Komponentenbasierte Software-Entwicklung: Adaption der KobrA-Methode für eingebettete Systeme

Stefan Josten

Komponentenbasierte Software-Entwicklung: Adaption der KobrA-Methode für eingebettete Systeme

Diplomarbeit

von

Stefan Josten

Juli 2001

Fraunhofer Einrichtung für
Experimentelles Software Engineering
IESE Siegelbach

Betreuer: Prof. Dr. H. Dieter Rombach
Dr. Christian Bunse

Erklärung

Hiermit erkläre ich, Stefan Josten, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kaiserslautern, den Datum

Zusammenfassung

Eingebettete Systeme erschließen sich durch ihre wachsende Leistungsfähigkeit immer neue Einsatzgebiete. Die steigende Verbreitung und insbesondere die erhöhten Anforderungen an Sicherheit und Zuverlässigkeit lassen die Notwendigkeit nach einer systematischen Vorgehensweise bei ihrer Entwicklung erkennen.

Die vom Fraunhofer IESE mitentwickelte KobrA-Methode bietet einen systematischen Prozess und legt einen Schwerpunkt auf moderne Wiederverwendungstechnologien wie Komponenten und Produktlinien. Im Gegensatz zu anderen Methoden spielen Komponenten bei KobrA nicht nur während der Implementierungsphase eine Rolle, sondern bilden einen zentralen Bestandteil der Methode während allen Phasen der Software-Entwicklung. Bisher wurde KobrA noch nicht im Kontext von eingebetteten Systemen erprobt.

Aufgabenstellung dieser Diplomarbeit ist die Untersuchung, in wie fern KobrA für die Entwicklung eingebetteter Systeme geeignet ist. Dazu findet eine Betrachtung der Besonderheiten eingebetteter Systeme statt und welche Probleme bei ihrer Entwicklung mit KobrA auftreten können. Die dabei entwickelten Lösungsansätze werden anschließend evaluiert. Dazu wird aus den LEGO Mindstorms Baukästen ein Roboter entwickelt, der als Beispielsystem dient. Der Produktlinienansatz von KobrA wird durch zwei ähnliche Aufgabenstellungen berücksichtigt, die der Roboter erfüllen soll.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Motivation

Eingebettete Systeme erschließen sich durch ihre wachsende Leistungsfähigkeit immer neue Einsatzgebiete und nehmen dadurch an Bedeutung zu. Mengenmäßig übertreffen sie die Zahl herkömmlicher Computer bei weitem. So beträgt die Zahl der eingebetteten Systeme allein in den Küchengeräten eines Durchschnittshaushaltes ein Vielfaches der Anzahl der vorhandenen PCs. Diese wachsende Verbreitung und insbesondere die erhöhten Anforderungen an Sicherheit und Zuverlässigkeit lassen die Notwendigkeit nach einer systematischen Vorgehensweise bei der Entwicklung von Software für eingebettete Systeme erkennen. Es gibt zwar viele Methoden zur systematischen Softwareentwicklung, aber nur wenige, die speziell für eingebettete Systeme entwickelt wurden.

Die vom Fraunhofer IESE mitentwickelte KobrA-Methode zeichnet sich durch einen systematischen Prozess aus und setzt einen Schwerpunkt auf moderne Wiederverwendungstechnologien wie Komponenten und Produktlinien. Viele aktuelle Methoden gehen bei der Unterstützung der Komponententechnologie nicht weit genug. So werden Komponenten eher als Ergebnis und nicht als zentraler Teil der Software-Entwicklung angesehen. An dieser Stelle setzt die KobrA-Methode an. Sie zeichnet sich durch eine durchgängige Unterstützung von Komponenten während aller Phasen der Entwicklung aus. Daneben unterstützt sie Produktlinien, UML und verfügt über einen systematischen Prozess. Allerdings wurde sie bisher noch nicht im Kontext von eingebetteten Systemen erprobt.

Aufgabenstellung dieser Diplomarbeit ist die Untersuchung, ob KobrA für die Entwicklung eingebetteter Systeme geeignet ist. Da eingebettete Systeme einige spezielle Eigenschaften aufweisen, wird erwartet, dass KobrA eventuell modifiziert werden muss. Dazu findet im ersten Teil eine theoretische Betrachtung statt, welche Probleme auftreten können und wie diese gelöst werden können. Im zweiten Teil der Diplomarbeit findet eine Evaluierung der identifizierten Probleme und der entwickelten Lösungsvorschläge statt. Dazu steht als Beispielsystem ein aus den LEGO Mindstorms Baukästen entwickelter Roboter zur Verfügung.

Der LEGO-Baukasten wird verwendet, weil damit relativ einfach ein eingebettetes System wie beispielweise ein Roboter entworfen werden kann. Außerdem existieren Umsetzungen vieler moderner Sprachen wie Java für das LEGO Mindstorms System. Für die Entwicklung wird auch kein teures Entwicklungssystem benötigt. Die Nachteile wie beschränkte Rechenleistung und geringe Zahl an Ein-

bzw. Ausgängen werden durch die Kombination von zwei LEGO Systemen, die zusammenarbeiten, behoben.

Lösungsansatz

Zuerst wird in Kapitel 2 eine Auswahl von Eigenschaften vorgestellt, die typisch für eingebettete Systeme sind. Außerdem werden einige grundlegende Begriffe eingeführt und erläutert. Danach werden Ansätze zur ingenieurmäßigen Softwareentwicklung wie Unified Process, OMT und KobrA vorgestellt und hinsichtlich ihrer Eignung für eingebettete Systeme untersucht und bewertet.

Da KOBRA eine relativ neue Methode ist und bisher nur an einer Fallstudie über ein Bibliothekssystem getestet wurde, wird erwartet, dass Anpassungen an die Besonderheiten eingebetteten Systeme notwendig sein werden. Dies ist Thema von Kapitel 3. Hier werden die Aktivitäten und Artefakte der KobrA-Methode auf notwendige Änderungen untersucht, Probleme dokumentiert und wenn nötig KobrA entsprechend modifiziert.

Die Aufgabenstellung der Diplomarbeit beinhaltet die Evaluierung der Ergebnisse aus Kapitel 3 an einem Beispielsystem. Dazu stellt Kapitel 4 den Roboter-Baukasten von LEGO, das *Robotics Invention System*, vor. Es wird auf den Lieferumfang, die Leistungsfähigkeit, die Funktionsweise und die verfügbaren Programmiersprachen eingegangen. Zwei Programmiersprachen, die JAVA-Varianten *TinyVM* und *leJOS*, werden ausführlicher erläutert, da sie später für die Programmierung des Roboters benutzt werden. Um den Produktlinienansatz von KobrA zu untersuchen, werden zwei Aufgabenstellungen entwickelt, die in wesentlichen Teilen übereinstimmen. Dazu wird ein Roboter konstruiert, der diese Aufgabenstellungen erfüllen soll. Teil der Aufgabenstellung ist auch eine Beschreibung der späteren Einsatzumgebung des Roboters. Um das Verständnis der nachfolgenden Kapitel zu erleichtern, wird ein grober Überblick gegeben, wie die Software des Roboters realisiert wurde. Eine detaillierte Beschreibung dazu befindet sich in Anhang A.

In Kapitel 5 werden die in Kapitel 3 identifizierten Probleme und Lösungsvorschläge an einem Beispiel evaluiert. Als Beispiel dient der im vorangegangenen Kapitel vorgestellte LEGO-Roboter. Es wird ein kompletter KobrA-Prozess beschrieben. Dabei werden in Kapitel 5 nur die Artefakte und Aktivitäten aufgeführt, die von Modifikationen an der KobrA-Methode betroffen sind. Alle anderen, während der Entwicklung erzeugten Artefakte, befinden sich als Teil der Systemdokumentation in Anhang A. Kapitel 6 fasst zum Abschluss die gewonnenen Ergebnisse zusammen und listet offen gebliebene Probleme auf .

Anhang A enthält alle Artefakte, die während der Entwicklung der Roboter-Software erstellt werden. Sie entstehen durch eine komplette Durchführung des KobrA-Prozesses für die Aufgabenstellung aus Kapitel 4. Dabei werden zwei Komponenten identifiziert. Anhang A enthält ein komplettes KobrA Dokument.

In Anhang B befindet sich eine Bauanleitung des entwickelten Roboters. Es werden die wesentlichen Baugruppen und ihre Funktionsweise vorgestellt, inklusive der Sensoren bzw. Aktoren und ihrer Anschlüsse. Außerdem wird ein Experiment zur Leistungsfähigkeit der Sensorik und der Aufbau der Zielobjekte beschrieben.

Der komplette Source Code des Roboters befindet sich in Anhang C. Da für die Konstruktion des Roboters zwei RCX-Bausteine verwendet werden, ist Anhang C in zwei Abschnitte für den jeweiligen Source Code unterteilt.

Kapitel 2

Methoden zur Software - Entwicklung für eingebettete Systeme

2.1 Besonderheiten eingebetteter Systeme

Eingebettete Systeme (*Embedded Devices*) arbeiten in der Regel im Verborgenen. Ihr Einsatzspektrum reicht von der Maschinensteuerung über Haushalts- und Unterhaltungsgeräte, bis zu mobilen Systemen wie Handys oder GPS-Empfängern. Ein eingebettetes System enthält häufig einen kompletten Minirechner, der Teil eines größeren Gerätes ist. Ein Beispiel dafür ist das ABS bei Autos, das aus einer Vielzahl von Sensoren und Aktoren besteht, die von einem solchen Minirechner ausgewertet bzw. geregelt werden. Es wird eine über die eines normalen PCs hinausgehende Funktionalität zur Verfügung gestellt [DOU98]. Eingebettete Systeme werden u. a. zur Steuerung von (analogen) Geräten eingesetzt und sind daher häufig mit integrierten Schnittstellen und A/D-, D/A-Wandlern ausgestattet. [DOU98] [OSW00]

Im Gegensatz zu handelsüblichen PCs, bei denen für eine neue Aufgabenstellung nur die Software neuentwickelt wird, werden bei eingebetteten Systemen häufig Hard- und Software an die Problemstellung angepasst. Dieses Hardware-Software-Codesign kommt u. a. dann zum Einsatz, wenn spezielle Anforderungen zu erfüllen sind, wie beispielsweise die sehr schnelle Reaktionszeit eines Airbags. Daneben gibt es aber auch universelle Standardkomponenten wie z. B. Microcontroller, die miteinander kombinierbar sind und die für verschiedene Anwendungen eingesetzt werden können. Angepasst wird in solchen Fällen vor allem die Software. So lohnt es sich zum Beispiel nicht, für die Steuerung einer Waschmaschine eine komplett neue Hardware zu entwickeln, wenn die gleiche Aufgabe durch billige Standardkomponenten genauso gut erledigt werden kann.

Obwohl sich eingebettete Systeme bezüglich ihrer Größe, Form, Funktion und Technik erheblich unterscheiden, haben sie einige charakteristische Merkmale.

Typische Merkmale eingebetteter Systeme [OSW00], [DOU98], [GOM00], [PUT99]

- **Zuverlässigkeit:** Eingebettete Systeme werden oft in sicherheitskritischen Bereichen eingesetzt. Im Gegensatz zu einem PC, der mehrmals am Tag neugestartet werden kann, wenn ein Fehler

auftritt, muss ein Airbag über Jahre zuverlässig funktionieren. Ein Zurücksetzen ist nicht möglich und Rückrufaktionen zur Fehlerbehebung sind teuer.

- **Robustheit:** Viele eingebettete Systeme werden in ganz anderen Umgebungen eingesetzt als PCs und müssen daher bezüglich Umgebungscharakteristika wie Temperatur, Strahlung, Erschütterungen, Druck oder Feuchtigkeit unempfindlich sein. Erschütterungsempfindliche Teile wie Festplatten sind für viele Einsatzgebiete nur bedingt geeignet.
- **Echtzeitfähigkeit:** Echtzeitfähigkeit spielt bei sogenannten Echtzeit-Systemen eine zentrale Rolle. Beispielsweise arbeiten Airbag oder ABS im Millisekunden-Bereich. Es ist absolut notwendig, dass eine bestimmte Reaktionszeit garantiert wird. Verantwortlich dafür ist sowohl das Betriebssystem als auch das Anwendungsprogramm.
- **Fehlerhafte Daten:** Eingebettete Systeme werden oft mit fehlerhaften oder unklaren Daten konfrontiert. Besonders der Einsatz von Sensoren, z. B. in einem autonomen mobilen Roboter, ist fehlerträchtig. So können beispielsweise Signale mit zu kurzen Impulsen übersehen oder Tastsensoren durch Erschütterungen ausgelöst werden. Ebenso kann es zu Fehlmessungen kommen, wenn ein Sensor bei starken Schwankungen der zu messenden Größe kurzzeitig falsche Daten liefert. Trotzdem muss das eingebettete System in der Lage sein, seine Aufgabe korrekt zu erfüllen.
- **Aktoren und Sensoren:** Eingebettete Systeme können mit Aktoren und Sensoren ausgestattet sein, über die sie mit ihrer Umwelt in Kontakt treten. Ein Heizungssteuerungssystem misst beispielsweise mit seinen Sensoren die Temperatur und verstellt über Aktoren die Regler an den Heizkörpern. Dabei muss berücksichtigt werden, dass die Umgebung sich nicht immer so verhält wie geplant und es zu unerwarteten Ereignissen kommen kann. So kann es beispielsweise durch das Öffnen eines Fensters in einem Raum kälter werden, obwohl die Heizung mit voller Leistung heizt.
- **Reaktivität:** Ein eingebettetes System muss in der Lage sein, auf Ereignisse, die in nicht vorhersagbarer Reihenfolge und zu unterschiedlichen Zeitpunkten eintreten, zu reagieren. Beispielsweise müssen bei Eintreten eines Unfalls die Airbags auslösen. Dass die Sensoren in der Motorsteuerung gleichzeitig merkwürdige Werte liefern, kann in dieser Situation ignoriert werden.
- **Leistungsfähigkeit:** Typisch für viele eingebettete Systeme sind ihre geringen Abmessungen. Es steht in der Regel auch kein leistungsfähiges Netzteil zur Verfügung. Um Platzbedarf und Stromaufnahme niedrig zu halten, wird mit möglichst geringen Taktzahlen und Speichergrößen gearbeitet. Dieses Prinzip der Minimalität trifft auch auf die Kosten zu. Warum sollen leistungsfähigere und damit teurere Bauteile verwendet werden als unbedingt nötig? Dem muss durch effiziente Programmierung Rechnung getragen werden. Besondere Vorsicht ist bei der Verwendung von Arrays oder Rekursion geboten, da sie schnell den kompletten Speicher belegen können.
- **Nebenläufigkeit:** Zum einen verhält sich die Umgebung oft nebenläufig. Ereignisse wie z. B. ein Unfall oder Motorüberhitzung beim Auto können gleichzeitig und unabhängig voneinander auf-

treten. Zum anderen kann das eingebettete System nebenläufig ausgelegt sein, d. h., dass die aktiven Prozesse nebenläufig sind und deshalb unabhängig voneinander ausgeführt werden können.

- **Verteilung:** Für viele Anwendungen ist Verteilung vorteilhaft. Beispielsweise existiert beim Auto kein Zentralrechner, der alle Aufgaben erledigt. Stattdessen sind viele Sensoren über das ganze Auto verteilt. Sie melden die gemessenen Werte an eingebettete Systeme, die z. B. den Airbag auslösen, die Einspritzung steuern oder das Kraftstoffgemisch ändern, weiter. Die Verbindung erfolgt nicht durch eigene Leitungen sondern über einen Bus. Vorteil einer solchen Verteilung ist die Robustheit des Gesamtsystems. Wenn eine Komponente ausfällt, funktioniert der Rest weiter. Besonders kritische Komponenten können auch doppelt ausgelegt werden (Redundanz). Die Verteilung ermöglicht auch eine schnelle Reaktion. Wenn ein Untersystem nur für die Airbagauslösung zuständig ist, kann seine Reaktionszeit besser garantiert werden als bei einem Zentralrechner, der zusätzlich andere Aufgaben erledigen muss.

Die aufgelisteten Merkmale sind keine notwendigen Bedingungen für eingebettete Systeme, sondern variieren je nach Einsatzgebiet. Auch ein PC kann beispielsweise als eingebettetes System angesehen werden, z. B. in Verbindung mit einem Computertomographen [DOU98]. Den grundlegenden Aufbau eines eingebetteten Systems zeigt Abb. 2-1.

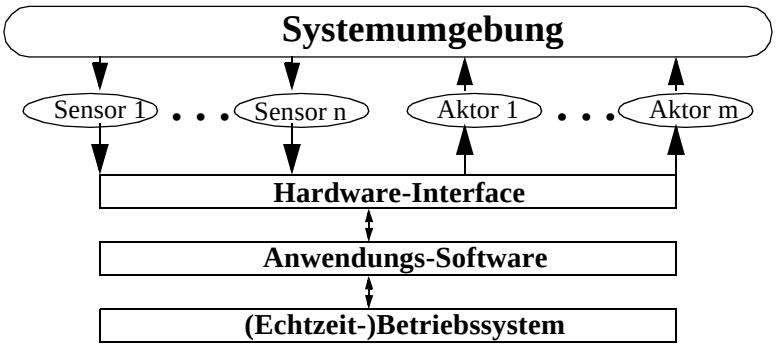


Abb. 2-1: Grundsätzliche Architektur eines eingebetteten Systems

Echtzeit-Systeme

Eine häufig benötigte Eigenschaft für eingebettete Systeme ist die Echtzeitfähigkeit. Dabei werden Aussagen bezüglich der Leistungsfähigkeit getroffen. Die kritische Größe ist die Zeit, die verstreicht, bis auf ein Ereignis reagiert wird. Die Größe der gewünschten Reaktionszeit hängt stark von der jeweiligen Anwendung ab. Bei Echtzeitsystemen wird zwischen harter und weicher Echtzeit unterschieden [REC97].

Bei harten Echtzeit-Anwendungen muss eine maximale Reaktionszeit unabhängig von der Systemlast garantiert werden. Die Dauer hängt von der Art der Anwendung ab und reicht z. B. vom Mikrosekunden- (Airbag) bis in den Sekundenbereich (Spülmaschine). Es darf kein relevantes Ereignis verpasst werden, da zu spät gelesene Daten ungültig sind. Reaktionen müssen immer in der geforderten Reaktionszeit erfolgen. Die durchschnittliche Reaktionszeit spielt keine Rolle. Ein Airbag muss z. B. immer und nicht nur in 99% aller Fälle rechtzeitig auslösen.

Anwendungen, bei denen die Reaktionszeit nicht kritisch ist, kommen mit sogenannter weicher Echtzeit aus. Dies bedeutet, dass ein Ereignis auch einmal übersehen werden oder die Reaktion verzögert erfolgen kann. Ältere Daten sind noch gültig oder lösen im Fehlerfall zumindest keine Katastrophe aus. Es wird mit durchschnittlichen Reaktionszeiten gearbeitet.

Ein aus einem einzigen Task bestehendes System erfordert keine besonderen Maßnahmen, um das Echtzeitverhalten zu realisieren. Wenn der Prozessor nicht in der Lage ist innerhalb des gewünschten Zeitraumes zu reagieren, kann er durch einen schnelleren ersetzt werden. Oft laufen auf einem System aber mehrere Tasks quasi-parallel. Die Rechenzeit der CPU wird durch einen *Scheduler* auf die Tasks aufgeteilt. Eine einfache Scheduling-Technik ist das *preemptive scheduling* (Abb. 2-2), auch als Zeitscheibenverfahren (*round robin*), bekannt. Hierbei wird jedem Task ein Zeitfenster zugewiesen, in dem er ausgeführt wird. Nach Ablauf der Zeitspanne wird er angehalten und zum nächsten Task in der Tabelle weitergeschaltet. Diese Verfahren ist zwar fair, d. h. jeder Task wird gleichberechtigt abgearbeitet, kann aber bei vielen gleichzeitig aktiven Tasks die Reaktionszeit nicht mehr garantieren.

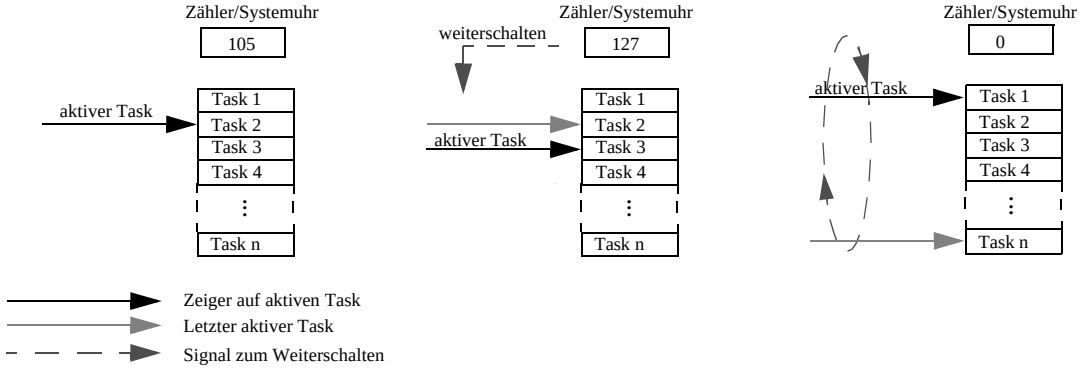


Abb. 2-2: *preemptive scheduling*

Da nicht alle dieselbe Wichtigkeit haben - beispielsweise ist der Task „Kollision_vermeiden“ bei einem Roboter wichtiger als ein Task „Suche_Licht“ - kann es sinnvoll sein, die Tasks nach ihrer Wichtigkeit zu ordnen. Ein Beispiel dafür ist das *priority based preemptive scheduling* (Abb. 2-3). Dieses Verfahren ermöglicht es Prioritäten zuzuweisen und damit einen Task durch einen anderen mit höherer Priorität zu unterbrechen. Bei seinem Einsatz muss beachtet werden, dass es zum Verhungern von Tasks kommen kann. Für Tasks mit hoher Priorität kann eine feste Reaktionszeit unter der Bedingung garantiert werden, dass nicht zu viele dieselbe hohe Priorität haben. [DUM00], [REC97]

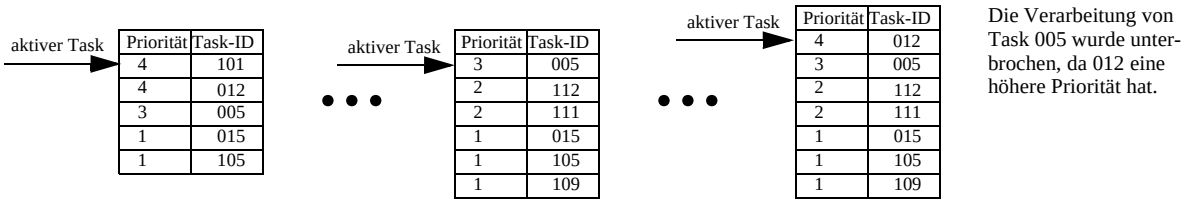


Abb. 2-3: *priority based preemptive scheduling* mit sortierter Warteschlange

Die objektorientierte Programmierung hat einige neue Probleme zur Folge. Bei klassischen Sprachen wie C oder PASCAL läuft die Abarbeitung sequentiell ab. So lange keine Tasks verwendet werden,

sind keine besonderen Maßnahmen bezüglich Synchronisation, Nebenläufigkeit oder Reihenfolge der Ausführung nötig. Ebenso ist es relativ leicht, Zeitabschätzungen vorzunehmen. Auf objektorientierte Sprachen trifft dies nicht zu. Hier kann nicht immer genau gesagt werden, in welcher Reihenfolge oder wann etwas ausgeführt wird. Besonders schwierig wird dies, wenn mit Ereignissen und Nachrichten gearbeitet wird. Typische Fragestellungen sind z. B. „In welcher Reihenfolge treffen die Nachrichten ein?“ oder „Wie und wann werden die unterbrochenen Tätigkeiten fortgesetzt?“.

Es existieren viele unterschiedliche Methoden zur systematischen Software-Entwicklung. Es gibt zwar Methoden, die speziell für eingebettete Systeme entwickelte wurden, auf die meisten trifft dies jedoch nicht zu. Im folgenden Abschnitt werden verschiedene Methoden vorgestellt und bezüglich ihrer Eignung für eingebettete Systeme untersucht und bewertet.

2.2 Methoden und Techniken

Es existiert eine große Zahl unterschiedlicher Methoden zur Softwareentwicklung. Sie unterscheiden sich teilweise stark in ihrem Umfang. Einige stellen nur Techniken für einzelne Phasen der Entwicklung zur Verfügung, andere versuchen den kompletten Entwicklungsprozess abzudecken. Auch bezüglich der formalen Strenge bestehen Unterschiede. So geben manche Methoden nur einen groben Rahmen in Form von Empfehlungen vor, den die einzelne Softwareorganisation gemäß ihren Bedürfnissen ausgestalten soll. Andere Methoden nehmen den Anwender bei der Hand, indem sie strenge Vorgaben machen und ihm damit wenig Handlungsspielraum lassen.

Die hier vorgestellten Methoden stellen nur eine kleine Auswahl dar. Sie werden kurz mit ihren Vor- und Nachteilen vorgestellt und hinsichtlich ihrer Eignung für den Entwurf eingebetteter Systeme untersucht.

2.2.1 OMT

Die **O**bject **M**odeling **T**echnique (OMT) wurde von James Rumbaugh et. al. [RUM91] am *General Electric Research and Development Center* entwickelt. Zu ihrer Blütezeit war sie eine der weitverbreitetsten Methoden in der objektorientierten Softwareentwicklung. Viele der Ideen von OMT wurden von nachfolgenden Methoden und Modellierungssprachen übernommen und weiterentwickelt. So ist die von Rumbaugh eingeführte Notation beispielsweise ein Vorläufer der UML [UML].

Der Entwicklungsprozess zeigt Ähnlichkeiten zum Wasserfallmodell. Er wird in die sequentiellen Phasen **Problembeschreibung**, **Analyse**, **Systementwurf**, **Objektentwurf**, **Implementierung** und **Test** zerlegt.

Ausgehend von der Problembeschreibung findet in der Analysephase die Abstraktion des zu lösenden Problems der realen Welt in die Form von Objekten und ihren Beziehungen untereinander statt. Es wird beschrieben, was das System tun soll und nicht wie es dies bewerkstelligt. Beim anschließenden Systementwurf wird die Architektur des Zielsystems auf einem hohen Abstraktionslevel konstruiert. Eventuell wird das System dabei in Subsysteme zerlegt. Der Objektentwurf realisiert eine weitere Verfeinerung. Dazu werden die identifizierten Klassen mit den benötigten Datenstrukturen und Metho-

den ausgestaltet. Die Übersetzung in eine Programmiersprache findet während der Implementierung statt.

OMT zeichnet sich durch eine ausdrucksstarke Notation aus. So werden in der Analysephase 3 verschiedene Modelle erstellt:

- Das **Object Model** verwendet ein Klassendiagramm. Dabei werden die für den Benutzer sichtbaren Objekte identifiziert. Ihre Beziehungen zueinander werden durch die Struktur modelliert.
- Das **Dynamic Model** benutzt Statecharts und Sequenz-Diagramme. Mit Hilfe von Zuständen, Ereignissen und Interaktionen wird das dynamische Verhalten des Systems modelliert.
- Beim **Functional Model** wird eine verbesserte Form von Data-Flow-Diagrammen benutzt, welche die Berechnungen und Datenmanipulationen des Systems zeigen.

Der Schwerpunkt dieser Methode liegt in der Analysephase, die ausführlich beschrieben ist. Schwächen treten bei den nachfolgenden Phasen auf, die nur oberflächlich erläutert werden [OES01]. Beispielsweise schweigt sich OMT darüber aus, wie das Subsystemdesign durchzuführen ist [DER95].

Auf Grund ihres Alters unterstützt OMT aktuelle Themen wie inkrementelle Entwicklung, Komponententechnologien, Design Patterns und Wiederverwendung nicht [ATK01].

2.2.2 Unified Process / Rational Unified Process

Grady Booch, James Rumbaugh und Ivar Jacobson sind drei prominente Autoren objektorientierter Vorgehensmodelle. Anstatt alleine weiter zu forschen, arbeiteten diese seit 1995 bei der Firma Rational daran, ihre Ansätze BOOCH [BOO94], OMT [RUM91] und OOSE [JAC94] zu einem Prozess zu vereinigen. Als ersten Schritt einigten sie sich auf gemeinsame Notation und legten damit den Grundstein zur **U**nified **M**odeling **L**anguage [UML], die sich zu einem Quasistandard für objektorientierte Modellierung entwickelte. UML allein enthält keine Aussagen über Vorgehensweisen, da sie eine reine Modellierungssprache und kein Prozess ist. Mit Hilfe einer Sammlung verschiedener Diagramme und Techniken soll sie alle Phasen der Softwareentwicklung unterstützen. Wie der konkrete Einsatz von UML aussieht, bleibt der jeweiligen Softwareorganisation überlassen. Die Weiterentwicklung und Standardisierung von UML wird von der **O**bject **M**anagement **G**roup [OMG] koordiniert. [FOW98]

1999 veröffentlichten die drei den *Unified Software Development Process*, der auch unter dem Namen *Unified Process* bekannt ist [JAC99]. Es war nicht das Ziel, einen sofort ohne Anpassungen und Änderungen einsetzbaren Prozess zu schaffen. Daher stellt der *Unified Process* nur einen Rahmenprozess für unterschiedliche Arten von Softwaresystemen, Anwendungsgebieten oder Organisationen dar. Die Firma Rational [RAT] bietet eine konkretere, fertig ausgestaltete Variante des *Unified Process* unter dem Namen **R**ational **U**nified **P**rocess (RUP) [KRU01] an und verkauft dazu entsprechende Werkzeuge. Der folgende Text bezieht sich auf den Unified Prozess, beschrieben unter [JAC99].

Zur Prozessbeschreibung werden Phasen und Iterationen, Prozesse (*Workflows*) und Aktivitäten, Rollen (*Worker*), Modelle und Produkte (*Artifacts*) unterschieden (vgl. Abb. 2-4).

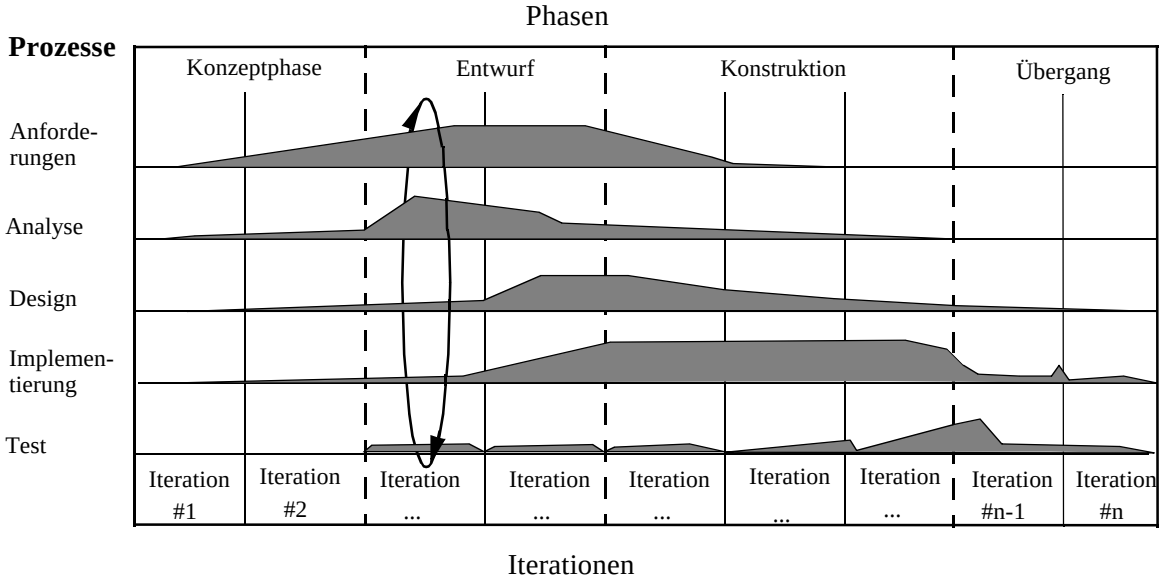


Abb. 2-4: Unified Process: Iteration, Prozesse und Phasen [OES01]

Phasen und Iterationen

Während die Managementsicht einem Phasenmodell folgt und dabei Konzept-, Entwurfs-, Konstruktions- und Übergangsphase unterscheidet, erfolgt die technische Realisierung in Iterationen. Am Ende jeder Phase steht ein größerer Meilenstein. Diese können als erste Anhaltspunkte für den Projektfortschritt dienen. Innerhalb der Phasen können je nach Komplexität kleinere Meilensteine entstehen, die den Iterationen entsprechen.

Der Unterschied zwischen kleinen und großen Meilensteine liegt auf der Managementebene. Bei kleineren diskutieren Management und Entwickler, wie mit der nächsten Iteration fortgefahren wird. Bei größeren Meilensteinen am Ende einer Phase werden strenge „Fertig“ oder „nicht Fertig“ Entscheidungen getroffen. Des Weiteren werden Fahrpläne, Budget und Anforderungen aufgestellt.

Der Entwicklungsprozess besteht aus den folgenden vier Phasen:

- **Konzeptphase (Inception)**
  In dieser Phase findet die Definition der Planungshorizonte statt. Das Produkt wird definiert und Überlegungen über die Machbarkeit und Wirtschaftlichkeit werden angestellt. Außerdem werden die größten Risiken identifiziert und Vorgaben zu ihrer Vermeidung aufgestellt. Eventuell werden bereits erste Schlüsselklassen gefunden. Diese Phase endet mit der Entscheidung über die Durchführung des Projektes.
- **Entwurfsphase (Elaboration)**
  Diese Phase umfasst die Anforderungsanalyse. Darauf aufbauend erfolgt iterativ die Konstruktion einer validierten, stabilen und ausführbaren Softwarearchitektur. Am Ende des Prozesses sollen die Kosten des Projektes abgeschätzt werden können und die nächste Phase planbar sein.
- **Konstruktionsphase (Construction)**
  Dies ist die aufwändigste Phase. In ihr wird der Funktionsumfang des Produktes iterativ weiterentwickelt, bis am Ende ein auslieferbares Produkt entstanden ist. Je nach Projektumfang können

viele Iterationen nötig sein. Die Planung der Inkremente kann nach Riskogesichtspunkten erfolgen. In einer riskanten Projektphase kann es z. B. sinnvoll sein, kleinere Iterationsschritte zu vollziehen und damit mehr Inkremente zu erzeugen als in einer weniger riskanten Phase.

- **Übergangsphase (Transition)**
  In dieser Phase findet die Übergabe der Software an den Anwender statt. Dazu müssen u. a. Betatests durchgeführt, die Handbücher geschrieben und gedruckt, CDs vervielfältigt und Anwender geschult werden. Am Ende dieser Phase ist das Produkt ausgeliefert und das Projekt beendet.

Die Phasen und die Inkremente verdeutlichen das Fortschreiten des Projektes in horizontaler Richtung (Abb. 2-4). Um von einem Inkrement zum nächsten zu gelangen, müssen verschiedene Schritte in vertikaler Richtung durchlaufen werden. Diese werden als Prozesse bezeichnet.

Prozesse, Aktivitäten und Produkte

Den vier Phasen werden die Hauptprozesse **Anforderungen**, **Analyse**, **Design**, **Implementierung** und **Test** gegenübergestellt. Wegen der Namensgleichheit mit den Phasen des Wasserfallmodells kann es allerdings zu Verwechslungen kommen. Der Unterschied ist, dass Anforderungen, Analyse, Design, Implementierung und Test beim Wasserfallmodell Phasen bezeichnen und während des Projekts nur einmal durchlaufen werden. Beim *Unified Process* sind sie Prozesse und werden pro Iteration einmal ausgeführt.

- Der Prozess **Anforderungen** umfasst alle Entwicklungsaktivitäten, die zur Aufnahme, Überprüfung und Dokumentation von Anforderungen durchgeführt werden. Da dies aus der Sicht des Anwenders geschieht, wird hier eine externe Sicht der Anforderungen beschrieben.
- Bei der **Analyse** wird die interne Sicht der Anforderungen formal beschrieben. Redundanzen und Inkonsistenzen werden entfernt. Im Gegensatz zum vorherigen Prozess werden die Anforderungen nicht in der Sprache des Anwenders, sondern in der Sprache der Informatik formuliert.
- Im Prozess **Design** wird aus den Anforderungen ein Lösungskonzept entwickelt. Es wird spezifiziert, wie die Anforderungen umgesetzt werden sollen.
- Die **Implementierung** umfasst alle Aktivitäten, die den Code und ablauffähige Produkte erzeugen.
- Mit Hilfe des Prozesses **Test** wird der Nachweis erbracht, dass das System die Anforderungen erfüllt.

Die Prozesse können wiederum aus mehreren Aktivitäten bestehen, die allerdings nur grob beschrieben sind. An dieser Stelle wird deutlich, dass der Unified Process nur einen idealisierten Rahmen vorgibt und in der Praxis noch ausgestaltet und verfeinert werden muss.

Die Prozesse erzeugen jeweils Modell-Artefakte (Produkte), die mit jeder Iteration weiterentwickelt werden. Der Aufwand für die Erstellung der Modelle variiert von Phase zu Phase. Beispielsweise sind nach der Entwurfsphase bereits 80% aller Use-Cases erzeugt.

Die wichtigsten Modelle sind:

- Das **Use-Case-Modell** repräsentiert die Umgebung und die gewünschte Funktionalität des Systems. Es soll die Kommunikation zwischen Anwendern und Entwicklern unterstützen und ist

Ausgangspunkt aller anderen Modelle. Bestandteile sind Aktoren, Anwendungsfälle (Use-Cases) und ihre Beziehungen untereinander.

- Das **Analyse-Modell** gibt einen Überblick über die erwartete Systemfunktionalität, indem die Anforderungen aus den Use-Cases näher spezifiziert werden. Es besteht aus Klassen- und Interaktions-Diagrammen.
- Beim **Design-Modell** wird das Lösungskonzept beschrieben. Es besteht aus Design-Klassen, Untersystemen, Paketen und den Beziehungen zwischen ihnen.
- Das **Implementierungs-Modell** besteht vor allem aus dem Quellcode des Softwaresystems.
- Das **Test-Modell** erweitert die Use-Cases um Testfälle.

Der Unified Process ist anwendungsfallgetrieben (durch die Use Cases), architekturzentriert (z. B. objektorientiert) und iterativ-inkrementell. Er liefert einen groben idealisierten Rahmen, der für die Praxis noch erheblich konkretisiert werden muss, da viele Aktivitäten noch zu unspezifisch beschrieben sind. Beispielsweise enthält der Prozess Implementierung die Aktivität „Eine Klasse implementieren“. Wie dieses zu geschehen hat, wird nicht näher spezifiziert [OES01].

Es war aber gar nicht beabsichtigt einen sofort einsetzbaren Prozess zu schaffen. Grady Booch unterstellte den Amateursoftware-Entwicklern „sie seien ständig auf der Suche nach dem Silver Bullet, der magischen Methode schlechthin, die den Entwicklungsprozess genau definiert und fast zwangsläufig einen guten Entwurf und ein funktionstüchtiges Programm liefert. Profis hingegen unterscheiden sich hiervon durch die erleuchtende Erkenntnis, dass es eine solche Methode einfach nicht gibt“ [ZER99]. *Rational* behauptet, der von ihnen vertriebene *Rational Unified Process* sei beides, sowohl ausgestaltbar und anpassbar als auch für Projekte mittlerer Größe sofort einsetzbar [KRU01].

Der *Unified Process* ist eine vielfältig einsetzbare Methode, die aber auch komplex und verwirrend ist und eine gewisse Einarbeitungszeit erfordert. Es werden fast alle Bereiche des modernen Software Engineerings abgedeckt, wie z. B. Use Cases, Iterationen, inkrementelle Entwicklung und Patterns. Komponenten werden unterstützt, allerdings nur auf der Ebene der ausführbaren Module. Um ihr volles Potential zu nutzen, müsste der *Unified Process* Komponenten in allen Phasen der Softwareentwicklung unterstützen und nicht erst kurz vor Fertigstellung [ATK01].

2.2.3 Catalysis

Catalysis: *„An acceleration of the rate of a process or reaction, brought about a catalyst, usually present in small managed quantities and unaffected at the end of the reaction. A catalyst permits reactions or processes to take place more effectively or under milder conditions than would otherwise be possible.“* [CAT]

Im Kontext von Systemmodellierung und Softwareentwicklung beschreibt Catalysis [SOU99] einen Ansatz, um den Entwicklungsprozess wiederholbar, skalierbar und effektiv zu machen. Entwickelt wurde er 1991, ursprünglich als Formalisierung von OMT gedacht, durch Desmond D’Souza und Alan Wills. Seitdem wurde diese Methode über viele Jahre weiterentwickelt und erweitert heute einige Entwicklungsmethoden der zweiten Generation wie z. B. *Fusion*. Außerdem umfasst Catalysis eine Form

der frameworkbasierten Entwicklung, eine Definition methodischer Verfeinerungen von der abstrakten Spezifikation bis zur Implementierung, den Umgang mit groben und feinen Komponenten, eine Use-Case-Formalisierung, die rekursive Dekomposition von Komponenten in Muster zusammenarbeitender Objekte, die Synthetisierung dieser Muster und die Verfeinerung der Transaktionen zwischen diesen Objekten, um die Verfolgbarkeit zu unterstützen. Daneben war Catalysis eine der ersten UML-basierten Methoden. [CAT], [SOUb]

Catalysis beansprucht für sich, das Komponentenkonzept auf den gesamten Entwicklungsprozess auszudehnen. Außerdem werden mit Ausnahme von *Product Line Engineering* moderne Wiederverwendungstechnologien wie *Architectural Styles, Design Patterns* und *Frameworks* unterstützt. Zum Einsatz von Frameworks gibt es einen Artikel [SOUa] von D'Souza und Wills. [ATK01]

Ein Beispiel für die Verwendung von Design Patterns ist der in Catalysis verwendete Prozess. Da es keinen universellen Prozess gibt, der für alle Projekte passt, existiert in Catalysis keine fixierte Vorgehensweise. Stattdessen gibt es eine Reihe von Prozess-Patterns, aus denen das passende ausgewählt und entsprechend ausgestaltet werden kann. Vorgeschrieben sind nur einige Merkmale wie komponentenbasierte Entwicklung, kurze Entwicklungszyklen, phasenbasierte Entwicklung, robuste Analysephase und organisatorische Reife. [SOU99]

Ein einfaches Standardprozess-Pattern kann z. B. folgenden Aufbau haben:

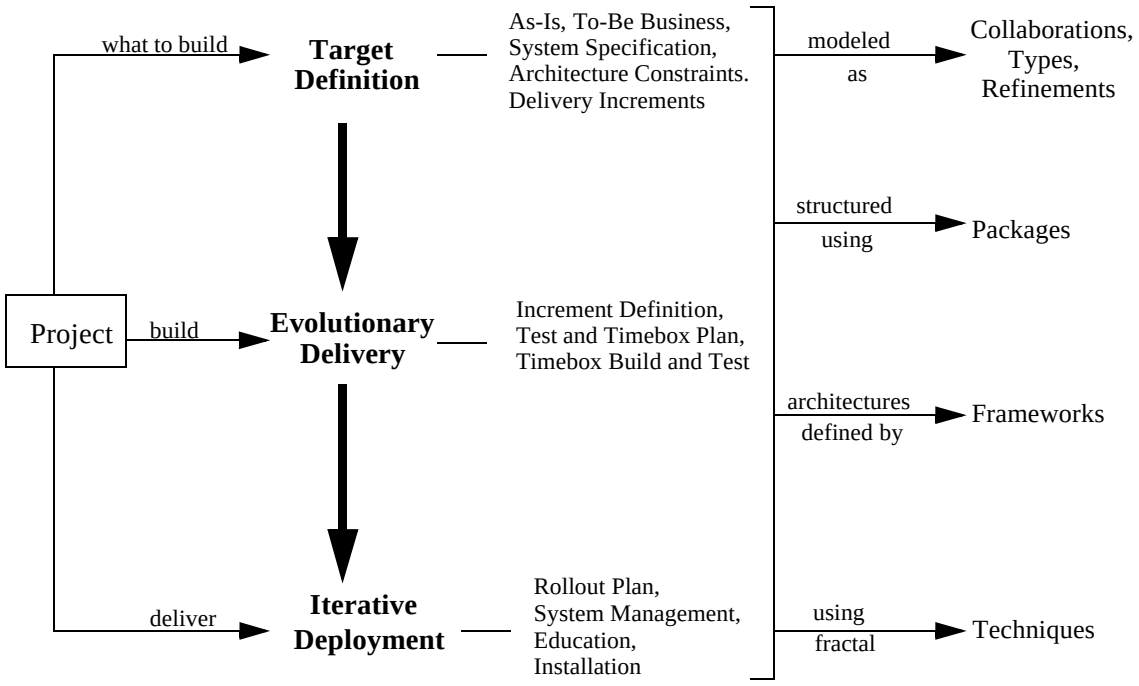


Abb. 2-5: Ausschnitt aus: CATALYSIS Concept Map 10 [CAT]

Der Entwicklungsprozess gliedert sich in mehrere Ebenen. Die obersten sind die Stufen (Stages). Diese werden in Schritte (Steps) und dann in Aufgaben (Tasks) unterteilt [ELT00]. Die durchlaufenen Projektstufen sind: (vgl. Abb. 2-5)

- **Target Definition:** Bestimmung der Ziele für das Grobdesign und die Implementierungsaktivitäten.

- **Evolutionary Delivery:** Es werden viele Inkremente erzeugt, jeweils mit kurzen Entwicklungszyklen und festgelegten Zeiträumen.
- **Iterative Deployment:** Auslieferung an den Kunden.

Durch die Verwendung von Prozess-Patterns ist Catalysis flexibel und für Eventualitäten gerüstet, welche die Autoren nicht vorhersehen konnten. Außerdem wird die Wiederverwendung von Prozessen ermöglicht. Das Fehlen einer genauen Anleitung zur Vorgehensweise erschwert aber den Einstieg in diese Methode. Es muss erst eine Entscheidung getroffen werden, welches Prozessmuster verwendet werden soll. Das Catalysis-Buch [SOU99] bietet hierzu keine Entscheidungshilfen.

Catalysis hat einige Schlüsselideen zur modernen objektorientierten Entwicklung beigetragen. Manche davon wie *Types*, *Collaborations* und *Refinement* fanden Eingang in die UML-Spezifikation (vgl. Abb. 2-6) [SOUb]. Außerdem wurde die wachsende Bedeutung rekursiver Entwicklungsprozesse erkannt. Diese werden von Catalysis unterstützt.

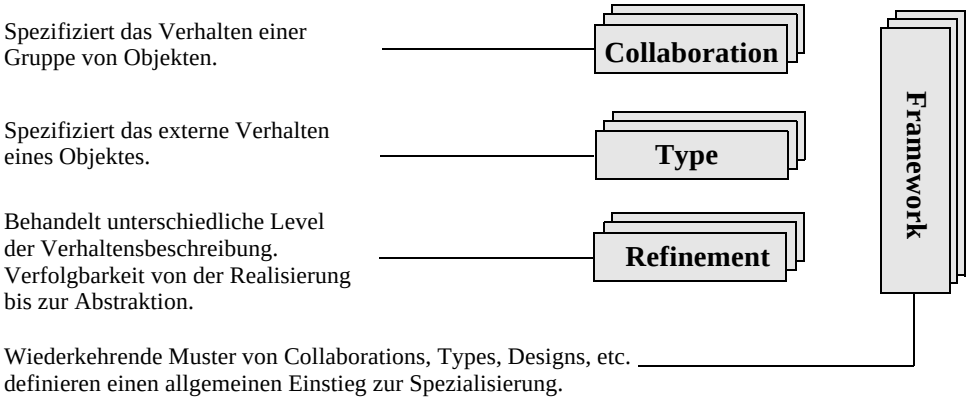


Abb. 2-6: Drei Modellierungskonstrukte mit Mustern als Frameworks. Aus [SOU99], Seite 33

Trotz dieser Leistungen hat Catalysis einige Schwachpunkte. Ähnlich wie der Unified Process ist diese Methode eher ein mit reichhaltigen Konzepten ausgestatteter Werkzeugkasten als ein sofort einsetzbares Werkzeug. Die Catalysis Methode krankt vielleicht mehr als andere Methoden an der Unfähigkeit, ihre Ideen in eine konsistente integrierte Form zu bringen. Die Methode ist hoch komplex und unfokussiert. Der potentielle Anwender erhält nur wenige Anhaltspunkte, wie er die vielen Ideen und Konzepte systematisch anwenden soll. [ATK01]

Dies führt dazu, dass der Einstieg in Catalysis nicht ganz einfach ist. Um diese Methode erfolgreich einzusetzten, müssen erst Erfahrungen gesammelt werden. Die Probleme fangen schon mit der Wahl eines geeigneten Prozesses an. Wie sollen ohne Erfahrung die zur Verfügung stehenden Prozess-Muster bewertet und angepasst werden?

Unter [ELT00] ist beschrieben, wie Catalysis eingesetzt werden kann, um einen Reifegrad von 5 nach dem *Capability Maturity Model* zu erreichen. Dass Catalysis sinnvoll eingesetzt werden kann, zeigt seine Verwendung bei Firmen wie EDS, Siemens, Texas Instruments, Credit Suisse, Olivetti, Hewlett Packard, Citibank Visa und Lockheed Martin [CAT]. Es ist allerdings davon auszugehen, dass diese Firmen über einen gewissen Reifegrad und genügend Erfahrung verfügen, um die Methode an ihre Bedürfnisse anzupassen.

2.2.4 ROOM

Real-Time **O**bject-**O**riented **M**odeling (ROOM) ist eine objektorientierte Methode, deren Einsatzgebiet die Entwicklung von Echtzeitsystemen ist. Sie wurde 1994 von Bran Selic, Garth Gullekson und Paul Ward entwickelt [SEL94]. Selic war Vizepräsident der Forschungs- und Entwicklungsabteilung bei der Firma ObjecTime, die zugehörige CASE-Tool *ObjecTime Toolset* entwickelt hat. Dadurch ist ROOM eng mit diesem Tool verzahnt.

ROOM basiert auf zwei wesentlichen Prinzipien:

- Komplexe Elemente können schrittweise aus einfacheren ausführbaren Modellelementen konstruiert und getestet werden.
- In allen Phasen der Systementwicklung kann das System durch dieselben formalen Konzepte repräsentiert werden.

Um die Operationen und Interaktionen zwischen verschiedenen konkurrierenden Objekten zu simulieren, enthält ROOM eine virtuelle Modellierungssprache. Ein Compiler übersetzt dann die Systembeschreibung in eine Form, die von einer virtuellen Maschine ausgeführt werden kann. Bei genügend detaillierter Ausarbeitung kann dies als früher Prototyp dienen. [COU97]

Zur Modellierung wurden neue Strukturelemente eingeführt: Aktoren (*actors*), Ports (*ports*), Protokolle (*protocols*) und Bindungen (*bindings*) (vgl. Abb. 2-7). **Aktoren** sind gekapselte Komponenten, die eine aktive Rolle im System modellieren. Sie kommunizieren mit der Umgebung, indem sie über ihre **Ports** Nachrichten schicken. Ein **Protokoll** spezifiziert die richtungsabhängigen Nachrichten, die zwischen den Aktoren ausgetauscht werden können. Ports können dabei als Interfaces angesehen werden. Nachrichten werden über **Bindungen** weitergeleitet, die zwei oder mehr Ports verbinden. Die Aktoren können eine hierarchische Struktur aufweisen, da Aktoren-Klassen wieder in einer Klassenhierarchie organisiert sein können. Durch Vererbung kann die gesamte Systemhierarchie verfeinert und wiederverwendet werden. [JAZ00]

Der Focus von ROOM liegt in der Prozesssicht. Um Systemverhalten und Dynamik zu modellieren, wird eine Abwandlung der State-Charts, die sogenannten ROOM-Charts (Abb. 2-7), benutzt. Diese modellieren das Verhalten der Aktoren. Dazu definieren die ROOM-Charts den Zustand eines Aktors, die Signale, welche die Zustandsübergänge auslösen und die Aktionen, die dabei durchgeführt werden. Dadurch hat ein Aktor zu jedem Zeitpunkt einen Zustand, der die Reaktion festlegt, wenn er über eine Port eine Nachricht empfangen wird.

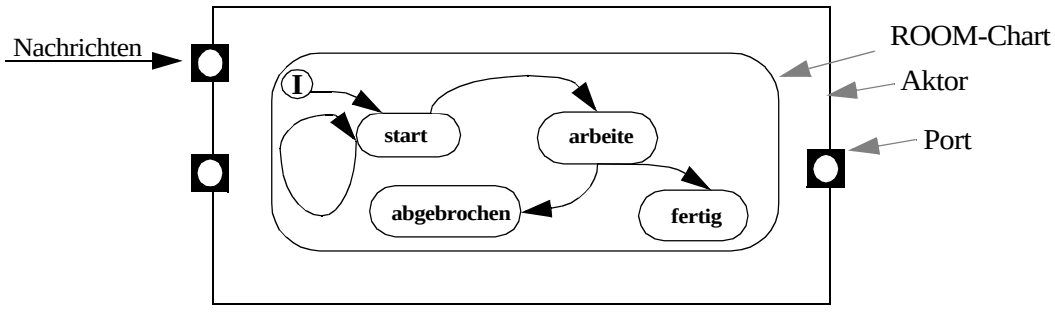


Abb. 2-7: Aktoren, Ports und ROOM-Charts

Im ROOM-Buch [SEL94] wird ausdrücklich darauf hingewiesen, dass es keinen festen Prozess gibt und die beschriebenen Beispiele nur als Vorschläge und Richtlinien anzusehen sind. Begonnen wird im Buch mit der Aufstellung der *model requirements*, die durch Szenarien beschrieben werden. Diese werden textuell oder durch *message sequence charts* modelliert. Der Prozess unterscheidet Modellierungsaktivitäten und auf einer höheren Ebene Produktentwicklungsaktivitäten. Die Modellierungsaktivitäten sind:

- **Discovery:** Ziel ist es, ein Verständnis davon zu bekommen, was das System tun soll. Dazu sollen die Anforderungen verstanden und ein Modell entwickelt werden. Dieses soll möglichst wenig mit späteren Lösungen zusammenhängen. Zu Beginn wird identifiziert, was zum System und was zur Umgebung gehört. Physikalische Objekte werden als Aktoren modelliert und mit Protokollklassen ausgestattet. Aufbauend auf den Protokollen wird eine erste Version des eng mit ihnen verbunden Verhaltens entwickelt.
- **Invention:** Durch das tiefere Verständnis werden neue Klasse mit zugehörigen Protokollen und Verhaltensweisen identifiziert. Wichtig ist, dass die in dieser frühen Phase modellierte „Lösung“ nur dem tieferen Verständnis dient. Sie kann später beim Übergang von der Analyse zum Design verworfen werden.
- **Validation:** Überprüfung der Übereinstimmung des Modells mit den Anforderungen.

Die Modellierungsaktivitäten zielen auf die Entwicklung eines Modells. Von zentraler Bedeutung sind dabei die Szenarien. Auf einem höheren Level laufen die Produktentwicklungsaktivitäten Analyse, Design & Implementierung und Test ab. Das hierbei erstellte Produkt muss außer der Erfüllung der Anforderungen nichts mehr mit der bei der Modellbildung erstellten Lösung zu tun haben. [SEL94]

Wie der Prozess in Verbindung mit dem ObjecTime Toolset ablaufen kann, beschreiben Steve Courtney und John Laws von der Firma ObjecTime in ihrem Artikel [COU97]. ROOM wird nicht mehr weiterentwickelt, sondern ist in einer UML-Variante aufgegangen (vgl. 2.2.5). Das *ObjecTime Toolset* wurde nach der Fusion der Firmen ObjecTime und Rational 1999 zum UML basierten *RoseRT* weiterentwickelt.

2.2.5 UML/ROOM

Unter [SEL98] ist von Bran Selic und James Rumbaugh ein auf ROOM basierender Ansatz beschrieben, wie Echtzeitsysteme mit Hilfe der UML entwickelt werden können. Dabei wurden die in ROOM eingeführten Begriffe nach UML umgesetzt. Die **Kapsel** (*capsule*) entspricht dem Aktor. Eine Kapsel kann Interfaces haben, die genau wie bei ROOM **Ports** (*ports*) heißen. Die möglichen Nachrichten werden in **Protokollen** (*protocols*) spezifiziert und können über **Konnektoren** (*connectors*) zwischen den Kapseln ausgetauscht werden. Daneben wird der Begriff der **zusammengesetzten Kapsel** (*composite capsule*) eingeführt. Diese repräsentieren ein Netzwerk von zusammenarbeitenden Kapseln, die eigene Ports haben können. Durch dieses Konzept soll die Wiederverwendung erleichtert werden. [SEL]

2.2.6 RealTime UML

Die **U**nified **M**odeling **L**anguage ist eine reine Modellierungssprache und keine Methode. Fast alle modernen Software-Entwicklung-Methoden und CASE-Tools unterstützen sie. Es wurden verschiedene Vorschläge bei der OMG eingereicht, um UML an Echtzeitaspekte anzupassen. Eine Entscheidung über eine Echtzeit-UML Variante steht aber noch aus. Eine Möglichkeit wie UML eingesetzt werden kann, um Echtzeitsysteme zu entwerfen, beschreibt Bruce Douglass in seinem Buch „RealTime UML“ [DOU98].

2.2.7 KobrA

Die KobrA-Methode (**Ko**mponenten**b**asie**r**te **A**nwendungsentwicklung) entstand im Rahmen des KobrA-Projekts, das vom Bundesministerium für Bildung und Forschung ins Leben gerufen wurde. Beteiligt waren die Softlab GmbH aus München als Projektleiter, die Psipenta GmbH aus Berlin, das Fraunhofer IESE aus Kaiserslautern [IESE] und die GMD FIRST aus Berlin. Das IESE ist primär für die hier beschriebene KobrA-Methode [KOBRA] verantwortlich. Mitarbeiter des IESE schrieben ein Handbuch zur Benutzung der Methode [ATK01].

Ein wesentlicher Aspekt von KobrA ist „Separation of Concerns“. Am deutlichsten wird dies durch die Trennung zwischen Produkt und Prozess. Produkt beschreibt „was“ und Prozess „wie“ es erzeugt werden soll. Auf diese Weise ist es möglich, Komponenten unabhängig von einer konkreten Implementierungstechnologie zu beschreiben. „Separation of Concerns“ findet sich auch bei der KobrA-Methode selber wieder. Sie besteht aus den drei orthogonalen Entwicklungsdimensionen „Abstraction“, „Genericity“ und „Composition“ (Abb. 2-8).

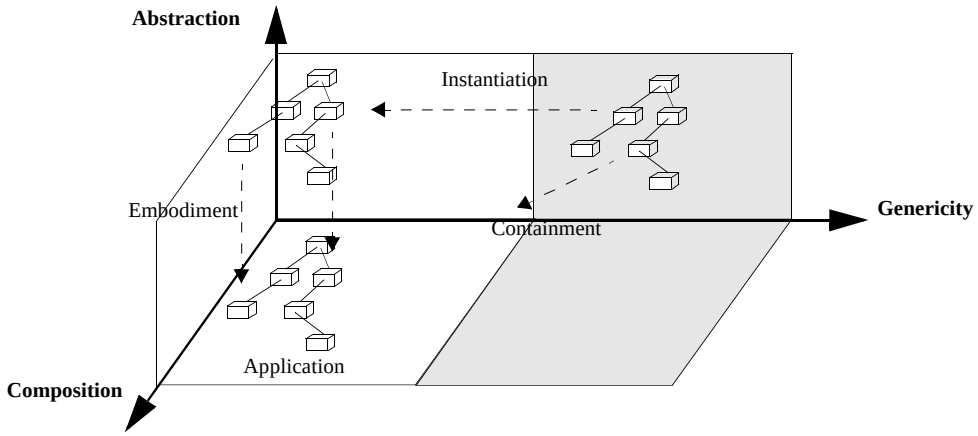


Abb. 2-8: Entwicklungsdimensionen (aus [ATK01])

„Genericity“ beschreibt den Spezialisierungsgrad. Dadurch korrespondiert „Genericity“ mit dem *Product Line Engineering*. So enthält das *Framework* die Funktionalität aller Produktlinien. Durch Instantiierung sinkt die „Genericity“ und die Modelle beschreiben nur noch eine Produktlinie. „Composition“ steht mit dem *Component Modeling* in Verbindung. Dabei findet durch *Containment* eine Dekomposition des Systems in feinere Teile statt. „Abstraction“ beschreibt den Abstraktionsgrad. Durch Reduzierung dieser Dimension, durch *Embodiment*, erhält man schließlich eine ausführbare Repräsentation des

Systems. Entwicklungsaspekte, die mit dieser Dimension zu tun haben, fallen unter den Begriff *Component Embodiment*. Dabei wird den abstrakten Modellen eine konkrete Form verliehen. [ATK01]

Durch Integration von *Product Line Engineering* und *Component Modeling* unterstützt KobrA zwei moderne Wiederverwendungstechnologien. Diese unterscheiden sich im Grad ihrer Granularität. Auf dem größten Granularitätslevel tritt das *Product Line Paradigm* zu Tage. Dieses unterteilt den Entwicklungszyklus in zwei Teile. Der erste beschäftigt sich mit der Entwicklung eines *Frameworks*, der zweite mit der Entwicklung einer *Application*. Ein *Framework* enthält alle Features einer Produktlinie. Features, die nicht in allen Produkten enthalten sind, werden mit dem Stereotyp <<variant>> gekennzeichnet. Durch Instantiierung des *Frameworks* ensteht die *Application*. Diese enthält nur noch die Funktionalität eines Produkts. Auf einer tieferen Ebene wird KobrA durch das *Component Paradigm* gesteuert. So sind sowohl *Frameworks* als auch *Applications* durch Komponentenhierachien organisiert. KobrA führt als neuen Begriff die *Komponent* (**K**obrA C**omponent**) ein. Dadurch soll eine Abgrenzung zu Komponententechnologien wie JavaBeans oder COM stattfinden, die Komponenten eher „physisch“ auffassen. Der Begriff *Komponent* steht für „logische“ Komponenten, welche die logischen Bausteine eines Software-Systems darstellen. Weiterhin unterstützt KobrA auch die inkrementelle Software-Entwicklung. [ATK01]

Prozess

KobrA’s *Product Line Engineering* Prozess (Abb. 2-9) wird durch die beiden Prozesse *Framework Engineering* und *Application Engineering* realisiert. Dies sind eigenständige Prozesse, die aus mehreren Aktivitäten und Subaktivitäten bestehen und die Artefakte *Framework* und *Application* erzeugen.

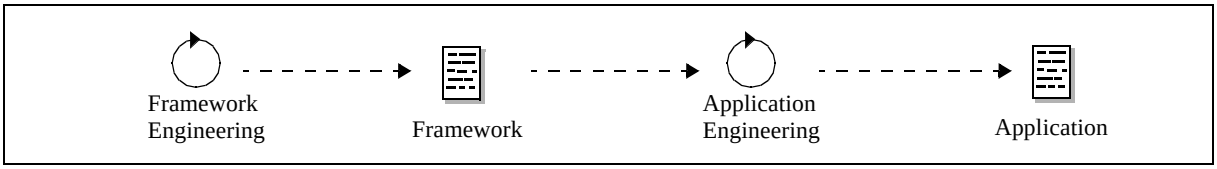


Abb. 2-9: Product Line Engineering (aus [ATK01])

Framework Engineering

Der Prozess *Framework Engineering* besteht im Wesentlichen aus den Aktivitäten: [ATK01]

- **Context Realization:** Es werden eine Reihe von Modellen erzeugt, welche die Systemumgebung beschreiben. Diese bilden das Artefakt *Generic Context Realization*. Es besteht aus *Structural Model*, *Activity Model*, *Interaction Model* und *Decision Model*.
- **Komponent Specification:** Durch die Spezifikation der Komponente werden die nach außen sichtbaren Eigenschaften definiert. Es wird das Artefakt *Generic Context Realization* erzeugt, das aus *Structural Model*, *Functional Model*, *Behavioural Model* und *Decision Model* besteht.
- **Komponent Realization:** Diese Aktivität erzeugt eine Beschreibung des internen Designs der Komponente. Dazu wird das Artefakt *Generic Komponent Realization* erzeugt, dessen Bestandteile *Structural Model, Activity Model, Interaction Model* und *Decision Model* sind.

- **Komponent Implementation:** Die erzeugten Modelle und Artefakte werden in eine Form übersetzt, aus der automatisch der ausführbare Code erzeugt werden kann. Die erzeugten Artefakte der *Komponent Implementation* sind *Source Code, Structural Model, Physical Component Model* und *Pseudo-Code*. Sie bilden das Artefakt *Generic Komponent Implementation.*

Weiterhin gibt es zur Wiederverwendung die Aktivität *Komponent Reuse*. Zur Qualitätssicherung werden die Aktivitäten *Inspection, Measurement of Structural Properties* und *Testing* unterschieden. Den kompletten Produktfluss der Aktivität *Framework Engineering* zeigt Abb. 2-10.

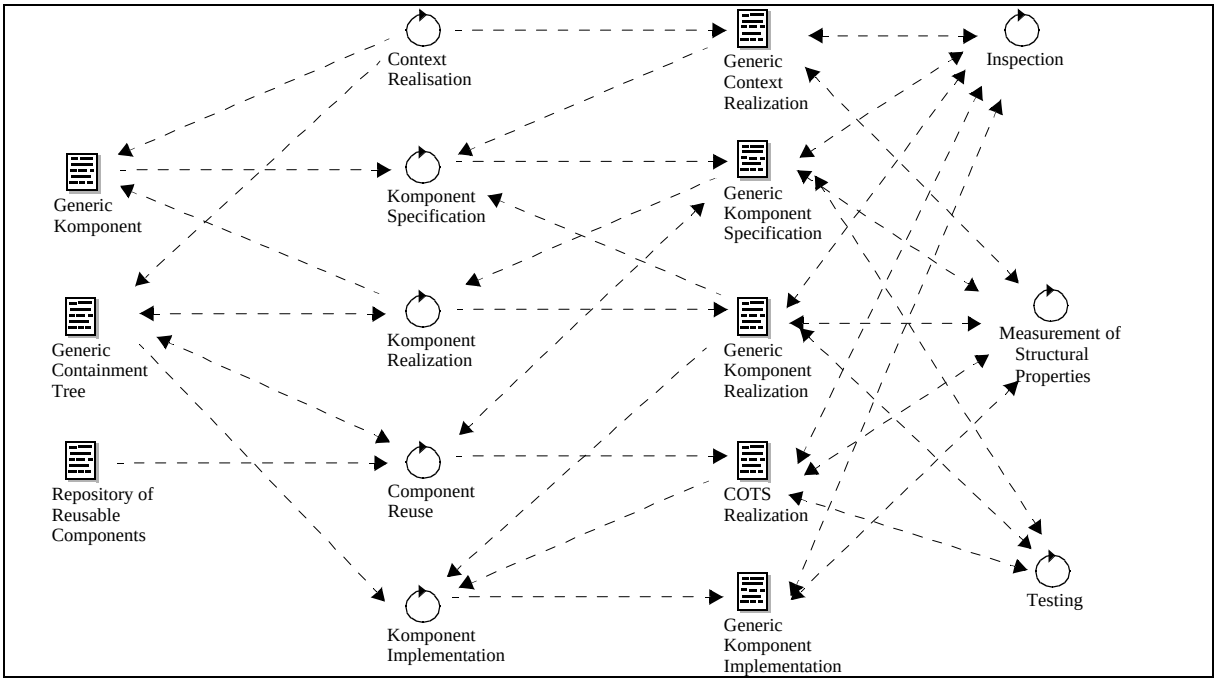


Abb. 2-10: Framework Engineering (aus [ATK01])

Während den Aktivitäten *Context Realisation, Komponent Specification, Komponent Realization* und *Komponent Implementation* werden sogenannte *Decision Models* aufgestellt (vgl. Tabelle 1-1), welche die variablen Teile der Diagramme, Tabellen und Modelle erfassen. Variabilitäten werden beispielsweise durch Features verursacht, die nicht von allen Produkten einer Produktlinie unterstützt werden. Gekennzeichnet werden die Variabilitäten durch den Stereotyp <<variant>>. Das *Framework* enthält die Features der kompletten Produktlinie. Durch Auflösung der *Decision Models* findet während des *Application Engineerings* die Instantiierung des *Frameworks* statt. Dies bedeutet, dass die instantiierten Modelle und Artefakte nur noch ein Produkt der Produktlinie unterstützen. Die variablen Teile werden durch die Instantiierung entfernt.

ID		Frage	Variation Point	Entschluss	Effekt
Diagramm-/ Modelltyp	1	Wird die Operation *XY* benötigt?	Operation *XY*	ja (default)	Entferne Stereotyp <<variant>>
				nein	Entferne Operation *XY*
	⋮	⋮	⋮	⋮	⋮

Tabelle 1-1: Entscheidungstabelle

Application Engineering

Die Aktivitäten *Application Context Realization*, *Application Komponent Specification*, *Application Komponent Realization* und *Application Komponent Implementation* des *Application Engineerings* hängen eng mit den entsprechenden Aktivitäten des *Framework Engineerings* zusammen. Allerdings bezieht sich *Application Engineering* auf ein konkretes Produkt der Produktlinie. Daher müssen die variablen Teile des *Frameworks* entfernt werden. Durch Auflösung der *Decision Models* findet eine Instantiierung des *Frameworks* statt und die Variabilitäten werden entfernt. Gestartet wird dazu mit der Instantiierung der *Generic Context Realization* zu einer *Specific Context Realization*. Danach wird die Instantiierung rekursiv für alle Komponenten und alle Entwicklungsebenen weitergeführt.

Zusätzlich zu den beschriebenen Aktivitäten enthält *Application Engineering* die Aktivitäten *Construction*, *Building* und *Releasing*. Diese hängen stark von den verwendeten Technologien ab und werden daher von KobrA nur oberflächlich beschrieben. *Construction* bedeutet die Zusammenstellung der während der Implementierung erzeugten Artefakte in eine Form, die von den verwendeten Tools verarbeitet werden kann. *Building* meint die Erzeugung einer direkt ausführbaren Repräsentation der physischen Komponente. Beim *Releasing* findet die Konfiguration der relevanten Komponenten statt, so dass ein auslieferbares Produkt entsteht. Den kompletten Produktfluss der Aktivität *Application Engineering* zeigt Abb. 2-11.

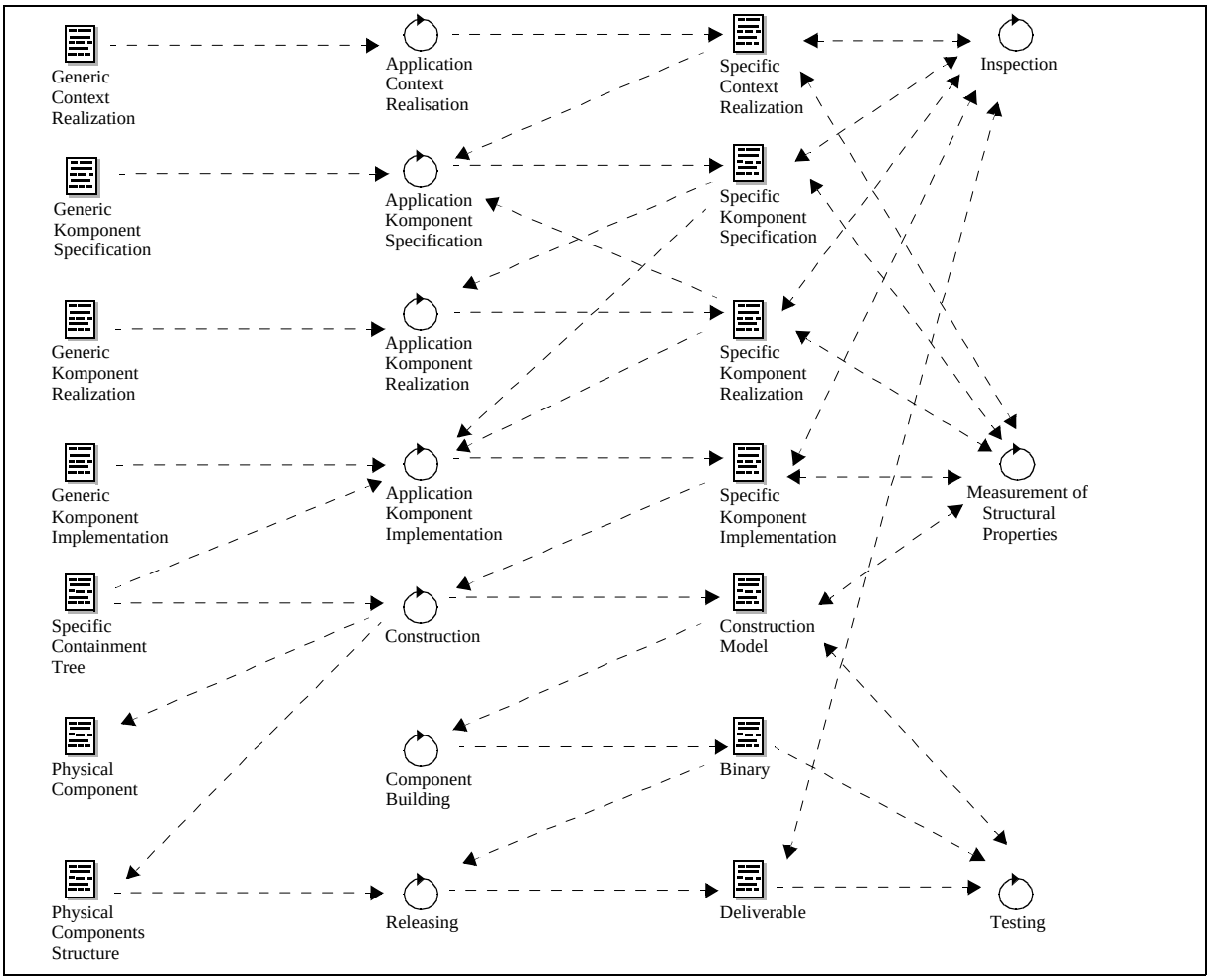


Abb. 2-11: Application Engineering (aus [ATK01])

2.2.8 Wiederverwendung und Komponententechnologien

Wiederverwendung

Angesichts immer komplexerer Softwaresysteme spielt die Wiederverwendung eine wichtige Rolle. Um „das Rad nicht jedesmal neu erfinden zu müssen“, wurden eine Reihe von Wiederverwendungstechniken entwickelt. Das Fehlen einer konkreten Anleitung hat zur Folge, dass Wiederverwendung meistens in einer ad hoc Manier oder zufällig stattfindet. Ein bekanntes Beispiel dafür ist die Objektorientierung, die für die Unterstützung der Wiederverwendung wie geschaffen schien. In der Praxis konnten diese Erwartungen jedoch nicht erfüllt werden. Das Wissen, dass sich Objekte zu Wiederverwendung eignen, hilft ohne eine Beschreibung des Wiederverwendungsprozesses nicht weiter. In den letzten Jahren wurden einige zusammenhängende Techniken entwickelt, die das Potential haben, die Wiederverwendungsproblematik zumindest teilweise zu lösen:

- **Komponentenbasierte Entwicklung**
- **Architectural Styles und Desing Patterns**
- **Product Line Engineering**
- **Klassenbibliotheken**

Diese Techniken ermöglichen Wiederverwendung auf unterschiedlichen Ebenen. Komponenten sind individuelle ausführbare Einheiten, aus denen Lösungen zusammengesetzt werden. Beispielsweise gibt es von Drittanbietern fertige COTS-Komponenten (**C**ommercial **O**f **T**he **S**helf) zu kaufen, die direkt eingesetzt werden können. Es wird aber auch die Wiederverwendung von Eigenentwicklungen erleichtert. Komponenten stellen das kleinste Wiederverwendungsgranulat dar. Dagegen ermöglichen *Architectural Styles* und *Design Patterns* Wiederverwendung auf einer höheren Ebene. In diesem Zusammenhang spielt die UML eine wichtige Rolle, da sie es möglich macht, Architektur und Design grafisch darzustellen und zu erfassen. In gewisser Weise wird dadurch auch die Wiederverwendung von Erfahrungen ermöglicht. *Product Line Engineering* stellt das gröbste Wiederverwendungsgranulat dar. Die Entwicklung neuer Applikationen besteht dabei eher in der Instantiierung wiederverwendbarer Teile zu einem neuen Ganzen als in der Programmierung. Diese Teile werden in einem Framework gespeichert, das alle Artefakte, die bei der Implementierung der Komponenten erstellt wurden, enthält. [ATK01]

Eine andere Form der Wiederverwendung stellen Bibliotheken dar. Viele moderne Programmiersprachen und Entwicklungswerkzeuge stellen Klassenbibliotheken zur Verfügung. Diese erleichtern die Software-Entwicklung, in dem für verschiedene Problemstellungen fertige Klassen zur Verfügung gestellt werden. Verbreitet ist dies z. B. bei GUI-Elementen wie Menüleisten oder Schaltknöpfen. Durch Benutzung dieser Klassen findet Wiederverwendung statt.

Komponententechnologien

Ähnlich wie bei ihrem Aufkommen die Objektorientierung [SEL94] wird das Komponentenparadigma als elegante Lösung zur Wiederverwendung angesehen. Diese Erwartungen erfüllten sich nur teilweise, da es an der methodischen Unterstützung fehlt. Der Schwerpunkt aktueller Komponententechnologien wie CORBA, COM/COM+, .NET, Java Beans oder EJB/J2EE liegt auf der Implementierungsebene.

Um die Vorteile des Komponentenparadigmas voll auszuschöpfen, ist aber die volle Unterstützung der Komponentenaspekte über den gesamten Entwicklungsprozess nötig, wie dies z. B. KobrA und Catalysis für sich beanspruchen. Es genügt daher nicht, wenn eine Methode z. B. Java Beans erzeugen kann. Vielmehr müssen Komponenten von Anfang an eine zentrale Rolle spielen. [ATK01]

Die bekanntesten Komponentenmodelle sind:

- **CORBA:** Die **C**ommon **O**bject **R**equest **B**roker **A**rchitecture [CORBA] ist die Spezifikation einer Infrastruktur für die Kommunikation zwischen verteilten Objekten. Die Basis bildet der **O**bject **R**equest **B**roker (ORB). Er stellt Dienste für verschiedene Anwendungsobjekte zur Verfügung. CORBA spezifiziert den Aufbau, die Schnittstellen und die Funktionalität eines ORBs. Es wird beschrieben, wie Anwendungen zu implementieren sind, bei denen Objektdienste durch Server angeboten und von Clients in Anspruch genommen werden. Daneben werden auch diverse Basisdienste, sogenannte Services, spezifiziert. Diese sind u. a. für Namensdienste, Sicherheit, Ereignisse, Persistenz und Objektverwaltung verantwortlich. Im Hinblick auf die Komponentenorientierung führt CORBA mit der Version 3 ein eigenes Komponentenmodell, das **C**ORBA **C**omponent **M**odel (CCM), ein. Entwickelt wird CORBA von der Object Management Group. [ANA00], [HEU97], [SEI99]
- **COM/COM+:** Das **C**omponent **O**bject **M**odel [COM] ist ein Produkt der Firma Microsoft. Es werden Komponentenobjekte eingeführt. Dies sind Objekte, die im Hinblick auf Wiederverwendung und Verteilung den Charakter von Komponenten haben. Der von COM definierte binäre Objektstandard lässt die Schnittstellen aller Objekte nach außen hin einheitlich erscheinen. Dadurch ist es gleichgültig, in welcher Programmiersprache die Komponente implementiert wurde. Beispielsweise kann eine mit Visual Basic erstellte Anwendung auf Methoden eines Objekts zugreifen, welches in C++ implementiert wurde. COM ist ein Standard zur Interaktion zwischen Objekten. Ähnlich wie CORBA bietet COM Standardservices wie Interfaces, Namensdienste, Persistenz und Objektverwaltung an. COM+, der Nachfolger von COM, stellt außerdem Services bereit, die sich um Transaktionen und Messaging automatisch kümmern. **D**istributed **COM** (DCOM) ist eine Infrastruktur, um die Verteilung von Komponenten zu realisieren. [ANA00]
- **.NET:** „.NET“ soll Microsofts zukünftige Plattform zur Entwicklung von Internetanwendungen werden. Sie steht damit in direkter Konkurrenz zu Suns J2EE. Ein zentraler Bestandteil von .Net sind Web-Services. Ein Web-Service stellt eine Black Box dar, deren Funktionalität flexibel eingesetzt werden kann, ohne Implementierungsdetails zu kennen. Die Funktionalität wird über wohldefinierte Schnittstellen zur Verfügung gestellt. Damit ist ein Web-Service vergleichbar mit einer Komponente. Im Gegensatz zu DCOM wird kein objektspezifisches Protokoll verwendet, sondern auf Http und XML zurückgegriffen. Dies hat den Vorteil, dass jedes System, das diese beiden Standards unterstützt, Web-Services integrieren kann. [STA00b], [WIL00], [WIT00]
- **Java Beans:** Definition: „eine Java Bean [BEAN] ist eine wiederverwendbare Softwarekomponente, die in einem visuellen Softwarekonstruktionswerkzeug (Builder) verwendet und manipuliert werden kann“ [STE98]. Um dies zu ermöglichen, verfügen sie über eine standardisierte Schnittstelle aus Properties, Methoden und Events. Im Wesentlichen besteht dieses Komponentenmodell aus Namenskonventionen und Schnittstellen für Ereignisbehandlung und Laufzeitin-

formationen. Diese liegen beispielsweise den Oberflächenkomponenten aus den Bibliotheken AWT und Swing zugrunde. [ANA00], [LOU98]

- **EJB/J2EE: E**nterprise **J**ava **B**eans (EJB) haben mit Java Beans kaum Gemeinsamkeiten. Diese Spezifikation beschreibt sogenannte Geschäftsobjekte, die von Containern unterstützt werden. Die Container stellen Dienste zur Lösung von Problemen wie Transaktionsmanagement, Persistenz, Lastverteilung und Ressourcen-Pooling zur Verfügung. Damit werden die Entwickler von Fachkomponenten entlastet. Die Idee ist, dass einmal entwickelte Enterprise Java Beans in beliebigen Containern und damit Applikationsservern eingesetzt werden können. EJBs sind Bestandteil der **J**ava **2 E**nterprise **E**dition (J2EE). In dieser definiert Sun eine Reihe von Technologien um die Entwicklung von Web-Applikationen zu unterstützen. [HOL01], [STA00a]

Komponentenmodelle sind keine Methoden zur Softwareentwicklung. Sie geben nur eine Spezifikation vor, die bei Einhaltung bestimmte Eigenschaften, abhängig vom verwendeten Modell, garantiert. Beispiele sind die Möglichkeit zur Zusammenarbeit oder die Verwendbarkeit in grafischen Entwicklungsumgebungen. Schwerpunkt der Komponentenmodelle ist die Implementierungsebene.

2.3 Bewertung der Methoden

Die in Kapitel 2.1 vorgestellten Methoden haben unterschiedliche Vor- und Nachteile. Um eine Bewertung vorzunehmen, werden in Abschnitt 2.3.1 zuerst wünschenswerte Eigenschaften einer idealen Methode zur Softwareentwicklung aufgezählt und kurz erläutert. Anschließend werden die Methoden in Abschnitt 2.3.2 bezüglich dieser Merkmale bewertet und Vor- und Nachteile aufgezählt. Dabei spielt besonders die Eignung der Methoden für die Entwicklung eingebetteter Systeme eine Rolle. In Abschnitt 2.3.3 werden die Ergebnisse zusammengefasst und die am besten geeignete Methode ausgewählt.

2.3.1 Eigenschaften einer idealen Methode zur Software-Entwicklung

Eine ideale Methode zur Software-Entwicklung soll einfach, systematisch, flexibel, skalierbar und anwendbar sein. Daneben ist die Unterstützung von UML, Komponenten, Wiederverwendung und Echtzeitaspekten sinnvoll. [ANA00], [ATK01]

Einfachheit ist eine wichtige Eigenschaft jeder Methode. Diese soll nicht mit redundanten Features überladen sein, die unnötige Komplexität schaffen. Durch Einfachheit werden die Einführung einer neuen Methode vereinfacht und Fehler vermieden. Außerdem sinkt die Hemmschwelle bei den Anwendern, die neue Technologie auch umzusetzen und zu unterstützen.

Systematik meint, dass die Methode einerseits klar und eindeutig definiert ist und andererseits dem Anwender genau sagt, was er als nächstes zu tun hat. Dies wird beispielsweise durch das Vorhandensein eines klaren Prozesses ermöglicht. Methoden, die eher einer Werkzeug- oder Techniksammlung entsprechen, wie der Unified Process oder Catalysis, lassen eine solche Systematik vermissen. Sie bieten dem Anwender eine Fülle von Möglichkeiten, ohne ihm zu sagen, welche er davon einsetzen soll.

Flexibilität ist wichtig, um eine Methode zu erweitern und an eigene Wünsche anzupassen. Wird dies übertrieben, besteht jedoch die Gefahr, dass dies mit der Forderung nach einer strengen Systematik kollidiert. Beispielsweise ist Catalysis flexibel, aber unsystematisch.

UML-Unterstützung ist sinnvoll, da die UML sich zu einem weitverbreiteten Modellierungsstandard entwickelt hat und es unsinnig ist, beim Umstieg auf eine neue Methode jedesmal eine neue Notation zu erlernen. Da UML nicht allumfassend sein kann, führen manche Methoden zusätzliche Diagrammtypen ein.

Die Unterstützung von **Wiederverwendung** durch moderne Wiederverwendungstechnologien wie Komponenten, Frameworks und Product Line Engineering gewinnt immer mehr an Bedeutung. Die Beurteilung der Methoden richtet sich auch danach, ob nur gesagt wird, dass Wiederverwendung möglich ist (z. B. bei ROOM) oder auch konkret beschrieben wird, wie dies geschehen soll. Bei der Komponentenunterstützung wird darauf geachtet, ob Komponenten in allen Phasen der Entwicklung unterstützt werden oder erst auf der Implementierungsebene eine Rolle spielen. Im Zuge immer komplexerer Programme steigt die Bedeutung der Wiederverwendung.

Echtzeitaspekte können für eingebettete Systeme eine große Rolle spielen. Außer bei ROOM werden sie von den meisten Methoden nicht explizit berücksichtigt. Trotzdem ermöglichen auch andere Methoden die Entwicklung von Echtzeitsystemen, beispielsweise OMT im SE I Praktikum [ROM98].

Skalierbarkeit meint die durchgängige Verwendung derselben Konzepte und Diagramme auf allen Abstraktionsebenen. Wenn möglich sollen Artefakte und Prozessschritte hierarchisch sein, d. h. es kann ausgehend vom größten Artefakt, dem Produkt, immer weiter hineingezoomt werden, ohne dass sich das Darstellungskonzept radikal ändert. Es tauchen immer wieder die selben Diagrammtypen auf.

Eine wichtige Eigenschaft ist die **Anwendbarkeit**. Sie bezieht sich auf den Kosten-Nutzen-Effekt der Methode. Der Nutzen einer Methode, die in der Theorie perfekt funktioniert, aber in der Praxis so aufwändig ist, dass sie nicht eingesetzt werden kann, ist begrenzt. Unter dem Aspekt der Anwendbarkeit spielt auch der spätere Anwender eine große Rolle. Der Widerstand gegen eine neue Methode ist um so größer, je umständlicher sie zu bedienen ist oder je mehr Zusatzaufwand sie verursacht. Die Anwendbarkeit einer Methode kann durch die Unterstützung von Standards, wie der UML gesteigert werden, da die Einarbeitungszeit sinkt.

Einige dieser Eigenschaften hängen zusammen. Ein klar definierter Prozess wirkt sich beispielsweise positiv auf die Einfachheit und Systematik der Methode aus. Dafür wird eventuell die Flexibilität eingeschränkt.

2.3.2 Methodenvergleich

OMT und eingebettete Systeme

Obwohl bei OMT keine besondere Berücksichtigung der Eigenheiten eingebetteter Systeme stattfindet, ist diese Methode so flexibel, dass sie durchaus für ihre Entwicklung verwendet werden. Beispielsweise wurde beim Software Engineering I Praktikum an der Universität Kaiserslautern [ROM98] OMT dazu benutzt, eine Gebäudesteuerung zu realisieren. Das System war mit einem einfachen Scheduler und einem Timer ausgestattet. Auf diese Weise können auch Echtzeitaspekte modelliert werden. Speziell

berücksichtigt werden sie allerdings von OMT nicht. Eine für eingebettete Systeme wichtige Neuerung von OMT war die Erkenntnis, dass die dynamische Modellierung ebenso bedeutend ist wie die statische. Dazu wurden *Statecharts* und *Sequence Diagrams* eingeführt [GOM00].

OMT hat aber auch einige Nachteile. So unterstützt diese Methode hauptsächlich die Analysephase. Ein wesentliches Merkmal dieser Phase ist die Erstellung einer möglichst abstrakten Beschreibung des Systems. Besonderheiten der Programmiersprachen und des physikalischen Modells werden dabei kaum berücksichtigt [VER98b]. Dies kann bei eingebetteten Systemen aber durchaus von Bedeutung sein. Auch leiden darunter Einfachheit, Systematik und Anwendbarkeit der Methode etwas. Auf der anderen Seite wird dadurch Flexibilität ermöglicht. OMT unterstützt die Skalierung der Problemstellung. Dies kann beispielsweise durch die Statecharts geschehen, mit denen zuerst das grobe Verhalten modelliert werden kann, bevor später eine Verfeinerung stattfindet. Moderne Wiederverwendungstechnologien wie Produktlinien und Komponenten werden von OMT ebensowenig unterstützt wie die UML. Insgesamt ist OMT damit etwas veraltet.

Unified Process und eingebettete Systeme

Der Unified Process ist eine Methode, die so flexibel und vielfältig ist, dass sie prinzipiell für alle Arten von SW-Systemen eingesetzt werden kann, also auch für die Entwicklung eingebetteter Systeme. Echtzeitaspekte werden von dieser Methode nicht explizit berücksichtigt. Daher sind eventuell Anpassungen erforderlich. Diese sind möglich, da der Unified Process eher eine Methodensammlung darstellt, aus der man sich das herausholt was benötigt wird, als eine konkrete Methode. Dadurch werden allerdings die Systematik und die Anwendbarkeit des Unified Process beeinträchtigt. Der Einstieg in diese Methode fällt durch die Vielfalt an unterschiedlichen Techniken schwer. Eine Verbesserung dazu, allerdings auf Kosten der Flexibilität und Abhängigkeit von der Firma Rational, bietet der Rational Unified Process.

Der Unified Process unterstützt die UML und moderne Wiederverwendungstechnologien. Problematisch dabei ist aber, dass Komponenten erst auf der ausführbaren Ebene unterstützt werden [ATK01]. Die durchgängige Verwendung derselben Diagramme und Modelle wird unterstützt. Damit kann der Unified Process gut skaliert werden.

Catalysis und eingebettete Systeme

D‘Souza und Wills behaupten in ihrem Buch [SOU99], dass Catalysis für das Design von kritischen Applikationen, wie z. B. für eingebettete Systeme, geeignet ist. Begründet wird dies mit der Präzision der Catalysis-Spezifikation und der Möglichkeit, beim Design mit variabler Strenge zu arbeiten. Als Beispiel geben sie an, dass Vor- und Nachbedingungen in natürlicher Sprache oder in einem Formalismus aufgeschrieben werden können. Ebenso kann beim Design mit mehr oder weniger Lagen gearbeitet werden. Abgesehen davon, dass die Strenge bei den meisten Methoden in gewissen Grenzen variabel ist, wird damit nur begründet, dass es mit Catalysis auch möglich ist, unzuverlässige Software zu entwerfen. Dies gilt natürlich für alle Methoden, wenn sie nicht sauber angewendet werden. Aber Catalysis erschwert die korrekte Anwendung durch seine Komplexität und die mangelnde Focussierung zusätzlich (vgl. 2.2.3).

Neben modernen Wiederverwendungstechnologien wie Komponenten und Frameworks unterstützt Catalysis die UML. Allerdings schafft es Catalysis nicht, seine vielfältigen Ideen in eine konsistente Form zu bringen [ATK01]. Darunter leiden die Skalierbarkeit und Einfachheit genauso wie die Anwendbarkeit erheblich. Dafür verfügt Catalysis, ähnlich wie der Unified Process, durch seine große Sammlung an unterschiedlichen Techniken über eine hohe Flexibilität.

Echtzeitaspekte werden wie bei den vorher behandelten Methoden nicht speziell unterstützt, können aber modelliert werden. Dass Catalysis für die Entwicklung eingebetteter Systeme nicht gänzlich ungeeignet sein kann, zeigt auch der Einsatz dieser Methode bei einer Reihe von Firmen wie Siemens, Texas Instruments, Olivetti und Lockheed Martin [CAT]. Diese Firmen verfügen allerdings über einen gewissen Reifegrad und setzen Catalysis teilweise seit Jahren ein.

ROOM und eingebettete Systeme

ROOM wurde speziell für Real-Time Aspekte entwickelt. Dynamisches Systemverhalten lässt sich durch die ROOM-Charts und die eingeführten Strukturelemente Aktor, Port und Bindung gut modellieren. Da in allen Phasen der Systementwicklung das System durch dieselben formalen Konzepte repräsentiert wird und komplexe Elemente schrittweise aus einfacheren aufgebaut sein können, ist diese Methode gut skalierbar. Im ROOM-Buch wird herausgestellt, dass kein fester Prozess existiert, sondern nur Empfehlungen und Richtlinien. Dadurch wird eine hohe Flexibilität auf Kosten der Einfachheit und Systematik ermöglicht.

Als ROOM entwickelt wurde, war eine der wichtigsten Neuerungen die Integration der Objektorientierung. Als deren Vorteil wird u. a. die Wiederverwendung sowohl im kleinen als auch im großen Maßstab durch abstrakte Klassen, Polymorphismus und Vererbung angegeben [SEL94]. Wie dies konkret geschehen soll, wird nicht erklärt. Zwar werden Klassen in ROOM gelegentlich als Komponenten bezeichnet, von der Integration moderner Wiederverwendungstechnologien wie Komponenten, Frameworks, Architectural-Styles oder Architectures kann aber keine Rede sein.

Insgesamt führte ROOM eine Menge guter Ideen ein, die aber nicht mehr ganz aktuell sind. OO und dynamische Systemmodellierung werden mittlerweile von fast allen modernen Methoden unterstützt. Außerdem fehlt die Unterstützung der UML. Nachteilig ist auch die Werkzeugzentrierung von ROOM auf das ObjecTime Toolset. Moderne CASE Tools unterstützen die Modellierungssprache von ROOM nicht. Zwar kann die Methode auch von Hand eingesetzt werden, in der Praxis ist dies aber nicht sinnvoll und beeinträchtigt die Anwendbarkeit erheblich.

ROOM/UML und eingebettete Systeme

Die Bewertung von ROOM trifft auch auf ROOM/UML zu. Es stimmt sicher, dass die Einführung zusammengesetzter Kapseln die Wiederverwendung erleichtert [SEL], aber wie dies konkret geschehen soll, darüber schweigen sich der Selic und Rumbaugh aus. Überhaupt bleiben die beiden Artikel [SEL] und [SEL98] unspezifisch und gehen nur auf Strukturelemente und Modellierungsaspekte ein. Eine konkrete Vorgehensweise oder gar ein Prozess werden nicht erwähnt. Die beiden Artikel behandeln für sich alleine gesehen keine Methode sondern erläutern nur, wie die Modellierungssprache von ROOM nach UML umgesetzt werden kann. Die wesentlichen Konzepte wurden unverändert übernommen.

KobrA und eingebettete Systeme

Moderne Wiederverwendungstechnologien wie Komponenten und Produktlinien sind ein zentraler Bestandteil der KobrA-Methode und werden durchgängig unterstützt. Genauso durchgängig findet die Verwendung derselben Diagramme und Konzepte auf allen Abstraktionsebenen statt. Damit ist KobrA gut skalierbar. Der Einstieg in KobrA braucht wie bei den anderen Methoden auch eine gewisse Zeit. Vor allem der genaue Startpunkt bei der *Context Realization* ist nicht klar definiert. Dadurch bleibt KobrA zunächst sehr flexibel und ist nicht auf bestimmte Problemstellungen beschränkt. Später wird nach einer strikteren Vorgehensweise verfahren.

Für KobrA gelten bezüglich der Unterstützung von Echtzeitaspekten die gleichen Aussagen wie bei den anderen Methoden, ausgenommen ROOM und ROOM/UML. Es ist möglich, Echtzeitverhalten zu modellieren. Dies wird aber nicht explizit unterstützt.

Nachteilig für die Anwendbarkeit der KobrA-Methode ist die fehlende Werkzeugunterstützung. Durch die Unterstützung der UML können die Modelle zwar mit den meisten CASE-Tools erzeugt werden. Zu wünschen ist aber eine durchgängige Werkzeugunterstützung. Dies würde auch den Einstieg in KobrA erleichtern. Ein solches Werkzeug ist zwar geplant, steht aber momentan nicht zur Verfügung.

2.3.3 Auswertung und Ergebnis des Methodenvergleichs

Keine der vorgestellten Methoden ist ideal. Sie haben unterschiedliche Vor- und Nachteile. Der Versuch sie alle zu vereinigen birgt jedoch die Gefahr, statt einer systematischen Methode nur eine weitere Sammlung unterschiedlicher Techniken zu generieren. Die Erweiterung einer einzelnen Methode erscheint daher sinnvoller. Natürlich spricht nichts dagegen, Ideen aus anderen zu integrieren. Die Ergebnisse des Methodenvergleichs sind in Tabelle 1-2 zusammengefasst:

	OMT	Unified Process	Catalysis	ROOM	ROOM/ UML	KobrA
Einfachheit	o	-	-	o	o	o
Systematik	o	o	-	o	-	+
Flexibilität	o	+	+	+	+	+
UML	-	+	+	-	+	+
Wiederverwendung	-	o	+	-	o	+
Echtzeitaspekte	o	o	o	+	+	o
Skalierbarkeit	o	+	-	+	+	+
Anwendbarkeit	o	o	-	-	-	o
Bewertungsschlüssel: + gut, o durchschnittlich, - schlecht						

Tabelle 1-2: Bewertung der Methoden

Speziell für eingebettete Systeme wurden nur ROOM und ROOM/UML entwickelt. ROOM/UML alleine ist keine vollwertige Methode, sondern bereitet nur den Übergang von der ROOM-Modellie-

rungssprache zu UML. ROOM und OMT sind beides ältere Methoden, deren Weiterentwicklung zum größten Teil eingestellt wurde und für die es daher immer schwerer wird, Support zu erhalten. Probleme bereitet ebenfalls die fehlende Unterstützung durch moderne Case-Tools. Wegen der weiten Verbreitung von OMT treten diese Nachteile für ROOM stärker zu Tage als für OMT. Außerdem berücksichtigen beide Methoden keine modernen Wiederverwendungstechnologien. Die Verwendung der in Rose RT enthaltenen Methode hätte ähnlich wie die Verwendung des Rational Unified Processes eine große Abhängigkeit von der Firma Rational zur Folge. Die KobrA - Methode erscheint wegen der durchgängigen Komponentenunterstützung am geeignetsten. Diese ist bei anderen Methoden wie beispielsweise dem Unified Process nur mit großem Aufwand nachrüstbar, da alle Aktivitäten und Phasen betroffen sind. Catalysis nimmt dies zwar auch für sich in Anspruch, ist aber ähnlich wie der Unified Process zu unspezifisch und versagt deshalb bei der Forderung nach Einfachheit und Systematik.

Trotzdem hat auch KobrA Schwächen. Diese werden in Kapitel 3 untersucht und die KobrA-Methode entsprechend modifiziert. In Kapitel 5 findet dann eine Evaluierung der Änderungen an einem Beispiel statt.

2.4 CASE-Tools

Computer **A**ided **S**oftware **E**ngineering Tools sollen Entwickler in allen Stadien des Software-Engineerings wie Analyse, Design, Implementierung und Dokumentation unterstützen. Dies erstreckt sich von der Aufstellung der Anforderungen bis zur Kodierung. Dazu bieten CASE-Tools eine durchgängige Beschreibungssprache an. Die hier vorgestellten Tools unterstützen z. B. alle UML. Da textuelle Beschreibungen meistens mehrdeutig sind, wurden unterschiedliche Diagrammtypen entwickelt, die der Kommunikation der Entwickler untereinander dienen. Außerdem ermöglichen sie die Modellierung von Sachverhalten in einer dem CASE-Tool verständlichen Form. Die Tools sind mit Zeichenfunktionen ausgestattet, um den Anwender bei der Erstellung der Diagramme zu unterstützen. Diese müssen in allen Phasen der Softwareentwicklung konsistent sein. Dazu muss das Tool auch mit Mehrbenutzerbetrieb und Zugriffsrechten umgehen können. Manche CASE-Tools können aus genügend detaillierten Diagrammen direkt den Code generieren (*Forward Engineering*) und so Kosten und Zeit sparen. Automatisch generierter Code erfordert allerdings häufig eine manuelle Nachbereitung, um ihn beispielsweise an bestimmte Style-Guides anzupassen. Vorhandene Klassen müssen oft z. B. zur Wiederverwendung analysiert und in Diagramme umgesetzt werden (*Reverse Engineering)*. Optimal ist, wenn das CASE-Tool beide Richtungen unterstützt (*Roundtrip Engineering*). Die lange Einarbeitungszeit macht den Einsatz von CASE-Tools nur für mittlere bis größere Projekte sinnvoll. [VER98a], [MUL00a]

Ausschnitt der auf dem Markt befindlichen CASE-Tools:

- **ObjecTime Toolset / ROSE RT:** Das ObjecTime Toolset ist eng verzahnt mit der ROOM-Methode (2.2.4). Beide wurden seit 1994 von der Firma ObjecTime entwickelt und vertrieben. Nach der Fusion der Firmen ObjecTime und Rational [RAT] 1999 ging das ObjecTime Toolset in einer speziellen Variante des Rational Produktes Rose in Rose RT auf.

- **Rational ROSE:** Dieses Produkt der Firma Rational [RAT] orientiert sich am Rational Unified Process (vgl. 2.2.2) und an der UML. Ziel ist die Unterstützung des gesamten Entwicklungsprozesses. Problematisch ist, dass die Implementierung nicht in Rational Rose selber, sondern in einer externen Entwicklungsumgebung erfolgt. Dazu wird aus dem Entwurf der Code generiert. Dieser wird von der Entwicklungsumgebung importiert, dort bearbeitet und danach wieder von Rational Rose reimportiert. Die dabei auftretenden Abweichungen machen eine manuelle Überprüfung nötig. So müssen eventuell Änderungen sowohl in der Entwicklungsumgebung als auch in Rational Rose von Hand vorgenommen werden. Im Rahmen der iterativen Entwicklung kommt dieser Zyklus mehrmals vor. Die Standardversion von Rational Rose unterstützt C++. Eine Java-Unterstützung ist gegen Aufpreis erhältlich. [WEG01]
- **StP/UML:** Produkte des Herstellers Aonix werden bereits seit 1985 unter dem Namen STP (**S**oftware **t**hrough **P**ictures) vermarktet [STP]. Seitdem wurden sie ständig weiterentwickelt und mit STP/UML liegt heute ein modernes UML basiertes Case-Tool vor. Eine frühere Version STP/OMT wurde beispielsweise im Software-Engineering I Praktikum [ROM98] an der Universität Kaiserslautern eingesetzt. STP/UML unterstützt die Codeerzeugung für die Programmiersprachen C++, IDL, Ada_95, TOOL und Java. *Reverse Engineering* ist ebenfalls möglich. Für die Erweiterung des Tools steht eine C-ähnliche Skriptsprache zur Verfügung. [BEN99]
- **OEW:** Die **O**bject **E**ngineering **W**orkbench wird vom Innovative Software [OEW] angeboten. Neben den UML-Diagrammen verfügt sie über Vererbungsdiagramme, die eine Schachtelung ermöglichen. Unterstützt wird die Generierung und Analyse von C++- und Java-Code. Erzeugter Quellcode lässt sich direkt in OEW betrachten und ändern. Änderungen werden direkt in die Diagramme übernommen. Damit ermöglicht OEW *Roundtrip Engineering*. [MUL00a]
- **OTW:** Die **O**bjekt**t**echnologie-**W**erkbank wurde von der OWiS Software GmbH entwickelt. OTW enthält ein eigenes Vorgehensmodell: SEPP/OT. Neben den Diagrammen der UML werden Entity-Relationship-Diagramme und der von OWiS entwickelte Objekt-Prozess-Diagrammtyp unterstützt. OTW kann C++- und Java-Code generieren. Dieser kann dann in externen Entwicklungsumgebungen bearbeitet werden. Umgekehrt wird auch der Import von Code unterstützt. Interessant ist auch der Versuch, die erzeugten Diagramme bezüglich ihrer Komplexität zu bewerten. OWiS wurde im Juni 2000 von Intershop übernommen. Die weitere Entwicklung von OTW ist unklar. [MUL00a]
- **objektiF:** Das Produkt des Herstellers microTOOL [OBJ] unterstützt die Erzeugung von C++- und Java-Code, der innerhalb von objektiF bearbeitet werden kann. Da Änderungen direkt in die Diagramme übernommen werden, ist *Roundtrip Engineering* möglich. Durch XML-basierte Im- und Export-Schnittstellen wird die verteilte Entwicklung unterstützt. Um objektiF an eigene Bedürfnisse anzupassen, kann das Tool in Visual Basic programmiert werden. [MUL00b]

Kapitel 3

Notwendige Anpassungen an KobrA

KobrA ist eine relativ neue Methode, die einen Schwerpunkt auf Komponenten- und Produktlinien-Technologien legt. Diese gewinnen im Zusammenhang mit Wiederverwendung und webbasierten Anwendungen immer mehr an Bedeutung. Bisher kam KobrA nur in einer Fallstudie zur Entwicklung eines Bibliothekssystems zum Einsatz [IESE00]. Gleichzeitig erobern eingebettete Systeme durch immer höhere Leistungsfähigkeit immer neue Anwendungsgebiete. Die wachsende Bedeutung eingebetteter Systeme macht Wiederverwendung und den Einsatz von Komponenten- und Produktlinien-Technologien auch für diese Aufgabenstellungen interessant.

In diesem Kapitel wird die Eignung der KobrA-Methode für die Entwicklung eingebetteter Systeme untersucht. Anhand der identifizierten Schwächen wird KobrA so modifiziert, dass die Schwachpunkte beseitigt und die Besonderheiten eingebetteter Systeme berücksichtigt werden.

KobrA besteht aus den beiden Phasen *Framework Engineering* und *Application Engineering*. *Framework Engineering* befasst sich mit der Erstellung eines Frameworks, dessen Modelle die Funktionalität aller Produktlinien enthält. *Application Engineering* generiert durch Instantiierung des Frameworks die Modelle einer spezielle Produktlinie. Beide Phasen sind in mehrere Aktivitäten unterteilt, die wiederum Artefakte erzeugen. Die Zusammenhänge werden durch Produktfluss-Diagramme dargestellt. Die meisten dieser Diagramme stimmen im Wesentlichen mit denen aus dem Buch [ATK01] überein. In diesem Kapitel werden nur die wichtigsten bzw. die modifizierten Produktfluss-Diagramme angegeben. Änderungen sind grau unterlegt. Für eine komplette Auflistung aller Produktfluss-Diagramme wird auf das entsprechende Kapitel des Buches verwiesen.

Die Tätigkeiten und Artefakte werden in der Reihenfolge untersucht, die bei der Entwicklung eines eingebetteten Systems auftritt. Neue Artefakte bzw. Aktivitäten werden an der Stelle ihres ersten Auftretens erläutert. Alle neuen Artefakte sind verträglich mit dem Stereotyp <<variant>>. Die Modifikationen sind teilweise durch die während der Fallstudie in Anhang A aufgetretenen Probleme motiviert. Für die Modelle, Diagramme und Aktivitäten werden die englischen Fachbegriffe verwendet. Diese werden zur besseren Abgrenzung kursiv geschrieben.

3.1 Framework Engineering

Das Grundkonzept des Prozesses *Framework Engineering* wurde bereits in Kapitel 2.2.7 erläutert. Dort befindet sich auch das zugehörige Produktfluss-Diagramm (Abb. 2-10). Im Wesentlichen besteht der Prozess aus den Aktivitäten *Context Realization, Komponent Specification, Komponent Realization* und *Komponent Implementation*.

3.1.1 Context Realization

Die *Context Realisation* besteht aus den Aktivitäten *Enterprise Modeling, Structural Modeling, Usage Modeling, Interaction Modeling, Variability Identification, Decision Modeling* und *Komponent Identification* welche die Artefakte *Generic Enterprise Model, Generic Structural Mode*l, *Generic Activity Model, Generic Interaction Model, Generic Data Dictionary, Generic Containment Tree, Generic Komponent* und *Decision Model* erzeugen. *Generic* bedeutet dabei, dass diese Modelle die Funktionalität aller Produktlinien umfassen. Erst durch Instantiierung des Frameworks während des Prozesses *Application Engineerings* ändert sich der Status der Modelle von *generic* zu *specific*. Zusätzlich zu den in der Abbildung aufgeführten Artefakten wird in [ATK01] die Erstellung eines weiteren Artefaktes, dem *Framework Scope* empfohlen.

3.1.1.1 Framework Scope

Sinn des *Framework Scope*s ist es, möglichst ohne eine detaillierte Untersuchung aller Facetten der Applikationsumgebung auszukommen. Die Analyse der Produktlinien wird auf die Erstellung einer *Scope Definition Table* reduziert. In den Spalten stehen die Produktlinien, in den Reihen die zu realisierenden Features. Wenn eine Produktlinie ein Feature enthält, wird dies in der Tabelle markiert. Auf diese Weise gelingt es relativ schnell die Zusammenhänge zwischen Produktlinien und Features herauszuarbeiten. [ATK01]

Scope Definition Table

Die *Scope Definition Table* ist auch für den Einsatz bei eingebetteten Systemen geeignet. Sie hält die Zusammenhänge zwischen den Produktlinien und den geforderten Features fest. Modifikationen können erforderlich sein, wenn eingebettete Systeme aus einzelnen zusammenarbeitenden Hardwarebaugruppen aufgebaut sind. Eine Baugruppe besteht aus mehreren Bauteilen, die logisch zusammengehören. So kann beispielsweise eine Ansammlung von Motoren als Antriebsbaugruppe bezeichnet werden. Durch die Zugehörigkeit zur Antriebsbaugruppe ist die Aufgabe dieser Motoren intuitiv klar. Häufig wird eine Baugruppe durch eine entsprechende Softwarekomponente gesteuert. Wird dann beispielsweise die Antriebsbaugruppe bei einem Produkt der Produktlinie nicht verwendet, können alle zugehörigen Softwarekomponenten weggelassen werden. Aus diesem Grund werden zusätzlich zu den Features die verwendeten Hardwarebaugruppen in der *Scope Definition Table* (vgl. Tabelle 3-1) aufgelistet. Für den Fall, dass neben der Software auch die Hardware an die Aufgabenstellung angepasst wird, kann die *Scope Definition Table* auch Zusammenhänge zwischen Features und Hardware darstellen.

		System A	System B
Features	Fahren	x	x
	Hupen		x
Einsatz-umgebung	Sand	x	
	Straße		x
Hardware-baugruppen	Fahrwerk	x	x
	Hupe		x

Tabelle 3-1: *Scope Definition Table*

Die Aufgabenbeschreibung eines eingebetteten Systems kann zusätzliche Anforderungen, beispielsweise an die Einsatzumgebung enthalten. Diese sind auch eine Art Feature. Wenn sich die Anforderungen für die verschiedenen Produkte einer Produktlinie unterscheiden, müssen sie in die *Scope Definition Table* aufgenommen werden, um die Zuordnung zu den Produkten der Produktlinie zu ermöglichen. Problematisch ist, dass viele dieser Anforderungen erst während der *Komponent Implementation* realisiert werden. Beispielsweise können Anforderungen bezüglich des Einsatzes eines Fahrzeugs auf einem bestimmten Bodenbelag durch eine entsprechende Geschwindigkeit erfüllt werden. Dies geschieht aber erst auf der Implementierungsebene. Damit kann das *Decision Model* am Ende der *Context Realisation*, welches die Features auf die Entscheidungen abbildet, solche Anforderungen nicht immer berücksichtigen. Trotzdem werden die Anforderungen, wenn sie eine Variabilität aufweisen, im *Decision Model* aufgeführt und auf die Stelle ihrer Erfüllung verwiesen (vgl. Tabelle 3-2). Weitere Modifikationen an der *Scope Definition Table* sind nicht nötig.

Feature		Entschluss	Effekt
Feature	Hupen	ja (default)	Context Realization Entscheidung ...
		nein	Context Realization Entscheidung ...
	⋮	⋮	⋮
Einsatz-umgebung	Sand	Bei *Komponent Implementation* der Komponente X erfüllt.	
	Straße	Bei *Komponent Implementation* der Komponente Y erfüllt.	

Tabelle 3-2: *Decision Model* für die Abbildung der Features auf die Entscheidungen

3.1.1.2 Enterprise Model - System Model

KobrA lässt dem Anwender bei der Erstellung des *Enterprise Models* große Freiheiten, um das Einsatzspektrum dieses Modells nicht einzuschränken. Im Gegensatz zu anderen Modellen stellt das *Enterprise Model* keinen wichtigen Teil des KobrA-Dokumentes dar, sondern ist vielmehr ein möglicher Startpunkt für die Erzeugung der Artefakte der *Context Realisation*.

Aufgabe des *Enterprise Models* ist es, die Natur des Unternehmens zu beschreiben, für das die zu erzeugende Komponente benötigt wird. Dabei werden Dinge von Bedeutung unter kompletter Ignorierung der Tatsache beschrieben, dass diese als Computersystem realisiert werden sollen. Es wird empfohlen, das mindestens die beiden Artefakte *Enterprise Concept Diagram* und *Enterprise Process Hierarchy* erzeugt werden. [ATK01]

Der Begriff *Enterprise* steht nicht zwingend für ein Unternehmen, sondern auch für andere Organisationsformen. Trotzdem ist er im Zusammenhang mit eingebetteten Systemen missverständlich. Durch Umbenennung des Modells von *Enterprise Model* in *System Model* lässt sich dieses Problem leicht beheben.

Enterprise Concept Diagram - System Concept Diagram

Aufgabe des *Enterprise Concept Diagrams* ist es, die organisatorischen Einheiten, Rollen und Dinge, für die diese verantwortlich sind, zu illustrieren. Im Prinzip lässt sich das *Enterprise Concept Diagram* auf eingebettete Systeme übertragen. Allerdings sind Anpassungen erforderlich, um einige Besonderheiten eingebetteter Systeme zu berücksichtigen. Es wird außerdem in *System Concept Diagram* umbenannt.

Ein eingebettetes System stellt einen viel kleineren und klarer abgegrenzten Sachverhalt dar, als ein Unternehmen. Im Gegensatz zur reinen Softwareentwicklung besteht durch die Wahl der Hardware bereits ein großes Vorwissen über die Art des Modellierung. Beispielsweise stehen die verwendeten Aktoren und Sensoren in der Regel bereits fest und finden sich später als Abstraktion in der Software wieder. Auch lassen sich eventuell logische Programmkomponenten identifizieren. Diese müssen sich nicht unbedingt in der späteren Realisierung wiederfinden. Die wesentlichen Zusammenhänge zwischen den logischen Programmkomponenten und den Aktoren und Sensoren werden im *System Concept Diagram* (Abb. 3-1) dargestellt.

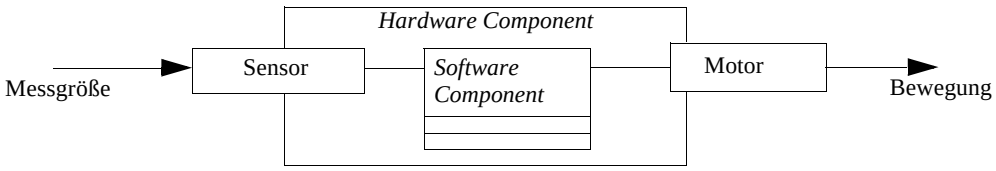


Abb. 3-1: *System Concept Diagram*

Der Aufbau des *System Concept Diagrams* ähnelt dem Aufbau eines Klassendiagramm. Zusätzlich zu Klassen und Komponenten wird der Begriff der *Hardware Component* eingeführt. Dieser bezeichnet Hardware, die Software enthält und ausführen kann. *Hardware Components* werden im Prinzip genauso behandelt wie *Software Components*. So werden für jede *Hardware Component* die Aktivitäten der KobrA-Methode durchgeführt. Um die *Hardware Components* voneinander abzugrenzen wer-

den sie eingerahmt. Hardwareelemente ohne eigene Rechenkapazität, wie z. B. Motoren, werden Hardwarebauteile genannt. Die Grenze zwischen der Software und der Außenwelt wird u. a. durch die Hardwarebauteile Aktor und Sensor überwunden. Diese werden auf der Grenzlinie der *Hardware Components* plaziert. Dadurch wird ein Überblick über die verwendeten Hardwarebauteile ermöglicht. Um darzustellen, auf was die Aktoren und Sensoren wirken bzw. reagieren, werden Pfeile benutzt, um die Wirkrichtung zu illustrieren. Zusätzlich werden die Pfeile mit der Einflussgröße beschriftet. Die Funktionalität der Hardwarebauteile wird später durch entsprechende Klassen abstrahiert. Eine *Hardware Component* enthält mindestens eine *Software Component*, die bei Bedarf weiter verfeinert werden kann. Dadurch, dass eine *Software Component* auf mehreren *Hardware Components* eingesetzt werden kann, ist Wiederverwendung von *Software Components* auch zwischen den *Hardware Components* möglich.

Die beschriebene Form der Darstellung eignet sich auch für den Entwurf eines Systems, das aus unabhängig programmierten zusammenarbeitenden *Hardware Components* besteht (Beispiel: Abb. 3-2). Diese können beispielsweise über Nachrichten miteinander kommunizieren. Das durch die einzelnen *Hardware Components* realisierte Gesamtsystem wird durch einen gestrichelten Rahmen umgeben, der die Systemgrenze markiert. Dabei muss darauf geachtet werden, dass Verbindungen zur Außenwelt wie beispielsweise der Motor in Abb. 3-2 auch über diese Grenze hinausgeht. Die Nachrichtenschnittstellen können in diesem Beispiel innerhalb der Systemgrenzen bleiben, da sie nur für die interne Kommunikation des Systems verantwortlich sind. Auf diese Weise sind beliebige Schachtelungen und Komplexitätsstufen möglich.

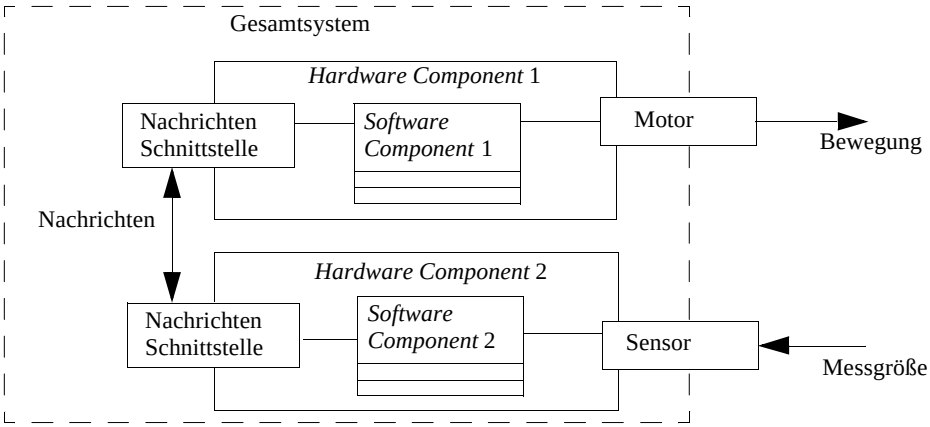


Abb. 3-2: *System Concept Diagram*

Enterprise Process Hierarchy - System Process Hierarchy

Aufgabe der *Enterprise Process Hierarchy* ist es, eine Hierarchie der Aktivitäten, in die das Unternehmen involviert ist, zu illustrieren. Um das Einsatzspektrum von KobrA nicht einzuschränken, wird die genaue Vorgehensweise von KobrA nicht vorgeschrieben. [ATK01]

Die *Enterprise Process Hierarchy* kann im Wesentlichen unverändert übernommen werden. Für eingebettete Systeme wird sie in *System Process Hierarchy* umbenannt. Um die Zugehörigkeit der Prozesse zu den *Hardware Components* darzustellen, kann eine obere organisatorische Ebene (gestrichelt) eingeführt werden (vgl. Abb. 3-3). Die Prozesse in der Blättern des Hierarchiebaumes werden laut [ATK01] *User Tasks* genannt, da sie die wichtigen Aufgaben der potentiellen Systembenutzter reflektieren. Für

eingebettete Systeme gibt es allerdings häufig keine *User Tasks*, da nicht unbedingt eine Interaktion mit einem Benutzer stattfinden muss. In solchen Fällen werden stellvertretend wichtige Prozesse, die zur Erfüllung der Aufgabe nötig sind und bereits jetzt identifiziert werden können, in die *System Process Hierarchy* eingetragen. Diese Prozesse bilden den Startpunkt für die funktionale Analyse des Systems. Optionale Prozesse werden grau unterlegt.

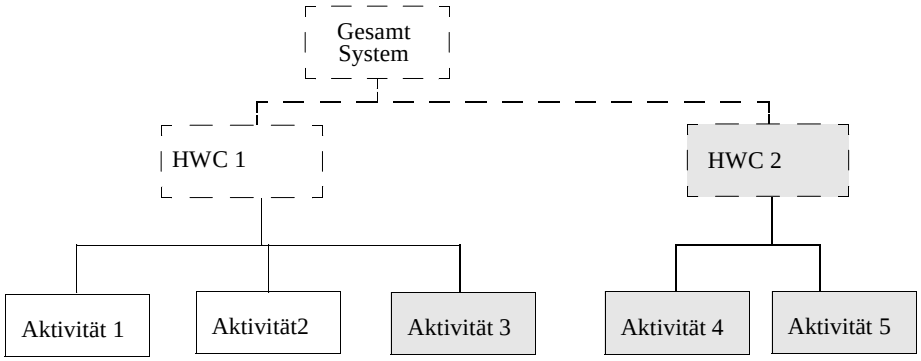


Abb. 3-3: *Roboter System Process Hierarchy*

Hardware Properties

Zusätzlich zu *System Concept Diagram und System Process Hierachy* wird KobrA um das Artefakt *Hardware Properties* (vgl. Tabelle 3-3) erweitert. Damit wird berücksichtigt, dass eingebettete Systeme technische Bestandteile wie Aktoren und Sensoren verwenden können. Diese können nicht einfach als logische Elemente abstrahiert werden. Beispielsweise genügt es bei einem Sensor nicht zu wissen, dass er etwas messen kann. Zusätzlich sind Kenntnisse über seine technischen Daten wie z. B. Messbereich und Messgröße nötig. In den *Hardware Properties* werden alle Hardwarebestandteile aufgelistet, die von den *Hardware Components* verwendet werden. Die technischen Daten komponentenspezifischer Bauteile können auch später bei der *Komponent Specification* dieser *Hardware Component* aufgeführt werden.

Auf den ersten Blick erscheint die Auflistung dieser Eigenschaften während der *Context Realization* verfrüht, da sie scheinbar erst bei der Implementierung benötigt werden. Wenn aber beispielsweise über die Antriebsmotoren bekannt ist, dass sie sich nur in eine Richtung drehen können, wird die Programmierung eines Rückwärtsganges sinnlos. Dieses Wissen kann bereits Auswirkungen auf die *Context Realization* haben. Die Kenntnis über die Fähigkeiten der Hardwarebauteile ist auch wichtig, um Entscheidungen zu treffen, wie und mit welchen Sensoren bzw. Aktoren eine Aufgabe erledigt werden soll. Beispielsweise beeinflusst die Wahl, ob zur Kollisionserkennung Tastsensoren oder Ultraschallsensoren verwendet werden, die weitere Entwicklung erheblich.

Bezeichnung	**Typ**	**gemessene / manipulierte Größe**	**Wertebereich**	**Einheit**
Tastsensor	Sensor	Hinderniskontakt	int, [0,100]	%

Tabelle 3-3: *Hardware Properties*

3.1.1.3 Structural Model

Class Diagrams

Die *Class Diagrams* erfordert keine tiefergehenden Modifikationen, es sei denn, dass der Aufbau mehrerer unabhängiger *Hardware Components* in einem Diagramm dargestellt werden soll. In diesem Fall wird das *Class Diagram* folgendermaßen erweitert. Für jede *Hardware Component* wird ein entsprechendes *Sub Class Diagram* erstellt und eingerahmt. *Software Components,* die miteinander kommunizieren, werden über einen Kommunikationskanal in Form einer gestrichelten Linie verbunden (vgl. Abb. 3-4). Für *Object Diagrams* gelten die gleichen Modifikationen, da sie eine Instantiierung der *Class Diagrams* darstellen.

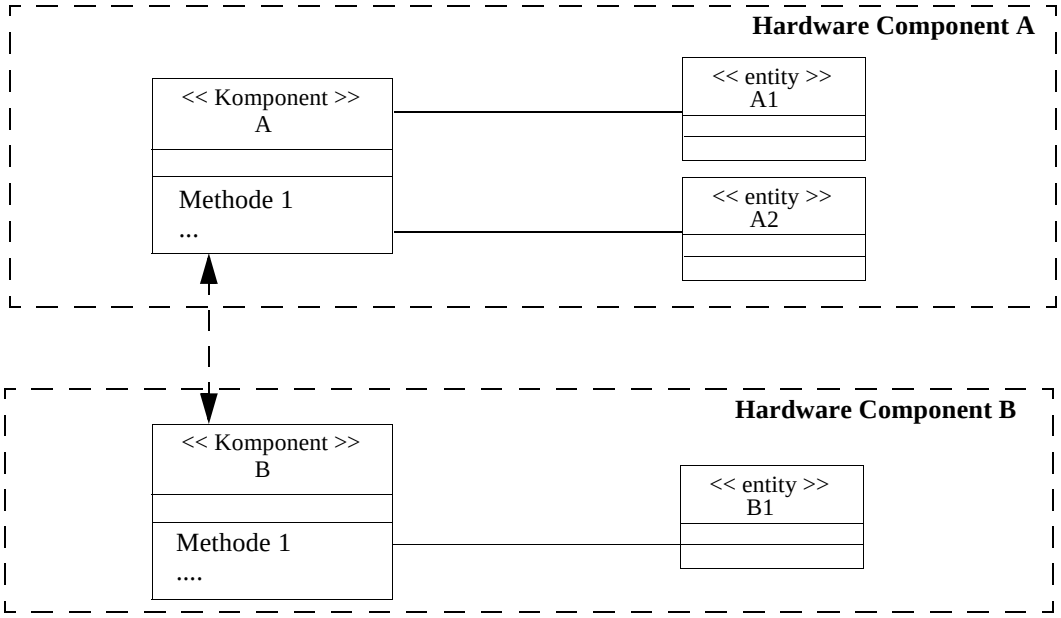


Abb. 3-4: *Class Diagram*

User Interface Artefacts

Für die *User Interface Artefacts* müssen keine Modifikationen durchgeführt werden. Allerdings entfallen sie häufig, da viele eingebettete Systeme über kein *User Interface* verfügen. Erforderlich sind *User Interface Artefacts* beispielsweise, wenn das System mit einem Bildschirm ausgestattet ist und vom Benutzer Anweisungen entgegen nimmt. Wenn kein *User Interface* existiert, entfallen auch die entsprechenden Aktivitäten, die *User Interface Activities*.

3.1.1.4 Hardware Component Model

Die KobrA-Methode wird um das *Hardware Component Model* erweitert. Dieses berücksichtigt einige Besonderheiten, die bei Verwendung von mehreren *Hardware Components* auftreten. Dabei werden die beiden Artefakte *Hardware Component Containment Model* und *Message Model* erzeugt. Diese enthalten wichtige Informationen, die für die *Komponent Specification benötigt werden*. Die Aktivität zur Erzeugung des *Hardware Component Model* heißt *Hardware Component Modeling*.

Hardware Component Containment Model

Das *Hardware Component Containment Model* hält den Zusammenhang zwischen *Hardware* und *Software Components* in einer Tabelle fest (vgl. Tabelle 3-4). Dies muss während der *Context Realization* geschehen, da die nächste Aktivität, die *Komponent Realization*, komponentenspezifisch ist und somit keine komponentenübergreifenden Sachverhalte darstellen kann. Eine *Hardware Component* kann bereits auf der Ebene der *Context Realization* aus mehreren *Software Components* bestehen.

Hardware Component	Software Component
HWC1	SWC1
	SWC2
	⋮
HWC2	SWC8

Tabelle 3-4: *Hardware Component Model*

Message Model

Bevor mehrere *Hardware Components* zusammenarbeiten können, müssen ein Nachrichtenformat und Nachrichten definiert werden. Dies muss geschehen, bevor mit der weiteren Verfeinerung der einzelnen Komponenten begonnen wird, da die Kommunikation komponentenübergreifend erfolgt. Außerdem werden die Nachrichten für die *Activity*- und die *Sequence Diagrams* benötigt.

Nachricht	Codierung	Sender	Empfänger	Wirkung
sendeX	0	HWC A	HWC B	Tätigkeit 1 wird ausgeführt.
⋮	⋮	⋮	⋮	⋮

Tabelle 3-5: *Message Model*

3.1.1.5 Activity Model

Unter 3.1.1.2 wurde festgestellt, dass eingebettete System häufig keine *User Tasks* haben. Problematisch kann dies bei den Artefakten des *Activity Models* werden, da die *User Tasks* Ausgangspunkt sowohl für die *Activity Diagrams* und die *Activity Specifications*, als auch für die *Use Cases* sind. Wenn das System über keine *User Tasks* verfügt, werden an ihrer Stelle die Prozesse in den Blättern der *System Process Hierarchy* verwendet. Wenn eine *Hardware Component* eine andere benutzt, agieren die beiden ähnlich miteinander wie ein System und ein Benutzer. Die Zusammenarbeit von *Hardware Components* kann somit etwas ähnliches darstellen wie ein *User Task*. Dies muss in den Diagrammen berücksichtigt werden.

Activity Diagrams

Startpunkt für die Aufstellung der *Activity Diagrams* sind die *User Tasks*. Um deren Funktionalität zu beschreiben, können freie Aktivitäten verwendet werden. Das sind Aktivitäten, die nicht zu speziellen Datenobjekten gehören [ATK01]. Für wichtige freie Aktivitäten werden ebenfalls *Activity Diagrams* erzeugt. Eine Besonderheit bei der Aufstellung der Diagramme ist, dass eventuell keine *User Tasks* existieren. In solchen Fällen dienen die Prozesse aus der untersten Ebene der *System Process Hierarchy* als Startpunkt. Es kann auch sinnvoll sein, die Zusammenarbeit zwischen *Hardware Components* durch *Activity Diagrams* zu illustrieren. Weitere Modifikationen an den *Activity Diagrams* sind nicht erforderlich.

Activity Specification

Für alle wichtigen freien Aktivitäten wird die Erzeugung der *Activity Specification* empfohlen. Im Wesentlichen können die Tabellen der *Activity Specification* aus dem Buch über KobrA [ATK01] übernommen werden. Wegen der Besonderheiten eingebetteter Systeme ist es jedoch sinnvoll, die Tabellen um zusätzliche Punkte zu erweitern (vgl. Tabelle 3-6). Die Modifikationen sind grau unterlegt.

Name	Methodenname
Beschreibung	Kurze Beschreibung der Funktionalität
HW-Komponente	Die *Hardware Component*, zu der die Methode gehört
Erhält	Übergebene Argumente der aufrufenden Methode
Übergibt	Zurückgelieferte Argumente
Annahmen	Vorbedingungen, die erfüllt sein müssen, um das beschriebene Verhalten zu garantieren
Misst	Sensoren, deren Werte ausgelesen werden
Manipuliert	Aktoren, die gesteuert werden
Empfängt	Nachrichten von einer anderen *Hardware Component*
Sendet	Nachrichten an eine andere *Hardware Component*
Ergebnis	Auswirkung der Methode
Regeln	Regeln

Tabelle 3-6: Aufbau der *Activity Specification*

Use Case Model

KobrA gibt eine präzise und konkrete Interpretation der *Use Cases* vor. Es werden *User Task*, *User Interface Activity* und *System Operation* unterschieden. Für eingebettete Systeme eignet sich diese Interpretation nicht. So existieren eingebettete Systeme, die weder *User Tasks* noch *User Interface Activities* besitzen. Andererseits können beispielsweise die Sensoren wie ein Benutzer agieren und ein Verhalten anstoßen. In diesem Fall muss die Sensorik als *Actor* in den *Use Cases* berücksichtigt werden. Dies gilt aber nicht generell für alle Sensoren, da diese auch passiv implementiert sein können.

Wenn Sensoren beispielsweise innerhalb einer Methode abgefragt werden und nicht selbstständig ein Verhalten anstoßen, können sie nicht als *Actor* modelliert werden.

Es muss ebenfalls berücksichtigt werden, dass auch *Hardware Components* die Rolle eines *Actors* übernehmen können, beispielsweise wenn sie Nachrichten an eine andere *Hardware Component* verschicken und damit Verhaltensweisen starten.

Eigentlich muss auch ein Timer als *Actor* modelliert werden [GOM]. Dies lohnt sich aber nur, wenn er direkt Verhaltensweisen anstößt. Falls der Timer nur dazu dient, beispielsweise bei *preemptive Scheduling* (vgl. 4.2.1) die Tasks nach einer festen Zeitspanne weiterzuschalten, kann der Aufwand, den Timer in den *Use Cases* aufzuführen vermieden werden.

3.1.1.6 Interaction Model

Beim *Interaction Model* tritt dasselbe Problem auf wie bei den *Use Cases*. Eingebettete Systeme müssen keine *User Tasks* oder *User Interface Activities* haben. An ihrer Stelle werden die Prozesse der *System Process Hierarchy* verwendet. Weitere Modifikationen des *Interaction Models* sind nicht erforderlich.

3.1.1.7 Zusammenfassung

Die *Context Realization* wurde um einige neue Artefakte erweitert. Abb. 3-5 illustriert die Zusammenhänge zwischen den Artefakten und den Aktivitäten, die diese erzeugen. Neu ist neben dem in diesem Abschnitt eingeführten *Hardware Component Model* das *Experience Model*. Dieses wird ähnlich wie das *Data Dictionary* parallel erzeugt und kann alle Ebenen der KobrA-Methode betreffen. Durch das *Experience Model* sollen gesammelte Erfahrungen berücksichtigt werden. Eine genaue Beschreibung befindet sich in Abschnitt 3.2. Das *Enterprise Model* wurde durch das *System Model* und die *Enterprise Process Hierarchy* durch die *System Process Hierarchy* ersetzt. Außer dass für die neu eingeführten Artefakte entsprechende *Decision Models* erzeugt werden müssen, erfordert die Aktivität *Decision Modeling* keine Änderungen.

Es wird erwartet, dass bei den folgenden Aktivitäten der KobrA-Methode weniger Änderungen erforderlich sind wie bei der *Context Realization*. Grund ist, dass die Ebene der *Hardware Components* verlassen wird und nur noch *Software Components* auftauchen. Diesbezüglich sind also keine weiteren Modifikationen nötig. Hardwarebauteile wie beispielsweise Sensoren werden durch entsprechende Klassen abstrahiert und erfordern ebenfalls keine gesonderte Behandlung.

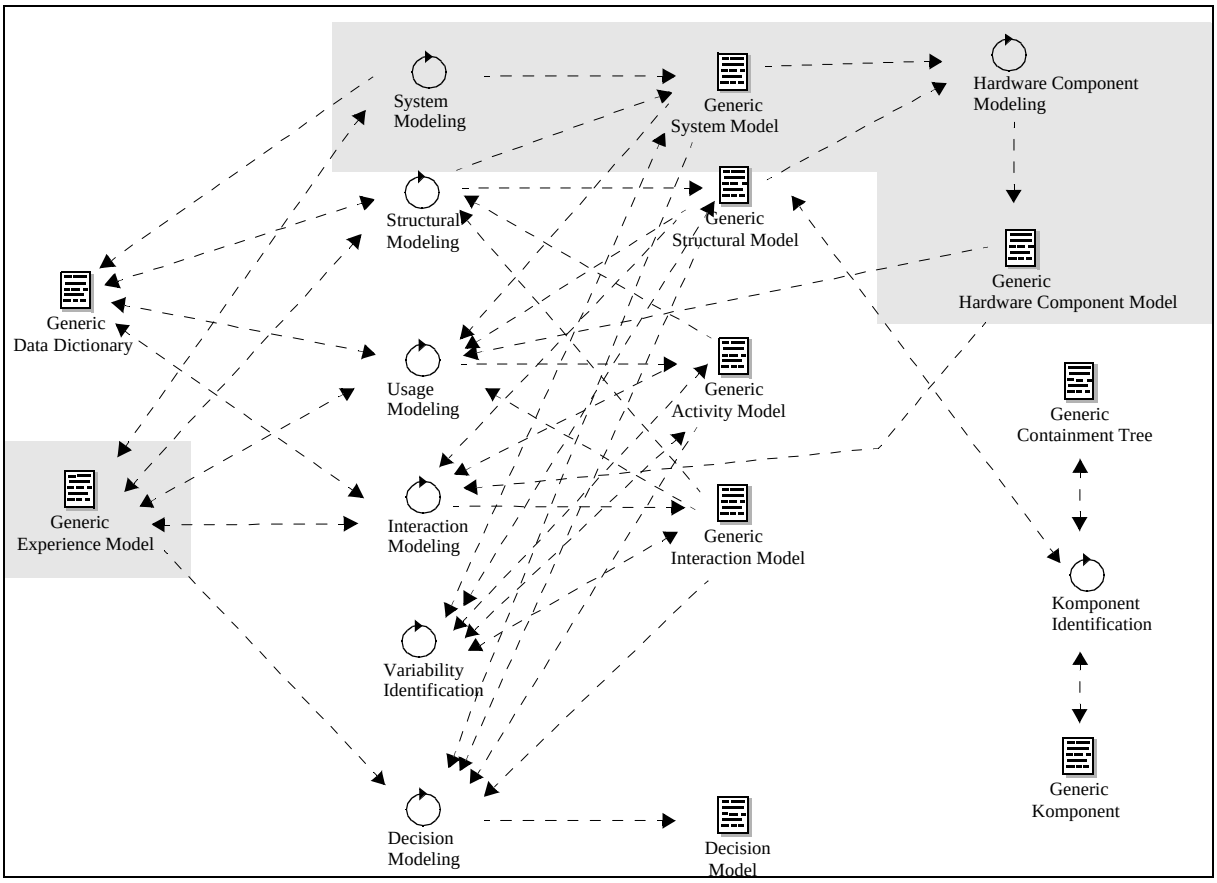


Abb. 3-5: Überarbeiteter Produktfluss der *Context Realization* (Original aus [ATK01])

3.1.2 Komponent Specification

Die *Komponent Specification* besteht aus den Aktivitäten *Structural Modeling*, *Functional Modeling*, *Behavioral Modeling*, *Variability Identification und Decision Modeling*. Die wichtigsten erzeugten Artefakte sind *Generic Structural Mode*l, *Generic Functional Model*, *Generic Behavioral Model* und *Decision Model*. Abb. 3-8 zeigt den überarbeiteten Produktfluss. Aufgabe der *Komponent Specification* ist es, einen Satz von Modellen zu erzeugen, die zusammen die nach außen sichtbaren Eigenschaften der Komponente zeigen. [ATK01]

3.1.2.1 Structural Model

Das *Structural Model* der aktuellen Komponente ist identisch mit dem Teil des *Structural Models* der *Context Realization*, der dieser Komponente entspricht. Da auf der Ebene der *Komponent Specification* die Komponente nur aus *Software Components* besteht, sind keine weiteren Anpassungen nötig.

3.1.2.2 Functional Model

Das *Functional Model* beschreibt die extern sichtbaren Effekte der Operationen, die von dieser Komponente unterstützt werden. Jede Operation wird in einer Tabelle beschrieben, der *Operation Specifica-*

tion. Die im Buch [ATK01] beschrieben Tabellen werden um die bereits bei der *Activity Specification* der *Context Realization* (vgl. Tabelle 3-6) neu eingeführten Punkte erweitert. Der Eintrag „HW-Komponente“ ist entfallen. Neu eingeführt wird der „Geräte-ID“. Dieser soll der Eigenschaft eingebetteter Geräte Rechnung tragen, dass externe Geräte wie Aktoren oder Sensoren angeschlossen werden können. Bisher wurde nur angegeben, was manipuliert bzw. gemessen wird. Für die weitere Entwicklung der Komponente ist es aber wichtig zu wissen, welcher Motor bzw. Sensor gemeint ist. Die *Operation Specifications* der einzelnen Operationen müssen nicht unbedingt alle Punkte enthalten. Es werden nur die sinnvollen aufgeführt.

Name	Name der Operation
Beschreibung	Kurze Beschreibung der Funktionalität
Constraints	Eigenschaften, die die Realisierung und Implementierung der Komponente begrenzen
Erhält	Übergebene Argumente der aufrufenden Methode
Übergibt	Zurückgelieferte Argumente
Geräte-ID	ID des Gerätes, das manipuliert bzw. mit dem gemessen wird.
Misst	Sensoren, deren Werte ausgelesen werden
Manipuliert	Aktoren, die gesteuert werden
Empfängt	Signale/Nachrichten von anderen Komponenten
Sendet	Signale/Nachrichten an andere Komponenten
Liest	Nach außen sichtbare Informationen, auf die von dieser Operation zugegriffen wird
Ändert	Nach außen sichtbare Informationen, die von dieser Operation geändert werden
Regeln	Regeln, die die Berechnung des Ergebnisses betreffen
Annahmen	Vorbedingungen über den Zustand der Komponente und die Eingaben, die erfüllt sein müssen, um das beschriebene Verhalten zu garantieren
Ergebnis	Auswirkung der Methode

Tabelle 3-7: Aufbau der *Operation Specification*

3.1.2.3 Behavioural Model

Beim *Behavioural Model* kommt zum Tragen, dass es bei eingebetteten Systemen durch äußere Einflüsse unerwartete oder logisch unmögliche Zustandsübergänge geben kann. So ist es denkbar, dass der Zustand verändert wird, ohne dass das System dies bemerkt. Abb. 3-6 zeigt dazu ein Beispiel, in dem ein eingebettetes System eine Tür steuert. Wenn das eingebettete System versucht die Tür zu öffnen, muss berücksichtigt werden, dass diese eventuell bereits durch eine Person von Hand geöffnet wurde. Dann muss beispielsweise der Türmotor abgeschaltet werden, bevor er beim vergeblichen Versuch, die Tür zu öffnen, zerstört wird.

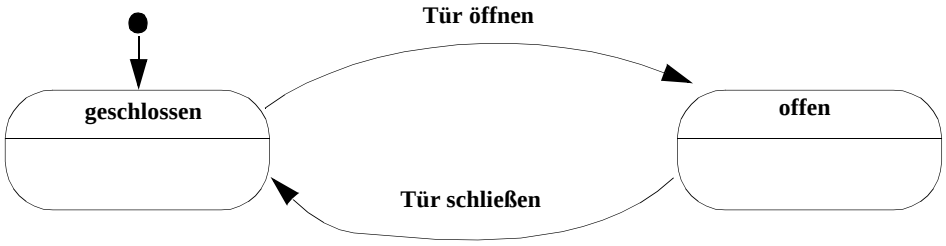


Abb. 3-6: *Statechart Diagram* mit Zustandsänderung von außen

Es ist auch denkbar, dass das eingebettete System gar kein Schließen der Tür vorsieht (vgl. Abb. 3-7). Wenn die Tür trotzdem von einer Person geschlossen wird, führt dies zu einem völligen Versagen des Systems. Abhilfe schafft der Übergang „Tür wurde geschlossen“. Dieser wird zwar theoretisch nicht gebraucht, erhöht aber die Robustheit des Systems erheblich.

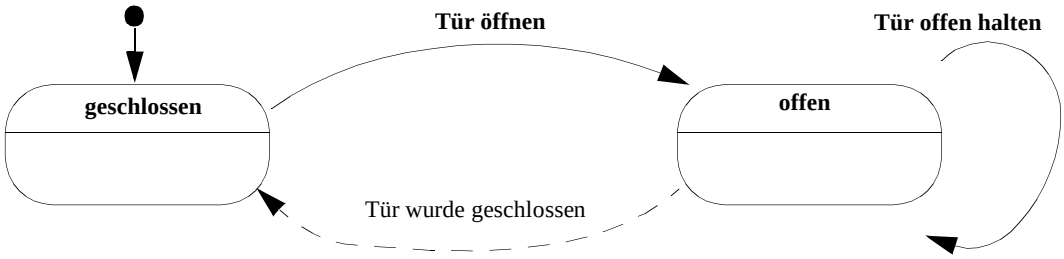


Abb. 3-7: *Statechart Diagram* mit logisch unmöglichen Zustandsübergängen

Es ist schwer, eine universelle Vorgehensweise zur Vermeidung der beschriebenen Probleme anzugeben. In den Beispielen ist es relativ leicht, die kritischen Punkte zu identifizieren. Bei komplexeren Problemstellungen sind diese nicht mehr so offensichtlich. Natürlich können Übergänge zwischen allen Zuständen vorgesehen werden. Allerdings ist diese Vorgehensweise in den meisten Fällen wegen der Erhöhung der Komplexität nicht sinnvoll. Als Anhaltspunkt können alle Zustände des Systems betrachtet werden, die von außen verändert werden können.

Das *Statechart Diagram* kann auch verwendet werden, um Nebenläufigkeit [FOW98] oder Zeitverhalten [ROM98] zu modellieren. Falls Nebenläufigkeit bzw. Zeitverhalten erst während einer späteren Aktivität der KobrA-Methode eine Rolle spielen, kann das entsprechende Artefakt um ein *Statechart Diagram* ergänzt werden.

3.1.2.4 Zusammenfassung

Es waren kleinere Modifikationen an den Artefakten *Functional Model* und *Behavioural Model* erforderlich. Als neues Artefact kann das unter 3.2 beschriebene *Experience Model* auftreten, da dieses alle Ebenen betreffen kann. Das Produktflussdiagramm (Abb. 3-8) zeigt die Zusammenhänge zwischen den Artefakten und Aktivitäten der *Komponent Specification*. Die *Decision Models* erfordern keine Anpassungen.

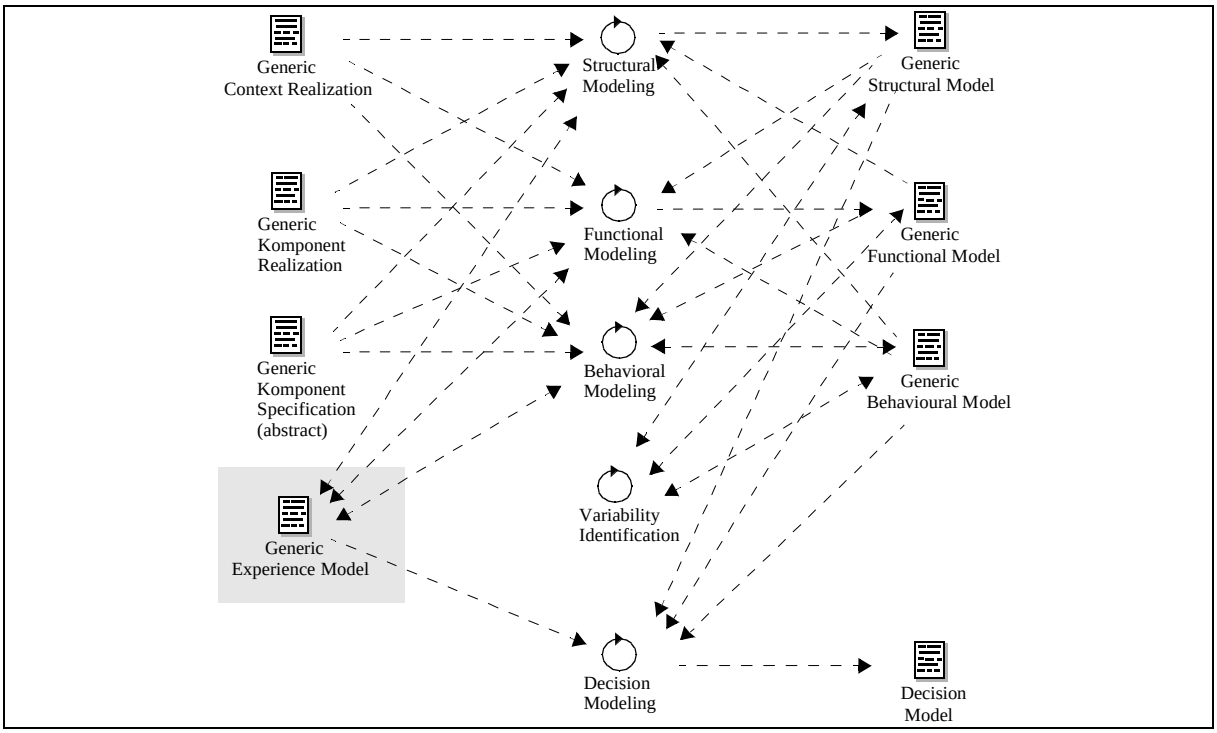


Abb. 3-8: Überarbeiteter Produktfluss der *Komponent Specification* (Original aus [ATK01])

3.1.3 Komponent Realisation

Die *Komponent Realisation* besteht aus den Aktivitäten *Structural Modeling*, *Activity Modeling*, *Interaction Modeling*, *Variability Identification*, *Decision Modeling*, *Komponent Identification* und *Komponent Containment Tree Refactoring*. Die wichtigsten erzeugten Artefakte sind *Generic Structural Model*, *Generic Activity Model*, *Generic Interaction Model* und *Decision Model*. Abb. 3-9 zeigt den überarbeiteten Produktfluss.

Der Hauptunterschied zur *Context Realization* ist, dass die *Komponent Realization* einen konkreteren Startpunkt hat, nämlich die *Komponent Specification* der Komponente. Die Modelle entstehen durch Verfeinerung der Artefakte aus der *Komponent Specification* und Erweiterung um das *Interaction Modeling*. [ATK01]

Da die Hardware auf der Ebene der *Komponent Realization* durch entsprechende Klassen abstrahiert wurde und auch keine *Hardware Components* auftreten können, sind diesbezüglich keine Anpassungen für *Structural Model*, *Activity Model*, *Interaction Model* oder *Decision Model* erforderlich. Allerdings könne andere Eigenschaften eingebetteter Systeme wie beispielsweise Threads oder Echtzeitverhalten die *Komponent Realization* betreffen. Die Modellierung von Threads wird durch *Sequence Diagrams* unterstützt [SEE00]. Das Zeitverhalten kann ebenfalls durch diesen Diagrammtyp modelliert werden. Beispielsweise durch Verwendung eines Timer Objektes, das nach festgelegten Zeitspannen die entsprechende Methode aufruft. Das *Interaction Model* der *Komponent Realization* verwendet *Collaborations Diagrams* um dynamisches Verhalten zu modellieren. Bei Bedarf kann dieses Modell um *Sequence Diagram* ergänzt werden. Zeitverhalten kann aber auch mit den *Collaborations Diagrams* realisiert werden, beispielsweise durch die Bedingung ’[Zeit abgelaufen]’.

Eingebettete Systeme verfügen häufig über eine große Zahl von Konstanten wie beispielsweise Schwellenwerte, ab denen auf ein Ereignis reagiert wird. Um diese Konstanten in der Darstellung von Attributen abzugrenzen, bietet es sich an, beim *Structural Model* den Stereotyp <<utility>> zu benutzen. Laut UML Spezifikation können durch ihn globale Variablen und Prozeduren in Form einer Klassendeklaration gruppiert werden [OMG01]. Geeignete Werte für die Konstanten werden erst später durch Messungen bzw. Experimente bestimmt.

Das unter 3.2 beschriebene *Experience Model* kann Auswirkungen auf die Aktivitäten *Structural Modeling*, *Activity Modeling*, *Interaction Modeling* und *Decision Modeling* haben bzw. von diesen beeinflusst werden.

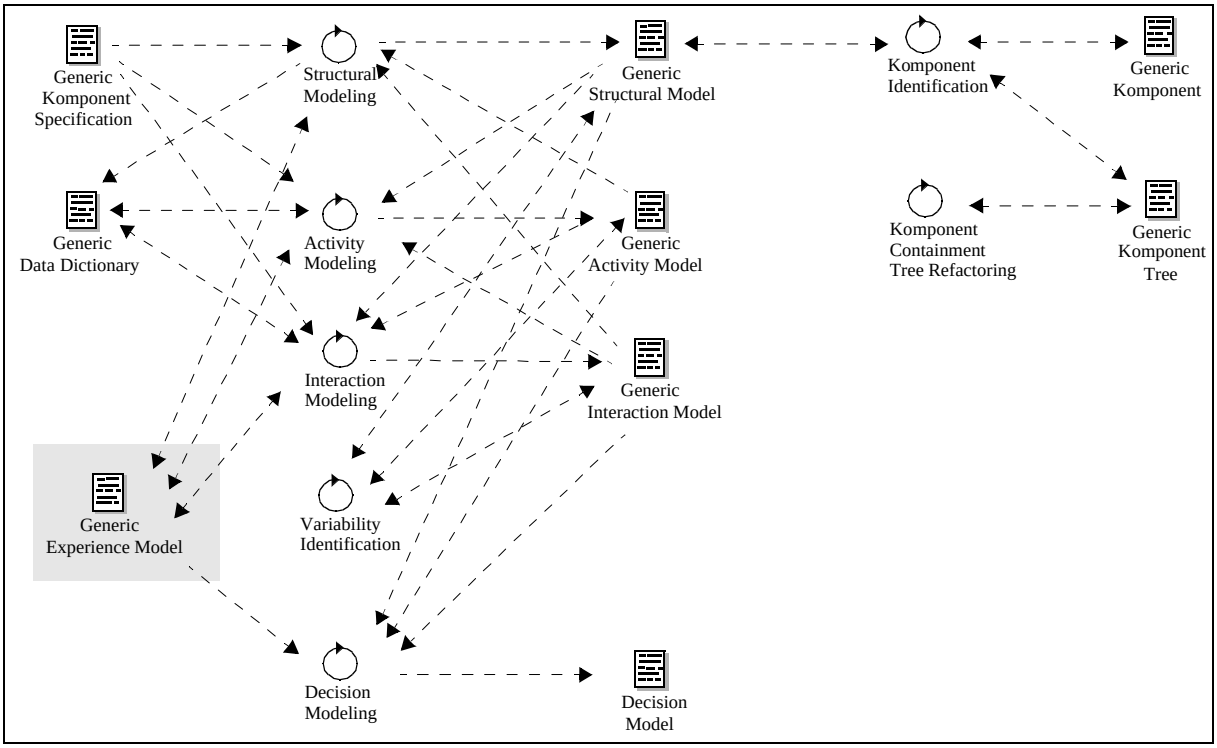


Abb. 3-9: Überarbeiteter Produktfluss der *Komponent Realization* (Original aus [ATK01])

3.1.4 Component Reuse

Auch bei eingebetteten Systemen können *Software Components* wiederverwendet werden. Allerdings ist die Auswahl an fertigen Komponenten deutlich geringer als bei anderen Systemen. Genauso sind für eingebettete Systeme nur wenige *Design Patterns* oder COTS-Komponenten (Commercial Of The Shelf) verfügbar. Wiederverwendung findet aber durch die Bibliotheken der Entwicklungsumgebung statt, die Klassen zur Abstraktion der Hardware enthalten können.

Natürlich spricht nichts dagegen, selbstentwickelte *Software Components* wiederzuverwenden. Dabei muss aber beachtet werden, dass *Software Components* für eingebettete System meistens sehr hardwarespezifisch sein können. Das bedeutet, dass Komponenten, die bei einem eingebetteten System funktionieren, bei einem anderen völlig versagen können. Daher muss für jede *Software Component* sorgfältig dokumentiert werden, welche Voraussetzungen die Hardware erfüllen muss, damit diese

Komponente eingesetzt werden kann. Eine andere Form der Wiederverwendung stellen die im *Experience Model* (vgl. 3.2) festgehaltenen Erfahrungen dar.

3.1.5 Komponent Implementation

Um eine ausführbare Komponente zu erhalten, muss diese erst in eine Form gebracht werden, die von Tools wie beispielsweise Compilern verarbeitet werden kann. Die entsprechende Aktivität heißt *Komponent Implementation.* Sie besteht aus den Aktivitäten *Flattening, Refinement, Translation, Finalization, Variability Identification* und *Decision Modeling*. Die wichtigsten erzeugten Artefakte sind *Generic Structural Mode*l, *Generic Source Code* und *Decision Model*. Abb. 3-10 zeigt den überarbeiteten Produktfluss.

Während des *Flattenings* wird ein physisches Komponenten Modell entwickelt. Dazu werden logische auf physische Komponenten abgebildet. Diese Aktivität wird auch dazu benutzt, den KobrA-Baum abzuflachen. Durch *Refinement* wird das *Structural Model* so weit verfeinert, dass es dem *Source Code* nahe kommt. *Translation* bildet die Komponente auf den Source Code ab. Dieser muss, insbesondere wenn er automatisch generiert wurde, überarbeitet werden. Die entsprechende Aktivität ist die *Finalization*.

Wie bei der *Komponent Realization* sind kaum Modifikationen an der KobrA-Methode erforderlich. Eine Besonderheit können allerdings die in 3.1.3 erwähnten Konstanten darstellen. Damit der *Source Code* erzeugt werden kann, müssen sie einen Wert haben. Hierbei gibt es verschiedene Vorgehensweisen. Es können Werte geraten werden, die anschließend beim *Testing* überprüft und angepasst werden. Eine andere Möglichkeit stellen Messungen bzw. Experimente dar. Dazu wird KobrA um die Aktivität *Measurement of Constants* erweitert (vgl. Abb. 3-10). Sie bestimmt geeignete Wertebelegungen für die Konstanten durch Messungen. Trotzdem stellt dies keine Garantie dar, dass die Komponente mit diesen Werten immer funktioniert. Äußere Störgrößen können insbesondere Sensorwerte stark verfälschen. Daher ist es sinnvoll, zur Sicherheit Toleranzen einzuplanen.

Es kann auch vorkommen, dass verschiedene Produktlinien unterschiedliche Konstanten benötigen, z. B. wenn sie in verschiedenen Einsatzumgebungen operieren. Dies läßt sich im *Structural Model* elegant durch Verwendung des Stereotyps <<utility>> modellieren. Dadurch können zwei Versionen der Konstantenbelegungen erzeugt werden, die beide mit <<variant>> versehen sind. Die Auflösung erfolgt dann über die *Decision Models*.

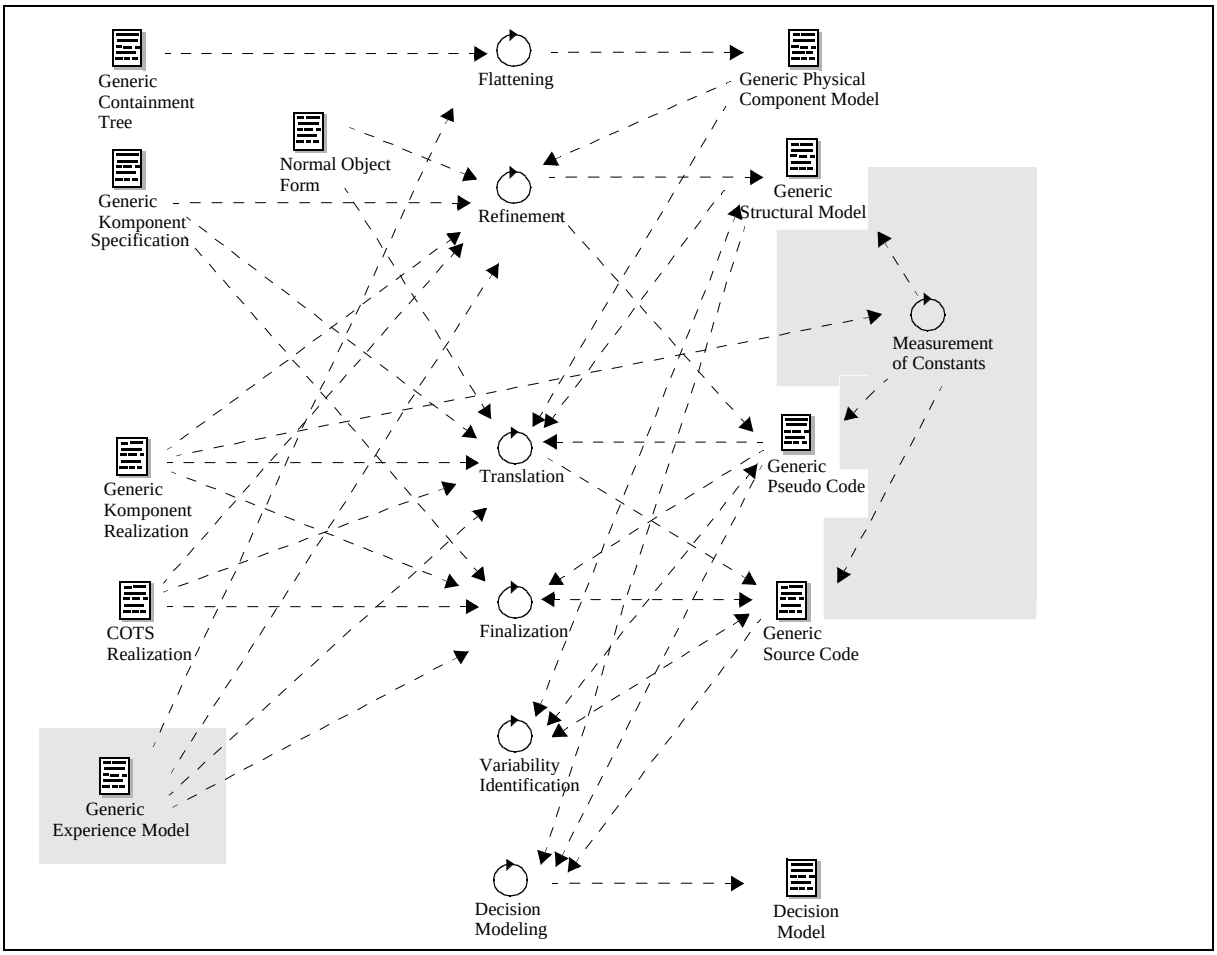


Abb. 3-10: Überarbeiteter Produktfluss der *Komponent Implementation* (Original aus [ATK01])

3.1.6 Inspection / Measurement of Structural Properties

Inspection und *Measurement of Structural Properties* sind Tätigkeiten zur Qualitätssicherung. Inspiziert werden alle größeren Artefakte. Diese werden durch qualifizierte Personen systematisch auf Fehlerfreiheit und Qualität untersucht. Die Aktivität *Inspection* bezieht sich nicht mehr direkt auf die Problemstellung, sondern beschäftigt sich mit den erzeugten Artefakten, die beispielsweise auf formale Kriterien wie Fehlerfreiheit untersucht werden. Die Eigenarten eingebetteter Systeme haben daher keinen Einfluss auf die Inspektion.

Die Aktivität *Measurement of Structural Properties* untersucht interne Qualitätsattribute wie Kopplung, Komplexität und Größe. Da sich diese ebenfalls direkt auf die Artefakte beziehen, sind keine Modifikationen dieser Aktivität erforderlich. Es kann aber zu Unterschieden in der Bewertung der Qualitätsattribute kommen. Beispielsweise ist die Größe bei einem eingebetteten System häufig eine kritische Eigenschaft. Dann kann beispielsweise eine hohe Kopplung akzeptabel sein, wenn dadurch die Größe minimiert wird. Eventuell sind spezielle Metriken für eingebettete Systeme erforderlich. Es kann aber auch sinnvoll sein, neue Qualitätsmodelle und entsprechende Metriken zu entwickeln. Beispielweise kann die Anzahl der direkten Aufrufe von Hardwarefunktionen als Maß für die Hardwareabhängigkeit der Software herangezogen werden.

3.1.7 Testing

Die Aktivität *Testing* erscheint sowohl im Produktflussdiagramm von *Framework Engineering* (Abb. 2-10) und *Application Engineering* (Abb. 3-11). Sie besteht aus den Aktivitäten *Test Suite Definition*, *Test Execution* und *Test Result Analysis*. Zum Testen wird ausführbarer Code benötigt. Wegen dieser Einschränkung kann *Testing* erst während oder nach der Implementierung stattfinden. Daher ist die *Test Suite Definition* Teil des *Framework Engineerings*, während *Test Execution* und *Test Result Analysis* zu *Application Engineering* zählen. Die Zeitdauer zwischen der Realisierung einer Komponente und der Verfügbarkeit der Testergebnisse kann sehr lang werden, wodurch der Aufwand zur Fehlerbeseitigung sehr hoch werden kann. Um frühe Inkremente zu testen, müssen daher *Stubs* oder *Proxies* erzeugt werden. [ATK01]

KobrA beschreibt den Testvorgang nur unspezifisch. So finden sich kaum Aussagen über die Durchführung von *Test Execution* und *Test Result Analysis*. Lediglich die *Test Suite Definition* ist näher beschrieben. Es wird empfohlen, dass die Test Suite *Functional*, *Structural* und *State-based Test Cases* enthalten soll. Die *Functional Test Cases* sind für die *Komponent Specification* angebracht, während *Structural Test Cases* nützlich für die *Komponent Realization* sind. Zur Ergänzung der beiden dienen die *State-based Test Cases*.

Im Buch [ATK01] wird verschwiegen, dass auch für die *Test Suite Definition* die Erstellung eines *Decision Models* sinnvoll sein kann. Da die *Test Execution* erst während des *Application Engineerings* stattfindet, können nur Testfälle überprüft werden, deren Funktionalität von der aktuellen Produktlinie unterstützt wird. Da das Buch diesbezüglich unpräzise ist, bestehen große Freiheiten für die Durchführung der *Test Suite Definition* und die Erzeugung des *Decision Models*. So können die *Functional Test Cases* bei der *Komponent Specification* und die *Structural Test Cases* bei der *Komponent Realization* aufgeführt werden. Die dortigen *Decision Models* müssen um die *Test Cases* ergänzt werden. Alternativ kann die Aufstellung der Test Suite als eigene Aktivität des *Framework Engineerings* betrachtet und mit einem *Decision Model* versehen werden, das komplette *Test Suites* für jede Produktlinie erzeugt.

Zusätzlich zu *Functional*, *Structural* und *State-based Test Cases* wird an dieser Stelle eine neue Art von Testfällen eingeführt, die *Environment-based Test Cases*.

Environment-based Test Cases

Environment-based Test Cases berücksichtigen, dass die Funktionstüchtigkeit eingebetteter Systeme sehr stark von Umgebungscharakteristika beeinflusst werden kann. Dazu können u. a. Bodenbeschaffenheit, Temperatur, Beleuchtungsverhältnisse oder Batteriezustand zählen. Tabelle 3-8 zeigt einen möglichen Aufbau der *Environment-based Test Cases*. Es werden die Einflussgröße, ihre Belegung und die betroffenen Testfälle aufgelistet. Im Extremfall muss die komplette *Test Suite* für verschiedene Belegungen dieser Einflussgrößen getestet werden müssen. Häufig betrifft eine Einflussgröße aber nur wenige Testfälle. Beispielsweise beeinflusst der Batteriezustand die Umdrehungsgeschwindigkeit und die Kraft der Motoren. Licht- oder Tastsensoren werden dagegen nicht beeinflusst.

Die *Environment-based Test Cases* bieten auch die Möglichkeit, Features verschiedener Produktlinien zu berücksichtigen, welche die Einsatzumgebung betreffen. Beispielsweise können zwei Versionen eines baugleichen Fahrzeuge existieren. Das eine soll auf Straßen, das andere auf Feldwegen operieren.

Ohne die *Environment-based Test Cases* wird nicht deutlich, dass es zwischen dem Einsatz auf Straßen und Feldwegen Unterschiede gibt und beide Varianten getestet werden müssen. Ein Fahrzeug, dass nur auf der Straße getestet wurde, kann auf Feldwegen völlig versagen.

ID	Einflussgröße	Wert	Betroffene Testfälle
1.	Batteriezustand	voll	Functional Test Cases: - ... Structural Test Cases: - ... State-based Test Cases: - ...
2.	Batteriezustand	halbvoll	
⋮	⋮	⋮	⋮

Tabelle 3-8: *Environment-based Test Cases*

Tests bei eingebetteten Systemen:

Neben der Ergänzung der *Test Suite* um die *Environment-based Test Cases* fallen einige weitere Besonderheiten beim Testen eingebetteter System auf. So kann mangelnde Werkzeugunterstützung, wie das Fehlen eines Debuggers oder Simulators, Probleme bereiten. Viele spezielle Sprachen für eingebettete Systeme sind nur auf diesem System ausführbar. Dann muss direkt auf dem eingebetteten System getestet und dazu eventuell eine komplexe Systemumgebung aufgebaut werden. Wenn dabei mit realen Sensorwerten gearbeitet werden muss, kann es Probleme bereiten ,die Sensoren so anzuregen, dass sie die für den Testfall benötigten Werte liefern.

Ein weiteres Problem bei eingebetteten Systemen kann das Fehlen einer graphischen Ausgabemöglichkeit sein. Dadurch wird das Auslesen von Debug-Infos erschwert. Auch muss eine zusätzliche Problemstellung berücksichtigt werden. So kann ein Programm völlig korrekt sein, aber beispielsweise durch fehlerhafte Messwerte versagen. Dies muss durch die Software abgefangen werden. Es ist aber auch möglich, dass für ein eingebettetes System auf Grund der Störgrößen nicht garantiert werden kann, dass es immer funktioniert. In solchen Fällen können durch wiederholte Testausführungen Aussagen getroffen werden, wie beispielsweise „Das System funktioniert in 99,9% aller Fälle“. Das korrekte Funktionieren eines eingebetteten Systems kann auch so definiert werden, dass es solange bestimmte Toleranzen eingehalten werden, seine Aufgabe korrekt erfüllt. Bei einem Eierkocher ist es beispielsweise egal, ob das Ei 5 min oder 5 min und 3 sec kocht.

Testing bietet die Möglichkeit festzustellen, ob der Speicherbedarf, eine kritische Größe eingebetteter Systeme, akzeptabel ist. Darüber sind vorher höchstens Abschätzungen möglich. Durch geeignete Messungen kann beispielsweise der durchschnittliche Speicherbedarf für Klassen, Methoden oder Variablen festgestellt werden. Hierbei ist aber keine Aussage über den tatsächlich benötigten Speicherbedarf, der auch durch Instantiierung weiterer Objekte innerhalb des Programmes beeinflusst wird, möglich.

3.2 Experience Model

Eine Erweiterung von KobrA ist das *Experience Model*. Dieses erlaubt die Berücksichtigung der während aller Phasen der Softwareentwicklung gesammelten Erfahrungen. Die bereits zu Beginn vorliegende Dokumentation der verwendeten Hard- und Software kann ebenfalls wichtige Informationen bzw. Erfahrungen enthalten. Beispielsweise enthält die Dokumentation eingebetteter Systeme häufig Hinweise, wie Speicher und Energie gespart werden können. Das *Experience Model* soll keine vollwertige Erfahrungsdatenbank ersetzen, sondern die für dieses Projekt wichtigen Informationen an zentraler Stelle verwalten und allen Komponenten jederzeit zur Verfügung stellen. Natürlich spricht nichts dagegen, die Erfahrungen zusätzlich in eine Erfahrungsdatenbank zu übertragen. Die gesammelten Erfahrungen und Regeln führen zu einer Art Richtliniensammlung (lessons learned). So kann beispielsweise die Erfahrung, dass der Arbeitsspeicher zu klein ist, zu der Richtlinie führen, den Datentyp long möglichst zu vermeiden. Diese Richtlinien werden in Tabellenform aufgeschrieben (vgl. Tabelle 3-9). Dabei wird auch notiert, ob es sich um eine zwingende Forderung oder nur um eine Empfehlung handelt. Da die Richtlinien verschiedene Abstraktionsebenen beeinflussen können, werden in die Tabelle die betroffenen Ebenen, z. B. die *Komponent Realization*, eingetragen. Aufgeführt wird auch die Quelle der Erfahrung, d. h. wo diese gesammelt und dokumentiert wurde, sowie die daraus hervorgegangene Richtlinie.

Viele Systeme haben die Eigenschaft, dass sie aus vielen, sich stark unterscheidenden Subsystemen bzw. Komponenten zusammengesetzt sind. Um dem Rechnung zu tragen, wird in der Tabelle notiert, für welche Komponenten die Richtlinien gelten.

Die Erzeugung des *Experience Models* ist keine eigenständige Aktivität. Sie wird parallel zur Entwicklung durchgeführt und kann das *Experience Models* während allen Aktivitäten des KobrA-Prozesses ergänzen bzw. überarbeiten. Dadurch steht das entstandene Artefakt allen Aktivitäten zur Verfügung und kann auch komponentenübergreifende Erfahrungen erfassen. Wenn eine Erfahrung nicht für alle Systeme einer Produktlinie gilt, wird sie mit dem Stereotyp <<variant>> versehen. In diesem Fall ist es erforderlich, für das *Experience Model* ein *Decision Model* aufzustellen. Die Verwendung des *Experience Models* ist nicht auf eingebettete Systeme beschränkt. Die Aufstellung kann auch für andere Systeme sinnvoll sein.

Betroffene Komponenten	Erfahrungen	
Alle	**Erfahrung 1**	
	Stichwort	Worum geht es?
	Richtlinie	Kurze Beschreibung der Erfahrung. Welche Richtlinien werden daraus abgeleitet?
	Quelle	Woher stammt die Erfahrung?
	Art	Ist es eine Vorschrift oder Empfehlung?
	Betrifft	Welche Ebenen von KobrA sind betroffen?
	Erfahrung 2	
	⋮	⋮
Komponente X	**Erfahrung X.1**	
	⋮	⋮
	Erfahrung X.2	
	⋮	⋮
⋮	⋮	⋮

Tabelle 3-9: *Experience Model*

3.3 Application Engineering

Aufgabe des *Application Engineering* ist es, aus den generischen Artefakten des *Framework Engineering* spezifische Artefakte zu erzeugen. Die generischen Artefakte des Frameworks enthalten die Funktionalität aller Produktlinien. Durch Instantiierung des Frameworks werden die Artefakte für eine spezielle Produktlinie generiert. Gestartet wird dazu mit der Instantiierung der *Generic Context Realization* zu einer *Specific Context Realization*. Danach wird die Instantiierung rekursiv für alle Komponenten weitergeführt. Die Instantiierung erfolgt durch Auflösung der *Decision Models*.

Teilweise sind die Aktivitäten des *Application Engineerings* identisch sind mit denen des *Framework Engineerings*. Der wesentliche Unterschied ist, dass die generischen Artefakte durch Auflösung der Entscheidungsmodelle zu spezifischen geworden sind. Daher gelten für diese Aktivitäten die in Abschnitt 3.1 beschriebenen Modifikationen. Die dort neu eingeführten Modelle müssen während des *Application Engineerings* ebenfalls instantiiert werden. Weitere Anpassungen sind nicht erforderlich. Dies gilt für *Application Context Realization*, *Application Komponent Specification*, *Application Komponent Realization* und *Application Komponent Implementation*. *Inspection* und *Measurement of Structural Properties* erfordern ebenfalls keine Anpassungen, da diese Aktivitäten sich nicht von ihren Gegenstücken beim *Framework Engineering* unterscheiden.

Die Aktivität *Testing* ist auf *Framework Engineering* und *Application Engineering* aufgeteilt. So ist die *Test Suite Definition* Teil des *Framework Engineering*, während *Test Execution* und *Test Result Analy-*

sis zu *Application Engineering* gehört. Trotzdem gehören die Teilaktivitäten des *Testing*s zusammen. Die Besonderheiten, die beim Testen eingebetteter Systeme auftreten können, sind bereits während des *Framework Engineerings* unter 3.1.7 beschrieben worden.

Construction, *Component Building* und *Releasing* hängen sehr stark von den verwendeten Technologien und Werkzeugen ab. Um den Einsatz von KobrA nicht einzuschränken, sind diese Aktivitäten im Buch [ATK01] weniger streng beschrieben. Damit können sie an die Bedürfnisse einer Software-Organisation angepasst werden. Dies gilt auch für den Einsatz bei eingebetteten Systemen. Eine allgemeine Modifikation, die für alle eingebetteten Systeme gilt, ist nicht möglich.

Die neue Aktivität *Application Experience Modeling* besteht aus der Instantiierung des *Experience Models* durch Auflösung des entsprechenden *Decision Models*. Dadurch wird das Artefakt *Specific Experience Model* generiert, welches nur für die aktuelle Produktlinie gilt. Die darin gesammelten Erfahrungen können *Application Context Realization*, *Application Komponent Specification*, *Application Komponent Realization* und *Application Komponent Implementation* beeinflussen.

Da für das *Experience Model* nicht unbedingt ein *Decision Model* erzeugt werden muss (vgl. 3.2), kann diese Aktivität auch entfallen. In diesem Fall wird das *Generic Experience Model* verwendet.

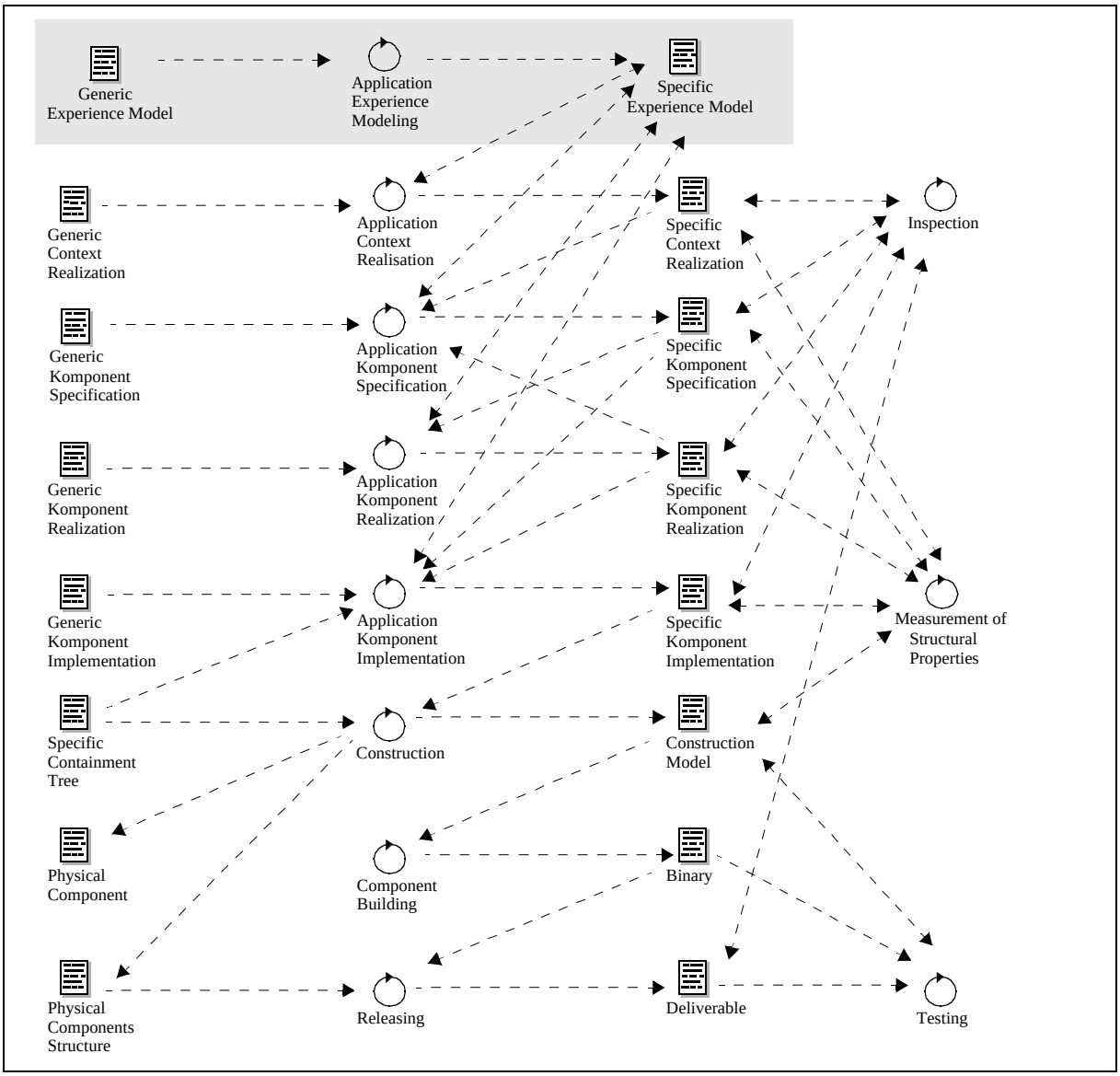
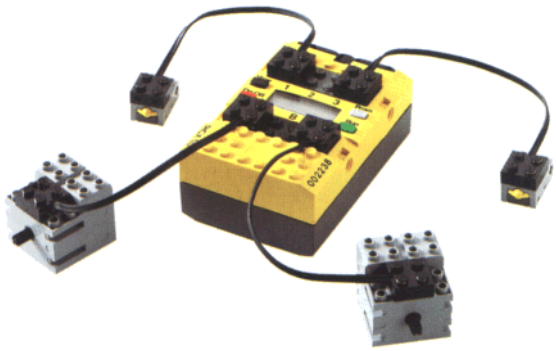


Abb. 3-11: Überarbeiteter Produktfluss des *Application Engineerings* (Original aus [ATK01])

Kapitel 4

Der Legoroboter

4.1 LEGO MINDSTORMS



MINDSTORMS [LEGO] ist der Oberbegriff für eine Reihe von Baukästen und Bausteinen der Firma LEGO zur Konstruktion von Robotern. Der erste Baukasten, das *Robotics Invention System* (RIS), wird seit Herbst 1998 angeboten. Seitdem wurde die Produktlinie MINDSTORMS um mehrere Ergänzungsbaukästen erweitert. Im Mittelpunkt steht der RCX-Baustein (*Robotics Command Center*). Dieser ist ein in einen LEGO-Stein integrierter Mini-Computer mit drei Ein- und drei Ausgängen zum Anschluss von Motoren bzw. Sensoren (Abb. 4-1). Der RCX-Baustein kann einzeln erworben werden oder als Teil des RIS, das neben einer großen Zahl von LEGO-Standardbausteinen zwei Tastsensoren, einen Lichtsensor, zwei Motoren und die Software zur Programmierung des RCX-Bausteins enthält. Der Begriff RIS ist nicht eindeutig definiert und wird in verschiedenen Zusammenhängen gebraucht, z. B. für den kompletten LEGO-Baukasten, für die grafische Programmiersprache oder für die Entwicklungsumgebung inklusive Firmware. Um Verwechslungen zu vermeiden, bezeichnet RIS in diesem Dokument nur den LEGO-Kasten. Programmiert wird mit der Entwicklungsumgebung *RCX Software Developers Kit* (RCX SDK), die RCX-Code erzeugt.

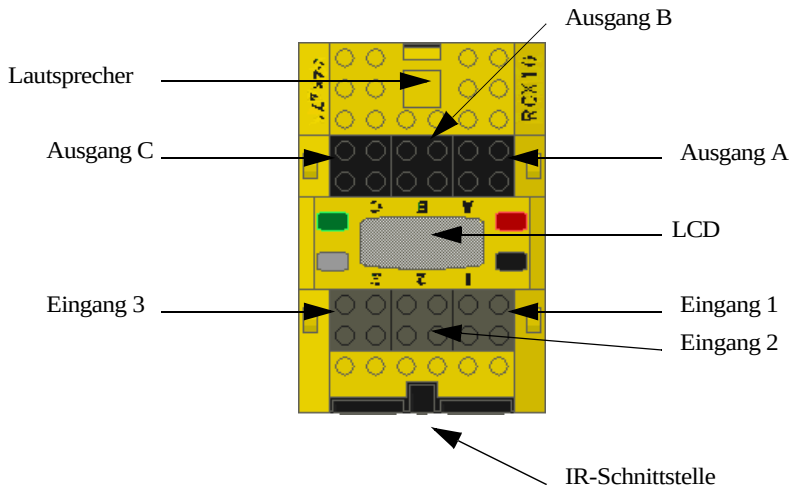


Abb. 4-1: Der RCX-Baustein

Das Gehirn des RCX-Bausteins bildet ein mit 16 Mhz getakteter Hitachi H8300 8 Bit Prozessor. Obwohl dieser mit seinem 16 Bit Adressbus 64 KB adressieren kann, ist der RCX-Baustein nur mit einem 32 KB großen RAM ausgestattet. Die Stromversorgung erfolgt über 6 AA Mignon Batterien, bzw. Akkus (7-9V). Frühere Versionen des RCX-Bausteins konnten auch mit einem Netzteil betrieben werden. Für den Anschluss von Aktoren und Sensoren stehen jeweils drei analoge Ausgänge bzw. Eingänge zur Verfügung. Über den integrierten Lautsprecher können akustische Signale ausgegeben werden. Zur grafischen Ausgabe dient ein LCD-Display mit 5 Stellen und mehreren Sondersymbolen. Über die 4 Druckknöpfe können der RCX-Baustein ein- bzw. ausgeschaltet, das Programm gestartet, zwischen mehreren Programmen hin- und hergeschaltet und Sensorenwerte ausgelesen werden. Intern ist der RCX-Baustein mit 4 unabhängigen Timern, die eine Auflösung von 100 ms haben, ausgestattet. Die IR-Schnittstelle des RCX-Bausteins arbeitet mit einer Frequenz von 27 MHz und ist für verschiedene Aufgaben zuständig. So erfolgt über sie der Download der Programme auf den RCX-Baustein. Über die IR-Schnittstelle kann aber auch die Fernsteuerung des Roboters durch den PC oder eine Fernbedienung erfolgen. Weiterhin können sich die RCX-Bausteine auch gegenseitig Nachrichten schicken bzw. empfangen. Dadurch ist die Kombination mehrerer RCX-Bausteine, die zusammenarbeiten möglich. Die IR-Schnittstelle kann Nachrichten von einem Byte Größe empfangen bzw. senden. Damit sind 256 verschiedene Nachrichten möglich. [SCH01], [GAS00]

Um komplexe Verhaltensweisen zu ermöglichen, existieren eine Reihe von Tricks, um die Beschränkung auf 3 Eingänge zu umgehen [GAS], [KNU99]. Eine Möglichkeit, bis zu 6 Sensoren anzuschließen ohne einen Lötkolben zur Hand zu nehmen, wird in 4.3.2 beschrieben.

Die Elemente des RIS können mit anderen LEGO-Bausteinen kombiniert werden. Daneben bietet LEGO eine Reihe spezieller Bausteine für den Anschluss an das RCX an (vgl. Tabelle 3). Es ist aber auch möglich, Sensoren bzw. Aktoren, die nicht von LEGO stammen, an den RCX-Baustein anzuschließen.

Der Tastsensor kann beispielsweise zur Erkennung von Kollisionen verwendet werden. Er liefert die booleschen Werte *true* (gedrückt) bzw. *false* (nicht gedrückt).	
Der Rotationssensor misst Umdrehungen. Dabei verfügt er über eine Auflösung von 16 Impulsen pro Umdrehung.	
Der Lichtsensor gibt die gemessene Helligkeit in Prozent (0% ... 100%) an. Beim Betreiben des Lichtsensors leuchtet eine rote Leuchtdiode auf. Damit wird es möglich, bei kurzen Entfernungen eine dunkle Linie auf hellem Untergrund zu erkennen.	
Über den Temperatursensor kann die Temperatur gemessen werden. Der Temperaturbereich reicht von -20 bis +50 Grad Celsius.	
Der Motor ist mit einem integrierten Getriebe ausgestattet. Dadurch verfügt er über genügend Kraft, um Antriebsräder direkt anzutreiben.	
Die LEGO-Lampe kann an die Ausgänge des RCX-Bausteins angeschlossen werden.	
Die Kamera ist über ein Kabel mit dem Computer verbunden. Die Kameradaten werden auf dem PC ausgewertet. Dieser kann den Roboter über Nachrichten entsprechend fernsteuern.	
Die Infrarot-Fernbedienung kann Nachrichten an den RCX-Baustein schicken bzw. den Roboter komplett fernsteuern.	

Tabelle 3: Verfügbare Erweiterungen zum Anschluss an den RCX-Baustein

Das RIS wird mit dem *RCX Software Developers Kit (SDK)* Version 1.5 ausgeliefert. Dieses enthält sowohl die Firmware des RCX-Bausteins als auch eine grafische Programmieroberfläche, um RCX-Code zu erzeugen. Programmiert wird durch das Zusammenstecken von Blöcken, die *Commands*, *Sensor Watcher* oder *Controller* darstellen können (Abb. 4-2). Einige Blöcke erlauben die Einstellung von Parametern, beispielsweise wie schnell der Motor fahren soll. Der Nachfolger RCX SDK 2.0 steht bei [LEGO] als Betaversion zum Download bereit. Neben einer neuen Firmware wird auch die Programmiersprache LEGO „Mind Script“ mitgeliefert. Diese wird als Implementierungssprache für die grafische Entwicklungsumgebung benutzt. Es ist aber genauso möglich, den Roboter direkt in LEGO Mind Script zu programmieren.

Zum Lieferumfang des RCX SDK gehört ein Tutorial, das auch jungen Benutzern den Einstieg in die Programmierung des RCX-Bausteins ermöglichen soll. In der einfachen Programmierbarkeit lag auch das Hauptaugenmerk bei der Entwicklung des RCX SDK. Für anspruchsvollere Programmieraufgaben ist es nur bedingt geeignet. Dass die grafische Darstellung bei großen Programmen unübersichtlich wird und einen gut ausgestatteten PC verlangt, mag noch akzeptabel sein. Von größerer Bedeutung ist,

dass es keine Variablen gibt und keine benutzerdefinierten Tasks erzeugt werden können. Die Definition von Unterprogrammen (My Commands) ist möglich. Allerdings können diese keine Unterprogramme bzw. sich selbst aufrufen. Damit kann keine Rekursion verwendet werden.

Da neben diesen Einschränkungen auch die objektorientierte Programmierung nicht unterstützt wird, müssen für den Einsatz in dieser Diplomarbeit Alternativen herangezogen werden. Neben den Programmiersprachen von LEGO sind für das RCX spezielle Versionen praktisch aller Sprachen erhältlich. Einige wie Visual Basic und NQC (*Not Quite C*) verwenden die original LEGO-Firmware. Andere wie LegOS und TinyVM überschreiben die Firmware durch eine Eigenentwicklung. LegOS ermöglicht die Programmierung in C, C++ und Assembler. TinyVM und leJOS implementieren eine abgespeckte Java Virtual Maschine und werden in einer Untermenge von Java programmiert.

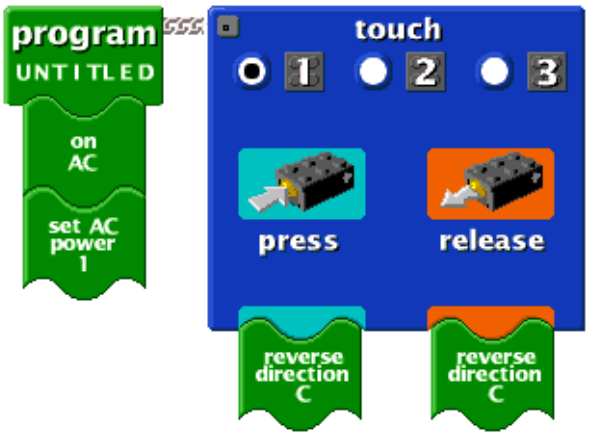


Abb. 4-2: Beispielprogramm

Weitere Informationen zum RIS und den verfügbaren Bausteinen sind auf der LEGO MINDSTORMS Homepage im Internet verfügbar [LEGO]. Leider werden der RCX-Baustein und die Sensoren nur oberflächlich beschrieben. Ein guter Startpunkt für die Suche nach tiefergehenden Details ist die Seite von Michael Gasperi [GAS]. Dort werden u. a. die Funktionsweise der Sensoren erklärt und Tipps für Eigenentwicklungen gegeben.

4.2 TinyVM/leJOS

TinyVM [TVMa] ist eine kleine Java Virtual Maschine, welche die LEGO-Firmware im RCX überschreibt. Entwickelt wurde sie ursprünglich von Jose Solorzano. Das Hauptziel bei ihrer Entwicklung war ein möglichst geringer Speicherbedarf; so benötigt sie weniger als 10 KB. Die Firmware kann von den 32 KB RAM des RCX nur 28 KB nutzen, womit für die Programmierung ungefähr 18 KB zur Verfügung stehen. Der geringe Speicherbedarf wurde dadurch möglich, dass nur die Packages java.lang und java.util (und diese auch nur teilweise) implementiert wurden, was unter anderem folgende Einschränkungen zur Folge hat:

- Der Datentyp float wurde nicht implementiert
- keine String Konstanten

- keine komplexen mathematischen Methoden wie sin, cos, tan, ...

Zu TinyVM gibt es eine Erweiterung,, deren Fokus mehr auf einer möglichst vollständigen Java-Implementierung liegt: leJOS [LEJOS]. Diese behebt die obigen Einschränkungen, benötigt dafür aber 17 KB Speicher. leJOS bietet auch flexiblere Möglichkeiten auf die Aktoren bzw. Sensoren zuzugreifen. Außerdem ist das Interface TimerListener implementiert, mit dem es möglich ist, auf das Verstreichen einer bestimmten Zeitdauer zu reagieren. Auch leJOS realisiert keine vollständige Java Implementierung, was einige Nachteile zur Folge hat, die auch für TinyVM gelten.

Das sowohl leJOS als auch TinyVM das switch Statement nicht kennen, beeinflusst nur die Übersichtlichkeit des Source Codes. Gravierender ist, dass beide keine *Garbage Collection* implementieren, wodurch besondere Aufmerksamkeit beim Umgang mit Variablen, Arrays und Rekursion erforderlich ist. TinyVM und leJOS ermöglichen die Verwendung von *Threads*. Allerdings können diesen keine Prioritäten zugewiesen werden.

Um auf die Hardware des RCX zuzugreifen, enthält TinyVM das Package tinyvm.rcx bzw. leJOS das Package josx.platform.rcx, in dem native Methoden für den Zugriff auf die Hardware bereitgestellt werden. So stellen die Klassen Sensor und Motor Abstraktionen der Ein- und Ausgänge des RCX-Bausteins dar. Beispielsweise wird mit Sensor.S1.readValue() der an Eingang 1 anliegende Signalpegel in Prozent ausgelesen. Weitere wichtige Klassen sind Serial für die Kommunikation über die IR-Schnittstelle und System für das Auslesen der Systemzeit. TextLCD stellt Methoden für die Textausgabe auf dem LCD-Display bereit und über Sound können Töne ausgegeben werden. Eine ausführliche Dokumentation befindet sich unter [TVMb], [LEJOSa].

Zunächst wird der Roboter mit leJOS programmiert. Wenn sich herausstellt, dass es zu Speicherproblemen kommt, wird TinyVM verwendet. Abb. 4-3 zeigt den Aufbau von leJOS. Auf der System-Ebene steht die Hardware des RCX-Bausteins. Auf der mittleren Ebene stellt LEGO im ROM des RCX-Bausteins native Methoden zur Verfügung. Auf diese kann die Java Virtual Machine über librcx zugreifen. Das API gehört zur Applikations-Ebene, da diese Klassen je nach Bedarf an die eigentliche Applikation gebunden werden. An oberster Stelle steht das vom Benutzer programmierte Programm zur Steuerung des RCX-Bausteins. [TVMc], [TVMd]

Zum Testen des Roboters und zum Download der Programme werden die Tools RCX-Direct-Mode und RCX-Download verwendet, die für leJOS von Tim Rinkens programmiert wurden [RIN01].

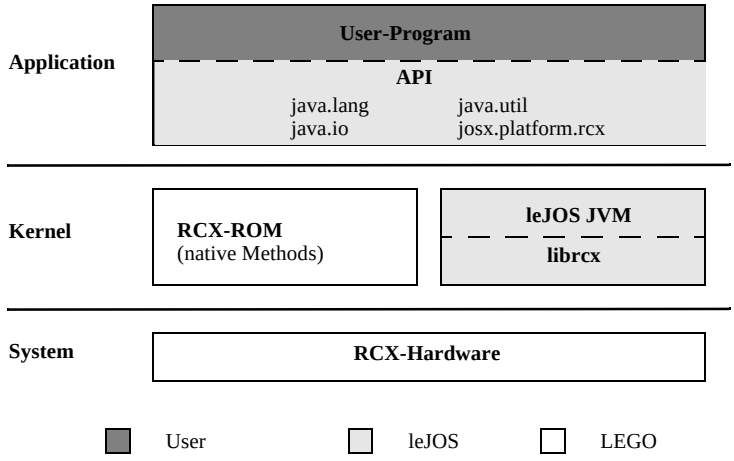


Abb. 4-3: Funktionsweise von leJOS

4.2.1 TinyVM/leJOS und Echtzeit

TinyVM unterstützt zwar Threads, bietet aber nicht die Möglichkeit, diesen Prioritäten zuzuweisen. Die Weiterschaltung von einem Thread zum nächsten wird durch ein einfaches, preemptives multi-threading Schema realisiert. Jedes Thread-Objekt hat einen Pointer, der auf das nächste in einer globalen Thread-Liste zeigt. Der letzte Thread in dieser Liste zeigt wieder auf den ersten. Der Scheduler (vgl. Abb. 4-4) besteht aus einer einzigen Funktion: switch_thread. Jedem Thread wird die Ausführung von 128 Opcodes ermöglicht, bevor zum nächsten weitergeschaltet wird. [TVMc]. Für ein hartes Echtzeitsystem ist die Vergabe von Prioritäten für die einzelnen Threads erforderlich, um eine bestimmte Reaktionszeit zu garantieren. Der Prozessor des RCX arbeitet mit einem Takt von 16 MHz. Unter der Annahme, dass für die Abarbeitung eines Opcodes im Durchschnitt 4 Takte erforderlich sind, ergibt sich für jeden Thread eine maximale Zeit von 32 ms. TinyVM erlaubt bis zu 255 Threads. Damit beträgt die Reaktionszeit maximal 8 Sekunden. Durch Reduktion der Anzahl der Threads kann diese drastisch verbessert werden. Gemäß der Definition eines Echtzeitsystems aus 2.1 hängt die zu garantierende Reaktionszeit von der Art der Anwendung ab. Der LEGO-Roboter kann mangels geeigneter Sensoren (beispielsweise Ultraschall) Hindernisse nur durch Berührungen erkennen. Eine Reaktionszeit von 1 Sekunde ist damit ausreichend, d. h. es dürfen etwa 30 Threads erzeugt werden, um diese zu garantieren. Notfalls kann die Anzahl der auszuführenden Opcodes bis zur Weiterschaltung angepasst werden. Bei der Bestimmung der Threadanzahl ist zu beachten, dass jedes Listener-Objekt beispielsweise SensorListener einen eigenen Thread erzeugt. Unter diesen Voraussetzungen kann der in 4.3 vorgestellte Roboter als ein Echtzeitsystem angesehen werden.

Bei leJOS funktioniert die Verarbeitung der Threads genauso wie bei TinyVM. Die Existenz des Interfaces TimerListener kann aber zu der Annahme verleiten, dass bestimmte Reaktionszeiten garantierbar sind. So wird nach Verstreichen einer bestimmten Zeitdauer die Klasse, die das Interface implementiert, zwar benachrichtigt, dies geschieht aber erst wenn der Thread, in dem sie enthalten ist, aktiv wird. Daher gelten die für TinyVM gemachten Aussagen auch bei Einsatz von leJOS.

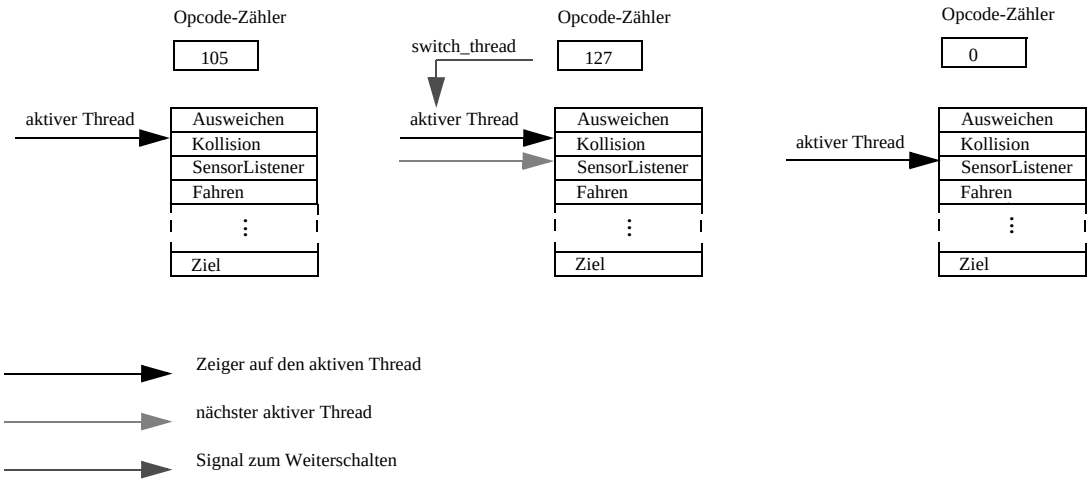


Abb. 4-4: Prinzipielle Funktionsweise des Schedulers von TinyVM

4.3 Beschreibung des Roboters und seiner Einsatzumgebung

Ein Teil der Aufgabenstellung ist angelehnt an das Robotikpraktikum an der Universität Kaiserslautern [KAS00]. Ziel des Praktikums war einen Roboter so zu programmieren, dass er in der Lage ist, Coladosen einzusammeln und in ein Ziel zu transportieren. Dabei sollten Kollisionen mit Hindernissen vermieden werden.

Da LEGO für den RCX-Baustein nur Tastsensor, Lichtsensor, Rotationssensor und Temperatursensor anbietet und der RCX-Baustein nur 3 Eingänge hat, kann die Aufgabenstellung nicht eins zu eins übernommen werden. Um ein Objekt zu lokalisieren wurde im Praktikum ein Entfernungsmesser verwendet. Dieser Sensortyp wird von LEGO nicht angeboten. Der einzige LEGO-Sensor, der sich ansatzweise zum Auffinden von Zielobjekten eignet, ist der Lichtsensor. Dies funktioniert aber nur auf kurze Entfernungen und ist anfällig gegen Umgebungslicht. Ergebnis des in diesem Zusammenhang durchgeführten Experiments aus Anhang B.3.1 es ist, dass eine wesentliche Verbesserung der Reichweite und der Zuverlässigkeit durch die Beleuchtung der Zielobjekte erreicht werden kann.

Die Ereignisse in der Einsatzumgebung treten wiederholt oder dauerhaft auf. Beispielsweise löst bei Kontakt mit einem Hindernis der Tastsensor solange aus, bis der Roboter zurückgefahren ist. Dabei spielt es eine untergeordnete Rolle, ob nach einer Millisekunde oder einer Sekunde reagiert wird. Da es keinen Sinn macht, die Motoren laufen zu lassen, wenn ein Hindernis im Weg steht, darf die Reaktionszeit aber nicht zu groß gewählt werden. Die Größe der gewünschten Reaktionszeit hängt stark von der jeweiligen Anwendung ab. Unter der Voraussetzung, dass für den LEGO-Roboter Reaktionszeiten im Sekundenbereich akzeptabel sind, kann dieser auch als hartes Echtzeitsystem angesehen werden.

Um die Eignung von KobrA für komponentenbasierte Softwareentwicklung und für Product Line Engineering zu demonstrieren, sind mehrere Aufgabenstellungen erforderlich. In Abschnitt 4.3.1 werden daher zwei Aufgabenstellungen, die in wesentlichen Teilen übereinstimmen, informell skizziert. Die Verwaltung und Organisation von unterschiedlichen Ausstattungsvarianten oder Hardwareversionen des Roboters und der zugehörigen Softwarekomponenten sind aufwändig und können zu Problemen

führen. Aus diesem Grund wird ein fertiger Roboter mit festen Ausstattungsmerkmalen benutzt, der für alle Aufgabenstellungen geeignet ist. Abschnitt 4.3.2 beschreibt den Aufbau dieses Roboters und seine Sensor- und Aktorausstattung. Es kann vorkommen, dass für die Erledigung einer Aufgabe nicht unbedingt alle Aktoren oder Sensoren benötigt werden. In Abschnitt 4.3.3 werden die zur Erfüllung der Aufgaben benötigten Verhaltensweisen des Roboters kurz skizziert. Eine komplette Anforderungsanalyse befindet sich in Anhang A als Teil der Systemdokumentation. Eine Bauanleitung für den Roboter ist in Anhang B zu finden.

4.3.1 Einsatzumgebung und Aufgabenstellung

Die Einsatzumgebung (Abb. 4-5) ist 1 Meter mal 1 Meter groß. Begrenzt wird das Feld durch Wände, die zum einen Umgebungslicht abschirmen und zum anderen verhindern, dass der Roboter sein Einsatzgebiet verlassen kann. An einer Wand befindet sich das Zielgebiet. Dieses ist durch eine Lampe gekennzeichnet. Um durch den besseren Kontrast die Lokalisation des Zielgebiets bzw. der Zielobjekte aus Aufgabenstellung A zu erleichtern, sollen die Wände möglichst dunkel sein. Die Einsatzumgebung kann mit verschiedenen Bodenbelägen ausgestattet sein.

Aufgabenstellung A:

Die Einsatzumgebung befindet sich auf einer glatten Oberfläche, beispielsweise einer Tischplatte. Im Feld befinden sich runde beleuchtete Zielobjekte (Abb. 4-6). Aufgabe des Roboters ist es, diese einzusammeln und in das durch eine Lampe gekennzeichnete Zielgebiet zu bringen. Damit es nicht zu Verwechslungen zwischen Zielgebiet und Zielobjekt kommt, darf die Lampe des Zielgebietes erst eingeschaltet werden, wenn der Roboter ein Zielobjekt eingesammelt hat. Denkbar ist, dass der Roboter in einer späteren Ausbaustufe die Lampe durch ein IR-Signal selber ein- und ausschaltet.

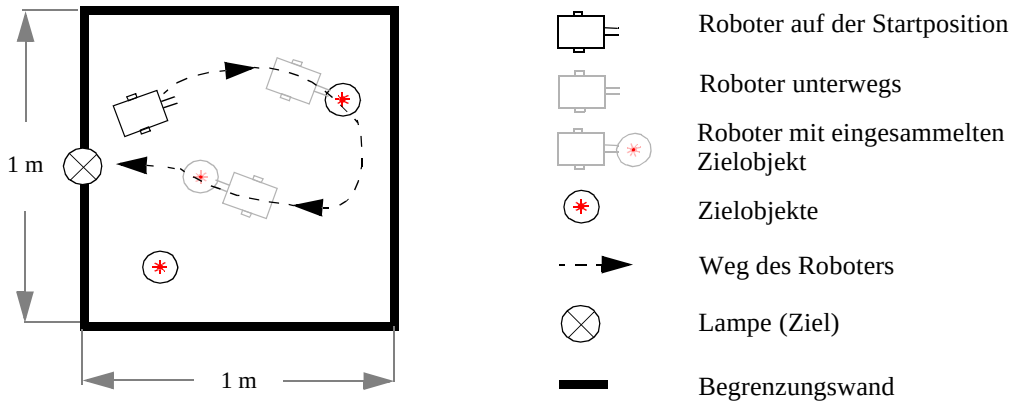


Abb. 4-5: Einsatzumgebung für Aufgabenstellung A

Entsprechend dem Ergebnis des in Anhang B.3.1 durchgeführten Experiments werden beleuchtete Zielobjekte verwendet (Abb. 4-6). Ihr Aufbau wird in Anhang B.3.2 beschrieben. Nachdem ein Zielobjekt eingesammelt wurde, muss ihre Beleuchtung ausgeschaltet werden, um die Lokalisation des Ziels nicht zu stören.

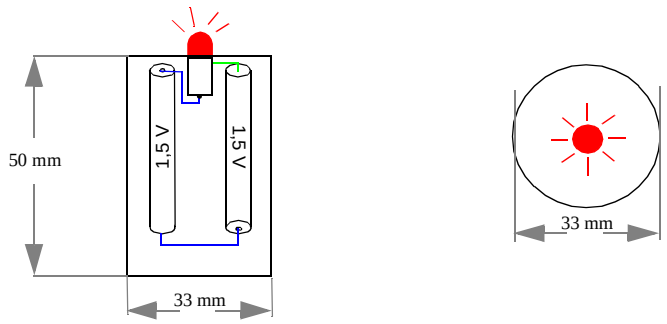


Abb. 4-6: Beleuchtung der Zielobjekte

Aufgabenstellung B:

Aufgabe des Roboters ist es, einen Weg zum Ziel zu finden und durch Einschalten seiner Lampe zu signalisieren, dass er angekommen ist. Als Oberfläche kommt ein glatter Teppichboden zum Einsatz.

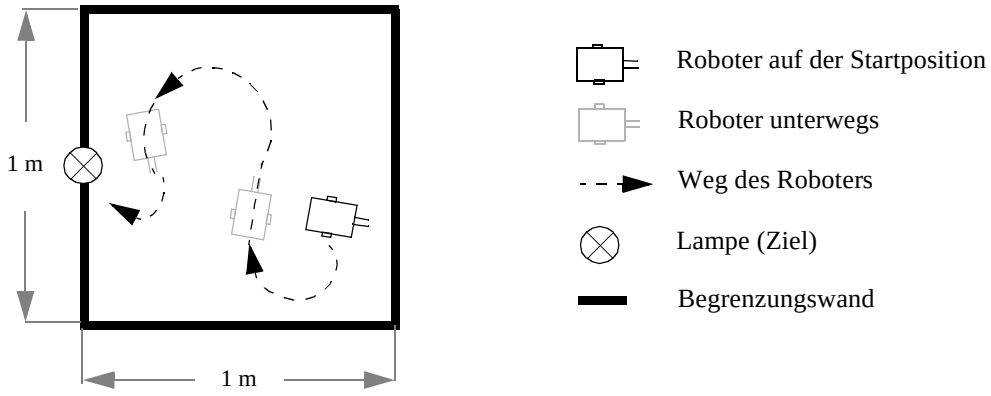


Abb. 4-7: Einsatzumgebung für Aufgabenstellung B

4.3.2 Sensoren und Aktoren des Roboters

Der Roboter besteht neben den nicht elektrischen LEGO-Bauteilen aus 2 RCX-Bausteinen, 3 Lichtsensoren, 6 Tastsensoren, 3 Motoren und 1 Lampe (Abb. 4-8). Jeder RCX-Baustein verfügt über 3 Ausgänge (A,B,C) und 3 Eingänge (1,2,3). Trotzdem ist es möglich, den RCX-Baustein mit mehr als drei Sensoren auszustatten. Beispielsweise können Lichtsensor und Tastsensor am selben Eingang angeschlossen werden. Dies ist möglich, da der RCX-Baustein Analogeingänge hat. So meldet ein Tastsensor, wenn er gedrückt wurde, eine Signalstärke von fast 100%. Ansonsten liegen 0% an. Damit ein Lichtsensor eine Signalstärke von 100% liefert, muss er direkt auf eine helle Lampe gerichtet sein. Die Zuordnung des gemessenen Eingangswertes zum entsprechenden Sensor kann durch die Software erfolgen. Bei Werten um 100% ist der Tastsensor gedrückt und die Lichtstärke kann nicht gemessen werden. Sonst ist der Tastsensor nicht gedrückt und der Wert steht für die gemessene Lichtstärke.

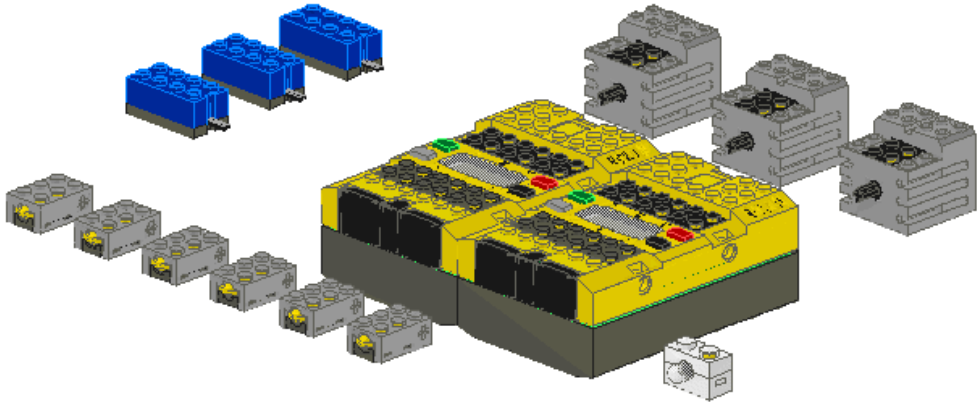


Abb. 4-8: Verwendete Bauteile

Trotzdem ist es nicht möglich, alle für die Aufgabenstellung benötigten Sensoren an einen RCX-Baustein anzuschließen. Abhilfe schafft die Möglichkeit, dass zwei RCX-Bausteine miteinander über die Infrarot-Schnittstelle kommunizieren können. Auf diese Weise wird eine Verdopplung der Zahl der Ein- und Ausgänge erreicht. Abb. 4-9 zeigt, welche Sensoren bzw. Aktoren an welchen RCX-Baustein angeschlossen sind.

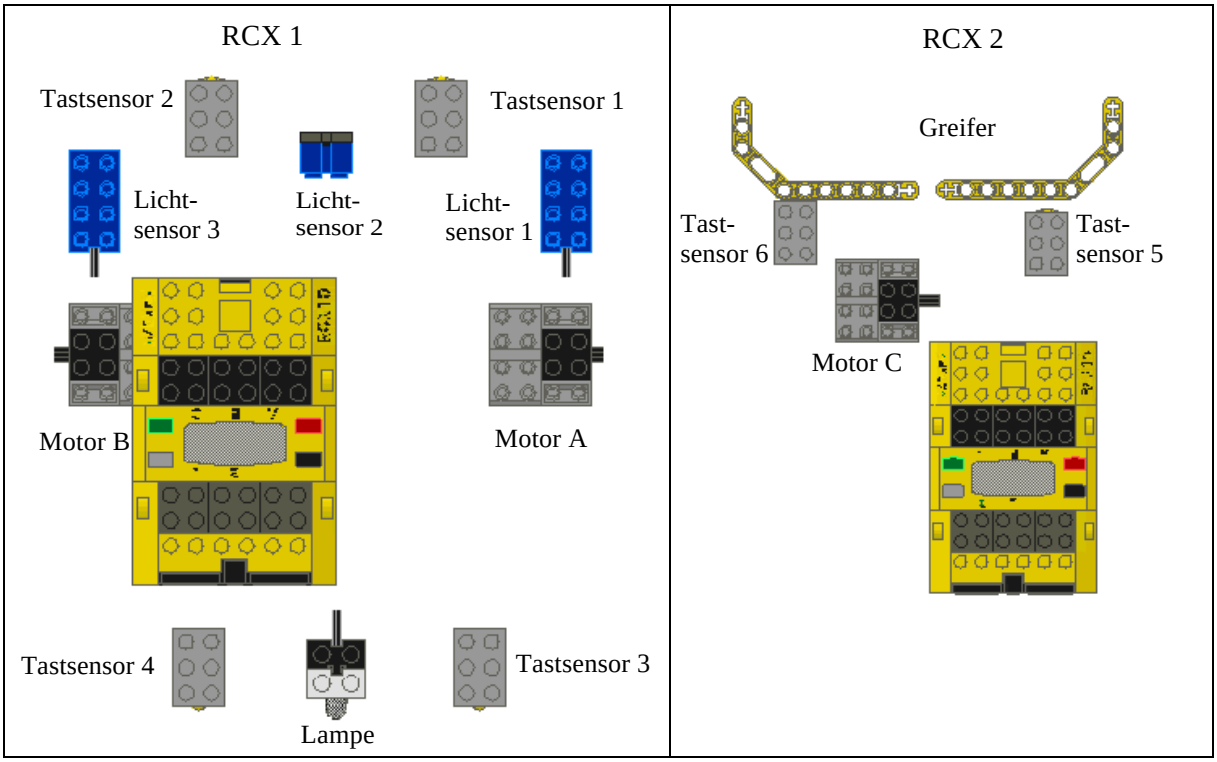


Abb. 4-9: Angeschlossene Aktoren/Sensoren an RCX 1 und RCX 2

RCX 1 ist für die Navigation zuständig. Dieser Baustein steuert die Motoren der beiden angetriebenen Räder und schaltet die Lampe ein und aus. Zur Lokalisation der Zielobjekte und des Zielgebiets ist der Roboter mit den Lichtsensoren 1 und 3 ausgestattet. Dadurch wird es dem Roboter ermöglicht zu unterscheiden, ob eine Lichtquelle sich eher links oder eher rechts von der momentanen Position befindet. Um ein Zielobjekt aufzunehmen, sind diese Messwerte wegen des großen Abstands der beiden Senso-

ren voneinander zu ungenau. Daher wird Lichtsensor 3 benötigt. Dieser ist senkrecht zum Boden ausgerichtet. Wenn sich die Lampe im Deckel des Zielobjektes genau unter dem Lichtsensor befindet, ist der Roboter und damit der Greifer genau vor dem Objekt positioniert.

Für die Kollisionserkennung sind die Tastsensoren 1 bis 4 zuständig. 1 und 2 melden Hinderniskontakte, die vorne links oder vorne rechts auftreten. Da nur drei Eingänge zur Verfügung stehen, sind die Tastsensoren 3 und 4 beide am selben Eingang angeschlossen. Kollisionen am hinteren Teil des Roboters können daher nicht nach links oder rechts aufgelöst werden.

RCX 2 steuert über Motor C den Greifer. Er verfügt über die Tastsensoren 5 und 6, die melden, wenn der Greifer vollständig geschlossen oder geöffnet ist. Im ersten Fall bedeutet dies, dass kein Zielobjekt aufgenommen wurde, da sonst ein völliges Schließen des Greifers verhindert wird.

Abb. 4-10 skizziert die Anordnung der Aktoren und der Sensoren. Eine detaillierte Bauanleitung und eine genaue Anschlussbelegung befinden sich in Anhang B.

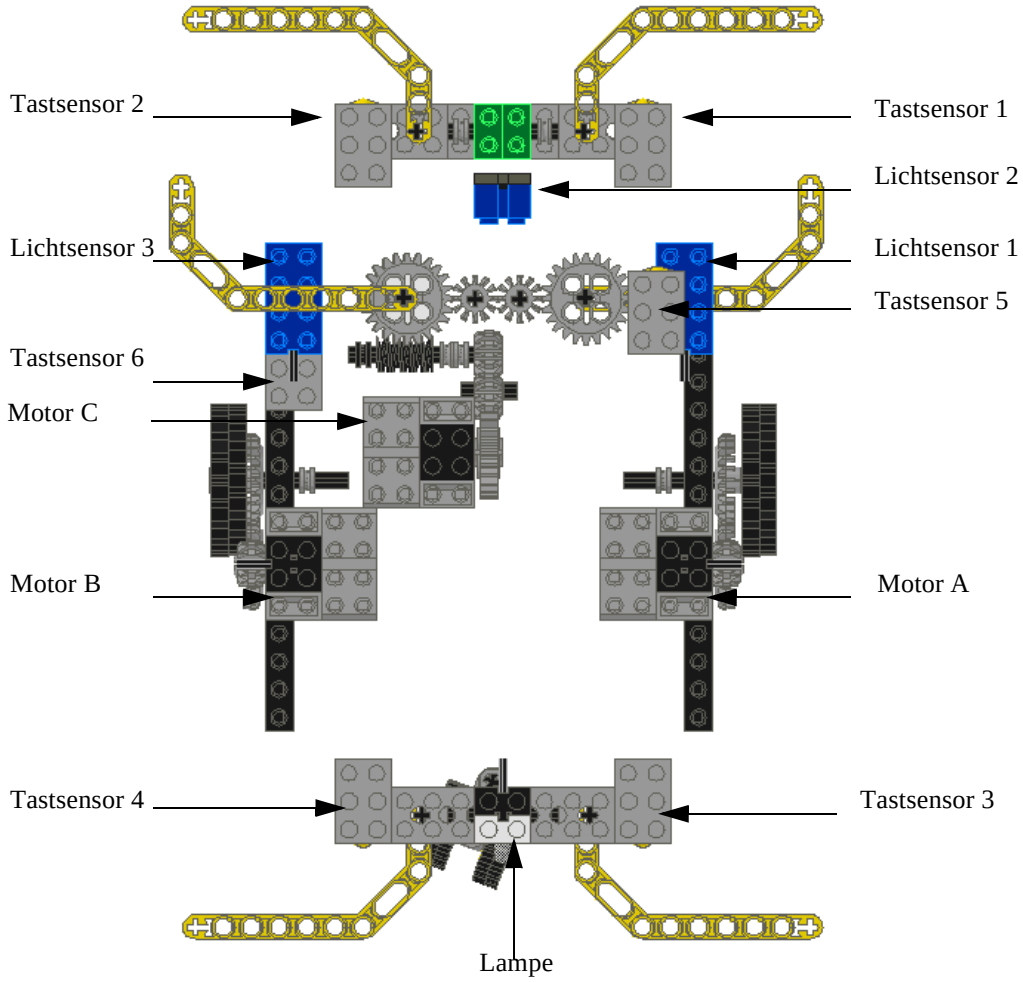


Abb. 4-10: Anordnung der Aktoren und Sensoren

4.3.3 Benötigte Verhaltensweisen

Um seine Aufgabe zu erfüllen, muss der Roboter einige Hauptverhaltensweisen beherrschen. Diese sind:

- **Zielobjekte finden und aufnehmen:**
  Um Zielobjekte grob zu lokalisieren, dreht der Roboter sich auf der Stelle und scannt mit den Lichtsensoren 1 und 3 die Lichtstärke der Umgebung. Dann dreht er sich bis zur maximal gemessenen Lichtstärke und fährt darauf zu. Wenn die Lichtstärke einen bestimmten Schwellenwert überschreitet, benutzt der Roboter Lichtsensor 2, um sich genau vor dem Zielobjekt zu positionieren. Danach schließt er seinen Greifer. Wenn der Greifer sich komplett schließt, hat der Roboter das Zielobjekt verfehlt und startet eine neue Suche. Sonst fährt er mit der Verhaltensweise „Ziel finden und ansteuern“ fort.
- **Ziel finden und ansteuern:**
  Das Zielgebiet ist mit einer Lampe markiert. Um es zu finden, dreht sich der Roboter einmal im Kreis und misst die Lichtstärke. Dann dreht er sich wieder bis zur hellsten Stelle und fährt auf die Lichtquelle zu. Korrekturen während der Fahrt sind durch die beiden Lichtsensoren 1 und 3 möglich. Wenn der linke Sensor eine größere Lichtstärke meldet, fährt der Roboter nach links, wenn der Wert des rechten Sensors größer ist, fährt er nach rechts. Um zu vermeiden, dass der Roboter sich z. B. durch Störlicht völlig verfährt, muss der Scan in bestimmten Abständen wiederholt werden. Wenn die gemessene Lichtstärke einen Grenzwert überschreitet und einer der vorderen Kollisionsmelder (Tastsensor 1 oder 2) auslöst, ist der Roboter im Ziel angekommen. Um das Zielobjekt abzusetzen, wird der Greifer geöffnet.
- **Kollisionserkennung und Ausweichen:**
  Die Kollisionserkennung ist über die Tastsensoren 1 bis 4 möglich. Der Roboter fährt nach einer Kollision ein Stück in die entgegengesetzte Richtung und dreht sich auf der Stelle. Wenn die Kollision vorne stattfand, ist bekannt auf welcher Seite sich das Hindernis befindet und die Drehung kann in die richtige Richtung erfolgen. Sonst muss „geraten“ und bei erneuter Kollision in die entgegengesetzte Richtung ausgewichen werden.
- **Lampe einschalten**
  Bei Erreichen des Zielgebietes schaltet der Roboter seine Lampe ein.

Die Verhaltensweisen haben unterschiedliche Prioritäten und können unterbrochen werden. Beispielsweise wird bei einer Kollision „Ziel finden“ unterbrochen und erst fortgesetzt, wenn dem Hindernis erfolgreich ausgewichen wurde. Diese Verhaltensweisen sollen nur die prinzipielle Funktionsweise des Roboters deutlich machen. Sie bestehen aus mehreren Schritten und finden sich nicht unbedingt in dieser Form im Programm des Roboters wieder. Zu beachten ist auch, dass der Roboter je nach Aufgabenstellung nicht unbedingt alle Verhaltensweisen beherrschen muss. Aufgrund des bei der Realisierung verwendeten Komponentenkonzeptes ist es einfach möglich, die Funktionalität entsprechend anzupassen. Eine genaue Beschreibung, wie der Roboter die Aufgabenstellungen erfüllt, befindet sich in der Systemdokumentation im Anhang A.

Kapitel 5

Der Entwicklungsprozess

Die in Kapitel 3 gemachten Aussagen und Änderungsvorschläge sollen an einem Beispielsystem bezüglich ihrer Tauglichkeit untersucht werden. Um die wichtigsten Sachverhalte zu verdeutlichen, wird die Entwicklung des Systems in Kapitel 5 auszugsweise beschrieben. Ein komplettes KobrA-Dokument mit allen erzeugten Artefakten befindet sich in Anhang A.

Um die Eignung von KobrA für komponentenbasierte Softwareentwicklung und für Product Line Engineering zu demonstrieren, muss die zu erfüllende Aufgabe eine gewisse Variabilität aufweisen. Es werden daher zwei Aufgabenstellungen, die in wesentlichen Teilen übereinstimmen, vorgestellt. Entsprechend der KobrA-Methode wird dann ein System entwickelt, das die Aufgaben erfüllt. Wegen der großen Ähnlichkeit der beiden Aufgabenstellungen sind die zentralen Anforderungen identisch. Durch das Komponentenkonzept von KobrA kann die abweichende Funktionalität leicht durch Hinzunahme, Entfernung oder Änderung von Komponenten realisiert werden.

5.1 Framework Engineering

Zunächst werden die Teilaktivitäten *Context Realization*, *Komponent Specification*, *Komponent Realization* und *Komponent Implementation* gemäß den in Kapitel 3 aufgeführten Modifikationen durchgeführt. Die Inspektion der Artefakte geschieht informell und wird nicht dokumentiert, da laut Abschnitt 3.1.6 keine Besonderheiten erwartet werden. Aus dem gleichen Grund entfällt *Measurement of Structural Properties* komplett. Für den in diesem Beispiel realisierten LEGO Roboter ist *Komponent Reuse* nicht möglich. So gibt es weder COTS-Komponenten (Commercial of the Shelf), noch entstehen im Rahmen der Aufgabenstellung KobrA-Komponenten, die wiederverwendet werden können. Der Mangel an COTS-Komponenten und *Design Patterns* ist ein Schwachpunkt eingebetteter Systeme. Ein Grund dafür, ist u. a. die starke Hardwarezentrierung zusammen mit der Hardwarevielfalt. Die einzige Form der Wiederverwendung, die für den das Beispielsystem verwendet wird, sind zu leJOS gehörende Klassenbibliotheken. So stellt „josx.platform.rcx“ Klassen für die Benutzung der Hardware zur Verfügung. Beispielsweise enthält die Klasse „Sensor“ Methoden, um die Eingänge des RCX-Bausteins abzufragen. *Testing* umfasst im Rahmen des *Framework Engineerings* auch die Aufstellung der Testfälle.

5.1.1 Context Realization

Die Context Realization erfordert ursprünglich die Artefakte *Enterprise Model*, *Structural Model*, *Activity Model*, *Interaction Model* und *Decision Model*. Wie in Kapitel 3 beschrieben, wird das *Enterprise Model* durch ein *System Model* ersetzt und die zusätzlich eingeführten Artefakte *Hardware Component Model* und *Experience Model* erzeugt.

Zuerst wird der *Framework Scope* untersucht und eine *Scope Definition Table* aufgestellt. Dies ist zwar nicht vorgeschrieben, wird aber als Startpunkt empfohlen [ATK01]. Ein Vorteil dieser Vorgehensweise zeigt sich auch bei diesem Beispiel. So wird ohne detaillierte und zeitaufwendige Analyse ein Überblick sowohl über die Aufgabenstellung, die Variabilitäten der Produktlinien und die Features ermöglicht.

5.1.1.1 Framework Scope

Als Vorbereitung für die Aufstellung der *Scope Definition Table* (vgl. Tabelle 5-1) werden zuerst *Domain*, *Product Lines* und *Features* untersucht. *Domain* beschreibt ganz allgemein, worum es in der Aufgabestellung geht. Die verschiedenen Produktlinien und ihre Unterschiede werden in den *Product Lines* beschrieben, während die *Features* die Punkte beschreiben, die in der *Scope Definition Table* in den Zeilen stehen.

Domain

Es steht ein fertiger mobiler autonomer Roboter zur Verfügung, der, nachdem er gestartet wurde, in seiner Einsatzumgebung selbstständig bestimmte Aufgaben erledigen soll.

Product Lines

Um den Produktlinienaspekt von KobrA zu untersuchen, muss die Aufgabenstellung eine gewisse Variabilität aufweisen. Im Beispiel werden zwei Produktlinien unterschieden, die in wesentlichen Teilen übereinstimmen. **Roboter A** soll Objekte einsammeln und in ein Ziel transportieren. Er wird auf glatten Oberflächen eingesetzt. **Roboter B** soll das Ziel finden und aufsuchen. Im Ziel angekommen, schaltet er seine Lampe ein. Eingesetzt wird dieser Roboter auf Teppichboden.

Features

Beide Roboter verfügen über bestimmte zur Erfüllung ihrer Aufgabenstellung erforderlichen Features. Dies sind zum einen **Fähigkeiten**, wie beispielsweise „Ziel aufsuchen“ und „Kollision erkennen und ausweichen“. Auch **Anforderungen** bezüglich der Einsatzumgebung wie „glatte Oberfläche“ bzw. „Teppichboden“ gehören zu den Features. Beide Roboter sind, da laut Domain ein fertiger Roboter benutzt wird, aus den gleichen **Hardwarebaugruppen** zusammengesetzt. Diese sind „Antrieb“, „Kollisionssensorik“, „horizontale Lichtsensorik“, „vertikale Lichtsensorik“, „IR-Schnittstelle“ und „Lampe“. Dass je nach Produktlinie nicht alle benötigt werden, kann später in der Scope Definition Table berücksichtigt werden (vgl. Tabelle 5-1).

Scope Definition Table

In der *Scope Definition Table* tauchen die in Kapitel 3 eingeführten Hardwarebaugruppen auf. Diese vereinigen Aktoren und Sensoren, die logisch zusammengehören. So besteht beispielsweise der Greifer aus einem Motor und zwei Tastsensoren. Durch die Gruppierung wird auch der Zweck deutlich. Der Begriff Greifer beinhaltet eine Interpretation, während das Wissen, dass der Roboter u. a. mit zwei Tastsensoren und einem Motor ausgestattet ist, keine Deutung über ihre Bestimmung zulässt. Noch komplexer als der Greifer ist die Kollisionssensorik. Sie besteht aus drei Tastsensoren, die an die gleichen Eingänge angeschlossen sind wie die drei Lichtsensoren.

Die Hardwarebaugruppen Greifer, Kollisionssensorik und Antrieb werden später durch entsprechende Klassen gesteuert. Für die Lampe, die Lichtsensorik und die IR-Schnittstelle existieren spezielle Methoden. Die Auflistung der Hardwarebaugruppen in der *Scope Definition Table* ermöglicht damit bereits für manche Klassen und Methoden eine grobe Zuordnung zu den Produktlinien. Wird beispielsweise der Greifer des Roboters nicht verwendet, können alle zugehörigen Softwarebestandteile weggelassen werden.

Die Punkte „Fähigkeiten“ und „Verwendete Hardwarebaugruppen“ sind teilweise voneinander abhängig. So bedingt beispielsweise der Wegfall des Greifers dass die Verhaltensweisen „Zielobjekt aufsuchen und greifen“ ebenfalls wegfallen können bzw. umgekehrt. Eventuell muss die *Scope Definition Table* daher später angepasst werden, da zu Beginn noch nicht klar sein muss, welche Hardwarebaugruppe für welches Verhalten benötigt wird. Beispielsweise stehen solange noch keine Strategie entwickelt wurde wie ein Zielobjekt lokalisiert wird, die dazu benötigten Sensoren noch nicht fest.

Zusätzlich enthält die Tabelle den Punkt „Anforderungen“. Dieser berücksichtigt die unterschiedlichen Einsatzumgebungen „Teppich“ und „Tischplatte“. Bei der Realisierung des Roboters stellte sich heraus, dass die Anforderungen erst auf der Implementierungsebene durch unterschiedliche Geschwindigkeiten und Zeitspannen realisiert werden konnten. Während der *Context Realization* wird mit Hilfe eines *Decision Models* eine Abbildung der *Features* der auf die Entscheidungen vorgenommen. Diese Abbildung ist für die „Anforderungen“ nicht möglich. Aus Verfolgbarkeitsgründen werden sie trotzdem im entsprechenden *Decision Model* aufgeführt. Dabei wird auf den Ort verwiesen, an dem die „Anforderungen“ erfüllt werden (vgl. Tabelle 5-2). Dagegen wird der Punkt „verwendete Hardwarekomponenten“ nicht im *Decision Model* aufgeführt, weil ein fertiger Roboter zur Verfügung steht.

Die *Scope Definition Table* ist eine große Hilfe, um die Variabilitäten im Auge zu behalten. In diesem Beispiel mit zwei relativ kleinen Produktlinien fällt dies nicht so sehr ins Gewicht. Bei komplexeren Aufgabenstellungen oder vielen Produkten in einer Produktlinie ist ohne eine zentrale Stelle, an der die Variabilitäten aufgelistet sind, die Gefahr groß, den Überblick zu verlieren.

		Roboter A	Roboter B
Fähigkeiten	Ziel aufsuchen	x	x
	Kollision erkennen und ausweichen	x	x
	Zielobjekt aufsuchen und greifen	x	
	Lampe einschalten		x
Anforderungen	Funktioniert auf glatter Oberfläche (Tischplatte)	x	
	Funktioniert auf Teppichboden		x
Verwendete Hardwarebaugruppen	Antrieb	x	x
	Kollisionssensorik	x	x
	horizontale Lichtsensorik	x	x
	vertikale Lichtsensorik	x	
	Greifer	x	
	IR-Schnittstelle	x	
	Lampe		x

Tabelle 5-1: *Scope Definition Table* des Roboter Framework

Feature		Entschluss	Effekt
Fähigkeiten	Zielobjekt aufsuchen und greifen	ja	Context Realization Entscheidung 1: ja
		nein (default)	Context Realization Entscheidung 1: nein
	Lampe einschalten	ja (default)	Context Realization Entscheidung 2: ja
		nein	Context Realization Entscheidung 2: nein
Anforderungen	Funktioniert auf glatter Oberfläche	Bei *Komponent Implementation* der Komponente DrivingSystem realisiert (Tabelle A-59).	
	Funktioniert auf Teppichboden	Bei *Komponent Implementation* der Komponente DrivingSystem realisiert (Tabelle A-59).	

Tabelle 5-2: Decision Model für die Abbildung der Features auf die Entscheidungen

5.1.1.2 System Model

Nachdem der *Framework Scope* einen Überblick über die Aufgabenstellung und die Variabilitäten der Produktlinien gegeben hat, untersucht das *System Model* den Aufbau des Roboters und identifiziert die

wesentlichen Prozesse. Dazu werden die Artefakte *System Concept Diagram* (Abb. 5-1), *System Process Hierarchy* (Abb. 5-2) und *Hardware Properties* (Tabelle 5-3) erzeugt.

System Concept Diagram

Das *System Concept Diagram* beschreibt den Aufbau des Roboters und seine Interaktionsmöglichkeiten mit der Umgebung. Der Beispielroboter enthält zwei RCX-Bausteine. Einer steuert die Greiferhardware, der andere RCX-Baustein realisiert mit Kollisionssensorik, Lichtsensorik, Antrieb und Lampe ein Fahrzeug. Für die Kombination aus Baugruppen und Rechenkapazität wurde in Kapitel 3 der Begriff der *Hardware Component* eingeführt. So besteht der Beispielroboter aus einer *Hardware Component* für das Fahrzeug und einer für den Greifer.

Um im Folgenden Begriffsverwirrungen zu vermeiden, werden die beiden *Hardware Components* mit „DriverHWC“ und „GrabberHWC“ (nicht zu verwechseln mit der Baugruppe „Greifer“) bezeichnet. Ihre entsprechenden Software-Komponenten sind „DrivingSystem“ bzw. „GrabberSystem“. Das System aus den zusammenarbeitenden Komponenten „DriverHWC“ und „GrabberHWC“ wird mit „RobotSystem“ bezeichnet.

Abb. 5-1 zeigt die Sensoren und Aktoren, mit denen die einzelnen *Hardware Components* ausgestattet sind. Sie stehen in direkter Verbindung mit den beiden *Software Components* „DrivingSystem“ und „GrabberSystem“. Die Identifizierung weiterer *Software Components* ist nicht Bestandteil des *Enterprise Modeling*, sondern geschieht erst während des *Structural Modeling*. „DriverHWC“ und „GrabberHWC“ kommunizieren über Nachrichten miteinander. Zusätzlich sind für alle Aktoren und Sensoren die Einflussgrößen und die Wirkrichtung angegeben. Dabei muss beachtet werden, dass die gleichen Sensortypen unterschiedliche Ereignisse melden können. So meldet beispielsweise der Tastsensor von „DriverHWC“ einen Wandkontakt, während die Tastsensoren von „GrabberHWC“ zur Bestimmung des Greiferstatus benötigt werden.

Die Hardwarebestandteile des *System Models* werden später in den Klassendiagrammen durch die entsprechenden Klassen aus der zum Lieferumfang von *leJOS* gehörenden Bibliothek „josx.platform.rcx“ abstrahiert. Damit bietet das *System Concept Diagram* einen ersten Anhaltspunkt für die Erstellung des *Class Diagrams*.

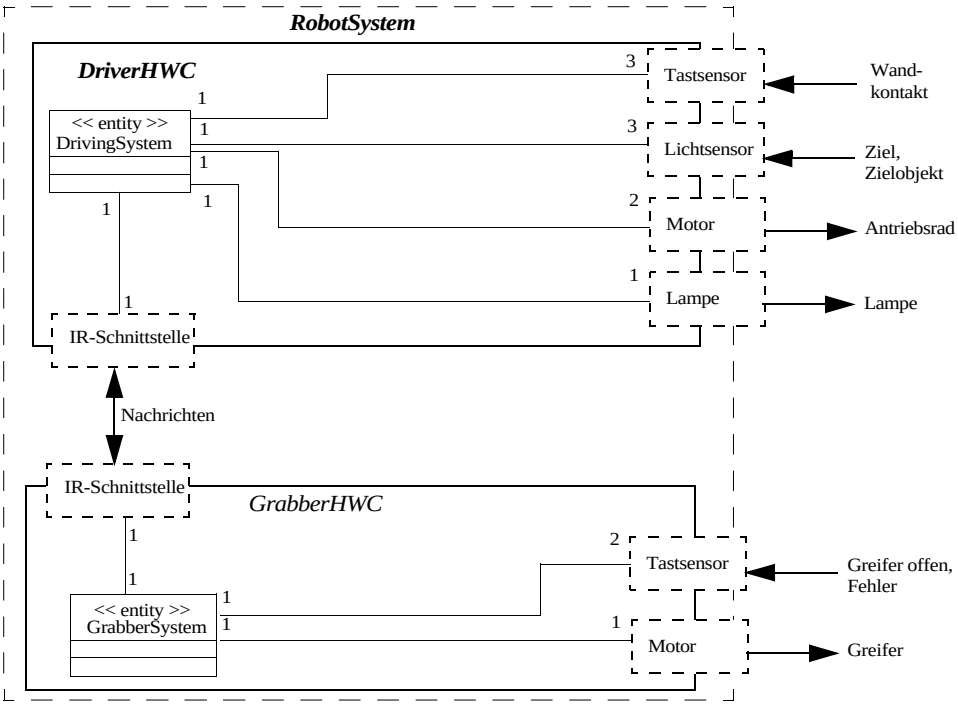


Abb. 5-1: RobotSystem *System Concept Diagram*

System Process Hierarchy

Die *System Process Hierarchy* (Abb. 5-2) des „RobotSystems“ gibt einen Überblick über die benötigten Tätigkeiten bzw. Prozesse des Roboters. Dabei werden auch Hierarchien gebildet. So besteht der Prozess „Kommunizieren“ aus den Teilprozessen „Nachricht senden“ und „Nachricht empfangen“. Als Wurzel der *System Process Hierarchy* dient das „RobotSystem“. Danach erfolgt die Aufspaltung in die beiden Äste „DriverHWC“ und „GrabberHWC“. Auf diese Weise ist es möglich, die Prozesse den beiden *Hardware Components* zuzuordnen. Im Gegensatz zu den Prozessen sind die Rahmen der Hardwarebestandteile in der *System Process Hierarchy* gestrichelt. Sie bilden ein organisatorisches Element für die Gruppierung der Prozesse. Optionale Prozesse sind grau unterlegt.

Zur Bestimmung der Prozesse gibt es keine klaren Regeln. Existierende Unternehmensprozesse, wie bei der *Enterprise Process Hierarchy*, die nur identifiziert werden müssen, existieren bei einem eingebetteten System nicht. Ein erster Anhaltspunkt waren die in der *Scope Definition Table* (vgl. Tabelle 5-1) aufgeführten Fähigkeiten. Die gefundenen Prozesse erleichtern später die Identifizierung der Aktivitäten für das *Structural Model* und die *Activity Models*.

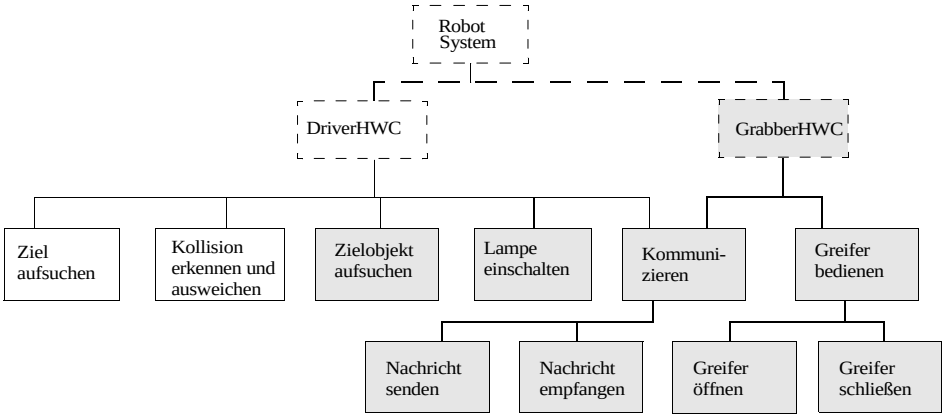


Abb. 5-2: RobotSystem *Process Hierarchy*

Hardware Properties

Auf den ersten Blick scheinen die Eigenschaften der in Tabelle 5-3 aufgelisteten Aktoren und Sensoren an dieser Stelle noch keine Rolle zu spielen, sondern vor allem die Implementierung zu betreffen. Da die Sensoren von verschiedenen *Hardware Components* verwendet werden, werden ihre Eigenschaften, um Wiederholungen zu vermeiden, in der *Context Realisation* aufgeführt. Ein anderer Vorteil wird beim Betrachten der Wertebereiche für Licht- und Tastsensoren deutlich. Dabei fällt auf, dass beide an einem Eingang angeschlossen werden können, wie unter 4.3.2 beschrieben. In unserem Beispiel war das durch die Verwendung des fertigen Roboters bereits bekannt. Es muss aber nicht immer eine fertige Hardware als Ausgangspunkt bereitstehen. Denkbar ist auch die parallele Entwicklung von Hard- und Software. In diesem Fall sparen die aus Tabelle 3 gewonnenen Erkenntnisse drei Eingänge ein, so dass statt drei RCX-Bausteinen zwei genügen.

Bezeichnung	Typ	gemessene/manipulierte Größe	Wertebereich	Einheit
Tastsensor	Sensor	Hinderniskontakt	int, [0,100]	%
Lichtsensor	Sensor	Helligkeit	int, [0,..,100]	%
Motor	Aktor	Geschwindigkeit	int, [0,..,7]	%
Lampe	Aktor	Lichtstärke	int, [0,..,7]	%

Tabelle 5-3: *Hardware Properties*

5.1.1.3 Structural Model

Da der Beispielroboter über keinen Bildschirm verfügt, entfallen die *User Interface Artefact*s. Ein *Object Diagrams* muss ebenfalls nicht aufgestellt werden. Laut [ATK01] können *Object Diagrams* dazu verwendet werden, eine typische Konfiguration an Instanzen aufzuzeigen, in der das System arbeitet. Im konkreten Beispiel ist dies überflüssig, da nur eine mögliche Konfiguration existiert. Damit wird im Rahmen des *Structural Models* nur das *Class Diagram* erstellt.

Class Diagrams

Abb. 5-3 zeigt das *Class Diagram* des Roboters. Da die Funktionalität auf zwei unabhängige *Hardware Components* aufgeteilt ist, besteht es aus zwei *Class Diagrams*. „GrabberHWC“ und „DriverHWC“ kommunizieren miteinander über Nachrichten. Deswegen sind die beiden *Class Diagrams* über einen Kommunikationskanal verbunden.

Bei der Aufstellung der *Class Diagrams* gab es keine Besonderheiten. Die *Hardware Component* „GrabberHWC“ besteht nur aus der Klasse „GrabberSystem“. Dagegen enthält „DriverHWC“ neben der Klasse“ DrivingSystem“ auch die beiden Klassen „CollisionDetect“ und „MotorControl“.

„CollisionDetect“ wurde eingeführt, da die Kollisionserkennung eine gewisse Komplexität aufweist. So muss die Kollisionsquelle (vorne links, vorne rechts oder hinten) identifiziert werden. Da bei „DriverHWC“ immer ein Lichtsensor und ein Tastsensor zusammen an einem Eingang angeschlossen sind, muss außerdem festgestellt werden, welcher Sensor die gemessenen Werte gerade bedingt. Die Klasse „MotorControl“ ist für den Antrieb des Roboters zuständig. Sie soll später ein Interface bieten, um komplexe Manöver wie beispielsweise „fahre links“ aufzurufen.

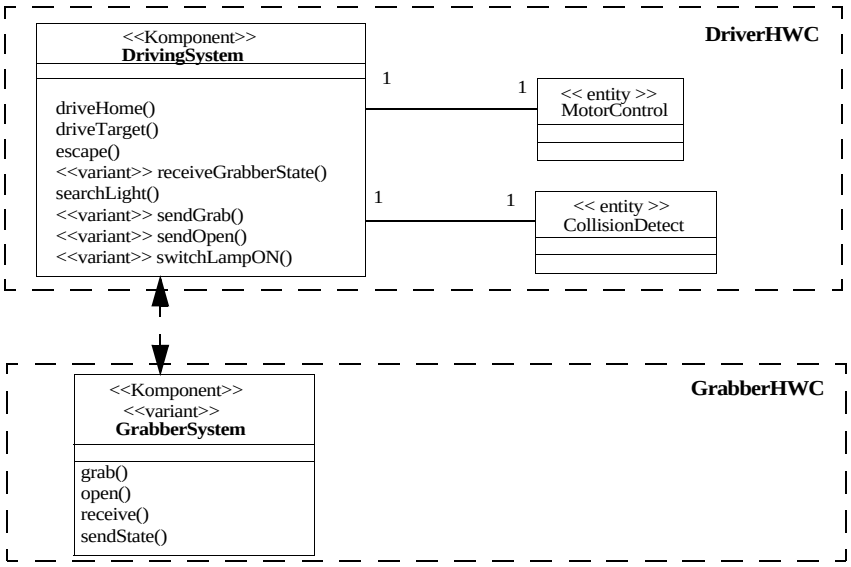


Abb. 5-3: *Class Diagram* der *Context Realization*

5.1.1.4 Hardware Component Model

Das in Kapitel 3 neu eingeführte *Hardware Component Model* berücksichtigt einige Besonderheiten eingebetteter Systeme. Dazu werden die beiden Modelle *Hardware Component Containment Model* und *Message Model* erzeugt.

Hardware Component Containment Model

Das *Hardware Component Containment Model* zeigt die Zusammengehörigkeit zwischen *Hard-* und *Software Components* in einer Tabelle (vgl. Tabelle 5-4). Im Fall des Beispielroboters ist dies wegen der 1:1 Beziehung zwischen den beiden trivial. So enthält die *Hardware Component* „GrabberHWC“ nur die *Software Component* „GrabberSystem“, während „DriverHWC“ nur aus „DrivingSystem“

besteht. Dies muss aber nicht der Fall sein. So kann eine *Hardware Component* durchaus mehrere *Software Components* enthalten.

Hardware Component	Software Component
GrabberHWC	GrabberSystem
DriverHWC	DrivingSystem

Tabelle 5-4: *Hardware Component Model*

Message Model

Damit die beiden *Hardware Components* „GrabberHWC“ und „DriverHWC“ zusammenarbeiten können, müssen das Nachrichtenformat und die Nachrichten definiert werden. Dazu wird das *Message Model* aufgestellt. Dies muss geschehen, bevor mit der weiteren Verfeinerung der einzelnen Komponenten begonnen wird, da das weitere Vorgehen davon beeinflusst wird. So werden die Nachrichten z. B. für die *Activity*- und die *Sequence Diagrams* benötigt.

Das Nachrichtenformat der RCX-Bausteine ist vorgegeben. Es können Pakete von einem Byte Größe versendet werden. Damit können 256 verschiedene Nachrichten codiert werden. Die Kommunikation zwischen den beiden Komponenten wurde aufgrund der Testergebnisse in A.6.3 überarbeitet. Ursprünglich wurde nicht zwischen den Nachrichten zum Öffnen und zum Schließen des Greifers unterschieden, da der Greifer sich nur abwechselnd öffnen und schließen kann. Die Tests haben aber gezeigt, dass Nachrichten verloren gehen können. Dies kann beispielsweise folgende Situation auslösen: „DriverHWC“ ist bekannt, dass der Greifer geschlossen ist und schickt eine Nachricht. Diese wird von „GrabberHWC“ nicht empfangen, der Greifer bleibt geschlossen. „DriverHWC“ nimmt aber an, dass die Nachricht das Öffnen des Greifers ausgelöst hat. Dies führt zu einem Versagen des Systems. Tabelle 5-5 zeigt die überarbeiteten Nachrichten, über die „GrabberHWC“ und „DriverHWC“ miteinander kommunizieren.

Nachricht	Codierung	Sender	Empfänger	Wirkung
<<variant>> Grab	0	DriverHWC	GrabberHWC	Der Greifer wird geschlossen.
<<variant>> Open	1	DriverHWC	GrabberHWC	Der Greifer öffnet sich.
<<variant>> Failed	2	GrabberHWC	DriverHWC	Empfänger wird über Scheitern des Greifvorgang informiert.
<<variant>> OK	3	GrabberHWC	DriverHWC	Empfänger wird über erfolgreichen Greifvorgang informiert.

Tabelle 5-5: *Message Model*

5.1.1.5 Activity Model

KobrA unterscheidet *User Task*, *User Interface Activity* und *System Operation*, die den Ausgangspunkt für *Activity Diagrams*, *Activity Specifications* und das *Use Case Model* bilden. *User Tasks* beziehen sich auf die grundlegenden Verantwortlichkeiten des Benutzers, während *User Interface Activities* die Interaktion zwischen Benutzer und System beschreiben. Sowohl *User Task*, als auch *User Interface Activities* sind Bestanteil der *Context Realization*. Dagegen tauchen *System Operation*s erst während der *Komponent Realization* auf, da sie den Effekt auf die Daten beschreiben. [ATK01]

Da der Beispielroboter über keinen Bildschirm verfügt, entfallen die *User Interface Artefact*s. *User Tasks* existieren, bis auf den Startbefehl, ebenfalls keine. Statt dessen werden stellvertretend die Prozesse der *System Process Hierarchy* des „RobotSystems“ als Ausgangspunkt für die Aufstellung der Modelle und Diagramme des *Activity Models* herangezogen.

Activity Diagrams

Gemäß [ATK01] sollen die *User Tasks* den Ausgangspunkt für die *Activity Diagrams* bilden. Da für den Beispielroboter keine *User Tasks* existieren, werden die Prozesse aus der untersten Ebene der *System Process Hierarchy* an ihrer Stelle verwendet und für die wichtigsten von ihnen ein *Activity Diagram* aufgestellt. Abb. 5-4 zeigt als Beispiel das *Activity Diagram* für den Prozess „Ziel aufsuchen“.

Activity Diagrams können aus „freien Aktivitäten“ aufgebaut sein, d. h. aus Aktivitäten, die noch keiner Klasse bzw. keinen Daten fest zugeordnet sind. In Abb. 5-4 sind beispielsweise „searchLight“ und „driveHome“ solche freien Aktivitäten. Die grau unterlegten Aktivitäten sind optional.

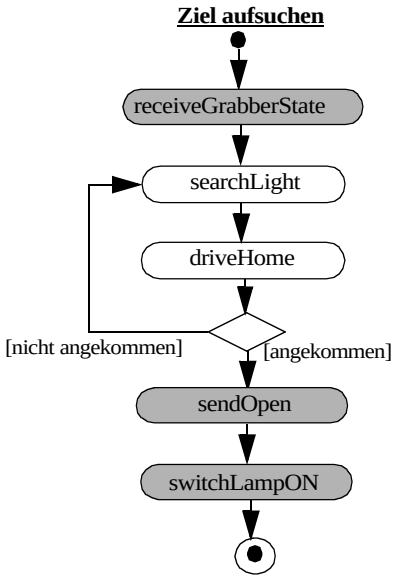


Abb. 5-4: „Ziel aufsuchen“ *Activity Diagram*

Abb. 5-5 zeigt, wie die beiden *Hardware Components* „DriverHWC“ und „GrabberHWC“ zusammenarbeiten. „DrivingSystem“ veranlasst durch eine Nachricht die Komponente „GrabberSystem“ dazu, den Greifer zu schließen. Danach sendet „GrabberSystem“ eine Nachricht über Erfolg bzw. Misserfolg des Greifvorgangs. Abhängig vom Ergebnis des Greifvorgangs startet „DrivingSystem“ eine neuen Versuch bzw. fährt ins Ziel um das Zielobjekt abzuliefern. Da „DrivingSystem“ und „GrabberSystem“

ähnlich miteinander agieren, wie ein System mit einem Benutzer, ist diese Situation mit einem *User Task* vergleichbar. Eine komplette Aufstellung aller *Activity Diagrams* befindet sich im Anhang unter A.2.1.5.

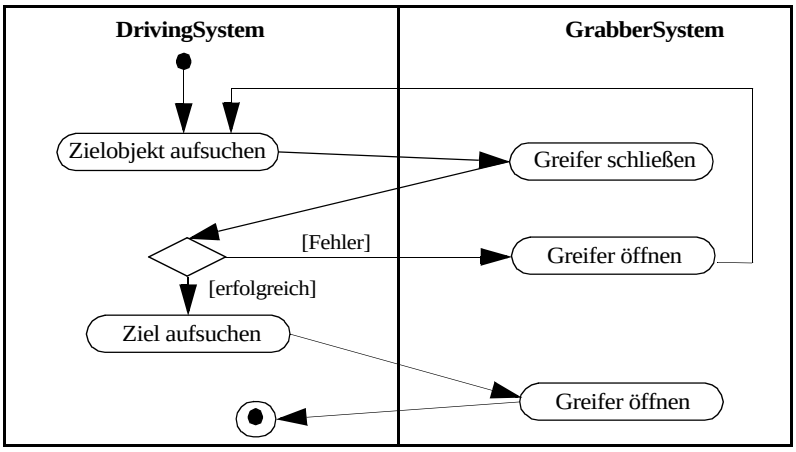


Abb. 5-5: *Activity Diagram* der Kommunikation zwischen „DrivingSystem“ und „GrabberSystem“

Activity Specification

In Kapitel 3 wurden einige neue Punkte für die *Activity Specification* eingeführt. Die *Activity Specification* wird für alle wichtigen freien Aktivitäten erzeugt. Nachfolgend werden die Neuerungen an zwei Beispielen demonstriert.

Tabelle 5-6 zeigt die *Activity Specification* der Aktivität „driveHome“. Der Punkt HW-Komponente zeigt, dass diese Aktivität zu „DriverHWC“ gehört. „driveHome“ fragt die Werte der Licht- und Tastsensoren ab. Die Lichtsensoren werden zur Lokalisierung des Ziels, die Tastsensoren zur Kollisionserkennung benötigt. Um das Ziel anzufahren, werden die Antriebsmotoren manipuliert.

In Tabelle 5-7 wird die Aktivität „sendGrab“ erläutert. Der Punkt Sender zeigt an, dass eine Nachricht verschickt wird. Damit diese gesendet werden kann, muss der Ausgang der IR-Schnittstelle manipuliert werden.

Durch die zusätzlichen Punkte in den Tabellen gelingt ein Überblick, welche Aktivitäten welche Aktoren bzw. Sensoren benötigen und ob die Aktivität mit einer anderen *Hardware Component* über Nachrichten interagiert. Die komplette *Activity Specification* befindet sich im Anhang in Abschnitt A.2.1.5.

Name	driveHome()
Beschreibung	Der Roboter fährt das Ziel an.
HW-Komponente	DriverHWC
Misst	- Lichtsensoren - Tastsensoren
Manipuliert	Antriebsmotoren
Ergebnis	Der Roboter befindet sich im Ziel.

Tabelle 5-6: „driveHome()“ *Activity Specification*

Name	driveHome()
Regeln	Der Roboter befindet sich im Ziel, wenn die Messwerte der horizontalen Lichtsensoren einen Schwellenwert übersteigen und die vorderen Tastsensoren eine Kollision melden.

Tabelle 5-6: „driveHome()“ *Activity Specification*

Name	<<variant>> sendGrab()
Beschreibung	Es wird Nachricht 0 (Greifen) gesendet.
HW-Komponente	DriverHWC
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 0 (Greifen)
Ergebnis	GrabberHWC wurde benachrichtigt.

Tabelle 5-7: „sendGrab()“ *Activity Specification*

Use Case Model

Da im Zusammenhang mit der Entwicklung des „RobotSystems“ der Begriff des Aktors bereits verwendet wurde, wird für die Aktoren der *Use Cases* der englische Begriff *Actor* verwendet.

Normalerweise werden *Use Cases* für *User Tasks* und *User Interface Activities* erstellt. Der Beispielroboter verfügt aber nur über einen einzigen *User Task* (Startbefehl) und über keine *User Interface Activities*. Andererseits stoßen beispielsweise die Kollisionssensoren den Ausweichvorgang an. Sie werden daher über die zugehörige Klasse „CollisionDetect“ als *Actor* in den *Use Cases* modelliert. Auch die beiden *Hardware Components* „DriverHWC“ und „GrabberHWC“ stellen *Actor*s dar, da sie sich Nachrichten schicken und damit Verhaltensweisen starten können.

Damit wurden vier *Actor*s identifiziert: „Benutzer“, „CollisionDetect“, „DriverHWC“ und „GrabberHWC“. Die zugehörigen *Use Cases* basieren auf den Prozessen der *System Process Hierarchy*. Da nicht alle durch die *Actors* angestoßen werden, enthält das *Use Case Diagram* nur eine Untermenge aller Prozesse. Abb. 5-6 zeigt das entsprechende *Use Case Diagram.*

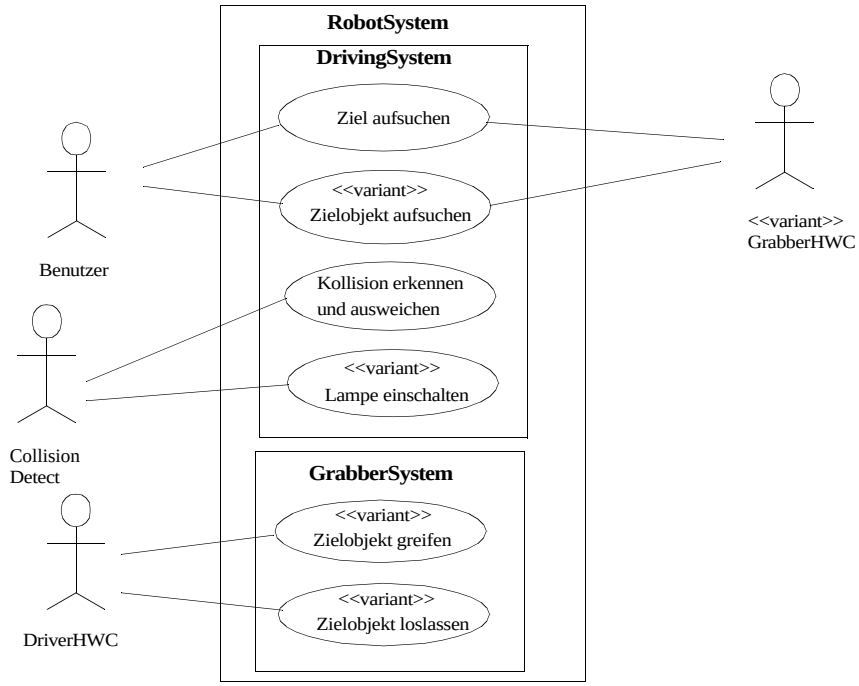


Abb. 5-6: *Use Cases* für das „RobotSystem“

5.1.1.6 Interaction Model

Das *Sequence Diagram* als Teil des *Interaction Models* bietet eine zusätzliche Sicht auf die Realisierung der *User Tasks* durch *System Operations* [ATK01]. Da, wie bereits während des *Activity Models* erläutert, weder *User Tasks* noch *User Interface Activities* existieren, werden an deren Stelle die beim *Activity Model* aufgeführten Aktivitäten herangezogen.

Abb. 5-7 zeigt das *Sequence Diagram* für die Aktivitäten „Zielobjekt greifen“ und „Zielobjekt loslassen“. Da beide eng zusammenhängen, werden sie in einem *Sequence Diagram* modelliert. „DrivingSystem“ wurde analog zum *Use Case Diagram* als *Actor* modelliert. Es wird die Situation betrachtet, dass der Greifversuch zuerst fehlschlägt, dann erfolgreich wiederholt wird und schließlich das Objekt losgelassen wird. „GrabberSystem“ informiert „DrivingSystem“ über Erfolg bzw. Misserfolg des Greifversuchs.

Die *Sequence Diagrams* für die Aktivitäten „Ziel aufsuchen“, „Lampe einschalten“, „Zielobjekt aufsuchen“ und „Kollision erkennen und ausweichen“ befinden sich im Anhang unter A.2.1.6.

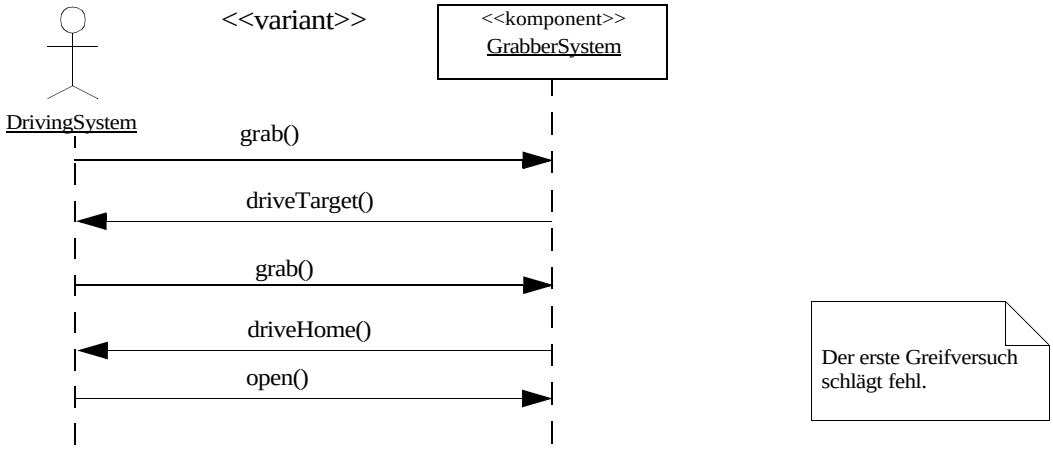


Abb. 5-7: „Zielobjekt greifen“, „Zielobjekt loslassen“ *Sequence Diagram*

5.1.1.7 Decision Model

Decision Models spielen eine zentrale Rolle bei der Instantiierung des Frameworks. Ihre Aufstellung für den Beispielroboter bereitete keine Probleme. Allerdings sind zusätzliche *Decision Models* für neu eingeführte Modelle wie das *Hardware Component Model* (5.1.1.4) erforderlich. Tabelle 5-8 zeigt einen Auszug aus dem zum *Structural Model* gehörenden *Decision Model.* Alle aufgestellten *Decision Models* sind im Anhang in Abschnitt A.2.1.7 aufgeführt. Für das *Decision Model*, welches die Abbildung der Features der *Scope Definition Table* auf die Entscheidungen vornimmt (Tabelle 5-2), gelten die in Abschnitt 5.1.1.1 identifizierten Besonderheiten.

ID		Frage	Variation Point	Entschluss	Effekt
Structural Model	1	Hardwarekomponente *GrabberHWC* benötigt?	Hardwarekomponente *GrabberHWC*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Hardwarekomponente *GrabberHWC*
	2	Operation *driveTarget* benötigt?	Operation *driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *driveTarget*
	⋮	⋮	⋮	⋮	⋮

Tabelle 5-8: *Decision Model* für das *Structural Model* der *Context Realization*

Fazit:

Das Hauptproblem während der *Context Realisation* waren die fehlenden *User Tasks* und *User Interface Activities.* Davon sind das *Activity Model*, das *Interaction Model* und das *Use Case Model* betroffen. Dieses Problem wurde mit Hilfe der *System Process Hierarchy*, die Teil des *System Models* ist, gelöst. Dazu wurden die Prozesse der untersten Ebene der *System Process Hierarchy* stellvertretend für die *User Tasks* und *User Interface Activities* verwendet.

Die Aufstellung der *System Process Hierarchy* löst auch ein anderes Problem. So fehlte für die Entwicklung des Beispielroboters ein fester Startpunkt. Im Gegensatz zur *Enterprise Process Hierarchy*,

für das bereits existierende Prozesse identifiziert werden, existierte für den Beispielroboter anfangs nur die Aufgabenstellung. Für die Aufstellung der *System Process Hierarchy* lautete die Fragestellung daher nicht „Über welche vorhandenen Prozesse verfügt der Roboter?“, sondern „Welche Prozesse benötigt der Roboter, um seine Aufgabenstellung zu erfüllen?

Während der *Context Realisation* ist eine durchgängige und zusammenhängende Vorgehensweise möglich. So kann die *Scope Definition Table* eine Hilfestellung bei der Aufstellung des *System Models* bieten. Dieses besteht aus der *System Process Hierarchy* und dem *System Concept Diagram*. Die Prozesse in der untersten Ebene der *System Process Hierarchy* bilden dann den Ausgangspunkt für einige der aufzustellenden Modelle. So finden sich die Prozesse der *System Process Hierarchy* in den *Use Cases*, den *Activity Specifications* und den *Sequence Diagrams* wieder. Da die Prozesse der *System Process Hierarchy* durch die Methoden des *Structural Models* realisierbar sein müssen, wird auch dieses Modell von der *System Process Hierarchy* beeinflusst. Das *System Concept Diagram* liefert ebenfalls Anhaltspunkte für den Aufbau des *Structural Models*. Die Erstellung der anderen Modelle bereitete keine Probleme. Da der Aufbau der *Decision Models* im Prinzip immer gleich ist, wird im weiteren Verlauf dieses Kapitels auf ihre Aufführung verzichtet.

Bei der Entwicklung des Beispielroboters wurden zwei Komponenten identifiziert. Die Komponente „GrabberSystem“ kontrolliert den Greifer, während die Komponente „DrivingSystem“ das Fahrzeug steuert, die Zielobjekte bzw. das Ziel lokalisiert und die Lampe ein- bzw. ausschaltet. Die entsprechenden KobrA-Dokumente befinden sich im Anhang, „GrabberSystem“ in A.3 und „DrivingSystem“ in A.4. Beide Komponenten verfügen weder über *User Tasks* noch *User Interface Activities.*

5.1.2 Komponent Specification

Die *Komponent Specification* wird für die beiden Komponenten „GrabberSystem“ und „DrivingSystem“ durchgeführt. Da „GrabberSystem“ keine Variabilität aufweist, werden für diese Komponente keine *Decision Models* benötigt. Die Entwicklung von „GrabberSystem“ kann somit als Beispiel für eine Einzelentwicklung mit KobrA angesehen werden.

In Kapitel 3 sind für die *Komponent Specification* keine wesentlichen Modifikationen vorgenommen worden. So wurde lediglich das *Functional Model* geringfügig erweitert und Tips für die Aufstellung des *Behavioural Models* gegeben.

5.1.2.1 Structural Model

Das *Structural Model* der Komponente „DrivingSystem“ enthält als <<subject>> „DrivingSystem“. Die während der *Context Realization* identifizierten Methoden werden übernommen. Außerdem enthält das *Structural Model* die beiden Klassen „CollisionDetect“ und „MotorControl“. „CollisionDetect“ ist für die Kollisionserkennung verantwortlich und soll die Ausführung von „escape()“ initiieren. „MotorControl“ stellt eine Schnittstelle zur Ansteuerung der Motoren zur Verfügung. Bei der Modellierung des *Structural Model* traten keine Besonderheiten eingebetteter Systeme auf.

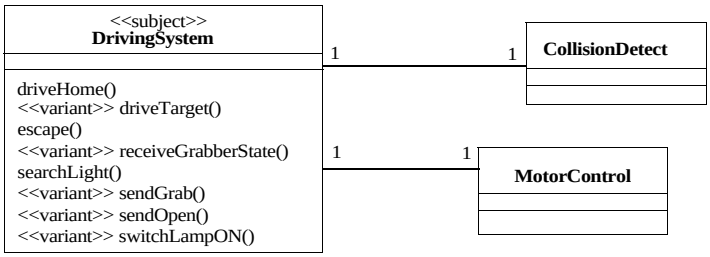


Abb. 5-8: „DrivingSystem“ *Structural Model*

5.1.2.2 Functional Model

Die in Kapitel 3 für die *Operation Specification* des *Functional Models* neu eingeführten Punkte „Geräte-ID“, „Manipuliert“ und „Misst“ werden beispielhaft anhand der Operation „escape()“ (Tabelle 5-9) erläutert. Diese ist für das Ausweichen nach einer Kollision zuständig. „Manipuliert“ zeigt, dass die Operation „escape()“ die Antriebsmotoren benutzt. Damit sinnvoll ausgewichen werden kann, „misst“ „escape()“ die Werte der Tastsensoren, um die Kollisionsquelle zu bestimmen. Unter „Geräte - ID“ wird notiert, welche Motoren bzw. Tastsensoren betroffen sind. Die „Regeln“ definieren die Zuordnung von Tastsensorwerten zur Kollisionsquelle.

Name	escape()
Beschreibung	Der Roboter weicht nach einer Kollision aus.
Erhält	ID der ausgelösten Tastsensoren
Geräte-ID	- Motor: A, C - Tastsensor: S1, S2, S3
Manipuliert	Antriebsmotoren
Misst	Tastsensoren
Regeln	- S1==true: Kollision vorne rechts - S2==true: Kollision hinten - S3==true: Kollision vorne links
Annahmen	Der Roboter befindet sich im Zustand „escape“ (Mindestens ein Tastsensor wurde ausgelöst.)
Ergebnis	Der Roboter ist dem Hindernis bei Kollision - vorne links: nach rechts - vorne rechts: nach links - hinten: zufällig ausgewichen.

Tabelle 5-9: DrivingSystem: „escape()“ *Operation Specification*

5.1.2.3 Behavioural Model

Als Beispiel wird das *Statechart Diagram* der Komponente „GrabberSystem“ gewählt (vgl. Abb. 5-9). Es verfügt über vier Zustände. „opened“ und „closed“ sind längerfristige Zustände, in denen auf eine Nachricht gewartet wird, während „open“ und „grab“ nur so lange andauern, wie der Greifer zum Öffnen bzw. Schließen benötigt. Auf den ersten Blick erscheinen die zwei gestrichelten Übergänge überflüssig zu sein, da es logisch keinen Sinn macht, einen Greifversuch zu unternehmen, wenn der Greifer geschlossen ist. Die Übergänge sind aber nötig, um eine mögliche Fehlerquelle eingebetteter Systeme abzufangen. So kann beispielsweise durch Verlust von Nachricht 1 (öffnen), die Nachricht 0 (greifen) im Zustand „closed“ empfangen werden. Damit das eingebettete System in einem kontrollierten Zustand bleibt, müssen diese Übergänge im *Statechart Diagram* berücksichtigt werden. An dieser Stelle wird Erfahrung 3 aus dem *Experience Model* benutzt (vgl. Tabelle A-77).

Die beiden Zustände „open“ und „grab“ haben keine Entsprechung innerhalb des Programms. Sie charakterisieren nur den äußeren Zustand des Greifers.

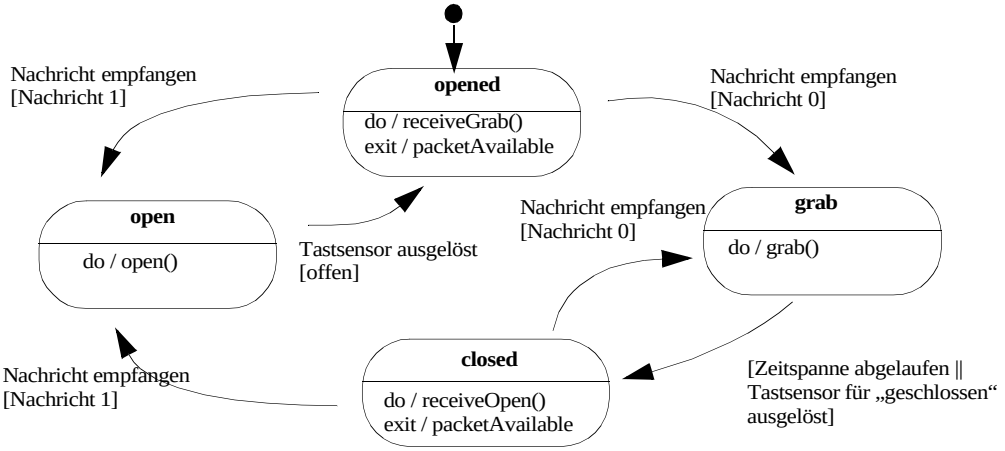


Abb. 5-9: „GrabberSystem“ *Statechart Diagram*

Fazit:

Das Hauptproblem der *Context Realization*, die fehlenden *User Tasks* und *User Interface Activities*, wirkten sich nicht auf die *Komponent Specification* aus. *Structural, Functional, Behavioural* und *Decision Models* ließen sich problemlos aufstellen.

5.1.3 Komponent Realization

Für die *Komponent Realization* wurden in Kapitel 3 keine spezifischen Probleme eingebetteter Systeme identifiziert. Sie wird daher gemäß der Beschreibung in [ATK01] durchgeführt. Die einzige Ausnahme stellt die gesonderte Behandlung von Konstanten dar. Dies ist nicht unbedingt notwendig, kann aber die Übersichtlichkeit des *Structural Models* verbessern. Der Aspekt der Nebenläufigkeit konnte nicht untersucht werden. Zwar besteht der Beispielroboter aus zwei unabhängigen Systemen, diese verhalten sich aber synchron. So macht es wenig Sinn den Greifer zu betätigen, während das Fahrzeug sich bewegt. Ähnlich verhält es sich mit Threads. Im Beispielsystem sind zwar zwei gleichzeitig aktiv, nämlich der Hauptprozess und „SensorListener“. Java übernimmt die komplette Verwaltung der beiden

Prozesse. Daher muss der Entwickler keine besonderen Maßnahmen zu ihrer Synchronisation treffen. Echtzeitverhalten kann ebenfalls nicht untersucht werden, da *leJOS* für Threads keine Vergabe von Prioritäten erlaubt.

5.1.3.1 Structural Model

Als Abstraktionen für die Hardware-Elemente kommen die Klassen „Sensor“, „Motor“ und „Serial“ hinzu. Dies ist auch eine Form von Wiederverwendung, da die Klassen aus den in *leJOS* enthaltenen Bibliotheken „josx.platform.rcx“ stammen. Da durch diese Klassen völlig von der Hardware abstrahiert wird, treten bei der Aufstellung des *Structural Models* (vgl. Abb. 5-10) keine Besonderheiten eingebetteter Systeme auf.

Die Klasse „CollisionDetect“ implementiert das Interface „SensorListener“. Dadurch kann sie selbstständig aktiv werden, wenn eine Kollision auftritt. In diesem Fall wird die Methode „stateChanged()“ aufgerufen, die das Attribut „collision“ setzt.

Die meisten Methoden von „DrivingSystem“ benötigen Konstanten z. B. für Wartezeiten. Diese sind aus Gründen der Übersichtlichkeit durch den Stereotyp <<utility>> in einem separaten Rahmen gruppiert. Implementiert werden sie später als Konstanten der Klasse „DrivingSystem“.

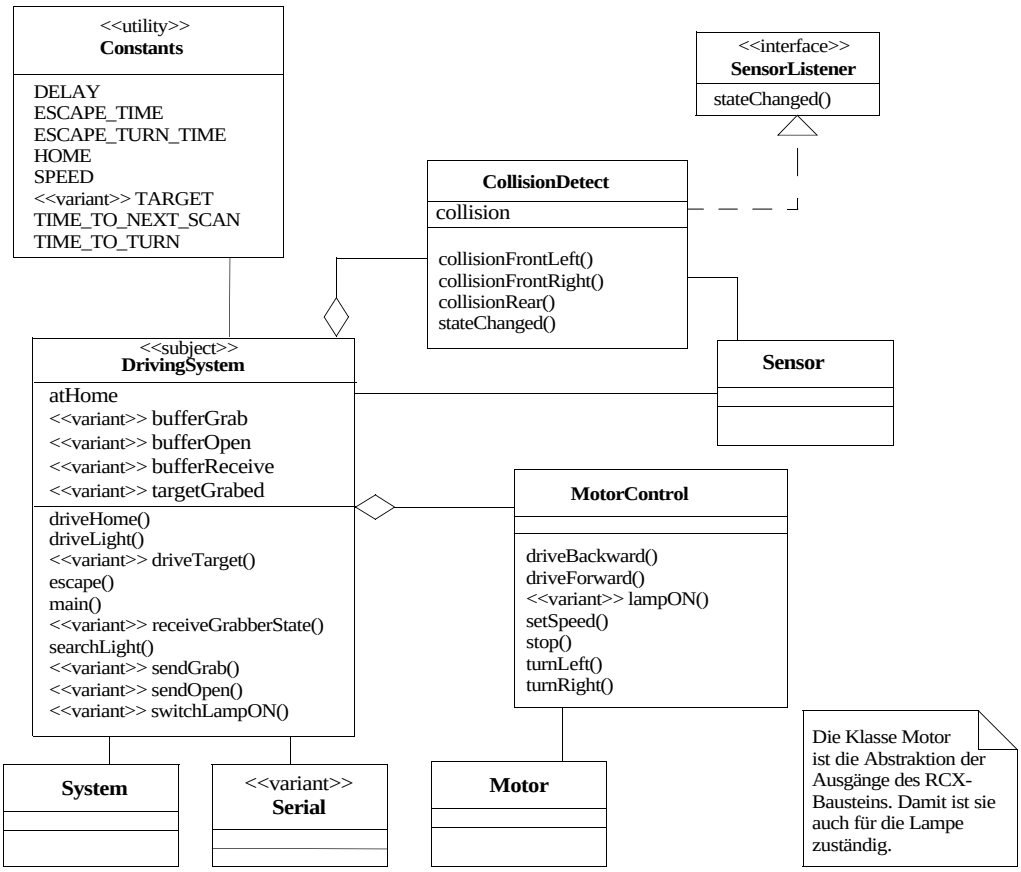


Abb. 5-10: „DrivingSystem“ *Structural Model*

Die Konstanten werden in einer Art *Data Dictionary* (vgl. Tabelle 5-10) aufgeführt und beschrieben. Ein komplettes *Data Dictionary*, welches die Bedeutung aller Methoden und Attribute spezifiziert, wird automatisch durch Javadoc erzeugt und steht als Html-Dokument zur Verfügung.

Name	**Beschreibung**
DELAY	Verzögerung zwischen Drehung und Geradeausfahrt in ms
ESCAPE_TIME	Zeit für die Fahrt in die entgegengesetzte Richtung bei escape() in ms
⋮	⋮

Tabelle 5-10: *Data Dictionary* für die Konstanten der Komponente „DrivingSystem“

5.1.3.2 Activity Model

Auf der Ebene der *Komponent Realization* werden die Modelle soweit verfeinert, dass für das *Activity Model* keine *User Tasks* bzw. *User Interface Activities* mehr betrachtet werden, sondern überwiegend *System Operation Activities*. Diese zeichnen sich dadurch aus, dass sie im Gegensatz zu freien Aktivitäten zu bestimmten Klassen gehören und einen Effekt auf die Daten haben können. [ATK01]

Die fehlenden *User Tasks* bzw. *User Interface Activities* wirken sich somit nicht auf das *Activity Model* aus. Abb. 5-11 zeigt als Beispiel die Methode „main“ der Komponente „DrivingSystem“. Diese ruft nacheinander die Hauptverhaltensweisen auf. Dabei ist zu beachten, dass jederzeit auf eine Kollision reagiert werden kann. Die grau unterlegten Felder sind optional.

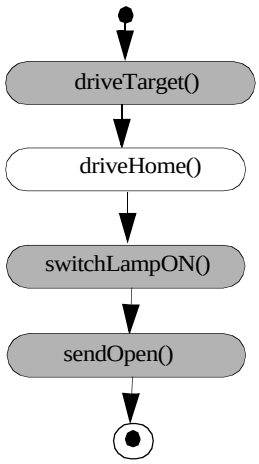


Abb. 5-11: DrivingSystem: „main“ *Activity Diagram*

5.1.3.3 Interaction Model

Im *Interaction Model* wird die Modellierung des dynamischen Verhaltens mit dem strukturellen Aufbau der Komponente kombiniert. Abb. 5-12 zeigt als Beispiel, wie der LEGO-Roboter auf eine Kollision reagiert. Durch das Interface „SensorListener“ hat die Klasse „CollisionDetect“ die Methode „stateChanged()“ geerbt. „stateChanged()“ wird bei jeder Änderung der Sensoren S1, S2, bzw. S3 automatisch aufgerufen. Danach überprüft „CollisionDetect“, ob sich auch der boolesche Wert der Sensoren

verändert hat. Dies ist nötig, um herauszufinden, ob die Licht- oder die Tastsensoren die Wertänderung ausgelöst haben.

Bei der Modellierung der *Interaction Models* traten keine Probleme oder Besonderheiten auf. Vollständig befinden sie sich im Anhang A, für „GrabberSystem“ unter A.3.2.3 und für „DrivingSystem“ unter A.4.2.3.

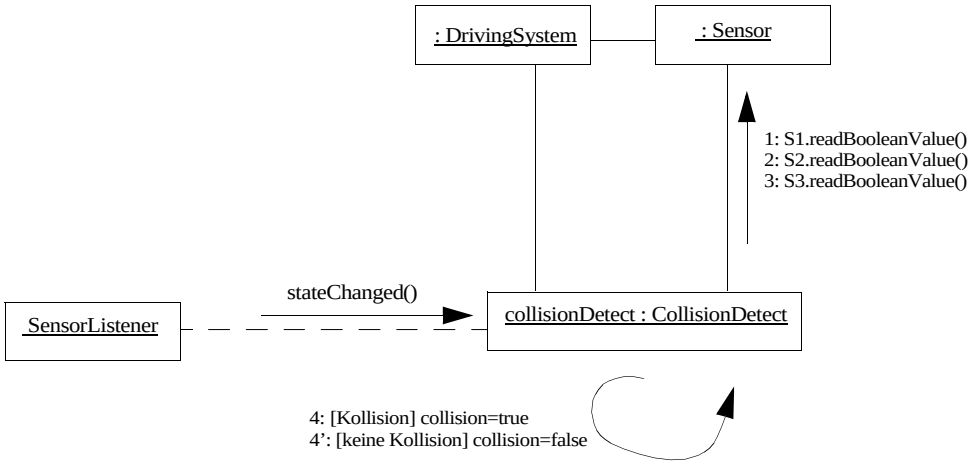


Abb. 5-12: *Collaboration Diagram* der Komponente „DrivingSystem“ für „stateChanged()“

Fazit:

Auf dieser Ebene traten kaum Besonderheiten eingebetteter Systeme auf. Sämtliche Hardwarebestandteile wurden durch die bei *leJOS* mitgelieferten Bibliotheken abstrahiert. Dadurch besteht für den Programmierer kein Unterschied zwischen dem Zugriff auf einen Sensor bzw. Aktor und dem Aufruf einer Methode. Die fehlenden *User Tasks* bzw. *User Interface Activities* hatten ebenfalls keine Auswirkungen.

5.1.4 Reuse

Da eine komplette Neuentwicklung stattfindet, existieren keine selbst entwickelten Komponenten zur Wiederverwendung. COTS Komponenten sind ebenfalls nicht verfügbar. Auf *Design Patterns* kann auch nicht zurückgegriffen werden, da für eingebettete Systeme nur wenige existieren. Allerdings fließen Erfahrungen, die bei der Entwicklung der Komponenten gesammelt wurden, in das *Experience Model* (vgl. A.5) ein. So stehen Erfahrungen, die bei der Entwicklung von *GrabberSystem* (Anhang A.3) gemacht wurden, für die Komponente „DrivingSystem“ zur Verfügung. Im Beispiel wurden Erfahrungen bezüglich des Energieverbrauchs und der Störanfälligkeit der Kommunikation wiederverwendet.

Eine andere Form der Wiederverwendung geschieht durch die von *leJOS* mitgelieferten Bibliotheken. Diese enthalten u. a. Klassen, über die direkt auf die Aktoren und Sensoren zugegriffen werden kann wie z. B. *josx.platform.rcx.Sensor* und *josx.platform.rcx.Motor*.

5.1.5 Komponent Implementation

Die *Komponent Implementation* besteht u. a. aus *Structural Model*, *Component Model* und *Source Code*.

5.1.5.1 Structural Model

Abb. 5-13 zeigt das *Structural Model* der Komponente „DrivingSystem“. Dabei sind zwei Besonderheiten zu beachten. Zum einen hat „DrivingSystem“ Assoziationen zu den Klassen „Motor“, Sensor“, Serial“ und „System“. „DrivingSystem“ erzeugt keine Instanzen von ihnen. Da „Motor“, Sensor“, Serial“ und „System“ *static* sind, kann „DrivingSystem“ trotzdem Methoden aus diesen Klassen aufrufen.

Zum anderen werden die Konstanten auf zwei <<utility>> Felder aufgeteilt. „Constants_A“ enthält die Belegungen für die Produktlinie A, „Constants_B“ hat die passenden Werte für Produktlinie B. Beide <<utility>> Felder sind mit dem Stereotyp <<variant>> versehen. Durch diese Gruppierung ist es möglich, dass die Konstanten je nach der gewählten Produktlinie mit passenden Werten versehen werden. Dazu muss ein passendes *Decision Model* aufgestellt werden (vgl. Tabelle 5-11). Bei der Instantiierung des Frameworks während des *Application Engineerings* wird durch Auflösung dieses *Decision Model* die richtige Belegung gewählt.

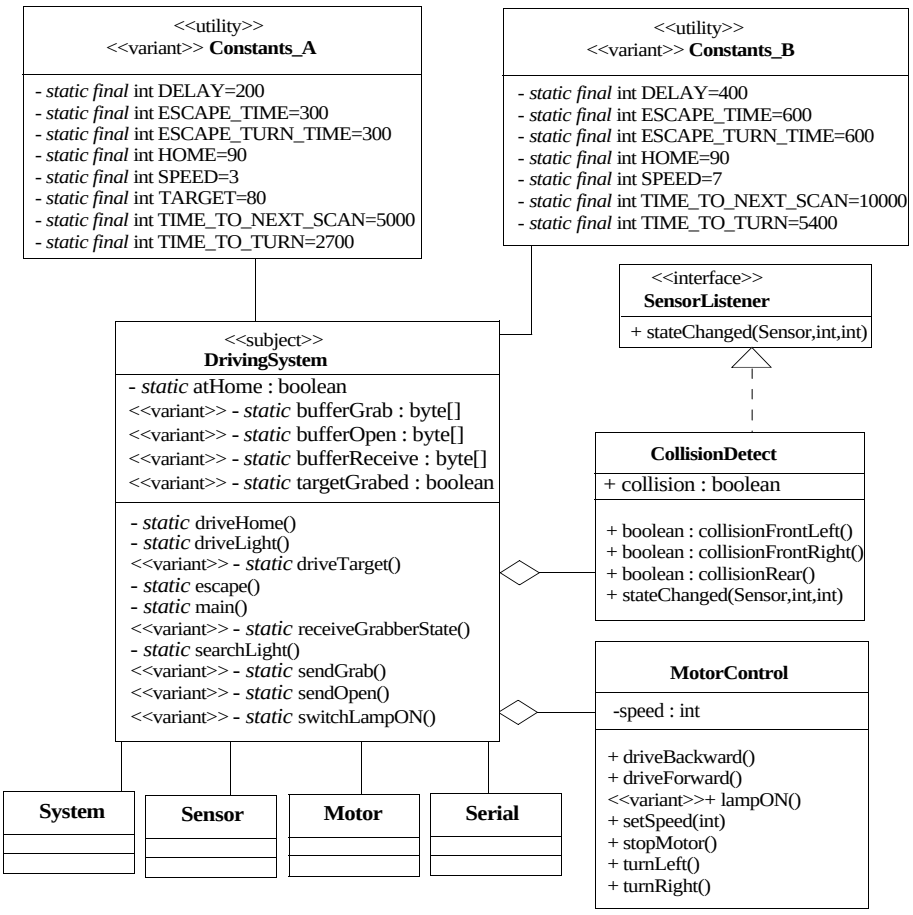


Abb. 5-13: *Structural Model* der Komponente „DrivingSystem“

ID		Frage	Variation Point	Entschluss	Effekt
Structural Model	⋮	⋮	⋮	⋮	⋮
	11	Utility-Klasse *Constants_A* benötigt?	Utility-Klasse *Constants_A*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Utility-Klasse *Constants_A*
	12	Utility-Klasse *Constants_B* benötigt?	Utility-Klasse *Constants_B*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Utility-Klasse *Constants_B*

Tabelle 5-11: *Decision Model* für das *Structural Model* der Komponente „DrivingSystem“

5.1.5.2 Component Diagram

Abb. 5-14 zeigt das *Component Diagram* der Komponente „DrivingSystem“. Bei der Erzeugung sind keine Besonderheiten aufgetreten.

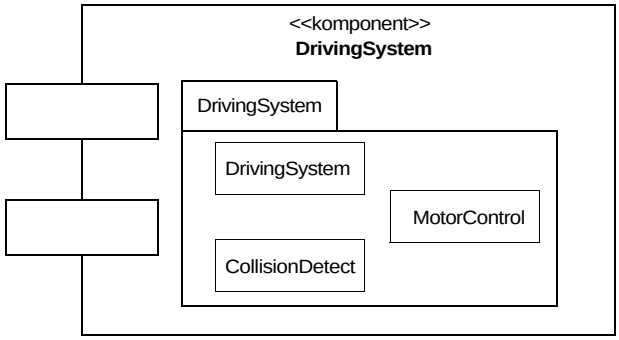


Abb. 5-14: *Component Diagram* der Komponente „DrivingSystem“

5.1.5.3 Source Code

Für den *Source Code* ist bei unserem Beispielsystem zu beachten, dass bei seiner Erzeugung bestimmte Richtlinien eingehalten werden. Diese sind im *Experience Model* vermerkt worden. Beispielsweise muss die main-Methode von „DrivingSystem“ und „GrabberSystem“ *static* sein. Der komplette *Source Code* befindet sich in Anhang C.

Fazit:

Ebenso wie bei der *Component Realization* (vgl. 5.1.1) traten während der *Component Implementation* kaum Besonderheiten eingebetteter Systeme auf. Die meisten der wenigen Auffälligkeiten betreffen das *Structural Model*. Diese sind aber nicht auf eingebettete Systeme beschränkt, sondern können auch in anderen Zusammenhängen auftreten.

5.1.6 Inspection / Measurement of Structural Properties

Die Aktivität *Inspection* findet hier informell statt, da die für eine korrekte Durchführung erforderlichen Inspektionsexperten nicht zur Verfügung standen. *Measurement of Structural Properties* entfällt komplett. Welche möglichen Änderungen und Besonderheiten für eingebettete Systeme erwartet werden, wird in Kapitel 3, Abschnitt 3.1.6 beschrieben.

5.1.7 Testing

Die Aktivität Testing verteilt sich auf *Framework Engineering* und *Application Engineering*. Während des *Framework Engineerings* werden die Testfälle aufgestellt. Die Ausführung und Auswertung der Tests erfolgt während *Application Engineerings*.

Die *Functional Test Cases* korrespondieren mit den *Collaboration Diagrams*. Sie werden daher während der Aktivität *Komponent Specification* aufgestellt. Tabelle 5-12 zeigt als Beispiel die Testfälle für die Operation „driveHome()“ der Komponente „DrivingSystem“.

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „driveHome“.	Kollision	Das Fahrzeug weicht aus.
⋮	⋮	⋮	⋮

Tabelle 5-12: *Functional Test Cases* für die Operation „driveHome()“ der Komponente „DrivingSystem“

Für die Aufstellung der *Structural Test Cases* spielen die *Statechart Diagrams* eine große Rolle. Daher werden die Testfälle parallel zur Aktivität *Komponent Realization* aufgestellt. Als Beispiel wird die Methode „main()“ der Komponente „DrivingSystem“ herangezogen (vgl. Tabelle 5-13). Dabei fällt auf, dass Testfall 1 mit dem Stereotyp <<variant>> versehen ist. Damit sollen die Unterschiede der Produktlinien auch bei den Testfällen berücksichtigt werden. Dies kann generell bei allen Testfällen auftreten, also auch bei den *Functional*, den *State-based* oder den *Environment-based Test Cases*. Im Beispiel gibt es zwei Pfade für „main()“, die je nach realisierter Produktlinie erforderlich sind. Pfad 1 entspricht Produktlinie A, Pfad 2 entspricht Produktlinie B.

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
<<variant>> 1.	main()	Pfad 1: driveTarget()->driveHome()->sendOpen()	sendOpen()
<<variant>> 2.	main()	Pfad 2: driveHome()-> switchLampON()	switchLampON()

Tabelle 5-13: *Structural Test Cases* der Komponente „DrivingSystem“ für das *Collaboration Diagram* zu „main()“

Die *State-based Test Cases* ergänzen die *Functional* und die *Structural Test Cases*. Tabelle 5-14 zeigt einen Ausschnitt der Testfälle für die Komponente „DrivingSystem“.

	Testfall Input			Erwartetes Ergebnis	
ID	Startzustand	Event	Testbedingungen	Aktion	Zustand
1.	driveHome	Kollision	Die Lichtstärke liegt unterhalb der Schwelle für Ziel.	Das Fahrzeug weicht aus.	escape
⋮	⋮	⋮	⋮	⋮	⋮

Tabelle 5-14: *State-based Test Cases* der Komponente „DrivingSystem“

Die *Environment-based Test Cases* sind in Kapitel 3 neu eingeführt worden und sollen die Wirkung äußerer Einflüsse auf das eingebettete System berücksichtigen. Tabelle 5-15 demonstriert anhand des Beispielroboters den Aufbau der Testfälle. Theoretisch müssen alle *Functional*, *Structural* und *State-based Test Cases* für beide Bodenbeläge wiederholt werden. Da das Testen aufwändig ist, lohnen sich in der Praxis Überlegungen, welche Testfälle tatsächlich betroffen sind. Allerdings ist dies nicht immer sofort ersichtlich. Auf den ersten Blick ist beispielsweise die Funktion „sendGrab()“, die nur eine Nachricht sendet, nicht vom Bodenbelag abhängig. Auf den zweiten Blick kann ein stark spiegelnder Boden durch Reflexionen die Kommunikation stören.

ID	Einflussgröße	Wert	Betroffene Testfälle
<<variant>> 1.	Bodenbelag	Teppich	Functional Test Cases: - Alle bis auf Tabelle A-67 (Operation switchLampON()) Structural Test Cases: - Alle bis auf Tabelle A-68 (Collaboration Diagram stateChanged()) State-based Test Cases: - Alle bis auf Testfall 2 und 12 in Tabelle A-74
<<variant>> 2.	Bodenbelag	Tischplatte	
⋮	⋮	⋮	⋮

Tabelle 5-15: *Environment-based Test Cases* der Komponente „DrivingSystem“

Um bei der Instantiierung des Frameworks die überflüssigen Testfälle zu eliminieren, muss ein entsprechendes *Decision Model* aufgestellt werden. Tabelle 5-16 zeigt das *Decision Model* für die Komponente „DrivingSystem“. Es findet eine Unterteilung in die beiden Produktlinien A und B statt. Für alle Testfallarten wird der Effekt notiert, den die Wahl der Produktlinie hat.

	Application A	Application B
Functional Test Cases	entferne: - <<variant>> bei Tabelle A-61 (Functional Test Cases für die Operation driveTarget()) - ...	entferne: - Tabelle A-61 (Functional Test Cases für die Operation driveTarget()) - ...

Tabelle 5-16: *Decision Model* für die Testfälle

	Application A	Application B
Structural Test Cases	entferne: - <<variant>> bei Tabelle A-71 (Structural Test Cases für Collaboration Diagram driveTarget()) - ...	entferne: - Tabelle A-71 (Structural Test Cases für Collaboration Diagram driveTarget()) - ...
State-based Test Cases	entferne in Tabelle A-74: - Testfall 2 - ...	entferne in Tabelle A-74: - <<variant>> Testfall 2 - ...
Environment-based Test Cases	entferne in Tabelle A-75: - Testfall 1 - ...	entferne in Tabelle A-75: - <<variant>> bei Testfall 1 - ...

Tabelle 5-16: *Decision Model* für die Testfälle

Fazit:

Bei der Aufstellung der *Functional*, *Structural* und *State-based Test Cases* traten keine Besonderheiten auf. Für die *Environment Test Cases* werden im Beispiel nur zwei variable Größen unterschieden. In vielen Einsatzumgebungen eingebetteter Systeme werden erheblich mehr Einflussgrößen auftreten. Problematisch kann die Identifikation der Einflussgrößen werden, da nicht immer klar ist, welche einen Effekt auf das System haben. Dies kann zu einer großen Zahl an Testfällen führen. Da an eingebettete Systeme oft hohe Anforderungen an die Funktionssicherheit gestellt werden, lässt sich dies nicht vermeiden.

Bei den Tests des Beispielroboters stellte sich heraus, dass er stark von den Lichtverhältnissen beeinflusst wird.

5.2 Experience Model

Das *Experience Model* wurde in Kapitel 3 neu eingeführt. Im Unterschied zum *Hardware Component Model* eignet es sich nicht nur für den Einsatz bei eingebetteten Systemen. Das *Experience Model* stellt Erfahrungen, die für die weitere Entwicklung des Systems von Bedeutung sind, in einer Tabelle dar. Dies können eigene Erfahrungen sein, aber auch Erfahrungen von anderer Seite, z. B. durch Richtlinien oder Vorschriften für den jeweiligen Kontext. Das Experience Modell soll keine Erfahrungsdatenbank ersetzen, sondern relevante Erfahrungen komprimiert in einer Tabelle auflisten.

Die Wichtigkeit einer zentralen Stelle, an der solche Erfahrungen und Richtlinien gesammelt werden, zeigte sich bei der Entwicklung des Roboters. Bei den Tests traten wiederholt Programmabstürze auf, da durch die fehlende *Garbage Collection* der Speicher zu klein wurde. Dies ist zwar in den ReadMe-Dateien zu leJOS dokumentiert, geriet aber bis zur Implementierung in Vergessenheit. Die Eintragung solcher Informationen in ein *Experience Models* hilft, solche Fehler und den Aufwand zu ihrer Lokalisierung und Behebung zu vermeiden.

Tabelle 5-17 zeigt die für die Entwicklung des Roboters relevanten Erfahrungen. Sie wurden zu verschiedenen Zeitpunkten gemacht und betreffen unterschiedliche Ebenen.

Betroffene Komponenten	Erfahrungen	
Alle Komponenten	**Erfahrung 1**	
	Stichwort	static
	Richtlinie	Die main()-Methode muss als „static“ deklariert werden. Dadurch müssen alle Attribute und Methoden der obersten Klasse ebenfalls „static“ sein.
	Quelle	Dokumentation von leJOS [LEJOS]
	Art	Vorschrift
	Betrifft	Implementierungsebene
	Erfahrung 2 <<variant>>	
	Stichwort	Energieverbrauch
	Beschreibung	Es sollen möglichst wenige Nachrichten verschickt werden, um Energie zu sparen.
	Quelle	Abschnitt A.3.1 in der „Komponent Realization“ der Komponente „GrabberSystem“
	Art	Empfehlung
	Betrifft	Algorithmus, d. h. *Message-*, *Activity-* und *Interaction Models* auf allen Ebenen
	Erfahrung 3 <<variant>>	
	Stichwort	Kommunikation
	Beschreibung	Da das eingebettete System anfällig gegen Störeinflüsse von außen ist, müssen Nachrichten mehrfach versandt werden, um eine erfolgreiche Kommunikation zu gewährleisten.
	Quelle	Abschnitt A.6.3 beim Testen der Komponente „GrabberSystem“
	Art	Empfehlung
	Betrifft	Algorithmus, d. h. *Message-*, *Activity-* und *Interaction Models* auf allen Ebenen, insbesondere die Statechart Diagramme können betroffen sein.
	⋮	⋮ ⋮
Komponente DrivingSystem	**Erfahrung DrivingSystem.1**	
	Stichwort	Lichtsensoren und Tastsensoren
	Beschreibung	An jedem Eingang des RCX-Bausteins sind jeweils ein Licht- und ein Tastsensor gleichzeitig angeschlossen. Dies bewirkt, dass bei Verwendung des Interfaces *SensorListener* ständig Alarm geschlagen wird, da die gemessene Lichtstärke stark schwankt. Es solle daher erst überprüft werden, ob sich auch der boolesche Wert am Eingang geändert hat, bevor das Event weitere Verhaltensweisen auslöst.
	Quelle	Anmerkungen zu Tabelle A-40
	Art	Richtlinie
	Betrifft	Realisierungsebene, Implementierungsebene

Tabelle 5-17: *Experience Model*

Erfahrung 1 stammt aus der Dokumentation zu leJOS. Sie gilt für alle Komponenten und ist bereits während der *Context Realization* bekannt. Dagegen entstand Erfahrung 2 erst während der Entwicklung

der *Komponent Realization* der Komponente „GrabberSystem“. Erfahrung 3 wurde erst während des Testens entwickelt. Die optimale Erfüllung von Erfahrung 2 und 3 ist nicht möglich, weil sie sich etwas wiedersprechen. Da diese Erfahrungen im Gegensatz zu Erfahrung 1 nur den Status einer Empfehlung haben, ist das nicht schlimm. Die Erfahrungen 2 und 3 wurden während der Entwicklung der Komponente „GrabberSystem“ gesammelt. Da sie für alle Komponenten gelten, werden die Erfahrungen für die Entwicklung der Komponente „DrivingSystem“ wiederverwendet. Dagegen betrifft die Erfahrung „DrivingSystem.1“ nur die Komponente „DrivingSystem“. Die Erfahrungen 2 und 3 werden mit dem Stereotyp <<variant>> versehen, da sie nur für die Produktlinie A gelten. Daher ist ein entsprechendes *Decision Model* erforderlich (vgl. Tabelle 5-18). Dabei genügt es, nur die Entscheidungen für die Ebene des *Experience Models* zu modellieren. Entscheidungen für höhere Ebenen werden in das *Decision Model* der *Context Realization* integriert (vgl. Tabelle 5-19).

ID		Frage	Variation Point	Entschluss	Effekt
Experience Model	1	*Erfahrung 2* benötigt?	*Erfahrung 2*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne *Erfahrung 2*
	2	*Erfahrung 3* benötigt?	*Erfahrung 3*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne *Erfahrung 3*

Tabelle 5-18: Decision Model für das Experience Model

ID		Frage	Gegenstand	Entschluss	Effekt
Context Realization	1	Ist das System mit einem Greifer ausgestattet?	Greifer	ja	- Structural Model Entscheidung 1,2,3,5,6:nein - Experience Model Entscheidung 2,3: ja - ...
				nein (default)	- ...
	⋮	⋮	⋮	⋮	⋮

Tabelle 5-19: *Decision Model* auf der Ebene der *Context Realization*

5.3 Application Engineering

Während des *Application Engineerings* findet die Instantiierung des Frameworks statt. Dies geschieht durch Auflösung der *Decision Models*, rekursiv für alle Aktivitäten. Tabelle 5-20 zeigt am Beispiel der *Context Realization* der Produktlinie A, wie durch Auflösung des *Decision Models* die Instantiierung erfolgt.

ID		Frage	Gegenstand	Entschluss
Context Realization	1	Ist das System mit einem Greifer ausgestattet?	Greifer	ja
	2	Ist das System mit einer Lampe ausgestattet?	Lampe	nein

Tabelle 5-20: *Decision Model* auf der Ebene der *Context Realization Instantiation*

Die in Kapitel 3 neu eingeführte Aktivität *Application Experience Modeling* besteht nur aus der Instantiierung des *Experience Models*. Bei den Aktivitäten *Application Context Realization*, *Application Komponent Specification*, *Application Komponent Realization* und *Application Komponent Implementation*, bei denen die Instantiierung des Frameworks erfolgt, treten keine Besonderheiten auf.

Das Produktflussdiagramm des *Application Engineerings* (vgl. Abb. 3-11) enthält genau wie *Framework Engineering* die Aktivität *Testing*. Die Ausführung ist erst während des *Application Engineerings* möglich. Die Testfälle wurden bereits während des *Framework Engineerings* aufgestellt.

Fazit:

Die meisten Aktivitäten korrespondieren direkt mit den entsprechenden Aktivitäten des *Framework Engineerings*. Der wesentliche Unterschied besteht darin, dass die entsprechenden Artefakte durch Auflösung der *Decision Models* instantiiert werden. Da die Besonderheiten eingebetteter Systeme bereits während des *Framework Engineerings* berücksichtigt wurden, waren keine weiteren Modifikationen des *Application Engineerings* nötig.

Die Aktivität *Building* hängt stark von den verwendeten Werkzeugen (RCX-Tools), der Hardware und der Programmiersprache (leJOS) ab. Da KobrA *Building* nur oberflächlich beschreibt, bereitet dies keine Probleme und es sind keine Modifikationen erforderlich. Bei der Verwaltung des Source Codes muss berücksichtigt werden, dass der Code von „GrabberSystem“ und der von „DrivingSystem“ physikalisch getrennt ausgeführt werden.

Kapitel 6

Nachbetrachtung und Ausblick

6.1 Zusammenfassung

Ausgangspunkt dieser Diplomarbeit war die Tatsache, dass die Bedeutung eingebetteter Systeme zunimmt und nur wenige Methoden zur systematischen Software-Entwicklung für diese Systeme existieren. Um ein besseres Verständnis dieser Problemstellung zu erhalten, wurden in Kapitel 1 zunächst spezifische Eigenschaften eingebetteter Systeme untersucht. Danach fand eine Vorstellung verschiedener Methoden zur systematischen Software-Entwicklung statt. Um die vorgestellten Methoden bewerten zu können, wurde eine Liste von Eigenschaften erstellt, die eine für die Entwicklung von eingebetteten Systemen geeignete Methode erfüllen soll. Dabei wurden auch Aspekte wie Systematik und die Unterstützung moderner Wiederverwendungstechnologien berücksichtigt. In der anschließende Bewertung stellte sich KobrA als geeignetste Methode heraus. Da diese Methode noch nicht im Kontext von eingebetteten Systemen eingesetzt wurde, waren diesbezüglich genauere Untersuchungen nötig. Dazu fand in Kapitel 3 zunächst eine theoretische Untersuchung statt, welche Probleme bei der Entwicklung von Software für eingebettete Systeme mit KobrA auftreten können. Außerdem wurden entsprechende Änderungen an KobrA vorgeschlagen, um diese zu lösen.

Um die identifizierten Probleme und Lösungsvorschläge zu evaluieren, wurde als Beispiel für ein eingebettetes System aus LEGO Mindstorms Baukästen ein Roboter konstruiert. Dieser ist in Kapitel 4 beschrieben. Im gleichen Kapitel befinden sich zwei ähnliche Aufgabenstellungen für den Roboter, damit auch der Produktlinienansatz von KobrA untersucht werden konnte. In Kapitel 5 wurde ein kompletter KobrA-Prozess durchgeführt, um die Ergebnisse aus Kapitel 3 zu evaluieren. Probleme, die während der Entwicklung der Robotersoftware auftraten, und ihre Lösungen wurden an Beispielen dokumentiert. Das komplette KobrA-Dokument, das alle im Rahmen des Prozesses erzeugten Artefakte enthält, befindet sich ebenso wie die Bauanleitung für den Roboter und der Source Code im Anhang.

Als eines der Hauptprobleme bei der Entwicklung der Roboter-Software stellte sich der fehlende Startpunkt für die *Context Realization* heraus. Das von KobrA vorgeschlagene *Enterprise Model* ist bewusst unspezifisch, um das Einsatzspektrum nicht einzuschränken. Für eingebettete Systeme eignet sich das *Enterprise Model* trotzdem nicht. Aus diesem Grund wurde das *System Model* eingeführt. Dieses behob auch ein weiteres schwerwiegendes Problem, das sich durch die ganze *Context Realization* zog. So ver-

fügte das Beispielsystem weder über *User Tasks* noch *User Interface Activities*. Diese spielen jedoch eine wichtige Rolle für das *Activity* und das *Interaction Model*. Als Alternative wurden die Prozesse aus der untersten Ebene der *System Process Hierarchy*, die Teil des *System Models* ist, herangezogen. Diese Prozesse gingen in die *Use Cases*, die *Activity Specification* und die *Sequence Diagrams* ein. Da diese Prozesse durch die Methoden des *Structural Models* realisiert werden, wurde dieses ebenfalls von der *System Process Hierarchy* beeinflusst.

Das im Rahmen des *System Models* eingeführte *System Concept Diagram* gibt einen Überblick über den Aufbau des Systems. Im Beispiel wurde die Aufteilung des Systems auf zwei unabhängige Hardwarekomponenten illustriert. Diese stand durch die vorhandene Hardware bereits fest.

Das *Hardware Component Model* unterstützt den Entwickler dabei, einige Besonderheiten eingebetteter Systeme im Auge zu behalten. Im Beispiel waren dies insbesondere die Nachrichtenkommunikation und die Eigenschaften der Aktoren und Sensoren. Besonderheiten, die in anderen Anwendungskontexten auftreten, können leicht in dieses Modell integriert werden. Das *Hardware Component Model* ist nicht zwingend erforderlich. Es dient lediglich dazu, die Eigenarten und Eigenschaften eines eingebetteten Systems in das KobrA-Dokuments zu integrieren. Ziel dabei ist, dass neben dem KobrA-Dokument keine weiteren Dokumente erforderlich sind.

Weitere Modifikation waren bei der *Context Realisation* nicht erforderlich. Die nachfolgenden Aktivitäten wie *Komponent Specification, Realization* und *Implementation* erforderten keine größeren Anpassungen der KobrA-Methode, da die Hauptprobleme bereits während der *Context Realization* gelöst wurden. Lediglich kleinere Anpassungen bezüglich der *Statecharts* und der Tabellen der *Activity Specifications* erwiesen sich als sinnvoll. Diese stellen wiederum nur hilfreiche Erweiterungen dar, die nicht zwingend notwendig sind.

Das neu eingeführte *Experience Model* korrespondierte mit allen Aktivitäten des *Framework Engineerings*. Es dient dazu, Erfahrungen und Vorwissen zentral zu erfassen. Der Einsatz des *Experience Models* ist nicht auf den Einsatz bei eingebetteten Systeme beschränkt. Es kann auch in anderen Kontexten sinnvoll sein.

Im Gegensatz zur Wiederverwendung von Erfahrungen durch das *Experience Model* fand die Wiederverwendung von Komponenten nicht statt. Zum einen war die Aufgabenstellung des Roboters zu klein gewählt, als dass eigene Komponenten zur Wiederverwendung entstanden sind. Zum anderen standen COTS-Komponenten (Commercial Of The Shelf) bzw. Design Patterns zu Beginn des KobrA-Prozesses nicht zur Verfügung. Der Mangel an fertigen Komponenten liegt nicht an KobrA, sondern ist eher ein Problem eingebetteter Systeme. Deren Vielfalt und Spezialisierung erschwert die Wiederverwendung von Komponenten, da die Übertragung einer Komponente von einem System auf ein anderes eventuell größere Anpassungen erfordert.

Ein weiteres Problem von COTS-Komponenten zeigte sich kurz vor Fertigstellung des Roboterprogramms. In der Mailing-Liste zu leJOS [TVMd] wurde die fertige Komponente „Navigator“ vorgestellt, die eine wesentlich größere Funktionalität bietet, als die selbst entwickelte Klasse „MotorControl“. „Navigator“ steht nur als .class File zur Verfügung, d. h. der Source Code ist nicht verfügbar. Als Dokumentation wird nur ein automatisch mit dem Tool Javadoc erzeugtes *Data Dictionary* mitgeliefert. Mit diesen spärlichen Informationen ist die Aufstellung eines vollständigen KobrA-

Dokuments unmöglich. Trotzdem wäre „Navigator“ prinzipiell in Frage gekommen, um die Klasse „MotorControl“ zu ersetzten. Dies scheiterte daran, dass geringfügige Anpassungen erforderlich gewesen waren, die ohne den Source Code nicht möglich waren. Diese Problematik trifft auf viele COTS-Komponenten zu, da deren Hersteller nicht daran interessiert sind, die interne Realisierung der Komponenten und damit das Know How der Firma öffentlich zugänglich zu machen.

Beim *Testing* wurde aus Zeitgründen nur die Aufstellung der Testfälle im Rahmen des *Framework Engineerings* durchgeführt. Dabei wurde eine neue Testfallart, die *Environment-based Test Cases* eingeführt. Diese berücksichtigt, dass eingebettete Systeme stark von der Einsatzumgebung abhängig sein können. Beispielsweise funktioniert die Roboterversion für den Einsatz auf einer Tischplatte nicht auf Teppichboden. Die Ausführung der Testfälle ist nicht Teil des *Framework Engineerings*, sondern gehört zum *Application Engineering*.

Beim *Application Engineering* traten keine Besonderheiten auf, da die wesentliche Aktivität in der Instantiierung des *Frameworks* besteht. Alle anderen Aktivitäten wie *Construction*, *Component Building* und *Releasing* sind allgemein genug gehalten, um problemlos an die jeweiligen Erfordernisse angepasst zu werden. Neu war die Aktivität A*pplication Experience Modeling*. Diese besteht lediglich aus der Instantiierung des *Experience Models*. Bei der Durchführung des *Application Engineerings* fiel außerdem auf, dass auch für die Testfälle ein *Decision Model* erforderlich sein kann. Diesbezüglich werden in der Literatur zu KobrA keine Angaben gemacht.

Fazit:

Prinzipiell ist KobrA für die Entwicklung eingebetteter Systeme geeignet. Die für das Beispiel des LEGO-Roboters an der KobrA-Methode vorgenommenen Modifikationen ließen sich relativ problemlos durchführen. Abgesehen davon, dass die Aufstellung eines *Enterprise Models* bei eingebetteten Systemen naturgemäß nicht möglich ist, war keine der Modifikationen zwingend erforderlich, sondern stellte eher eine sinnvolle Erweiterungen der KobrA-Methode dar.

Die meisten Änderungen betrafen die oberste Ebene des *Framework Engineerings*, die *Context Realization*, da hier die entscheidenden Abstraktionen von der Systemumgebung in die Modellwelt stattfinden. Danach traten größere Besonderheiten eingebetteter Systeme erst wieder während der Implementierung auf. Allerdings können die in Abschnitt 6.2 aufgelisteten offen gebliebenen Fragestellungen auch andere Ebenen des KobrA-Prozesses betreffen.

Während der Entwicklung des LEGO-Roboters gab es Schwierigkeiten bezüglich der Erfüllbarkeit der Aufgabenstellung. So wurde die Software formal korrekt entworfen. Trotzdem versagte der Roboter wegen der schlechten Sensorik. Als Folge musste die Aufgabenstellung dahingehend modifiziert werden, dass die Einsatzumgebung durch Wände abgeschirmt wurde, um Störungen des Roboters durch Streulicht zu unterbinden. Bei einem realen System lässt sich die Einsatzumgebung nicht ohne weiteres modifizieren. Um Situationen zu vermeiden, in denen die Anforderungen nicht erfüllbar sind, können eventuell Machbarkeitsstudien bzw. Tests durchgeführt werden. Die Konstruktion eines Prototyps hilft ebenfalls solche Probleme zu vermeiden.

Der LEGO-Roboter repräsentiert nur ein begrenztes Einsatzspektrum. Die Bandbreite eingebetteter Systeme ist so vielfältig, dass zu erwarten ist, dass in speziellen Situationen eventuell weitere Anpassungen nötig sind. Einige der offen gebliebene Probleme sind in Abschnitt 6.2 beschrieben.

6.2 Ausblick:

Eine Reihe von Fragestellungen und Problemen konnten im Rahmen dieser Diplomarbeit nicht untersucht werden und sind offen geblieben, da die Aufgabenstellung wegen der beschränkten Leistungsfähigkeit des LEGO-Roboters nicht zu umfangreich werden durfte.

So konnten mangels geeigneter Komponenten und Design Patterns zur Komponenten-Wiederverwendung nur theoretische Überlegungen angestellt werden. Wünschenswert ist eine Sammlung von Wiederverwendungskandidaten für eingebette Systeme. Erschwert wird dies durch die große Variabilität eingebetteter Systeme. Manche Komponenten sind so speziell, dass sie nur im Zusammenspiel mit einer bestimmten Hardware funktionieren. Diese Informationen müssen bei der Bewertung von Wiederverwendungskandidaten bekannt sein. Ein anderer Aspekt stellt die gemeinschaftliche Wiederverwendung von Hard- und Softwarekomponenten dar, beispielsweise wenn Hard- und Software parallel entwickelt werden.

Wegen der begrenzten Aufgabenstellung konnte die rekursive Entwicklung von Komponenten nicht untersucht werden. So trat der Fall, dass eine Komponente aus Komponenten zusammengesetzt ist, im Beispielsystem nicht auf. Dies hat zur Folge, dass der *Containment Tree* sehr einfach aufgebaut ist und Aktionen wie *Flattening* keine weitere Vereinfachung des Baumes zur Folge hatten. Dabei werden zwar keine Änderungen erwartet, trotzdem muss dies evaluiert werden.

Völlig ausgeklammert wurde auch der Aspekt der Nebenläufigkeit. Der Beispielroboter besteht zwar aus den beiden unabhängigen Komponenten „GrabberSystem“ und „DrivingSystem“. Diese verhalten sich aber synchron, d. h. die beiden Komponenten warten aufeinander. Nebenläufigkeit ist keine Eigenart eingebetteter Systeme, sondern kann auch in anderen Anwendungsbereichen auftreten.

Ähnlich verhält es sich mit Threads. Im Beispielsystem sind zwar zwei gleichzeitig aktiv, nämlich der Hauptprozess und „SensorListener“. Da Java die komplette Verwaltung der beiden Prozesse übernimmt, muss der Entwickler keine besonderen Maßnahmen zu ihrer Synchronisation treffen.

Measurement of Structural Properties ist komplett entfallen. Diesbezüglich sind weitere Untersuchungen erforderlich. So müssen Metriken, die für andere Zusammenhänge entwickelt werden, nicht unbedingt für eingebettete Systeme geeignet sein. Diesbezügliche Erfahrungen sind nicht ungeprüft übertragbar. Beispielsweise ist *Size* bei eingebetteten Systemen häufig eine kritische Größe. Daher kann es sinnvoll sein eine höhere Kopplung zu erlauben, um *Size* zu minimieren. Es können aber auch völlig neue Größen auftreten, die gemessen werden können. Daher müssen für eingebettete Systeme spezielle Metriken entwickelt werden.

Echtzeitaspekte und besondere Anforderungen an die Ausfallsicherheit von eingebetteten Systemen wurden nicht untersucht. Beide können besondere Maßnahmen bei der Entwicklung, der Inspektion und den Tests erfordern.

Inspection wurde für das Beispielsystem nur informell durchgeführt. Bei einer korrekten Durchführung dieser Aktivität werden keine Besonderheiten erwartet. Es kann aber sinnvoll sein, dass sie von Experten durchgeführt wird, die mit eingebetteten Systemen vertraut sind.

Die Aktivität *Testing* wurde nur teilweise durchgeführt. Es wurden nur die Testfälle aufgestellt. Die Ausführung ist entfallen. Für die Ausführung werden die in 3.1.7 beschriebenen Probleme erwartet.

Dies muss noch evaluiert werden. Da die Aktivität Testing bei KobrA nur allgemein beschrieben wird, stellen Anpassungen kein Problem dar. Allerdings fehlt bei KobrA der Hinweis, dass die Testfälle ein *Decision Model* erfordern können. Auch beschreibt KobrA nur Komponententests und keine Systemtests.

Literaturverzeichnis

[ANA00] M. Anastasopoulos, C. Andriessens, H. Bär, C. Bunse, J. Girad, I. John, D. Muthig, T. Romberg *Überblick Stand der Technik* Fraunhofer IESE / Forschungszentrum Informatik an der Universität Karlsruhe, 2000

[ATK01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel *Component-Based Product Line Engineering with UML* Addison-Wesley, 2001

[BEN99] F. Bensberg, L. Dewanto, M. Klein, C. Reichel *Im Überblick: CASE-Tools für Java* Java Magazin 3/99, Software & Support Verlag, Frankfurt Main

[BOO94] Grady Booch *Object-Oriented Analysis and Design with Applications* Benjamin/Cummings, 1994

[COU97] Steve Courtny, John Laws *Developing Real-Time Distributed Systems with executable Object-Oriented Models* ObjecTime Limited, 1997

[DER95] Kurt W. Derr *Applying OMT. A Practical Step-by-Step Guide to Using the Object Modeling Technique* SIGS Books, 1995

[DOU98] Bruce Powel Douglass *Real-Time UML: Developing Efficient Objects For Embedded Systems* Addison-Wesley, 1998

[DUM00] Uli Dumschat *Echtzeitanwendungen unter RTLinux:Wie schnell ist Echtzeit?* C'T 23/2000, Heise Verlag, Hannover

[ELT00] A. Eltig, W. Huber *Stufenplan - Gute Programme mit CMM und Catalysis* IX 4/2000, Heise Verlag, Hannover

[FOW98] Martin Fowler, Kendall Scott *UML konzentriert* Addison-Wesley, 1998

[GAS00] M. Gasperi, D. Baum, R. Hempel, L. Villa *EXTREME MINDSTORMS An Advanced Guide to LEGO MINDSTORMS* Apress2000

[GOM00] Hassan Gomaa *Designing Concurrent, Distributed, and Real-Time Applications with UML* Adison-Wesley, 2000

[HEU97] Andreas Heuer *Objektorientierte Datenbanken* Adison-Wesley, 1997

[HOL01] Bernhard Hollunder *Friedliche Koexistenz - CORBA und J2EE in unternehmensweiten Anwendungen* Java Magazin 5/01, Software & Support Verlag, Frankfurt Main

[IEEE97] IEEE *IEEE* Standards Collection. *Software Engineering* IEEE, 1997

[IESE00] Joachim Bayer, Dirk Muthig *Library Systems - A KobrA Case Study* Fraunhofer IESE, April 2000

[JAC94] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard *Object-Oriented Software Engineering. A Use Case Driven Approach* Addison-Wesley, 1994

[JAC99] I. Jacobson, G. Booch, J. Rumbaugh *The Unified Software Development Process* Addison-Wesley, 1999

[JAZ00] M. Jazayeri, A. Ran, F. van der Linden *Software Architectur for Product Families* Addison-Wesley, 2000

[JON96] Joseph L. Jones, Anita M. Flynn *Mobile Roboter. Von der Idee zur Implementierung* Addison-Wesley, 1996

[KAS00] Michael Kasper *Robotik-Praktikum WS 00/01* AG Robotik & Prozessrechentechnik, Prof.

E. v. Puttkamer, Universität Kaiserslautern

[KNU99] Jonathan B. Knudsen *The Unofficial Guide to LEGO MINDSTORMS Robots* o'Reilly, 1999

[KRU01] Philippe Kruchten *What is the Rational Unified Process?* the Rational Edge, 2001, http://www.therationaledge.com/content/jan_01/f_rup_pk.html

[LAR98] Craig Larman *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design* Prentice-Hall, Inc., 1998

[LEGO99] LEGO Mindstorms Robotics Invention System 1.5 - Constructopedia The LEGO Group, 1999

[LOU98] D. Louis, P. Müller *Jetzt lerne ich Java. Der einfache Einstieg in die Internet-Programmierung* Markt&Technik Buch- und Software-Verlag GmbH, München 1998

[MUL00a] Frank Müller *Was der Fall ist. Entwicklungswerkzeuge im Vergleich: OEW und OTW*. IX 2/2000, Heise Verlag, Hannover

[MUL00b] Frank Müller *Der zweite Fall. Entwicklungswerkzeuge im Vergleich: objectiF und ROSE* IX 3/2000, Heise Verlag, Hannover

[OES01] B. Oestereich, P. Hruschka, N. Josuttis, H. Kocher, H. Krasemann, M. Reinhold *Erfolgreich mit Objektorientierung* Oldenbourg, 2001

[OMG01] OMG *Unified Modeling Language Specification* OMG 2001

[OSW00] Michael Oswald, Oliver Diedrich *Linux für die Waschmaschine: Im Embedded-Dschungel* C'T 23/2000, Heise Verlag, Hannover

[PUT99] E. v. Puttkammer *Autonome Mobile Roboter* Arbeitsgruppe Robotik und Prozessrechentechnik, Fachbereich Informatik, Universität Kaiserslautern, 1999

[REC97] P. Rechenberg, G. Pomberger *Informatik-Handbuch* Carl Hanser Verlag,1997

[RIN01] Tim Rinkens *RCXTools for leJOS* http://rcxtools.sourceforge.net/

[ROM97] D. Rombach *Software Engineering I* Arbeitsgruppe Software Engineering, Fachbereich Informatik, Universität Kaiserslautern,1997

[ROM98] D. Rombach *Praktikum Software Engineering I* Arbeitsgruppe Software Engineering, Fachbereich Informatik, Universität Kaiserslautern,1998

[ROM99] D. Rombach *Software Engineering II* Arbeitsgruppe Software Engineering, Fachbereich Informatik, Universität Kaiserslautern,1999

[RUM91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson *Object-Oriented Modeling and Design* Prentice Hall, 1991

[SCH01] G. Schiedermeier *Praktikum Lego-Mindstorms: RCX-Innere Bauteile* http://www.if-landshut.de/~gschied/dvt/slide0235.html

[SEE00] Jochen Seemann, Jürgen Wolff von Gudenberg *Software Entwurf mit UML* Springer-Verlag, 2000

[SEI99] Uwe Seimet *CORBA, übernehmen Sie! Services: Basisdienste für den ORB* IX 1/99, Heise Verlag, Hannover

[SEL94] Bran Selic, Garth Gullekson, Paul T. Ward *Real-Time Object-Oriented Modeling* John Wiley & Sons, Inc., 1994

[SEL98] Bran Selic, James Rumbaugh *Using UML for Modeling Complex Real-Time Systems* , 1998

[SEL] Bran Selic, James Rumbaugh *Using UML to Model Complex Real-Time Architectures* ObjecTime, Limited

[SOUa] Desmond F. D'Souza, Alan C. Wills *PLATINUM White Paper: Catalysis: Next Generation Component-based Development from Object Frameworks* Computer Associates, Website: http://www.platinum.com/products/wp/crc/wp_objfr.htm

[SOUb] Desmond F. D'Souza, Alan C. Wills *PLATINUM WhitePaper: Composing Modeling Frameworks in Catalysis* Computer Associates, Website: http://www.platinum.com/products/wp/crc/wp_modfr.htm

[SOU99] Desmond F. D'Souza, Alan C. Wills *Objects, Components, and Frameworks with UML. The Catalysis Approach* Addison-Wesley, 1999

[STA00a] Michael Stal *Reich der Mitte - Die Komponententechnologien COM+, EJB und „CORBA Components"* OBJEKTspektrum 3/2000, SIGS Conference GmbH

[STA00b] Michael Stal *„Microsoft .NET" - Evolution oder Revolution?* OBJEKTspektrum 6/2000, SIGS Conference GmbH

[STE98] Uwe Steinmüller *Konfektioniert: Softwarekomponenten in Java: Beans* IX 2/1998, Heise Verlag, Hannover

[VER98a] Cornelia Versteegen, Gerhard Versteegen *UML inside. CASE-Markt: Stand der Dinge I*X 9/98, Heise Verlag, Hannover

[VER98b] Gerhard Versteegen *Nicht nur Technik. Drei Methoden für UML: Objectory Process, SEPP/OT, V-Modell* IX 10/98, Heise Verlag, Hannover

[WEG01] Hans Wegener *Erfahrungen mit Roundtrip Engineering in Rational Rose: Keine rosigen Zeiten* IX 1/2001, Heise Verlag, Hannover

[WIL00] Michael Willers *„Microsoft.NET" - das programmierbare Web* OBJEKTspektrum 6/2000, SIGS Conference GmbH

[WIT00] Matthias Withopf *Windows im Umbruch - Software-Dienste statt Windows-Programme* C'T 18/2000, Heise Verlag, Hannover

[ZER99] Klaus Zerbe *Bauplan für Objekte - Entwicklungsprozess* C'T 21/1999, Heise Verlag, Hannover

Links

[BEAN] Sun Microsystems *Enterprise JavaBeans(TM) Technologie* http://www.javasoft.com/products/ejb/index.html

[CAT] *Catalysis.org* Website: http://www.catalysis.org

[COM] Cetus Links - Object-Orientation *Distributed Objects & Components: General Information* http://www.cetus-links.org/oo_ole.html

[CORBA] Object Management Group *Welcome to OMG's CORBA Website* http://www.corba.org/

[GAS] Michael Gasperi *Mindstorms Sensor Input* http://www.plazaearth.com/usr/gasperi/lego.htm

[IESE] Fraunhofer Institut Experimentelles Software Engineering Website: http://www.iese.fhg.de

[KOBRA] Fraunhofer IESE *The KobrA Method* Website: http://www.iese.fhg.de/KobrA/

[LDRAW] LDraw.org *Centralized LDraw Resources* http://www.ldraw.org/

[LEGO] LEGO *LEGO MINDSTORMS* http://www.mindstorms.com

[LEJOS] leJOS *Project Info - leJOS* http://sourceforge.net/projects/lejos

[LEJOSa] leJOS *leJOS API Documentation* http://sourceforge.net/projects/lejos/?

[LPE] LEGO Dacta *Distributor für LEGO-Einzelteile* Technik-LPE GMbH, Eberbach, http://www.technik-lpe.com

[MLCAD] Mike Lachmann *MLCad-Mike's Lego CAD* http://www.lm-software.com/mlcad/

[OBJ] microTOOL *objectiF* http://www.microTOOL.de

[OEW] Innovative Software *Object Engineering Workbench (OEW)* http://germany.isg.de

[OMG] Object Management Group *The Object Management Group* http://www.omg.com

[RAT] Rational Software Website: http://www.rational.com

[STP] Aonix *Software through Pictures (STP)* http://www.aonix.de

[TVMa] TinyVM *Project Info - TinyVM* http://sourceforge.net/projects/tinyvm

[TVMb] TinyVM *API Documentation* http://tinyvm.sourceforge.net/tinyvmapi/index.html

[TVMc] Jose Solorzano *TinyVM: Technical Notes* http://tinyvm.sourceforge.net/technical.html

[TVMd] TinyVM *tinyvm-discussion mailing list* http://lists.sourceforge.net/lists/listinfo/tinyvm-discussion

[UML] Object Management Group *UML Ressource Page* http://www.omg.com/technology/uml/index.htm

[

Anhang A

Systemdokumentation

Die Systemdokumentation enthält eine Problembeschreibung, die Benutzeranforderungen und alle während der Entwicklung des Roboters erzeugten Artefakte. Es wird nach der um die Änderungen aus Kapitel 3 modifizierten KobrA-Methode vorgegangen. Die bei der Erweiterung neu hinzugekommenen Artefakte sind zusätzlich in Kapitel 5 aufgeführt.

A.1 Anforderungen / Spezifikation

A.1.1 Problembeschreibung

Es steht ein fertiger Roboter zur Verfügung. Dieser ist mit einem Greifer, einem vorderen und einem hinteren Stoßfänger zur Kollisionserkennung und einer Lampe ausgestattet. Zur Lokalisierung von Lichtquellen stehen zwei neben dem Greifer parallel angeordnete Lichtsensoren zur Verfügung. Ein weiterer ist senkrecht zum Boden über dem Greifer angebracht. Angetrieben wird der Roboter durch zwei Motoren. Der Greifer besitzt einen eigenen Motor um sich zu öffnen oder zu schließen. Eine genaue Beschreibung des Roboters befindet sich in Anhang B.

Die Einsatzumgebung des Roboters ist 1 Meter mal 1 Meter groß und kann mit verschiedenen Bodenbelägen ausgestattet sein. Begrenzt wird das Feld durch Wände, die zum einen Umgebungslicht abschirmen und zum anderen verhindern, dass der Roboter sein Einsatzgebiet verlassen kann. Der Roboter kann Kollisionen mit den Wänden erkennen und ausweichen. An einer Wand befindet sich das Zielgebiet. Dieses ist durch eine Lampe gekennzeichnet. Wenn die von den Lichtsensoren gelieferten Werte einen bestimmten Schwellenwert überschreiten und eine Kollision gemeldet wird ist der Roboter im Ziel angekommen. Um durch den besseren Kontrast die Lokalisation des Zielgebiets bzw. der Zielobjekte aus Aufgabenstellung 1 zu erleichtern, sollen die Wände möglichst dunkel sein.

Aufgabenstellung A:

Im Feld befinden sich runde beleuchtete Zielobjekte (vgl. B.3). Aufgabe des Roboters ist es, diese einzusammeln und in das durch eine Lampe gekennzeichnete Zielgebiet zu bringen. Dazu ist er mit einem Greifer ausgestattet. Damit es nicht zu Verwechslungen zwischen Zielgebiet und Zielobjekt kommt, darf die Lampe des Zielgebietes erst eingeschaltet werden, wenn der Roboter ein Zielobjekt eingesam-

melt hat. Die Einsatzumgebung befindet sich auf einer glatten Oberfläche, beispielweise einer Tischplatte.

Aufgabenstellung B:

Aufgabe des Roboters ist es einen Weg zum Ziel zu finden und durch Einschalten seiner Lampe zu signalisieren, wenn er angekommen ist. Als Bodenbelag kommt ein glatter Teppichboden zum Einsatz.

A.1.2 Benutzeranforderungen

Nummer			Beschreibung
Anforderungen	Allgemein	BA-1	Es soll ein fertiger Roboter verwendet werden, der mit Kollisionserkennung, Greifer, Antrieb und Lichtsensoren ausgestattet ist.
		BA-2	Die Einsatzumgebung soll 1 Meter mal 1 Meter groß sein und durch schwarze Wände begrenzt werden.
		BA-3	Der Roboter soll Kollisionen erkennen und anschließend dem Hindernis ausweichen können.
		BA-4	In der Einsatzumgebung soll sich ein durch eine Lampe markiertes Ziel befinden.
		BA-5	Der Roboter soll das Ziel finden und aufsuchen können.
		BA-6	Wenn der Roboter das Ziel erreicht hat, soll er stoppen.
	Aufgabe A	BA-A1	Im Feld sollen sich runde beleuchtete Zielobjekte befinden.
		BA-A2	Der Roboter soll die Zielobjekte finden können.
		BA-A3	Der Roboter soll Zielobjekte mit seinem Greifer aufnehmen und wieder loslassen können.
		BA-A4	Der Roboter soll die Zielobjekte ins Ziel bringen.
		BA-A5	Die Beleuchtung des Ziels darf nur eingeschaltet sein, wenn der Roboter ein Zielobjekt aufgenommen hat.
		BA-A6	Der Roboter soll auf glatten Oberflächen funktionieren.
	Aufgabe B	BA-B1	Wenn der Roboter im Ziel angekommen ist, soll er seine Lampe einschalten.
		BA-B2	Der Roboter soll auf Teppichboden funktionieren.

Tabelle A-1: Benutzeranforderungen

A.2 Framework Engineering

Der Anhang enthält aus Platzgründen kein Data Dictionary. Statt dessen werden parallel zur Entwicklung die Funktionsrümpfe erzeugt und entsprechend dokumentiert. Mit Hilfe von *Javadoc* kann dann jederzeit automatisch ein Html-Dokument generiert werden, welches das Data Dictionary enthält.

A.2.1 Context Realization

A.2.1.1 Framework Scope

A.2.1.1.1 Domain

Es steht ein fertiger mobiler autonomer Roboter zur Verfügung, der, nachdem er gestartet wurde, in seiner Einsatzumgebung selbstständig bestimmte Aufgabe erledigen soll.

A.2.1.1.2 Product Lines

- **Roboter A:** Es sollen Objekte eingesammelt und in ein Ziel transportiert werden. Er wird auf glatten Oberflächen eingesetzt.
- **Roboter B:** Der Roboter soll das Ziel finden und aufsuchen. Im Ziel angekommen, schaltet er seine Lampe ein. Eingesetzt wird der Roboter auf Teppichboden.

A.2.1.1.3 Features

Fähigkeiten:

- **Ziel aufsuchen:** Der Roboter sucht das Ziel und fährt hin**.**
- **Kollision erkennen und ausweichen:** Nach einer Kollision weicht der Roboter sinnvoll aus.
- **Zielobjekt aufsuchen:** Der Roboter sucht ein Zielobjekt, fährt hin und greift es.
- **Lampe ein- und ausschalten:** Der Roboter schaltet seine Lampe ein bzw. aus.

Anforderungen:

- Funktioniert auf glatter Oberfläche
- Funktioniert auf Teppichboden

Verwendete Hardwarebaugruppen:

- Der **Antrieb** des Roboters ist durch zwei Antriebsräder realisiert.
- Die **Kollisionssensorik** meldet, wenn der Roboter mit einem Hindernis kollidiert ist.
- Die **horizontale Lichtsensorik** verfügt über zwei parallel zum Boden ausgerichtete Lichtsensoren.
- Die **vertikale Lichtsensorik** verfügt über einen senkrecht zum Boden ausgerichtete Lichtsensor.
- Der **Greifer** kann Zielobjekte greifen und wieder loslassen. Mit seinen Sensoren kann er feststellen, ob der Greifvorgang erfolgreich war.
- Über die **IR-Schnittstelle** kann ein RCX-Baustein mit einem anderen kommunizieren.
- Die **Lampe** kann ein- und ausgeschaltet werden.

A.2.1.1.4 Scope Definition Table

		Roboter A	Roboter B
Fähigkeiten	Ziel aufsuchen	x	x
	Kollision erkennen und ausweichen	x	x
	Zielobjekt aufsuchen und greifen	x	
	Lampe einschalten		x
Anforderungen	Funktioniert auf glatter Oberfläche (Tischplatte)	x	
	Funktioniert auf Teppichboden		x
Verwendete Hardwarebaugruppen	Antrieb	x	x
	Kollisionssensorik	x	x
	horizontale Lichtsensorik	x	x
	vertikale Lichtsensorik	x	
	Greifer	x	
	IR-Schnittstelle	x	
	Lampe		x

Tabelle A-2: *Scope Definition Table* des Roboter Framework

A.2.1.2 System Model

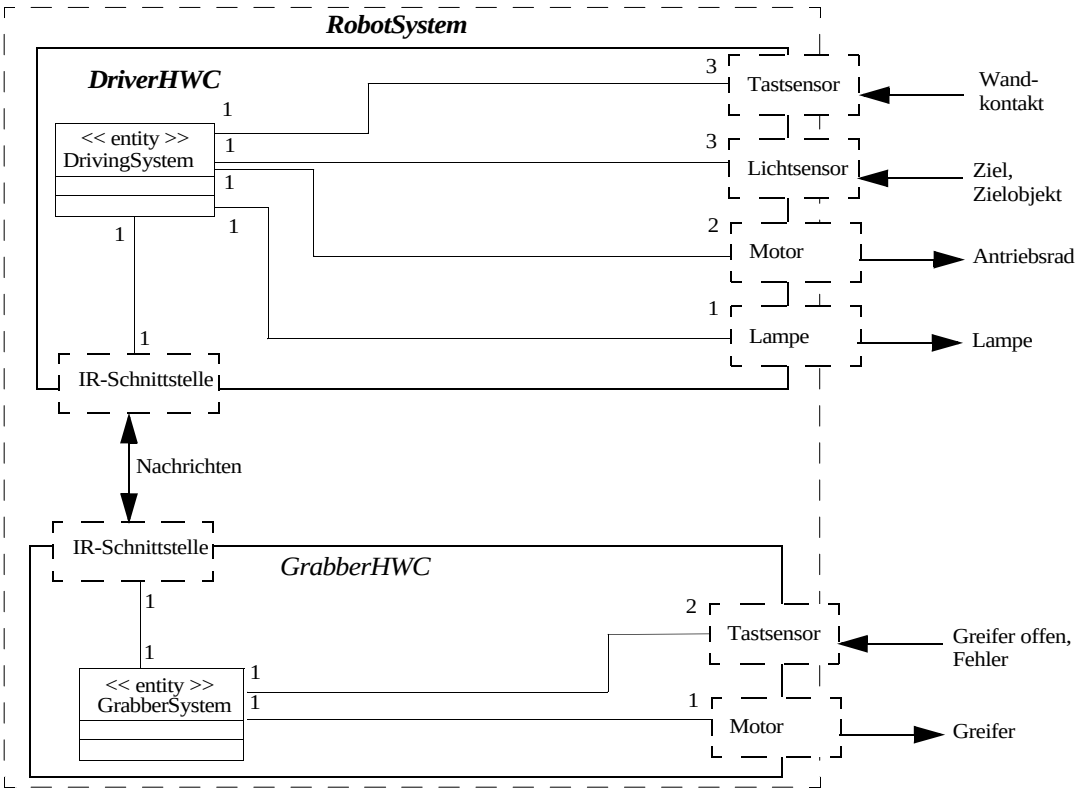


Abb. A-1: RobotSystem System Concept Diagram

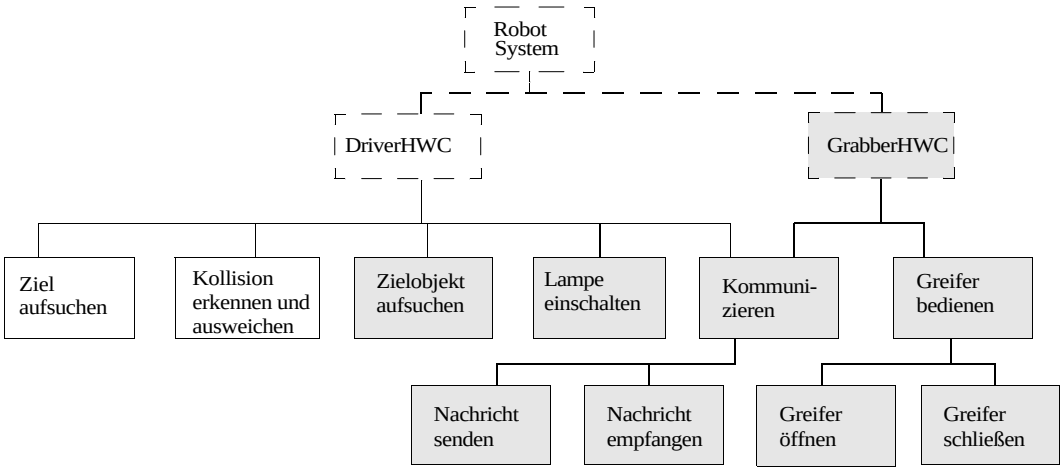


Abb. A-2: RobotSystem Process Hierarchy

Bezeichnung	Typ	gemessene / manipulierte Größe	Wertebereich	Einheit
Tastsensor	Sensor	Hinderniskontakt	int, [0,100]	%
Lichtsensor	Sensor	Helligkeit	int, [0,..,100]	%
Motor	Aktor	Geschwindigkeit	int, [0,..,7]	%
Lampe	Aktor	Lichtstärke	int, [0,..,7]	%

Tabelle A-3: Hardware Properties

A.2.1.3 Structural Model

A.2.1.3.1 Class Diagrams

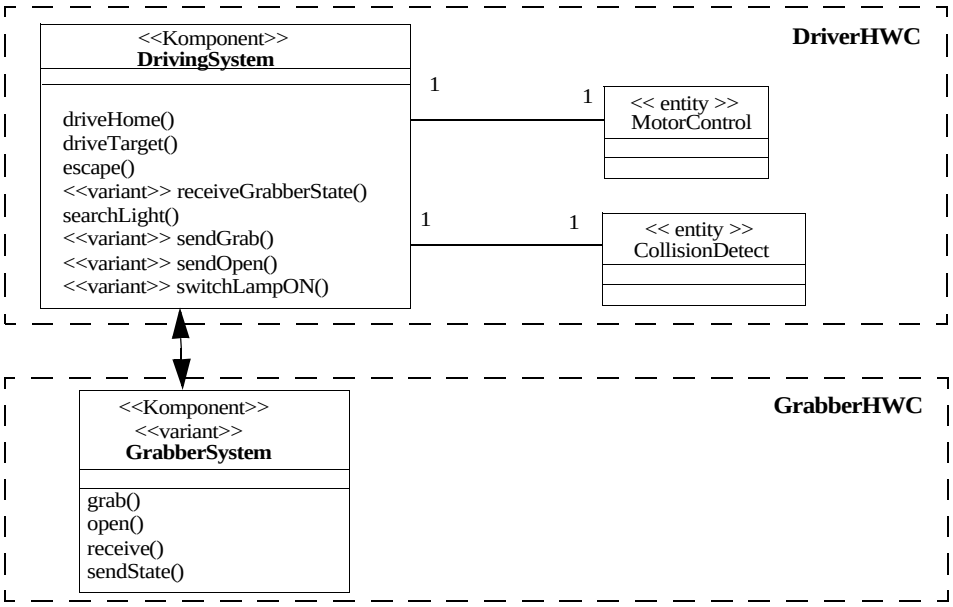


Abb. A-3: Roboter Context Realization Class Diagram

A.2.1.3.2 User Interface Artefacts / Object Diagrams

Entfallen.

A.2.1.4 Hardware Component Model

A.2.1.4.1 Hardware Component Containment Model

Hardware Component	Software Component
GrabberHWC	GrabberSystem
DriverHWC	DrivingSystem

Tabelle A-4: Hardware Component Model

A.2.1.4.2 Message Model

Nachricht	Codierung	Sender	Empfänger	Wirkung
<<variant>> Grab	0	DriverHWC	GrabberHWC	Der Greifer wird geschlossen.
<<variant>> Open	1	DriverHWC	GrabberHWC	Der Greifer öffnet sich.
<<variant>> Failure	2	GrabberHWC	DriverHWC	Empfänger wird über Scheitern des Greifvorgang informiert.
<<variant>> OK	3	GrabberHWC	DriverHWC	Empfänger wird über erfolgreichen Greifvorgang informiert.

Tabelle A-5: Message Model

A.2.1.5 Activity Model

A.2.1.5.1 Activity Diagrams

Für die trivialen Prozesse „Lampe einschalten“, „Nachricht senden“ und „Nachricht empfangen“ werden keine *Activity Diagrams* erzeugt.

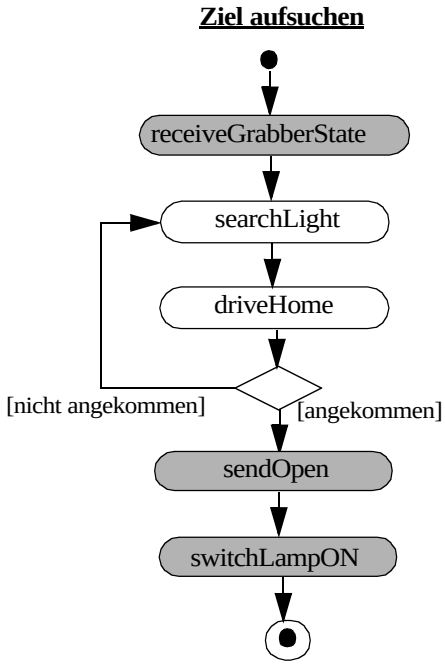


Abb. A-4: „Ziel aufsuchen“ Activity Diagram

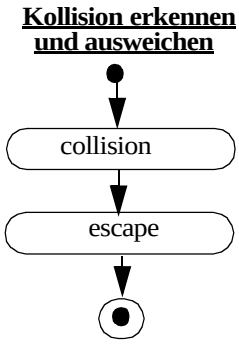


Abb. A-5: „Kollision erkennen und ausweichen“ Activity Diagram

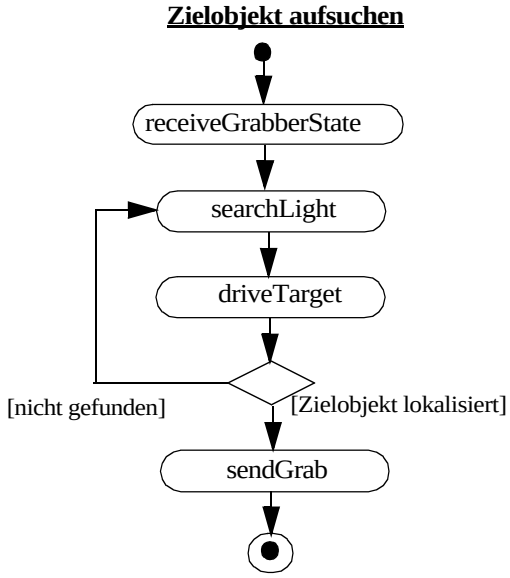


Abb. A-6: „Zielobjekt aufsuchen“ Activity Diagram

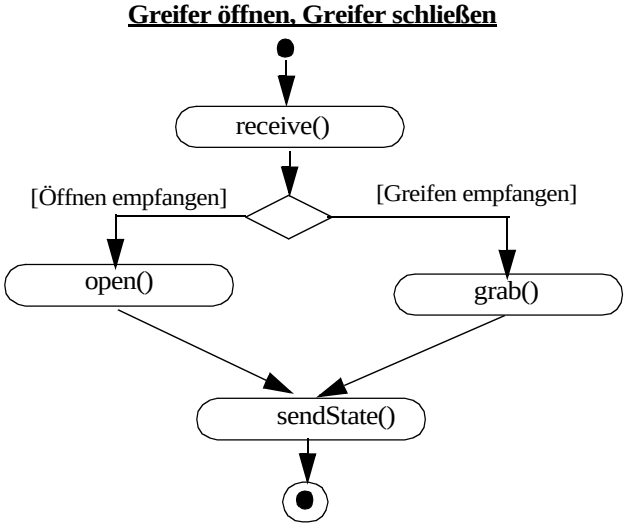


Abb. A-7: „Greifer öffnen“, „Greifer schließen“ Activity Diagram

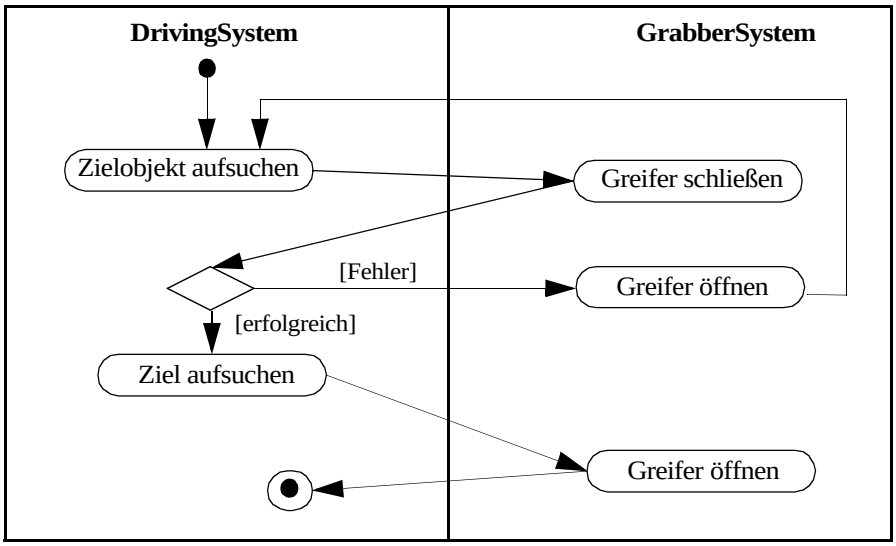


Abb. A-8: Kommunikation Activity Diagram

A.2.1.5.2 Activity Specification

Name	collision()
Beschreibung	Diese Methode ist für die Überwachung der Kollisionssensoren zuständig und stößt selbstständig den Ausweichvorgang an.
HW-Komponente	DriverHWC
Misst	Tastsensoren
Übergibt	ID der ausgelösten Kollisionssensoren
Ergebnis	Der Ausweichvorgang wurde angestoßen.

Tabelle A-6: „collision()“ Activity Specification

Name	driveHome()
Beschreibung	Der Roboter fährt das Ziel an.
HW-Komponente	DriverHWC
Misst	- Lichtsensoren - Tastsensoren
Manipuliert	Antriebsmotoren
Ergebnis	Der Roboter befindet sich im Ziel.
Regeln	Der Roboter befindet sich im Ziel, wenn die Messwerte der horizontalen Lichtsensoren einen Schwellenwert übersteigen und die vorderen Tastsensoren eine Kollision melden.

Tabelle A-7: „driveHome()“ Activity Specification

Name	<<variant>> driveTarget()
Beschreibung	Der Roboter fährt das Zielobjekt an.
HW-Komponente	DriverHWC
Annahmen	- Der Roboter hat kein Zielobjekt gegriffen. - Der Greifer ist offen.
Misst	Lichtsensoren
Manipuliert	Antriebsmotoren
Ergebnis	Der Roboter ist so ausgerichtet, dass sich das Zielobjekt vor dem Greifer befindet.
Regeln	Der Roboter befindet sich vor dem Zielobjekt, wenn der Messwert des vertikalen Lichtsensors einen Schwellenwert übersteigen.

Tabelle A-8: „driveTarget()“ Activity Specification

Name	escape()
Beschreibung	Der Roboter weicht nach einer Kollision aus.
HW-Komponente	DriverHWC
Erhält	ID der ausgelösten Tastsensoren
Annahmen	Mindestens ein Tastsensor wurde ausgelöst.
Misst	Tastsensoren
Manipuliert	Antriebsmotoren
Ergebnis	Der Roboter ist dem Hindernis ausgewichen.

Tabelle A-9: „escape()“ Activity Specification

Name	<<variant>> receiveGrabberState()
Beschreibung	Es wird auf eine Nachricht über den Erfolg des Greifvorgangs gewartet.
HW-Komponente	DriverHWC
Misst	Eingang der IR-Schnittstelle
Empfängt	Nachricht 3 (Objekt gegriffen) oder Nachricht 2 (Objekt nicht gegriffen)
Ergebnis	Nachdem die Nachricht empfangen wurde ist bekannt, ob das Objekt gegriffen wurde oder nicht.

Tabelle A-10: „receiveGrabberState()“ Activity Specification

Name	searchLight()
Beschreibung	Der Roboter wird in Richtung der größten Lichtintensität ausgerichtet.
HW-Komponente	DriverHWC
Misst	Lichtsensoren
Manipuliert	Antriebsmotoren
Ergebnis	Der Roboter ist in Richtung der größten Lichtintensität ausgerichtet.

Tabelle A-11: „searchLight()“ Activity Specification

Name	<<variant>> sendGrab()
Beschreibung	Es wird Nachricht 0 (Greifen) gesendet.
HW-Komponente	DriverHWC
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 0 (Greifen)
Ergebnis	GrabberHWC wurde benachrichtigt.

Tabelle A-12: „sendGrab()“ Activity Specification

Name	<<variant>> sendOpen()
Beschreibung	Es wird Nachricht 1 (Öffnen) gesendet.
HW-Komponente	DriverHWC
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 1 (Öffnen)
Ergebnis	GrabberHWC wurde benachrichtigt.

Tabelle A-13: „sendOpen()“ Activity Specification

Name	<<variant>> switchLampON()
Beschreibung	Die Lampe wird eingeschaltet.
HW-Komponente	DriverHWC
Manipuliert	Lampe
Ergebnis	Die Lampe wurde eingeschaltet.

Tabelle A-14: „switchLampON()“ Activity Specification

Name	<<variant>> grab()
Beschreibung	Der Greifer wird geschlossen.
HW-Komponente	GrabberHWC
Misst	Tastsensor
Manipuliert	Motor
Ergebnis	Der Greifer wurde geschlossen.

Tabelle A-15: „grab()“ Activity Specification

Name	<<variant>> open()
Beschreibung	Der Greifer wird geöffnet.
HW-Komponente	GrabberHWC
Misst	Tastsensor
Manipuliert	Motor
Ergebnis	Der Greifer wurde geöffnet.

Tabelle A-16: „open()“ Activity Specification

Name	<<variant>> receive()
Beschreibung	Es wird auf die Nachricht zum Schließen, bzw. Öffnen des Greifers gewartet.
HW-Komponente	GrabberHWC
Misst	Eingang der IR-Schnittstelle
Empfängt	Nachricht 0 (greifen), bzw. 1 (öffnen)
Ergebnis	Nach Empfang der Nachricht wird die Aktion zum Schließen bzw. Öffnen des Greifers gestartet.

Tabelle A-17: „receive()“ Activity Specification

Name	<<variant>> sendState()
Beschreibung	Es wird eine Nachricht über Misserfolg bzw. Erfolg des Greifvorgangs gesendet.
HW-Komponente	GrabberHWC
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 2 (Misserfolg), bzw. 3 (Erfolg)
Ergebnis	DriverHWC wurde benachrichtigt.

Tabelle A-18: „sendState()“ Activity Specification

A.2.1.5.3 Use Case Model

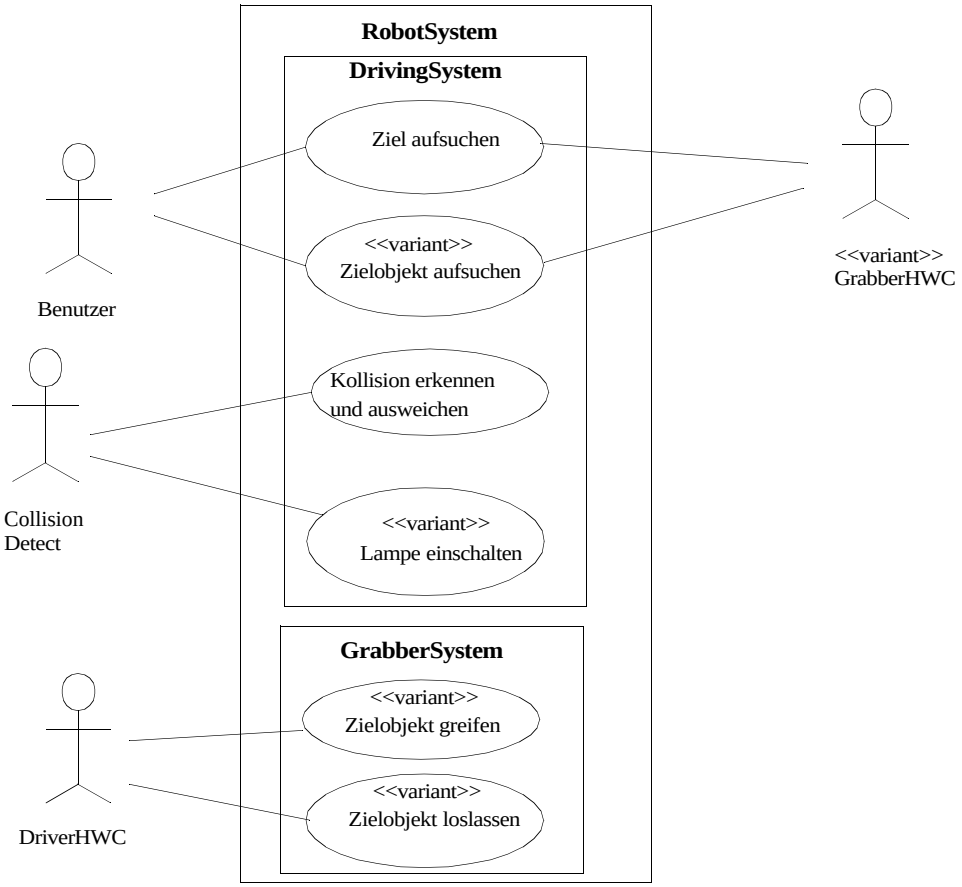


Abb. A-9: Use Cases für das „RobotSystem“

A.2.1.6 Interaction Model

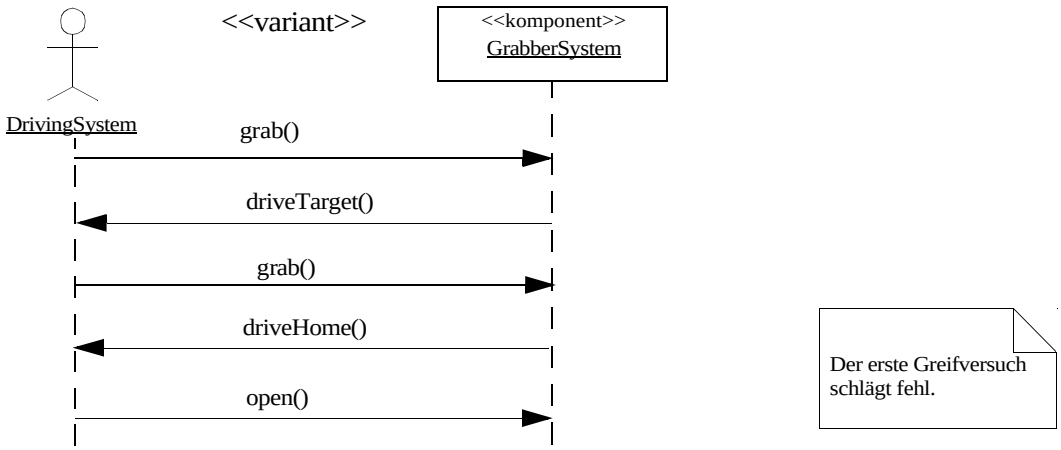


Abb. A-10: „Zielobjekt greifen“, „Zielobjekt loslassen“ Sequence Diagram

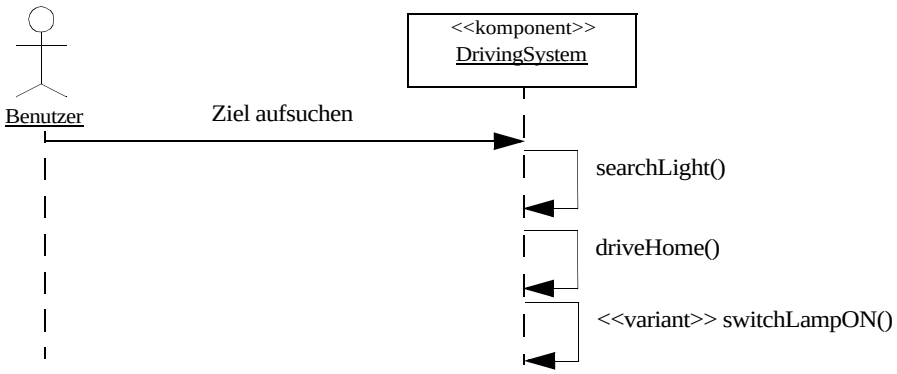


Abb. A-11: „Ziel aufsuchen“, „Lampe einschalten“ Sequence Diagram

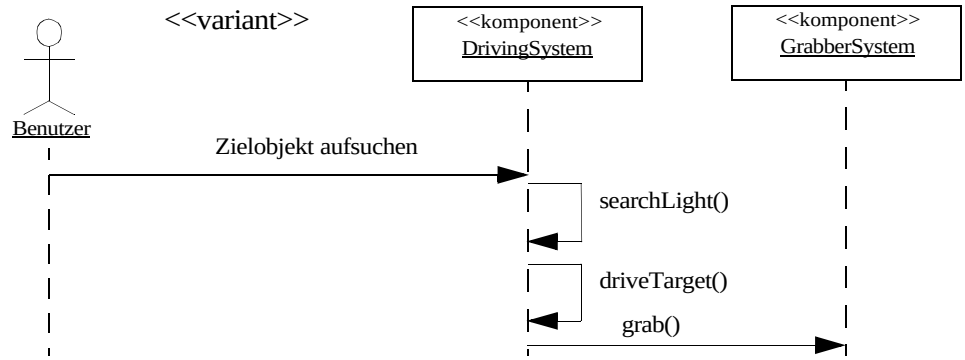


Abb. A-12: „Zielobjekt aufsuchen“ Sequence Diagram

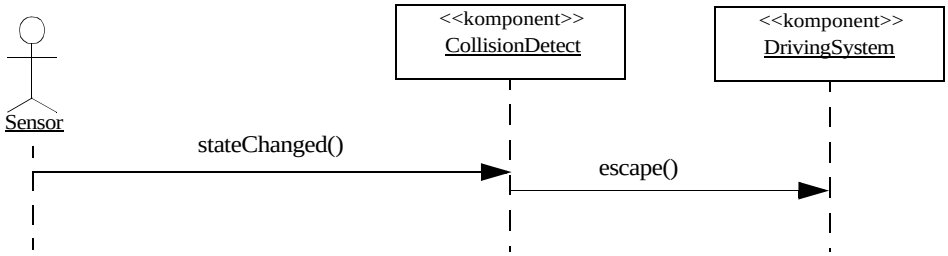


Abb. A-13: „Kollision erkennen und ausweichen“ Sequence Diagram

A.2.1.7 Decision Model

ID		Frage	Variation Point	Entschluss	Effekt
System Process Hierarchy	1	Prozess *Zielobjekt aufsuchen* unterstützt?	Prozess *Zielobjekt aufsuchen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Prozess *Zielobjekt aufsuchen*
	2	Prozess *Lampe einschalten* unterstützt?	Prozess *Lampe einschalten*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Prozess *Lampe einschalten*
	3	Prozess *Nachricht senden* unterstützt?	Prozess *Nachricht senden*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Prozess *Nachricht senden*
	4	Prozess *Nachricht empfangen* unterstützt?	Prozess *Nachricht empfangen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Prozess *Nachricht empfangen*
	5	Prozess *Greifer öffnen* unterstützt?	Prozess *Greifer öffnen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Prozess *Greifer öffnen*
	6	Prozess *Greifer schließen* unterstützt?	Prozess *Greifer schließen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Prozess *Greifer schließen*

Tabelle A-19: Decision Model für die System Process Hierarchy

ID		**Frage**	**Variation Point**	**Entschluss**	**Effekt**
Structural Model	1	Hardwarekomponente *GrabberHWC* benötigt?	Hardwarekomponente *GrabberHWC*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Hardwarekomponente *GrabberHWC*
	2	Operation *driveTarget* benötigt?	Operation *driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *driveTarget*
	3	Operation *receiveGrabberState* benötigt?	Operation *receiveGrabberState*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *receiveGrabberState*
	4	Operation *switchLampON* benötigt?	Operation *switchLampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Operation *switchLampON*
	5	Operation *sendGrab* benötigt?	Operation *sendGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *sendGrab*
	6	Operation *sendOpen* benötigt?	Operation *sendOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *sendOpen*

Tabelle A-20: Decision Model für das Structural Model

ID		Frage	Variation Point	Entschluss	Effekt
Message Model	1	Nachricht *Grab* benötigt?	Nachricht *Grab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Nachricht *Grab*
	2	Nachricht *Open* benötigt?	Nachricht *Open*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Nachricht *Open*
	3	Nachricht *Failure* benötigt?	Nachricht *Failure*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Nachricht *Failure*
	4	Nachricht *OK* benötigt?	Nachricht *OK*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Nachricht *OK*

Tabelle A-21: Decision Model für das Message Model

ID		Frage	Variation Point	Entschluss	Effekt
Use Case Diagram	1	Use-Case *Zielobjekt Aufsuchen* unterstützt?	Use-Case *Zielobjekt Aufsuchen Grabber-HWC*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Use-Case *Zielobjekt Aufsuchen* entferne Assoziationen zu den Actors *Benutzer* und *GrabberHWC*
	2	Use-Case *Lampe einschalten* unterstützt?	Use-Case *Lampe einschalten*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Use-Case *Lampe einschalten* entferne Assoziation zum Actor *CollisionDetect*
	3	Use-Case *Zielobjekt greifen* unterstützt?	Use-Case *Zielobjekt greifen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Use-Case *Zielobjekt greifen* entferne Assoziation zum Actor *Driver-HWC*
	4	Use-Case *Zielobjekt loslassen* unterstützt?	Use-Case *Zielobjekt loslassen*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Use-Case *Zielobjekt loslassen* entferne Assoziation zum Actor *Driver-HWC*

Tabelle A-22: Decision Model für das Use Case Diagram

ID		Frage	Variation Point	Entschluss	Effekt
Sequence Diagram *Zielobjekt greifen, Zielobjekt loslassen*	1	Werden die Prozesse *Zielobjekt greifen* und *Zielobjekt loslassen* benötigt?	Sequence Diagram *Zielobjekt greifen, Zielobjekt loslassen*	ja	entferne Stereotyp *<<variant>>*
				nein (default)	entferne Sequence Diagram *Zielobjekt greifen, Zielobjekt loslassen*

Tabelle A-23: Decision Model für die Sequence Diagrams

ID		Frage	Variation Point	Entschluss	Effekt
Sequence Diagram *Zielobjekt aufsuchen*	2	Wird der Prozess *Zielobjekt aufsuchen* benötigt?	Sequence Diagram *Zielobjekt aufsuchen*	ja	entferne Stereotyp *<<variant>>*
				nein (default)	entferne Sequence Diagram *Zielobjekt aufsuchen*
Sequence Diagram *Ziel aufsuchen, Lampe einschalten*	3	Wird die Nachricht *switchLampON* gesendet?	Nachricht *switchLampON*	ja (default)	entferne Stereotyp *<<variant>>*
				nein	entferne Nachricht *switchLampON*

Tabelle A-23: Decision Model für die Sequence Diagrams

ID		Frage	Gegenstand	Entschluss	Effekt
Context Realization	1	Ist das System mit einem Greifer ausgestattet?	Greifer	ja	- System Process Hierarchy Entscheidung 1,3,4,5,6: nein - Structural Model Entscheidung 1,2,3,5,6:nein - Message Model Entscheidung 1,2,3,4: ja - Experience Model Entscheidung 2,3: ja - Use Case Diagram Entscheidung 1,3,4: ja - Sequence Diagrams Entscheidung 1,2: ja
				nein (default)	- System Process Hierarchy Entscheidung 1,3,4,5,6: nein - Structural Model Entscheidung 1,2,3,5,6:nein - Message Model Entscheidung 1,2,3,4: nein - Experience Model Entscheidung 2,3: nein - Use Case Diagram Entscheidung 1,3,4: nein - Sequence Diagrams Entscheidung 1,2: nein

Tabelle A-24: Decision Model auf der Ebene der Context Realization

ID		Frage	Gegenstand	Entschluss	Effekt
Context Realization	2	Ist das System mit einer Lampe ausgestattet?	Lampe	ja (default)	- System Process Hierarchy Entscheidung 2: ja - Structural Model Entscheidung 4: ja - Use Case Diagram Entscheidung 2: ja - Sequence Diagrams Entscheidung 3: ja
				nein	- System Process Hierarchie Entscheidung 2: nein - Structural Model Entscheidung 4: nein - Use Case Diagram Entscheidung 2: nein - Sequence Diagrams Entscheidung 3: nein

Tabelle A-24: Decision Model auf der Ebene der Context Realization

Feature		Entschluss	Effekt
Fähigkeiten	Zielobjekt aufsuchen und greifen	ja	Context Realization Entscheidung 1: ja
		nein (default)	Context Realization Entscheidung 1: nein
	Lampe einschalten	ja (default)	Context Realization Entscheidung 2: ja
		nein	Context Realization Entscheidung 2: nein
Anforderungen	Funktioniert auf glatter Oberfläche	Bei *Komponent Implementation* der Komponente DrivingSystem realisiert (Tabelle A-59).	
	Funktioniert auf Teppichboden	Bei *Komponent Implementation* der Komponente DrivingSystem realisiert (Tabelle A-59).	

Tabelle A-25: Decision Model für die Abbildung der Features auf die Entscheidungen

A.3 Komponente GrabberSystem

A.3.1 GrabberSystem Komponent Specification

Bemerkung: Da diese Komponente keine Variabilität aufweist, entfallen die *Decision Models*.

A.3.1.1 Structural Model

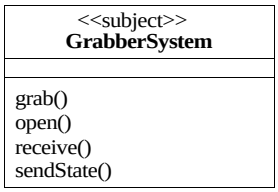


Abb. A-14: Structural Model

A.3.1.2 Functional Model

Name	grab()
Beschreibung	Der Greifer wird geschlossen.
Geräte-ID	- Tastsensor S1 - Motor C
Misst	Tastsensor
Manipuliert	Motor
Regeln	Wenn Sensor S1 ausgelöst hat, war der Greifvorgang erfolglos. Sonst wurde das Objekt erfolgreich gegriffen.
Annahmen	Die Komponente befindet sich im Zustand „opened“ oder „closed“.
Ergebnis	Der Greifer ist vollständig geschlossen oder hat ein Objekt gegriffen.

Tabelle A-26: „grab()“ Operation Specification

Name	open()
Beschreibung	Der Greifer wird geöffnet
Geräte-ID	- Tastsensor S1 - Motor C
Misst	Tastsensor
Manipuliert	Motor
Annahmen	Die Komponente befindet sich im Zustand „opened“ oder „closed“.

Tabelle A-27: „open()“ Operation Specification

Name	open()
Ergebnis	Der Greifer ist geöffnet.

Tabelle A-27: „open()“ Operation Specification

Name	receive()
Beschreibung	Es wird auf eine Nachricht zum Greifen bzw. Öffnen des Greifers gewartet.
Geräte-ID	IR-Schnittstelle
Misst	Eingang der IR-Schnittstelle
Empfängt	Nachricht 0 (greifen), bzw. 1 (öffnen)
Annahmen	Die Komponente befindet sich im Zustand „opened“ oder „closed“.
Ergebnis	Die Warteschleife wurde beendet und die Aktion gestartet.

Tabelle A-28: „receive()“ Operation Specification

Name	sendState()
Beschreibung	Sendet an die *Hardware Component* „DriverHWC“ die Nachricht, über Erfolg bzw. Misserfolg des Greifvorgangs.
Geräte-ID	IR-Schnittstelle
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 2 (Misserfolg), bzw. 3(erfolg)
Annahmen	Die Komponente befindet sich im Zustand „closed“.
Ergebnis	„DrivingSystem“ wurde benachrichtigt.

Tabelle A-29: „sendState()“ Operation Specification

A.3.1.3 Behavioural Model

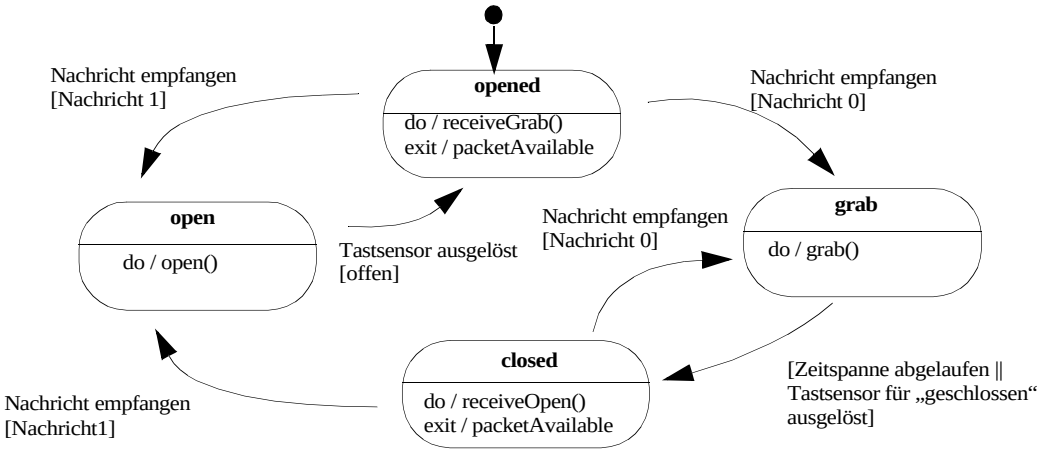


Abb. A-15: „GrabberSystem“ Statechart Diagram

A.3.2 GrabberSystem Komponent Realization

A.3.2.1 Structural Model

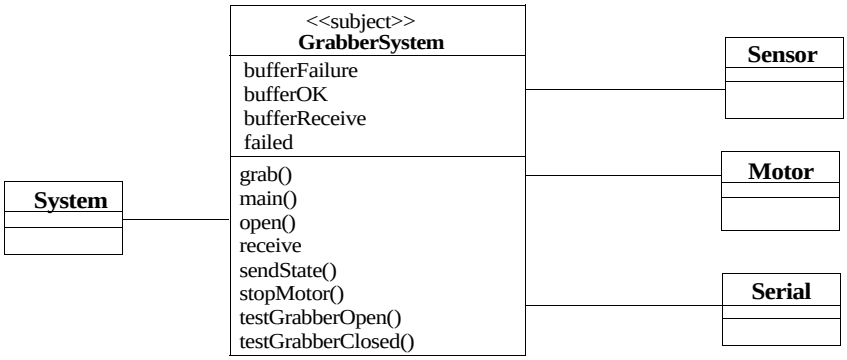


Abb. A-16: Structural Model

A.3.2.2 Activity Model

Die trivialen *Activity Diagrams* für „sendState“, „stopMotor“, „testGrabberClosed“ und „testGrabberOpen“ werden nicht erzeugt.

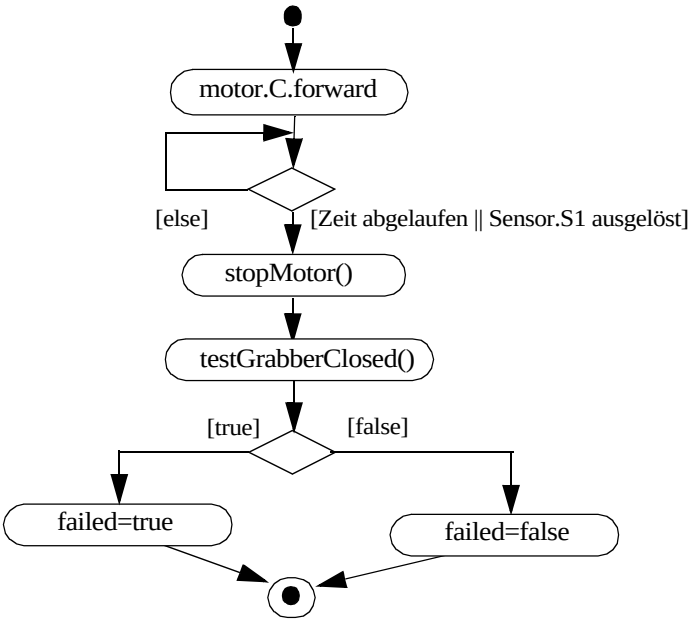


Abb. A-17: „grab“ Activity Diagram

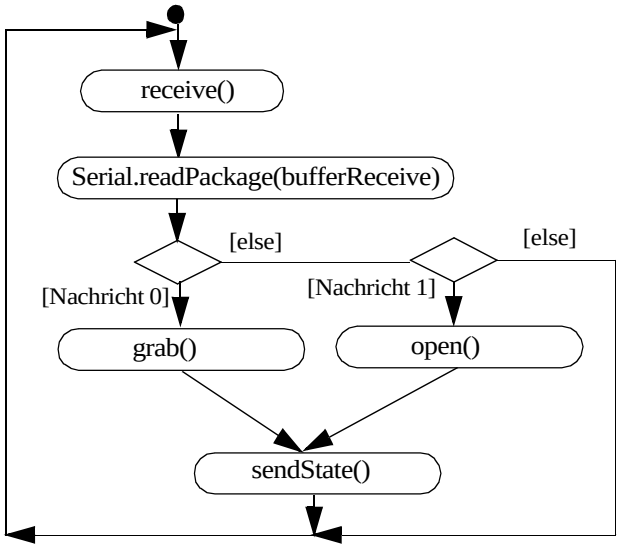


Abb. A-18: „main“ Activity Diagram

Anmerkung:

Um mögliche Nachrichtenverluste abzufangen wird „sendState()“ mehrfach aufgerufen. Daher erscheint der Aufruf von „sendState()“ in der Warteschleife von „receive()“ (vgl. Tabelle A-28) am sinnvollsten. Eine häufige Eigenschaft eingebetteter Systeme sind jedoch begrenzte Energieressourcen. Das ständige Senden von Nachrichten verbraucht zuviel Energie. Als Kompromiss wird jede Nachricht dreimal versendet.

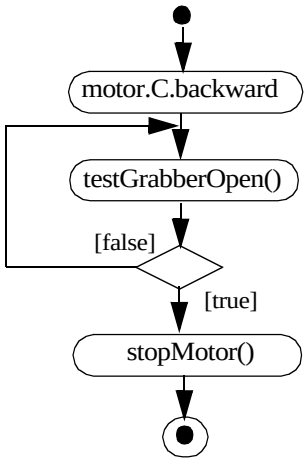


Abb. A-19: „open“ Activity Diagram

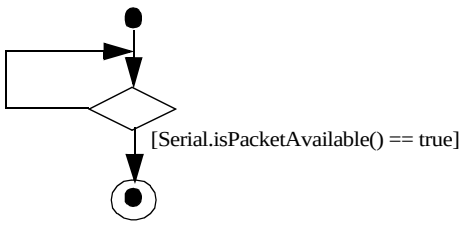


Abb. A-20: „receive“ Activity Diagram

A.3.2.3 Interaction Model

Für die trivialen Aktivitäten *receive()*, *sendState()*, *testGrabberClosed()* und *testGrabberOpen()* werden keine *Collaboration Diagrams* aufgestellt.

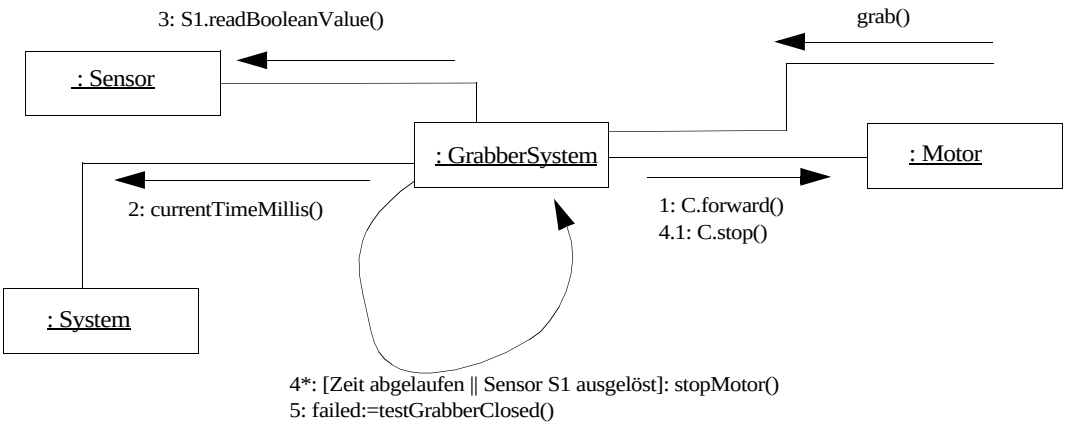


Abb. A-21: Collaboration Diagram für „grab()“

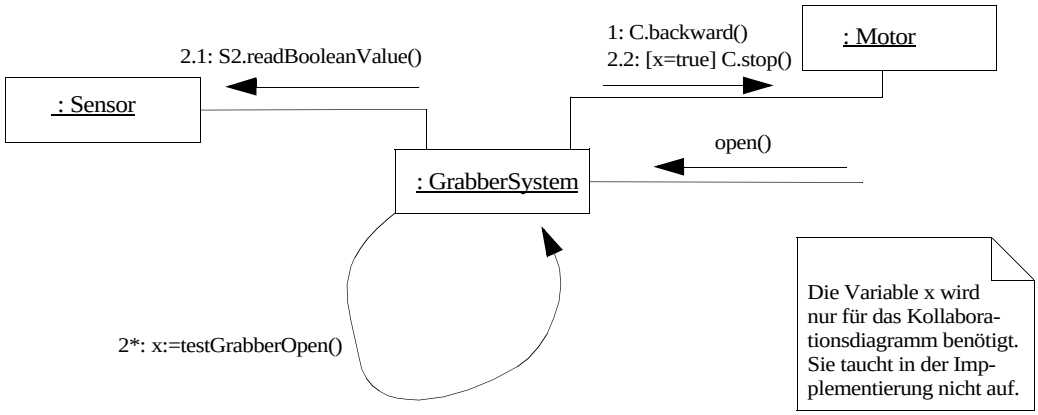


Abb. A-22: Collaboration Diagram für „open()“

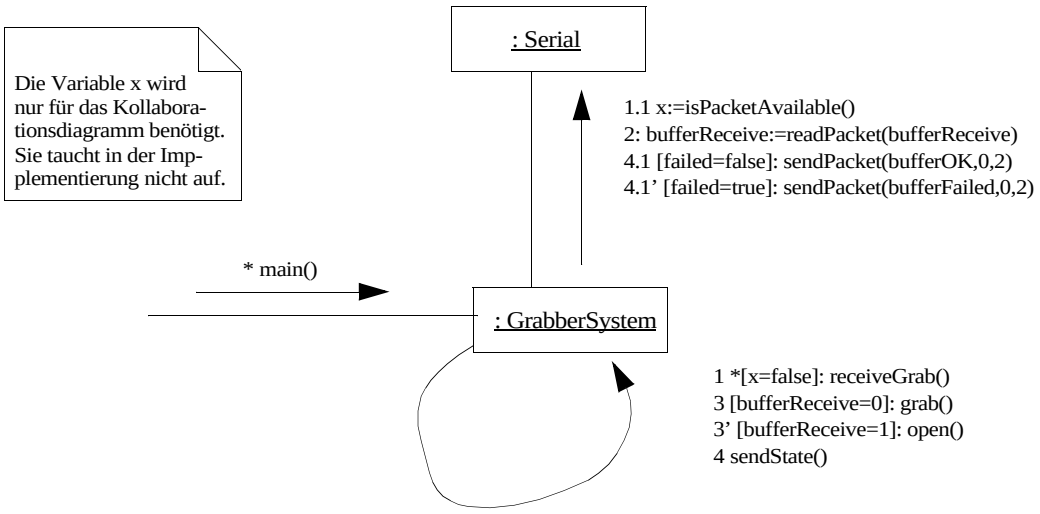


Abb. A-23: Collaboration Diagram für „main()“

A.3.3 Component Reuse

Hat nicht stattgefunden

A.3.4 GrabberSystem Komponent Implementation

Besonderheiten dieses eingebetteten Systems:

- Alle Methoden und Attribute der Hauptklasse „GrabberSystem“ müssen als *static* deklariert werden. Der Grund dafür liegt in den Abstraktionen der Ein- und Ausgänge durch die Klassen *Sensor* und *Motor*. Diese besitzen die Attribute *S1*, *S2*, *S3* für die Eingänge bzw. *A*, *B*, *C* für die Ausgänge. Da die Attribute als *static* deklariert sind, ist der Zugriff nur von Methoden, die ebenfalls *static* sind, möglich. Da außerdem Methoden, die *static* sind, nur Methoden aufrufen können, die *static* sind, müssen alle Attribute und Methoden in „GrabberSystem“ *static* sein.
- Da *leJOS* keine *Garbage Collection* implementiert hat, müssen einige Regeln beachtet werden, um einen Speicherüberlauf zu vermeiden. So muss mit Instantiierungen sparsam umgegangen werden und möglichst auf Rekursion verzichtet werden.

A.3.4.1 Implementation Structural Model

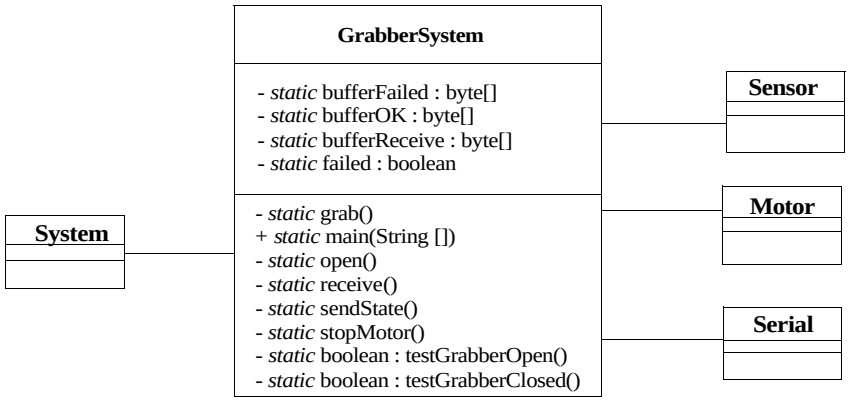


Abb. A-24: Class Diagram

A.3.4.2 Implementation Component Diagram

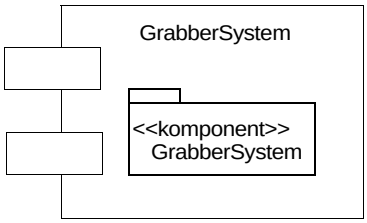


Abb. A-25: Component Diagram

A.3.4.3 Source Code

Der Source Code für die Komponente GrabberSystem befindet sich in Anhang C.1.

A.3.5 Inspection

Wurde informell durch den Autor und den Betreuer dieser Diplomarbeit durchgeführt.

A.3.6 Measurement of Structural Properties

Entfällt.

A.3.7 Testing

A.3.7.1 Functional Test Cases

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „opened“.	Nachricht 0 (Greifen)	Der Greifer schließt sich.
2.	Die Komponente befindet sich im Zustand „closed“.	Nachricht 0 (Greifen)	Der Greifer bleibt geschlossen.

Tabelle A-30: Functional Test Cases für die Operation grab()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „closed“.	Nachricht 1 (Öffnen)	Der Greifer öffnet sich.
2.	Die Komponente befindet sich im Zustand „opened“.	Nachricht 1 (Öffnen)	Der Greifer bleibt offen.

Tabelle A-31: Functional Test Cases für die Operation open()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „opened“.	Nachricht 0 (Greifen)	Aufruf von grab().
2.	Die Komponente befindet sich im Zustand „opened“.	Nachricht 1 (Öffnen)	Aufruf von open().

Tabelle A-32: Functional Test Cases für die Operation receiveGrab()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „closed“.	Nachricht 1 (Öffnen)	Aufruf von open().
2.	Die Komponente befindet sich im Zustand „closed“.	Nachricht 0 (Greifen)	Aufruf von grab().

Tabelle A-33: Functional Test Cases für die Operation receiveOpen()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „grab“.	Tastsensor S2 löst aus bevor die Zeit abgelaufen ist. (Objekt wurde nicht gegriffen).	Aufruf von receiveOpen(). Nachricht 2 (Fehler) wird gesendet.

Tabelle A-34: Functional Test Cases für die Operation sendFailure()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „grab“.	Tastsensor S2 löst nicht aus bevor die Zeit abgelaufen ist. (Objekt wurde gegriffen).	Aufruf von receiveOpen(). Nachricht 3 (Erfolg) wird gesendet.

Tabelle A-35:Functional Test Cases für die Operation sendOK()

A.3.7.2 Structural Test Cases

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	grab()	Pfad: C.forward()->currentTime-Millis()->S1.readBooleanValue()->C.stop()->testGrabberClosed()	failed:=testGrabberClosed()

Tabelle A-36: Structural Test Cases für *Collaboration Diagram* grab()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	open()	Pfad: C.backward()->testGrabberO-pen()->S2.readBooleanValue()->C.stop()	C.stop()

Tabelle A-37: Structural Test Cases für *Collaboration Diagram* open()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	main()	Pfad 1: receive()->grab()->send-State()	open()
2.	main()	Pfad 2: receive()->open()->send-State	open()

Tabelle A-38: Structural Test Cases für *Collaboration Diagram* main()

A.3.7.3 State-based Test Cases

	Testfall Input			Erwartetes Ergebnis	
ID	Startzustand	Event	Testbedingungen	Aktion	Zustand
1.	opened	Nachricht 0 empfangen (Greifen).	Vor dem Greifer ist ein Zielobjekt.	Der Greifer schließt sich, greift das Zielobjekt und sendet Nachricht 3 (Erfolg).	closed
2.	opened	Nachricht 1 empfangen (Öffnen).	keine	Es wird versucht den Greifer zu öffnen. Der Greifer bleibt offen.	opened
3.	opened	Nachricht 0 empfangen (Greifen).	Vor dem Greifer ist kein Zielobjekt.	Der Greifer schließt sich und sendet Nachricht 2 (Fehler).	closed
4.	closed	Nachricht 0 empfangen (Greifen).	Der Greifer hat ein Zielobjekt gegriffen.	Es wird versucht den Greifer zu schließen. Das Zielobjekt bleibt gegriffen und es wird die Nachricht 3 gesendet (Erfolg).	closed
5.	closed	Nachricht 1 empfangen (Öffnen).	Der Greifer hat ein Zielobjekt gegriffen.	Der Greifer öffnet sich und läßt das Zielobjekt los.	opened
6.	closed	Nachricht 0 empfangen (Greifen).	Der Greifer ist vollständig geschlossen (kein Zielobjekt gegriffen).	Es wird versucht den Greifer zu schließen. Der Greifer bleibt geschlossen und es wird die Nachricht 2 gesendet (Fehler).	closed
7.	closed	Nachricht 1 empfangen (Öffnen).	Der Greifer ist vollständig geschlossen (kein Zielobjekt gegriffen).	Der Greifer wird geöffnet.	opened
8.	closed	Nachricht 0 empfangen (Greifen).	Der Greifer ist halb geschlossen und hat das Zielobjekt verloren.	Der Greifer schließt sich vollständig. Es wird die Nachricht 2 gesendet (Fehler).	closed
9.	closed	Nachricht 1 empfangen (Öffnen).	Der Greifer ist halb geschlossen und hat das Zielobjekt verloren.	Der Greifer öffnet sich.	opened

Tabelle A-39: State-based Test Cases

A.4 Komponente DrivingSystem

A.4.1 DrivingSystem Komponent Specification

A.4.1.1 Structural Model

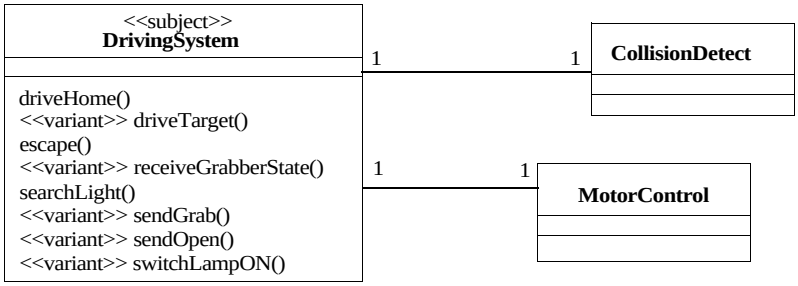


Abb. A-26: Structural Model

A.4.1.2 Functional Model

Name	driveHome()
Beschreibung	Der Roboter fährt das Ziel an.
Geräte-ID	- Motor: A, C - Lichtsensor: S1, S3 - Tastsensor: S1, S3
Misst	- Lichtsensoren - Tastsensoren
Manipuliert	Antriebsmotoren
Regeln	Der Roboter befindet sich im Ziel, wenn die Messwerte der horizontalen Lichtsensoren einen Schwellenwert übersteigen und die vorderen Tastsensoren ein Kollision melden.
Annahmen	Die Komponente befindet sich im Zustand „driveHome“.
Ergebnis	Der Roboter befindet sich im Ziel.

Tabelle A-40: „driveHome“ Operation Specification

Anmerkungen:

Bei Betrachtung der Geräte-IDs fällt auf, dass an jedem Eingang des RCX-Bausteins jeweils ein Licht- und ein Tastsensor gleichzeitig angeschlossen sind. Das Interface *SensorListener* ist nicht zur Überwachung der Lichtsensoren geeignet. Da die gemessene Lichtstärke stark schwankt, wird ständig das Event *stateChanged* gefeuert. Dagegen kann das Interface für die Überwachung von Tastsensoren mit ihren booleschen Werten gut eingesetzt werden.

Da auf Kollisionen sofort reagiert werden muss, lässt sich der Einsatz von *SensorListener* schwer vermeiden. Um die beschriebenen Nachteile zu vermeiden, muss die durch das Event aufgerufene

Methode *stateChanged()* überprüfen, ob sich auch der boolesche Wert am Eingang geändert hat. Erst danach werden weitere Verhaltensweisen ausgelöst.

Name	<<variant>> driveTarget()
Beschreibung	Der Roboter fährt das Zielobjekt an.
Geräte-ID	- Motor: A, C - Lichtsensor: S1, S2, S3
Misst	Lichtsensoren
Manipuliert	Antriebsmotoren
Regeln	Der Roboter befindet sich vor dem Zielobjekt, wenn der Messwert des vertikalen Lichtsensors (S2) einen Schwellenwert übersteigen.
Annahmen	- Die Komponente befindet sich im Zustand „driveTarget“ - Der Greifer ist offen
Ergebnis	Der Roboter ist so ausgerichtet, dass sich das Zielobjekt vor dem Greifer befindet.

Tabelle A-41: „driveTarget“ Operation Specification

Name	escape()
Beschreibung	Der Roboter weicht nach einer Kollision aus.
Erhält	ID der ausgelösten Tastsensoren
Geräte-ID	- Motor: A, C - Tastsensor: S1, S2, S3
Manipuliert	Antriebsmotoren
Misst	Tastsensoren
Regeln	- S1==true: Kollision vorne rechts - S2==true: Kollision hinten - S3==true: Kollision vorne links
Annahmen	Der Roboter befindet sich im Zustand „escape“ (Mindestens ein Tastsensor wurde ausgelöst.)
Ergebnis	Der Roboter ist dem Hindernis bei Kollision - vorne links: nach rechts - vorne rechts: nach links - hinten: zufällig ausgewichen.

Tabelle A-42: „escape()“ Operation Specification

Name	<<variant>> receiveGrabberState()
Beschreibung	Es wird auf eine Nachricht über den Erfolg/Misserfolg des Greifvorgangs gewartet.
Geräte-ID	IR-Schnittstelle
Misst	Eingang der IR-Schnittstelle
Empfängt	Nachricht 3 (Objekt gegriffen) oder Nachricht 2 (Objekt nicht gegriffen)
Annahmen:	Der Roboter befindet sich im Zustand „closeGrabber“ (Nachricht 0 (Greifen) wurde gesendet.)
Ergebnis	Nachdem die Nachricht empfangen wurde ist bekannt, ob das Objekt gegriffen wurde oder nicht.

Tabelle A-43: „receiveGrabberState()“ Operation Specification

Name	searchLight()
Beschreibung	Der Roboter dreht sich auf der Stelle und hält an, wenn er in Richtung der größten Lichtintensität ausgerichtet ist.
Geräte-ID	- Motor A, C - Lichtsensor S1, S3
Misst	Lichtsensoren
Manipuliert	Antriebsmotoren
Regeln	Die Werte der Lichtsensoren S1 und S3 werden addiert.
Annahmen	Der Roboter befindet sich im Zustand „driveHome“ oder „driveTarget“
Ergebnis	Der Roboter ist in Richtung der größten Lichtintensität ausgerichtet.

Tabelle A-44: „searchLight()“ Operation Specification

Name	<<variant>> sendGrab()
Beschreibung	Sendet an die *Hardware Component* „GrabberHWC“ die Nachricht, dass das Zielobjekt gegriffen werden soll.
Geräte-ID	IR-Schnittstelle
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 0 (Greifen)
Annahmen	- Der Roboter befindet sich im Zustand „closeGrabber“ - der Greifer ist offen.
Ergebnis	„GrabberSystem“ wurde benachrichtigt und damit der Vorgang zum Schließen des Greifers angestoßen.

Tabelle A-45: „sendGrab()“ Operation Specification

Name	<<variant>> sendOpen()
Beschreibung	Sendet an die *Hardware Component* „GrabberHWC“ die Nachricht, dass der Greifer geöffnet werden soll.
Geräte-ID	IR-Schnittstelle
Manipuliert	Ausgang der IR-Schnittstelle
Sendet	Nachricht 1 (Öffnen)
Annahmen	- Der Roboter befindet sich im Zustand „openGrabber“ - Greifer ist vollständig geschlossen oder hat ein Zielobjekt gegriffen.
Ergebnis	„GrabberSystem“ wurde benachrichtigt und damit der Vorgang zum Öffnen des Greifers angestoßen.

Tabelle A-46: „sendOpen“ Operation Specification

Name	<<variant>> switchLampON()
Beschreibung	Die Lampe wird eingeschaltet.
Geräte-ID	Motor B
Manipuliert	Lampe
Annahmen	Der Roboter befindet sich im Zustand „switchLampON“ (Der Roboter befindet sich im Ziel.)
Ergebnis	Die Lampe wurde eingeschaltet.

Tabelle A-47: „switchLampON()“ Operation Specification

A.4.1.3 Behavioural Model

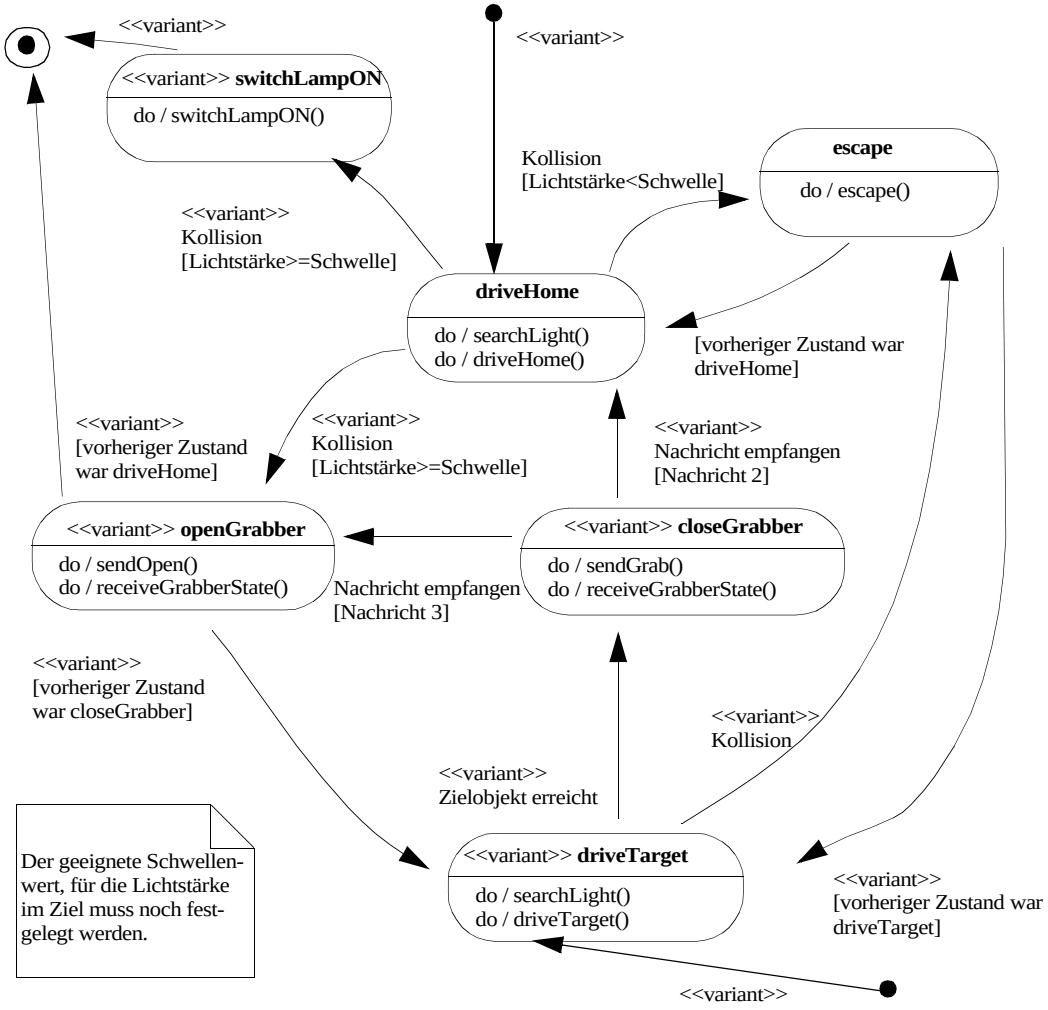


Abb. A-27: DrivingSystem Statechart Diagram

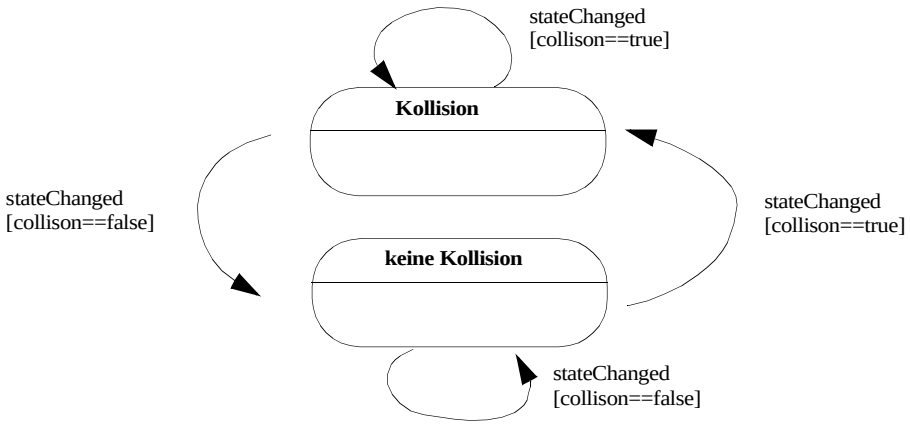


Abb. A-28: CollisionDetect Statechart Diagram

A.4.1.4 Decision Model

ID		Frage	Variation Point	Entschluss	Effekt
Structural Model	1	Operation *DrivingSystem.driveTarget* benötigt?	Operation *DrivingSystem.driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.driveTarget*
	2	Operation *DrivingSystem.receiveGrabberState* benötigt?	Operation *DrivingSystem.receiveGrabberState*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.receiveGrabberState*
	3	Operation *DrivingSystem.sendGrab* benötigt?	Operation *DrivingSystem.sendGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendGrab*
	4	Operation *DrivingSystem.sendOpen* benötigt?	Operation *DrivingSystem.sendOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendOpen*
	5	Operation *DrivingSystem.switchLampON* benötigt?	Operation *DrivingSystem.switchLampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	Operation *DrivingSystem.switchLampON*

Tabelle A-48: Decision Model für das Structural Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
Functional Model	1	Operations-Spezifikation *DrivingSystem.driveTarget* benötigt?	Operations-Spezifikation *DrivingSystem.driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *driveTarget*
	2	Operations-Spezifikation *DrivingSystem.receiveGrabberState* benötigt?	Operations-Spezifikation *DrivingSystem.receiveGrabberState*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *receiveGrabberState*
	3	Operations-Spezifikation *DrivingSystem.sendGrab* benötigt?	Operations-Spezifikation *DrivingSystem.sendGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *sendGrab*
	4	Operations-Spezifikation *DrivingSystem.sendOpen* benötigt?	Operations-Spezifikation *DrivingSystem.sendOpen*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Operation *sendOpen*
	5	Operations-Spezifikation *DrivingSystem.switchLampON* benötigt?	Operations-Spezifikation *DrivingSystem.switchLampON*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *switchLampON*

Tabelle A-49: Decision Model für das Functional Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
Statechart Diagram	1	Wird die Starttransition zu Zustand *driveHome* benötigt?	Starttransition zu Zustand *driveHome*	ja (default)	entferne den Stereotyp <<*variant*>> bei der Transition
				nein	entferne die Starttransition zu Zustand *driveHome*
	2	Wird die Starttransition zu Zustand *driveTarget* benötigt?	Starttransition zu Zustand *driveTarget*	ja	entferne den Stereotyp <<*variant*>> bei der Transition
				nein (default)	entferne die Starttransition zu Zustand *driveTarget*
	3	Wird der Zustand *switchLampON* benötigt?	Zustand *switchLampON*	ja (default)	entferne den Stereotyp <<*variant*>> beim Zustand und allen zugehörgen Transitionen
				nein	entferne Zustand *switchLampON* und alle zugehörigen Transitionen
	4	Wird der Zustand *driveTarget* benötigt?	Zustand *driveTarget*	ja	entferne den Stereotyp <<*variant*>> beim Zustand und allen zugehörgen Transitionen
				nein (default)	entferne Zustand *driveTarget* und alle zugehörigen Transitionen
	5	Wird der Zustand *closeGrabber* benötigt?	Zustand *closeGrabber*	ja (default)	entferne den Stereotyp <<*variant*>> beim Zustand und allen zugehörgen Transitionen
				nein	entferne Zustand *closeGrabber* und alle zugehörigen Transitionen
	6	Wird der Zustand *openGrabber* benötigt?	Zustand *openGrabber*	ja	entferne den Stereotyp <<*variant*>> beim Zustand und allen zugehörgen Transitionen
				nein (default)	entferne Zustand *openGrabber* und alle zugehörigen Transitionen

Tabelle A-50: Decision Model für das Behavioural Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
DrivingSystem Komponent Specification	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	ja (default)	- Structural Model Entscheidung 5: ja - Functional Model Entscheidung 5: ja - Statechart Diagram Entscheidung 1,3: ja
				nein	- Structural Model Entscheidung 5: nein - Functional Model Entscheidung 5: nein - Statechart Diagram Entscheidung 1,3: nein
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	ja	- Structural Model Entscheidung 1,2,3,4: ja - Functional Model Entscheidung 1,2,3,4: ja - Statechart Diagram Entscheidung 2,4,5,6: ja
				nein (default)	- Structural Model Entscheidung 1,2,3,4: nein - Functional Model Entscheidung 1,2,3,4: nein - Statechart Diagram Entscheidung 2,4,5,6: nein

Tabelle A-51: Top Level Entscheidungen für die Decision Models der *DrivingSystem Komponent Specification*

A.4.2 DrivingSystem Komponent Realization

A.4.2.1 Structural Model

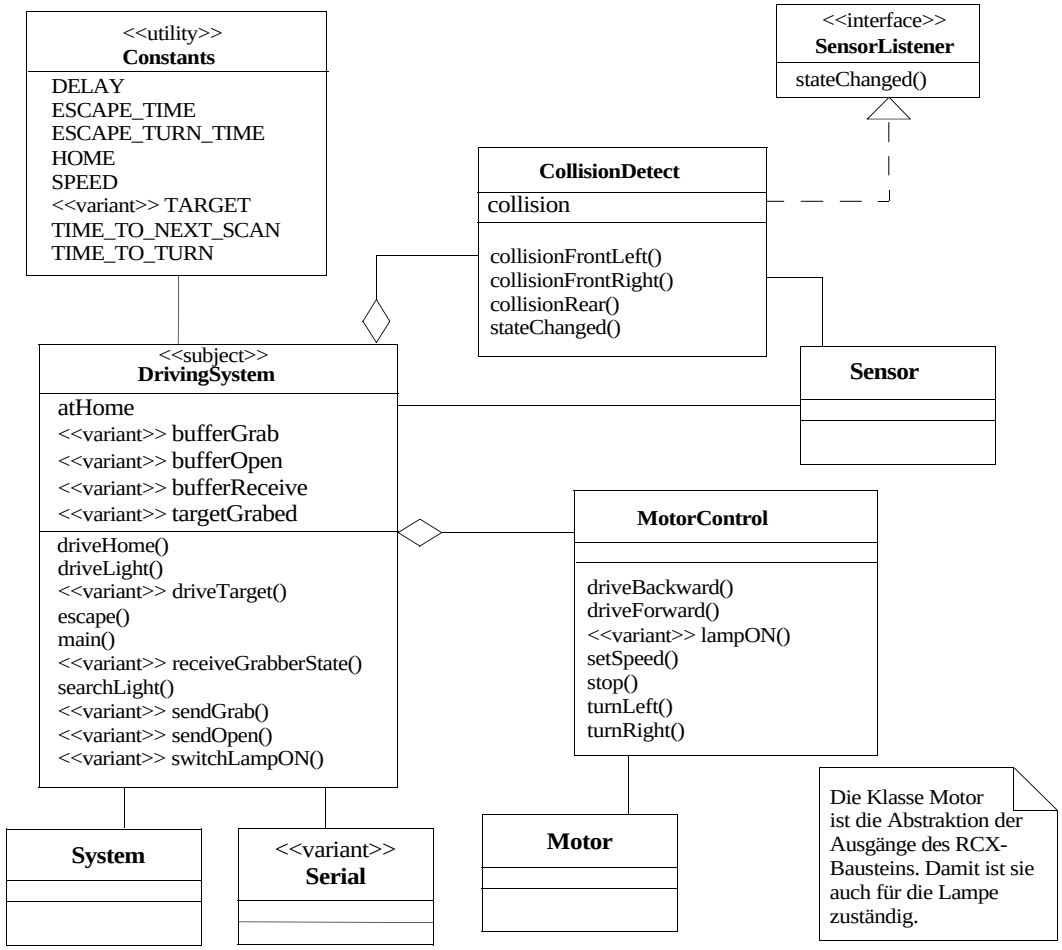


Abb. A-29: Structural Model

Die Konstanten werden in einem *Data Dictionary* erklärt. Die Bedeutung der restlichen Methoden und Attribute ist durch das *Activity Model* definiert. Ein komplettes *Data Dictionary* wurde automatisch durch Javadoc erzeugt und steht als Html-Dokument zur Verfügung.

Name	Beschreibung
DELAY	Verzögerung zwischen Drehung und Geradeausfahrt in ms
ESCAPE_TIME	Zeit für die Fahrt in die entgegengesetzte Richtung bei escape() in ms
ESCAPE_TURN_TIME	Zeit für die Drehung bei escape() in ms
HOME	Lichtstärke im Ziel
SPEED	Geschwindigkeit des Fahrzeugs. Werte zwischen 1 und 7 sind zulässig.
TARGET	Lichtstärke über Zielobjekt
TIME_TO_NEXT_SCAN	Zeit bis zur nächsten Lichtsuche in ms

Tabelle A-52: Data Dictionary für die Konstanten

TIME_TO_TURN	Zeit für eine Umdrehung in ms

Tabelle A-52: Data Dictionary für die Konstanten

A.4.2.2 Activity Model

Für die Aktivitäten von *CollisionDetect* und *MotorControl* werden keine *Activity Diagrams* aufgestellt, da die beiden Klassen nur triviale Aktivitäten beinhalten. Die Methoden *sendOpen()*, *sendGrab()* und *switchLampON()* der Klasse *DrivingSystem* werden ebenfalls nicht modelliert.

Die Methoden von MotorControl setzen die nötigen Parameter für die Motoren und werden danach sofort beendet. Ihre Ausführung verbraucht praktisch keine Zeit. Da die Anweisungen nur Sinn machen, wenn das Fahrzeug auch eine gewisse Zeit nach vorne fährt, muss sicher gestellt werden, dass beispielsweise nach *driveForward()* die nächste Anweisung an die Motoren zeitverzögert erfolgt. Realisiert wird dies durch eine Warteschleife. Um die *Activity Diagrams* möglichst einfach zu halten, wird diese getrennt modelliert (Abb. A-30). Warteschleifen haben den Nachteil, dass sie Rechenzeit verschwenden. In diesem Fall spielt das keine Rolle, da außer der Kollisionserkennung keine anderen Aufgaben anstehen. Die Kollisionserkennung wird über das Interface *SensorListener* realisiert. Dieses läuft in einem eigenen Thread und wird daher von der Warteschleife nicht beeinträchtigt.

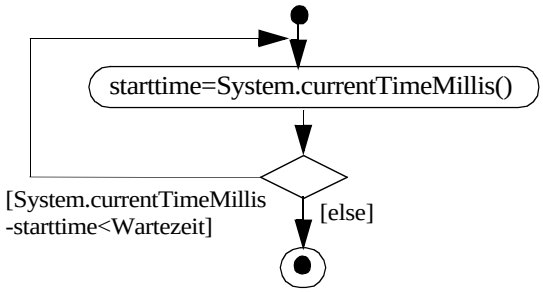


Abb. A-30: Activity Diagram der Warteschleife

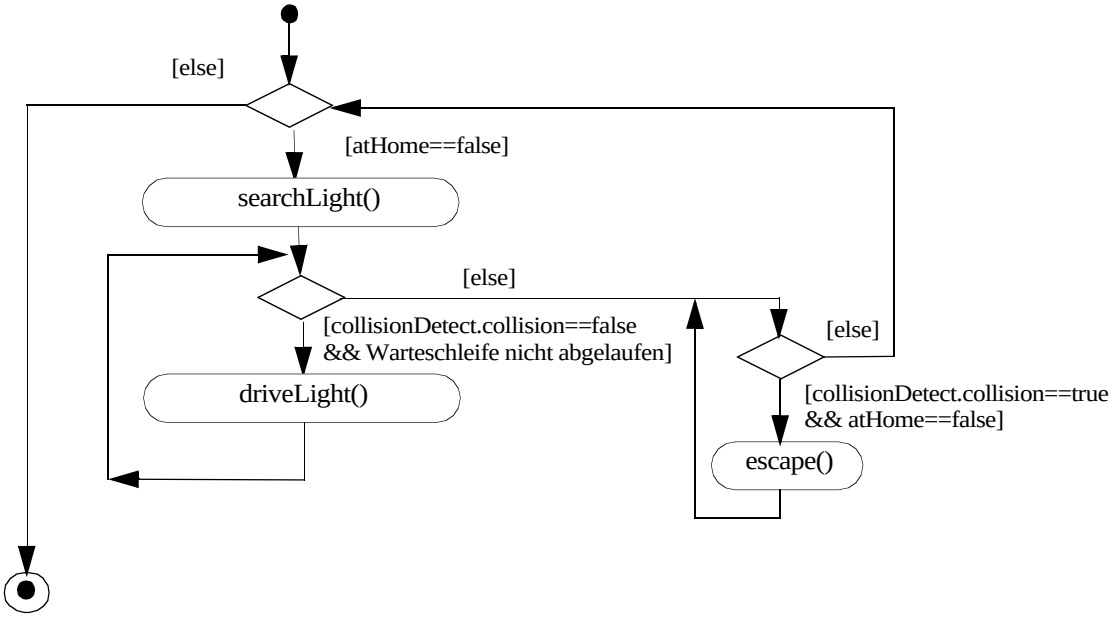


Abb. A-31: „driveHome“ Activity Diagram

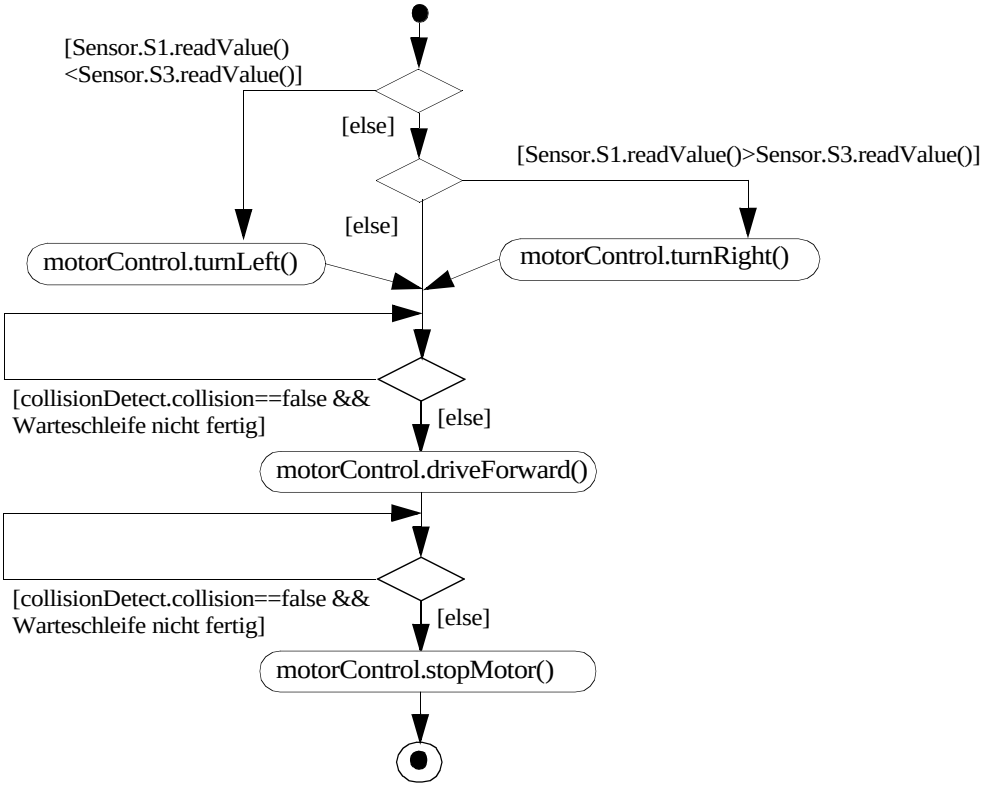


Abb. A-32: „driveLight“ Activity Diagram

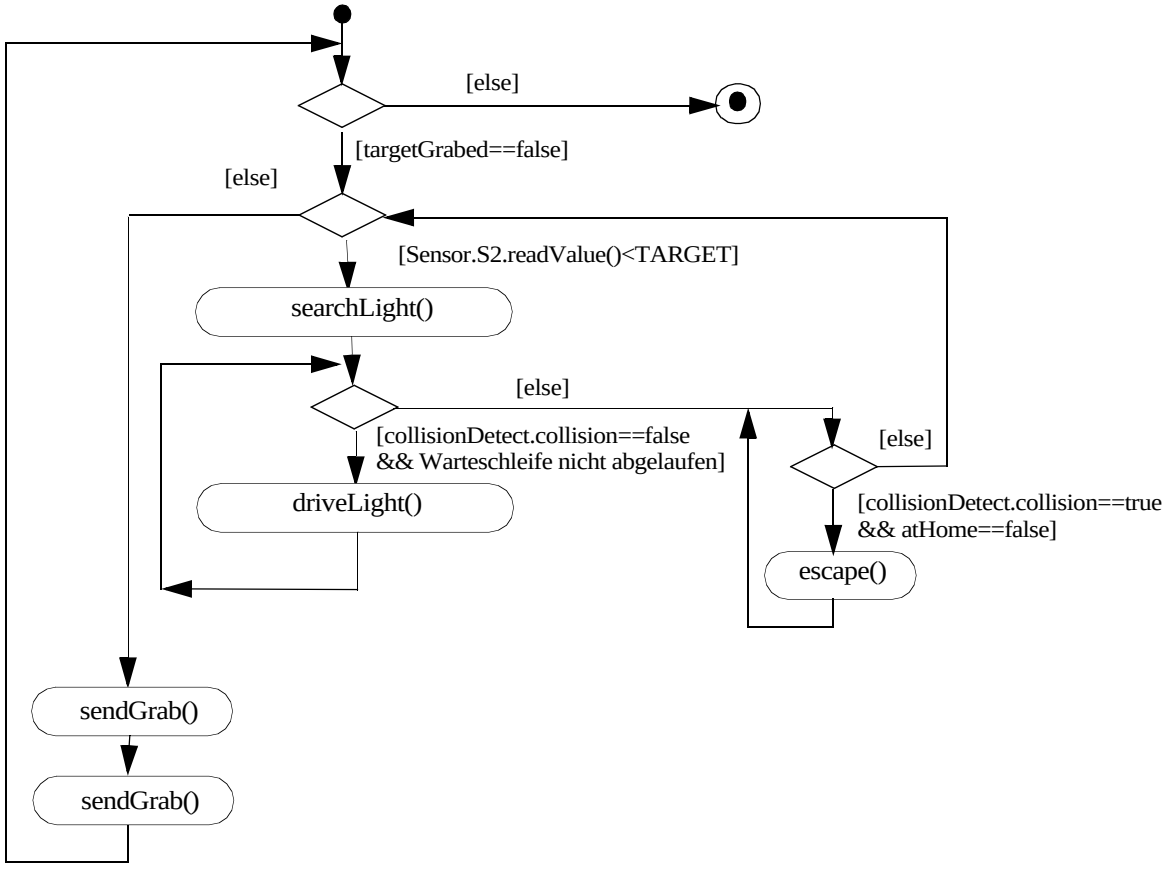


Abb. A-33: <<variant>> „driveTarget“ Activity Diagram

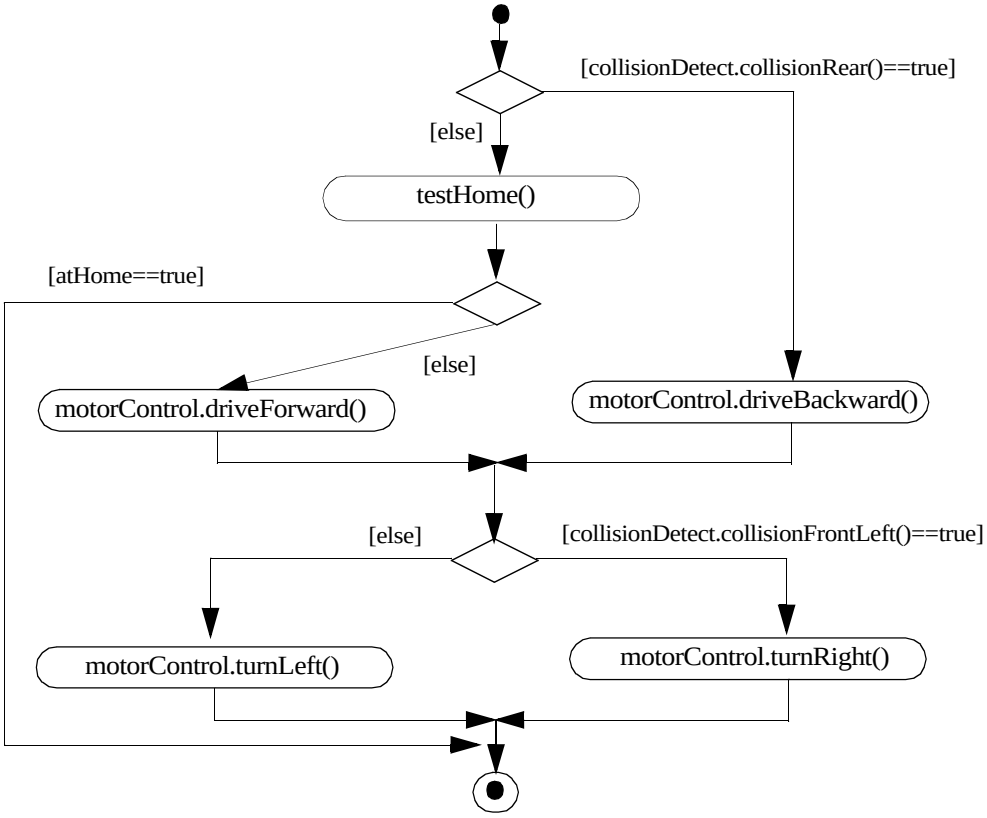


Abb. A-34: „escape“ Activity Diagram

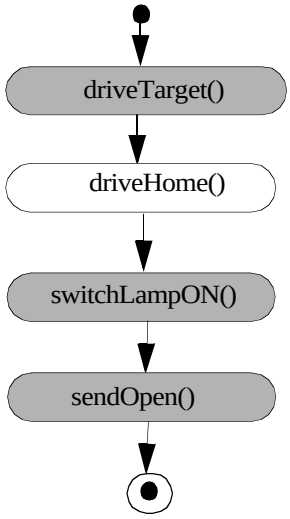


Abb. A-35: „main“ Activity Diagram

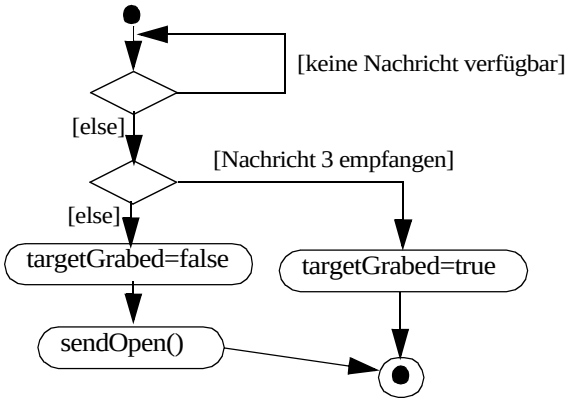


Abb. A-36: „receiveGrabberState“ Activity Diagram

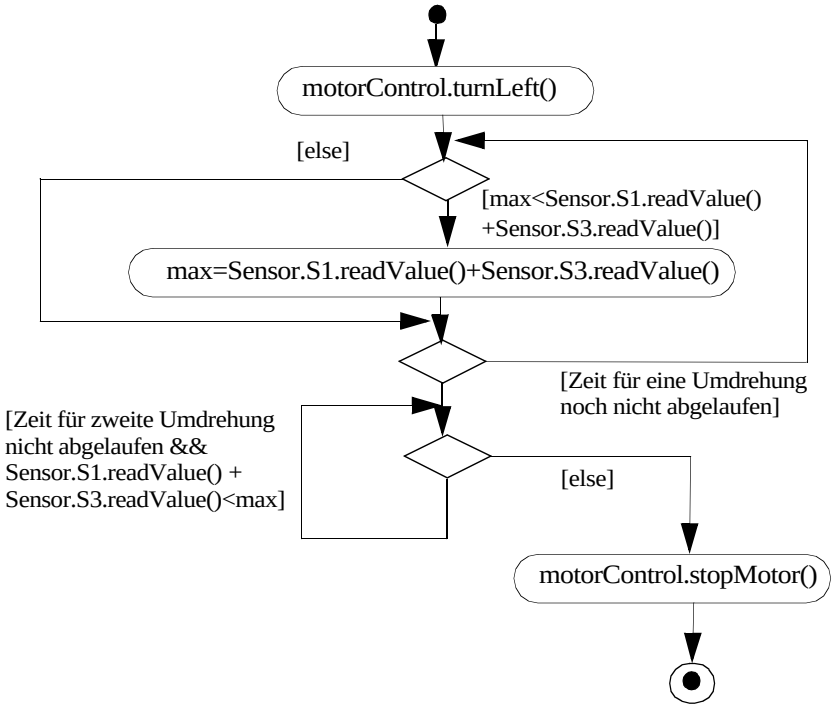


Abb. A-37: „searchLight“ Activity Diagram

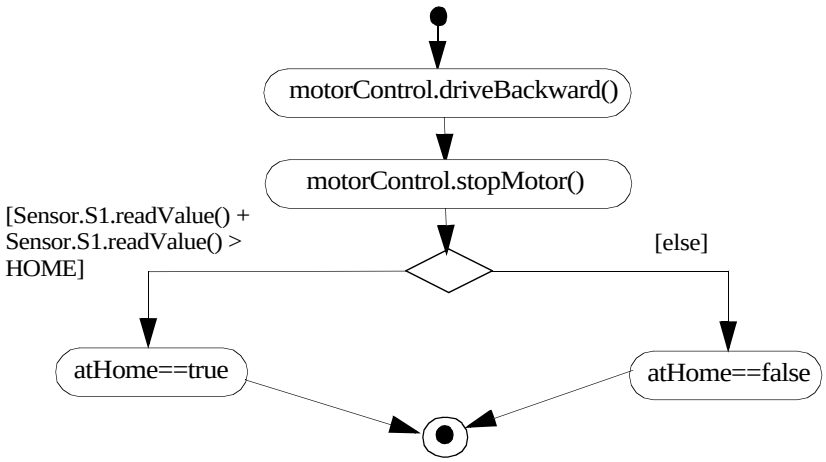


Abb. A-38: „testHome“ Activity Diagram

A.4.2.3 Interaction Model

Für die trivialen Aktivitäten *receiveGrabberState()*, *sendGrab()*, *sendOpen()*, *searchLight()* und *switchLampON()* der Klasse *DrivingSystem* werden keine *Collaboration Diagrams* aufgestellt. Die Aktivitäten der Klassen *CollisionDetect* und *MotorControl* werden ebenfalls nicht modelliert.

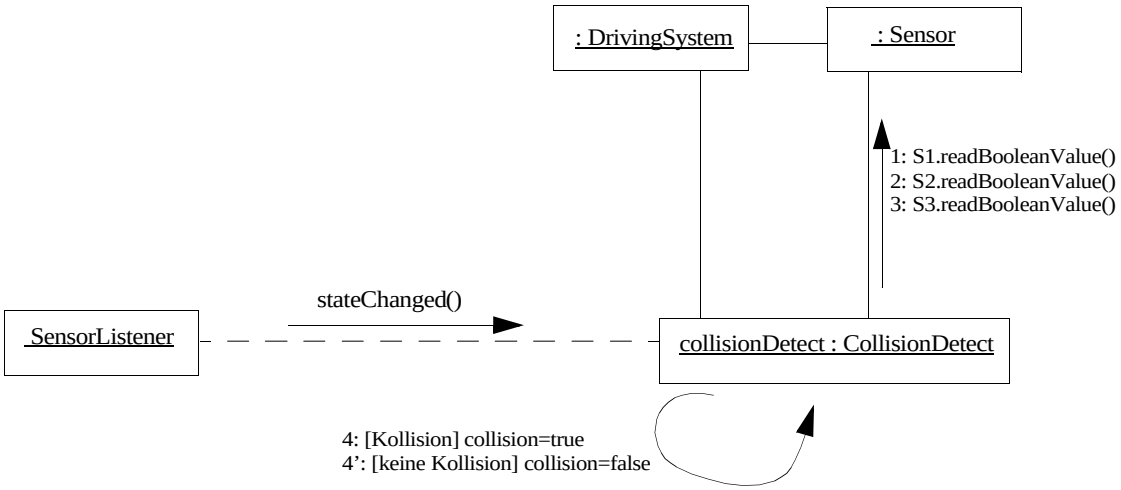


Abb. A-39: Collaboration Diagram für „stateChanged()“

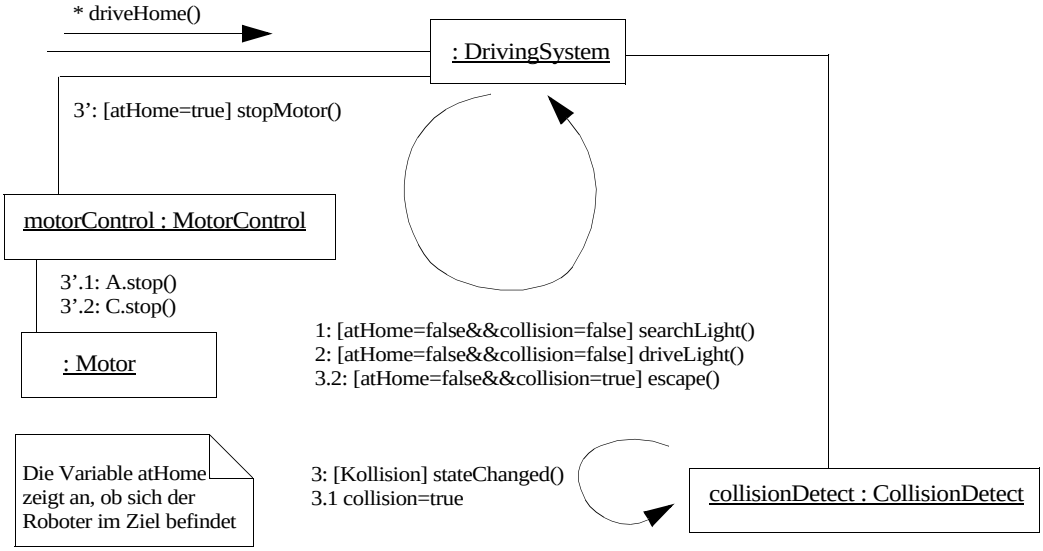


Abb. A-40: Collaboration Diagram für „driveHome()“

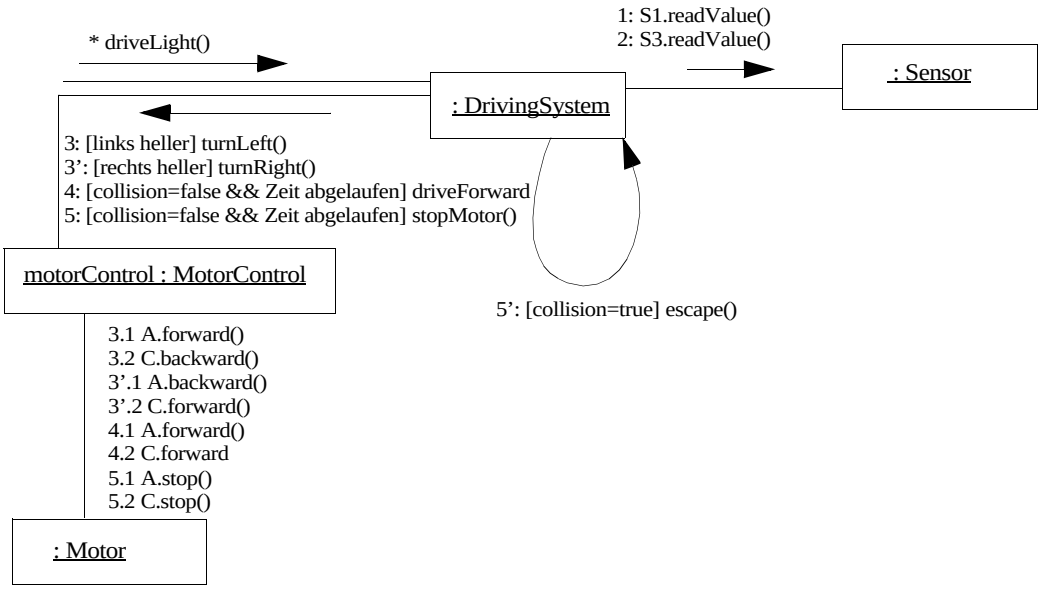


Abb. A-41: Collaboration Diagram für „driveLight()“

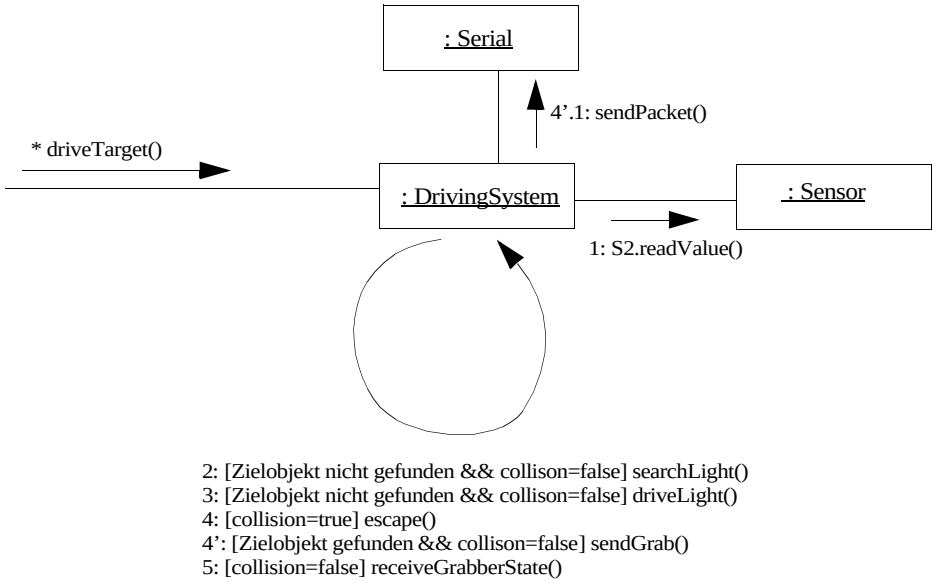


Abb. A-42: <<variant>> Collaboration Diagram für „driveTarget()“

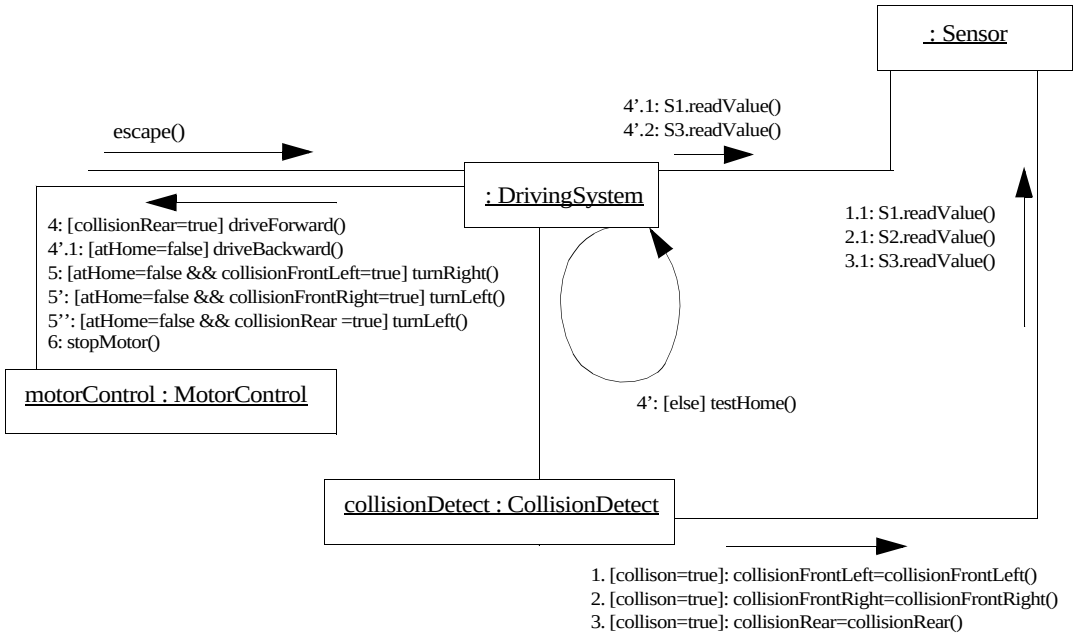


Abb. A-43: Collaboration Diagram für „escape()“

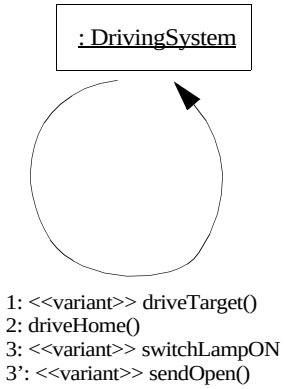


Abb. A-44: Collaboration Diagram für „main“

A.4.2.4 Decision Model

	ID	Frage	Variation Point	Entschluss	Effekt
Structural Model	1	Operation *DrivingSystem.driveTarget* benötigt?	Operation *DrivingSystem.driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.driveTarget*
	2	Operation *DrivingSystem.receiveGrabberState* benötigt?	Operation *DrivingSystem.receiveGrabberState*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.receiveGrabberState*
	3	Operation *DrivingSystem.sendGrab* benötigt?	Operation *DrivingSystem.sendGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendGrab*
	4	Operation *DrivingSystem.sendOpen* benötigt?	Operation *DrivingSystem.sendOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendOpen*
	5	Operation *DrivingSystem.switchLampON* benötigt?	Operation *DrivingSystem.switchLampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	Operation *DrivingSystem.switchLampON*
	6	Operation *MotorControl.LampON* benötigt?	Operation *MotorControl.LampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Operation *MotorControl.LampON*
	7	Attribut *DrivingSystem.bufferGrab* benötigt?	Attribut *DrivingSystem.bufferGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferGrab*
	8	Attribut *DrivingSystem.bufferOpen* benötigt?	Attribut *DrivingSystem.bufferOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferOpen*
	9	Attribut *DrivingSystem.bufferReceive* benötigt?	Attribut *DrivingSystem.bufferReceive*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferReceive*
	10	Attribut *DrivingSystem.targetGrabed* benötigt?	Attribut *DrivingSystem.targetGrabed*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.targetGrabed*
	11	Konstante *DrivingSystem.TARGET* benötigt?	Konstante *DrivingSystem.TARGET*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Konstante *DrivingSystem.TARGET*

Tabelle A-53: Decision Model für das Structural Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
Activity Model	1	Activity Model *driveTarget* (Abb. A-33) benötigt?	Activity Model *driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Activity Model *driveTarget*
	2	Operation *driveTarget* in Activity Model *main* (Abb. A-35) benötigt?	Operation *driveTarget* in Activity Model *main*	ja	entferne Schattierung
				nein (default)	entferne Operation *driveTarget* in Activity Model *main*
	3	Operation *switchLampON* in Activity Model *main* (Abb. A-35) benötigt?	Operation *switchLampON* in Activity Model *main*	ja (default)	entferne Schattierung
				nein	entferne Operation *switchLampON* in Activity Model *main*
	4	Operation *sendOpen* in Activity Model *main* (Abb. A-35) benötigt?	Operation *sendOpen* in Activity Model *main*	ja	entferne Schattierung
				nein (default)	entferne Operation *sendOpen* in Activity Model *main*

Tabelle A-54: Decision Model für das Activity Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
Interaktionsmodell	1	Collaboration Diagram *driveTarget* (Abb. A-42) benötigt?	Collaboration Diagram *driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Collaboration Diagram *driveTarget*

Tabelle A-55: Decision Model für das Interaction Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
DrivingSystem Komponent Implementation	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	ja (default)	- Structural Model Entscheidung 5,6: ja - Activity Model Entscheidung 3: ja
				nein	- Structural Model Entscheidung 5,6: nein - Activity Model Entscheidung 3: nein
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	ja	- Structural Model Entscheidung 1,2,3,4,7,8,9,10,11: ja - Activity Model Entscheidung 1,2,4: ja - Interaction Model Entscheidung 1: ja
				nein (default)	- Structural Model Entscheidung 1,2,3,4,7,8,9,10,11: nein - Activity Model Entscheidung 1,2,4: nein - Interaction Model Entscheidung 1: nein

Tabelle A-56: Top Level Entscheidungen für die Decision Models der *DrivingSystem Komponent Implementation*

A.4.3 Reuse

Erfahrungen bei der Entwicklung von *GrabberSystem* aus Anhang A.3 werden durch das *Experience Model* der *Context Realization* (vgl. A.5) wiederverwendet.

A.4.4 DrivingSystem Komponent Implementation

A.4.4.1 Implementation Structural Model

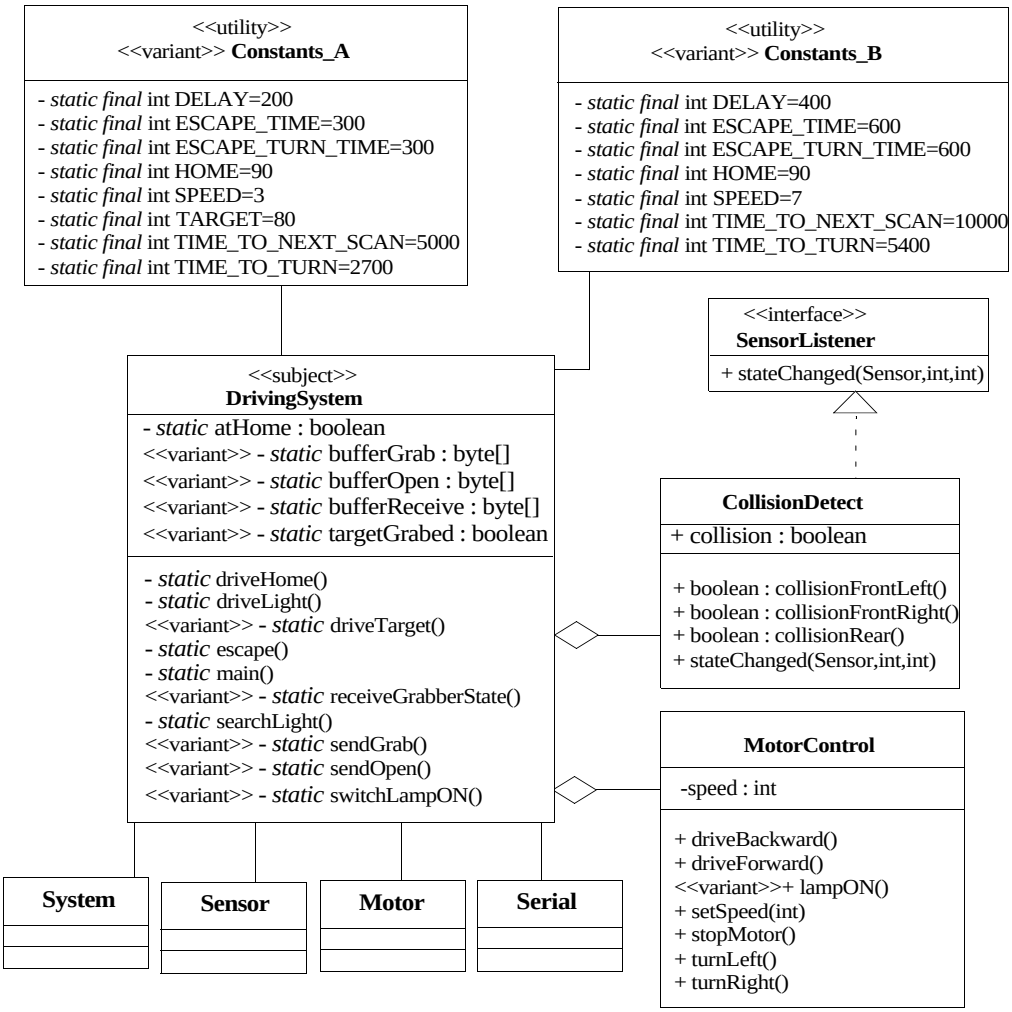


Abb. A-45: Class Diagram

A.4.4.2 Implementation Component Diagram

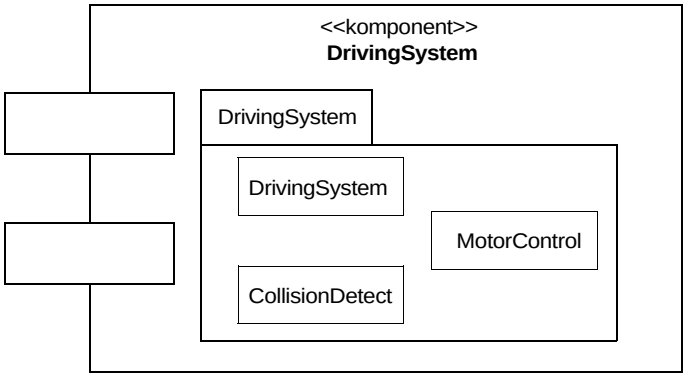


Abb. A-46: Component Diagram

A.4.4.3 Source Code

Der Source Code für die Komponente *DrivingSystem* befindet sich in Anhang C.2.

A.4.4.4 Decision Model

ID		Frage	Variation Point	Entschluss	Effekt
Structural Model	1	Operation *DrivingSystem.driveTarget* benötigt?	Operation *DrivingSystem.driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.driveTarget*
	2	Operation *DrivingSystem.receiveGrabberState* benötigt?	Operation *DrivingSystem.receiveGrabberState*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.receiveGrabberState*
	3	Operation *DrivingSystem.sendGrab* benötigt?	Operation *DrivingSystem.sendGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendGrab*
	4	Operation *DrivingSystem.sendOpen* benötigt?	Operation *DrivingSystem.sendOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendOpen*
	5	Operation *DrivingSystem.switchLampON* benötigt?	Operation *DrivingSystem.switchLampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	Operation *DrivingSystem.switchLampON*
	6	Operation *MotorControl.LampON* benötigt?	Operation *MotorControl.LampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Operation *MotorControl.LampON*
	7	Attribut *DrivingSystem.bufferGrab* benötigt?	Attribut *DrivingSystem.bufferGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferGrab*
	8	Attribut *DrivingSystem.bufferOpen* benötigt?	Attribut *DrivingSystem.bufferOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferOpen*
	9	Attribut *DrivingSystem.bufferReceive* benötigt?	Attribut *DrivingSystem.bufferReceive*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferReceive*
	10	Attribut *DrivingSystem.targetGrabed* benötigt?	Attribut *DrivingSystem.targetGrabed*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.targetGrabed*
	11	Utility-Klasse *Constants_A* benötigt?	Utility-Klasse *Constants_A*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Utility-Klasse *Constants_A*
	12	Utility-Klasse *Constants_B* benötigt?	Utility-Klasse *Constants_B*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Utility-Klasse *Constants_B*

Tabelle A-57: Decision Model für das Structural Model der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
Source Code	1	Operation *DrivingSystem.driveTarget* benötigt?	Operation *DrivingSystem.driveTarget*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.driveTarget*
	2	Operation *DrivingSystem.receiveGrabberState* benötigt?	Operation *DrivingSystem.receiveGrabberState*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.receiveGrabberState*
	3	Operation *DrivingSystem.sendGrab* benötigt?	Operation *DrivingSystem.sendGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendGrab*
	4	Operation *DrivingSystem.sendOpen* benötigt?	Operation *DrivingSystem.sendOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Operation *DrivingSystem.sendOpen*
	5	Operation *DrivingSystem.switchLampON* benötigt?	Operation *DrivingSystem.switchLampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	Operation *DrivingSystem.switchLampON*
	6	Operation *MotorControl.LampON* benötigt?	Operation *MotorControl.LampON*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Operation *MotorControl.LampON*
	7	Attribut *DrivingSystem.bufferGrab* benötigt?	Attribut *DrivingSystem.bufferGrab*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferGrab*
	8	Attribut *DrivingSystem.bufferOpen* benötigt?	Attribut *DrivingSystem.bufferOpen*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferOpen*
	9	Attribut *DrivingSystem.bufferReceive* benötigt?	Attribut *DrivingSystem.bufferReceive*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.bufferReceive*
	10	Attribut *DrivingSystem.targetGrabed* benötigt?	Attribut *DrivingSystem.targetGrabed*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.targetGrabed*
	11	Konstante *DrivingSystem.TARGET* benötigt?	Attribut *DrivingSystem.TARGET*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Attribut *DrivingSystem.TARGET*
	12	Aufruf von *driveTarget* in der Funktion *main* benötigt?	Aufruf von *driveTarget* in der Funktion *main*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne Aufruf von *driveTarget* in der Funktion *main*
	13	Aufruf von *switchLampON* in der Funktion *main* benötigt?	Aufruf von *switchLampON* in der Funktion *main*	ja (default)	entferne Stereotyp <<variant>>
				nein	entferne Aufruf von *switchLampON* in der Funktion *main*

Tabelle A-58: Decision Model für den Source Code der Komponente *DrivingSystem*

ID		Frage	Variation Point	Entschluss	Effekt
DrivingSystem Komponent Realization	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	ja (default)	- Structural Model Entscheidung 5,6: ja - Source Code Entscheidung 5,6,13: ja
				nein	- Structural Model Entscheidung 5,6: nein - Source Code Entscheidung 5,6,13: nein
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	ja	- Structural Model Entscheidung 1,2,3,4,7,8,9,10: ja - Source Code Entscheidung 1,2,3,4,7,8,9,10,11112: ja
				nein (default)	- Structural Model Entscheidung 1,2,3,4,7,8,9,10: nein - Source Code Entscheidung 1,2,3,4,7,8,9,10,11,12: nein
	3	Fährt der Roboter auf glatter Oberfläche?	Geschwindigkeit, Wartezeiten	ja	Structural Model Entscheidung 11: ja
				nein (default)	Structural Model Entscheidung 11: nein
	4	Fährt der Roboter auf Teppichboden?	Geschwindigkeit, Wartezeiten	ja (default)	Structural Model Entscheidung 12: ja
				nein	Structural Model Entscheidung 12: nein

Tabelle A-59: Top Level Entscheidungen für die Decision Models der *DrivingSystem Komponent Realization*

A.4.5 Inspection

Wurde informell durch den Autor und den Betreuer dieser Diplomarbeit durchgeführt.

A.4.6 Measurement of Structural Properties

Entfällt.

A.4.7 Testing

A.4.7.1 Functional Test Cases

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „driveHome“.	Kollision	Das Fahrzeug weicht aus.
2.	Die Komponente befindet sich im Zustand „driveHome“.	Kollision && Lichtstärke vor den Lichtsensoren S1 und S3 übersteigt die Schwelle für Ziel	Das Fahrzeug bleibt stehen.
3.	Die Komponente befindet sich im Zustand „driveHome“.	links heller	Das Fahrzeug fährt nach links.
4.	Die Komponente befindet sich im Zustand „driveHome“.	rechts heller	Das Fahrzeug fährt nach rechts.
5.	Die Komponente befindet sich im Zustand „driveHome“.	links fast genauso hell wie rechts	Das Fahrzeug fährt geradeaus.

Tabelle A-60: Functional Test Cases für die Operation driveHome()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „driveHome“.	Kollision	Das Fahrzeug weicht aus.
2.	Die Komponente befindet sich im Zustand „driveTarget“.	Lichtstärke vor Lichtsensor S2 übersteigt die Schwelle für Zielobjekt	Das Fahrzeug bleibt stehen.
3.	Die Komponente befindet sich im Zustand „driveTarget“.	links heller	Das Fahrzeug fährt nach links.
4.	Die Komponente befindet sich im Zustand „driveTarget“.	rechts heller	Das Fahrzeug fährt nach rechts.
5.	Die Komponente befindet sich im Zustand „driveTarget“.	links fast genauso hell wie rechts	Das Fahrzeug fährt geradeaus.

Tabelle A-61: <<variant>> Functional Test Cases für die Operation driveTarget()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „escape“.	Tastsensor S1 hat ausgelöst.	Das Fahrzeug weicht nach hinten links aus.
2.	Die Komponente befindet sich im Zustand „escape“.	Tastsensor S3 hat ausgelöst.	Das Fahrzeug weicht nach hinten rechts aus.
3.	Die Komponente befindet sich im Zustand „escape“.	Tastsensor S2 hat ausgelöst.	Das Fahrzeug weicht nach vorne links aus.

Tabelle A-62: Functional Test Cases für die Operation escape()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
4.	Die Komponente befindet sich im Zustand „escape“.	Tastsensoren S1 und S3 haben ausgelöst.	Das Fahrzeug weicht nach hinten links aus.

Tabelle A-62: Functional Test Cases für die Operation escape()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „closeGrabber“.	Nachricht 2 (Fehler)	Fahrzeug sendet den Befehl zum Öffnen des Greifers und sucht das Zielobjekt erneut.
2.	Die Komponente befindet sich im Zustand „closeGrabber“.	Nachricht 3 (Erfolg)	Fahrzeug fährt ins Ziel.

Tabelle A-63: <<variant>> Functional Test Cases für die Operation receiveGrabberState()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „driveHome“.	Eine Lichtquelle ist angeschaltet	Das Fahrzeug ist in Richtung Lichtquelle ausgerichtet.
<<variant>> 2.	Die Komponente befindet sich im Zustand „driveTarget“.	Eine Lichtquelle ist angeschaltet	Das Fahrzeug ist in Richtung Lichtquelle ausgerichtet.

Tabelle A-64: Functional Test Cases für die Operation searchLight()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „closeGrabber“.		Nachricht 0 (Greifen) wird gesendet.

Tabelle A-65: <<variant>> Functional Test Cases für die Operation sendGrab()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „openGrabber“.		Nachricht 1 (Öffnen) wird gesendet.

Tabelle A-66: <<variant>> Functional Test Cases für die Operation sendOpen()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	Die Komponente befindet sich im Zustand „switchLampON“.		Die Lampe wird eingeschaltet.

Tabelle A-67: <<variant>> Functional Test Cases für die Operation switchLampON()

A.4.7.2 Structural Test Cases

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	stateChanged()	Pfad 1: S1.readValue()->S2.readValue()S3.readValue()->collision=true	collision=true
2.	stateChanged()	Pfad 2: S1.readValue()->S2.readValue()S3.readValue()->collision=false	collision=false

Tabelle A-68: Structural Test Cases für *Collaboration Diagram* stateChanged()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	driveHome()	Pfad 1: searchLight()->escape()	escape()
2.	driveHome()	Pfad 2: searchLight()->driveLight()->escape()	escape()
3.	driveHome()	Pfad 3: searchLight()->driveLight()->stopMotor()	stopMotor()

Tabelle A-69: Structural Test Cases für *Collaboration Diagram* driveHome()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	driveLight()	Pfad 1: S1.readValue()-> S3.readValue()->turnLeft()-> driveForward ->stopMotor()	stopMotor()
2.	driveLight()	Pfad 2: S1.readValue()-> S3.readValue()->turnRight()-> driveForward->stopMotor()	stopMotor()
3.	driveLight()	Pfad 3: S1.readValue()-> S3.readValue()->driveForward->stopMotor()	stopMotor()
4.	driveLight()	Pfad 4: S1.readValue()-> S3.readValue()->turnLeft()-> driveForward ->escape()	escape()
5.	driveLight()	Pfad 5: S1.readValue()-> S3.readValue()->turnRight()-> escape()	escape()

Tabelle A-70: Structural Test Cases für *Collaboration Diagram* driveLight()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	driveTarget()	Pfad 1: S2.readValue()->searchLight()->driveLight()->sendGrab() ->receiveGrabberState()	sendGrab()
2.	driveTarget()	Pfad 2: S2.readValue()->searchLight()->escape()	escape()
3.	driveTarget()	Pfad 3: S2.readValue()->searchLight()->driveLight()->escape()	escape()

Tabelle A-71: <<variant>> Structural Test Cases für *Collaboration Diagram* driveTarget()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
1.	escape()	Pfad 1: collisionFrontLeft()->collisionFrontRight()->collisionRear()->driveForward()-> turnLeft()	turnLeft()
2.	escape()	Pfad 2: collisionFrontLeft()->collisionFrontRight()->collisionRear()->testHome()-> driveBackward()->turnLeft()	turnLeft()
3.	escape()	Pfad 3: collisionFrontLeft()-> collisionFrontRight()->collisionRear()->testHome()->stopMotor()	stopMotor()

Tabelle A-72: Structural Test Cases für *Collaboration Diagram* escape()

ID	Vorbedingung	Testfall Input	Erwartetes Ergebnis
<<variant>> 1.	main()	Pfad 1: driveTarget()->driveHome()->sendOpen()	sendOpen()
<<variant>> 2.	main()	Pfad 2: driveHome()-> switchLampON()	switchLampON()

Tabelle A-73: Structural Test Cases für *Collaboration Diagram* main()

A.4.7.3 State-based Test Cases

	Testfall Input			Erwartetes Ergebnis	
ID	Startzustand	Event	Testbedingungen	Aktion	Zustand
1.	driveHome	Kollision	Die Lichtstärke liegt unterhalb der Schwelle für Ziel.	Das Fahrzeug weicht aus.	escape
<<variant>> 2.	driveHome	Kollision	Die Lichtstärke liegt oberhalb der Schwelle für Ziel.	Das Fahrzeug stoppt.	switchLam-pON
<<variant>> 3.	driveHome	Kollision	Die Lichtstärke liegt oberhalb der Schwelle für Ziel.	Das Fahrzeug stoppt.	openGrabber
<<variant>> 4.	driveTarget	Kollision		Das Fahrzeug stoppt.	escape
<<variant>> 5.	driveTarget	Zielobjekt gefunden	Zielobjekt befindet sich vor dem Greifer.	Das Fahrzeug versucht das Zielobjekt zu greifen.	closeGrabber
6.	escape	Ausweichen beendet	Vorheriger Zustand war driveHome	Die unterbrochene Aktion wird fortgesetzt.	driveHome
<<variant>> 7.	escape	Ausweichen beendet	Vorheriger Zustand war driveTarget	Die unterbrochene Aktion wird fortgesetzt.	driveTarget
<<variant>> 8.	closeGrab-ber	Nachricht 2 empfangen (Erfolg).	Nachricht 2 wird von zweiten RCX-Bau-stein gesendet.	Das Fahrzeugt steuert das Ziel an.	driveHome

Tabelle A-74: State-based Test Cases

	Testfall Input			Erwartetes Ergebnis	
ID	Startzustand	Event	Testbedingungen	Aktion	Zustand
<<variant>> 9.	closeGrabber	Nachricht 3 empfangen (Fehler).	Nachricht 3 wird von zweiten RCX-Baustein gesendet.	Das Fahrzeugt sendet das Signal zum Öffnen des Greifers.	openGrabber
<<variant>> 10.	openGrabber	Transition von closeGrabber.	Vorheriger Zustand war closeGrabber.	Da der Greifvorgang erfolglos war, wird die Nachricht zum öffnen des Greifers gesendet, um einen neuen Versuch zu starten.	driveTarget
<<variant>> 11.	openGrabber	Transition von driveHome.	Vorheriger Zustand war driveHome.	Im Ziel wird Nachricht zum Öffnen des Greifers gesendet.	Endzustand
<<variant>> 12.	switchLampON	Transition von driveHome.		Die Lampe wird eingeschaltet.	Endzustand

Tabelle A-74: State-based Test Cases

A.4.7.4 Environment-based Test Cases

ID	Einflussgröße	Wert	Betroffene Testfälle
<<variant>> 1.	Bodenbelag	Teppich	Functional Test Cases: - Alle bis auf Tabelle A-67 (Operation switchLampON()) Structural Test Cases: - Alle bis auf Tabelle A-68 (Collaboration Diagram stateChanged()) State-based Test Cases: - Alle bis auf Testfall 2 und 12 in Tabelle A-74
<<variant>> 2.	Bodenbelag	Tischplatte	
3.	Beleuchtung	Heller Raum	Functional Test Cases: - Alle bis auf Tabelle A-62 (Operation escape()) und Tabelle A-67 (Operation switchLampON()) Functional Test Cases: - Alle bis auf Tabelle A-72 (Collaboration Diagram escape()) State-based Test Cases: - Alle bis auf Testfall 4 und 12 in Tabelle A-74
4.	Beleuchtung	Abgedunkelter Raum	

Tabelle A-75: Environment-based Test Cases

A.4.7.5 Decision Model für die Test Cases

	Application A	Application B
Functional Test Cases	**entferne:** - <<variant>> bei Tabelle A-61 (Functional Test Cases für die Operation driveTarget()) - <<variant>> bei Tabelle A-63 (Functional Test Cases für die Operation receiveGrabberState()) - <<variant>> bei Testfall 2 in Tabelle A-64 (Functional Test Cases für die Operation searchLight()) - <<variant>> bei Tabelle A-65 (Functional Test Cases für die Operation sendGrab()) - <<variant>> bei Tabelle A-66 (Functional Test Cases für die Operation sendOpen()) - Tabelle A-67 (Functional Test Cases für die Operation switchLampON())	**entferne:** - Tabelle A-61 (Functional Test Cases für die Operation driveTarget()) - Tabelle A-63 (Functional Test Cases für die Operation receiveGrabberState()) - Testfall 2 in Tabelle A-64 (Functional Test Cases für die Operation searchLight()) - Tabelle A-65 (Functional Test Cases für die Operation sendGrab()) - Tabelle A-66 (Functional Test Cases für die Operation sendOpen()) - <<variant>> bei Tabelle A-67 (Functional Test Cases für die Operation switchLampON())
Structural Test Cases	**entferne:** - <<variant>> bei Tabelle A-71 (Structural Test Cases für Collaboration Diagram driveTarget()) - <<variant>> bei Testfall 1 in Tabelle A-73 (Structural Test Cases für Collaboration Diagram main()) - Testfall 2 in Tabelle A-73 (Structural Test Cases für Collaboration Diagram main())	**entferne:** - Tabelle A-71 (Structural Test Cases für Collaboration Diagram driveTarget()) - <<variant>> bei Testfall 2 in Tabelle A-73 (Structural Test Cases für Collaboration Diagram main()) - Testfall 1 in Tabelle A-73 (Structural Test Cases für Collaboration Diagram main())
State-based Test Cases	**entferne in Tabelle A-74:** - Testfall 2 - <<variant>> bei Testfall 3 - <<variant>> bei Testfall 4 - <<variant>> bei Testfall 5 - <<variant>> bei Testfall 7 - <<variant>> bei Testfall 8 - <<variant>> bei Testfall 9 - <<variant>> bei Testfall 10 - <<variant>> bei Testfall 11 - Testfall12	**entferne in Tabelle A-74:** - <<variant>> Testfall 2 - bei Testfall 3 - bei Testfall 4 - bei Testfall 5 - bei Testfall 7 - bei Testfall 8 - bei Testfall 9 - bei Testfall 10 - bei Testfall 11 - <<variant>>Testfall12
Environment-based Test Cases	**entferne in Tabelle A-75:** - Testfall 1 - <<variant>> bei Testfall 2	**entferne in Tabelle A-75:** - <<variant>> bei Testfall 1 - Testfall 2

Tabelle A-76: Decision Model für die Test Cases

A.5 Experience Model

Betroffene Komponenten	Erfahrungen	
Alle Komponenten	**Erfahrung 1**	
	Stichwort	static
	Richtlinie	Die main()-Methode muss als „static“ deklariert werden. Dadurch müssen alle Attribute und Methoden der obersten Klasse ebenfalls „static“ sein.
	Quelle	Dokumentation von leJOS [LEJOS]
	Art	Vorschrift
	Betrifft	Implementierungsebene
	Erfahrung 2 <<variant>>	
	Stichwort	Energieverbrauch
	Beschreibung	Es sollten möglichst wenige Nachrichten verschickt werden, um Energie zu sparen.
	Quelle	Abschnitt A.3.1 in der „Komponent Realization“ der Komponente „GrabberSystem“
	Art	Empfehlung
	Betrifft	Algorithmus, d. h. *Message-*, *Activity-* und *Interaction Models* auf allen Ebenen
	Erfahrung 3 <<variant>>	
	Stichwort	Kommunikation
	Beschreibung	Da das eingebettete System anfällig gegen Störeinflüsse von außerhalb ist, müssen Nachrichten mehrfach versandt werden, um eine erfolgreiche Kommunikation zu gewährleisten.
	Quelle	Abschnitt A.6.3 beim Testen der Komponente „GrabberSystem“
	Art	Empfehlung
	Betrifft	Algorithmus, d. h. *Message-*, *Activity-* und *Interaction Models* auf allen Ebenen, insbesondere die Statechart Diagramme können betroffen sein.
	Erfahrung 4	
	Stichwort	Speicherverbrauch
	Beschreibung	Da keine *Garbage Collection* existiert sollen während der Laufzeit keine wiederholten Instantiierungen vorgenommen werden. Rekursion soll ebenfalls vermieden werden.
	Quelle	Abschnitt A.3.4 bei der Implementierung und A.6.3 beim Testen der Komponente „GrabberSystem“ Dokumentation von leJOS: „Technical Notes“ [LEJOS]
	Art	Empfehlung
	Betrifft	Alle Ebenen

Tabelle A-77: Experience Model

Betroffene Komponenten	Erfahrungen	
Alle Komponenten	**Erfahrung 5**	
	Stichwort	long
	Beschreibung	Der Datentyp long sollte nicht verwendet werden.
	Quelle	Fehlermeldung des Compilers
	Art	Empfehlung
	Betrifft	Implementierung
Komponente DrivingSystem	**Erfahrung DrivingSystem.1**	
	Stichwort	Lichtsensoren und Tastsensoren
	Beschreibung	An jedem Eingang des RCX-Bausteins sind jeweils ein Licht- und ein Tastsensor gleichzeitig angeschlossen. Dies bewirkt, dass bei Verwendung des Interfaces *SensorListener* ständig Alarm geschlagen wird, da die gemessene Lichtstärke stark schwankt. Es muss daher erst überprüft werden, ob sich auch der boolesche Wert am Eingang geändert hat, bevor das Event weitere Verhaltensweisen auslöst.
	Quelle	Anmerkungen zu Tabelle A-40
	Art	Richtlinie
	Betrifft	Realisierungsebene, Implementierungsebene

Tabelle A-77: Experience Model

Decision Model

ID		Frage	Variation Point	Entschluss	Effekt
Experience Model	1	*Erfahrung 2* benötigt?	*Erfahrung 2*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne *Erfahrung 2*
	2	*Erfahrung 3* benötigt?	*Erfahrung 3*	ja	entferne Stereotyp <<variant>>
				nein (default)	entferne *Erfahrung 3*

Tabelle A-78: Decision Model für das Experience Model

A.6 Application Engineering

Nur für die Top Level *Decision Models* durchgeführt.

A.6.1 Application A - Decision Model Instance

A.6.1.1 Instantiierung der Context Realization

ID		Frage	Gegenstand	Entschluss
Context Realization	1	Ist das System mit einem Greifer ausgestattet?	Greifer	ja
	2	Ist das System mit einer Lampe ausgestattet?	Lampe	nein

Tabelle A-79: Decision Model auf der Ebene der *Context Realization Instantiation*

A.6.1.2 Instantiierung der Specification, Realization und Implementation für die Komponente GrabberSystem

Entfällt, da es innerhalb dieser Komponente keine Variabilität gibt.

Implementation Structural Model

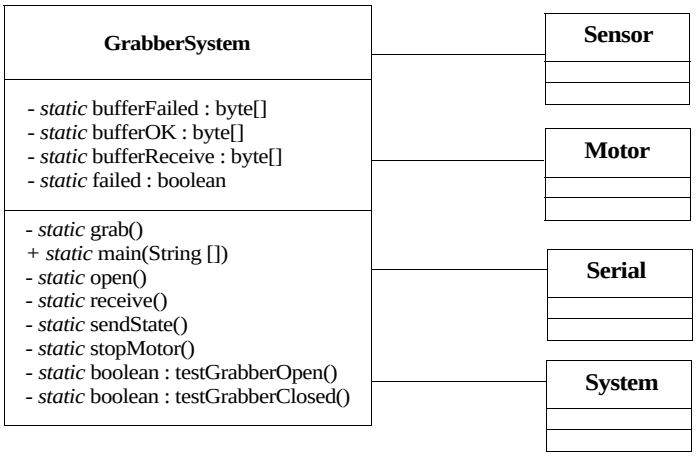


Abb. A-47: Class Diagram der Komponente GrabberSystem

A.6.1.3 Instantiierung der Specification, Realization und Implementation für die Komponente DrivingSystem

ID		Frage	Variation Point	Entschluss
DrivingSystem Komponent Specification	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	nein
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	ja

Tabelle A-80: *Komponent Specification Instantiation* der Komponente DrivingSystem

ID		Frage	Variation Point	Entschluss
DrivingSystem Komponent Realization	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	nein
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	ja

Tabelle A-81: *Komponent Realization Instantiation* der Komponente DrivingSystem

ID		Frage	Variation Point	Entschluss
DrivingSystem Komponent Implementation	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	nein
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	ja

Tabelle A-82: *Komponent Implementation Instantiation* der Komponente DrivingSystem

Implementation Structural Model

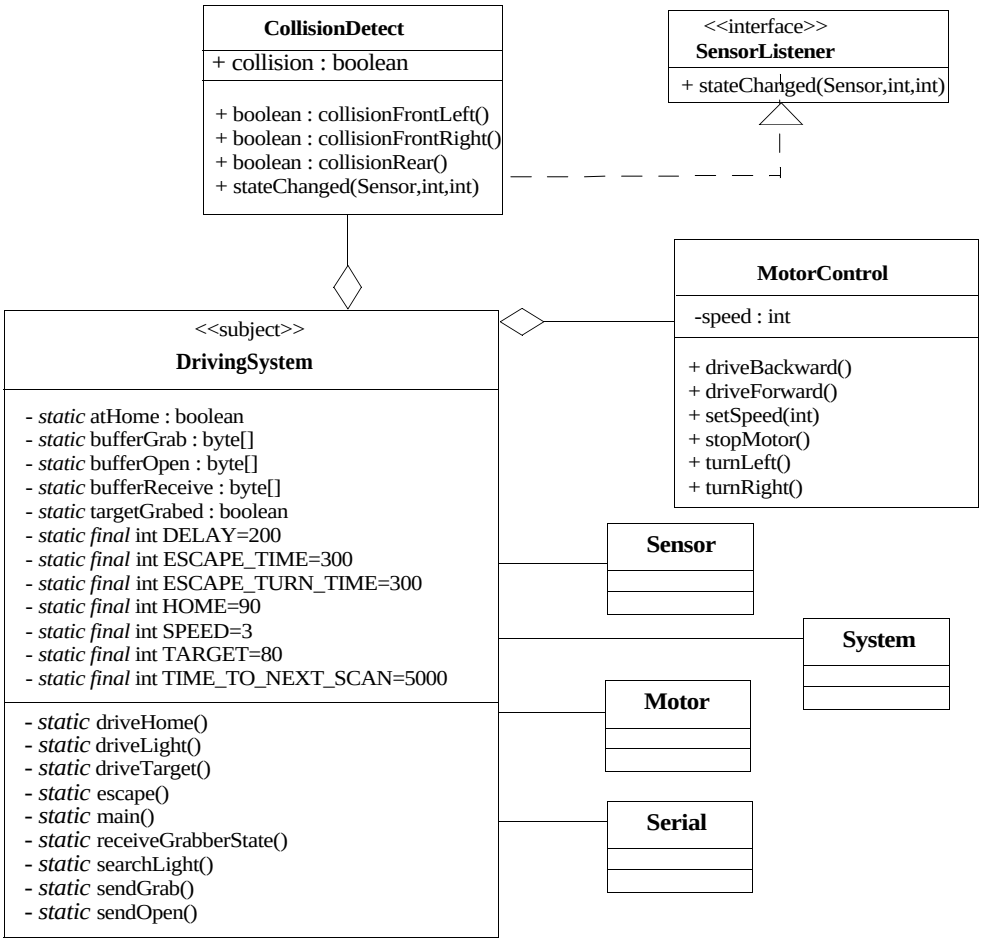


Abb. A-48: Class Diagram der Komponente DrivingSystem

A.6.2 Application B - Decision Model Instance

A.6.2.1 Instantiierung der Context Realization

ID		Frage	Gegenstand	Entschluss
Context Realization	1	Ist das System mit einem Greifer ausgestattet?	Greifer	nein
	2	Ist das System mit einer Lampe ausgestattet?	Lampe	ja

Tabelle A-83: Decision Model auf der Ebene der *Context Realization Instantiation*

A.6.2.2 Instantiierung der Specification, Realization und Implementation für die Komponente DrivingSystem

ID		Frage	Variation Point	Entschluss
DrivingSystem Komponent Specification	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	ja
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	nein

Tabelle A-84: *Komponent Specification Instantiation* der Komponente DrivingSystem

ID		Frage	Variation Point	Entschluss
DrivingSystem Komponent Realization	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	ja
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	nein

Tabelle A-85: *Komponent Realization Instantiation* der Komponente DrivingSystem

ID		Frage	Variation Point	Entschluss
DrivingSystem Komponent Implementation	1	Ist *DrivingSystem* mit einer Lampe ausgestattet?	Lampe	ja
	2	Kommuniziert *DrivingSystem* mit einer Greiferkomponente?	Greifer	nein

Tabelle A-86: *Komponent Implementation Instantiation* der Komponente DrivingSystem

Implementation Structural Model

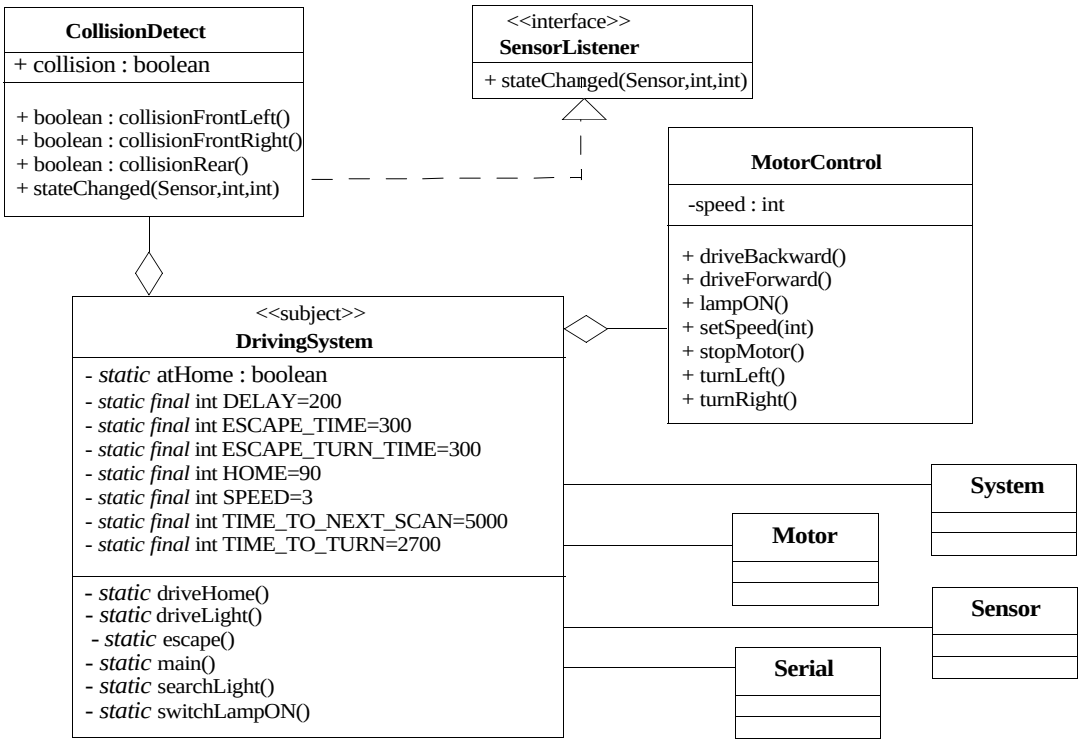


Abb. A-49: Class Diagram der Komponente DrivingSystem

A.6.3 Testing

Beim Testen der Komponente „GrabberSystem“ stellte sich eine weitere Besonderheit eingebetteter Systeme heraus. Die Signale können durch Störeinflüsse der Umwelt verfälscht werden oder verloren gehen. Aus diesem Grund wurde die Komponente dahingehend überarbeitet, dass alle Signale, die für Komponenten außerhalb der Hardwarekomponente „GrabberHWC“ bestimmt werden, mehrfach gesendet werden.

Weiterhin können unlogische Zustandübergänge auftreten. Ursprünglich war die Komponente so realisiert, dass die Komponente „GrabberHWC“ nur eine Signalart erhielt und den Greifer abwechselnd öffnete und schloss. Beim Start des Roboters wurde davon ausgegangen, dass der Greifer offen ist. In der Realität kann es aber vorkommen, dass z. B. der Benutzer den Greifer schließt oder das Signal zum Schließen verpasst wird. Wenn dies eingetreten ist, reden „DriverHWC“ und „GrabberHWC“ aneinander vorbei und das Gesamtsystem versagt. Daher wurde „GrabberHWC“ so verändert, dass es unterschiedliche Nachrichten für das Öffnen und das Schließen des Greifers gibt. Die vorangehenden Abschnitte wurden bereits entsprechend überarbeitet.

Ein erstes Ergebnis der Tests war, dass der Speicher des Beispielsystems knapp war. Grund war, dass die für dieses Projekt verwendete *Virtual Machine* „leJOS“ keine *Garbage Collection* kennt. Da viele Instanzen entsprechend dem Grundsatz „*Information Hiding*“ lokal in den Methoden erzeugt wurden, verschwendete jeder Methodenaufruf Speicher. Abhilfe schaffte der Verzicht auf wiederholte Instantiierung und die Vermeidung von Rekursion.

Zur Durchführung der Tests wurde ein zweiter RCX-Baustein verwendet, um Nachrichten zu senden und zu empfangen. Probleme traten durch den Tastsensor S2 auf, der anzeigt, dass der Greifer vollständig geöffnet wurde. Nach einer Überarbeitung der Roboterhardware war diese Fehlerquelle behoben. Unter der Bedingung, dass die Fernbedienung höchstens 50 cm vom RCX-Baustein entfernt war und sich direkt vor der IR-Schnittstelle befand, wurden alle Testfälle bestanden.

Anhang B

Bauanleitung Roboter

Zum Bau des Roboters wurden die Bauteile eines *Robotics Invention Systems 1.5* (RIS 1.5) LEGO Baukastens verwendet. Zusätzlich werden ein zweiter RCX Baustein, 4 Tastsensoren, 2 Lichtsensoren, 1 Motor und 1 Lampe benötigt. Ein von LEGO angebotener Erweiterungskasten, das *Utimate Accessory Set* enthält u. a. 1 Rotationssensor, 1 Tastsensor, 1 Lampe und 1 Fernbedienung. Tabelle B-1 zeigt verschiedene Möglichkeiten, die benötigten Teile zusammenzustellen. Dabei muss berücksichtigt werden, dass ein RIS-Baukasten über 700 Teile enthält. Vor diesem Hintergrund ist Variante 3 zwar am billigsten, aber nicht unbedingt am preiswertesten. Die Exklusivrechte für den Vertrieb von LEGO-Einzelteilen in Deutschland besitzt die Technik-LPE GmbH [LPE].

Produktbezeichnung	Preis	Variante 1	Variante 2	Variante 3
RIS 1.5	ca. 450,-	2	2	1
Ultimate Accessory Set	ca. 118,-	1		
RCX-Baustein	ca. 223,-			1
Motor	ca. 37,-			1
Lichtsensor	ca. 33,-	1	1	2
Tastsensor	ca. 24,-	1	2	4
Lampe (nur im 3er Pack erhältlich)	ca. 56,-		1	1
Summe		ca. 1075,-	ca. 1037,-	ca. 928,-

Tabelle B-1 : Preisbeispiele [LPE]

Statt der teuren LEGO-Lampe, die es einzeln nur im 3er Pack zu kaufen gibt, kann auch eine andere Lampe angeschlossen werden. Eventuell ist ein Vorwiderstand erforderlich, damit diese die am RCX-Ausgang anliegende Spannung von 9 Volt verkraftet.

Die Bauanleitung in Abschnitt B.1 skizziert kurz die wesentlichen Baugruppen des Roboters. Für das Funktionieren des Roboters spielt die tatsächliche Bauweise nur eine untergeordnete Rolle. Wichtig ist nur, dass die Sensoranordnung beibehalten wird. Eventuell müssen bei alternativen Bauweisen die Ziel-

objekte aus Abschnitt B.3.2 angepasst werden, damit sie von den Lichtsensoren detektiert und vom Greifer aufgenommen werden können.

Die Bilder wurden mit den Programmen LDraw [LDRAW] und MLCAD [MLCAD] erzeugt und zeigen die Baugruppen aus verschiedenen Sichten (Abb. B-1).

Vorne	Links
Oben	3 D

Abb. B-1: Ansichtswinkel der einzelnen Fenster

B.1 Komponenten des Roboters

Der Roboter besteht im Wesentlichen aus den Baugruppen „Antrieb“, „Greifer“, „Stoßfänger vorne“ und „Stoßfänger hinten“. Außerdem wird in diesem Abschnitt die Anordnung der Aktoren / Sensoren und ihres Anschlusses an die RCX-Baustein beschrieben.

B.1.1 Antrieb

Das Fahrwerk des Roboters verwendet 3 Räder (Abb. B-2). Davon werden zwei unabhängig angetrieben und eins ist als Freilaufrad realisiert. Diese Anordnung hat den Vorteil, dass der Roboter auf der Stelle wenden kann. Die Motoren sind mit den Ausgängen A und B des RCX 1 verbunden.

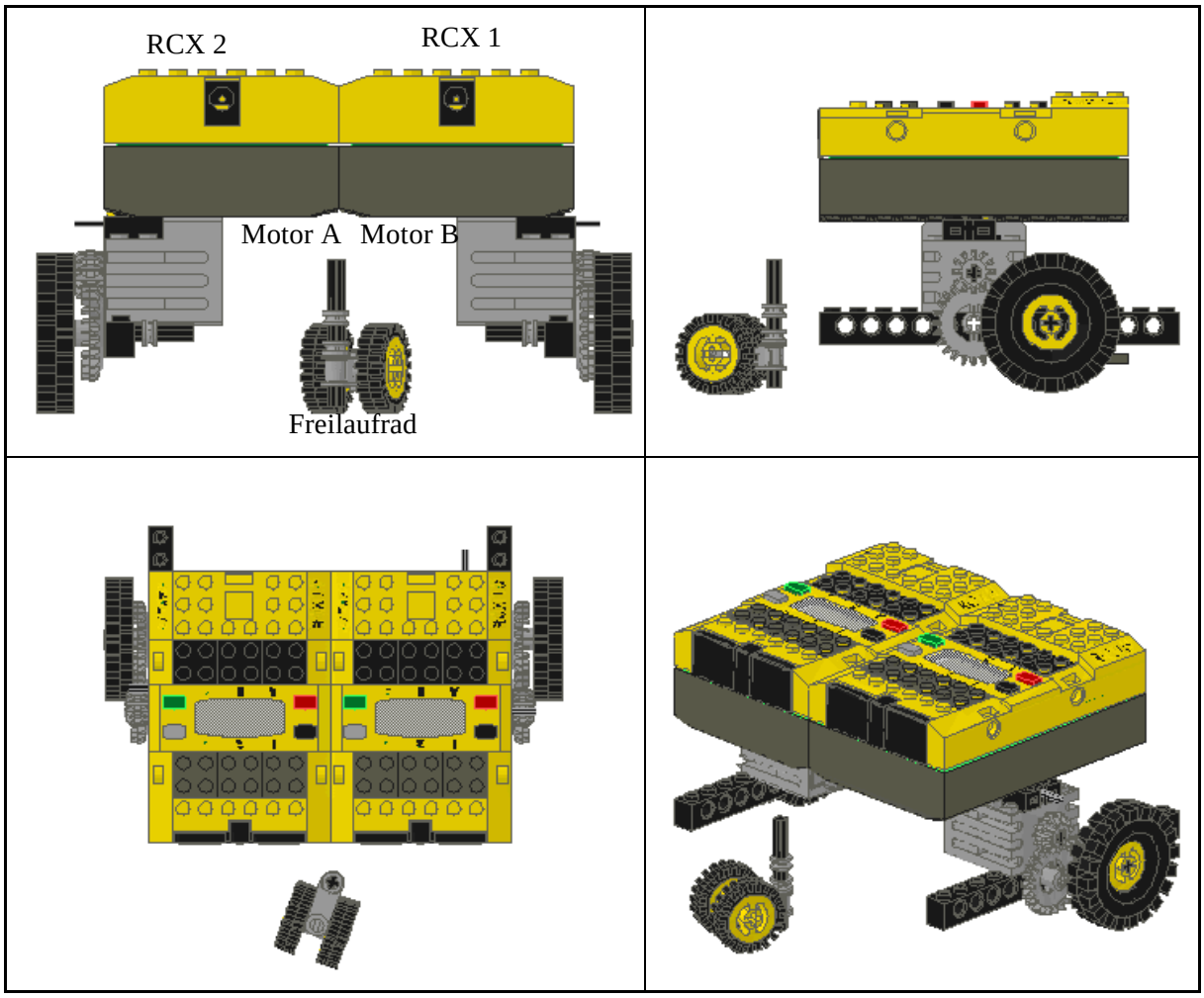


Abb. B-2: Antrieb und Radanordnung

B.1.2 Greifer

Abb. B-3 zeigt den Aufbau des Greifers. Die Mechanik des Greifers ist mit einem Wurmzahnrad realisiert worden, um genügend Kraft zu erzeugen. Wenn der Greifer am Anschlag angekommen ist, sorgt das weiße Schlupfzahnrad dafür, dass der Motor weiterlaufen kann. Trotzdem muss er irgendwann abgeschaltet werden. Die Tastsensoren bestimmen den richtigen Zeitpunkt dafür. Tastsensor 6 meldet, wenn der Greifer ganz geöffnet ist. Tastsensor 5 zeigt an, dass der Greifer komplett geschlossen wurde. Dies bedeutet, dass er kein Zielobjekt aufgenommen hat. Angeschlossen sind die Tastsensoren an die Schnittstellen 1 und 2 des RCX 2.

Zusätzlich ist der Greifer mit den beiden Lichtsensoren 1 und 3 ausgestattet. Diese befinden sich auf Höhe der Beleuchtung der Zielobjekte und ermöglichen eine grobe Lokalisation derselben. Außerdem gehört zu ihren Aufgaben das Auffinden des Zielgebiets. Angeschlossen sind sie an die Eingänge 1 und 3 des RCX 1.

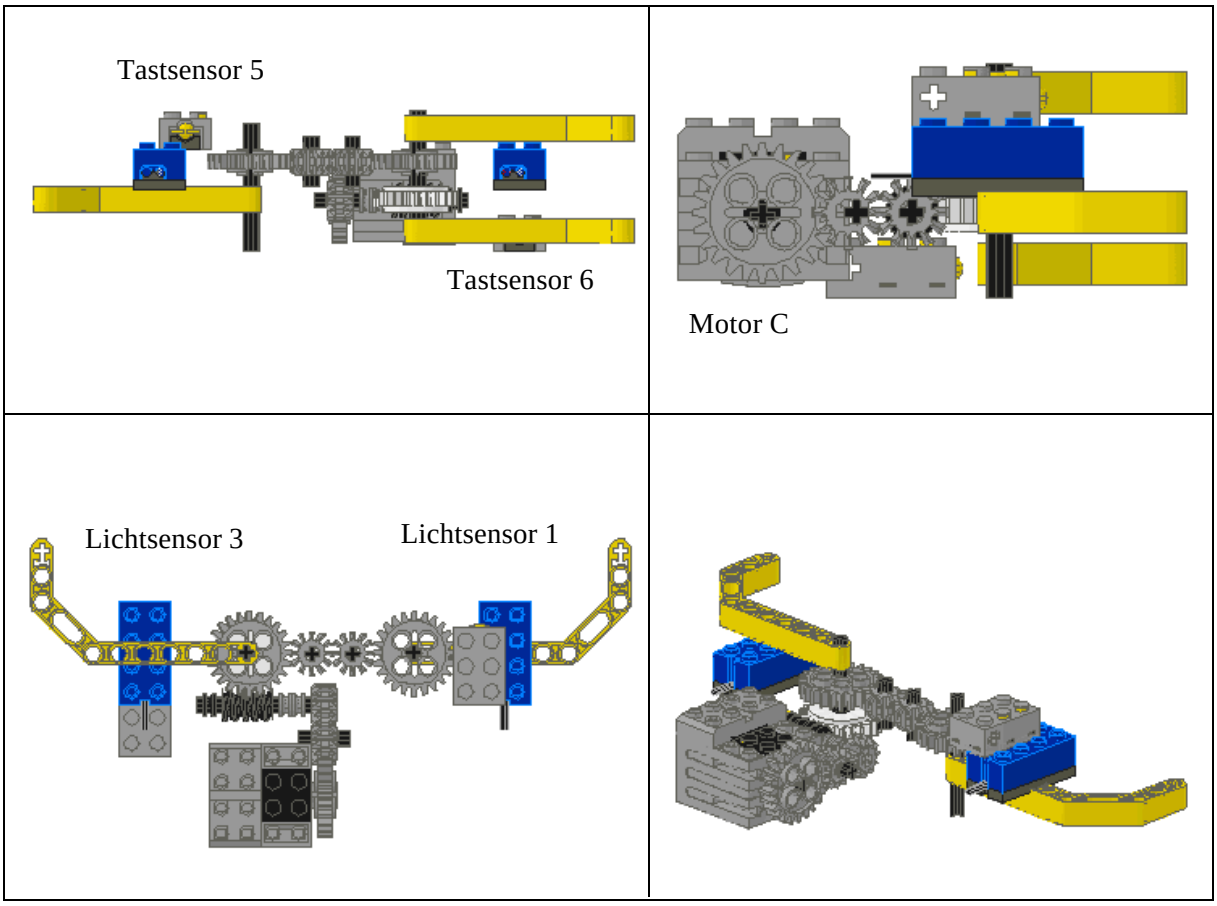


Abb. B-3: Aufbau des Greifers

Ein weiterer Lichtsensor befindet sich am vorderen Stoßfänger (Abb. B-4). Er ermöglicht es, den Roboter und damit den Greifer so auszurichten, dass sich das Zielobjekt genau vor dem Greifer befindet. Lichtsensor 2 ist an Eingang 2 des RCX 1 angeschlossen.

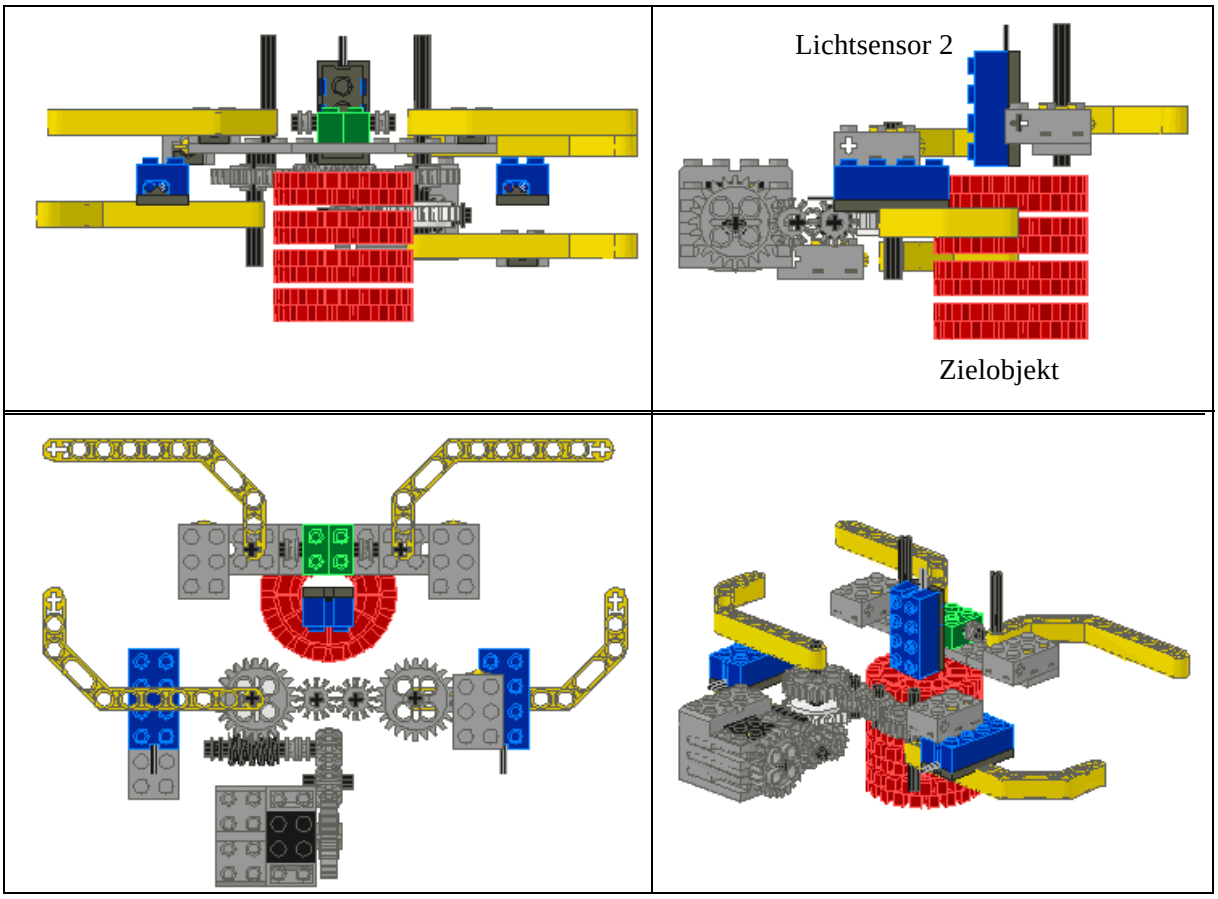


Abb. B-4: Greifer, vorderer Stoßfänger mit Lichtsensor und Zielobjekt

B.1.3 Stoßfänger

Die Stoßfänger entsprechen im wesentlichen dem in der „Robotics Invention System 1.5 - Constructopedia“ beschriebenen [LEGO99]. Der Hauptunterschied ist, dass die beiden Tastsensoren im hinteren Stoßfänger (Abb. B-6) beide mit Eingang 2 des RCX 1 verbunden sind. Es kann daher nicht unterschieden werden, ob der Roboter hinten links oder hinten rechts mit einem Hindernis kollidiert ist. Der hintere Stoßfänger ist zusätzlich mit einer Lampe ausgestattet, die mit Ausgang C des RCX 1 verbunden ist. Die Tastsensoren des vorderen Stoßfängers (Abb. B-5) sind mit den Eingängen 1 und 3 des RCX 1 verbunden. Zusätzlich befindet sich ein Lichtsensor am vorderen Stoßfänger. Dieser dient dazu, den Roboter so auszurichten, dass sich das Zielobjekt genau vor dem Greifer befindet. Der Lichtsensor ist mit Eingang 2 des RCX 1 verbunden.

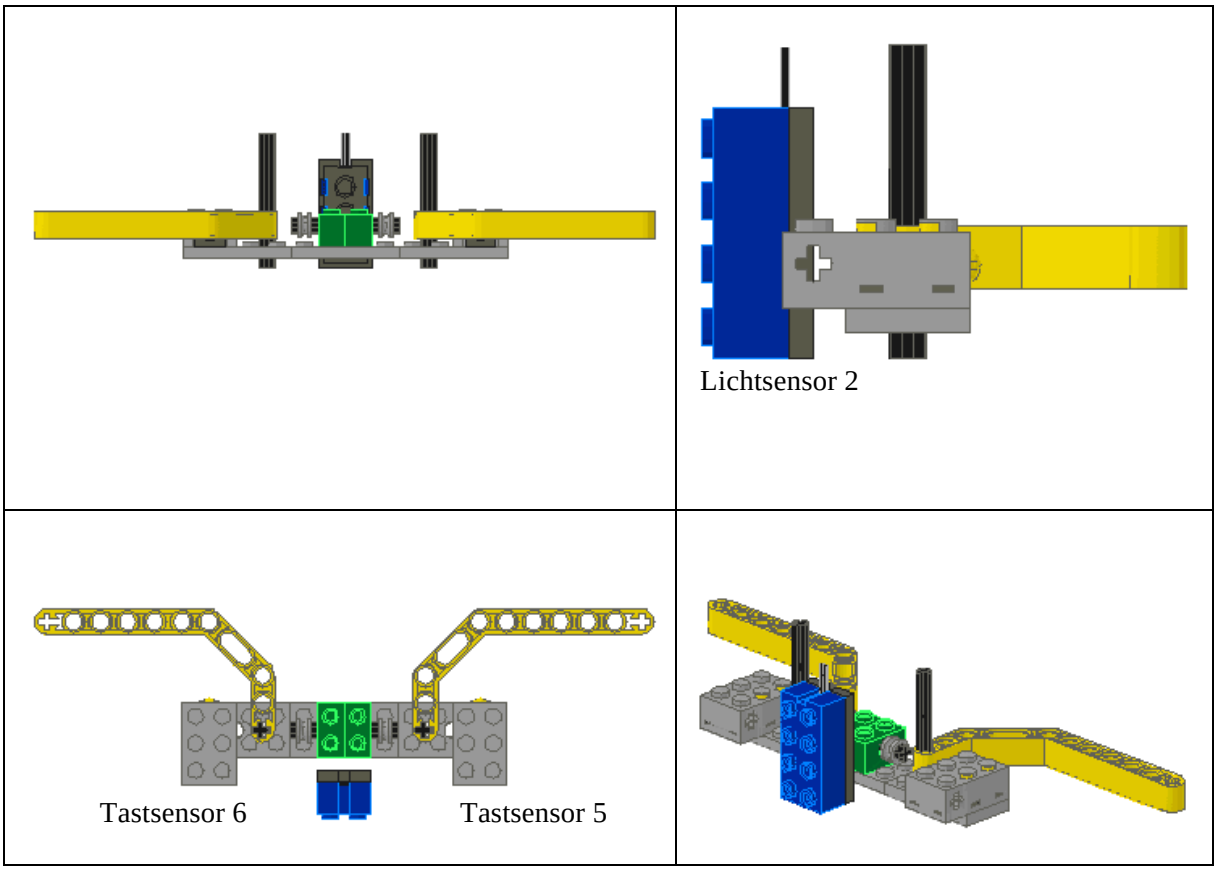


Abb. B-5: Vorderer Stoßfänger

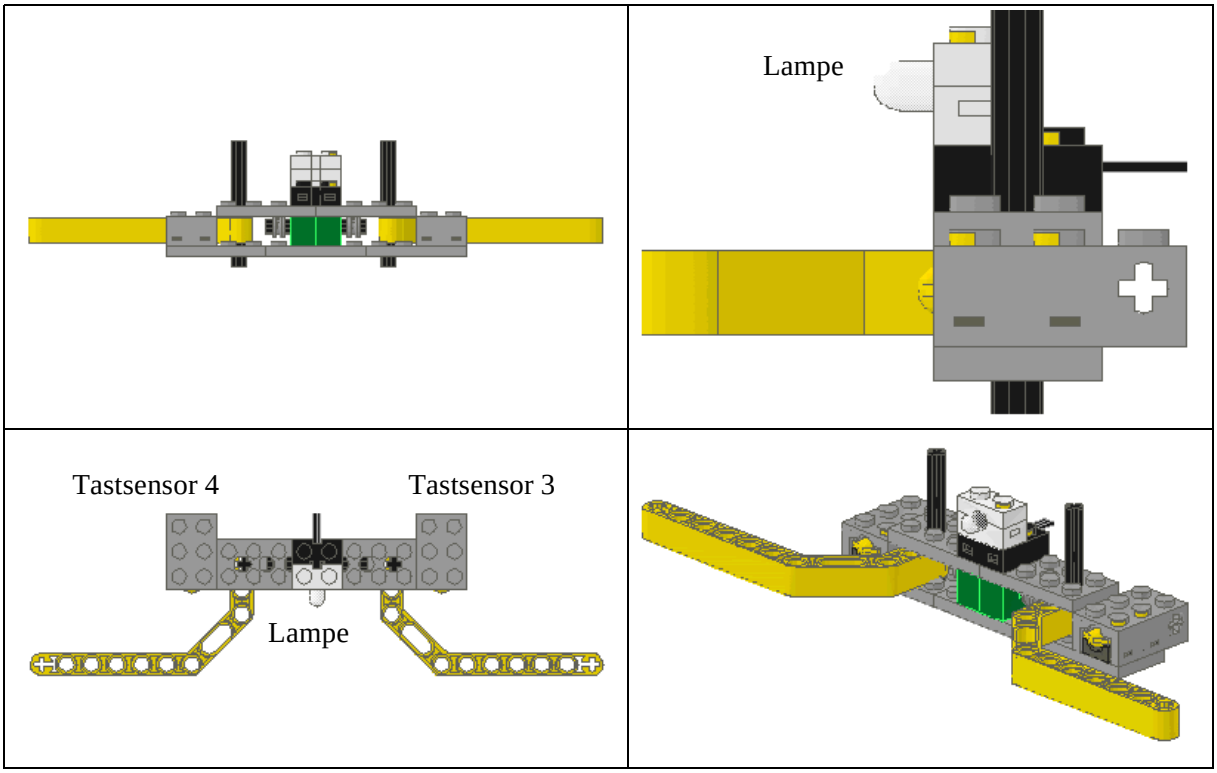


Abb. B-6: Hinterer Stoßfänger

B.1.4 Zusammenbau der Baugruppen

Der Roboter besteht aus den Baugruppen:

- Antrieb
- Greifer
- Stoßfänger vorne
- Stoßfänger hinten mit Lampe
- RCX Bausteine

Voraussetzung für das Funktionieren des Roboters ist die korrekte Anordnung der Sensoren / Aktoren und deren Anschluss an die beiden RCX-Bausteine. Wie die einzelnen Komponenten tatsächlich miteinander verbunden sind, ist weniger von Bedeutung.

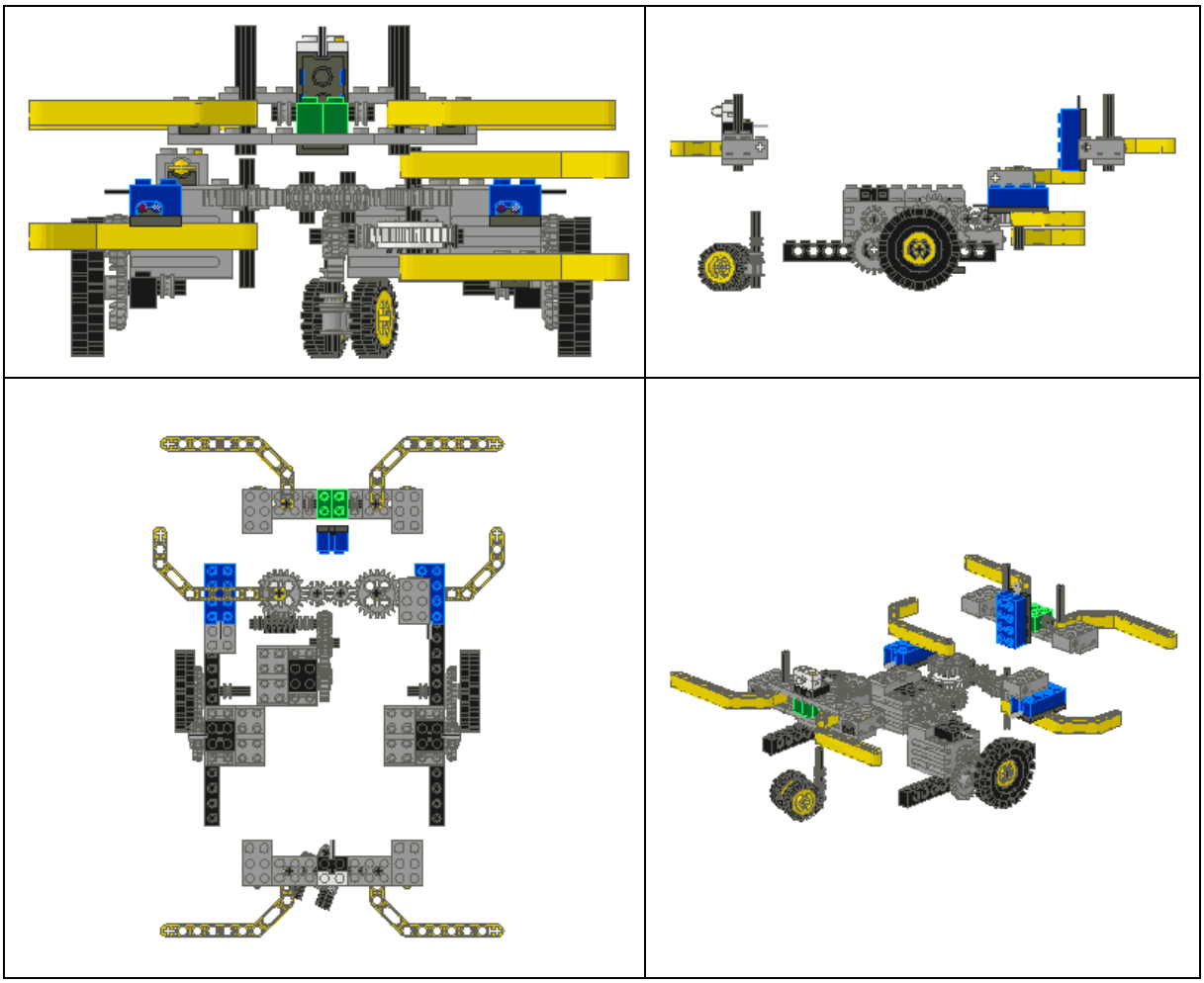


Abb. B-7: Stoßfänger, Greifer und Antrieb

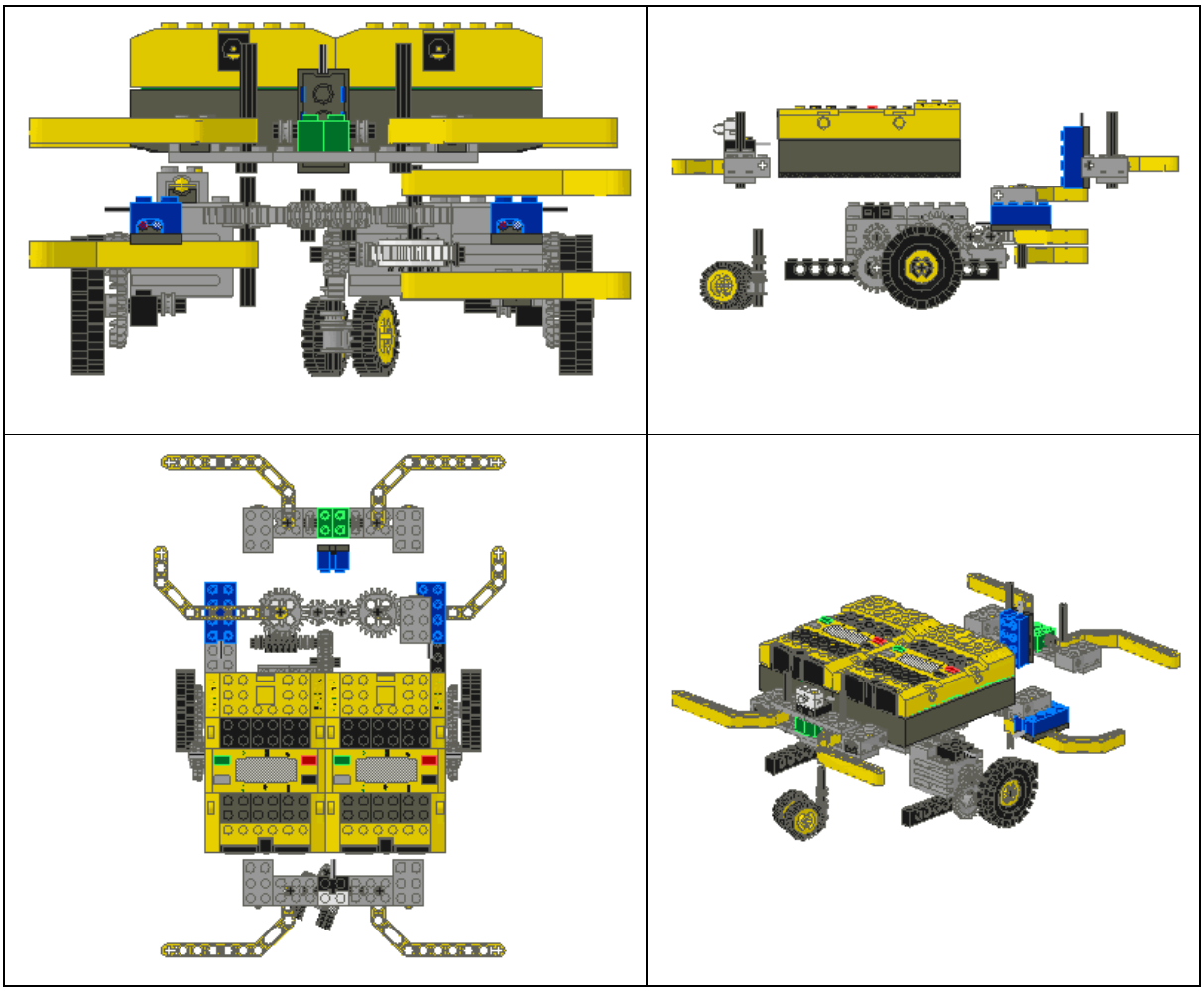


Abb. B-8: Stoßfänger, Greifer, Antrieb und RCX Bausteine

B.2 Anschluss der Aktoren und Sensoren

Der Roboter ist mit 6 Tastsensoren, 3 Lichtsensoren, 3 Motoren und einer Lampe ausgestattet. Die Anordnung der Sensoren und Aktoren zeigt Abb. B-9.

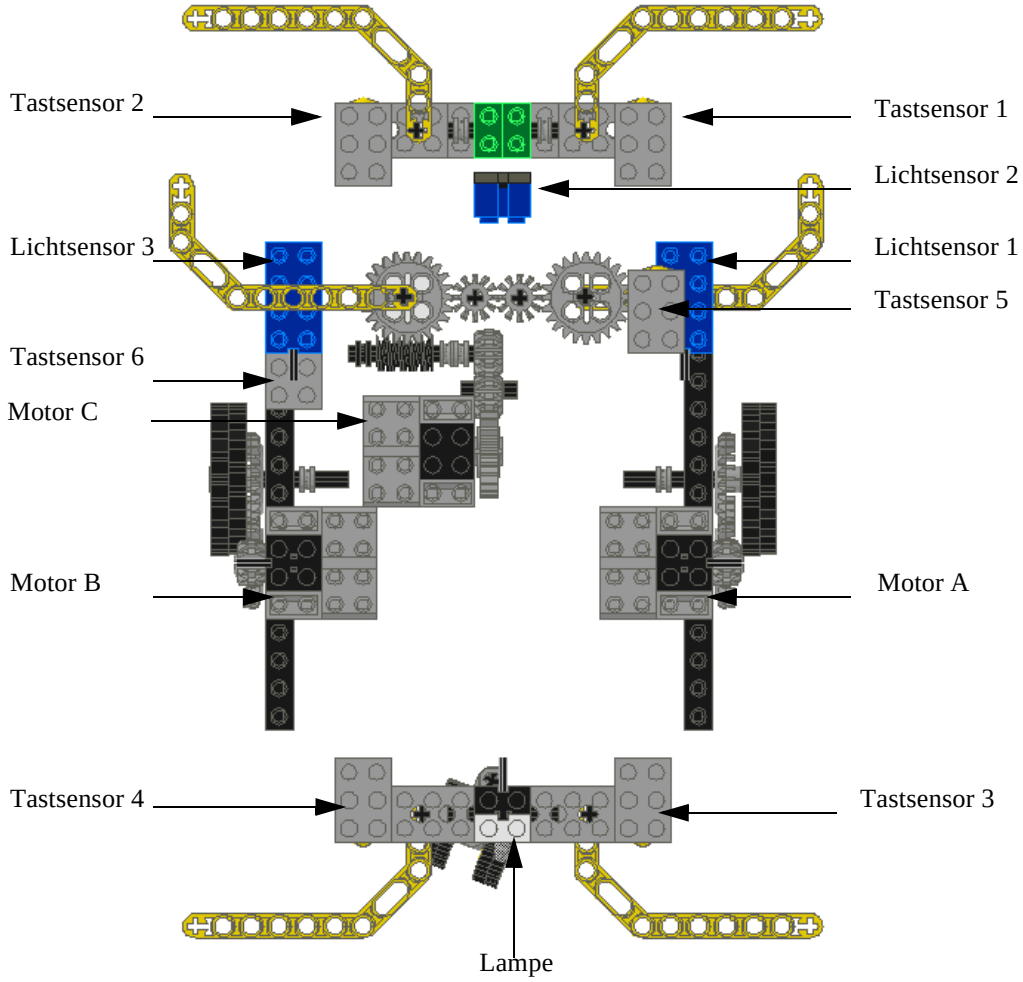


Abb. B-9: Aktor-/Sensoranordnung

Für das Funktionieren des Roboters ist die Sensoranordnung und der korrekte Anschluss an die Schnittstellen der beiden RCX Bausteine wesentlich. Die beiden Tabellen listen alle Sensoren und Aktoren und die jeweiligen Anschlüsse auf.

Typ		Nr.	RCX	Anschluss	Aufgabe
Sensoren	Tastsensor	1	1	1	Stellt Kollision rechts vorne fest.
	Tastsensor	2	1	3	Stellt Kollision links vorne fest.
	Tastsensor	3/4	1	2	Stellt Kollision hinten fest.
	Tastsensor	5	2	1	Meldet, dass der Greifer geschlossen ist und kein Zielobjekt aufgenommen wurde.
	Tastsensor	6	2	2	Meldet, dass der Greifer offen ist.
	Lichtsensor	1	1	1	Lichtquelle rechts finden.
	Lichtsensor	2	1	2	Zur Positionierung des Greifers vor dem Zielobjekt.
	Lichtsensor	3	1	3	Lichtquelle links finden.

Tabelle B-2 : Anschluss der Sensoren und Aktoren

Typ		Nr.	RCX	Anschluss	Aufgabe
Aktoren	Motor	A	1	A	Motor A treibt das rechte Rad und den Rotationssensor an.
	Motor	B	1	B	Motor B treibt das linke Rad an.
	Motor	C	2	C	Motor C öffnet und schließt den Greifer.
	Lampe		1	C	Kann ein- und ausgeschaltet werden.

Tabelle B-2 : Anschluss der Sensoren und Aktoren

B.3 Zielobjekte

Die Aufgabenstellung A verlangt, dass der Roboter Zielobjekte aufsammeln kann. Deren Aufbau und Strategien zu ihrer Lokalisation sind Thema dieses Abschnittes.

B.3.1 Versuch zur Erkennung von Zielobjekten

Das Hauptproblem des in Anhang A.1.1 und Kapitel 4.3 beschrieben Roboterverhaltens ist die Identifizierung der Zielobjekte. Dazu stehen drei Strategien zur Auswahl, die durch einen Versuch bewertet wurden. Die erste Variante funktioniert wie ein Radar. Der RCX-Baustein sendet Infrarotsignale aus, die von Hindernissen reflektiert werden. Die reflektierten Signale können mit einem Lichtsensor gemessen werden. Bei der zweiten Variante wird versucht, Zielobjekte nur durch die gemessenen Helligkeitsunterschiede zur Umgebung mit einem Lichtsensor zu messen. Die dritte Variante verwendet beleuchtete Zielobjekte, die durch einen Lichtsensor erkannt werden sollen. Die Beleuchtung wird durch eine 3 Volt Taschenlampenbirne realisiert.

Alle drei Varianten verwenden den Lichtsensor zur Erkennung der Zielobjekte. Dieser bietet die Möglichkeit, die gemessene Lichtstärke als Prozentwerte auszugeben. 0% bedeutet dunkel und 100% sehr hell. Ein Hindernis gilt als erkannt, wenn eine Lichtstärke von 70% gemessen wurde oder bei kontinuierlicher Messung der Lichtstärke ein Sprung von 10% auftritt.

Es werden 20 Messungen für jede Variante durchgeführt. In der Tabelle ist der absolute und der relative Wert der erfolgreich erkannten Zielobjekte aufgeführt. Bei der ersten Messreihe wird die Testumgebung von oben durch eine 40 Watt Glühbirne in einem Meter Höhe beleuchtet. Um das Verhalten der Sensoren bei wechselnden Beleuchtungsverhältnissen zu testen, wird eine zweite Versuchsreihe bei Tageslicht neben einem Fenster durchgeführt. Zusätzlich wird getestet, ob die Farbe der Zielobjekte einen Einfluss auf die Messergebnisse hat. Die Messergebnisse sind in Tabelle B-3 aufgeführt.

		Beleuchtung von oben						Beleuchtung durch Fenster					
		5 cm		10 cm		15 cm		5 cm		10 cm		15 cm	
		abs	%	abs	%	abs	%	abs	%	abs	%	abs	%
IR-Radar	s	16	80	9	45	0	0	15	75	7	35	0	0
	w	17	85	9	45	0	0	14	70	8	40	0	0
Lichtsensor Ziel unbeleuchtet	s	19	95	13	65	5	25	19	95	12	60	4	20
	w	19	95	12	60	4	20	20	100	11	55	4	20
Lichtsensor Ziel beleuchtet	s	20	100	17	85	12	60	20	100	16	80	11	55
	w	20	100	16	80	9	45	20	100	15	70	9	45
w: weißes Hindernis; s: schwarzes Hindernis													

Tabelle B-3 : Messergebnisse

In Tabelle B-4 sind die Messergebnisse ausgewertet und zusammengefasst worden. Die Messungen wurden vor Erstellung der Systemdokumentation durchgeführt, da die Messergebnisse die Aufgabenstellung beeinflusst haben.

Merkmal	IR-Radar	Lichtsensor, Ziel unbeleuchtet	Lichtsensor, Ziel beleuchtet
Zuverlässigkeit	-	o	+
Reichweite	-	+	+
Störlichtanfälligkeit	-	-	o
Aufwand	o	+	-
Wanderkennung	+	-	o
Unterscheidung Wand / Hindernis	-	-	+
Sonstige Nachteile	Es kann nur ein RCX-Baustein verwendet werden, da die IR-Schnittstelle als Radar benutzt wird.		Die Benutzeranforderungen müssen überarbeitet werden.
+: gut (1 Punkt); o: akzeptabel (0 Punkte); -: schlecht (-1 Punkt);			

Tabelle B-4 : Bewertung

Schlussfolgerung:

Unter der Voraussetzung, dass alle in der Tabelle aufgelisteten Aspekte die gleiche Wichtigkeit haben und die aufgeführten Nachteile mit einem Abzug von 1 Punkt bewertet werden, schneidet der Lichtsensor mit beleuchteten Zielobjekten mit einem Punkt am besten ab. Das IR-Radar landet mit -4 Punkten hinter dem normalen Lichtsensor mit -1 Punkt. Außerdem ist ein Zusammenhang zwischen der Anzahl der Treffer und der Zielobjektfarbe zu erkennen. Es werden daher beleuchtete schwarze Zielobjekte

verwendet. Um den Kontrast weiter zu verstärken, müssen die Wände ebenfalls schwarz sein. Da der Roboter keine Möglichkeit hat, das Ziel von einem Zielobjekt zu unterscheiden, darf die Beleuchtung im Ziel erst eingeschaltet werden, wenn der Roboter ein Zielobjekt aufgenommen hat.

B.3.2 Aufbau der Zielobjekte

Zur Beleuchtung der Zielobjekte wird eine 3 Volt Taschenlampenbirne verwendet. Die Stromversorgung erfolgt über zwei 1,5 V Microzellen. Als Gehäuse dient eine schwarze Filmdose. 6 Leuchtdioden werden ringförmig in Höhe der Lichtsensoren des Roboters angeordnet. Dies Birne wird im Deckel zentriert angebracht. Die Birne soll nur so lange leuchten, bis sie gegriffen wird. Dazu wird das Zielobjekt auf ein Stück Metallfolie gestellt. Durch die Kontakte an der Unterseite ist der Stromkreis geschlossen und die Birne leuchtet, solange das Zielobjekt nicht bewegt wird.

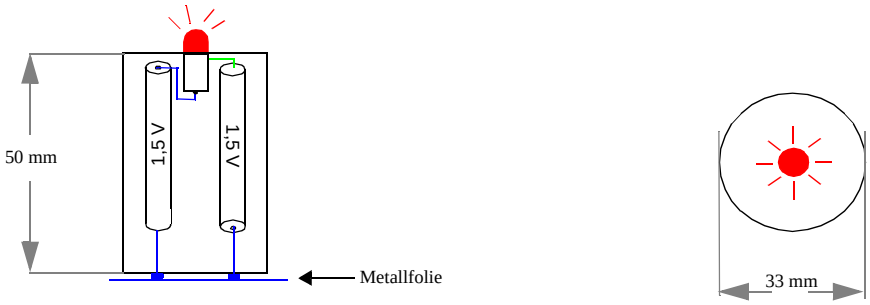


Abb. B-10: Aufbau und Beleuchtung der Zielobjekte

Anhang C

Code

C.1 Komponente GrabberSystem

```
import josx.platform.rcx.*;
/**
 * <b>variant</b>
 * <p align="justify">Die Hauptklasse für die Greiferhardware des Roboters.
 * Sie steuert den Greifer und kommuniziert mit dem DrivingSystem.</p><br>
 * Creation date: (07.05.01 20:37:38)<br>
 * @author: Stefan Josten
 */
public class GrabberSystem {
	/**
	 * Speichert den Greiferzustan (Erfolg/Misserfolg) zwischen.
	 */
	private static boolean failure = false;
	/**
	 * Empfangspuffer.
	 */
	private static byte[] bufferReceive = new byte[8] ;
	/**
	 * Sendepuffer für Nachricht 2 (Fehler)
	 */
	private static byte[] bufferFailure = new byte[] {(byte)0xf7, (byte) 2};
	/**
	 * Sendepuffer für Nachricht 3 (Erfolg).
	 */
	private static byte[] bufferOK = new byte[] {(byte)0xf7, (byte) 3};
/**
 * GrabberSystem Konstruktor.
 */
public GrabberSystem() {
}
/**
 * Realisierung des Greifvorgangs<br>
 * Creation date: (07.05.01 20:38:35)
 */
private static void grab() {
	TextLCD.print(„greif");
	Sound.beep();
	Motor.C.forward();
	Motor.C.setPower(7);
	int start = (int)System.currentTimeMillis();
	while (((int)System.currentTimeMillis() - start < 2500) && (testGrabberClosed() == false)) {
```

```
		}
		stopMotor();
		if (testGrabberClosed() == true)
			failure = true;
		else
			failure = false;
	}
	/**
	 * <p align="justify">Hauptschleife. GrabberSystem ist im Empfangsmodus und wartet auf Befehle. </p><br>
	 * Creation date: (07.05.01 21:24:44)
	 */
	public static void main(String[] aArg) {
		while (true) {
			receive();
			Sound.beep();
			Serial.readPacket(bufferReceive);
			if (bufferReceive[1] == (byte) 0) {
				grab();
				sendState(); // Nehrfach senden um Nachrichtenverlust abzufangen
				sendState();
				sendState();
			} else {
				if (bufferReceive[1] == (byte) 1)
					open();
			}
		}
	}
	/**
	 * Der Greifer wird geöffnet.<br>
	 * Creation date: (07.05.01 20:38:52)
	 */
	private static void open() {
		TextLCD.print(„loslassen");
		Sound.beep();
		Motor.C.backward();
		Motor.C.setPower(7);
		int start = (int) System.currentTimeMillis();
		while (((int) System.currentTimeMillis() - start < 2500) && (testGrabberOpen() != true));
		stopMotor();
	}
	/**
	 * <p align="justify">GrabberSystem warten auf einen Befehl zum Öffnen oder Schließen des Greifers. </p><br>
	 * Creation date: (07.05.01 21:24:44)
	 */
	private static void receive() {
		while (Serial.isPacketAvailable() == false);
		Sound.beep();
	}
	/**
	 * Der Status des Greifvorgangs wird gesendet.<br>
	 * Creation date: (07.05.01 20:39:21)
	 */
	private static void sendState() {
		for (int i = 0; i < 3; i++) { // Nehrfach senden um Nachrichtenverlust abzufangen
			if (failure == true)
				Serial.sendPacket(bufferFailure, 0, 2);
			else
				Serial.sendPacket(bufferOK, 0, 2);
		}
	}
	/**
	 * Der Greifermotor wird angehalten.<br>
	 * Creation date: (07.05.01 20:39:21)
```

```
 */
private static void stopMotor() {
	Motor.C.stop();
}
/**
 * <p align="justify">Es wird überprüft, ob der Greifer geschlossen ist. In diesem
 * Fall ist der Greifvorgang fehlgeschlagen.</p><br>
 * Creation date: (07.05.01 20:40:40)<br>
 * @return boolean
 */
private static boolean testGrabberClosed() {
	return Sensor.S1.readBooleanValue();
}
/**
 * Es wird überprüft, ob der Greifer offen ist.<br>
 * Creation date: (07.05.01 20:40:40)<br>
 * @return boolean
 */
private static boolean testGrabberOpen() {
	return Sensor.S2.readBooleanValue();
}
}
```

C.2 Komponente DrivingSystem

C.2.1 Klasse DrivingSystem

```
import josx.platform.rcx.*;
/**
 * <p align="justify">Die Hauptklasse für den fahrbaren Teil des Roboters.
 * Sie steuert das Fahrzeug und kommuniziert mit dem GrabberSystem.</p>
 * Creation date: (09.05.01 11:54:50)<br>
 * @author: Stefan Josten
 */
public class DrivingSystem {
	/**
	 * <b>variant</b> Sendepuffer für Nachricht 0 (Greifen), (0xf7 Opcode für Senden)
	 */
	private static byte[] bufferGrab = new byte[] { (byte)0xf7,(byte) 0};
	/**
	 * <b>variant</b> Sendepuffer für Nachricht 1 (Öffnen)
	 */
	private static byte[] bufferOpen = new byte[] { (byte)0xf7,(byte) 1};
	/**
	 * <b>variant</b> Puffer für den Nachrichtenempfang
	 */
	private static byte[] bufferReceive = new byte[8];
	/**
	 * Instanz von MotorControl
	 */
	private static MotorControl motorControl = new MotorControl();
	/**
	 * Instanz von CollisionDetect
	 */
	private static CollisionDetect collisionDetect = new CollisionDetect();
	/**
	 * Zeigt an ob der Roboter im Ziel angekommen ist
	 */
```

```
	private static boolean atHome = false;
	// Konstanten
	/**
	 * Verzögerung zwischen Drehung und Geradeausfahrt in ms
	 */
	private static final int DELAY = 100;
	/**
	 * Zeit für die Fahrt in die entgegengesetzte Richtung bei escape() in ms
	 */
	private static final int ESCAPE_TIME = 500;
	/**
	 * Zeit für die Drehung bei escape() in ms
	 */
	private static final int ESCAPE_TURN_TIME = 300;
	/**
	 * Lichtstärke im Ziel
	 */
	private static final int HOME = 85;
	/**
	 *  <b>variant</b> Lichtstärke über Zielobjekt
	 */
	private static final int TARGET = 80;
	/**
	 * Geschwindigkeit des Fahrzeugs. Werte zwischen 1 und 7 sind zulässig.
	 */
	private static final int SPEED = 3;
	/**
	 * Zeit bis zur nächsten Lichtsuche in ms
	 */
	private static final int TIME_TO_NEXT_SCAN = 5000;
	/**
	 * Zeit für eine Umdrehung in ms
	 */
	private static final int TIME_TO_TURN = 2700;
	/**
	 *  <b>variant</b> Zeigt an, ob das Zielobjekt gegriffen wurde
	 */
	 static private boolean targetGrabed=false;
/**
 * DrivingSystem Konstruktor.
 */
public DrivingSystem() {
}
/**
 * Fährt zum Ziel.<br>
 * Creation date: (09.05.01 11:56:12)
 */
private static void driveHome() {
	int activtime = 0;
	while (atHome == false) {
		searchLight();
		TextLCD.print(„dh“);
		activtime = (int) System.currentTimeMillis();
		while (collisionDetect.collision == false && (int) System.currentTimeMillis() - activtime <
TIME_TO_NEXT_SCAN) {
			driveLight();
		}
		while (collisionDetect.collision == true && atHome == false)
			escape();
	}
	motorControl.stopMotor();
}
/**
```

```
 * Fährt zur Lichtquelle.<br>
 * Creation date: (09.05.01 11:56:12)
 */
private static void driveLight() {
	int starttime = 0;
	if (Sensor.S1.readValue() < 0.9 * Sensor.S3.readValue())
		motorControl.turnLeft();
	else
		if (0.9 * Sensor.S1.readValue() > Sensor.S3.readValue())
			motorControl.turnRight();
	starttime = (int) System.currentTimeMillis();
	while ((int) System.currentTimeMillis() - starttime < DELAY && collisionDetect.collision == false);
	motorControl.driveForward();
	starttime = (int) System.currentTimeMillis();
	while ((int) System.currentTimeMillis() - starttime < DELAY && collisionDetect.collision == false);
	motorControl.stopMotor();
}
/**
 * <b>variant</b><br>
 * Fährt zum Zielobjekt.<br>
 * Creation date: (09.05.01 11:57:07)
 */
private static void driveTarget() {
	int activtime = 0;
	while (targetGrabed == false) {
		while (Sensor.S2.readValue() <= TARGET) {
			searchLight();
			TextLCD.print(„dt“);
			activtime = (int) System.currentTimeMillis();
			while (collisionDetect.collision == false && Sensor.S2.readValue() <= TARGET && (int)
System.currentTimeMillis() - activtime < TIME_TO_NEXT_SCAN) {
				driveLight();
			}
			while (collisionDetect.collision == true && Sensor.S2.readValue() <= TARGET)
				escape();
		}
		motorControl.stopMotor();
		sendGrab(); // Um Nachrichtenverlust abzufangen
		sendGrab();
		sendGrab();
		receiveGrabberState();
	}
}
/**
 * Weicht nach einer Kollision sinnvoll aus.<br>
 * Creation date: (09.05.01 11:56:48)
 */
private static void escape() {
	// Zwischenspeichern der Kollisionsquelle:
	boolean collisionFrontLeft = collisionDetect.collisionFrontLeft();
	boolean collisionFrontRight = collisionDetect.collisionFrontRight();
	boolean collisionRear = collisionDetect.collisionRear();
	TextLCD.print(„esc“);
	Sound.beep();
	int starttime = (int) System.currentTimeMillis();
	testHome();
	// In entgegengesetzte Richtung ausweichen:
	if (collisionRear == true)
		motorControl.driveForward();
	else {
		if (atHome == false)
			motorControl.driveBackward();
	}
```

```
		// Nur wenn das Ziel nicht erreicht ist, ausweichen:
		if (atHome == false) {
			while ((int) System.currentTimeMillis() - starttime < ESCAPE_TIME);
			// Drehen:
			if (collisionFrontLeft == true)
				motorControl.turnRight();
			else
				motorControl.turnLeft();
			starttime = (int) System.currentTimeMillis();
			// Warten bis Drehung abgeschlossen. Erneute Kollision wird abgefangen:
			while ((int) System.currentTimeMillis() - starttime < ESCAPE_TURN_TIME && collisionDetect.collision
== false) {
			}
			motorControl.stopMotor();
		}
	}
	/**
	 * Steuert die einzelnen Verhaltensweisen.<br>
	 * Creation date: (09.05.01 11:55:30)<br>
	 * @param aArg java.lang.String[]
	 */
	public static void main(String[] aArg) {
		Sensor.S1.activate();
		Sensor.S2.activate();
		Sensor.S3.activate();
		motorControl.setSpeed(SPEED);
		/* variant */driveTarget();
		driveHome();
		/* variant */switchLampON();
		/* variant */sendOpen();
	}
	/**
	 * <b>variant</b><br>
	 * Empfängt die Nachricht über den Greiferstatus.<br>
	 * Nachricht 3: OK<br>
	 * Nachricht 2: Fehler<br>
	 * Creation date: (09.05.01 11:59:36)
	 */
	private static void receiveGrabberState() {
		while (Serial.isPacketAvailable() == false);
		Serial.readPacket(bufferReceive);
		if ((int) bufferReceive[1] == (byte) 3) { // bufferReceive[0] enthält Opcode für Nachricht empfangen
			TextLCD.print(„OK“);
			targetGrabed = true;
		} else {
			TextLCD.print(„Fehler“);
			targetGrabed = false;
			sendOpen(); // Um Nachrichtenverlust abzufangen mehrfach versenden
			sendOpen();
			sendOpen();
		}
	}
	/**
	 * Sucht die Lichtquelle.<br>
	 * Creation date: (09.05.01 11:55:59)
	 */
	private static void searchLight() {
		int a = 0;
		int max = 0;
		int starttime = (int) System.currentTimeMillis();
		TextLCD.print(„sl1“);
		if (collisionDetect.collision == false) {
			motorControl.turnLeft();
```

```
		}
		// Sectorscan: hellste Lichtstärke finden
		while (((int) System.currentTimeMillis() - starttime) < TIME_TO_TURN && collisionDetect.collision == false) {
			a = Sensor.S1.readValue() + Sensor.S3.readValue();
			if (a > max)
				max = a;
		}
		// Weiterdrehen bis zum hellsten Punkt:
		Sound.beep();
		TextLCD.print(„sl2“);
		a = 0;
		starttime = (int) System.currentTimeMillis();
		while ((a < 0.95 * max) && ((int) System.currentTimeMillis() - starttime < TIME_TO_TURN) && collisionDe-
tect.collision == false)
			a = Sensor.S1.readValue() + Sensor.S3.readValue();
		motorControl.stopMotor();
}
/**
 * <b>variant</b><br>
 * Sendet die Nachricht zum Schließen des Greifers:<br>
 * Nachricht 0<br>
 * Creation date: (09.05.01 11:58:19)
 */
private static void sendGrab() {
		Serial.sendPacket(bufferGrab, 0, 2);
		Sound.beep();
}
/**
 * <b>variant</b><br>
 * Sendet die Nachricht zum Öffnen des Greifers:<br>
 * Nachricht 1<br>
 * Creation date: (09.05.01 11:58:19)
 */
private static void sendOpen() {
		Serial.sendPacket(bufferOpen, 0, 2);
		Sound.beep();
}
/**
 * <b>variant</b><br>
 * Schaltet die Lampe an Ausgang B ein.<br>
 * Creation date: (09.05.01 11:57:31)
 */
private static void switchLampON() {
		motorControl.lampON();
}
/**
 * Überprüft ob der Roboter sich im  Ziel befindet.<br>
 * Creation date: (21.05.01 20:00:48)
 */
private static void testHome() {
		// Da Lichtsensoren und Kollisionssensoren gekoppelt sind, muss erst zurückgesetzt werden um
		// gültige Messwerte der Lichtsensoren zu erhalten: Dabei wird über die Zeit der Fall einer
		// erneuten Kollision angefangen
		motorControl.driveBackward();
		int starttime = (int) System.currentTimeMillis();
		while (collisionDetect.collision = true && (int) System.currentTimeMillis() - starttime < (int) (ESCAPE_TIME / 2));
		motorControl.stopMotor();
		if (Sensor.S1.readValue() + Sensor.S3.readValue() > HOME)
			atHome = true;
}
}
```

C.2.2 Klasse CollisionDetect

```
import josx.platform.rcx.*;
/**
 * <p align="justify">Die Klasse CollisionDetect überwacht die Sensoren. Bei Veränderung der Eingangswerte wird
 * stateChanged() aufgerufen. Diese Methode überprüft erst ob sich auch der boolesche Wert des
 * Eingangs verändert hat, bevor das Attribut collision gesetzt wird.</p>
 * Creation date: (17.05.01 20:48:14)<br>
 * @author: Stefan Josten
 */
public class CollisionDetect implements josx.platform.rcx.SensorListener {
	/**
	 * Wird automatisch von der Methode stateChanged() gesetzt.
	 */
	public boolean collision = false;
/**
 * CollisionDetect Konstruktor.<br>
 * Die zu überwachenden Sensoren werden bei SensorListener angemeldet.
 */
public CollisionDetect() {
	Sensor.S1.addSensorListener(this);
	Sensor.S2.addSensorListener(this);
	Sensor.S3.addSensorListener(this);
	collision = false;
}
/**
 * Überprüft ob eine Kollision stattgefunden hat.<br>
 * Creation date: (21.05.01 17:42:31)
 * @return boolean
 */
public boolean collisionFrontLeft() {
	if (Sensor.S3.readValue() > 98)
		return true;
	else
		return false;
}
/**
 * Überprüft ob eine Kollision stattgefunden hat.<br>
 * Creation date: (21.05.01 17:42:31)
 * @return boolean
 */
public boolean collisionFrontRight() {
	if (Sensor.S1.readValue() > 98)
		return true;
	else
		return false;
}
/**
 * Überprüft ob eine Kollision stattgefunden hat.<br>
 * Creation date: (21.05.01 17:42:31)
 * @return boolean
 */
public boolean collisionRear() {
	if (Sensor.S2.readValue() > 98)
		return true;
	else
		return false;
}
/**
 * <p align="justify">stateChanged wird bei Änderung der Messwerte an den Eingängen aufgerufen. Bevor das Attribut
 * collision gesetzt wird, wird erst überprüft, ob sich auch der boolsche Wert des Eingangs
 * geändert hat.</p><br>
 * Creation date: (21.05.01 17:43:35)
```

```
 */
public void stateChanged(josx.platform.rcx.Sensor aSource, int aOldValue, int aNewValue) {
	if (Sensor.S1.readValue() >= 98 || Sensor.S2.readValue() >= 98 || Sensor.S3.readValue() >= 98) {
		if (collision == false) { //Noch nicht gestoppt
			Motor.A.stop();
			Motor.C.stop();
		}
		collision = true;
	} else
		collision = false;
}
}
```

C.2.3 Klasse MotorControl

```
import josx.platform.rcx.*;
/**
 * <p align="justify">Die Klasse MotorControl stellt einfache Fahrverhaltensweisen zur Verfgung.
 * Dazu mssen an Ausgang A und C Motoren angeschlossen sein. Wenn es nicht funktioniert
 * kann das an der falschen Polung der Motoren liegen. (Stecker umdrehen)<p>
 * Creation date: (11.05.01 15:51:04)<br>
 * @author: Stefan Josten
 */
public class MotorControl {
	/**
	 * Attribut fr die Motorgeschwindigkeit.
	 */
	private int speed = 7;
/**
 * MotorControl Konstruktor.
 */
public MotorControl() {
}
/**
 * Fhrt rckwrts.<br>
 * Creation date: (17.05.01 20:15:20)
 */
public void driveBackward() {
	Motor.A.backward();
	Motor.C.backward();
	Motor.A.setPower(speed);
	Motor.C.setPower(speed);
}
/**
 * Fhrt vorwrts<br>
 * Creation date: (17.05.01 20:15:20)
 */
public void driveForward() {
	Motor.A.forward();
	Motor.C.forward();
	Motor.A.setPower(speed);
	Motor.C.setPower(speed);
}
/**
 * <b>variant</b><br>
 * Die an Ausgang B angeschlossene Lampe wird eingeschaltet.<br>
 * Creation date: (17.05.01 20:15:20)
 */
public void lampON() {
	Motor.B.forward();
```

```
		Motor.B.setPower(7);
}
/**
 * Setzt das Attribut speed.<br>
 * Creation date: (17.05.01 20:15:20)
 * @param power int
 */
public void setSpeed(int power) {
		speed = power;
}
/**
 * Die Motoren werden angehalten.<br>
 * Creation date: (17.05.01 20:15:20)
 */
public void stopMotor() {
		Motor.A.stop();
		Motor.C.stop();
}
/**
 * Linksdrehung auf der Stelle.<br>
 * Creation date: (17.05.01 20:15:20)
 */
public void turnLeft() {
		Motor.A.forward();
		Motor.C.backward();
		Motor.A.setPower(speed);
		Motor.C.setPower(speed);
}
/**
 * Rechtsdrehung auf der Stelle.<br>
 * Creation date: (17.05.01 20:15:20)
 */
public void turnRight() {
		Motor.A.backward();
		Motor.C.forward();
		Motor.A.setPower(speed);
		Motor.C.setPower(speed);
}
}
```