

Diplomarbeit

Entwicklung einer eingebetteten, komponentenbasierten Telematikanwendung im Fahrzeug unter Verwendung von UML

Guido Laures
Matrikelnummer 181091

Betreuer: Prof. Dr. Stefan Jähnichen

April 2000

Technische Universität Berlin
Institut für Software- und Systemgestaltung
Forschungsgruppe Softwaretechnik
10587 Berlin

Inhaltsverzeichnis

1	<i>Einleitung</i>	5
1.1	<i>Ziel</i>	5
1.2	<i>Kontext und Motivation</i>	5
1.3	<i>Vorgehensweise</i>	6
1.4	<i>Aufbau</i>	7
1.5	<i>Informatik und die deutsche Sprache</i>	7
1.6	<i>Danksagung</i>	8
1.7	<i>Erklärung</i>	8
2	<i>UML</i>	9
2.1	<i>Motivation</i>	9
2.2	<i>Ziele</i>	10
2.3	<i>Sichten</i>	10
2.4	<i>Statische Sicht</i>	10
2.5	<i>Use Case Sicht</i>	15
2.6	<i>State Machine Sicht</i>	16
2.7	<i>Aktivitätssicht</i>	18
2.8	<i>Interaktionssicht</i>	18
2.9	<i>Physikalische Sicht</i>	20
2.10	<i>Modellmanagement Sicht</i>	21
3	<i>Komponenten</i>	23
3.1	<i>Motivation</i>	23
3.2	<i>Historie</i>	23
3.3	<i>Schnittstellen</i>	24
3.4	<i>Vorteile</i>	25
4	<i>Eingebettete Systeme</i>	29
4.1	<i>Motivation</i>	29
4.2	<i>Hardware</i>	29
4.3	<i>Beispiele und Eigenheiten</i>	30
5	<i>Entwicklungsprozess für verteilte Telematik</i>	33
5.1	<i>Motivation</i>	33
5.2	<i>Was ist ein Software Entwicklungsprozess?</i>	34
5.3	<i>Etablierte Prozesse</i>	34
5.4	<i>Use Case Basierung</i>	35
5.5	<i>Architektur-Zentrierung</i>	39
5.6	<i>Iterativ und inkrementell</i>	43
5.7	<i>Modellkonsistenz</i>	50
5.8	<i>Prozessmuster</i>	52
5.9	<i>Nachrichten und Ereignisse</i>	54
5.10	<i>Threads und Ressourcen</i>	56
5.11	<i>Zusammenfassung</i>	57
6	<i>Entwurf einer verteilten Telematikanwendung</i>	59
6.1	<i>Notation</i>	59
6.2	<i>Motivation</i>	59
6.3	<i>Anforderungen</i>	60
6.4	<i>Einteilung in Packages</i>	64
6.5	<i>Startsequenz</i>	66
6.6	<i>String-Verwaltung</i>	67
6.7	<i>Konfiguration</i>	68
6.8	<i>Komponentenverwaltung</i>	69

6.9 Zugriffs- und Benutzerverwaltung	71
6.10 Geräte	76
6.11 Applikationen	92
6.12 Grafische Benutzungsschnittstelle (Human Machine Interface –HMI)	94
6.13 Monitoring und Test	97
6.14 CD-Spieler	98
7 Bewertung der angewandten Methoden und Werkzeuge	101
7.1 UML	101
7.2 Komponentenarchitektur	102
7.3 Programmiersprache und API	103
7.4 Entwicklungsprozess	103
7.5 CASE-Tool: Together	105
8 Anhang	115
8.1 Entwurfsmuster	115
8.2 Java und Internationalisierung	118
9 Quellen	119

1 Einleitung

1.1 Ziel

Diese Arbeit soll Antworten auf die Frage liefern, welche Methoden und Werkzeuge für einen Entwicklungsprozess für verteilte Telematik-Anwendungen besonders geeignet sind. Zu diesem Zweck werden zunächst einige allgemeine Techniken des Softwareentwurfs genauer untersucht. Die Ergebnisse daraus fließen dann in eine Zusammenstellung der wichtigsten Eigenschaften eines Entwicklungsprozesses für Telematikanwendungen ein.

Die praktische Anwendbarkeit dieser Prozesseigenschaften im Telematik-Bereich wird anhand der Implementation einer verteilten Architektur untersucht. Anschließend werden die bei der Entwicklung verwendeten Techniken und Werkzeuge einer Bewertung unterzogen.

1.2 Kontext und Motivation

In den letzten Jahren hat der Anteil an elektronischen Steuerungs-Komponenten, die in Fahrzeugen eingesetzt werden, stark zugenommen. Sie ersetzen und unterstützen immer mehr analoge Techniken, um die Sicherheit und Verlässlichkeit der Fahrzeuge und deren Handhabung zu verbessern. Zudem hat die Elektronik auch in vielen anderen Bereichen des Fahrzeugs Einzug gehalten. Softwarekomponenten für Navigation, Kommunikation über Mobilfunknetze oder auch digitale Video- und Audio-Funktionen gehören in modernen Fahrzeugen der Luxusklasse bereits zum Standard. Es besteht jedoch immer noch ein Problem darin, alle elektronischen Bestandteile in einem modernen standardisierten Gesamtkonzept zu integrieren.

Aus diesem Grund formieren sich Entwicklungsgruppen, die einen Standard für derartige Telematik-Systeme erarbeiten. Unter dem Begriff Telematik werden hier Teileinheiten der Fahrzeugelektronik wie Navigations-, Warn- und Kommunikationssystem zusammengefasst. Ziel ist es, die Kommunikation von im Fahrzeug befindlichen Komponenten zu ermöglichen. So ist es zum Beispiel wünschenswert, dass ein Warnsystem die Möglichkeit hat, ein laufendes Entertainmentssystem zu unterbrechen, um auf elektronische Ressourcen des Fahrzeugs zuzugreifen. Hierbei hat die Verteiltheit der einzelnen Komponenten erheblichen Einfluss auf die Systemarchitektur.

Bisher existierten nur wenige Berührungspunkte zwischen reinem Softwareentwurf und Automobilindustrie. Der Entwurf von elektronischen Steuerelementen wird eher von Experten im Gebiet der Elektro- und Steuerungstechnik als von Informatikern vorgenommen. Die immer stärkere Nachfrage nach integrierten Anwendungen im Fahrzeug, wie z.B. Webzugriff, macht jedoch auch die Involvierung von Spezialisten im Bereich der Soft- und Hardwarearchitektur notwendig.

Die Zusammenarbeit der einzelnen, an der Entwicklung eines Fahrzeugs beteiligten Gruppen kann nur funktionieren, wenn ein festgelegter Prozess den Entwicklungsverlauf regelt. Zudem ist es nötig, eine gemeinsame Sprache zu finden, die eine reibungslose und unmissverständliche Kommunikation über den Problembereich zwischen allen Projektbeteiligten ermöglicht.

1.3 Vorgehensweise

Diese Arbeit gibt in einem ersten, theoretischen Teil zunächst einen Überblick über die beim Entwurf von verteilten Telematik-Anwendungen relevanten Techniken. Zu ihnen gehört die mittlerweile etablierte Software-Modellierungssprache Unified Modelling Language (UML). Diese bietet die Möglichkeit, alle Eigenschaften eines Systems aus verschiedenen Blickwinkeln darzustellen. Die Arbeit untersucht, ob die UML als Kommunikationsmittel zwischen allen in einem Telematik-Entwicklungsprozess involvierten Personengruppen geeignet ist.

Eine Technik, die in den letzten Jahren immer mehr Verbreitung in der Softwareentwicklung verzeichnet, sind Komponenten. Diese bieten sich vor allem bei verteilten Systemen an. Nach einer Erläuterung der Technik, folgt eine Untersuchung der Vor- und Nachteile bei der Verwendung von Komponenten in einem Entwicklungsprozess.

Zudem werden die speziellen Anforderungen von eingebetteten Systemen, wie die in einem Fahrzeug, an eine Softwareentwicklung untersucht und zusammengestellt. Es folgt eine Prüfung der Ergebnisse auf ihre Relevanz bezüglich des Entwicklungsprozesses.

Der zweite Teil der theoretischen Grundlagen diskutiert die wichtigsten Eigenschaften eines objekt-orientierten Entwicklungsprozess für verteilte Telematikanwendungen. Diese werden aus den Prozessen *Unified Software Development Process*, *Rapid Object-Oriented Process for Embedded Systems* (ROPES) und dem *Catalysis*-Ansatz zusammengetragen und auf den Telematikbereich bezogen.

Zu den wichtigsten Eigenschaften des Prozesses gehören neben der Verwendung von UML und Komponenten auch die Zentrierung auf die Architektur des Systems. Die Verwendung des iterativen Ansatz zum Softwareentwurf wird ebenso als wichtiger Bestandteil des Prozesses herausgearbeitet wie die Erstellung von Prototypen.

Der praktische Teil der Arbeit testet die erarbeiteten Prozesseigenschaften anhand einer Beispiel-Entwicklung. Es wird ein verteiltes System entwickelt, das neben einigen komponentenbasierten Systemdiensten einen mehrbenutzer-fähigen

gen CD-Spieler bereitstellt. Dieser steht beispielhaft für eine durch das System verwaltete Ressource, deren Zugriff durch Prioritätswerte des Benutzers bestimmt wird.

Die bei der Entwicklung gesammelten Erfahrungen bezüglich Prozess, Werkzeuge, Sprachen und Komponenten fließen zum Schluss der Arbeit in eine Bewertung ein. Zudem werden noch offene Probleme aufgezeigt und eventuelle Lösungsansätze erarbeitet.

1.4 Aufbau

Die Arbeit besitzt drei Hauptteile: theoretische Grundlagen, Beispielentwicklung und Bewertung.

Kapitel 2 - 4 befassen sich mit den Techniken UML, Komponenten und eingebettet Systeme. Leser, die mit diesen Techniken bereits vertraut sind, können diese Kapitel auch überspringen und später gegebenenfalls bei Unklarheiten von Begrifflichkeiten u.ä. dort nachschlagen.

Kapitel 5 stellt die wichtigsten Eigenschaften eines Entwicklungsprozesses für verteilte Telematikanwendungen heraus. Hier fließen die Ergebnisse der vorangegangenen Kapitel mit ein. Leser, die primär daran interessiert sind, am praktischen Beispiel den Ablauf eines Entwicklungsprozesses nachzuvollziehen, können auch zuerst Kapitel 6 lesen und die dort gegebenen Referenzen auf einzelne Aspekte aus Kapitel 5 bei Bedarf nachschlagen.

Die im theoretischen Teil herausgearbeiteten Grundlagen finden ihre Anwendung in der beispielhaften Entwicklung einer verteilten Anwendung in Kapitel 6.

Eine Bewertung der verwendeten Werkzeuge und Methoden befindet sich im letzten Kapitel der Arbeit.

1.5 Informatik und die deutsche Sprache

Die deutsche Sprache wird aufgrund der immer weiter fortschreitenden globalen Kommunikation, die vor allem im IT-Bereich starken Einfluss hat, immer mehr durch englische Begriffe zurückgedrängt. Viele Versuche diesem Trend durch vollständige Übersetzungen entgegenzuwirken, kranken meist daran, dass wenig Akzeptanz gegenüber immer anderen, nicht standardisierten Begriffen und Formulierungen besteht. Häufig ist es auch schlichtweg unmöglich, bereits verbreitete, de facto standardisierte Begriffe sinnvoll ins Deutsche zu übersetzen, ohne dabei ins Unverständliche oder gar Lächerliche zu verfallen.

In dieser Arbeit wird ein Mittelweg versucht, der dort Übersetzungen anwendet, wo sie Sinn machen und akzeptabel erscheinen. Worte, die im täglichen Sprachgebrauch eines Informatikers für gewöhnlich nicht übersetzt werden, werden jedoch im Englischen belassen, nachdem eine deutsche Definition oder Beschreibung derselben gegeben worden ist. Quelltext und Programmwürfe werden hingegen konsequent englisch gehalten, da sonst ein unangenehmer Mix der beiden Sprachen der Verständlichkeit entgegenwirken würde.

1.6 Danksagung

Ich möchte mich an dieser Stelle besonders bei Martin Simons für die Betreuung und Ratschläge bedanken, die diese Arbeit maßgeblich beeinflusst haben.

Ich danke meiner Familie für die jahrelange Unterstützung meines Studiums, Christine Mayer für Geduld und Ausdauer während der Erstellung der Arbeit, Elko Jacobs und Joachim Barheine für Tips und Anregungen und Martin Burmeister für die interessanten Diskussionen über Komponententechnologien.

1.7 Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit alleine und ohne fremde Hilfe angefertigt habe. Alle verwendeten Hilfsmittel sind dokumentiert.

Guido Laures

2 UML

2.1 Motivation

Bei der Konstruktion einer Brücke existiert ein vordefinierter Prozess, nach dem ein derartiges Vorhaben ablaufen soll. Ein solcher Prozess hat sich anhand jahrhundertelanger Erfahrung mit dem Bau von Brücken entwickelt. Er beschreibt alle wichtigen Schritte von der Analyse der Anforderungen, die die Brücke erfüllen soll, über die Planung des Baus durch Architekten und Bauingenieure, bis hin zur Durchführung des endgültigen Baus der Brücke. Die Kosten und Risiken beim Bau können mit Hilfe von Erfahrungswerten, die bei Anwendung des Verfahrens gesammelt wurden, eingeschätzt und begrenzt werden. Ähnliche Vorgehensweisen findet man in den meisten Ingenieursbereichen von der Elektrotechnik bis hin zum Maschinenbau.

Vergleicht man den oben beschriebenen Sachverhalt mit dem Entwurf und der Implementierung von Software, so muss festgestellt werden, dass ein solches Verfahren in diesem Bereich noch nicht standardisiert wurde. Das liegt sicherlich auch daran, dass die Disziplin des Softwareentwurfs noch sehr jung ist und deshalb auf entsprechend wenig Erfahrungswerte zurückgegriffen werden kann. Die Flexibilität und Unterschiedlichkeit von Software wird oft als Gegenargument zur Verwendung eines vordefinierten Prozesses angeführt.

Seit Bestehen des Forschungsgebietes Informatik wird versucht, Mittel, Sprachen und Werkzeuge zu finden, die das Entwickeln von Software vereinfachen sollen. Es werden Prozesse entwickelt, die eine berechenbarere und damit weniger risikoreiche Entwicklung von Software unterstützen sollen. Diese Prozesse benötigen eine Sprache, die die Modellierung von Software und der damit zusammenhängenden Komponenten verständlich ausdrücken kann. Hier bietet sich die Unified Modelling Language (UML) an.

In diesem Kapitel soll ein Überblick über die Unified Modelling Language (UML) gegeben werden. Nach einem kurzen Abriss über die Kernziele der Sprache folgt ein Überblick über die wichtigsten Konzepte, die anhand der einzelnen Sichten, die die Sprache definiert, erläutert werden. Das Kapitel stützt sich in weiten Teilen auf die Sprachreferenz in [Rumbaugh-99].

2.2 Ziele

Folgende Hauptziele wurden beim Entwurf der UML verfolgt:

- Bereitstellung einer sofort vom Endanwender einsetzbaren, ausdrucksstarken visuellen Modellierungssprache zum Entwickeln und Austauschen von sinnvollen Modellen
- Möglichkeit der Erweiterung und Spezialisierung der Kernkonzepte der Sprache
- Unabhängigkeit von speziellen Programmiersprachen und Entwicklungsprozessen.
- Begünstigung des Wachstums des OO-Marktes.
- Unterstützung von Meta-Konzepten wie Komponenten, Rahmenwerken (Frameworks) und Entwurfsmustern (Design Patterns).

2.3 Sichten

In diesem Kapitel werden die Grundkonzepte der UML vorgestellt. Hierbei sollen insbesondere die verschiedenen Sichten und deren Diagrammtypen diskutiert werden. Eine Sicht vermittelt immer einen Teil des Gesamtsystems mit einer ganz bestimmten Intention. Alle Sichten in ihrer Gesamtheit versuchen eine Be- und Durchleuchtung aller Aspekte des Modells zu erreichen.

Es geht im folgenden vor allem darum, ein Gefühl für die Sprache und ihrer Anwendung in der Praxis zu vermitteln und weniger um die Sprachspezifikation an sich. Zu Gunsten des Verständnisses und der Lesbarkeit wurde an einigen Stellen auf Vollständigkeit verzichtet und sich auf die für diese Arbeit relevanten Konzepte beschränkt. Der weitergehend interessierte Leser sei auf die vollständige Sprachreferenz in [Rumbaugh-99] verwiesen.

2.4 Statische Sicht

Die statische Sicht auf das Modell bildet die Grundlage der UML. Ihre Hauptelemente sind Klassifikatoren und deren Beziehungen zueinander. Es gibt verschiedene Arten von Klassifikatoren, die vom Verhalten der Objekte des Systems, über deren semantischen Hintergrund bis hin zur Spezifikation ihrer Implementation alle Aspekte eines objektorientierten Softwaresystems modellieren. Die Beziehungen dieser Klassifikatoren beschreiben schließlich deren Verzahnungen und Abhängigkeiten untereinander.

2.4.1 Klassifikatoren

Ein Klassifikator ist ein diskretes Konzept des Modells mit definierter Identität, Zustand, Verhalten und Beziehungen. Die wichtigsten Klassifikatoren sind Klassen, Schnittstellen (Interfaces), Akteure (Actors) und Datentypen.

Klassen

Klassen bilden die Grundlage jedes objektorientierten Systems. Eine Klasse beschreibt eine Menge von Objekten von gleicher Struktur, Verhalten und Beziehungen. Alle Attribute und Operationen, die ein System besitzt, werden in Klassen spezifiziert. Jedes Objekt gehört einer bestimmten Klasse an. Es bildet somit eine diskrete Instanz mit eigener Identität, Zustand und Verhalten.

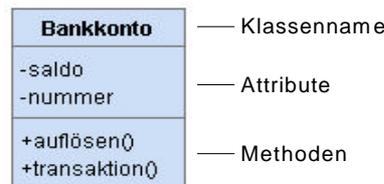


Abbildung 2.1: Klassennotation

Der Zustand eines Objektes einer Klasse ist durch die Belegung seiner Attribute bestimmt. Für gewöhnlich sind Attribute einer Klasse einfache Datentypen ohne Identität und Verhalten. Verbindungen zwischen einzelnen Objekten werden mit Assoziation beschrieben. Eine Änderung im Zustand des Objektes erfolgt immer über Operationen, die Manipulationen auf den Attributen des Objektes ausführen.

Eine Klasse hat innerhalb ihres Containers, der üblicherweise ein Package (siehe 2.10.1) ist, einen eindeutigen Namen. Ebenso besitzt sie eine Sichtbarkeit, die beschreibt, wie und ob sie von anderen Klassen außerhalb des Containers verwendet werden kann.

Abbildung 2.1 zeigt die Beispielklasse `Bankkonto` in der Darstellung des in dieser Arbeit verwendeten Werkzeugs *MagicDraw*. (siehe 7.5.5). Die Sichtbarkeit von Attributen und Operationen werden durch ein vorangestelltes Zeichen beschrieben. Hierbei bedeutet ein Minus *private*, ein Plus *public* und ein Doppelkreuz *protected*.

Schnittstellen (Interfaces)

Ein Interface beschreibt mit der Menge der zu ihr gehörenden Operationen ein externes Verhalten. Es kann von einer Klasse realisiert werden, was bedeutet, dass diese alle Operationen des Interfaces implementiert und somit ihr Verhalten bezüglich des Interfaces spezifiziert.

Datentypen

Ein Datentyp ist ein primitiver Wert, der keine Identität besitzt. Beispiele hierfür sind Nummern, Zeichenketten oder Aufzählungstypen. Sie werden in den meisten Fällen als Wert und nicht als Objekt behandelt und besitzen keine Attribute. Operationen eines Datentyps dürfen keine Manipulationen an Daten vornehmen, sondern dienen lediglich als Datenvermittler.

2.4.2 Beziehungen

Beziehungen, die zwischen Klassifikatoren bestehen können, sind Assoziation, Generalisierung und weitere Arten von Abhängigkeiten.

Assoziation

Assoziationen beschreiben Verbindungen zwischen Objekten des Systems. Eine Assoziation repräsentiert eine geordnete Liste von Klassifikatoren. Der am häufigsten verwendete Typ von Assoziationen ist der zwischen genau zwei Objekten. Ohne Assoziationen gäbe es nur einzelne Klassifikatoren, die nicht miteinander kommunizieren können.

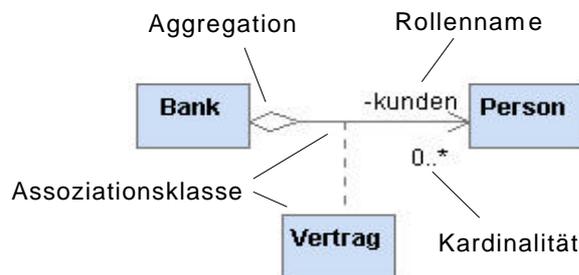


Abbildung 2.2: Beispiel Assoziation

Eine Assoziation kann durch Attribute, Kardinalitäten, Richtungen und Qualifizierer näher beschrieben werden. Abbildung 2.2 zeigt ein Beispiel für verschiedene Eigenschaften einer Assoziation.

Eine Aggregation ist eine Assoziation, die eine Teil-Ganzes-Beziehung repräsentiert. In einem Diagramm wird eine solche Beziehung durch einen leeren Diamanten am Ende des Pfades zur aggregierenden Klasse dargestellt. Ein gutes Beispiel für Aggregationen sind Containerklassen, die nur eine lose Sammlung von Objekten darstellen.

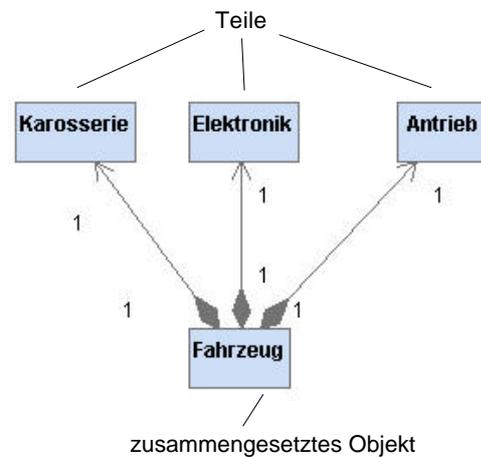


Abbildung 2.3: Komposition

Eine Komposition ist eine strengere Form der Aggregation, die durch einen ausgefüllten Diamanten dargestellt wird und anzeigt, dass die in ihr assoziierten Objekte repräsentativ für ein großes Objekt stehen und deshalb nur innerhalb einer solchen Komposition bestehen können. Deshalb sind die zu einer Komposition gehörende Objekte vom komponierten existenzabhängig. Eine Komposition ist in Abbildung 2.3 dargestellt.

Assoziationen können gerichtet sein. Das bedeutet, dass die durch sie dargestellte Beziehung nur in eine bestimmte Richtung navigiert werden kann. Man sollte hier die Vorstellung vermeiden, eine Assoziation würde mit einem Paar von Zeigern (Pointer), die jeweils auf den assoziierten Partner verweisen, gleichbedeutend sein. An zwei solchen Pointern kann nicht mehr abgelesen werden, ob sie eine bestimmte Beziehung zwischen zwei Objekten repräsentieren oder vielmehr durch andere Wege der Implementation zustande gekommen sind. Eine Assoziation hingegen spiegelt die Designentscheidung wider, dass es Objekte geben kann, die aus einem ganz bestimmten Grund miteinander verknüpft sind.

Generalisierung

Unter einer Generalisierung versteht man die Beziehung zwischen einem generellen und einem – diesen weiter spezialisierenden – Klassifikator. Dieser Mechanismus dient der Erweiterung einer bereits bestehenden Spezifikation um neue oder genauere Eigenschaften. Die speziellere Fassung eines Modellelements muss kompatibel zur allgemeinen sein. Das bedeutet, dass alles, was für Eigenschaften, Operationen und Regeln des generalisierenden Klassifikators gilt, auch für den spezialisierenden übernommen werden muss.

In einem Diagramm wird eine Generalisierung durch einen Pfeil mit nicht ausgefülltem Kopf dargestellt, der vom speziellen zum generellen Klassifikator zeigt.

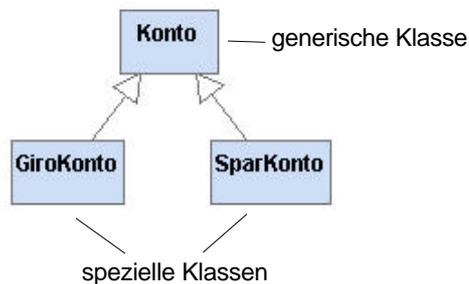


Abbildung 2.4: Generalisierung

Eine Generalisierung bewirkt, dass ein Objekt in verschiedenen Kontexten verwendet werden kann. So kann beispielsweise sowohl ein Girokonto als auch ein Sparkonto überall dort verwendet werden, wo ein Konto benötigt wird. Die Verwendung einer bestimmten Operation eines in einer Generalisierungshierarchie befindlichen Objekts hängt also vom Objekt und seiner ihr eigenen Version der gewünschten Operation ab. Operationen, die nicht implementiert sind nennt man abstrakt. Eine konkrete Klasse darf in ihrer Klassenhierarchie keine abstrakte Operation besitzen.

Realisierung

Eine Realisierung ist eine Beziehung zwischen einem Modellelement, z.B. einer Klasse, und einem anderen Element, das lediglich sein Verhalten, nicht aber seine innere Struktur oder Implementierung spezifiziert, z.B. einem Interface. Es ist auch möglich andere Elemente durch sie zu verbinden. So können verschiedene Versionen einer Implementation ein gleiches Verhalten aufweisen und somit mittels Realisierung modelliert werden. Eine Realisierung wird aufgrund ihrer Ähnlichkeit zur Generalisierung genau so wie diese dargestellt, nur dass der Pfeil gestrichelt wird.

Der Unterschied zwischen Generalisierung und Realisierung ist, dass bei einer Generalisierung immer zwei Elemente des selben semantischen Levels miteinander verbunden werden müssen, meistens sogar innerhalb des selben Modells. Bei der Realisierung können auch Elemente verschiedenster Modelle miteinander in Beziehung treten, die lediglich ein und das selbe, durch einem Klassifikator spezifiziertes Verhalten aufweisen.

2.4.3 Abhängigkeiten

Abhängigkeiten können in einem Modell zwischen zwei oder mehr Elementen bestehen und signalisieren eine semantische Abhängigkeit der sogenannten Clients von deren Server. Findet im Serverelement einer Abhängigkeitsbeziehung eine Veränderung statt, so signalisiert die im Modell definierte Art der Abhängigkeit, ob und welche dieser Modifikationen Auswirkungen auf Clients haben können. Assoziation, Generalisierung und Realisierung sind also spezielle

Formen von Abhängigkeiten. Alle anderen Arten werden in einem Diagramm mit einem gestrichelten Pfeil mit offenem Kopf vom Client- zum Serverelement hin gekennzeichnet, der mit einem, die Art der Abhängigkeit beschreibenden Schlüsselwort versehen wird. Beispiele hierfür sind Zugriffsbeschränkungen, Freundschaftsbeziehungen oder Verwendungsabhängigkeiten.

2.5 Use Case Sicht

In der Use Case Sicht wird das Verhalten des zu modellierenden Systems betrachtet, wie es sich dem Benutzer darstellt. Alle abgeschlossenen Interaktionsschritte, die ein Benutzer mit dem System eingehen kann, werden identifiziert und in einem oder mehreren Use Cases dargestellt. Ein Use Case beinhaltet für das System selbst eine Reihe von Nachrichten, Ereignissen und Operationsaufrufen. Diese sind für den Akteur, der der Initiator des Use Cases ist, weitestgehend unsichtbar. Ein solcher Akteur kann eine natürliche Person, das System oder eines seiner Subsysteme oder eine andere Komponente sein. In der Use Case Sicht wird also die eigentliche Funktionalität, die das System später bieten soll, dargestellt.

2.5.1 Akteur

Ein Akteur ist eine mit dem System in Interaktion tretende Einheit. Er kann mit einem oder mehreren Use Cases in Verbindung stehen. Die Interaktion zwischen Akteur und Use Case findet über den Austausch von Nachrichten statt.

Ein Akteur wird mit einem beschriftetem Strichmännchen-Symbol dargestellt. Die Verbindungen zu Use Cases kennzeichnet man mit einer Linie.

2.5.2 Use Case

Ein Use Case wird dazu verwendet, einen Teil des Systemverhaltens zu beschreiben, ohne auf dessen innere Struktur einzugehen. Er vertritt alle Variationen, die bei der Verwendung der durch ihn beschriebenen Funktionalität auftreten können. Dazu zählen auch Sonderfälle wie die Verletzung von notwendigen Voraussetzungen oder Fehlern.

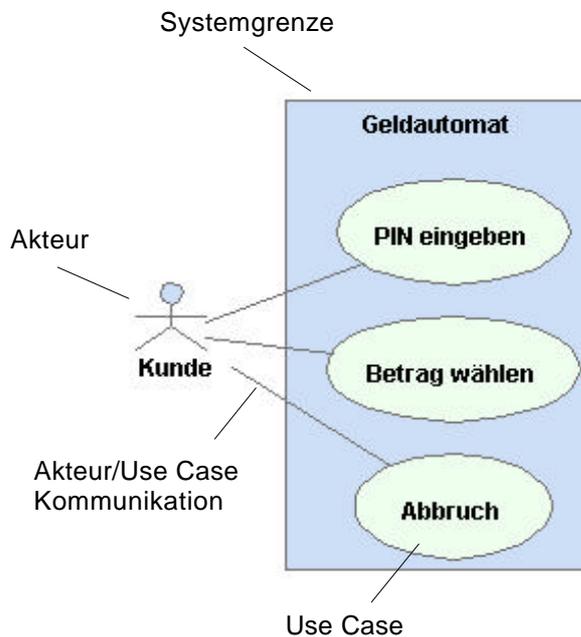


Abbildung 2.5: Use Case Diagramm

Da Use Cases Klassifikatoren sind, können sie mit den gleichen Assoziationen verknüpft werden, wie Klassen.

2.6 State Machine Sicht

In der State Machine (Zustandsmaschine) Sicht wird das dynamische Verhalten von Objekten beschrieben. Dabei wird jedes Objekt als isolierte Einheit betrachtet, die auf Ereignisse von außen reagiert. Der Zustand eines Objekts ist durch die Summe der Zustände seiner Attribute gegeben. Dieser Zustand bestimmt ausschließlich die Art und Weise wie es auf bestimmte Ereignisse reagiert.

Für gewöhnlich wird in einer State Machine das Verhalten von Klassen dargestellt. Es ist jedoch auch denkbar mit ihnen das Verhalten von Use Cases, Kollaborationen oder Methoden zu spezifizieren.

2.6.1 State Machine

In einem State Machine Diagramm wird mittels eines Graphen das Verhalten eines Objekts von seiner Konstruktion bis zu seiner Destruktion vollständig dargestellt. Die Knoten des Graphen sind die Zustände des Objekts, ein Pfad stellt einen Zustandsübergang dar. Jegliche Einflüsse, die die außenwelt auf das dargestellte Objekt haben kann, werden mit Ereignissen dargestellt. Mögliche Reaktionen auf einen solches Ereigniss können das Ausführen einer Aktion oder ein Zustandsübergang sein.

State Machines eignen sich besonders gut zur exakten Darstellung der Funktionsweise eines bestimmten, atomaren Teils des Gesamtsystems. Aus diesem Grund werden sie auch sehr gern im Zusammenhang mit Telematik-Systemen verwendet, was in Kapitel 6 deutlich wird.

Ereignis

Ereignisse einer Zustandsmaschine sind punktuell und für verschiedene Teile des Gesamtsystems von Bedeutung. Man kapselt für gewöhnlich ein Ereignis in einen Deskriptor, der dessen individuellen Eigenschaften in Attribute verpackt. Es ist möglich, solche Deskriptoren in einer Hierarchie zu organisieren, die die verschiedensten Ereignis-Typen in Klassen gruppiert.

Zustand

Ein Zustand (State) wird in einem UML-Diagramm mit einem optional beschrifteten Rechteck mit abgerundeten Ecken dargestellt. Es steht stellvertretend für die Zeitspanne, in der sich das betreffende Objekt in einem bestimmten Zustand befindet. Zustände können durch Transitionen miteinander verbunden werden.

Zudem gibt es die Möglichkeit, Zustände zu neuen, umgebenden Zuständen zusammenzusetzen.

Transition

Eine Transition, die einen Zustand verlässt, kennzeichnet eine Reaktion auf ein Ereignis.

Man unterscheidet zwischen externen und internen Transitionen. Bei den häufiger vorkommenden externen Transitionen, wechselt das Objekt seinen Zustand. Sie werden mit einem Pfeil vom alten zum neuen Zustand dargestellt, der mit einem Text versehen sein kann, der die Eigenschaften der Transition näher beschreibt. Bei internen Transitionen reagiert das Objekt auf ein Ereignis, in dem es lediglich Aktionen ausführt, jedoch keine Änderung des Zustandes vollführt. In Abbildung 2.6 sind verschiedene Transitionsarten dargestellt, deren Erläuterung in [Rumbaugh-99] nachzulesen ist.

Interne Transitionen gehören zu einem Ursprungszustand besitzen jedoch keinen Zielzustand. Sie werden ebenso wie Aktionen im Zustand deklariert und werden beim Feuern des auslösenden Ereignis verwendet. Bei einer internen Transition wird zunächst die Austritts-, dann die mit dem Ereignis verknüpfte und zuletzt die Eintrittsaktion aufgerufen. Man könnte sie also auch mit einer gewöhnlichen Transition auf das gleiche Objekt spezifizieren, dabei ginge jedoch die Unabhängigkeit eines Zustands von seinen Transitionen verloren.

2.7 Aktivitätssicht

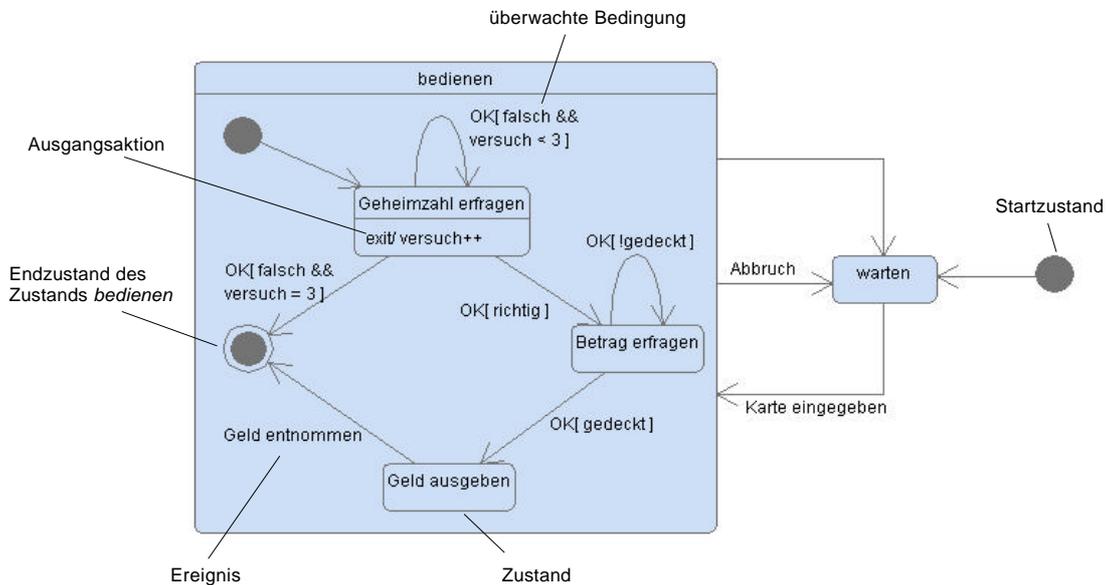


Abbildung 2.6: Zustandsdiagramm eines Geldautomaten

In der Aktivitätssicht arbeitet man mit einer Abwandlung der Zustandsmaschine: dem Aktivitätsgraphen. Im Gegensatz zur Zustandsmaschine repräsentiert dieser keine Objektzustände, sondern vielmehr den Fortschritt einzelner Berechnungsschritte, die auch auf mehrere Objekte verteilt sein können. Für gewöhnlich werden diese nicht durch Ereignisse gesteuert, sondern durch Beendigungen von Berechnungen.

Die Darstellung erfolgt in einem Graphen, der sehr ähnlich zum Zustandsgraphen der Zustandsmaschine ist. Ein solcher Graph ist im Grunde nichts anderes als ein gewöhnliches Flussdiagramm mit dem Unterschied, dass es Wege geben kann, die parallel zueinander ablaufen und an einem Synchronisationspunkt wieder zusammenfinden können.

Da die Aktivitätssicht in dieser Arbeit keine Rolle mehr spielen wird, wird an dieser Stelle von einer genaueren Spezifizierung abgesehen.

2.8 Interaktionssicht

Objekte in einem System müssen miteinander agieren, um ein bestimmtes Verhalten zu implementieren. Das Verhalten einzelner Objekte kann mit Zustandsmaschinen dargestellt werden. Da diese jedoch den Blick für den Kontext des Objektes verlieren und ein Verständnis des Gesamtsystems erschweren, sollte man zur Darstellung des Zusammenspiels mehrerer Objekte eine

Kollaboration wählen. Eine Kollaboration ist eine Menge von Objekten, die durch ihr Zusammenspiel ein Verhalten implementiert.

In der statischen Sicht des Systems werden nur Klassifikatoren und deren zugrundeliegenden Eigenschaften beschrieben. Kollaborationen zeigen diese Klassifikatoren nun in einer Rolle in einem bestimmten Kontext. Kollaborationen stellen sogenannte Slots von Klassifikatoren und Beziehungen zur Verfügung, die stellvertretend für ein Objekt oder Link in einer bestimmten Rolle innerhalb der Kollaboration stehen. Eine Kollaboration bietet somit die Möglichkeit einer umfassenderen Darstellung von Datenstrukturen und Algorithmen gleichermaßen.

2.8.1 Interaktion

Eine Interaktion ist das Versenden von Nachrichten zwischen den Rollen von Klassifikatoren einer Kollaboration über deren Beziehungen. Mit Interaktionen können sowohl ein spezielles Verhalten des Systems als auch bestimmte Use Cases oder Operationen modelliert werden.

Eine Nachricht ist eine Ein-Weg-Information mit optionalen Parametern von einem Sende- zu einem Empfängerobjekt. Man unterscheidet asynchrone Signale und synchrone Operationsaufrufe. Nachrichten werden in einem oder mehreren sequentiellen Kontrollflüssen angeordnet. Diese haben in der UML zwei Darstellungsmöglichkeiten. Sequenzdiagramme legen den Fokus auf den Fluss der Nachrichten im Laufe der Zeit, Kollaborationsdiagramme hingegen auf das Zusammenspiel der Objekte untereinander.

2.8.2 Sequenzdiagramm

Sequenzdiagramme sind zweidimensionale Diagramme, die auf ihrer vertikalen Achse die Zeit und auf der horizontalen die verschiedenen Klassifikatoren-Slots besitzen. Jeder Slot besitzt eine vertikale Linie nach unten, die die Lebensdauer des Objektes, das in dem Slot sitzt, darstellt. Alle Nachrichten, die zwischen Objekten des Diagramms ausgetauscht werden, werden durch einen horizontalen Pfeil von der Lebenslinie des Senderobjektes zu der des Empfängerobjektes dargestellt.

Wird ein Objekt im Laufe der Zeit, die im Diagramm dargestellt wird, zerstört, so endet dort die Lebenslinie mit einem diagonalem Kreuz. Die Lebenslinie eines Objektes ist zur Lebenszeit des Objektes immer dann gestrichelt, wenn es nicht aktiviert ist. Ein aktiviertes Objekt hat immer eine doppelte Lebenslinie. Aktiv ist ein Objekt immer zu der Zeit, in der es eine bestimmte Operation ausführt oder auf die Beendigung einer von ihm angestoßenen Operationen wartet.

Sequenzdiagramme werden gerne zur Darstellung von verschiedenen Szenarios, in denen die gleichen Klassifikatoren mitspielen, verwendet. Man kann an ihnen besser ablesen, wie sich der Nachrichtenfluss von dem eines anderen Szenarios unterscheiden. In Kapitel 6 werden mehrere Sequenzdiagramme verwendet und erklärt, weshalb an dieser Stelle auf weitere Erläuterungen verzichtet wird.

2.8.3 Kollaborationsdiagramm

Ein Kollaborationsdiagramm ist nichts anderes als ein Klassendiagramm, das die Klassifikatoren und Assoziationen in bestimmten Rollen darstellt, in die Objekte zur Laufzeit geraten können. Die Beziehungen in einem Kollaborationsdiagramm können nicht nur durch statische Assoziationen zwischen Klassifikatoren, sondern auch temporär durch Parameterübergabe in einer Nachrichten entstehen. Man stellt in einem Kollaborationsdiagramm immer nur die Klassifikatorenrollen dar, die in der darzustellenden Prozedur relevant sind.

Nachrichten, die zwischen Objekten versandt werden, werden über der Assoziation durch einen wie im Sequenzdiagramm gerichteten beschrifteten Pfeil dargestellt. Dieser kann optional mit einer Nummer versehen werden, die die Position der Nachricht innerhalb der Sequenz repräsentiert. Zudem kann eine Nachricht Parameter und einen Rückgabewert besitzen.

Kollaborationsdiagramme dienen vor allem dazu, Beziehungen von Objekten und detaillierte Prozessimplementationen darzustellen. Sie sind semantisch äquivalent zu Sequenzdiagrammen.

2.9 Physikalische Sicht

In der physikalischen Sicht werden die Implementationsaspekte des Systems besprochen. Diese sind wichtig, um das System auf Wiederverwendbarkeit und Performanz zu untersuchen. Es gibt zwei Diagrammtypen, die zur physikalischen Sicht gehören. Die Implementationssicht legt den Schwerpunkt auf Komponenten (siehe Kapitel 3), die Deployment Sicht beschäftigt sich mit der physikalischen Verteilung des Systems auf verschiedene Rechner und Subsysteme und deren Verbindungen zueinander.

2.9.1 Komponente

Da Komponenten in Kapitel 3 noch ausgiebigst besprochen werden, wird hier nur ihre Darstellung in UML-Diagrammen besprochen.

Eine Komponente wird mittels eines beschrifteten Rechteck dargestellt, an dem an der Seite zwei weitere kleine Rechtecke angebracht sind. Schnittstellen, die die Komponente unterstützt, werden durch kleine beschriftete Kreise dargestellt, die durch eine Linie mit der Komponente verbunden sind. Abhängigkeiten zwischen Komponenten oder zwischen Komponenten und Schnittstellen anderer Komponenten werden durch einen gestrichelten Pfeil vom abhängigen Teil weg gekennzeichnet. Beispiele für Komponenten finden sich in Abbildung 6.12 und Abbildung 6.13.

2.9.2 Knoten

Ein Knoten ist ein Objekt, das eine Ressource im System repräsentiert. Dies kann z.B. ein Rechner, eine Datenbank oder auch ein Platz im Hauptspeicher sein, was man im Diagramm mit einem Stereotypen kennzeichnen kann. Knoten selbst können wieder Objekte und Komponenten beinhalten.

Die Darstellung eines Knotens in einem UML Diagramm erfolgt durch einen optional beschrifteten stilisierten Würfel. Verbindungen zwischen Knoten, die die Kommunikationswege darstellen werden durch eine durchgezogene Linie repräsentiert.

Komponenten und Objekte, die innerhalb eines Knotens liegen, können durch eine Einbettung in den Würfel dargestellt werden.

2.10 Modellmanagement Sicht

Jedes System muss in kleinere Teile zerlegt werden, damit erstens der Überblick gewahrt bleibt und zweitens eine Aufteilung in möglichst unabhängig zu bearbeitende Teile ermöglicht wird. Die UML bietet dafür das Konzept Package (Paket) an.

2.10.1 Package

Alle Elemente eines Modells müssen zu genau einem Package gehören. Die Aufteilung in Packages sollte einem nachvollziehbarem Prinzip folgen, um die Wartbarkeit und Verständlichkeit des Gesamtmodells zu unterstützen. Die UML selbst schreibt keine Regel dafür vor.

Ein Package ist der Container für eine beliebige Anzahl von Elementen, wie z.B. Klassen, Use Cases oder Zustandsmaschinen. Ein Package wird in einem UML-Diagramm durch ein Rechteck mit darauf gesetzter Lasche, ähnlich einem Ordner, dargestellt (siehe Abbildung 6.2).

2.10.2 Abhängigkeiten und Sichtbarkeit

Abhängigkeiten zwischen Packages werden durch einen gestrichelten Pfeil dargestellt. Ein solcher symbolisiert lediglich die Existenz einer Abhängigkeit mindestens eines Elementes des einen Packages von einem Element des anderen. Viele solche Abhängigkeiten werden zu einem Pfeil zusammengefasst. Falls die Abhängigkeiten verschiedene Stereotypen besitzen, kann man die Beschriftung zu Gunsten der Übersichtlichkeit weglassen.

Ein Modell ist ein Package, das eine komplette Definition einer bestimmten Sicht auf das Gesamtsystem enthält. Ein Subsystem ist ein Package mit separaten Spezifikations- und Realisationsteilen. Es repräsentiert eine funktionale Einheit des Gesamtsystems mit wohldefinierten Schnittstellen.

Die hier erarbeiteten theoretischen Grundlagen finden sich in ihrer praktischen Verwendung in Kapitel 6 wieder.

3 Komponenten

3.1 Motivation

Definition: Softwarekomponenten sind binäre Einheiten von unabhängiger Entwicklung, Akquisition und Verbreitung, die miteinander agieren, um ein funktionierendes System zu bilden [Szyperski-97]. Die Technik zur Realisierung des Zusammenspiels einzelner Komponenten nennt man Komponententechnik.

Strapaziert man noch einmal das Beispiel des Brückenbaus, so lässt sich feststellen, dass in den seltensten Fällen alle Teile einer zu erstellenden Brücke neu entworfen und erstellt werden. Vielmehr werden bereits bei anderen Bauwerken verwendete und erprobte Teile zu einem neuen Ganzen miteinander verbunden. So existieren sicherlich fertige Straßenplattensegmente, Brückenpfeiler, Träger, Geländerstücke, Leitplanken und Bordsteine, die nur noch zusammengefügt werden müssen. Einzig neu gestaltet werden müssen also die Teile, die Besonderheiten aufweisen, die bei keiner anderen Brücke in dieser Form vorhanden sind (eigenartige Streckenführung mit Kurven, besondere Statik). Die Planung beschränkt sich dann auf den Entwurf dieser Teile und der Spezifizierung, wie alle Bauelemente letztendlich zusammengefügt werden.

Einen ähnlichen Sachverhalt findet man in der Softwareentwicklung wieder. Die schon fertigen Softwarebausteine, die zur Erstellung eines neuen Produktes verwendet werden, nennt man Komponenten. Durch den sogenannten Glue-Code werden Komponenten miteinander zu einer neuen Einheit verschmolzen. Wie es zu dieser Entwicklung kam und welche Techniken zur Verwendung von Komponenten bereitstehen, beschreibt dieses Kapitel.

3.2 Historie

Nicht weniger als 30 Jahre dauerte es bis die Vision von Doug McIlroy aus dem Jahre 1968 von einer Massenproduktion von Softwarekomponenten Ende der neunziger Jahre wahr wurde [McIl-68]. Letztendlich verhalf das Zusammenspiel von Technologie und Marktstrategien den Komponenten zu ihrer lange vorhergesagten Rolle als *die* treibende Kraft in der Softwareentwicklung.

Ohne die Technik des objektorientierten Softwareentwurfs wäre eine Entwicklung zur Komponententechnologie nicht möglich gewesen. Erst die dadurch gewonnene Eigenständigkeit von Code-Segmenten konnte zur weiterführenden Eigenständigkeit von Programm-Segmenten führen. Erste Versuche, Komponen-

3 Komponenten

tentechniken einzusetzen, wurden in den Labors von XeroX PARC und NeXT in den späten Achtzigern unternommen. Hier wurde der Grundstein für Microsofts VBX (Visual Basic Components) gelegt, das dann in den Weiterführungen OLE und COM den Durchbruch der Komponententechnik auch im kommerziellen Bereich mit sich brachte. Vor allem die exponentiell steigende Verbreitung von Netzwerktechnologien, begünstigt durch das Internet, führte zu einer Schwerpunktverlagerung auf die Verteilung von Komponenten, die über Rechengrenzen hinweg miteinander kommunizieren können. Resultat dieser Entwicklung sind der vom OMG-Konsortium 1995 verabschiedete Standard für verteilte Komponententechnik CORBA (Common Object Request Broker Architecture), Sun's Java Beans Technik und Microsoft's DCOM.

Mittlerweile findet man Komponenten überall in der IT-Welt. Ein Beispiel sind die sogenannten Plug-ins für Programme wie Browser (Netscape, IE), Entwicklungsumgebungen (Forté for Java) oder auch Betriebssysteme (MacOS). Ganze Firmen haben sich auf die Entwicklung von Komponenten spezialisiert, die teilweise in immensen Bibliotheken vertrieben werden.

3.3 Schnittstellen

Jede Komponente ist bestimmt durch die Schnittstellen, mit deren Hilfe man auf ihre Funktionalitäten zugreifen kann.

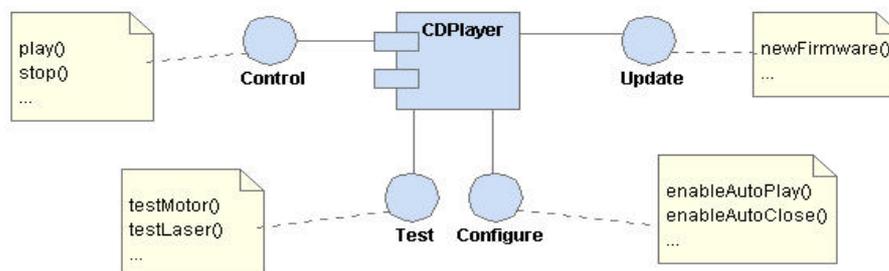


Abbildung 3.1: Beispielkomponente CD-Spieler

Abbildung 3.1 zeigt eine CD-Spieler-Komponente. Diese stellt verschiedene Schnittstellen zur Verfügung, die durch Kreise gekennzeichnet sind. Wichtig hierbei ist, dass Schnittstellen immer ausreichend dokumentiert sein müssen.

Abbildung 3.2 zeigt die Update-Schnittstelle als Beispiel für eine noch nicht



Abbildung 3.2: unzulängliche Schnittstellenspezifikation

ausreichend dokumentierte Zugriffsmöglichkeit auf den CD-Spieler. Hier zeigt sich, dass eine Liste von Methoden nicht ausreicht, um ein Verhalten zu spezifizieren. Anhand der Methodenamen könnte man ein bestimmtes Verhalten erahnen und dieses einfach ausprobieren. Bei der dargestellten Update-Schnittstelle droht jedoch bei falscher Verwendung im schlimmsten Fall die Beschädigung des Gerätes.

Aus diesem Grund ist es immer notwendig, genau zu beschreiben, was eine Operation wirklich bewirkt soll und welche Vor- und Nachbedingungen für ihre fehlerfreie Ausführung gelten müssen. Zudem sollte die Schnittstellen-Spezifikation einer Komponente so präzise sein, dass sie verwendet werden kann, ohne dass man dabei *in sie hineinsehen* muss.

Beim Entwurf einer Schnittstelle muss der Entwickler immer beachten, dass diese so allgemein verwendbar wie möglich gehalten sind. Dazu gehört nicht nur eine Entkopplung der Komponenten untereinander, sondern auch die Verwendung von allgemeinen Datentypen als Parameter und Rückgabewerte.

Eine Schnittstelle, die neben den oben angegebenen funktionalen auch noch qualitative Spezifikationen bereithält nennt man Vertrag (Design By Contract [Meyer-90]).

3.4 Vorteile

3.4.1 Wiederverwendung

Es gibt zwei Extremvarianten, auf welche Art und Weise man eine Softwareunterstützung für bestimmte Tätigkeiten erreichen kann. Die eine ist das Entwickeln eines völlig neuen Softwaresystems „from scratch“, die andere ist der Einsatz einer bereits bestehenden Standardlösung.

Die Vorteile eines völlig neuen Produkts sind offensichtlich. Man hat als Entwickler die Möglichkeit, auf alle Wünsche des Kunden explizit einzugehen und sie in das System zu integrieren. Es entsteht also ein maßgeschneidertes, voll den Bedürfnissen angepasstes System. Der große Nachteil liegt hier jedoch in den hohen Entwicklungskosten und der möglicherweise eingeschränkten Interoperabilität mit anderen Systemen. Hinzu kommt noch, dass durch die meist lange

Entwicklungszeit die entwickelte Lösung aufgrund von technischen Neuerungen schon wieder veraltet oder unnütz ist. Bei der völligen Neuentwicklung muss man in den meisten Fällen davon ausgehen, dass sie in vielen Bereichen nur suboptimal ist, da entsprechende Experten für diese Spezialgebiete bei der Entwicklung nicht verfügbar waren.

Bei Standardsoftware verhält es sich so, dass der Benutzer sich und seine Arbeitsabläufe der Software anpassen muss und nicht umgekehrt. Dies kann zu erheblichen Kosten führen oder im schlimmsten Fall schlichtweg unmöglich sein.

Der goldene Mittelweg zwischen diesen beiden Extremen sind Komponenten. Bei der Verwendung von Komponenten greift man genau dort auf bestehende Lösungen zurück, wo es benötigt wird. Alles andere ist selbst entwickelt und kann so auf die Kundenwünsche zugeschnitten werden.

Beispiel. Eine Firma möchte ein Produkt erstellen, das ein Retrieval auf einer Datenbank ausübt, in der Dokumente in verschiedenen Grafikformaten vorliegen. Diese Dokumente haben eine ganz spezifische Attributierung innerhalb der Datenbank, über die sie gesucht werden können. Das zu entwickelnde System soll den Datentransfer zwischen Clients und Datenbankserver verschlüsseln und dem Benutzer über eine Eingabemaske die Möglichkeit geben bequem nach Dokumenten zu suchen und diese am Bildschirm anzuschauen.

Bei der Entwicklung eines solchen Systems bietet sich der Einsatz von Komponenten für die Verschlüsselung und zum Betrachten der Dokumente an. Das einzig wirklich spezifische sind die Eingabemasken, die selbst entwickelt werden, wozu wiederum auf grafische Komponenten zurückgegriffen werden kann.

In diesem Beispiel wird klar, dass immer genau abgewogen werden muss, wann sich der Kauf einer bestehenden Komponentenlösung rechnet und wann man lieber auf eigene Entwicklung setzen sollte. Die Flexibilität eines Systems sinkt mit dem Anteil an bestehender Komponenten bei gleichzeitiger Verbesserung der Kosteneffizienz.

Die Möglichkeit von Wiederverwendung birgt große Vorteile in sich. So steigt natürlich die Zuverlässigkeit und Korrektheit eines Programms, wenn bewährte und geprüfte Komponenten verwendet werden. Bei Eigenentwicklungen liegt eine große Gefahr in der kaum zu vermeidenden Fehlerträchtigkeit.

Die Philosophie der Wiederverwendung von Komponenten unterscheidet sich von der in der klassischen OOP (objektorientierte Programmierung). Komponenten werden mittels Komposition, Klassen mittels Adaption wiederverwendet.

3.4.2 Versionierung

Ein großer Vorteil beim Einsatz von Komponenten ist der wesentlich vereinfachte Upgrading-Mechanismus, den diese Technik mit sich bringt. Versionsänderungen können nun auf Teilen des Gesamtprogramms ausgeführt werden, der Rest ist davon nicht betroffen. Ein großes Softwareprodukt kann auf diese Art und Weise mit den Neuerungen auf technischer Ebene besser Schritt halten und verwandelt sich so in ein evolutionäres System.

Der Austausch von bestehenden Komponenten durch neuere ist unproblematisch, solange sich die Schnittstellen der Komponente nicht ändern. Es kann dann auf diese immer noch genauso zugegriffen werden, wie zuvor. Für Clients dieser Komponente ist der Austausch dann vollkommen transparent.

3.4.3 Verteilung

Eine Verteilung von verschiedenen miteinander kommunizierenden Komponenten über Rechnergrenzen hinweg ist sehr gut möglich. Das liegt vor allem daran, dass Komponenten i.A. so entworfen werden, dass sie unabhängig von anderen Komponenten arbeiten können.

RMI

RMI (Remote Method Invocation) ist die Kommunikationsplattform von Java-Komponenten. Sie wird an dieser Stelle aufgrund ihres Gebrauchs in späteren Kapiteln kurz erläutert. Mittels RMI ist es möglich, Schnittstellen entfernter Komponenten anzufordern und zu benutzen, als ob sie lokal verfügbar seien.

Eine Komponente, die Schnittstellen remote zur Verfügung stellen möchte, kann dies mittels Anmeldung an einer zentralen Verwaltungseinheit – der sogenannten RMI-Registry – tun. Diese kümmert sich ab dann um die Vergabe von Referenzen auf bei ihr gemeldete Schnittstellenimplementationen. Um eine solche Referenz zu erhalten, muss ein Client eine Anfrage bei der Registry starten. Eine Besonderheit von RMI ist, dass eine verteilte Garbage Collection durchgeführt wird. Ein Client muss sich also nicht um die Freigabe von Ressourcen kümmern, die von entfernten Komponenten besetzt werden.

Abbildung 3.3 zeigt die Verwendung der Registry in einem Sequenzdiagramm: Nachdem der `CDPlayer` sein `Control`-Interface bei der Registry angemeldet hat, hat ein Client die Möglichkeit den `CDPlayer` über diese Schnittstelle anzusprechen. Die notwendige Referenz erhält er bei der Registry auf Anfrage. Die drei in dieser Interaktion beteiligten Objekte können alle auf verschiedenen Rechnern in verschiedenen Prozessen laufen. Die Kommunikation findet dann vollständig über das Netzwerk statt.

3 Komponenten

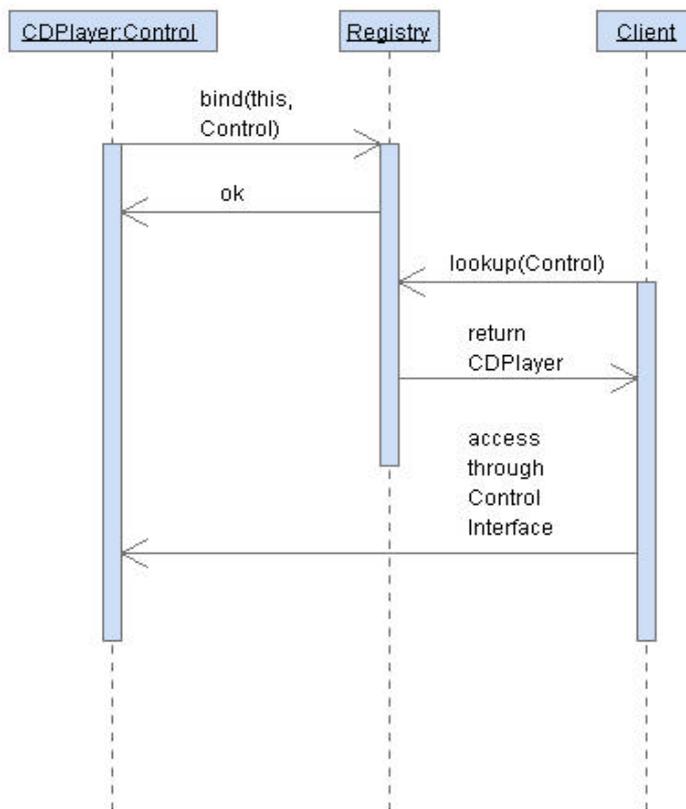


Abbildung 3.3: Funktionsweise der RMI-Registry

Der an Komponententechnologie weitergehend interessierte Leser sei an dieser Stelle auf die angegebene Literatur und insbesondere auf [Szyperski-97] verwiesen.

4 Eingebettete Systeme

4.1 Motivation

Die Inkonsistenz der Begrifflichkeiten rund um den Themenbereich der eingebetteten Systeme macht dieses Kapitel notwendig. Es soll einen kleinen Überblick über den Themenkomplex nebst wichtigen Begriffsdefinitionen liefern. Dies ermöglicht es, in späteren Kapiteln ohne weitere Erläuterungen mit diesen Begriffen zu arbeiten. Da der Themenbereich sehr weitläufig ist, beschränkt sich dieses Kapitel lediglich auf die allgemeinen und in dieser Arbeit weiterverwendeten Bereiche.

Zunächst werden die wichtigsten Eigenschaften eines eingebetteten Systems zusammengestellt. Die Regeln und Einschränkungen, die ein Software-Entwickler bei einem solchen System beachten muss, werden danach anhand von Beispielen aus der Automobilindustrie erläutert.

4.2 Hardware

Jedes eingebettete System besitzt einen Mikroprozessor. Es gibt eine Reihe unterschiedlichster Prozessoren. Die Vielzahl der Prozessorarten rührt vor allem daher, dass ein eingebettetes System in den meisten Fällen für ein ganz spezielles Problem optimiert werden muss und deshalb auch ganz auf dieses Problem hin optimierte Hardware benötigt.

Der Prozessor verfügt über zwei Arten von Speicher. Der eine ist für die Speicherung des abzuarbeitenden Programms zuständig. Dies ist ein permanenter Speicher, der auch beim Ausschalten des Systems nicht gelöscht werden darf. In einem Desktop-System wird dieses Problem dadurch gelöst, dass Programme auf Festplatten oder andere Speichermedien gesichert werden, um sie von dort beim nächsten Start zu laden. Solche Medien sind im Bereich der eingebetteten Systeme eher die Seltenheit, da sie teuer, langsam und groß sind. Der Datenspeicher ist häufig ein nur lesbarer Speicher (ROM), was ihn zur Speicherung von dynamischen Daten, die während des Programmablaufs verwendet werden, natürlich unbrauchbar macht. Aus diesem Grund werden in eingebetteten Systemen für Daten andere Speicherarten verwendet als für Programme. Die Daten, mit denen ein solches System arbeitet, müssen nicht unbedingt beim nächsten Systemstart wieder parat sein. Sollte dies doch der Fall sein, können sie unter Umständen mit einer sparsamen Stromquelle auch im ausgeschalteten Zustand versorgt werden.

Eine Uhr in einem Fahrzeug wird beispielsweise auch noch mit Strom versorgt, wenn die Zündung aus ist.

Des Weiteren besticht die Hardware eines eingebetteten Systems eher dadurch, was es *nicht* benötigt. Da es häufig keine Interaktion mit Anwendern benötigt, muss es nicht zwingend Eingabegeräte wie Tastatur oder Maus besitzen. In vielen Fällen ist die Interaktionsmöglichkeit wesentlich eingeschränkter. Die Ausgabe der Funktionen des Systems erfolgt in den seltensten Fällen auf einem Bildschirm, sondern ist auf einfachere visuelle oder akustische Signale beschränkt.

4.3 Beispiele und Eigenheiten

Um den Problembereich etwas zu konkretisieren, werden nun anhand von Beispielen aus dem Automobilbereich die Eigenheiten eines eingebetteten Systems erörtert.

4.3.1 Beispiel: Anti-Blockier-System (ABS)

Ein Anti-Blockier-System bewirkt, dass bei einer starken Verzögerung eines Automobils, die Räder nicht blockieren. Falls die Räder des Wagens blockieren, schaltet das ABS eine Stotterbremsung ein, die die Räder in kurzen Intervallen abbremst und somit den Wagen auch während des Bremsvorgangs noch manövrierfähig hält.

Auch wenn es sich in der Praxis sicherlich anders verhält, gehen wir im folgenden einmal davon aus, dass das ABS ein Teil der Gesamtelektronik des Wagens ist, der über Verbindungsleitungen die Möglichkeit hat, die wirkliche Geschwindigkeit des Wagens und die Umdrehungen eines Rades zu messen und über eine weitere Leitung, eine Stotterbremsung zu starten und zu stoppen.

Welche Eigenschaften muss die Software, die den Ablauf eines solchen System regelt, haben?

Kurze Antwortzeiten

Es ist wichtig, dass das System die Fähigkeit besitzt, sofort auf ein eintretendes Ereignis zu reagieren, auch wenn es gerade vielleicht noch eine andere Berechnung ausführt. Beim Blockieren der Reifen kann es auf Bruchteile von Sekunden ankommen, die darüber entscheiden, ob der Fahrer die Kontrolle über sein Fahrzeug behalten kann oder nicht.

Testfähigkeit

Eingebettete Software muss die Fähigkeit besitzen, in einer Testumgebung einsetzbar zu sein. Anders als bei Desktop Systemen, muss eine sicherheitskritischen Software, wie die eines ABS, vor dem Einsatz in der eigentlichen Laufzeitumgebung ausgiebigst getestet werden. Es wird überprüft, ob sich das System auch bei ungewöhnlichen Konstellationen und Fehlern der Umgebung, gemäß seiner Spezifikation verhält. Oft ist es schwierig, solche Situationen im Labor nachzustellen, da dort völlig andere Bedingungen herrschen. Hier ist darauf zu

achten, dass die Entwicklungsumgebung und die entwickelte Software selbst Schnittstellen zum Testeinsatz bereitstellt.

Unterstützung der Fehlersuche

Die Fehlersuche ist in eingebetteten Systemen schwierig, da oft keine Möglichkeit besteht, eine Ausgabe in irgendeiner Form (Bildschirm, Akustik) zu generieren. Die Reaktion auf einen Fehler in der Software ist sehr häufig ein vollständiger Ausfall des Systems, dessen Ursachen dann nur sehr schwer nachvollziehbar sind.

In unserem Beispiel würde das ABS bei einem Fehler in der Programmierung vermutlich einfach seinen Dienst einstellen, was dem Entwickler nur wenig Information über die Ursache des Fehlers liefern kann. Deshalb ist bei der Entwicklung eines eingebetteten System darauf zu achten, dass beim Auftreten eines Softwarefehler dessen Herkunft immer so gut wie möglich nachvollziehbar ist und man im besten Fall den Fehler reproduzieren kann.

Robustheit

Wie die meisten eingebetteten Systeme ist bei einem ABS eine absolute Fehlerrobustheit Pflicht. Ein Ausfall des Systems bedeutet eine Gefahr für den Fahrer und ist deshalb nicht akzeptabel.

Bei einem Desktop System gibt es im Fehlerfall die Möglichkeit den Benutzer zu informieren und ihn über die weiteren Schritte entscheiden zu lassen. Da die meisten eingebetteten Systeme keinerlei Möglichkeit der Interaktion mit dem Benutzer haben, müssen sie sich selbständig für die beste Reaktion auf einen Fehler entscheiden können.

Geringer Speicherverbrauch

Es ist für einen Automobilhersteller wohl kaum hinzunehmen, für jeden kleinen Teil der Elektronik eines Fahrzeuges Unmengen von Speicher bereitzustellen. Obwohl Speicher immer preiswerter wird, fallen bei einer Massenproduktion auch kleinere Beträge ins Gewicht. Die Software eines eingebetteten Systems muss so klein wie irgend möglich sein, um möglichst wenig Speicherplatz in Anspruch zu nehmen. Dazu gehört auch, dass sie dynamische Daten, die nicht mehr benutzt werden, erkennt und aus dem Speicher entfernt (Garbage Collection).

4.3.2 Beispiel: Telematik im Fahrzeug

Um der in Kapitel 6 entwickelten Anwendung ein wenig vorzugreifen soll als zweites Beispiel ein Telematik-System im Fahrzeug mit Bildschirm und den verschiedenen Anwendungen Navigationssystem, Entertainment und Kommunikation dienen. Es kann mit Steuerkomponenten des Fahrzeugs kommunizieren und auf Telematik-Daten reagieren.

Effizienz

Beim Entwurf einer eingebetteten Software ist die Komplexität des Programms so gering wie möglich zu halten. Jeder Effizienzverlust resultiert in einem erhöhtem Zeit- und Hardwarebedarf. Um die Antwortzeiten des Systems so kurz wie möglich zu halten, ist eine saubere Programmierung mit ausgereiften Algorithmen unabdingbar. Ein Telematik-System muss beispielsweise Datenströme schnell verarbeiten und verwalten können. Hierzu kann es notwendig sein, ein spezielles Protokoll für die Kommunikation zwischen einzelnen Komponenten zu entwickeln.

Installation

Ein solches Telematik-System muss natürlich um weitere Anwendungen erweiterbar sein. Anders als bei einem Desktop-System bevorzugt der Benutzer hier eine möglichst interaktionsarme Installation und Konfiguration. Alles muss, so weit es möglich ist, eigenständig ablaufen und trotzdem den Bedürfnissen des Anwenders entsprechen.

Geringe Kosten

Nicht nur der Speicherverbrauch, sondern jedes einzelne Bauteil wirkt sich auf den Preis für eine eingebettete Software aus. In einem ABS ist es wohl kaum notwendig einen leistungsstarken 32-bit Prozessor einzubauen, da die Berechnungen, die es anstellen muss wenig komplex sind. Bei einem Telematik-System ist genauestens abzuschätzen, wie leistungsstark die zu Grunde liegende Hardware sein muss, um die gewünschte Funktionalität bereitstellen zu können.

Geringer Energieverbrauch

Es ist wichtig, dass ein eingebettetes System so entwickelt wird, dass es möglichst wenig Energie verbraucht und eine sinnvolle Reaktion auf eine schwächer werdende Stromquelle parat hält. So ist es bei einem Telematik-System wichtig, dass der Energieverbrauch geringer bleibt als die Menge von Energie, die durch die Lichtmaschine nachgeliefert werden kann. Bei einem Abfall der Energiezufuhr muss unter Umständen eine Selbstabschaltung des Systems zu Gunsten des vielleicht von der selben Energiequelle gespeisten ABS erfolgen.

Alle in diesem Kapitel erarbeiteten Eigenschaften muss der Entwickler einer Telematikanwendung während des gesamten Entwicklungsprozess berücksichtigen.

5 Entwicklungsprozess für verteilte Telematik

5.1 Motivation

In den vorangegangenen Kapiteln wurde auf die Themen Komponenten, eingebettete Systeme und UML eingegangen. Es wurde herausgearbeitet, dass Komponententechnik in der heutigen IT-Welt auch aufgrund der immer weiter voranschreitenden Verteilung von Software immer mehr Verbreitung auch im Gebiet der eingebetteten Systeme findet. Die UML bietet die Grundlage dafür, komplexe Softwaresysteme in einer verständlichen und normierten Syntax zu beschreiben.

In diesem Kapitel soll nun ein Prozess skizziert werden, der die Rahmenbedingungen für die Verwendung von UML auf den Gebieten Komponenten und eingebettete Systeme festlegt. Die vollständige Spezifikation eines solchen Prozesses würde den Rahmen dieser Arbeit sprengen, so dass hier lediglich drei bereits bestehende Prozesse betrachtet werden sollen, aus denen dann die interessantesten Eigenheiten extrahiert werden. Zudem werden Hinweise gegeben, warum bestimmte Prozesseigenschaften gerade im Telematik-Umfeld besonders nützlich sind.

Zu den Haupteigenschaften, die der skizzierte Prozess haben soll, gehören unter anderem

- Verwendung von UML, Komponenten und Design Pattern,
- Zentrierung auf die funktionalen Anforderungen an das Produkt,
- Verwendung einer iterativen und inkrementellen Herangehensweise,
- Verwendung der 4+1 Sicht und Schichtenmodellen auf die Softwarearchitektur,
- Modellkonsistenz,
- Einfachheit.

5.2 Was ist ein Software Entwicklungsprozess?

Ein Prozess definiert *wer was wann wie* macht, um ein bestimmtes Ziel zu erreichen [Jacobson-99]. Er beschreibt die Aktivitäten, die ein Entwickler vollziehen muss, um ein Produkt zu erhalten, das den Anforderungen des Kunden genügt. Die Qualität einer Software hängt stark davon ab, wie effizient die Richtlinien für den Entwicklungsprozess waren und wie strikt sie eingehalten wurden.

Ein Entwicklungsprozess sollte so flexibel sein, dass er mit den Veränderungen und Fortschritten in den Bereichen Technik, Tools und Personal mithalten kann. Im Technik-Bereich ergeben sich laufend Neuerungen in den Gebieten Betriebssystem, Netzwerktechnologie, Programmiersprachen und Entwicklungsumgebungen, auf die der Prozess angepasst werden muss. Auch der Ausbildungsstand der Entwickler und deren Verfügbarkeit spielen hier eine nicht zu unterschätzende Rolle. Ein Software Entwicklungsprozess muss auch nach mehreren Jahren, z.B. bei einem Redesign einer Legacy-Software, die bereits nach dem gleichen Prozess entwickelt wurde, noch anwendbar sein, ohne dabei auf Neuerungen in den verschiedensten Bereichen des Engineerings verzichten zu müssen.

5.2.1 Artefakte

Während der Entwicklung einer Software entstehen sogenannte Artefakte, die das eigentlich zu entwickelnde System beschreiben und formen. Zu ihnen zählen nicht nur der Code und ausführbare Programme, sondern unter anderem auch entworfene Modellelemente, Testberichte, Anforderungs- und Marktanalysen.

Man unterscheidet zwei Arten von Artefakte: Ingenieurs- und Managementartefakte. Der in diesem Kapitel skizzierte Prozess wird sich jedoch lediglich mit den Ingenieursartefakten auseinandersetzen, die während der einzelnen Entwicklungsphasen abfallen. Diese bestehen zum größten Teil aus den im Kapitel 2 besprochenen UML-Diagrammen. Mit Hilfe von Festlegungen darüber, welche Artefakte zu welcher Entwicklungsphase einer Telematikanwendung erstellt und geprüft werden müssen, ist es möglich, einen wiederverwendbaren Rahmenprozess zu schaffen.

5.3 Etablierte Prozesse

Im folgenden werden Prozesseigenschaften zusammengestellt, die für die Entwicklung von Telematik-Anwendungen interessant sind. Diese Eigenschaften stammen zum größten Teil aus einen der drei folgenden Entwicklungsprozesse:

Der Unified Software Development Process ist ein sehr weit verbreitetes Rahmenwerk, an dem sich Software Entwicklungsprozesse orientieren können. Der ROPES (Rapid Object-Oriented Process For Embedded Systems) ist für die Verwendung in eingebetteten und Echt-Zeit Systemen entwickelt worden, wohingegen der noch recht junge Catalysis-Ansatz kein eigentlicher Entwicklungsprozess ist, sondern vielmehr Herangehensweisen an Spezifikation und Design komponentenbasierter Systeme definiert.

5.3.1 *Unified Software Development Process (USDP)*

Der Unified Software Process wurde von den Hauptentwicklern der UML entworfen. Er ist ein umfassendes Rahmenwerk nicht nur für den Ingenieursbereich sondern auch den Managementbereich eines Prozesses. Im Rahmen dieser Arbeit ist jedoch lediglich die Ingenieursicht interessant.

Die Haupteigenschaften des USDP sind Use Case Basierung, inkrementelles, iteratives Vorgehen und Architekturzentrierung. Auf alle diese Eigenschaften wird später noch näher eingegangen werden.

5.3.2 *Rapid Object-Oriented Process for Embedded Systems (ROPES)*

ROPES ist ein speziell auf die Entwicklung im Echtzeit- und Embedded-Bereich zugeschnittener Prozess. Der in [Douglass-99-2] ausführlich vorgestellte Prozess betont die Notwendigkeit von schnellen Entwicklungszyklen mit lauffähigen Prototypen. Viel Wert wird auch auf die Möglichkeit einer frühen Konsistenz- und Korrektheitsprüfung gelegt, mit der eine Einschränkung der finanziellen Risiken beim Entwurf von Software erreicht werden soll.

5.3.3 *Catalysis*

Catalysis ist ein recht neuer Ansatz, der erstmalig in [D'Souza-98] im Oktober 1998 ausführlich vorgestellt wurde. Sein Schwerpunkt liegt im Bereich der offenen, verteilten, komponentenbasierten Systeme.

Er verwendet sogenannte Prozessmuster, die eine Art *Kochrezept* für die Vorgehensweise bei bestimmten Problemstellungen bieten sollen. Des Weiteren wird viel Wert auf die Kompatibilität zwischen verschiedenen UML-Modellen und die Möglichkeit der Korrektheitsprüfung von Code bezüglich einer Spezifikation gelegt. Es soll dadurch die Wartung und Qualitätssicherung der Software verbessert werden.

5.4 *Use Case Basierung*

Eine Basis des gesamten Entwicklungsprozess bildet das Use Case Modell. Es besteht aus einer Reihe von Use Case Diagrammen und deren Szenarien, wie sie in 2.5 beschrieben wurden.

Zu Beginn eines jeden Softwareentwurfs steht die Anforderungsanalyse, die beschreibt, welche Funktionen und Qualitätsmerkmale das zu entwickelnde System haben soll. Da diese Analyse am besten in Zusammenarbeit mit dem zukünftigen Benutzer des Systems vollzogen wird, ist es wichtig, dass die Ergebnisse in einer einfach verständlichen Form festgehalten werden. Use Cases sind dazu ein geeignetes Mittel. Ein UseCase-Modell bietet sich deshalb auch als Grundlage für einen Vertrag zwischen Entwicklern und Kunden an.

5.4.1 Use Cases treiben den Prozess voran

Use Cases sind der Start- und Basispunkt für viele Aktivitäten innerhalb des Entwicklungsprozesses.

Sie helfen dem Entwickler beim Auffinden der grundlegendsten Klassen. Beim Lesen eines Use Cases versucht der Entwickler die Klassen zu entwerfen, die eine Realisierung des Use Cases ermöglichen. Die Entwicklung von grafischen Benutzerschnittstellen ist ebenfalls stark von Use Cases beeinflusst, da die Funktionalität, die ein Use Case darstellt, vom Benutzer in Interaktion mit dem System initiiert werden muss. Die Erstellung von Dokumentationen über die Benutzung der Software wird durch die Orientierung an Use Cases wesentlich vereinfacht. Nicht zuletzt werden die meisten Testfälle auf der Basis von Use Cases entworfen. Das System muss im Hinblick darauf geprüft werden, ob es die im Use Case Modell aufgestellten Funktionalitäten bereitstellen kann.

Das Use Case Modell kann auch hilfreich sein, wenn es darum geht, die Performance des Systems zu verbessern. Stellt man Überlegungen dazu an, wie oft ein bestimmter Use Case auftritt, so lässt sich leicht erkennen, bei welchen Funktionalitäten besonderes Augenmerk auf die Komplexität des dahinter verborgenen Algorithmus zu legen ist.

In verteilten Systemen spielt gerade die Kommunikation zur Realisierung von Use Cases eine große Rolle. Häufig werden dazu im Telematikbereich spezielle Protokolle verwendet, deren Komplexität bei der Realisierung von Use Cases beachtet werden muss. Bei der entworfenen Telematikanwendung in Kapitel 6 wird sich das vor allem bei der Ereigniskommunikation zwischen verteilten Komponenten zeigen.

Die Nachvollziehbarkeit des Gesamtentwurfs wird durch Use Cases vorangetrieben. Jeder Entwicklungsstrang ist von einem bestimmten Use Case angestoßen. Ausgehend von diesem können nun die im weiteren Prozess entwickelten Artefakte wie Klassen, Kollaborationen, Szenarien und Testfälle nachverfolgt werden. Alle späteren Iterationen (siehe 5.6) werden daraufhin geprüft, ob sie einen bestimmten Use Case realisieren. Jede Änderungen im Gesamtprojekt muss auf Kompatibilität mit dem Use Case Modell überprüft werden.

5.4.2 Auffinden von Use Cases

Das Auffinden der relevanten Use Cases ist eine der Hauptaufgaben während der Anforderungsanalyse.

Die Akteure lassen sich finden, indem man untersucht, welche Rollen ein Benutzer eines System einnehmen kann. Jede Rolle bildet dann einen Akteur. Dabei können reale Personen mehrere Rollen annehmen. Ein und die selbe Person kann beispielsweise als normaler Benutzer oder als Administrator eines Systems auftreten. Zusätzlich ist zum Auffinden der Akteure wichtig, zu untersuchen, welche anderen Systeme wie mit dem zu entwerfenden System interagieren.

Bei einer Telematik-Anwendung können nicht nur Fahrzeuginsassen, sondern auch bestimmte Komponenten des Systems als Akteure eines Use Cases auftre-

ten. So kann z.B. ein Notfallsystem beim Use Case *Notfall melden* als Akteur *Melder* auftreten.

Man untersucht zum Auffinden von Use Cases das System also auch darauf, wie der Benutzer oder eine Komponente des Systems Funktionen verwendet, um eine Aufgabe zu erfüllen. Die so gefundenen Use Cases werden daraufhin verfeinert, zusammengefasst und geordnet und mit den entsprechenden Akteuren zu einem Diagramm verknüpft.

5.4.3 Auffinden von Klassen anhand von Use Cases

Use Cases helfen dabei, eine iterative und inkrementelle Vorgehensweise im Prozess anzuwenden. Anhand der Use Cases lassen sich bequem die nächsten Schritte einteilen, die der Prozess nehmen soll. Jeder Schritt realisiert einen bestimmten Use Case.

Während der Analyse werden Schritt für Schritt die einzelnen Use Cases analysiert und Klassifikatoren gefunden, die diese realisieren können. Mit jedem weiteren analysierten Use Case wächst das Analysemodell an. Die Realisierung eines Use Cases muss dann mit Hilfe einer Kollaboration beschrieben werden. Ist bereits eine Architekturbeschreibung vorhanden, kann sie dabei hilfreich sein, bestimmte Klassifikatoren wiederzuverwenden oder zu spezialisieren. Jeder Klassifikator spielt bei der Realisierung eines Use Cases eine bestimmte Rolle, die bereits erste Attribute der Klasse bestimmen kann. In dieser Phase des Design werden jedoch nur diese Attribute spezifiziert, die zur Realisierung von Use Cases notwendig sind.

Betrachtet man nun die entwickelten Klassen, lassen sich Verantwortlichkeiten und Beziehungen finden, die wiederum in die Architekturbeschreibung mit einfließen können. Man sieht hier dass Architektur und Use Cases sich gegenseitig beeinflussen. Auf die Entwicklung der Architekturbeschreibung wird in 5.5 noch einmal genauer eingegangen.

5.4.4 Sequenzdiagramme zur Use Case Beschreibung

Nicht triviale Use Cases sollten mit einem Sequenzdiagramm versehen werden. Dieses beschreibt am besten die Kommunikation zwischen System und Akteur. Gerade in Telematik-Anwendungen ist eine detaillierte Kommunikationsbeschreibung unabdingbar, da sie häufig stark von der umgebenden Architektur abhängig ist. Auch hier müssen Use Cases und Architekturbeschreibung konform zueinander bleiben, um die Modellkonsistenz zu wahren.

Die Spezifikation von Sequenzdiagrammen wurde ja bereits in 2.8 gegeben. Hier sollen nun die Besonderheiten bei eingebetteten Systemen herausgestellt werden.

Häufig ist es sinnvoll zusätzlich zur normalen Notation, einige Nachrichten mit Markierungen zu versehen, die Bedingungen und Eigenschaften der Nachricht beschreiben. So bietet es sich beispielsweise an, zwischen aufeinanderfolgenden Nachrichten einen Zeitvermerk anzuordnen, der beschreibt, wieviel Zeit zwischen diesen Nachrichten höchstens verstreichen darf.

Bei verteilten Anwendungen spielt die Synchronisation von Nachrichten und Ereignissen eine besonders große Rolle. Sequenzdiagrammen sind zum Auffinden von Synchronisationsproblemen ein geeignetes Mittel.

5.4.5 Test anhand von Use Cases

Durch einen Test lässt sich überprüfen, ob ein System die anfänglich geforderten Funktionalitäten erfüllen kann.

Ein Test besteht im allgemeinen aus einer Reihe von Eingabeparametern, Bedingungen für die Ausführung und einer Spezifikation der Erwarteten Ergebnisse. Es kann für jeden Use Case überprüft werden, ob die an der Realisierung des Use Cases beteiligten Klassen so miteinander arbeiten, wie es in den zum Use Case gehörenden Interaktionsdiagrammen spezifiziert wurde. Die Tests, die ein negatives Ergebnis liefern werden gesammelt und nach Prioritäten sortiert bearbeitet.

Das besondere an dieser Art des Testens ist, dass schon zur Anforderungsanalyse, also in dem Moment, in dem die ersten Use Cases aufgestellt werden, auch die erforderlichen Tests feststehen. Testfälle lassen sich komfortabel von Use Cases ableiten.

Test können aus verschiedenen Sichten auf das System heraus durchgeführt werden. Es gibt die Möglichkeit des Testens in der Rolle des mit dem System interagierenden Akteur oder auch als Software-Designer zur Überprüfung von Objektkollaborationen.

Eingebettete Systeme benötigen grundsätzlich ein umfangreicheres Testen als „gewöhnliche“ Software. Dies hängt vor allem damit zusammen, dass es hier nicht so einfach möglich ist, ein bereits eingebautes Programm noch einmal von Fehlern zu bereinigen. Häufig werden große Teile der Software auch fest verdrahtet, was diesen Umstand noch verschärft. Bei der Entwicklung einer eingebetteten Telematikanwendung muss deshalb immer auch an ausreichendes Testen während aller Phasen des Prozesses gedacht werden.

5.4.6 Use Case Regeln

Im folgende sollen einige Regeln aufgestellt werden, die zu einer guten Use Case Analyse verhelfen können.

- Es sollte darauf geachtet werden, dass ein Diagramm nie mehr als 15 Use Cases enthält.
- Mehr als eine Generalisierungsebene verkomplizieren das Diagramm unnötig.
- Jedes Diagramm sollte eine ausreichende Beschriftung haben, die einen aussagekräftigen Titel, den Autor und die Version (Datum) des Diagramms enthält. Wie in 7.5 nachzulesen ist, ist dies nicht mit allen UML-Werkzeugen möglich.
- Man sollte aus Gründen der Übersichtlichkeit versuchen, Use Cases möglichst nach Akteuren zu gruppieren.

5.5 Architektur-Zentrierung

Unter Architektur werden in diesem Kapitel Modellelemente verstanden, die die Grundstruktur der Software bestimmen. Solche Elemente kann man daran erkennen, dass Änderungen, die an ihnen vorgenommen, relevante und weitreichende Auswirkungen auf Elemente anderer Packages haben können.

Use Cases sind zur Entwicklung einer Software nicht ausreichend. Jeder Entwickler hat beim Programmieren eine bestimmte Vision von dem Gesamtsystem, die nicht nur von funktionalen Aspekten sondern vielmehr auch von der Software als eine Art Bauwerk ausgeht. Die für die Software-Architektur signifikanten Elemente wie Subsysteme und ihre Abhängigkeiten voneinander, Schnittstellen, Komponenten, Knoten, Threads und Prozesse bilden neben den Use Case-basierten funktionalen Eigenschaften des Systems die zweite Säule auf die sich die Entwicklung stützt.

Eine Beschreibung der Softwarearchitektur für ein Telematik-System durchläuft so wie der gesamte Prozess mehrere Iterationen. Dennoch ist es wichtig bereits früh eine stabile Architekturbeschreibung zu haben, die sich nur noch in Nuancen verändern kann. Nach Einstieg in die Designphase müssen sich alle Entwickler auf eine stabile Architektur verlassen können.

5.5.1 Nutzen der Architekturbeschreibung

Eine Telematik-Anwendung ist ein abstraktes Gebilde, das sich nur sehr schwer verbildlichen lässt. Eine bildhafte Beschreibung, wie sie das Architekturmodell liefern soll, hilft dem Entwickler jedoch dabei, den von ihm entwickelten Teil ins Gesamtsystem einzuordnen und dadurch besser zu verstehen.

Im Telematik-Umfeld ist eine Beschreibung des Ist-Zustandes der Ausführungsumgebung der Software, und dessen Auswirkung auf die weitere Entwicklung unabdingbar. Vorhandene Komponenten und Dienste, die die Umgebung der Software bietet, müssen effizient (wieder-)verwendet und eingebunden werden können.

Es ist wichtig, dass eine für alle in den Entwicklungsprozess involvierten Akteure verständliche Architekturspezifikation existiert. Zu ihnen zählen nicht nur die Entwickler, sondern auch Kunden, Manager und andere Entscheidungsträger, die ihre Tätigkeiten und Bedürfnisse an der Architektur ausrichten.

Gerade im Telematik-Bereich, wo viele Berührungspunkte von Hard- und Software beachtet werden müssen, arbeiten oft Experten aus verschiedensten Bereichen miteinander. Es muss deshalb ein besonderes Augenmerk darauf gelegt werden, dass alle –und nicht nur der Softwarearchitekt– die Architekturbeschreibung verstehen und daran weiterarbeiten können.

Im Zuge der Globalisierung ist eine Aufteilung eines großen Projekts in unabhängige Teilbereiche immer wichtiger geworden. Es geht darum, die Entwicklung unter Umständen auch über Ländergrenzen hinweg zu organisieren. Dies ist jedoch nur dann möglich, wenn die Kommunikation der einzelnen Teams durch eine klare Architekturbeschreibung unterstützt wird. Die durch die Architektur

vorgegebenen Schnittstellen des Systems und deren Semantik sollten die Berührungspunkte der einzelnen Teams sein. Sie erfordern deshalb eine ausgereifte und genaue Beschreibung, die möglichst stabil bleibt und allgemein verständlich ist. Hilfreich sind hier die sogenannten Entwurfsmuster (Design Patterns), die in 5.5.3 noch einmal aufgegriffen werden und in Kapitel 6 ausgiebig zum Einsatz kommen.

Durch klar abgegrenzte Schnittstellen und Vermeidung von Abhängigkeiten einzelner Schichten des Systems untereinander kann erreicht werden, dass sich diese aus dem speziellen Problemkontext der Software herauslösen und anderweitig wiederverwendbar werden. Die Architekturbeschreibung kann dabei helfen, diese Teile aufzuspüren. Die UML ist ein weiterer Schritt in diese Richtung, da sie durch ihre allgemeine Bekanntheit auch einem projektfremden Entwickler ein Verständnis für die Architektur vermitteln kann.

Ein weiterer Vorteil, den eine saubere Architektur mit sich bringt, ist die Vereinfachung der Wartbarkeit und Weiterentwicklung der Software. Ist das System klar in Subsysteme unterteilt, die unabhängig voneinander arbeiten, wirken sich Änderungen immer nur auf dieses Subsystem auf und sind deshalb überschaubarer.

5.5.2 *Einflussfaktoren der Architektur*

Dass sich Use Cases und Architektur gegenseitig beeinflussen, wurde schon im vorigen Kapitel angesprochen. Nun sollen diese und andere Abhängigkeiten näher beleuchtet werden.

Beim Entwurf der Architektur ist darauf zu achten, dass sie von den Use Cases angetrieben wird. Damit ist gemeint, dass ein Architekturdesign immer die Realisierung der bereits aufgestellten Use Cases unterstützen muss. Man geht für gewöhnlich so vor, dass die für die Architektur relevanten Use Cases herausgestellt werden und anhand von ihnen ein erster Architekturentwurf unternommen wird.

Die Architektur unterliegt einem iterativen Entwicklungsprozess. Man sollte Schritt für Schritt versuchen zu einer immer stabileren Architekturbeschreibung zu gelangen. Beim Erstellen lernt der Entwickler auch mit den Begriffen des Problembereichs umzugehen und diesen zu strukturieren. Als Reihenfolge der einzelnen Iterationen bietet sich einer Abarbeitung der in 5.5.3 erläuterten Systemschichten von unten nach oben an.

Ein Inkrement (siehe 5.6.4) sollte immer eine Architektur haben, die einen relevanten Teil von Use Cases realisieren kann.

Wie beeinflusst nun die Architektur das Use Case Modell? Use Cases sind zum größten Teil von den Vorgaben des Kunden beeinflusst. In einigen Fällen bedingt jedoch die Architektur eine Verfeinerung des Use Case Modells.

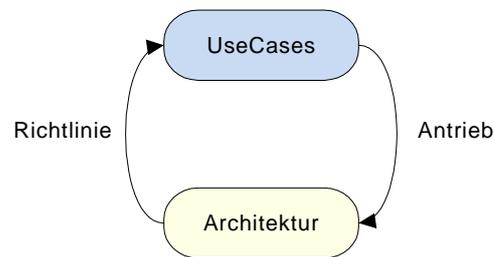


Abbildung 5.1: Gegenseitige Beeinflussung

Es kann beispielsweise Vorkommen, dass ein Änderungswunsch eines Kunden neue Use Cases mit sich bringt. Bei der Definition dieser Use Cases und deren Aufnahme in das Use Case Modell ist es ratsam, die bereits vorhandene Architektur zu verwenden. So läuft man nicht Gefahr, durch eine voreilige Use Case Beschreibung Änderungen im Architekturmodell zu erzwingen. Manchmal ist es sogar ratsam, mit Absprache des Kunden eine Modifikation des Use Cases zu Gunsten einer stabilen Architektur vorzunehmen. Eine typische Frage, die der Entwickler sich stellen sollte, bevor er eine Änderung an der Architektur aufgrund eines neuen Use Cases vornimmt, lautet etwa: Kann eine den Kunden zufriedenstellende, ähnliche Funktionalität auch durch Benutzung der vorhandenen Architektur realisiert werden?

5.5.3 Strukturmuster

Beim Erstellen einer Architekturbeschreibung helfen dem Entwickler sogenannte Muster dabei, nicht bei jedem Problem das Rad neu erfinden zu müssen. Viele strukturelle Probleme, die mit einer zugrundeliegenden Architektur bewältigt werden sollen, wurden schon in ähnlicher Form von anderen Entwicklern gelöst. Viele dieser Muster sind beispielsweise in [Ganssle-91] ausführlich beschrieben und mit einem allgemein gebräuchlichen Namen versehen worden. Es sollte versucht werden, möglichst viele dieser bewährten Techniken ins eigene Projekt zu übernehmen. In Kapitel 6 werden einige dieser Muster verwendet und im Anhang näher beschrieben. Ein Architekturmodell, das Strukturmuster verwendet, sollte diese kurz erklären, da es noch keine allgemein anerkannte Standardisierung der Begriffe und Muster gibt. Observer, Factory und Command sind nur einige Beispiele für die in Kapitel 6 verwendeten Muster.

Ein im Telematik-Bereich oft verwendetes Muster ist die bereits oben erwähnte und in Kapitel 6 verwendete Schichteneinteilung der Architektur. Diese bewirkt eine einfachere Aufteilung des Gesamtprojekts in Gruppen und verringert den Kommunikations- und Abstimmungsbedarf während der Entwicklung.

Häufig vorkommende Schichten sind Applikations-, Dienst- und Systemschicht. Die Systemschicht ist die Schicht, die das Betriebssystem zur Verfügung stellt. Dazu gehört oft eine Abstraktion der Hardware durch Treiber und grundlegende

Funktionalitäten wie Ein-/Ausgabe oder Prozessverwaltung. Die Dienstsicht setzt auf der Systemschicht auf und bereitet den eigentlichen Applikationen eine bequemere Verwendung der Funktionalitäten des Systems durch Kommunikationsmechanismen, Grafikbibliotheken, Ereignissteuerung und Sicherheitsmechanismen. Die Applikationschicht wendet die Daten und Funktionen der Dienstsicht auf die spezifische Applikationslogik an und die Repräsentationsschicht dient zur Interaktion mit dem Anwender.

5.5.4 Die 4+1 Sicht

Beim Entwurf der Architektur kann sich der Entwickler auch an der sogenannten 4+1 Sicht orientieren, die im folgenden kurz vorgestellt wird. Sie wird in [Quartani-98] beschrieben. Abbildung 5.2 zeigt eine bildliche Darstellung dieser Betrachtungsweise auf die Architektur eines Softwaresystems.

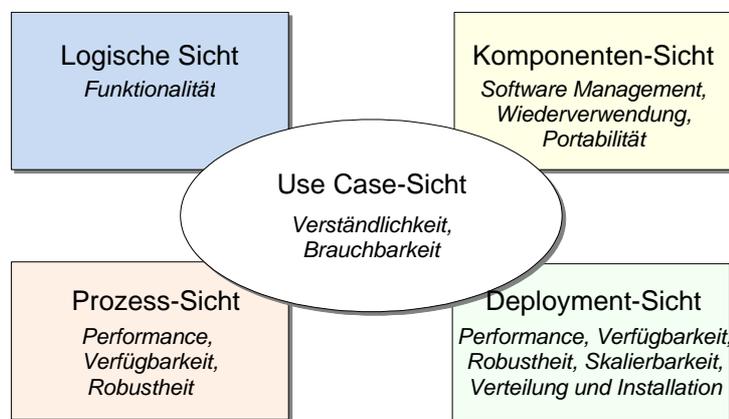


Abbildung 5.2: Die 4+1 Sicht

Logische Sicht In der logischen Sicht des 4+1 Modells wird betrachtet, wie Funktionalitäten innerhalb des Systems entworfen wurden. Zur Darstellung verwendet man hier meist UML-Syntax. Die verwendeten Techniken, die die logische Sicht auf das System ausarbeiten sind Abstraktion, Kapselung und Vererbung.

Prozess-Sicht In der Prozess Sicht wird der Ablauf des Programms betrachtet. Man spezifiziert hier, welcher Programmthread, welche Operationen ausführt und wie die einzelnen Threads nebeneinander existieren. Gegenstand der Betrachtung sind hier auch Eigenschaften des Systems wie Verfügbarkeit, Fehlertoleranz und Verteilung. Bei der Entwicklung von verteilten Telematikanwendungen ist diese Sicht besonders wichtig.

Komponenten-Sicht Die Komponenten Sicht, die manchmal auch als Entwicklungssicht bezeichnet wird, wirft einen Blick auf die Organisation der kodierten oder binären Elemente des Systems. Wie spielen diese zusammen und welche

Schnittstellen verbinden sie? Die Begriffe, die hier einer genaueren Untersuchung unterzogen werden, sind unter anderem Subsysteme, Module, Komponenten und Schichten (Layer).

Deployment-Sicht Die Deployment Sicht oder auch physikalische Sicht beschäftigt sich mit der Frage, wie die verschiedenen Systemelemente, die in den drei oben beschriebenen Sichten herausgearbeitet wurden, auf die verschiedenen Knoten des Systems verteilt werden. Unter einem Knoten versteht man eine beliebige Ressource des Systems (siehe auch 2.9). Betrachtungsschwerpunkte sind hier Performanz und Skalierung.

Bei eingebetteten Systemen ist eine solche Beschreibung unabdingbar, da sie erst den Rahmen für das neu zu entwickelnde System beschreibt. Meist erstellt man deshalb ein solches Modell bereits zur Anforderungsanalyse. Zur Beschreibung der Thread- und Prozesseigenschaften von Software wird separat noch einmal in 5.10 eingegangen.

Use Case-Sicht In den Use Cases der Use Case Sicht wird die Funktionalität des Systems aus Sicht eines externen Akteurs untersucht. Die verwendete Technik sind Szenarien, die die in den vier anderen Sichten definierten Objekte verwenden. Sie stellt eine Abstraktion der wichtigsten Anforderungen an das Softwareprodukt dar.

5.5.5 Architekturbeschreibung

Es wurde in den vorherigen Abschnitten viel über die Wege und Hilfsmittel zum Auffinden und definieren einer Architektur gesagt. Wie aber sieht nun genau ein Architekturmodell aus und welche wichtigen Elemente beinhaltet es?

Nicht alle Elemente des Gesamtmodells sind relevant für die Architektur. Kann eine Änderung der Spezifikation eines Modellelementes lediglich das Subsystem verändern, in dem es sich befindet, so handelt es sich hierbei um kein architekturrelevantes Element. Es gehört also nicht in die Architekturbeschreibung. In diese Gruppe fallen die meisten Klassen, Schnittstellen und Operationen, die privaten Zugriff innerhalb ihres Subsystems haben. Zur Architekturbeschreibung gehören also nur die architekturrelevanten Use Cases, Subsysteme, Schnittstellen, einige Klassen, Komponenten, Knoten und Kollaborationen. Des weiteren sollte sie eine Beschreibung der verwendeten Plattform, Komponentenarchitektur, Datenbanken u.ä. beinhalten.

5.6 Iterativ und inkrementell

5.6.1 Entwicklungsphasen

Die Abwicklung eines Softwareprojektes lässt sich in folgende Hauptphasen einteilen: Analyse, Design, Implementation und Test.

In der Analyse geht es vor allem darum, den Problemkontext der zu erstellende Software zu erfassen und zu untersuchen. Des weiteren werden die Anforderungen definiert, die bei der Auslieferung der fertigen Software von dieser erfüllt

sein müssen. Zu ihnen gehören sowohl funktionale Anforderungen, die die Funktionalitäten des Produkts festlegen, als auch qualitative, die Rahmenbedingungen bezüglich Performanz, Sicherheit, Verfügbarkeit und Fehlerrobustheit beschreiben. Zudem versucht man in der Analyse-Phase, das Verhalten des Systems in bestimmten Szenarien festzulegen. Eine Untersuchung der Architektur, in der die Software eingesetzt werden soll ist ebenso Bestandteil der Analyse wie erste Grundsatzentscheidungen zur Architektur der Software selbst.

Auf den Ergebnissen der Analyse aufbauend wird in der Designphase ein spezieller Lösungsansatz entwickelt, der alle ausgearbeiteten Anforderungen erfüllen kann. Hier geht es darum, die verschiedenen Aspekte des neuen Systems vollständig zu spezifizieren. Ein gutes Design lässt sich daran erkennen, dass ein Entwickler, der erst nach der Designphase dem Entwicklerteam beitrifft, nur mit Hilfe der Dokumente der Designphase die Implementation des Systems übernehmen könnte. Deshalb werden beim Design auch alle Klassen von Objekten, ihr Zusammenspiel und Verteilung in der Gesamtarchitektur ausgearbeitet und festgehalten.

Die Implementation ist dann lediglich eine Instantiierung des entworfenen Lösungsansatzes. Erst hier werden Entscheidungen für oder gegen bestimmte Techniken, wie die verwendete Programmiersprache, Komponentenarchitektur oder Betriebssystem, wirksam. In der Designphase versuchte man noch eine eher allgemeine Beschreibung zu finden, die bei der Implementation dann ihre konkrete Umsetzung findet.

Die letzte Phase ist dann die Testphase, in der das implementierte Produkt daraufhin untersucht wird, ob es konform mit dem in der Designphase entworfenen Modell ist und allen aufgestellten Anforderungen genügt.

Die Beschreibung der Phasen selbst reicht noch nicht aus, um als Softwareentwicklungsprozess zu dienen. Es ist auch wichtig zu wissen, wann welche Phase wie oft auftreten sollte.

5.6.2 Lebenszyklus

Die Menge von Tätigkeiten bei der Entwicklung einer Software, die aufeinanderfolgend realisiert werden, nennt man einen Lebenszyklus. Es existieren in der Hauptsache zwei unterschiedliche Lebenszyklus-Modell: Das Wasserfallmodell und der iterative Ansatz.

Wasserfallmodell

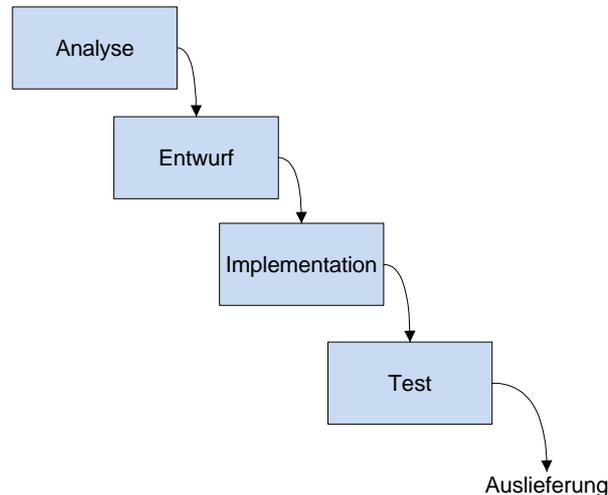


Abbildung 5.3: Wasserfallmodell

Das sogenannte Wasserfallmodell basiert auf einer streng sequentiellen Abfolge der einzelnen Entwicklungsstufen. Erst wenn eine Phase vollständig abgeschlossen ist, kann die nächste begonnen werden. Der Vorteil dieses Ansatzes ist ein einfaches Planen, da es lediglich nötig ist, den Startpunkt der einzelnen Phasen festzulegen und diese dann unabhängig voneinander zu planen.

Das Wasserfallmodell hat jedoch auch große Nachteile. So kann eine neue Phase erst beginnen, wenn die vorherige abgeschlossen wurde. Dies kann zur Folge haben, dass einige Mitarbeiter für längere Zeit darauf warten müssen, dass andere ihre Aufgabe beenden. Oftmals ist es auch schlichtweg nicht möglich die Analyse-Phase ohne jegliche Designentscheidung abzuschließen.

Der größte Nachteil des Wasserfallmodells ist jedoch, dass oft erst zu späten Phasen der Entwicklung Fehlentscheidungen entdeckt werden. Die Entwicklung des gesamten Projekts hängt bereits unabänderlich von den in der Analyse-Phase getroffenen Entscheidungen ab. Stellt sich bei der Implementation oder gar erst beim Testen heraus, dass einige Grundsatzentscheidungen falsch oder unvollständig waren, so muss man das gesamte Projekt von Neuem beginnen. Das kostet Zeit und damit Geld. Durch die Unflexibilität des Wasserfallmodells ist es schwierig, auf Neuerungen im technischen Bereich noch während der Entwicklung einzugehen. Vor allem bei großen Projekten, die sich über mehrere Monate erstrecken können, läuft man somit Gefahr bei Auslieferung des fertigen Produkts ein bereits auf obsolete Techniken aufbauendes, unbrauchbares System entwickelt zu haben.

5 Entwicklungsprozess für verteilte Telematik

Ziel muss es also sein, Fehler in möglichst frühen Phasen des Prozesses zu entdecken, um die Kosten eines neuen Design so gering wie möglich zu halten. Dies versucht der iterative Ansatz zu erreichen.

Iteratives Modell

Das iterative Modell ist wesentlich komplexer gehalten als das Wasserfallmodell. Es beruht im Groben darauf, dass innerhalb der Entwicklungszeit mehrere Lebenszyklen durchlaufen werden. Eine Iteration ist dabei eine Art Mini-Projekt. Der Gesamtprozess ist –ähnlich dem Wasserfallmodell- in vier Hauptphasen eingeteilt: Projektbeginn (Inception), Entwicklung (Elaboration), Konstruktion (Construction) und Auslieferung (Transition). Diese Phasen sind dann in mehrere Iterationen unterteilt.

Der Abschluss einer dieser vier Phasen ist ein Meilenstein. Der Abschluss einer Iteration kennzeichnet einen kleinen Meilenstein. Ist der Prozess an einem Meilenstein angelangt, muss beschlossen werden, wie das Gesamtprojekt in die nächste Iteration voranschreitet.

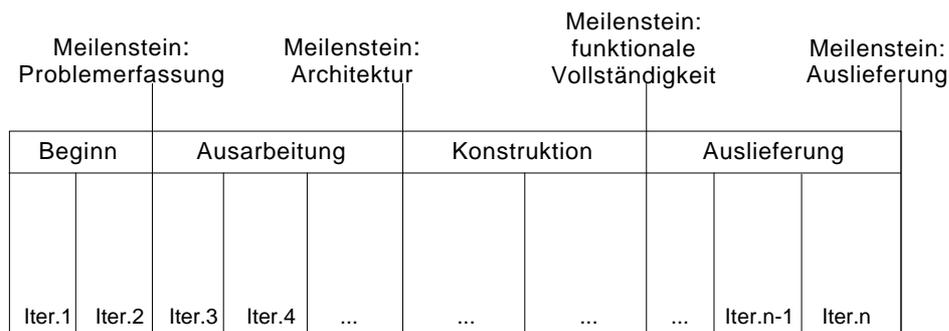


Abbildung 5.4: Meilensteine

In Kapitel 6 werden alle in Abbildung 5.4 dargestellten Phasen und Meilensteine einmal prototypisch durchlaufen.

5.6.3 Iterationen

Jede Iteration ist im allgemeinen aus fünf Workflowschritten aufgebaut, die jedoch nicht alle zwingend durchlaufen werden müssen. Beginnend mit einer Planungsphase endet jede Iteration in einem abgeschlossenen, getesteten Programm, das zur internen Veröffentlichung freigegeben werden kann. Ein solches Programm ist nicht für den Kunden bestimmt, sondern kennzeichnet lediglich den Abschluss einer Iteration. Man nennt einen solchen Iterationsabschluss auch Inkrement (5.6.4). Die fünf Schritte einer Iteration sind in Abbildung 5.5 dargestellt.

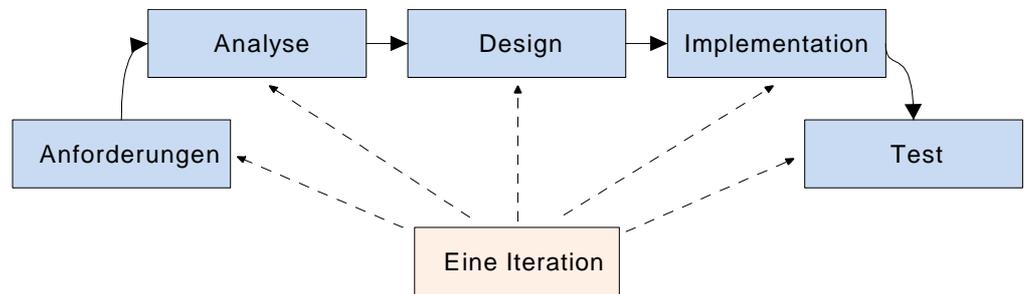


Abbildung 5.5: Eine Iteration

Der Schwerpunkt von Iterationen in frühen Phasen des Gesamtprozesses liegt in der Erforschung des Problembereichs und der eingesetzten Technologien und Werkzeuge. Später dient eine Iteration dazu, Teile der Grundarchitektur festzulegen und zu entwickeln. Aufbauend darauf werden in den Iterationen der Implementationsphase Teile des Produkts selbst entwickelt. Die Iterationen der Transitionsphase zielen dann konkret darauf ab, aus den Artefakten der vorangegangenen Phasen ein auslieferbares Produkt zu machen. Im Laufe der Entwicklung des Gesamtsystems verschiebt sich innerhalb einer jeden Iteration der Schwerpunkt der Tätigkeit von der Anforderungsanalyse hin zu Implementation und Test. Dies veranschaulicht Abbildung 5.6.

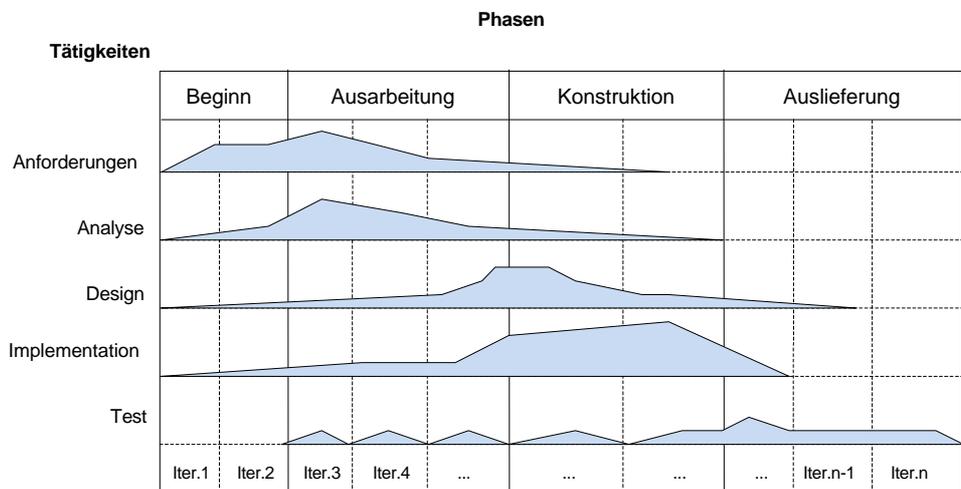


Abbildung 5.6: Schwerpunkte in den einzelnen Entwicklungsphasen

Nach jeder Iteration ist es wichtig, festzustellen, ob neue Anforderungen an das Produkt aufgetreten sind, die nachfolgende Iterationen beeinflussen könnten. Zusätzlich muss nach jeder Iteration überprüft werden, ob die Ergebnisse kompa-

tibel zu den vorangegangenen Iterationen und zu den Anforderungen zu Beginn der Iteration sind.

5.6.4 Inkrement

Das Resultat einer Iteration ist ein Inkrement. Ein Inkrement kennzeichnet den Zuwachs, den das Produkt innerhalb einer Iteration erfahren hat. Nach und nach nimmt so das zu entwickelnde System immer mehr die Form und die Funktionalität an, die zu Projektbeginn gefordert wurden.

Alle Modelle müssen zum Zeitpunkt eines Inkrements konform zueinander sein. Das bedeutet, dass beispielsweise das Use Case Modell, von dem unter Umständen nur Teile durch das Inkrement realisiert werden, konsistent zum Design Modell sein muss. Alle Elemente der Modelle müssen untereinander ebenfalls konform sein.

Es wird im allgemeinen der Fall auftreten, dass einige Subsysteme bereits die volle Funktionalität bieten, andere hingegen noch Code-Gerüste ohne Funktionalität sind. Innerhalb eines Inkrements müssen aber alle Teile miteinander arbeiten können. In bestimmten Fällen bietet sich dann das Erstellen eines Prototypen an.

5.6.5 Prototyp

Ein Prototyp ist eine Instanz des Systemmodells. Bei der Softwareentwicklung bedeutet das in den meisten Fällen ein ausführbares Programm. Man unterscheidet zwischen iterativen und Wegwerf-Prototypen. Wegwerf-Prototypen haben Elemente, die in einer bestimmten, späteren Iteration vollständig überarbeitet werden. Das können z.B. grafische Oberflächen sein, die später von einem Designer überarbeitet werden oder auch eine Komponente, die lediglich eine Standardimplementierung ihrer Schnittstelle bietet, die aber noch nicht die Funktionalität des Endprodukts hat.

Bei einem iterativen Prototypen werden die Elemente, aus denen er zusammengesetzt ist, in späteren Iterationen weiterentwickelt. Deshalb müssen diese auch von einer bessern Qualität als bei einem Wegwerf-Prototypen sein. Jeder iterative Prototyp ist eine signifikante Weiterentwicklung der vorherigen. Er sollte also beispielsweise einen bestimmten Use Case realisieren, der vorher noch nicht realisiert war. Beim inkrementellen Vorgehen ist es wichtig einen vertikalen Prototypen zu entwickeln. Was vertikal in diesem Kontext bedeutet, zeigt Abbildung 5.7.

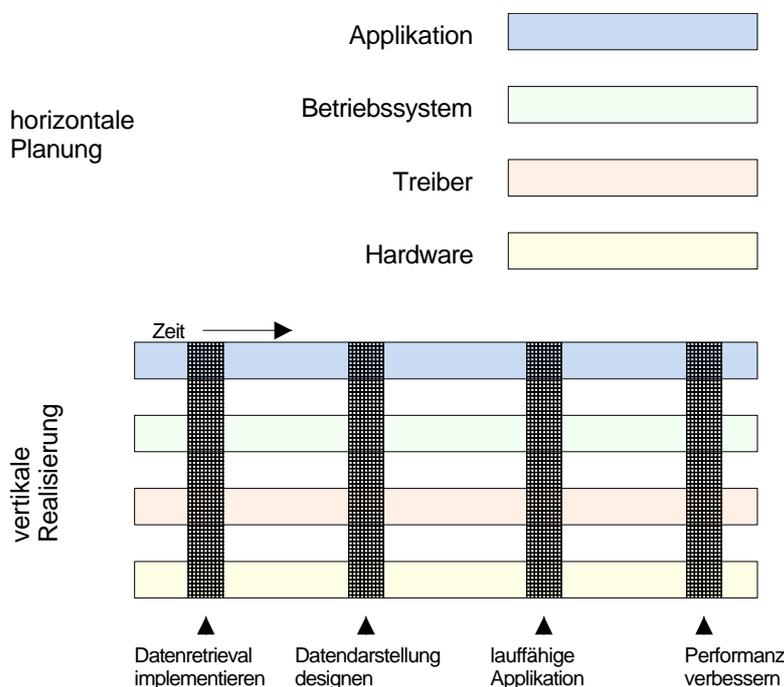


Abbildung 5.7: vertikale Prototypen

Bei der Realisierung eines inkrementellen Prototypen ist es wichtig, alle Schichten mit einzubeziehen. Besonders bei Telematik-Anwendungen muss sichergestellt sein, dass ein Prototyp nicht nur auf der Entwicklungs- sondern auch auf der Zielplattform lauffähig ist. Dazu kann unter Umständen die Entwicklung von speziellen Treibern oder Hardwarekomponenten wichtig sein. Wegen falschen Voraussetzungen bezüglich Hardware und Betriebssystemen kann ein ganzes Projekt scheitern. Möchte man zum Beispiel eine Telematik-Anwendung für die Massenproduktion herstellen, können Änderungen in den Hardwarevoraussetzungen immense Kosten verursachen.

Gerade bei der Erstellung der frühen Prototypen ist man als Entwickler dazu gezwungen, sich über die Realisierung von wichtigen Systemkomponenten Gedanken zu machen, da sonst die Erstellung eines ausführbaren Programms nicht möglich ist. Es müssen Entscheidungen darüber getroffen werden, wie das System sinnvoll in Komponenten zerlegt wird, welche Systemdienste man wie in Anspruch nimmt und wie verteilte Systemteile miteinander kommunizieren. Die Definition von sauberen Schnittstellen wird also durch das Erstellen und Untersuchen von Prototypen erleichtert und gefördert. Das in Kapitel 6 entworfene System stellt einen solchen Prototypen dar.

5.6.6 Anforderungsänderungen

Ein großer Vorteil des iterativen Ansatzes ist die Möglichkeit, Änderungen in den Anforderungen an die entwickelte Software schnell zu erkennen und in den Projektplan zu integrieren.

Durch die ausführbaren Prototypen der frühen Entwicklungsphasen hat der Kunde oder Endbenutzer des System frühzeitig die Möglichkeit eine eingeschränkten Version zu testen. Der Fortschritt und Stand der Entwicklung ist mit einem lauffähigen Programm, auch wenn es nur eingeschränkte Funktionalität bieten kann, für einen Kunden wesentlich besser zu erfassen als eine Masse an Dokumentation. Oft ist der Problembereich, in dem die Software eingesetzt wird, dem Kunden besser bekannt als dem Entwickler, so dass dieser auch eher grundlegende Designfehler des Programms in Hinblick auf Arbeitsablauf und Problemerkennung erkennen kann.

In einem inkrementellen Prozess beziehen sich Änderungen immer nur auf das aktuelle Inkrement, da frühere Inkremente ja bereits durch die Qualitätssicherung gekommen sind. Der Risikofaktor ist somit wesentlich geringer als beim Wasserfallmodell, bei dem unter Umständen der ganze Prozess von vorn aufgerollt werden muss.

Angesichts der schnellen Weiterentwicklung von Hardware ist auch in einem Telematikprojekt ein inkrementelles Vorgehen sinnvoll. Nur so kann eine hinreichende Flexibilität des Entwurfs bezüglich neuer Systemvoraussetzungen erreicht werden.

5.7 Modellkonsistenz

Die Gewährleistung und Überprüfung der Konsistenz der einzelner Modellelemente stellen ein großes Problem dar. Häufig kann eine Überprüfung nicht werkzeuggestützt erfolgen, da beispielsweise Inkonsistenzen in der Semantik von Operationen aufgetreten sind. Auch die Überprüfung der Realisierung von Use Cases ist nur sehr schwer mittels Werkzeugen durchführbar.

Abstraktion, Präzision und kompatible Komponenten sind drei Prinzipien, die die Nachvollziehbarkeit und Konsistenz des Gesamtmodells unterstützen.

5.7.1 Abstraktion

Abstraktion ist ein Prozess zur Verringerung der Detailstufe. Dieser hat den Vorteil, dass er eine Fokussierung auf die wirklich wichtigen Dinge des Problemereichs erleichtert. Abstraktion ist vor allem beim Bearbeiten von sehr komplexen Systemen hilfreich.

Stellt man sich einen Softwareentwicklungsprozess vor, so beinhaltet dieser eine Reihe von Entscheidungen, die getroffen werden müssen und in den meisten Fällen von früheren Entscheidungen abhängen. Einige dieser Entscheidungen sind für das Fortkommen des Projekts unabdingbar. So würde man nie auf die Idee kommen eine Datenbank für ein Informationssystem zu entwerfen, bevor

man nicht spezifiziert hat, welche Information das System überhaupt bereitstellen soll. In vielen Projekten werden solche wichtigen Entscheidungen nicht oder zu spät getroffen, was nicht selten zur Folge hat, dass scheinbar kleine Probleme große Kosten verursachen.

Folgende Prozessartefakte sollten immer zunächst in einer möglichst abstrakten Form spezifiziert werden:

- Problembereich, in dem die Software arbeitet
- Anforderungen an die Software
- Interaktionen der Klassen und Objekte ohne detaillierte Protokollbeschreibung
- Architektur

Die UML ist ein gutes Werkzeug, um abstrakte Beschreibungen festzuhalten. Häufig wird zusätzlich eine umgangssprachliche Beschreibung verwendet. In Kapitel 6 werden die oben genannten Punkte noch einmal am praktischen Beispiel nachvollziehbar gemacht.

5.7.2 Präzision

Bei allen Abstraktionen bleibt zu beachten, dass ein hohes Maß an Präzision gewahrt bleiben muss. Präzision hilft dabei, Inkonsistenzen und Lücken früh zu erkennen. Weiterhin wird es einfacher, den Weg von den Anforderungen bis hin zum Code nachzuvollziehen, wenn man durch eine präzise Dokumentation dabei unterstützt wird. Eine semantische Beschreibung von Modellelementen ist nur dann sinnvoll, wenn sie präzise und vollständig ist.

Das präziseste Artefakt innerhalb eines Entwicklungsprozess ist der fertige Code. Hier sind keine Ungenauigkeiten oder Lücken mehr möglich. Da Code aber oft erst zu einem sehr späten Zeitpunkt des Projekts erstellt wird und zudem für Projektfremde nur sehr schwer zu verstehen ist, ist er ein ungeeignetes Mittel zur Präzisierung.

Ein unpräzises Modell erschwert dessen Wartung erheblich, da die Semantik vieler Elemente erst durch eine präzise Dokumentation völlig eindeutig und klar wird. Besonders wichtig ist ein hohes Maß an Präzision in der Beschreibung von Schnittstellen, die zwischen einzelnen Projektgruppen oder auch zwischen Hard- und Software in einem Telematikprojekt bestehen.

5.7.3 Kompatible Komponenten

Die Verwendung von Komponenten macht nur Sinn, wenn diese miteinander kollaborieren können. Das Zusammenspiel mehrerer Komponenten setzt ein hohes Maß an Präzision in der Schnittstellenbeschreibung voraus. Eine gute Komponente erkennt man daran, dass sie mit verschiedensten anderen interagieren kann. Aus diesem Grunde ist es auch notwendig zu spezifizieren, was eine Komponente von anderen innerhalb einer Kollaboration erwartet. Die Idee von zusammensetzbaren Komponenten sollte sich durch den gesamten Entwicklungs-

prozess einer Telematikanwendung ziehen und beim Entwurf immer berücksichtigt werden.

Wiederverwendung ist eine nicht zu unterschätzende Zeit- und Kostenersparnis. Zudem sinkt das Risiko, Fehler zu begehen, die in anderen Projekten vorher schon in ähnlichen Kontexten zu Problemen geführt haben.

5.8 *Prozessmuster*

Jeder Software Entwicklungsprozess verfolgt ein anderes Ziel unter anderen Voraussetzungen. Deshalb ist es auch utopisch anzunehmen, dass jeder Prozess nach dem gleichen Schema ablaufen könnte. Vielmehr muss die Entwicklung den Begebenheiten angepasst werden. Da der Unified Process dafür nur wenig Anhaltspunkte liefert, hat der Catalysis-Ansatz versucht, Regeln und Muster zu finden, anhand derer die nächsten Schritte des Prozess geplant werden können.

Es würde den Rahmen dieser Arbeit sprengen, all diese Muster im einzelnen aufzuzählen, deshalb soll hier nur ein kleiner Überblick gegeben werden, was diese Muster gemeinsam haben und in welchen Fällen sie hilfreich sein können.

Steht der Entwickler vor einer neuen Iteration, muss er entscheiden, was sein nächster Schritt sein soll. Aber welche Kriterien bestimmen, welcher Schritt der richtige ist? Hier können Prozessmuster Hinweise geben. Der Entwickler muss lediglich herausfinden, welches Muster am besten auf seine jetzige Situation passt.

Jedes Muster verfolgt ein bestimmtes Ziel. Sollte dieses Ziel zur aktuellen Situation passen, ist zu überlegen, ob man es anwenden soll. Dazu verhelfen weiterführende Überlegungen, die Risiken, Seiteneffekte und Zeitaufwand bei Anwendung dieses Musters betreffen. Sollten all diese Überlegungen, eine Anwendung des Musters rechtfertigen, liefert die Strategie, die zu dem Muster gehört eine Anleitung, wie weitergearbeitet werden soll.

Häufig liefern Muster nur Regeln für einen Teilaspekt des nächsten Schritts. Es ist also ratsam, mehrere Muster miteinander zu verknüpfen, um einen möglichst großen Bereich des gesamten Spektrum des nächsten Schritts mit Mustern abzudecken.

Prozessmuster

Komponentenspezifikation

Ziele

- Frühe Identifizierung konzeptioneller Probleme und Hindernisse
- Reduzierung von Missverständnissen bei Verantwortlichen und Entwicklern
- Verlängerung der Lebensdauer des Produktes

Überlegungen

Schnelles Codieren kann hilfreich sein, wenn man zu einem Prototypen kommen möchte. Dies funktioniert jedoch nur reibungslos in einem kleinen Projekt. Bei größeren Projekten muss man weniger wichtige von wichtigen Designentscheidungen trennen, um über die einzelnen Teile separat nachdenken zu können. Auch für existierende Komponenten ist eine abstrakte Spezifikation notwendig. Eine Spezifikation aller Komponenten hat zwei Vorteile:

- Spätere Wartung des Codes wird vereinfacht.
- Benutzer der Komponenten wissen genau, welche Eigenschaften und Funktionen die Komponente bereitstellt.

Strategie

Schreibe eine Spezifikation für jede Komponente. Ausnahmen von dieser Regel können bei sehr kleinen Programmteilen mit möglicherweise geringer Lebensdauer gemacht werden. Manchmal ist es auch möglich, die Spezifikation nach der Erstellung des Codes zu liefern.

Konsequenzen

Die Erstellung einer Spezifikation hilft dabei, Probleme zu erkennen, die eine normale Anforderungsanalyse eventuell nicht liefern kann.

Manchmal ist es sehr ermüdend eine Spezifikation zu schreiben und man möchte gerne in die „richtige“ Codierungsphase eintreten. Es sollte darauf geachtet werden, dass man nicht immer eine hundertprozentige Spezifikation fertig haben muss, bevor die Codierung beginnt. Vielmehr können manche Probleme erst während der Programmierung selbst erkannt werden.

Text 1 - Beispiel für ein Prozessmuster

Text 1 ist ein verkürztes Beispiel eines Prozessmusters aus [D'Souza-98]. Dieses Prozessmuster wurde auch bei der Entwicklung der Telematikanwendung in Kapitel 6 verwendet.

5.9 Nachrichten und Ereignisse

Bei verteilten Telematik-Systemen ist eine spezielle Untersuchung von Nachrichten und Ereignissen sinnvoll. In der UML ist die Nachricht (Message) das Kommunikationsmittel zwischen Objekten. Erst in späteren Phasen des Prozesses wird entschieden, ob eine Nachricht als Klassenmethode, Objektverhalten, Nachricht über einen Nachrichtenbus oder als Ereignis implementiert wird. Dies hängt von den Eigenschaften ab, die eine Nachricht besitzt. Welche das sind, soll im folgenden erarbeitet werden.

5.9.1 Eigenschaften von Nachrichten und Ereignissen

Ein Ereignis (Event) ist ein Vorkommnis, das von Bedeutung für das System ist. Häufig wird ein Ereignis in ein Objekt verpackt, das Daten beinhaltet, die das Ereignis näher beschreiben. Beispielsweise kann das Drehen eines Knopfes an einem Gerät in ein Ereignis verpackt werden, das lediglich die Richtung der Drehung angibt. Je nachdem, wie weit der Benutzer den Knopf dreht, desto mehr dieser Ereignisse werden erzeugt. Eine andere Möglichkeit wäre das Einpacken dieses Ereignisses in ein Objekt, das zusätzlich den Grad der Drehung angibt. In diesem Fall würde nur ein Ereignis generiert werden und zwar in dem Moment, in dem der Benutzer aufhört zu drehen. Egal welche der beiden oberen Arten verwendet wird, in allen Fällen wird das Auftreten eines Ereignisses dazu führen, dass eine oder mehrere Nachrichten generiert werden, die an interessierte Objekte weitergeleitet oder von diesen abgerufen werden. Aus diesem Grund kann man die Begriffe Nachricht und Ereignis gleichwertig behandeln. Gerade bei der Kommunikation über Netzwerke, wie sie in verteilten Anwendungen auftritt, sind Entscheidungen über das Design von Nachrichten wichtig.

Muster des Auftretens

Das Muster des Auftretens einer Nachricht beschreibt ihr Zeitverhalten. Das Wissen über das Zeitverhalten von Nachrichten hilft dabei, die richtigen Hardwarevoraussetzungen für das System zu ermitteln und somit das Kostenrisiko des Projekts zu senken. Eine Nachricht hat zwei mögliche Muster für ihr Auftreten: episodisch und periodisch.

Unter episodisch auftretenden Nachrichten versteht man solche, die nicht in einem bestimmten Intervall wiederkehren, sondern ein echtes Ereignis für das System darstellen. Häufig sind das die Nachrichten, die von einem Akteur von ausserhalb initiiert werden. Eine solche Nachricht ist nicht vorhersehbar, kann aber dennoch durch Regeln und Grenzen, die sie einhalten muss, näher beschrieben werden. So kann man die Mindest- oder Durchschnittszeit festlegen, die zwischen dem Auftreten zweier solcher Nachrichten liegen muss. Interessant ist auch die Zeit, die das System zur Abarbeitung einer solchen Nachricht benötigt. Eine solche Spezifikation hilft bei der Analyse des Gesamtsystems. Viele Nachrichten hängen voneinander ab, weshalb es häufig vorkommen kann, dass ganze Nachrichtensequenzen generiert werden, deren gesamter Zeitaufwand natürlich von den Zeiten der einzelnen Nachrichten abhängt.

Eine periodisch auftretende Nachricht tritt immer nach Ablauf eines gewissen Zeitintervalls auf. Solche Nachrichten werden oft zu Synchronisationszwecken generiert. Ein Beispiel für eine solche periodische Benachrichtigung ist z.B. ein GPS (Global Positioning System) in einem Fahrzeug, das in bestimmten Zeitabständen eine neue Position meldet.

Synchronisationsart

Man unterscheidet drei Synchronisationsarten einer Nachricht: blockierend, wartend und asynchron.

Nachrichten vom Typ blockierend und wartend sind sehr ähnlich. In beiden Fällen muss das Objekt, das die Nachricht sendet, auf die Abarbeitung warten. Im Fall einer blockierenden Nachricht geschieht dies im selben Thread, was bedeutet, dass das Senderobjekt den Kontrollfluss an das Empfängerobjekt abgibt und auf dessen Rückkehr warten muss. Beim Versenden einer Nachricht mit der Synchronisationsart wartend übergibt der Sender die Kontrolle an einen anderen Thread oder Prozess und wartet auf die Rückkehr.

Nach Versenden einer asynchronen Nachricht kehrt das Programm sofort zurück. Die Antwort auf die Nachricht erfolgt dann oft über eine sogenannte Callback-Methode, die der Sender dem Empfänger für dessen Antwort zur Verfügung stellt. Beispiele für asynchrone Kommunikationsmechanismen finden sich in 6.10.5.

Sender und Empfänger

Eine weitere wichtige Eigenschaft einer Nachricht, die in der Analyse eines zeitkritischen System nicht fehlen darf, ist die Festlegung, welche Objekte welche Nachrichten wohin verschicken dürfen. Hierdurch können Schlüsse bezüglich der benötigten Technik und Softwarearchitektur gezogen werden.

5.9.2 Ereignisliste

Typischerweise fasst man die erlangten Analyseergebnisse bezüglich Nachrichten und Ereignissen in einer Ereignisliste zusammen. Diese enthält die wichtigsten Ereignisse, auf die das System reagieren soll, und deren Eigenschaften. Tabelle 5.1 zeigt eine vereinfachte Ereignisliste für das Beispiel ABS aus 4.3.1.

<i>Ereignis</i>	<i>Beschreibung</i>	<i>Empfänger</i>	<i>Muster</i>	<i>Zeitregeln</i>
Abfrage der Raddrehung	Fragt die momentane Drehgeschwindigkeit der Räder ab.	System	periodisch	10 ms Periode
Blockieren der Reifen	Meldet das Auftreten einer Radblockierung trotz Vorwärtsbewegung des Fahrzeugs	System	episodisch	< 10 ms Antwortzeit

Tabelle 5.1: Auszug aus der Ereignisliste eines ABS

5.10 Threads und Ressourcen

Verteilte Systeme bestehen immer aus mehreren nebenläufigen Threads, die im allgemeine nicht voneinander unabhängig sind. Das hat zur Folge, dass die von mehreren Threads gemeinsam benutzten Ressourcen mit Hilfe einer bestimmten Zugriffssteuerung geschützt werden müssen.

Man verwendet dazu oft sogenannte Synchronisationspunkte. Ein Synchronisationspunkt ist ein bestimmter Zustand des Systems auf den Threads warten, um ihre Ausführung fortzusetzen. Die UML bietet zur Modellierung von Synchronisationspunkten die Aktivitätsdiagramme (2.7) an. Abbildung 5.8 zeigt ein Beispiel für eine Threadsynchronisation. Beide Threads warten darauf, dass der andere seine Berechnung –in diesem Fall die einer Reihe von Bildzeilen– beendet hat, damit die Ergebnisse zu einem Bild zusammengesetzt werden können.

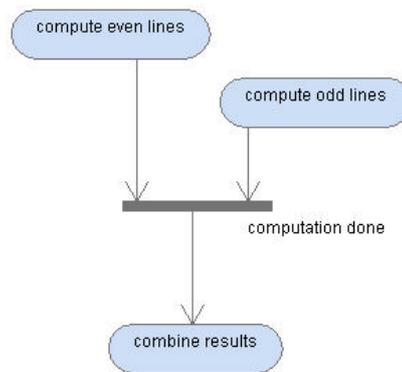


Abbildung 5.8: Threadsynchronisation

Es gibt verschiedene Möglichkeiten, wie Threads, die eine bestimmte Ressource verwenden möchten, synchronisiert werden können. Bei der einfachsten blockiert der Thread einfach solange, bis die von ihm gewünschte Ressource frei wird. Diese Blockierung kann auch mit einem Zeitintervall versehen werden, nach dessen Ablauf der Thread entscheiden muss, welche Aktionen er vornehmen möchte. Bei einer asynchronen Ressourcenanforderung meldet der Thread lediglich, dass er an einer bestimmten Ressource interessiert ist und fährt dann mit der Abarbeitung seines Codes fort. Die Ressource wird ihm dann mit Hilfe einer Nachricht zur Verfügung gestellt, sobald dies möglich ist.

Gerade in verteilten, komponentenbasierten Programmen muss sich der Entwickler früh über den Algorithmus zur Vergabe von beschränkten Ressourcen und Komponenten Gedanken machen. Zudem sollte beachtet werden, in wie weit die verwendete Komponententechnik ein solches Management bereits durch einen Service zur Verfügung stellt. In 6.10.4 wird beispielhaft ein auf Prioritäten basierender Algorithmus entwickelt und spezifiziert.

5.11 Zusammenfassung

Es wurde in diesem Kapitel erarbeitet, dass ein Telematik-Entwicklungsprozess ähnliche Eigenschaften haben muss wie ein Prozess für gewöhnlich Software. Auch hier macht eine Zentrierung auf Use Cases, vor allem in Hinblick auf die Erstellung von vertikalen Prototypen, Sinn. Die Entwicklung solcher Prototypen macht ein iteratives Vorgehen, wie es oben beschrieben wurde unabdingbar.

Da die Beteiligten einer Software-Entwicklung im Telematik-Bereich nicht nur Software-Ingenieure sondern Techniker aller Art sein können, ist die Zentrierung auf eine saubere Architekturbeschreibung als Kommunikationsgrundlage hier besonders wichtig.

Die besonderen Anforderungen und Eigenschaften einer Telematikanwendung machen die Beschreibung von Ereignissen, sowie Threads- und Ressourcen-Verwaltung zu einem weiteren Schwerpunkt des Entwicklungsprozesses.

Die in diesem Kapitel erarbeiteten Eigenschaften sollen nun auf ihre praktische Anwendbarkeit anhand eines Beispiels geprüft werden.

6 Entwurf einer verteilten Telematikanwendung

6.1 Notation

In diesem Kapitel wird es vermehrt Bezüge zu den in Kapitel 5 erarbeiteten Prozesseigenschaften geben. Diese werden vom eigentlichen Entwurf durch einen solchen Kasten abgehoben.

Zudem werden häufig Referenzen auf Kapitel 5 explizit mit Kapitelnummer und Titel angegeben. Dem Leser soll so die Möglichkeit gegeben werden, den Entwicklungsprozess nachzuvollziehen und aktuell umgesetzte theoretische Grundlagen noch einmal nachzuschlagen.

Auch Quelltext wird durch eine eigene Formatierung vom übrigen Text abgesetzt.

6.2 Motivation

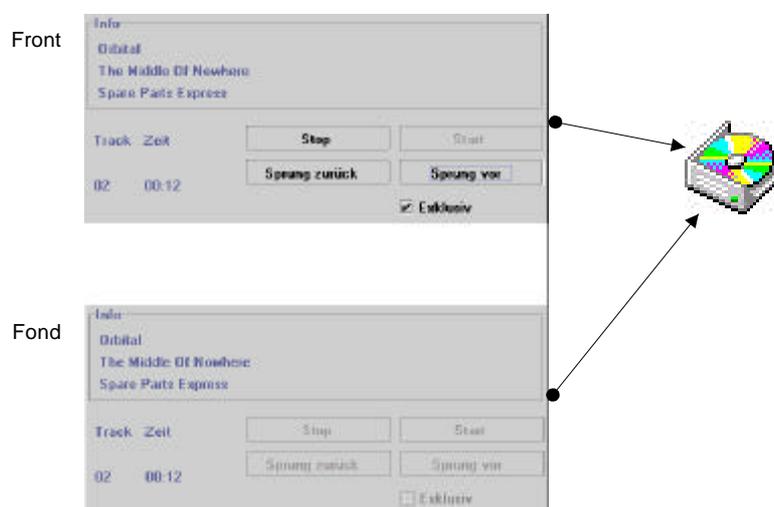


Abbildung 6.1: Zwei Einheiten mit laufender CD-Spieler Applikation

In diesem Kapitel soll ein Prototyp einer Telematik-Anwendung nach den in den vorangegangenen Kapiteln erarbeiteten Richtlinien entworfen werden. Es handelt sich hierbei um ein System, das in einem Fahrzeug eingesetzt werden kann. Das

6 Entwurf einer verteilten Telematikanwendung

System verfügt über die Besonderheit, dass es mehrere Bedieneinheiten gibt, die unabhängig voneinander arbeiten können. So kann beispielsweise der Fahrer den CD-Spieler bedienen, während ein weiterer Insasse auf einem anderen Terminal im Internet surft.

Dieses Kapitel stellt eine Entwicklungsstudie dar, deren Ergebnisse beim Entwurf einer Gesamtarchitektur hilfreich sein können. Der hier entwickelte Prototyp wird trotz seiner geringen Komplexität alle Phasen eines großen Entwicklungsprozesses durchlaufen, damit dieser erläutert werden können.

Referenzen: 5.6.5 Prototyp

6.3 Anforderungen

Die Entwicklung beginnt mit der Sammlung der Anforderungen an das zu entwickelnde Softwaresystem. Diese werden im Dialog mit dem Auftraggeber ausgehandelt und schriftlich festgehalten. Implementationsdetails bleiben hier außen vor.

*Phase: Beginn
1. Iteration*

Es kommt jetzt darauf an, eine abstrakte Beschreibung des Problemraums zu erarbeiten. Dieser dient dann als Vertragsgrundlage zwischen dem Auftraggeber und dem Entwickler. Es wird hier schon eine Gruppierung der Eigenschaften nach Funktionalitäten vorgenommen, um eine Strukturierung zu erlangen. Diese kann später dazu dienen, Komponenten des Systems zu finden.

Referenzen: 5.6.3 Iterationen, 5.7.1 Abstraktion

Es sollen nun alle Anforderungen aufgestellt werden, die das System erfüllen soll. Es wird im folgenden CarTel, abgeleitet von *Car* und *Telematik*, genannt. Bei den Anforderungen handelt es sich um natürlichsprachliche formulierte Funktions- und Qualitätsmerkmale, die vor Beginn der eigentlichen Entwicklung ausgehandelt werden. Zusätzlich enthalten sie Richtlinien zu den verwendeten Technologien und der Zielplattform der zu entwickelnden Software.

Es wird bei dieser Aufstellung ein Top-Down-Weg verwendet, bei dem zunächst die Anforderungen an das Gesamtsystem und seine Umgebung, dann an einzelne Komponenten des Systems und schließlich an konkrete Details dieser Komponenten spezifiziert werden.

Die Zielplattform von CarTel ist WindowsNT. Diese ist gewählt worden, damit in späteren Ausbaustufen eine Realisierung für WindowsCE erleichtert wird. CarTel verwendet das Netzprotokoll TCP/IP, auf dem die Kommunikation zwischen den Komponenten und Recheneinheiten des Systems basiert. Beim Einsatz in einem Auto könnte auch ein anderes Protokoll verwendet werden, das den Anforderungen im Telematikbereich besser gerecht wird. Da jedoch das Protokoll vollkommen transparent verwendet wird, ist dessen Wechsel mit geringem Aufwand möglich.

CarTel stellt bestimmte Dienste zur Verfügung, die verteilt angesprochen werden können und nun im einzelnen erklärt werden:

6.3.1 Zugriffs- und Benutzerverwaltung

Benutzer von CarTel müssen sich am System nur anmelden, wenn sie mit persönlichen Daten arbeiten möchten (Terminplaner, Mail, etc.). Die Benutzung der meisten Komponenten des Systems kann ohne eine Anmeldung erfolgen. Der CD-Spieler beispielsweise benötigt keine Informationen über den derzeitigen Benutzer. Ist der Benutzer eines Rechners mit laufendem CarTel-System(Knoten) nicht persönlich angemeldet, wird sein Berechtigungsprofil nur durch diesen Rechner und dessen Position (Front, Fond) bestimmt. Ein Berechtigungsprofil beschreibt die Regeln und Einschränkungen beim Zugriff auf Komponenten und Applikationen des Systems. Die Zugriffssteuerung hat die Möglichkeit, bestimmte Bereiche des Systems vor unberechtigten Zugriffen zu schützen.

Konflikte, die bei gleichzeitiger Verwendung von System-Funktionalitäten entstehen können, müssen durch die Zugriffsverwaltung aufgelöst werden. CarTel wird dazu eine Prioritätsverwaltung verwenden, die in 6.10.4 erläutert wird.

Hier hat die Architektur zum ersten mal im Prozess Einfluss auf die nachfolgende Entwicklung. Dadurch dass eine verteilte Anwendung entsteht, muss ein besonderes Augenmerk auf die Deployment Sicht des 4+1 Modells gelegt werden. Viele weitere Entwurfsentscheidungen fallen vor dem Hintergrund der Verteiltheit.

Referenzen: 5.5 Architekturzentrierung, 5.5.4 Die 4+1 Sicht

6.3.2 Verteiltheit

Da viele Elemente (Geräte, Dienste, etc.) im Gesamtsystem nur einmal vorhanden sind, aber zeitweise parallel von verteilten Knoten verwendet werden sollen, existiert ein Mechanismus, der Geräte systemweit zur Verfügung stellt. Für den Anwender ist es jedoch vollkommen transparent, ob das Gerät, das er gerade verwendet, lokal auf seinem Knoten vorhanden ist oder remote (über das Netzwerk) angesprochen wird.

Programmiersprache und Komponententechnik

Die Wahl der Programmiersprache für das entwickelte System fällt auf Java. Sicherlich bringt dies einige Probleme im Bereich Performance mit sich, da Java ja durch die Notwendigkeit einer virtuellen Maschine und den immensen Heap-Hunger einige Geschwindigkeits- und Speichernachteile gegenüber klassischen Sprachen wie C oder C++ hat. Bei der Entwicklung von CarTel liegt der Schwerpunkt jedoch weniger in maximaler Leistung und Schnelligkeit, sondern vielmehr in der Realisierung der Verteilung und Zugriffssteuerung mit Hilfe von Komponenten. Die Vorteile, die Java mit sich bringt, sind in 7.3 herausgearbeitet.

Einzelne Komponenten, die Schnittstellen für den verteilten Zugriff zur Verfügung stellen wollen, tun dies über die in Java verwendete Verteilungstechnik

RMI, die bereits in Kapitel 3 genauer beschrieben wurde. Zwischen den beiden Knoten liegt ein TCP/IP-Netzwerk, auf dem die RMI-Kommunikation aufsetzt.

Da die Komponenten klar strukturierte Schnittstellen bieten, die durchaus auch mit anderen Sprachen implementiert werden können, ist die Wahl der Programmiersprache zweitrangig. Eine eventuell später notwendige Performance-Optimierung sollte keinerlei Auswirkung auf das in diesem Kapitel vorgestellte Programmdesign haben.

Auf Grund der Verteilung und der Entscheidung, Komponententechnik einzusetzen, ist hier sehr früh ein Beschluss bezüglich Programmiersprache und eingesetzter Komponententechnik gefasst worden. Solche Entscheidungen gehören zur Komponentensicht der 4+1 Sicht.

Eine frühe Festlegung ist in diesem Fall sinnvoll, da CarTel einen lauffähigen vertikalen Prototyp darstellt und der Prozess deshalb möglichst schnell in die Konstruktionsphase einsteigen muss.

Referenzen: 5.6.5 Prototyp, 5.5.4 Die 4+1 Sicht

6.3.3 Monitoring

CarTel stellt eine rudimentäre Möglichkeit zur Verfügung, Informationen über den internen Zustand des Systems und einzelner Komponenten zu erlangen. Über eine Test-Applikation kann man sich über alle Komponenten des Systems, die eine bestimmte Test-Schnittstelle zur Verfügung stellen, Informationen einholen. Diese ist jedoch nur ein Prototyp, um zu verdeutlichen, dass durch die komponentenbasierte Vorgehensweise ein Testen und Überwachen der einzelnen Module, leicht zu realisieren ist.

6.3.4 Persistenzmanagement

Alle persistenten Daten des Systems werden nicht durch Zugriffe beliebiger Komponenten auf das File-System oder anderer Speicherorte, sondern zentral von einer Einheit geregelt. Das hat den Vorteil, dass es transparent bleibt, ob die Daten in Dateien einer Datenbank oder auf andere Art gesichert werden. Weiterhin ist so maximale Flexibilität gewährleistet, was die dem System zu Grunde liegende Speicherhardware anbetrifft.

CarTel stellt zu diesem Zweck ein Konfigurations- und Property-Management zur Verfügung. Erstgenanntes sichert Einstellungen und Optionen, die für die Benutzung des Systems wichtig sind. Das Property-Management stellt Strings zur Verfügung, die von Programmen verwendet werden können. Durch dieses zentrale Management ist beispielsweise der Wechsel in eine andere Sprache sehr leicht zu realisieren, da alle in Programmen verwendeten Strings durch den Propertymanager, abhängig von der Spracheinstellungen geliefert werden.

6.3.5 Geräte

Da CarTel ein verteiltes System mit einer auf Berechtigungen basierenden Zugriffsregelung ist, soll ein Gerät eine Funktion bieten, mit der man exklusiven

Zugriff erlangen kann. Dies ist vor allem vor dem Hintergrund wichtig, dass in Gefahrensituationen im Fahrzeug die Möglichkeit gegeben sein muss, ein Gerät zu sperren (z.B. die Kommunikation, um einen Notruf abzusetzen).

Der Zugriff auf die meisten im System befindlichen Geräte (Grafik, Platten, Netzwerk, etc.) wird bereits von der Java VM bereitgestellt. Um jedoch die Situation in einem tatsächlich eingebetteten System zu simulieren, werden zwei Geräte im Sinne der oben erklärten Zugriffssteuerung über einen Controller implementiert.

Kommunikationsplattform

Der Aufbau einer Verbindung zu einem Internet Service Provider (ISP) wird über ein virtuelles Gerät gesteuert. Über dieses ist es möglich, den Inhalt einer URL anzufordern. Es baut dann automatisch eine Verbindung auf und liefert die angeforderten Daten. Diese Funktionalität wird als Device entworfen, damit sie auch von Knoten ohne Internetzugang genutzt werden kann.

CD-Spieler

Der CD-Spieler, den CarTel zur Verfügung stellt, bietet beispielhaft nur einige rudimentäre Funktionalitäten.

- Start/Stop der Wiedergabe
- Sprung zum nächsten/vorherigen Stück der CD
- Informationen über das eingelegte Medium
- Informationen über die derzeitig gespielte Position auf diesem Medium in Zeit und Nummer des Stücks

Mit Hilfe dieser Funktionsaufzählung lassen sich im Architekturmodell später leicht Use Cases entwickeln, die den Prozess vorantreiben.

Referenzen: 5.4 Use Case Basierung

6.3.6 Applikationen

Die Applikationen, die CarTel bedient, sollen im Kontext des Knotens laufen, auf dem sie gestartet wurden. Folgende Applikationen sind im CarTel-Prototypen enthalten:

Applikationsmanager

Der Applikationsmanager ist eine Applikation, die den Zugriff auf alle anderen Applikationen, die eine Benutzerschnittstelle (Human Machine Interface – HMI) zur Verfügung stellen, ermöglicht. Sie wird immer zum Systemstart gestartet und bildet somit sozusagen die Wurzel-Applikation.

CDDB (CD DataBase)

Ein CDDB-Server ist eine durch das Internet bereitgestellte Datenbank, mit deren Hilfe Informationen über CDs eingeholt werden können¹.

Die CDDB-Applikation ist eine Applikation, die kein HMI zur Verfügung stellen muss. Sie hat lediglich die Funktion über die Kommunikationsplattform eine Verbindung zu einem CDDB-Server aufzubauen und dessen Ergebnisse zurückzuliefern. Sie wird von der CD-Spieler Applikation benötigt.

CD-Spieler

Die CD-Spieler Applikation bietet Zugriff auf alle Funktionen des CD-Spielers mittels eines HMI. Zudem ist sie in der Lage, mit der CDDB-Applikation zu kommunizieren und die gelieferten Daten in einem separaten Bereich darzustellen.

*1. Meilenstein: Problemerkfassung
nächste Phase: Ausarbeitung
Start der 2. Iteration*

An dieser Stelle tritt der Prozess in die Ausarbeitungsphase ein, da die Anforderungsanalyse abgeschlossen ist. Es geht nun darum einen Entwurf zu entwickeln, der all diesen Anforderungen genügt.

Referenz: 5.6.3 Iterationen

6.4 Einteilung in Packages

Die oben spezifizierten Anforderungen an das System verhelfen dazu, ein erstes Überblicksdiagramm zu erstellen. Immer mit dem Hintergedanken der Verteiltheit werden die einzelnen funktionalen Aspekte gruppiert und in eine Einheit (Package) verpackt. Packages aus verschiedenen Schichten, die keine Abhängigkeiten besitzen, können parallel entwickelt werden. Abbildung 6.2 bietet einen nach Schichten sortierten Überblick über die einzelnen Packages, mit denen die verschiedenen Anforderungen erfüllt werden sollen.

Jetzt beginnt die Architekturbeschreibung mit einer Einteilung des Gesamtsystems in Schichten.

Referenz: 5.5.3 Strukturmuster

Alle blau eingefärbten Packages beinhalten integrale Bestandteile und Bibliotheken des CarTel-Systems, wohingegen gelbe Packages konkrete Applikationen oder Devices sind, die keine Kernbestandteile beinhalten. Wie man an den Abhängigkeitspfeilen erkennen kann, greift eine Schicht grundsätzlich nur auf ihre eigene oder eine ihr zugrundeliegende Schicht zu.

Die unterste Schicht stellt das Fundament des Gesamtsystems dar. Es teilt sich auf in die Packages für Dienste (*service*) und Geräte (*device*). In *service* befinden sich die Packages, die Authentifikations- und Zugriffsdienste (*access*),

¹ siehe dazu auch <http://www.cddb.com>

Komponenten-Zugriffsdienste (`broker`), String-Verwaltung (`prop`) und Konfigurationsmanagement (`config`) beinhalten. In `device` liegen alle zur Bereitstellung eines Gerätes wichtigen Schnittstellen und Helferklassen. `net` enthält die Implementierung der Kommunikationsplattform und in `player` liegt das Framework für die Implementierung eines allgemeinen Abspielgerät für Multi-Media-Daten (CD, MD, Video, etc.). Die Kommunikationsplattform ist deshalb innerhalb von CarTel implementiert, da sie einen fundamentaler Bestandteil eines jeden Telematik-Systems darstellt. Das gelbe Package `cdplayer.device` beinhaltet die Implementierung eines CD-Players und leitet sich deshalb von `cartel.device.player` ab.

Über der Dienst- und Geräteschicht liegt die Applikations-Schicht. Im Package `application` liegt das Framework für Applikationen, die im CarTel-System laufen sollen. Es gibt zwei Beispiel-Packages, die dieses Framework verwenden: `cddb` (Applikation, die auf eine CD-Datenbank zugreifen kann) und `cdplayer` (CD-Player Applikation).

6 Entwurf einer verteilten Telematikanwendung

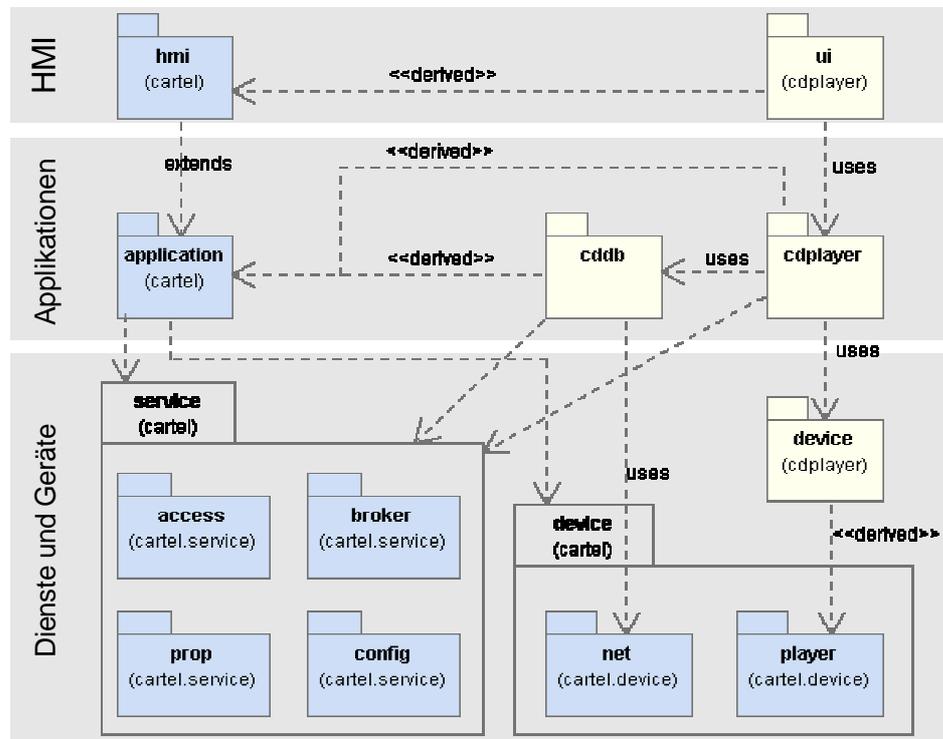


Abbildung 6.2: Die Schichten des CarTel-Systems

Das `hmi` Package beinhaltet die HMI-Bibliothek. Da eine Applikation nicht unbedingt ein HMI haben muss, kann eine Applikation optional mit Hilfe dieses Packages um ein solches erweitert werden. Das einzige Paket, das dieses Package verwendet, ist das Package `cdplayer.ui`, in dem die Benutzungsschnittstelle der CD-Player Applikation liegt.

Eine derartige Darstellung des Systems ist sehr hilfreich, wenn man einen ersten Überblick über Aufgaben erlangen möchte, die bei der weiteren Entwicklung anstehen werden. Alle Entwickler und Entscheidungsträger haben somit eine Grundlage und Vorstellung, auf deren Basis die weiteren Entscheidungen und Entwicklungen getroffen werden können. Zudem kann nun dazu übergegangen werden, Aufgaben zu verteilen und parallel an voneinander unabhängigen Bereichen zu arbeiten.

Die UML dient als Hauptausdrucksmittel in der Architekturbeschreibung.

Referenz: 5.5 Einflussfaktoren der Architektur, Kapitel 2

6.5 Startsequenz

CarTel besteht aus einem Server- und einem Client-Programm. Das Server-Programm initialisiert alle Komponenten der Service- und Device-Schichten und registriert diese bei der RMI-Registry. All diese Komponenten kommen im

gesamten System nur einmal vor. Beim Initialisieren ist durch die Abhängigkeiten einzelner Komponenten untereinander die Reihenfolge von Bedeutung, die in einer Konfiguration festgehalten werden muss. In einem Fahrzeug muss der Start des Systems in die allgemeine Initialisierung des Fahrzeugs integriert werden.

Jeder Knoten hat nun die Möglichkeit, ein Client-Programm zu starten, das auf die vom Server-Programm in der Registry registrierten Komponenten zugreift. Voraussetzung für den erfolgreichen Start eines Clients ist das Vorhandensein einer Registry und der bei ihr vom Server-Programm registrierten Komponenten.

In den folgenden Kapiteln werden die einzelnen Schritte des Entwicklungsprozesses, die im wesentlichen aus der Realisierung der einzelnen Packages bestehen, vollständig dokumentiert. Die Reihenfolge spiegelt dabei auch zum Teil die bestehenden Abhängigkeiten wieder. So werden zunächst die fundamentalen Komponenten aus Service- und Device-Package entwickelt und danach darauf aufbauend die Applikationen und Benutzungsschnittstellen.

Referenzen: 5.7.3 Kompatible Komponenten, 5.8 Prozessmuster

6.6 String-Verwaltung

Bei der Erstellung eines Programms kann es Probleme geben, wenn man Texte, die dem Benutzer angezeigt werden sollen, fest in den Programmcode aufnimmt. Eine Änderung eines solchen Textes, z.B. beim Wechsel in eine andere Sprache, hat dann erstens ein aufwendiges Suchen im Programmcode und zweitens eine Neukompilierung des Programms zur Folge. Es ist deshalb in einem größeren System wünschenswert, eine zentrale Verwaltungseinheit zu haben, die sich um alle benötigten Strings kümmert und zur Laufzeit entscheiden kann, in welcher Sprache sie ein bestimmtes Wort liefert. Diese Aufgabe ist bei CarTel im Package `cartel.service.prop` angesiedelt. Es ist wünschenswert, dass von ihr im gesamten System nur eine Instanz vorhanden ist, die für die String-Verwaltung zuständig ist, um eine zentrale Haltung der Daten zu ermöglichen.

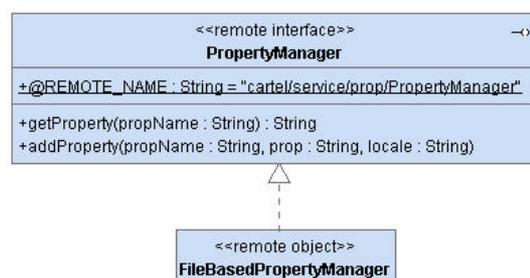


Abbildung 6.3: String-Verwaltung

Abbildung 6.3 zeigt einen Lösungsansatz des Problems. Statische Elemente werden unterstrichen, ein vorangestellter *Klammeraffe* kennzeichnet Konstanten. Das Interface `PropertyManager` besteht aus zwei Methoden. Die Methode

`getProperty()` ermöglicht die Anfrage eines dem `PropertyManager` bekannten Strings. Dieser entscheidet anhand des in Java eingebauten Internationalisierungs-Mechanismus (siehe Anhang), in welcher Sprache der geforderte String zurückgeliefert werden muss. Die Methode `addProperty()` registriert einen String unter einem bestimmten Code für eine bestimmte Sprache. Dieser wird persistent gehalten und ist danach jederzeit wieder abrufbar. Für gewöhnlich wird diese Methode also nur dann aufgerufen, wenn aufgrund von Installationen oder Erweiterungen neue Werte gespeichert werden müssen. Die Implementierung, die bei CarTel zum Einsatz kommt, liefert die Klasse `FileBasedPropertyManager`, die auf Datei-Basis arbeitet.



Abbildung 6.4: Kennzeichnung für Stereotyp *remote interface*

`PropertyManager` ist vom Stereotyp *remote interface*. Es signalisiert, dass auf dieses Interface verteilt zugegriffen werden kann. In den Diagrammen dieses Kapitels wird eine Klasse vom Stereotyp *remote interface* mit dem Zeichen aus Abbildung 6.4 gekennzeichnet. Der Stereotyp *remote object* signalisiert, dass Instanzen dieser Klasse ein *remote interface* implementieren und über die Registry verfügbar sind.

Da es sich bei `PropertyManager` um einen Service handelt, der nur einmal im gesamten System vorhanden ist, ist der Name, unter dem die Implementierung dieses Interfaces angemeldet wird, direkt als statisches, konstantes Attribut `REMOTE_NAME` codiert. Clients, die diesen Service nutzen wollen, müssen also zur Kompilierzeit lediglich die Klasse `PropertyManager` kennen, nicht aber die implementierende Klasse `FileBasedPropertyManager`, da sie mit Hilfe von `REMOTE_NAME` beim Broker (siehe 6.8) eine Implementation anfordern können.

Durch die Trennung von Interface und Implementation ist es kein Problem, die Implementation durch eine andere auszutauschen, die persistente Daten in einem anderen Speicher ablegt, zum Beispiel in einer Datenbank. Die Entscheidung darüber, welche Implementierungen beim Start des Systems gewählt werden, erfragt CarTel bei der Konfigurationsverwaltung, die nun erläutert werden soll.

6.7 Konfiguration

Es ist bei Software immer von Vorteil, wenn bestimmte Einstellungen nicht im Code verankert werden, sondern erst zur Laufzeit an einer bestimmten Stelle erfragt werden, da diese dann ohne Veränderungen am Code geändert werden können. Jedes Objekt, das solche Einstellungen benötigt und speichern möchte, hat die Möglichkeit, Konfigurationen unter einem bestimmten Namen anzulegen. Die dadurch angelegte Konfiguration ist auch beim nächsten Programmstart noch abrufbar und veränderbar. Im Unterschied zur Stringverwaltung ist hier keine Internationalisierung notwendig. Zudem kann eine Konfiguration nicht nur Strings, sondern beliebige serialisierbare Objekte enthalten.

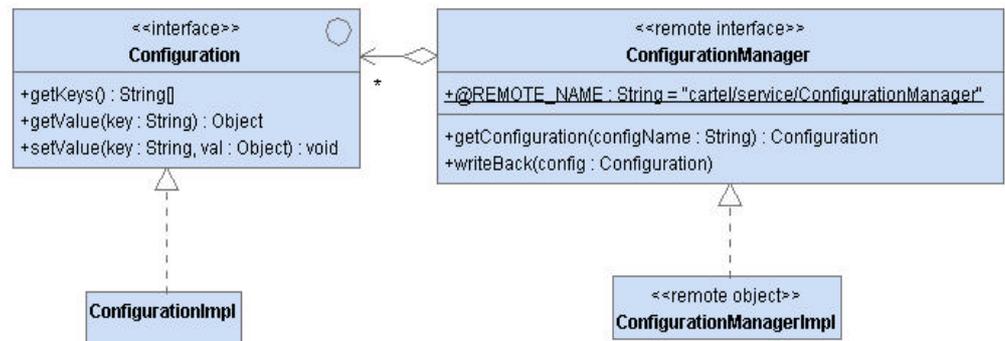


Abbildung 6.5: Konfigurationsverwaltung

`Configuration` ist ein sehr einfach gehaltenes Interface, das die Abfrage und das Setzen von Werten respektive eines als String codierten Namens ermöglicht. Zudem besteht die Möglichkeit, alle verfügbaren Schlüssel zum Abruf von Daten mit `getKeys()` zu erfragen.

Alle `Configuration`-Objekte werden im Service `ConfigurationManager` gehalten. Auch dieser ist nur einmal im System vorhanden und hat deshalb seinen Namen, mit dem seine Implementierung in der Registry bekannt gemacht wird, hart codiert. Um eine bestimmte `Configuration` zu erhalten, ruft man die Methode `getConfiguration()` mit dem Namen der gewünschten Konfiguration als Parameter auf. Ist unter diesem Namen keine Konfiguration vorhanden, so wird automatisch eine leere angelegt und zurückgeliefert. Möchte man Änderungen, die in einer bestimmten Konfiguration gemacht wurden, sichern, so muss man dies mit Hilfe der Methode `ConfigurationManager.writeBack()` tun. Diese Funktion ist nicht im Interface `Configuration` angesiedelt worden, damit ein Transfer von Konfigurationen nicht mittels der Referenz auf ein entferntes Objekt, sondern via Serialisierung möglich wird, was eine Verringerung der Aufrufe über das Netzwerk zur Folge hat. Beim Zurückschreiben wird ein Vergleich mit der ursprünglich herausgegebenen Konfiguration angestellt und alle Änderungen übernommen. Somit ist gewährleistet, dass eventuelle, parallel gemachte Änderungen durch einen anderen Client nur dann überschrieben werden, wenn die zugehörigen Werte auch in dieser Version geändert wurden.

6.8 Komponentenverwaltung

Für gewöhnlich ist es in einem objekt-orientierten Softwaresystem so, dass Grundfunktionalitäten mit Hilfe von Interfaces modelliert werden, die dann von bestimmten Komponenten realisiert werden. Dies hat den Vorteil, dass es relativ unproblematisch ist, die Implementierung eines Interfaces auszutauschen, da eine Bindung von Implementierung und Schnittstelle erst zur Laufzeit unternommen wird. Ein Problem, das mit diesem Ansatz jedoch entsteht, ist der Erhalt einer Referenz auf die Implementation eines bestimmten Interfaces. Zu diesem Zweck

6 Entwurf einer verteilten Telematikanwendung

existiert in CarTel ein Dienst, der es ermöglicht, Implementierungen von bestimmten Interfaces mit Hilfe eines Namens auffindig zu machen. Dieser Dienst liegt im Package `cartel.service.broker` und ist die zentrale Vergabestelle von Implementationsreferenzen.

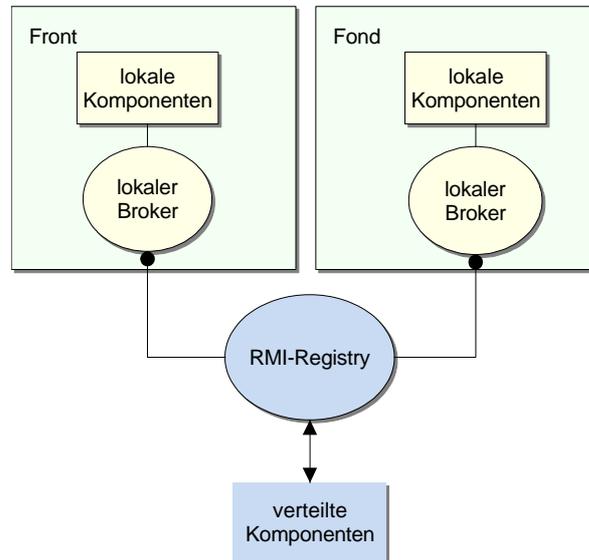


Abbildung 6.6: Broking Topologie

Die Funktionalität, die der CarTel-Broker bietet, entspricht in weiten Teilen der der RMI-Registry. Diese hat jedoch die Einschränkung, dass sie nur Interfaces verwaltet, die `java.rmi.Remote` erweitern. Der CarTel-Broker hingegen verwaltet alle Arten von Interfaces und geht bei der Suche nach Implementierungen folgendermaßen vor: Zunächst versucht er, eine lokal bei ihm angemeldete Implementation für den gewünschten Namen zu liefern. Ist dies nicht möglich, verwendet er die Registry, um ein eventuell entferntes Objekt des gewünschten Typs zu finden. Nur wenn diese beiden Versuche erfolglos waren, wird ein Fehler gemeldet. Damit ist erreicht, dass es für ein Programm vollkommen transparent ist, ob ein Objekt, das es beim Broker angefordert hat, lokal oder entfernt im CarTel-System vorhanden ist. Lokal werden beim Broker nur solche Objekte angemeldet, die zwar über den lokalen Broker verfügbar gemacht werden sollen, jedoch für entfernte Knoten uninteressant sind.

Wie in Abbildung 6.6 dargestellt verfügt jeder Knoten des Systems über einen lokalen Broker. Dieser ist eine nach dem Singleton-Entwurfsmuster (siehe Anhang) entwickelte Klasse, die beim Start eines Knotens instantiiert und mit einer Registry verbunden wird. Alle weiteren Operationen des Brokers ähneln denen einer RMI-Registry, mit dem einzigen Unterschied, dass beim Binding ein beliebiges Objekt verwendet werden kann. `lookup()` hat einen Rückgabewert vom Typ `Object`.

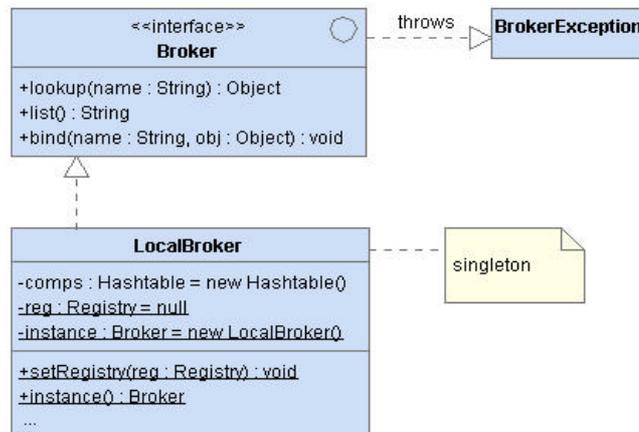


Abbildung 6.7: Das Package `cartel.service.broker`

Das Interface `Broker` hätte eigentlich auch weggelassen werden können, da ein Client des Brokers ohnehin die implementierende Klasse `LocalBroker` kennen muss, um eine `Broker`-Instanz zu erlangen. Zudem ist eine `Broker`-Implementierung nur lokal verfügbar. Dennoch wurde dieses Interface beibehalten, da somit der Weg freigehalten wird, auch den auf einem Knoten lokal vorhandenen `Broker` in der Registry öffentlich zu machen. Damit könnte man beispielsweise erreichen, dass ein Netzwerk von Brokern entsteht, die untereinander kommunizieren können. Aus diesem Grund ist das Package `broker` auch ein Unterpaket von `cartel.service`, obwohl es keinen verteilten Systemservice enthält.

Die Entwicklung dieses Packages ist an dieser Stelle des Prozesses notwendig, um ein Zusammenspiel der einzelnen Komponenten zu ermöglichen. Parallel entwickelte Komponenten können sich von nun an gegenseitig nutzen.

6.9 Zugriffs- und Benutzerverwaltung

Wie oben schon erläutert, verfügt CarTel über eine Zugriffsverwaltung. Ein Benutzer muss sich namentlich nicht beim System anmelden, sofern er keine persönlichen Daten bearbeiten möchte. Vielmehr ist es so, dass ein Knoten sich automatisch bei dessen Start im System meldet und dabei seinen Berechtigungskontext zugewiesen bekommt.

Die Entwicklung dieser Komponente basiert auf der in den Anforderungen spezifizierten Eigenschaft des verteilten Zugriffs auf Komponenten.

Der Zugriff auf Funktionen des Systems ist durch Prioritätswerte, die über den Berechtigungskontext erfragt werden können, geregelt. Zugriffe werden also nicht über die Erteilung exklusiver Rechte erteilt, sondern durch einen differenzierteren Prioritätswert-Mechanismus. Dieser ermöglicht es, dass beispielsweise mehrere Benutzer oder Applikationen gleichzeitig ein Gerät verwenden können.

6 Entwurf einer verteilten Telematikanwendung

Die Applikation, die beispielsweise die Steuerung des CD-Spielers ermöglicht, funktioniert auch dann, wenn es im System mehrere Instanzen dieser Applikation gibt. Wie der Zugriff auf das Gerät geregelt wird, bestimmt die Zugriffssteuerung des CD-Gerätes, in dem sie fein granulierte Prioritätswerte für bestimmte Funktionalitäten zuweist.

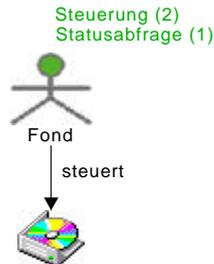


Abbildung 6.8: Steuerung des CD-Spielers von einem Benutzer

Abbildung 6.8 zeigt die Verwendung des CD-Players durch einen Benutzer im Fond des Wagens. In dieser Situation ist dieser Benutzer der einzige, der den CD-Spieler verwenden möchte. Ihm wurden für zwei Funktionen, die der CD-Spieler anbietet, Prioritäten zugewiesen. Für die Steuerung des Spielers besitzt er einen Prioritätswert von 2 und für die Abfrage von Statusanzeigen, wie z.B. Titel der CD oder aktuelle Zeiten, besitzt er Prioritätswert 1. In diesem Augenblick können beide Funktionalitäten uneingeschränkt verwendet werden, da es zu keiner Konfliktsituation kommt. Dies soll durch die grüne Schrift verdeutlicht werden.

Abbildung 6.9 zeigt die Situation, die entsteht, wenn ein weiterer Benutzer (blauer Kopf), den CD-Spieler verwenden möchte. Die Zugriffssteuerung erkennt nun anhand der Prioritätswerte, der einzelnen Benutzer, welche Regelungen zu treffen sind. Da der blaue Benutzer einen größeren Prioritätswert zur Steuerung des Spielers besitzt, wird dem grünen Benutzer diese Funktionalität entzogen (rote Schriftfarbe). Da die Prioritätswerte für die Funktion Statusanzeige gleich sind, entscheidet die Zugriffssteuerung, dass in diesem Fall diese Funktionalität von beiden parallel verwendet werden darf. Für andere Fälle, in denen eine gleichzeitige Verwendung nicht möglich ist, müssen entsprechende Strategien entwickelt werden.

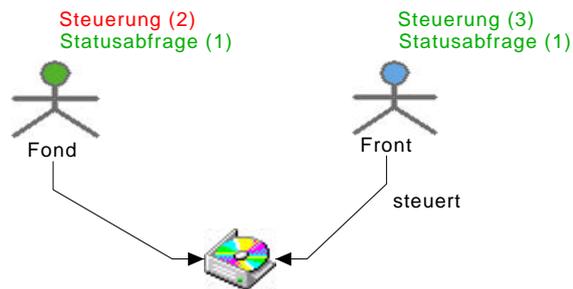


Abbildung 6.9: gleichzeitige Verwendung des CD-Spielers

Durch eine solche Prioritätsverwaltung ist ein sehr differenziertes Verhalten des Systems möglich. So ist nicht immer ein exklusiver Gebrauch einer Funktionalität oder Resource notwendig. Nimmt man beispielsweise die Nutzung einer Kommunikationsplattform zur Verbindung zum Internet, ist es auch denkbar, dass die Bandbreite, die den Benutzern zur Verfügung gestellt wird, anteilig aus ihren Prioritätswerten berechnet wird.

6.9.1 Knoten

Alle Knoten, die im System gemeldet sind, werden in bestimmten Zeitabständen auf ihre Verfügbarkeit hin geprüft. Wird festgestellt, dass ein Knoten nicht mehr ansprechbar ist, weil ein Fehler aufgetreten ist, so muss das System sinnvoll reagieren.

6 Entwurf einer verteilten Telematikanwendung

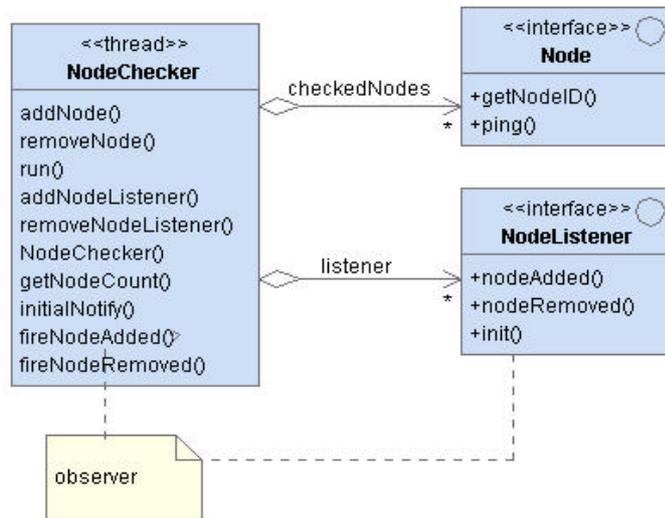


Abbildung 6.10: Überwachungsthread der einzelnen Knoten

Diese Aufgabe wird von einem separaten Thread namens `NodeChecker` übernommen. Er überwacht alle bei ihm über die Methode `addNode()` gemeldeten Knoten. Ein Knoten wird über das `Node`-Interface angesprochen. Dieses besteht zum einen aus der Methode `ping()`, die aufgerufen wird, um zu überprüfen, ob dieser Knoten noch ansprechbar ist. Zum anderen dient `getNodeID()` zur Identifikation des Knotens mittels einer ID. Der `NodeChecker` meldet ein Ereignis, wenn er einen neuen Knoten gemeldet bekommt und wenn er einen Knoten aus seinem Überwachungskontext entfernt. Dies kann explizit durch einen Aufruf von `removeNode()` passieren oder aufgrund eines Fehlers, der beim `ping()`-Aufruf zu einem bestimmten Knoten aufgetreten ist. Interessenten dieser Ereignisse müssen das Interface `NodeListener` implementieren und sich über `addNodeListener()` beim `NodeChecker` als Observer (siehe Anhang) anmelden.

Gestartet wird der `NodeChecker` bei der Initialisierung des `LoginManager`. Jeder Knoten des CarTel-Systems besitzt eine eindeutige ID, die der `LoginManager` bei der Anmeldung über `login()` ermittelt und intern speichert. Zudem meldet er den neu angemeldeten Knoten dem `NodeChecker`, damit dieser ihn in seine Überprüfungen mit einbeziehen kann.

Die Robustheit des Systems liegt bei der Entwicklung eines solchen Überwachungsthreads im Vordergrund. Zudem werden erste Nachrichtensysteme geschaffen, die die Objektkommunikation vorantreiben.

Referenzen: 5.5.4 4+1 Sicht, 5.9 Nachrichten und Ereignisse, 5.10 Threads und Ressourcen

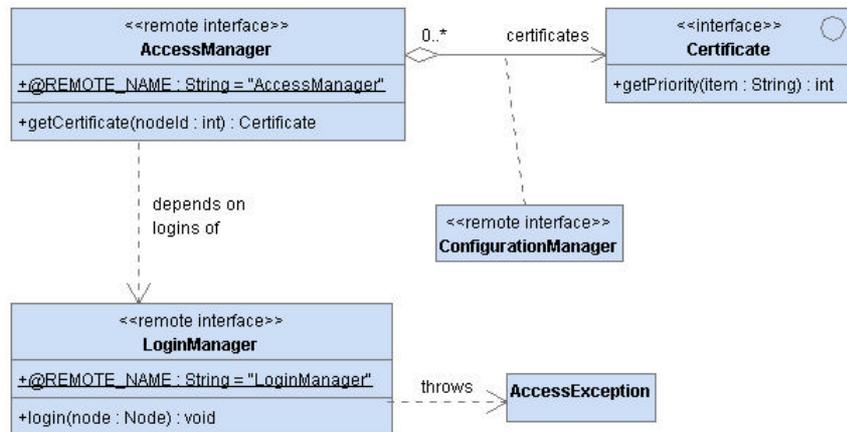


Abbildung 6.11: AccessManager

Der AccessManager ist die zentrale Vergabestelle von Zertifikaten. Jeder Node kann das zu seiner ID passende Zertifikat beim AccessManager abrufen. Ein solches Zertifikat beinhaltet zu jeder Funktion des Systems, die eine Zugriffsregelung durch Prioritätswerte benötigt, den dem Knoten entsprechenden Wert. Dieser kann mit dem Namen der Funktion abgefragt werden. Welche Funktionen es gibt und welche Prioritätswerte sie auf den einzelnen Knoten bekommen, ist in einer Konfiguration gesichert, die vom AccessManager beim ConfigurationManager abgerufen wird.

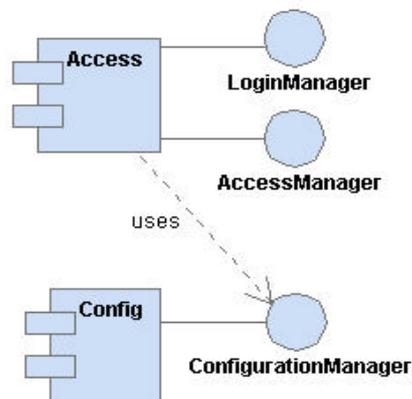


Abbildung 6.12: Komponenten Config und Access

Betrachtet man die Packages `cartel.service.access` und `cartel.service.configuration`, so stellt man fest, dass diese eine autarke Einheit darstellen. Es existieren keinerlei Abhängigkeiten zu anderen Komponenten, was ein hohes Maß an Wiederverwendbarkeit mit sich bringt. Zudem wurden keine implementierungssprachlichen Typen gewählt, so dass die beiden Packages

6 Entwurf einer verteilten Telematikanwendung

in jeder beliebigen objektorientierten Sprache implementiert werden können. Abbildung 6.12 zeigt ein Komponentenmodell, das diesen Umstand näher erläutern soll.

*Phase: Ausarbeitung
Start der 3. Iteration*

Mit Abschluss dieser Iteration, kann nun der zweite Teil des CarTel-Fundaments, das Device-Package, angegangen werden. Die Reihenfolge der abzuarbeitenden Teilprojekte ergibt sich aus dem in Abbildung 6.2 dargestellten Schichtenkonzept. Bevor mit der Entwicklung der Applikationsschicht begonnen werden kann, sollten zumindest die Rahmenbedingungen und wichtigsten Schnittstellen der Geräte-Schicht erarbeitet sein.

Referenzen: Abbildung 5.7

6.10 Geräte

6.10.1 Verteilter Zugriff

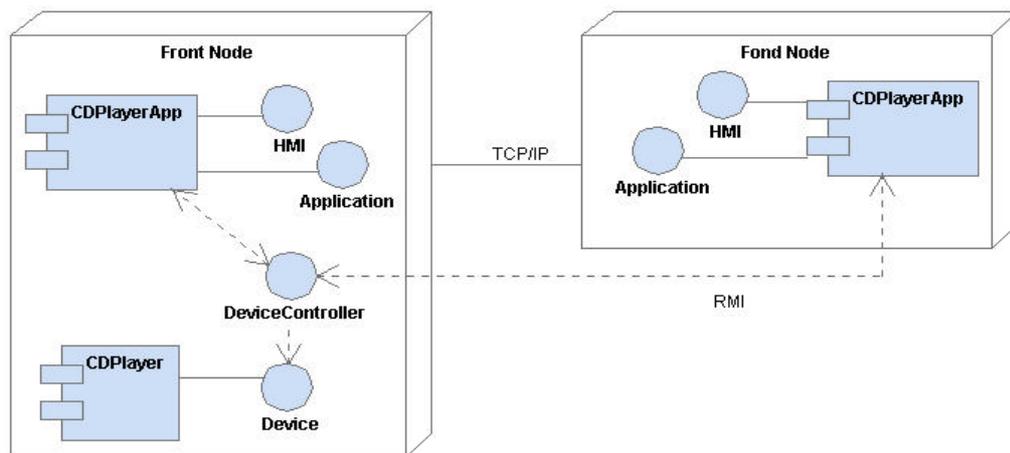


Abbildung 6.13: verteilter Zugriff auf ein Gerät

Der verteilte Zugriff wird dadurch gelöst, dass jedes Gerät (Device) einen Controller besitzt, der für dessen Zugriffsverwaltung zuständig ist (Abbildung 6.13). Alle Applikationen und Komponenten, die ein Gerät verwenden wollen, müssen sich beim DeviceController anmelden, und so ihr Interesse an dem Gerät bekunden. Der Zugriff wird dann über sogenannte CommandSets, die der Controller abhängig vom Berechtigungskontext bereitstellt, ermöglicht. Wie dies im einzelnen geschieht, wird in 6.10.4 näher beleuchtet.

Da es möglich sein soll, verschiedene Geräte mit ein und derselben Schnittstelle anzusprechen, wird im folgenden eine Abstraktion nötig. Diese soll auf alle Geräte passen, so dass ein allgemeines Rahmenwerk entsteht, das dann für spezi-

elle Geräte erweitert werden muss.

Der nächste Schritt im Prozess ist also in diesem Fall maßgeblich davon bestimmt, dass das zu entwickelnde System für andere Entwickler offen gestalten sein soll.

Referenz: 5.7 Modellkonsistenz, 5.7.1 Abstraktion

6.10.2 Entwurf des Rahmenwerks

Das Package `cartel.device` beinhaltet das Rahmenwerk für alle Geräte, die in CarTel benutzt werden sollen. Es soll für alle möglichen Typen von Geräten verwendet und gegebenenfalls den speziellen Bedürfnissen angepasst werden können. Beim Entwurf eines solchen Rahmenwerks ist also auf größtmögliche Generalisierung zu achten, ohne dabei zu unspezifisch zu bleiben. Der Weg, der hier gegangen wird, durchläuft folgende Phasen:

- Entwurf der wichtigsten Schnittstellen für Geräte,
- Implementierung von generischen Funktionalitäten in abstrakten Klassen, die als Schablone oder Spezialisierungsgrundlage dienen sollen,
- Entwurf von spezielleren Geräte-Rahmenwerken (z.B. Player).

Zu erläutern, wie in einem eingebetteten System ein Gerätetreiber entwickelt wird, würde den Rahmen dieser Arbeit sprengen. Deshalb wird im folgenden davon ausgegangen, dass ein Treiber den Zugriff auf ein Gerät mittels eines Interfaces zur Verfügung stellt, dass alle wichtigen Operationen, die das Gerät bedient, enthält. Ein solches Interface erbt grundsätzlich von einem in CarTel enthaltenen allgemeinen Geräte-Interface, das eher als Objekt-Kennzeichnung (Flag-Interface) und nicht als Funktionssammlung dient.

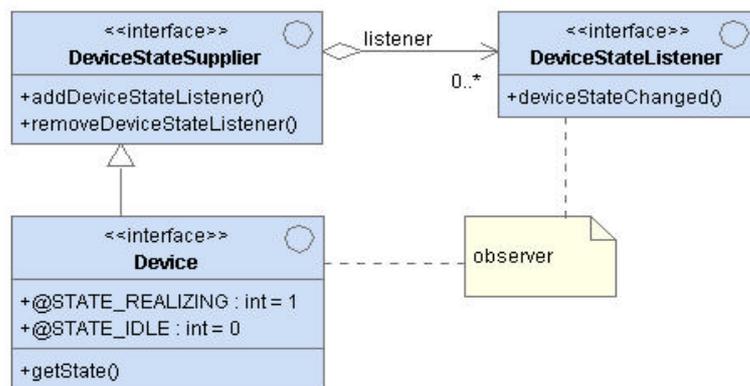


Abbildung 6.14: Device und Observer

Ein Gerät hat grundsätzlich zwei mögliche Gerätestati. Diese kennzeichnen, ob das Gerät gerade mit der Bearbeitung einer in ihrem Interface befindlichen Methoden befindet oder gerade in einem Wartezustand ist. Der Gerätestatus ist

6 Entwurf einer verteilten Telematikanwendung

über die Methode `getState()` zugänglich und gibt eine der beiden Konstanten der Schnittstelle zurück. Zudem ist es möglich, sich als Listener (Observer) bei einem Device anzumelden, und somit automatisch über Statusänderungen informiert zu werden.

Zu beachten ist hierbei, dass ein Gerät keine Schnittstelle zur Verfügung stellt, auf die verteilt zugegriffen werden kann. Alle Schnittstellen in Abbildung 6.14 sind gewöhnliche, lokale Schnittstellen. Die Regelung der verteilten Zugriffe ist nicht Aufgabe eines Device, weil dadurch eine Wiederverwendung einer Geräteschnittstelle in einem System ohne Zugriffssteuerung nur sehr schwer zu realisieren wäre. Analog wäre die Einbindung von bereits implementierten Geräteschnittstellen in das CarTel-System mit vielen Änderungen in der Device-Klasse selbst verbunden. Aus diesem Grund wird der Zugriff auf jedes Device mittels eines DeviceController realisiert.

Device und (Device)Controller gehören immer zusammen. Zugriff auf ein Device kann immer nur über den DeviceController erlangt werden. Dieser ist in der Registry registriert und kann somit über den LocalBroker angefordert werden. Der Zugriff auf den Controller ist nicht beschränkt, was bedeutet, dass alle Clients eine Referenz auf den DeviceController eines bestimmten Gerätes erhalten, unabhängig davon, ob sie dieses auch nutzen dürfen.

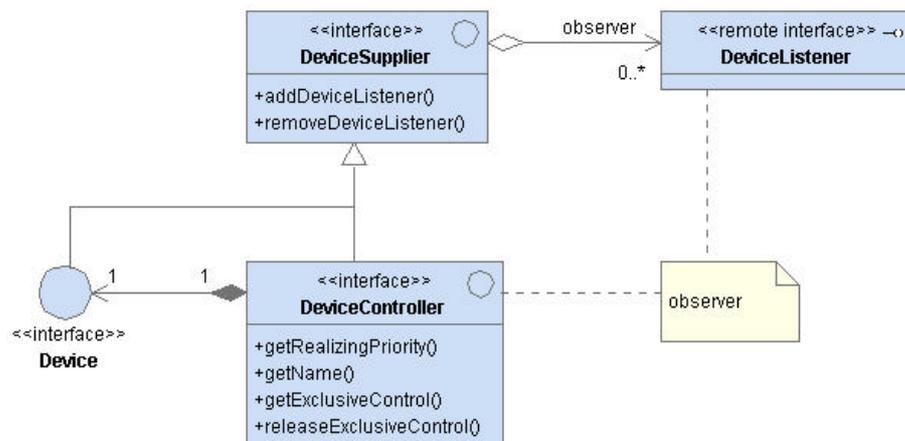


Abbildung 6.15: DeviceController

Die meisten Geräte propagieren Ereignisse bezüglich ihres derzeitigen Zustands. Ein CD-Spieler meldet beispielsweise, wenn die eingelegte CD gewechselt wird, oder er das Abspielen einer CD beendet hat. Solche Ereignisse erfährt man, wenn man sich als `DeviceListener` bei einem `DeviceSupplier` anmeldet. Wie in Abbildung 6.15 zu sehen, implementieren sowohl `Device` als auch `DeviceController` das Interface `DeviceSupplier`. Remote ist jedoch nur die Implementation des `DeviceController` verfügbar, wohingegen das `Device`

weiterhin nur lokal angesprochen werden kann. Eine Anmeldung als `DeviceListener` beim `DeviceController` wird von diesem an das `Device` weitergeleitet (siehe 6.10.5).

Alle Ereignisse, die ein Gerät propagiert, werden an einen oder mehrere beim Gerät gemeldete `DeviceListener` weitergegeben. Die Schnittstelle `DeviceListener` ist nur ein Flag-Interface und dient als Basisklasse für alle speziellen `DeviceListener`. Die Anmeldung als `DeviceListener` erfolgt über den `DeviceController`. Auf die Ereignisbehandlung wird noch einmal genauer in 6.10.5 eingegangen.

Es sei noch erwähnt, dass es nicht möglich ist, beim `DeviceController` eine Referenz auf das `Device` zu erlangen. Eine Manipulation des Gerätes am Controller vorbei ist dadurch ausgeschlossen, was die Verfügbarkeit von Geräten erhöhen soll.

An dieser Stelle muss eine Spezifikation des verteilten Nachrichten- und Ereignisbehandlung folgen, da durch die Einführung von Geräten ein erhöhter Bedarf an Kommunikation zwischen Komponenten besteht.

Referenz: 5.9 Nachrichten und Ereignisse

Bisher wurde nur geklärt, wie man sich als Observer für bestimmte Ereignisse des Gerätes anmelden kann. Zur Verwendung und Steuerung eines Gerätes wurde hingegen das Command-Entwurfsmuster in einer modifizierten Version verwendet. Wie dies realisiert wurde, soll im folgenden Abschnitt geklärt werden.

6.10.3 Commands

Das Command-Muster basiert auf der Idee, bestimmte Funktionalitäten oder Methoden in eine Klasse zu verpacken, um die Möglichkeit zu gewinnen, sie zu parametrisieren.

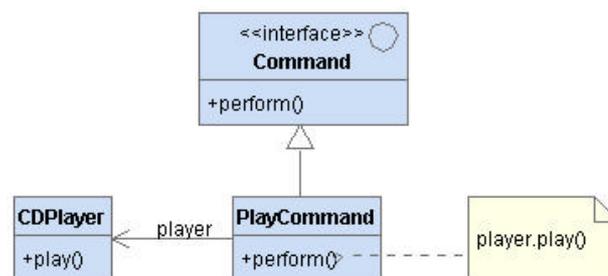


Abbildung 6.16: Beispiel: Command-Muster

Abbildung 6.16 zeigt ein Beispiel für das Command-Muster. Ein `Command` ist ein Interface mit einer Methode zum Aufrufen der dem `Command` entsprechenden Methode des Zielobjektes (`perform()`). `PlayCommand` ist beispielsweise eine `Command`-Implementierung, die beim Aufruf der Methode `perform()` die `play()`-Methode eines `CD-Spielers` aufruft. Dies macht auf den ersten Blick

6 Entwurf einer verteilten Telematikanwendung

kleinen Sinn, da man ja auch gleich `CDPlayer.play()` hätte aufrufen können. Durch Verwendung des Command-Musters ist es aber jetzt möglich dieser Methode Eigenschaften zuzuweisen. So könnte man beispielsweise durch Hinzufügung einer Methode `setEnabled()` zu dem Command eine Sperrung der zugrundeliegenden Funktionalität des CD-Spielers erreichen. Zudem verwendet man das Command-Muster auch oft dazu, einer Funktionalität einen Namen oder ein Icon zu geben, der dann in einem Menu oder auf einem Knopf verwendet werden kann.

Alle diese Eigenschaften führten bei der Entwicklung des CarTel-Systems zu der Idee, mit Hilfe des Command-Musters jegliche Zugriffe auf ein Gerät zu regeln. Doch bevor darauf näher eingegangen wird, soll zunächst einmal eine allgemeine Command-Struktur entwickelt werden. Diese hat noch nichts mit Geräten und Zugriffsregelung zu tun hat, sondern kann als wiederverwendbare Grundlage für andere Systeme, die das Command-Muster verwenden wollen, dienen.

Die Verwendung von Entwurfsmustern wird durch die verstärkte Kollaboration von Komponenten immer wichtiger.

Referenzen: 5.5.1 Nutzen der Architekturbeschreibung

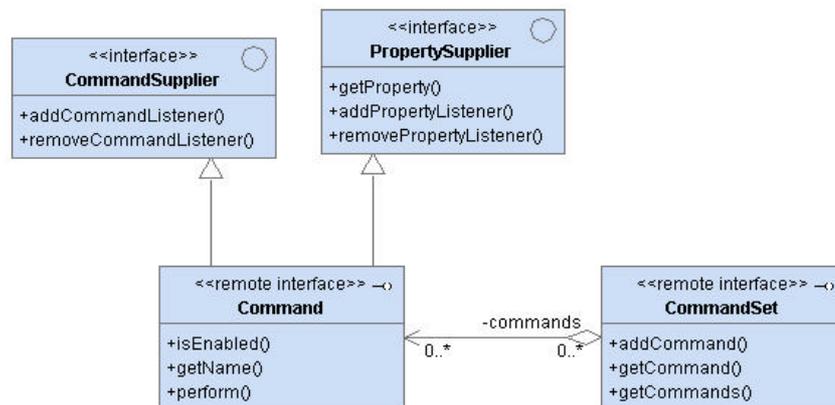


Abbildung 6.17: Command

Zunächst einmal ist es wichtig, dass ein Command mit einer beliebigen Anzahl und Art von Eigenschaften versehen werden kann. So ist es denkbar, dass ein Command noch mit Parametern versorgt werden muss, bevor er ausgeführt werden kann. Dies wird durch die Erweiterung von `PropertySupplier` erreicht, der auch gleichzeitig Änderungen von Eigenschaftswerten an Observer weitergeben kann. Zudem ist es möglich, benachrichtigt zu werden, wenn ein Command ausgeführt wurde, was über die Anmeldung an einem `CommandSupplier` verwirklicht werden kann. Ein `CommandSet` bietet die Möglichkeit, Commands zu sammeln und zu verwalten.

Dieses Modell soll nun um einen Zugriffsmechanismus für Geräte erweitert werden. Der Plan besteht darin, alle Funktionen eines Gerätes in Commands zu

Abbildung 6.18: *CommandSetFactory*

verpacken, die vom `DeviceController` angefordert werden können. Die Zugriffsregelung wird dann `Command`-intern geregelt, in dem sich `Commands` zur Laufzeit je nach Berechtigungskonfiguration aktivieren und deaktivieren. Zu diesem Zweck wird das in Abbildung 6.18 gezeigte Interface `CommandSetFactory` eingeführt. Es dient als Factory (siehe Anhang) für `CommandSet` in Abhängigkeit eines Benutzers (in CarTel einer `NodeID`; siehe Abbildung 6.10). Die `CommandSetFactory` überprüft die Berechtigungen und erzeugt abhängig davon alle `Commands`, die zu dem von ihr bereitgestellten `CommandSet` gehören.

Durch immer konkreter werdende Funktionalität des Systems, in der mehrere Komponenten involviert sind, werden Interaktionsdiagramme wichtiger. Diese können dazu verhelfen, architekturelevante Eigenheiten herauszufiltern.

Referenz: 5.5.2 Einflussfaktoren der Architektur

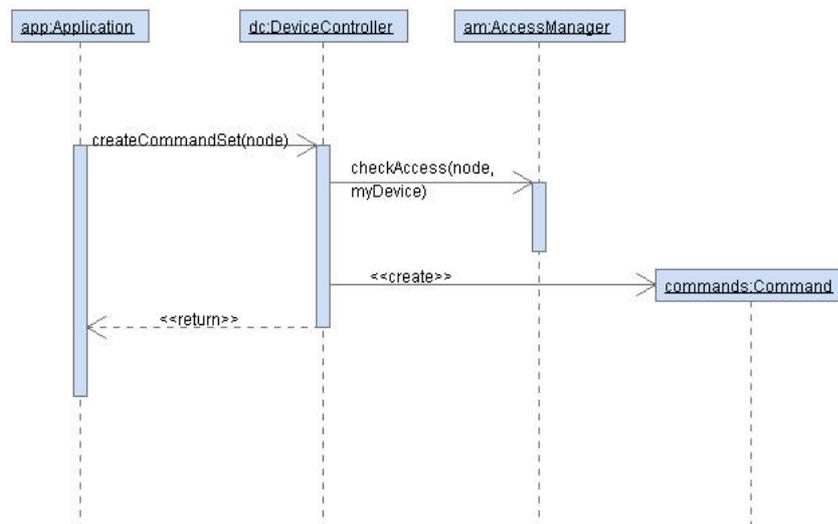
Abbildung 6.19: *Anforderung eines CommandSets*

Abbildung 6.19 zeigt den Ablauf bei der Anforderung der `Commands` eines Gerätes. In diesem Fall möchte eine `Application` Zugriff auf ein Gerät haben und fordert deshalb beim entsprechenden `DeviceController` ein `CommandSet` an. Dabei weist sich die `Application` mit Hilfe der ID des Knotens aus, auf dem sie

6 Entwurf einer verteilten Telematikanwendung

läuft. Der DeviceController erfragt nun beim AccessManager die Zugriffsdaten für diesen Knoten bezüglich des zu ihm gehörende Device. In Abhängigkeit dieser Zugriffsdaten werden nun Commands erzeugt, entsprechend konfiguriert, in ein CommandSet verpackt und zurückgegeben.

Natürlich wurde bei CarTel versucht, den Entwicklungsaufwand bei der Implementierung von Applikationen und Geräten möglichst gering zu halten. Zu diesem Zweck gibt es einige mächtige abstrakte Klassen, die bereits einen großen Teil der Funktionalität implementieren, die für den oben beschriebenen Entwurf notwendig ist.

Referenz: 5.7.2 Präzision, 5.8 Prozessmuster

Eine dieser Klassen ist AbstractDeviceCommand. Sie implementiert das von DeviceListener abgeleitete DeviceConnectable-Interface. Ein DeviceConnectable kann über connect() mit einem Device verbunden werden. AbstractDeviceCommand reagiert standardmäßig auf einen solchen Aufruf, in dem es sich als DeviceStateListener bei dem Device anmeldet und sich das Device als Member sichert. Ein Command ist abgesehen vom DeviceController als einziger befähigt, direkt auf ein Device zuzugreifen. Dies ist deshalb möglich, da alle Commands in dem Prozess instantiiert werden, in dem die Device-Implementation liegt. Damit ist ein AbstractDeviceCommand bereits befähigt auf eine Statusänderung des zugrundeliegenden Gerätes angemessen zu reagieren.

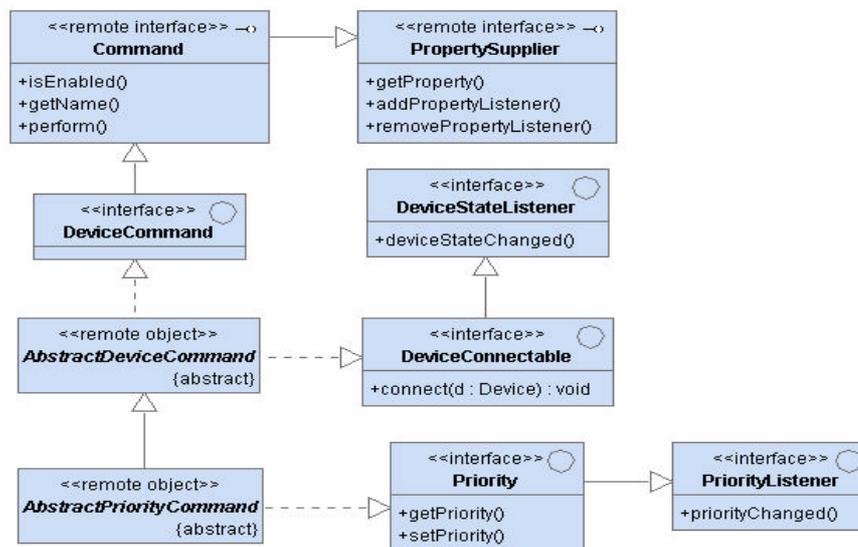


Abbildung 6.20: Abstrakte Standardimplementierungen von Geräte-Commands

Einen Schritt weiter geht die abstrakte Klasse AbstractPriorityCommand. Sie implementiert zusätzlich das Interface Priority. Jeder Command, der Priority implementiert, signalisiert dadurch, dass die ihm zugrundeliegende

Funktionalität mittels Prioritäten geregelt werden soll. Der DeviceController, der einen solchen Command instantiiert, meldet ihm mittels des Priority-Interfaces mit welcher Priorität er läuft. Diese Priorität wirkt sich dann auf das weitere Verhalten des Commands aus. Wie dieser Prioritätswert den Command beeinflusst wird später genauer erklärt.

Wichtig ist, dass jeder Command standardmäßig nur Command und PropertySupplier remote zur Verfügung stellt. Alle anderen Schnittstellen sind lediglich für den Controller sichtbar, der diesen Command instantiiert hat. Dies bewirkt ein Maximum an Sicherheit vor unberechtigten Zugriffen auf Geräte, da ein Command die einzige Zugriffsmöglichkeit auf Gerätefunktionen für einen außenstehenden Client darstellt.

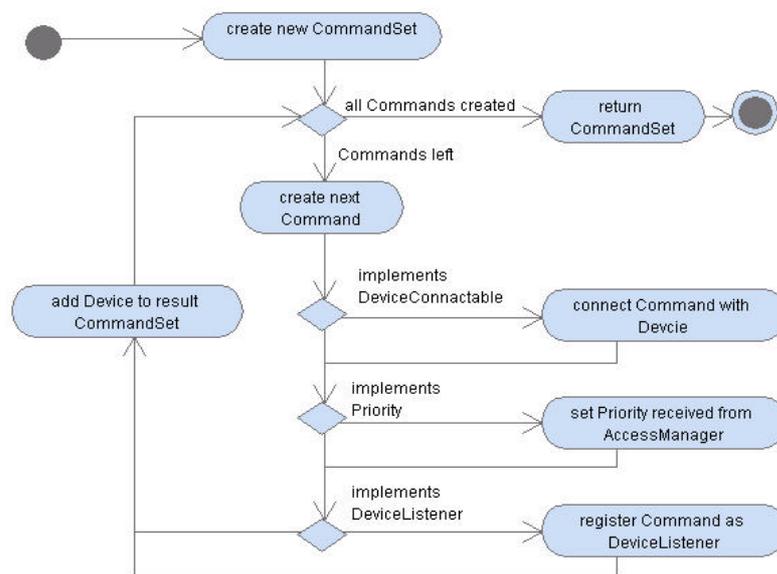


Abbildung 6.21: Aktivitäten des DeviceControllers bei Erstellung eines CommandSets

Die Aktivitäten des DeviceControllers bei der Erstellung des zu seinem Device gehörenden CommandSets zeigt Abbildung 6.21. Nach der Erstellung des CommandSet sind alle darin enthaltenen Commands optimal initialisiert und auf Veränderung in Zusammenhang mit dem Gerät vorbereitet. Bemerkenswert ist hierbei, dass der DeviceController alle Commands mit Prioritäten kennt und diese dadurch über Änderungen der Zugriffsregelungen des Gerätes benachrichtigen kann.

6.10.4 Prioritätsverwaltung

Nun soll geklärt werden, wie die Prioritäten der Commands das Verhalten des Systems beeinflussen.

6 Entwurf einer verteilten Telematikanwendung

Wie bereits erläutert, bekommen alle Commands, die das `Priority`-Interface zur Verfügung stellen, bei ihrer Initialisierung vom Controller einen Prioritätswert zugewiesen. Der Command ist danach darüber informiert, welche Priorität er bezüglich der von ihm symbolisierten Funktionalität besitzt.

Der Controller meldet sich bei jedem Command bei dessen Initialisierung als `CommandListener` an, was zur Folge hat, dass er immer benachrichtigt wird, wenn ein bestimmter Command ausgeführt wurde. Dies bewirkt zusätzlich, dass der Controller jederzeit darüber informiert ist, welcher Command als letzter auf sein Gerät zugegriffen hat. Jedesmal, wenn ein Command ausgeführt wurde, der das `Priority`-Interface implementiert, erfragt der Controller dessen Prioritätswert und sichert ihn intern ab. Falls sich das Gerät in einem Zustand befindet, der die Ausführung eines Kommandos signalisiert (z.B: `Device.STATE_REALIZING`) bedeutet das, dass die derzeitige Priorität unter der die aktuelle Funktion ausgeführt wird der Priorität des letzten Commands mit Priorität entsprechen muss. Dieser Wert kann jederzeit mittels `getRealizingPriority()` beim Controller erfragt werden. Sie beträgt `-1`, wenn momentan keine priorisierte Funktionsausführung vorliegt. Bei einer Veränderung des Prioritätswertes informiert der `DeviceController` nun alle Commands, die Prioritäten unterstützen. Er verwendet dazu die `PriorityListener`-Schnittstelle von der jedes `Priority`-Objekt erbt (siehe Abbildung 6.20). Es liegt nun in der Verantwortung des Commands angemessen auf diese Benachrichtigung zu reagieren.

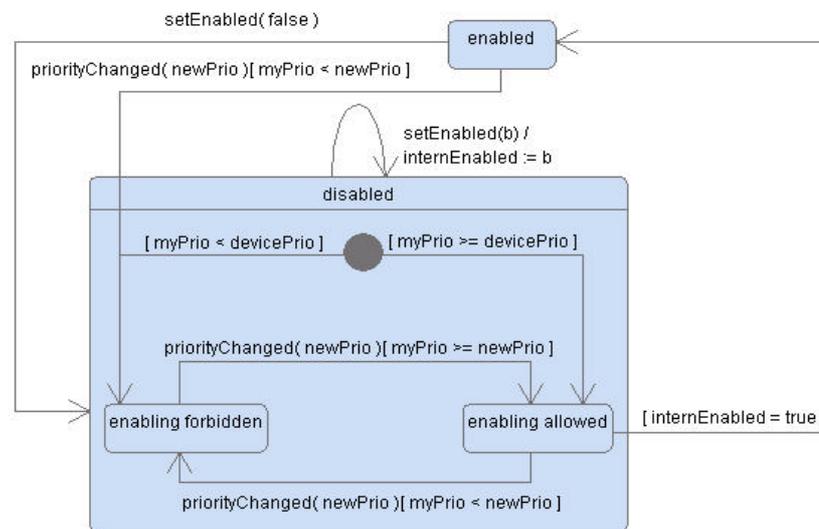


Abbildung 6.22: Verhalten eines prioritätsgesteuerten Commands

Abbildung 6.22 zeigt das Standardverhalten eines prioritätsgesteuerten Commands. Je nach Command-Art befindet sich dieser zu Beginn seines Lebenszyklus im aktivierten oder deaktivierten Zustand. Den aktivierten Zustand verlässt der Command beim Auftreten von einer der beiden folgenden Ereignisse:

1. Aufgrund einer Zustandsänderung des Gerätes deaktiviert er sich selbst (Beispiel: *Play* bei einem bereits spielenden CD-Spieler).
2. Es wird eine Prioritätsänderung des Gerätes gemeldet, wobei die neue Priorität, die des Commands überschreitet.

Der inaktive Zustand hat zwei Unterzustände:

1. Der Command ist nicht berechtigt, den inaktiven Zustand zu verlassen, auch wenn ein Ereignis zum neuerlichen Aktivieren gemeldet wird. Eine solche Meldung bewirkt im deaktivierten Zustand lediglich das Setzen der internen Bedingung `internEnabled` auf den gemeldeten booleschen Wert.
2. Die Berechtigung in den aktivierten Zustand zu wechseln, ist durch einen Prioritätswechsel wieder möglich geworden, da die neue Priorität, nicht größer als die eigene ist. Wird in diesem Zustand die interne Bedingung `internEnabled` erfüllt, kehrt der Command automatisch in den aktivierten Zustand zurück.

All diese Funktionalität ist in den Klassen `AbstractPriorityCommand` enthalten. Weiterhin existieren noch abstrakte Klassen zur Implementierung von eigenen DeviceControllern. Einen Überblick über die gerade erarbeitete Struktur gibt die folgende Abbildung 6.23.

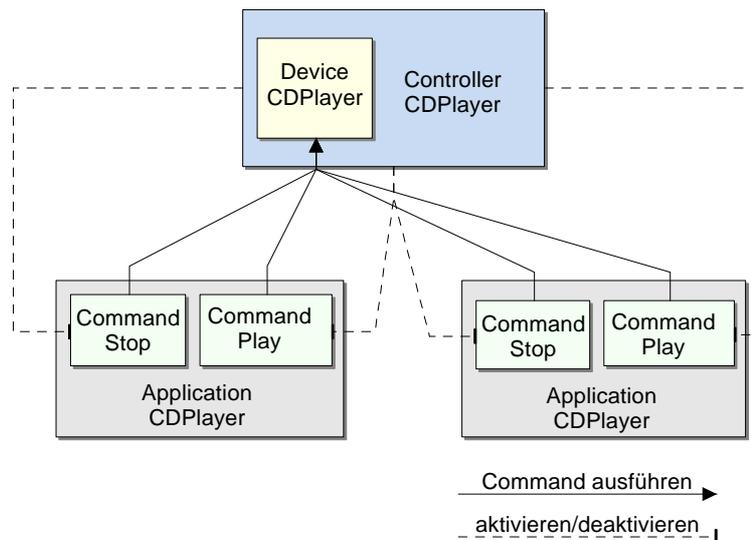


Abbildung 6.23: Überblick: Gerätezugriff

6.10.5 Ereignisse

Es gibt also zwei Möglichkeiten mit einem Gerät zu kommunizieren: Die eine sind die im vorigen Abschnitt behandelten Commands. Diese dienen im allgemeinen dazu, den Status des Gerätes aktiv zu beeinflussen, repräsentieren also die Kommunikation vom Benutzer zum Gerät. Für die entgegengesetzte Richtung, also vom Gerät zum Benutzer, bietet jedes Gerät passende `DeviceListener` an,

die über den Controller beim Device angemeldet werden können. Jedes Objekt, das an Ereignissen oder Nachrichten des Gerätes interessiert ist, implementiert einen oder mehrere dieser vom Gerät bereitgestellte, Callback-Interfaces und registriert sich (Observer-Entwurfsmuster, siehe Anhang).

Das besondere bei der Anmeldung eines DeviceListener beim DeviceController mittels `addDeviceListener()` ist, dass nicht explizit mitgeteilt wird, für welche Benachrichtigungsart des Gerätes sich das anzumeldende Objekt interessiert. Deshalb testet der DeviceController das Objekt auf alle zum Gerät kompatiblen DeviceListener-Schnittstellen und meldet alle implementierten beim Gerät an. Wird keiner der möglichen DeviceListener implementiert wird eine `IncompatibleDeviceListenerException` propagiert. Dieser Weg wurde gewählt, damit nicht für alle möglichen Benachrichtigungsarten eigene Anmeldeverfahren im Controller verankert werden müssen. Zudem wird der Kommunikationsaufwand beim Anmelden eines Listeners gering gehalten, da ein einziger Aufruf von `addDeviceListener()` pro Objekt genügt.

Die Anmeldung als DeviceListener unterliegt keinerlei Zugriffsbeschränkungen, da sie den Zustand eines Gerätes nicht beeinflussen. Durch die Trennung in Commands und Listener ist es beispielsweise bei einem CD-Spieler möglich, die Informationen über die aktuelle CD zu erfahren, auch wenn der Spieler selbst momentan von einem anderen Benutzer exklusiv gesteuert wird.

Tritt bei der Benachrichtigung eines Listeners ein Fehler oder eine Zeitüberschreitung auf, so wird dieser für diesen Ereignistyp aus der Liste der zu benachrichtigenden Objekte herausgenommen. Dies sichert die Robustheit eines Gerätes gegenüber Fehlern in Client-Implementierungen.

Die Spezifikation des Verhaltens bezüglich Nachrichten und Ereignissen verhilft nun dazu, weitere Anforderungen in das System mit einzubeziehen. Zudem wird durch die obige Entscheidung die Robustheit gegenüber Fehlern erhöht.

Referenzen: 5.9 Nachrichten und Ereignisse, 5.5.4 4+1 Sicht

6.10.6 Exklusive Kontrolle

CarTel bietet die Möglichkeit, ein Gerät exklusiv zu kontrollieren. Dies findet beispielsweise dann Anwendung, wenn der Fahrer alleine bestimmen möchte, welche Musik der CD-Spieler spielen soll.

Die Entwicklung wird weiter vorangetrieben durch den Anwendungsfall „Exklusive Kontrolle“.

Referenz: 5.4 Use Case Basierung

Zu diesem Zweck bietet jeder DeviceController die Methode `getExclusiveControl()` zum Erlangen der exklusiven Kontrolle über das Gerät. Voraussetzung dafür ist, dass das aufrufende Objekt einen Prioritätswert besitzt, der nicht geringer als die derzeitige Ausführungspriorität ist. Ist diese Bedingung erfüllt, wird bei der Standardimplementierung des DeviceController dieser Wert auf den des Aufrufenden gesetzt und erst dann

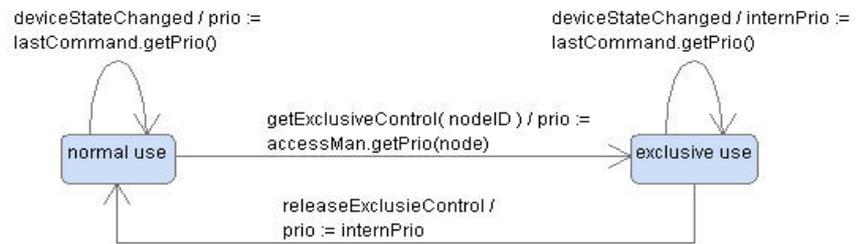


Abbildung 6.24: Verhalten des Controllers bezüglich exklusiver Benutzung

wieder zurückgesetzt, wenn die exklusive Kontrolle mittels `releaseExclusiveControl()` wieder abgegeben wird.

Wie man in Abbildung 6.24 erkennen kann, ist während der gesamten Zeit einer exklusiven Kontrolle die Ausführungspriorität konstant. Die Ereignisse, die normalerweise einen Prioritätswechsel herbeiführen würden, werden protokolliert, damit bei Freigabe des Gerätes die Ausführungspriorität entsprechend gesetzt werden kann.

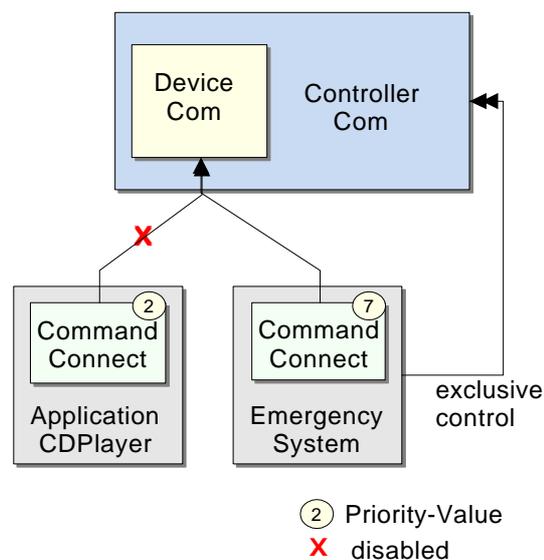


Abbildung 6.25: exklusive Kontrolle der Kommunikationsplattform

Die Exklusivität der über `getExclusiveControl()` erlangten Kontrolle bezieht sich nur auf Anwender mit niedrigerer Priorität, als der des Inhabers der Kontrolle. Alle Komponenten mit höherer Priorität bleiben davon unberührt. Der exklusive Benutzer eines Gerätes erhält einen Code, mit dem er das Gerät wieder freigeben kann. Die Freigabe ist also nur möglich, wenn man entweder im Besitz

6 Entwurf einer verteilten Telematikanwendung

eines solchen Codes ist, oder eine noch höhere Priorität als der Inhaber der exklusiven Kontrolle hat. Wird die exklusive Kontrolle von einem höher priorisierten Objekt weggenommen, wird ein neuer Freigabecode generiert, der alte verfällt.

Start der 4. Iteration

An diesem Punkt des Entwicklungsprozesses angelangt, muss noch eine weitere Präzisierung des Gerätepaketes stattfinden. Ansonsten wäre die Entwicklung eines wirklich funktionierenden Prototypen (CD-Spieler) mit zu großem Aufwand verbunden.

Referenz: 5.7.2 Präzision, 5.6.5 Prototyp

6.10.7 Player

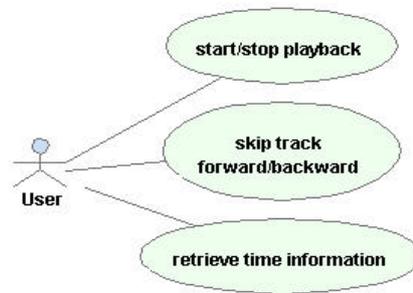


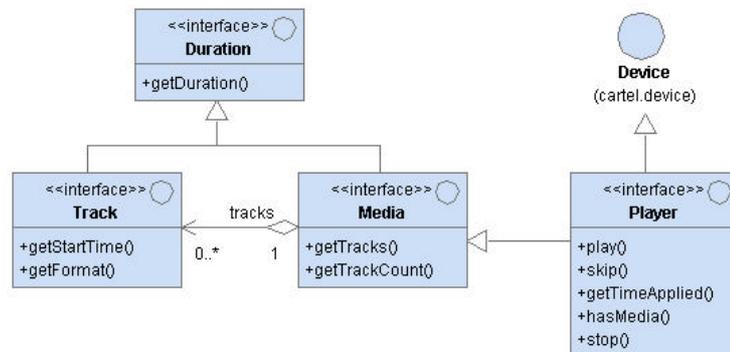
Abbildung 6.26: UseCases eines CarTel-Players

CarTel enthält eine Bibliothek, die Entwickler dabei unterstützt, ein Player-Gerät in das System zu integrieren. Unter einem Player wird hier ein beliebiges Gerät oder Software zum Abspielen von Medien verstanden.

An dieser Stelle treten zum ersten mal echte Use Cases auf, in denen der Anwender mit dem System interagiert. Im Unterschied zu den vorangegangenen Phasen wird nun eine Architektur-Schicht erreicht in der mehr und mehr Use Cases die Basis der Weiterentwicklung bilden.

Referenz: 5.4 Use Case Basierung

Zwei Funktionen, die jeder Player bietet, sind der Start und die Beendigung des Abspielens eines Mediums und das Anspringen einer anderen Spur (Track) eines mehrspurigen Mediums. Zudem sollte es möglich sein, Informationen über die aktuelle Position des Players innerhalb des gerade gewählten Mediums zu erlangen (Abbildung 6.26).

Abbildung 6.27: Package *cartel.device.player* (Auszug)

Die Steuerung eines Players folgt nun exakt dem in 6.10.3 und 6.10.5 erarbeiteten Command/Event-Muster. Alle Commands eines Players erweitern die in *cartel.device* verankerte Standardimplementierung eines priorisierten Befehls *AbstractPriorityCommand*.

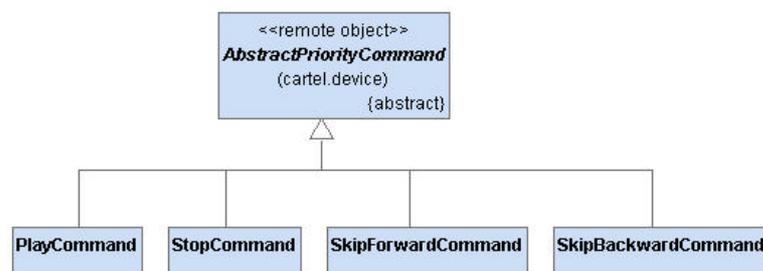


Abbildung 6.28: Commands eines Players

Die in Abbildung 6.28 dargestellten Commands genügen, um den Player zu steuern. Das *CommandSet*, das der *DeviceController* eines Players liefert, beinhaltet also mindestens jeweils eine Instanz dieser Commands. Da in diesen Command-Klassen ein Name zur Identifikation des Commands statisch codiert ist, ist es einfach, diese im *CommandSet* mittels *getCommand(name)* zu finden, ohne dabei eine Typenprüfung vornehmen zu müssen.

6 Entwurf einer verteilten Telematikanwendung

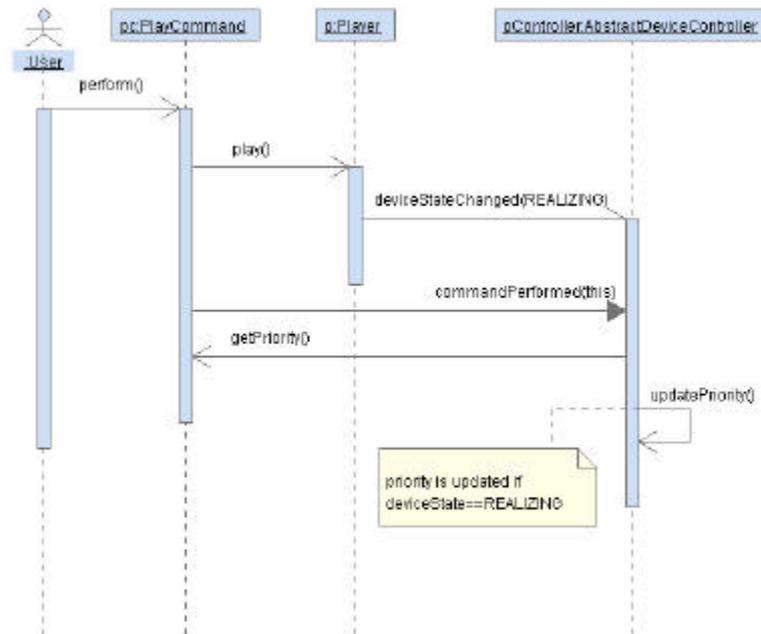


Abbildung 6.29: Ausführung des PlayCommand

Abbildung 6.29 soll dazu verhelfen, den Zugriffsmechanismus am konkreten Beispiel des CD-Spielers zu erläutern.

Der Aufruf von `updatePriority()` benachrichtigt alle beim DeviceController angemeldeten PriorityListener davon, dass eine Prioritätsänderung stattgefunden hat. Die benachrichtigten Objekt können nun darauf entsprechend reagieren. Ein PriorityCommand mit geringerer Priorität beispielsweise deaktiviert sich so lange, bis wieder eine niedrigere Priorität gemeldet wird.

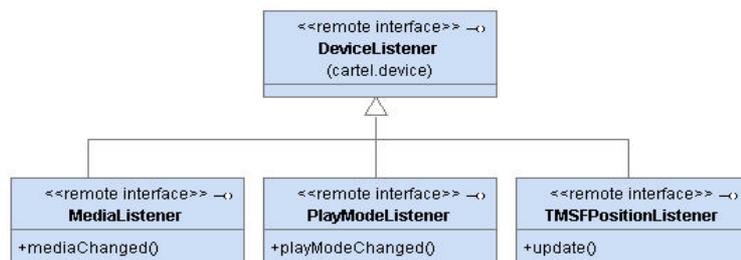


Abbildung 6.30: Zu einem Player kompatible DeviceListener

Alle von einem Player unterstützten DeviceListener sind in Abbildung 6.30 zusammengestellt. Über `addDeviceListener()` des Player-Controllers angemeldeten MediaListener werden benachrichtigt, wenn sich das gewählte

Medium des Players geändert hat. Ein `PlayModeListener` erhält eine Benachrichtigung, wenn ein Player seinen Modus gewechselt hat. Mögliche Werte hierfür sind Konstanten, die die Modi *Spielend*, *Gestoppt* und *Pause* repräsentieren.

Ein `TMSFPositionListener` schließlich erhält in einem bestimmten Intervall eine Benachrichtigung über die aktuelle Position des Players im Format *Track:Minute:Sekunde:Frame* relativ zur Gesamtzeit des Mediums. Ein Frame entspricht laut Red-Book Spezifikation [EE498] 1/75 Sekunde.

Die Klassen dieses Packages wurden zu großen Teilen aus den Use Cases abgeleitet, wie es in **5.4.3 Auffinden von Klassen anhand von Use Cases** beschrieben wird.

Ein Anwendungsbeispiel für einen `PlayModeListener` ist der `PlayCommand`. Wenn diesem gemeldet wird, dass der Player, den er ansteuert, in den Modus *spielend* gewechselt ist, deaktiviert sich der Command solange er keinen anderen Modus gemeldet bekommt.

Im folgenden eine Aufstellung der von einem Player generierten Ereignisse (vgl. 5.9.2):

<i>Ereignis</i>	<i>Beschreibung</i>	<i>Empfänger</i>	<i>Muster</i>	<i>Zeitregeln</i>
mediaChanged	Das aktuell im Player befindliche Medium wurde entfernt oder gewechselt.	alle gemeldeten <code>MediaListener</code>	episodisch	
playModeChanged	Der Spielmodus des Player hat gewechselt.	alle gemeldeten <code>PlayModeListener</code>	episodisch	
update	Die Position des Players hat gewechselt	alle gemeldeten <code>TMSFPositionListener</code>	periodisch	alle 300 ms

Tabelle 6.1 - Ereignisliste eines Players

Eine Implementierung eines CD-Spielers auf der Basis des `device.player` Packages wird in 6.14 noch genauer betrachtet.

*Meilenstein 2: Architektur
nächste Phase: Konstruktion
Start der 5. Iteration*

An dieser Stelle ist der Entwurf der Grund-Architektur abgeschlossen. Die untersten Schichten des Systems sind entworfen und können nun als Grundlage für die weitere Entwicklung dienen.

Es folgt zunächst ein gründliches Austesten der bestehenden Komponentenschnittstellen bezüglich der gestellten Anforderungen. Sind diese erfolgreich

6 Entwurf einer verteilten Telematikanwendung

absolviert, kann die Konstruktion der höher gelegenen Schichten beginnen.
Referenzen: 5.6.3 Iterationen, 5.7.3 Kompatible Komponenten

6.11 Applikationen

CarTel besitzt die Fähigkeit, eine beliebige Anzahl von Applikationen zu verwalten. Applikationen müssen nicht unbedingt eine Benutzerschnittstelle haben, sie können auch lediglich zur Bereitstellung bestimmter Funktionen für andere Applikationen dienen. Applikationen laufen im Kontext des Knotens, auf dem sie gestartet wurden (siehe Anforderungen 6.3.6). Sie stellen im Gegensatz zu Geräten keine verteilt ansprechbaren Objekte dar, wodurch die Kommunikation zwischen ihnen nur knotenintern möglich ist.

Um Ressourcen zu sparen, wird eine Applikationen erst dann instantiiert, wenn der Benutzer durch eine Auswahl signalisiert, dass er sie benutzen möchte. Eine zentrale Verwaltungseinheit verfügt über alle Informationen der installierten Applikationen.

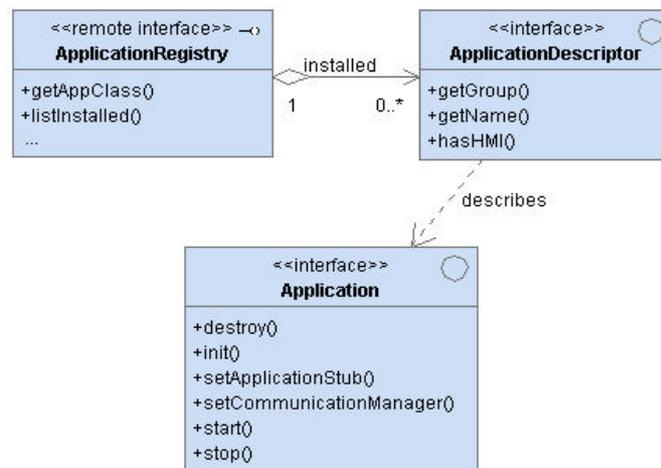


Abbildung 6.31: Applikationsverwaltung

Die ApplicationRegistry ist ein remote ansprechbares Objekt, in dem alle ApplicationDescriptor-Objekte registriert sind. Ein ApplicationDescriptor dient dazu, Informationen über eine Applikation zu liefern, ohne dass dazu die Applikation selbst instantiiert werden muss. In dieser Version von CarTel gehört zu diesen Informationen der Name der Applikation selbst und der Name der Programmgruppe der Applikation. Zudem gibt ein ApplicationDescriptor über hasHMI() Auskunft darüber, ob die von ihm beschriebene Applikation eine grafische Benutzerschnittstelle bietet. Die Informationen über installierte Applikationen werden zentral in einer Configuration gehalten.

Jede Applikation im CarTel System muss die Schnittstelle `Application` implementieren. Eine Applikation kommuniziert mit anderen Objekten über den `ApplicationStub`. Dieser wird der Applikation mittels `setApplicationStub()` zugewiesen. Normalerweise würde man eine solche Zuweisung im Konstruktor einer allgemeinen Basisklasse ansiedeln. `Application` ist jedoch als Interface entworfen worden, was die Initialisierung etwas erschwert. Dieser Entwurf macht aber Sinn, da damit eine eventuelle spätere Erweiterung von Applikationen für den verteilten Zugriff wesentlich einfacher ist. Die eigentliche Initialisierung der Applikation sollte diese in `init()` vornehmen. Beim Aufruf dieser Methode ist gewährleistet, dass die Aufrufe von `setApplicationStub()` und `setCommunicationManager()` bereits erfolgt sind. Ein Aufruf von `start()` signalisiert der Applikation, dass sie gestartet wird, `stop()`, dass sie momentan nicht verwendet wird. Dies gibt der Applikation die Möglichkeit sich zwischen den Aufrufen von `stop()` und `start()` in einen ressourcenschonenden Wartezustand zu versetzen. Mittels `destroy()` wird signalisiert, dass sie alle von ihr verwendeten Ressourcen freigeben kann, da sie von nun an nicht mehr gestartet wird. Diese Methode wäre in Java aufgrund der automatischen Garbage-Collection nicht notwendig gewesen. In Hinblick auf andere Sprachen, wurde sie dennoch in die Schnittstelle aufgenommen.



Abbildung 6.32: Schnittstelle zur Applikationskommunikation

Die Implementation von `InterAppCommunicationManager` ist ein auf dem Knoten lokal verfügbares Objekt. Über ihn ist es für eine Applikation möglich, eine Referenz auf eine Instanz einer beliebigen anderen installierten Applikation zu erlangen, um mit dieser zu kommunizieren. Wenn keine solche Applikation vorhanden ist, wird vom `InterAppCommunicationManager` eine solche beim Aufruf von `connect()` instantiiert.

Der `ApplicationStub` dient hingegen zur Vereinfachung der Kommunikation mit entfernten Objekten. Über ihren `ApplicationStub` kann sich eine Applikation Zugriff auf ihre Konfiguration und beliebige Strings verschaffen, ohne Referenzen auf `Configuration-` oder `PropertyManager` haben zu müssen. Dies erspart den umständlichen Weg über den Broker des Knotens. Zudem bietet der `ApplicationStub` alle Funktionen der `ApplicationRegistry`, wodurch Informationen über die anderen verfügbaren Applikationen leicht zugänglich sind.

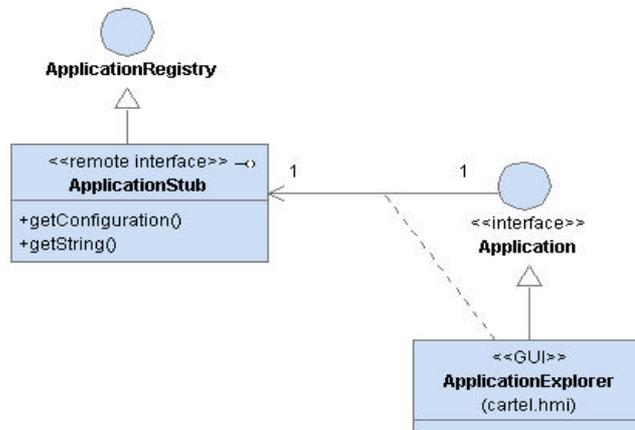


Abbildung 6.33: ApplicationStub

6.11.1 ApplicationExplorer

Die Verbindung zwischen Application und ApplicationStub bzw. InterAppCommunicationManager wird vom ApplicationExplorer des Knotens vorgenommen. Dieser stammt zwar aus dem Package hmi (siehe 6.12), soll aber wegen seiner engen Zusammenarbeit mit Applikationen schon hier erwähnt werden.

Er ist selbst eine Application, die auf jedem Knoten vorhanden sein muss. Beim Start von CarTel wird nach Abschluss aller Initialisierungen eine Instanz des ApplicationExplorer erzeugt, initialisiert und angezeigt. Der ApplicationExplorer hat Kenntnis über alle verfügbaren Applikationen und gewährt über seine Benutzerschnittstelle Zugang zu diesen.

Der Stereotyp *GUI* zeigt an, dass es sich hierbei um eine grafische Komponente handelt.

*Phase: Konstruktion
Start der 6. Iteration*

Hier bietet sich ein weiterer Schnitt im Prozess an. Es sind nun die drei untersten Schichten entworfen. Das Zusammenspiel der Komponenten kann nun ausgiebig getestet werden, bevor die Hinzunahme des hmi-Packages den Prototypen funktional vervollständigt.

6.12 Grafische Benutzungsschnittstelle (Human Machine Interface –HMI)

Die grafische Benutzungsschnittstelle dient zur Interaktion des Benutzers mit dem System. Auf die Entwicklung einer vollständigen Grafikbibliothek wird an dieser Stelle verzichtet, da dies nicht Gegenstand der Arbeit ist. Statt dessen wird die von Java bereitgestellte Swing-Bibliothek [JFC-00] zur Bildschirmdarstellung

6.12 Grafische Benutzungsschnittstelle (Human Machine Interface –HMI)

verwendet. Alle bisher behandelten Packages besitzen keinerlei Abhängigkeiten zum nun behandelten hmi-Package. Dadurch ist gewährleistet, dass die Entwicklung einer HMI-Bibliothek keinerlei Änderungen in tiefer liegenden Schichten (siehe Abbildung 6.40) mit sich zieht.

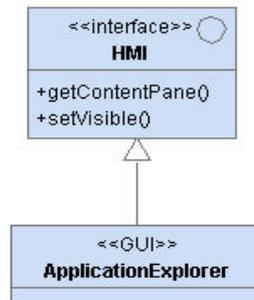


Abbildung 6.34: HMI

Abbildung 6.34 zeigt die Schnittstelle HMI. Diese muss von Applikationen implementiert werden, die ein HMI bereitstellen möchten. Alle diese Applikationen können dann vom `ApplicationExplorer` dargestellt werden. Mittels `getContentPane()` erhält der `ApplicationExplorer` die grafische Repräsentation der Applikation (in dieser Version eine `java.awt.Component`) und bettet sie in seine Anzeige ein.

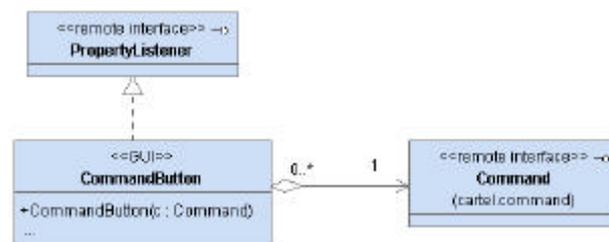


Abbildung 6.35: CommandButton

Als Beispiel für die sinnvolle Ausnutzung des in 6.10.3 eingeführten Command-Konzepts existiert in hmi die grafische Komponente `CommandButton`. Diese stellt einen Knopf dar, bei dessen Betätigung der zu diesem Knopf gehörende `Command` ausgeführt wird. Wenn der `Command` das Attribut `PROP_DISPLAYNAME` über `getProperty()` anbietet, so wird der zurückgegebene Wert als Aufschrift für den Knopf verwendet. Ändern sich Eigenschaften des `Command`, so erhält der `CommandButton` über das `PropertyListener`-Interface Meldung darüber und kann darauf reagieren. Wird z.B. der zugrundeliegende `Command` deaktiviert, so deaktiviert sich auch der Knopf.

6 Entwurf einer verteilten Telematikanwendung

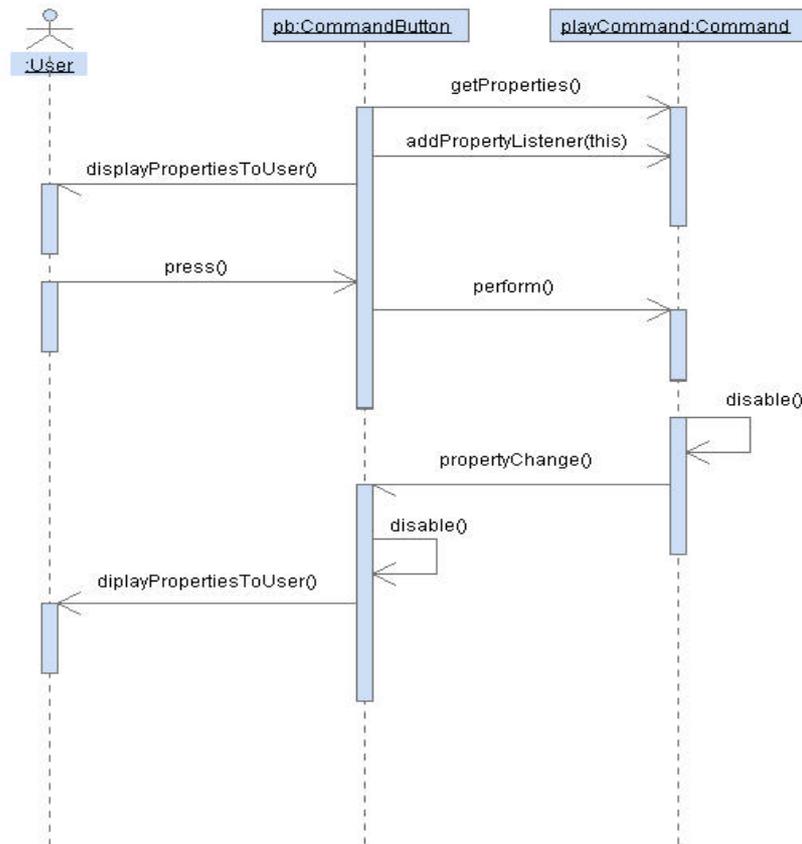


Abbildung 6.36: Nachrichten zwischen CommandButton und Command

Abbildung 6.36 zeigt die Initialisierung und Ereignisbehandlung des CommandButton.

Ein Beispiel für den Einsatz von CommandButton findet sich in den Steuerknöpfen der CD-Spieler Applikation in 6.14.

Der Prozess berücksichtigt bei der hmi-Schicht, dass es sich um den Entwurf eines Prototypen handelt und deshalb auf die Entwicklung einer umfangreichen Grafikbibliothek verzichtet werden muss.

So betrachtet, schafft der Prototyp nicht den ganzen vertikalen Schnitt durch alle Schichten aus Abbildung 5.7. Dies ist aber aus oben genannten Gründen für einen solchen Prototypen auch nicht zwingend notwendig.

Auch die Testarchitektur des nächsten Abschnitts weist diesen Prototypcharakter auf.

Referenz: 5.6.5 Prototyp

6.13 Monitoring und Test

CarTel bietet die Möglichkeit, Komponenten des Systems zu überwachen. Diese Überwachungsfunktion soll nur ein Beispiel dafür sein, dass eine komponentenbasierte Systemarchitektur den Einbau von Monitor- und Testfunktionen in ein System stark vereinfachen kann. Eine robuste Testarchitektur hätte den Zeitrahmen zur Entwicklung von CarTel sicherlich gesprengt. Es wurde deshalb nur ein simples Test-Interface entwickelt, über das eine kleine Applikation Statusdaten von Komponenten abfragt.

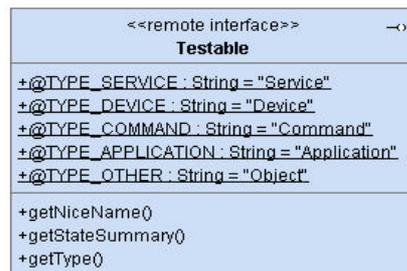


Abbildung 6.37: Test-Schnittstelle

Ein Objekt, das überwachbar sein soll, sollte die Testable-Schnittstelle zur Verfügung stellen und sich über den Broker verfügbar machen. Testable Objekte geben über `getStateSummary()` Auskunft über ihren derzeitigen Zustand. Dieser wird als einfacher String zurückgegeben. Zudem ordnet es sich noch einem der über die öffentlichen Attribute (`TYPE_SERVICE`, `TYPE_DEVICE`, etc.) zugänglichen Typen zu.

Remote Name	Typ	Status
CDDB	Application	running; server=us.cddb.com/~cdd...
TestCentral	Application	running
ApplicationExplorer	Application	initialized: CDPlayer, TestCentral; c...
cartel.device.net.Networkl...	Device	prio: -1; state = idle
ApplicationRegistry	Object	registered: 3
PropertyManager	Service	running; prefix=res
LoginManager	Service	logged in Nodes:0
AccessManager	Service	running.
Configuration Manager	Service	running; path=config
cdplayer.device.CDPlayer...	Device	prio: 5; state = realizing

Abbildung 6.38: Applikation zum Testen von Komponenten

Abbildung 6.38 zeigt einen Bildschirmabdruck der Applikation, die das Testable-Interface verwendet. Alle Komponenten, die über den Broker verfügbar sind und Testable implementieren, werden in einer Tabelle zusammen mit den durch Testable erfragten Informationen dargestellt. Die Applikation führt

diese Befragung in einem bestimmten Intervall durch und pflegt entsprechende Änderungen in die Tabelle ein.

*Meilenstein 3: Funktionale Vollständigkeit
nächste Phase: Auslieferung
Start der 7. Iteration*

Mit Abschluss der Entwicklung aller Schichten ist die funktionale Vollständigkeit des Systems erreicht. Um einen lauffähigen Prototypen zu erhalten, müssen jedoch noch plattformabhängige Implementierungen der Schnittstellen erstellt werden.

Referenz: 5.6.3 Iterationen

6.14 CD-Spieler

Damit ist die Besprechung des Kerns des CarTel-Systems abgeschlossen. Er bildet die Grundlage für alle möglichen zu entwickelnden Applikationen, Dienste und Geräte. Im folgenden wird beispielhaft eine CD-Spieler Applikation entwickelt, die auf ein ebenfalls zu entwickelndes CD-Spieler Gerät zugreift. Auch die Kommunikation mit anderen Applikationen wird behandelt. Die CD-Spieler Applikation kommuniziert mit einer CDDB-Applikation, die es ermöglicht, CD-Daten mittels der Kommunikationsplattform zu ermitteln.

6.14.1 Device

Da CarTel für Windows NT entwickelt wird, bietet es sich an, ein in den Rechner eingebautes CD-ROM-Laufwerk als CD-Device ins System einzubinden. Die Schwierigkeit hierbei besteht jedoch darin, dass Java keinen direkten Zugriff auf die Audio-Fähigkeiten des CD-ROM zulässt. Aus diesem Grund wird das CD-Device mit Hilfe der Spracherweiterungen angesprochen, die die Microsoft VM zur Verfügung stellt. Da diese auch RMI unterstützt, ist es möglich, lediglich zum Ansprechen der Hardware auf Spracherweiterungen zurückzugreifen und alle restlichen Komponenten in *purem* Java zu implementieren.

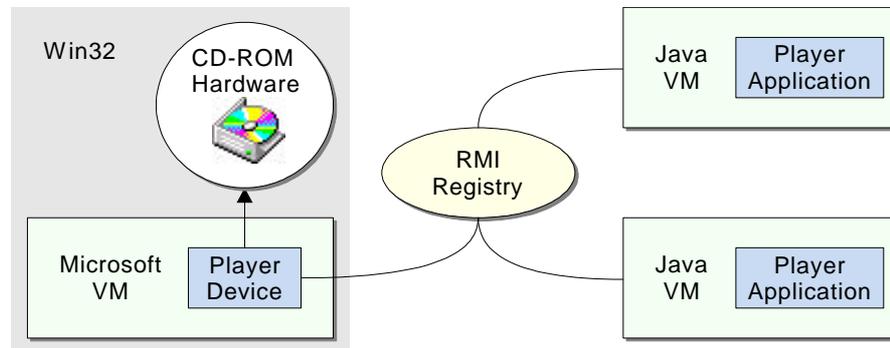


Abbildung 6.39: Java-Zugriff auf plattformabhängige Hardware

Die Implementierung der Schnittstelle `Player` aus `device.player` besteht lediglich aus einem Wrapping der windows-eigenen MCI-Bibliothek. Der `PlayerController` ist eine Spezialisierung der Klasse `device.AbstractDeviceController`. Als Commands werden die von `device.player` bereitgestellten und in Abbildung 6.28 dargestellten Standardimplementierungen verwendet.

Hier findet bereits die konkrete Umsetzung der Konzepte auf bestimmte Plattformen statt.

6.14.2 Applikation

Die Implementierung der Applikation besteht im wesentlichen aus der HMI-Programmierung. Es werden für alle Commands `CommandButtons` verwendet und ein weiteres Control für den exklusiven Zugriff entwickelt. Durch die Implementierung des `TMSFPositionListener` erhält die Applikation Informationen über die Position des CD-Spielers und kann diese als Text darstellen.

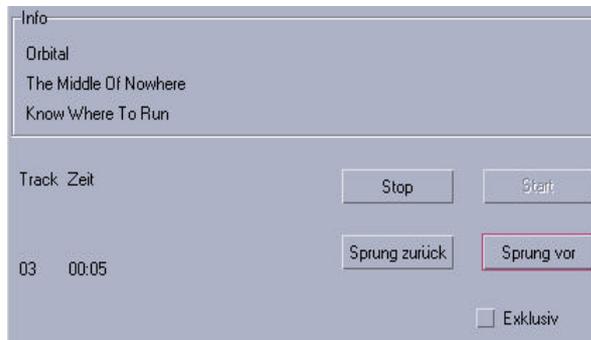


Abbildung 6.40: CD-Spieler Applikation

Die Informationen zur eingelegten CD erhält die CD-Spieler Applikation über die CDDDB-Applikation, die lediglich eine Methode zum Abruf von CDDDB-Daten zur Verfügung stellt und deshalb nicht weiter betrachtet wird.

Auf Details zur Implementierung der einzelnen Schnittstellen wird an dieser Stelle verzichtet. Es sei jedoch darauf hingewiesen, dass die Entwicklung der Applikation durch die bereits vorgegebene Infrastruktur des CarTel-System sehr einfach und schnell realisierbar ist. Die Hauptarbeit bei der Entwicklung des Device ist das Wrapping der Bibliotheksschnittstellen. Um die Zugriffssteuerung kümmern sich vollständig die abstrakten Klassen aus den `device` und `device.player` Packages.

4. Meilenstein: Auslieferung Abschluss des Prototyp-Projektes

Die Entwicklung des Prototypen ist damit abgeschlossen.

Endprodukt des Entwicklungsprozesses ist ein lauffähiges Programm, das alle in 6.3.6 aufgestellten Anforderungen erfüllt.

Die Erfahrungswerte bei der Verwendung des Command-Musters zur Zugriffssteuerung können nun ebenso wie die generellen Erfahrungen mit Java/RMI in die Entwicklung des Gesamtprojekts einfließen.

7 Bewertung der angewandten Methoden und Werkzeuge

7.1 UML

Die UML hat sich mittlerweile als Standard etabliert. Die Fülle an Werkzeugen, mit denen ein UML-Modell erstellt und verwaltet werden kann, wächst beständig. Dieser Umstand bringt leider einige Nachteile mit sich. Es hat sich noch immer kein Standard-File-Format für ein UML-Modell etablieren können. Ein Austausch von Modellen zwischen verschiedenen Werkzeugen ist deshalb meist nur mit verlustreichen Im- und Exporten möglich. Dabei wird oft das proprietäre, binäre Rose-Format verwendet. Hier würde sich beispielsweise ein XML-Format anbieten, das den Transfer von Modellen extrem vereinfachen würde. Erste Ansätze dazu finden sich in [DSTC].

Ein großes Problem, das viele Entwickler noch haben, besteht darin, dass ab einer bestimmten Komplexität und Größe eines Modells Code und Plan auseinanderlaufen. Häufig aus Zeitgründen beginnt der Entwickler sein eigenes Modell und die ihm damit selbst auferlegten Einschränkungen auszuhebeln. Ab diesem Zeitpunkt verliert das Modell beständig an Wert, da dessen Angleichung an den bestehenden Code häufig nicht mehr werkzeuggestützt durchgeführt werden kann. Together versucht dem durch eine verstärkte Nähe von Code und Modell entgegenzuwirken (siehe 7.5). Es ist also bei einem UML-Entwurf eine große Disziplin des Entwicklers erforderlich. Projektleiter und Kunden, die oft den Fortschritt eines Projekts am Anteil des fertigen Codes bemessen, müssen lernen, dass die Erstellung eines sauberen Modells und vor allem dessen Pflege, zeitaufwendig ist.

Es gibt aber noch einen anderen Faktor, der der Konsistenzhaltung eines Modells entgegenwirkt: schlecht entworfene API's. Diese zwingen den Entwickler oft zu einem unsauberen Entwurf. Das zu entwickelnde Programm muss sich in teilweise starre Strukturen einpassen, die die Übersichtlichkeit und Verständlichkeit des Modells verringern.

Fraglich ist auch, ob die UML ihr Ziel erreicht hat, die Kommunikation zwischen Kunde und Entwickler zu verbessern. Sicherlich ist ein Use Case Diagramm beim Beschreiben von Funktionalitäten des Systems hilfreich. Ob sich ein Kunde mit einem Diagramm aus Strichmännchen und Textblasen als Beschreibung von funktionalen Forderungen zufrieden gibt, ist jedoch zweifelhaft. In den meisten Fällen ist hier eine textuelle Beschreibung notwendig, die ein Use Case

Diagramm häufig macht. Andere Systemsichten der UML sind für Nicht-Entwickler nur schwer zugänglich und somit nur für den internen Gebrauch nutzbar.

Die UML wird von vielen Entwicklern noch als *Code-Beschreibungs-Sprache* missverstanden. So kommt es häufig vor, dass zunächst mit einem sehr rudimentären Modell schnell in die Implementationsphase eingestiegen wird, um im Laufe der Zeit gelegentlich das Modell dem bestehenden Code anzupassen. Together ist ein Werkzeug, das zu diesem Ansatz *vom Code zum Modell* verleitet. Eigentlich sollte es jedoch so ablaufen, dass mittels UML zuerst ein Modell entwickelt wird, das dann ausprogrammiert wird. Von Beginn der Programmierung an sollten kaum noch Änderungen im Modell notwendig sein. Code-Beschreibung mittels UML macht nur Sinn bei noch unmodellierten Alt-Systemen, die weiterentwickelt werden sollen.

7.2 Komponentenarchitektur

Bei der Entwicklung eines Softwaresystems trifft man unweigerlich irgendwann auf Komponenten. Sei es zur besseren Verteilung eines Systems über Rechengrenzen hinweg oder – aus Sicht eines Projektleiters – zur Verteilung der Entwicklergruppen auf die verschiedenen Unterkomponenten eines zu erstellenden Produkts.

Komponenten zwingen einen Softwareentwickler dazu, saubere Schnittstellen zu entwerfen und aussagekräftige Spezifikationen zu verfassen.

Die Integration von Komponenten in verschiedene Entwicklungstools lässt jedoch noch Wünsche offen. Eine unbestritten gelungene Integration ist Microsoft bei COM und Visual Basic gelungen. Dass das aufgrund der Einschränkungen, die Basic mit sich bringt (Verteilung, Robustheit, Portabilität), nicht ausreicht, ist einleuchtend. Gelungene Integrationen der weit verbreiteten Java-Beans werden immer häufiger; so z.B. bei Sun's Forté for Java und Inprise' JBuilder². Viele CASE-Tools bieten mittlerweile Code-Erzeugungsmechanismen für verschiedene Architekturen an (Together, Rose³). Ein großes Maß an Handarbeit bleibt aber immer noch häufig dem Entwickler überlassen.

Wünschenswert wäre eine bessere Sprach- und Plattformunabhängigkeit. DCOM ist zwar auf anderen Plattformen als Windows erhältlich, dort aber so gut wie unbekannt. Java-Komponenten sind plattformunabhängig aber nicht sprachunabhängig und CORBA krankt immer noch an der Masse von verschiedenen ORB Implementierungen, die häufig nicht alle Sprachen unterstützen und keine Interoperabilität bieten. Es bleibt zu hoffen, dass die Techniken weiter verschmelzen und sich gegenseitig unterstützen werden, um eine möglichst einfache Verwendung auf heterogenen Systemen zu gewährleisten.

Aktuelle Komponententechnologien haben immer noch Probleme mit der Performance, was deren Akzeptanz gerade bei Telematik-Systemen verringert. Man kann aber davon ausgehen, dass allein aus ökonomischen Aspekten, die Verwen-

² siehe <http://sun.java.com> und <http://www.borland.com>

³ siehe <http://www.togethersoft.com> und <http://www.rational.com>

derung von Komponenten auch in diesem Sektor weiter voranschreiten und später vielleicht in eine hardware-unterstützte Komponenten-Technik münden wird.

7.2.1 RMI

Da zur verteilten Kommunikation RMI verwendet wurde, soll kurz auf die praktischen Erfahrung damit eingegangen werden.

RMI hat den großen Vorteil, dass die Verwendung dieser Technologie relativ transparent für den Entwickler abläuft. Natürlich muss er sich Gedanken darüber machen, welche Schnittstellen entfernt und welche lokal angesprochen werden. Hat er aber einmal diese Entscheidung getroffen, läuft der restliche Entwurf wie gewohnt. Dies liegt vor allem an der Unterstützung der gesamten Java-Plattform. Es wird also nicht, wie bei COM und CORBA eine eigene Schnittstellen-Beschreibungssprache (IDL) benötigt. Ein Nachteil besteht darin, dass Komponenten, die für die RMI-Kommunikation entworfen wurden, auch nur über Java ansprechbar sind.

7.3 Programmiersprache und API

Die in Kapitel 6 verwendete Sprache Java, besticht vor allem durch die geringe Größe des kompilierten Programms. Die Gesamtgröße von CarTel ohne die virtuelle Maschine (VM) beträgt ca. 300K, was sehr wenig ist, wenn man bedenkt, dass sogar ein einfaches HMI darin enthalten ist. Dennoch ist fraglich, ob sich Java im Bereich der eingebetteten Systeme etablieren kann. Jüngste Vorstöße in diesen Bereich wie JavaCard, EmbeddedJava und die Micro Edition von Java 1.2 lassen auf ein erhöhtes Interesse diesbezüglich schließen.

Dass der Quellcode zu den meisten Java-bezogenen Projekten von Sun mittlerweile frei verfügbar ist, könnte ein Vorteil gegenüber geschlossenen, proprietären API's, wie beispielsweise Win32, sein. Fehler und Performanz-Probleme können auf diese Art schneller gefunden und übergearbeitet werden. Zudem steigt das Vertrauen in ein eingesetztes System erheblich, wenn alle Eigenheiten offengelegt und für jeden einsehbar sind.

7.4 Entwicklungsprozess

Die Notwendigkeit eines Entwicklungsprozesses bei der Erstellung eines umfangreichen Softwaresystems ist unbestritten. Kein Entwicklerteam kann es sich heute noch erlauben, plan- und konzeptlos *ins Blaue* zu programmieren.

Trotzdem muss bei der Anwendung eines Prozesses auch immer die Flexibilität gewahrt bleiben. Es muss möglich sein, auf neue Techniken und Entwicklungen einzugehen und den Prozess entsprechend anzupassen. Damit haben viele Prozesse noch Probleme.

Ein evolutionärer Prototyp macht wenig Sinn, wenn die Technik, auf dem er basiert, bei Fertigstellung bereits veraltet ist. Geld und Zeit in einen Wegwerf-Prototypen zu investieren, um danach eventuell wieder bei Null anzufangen, ist auch ökonomisch gesehen wenig sinnvoll.

Es sieht so aus, als sei ein bestimmtes Maß an Chaos in einem Softwareprojekt nicht zu vermeiden. Vielleicht sollte daran gearbeitet werden, dieses Chaos als kreativen Schub zu akzeptieren. Ziel eines Prozesses wäre dann der Versuch, chaotische Züge in Bahnen zu lenken und nicht, diese zu bekämpfen.

Angesichts der vielen bereits gescheiterten Softwareprojekte, ist fraglich, ob der optimale Prozess schon gefunden wurde.

7.4.1 Bewertung der Prozessrichtlinien

Use Case Basierung

Bei der Entwicklung einer verteilten Architektur hat man oft das Problem, dass es oft schwierig ist, Use Cases zu finden. Während der in Kapitel 6 erläuterten Entwicklung musste immer wieder festgestellt werden, dass Use Cases in vielen Bereichen oftmals entweder zu abstrakt oder einfach nutzlos sind.

Betrachtet man beispielsweise den Entwurf einzelner Dienste des Systems, die selbst nur von Komponenten innerhalb des Systems verwendet werden, so fällt auf, dass Use Cases hier kaum weiterhelfen. Sinnvoll hingegen sind sie beim Entwurf von Anwendungen, deren Funktionalität nach aussen verfügbar gemacht wird (Beispiel: CD-Spieler).

Architektur-Zentrierung

Die Zentrierung auf die Aspekte der Architektur waren bei der Beispielentwicklung sehr hilfreich. Vor allem das Schichtenmodell half dabei, einen ersten Überblick über das System zu erlangen. Das dabei entstandene Package-Diagramm (Abbildung 6.2) eignet sich durch seine Einfachheit gut als Diskussionsgrundlage zwischen Auftraggeber und Entwickler.

Die Architekturbeschreibung kann zudem eine große Hilfe dabei sein, schwerwiegende Designfehler zu vermeiden, die eine unnötige Verzahnung von Klassen zur Folge haben könnte. Eine solche Abhängigkeit verringert den Grad an Wiederverwendbarkeit von Komponenten in anderen Projekten.

Iterative und inkrementelle Vorgehensweise

Da nur ein einziges Inkrement bei der Beispielentwicklung in Kapitel 6 erstellt wurde, ist es schwierig, eine Bewertung dieses Ansatzes vorzunehmen. Die Erfahrung, die ich in anderen Projekten gemacht habe, ist aber die, dass inkrementelles Entwickeln nur sehr schwer zu verwirklichen ist.

Das inkrementelle Anwachsen von Eigenschaften eines Systems setzt immer einen sehr guten Überblick über das Gesamtsystem voraus. Es ist sehr schwierig, eine Software so zu entwickeln, dass sie ohne großen Aufwand erweiterbar wird. Häufig passiert es, dass bei der Entwicklung so viel Aufmerksamkeit auf Erweiterbarkeit und Offenheit gelegt wird, dass der resultierende Entwurf zwar sehr sauber wird, jedoch auch sehr teuer. Bei der Entwicklung muss immer die Frage gestellt werden, ob es besser ist, mit viel Aufwand ein erweiterbares Inkrement zu

entwickeln oder vielleicht doch lieber das gesamte Produkt. Dass das inkrementelle Entwickeln durchaus möglich ist, zeigen erfolgreiche offene Projekte, an denen beständig, mit wechselnden Entwicklern weitergearbeitet wird. Beispiele hierfür sind Linux, Apache oder seit neuerem auch Java.

Die Entwicklung eines Softwaresystems in mehreren Iteration macht in jedem Fall Sinn. Meilensteine vermitteln ein Gefühl von Sicherheit. Jeder dieser Meilensteine kennzeichnet ein erreichtes Ziel, auf das im weiteren Prozess aufgebaut werden kann.

Der Nutzen der Aufteilung einer Iteration in die verschiedenen Phasen von Anforderungsanalyse bis Test wird oft unterschätzt. Gerade die Entwurfsphase, die voreilige Programmierer oft auf eine Geduldsprobe stellt, ist von großer Bedeutung. Mit dem Programm- und Architekturdesign steht und fällt das gesamte Projekt. Eventuelle Unwägbarkeiten während der Implementationsphase, können häufig durch einen ausgefeilten Programmentwurf abgefangen werden.

Bei CarTel trat eine solche Unwägbarkeit bei der Umsetzung des konkreten Zugriffs auf den CD-Spieler unter WindowsNT auf. Da aber durch den Entwurf eine klare Trennung des CD-Spielers von den restlichen Teilen des Systems vorgegeben war, konnte parallel zur Suche nach einer Lösung an anderen Teilen weiterentwickelt werden. Es bestand also nie eine Gefährdung des gesamten Projekts, da das Problem nur Auswirkung auf einen sehr kleinen Teil des Programms hatte.

Prozessmuster

Um den Umfang der Arbeit nicht zu sprengen, wurden Prozessmuster nur am Rande angeschnitten. Es sei hier jedoch noch einmal auf den Gewinn an Flexibilität für den Prozess hingewiesen, den solche Muster bringen können. Gerade beim Auffinden und Entwurf von Komponenten hilft der Catalysis-Ansatz enorm weiter. Vielleicht gelingt es mit ihm am ehesten, das bereits erwähnte Chaos während der Entwicklung komplexer Systeme in geordnete Bahnen zu lenken.

7.4.2 *Ergänzung*

Das Studium und die Verwendung von Entwurfsmustern sollte noch stärker von einem Prozess gefordert werden. Bei der Entwicklung von CarTel stellte sich immer wieder heraus, wie hilfreich die Kenntnis dieser Muster sein kann. So ist beispielsweise der gesamte Zugriffsmechanismus auf dem Command-Muster aufbauend entwickelt wurden.

7.5 *CASE-Tool: Together*

Als entwicklungsbegeleitendes Werkzeug wurde das vollständig in Java programmierte Together der Firma Togethersoft (ehemals Object International) verwendet. In der ersten Phase wurde dabei die Version Together/J 3.1 (Build 720) verwendet. Diese erwies sich jedoch als so fehlerhaft, dass eine weitere Verwendung erst mit Build 874 gewagt wurde.

7.5.1 Die „Immer-Synchron“-Philosophie

Together ist ein UML-Werkzeug, das alle UML-Sprachelemente und Diagramme außer dem Objektdiagramm unterstützt. Die Philosophie des Programms liegt darin, das Modell, die Dokumentation und den Quellcode eines Softwareprodukts immer synchron zu halten. Das bedeutet, dass jede Änderung im Modell sofort eine Änderung des zugehörigen Quellcodes mit sich bringt. Umgekehrt bewirken Kommentare im Quellcode Dokumentationseinträge im Modell. Auf diese Weise soll verhindert werden, dass irgendwann einmal Code und Modell nicht mehr zueinander passen und ein aufwendiger Abgleich durchgeführt werden muss.

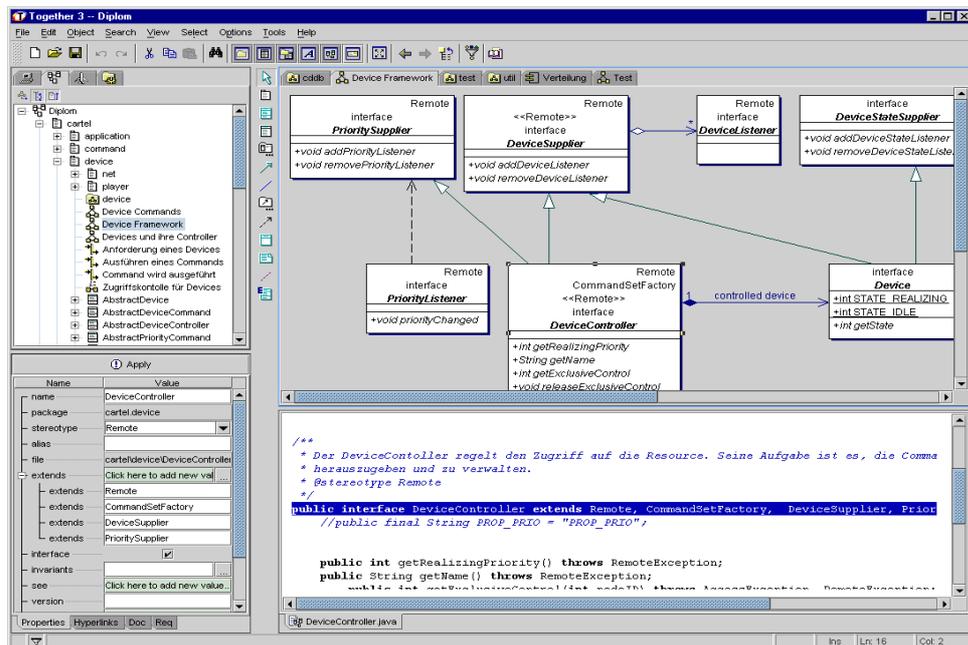


Abbildung 7.1: Together Hauptbildschirm

Die Gefahr bei diesem Ansatz liegt auf der Hand: durch unüberlegtes Entfernen von Modellelementen ist es sehr leicht möglich, den Quellcode so zu verändern, dass er völlig unbrauchbar wird. Da das Zurücknehmen von Fehlaktionen nicht immer zu dem Zustand vor der Aktion führt (siehe 7.5.2), kann dies fatale Folgen haben. Umgekehrt erscheinen ständig Elemente im Modell, die vielleicht vollkommen implementationspezifisch sind, die man lieber nur im Code aber nicht im Modell haben möchte.

Wie in Kapitel 2 beschrieben, wurde die UML auch dazu entwickelt, die Kommunikation zwischen Entwicklern und Auftraggebern zu verbessern. UML-Modelle sollten so aufgebaut sein, dass ihre Aussage auch von solchen Personen

4 „Always-In-Synch“: Claim des Together-Produktes

verstanden werden kann, die keinerlei Erfahrung mit Softwareentwurf haben. Aus diesem Grund wird bei UML-Modellen oft nur das Wichtigste in stark vereinfachter Form dargestellt. Diese Vorgehensweise wird von Together sicherlich nicht ausreichend berücksichtigt. Es ist nicht möglich auch nur eine Klasse in einem Diagramm zu verwenden, zu der nicht auch gleichzeitig Source-Code gehört.

Together eignet sich sehr gut zur Fortführung, Wartung und Dokumentation von älterer Software, für die vielleicht noch kein Modell existiert. Das Werkzeug ermöglicht es, sofort eine Klassendiagramm-Sicht auf den vorhandenen Code zu erstellen. Diese trägt erheblich zur Verständlichkeit bei und kann sofort als Grundlage für Veränderungen oder Erweiterungen dienen. Dass jedoch keine Stub-Code-Erzeugung (Erzeugung von Code-Skeletten ohne Implementation) aus einem Modell möglich ist, ist unverständlich. Damit wäre es möglich Schnittstellen einer Legacy-Software beizubehalten und die Implementation auszutauschen.

Im folgenden sollen die funktionalen und qualitativen Merkmale des Programms genauer untersucht werden.

7.5.2 Funktionale Analyse

allgemeine Eigenschaften

Together weist einige Eigenheiten auf, die in vielen anderen UML-Werkzeugen nicht oder anders vorhanden sind. Umgekehrt fehlen aber auch einige wichtige Funktionen.

positiv

- Die Erstellung von Dokumentationen und Reporten ist in Together sehr flexibel und umfangreich möglich.
- Es ist nahezu alles frei konfigurierbar und kann den persönlichen Vorlieben des Anwenders angepasst werden.
- Viele externe Werkzeuge wie Versions- und Anforderungsverwaltung sind in das Programm integriert und von dort aus aufrufbar.
- Diagramme besitzen ein gutes Auto-Layout, was die Darstellung von importierten Quellcode und damit sein Verständnis verbessert.
- Der in Together bearbeitete Quellcode kann vom Programm automatisch formatiert werden. Damit kann erreicht werden, dass der Quellcode auch bei mehreren Entwicklern einheitlich aussieht.

negativ

- Together ist kein „reines“ Java-Programm. Es werden plattformabhängige Aufrufe zum Starten von externen Werkzeugen wie beispielsweise Rose verwendet. Diese Eigenschaft scheint auf den ersten Blick nicht sonderlich nachteilig zu sein. Man muss sich jedoch fragen, warum überhaupt in Java programmiert wird, wenn die Plattformunabhängigkeit nicht gewahrt bleibt.

Eine performantere Sprache hätte einige qualitative Mängel des Programms sicherlich beseitigen können.

- Man kann immer nur ein Projekt gleichzeitig bearbeiten. Der Austausch von Modell-Elementen aus anderen Projekten ist nur über den Quellcode möglich.
- Es ist keinerlei Beschriftung in Diagrammen möglich.
- Assoziationen werden im Code mit Kommentaren gespeichert, was diesen nicht unbedingt übersichtlicher macht. Diese werden manchmal auch nicht entfernt, wenn die Assoziation entfernt wurde.
- Laut Togethersoft bietet Together die Option eines Modellex- und import nach/von Rational Rose. Dieser funktioniert jedoch nur, wenn auf dem gleichen Rechner, auf dem Together läuft, Rose installiert ist. Um also mit einem anderen Entwickler Modelle austauschen zu können, der nur Rational Rose verwendet, muss der Together-Lizenznehmer auch eine Rose-Lizenz erwerben.
- In Diagrammen werden Generalisierungen/Realisierungen nur über eine Ebene automatisch angezeigt. Möchte man vor allem die Realisierung eines bestimmten Interfaces darstellen, so erscheint das natürlich auch im Quellcode, obwohl die Superklasse, das Interface ja schon implementiert. Dieses Manko wurde in Build 874 angegangen, in dem eine erweiterte Klassendarstellung eingeführt wurde. Hier werden jetzt alle nicht explizit im Diagramm erkennbaren Generalisierungen als Text im oberen Bereich der Klasse angezeigt.

Abbildung 7.2 zeigt zwei Diagramme, die dieses Verhalten erläutern sollen. Diagramm1 zeigt eine Klassenhierarchie, in der KlasseA InterfaceB implementiert, das wiederum InterfaceA erweitert. In Diagramm2 möchte man nun darstellen, dass KlasseA InterfaceA implementiert, was durch die Implementierung von InterfaceB implizit gegeben ist. Leider ist das in Together nicht ohne weiteres möglich. Würde man in Diagramm2 die Implementation von InterfaceA durch KlasseA einfügen (Diagramm 2.1), so würde dies auch gleichzeitig im Quellcode (`class KlasseA implements InterfaceB, InterfaceA`) und in Diagramm1 (Diagramm 1.1) erscheinen.

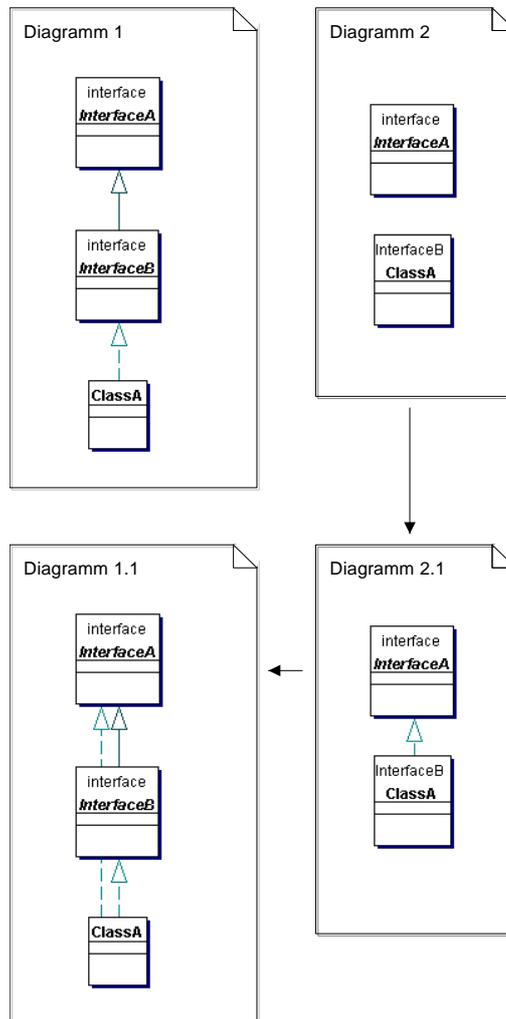


Abbildung 7.2: Together Diagramme (siehe Text)

- Bei einem Werkzeug dieser Preisklasse hätte man eine Internationalisierung erwarten dürfen, zumal diese in Java kein großes Problem darstellt (siehe Anhang).

Handhabung

Ein Entwicklungswerkzeug begleitet einen Prozess über einen langen Zeitraum. Die Handhabung des Programms beeinflusst dabei wesentlich seine Akzeptanz im Entwicklerteam. Eine unpraktische oder komplizierte Bedienung kann zu Zeitverlust und Frustration führen und den Fortschritt der Entwicklung negativ beeinflussen. Together besitzt einige Eigenschaften, die die Akzeptanz des Programms sicherlich nicht begünstigen.

7 Bewertung der angewandten Methoden und Werkzeuge

- Erstellt man ein neues Diagramm, in dem man bereits im Modell vorhandene Elemente verwenden möchte, so muss man einen sogenannten Shortcut auf dieses Element verwenden. Die Einfügung eines solchen Shortcuts wird über einen recht umständlichen Mechanismus mit Kontextmenu und Dialog realisiert. Ein Shortcut hat im Gegensatz zu einer „echten“ Klasse die Eigenschaft, dass das Löschen nicht gleichzeitig das Löschen der Klasse als Quellcode bewirkt. Dass dies nicht ganz so funktioniert wie geplant wird im Abschnitt Fehler behandelt.
- Es ist kein explizites Speichern des Modells möglich. Der Code ist das Modell und umgekehrt. Alle Änderungen werden sofort im Code übernommen und gespeichert. Der Sinn der *Speichern*-Funktion, die bei Betätigung keinerlei Feedback gibt, was passiert ist, bleibt unklar.
- Gerichtete Assoziationen können ihre Richtung nicht mehr ändern. Möchte man also eine solche Assoziation umkehren, muss die alte gelöscht werden und eine neue eingefügt werden.
- Das Programm bietet einen integrierten Quellcode-Editor. Dieser beherrscht weder Syntax-Hervorhebung, noch automatische Methoden- und Attributsauswahl, wie sie in fast allen anderen größeren Werkzeugen (z.B. JBuilder, Forté, Visual J++) vorhanden sind.
- Notes werden nicht mehrzeilig, wenn man sie verkleinert. Die AutoSize-Funktion funktioniert nicht.
- Es gibt kein Drag und Drop aus dem ElementBrowser, wodurch das Wählen einer bestimmten Klasse immer über einen Dialog geregelt wird, der eigentlich nur wieder den ElementBrowser enthält. Das Einfügen einer Klasse in das aktuelle Diagramm ist ebenfalls nur über diesen Dialog möglich. Ein Eintrag im Kontextmenu der Klasse wäre hier sicherlich hilfreich gewesen.
- Klassen verschwinden aus Diagramm, wenn ein Syntaxfehler im Quellcode existiert. Erst nachdem man sie mit einem externen Editor verbessert hat erscheinen sie wieder. Dieses Verhalten tritt auch dann auf, wenn man eine Klasse im Quellcode umbenennt, anstatt über die vom Werkzeug bereitgestellten Dialoge
- Es ist nicht möglich, zwei miteinander auf irgendeine Weise assoziierte Elemente in einem Diagramm zu verwenden, ohne dass dabei diese Assoziationen auch angezeigt werden. Das ist einer der Hauptgründe, weshalb Together zum Erstellen von aussagekräftigen Diagrammen nicht geeignet ist. Die Verständlichkeit von Diagrammen leidet darunter, dass immer auch Elemente aus anderen Diagrammen angezeigt werden, die mit der Hauptintension des aktuellen Diagramms nichts zu tun haben.

Fehler

Das Programm weist einige offensichtliche Fehler auf, die nur darauf zurückzuführen sind, dass die Synchronisation zwischen Quellcode und Modell nicht ausgereift ist. Zu den auffälligsten gehören:

- Klassen erscheinen manchmal im Klassenbaum doppelt und dreifach, können aber nicht gelöscht werden.
- Manche Klassen sind angeblich nur im kompiliertem Zustand vorhanden. Diese können nicht mehr aus dem Modell gelöscht werden, selbst wenn man die zugehörige `.class`-Datei gelöscht hat. Nur durch Schließen und erneutem Öffnen des Projektes ist dieses Problem manchmal behebbar. Zudem ist es nicht möglich, eine neue Klasse zu erstellen, wenn im Pfad eine `.class`-Datei existiert, die eine Klasse gleichen Namens definiert.
- Es kann vorkommen, dass ein Diagramm sein Aussehen nach Schließen und erneutem Öffnen eines Projekts verändert hat.
- Der Quellcode-Generator wirft Klassen durcheinander, wodurch manche Operationen in falschen Klassen oder an falschen Stellen im Code landen.
- Das Drucken funktioniert nur mit der Microsoft VM wie erwartet. Bei allen anderen sind oft Ausrichtung und Skalierung falsch. Da andere Programme diese Probleme nicht haben, muss hier ein Implementationsfehler vorliegen. Zudem bewirkt Druckoption „Print Footer“ immer genau das Gegenteil von dem, was gewählt wurde.
- Hat man in einem Komponenten-Diagramm Komponenten und Shortcuts auf Klassen des Modells gemischt und löscht eine Komponente gemeinsam mit Shortcuts, so werden die den Shortcuts zugrundeliegenden Modellelemente ebenfalls gelöscht. Dies hat einen Verlust von ganzen Quellcode-Dateien zur Folge. Gleiches gilt für das Löschen von Assoziationen zwischen Shortcuts.

7.5.3 Qualitative Analyse

Da Together vollständig in Java implementiert ist, müssen von vornherein Abstriche bezüglich Performance gemacht werden. Eine dermaßen schlechte Performance, wie sie dieses Programm an den Tag legt, steht jedoch in keiner Relation. Auf einem doch noch recht zeitgemäßen Pentium II-Rechner mit 128MB RAM wird der Anwender oft mit sekundenlangen Wartepausen konfrontiert. Zudem wächst der Hauptspeicherbedarf des Programms kontinuierlich mit der Zeit bis weit über 100MB. Dies ist auch schon bei kleineren UML-Modellen der Fall. In solchen Fällen hilft dann nur der Neustart des Programms, nach dem es dann wieder mit etwa 50MB beginnt. Das Starten des Programms, ohne ein Modell zu laden, dauert auf dem oben beschriebenen Rechner eine knappe Minute.

Beim Blick auf die Ausgaben der Java-VM erkennt man, dass es wohl noch erhebliche Fehler im Programm zu beheben gilt. Manchmal gelangt Together in einen Zustand, in dem es bei fast jedem Mausklick eine Exception auslöst.

7.5.4 Fazit

Together ist ein Programm mit guten Zielsetzungen. Es ist sicher der Wunsch jedes Entwicklers, alle zu einem Projekt gehörenden Dokumente zentral zu verwalten und sie automatisch zu synchronisieren. Leider schafft Together es nicht, diesem Anspruch vollständig gerecht zu werden. Das Programm leidet

7 Bewertung der angewandten Methoden und Werkzeuge

unter zu vielen Fehlern und einer umständlichen und schwerfälligen Handhabung. Viele gute Ansätze wie die Einbindung von externen Werkzeugen, die Anforderungs- und Versionsverwaltung und das gelungene Report-Modul können nicht über den Verlust von Quellcode, der aufgrund von Fehlern des Programms droht, hinwegtrösten.

Es bleibt also abzuwarten, ob Together in neuen, bereinigten Versionen diese Mängel beseitigen kann.

7.5.5 Alternative: MagicDraw

Ein UML-Werkzeug, das vielleicht eine Alternative zu Together darstellen könnte, ist das recht unbekanntere MagicDraw⁵. Die UML-Diagramme in Kapitel 6 wurden aufgrund der oben beschriebenen Probleme mit Together gänzlich mit diesem Java-Programm erstellt.

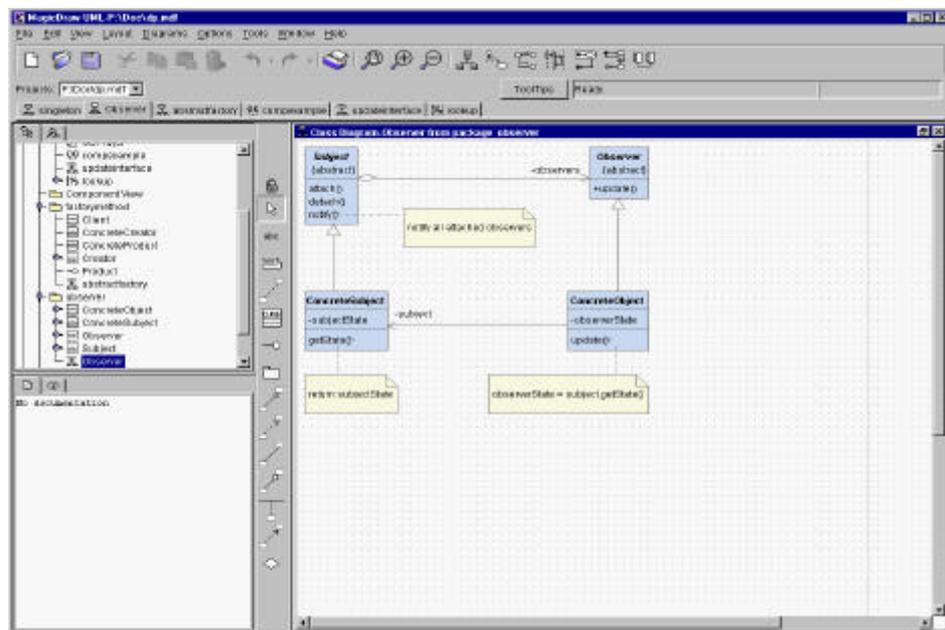


Abbildung 7.3: Hauptbildschirm Magic Draw

Es weist nicht die Menge von Werkzeugen auf, die Together bietet, beherrscht aber die gleichen Diagrammtypen. Die Synchronisation von Code und Modell wird hier immer vom Anwender angestoßen, der entweder das Modell anhand von Code erneuert, oder umgekehrt, Code aus dem Modell erzeugt. Dabei wird vorhandener Code nicht einfach überschrieben, sondern mit dem Modell kombiniert. Methodenrumpfe werden grundsätzlich beibehalten, so dass keine Gefahr besteht, Quellcode zu verlieren. Bis auf einige kleine aber ungefährliche Mängel

5 <http://www.nomagic.com>

funktioniert dieses Verfahren sehr gut. Zudem ist es möglich, anzugeben für welche Modellelemente Code in welcher Sprache erzeugt werden soll.

Auf eine vollständige Analyse des Programms soll an dieser Stelle unter Hinweis auf die frei zu beziehende Demo-Version unter <http://www.nomagic.com> verzichtet werden.

8 Anhang

8.1 Entwurfsmuster

Die hier verwendeten Erläuterungen zu den in Kapitel 6 benutzten Entwurfsmustern stammen aus [Ganssle-91].

8.1.1 Observer

Das Observer-Muster dient zum Entwurf einer 1:n-Abhängigkeit von Objekten, bei der bei einer Statusänderung des einen, alle anderen benachrichtigt werden.

Motivation

In einem stark partitionierten System ist es oft schwierig, Konsistenz zwischen einzelnen, voneinander abhängigen Objekten zu wahren. Eine zu starke Verzahnung der Objekte sollte zugunsten der Wiederverwendbarkeit vermieden werden.

So ist vor allem in einem System mit grafischer Repräsentation (View) von Datenobjekten (Model) eine Benachrichtigung bei Änderungen des Modells hilfreich.

Struktur

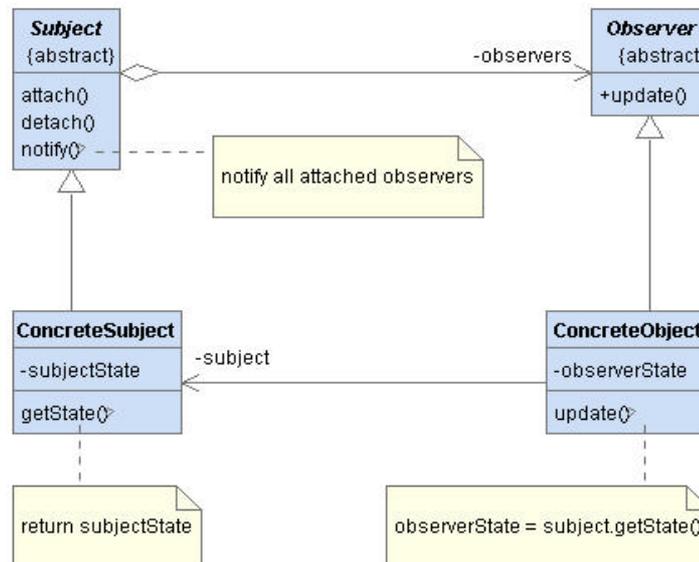


Abbildung 8.1: Das Observer Entwurfsmuster

8.1.2 Singleton

Mit der Verwendung des Singleton-Muster kann sichergestellt werden, dass immer nur eine Instanz einer bestimmten Klasse existiert.

Motivation

Bestimmte Systemdienste (Druck-Spooler, Dateisystem, etc.) sollten immer nur einmal vorhanden sein, damit der Verwaltungsaufwand so gering wie möglich ist.

Struktur

Eine Klasse, die nach dem Singleton-Muster entworfen ist, verwaltet eine statische Instanz auf die über eine statische Methode zugegriffen werden kann. Eine andere Möglichkeit, eine Instanz von Singleton zu erlangen, gibt es nicht.

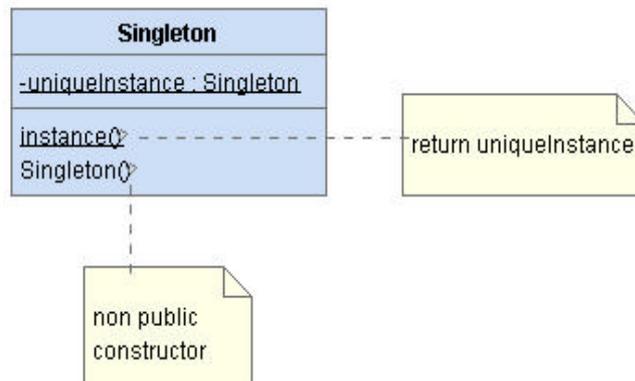


Abbildung 8.2: Singleton-Struktur

8.1.3 Factory Method

Mit Hilfe einer Factory kann man eine Schnittstelle zur Instanziierung von Objekten entwerfen, bei der die konkrete Implementation entscheidet, aus welcher Klasse, die Instanz kommt. Man erreicht damit eine Verschiebung von Instanziierungen in Subklassen. Dieses Entwurfsmuster wird deshalb auch oft als *virtueller Konstruktor* bezeichnet.

Struktur

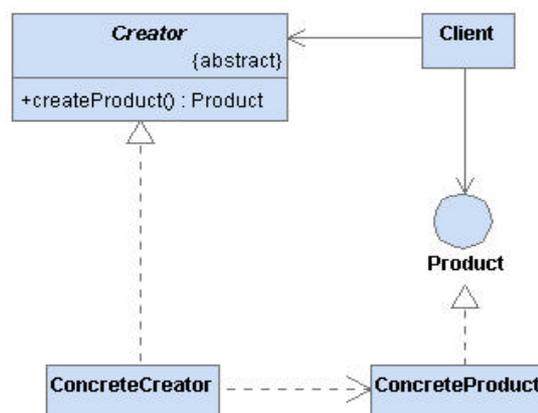


Abbildung 8.3: Struktur Abstract Method

8.2 Java und Internationalisierung

Unter Internationalisierung versteht man den Prozess, eine Applikation so zu entwerfen, dass es mehrere Sprachen und Regionen unterstützt. Diese Unterstützung soll dabei ohne Änderung im Programm möglich sein. Internationalisierung wird oft mit i18n abgekürzt, da zwischen dem ersten i und dem letzten n im englischen Wort internationalization 18 Buchstaben liegen.

Ein internationalisiertes Programm hat folgende Eigenschaften:

Durch die Hinzunahme von lokalisierten Daten kann das gleiche Programm weltweit laufen. Es muss also durch die Hinzunahme einer neuen Sprache nicht erneut kompiliert werden.

Textuelle Elemente wie Nachrichten oder Text auf GUI-Elementen sind nicht hart codiert sondern extern gespeichert und werden dynamisch geladen.

Eine Lokalisierung muss schnell möglich sein.

All diese Eigenschaften werden durch Java unterstützt. So ist es beispielsweise möglich, sogenannte ResourceBundles anzulegen, von denen je nach Region, dynamisch die mit der richtigen Sprache geladen wird.

Ein großes Problem stellt oft die veränderte Länge von Texten in einer anderen Sprache dar. Diese kann zur Folge haben, dass Knöpfe oder andere Elemente der grafischen Benutzerschnittstelle zu klein sind, um den neuen Text darzustellen. Auch hier bietet Java eine Lösung, da auch die Größen und Positionen von GUI-Elementen dynamisch errechnet werden.

9 Quellen

- [**BOGUSCH-99**] R. Bogusch, A. Kulms, G. Reinhold, F. Schiller *DC-Project: Architectural Requirements (Draft)*. Bosch K6-SM/ENG2, 1999.
- [**CAT-98**] Alan Cameron Wills *Catalysis, Technical Briefing*. www.trireme.com, 1998.
- [**COAD-99**] Peter Coad, Eric Lefebvre, Jeff De Luca. *Feature-Driven Development*. TogetherSoft LLC, 1999.
- [**DOUGLASS-99**] Bruce Powel Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, Reading, Mass., 1999.
- [**DOUGLASS-99-2**] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
- [**DOWNING-99**] Troy Bryan Downing. *Java Rmi: Remote Method Invocation*. IDG Books Worldwide, 1999.
- [**D'SOUZA-98**] Desmond D'Souza. *Objects, Components, and Frameworks with UML. The Catalysis Approach*. <http://www.iconcomp.com>.
- [**DSTC**] DSTC. *DSTC's Meta-Object Facility Publications*. <http://www.dstc.edu.au/Research/Projects/MOF/Publications.html>, 2000.
- [**EDDON-98**] Guy Eddon, Henry Eddon. *Inside Distributed Com*. Microsoft Press, 1998.
- [**EE498**] Kuhn, Kelin J. *CD-ROM –An extension of the CD audio standard*. <http://www.ee.washington.edu/conselec/CE/kuhn/cdrom/95x8.htm>.
- [**FLAN-99**] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference*. O'Reilly, 1999.
- [**FOWLER-96**] Martin Fowler. *The Art of Programming Embedded Systems*. Academic Press Inc., 1991.
- [**GANSSE-91**] Jack G. Gansse. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

- [**GAMMA-94**] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.
- [**HÄUBLER-97**] Bärbel Häußler. *Die Basiskonzepte der Unified Modelling Language und ihre Abbildungen im Metamodell*. Fachhochschule Konstanz, September 1997.
- [**JACOBSON-99**] Ivar Jacobson, Grady Booch, James Rumbaugh. *The unified software development process*. Addison-Wesley, 1999.
- [**JFC-00**] Sun Microsystems. *Java Foundation Classes (JFC)*. <http://java.sun.com/products/jfc>, 2000.
- [**McIL-68**] McIlroy M.D. *Mass produced software components*. In (Natur et al., 1996), pp. 88-98, 1968.
- [**MEYER-90**] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1990.
- [**MULL-93**] Sape Mullender (ed.). *Distributed Systems, Second Edition.*, Addison Wesley, 1993.
- [**MURRAY-99**] John Murray. *Inside Microsoft Windows CE*. Microsoft Press, 1999.
- [**ORFALI-99**] Dan Harkey (ed.), Robert Orfali. *Client/Server Programming with Java and CORBA, Second Edition*. John Wiley & Sons, 1999.
- [**QUARTANI-98**] Terry Quartani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.
- [**RUMBAUGH-99**] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference manual*. Addison-Wesley, Reading, Mass., 1999.
- [**RIEL-96**] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
- [**SIEMENS-99**] U.Margull, et.al.. *Software Architecture Headunit BR211 HighLine*. Siemens AG, 1999.
- [**SIMON-99**] David E. Simon. *An Embedded Software Primer*. Addison-Wesley , 1999.
- [**STEVENS-99**] Perdita Stevens, Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.
- [**STÜMPFLE-99**] M.Stümpfle, et.al. *COSIMA - A Component System Information and Management Architecture*. Daimler Chrysler AG - Research and Technology3, 1999.
- [**SZYPERSKI-97**] Clemens Szyperski. *Component Software*. Addison-Wesley, Longman Limited, 1997.
- [**TOGETHER-99**] Together. *Together - User Manual Set*. TogetherSoft LLC, 1999.

[VLISSIDES-98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.