformationen können für die *Optimierung von Modellen* (teilweise auch als *horizontale Transformationen* bezeichnet), *Konsistenzprüfungen von Modellen*, zur *Automatisierung des Softwareentwicklungsprozesses* (auch *vertikale Transformationen* genannt) oder für *Reverse Engineering-Zwecke* eingesetzt werden. Für die horizontalen Transformationen, die eine Art *Modellevolution* darstellen, kann man zusätzlich drei verschiedene Arten unterscheiden: *perfektionierend* (im Sinne von

Refactoring oder Optimierung), *korrigierend* (im Sinne der automatischen Korrektur von unerwünschten Strukturen im Modell) und *adaptiv* (für die Anpassung an andere Gegebenheiten).

- **Generizität**: Ansätze können auf bestimmte Typen von Modellen beschränkt sein oder durch den Einsatz von Metamodellierung für unterschiedliche Modelltypen offen sein.
- **Beziehung Quelle-Ziel**: Hier wir festgelegt, ob Quell- und Zielmodelle identisch sein dürfen. Außerdem spielt es eine Rolle, ob Zielmodelle neu generiert werden oder bereits existierende Zielmodelle eventuell aktualisiert werden müssen.
- **Richtung**: Für konkrete Ansätze ist es von Bedeutung, ob die Transformation immer nur in einer festgelegten Richtung (unidirektional) erfolgen kann oder ob eine definierte Transformation auch in Gegenrichtung, also von Zielmodellen zu Quellmodellen, durchführbar ist (bidirektional).
- **Unterstützung von Nachverfolgbarkeit** (*Traceability*): Für dieses Merkmal ist es in erster Linie entscheidend, ob inhaltliche Verbindungen zwischen Elementen aus Quell- und Zielmodellen im Rahmen der Transformation festgehalten werden können und somit ein spezieller Mechanismus für eine spätere Auswertung zur Verfügung gestellt werden kann.
- **Variabilität**: Ansätze können die Formulierung verschiedener Varianten einer Transformation erlauben, die von gewissen vorgegebenen, oder sich aus den Quellmodellen ergebenden, Bedingungen abhängen.

3.3.2.c Kategorien für Realisierungsansätze

Unabhängig von der im vorherigen Abschnitt vorgenommenen Differenzierung hinsichtlich unterschiedlicher Anforderungsmerkmale, lassen sich auch bei den Realisierungsansätzen Kategorien bilden (siehe Abbildung 3.52).

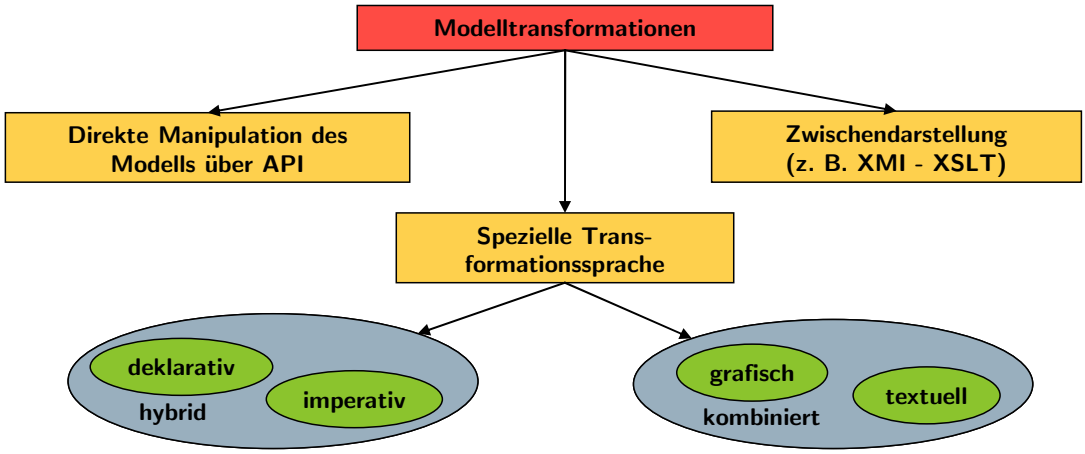


Abbildung 3.52: Kategorien der technischen Realisierung von Modelltransformationen

Viele Hersteller von UML-Werkzeugen bieten die Möglichkeit, Modelltransformationen durch *direkte Manipulation des Modells* mit Hilfe von Programmierschnittstellen (APIs) zu realisieren.

In diesem Fall werden einzelne Transformationen direkt in einer (i. d. R. objektorientierten) Programmiersprache implementiert. Dazu steht i. d. R. eine spezielle Bibliothek zur Verfügung, die häufig verwendete Operationen, wie z. B. das Lesen vorhandener oder das Erzeugen neuer Modelle und Modellelemente, bereitstellt.

Eine weitere Kategorie von Ansätzen versucht Modelltransformationen durch eine Übertragung des Problems auf eine Technologie zu lösen, für die es bereits ausgereifte Transformationsmechanismen gibt. Die Modelle werden entsprechend vor der Transformation in eine Zwischendarstellung konvertiert und danach wieder in das Ausgangsformat zurückkonvertiert. Die am häufigsten für diese Zwischendarstellung verwendete Technologie ist XML, da es hier mit *XSLT* (Extensible Stylesheet Language Transformations) [W3C '05] bereits einen sehr etablierten Transformationsmechanismus gibt. Die Nutzung von XML bietet sich hierbei auch deswegen an, weil es mit *XMI* (XML Metadata Interchange) [OMG '05a] einen Standard gibt, der beschreibt, wie über MOF definierte Modelle in XML repräsentiert werden.

Die meisten Ansätze für Modelltransformationen schlagen eine spezielle Transformationssprache vor, die in einer problemorientierten Syntax Konstrukte und Mechanismen für die Formulierung von Transformationsregeln bereitstellt. Bei den hierzu verfügbaren Arbeiten lassen sich folgende Unterscheidungsmerkmale feststellen:

- **Paradigma**: Dabei wird hauptsächlich in *deklarative* und *imperative* Ansätze getrennt. In einer deklarativen Sprache werden zunächst nur Beziehungen zwischen Variablen beschrieben, die anhand einer festgelegten Semantik ausgewertet werden, um ein Ergebnis zu erzeugen. Beispiele hierfür sind logische und funktionale Programmiersprachen. Im Gegensatz dazu werden in imperativen Sprachen die einzelnen Schritte zur Erzeugung eines Ergebnisses explizit im Programm aufgeführt. *Hybride* Ansätze beinhalten Elemente aus beiden Paradigmen.
- **Darstellung**: Viele Ansätze legen sich bereits bezüglich der Notation auf eine *graphische* oder *textuelle* Darstellung fest, obwohl die abstrakte Syntax einer Sprache grundsätzlich beide Möglichkeiten offen lässt. Da die Wahl der Darstellungsform große Auswirkungen auf die Handhabung einer Modelltransformationssprache hat, ist es dennoch sinnvoll, diese als Unterscheidungsmerkmal heranzuziehen. Auch hier besteht die Möglichkeit, beide Varianten in einer kombinierten Form zu verwenden.

3.3.2.d Auswahlkriterien für die weiteren Untersuchungen

Da es sich bei dem im Rahmen dieser Arbeit zu erarbeitenden Ansatz um eine werkzeugunabhängige und problemorientierte Lösung handeln soll, werden im Folgenden nur Arbeiten betrachtet, die der dritten Kategorie (spezielle Transformationssprache) zuzuordnen sind. Schwerpunkte der Recherchen sind zum einen die Vorschläge, die im Umfeld von MDA entstanden sind und i. d. R. durch die Industrie getrieben werden. Diese müssen bestimmte Anforderungen erfüllen, die in dem von der OMG lancierten RFP *QVT* definiert sind. Zum anderen werden aber auch von MDA unabhängige Ansätze und Projekte betrachtet, die eher im akademischen Umfeld entstanden sind.

3.3.3 Ansätze aus dem Umfeld der OMG

Der *MDA Guide* der OMG [Miller et al. '03] beschreibt zu Modelltransformationen nur die Grundidee und typische Einsatzszenarien, ohne eine genaue Aussage bzgl. einer konkreten Lösung zu machen. Um auch für diesen elementaren Bestandteil der MDA eine standardisierte und damit austauschbare Lösung zur Verfügung zu haben, lancierte die OMG im April 2002 den *Request for Proposal* (RFP) für einen neuen Standard namens *MOF 2.0 Query/Views/Transformations RFP* (*QVT*). Die zahlreichen ersten Einreichungen zu diesem RFP verdichteten sich nach und nach auf zwei Ansätze: „EXMOF" von den Firmen *Compuware* und *Sun* und „QVT" von der *QVT-Merge Group*, einem Zusammenschluss zahlreicher Firmen. Mittlerweile ist unter dem Titel *MOF QVT* eine Version des letzten Vorschlags verfügbar, die als *Final Adopted Specification* bezeichnet wird [OMG '07]. Damit ist zu erwarten, dass dieser Vorschlag ohne größere Änderungen als offizieller OMG-Standard verabschiedet wird. Im Folgenden werden die letzen beiden konkurrierenden Ansätze im Standardisierungsprozess näher analysiert.

3.3.3.a QVT-Merge Group: *QVT*

In [QVTMG '04] werden für *Queries* eine Erweiterung der OCL 2.0 vorgeschlagen und *Views* werden als Ergebnisse spezieller Transformationen nicht separat behandelt. Bei den Transformationen wird in rein deklarative *Relations* als Spezifikation und operationale *Mappings* als Implementierung einer Transformation differenziert. Mit Hilfe einer speziellen *Pattern Matching Language* können beliebigen Strukturen aus Elementen des für den jeweiligen Kontext gültigen Metamodells (kann jeweils für Quelle und Ziel vorgegeben werden) definiert werden. Zudem können bei der Definition von *Patterns* Bedingungen eingeflochten sowie freie Variablen verwendet werden, die bei einem *Matching* des *Pattern* mit passenden tatsächlichen Elementen im Modell belegt werden.

Sowohl bei den *Relations* als auch bei *Mappings* kommt das *Pattern Matching* zur Anwendung. *Realtions* bestehen aus einer Definition der freien Variablen und *Domains*, die wiederum jeweils ein *Pattern* und eine Bedingung beinhalten. Auch für die gesamte *Relation* kann eine globale Bedingung festgelegt werden. Jede *Domain* steht für ein Modell, wobei eine Transformation beliebig viele *Domains* verwenden kann. Für eine Transformation mit einem Quell- und einem Zielmodell werden entsprechend zwei *Domians* verwendet. In der mitgelieferten textuellen Syntax besitzen *Relations* folgende Struktur:

```
relation <Name der Relation> {
var <freie Variablen>;
domain <Pattern_1> when <Bedingung_1>;
domain <Pattern_n> when <Bedingung_n>;
when {<globale Bedingung>}
}
```

Bei einem *Mapping* handelt es sich um eine Verfeinerung einer *Relation*. Es umfasst eine Liste mit typisierten Parametern, einen sog. Wächter (*guard*), der als Bedingung oder *Matching*

formuliert werden kann und einem Rumpf (*body*). *Domains*, die sich in der *Relation* auf Quellmodelle beziehen, werden zu Input-Parametern, auf Zielmodelle bezogene *Domains* werden als Output im Rumpf erzeugt. Der Rumpf kann optional eine einführende *init*-Section und eine abschließende *end*-Section enthalten. Innerhalb des Rumpfes stehen eine Reihe aus imperativen Programmiersprachen bekannte Anweisungen zur Verfügung, wie z. B. Schleifen oder bedingte Verzweigungen. In der textuellen Syntax hat ein Mapping die folgende Struktur:

```
mapping <Name des Mappings> (<Input-Parameter>) : <Output>
guard <Optionale Bedingungen oder Matchings>
{
init {<Optionale Init-Section>}
<Ausdrücke zur Generierung des Outputs>
end {<Optionale End-Section>}
}
```

Sowohl *Relations* als auch *Mappings* können auf verschiedene Arten komponiert und redefiniert werden. Als Ergänzung der textuellen Notation wird auch eine graphische Notation auf Basis von UML-Objektdiagrammen vorgeschlagen, die die Transformationen allerdings nur auf Ebene der Patterns beschreibt. Die Formulierung von Bedingungen und die Implementierung der Transformation sind daher nur mit Hilfe der textuellen Syntax möglich. Unter dem Namen „QVT Interoperabilität“ wird die Einbettung in ein Framework über eine Abstraktionsschicht für Modell- und Metamodell-Repositories vorgeschlagen, das auch die Ausführung von manuell programmierten Transformationen in verschiedenen Programmiersprachen und eine Wechselwirkung mit diesen unterstützen soll. Verfolgbarkeit wird im Metamodell von QVT durch ein eigenes Paket adressiert. Dieses sieht für alle *Relations* und *Mappings* jeweils eine *Trace*-Klasse vor, die für jede Auswertung einer Regel instanziiert wird. Bei *Relations* wird die Belegung der *Patterns* und bei *Mappings* die Belegung der Parameter festgehalten. Die Instanzen der *Trace*-Klassen können mit Hilfe einer speziellen Modelltransformation erzeugt werden, die bereits in der Einreichung angegeben ist. Als *Proof of Concepts* wird in dieser Einreichung auf Implementierungen der einzelnen Mitglieder hingewiesen, die unterschiedliche Merkmale des Ansatzes erfolgreich realisiert haben.

3.3.3.b Compuware/Sun: *EXMOF*

EXMOF [Compuware et al. '04a] ist eine tiefgreifende Weiterentwicklung des von *Compuware* und *Sun* zuvor eingereichten XMOF. Das Metamodell von EXMOF setzt sich auf oberster Ebene aus den Paketen *EMOF (abgleitet von* MOF 2.0) und *Expressions* (abgleitet von der OCL 2.0) zusammen. Die Struktur einer Transformation umfasst Richtungen (*Directions*) und Klassendefinitionen (*Classes*) mit enthaltenen Eigenschaften (*Properties*). Eine Richtung steht als Variable für ein oder mehrere Pakete aus einem Metamodell. Klassen enthalten *Mappings*, die Elemente aus den über die *Directions* definierten Metamodellen verknüpfen. Die Eigenschaften der Klassen können optional mit einer *Direction* assoziiert werden und stehen dann für Objekte, die Quelle oder Ziel einer Transformation sind. Ansonsten werden sie als Attribute des jeweiligen *Mappings* verwendet. Die eigentliche Transformation wird dann innerhalb der Klassen in Zuweisungsregeln (*Assignment Rules*) definiert, in denen die Eigenschaften mit Werten

belegt werden. Durch den Einsatz von OCL-Ausdrücken können dabei auch Bedingungen formuliert werden.

Als konkrete Notation für dieses Konzept wird eine Erweiterung von UML-Klassendiagrammen vorgeschlagen, in denen *Directions* als Doppelstriche mit Namen dargestellt werden. Das in Abbildung 3.53 gezeigte Beispiel vermittelt einen groben Einblick in diese Notation. Es zeigt eine Transformation zwischen einem UML-Klassenmodell und einem relationalen Datenbankschema. Die Doppelstriche stehen für die *Directions*, die Klassen in der Mitte stehen für die Transformationsklassen. Sie enthalten die *Mappings* (die *Directions* schneidende Assoziationen) zwischen den Elementen auf der linken und rechten Seite, die jeweils aus dem der *Direction* zugeordneten Metamodell stammen. Die Kompositionsbeziehung zwischen den Transformationsklassen impliziert, dass die Ausführung der Transformation „AttributeToColumn“ immer mit der Ausführung von „ClassToTable“ verknüpft ist.

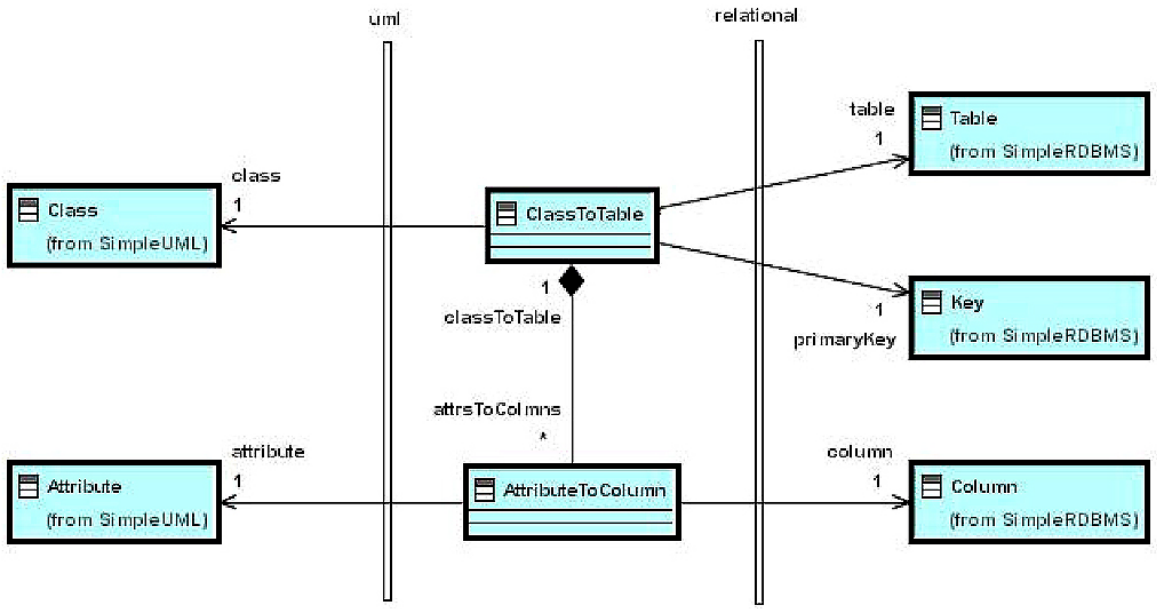


Abbildung 3.53 Notation von EXMOF anhand eines Beispiels [Compuware et al. '04b]

Da auf Basis der graphischen Notation jedoch nicht alle Feinheiten der abstrakten Syntax realisiert werden können, gibt es darüber hinaus noch eine grammatikalische Syntax. Diese ist bezüglich des Sprachumfangs relativ einfach und dadurch leicht verständlich, führt aber aufgrund der Vielzahl an Klassen in einer Transformationsdefinition zu einer großen Menge an Code.

Insgesamt ist der Ansatz rein deklarativ und unterstützt so auch die Ausführung von Transformationen in verschiedene Richtungen und eine inkrementelle Anwendung. Für die Realisierung von Verfolgbarkeit wird darauf verwiesen, dass die *Mappings* auf MOF zurückführbar sind und somit über existierende Technologien persistent gehalten und ausgelesen werden können. Ein eigener Mechanismus wird daher nicht bereitgestellt. Als *Proof of Concept* wird auf das Werkzeug *OptimalJ* von *Compuware* verwiesen.

3.3.4 Ansätze aus dem akademischen Umfeld

Die im letzten Unterkapitel erläuterten Ansätze sind Antworten auf den RFP für QVT, die überwiegend aus dem industriellen Umfeld stammen und auf die durch die OMG formulierten Anforderungen ausgerichtet sind. Daneben gibt es aber auch in der wissenschaftlichen Literatur eine Vielzahl von Beiträgen, die unabhängig von QVT Modelltransformationsansätze präsentieren. Ausgehend von verschiedenen anderen Zusammenfassungen, wie z. B. in [Guelfi et al. '03a] und [Sendall et al. '03a], werden im Folgenden einige Beiträge zusammengefasst und eingeordnet. Über die in Kapitel 3.3.2.c vorgestellten Kategorien von Realisierungsansätzen hinaus, lassen sich die Ansätze noch anhand der folgenden grundlegenden Lösungsideen unterscheiden:

- **Ansätze mit formaler Fundierung**: Den Modellierungssprachen der OMG wird oft ein geringer Grad an Formalisierung vorgeworfen, da Teile der Semantik oft in natürlicher Sprache definiert sind. Sofern Modellierungssprachen der OMG verwendet werden, überträgt sich dieser Mangel auch auf Modelltransformationen. Einige Ansätze versuchen daher, das „Problem Modelltransformation" durch Übertragung auf bereits existierende formale Konzepte wie Attributgrammatiken oder logische Programmiersprachen zu lösen. Darüber hinaus werden spezifische formale Grundlagen für eine Transformationssprache entwickelt. Zu dieser Kategorie gehören z. B. die Attributgrammatiken, das Framework auf Basis von Maude und BOTL (siehe Kapitel 3.3.5).
- **Anlehnung an Graphtransformationen:** Da sich objektorientierte Modelle auch als Graphen interpretieren lassen, liegt der Gedanke nahe, sich an Techniken für Graphtransformationen zu orientieren, für die es eine umfangreiche Theorie und zahlreiche Implementierungen gibt. Betrachtet man ein UML-Modell als Graph, so lassen sich Modelltransformationen über Graphersetzungsregeln realisieren. Jede solche Regel wird über ein linksseitiges und ein rechtsseitiges Graph-Muster dargestellt. Wird eine Transformation ausgeführt, so führt jedes Vorkommen eines linksseitigen Graph-Musters einer Regel dazu, dass dieses Vorkommen durch das rechtsseitige Graph-Muster ersetzt wird. Für dieses allgemeine Prinzip gibt es eine Vielzahl von Methoden aus der Graphentheorie. [Agrawal '04] weist jedoch auf viele Unzulänglichkeiten dieser Ansätze in Bezug auf diesen Anwendungskontext hin. So muss beispielsweise mit typisierten, attributierten und gelabelten Graphen gearbeitet werden, um UML-Modelle als Graph darzustellen. Da es sich bei Graphtransformationen um Ersetzungsregeln handelt, ist es zudem meist erforderlich, auf Kopien der Quellmodelle zu arbeiten. Deshalb werden Graphtransformationen für viele Ansätze nur als formale Grundlage oder als Inspirationsquelle für eigene, erweiterte Mechanismen verwendet. Zu dieser Kategorie gehören z. B. GReAT, die Story Diagramms, VMT und die Ansätze von Khriss und Judson.

Die Kategorien stellen keine disjunkten Klassen im Sinne einer vollständigen Klassifikation dar, sondern dienen lediglich der Einordnung nach wesentlichen Charakteristiken der jeweiligen Ansätze. Die in Kapitel 3.2.4.g skizzierte Systematik impliziert, dass Muster in einer festen Reihenfolge instanziiert werden müssen (Anforderung ZBE). Um die Untersuchung der zahl-

reichen Beiträge zu fokussieren, werden einige Ansätze, die für diese Systematik keine geeignete operationale Semantik definieren, nur überblicksartig im Kapitel 3.3.5 erwähnt.

3.3.4.a J. Whittle: Framework auf Basis von Maude

In [Whittle '02] wird ein Ansatz präsentiert, der Elemente eines UML-Klassenmodells in einer Faktenbasis hinterlegt und auf dieser Basis mit Hilfe eines Frameworks in der logischen Programmiersprache *Maude* Modelltransformationen realisiert. Bei den Modellelementen beschränkt sich der Beitrag auf Klassen, Assoziationen und Constraints, die etwa in der Form `class(Klassenname,Attribute,Operationen)` in der Faktenbasis repräsentiert werden. Die Transformationsregeln bestehen aus einer linken und einer rechten Seite und werden so interpretiert, dass jeder Teil eines Modells, der mit der linken Seite in Einklang gebracht werden kann, unter der entsprechenden Belegung der Variablen durch die rechte Seite ersetzt wird.

Das in dem Beitrag beschriebene Beispiel beschäftigt sich mit einer Transformation die zeigt, dass ein Klassendiagramm ein Refactoring eines anderen Klassendiagramms ist. Unter Refactoring wird hierbei eine Strukturänderung des Klassendiagramms aus Designoptimierungsgründen verstanden, wobei das Verhalten nicht geändert wird. Der Einsatzzweck des Ansatzes ist schwerpunktmäßig bei der Optimierung von Modellen und der Konsistenzprüfung zu sehen.

Die Transformation ist in dem beschriebenen Stand nur teilweise automatisiert, da die einzelnen Regeln manuell in einer bestimmten Reihenfolge ausgeführt werden müssen. Für den Fall, dass ein Element durch eine Transformation seinen Namen ändert, wird vorgeschlagen, Klassen mit dem Stereotyp `<<trace>>` automatisiert anzulegen, die Ein- und Ausgabeelemente einer Transformation verlinkt. Die Ausführung von Regeln kann an Bedingungen geknüpft werden, die das Modell abfragen. Auf diese Weise kann auch Variabilität von Transformationsregeln indirekt unterstützt werden.

3.3.4.b Graph Rewriting and Transformation Language (GReAT)

Das in [Agrawal '04] präsentierte System für Modelltransformationen *GReAT* beinhaltet einen UML-basierten Ansatz für die Spezifikation von Modelltransformationen, der sich auf formale Graphgrammatiken und -transformationen stützt. Er besteht aus drei Teilen:

- Einer auf UML-Klassendiagramme zugeschnittenen *Pattern Specification Language*. Ein damit spezifiziertes Pattern beschreibt einen Subgraphen und kann Kardinalitäten enthalten. Darüber hinaus sind hierarchisch verschachtelte Patterns möglich.
- Einer Graphersetzungs-Transformationssprache (*Graph Rewriting Transformation Language*), mit der sich Transformationsregeln in Form von sogenannten *Produktionen* formulieren lassen. Da auf vorhandene Graphtransformationstechniken zurückgegriffen wird, die nur innerhalb eines Graphen operieren, werden Quell- und Zielmodell als ein Graph dargestellt. Damit kann eine Produktion als 4-Tupel aus einem *Pattern Graph*, einer *Mapping Funktion*, einem *Guard*-Ausdruck und einer Attributbelegung dargestellt werden. Der Pattern Graph wird über die *Pattern Specification Language* ausgedrückt und enthält – im

Gegensatz zu der gängigen zweiseitigen Notation bei Graphtransformationen - *ein* Pattern mit Elementen aus dem Quell- und Zielmodell. Die *Mapping*-Funktion definiert für jedes Element aus dem Pattern, ob es bei Anwendung der Regel im Graph gematched, gelöscht oder neu erzeugt werden soll. In dem *Guard* können über OCL-Ausdrücke zusätzliche Bedingungen für die Ausführung der Regel formuliert werden, die Attributbelegung weist den neu erzeugten Elementen Werte zu.

- *Kontrollstrukturen* für die Graphersetzung und -transformation, die über Konzepte wie Sequenzierung, parallele Ausführung, hierarchische und rekursive Definitionen oder bedingte Verzweigungen eine Steuerung der Ausführungsreihenfolge der Produktionen erlauben.

Im Rahmen der Recherchen konnte nicht geklärt werden, ob und wie Verfolgbarkeit unterstützt wird. Anhand einer selbst entwickelten Werkzeugkette wurde der Ansatz evaluiert. In einem anderen Beitrag [Karsai et al. '03] diskutieren die Autoren auch den Zusammenhang von GReAT und QVT und listen eine Reihe von „Herausforderungen und Möglichkeiten" der Verwendung von Graphtransformationen für die MDA auf, die vor allem die intuitive Darstellung und den Rückgriff auf bewährte Technologien als Vorteile hervorheben.

3.3.4.c Visual Model Transformation (VMT)

Ein weiterer Ansatz für die Transformation von UML-Modellen, dessen abstrakte Semantik auf Graphtransformationen beruht, wird mit der deklarativen Transformationssprache VMT in [Sendall et al. '03b] präsentiert. Hier setzt sich eine Transformation aus einer Menge von Transformationsregeln zusammen, die sich wiederum in ein *Matching-Schema* und ein *Ergebnis-Schema* untergliedern. Ein Matching-Schema enthält Ausführungsbedingungen und Input-Argumente für die Regel und wird als Graph repräsentiert, in dem die Knoten als Platzhalter für Elemente oder Mengen im Quellmodell dienen, die vorhanden sein müssen bzw. nicht vorhanden sein dürfen. Das Ergebnis-Schema definiert ein Zieldiagramm als Funktion der Elemente aus dem Matching-Schema und wird ebenfalls als Graph repräsentiert. In einem *Regelanordnungs-Schema* kann durch Iteration oder bedingte Verzweigung der Kontrollfluss zwischen verschiedenen Transformationsregeln festgelegt werden.

In dem Beitrag werden alle VMT-Konzepte auf Begriffe der Theorie zu Graphtransformationen zurückgeführt, um den Ansatz formal zu fundieren. Inspiriert wurde der Ansatz unter anderem durch das Werkzeug MEDAL [Guelfi et al. '03b]. Dieses Add-in zu der UML-Moddellierungsumgebung *IBM Rational XDE* [IBM b] (siehe Kapitel 3.2.4.f) realisiert einen Modelltransformator, der eine in einem eigenen *UML-Profil* definierte Transformationssprache interpretieren kann. Diese Transformationssprache enthält bereits einige wesentliche Konzepte, die in VMT mit eingeflossen sind.

3.3.4.d I. Khriss: Design-Unterstützung durch Muster-basierte Transformationen

In [Khriss et al. '99a] wird ein Ansatz beschrieben, in dem mit Hilfe von Modelltransformationen ein UML-Designmodell schrittweise verfeinert wird. Transformationsregeln werden in Form von Verfeinerungsschemata in einer Bibliothek zusammengefasst, die jeweils ein

abstraktes und ein detailliertes Modell umfassen. Das detaillierte Modell verfeinert das abstrakte Modell durch die Anwendung eines Designmuster, das für diese Verfeinerung charakteristisch ist. Neben Klassenmodellen sind auch Zustands- und/oder Kollaborationsmodelle möglich. Die Verfeinerungsschemata werden aus einzelnen elementaren Verfeinerungen aufgebaut, sog. *Micro-Refinements*, für die Korrektheitsbeweise im Sinne formaler Verifikation angegeben werden (diese werden in [Khriss et al. '99b] genauer beschrieben). Damit ist nach Ansicht der Autoren auch das gesamte Verfeinerungsschema verifizierbar. Eine Transformation eines Modells erfolgt schrittweise durch Anwendung ausgewählter Verfeinerungsschemata auf ausgewählte Teile des Modells. Als Vorteil der Verfeinerung auf Basis von Mustern wird die Verfolgbarkeit von Designentscheidungen angeführt. Ein Konzept, das den Einsatz der Verfeinerungsschemata dauerhaft verfolgbar macht, wird jedoch nicht beschrieben, sondern stichpunktartig als Aufgabe an ein Modellierungswerkzeug formuliert.

3.3.4.e Judson et al.: Kontrolliertes Refactoring von Modellen

In [Judson et al. '03] wird ein auf perfektionierende Modellevolution abzielender Ansatz vorgelegt, der ähnlich wie bei dem zuvor betrachteten Ansatz von [Khriss et al. '99a] durch den Einsatz von Designmustern eine schrittweise Verfeinerung eines Modells erreicht. Eine solche Verfeinerung wird in einem *Transformation Pattern* beschrieben, das aus drei Teilen besteht:

- Einem *Source Pattern*, das die Menge von Elementen aus dem Quellmodell definiert, auf die die Transformation angewendet wird. Das Source Pattern wird als Metamodellfragment ausgedrückt, das aus Unterklassen der Klassen im UML-Metamodell besteht.
- Einem *Transformation Schema*, das die Struktur des Zielmodells beschreibt, indem es die Klassen von Modellelementen zeigt, die von den Transformationen erzeugt werden und die Klassen von Quellmodellelementen, die entfernt werden. Dies wird in einer speziellen Notation ebenfalls als Metamodellfragment ausgedrückt.
- Einem *Transformation Constraint*, welches als eine Art Nachbedingung die Beziehungen bestimmt, die nach der Transformation zwischen Ziel- und Quellelementen gelten.

Das in dem Beitrag verwendete Beispiel für ein *Transformation Pattern* auf der Basis des Musters *Abstract Factory* von [Gamma et al. '95] ist in Abbildung 3.54 gezeigt.

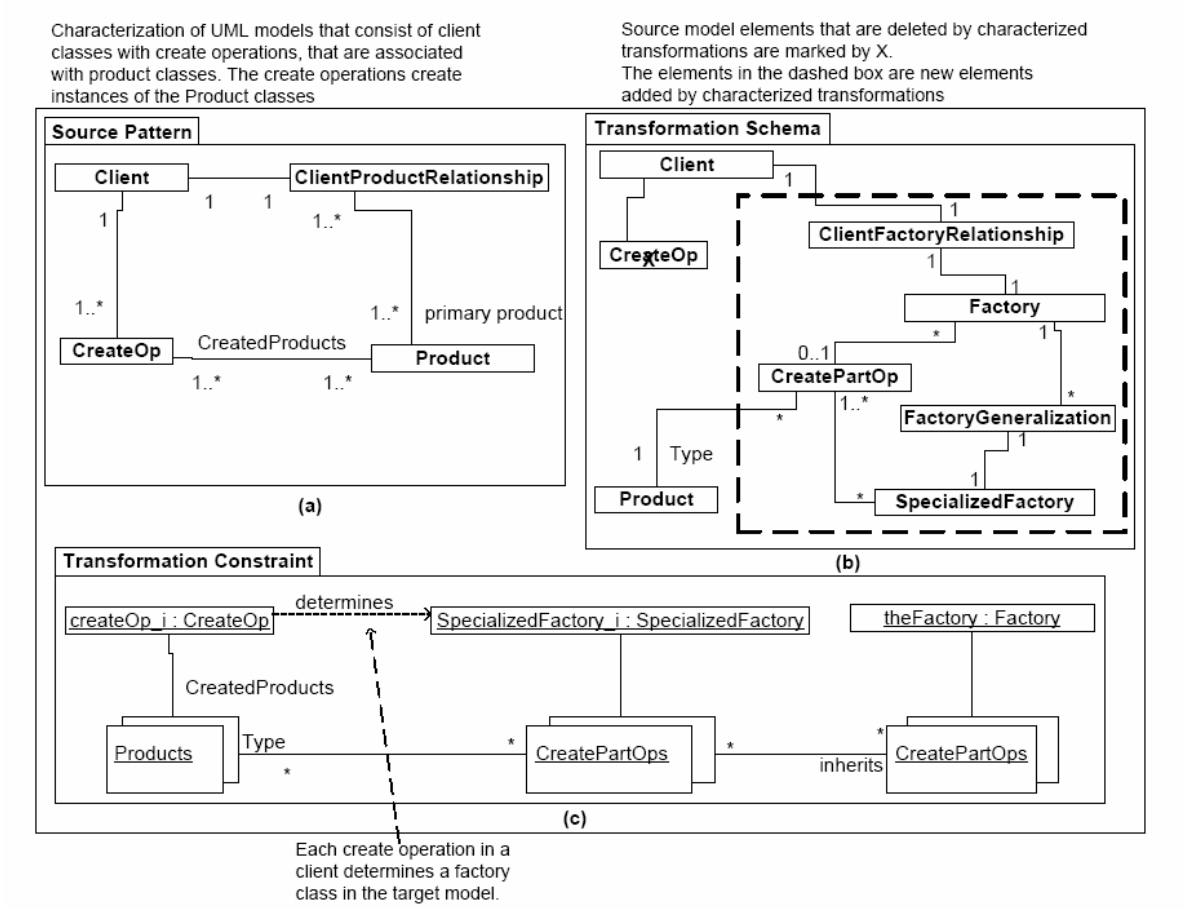


Abbildung 3.54: Beispiel eines *Transformation Pattern* [Judson et al. '03]

3.3.4.f D. Milicev: Erweiterte UML-Objektdiagramme

In [Milicev '02] wird eine auf UML-Objektdiagrammen basierende Notation für die Beschreibung von *Mappings* zwischen Modellen vorgeschlagen. Ziel ist dabei die Erzeugung eines Zwischenmodells (*Intermediate Model*) zur einfacheren Codegenerierung. Zunächst müssen Metamodelle für Quell- und Zwischenmodell definiert werden, deren Elemente dann in dem erweiterten Objektdiagramm verwendet werden können: Instanzen der zu erzeugenden Elemente aus dem Zwischenmodell als Objekte, die eigentliche Logik der Transformation in stereotypisierten UML-Paketen, die Bedingungen, Schleifen, Parametrisierung u. Ä. ausdrücken. In diesen Paketen können alle Elemente aus dem Quellmetamodell als Werte bestimmter vordefinierter Tagged Values verwendet werden. Mit dieser Modellierungsweise lässt sich nach Ansicht des Autors effektiv automatisch Code für die Transformation erzeugen. Der Beitrag erwähnt einige Beispiele und Fallstudien, in denen die Transformationstechnik für die Erzeugung eines in Hinblick auf die Codegenerierung optimierten Zwischenmodells erfolgreich eingesetzt wurde, wobei jedoch offen bleibt, ob sich der Ansatz auch für andere Szenarien eignet. Ein Verfolgbarkeitsmechanismus wird nicht erwähnt.

3.3.4.g E. Willink: UMLX

Die in [Willink '03b] vorgestellte Transformationssprache UMLX nutzt ebenfalls erweiterte UML-Diagramme für die Beschreibung von Transformationen zwischen sog. Schemata (vergleichbar mit Metamodellen). Hier werden UML-Klassendiagramme durch eine zusätzliche Transformationssyntax erweitert, die unter anderem ein Rauten-förmiges Symbol für eine Transformation einführt. Mit diesem in [Willink '03a] genauer beschriebene Element können Verknüpfungen zwischen Quell- und Zielelementen vorgenommen werden. Damit lässt sich über jeweils zusammenhängende Elemente eine Trennung in die linke Seite ("Left Hand Side", Quelle) und die rechte Seite ("Right Hand Side", Ziel) einer Transformation vornehmen. Die genaue Ausgestaltung der Transformation wird über die direkte Verbindung von Quell- und Zielelementen in Form spezieller Abhängigkeitspfeile ausgedrückt. Für diese gibt es drei Transformationsoperatoren:

- Erhaltung (*Preservation*)*:* Das Quellelement wird übernommen
- Weiterentwicklung (*Evolution*): Erzeugung eines neuen Elementes
- Entfernung (*Removal*): Entfernen des Quellelementes

Ähnlich wie bei Graphtransformationen wird jedes Auftreten einer Struktur, das der linken Seite einer Transformation entspricht, durch deren rechte Seite gemäß der zusätzlichen Notation ersetzt. Abbildung 3.55 zeigt einige Elemente der Notation anhand eines einfachen Beispiels, bei dem UML-Klassen in ein Modell für relationale Datenbanken überführt werden. Die beschrifteten Rauten mit den Pfeilen verdeutlichen, dass die Zielelemente auf Basis des Quellelementes neu erzeugt werden sollen. Für Attributwerte der neu erzeugten Elemente kann dazu auch auf Attribute der Quellelemente zugegriffen werden. Die große Raute in der Mitte bedeutet den verschachtelten Aufruf einer weiteren Transformation, die analog der abgebildeten Transformation definiert ist. Die „halben" Rauten links und rechts am Rand verdeutlichen den Input und Output der Transformation.

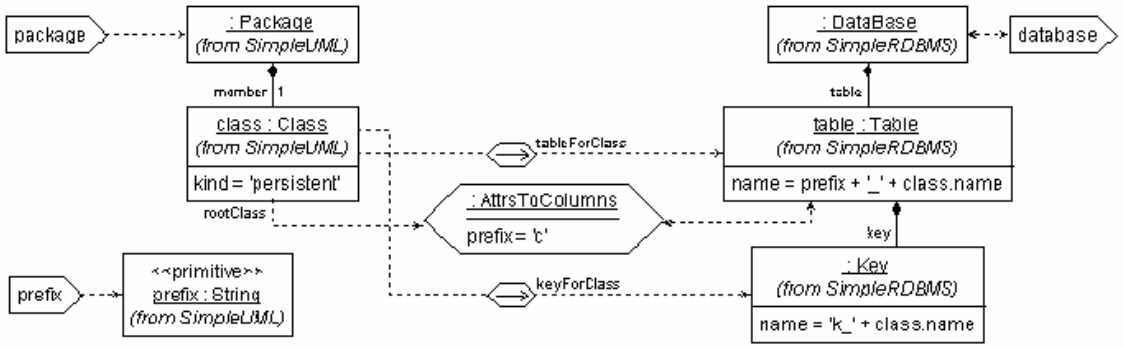


Abbildung 3.55: Beispiel-Definition einer Transformationsregeln in UMLX [Willink '03a]

Als prototypische Realisierung des Ansatzes wird auf einen Editor für das Metamodellierungs-Tool GME [Ledeczi et al.] verwiesen. Ein Transformator, der selbst als Sammlung von UMLX-

Transformationen definiert und in XSLT implementiert werden soll, war zum Zeitpunkt der Veröffentlichung des Beitrags noch in der Entwicklung.

3.3.4.h Model Transformation Language MOLA

Der in [Kalnins et al. '04] vorgestellte Ansatz kombiniert die Konzepte einer strukturierten Programmiersprache und von Mustern in einer graphischen Transformationssprache. Ein MOLA-*Programm* (engl. *program*) transformiert ein Quell- in ein Zielmodell, wobei beide Modelle jeweils konform zu einem Metamodell sein müssen. Das Programm besteht aus einer Sequenz von sog. *Statements*, die in einer graphischen Form modelliert werden, die von den Autoren mit einem Kontrollflussgraphen (engl. *structured flowchart*) verglichen wird. Die wichtigsten *Statements* sind Regeln (engl. *Rules*) und Schleifen (engl. *Loops*). Eine Regel enthält ein Muster (engl. *Pattern*), das für den *Matching*-Teil der Regel steht, und eine *Action*-Spezifikation. Letztere umfasst Klassen oder Assoziationen, die bei einem Auftreten des Musters erzeugt oder gelöscht werden sollen sowie Modifikationen von Attributwerten. Für die graphische Modellierung der Muster und des *Action*-Teils wird eine eigene Form von UML-Objektdiagrammen verwendet. Abbildung 3.56 zeigt einen Ausschnitt des auch in den anderen Ansätzen oft verwendeten Beispiels, bei dem UML-Klassen in ein Modell für relationale Datenbanken überführt werden.

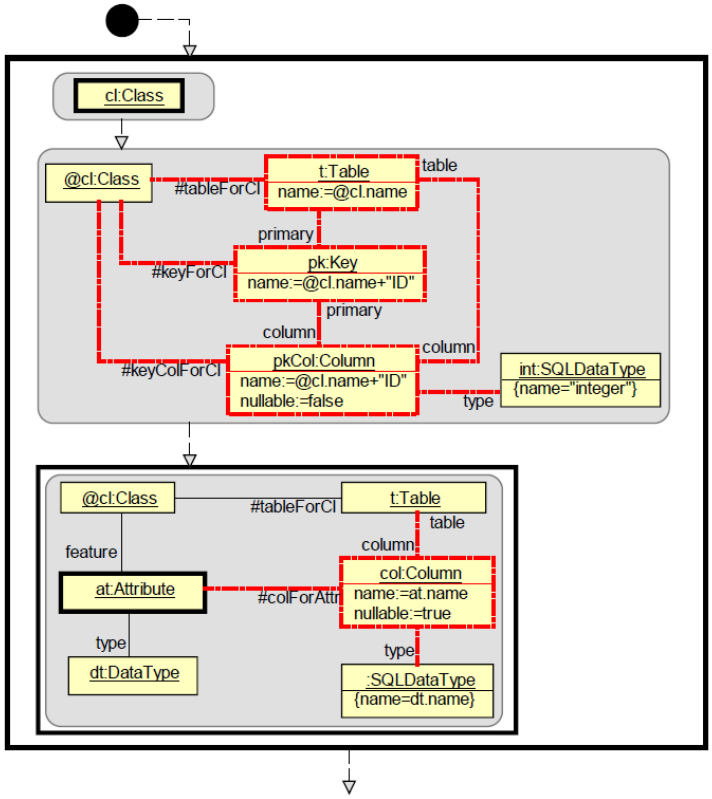


Abbildung 3.56: Ausschnitt aus einer MOLA-Transformationsregel [Kalnins et al. '04]

Eine Schleife enthält einen Schleifenkopf, der durch ein Muster mit einem Element realisiert wird und kann als einer der beiden Typen FOREACH oder WHILE deklariert werden, der den Schleifenkopf entweder als Platzhalter für ein Element (bei FOREACH) oder als Bedingung (bei WHILE) verwendet. Bei der Ausführung der Transformation werden dann für ein Quellmodell, das eine Instanz des Quellmetamodells ist, alle definierten Statements ausgeführt. In einer Erweiterung dieser „basic" MOLA können zusätzlich noch Kardinalitätsbeschränkungen und verschachtelte Muster verwendet werden.

Die mit diesen Erweiterungen versehene Sprache wurde nach Angaben der Autoren anhand verschiedener typischer Beispiele aus dem MDA-Umfeld erfolgreich getestet. Als Vorteil wird insbesondere die intuitive Verständlichkeit durch die graphische Repräsentation hervorgehoben. Die in den Beispielen verwendeten Muster und Regeln sind allerdings sehr einfach und beziehen sich auf separierbare Teilaspekte einer Transformation. Damit bleibt offen, ob sich der Ansatz auch für Transformationen eignet, deren Komplexität mit der des Beispiels aus Kapitel 3.2.4.g vergleichbar ist.

3.3.5 Weitere Ansätze

Bidirectional Object-Oriented Transformation Language (BOTL)

Die in [Braun et al. '03a] vorgestellte graphische Transformationssprache BOTL ist vor allem durch die Integration von Werkzeugen motiviert, die Objekte austauschen, die nach unterschiedlichen Metamodellen strukturiert sind. Der Einsatzzweck liegt daher schwerpunktmäßig bei horizontalen Transformationen.

Der Beitrag enthält einen Modellierungssprachen-unabhängigen Formalismus sowie eine konkrete Abbildung desselbigen auf ein *UML-Profil*. Eine BOTL-Spezifikation besteht aus einer Menge von Regeln, die jeweils Fragmente aus einem Quellmodell in Zielmodellfragmente überführen und letztere mit dem bestehenden Zielmodell vereinigen. Die Fragmente werden als sog. *Modellvariablen* dargestellt, die UML-Objektdiagrammen entsprechen, in denen Attribute und Bezeichner mit Termen anstatt konkreter Werte belegt sind. Wird eine Regel angewendet, so wird im Quellmodell nach einem Auftreten des definierten Quellfragments gesucht. Ist die Suche erfolgreich, so werden die zugehörigen Zielmodellfragmente erzeugt. Mit Hilfe der Unterlegung des Ansatzes durch einen mathematischen Formalismus, der in [Braun et al. '02] beschrieben ist, kann die Anwendbarkeit von Regeln oder die Konformität des Ergebnisses zu einem Metamodell verifiziert werden, das im Rahmen von BOTL als Klassendiagramm modelliert wird. Trotz der Beschränkung des Ansatzes auf Klassenmodelle ist nach Ansicht der Autoren durch ein *Mapping* der Klassen auf UML- oder MOF-Metaklassen auch ein Einsatz für die MDA möglich [Braun et al. '03b]. An einer Werkzeugunterstützung für den Ansatz wird momentan noch gearbeitet. Ein dezidierter Mechanismus für Verfolgbarkeit und Variabilität wird nicht beschrieben.

Nutzung von Attributgrammatiken

Der in [Dehayni et al. '03] vorgestellte Ansatz, schlägt für die Realisierung von Modelltransformationen die Nutzung von Attributgrammatiken vor. In der vorgestellten Lösung müssen alle Modelle bzw. Metamodelle in ein textuelles Format überführt werden. Im Falle von MOF-basierten Modellen wird dabei auf die ebenfalls von der OMG standardisierte Sprache *HUTN* (Human Usable Textual Notation) zurückgegriffen, die eine textuelle Repräsentation von MOF-Modellen bietet. Die einzelnen Konzepte aus dem Quell-Metamodell werden als Nichtterminale der Attributgrammatik dargestellt, die zusätzliche, auf das Ziel-Metamodell bezogene Attribute enthalten. Die eigentliche Transformationslogik wird in den Berechnungsvorschriften (semantischen Regeln) der Attributgrammatik definiert, die für jede Produktion bei der Traversierung des Quellsyntaxbaumes ausgeführt werden. Das Zielmodell liegt dann ebenfalls in HUTN in einem Attribut des Wurzelknotens dieses Syntaxbaumes vor.

Als Vorteil wird angeführt, dass die Lösung rein deklarativ, sehr präzise und durch Verwendung vorhandener Technologien im Umfeld von Attributgrammatiken einfach und effizient implementierbar ist. Bei dem im Beitrag geschilderten Beispiel handelt es sich um die häufig diskutierte Transformation, die eine UML-Klasse mit Attributen in eine Tabelle einer relationalen Datenbank transformiert. Weitere Anwendungsszenarien waren bislang auf Migrationen beschränkt, so dass keine Erfahrungen für die Einsetzbarkeit des Ansatzes für komplexere Transformationen vorliegen.

Erweiterung der Object Constraint Language (OCL)

Viele der Einreichungen zu QVT (siehe Kapitel 3.3.3) stützen sich auf Teile der OCL (siehe Kapitel 3.2.1), beschränken sich jedoch dabei auf Abfragen oder die Formulierung von Bedingungen innerhalb einer Transformation. Es gibt darüber hinaus aber auch Ansätze, die die OCL direkt durch Erweiterungen zu einer Transformationssprache ausbauen.

In [Pollet et al. '02] wird vorgeschlagen, das Ergebnis einer Transformation mit Hilfe von Vor- und Nachbedingungen darzustellen, die als OCL-Constraints auf Metamodell-Ebene ausgedrückt werden. Um diesen deklarativen Teil einer Transformation um eine Ausführungssemantik zu ergänzen, wird eine Erweiterung der OCL um *Actions* als Möglichkeit gesehen. Eine so erweiterte OCL hätte nach Ansicht der Autoren den Vorteil, dass keine weitere Sprache für Modelltransformationen hinzukäme und bestehende Mechanismen in Tools für Transformationen wiederverwendet werden könnten. Letzteres wird allerdings nur mit einem einfachen theoretischen Architekturmodell für ein Tool zur Manipulation von Modellen belegt. Das mitgelieferte Beispiel beschreibt leider nur eine sehr einfache Transformation (Generierung von *getter*- und *setter*-Methoden für Klassen), was eine Beurteilung des Ansatzes hinsichtlich komplexerer Transformationen schwierig macht.

In einem weiteren Vorschlag [Akehurst '04] wird dagegen die Erweiterung der OCL um das Konzept der *Relationen* vorgeschlagen. Da sowohl die OCL als auch die mathematische Relationentheorie auf der Mengentheorie basieren, sind nach Ansicht des Autors nur wenige Erweiterungen der OCL nötig, von denen die wichtigsten aufgeführt werden. Diese in OCL

ausgedrückten Relationen werden als mögliche Notation für die *relations* aus dem Ansatz der *QVT-Merge Group* (siehe Kapitel 3.3.3.a) vorgeschlagen. Eine operationale Semantik, die z. B. in Form von *Actions* definiert werden könnte, wird explizit ausgeklammert. Eine ähnliche Idee wird etwas ausführlicher in einem früheren Beitrag über einen relationalen Ansatz für Modelltransformationen beschrieben [Akehurst et al. '02]. Basis ist hier die Idee, Elemente der Relationentheorie als Objektmodell zu modellieren und auf dieser Basis Transformationsbeziehungen zwischen Modellen zu beschreiben. Für die Modellierung von Transformationen werden Paare und Relationen verwendet. Diese werden zunächst in UML modelliert, spezifische Eigenschaften der Relationen müssen jedoch ebenfalls mit Hilfe von OCL-Ausdrücken beschrieben werden. In dem Beitrag geht es primär um Transformationen zwischen Elementen ein und derselben Sprache (abstrakte Syntax, Semantik-Domäne, konkrete Syntax), die Autoren schließen aber nicht aus, dass der Ansatz auch für Modelltransformationen zwischen verschiedenen Modellen geeignet ist. Allerdings bezieht sich der Ansatz nur auf Relationen und lässt die genaue Spezifikation der operationalen Semantik offen.

3.3.6 Bewertung

Die untersuchten Ansätze für Modelltransformationen werden im Folgenden anhand der bisher gesammelten domänenspezifischen Anforderungen bewertet. Abschließend soll die Frage beantwortet werden, ob eine Modelltransformation auf Basis der in Kapitel 3.2.4.g skizzierten Systematik mit Hilfe der untersuchten Ansätze möglich ist.

3.3.6.a Vorauswahl anhand grober Unterscheidungsmerkmale

Die in Kapitel 3.2.5 gesammelten Anforderungen sind so detailliert, dass eine volle Unterstützung von existierenden Ansätzen nicht erwartet werden kann. Insbesondere die adäquate Umsetzung von Templates (Anforderung TEM) und der Expansionsmechanismus (Anforderung EXP) erscheinen als zu speziell, um sie als bereits vorhandene Merkmale bestehender Ansätze vorauszusetzen. Deshalb wird bei der Bewertung nicht verlangt, dass jede Anforderung in der formulierten Form direkt erfüllt wird, sondern vielmehr beurteilt, in welchem Maße die geforderte Funktionalität durch Mechanismen des jeweiligen Ansatzes realisiert werden kann.

Die ausführlicher untersuchten Ansätze werden zunächst anhand des Schemas von [Czarnecki et al. '03] (siehe Kapitel 3.3.2) charakterisiert. Um weiter betrachtet zu werden, müssen sie mit den Vorgaben aus Tabelle 3.11 übereinstimmen, die sich aus den bisher gesammelten Anforderungen ableiten.

Merkmal	**Vorgabe**
Ziel / Einsatzzweck	Automatisierung des Softwareentwicklungsprozesses.
Generizität	Stereotypisierte UML-Klassendiagramme müssen unterstützt werden.
Beziehung Quelle-Ziel	Quelle und Ziel sind unterschiedliche Modelle.
Richtung	Unidirektional (Analyse- zu Designmodell).
Verfolgbarkeit	Möglichst dedizierter Mechanismus (Festhalten von Designentscheidungen).
Variabilität	Ja (in Abhängigkeit von nicht-funktionalen Anforderungen und Plattform).

Tabelle 3.11: Vorgegebene Merkmale entsprechend des Schemas von [Czarnecki et al. '03]

Für das Merkmal Variabilität reicht es aus, wenn der Ansatz die Möglichkeit bietet, die Ausführung einer Regel an eine Bedingung zu knüpfen, die ein Modell abfragt. Auf diese Weise kann ein Transformationsparameter als zusätzliches Modellelement realisiert werden. Das Merkmal Verfolgbarkeit ist zwar grundsätzlich erforderlich, ein dezidierter Mechanismus wird aber in diesem ersten Vergleich nicht zwingend gefordert. Zum einen lässt sich eine sehr simple Form von Verfolgbarkeitslinks zwischen den Quell- und Zielelementen bei jedem Ansatz realisieren – hierbei handelt es sich schwerpunktmäßig um eine Werkzeugfrage. Zum anderen bietet nur ein untersuchter Ansatz einen dezidierten Mechanismus. Um bei der weiteren Analyse noch mehrere Ansätze vergleichen zu können, wird der Verfolgbarkeitsmechanismus als ein Merkmal betrachtet, das im Zweifelsfall durch eine eigene Erweiterung des Ansatzes erfüllt werden muss.

In Tabelle 3.12 werden die untersuchten Ansätze für Modelltransformationen in Bezug auf die vorgegebenen Merkmale bewertet. Die Einschätzungen lehnen sich an eine Diplomarbeit [Buchwald '05] an, die im Umfeld dieser Arbeit durchgeführt wurde (siehe Kapitel 6.2). Wie aus der Tabelle hervorgeht, sind die Ansätze EXMOF, GReAT, VMT und der Vorschlag der *QVT-Merge Group* mit den Vorgaben kompatibel. Nur der letztgenannte Vorschlag erfüllt auch das optionale Verfolgbarkeitskriterium durch einen dezidierten Mechanismus.

	Ziel/ Einsatz-zweck	Generizität	Beziehung Quelle-Ziel	Richtung	Verfolgbarkeit	Variabilität	Kompatibel zu Vorgaben
EXMOF	allgemein	Meta-modellierung (MOF)	alles möglich	bidirektional	nein	indirekt über OCL-Bed.	ja
QVT	allgemein	Meta-modellierung (MOF)	alles möglich	Relationen bidirektional, Mappings unidirektional	ja	Trans-formations-parameter	ja
Whittle	perfektion-ierend, Konsistenz-prüfung	Elemente in Faktenbasis	nur innerhalb Modell	unidirektional	nein	indirekt über Bedingungen	nein
GReAT	allgemein	Meta-modellierung (UML)	nur ver-schiedene Modelle	unidirektional	nein	indirekt über OCL-Bed.	ja
VMT	allgemein	UML-Modelle	alles möglich	unidirektional	nein	indirekt, Kontrollfluss zwischen Regeln	ja
Khriss	perfektion-ierend	UML-Modelle	nur innerhalb Modell	unidirektional	nein	Benutzer wählt Ver-feinerungs-schema	nein
Judson	perfektion-ierend	UML-Modelle	nur innerhalb Modell	unidirektional	nein	nein	nein
Milicev	Zwischen-modell für Code-generierung	Meta-modellierung (UML)	nur ver-schiedene Modelle	unidirektional	nein	nein	nein
UMLX	allgemein	Meta-modellierung (MOF)	nur ver-schiedene Modelle	unidirektional	nein	nein	nein
MOLA	allgemein	Meta-modellierung (UML)	nur ver-schiedene Modelle	unidirektional	nein	nein	nein

Tabelle 3.12: Charakterisierung der Ansätze über Unterscheidungsmerkmale

3.3.6.b Bewertung anhand der Anforderungen

Tabelle 3.13 zeigt die Ergebnisse einer Analyse, in der die Erfüllung der Anforderungen aus Kapitel 3.2.5 durch die noch verbleibenden Ansätze ermittelt wurden.

Im Falle von QVT und EXMOF konnte bei der Bewertung auf Transformationsregeln zurückgegriffen werden, die im Rahmen einer Diplomarbeit erstellt wurden (siehe unten). Für die anderen beiden Ansätze konnte auf Basis der vorhandenen Dokumentation keine konkrete Implementierung vorgenommen werden, weshalb die Einschätzungen hier nur auf konzeptionellen Überlegungen beruhen.

Kriterium / Ansatz	UML	TEM	EXP	AKR	ZI	BED	TRC	ZBE	IT
EXMOF	●	⊙	○	●	◉	●	⊙	⊙	●
QVT	●	◉	⊙	●	●	●	◉	◉	●
GReAT	●	⊙	○	●	⊙	●	⊙	○	◉
VMT	●	⊙	○	●	◉	●	⊙	○	◉

● Wird bereits im vollen Umfang unterstützt.
◉ Kann indirekt oder mit Hilfe kleinerer Erweiterungen unterstützt werden.
⊙ Kann mit Hilfe größerer Erweiterungen unterstützt werden.
○ Wird nicht unterstützt.

Tabelle 3.13: Erfüllungsgrad der verbliebenen Ansätze in Bezug auf die Anforderungen aus Kapitel 3.2.5

Zwei Kriterien werden von allen Ansätzen voll erfüllt: So unterstützen alle Kandidaten UML-Klassendiagramme (Anforderung UML) und bieten in unterschiedlicher Form Möglichkeiten, Template-Instanzen aufzufinden und diese als Ausführungskriterium für eine Transformationsregel zu formulieren (Anforderung AKR). Die Wiederverwendung von Templates über Transformationen hinaus (Anforderung TEM) ist bei keinem Ansatz möglich, es bestehen lediglich verschieden ausgeprägte Möglichkeiten, innerhalb von Transformationen durch eigene Implementierung eine vergleichbare Definition vorzunehmen. Einen expliziten Bindungs- und Expansionsmechanismus (Anforderung EXP) bietet keiner der Ansätze, lediglich bei dem Vorschlag der *QVT-Merge Group* lässt sich dieser in Ansätzen durch spezielle Transformationsregeln nachbilden. Die Abhängigkeit von Zusatzinformationen (ZI) kann bei ebendiesem Ansatz über Transformationsparameter erreicht werden, die anderen Ansätze bieten hier lediglich den Umweg über zusätzliche Quellmodelle. Die gewünschte Verfolgbarkeit auf der Basis verlinkter Template-Instanzen (Anforderung TRC) muss bei allen Ansätzen über eine Erweiterung realisiert werden, wobei das Konzept der *QVT-Merge Group* mit dem bereits existierenden

Verfolgbarkeitsmechanismus eine bessere Ausgangsbasis bietet. Die Möglichkeit, bereits erzeugte Elemente im Zielmodell zu benennen und später erneut auf sie zuzugreifen (Anforderung ZBE) kann bei QVT und EXMOF über die Definition von Attributen bzw. Variablen realisiert werden, wobei dies in den imperativen *Mappings* der *QVT-Merge Group* flexibler ist. Die beiden graphischen Ansätze bieten diese Möglichkeit nicht. Über Mengen kann in allen Ansätzen iteriert werden (Anforderung IT), bei QVT und EXMOF über OCL, bei den anderen teilweise etwas eingeschränkt über Kontrollstrukturen.

Zusammenfassend muss festgehalten werden, dass Kernanforderungen wie die Handhabung von Templates, das Variabilitätsmodell und die Möglichkeit der Nachverfolgbarkeit von keinem Ansatz umfassend unterstützt werden. EXMOF und QVT kommen den gestellten Anforderungen am nächsten und lassen es sinnvoll erscheinen, konkreter zu prüfen, inwiefern sich die nicht erfüllten Anforderungen durch Erweiterungen realisieren lassen. In der Diplomarbeit von [Buchwald '05] wurde die in Kapitel 3.2.4.g beschriebene Systematik zu Evaluierungszwecken auf Basis der Ansätze QVT und EXMOF für das ebenfalls in diesem Kapitel beschriebene Beispiel zu großen Teilen umgesetzt. Da die Realisierung der fehlenden Anforderungen mit Bordmitteln zu sehr umfangreichen und verschachtelten Transformationsregeln führt bzw. teilweise überhaupt nicht möglich erscheint, kommt die Arbeit zu dem Ergebnis, dass sich die Systematik aus Kapitel 3.2.4.g mit diesen Ansätzen für eine praktische Nutzung nicht geeignet umsetzten lässt.

3.3.7 Zwischenergebnis

Die Fragestellung dieses Kapitel leitet sich aus der in Kapitel 3.2.5 eröffneten Suche nach einem Modelltransformationsmechanismus ab, der die dort gesammelten Anforderungen erfüllt. Zunächst wurde diesem Thema mit der Untersuchung der *Model Driven Architecture* (MDA) in Kapitel 3.3.1 ein Rahmen gesetzt. Hierbei handelt es sich um einen Ansatz für modellgetriebene Softwareentwicklung, der viele Konzepte rund um Modelltransformationen skizziert und viele Begriffe in diesem Umfeld geprägt hat.

Die Untersuchungen haben ergeben, dass die MDA-Konzepte zu der hier angestrebten Integration von System- und Softwaremodellen passen. So lässt sich der Systembegriff der MDA auf elektronische Systeme im Fahrzeug übertragen und die verlangte Formalisierung für die beteiligten Modelle ist gegeben. Mit Verfolgbarkeitsinformationen und den Transformationsparameter werden hier zudem zwei Punkte berücksichtigt, die bereits in Kapitel 3.1.4 als wichtige Anforderung für Transformation in dieser Domäne erkannt wurden. Die Tatsache dass Muster als ein mögliches Mittel für Modelltransformation genannt werden, passt ebenfalls zu der in Kapitel 3.2.4.g erläuterten Systematik und der in der ROPES-Methode vorgesehen Vorgehensweise.

Für den konzeptionellen Rahmen, der durch die MDA gesetzt wird, gibt es speziell für das Thema der Modelltransformationen zahlreiche Arbeiten, die konkrete Lösungen vorschlagen. Aufbauend auf einer Festlegung wichtiger Grundbegriffe sowie konzeptioneller Unterscheidungsmerkmale und technische Alternativen, wurden die Arbeiten im zurückliegenden

Kapitel vergleichend analysiert. Die Analyse umfasst die beiden letzten Einreichungen für den OMG-Standard (QVT und EXMOF) sowie einige Arbeiten aus dem wissenschaftlichen Umfeld. Einige davon basieren auf der OCL, Attributgrammatiken, logischen Programmiersprachen oder Graphtransformationen.

Die abschließende Bewertung der Ansätze stützt sich auf die Kapitel 3.2.5 gesammelten Anforderungen. Viele dieser Kriterien leiten sich dabei aus der Kernanforderung ab, die auf der Instanziierung von Mustern basierende Systematik aus Kapitel 3.2.4.g abbilden zu können. Der zweistufige Abgleich der Ansätze mit diesen Kriterien liefert schließlich das Ergebnis, dass die Anforderungen von keinem der untersuchten Ansätze im vollen Umfang unterstützt werden. Auch die Prüfung, ob sich die Ansätze, die viele Kriterien bereits erfüllen, entsprechend erweitern lassen, führte zu einem negativen Ergebnis.

3.4 Präzisierte Problemstellung

Die Untersuchungen dieses Kapitels haben die Möglichkeiten und Mängel existierender Ansätze für Modelltransformationen in Bezug auf erarbeitete Anforderungen der Domäne *Automotive Software* aufgezeigt.

Die hier betrachteten Modelltransformationen sollen im Kontext einer stark integrierten System- und Softwareentwicklung eingesetzt werden. Sie verfolgen das Ziel, ein als Analysemodell erfasstes Funktionsnetzwerk automatisiert in ein Grundgerüst eines Software-Designmodells zu überführen. Neben der Möglichkeit, die von der Entwicklungsmethode vorgesehenen Analyse- und Designmodelle verarbeiten zu können, wird eine zentrale Herausforderung darin gesehen, bei der Formulierung von Transformationsregeln die in der Domäne übliche Entwicklungssystematik auf einfache Weise abbilden zu können.

Während die Verarbeitung der relevanten Modelle von vielen der untersuchten Ansätze unterstützt wird, ergeben sich aus dem zweiten Punkt folgend Problemfelder:

1. Für die betrachtete Domäne wurden die in der Referenzmethode ROPES verankerten Muster als Grundlage einer Systematik für den Übergang von Analyse- zu Designmodellen identifiziert. Mit Hilfe einer Ad-hoc-Notation wurde anhand einzelner Beispiele aufgezeigt, wie sich eine Modelltransformation als Instanziierung von Designmustern beschreiben lässt und somit ein etabliertes Instrument der Entwicklungsmethode wiederverwendet wird. Die Umsetzung dieser Systematik verlangt von dem Modelltransformationsmechanismus, die Klassenstruktur eines Musters in Form eines Templates mit Parametern beschreiben zu können und eine Musterinstanziierung durch Bindung der Parameter und Expansion der Klassenstruktur im Zielmodell durchführen zu können. Diese Grundfunktionalität wird zwar von einigen Entwicklungswerkzeugen unterstützt, kann im Rahmen der betrachteten Modelltransformationsansätze aber nicht in dem notwendigen Umfang genutzt werden. Um eine Modelltransformation nach der oben beschriebenen Systematik zu realisieren, muss daher ein neuer Ansatz konzipiert werden.

1. Die Analyse der Anwendungsdomäne hat gezeigt, dass das Softwaredesign – und damit auch die Auswahl der verwendeten Designmuster – stark mit den unterschiedlichen nichtfunktionalen Anforderungen und der technischen Systemarchitektur variiert. Ein Modelltransformationsmechanismus muss diese Variabilität im Zielmodell entsprechend ermöglichen. Viele der untersuchten Ansätze unterstützen dies indirekt über die Möglichkeit, die relevanten Informationen in einem zusätzlichen Eingabemodell abzulegen und die Ausführung einer Transformationsregel dann von einer Bedingung abhängig zu machen, die dieses Modell abfragt. Hier wäre ein noch feinerer Mechanismus wünschenswert, der es erlaubt auf einfache Weise die Instanziierung von Mustern von nichtfunktionalen Eigenschaften oder der technischen Systemarchitektur abhängig zu machen.
2. Das Eingabemodell leitet sich in dem hier betrachteten Kontext aus dem Modell der Systementwicklung ab. Die beschriebene Verzahnung von System- und Softwareentwicklung

in der Domäne macht es notwendig, dass der Zusammenhang zwischen den Modellen der System- und Softwareentwicklung dauerhaft verfolgbar ist. Entsprechend muss ein Modelltransformationsmechanismus Elemente der Ein- und Ausgabemodelle geeignet verlinken können. Die wenigen in den untersuchten Ansätzen enthaltenen Verfolgbarkeitskonzepte beruhen im Wesentlichen auf einer simplen Verlinkung von Elementen aus dem Quell- und Zielmodell oder dem persistenten Festhalten einer Regelanwendung. Hier wäre ein verfeinerter Mechanismus wünschenswert, der auch Designentscheidungen bei Variabilität im Zielmodell verfolgbar und nachvollziehbar macht.

Die Untersuchung der existierenden Ansätze ergab, dass es zurzeit keine Lösung gibt, die die domänenspezifischen Anforderungen und die beschriebene Systematik praxistauglich unterstützt. Aus diesem Grund wird in dem folgenden Kapitel ein eigener Ansatz für Modelltransformationen entwickelt, der die genannten Problemfelder adressiert und insbesondere die in Kapitel 3.2.5 gesammelten Anforderungen erfüllt.

4 Eigener Lösungsansatz POTAD

Die Methodik *Pattern-Oriented Transformations between Analysis and Designmodels* (POTAD) baut auf den im letzten Kapitel vorgestellten Ansätzen auf, erweitert und integriert sie zu einem neuen Ansatz für Modelltransformationen: Die bei [Khriss et al. '99a] (siehe Kapitel 3.3.4.d) und [Judson et al. '03] (siehe Kapitel 3.3.4.e) zu findende Idee, bei Modelltransformationen Designmuster als Konstruktionselement zu nutzen, wird zusammen mit einem erweiterten Template-Konzept der UML zu einem Ansatz kombiniert, der dem Prinzip der imperativen Transformationssprachen folgt. Im Gegensatz zu vielen untersuchten Transformationssprachen, die im Matching-Teil nach Mustern auf Metamodellebene (Ebene M2 in Tabelle 3.4) suchen, wird bei POTAD auf Modellebene (Ebene M1) nach Template-Bindungen eines bestimmten Templates gesucht. Damit gehört POTAD zu den Transformationsansätzen, bei denen das Quellmodell *Marks* enthält, die im Rahmen einer Transformation ausgewertet werden, um Templates zu instanziieren (siehe Kapitel 3.3.1.b).

Der Kernteil besteht aus einer Sprache für Transformationsregeln, die den Übergang von Analyse- zu Designmodellen auf der Basis von Mustern systematisieren und automatisierbar machen.

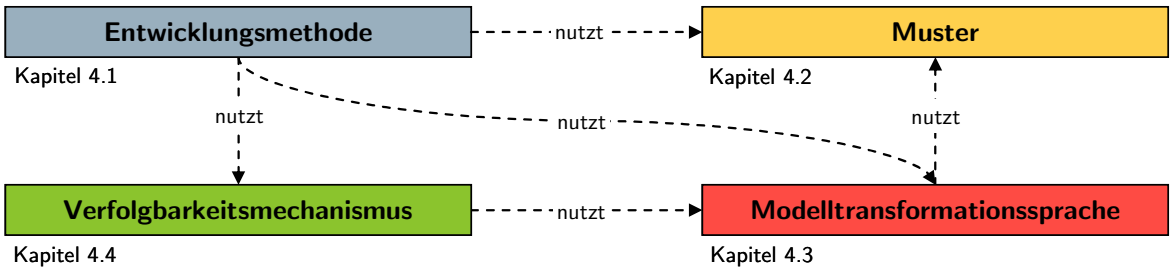


Abbildung 4.1: Die POTAD-Bausteine

Die folgenden Kapitel beschreiben die einzelnen POTAD-Bausteine, wie in Abbildung 4.1 dargestellt. Kapitel 4.1 stellt eine Erweiterung der Softwareentwicklungsmethode ROPES vor, die einerseits auf den Kernprozess der Systementwicklung abgestimmt ist und andererseits die übrigen POTAD-Bausteine in den Entwicklungsprozess integriert. Das darauf folgende Kapitel 4.2 umfasst eine Beschreibung des verwendeten Template-Metamodells, eine eigene Systematik zur Erfassung von Analyse- und Designmustern und einen entsprechenden Musterkatalog, der als Beispiel im Rahmen dieser Arbeit verwendet wird. Die in Kapitel 4.3 beschriebene Modelltransformationssprache ist der Kernteil dieser Arbeit und stellt einen neuen Ansatz für Modelltransformationen dar. Dieser stützt sich auf die zuvor beschriebenen Templates und bildet gleichzeitig die Grundlage für den Mechanismus zur Verfolgung von Designentscheidungen aus Kapitel 4.4. Abbildung 4.2 gibt in der Form eines semiformalen UML-Klassendiagramms einen Überblick über den Zusammenhang der einzelnen POTAD-Artefakte und soll als Orientierungshilfe für die folgenden Unterkapitel dienen.

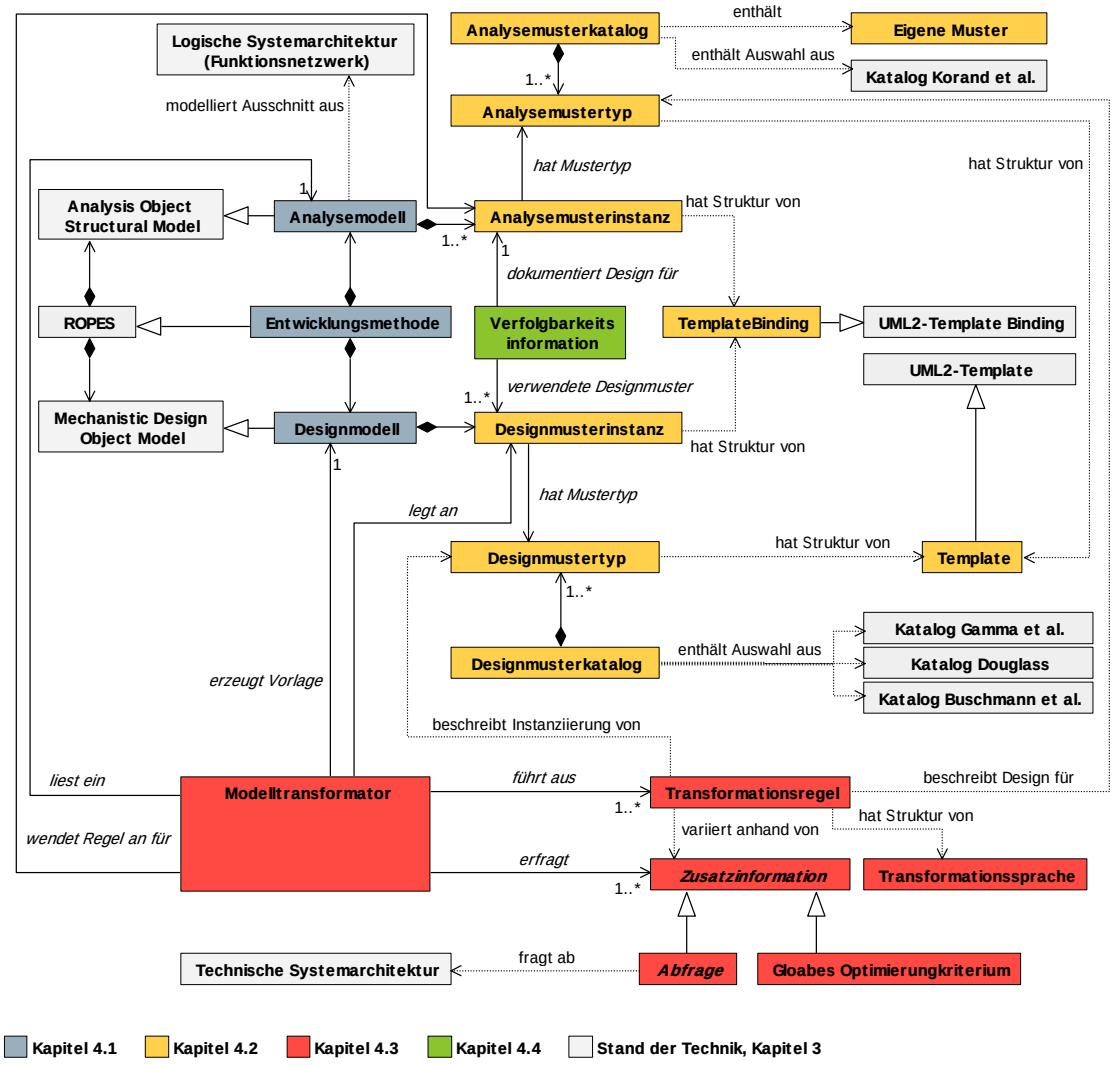


Abbildung 4.2: Die POTAD-Artefakte

4.1 Erweiterung der Entwicklungsmethode

POTAD verwendet eine angepasste Form der in Kapitel 3.2 beschriebenen Softwareentwicklungsmethode ROPES im Bereich Analyse und Design. Die Anpassungen umfassen einerseits die Modifikation bzw. Konkretisierung einiger Aktivitäten und Artefakte, um die Methode in den Kernprozess und die Modelllandschaft der Systementwicklung (siehe Kapitel 3.1.2) zu integrieren. Andererseits werden aber mit den Analysemustern, den im Rahmen des Designs von einem Modelltransformator genutzten Transformationsregeln und den Verfolgbarkeitsinformationen auch völlig neue Artefakte eingeführt. Abbildung 4.3 zeigt die ursprüngliche Übersicht der ROPES-Artefakte aus [Douglass '99], mit den durch POTAD hinzugekommenen Ergänzungen (farblich markiert). Die neuen Artefakte wurden nur mit den hier betrachteten Analyse- und Designphasen in Verbindung gebracht, Pfeile zu den anderen Phasen sind nicht ausgeschlossen.

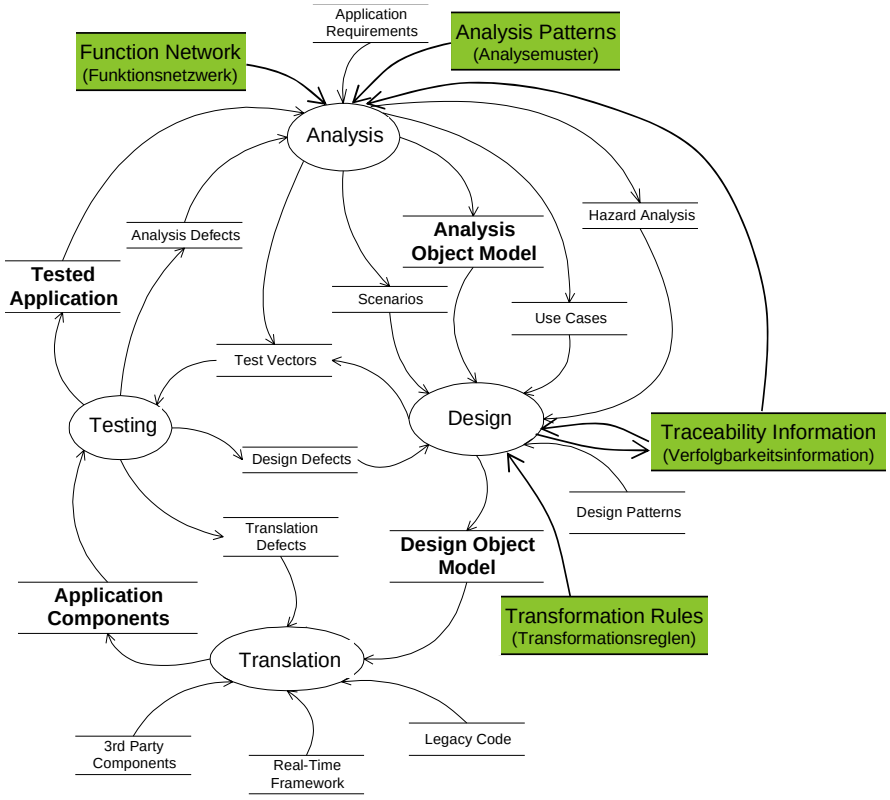


Abbildung 4.3: Die durch POTAD ergänzten Artefakte in der ROPES-Methode

Auf der Ebene der Systementwicklung wird der in Kapitel 3.1.2 erläuterte Kernprozess angenommen, der auf dem V-Modell basiert und weitgehend nach dem Wasserfallprinzip organisiert ist. Im Rahmen der Softwareentwicklung kommt die in Kapitel 3.2.2.a behandelte Variante *SemiSpiral* der ROPES-Methode zum Einsatz, die eine am Spiralmodell orientierte Softwareentwicklung in einen solchen Systementwicklungsprozess einbettet. Es wird von einem, über System- und Softwareentwicklung hinweg, durchgängig modellbasierten Entwicklungsprozess ausgegangen, der die in Kapitel 3.2.2.g skizzierte Modelllandschaft nutzt. Im Rahmen der Systementwicklung werden die in Tabelle 3.1 - Tabelle 3.3 zusammengefassten Inhalte auf Basis der EAST-ADL modelliert, während die Softwareentwicklung für die Modellierung die UML entsprechend der ROPES-Methode nutzt.

Die in Kapitel 3.2.2.g getroffenen Festlegungen beschreiben die Modelllandschaft der Softwareentwicklung etwas konkreter. Demnach übergibt die Systementwicklung ihre Modelle in Form der EAST-ADL. Während die *Functional Analysis Architecture* für die Softwareentwicklung in das *Analysis Object Model* übertragen wird (siehe folgendes Unterkapitel), werden alle Modelle der technischen Systemarchitektur direkt genutzt. Auf eine entsprechende Übertragung in UML-Diagramme wird verzichtet.

Im Rahmen der Softwareentwicklung wird jedoch nur ein Ausschnitt der Systementwicklungsmodelle genutzt, da sich ein Softwareentwicklungsprojekt im Rahmen von POTAD auf die Entwicklung der Software eines Steuergeräts bezieht. Der relevante Teil des Funktionsnetzwerks wird durch Auswertung des der in Tabelle 3.3 aufgeführten Mappings

„Funktion ⇔ Mikroprozessor“ (F2Proz) bestimmt. Zu berücksichtigende Schnittstellen ergeben sich aus den Mappings „Logischer Sensor/Aktuator ⇔ E/E-Komponente“ (SA2Komp) und „Abstraktes Datenelement ⇔ Bussignal/Botschaft“ (DE2Sig).

Das bereits in Kapitel 3.2.5 skizzierte Grobkonzept wird durch POTAD konkretisiert. Der dort in Abbildung 3.50 gezeigte Ablauf wird in Abbildung 4.4 verfeinert und mit den POTAD-Artefakten in Bezug gesetzt. Die einzelnen Aktivitäten werden in den folgenden Unterkapiteln näher erläutert.

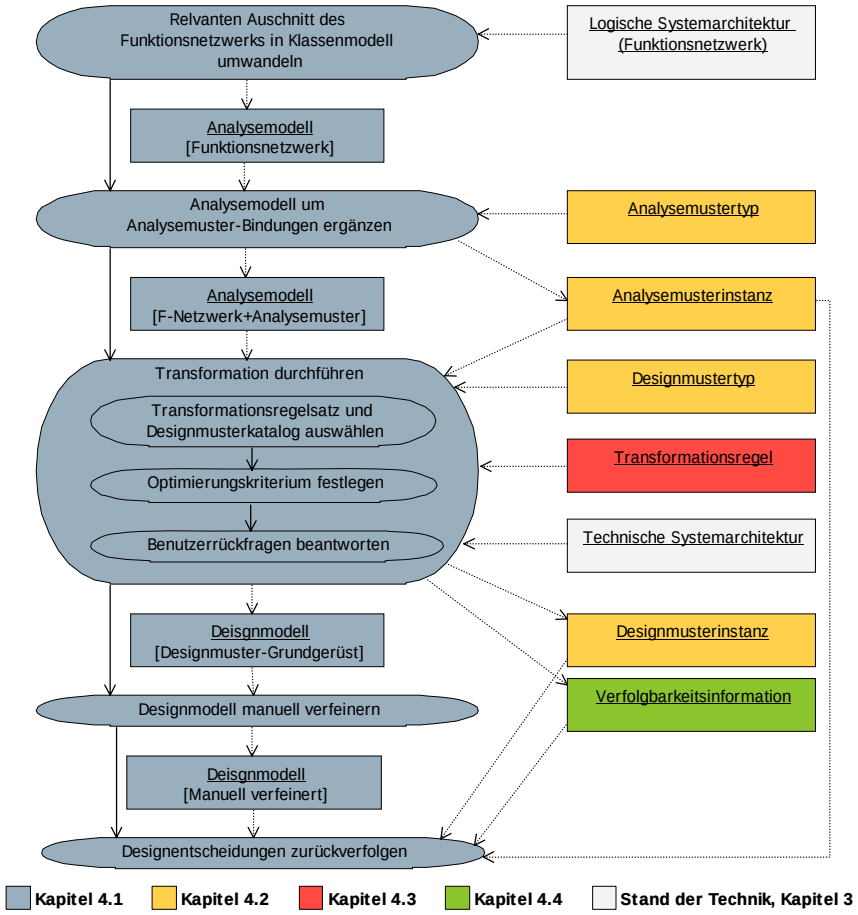


Abbildung 4.4: Typischer Ablauf von Aktivitäten in POTAD

4.1.1 Analyse

Im Rahmen von POTAD wird die Analysephase von ROPES erheblich modifiziert. Unter der bei ROPES bezeichneten Phase *System Engineering* wird die in Kapitel 3.1 erläuterte Systementwicklung verstanden, im Rahmen derer EAST-ADL-Modelle erstellt werden. Die methodischen Vorgaben zur Phase *System Engineering* finden daher keine direkte Anwendung. Entsprechend der Untersuchungsergebnisse aus Kapitel 3.2.2.g, ist die gewählte Vorgehensweise aber weitgehend mit den nur grob formulierten Vorgaben von ROPES kompatibel. Es wird außerdem vorausgesetzt, dass der für das Steuergerät relevante Ausschnitt des Funktionsnetzwerks in Form der EAST-ADL vorliegt. Die hierin enthaltenen umfangreichen und formalisierten

Informationen bzgl. der logischen Systemarchitektur weisen große inhaltliche Überschneidungen mit dem nach ROPES-Vorgabe nun zu erstellenden *Analysis Object Model* auf. Aus diesem Grund wird das Funktionsnetzwerk unter Verwendung der in Kapitel 3.2.1.c getroffenen Konvention in ein Klassenmodell umgewandelt, das als initiales *Analysis Object Strutural Model* verwendet wird. Im Modell werden die Funktionen und logischen Sensoren/Aktuatoren als Klassen modelliert, die über Signale (aus dem UML-Metamodell) miteinander kommunizieren. Die bei der Umwandlung anzuwendenden Abbildungsregeln sind sehr einfach und lassen sich z. B. per Skript realisieren. Die manuelle Erstellung des Analysemodells entfällt dadurch weitgehend, da sich dieses fast vollständig aus den Systementwicklungsmodellen ableitet.

Eine entscheidende Erweiterung der ROPES-Methode erfolgt durch die Einführung von Mustern in das *Analysis Object Model*. Diese in Kapitel 4.2.3.a näher erläuterten Analysemuster müssen zum Ende der Analysephase möglichst umfassend in das Modell in Form von Musterinstanzen eingebracht worden sein. Analysemustertypen und -instanzen basieren dabei auf einem eigenen Template-Mechanismus, der in Kapitel 4.2.1 näher erläutert wird. Als Beispiel eines solchen *Object Analysis Models* kann der obere Teil von Abbildung 3.45 herangezogen werden, in dem das Analysemuster `ClosedLoopControl` für das Funktionsnetzwerk eines Tempomaten gebunden wurde.

Das Einbringen der Analysemuster erfordert Expertise in der Systementwicklung. Wie an dieser Stelle genau die Arbeitsteilung zwischen System- und Softwareentwicklung aussieht, wird hier nicht weiter ausgearbeitet. Da die Analysemuster inhaltlich den Fachbereich der Systementwicklung adressieren, erscheint die Vorstellung nicht abwegig, dass diese Muster schon vor der Übergabe durch die Systementwicklung eingebracht werden. Ein Mustereinsatz auf dem hier notwendigen Formalisierungsniveau entspricht in diesem Bereich zurzeit jedoch nicht dem Stand der Technik.

4.1.2 Design

Die Designphase startet entsprechend der ROPES-Vorgabe mit dem *Architectural Design*. In Übereinstimmung mit [Douglass '04] wird der Einsatz von Modelltransformationen in dieser Phase nicht für sinnvoll gehalten. Für die weitere Diskussion wird eine Standardarchitektur auf der Basis des Architekturmusters *Cyclic Execution* [Douglass '02] angenommen. Dieses Muster beschreibt einen Scheduler, der zyklisch hintereinander die zugewiesenen Methoden von Objekten aufruft, die sich alle in demselben Adressraum befinden. Da es bei diesem in der Automobilindustrie sehr verbreiteten Verfahren keine echte Nebenläufigkeit gibt, müssen auch keine Synchronisierungsmechanismen für den konkurrierenden Zugriff auf Ressourcen vorhanden sein. Die Behandlung von Nebenläufigkeit wird bei der folgenden Diskussion von Designlösungen daher nicht diskutiert (entsprechend der Abgrenzung aus Kapitel 1.3). Bei variierenden Lösungen im *Architectural Design* müssen die nachfolgend erläuterten Transformationsregeln erweitert werden.

Die entscheidende Erweiterung der ROPES-Methode durch POTAD erfolgt bei der Erstellung des *Mechanistic Design Model*. In dieser Phase geht es um die Optimierung einzelner Objekt-

Kollaborationen hinsichtlich der nichtfunktionalen Anforderungen und der technischen Systemarchitektur. Die Modelle haben bereits einen sehr starken Bezug zur letztendlichen Implementierung. Diese bisher manuelle Tätigkeit wird mit Hilfe eines Modelltransformators automatisiert. Der Transformator durchsucht dazu das *Analysis Object Model* nach Analysemusterinstanzen, um für die am Muster beteiligten Elemente eine Designvorlage im *Mechanistic Design* Model zu erzeugen. Dies geschieht durch die Instanziierung von Designmustern, die bereits in der ursprünglichen ROPES-Methode so vorgesehen sind. Im Rahmen von POTAD werden Muster jedoch auf Basis formalisierter Templates eingebracht, die in Kapitel 4.2.1 beschrieben sind.

Welche Designmuster für ein Analysemuster instanziiert werden und wie Informationen aus dem Analysemuster für die Parameterbelegung des Designmusters verwendet werden, ist in Transformationsregeln hinterlegt, die in einer eigens dafür entwickelten Syntax definiert werden. Eine Transformationsregel bezieht sich somit auf einen Analysemustertyp und beschreibt die Instanziierung von einem oder mehreren Designmustern. Vor dem Start einer Transformation kann sowohl der Transformationsregelsatz (alternative Regeln für denselben Analysemustertyp) als auch der Designmusterkatalog ausgewählt werden, dessen Muster bei der Instanziierung verwendet werden. Die Templates in den Designmusterkatalogen müssen namens- und signaturgleich sein, können aber unterschiedliche Klassenmodelle als Lösung enthalten. Der Austausch von Designmusterkatalogen wird z. B. genutzt, um das Design für unterschiedliche Plattformen zu optimieren (siehe Kapitel 6.2).

Die Instanziierung eines Designmusters kann durch einen entsprechenden Ausdruck in der Transformationsregel abhängig von Zusatzinformationen gemacht werden. Eine Zusatzinformation kann ein *Globales Optimierungskriterium* oder eine *Abfrage* sein. Mit dem globalen Optimierungskriterium wird festgelegt, in welche Richtung das Design optimiert werden soll. Entsprechend der Festlegung aus Kapitel 3.2.4.g kann bei dem Start einer Transformation zwischen folgenden Alternativen gewählt werden:

1. Laufzeit
2. Speicherverbrauch
3. Sicherheit und Verfügbarkeit
4. Wartbarkeit, Portabilität, Erweiterbarkeit, Wiederverwendung

Abfragen dienen dagegen zur „lokalen“ Designoptimierung. Sie werden bei einzelnen Musterinstanziierungen verwendet, um z. B. die technische Systemarchitektur abzufragen oder sonstige Fragen für die Instanziierung zu klären, die durch das globale Optimierungskriterium offen bleiben. Abfragen unterteilen sich in maschinelle Abfragen und Benutzerabfragen. Maschinelle Abfragen werden durch einen Interpreter ausgeführt und können z. B. dazu genutzt werden, die EAST-ADL-Modelle abzufragen. Damit könnte folgende für das Design relevante Frage beantwortet werden: „Wird das Signal über einen diskret angeschlossenen Sensor oder über den Bus empfangen?“

Liegen die Informationen nicht maschinenlesbar vor, kann alternativ der Benutzer während der Transformation durch einen Fragedialog zur Klärung der Frage aufgefordert werden. Die in dieser Arbeit gezeigten Transformationsregeln stützen sich ausschließlich auf Benutzerrückfragen, da für die Implementierung der maschinellen Abfrage in der prototypischen Umsetzung (siehe Kapitel 1) der Zeitrahmen nicht ausreichte. Der Aufbau der Transformationsregeln und möglichen Parameter sind ausführlich in Kapitel 4.3 beschrieben.

Nach dem alle Benutzerrückfragen beantwortet sind, erzeugt der Transformator ein aus Designmusterinstanzen bestehendes Grundgerüst des *Mechanistic Design Models*. Dieses muss i. d. R. nachbearbeitet und um Klassen ergänzt werden, die sich nicht aus dem Analysemodell ableiten lassen.

Ein weiteres Ergebnis der Modelltransformation sind die Verfolgbarkeitsinformationen. Dieser in Unterkapitel 4.4 genauer beschriebene Mechanismus verbindet die Designmusterinstanzen während der Transformation mit den Analysemusterinstanzen, aus denen sie hervorgegangen sind. Um die nachfolgende Designaktivität zu unterstützen, muss das eingesetzte Modellierungswerkzeug Abfragen dieser Links für unterschiedliche Kontexte implementieren. Folgende Abfragen der Verfolgbarkeitsinformation sind vorgesehen:

1. Kontext Analysemuster: *Zeige alle Designmuster an, die aus diesem Analysemuster hervorgehen.*
2. Kontext Designmuster: *Zeige das Analysemuster an, aus dem dieses Designmuster hervorgeht.*
3. Kontext Klasse: *Zeige die Muster an, an dem diese Klasse beteiligt ist.*

Für die letzte Abfrage reicht die Auswertung der Musterbindung. Methodisch relevant werden diese Abfragen z. B., wenn die Phase *Mechanistic Design* im Rahmen des Spiralmodells erneut durchlaufen wird. Wird hier eine Modelltransformation erneut ausgeführt, wird das Ergebnis grundsätzlich mit dem Designmodell der letzten Iteration entsprechend der im folgenden Unterkapitel genannten Regeln verschmolzen. Dies funktioniert uneingeschränkt, wenn für die Iteration folgende Bedingungen gelten:

- Das Analysemodell wurde nur erweitert: Im Vergleich zur letzten Iteration gibt es neue Modellelemente und Analysemusterinstanzen. Bei Analysemusterinstanzen, die schon in der letzten Iteration vorhanden waren, sind nur neue tatsächliche Parameter hinzugekommen.
- Das Designmodell wurde nur erweitert: Im Vergleich zur letzten Iteration sind neue Elemente hinzugekommen. Die Namen der durch Modelltransformation erzeugten Elemente wurden nicht geändert. Die durch Modelltransformation erzeugten Elemente können aber neue Kind-Elemente (bei einer Klasse z. B. Attribute oder Operationen) und Beziehungen haben.

Wurde beispielsweise der Name eines tatsächlichen Parameters im Analysemodell oder eines durch Modelltransformation erzeugten Elements im Designmodell geändert, können mit dem in Kapitel 5 vorgestellten Prototyp bei erneuter Ausführung der Transformation „redundante

Elemente“ entstehen. Im Beispiel aus Abbildung 3.45 wird mit der ersten Modelltransformation aus der Klasse `CruiseController` im Analysemodell die Klasse `CruiseController_Master` im Designmodell erzeugt. Wird die Klasse des Analysemodells dann in `CC` unbenannt, entsteht im Designmodell bei einer erneuten Transformation, zusätzlich zu der schon existierenden Klasse `CruiseController_Master`, die Klasse `CC_Master`. Um zu erreichen, dass die Klasse `CruiseController_Master` in `CC_Master` unbenannt wird, müssen die Verfolgbarkeitsinformationen ausgewertet werden. Dies kann vermutlich mit Hilfe der Verfolgbarkeitslinks im Rahmen einer Transformation automatisch geschehen, ist in dem Prototyp aus Kapitel 5 aber nicht implementiert und daher ein Thema für zukünftige Arbeiten (siehe Kapitel 7.1).

Können die oben genannten Bedingungen bei einer Iteration nicht eingehalten werden, ist in der hier vorliegenden Fassung von POTAD die manuelle Neuexpansion der betroffenen Designmuster vorgesehen (wie in Kapitel 3.2.4.f beschrieben). Der Entwickler kann über die bei der letzten Transformation angelegten Verfolgbarkeitsinformationen ermitteln, welche Designmusterinstanzen aus der betreffenden Analysemusterinstanz hervorgegangen sind und Namensänderungen in den Designmuster-Bindungen manuell nachvollziehen. Durch eine Neuexpansion des Musters wird das Designmodell aktualisiert. Der im folgenden Unterkapitel beschriebene Muster-Expansionsmechanismus gewährleistet dabei, dass Musterinstanzen bei geänderter Parameterbelegung neu expandiert werden können und dabei bereits erzeugte Modellfragmente wiederverwendet und mit neu hinzukommenden Modellelementen verschmolzen werden.

Weitere Nutzungsmöglichkeiten der Verfolgbarkeitsinformation werden für die Phasen *Detailed Design* (siehe unten) und *Analysis* gesehen. Das Ermitteln von realisierenden Klassen für einzelne Elemente des Funktionsnetzwerks kann z. B. interessant sein, wenn Änderungen im Funktionsnetzwerk diskutiert werden und die Auswirkungen auf das Design analysiert werden sollen. Evtl. gibt es auch Anwendungsmöglichkeiten für das Testen und weitere hier nicht betrachtete Phasen.

Das *Detailed Design* Model wird grundsätzlich manuell erstellt, wie ursprünglich bei ROPES vorgesehen. In seltenen Fällen kann auch hier das Instrument der Neuexpansion von bereits instanziierten Mustern genutzt werden, um das Design weiter zu verfeinern. Vereinfachungspotenzial bietet POTAD in der *Detailed Design*-Phase durch die erwähnte Möglichkeit Designmusterkataloge, die während der Transformation verwendet werden, auszutauschen. So kann beispielsweise ein Katalog gewählt werden, dessen Muster bereits sehr stark auf eine Plattform optimiert sind, oder eine Alternative, die viele Details des *Detailed Design* noch offen lässt (siehe Kapitel 6.2). In Kapitel 6.2 wird ebenfalls auf die Möglichkeit hingewiesen den Transformationsmechanismus für Transformationen zwischen Designmustern und „*Detailed Design-Mustern”* zu nutzen, dieser Weg wird im Folgenden aber nicht näher beschrieben.

Bis auf die Nutzung der Verfolgbarkeitsinformationen wird der ROPES-Prozess anschließend ohne Änderungen durchlaufen.

4.2 Muster

Im vorangegangenen Unterkapitel wurde bereits auf den breiten Einsatz von Analyse- und Designmustern hingewiesen. Für beide Ebenen wird der Musterbegriff in den folgenden Unterkapiteln präzisiert und ein Beispielkatalog vorgestellt. Für die POTAD-Modelltransformation müssen Mustertypen in einer besonders formalisierten Weise beschrieben und instanziiert werden. Der hierbei verwendete Template-Mechanismus, den die Musterkataloge nutzen, wird im folgenden Unterkapitel beschrieben.

4.2.1 Template-Metamodell

POTAD verwendet eine modifizierte Variante der UML2-Templates. Das entsprechende Paket `Templates` im UML2-Metamodell wird durch das Paket `POTAD_Templates` ersetzt. Abbildung 4.5 zeigt das neue Paket und wichtige Nachbarpakete inklusive der Beziehungen. Das Paket `POTAD_Transformations` enthält das Metamodell der Transformationssprache und wird in Kapitel 4.3 näher erläutert. Der Inhalt des Pakets `POTAD_Traceability` definiert den Verfolgbarkeitsmechanismus zwischen Template-Instanzen und wird in Kapitel 4.4 beschrieben.

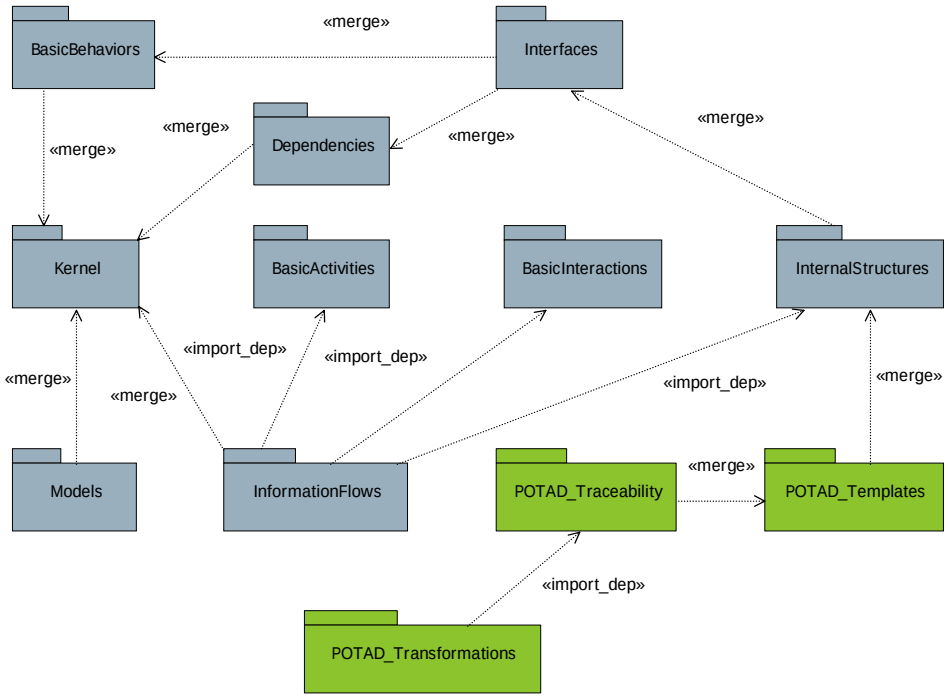


Abbildung 4.5: Die POTAD-Metamodellpakete im Kontext des UML2-Metamodells

Die Modifikationen im Template-Metamodell schränken zum einen die sehr allgemein gehaltenen UML-Templates auf die Fälle ein, die im Rahmen von POTAD verwendet werden und präzisieren für diese die Semantik. Zum anderen werden auch neue Elemente hinzugefügt, um einige der in Kapitel 3.2.4.e identifizierten Lücken zu schließen. Die modifizierten Teile im Bereich der Template-Definition sind in Abbildung 4.6 farblich markiert.

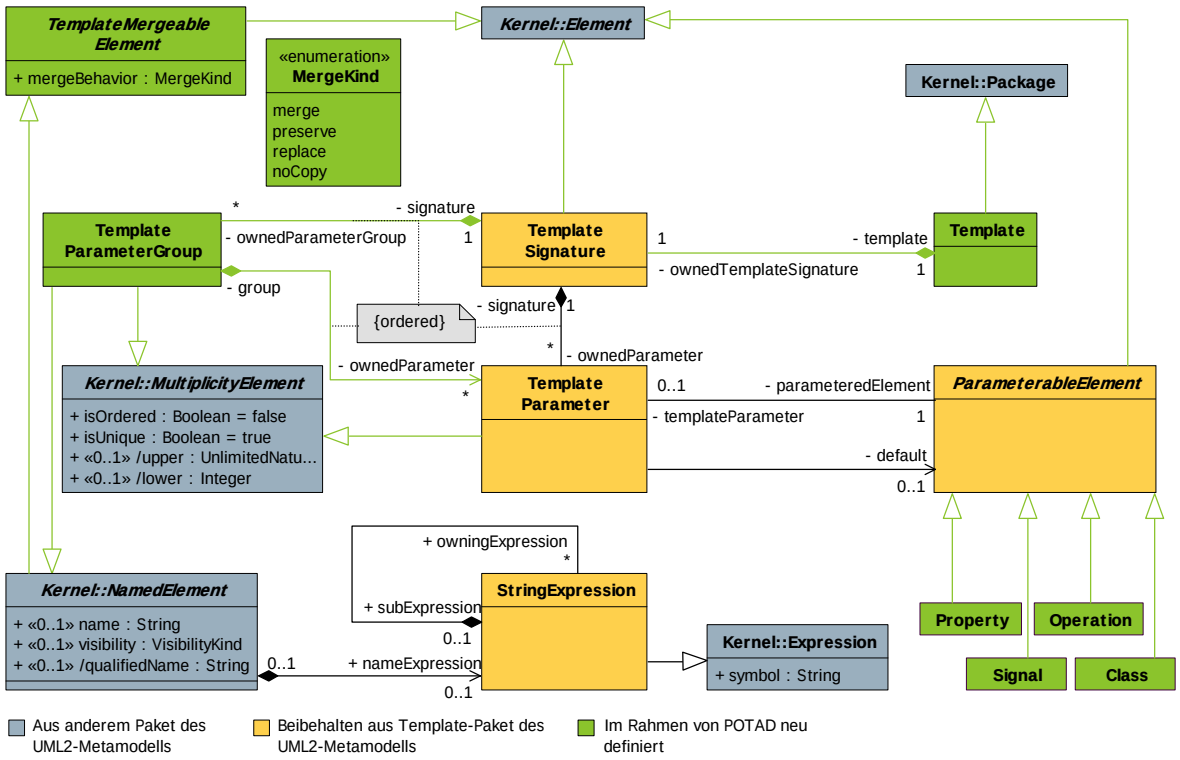


Abbildung 4.6: Modifiziertes Metamodell zur Template-Definition aus der UML2

Die Vererbungshierarchie ist in dem Diagramm nach unten hin abgeschlossen, d. h., es gibt keine weiteren Subklassen. Im Detail wurden folgende Änderungen vorgenommen:

- Indem die abstrakte Klasse `TemplateableElement` durch die konkrete Klasse `Template` ersetzt wird, gibt es nur noch eine Template-Art. `Template` ist das Wurzelelement eines Templates und nimmt alle statischen und parametrisierbaren Elemente eines Templates auf. Die Möglichkeit Elemente aufzunehmen, erhält die Klasse durch die Vererbungsbeziehung zu `Package`. Die Eigenschaft, dass parametrisierbare Elemente eine Kompositionsbeziehung zu `TemplateParameter` haben, wurde entfernt. Demgegenüber wird `TemplateParameter` immer als Kompositionselement von `TemplateSignature` eingebunden.
- Durch die Einführung der abstrakten Klasse `TemplateMergeableElement` kann für jedes Template-Element das Verschmelzungsverhalten gesteuert werden und somit die in Kapitel 3.2.4.e identifizierte Lücke der UML-Templates geschlossen werden. Für die Semantik der vier möglichen Einstellungen `merge`, `preserve`, `replace` und `noCopy` gelten die Erläuterungen in Tabelle 3.10. Die Lösung `TemplateMergeableElement` als Superklasse von `NamedElement` zu modellieren löst das Problem auf einfache, aber radikale Weise, da somit fast alle UML-Elemente über diese Eigenschaft verfügen, auch wenn sie nicht im Kontext von Templates verwendet werden. Bei einer etwaigen Weiterentwicklung sollte hier nach einer Lösung gesucht werden, bei der die Verschmelzungseigenschaft nur bei Template-Elementen vorhanden ist.

- Von ParameterableElement wurden alle Vererbungsbeziehungen zu Subtypen bis auf die zu Signal, Class, Property und Operation entfernt. Somit kann im Rahmen von POTAD ein Template-Parameter nur einen dieser vier Typen haben. In den Beispielen dieser Arbeit werden die Parametertypen Property (in der Form als Klassenattribut) und Operation nur für Templates der Designebene verwendet, der Parametertyp Signal kommt hingegen nur bei Templates der Analyseebene zum Einsatz.
- Durch Einführung einer neuen Vererbungsbeziehung ist nun auch TemplateParameter Subtyp von MultiplicityElement. Damit kann einem Parameter eine Multiplizität zugewiesen werden und die in Kapitel 3.2.4.e identifizierte Lücke geschlossen werden.
- Das Element TemplateParameterGroup wurde neu hinzugefügt. Es realisiert die in Kapitel 3.2.4.e als Lücke identifizierte Gruppierung von Parametern. Eine Gruppe besitzt eine Multiplizität und kann keine weiteren Untergruppen enthalten. Letzteres ist eine komplexitätsreduzierende Einschränkung, die für die hier betrachteten Beispiele keine Nachteile hat, in zukünftigen Arbeiten aber beseitigt werden sollte.
- Die Klasse StringExpression wird über eine Kompositionsbeziehung von NamedElement eingebunden. Von NamedElement erben wiederum viele Elemente, die in einem Template enthalten sein können. Da TemplateableElement entfernt wurde (siehe oben) existiert auch die Vererbungsbeziehung zu StringExpresseion nicht mehr. Das neue Element Template hat die Vererbungsbeziehung nicht übernommen. Es ist somit möglich Stringausdrücke innerhalb eines Templates zu verwenden, ein String-Ausdruck selber kann aber nicht als Template deklariert werden. Die String-Ausdrücke verwenden nicht die im UML-Standard definierte Syntax, sondern die von *Rational XDE* (siehe Kapitel 3.2.4.f).

Abbildung 4.7 zeigt die Modifikation im Bereich der Template-Bindung. Hier sind als Änderung die Einführung der Klassen TemplateParameterGroupSubstitution und TemplateInstance zu nennen. Nachdem mit TemplateParameterGroup die Möglichkeit geschaffen wurde, Parameter bei der Template-Definition zu gruppieren, realisiert TemplateParameterGroupSubstitution diese Gruppenbildung entsprechend bei der Bindung. Da es wie bereits erwähnt nur noch eine Template-Art gibt, wird TemplateableElement durch TemplateInstance ersetzt. Neu hinzugekommen ist außerdem eine Assoziation zwischen TemplateBinding und Package. Das so assoziierte Package nimmt alle nicht parametrisierten Elemente auf und fungiert so als *Root Context* (siehe Kapitel 3.2.4.f). Abschließend wurde die Kompositionsbeziehung zwischen TemplateParameterSubstitution und ParameterableElement entfernt.

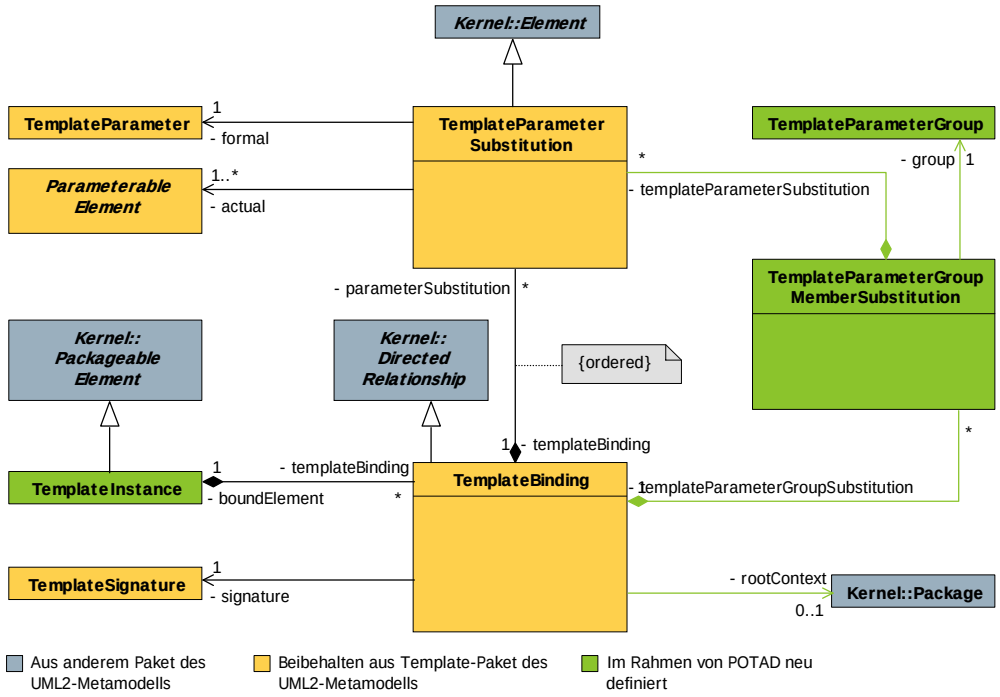


Abbildung 4.7: Modifiziertes Metamodell zur Template-Bindung aus der UML2

4.2.2 Notation und Werkzeugumsetzung

Nachdem das Metamodel und die Semantik der Elemente erläutert wurden, soll nun die entsprechende graphische Notation vorgestellt werden. Diese wurde so gewählt, dass sie auch mit Modellierungswerkzeugen nach dem UML 1.4-Standard dargestellt werden kann. Die Notation ist außerdem stark an die Darstellung von *Patterns* in *Rational XDE* (siehe Kapitel 3.2.4.f) angelehnt, da die prototypische Umsetzung von POTAD (siehe Kapitel 5) auf diesem Werkzeug aufsetzt. Im Folgenden wird bezugnehmend auf die Elemente aus dem Metamodell des vorherigen Unterkapitels die Notation anhand eines Beispiels erläutert.

Abbildung 4.8 zeigt eine Template-Definition am Beispiel des Musters *Strategy* [Gamma et al. '95].

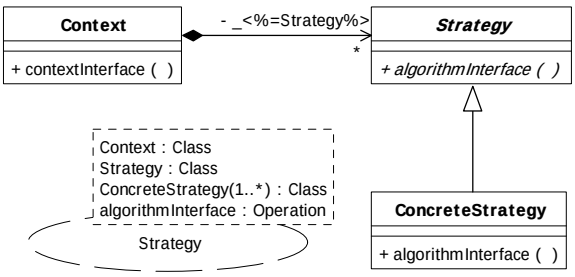


Abbildung 4.8: Eine Template-Definition am Beispiel des *Strategy*-Musters

Ein `Template` wird mit einer gestrichelten Ellipse dargestellt. Dieses Symbol wird in der UML für Kollaborationen verwendet, hat im Kontext von POTAD aber ausschließlich die durch das

Metamodell aus dem vorherigen Unterkapitel beschriebene Semantik. Die TemplateSignature wird an der Ellipse rechts oben als Kasten dargestellt. Es enthält eine geordnete Liste der TemplateParameter in der Form Name : Typ. Die Multiplizität eines Parameters wird in Klammern angegeben, sofern sie nicht Eins ist. Unterhalb der Ellipse ist die Klassenstruktur des Templates zu sehen. In dem Beispiel ist jede Klasse als Parameter ausgewiesen, es wären aber auch weitere statische Klassen möglich. Bei der Expansion wird die hier gezeigte Struktur der Parameterklasse mit der Struktur der tatsächlichen Klassen verschmolzen. Die Notation <%=Parametername%> ist ein Beispiel für StringExpression, der in diesem Fall den Namen des tatsächlichen Template-Parameters zurückgibt.

Abbildung 4.9 zeigt eine mögliche Instanz des *Strategy*-Musters. Eine TemplateInstance wird ebenfalls als Ellipse dargestellt und hat einen eigenen Namen. Das TemplateBinding wird als Abhängigkeitsbeziehung zwischen den Ellipsen von Template und TemplateInstace modelliert. Das Bind()-Kommando repräsentiert die ParameterSubstitution durch eine Liste von Parameterzuweisungen in der Form formal:=actual. Bei der Instanziierung wird die Template-Struktur entsprechend der Verschmelzungseinstellung mit dem Zielmodell verschmolzen. Das Attribut mergeBehavior kann im Werkzeug editiert werden, ist in Diagrammen aber nicht sichtbar. Dies gilt auch für die Eigenschaft rootContext in einem TemplateBinding. Die Standardeinstellung von mergeBehavior ist merge. Sollte davon abgewichen werden, wird dies bei der Erläuterung des entsprechenden Beispiels erwähnt. Der Pfeil am linken oberen Rand der Template-Ellipse ist ein Hinweis darauf, dass dieses Modellelement aus einem externen Modell kommt (in diesem Fall die Musterbibliothek). Dieser Hinweis kann für den Rest dieser Arbeit ignoriert werden.

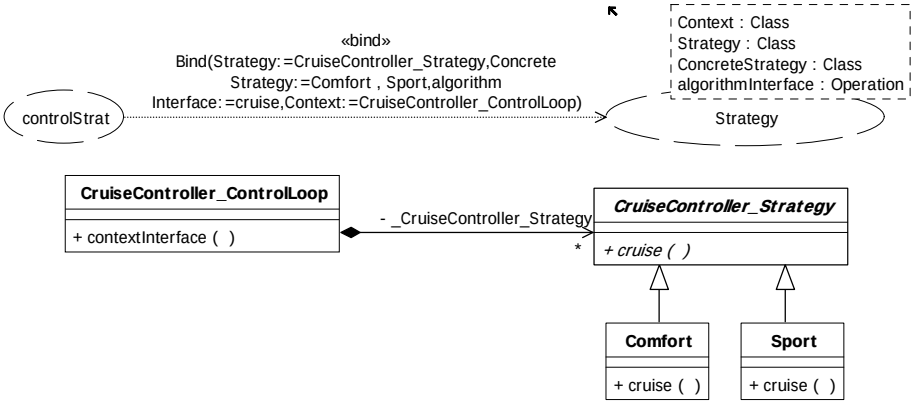


Abbildung 4.9: eine Template-Instanz am Beispiel des *Strategy*-Musters

Das Beispiel in Abbildung 4.10 zeigt die Darstellung von Multiplizitäten und Parametergruppen. Eine TemplateParameterGroup lässt sich durch die Einführung eines Gruppenbezeichners anlegen, der als Präfix dem Template-Parameter vorangestellt wird (getrennt durch einen Punkt). Angaben zur Multiplizität werden in Klammern hinter dem Parameter oder der Gruppe notiert.

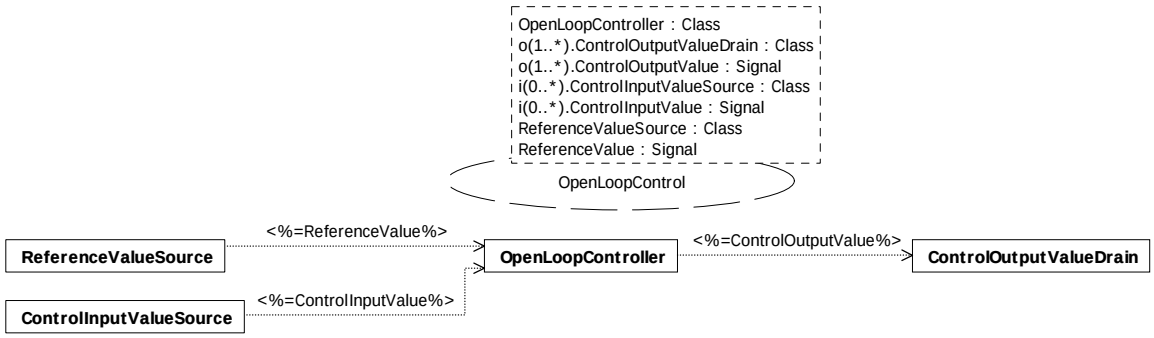


Abbildung 4.10: Beispiel für gruppierte und mit Multiplizitäten versehene Parameter

Im Beispiel werden die Parameter `ControlOutputValueDrain` und `ControlOutputValue` in der mit „o" bezeichneten Gruppe zusammengefasst. Nach den Multiplizitätsangaben muss mindestens eine solche Gruppe angelegt sein und innerhalb dieser Gruppe muss den Parametern `ControlOutputValueDrain` und `ControlOutputValue` jeweils ein Element zugewiesen werden. Bei der Bindung der Parameter wird die Gruppenbildung dadurch realisiert, dass die tatsächlichen Parameter geordnet sind und entsprechend der Gruppenmultiplizität ausgelesen werden. Im Beispiel werden somit zunächst die ersten tatsächlichen Parameter von `ControlOutputValueDrain` und `ControlOutputValue` zu einer Gruppe zusammengefasst, dann die an zweiter Stelle usw. Diese nicht ganz einfach zu lesende Darstellungsweise ist dem Umstand geschuldet, dass *Rational XDE* keine dezidierte Unterstützung von Parametergruppen bietet und somit ein Workaround gefunden werden muss, der die bestehenden Darstellungsmittel nutzt.

4.2.3 Musterkatalog

In den beiden folgenden Unterkapiteln wird jeweils für die Analyse- und die Designphase ein Musterkatalog für die hier betrachtete Domäne vorgestellt. Die Auswahl dieser Muster entstand während der Bearbeitung der Fallstudie durch Literaturrecherche und Expertenbefragung. Der Katalog von Analysemustern enthält einige selbst definierte Muster.

Die hier vorgestellten Kataloge sind als reine Beispiele für die Domäne zu verstehen. Sie wurden nicht durch eine systematische Analyse der Domäne ermittelt, wie z. B. durch eine breit angelegte statistische Untersuchung existierender Systeme. Vielmehr zeichnen sich diese Muster dadurch aus, dass sie im Rahmen der Fallstudie sinnvoll einsetzbar sind und das Potential zur Wiederverwendung in ähnlichen Systemen haben.

Die getroffene Auswahl an Mustern stellt eine gewisse Konsolidierung der vielen Veröffentlichungen in diesem Bereich dar. Da die ausgewählten Muster jedoch bzgl. des Umfang ohne Änderungen von den Autoren übernommen wurden, beseitigt diese Konsolidierung jedoch nicht inhaltliche Überlappungen zwischen den Mustern. So enthält z. B. das Muster *ReactiveControlLoop* implizit des Muster *State* und das Muster *Model-View-Controller* nutzt die Grundidee von *Publisher-Subscriber*. Auch wurde die Einordnung des Musters als Analyse- oder Designmuster von den Autoren übernommen, obwohl sich bei einigen Designmustern von [Douglass '02] die Frage stellt, ob der Abstraktionsgrad des Musters nicht auch zur Analyse-

phase passt (z. B. bei den Mustern `MonitorActuator`, `HomogeneousRedundancy` und `HeterogeneousRedundancy`). Eine Vertiefung dieser offenen Punkte bietet sich als eine mögliche zukünftige Arbeit an (siehe Kapitel 7.1).

4.2.3.a Muster für die Analyse

Die hier gesammelten Muster basieren auf dem in Kapitel 3.2.4.c beschriebenen Begriff für Analysemuster. Entsprechend der hier betrachteten Domäne modelliert ein solches Muster einen wiederholt anzutreffenden Sachverhalt, der bei der Entwicklung von *Automotive Software* in der Analysephase modelliert werden muss. Es bildet komplexe Zusammenhänge aus der Systementwicklung modellhaft nach und verwendet die Sprache des Systementwicklers. Das UML-Modell ist aus Sicht der Software rein konzeptuell und enthält keine Designaspekte. Aus Sicht der Systementwicklung handelt es sich um Muster für die logische Systemarchitektur. Aus dieser Perspektive kann nur eingeschränkt von „Analysemustern“ gesprochen werden, weil auf dieser Ebene die modellierten Konzepte als Realisierungsentscheidung betrachtet werden können.

Die inhaltliche Granularität und die Beschreibungsform orientieren sich an den Mustern von [Konrad et al. '04b]. Die Klassenstruktur des Musters wird allerdings durch die Verwendung des in Kapitel 4.2.1 vorgestellten Template-Konzepts formaler beschrieben. Ein Muster wird konkret mit folgendem Schema erfasst:

- **Name**: Ein aus einem Wort bestehender Bezeichner, der den Mustertyp innerhalb des Katalogs und der Transformationsregeln eindeutig identifiziert.
- **Zweck**: Fasst in einem Satz den Inhalt des Musters unter Verwendung fachspezifischer Kernbegriffe zusammen.
- **Motivation**: Beschreibung des Fachkonzepts, das durch das Muster modelliert wird, in der domänenspezifischen Fachsprache. Üblicherweise werden Beispiele genannt und auf weitere Arbeiten und Standards verwiesen.
- **Struktur und Informationsfluss**: beschreibt die Lösung des Musters in Form eines Templates (siehe Kapitel 4.2.1) unter Verwendung der in Kapitel 4.2.2 gezeigten Notation. Ein Template umfasst bei Analysemustern ein Modell aus Klassen, die über Signale miteinander kommunizieren. Einzelne Klassen und oder Signale können als Template-Parameter ausgewiesen sein. Bei dem dargestellten Diagramm werden die Signalbeziehungen entsprechend der in Kapitel 3.2.1.c getroffenen Konvention in Form von *Information Flows* modelliert. Für jedes Template-Element werden in einem Satz die Aufgabe und der Informationsaustausch mit anderen Template-Elementen erläutert.
- **Beispiel**: Ein Diagramm mit erläuterndem Text, das eine Instanz des Musters für ein Beispiel enthält.
- **Anwendbare Designmuster**: Eine Liste von Designmustern, die bei der Realisierung dieses Analysemusters zum Einsatz kommen können. In Klammern wird vermerkt, zu welchem Optimierungskriterium das Designmuster passt.

Der mit diesem Schema beschriebene Umfang wird für den Umgang mit Transformationssregeln als ausreichend betrachtet. Für eine noch ausführlichere Beschreibung eines Analysemusters wird das erweiterte Schema von [Konrad et al. '04b] empfohlen.

Der Analysemusterkatalog findet sich in Anhang A.1. Tabelle 4.1 gibt einen Überblick über die im Katalog befindlichen Muster.

Name	**Zweck**	Seite
ClosedLoopControl	Einen geschlossenen Regelkreis (engl. *feedback control*) beschreiben.	193
OpenLoopControl	Eine Steuerkette (engl. *feedforward control*) beschreiben.	195
UserInterface	Abstrakte Beschreibung einer Benutzerschnittstelle.	198
FaultHandler	Beschreibt eine hierarchische Fehlerbehandlung, bei der eine globale Fehlerbehandlungskomponente mehrere lokale Fehlerbehandlungskomponenten koordiniert.	200
DetectorCorrector	Beschreibt ein generisches Überwachungskonzept, das korrigierende Maßnahmen initiiert, wenn sich das System oder einzelne Komponenten nicht mehr in einem gültigen Zustand befinden.	201

Tabelle 4.1: Muster des Analysemusterkatalogs

Die beiden Muster *ClosedLoopControl* und *OpenLoopControl* finden in der Fallstudie die häufigste Verwendung. Sie beschreiben den charakteristischen Aufbau einer Regelung und einer Steuerung, so wie er auf der Ebene der Systementwicklung betrachtet wird. Beide Muster wurden auf Basis der Beschreibung DIN 19226-1 [DIN '94] selbst definiert.

Die Muster *UserInterface*, *FaultHandler* und *DetectorCorrector* spiegeln ebenfalls Konzepte wieder, die auf der Ebene der Systementwicklung festgelegt werden. Diese drei Muster sind eigene Weiterentwicklungen bzw. Präzisierungen von [Konrad et al. '04b].

4.2.3.b Muster für das Design

Designmuster beschreiben Lösungsschablonen für wiederkehrende Designprobleme. Der im Rahmen dieser Arbeiten verwendete Musterbegriff für das Design lehnt sich an [Gamma et al. '95] an. Designmuster sind im Gegensatz zu Analysemustern umfangreich publiziert. Die im Anhang A.2 erfasst Designmuster verwenden folgendes Schema:

- **Name**: Ein aus einem Wort bestehender Bezeichner, der den Mustertyp innerhalb des Katalogs und der Transformationsregeln eindeutig identifiziert.
- **Zweck**: Fasst in einem Satz das Designziel des Musters zusammen.

- **Struktur**: Beschreibt die Lösung des Musters in Form eines Templates (siehe Kapitel 4.2.1) unter Verwendung der in Kapitel 4.2.2 gezeigten Notation. Ein Template bei Designmustern umfasst ein Klassenmodell, das einzelne Klassen, Operationen oder Attribute als Parameter ausweisen kann.
- **Erläuterung**: Beschreibung des Musters in Textform im Kontext der Domäne.

Dieses im Vergleich zu [Gamma et al. '95] reduzierte Schema wurde gewählt, da diese Muster in ausführlicher Form publiziert sind und daher im Rahmen dieser Arbeit nicht detailliert dokumentiert werden müssen. Der Schwerpunkt des Schemas liegt somit in der formalisierten Beschreibung der Klassenstruktur als Template. Die im Designmusterkatalog enthaltenen Muster sind in Tabelle 4.2 aufgelistet. In den rechten Spalten ist eine Bewertung der Muster hinsichtlich der in POTAD betrachteten nichtfunktionalen Anforderungen zu sehen. Diese Bewertung wurde als Grundlage für die Erstellung der Transformationsregeln verwendet, ist jedoch nicht das Ergebnis einer systematischen Evaluierung. Sie stützt sich hauptsächlich auf Erfahrungen und Einschätzungen, die im Rahmen einer an POTAD beteiligten Diplomarbeit gewonnen wurden (siehe Kapitel 6.2).

Grundsätzlich wurde bei der Bewertung eines Kriteriums als Referenz ein Design mit möglichst wenigen Objekten herangezogen, in dem Kommunikation zwischen Objekten über öffentliche Attribute realisiert wird und komplexe Algorithmen in einer Operation unter intensiver Nutzung von Kontrollstrukturen implementiert sind. Entstehen durch die Anwendung eines Designmusters redundante oder aus funktionaler Sicht überflüssige Daten, wird der Speicherverbrauch mit „–“ bewertet. Die Laufzeit wird mit „–“ bewertet, wenn neue Iterationen oder Suchaktivitäten notwendig werden. Kommen durch das Muster nur wenige Anweisungen ohne Schleifen hinzu, wird dies mit „0“ bewertet. Diese Regeln können jedoch nur als grobe Orientierung verstanden werden.

Muster / Bewertung hinsichtlich nichtfunktionaler Anforderungen			Laufzeit	Speicherverbrauch	Sicherheit/Verfügbarkeit	Wart/Port/Erw/Wied[1]
Name	**Zweck**	Seite				
Blackboard	Verteilt erzeugte Daten zentral in einem Objekt kapseln und über eine generische Schnittstelle lesen und schreiben.	210	–	–	0	+
ControlLoop	Den Algorithmus einer Regelung-/Steuerung in einem Objekt kapseln.	211	0	0	0	+
DataBus	Den Zugriff auf einen Datenbus in einer einheitlichen Objektstruktur kapseln.	212	–	–	0	+
Decorator	Ein Objekt um Zuständigkeiten erweitern.	213	–	0	0	+
ExceptionMonitor	Fehler auf der Ebene einer Objektkollaboration erkennen und behandeln.	214	–	–	+	–
HeterogeneousRedundancy	Heterogene Redundanz in einem Objektmodell abbilden.	215	–	–	+	–
HomogeneousRedundancy	Homogene Redundanz in einem Objektmodell abbilden.	216	–	–	+	–
Model-View-Controller (ModelViewContr)	Modell, Darstellung und Steuerung einer interaktiven Anwendung in separaten Objekten kapseln.	217	–	–	0	+
MonitorActuator	Ein Überwachungskonzept eines Aktuators in einem Objektmodell abbilden.	218	–	–	+	–
Publisher-Subscriber (PubSubscriber)	Zustandsänderungen eines Objekts unidirektional an registrierte Abonnenten schicken.	219	–	–	0	+
ReactiveControlLoop	Algorithmen für unterschiedliche Zustände einer Regelung-/Steuerung in separaten Objekten kapseln.	219	0	–	0	+
Repository	Verteilt erzeugte Daten zentral in einem Objekt kapseln.	220	0	–	0	+

Tabelle 4.2 (Teil 1): Muster des Designmusterkatalogs

Muster \ Bewertung hinsichtlich nichtfunktionaler Anforderungen			Laufzeit	Speicherverbrauch	Sicherheit/Verfügbarkeit	Wart/Port/Erw/Wied[1]
Name	**Zweck**	Seite				
SecondGuess	Eine Berechnung durch unterschiedliche Algorithmen absichern und diese in separaten Objekten kapseln.	221	–	–	+	–
Singleton	Sicherstellen, dass es von einer Klasse nur eine Instanz gibt und diese global zur Verfügung stellen.	221	0	0	+	0
State	Verhalten für unterschiedliche Zustände in eigenen Objekten kapseln.	222	0	–	0	+
Strategy	Eine Familie von austauschbaren Algorithmen definieren und in separaten Objekten kapseln.	223	0	–	0	+
TemplateMethod	Das Skelett eines Algorithmus definieren und einzelne Rechenschritte in separate Objekte auslagern.	223	0	–	0	+
Watchdog	Eine Zeitüberwachung in einem Objekt kapseln.	224	–	–	+	–

+	Muster wirkt positiv auf diese Anforderung	[1]Wartbarkeit, Portabilität, Erweiterbarkeit, Wiederverwendung
0	Muster wirkt neutral auf diese Anforderung	
–	Muster wirkt negativ auf diese Anforderung	

Tabelle 4.2 (Teil 2): Muster des Designmusterkatalogs (Teil 2)

4.3 Modelltransformationssprache

Ziel der in diesem Kapitel beschriebenen Transformationsregeln ist es, für ein Analysemuster in Abhängigkeit von nichtfunktionalen Anforderungen und der technischen Systemarchitektur durch die Instanziierung von Designmustern eine Designvorlage zu erzeugen. Grundlage der Lösung ist die bereits in Kapitel 3.2.4.g beschriebene Systematik und das in Kapitel 3.2.5 skizzierte Grobkonzept mit den dort ebenfalls gesammelten Anforderungen. Der im Folgenden vorgestellte Ansatz für Transformationsregeln greift die in Abbildung 3.48 skizzierte Grundidee auf, die Parameterabhängigkeiten zwischen Analyse- und Designmustern zu beschreiben.

In Anlehnung an die Vorgehensweise vieler Vorschläge zu QVT (siehe Kapitel 3.3.3) wurde die entwickelte Transformationssprache über ein Metamodell definiert, das die abstrakte Syntax der Transformationsregeln beschreibt. In den Erläuterungen zu den einzelnen Klassen aus diesem

Metamodell wird dann die statische und prozedurale Semantik in natürlicher Sprache beschrieben. Die prozedurale Semantik liefert somit eine informelle Beschreibung des Mechanismus, der umgesetzt werden muss, um die Transformationsregeln auszuwerten.

4.3.1 Metamodell

Abbildung 4.11 zeigt das Metamodell der POTAD-Transformationssprache. Es ist in dem Paket `POTAD_Transformations` enthalten, das durch die in Abbildung 4.5 gezeigten Beziehungen mit dem UML2-Metamodell integriert ist.

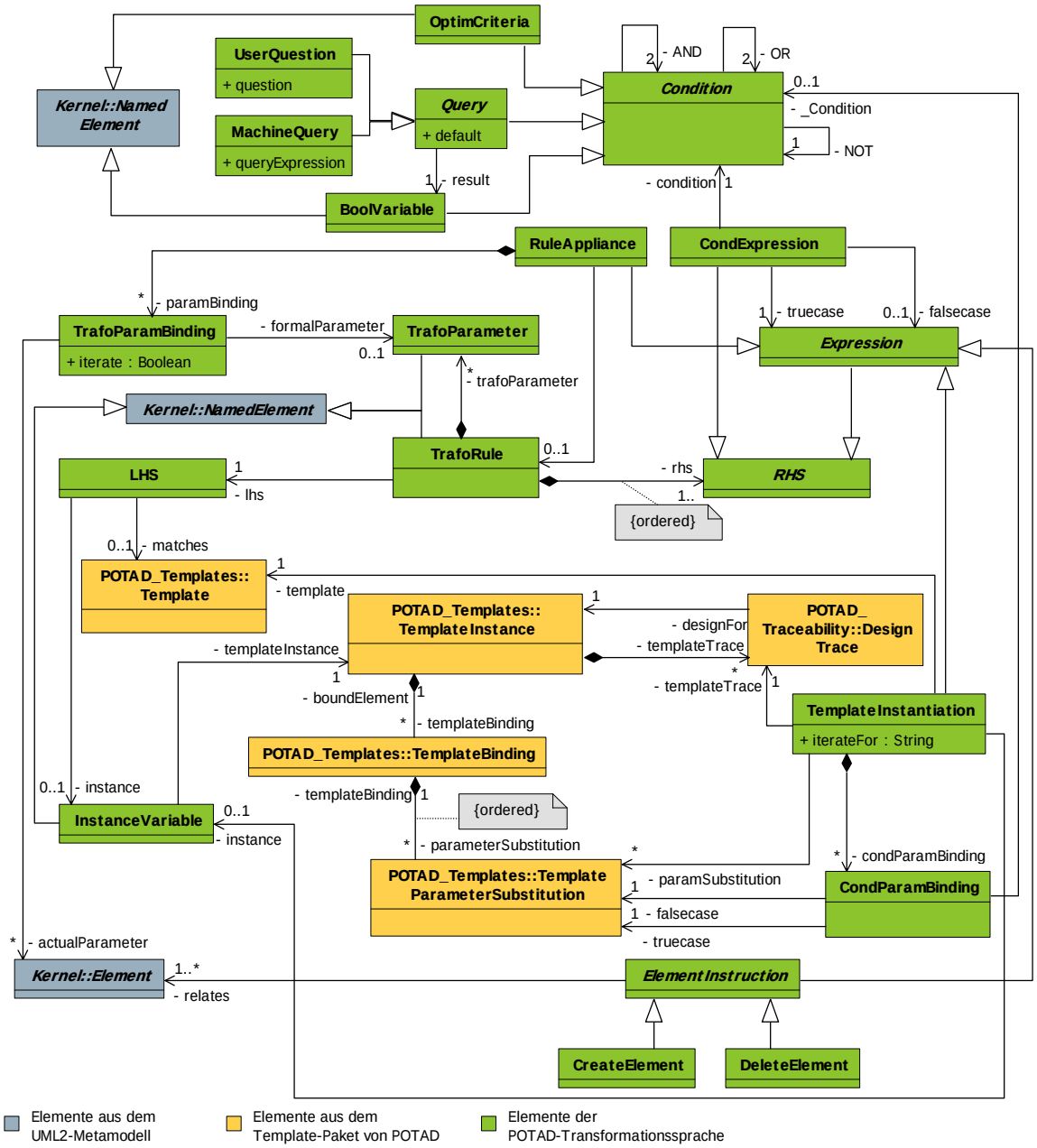


Abbildung 4.11: Metamodell für die Transformationsregeln

Im Folgenden wird die Semantik der Klassen aus dem Metamodell in natürlicher Sprache beschrieben.

TrafoRule (Transformation Rule)

Steht für eine komplette Transformationsregel. Diese hat einen Namen und besteht aus einer linken Seite (`LHS` für „left hand side“) und einer (mindestens ein Element enthaltenden) geordneten Liste von Elementen der rechten Seite (`RHS` für „right hand side“). Optional kann die Transformationsregel beliebig viele Transformationsparameter (`TrafoParameter`) haben.

TrafoParameter (Transformation Parameter)

Bezeichnet einen nicht typisierten Parameter für eine Transformationsregel, der über einen eindeutigen Namen identifiziert wird und über diesen in der gesamten Regel angesprochen werden kann. Wird ein Parameter definiert, so muss dieser auch beim Aufruf belegt werden, optionale Parameter mit vordefinierten Werten sind nicht vorgesehen.

LHS (Left Hand Side)

Legt die Ausgangslage für die Anwendung einer Transformationsregel fest. Dafür gibt es zwei Möglichkeiten:

1. Die linke Seite enthält kein Element. In diesem Fall handelt es sich um eine Hilfsregel, die mindestens einen Parameter besitzen muss und bei Aufruf aus einer anderen Regel immer ausgeführt wird.
2. Die linke Seite verweist auf ein `Template` (aus dem Template-Metamodell, siehe Kapitel 4.2.1) und eine `InstanceVariable`. Tritt eine `TemplateInstance` des angegebenen Templates im Analysemodell auf, so wird die angegebene `InstanceVariable` mit dieser belegt und die rechte Seite der Regel ausgeführt. Die Templateinstanz aus dem Analysemodell kann im gesamten rechten Teil der Regel über die `InstanceVariable` angesprochen werden, um die Belegungen ihrer Parameter aufzulösen.

RHS (Right Hand Side)

Die rechte Seite dient als Abstraktion für ein einzelnes Ausführungselement einer Regel. Sie kann entweder durch einen bedingten Ausdruck (`CondExpression`) oder einen unbedingten Ausdruck (`Expression`) realisiert werden.

CondExpression (Conditional Expression)

Stellt einen bedingten Ausdruck dar. Verweist auf eine Bedingung (`Condition`), einen obligatorischen Ausdruck (`Expression`) für den Fall, dass die Bedingung erfüllt ist sowie einen optionalen Ausdruck für den Fall, dass die Bedingung nicht erfüllt ist.

Expression

Dient als Abstraktion der verschiedenen Möglichkeiten für Ausdrücke auf der rechten Seite (`RuleAppliance`, `TemplateInstantiation` und `ElementInstruction`).

Condition

Dient als Abstraktion für die konkreten Bedingungen `OptimCriteria`, `MachineQuery`, `UserQuestion` und `BoolVariable`. Eine `Condition` kann zwei weitere Bedingungen mit `AND` und `OR` logisch verknüpfen sowie eine andere `Condition` mit `NOT` negieren.

OptimCriteria (Optimisation Criteria)

Stellt eine globale boolesche Variable für eine nichtfunktionale Anforderung dar, die die Transformationsregel von außen vorgegeben bekommt und wird über den Namen angesprochen.

Query

Abstraktion für eine Abfrage. Eine abgeleitete Abfrage liefert einen booleschen Wert zurück, der in der Variable `BoolVariable` gespeichert wird. Für den Fall, dass die Abfrage nicht durchgeführt werden kann, kann ein Standardwert vorgegeben werden.

MachineQuery

Eine Abfrage, die maschinell durchgeführt wird. In dem Attribut `queryExpression` kann eine maschinenlesbare Abfrage gespeichert werden, die den geforderten booleschen Wert zurückliefert. Mit dieser Abfrage werden insbesondere die Modelle der technischen Systemarchitektur abgefragt. Die Syntax von `queryExpression` kann implementierungsspezifisch festgelegt werden. Die Implementierung muss einen entsprechenden Interpreter zur Verfügung stellen.

UserQuestion

Eine Benutzerabfrage, die vom Benutzer während der Transformation beantwortet wird. Die in `question` gespeicherte Frage muss mit „Ja" oder „Nein" beantwortet werden.

BoolVariable

Speichert die Antwort einer `Query` für die restliche Dauer der Ausführung der Regel zur späteren Wiederverwendung in einer Bedingung und wird über den Namen angesprochen.

InstanceVariable

Kennzeichnet eine bestimmte `TemplateInstance` (aus dem Template-Metamodell, siehe Kapitel 4.2.1), so dass diese innerhalb einer Regel über den Namen angesprochen werden kann.

TemplateInstantiation

Stellt eine mögliche Variante einer `Expression` dar und kennzeichnet die Erzeugung einer neuen Template-Instanz. Dazu wird das `Template` sowie eine Menge von `TemplateParameterSubstiutions` (beide aus dem Template-Metamodell, siehe Kapitel 4.2.1) angegeben, wobei die `TemplateParameterSubstiutions` sich auf alle notwendigen Parameter des angegeben Templates beziehen und mit diesen bezüglich Typ und Multiplizität vereinbar sein müssen. Einzelne Parameter können auch mit bedingter Verzweigung belegt werden (siehe `ConcdParamBinding`). Bei der Erzeugung wird eine `TemplateInstance` des angegebenen Templates expandiert, indem die Para-

meterersetzung getreu der Bindung vorgenommen und die Struktur mit dem Zielmodell verschmolzen wird. Zusätzlich wird eine Verfolgbarkeitsbeziehung von der neuen Instanz zur Instanz des Analysemusters auf der linken Seite erzeugt, für die die Regel ausgeführt wird. Falls es sich um einen bedingten Ausdruck handelt, werden mit dieser Beziehung auch die Bedingungen festgehalten. Optional kann mit der `TemplateInstantiation` auch eine `InstanceVariable` definiert werden, die für den restlichen Teil der Regel als Kennzeichner der neu angelegten Template-Instanz dient. Mit dem Attribut `iterateFor` wird gegebenenfalls der Name eines mengenwertigen Parameters gekennzeichnet, über den iteriert werden soll. Bei Belegung des gekennzeichneten Parameters mit einer Menge wird das Muster dann für jedes Element aus dieser Menge einzeln erzeugt.

CondParamBinding (Conditional Parameter Binding)

Bezieht sich auf die Erzeugung einer Template-Instanz (`TemplateInstantiation`) und steht für eine bedingte Belegung von Parametern. Beinhaltet eine Bedingung (`Condition`) sowie jeweils ein `TemplateParameterSubstitution` für die beiden möglichen Fälle.

ElementInstruction

Stellt eine mögliche Variante einer `Expression` dar und kennzeichnet eine Operation, die sich auf eine Menge von Designmodellelementen (`Element`, aus dem UML-Metamodell) bezieht. Zu beachten ist, dass die `ElementInstruction` nur Elemente im Designmodell betrifft. Somit kommen als mögliche Elemente Klassen (`Class`), Operationen (`Operation`), Attribute (`Property`), Assoziationen (`Association`), Vererbungsbeziehungen (`Generalization`) und Realisierungsbeziehungen (`Realization`) in Frage. Die Klasse dient als Abstraktion für die konkreten Operationen `CreateElement` und `DeleteElement`.

CreateElement

Steht für eine Operation, die die assoziierte Menge von Modellelementen im Designmodell erzeugt.

DeleteElement

Steht für eine Operation, die die assoziierte Menge von Modellelementen im Designmodell löscht.

RuleAppliance

Stellt eine mögliche Variante einer `Expression` dar und kennzeichnet den Aufruf einer Hilfsregel. Sie enthält einen Verweis auf die aufzurufende Regel (`TrafoRule`) sowie eine Menge von `TrafoParamBindings`, die alle Parameter der aufzurufenden Regel mit Mengen von Modellelementen belegen.

TrafoParamBinding (Transformation Rule Parameter Binding)

Bezieht sich auf den Aufruf einer Transformationsregel (RuleAppliance) und stellt eine Verknüpfung zwischen einem TrafoParameter aus der zugehörigen Regel (TrafoRule) und einer Menge von Modellelementen (Element) her. Bei Verwendung des TrafoParameters auf der rechten Seite einer Regel wird dieser durch die über das TrafoParamBinding angegebene Menge ersetzt. Mit dem Attribut iterate wird festgelegt, ob bei einer Belegung des Parameters mit einer Menge von Elementen über diese iteriert, also die zugehörige Hilfsregel für alle Elemente aus der Menge ausgeführt werden soll.

4.3.2 Offene Punkte

Das Metamodell der Transformationssprache regelt nicht jedes Detail. So bleibt die Frage offen, wie die Erzeugung von Elementen im Detail formuliert werden soll. Hier ist zum einen entscheidend, ob man nur das Element an sich (z. B. nur eine Klasse) oder auch bereits abhängige Elemente (z. B. eine Klasse inklusive aller ihrer Attribute und Operationen) erzeugen will. Zum anderen ergibt sich das Problem, dass für neu erzeugte Elemente im Designmodell, die keine Grundlage im Analysemodell haben, Namen gefunden werden müssen. Dafür gibt es verschiedene Möglichkeiten, die je nach Anwendungsfall Vor- und Nachteile haben:

- Namen für erzeugte Elemente werden grundsätzlich in den Transformationsregeln fest vorgegeben. Dies führt zwar meist zu passenden Bezeichnungen, kann bei mehrfacher Anwendung derselben Regel oder bei Iterationen aber zu Problemen führen, wenn mehrmals Elemente gleichen Namens erzeugt werden, die eigentlich unterschiedliche Identitäten besitzen müssten.
- Namen für erzeugte Elemente werden auf Basis eines automatischen Schemas generiert, z. B. <Name der Template-Instanz>_<Parametername>. Sofern jede Template-Instanz über einen Namespacing-Mechanismus einen eigenen Namen erhält, vermeidet dies Widersprüche, macht es aber in den meisten Fällen notwendig, die Namen der Elemente später anzupassen.
- Der Benutzer der Transformation erhält die Möglichkeit, Namen für neu erzeugte Elemente über die Benutzeroberfläche einzugeben. Dies ermöglicht sinnvolle Namen, kann aber bei missbräuchlicher Nutzung oder Fehlern durch den Benutzer zu Widersprüchen führen, wenn dieser z. B. den Namen eines Elementes eingibt, das in einem anderen Kontext bereits existiert.

Das Metamodell beschreibt in diesem Punkt also nur das abstrakte Konzept der Erzeugung eines Elements und überlässt die konkrete Ausgestaltung der praktischen Realisierung des Mechanismus.

4.3.3 Konkrete Syntax

Im vorigen Abschnitt wurde das abstrakte Konzept der Transformationssprache beschrieben. Für eine Umsetzung in die Praxis bedarf es allerdings einer konkreten Schreibweise, in der die in

der abstrakten Syntax beschriebenen Konzepte ausgedrückt werden können. Wie schon in Kapitel 3.3 gesehen, gibt es hierfür verschiedene Möglichkeiten, die sich in erster Linie auf graphische und textuelle Notationen aufteilen. Im Folgenden wird eine textuelle Notation vorgestellt, die im weiteren Verlauf dieser Arbeit verwendet wird.

Die erarbeitete textuelle Notation versucht, die Konzepte aus dem Metamodell möglichst direkt in Schlüsselwörter oder -konstrukte einer Art Programmiersprache umzusetzen. Die in den offenen Punkten ausgeführten Gestaltungsmöglichkeiten bei Konstruktoren für Elemente und bei der Namenswahl wurden dabei bereits konkretisiert. Die entstandene Sprache wird nachfolgend zunächst informell beschrieben, um einen Bezug zur abstrakten Syntax und damit zur Semantik herzustellen. Anschließend wird die Grammatik der Sprache in der bei kontextfreien Sprachen üblichen EBNF (*Extended-Backus-Naur-Form*) dargestellt.

4.3.3.a Informelle Erläuterung

Transformationsregeln bestehen in der textuellen Syntax aus folgendem Schema:

```
Regelname(Regelparameter):Linke Seite -> Rechte Seite
```

Bei Regeln ohne Parameter, die bei Vorkommen einer Template-Instanz eines bestimmten Typs ausgeführt werden, wird die linke Seite mit

```
Template <Template> <Variablenname>
```

notiert, bei den Hilfsregeln bleibt sie bekanntermaßen leer. Über <Variablenname>.<Parametername> können dann auf der rechten Seite die an die einzelnen Parameter gebundenen Elemente aufgelöst werden. Die Erzeugung einer Template-Instanz auf der rechten Seite wird mit dem Ausdruck

```
Template <Template>(<Argument_1>,...,<Argument_n>) <Variablenname>
```

erreicht, wobei der Variablenname optional ist. Dies setzt voraus, dass die vorgegebene Reihenfolge der Parameter eingehalten wird. Bei der Auswertung wird für jedes Argument ein Element im Designmodell erzeugt, das den Typ des zugeordneten Parameters und den Namen des Argumentes erhält. Existiert ein solches Element bereits, so wird das bestehende Element verwendet. Mit diesen Elementen werden dann die Musterexpansion und das Anlegen der Bindungen im Designmodell vorgenommen. Einzelne Ausdrücke auf der rechten Seite werden kommasepariert hintereinander aufgelistet. Für die übrigen Modellelemente gibt es Konstruktoren.

```
Class(<Name>,<Boolescher Wert für die Eigenschaft abstrakt>)
Attribute(<Name der Klasse>,<Name des Attributs>,<Typ>)
Operation(<Name der Klasse>,<Name der Operation>,<Rückgabetyp>)
Association(<Name der Quellklasse>,<Name der Zielklasse>,<gerichtet>)
Inheritance(<Name der Unterklasse>,<Name der Oberklasse>)
```

Bedingte Verzweigungen werden nach dem Schema

```
 [<Bedingung>]? <Erfüllter Fall>! <Unerfüllter Fall>
```

notiert, wobei der unerfüllte Fall, wie im Metamodell vorgesehen, optional ist. Bedingungen werden wie folgt realisiert:

- Durch Bezug zum Optimierungskriterium. Für die Angabe des Optimierungskriteriums sind vier Schlüsselwörter (für <Name>) reserviert, die die folgende Semantik haben:
    - Performance: Performance
    - Resources: Speicherverbrauch
    - Safety: Sicherheit und Verfügbarkeit
    - Reuse: Wiederverwendung, Wartbarkeit, Portabilität, Erweiterbarkeit

```
OptimCriteria <Name>
```

- Durch eine Benutzerrückfrage.

```
UserQuestion("<Fragetext>",<Boolescher Standardwert>,<Ergebnisvariable>)
```

- Durch eine maschinenlesbare Anfrage an eine externe Informationsquelle. Mit dieser Abfrage werden insbesondere die Modelle der technischen Systemarchitektur abgefragt. Die Syntax von Anfrage kann implementierungsspezifisch festgelegt werden.

```
MaschineQuery("<Anfrage>",<Boolescher Standardwert>,<Ergebnisvariable>)
```

- Durch logische Operationen zwischen OptimCriteria, UserQuestion und MaschineQuery (mit den Schlüsselwörtern NOT, AND, OR).

Die Ergebnisvariable einer Benutzerrückfrage kann später ebenfalls als Bedingung verwendet werden. Hilfsregeln werden über

```
<Name der Hilfsregel>(<Argument_1>,...,<Argument_n>)
```

aufgerufen. Einen Ausdruck für das Löschen von Elementen gibt es in dieser Realisierung nicht.

Für die Realisierung der Iterationsfunktion kann einem mengenwertigen Ausdruck das Schlüsselwort .elements angefügt werden, das beim Aufruf von Hilfsregeln oder bei einem Argument einer Template-Instanziierung verdeutlicht, dass über diese Menge iteriert werden soll. Für eine komfortablere Handhabung von Mengen wurde zudem die Möglichkeit eingeführt, Elemente und/oder Mengen über das +-Symbol zu vereinigen.

Für die Namensermittlung sind verschiedene Möglichkeiten vorgesehen:

- Der Programmierer der Transformationsregeln gibt den Namen vor, indem er einen String (in Anführungszeichen gefasster Text) als Argument angibt.
- Der Name wird von einem Element vorgegeben, das bereits über den Variablennamen einer Template-Instanz (im Analyse- oder Designmodell) ansprechbar ist. In diesem Fall kann der Name über <Variablenname>.<Parametername> aufgelöst werden.
- Der Benutzer muss eigene Namen vorgeben. Dazu wird die Methode getNames("<Aufforderungstext>",<Anzahl>) bereitgestellt, die insgesamt Anzahl mal über einen Benutzerdialog einen String abfragt. Wird als Anzahl -1 übergeben, wird die Abfrage

in einer offenen Schleife wiederholt, bis der Benutzer sie beendet. Der Benutzer kann somit beliebig viele Namen eingeben.

- Mit der Methode getNamesByMachineQuery(„<Anfrage>“) wird analog zu MachineQuery eine maschinenlesbare Anfrage an eine externe Informationsquelle gestellt, die ein Array aus Strings zurückliefert. Mit dieser Abfrage werden insbesondere die Modelle der technischen Systemarchitektur abgefragt. Die Syntax von Anfrage kann implementierungsspezifisch festgelegt werden.

Für die kontextabhängige Formulierung von Fragen bei Benutzerrückfragen oder bei der Aufforderung zur Namenseingabe ist auch die Konkatenation von Strings sinnvoll. Dafür wurde das „*”-Zeichen vorgesehen, um z. B. Fragen der Art

```
"Name der Zustandsklasse für " * <Variablenname>
```

zu formulieren.

4.3.3.b Grammatik in EBNF

Die textuelle Syntax der Transformationssprache wird im Folgenden in EBNF [ISO/IEC '98] dargestellt. Dabei wird die häufig in der Literatur zu findende Variante der EBNF verwendet, in der Terminalsymbole in einfache Hochkommata eingefasst, optionale Ausdrücke mit einem nachgestellten Fragezeichen und beliebig oft vorkommende Ausdrücke mit einem nachgestellten Sternsymbol gekennzeichnet werden. Die Basissymbole, Stringliterale, Ganzzahlen und Variablenbezeichner sind hier im Sinne der Übersichtlichkeit nur informell beschrieben.

```
<traforule> ::= 'Rule' <ident> '(' <paramlist>? ')' ':' <lhs> '->' <rhs> '.'
<lhs> ::= ('Template' <ident> <ident>)?
<paramlist> ::= (<paramlist> ',')? <ident>
<rhs>      ::= '[' <condition> ']?' <condexp> (',' <rhs>)?
      | <expression> (',' <rhs>)?
<condexp>  ::= <expression> ('!' <expression>)?
<ccondition> ::= <condition> (('AND'|'OR') <ccondition>)?
<condition> ::= ('NOT' <condition>)
      | 'UserQuestion?(' <string> ',' <boolean> ',' <ident> ')'
      | 'MachineQuery?(' <string> ',' <boolean> ',' <ident> ')'
      | 'OptimCriteria' <ident>
      | <ident>
<expression> ::= 'Template' <ident> '('<argumentlist>')' <ident>?
      | <ident> '(' <argumentlist> ')'
      | 'Class(' <argument> ',' <boolean> ')'
      | 'Attribute(' <argument> ',' <argument> ',' <argument> ')'
      | 'Operation(' <argument> ',' <argument> ',' <argument> ')'
      | 'Association(' <argument> ','<argument>',' <boolean>')'
      | 'Inheritance(' <argument> ',' <argument> ')'
<argumentlist> ::= (<argumentlist> ',')? <argument>
```

```
<argument> ::= <ident> ('.' <ident>)? '.elements'?
      | <ident> ('.' <ident>)? ('+' <argument>)?
      | '[' <ccondition> ']?' <argument> '!' <argument>
      | <string> ('+' <argument>)?
      | 'getNames(' <string> ',' <int> ')'
      | 'getNamesByMachineQuery(' <string> ')'
<boolean>  ::= 'true' | 'false'
<string> ::= <string> '*' <string>
      | <ident> ('.' <ident>)?
      | <string_lit>
<string_lit> ::= in Hochkommata eingefasster Text
<int> ::= Ganzzahl
<ident> ::= Bezeichner für eine Variable
```

4.4 Verfolgbarkeitsmechanismus

Der Verfolgbarkeitsmechanismus von POTAD basiert auf der bereits in Kapitel 3.2.4.g beschriebenen Idee der Verlinkung von Musterinstanzen (siehe insb. Abbildung 3.49). Ein entsprechendes Metamodell ist in Abbildung 4.12 zu sehen. Es zeigt den Inhalt des Pakets `POTAD_Traceability`, das sich wie in Abbildung 4.5 gezeigt, in die restliche Metamodelllandschaft integriert.

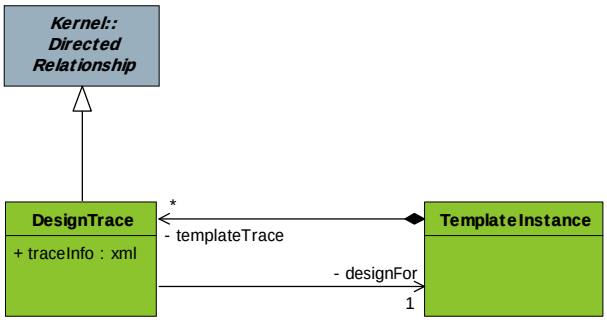


Abbildung 4.12: Metamodell für Verfolgbarkeitsinformationen

Bei dem Element `DesignTrace` handelt es sich um eine gerichtete Beziehung zwischen einer Designmuster- und einer Analysemusterinstanz. In dem Attribut `traceInfo` sind Designentscheidungen in maschinenlesbarer Form dokumentiert.

Im Werkzeug wird diese Verlinkung der Musterinstanzen durch die Einführung einer Abhängigkeitsbeziehung mit dem Stereotyp `<<Design For>>` zwischen Design- und Analysemuster-Bindungen realisiert. Für die Dokumentation von Designentscheidungen wurde ein XML-Format definiert, dass die Designentscheidungen für eine Ausführung einer Transformationsregel protokolliert und dem zugrunde liegenden Analysemuster als *tagged value* des Stereotyps `<<AnalysisCollaboration>>` hinzugefügt wird. Diese mit der übrigen Begriffswelt von POTAD nicht ganz konsistente Bezeichnung hat implementierungstechnische Gründe. Abbildung 4.13 zeigt die Verfolgbarkeitselemente beispielhaft.

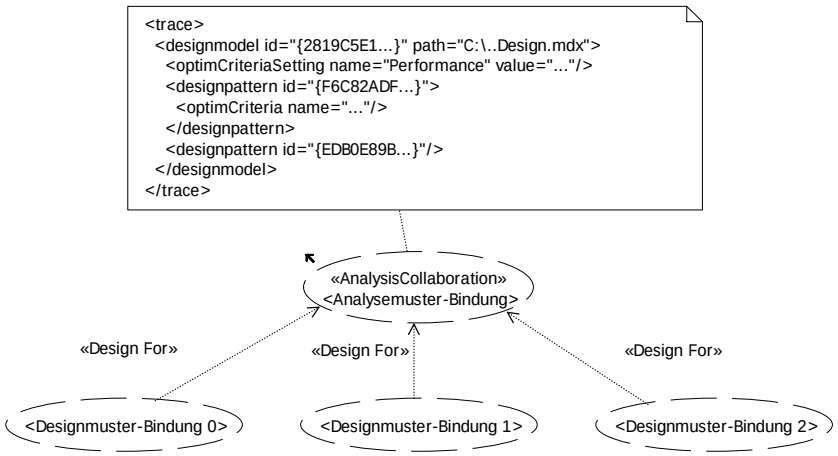


Abbildung 4.13: <<Design For>>-Beziehung mit Dokumentation der Designentscheidungen

Das folgende Listing zeigt ein Beispiel für das XML-Dokument in Auszügen. Das Format wird anschließend informell beschrieben.

```
<trace>
 <designmodel id="{20B80688-5F01…}" path=".\Design.mdx">
  <optimCriteriaSetting name="Performance" value="false"/>
  <designpattern id="{848E0EA5-…}"></designpattern>
  <designpattern id="{9A84CBA6-7047-4213-B9D5-229E0F9B20CB}">
   <userquestion text="Werden die Daten über einen Datenbus zur Verfügung gestellt?"
answer="false"/>
  </designpattern>
  <designpattern id="{96B2E34D-…}"></designpattern>
  <designpattern id="{50032718…}">
   <userquestion text="Soll der Regelalgorithmus verschiedene Strategien besitzen?"
answer="true"/>
  </designpattern>
 </designmodel>
</trace>
```

Eine Aufzeichnung wird immer eingeleitet durch das Wurzelelement <trace>, das beliebig viele Elemente vom Typ <designmodel> enthält, so dass die Information auch für mehrere Transformationen mit unterschiedlichen Zielmodellen festgehalten werden kann (in der bisherigen Implementierung wurde allerdings die Beschränkung auf ein Designmodell vorgenommen). Für das Designmodell wird jeweils die ID innerhalb von *Rational XDE* und der Pfad angegeben. Daraufhin folgt eine Reihe von Elementen <optimCriteriaSetting> mit Name und Wert, die für die Wahl der einzelnen Optimierungskriterien stehen. Obwohl diese für eine bestimmte Transformation jeweils identisch sind, werden sie für jedes Analysemuster erneut protokolliert. Anschließend werden die erzeugten Designmuster mit dem Tag <designpattern> und dem Attribut id der Reihe nach aufgeführt. Über die IDs sind die Muster innerhalb eines Namensraumes eindeutig identifizierbar. Falls die Erzeugung des Musters an Bedingungen geknüpft war, werden diese innerhalb des <designpattern>-Tags mit Hilfe der Elemente <userquestion> und <OptimCriteriaSetting> abgelegt. Für die Benutzerrückfragen wird dabei der Text (Attribut

text) und die Antwort (Attribut answer) sowie für Optimierungskriterien der Name (Attribut name) festgehalten. Ist die Bedingung negiert, so wird sie zusätzlich von einem Tag <negated> umrahmt. Insgesamt kann so für eine Template-Instanz im Analysemodell festgehalten werden, welche Designmuster unter welchen Bedingungen in einer bestimmten Transformation aus ihr hervorgingen und welche Randbedingungen in Form der Optimierungskriterien bei dieser Transformation galten.

5 Prototypische Umsetzung

In diesem Kapitel wird der Nachweis der Anwendbarkeit für den in Kapitel 4 beschriebenen Lösungsansatz durch eine prototypische Implementierung erbracht. Die Hintergründe und die Architektur der prototypischen Implementierung werden vorgestellt und erläutert.

Die Lösung ist eine Erweiterung des Werkzeugs *Rational XDE* und besteht aus zwei Teilen: Dem Modelltransformator, der die in Kapitel 4 beschriebene Modelltransformation durchführt und dem Verfolgbarkeitsnavigator, der die während der Transformation angelegten Verfolgbarkeitsinformationen auswertet und den Benutzer bei Verfolgung von Designentscheidungen unterstützt. Die Benutzung des so erweiterten Modellierungswerkzeugs ist an die in Kapitel 4.1 dargestellte POTAD-Methode angelehnt. An der Implementierung waren die in Kapitel 6.2 erwähnten studentischen Arbeiten in großem Umfang beteiligt.

5.1 Modelltransformator

Zunächst werden die Werkzeugplattform sowie die an der prototypischen Umsetzung beteiligten Komponenten und deren Zusammenhang in Form einer Grobarchitektur skizziert. Im Rahmen der bereits in Kapitel 3.2.4.f erwähnten Evaluierung von Werkzeugen mit Musterunterstützung, erwies sich das dort beschriebene Werkzeug *Rational XDE*, bezogen auf die Anforderungen nach der Darstellung und Expansion von Mustern, als die passendste Lösung. Aus diesem Grund wurde XDE als Werkzeugplattform für die prototypische Umsetzung von POTAD gewählt. Einige Elemente aus dem POTAD-Metamodell, die die Modellierung und insbesondere die Verarbeitung von Mustern betreffen, können somit auf existierende Mechanismen der Werkzeugplattform abgebildet werden. Die in Kapitel 4.3.3 vorgestellte Transformationssprache wird von einem eigens entwickelten Parser verarbeitet, der die Schnittstellen der übrigen Komponenten nutzt, um Elemente in den beteiligten Modellen zu suchen, zu ändern oder hinzuzufügen und die Interaktion mit dem Benutzer realisiert.

5.1.1 Architektur

Neben der integrierten Musterunterstützung bietet XDE als Eclipse-basiertes Werkzeug eine Vielzahl von Erweiterungsmöglichkeiten in Form von Plug-ins. Mit der *RXE API* (*Rational XDE Extensibility*) steht eine öffentliche Schnittstelle zur Verfügung, mit der allgemeine Operationen auf Modellen wie das Auslesen oder Erzeugen von Elementen über Java oder .NET vorgenommen werden können. Leider beinhaltet die RXE-Schnittstelle keine direkte Unterstützung der *Pattern Engine*, so dass es in dieser Bibliothek keine Befehle für die Expansion von Mustern und das Anlegen von Bindungen gibt. Diese Lücke kann jedoch durch die Verwendung des *MDA Toolkit for Rational XDE Java* [IBM c], einer Sammlung von Plug-ins für Modelltransformationen, geschlossen werde. Die hier zur Verfügung gestellte *MDA Toolkit API* bietet unter anderem die gesuchten Methoden, um Muster zu binden und zu expandieren. Des Weiteren ermöglicht das Toolkit auch die komfortable Programmierung von Transformationen als weitere Plug-ins, die von einer Vorlage abgeleitet werden und auf zahlreiche Hilfsmethoden zugreifen

können. Ist das MDA-Toolkit installiert, müssen solche Transformationen nur noch als Plug-in exportiert und in ein spezielles Verzeichnis kopiert werden, damit sie innerhalb der Benutzeroberfläche von XDE als Menüeintrag auftauchen und gestartet werden können. Für die Transformationen lassen sich auf einfache Weise Parameter definieren, die in einem automatisch generierten Dialog vor deren Start vom Benutzer abgefragt werden. Die Implementierung einer Transformation kann beliebigen Java-Code enthalten, der insbesondere auch auf das RXE- und MDA-Toolkit-API zurückgreifen kann. Abbildung 5.1 zeigt im schematischen Überblick, wie das realisierte XDE-Plug-in aufgebaut ist. Diese Grundarchitektur wurde in Zusammenarbeit mit einem Diplomanden entwickelt. Teile der folgenden Erläuterungen lehnen sich daher stark an die in der entsprechenden Diplomarbeit [Buchwald '05] gemachte Dokumentation an.

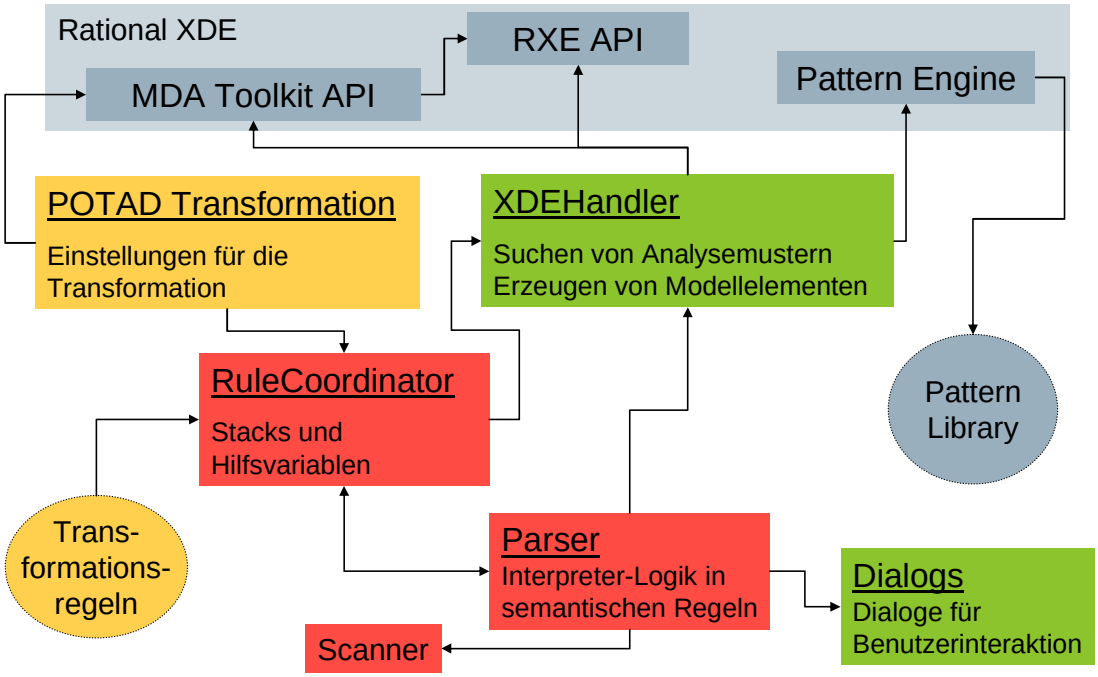


Abbildung 5.1: Schematische Darstellung der Architektur des XDE-Plug-ins

Die Klasse *POTAD Transformation* repräsentiert ein Grundgerüst aus mehreren Klassen, das nach dem Schema des *MDA Toolkits* bei der Definition der Transformation als Plug-in benötigt wird. Dazu gehören Klassen für die Anpassung der Parameter, des Menüs und der vor der Transformation angezeigten Dialoge.

Die Klasse *POTAD Transformation* selbst enthält eine Methode, die von dem *MDA Toolkit* aufgerufen wird, wenn der Benutzer die Transformation startet. In dieser Methode werden die Parameter ausgelesen und verarbeitet sowie anschließend über eine Methode der Klasse *RuleCoordinator* die Ausführung der Regeln gestartet. Diese Klasse enthält eine Reihe von statischen Feldern, in denen die Werte der Transformationsparameter, Stacks für Variablen und Argumente aus den Transformationsregeln sowie Flags für verschiedene Modi im gesamten Paket zugreifbar gemacht werden. Im operationalen Teil liest die Klasse alle Regeln in dem als

Parameter angegebenen Verzeichnis für die Regeldateien aus und koordiniert jeweils die Initialisierung und den Aufruf von Scanner und Parser.

Die Klasse *Parser* ist Ergebnis der Generierung mittels CUP und beinhaltet in den semantischen Regeln die Aktionen, die bei Erkennung eines bestimmten Teilworts durch den Parser ausgeführt werden (siehe folgendes Unterkapitel). Diese Aktionen bestehen darin, den Inhalt der Stacks oder die Modi zu verändern sowie über die beiden Klassen *XDEHandler* und *Dialogs* Muster im Analysemodell zu suchen, Elemente im Designmodell zu erzeugen bzw. Dialoge für die Benutzerinteraktion aufzurufen. Für die Handhabung von verschachtelten Transformationsregeln ruft der Parser erneut die Klasse *RuleCoordinator* zu Hilfe. Die Klasse *XDEHandler* sorgt für die Erzeugung von Designmustern inklusive Expansion und Anlegen von Bindungen, die Erzeugung der übrigen Elemente (Klassen, Attribute, Operationen, Assoziationen und Vererbungsbeziehungen), das Auffinden von Analysemustern sowie die Auflösung von Variablen. Für die Erzeugung der Muster wird dabei auf die *Pattern Engine* von XDE zurückgegriffen, die die benötigten Informationen über die Templates aus der als Parameter angegebenen Musterbibliothek bezieht. Die *Dialogs*-Klasse ermöglicht schließlich über einfache Methodenaufrufe die Realisierung der Benutzerrückfragen und der Namensabfrage.

Für eine Umsetzung des *Traceability*-Mechanismus wurde das Anlegen der `<<design for>>`-Beziehungen bei der Erzeugung eines Designmusters sowie die Protokollierung der Designentscheidungen im XML-Format bereits in die semantischen Regeln integriert. Die XML-Informationen über erzeugte Designmuster und damit verbundene Designentscheidungen werden dem Analysemuster in einem Tagged Value hinzugefügt, welcher zu dem Stereotyp `<<AnalysisCollaboration>>` gehört. Für diesen Zweck wurde mit dem ebenfalls zum *MDA Toolkit* gehörenden *Profile Manager* ein eigenes Profil erstellt, das für die Nutzung des Plug-ins separat installiert werden muss. Schließlich konnte das gesamte Projekt als Plug-in in ein Java-Archiv exportiert werden, so dass es in alle XDE-Installationen, die das *MDA Toolkit* enthalten, integrierbar ist.

5.1.2 Scanner und Parser für die Transformationsregeln

Da für die Formulierung der Transformationsregeln die erarbeitete textuelle Syntax aus Kapitel 4.3.3 verwendet werden sollte, war nach einer Möglichkeit gesucht, solche Regeln einzulesen und auszuwerten. Aufgrund der Charakteristik einer Programmiersprache wurde entschieden, eine Kombination aus Scanner- und Parsergenerator zu verwenden, die – bedingt durch die Integration in XDE – jedoch auf der Basis von Java arbeiten musste. Eine hierzu passende Lösung bieten die freien Werkzeuge *JFlex* [Klein] und *CUP* [Hudson]. JFlex ist ein Java-basierter Scannergenerator, der sich an dem verbreiteten C/C++-Tool *flex* [flex] orientiert, und *CUP* ist ein Generator für LALR-Parser, der die meisten Funktionen des oft referenzierten *YACC* [Corbett] bietet. CUP ist selbst in Java geschrieben und produziert Parser, die ebenfalls in Java implementiert sind. Er bietet eine Scannerschnittstelle, die zu JFlex kompatibel ist, und erlaubt die Integration von beliebigem Java-Code in den semantischen Regeln. Mit diesen beiden Werkzeugen ist es möglich, einen Java-basierten Parser für einzelne Transformations-

regeln zu erzeugen und einen Großteil der Auswertungslogik bereits in den semantischen Regeln der Parserspezifikation zu integrieren.

Zunächst wurde eine Scannerspezifikation auf Basis einer an Java orientierten Vorlage verfasst, die Konstrukte wie die Berücksichtigung von Kommentaren, der Aufbau von Bezeichnern und die Erkennung von Strings bereits enthält. Dementsprechend erlaubt das Plug-in in den Transformationsregeln die Verwendung von Kommentaren, Bezeichnern und Strings nach dem Vorbild von Java. Auf Basis dieser Spezifikation wurde der Java-Code für den Scanner erzeugt, der nur aus einer Klasse besteht.

Bei der Spezifikation des Parsers im nächsten Schritt wurden die vom Scanner zurückgelieferten Symbole als Terminale verwendet sowie in Anlehnung an die Grammatik in EBNF (siehe Kapitel 4.3.3.b) verschiedene Nichtterminale definiert. Die Grammatik in EBNF-Darstellung konnte als Grundlage dienen, musste allerdings an CUP-spezifischen Vorgaben angepasst werden. Innerhalb der semantischen Regeln wurde die Auswertungslogik für die Transformationsregeln implementiert. Abschließend wurde der Parser generiert, der aus einer Klasse für die Symboltabelle und einer weiteren für die Parserlogik besteht.

5.1.3 Benutzung

Im Folgenden soll ein typisches Anwendungsszenario des Plug-ins mit Hilfe einiger Screenshots erläutert werden. Vor dem Start der Transformation muss der Benutzer ein Analysemodell unter Verwendung von Analysemustern erstellt haben und die zu verwendenden Transformationsregeln als separate Dateien mit der Dateiendung `.tra` in einem gemeinsamen Verzeichnis abgelegt haben. Außerdem müssen in einem (nicht notwendigerweise separaten) Modell alle Designmuster, die im Rahmen der Transformationsregeln instanziiert werden, als *Pattern Asset* definiert sein.

Mit diesen Voraussetzungen kann die Transformation über einen Menüeintrag aufgerufen werden (Abbildung 5.2). Daraufhin erscheint zunächst der Dialog mit den Transformationsparametern (Abbildung 5.3), die über die Benutzervorgaben auch mit Standardwerten versehen werden können. Hier müssen die Dateien mit dem Analysemodell, dem Designmodell und der Musterbibliothek sowie das Verzeichnis mit den Transformationsregeln angegeben werden. Außerdem kann hier eines der vier Optimierungskriterien ausgewählt und bei Bedarf die Benutzerrückfragen unterdrückt werden. Die Wahl der auszuführenden Aktionen ist vom *MDA Toolkit* vorgegeben und kann nicht unterdrückt werden. Für den Prototyp ist diese Wahl unnötig, da die Validierung der Parameter sowie die Verifikation des Ergebnisses nicht implementiert sind.

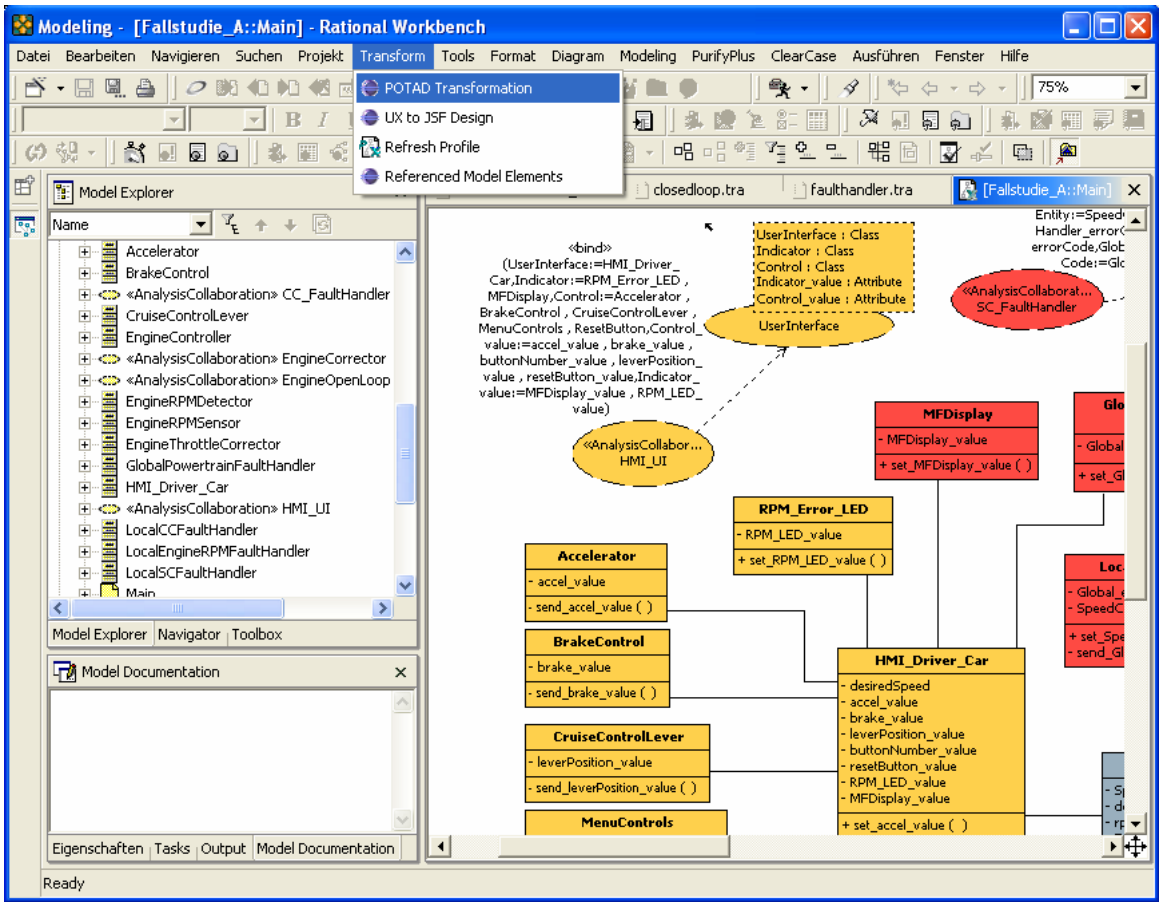


Abbildung 5.2: Start der Transformation in XDE

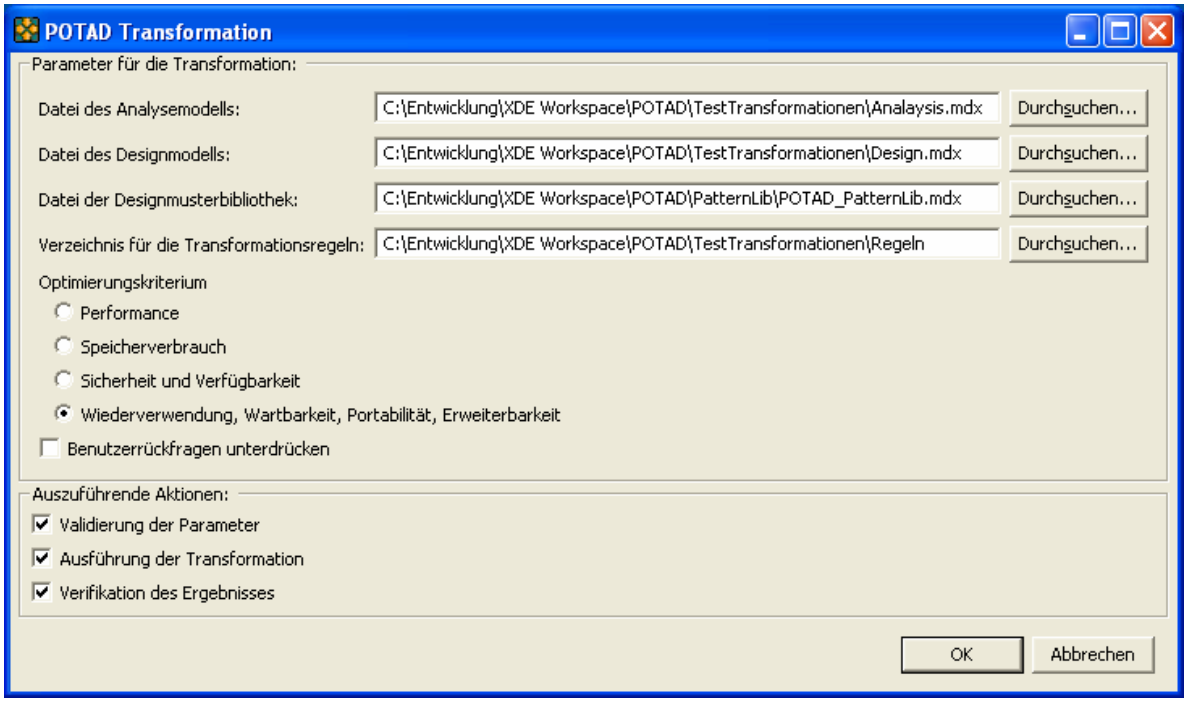


Abbildung 5.3: Dialog mit Einstellungen für die Transformation

Mit „OK“ wird die Transformation gestartet. Nach dem Einlesen der Modelle erscheint bei Erkennung eines Musters im Analysemodell und dem Vorhandensein einer zu dem Template passenden Transformationsregel eine Meldung. Nach Bestätigung wird der zugehörige rechte Teil der Regel ausgeführt. Wird in der Transformationsregel eine Benutzerrückfrage gestellt, so erscheint (sofern nicht unterdrückt) ein Fenster mit dem Fragetext und den zwei möglichen Antworten (oberer Dialog in Abbildung 5.3). Die Transformation wird unterbrochen, bis die Frage beantwortet wird. Bei der Abfrage von Namen erscheint ebenfalls ein Dialog, der den Fragetext ausgibt und eine Zeichenkette als Eingabe verlangt. Der untere Dialog in Abbildung 5.3 zeigt dies für den Sonderfall der Abfrage einer unbestimmten Anzahl von Namen, bei dem durch Bestätigen mit „OK“ dieselbe Abfrage immer wieder erscheint und erst mit „Abbrechen“ beendet wird.

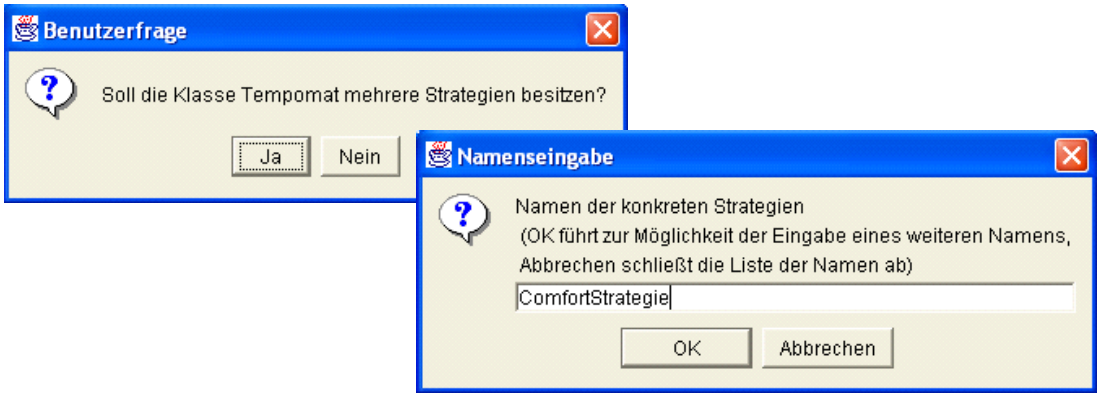


Abbildung 5.4: Benutzerrückfragen während der Transformation

Auf diese Weise werden die Transformationsregeln für alle gefundenen Template-Instanzen im Analysemodell durchgeführt. Am Ende wird die erfolgreiche Durchführung der gesamten Transformation mit einem weiteren Dialog bestätigt. Mit erfolgreicher Durchführung findet sich das Ergebnis der Transformation in Form von Modellelementen in dem als Parameter angegebenen Designmodell. Die erzeugten Elemente sind zunächst nur in einer Baumansicht im *Model Explorer* von XDE zu sehen, können aber zu Visualisierungszwecken individuell in Diagrammen angezeigt und angeordnet werden.

5.2 Verfolgbarkeitsnavigator

Der Verfolgbarkeitsnavigator realisiert die in Kapitel 4.1.2 beschriebenen drei Abfragen der während der Modelltransformation angelegten Verfolgbarkeitsinformationen als Plug-in des Werkzeugs *Rational XDE*. Somit erlaubt die prototypische Umsetzung von POTAD die Modellierung, die Modelltransformation und die Abfrage der Verfolgbarkeitsverknüpfungen in einem integrierten Werkzeug.

Das in Java programmierte Plug-in wertet den XML-String aus, der einer Analysmusterinstanz als Tagged Value `TraceInfo` des Stereotyps `AnalysisCollaboration` aus dem Profil `Traceability` während des Transformationsvorgangs hinzugefügt wurde. Eingebunden in das Werkzeug

werden die Abfragen über den *XDE Menu Extender-Mechanismus*. Dieser erlaubt es, eigene Software in das XDE-Menüsystem einzubinden – auch in die Kontextmenüs. Dem Werkzeug bekannt gemacht wird solch eine *menu extension* mit einem *menu specification file*, das in einem entsprechenden Verzeichnis liegen muss.

Auf diese Weise können die Abfragen über Kontextmenüs genau für die Kontexte gestartet werden, die in Kapitel 4.1.2 vorgesehen sind. Abbildung 5.5 zeigt den Aufruf über das Kontextmenü einer Analysemusterinstanz. Abfragen, die nicht zu dem Kontext passen, sind deaktiviert.

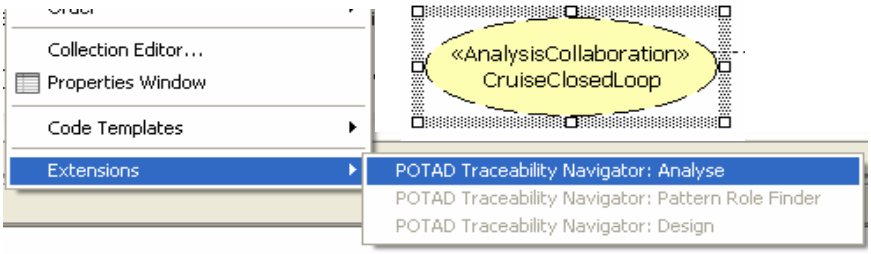


Abbildung 5.5: Aufruf des Verfolgbarkeitsnavigators über das Kontextmenu

Im Folgenden sind die drei Abfragen noch einmal implementierungsspezifisch beschrieben:

1. Die Abfrage *Analyse* kann über das Kontextmenü einer Musterinstanz im Model Explorer oder einem Diagramm aufgerufen werden, die über einen Stereotyp `<<AnalysisCollaboration>>` verfügt und außerdem eine `<<design for>>`-Beziehung eingeht. Ziel der Abfrage ist es zu ermitteln, welche Designmuster durch eine Transformation aus diesem Analysemuster hervorgegangen sind. Wie in Abbildung 5.6 zu sehen ist, werden die bei der Transformation verwendeten Optimierungskriterien des Transformationsvorgangs, die entsprechenden Designmuster sowie ggf. die Antworten auf gestellte Benutzerrückfragen angezeigt.

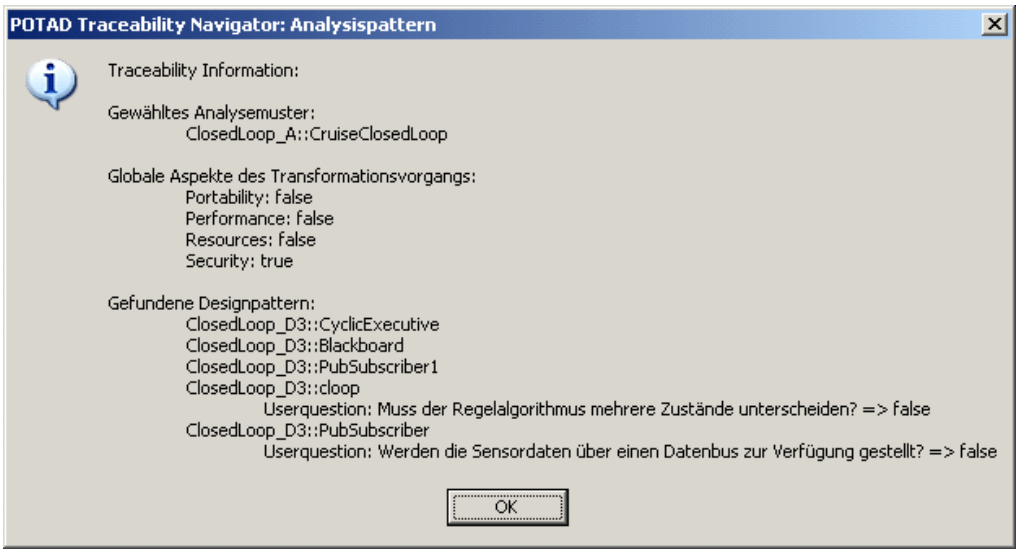


Abbildung 5.6: Ergebnis einer Abfrage für den Kontext *Analysemuster*

2. Die Abfrage *Design* kann über das Kontextmenü einer Musterinstanz im Model Explorer oder einem Diagramm aufgerufen werden, die eine <<design for>>-Beziehung eingeht. Ziel der Abfrage ist es zu ermitteln, aus welchem Analysemuster dieses Designmuster durch eine Transformation hervorgegangen ist. Wie in Abbildung 5.7 zu sehen ist, werden die bei der Transformation verwendeten globalen Optimierungskriterien des Transformationsvorgangs, die entsprechenden Designmuster sowie ggf. die Antworten auf gestellte Benutzerrückfragen angezeigt.

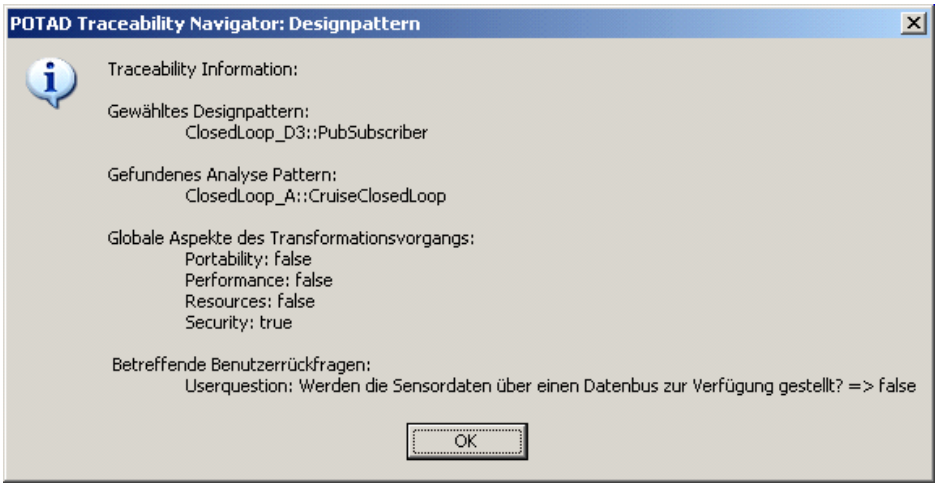


Abbildung 5.7: Ergebnis einer Abfrage für den Kontext *Designmuster*

3. Die Abfrage *Pattern Role Finder* kann über das Kontextmenü einer Klasse im Model Explorer oder einem Diagramm aufgerufen werden. Ziel der Abfrage ist es zu ermitteln, an welchen Musterinstanzen diese Klasse beteiligt ist. Wie in Abbildung 5.8 zu sehen ist, enthält das Ergebnis die entsprechenden Muster und die Parameter, die die Klasse jeweils belegt.

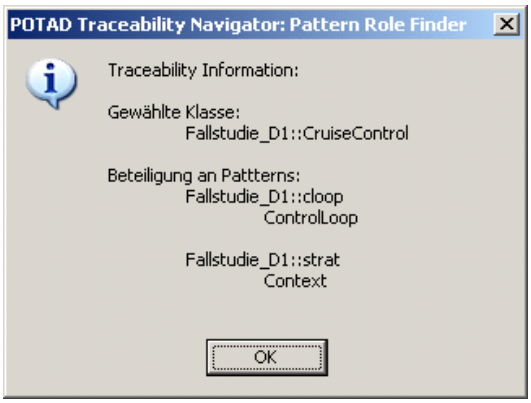


Abbildung 5.8: Ergebnis einer Abfrage für den Kontext *Klasse*

5.3 Beispiel

Im Folgenden wird eine Transformationsregel für das Muster *ClosedLoopControl* detaillierter erläutert, das unter anderem die in Abbildung 3.45 und Abbildung 3.47 gezeigten Designmodelle erzeugen kann. Die hierbei angestellten Designüberlegungen sind in Kapitel 3.2.4.g geschildert. Die Auswirkung der Regel wird anhand des ebenfalls dort verwendeten Beispiels *CruiseController* erklärt. Bei der Diskussion des Verschmelzungsverhaltens wird angenommen, das das Attribut `mergeBehavior` bei jedem Musterelement die Einstellung `merge` hat.

Der Kopf der Regel `ClosedLoopControl` sieht wie folgt aus:

```
Rule ClosedLoopControl ():
Template ClosedLoopControl cl ->
```

Zunächst wird eine parameterlose Regel mit dem Namen `ClosedLoopControl` angelegt. Auf der linken Seite der Regel ist das Template `ClosedLoopControl` mit der Variable `cl` angegeben. Damit wird die rechte Seite der Regel ausgeführt, wenn im Modell eine Bindung des Templates `ClosedLoopControl` gefunden wird. Über die Variable `cl` können Anweisungen auf der rechten Seite auf die gefundene Template-Bindung zugreifen.

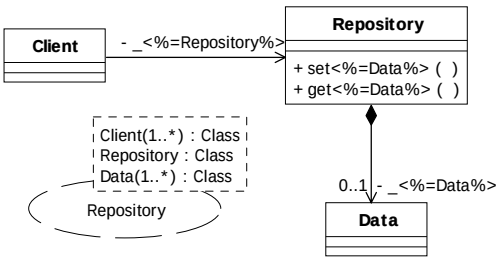
Die rechte Seite der Regel beginnt mit Template-Instanziierung, die abhängig von einer Bedingung ist:

```
[OptimCriteria Reuse]
  ?Template Repository(
    cl.ClosedLoopController*"_Master",      // Client
    "InputRepository",                      // Repository
    cl.FeedbackValue+cl.ReferenceValue      // Data
  ) inputRep,
```

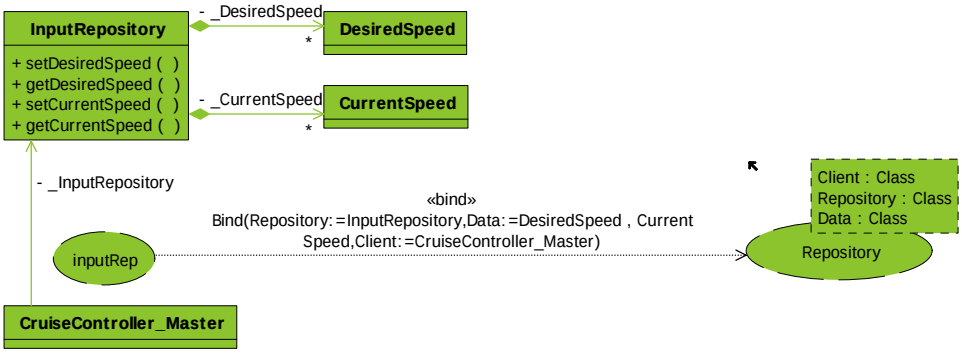
Wenn das Optimierungskriterium `Reuse` wahr ist, wird das Template `Repository` instanziiert. Der Template-Parameter `Client` wird mit einer Klasse belegt, die den Namen der Klasse `ClosedLoopController` aus dem Analysemuster trägt (durch Auslesen der Variable `cl`). Mit dem *-Operator wird der Postfix „`_Master`“ angehängt. Dem Template-Parameter `Repository` wird

eine Klasse mit konstantem Namen „InputRepository“ zugewiesen. Der Template-Parameter Data erhält mit Hilfe des +-Operators zwei Klassen. Diese haben die Namen der beiden Eingangssignale aus dem ClosedLoopControl-Muster (FeedbackValue und ReferenceValue). Der nachgestellte Bezeichner inputRep gibt der Instanz einen Namen und macht sie für spätere Anweisungen in der Regel referenzierbar.

Im Beispiel wird die Struktur des Musters Repository



wie folgt in das Zielmodell eingebracht:

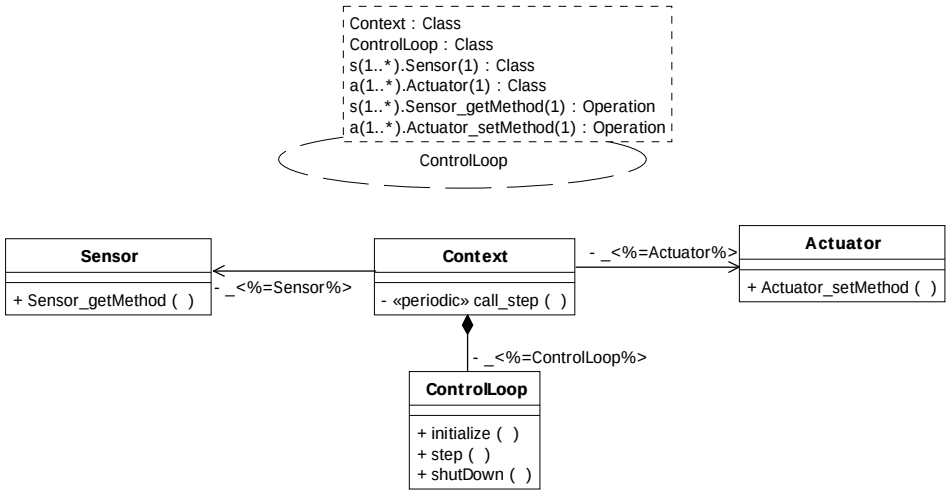


Im nächsten Schritt wird in jedem Fall das Template ControlLoop instanziiert:

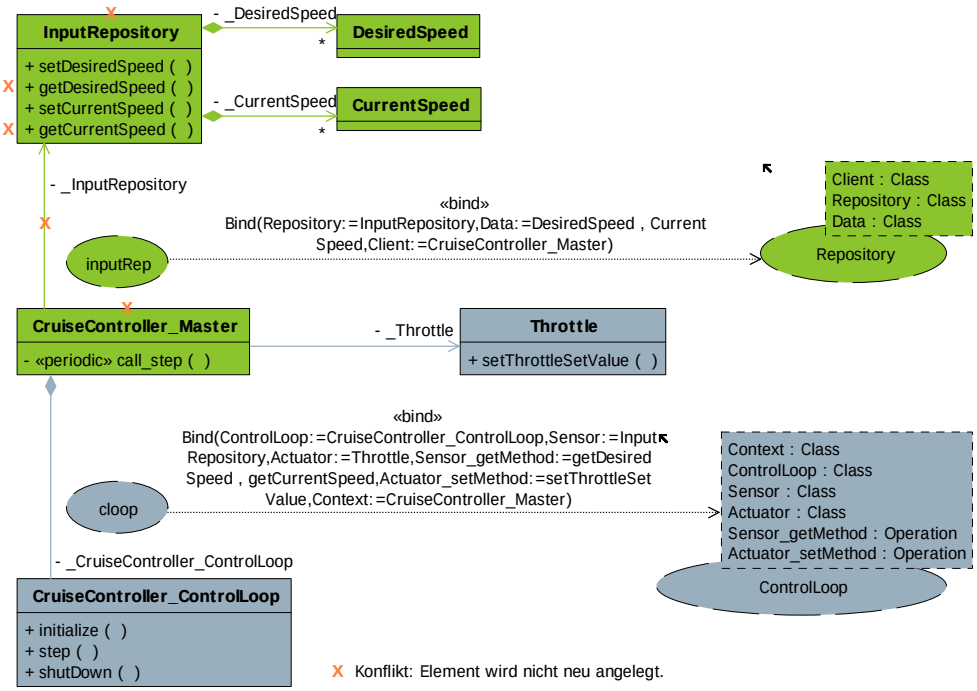
```
Template ControlLoop(
  cl.ClosedLoopController*"_Master",              // Context
  cl.ClosedLoopController*"_ControlLoop",         // ControlLoop
  [Reuse] ?inputRep.Repository+inputRep.Repository // Sensor
    !cl.FeedbackValueSource+cl.ReferenceValueSource,
  cl.ControlOutputValueDrain,                     // Actuator
  "get"*cl.FeedbackValue+"get"*cl.ReferenceValue, // Sensor_getMethod
  "set"*cl.ControlOutputValue                     // Actuator_getMethod
) cloop,
```

Bis auf den Parameter Sensor erfolgt die Parameterbelegung nach dem bereits bekannten Muster. Die Belegung bei Sensor ist abhängig vom Optimierungskriterium Reuse. Wenn Reuse wahr ist, werden die beiden Sensorwerte über das zuvor angelegte Repository mit Namen inputRep bezogen. Durch die Zuweisung agiert die Klasse InputRepository des Templates Repository im Kontext von ControlLoop in der Rolle von Sensor. Dies ist ein Beispiel wie Muster miteinander verwoben werden. Wenn Reuse nicht wahr ist, werden zwei Klassen angelegt, deren Namen sich wieder aus der Instanz von ClosedLoopControl ableiten. Dieser Pfad der Regel wird weiter unten besprochen.

Die Struktur des Musters ControlLoop



wird wie folgt mit dem Zielmodell verschmolzen:



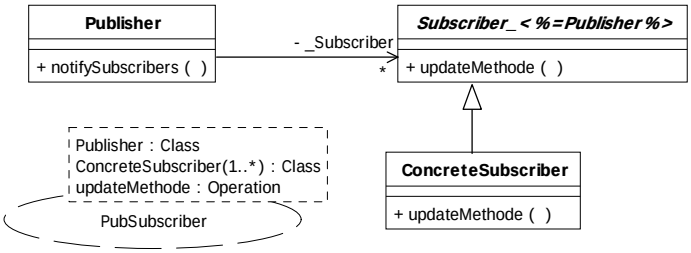
Bei dieser Verschmelzung treten einige Konflikte auf (Kreuze in Orange), die durch die Verschmelzungsregeln behandelt werden. So werden für die Parameter Context (hier CruiseController_Master) und Sensor keine neuen Klassen angelegt, da namensgleiche Klassen bereits im Modell existieren. Auch die in der Struktur von ControlLoop definierte gerichtete Assoziation zwischen Context (hier: CruiseController_Master) und Sensor (hier: Repository) wird nicht angelegt, da solch eine Assoziation bereits durch das Repository-Muster erzeugt wurde. Gleiches gilt für den Parameter Sensor_getMethod. Die sich hier ergebenden get-Methoden sind bereits vorhanden und werden „wiederverwendet“.

Im nächsten Schritt wird das Template PubSubscriber (*PublischerSubscriber*) zweimal instanziiert:

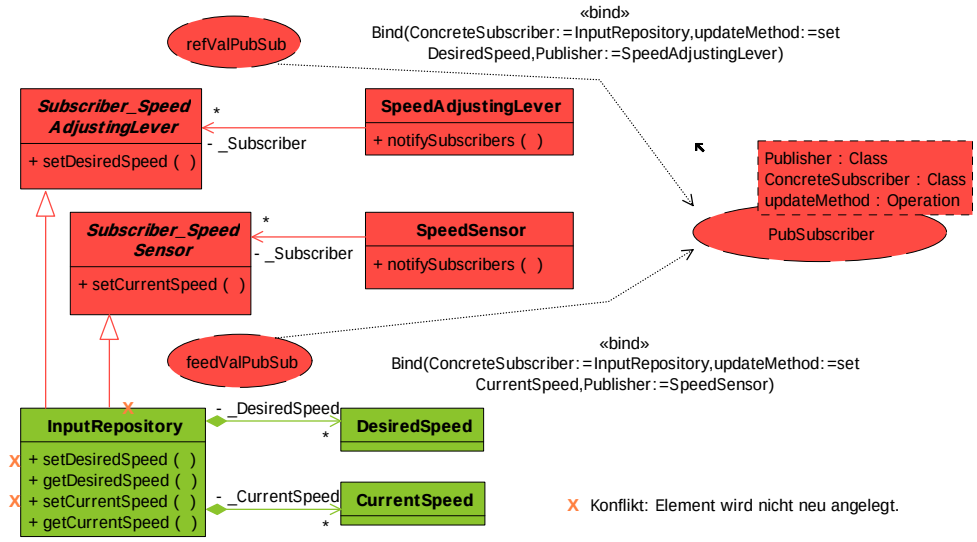
```
[OptimCriteria Reuse]
  ?Template PubSubscriber(
    cl.ReferenceValueSource,   // Publisher
    inputRep.Repository,             // ConcreteSubscriber
    "set"*cl.ReferenceValue          // updateMethode
  ) refValPubSub,
[OptimCriteria Reuse]
  ?Template PubSubscriber(
    cl.FeedbackValueSource,    // Publisher
    inputRep.Repository,             // ConcreteSubscriber
    "set"*cl.FeedbackValue           // updateMethode
  ) feedValPubSub,
```

Bei der Parameterzuweisung findet eine Verwebung mit dem zuvor angelegten Repository-Muster statt. Darüber hinaus gibt es bei dieser Instanziierung keine Besonderheiten.

Die Struktur des Musters PubSubscriber



wird mit dem Zielmodell (Ausschnitt) wie folgt verschmolzen:



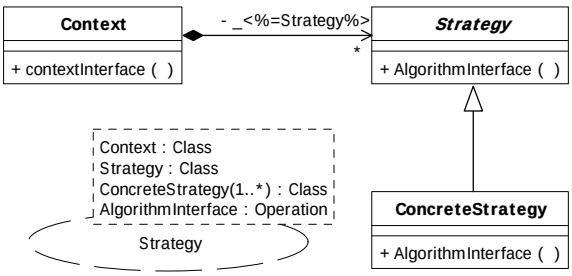
Die dem Parameter ConcreteSubscriber zugewiesene Klasse wurde bereits durch das Template Repository angelegt und wird daher „wiederverwendet“. Allerdings entstehen neue Vererbungsbeziehungen zu den abstrakten Klassen Subscriber_*.

Das Muster Strategy wird in Abhängigkeit des Optimierungskriteriums Reuse und einer Benutzerrückfrage instanziiert:

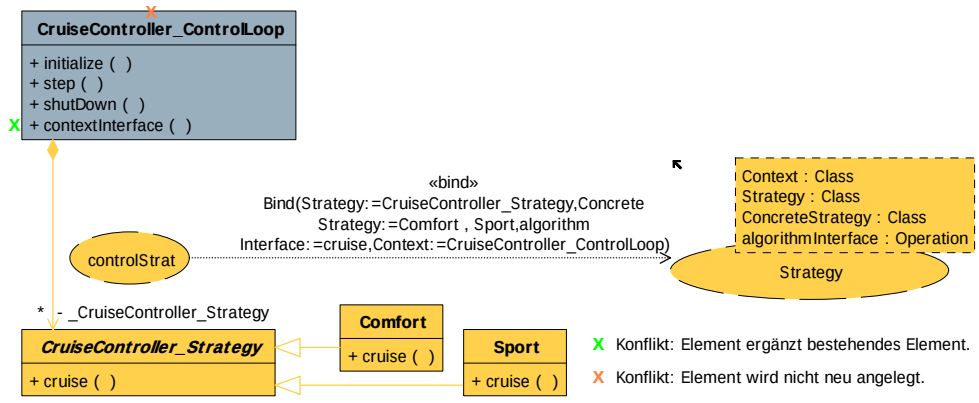
```
[OptimCriteria Reuse AND
UserQuestion("Müssen alternative Regelstrategien verwendet werden?",false, strat)]
  ?Template Strategy(
    cloop.ControlLoop,                          // Context
    cl.ClosedLoopController*"_Strategy",        // Strategy
    getNames("Namen der alternative Regelalgorithmen",-1),     // ConcreteStrategy
    getNames("Interface-Name des Regelalgorithmus",1)          // AlgorithmInterface
  ) controlStrat,
```

Das zweite Argument false ist der Default-Wert. Mit dem dritten Argument strat wird eine Variable deklariert, über die die Antwort im späteren Verlauf der Regel erneut ausgelesen werden kann. Da sich die Parameter ConcreteStrategy und AlgorithmInterface nicht aus dem Analysemodell ableiten, müssen Namen für neu anzulegende Elemente vergeben werden. Dies geschieht über getNames, wobei im ersten Fall beliebig viele Elemente erzeugt werden (zweites Argument ist „-1“) und im zweiten Fall genau ein Element erzeugt wird (zweites Argument ist „1“).

Wenn die UserQuestion mit „Ja“ beantwortet wird und bei getNames für ConcreteStrategy „Comfort“ und „Sport“ und für AlgorithmInterface „cruise“ eingegeben wird, wird die Struktur von Strategy



mit dem bisher erzeugten Modell (Ausschnitt) wie folgt verschmolzen:



Da der Parameter Context mit der existierenden ControlLoop-Instanz verwoben ist, gibt es hier einen Konflikt und die Klasse CruiseController_ControlLoop wird nicht neu angelegt. Im

Rahmen der Verschmelzung wird jedoch der existierenden Klasse die Operation contextInterface hinzugefügt (grünes Kreuz), die in der Template-Struktur von Strategy für Context vorgesehen ist.

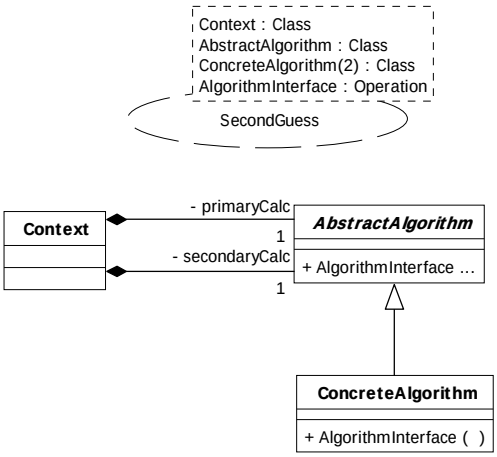
Nach der Instanziierung des Musters Strategy ist das in Abbildung 3.45 gezeigte Design erzeugt und die Regel für das Optimierungskriterium Reuse abgeschlossen.

Die folgenden Template-Instanziierungen werden nur ausgeführt, wenn das Optimierungskriterium Safety gewählt wurde. In diesem Fall wurden bis auf ControlLoop alle bisher erwähnten Template-Instanziierungen nicht ausgeführt. Ausgehend von einer ControlLoop-Instanz wird dann das SecondGuess-Template instanziiert:

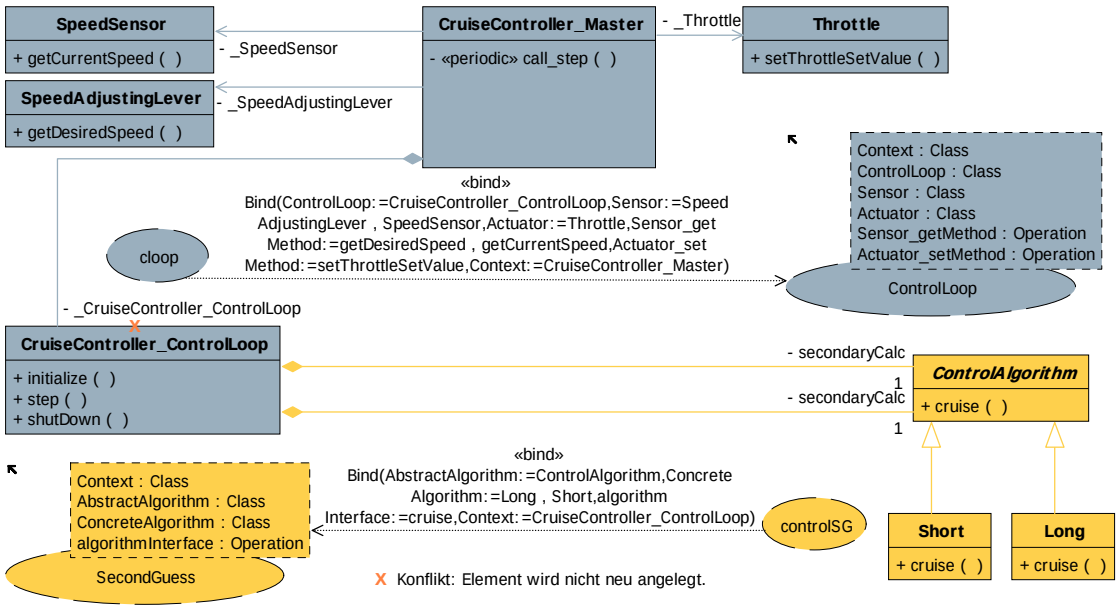
```
[OptimCriteria Safety]
  ?Template SecondGuess (
    cloop.ControlLoop,  // Context
    "ControlAlgorithm", // AbstractAlgorithm
    "Short"+"Long",           // Concrete
    getNames("Interface-Name des redundanten Regelalgorithmus",1)// AlgorithmInterface
  ) controlSG,
```

Über den Parameter Context wird diese Instanz mit der Instanz von ControlLoop verwoben. Die tatsächlichen Parameter für AbstractAlgorithm und ControlAlgorithm werden in diesem Fall durch konstante Strings vorgegeben. Der Name des Algorithmus kann, ähnlich wie bei der Instanziierung von Strategy, durch den Benutzer eingegeben werden.

Die Struktur des Templates SecondGuess



wird mit dem bisher erzeugten Modell wie folgt verschmolzen:

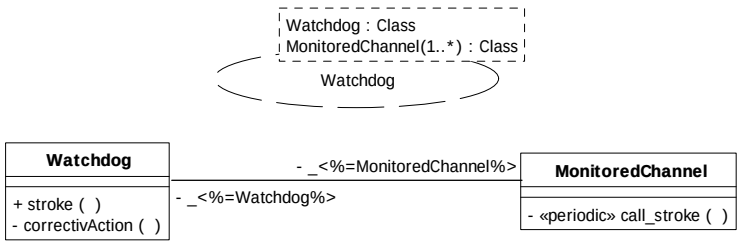


Ein Konflikt tritt bei dieser Instanziierung nur bei dem tatsächlichen Parameter für `Context` auf. Hier wird die bereits bestehende Klasse `CruiseController_ControlLoop` wiederverwendet.

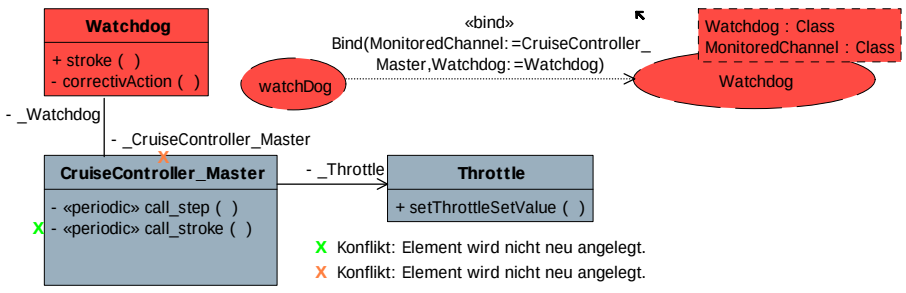
Abschließend wird das Template `Watchdog` instanziiert:

```
[OptimCriteria Safety]
  ?Template Watchdog (
    "Watchdog",  // Watchdog
    cloop.Context// MonitoredChannel
  ) watchDog.
```

Die Struktur des Templates `Watchdog`

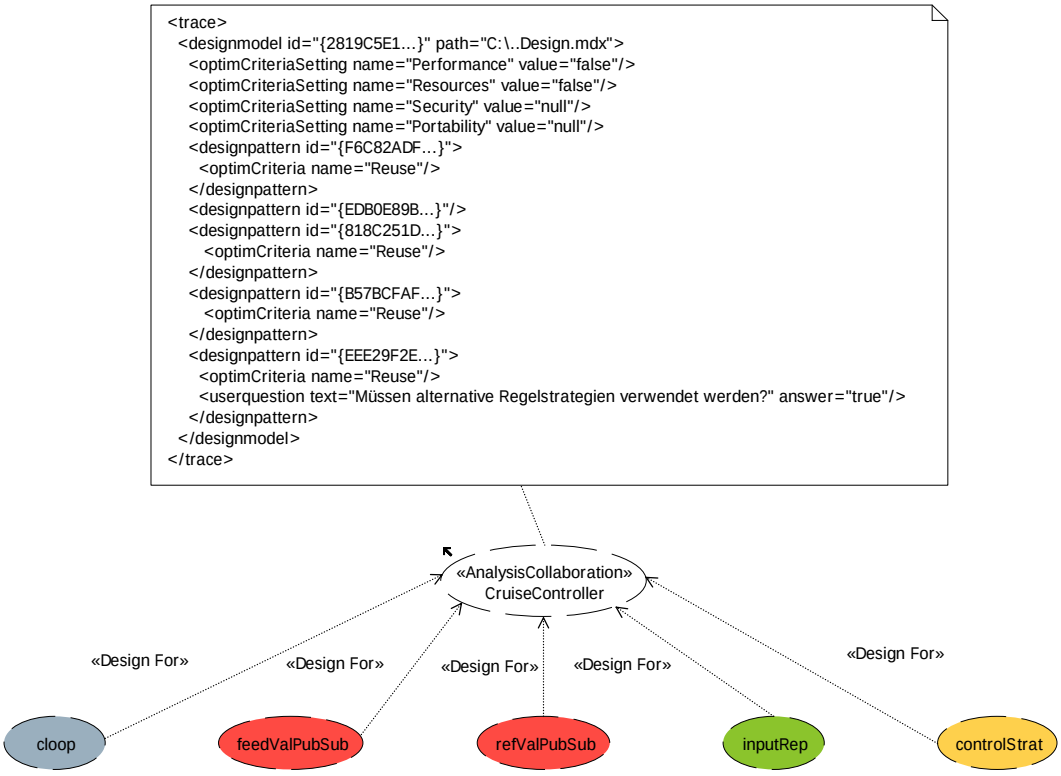


wird wie folgt mit dem existierenden Modell (Ausschnitt) verschmolzen:



Die dem Parameter `MonitoredChannel` zugewiesene Klasse `CruiseController_Master` existiert schon und wird daher nicht neu angelegt. Allerdings kommt die in dem Template definierte Operation `call_step()` hinzu.

Am Ende der Transformation sind die Musterinstanzen zu Verfolgbarkeitszwecken wie folgt verlinkt:



Die vorgestellte Regel kann das in Abbildung 3.45 und Abbildung 3.47 gezeigte Klassenmodell bis auf das graphische Layout zu 100% erzeugen. Die Regel erzeugt auch keine überflüssigen Elemente, die wieder gelöscht werden müssen. Zu den notwendigen manuellen Nacharbeiten einer Transformation gehört die Verfeinerung der Beziehungen. So müssen beispielsweise einfache Assoziationen in Kompositionen umgewandelt werden oder die Multiplizitäten angepasst werden.

Im B.1 ist eine umfangreichere Regel für das Analysemuster `ClosedLoopControl` enthalten, die unter anderem die Muster `ReactiveControlLoop` für zustandsbehaftete Regelungen und `HeterogeneousRedundancy` für redundant erfasste Sensorwerte verwendet. Darüber hinaus sind im Anhang – B weitere Transformationsregeln für weitere Muster zu finden.

6 Validierung und Bewertung

In diesem Kapitel wird der vorgestellte Lösungsansatz den gesteckten Zielen gegenübergestellt und bewertet. Zunächst wird geprüft, ob die gestellten Anforderungen durch den vorgestellten Lösungsansatz erfüllt werden, die Anforderungen werden validiert. Danach wird der Lösungsansatz ausgehend von diversen studentischen Projekten und eigenen Erfahrungen bewertet. Dabei werden die Erfahrungen mit POTAD im Rahmen der Fallstudie *KFS* zur Bewertung herangezogen. Abschließend werden derzeitige Grenzen der Lösung aufgezeigt.

6.1 Überprüfung des eigenen Ansatzes

Anhand der Problemstellung aus Kapitel 1.2 und der präzisierten Problemstellung aus Kapitel 3.4 wird hier der eigene Lösungsansatz aus Kapitel 4 überprüft. Basierend auf der Untersuchung von neuen modellbasierten Methoden im Umfeld der System- und Softwareentwicklung, wurden Modelltransformationen als ein mögliches Instrument identifiziert, um einzelne Aufgaben der Softwareentwicklung durch die Wiederverwendung von Modellen aus der Systementwicklung zu automatisieren. In Kapitel 3 wurden nach der Aufarbeitung der domänenspezifischen Anforderungen existierende Modelltransformationsansätze auf ihre Eignung in diesem Anwendungskontext hin untersucht. Bei diesen Untersuchungen wurden Unzulänglichkeiten der existierenden Ansätze entdeckt, die in der präzisierten Problemstellung zu folgenden drei Kernforderungen als Ergebnis der Analyse des Stands der Technik geführt haben:

1. Ein Modelltransformationsansatz muss das in der Methode bereits etablierte Musterkonzept wiederverwenden und weiterentwickeln. Die Umsetzung einer darauf basierenden Systematik verlangt von einem Modelltransformationsmechanismus die Möglichkeit, parametrisierbare Mustertypen und ihre Instanzen beschreiben zu können sowie die für die Musterinstanziierung notwendige Parameter-Bindung und die Musterexpansion durchzuführen.
2. Ein Modelltransformationsansatz muss die mit wechselnden nichtfunktionalen Anforderungen und unterschiedlichen technischen Systemarchitekturen einhergehende Variabilität im Designmodell unterstützen.
3. Ein Modelltransformationsansatz muss den Zusammenhang zwischen den Modellen der System- und Softwareentwicklung durch geeignete Verlinkung dauerhaft verfolgbar machen, so dass Designentscheidungen bei Variabilität im Zielmodell nachvollziehbar bleiben.

Diese drei Kernforderungen sind eine Zusammenfassung der in Kapitel 3.2.5 aufgestellten Liste mit Anforderungen an einen Modelltransformationsansatz. In Kapitel 4 wurde ausführlich ein Modelltransformationsansatz beschrieben, dessen Grundidee eine Neuerung im Bereich der Modelltransformationen darstellt: Den Zusammenhang zwischen formalisiert beschriebenen Analyse- und Designmustern zu systematisieren und daraus Transformationsregeln abzuleiten, die für eine Analysemusterinstanz Designmuster instanziieren. Für einen selbst erstellten Katalog aus 5 Analysemustern wurden mit der eigenen Transformationssprache Regeln erstellt,

die insgesamt 18 Designmuster aus einem ebenfalls im Rahmen dieser Arbeit zusammengestellten Katalog instanziieren. Durch diese Transformationsregeln und die entsprechenden Werkzeuge, die im letzten Kapitel vorgestellt wurden, ist die automatisierte Erstellung eines Designmodells auf der Basis von Mustern möglich. Daher sind die unter Punkt 1) aufgeführten Anforderungen erfüllt.

Die in Kapitel 4.3 vorgestellte Transformationssprache sieht zwei Arten von Variabilität vor. In Bezug auf nichtfunktionale Anforderungen kann ein Designmodell global in Richtung 1. Performance, 2. Ressourcenverbrauch, 3. Sicherheit und Verfügbarkeit oder 4. Wartbarkeit, Portabilität, Erweiterbarkeit und Wiederverwendung optimiert werden. Mit diesen vier Kategorien wurden nichtfunktionale Anforderungen zusammengefasst, die häufig zusammen auftreten. Die Benutzerrückfragen ermöglichen Variabilität in Bezug sehr spezifischer Details der technischen Systemarchitektur oder lokaler Optimierungsfragen. Für diese Variabilitäten sind die Anforderungen unter Punkt 2) erfüllt.

Das in Kapitel 4.4 vorgestellt Verfolgbarkeitskonzept verlinkt die Analysemusterinstanzen mit den Designmusterinstanzen, die durch eine Transformation aus dieser Analysemusterinstanz hervorgegangen sind. Die gesetzten globalen Optimierungsaspekte und Antworten auf Benutzerrückfragen werden dabei in einem maschinenlesbaren Format gespeichert. Auf diese Weise wird der Zusammenhang zwischen Analyse- und Designmodell systematisiert und Designentscheidungen sind auch im Nachhinein nachvollziehbar. Damit sind die Anforderungen unter Punkt 3) erfüllt.

Abschließend zeigt Tabelle 6.1, durch welche konkreten POTAD-Elemente die Anforderungen aus Kapitel 3.2.5 erfüllt werden.

Anforderung (Kapitel 3.2.5)	**Realisiert in POTAD durch**
UML-Klassenmodelle (UML)	Die Metamodelle `POTAD_Templates`, `POTAD_Traceability` und `POTAD_Transformations` sind in das UML2-Metamodell integriert. Quell- und Zielmodell einer Transformation sind UML-Klassendiagramme.
Template-Definition und -Bindung (TEM)	`Template` und `TemplateBinding`.
Template-Expansion (EXP)	Semantik von `TemplateInstantiation`.
Template-Instanz als Ausführungskriterium einer Regel (AKR)	Variante der `LHS`, die ein `Template` und eine `InstanceVariable` enthält (matched auf ein Analysemuster).
Zusatzinformation (ZI)	Festlegung der globalen Optimierungsrichtung der Transformation mit `OptimCriteria`, Subtypen von `Query` für lokale Designentscheidungen (z. B. bzgl. der technischen Systemarchitektur).
Bedingungen (BED)	Formulierung von bedingten Ausdrücken (`CondExpression`) und bedingten Parameterbelegungen (`CondParamBinding`).
Verfolgbarkeitsinformationen (TRC)	Anlegen der `DesignTrace`-Beziehung durch `TemplateInstantiation` und Speicherung der Designentscheidungen entsprechend des XML-Schemas im Attribut `traceInfo`.
Zugriff auf bereits erzeugte Elemente im Zielmodell (ZBE)	Geordnete Reihenfolge der `RHS` und die Möglichkeit, erzeugte Template-Instanzen über `InstanceVariable` anzusprechen.
Iterationsmechanismus für Mengen von Modellelementen (IT)	Attribute `iterateFor` in `TemplateInstantiation` und `iterate` in `TrafoParamBinding`.

Tabelle 6.1: Abgleich der Lösung mit den gestellten Anforderungen aus Kapitel 3.2.5

6.2 Historie der Lösung

Die im Lösungsansatz dieser Arbeit vorgestellten Konzepte wurden durch drei Diplomarbeiten überprüft. Grundlage für die Evaluierung war bei jeder Arbeit das in Kapitel 2 als Fallstudie vorgestellte Kombiinstrument- und Fahrfunktionen-System (KFS).

Im Rahmen der ersten Arbeit [Kolodziejczyk '05] wurde das KFS auf Basis der im Rahmen von POTAD verwendeten ROPES-Methode entwickelt, ohne den in Kapitel 4 vorgestellten Modelltransformationsmechanismus zu verwenden. Hierbei wurden in der Design- und auch der Analysephase intensiv Muster eingesetzt, wobei letztere eine Erweiterung der Methode darstellen. Ein Thema dieser Arbeit war es auch, Designmuster in Hinblick auf die Erfüllung unterschiedlicher nichtfunktionaler Anforderungen zu bewerten und auf dieser Basis die Variabilität im Designmodell bei wechselnden Anforderungen zu diskutieren. Das Ergebnis enthält im Analysemodell 5 Muster und das letztendlich gewählte Designmodell 12 Muster. Dieses Designmodell diente für die Verifikation des Modelltransformationsansatzes aus Kapitel 4 als Referenzmodell. Hier sollte der Nachweis erbracht werden, dass sich dieses manuell erstellte Referenzmodell zu großen Teilen auch mit der prototypischen Implementierung aus Kapitel 5 erzeugen lässt.

Dieser Nachweis war – neben der Unterstützung bei der Konzeption der Transformationsregeln und der Implementierung des Transformators – Thema der zweiten Arbeit [Buchwald '05]. Nach den intensiven Untersuchungen einzelner Analysemuster und der Aufstellung entsprechender Transformationsregeln wurde mit Hilfe des Modelltransformators unter Verwendung desselben Analysemodells und derselben Anforderungen wie in der ersten Arbeit, ein Designmodell erzeugt. Der Vergleich mit dem Referenzmodell ergab, dass das erzeugt Modell in großen Teilen mit dem Referenzmodell übereinstimmt. Der Modellierer des Referenzmodells stufte das erzeugte Modell als ein verwendbares Grundgerüst ein. Es wurde jedoch beobachtet, dass in dem manuell erzeugten Referenzmodell zusätzliche Designmuster vorkommen, die sich nicht direkt mit einem Analysemuster in Verbindung bringen lassen, im Zusammenspiel mit den übrigen Designmustern aber sinnvoll erscheinen. Außerdem wurden manche Designmuster durch Regeln unterschiedlicher Analysemuster mehrfach angelegt. Hier war ein Optimierungsschritt notwendig, der einzelne Instanzen desselben Mustertyps zu einer Instanz vereinigte. Diese beiden Punkte sind Grenzen von POTAD, die in Kapitel 6.3 vertieft werden.

Die dritte Arbeit [Fischer '05] beschäftigte sich neben der Evaluation unterschiedlicher Werkzeugumgebungen und der Implementierung des Verfolgbarkeitsnavigators auch mit der Nutzung von POTAD in der *Detailed Design*-Phase. Die bisherigen Arbeiten hatten Designmusterbibliotheken verwendet, deren Muster zwar die Verwendung einer objektorientierten Programmiersprache voraussetzten, programmiersprachenspezifische Details aber offen lassen. Als Beispiele seien hier Unterschiede bei Java und C++ in Bezug auf Destruktoren und Interface-Konzepte genannt. Im Rahmen dieser Diplomarbeit wurde nun jeweils eine spezifische Variante der Designmusterbibliothek für Java und C++ erstellt. Es wurde außerdem die Möglichkeit von XDE genutzt, über den integrierten *Code Template-Mechanismus* Code-Gerüste für die Muster zu erstellen. Neben der Erzeugung von Methodenrümpfen, der richtigen Typdefinition bei Klassen, Attribute und Parametern, konnte für einzelne Beispiele auch aufgezeigt werden, dass sich auch Code für das charakteristische Verhalten des Musters erzeugen lässt.

Bei der Verwendung dieser programmiersprachenspezifischen Designmusterbibliotheken werden bereits einzelne Aktivitäten der *Detailed Design* Phase adressiert. Die im Rahmen des *Detailed Design* zu leistende genaue Auslegung von Algorithmen und Auswahl geeigneter Daten-

strukturen ist aber in Abhängigkeit unterschiedlicher Anforderungen wieder variabel, so dass hier unterschiedliche Varianten des Musters notwendig wären. Hier wurde nun der Ansatz untersucht, denselben Transformationsmechanismus, der bisher für eine Transformation zwischen Modellen der *Object Analysis* und dem *Mechanistic Design* verwendet wurde, nun für eine Transformation zwischen Modellen des *Mechanistic Design* und des *Detailed Design* zu verwenden. Dies wurde anhand des Designmusters `ControlLoop_Open` untersucht, bei dem der Steuerungsalgorithmus (Methode `step()`) häufig mit Kennlinien arbeitet. In einer Kennlinie werden Ansteuerungswerte für Aktuatoren in Abhängigkeit von gemessenen Sensorwerten abgelegt. Die abgespeicherten Werte werden Stützstellen genannt, dazwischenliegende Werte müssen interpoliert werden. Je nachdem, ob die Steuerung eine Interpolation benötigt, ob die Stützstellen feste Abstände haben und wie das zu erwartende Abfrageverhalten ist (z. B. ob aufeinander folgende Abfragen häufig nah beieinanderliegende Sensorwerte enthalten) werden unterschiedliche Suchalgorithmen zum Auffinden von Stützstellen und unterschiedliche Datenstrukturen zur Ablage der Kennlinie verwendet. Diese Alternativen wurden als Muster erfasst, wobei diese hauptsächlich aus Code Templates bestehen und kaum UML-Elemente enthalten. Auch für diese Muster konnte eine Transformationsregel spezifiziert werden, die diese Muster in Abhängigkeit von Designentscheidungen instanziiert.

Somit konnte mit dieser Arbeit zum einen gezeigt werden, dass bei der Formulierung der Transformationssregeln von der Programmiersprache abstrahiert werden kann, da programmiersprachenspezifische Details über die austauschbaren Muster eingebracht werden können. Zum anderen wurde für ein Beispiel gezeigt, dass sich der Transformationsmechanismus auch für die *Detailed Design*-Phase eignet.

6.3 Grenzen des Ansatzes

Der in dieser Arbeit vorgestellte Ansatz POTAD hat sich in den beschriebenen studentischen Projekten (siehe Kapitel 6.2) und im Umfeld der Fallstudie (siehe Kapitel 2) bewährt. Die Grenzen von POTAD ergeben sich einerseits aus den allgemeinen Besonderheiten von Modelltransformationen selbst und andererseits aus den Erfahrungen der bisher durchgeführten Projekte mit der hier vorgestellten Lösung.

- Die Erfassung von Mustern nach dem Schema von POTAD und die Formulierung von Transformationsregeln sind mit anfänglichem Aufwand verbunden. Dieser Aufwand amortisiert sich, wenn Muster und Transformationsregeln wiederverwendet werden. Eine Wiederverwendung kann i. d. R. realisiert werden, wenn die Entwicklung mehrerer „ähnlicher“ Funktionen (Verwendung derselben Analysemuster) geplant ist und die Zielplattform sich nicht wesentlich ändert. Eine empirische Untersuchung, die die Frage beantworten könnte, ab wie vielen Anwendungen einer Transformationsregel eine Amortisierung der anfänglichen Kosten erreicht wird, war im Rahmen dieser Arbeit nicht möglich.
- POTAD wurde anhand des Fallbeispiels *Kombiinstrument- und Fahrfunktionen-System* (KFS) evaluiert. Die Auswahl der im KFS enthaltenen Funktionen wurde so gewählt, dass möglichst unterschiedliche Funktionseigenschaften und Subsysteme eines Fahrzeugs (siehe

Kapitel 3.1.1.f) vertreten sind. Trotz dieser unterschiedlichen Funktionen, bei deren Entwicklung POTAD erfolgreich eingesetzt wurde, wird in dieser Arbeit nicht der Anspruch erhoben, POTAD für alle Softwarefunktionen im Fahrzeugumfeld einsetzen zu können. Auch zu der Einsetzbarkeit in anderen Domänen kann an dieser Stelle keine allgemeine Aussage getroffen werden (für grundsätzliche Überlegungen sei jedoch auf Kapitel 7.1 verwiesen). Durch die Möglichkeit, eigene Muster zu beschreiben und Transformationsregeln mit Hilfe der beschriebenen Syntax selbst zu definieren, kann POTAD jedoch grundsätzlich flexibel auf den eigenen Anwendungskontext angepasst werden.

- Mit einer POTAD-Transformationsregel kann ein optimiertes Design für ein Analysemuster beschrieben werden. Umfasst das Analysemodell mehrere Analysemusterinstanzen, werden die Transformationsregeln seriell hintereinander ausgeführt und erzeugen ein in Grenzen zusammenhängendes Designmodell. Wie bei der Evaluierung in Kapitel 6.2 beobachtet wurde, können hier einzelne Muster zusammengefasst werden. Außerdem können einzelne Analysemuster-Kombination neue Designmuster sinnvoll erscheinen lassen, die durch die Transformationsregeln nicht erzeugt werden können. Ein von POTAD erzeugtes Designmodell muss somit i. d. R. in einem gewissen Umfang manuell nachbearbeitet werden. Durch die Integration von Musterkompositionstechniken, wie z. B. bei [Yacoub et al. '04] beschrieben, könnte diese Aktivität evtl. vereinfacht oder sogar automatisiert werden.
- POTAD verwendet für die unterschiedlichen Designvarianten eines Analysemusters Bedingungen, die an beliebiger Stelle einer Transformationsregel vorkommen können. Unter anderem kann die Instanziierung eines Musters von einer Bedingung abhängig gemacht werden. Soll dieses Muster später mit einem weiteren Muster verwoben werden, muss geprüft werden, ob das Muster instanziiert wurde oder nicht. Bei sehr viel Variabilität werden die Regeln durch die vielen Fallunterscheidungen daher unübersichtlich und schlecht wartbar. Mit dem Konzept der Hilfsregeln wurde eine erste Maßnahme getroffen, Transformationsregeln in konsistente und wiederverwendbare Einheiten zu zerlegen. Das Problem wird damit jedoch nicht vollständig beseitigt.
- Mit der prototypischen Implementierung lassen sich die Transformationsregeln nur eingeschränkt im Kontext einer iterativen Vorgehensweise nutzen. Ein durch Modelltransformation erzeugtes Designelement, dessen Name nachträglich manuell geändert wird, wird bei erneuter Ausführung der Transformation nicht wieder gefunden und daher neu erzeugt. Hierbei handelt es sich jedoch nur um eine Schwäche des Prototyps – eine Ausbaustufe könnte die angelegten Verfolgbarkeitsinformationen auswerten und so das erzeugte Element finden und wiederverwenden.

7 Zusammenfassung und Ausblick

Diese Arbeit bewegt sich im Umfeld von modellgetriebenen System- und Softwareentwicklungsansätzen, deren grundlegende Idee eine durchgängige Nutzung von formalisierten Modellen in den Entwicklungsphasen ist. Auf dieser Basis werden mit Hilfe von Modelltransformationen Modelle unterschiedlicher Entwicklungsphasen automatisiert ineinander überführt und deren Modellelemente so miteinander verknüpft, dass ihr Zusammenhang im Nachhinein systematisch verfolgbar ist.

Im Rahmen dieser Arbeit wird überprüft, inwiefern sich diese Ansätze für die Integration von System- und Softwareentwicklung bei der Entwicklung von softwareintensiven Fahrzeugfunktionen nutzen lassen. Zielstellung ist es, die Modelle der Systementwicklung als Analysemodell im Rahmen der Softwareentwicklung wiederzuverwenden und mit Hilfe von Modelltransformationen automatisiert in eine Vorlage für ein Softwaredesignmodell zu überführen.

Die spezifischen Anforderungen, die für dieses Zielszenario an einen Modelltransformationsmechanismus gestellt werden müssen, wurden aus den grundlegenden Anforderungen der Domäne, den herkömmlichen Methoden der System- und Softwareentwicklung und den einsetzbaren Modellen abgleitet. Mit dem Kernprozess und ROPES bzw. der EAST-ADL und der UML wurden zunächst zwei Referenzmethoden bzw. Modellierungssprachen für die System- und Softwareentwicklung identifiziert. In diesem Kontext lassen sich die Kernanforderungen dann wie folgt zusammenfassen: Eine Modelltransformation muss für eine Funktion der logischen Systemarchitektur ein Softwaredesign erzeugen und dies anhand unterschiedlicher nichtfunktionaler Anforderungen und der technischen Systemarchitektur variieren können.

Bei den Untersuchungen erwiesen sich Muster als eine geeignete Grundlage, um den Zusammenhang zwischen Analyse- und Designmodellen unter Berücksichtigung dieser Variabilität zu systematisieren. Eine im ersten Teil der Arbeit nur grob skizzierte Systematik basiert im Kern darauf, für eine Analysemusterinstanz in Abhängigkeit von nichtfunktionalen Anforderungen und der Zielplattform Designmuster zu instanziieren, wobei die Parameter der Designmuster mit Inhalten des Analysemusters belegt werden. Die sich daraus ergebenden Verknüpfungen zwischen Analyse- und Designmusterinstanzen lassen sich auch für Verfolgbarkeitsfragen nutzen. Aus dieser Systematik leiten sich weitere Anforderungen an den Modelltransformationsmechanismus ab, insbesondere in Bezug auf den Umgang mit Mustern und die Erzeugung geeigneter Verknüpfungen für die nachträgliche Verfolgung von Designentscheidungen. Bei der Untersuchung des Stands der Technik konnte kein Ansatz für Modelltransformationen gefunden werden, der diese Kernanforderungen in befriedigender Weise adressiert.

Aufgrund der erkannten Mängel wurde der hier vorgestellte Ansatz POTAD entwickelt. Er besteht aus der Erweiterung der Softwareentwicklungsmethode ROPES, einem erweiterten Template-Metamodell der UML2, einer neuen Modelltransformationssprache und einem Verfolgbarkeitsmodell.

Die ROPES-Methode ist bei POTAD an den Kernprozess und die Modelllandschaft der Systementwicklung angepasst und die Analysephase um den Einsatz von Analysemustern ergänzt. Außerdem werden das Konzept der Modelltransformation als Designschritt und die Nutzung der Verfolgbarkeitsinformationen im Prozess verankert. Das Template-Metamodell der UML2 ist in der Weise ergänzt, dass die Beschreibung und Instanziierung von Mustern für eine Transformation ausreichend formalisiert ist. Damit kann nun z. B. das Verschmelzungsverhalten für jedes Template-Element festgelegt werden, Parameter gruppiert und hinsichtlich Multiplizität genauer spezifiziert werden. Die Transformationssprache basiert auf der zuvor grob skizzierten Systematik. In einer textuellen Syntax kann für eine gefundene Analysemusterinstanz, in Abhängigkeit von nichtfunktionalen Anforderungen und der Zielplattform, die Instanziierung von Designmustern angestoßen werden. Die dabei angelegten Verfolgbarkeitsinformationen verknüpfen die Analyse- und Designmusterinstanzen und dokumentieren eine getroffene Auswahl zwischen Designalternativen.

Eine prototypische Implementierung wurde als Erweiterung der UML-Entwicklungsumgebung *Rational XDE* realisiert. Ein eigener Interpreter kann POTAD-Transformationsregeln verarbeiten, in dem er im Analysemodell nach Analysemusterinstanzen sucht und entsprechend der Regel-Anweisungen Designmuster instanziiert. Im Rahmen von studentischen Arbeiten wurden einige Transformationsregeln entwickelt und erfolgreich anhand einer Fallstudie erprobt.

7.1 Zukünftige Arbeiten

Der in dieser Arbeit vorgestellte Ansatz POTAD wurde in einer Fallstudie eingesetzt und hat Grenzen, die in Kapitel 6.3 aufgezeigt wurden. Zur Erweiterung dieser Grenzen und Verbesserung von POTAD sind weitere Arbeiten notwendig.

- Bei der Entwicklung von POTAD wurde davon ausgegangen, dass die betrachteten Systeme „auf der grünen Wiese“ nach Top-down-Vorgehensweise neu entwickelt werden. Es ist zu prüfen, wie POTAD genutzt werden kann, wenn ein bestehendes System zu dem Modelle nur unvollständig vorliegen, nach Bottom-up-Vorgehensweise erweitert werden soll. Evtl. ist es möglich, die Transformationsregeln so zu erweitern, dass sie das Design anhand der Frage variieren, ob für eine bestimmte Teilfunktionalität eine existierende Implementierung eingebunden werden muss oder nicht. Im ersten Fall würden dann entsprechende Designmuster für die Adaptierung und die Integration in das Gesamtdesign in der Transformationsregel instanziiert.
- POTAD wurde anhand einer Fallstudie aus der Domäne *Automotive Software* überprüft. Eine breitere Evaluierung, vor allem auch in anderen Domänen, steht noch aus. Die Anwendbarkeit in einer anderen Domäne wird grundsätzlich als positiv eingeschätzt, wenn folgende Bedingungen erfüllt sind:
  1. In der Domäne wird bereits ein formalisiertes Fachmodell erstellt. Dieses lässt sich mit den Mitteln der UML modellieren.

2. In diesem Fachmodell lassen sich wiederkehrende Muster finden. Idealerweise sind diese schon bekannt und dokumentiert.
3. Damit sich die Erstellung der Transformationsregeln lohnt, muss die Entwicklung mehrerer Systeme geplant sein, die im Fachmodell immer wieder dieselben Muster nutzen.

- Die Variabilität des Designs wird bei POTAD durch einfache Bedingungen und Fallunterscheidungen in den Transformationsregeln realisiert. Bei umfangreicher Variabilität führt dies zu schwer lesbaren und schwer wartbaren Transformationsregeln. Hier ist zu prüfen, wie Transformationsregeln besser in konsistente und wiederverwendbare Einheiten zerlegt werden können. Evtl. könnten dabei Konzepte aus dem Umfeld der Generativen Programmierung [Czarnecki et al. '00] eine Verbesserung bringen.
- Die maschinelle Abfrage der technischen Systemarchitektur wurde zwar im Metamodell berücksichtigt, in der prototypischen Umsetzung jedoch nicht implementiert. Stattdessen werden die benötigten Informationen durch Benutzerrückfragen erfragt. Es ist zu vermuten, dass sich diese Benutzerrückfragen durch die systematische Auswertung der technischen Systemarchitektur reduzieren lassen. Arbeiten zu den entsprechenden Abfragen und Auswertungsmechanismen stehen noch aus.
- POTAD wurde zu einem Zeitpunkt entwickelt, zu dem noch kein Standard für eine Transformationssprache absehbar war. Mittlerweile steht *Query/Views/Transformations* (QVT) kurz vor der finalen Verabschiedung als OMG-Standard. Eine Untersuchung der beiden letzten Einreichungen zu QVT hat ergeben, dass auch der sich jetzt durchsetzende Ansatz nicht alle erarbeiteten Anforderungen erfüllt. Wie eine exemplarische Prüfung ergeben hat, lässt sich eine POTAD-Transformationsregel nur sehr umständlich und mit vielen Nachteilen in QVT nachbilden. Sobald der Standard aber verabschiedet ist und entsprechende Werkzeuge verfügbar sind, erscheint es trotzdem sinnvoll zu prüfen, inwiefern POTAD in QVT integriert werden kann. Vermutlich ist im Bereich der Behandlung von Mustern eine Erweiterung notwendig. Der Vorteil einer Integration wäre jedoch, dass sich in einer Transformationsregel neben POTAD-Konzepten, auch andere Transformationsansätze nutzen ließen.
- Im Rahmen der POTAD-Transformationsregeln wurden bisher nur Muster verwendet, die ein Klassenmodell beschreiben. In [Konrad et al. '04b] umfasst ein Analysemuster aber auch Sequenz- und Zustandsmodelle. In dem Werkzeug *Rational XDE* lassen sich für Muster ebenfalls Sequenz- und Zustandsmodelle beschreiben, die ebenso wie Klassenmodelle parametrisiert und instanziiert werden können. In [Douglass '99] werden zudem zahlreiche Muster für Zustandsmodelle beschrieben. Hier ist zu prüfen, ob auf Basis solcher Muster auch Verhaltensmodelle transformiert werden können.
- Die Designmuster im Beispielkatalog weisen inhaltliche Überlappungen auf. Hier ist zu prüfen, inwiefern die Muster in „atomare“ Muster zerlegt werden können und ob dies Vorteile bei der Formulierung der Transformationsregeln bringt.

- Template-Instanzen erhöhen das Datenaufkommen eines Modells. Es ist daher zu prüfen, inwiefern das Metamodell für eine Template-Bindung in Richtung Speicherverbrauch optimiert werden kann.
- Die im Rahmen dieser Arbeit gezeigten Transformationsregeln erzeugen ein Design, das sich auf „Expertenwissen“ begründet. Es gibt keinen formalen Nachweis, dass das erzeugte Design mit den Eigenschaften des Analysemusters kompatibel ist. Eine Prüfung, ob und wie ein solcher Nachweis für eine POTAD-Transformationsregel erbracht werden kann, steht noch aus. Zunächst müssten die einzuhaltenden Eigenschaften eines Analysemusters formal erfasst werden. Ein Ansatzpunkt könnten hier die bei [Konrad et al. '04b] beschriebenen Constraints für Analysemuster auf Basis temporaler Logik sein.
- POTAD setzt UML-basierte Modellierungswerkzeuge voraus. Obwohl UML bereits in vielen Domänen etabliert ist und es auch spezialisierte UML-Werkzeuge und Methoden für eingebettete Systeme gibt, werden im Automobilbereich oft noch proprietäre Modellierungssprachen verwendet. Um POTAD auch in diesem Kontext zu verwenden, ist eine Anpassung des Mustermetamodells an die Modellierungssprache notwendig.

Anhang – A: Muster

A.1 Analysemuster

1 ClosedLoopControl

1.1 Zweck

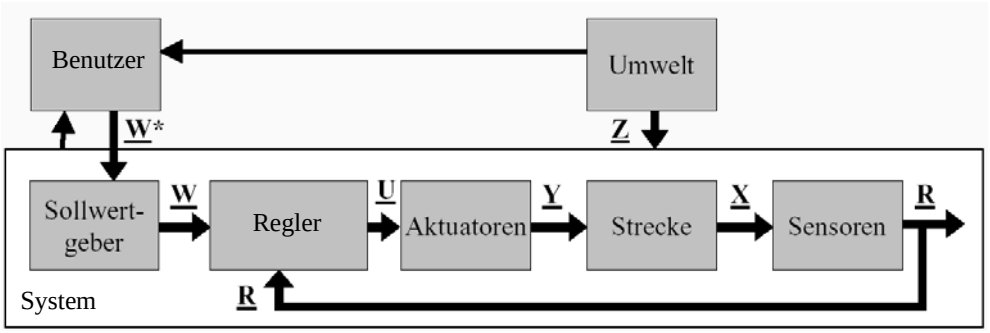
Einen geschlossenen Regelkreis (engl. *feedback control*) beschreiben.

1.2 Motivation

Viele Funktionen im Fahrzeug haben regelungstechnischen Charakter. Nach DIN 19226-1 [DIN '94] wird eine Regelung wie folgt definiert:

Das Regeln, die Regelung, ist ein Vorgang, bei dem fortlaufend eine Größe, die Regelgröße (die zu regelnde Größe) erfasst, mit einer anderen Größe, der Führungsgröße, verglichen und im Sinne einer Angleichung an die Führungsgröße beeinflusst wird. Kennzeichen für das Regeln ist der geschlossene Wirkungsablauf, bei dem die Regelgröße im Wirkungskreis des Regelkreises fortlaufend sich selbst beeinflusst.

Das folgende Blockdiagramm zeigt die beteiligten Komponenten und Größen (angelehnt an [Schäuffele et al. '03]).

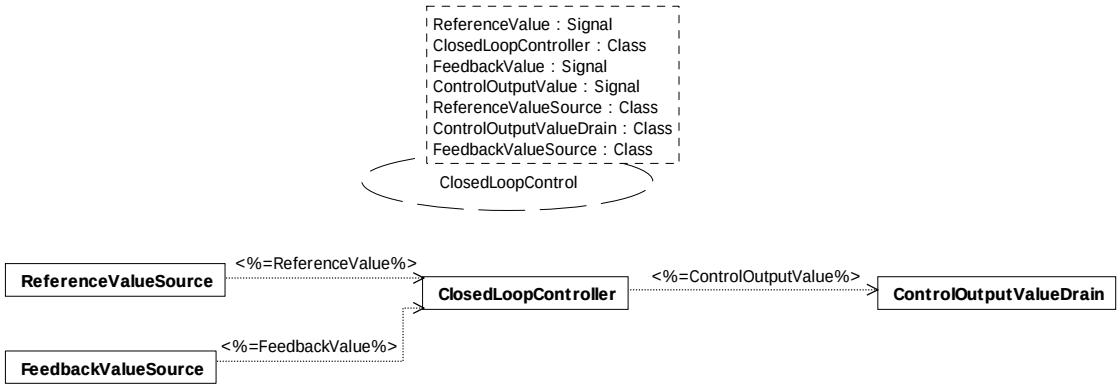


- R = Mess- oder Rückführgröße
- U = Ausgangs- oder Stellgröße des Reglers
- W = Führungs- oder Sollgröße
- W* = Sollwert des Benutzers
- X = Regelgröße
- Y = Stellgröße
- Z = Störgröße(n)

Der geschlossene Wirkungsablauf wird dadurch hergestellt, dass es sich bei der Führungs- und der Rückführgröße um denselben Typ handelt. Damit wird ein fortlaufender Vergleich von Soll- und Istwert möglich und die Berücksichtigung einzelner Störgrößen überflüssig.

Ein übliches Verfahren für die Berechnung der Ausgangsgröße ist *Proportional-Integral-Differential Control* (PID) [Pont '01].

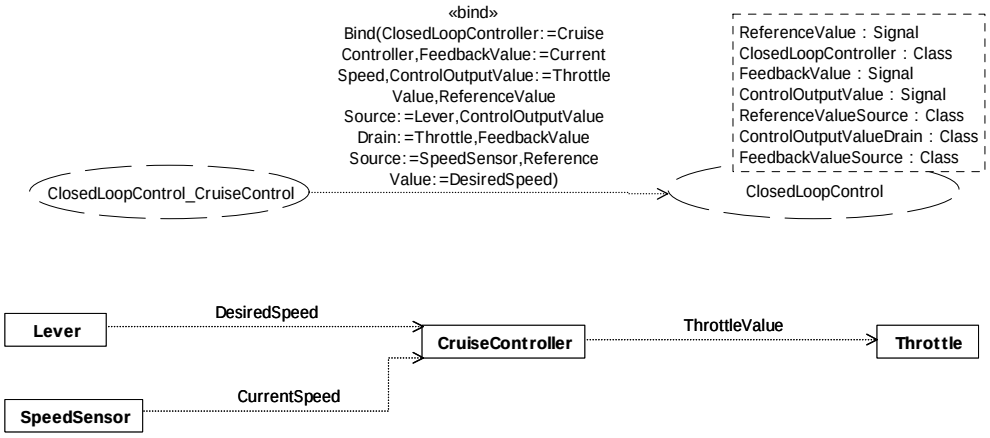
1.3 Struktur und Informationsfluss



- **ReferenceValueSource**: Sendet die Sollgröße `ReferenceValueSource` an `ClosedLoopController`.
- **FeedbackValueSource**: Sendet die Rückführgröße `ControlInputValue` an `ClosedLoopController`.
- **ClosedLoopController**: Vergleicht fortlaufend `ReferenceValue` sowie `FeedbackValue` und berechnet für die Angleichung einen entsprechenden Wert für `ControlOutputValue`.
- **ControlOutputValueDrain**: Empfängt `ControlOutputValue`.

1.4 Beispiel

Ein Beispiel für eine Regelung ist ein klassischer Tempomat: Der Fahrer gibt über ein Bedienelement (`ReferenceValueSource:=Lever`) eine Sollgeschwindigkeit (`ReferenceValue:=DesiredSpeed`) vor. Die Istgeschwindigkeit (`FeedbackValue:=CurrentSpeed`) wird laufend über den Drehzahlsensor der Räder (`FeedbackValueSource:=SpeedSensor`) ermittelt und mit dem der Sollgeschwindigkeit verglichen. Wird eine Abweichung festgestellt, wird ein neuer Wert (`ControlOutputValue:=ThrottleValue`) für die Ansteuerung der Drosselklappe (`ControlOutputValueDrain:=Throttle`) berechnet, der mit einem konkreten Öffnungswinkel korrespondiert. Durch die Rückkopplung über den Drehzahlsensor wird die Sollgeschwindigkeit auch bei Steigungen oder größerem Windwiderstand nahezu gehalten.



Weitere Beispiele für Regelungen im Kfz sind [BOSCH '03]: Lambda-Regelung, Drehzahlregelung bei Dieselmotoren, Anti-Blokier-System (ABS) und Klimaanlage.

1.5 Anwendbare Designmuster

- `Repository` (Wiederverwendung): Die Sensorwerte in einer Klasse kapseln und so die eigentliche Regelung von der Datenbeschaffung entkoppeln.
- `DataBus`: Für den Fall, dass die Sensordaten über einen Datenbus empfangen werden, wird der Zugriff auf diesen in einer einheitlichen Objektstruktur gekapselt.
- `ReactiveControlLoop`: Für den Fall, dass die Regelung zustandsbehaftet ist, werden die Regelungsalgorithmen für die unterschiedlichen Zustände in separaten Objekten gekapselt.
- `ControlLoop`: Der Regelungsalgorithmus wird in einem separaten Objekt gekapselt.
- `PublisherSubscriber` (Wiederverwendung): Mögliche zukünftige „Datenkonsumenten“ der Sensoren können einfach durch Registrierung als Abonnent hinzugefügt werden.
- `Strategy` (Wiederverwendung): Der Regelungsalgorithmus kann leicht ausgetauscht werden.
- `Template` (Wiederverwendung): Gibt eine für Regelungen typische Algorithmenstruktur vor: `calcControlDeviation` berechnet die Regelabweichung, `calcControlValue` berechnet den Stellwert. Es wird eine konkrete Klasse generiert, die diese Algorithmen implementiert. Weitere Implementierungen können durch Subtypbildung hinzugefügt werden.
- `HeterogeneousRedundancy` (Sicherheit): Heterogene Redundanz der Sensorik/Aktuatorik und des Reglers unterstützen.
- `SecondGuess` (Sicherheit): Den in der Regelschleife verwendeten Algorithmus durch einen kürzeren Algorithmus absichern und beide Algorithmen in separaten Objekten kapseln.
- `Watchdog` (Sicherheit): Periodisch prüfen, ob die Regelschleife noch aktiv ist.

2 OpenLoopControl

2.1 Zweck

Eine Steuerkette (engl. *feedforward control*) beschreiben.

2.2 Motivation

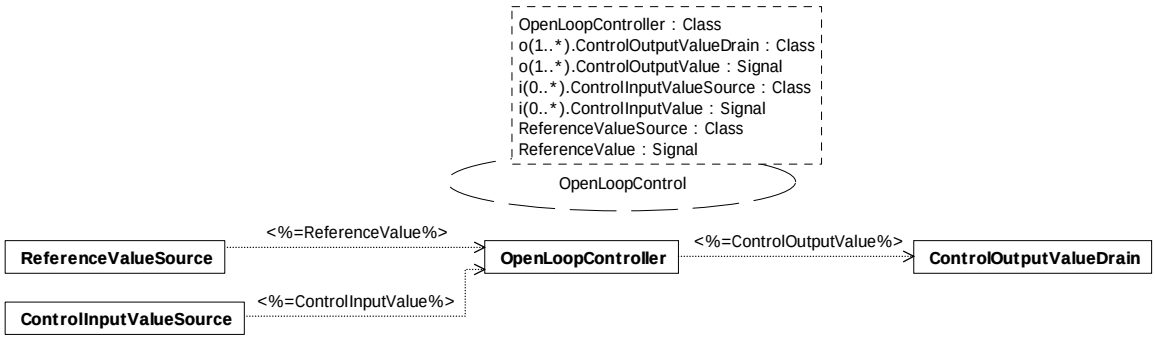
Viele Funktionen im Fahrzeug haben steuerungstechnischen Charakter. Nach DIN 19226-1 [DIN '94] wird eine Steuerung wie folgt definiert:

Die Steuerung ist der Vorgang in einem System, bei dem eine oder mehrere Größen als Eingangsgrößen andere Größen als Ausgangsgrößen aufgrund der dem System eigentümlichen Gesetzmäßigkeiten beeinflussen. Kennzeichen für das Steuern ist der offene Wirkungsweg, bei dem die durch die Eingangsgrößen beeinflussten Ausgangsgrößen nicht fortlaufend und nicht wieder über dieselben Eingangsgrößen auf sich selbst wirken.

Im Gegensatz zum Muster `ClosedLoopControl` kann es hier mehrere Ein- und Ausgangsgrößen geben, wobei es sich bei den Eingangsgrößen definitionsgemäß nicht um eine Rückführgröße handeln darf. Somit werden bei der Steuerung auch nicht alle Störgrößen berücksichtigt.

Übliche Ansätze bei der Steuerungsfunktion sind einfache *Lookup*-Tabellen oder mathematische Funktionen [Pont '01].

2.3 Struktur und Informationsfluss

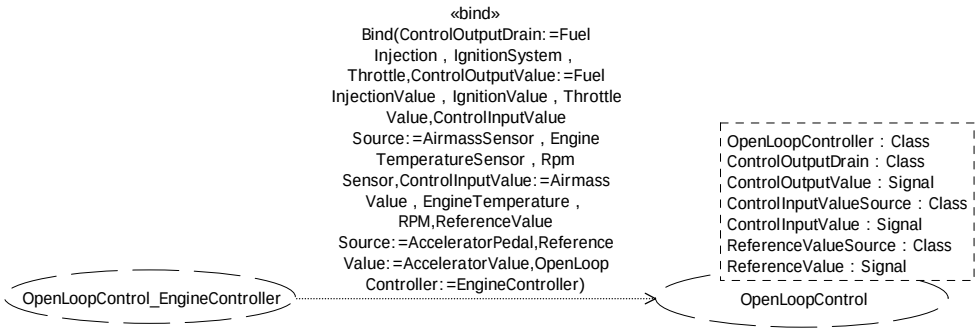


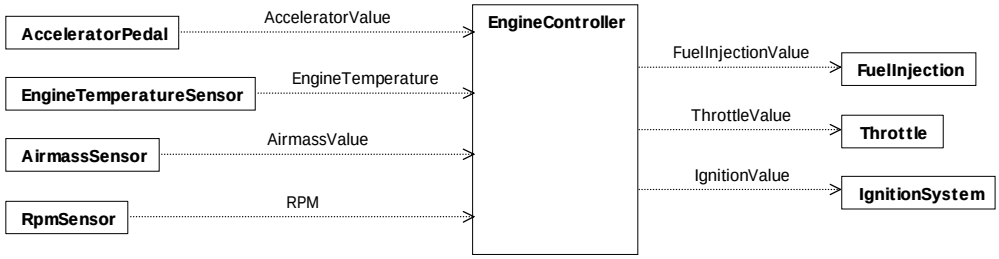
- **ReferenceValueSource**: Sendet die Sollgröße `ReferenceValueSource` an `OpenLoopController`.
- **ControlInputValueSource**: Weitere für die Steuerung relevante Informationsquellen, die über das Signal `ControlInputValue` Umgebungsinformationen zur Verfügung stellen.
- **OpenLoopController**: Berechnet auf der Grundlage von `ReferenceValue` und `ControlInputValue` das Signal `ControlOutputValue`.
- **ControlOutputValueDrain**: Empfängt `ControlOutputValue`.

2.4 Beispiel

Ein Beispiel für eine Steuerung ist ein elektronisches Motormanagement. Der Fahrer kann über das Gaspedal (`ReferenceValueSource:=AcceleratorPedal`) dem System den gewünschten Vortrieb anzeigen. Die Motor-Steuerung muss dazu die Aktuatoren für die Einspritzung, die Drosselklappe und das Zündsystem (`ControlOutputDrain:=FuelInjection, Throttle, IgnitionSystem`) entsprechend steuern. Dazu benötigt sie Informationen bzgl. des aktuellen Betriebspunktes des Motors, die über Sensoren für die Luftmasse, die Motortemperatur und die aktuelle Drehzahl beschafft werden (`ControlInputSource:= EngineTemperatureSensor, AirmassSensor, RpmSensor`).

Das hier gezeigte Modell ist unvollständig, eine detaillierte Beschreibung aller Sensoren, Sollwertgebern und Aktuatoren einer Motorsteuerung findet sich bei [BOSCH '03].





2.5 Anwendbare Designmuster

- `Blackboard` (Wiederverwendung): Die Sensorwerte in einer Klasse kapseln, über eine generische Schnittstelle lesen und schreiben und so die eigentliche Steuerung von der Datenbeschaffung entkoppeln.
- `DataBus`: Für den Fall, dass die Sensordaten über einen Datenbus empfangen werden, wird der Zugriff auf diesen in einer einheitlichen Objektstruktur gekapselt.
- `ReactiveControlLoop`: Für den Fall, dass die Steuerung zustandsbehaftet ist, werden die Steuerungsalgorithmen für die unterschiedlichen Zustände in separaten Objekten gekapselt.
- `ControlLoop`: Der Steuerungsalgorithmus wird in einem separaten Objekt gekapselt.
- `PublisherSubscriber` (Wiederverwendung): Mögliche zukünftige „Datenkonsumenten“ der Sensoren können einfach durch Registrierung als Abonnent hinzugefügt werden.
- `Strategy` (Wiederverwendung): Der Steuerungsalgorithmus kann leicht ausgetauscht werden.
- `HeterogeneousRedundancy` (Sicherheit): Heterogene Redundanz der Sensorik/Aktuatorik und des Reglers unterstützen.
- `SecondGuess` (Sicherheit): Den in der Steuerung verwendeten Algorithmus durch einen kürzeren Algorithmus absichern und beide Algorithmen in separaten Objekten kapseln.
- `Watchdog` (Sicherheit): Periodisch prüfen, ob die Steuerung noch aktiv ist.

3 UserInterface

3.1 Zweck

Abstrakte Beschreibung einer Benutzerschnittstelle.

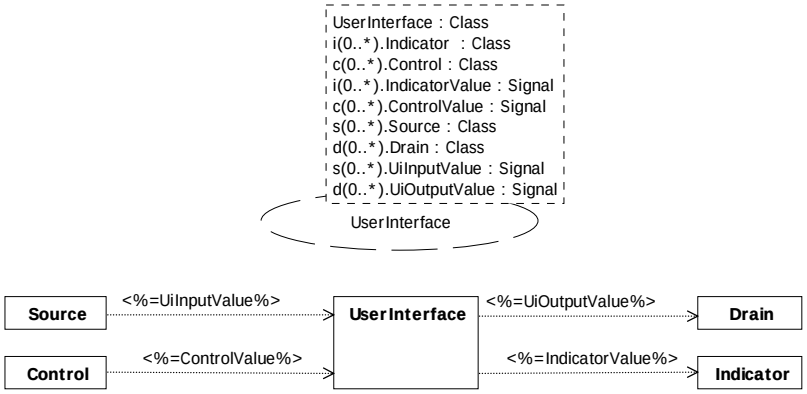
3.2 Motivation

Einige Steuerungs- und Regelungssysteme benötigen Interaktion mit dem Fahrer oder anderen Insassen. So muss z. B. der Sollwert für eine Regelung eingestellt oder aktuelle Sensorwerte angezeigt werden. Eine weitere wichtige Aufgabe der Benutzerschnittstelle ist das Anzeigen von Fehlern.

Bei sehr einfachen Benutzerschnittstellen können die Bedien- und Anzeigeelemente als Sensoren bzw. Aktuatoren modelliert werden. Bei komplexeren Benutzerschnittstellen gibt es keine direkte Verbindung zu den Steuerungs- und Regelungsfunktionen, sondern die Werte der Bedien- und Anzeigeelemente werden Vor- bzw. Nachbearbeitet. So kann die Benutzerschnittstelle z. B. zustandsbehaftet sein und je nach Zustand einzelne Elemente aktivieren oder deaktivieren.

Auf der Analyseebene wird von der konkreten technischen Realisierung der Benutzerschnittstelle abstrahiert. Im Fokus stehen lediglich die dem Benutzer zur Verfügung stehenden Bedien- und Anzeigeelemente sowie die entsprechenden Ein- und Ausgangssignale zwischen der Benutzerschnittstelle und den Steuerungs- und Regelungsfunktionen.

3.3 Struktur und Informationsfluss

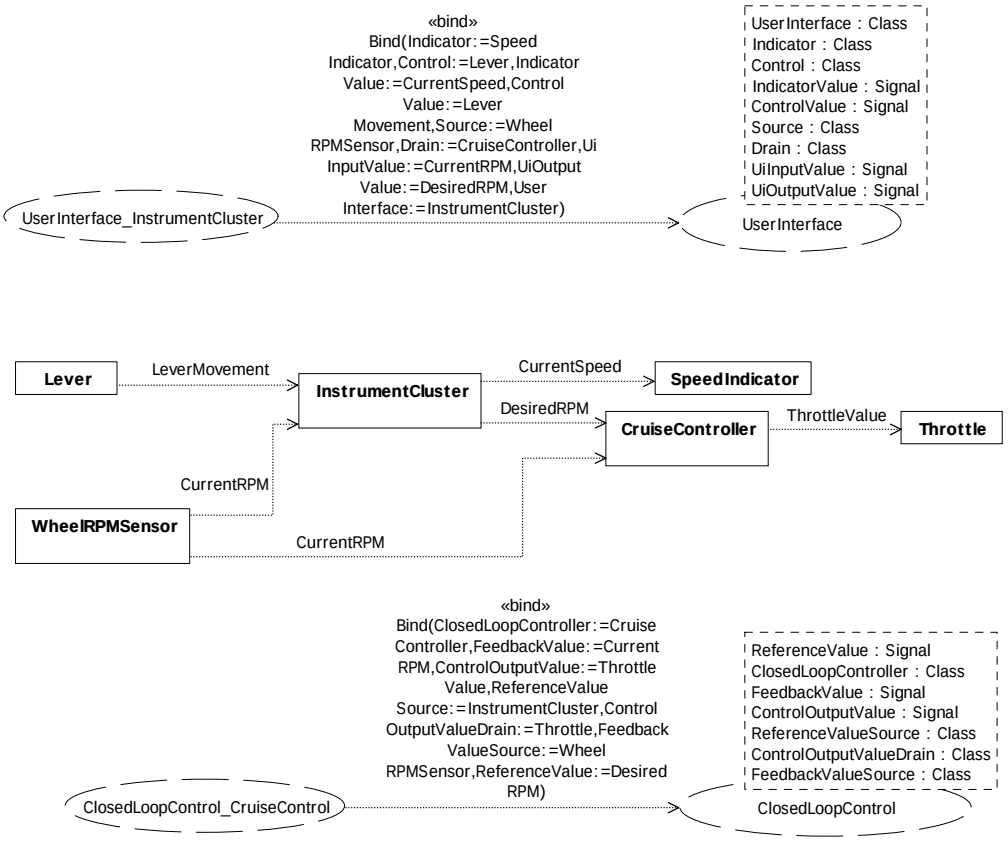


- **Source**: Eine Steuerungs- und Regelungsfunktion, deren Signal `UiInputValue` an der Benutzerschnittstelle angezeigt werden soll.
- **Drain**: Eine Steuerungs- und Regelungsfunktion, die das über die Benutzerschnittstelle erfasste Signal `UiOutputValue` verarbeitet.
- **UserInterface**: Vermittelt zwischen den Bedien- und Anzeigeelementen und den Steuerungs- und Regelungsfunktionen.
- **Indicator**: Anzeigeelement, das den von `UserInterface` empfangenen Wert `IndicatorValue` anzeigt.

- **Control**: Bedienelement, das den vom Benutzer eingestellten Wert `ControlValue` an `UserInterface` sendet.

3.4 Beispiel

Das Beispiel zeigt eine Erweiterung des Tempomaten (Beispiel zu Muster `ClosedLoopControl`) um ein Kombiinstrument (`UserInterface:=InstrumentCluster`). Dieses umfasst einen Hebel für die Einstellung der Sollgeschwindigkeit (`Control:=Lever`) und eine Anzeige der aktuellen Geschwindigkeit (`Indicator:=SpeedIndicator`). Die Funktion `InstrumentCluster` übersetzt die Bewegungen des Hebels (hoch, runter, vor, zurück) in eine Sollgeschwindigkeit (`DesiredRPM`) für `CruiseController` um. `InstrumentCluster` ist darüber hinaus auch für die Umrechnung Radrehzahl (`CurrentRPM`) in km/h (`CurrentSpeed`) verantwortlich. Auf diese Weise wird der von `WheelRPMSensor` erfasste Wert über `SpeedIndicator` (digitale Anzeige) in der vom Fahrer gewünschten Größe angezeigt.



3.5 Quelle / Verwandte Muster

Das hier vorgestellte Muster geht zurück auf das Analysemuster *UserInterface* von [Konrad et al. '04a]. Neben der Präzisierung bzgl. Signale und Musterparameter ist das hier gezeigte Muster auch strukturell leicht erweitert. So handelt es sich bei der Klasse `UserInterface` dezidiert um eine eigene Funktion, die zwischen den Bedien- und Anzeigeelementen und den Steuerungs- und Regelungsfunktionen vermittelt. Bei [Konrad et al. '04a] gibt es eine direkte Verbindung.

3.6 Anwendbare Designmuster

- Blackboard (Wiederverwendung): Die für das User-Interface relevanten Daten in einem Objekt kapseln und so das User-Interface von der Datenbeschaffung entkoppeln.
- DataBus: Für den Fall, dass die Daten über einen Datenbus empfangen oder gesendet werden, wird der Zugriff auf diesen in einer einheitlichen Objektstruktur gekapselt.
- ModelViewController: Modell, Darstellung und Steuerung des User-Interfaces in separaten Objekten kapseln.
- Decorator (Wiederverwendung): Daten für die Darstellung aufbereiten.
- State: Zustandsbehaftetes Verhalten der User-Interface-Steuerung unterstützen.

4 FaultHandler

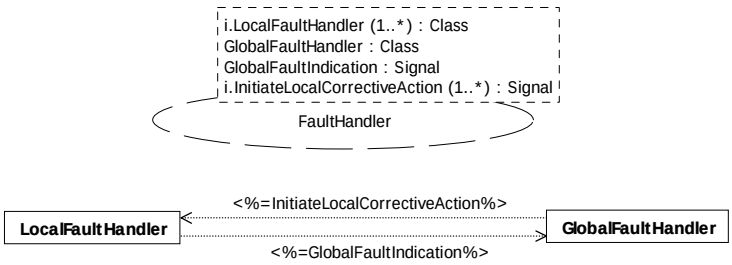
4.1 Zweck

Beschreibt eine hierarchische Fehlerbehandlung, bei der eine globale Fehlerbehandlungskomponente mehrere lokale Fehlerbehandlungskomponenten koordiniert.

4.2 Motivation

Ein Fehler in einer Funktion kann in manchen Fällen nicht durch lokale Maßnahmen behandelt werden. Die Sicherheitslogik sieht für solche Fehler Reaktionen vor, die die gesamte Fahrzeugelektronik betreffen können. Beispielsweise ist für manche Fehler ein Zustandswechsel in den sog. „Notlauf“ vorgesehen, bei der nur noch die absolut notwendigen Funktionen aktiv sind. Diese Art der Fehlerbehandlung muss zentral koordiniert werden.

4.3 Struktur

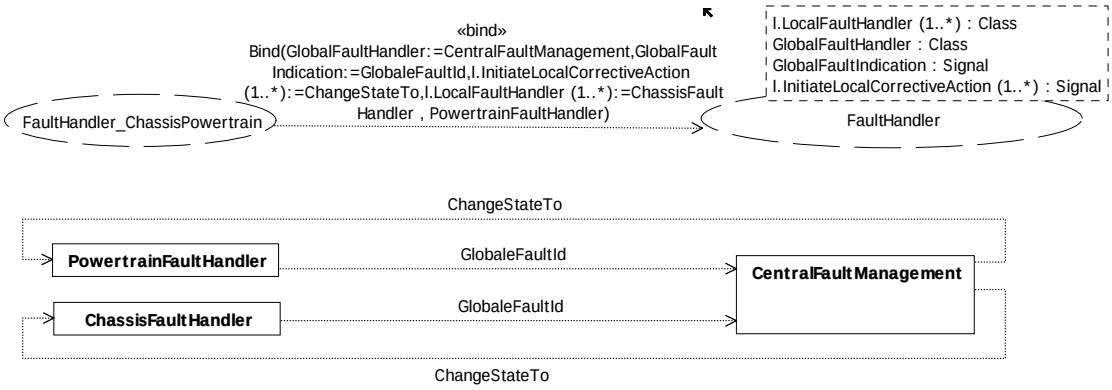


- **LocalFaultHandler**: Behandelt lokale Fehler und sendet ggf. GlobalFaultIndication an GlobalFaultHandler, wenn sich diese nicht lokal beheben lassen.
- **GlobalFaultHandler**: Führt nach dem Empfang von GlobalFaultIndication die globale Sicherheitslogik aus und sendet ggf. InitiateLocalCorrectiveAction an alle oder einzelne LocalFaultHandler.

4.4 Beispiel

Das Beispiel zeigt einen LocalFaultHandler der Chassis-Domäne und einen LocalFaultHandler der Powertrain-Domäne (LocalFaultHandler:= ChassisFaultHandler, PowertrainFaultHandler), die

beide durch eine globale Fehlerbehandlung (FlobalFaultHandler:= CentralFaultManagement) koordiniert werden. Die globale Fehlerbehandlung kann für die Durchsetzung der globalen Sicherheitslogik lokale Zustandswechsel über das Signal ChangeStateTo (InitiateLocalCorrectiveAction:= ChangeStateTo) veranlassen.



4.5 Quelle / Verwandte Muster

Das hier gezeigte Muster geht zurück auf das Analysemuster *FaultHandler* von [Konrad et al. '04a]. Die hier gezeigte Fassung ist hinsichtlich der Signale und Musterparameter erweitert worden.

4.6 Anwendbare Designmuster

- ExceptionMonitor: Fehler auf der Ebene einer Objektkollaboration erkennen und behandeln.
- Singleton: Sicherstellen, dass es nur eine Instanz der globalen Fehlerüberwachung und des globalen Logging-Mechanismus gibt.

5 DetectorCorrector

5.1 Zweck

Beschreibt ein generisches Überwachungskonzept, das korrigierende Maßnahmen initiiert, wenn sich das System oder einzelne Komponenten nicht mehr in einem gültigen Zustand befinden.

5.2 Motivation

Viele Softwarefunktionen im Fahrzeug unterliegen besonderen Sicherheits- und Zuverlässigkeitsanforderungen. Erfüllt eine solche Funktion ihre Aufgabe nicht mehr sicher oder zuverlässig, muss eine Reaktion nach einer festgelegten Sicherheitslogik erfolgen [Schäuffele et al. '03]. Da viele Überwachungskonzepte den kombinierten Einsatz von Hard- und Softwaremaßnahmen erfordern, müssen sie bereits in der Systementwicklung berücksichtigt werden.

Eine Überwachungsfunktion untergliedert sich in eine Fehlererkennungs- und eine Fehlerbehandlungsfunktion. Dabei kommen z. B. folgende Verfahren zu Einsatz [Schäuffele et al. '03]:

- Fehlererkennungsfunktionen:
  - Beobachten von physikalischen Eigenschaften

- Referenzwertüberprüfung
- Überprüfung anhand redundanter Werte
- Überprüfung der Datenintegrität (z. B. CRC, Hemming-Codes)
- Senden von Bestätigungen
- Beobachtung des zeitlichen Ablaufs (Watchdog)

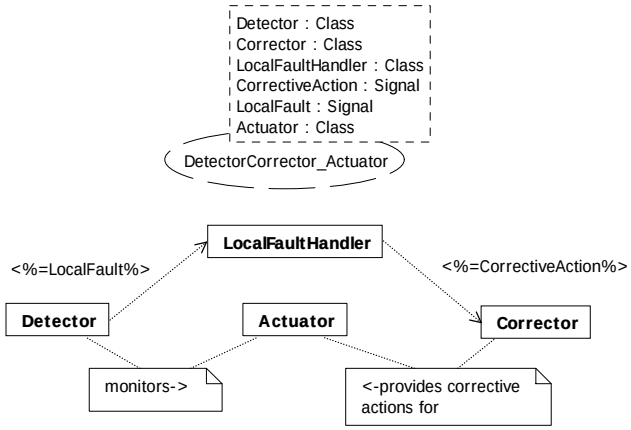
- Fehlerbehandlungsfunktionen:
  - Verwendung redundanter Werte
  - Fehlerbeseitigung durch Reset
  - Zustandswechsel (z. B. Notlauf)
  - Fehlerspeicherung

Die unterschiedlichen Verfahren für Fehlererkennung und -behandlung können zu unterschiedlichen Überwachungskonzepten kombiniert werden. Die Strukturen der entsprechenden Muster unterscheiden sich leicht. Das hier beschriebene Muster `DetectorCorrector_Actuator` wurde als ein repräsentatives Beispiel für eine Überwachungsfunktion aus dieser Gruppe gewählt. Es überprüft physikalische Eigenschaften eines Aktuators mit Hilfe eines zusätzlichen Sensors. Das Überwachungssystem wird durch drei Klassen beschrieben:

- `Detector`: Die Fehlererkennung.
- `LocalFaultHandler`: Implementiert die lokale Sicherheitslogik und koordiniert gegebenenfalls die lokale mit einer globalen Fehlerbehandlung (siehe Muster *FaultHandler*).
- `Corrector`: Setzt die korrigierenden Maßnahmen um.

Diese Trennung muss sich nicht in der Implementierung widerspiegeln. Hier kommt es vor, dass alle drei Klassen durch eine Funktion implementiert werden.

5.3 Struktur und Informationsfluss



- **Detector**: Sendet nach der Entdeckung eines Fehlers oder einer Störung `LocalFault` an `LocalFaultHandler`.

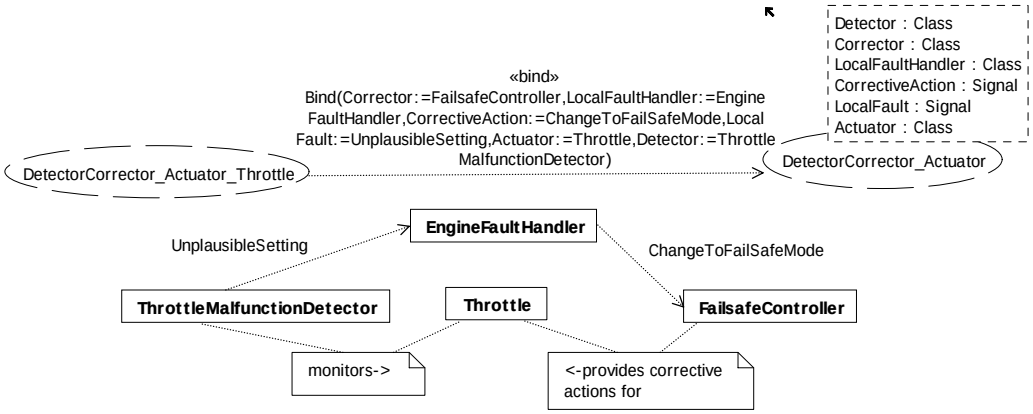
- **LocalFaultHandler**: Wendet nach Erhalt von `FaultIndication` eine lokale Sicherheitslogik an und versendet gegebenenfalls `CorrectiveAction` an Corrector.
- **Corrector**: Führt nach Erhalt von `IndicateCorrectiveAction` korrigierende Maßnahmen durch.
- **Actuator**: der Aktuator, der überwacht wird.

5.4 Quelle / Verwandte Muster

Das Muster stammt von [Konrad et al. '04b], ist in der hier gezeigten Fassung aber für die Überwachung eines Aktuators spezialisiert und hinsichtlich der Signale und der Musterparameter präzisiert worden.

5.5 Beispiel

Das Beispiel zeigt die Überwachung einer Drosselklappe (`Throttle`). Wird durch die Erkennungsfunktion `ThrottleMalfunctionDetector` eine unplausible Stellung (`UnplausibleSetting`) erkannt, aktiviert die Funktion `EngineFaultHandler` einen Notlauf (`FailSafeController`).



5.6 Anwendbare Designmuster

- `MonitorActuator`: Überwachungskonzept des Aktuators in einem Objektmodell auf Designebene abbilden.
- `ExceptionMonitor`: Fehler auf der Ebene einer Objektkollaboration erkennen und behandeln.
- `Singleton`: Sicherstellen, dass es nur eine Instanz der globalen Fehlerüberwachung und des globalen Logging-Mechanismus gibt.

6 Detector-Corrector (Originalauszug aus [Konrad et al. '04a])

Im Folgenden ist die Originalbeschreibung des Musters *Detector-Corrector* aus [Konrad et al. '04a] wiedergeben. Damit soll ein Beispiel für die in Kapitel 3.2.4.c erwähnten *Object Analysis Patterns* geliefert werden.

Detector-Corrector

Intention:

Monitor a device or system conditions and initiate corrective action(s) if a violation is found.

Motivation:

Embedded systems typically have tight timing and operational constraints. Providing a mechanism to assure that a component is operational or that specific constraints on the system are not violated is the objective of the Detector-Corrector Pattern. The Detector-Corrector Object Analysis Pattern leverages the concept of detectors and correctors from the fault tolerance community [Arora et al. '98]. The objective of the Detector-Corrector Pattern is achieved by a local fault handler that interacts with detectors (components that detect whether some state predicate is satisfied [Arora et al. '98]) and correctors (components that correct the system state in order to satisfy a violated predicate [Arora et al. '98]). Examples of detectors include comparators, watchdogs, and exception conditions. Examples of correctors include recovery procedures (such as reset procedures and backup devices) or exception handlers. In embedded systems, detectors commonly capture the following types of inputs [Douglass '99]:

- Timeout messages from watchdogs.
- Assertions of software errors.
- Results of built-in-tests (BITs) that run on a periodic or continuous basis.

Applicability:

The *Detector-Corrector* Pattern is applicable

- To detect and correct the violation of specific system constraints.

Structure:

The class diagram for the *Detector-Corrector* Pattern can be seen in Figure 7.1. A LocalFaultHandler interacts with its `Detectors` and `Correctors` which provide fault detection and correction capabilities for a Device. The `LocalFaultHandler` also reports occurring faults to the `GlobalFaultHandler`, which can override the `LocalFaultHandler`.

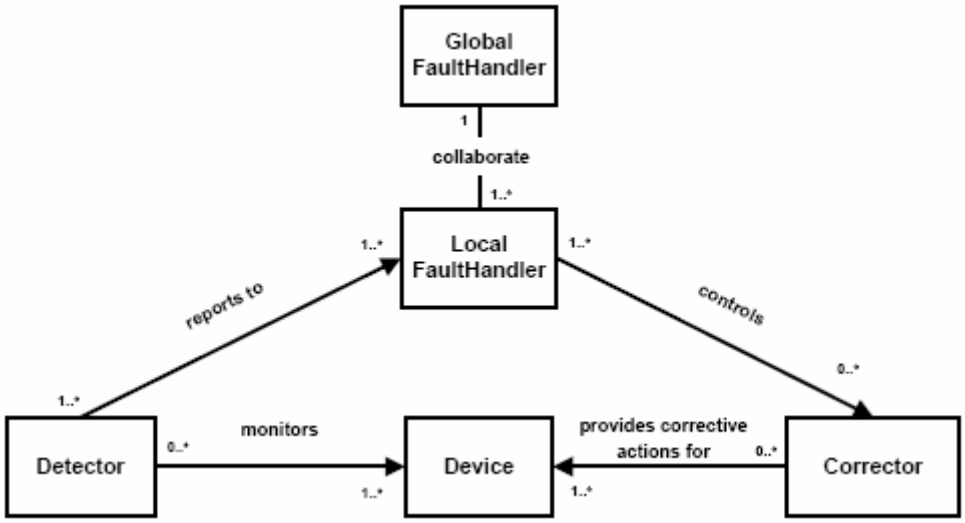


Figure 7.1: UML class diagram of the *Detector-Corrector* Pattern

Behavior:

Figure 7.2 shows the behavioral diagram for the `LocalFaultHandler`. The `LocalFaultHandler` receives fault reports from `Detectors` and initiates recovery actions in the `Correctors`. Additionally, the `LocalFaultHandler` can be overridden by the `GlobalFaultHandler` and be returned to normal behavior on a `Reset()`.

Figure 7.3 and Figure 7.4 show state diagrams for the `Detector` class. Figure 7.3 represents a state diagram for a detector that is waiting for a periodic service (denoted by an `Update()` message) by a `Device`. Figure 7.4 represents a state diagram for a detector that periodically checks if certain system conditions are violated. Additionally, Figure 7.5 shows a state diagram for the `Corrector` component in which the `Corrector` initiates corresponding recovery actions.

Figure 7.6 shows a sequence diagram where a detector of the first type (Figure 7.3) detects a timeout of the `Device` and reports the error to the `LocalFaultHandler`. The `LocalFaultHandler`, in turn, reports the fault to the `GlobalFaultHandler` and activates the `Corrector`, which initiates a reset of the `Device`.

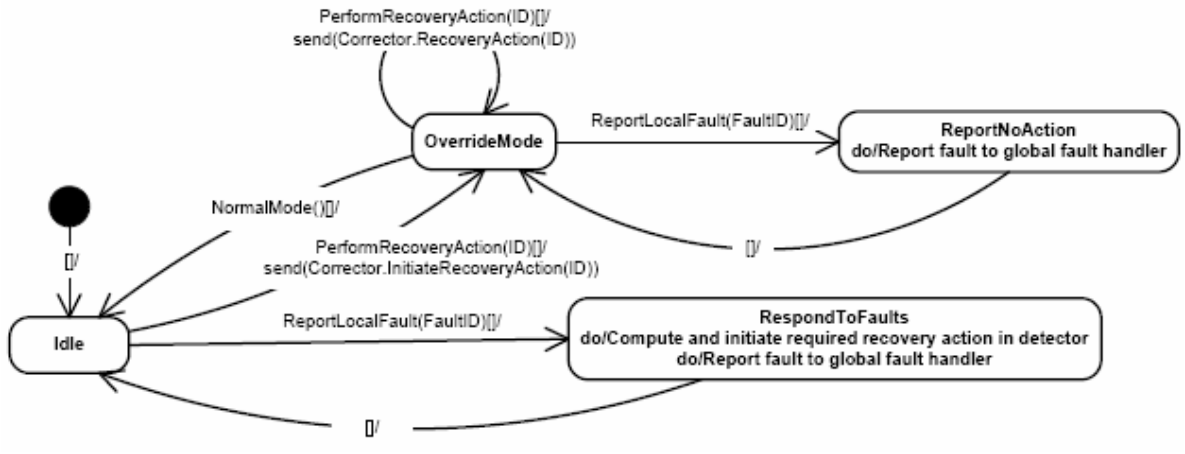


Figure 7.2: UML state diagram of the `LocalFaultHandler` in the *Detector-Corrector* Pattern

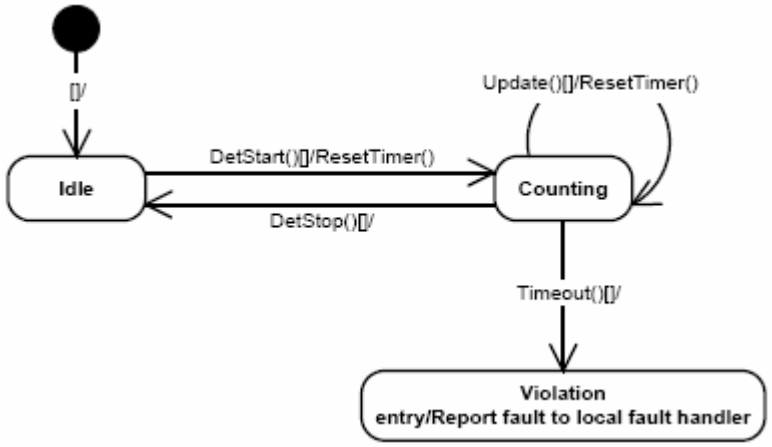


Figure 7.3: UML state diagram of a timeout detector in the *Detector-Corrector* Pattern

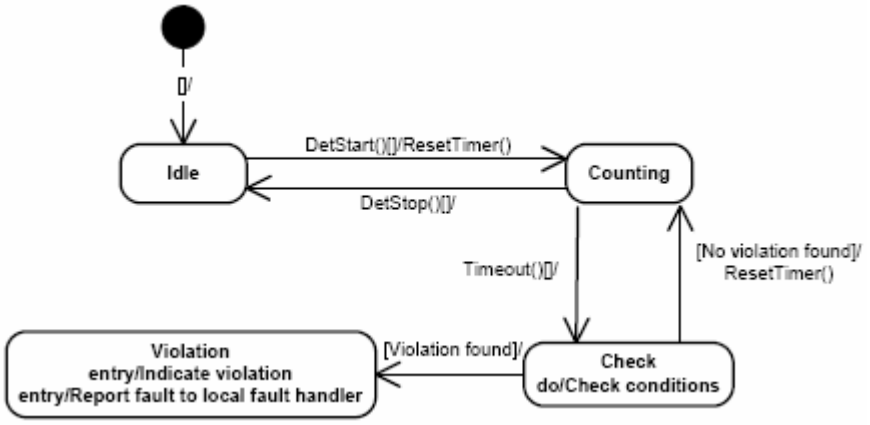


Figure 7.4: UML state diagram of a constraint violation detector in the *Detector-Corrector* Pattern

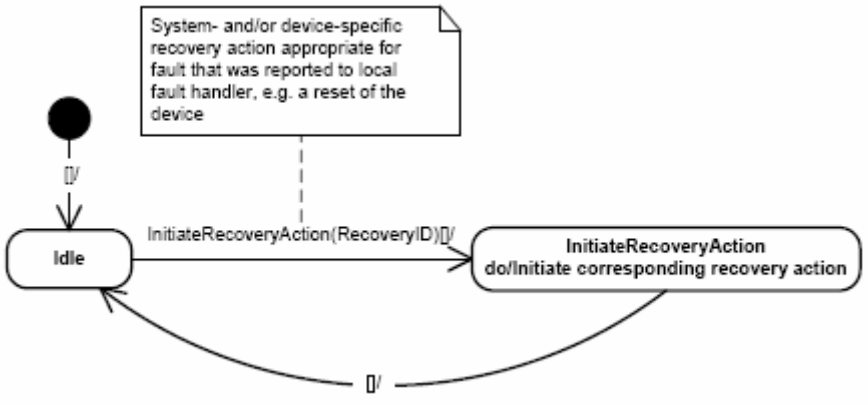


Figure 7.5: UML state diagram for the Corrector in the Detector-Corrector Pattern

Participants:

- Watchdog: Watchdog monitoring the device. The Watchdog can be implemented in hardware to protect it from software faults.
- Device: Device monitored by the Watchdog.
- FaultHandler: Central fault handler of the system.

Collaborations:

- The Detector waits for a message from a Device or monitors conditions of a device (such as the value of a sensor) on a periodic basis. If this message does not arrive on time or a condition is violated, then the watchdog performs recovery actions, such as resetting a device or shutting the system down and reports the fault to the FaultHandler.
- The FaultHandler handles the fault message and may initiate additional actions.

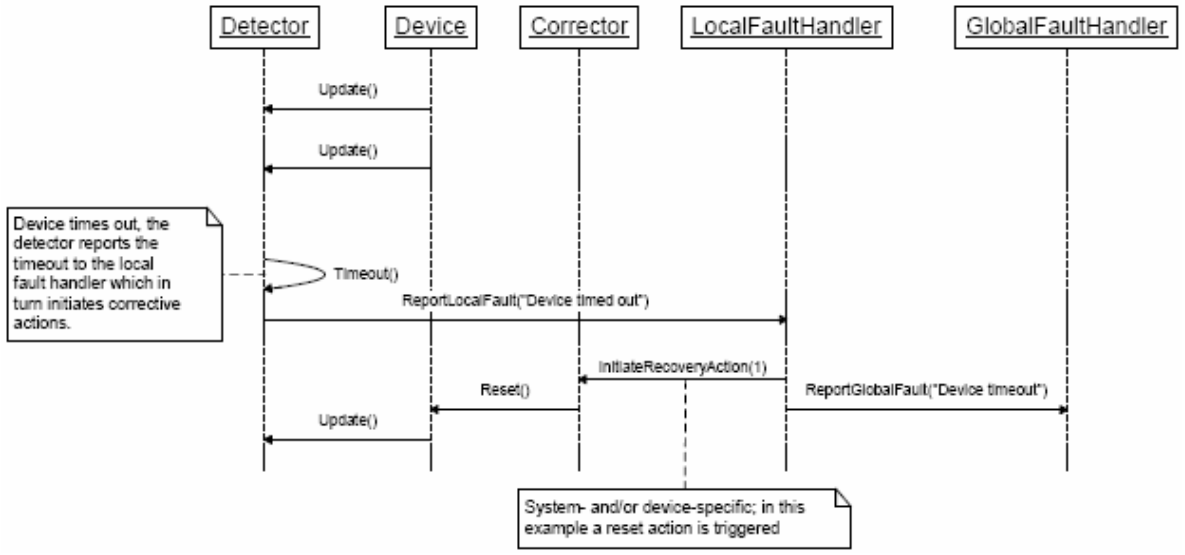


Figure 7.6: UML sequence diagram example of a timeout detector in the *Detector-Corrector* Pattern

Consequences:

- If a detector monitors a device on a periodic basis, then the device must send life ticks periodically to the detector.
- A local fault handler must be present to handle fault messages from a detector and to activate correctors.
- The system must contain a reset operation or a more elaborate fault recovery mechanism that the corrector can perform in case of a violation.

Constraints:

The following constraints must be satisfied after the Detector-Corrector pattern has been applied (*Anmerkung: Ausdrücke in der formalen Constraint-Sprache wurden weggelassen*):

- If there is a violation, then the detector should eventually detect the violation.
- When the detector detects a violation, the corresponding local fault handler must be notified.
- If a fault is reported to the local fault handler, then the local fault handler must activate the corresponding corrector appropriate to the fault reported.
- If a corrector is activated, then the local fault handler will eventually perform the corresponding recovery action appropriate to the system being modeled (e.g., activating a corrector that resets the device).
- When a fault is reported to the local fault handler, the global fault handler is eventually notified.

Applicable Design Patterns:

- Watchdog Design Pattern [Douglass '99]: Describes more implementation specific details about the watchdog, such as implementation strategies.

Also Known As:

To be determined.

Known Uses:

To be determined.

Related Object Analysis Patterns:

- Fault Handler Object Analysis Pattern: Stores and handles fault messages from the watchdog.

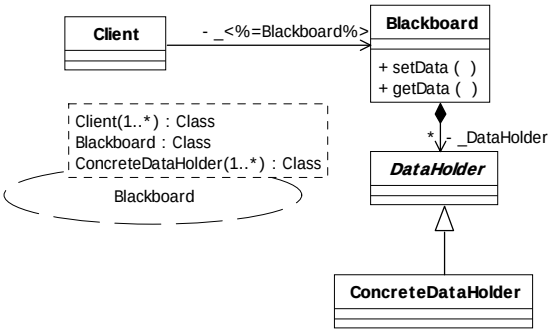
A.2 Designmuster

1 Blackboard

1.1 Zweck

Verteilt erzeugte Daten zentral in einem Objekt kapseln und über eine generische Schnittstelle lesen und schreiben.

1.2 Struktur



1.3 Erläuterung

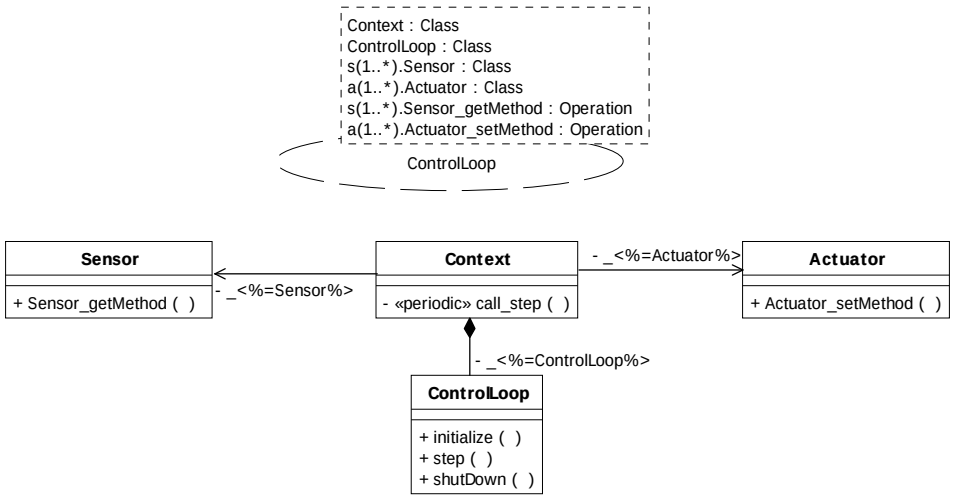
Das Blackboard-Designmuster in der gezeigten Form stammt von [Yacoub et al. '04] und stellt eine stark abgewandelte Version des ursprünglichen Musters von Buschmann [Buschmann '98] dar. Es dient in dieser Fassung als abstrakter Datenspeichermechanismus. Die Blackboard-Klasse selbst verwaltet eine Reihe von Datenklassen (DataHolder), die jeweils als konkrete Datenklasse (ConcreteDataHolder) spezialisiert werden müssen. Über den Client wird der Benutzer des Datenspeichers festgelegt, der über die Operationen setData und getData der Klasse Blackboard Daten setzen und auslesen kann. In der Formulierung ist absichtlich offen gehalten, wie die Blackboard-Klasse die Identifikation der konkreten Datenklassen vornimmt. Möglich ist dies beispielsweise über die Vergabe von Daten-IDs oder die Auswertung von Typinformationen. Das Muster kann überall dort eingesetzt werden, wo ein lokaler Datenspeicher zur Ablage und zum Austausch von Daten sinnvoll ist.

2 ControlLoop

2.1 Zweck

Den Algorithmus einer Regelung-/Steuerung in einem Objekt kapseln.

2.2 Struktur



2.3 Erläuterung

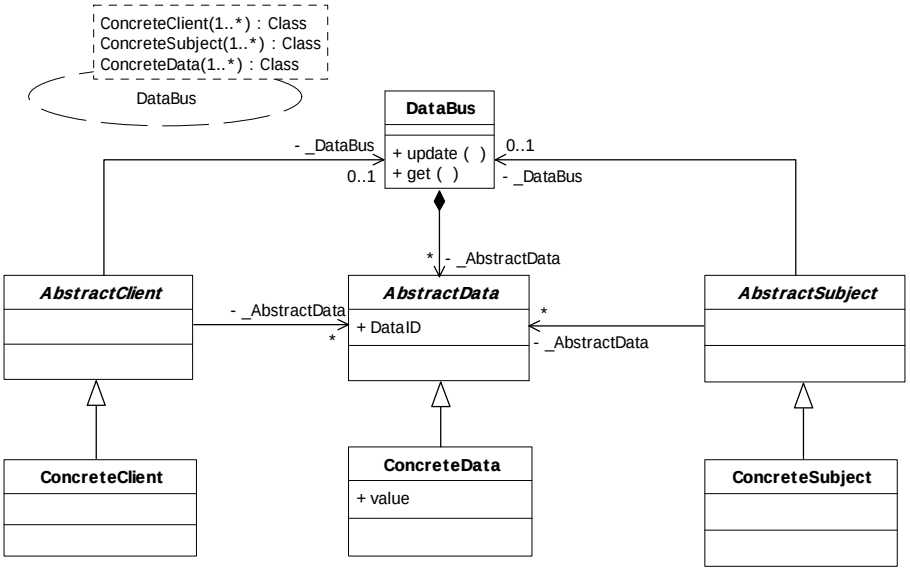
Das ControlLoop-Muster von [Douglass '99] stellt eine einfache Realisierung einer Regelung oder Steuerung auf Designebene dar. Der Regelungsalgorithmus wird in die Klasse ControlLoop ausgelagert und auf die drei Operationen initialize, step und shutDown aufgeteilt, die von einer übergeordneten Klasse (Context) aufgerufen werden. In den drei Operationen (vor allem der Operation step) wird das eigentliche Regelungs- bzw. Steuerungsverhalten implementiert. Dazu werden die Daten der Sensoren (Sensor) gelesen und nach Ausführung der benötigten Berechnungen in Aktuatorwerte (Actuator) umgesetzt.

3 DataBus

3.1 Zweck

Den Zugriff auf einen Datenbus in einer einheitlichen Objektstruktur kapseln.

3.2 Struktur



3.3 Erläuterung

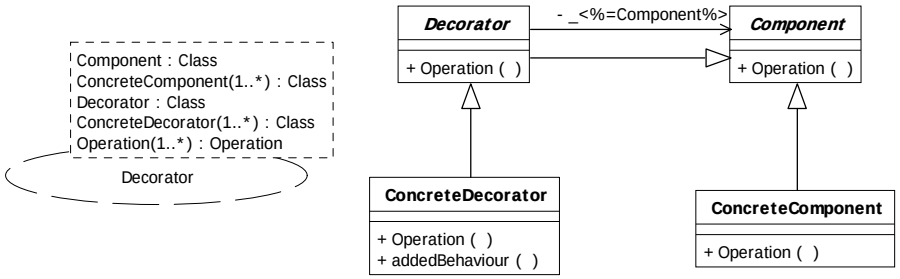
Das ebenfalls von Douglass [Douglass '02] übernommene DataBus-Muster stellt einen Datenspeicher dar, der im Gegensatz zum Blackboard-Muster nicht nur lokal zur Verfügung steht, sondern über einen Bus auch Daten mit anderen Komponenten oder Subsystemen austauschen kann. Dies wird dadurch verdeutlicht, dass die wesentliche Busstruktur in der Musterdefinition nicht parametrisiert ist und somit für alle Anwendungen des Musters in einem Designmodell gleich ist. Diese Struktur beinhaltet eine zentrale Datenbusklasse (DataBus), die eine Menge von Daten (AbstractData) verwaltet und auf die lesende (AbstractClient) und schreibende (AbstractSubject) Elemente zugreifen können. Ändert sich ein Datum, so wird der Datenbus von den Subjects über die Operation update sofort über diese Änderung informiert. Somit ist gewährleistet, dass die Clients über die Operation get immer den aktuellen Wert der jeweiligen konkreten Datenklasse erhalten. Die Aggregationsbeziehungen von AbstractClient und AbstractSubject zu AbstractData bedeuten nicht, dass diese auch direkt auf alle Daten zugreifen dürfen, sondern dienen nur der Festlegung derjenigen Daten, die diese über den Datenbus auslesen bzw. für die sie neue Werte bereitstellen. Für eine Instanz des Musters können jeweils beliebig viele konkrete lesende oder schreibende Elemente (ConcreteClient bzw. ConcreteSubject) und konkrete Datenklassen (ConcreteData) dem allgemeinen Busmechanismus untergeordnet werden.

4 Decorator

4.1 Zweck

Ein Objekt um Zuständigkeiten erweitern.

4.2 Struktur



4.3 Erläuterung

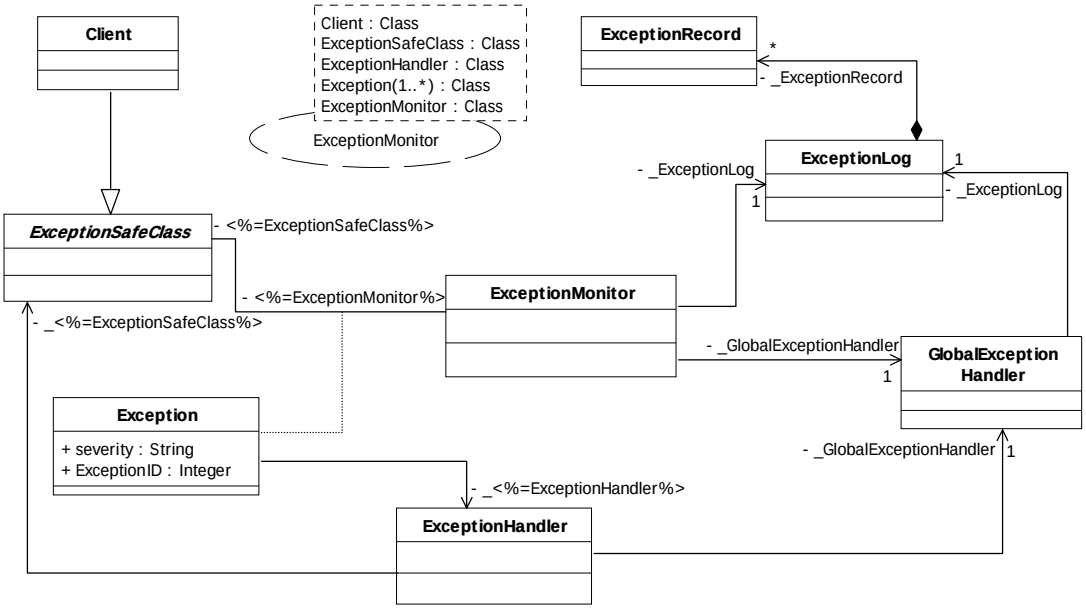
Das Decorator-Muster von Gamma et al. [Gamma et al. '95] erweitert ein Objekt dynamisch um zusätzliche Funktionalität. Dazu wird zunächst eine abstrakte Dekoriererklasse (Decorator) definiert, die eine Referenz auf eine abstrakte Komponente (Component) hält, diese aber zugleich beerbt und somit eine identische Schnittstelle (Operation) liefert. Ein konkreter Dekorierer (ConcreteDecorator) kann somit einer Konkretisierung der Komponente (ConcreteComponent) zusätzliche Funktionalität, in diesem Fall in Form einer weiteren Operation (addedBehavior), hinzufügen. Das Muster kann zum Beispiel eingesetzt werden, wenn in Objekten verwaltete Werte auf verschiedene Arten aufbereitet werden sollen.

5 ExceptionMonitor

5.1 Zweck

Fehler auf der Ebene einer Objektkollaboration erkennen und behandeln.

5.2 Struktur



5.3 Erläuterung

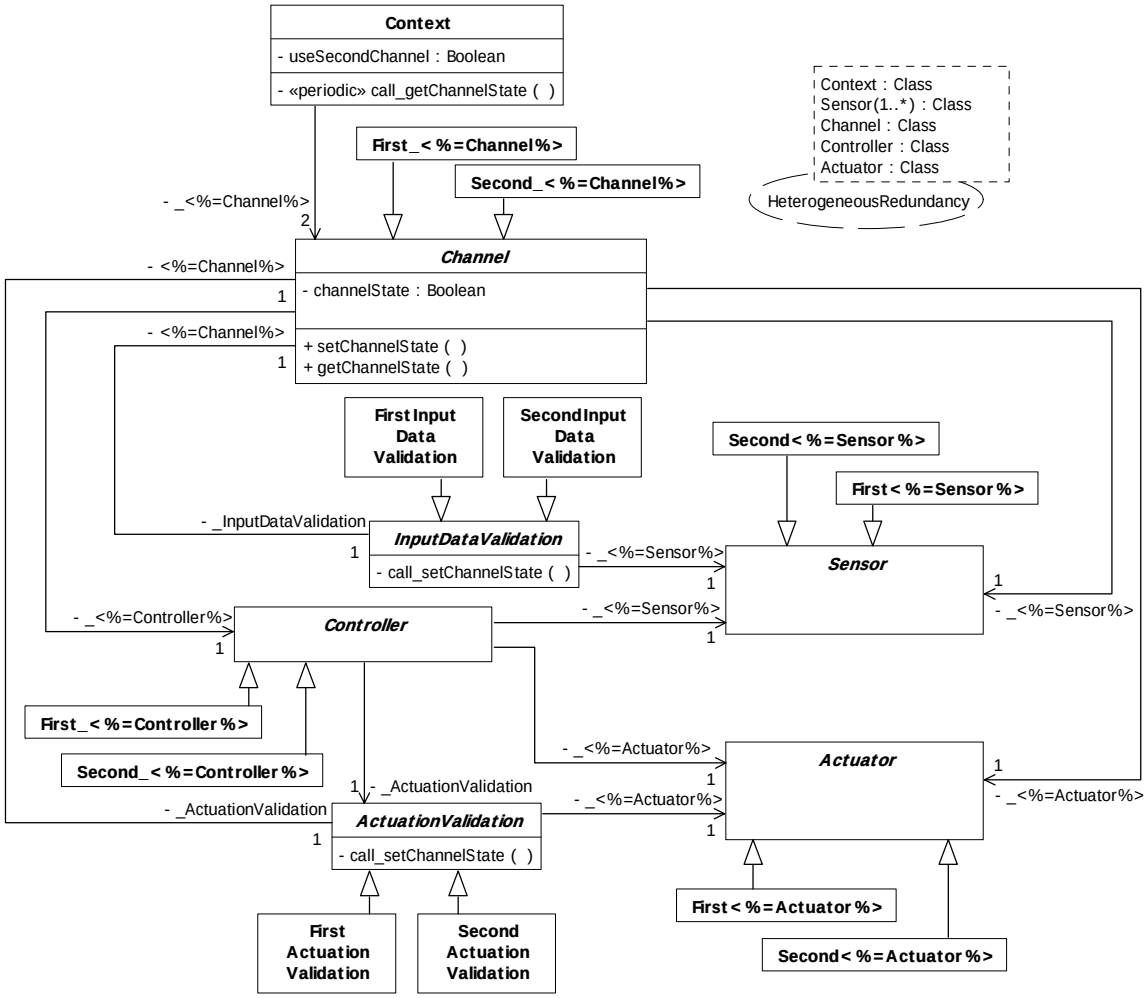
Das ExceptionMonitor-Muster von [Douglass '99] stellt einen Mechanismus für die Behandlung von Ausnahmen auf Designebene dar. Die ExceptionSafeClass stellt eine abstrakte Schnittstelle zur Benutzung dieses Mechanismus bereit, die ein Client durch Beerbung verwenden kann. Sie sendet einzelne Exceptions an den ExceptionMonitor, der eine Aufzeichnung über die Ausnahmen führt (ExceptionLog und ExceptionRecord) und sie gegebenenfalls an die globale Ausnahmebehandlung (GlobalExceptionHandler) weiterleitet. Jede Exception verweist auf den für sie zuständigen ExceptionHandler, der wiederum der ExceptionSafeClass bzw. dem GlobalExceptionHandler mitteilt, ob die Ausnahme behandelt werden konnte oder nicht. Die Multiplizitäten der Parameter sind so gesetzt, dass sich das Muster immer auf eine Kombination von ExceptionMonitor, ExceptionHandler und ExceptionSafeClass bezieht. Die Klassen ExceptionLog und GlobalExceptionHandler sollten laut Douglass als *Singleton* (siehe Designmuster Singleton) implementiert werden.

6 HeterogeneousRedundancy

6.1 Zweck

Heterogene Redundanz in einem Objektmodell abbilden.

6.2 Struktur



6.3 Erläuterung

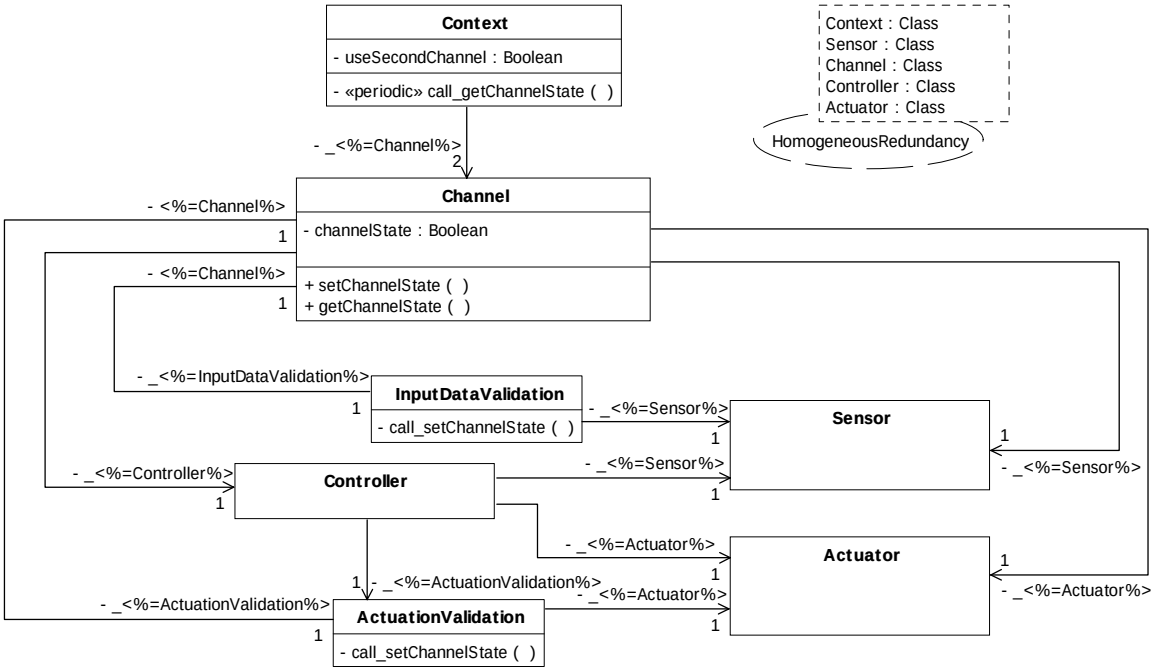
Das hier gezeigte HeterogeneousRedundancy-Muster wurde von [Douglass '02] übernommen. Es unterscheidet sich von dem Muster HomogeneousRedundancy dadurch, dass redundante Elemente unterschiedlich implementiert werden. Auf diese Weise können auch systematische Implementierungsfehler behandelt werden. Im Klassenmodell schlägt sich diese Tatsache dadurch nieder, dass es eigene Klassen für jedes redundante Element gibt.

7 HomogeneousRedundancy

7.1 Zweck

Homogene Redundanz in einem Objektmodell abbilden.

7.2 Struktur



7.3 Erläuterung

Das von [Douglass '02] übernommene HomogeneousRedundancy-Muster beschreibt ein Designmodell für ein System, in dem die Verfügbarkeit durch die redundante Auslegung der Sensorik, Aktuatorik und der Steuerung/Regelung erhöht wird. Charakteristisch ist dabei, dass es sich bei den redundanten Elementen um dieselbe Implementierung handelt. Auf diese Weise kann zwar z. B. der Ausfall einer Hardware behandelt werden (indem auf die redundanten Elemente umgeschaltet wird), systematische Implementierungsfehler werden damit jedoch nicht adressiert.

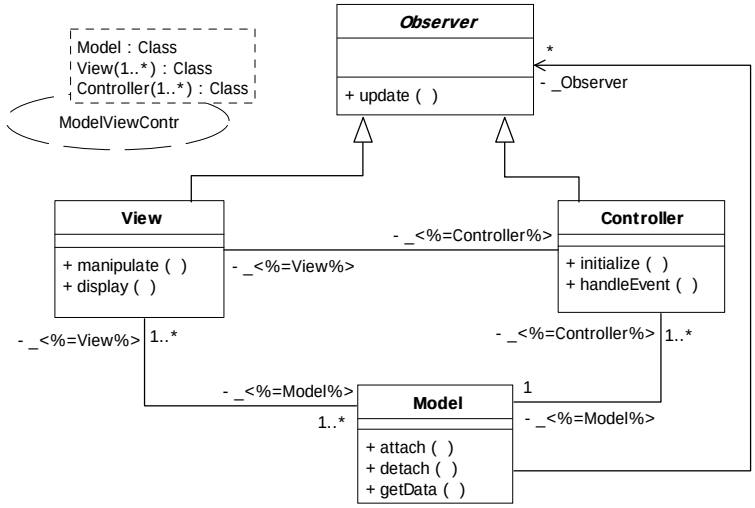
Ein Channel fasst in dem Muster die Klassen Controller, Sensor, Actuator, InputDataValidation und ActuationValidtion zusammen. Dieses Gespann wird zweimal instanziiert, so dass die Klasse Context Zugriff auf zwei Channel hat. Grundsätzlich wird der zuerst instanziierte Channel verwendet. Erkennen die Klassen InputDataValidation oder ActuationValidation einen Fehler, wird das Attribut channelState auf false gesetzt. Die Klasse Context fragt dieses Attribut periodisch ab und setzt im Falle von false das Attribut useSecondChannel auf true, womit auf den zweiten Channel umgeschaltet wird.

8 Model-View-Controller (ModelViewContr)

8.1 Zweck

Modell, Darstellung und Steuerung einer interaktiven Anwendung in separaten Objekten kapseln.

8.2 Struktur



8.3 Erläuterung

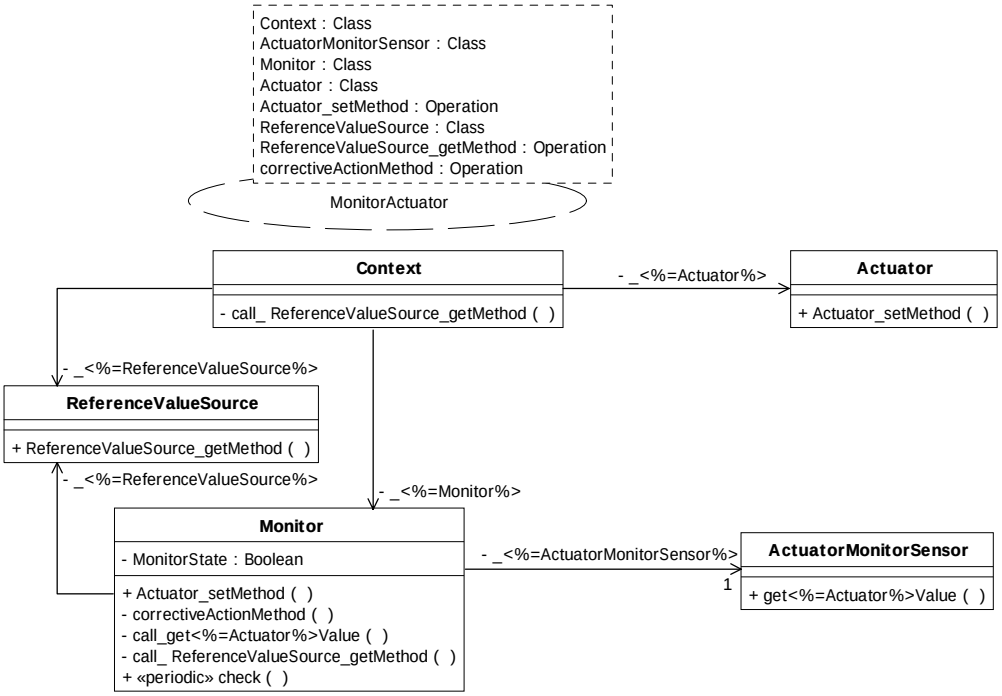
Das Muster Model-View-Controller (in der Literatur auch oft als „MVC“ abgekürzt) wurde von [Buschmann '98] übernommen und in ein Klassendiagramm übertragen. Es beschreibt die Unterteilung einer interaktiven Anwendung in drei Komponenten: Ein Modell (Model), das die Kernfunktionalität und die Daten enthält, Ansichten (Views), die dem Benutzer Informationen präsentieren und eine Steuerungskomponente (Controller), die die Interaktion steuert. Ein Benachrichtigungsmechanismus (Observer) sichert die Konsistenz zwischen den drei Komponenten. Ansichten und Steuerung werden beim Model an- (attach) bzw. abgemeldet (detach). Das Modell ruft bei allen angemeldeten Ansichten und Steuerungskomponenten update auf, wenn sich Daten ändern. Ansichten initialisieren bei Bedarf die Steuerung (initialize), fragen Daten beim Modell ab (getData) und implementieren die abstrakte update-Operation der Observer-Klasse bezüglich der konkreten Darstellung aus. Controller übersetzen Bedieneingaben in Aufrufe von Operationen an das Modell oder Darstellungsanforderungen an die Ansicht (handleEvent). Falls notwendig, können sie ebenfalls die abstrakte update-Operation von Observer implementieren.

9 MonitorActuator

9.1 Zweck

Ein Überwachungskonzept eines Aktuators in einem Objektmodell abbilden.

9.2 Struktur



9.3 Erläuterung

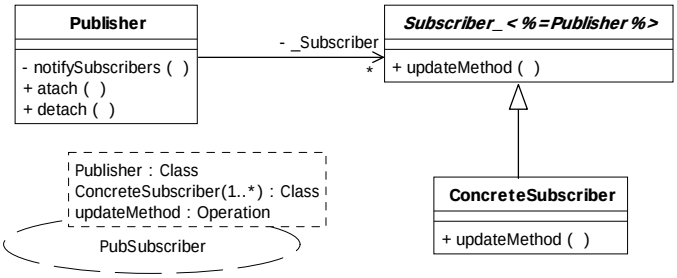
Das von [Douglass '02] adaptierte `MonitorActuator`-Muster beschreibt ein Objektmodell für die Überwachung eines Aktuators (`Actuator`) auf Designebene. Dies erfolgt über einen speziellen Sensor (`ActuatorMonitorSensor`), der durch ein Objekt vom Typ `Monitor` ausgelesen wird. Der gemessene Wert wird im Rahmen der periodischen Methode `check` mit dem Sollwert (`ReferenceValueSource_getMethod`) und der Stellgröße des Aktuators (`Actuator_setMethod`) verglichen. Wird hier ein kritischer Zusammenhang festgestellt, werden über die Methode `correctiveActionMethod` korrigierende Maßnahmen eingeleitet.

10 Publisher-Subscriber (PubSubscriber)

10.1 Zweck

Zustandsänderungen eines Objekts unidirektional an registrierte Abonnenten schicken.

10.2 Struktur



10.3 Erläuterung

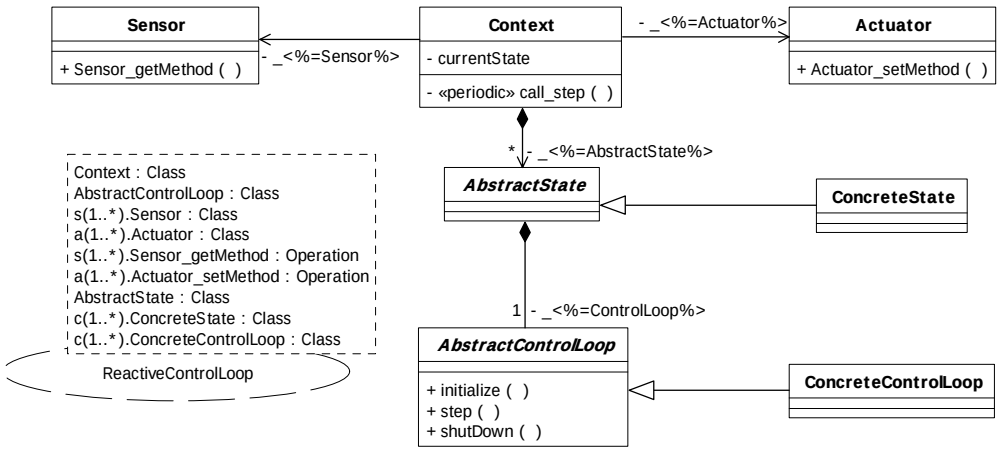
Das Publisher-Subscriber-Muster stammt ursprünglich von Buschmann [Buschmann '98], wurde in der hier gezeigten Form jedoch für die Darstellung als UML-Klassendiagramm angepasst. Es stellt einen Benachrichtigungsmechanismus zwischen einem Herausgeber (Publisher) und mehreren Abonnenten (ConcreteSubscriber), die über eine abstrakte Schnittstelle (Subscriber) definiert sind, dar. Ändern sich Daten bei dem Herausgeber, so schickt dieser den neuen Datenstand an alle registrierten Abonnenten (Operationen notifySubscribers und updateMethod). Das Muster kann überall dort eingesetzt werden, wo Signale nur in eine Richtung, aber an mehrere Empfänger kommuniziert werden. Im Gegensatz zum Muster Oberserver von [Gamma et al. '95] handelt es sich um reine Push-Kommunikation.

11 ReactiveControlLoop

11.1 Zweck

Algorithmen für unterschiedliche Zustände einer Regelung-/Steuerung in separaten Objekten kapseln.

11.2 Struktur



11.3 Erläuterung

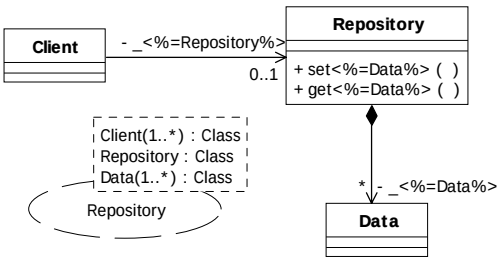
Dieses Muster von [Douglass '99] ist eine Erweiterung des Musters ControlLoop. Es unterstützt zustandsabhängiges Steuerungs- bzw. Regelungsverhalten, in dem es das State-Muster integriert. Dies wird im Muster durch den Zusammenhang zwischen jeweils einem Zustand (AbstractState) und einem Regelkreis (AbstractControl-Loop) realisiert. Diese können durch konkrete Vertreter (ConcreteState und ConcreteControlLoop) ausgestaltet werden, wobei darauf zu achten ist, dass jedem Zustand genau ein Regelkreis zugeordnet ist. In den Kontext eingebunden wird das Muster über die Klasse Context, die die vorhandenen Zustände aggregiert. Laut Douglass eignet sich das Muster vor allem, wenn die Eingabedaten für den Regelkreis stückweise kontinuierlich sind und verschiedene Berechnungsgleichungen für einzelne Intervalle verwendet werden müssen.

12 Repository

12.1 Zweck

Verteilt erzeugte Daten zentral in einem Objekt kapseln.

12.2 Struktur



12.3 Erläuterung

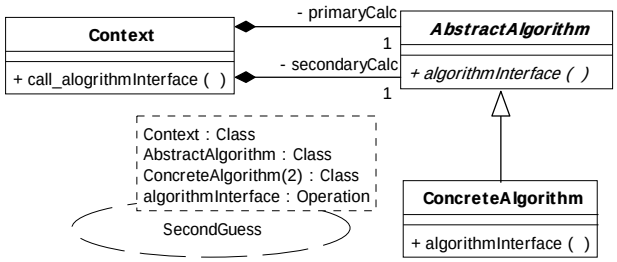
Das Muster beschreibt einen zentralen Datenspeicher für Daten, die an unterschiedlichen Stellen erzeugt werden. Durch die zentrale Datenhaltung können schreibende und lesende Klassen voneinander entkoppelt werden. Die Idee des Musters ist z. B. bei [Lalanda '98] beschrieben.

13 SecondGuess

13.1 Zweck

Eine Berechnung durch unterschiedliche Algorithmen absichern und diese in separaten Objekten kapseln.

13.2 Struktur



13.3 Erläuterung

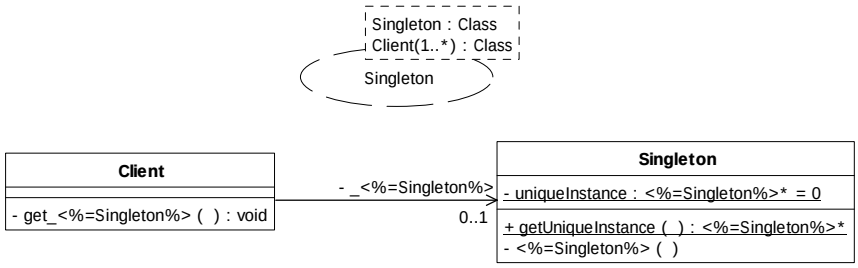
Das bei [Douglass '99] beschriebene Muster SecondGuess beschreibt, wie kritische Berechnungen auf Softwareebene abgesichert werden können. Die Klasse Context hat zwei Assoziationen zu der Klasse AbstractAlgorithm, die die Signatur für einen Algorithmus vorgibt (algorithmInterface). Die erbende Klasse ConcreteAlgorithm enthält entsprechend eine implementierende Operation. Ein Objekt vom Typ Context verfügt somit über zwei Objekte, die die Operation bereitstellen und führt die Berechnung mit beiden aus. Üblicherweise verfügt das erste Objekt über einen genauen und laufzeitintensiven Algorithmus und das zweite über einen weniger laufzeitintensiven Algorithmus, dessen Ergebnis auch ungenauer sein kann. Das Ergebnis der ersten Implementierung wird mit dem Ergebnis der zweiten Implementierung verifiziert in dem geprüft wird, ob ein signifikanter Unterschied besteht. Die Struktur des Musters hat Ähnlichkeit mit der Struktur des Musters Strategy.

14 Singleton

14.1 Zweck

Sicherstellen, dass es von einer Klasse nur eine Instanz gibt und diese global zur Verfügung stellen.

14.2 Struktur



14.3 Erläuterung

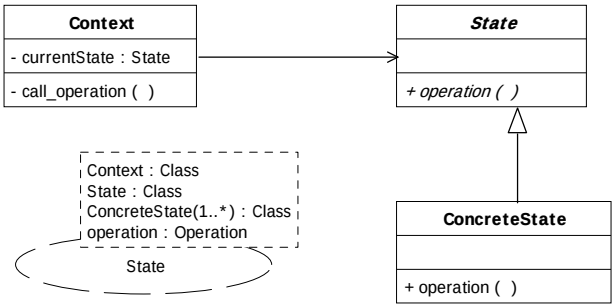
Durch die Anwendung des Singleton-Musters von [Gamma et al. '95] kann sichergestellt werden, dass es von einer Klasse nur eine Instanz gibt. Dies wird dadurch erreicht, dass die Klasse Singleton nur über eine statische Klassenoperation (getUniqueInstance) instanziiert werden kann, die die einzige Instanz der Klasse zurückliefert. Beim ersten Aufruf ruft diese Methode den privaten Konstruktor der Klasse auf und belegt das ebenfalls private Klassenattribut uniqueinstance mit der erzeugten Instanz. Bei weiteren Aufrufen liefert die Operation getUniqueInstance nur noch die gespeicherte Instanz zurück. Das Muster wird vor allem für globale Behandlungsmechanismen (siehe z. B. ExceptionMonitor) oder Diensten verwendet.

15 State

15.1 Zweck

Verhalten für unterschiedliche Zustände in eigenen Objekten kapseln.

15.2 Struktur



15.3 Erläuterung

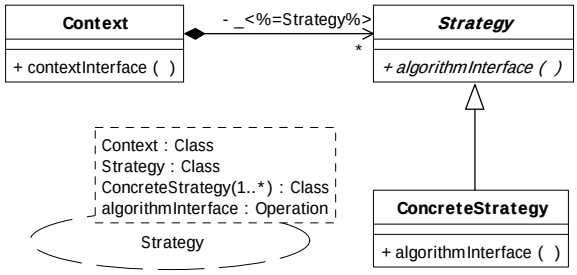
Das State-Muster von [Gamma et al. '95] ermöglicht die Realisierung einer oder mehrerer Operationen in Abhängigkeit verschiedener Zustände eines Objekts. Eine Kontextklasse (Context) assoziiert eine Menge von Zuständen (State), wobei sie sich den aktuellen Zustand merkt (Attribut currentState). Die abstrakte Zustandsklasse kann durch beliebig viele konkrete Zustände (ConcreteState) spezialisiert werden. Bei Aufruf der call_operation der Kontextklasse wird dann jeweils die Implementierung aus der aktuellen Zustandsklasse verwendet. Das Muster bietet sich an, wenn das Verhalten eines Objekts stark von seinem Zustand abhängt.

16 Strategy

16.1 Zweck

Eine Familie von austauschbaren Algorithmen definieren und in separaten Objekten kapseln.

16.2 Struktur



16.3 Erläuterung

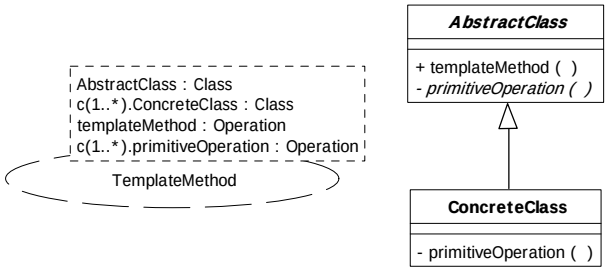
Das Strategy-Muster von Gamma et al. [Gamma et al. '95] definiert eine Familie von austauschbaren Algorithmen, die unabhängig von dem benutzenden Kontext variiert werden können. Dies wird durch eine abstrakte Strategieklasse (`Strategy`) realisiert, die ihren Unterklassen (`ConcreteStrategy`) die Implementierung einer Operation (`algorithmInterface`) vorgibt und somit eine einheitliche Schnittstelle deklariert. Die Strategien werden von der Kontextklasse (`Context`) gebündelt. Deren Operation `contextInterface` ruft die Strategien dann über die Schnittstelle auf. Die genaue Auswahl der konkreten Strategie bleibt in der Formulierung von Gamma offen, es heißt nur, dass die Kontextklasse mit einem `ConcreteStrategy`-Objekt konfiguriert werden kann. Das Muster sollte eingesetzt werden, wenn es für komplexe Algorithmen mehrere Alternativen gibt.

17 TemplateMethod

17.1 Zweck

Das Skelett eines Algorithmus definieren und einzelne Rechenschritte in separate Objekte auslagern.

17.2 Struktur



17.3 Erläuterung

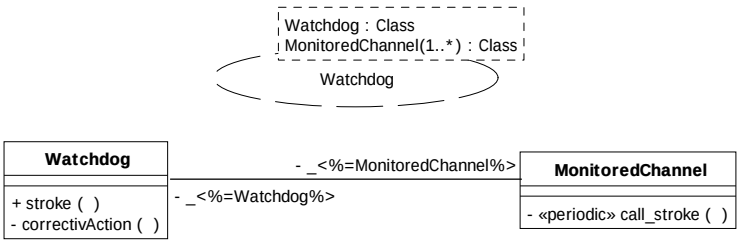
Das TemplateMethod-Muster von [Gamma et al. '95] ermöglicht die Definition eines Skelettes für einen Algorithmus (TemplateMethod), indem einzelne Schritte auf separate Operationen (PrimitiveOperation) aufgeteilt werden. Die Schablonenmethode wird nur in einer abstrakten Klasse (AbstractClass) definiert und überlässt die Implementierung der primitiven Operationen deren Unterklassen (ConcreteClass). Damit können die Schritte auch einzeln ausgetauscht werden. Der Einsatz dieses Musters bietet sich an, wenn die grobe Struktur eines komplexen Algorithmus konstant ist, seine konkrete Implementierung jedoch in verschiedenen Teilen variiert.

18 Watchdog

18.1 Zweck

Eine Zeitüberwachung in einem Objekt kapseln.

18.2 Struktur



18.3 Erläuterung

Das Watchdog-Muster, dessen Idee von Douglass [Douglass '02] übernommen wurde, beinhaltet einen einfachen Überwachungsmechanismus für Geräte: Ein „Wachhund“ (Watchdog) ist mit einem Gerät (abstrahiert als MonitoredChannel) verbunden und erwartet, dass dieses in definierten Zeitabständen die Operation notify aufruft, um ein Lebenszeichen von sich zu geben. Bleibt dieses aus, so sendet der Watchdog auf Hardwareebene ein Signal zum Zurücksetzen des Geräts (dies ist im Klassendiagramm, das sich nur auf die Software bezieht, nicht darstellbar). Das Muster kann in sicherheitskritischen Szenarien eingesetzt werden, wenn es nur darum geht, die prinzipielle Arbeitsfähigkeit einzelner Geräte zu überprüfen und diese gegebenenfalls neu zu starten.

Anhang – B: Transformationsregeln

Im Folgenden werden Transformationsregeln für Analysemuster in der konkreten Syntax aus Kapitel 4.3.3 vorgestellt. Der Zweck der in den Regeln instanziierten Designmuster wird jeweils in dem Kapitel „Anwendbare Designmuster“ der Analysemusterbeschreibung in Anhang A.1 erläutert.

B.1 ClosedLoopControl

1 Transformationsregel

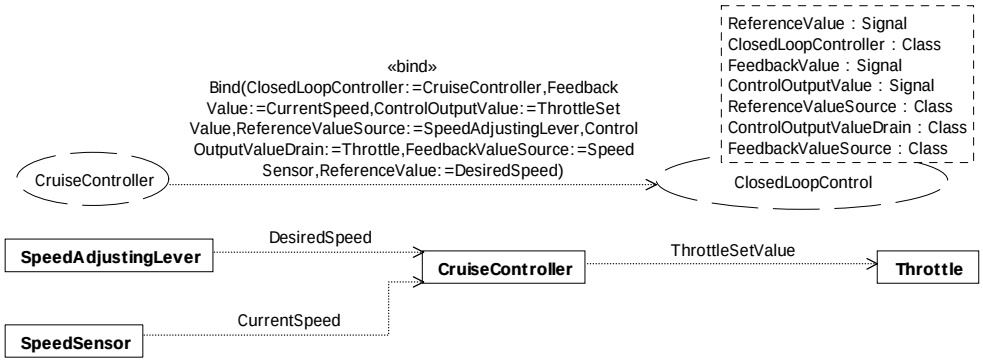
```
Rule ClosedLoop():
Template ClosedLoopControl cl ->
// Sensorwerte in einem Repository kapseln
[OptimCriteria Reuse]
  ?Template Repository(
    cl.ClosedLoopController*"_Master",      // Client
    "InputRepository",                      // Repository
    cl.FeedbackValue+cl.ReferenceValue      // Data
  ) inputRep,
// Sensorwerte von einem Datenbus lesen
[UserQuestion("Werden die Sensordaten über einen Datenbus zur Verfügung gestellt
(DE2Sig)?",false,Bus)]
  ?Template DataBus(
    [Reuse] ?inputRep.Repository+inputRep.Repository                          // ConcreteClient
          !cl.FeedbackValueSource+cl.ReferenceValueSource,
    cl.FeedbackValueSource*"_remote" + cl.ReferenceValueSource*"_remote",    // ConcreteSubject
   "Bus_"*cl.ReferenceValue + "Bus_"*cl.FeedbackValue                        // ConcreteData
   ) bus,
// Sensorwerte für mögliche neue Konsumenten zugreifbar machen
[OptimCriteria Reuse AND NOT Bus]
  ?Template PubSubscriber(
    cl.ReferenceValueSource,  // Publisher
    inputRep.Repository,      // ConcreteSubscriber
    "set"*cl.ReferenceValue   // updateMethode
  ) refValPubSub,
[OptimCriteria Reuse AND NOT Bus]
  ?Template PubSubscriber(
    cl.FeedbackValueSource,   // Publisher
    inputRep.Repository,      // ConcreteSubscriber
    "set"*cl.FeedbackValue    // updateMethode
  ) feedValPubSub,
// Unterschiedliche Klassen für Zustände
[UserQuestion("Muss der Regelalgorithmus mehrere Zustände unterscheiden?", false, Reactive)]
  ?Template ReactiveControlLoop(
    cl.ClosedLoopController*"_Master",                  // Context
    cl.ClosedLoopController*"_AbstractControlLoop",     // AbstractControlLoop
    [Reuse] ?inputRep.Repository+inputRep.Repository    // Sensor
          !cl.FeedbackValueSource+cl.ReferenceValueSource,
```

```
    cl.ControlOutputValueDrain,                          // Actuator
    "get"*cl.FeedbackValue+"get"*cl.ReferenceValue,      // Sensor_getMethod
    "set"*cl.ControlOutputValue,                         // Actuator_getMethod
    cl.ClosedLoopController*"_AbstractState",            // AbstractState
    getNames("Namen der konkreten Zustände",-1),         // ConcreteState
    getNames("Namen der konkreten Regelkreise",-1)       // ConcreteControlLoop
  ) reacloop
  // Wenn keine zustandbehaftetes Verhalten…
  !Template ControlLoop(
    cl.ClosedLoopController*"_Master",                   // Context
    cl.ClosedLoopController*"_ControlLoop",              // ControlLoop
    [Reuse] ?inputRep.Repository+inputRep.Repository     // Sensor
          !cl.FeedbackValueSource+cl.ReferenceValueSource,
    cl.ControlOutputValueDrain,                          // Actuator
    "get"*cl.FeedbackValue+"get"*cl.ReferenceValue,      // Sensor_getMethod
    "set"*cl.ControlOutputValue                          // Actuator_getMethod
  ) cloop,
// Den Regelungsalgorithmus austauschbar machen
[OptimCriteria Reuse AND UserQuestion("Müssen alternative Regelstrategien verwendet
werden?",false, Strat)]
  ?Template Strategy(
    [Reactive] ?reacloop.AbstractControlLoop !cloop.ControlLoop,      // Context
    cl.ClosedLoopController*"_Strategy",                              // Strategy
    getNames("Namen der alternative Regelalgorithmen",-1),            // Concrete Strategy
    getNames("Interface-Name des Regelalgorithmus",1)                 // algorithmInterface
  ) controlStrat,
// Die Struktur des Regelungsalgorithmus vorgeben
[OptimCriteria Reuse AND Strat]
  ?Template TemplateMethod(
    controlStrat.Strategy,                        // AbstractClass
    controlStrat.ConcreteStrategy,                // ConcreteClass
    controlStrat.algorithmInterface,              // templateMethod
    "calcControlDeviation"+"calcControlValue"     // primitiveOperation
    ) template,
[OptimCriteria Reuse AND NOT Strat]
  ?Template TemplateMethod(
    [Reactive] ?reacloop.AbstractControlLoop !cloop.ControlLoop,      // AbstractClass
    cl.ClosedLoopController*"_ConrolAlgoImpl",                        // ConcreteClass
    "closedLoopAlgorithm",                                            // templateMethod
    "calcControlDeviation"+"calcControlValue"                         // primitiveOperation
    ) template,
// Heterogene Redundanz unterstützen
[OptimCriteria Safety AND UserQuestion("Muss heterogene Redundanz unterstützt werden?",false, Rd)]
  ?Template HeterogeneousRedundancy(
    "RedundancyMaster",                           // Context
    [Reactive] ?reacloop.Sensor !cloop.Sensor,    // Sensor
    cl.ClosedLoopController*"_Channel",           // Channel
    [Reactive] ?reacloop.Context !cloop.Context,  // Controller
    [Reactive] ?reacloop.Actuator !cloop.Actuator // Actuator
  ) heteroRed,
```

```
// Typische Sicherheitsmechanismen über Hilfsregel einbringen
[OptimCriteria Safety]
  ?makeStandardSafety([Reactive] ?reacloop.AbstractControlLoop !cloop.ControlLoop).
```

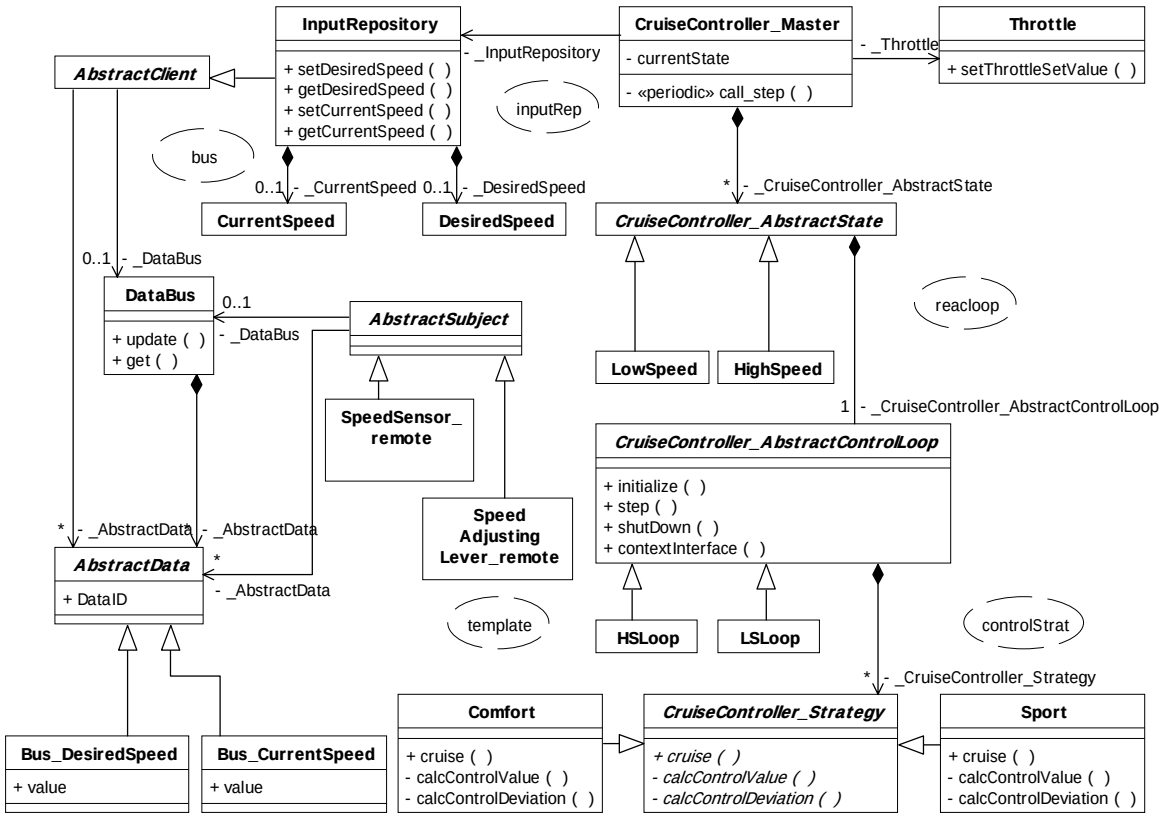
2 Beispiel

2.1 Analysemodell



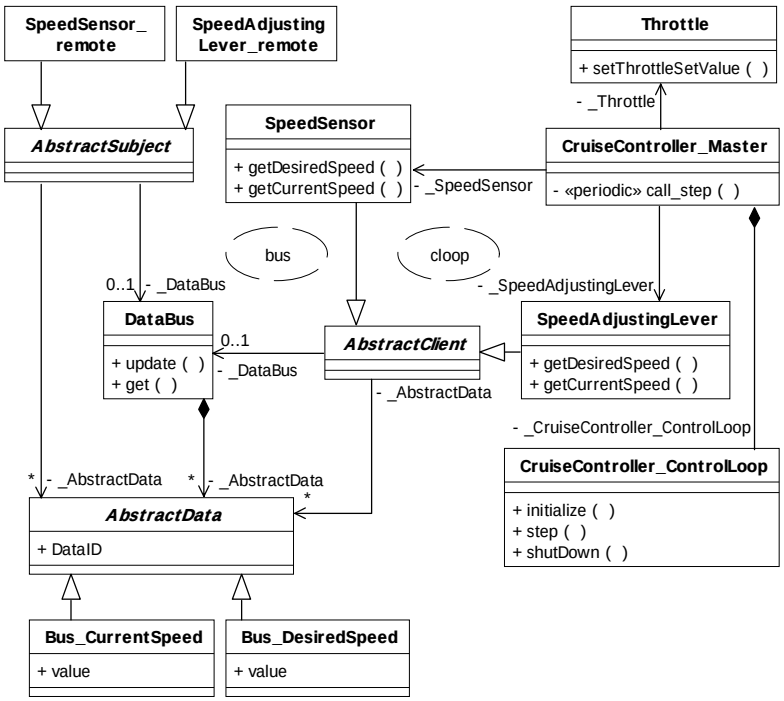
2.2 Designmodell 1

Frage	**Antwort**
Optimierungskriterium:	*Wiederverwendung (Reuse)*
Werden die Sensordaten über einen Datenbus zur Verfügung gestellt (DE2Sig)?	*Ja*
Muss der Regelalgorithmus mehrere Zustände unterscheiden?	*Ja*
Namen der konkreten Zustände:	*LowSpeed, HighSpeed*
Namen der konkreten Regelkreise:	*LSLoop, HSLoop*
Müssen alternative Regelstrategien verwendet werden?	*Ja*
Namen der alternativen Regelalgorithmen:	*Comfort, Sport*
Interface-Name des Regelalgorithmus:	*cruise*



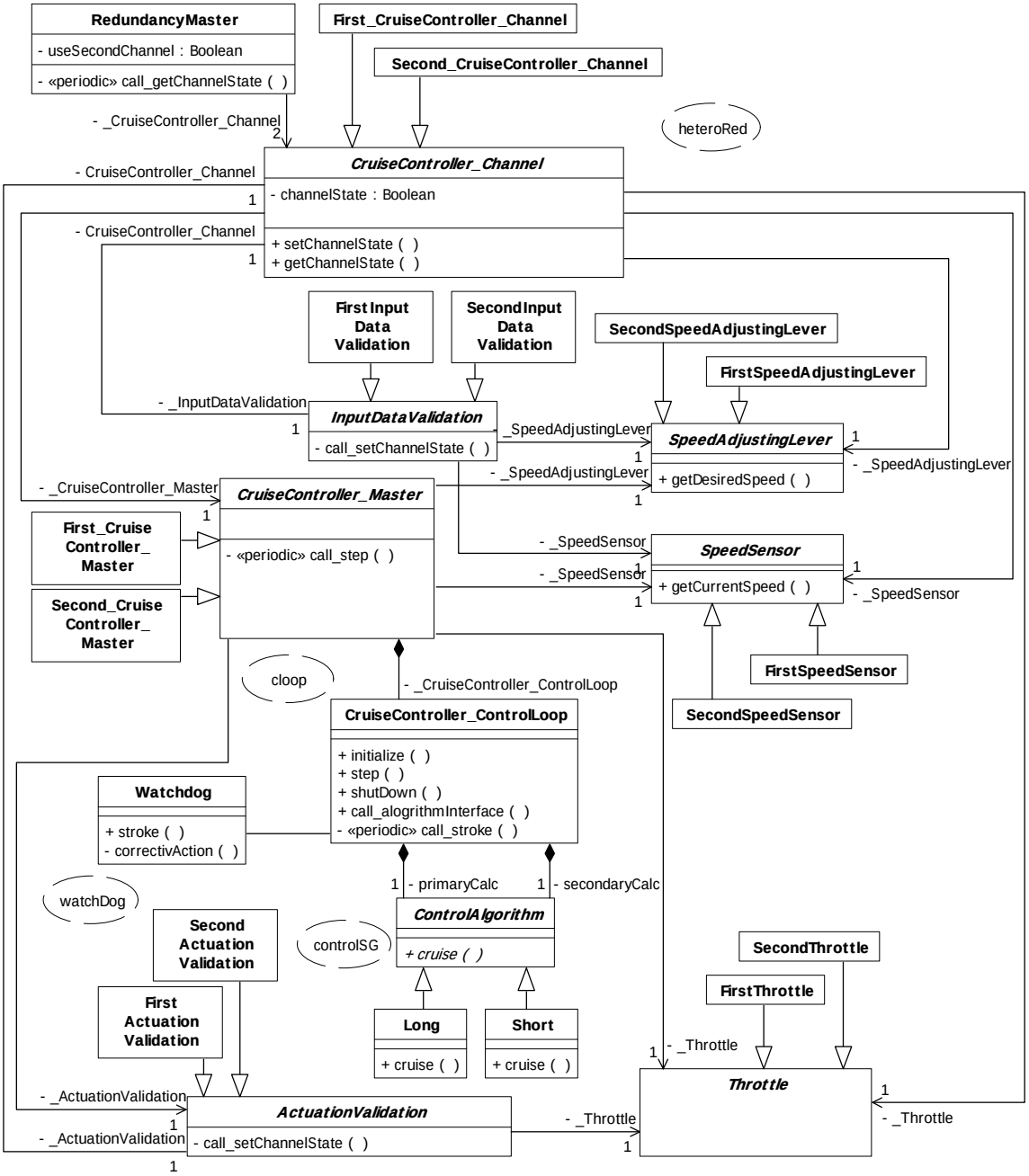
2.3 Designmodell 2

Frage	**Antwort**
Optimierungskriterium:	*Performance*
Werden die Sensordaten über einen Datenbus zur Verfügung gestellt (DE2Sig)?	*Ja*
Muss der Regelalgorithmus mehrere Zustände unterscheiden?	*Nein*



2.4 Designmodell 3

Frage	**Antwort**
Optimierungskriterium:	*Sicherheit und Verfügbarkeit (Safety)*
Werden die Sensordaten über einen Datenbus zur Verfügung gestellt (DE2Sig)?	*Nein*
Muss der Regelalgorithmus mehrere Zustände unterscheiden?	*Nein*
Muss heterogene Redundanz unterstützt werden?	*Ja*
Interface-Name des redundanten Regelalgorithmus:	*cruise*



B.2 OpenLoopControl

1 Transformationsregel

```
Rule OpenLoop():
Template OpenLoopControl ol ->
// Sensorwerte in einem Repository kapseln
[OptimCriteria Reuse]
  ?Template Blackboard(
    ol.OpenLoopController*"_Master",        // Client
    "InputRepository",                      // Blackboard
```

```
    ol.ControlInputValue+ol.ReferenceValue  // ConcreteDataHolder
  ) inputRep,
// Sensorwerte von einem Datenbus lesen
[UserQuestion("Werden die Sensordaten über einen Datenbus zur Verfügung gestellt
(DE2Sig)?",false,Bus)]
  ?Template DataBus(
    [Reuse] ?inputRep.Repository+inputRep.Repository                        // ConcreteClient
          !ol.ControlInputValueSource+ol.ReferenceValueSource,
    ol.ControlInputValueSource*"_remote"
          + ol.ReferenceValueSource*"_remote",                             //ConcreteSubject
   "Bus_"*ol.ReferenceValue + "Bus_"*ol.ControlInputValue                  // ConcreteData
   ) bus,
// Sensorwerte für mögliche neue Konsumenten zugreifbar machen
[OptimCriteria Reuse AND NOT Bus]
  ?Template PubSubscriber(
    ol.ReferenceValueSource,  // Publisher
    inputRep.Blackboard,      // ConcreteSubscriber
    "setData"                 // updateMethode
  ) refValPubSub,
[OptimCriteria Reuse AND NOT Bus]
  ?Template PubSubscriber(
    ol.ControlInputValueSource.elements,    // Publisher
    inputRep.Blackboard,                    // ConcreteSubscriber
    "setData"                               // updateMethode
  ) InputValPubSub,
// Unterschiedliche Klassen für Zustände
[UserQuestion("Muss der Regelalgorithmus mehrere Zustände unterscheiden?", false, Reactive)]
  ?Template ReactiveControlLoop(
    ol.OpenLoopController*"_Master",                    // Context
    ol.OpenLoopController*"_AbstractControlLoop",       // AbstractControlLoop
    [Reuse] ?inputRep.Blackboard+inputRep.Blackboard    // Sensor
          !ol.ControlInputValueSource+ol.ReferenceValueSource,
    ol.ControlOutputValueDrain,                         // Actuator
    "getData",                                          // Sensor_getMethod
    "setData",                                          // Actuator_getMethod
    ol.OpenLoopController*"_AbstractState",             // AbstractState
    getNames("Namen der konkreten Zustände",-1),        // ConcreteState
    getNames("Namen der konkreten Regelkreise",-1)      // ConcreteControlLoop
  ) reacloop
  // Wenn keine zustandbehaftetes Verhalten…
  !Template ControlLoop(
    ol.OpenLoopController*"_Master",                    // Context
    ol.OpenLoopController*"_ControlLoop",               // ControlLoop
    [Reuse] ?inputRep.Blackboard+inputRep.Blackboard    // Sensor
          !ol.ControlInputValueSource+ol.ReferenceValueSource,
    ol.ControlOutputValueDrain,                         // Actuator
    "getData",                                          // Sensor_getMethod
    "setData"                                           // Actuator_getMethod
  ) cloop,
// Den Steuerungsalgorithmus austauschbar machen
```

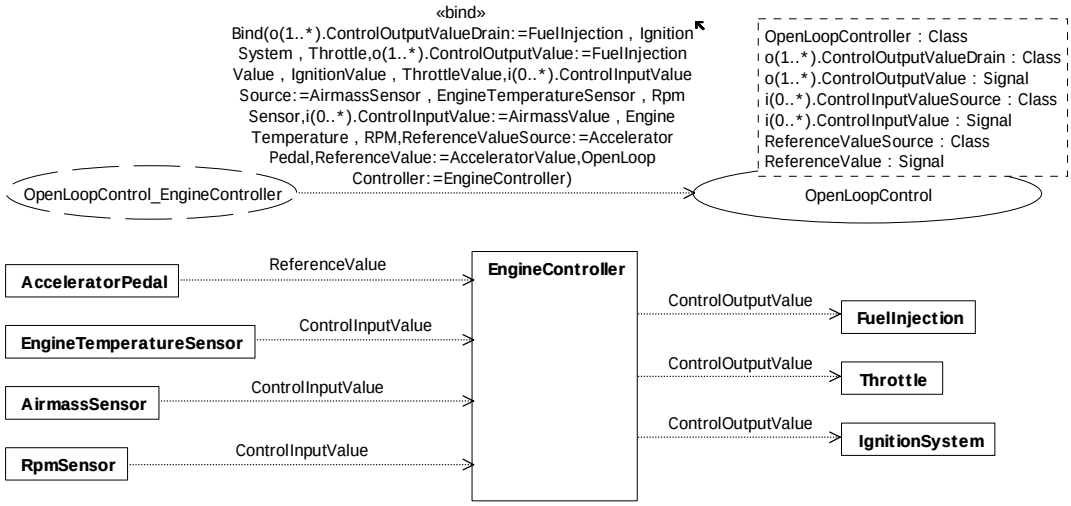
```
[OptimCriteria Reuse AND UserQuestion("Müssen alternative Steuerungsstrategien verwendet
werden?",false, Strat)]
  ?Template Strategy(
    [Reactive] ?reacloop.AbstractControlLoop !cloop.ControlLoop,      // Context
    ol.OpenLoopController*"_Strategy",                                // Strategy
    getNames("Namen der alternative Steuerungsalgorithmen",-1),       // Concrete Strategy
    getNames("Interface-Name des Steuerungsalgorithmus",1)            // algorithmInterface
  ) controlStrat,
// Heterogene Redundanz unterstützen
[OptimCriteria Safety AND UserQuestion("Muss heterogene Redundanz unterstützt werden?",false, Rd)]
  ?Template HeterogeneousRedundancy(
    "RedundancyMaster",                          // Context
    [Reactive] ?reacloop.Sensor !cloop.Sensor,   // Sensor
    ol.OpenLoopController*"_Channel",            // Channel
    [Reactive] ?reacloop.Context !cloop.Context, // Controller
    [Reactive] ?reacloop.Actuator !cloop.Actuator // Actuator
  ) heteroRed,
// Typische Sicherheitsmechanismen über Hilfsregel einbringen
[OptimCriteria Safety]
  ?makeStandardSafety([Reactive] ?reacloop.AbstractControlLoop !cloop.ControlLoop).
```

Die Regel OpenLoop() hat große Ähnlichkeiten mit der Regel ClosedLoop(). Die Unterschiede resultieren hauptsächlich aus der Tatsache, dass im Template des Analysemusters OpenLoopControl dem Parameter ControlInputValueSource beliebig viele tatsächliche Parameter zugewiesen werden können.

- Das Repository-Muster wurde durch das Blackboard-Muster ausgetauscht. Es hat einen ähnlichen Zweck wie das Muster Repository, generiert jedoch nicht für jeden Sensorwert ein Methodenpaar getX/setX. Stattdessen wird der Zugriff auf die Sensorwerte über die Methoden getData und setData generisch gelöst.
- Da ControlInputValueSource beliebig viele tatsächliche Parameter zugewiesen werden können, muss das Muster PubSubscriber auch entsprechend oft instanziiert werden. Dies geschieht durch das Anhängen von .elements bei der Parameterzuweisung von Publisher.

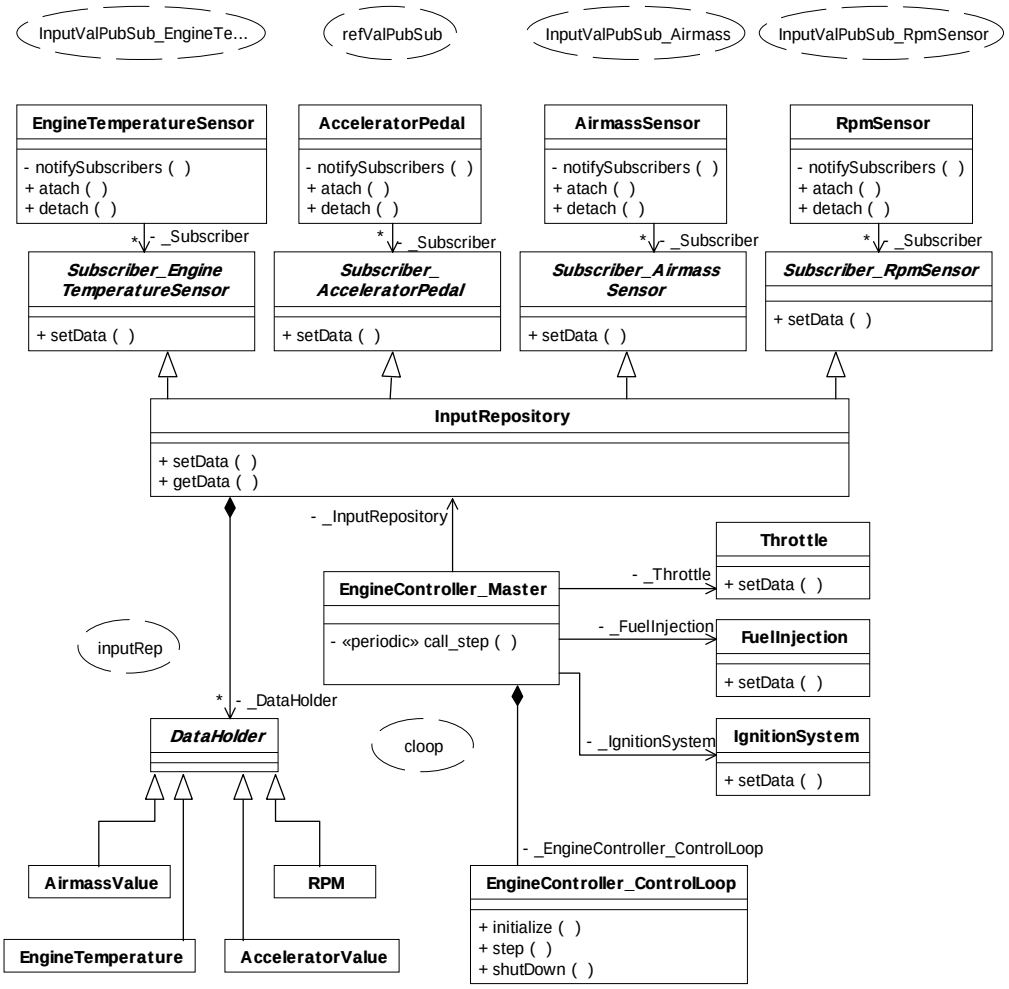
2 Beispiel

2.1 Analysemodell



2.2 Designmodell

Frage	**Antwort**
Optimierungskriterium:	*Wiederverwendung (Reuse)*
Werden die Sensordaten über einen Datenbus zur Verfügung gestellt (DE2Sig)?	*Nein*
Muss der Regelalgorithmus mehrere Zustände unterscheiden?	*Nein*
Müssen alternative Steuerungsstrategien verwendet werden?	*Nein*



B.3 User Interface

1 Transformationsregel

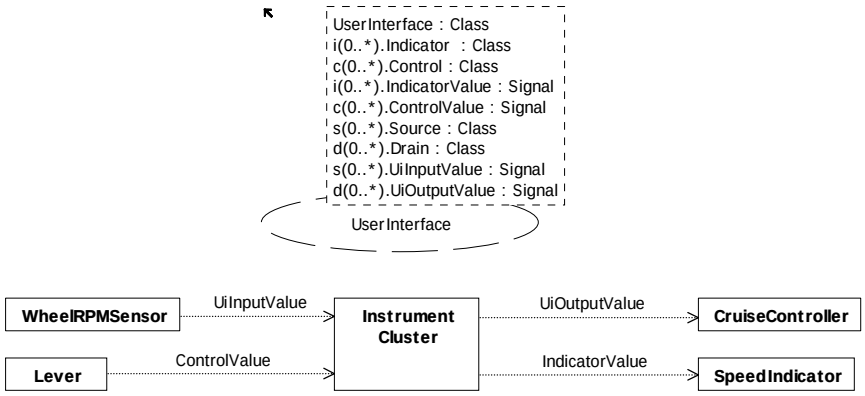
```
Rule UserInterface():
Template UserInterface ui ->

// Daten in einem Repository kapseln
[OptimCriteria Reuse]
  ?Template Blackboard(
    ui.Indicator+ui.Source+ui.Drain+ui.UserInterface*"_Controller",   // Client
    ui.UserInterface*"_Model",                                        // Blackboard
    ui.UiInputValue+ui.UiOutputValue+ui.IndicatorValue                // ConcreteDataHolder
  ) modelRep,
// Daten von einem Datenbus lesen und schreiben
[UserQuestion("Werden die Input/Output-Daten des User Interfaces über einen Datenbus zur Verfügung
gestellt (DE2Sig)?",false,Bus)]
  ?Template DataBus(
```

```
    [Reuse] ?modelRep.Blackboard                  // ConcreteClient
          !ui.Source+ui.Drain,
    ui.Source*"_remote"+ui.Drain*"_remote",       // ConcreteSubject
    "Bus_"*ui.UiInputValue+"Bus_"*ui.UiOutputValue// ConcreteData
   ) bus,
// Modell, Darstellung und Steuerung der User Interfaces in separaten Objekten kapseln
Template ModelViewContr(
  ui.UserInterface*"_Model",          // Model
  ui.Indicator+ui.Control,            // View
  ui.UserInterface*"_Controller"      // Controller
) mvc,
// Daten für die Darstellung aufbereiten
[OptimCriteria Reuse AND UserQuestion("Sollen die angezeigten Werte durch einen Dekorierer
aufbereitet werden?",false,Decor)]
  ?Template Decorator(
    "AbstractView",     // Component
    mvc.View,           // ConcreteComponent
    "ViewDecorator",    // Decorator
    getNames("Namen der konkreten View-Dekorierer",-1), // ConcreteDecorator
    "display"           // Operation
  ) viewDecorator,
// Zustandsbehaftetes Verhalten des Controllers unterstützen
makeStates(mvc.Controller).
```

2 Beispiel

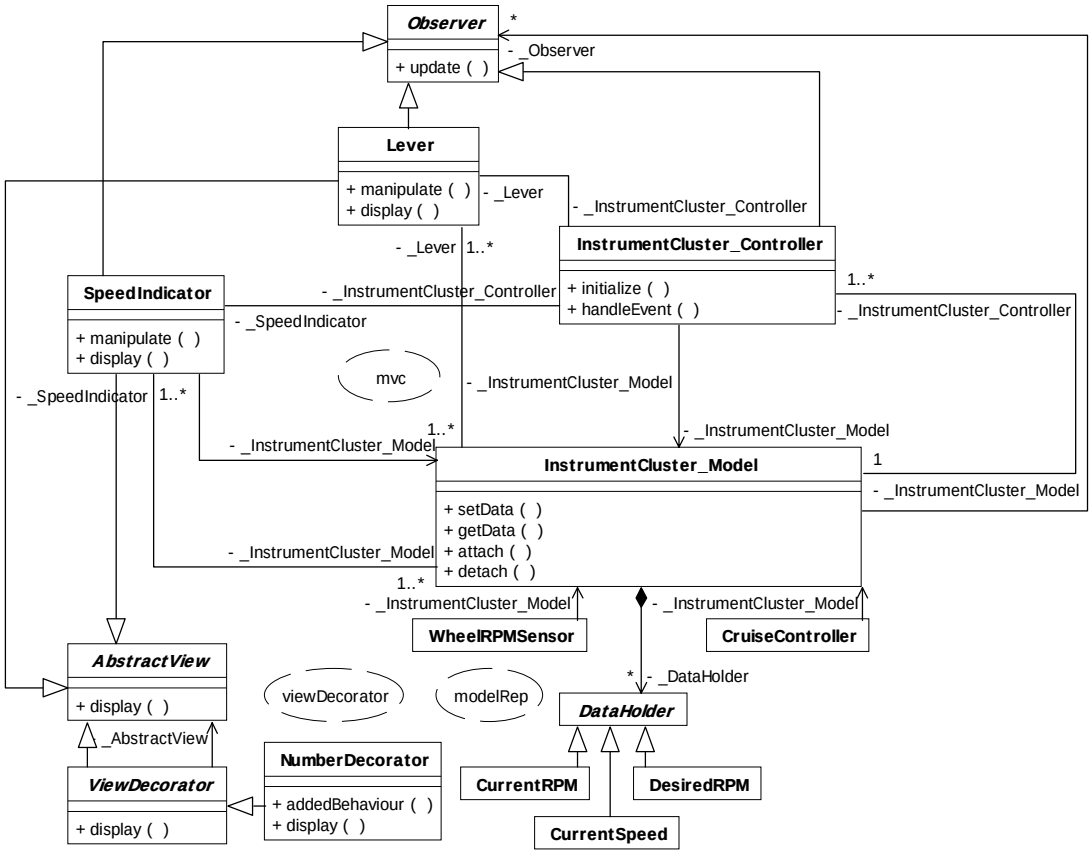
2.1 Analysemodell



2.2 Designmodell 1

Frage	Antwort
Optimierungskriterium:	*Wiederverwendung (Reuse)*
Werden die Input/Output-Daten des User Interfaces über einen Datenbus zur Verfügung gestellt (DE2Sig)?	*Nein*

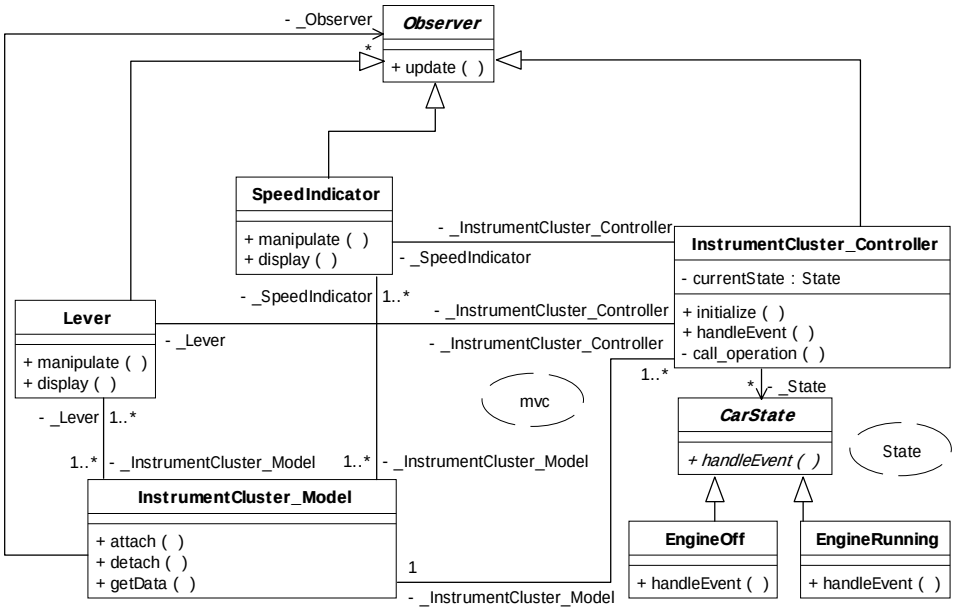
Frage	Antwort
Sollen die angezeigten Werte durch einen Dekorierer aufbereitet werden?	*Ja*
Namen der konkreten View-Dekorierer:	*NumberDecorator*
Ist die Klasse InstrumentCluster_Controller zustandsbehaftet?	*Nein*



2.3 Designmodell 2

Frage	Antwort
Optimierungskriterium:	*Wiederverwendung (Reuse)*
Werden die Input/Output-Daten des User Interfaces über einen Datenbus zur Verfügung gestellt (DE2Sig)?	*Nein*
Sollen die angezeigten Werte durch einen Dekorierer aufbereitet werden?	*Nein*
Ist die Klasse InstrumentCluster_Controller zustandsbehaftet?	*Ja*
Name der Zustands-Klasse:	*CarState*

Frage	Antwort
Namen der konkreten Zustände:	*EngineRunning, EngineOff*
Namen der Zustandsoperationen:	*handleEvent*



B.4 DetectorCorrector

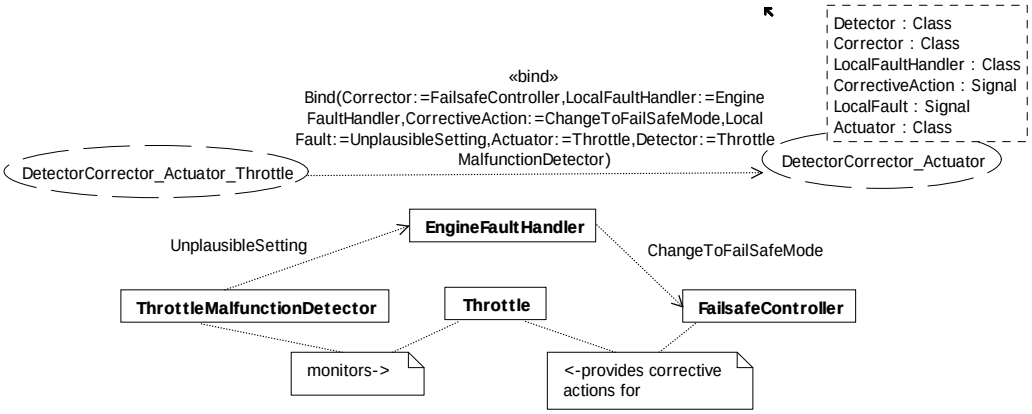
1 Transformationsregel

```
Rule DetectorCorrector_Actuator():
Template DetectorCorrector_Actuator dc ->
// Überwachungskonzept des Aktuators in einem Objektmodell abbilden
Template MonitorActuator (
  getNames("Klasse die den Kontext der Überwachung darstellt", 1),    // Context
  getNames("Name des Überwachungssensors", 1),          // ActuatorMonitorSensor
  dc.Detector,                                          // Monitor
  dc.Actuator,                                          // Actuator
  "set"*dc.Actuator*"Value",                            // Actuator_setMethode
  getNames("Klasse die den Sollwert liefert", 1),       // ReferenceValueSource
  getNames("Methode für die Abfrage des Sollwerts", 1), // ReferenceValueSource_getMethod
  dc.CorrectiveAction                                   // correctiveActionMethod
) monAct,
// Fehler auf der Ebene einer Objektkollaboration erkennen und behandeln
Template ExceptionMonitor(
  monAct.Monitor,                   // Client
  "ExceptionSafeClass",             // ExceptionSafeClass
  monAct.Monitor*"_ExceptionHandler", // ExceptionHandler
  dc.LocalFault*"_Exception",       // Exception
  dc.LocalFaultHandler              // ExceptionMonitor
```

```
) exMon,
// Sicherstellen dass es nur Instanz von GlobalExceptionMonitor gibt
Template Singleton(
  "GlobalExceptionHandler",                        // Singleton
  exMon.ExceptionMonitor+exMon.ExceptionHandler    // Client
) singGlobExHand,
// Sicherstellen dass es nur eine Instanz von ExceptionLog gibt
Template Singleton(
  "ExceptionLog",                                  // Singleton
  "GlobalExceptionHandler"+exMon.ExceptionMonitor  // Client
) singExLog.
```

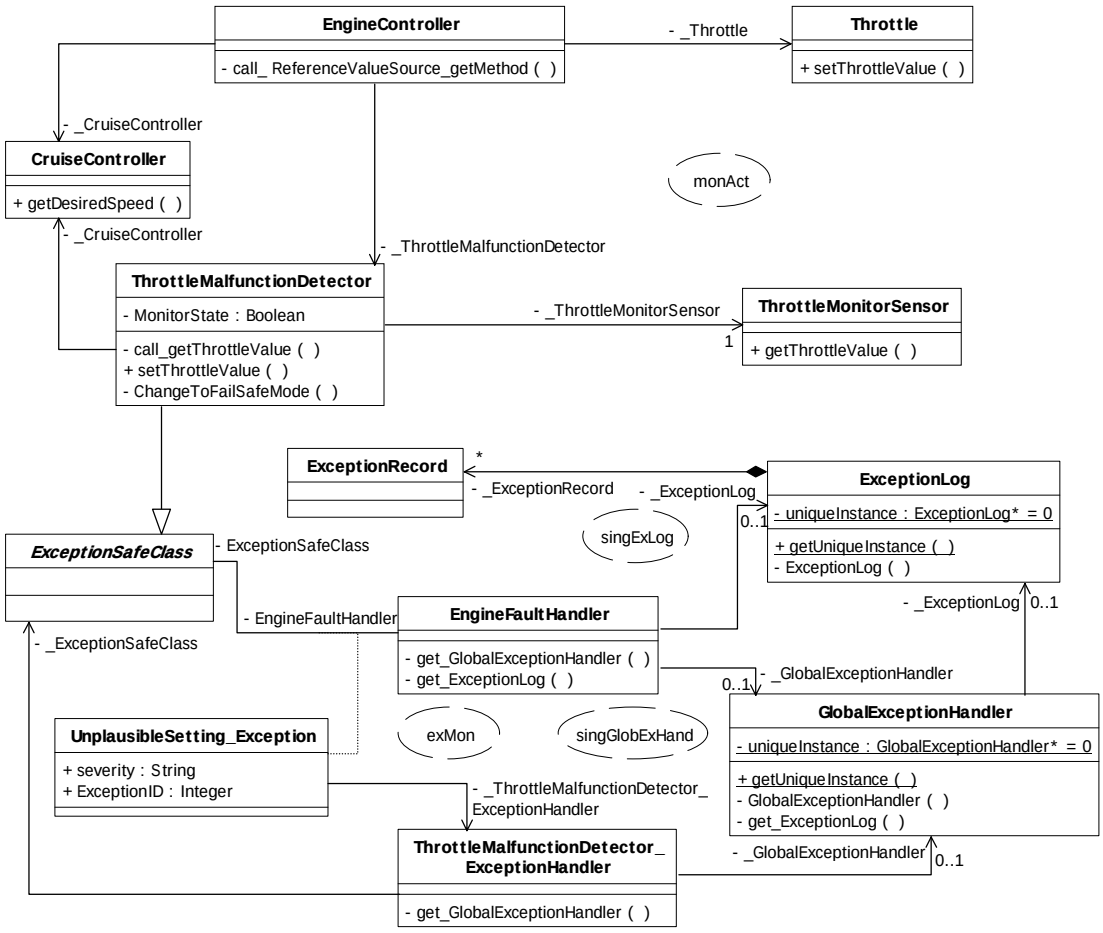
2 Beispiel

2.1 Analysemodell



2.2 Designmodell

Frage	**Antwort**
Optimierungskriterium:	(wird in dieser Regel nicht berücksichtigt)
Klasse die den Kontext der Überwachung darstellt:	*EngineController*
Name des Überwachungssensors:	*ThrottleMonitorSensor*
Klasse die den Sollwert liefert:	*CruiseController*
Methode für die Abfrage des Sollwerts:	*getDesiredSpeed*



B.5 Fault Handler

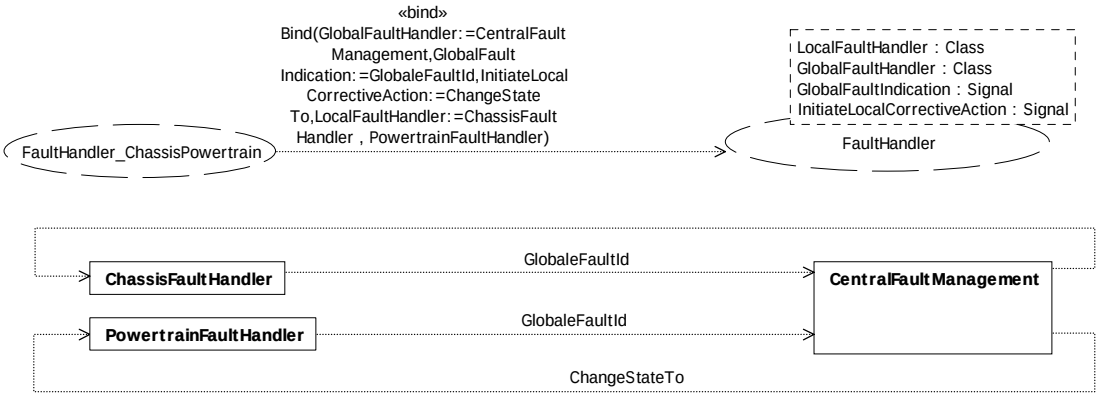
1 Transformationsregel

```
Rule FaultHandler():
Template FaultHandler fh ->
// Fehler auf der Ebene einer Objektkollaboration erkennen und behandeln
Template ExceptionMonitor(
  getNames("Namen der Klassen die einen Fehler erzeugt",-1),    // Client
  "ExceptionSafeClass",                                        // ExceptionSafeClass
  fh.LocalFaultHandler*"_ExceptionHandler",                    // ExceptionHandler
  fh.LocalFaultHandler*"_Exception",                           // Exception
  fh.LocalFaultHandler                                         // ExceptionMonitor
) exMon,
// Sicherstellen dass es nur Instanz von GlobalExceptionMonitor gibt
Template Singleton(
  "GlobalExceptionHandler",                      // Singleton
  exMon.ExceptionMonitor+exMon.ExceptionHandler  // Client
) singGlobExHand,
// Sicherstellen dass es nur eine Instanz von ExceptionLog gibt
Template Singleton(
  "ExceptionLog",                                // Singleton
```

```
  "GlobalExceptionHandler"+exMon.ExceptionMonitor  // Client
) singExLog.
```

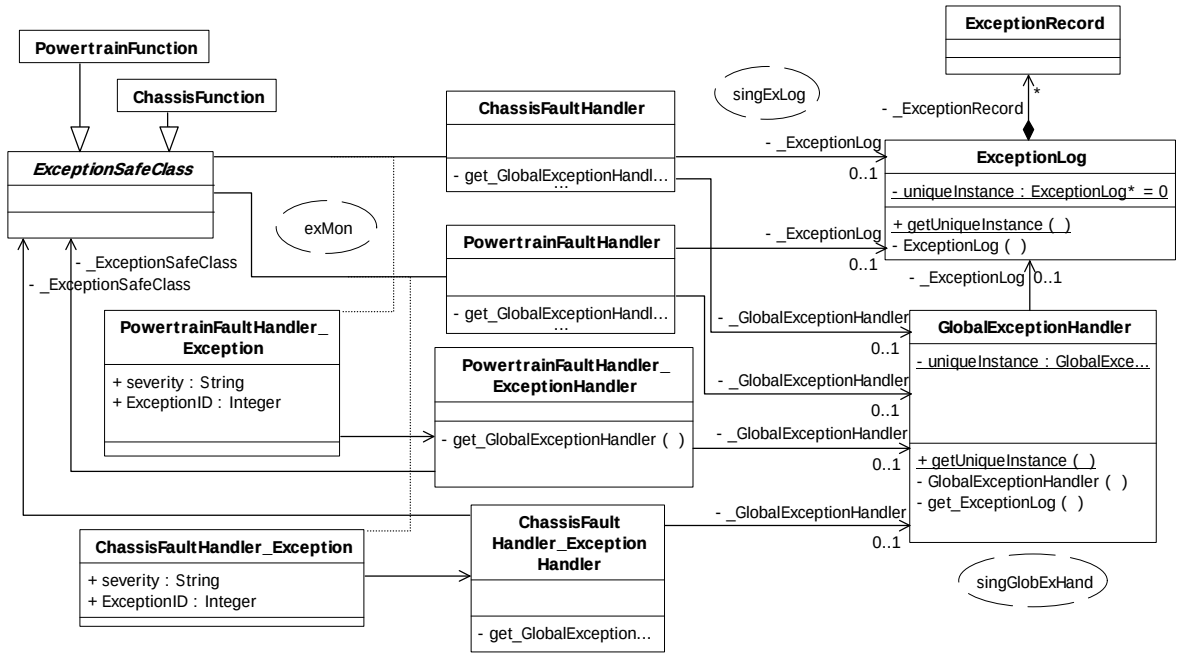
2 Beispiel

2.1 Analysemodell



2.2 Designmodell

Frage	Antwort
Optimierungskriterium:	(wird in dieser Regel nicht berücksichtigt)
Namen der Klassen die einen Fehler erzeugt:	*PowertrainFunction, ChassisFunction*



(Einige Rollennamen sind aus Platzgründen ausgeblendet)

B.6 Hilfsregeln

1 Hilfsregel makeStandardSafety

```
Rule makeStandardSafety(loop):
 ->
// Den Regelungsalgorithmus absichern
Template SecondGuess (
  loop,                 // Context
  "ControlAlgorithm",   // AbstractAlgorithm
  "Short"+"Long",       // Concrete
  getNames("Interface-Name des redundanten Regelalgorithmus",1) // AlgorithmInterface
) controlSG,
// Zeitüberwachung unterstützen
Template Watchdog (
  "Watchdog",     // Watchdog
  loop            // MonitoredChannel
) watchDog.
```

2 Hilfsregel makeStates

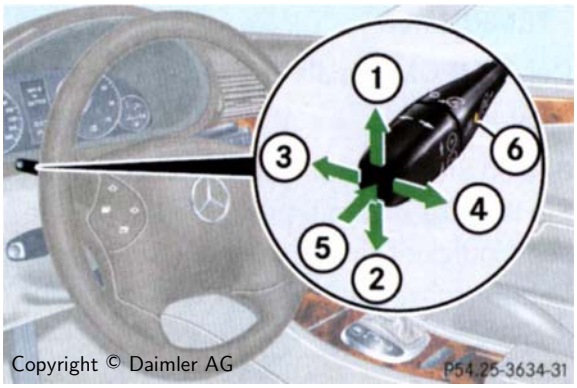
```
Rule makeStates(context):
 ->
[UserQuestion("Ist die Klasse " * context * " zustandsbehaftet?",false,state)]
  ?Template State(
    context, // Context
    getNames("Name der Zustands-Klasse",1),       // State
    getNames("Namen der konkreten Zustände",-1),  // ConcreteState
    getNames("Namen der Zustandsoperationen",-1)  // operation
  ).
```

Anhang – C: Fallstudie

Die folgenden Inhalte zu der in Kapitel 2 vorgestellten Fallstudie *Kombiinstrument- und Fahrfunktionen-System* (KFS) entstanden im Rahmen der studentischen Arbeiten (siehe Kapitel 6.2). Das Designmodell wurde unter Verwendung von POTAD-Transformationsregeln aus dem Analysemodell erzeugt und in geringem Umfang nachbearbeitet. Dabei wurde das Optimierungskriterium „Sicherheit und Verfügbarkeit“ gewählt und alle Fragen, bis auf die nach einem Datenbus und alternativen Regelungsstrategien, negativ beantwortet. Da diese studentischen Arbeiten durchgeführt wurden, als die Entwicklung von POTAD noch nicht abgeschlossen war, entsprechen die eingesetzten Regeln und Muster nicht dem in Anhang – A und Anhang – B dokumentierten Stand. Das Analysemodell verwendet zudem nicht die in anderen Analysemodellen dieser Arbeit eingesetzten *Information Flows*, sondern eine Klassendarstellung mit Assoziationen, die der in Abbildung 3.14 gezeigten Modellierungsweise nahe kommt. Die Abweichungen bewegen sich aber insgesamt in einem Rahmen, der ein Verständnis ohne weitere Erläuterungen möglich macht.

C.1 Kundenanforderungen

Das zu entwickelnde System soll folgende Funktionen und Wartungsaufgaben übernehmen: Tempomat, *Speedtronic*, Tages- und Kilometerzähler, digitaler Tachometer, Wartungsanzeige, Fehleranzeige, Anzeige des Benzinverbrauchs, Anzeige der Durchschnittsgeschwindigkeit, Reichweite abfragen:



1 Benutzerinterface Tempomat/Speedtronic

1. Ein Hebel auf der linken Seite des Lenkrads ist für die Tempomat- bzw. *Speedtronic*-Steuerung zuständig:
   (1) Hebel kann in Pfeilrichtung nach oben bewegt werden.
   (2) Hebel kann in Pfeilrichtung nach unten bewegt werden.
   (3) Hebel kann in Pfeilrichtung nach vorne bewegt werden.
   (4) Hebel kann in Pfeilrichtung nach hinten bewegt werden.

(5) Hebel kann in Pfeilrichtung nach innen gedrückt werden.

(6) Statusanzeige leuchtet, falls *Speedtronic* aktiv ist.

2 Tempomat

1. Der Tempomat hält eine vorgegebene Geschwindigkeit über eine beliebige Strecke oder Zeit.
2. Nur bei laufendem Motor aktiv.
3. Jede Geschwindigkeit über 30 km/h einstellbar.
4. Eingaben des Fahrers:

   Hebel nach oben, um aktuelle oder höhere Geschwindigkeit zu speichern.

   Hebel nach unten, um aktuelle oder niedrigere Geschwindigkeit zu speichern.

   Hebel nach vorne, um Tempomat auszuschalten.

   Hebel nach hinten, um die zuletzt gespeicherte Geschwindigkeit abzurufen.

   Hebel nach innen drücken, um zwischen Tempomat und *Speedtronic* zu wechseln.

5. Durch kurzzeitiges Betätigen des Hebels nach oben bzw. nach unten wird die gegenwärtige Geschwindigkeit gespeichert und konstant gehalten.
6. Der Fahrer hat nun die Möglichkeit durch Antippen des Hebels nach oben bzw. nach unten die Geschwindigkeit um jeweils 1 km/h zu erhöhen bzw. zu verringern.
7. Ein längeres Betätigen des Hebels nach oben bzw. nach unten hat zur Folge, dass das Fahrzeug kontinuierlich beschleunigt bzw. verlangsamt wird.
8. Sobald der Fahrer das Gaspedal durchdrückt, wird das Fahrzeug beschleunigt und beim Loslassen stellt sich wiederum die gespeicherte Geschwindigkeit ein.
9. Der Tempomat wird deaktiviert, wenn das Bremspedal betätigt wird oder durch Drücken des Hebels, um in den *Speedtronic*-Modus zu wechseln.

3 Speedtronic

1. *Speedtronic* sorgt dafür, dass eine bestimmte Geschwindigkeit nicht überschritten wird.
2. Nur bei laufendem Motor aktiv.
3. Jede Geschwindigkeit über 30 km/h begrenzbar.
4. Eingaben des Fahrers:

   Hebel nach oben, um aktuelle oder höhere Geschwindigkeit zu speichern, gerundet auf den nächstgrößeren Zehnerwert.

   Hebel nach unten, um aktuelle oder niedrigere Geschwindigkeit zu speichern, gerundet auf den nächstkleineren Zehnerwert.

   Hebel nach vorne, um die Geschwindigkeitsbegrenzung zu deaktivieren.

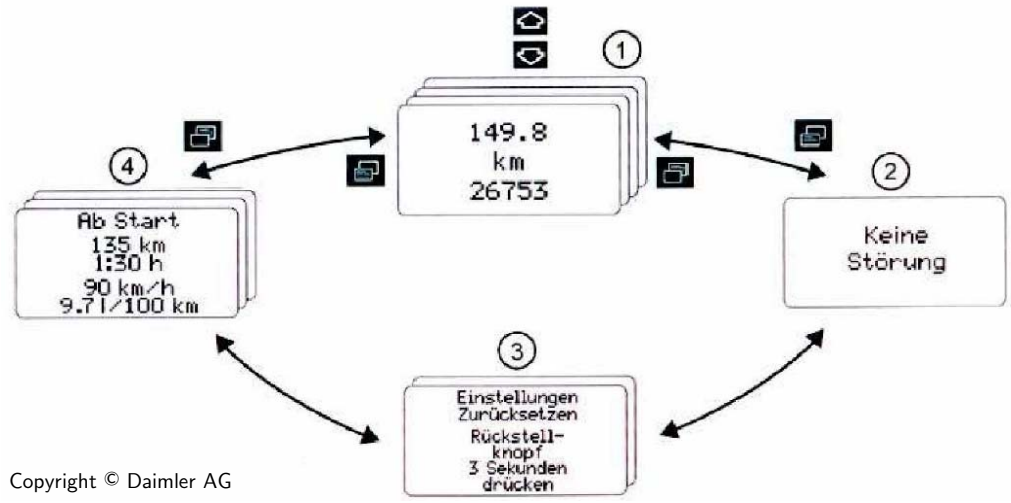
Hebel nach hinten, um die zuletzt gespeicherte Geschwindigkeitsbegrenzung abzurufen oder eine Feineinstellung in 1 km/h-Schritten vorzunehmen.

Hebel nach innen drücken, um zwischen Tempomat und *Speedtronic* zu wechseln.

5. Durch kurzzeitiges Betätigen des Hebels nach oben bzw. nach unten wird die gegenwärtige Geschwindigkeitsbegrenzung aufgerundet und gespeichert.
6. Der Fahrer hat nun die Möglichkeit durch Antippen des Hebels nach oben bzw. nach unten die Geschwindigkeit um jeweils 10 km/h zu erhöhen bzw. zu verringern.
7. Ein längeres Betätigen des Hebels nach oben bzw. nach unten hat zur Folge, dass die Begrenzung kontinuierlich erhöht bzw. verringert wird.
8. Mit dem Betätigen des Hebels nach hinten kann eine Feineinstellung in 1 km/h-Schritten vorgenommen werden.
9. Sobald der Fahrer das Gaspedal über den Druckpunkt (Kick-down) hinaustritt und gleichzeitig die aktuelle Geschwindigkeit nicht mehr als 20 km/h von der begrenzten abweicht, schaltet sich die *Speedtronic* ab.
10. Bei Überschreiten der Geschwindigkeit ertönt ein akustisches Signal und es erscheint eine Meldung.
11. Die Geschwindigkeitsbegrenzung wird beim Speichern angezeigt.

4 Benutzerinterface Bordcomputer

1. Der Bordcomputer wird über die Tasten am Multifunktionslenkrad gesteuert und die Anzeige erfolgt über ein Multifunktions-Display (Punkte 2 und 3 nicht relevant):
   (1) Multifunktionsdisplay.
   (2) Untermenü auswählen (aufwärts, abwärts).
   (3) Telefonieren (Gespräch annehmen, Gespräch beenden).
   (4) Von Menü zu Menü springen (vor, zurück).
   (5) Im Menü springen (vor, zurück).
2. Das Multifunktions-Display verfügt über eine Menüführung:



(1) Betriebsmenü:

Standardanzeige mit Tages- und Kilometerzähler.

Digitaler Tachometer.

Wartungsanzeige.

(2) Fehleranzeige:

Störungen abfragen.

(3) Einstellungen:

Auf Werkseinstellungen zurücksetzen.

Schaltprogramm wählen.

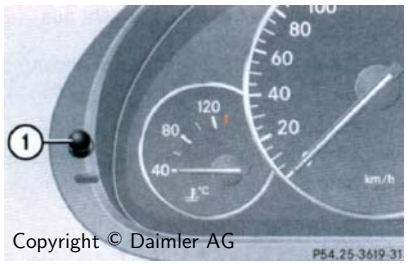
(4) Reiserechner:

Verbrauchs- und Geschwindigkeitsstatistiken ab Start.

Verbrauchs- und Geschwindigkeitsstatistiken ab Reset.

Reichweite abfragen.

1. Über den Rückstellknopf auf der linken Seite des Cockpits lassen sich Einstellungen wieder zurücksetzen.



(1) Rückstellknopf

5 Tages- und Kilometerzähler

1. Anzeige der gefahrenen Tageskilometer (Rücksetzen mit Rückstellknopf).
2. Anzeige der gesamt gefahrenen Kilometer.

6 Digitaler Tachometer

1. Anzeige der aktuellen Geschwindigkeit.

7 Wartungsanzeige

1. Der Service findet alle 25.000 km statt.
2. Informationen über den nächsten Service-Termin werden angezeigt:

   In Kilometern.

   Termin ist fällig.
3. Sobald der Service-Termin überschritten wird, werden die überzogenen Kilometer angezeigt und außerdem ertönt ein akustisches Signal.
4. Die Service-Anzeige geht nach 30 Sekunden automatisch aus oder mit Hilfe des Rückstellknopfes.
5. Service-Termine können jederzeit abgerufen werden.
6. Die Wartungsanzeige kann nur durch einen Service-Techniker zurückgesetzt werden.

8 Fehleranzeige

1. Sobald Störungen im Kfz auftreten, werden diese von einem Kontrollmodul erfasst und können in Form von Textmeldungen oder Symbolen im Display angezeigt werden.
2. Das Fehleranzeigemenü wird automatisch aufgerufen, wenn ein Fehler während der Fahrt auftaucht oder wenn das Fahrzeug gestartet wird und die Störung noch nicht beseitigt worden ist.
3. Die Fehlermeldungen können mit jeder beliebigen Taste (4 und 5 am Multifunktionlenkrad) oder dem Rückstellknopf quittiert werden.

9 Auf Werkseinstellungen zurücksetzen

1. Das Betätigen des Rückstellknopfes über einen Zeitraum von 3 Sekunden bewirkt ein Rücksetzen auf Werkseinstellungen.
2. Alle Einstellungen (Verbrauch, Durchschnittsgeschwindigkeit, Tageskilometer, Schaltprogramm) werden auf „0“ bzw. auf Standardwerte gesetzt.

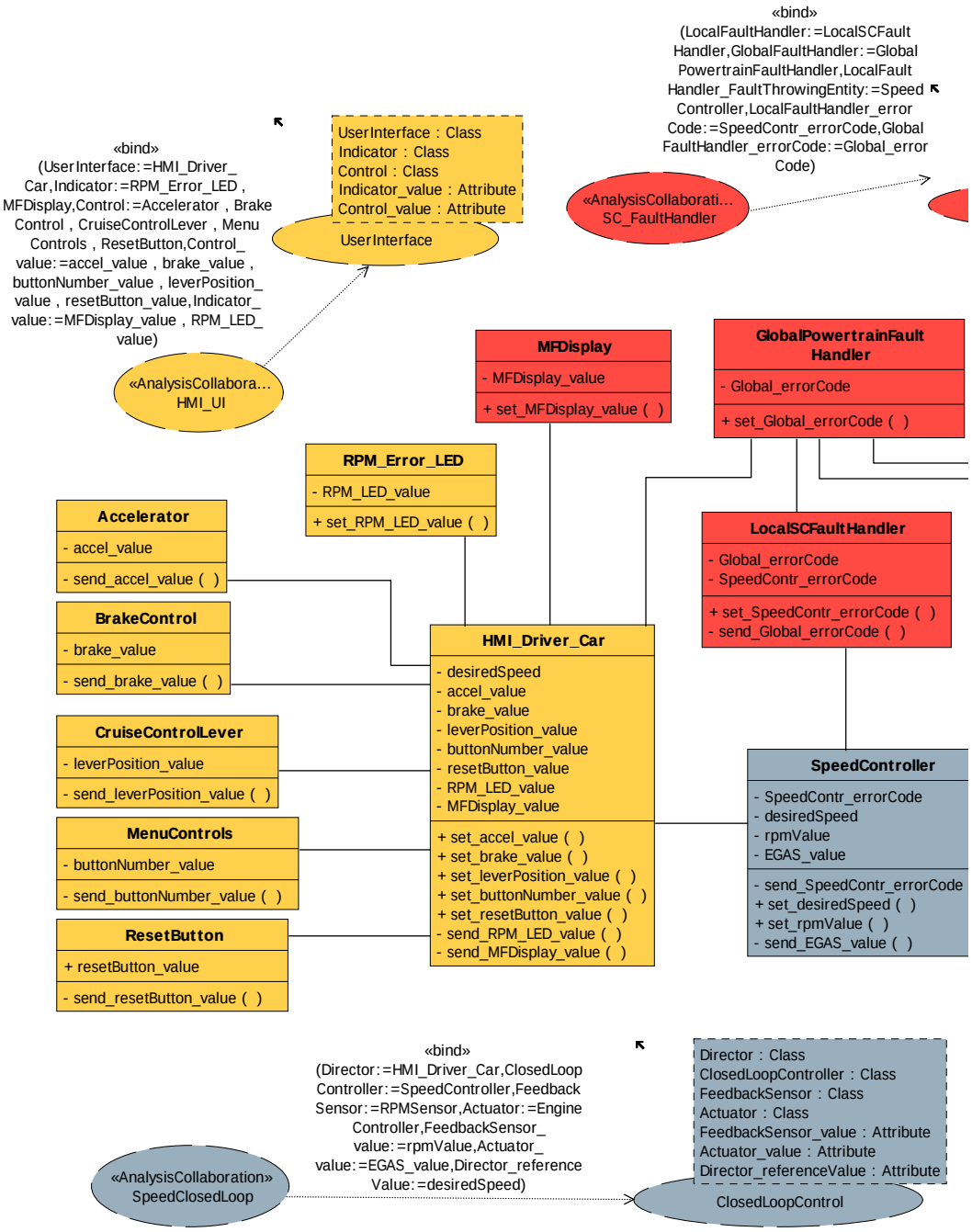
10 Schaltprogramm auswählen

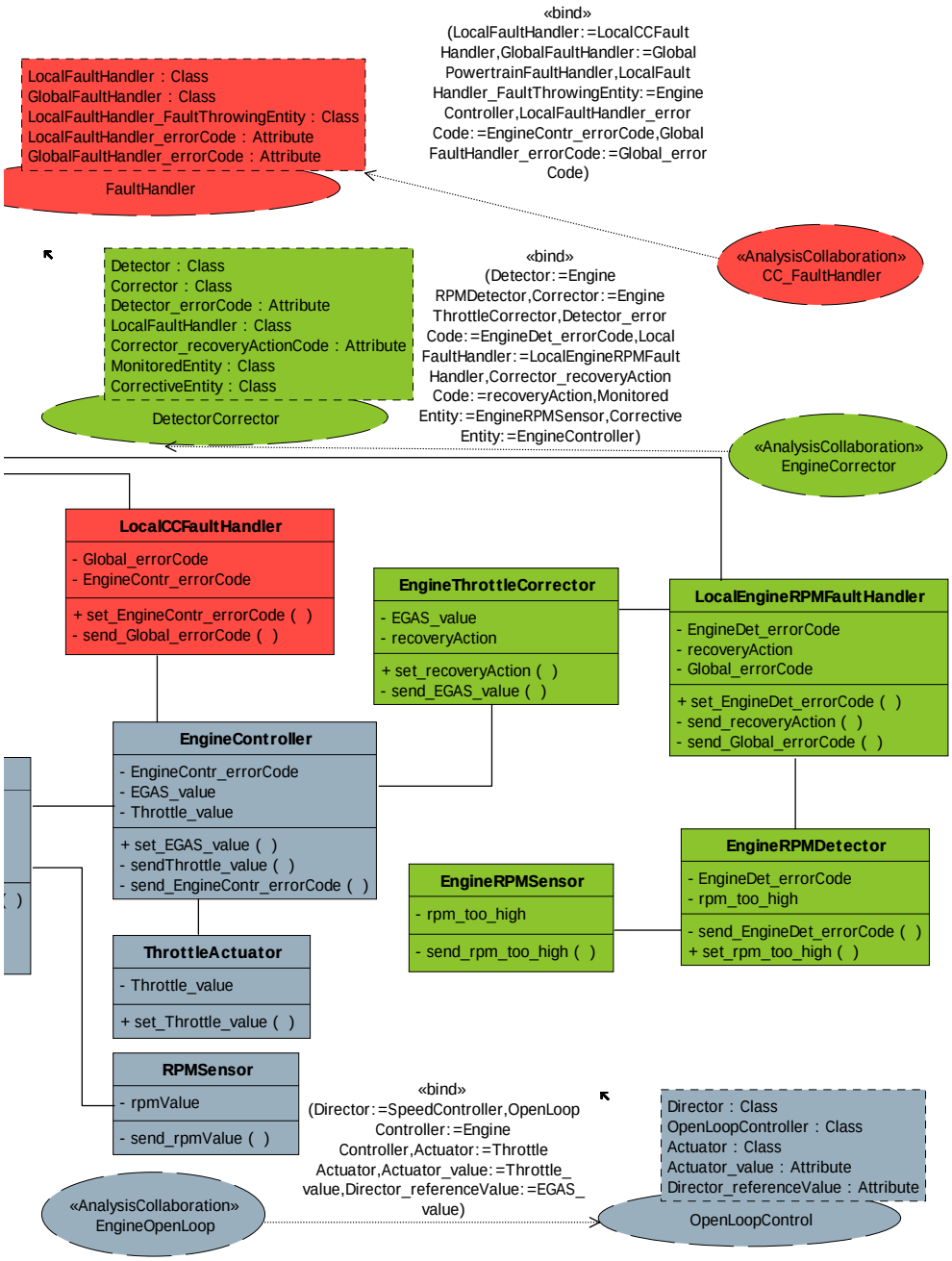
1. Es kann ein Programm (C bzw. S) über den Rückstellknopf gewählt werden, dass die Schaltvorgänge beim Automatikgetriebe optimiert und außerdem das Anfahr- bzw. Bremsverhalten verändert.
2. Schaltprogramme:

   „Comfort“ für sanfteres Anfahren, Fahrstabilitätsverbesserung und frühes Hochschalten.

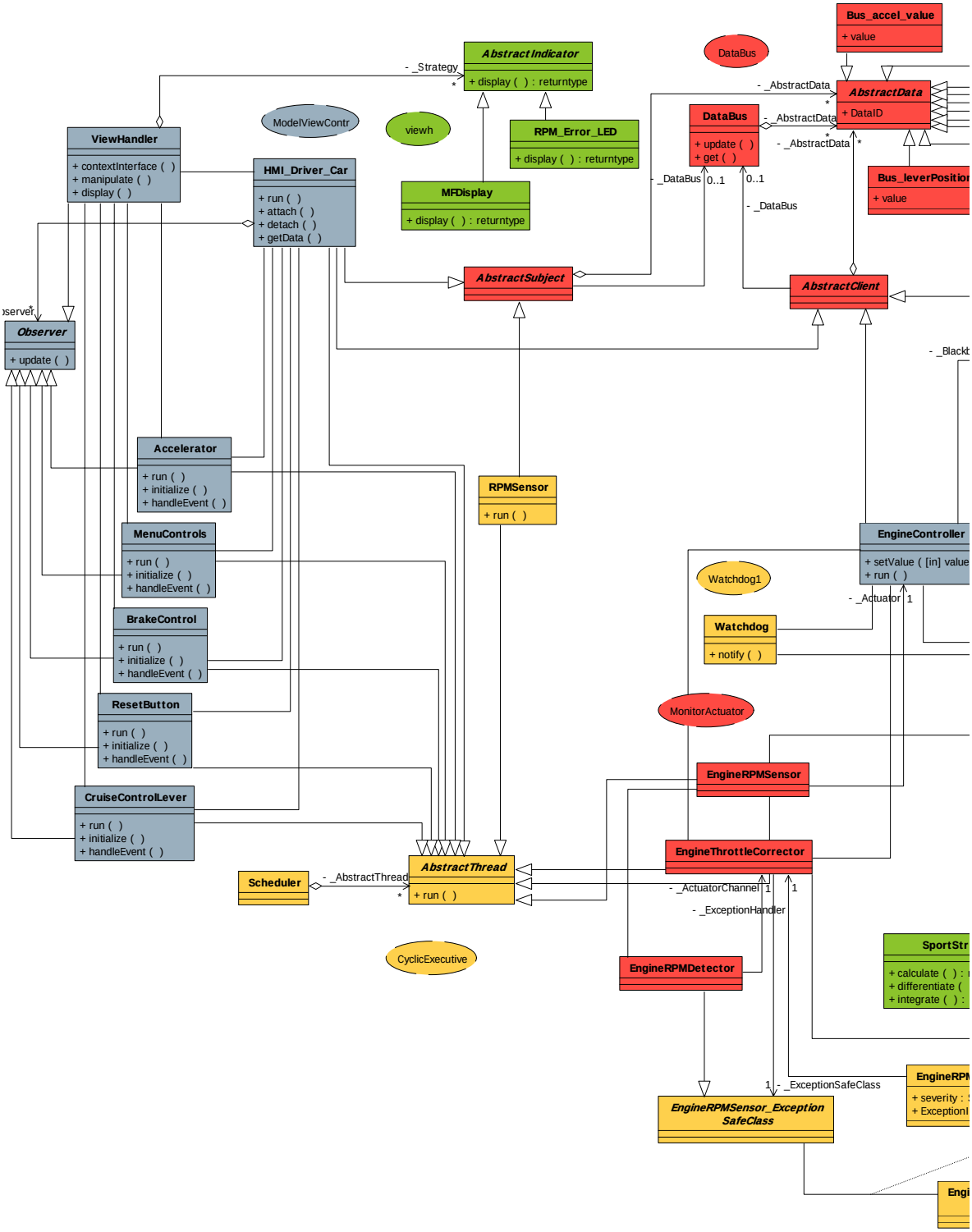
   „Sport“ für alle normalen Fahrsituationen.

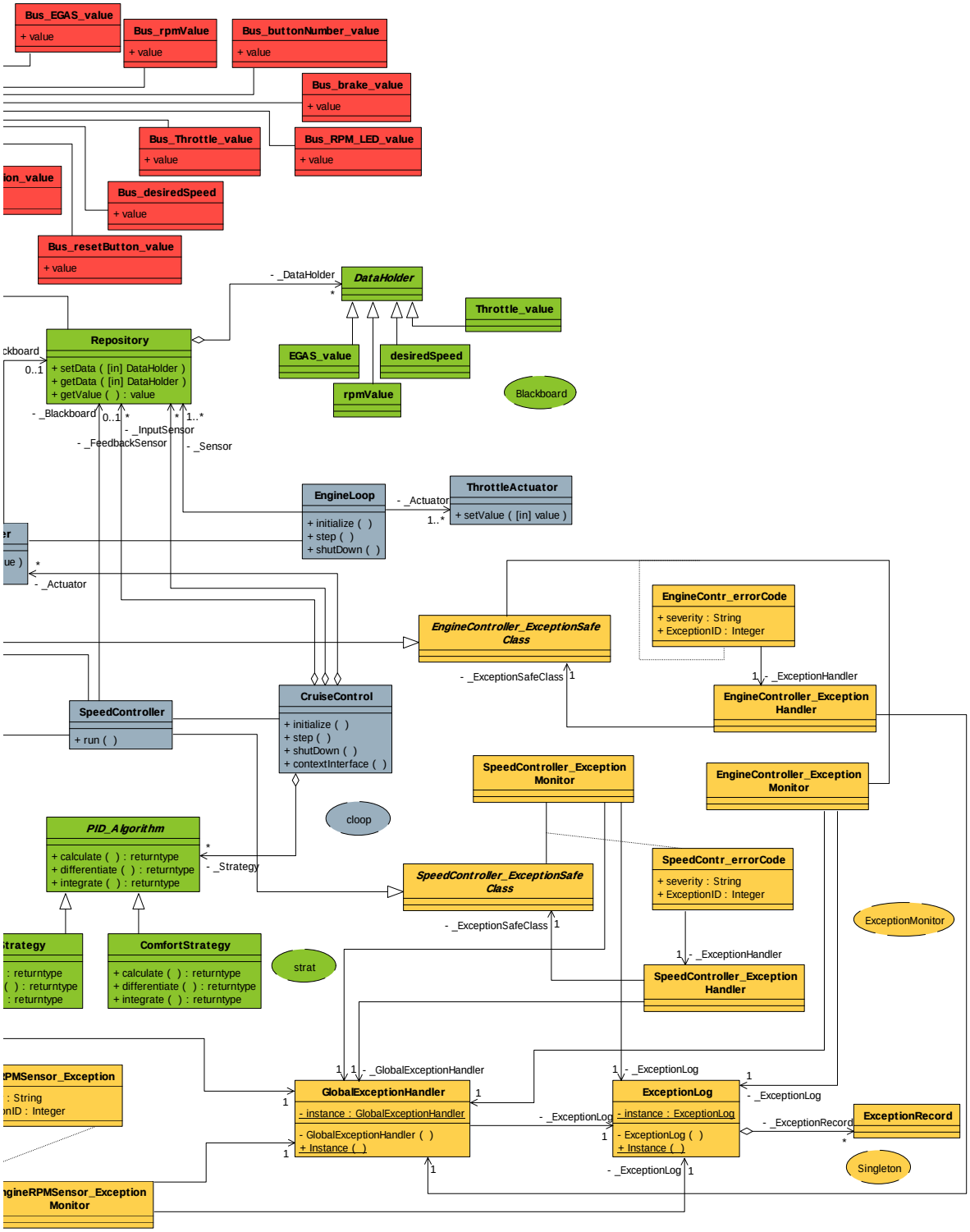
C.2 Analysemodell




«bind»
(LocalFaultHandler:=LocalCCFault Handler,GlobalFaultHandler:=Global PowertrainFaultHandler,LocalFault Handler_FaultThrowingEntity:=Engine Controller,LocalFaultHandler_error Code:=EngineContr_errorCode,Global FaultHandler_errorCode:=Global_error Code)
LocalFaultHandler : Class
GlobalFaultHandler : Class
LocalFaultHandler_FaultThrowingEntity : Class
LocalFaultHandler_errorCode : Attribute
GlobalFaultHandler_errorCode : Attribute
FaultHandler
«AnalysisCollaboration»
CC_FaultHandler
Detector : Class
Corrector : Class
Detector_errorCode : Attribute
LocalFaultHandler : Class
Corrector_recoveryActionCode : Attribute
MonitoredEntity : Class
CorrectiveEntity : Class
DetectorCorrector
«bind»
(Detector:=Engine RPMDetector,Corrector:=Engine ThrottleCorrector,Detector_error Code:=EngineDet_errorCode,Local FaultHandler:=LocalEngineRPMFault Handler,Corrector_recoveryAction Code:=recoveryAction,Monitored Entity:=EngineRPMSensor,Corrective Entity:=EngineController)
«AnalysisCollaboration»
EngineCorrector
LocalCCFaultHandler
- Global_errorCode
- EngineContr_errorCode
+ set_EngineContr_errorCode ( )
- send_Global_errorCode ( )
EngineThrottleCorrector
- EGAS_value
- recoveryAction
+ set_recoveryAction ( )
- send_EGAS_value ( )
LocalEngineRPMFaultHandler
- EngineDet_errorCode
- recoveryAction
- Global_errorCode
+ set_EngineDet_errorCode ( )
- send_recoveryAction ( )
- send_Global_errorCode ( )
EngineController
- EngineContr_errorCode
- EGAS_value
- Throttle_value
+ set_EGAS_value ( )
- sendThrottle_value ( )
- send_EngineContr_errorCode ( )
EngineRPMDetector
- EngineDet_errorCode
- rpm_too_high
- send_EngineDet_errorCode ( )
+ set_rpm_too_high ( )
EngineRPMSensor
- rpm_too_high
- send_rpm_too_high ( )
ThrottleActuator
- Throttle_value
+ set_Throttle_value ( )
RPMSensor
- rpmValue
- send_rpmValue ( )
«bind»
(Director:=SpeedController,OpenLoop Controller:=Engine Controller,Actuator:=Throttle Actuator,Actuator_value:=Throttle_ value,Director_referenceValue:=EGAS_ value)
Director : Class
OpenLoopController : Class
Actuator : Class
Actuator_value : Attribute
Director_referenceValue : Attribute
«AnalysisCollaboration»
EngineOpenLoop
OpenLoopControl

C.3 Designmodell




Bus_EGAS_value
+ value
Bus_rpmValue
+ value
Bus_buttonNumber_value
+ value
Bus_brake_value
+ value
Bus_Throttle_value
+ value
Bus_RPM_LED_value
+ value
Bus_desiredSpeed
+ value
Bus_resetButton_value
+ value
- _DataHolder
DataHolder
Throttle_value
Repository
+ setData ([in] DataHolder)
+ getData ([in] DataHolder)
+ getValue ( ) : value
EGAS_value
desiredSpeed
rpmValue
Blackboard
- _Blackboard
- _InputSensor
- _FeedbackSensor
- _Sensor
EngineLoop
+ initialize ( )
+ step ( )
+ shutDown ( )
- _Actuator
ThrottleActuator
+ setValue ([in] value)
EngineContr_errorCode
+ severity : String
+ ExceptionID : Integer
EngineController_ExceptionSafe Class
- _ExceptionSafeClass
- _ExceptionHandler
EngineController_Exception Handler
SpeedController
+ run ( )
CruiseControl
+ initialize ( )
+ step ( )
+ shutDown ( )
+ contextInterface ( )
SpeedController_Exception Monitor
EngineController_Exception Monitor
cloop
PID_Algorithm
+ calculate ( ) : returntype
+ differentiate ( ) : returntype
+ integrate ( ) : returntype
- _Strategy
SpeedController_ExceptionSafe Class
SpeedContr_errorCode
+ severity : String
+ ExceptionID : Integer
ExceptionMonitor
ComfortStrategy
+ calculate ( ) : returntype
+ differentiate ( ) : returntype
+ integrate ( ) : returntype
strat
SpeedController_Exception Handler
- _GlobalExceptionHandler
GlobalExceptionHandler
- instance : GlobalExceptionHandler
- GlobalExceptionHandler ( )
+ Instance ( )
- _ExceptionLog
ExceptionLog
- instance : ExceptionLog
- ExceptionLog ( )
+ Instance ( )
- _ExceptionRecord
ExceptionRecord
Singleton

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Agrawal '04] Agrawal, A.: *A Formal Graph-Transformation Based Language for Model-to-Model Transformations*. *PhD Dissertation, Vanderbilt University, Dept of EECS*. 2004. URL: http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_8_0_2004_A_Formal_G.pdf.

[Akehurst '04] Akehurst, D.: *Relations in OCL*. *UML 2004 Workshop OCL and Model Driven Engineering*. Lisbon, Portugal, 2004. URL: http://www.cs.kent.ac.uk/pubs/2004/2007.

[Akehurst et al. '02] Akehurst, D.; Kent, S.: *A Relational Approach to Defining Transformations in a Metamodel*. In (Jezequel, J.-M.,Hussmann, H. Hrsg.): *2002 - The Unified Modeling Language: Model Engineeing, Concepts, and Tools*. Ausgabe 2460. Springer, 2002. URL: http://www.cs.ukc.ac.uk/pubs/2002/1559.

[Alexander '77] Alexander, Christopher: *A Pattern Language*. Oxford University Press, 1977.

[Aquintos] Aquintos: *PREEvision (Produktwebseite)*. URL: http://www.aquintos.de/de/products/ee.php.

[Aquintos '06] Aquintos: *Benutzerhandbuch PREEvision Release 1.0*. 2006.

[Arora et al. '98] Arora, A.; Kulkarni, S.: *Detectors and correctors: A theory of fault-tolerance.*: *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. 1998. S. 436–443.

[ATESST] ATESST: *ATESST (Projektwebseite)*. URL: http://www.atesst.org.

[AUTOSAR] AUTOSAR: *AUTomotive Open System ARchitecture*. URL: http://www.autosar.org.

[Balzert '99] Balzert, Heide: *Lehrbuch der Objektmodellierung*. Spektrum Akademischer Verlag, 1999.

[Beck et al. '87] Beck, Kent; Cunningham, Ward: *Using Pattern Languages for Object-Oriented Programs*. *Proceedings of OOPSLA'87*. 1987.

[Belschner et al. '05] Belschner, Ralf; Bortolazzi, Jürgen; Frees, Jascha; Hettich, Gerhard; Mroßko, Markus: *Gesamtheitlicher Entwicklungsansatz für Entwurf, Dokumentation und Bewertung von E/E-Architekturen*. Ausgabe II/2005. Automotive Electronics, 2005. S. 18-23.

[Bertram et al. '01] Bertram, Torsten; Opgen-Rhein, Peter: *Modellbildung und Simulation mechatronischer Systeme - Virtueller Fahrversuch als Schlüsseltechnologie der Zukunft*. *ATZ/MTZ Automotive Electronics*. Ausgabe Ausgabe September 2001. 2001. S. 20-26.

[Boehm '83] Boehm, B.: *Seven Basic Principles of Software Engineering*. *Journal of Systems and Software*. 1983. S. 3-24.

[Born et al. '05] Born, Marc; Holz, Eckhardt; Kath, Olaf: *Softwareentwicklung mit UML 2*. Addison-Wesley, München, 2005.

[BOSCH '03] BOSCH: *Kraftfahrtechnisches Taschenbuch*. Ausgabe 25. Auflage. Vieweg, 2003.

[Braun et al. '02] Braun, P.; Marschall, F.: *Transforming Object Oriented Models with BOTL*. In (Bottoni, P.,Minas, M. Hrsg.): *International Workshop on Graph Transformation and Visual Modeling Techniques, ENTCS 72.3, Elsevier Science B. V.*, 2002. URL: http://www4.in.tum.de/~marschal/pub/entcs-bm02.pdf.

[Braun et al. '03a] Braun, P.; Marschall, F.: *BOTL - The Bidirectional Object Oriented Transformation Language*. Ausgabe Technical Report. 2003a. URL: http://wwwbib.informatik.tu-muenchen.de/infberichte/2003/TUM-I0307.pdf.

[Braun et al. '03b] Braun, P.; Marschall, F.: *Model Transformations for the MDA with BOTL*. *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications*. Univeristy of Twente, 2003b. URL: http://www4.in.tum.de/publ/papers/marschall_braun-mdafa03.pdf.

[Brown et al. '98] Brown, W.; Malveau, R.; McCormick, H.; Mowbray, T.: *AntiPatterns - Refactoring Software, Architectures and Projects in Crisis.* Wiley Verlag, 1998.

[Buchwald '05] Buchwald, Jan: *Eine Transformationssprache für musterbasierte Modelltransformationen*. *Diplomarbeit*. Universtität Ulm, 2005.

[Buschmann '98] Buschmann, Frank: *Pattern-orientierte Software-Architektur - Ein Pattern-System*. Addison-Wesley, 1998.

[Coad et al. '97] Coad, Peter; North, David; Mayfield, Mark: *Object models: strategies, patterns, and applications*. *Yourdon Press computing series*. Yourdon Press, Upper Saddle River, N.J., 1997. S. xviii, 515 p.

[Compuware] Compuware: *OptimalJ (Produktseite)*. URL: http://www.compuware.com/products/optimalj/default.htm.

[Compuware et al. '04a] Compuware; Sun Microsystems: *EXMOF - MOF 2.0 Q/V/T 2nd revised submission presentation*. 2004a. URL: http://www.omg.org/cgi-bin/doc?ad/04-11-06.pdf.

[Compuware et al. '04b] Compuware; Sun Microsystems: *EXMOF - Queries, Views and Transformations on Models using MOF, OCL and EXMOF*. 2004b. URL: http://www.omg.org/docs/ad/04-10-03.pdf.

[Coplien '91] Coplien, James: *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1991.

[Corbett] Corbett, Robert: *Berkeley Yacc (byacc)*. URL: http://dickey.his.com/byacc/byacc.html.

[Cross '02] Cross, Joseph K.: *Proactive and Reactive Resource Allocation*. *PLoP 2002 Proceedings*. 2002. URL: http://jerry.cs.uiuc.edu/%7Eplop/plop2002/final/ProReResReal_Lardieri.pdf.

[Czarnecki et al. '00] Czarnecki, Krzysztof; Eisenecker, Ulrich: *Generative programming: methods, tools, and applications*. Addison Wesley, Boston, 2000. S. xxvi, 832 p.

[Czarnecki et al. '03] Czarnecki, Krzysztof; Helsen, Simon: *Classification of Model Transformation Approaches*. *Online proceedings of the 2nd OOPSLA 03 Workshop on Generative Techniques in the Context of MDA. Anaheim, October, 2003*. 2003. URL: http://www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf.

[Dehayni et al. '03] Dehayni, M.; Féraud, L.: *An Approach of Model Transformation Based on Attribute Grammars*. *Object-Oriented Information Systems*. Ausgabe Lecture Notes in Computer Science. Springer Verlag, 2003. S. 412-423.

[DIN '94] Deutsches Institut für Normung e.V. - DIN: *DIN 19226-1 - Leittechnik; Regelungstechnik und Steuerungstechnik. Allgemeine Grundbegriffe*. 1994.

[Douglass '99] Douglass, Bruce Powel: *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison-Wesley Pub Co, 1999.

[Douglass '02] Douglass, Bruce Powel: *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Professional, 2002.

[Douglass '04] Douglass, Bruce Powel: *Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)*. Addison-Wesley Professional, 2004.

[Fernandez et al. '00] Fernandez, E. B.; Yuan, X.: *Semantic Analysis Patterns*. *Proceeding of the 19th International Conference on Conceptual Modeling (ER 2000)*. 2000. S. 183-195.

[Fischer '05] Fischer, Florian: *Werkzeugunterstützung für musterbasierte Modelltransformationen*. *Diplomarbeit*. Universität Ulm, 2005.

[flex] flex: *flex: The Fast Lexical Analyzer*. URL: http://flex.sourceforge.net/.

[Fowler '97] Fowler, Martin: *Analysis patterns: reusable object models*. *Addison-Wesley series in object-oriented software engineering*. Addison Wesley, Menlo Park, Calif., 1997. S. xxi, 357.

[Gamma et al. '95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. *Addison-Wesley professional computing series.* Addison-Wesley, Reading, Mass., 1995. S. xv, 395.

[Geyer-Schulz et al. '01] Geyer-Schulz, Andreas; Hahsler, Michael: *Software Engineering with Analysis Patterns*. *Technical Report 01/2001*. Institut für Informationsverarbeitung und -wirtschaft, Wirschaftsuniversität Wien, 2001. URL: http://wwwai.wu-wien.ac.at/~hahsler/research/virlib_working2001/virlib/.

[Ghezzi et al. '91] Ghezzi, Carlo; Jazayeri, Mehdi; Mandrioli, Dino: *Fundamentals of software engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991. S. xv, 573 p.

[Gomaa '01] Gomaa, Hassan: *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley, Boston, 2001. S. XXIII, 785 S.

[Guelfi et al. '03a] Guelfi, N.; Perrouin, G.; Sterges, P.; Ries, B.; Sendall, S.: *MEDAL 1.0 Reference*. University of Luxembourg, Faculty of Science, Technology and Communication, 2003a. URL: http://se2c.uni.lu/tiki/se2c-bib_download.php?id=210.

[Guelfi et al. '03b] Guelfi, N.; Ries, B.; Sterges, P.: *MEDAL: A CASE Tool Extension for Model-Driven Software Engineering*. *IEEE International Conference on Software-Science, Technology & Engineering*. IEEE, Herzlia, Israel, 2003b. URL: http://se2c.uni.lu/tiki/se2c-bib_download.php?id=227.

[Hillside Group] Hillside Group: *Patterns home page (Webseite)*. URL: http://hillside.net/patterns.

[Homann '04] Homann, Matthias: *OSEK: Betriebssystem-Standard für Automotive und Embedded Systems*. mitp-Verl., Bonn, 2004. S. 464 S.

[Hudson] Hudson, Scott: *CUP Parser Generator for Java*. URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[IABG '97] Industrieanlagen-Betriebsgesellschaft - IABG: *V-Modell - Entwicklungsstandard für IT-Systeme des Bundes. Vorgehensmodell Kurzbeschreibung.*, 1997. URL: http://www.v-modell.iabg.de/vm97.htm.

[IBM a] IBM a: *Rational Software Architect (Produktseite)*. URL: http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html.

[IBM b] IBM b: *Rational Rose XDE Developer (Produktseite)*. URL: http://www-306.ibm.com/software/awdtools/developer/rosexde/.

[IBM c] IBM c: *MDA toolkit for Rational XDE Java*. URL: http://www-306.ibm.com/software/rational/mda/toolkit.html.

[IESE a] Fraunhofer - IESE a: *UML 1.x Überblick*. *Software-Engineering-Wissensdatenbank*. URL: http://www.software-kompetenz.de.

[IESE b] Fraunhofer - IESE b: *Software-Engineering-Wissensdatenbank*. URL: http://www.software-kompetenz.de.

[IESE c] Fraunhofer - IESE c: *4-Schichten Metamodellarchitektur*. *Software-Engineering-Wissensdatenbank*. URL: http://www.software-kompetenz.de.

[INCOSE] INCOSE, The International Council on Systems Engineering: *A Consensus of the INCOSE Fellows*. URL: http://www.incose.org/practice/fellowsconsensus.aspx.

[ISO/IEC '98] ISO/IEC, International Organization for Standardization/Internationl Electrotechnical Commision: *ISO/IEC 14977:1996: Information technology - Syntactic metalanguage - Extended BNF (EBNF)*. 1998.

[ISO/IEC '05] ISO/IEC, International Organization for Standardization/Internationl Electrotechnical Commision: *19501:2005. Information technology - Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.2*. 2005.

[Jacobson et al. '99] Jacobson, Ivar; Booch, Grady; Rumbaugh, James: *The unified software development process*. Addison-Wesley, 1999.

[Jeckle et al. '03] Jeckle, Mario; Rupp, Chris; Hahn, Jürgen; Zengler, Barbara; Queins, Stefan: *UML 2 glasklar*. Hanser Fachbuchverlag, 2003.

[Judson et al. '03] Judson, S.; France, R.; Carver, D.: *Specifying Model Transformations at the Metamodel Level. Workshop in Software Model Engineering, WiSME@UML'2003*. San Francisco, USA, 2003. URL: http://www.metamodel.com/wisme-2003/19.pdf.

[Kalnins et al. '04] Kalnins, A.; Barzdins, J.; Celms, E.: *Model Transformation Language MOLA: Extended Patterns*. *Proceedings of Baltic DB|&IS 2004*. Riga, Latvia, 2004. URL: http://melnais.mii.lu.lv/audris/RuleLang3_Final.pdf.

[Karsai et al. '03] Karsai, G.; Agrawal, A.: *Graph Transformations in OMG's Model-Driven Architecture*. *Application of Graph Transformations with Industrial Relevance, Proc. AGTIVE 2003*. Ausgabe Lecture Notes in Computer Science. Springer, 2003. S. 243-259. URL: http://www.isis.vanderbilt.edu/publications/archive/Karsai_G_12_0_2003_Graph_Tran.pdf.

[Kempa et al. '05] Kempa, Martin; Mann, Zoltán Ádám: *Model Driven Architecture*. In (Die Gesellschaft für Informatik e.V. Hrsg.): *Informatiklexikon*. 2005.

[Khriss et al. '99a] Khriss, I.; Keller, R.; Hamid, I.: *Supporting Design by Pattern-based Transformations*. *Proceedings of the Second International Workshop on Strategic Knowledge and Concept Formation*. Iwate, Japan, 1999a. S. 157-167. URL: http://se2c.uni.lu/tiki/se2c-bib_download.php?id=612.

[Khriss et al. '99b] Khriss, Ismaïl; Keller, Rudolf K.; Hamid, Issam A.: *Supporting Design by Pattern-based Transformations*. *Proceedings of the International Workshop on Software Transformation Systems (STS'99)*. Los Angeles, CA, USA, 1999b. S. 50-58. URL: http://www.iro.umontreal.ca/~keller/Publications/Papers/sts99.pdf.

[Kircher et al. '01] Kircher, Michael; Jain, Prashant; Schmidt, Douglas; Corsaro, Angelo: *Proceedigns of th Wokshop "Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems"*. OOPSLA Workshop, Tampa, 2001. URL: http://www.posa3.org/workshops/RealTimePatterns/OOPSLA2001/.

[Kircher et al. '02] Kircher, Michael; Jain, Prashant; Schmidt, Douglas; Gokhale, Andy: *Proceedigns of th Wokshop on Patterns in Distributed Real-Time and Embedded Systems*. OOPSLA Workshop, Washington, USA, 2002. URL: http://www.posa3.org/workshops/RealTimePatterns/OOPSLA2002/.

[Klein] Klein, Gerwin: *JFlex - The Fast Scanner Generator for Java*. URL: http://www.jflex.de/.

[Kleppe et al. '03] Kleppe, Anneke; Warmer, Jos; Bast, Wim: *MDA Explained: The Model Driven Architecture - Practice and Promise*. Addison-Wesley Professional, 2003.

[Koenig '95] Koenig, A.: *Patterns and Antipatterns*. *Journal of Object-Oriented Programming*. Ausgabe March. 1995.

[Kolodziejczyk '05] Kolodziejczyk, Gregor: *Der Einsatz von Mustern bei Analyse und Design von eingebetteten Echtzeitsystemen*. *Diplomarbeit*. Fachhochschule Bochum, 2005.

[Konrad et al. '04a] Konrad, Sascha; H. C. Cheng, Betty; Campbell, Laura A.: *Object Analysis Patterns for Embedded Systems*. Department of Computer Science, Michigan State University, East Lansing, Michigan, 2004a.

[Konrad et al. '04b] Konrad, Sascha; Cheng, Betty H. C.; Campbell, Laura A.: *Object Analysis Patterns for Embedded Systems*. *IEEE Transactions on Software Engineering*. Ausgabe 30. IEEE, 2004b. S. 970-992.

[Kruchten '04] Kruchten, Philippe: *The Rational Unified Process. An Introduction.: An Introduction*. *Addison-Wesley Object Technology Series*. Addison-Wesley Longman, Amsterdam, 2004.

[Lalanda '98] Lalanda, Philippe: *Shared repository pattern*. *Proceedings of Pattern Languages of Programs '98*. 1998. URL: jerry.cs.uiuc.edu/~plop/plop98/final_submissions/P24.pdf.

[Lea et al. '94] Lea, D.; Alexander, Christopher: *An Introduction for Object-Oriented Designers*. *ACM Software Engineering Notes*. 1994. URL: http://g.oswego.edu/dl/ca/ca/ca.html.

[Ledeczi et al.] Ledeczi, A.; Maroti, M.; Bakay, A.; Karsai, G.; Garrett, J.; Thomason, C.; Nordstrom, G.; Sprinkel, J.; Volgyesi, P.: *The Generic Modeling Environment*. URL: http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf.

[Lönn et al. '04] Lönn et al., Henrik: *Report: Definition of language for automotive embedded electronic architecture description (Version 1.02)*. EAST-EEA - Embedded Electronic Architecture, 2004. URL: http://www.east-eea.net.

[Milicev '02] Milicev, Dragan: *Domain Mapping Using Extended UML Object Diagrams*. *IEEE Softw.* Ausgabe 19. IEEE Computer Society Press, 2002. S. 90-97. URL: http://se2c.uni.lu/tiki/se2c-bib_download.php?id=615.

[Miller et al. '03] Miller, Joaquin; Mukerji, Jishnu: *MDA Guide Version 1.0.1*. OMG, 2003. URL: http://www.omg.org/docs/omg/03-06-01.pdf.

[Oestereich '04] Oestereich, Bernd: *Die UML 2.0 Kurzreferenz für die Praxis. Kurz, bündig, ballastfrei*. Verlag Oldenbourg, 2004.

[OMG '02a] Object Management Group - OMG: *Meta Object Facility (MOF) Specification Version 1.4*. 2002a. URL: http://www.omg.org/cgi-bin/doc?formal/02-04-03.

[OMG '02b] Object Management Group - OMG: *MOF 2.0 query, views, transformations (QVT) request for proposals*. 2002b. URL:

http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Query_View_Transf._RFP.html.

[OMG '03] Object Management Group - OMG: *Common Warehouse Metamodel (CWM) Specication, v1.1.*, 2003. URL: http://www.omg.org/cgi-bin/doc?formal/03-03-02.

[OMG '05a] Object Management Group - OMG: *MOF 2.0/XMI Mapping Specification, v2.1 (formal/05-09-01)*. 2005a. URL: http://www.omg.org/docs/formal/05-09-01.pdf.

[OMG '05b] Object Management Group - OMG: *Unified Modeling Language: Infrastructure Version 2.0*. 2005b. URL: http://www.omg.org/cgi-bin/doc?formal/05-07-05.

[OMG '05c] Object Management Group - OMG: *Unified Modeling Language: Superstructure Version 2.0*. 2005c. URL: http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[OMG '06] Object Management Group - OMG: *Object Constraint Language Version 2.0*. 2006. URL: http://www.omg.org/cgi-bin/doc?formal/2006-05-01.

[OMG '07] Object Management Group - OMG: *MOF QVT Final Adopted Specification*. 2007. URL: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[OMG a] Object Management Group - OMG a: *Systems Modeling Language (Webseite)*. URL: http://www.omgsysml.org/.

[OMG b] Object Management Group - OMG b: *Model Driven Architecture (Webseite)*. URL: http://www.omg.org/mda/.

[OMG c] Object Management Group - OMG c: *Unified Modeling Language (Webseite)*. URL: http://www.uml.org/.

[OSEK/VDX '05] Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug - OSEK/VDX: *Operating System Specification 2.2.3*. 2005.

[Pollet et al. '02] Pollet, D.; Vojtisek, D.; Jézéquel, J.-M.: *OCL as a core UML transformation language*. *WITUML 2002 Position paper, Malaga, Spain*. 2002. URL: http://www.irisa.fr/triskell/publis/2002/Pollet02a.pdf.

[Pont '01] Pont, Michael J.: *Patterns for Time-Triggered Embedded Systems*. Addison Wesley, 2001.

[QVTMG '04] QVT-Merge Group - QVTMG: *Revised submission for MOF 2.0 Query/Views/Transformations, version 1.8*. 2004. URL: http://www.omg.org/docs/ad/04-10-04.pdf.

[Schäuffele et al. '03] Schäuffele, Jörg; Zurawka, Thomas: *Automotive Software Engineering*. Vieweg, 2003.

[Sendall et al. '03a] Sendall, S.; Kozaczynski, W.: *Model Transformation - The Heart and Soul of Model-Driven Software Development*. *IEEE Software*. Ausgabe 20. 2003a. S. 42-45. URL: http://cui.unige.ch/~sendall/files/sendall-tech-report-EPFL-model-trans.pdf.

[Sendall et al. '03b] Sendall, S.; Perrouin, G.; Guelfi, N.; Biberstein, O.: *Supporting Model-to-Model Transformations: The VMT Approach*. In (Rensink, A. Hrsg.) Ausgabe CTIT Technical Report. University of Twente, 2003b. S. 61-72. URL: http://trese.cs.utwente.nl/mdafa2003/proceedings.pdf.

[Stevens et al. '98] Stevens, Richard; Brook, Peter; Jackson, Ken; Arnold, Stuart: *Systems Engineering. Coping with Complexity*. Prentince Hall, 1998.

[The Mathworks '04] The Mathworks: *Simulink 6 - Simulation and model-based design*. 2004. URL: https://tagteamdbserver.mathworks.com/ttserverroot/Download/20594_SL_9320v05.pdf.

[W3C '05] World Wide Web Consortium - W3C: *XSL Transformations (XSLT) Version 2.0*. In (Kay, M. Hrsg.): *W3C Working Draft*. 2005. URL: http://www.w3.org/TR/xslt20/.

[Welch et al. '02] Welch, Lonnie R.; Marinucci, Toni: *Dynamic Resource Management Architecture Patterns*. *PLoP 2002 Proceedings*. 2002. URL: http://jerry.cs.uiuc.edu/~plop/plop2002/final/RM_patterns.pdf.

[Whittle '02] Whittle, J.: *Transformations and Software Modeling Languages: Automating Transformations in UML*. In (Jézéquel, J.-M., Hussmann, H., Cook, S. Hrsg.): *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002.* Ausgabe LNCS. Springer, 2002.

[Willink '03a] Willink, E. D.: *A concrete UML-based graphical transformation syntax:The UML to RDBMS example in UMLX*. *Eclipse Foundation Whitepaper*. 2003a. URL: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/doc/umlx/M4M03.pdf?rev=1.2.

[Willink '03b] Willink, E.D.: *UMLX: A graphical transformation language for MDA*. *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. Anaheim, California, USA, 2003b. URL: http://se2c.uni.lu/tiki/se2c-bib_download.php?id=799.

[Yacoub et al. '04] Yacoub, Sherif M.; Ammar, Hany H.: *Pattern-oriented analysis and design: composing patterns to design software systems*. Addison-Wesley, Boston [u.a.], 2004. S. XXIV, 385 S.

Thesen

1. Die logische Systemarchitektur der Systementwicklung lässt sich bei einem durchgängig modellbasierten Entwicklungsprozess im Rahmen der Softwareentwicklungsmethode ROPES als Analysemodell wiederverwenden.
2. Sowohl in solchen Analysemodellen als auch in den nachgelagerten Designmodellen lassen sich wiederverwendbare Muster finden.
3. Zwischen den Analyse- und Designmustern lassen sich wiederverwendbare Realisierungsbeziehungen finden.
4. Diese Beziehungen zwischen den Analyse- und Designmustern variieren mit den nichtfunktionalen Anforderungen und der technischen Systemarchitektur.
5. Werden für den Mustereinsatz formalisierte Templates mit Parametern verwendet, lassen sich die Beziehungen zwischen den Analyse- und Designmustern systematisch beschreiben. Kernelement ist hier die Zuweisung von Analysemusterparametern zu Designmusterparametern.
6. Auf Basis dieser Systematik lassen sich Modelltransformationen definieren, mit denen der halbautomatische Übergang zwischen Analyse- und Designmodellen möglich ist.
7. Durch Verlinkung der angelegten Designmusterinstanzen mit der Instanz des entsprechenden Analysemusters, lässt sich die Verfolgbarkeit von Designentscheidungen unterstützen.

Ulm, den 17. September 2007

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zur Folge hat.

Ulm, den 17. September 2007