UNIVERSITÄT DORTMUND

Fachbereich Informatik
Lehrstuhl für Software-Technologie

M E M O Nr. 138

Endbericht der Projektgruppe 415: Konzeption und Implementierung eines digitalen und hypermedialen Automobilcockpits (HyCop)

K. Alfert A. Fronk Ch. Veltmann (Eds.)

Stefan Borggraefe Leonore Brinker Evgenij Golkov
Rafael Hosenberg Bastian Krol Daniel Mölle
Markus Niehammer Ulf Schellbach Oliver Szymanski
Tobias Wolf Yue Zhang

Mai 2003

Internes Memorandum des
Lehrstuhls für Software-Technologie
Prof. Dr. Ernst-Erich Doberkat
Fachbereich Informatik
Universität Dortmund
Baroper Straße 301

D-44227 Dortmund

ISSN 0933-7725

Endbericht der Projektgruppe 415: Konzeption und Implementierung eines digitalen und hypermedialen Automobilcockpits (HyCop)

K. Alfert, A. Fronk, Ch. Veltmann (Hrsg.)

Stefan Borggraefe, Leonore Brinker, Evgenij Golkov
Rafael Hosenberg, Bastian Krol, Daniel Mölle
Markus Niehammer, Ulf Schellbach, Oliver Szymanski
Tobias Wolf, Yue Zhang

20. Mai 2003

Inhaltsverzeichnis

Kapitel 1

Einführung in die PG HyCop

Autor: *Alexander Fronk*

1.1 PG-Aufgabe

Die Projektgruppe soll sich mit der GUI-Entwicklung auf Basis der Konstruktion hypermedialer Systeme befassen: Die aus der Konstruktion von Hyperdokumenten bekannten Techniken sollen auf die GUI-Entwicklung übertragen werden und so zu neuen Ansätzen führen, die insbesondere in der industriellen Entwicklung hypermedialer Cockpits im Automobilbereich Aufmerksamkeit finden.

1.1.1 Motivation

Die Entwicklung von Mensch-Maschine-Interaktionssystemen im Automobilbereich ist ein aktuelles Arbeitsgebiet, dessen Ergebnisse zunehmend am Markt sichtbar und angeboten werden. Diese Systeme umfassen neben der digitalen Bereitstellung der üblichen Informationen über Geschwindigkeit und Drehzahl auch Fahrerinformations- und Assistenzsysteme. Zu ersteren zählen Navigation oder Internetzugang, zu letzteren Abstandsregelsysteme oder eine elektronische Lenkunterstützung. Wesentlicher Bestandteil dieser Systeme ist Software zur Verarbeitung der nötigen Informationen und deren angemessener Darstellung in geeigneten Displays.

Aber auch fachlich ist das recht spannend. Aus software-technischer Sicht spielt die Implementierung von grafischen Benutzungsoberflächen (kurz GUIs) eine tragende Rolle. Dabei wird meist *top-down* vorgegangen, d.h. ausgehend vom Aussehen der Oberfläche wird ihre Funktionalität quasi nachträglich „hineingehängt“. Sowohl die Konstruktion der Oberfläche als auch die Implementierung ihrer Funktionalität erfolgen meist objektorientiert. Hierzu existieren sog. *GUI-Frameworks*, die auf ordentlichen Konzepten wie etwa dem Modell-View-Controller-Ansatz aufbauen. Java Swing ist dafür ein Beispiel. Entscheidend ist dabei, dass sich die Funktionalität als möglichst stabil gegen das sich rasant verändernde und dem Zeitgeist unterworfene Design eines Cockpits erweist und dies auch soll.

Die Tatsache digital zu sein, macht ein solches Cockpit in einem anderen Sinne dennoch recht wandlungsfähig. Es gelingt, zu verschiedenen Zeitpunkten verschiedene Dinge darzustellen, also situationsbezogen die „Inhalte“ des Cockpits zu verändern. Gemeint ist hier zum Beispiel, ein *Cruise-Control*-System bei Autobahnfahrten anzubieten, bei Stadtfahrten hingegen nicht; ein Mechaniker einer Werkstatt möchte bei Probefahrten sicher detaillierte Informationen über Brems- und Öldruck sehen; auch kann das Layout des Cockpits an den Kunden individuell angepasst werden. Solche und ähnliche Szenarien in den Griff zu bekommen bedarf dann mehr als der bloßen Implementierung von Benutzungsschnittstellen.

Es hat sich gezeigt, dass Interaktionssysteme der beschriebenen Art als Hyperdokumente im Sinne einer Darstellung nicht-linear verknüpfter Daten betrachtet werden können. Eine grafische Benutzungsschnittstelle ist – vereinfacht dargestellt – genau wie ein Hyperdokument als ein Graph zu verstehen, dessen Knoten Informationen entsprechen und dessen Kanten eine mögliche Informationsverknüpfung meinen, und der somit ein Hyperdokument (resp. eine Benutzungsschnittstelle) im konzeptionellen Sinn repräsentiert. So kann man beispielsweise eine Menüleiste einer Oberfläche als eine Menge von Knoten verstehen, ihre jeweiligen Aufgaben ebenfalls. Eine Kante e zwischen zwei Knoten v und w zeigt dann an, dass ein Menüpunkt v die Funktionalität w besitzt. Auch die bildschirmgesteuerte Bedienung eines Radio/CD-Players ist auf diese Weise fassbar: ein Bedienelement (ein Knoten v) ermöglicht das Umschalten zwischen Radioempfang (ein Knoten w) und Abspielen einer CD (Knoten u). Dieses Faktum wird als ein Graph repräsentiert, in dem die Knoten v und w sowie v und u mit Kanten verbunden sind. Ebenso gelingt die Implementierung dieses Konzeptes: Eine Mensch-Maschine-Schnittstelle kann in HTML codiert werden. Die am Bildschirm gezeigten Informationen entsprechen Medienobjekten, die mit *Tags* der Form <a href ...> – also den üblichen Links – im oben skizzierten Sinne verknüpft werden. Verschiedene HTML-Seiten zeigen dann verschiedene Informationen an und sind mit der oben beschriebenen siutationsbezogenen Darstellung von Daten vergleichbar.

Wie aber passen die objektorientierte Implementierung von GUIs und die konzeptionelle Sicht „Hyperdokument“, die in unmittelbarem Zusammenhang zur HTML-Welt steht, zusammen?

Am Lehrstuhl für Software-Technologie der Universität Dortmund ist eine objektorientierte Sprache namens DoDL (kurz für **Do**cument **D**escription **L**anguage) zur Konstruktion von Hyperdokumenten aktueller Forschungsgegenstand. Die Sprache stellt genau diejenigen Sprachkonstrukte zur Verfügung, die zur Konstruktion einfacher Hyperdokumente benötigt werden. Ferner ist ihre Semantik hinlänglich bekannt, formal beschrieben und in einem Compiler fixiert worden. Es hat sich gezeigt, dass DoDL konzeptionell tragfähig ist. Der Ansatz ist nun reif, auf seine praktische Einsetzbarkeit hin untersucht zu werden.

1.1.2 Ziele der PG

Es soll ein digitales Cockpit für Automobile konzipiert und realisiert werden. Der oben beschriebene Ansatz, grafische Benutzungsschnittstellen als Hyperdokumente aufzufassen, ist dabei essentiell. Die Projektgruppe soll das Wissen aus den Bereichen GUI-Implementierung und Hyperdokumentenerstellung kombinieren. Der objektorientierte Konstruktionsansatz von DoDL soll durch eine bekannte objektorientierte Entwurfsmethode für Hyperdokumente, beispielsweise OOHDM (siehe [74]), grafisch ergänzt werden. In der vorgestellten Projektgruppe

soll eine industrienahe Anwendung mit DoDL realisiert werden. Hierbei müssen Vorgaben für die Funktionalität des Cockpits erfasst und umgesetzt werden. Auch sind Design und Layout bei der Implementierung in einer Weise zu berücksichtigen, die das Cockpit flexibel an Kundenwünsche anpassbar machen.

Es sollte sich darüber hinaus eine Herangehensweise an Entwurf und Implementierung eines digitalen Cockpits herauskristallisieren, die sich von den üblichen objektorientierten Vorgehen zur Implementierung von Oberflächen dahingehend unterscheidet, dass Probleme wie zum Beispiel die Unüberschaubarkeit der grafischen Darstellung der Oberflächenfunktionalität gemindert oder gar behoben werden. Eine strukturierte Herangehensweise sollte auch die Wartung und Erweiterbarkeit existierender Produkte unterstützen und an zukünftige Projekte dieser Art anpassbar sein.

1.2 PG-Realisierung

Die Projektgruppenarbeit kann grob in folgende Phasen aufgeteilt werden: *Einarbeitungsphase*, *Anforderungsanalysephase*, *Entwurfsphase*, *Spezifikationsphase*, *Implementierungsphase* und, als mögliche Erweiterung, eine *Anwendungsphase*. Diese Phasen laufen teilweise parallel. Folgende Abschnitte erläutern diese Phasen im Detail.

1.2.1 Einarbeitung

In Form von Seminarvorträgen durch die Projektgruppenteilnehmer wird die Projektgruppe an die zu lösende Aufgabe herangeführt. Dies dient der Aneignung des nötigen Fachwissens. Die Einarbeitung erfolgt in folgende Themenbereiche und Problemfelder:

- Grafische Notationen (UML, etc.)
- Objektorientierte Implementierung von GUIs
- Hypermediasysteme
- Objektorientierte Modellierung von Hypermediadokument (OOHDM)
- Spezifikation von Hyperdokumenten mit DoDL
- Digitale Cockpits als Anwendungskontext

1.2.2 Anforderungsanalyse

In dieser Phase stellen die Teilnehmer die Anforderungen an das digitale Cockpit auf. Dabei sind bereits existierende Produkte zu untersuchen und daraus Anforderungen ebenfalls an die benötigten Konzepte abzuleiten.

1.2.3 Entwurf und Spezifikation

Diese Phase bearbeiten die Teilnehmer in getrennten Gruppen. Eine Gruppe entwirft das grafische, hypermediale Layout des Cockpits, die andere Gruppe stellt einen objektorientierten Entwurf für die Funktionalität des Cockpits auf. Diese Tätigkeiten können allerdings nicht unabhängig voneinander durchgeführt werden, da sie sich teilweise gegenseitig beeinflussen. Aus diesem Grund ist eine gewisse Zusammenarbeit der Teilgruppen notwendig. Hierbei werden auch fachimmanente Kommunikationsfähigkeiten der Teilnehmer geübt. Die Funktionalität soll dabei so konzipiert werden, dass sie vom Aussehen des Cockpits unabhängig und in anderen Gestaltungsvarianten einsetzbar wird. Wir verwenden UML als Basisnotation für Entwurf und Spezifikation.

1.2.4 Implementierung

Diese Phase soll den wesentlichen Teil des zweiten Semesters umschließen. Wir verwenden einen auf *eXtreme Programming* basierenden Ansatz als Vorgehensmodell zur Implementierung des Cockpits.

1.2.5 Zwischenbericht

Die gesamten Arbeiten werden jeweils durch einen *Zwischen-* und einen *Abschlussbericht* dokumentiert. Der Zwischenbericht hält die Ergebnisse des ersten Semesters in angemessener Form fest. Darüberhinaus wird in einer Seminarphase am Ende des ersten Semesters die Implementierung vorbereitet. Die nötigen Schritte werden ebenfalls im Zwischenbericht erfasst.

1.2.6 Abschlussbericht

Der Abschlussbericht wird den gesamten Projektverlauf festhalten. Die Ergebnisse der einzelnen Phasen werden vorgestellt und bewertet.

Teil I

Erste Seminarphase

Kapitel 2

Embedded Systems

Autor: *Yue Zhang*

2.1 Einleitung

Jeder weiß, dass ein PC ein Computersystem ist, weil es Berechnungen durchführen kann. Manche elektronische Anlagen, z.B. PDAs, können jedoch ebenfalls Berechnungen durchführen. Ist somit ein PDA auch ein Computersystem? Wie sieht ein solches System aus? Mit welcher Methode kann man ein Programm mit einer grafischen Benutzeroberfläche für ein solches Gerät entwickeln? Um solche Fragen zu klären, soll in diesem Dokument das sogenannte „Embedded System“ vorgestellt werden. Dazu soll zunächst der Begriff des Embedded System definiert werden. Danach werden dessen drei Ebenen, Hardware, Betriebssystem und Entwicklungsumgebung, vorgestellt. Zum Schluss wird dann noch Embedded Java besprochen.

2.2 Embedded Systems

Ein Embedded System ist ein spezielles Computersystem, das durch Informationsbearbeitungstechnologie physikalische Daten liest, bearbeitet und kontrolliert [52]. Wie jedes herkömmliche Computersystem besteht auch ein Embedded System aus einer Hardwareplattform, einem Betriebssystem und Anwendungsprogrammen. Darüber hinaus sind auch in diesem Bereich zahlreiche Entwicklungsumgebungen verfügbar.

Ein Beispiel für ein Embedded System ist das **Airbag-System** eines Autos. Ein Sensor des Autos misst ständig die Geschwindigkeit. Alle gemessenen Daten werden von dem Prozessor durch A/D Wandler eingelesen. Anhand der Daten beurteilt der Prozessor, ob ein Unfall passiert ist. Ist die Veränderung der Geschwindigkeit zu groß, so wird der Airbag ausgelöst, wodurch die Insassen vor Verletzungen bewahrt werden können. Diese ganze Regelungsanlage stellt die Hardwareplattform dar, auf der ein Realzeit-Betriebssystem (im folgenden RTOS - real time operating system) ausgeführt wird. Das Regelungsprogramm berechnet die Geschwindigkeit und alle andere notwendigen Daten. Für die Wartung und Weiterentwicklung sicher Systeme gibt es ein entsprechendes **SDK**.

Der Unterschied zwischen herkömmlichen Computersystemen und Embedded Systems liegt darin, dass erstere häufig nur einen einzigen speziellen Zweck erfüllen. Außerdem hat ein Embedded System, im Gegensatz zu herkömmlichen Computersystemen, sehr wenig Peripherie und externe Ressourcen wie beispielsweise Speicher zur Verfügung und muss innerhalb strenger Zeitlimits arbeiten.

2.3 Hardware für Embedded Systems

Als Hardware kann man alle von einem Computer benutzten elektronischen Komponenten verstehen. Sie besteht normalerweise aus folgenden Elementen[50]:

- Prozessor
- Interner Speicher
- I/O Komponenten
- Externe Speicher
- Andere spezielle Komponenten

Ein herkömmliches Computersystem lässt sich aus gebräuchlicher Hardware wie CPU, Speicher (RAM), Monitor, Tastatur, Festplatte, CD-ROM Laufwerk, Netzteil, usw. aufbauen. Die Embedded Systems basieren ebenfalls auf einer Hardwareplattform. Hier jedoch bezeichnet der Begriff „Plattform“ eine Maschine, die nicht mit einem normalen Pc vergleichbar ist.

Die Prozessoren, die zum Aufbau von Embedded Systems benutzt werden, sind meist nur für spezielle Anwendungen geeignet. Z.B. handelt es sich beim INFINEON C165UTAH nur um einen USB- und HDLC-Controller, der sich nicht für den Aufbau eines Rechners, der die typischen Aufgaben eines PCs erledigen kann eignet. Statt eines großen RAM-Speichers (jeder PC hat inzwischen 256 MByte RAM und mehr) als internen Speicher und mehrerer Festplatten und CD-ROM-Laufwerke als externe Speicher verfügen die meisten Embedded Systems nur über ROM oder EPROM-Bausteine mit weniger als 2 MByte Kapazität als internen Speicher. Sie besitzen überhaupt keine externen Speicher, da sie diese für ihre häufig sehr speziellen Aufgaben nicht benötigen. Ebenso unterscheiden sich die I/O-Komponenten von Embedded Systems deutlich von denen herkömmlicher Rechner. Ein Embedded System besitzt häufig nur Sensoren und Controller oder Regelensysteme als I/O-Komponenten. Im Gegensatz dazu werden Monitor und Tastatur als typische I/O-Komponenten bei PCs verwendet. Insgesamt sind die von Embedded Systems benutzten speziellen Geräte zahlreicher und vielfältiger.

Jedes Embedded System trägt ein individuelles Gesicht. Lediglich die prinzipielle Rechnerstruktur eines Embedded System entspricht der eines herkömmlichen Computers. Im Folgenden werden einige bedeutende Hardwarehersteller und Produkte aus dem Bereich der Embedded Systems vorgestellt.

- **Prozessoren**: Intel x86, Motorola 68000er-Serie, MIPS, PowerPC, StrongARM-Serie, SH4-Serie, Texas Instruments MSP430-Serie, usw[51].

- **I/O Controller**: 8250, standardmäßig parallele, serielle- und Drucker-Schnittstelle.
- **Graphikchips**: Zahlreiche Chips von 3dfx, ATI, C&T, Generic, Intel, Matrox, National Semiconductor, Neomagic, S3, SGS, TVIA[29].
- **Soundkarten**: Prozessor von den Firmen Creative Labs, Intel, Neomagic, TIVA, Trident, Via Technik, YAMAHA.
- **Netzwerke**: USB , Ethernet, Tulip Adapter von 3Com, AMD, Generic, Intel, Macronix, Novell und andere Firmen.
- **Schnittstellen und Protokolle**: Bluetooth, Open Services Gateway Initiative (OSGI), TCP/IP, Wireless Application Protocol (WAP), Car Automotive Network (CAN)[21], usw.

2.4 Betriebssysteme für Embedded Systems

Ein Betriebessystem ist eine vom Computer benutzte Software, die verfügbare Ressourcen an die einzelnen Komponenten eines Systems verteilt. Es lassen sich drei wichtige Anforderungen an Betriebssysteme für Embedded Systems identifizieren: Die erste ist die hohe Reaktionsgeschwindigkeit, die nötig ist, da Embedded Systems häufig unter der Vorgabe einer sehr geringen Zeitbeschränkung arbeiten müssen. Zum Zweiten müssen solche Betriebsysteme sich den zahlreichen externen Komponenten flexibel anpassen, damit die verschiedenen Embedded Systems ihren speziellen Anwendungszweck erfüllen können. Als drittes ist der direkte I/O-Zugriff eine wichtige Eigenschaft.

Herkömmliche Betriebssysteme wie Windows NT und UNIX sind für diese Anforderungen zu langsam und überladen. Hier sind unterschiedliche RTOS besser geeignet. RTOS sind sehr einfach gehalten, arbeiten dafür aber mit einer hohen Geschwindigkeit. Manchmal wird dabei sogar statisches Scheduling verwendet, um die Geschwindigkeit zu erhöhen. Ein RTOS ist dabei flexibel genug, um sich verschiedenen Komponenten anzupassen. Im Folgenden werden drei typische RTOS vorgestellt.

2.4.1 Embedded Linux

Bei Embedded Linux handelt es sich um eine Open-Source Software, man kann also ohne weiteres auf den Quellcode zugreifen. Dieses System hat sehr geringe Anforderungen an den benötigten Speicherplatz, es ist flexibel, schnell, effizient und kostenlos. Allerdings gibt es kein Unternehmen, das für die Entwicklung und Wartung des Systems zuständig ist. [49]Dies ist der einzige, allerdings sehr große Nachteil von Linux.

2.4.2 QNX

QNX ist Linux sehr ähnlich und besitzt einen sehr großen Marktanteil. Das System einer kanadischen Firma unterstützt eine große Bandbreite häufig verwendeter Hardware. Die Firma

bietet sehr guten Service und technische Unterstützung für die Kunden. Als Nachteile sind die Kosten sowie die Abhängigkeit von einem Unternehmen zu nennen.

2.4.3 Windows CE

Windows CE ist eine Variante der Windows Serie[56]. Ein Vorteil dieses Systems ist, dass es vom stärksten Softwarekonzern vertrieben wird und zahlreiche Entwicklungsumgebungen von Microsoft zur Verfügung stehen. Die Unternehmensabhängigkeit ist auf der anderen Seite aber auch ein Nachteil. Darüber hinaus kann Windows CE eigentlich nicht als echtes RTOS gelten, da seine Reaktionsgeschwindigkeit oft zu gering ist. Aus diesem Grunde wird es auch als Soft-RTOS bezeichnet[52].

2.5 Entwicklungsumgebungen für Embedded Systems

Für die Entwicklung von Anwendungsprogrammen für Embedded Systems kann Assembler eingesetzt werden. Die am häufigsten benutzte Programmiersprache ist allerdings Embedded C. Fast alle Firmen haben individuelle Varianten der Sprache C für ihr jeweiliges System entworfen. Manche Firmen bieten sogar die Programmiersprache C++ als ein Werkzeug für die Softwareentwicklung für Embedded Systems an. Die meisten Embedded Systems haben standardmäßig keine grafischen Ausgabegeräte wie Monitor, Printer oder FDP. Aus diesem Gund haben Embedded Systems auch keine standardmäßige GUI-Entwicklungsumgebung. Manche Firmen bieten den Kunden eine Reihe systemorientierter SDKs an, um grafische Benutzeroberflächen für ihre Anwendungen entwickeln zu können. Zur Beispiel bietet QNX ein GUI-System, QNX Photon microGUI[68], an. Oder die Firma Windriver bietet auch eingne SDK[82]. Für bestimmte Systeme kann man auch populäre Entwicklungswerkzeuge wie Qt, Visual Studio oder Java einsetzen.

2.5.1 Qt

Qt ist eine C++-Bibliothek für die Entwicklung von grafischen Benutzeroberflächen[78]. Man kann Qt unter Linux, Windows, Unix und anderen Betriebsystemen einsetzen[7]. Zur Zeit bietet Trolltech, der Hersteller von Qt, auch die Variante Embedded Qt auf dem Markt an. Der Vorteil von Qt ist, dass es sehr schnell arbeitet. Außerdem gilt Qt als ein sehr ausgereiftes System. Ein wesentlicher Nachteil von Qt sind sind die hohen die Kosten für die Embedded-Version. Außerdem ist Embedded Qt wenig portabel , da es lediglich auf Embedded Linux läuft[79].

2.5.2 Visual Studio

Hierbei handelt es sich um ein traditionelle Entwicklungswerkzeug für Windows-Systeme. Laut Microsoft kann man Visual Studio auch für die Entwicklung von Windows CE-Programmen benutzen[55]. Vorteile von Visual Studio sind die benutzerfreundliche IDE sowie die Unterstützung der Plattform durch Microsoft. Als Nachteile sind die Plattformabhängigkeit, die Größe

der Programme und die schlechten Realtime-Eigenschaften von Windows CE zu nennen.

2.5.3 Java

Java ist eine weitere mögliche Lösung. Der Vorteil von Java ist die Plattformunabhängigkeit, getreu dem Motto „Write once, run anywhere“. Java ist eine einfache OO-Programmiersprache. Java ist kostenlos und wird von der Firma Sun unterstützt. Daher ist Java sehr populär. Der Nachteil von Java ist die geringe Geschwindigkeit. Wegen der JVM und des nicht vorhandenen direkten Hardwarezugriff sind Java-Programme langsamer als C und C++-Programme.

2.6 J2ME

Java ist eine sehr populäre objektorientierte Programmiersprache. Der Erfinder, die US-amerikanische Firma Sun Microsystems, bietet den Programmierern drei Versionen an, nämlich J2EE, J2SE und J2ME. J2ME oder „Java 2 Micro Edition“[44] ist die Java-Variante für Embedded Systems. Von ihr existieren zwei Varianten, CDC und CLDC. Diese unterstützen zahlreiche Embedded Systems. J2ME unterscheidet sich durch drei Merkmale von den anderen beiden Versionen. Dies sind die unterstütze Hardwareplattform, die verwendete JVM und die zur Verfügung stehenden Pakete.

Verschiednen Versionen von Java sind für unterschiedliche Anwendenszwecke und Maschinen geeignet.

- J2EE ist passend für Systeme in Unternehmen. Es wird häufig auf der Serverseite verwendet. Solche Server besitzten häufig eine rechenstarke CPU (z.B. einen 64 Bit-Prozessor) und viel Speicher (mehr als 10 MByte).
- J2SE ist die passende Lösung für private Anwendungen. Es läuft oft auf PCs oder Notebooks.
- J2ME ist die Java-Variante für Embedded Systems.

CDC ist für Geräte mit 32 Bit-Prozessor und mindestens 512 KByte Speicher geeignet, z.B. für PDAs[44].

CDLC läuft auf schon auf Geräten mit 16 Bit-Prozessor und ca. 160KByte Speicher, also z.B. Handys[44].

Die J2ME verwendet zwei Varianten der JVM. Die eine ist die CVM für CDC. Die Andere ist die KVM für CLDC[44]. CLDC ist eine echte Teilmenge der CDC, d.h. alle Programme, die auf der KVM laufen, lassen sich auch auf CVM ausführen.

Die J2ME besitzt alle wesentlichen Klassen der J2SE. Außerdem besitzt die J2ME einige neue Pakete, die jeweils mit „javax.microedition“[44] beginnen. Allerdings liegen viele Pakete und Klassen bei der J2ME in einer vereinfacheten Variante vor. J2SE hat ein GUI-Paket, das AWT. Im Gegensatz dazu hat die J2ME keine mächtige GUI-Bibliothek und sie unterstützt

die Pakete AWT und Swing der J2EE und J2SE nicht. Speziell für PDA-Entwickler bietet die J2ME eine vereinfachte GUI-Bibliothek namens „KJAVA“. Sie ist ähnlich aufgebaut wie AWT, aber nicht so komplex und leistungsstark.

Die **Vorteile** der J2ME sind natürlich zum einen alle Vorteile von Java[76], d.h. Plattformunabhängigkeit, der saubere objektorientierte Aufbau, die Stabilität der Programme, die Unterstützung aus Sun und IBM[43], usw. Die **Nachteile** sind auch sehr klar. Zum einen ist die Geschwindigkeit der I/O-Zugriffe, zum anderen gibt es auch kein herkömmliches GUI-Paket, da die J2ME auf kein bestimmtes Windowing-System ausgelegt ist.

In unserem Projekt muss jedoch eine GUI entwickelt werden, so dass die Benutzung von J2ME hier wenig empfehlenswert ist.

Kapitel 3

Anwendungsdomäne „Hypermediales Cockpit“

Autor: *Tobias Wolf*

3.1 Einleitung

In diesem Text soll dargelegt werden, aus welchen Bestandteilen ein „hypermediales Autocockpit“ besteht. Dabei liegt hier der Schwerpunkt auf dem Autocockpit mit seinen diversen Komponenten und nicht auf seiner Modellierung als Hypermediadokument. Meine Ausführungen stützen sich hierbei auf die Spezifikationen der AMI-C.

Im folgenden wird kurz die Vereinigung AMI-C vorgestellt und in groben Zügen die Architektur [1] erklärt. Anschließend werden die grundlegenden Funktionen des Fahrzeugsinterface [3] aufgeführt. Schließlich wird noch kurz auf die Modellierung des Netzwerkes [4] eingegangen. Abschliessend wird eine Übersicht über die, von AMI-C, vorgeschlagenen Use Cases[2] gegeben, von denen ein paar detaillierter wiedergegeben werden. Bei der Benennung der Use Cases orientiere ich mich an der Namensgebung durch AMI-C.

3.2 AMI-C

Die „Automotive Multimedia Interface Collaboration,Inc“ ist eine Vereinigung die aus Herstellern (Fiat, Ford, GM, Hyundai, Peugeot, Renault, Nissan, Toyota), Zulieferern (Sun, Alpine, Motorola, Toshiba, NavTech ...) und anderen Vereinigungen besteht. Sie verfolgen die Vision ein einheitliches System für Multimediageräte in Fahrzeugen zu verwirklichen, das von allen Herstellern und Zulieferern genutzt wird. AMI-C formuliert ihre Mission in folgenden vier Punkten:

- Kostenreduzierung für Multimediageräte
- Die Zufriedenheit des Kunden erhöhen

- Sichere Bedienung der Geräte ermöglichen
- Erhöhung der Qualität

3.3 Die Architektur

Damit ein System als AMI-C konform gilt, hat es folgende Eigenschaften zu erfüllen:

- **Flexibilität**

  Es wird eine große Bandbreite von Geräten mit den unterschiedlichsten Ansprüchen geben, die das System zu verwalten hat.

- **Erweiterbarkeit**

  In ein bestehendes System wird während seiner Lebenszeit immer wieder neue Software und Hardware installiert werden müssen.

- **Universelle Benutzbarkeit**

  Geräte sollen nicht speziell für ein Fahrzeug konzipiert werden, sondern in allen AMI-C konformen Systemen benutzbar sein.

- **Auf- und Abwärtskompabilität**

  Diese Forderung folgt aus der Erweiterbarkeit und der universellen Benutzbarkeit. Es sollen sowohl zukünftige als auch ältere Komponenten einsetzbar sein.

Es werden folgende Bestandteile vorgeschlagen:

- **mind. ein AMI-C-Netzwerk**

  über das Netzwerk kommunizieren die verschiedenen Geräte miteinander. Es wird vorgeschlagen ein schmalbandiges Netzwerk mit einer sicheren übertragung einzusetzen um kritische Komponenten miteinander zu verbinden und ein breitbandiges Netzwerk für unkritische Anwendungen wie Videostreaming usw. zu realisieren.

- **das Fahrzeuginterface**

  über das Fahrzeuginterface können die Benutzer das System bedienen, bzw. Informationen vom System bekommen.

- **die AMI-C-Komponenten**

  Hier werden verschiedene Ausprägungen unterschieden, je nach ihren Fähigkeiten werden sie

  - Geräte, die eine bestimmte Funktion im System erfüllen,
  - Controller, die Geräte steuern, oder
  - Hosts, die von anderen Komponenten benutzt werden,

  genannt.

3.4 Das Fahrzeuginterface

Hier können diverse Gruppen von Dienstleistungen unterschieden werden, von denen jeweils eine Auswahl hier genannt werden soll.

3.4.1 Basisleistungen

Die Basisleistungen stellen die Gruppe von Leistungen dar, die in jedem AMI-C konformen System zwingend enthalten sein müssen.

- VIN

  die Vehicle Information Number dient zur Identifizierung des Fahrzeugs
- Hersteller/Modell
- Fahrzeugbeschreibungsdaten

  darunter fällt insbesondere eine physische Beschreibung des Fahrzeugs und der verfügbaren Systeme
- AMI-C Versionsnummer
- Verfügbare und konfigurierte Dienstleistungen

  hiermit sind z.B. Sicherheitsdienste, GPS und Fahrzeugdiagnostik gemeint
- Datum und Zeit
- Systemstatus

  Schlafmodus, Aktiv, Bootmodus, Shutdownmodus
- Energiestatus

  hier können andere Geräte über einen Wechsel im Energiestatus benachrichtigt werden, damit sie gegebenenfalls entsprechend darauf reagieren können
- Bootsequenz-Nachrichten
- Shutdownsequenz-Nachrichten
- Netzwerkaktivitätsnachrichten und Fehlerstatus

  Damit sind insbesondere Informationen über Kollisionen oder verlorene Pakete gemeint
- AMI-C Diagnosen

  Dieser Punkt ist in der AMI-C Spezifikation 1 noch recht allgemein gehalten. Es soll hier zumindest festgehalten werden, das es einen Diagnosemodus geben kann, der wichtige Informationen über den Zustand des Systems liefert

3.4.2 Sicherheitsdienstleistungen

Die Gruppe umfasst Authentifizierungsprotokolle, Verschlüsselungsprotokolle und Zugangsprotokolle. Sie werden für vier Bereiche genutzt:

- Diebstahlschutz

  Installierte Geräte funktionieren nicht in anderen Fahrzeugen, wenn die Sicherheitsidentifizierung misslingt.

- Geräteauthentifikation

  Nur AMI-C konforme Geräte sollen in einem AMI-C System funktionieren.

- Schutz von Copyright geschützten Medien wie Video und Musik.

- Fahrer- und Benutzerauthentifikation und Schutz der persönlichen Daten.

3.4.3 Fahrzeugstatusdienstleistungen

Damit sind Informationen über den aktuellen Zustand des Fahrzeugs gemeint. Die Informationen können auf Displays angezeigt, aber auch von anderen Geräten abgefragt werden.

- Geschwindigkeit
- Kilometerstand
- Tank
- Batterie
- Airbag
- Bremsen
- Traktionskontrolle
- Warnlichter

3.4.4 Motorstatus

Die Dienstleistungen über den Motorstatus befinden sich in einer Grauzone, da sie eigentlich nicht mehr von der AMI-C Spezifikation abgedeckt werden sollen, einige Funktionen werden allerdings von Komponenten benötigt, z.B. das Abschalten des Motors bei diversen Sicherheitsdienstleistungen.

- „Motor kann gestartet werden“ - Status
- Motor abschalten
- Motorleistungsstatus/-kontrolle

- Fernstarten
- Umdrehungszahl
- Schaltung
- öldruck

3.4.5 Unterhaltungssystem

Da das Unterhaltungssystem sehr vielfältig ist, seien hier nur einige grundlegende Funktionen genannt, die von den meisten Medien benötigt werden.

- Audiokontrolle
- Lautstärkeregelung (sowohl für alle Lautsprecher, als auch getrennt für einzelne Lautsprecher im Wagen)
- Stummschaltung
- Verbinden von Audiokanälen (damit ist auch das Weiterleiten von Audioquellen an bestimmte Ziele im Wagen gemeint)
- Radio
- Videospiele
- DVD/TV
- Internet

3.4.6 Monitore

Im Fahrzeug wird es mindestens einen Monitor für den Fahrer geben, in der Regel aber auch für jeden Insassen einen eigenen Monitor. Hier werden diverse Informationen angezeigt:

- Textanzeigen
- Geschwindigkeitsanzeigen
- ISO Anzeigen (Blinker usw.)
- Telefonwählknopf
- Graphische Anzeigen
- DVD/TV Wiedergabe
- Anzeigen der Videospiele

3.4.7 Sonstiges

Alle weiteren Komponenten deren Informationen durch das Fahrzeuginterface angezeigt oder kontrolliert werden, sollen hier noch kurz erwähnt werden.

- Türen (Verschlussstatus und Verriegelungskontrolle)
- Kofferraum
- Fenster
- Schiebedach
- Scheinwerfer/Blinker/Innenraumbeleuchtung
- Warnlicht
- Rückspiegel
- Sitze
- Antenne
- Reifenluftdruck
- Sicherheitsgurte
- Hupe
- Sicherheitsalarm
- Scheibenwischer
- Innen-/Außentemperatur

3.5 Netzwerk - Das Common Message Set

In Abschnitt 3.4 wird deutlich das viele Dienstleistungen zu erbringen sind und das eine Menge Informationen anzuzeigen, bzw. zwischen verschiedenen Geräten auszutauschen sind. Mit der Spezifikation des Common Message Set hat AMI-C versucht möglichst viele Geräte zu erfassen und darzulegen welche Nachrichten diese Geräte empfangen bzw. senden können. Um sich nicht schon auf eine spezielle Implementation festzulegen wurde ASN.1, die Abstract Syntax Notification benutzt. Damit kann man auf einem hohen Level die Nachrichten beschreiben, die dann später auf die jeweiligen Netzwerkprotokolle umgesetzt werden können. Die Entscheidung für ASN.1 wurde durch vier Forderungen getroffen, die in der Form nur ASN.1 unterstützt:

1. Der Nachrichtenbeschreibungsmechanismus soll auf einem hohen Level sein und muss in Bitmuster konvertierbar sein
2. Der Mechanismus muss unabhängig vom einzusetzenden Protokoll sein
3. Es sollen kurze Nachrichten generiert werden
4. Sie muss robust sein

3.6 Use Cases

In diesem letzten Abschnitt folgt eine komplette Auflistung der von AMI-C vorgeschlagenen Use Cases. Einige wenige davon werden detaillierter beschrieben um einen Eindruck zugeben, welche Interaktionen und Besonderheiten in einem multimedialen Autocockpit vorherrschen.

- COMB 1: „Combination Use Cases“
- COMB 1.1: „Multimedia Mini Van“

  Beschreibung:

  Hier wird die gleichzeitige Benutzung mehrerer AMI-C Geräten in einem Wagen für 6 Passagiere demonstriert. Dafür ist das Fahrzeug mit folgenden Geräten ausgestattet:

  - Navigationssystem, mit graphischer Anzeige und akustischer Abbiegeansage
  - DVD Player
  - Radio
  - Videospiel
  - Mobiles Telefon mit Freisprechanlage
  - Videoanzeigen in der Frontkonsole und in den Rückseiten der Sitze zur individuellen Benutzung durch die Mitfahrer.

  Ereignisse:

  1. Der Fahrer teilt dem Navigationssystem den Zielort mit und stellt die Abbiegeansage ein. Auf dem Fahrermonitor wird die aktuelle Position des Fahrzeugs angezeigt.
  2. Danach wählt der Fahrer das Radio. Dieses erscheint im Monitor und der Fahrer kann einen Sender einstellen. Die Fahrzeuglautsprecher werden zum Abspielen des Radioprogramms benutzt. Nach kurzer Zeit erscheinen wieder die Informationen des Navigationssystems auf dem Bildschirm.
  3. Zwei Mitfahrer entscheiden sich ein Videospiel zusammen zu spielen. Dazu benutzt jeder sein eigenes Display und Spielkontrollen. Der Ton wird über Kopfhörer abgespielt.
  4. Ein Mitfahrer will einen DVD Film schauen. über sein Display wählt er die DVD Applikation und beginnt den gewünschten Film zu schauen. Auch hier wird der Ton per Kopfhörer ausgegeben.
  5. Ein anderer Mitfahrer will diesen Film auch schauen und wählt auf seinem Display die DVD-Applikation. Hier kann er wählen ob er einen anderen Film schauen will, oder den der gerade abgespielt wird. Er wählt letzteres und Bild und Ton werden nun auch zum ihm geleitet.
  6. Das Navigationssystem will dem Fahrer mitteilen das er bald abbiegen muss. Dazu wird der Ton des Radios leiser gestellt. Die Ankündigung wird sowohl akustisch als auch graphisch bekannt gegeben. Danach wird die Radiolautstärke wiederhergestellt.

7. Ein Anruf kommt an. Das Radio wird leise gestellt und im Display wird die Nummer des Anrufers dargestellt.
8. Der Fahrer nimmt den Anruf entgegen und erkennt das der Anruf für einen Beifahrer bestimmt ist. Also stellt er den Anrufer zu dem entsprechenden Beifahrer durch.
9. Der Mitfahrer bekommt eine Meldung über den Telefonanruf und kann das Videospiel pausieren. Dann nimmt er das Gespräch an. Es führt das Gespräch per Kopfhörer und eigenem Mikrophon. Nachdem er das Gespräch beendet hat, kann das Videospiel weitergespielt werden.
10. Das Fahrzeug erkennt, das der Bezinstand niedrig ist und alarmiert den Rest des System. Dadurch wird ein Warnanzeige im Display aktiviert.
11. Das Navigationssystem sucht auf Grund der aktuellen Position eine Auswahl von möglichen Tankstellen und berechnet auf Grund der Wahl des Fahrers die neue Route.
12. Das Fahrzeug kommt an der Tankstelle an und der Motor wird abgeschaltet. Dies führt dazu das Radio, Videospiel und der Film stoppen. Das Fahrzeug wird nun betankt.
13. Sobald das Tanken beendet ist und der Motor wieder gestartet wird kehren alle Geräte wieder in ihren alten Zustand zurück.
14. Der Film endet und einer der Passagiere wählt die Ansicht des Navigationssystems um sich die aktuelle Position des Fahrzeugs anzusehen.
15. Der andere Passagier entschließt sich das gerade laufende Videospiel anzusehen.
16. Der Zielort wird erreicht und dieser Use Case endet.

- COMB 1.2: „CD Audio, Mobile Phone & Audio Navigation Interaction“
- COMB 1.3: „CD and Phone Mixed Audio“
- COMB 1.4: „Shared Display Example“

  Beschreibung: Hier wird die Benutzung des Displays demonstriert.

  Ereignisse:

  1. Zuerst wird das Navigationssystem aktiviert und gibt das Ziel und die Anzeigeoptionen ein.
  2. Das Display zeigt eine Karte und in einem eigenen Bereich die Abbiegeinformationen an.
  3. Dann werden die Kontrollen der Klimaanlage aktiviert, dabei wird ein Teil der Navigationsanzeigen überlagert.
  4. Der Fahrer kann nun Einstellungen vornehmen.
  5. Während der Fahrt kommt eine Wetterwarnung an, die als extra Fenster im Display angezeigt wird.
  6. Bei Berührung des Alarms, erhält man die Wahl das Fenster zu schließen oder weitere Informationen anzeigen zu lassen.

7. Die weiteren Informationen füllen dann das gesamte Display.
8. Der Wetteralarm wird geschlossen und die vorherigen Informationen werden wieder angezeigt.

- COMM 1: „Parking Garage Fee Payment“
- COMM 2: „Road Toll Payment while Vehicle is moving“
- COMM 3: „Interactive Payment Authorization for Goods/Services“
- COMM 4: „Food Discovery & Purchase“
- COMM 5: „DVD Movie Rental while Fuelling“

  Beteiligte:

  In diesem Use Case interagieren das Fahrzeug/System, der Fahrer. Weiterhin werden folgende Systeme genutzt:

  – Hochgeschwindigkeits-Nahbereichs-Funknetzwerk
  – elektronisches Bezahlsystem
  – Massenspeicher im Fahrzeug

  Beschreibung:

  Während des Tankens kann der Fahrer ein Video auswählen und kaufen, um es entweder im Fahrzeug oder zu Hause zu schauen.

  Ereignisse:

  1. Das Fahrzeug stoppt an einer Tankstelle die Benzin und digitale Filme anbietet.
  2. Eine Hochgeschwindigkeitsverbindung wird hergestellt.
  3. Es werden Authorisationsdaten und Bezahlinformationen ausgetauscht.
  4. Der Fahrer beginnt mit dem Tanken und kann dabei die Filmdatenbank sichten.
  5. Er wählt einen Film aus und bestätigt die Bezahlung.
  6. Der Film wird übertragen und im Fahrzeug abgespeichert.
  7. Tanken und die Datenübertragung enden.
  8. Der Fahrer fährt weiter.

- CUST 1: „Acquisition of Status“
- CUST 2: „Commercial Information from Dealer to Customers“
- CUST 3: „Shared Information and Function among Family‘s or Friends Cars“
- CUST 4: „Move Personal Data/Function to new Car by Service Provider“

  Beschreibung:

  Es werden persönliche Daten und Software über einen Service Provider vom alten Fahrzeug zu einem neuen Fahrzeug übertragen.

  Ereignisse:

1. Der Fahrer wählt die „Send Data/Software to other Car“ Funktion.
2. Er gibt seine PIN ein.
3. Er wird per Display und akustische Signale über den Status der übertragung informiert.
4. Die übertragung zum Provider ist beendet.
5. Im neuen Fahrzeug wird die „Get Data/software from other Car“-Funktion aktiviert.
6. Die PIN wird eingegeben.
7. Das Status der übertragung wird angezeigt und mit Beendigung der übertragung endet dieser Use Case.

- CUST 5: „Wide Area general Cooperation“
- EMER 1: „Vehicle Accident (Automated Call for Help)“

  Beschreibung:

  Nach einem Unfall tritt ein automatisches SOS System in Kraft

  Ereignisse:

  1. Das Fahrzeug ist in einen Unfall verwickelt.
  2. Das SOS System ermittelt die Schwere des Unfalls.
  3. Es wird versucht mit den Insassen Kontakt aufzunehmen.
  4. Wenn das System keine Antwort bekommt wird eine Verbindung zu einem Notfallcenter aufgenommen und die Fahrzeugkoordinaten übermittelt.
  5. Im Notfallcenter werden die Daten ausgewertet und die nötigen Hilfskräfte informiert.

- EMER 2: „Emergency Call with Diagnostics“
- EMER 3: „Manual Call for Help“
- ENTE 1: „Continuous Feed Internet Audio“
- ENTE 2: „Continuous Feed Video & Audio“
- ENTE 3: „Web Browsing“

  Beschreibung:

  Ein Mitfahrer benutzt das Internet aus dem Fahrzeug heraus.

  Ereignisse:

  1. Der Webbrowser wird gestartet.
  2. Die Anwendung wird auf dem, vom Benutzer gewähltem, Display angezeigt.
  3. Es wird eine Internetverbindung hergestellt.
  4. Der Benutzer interagiert mit dem Internet.
  5. Der Webbrowser wird geschlossen.
  6. Die Internetverbindung wird abgebaut.

- ENTE 4: „Broadcast Digital/Analog Audio“
- ENTE 5: „Audio Media Player (Cassette, DVD, CD)“
- ENTE 6: „Broadcast Analog/Digital Video“
- ENTE 7: „Video Media Player (DVD, VCR)“
- ENTE 8: „Video Games in Vehicle“
- FLEE 1: „Vehicle Status“
- FLEE 1.2: „Vehicle Status- Ad Hoc Request, Proximity Device“

  Beschreibung:

  Das Fahrzeug liefert Routenaufzeichnungen, Service Informationen, Fahrzeugdaten an eine Werkstatt, wenn ein Annäherungssensor das Fahrzeug registriert und eine entsprechende Anfrage stellt. Mit diesem Mechanismus können Daten abgefragt werden wenn sich das Fahrzeug in einer Werkstatt befindet, oder seine Garage erreicht/verlässt.

  Ereignisse:

  1. Das Fahrzeug erreicht eine Werkstatt und wird von einem Annäherungssensor erfasst.
  2. Der Sensor unterrichtet das Werkstattsystem von der Ankunft des Fahrzeugs.
  3. Das Werkstattsystem authentifiziert sich beim Fahrzeug.
  4. Anschließend werden die Daten des Fahrzeugs übertragen.
  5. Das Werkstattsystem unterrichtet das Fahrzeug über die erfolgreiche übertragung der Daten.
- FLEE 2: „Software Installation“
- FLEE 3: „Mission Planning“
- FLEE 4: „Report Mission Status“
- FLEE 5: „Commercial Route Management“
- GUID 1: „Navigation On-Board System“
- GUID 2: „Navigation off-board System“
- GUID 3: „Navigation with Communication System“
- GUID 4: „Traffic Information using Multiplex Broadcasting“
- GUID 5: „Roadside Traffic Camera“

  Beschreibung:

  Das Fahrzeug fordert ein Bild der Straße an, das von einer Kamera am Straßenrand aufgenommen wurde.

  Ereignisse:

1. Der Fahrer wählt das Verkehrsinformationssystem.
2. Er wählt aus dem Informationsangebot den Straßenbildservice.
3. Die Position des Fahrzeugs wird ermittelt und an den Service übertragen.
4. Als Antwort erhält der Fahrer eine Liste von Kameras die auf dem Weg liegen.
5. Der Fahrer wählt die gewünschte Kamera.
6. Das gewünschte Bild wird übertragen und auf dem Display angezeigt.
7. Der Fahrer kann weitere Kameras auswählen, evtl. auch für andere Routen.
8. Die Bilder der neuen Auswahl werden geladen und angezeigt.
9. Der Fahrer beendet den Bild Service.

- GUID 6: „Obtaining Roadside Information“
- HOME 1: „Home Activation of Lights, Door Locks and Media from Vehicle“
- HOME 2: „Home Status Interrogation from Vehicle“
- INFO 1: „Personal Information Service from Service Provider“
- INFO 2: „Vehicle Information to Service Provider“
- MESS 1: „Video Phone“
- MESS 2: „Paging“
- MESS 3: „Fax“
- MESS 4: „E-mail - Receive and Send“
- MESS 5: „Audio Chat“
- MESS 6: „Voice Mail Access“
- MOBI 1: „Smart Card Access“
- MOBI 2: „Mobile Phone Voice Call“
- MOBI 3: „Mobile Phone Data Call“
- MOBI 4: „Mobile Device Messaging“
- MOBI 5: „Mobile Device Capability Sharing“
- MOBI 6: „Mobile Device Synchronisation“
- PREF 1: „Personal Preferences“
- PREF 2: „Personal Privileges“
- PROD 1: „Access Productivity Applications“
- PROD 2: „Dictation“

- SECU 1: „Theft of AMI-C Components from Vehicle“
- SECU 2: „Vehicle Theft Countermeasures“
- SECU 3: „Remote Door Status and unlocking“
- SERV 1: „Online User/Owner Manuals“

  Beschreibung:

  Der Benutzer wählt die Website des Fahrzeugs um Service-Informationen, Handbücher oder Garantie-Informationen zu sehen. Die Informationen werden auf dem Display angezeigt.

  Ereignisse:

  1. Der Fahrer wählt die URL des Dokumentenservice.
  2. Es wird eine Internetverbindung aufgebaut und die Site angewählt.
  3. Es wird der Index der vorhandenen Dokumente angezeigt.
  4. Der Fahrer wählt das gewünschte Dokument.
  5. Durch ein Menüsystem angeleitet, findet der Fahrer die gewünschte Information.
  6. Anschließend beendet der Fahrer den Dienst und der Hauptbildschirm wird wieder angezeigt.
  7. Die Internetverbindung wird abgebaut.
- SERV 2: „Service Technician installs new Device/Function“
- SERV 3: „Diagnose System Fault“
- SERV 4: „Software Applications Downloaded“

  Beschreibung:

  Der Kunde überträgt eine neue Anwendung auf sein Fahrzeug.

  Ereignisse:

  1. Der Kunde aktiviert die Quelle der Software und trifft seine Auswahl.
  2. Der Download der Software wird gestartet.
  3. Im Anschluss wird die Software automatisch installiert und gestartet.
  4. Der Kunde benutzt die neue Anwendung oder beendet die Software.
- SERV 5: „Subscribe to new Service“
- USER 1: „Shared Multifunction Display (With Vehicle Manufacturer Functions)“
- USER 2: „Shared Character Display (With Vehicle Manufacturer Functions)“
- USER 3: „Shared Audio (With Vehicle Manufacturer)“
- USER 4: „User Input via Switch/Buttons (no Voice recognition)“
- USER 5: „User Input Control via Voice Recognition System“

- USER 6: „Shared Multifunction Head-Up Display“
- USER 7: „User Input Control via Universal Remote Control“
- USER 8: „Driver Controls Input from Buttons on Steering Wheel“

3.7 Fazit

In einem modernen Autocockpit wird sich eine Vielzahl von alten, bekannten Funktionen mit neuen Funktionen mischen. Durch neue Technologien muss die Visualisierung und Steuerung der alten Funktionen neu überdacht werden und die neuen Funktionen möglichst sinnvoll in das Automobilumfeld integriert werden. Dabei spielen Aspekte wie Sicherheit und Bedienbarkeit eine große Rolle. Vor allem die Use Cases geben einen Eindruck von großen Anzahl der Möglichkeiten und der Komplexität der Kommunikation zwischen den Geräten untereinander und zwischen System und Anwender.

Kapitel 4

GUIs mit Swing

Autor: *Stefan Borggraefe*

4.1 Einführung

Swing ist die aktuelle GUI-Bibliothek für Java. Sie enthält Klassen für die Erstellung der unterschiedlichsten grafischen Oberflächenelemente. Die Spanne reicht von einfachen Elementen, wie z.B. Buttons oder Pull-down-Menüs bis hin zu komplexen Dingen wie Tabellen und Bäumen. Eine Übersicht über die Standard-Komponenten der Swing-Bibliothek ist unter [17] online verfügbar. Die Swing Klassen sind in den Paketen, die mit javax.swing beginnen, enthalten.

Swing benutzt *Layout-Manager*, um *Komponenten* innerhalb von *Containern* anzuordnen. Die typische Vorgehensweise bei der Programmierung von Swing-basierten Benutzeroberflächen sieht folgendermaßen aus:

1. Komponenten erzeugen. Beispielsweise JButtons oder JLists.
2. Container erzeugen und einen Layout-Manager zuordnen. Eine typische Container-Klasse ist JPanel.
3. Komponenten dem Container hinzufügen, evtl. mit Optionen (layout constraints), die die Behandlung durch den Layout-Manager beeinflussen.
4. Container sind auch Komponenten, können also übergeordneten Containern zugeordnet werden. So lassen sich komplexe Oberflächen durch Schachtelung und Benutzung verschiedener Layout-Manager erzeugen.

Layout-Manager ordnen die Komponenten nach bestimmten Algorithmen innerhalb von Containern an. Diese Algorithmen können beim hinzufügen der Komponenten durch sog. layout constraints beeinflusst werden.

Damit die so erstellte Benutzeroberfläche eine sinnvolle Funktion erfüllen kann, müssen an ihr auftretende Benutzereingaben durch das Verarbeiten von Events behandelt werden (s.

Abschnitt 4.2) und die Oberfläche mit den darzustellenden Anwendungsdaten verbunden werden (s. Abschnitt 4.3). Im weiteren Verlauf von Abschnitt 4.1 werden zunächst noch einige grundlegendere Eigenschaften von Swing besprochen.

4.1.1 Swing und AWT

Bevor Swing entwickelt wurde, wurde für die Erstellung grafischer Benutzeroberflächen mit Java standardmäßig das AWT (Abstract Window Toolkit) verwendet. Allerdings hatte diese Bibliothek einige gravierende Schwächen, die mit der Entwicklung von Swing überwunden werden sollten. Um einige zentrale Konzepte von Swing besser verstehen zu können, ist es nach der Meinung des Autors sinnvoll, es mit AWT zu vergleichen. Dadurch können einige Design-Entscheidungen, die bei der Entwicklung von Swing getroffen wurden, besser nachvollzogen und somit ein besseres Verständnis für den Aufbau dieser GUI-Bibliothek erlangt werden.

Java ist eine plattformunabhängige Programmierumgebung. Das bedeutet für die GUI-Entwicklung unter Java, dass man die Oberflächen-Funktionen der einzelnen Betriebssysteme nicht direkt ansprechen kann, sondern plattformunabhängige Java-Klassen benutzen muss. Diese Klassen stellen die notwendige Abstraktionsschicht zwischen den einzelnen Betriebssystemen und der plattformunabhängigen Java-Umgebung zur Verfügung. Sowohl Swing als auch AWT stellen eine solche Klassensammlung dar.

Da Benutzeroberflächen jedoch dazu da sind, Ausgaben auf konkreter Hardware darzustellen und Eingaben von konkreten Geräten zu verarbeiten, muss an irgend einer Stelle die Grenze zwischen der plattformunabhängigen Java-Welt und der rauhen Welt der realen Rechner mit den unterschiedlichsten Betriebssystemen und Hardwareausstattungen überschritten werden. Dazu müssen auf einer bestimmten Ebene letztlich Java-Objekte vorhanden sein, die durch native Methoden mit dem Host-System interagieren. Der Hauptunterschied zwischen Swing und AWT liegt darin, auf welchem Abstraktionsniveau sich diese Ebene befindet.

Bei AWT wird die Plattformunabhängigkeit dadurch gewährleistet, dass austauschbare sog. *Toolkits* verwendet werden, die mit dem jeweiligen Wirts-Betriebssystem kommunizieren, um die Oberflächenelemente darzustellen. Diese Toolkits implementieren dabei eine Menge von sog. *Peer*-Interfaces, die die Schnittstelle zu den Java-Klassen mit den nativen Methoden darstellen. Durch diese Peer-Interfaces wird es für plattformunabhängige AWT Oberflächen-Komponenten möglich, die entsprechenden plattformspezifische Komponenten der einzelnen Betriebssysteme zu nutzen. Möchte man beispielsweise mittels AWT in einem Java-Programm einen Button erzeugen, wird beim Ablauf des Programms das passende betriebssystemspezifische Toolkit über die definierte Schnittstelle der Peer-Interfaces damit beauftragt, einen Button über Betriebssystemfunktionen zu erzeugen (s. Abbildung 4.1). Dadurch erhält man unter UNIX einen Motif-Button, unter Mac OS einen Mac-Button usw.

Was sind nun die oben angedeuteten gravierenden Nachteile dieser Architektur von AWT? Durch die Verwendung plattformspezifischer Toolkits, kommt es in der Praxis häufig vor, dass sich AWT-Applikationen unter verschiedenen Betriebssystemen unterschiedlich verhalten ([27], S. 5), was einen erhöhten Aufwand beim Testen zur Folge hat und im Widerspruch zur Java-Philosophie steht. Ein weiterer Nachteil besteht darin, dass die nativen Peer-Klassen es nahezu unmöglich machen, die AWT-Komponenten sinnvoll durch Vererbung zu erweitern. Weiterhin bedeutet die Benutzung von nativem Code, dass die Portierung von Java auf unter-

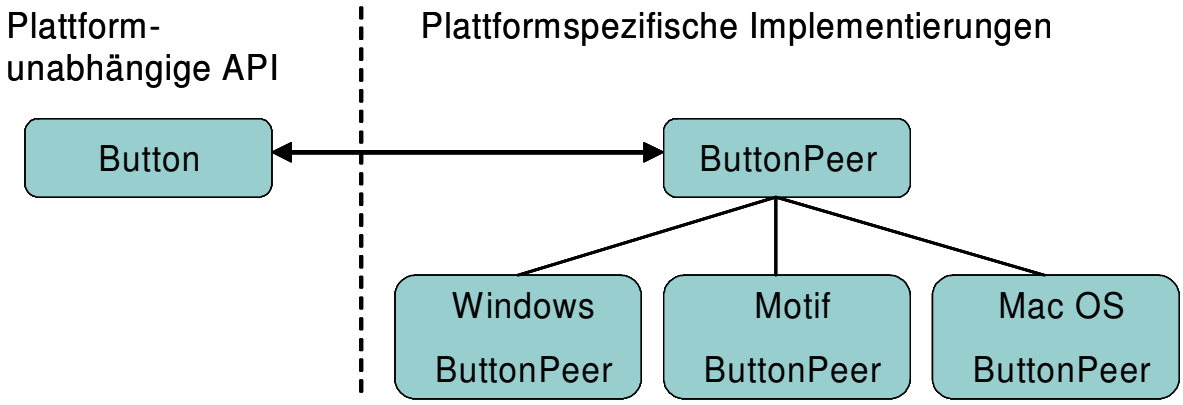

Abbildung 4.1: Das Prinzip der AWT-Klassen und ihrer nativen Peer-Klassen

schiedliche Plattformen schwieriger wird, und sich dabei tendenziell mehr Fehler einschleichen. Schließlich kann AWT durch die Abhängigkeit von nativen Peer-Klassen nur die Funktionalität des kleinsten gemeinsamen Nenners der unterstützten Plattformen, also die GUI-Funktionen die bei allen Plattformen vorhanden sind, in die Java-Welt abbilden.

Bei den AWT-Komponenten handelt es sich um sog. *„schwergewichtige“* Komponenten. Der Name rührt daher, dass die Peer-Objekte immer einen rechteckigen, undurchlässigen Bereich für ihre Komponenten reservieren, so dass sich diese ähnlich verhalten müssen, als würden sie sich selbst ihr eigenes kleines Fenster zur Verfügung stellen. Ein weiterer Grund für dieses Bezeichnung ist, dass die AWT-Komponenten durch ihre Peer-Klassen oft betriebssystemspezifischen Zusatzbalast mit sich herumtragen, der von Java letztlich nicht benutzt wird.

Die Lösung für all diese Probleme ist nach den bisherigen Ausführungen nun naheliegend und besteht darin, die nativen Peer-Klassen los zu werden und deren Funktionalität in Java zu implementieren. Genau diesen Ansatz verwirklicht Swing. Die Grenze zwischen der von den Plattformen abstrahierenden Java-Welt und den einzelnen Betriebssystemen wird dabei auf der viel atomareren Ebene der primitiven Zeichenbefehle überschritten. Die Swing-Komponenten besitzen *Java-Code*, der festlegt, wie die Komponente zu zeichnen ist. Dadurch müssen diese Komponenten keinen Peer-Klassen-Balast mehr mit sich herumschleppen und werden folgerichtig als *„leichtgewichtige Komponenten“* bezeichnet ([27], S. 7). Die einzigen schwergewichtigen Komponenten mit nativen Peer-Klassen, die Swing benötigt, sind die Haupt-Fenster-Klassen (JFrame, JDialog und JApplet). Diese reservieren einen rechteckigen Bildschirmbereich vom Betriebssystem, der dann von den Swing-Komponenten beliebig genutzt werden kann.

Durch diese Architektur der Swing-Komponenten, wird eine wesentlich höhere Flexibilität erreicht. Swing ist nicht durch die Möglichkeiten der einzelnen Wirtsbetriebssysteme beschränkt und es können sehr komplexe GUI-Komponenten realisiert werden, die im Gegensatz zu AWT-Komponenten nicht mehr unbedingt ein rechteckiges Erscheinungsbild haben müssen, sondern auch transparente Bereiche besitzen dürfen.

4.1.2 Pluggable Look & Feel

Wie im vorhergehenden Abschnitt erläutert, zeichnet Swing seine Oberflächenelemente ohne die Hilfe der GUI-Bibliothek des jeweiligen Betriebssystems, auf dem das Java-Programm momentan läuft. Neben den bereits genannten Vorteilen, wirft dieses Vorgehen auch ein neues Problem auf. Normalerweise ist es wünschenswert, dass die Benutzeroberflächen von Java-Programmen nicht wie Fremdkörper wirken und nach Möglichkeit so aussehen, wie die Programme, die eine native GUI-Bibliothek des jeweiligen Betriebssystems nutzen. Bei AWT war dieses Ziel leicht zu erreichen, da ja ohnehin native Klassen benötigt wurden. Da Swing-Komponenten jedoch in Java implementiert sind und daher nichts von dem Look & Feel der einzelnen Betriebssysteme wissen, ist hier eine andere Vorgehensweise notwendig.

Swing kann durch einen Pluggable Look & Feel (PLAF) genannten Mechanismus seinen Komponenten ein beliebiges Aussehen geben. Standardmäßig werden Look & Feel-Pakete mitgeliefert, die das Aussehen der GUI-Bibliotheken von MS Windows und Motif emulieren. Weiterhin existiert ein solches Paket auch für die Mac OS-Oberfläche und darüber hinaus gibt es einen Java-eigenen Stil mit dem Namen „Metal“.

Die Bezeichnung „Pluggable“, also einsteckbar, rührt daher, dass das Aussehen der einzelnen Komponenten während der Laufzeit eines Java-Programms geändert werden kann. Dies wird durch lose Kopplung zwischen den Klassen, die für das Zeichnen der Komponenten verantwortlich sind, und dem Rest der Komponenten erreicht. Diese beiden Teile sind während der Laufzeit nur durch eine Referenz miteinander verbunden, die durch entsprechende Methoden geändert werden kann.

Während eine Swing-Komponente normalerweise über die Methode `setUI()` auch einzeln ein neues Look & Feel zugewiesen werden kann, existiert darüber hinaus die Möglichkeit über die Klasse `UIManager` mit *einem* Methodenaufruf das Aussehen der gesamten Applikation zu ändern. Das Vorgabe-Look & Feel von Swing-Applikationen ist Metal. Mit der Zeile

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

wird der `UIManager` dazu angewiesen, dass das Look & Feel, welches das momentane Wirts-Systems emuliert, für die Swing-Applikation genutzt wird. Dadurch sieht ein und das selbe Swing-Programm unter verschiedenen Betriebssystemen aus wie eine native Applikation, ohne das es dafür neu kompiliert werden müsste. Diese Programmzeile löst also das oben in diesem Abschnitt aufgeworfene Problem.

4.2 Das Java Event-Modell

Events dienen als Nachrichten über Änderungen, die an bestimmten Objekten aufgetreten sind. Ein Event kann beispielsweise durch eine Benutzereingabe oder durch eine Änderung bei bestimmten Anwendungsdaten auftreten. Java benutzt in seiner gesamten API ein einheitliches Event-Modell, das sich vor allem in den Bereichen Swing und den JavaBeans wiederfindet. Als Entwickler hat man darüber hinaus auch die Möglichkeit beliebige selbst geschriebene Klassen zu Eventquellen und Eventempfängern zu machen und dabei auf die Standardmechanismen

des Java Event-Modells zurückzugreifen. Dieser Abschnitt erläutert zunächst das beim Java Event-Modell verwendete Observer-Muster und betrachtet dann die konkrete Umsetzung in Swing.

4.2.1 Das Observer-Muster

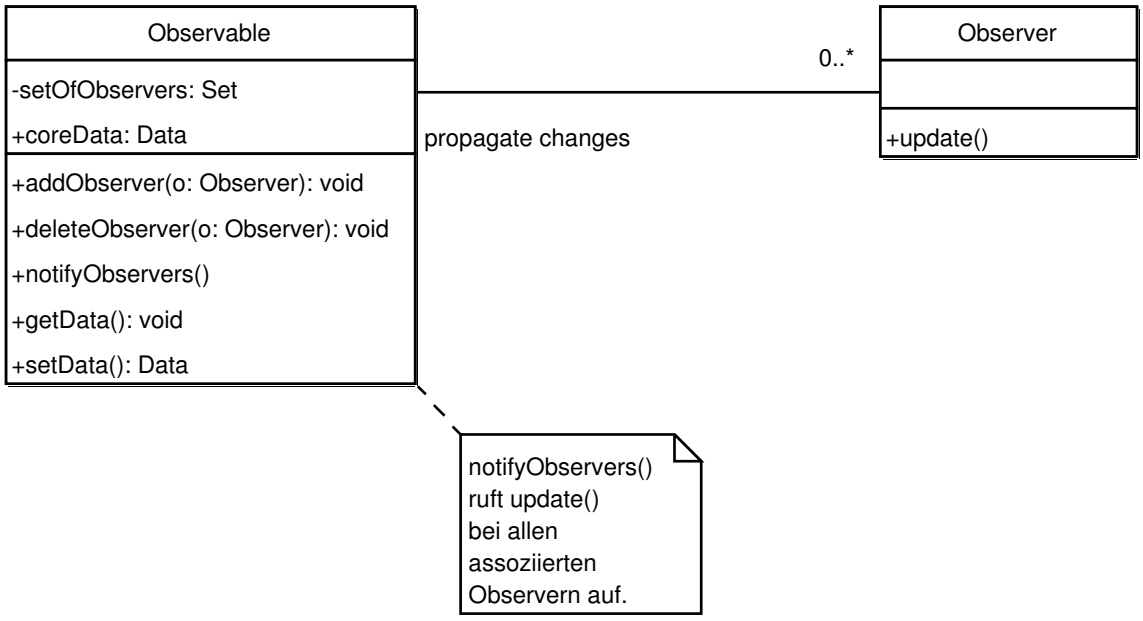

Abbildung 4.2: Konzeptionelles Klassendiagramm zum Observer-Muster

Beim Java Event-Modell wurde das Observer-Muster verwendet, dass manchmal auch als Publisher-Subscriber oder Dependents-Muster bezeichnet wird. Eine ausführlichere Beschreibung dieses Musters ist u.a. in [36] zu finden, hier soll nur ein kurzer Überblick gegeben werden.

Die Verwendung des Observer-Musters erlaubt es, beliebig vielen Objekte Nachrichten über Änderungen an einem anderen Objekt zu beziehen, ohne das feste Aufrufbeziehungen eingeführt werden müssen. Abbildung 4.2 zeigt ein konzeptionelles Klassendiagramm dieses einfachen Musters. Die Klasse Observable hat dabei die Rolle des Nachrichtensenders, die Klasse Observer die des Empfängers. Damit die Klasse Observable alle an den Änderungen interessierten Objekte benachrichtigen kann, verwaltet sie eine Menge von Observern. Observer-Objekte können sich über die Methoden addObserver() und deleteObserver() zu dieser Menge hinzufügen und wieder entfernen, wenn sie nicht mehr an den Änderungen interessiert sind. Tritt eine Änderung, die von Interesse ist, am Observable-Objekt auf, so wird mittels der Methode notifyObservers() die update()-Methode aller registrierter Observer aufgerufen. Die Observer-Objekte können dann in ihrer Implementierung der update()-Methode entscheiden, ob sie im Moment Informationen über die aktuellen Änderungen am Observable-Objekt benötigen und gegebenenfalls Informationen über diese Änderungen über die Methode getData() von Observable erfahren.

4.2.2 Java Event-Programmierung

Wie sieht nun die konkrete Umsetzung dieses Musters in Java aus? Alle Klassen, die Events nach außen geben wollen, nehmen die Rolle des Observable, alle Klassen, die Events empfangen wollen, die des Observers ein.

Bei Java wird für jede Art von Event eine spezielle Event-Klasse verwendet, die alle von der Klasse EventObject erben. So kann beispielsweise ein Button unterschiedliche Events aussenden, wenn er gedrückt wurde (Klasse ActionEvent) oder wenn er den Tastatur-Fokus (Klasse FocusEvent) erhalten hat. Ein Observer, im Java-Event Modell als *Listener* bezeichnet, hat dann die Möglichkeit sich nur für die Event-Typen zu registrieren, die für ihn interessant sind. Eine Klasse wird dadurch zu einem Listener, indem es das entsprechende Listener-Interface implementiert. In der allgemeinen Muster-Beschreibung entsprechen diese implementierten Listener-Methoden der update()-Methode in der Klasse Observer. Für jeden Event-Typ gibt es in Java ein zugehöriges Listener-Interface. Ein Objekt, das mehrere verschiedene Event-Typen feuert, muss dementsprechend für jeden Event-Typ Methoden zum Hinzufügen und Entfernen von Listenern dieses Typs besitzen. Die Signaturen dieser Methoden sind dabei immer gleich aufgebaut. Bietet beispielsweise eine Java-Klasse FooEvents nach außen an, so muss sie folgendes Methodenpaar besitzen:

```
addFooListener(FooListener listener)
removeFooListener(FooListener listener)
```

Manche Event-Quellen unterstützen nur maximal einen Listener („unicast delivery“). Die add-Methode kann dann eine TooManyListeners-Exception werfen, wenn sich ein weiterer Listener registrieren möchte. Durch diesen einheitlichen Aufbau der Event-Methoden haben grafische Entwicklungs-Tools zum Aufbau von Benutzeroberflächen die Möglichkeit, über die Java Reflection-API herauszufinden, welche Arten von Events eine Komponente unterstützt.

Die Listener-Schnittstellen erben alle von dem leeren Interface EventListener. Je nach Event-Typ können sie eine oder mehrere Methoden enthalten. Beispielsweise hat die Schnittstelle FocusListener zwei Methoden. Eine wird aufgerufen, wenn eine Komponente den Tastaturfokus erhalten hat, die andere, wenn sie ihn verloren hat. Ist man nicht an allen Ereignissen interessiert, die durch eine Listener-Schnittstelle unterstützt werden, so läßt man die Implementierung der anderen Methoden einfach leer. Hier ein Codebeispiel für diese Vorgehensweise, das auch einige andere schon genannte Aspekte der Java Event-Programmierung noch einmal verdeutlicht:

```
...
  // einen Button mit der Aufschrift "Button 1" erzeugen.
  JButton button1 = new JButton("Button 1");

  // unseren FocusListener registrieren
  button1.addFocusListener(new MyFocusListener());
...
```

```
// Das interface FocusListener implementieren
class MyFocusListener implements FocusListener {

  // wird aufgerufen, wenn button1 den Tastaturfokus erhalten hat
  public void focusGained(FocusEvent e) {
    System.out.println("button1 hat den Tastaturfokus erhalten.");
  }

  // wird aufgerufen, wenn button1 den Tastaturfokus verloren hat
  public void focusLost(FocusEvent e) {  }
}
```

Um Tipparbeit zu sparen und den Code kompakter zu gestalten, implementiert man Event-Listener in der Praxis häufig als anonyme innere Klassen. Außerdem hält die Swing-Bibliothek auch Adapterklassen für alle Listener-Schnittstellen mit mehr als einer Methode bereit, die für alle Methoden eine leere Implementierung enthalten. Wenn man von einer solchen Klasse erbt, muss man nur noch die Methoden erweitern, an denen man interessiert ist, handelt sich dadurch aber den Nachteil ein, dass man dann von keiner anderen Klasse mehr erben kann.

4.3 Anbindung von Anwendungsdaten an Swing-Oberflächen

Die Swing-Bibliothek stellt nicht nur grafische Oberflächenelemente zur Verfügung, sondern enthält außerdem ein Framework, dass eine elegante Kopplung von Anwendungsdaten an diese Komponenten ermöglicht. Dieses Framework ist gemäß dem Model-View-Controller Muster aufgebaut. In diesem Abschnitt wird zunächst das Muster so beschrieben wie es in der Literatur zu finden ist (nach [16], S. 124 ff.) und daraufhin die konkrete Umsetzung in Swing betrachtet. Diese enthält einige Abweichungen von der allgemeinen Musterbeschreibung.

4.3.1 Das Muster Model-View-Controller

Das Model-View-Controller Muster (MVC) dient zum Entwurf interaktiver Anwendungen mit flexiblen Mensch-Maschine-Schnittstellen. Diese wird in drei Komponenten unterteilt. Das Model enthält die Kernfunktionalität und die Anwendungsdaten. Views präsentieren dem Anwender die in den Anwendungsdaten kodierten Informationen. Für jeden View gibt es einen passenden Controller. Dieser ist für die Verarbeitung von Benutzereingaben verantwortlich und stellt zusammen mit dem View die Benutzeroberfläche dar. Als Event-Mechanismus sieht das MVC-Muster das bereits besprochene Observer-Muster vor. View und Controller haben dabei die Rolle der Observer und sind an den Änderungen der Anwendungsdaten im Model interessiert, das dementsprechend die Rolle des Observables übernimmt. Abbildung 4.3 zeigt ein konzeptionelles Klassendiagramm des MVC-Musters, in dem man sehr gut erkennen kann, das das Observer-Muster Teil des MVC-Musters ist.

Zusätzlich zu den Methoden aus dem Observer-Muster sind einige weiter Methoden vorhanden, die sich in die Bereiche Initialisierung und weiterer Verlauf gruppieren lassen.

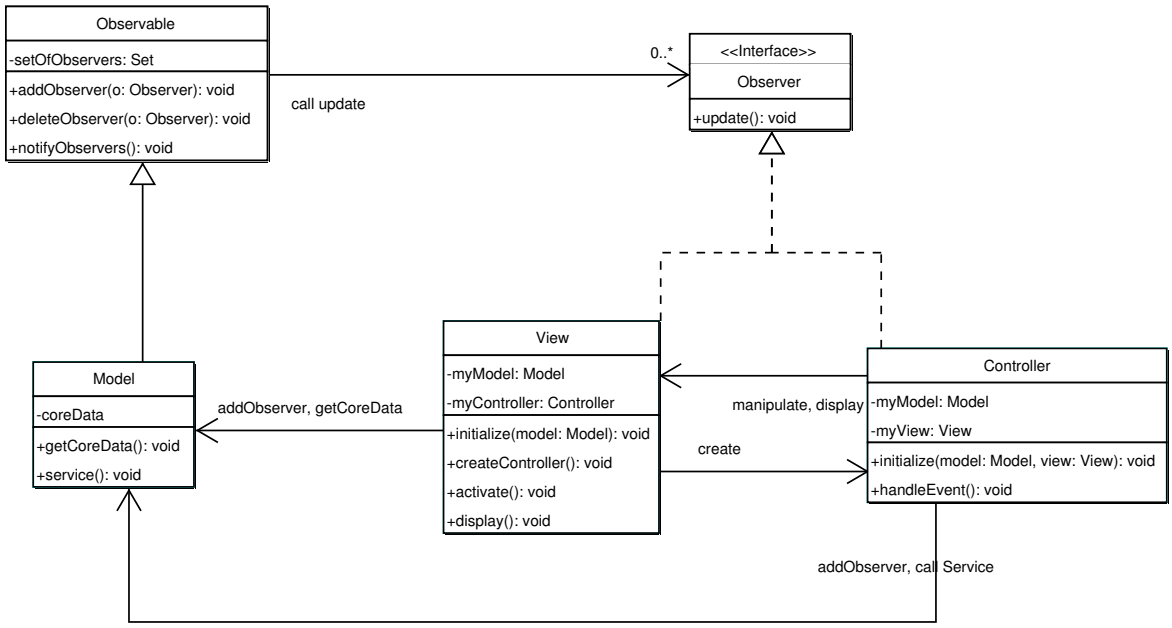

Abbildung 4.3: Konzeptionelles Klassendiagramm zum MVC-Muster

Bei der Initialisierung wird zunächst dem View-Objekt ein Model-Objekt über seine Methode initialize() übergeben. Die View-Komponente erzeugt daraufhin mittels createController() einen Controller und übergibt dabei eine Referenz auf sich selbst und eine auf das Model-Objekt. View und Controller registrieren sich als Observer beim Model und das View holt sich mittels der Methode getData() den anfänglichen Inhalt des Model-Objekts. Ist dies alles geschehen, kann sich das View selbst zeichnen und der Controller kann mittels seiner manipulate()-Methode die Benutzerschnittstelle passend zum Zustand des Models beeinflussen (z.B. bestimmte Oberflächenelemente ausgrauen).

Bei der im weiteren Verlauf stattfindenden Benutzerinteraktion, ist der Controller für die Verarbeitung von Benutzereingaben verantwortlich (Methode handleEvent()) und ruft gegebenfalls Anwendungsfunktionalität am Model aus (Methode service()). Ergibt sich dadurch eine Änderung des Zustand des Models, werden sowohl Controller als auch View über den Observer-Mechanismus darüber informiert und können die Darstellung und die Menge der erlaubten Benutzerinteraktionen entsprechend anpassen.

Durch diesen Aufbau besteht nur eine lose Kopplung zwischen Benutzeroberfläche und Anwendungslogik. Dies ist sehr wünschenswert, da es in der Praxis häufig Änderungsanforderungen an die Benutzeroberfläche gibt, die dahinter liegende Anwendungslogik aber vergleichsweise stabil bleibt. Außerdem wird durch diesen Aufbau die Möglichkeit gegeben, mehrere Ansichten auf die selben Anwendungsdaten zu erstellen. Als Beispiel kann man sich eine Anwendung für die Verarbeitung von Wahlergebnissen vorstellen. Hat man die Wahlergebnisse ín seinen Anwendungsobjekten vorliegen, möchte man diese beispielsweise sowohl als Balkendiagramm als auch als Tortendiagramm oder als textuelle Ansicht anzeigen lassen können. Die Anwendung des MVC-Musters ermöglicht es in diesem Fall, durch die Austausch der View-

Komponente mehrere Ansichten auf immer die gleichen Anwendungsdaten darzustellen, ohne das diese geändert werden müssten. Möchte man in den unterschiedlichen Ansichten auch unterschiedliche Arten der Benutzerinteraktion realisieren, kann dies durch den Austausch der Controller-Komponente geschehen.

Weiterhin ermöglicht das MVC-Muster die Synchronisation verschiedener Ansichten auf die gleichen Daten. Da sich beliebig viele Views und Controller als Observer beim Model registrieren können und somit immer über aktuelle Änderungen am Model informiert werden, ist eine synchrone Darstellung der aktuellen Daten in allen Ansichten gewährleistet.

4.3.2 Model-View-Controller in Swing

Swing ist gemäß dem MVC-Muster aufgebaut, enthält jedoch einige Abweichungen. Die augenfälligste ist die, dass View und Controller zum sog. *UI-delegate* zusammengefasst sind. Somit besteht eine Swing-Komponente immer aus einem UI-delegate und einem Model-Objekt. Diese Variante des MVC-Musters, in der View und Controller zu einer Komponente zusammenfallen wird auch Document-View-Architektur genannt ([16], S. 141). Diese Bezeichnung ist in der Java-Welt allerdings nicht sehr verbreitet.

Außerdem ist es bei Swing so, dass viele einfachere Komponenten (z.B. JButton) bei der Instanziierung automatisch ein passendes Model-Objekt anlegen. Durch diesen Automatismus wird man von der für sehr einfache Benutzeroberflächen recht hohen Komplexität des MVC-Musters befreit und muss sich nicht weiter um die im Hintergrund vorhandenen Model-Objekte kümmern.

Um komplexere Swing-Komponenten (z.B. JTree oder JTable) zu nutzen, ist es allerdings zwingend notwendig, zunächst die darzustellenden Daten in ein passendes Model-Objekt zu bringen. Dieses wird dann der Swing-Komponente übergeben.

Eine weitere Abweichung ist die, dass viele Swing-Komponenten eine Kopie der Signaturen der Methoden ihrer Modelle zum Hinzufügen und Entfernen von Listenern besitzen. Somit muss man seine Listener also nicht mehr am Model-Objekt registrieren, sondern kann dies auch an der Komponente selbst tun. Diese delegieren die Methodenaufrufe an ihr jeweiliges Model weiter.

Zur Verdeutlichung der Nutzung des MVC-Musters in Swing ein kurzes Code-Beispiel, in dem sich zwei Oberflächenelemente ein gemeinsames Model-Objekt teilen:

```
...
String[] languages = {"C", "Java", "DoDL", ... };
final DefaultComboBoxModel model = new DefaultComboBoxModel(languages);
JComboBox comboBox = new JComboBox(model);

combobox.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
      JComboBox source = (JComboBox) e.getSource();
      String newItem = (String) source.getSelectedItem();
      model.insertElementAt(newItem,0);
```

```
    }
  });

JList list = new JList((ListModel) model);
...
```

In diesem Beispiel wird ein Model vom Typ DefaultComboBoxModel mittels eines String-Arrays initialisiert und eine JComboBox mit diesem Model als Datenbasis erzeugt. Durch den als anonyme innere Klasse realisierten ActionListener wird erreicht, dass eine neu in die Combobox eingegebene Zeile dem Model-Objekt hinzugefügt wird. Die in der letzten Zeile instanziierte JList verwendet das gleiche Model-Objekt (DefaultComboBoxModel implementiert das Interface ListModel). Wir haben uns hier also zwei verschiedene Views auf ein und das selbe Datenobjekt erzeugt. Das Ergebnis ist, dass bei der Ausführung des Programms der Inhalt der JList automatisch aktualisiert wird, wenn sich mittels Eingabe über die JComboBox die Datenbasis des Models ändert. Dies funktioniert, weil das Model den beiden darstellenden Komponenten automatisch Events über Änderungen schickt und die Komponenten diese dann darstellen. Wir haben also in diesem kleinen Beispiel zwei synchronisierte Ansichten auf ein Datenobjekt unter Nutzung der Automatismen des MVC-Musters in Swing konstruiert.

Kapitel 5

Evaluation und Usability

Autor: *Leonore Dietrich*

5.1 Einleitung

Die vorliegende Ausarbeitung behandelt die Problematik der Usability von Systemen bzw. deren Graphischen Benutzerschnittstellen - im Folgenden GUIs genannt - an sich sowie Methoden zu deren Sicherung.

Zunächst soll in Kapitel 5.2 Usability allgemein als (Qualitäts-) Merkmal eines Produktes eingeführt und die Notwendigkeit einer Untersuchung der Usability diskutiert werden.

Anschließend werden in Kapitel 5.3 Voraussetzungen und Ziele des Usability-Testing geklärt sowie die grundsätzliche Vorgehensweise und die Einbindung des Verfahrens in die Projektabläufe behandelt. Den Schluss des ersten Abschnittes bildet die Diskussion der Vor- und Nachteile der Usability-Untersuchung im Allgemeinen.

Kapitel 5.4 beschäftigt sich nun mit der heuristischen Evaluation als Methode des Usability-Testings nach Jakob Nielsen. Nach der Einordnung in die Gesamtheit der Verfahren zur Untersuchung und Beurteilung der Usability erfolgt die Vorstellung der Beteiligten sowie der gängigen Heuristiken. Im Folgenden werden dann in die konkrete Vorgehensweise und Auswertung eingeführt und die spezifischen Vor- und Nachteile der Heuristischen Evaluation diskutiert.

Abschließend werden in Kapitel 5.5 als Fazit die wesentlichen Aspekte noch einmal zusammengefasst und diskutiert. Ziel sind hier Aussagen über die Notwendigkeit der Usability-Untersuchung im Allgemeinen sowie die Eignung speziell der Heuristischen Evaluation als Methode.

5.2 Usability

Das folgende Kapitel stellt die Usability vor und beschäftigt sich mit der Notwendigkeit von Usability in interaktiven Systemen.

5.2.1 Charakterisierung

Traditionell wird die Usability von Systemen mit fünf Parametern assoziiert:

- leicht und schnell erlernbar
- effizient zu gebrauchen
- leicht zu merken
- geringe Fehleranfälligkeit
- angenehme Nutzung

Der Nutzer eines Systems sollte möglichst schnell zu guten Ergebnissen kommen und nach dem Erlernen ein hohes Produktivitätslevel erreichen können. Auch der gelegentliche Nutzer muss ohne neues Erlernen und aufwendiges Nachschlagen in Dokumentationen effektiv arbeiten können. Eine intuitiv erfassbare Benutzerführung erhöht also die Usability eines jeden Systems.

Darüber hinaus muss eine GUI so entworfen werden, dass Fehler des Benutzers möglichst vermieden werden. Sollte dennoch ein Fehler unterlaufen, muss der Benutzer jederzeit die Möglichkeit haben, die vorhergegangenen Aktionen ohne großen Aufwand rückgängig zu machen. Insgesamt sollte der Benutzer mit dem System subjektiv zufrieden sein und gerne damit arbeiten [59].

5.2.2 Notwendigkeit

Eine schlechte GUI kann die Interaktion des Nutzers mit dem System beeinträchtigen. Ein schlechtes Design ist weder intuitiv erfassbar noch in kurzer Zeit beherrschbar. Darüber hinaus müssen Fehler aufwendig behoben und Arbeitsschritte häufig nachgeschlagen oder gar ausprobiert werden. Mögliche Folgen sind Verwirrung und Frustration beim Nutzer, Missverständnisse beim Nachvollziehen der Systemreaktionen sowie Fehler und Schwierigkeiten bei der Nutzung des Systems [70].

Besonders im Automobil-Bereich, also insbesondere bei der Entwicklung von HyCop ist eine Minimierung der aufzuwendenden Konzentration auf interaktive Systeme aus Sicherheitsgründen notwendig. Der Fahrer soll sich voll und ganz auf die Straße konzentrieren und trotzdem die Vorzüge eines interaktiven System nutzen können.

5.3 Usability-Testing

Usability-Testing dient der Sicherung von Usability-Kriterien während eines Software-Entwicklungsprozesses. Dieses Kapitel beschäftigt sich mit den Voraussetzungen und Zielen des Usability-Testings und führt in die grundsätzliche Vorgehensweise ein.

5.3.1 Notwendigkeit

In der Entwicklungsphase liegt die Konzentration der Entwickler meist auf dem System an sich, nicht auf dem späteren Benutzer. Mit dem Einzug der neuen Technologien in den Verbrauchermarkt entstand eine neue Zielgruppe, der technisch-spezialisierter Hintergrund fehlt. An diese Veränderung der Benutzer hat sich die Produktenwicklung zu langsam angepasst, so dass hier ein großes Potential für Verbesserungen der Usability liegt. Hinzu kommt die Unterschätzung der Aufgabe des GUI-Designs, die eine schwere, anspruchsvolle Herausforderung darstellt, aber von vielen Unternehmen als trivial oder minder schwerer Entwicklungsaspekt eingeschätzt wird. Entwurf und technische Implementierung einer GUI müssen als eigenständige Aufgaben mit unterschiedlichen Anforderungen an die Entwickler differenziert betrachtet und der Bedeutung des Entwurfes mehr Beachtung entgegengebracht werden [70].

5.3.2 Voraussetzungen und Ziele

Grundlage für die Bewertung der Usability eines Systems ist zunächst einmal die frühzeitige Formulierung der Anforderungen an die Bedienerfreundlichkeit. Dies sollte bereits in der Konzeptionsphase geschehen und eindeutige, klar kontrollierbare Bewertungskriterien festgehalten werden. Darüber hinaus werden ein leicht handhabbares Beurteilungsverfahren sowie relativ kompetente Bewerter zur Durchführung benötigt, so dass eine schnelle Bewertung im Rahmen eines Bewertungsprozesses stattfinden kann. Die Bewerter sollten hierzu in der Lage sein, das System zu bedienen und so anhand der Kriterien die einzelnen Komponenten zu bewerten [69].

Identifikation und Behebung der Usability-Defizite vor dem Produktrelease sind Ziel der Usability-Untersuchung. Die Dokumentation der Usability-Kriterien spart Kosten bei der Weiterentwicklung und trägt so zur Erhaltung der Qualität die zukünftigen Versionen bei. Eine gute Usability minimiert Service- und Hotlinekosten durch geringere Support-Anfrage, kann Verkaufszahlen erhöhen und Marktanteile sichern. Ein zufriedener Benutzer wechselt kaum das Produkt und arbeitet auch länger mit einer ausgereiften Version, so dass die Versionsintervalle vergrößert werden können. Darüber hinaus werden durch die Usability-Untersuchung die Risiken beim Marktstart minimiert, da ein ausgereiftes Produkt sich schnell durchsetzen und gut am Markt halten kann [60].

5.3.3 Verfahren

Es gibt verschiedenste Ansätze für Usability-Testing aus Wissenschaft und Industrie. Einige Beispiele sind die formale Usability-Untersuchung, die Standard-Untersuchung, die Konsistenzuntersuchung, der Cognitive Walkthrough, individuelle Anforderungskataloge von Herstellern oder die Heuristische Evaluation, die in Kapitel 4 dieser Arbeit vorgestellt wird.

Ideal wäre eine allgemeingültige Definition von Anforderungen und Bewertungskriterien sowie die Nutzung eines einzigen Beurteilungsverfahrens für die Gesamtheit aller Systeme. Verschiedene Produktgruppen benötigen jedoch verschiedene Testmethoden, und auch die Kriterien für die Tests sind produktspezifisch. Damit ist ein Vergleich unterschiedlicher Verfahren nur sehr eingeschränkt möglich und nur ein paar wenige, relativ allgemeine Kriterien auf eine

größere Gruppe von Systemen anwendbar. Für die jeweilige Untersuchung müssen diese dann gegebenenfalls konkretisiert und ergänzt werden [69].

5.3.4 Vor- und Nachteile, Probleme

Die Vorteile eines Produktes mit guter Usability liegen in der kurzen Einarbeitungszeit, einer effektiven Nutzung sowie der wenigen Benutzungsfehler. Hinzu kommt, besonders beim Einsatz von interaktiven Systemen im Automobilbereich, ein erhöhtes Maß an Sicherheit. Insgesamt wird die Akzeptanz beim Nutzer erhöht, wodurch wiederum Absatzzahlen verbessert, Versionsintervalle verlängert und die Kundenbindung verstärkt werden können.

Die augenscheinlichen Nachteile eines Einsatzes von Usability-Testing sind die anfallenden Kosten und der Personalaufwand. Diese amortisieren sich jedoch durch Einsparungen im Bereich Redesign und durch die Vermeidung größerer Änderungen in fortgeschrittenen Entwicklungsstadien [62].

Ein zentrales Problem des Usability-Testings sind die Unterschiede zwischen den Bewertern als Testpersonen und den späteren Benutzern. Zudem ist die Testsituation keine natürliche Anwendungssituation, da nach bestimmten Mustern und nicht anhand konkreter Aufgabenstellungen vorgegangen wird. Die durch Usability-Kriterien entstehenden Vorgaben werden darüber hinaus von vielen Entwicklern als Einschränkung der Entwurfsfreiheit aufgefasst und daher häufig negativ aufgenommen [70].

5.4 Heuristische Evaluation

Die heuristische Evaluation ist eine Methode zur schnellen, kostensparenden, systematischen und leichten Evaluation eines GUI-Designs, um es im Rahmen eines iterativen Entwurfsprozesses zu untersuchen. Nach einem Evaluationsdurchgang werden die gefundenen Fehler behoben und das Redesign anschließend wiederum einer Evaluation unterzogen, um sicherzustellen, dass keine neuen Fehler auftreten. Die heuristische Evaluation stellt eine der bekanntesten Methode zur Untersuchung der Usability von Systemen dar. Sie soll hier nach Jakob Nielsen vorgestellt werden [61].

5.4.1 Beteiligte

An der heuristischen Evaluation sind zunächst einige fachlich relativ kompetente Bewerter beteiligt, die die GUI untersuchen und anhand sogenannter Heuristiken bewerten. Als optimale Anzahl wurden in entsprechenden Studien drei bis fünf Bewerter ermittelt, da so das beste Verhältnis zwischen gefundenen Usability-Problemen und Aufwand für die Untersuchung erreicht wird.

Die Bewerter erstellen eine Liste mit Usability-Problemen. Die einzelnen Punkte müssen in Bezug auf die Heuristiken genannt und kommentiert werden. Dabei sollte die Begründung des jeweiligen Problems möglichst genau festgehalten werden. Zusätzlich sollen auch weitere, während der Untersuchung aufgefallene Problembereiche und hierdurch entstehende neue

Kriterien in die Liste aufgenommen und diskutiert werden.

Neben den Bewertern kann noch ein Beobachter an der Untersuchung beteiligt sein. Dieser kann gegebenenfalls Hilfestellung bei Problemen mit dem System geben und im weiteren Verlauf die Ergebnisse der einzelnen Bewerter sammeln und so schnell einen Überblick über die Gesamtheit der Probleme und deren Ausmaß erlangen. Anschließend leitet und beobachtet er dann die Diskussion der Bewerter und verfasst einen Abschlussbericht, in dem die Usability-Probleme klassifiziert, ausführlich kommentiert und begründet werden. Der Beobachter erhöht zwar den personellen Overhead der Evaluation, reduziert aber den Arbeitsaufwand der Bewerter und trägt zu einer schnellen Auswertung bei, da bei ihm alle Einzelergebnisse zusammenlaufen und er so einen guten Überblick über das Gesamtergebnis der Evaluation hat. [62]

5.4.2 Heuristiken

Jakob Nielsen nennt in [64] zehn wesentliche anerkannte Usability-Kriterien, die im Rahmen einer heuristischen Evaluation überprüft werden müssen:

- Sichtbarkeit des Systemstatus: Der Benutzer sollte immer darüber informiert werden, was im System gerade passiert.
- Übereinstimmung System - reale Welt: Das System sollte die Sprache des Benutzers sprechen und keine systemorientierten Ausdrücke gebrauchen. Dialoge und Funktionen sollten einer natürlichen und logischen Abfolge entsprechen und so für den Benutzer ohne technisches Hintergrundwissen verständlich und nachvollziehbar sein.
- Nutzerkontrolle und Freiheit: Dem Nutzer sollte in jeder Situation ein Notausgang zur Verfügung stehen, um unerwünschte Prozesse abzubrechen. Ebenso muss er jederzeit in der Lage sein, ausgeführte Schritte rückgängig zu machen.
- Konsistenz und Standards: In der Benennung der Funktionen sollte ebenso Konsistenz herrschen wie allgemein gültige Standards, beispielsweise Plattformkonventionen eingehalten werden müssen.
- Fehlervermeidung: Ein gutes Design, das durch eine gute Benutzerführung Fehler von vornherein vermeidet, ist immer besser als gute Fehlermeldungen.
- Präsenz benötigter Informationen: In Dialogen sollten die relevanten Informationen für den Benutzer immer sichtbar oder zumindest leicht erreichbar sein. In einem gut entworfenen System muss sich der Benutzer möglichst wenige Informationen merken, um mit dem System arbeiten zu können.
- Flexibilität und Effiziente Nutzung: Der erfahrene Benutzer sollte in der Lage sein, beispielsweise durch Tastenkombinationen oder Skripte häufig benötigte Abläufe zu beschleunigen. Auch sollte der Benutzer das System an seine Bedürfnisse anpassen können, indem sich beispielsweise Menüs individuell zusammenstellen lassen.
- Ästhetik und minimalistisches Design: In der gesamten GUI und besonders in den Dialogen dürfen ausschliesslich wirklich relevante Informationen enthalten sein. Insgesamt sollte die GUI ansprechend und klar strukturiert sein.

- Fehlerbehebung: In Fehlermeldungen sollten keine Codes verwendet, sondern in natürlicher Sprache und verständlicher Formulierung das Problem präzise beschrieben werden. Darüber hinaus muss dem Benutzer eine konkrete Problemlösung angeboten und möglichst in einzelnen Schritten erörtert werden.
- Hilfe und Dokumentation: Im Idealfall kann ein System ohne eine Hilfe genutzt werden. Sollten Hilfe und Dokumentation dennoch benötigt werden, müssen die gewünschten Informationen leicht zu suchen und auf die Aufgaben des Benutzers zugeschnitten sein. So sollte die Dokumentation nicht zu umfangreich sein und konkrete Schritte zur Problembehandlung und Aufgabenlösung nennen.

5.4.3 Verfahren

Eine Evaluationssitzung sollte ca. 1-2 Stunden dauern, für größere Projekte müssen daher mehrere Sitzungen eingeplant werden. In einer Sitzung sollte jeder Bewerter wenigstens zweimal die gesamte GUI durcharbeiten - im ersten Durchgang kann er sich an die GUI gewöhnen, um sich dann im zweiten Durchgang auf spezielle GUI-Elemente zu konzentrieren. Das konkrete Vorgehen ist hierbei beliebig, die Bewerter können eigene Abläufe auswählen. Die Untersuchung kann durch eine Debriefing-Sitzung nach der letzten Evaluationssitzung ergänzt werden. An diesem Brainstorming nehmen Bewerter, Beobachter sowie Vertreter des Design-Teams teil und erarbeiten Vorschläge zur Behebung der zentralen Usability-Probleme. [62]

Zusätzlich können Nutzertests eine iterative Erweiterung darstellen, die mit der Heuristischen Evaluation alternierend als iterative Methode kombiniert wird. In diesem Fall wird zunächst eine Evaluation durchgeführt, anschließend die GUI überarbeitet und die neue Version dann einem Nutzertest unterzogen. [65]

5.4.4 Auswertung

Die im Rahmen der Heuristischen Evaluation gefundenen Usability-Probleme müssen nun anhand von Bewertungsmassstäben näher charakterisiert und ihre Bedeutung beschrieben werden. Die Schwere eines Usability-Problems wird durch 3 Faktoren bestimmt:

- die Regelmäßigkeit, mit der das Problem auftritt
- die Auswirkungen, die es beim Auftreten auf Benutzer und System hat
- die Beständigkeit des Problems

Diese drei Faktoren werden in einer Bewertung zusammengefasst:

	Bewertung	Priorität bei der Behebung
0	kein Usability-Problem	
1	kosmetisches Problem	braucht nur beseitigt werden, wenn Zeit übrig ist
2	kleines Usability-Problem	geringe Priorität bei der Fehlerbehebung
3	großes Usability-Problem	wichtig zu beheben, hohe Priorität
4	Usability-Katastrophe	unbedingt beheben

Da die Bewertungen eines einzelnen Bewerters nicht besonders aussagekräftigsind, werden in der Praxis die Mittelwerte eines Sets von Bewertungen dreier Bewerter genutzt. Die Qualität der Durchschnittsbewertung steigt rapide mit der Zahl der Bewerter. [63]

5.4.5 Beispiel

Das folgende Beispiel (siehe Abb. 5.1) ist [58] entnommen. Es zeigt ein Wetterinformationssystem für Reisende, das im Internet verfügbar sein soll. Der Benutzer kann durch Eingabe von Datum, Uhrzeit und Koordinaten eine Wetterübersicht für sein Reiseziel erhalten. Eingaben werden grundsätzlich durch einen Mausklick außerhalb des aktuellen Eingabefeldes übernommen. Im Folgenden werden einige zentrale Usability-Probleme dieser GUI beispiel-

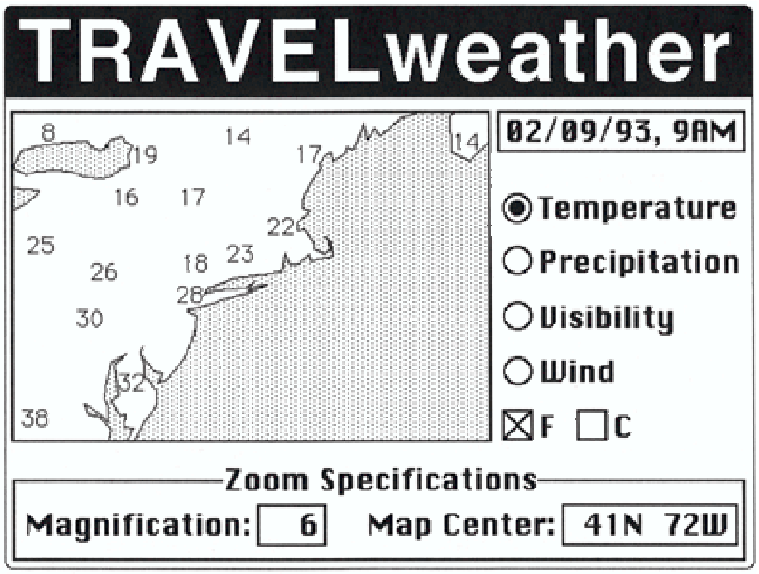

Abbildung 5.1: Wetterinformationssystem für Reisende

haft behandelt.

Datums- und Uhrzeiteingabe

Als gültige Eingaben werden nur die Uhrzeiten „3AM“, „9AM“, „3PM“ und „9PM“ akzeptiert, ebenso muss das Datum dem dargestellten Format entsprechen und das des aktuellen, des folgenden oder des übernächsten Tages sein. Ist eine dieser Eingaben nicht korrekt, so erscheint eine Fehlermeldung (siehe Abb.5.2). „OK“ setzt den Inhalt des Eingabefeldes wieder auf die alten Werte zurück. Hier werden die Heuristiken Übereinstimmung System-reale Welt, Nutzerkontrolle, Fehlervermeidung und Fehlerbehebung verletzt. Der Benutzer erhält keinerlei Informationen über seinen Fehler und die möglichen Eingabewerte. Sinnvoll wären an dieser Stelle zwei Listen mit den möglichen Daten und Uhrzeiten. Mit je 3 bzw. 4 Werten erhielte man eine übersichtliche Auswahl, die dem Benutzer eine schnelle und einfache Interaktion mit dem System ermöglichen würde.

Abbildung 5.2: Fehlermeldung bei ungültiger Datumseingabe

Datum und Uhrzeit sind zentrale Funktionen und werden äußerst regelmäßig genutzt. Mangels aussagekräftiger Fehlermeldungen kann der Benutzer nur durch Probieren die zulässigen Eingaben herausfinden. Er ist bis dahin nicht in der Lage, das System zu bedienen. Es ist vorstellbar, dass die Interaktion ohne Informationsgewinn abgebrochen wird. Bis der Benutzer die gültigen Eingaben ermittelt hat, muss er sich immer wieder dem Problem stellen. Angesichts der Schwere der Mängel ist die Datumseingabe als Usability-Problem der Kategorie 3-4 zu beurteilen.

Kartennavigation

Um auf der Karte zu navigieren, muss der Benutzer die Koordinaten des Zielortes kennen. Zulässig sind die Werte 0 bis 90N/S und 0 bis 179E/W. Werden unzulässige Werte eingegeben, so erscheint eine Fehlermeldung (siehe Abb.5.3). Durch „OK“ wird der Inhalt des Eingabefel-

Abbildung 5.3: Fehlermeldung bei ungültiger Gradeingabe

des wieder auf die alten Werte zurückgesetzt. Hier werden die Heuristiken Übereinstimmung System-reale Welt, Nutzerkontrolle, Fehlervermeidung und Fehlerbehebung verletzt. Der Benutzer erhält keinerlei Informationen über seinen Fehler und die möglichen Eingabewerte. Er muss die Koordinaten des Zielortes kennen, um mit dem System zu arbeiten. Hier wäre es sinnvoll, die Navigation gänzlich anders zu realisieren. Scrollbalken zur Positionsänderung sind unabhängig von weiteren Informationen über die geographische Lage und allgemein als Navigationselemente bekannt.

Die Eingabe des Zielortes ist Grundlage für die Benutzung des Systems und findet daher regelmäßig statt. Die schlechte Navigation macht eine Benutzung teilweise unmöglich. Kennt er die Zielkoordinaten, so kann der Benutzer trotzdem nur durch Probieren das zulässige Eingabeformat herausfinden. Bei jeder neuen Benutzung des Systems steht der Benutzer vor dem Problem der Koordinateneingabe. Auch dieses Problem kann angesichts der Schwere der einzelnen Faktoren in die Kategorie 3 eingestuft werden.

Orientierung

Der Benutzer hat über die im Fenster angezeigten Daten hinaus keinerlei weitere Informationen zur Verfügung. Hier wird die Heuristik Präsenz benötigter Informationen gleich mehrfach verletzt. Es steht keine Legende zur Verfügung, die die Schraffierungen in der Karte erklärt. So muss der Nutzer erahnen, wobei es sich um Wasser und wobei um Land handelt. Namen großer Städte oder Staaten zur Orientierung fehlen gänzlich. Besonders im Inland großer Kontinente stellen mangelhafte Orientierungsmöglichkeiten ein schwerwiegendes Problem dar. Hier kann auch durch Staats- und Landesgrenzen Abhilfe geschaffen werden. Die Differenzierung zwischen verschiedenen Elementen kann auch farblich oder durch aussagekräftige Muster wie Wellen eindeutig gekennzeichnet werden, so dass eine Legende nicht unbedingt notwendig ist.

Die Orientierung ist Grundlage für die Benutzung des Systems und durchgehend nötig. Die schlechte Orientierung macht eine Benutzung teilweise unmöglich. Nur durch Kenntnis von Küstenlinien o.ä. ist eine grobe Einordnung des angezeigten Ausschnittes möglich und es besteht die Gefahr, dass der Benutzer abwandert. Bei jeder neuen Eingabe steht der Benutzer vor diesem Problem. Insgesamt handelt es sich auch hier um ein sehr schwerwiegendes Usability-Problem, das man in Kategorie 3-4 einordnen kann.

Weitere Usability-Probleme finden sich im Gesamtlayout, der Wahl der Interaktionselemente, der Kennzeichnung der editierbaren Bereiche, mangelnder Einhaltung internationaler Standards und weiteren Elementen.

5.4.6 Vor- und Nachteile

Ein zentraler Vorteil der Heuristischen Evaluation ist die Möglichkeit des Einsatzes bereits in sehr frühen Entwurfsphasen. So können auch Projekte, die bisher nur auf Papier exstieren, bereits untersucht und Probleme so mit geringem Aufwand behoben werden [62]. Im Rahmen der Heuristische Evaluation werden sowohl größere als auch kleinere Fehler gefunden und als iteratives Verfahren bringt besonders die Kombination mit Nutzertests eine kontinuierliche Verbesserung bei relativ geringem Aufwand [65]. Die Heuristische Evaluation hat sich darüber hinaus als vergleichbar kostengünstig erwiesen. Für die Durchführung sind geschulte Bewerter und eventuell zusätzlich ein Beobachter nötig. Diese stellen einen gewissen Kostenaufwand dar, der sich jedoch in einer Fallstudie in [62] als sinnvolle Investition bewiesen hat. Der Kosten-Nutzen-Faktor wurde dort mit 48 beziffert - der wirtschaftliche Nutzen war also 48 mal so hoch wie der Kostenaufwand.

Ein zentraler Nachteil der reinen Heuristischen Evaluation ist der Unterschied zwischen den Bewertern und den späteren Nutzern sowie zwischen Untersuchungs- und realer Arbeitssituation. Darüber hinaus stellt die Heuristische Evaluation eine Methode zur reinen Problemsuche dar, die Behebung der Fehler wird nicht behandelt und muss in einem eigenen Verfahren geschehen [62].

5.5 Fazit

Usability-Testing an sich ist unverzichtbar, da die Risiken beim Marktstart deutlich minimiert und die Akzeptanz beim Benutzer erhöht werden. Die Heuristische Evaluation ist als einfache, schnelle und effektive Methode zur Überprüfung der Usability eines Produktes sehr gut geeignet, wobei sich insbesondere die Ergebnisse, die aus der Kombination mit Nutzertests hervorgehen als besonders gut erwiesen haben. Die entstehenden Kosten können sich durch den Einsatz geeigneter Techniken sogar mehr als nur amortisieren, so dass insgesamt die positiven Faktoren der Usability-Untersuchung bei Weitem überwiegen.

Kapitel 6

Das Dexter Hypertext-Referenzmodell

Autor: *Markus Niehammer*

6.1 Einleitung

Ein Hypertextsystem strukturiert Informationen durch relationale Verknüpfungen. Dabei sind die Daten nicht einfach sequentiell angeordnet, sondern werden als Knoten in einem Graphen betrachtet. Die Kanten des Graphen entsprechen den Verknüpfungen. Prinzipiell müssen bei einem solchen Graphen zunächst keine Einschränkungen gemacht werden. Allerdings werden die Möglichkeiten des Konzeptes in bestehenden Hypertextsystemen kaum ausgeschöpft, z.B. wird oft der Grad eines Knotens begrenzt. Welche Möglichkeiten zur Gestaltung von Hypertextsystemen bietet nun das Prinzip an sich? In wie weit können diese Möglichkeiten beim Entwurf von Hypertextsystemen ausgeschöpft werden? In wie weit nutzen bereits bestehende Hypertextsysteme die Möglichkeiten aus?

Das Dexter Hypertext-Referenzmodell stellt einen Standard für den Aufbau eines Hypertextsystems dar. Der grundlegende Gedanke, Daten durch einen Graphen mit Verknüpfungen zu strukturieren, wird im Referenzmodell formalisiert, so dass es beim Entwurf von Hypertextsystemen als Basis dienen kann. Bestehende Hypertextsysteme können mit Hilfe des Modells im Hinblick auf ihre Funktionalität hin verglichen, sowie gegen andere Systeme – wie zum Beispiel sequentielle Datenbanken – abgegrenzt werden.

Durch ein formales Modell wird eine einheitliche Terminologie für bestehende und neu zu entwickelnde Hypertextsysteme festgelegt. Das Dexter-Modell schafft durch Benennung der einzelnen Bestandteile des Systems nicht nur eine Möglichkeit des Vergleichs verschiedener Systeme, sondern liefert auch eine Basis zur Entwicklung von Standards für den Datenaustausch und die Kombination verschiedener Hypertextsysteme.

6.2 Was ist Hypertext?

Zweck eines Hypertextsystems ist die Organisation von Daten, die hier in Form von *Komponenten* vorliegen. Dabei wird auf die besonderen Eigenschaften der Daten nicht weiter eingegangen, sondern eine Komponente im System nur als ganzes betrachtet. Die Komponenten sind die Träger der Informationen, die aus Text, Grafik, Bildern, Animationen, Musik usw. bestehen können. Durch die abstrakte Betrachtung der Komponenten als „Black Box“ eröffnet sich für den eigentlichen Inhalt der Komponenten ein breites Spektrum an Möglichkeiten.

Die Strukturierung der Daten erfolgt durch relationale Verknüpfungen. Diese *Links* verbinden mehrere Komponenten untereinander und erlauben das Bewegen im Hypertext entlang dieser Verknüpfungen. Links müssen natürlich nicht unbedingt binär sein. Dadurch können beliebig viele Komponenten in einer Relation erfasst werden, und das Navigieren im Hypertext wird nicht auf das bloße Springen von einer Komponente zur anderen begrenzt.

Ein weiterer Teil eines Hypertextsystems ist die Schnittstelle zum Benutzer. Dieser soll die Daten in den Komponenten betrachten, und sich anhand der Links im Hypertext bewegen können. Es müssen neue Komponenten erzeugt und bestehende manipuliert werden können. Auch die Verknüpfungsstruktur sollte veränderbar sein.

6.3 Layers und Interfaces

Die im vorigen Abschnitt deutlich gewordene Dreiteilung eines Hypertextsystems spiegelt sich in den drei Schichten (*layers*) des Dexter-Modells wieder:

Run-Time Layer
Im Run-Time-Layer wird eine Instanz des Hypertextes visualisiert. Die Schicht bietet dem Benutzer Möglichkeiten zur Interaktion mit dem System.

Storage Layer
Der Mechanismus der Verknüpfung von Daten, d.h. das abstrakte Netzwerk aus miteinander verbundenen Komponenten, wird im Storage Layer betrachtet. Losgelöst von der inneren Struktur der einzelnen Komponenten, werden diese hier als „Black Box“ angesehen.

Within-Component Layer
Im Gegensatz zum Storage Layer behandelt der Within-Component Layer den Inhalt und die innere Struktur der Komponenten.

Um die Vielfalt an möglichen Typen von Komponenten nicht unnötig einzugrenzen, wird der Within-Component Layer im Dexter-Modell nicht näher betrachtet. Daher sind prinzipiell beliebige Arten von (Medien-)Objekten als Bestandteile eines Hypertextes denkbar.

Das Prinzip des Run-Time Layers lässt sich am Beispiel eines WWW-Browsers veranschaulichen: Die Komponenten, d.h. die Webseiten werden hierbei im Browser-Fenster dargestellt.

Der Benutzer hat die Möglichkeit, sich über Links im Hypertext zu bewegen. Das WWW bietet allerdings kaum Möglichkeiten zur Manipulation der Komponenten, so dass sich die Interaktion in der Regel auf das Betrachten einer Webseite beschränkt.

Der Hauptbestandteil des Dexter-Modells ist der Storage-Layer. Hier wird der eigentliche Hypertext abstrakt dargestellt. Zum Zugriff auf Teile innerhalb von Komponenten sowie zum Betrachten im Run-Time Layer dienen zwei Schnittstellen (*interfaces*):

Run-Time Layer

Presentation Specification

Storage Layer

Anchoring

Within-Component Layer

Die Presentation Specification bestimmt das Aussehen einer Komponente im Run-Time Layer. Dabei kann es durchaus verschiedene Ansichten geben, da das Erscheinungsbild von verschiedenen Faktoren abhängt: Zum einen bestimmt eine Komponente selbst ihr Aussehen. Allerdings kann es auch davon abhängen, über welchen Link die Komponente erreicht wird. Auch im Run-Time Layer selbst kann die Art der Präsentation verändert werden, etwa durch Anpassung an verschiedene Benutzer mit unterschiedlichen Zugriffsrechten. Ein anschauliches Beispiel für Unterschiede in der Präsentation ist auch das Betrachten einer Webseite in verschiedenen Browsern.

Da der Storage Layer das Innere der Komponenten nicht betrachtet, ist zum Zugriff auf Teile oder Positionen innerhalb von Komponenten eine Schnittstelle zum Within-Component Layer nötig. Diese Möglichkeiten bietet für den Storage Layer das Anchoring.

6.4 Storage Layer

Der Storage Layer besteht aus einer endlichen Menge von Komponenten. Die Funktionen *resolver* und *accessor* regeln den Zugriff auf die Komponenten. Weiterhin steht zum Bearbeiten der Struktur des Hypertextes ein Satz aus Instruktionen zur Verfügung, womit z.B. Links definiert oder Komponenten erzeugt, geändert oder gelöscht werden können.

Definition: Eine (Basis-)Komponente ist

- eine atomare, primitive Struktur aus dem Within-Component Layer (*atom*)
- ein Tupel (*link*) aus mehreren Komponenten, den Endpunkten des Links
  oder
- eine Komposition (*composite*) von Komponenten, die azyklisch und gerichtet ist.

Dabei muss ein Link nicht unbedingt binär sein. Bei der Komposition von Komponenten ist zu beachten, dass eine Komponente sich nicht selbst enthalten darf und dass zwei Komponenten sich nicht wechselseitig enthalten. Nicht ausgeschlossen ist, dass eine Komponente in mehreren anderen Komponenten enthalten ist, ebenso wie eine Komponente auch mehrere andere Komponenten - auch mehrfach - enthalten kann.

Definition: Bestandteile einer Komponente (*component specification*) im Storage Layer sind

- eine solche Basiskomponente
- eine global eindeutige Identifikation (*UID*)
- Verweise auf Stellen im Inhalt der Komponente (*anchors*)
- eine presentation specification
- weitere Attribute.

Die Komponente muss durch ihren UID (*unique identifier*) global, d.h. nicht nur in einem speziellen Hypertextsystem, eindeutig identifizierbar sein. Im WWW ist dieser UID der URL (uniform resource locator). Die Presentation Specification legt wie oben bereits beschrieben, das Aussehen der Komponente fest. Auf den Mechanismus der Verankerung wird später noch näher eingegangen. Weitere Attribute einer Komponente können zum Beispiel deren Größe, Typ oder bestimmte Schlüsselwörter sein, die zur Charakterisierung der Komponente dienen.

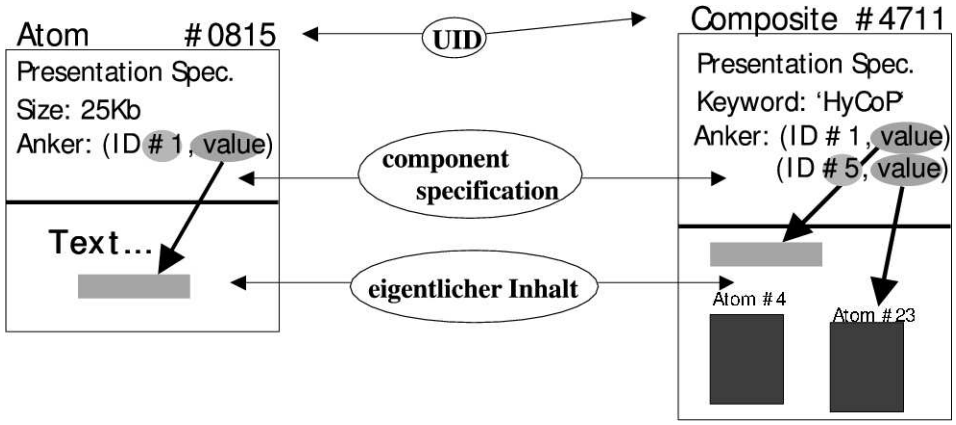

Abbildung 6.1: Aufbau von Komponenten

Die accessor- und die resolver-Funktion sind für das Auflösen der Links zuständig. Dabei gibt der accessor auf die Frage nach einem UID die Komponente mit diesem UID zurück. Die accessor-Funktion muss eine umkehrbare Funktion sein, d.h. jede Komponente muss auch einen UID haben. Eine Komponente ohne UID könnte nicht adressiert werden, und wäre damit nicht Bestandteil des Hypertextes.

Der resolver sucht nach Komponenten mit bestimmten Spezifikationen, z.B. die größte Komponente im Hypertext oder alle Komponenten, die ein bestimmtes Schlüsselwort enthalten. Das Ergebnis der Anfrage ist ein UID, der dann dem accessor übergeben wird. Natürlich muss

auf eine Anfrage hin nicht unbedingt eine Komponente gefunden werden. Daher ist der resolver eine partielle Funktion. Umgekehrt hat aber jede Komponente eine Spezifikation, zu der der resolver einen UID liefert. Wird demnach einfach

nur nach einem solchen UID gefragt, berechnet der resolver die Identität. Den Ablauf verdeutlicht das folgende Schema:

$$\text{Spezifikation} \xrightarrow{\text{resolver}} \text{UID} \xrightarrow{\text{accessor}} \text{Komponente}$$

Eine weitere Aufgabe der resolver-Funktion ist das Adressieren der Anker, die ja auch Bestandteil der Spezifikation einer Komponente sind.

Definition: Ein Anker (*anchor*) ist ein Paar aus

- Anker-ID (*identifier*) und
- Anker-Wert (*value*).

Der ID des Ankers identifiziert dabei, genau wie der UID einer Komponente, den Anker eindeutig. Da der Anker Bestandteil der Spezifikation einer Komponente ist, ist eine global eindeutige Adressierung durch Angabe von UID und Anker-ID möglich. Zugriff auf einen Anker und damit auf einen Teil bzw. eine bestimmte Stelle innerhalb einer Komponente erfolgt aus der Sicht des Storage Layers nur über die ID.

Der Wert des Ankers dagegen kann innerhalb des Within-Component Layers gesetzt werden. In der Spezifikation einer Komponente wird zwar einem ID eindeutig ein Wert zugeordnet, doch ist dieser Wert, d.h. ein Teil oder Stelle in einer Komponente, innerhalb der Komponente selbst durchaus veränderbar. (vgl. Abb. 6.1)

Definition: Ein Link ist ein Tupel aus zwei oder mehr *specifiers*.

Definition: Ein specifier besteht aus

- der Spezifikation einer Komponente (*component specification*)
- einem Anker-ID
- einer Richtung (*direction*), und zwar: FROM, TO, BIDIRECT oder NONE
- einer presentation specification.

Der specifier gibt die Endpunkte eines Links an. Bemerkenswert ist, dass ein solcher Link im Storage Layer auch als Komponente angesehen wird (vgl. Definition (Basis-)Komponente). Während sich aber atomare oder zusammengesetzte Komponenten aus Inhalt und Spezifikation zusammensetzen, enthält der Link mehrere specifiers.

Auf eine Komponente wird im specifier des Links einfach durch Angabe des UID verwiesen. Der Anker-ID verweist dementsprechend auf einen Anker in der Spezifiaktion der Komponente, dessen Wert wiederum einen bestimmten Teil im Inhalt adressiert. Ein Link muss nicht

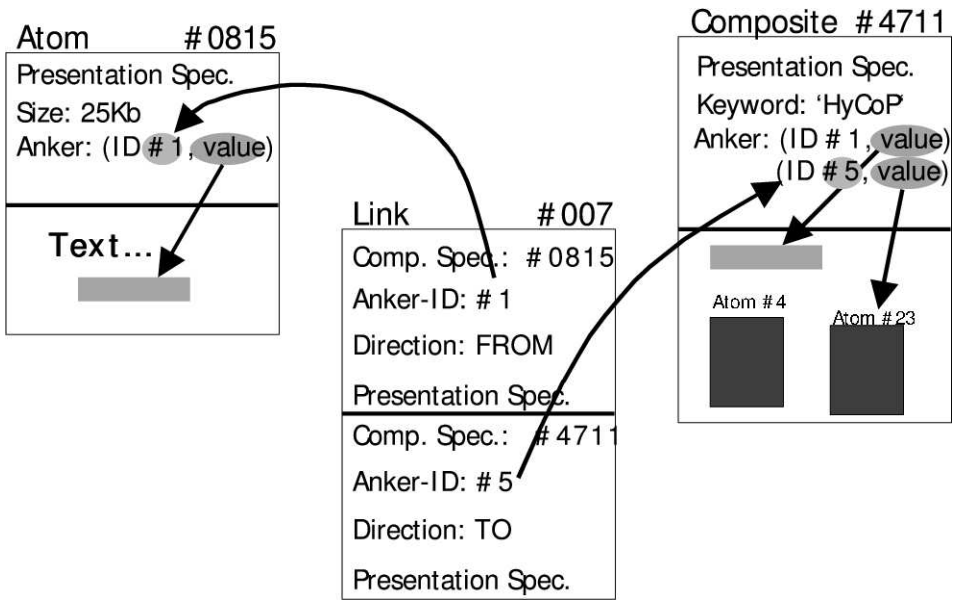

Abbildung 6.2: Konzept des Links zwischen zwei Komponenten

unbedingt nur eine Richtung haben. Bidirektionale Links sind genauso möglich wie Links ohne Richtung. Die presentaion specification definiert das Aussehen der Komponente, auf die sich der specifier bezieht. Damit konkurriert sie natürlich mit der presentation specification der Komponente selbst.

6.5 Run-Time Layer

Der Run-Time Layer stellt dem Benutzer eine Oberfläche zur Interaktion mit dem Hypertextsystem zur Verfügung, indem es den Hypertext in geeigneter Weise präsentiert, z.B. durch Grafik oder Text auf einem Computer. Zum Betrachten oder zur Manipulation einer Komponente wird dazu eine Kopie derselben in einem Laufzeit-Speicher gehalten, was etwa dem Laden einer Webseite ins Browser-Fenster entspricht. Prinzipiell sind natürlich mehrere Instanzen einer Komponente gleichzeitig möglich. Jede dieser Instanzen bekommt daher einen eindeutigen *instatiation identifier* (IID). Links werden zur Laufzeit zu *link markers*. Diese werden fest in eine Komponente eingebunden, und verweisen fest auf einen Teil in einer anderen (oder derselben) Komponente. Eine Funktion bildet zur Laufzeit diese link markers auf die zugehörigen Anker ab.

Definition: Eine Session auf einem Hypertext besteht aus

- einem Hypertext
- einer Abbildung der IID auf die Komponenten
- einer History
- einer *runtime-resolver-Funktion*

- einer *instantiator-Funktion*
- einer *realizer-Funktion* und
- Instruktionen zum Arbeiten auf dem Hypertext (Session öffnen, Link folgen, ...).

Die runtime-resolver-Funktion entspricht im wesentlichen der resolver-Funktion des Storage Layers. Allerdings ist es ihr zusätzlich möglich, Informationen über den Benutzer oder die aktuelle Session zu benutzen, wie etwa die History. Diese enthält im Gegensatz zur History eines Webbrowsers nicht nur eine einfache Liste der zuletzt bearbeiteten Komponenten, sondern sämtliche Aktionen aus der aktuellen Session.

Die instantiator-Funktion erzeugt aus einem UID und einer presentation specification eine Instanz einer Komponente. Dabei benötigt sie natürlich die runtime-resolver-Funktion. Die instantiator-Funktion entscheidet letztendlich darüber, wie eine Komponente für den Benutzer aussieht. Dazu kombiniert sie die presentation specifications von Komponente und Link. Auch weitere Informationen, etwa solche über den Benutzer können hierbei einfließen. Z.B. kann ein Dokument so für Lehrer und Schüler unterschiedlich dargestellt werden.

Die inverse Funktion dazu ist der realizer. Er schreibt bearbeitete Komponenten wieder ins Hypertextsystem zurück.

6.6 Bemerkungen und Zusammenfassung

In bekannten Hypertextsystemen werden die Möglichkeiten, die aus dem Dexter Hypertext Referenzmodell hervorgehen, nur sehr begrenzt ausgenutzt. Die Möglichkeit der Komposition von Komponenten ist im WWW zum Beispiel auf das Einbetten von Bildern in eine Webseite und damit auf nur zwei Ebenen beschränkt. Allerdings ist es auch verständlich, dass ein formales Modell für spätere Realisierungen einen breiten Spielraum lässt. So ist zum Beispiel das Konzept der Verlinkung von Medienobjekten sehr vielfältig und wird in vollem Umfang in kaum einem bestehenden Hypertextsystem verwendet.

Links haben einerseits die bereits erwähnten Eigenschaften wie z.B. vier Richtungen oder die Möglichkeit, beliebig viele Endpunkte zu haben. In dem abstrakten Netzwerk, welches der Storage Layer beschreibt, werden Links auf einer Stufe mit anderen Medienobjekten ebenfalls als Komponenten angesehen. Dadurch ist es prinzipiell möglich, dass ein Link auch auf einen anderen Link zeigt. Ein weiterer wichtiger Aspekt ist die Konsistenz. Jeder specifier eines Links muss stets eine component specification haben, und damit auf eine Komponente verweisen. Damit kein Link ins Leere führt, ist es nötig, dass beim Löschen einer Komponente auch alle betreffenden Specifiers in den entsprechenden Links gelöscht werden. Ebenso werden zur Laufzeit auch alle Instanzen der Komponente sowie link marker gelöscht.

6.7 Diskussionsansätze, weiterführende Fragen

- Inwieweit soll das Dexter Modell in die Projektgruppe einfließen?

Das formale Modell mag strenge Vorgaben an ein Hypertextsystem stellen. Es stellt sich die Frage, inwieweit man sich am Modell orientieren soll oder vielleicht besser ein an bestehenden Hypertextsystemen orientiertes Cockpit entwirft.

- Welche Elemente des Modells erscheinen diesbezüglich sinnvoll, welche nicht?

  Komponenten und Links zur Verknüpfung dieser sind obligatorisch. Können aber die Möglichkeiten, die duch die Komposition von Komponenten und das Konzept der Verlinkung entstehen, im Projekt ausgenutzt werden? Wie soll das konkret geschehen?

- Welche Schwierigkeiten / Konflikte mit dem Referenzmodell könnten beim Entwurf des Hypermedialen Cockpits auftreten?

  Es mag Elemente im Modell geben, die sich im Cockpit nicht realisieren lassen, oder es sind vom Modell abweichende Elemente wünschenswert. Es kann also sinnvoll sein, sich in gewissem Maße vom Modell zu lösen. Dabei stellt sich allerdings die Frage, inwieweit das entworfene System dann noch als Hypertext bezeichnet werden darf.

Kapitel 7

HDM & OOHDM

Autor: *Bastian Krol*

7.1 Einleitung

Das Hypermedia Design Model (HDM) und die Object Oriented Hypermedia Design Method (OOHDM) befassen sich mit dem Entwurf von Hypermediaanwendungen. HDM ist eine Modellierungssprache, die sich vor allem mit der Verknüpfungsstruktur von Hypermediaanwendungen befasst. OOHDM hingegen ist eine objektorientierte Entwurfsmethode.

7.2 HDM

Das Hypermedia Design Model bietet eine Modellierungssprache, mit der Hypermediaanwendungen beschrieben und analysiert werden können. HDM beschäftigt sich dabei vor allem mit den Aspekten, welche die Autoren von [37] (der grundlegende Artikel über HDM von Garzotto et al.) als *authoring-in-the-large* bezeichnen. Damit ist der Entwurf und die Spezifikation der Verweisstruktur der Anwendung, also der Navigationsbeziehungen zwischen den Knoten (siehe Kapitel über das Dexter Modell, Abschnitt 1) gemeint. *Authoring-in-the-small*, also das Füllen der Knoten mit Inhalt, wird von HDM nicht betrachtet. Mit HDM sollen folgende Ziele erreicht werden (vgl. [37], Abschnitt 2.1):

- Die Modellierung der Anwendung soll die Kommunikation zwischen den Beteiligten (Analyst, Designer, Programmierer, Benutzer) erleichtern.
- Die Wiederverwendbarkeit der Modellierung und der daraus resultierenden Software soll erhöht werden.
- Die Modellierung soll helfen, Inkonsistenzen zu vermeiden.
- Die Modellierungssprache kann die Grundlage für Entwurfswerkzeuge bilden.
- Die Entwicklung von Entwurfsmethoden und Vorgehensmodellen auf Grundlage des Modells wird angestrebt.

Im wesentlichen sind das allerdings die Ziele jeder Modellierungssprache.

7.2.1 Überblick

Um einen Überblick über HDM zu geben, zählen wir einige der Modellierungselemente auf, die HDM enthält: *Entitäten* sind (relativ grosse) Informationsstrukturen, die reale Objekte aus der Anwendungsdomäne repräsentieren. Entitäten setzen sich aus *Komponenten* zusammen, die in einer baumartigen Hierarchie angeordnet sind. Ein Beispiel wäre die Entität „La Traviata“, eine Oper von Giuseppe Verdi, mit den Komponenten „Ouvertüre“, „Akt 1“, „Akt 2“ und „Akt 3“. Komponenten wiederum enthalten *Einheiten*. Da in Hypermediaanwendungen dieselben Informationen oft aus unterschiedlichen Blickwinkeln präsentiert werden sollen, führt HDM *Perspektiven* ein. Eine Komponente enthält für jede gegebene Perspektive genau eine Einheit. Einheiten dienen als Behälter für die Informationen, welche die Hypermediaanwendung darstellen soll.

Entitäten können mit dem Mittel des *Entitätentyps* klassifiziert werden. Dieser legt unter anderem die Perspektiven für seine Instanzen fest. In unserem Beispiel hätte der Entitätentyp „Oper“ die beiden Perspektiven „Musik/Audio“ und „Partitur“. Verdi's „La Traviata“ wäre dann eine Instanz dieses Entitätentyps, und da der Entitätentyp „Oper“ zwei Perspektiven hat, besitzt auch jeder Akt und die Ouvertüre die beiden angegebenen Perspektiven. Ein weiterer Entitätentyp dieser Anwendung könnte „Komponist“ sein. Abbildung 7.1 zeigt den Komponentenbaum für die Entität „LaTraviata“.

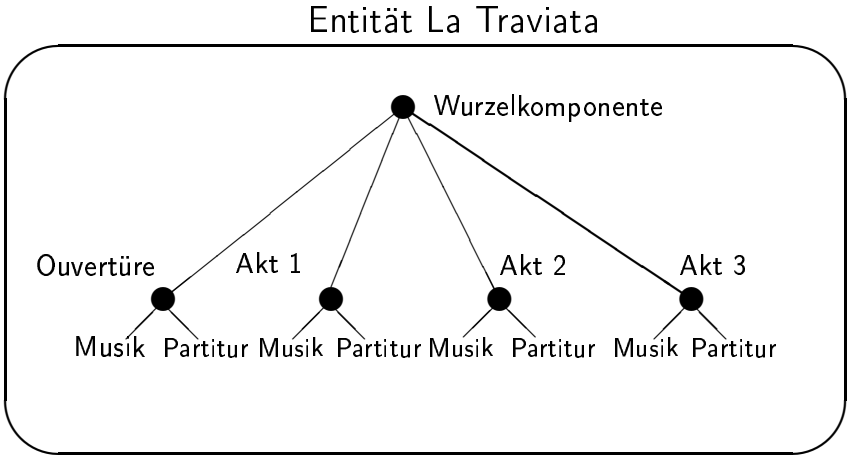

Abbildung 7.1: Die Entität „La Traviata“, eine Ausprägung des Entitätentyps Oper

7.2.2 Verweise in HDM

Ein wichtiger Aspekt in HDM ist die Verweisstruktur der Anwendung. Alle Einheiten derselben Komponente sind untereinander durch *Perspektivenverweise* verbunden. Das Aktivieren eines Perspektivenverweises entspricht also dem Wechsel der Perspektive. Die Komponenten einer Entität sind untereinander wiederum durch *strukturelle Verweise* verbunden, die im wesentlichen der Baumstruktur des Komponentenbaums entsprechen. Diese beiden Arten von

Verweisen können offensichtlich zu einem großen Teil automatisch aus dem Modell abgeleitet werden. In unserem Beispiel gibt es also einen Perspektivenverweis von der Musik der Ouvertüre zu ihrer Partitur und umgekehrt, ebenso für die drei Akte. Weiterhin könnte es Komponentenverweise der Art „nächster Akt“ bzw. „vorheriger Akt“ geben.

Zusätzlich kann der Designer *Anwendungsverweise* definieren, die domänenabhängigen Beziehungen zwischen den Komponenten oder Entitäten entsprechen. Anwendungsverweise können durch *Verweistypen* spezifiziert werden. Ein Verweistyp legt den Quell- und den Ziel-Entitätentyp fest. Der Verweistyp „ist Komponist von“ könnte den Quell-Entitätentyp „Komponist“ und den Ziel-Entitätentyp „Oper“ haben. Eine Ausprägung dieses Verweistyps wäre der Verweis von der Entität „Giuseppe Verdi“ auf die Entität „La Traviata“.

7.2.3 Browsing Semantics

Die konkrete Hypermediaanwendung, die aus dem Modell resultiert, wird zu einem großen Teil von ihrer *browsing semantics* bestimmt. Der Begriff browsing semantics fasst die Antworten zu folgenden Fragen zusammen:

- Welche Objekte sollen dem Benutzer als Ganzes angezeigt werden, d.h. welche Konstrukte (Einheiten, Komponenten oder Entitäten) sollen die *Knoten* darstellen?
- Wie werden die Verweise des Modells auf die *konkreten Verweise* auf der gewählten Hypermediaplattform abgebildet?

HDM definiert hier eine (relativ simple) *default browsing semantics*. In dieser entspricht ein Knoten genau einer Einheit, dem Benutzer wird also zu jedem Zeitpunkt genau eine Einheit angezeigt. Demzufolge kann der Benutzer auch nur Verweise von einer Einheit auf eine andere wahrnehmen, und nicht zwischen Komponenten oder Entitäten. Daher werden Verweise zwischen Einheiten in der default browsing semantics als *konkrete* Verweise bezeichnet. Perspektivenverweise sind ein Beispiel für konkrete Verweise. Strukturelle Verweise und Anwendungsverweise bestehen zwischen Komponenten bzw. zwischen Entitäten. Diese *abstrakten* Verweise müssen nun geeignet auf konkrete Verweise, also auf Verweise zwischen Einheiten abgebildet werden. Dazu erhält jeder Entitätentyp eine default-Perspektive. Abstrakte Verweise zwischen Entitäten werden abgebildet auf einen abstrakten Verweis zwischen ihren Wurzelkomponenten, in diesem Sinne steht die Wurzelkomponente stellvertretend für die ganze Entität. Abstrakte Verweise zwischen Komponenten werden dann abgebildet auf eine Menge von konkreten Verweisen, so dass jede Einheit der Quellkomponente auf die Einheit der Zielkomponente unter der default-Perspektive verweist.

Abgesehen von der default browsing semantics kann natürlich jeder Autor eine eigene browsing semantics definieren, die seinen Anforderungen entspricht. Es wäre zum Beispiel denkbar, ganze Entitäten als Knoten anzusehen und alle Einheiten einer bestimmten Perspektive gleichzeitig anzuzeigen. Strukturelle Verweise wären dann natürlich überflüssig und Anwendungsverweise wären in diesem Fall konkrete Verweise.

7.3 OOHDM

Die „Object Oriented Hypermedia Design Method“ ist ein modellbasierter Ansatz zum Entwerfen und Implementieren großer Hypermediaanwendungen. Die hier folgende Beschreibung dieser Methode bezieht sich im wesentlichen auf die Artikel [74] und [73].

7.3.1 Überblick

Hypermediaanwendungen beschäftigen sich oft mit einer komplexen Anwendungsdomäne und ermöglichen typischerweise eine komplexe Navigation. OOHDM versucht, die daraus resultierenden Schwierigkeiten beim Entwurf von Hypermediaanwendungen zu entschärfen, indem es den Entwurfsprozess in vier Phasen unterteilt:

- Entwurf des konzeptionellen Modells
- Navigationsentwurf
- Entwurf der abstrakten Nutzungsschnittstellen
- Implementierung

Diese Einteilung soll helfen, den Entwurfsprozess systematisch zu gestalten. Jede dieser Phasen beschäftigt sich mit einem bestimmten Aspekt des Entwurfs, daher werden die einzelnen Gesichtspunkte (Anwendungsdomäne, Navigation, Nutzungsschnittstellen, Implementierung) sauber voneinander getrennt. Weiterhin benutzt OOHDM objektorientierte Modellierungsprinzipien wie Aggregation, Generalisierung und Spezialisierung. Aufgrund der Namensgebung könnte man vermuten, dass OOHDM syntaktisch oder konzeptionell auf HDM aufbaut. Zumindest für die Syntax ist dies ist nicht der Fall. Es wurden allerdings einige Konzepte, z.B. die Perspektiven, siehe Abschnitt 7.3.2 von HDM übernommen. Syntaktisch wird in weiten Teilen die UML benutzt.

Insgesamt sollen folgende Ziele verfolgt werden:

- Die Trennung des konzeptionellen Modells von der Navigationsstruktur.
- Hypermediaanwendungen ohne großen Zusatzaufwand für verschiedene Benutzergruppen mit unterschiedlichen Anforderungsprofilen nutzbar machen.
- Die strikte Trennung von Entwurf und Implementierung.

OOHDM trägt den ersten beiden Zielen Rechnung, indem in der ersten Phase ein konzeptionelles Modell der gesamten Anwendungsdomäne erstellt wird, ohne die Besonderheiten der einzelnen Benutzergruppen mit in Betracht zu ziehen. Erst in der zweiten Phase werden aus den konzeptionellen Objekten Navigationsobjekte zusammengesetzt, indem Navigationsobjekte als Sichten auf die konzeptionellen Objekte aufgefasst werden (ähnlich einer Datenbanksicht). Dieses Navigationsmodell ist auf die Bedürfnisse und Aufgaben einer bestimmten Benutzergruppe abgestimmt sind. Es ist also möglich, dasselbe konzeptionelle Modell für alle Benutzergruppen wiederzuverwenden. In der dritten Phase werden dann auf Grundlage der Navigationsstruktur

Schnittstellenobjekte definiert, die dem Benutzer Informationen anzeigen und mit denen der Benutzer interagieren kann. Hier soll eine Trennung von Navigationsaspekten und Nutzungsschnittstellenaspekten erreicht werden. Die Schnittstellenobjekte sind in dem Sinne abstrakt, als dass die konkrete Plattform, auf der die Anwendung realisiert wird, noch nicht betrachtet wird. Bis hierhin verläuft der Prozess also unabhängig von der Implementierung. Die Autoren von [73] argumentieren hier, dass es Entwurfsentscheidungen gibt, die erst bei der Implementierung getroffen werden müssen und sollten. In der vierten Phase wird dann die gewählte Hypermediaplattform mit einbezogen und das Modell auf dieser Plattform umgesetzt. (vgl. [73], Seite 3, Absatz „The cornerstones of the OOHDM approach")

Dem Entwurf der Anwendung mit OOHDM geht üblicherweise eine Anforderungsanalyse voraus, in dem mindestens die beteiligten Benutzergruppen, deren Aufgaben und Anforderungen an die Anwendung identifiziert werden. Wir betrachten nun die vier Phasen im Detail.

7.3.2 Entwurf des konzeptionellen Modells

In dieser Phase geht es darum, ein Klassenmodell der Anwendungsdomäne zu erstellen, welches die Semantik der Anwendungsdomäne korrekt erfasst. In dieser Phase unterscheidet sich OOHDM nicht wesentlich von anderen Entwurfsmethoden für Anwendungen außerhalb des Hypermediaparadigmas, daher unterscheidet sich auch das Klassenschema und die darin benutzten Modellierungselemente kaum von der gebräuchlichen UML-Notation (z.B. [33]). Wohlbekannte Modellierungsprinzipien wie Aggregation, Komposition und Vererbung stehen beim Entwurf des konzeptionellen Modells natürlich zur Verfügung.

Abbildung 7.2 zeigt das Klassenschema für ein Onlinemagazin. In diesem Modell gibt es Stories, die Essays, Translations oder Interviews sein können. Weiterhin gibt es die Personen, die Autoren von Geschichten sein können und Interviews geben. Interviews wiederum sind eine Ansammlung von Fragen und Antworten (Klasse Q&A).

Ein Spezialelement der Notation, mit dem UML hier erweitert wird, sind *Perspektiven*. Der Typ eines Attributs wird in der OOHDM-Terminologie als Perspektive bezeichnet, ähnlich den Perspektiven in HDM (siehe Abschnitt 7.2.1). Das Attribut Illustration der Klasse Essay in Abbildung 7.2 hat die beiden Perspektiven Photo und Video, was durch die Notation [Photo+, Video] gekennzeichnet wird. Das „+" markiert die default-Perspektive. Diese muss in jedem Exemplar der Klasse vorhanden sein, die anderen Perspektiven können vorhanden sein oder fehlen.

Ein wichtiger Punkt beim konzeptionellen Entwurf ist, dass das erstellte Modell die Semantik der Anwendungsdomäne möglichst allgemeingültig erfasst. Insbesondere werden hier die Besonderheiten der einzelnen Benutzergruppen möglichst ganz ausgeblendet.

7.3.3 Navigationsentwurf

In dieser Phase wird in zwei Schritten die Navigationsstruktur der Anwendung entworfen. Im ersten Schritt wird ein Navigationklassensmodell erstellt. Dieses legt fest, wie die Navigationsobjekte beschaffen sein sollen, und welche Beziehungen zwischen ihnen existieren. Navigationsobjekte gleichen den Knoten in der Hypertextterminologie, Beziehungen zwischen

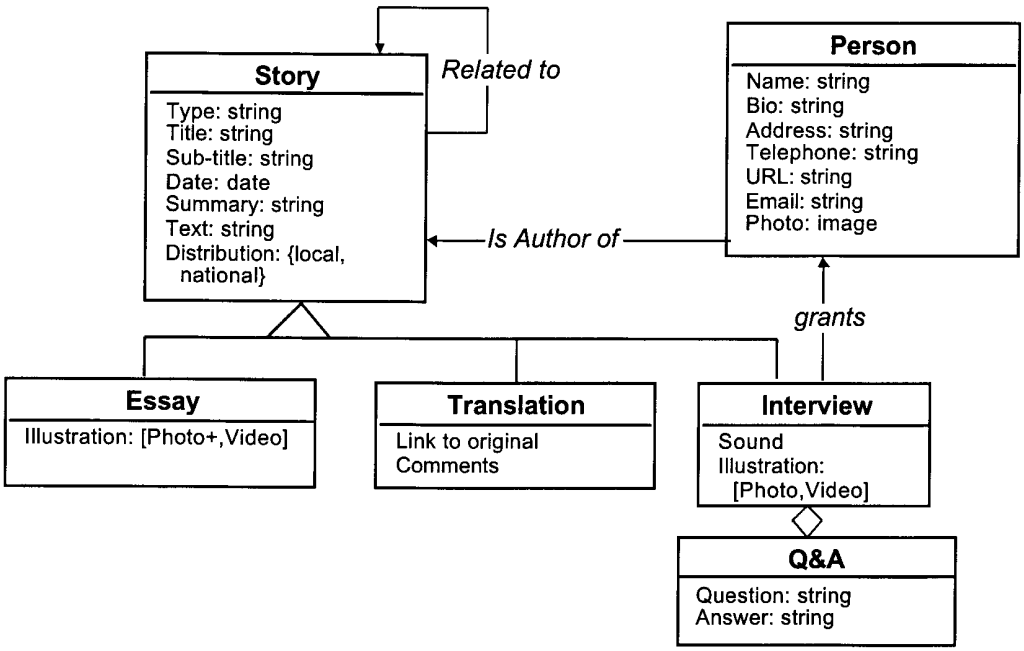

Abbildung 7.2: Konzeptionelles Modell eines Onlinemagazins

ihnen stellen Verweise dar. Die Attribute eines Navigationsobjektes sind die Inhalte, die dem Benutzer später an diesem Knoten präsentiert werden.

Im Gegensatz zur ersten Phase werden beim Navigationsentwurf die besonderen Anforderungen der einzelnen Benutzergruppe mit einbezogen. Für jede Benutzergruppen wird also ein eigenes Navigationsklassenmodell erstellt, das auf deren Aufgaben und Erwartungen abgestimmt ist.

Die Navigationsklassen und -beziehungen setzen sich aus den konzeptionellen Klassen und Beziehungen zusammen. In diesem Sinne ist jedes Navigationsklassenmodell eine Sicht auf das konzeptionelle Modell. Dies bedeutet, dass eine Navigationsklasse Attribute verschiedener konzeptioneller Klassen enthalten kann, deren Attributwerte sich dann aus einer Anfrage an das konzeptionelle Modell ergeben. Somit kann das in der ersten Phase erstellte konzeptionelle Modell für alle Benutzerklassen wiederverwendet werden.

Abbildung 7.3 zeigt das Navigationsklassenmodell für das Onlinemagazin für die Benutzergruppe „Leser“. Die Klasse *Person* ist in diesem Modell nicht mehr vorhanden, vielmehr sind die Attribute des Autors, die für den Leser von Interesse sind, nun Attribute der Navigationsklasse Story. Es handelt sich um die beiden Attribute Author und Author_Bio. Um die Beziehung dieser Attribute zum konzeptionellen Modell zu beschreiben, bietet OOHDM einen SQL-ähnlichen Syntax (siehe [46]). Die Definition des Attributs Author lautet: Author: string [SELECT Name] [FROM] Person:Pr WHERE Pr Is Author of St]. Dies bedeutet nichts anderes, als dass das Attribut Author einer Story als Wert den Namen derjenigen Person erhält, für die die Beziehung Person Is Author Of Story gilt, also gerade den Namen der Person, die diese Story geschrieben hat. Das gleiche gilt für das Attribut Author_Bio.

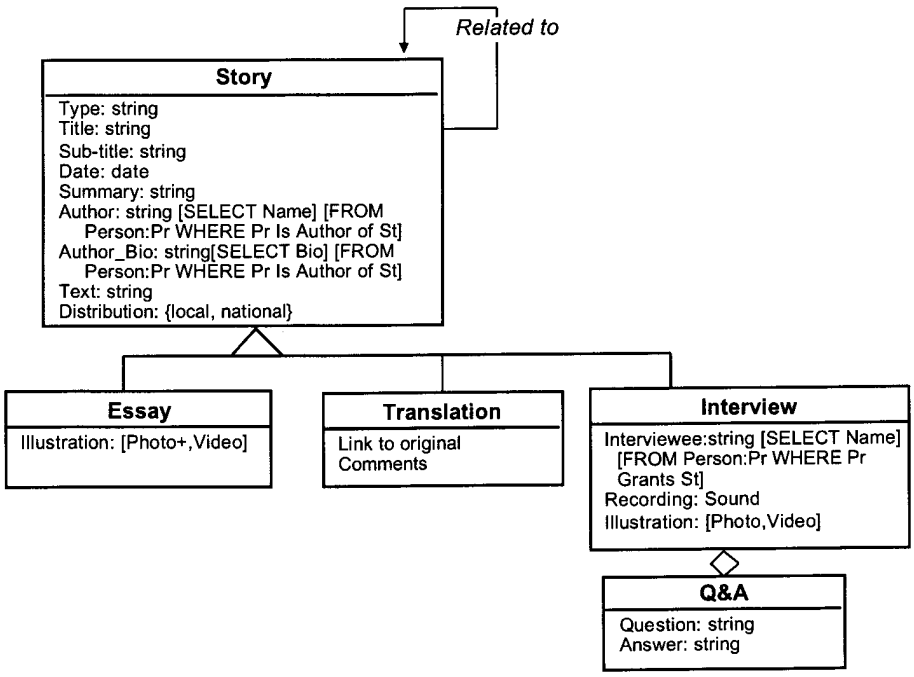

Abbildung 7.3: Navigationsmodell des Onlinemagazins

Der zweite Schritt der Phase Navigationsentwurf besteht darin, den Navigationsraum für den Benutzer zu strukturieren. Zu diesem Zweck bietet OOHDM das Modellierungselement der *Kontexte* und das *Kontextdiagramm*. Ein Kontext definiert eine Menge von Navigationsobjekten (also Knoten) und ihre interne Verweisstruktur. Außerdem kann ein Kontext weitere (geschachtelte) Kontexte enthalten. Die Menge der Navigationsobjekte kann unter anderem auf folgende Art und Weise gebildet werden:

Klassenbasiert: Die Menge aller Objekte einer Klasse K, die einer bestimmten Bedingung genügen, z.B. enthält $\mathsf{StoriesOfApril2002} = \{\mathsf{story} \mid \mathsf{story} \in \mathsf{Story} \wedge \mathsf{story.date} \geq 01.04.2002 \wedge \mathsf{story.date} < 01.05.2002\}$ alle Exemplare der Klasse Story, die im April 2002 erschienen sind. (Anmerkung: Die Notation $\mathsf{story} \in \mathsf{Story}$ bedeutet hier, dass story ein Exemplar der Klasse Story ist.)

Klassenbasierte Gruppe: Eine Menge von klassenbasierten Kontexten. So könnte z.B. StoriesByMonth eine Menge von Kontexten sein, in der jeder Kontext alle Stories für einen bestimmten Monat enthält.

Verweisbasiert: Die Menge aller Objekte einer Klasse K, die mit einem bestimmte Objekt in Beziehung stehen, z.B. die Menge aller Stories von Wladimir Kaminer: $\mathsf{StoriesOfWK} = \{\mathsf{story} \mid \mathsf{story} \in \mathsf{Story} \wedge \mathsf{IsAuthorOf(Wladimir\ Kaminer, story)}\}$.

Verweisbasierte Gruppe: Menge von verweisbasierten Kontexten. StoriesByAuthor könnte z.B. eine Menge von Kontexten sein, in der jeder Kontext alle Stories eines bestimmten Autors enthält.

Aufzählung: Hier wird die Menge der Navigationsobjekte einfach aufgezählt. Ein typisches Beispiel wäre eine Guided Tour.

Im Kontextdiagramm werden dann die definierten Kontexte in Beziehung zueinander gesetzt. Die Navigation zwischen den Kontexten wird beschrieben und Zugriffsstrukturen für die Kontexte werden definiert. Zugriffsstrukturen können im einfachsten Fall Indexe sein, aber auch andere, z.B. dynamische Zugriffsstrukturen sind möglich. Beispiele für dynamische Zugriffsstrukturen wären der „Warenkorb", den der Benutzer während der Navigation selber aufbaut oder die History Funktion, die es bei einigen Browsern gibt.

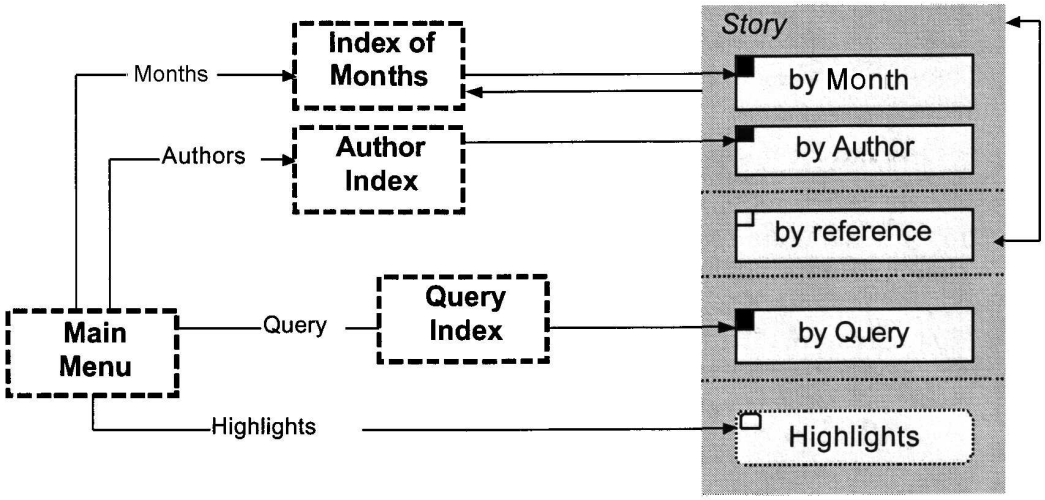

Abbildung 7.4: Kontextdiagramm für das Onlinemagazin

Abbildung 7.4 zeigt das Kontextdiagramm für das Onlinemagazin. Die fett gestrichelt umrandeten Rechtecke sind Zugriffsstrukturen, das graue Rechteck auf der rechten Seite enthält die Kontexte für die Klasse Story. Der Kontext Story by Author ist ein Beispiel für eine verweisbasierte Gruppe von Kontexten, der Kontext Highlights ist ein Beispiel für eine Aufzählung. Ein schwarzes Quadrat in der linken, oberen Ecke zeigt an, dass ein Kontext eine Zugriffsstruktur besitzt.

7.3.4 Entwurf der abstrakten Nutzungsschnittstellen

In dieser Phase werden die für den Benutzer sichtbaren Objekte, ihr Layout und ihre Funktionalität definiert. OOHDM benutzt hierfür den *Abstract Data View* Ansatz [20]. ADVs sind Objekte; sie haben interne Zustände und Schnittstellen. Diese Schnittstellen können über externe (insbesondere benutzergenerierte) Ereignisse angesprochen werden. ADVs können voneinander erben, außerdem können ADVs Aggregationen anderer ADVs sein so dass man komplexere ADVs aus primitiven Bausteinen (z.B. Texteingabefelder, Knöpfe, etc.) zusammensetzen kann.

Abbildung 7.5 zeigt ein einfaches ADV-Diagramm für Navigationsobjekte der Klasse Story. Im oberen Teil wird der Titel und der Untertitel, sowie der Name des Autors angezeigt, danach

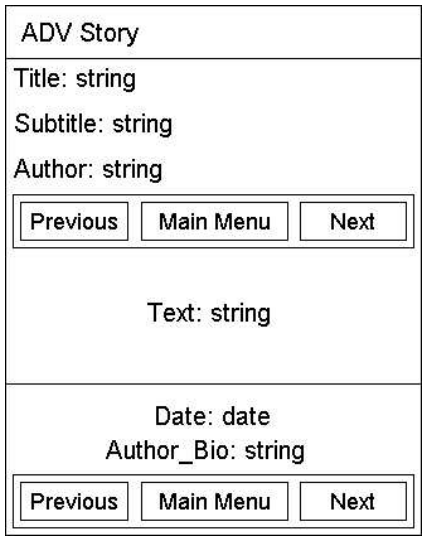

Abbildung 7.5: ADV Story

folgt eine Navigationsleiste, dann der Text der eigentlichen Geschichte. Im unteren Teil wird nach der Geschichte ihr Erscheinungsdatum und die Biographie des Autors angezeigt.

In OOHDM werden ADVs benutzt, um folgendes zu spezifizieren:

- Das Layout der Nutzungsschnittstellenobjekte. Im ADV-Diagramm kann dann festgelegt werden, wo welche Attribute dargestellt werden, und wie die Bedienelemente platziert werden.
- Wie sie mit den Navigationsobjekten in Beziehungen stehen. Die Attribute der Navigationsobjekte sind die Inhalte die im ADV angezeigt werden.
- Wie sie auf Benutzereingaben reagieren. Dabei wird zwischen lokalen Schnittstellentransformationen und Navigationsoperationen unterschieden. Navigationsoperationen resultieren in einem Wechsel des aktuellen Knotens, während lokale Transformationen nur das Aussehen oder Verhalten der Nutzungsschnittstelle verändern.

ADVs sind in dem Sinne abstrakt, als dass sie plattformunabhängig sind und von der Implementierung abstrahieren, sie beschreiben nur, wie die Nutzungsschnittstellen aussehen und sich verhalten, nicht aber wie sie realisiert werden.

7.3.5 Implementierung

In dieser Phase wird, im Gegensatz zu den vorherigen Phasen, die gewählte Hypermediaplattform mit in Betracht gezogen und das Modell darauf umgesetzt. Natürlich ist es auch möglich, dass Modell auf verschiedene Plattformen umzusetzen, die Trennung von Entwurf und Implementierung vereinfacht dies. Wie genau bei der Implementierung vorzugehen ist, hängt stark von der tatsächlichen Plattform ab. Wir wollen hier kurz die Umsetzung mit einer objektorientierten Programmiersprache wie Java betrachten.

Wenn die Plattform objektorientiert ist, liegt es nahe, sowohl das konzeptionelle Modell als auch das Navigationsklassenmodell direkt umzusetzen. Die konzeptionellen Objekte fungieren dann im wesentlichen als Datenbehälter, während die Navigationsobjekte ihre Attributewerte durch Anfragen an die konzeptionellen Objekte berechnen. Die durch ADVs spezifizierten Nutzungsschnittstellen können offensichtlich mit Hilfe eines GUI-Frameworks wie AWT oder Swing umgesetzt werden. Wenn eine nicht vollständig objektorientierte Datenbank als Backend benutzt werden soll, was bei umfangreichen Hypermediaanwendungen sicherlich sinnvoll ist, müssen die modellierten Klassen geeignet auf Datenbanktabellen abgebildet werden.

7.4 Kommentar zu den Literaturangaben

[37] enthält eine vollständige und gut verständliche Beschreibung von HDM. [71] ist einer der ersten Artikel zur OOHDM, die zentralen Ideen und Ansätze sind hier zwar schon erkennbar, allerdings noch nicht ganz ausgereift. Der wenig später erschienene Artikel [72] gibt eine sehr knappe Übersicht über die OOHDM und kann nur als Überblick dienen. Detaillierte Informationen zu den vier Phasen sind hier nicht enthalten. In [74] und [73] wird die Methode vollständig und detailliert beschrieben, wobei [73] noch genauer auf die benutzte Modellierungssprache (z.B. das Kontextdiagramm) eingeht. Weiterhin wird in [73] die Umsetzung einer OOHDM Modellierung auf eine webbasierten Plattform detailliert besprochen.

Kapitel 8

XML

Autor: *Oliver Szymanski*

Einleitung

Der folgende Text handelt über XML im Allgemeinen, beschreibt wie man XML-Dokumente einliest, die Daten darin verarbeitet und nennt Linkkonzepte, die man in XML-Dokumenten benutzen kann.

8.1 Allgemeines zu XML

Im Folgenden wird eine Einführung in XML gegeben, die Syntax von XML erläutert, bestimmte Sprachregelungen im Umfeld von XML genannt, XML mit dem Vorgänger SGML verglichen und Document Type Definition für das Beschränken der Struktur von XML-Dokumenten beschrieben.

8.1.1 Einführung in XML

XML heißt „eXtensible Markup Language“ (erweiterbare Auszeichnungssprache) [53][13]. Eine Auszeichnungssprache ist eine Syntax, mit der man die Struktur von Dokumenten angeben kann. XML ist wie der Vorgänger SGML (Standardized Generalized Markup Language) eine Metasprache, mit der man andere Sprachen definieren kann. XML ist jedoch viel einfacher und weniger umständlich als SGML. Auch unterliegt XML stärker einschränkenden Regeln, um XML leichter zu erlernen, und um einfacher Programme entwickeln zu können, welche ein XML-Dokument verarbeiten. Die vollständige XML-Spezifikation findet sich unter [13].

Mit XML kann man Daten auf verschiedene Weisen strukturiert ablegen. XML ist also nicht auf eine Dokumentstruktur festgelegt, sondern lässt die Definition beliebiger Strukturen zu.

8.1.2 Syntax von XML Dokumenten

XML Dokumente bestehen, wie auch aus der SGML Sprache HTML bekannt, aus Tags. Diese werden in XML Elemente genannt. Ein Element ist ein in spitzen Klammern stehendes Wort, welches jeweils als Start und End-Element, letzteres mit dem Wort vorangestelltem Schrägstrich, ein Paar bilden. Es muss einen sogenannten Root-Element geben, der alles andere umschließt, darin können beliebige andere Elemente geschachtelt werden. Ein Element kann einen Inhalt (inklusive weiterer Elemente) und Attribute besitzen, die innerhalb der spitzen Klammern als Name/Wert Paare angegeben werden. Alle Elemente müssen in XML im Gegensatz zu HTML und SGML in der richtigen Reihenfolge wieder durch einen End-Element geschlossen werden. Dies nennt man „well-formed“ (wohlgeformt). Im Gegensatz zu HTML gibt es keine Menge bestimmter erlaubter Elemente, sondern die Elemente sind frei wählbar. Auch wird in XML streng zwischen Groß- und Kleinschreibung unterschieden.

Bsp.: XML

```
<adressverzeichnis>
  <eintrag name="Oliver">
    <adresse typ="email">osz@bov.de</adresse>
    <adresse typ="fax">+49 1212-5-127-83-631</adresse>
  </eintrag>
</adressverzeichnis>
```

Hier ist „addressverzeichnis“ das Root-Element mit dem Element „eintrag“ als Inhalt. Das Element „eintrag“ hat als Attribut „name“ mit dem Wert „Oliver“ und als Inhalt zwei Elemente „adresse“, welche als Attribut jeweils „typ“ mit Wert „email“ / „fax“ und lediglich jeweils eine eMail-Adresse / Faxnummer als Inhalt haben.

Bei Elementen, die keinen Inhalt haben, kann man Start- und End-Element auch zusammenfassen, in dem man den Schrägstrich vor die spitze Klammer am Ende des Start-Elementes einfügt und das End-Element weglässt. Wollte man eine HTML-Datei also XML konform umschreiben, muss man z.B. unter anderem das „
“-Element für einen Zeilenumbruch durch „
“ ersetzen, da in HTML dieses Element nicht unbedingt geschlossen werden muss, dies aber in XML erforderlich ist.

XML konforme Dokumente sind Dokumente, die aus Elementen bestehen, welche genau ein umschließendes Root-Element besitzen, und in dem alle Elemente geschlossen werden, wobei beim Schließen die richtige Reihenfolge eingehalten werden muss.

Bsp.: Schließen von Elementen in falscher Reihenfolge:

```
<adressverzeichnis>
  <eintrag name="Oliver">
  <adresse typ="email">osz@bov.de
  </eintrag>
  </adresse>
</adressverzeichnis>
```

XML ist ein Formalismus, mit dem sich Daten strukturiert aufschreiben lassen. XML gibt nicht vor, welche Namen, Attribute und sonstiges zugelassen sind, die Grammatik für Dokumente, die XML konform sind, ist also nicht vom XML Standard festgelegt.

8.1.3 Sprachregelungen im XML Kontext: URI, URN, URL

Eine Uniform Resource Identifier (URI) ist eine Zeichenfolge zur Kennzeichnung einer Ressource (z. B. einer Datei) im Internet über Typ und Adresse. Unter dem Begriff Uniform Resource Identifier werden die Adressformen Uniform Resource Name (URN) und Uniform Resource Locator (URL) zusammengefasst.

Eine URL (Uniform Resource Locator) darf weltweit nur einmal vorhanden sein, da eine URL eine Ressource weltweit eindeutig lokalisiert. Internet-Adressen sind z.B. URL. URL sehen wie folgt aus: `resource_type://user:password@host.domain:port/path`.

URN (Uniform Resource Name) kennzeichnen eindeutig Ressourcen, welche im Internet verfügbar sein können, anhand des Namens. Im Gegensatz zu URL berücksichtigen sie nicht, wo die Ressourcen tatsächlich hinterlegt sind.

Näheres zu URI, URL und URN findet man unter [19].

8.1.4 Vergleich von XML und SGML

XML ist eine SGML Sprache, aber es gibt einige Einschränkungen im Vergleich zu SGML, wie in [53] und [13] beschrieben. XML unterscheidet im Gegensatz zu SGML zwischen Groß- und Kleinschreibung, und ein Element muss aus Start-Tag und End-Tag bestehen. D.h. in SGML kann ein Element zwar mit „`<tagname>`“ eingeleitet werden, es muss jedoch kein „`</tagname>`“ folgen. Auch müssen Elemente nicht in gleicher Reihenfolge geschlossen werden (siehe auch bei der SGML Sprache HTML). Es wurde mit XML eine einfache Variante von SGML geschaffen, in dem man die nützlichen Features von SGML übernommen hat und die eher nicht gebräuchlichen und komplizierten Sprachbestandteile weggelassen hat. Somit wurde die Arbeit mit XML im Vergleich zu SGML erheblich erleichtert.

8.1.5 Document Type Definition (DTD)

Eine DTD legt Einschränkungen für die XML Struktur eines XML-Dokumentes anhand von Regeln fest und ist somit eine Grammatik für XML-Dokumente [1,2]. Im Normalfall ist eine DTD eine zusätzliche Datei, auf die im XML Dokument verwiesen wird. Statt dieser externen DTDs kann man DTDs auch intern zu Beginn eines XML-Dokumentes angeben.

Anhand einer DTD kann man überprüfen, ob ein XML-Dokument diese Regeln einhält. Ein XML-Dokument, welches den Regeln der im XML-Dokument verwiesenen DTD entspricht, nennt man „valid“ (zulässig). Ein XML-Dokument kann folglich „well-formed“ sein, ohne das es „valid“ ist.

Durch Definition einer Grammatik in Form einer DTD kann man die erlaubten Sätze in einem XML-Dokument einschränken. Eine DTD besteht aus Regeln für Elemente und Regeln

für Attribute, die zeilenweise aufgelistet werden. Regeln für Elemente sehen wie folgt aus:

```
<!ELEMENT Elementname Typ>
```

Der Elementname gibt an, auf welches Element sich diese Regel bezieht. Der Typ gibt an, wie das Element auszusehen hat, also was innerhalb des Elementes erlaubt ist.

Im folgenden Beispiel wird für das Element „`adressverzeichnis`“ festgelegt, dass es aus beliebig vielen Elementen „`eintrag`“ bestehen darf. Für das Element „`eintrag`“ wird festgelegt, dass es aus mindestens einem Element „`adresse`“bestehen muss, und für das Element „`adresse`“ wird festgelegt, dass es sich bei dem Inhalt dieses Elementes um Zeichendaten handelt.

Bsp.: Element DTD-Regeln für vorheriges XML Beispiel aus 1.2

```
<!ELEMENT adressverzeichnis (eintrag*)>
<!ELEMENT eintrag (adresse+)>
<!ELEMENT adresse (#PCDATA)>
```

Regeln für Attribute sehen wie folgt aus:

```
<!ATTLIST Elementname
          Attributname Typ Modifikator
          ...
>
```

Der Elementname gibt an, auf welches Element sich diese Regel bezieht. Der Typ gibt an, welche Werte das Attribut annehmen kann. Es gibt den Typ CDATA für Zeichendaten und einen Aufzählungstypen, wenn nur bestimmte Werte benutzt werden sollen. Für Aufzählungen gibt man den Typ an, in dem man die erlaubten beliebig gewählten Werte in Klammern setzt. Der Modifikator gibt an, ob das Attribut optional ist. Anstelle eines Modifikators kann man auch einen Defaultwert angeben.

Im folgenden Beispiel wird für das Attribut „`name`“ des Elementes „`eintrag`“ als Werte ausschließlich Zeichendaten erlaubt und es wird angegeben, dass das Attribut nicht optional ist. Für das Attribut „`typ`“ des Elementes „`adresse`“ wird eine Liste mit erlaubten Werten angegeben und der Attributwert als Standard auf „`text`“ gesetzt.

Bsp.: Attlist DTD-Regeln für vorheriges XML Beispiel 1.2

```
<!ATTLIST eintrag
         name CDATA #REQUIRED
>
<!ATTLIST adresse
          typ (email | fax | handy | telefon | text) "text"
>
```

8.1.6 XML-Schema

XML-Schema ist ein neuer Ansatz zur Definition von Dokument Typen und damit zur Spezifikation von XML-Sprachen. Schemas sind selber XML Dateien. Allerdings wird XML Schema selbst wieder durch eine DTD beschränkt, anhand der überprüft wird, ob das Schema gültig ist.

XML-Schemata sollen die bislang benutzten Dokumenttyp-Definitionen (DTD's) mittelfristig ablösen. Im Vergleich zu DTD's bieten XML-Schemata mehr Möglichkeiten, indem sie beispielsweise die Zuordnung von Elementen zu bestimmten Datentypen erlauben. Außerdem lassen sich vorhandene Schemata in eigene einbinden und überschreiben. XML-Schema ermöglicht die Definition neuer Elementtypen auf Basis vorangegangener Definitionen.

XML-Schemas werden hier nur erwähnt und nicht näher beschrieben. Detaillierte Informationen findet man in [53] und [14].

8.2 Einlesen und Verarbeiten von XML-Dokumenten

Im Folgenden werden zwei Konzepte zum Einlesen und Verarbeiten von XML-Dokumenten näher beschrieben, die Simple API for XML und das Document Object Model.

8.2.1 SAX zum Parsen von XML

Um ein XML-Dokument zu verarbeiten muss man dieses zuerst mit Hilfe eines XML-Parsers einlesen, dieser verarbeitet das XML-Dokument und stellt die Daten zur Verfügung. Manche Parser können ein XML-Dokument auch beim Parsen anhand einer DTD validieren.

Eine Möglichkeit, ein XML zu parsen und im eigenen Programm die Daten zu erhalten, ist die Simple API for XML (SAX) [1,8]. Hier wird SAX 2.0 beschrieben. Die SAX API beinhaltet einen Mechanismus, der ein XML-Dokument sequentiell verarbeitet, und bei Start- und Ende von Elemente, bei auftretenden Fehlern oder anderen Ereignissen Callbacks an vorher angemeldete Objekte sendet. SAX beschreibt also nicht, was mit den Daten passiert, es liefert nur einen Mechanismus, um ein XML-Dokument sequentiell zu verarbeiten und alle auftretenden Daten dem eigenen Programm zur Verfügung zu stellen. SAX beinhaltet dafür Handler-Interfaces, welche man zu implementieren hat und dann dem Parser übergibt, bevor man ihn das XML-Dokument parsen lässt. Der Parser ruft dann entsprechende Methoden dieser Handler-Objekte auf, je nachdem, welche Stelle im XML-Dokument gerade verarbeitet wird.

Ein möglicher Ablauf mit SAX zu arbeiten wäre die entsprechenden und benötigten Handler-Interfaces zu implementieren, die Handler-Objekte und ein Parser-Objekt (welches in SAX fertig enthalten ist) zu instantieren und die Handler-Objekte beim Parser-Objekt zu registrieren, so dass dieses beim Parsen Methoden der Handler-Objekte aufrufen kann [54].

SAX ContentHandler

Zur Verarbeitung der Daten in einem XML-Dokument ist das Interface ContentHandler das wichtigste Handler-Interface. Es beinhaltet u.a. die wichtigen Methoden:

startDocument Wird vom Parser aufgerufen, wenn er beginnt das XML-Dokument zu parsen.

endDocument Wird nach allen anderen Callbacks vom Parser aufgerufen, wenn das Parsen des Dokuments zu Ende ist.

startElement Wird zu Beginn eines Elemente aufgerufen. Es werden u.a. der Name des Elementes und die vorhandenen Attribute übergeben.

endElement Wird aufgerufen, wenn der Parser ein End-Element erreicht hat. Der Name des Elementes wird u.a. übergeben.

characters Wird vom Parser aufgerufen, wenn er beim Inhalt eines Elemente auf Zeichendaten stößt. Die Zeichendaten werden als Parameter übergeben.

Es werden noch weitere an dieser Stelle nicht beschriebene Methoden in dem Interface deklariert. Im nachfolgenden Beispiel wird ein `ContentHandler` zur Verarbeitung des vorherigen XML-Beispieldokumentes implementiert. Dieser `ContentHandler` soll zu Beginn und Ende des Parsens eine Meldung auf der Standardausgabe anzeigen und während des Parsens alle gefundenen Adressen ausgeben. Wird eine Instanz der Klasse `MyContentHandler` einem XML-Parser übergeben, so werden beim Parsen vom Parser die Methoden dieses `ContentHandlers` aufgerufen. Bei Beginn des Parsens wird die Methode „`startDocument`“ aufgerufen. Trifft der Parser auf ein neues Element (bei Start-Tag des Elementes), wird die Methode „`startElement`“ aufgerufen. In dieser Methode wird im Beispiel geprüft, um welchen Elementnamen es sich handelt, und beim Element „`eintrag`“ der Wert des Attributes „`name`“ zwischengespeichert. Bei dem Element „`adresse`“ wird der Wert des Attributes „`typ`“ zwischengespeichert. Die Methode „`endElement`“ wird vom Parser aufgerufen, wenn er das Ende eines Elementes erreicht, hier findet in diesem Beispiel innerhalb der Methode keine Aktion statt. Findet der Parser Zeichendaten innerhalb eines Elementes, ruft er die Methode „`characters`“ des ContentHandler auf. In unserem Beispiel befinden sich nur im Element „`adresse`“ Zeichendaten als Inhalt. In der Methode „`characters`“ wird hier der zugehörige Name des Eintrages, zu dem die Adresse gehört, und der Typ der Adresse sowie die Adresse selbst, also die Zeichendaten, ausgegeben. Ist der Vorgang des Parsens beendet, wird vom Parser die Methode „`endDocument`“ aufgerufen.

Bsp.: Codeausschnitt eines ContentHandlers für vorheriges XML-Beispieldokument um die Liste der Adressen auszugeben.

```
class MyContentHandler implements ContentHandler {
  private String name;
  private String typ;

 public void startDocument () throws SAXException {
    System.out.println("Parsen gestartet");
```

```
  }

  public void endDocument () throws SAXException {
     System.out.println("Parsen beendet");
  }

  public void startElement (String namespaceURI, String element,
                            String rawName, Attributes atts)
              throws SAXException {
     // letzten gelesenen Namen merken
     if (element.equals("eintrag")) this.name=atts.getValue("name");
     else
     // letzten gelesenen Typ merken
     if (element.equals("adresse")) this.adresstyp=atts.getValue("typ");
  }

  public void endElement (String namespaceURI, String localName,
                          String rawName) throws SAXException {
  }

  public void characters (char[] ch, int start, int length)
              throws SAXException {
     System.out.print(this.typ+"-Adresse von "+this.name+": ");
     System.out.println(new String(ch,start,length));
  }
  ...
}
```

Weitere Interfaces

Weitere Interfaces sind `ErrorHandler`, `DTDHandler`, `EntityResolver`. Die Methoden eines `ErrorHandler`-Objektes werden aufgerufen, wenn der Parser beim Verarbeiten des XML-Dokumentes auf Fehler stößt. Man sollte einen `ErrorHandler` implementieren, wenn man auf Parserfehler reagieren möchte. Ein `DTDHandler` definiert Methoden, die der Parser beim Lesen einer DTD aufruft. Der Handler `EntityResolver` ermöglicht es zu beeinflussen, wie der Parser Referenzen auf externe Dateien behandelt.

Vorteile und Nachteile von SAX

SAX ermöglicht zwar XML-Dokumente schnell sequentiell zu parsen, jedoch ist die Verwendung von SAX nicht immer ideal. Der sequentielle Mechanismus von SAX eignet sich nicht zum wahlfreien Zugriff auf die Daten des XML-Dokumentes. Es wird nicht deutlich, in welcher Ebene man sich beim Parsen befindet, es sei denn man speichert dies aufwendig temporär ab. SAX ist also dafür von Vorteil, wenn man ein XML-Dokument einmal von Anfang bis Ende durchgehen möchte und die Daten dabei verarbeitet, ohne durch SAX selber auf vorherige Da-

ten erneut zurück zu greifen. Um die Daten im Speicher für wahlfreien Zugriff zu halten, gibt es einen anderen Ansatz, das im Folgenden beschriebene Document Object Model (DOM).

8.2.2 Document Object Model (DOM)

DOM ist ein Modell, welche die Daten eines XML-Dokumentes in einer Baumstruktur darstellt, die sich mit der zu DOM gehörenden API auslesen und modifizieren lässt [53][42]. Diese Baumstruktur ist der Syntaxbaum des XML-Dokumentes.

Um ein XML-Dokument als DOM-Baum darstellen zu lassen, benötigt man einen DOM fähigen Parser. Damit dieser DOM-Parser ein XML-Dokument als einen DOM-Baum darstellen kann, muss der DOM-Parser erst das XML-Dokument verarbeiten und dieses beim Verarbeiten als Baum im Speicher ablegen. Meist wird im DOM-Parser intern der SAX Ansatz genutzt.

DOM und die Baumstruktur des XML Dokumentes

Bei DOM wird das XML-Dokument als Syntaxbaum, entsprechend der Struktur des XML-Dokumentes, im Speicher gehalten. Als Wurzel des Baumes gibt es ein „`Document`"-Object, als Kindknoten folgt direkt das Wurzelelement (Root-Element) des XML-Dokumentes, danach die Kindelemente des Wurzelelementes, usw.

DOM definiert Interfaces, um auf die einzelnen Elemente des Baumes zugreifen zu können. Die Implementation ist in den XML-Parsern vorhanden, d.h. im Gegensatz zu SAX müssen diese Interfaces nicht implementiert werden. Ein DOM-Parser liefert dann den DOM-Baum zurück, der mit Hilfe der Interfaces genutzt werden kann. Mit den in den Interfaces deklarierten Methoden kann man dann den Baum traversieren, auf einzelne Elemente zugreifen, sie modifizieren, usw.

Das Interface `Node`, von dem alle anderen Interfaces abgeleitet sind, ist das zentrale Interface zum Arbeiten mit dem DOM-Baum. Die von `Node` abgeleiteten Interfaces entsprechen dann der jeweiligen Elementart, also Element, Attribut, Zeichendaten, usw. `Node` selber enthält sämtliche Methoden um den Baum zu traversieren, in den abgeleiteten Schnittstellen sind spezifische Methoden für die Elementart enthalten (z.B. „`name`" und „`value`" bei Attributen).

Bsp: Verwendung von DOM

```
Document doc = ... // DOM Modell vom Parser zurückgeben lassen
 Node root=doc.getDocumentElement(); // Root-Element geben lassen
 NodeList eintraege=root.getChildNodes(); // Liste der Einträge geben lassen

for (int i=0; i<eintraege.getLength(); i++) {
   Node eintrag = eintraege.item (i);
   NodeList adressen = eintrag.getChildNodes();
   for (j=0; j<adressen.getLength(); j++) {
     Node adresse = adressen.item(j);
    CharacterData adressetext = (CharacterData) adresse.getFirstChild();
      // auch reiner Text ist ein Node
```

```
      System.out.print("Adresse: "+adressetext.getData());
    }

}
```

Vorteile und Nachteile von DOM

DOM ermöglicht es übersichtlich mit den Daten eines XML-Dokumentes zu arbeiten. Will man Daten modifizieren oder mehrfach auf die gleichen Daten zugreifen und sie nicht nur sequentiell verarbeiten, so ist DOM natürlich im Vergleich zu SAX der elegantere Ansatz.

Ein wichtiger Nachteil von DOM ist, dass sich der DOM-Baum komplett im Speicher befindet. Bei im Vergleich zum Speicher großen XML-Dokumenten sollte man daher eventuell auf den DOM Ansatz verzichten.

8.3 Linkkonzepte in XML

Im Folgenden wird XLink als Konzept für das Verlinken von Ressourcen untereinander und XPath, bzw. XPointer für das Verlinken auf Ressourcen beschrieben.

8.3.1 XLink

XLink ist ein Standard für Verknüpfungen in XML [24]. Es wird ein Vokabular von Attributen genannt, mit denen man beliebige XML-Elemente unter Verwendung dieser Attribute zu einem XLink ändern kann. Ein XLink besteht dabei aus mehreren gerichteten Verweisen, so dass ein XLink eine beliebige Anzahl von Ressourcen miteinander verknüpfen kann. Die Verweise können beliebig zwischen den Ressourcen angegeben werden (n:m Beziehungen). Es kann sich bei den Ressourcen um jede Art von Daten (Dateien, Datenbankinhalte, Teilstrukturen aus XML-Dokumenten, usw.) handeln.

Verknüpfungen kennt man bereits aus HTML. In HTML gibt es zum Beispiel zwei Arten von Verknüpfungen, den „`Image`“-Tag und den „`A`“-Tag.

Das Erste ist eine Einbindung einer externen Grafik und das Zweite eine Verzweigung zu einer anderen Addresse. Während Links in HTML genau zwei Ressourcen in eine Richtung miteinander verbinden und sich die Beschreibung beim Startpunkt befindet, gibt es diese Einschränkungen bei XLink nicht.

XLink wird in XML-Dokumenten benutzt, in dem man gewünschte XML-Elemente um bestimmte Attribute erweitert, die das Element dann entweder als XLink, oder Ressource oder Verknüpfung kennzeichen.

Im nachfolgenden Beispiel wird das Element Adresse zu einem XLink erweitert. Dies geschieht durch die Verwendung des Attributes „`xlink:type`“, welches hier angibt, dass es sich um einen einfachen XLink handelt, und durch das Attribut „`xlink:href`“, welches das Ziel des einfachen XLinks angibt. Durch diese eindeutige Angabe der Attributnamen wird ein XLink definiert,

der die lokale Ressource (hier nur der reine Text `http://www.uni-dortmund.de`) mit einer entfernten Ressource (hier der Homepage mit der Adresse `http://www.uni-dortmund.de`) verknüpft.

Bsp.: Verweis in einer Adresse des XML-Beispieldokumentes auf eine Homepage

```
<adressverzeichnis>
  <eintrag name="Oliver">
    <adresse type="homepage"
     xlink:type="simple" xlink:href="http://www.uni-dortmund.de">
      http://www.uni-dortmund.de
    </adresse>
  </eintrag>
</adressverzeichnis>
```

Das nachfolgende zweite Beispiel demonstriert einen Teil der erweiterten Eigenschaften von XLink im Vergleich zu HTML-Links. Es zeigt wie man mit einem XLink einen Text (der bei HTML einen Textanker darstellen würde) sowohl mit einer Homepage, wie auch gleichzeitig mit einem Mirror verknüpfen kann. Es wird ein Ursprung als lokale Ressource (hier als Text „`Die Homepage`“), zwei entfernte Ressourcen (die eigentliche Homepage als Zieladresse und ein Mirror), und dann jeweils ein Verweis von der lokalen Ressource zu den entfernten angegeben. Es handelt sich hierbei um einen erweiterten XLink.

Bsp.: Webseite mit einer Homepage und einem Mirror

```
<website xlink:type="extended">
  <ursprung xlink:type="resource" xlink:label="src">Die Homepage</ursprung>

  <homepage xlink:type="locator"
     xlink:href="http://www.homepage.de/"
     xlink:label="dest" />

  <homepage xlink:type="locator"
     xlink:href="http://www.mirror.de/"
     xlink:label="mirror" />

  <link xlink:type="arc"
     xlink:from="src"
     xlink:to="dest" />

  <link xlink:type="arc"
     xlink:from="src"
     xlink:to="mirror" />

</website>
```

8.3.2 XPath und XPointer

XPath [22] und XPointer [23] dienen zur Adressierung innerhalb von XML-Dokumenten. XPath ist dazu gedacht ein Dokument nicht als Ganzes zu adressieren (wie es bei URI's der Fall ist), sondern auf bestimmte Teile eines Dokumentes (z.B. auf den 5. Eintrag, auf eine bestimmte Adresse, usw.) zu verweisen. Ein XPath-Ausdruck kann auch Funktionen um zu filtern, Bereiche auszuwählen, numerische Daten zu addieren, usw., Klammerungen und logische / arithmetische Operatoren beinhalten. Ausdrücke können einen boolschen Wert, eine Knotenmenge, eine Fließkommazahl, eine Zeichenkette oder mehrere Abschnitte als Ergebnis liefern.

Zur Veranschaulichung, wie ein XPath aussieht, folgt ein Beispiel, in dem der Ausdruck in allen „`table`"-Elementen des XML-Dokumentes die „`width`"-Attribute auswählt. Hier lässt sich direkt sehen, wie komplex ein solcher Ausdruck selbst bei einer einfach klingenden Auswahl ist.

Bsp.: Ein komplexer XPath

```
/descendant-or-self::node()/child::table/attribute::width
```

XPointer ist eine auf XPath aufbauende Sprache, die in URIs verwendet werden kann, um Teile eines XML-Dokumentes zu adressieren. XPointer können z.B. bei XLinks im „`xlink:href`" Attribut genutzt werden. XPointer erweitert XPath darin, auf einzelne Punkte in einer XML-Ressource (z.B. auf ein bestimmtes Element oder auf eine bestimmte Position in der Datei) und auf Bereiche (Abschnitte zwischen zwei Punkten) verweisen zu können. Dazu gibt es in der XPointer Spezifikation zusätzliche Funktionen.

Im folgenden Beispiel werden drei Arten gezeigt, wie man eine Verknüpfung mit XPointer realisieren kann. Das erste Beispiel (siehe erstes „`person`"-Element) ist eine Möglichkeit ein Element über die Indexzahlen zu referenzieren. Im zweiten Fall (siehe zweites „`person`"-Element) geht es direkt über die ID („`#`" steht als URI dafür, das sich die Verknüpfung im gleichen Dokument befindet). Dies ist die Kurzform der Schreibweise, wie sie im dritten Beispiel (siehe drittes „`person`"-Element) benutzt wird. Von den beiden genannten Kurzformen abgesehen, wird normalerweise wird immer „`xpointer(ausdruck)`" geschrieben, wobei es sich bei dem Ausdruck dann um die XPointer Adresse handelt, also um einen XPath mit den gültigen Erweiterungen.

Bsp: Verknüpfung von Einträgen mit XPointer

```
<adressverzeichnis>
  <eintrag name="Peter" id="101">
  </eintrag>

 <eintrag name="Anna" id="123">
    <person xlink:type="simple" xlink:href="#/1/1">
      Vater
    </person>
```

```
    <person xlink:type="simple" xlink:href="#101">
       Vater
     </person>

    <person xlink:type="simple" xlink:href="#xpointer(//id("101"))">
       Vater
     </person>
   </eintrag>
</adressverzeichnis>
```

Kapitel 9

HyTime

Autor: *Daniel Moelle*

Überblick

Der vorliegende Artikel soll die auf SGML aufbauende Dokumentstrukturierungssprache HyTime vorstellen. Dazu werden im ersten Abschnitt zunächst eine Motivation der Strukturierung von Dokumenten und ein kleiner Einblick in SGML geliefert. Die Konzepte von HyTime sind Gegenstand des zweiten Abschnitts, in dem mit den Linkklassen und den *finite coordinate spaces* die beiden mächtigsten Sprachmittel behandelt werden, die HyTime zur Verfügung stellt. Im Rahmen der Vorstellung der Linkklassen wird dabei auch auf die verschiedenen Adressierungsarten und Traversierungsoptionen eingegangen, die bei Verweispunkten, also Ankern, eingesetzt werden können. Der dritte Abschnitt schließlich dient der Diskussion einiger Probleme, die sich beim Einsatz von HyTime in der Praxis ergeben können.

9.1 Grundlagen: Strukturierungssprachen

Die Notwendigkeit einer Strukturierung von elektronisch gespeicherten Dokumenten, z.B. aus Texten und Bildern zusammengesetzten Schriftstücken, ergibt sich unmittelbar aus den Anforderungen, die bereits im täglichen Umgang mit Informationen auftreten. Eines der wichtigsten Kriterien ist zweifelsohne die Möglichkeit des Austauschs zwischen verschiedenen Personen; Dokumente wären als Informationsträger wertlos, wenn sie von der Zielgruppe nicht interpretiert werden könnten. Wenn ein Dokument jedoch so strukturiert wird, dass die verschiedenen Bausteine, wie z.B. Überschriften, Absätze oder Verweise auf Abbildungen, auch als solche ausgezeichnet werden, wird eine korrekte Interpretation möglich.

In ähnlicher Weise müssen die einzelnen Bausteine eines Dokuments erkenn- und deutbar gemacht werden, damit eine elektronische Verarbeitung möglich wird. Die Ableitung eines automatisch generierten Inhaltsverzeichnisses aus den Kapitelüberschriften z.B. erfordert natürlich eine Markierung von Kapiteln als solche. Andererseits ist die Auszeichnung von Bausteinen nicht zuletzt für die Formatierung und für die Einbettung in Archivierungssysteme

von Belang. Wenn beispielsweise Texte archiviert werden, in denen Angaben zu Autor, Titel und Erscheinungsjahr entsprechend ausgezeichnet sind, können spezifische Suchfunktionen angeboten werden.

Diese Strukturierung und Interpretierbarkeit wird zumeist durch die Verwendung spezieller, anwendungs- und herstellerspezifischer Dateiformate sichergestellt; bei Text- und Bildverarbeitungssystemen wird dies auf extreme Weise deutlich. Offensichtlich verletzen solche Formate aber die Forderung, dass das Dokument auch für andere Personen lesbar sein soll (Austausch, s.o.). Spätestens an diesem Punkt wird deutlich, welche enormen Vorteile eine verallgemeinerte Strukturierungssprache mit sich bringt.

Eine solche verallgemeinerte Strukturierungssprache ist SGML (*standard generalized markup language*), die auch als Basis für HyTime dient (s. 9.2). Die allgemeine Anwendbarkeit von SGML wird durch das DTD-Konzept gesichert: jede DTD (*document type definition*) beschreibt eine bestimmte Art von Dokumenten, indem sie die verschiedenen Bausteine, die in dem entsprechenden Dokumententyp auftreten können, auflistet. Dabei werden auch die Möglichkeiten zur Verschachtelung von Dokumentbausteinen vorgegeben. Eine DTD gibt also die Grammatik einer Sprache vor, in der Dokumente des zugehörigen Typs beschrieben werden können. Insofern kann SGML als eine Meta-Sprache (vgl. [66]) aufgefasst werden: es handelt sich um eine Sprache, die die Definition spezieller Sprachen ermöglicht.

Das populärste Beispiel einer mit SGML konstruierten Sprache (auch als „SGML-Anwendung“ bezeichnet) ist sicherlich HTML. Zur Veranschaulichung soll hier nur eine simple DTD zur Beschreibung von Telefonbüchern vorgestellt werden; als ausführliche und brauchbare Einführung sei „A Gentle Introduction to SGML“ ([77]) empfohlen.

```
<!DOCTYPE telefonbuch [
<!ELEMENT telefonbuch - - (kopf, eintrag+) >
<!ATTLIST telefonbuch typ (privat | beruflich) (privat) >
<!ELEMENT kopf - - (besitzer, stand) >
<!ELEMENT (besitzer | stand) - o (#PCDATA) >
<!ELEMENT eintrag - - (name, nummer+) >
<!ELEMENT (name | nummer) - o (#PCDATA) >
<!ATTLIST nummer anschluss (fon | mobil | fax) (fon) >
<!ENTITY do "++49-231-" >
] -- Dokumententyp zur Darstellung von Telefonbuechern -->
```

Die DTD wird mit dem Schlüsselwort `!DOCTYPE` eingeleitet, wobei der Name „telefonbuch“ vereinbart wird. In den eckigen Klammern folgt eine Auflistung der Dokumentenbausteine. Jeder Baustein wird über einen `!ELEMENT`-Ausdruck definiert, in dem zunächst ein Bezeichner festgelegt wird. Die beiden Minuszeichen deuten an, dass die einleitenden bzw. abschließenden *tags*, die genau wie in HTML zu verwenden sind, angegeben werden müssen; ein kleines „o“ hingegen kennzeichnet optionale *tags* („omit“). Abschließend werden in runden Klammern die möglichen Inhalte des Elements festgelegt. Dies können wieder Elemente (z.B. `kopf`), Listen von Elementen (z.B. `eintrag+`) oder Zeichenketten (z.B. `#PCDATA`: *parsed character data*, s.u.) sein.

Weiterhin können Elementen durch `!ATTLIST`-Ausdrücke mögliche Attribute zugeordnet wer-

den. Dabei werden neben dem Bezeichner des zugehörigen Elements und dem Namen des Attributs auch die verschiedenen Attributwerte angegeben. So kann ein Telefonbuch über das Attribut `typ` als privat (Vorgabewert) oder beruflich gekennzeichnet werden, während sich verschiedene Anschlusstypen über das Attribut `anschluss` markieren lassen. Entitäten schließlich können als Textmakros aufgefasst werden: innerhalb eines `#PCDATA`-Blocks würde ein SGML-Parser die Zeichenkette „&do;“ (Entität `do`) durch den zugewiesenen Text ersetzen.

Eine konkrete Telefonbuch-Instanz könnte demnach wie folgt aussehen:

```
<!DOCTYPE telefonbuch>
<telefonbuch typ="beruflich">
<kopf><besitzer>PG 415<stand>01.01.2002</kopf>
<eintrag>
  <name>PG-Pool
  <nummer anschluss="fon">&do;755-????
  <nummer anschluss="fax">&do;755-????
</eintrag>
</telefonbuch>
```

Über das `!DOCTYPE`-Element wird das Dokument der entsprechenden DTD zugeordnet. Die Verschachtelung der Elemente ist offensichtlich mit den in der DTD formulierten Vorgaben vereinbar. Bei den Angaben zu Besitzer, Stand, Name und Nummern fehlen die abschließenden *tags*, was aber ebenfalls syntaktisch korrekt ist.

Die Vorgabe einer Syntax ist für die automatische Verarbeitung von besonderer Wichtigkeit. Aufgrund der allgemeinen Anwendbarkeit von SGML ist es sogar möglich, Parser zu konstruieren, die die Wohlgeformtheit von Dokumenten gemäss einer beliebigen DTD, also unabhängig vom tatsächlichen Kontext, überprüfen. Die Interpretation hingegen ist selbstverständlich anwendungsspezifisch.

Aufgrund der Verschachtelung der verschiedenen Elemente können DTDs baumartig gelesen werden, was aus naheliegenden Gründen auch für die Verarbeitung nützlich ist. Dementsprechend ist die Ausgabe eines SGML-Parsers i.A. einfach der Syntaxbaum des Eingabedokuments.

9.2 HyTime

Bemerkenswert ist die Tatsache, dass HyTime, die *hypermedia/time-based document structuring language*, während der Entwicklung der Musikbeschreibungssprache SMDL (*standard music description language*) gewissermaßen als Nebenprodukt angefallen ist. Für die Beschreibung von Musikstücken ist reines SGML vor allem deshalb wenig geeignet, weil zeitabhängige Inhalte in SGML schlichtweg nicht auszudrücken sind; SGML dient in erster Linie der Beschreibung statischer Dokumente. Andererseits ist leicht einzusehen, dass die Zeit auch in ganz anderen Zusammenhängen eine wesentliche Rolle spielen kann.

HyTime ist eine SGML-Erweiterung in dem Sinne, dass es als Meta-DTD neue Konzepte zur Verfügung stellt, die dann in konkreten DTDs verwendet werden können. Dabei bietet HyTi-

me aber nicht nur Sprachmittel, die die Berücksichtigung der Zeit als Komponente gestatten. Tatsächlich erlaubt das FCS-Modul (*finite coordinate space*, s. 9.2.2) die Modellierung beinahe beliebig komplexer Vorgänge in Raum und Zeit. Darüberhinaus wird das eher monotone Linkkonzept von SGML nachhaltig ausgeweitet.

Die verschiedenen HyTime-Module definieren hauptsächlich Prototypen von besonderen Elementen, aus denen im Rahmen einer DTD konkrete Elemente abgeleitet werden können, denen die gewünschten Eigenschaften des Prototyps zueigen sind. Beispielsweise stellt HyTime Linkklassen zur Verfügung, aus denen sich Verweiselemente konstruieren lassen, denen somit schon eine besondere Bedeutung inneliegt. Reine SGML-Elemente hingegen sind bis zur Interpretation durch ein entsprechendes Programm weitestgehend bedeutungsfrei.

Angesichts der Tragweite der von HyTime realisierten Erweiterungen ist klar, dass ein reiner SGML-Parser nicht mehr dazu in der Lage sein kann, DTDs und zugehörige Dokumente zu verarbeiten, in denen HyTime-Module zum Einsatz kommen. Programme, die mit derartigen Dokumenten umgehen können sollen, müssen also dementsprechend erweitert werden.

9.2.1 Linkklassen

In diesem Abschnitt werden die verschiedenen mit den Linkklassen von HyTime zur Verfügung stehenden Verweistypen vorgestellt. Allerdings erwächst die hohe Flexibilität von Verknüpfungen in HyTime nicht nur aus den Linkklassen, sondern auch aus einer Vielzahl weiterer Sprachmittel, die zum Teil in SGML nicht einmal ansatzweise vorkommen. Deshalb werden im Folgenden mit den Adressierungsarten und den Traversierungsoptionen auch zwei dieser ungewöhnlichen Konzepte diskutiert.

Adressierung

HyTime bietet ein wesentlich differenzierteres System zur Lokalisierung von Ankern, also den Ausgangs- oder Zielpunkten von Verweisen, als SGML. In Verbindung mit diesen Adressierungsmöglichkeiten werden die HyTime-Linkklassen zu einem besonders nützlichen Instrument (vgl. [57]).

- **Addressing by Name**
  SGML-Elemente können, auch wenn sie in anderen Dokumenten liegen, über eindeutige Bezeichner adressiert werden. Dies ähnelt der Vorgehensweise in reinem SGML, obwohl dort die Referenzierung externer Elemente problematisch ist. Der folgende *clink* (s. Abschnitt 9.2.1) verweist auf das SGML-Element mit dem Bezeichner „abs2“:

  ```
  <p>Im <clink linkend=abs2>folgenden Absatz</clink>
  wird dies detailliert beschrieben.</p>
  ...
  <p id=abs2>Es folgt eine Beschreibung...</p>
  ```

- **Addressing by Position**
  Bestandteile eines Dokuments, für die es möglich ist, eine Position innerhalb einer bestimmten Umgebung anzugeben, können auf genau diese Weise lokalisiert werden. Unter

anderem trifft dies auf Knoten im SGML-Syntaxbaum genauso zu wie auf Teilstrings in Zeichenketten:

```
<p id=pa>Die Zeichenkette</p>
<dataloc id=dl locsrc=pa quantum=str><dimlist>5 7</dataloc>
<clink linkend=dl>Das Wort "Zeichen"</clink>
```

Der *dataloc-tag* stellt eine Adressangabe dar. Er selbst trägt den Bezeichner „dl“, und als Suchraum wird das Element mit dem Namen „pa“ vereinbart. Über das Attribut *quantum* wird festgelegt, auf welche Weise das folgende Zahlentupel eine Position beschreibt; im Falle von *str* (für String) werden genau zwei Zahlen verlangt, die die Startposition bzw. die Länge des Teilstrings angeben. Im obigen Beispiel wird also der Teilstring „Zeichen“ adressiert.

- **Addressing by Semantic Construct**
  In einer einfachen Variante dieser Adressierungsart kann über ein Attribut vorgegangen werden, dessen Wert ermittelt werden soll. Die Suche nach dem Attribut unterliegt dabei dem Parser. Im folgenden Beispiel ist im Element mit dem Bezeichner „dl“ (s.o.) nach der Belegung des Attributs *quantum* zu suchen:

  ```
  <proploc locsrc=dl>attval[quantum]</proploc>
  ```

  Andererseits, und wesentlich allgemeiner, kann die Ermittlung des Verweisziels einer externen Anwendung überlassen werden (z.B. die Suche nach dem zweiten Tempowechsel in einem Musikstück).

Independent links

Die Bezeichnung *independent* ist darauf zurückzuführen, dass bei Links dieser Art, die auch kurz *ilinks* genannt werden, eine vollständige Trennung zwischen der Stelle des Verweises (dem *link markup*) und den Angaben der Verweisziele möglich ist. Insbesondere lassen sich somit Verweisziele in externen Dateien (sog. *hub documents*) zentral verwalten.

Das folgende Beispiel zeigt einen *ilink* mit zwei Verweiszielen. Die Positionen der beiden Zielanker werden dabei in separaten *address-tags* beschrieben, die jeweils angeben, in welchem Dokument und unter welcher Bezeichnung die zugehörigen Elemente zu finden sind.

```
<ilink linkends="bauerngabel1 bauerngabel2">
Es gibt zahlreiche Beispiele fuer Figurenverluste durch Bauerngabeln
in der Anfangsphase.
</ilink>
...
<address ID="bauerngabel1" DOC=wien-1932 LOCAL=halicz-lanz>
<address ID="bauerngabel2" DOC=stockholm-1937 LOCAL=ozols-reid>
```

Desweiteren können *ilinks* mit allen Adressierungsarten verwendet werden. Im obigen Beispiel könnte die Suche nach passenden Partien also auch einem geeigneten Schachprogramm überlassen werden (Adressierung über ein semantisches Konstrukt).

Es gibt zwei besondere Subtypen von *ilinks*, nämlich *contextual* (*clinks*) und *property links* (*plinks*). Beide unterliegen der Einschränkung, dass sie stets zwei SGML-Elemente miteinander verbinden. Bei Verknüpfungen vom Typ *clink* ist eines dieser beiden Elemente der Verweis selbst, der somit implizit auch als Anker aufgefasst wird. Tatsächlich beschreibt ein *contextual link* also genau die Form von wechselseitiger Abhängigkeit, die der Begriff *cross reference* suggeriert (vgl. [48]).

```
<p>
Fussnoten sind meist nur dann angebracht, wenn eine Einbettung in den
Fliesstext einer Unterbrechung gleichkaeme<clink linkend=fn1>[1]</clink>.
</p>
...
<footnote ID=fn1>Dies wird oft zu einer reinen Ermessensfrage.</footnote>
```

Property links hingegen sind ein rein syntaktisches Hilfsmittel und verbinden ein SGML-Element mit einer Eigenschaft, also einem Paar aus Attributname und -wert. Auf diese Weise können also die Eigenschaften von Elementen, die in externen Dokumenten auftauchen, ohne Eingriff in diese Dokumente angepasst werden. Beispielsweise könnten in einem Artikel, der nur mit Lesezugriff abgerufen werden kann, Abschnitte farblich hervorgehoben werden, die *im lokalen Kontext* wichtig sind.

Leider gilt für *plinks* wie für die im Folgenden beschriebenen *constructed location links*, dass sie in der dem Autor zugänglichen Literatur nur abstrakt behandelt werden. Aus diesem Grund können an dieser Stelle keine Beispiele zur Vorstellung der Syntax geliefert werden.

Constructed location links

In HyTime dienen Verknüpfungen nicht zwangsweise als Hyperlinks, die es dem Betrachter des Dokuments gestatten, durch die verschiedenen Komponenten zu navigieren (Hyperdokument-Traversierung). Bereits die oben erwähnten *property links* sind ein Beispiel für eine „interne“ Verwendung von Verweisen. Tatsächlich existieren mit den *aggregate location* (*agglinks*) und *span links* zwei rein konstruktive Verknüpfungstypen, die ausschließlich für die Komposition von Dokumenten relevant sind.

Agglinks dienen der Zusammenfassung verschiedener Objekte zu einer neuen Entität, ähnlich der Aggregation in UML. Da HyTime wie SGML auch die Einbindung fremdformatiger Objekte (*notation data*) gestattet, sofern ein sie interpretierendes Programm deklariert wird, ist die Zahl der möglichen Anwendungen nahezu unbegrenzt. Insbesondere können, ein geeignetes externes Programm vorausgesetzt, auch nicht-uniforme Komponenten verbunden werden, z.B. ein Text, eine Animation und ein Musikstück zu einer Präsentation.

Span links schließlich bieten die Möglichkeit, Auszüge aus anderen SGML-Dokumenten einzubinden, die eventuell aus mehreren SGML-Elementen bestehen, also SGML-*markup* enthalten.

Ein derart importierter Ausschnitt wird vom Parser nicht durchlaufen; es ist somit also auf einfache Weise möglich, in einem SGML-Dokument über SGML zu schreiben.

Traversierungsoptionen

Das Verfolgen eines Verweises zu einem Anker, also einem Quell- oder Zielpunkt des Verweises, wird als Traversierung des Ankers bezeichnet. Ein naheliegendes Beispiel ist die Navigation auf einer HTML-Seite, bei der ein Betrachter von einem Inhaltsverzeichnis zu einem entsprechend verankerten Sinnabschnitt springt. HyTime hebt sich auch dadurch deutlich von SGML ab, dass die möglichen Arten der Traversierung eines Ankers spezifiziert werden können (vgl. [48]):

- **E: External arrival**
  Nach Ankunft am Anker können auch alle anderen Anker des zugehörigen Verweises traversiert werden.
- **I: Internal arrival**
  Wie **E**, allerdings muss der Anker über den zugehörigen Link angesprungen worden sein.
- **R: Return**
  Nachdem der Anker angesprungen wurde, kann nur der Quellanker des zugehörigen Verweises traversiert werden.
- **D: Departure**
  Nachdem der Anker angesprungen wurde, darf der Kontext des Ankers (und somit auch der zugehörige Verweis selbst) verlassen werden.
- **N: No further**
  Nachdem der Anker angesprungen wurde, gibt es keine weiteren Traversierungsmöglichkeiten.
- **P: Prohibited**
  Der Anker darf von allen anderen Ankern des zugehörigen Verweises aus nicht angesprungen werden.

Mit „Ankunft“ wird dabei das Erreichen des Ankers auch ohne die Verwendung eines Verweises bezeichnet, wie z.B. durch „Scrollen“. Analog gilt dies für „Verlassen“. Die Optionen sind kombinierbar. Insbesondere die Regeln **I** und **R** verlangen offenbar, dass ein Programm zur Darstellung von HyTime-Dokumenten die vom Benutzer verfolgten Verweise speichern muss, um über die Anwendbarkeit der Regeln entscheiden zu können.

9.2.2 *Finite coordinate spaces*

Während die bisher behandelten Aspekte und Beispiele lediglich darauf hindeuten, dass HyTime eine mächtige Sprache zur Beschreibung von Hypertexten darstellt, erlaubt das FCS-Modul den Übergang zu echten Hypermedia-Dokumenten. Diese Behauptung ist so zu verstehen, dass neben den üblichen Texten und Grafiken auch weitere Medien berücksichtigt werden können, die räumliche oder zeitliche Ausdehnung haben (vgl. [57]).

Ein FCS (*finite coordinate space*) ist zunächst nicht mehr als ein endlicher Raum, dessen Dimension benutzerdefinierbar ist. Die Achsen werden mit frei wählbaren Referenzeinheiten versehen. Im Normalfall sind dies standardisierte Längen- und Zeiteinheiten; entsprechend der allgemeinen Konzeption sind aber auch völlig andere Einteilungen denkbar.

Objekte und Ereignisse

Die Inhalte eines FCS sind natürlich wieder Objekte. Da HyTime die Einbettung beliebiger Daten gestattet, können dies z.B. in externen Dateien gespeicherte Bilder, Klänge oder Filmsequenzen, aber auch in SGML-Elementen beschriebene Zeichenketten oder Polygone sein. Allerdings werden diese Objekte nicht direkt eingefügt, sondern in Ereignisse eingebettet, die einen *event schedule* (Ablaufplan) bilden. Neben dem Objekt enthält ein Ereignis Informationen darüber, in welchem Teil des FCS es eintritt. In einem FCS, der einen Ausschnitt der Raumzeit modelliert, könnte ein Ereignis beispielsweise wie folgt charakterisiert werden: zwischen den Punkten 5 und 17 auf der Zeitachse soll an der Position $(0, 2, 0)$ des Raums das Objekt A dargestellt werden.

Der folgende, beispielhafte Ausschnitt aus einer DTD zeigt die Definition eines FCS, mit dessen Hilfe der Ablauf einer Tagung beschrieben werden kann.

```
<!ELEMENT tagungfcs - - (evsched+) >
<!ATTLIST tagungfcs
   HyTime    NAME   #FIXED "fcs"
   id        ID     #IMPLIED
   axisdefs  NAMES  #FIXED "zeit" >
```

Das zugehörige SGML-Element `tagungfcs` soll also stets mindestens einen Ablaufplan (*event schedule*) beinhalten. Durch die Attributliste wird festgelegt, dass es sich bei diesem Element um einen FCS handelt, der optional mit einem Bezeichner (`id`) versehen werden kann und der genau eine Achse hat, der dabei der feste Name „zeit“ zugewiesen wird.

Zusätzlich muss auch diese Zeitachse näher definiert werden:

```
<!ELEMENT zeit - o EMPTY >
<!ATTLIST zeit
   HyTime    NAME   #FIXED "axis"
   axismeas  CDATA  #FIXED "SIMINUTE"
   axisdim   CDATA  #FIXED "180" >
```

Das SGML-Element `zeit` dient lediglich der Beschreibung der Achse; es wird daher in konkreten Dokumenten nicht mehr verwendet und kann somit als leer deklariert werden. Die Attributliste gibt an, dass das Element eine Achse definiert, die sich über 180 Minuten erstreckt.

Eine Tagung kann nun, unter Einsatz dieser DTD, wie folgt beschrieben werden:

```
<tagungfcs>
  <evsched id=tagung2002>
    <event data=vortragA   exspec=vt_a>
    <event data=vortragB   exspec=vt_b>
    <event data=diskussion exspec=disk>
  </evsched>
</tagungfcs>
<extlist id=vt_a><dimspec>  1  40</dimspec></extlist>
<extlist id=vt_b><dimspec> 61 100</dimspec></extlist>
<extlist id=disk><dimspec>121 180</dimspec></extlist>
```

Dieser FCS enthält also einen Ablaufplan mit der Bezeichnung `tagung2002`, der drei Ereignisse umfasst. Die Bezeichner der zugehörigen Objekte, z.B. Foliensätze für die Vorträge oder eine Folie mit Diskussionsthesen, werden über das Attribut `data` angegeben. Die Ausdehnung der einzelnen Ereignisse wird in separaten Listen mit Koordinatenangaben (*extension lists*) festgelegt. Im obigen Beispiel erstreckt sich also der erste Vortrag von der ersten bis zur vierzigsten Minute auf der Zeitachse des FCS.

Objektmodifikation

Die Verwendung von *event schedules* erlaubt zwar bereits die Modellierung beliebiger endlicher Vorgänge, ist aber in bestimmten Fällen zu aufwendig. Wenn ein Objekt in dem Zeitraum, in dem es sichtbar ist, eine fliessende Bewegung im Raum durchführen soll, müsste diese Animation aus einer immensen Zahl von Ereignissen zusammengesetzt werden, die das Objekt zu den gegebenen Zeitpunkten an den entsprechenden Positionen einblenden. Offenbar ist es viel naheliegender, einen Objektmodifikator zu verwenden, der die Verschiebung des Objekts bewerkstelligt.

Analog zu der Einbettung von Objekten in Ereignisse werden solche Modifikatoren in *modscopes* untergebracht. Ein *modscope* muss dabei die Information enthalten, welchen Ausschnitt und welche Objekte des FCS der enthaltene Modifikator beeinflusst.

Projektion

Eine weitere Besonderheit des FCS-Konzepts ist die Aufteilung in einen Quell- und einen Zielraum. Die im Quellraum im Rahmen von Ereignissen auftretenden und durch Objektmodifikatoren manipulierten Objekte werden durch sogenannte Projektoren in den für den Betrachter des Dokuments wahrnehmbaren Zielraum abgebildet (*rendering*).

Das vorgestellte System wird selbstverständlich erst dadurch brauchbar, dass die auftretenden Objekte beliebiger Natur sein können. Die Darstellung des Zielraums bleibt allein der Anwendung überlassen und ist deshalb wieder kontextabhängig. Zu einer einzelnen DTD könnten also durchaus völlig verschiedene Anwendungen erstellt werden.

Das bisher verwendete, naheliegende Beispiel einer 3D-Animation mag über die Vielseitigkeit des FCS-Moduls hinwegtäuschen. Aus diesem Grund soll dieser Abschnitt mit einem gänzlich anderen Beispiel abgeschlossen werden.

Ein Musikstück lässt sich in begrenztem Masse als ein FCS auffassen, der eine Zeitachse sowie für jede Stimme eine eigene Achse enthält, die nach Noten (oder Frequenzen) skaliert ist. Bei dieser Modellierung ist das Anspielen einer Note als Ereignis aufzufassen. Aufgrund der Achsenwahl könnte ein Ereignis aber auch das gleichzeitige Anspielen mehrerer Noten durch verschiedene Stimmen beschreiben. Als Objektmodifikatoren kommen in diesem Zusammenhang alle Manipulationen von Tönen in Frage, z.B. die Forderung, dass eine Note leiser als üblich gespielt wird.

An dieser Stelle könnte ein Projektor eingesetzt werden, um für jede Stimme des Stücks das zugehörige Instrument festzulegen. Die Abbildung (*rendition*, Wiedergabe) des Musikstücks würde also eine detaillierte Abspielvorschrift liefern, die dann von einem entsprechenden externen Programm angezeigt oder in echte Klänge umgewandelt werden könnte. Andererseits wäre es alleine durch Änderungen an den Projektoren möglich, das Stück zu transponieren, andere Instrumente einzusetzen oder das Tempo zu ändern.

Der hohe Abstraktionsgrad, mit dem die Objektmodifikation und Projektion bei HyTime betrachtet werden, findet sich auch in der zugehörigen Literatur wieder. Aus diesem Grund kann der Autor auch an dieser Stelle keine Beispiele (in Syntax) liefern.

9.3 HyTime in der Praxis

Die enorme allgemeine Anwendbarkeit von HyTime spricht zunächst für einen ausgedehnten praktischen Einsatz, genau wie es für SGML der Fall ist. Insbesondere ist die Möglichkeit des Austauschs von Dokumenten mit anderen Personen (s. 9.1) nicht zuletzt aufgrund der zunehmenden Vernetzung ein wesentliches Argument.

Andererseits ist es gerade der Umstand, dass HyTime sehr allgemein gehalten ist, der eine Implementierung zu einer relativ schwierigen Aufgabe macht. Angesichts des wachsenden Zeitdrucks, mit dem sich vor allem kommerzielle Software-Entwickler konfrontiert sehen, ist durchaus nachzuvollziehen, dass auch neue Programme eher mit speziellen Formaten arbeiten. Weiterhin stellte sich schon bei SGML das Problem, dass es auch für häufig verwendete Dokumenttypen wie z.B. Schriftstücke keine normierten Vorgaben oder Elementbausätze gibt (vgl. [18]).

Ein ganz konkretes Problem ergibt sich aus der Tatsache, dass Dokumente in durch HyTime-DTDs beschriebenen Sprachen oft zum grossen Teil aus *markup* bestehen müssen, um die Interpretierbarkeit durch die zugehörigen Anwendungen sicherstellen zu können. Die unabdingbare Eindeutigkeit der Semantik erfordert also unter Umständen einen deutlichen *blow-up* der gespeicherten Informationen, der bei zeitsensitiven Anwendungen nicht mehr vertretbar ist, weil die vom Parser gelieferten Syntaxbäume schlichtweg zu gross werden (vgl. [15]).

Kapitel 10

Das Positionskonzept von DoDL

Autor: *Rafael Hosenberg*

10.1 Einleitung

Diese Arbeit gibt einen Einblick in das Positionskonzept von DoDL. Auf die Sprache selbst wird in Kapitel 11 näher eingegangen. DoDL befasst sich mit der Modellierung von Hypermediadokumenten. Das Positionskonzept ist ein Teil des Beschreibungs-Modells von DoDL. Im Dexter-Modell wird die Position auch als *Anchor* bezeichnet, siehe [75]. Medienobjekte, auch *node* genannt, siehe [75] Kapitel 3, spielen in DoDL eine wichtige Rolle. Ausgehend von den Medienobjekten wird es möglich Strukturen zu schaffen, durch die eine Positionsbestimmung möglich wird. Mit Hilfe der Positionen, Links und bestimmten Eigenschaften dieser, die im weiteren noch erklärt werden, wird es möglich, *Hyperdokumente* formal zu erfassen.

10.2 Medienobjekte

Medienobjekte beschreiben eine „beliebige, digital speicherbare Informationseinheit“, siehe [34] Seite 15. Diese sind modular aufgebaut und können aus *atomaren* Komponenten oder aus *zusammengesetzten Komponenten* bestehen. Ein *Atom* stellt die kleinste Einheit von zusammengesetzten Medienobjekten dar. Um Positionen erfassen zu können, wird im folgenden eine einfache Strukturierung von Medienobjekten vorgestellt. Durch diese Strukturierung wird es möglich, Medienobjekte anhand von Positionen und Links miteinander in hypermediale Beziehung zu setzen, also zu verlinken.

10.2.1 Zusammengesetzte Medienobjekte und ihre Komponenten

Zusammengesetzte Medienobjekte können aus Atomen oder aus zusammengesetzten Medienobjekten mit mehr als einem Bestandteil bestehen. Ein Teil eines Medienobjektes wird als *Submedienobjekt* bezeichnet. Medienobjekte fassen wir in einer Menge, dem *Universum* al-

ler Medienobjekte, zusammen. Durch Dekomposition kann man ein Medienobjekt in seine Bestandteile zerlegen. Diese Bestandteile werden *Komponenten* genannt.

Die Komponenten eines Medienobjektes sind rekursiv definiert. Falls das Medienobjekt atomar ist, dann besteht die Menge nur aus einem Element, nämlich dem Medienobjekt selbst. Sonst bestehen die Komponenten des Medienobjektes aus den Bestandteilen des Medienobjektes und, rekursiv aus den Komponenten dieser Bestandteile. Was die Begriffe im einzelnen bedeuten wird im folgenden erläutert.

Unter einem Medienobjekt kann man sich verschiedene Dinge vorstellen. Ein Medienobjekt kann zum Beispiel ein ganzes Cockpit eines Autos sein, oder die Darstellung eines Radios, oder eine einfache Anzeige wie die Geschwindigkeitsanzeige. Im weiteren wird auf das Beispiel eines Auto-Cockpits näher eingegangen. Es können verschiedene Medienobjekttypen existieren, die man in beliebiger Reihenfolge kombinieren kann. Ein Cockpit umfasst zum Beispiel nicht nur informative Elemente, sondern lässt auch Aktionen zu, die die Steuerung eines Autos betreffen. Die Geschwindigkeitsanzeige besteht zum Beispiel aus einem LCD-Feld, welches auch wieder aus kleineren Medienobjekt-Bestandteilen zusammengesetzt wird, den einzelnen LCD-Elementen, die als atomare Medienobjekte verwaltet werden können. Ein Radio kombiniert informative Elemente und Bedienelemente. Genauso kann man Videos oder Grafiken als Atome darstellen, wenn ihre Struktur nicht von Bedeutung ist. Die Präsentation der einzelnen Medienobjekte wird als *Layout* bezeichnet. Das Layout, oder in bestimmten Fällen auch zum Beispiel die grafische Darstellung von Medienobjekten, ist unabhängig von der Position von Medienobjekten. Darum wird dies nicht weiter betrachtet.

10.2.2 Struktur von Medienobjekten

Zusammengesetzte Medienobjekte werden also baumartig angeordnet. Innere Knoten repräsentieren zusammengesetzte Medienobjekte. Die Atome entsprechen den Blättern in der Baumstruktur. Die Kanten im Graphen verweisen auf Submedienobjekte. Submedienobjekte können auch wiederholt vorkommen. Ein Beispiel einer einfachen Baumstruktur ist in Abbildung 10.1 zu sehen. Das `Cockpit` ist ein Medienobjekt, welches aus den Submedienobjekten `Licht`, `km/h-Anzeige` und `Radio` besteht.

10.3 Gleichheit von Medienobjekten

Die Anordnung der atomaren Bestandteile eines Medienobjektes kann genau wie seine Baumstruktur zur Unterscheidung von Medienobjekten herangezogen werden. Medienobjekte können sich also unterscheiden, obwohl ihre Bestandteile gleich sind.

10.3.1 Saum-Äquivalenz

Zwei Medienobjekte sind *saum-äquivalent*, wenn die Inhalte ihrer Blätter von links nach rechts gelesen gleich ist. Wir vergleichen das Medienobjekt `Cockpit` (Abbildung 10.1) mit dem Medienobjekt `Cockpit2`, siehe Abbildung 10.2. Anhand der Darstellungen wird deutlich, dass

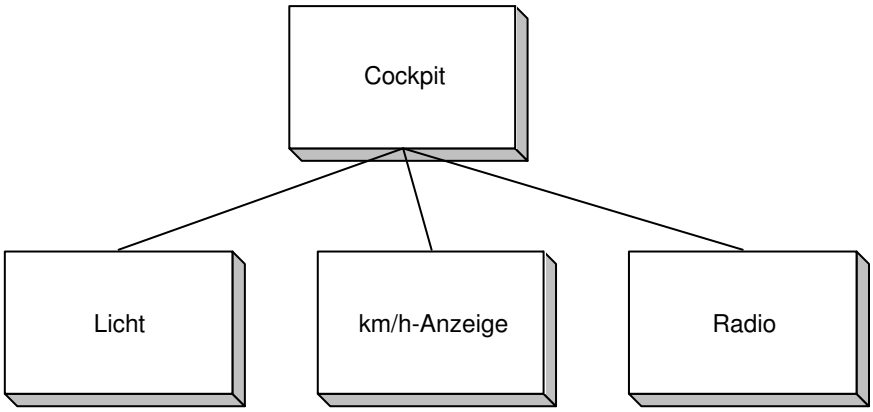

Abbildung 10.1: Baumdarstellung des Medienobjektes Cockpit

diese verschiedene *Baumstrukturen* besitzen. Ihr *Saum* ist allerdings gleich. Denn die Inhalte der Blätter von links nach rechts gelesen ergibt in beiden Fällen: `Licht`, `km/h-Anzeige`, `Radio`.

10.3.2 Struktur-Äquivalenz

Haben Medienobjekte die gleiche Baumstruktur, so sind sie *struktur-äquivalent*. Diese Gleichheit ist unabhängig von der Saum-Äquivalenz. Zum Beispiel sind die Medienobjekte `Cockpit` (Abbildung 10.1) und `Cockpit2` (Abbildung 10.2) nicht struktur-äquivalent aber, wie schon gezeigt, saum-äquivalent. `Cockpit1` und das Medienobjekt `Cockpit3` (Abbildung 10.3), sind struktur-äquivalent, wie anhand der Baumstruktur ersichtlich wird. Auch wird deutlich, dass die Säume vom Medienobjekt `Cockpit2` und vom Medienobjekt `Cockpit3` verschieden sind. Dieser Unterschied schließt die Struktur-Äquivalenz nicht aus, da diese Eigenschaften nicht voneinander abhängen.

10.3.3 Identische Medienobjekte

Medienobjekte sind *identisch*, wenn sie sowohl saum-äquivalent als auch struktur-äquivalent sind. Sie müssen demnach eine gleiche Baumstruktur besitzen und ihre Säume müssen gleich sein.

10.3.4 Repräsentative Mengen

Unter der *repräsentativen Menge* versteht man eine Menge von Medienobjekten, deren Säume alle gleich sind. Es gibt aber kein Medienobjekt in der Menge, das eine gleiche Struktur wie ein anderes Medienobjekt innerhalb der Menge besitzt. Eine solche Menge gibt es zu jedem Medienobjekt. Sie kann durch einen Algorithmus eindeutig bestimmt werden, siehe [34], Seite 32 ff.. Im einfachsten Fall ist das Medienobjekt ein Atom. Dessen repräsentative Menge besteht lediglich aus diesem. Auch schon bei wenigen Atomen besteht die repräsentative Menge eines Medienobjekts aus vielen struktur-verschiedenen Medienobjekten mit gleichen Säumen.

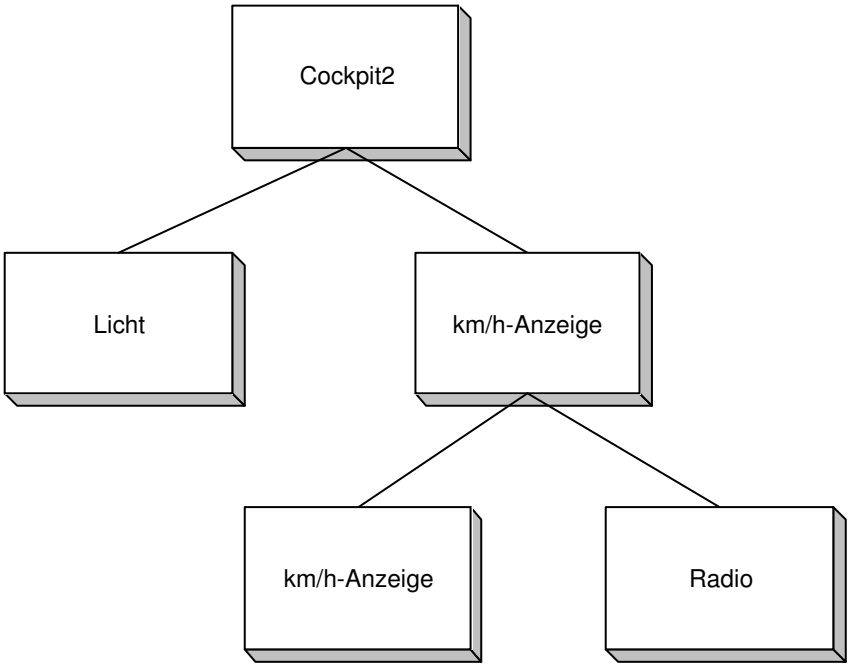

Abbildung 10.2: Baumdarstellung des Medienobjektes Cockpit2

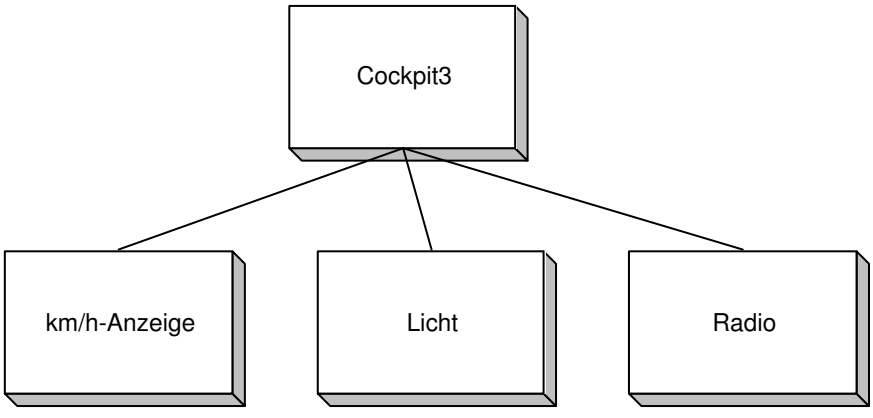

Abbildung 10.3: Baumdarstellung des Medienobjektes Cockpit3

10.4 Hyperdokumente

Ein *Hypermediadokument*, oder auch *Hyperdokument*, enthält als wesentlichen Bestandteil *Positionen*. Diese Positionen können in Paaren angeordnet werden. Ein Paar von Positionen ergibt einen *Link*. DoDL bietet eine Möglichkeit, diesen wichtigen Bestandteil von Hypermediadokumenten zu beschreiben. Dazu wurde zu Beginn die Struktur der zusammengesetzten Medienobjekte erklärt. Die Struktur, die durch die Komposition der Medienobjekte geschaffen wird und durch verschiedene Baumstrukturen ersichtlich ist, liefert eine Möglichkeit, Positionen in Medienobjekten zu erfassen. Die Position wird durch einen *Pfad* dargestellt (vgl.[8], Seite 36).

10.4.1 Pfade in Medienobjekten

Ein *Pfad* beschreibt die Position eines Submedienobjektes. Ein Pfad wird durch einen *Zahlenstring* repräsentiert. Jede Zahl des Zahlenstrings gibt an, welche Kante wir innerhalb eines inneren Knotens in der Baum-Darstellung wählen, um zu dem im Pfad referenzierten Submedienobjekt zu kommen. Neben den *starken Pfaden* existieren auch *schwache Pfade*. Während starke Pfade die Vorkommen aller struktur- und saum-äquivalenten Submedienobjekte zu einem gegebenen Medienobjekt liefern, führen *schwache Pfade* zu Submedienobjekten, die nur saum-äquivalent zu dem gegebenen Submedienobjekt sind.

In Abbildung 10.4 wird ein Pfad innerhalb des Medienobjektes `Cockpit4` zu dem Submedienobjekt `Anzeige` gesucht. Anhand der Darstellung wird ersichtlich, dass es zu diesem Medienobjekt zwei starke Pfade gibt. Einmal ein Pfad, der im Medienobjekt `Cockpit4` beginnt, über `Armaturen` verläuft und in dem Medienobjekt `Anzeige` endet. Der Zahlenstring des Pfades ist 21. Der zweite Pfad beginnt in dem Medienobjekt `Cockpit4` und verläuft über `Armaturen`. Dieser Pfad setzt sich dann über das Medienobjekt `Mittelkonsole` fort und endet schließlich im Medienobjekt `Anzeige`. Der Zahlenstring des Pfades ist 222.

10.4.2 Positionen in Medienobjekten

Durch zusammengesetzte Medienobjekte wird die Bestimmung der Position von festen Bezugssystemen oder konkreten Messsystemen abstrahiert. Jedes Medienobjekt, als Modul gesehen, stellt Pfad-Informationen bereit, die unabhängig von anderen Medienobjekten oder Bezugssystemen zur Bestimmung von Positionen innerhalb des Medienobjektes nötig sind. Die Submedienobjekte liefern, durch ihre eindeutige Anordnung, die nötigen Pfad-Teil-Informationen. Desweiteren existieren spezielle Positionen. Der *Beginn* eines Medienobjektes ist ein Pfad zum am weitesten links stehenden Atom des Saums. Analog findet man das *Ende* eines Medienobjektes in dem im Saum am weitesten rechts stehenden Atom.

Das Enthaltensein eines Medienobjektes innerhalb eines anderen wird als *Vorkommen* bezeichnet. Die Vorkommen entsprechen Positionen. Dabei kann ein Medienobjekt innerhalb eines anderen Medienobjektes auch mehr als einmal enthalten sein. Zu jedem Vorkommen gibt es einen Pfad. Die Menge θ enthält alle Vorkommen eines Submedienobjektes. Das Medienobjekt `Anzeige` kommt zum Beispiel in Abbildung 10.4 zweimal vor. Man erreicht das Medienobjekt vom Medienobjekt `Cockpit4` ausgehend über die Pfade 222 und 21. Damit ist

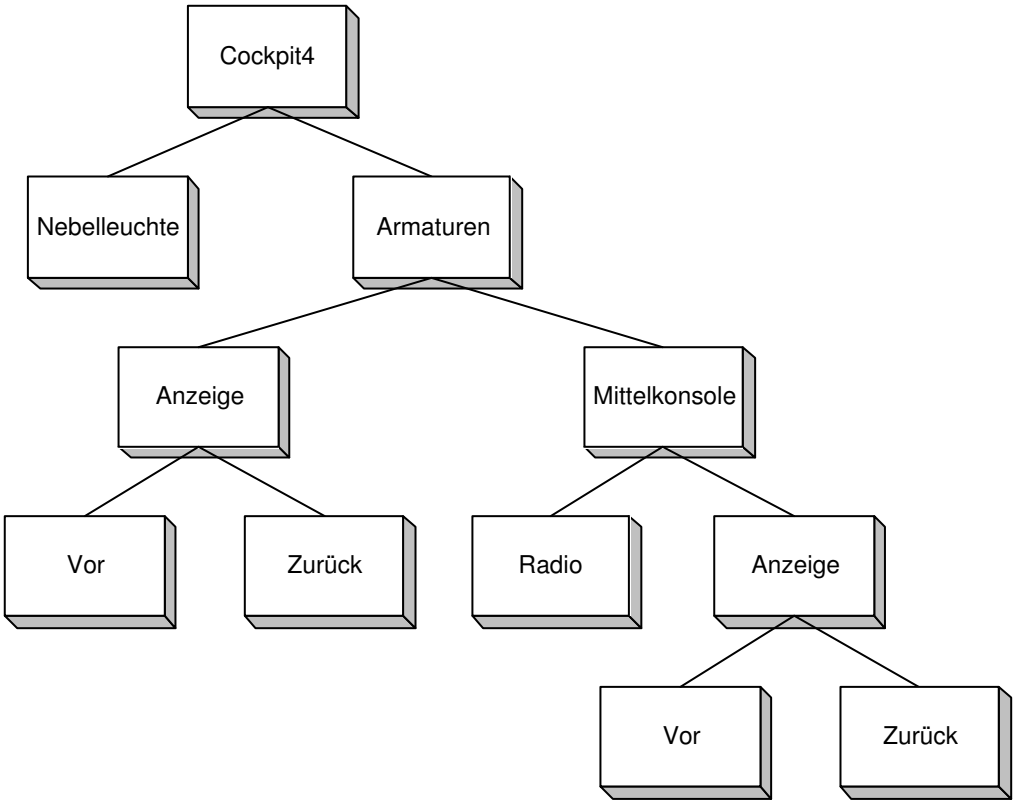

Abbildung 10.4: Baumdarstellung des Medienobjektes Cockpit4

die Menge θ in diesem Beispiel wie folgt definiert: $\theta = \{(222), (21)\}$.
Unter der *Menge aller Positionen* eines Medienobjektes versteht man die Menge aller Pfade innerhalb eines Medienobjektes.

10.4.3 Links in Medienobjekten

Ein *Link* besteht aus einem Paar von Positionen. Diese Positionen müssen nicht in verschiedenen Medienobjekten liegen. Die erste Position stellt die Quelle dar, und die zweite Position entspricht dem Ziel des Links.
So ist zum Beispiel „l=(1,22)“ innerhalb des Medienobjektes `Cockpit4` ein Link, der vom Beginn des Medienobjektes, `Nebelleuchte`, zu dem Submedienobjekt `Mittelkonsole` führt.

10.4.4 Eigenschaften von Positionen und Links

Ein Hyperdokument stellt Anforderungen an Positionen und Links. Diese sind:

- Eine Position kann Ziel beliebig vieler Links sein. So kann eine Quelle von verschiedenen Stellen aus erreicht werden.
- Eine Position kann Quelle nur genau eines Links sein. So ist ausgeschlossen, dass man von einer Position zu verschiedenen Zielen gelangt.
- Quelle und Ziel brauchen nicht verschieden sein. Damit sind Links deren Quell-Position Ziel-Position gleich sind möglich. Somit kann man Aktionen ausführen die nicht zu neuen Positionen führen. Sinnvoll wären zum Beispiel Zähler, oder das Starten von Applikationen.
- Quelle und Ziel müssen nicht in verschiedenen Medienobjekten liegen. Damit kann man Links innerhalb genau eines zusammengesetzten Medienobjektes definieren.
- Jede Quelle kann ein Ziel sein und jedes Ziel eine Quelle. Damit sind zyklische Verknüpfungsstrukturen erlaubt. Nicht jede mögliche Position muß zur Linkbildung herangezogen werden.

10.4.5 Eine Definition für Hyperdokumente

Medienobjekte, Links und Positionen charakterisieren ein Hyperdokument. Also besteht ein Hyperdokument H aus einem Tripel $H = (M, P, L)$, mit

- einer nicht-leere Menge M von Medienobjekten,
- einer beliebigen Menge P von Positionen in Medienobjekten aus der Menge M,
- und der Menge L von Links der Form (p, p') mit $p, p' \in P$.

Zusätzlich gelten die Eigenschaften von Positionen und Links für ein Hyperdokument H.

10.5 Fazit

Medienobjekte stellen einen wichtigen Bestandteil von Hyperdokumenten dar. Die Komposition von Medienobjekten gibt ihnen eine Struktur. Anhand dieser Struktur wird es möglich, in jedem Medienobjekt Positionen zu bestimmen. Nur die Struktur des Medienobjektes hat Einfluß auf die Bestimmung von Positionen in diesem. Mit Hilfe von Positionen können dann Links einfach konstruiert werden. Mit diesen Ansätzen ist es möglich mit Positionen, Links und Eigenschaften der Hyperdokumente algebraisch zu erfassen, siehe [34], Kap. 4. Darauf soll hier aber nicht eingegangen werden.

Kapitel 11

Die Sprache DoDL

Autor: *Evgenij Golkov*

11.1 Einleitung

In diesem Kapitel wird kurz vorgestellt, was die DoDL ist sowie deren Herkunft und Zweck beschrieben.

11.1.1 Was ist DoDL

DoDL, kurz für englische „Document description language“, wurde an dem Lehrstuhl für Softwaretechnik der Universität Dortmund entwickelt und erstmals in 1996 von Prof. Doberkat vorgestellt.

Der Zweck der Sprache DoDL ist die objektorientierte Beschreibung von Hyperdokumenten. Die wichtigste Eigenschaft der Spache ist dabei die Trennung zwischen dem Inhalt und der Linkstruktur der Dokumente

11.2 Grundlegende Ideen

In diesem Kapitel werden die grundlegende Ideen der Sprache vorgestellt. Dabei werden zuerst grundlegende Begriffe wie Medienobjekte, Hyperdokumente sowie Dokumente definiert, dann auf die wichtigsten Merkmale der Sprache wie Positionskonzept und Verknüpfungsstruktur eingegangen und anschliessend der Aufbau eines DoDL-Programms vorgestellt.

11.2.1 Medienobjekte, Hyperdokumente, Dokumente

*Definition: Ein **Medienobjekt** ist eine beliebige, digital speicherbare Informationseinheit.*

*Definition: Ein **Hyperdokument** ist eine Kollektion von Medienobjekten, die durch Links miteinander verbunden sind.*

*Definition: Ein **Dokument** ist ein Medienobjekt oder Hyperdokument.*

11.2.2 Positionskonzept

Das Positionskonzept ist der zentrale Begriff von DoDL.

*Definition: **Positionen** sind ausgezeichnete Stellen in Medienobjekten.*

Eine Position kann das Beginn oder das Ende eines Medienobjekts oder das Vorkommen eines Medienobjekts in einem anderen sein.

*Definition: Ein **Link** ist ein Paar ausgezeichneter Stellen in Medienobjekten, die Quelle und Ziel eines Links bilden.*

11.2.3 Verknüpfungsstruktur

Die Verknüpfungsstruktur baut auf Positionen auf.

*Definition: Eine **Verknüpfungsstruktur** ist ein gerichteter, attributierter Graph, der sämtliche in einem Hyperdokument verwendete Medienobjekte, die darin ermittelten Positionen sowie Links zwischen Positionen subsumiert.*

Die Knoten des Graphen entsprechen den Positionen, die Kanten den Links, und Attribute charakterisieren die Positionen.

11.2.4 DoDL-Programme

Ein DoDL-Programm besteht aus einer Spezifikation und einer Bindung.

Die Spezifikation besteht ihrerseits aus einer oder mehreren Klassen. Genau eine Klasse muss einen Eintrittspunkt besitzen und wird als „ausgezeichnete Klasse“bezeichnet. Diese Klasse muss eine Startmethode besitzen. Dieser Aufbau lässt ähnlichkeiten mit der Sprache Java feststellen: Die ausgezeichnete Klasse mit der Startmethode entspricht dann einer Java-Klasse mit main-Methode.

Der Begriff „Binding“ (engl. für Bindung) wird im nächsten Kapitel vorgestellt.

11.3 Funktionsweise

In diesem Kapitel wird die Funktionsweise der Sprache beschrieben. Es wird dabei auf die Begriffe eingegangen, die auch aus anderen objektorientierten Sprachen bekannt sind (Klassen, Aggregation und Benutzung, Vererbung). Dann wird der Begriff der Bindung erläutert, und anschliessend werden die Konstrukte an einem Code-Beispiel vorgestellt.

11.3.1 Klassen

In DoDL wird zwischen einfachen und komplexen Klassen unterschieden.

Eine einfache Klasse enthält eine documents-Sektion, in der Dokumentenvariablen (entsprechen den Attributen in einer OO-Sprache) definiert werden, und einer construct- Sektion, die Methoden zur Beschreibung der Verknüpfungsstruktur definiert. Diese sind:

- getBegin: gibt den Beginn eines Medienobjekts zurück
- getEnd: gibt das Ende eines Medienobjekts zurück
- getOcc („get occurence“): gibt das Vorkommen eines Medienobjeks im anderen zurück
- setLink: setzt einen Link zwischen zwei Positionen
- linkAll: definiert eine Liste von Links, die mehrere Quellen mit einem Ziel verbinden

Eine kompexe Klasse besitzt wie eine einfache Klasse eine documents- und eine construct-Sektion, zusätzlich aber auch eine declare-Sektion, die lokale Klassen definiert.

11.3.2 Aggregation und Benutzung

Wie auch in anderen objektorientierten Sprachen kann in DoDL Aggregation und Benutzung verwendet werden. Wird ein Dokumententyp in Dokumentendeklarationen verwendet, spricht man in DoDL von Aggregation. Wird dieser lediglich als Parameter in Methodendeklaration verwendet, spricht man dann von Benutzung.

Es ist anzumerken, dass eine Aggregation in DoDL immer eine Komposition ist, da DoDL keine Möglichkeit zur Referenzierung von Objekten bietet.

11.3.3 Vererbung

Ein weiteres aus anderen objektorientierten Sprachen bekanntes Konstrukt ist die Vererbung. In DoDL weist die Vererbung folgende Eigenschaften auf:

- Einfache Erbung: Eine Klasse kann von maximal einer Klasse erben, es ist also keine Mehrfacherbung möglich.
- Klassenvererbung: Eine Subklasse besitzt alle Dokumentenvariablen, Methoden und lokale Subklassen der Superklasse und kann eigene definieren
- Redefinition: eine Methode der Subklasse, die den gleichen Namen und die gleiche Signatur wie in der Superklasse besitzt, redefiniert die Methode der Superklasse
- Spätes Binden: Falls der Compiler die richtige Instanz für eine Methode nicht statisch ermitteln kann, kann die Zuordnung später erfolgen
- überladen: Unter der Voraussetzung, dass sich die Signaturen unterscheiden, können Methoden gleiche Namen haben.

11.3.4 Bindungen

em Definition: **Bindung** ist die Zuweisung eines Wertes jeder Dokumentenvariablen zur übersetzungszeit durch eine einfache Belegung.

Die Bindung erfolgt rekursiv, dass heißt alle Attribute der Superklasse werden auch belegt, sowie die Attribute der benutzten Klassen. Nach der Bindung dürfen die Variablenwerte nicht mehr geändert werden.

Wesentliches Merkmal einer Bindung ist eine von der Struktur des Dokuments getrennte Beschreibung der an der Konstruktion beteiligten Medienobjekte.

11.3.5 Code-Beispiel

Am Beispiel des folgenden Codes soll gezeigt werden, wie die vorgestellten Konstrukte in DoDL umgesetzt werden. Die Zeilennummern dienen lediglich als Erklärungshilfe und gehören nicht zum DoDL-Programm.

```
1       Class myClass is mySuperClass with
3       declare class myLocalClass is
4               ...
5               end myLocalClass;
6               ...
8       documents doc1: myDocType;
9               doc2: myDocType;
10              localClass: myLocalClass;
12      construct ...
14      end myClass;
```

Zeile 1 deklariert die DoDL-Klasse myClass, die von einer anderen DoDL-Klasse namens mySuperClass erbt.

Zeile 2 deklariert eine lokale Klasse myLocalClass.

Zeilen 3 bis 6 zeigen schematisch eine declare-Sektion. In der Zeile 3 wird die lokale Klasse myLocalKlass deklariert. Zeile 4 würde durch eine Beschreibung der Klasse ersetzt werden. Zeile 5 schliesst die Deklaration dieser Klasse, und anstelle der Zeile 6 könnten Deklarationen weiterer lokalen Klassen folgen.

Zeilen 8 bis 10 stellen eine documents-Sektion dar. In der 8.Zeile wird die documents-Sektion eingeleitet und die Dokumentenvariable doc1 vom Typ myDocType deklariert. Zeile 9 deklariert eine weitere Dokumentenvariable, und Zeile 10 deklariert die Variable localClass vom Typ myLocalClass.

Anstelle der Zeile 12 würde eine construkt-Sektion beschrieben werden, hier aber wird darauf nicht detailirt eingegangen.

Zeile 14 schliesst die Definition der Klasse.

11.3.6 Fehlende Konstrukte

Obgleich die Sprache DoDL viele Eigenschaften einer objektorientierten Sprache aufweist, fehlen einige Konstrukte. Dies sind insbesondere:

- Referenzen: DoDL bietet keine Möglichkeit, Referenzen zu definieren. Aus diesem Grund können wie schon erwähnt auch keine Assoziationen angelegt werden.
- Es gibt keine Objektkonstruktoren.
- Es sind keine Qualifizierer zur Einschränkung der Sichtbarkeit vorhanden.

11.4 Vorgehensmodell

Nachdem die grundlegenden Ideen und zentralen Konstrukte der Sprache DoDL vorgestellt wurden, wird hier die Vorgehensweise beim Erstellen von DoDL-Programmen gezeigt.

Als erster Schritt ist eine Untersuchung des Hyperdokuments in Hinblick auf die Anordnung der Medienobjekte notwendig. Die wichtigsten Anordnungsmöglichkeiten sind:

- Lineare Abbildung: Die Medienobjekte sind linear geordnet (Beispiel: Bücher)
- Vollständige Abbildung: Als Beispiel kann hier eine Museumsführung genannt werden, bei der nicht die Reihenfolge die entscheidende Rolle spielt, sondern die Tatsache, dass alle Exponate besichtigt werden.
- Zielorientierte Anordnung: Als Beispiel ist hier ein Lernsystem zu nennen, das entsprechend den Kenntnissen und Fähigkeiten des Benutzers entsprechenden Einstiegspunkt bietet, ohne alle Medienobjekte besichtigen zu mussen.
- Partielle Anordnung: Solche Anordnung besitzen zum Beispiel Lexika, in denen sehr viele verschiedenartige Medienobjekte enthalten sind, jeweils im Rahmen der Erklärung eines Begriffs geordnet.
- Hochgradig ungeordnet: Hier lässt sich keine Anordnung der Medienobjekte feststellen.. Als Beispiel kann man hier Literaturverzeichnisse nennen.

Nachdem die Anordnung der Medienobjekte ermittel wurde, wird die Verknüpfungsstruktur aufgestellt, wonach diese auf die DoDL-Spezifikation abgebildet wird. Dabei wird im wesentlichen zwischen folgenden drei Abbildungsmöglichkeiten unterschieden:

- Isomorphe Abbildung: Jedes Medienobjekt wird in einer eigenen Klasse abgebildet.
- Verknüpfungsstrukturen werden zu Mustern zusammengefasst. Jedes Muster wird in einer DoDL-Klasse abgebildet.
- Monolitische Abbildung: Hier gibt es nur eine DoDL-Klasse, in der das gesamte Dokument abgebildet wird.

Bei der Abbildung der DoDL-Spezifikation müssen aber auch Abbildungskriterien berücksichtigt werden. Dies sind insbesondere:

- Softwaretechnische Kriterien: Dabei steht die Verständlichkeit der DoDL- Spezifikation im Mittelpunkt, wodurch deren Wartbarkeit, Erweiterbarkeit, Testbarkeit und überschaubarkeit gefördert werden. Spielt dieses Kriterium die wichtigste Rolle, kann für die isomorphe Abbildung entschieden werden.
- Kriterien aus dem dem Einsatzbereich: Hier spielt die Möglichkeit, neue Medienobjekte in ein Hyperdokument aufzunehmen, die entscheidende Rolle. Um eine Entscheidung hinsichtlich der Abbildung zu treffen, werden hier weitere Kriterien benötigt.
- Systemtechnische Kriterien: Prozessorleistung und verfügbare Speichergrösse schränken hier die maximale Grösse der Spezifikation ein. Dabei kann eine monolitische Abbildung eine gute Lösung sein.

11.5 Fazit

Diese Arbeit sollte dazu dienen, den Einblick in die Sprache DoDL zu geben. An dieser Stellen werden die wichtigsten Merkmale der Spache zusammengefasst.

DoDL ist eine Sprache zur objektorientierten Beschreibung von Hypertextdokumenten, deren wichtigstes Merkmal die Trennung zwischen den Inhalten und der Linkstruktur ist. Sie besitzt zwar ähnlichkeiten mit anderen objektorientierten Sprachen, einige für diese Sprachen typischen Konstrukte fehlen jedoch. Dagegen weist DoDL bestimmte spezifischen Spracheigenschaften auf, die speziell auf die Konstruktion von Hyperdokumenten ausgerichtet sind.

Kapitel 12

Das DoDL-Cockpit

Autor: *Ulf Schellbach*

12.1 Einleitung

Eines der Ziele der PG HyCop ist es, ein digitales Autocockpit zu realisieren. Dieses Ziel soll jedoch nicht auf irgendeine beliebige Art und Weise erreicht werden, sondern indem wir das Cockpit als Hyperdokument auffassen und es unter Anwendung der Konzepte von DoDL spezifizieren und implementieren.
Der vorliegende Aufsatz soll mögliche Schritte auf dem Weg zum oben genannten Ziel aufzeigen und anhand einfacher Beispiele erläutern. Wir werden uns im Hinblick auf unser Ziel sinnvollerweise zunächst darüber Gedanken machen, welche funktionellen Eigenschaften unser Cockpit haben soll (Abschnitt 12.2). Davon ausgehend ist zu ergründen, auf welche Weise das Cockpit als Hyperdokument aufgefasst werden kann, d.h. es sind sinnvolle, nicht-lineare Verknüpfungen innerhalb des Cockpits und die unterliegenden Medienobjekte zu identifizieren. Diesem Thema widmet sich die Einleitung von Abschnitt 12.3. Nun stehen wir vor dem Problem, wie wir ein Hyperdokument auf eine objektorientierte Klassenstruktur abbilden können. Die Abschnitte 12.3.1 und 12.3.2 gehen darauf ein, wie wir uns zur Lösung dieses Problems die Konzepte von DoDL zunutze machen können.
Es sei darauf hingewiesen, dass die folgenden Ausführungen eine Zusammenfassung wesentlicher Ideen aus [34] darstellen.

12.2 Ein digitales Autocockpit

Das Cockpit soll sowohl Anzeigeinstrumente als auch Bedienelemente auf einem berührungssensitiven Flachbildschirm anzeigen. Als Anzeigeinstrumente wären z.B. ein Geschwindigkeitsmesser, ein Drehzahlmesser, eine Temperaturanzeige sowie eine Tankanzeige und ein Kilometerzähler denkbar. Die verschiedenen Bedienelemente könnten etwa die Nutzung einer Klimaanlage, eines Radios und CD-Spielers, eines Bordcomputers, eines Navigationssystems und eines Tempomaten erlauben. Die Verwendung eines berührungssensitiven Flachbildschirmes

soll die Interaktion des Fahrers mit dem Cockpit vereinfachen, sodass dieser durch jenes nicht allzu sehr vom Geschehen auf der Straße abgelenkt wird.
Die Eigenschaft, digital zu sein, macht das Autocockpit recht wandlungsfähig. Form und Farbe der Anzeigen können variiert werden. Darüber hinaus ist es denkbar, dass das Cockpit in unterschiedlichen Situationen unterschiedliche Auswahlen von Anzeige-instrumenten undBedienelementen zeigt. Wenn man sich beispielsweise im Stadtverkehr befindet, benötigt man in der Regel keinen Tempomaten, was dessen Anzeige demnach erübrigt. Fährt man hingegen auf der Autobahn, kann ein Tempomat durchaus von Nutzen und seine Anzeige gerechtfertigt sein.
Die Menge aller zu einem gegebenen Zeitpunkt angezeigten Anzeigeinstrumente und Bedienelemente soll im Folgenden als Ansicht des Cockpits bezeichnet werden. Der Begriff der Ansicht eines Cockpits entstand in Anlehnung an den Begriff der Sicht auf ein Hyperdokument, um den engen Bezug des Cockpits zum Hyperdokument zu verdeutlichen und einen Hinweis darauf zu geben, inwieweit das Cockpit als Hyperdokument aufgefasst werden kann.

12.3 Das Cockpit als Hyperdokument

Wenn wir das Cockpit als Hyperdokument auffassen wollen, müssen wir sowohl sinnvolle, nicht-lineare Verknüpfungen innerhalb des Cockpits als auch die unterliegenden Medienobjekte identifizieren.
Als Medienobjekte sind insbesondere die verschiedenen Anzeigeinstrumente und Bedienelemente zu erkennen. Die meisten Anzeigeinstrumente sind dynamischer Natur. Die durch sie angezeigten physikalischen Größen, wie z.B. die momentane Geschwindigkeit des Autos, ändern sich fast kontinuierlich. Im Gegensatz dazu sind die meisten Bedienelemente eher statischer Natur. Ihre Funktionalität beschränkt sich zumeist auf diejenige eines Schalters. Somit hat man zwei Sorten von Medienobjekten, die letztlich in sehr unterschiedlicher Weise zu realisieren sein werden.
Untersucht man das Cockpit nun hinsichtlich nicht-linearer Informationsdarstellung, so findet man auch hier zwei verschiedene Sorten von Verknüpfungen. Veränderungen der dargestellten Information können nämlich sowohl durch situationsbezogene Zusammenhänge als auch durch nutzerdefinierte Aktionen erfolgen. Wenn durch das „Anklicken“ der Tankanzeige, indem der Bildschirm an der entsprechenden Stelle berührt wird, z.B. die Anzeige detaillierterer Informationen bzgl. des Motorzustandes ausgelöst wird, handelt es sich dabei offenbar um eine Veränderung der dargestellten Information durch eine nutzerdefinierte Aktion. Als Beispiel für einen situationsbezogenen Zusammenhang, der zum Auslöser für die Veränderung der dargestellten Information wird, mag Folgendes dienen. Falls das Laufzeitsystem über spezielle Sensoren erfährt, dass die Benzinreserven knapp geworden sind, löst es die Anzeige einer geeigneten Warnmeldung aus.

Nachdem wir uns klargemacht haben, wie das Cockpit als Hyperdokument aufgefasst werden kann, stehen wir nun vor der Frage, wie wir ein Hyperdokument auf eine objektorientierte Klassenstruktur abbilden können, denn wir wollen unser Cockpit ja letztendlich mit einer objektorientierten Programmiersprache implementieren. Wie in der Einleitung erwähnt, liefern uns die Konzepte von DoDL eine Antwort auf diese Frage.

12.3.1 Die Abbildung eines Hyperdokumentes auf eine objektorientierte Klassenstruktur mittels der wesentlichen Konzepte von DoDL

Die Abbildung eines Hyperdokumentes auf eine objektorientierte Klassenstruktur erfolgt gemäß DoDL-Konzepten unter strikter Trennung von Verknüpfungsstruktur und konkreten Medienobjekten. Jedes Medienobjekt findet sich in der objektorientierten Klassenstruktur als Attribut einer Klasse wieder. Die konkrete Belegung der Attribute erfolgt getrennt von der Beschreibung der Verknüpfungsstruktur durch ein abschließendes sogenanntes Binding. Die Definition von Positionen und Links geschieht konstruktiv mittels spezieller Methoden, die auf die durch Attribute repräsentierten Medienobjekte angewendet werden. So liefert die Methode m.getBegin() für ein als Attribut gegebenes Medienobjekt m beispielsweise die Anfangsposition von m. Medienobjekte werden also durch die Definition von Positionen und Links in keinerlei Weise verändert. Dadurch können einzelne Medienobjekte unter gleichbleibender Verknüpfungsstruktur ohne zusätzlichen Aufwand ausgetauscht werden. Diese Tatsache verbessert die Wartbarkeit eines Hyperdokumentes.
Die Trennung von Verknüpfungsstruktur und konkreten Medienobjekten bildet einen grundlegenden Unterschied zur Realisierung von Hyperdokumenten mittels Sprachen wie etwa HTML. Bei HTML werden Links durch spezielle Markierungen („tags“) innerhalb der HTML-Dateien realisiert. In aller Regel werden dabei in Form von Textdateien gegebene Medienobjekte zur Definition von Links verändert, indem die Verknüpfungsinformationen in ihnen eingebettet werden.

12.3.2 Umsetzung der DoDL-Konzepte

Dieser Abschnitt soll andeuten, wie die vorangegangenen Ideen und Konzepte zur Realisierung des digitalen Autocockpits umgesetzt werden können. Dem Konzept der Trennung von Verknüpfungsstruktur und konkreten Medienobjekten folgend wird man sich zunächst mit der Analyse, der Spezifikation und der Implementierung der Verknüpfungsstruktur des Cockpits befassen. Erst danach wird es darum gehen, konkrete Medienobjekte durch Belegung der Dokumentvariablen einzubinden. Die nachfolgenden Abschnitte gehen skizzierend auf die Analyse und Spezifikation der Verknüpfungsstruktur ein.

Analyse der Verknüpfungsstruktur

Man kann die Verknüpfungsstruktur in zwei Schritten analysieren. Zunächst kann die Interaktion zwischen Benutzer und System anhand von Anwendungsfalldiagrammen untersucht und veranschaulicht werden. Nachdem alle Systemkomponenten und Medienobjekte sowie deren Verhältnis zum Benutzer des Systems identifiziert worden sind, kann man sich der Untersuchung der hypermedialen Verknüpfungsstruktur unter Verwendung konzeptioneller UML-Klassendiagramme zuwenden.

Abbildung 12.1 zeigt die Sicht des Fahrers auf das Cockpit. Es gibt verschiedene Ansichten des Cockpits für verschiedene Situationen. Dabei werden die einzelnen Ansichten als Erweiterungen einer allgemeinen Ansicht aufgefasst. Sie unterscheiden sich durch die jeweilige Auswahl an konkret dargestellten Anzeigeinstrumenten und Bedienelementen. Diese grobe Sicht auf

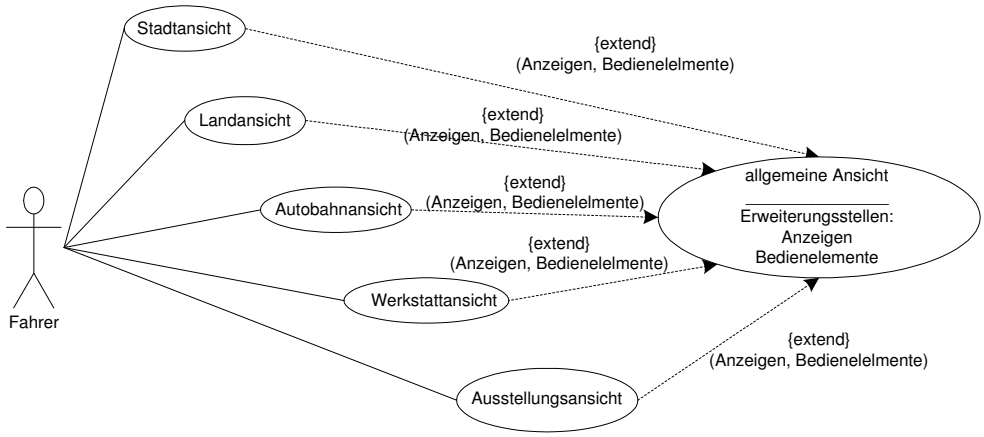

Abbildung 12.1: Cockpitansichten

das Cockpit kann verfeinert werden. Schauen wir uns beispielsweise die Stadtansicht genauer an (vgl. Abb. 12.2):

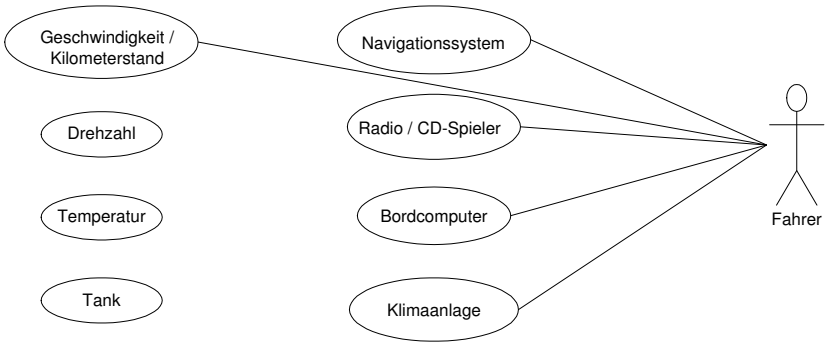

Abbildung 12.2: Stadtansicht

Das Anwendungsfalldiagramm zeigt auf der linken Seite verschiedene Anzeigeinstrumente und in der Mitte eine Auswahl an Bedienelementen des Cockpits. Der Fahrer interagiert mit dem System lediglich über die Bedienelemente, was durch die Verbindungslinien veranschaulicht wird. Einzig der Tageskilometerzähler, der zur Menge der Anzeigeinstrumente gezählt wird, bildet hier eine Ausnahme, denn er soll vom Fahrer zurückgesetzt werden können. Es ist klar, dass das Laufzeitsystem einen maßgeblichen Einfluss auf die Funktionalität des Systems und all seiner Komponenten hat. Daher versteht man das Laufzeitsystem als implizit mit allen Use-Cases verbunden.

Die hypermediale Verknüpfungsstruktur wird unter Verwendung von konzeptionellen UML-Klassendiagrammen veranschaulicht. Die Anwendungsfallanalyse aus Abbildung 12.1 liefert uns die Information, dass keine der modellierten Ansichten gleichzeitig mit einer anderen Ansicht zu sehen sein soll. Es muss jedoch hypermediale Verknüpfungen zwischen den Ansichten geben, da ein situationsabhängiger und kontrollierter Wechsel der Ansichten möglich sein soll.

In Abbildung 12.3 stellen die Klassen jeweils Ansichten dar, während die Assoziationen die Rolle von hypermedialen Verknüpfungen übernehmen.

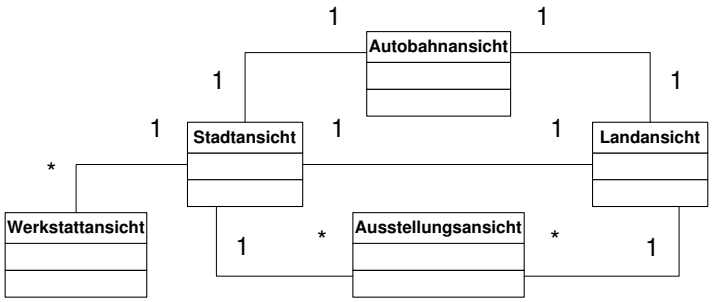

Abbildung 12.3: Konzeptionelles Modell für Klassenansichten

Stadt-, Land- und Autobahnansicht sind jeweils eindeutig und stehen daher in einer (1:1)-Beziehung zueinander. Angenommen, wir fahren auf der A3 von Köln in Richtung Duisburg und wechseln am Kreuz Breitscheid auf die A52 in Richtung Essen. Die Eindeutigkeit der Autobahnansicht sichert, dass sich die Ansicht des Cockpits beim Autobahnwechsel nicht verändert und der Fahrer somit nicht irritiert wird. Auch mehrdeutige Beziehungen sind denkbar. Die (1:*)-Beziehung zwischen Stadt- und Werkstattansicht soll etwa verdeutlichen, dass für unterschiedliche Werkstätten auch unterschiedliche Werkstattansichten erforderlich sein können. Zwischen manchen Ansichten bestehen keine direkten Verknüpfungen. Das Fehlen einer direkten Verbindung von der Autobahnansicht zur Ausstellungsansicht soll beispielsweise verdeutlichen, dass man von der Ausfahrt eines Autohauses nur in äusserst seltenen Fällen irekt auf eine Autobahn auffährt, ohne zuvor eine Landstraße zu befahren oder eine Stadt zu durchqueren. Hier geht es also um ie Erfassung der groben Struktur des Cockpits.

Im nächsten Verfeinerungsschritt betrachten wir bereits Medienobjekte und ihre Verknüpfung. Abbildung 12.4 zeigt inen Teil der Stadtansicht. Bei den Klassen handelt es sich immer noch nicht um „echte“ Klassen im bjektorientierten Sinn. Sie nehmen konzeptionell die Rolle „Medienobjekt“ ein. Die Relationen, welche hier als Navigierbarkeiten egeben sind, stellen Links dar. Von jedem Anzeigeinstrument existiert ein Link zu individuellen Warnmeldungen. Das bedeutet, dass von en Anzeigeinstrumenten in bestimmten Situationen Warnmeldungen ausgelöst werden. Ein nahezu leerer Tank, überhöhte eschwindigkeit oder eine zu hohe Drehzahl können hier als Auslöser dienen. Die durch Navigierbarkeiten gegebene Verlinkung der Anzeigeinstrumente impliziert eine Reihenfolge unter den Instrumenten. Diese kann später dazu genutzt werden, die Instrumente auf dem digitalen Cockpit z.B. reihenfolgegetreu von links nach rechts anzuzeigen. Die Attribute der Anzeigeinstrumente vermitteln die physikalischen Maßeinheiten der jeweils angezeigten Werte.

Auch das Navigationssystem kann zum Auslöser von Warnmeldungen werden. Ein möglicher Auslöser könnte z.B. das Verlassen des vom Navigationssystem vorgeschlagenen Weges zur Zieladresse sein. Die übrigen vom Navigationssystem ausgehenden Links zeigen, zu welcher Art von Information das Navigationssystem Zugang geben soll.

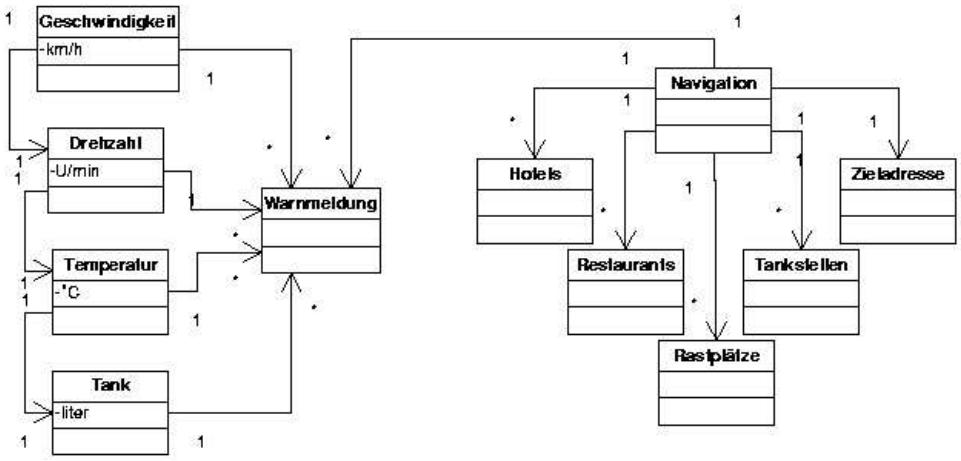

Abbildung 12.4: Verfeinerung der Stadtansicht

Betrachten wir zuletzt noch den Übergang von der Analyse der Verknüpfungsstruktur zu ihrer Spezifikation.

Spezifikation der Verknüpfungsstruktur

Die Abbildung der hypermedialen Verknüpfungsstruktur des Cockpits auf eine objektorientierte Klassenstruktur erfolgt nun, indem wir Medienobjekte als Attribute von Klassen repräsentieren sowie Positionen und Links durch Methoden definieren. Der folgende die Stadtansicht betreffende Ausschnitt aus einem spezifizierenden UML-Klassendiagramm (Abb. 12.5) weist eine Klassenstruktur auf, die aus einer sowohl nicht-isomorphen als auch nicht-monolithischen Abbildung resultiert (vergleiche hierzu Evgenijs Aufsatz zum Thema „Die Sprache DoDL“).

Unter monolithischer Abbildung wäre das gesamte Cockpit als eine einzige Klasse realisiert worden, sodass die interne Verknüpfungsstruktur nur noch anhand der Methoden hätte erkannt werden können. Im Gegensatz dazu wäre unter isomorpher Abbildung jedes Medienobjekt einer eigenen Klasse zugeordnet worden. In diesem Fall spiegelte die Klassenstruktur in direkter Manier die Verknüpfungsstruktur wider. Im obigen Beispiel hat man jedoch die allgemeine Verknüpfungsstruktur der Ansicht, die ja einen echten Teil der Verknüpfungsstruktur des Cockpits bildet, als eigene Klasse erfasst, welche speziellen Ansichten als Oberklasse dient. Hier bilden Klassen also sinnvoll erscheinende Gruppierungen von Medienobjekten.

12.4 Fazit

Die Ausführungen dienen dazu, zwei Dinge bewusst zu machen. Zum einen, dass ein digitales Autocockpit auf natürliche Art und Weise als Hyperdokument aufgefasst werden kann. Zum anderen, dass Hyperdokumente vermöge der Konzepte von DoDL auf eine objektorientierte Klassenstruktur abgebildet werden können. Dies geschieht so, dass ihre Wartbarkeit gegenüber

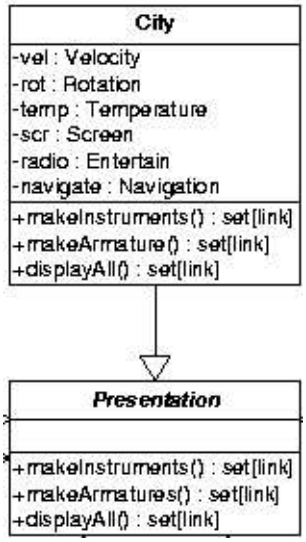

Abbildung 12.5: Ausschnitt aus einer objektorientierten Klassenstruktur des Cockpits

Dokumenten, die z.B. mit HTML realisiert sind, deutlich verbessert wird, wenn insbesondere die Austauschbarkeit der Medienobjekte von Belang ist.

Teil II

Anforderungen

Kapitel 13

Eine Urlaubsfahrt mit HyCop

Das folgende Szenario stellt den Ausgangspunkt für die Anforderungsanalyse dar. Es dient im Wesentlichen der Zusammenfassung vieler typischer Situationen, die sich vor und bei einer Autofahrt ergeben können. Selbstverständlich wird hierbei dem Fahrzeugcockpit und seiner Funktionalität die meiste Beachtung zuteil. Die verschiedenen Elemente des Szenarios wurden im Rahmen einer PG-Sitzung zusammengetragen, gruppiert und später zu einer durchgängigen Geschichte komponiert.

Die zugrundeliegende Absicht ist, mit den aus dem Szenario ableitbaren *Anwendungsfällen* bereits einen Großteil der Funktionalität des Cockpits beschreiben zu können.

13.1 Packen, Personalisieren und Losfahren

Autoren: *Daniel Mölle*
Markus Niehammer

Der Fahrer öffnet das Auto per Fernbedienung. Dabei schaltet das Cockpit in die Stand-Ansicht. Das Gepäck wird im Kofferraum verstaut und ein Koffer auf dem Dachgepäckträger platziert. Daraufhin gibt das Fahrzeug eine Warnmeldung aus, weil die Gewichtssensoren im Dach erkennen, dass die zulässige Dachlast überschritten ist. Es wird umgepackt und erneut die Dachlast überprüft. Das Gesamtgewicht des Fahrzeugs wird laufend im Fahrer-Display angezeigt, ist aber unkritisch.

Der Fahrer nimmt das Bedienteil für den Routenplaner aus dem Cockpit, der im Haus programmiert wird. Zusätzlich zum eigentlichen Zielort der Reise werden auch einige Sehenswürdigkeiten als Zwischenpunkte in die Route eingeplant. Die Berechnung der Route erfolgt später beim Einsetzen des Bedienteils in das Cockpit.

Alle steigen ins Auto. Die Insassen identifizieren sich durch mitgeführte Chipkarten, die sie in dafür vorgesehene Kartenleser an den Sitzplätzen stecken. Daraufhin werden die individuellen Einstellungen hergestellt. Diese sind auf den Chipkarten gespeichert, so dass sie auch in anderen Fahrzeugen verwendet werden können. Die Einstellungen betreffen nicht nur die Sitzpositionen und die Ausrichtung der Spiegel, sondern auch die Konfiguration der Anzeige-

und Bedienelemente.

Das Cockpit zeigt für den Fahrer weiterhin die Stand-Ansicht, auf der Gesamtgewicht, Reifendruck und ähnliche Fahrzeugdaten abgelesen werden können. Gemäß den Voreinstellungen ist der Monitor des Beifahrers zunächst aus, während für die beiden Mitfahrer ein Videospiele-Menü bzw. die Radiobedienung angezeigt werden.

Der Fahrer setzt das Routenplaner-Modul ein. Daraufhin wird eine Route über die vorher festgelegten Stationen berechnet. Zusätzlich wird eine Internetverbindung zur Informationszentrale aufgebaut, um Verkehrsmeldungen in die Routenplanung einzubeziehen. Weiterhin ist dadurch ein Vergleich der aktuellen Benzinpreise an den umliegenden Tankstellen möglich.

Vor dem Starten des Motors arbeitet das Cockpit eine Checkliste ab, die u.a. das Gesamtgewicht und den Reifendruck umfasst. Dabei wird durch Sensoren in den Gurtschlössern erkannt, dass einer der Mitfahrer noch nicht angeschnallt ist, und ein entsprechender Hinweis auf dessen Display sowie beim Fahrer ausgegeben. Nach einem erfolgreichen Durchlauf der Checkliste startet der Motor und die Anzeige des Fahrers wechselt in die Stadtansicht. Der Fahrer startet die Wiedergabe von MP3-Dateien über das Lautsprechersystem des Fahrzeugs.

Beim Verlassen der Einfahrt meldet das Cockpit ein Hindernis, da ein Abstandswarner einen zu geringen Abstand zwischen Fahrzeug und Hecke feststellt. Nach dem Umfahren des Hindernisses wird die Straße erreicht.

13.2 Tanken

Um nicht während der Autobahnfahrt eine Tankstelle ansteuern zu müssen, entscheidet der Fahrer, die nächstgelegene Tankstelle in der Stadt zu besuchen. Er informiert das Navigationssystem über sein Vorhaben und erhält als Rückmeldung eine Liste der nächstgelegenen Tankstellen. Der Fahrer wählt aus dieser Liste eine Tankstelle aus und veranlasst das Navigationssystem, ihm auf geeignete Weise den Weg zur ausgewählten Tankstelle zu beschreiben.

Bald darauf wird die Tankstelle erreicht. Bei der Auffahrt auf das Tankstellengelände baut sich automatisch eine drahtlose Hochgeschwindigkeitsverbindung zwischen dem Cockpit und der Bedienstation der Tankstelle auf, über die die Tankstelle ihr Serviceangebot übermittelt. Der Fahrer wählt den Service *Tanken* und erhält daraufhin von der Bedienstation der Tankstelle eine Liste der momentan verfügbaren Zapfsäulen. Von diesen wählt der Fahrer eine aus, bringt sein Gefährt in die entsprechende Position und tankt.

Währenddessen hat der Beifahrer das Bedürfniss eine Flasche Wasser zu kaufen. Also wählt er aus den Serviceleistungen der Tankstelle, die das Cockpit anzeigt, den Service *Einkaufen*. Daraufhin erscheint auf dem Display des Cockpits eine Liste der von der Tankstelle zum Verkauf angebotenen Konsumgüter, anhand derer der Beifahrer eine Bestellung aufgibt. Kurze Zeit später reicht ihm ein Angestellter der Tankstelle die bestelle Flasche Wasser.

Nachdem alle gewünschten Serviceleistungen erbracht wurden, wickelt der Fahrer die Bezahlung auf elektronischem Wege über das elektronische Bezahlsystem ab. Nun kann die Fahrt weitergehen. Das Verlassen der Tankstelle hat den automatischen Abbau der Verbindung zwischen dem Cockpit und der Bedienstation der Tankstelle zur Folge.

Die Fahrt wird über die Landstraße Richtung Autobahn fortgesetzt. Der Wechsel von der Stadt auf die Landstraße sowie von der Landstraße auf die Autobahn verursacht jeweils einen Wechsel der Cockpitansicht.

Die Mitfahrer auf den Rücksitzen entscheiden sich dazu, sich von anderen Medien unterhalten zu lassen als von den MP3-Dateien des Fahrers. Der eine wählt einen DVD-Film, der andere schaut fern. Beide lassen dabei den Ton über Kopfhörer wiedergeben.

13.3 Pause auf dem Rastplatz

Nach einiger Zeit steht eine Pause an. Daher stellt der Beifahrer eine entsprechende Anfrage an das Navigationssystem, das einen passenden Rastplatz mit Toilette, Restaurant und Spielplatz sucht. Das Cockpit zeigt den nächstgelegenen passenden Rastplatz inklusive Entfernung und weiteren Einrichtungen an und integriert ihn in die geplante Route. Der Fahrer fährt auf den Rastplatz und sucht sich einen Parkplatz. Beim Abschalten des Motors wird die MP3-Wiedergabe automatisch angehalten. Die Mitfahrer im Fond unterbrechen ebenfalls die Wiedergabe ihrer derzeit abgespielten Medien. Alle Insassen verlassen das Auto und die Sicherungssysteme werden aktiviert.

Nach der Pause findet ein Fahrerwechsel statt, woraufhin die Sitzplätze durch Einstecken der Chipkarten neu konfiguriert werden. Beim Ausparken wird der neue Fahrer durch das Cockpit unterstützt, anschließend folgt er den Anweisungen des Navigationssystems und setzt seine Fahrt auf der Autobahn fort. Der neue Fahrer entscheidet sich für die Wiedergabe eines Radiosenders, die Mitfahrer hinten im Auto schauen ihre schon zuvor gewählten Medien weiter an. Nach einiger Zeit legt der Fahrer eine Audio-CD ein und die anderen Mitfahrer spielen unter Nutzung ihrer jeweiligen Kopfhörer und Flachbildschirme ein Videospiel miteinander.

Es wird spät am Tag, und langsam setzt die Dämmerung ein. Bei Unterschreiten einer gewissen Lichtintensität schalten sich die Scheinwerfer und Rücklichter automatisch ein. Auch Instrumente, Bildschirme und Innenbeleuchtung werden an die veränderten Lichtverhältnisse angepasst.

Das Navigationssystem erhält von der Informationszentrale einen Stauhinweis auf der geplanten Route. Das Cockpit meldet dies dem Fahrer und schlägt eine alternative Route vor. Der Fahrer akzeptiert die neue Route und folgt den Anweisungen des Navigationssystems. Unterdessen haben die Mitfahrer ihr Videospiel beendet. Der Beifahrer liest danach einige Webseiten und seine E-Mails. Die Mitfahrer im Fond schlafen ein, ihre Displays werden nach einiger Zeit automatisch abgeschaltet.

Es wird Nacht. Auch der Fahrer wird müde, und eine Übernachtung steht an. Der Beifahrer stellt eine Hotelanfrage an das Navigationssystem. Hierfür gibt er einen Preis bzw. eine Preiskategorie sowie die benötigte Anzahl an Zimmern und die gewünschte Entfernung zum aktuellen Standpunkt an. Die Eingaben werden an die Informationszentrale übermittelt, die ein passendes Hotel sucht.

Das Navigationssystem zeigt alle gefundenen Hotels mit detaillierten Informationen über Verfügbarkeit und Ausstattung, sowie weitere Unterkünfte mit Telefonnummer an. Der Beifahrer wählt ein Hotel aus. Ist dieses in der Informationszentrale registriert, so kann er direkt

reservieren und bezahlen. Andernfalls verbindet das Cockpit ihn telefonisch mit dem Hotel, um die Zimmerverfügbarkeit zu klären und eventuell zu buchen.

Das gewählte Hotel wird automatisch in die Route integriert und das Navigationssystem leitet den Fahrer zum Hotel. Dort wird das Auto über Nacht geparkt und die Sicherungssysteme werden aktiviert.

13.4 Weiterfahrt mit Panne

Autoren: *Evgenij Golkov*
Tobias Wolf

Am nächsten Morgen geht die Fahrt weiter. Plötzlich geht der Motor aus und die Warnblinker werden eingeschaltet. Der Fahrer liest auf seinem Display einen Warnhinweis über einen nicht näher spezifizierten Motordefekt und lenkt den Wagen an den Fahrbahnrand. Das Cockpit wird in den Stromsparmodus geschaltet, d.h. alle Komponenten bis auf die wichtigsten werden abgeschaltet. Auf dem Display erscheint nun ein Dialog vom Fahrzeugdiagnosetool, in dem dem Fahrer angeboten wird, einen Pannendienst zu benachrichtigen. Dieses bestätigt der Fahrer. Es wird eine Verbindung zum nächstgelegenden Pannendienst hergestellt, wobei alle benötigten Daten übermittelt werden. Dazu gehören die Position und der Typ des Wagens und die vom Fahrzeugdiagnosetool ermittelten Informationen über die Art der Panne. Anhand dieser kann die Werkstatt entscheiden, ob eine Vor-Ort-Reparatur möglich ist oder ob ein Abschleppwagen geschickt werden muss. Die Werkstatt entscheidet sich für einen Abschleppwagen.

Kurze Zeit später trifft der Abschleppwagen beim Fahrzeug ein. Der Mechaniker führt mittels eines mobilen Diagnosegerätes eine detailierte Diagnose durch und ermittelt die benötigten Ersatzteile. Nach Absprache mit dem Fahrer können die Ersatzteile direkt bei der Werkstatt bestellt werden, während der Abschleppwagen das Pannenauto zur Werkstatt schleppt.

Der Abschleppwagen trifft an der Werkstatt ein, und das Werkstattsystem authentifiziert sich am Fahrzeug, um den Motorraum frei- und die Werkstattansicht einzuschalten. Durch die im Vorfeld ermittelten Daten liegen die benötigten Ersatzteile schon bereit und der Wagen kann repariert werden. Während sich der Wagen in der Werkstatt befindet, werden, wenn nötig, aktuellere Softwareversionen aufgespielt und anonymisierte Statistikdaten an den Hersteller übermittelt.

Die Bezahlung der Reparatur erfolgt über das elektronische Zahlsystem. Im Anschluss kann die Reise fortgesetzt werden, und alle Komponenten nehmen wieder ihren Betrieb auf.

Kapitel 14

Anwendungsfälle

Einleitung

Dieses Kapitel beschreibt die Anwendungsfälle des hypermedialen Cockpits. Wir unterscheiden hierbei drei verschiedene menschliche Akteure. Der allgemeinste Akteur ist der *Insasse*, also eine Person, die sich im Fahrzeug aufhält. Ein spezieller Insasse ist der *Fahrer*, der das Fahrzeug steuert. Alle Insassen, die nicht Fahrer sind, sind *Mitfahrer*. Der Akteur *HyCop* steht für das hypermediale Cockpit, weitere wichtige Komponenten wie das *Navigationssystem* treten als eigenständige Akteure auf. Auch Systeme können als Akteure auftreten, wenn sie mit anderen System interagieren.

Abbildung 14.1 gibt einen Überblick über die Anwendungsfälle des hypermedialen Cockpits.

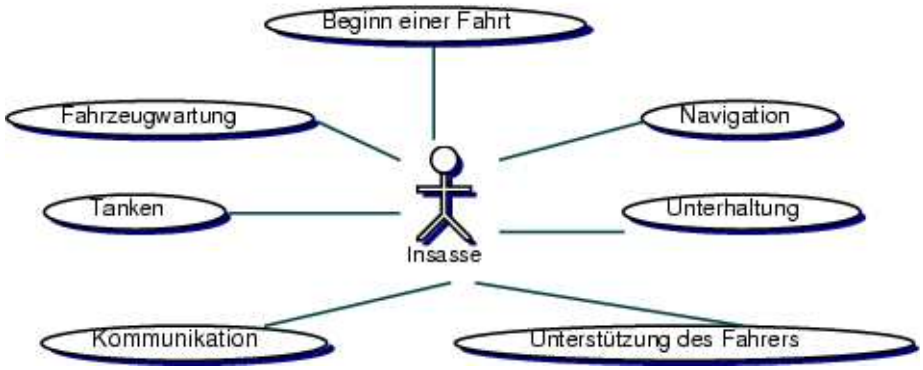

Abbildung 14.1: Die Anwendungsfälle des hypermedialen Cockpits

14.1 Beginn einer Fahrt

Autoren: *Daniel Moelle, Markus Niehammer*

Dieser Abschnitt beschreibt Anwendungsfälle, die beim Beginn einer Fahrt eine Rolle spielen.

Die in den Abschnitten 14.1.2 und 14.1.3 beschriebenen Anwendungsfälle treten bei jeder Fahrt auf. Der in Abschnitt 14.1.1 beschriebene Anwendungsfall spielt dagegen im Allgemeinen nur bei längeren Reisen oder Transportfahrten ein Rolle.

14.1.1 Packen

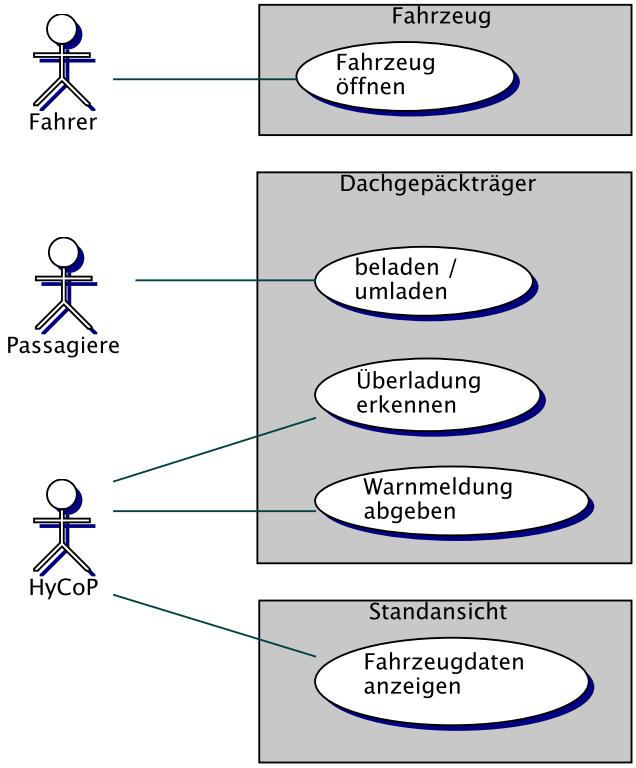

Abbildung 14.2: Anwendungsfalldiagramm Packen

Dieser Anwendungsfall (vgl. Abbildung 14.2) umfasst das Öffnen und Beladen des Fahrzeugs sowie die entsprechenden Reaktionen des hypermedialen Cockpits. Der Fahrer als Akteur kann das Fahrzeug, das in diesem Zusammenhang als physikalisches System aufgefasst werden soll, z.B. durch eine im Schlüssel enthaltene Fernbedienung öffnen. Beim Beladen des Dachgepäckträgers, der ein Teilsystem des Fahrzeugs darstellt, kommen hingegen alle Mitfahrer als Akteure in Frage.

Für HyCop ergeben sich hierbei mehrere Aktivitäten. Einerseits kann es eine Überladung des Gepäckträgers erkennen und eine entsprechende Warnmeldung abgeben, andererseits soll es die Fahrzeugdaten, darunter insbesondere die Zuladung, in der Standansicht anzeigen. Diese Ansicht ist wiederum ein Teilsystem der graphischen Benutzerschnittstelle des Cockpits.

Finale Verfeinerung

Der Fahrer öffnet über einen Funkschlüssel das Fahrzeug. Auf das Signal des Funkempfängers für die Entriegelung hin schaltet HyCop in die Standansicht. Auf der Fahrzeugstatus-Anzeige werden laufend Gewicht, Reifendruck, Batterieladung, Öl- und Benzinstand sowie Kontrolldaten der Funktionssensoren für Licht, Tür- und Gurtschlösser angezeigt.

Wenn HyCop über den Gewichtssensor eine Überladung erkennt, gibt es dies in einem Nachrichtenfenster auf dem Fahrer-Display bekannt.

14.1.2 Personalisieren

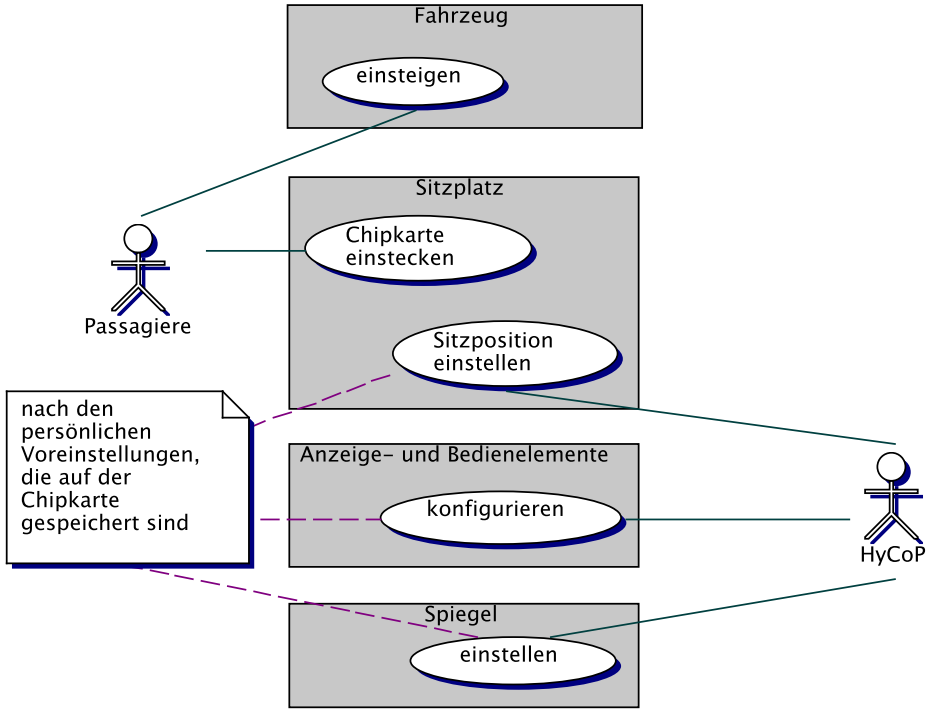

Abbildung 14.3: Anwendungsfalldiagramm Personalisieren

Bei Antritt einer Fahrt können die in das Fahrzeug einsteigenden Passagiere über ein noch zu bestimmendes Verfahren (z.B. eine Chipkarte) auf ihre persönliche Voreinstellungen zurückgreifen. HyCop, das diese Vorgaben berücksichtigen soll, muss dazu jeweils die bevorzugte Sitzposition einstellen, die Anzeige- und Bedienelemente im entsprechenden Monitor konfigurieren und die Spiegel ausrichten können. Die hier aufgeführten Anwendungen werden in Abbildung 14.3 veranschaulicht.

Finale Verfeinerung

Der Kartenleser an einem Sitzplatz erkennt das Einstecken einer Chipkarte. Die Einstellung der Sitzposition und die Ausrichtung der Spiegel werden in einem Nachrichtenfenster als Animation angezeigt.

Das Display am Sitplatz wird nach den Voreinstellungen konfiguriert, es zeigt z.B. ein Medienfenster mit Mediennavigation oder den Routenplaner an.

14.1.3 Starten

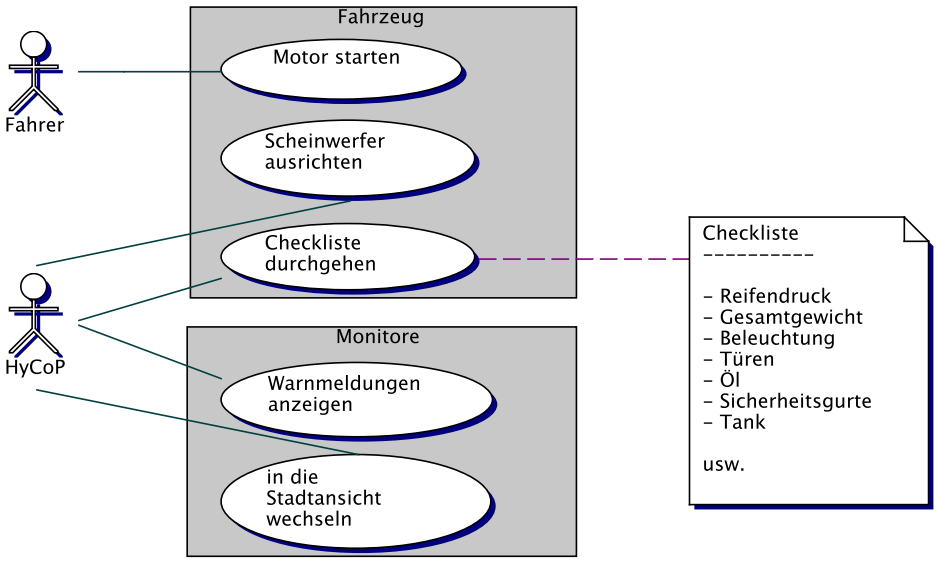

Abbildung 14.4: Anwendungsfalldiagramm Starten

Beim Starten des Motors, einer durch den Fahrer einzuleitenden Aktivität, muss das HyCop die Fahrbereitschaft des Fahrzeugs sicherstellen. Dieser Vorgang umfasst mehrere Aktivitäten, die sowohl das Fahrzeug als auch die Monitore der Passagiere betreffen. So müssen z.B. je nach Zuladung und Gewichtsverteilung die Scheinwerfer neu ausgerichtet werden. Weiterhin soll HyCop eine Checkliste durchgehen, die wichtige Fahrzeugdaten wie Reifendruck, Gesamtgewicht, Beleuchtung, Ölstand und Kraftstoffreserven, aber auch den Status von Tür- und Gurtschlössern umfasst. Für den Fall, dass hierbei ein kritischer Zustand erkannt wird, können entsprechende Warnmeldungen auf den Monitoren angezeigt werden. Andernfalls kann das Cockpit auf dem Fahrer-Display von der Stand- in die Stadtansicht umschalten. Eine Übersicht über die hier beschriebenen Aktivitäten liefert Abbildung 14.4.

Finale Verfeinerung

Nach den Daten der Gewichtssensoren stellt HyCop die Scheinwerfer ein, was in einem Nachrichtenfenster verfolgt werden kann.

Das Durchgehen der Checkliste betrifft nahezu die gesamte Fahrzeugsensorik. Es wird in einem Nachrichtenfenster angezeigt. Warnmeldungen werden in Pop-Up-Dialogen auf den Displays angezeigt.

Beim Umschalten in die Stadtansicht werden auf der Fahrzeugstatus-Anzeige die Anzeigen für Gewicht sowie Tür- und Gurtschloss-Kontrolldaten durch Tacho, Drehzahlmesser und Temperatur-Anzeige ersetzt.

14.2 Navigation

Autoren: *Yue Zhang, Oliver Szymanski, Leonore Dietrich*

Dieser Abschnitt beschäftigt sich mit Anwendungsfällen, die mit dem Navigationssystem und dessen Bedienung zusammen hängen.

14.2.1 Navigationssystem

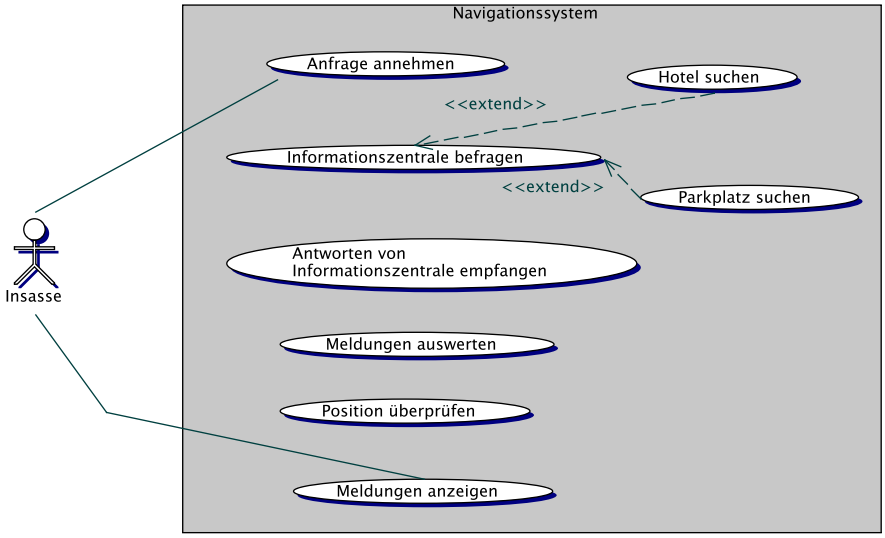

Abbildung 14.5: Anwendungsfalldiagramm Navigationssystem

Das Navigationssystem nimmt Anfragen vom Insassen entgegen (vgl. Abbildung 14.5). Es befragt die Informationszentrale z.B. um Rastplätze oder Hotels zu finden. Es empfängt Signale von der Informationszentrale, z.B. Staumeldungen oder Antworten auf gestellte Anfragen, und wertet die Signale aus. Das Navigationssystem überprüft die Position, z.B. um evtl. Abweichungen von der aktuellen Route zu bemerken. Darüber hinaus sendet es aktuelle Meldungen, z.B. Ergebnisse auf Anfrage oder Routeninformationen, an den Insassen.

Finale Verfeinerung

Der Benutzer gibt die Eingaben in die Suchmaske ein. Das HyCop übermittelt die Suchanfrage an die Informationszentrale. Die Informationszentrale wertet die Suchanfrage aus. Sie liefert die Ergebnisse an das HyCop zurück. Das HyCop zeigt eine Ergebnisliste im Auswahlmenü an. Das Navigationssystem empfängt aktuelle Staumeldungen von der Informationszentrale. Es vergleicht diese Daten mit der Route und benachrichtigt ggf. das HyCop. Das Navigationssystem empfängt Daten vom Positionssensor. Es vergleicht die Daten mit der Route und berechnet die Route ggf. neu.

14.2.2 Anfrage an das Informationssystem

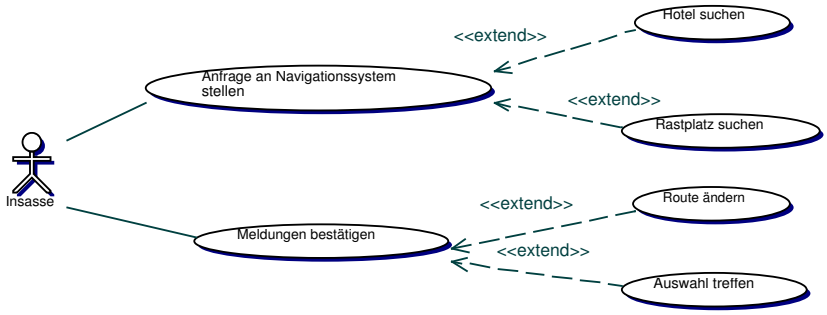

Abbildung 14.6: Anwendungsfalldiagramm Anfrage an das Informationssystem

Die Insassen können Anfragen an das Informationssystem stellen, z.B. ein Hotel oder Rastplätze suchen. Die Insassen können Meldungen bestätigen oder mögliche Auswahlen treffen (vgl. Abbildung 14.6).

Finale Verfeinerung

Um nach einer Übernachtungsmöglichkeit zu suchen, geben die Insassen die Suchkriterien in die entsprechende Suchmaske ein. Durch Auslösen des Submit-Buttons durch einen Insassen wird die Suchanfrage an das HyCop gesendet. Dieses zeigt die gewünschten Informationen im Auswahlmenü an. Die Insassen können aus der Hotelliste ein Objekt auswählen und so direkt buchen. Im Eingabe-Dialog wird nun ein Bezahldialog angezeigt. Für die Parkplatzsuche geben die Insassen ihre Anfrage in die Suchmaske für Parkplätze ein. Die Suchanfrage wird durch Auslösen des Submit-Buttons an das HyCop gesendet. Eine Liste mit passenden Rastplätzen wird im Auswahlmenü angezeigt. Der Insasse kann ein Objekt daraus wählen, das nun automatisch in die Route mit eingeplant wird. Alternativ kann zunächst eine Detailansicht zu den einzelnen Rastplätzen aufgerufen werden, die dann im Pop-Up-Dialog erscheint. Erhält das HyCop eine aktuelle Staumeldung vom Navigationssystem, so zeigt es einen Pop-Up-Dialog mit der Staumeldung und einer Alternativroute an. Der Insasse kann in dem Pop-Up-Dialog die Alternativroute annehmen oder die ursprüngliche Route beibehalten.

14.2.3 Route planen

Das Routenplaner-Bedienteil als Teilsystem des Cockpits kann als Steckmodul konzipiert sein, damit die Reiseplanung auch außerhalb des Fahrzeugs stattfinden kann. Aus dieser Forderung ergeben sich für den Fahrer die beiden Aktivitäten des Entnehmens und Einsetzens des Moduls in die Fahrzeugkonsole. Bei der Planung der Route, die eine Eingabe aller Zwischenstationen und des Zielorts umfasst, können sich alle Passagiere als Akteure beteiligen. Die Berechnung der Route erfolgt durch das Navigationssystem, also einen Teil des Akteurs HyCop. Dabei ergeben sich diverse Nebenaktivitäten. Zum einen müssen Verkehrsmeldungen, z.B. Hinwei-

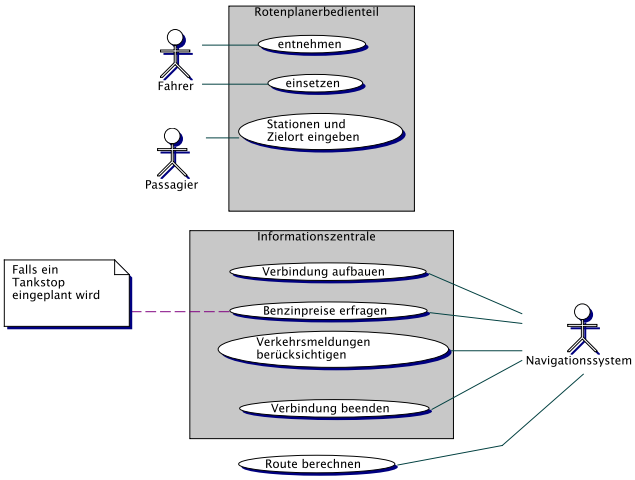

Abbildung 14.7: Anwendungsfalldiagramm Route planen

se auf Staus oder Sperrungen, berücksichtigt werden, die von einer Informationszentrale zur Verfügung gestellt werden. Zum anderen sollten die Benzinpreise an relevanten Tankstellen erfragt werden, sobald ein Tankstopp eingeplant wird. Dies soll über eine Internetverbindung zu der Informationszentrale ermöglicht werden, die das Navigationssystem eigenständig aufbauen und beenden kann. Abbildung 14.7 zeigt diesen Anwendungsfall.

14.3 Unterstützung des Fahrers

Autoren: *Ulf Schellbach*
Rafael Hosenberg
Oliver Szymanski
Yue Zhang
Leonore Dietrich

Das hypermediale Cockpit kann den Fahrer beim Manövrieren des Fahrzeugs in schwierigen Situationen unterstützen, z.B. beim Ein- und Ausparken. Dazu verfügt das Fahrzeug über Abstandssensoren und Kameras, die verschiedene Außenansichten zur Verfügung stellen.

14.3.1 Parkhilfe

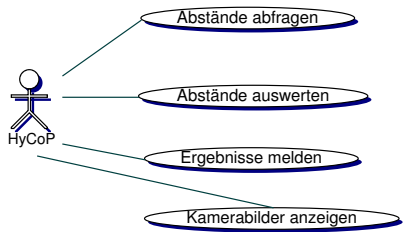

Abbildung 14.8: Anwendungsfalldiagramm Parkhilfe

Das HyCop wertet die gemessenen Abstände aus, zeigt die Ergebnisse dem Benutzer an und stellt benötigte Kamerabilder zur Verfügung. (vgl. Abbildung 14.8)

Finale Verfeinerung

Das Nachrichtenfenster zeigt die Daten der Abstandsmesser sowie eventuelle Kollisionsgefahr an. Auf letztere wird darüber hinaus akustisch hingewiesen. Die Bilder der Kameras werden in Kamerafenstern angezeigt.

14.3.2 Hindernis erkennen

Über die Sensorik des Fahrzeugs ist das HyCop dazu in der Lage, Hindernisse zu erkennen; daraus resultierende Hinderniswarnungen können von HyCop auf dem Monitor des Fahrers ausgegeben werden. Das Ausblenden einer solchen Warnmeldung kann sowohl durch das HyCop, z.B. wenn sich das Fahrzeug wieder von einem Hinderniss entfernt hat, als auch durch den Fahrer veranlasst werden (vgl. Abbildung 14.9).

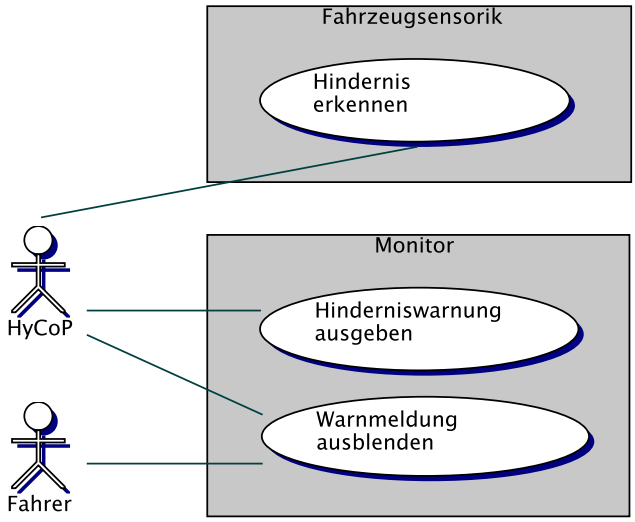

Abbildung 14.9: Anwendungsfalldiagramm Hindernis erkennen

Finale Verfeinerung

Auch hier liefern die Abstandsmesser Daten, die laufend in einem Nachrichtenfenster (grafisch) angezeigt werden. Auf kritische Abstände wird in einem Pop-Up-Dialog hingewiesen. Bilder der Kameras werden in Kamerafenstern angezeigt.

14.3.3 Lichtverhältnisse anpassen

Dieser Abschnitt beschäftigt sich mit der Anpassung an die aktuellen Lichtverhältnisse außerhalb des Fahrzeugs.

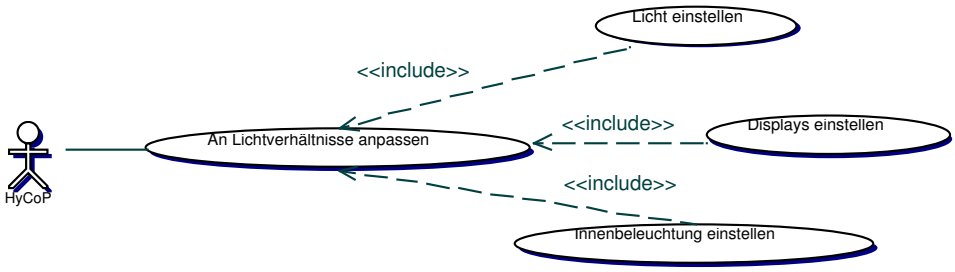

Abbildung 14.10: Anwendungsfalldiagramm Lichtverhältnisse anpassen

Das HyCop passt die Fahrzeugkomponenten, z.B. Licht, Innenbeleuchtung, Displays, den ak-

tuellen Lichtverhältnissen sowie den persönlichen Einstellungen der Insassen an. Abbildung 14.10 zeigt diesen Anwendungsfall.

Finale Verfeinerung

Die Beleuchtung wird von HyCop an die vom Helligkeitssensor gelieferten Werte angepasst. Dabei werden die persönlichen Einstellungen der Insassen mit berücksichtigt. Alternativ können die Insassen die Einstellungen der Beleuchtung und der Displays an ihren Sitzplätzen manuell ändern. Der Fahrer kann darüber hinaus die Scheinwerfer ein- und ausschalten. Sollten hierdurch Sicherheitsgrenzen unterschritten werden, gibt HyCop eine entsprechende Warnmeldung im Nachrichtenfenster aus und weist zusätzlich akustisch darauf hin.

14.4 Tanken

Autoren: *Ulf Schellbach, Rafael Hosenberg*

Im folgenden werden Szenarien beschrieben, die auftreten können, sobald entweder der Fahrer entscheidet eine Tankstelle anzusteuern oder das HyCop aufgrund knapper Benzinreserven das Aufsuchen einer Tankstelle empfiehlt. Es wird der gesamte Zeitraum bis zur Auffahrt auf die Autobahn nach Beendigung des Tankens berücksichtigt.

14.4.1 Beteiligte Akteure

An den folgenden Szenarien sind der Fahrer, die Bedienstation einer Tankstelle, das HyCop und das Navigationssystem beteiligt.

14.4.2 Fahrt durch die Stadt zur Tankstelle

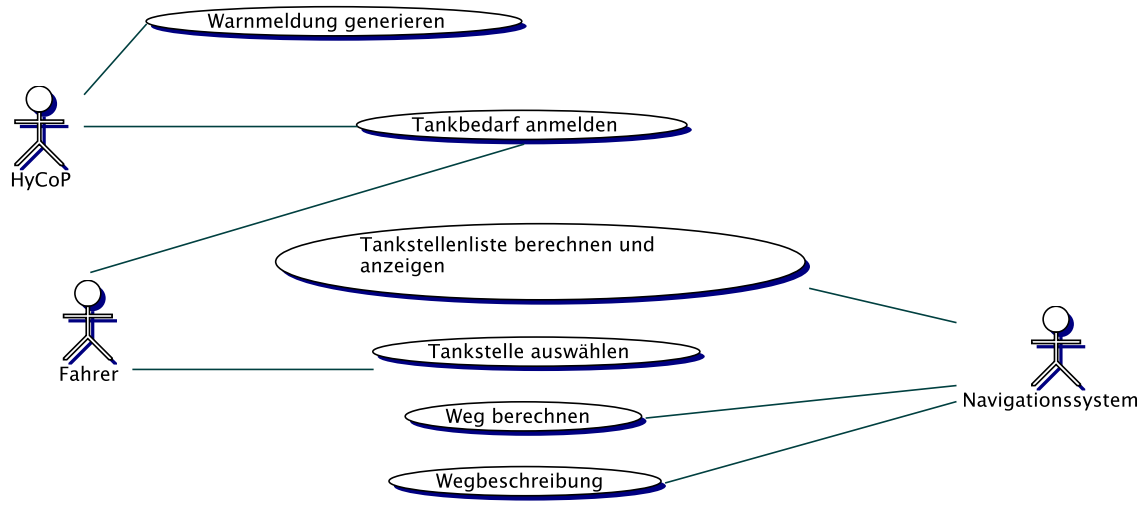

Abbildung 14.11: Anwendungsfalldiagramm Fahrt durch die Stadt zur Tankstelle

Wenn das HyCop über Sensoren erfährt, dass der Tankinhalt knapp geworden ist, meldet es Tankbedarf an, was auch mit dem Generieren einer Warnmeldung verbunden ist. Der Fahrer kann nach Belieben jederzeit Tankbedarf anmelden. Daraufhin berechnet das Navigationssystem eine Tankstellenliste und lässt diese anzeigen. Der Fahrer kann dann eine Tankstelle auswählen, woraufhin das Navigationssystem unter Einbeziehung der ausgewählten Tankstelle die Reiseroute neu berechnet. Die gesamte Fahrt hindurch beschreibt das Navigationssystem dem Fahrer auf geeignete Weise den Weg. Abbildung 14.11 zeigt diesen Anwendungsfall.

Finale Verfeinerung

Der Tanksensor signalisiert dem HyCop, dass der Tankinhalt knapp geworden ist. Daraufhin generiert das HyCop eine Warnmeldung, die im Nachrichtenfenster angezeigt wird. Der Routenplaner berechnet eine Tankstellenliste und zeigt sie im Auswahlmenü an. Der Fahrer wählt eine Tankstelle aus. Der Routenplaner berechnet die Route neu und gibt sie aus.

Alternative: Der Routenplaner wird durch eine Eingabe des Fahrers in die Suchmaske zur Berechnung einer Tankstellenliste veranlasst und nicht durch ein Signal des Tanksensors.

14.4.3 Tanken

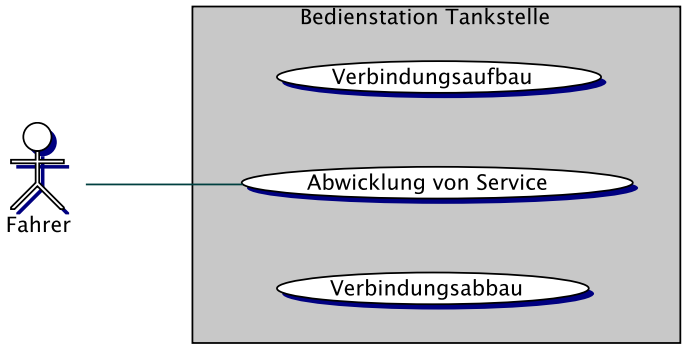

Abbildung 14.12: Anwendungsfalldiagramm Tanken

Wenn der Fahrer das Fahrzeug auf das Gelände einer Tankstelle lenkt, wird eine drahtlosen Verbindung zwischen dem HyCop und der Bedienstation der Tankstelle aufgebaut. An der Abwicklung des Service sind sowohl der Fahrer als auch das HyCop und die Bedienstation der Tankstelle beteiligt. Verlässt das Fahrzeug das Gelände der Tankstelle, wird die Verbindung zwischen dem HyCop und der Bedienstation der Tankstelle automatisch abgebaut. Abbildung 14.12 zeigt diesen Anwendungsfall.

Finale Verfeinerung

1. Der Positionssensor löst den Aufbau einer drahtlosen Verbindung zwischen dem HyCop und der Bedienstation der Tankstelle aus.
2. Abwicklung von Service.
3. Der Positionssensor löst den Abbau der drahtlosen Verbindung zwischen dem HyCop und der Bedienstation der Tankstelle aus.

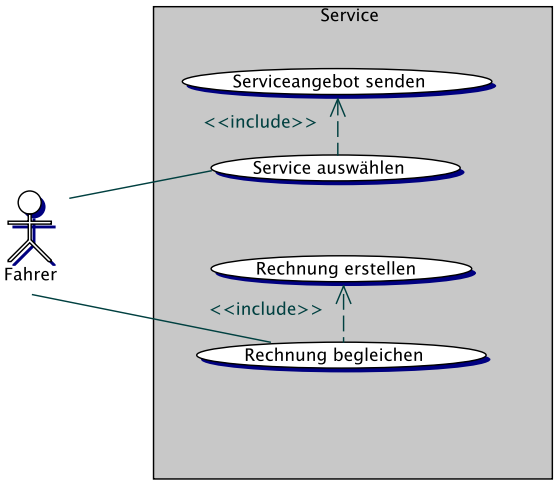

Abbildung 14.13: Anwendungsfalldiagramm Abwicklung von Service

14.4.4 Abwicklung von Service

Notwendige Vorbedingung für den Service ist eine bestehende Verbindung zwischen Cockpit und Bedienstation der Tankstelle. Über die bestehende Verbindung sendet die Bedienstation der Tankstelle ihr Serviceangebot an das HyCop. Der Fahrer kann nun einen Service auswählen, der daraufhin von der Bedienstation der Tankstelle erbracht wird. Die Bedienstation der Tankstelle erstellt zu allen erbrachten Dienstleistungen eine Rechnung. Der Fahrer begleicht die Rechnung auf elektronischem Wege. Als verschiedene Serviceleistungen der Tankstelle sind denkbar: Tanken, Verkauf von Konsumgütern, Reinigung des Autos, Wartung des Autos, Download von verschiedenartigen Daten. Abbildung 14.13 zeigt diesen Anwendungsfall.

Finale Verfeinerung

Das HyCop zeigt das Serviceangebot der Tankstelle im Auswahlmenü an. Der Fahrer wählt verschiedene Services aus. Zum begleichen der Rechnung erscheint im Nachrichtenfenster der Bezahldialog.

14.4.5 Tankstelle - Landstraße - Autobahn

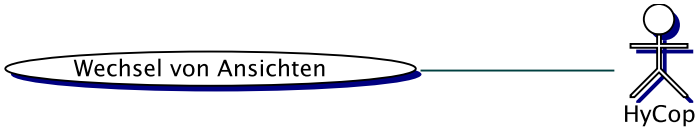

Abbildung 14.14: Anwendungsfalldiagramm Tankstelle - Landstraße - Autobahn

Auf der Fahrt von der Tankstelle über die Landstraße zur Autobahn veranlaßt das HyCop einen Wechsel der Ansichten. Es sind verschiedene Ansichten für die Fahrt in der Stadt, auf

der Landstraße oder auf der Autobahn denkbar. Der Wechsel der Ansichten kann von einer Kombination aus Informationen über die gefahrene Geschwindigkeit und über eine Ortung abhängig sein. Abbildung 14.14 zeigt diesen Anwendungsfall. Die genaue Ausprägung der einzelnen Ansichten wird noch zu erarbeiten sein.

Finale Verfeinerung

Der Positionssensor und der Geschwindigkeitsmesser lösen den Wechsel von Ansichten aus.

14.5 Fahrzeugwartung

Autoren: *Tobias Wolf, Evgenij Golkov*

Dieser Abschnitt beschäftigt sich mit Anwendungsfällen, die der Wartung oder Reparatur des Fahrzeugs dienen.

14.5.1 Panne und Werkstatt

Im folgenden wird das Szenario *Panne und Werkstattbesuch* beschrieben. Folgende Akteure sind an diesem Szenario beteiligt:

- Fahrzeug-Diagnosetool
- Werkstatt-System
- HyCop
- mobiles Diagnosetool

Im Folgenden werden die im Diagramm *Panne und Werkstatt* (siehe Abbildung 14.15) dargestellten Anwendungsfälle beschrieben.

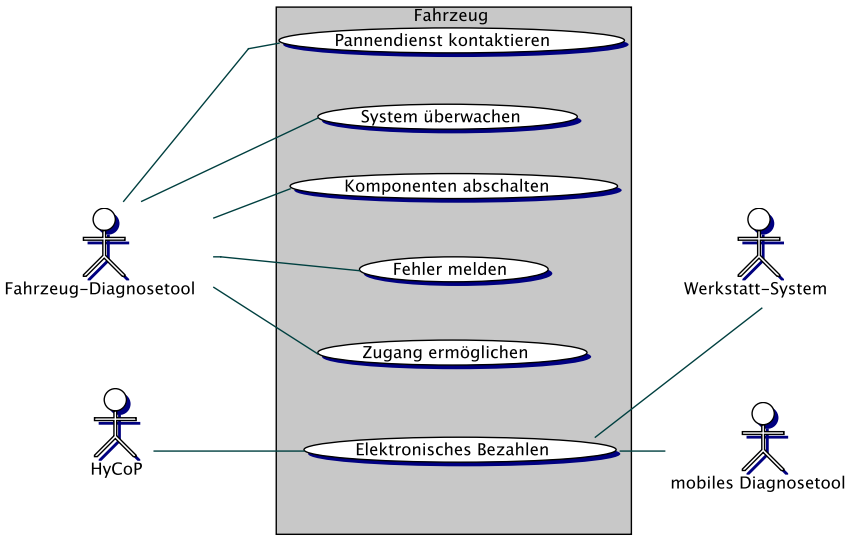

Abbildung 14.15: Anwendungsfalldiagramm Panne und Werkstatt

Das Fahrzeug-Diagnosetool überwacht laufend das System und leitet bei Bedarf entsprechende Aktionen ein. Eine detailliertere Beschreibung folgt im Abschnitt 14.5.3 *System überwachen*. Hat das Fahrzeug-Diagnosetool eine Panne erkannt, so kann es eine Verbindung zu einem Pannendienst herstellen. Dies wird im Abschnitt 14.5.2 *Pannendienst kontaktieren* näher erläutert. Bei Bedarf können Komponenten von dem Fahrzeug-Diagnosetool abgeschaltet werden. Als

Beispiel kann man hier das Abstellen des Motors, falls der Öldruck zu niedrig oder die Kühlwassertemperatur zu hoch ist, nennen. Ermittelte Fehler werden angezeigt bzw. an andere Tools weitergeleitet. Befindet sich das Fahrzeug schließlich in einer Werkstatt, so regelt das Fahrzeug-Diagnosetool den Zugriff von externen Tools und Systemen. Soll schließlich bezahlt werden, so findet ein elektronischer Bezahlvorgang zwischen dem Fahrzeug und dem Werkstattsystem bzw. dem mobilen Diagnosetool statt. Dieser Anwendungsfall wird im Abschnitt 14.7.3 „Elektronisches Bezahlen“ näher erläutert.

Finale Verfeinerung

Bei der Abfrage systemkritischer Sensoren bzw. durch Meldung eines systemkritischen Sensors leitet das Fahrzeug-Diagnosetool die Abschaltung von Komponenten ein. Dies wird dem Fahrer in einem Nachrichtenfenster sowie akustisch mitgeteilt. Tritt ein Fehler auf, wird in einem Nachrichtenfenster der Typ und eine Beschreibung angezeigt. Will ein externes System Zugang zum Fahrzeug-Diagnosetool haben, wird dies in einem Pop-Up-Dialog angezeigt. Der Fahrer kann die Verbindung ablehnen oder ihr zustimmen. Bleibt eine Reaktion des Fahrers aus, weil z.B. der Fahrer sich nicht im Wagen befindet, so baut das Fahrzeug-Diagnosetool nach einer erfolgreichen Authentifizierung selbständig eine Verbindung auf.

14.5.2 Pannendienst kontaktieren

Hier werden die zum Unterszenario *Pannendienst kontaktieren* gehörenden Anwendungsfälle beschrieben. Abbildung 14.16 bezieht sich auf dieses Szenario.

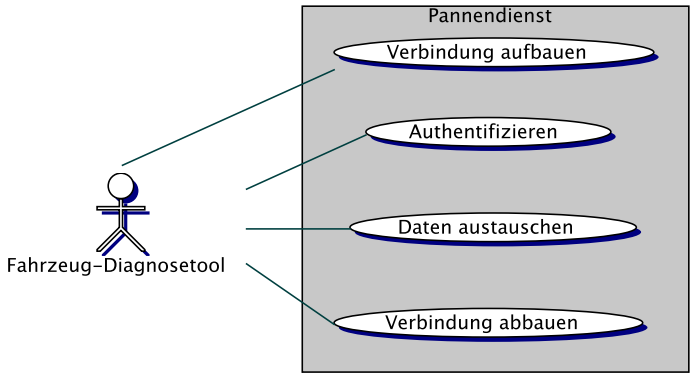

Abbildung 14.16: Anwendungsfalldiagramm Pannendienst kontaktieren

Das Fahrzeug-Diagnosetool kann eine Verbindung zu einem Pannendienst aufbauen. Dazu überprüfen sich Fahrzeug-Diagnosetool und das Werkstattsystem gegenseitig, ob sie zu einer Verbindung berechtigt sind. Anschließend werden zwischen dem Fahrzeug-Diagnosetool und dem Werkstatt-System eine Reihe von Daten ausgetauscht. Dazu gehören unter anderem der Standort des Fahrzeugs und wichtige Diagnosedaten über die Art der Panne. Nach der erfolgreichen Übertragung der benötigten Daten wird die Verbindung abgebaut.

Finale Verfeinerung

Im Falle einer Panne wird dem Fahrer durch ein Auswahlmenü die Wahl zwischen verschiedenen Pannendiensten angeboten. Sobald der Fahrer seine Wahl getroffen hat, wird eine Verbindung zu diesem Pannendienst aufgebaut. Der Fahrer wird durch ein Nachrichtenfenster über den Erfolg bzw. Misserfolg der Authentifizierung informiert. In einem Nachrichtenfenster wird der Fortschritt der Datenübertragung angezeigt. Ist die Datenübertragung abgeschlossen und die Verbindung abgebaut, so wird dies in einem Nachrichtenfenster angezeigt.

14.5.3 System überwachen

Hier werden die zum Unterszenario *System überwachen* gehörenden Anwendungsfälle beschrieben. Das Fahrzeug-Diagnosetool überwacht die Systeme des Fahrzeugs und stößt beim Unter- bzw. Überschreiten von festgelegten Werten bestimmte Aktionen an. Abbildung 14.17 bezieht sich auf dieses Szenario.

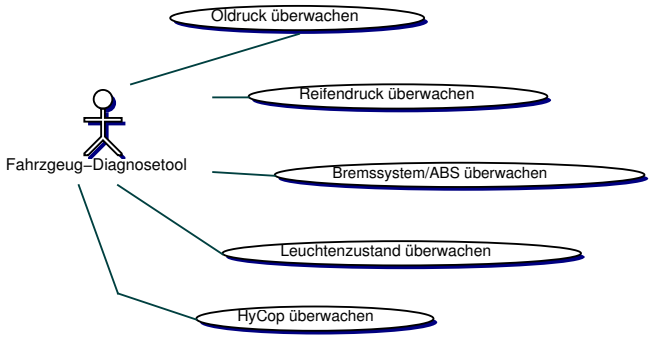

Abbildung 14.17: Anwendungsfalldiagramm System überwachen

Das Fahrzeug-Diagnosetool überwacht fortlaufend wichtige Systeme des Fahrzeugs. Sinkt der Öldruck unter einen bestimmten Wert, so wird eine Warnmeldung angezeigt. Sollte allerdings ein plötzlicher Abfall des Öldrucks registriert werden, so kann der Motor abgeschaltet werden. Ein zu geringer Reifendruck wird mit einem Hinweis auf den entsprechenden Reifen angezeigt. Bei einem geplatzten Reifen wird dies an das HyCop gemeldet, damit andere Systeme den Fahrer unterstützen, um das Fahrzeug unter Kontrolle zu halten. Der Füllstand der Bremsflüssigkeit sowie die Funktionstüchtigkeit des ABS-Systems wird kontrolliert. Funktioniert eines der Systeme nicht korrekt, wird dies dem Fahrer angezeigt. Vor jedem Start und bei der Beendigung einer Fahrt werden die Scheinwerfer überprüft. Defekte Schweinwerfer werden dem Fahrer gemeldet. Alle relevanten HyCop-Komponenten werden überwacht. Eventuelles Fehlverhalten wird über das Display sowie akustisch gemeldet.

Finale Verfeinerung

Der Öldrucksensor wird in regelmässigen Abständen abgefragt, um die Öldruckanzeige zu aktualisieren. Registriert der Sensor kritische Werte, so kann er selbständig eine Nachricht schicken. Die Reifendrucksensoren schicken bei einem kritischen Druckabfall eine Nachricht.

Der Bremsflüssigkeitssensor wird regelmäßig abgefragt und es wird bei einem kritischen Wert eine Nachricht geschickt. Die Sensoren des ABS-Systems werden regelmäßig abgefragt, um die Funktionalität des Systems zu überprüfen. Fällt diese Überprüfung negativ aus, wird eine Nachricht geschickt. Die Scheinwerferzustandssensoren senden eine Nachricht, wenn eine Lampe ausfällt. Vom Fahrzeug-Diagnosetool werden die HyCop-Komponenten regelmäßig überprüft. Meldet sich eine Komponente nicht ordnungsgemäß zurück, wird in einem Nachrichtenfenster eine Meldung angezeigt.

14.6 Unterhaltung

Autoren: *Stefan Borggraefe, Bastian Krol*

Dieser Abschnitt beschreibt die Anwendungsfälle des Unterhaltungsbereichs. Dieser Bereich umfasst alle Dienstleistungen, welche den Insassen zum Zeitvertreib angeboten werden, z.B. Musik hören oder Video gucken. Bei diesen Anwendungsfällen ist zu beachten, dass der Fahrer des Wagens eine Sonderrolle einnimmt, da er sich auf den Straßenverkehr konzentrieren muss. Daher stehen einige Unterhaltungsangebote für ihn während der Fahrt nicht oder nur eingeschränkt zur Verfügung. Wenn der Motor abgeschaltet ist, entfällt diese Unterscheidung zwischen Fahrer und Mitfahrer natürlich.

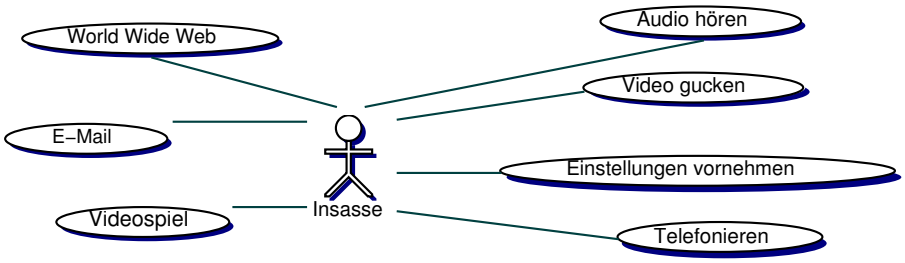

Abbildung 14.18: Anwendungsfalldiagrammdiagramm Unterhaltung

Abbildung 14.18 zeigt das Anwendungsfalldiagramm für den Bereich Unterhaltung. Zwischen den einzelnen Medien kann folgendermaßen umgeschaltet werden:

- Durch das Einlegen eines neuen Mediums wird dieses Medium angespielt.
- Durch drücken eines Knopfes, mit dem man eine Liste der verfügbaren Medien anzeigen lassen kann. Aus dieser Liste kann man dann das gewünschte Medium per Touchscreen auswählen.
- Beim Medium Videospiel kann ein Insasse einen anderen zum Mitspielen auffordern. Geht er auf diese Aufforderung ein, wird zu dem Videospiel gewechselt.

14.6.1 Audio

Der Audiobereich umfasst alle Möglichkeiten, während der Fahrt Musik oder Ähnliches (z.B. Nachrichten im Radio) zu hören. Die Anwendungsfälle des Audiobereichs unterscheiden sich insofern von den anderen Anwendungsfällen der anderen Bereichen, als dass irgendeine Art der Audiowiedergabe typischerweise ständig aktiv ist und man oft nur zwischen verschiedenen Medien (CD, Radio) wechselt. Im Gegensatz dazu werden Unterhaltungsangebote aus dem Bereich Video (siehe Abschnitt 14.6.2) meistens während der Fahrt gezielt gestartet und wieder beendet. Abbildung 14.19 zeigt die Anwendungsfälle für die Audiomedien.

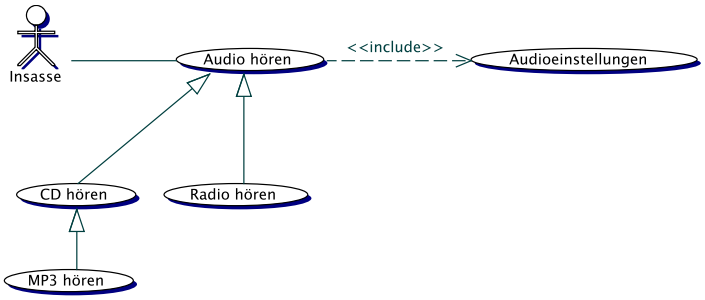

Abbildung 14.19: Anwendungsfalldiagramm Audio

Audio hören

1. Ein Insasse startet das Audiomedium. Auf dem Display erscheint der im Moment gespielte Titel.
2. Der Ton erklingt über die Lautsprecher des Fahrzeugs.

Alternative: Die einzelnen Insassen können ein Audiomedium zur Wiedergabe über ihren Kopfhörer wählen.

Finale Verfeinerung

1. Ein Insasse startet das Audiomedium. Dies kann entweder durch das Einlegen eines neuen Mediums erfolgen oder durch das Anwählen eines bereits verfügbaren Mediums.
2. Auf dem Display erscheinen in Anzeigebereich „Medieninformationen“ Daten über das derzeit spielenden Medium, die sich je nach Medium unterscheiden. Typische Beispiele sind der Titel des derzeit spielenden Musikstückes und die Laufzeit des Titels.
3. Der Ton erklingt über die Lautsprecher des Fahrzeugs. Von bisher ein anderes Medium abgespielt wurde, wird dieses unterbrochen, da immer nur ein Medium zur Audio-Wiedergabe ausgewählt sein darf.
4. Die Wiedergabe des Mediums kann durch das Betätigen eines Stopp-Knopfes, dem Auswerfen des Mediums, durch Betätigen eines Auswurf-Knopfes oder durch die Wahl eines anderen Mediums beendet werden.

Alternative: Die einzelnen Insassen können ein eigenes Audiomedium zur Wiedergabe über ihren Kopfhörer wählen. Auch dies kann durch das Einlegen eines neuen Mediums in das Laufwerk an ihrem Platz oder durch das Anwählen eines bereits verfügbaren Mediums geschehen.

CD hören

1. Ein Insasse legt eine CD ein.
2. Der CD-Spieler spielt vom ersten Titel an die CD ab.

Final Verfeinerung

1. Ein Insasse legt eine CD ein oder wählt eine bereits eingelegte CD zur Wiedergabe aus.
2. Der CD-Spieler spielt vom ersten Titel an die CD ab. Gegebenenfalls wird das bisher gespielte Medium dazu unterbrochen.
3. Während des Abspielens einer Audio-CD werden im Display-Bereich „Medieninformationen“ in der Grundeinstellung die aktuelle Track-Nummer und die Anzahl der Tracks, der Name des Interpreten und des Stückes und die derzeitige Position im Track dargestellt. Durch Berühren dieser Anzeigeelemente können weitere Informationen eingeblendet werden. Beispielsweise kann die Position im Track zwichen noch verbleibender und als bereits abgespielter Zeitspanne umgeschaltet werden.

MP3 hören

1. Ein Insasse kann sich einen Einzeltitel aussuchen oder eine Playlist zusammenstellen.
2. MP3 wird abgespielt.

Finale Verfeinerung

1. Ein Insasse kann sich einen Einzeltitel aussuchen oder eine Playlist zusammenstellen. Wenn eine Medium mit MP3-Dateien eingelegt wird, wird ebenfalls die MP3-Wiedergabe gestartet.
2. Die erste MP3-Datei wird abgespielt. Gegebenenfalls wird das bisher gespielte Medium dazu unterbrochen.
3. Während des Abspielens von MP3-Dateien werden im Display-Bereich „Medieninformationen“ in der Grundeinstellung der Name des Interpreten und des Stückes und die derzeitige Position im Track dargestellt. Durch Berühren dieser Anzeigeelemente können weitere Informationen eingeblendet werden. Beispielsweise kann die Position im Track zwichen noch verbleibender und als bereits abgespielter Zeitspanne umgeschaltet werden.

Radio hören

1. Ein Insasse schaltet die Audiowiedergabe auf Radio.
2. Der Insasse wählt einen Radiosender aus.
3. Das Radio spielt den Sender.

Finale Verfeinerung

1. Ein Insasse schaltet die Audiowiedergabe auf Radio.
2. Der Insasse wählt einen Radiosender aus. Der zuletzt gespielte Sender wird automatisch angespielt und beibehalten, wenn der Insasse hier nichts anderes wählt.
3. Das Radio spielt den Sender.
4. Das Wechseln des Senders kann durch erneuten Aufruf der Liste der verfügbaren Medien geschehen.

Alternative: Durch betätigen der Zapping-Taste kann ein Sendersuchlauf durchgeführt werden. Wird ein Sender mit ausreichender Stärke gefunden, wird dieser zenh Sekunden lang angespielt. Wird während des Anspielens eines Senders ein weiteres Mal die Zapping-Taste gedrückt, wird der Sender beibehalten.

14.6.2 Video

Der Bereich Video umfasst alle visuellen Unterhaltungsmedien, die im Fahrzeug vorhanden sind. Dazu gehören natürlich das Fernsehen und das Abspielen von Videos, aber auch Videospiele. Abbildung 14.20 zeigt das Anwendungsfall Diagramm für die visuellen Medien.

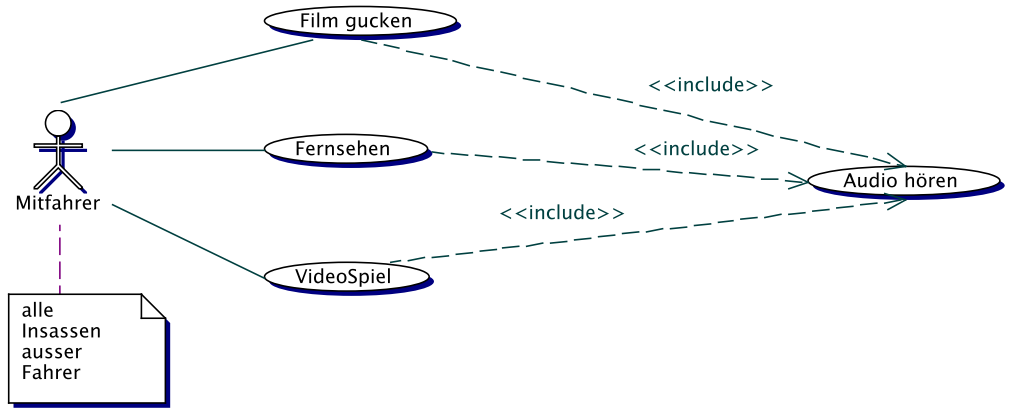

Abbildung 14.20: Anwendungsfalldiagramm Video

Film gucken

1. Ein Mitfahrer legt ein Medium(z.B. DVD) ein.
2. Die Wiedergabe wird auf den gewünschten Displays gestartet. Eine Wiedergabe auf dem Fahrerdisplay ist während der Fahrt nicht möglich.
3. Die Audiowiedergabe erfolgt über Kopfhörer.
4. Die Wiedergabe wird beendet.

Finale Verfeinerung

Das Medienfenster auf dem Bildschirm zeigt den Film. Durch Berühren des Bildschirms wird das Medienmenü eingeblendet, dass in diesem Fall aus Mediennavigation, Audioeinstellungen und Videoeinstellungen besteht. Durch Auswählen eines dieser drei Menüpunkte wird zum entsprechenden Menü gewechselt.

Alternative - Motor ausgeschaltet:

- Die visuelle Wiedergabe ist auch auf dem Fahrerdisplay möglich.

- Die Audiowiedergabe ist auch über die Lautsprecheranlage möglich.

Fernsehen

1. Ein Mitfahrer startet auf seinem Display die Fernsehwiedergabe.
2. Der Mitfahrer wählt einen Fernsehsender.
3. Auf Wunsch ist auch die Wiedergabe auf mehreren verschiedenen Displays gleichzeitig möglich. Eine Wiedergabe auf dem Fahrerdisplay ist während der Fahrt nicht möglich.
4. Die Audiowiedergabe erfolgt über Kopfhörer.
5. Die Wiedergabe wird beendet.

Finale Verfeinerung

Das Medienfenster auf dem Bildschirm zeigt die Fernsehwiedergabe. Die Mediennavigation ist eingeblendet. Durch Berühren des Bildschirms wird das Medienmenü eingeblendet. Diese besteht in diesem Fall aus Audioeinstellungen und Videoeinstellungen. Durch Auswählen eines dieser Menüpunkte wird zum entsprechenden Menü gewechselt.

paragraphAlternative - Motor aus:

- Die visuelle Wiedergabe ist auch auf dem Fahrerdisplay möglich.
- Die Audiowiedergabe ist auch über Lautsprecher möglich.

Videospiel

Videospiele benötigen neben dem Display evtl. auch noch besondere Eingabegeräte wie Gamepads oder Ähnliches. Eine Steuerung über die berührungssensitiven Bildschirme ist zwar vorstellbar, aber je nach Spiel nicht unbedingt ideal.

1. Ein Mitfahrer startet ein Spiel.
2. Er sucht sich Mitspieler aus. Das Spielen mit dem Fahrer ist während der Fahrt verboten.
3. Die Mitspieler können dieses Angebot ablehnen oder akzeptieren.
4. Das Display des jeweiligen Spielers zeigt seine Spielperspektive an, der Ton kann nur über Kopfhörer wiedergegeben werden.

Finale Verfeinerung

Das Medienfenster auf dem Bildschirm des Spielers zeigt die Spielansicht. Der Spieler kann über die Mitspielerauswahl Gegner/Mitspieler bestimmen. Diese Mitspielerauswahl kann z.B. eine schematische Darstellung des Wagens aus der Vogelperspektive sein.

Alternative - Motor aus:

- Auch der Fahrer kann an Videospielen teilnehmen oder ein Spiel starten.
- Audiowiedergabe auch über Lautsprecher möglich.

14.6.3 Einstellungen

Abbildung 14.21 zeigt das Anwendungsfalldiagramm für die Einstellungen, die den Unterhaltungsbereich betreffen.

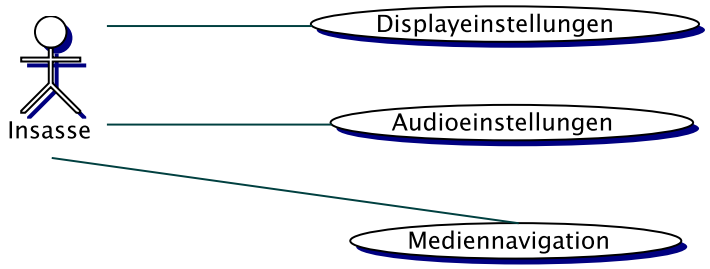

Abbildung 14.21: Anwendungsfalldiagramm Einstellungen

Displayeinstellungen

Dieser Anwendungsfall dient dazu, die Darstellung auf dem Display anzupassen. Diese Einstellungen sind Teil der Personalisierungsinformationen. Die Displayeinstellungen enthalten folgende Komponenten:

1. Kontrastregler
2. Helligkeitsregler

Finale Verfeinerung

Während der Videowiedergabe aktiviert der Insasse die Displayeinstellungen. Das Videoeinstellungsmenü erscheint auf dem Bildschirm. Der Insasse berührt einen Regler und verstellt den Wert. Der Insasse aktiviert den „Bestätigen“-Knopf um die Einstellungen zu übernehmen.

Audioeinstellungen

Dieser Anwendungsfall gehört zur Audiowiedergabe. Diese Einstellungen sind Teil der Personalisierungsinformationen. Die Audioeinstellungen für die Lautsprecherwiedergabe enthalten folgende Komponenten:

- Lautstärkeregler

- Regler für Höhen/Tiefen
- Balanceregler (Lautstärkeverhältnis links/rechts)
- Regler Lautstärkeverhältnis vorne/hinten

Die Audioeinstellungen für die Kopfhörerwiedergabe enthalten nur die Lautstärke und Höhen/Tiefen.

Finale Verfeinerungen

Während der Audiowiedergabe aktiviert der Insasse die Audioeinstellungen. Dies können entweder die Audioeinstellungen für die Lautsprecher oder für seine Kopfhörer sein. Das Audioeinstellungsmenü erscheint auf dem Bildschirm. Der Insasse berührt einen Regler und verstellt den Wert. Der Insasse aktiviert den „Bestätigen"-Knopf um die Einstellungen zu übernehmen.

Mediennavigation

Dieser Anwendungsfall gehört zur Medienwiedergabe und dient dazu, die Wiedergabe zu steuern. Die Art der Navigation ist vom Medium abhängig und gestaltet sich je nach Medium anders. Wir listen die möglichen Navigationsaktionen und die Medien, bei deren Wiedergabe diese Aktionen erlaubt sind, auf:

- Track vor/zurück (CD, MP3, DVD)
- Vor- und Rücklauf (CD, MP3, DVD)
- Playlist programmieren (CD, MP3)
- Festplatte über Verzeichnisbaum durchstöbern (MP3)
- Sender wählen (Radio, Fernseher)
- Audio aufnehmen (Radio)
- Pause/Fortsetzen (CD, MP3, DVD, Videospiel)
- Medium auswerfen (CD, DVD)
- Zufallswiedergabe (CD, MP3)

Finale Verfeinerungen

Während der Medienwiedergabe aktiviert ein Insasse die Mediennavigation. Das Mediennavigationsfenster wird eingeblendet. Für jede der oben beschriebenen Aktionen steht ein Bedienelement zur Verfügung, falls die entsprechende Aktion im Moment erlaubt ist. Durch Berühren des jeweiligen Bedienelements wird diese Aktion ausgelöst.

14.6.4 Sonstiges

Die beim Start des Wagens erfolgte Personalisierung bezieht sich natürlich auch auf die Unterhaltungseinstellungen. Beispiele hierfür wären der bevorzugte Radiosender oder eine MP3-

Playlist, die gesamten Audioeinstellungen (siehe Abschnitt 14.6.3) und die Displayeinstellungen (siehe Abschnitt 14.6.3).

14.7 Kommunikation

Autoren: *Stefan Borggraefe, Bastian Krol*

Der Bereich Kommunikation beschreibt die Kommunikation der Insassen mit der Außenwelt per Telefon, E-Mail und Internet.

14.7.1 Telefonieren

Abbildung 14.22 zeigt das Anwendungsfalldiagramm für den Bereich Telefonie.

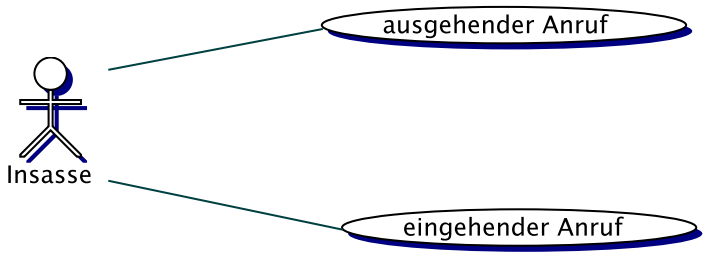

Abbildung 14.22: Anwendungsfalldiagramm Telefonie

Eingehender Anruf

1. Die Audiowiedergabe läuft.
2. Ein Anruf geht ein.
3. Die Audiowiedergabe wird ausgeblendet und evtl. pausiert.
4. Eine Meldung auf dem Display und ein akustisches Signal melden den Anruf (inkl. Telefonnummer und/oder Name des Anrufers).
5. Der Anruf wird von einem der Insassen akzeptiert.
6. Der Insasse führt das Gespräch und beendet es.
7. Die Audiowiedergabe wird wieder eingeblendet.

Alternative: Der Anruf kann abgelehnt werden. Die letzten drei Punkte entfallen dann.

Ausgehender Anruf

1. Die Nummer wird eingegeben oder aus dem Telefonbuch gewählt.
2. Nach dem Wählen wird die Audiowiedergabe ausgeblendet.

3. Das Gespräch wird geführt.
4. Nach dem Gespräch wird die Audiowiedergabe wieder eingeblendet.

Alternative: Es kommt keine Verbindung zustande. Nach dem versuchten Verbindungsaufbau wird dann eine Fehlermeldung ausgegeben und die Audiowiedergabe wieder eingeblendet.

14.7.2 E-Mail und Internet

Als weitere Anwendungsfälle können E-Mail zur Kommunikation und Internetzugriff zur Unterhaltung identifiziert werden. Einerseits ist die genaue Modellierung dieser beiden Anwendungsfälle kompliziert, andererseits ist ihr Ablauf aber auch hinlänglich bekannt. Da es der

PG im Kern nicht darum geht, einen Browser oder einen E-Mail-Client nachzubauen, wird auf die exakte Modellierung dieser beiden Anwendungsfälle hier verzichtet.

14.7.3 Elekronisches Bezahlen

Hier wird der im Diagramm *Elektronisches Bezahlen* (Abbildung 14.23) dargestellte Anwendungsfall beschrieben.

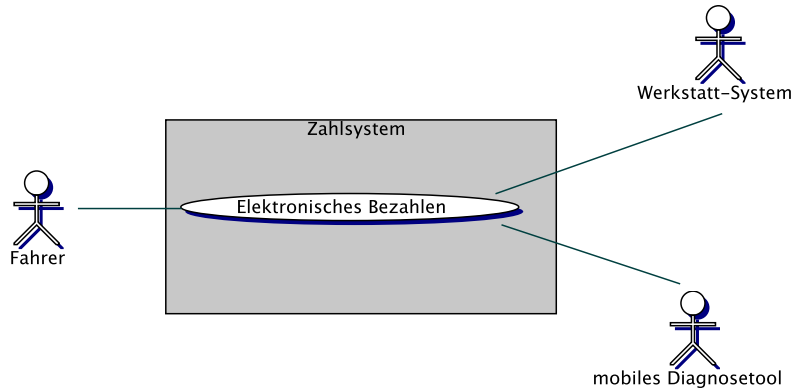

Abbildung 14.23: Anwendungsfalldiagramm Elektronisches Bezahlen

Reparaturen, Hotelbuchungen, Downloads und sonstige Leistungen werden über das Zahlsystem nach Bestätigung durch den Fahrer elektronisch bezahlt. Das Zahlsystem wird nur durch eine einfache Oberfläche mit der Schnittstelle zum Benutzer realisiert.

Finale Verfeinerung

Will der Fahrer elektronisch bezahlen, so ruft er das Zahlsystem auf. Bezahlanforderungen anderer Systeme werden entgegengenommen und in einem Pop-Up-Dialog dem Fahrer präsentiert. Dort kann er die Bezahlung akzeptieren oder ablehnen.

Alternative: Registriert das Zahlsystem eine Bezahlanforderung eines anderen Systems, so erscheint ein Pop-Up-Dialog. Dort kann der Fahrer die Bezahlung akzeptieren oder ablehnen.

Kapitel 15

Übersicht über Anzeigen, Bedienelemente und Sensoren

Autor: *Markus Niehammer*

Einleitung

In den Anwendungsfällen können eine Reihe von Anzeigen/Instrumenten, Bedienelementen und Sensoren identifiziert werden. Diese sollen hier als lose Aufzählung festgehalten werden, um einen Gesamtüberblick zu bekommen.

15.1 Anzeigen und Instrumente

Anzeigen sind als Meldungen oder Grafiken zur reinen Information der Insassen zu verstehen. Es besteht keine direkte Möglichkeit, auf den Status einer Anzeige Einfluss zu nehmen.

- Standard-Cockpit:
  - Tacho
  - Drehzahlmesser
  - Tankanzeige
  - Kilometerstand
  - Temperatur
  - Kontrollleuchten (z.B. für Licht und Fernlicht)
- Nachrichtenfenster
- Kamerafenster (Spiegel)
- Medieninformation (Titel, Interpret)
- Medienfenster - zeigt Videos, Fernsehen, Videospiel etc.

- Fahrzeugstatus-Anzeige
- Routen-Anzeige

15.2 Bedienelemente

Bedienelemente dienen zur Interaktion mit HyCop, d.h. um Einstellungen vorzunehmen oder Meldungen zu bestätigen.

- Pop-Up-Dialog (Nachrichtenfenster mit Buttons)
- Medienmenü Video
  - Mediennavigation
  - Videoeinstellungen
  - Audioeinstellungen
- Medienmenü Fernsehen
  - Videoeinstellungen
  - Audioeinstellungen
- Medienmenü Audio
  - Mediennavigation
  - Audioeinstellungen
- Mediennavigation:
  - Track vor
  - Track zurück
  - Vorlauf
  - Rücklauf
  - Sender wählen
  - Audio aufnehmen
  - Pause
  - Fortsetzen
  - Medium auswerfen
  - Zufallswiedergabe
- Mitspielerauswahl für das Videospiel
- Routenplaner-Bedienteil
- Eingabedialog (Hotels, Tankstellen,...)
- Diagnosemenüs
- Auswahlmenü

15.3 Sensoren

Sensoren versorgen HyCop mit Informationen und Daten, auf die die Insassen des Fahrzeugs keinen direkten Einfluss haben. Sensoren sind keine Bestandteile des Cockpits, sondern externe Bausteine.

- Funkempfänger (für die Ver- bzw. Entriegelung)
- Kartenleser (für die Personalisierung)
- Geschwindigkeitsmesser (Nachrichtenfenster mit Buttons)
- Tanksensor
- Temperatursensor
- Regensensor
- Gewichtssensoren
- Abstandsmesser
- Helligkeitssensor
- Kameras
- Funktionssensoren
- Positionssensor
- Staumelder

Kapitel 16

Zustandsübergangsdiagramme

Einleitung

Dieses Dokument beschreibt die Zustandsdiagramme des hypermedialen Cockpits. Zunächst werden einige wichtige Gesamtabläufe beispielhaft vorgestellt. Darauf folgt die Behandlung der einzelnen Komponenten, die sich in Ansichten, Anzeigen und Bedienelemente gliedern.

16.1 Gesamtabläufe

Dieser Abschnitt beschreibt sämtliche Zustandsübergangsdiagramme, die sich nicht auf *eine* „Ansicht“, „Anzeige“ oder *ein* „Bedienelement“ reduzieren lassen, sondern mehrere davon einbeziehen und Interaktionen zwischen ihnen beschreiben.

16.1.1 Starten

Anfangs befindet sich das diesen Vorgang beschreibende Zustandsübergangsdiagramm (vgl. Abbildung 16.1) im Zustand Motor aus. Wenn das Fahrzeug gestartet werden soll, erfolgt ein Übergang in den Zustand Checklistendurchlauf. Dort werden die einzelnen Prüfungen, die jeweils als Zustände modelliert wurden, sequenziell durchlaufen. Die Übergänge zwischen diesen Zuständen sind davon abhängig, ob die Prüfungen kritische Werte bzw. Fehler ermitteln oder nicht. Im Fehlerfall wechselt das System unter Ausgabe einer entsprechenden Warnmeldung in einen Fehlerzustand, und von dort aus, nach der Bestätigung der Warnmeldung durch den Fahrer, wieder in den Startzustand.

Wenn hingegen keine der Prüfungen einen Fehler ergibt, endet die Sequenz im Zustand Motor an. Der letzte Übergang ist dabei mit dem Seiteneffekt verbunden, dass auf dem Fahrer-Display in die Stadtansicht gewechselt wird.

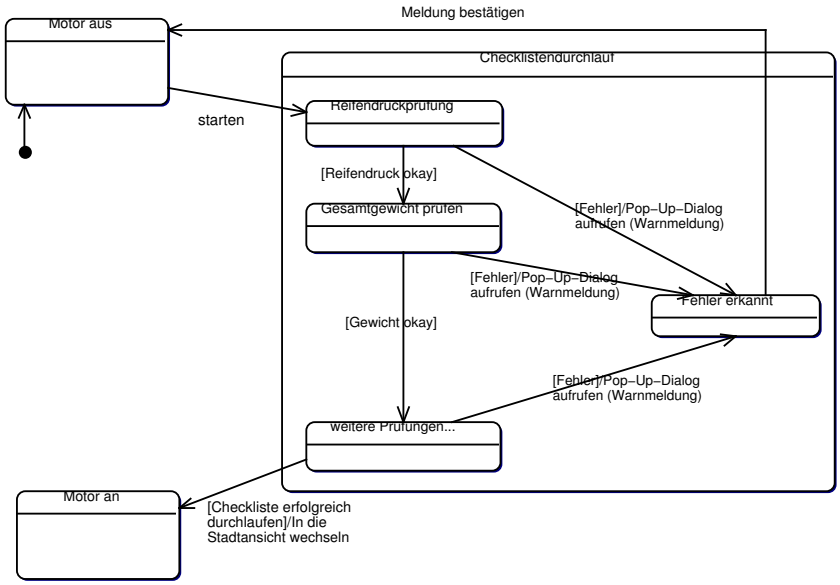

Abbildung 16.1: Zustandsübergangsdiagramm Starten

16.1.2 Während der Fahrt wird der Tankfüllstand knapp

Das in Abbildung 16.2 gezeigte Zustandsübergangsdiagramm beschreibt, was im HyCop passiert, wenn während der Fahrt die Benzinreserven knapp werden. Sobald der Tankfüllstand knapp wird, wird dies vom HyCop anhand der Signale des Tanksensors erkannt. Daraufhin wird eine Warnmeldung in einem Pop-Up-Dialog angezeigt. Sobald der Fahrer die Warnmeldung schließt, berechnet der Routenplaner eine Liste der nächstgelegenen Tankstellen und zeigt diese in einem Auswahlmenü an. Wählt der Fahrer nun eine Tankstelle aus, berechnet der Routenplaner die Route entsprechend neu und beschreibt den Weg zur Tankstelle. Solange der Motor vor Erreichen der nächsten Tankstelle nicht abgestellt wird, erfolgt nun keine Warnmeldung mehr.

16.1.3 Abstandswarner

Der Abstandswarner (vgl. Abbildung 16.3) hat die beiden Zustände kein Hindernis und Hindernis erkannt, von denen der erstgenannte der Startzustand des Systems ist. Wenn in diesem das Ereignis eintritt, dass der Abstandsmesser ein Hindernis in weniger als 25 cm Entfernung misst, erfolgt der Übergang in den Zustand Hindernis erkannt, wobei ein Pop-Up-Dialog mit einer entsprechenden Warnmeldung aufgerufen wird.

Wenn nun wiederum das Ereignis eintritt, dass der Abstandsmesser eine unkritische Entfernung meldet, so wechselt das System unter Ausblendung der Warnmeldung zurück in den Startzustand.

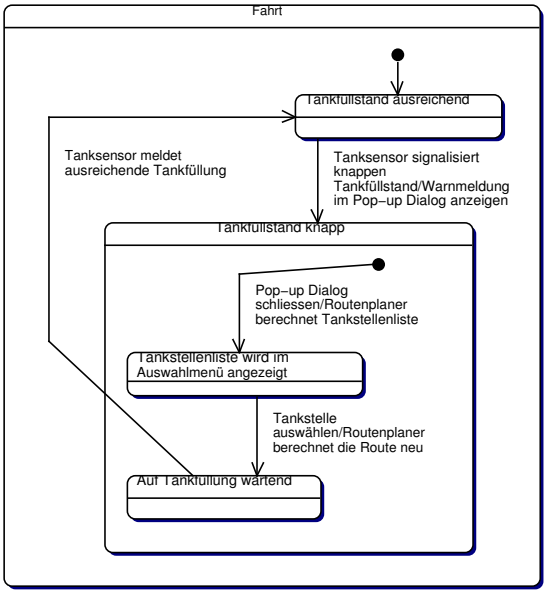

Abbildung 16.2: Zustandsübergangsdiagramm Tankfüllstand knapp

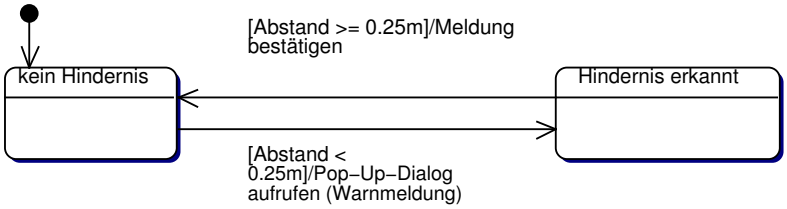

Abbildung 16.3: Zustandsübergangsdiagramm Abstandswarner

16.1.4 Medienwiedergabe

In diesem Zustandsübergangsdiagramm (siehe Abbildung 16.4 ist eine Übersicht der Abläufe bei der Wiedergabe von verschiedenen Unterhaltungsmedien zu sehen. Eine wichtige Idee dabei ist, dass das beim Abschalten des HyCop-Systems gewählte Medium beim erneuten Einschalten weitergespielt wird. Dies spiegelt sich im Zustandsübergangsdiagramm durch das History-Symbol wieder. Wie zu erkennen ist, befindet man sich in einem der Zustände Medium inaktiv, Medium aktiv, Medienliste angezeigt, Audioeinstellungen oder Videoeinstellungen. Der Zustand Medium aktiv ist dabei in zahlreiche Teilzustände unterteilt, die angeben, *welches* Medium momentan aktiv ist. Die Transitionen lassen leicht erkennen, auf welche Weise zwischen Medien gewechselt werden kann. Die Bedingungen an den Transitionspfeilen, die im Zustand Videomedium aktiv enden, gewährleisten, dass der Fahrer während der Fahrt kein Videomedium abspielen kann.

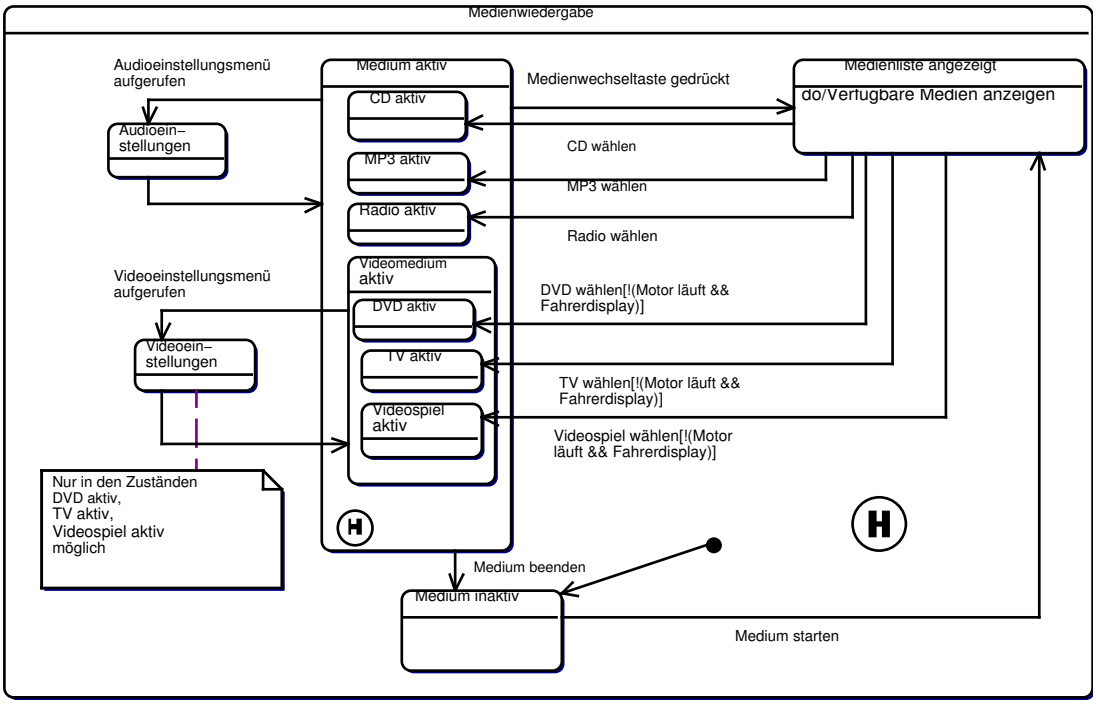

Abbildung 16.4: Zustandsübergangsdiagramm Medienwiedergabe

16.1.5 Wiedergabe von CDs, MP3s und DVDs

Die Zustände der Wiedergabe der Medien Audio-CD, MP3 und DVD sind sehr ähnlich und können deshalb in einem Zustandsübergangsdiagramm zusammengefasst werden. Wird eines dieser Medien aktiviert, befindet es sich anfangs im Zustand Abspielen. Das bedeutet, dass dieses Medium direkt angespielt wird. Durch das Betätigen bestimmter Tasten kann die Wiedergabe gesteuert werden, beispielsweise kann man einen Track vor- oder zurückspringen oder die Wiedergabe pausieren. Die Zappingtaste dient bei diesen Medien zum Anspielen von Tracks (Indexplay). Wird sie ein zweites Mal gedrückt, wird der in diesem Moment angespielte Track beibehalten. Trifft während der Wiedergabe ein Telefonanruf ein, wird das Audiomedium ausgeblendet und ein Pop-Up-Dialog angezeigt, in dem man sich dafür entscheiden kann, den Anruf anzunehmen oder abzulehnen. Wird der Anruf abgelehnt, wird das Audiomedium wieder eingeblendet, wird er angenommen, geschieht dies erst nach dem Beenden des Gesprächs.

16.1.6 Wiedergabe von TV und Radio

Diese beiden Medien können ebenfalls in einem Zustandsübergangsdiagramm (siehe Abbildung 16.6) zusammengefasst werden, da sie beide das Konzept der Sender beinhalten und somit vergleichbare Zustände besitzen. Dementsprechend haben hier die Vor- und Rücklauftaste die Funktion, den nächsten oder vorherigen Fernseh- bzw. Radiosender anzuspringen. Die Zappingtaste dient zum automatischen Durchlaufen der Sender. Wird sie ein weiteres Mal gedrückt, dann wird der in diesem Moment beim Durchlaufen angezeigte Sender beibehalten. Die

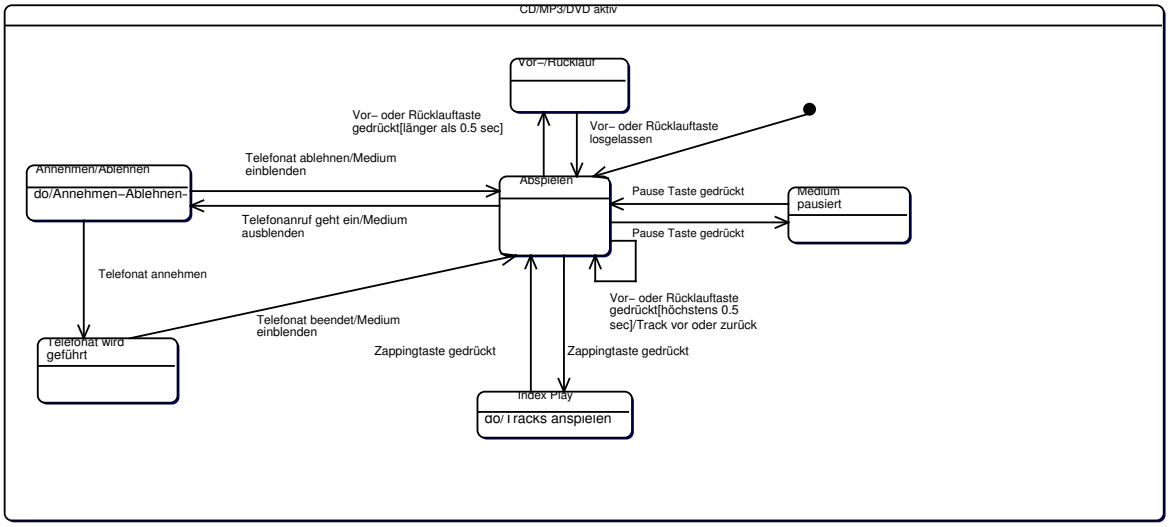

Abbildung 16.5: Zustandsübergangsdiagramm für die CD-, MP3- und DVD-Wiedergabe

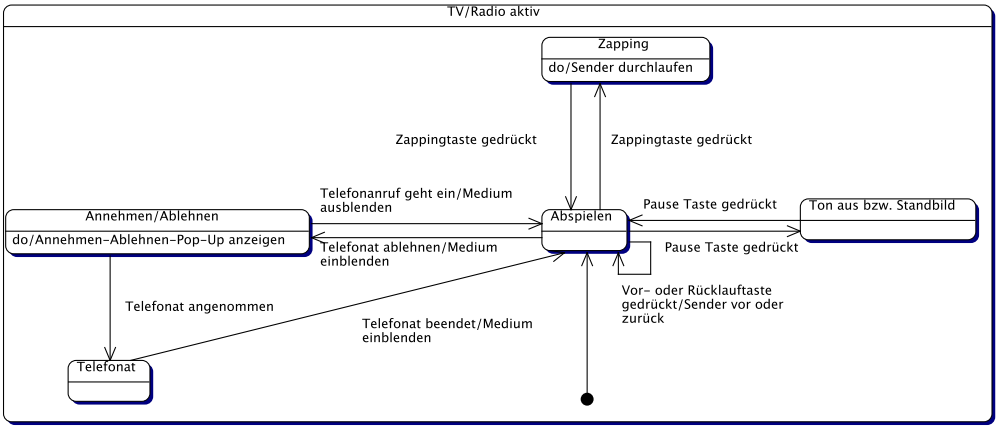

Abbildung 16.6: Zustandsübergangsdiagramm für TV und Radio

Pause-Taste dient bei diesen Medien zum Stummschalten des Tons. Bei der TV-Wiedergabe bewirkt sie darüberhinaus das Einfrieren des Bildes (Standbild). Telefonanrufe werden genau so wie in Abschnitt 16.1.5 beschrieben behandelt.

16.1.7 Videospiel

Das Unterhaltungsmedium Videospiel lässt sich schlecht in einem allgemeinen Zustandsübergangsdiagramm darstellen, da es für jedes Spiel etwas anders aussehen müsste. Daher beschränkt sich das hier angegebene Diagramm (siehe Abbildung 16.7) auf die Funktionen, die bei allen Videospielen sinnvollerweise identisch sind. Nach dem Start eines Videospiels wird dem Spieler zunächst ein Startbildschirm angezeigt. Von diesem Startbildschirm aus kann er andere Insassen zum Mitspielen einladen oder einfach das Spiel direkt starten, wenn er alleine

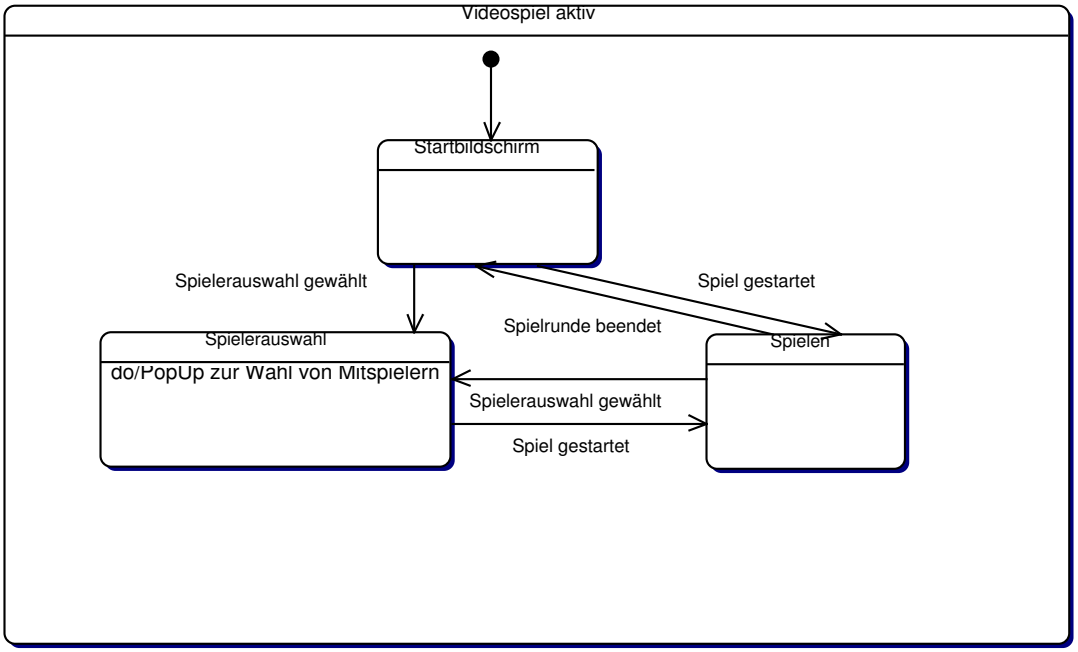

Abbildung 16.7: Zustandsübergangsdiagramm Videospiel

spielen möchte. Sollte er es sich während des Spiels doch noch anders überlegen, kann er auch von dort aus noch andere Insassen zum Mitspielen einladen. Zur Mitspielerauswahl wird dabei jeweils ein Pop-Up-Dialog mit einer Liste der für ein Spiel verfügbaren Insassen angezeigt.

16.1.8 Pannendienst kontaktieren

Das in Abbildung 16.8 Statechart weist Zustände Idle, Auswahlmenue angezeigt, Pannendienst kontaktiert und Daten ausgetauscht auf. Der Startzustand ist der Leerlauf, also der Idle-Zustand. Wird ein Pannendienst-Kontakt angefordert, nimmt das System den Zustand Auswahlmenue angezeigt an. Befindet sich das System im Zustand Auswahlmenue angezeigt, kann entweder bei dem Auswahl eines Eintrags und einer erfolgreichen Verbindung der Zustand Pannendienst kontaktiert angenommen, oder, bei einem Verbindungs- oder Eingabeabbruch, der Zustand beibehalten werden. Ist das System im Zustand Pannendienst kontaktiert, können Daten ausgetauscht und der Zustand Daten ausgetauscht angenommen werden. Aus dem Zustand Daten ausgetauscht kann das System durch einen Verbindungsabbau in den Idle-Zustand übergehen.

16.1.9 Personalisieren

In diesem Statechart (siehe Abbildung 16.9) werden sowohl der Kartenleser als auch das Konfigurationssystem beschrieben.

Der Kartenleser an einem Sitzplatz ist entweder belegt oder leer, was auch der Startzustand ist. Der Wechsel zwischen diesen beiden Zuständen erfolgt, wenn die Ereignisse Karte einstecken bzw. Karte entnehmen eintreten. Im erstgenannten Fall wird beim Übergang die Handlung

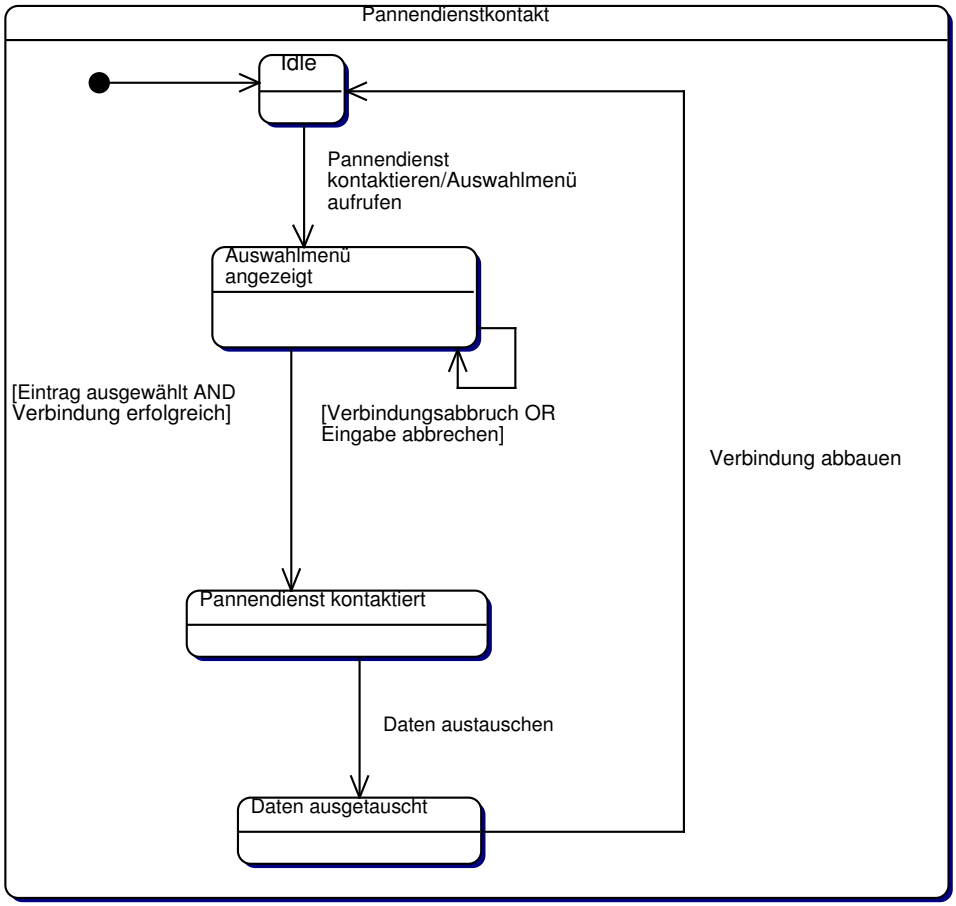

Abbildung 16.8: Zustandsübergangsdiagramm Pannendienst kontaktieren

Konfigurieren angestossen. Das Konfigurationssystem hat lediglich einen Zustand. Wenn eine Konfiguration des Sitzes, der Spiegel oder der Anzeigen erfolgen soll, also das auslösende Ereignis Konfigurieren eintritt, werden die auf der Chipkarte hinterlegten Einstellungen hergestellt.

16.1.10 Tanken

Das in Abbildung 16.10 gezeigte Statechart beschreibt folgenden Vorgang: Sobald das Fahrzeug eine Tankstelle erreicht, wird automatisch eine drahtlose Verbindung zwischen dem HyCop und der Bedienstation der Tankstelle aufgebaut. Wenn diese Verbindung aufgebaut wurde, wird das Serviceangebot der Tankstelle in einem Auswahlmenü angezeigt. Nun kann der Fahrer verschiedene Services auswählen. Bestätigt der Fahrer die Serviceauswahl, wird ein entsprechender Bezahldialog angezeigt. Durch Bestätigung der Rechnung wird der Bezahldialog geschlossen, woraufhin erneut die Möglichkeit zur Serviceauswahl besteht.

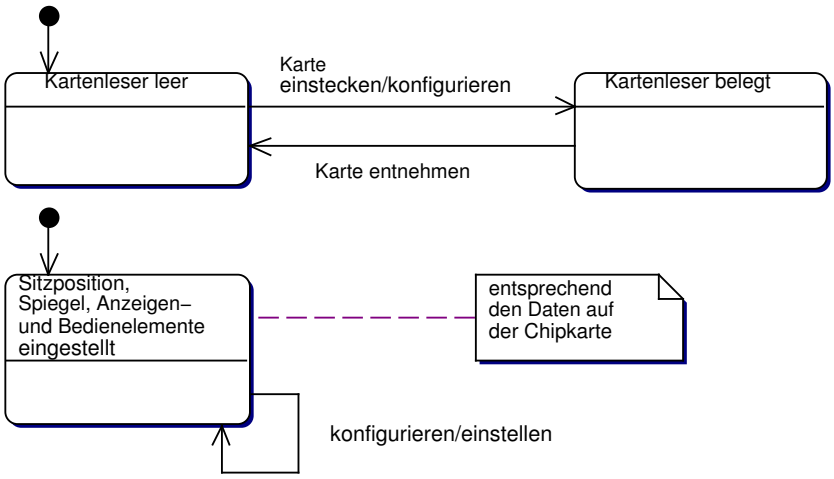

Abbildung 16.9: Zustandsübergangsdiagramm Personalisieren

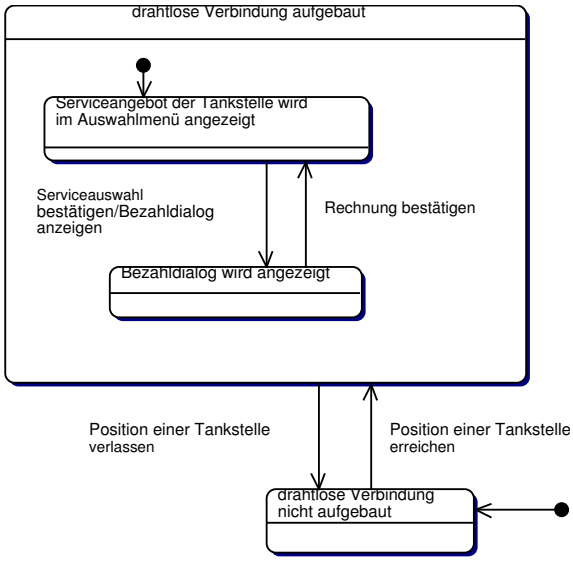

Abbildung 16.10: Zustandsübergangsdiagramm Tankstellenaufenthalt

16.1.11 Elektronisches Bezahlen

Das, in Abbildung 16.11 gezeigte, System kann folgende Zustände annehmen: Idle, Eingabe-Dialog angezeigt, Authentifizierung beginnt und Authentifizierung beendet.

Wenn sich das System im Leerlauf, also im Zustand codeIdle befindet, und ein Bezahlvorgang angefordert wird, geht das System in den Zustand Eingabe-Dialog angezeigt. Wird die Bezahlung vom Nutzer abgelehnt, kehrt das System in den Idle-Zustand zurück. Stimmt der Benutzer zu, wird der Zustand Authentifizierung beginnt angenommen.

Ist das System im Zustand Eingabe-Dialog angezeigt und die Authentifizierung nicht erfolgreich,

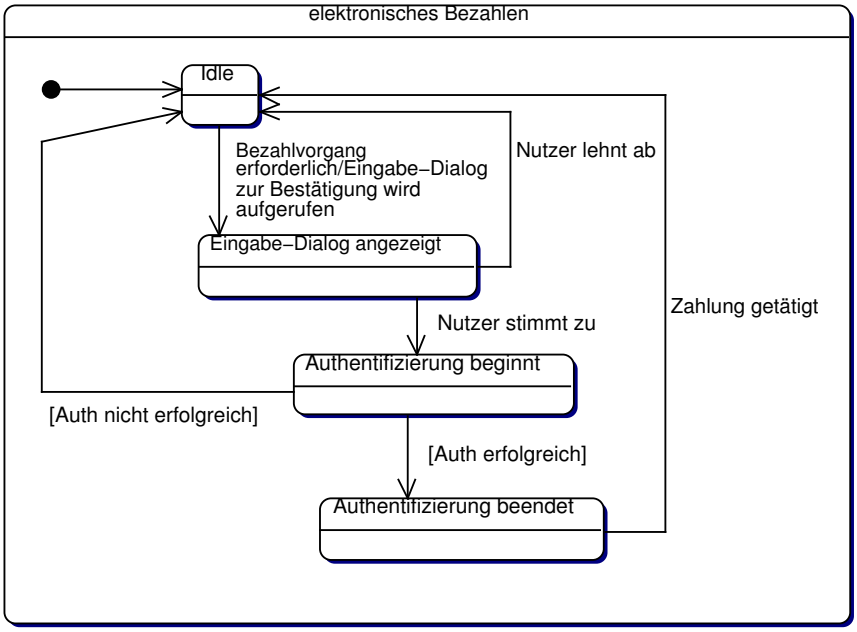

Abbildung 16.11: Zustandsübergangsdiagramm Elektronisches Bezahlen

wird der Zustand Idle angenommen. Bei erfolgter Authentifizierung geht das System in den Zustand Authentifizierung beendet über.

Befindet sich das System im Zustand Authentifizierung beendet und ein Zahlungsvorgang erfolgt, kehrt das System in den Idle-Zustand zurück.

16.1.12 Überladungswarner

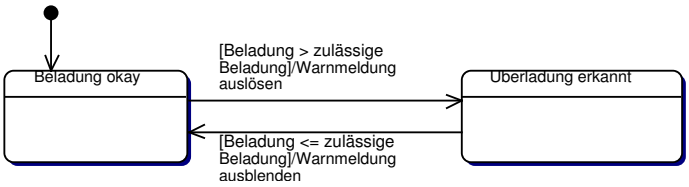

Abbildung 16.12: Zustandsübergangsdiagramm Überladungswarner

Der Überladungswarner (vgl. Abbildung 16.12) ähnelt in seiner Funktionsweise dem Abstandswarner. Vom Startzustand, Beladung okay, wird in den Zustand Überladung erkannt gewechselt, sobald die vom Gewichtssensor gemessene Beladung die zulässigen Werte überschreitet. Bei diesem Wechsel wird zusätzlich ein Pop-Up-Dialog mit einer entsprechenden Warnmeldung aufgerufen. Wenn im Überladungszustand das Ereignis eintritt, dass das gemessene Gewicht wieder innerhalb des zulässigen Bereichs liegt, erfolgt der Rücksprung in den Startzustand. Dabei wird der vorher aufgerufene Pop-Up-Dialog automatisch geschlossen.

16.1.13 Verriegelungssystem

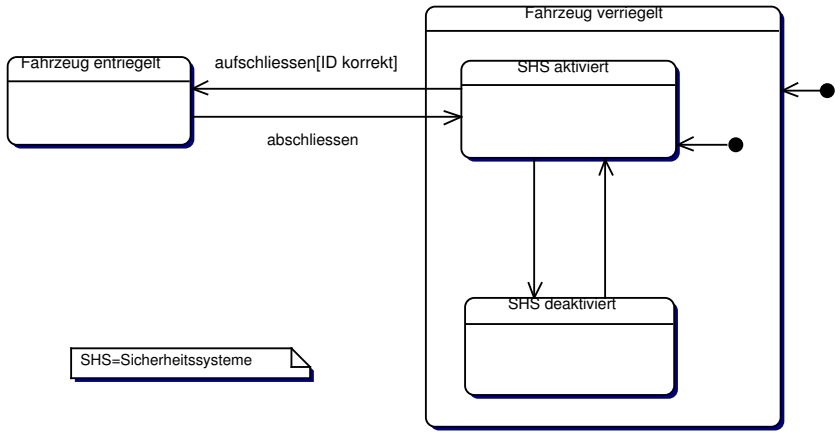

Abbildung 16.13: Zustandsübergangsdiagramm Verriegelungssystem

Das Fahrzeug ist entweder verriegelt, was auch der Startzustand des Systems ist, oder entriegelt. Die Übergänge zwischen diesen Zuständen erfolgen durch das Auf- bzw. Abschließen. Beim Aufschließen tritt hierbei zusätzlich die Bedingung auf, dass das Authentifizierungssystem eine korrekte ID melden muss. Vergleiche hierzu Abbildung 16.13. Der Zustand Fahrzeug verriegelt umfasst zwei innere Zustände für das Sicherheitssystem. Normalerweise ist das System aktiviert. Für Fälle, in denen eine Alarmanlage keinen Sinn macht, z.B. bei einem Werkstattaufenthalt oder dem Besuch einer Waschanlage, kann sie aber deaktiviert werden.

16.1.14 Zugriff externer Systeme

Das Statechart (siehe Abbildung 16.14 weist folgende Zustände auf: Idle, Eingabe-Dialog angezeigt, Authentifizierung und Zugriff hergestellt. Wenn sich das System im Zustand Idle befindet und ein externes System einen Zugriff wünscht, wird der Zustand Eingabe-Dialog angezeigt angenommen. Ist das System im Zustand Eingabe-Dialog angezeigt und eine Zustimmung seitens des Nutzers oder des Systems kommt, wird der Zustand Authentifizierung angenommen. Befindet sich das System im Zustand Authentifizierung und eine Authentifizierung erfolgt, wird der Zustand Zugriff hergestellt angenommen. Solle die Authentifizierung nicht erfolgreich sein, kehrt das System in den Zustand Idle zurück. Ist das System im Zustand Zugriff hergestellt und der Zugriff beendet wird, wird der Zustand Idle angenommen.

16.1.15 Diagnose

Dieses Statechart (siehe Abbildung 16.15 besitzt zwei Zustände: System nicht OK und System OK.

Werden vom Diagnosesystem Fehler gefunden, wird der Zustand System nicht OK angenommen (falls sich das System im Zustand System OK befindet), oder beibehalten (falls das System im Zustand System nicht OK ist).

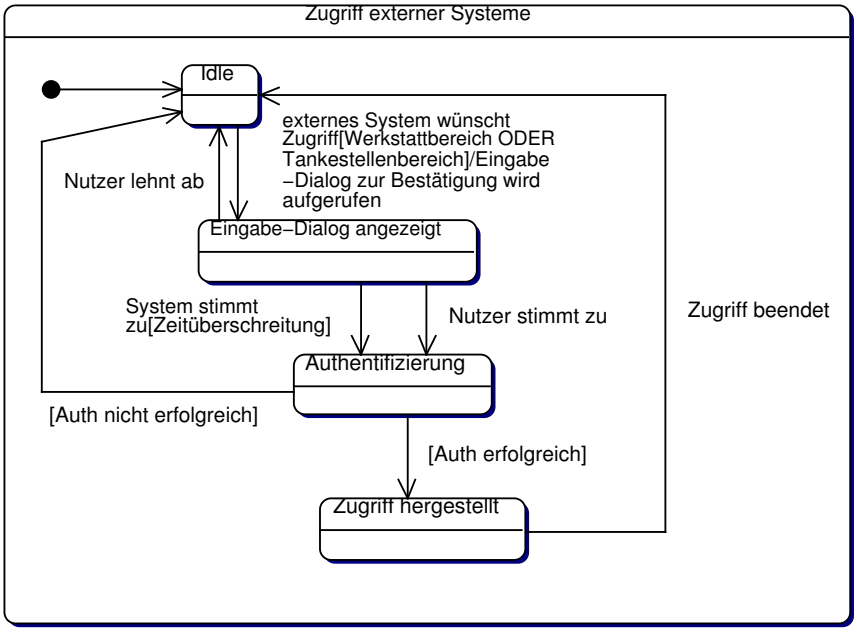

Abbildung 16.14: Zustandsübergangsdiagramm Zugriff externer Systeme

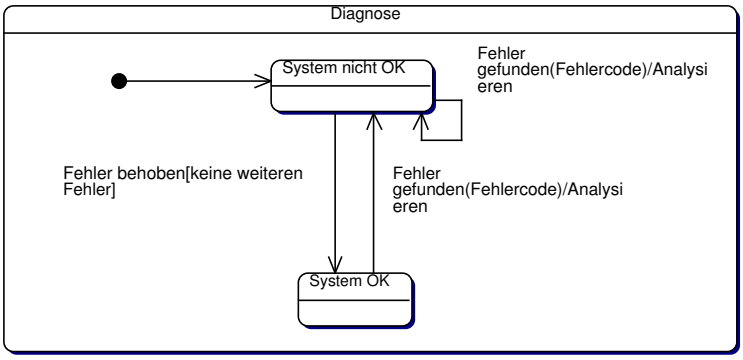

Abbildung 16.15: Zustandsübergangsdiagramm Diagnose

Werden keine Fehler gefunden, wird der Zustand System OK angenommen.

16.1.16 Fehleranalyse

Dieses Statechart (siehe Abbildung 16.16 besitzt den Zustand Fehleranalyse, der seinerseits drei Unterzustände besitzt: Idle, Fehler analysiert und Massnahmen.

Der Startzustand ist der Leerlauf, also der Idle-Zustand. Daraus kann der Zustand Fehler analysiert erreicht werden.

Befindet sich das System im Zustand Fehler analysiert, und der Fehler bekannt ist, wird dieser ignoriert und das System kehrt in den Idle-Zustand zurück. Ansonsten wird der Zustand

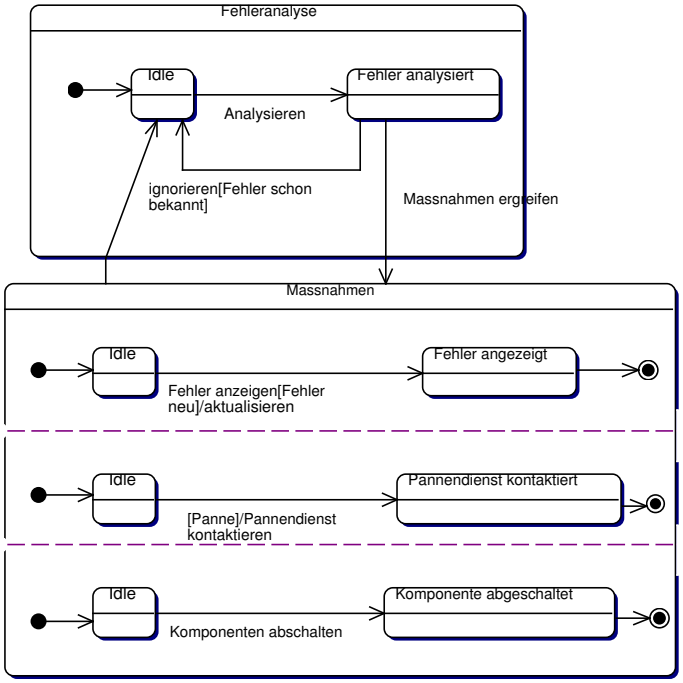

Abbildung 16.16: Zustandsübergangsdiagramm Fehleranalyse

Massnahmen angenommen.

In dem Zustand Massnahmen sind drei parallele Abläufe möglich. Der Startzustand für alle drei Abläufe ist der Idle-Zustand. Ist der Fehler neu, kann der Zustand Fehler angezeigt angenommen werden. Handelt es sich bei dem Fehler um eine Panne, kann gleichzeitig der Zustand Pannendienst kontaktiert erreicht werden. Parallel dazu kann durch Komponentenabschaltung der Zustand Komponenten abgeschaltet angenommen werden.

16.2 Ansichten

Autor: *Leonore Dietrich*

Dieser Abschnitt beschreibt die sechs möglichen Ansichten in HyCop. Hierbei sind Stand-, Stadt-, Land-, Autobahn- und Werkstattansicht ausschließlich dem Fahrerdisplay zugeordnet.

16.2.1 Standansicht

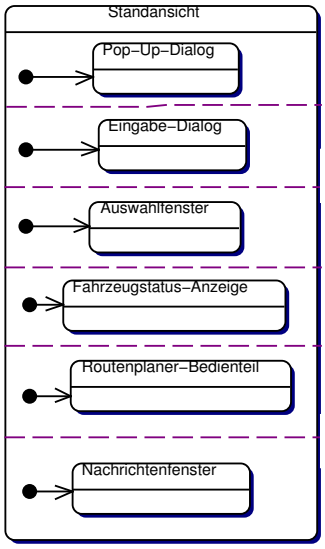

Abbildung 16.17: Standansicht

Die Standansicht setzt sich zusammen aus den Elementen Pop-Up-Dialog, Eingabedialog, Auswahlfenster, Fahrzeugstatusanzeige, Routenplanerbedienteil und Nachrichtenfenster (vgl. Abbildung 16.17). Diese Elemente können parallel angezeigt werden. Typischerweise erscheinen Pop-Up-Dialog, Eingabedialog, und Auswahlfenster nur bei Bedarf. Die restlichen Komponenten werden permanent angezeigt.

16.2.2 Stadtansicht

Die Stadtansicht setzt sich zusammen aus den Elementen Pop-Up-Dialog, Eingabedialog, Auswahlfenster, Fahrzeugstatusanzeige, Routenanzeige, Nachrichtenfenster und Kamerafenster (vgl. Abbildung 16.18). Diese Elemente können parallel angezeigt werden. Typischerweise erscheinen Pop-Up-Dialog, Eingabedialog, Kamerafenster und Auswahlfenster nur bei Bedarf. Die restlichen Komponenten werden permanent angezeigt.

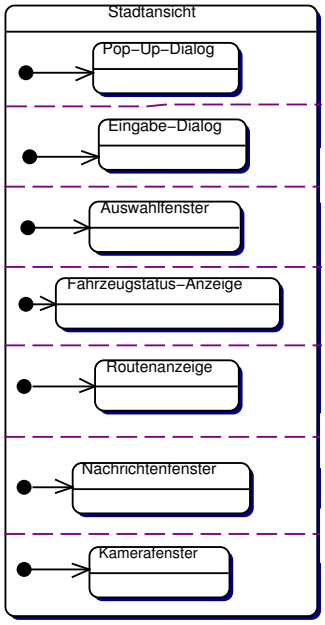

Abbildung 16.18: Stadtansicht

16.2.3 Landansicht

Die Landansicht setzt sich zusammen aus den Elementen Pop-Up-Dialog, Eingabe-Dialog, Auswahlfenster, Fahrzeugstatusanzeige, Routenanzeige, Nachrichtenfenster und Kamerafenster (vgl. Abbildung 16.19). Diese Elemente können parallel angezeigt werden. Typischerweise erscheinen Pop-Up-Dialog, Eingabedialog, Kamerafenster und Auswahlfenster nur bei Bedarf. Die restlichen Komponenten werden permanent angezeigt.

16.2.4 Autobahnansicht

Die Autobahnansicht setzt sich zusammen aus den Elementen Pop-Up-Dialog, Eingabedialog, Auswahlfenster, Fahrzeugstatusanzeige, Routenanzeige, Nachrichtenfenster und Kamerafenster (vgl. Abbildung 16.20). Diese Elemente können parallel angezeigt werden. Typischerweise erscheinen Pop-Up-Dialog, Eingabedialog, Kamerafenster und Auswahlfenster nur bei Bedarf. Die restlichen Komponenten werden permanent angezeigt.

16.2.5 Werkstattansicht

Die Werkstattansicht setzt sich zusammen aus den Elementen Pop-Up-Display, Eingabedialog, Auswahlfenster, Fahrzeugstatusanzeige, Nachrichtenfenster und Diagnosemenü (vgl. Abbildung 16.21). Diese Elemente können parallel angezeigt werden. Typischerweise erscheinen Pop-Up-Dialog, Eingabedialog und Auswahlfenster nur bei Bedarf. Die restlichen Komponenten werden permanent angezeigt.

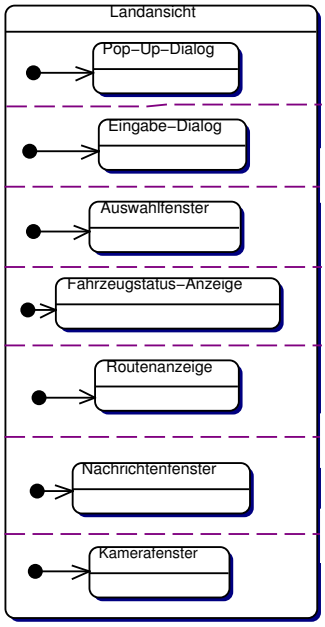

Abbildung 16.19: Landansicht

16.2.6 Mitfahreransicht

Die Mitfahreransicht setzt sich zusammen aus den Elementen Pop-Up-Dialog, Eingabedialog, Auswahlfenster, Fahrzeugstatusanzeige, Nachrichtenfenster, Routenanzeige, Routenplanerbedienteil, Medienfenster, -menü, -navigation und -wiedergabe (vgl. Abbildung 16.22). Diese Elemente können parallel angezeigt werden. Typischerweise erscheinen Pop-Up-Dialog, Eingabedialog und Auswahlfenster nur bei Bedarf. Elemente des Fahrerdisplays, wie die Fahrzeugstatusanzeige oder das Nachrichtenfenster, können einzeln eingeblendet werden. Die restlichen Komponenten werden je nach aktuell gewähltem Medium konfiguriert, wobei die Medienauswahl über das Medienmenü erfolgt, welches zu jeder Zeit angezeigt wird bzw. aktiviert werden kann.

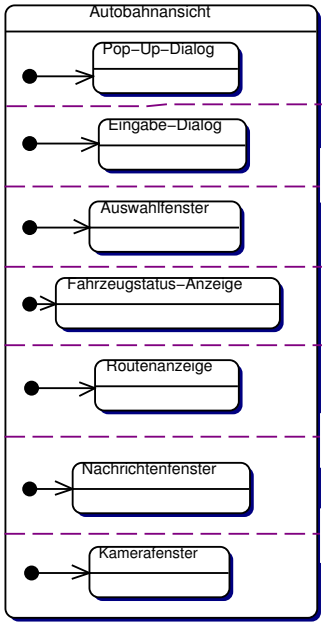

Abbildung 16.20: Autobahnansicht

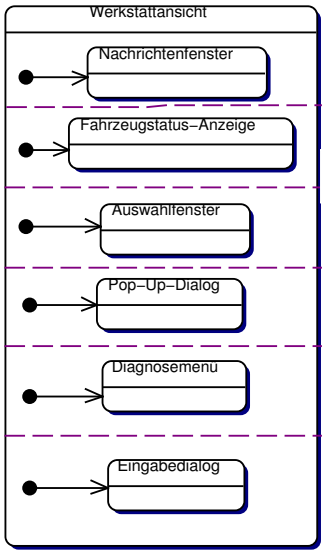

Abbildung 16.21: Werkstattansicht

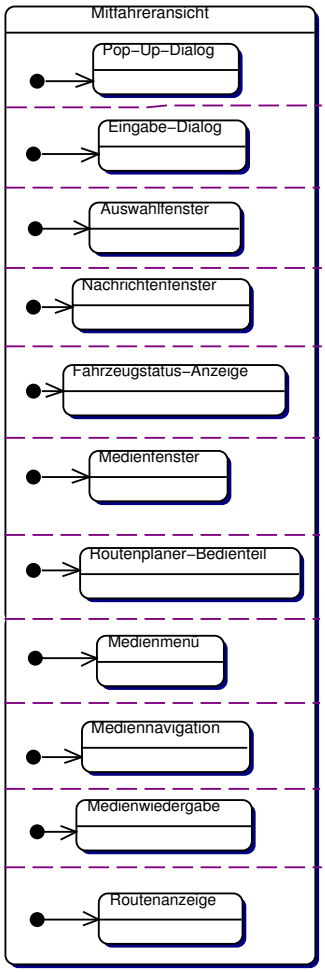

Abbildung 16.22: Mitfahreransicht

16.3 Anzeigen

Autoren:

Dieser Abschnitt beschreibt die unterschiedlichen Anzeigen, die in HyCop benötigt werden.

16.3.1 Fahrzeugstatusanzeige

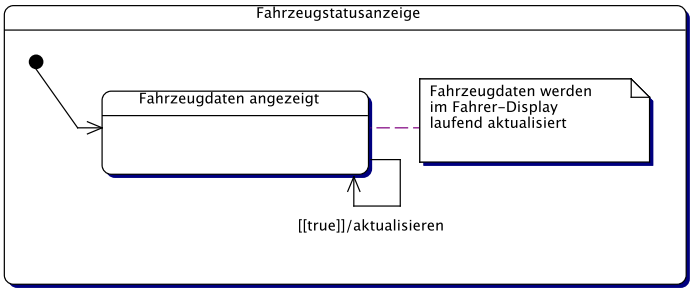

Abbildung 16.23: Fahrzeugstatusanzeige

Die Fahrzeugstatusanzeige (vgl. Abbildung 16.23) hat lediglich einen Zustand. Eine permanent schaltende Transition von diesem Zustand in sich selbst sorgt dafür, dass die angezeigten Fahrzeugdaten stets auf dem aktuellen Stand sind.

16.3.2 Kamerafenster

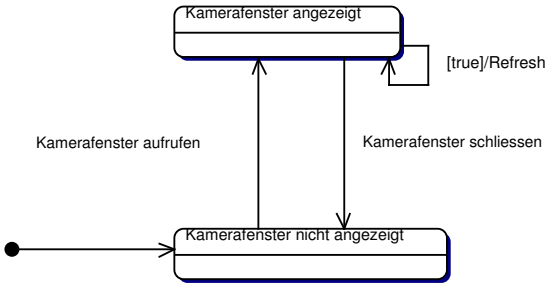

Abbildung 16.24: Kamerafenster

Das Kamerafenster besitzt zwei Zustände. Startzustand ist Kamerafenster nicht angezeigt. Wird das Kamerafenster aufgerufen, findet ein Übergang in den Zustand Kamerafenster angezeigt statt (vgl. Abbildung 16.24). In diesem Zustand wird laufend ein Refresh ausgeführt. Die Aktion Kamerafenster schließen führt zum Wechsel in den Startzustand.

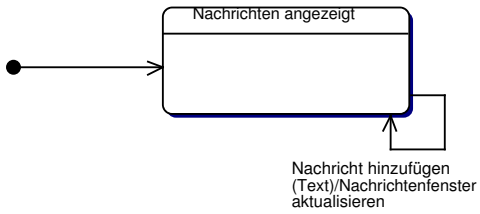

Abbildung 16.25: Nachrichtenfenster

16.3.3 Nachrichtenfenster

Das Nachrichtenfenster (vgl. Abbildung 16.25) hat, ähnlich der Fahrzeugstatus-Anzeige, nur einen Zustand. Wenn das Ereignis Nachricht hinzufügen eintritt, wird ein Übergang von diesem Zustand in sich selbst vorgenommen, bei dem die Aktualisierung der Fensterinhalte als Seiteneffekt auftritt.

16.3.4 Diagnosemenü

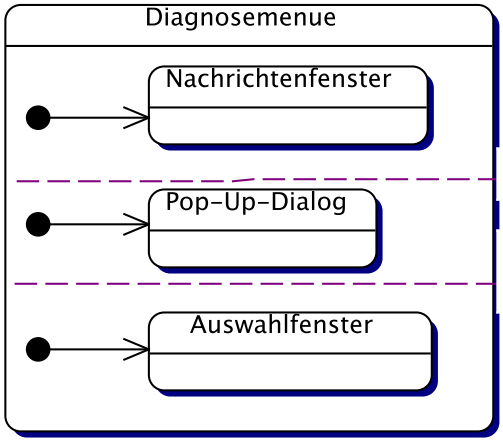

Abbildung 16.26: Diagnosemenü

Das Diagnosemenü (vgl. Abbildung 16.26) besitzt drei parallele Automaten. Im Automat Nachrichtenfenster werden laufend Informationen über das Fahrzeug und Diagnosergebnisse angezeigt. In den Zustand Pop-Up-Dialog gelangt man, wenn wichtige Meldungen angezeigt werden müssen.

16.4 Bedienelemente

Autoren:

Dieser Abschnitt beschreibt die in den unterschiedlichen Ansichten vorkommenden Bedienelemente.

16.4.1 Eingabedialog

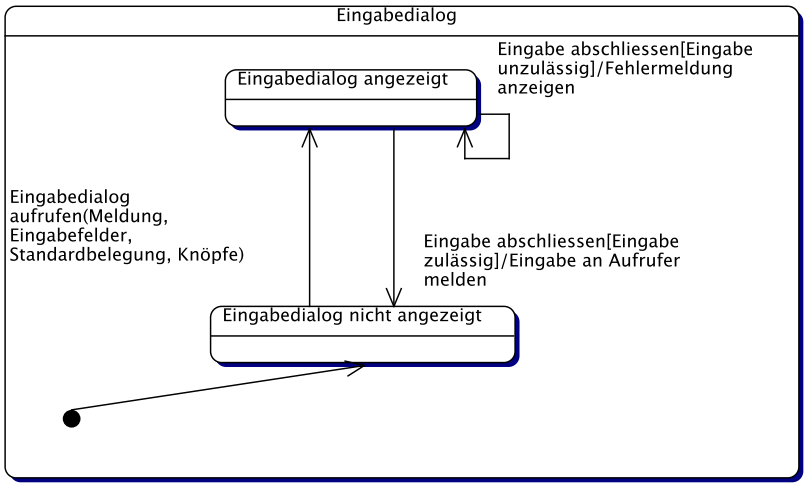

Abbildung 16.27: Eingabedialog

Der Eingabedialog besitzt zwei Zustände, wobei Eingabedialog nicht angezeigt Startzustand ist. Wird der Eingabedialog mit den Parametern Meldung, Eingabefelder, Standardbelegung und Knöpfe aufgerufen, findet ein Übergang in den Zustand Eingabedialog angezeigt statt (vgl. Abbildung 16.27). Ein Wechsel zurück in den Startzustand findet statt, wenn die Eingabe zulässig ist und abgeschlossen wird. Die Eingabe wird dann an die aufrufende Komponente gemeldet. Ist die Eingabe unzulässig, wird eine entsprechende Fehlermeldung ausgegeben und der Eingabedialog verbleibt im Zustand Eingabedialog angezeigt.

16.4.2 Auswahlmenü

Das Auswahlmenü besitzt zwei Zustände. Startzustand ist Auswahlmenü nicht angezeigt. Wird das Auswahlmenü mit dem Parameter Liste aufgerufen, so findet ein Übergang in den Zustand Auswahlmenü angezeigt statt (vgl. Abbildung 16.28). Von diesem Zustand aus wird ein Pop-Up-Dialog aufgerufen, wenn der Benutzer Detailinformationen auswählt. Durch Abbrechen der Eingabe oder Auswahl eines Listeneintrages wird in den Zustand Auswahlmenü nicht angezeigt gewechselt. Bei letzterer Aktion wird zusätzlich der gewählte Eintrag an die aufrufende Komponente gemeldet.

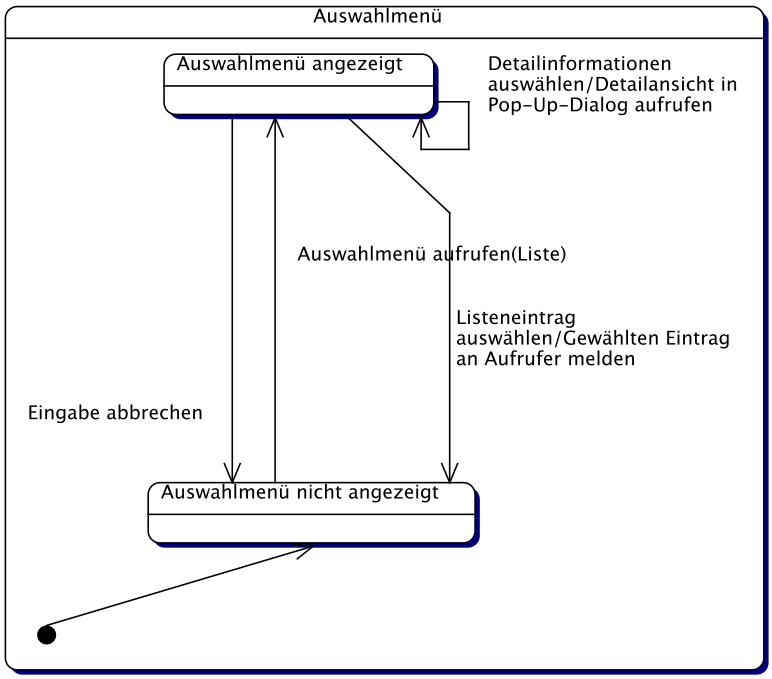

Abbildung 16.28: Auswahlmenü

16.4.3 Auswahlfenster

Das Auswahlfenster (siehe Abbildung 16.29) besitzt zwei Zustände. Startzustand ist keine Diagnose ausgewählt. Wird nun eine Diagnose gewählt, wird in den Zustand Diagnose gewechselt und die Ergebnisse werden im Nachrichtendenster angezeigt. Anschließend wird wieder der Zustand keine Diagnose ausgewählt angenommen.

16.4.4 Pop-Up-Dialog

Der Pop-Up-Dialog besitzt zwei Zustände. Startzustand ist Pop-Up-Dialog nicht angezeigt. Wird der Pop-Up-Dialog mit dem Parameter Meldung aufgerufen, findet ein Übergang in den Zustand Pop-Up-Dialog angezeigt statt (vgl. Abbildung 16.30). Aus diesem Zustand wird bei Bestätigung der Meldung wieder in den Startzustand gewechselt.

16.4.5 Routenplaner-Bedienteil

Das Routenplaner-Bedienteil besitzt zwei Hauptzustände. Startzustand ist Bedienteil nicht angezeigt. Wird die Routenplanung aktiviert, findet ein Übergang in den Zustand Bedienteil angezeigt statt. Dieser Zustand besteht wiederum aus zwei Zuständen (vgl. Abbildung 16.31). Sollte keine History vorliegen, ist der Zustand Route nicht festgelegt Startzustand. Durch Eingabe einer Station wird eine Route berechnet und in den Zustand Route festgelegt gewechselt. Hier können weitere Stationen eingegeben oder Routenvorgaben geändert werden, wodurch

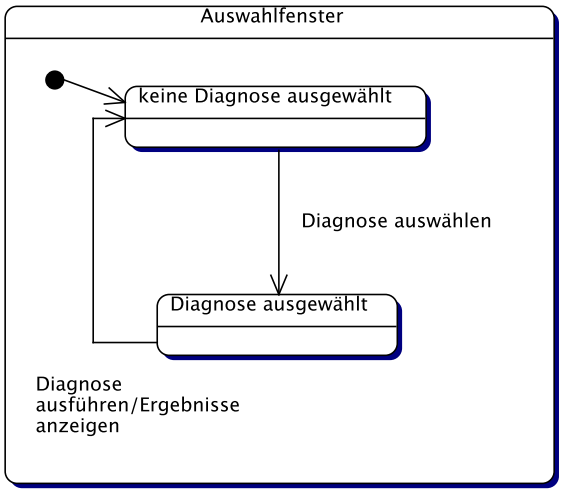

Abbildung 16.29: Auswahlfenster

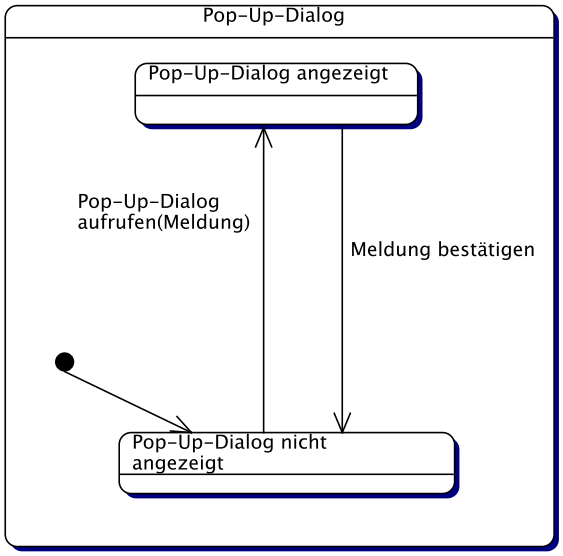

Abbildung 16.30: Pop-Up-Dialog

jeweils die Route neu berechnet wird. Entfernt man eine Station, so wird ebenfalls die Route neu berechnet und, sofern mehr als eine Station vorhanden war, im Zustand verblieben. Ist bei Entfernen einer Station insgesamt nur eine Station vorhanden oder wird explizit die Route abgebrochen, so findet ein Übergang in den Zustand Route nicht festgelegt statt. Der Zustand Bedienteil nicht angezeigt wird durch Abschließen der Routenplanung erreicht.

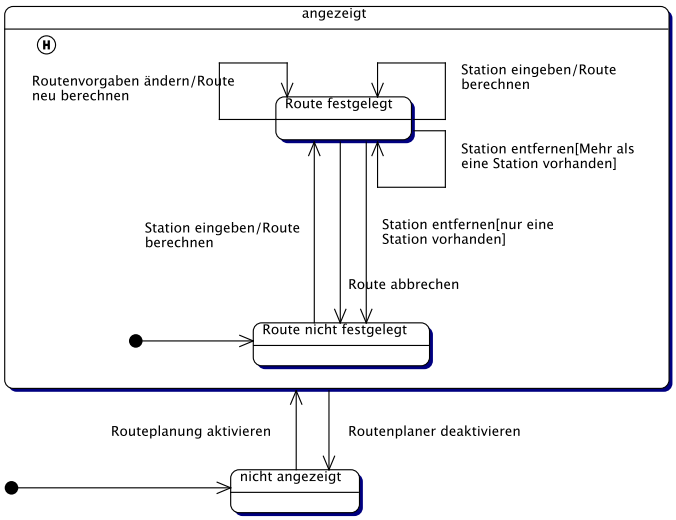

Abbildung 16.31: Routenplaner-Bedienteil

Kapitel 17

Medienobjektdiagramme

17.1 Medienobjekttabelle

Autor: *Tobias Wolf*

Die folgende Tabelle 17.1 listet alle bisher identifizierten Medienobjekte des hypermedialen Cockpits auf. Dabei ist hier schon eine deutliche Verfeinerung gegenüber der Übersicht über die Anzeigen, Bedienelemente und Sensoren in Abschnitt 15 zu erkennen. Die Sensoren fallen bei der Betrachtung der Systemkomponenten als Medienobjekte völlig heraus, da sie nicht Bestandteil des Cockpits selbst sind, sondern es nur mit externen Daten beliefern. Das heißt, ein Insasse hat keinen direkten Einfluss auf die Sensoren.

Eine Beschreibung der einzelnen Medienobjekte und ihrer Beziehungen untereinander erfolgt in den Abschnitten 17.2, 17.3 und 17.5.

Es erfolgt eine Zuweisung der Medienobjekte zu den Ansichten des Cockpits, wobei nach der Verfügbarkeit und Relevanz der Objekte klassifiziert werden soll. Dabei wird folgende Unterscheidung gemacht:

- \+ Das Medienobjekt ist in der Ansicht enthalten. Es muss ständig sichtbar sein können, d.h. es kann nicht von Insassen abgeschaltet werden. Allerdings kann HyCop es ggf. ausblenden (z.B. Kontrolleuchten, Pop-Up-Dialoge)
- O Das Medienobjekt kann in der Ansicht enthalten sein. Es ist vom Insassen auf einfachem Weg (z.B. über Menus) ein- oder ausblendbar.
- K Das Medienobjekt ist nicht in der Ansicht enthalten, es ist aber möglich es durch konfigurieren der Ansicht in die Ansicht aufzunehmen.
- \- Das Medienobjekt darf nicht in der Ansicht enthalten sein.

Medienobjekte	Stand-ansicht	Stadt-ansicht	Land-/Auto-bahn-ansicht	Werk-statt-ansicht	Mitfah-reran-sicht
Fahrzeugstatus-Anzeige:					
Kilometerstand	K	+	+	O	K
Tageskilometerstand	K	+	+	O	K
Öltemperatur	K	O	O	O	-
Tankanzeige	+	+	+	O	K
Ölstandsanzeige	+	O	O	O	-
Reifendruckanzeige	+	O	O	O	-
Gesamtgewicht	+	K	K	-	-
Tachometer	-	+	+	O	K
Drehzahlmesser	-	O	O	O	K
Innentemperatur	O	K	K	-	K
Aussentemperatur	O	K	K	-	K
Kühlwassertemperatur	K	+	+	O	-
Kühlwasserstandanzeige	+	O	O	O	-
Benzinverbrauchsanzeige	-	O	O	O	K
Uhr	O	O	O	O	O
Abstandswarner	-	+	+	-	-
Kontrolleuchten:					
Batterie	+	+	+	O	-
Tankwarnleuchte	+	+	+	O	-
Reifendruck kritisch	+	+	+	O	-
Ölstand kritisch	+	+	+	O	-
Motortemperatur kritisch	+	+	+	O	-
Gurtwarnleuchte	+	+	+	O	-
Standleuchten	+	+	+	O	-
Abblendlicht	+	+	+	O	-
Fernlicht	+	+	+	O	-
Blinker	+	+	+	O	-
Rücklicht	+	+	+	O	-
Nebelschlussleuchte	+	+	+	O	-
Rückfahrscheinwerfer	+	+	+	O	-
Nebelscheinwerfer	+	+	+	O	-
Innenbeleuchtung	+	+	+	O	-
Gesamtgewicht kritisch	+	+	+	O	-
ABS	+	+	+	O	-
ASR	+	+	+	O	-
ESP	+	+	+	O	-
Airbags	+	+	+	O	-
Kindersicherung	+	+	+	O	-
Reifendruck	+	+	+	O	-
Glatteiswarnleuchte	+	+	+	O	-
Handbremsenwarnleuchte	+	+	+	O	-
Türwarnleuchte	+	+	+	O	-
Nachrichten:					
Durchlaufe Checkliste	+	-	-	+	-
Kindersicherung aktiv	+	+	+	+	+
Checkliste durchlaufen	+	-	-	+	-
Hotel gebucht	+	+	+	-	+
Hotel in Route aufgenommen	+	+	+	-	+
Parkplatz in Route aufgenommen	+	+	+	-	+
Tankstelle in Route aufgenommen	+	+	+	-	+
Rechnung bezahlt	+	+	+	-	+
Erfolgreiche Kontaktaufnahme mit einem Pannendienst	+	+	+	-	+
Nichterfolgreiche Kontaktaufnahme mit einem Pannendienst	+	+	+	-	+
Datenaustausch mit einem Pannendienst	+	+	+	-	+
Verbindungsabbau von einem Pannendienst	+	+	+	-	+

Medienobjekte	Stand-ansicht	Stadt-ansicht	Land-/Auto-bahn-ansicht	Werk-statt-ansicht	Mitfah-reran-sicht
Erfolgreiche Verbindung mit einem externen System	+	+	+	+	+
Verbindungsabbau von einem externen System	+	+	+	+	+
Aktivitäten von externen Systemen	+	+	+	+	+
Pop-Up-Dialoge:					
"Beifahrer nicht angeschnallt"	+	+	+	-	+
"Kritischer Fehler bei Checklisten-durchlauf: ..."	+	-	-	+	-
Hinderniswarnung	-	+	+	-	-
Warnung: Tankfüllstand knapp	+	+	+	-	-
Staumeldung	+	+	+	-	K
Parkplatzdetails	+	+	+	-	+
Hoteldetails	+	+	+	-	+
Anruferinformation	+	+	+	-	+
Tankstellendetails	+	+	+	-	+
"Pannendienst kontaktieren"	+	+	+	-	+
Eingabedialoge:					
Bezahldialog	+	+	+	-	+
Hotelsuchmaske	O	O	O	-	O
Buchungsdialog	O	O	O	-	O
Parkplatzsuchmaske	O	O	O	-	O
Tankstellensuchmaske	O	O	O	-	O
Routenplanerbedienteil	O	O	O	-	O
Zugriff externer Systeme	O	O	O	+	-
Auswahlmenüs:					
Tankstellenliste	O	O	O	-	O
Serviceangebot der Tankstelle	O	O	O	-	O
Parkplatzliste	O	O	O	-	O
Hotelliste	O	O	O	-	O
Medienliste	O	O	O	-	O
Diagnosemenü	-	-	-	+	-
Konfigurationsmenüs:					
Display-Layout (Farben, ...)	O	-	-	-	O
Akustische Signale	O	-	-	-	O
Funktionsumfang	O	-	-	-	O
Zusammenstellen der Ansichten	O	-	-	-	O
Anzeigeelemente:					
Unterhaltungsmedienfenster	O	-	-	-	O
Unterhaltungsmedien-Informationsanzeige	O	O	O	-	O
Telefongespräch-Informationen	O	O	O	-	O
Routenplaner	O	O	O	-	O
Kamerafenster	O	+	O	-	K
Werkstattnachrichtenfenster	-	-	-	+	-
Bedienelemente:					
Unterhaltungsmediennavigation	O	O	O	-	O
Lautstärkeregler	+	+	+	-	+
Mitspielerauswahl	O	-	-	-	O
Diagnosetool	-	-	-	+	-

Tabelle 17.1: Medienobjekte von HyCop

17.2 Packen und Beginn einer Fahrt

Autoren: *Markus Niehammer*
Daniel Moelle

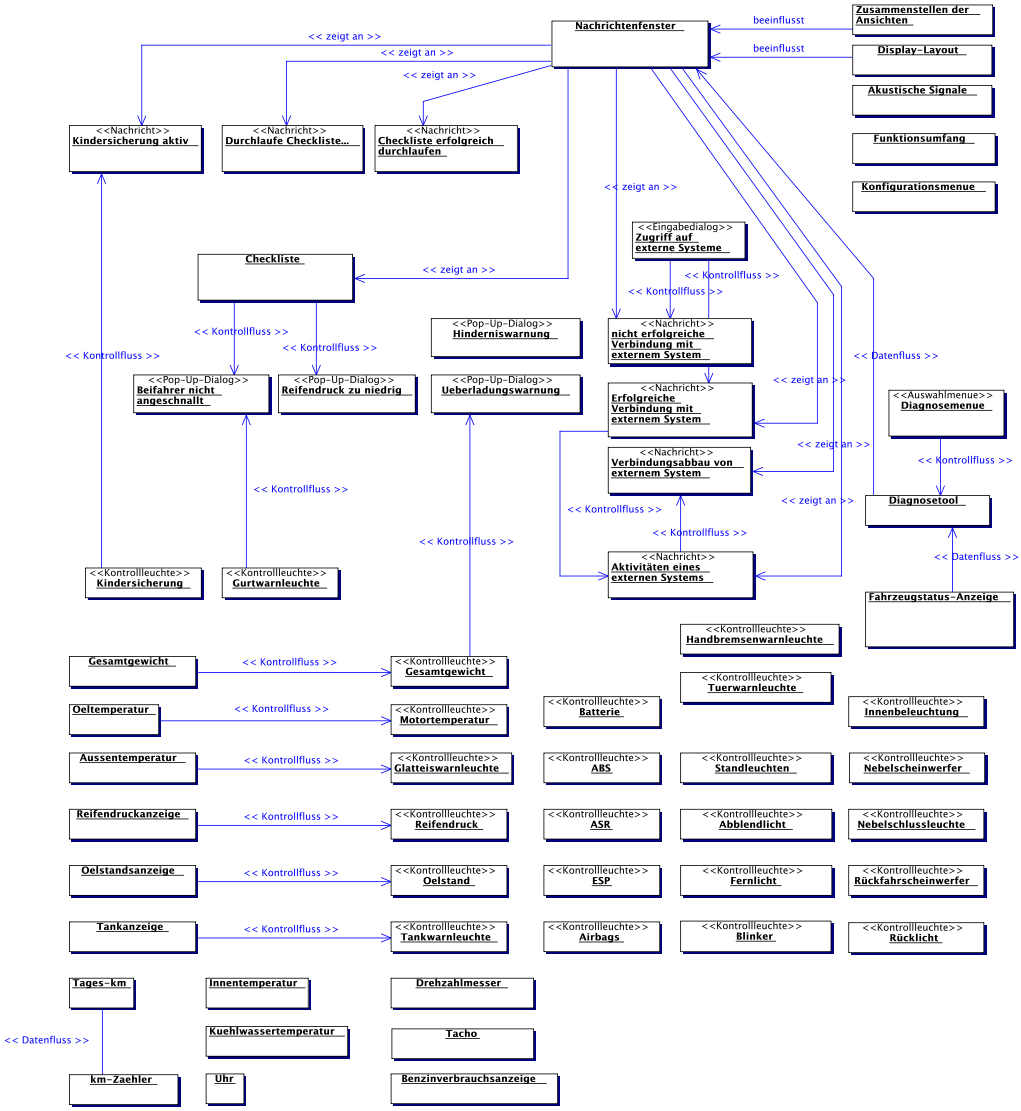

Abbildung 17.1: Packen und Beginn einer Fahrt

Im Folgenden soll das Diagramm 17.1 betrachtet werden. Die zentralen Bestandteile der Standansicht sind das Nachrichtenfenster und die Fahrzeugstatusanzeige. Das Nachrichtenfenster zeigt sowohl die Checkliste, die beim Anlassen des Motors durchlaufen wird, als auch alle Nachrichten an. Exemplarisch wurden einige dieser Nachrichten als Medienobjekte in das Diagramm übernommen.

Die Fahrzeugstatusanzeige ist ein Behälterobjekt für alle Anzeigen und Kontrollleuchten, die in der Standansicht sichtbar sein können. Das Konfigurationsmenü hingegen bündelt Einstellungen

zum Displaylayout, zu akustischen Signalen und zum Funktionsumfang sowie das Zusammenstellen der Ansichten. Eine Konfiguration beeinflusst potenziell die Darstellung aller anderen Medienobjekte.

Bezüglich der verschiedenen Anzeigen zum Fahrzeugstatus gibt es einige grundlegende Kontrollflüsse. Hierbei werden komplexe Anzeigen mit zugehörigen Kontrollleuchten in Verbindung gebracht:

Anzeige	Kontrollleuchte
Gesamtgewicht	Gesamtgewicht (Überladung)
Öltemperatur	Motortemperatur
Außentemperatur	Glatteiswarnleuchte
Reifendruckanzeige	Reifendruck
Ölstandsanzeige	Ölstand
Tankanzeige	Tankwarnleuchte

Weiterhin besteht ein Datenfluss zwischen den Anzeigen zum Gesamt- und Tageskilometerstand. Die Gesamtgewicht-Kontrollleuchte kann über einen Kontrollfluss den Pop-Up-Dialog Überladungswarnung auslösen. Das Gleiche gilt für die Gurtwarnleuchte und den Pop-Up-Dialog Beifahrer nicht angeschnallt (exemplarisch) sowie für die Kontrollleuchte für die Kindersicherung und die Nachricht Kindersicherung aktiv.

Weitere Pop-Up-Dialoge können auch von der Checkliste ausgelöst werden, was wieder durch Kontrollflüsse dargestellt wird.

17.3 Pause, Stau, Übernachtung und Tanken

Autoren: *Leonore Dietrich, Oliver Szymanski, Yue Zhang*

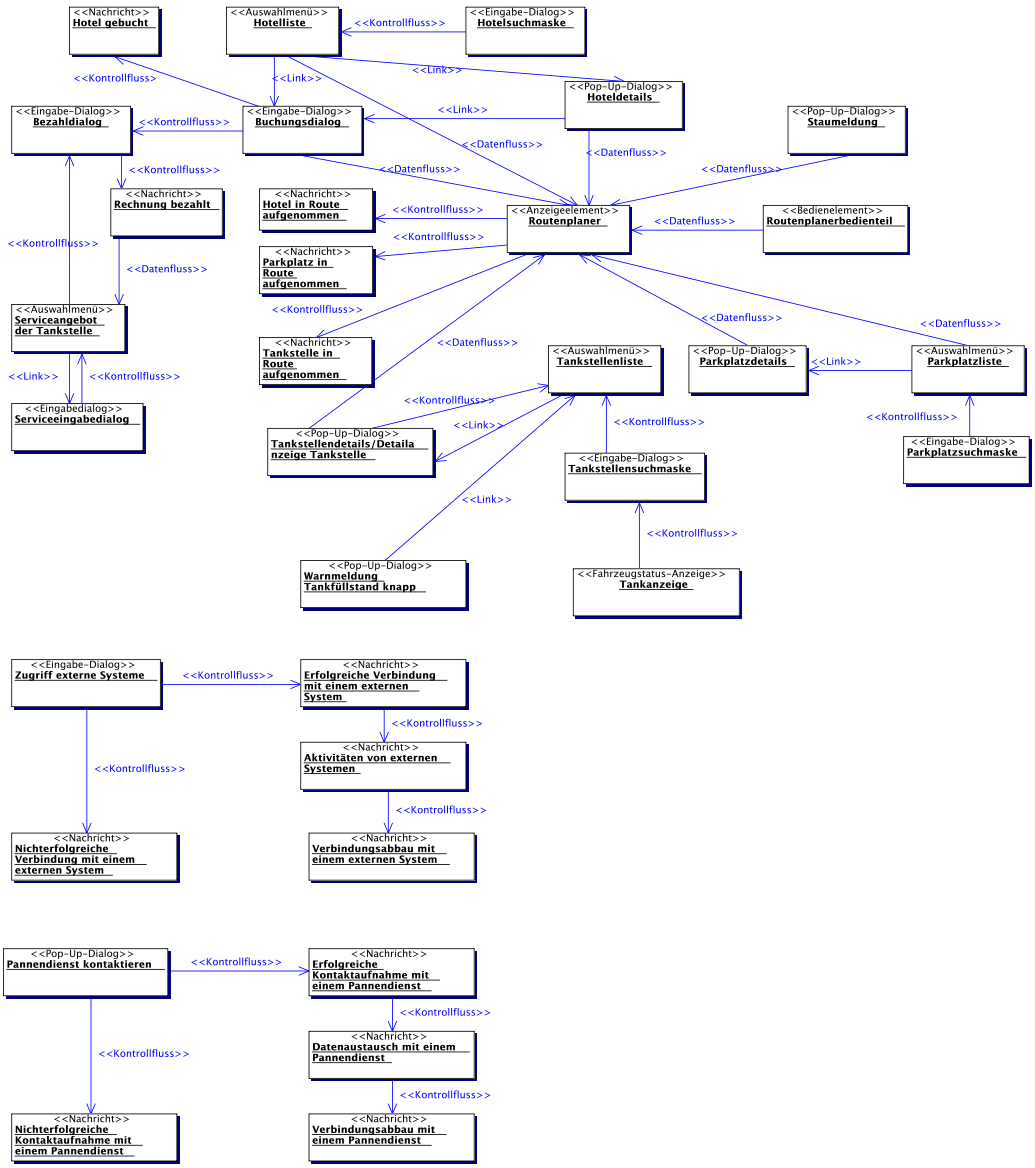

Abbildung 17.2: Pause, Stau, Übernachtung und Tanken

Das in Abbildung 17.2 dargestellte Medienobjektdiagramm beschreibt die Interaktionen zwischen den einzelnen Medienobjekten in den verschiedenen Fahrtansichten. Es teilt sich auf in drei voneinander unabhängige Bereiche, die im Folgenden näher beschrieben werden.

17.3.1 Zugriff auf externe Systeme

Es gibt einen Eingabedialog Zugriff externe Systeme, auf dessen Bestätigung bei erfolgreicher Kontaktaufnahme die Nachricht Erfolgreiche Verbindung und sonst Nichterfolgreiche Verbindung folgt. Nach dem erfolgreichen Verbindungsaufbau erscheinen Nachrichten Aktivitäten von externen Systemen, falls Datenaustausch stattfindet. Hiernach wird im Falle der Trennung der Verbindung entsprechend eine Nachricht Verbindungsabbau angezeigt. Der Kontrollfluss äussert sich in diesem Diagramm also in Form von Abfolgen der einzelnen angezeigten Medienobjekte.

17.3.2 Pannendienst kontaktieren

Der hier beschriebene ablauf ähnelt dem im Abschnitt 17.3.1 beschriebenem. Auch hier handelt es sich um einen Datenaustausch, allerdings in einem anderen Kontext. Der Verbindungsaufbau wird mit einem Pop-Up-Dialog Pannendienst kontaktieren eingeleitet. Auch hier wird bei Bestätigung eine erfolgreiche Verbindung mit der Nachricht Erfolgreiche Kontaktaufnahme und eine nichterfolgreiche Verbindung mit der Nachricht Nichterfolgreiche Kontaktaufnahme quittiert. Werden Daten ausgetauscht, wird die mit der Nachricht Datenaustausch mit dem Pannendienst bekanntegegeben stattfindet. Wie im Abschnitt 17.3.1 wird bei Beendigung der Verbindung die Nachricht Verbindungsabbau angezeigt. Der Kontrollfluss ist hier von derselben Art wie im Abschnitt 17.3.1.

17.3.3 Parkplatz, Hotel, Stau, Routenplaner und Tanken

In diesem Teil des Medienobjektdiagrammes gibt es neben dem Pop-Up-Dialog Staumeldung und dem Bedienelement Routenplanerbedienteil, welche Daten an das Anzeigeelement Routenplaner übermitteln, die drei Teilszenarien Parkplatz, Hotel und Tanken, deren zentraler Schnittpunkt das Anzeigeelement Routenplaner ist. Daher werden die einzelnen Szenarien im Folgenden näher erläutert.

Parkplatz

Nach dem Bestätigen des Eingabedialoges Parkplatzsuchmaske wird das Auswahlmenü Parkplatzliste angezeigt. Hierauf können entweder Parkplatzdetails als Pop-Up-Dialog angezeigt oder durch direkte Auswahl eines Parkplatzes die Route geändert und somit Daten an das Anzeigeelement Routenplaner gesendet werden. Letzteres kann sich auch an die Anzeige der Parkplatzdetails anschließen. Bei Änderung der Route wird eine entsprechende Nachricht Parkplatz in Route aufgenommen angezeigt.

Hotel

Soll nach einem Hotel gesucht werden, kann man den Eingabedialog Hotelsuchmaske aufrufen. Nach Bestätigung wird das Auswahlmenü Hotelliste angezeigt. Detailliertere Informationen werden im Pop-Up-Dialog Hoteldetails präsentiert. Durch Auswahl eines Hotels, entweder aus der Hoteliste oder aus den Hoteldetails, wird die Route geändert. Dadurch müssen Daten an

das Anzeigeelement Routenplaner übergeben werden. Diese Änderung wird durch eine entsprechende Nachricht Parkplatz in Route aufgenommen angezeigt. Sowohl von dem Auswahlmenü Hotelliste als auch vom Pop-Up-Dialog Hoteldetails aus kann man über einen Link zum Eingabedialog Buchungsdialog gelangen. Von dort aus wird über die Bestätigung des Eingabedialoges Buchungsdialog die Route geändert, die Daten an das Anzeigeelement Routenplaner gesendet und eine entsprechende Nachricht Parkplatz in Route aufgenommen angezeigt. Darüber hinaus wird die Nachricht Hotel gebucht bei erfolgreicher Buchung angezeigt. Optional kann sich dem Eingabedialog Buchungsdialog der Eingabedialog Bezahldialog anschließen. Von dort aus wird nach Ausführung der Transaktion die Nachricht Rechnung bezahlt angezeigt.

Tanken

Ausgangspunkt unserer Betrachtung sind die Medienobjekte Tankanzeige und Warnmeldung Tankfüllstand knapp. Die Tankestellenliste kann durch zwei Ereignisse angezeigt werden. Entweder wird die Fahrzeugstatusanzeige Tankanzeige auf dem Tastbildschirms berührt(Link). Es erscheint der Eingabedialog Tankstellensuchmaske. Hat der Fahrer die entsprechenden Eingaben gemacht und bestätigt, wird vom System eine den Eingaben gemäße Liste von Tankstellen berechnet und im Auswahlmenü Tankstellenliste angezeigt (Kontrollfluss). Oder der Pop-Up-Dialog Tankfüllstand knapp erscheint, wenn der Tanksensor feststellt, dass die Benzinreserven nahezu aufgebraucht sind. Bestätigt der Fahrer diese Warnmeldung, so berechnet das System eine Liste der nächstgelegenen Tankstellen und zeigt diese im Auswahlmenü Tankstellenliste an (Kontrollfluss). Dem Fahrer bleibt es nun überlassen eine Tankstelle aus dem Auswahlmenü auszuwählen. Im Falle der Auswahl einer bestimmten Tankstelle öffnet sich der Pop-Up-Dialog Tankstellendetails (Link). Innerhalb des Dialogs werden nähere Informationen zu der gewählten Tankstelle angezeigt. Der Fahrer hat nun die Möglichkeit die Wahl der Tankstelle zu bestätigen oder zu verwerfen. Durch Verwerfen der Wahl schließt sich der Pop-Up-Dialog Tankstellendetails (Kontrollfluss) und dem Fahrer wird erneut die Möglichkeit gegeben, eine Tankstelle aus der Tankstellenliste auszuwählen. Durch Bestätigung der Wahl schließt sich der Pop-Up-Dialog Tankstellendetails und der Routenplaner erhält die Koordinaten der gewählten Tankstelle (Datenfluss). Der Routenplaner berechnet die Route unter Einbeziehung dieser Koordinaten neu und lässt die Nachricht Tankstelle in Route aufgenommen anzeigen (Kontrollfluss).

Nun zu einer anderen Situation. Der Einstiegspunkt im Diagramm ist das Auswahlmenü Serviceangebot der Tankstelle. Sobald eine drahtlose Verbindung zwischen der Bedienstation einer Tankstelle und dem Cockpit hergestellt wird, öffnet sich das Auswahlmenü Serviceangebot der Tankstelle. Der Insasse kann nun einen Servicebereich wie z.B. Tanken, Autowäsche, Wartung oder Einkauf wählen, woraufhin ein entsprechender Eingabedialog Serviceeingabedialog angezeigt wird (Link). In diesem Eingabedialog kann der Insasse den gewünschten Service näher spezifizieren. Schließt er den Serviceeingabedialog, werden die vorhandenen Eingaben gespeichert und die Kontrolle geht wieder auf das Auswahlmenü Serviceangebot der Tankstelle über. Es besteht nun erneut die Möglichkeit, einen Servicebereich auszuwählen. Wird kein weiterer Service gewünscht, kann die Wahl der Serviceleistungen bestätigt werden, was zur Folge hat, dass schließlich der Eingabedialog Bezahldialog geöffnet wird (Datenfluss). Bei Eingabe der notwendigen und hinreichenden Daten sowie der Bestätigung der Rechnung, wird der Eingabedialog Bezahldialog geschlossen, die Nachricht Rechnung angezeigt (Kontrollfluss) und die Kontrolle geht wieder auf das Auswahlmenü Serviceangebot der Tankstelle über.

17.4 Panne

Autoren: *Tobias Wolf, Evgenij Golkov*

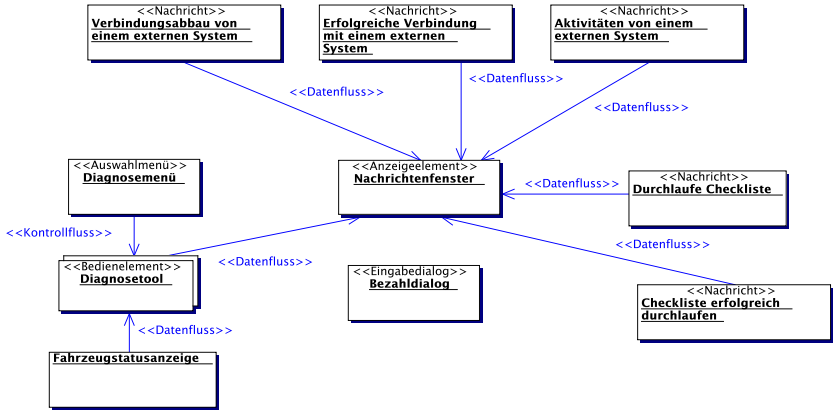

Abbildung 17.3: Panne

Das zentrale Bestandteil der Ansicht Panne ist das Nachrichtenfenster, welches die durch andere Objekte dieser Ansicht gelieferten Daten darstellt (siehe Abbildung 17.3).

Zu einem werden im Nachrichtenfenster Informationen über eine erfolgreiche Verbindung mit einem externen System dargestellt, genauso wie Informationen über den Abbau einer solchen Verbindung. Auch Aktivitäten von einem externen System können Daten an das Nachrichtenfenster senden. Desweiteren bestehen Datenflüsse zwischen der Checkliste und dem Nachrichtenfenster. Beim Durchlaufen der Checkliste wird eine Nachricht an das Nachrichtenfenster geschickt und in diesem angezeigt, und ebenso eine Nachricht über das erfolgreiche Durchlaufen der Checkliste.

Ein weiteres wichtiges Bestandteil der Ansicht ist das Diagnosetool. Das Diagnosemenü wird bedient, um zu einem Diagnosetool zu gelangen. So enstehen Kontrollflüsse zwischen dem Objekt “Diagnosemenü” und dem Diagnosetool. Die Fahrzeugstatusanzeige liefert die vom Diagnosetool benötigten Daten.

Als letztes Objekt wird das Bezahldialog erwähnt. Dieses erscheint auf Zahlungsaufforderung seitens Benutzers oder eines externen Systems.

17.5 Medienobjektdiagramm Unterhaltung

Autoren: *Stefan Borggraefe, Bastian Krol*

Abbildung 17.4 zeigt die für den Bereich Unterhaltung wichtigen Medienobjekte. Zentral dabei ist das Medienobjekt „Unterhaltungsmedienfenster“, das den Bereich eines Displays, das zur Darstellung von Unterhaltungsmedien dient, repräsentieren soll. Das Unterhaltungsmedienfenster wird durch das Medienobjekt „Unterhaltungsmediennavigation“ beeinflusst. Darunter kann man sich Bedienelemente zur Steuerung der Medienwiedergabe vorstellen (z.B. Vorlauf, Rücklauf, Pause, Stopp). Die “Unterhaltungsmedieninformationsanzeige” ist durch einen Datenfluss mit dem Unterhaltungsmedienfenster verbundenen. Sie zeigt während der Wiedergabe Informationen über das gewählte Unterhaltungsmedium an (z.B. Name des derzeit spielenden Titels, Länge eines Titels usw.).

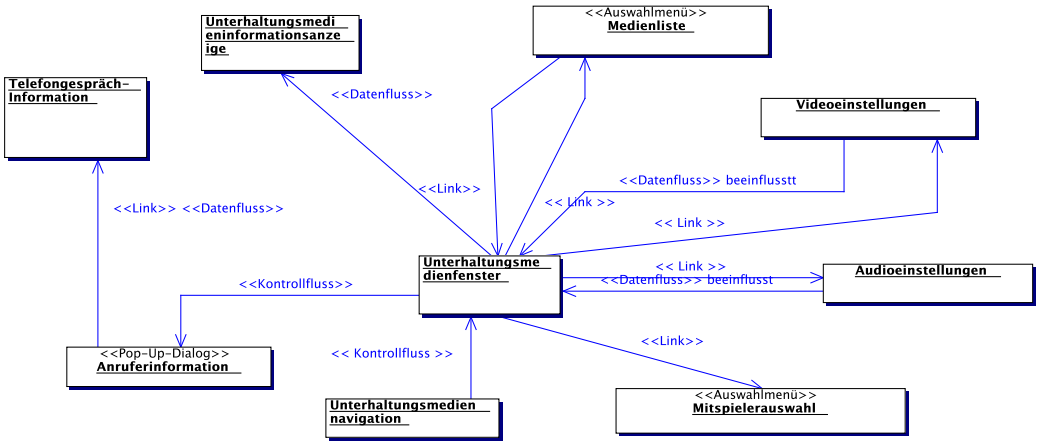

Abbildung 17.4: Medienobjektdiagramm Unterhaltung

Vom Unterhaltungsmedienfenster aus kann ein Benutzer die Audio- und Videoeinstellungen erreichen. Diese sind im Medienobjektdiagramm als gleichnamige Medienobjekte dargestellt. Zur Auswahl eines Unterhaltungsmediums kann der Benutzer eine „Medienliste“ benutzen. Die Wahl eines Mitspielers für das Unterhaltungsmedium Videospiel kann über den Link zum Medienobjekt „Mitspielerauswahl“ erfolgen.

Trifft während der Wiedergabe eines Mediums ein Telefonanruf ein, wird der Kontrollfluss vom Unterhaltungsmedienfenster an den Pop-Up-Dialog „Anruferinformation“ weitergegeben. Durch Bedienung dieses Dialogs kann der Benutzer einen eingehenden Anruf annehmen oder ablehnen. Um die Entscheidungsfindung zu unterstützen wird dabei die Telefonnumer des Anrufenden eingeblendet. Nimmt der Benutzer den Anruf an, tritt das Medienobjekt „Telefongesprächinformation“ in Erscheinung, das während eines Telefonats solche Informationen wie die Gesprächsdauer und die Telefonnummer des Gesprächspartners bereit hält.

17.6 Gesamtmedienobjektdiagramm

Autoren: *Ulf Schellbach, Bastian Krol*

Beim Gesamtmedienobjektdiagramm handelt es sich im Wesentlichen um die Zusammenführung der Medienobjektdiagramme aus den Abschnitten 17.2, 17.3 und 17.5.

Bei der Zusammenführung der einzelnen Medienobjektdiagramme wurde folgendermaßen vorgegangen: Zunächst wurden alle Medienobjekte der Einzeldiagramme mitsamt ihren Beziehungen übernommen. Dabei ergeben sich natürlicherweise gerade die Medienobjekte, die in mehreren Einzeldiagrammen auftauchen als Verknüpfungspunkte.

Zusätzlich wurden neue Beziehungen zwischen Medienobjekten entdeckt. Ein Beispiel hierfür ist sind die Beziehungen zwischen den Medienobjekten Tankanzeige, Tankwarnleuchte und Tankfüllstand knapp. Bei knappen Benzinreserven wird die Tankanzeige zum Auslöser für das Aufblinken der Tankwarnleuchte (Kontrollfluss). Diese wiederum verursacht die Anzeige des Pop-Up-Dialogs Tankfüllstand knapp (Kontrollfluss).

Da wir uns des Weiteren bei den Beziehungen zwischen Medienobjekten auf solche beschränken wollten, die mit den Stereotypen Kontrollfluss, Datenfluss und Link bezeichnet werden können, wurden die zeigt-an-Beziehungen, sowie die Teil-von-Beziehungen nicht aus den Einzeldiagrammen übernommen. Weiterhin wurden zwei neue Medienobjektstereotypen, Anzeige und Fenster, identifiziert.

Die zeigt-an-Beziehungen, die in den Einzeldiagrammen zwischen dem Nachrichtenfenster und den verschiedenen Nachrichten bestanden, wurden durch Datenfluss-Beziehungen vom Erzeuger der Nachricht, der beispielsweise das Diagnosetool sein kann, zum Nachrichtenfenster ersetzt.

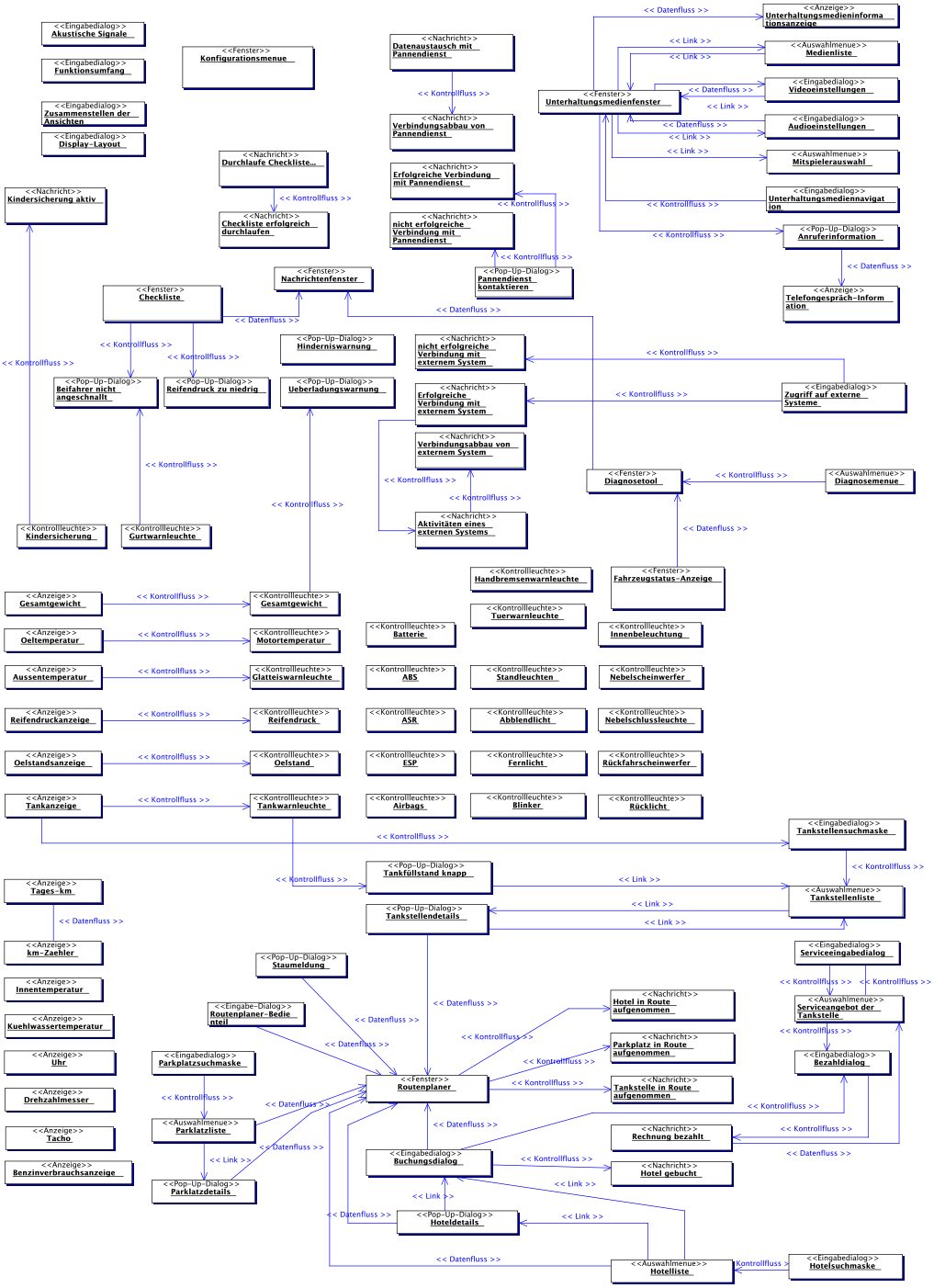

Abbildung 17.5: Gesamtmedienobjektdiagramm

Kapitel 18

Fazit zum ersten Semester

18.1 Fazit

Autoren: *Bastian Krol*
Daniel Mölle

Das schrittweise Vorgehen in der Entwurfsphase entspricht der üblichen Methodik des Entwurfs objektorientierter Programme mit besonderer Ausrichtung auf die beteiligten Medienobjekte und ihre Verknüpfungsstruktur.

18.1.1 Arbeitsschritte

Das anfängliche Zusammentragen vor und während einer Autofahrt denkbarer Abläufe hat wie gewünscht eine grosse Zahl interessanter Anwendungen eines Fahrzeugcockpits geliefert. Das aus diesen Abläufen komponierte Szenario sollte einen Großteil der Funktionalität eines praktisch einsetzbaren Cockpits widerspiegeln und schien somit als Ausgangspunkt für den weiteren Entwurf besonders geeignet zu sein. Ob allerdings alle Kernfunktionen vollständig erfasst worden sind, kann sich selbstverständlich erst später zeigen.

In den Anwendungsfall-Diagrammen wurden die im ersten Schritt erarbeiteten Anwendungen konkretisiert, wobei viele Akteure und Komponenten identifiziert werden konnten. Insbesondere aufgrund der schrittweisen Verfeinerungen wurde ein recht hoher Detaillierungsgrad in den Beschreibungen erreicht. Zusätzlich wurden Komponenten, die von mehreren der fünf Arbeitsgruppen eingeführt wurden, zum Teil vereinheitlicht. Es ist aber auch bezüglich der Anwendungsfälle zur Zeit noch nicht vorherzusehen, ob wirklich alle bzw. viele der praktisch denkbaren Fälle beachtet worden sind.

Beim Übergang zu den Zustandsdiagrammen wurden die Anwendungsfälle bezüglich der Frage untersucht, welche Zustände die beteiligten Anzeigen und Bedienelemente annehmen können. Zu diesem Zeitpunkt begann also bereits die Konzentration auf Medienobjekte. Dabei sollte eine Liste aller bis dahin identifizierten Komponenten die benötigten Namenskonventionen liefern.

Die in den Zustandsdiagrammen angegebenen Transitionen brachten die Definition eines rudimentären Kommunikationsprotokolls zwischen den Medienobjekten mit sich, das sich der üblichen Transitionsattribute, nämlich Ereignisse und Handlungen, bedient. Allerdings stellt sich die Frage, inwiefern dieses Protokoll später Verwendung finden wird, da zum Beispiel viele der Aufgaben des Cockpits einfacher und übersichtlicher zu handhaben sein könnten, wenn zentrale, übergeordnete Steuermodule verwendet würden.

Auch bei den Zustandsdiagrammen wurden die Ergebnisse aus den einzelnen Arbeitsgruppen zusammengefasst, was sich aufgrund der Parallelisierung von Zuständen aber als unproblematisch erwies.

Der nächste Arbeitsschritt war das Erstellen von Medienobjektdiagrammen, in denen die wichtigen Beziehungen zwischen den zuvor identifizierten Objekten herauszustellen waren. Das Zusammenführen der in den einzelnen Arbeitsgruppen erarbeiteten Diagramme erwies sich aufgrund der großen Anzahl von Objekten und Zusammenhängen als schwierig, so dass zunächst auf eine tabellarische Darstellung gewechselt wurde. Aus dieser Tabelle kann vor allem entnommen werden, welche Medienobjekte in welchen Ansichten verboten, verfügbar oder sogar vorgeschrieben sind.

Unter Auslassung bereits dokumentierter und naheliegender Relationen, insbesondere von „enthält“-Beziehungen, konnte anschließend die Zusammenfassung zu einem Gesamtdiagramm erfolgen.

18.1.2 Kritik

Aufgrund der schrittweisen und von dem Szenario ausgehenden Entwicklung ist die Vollständigkeit der Beschreibung keineswegs garantiert. Wie gewöhnlich muss damit gerechnet werden, dass sich in der Implementierungsphase weitere Anwendungsfälle herausstellen, die dann nachträglich einzubringen wären.

Wegen der starken Konzentration auf Medienobjekte (Anzeigen und Bedienelemente) wurden Sensorik und zentrale Steuermodule nicht beachtet, weshalb die modellierten Objekte und Abläufe eventuell autonomer erscheinen, als sie es eigentlich sein sollten.

Problematisch war der Übergang von den Zustandsdiagrammen zu den Medienobjektdiagrammen. In vielen Fällen verleitete die Ableitung der Informationen aus den erstgenannten dazu, die Medienobjekte bzw. die Tatsache, dass sie angezeigt werden, als Zustände aufzufassen, so dass die Kontrollflussrelationen fälschlicherweise als Transitionen betrachtet wurden.

Ohnehin gerieten die Medienobjektdiagramme besonders in die Kritik. So führt z.B. das Verbergen koordinierender Steuermodule dazu, dass Daten- und Kontrollflüsse direkt zwischen den Objekten verlaufen, was in vielen Fällen aber nicht der Arbeitsweise entspricht, die ursprünglich vorgesehen war oder naheliegend ist.

Immense Schwierigkeiten bereitete der Relationen-Stereotyp „Kontrollfluss“. Die Definition über Wechsel der Fokusse (Objekt A überträgt Fokus an Objekt B, das dann mit dem Benutzer interagiert) überdeckt nur die echten Dialoge; es tauchen aber etliche andere, wichtige Medienobjekte auf, bei denen nicht von einem Fokus gesprochen werden kann, da sie gleichzeitig und mit gleicher Priorität angezeigt werden müssen.

Weiterhin existieren globale Zusammenhänge, die sich in Diagrammen diesen Typs nicht ausdrücken lassen, zum Beispiel das Konfigurationsmenü, das alle Medienobjekte beeinflusst, oder die Tatsache, dass viele Objekte in anderen gebündelt werden und insbesondere ansichtsspezifisch sein können. Die aus den Diagrammen ersichtliche Kommunikationsstruktur ist also alles andere als vollständig, und es bleibt die Frage, wie nützlich das zusammengefasste Gesamtdiagramm bei der Implementierung sein wird.

Teil III

Zweite Seminarphase

Kapitel 19

Das Avalon-Komponentenmodell

Autoren: *Evgenij Golkov*
Daniel Mölle

In diesem Kapitel soll das Avalon-Komponentenmodell vorgestellt und bezüglich softwaretechnologischer Gesichtspunkte analysiert werden. Dazu werden im ersten Abschnitt grundlegende theoretische Aspekte beleuchtet. Im zweiten Abschnitt wird auf die wesentlichen Schnittstellen des Avalon-Frameworks und auf die Lebenszyklen von Komponenten eingegangen. Der dritte und letzte Abschnitt dient der Gegenüberstellung der Vor- und Nachteile des Avalon-Komponentenmodells und endet mit einer abschließenden Bewertung.

19.1 Das Avalon-Komponentenmodell

In diesem Abschnitt wird zunächst auf die Entwicklungsgeschichte und die Motivation des Modells eingegangen sowie eine Einordnung in die Kategorien der Softwaretechnologie vorgenommen. Darauf aufbauend werden die zugrundeliegenden Entwurfsprinzipien und die daraus resultierenden Eigenschaften des Avalon-Frameworks beschrieben. Anschließend wird die komponenten-orientierte Programmierung – als Paradigma – mit ihren Konsequenzen bezüglich des Programmierstils vorgestellt.

Dieser Kapitel basiert auf dem Artikel „Developing With Apache Avalon“ von Berin Loritsch, dem ofiziellen Developer's Guide von Apache Avalon Project.

19.1.1 Einleitung

Anfangs handelte es sich bei Avalon um ein Java Apache Server Network. Mit der Zeit wurde dieses Projekt in mehrere kleinere Unterprojekte unterteilt. Um die wichtigsten zu nennen, sind es:

Framework: Die Basis für alle unter dem Avalon-Namen laufenden Projekte. Hier werden Schnittstellen, Verträge sowie die Standardimplementierungen definiert.

Excalibur: Eine Ansammlung von serverseitigen Komponenten, die Entwickler in ihren eigenen Projekten benutzen können. Hierzu zählen unter anderen Poolingimplementierungen, Datenbankverbindungsmanagement sowie Komponentenmanagementimplementierungen.

LogKit: Ein Hochgeschwindigkeitswerkzeug zur Erstellung von Logs.

Phoenix: Der Serverkernel, der das Deployment und die Ausführung von Diensten steuert.

Cornerstone: Eine Ansammlung von Diensten, die in Avalon Phoenix verwendet werden können.

Zusammenfassend kann man Avalon als ein Framework bezeichnen, das sowohl Schnittstellen als auch Standardimplementierungen beinhaltet.

Das Fokus von Avalon ist die serverseitige Programmierung und vereinfachte Wartung und Design von auf Server ausgerichteten Projekten. Bei Avalon spricht man von einem horizontalen Framework, also von einem generischen Framework, das in mehreren Anwendungsbereichen benutzt werden kann, im Gegensatz zu einem vertikalen Framework, welches immer auf einzelne Aufgabenbereiche ausgerichtet ist.

Avalon ist kein universelles Tool. Einerseits ist es zwar gut für Projekte geignet, die in einer Serverumgebung laufen und / oder Dienste zur Verfügung stellen. Andererseits sollte Avalon in Anwendungsbereichen nicht eingesetzt werden, für die bereits spezielle Frameworks existieren (zum Beispiel Swing/AWT for GUIs), sowie in reinen Clientanwendungen, bei denen keine Interaktion stattfindet und die keine Dienste bereitstellen.

19.1.2 Besondere Eigenschaften

In diesem Kapitel werden die wichtigsten Eigenschaften beschrieben, die Avalon auszeichnen. Neben solchen Aspekten wie Inversion of Control und Separation of Concerns wird auch gleichzeitige Dienst- und Komponentenorientirung von Avalon vorgestellt.

An dieser Stelle müssen zwei grundlegende Begriffe definiert werden, die für dieses Dokument von zentraler Bedeutung sind:

Definition: Eine Komponente ist eine Kombination einer Schnittstelle und der Implementierung dieser Schnittstelle.

Definition: Ein Dienst (engl. Service) ist eine Gruppe aus einer oder mehreren Komponenten, die eine komplette Lösung bereitstellen-

Inversion of Control

Zu einer der wichtigsten Eigenschaften von Avalon zählt die Inversion of Control (Umkehrung des Kontrollflusses). Diese Umkehrung wird duch folgende Merkmale charakterisiert:

- Eine Komponente kann immer extern gesteuert werden. Hier bestimmt also der Benutzer das Verhalten der Komponente.
- Eine Komponente bekommt alle Daten von aussen.
- Jede Lebensphase einer Komponente wird durch den Code bestimmt, der die Komponente krieiert hat.

Durch die Umkehrung des Kontrollflusses wird eine sichere Methode für die Interaktion zwischen Komponent zur Verfügung gestellt.

Separation of Concerns

Eine weitere wichtige Eigenschaft von Avalon ist Separation of Concerns (Trennung von Aufgabebereichen). Das bedeutet, dass die (auch voneinander abhängigen) Aufgabebereiche separat betrachtet werden. Als Beispiel für verschiedene Aufgabenbereiche bei einem Webserver könnte man unter anderem Sicherheit, Stabilität, Steuerbarkeit und Konfigurierbarkeit nennen.

Component Oriented Programming

Bei der Entwicklung mit Avalon setzt man neben dem weiter unten beschriebenen dienstorientierten Ansatzes die komponentenorientierte Programmierung ein. Dabei wird das System in Komponenten geteilt, wonach Verträge (engl. Contracts) für jede Komponente definiert werden. Zu einer der wichtigsten Eigenschaften bei dieser Vorgehensweise gehört die Möglichkeit, Implementierungen zu ersetzen, ohne dabei andere Teile des Systems ändern zu müssen. Vergleicht man den komponentenorientierten Ansatz mit dem objektorientierten, stellt sich heraus, daß sich der komponentenbasierte Ansatz durch einen geringeren Abhängigkeitsgrad auszeichnet, wodurch eine bessere Wiedereinsetzbarkeit des Codes resultiert und ein einfacheres Management des Systems ermöglicht wird. Zu weiteren Vorteilen des komponentenorientierten Ansatzes gehört zu einem die Möglichkeit, Teile des Projektcodes zu modifizieren, ohne dass das gesamte System dadurch gefährdet wird, und zum anderen die Möglichkeit, verschiedene Implementierungen einer Komponente zur Laufzeit zu wählen.

Service Oriented Programming

Ein weiterer bei der Entwicklung von Avalonprojekten eingesetzer Programmieransatz ist die Dienstorientierte Programmierung (engl. Service Oriented Programming). Dieser Ansatz zeichnet sich dadurch aus, dass das System in die von diesem System anzubietende Dienste geteilt wird. Ein Dienst wird dabei als Schnittstelle betrachtet, und eine Implementierung dieses Dienstes als Block bezeichnet. Ein Server wird durch mehrere Dienste bestimmt. Als Beispiel kann man hier den Mail-Server mit etnsprechenden Protokollen wie die Authorizierung und Authentifizierung, der Administrationsdienst sowie der grundlegene Dienst zur Behandlung von E-Mails nennen.

19.1.3 Zerlegung des Systems bzw. Systemanalyse

Damit eine effiziente Lösung, die auf dem komponentenorientierten Ansatz basiert, entwickelt werden kann, muss das System in Komponenten zerlegt werden. Dazu ist eine Systemanalyse notwendig, die nach der Top-Down-Vorgehensweise folgende Phasen durchläuft:

1. Bestimmung der Projektziele: In dieser Phase wird betrachtet, was die eigentliche Ziele des Projekts sind. Bei den kommerziellen Produkten werden diese Ziele in der Regel in einem Statement of Work festgehalten, bei Open Source Projekten geschieht es oft durch ein Brainstorming.

2. Auffinden der Dienste: In dieser Phase werden die Dienste aus der Projektdefinition abgeleitet. Dabei werden Dienste in explizite (vom Statement of Work abgeleitete) und implizite (einzelne Aufgaben erfüllende oder explizite Dienste unterstützende) unterteilt. Desweiteren wird hier eine Entscheidung getroffen, welche Dienste entwickelt und welche gekauft werden ('make or buy'-Entscheidung).

3. Auffinden der Komponenten: Nachdem die Projektziele definiert und daraus die zu bereitstellende Dienste abgeleitet wurden, werden benötigte Komponenten beschrieben. Hierbei wird auf das Konzept der Rollenverteilung (engl. casting) gegriffen. Dieses Konzept wird im zweiten Teil dieser Arbeit erläutert, an dieser Stelle wird es nur kurz vorgestellt. Bei dem Konzept der Rollenverteilung werden die Komponenten den Akteuren (engl. actors), die Aufgabenbereiche den Rollen (engl. roles) und die Interaktion dem Drehbuch (engl. script) gleichgestellt.

19.2 Framework und Vertragssystem

In diesem Abschnitt werden zunächst die Schnittstellen, die die Grundlagen des Avalon-Frameworks ausmachen, vorgestellt, wobei auch der Begriff des Vertrages herangezogen wird. Danach wird die Beschreibung des Vertragssystems mit dessen wichtigstem Bestandteil, nämlich den sogenannten Lebenszyklen von Komponenten, fortgesetzt. Abschließend werden einige Auswirkungen der Verwendung von Avalon anhand eines Beispielprogramms demonstriert.

Die hier verwendeten Informationen wurden zum Großteil [30] entnommen.

19.2.1 Schnittstellen

Obwohl der Kern des Avalon-Frameworks lediglich aus einer überschaubaren Anzahl einfach gehaltener Schnittstellen besteht, hat die Anwendung des Komponentenmodells enorme Auswirkungen auf die Art und Weise, in der Programme zusammengesetzt werden. Der Grund dafür ist, dass den Schnittstellen eine Bedeutung inneliegt, die über das übliche Maß hinausgeht. Wie so oft im Zusammenhang mit objektorientierter Software-Entwicklung bietet sich auch zur Erklärung dieser Tatsache die Verwendung der Vertragsmetapher an.

Im Normalfall ist der Vertrag, der einer Schnittstelle zugrundeliegt, schlichtweg der folgende: In einer konkreten Klasse, die diese Schnittstelle implementiert, müssen auch alle in der

Schnittstelle deklarierten Methoden definiert werden. Diese Vereinbarung ist so trivial, dass ihre Einhaltung sogar vom Java-Compiler überprüft werden kann. Ob die derart definierten Methoden allerdings auch semantisch das leisten, was die Autoren der Schnittstelle ursprünglich vorsahen, entzieht sich im Allgemeinen jeder automatisierbaren Kontrolle; man bedenke hierzu beispielsweise die Nichtrekursivität des Halteproblems.

Auch die Schnittstellen innerhalb des Avalon-Frameworks bringen Verträge mit sich, die sich auf das Verhalten der sie implementierenden Klassen beziehen. Die Sicherstellung der durchgängigen Erfüllung dieser Verträge obliegt allein den Software-Entwicklern.

Komponenten

Einleitend soll zunächst die grundlegende und gleichzeitig einfachste Schnittstelle des Frameworks betrachtet werden. Sie hat erwartungsgemäß den Namen Component und ist, was bei der ersten Betrachtung verblüffend sein kann, durch den folgenden Java-Code gegeben:

```
public interface Component
{
}
```

Im Gegensatz zu der extremen Einfachheit dieser Schnittstelle ist der zugehörige Vertrag bereits sehr spezifisch:

- Jede Komponente besteht aus einer **Arbeitsschnittstelle** (*working interface*) und ihrer **Implementierung** in genau einer entsprechenden Klasse.
- Jeder Komponente wird eine **Rolle** zugewiesen. In Analogie zu einer Rolle innerhalb eines Theaterstücks bestimmt die Rolle den Aufgabenbereich und den Handlungsspielraum der Komponente.
- Eine Komponente wird grundsätzlich nicht über den Konstruktor, sondern über spezielle Methoden mit Konfigurationsdaten bzw. Parametern versorgt. Der **Konstruktor** hat also **keine Argumente**.
- Jede Komponente unterliegt einem **Lebenszyklus** mit Vorgaben darüber, in welcher Reihenfolge bestimmte Methoden aufgerufen werden dürfen bzw. müssen.

Die Arbeitsschnittstelle und die Rolle einer Komponente sind eng miteinander verknüpft; Sie sollten einander sogar eindeutig zugeordnet sein. Deshalb wird die Rolle durch einen String namens ROLE festgelegt, der bereits Bestandteil der Arbeitsschnittstelle – und nicht etwa der Implementierung – ist. Für die Benennung der Rollen gelten dieselben Konventionen wie für die Benennung von Paketen, wie das folgende Beispiel veranschaulicht:

```
import org.apache.avalon.framework.component.*;

public interface PrintJobSpooler extends Component
```

```
{
  String ROLE = "de.uni-dortmund.cs.ls10.hycop.printjobspooler";

  public PrintJobId  enqueue (PrintableDocument doc);
  public void        dequeue (PrintJobId id);
}
```

Schnittstellen für spezielle Komponententypen

Der Begriff der Komponente ist natürlich sehr allgemein, so dass die Deklaration einer Klasse als Avalon-Komponente, von den oben beschriebenen Vertragspunkten abgesehen, beinahe als bedeutungslos erscheinen mag. Allerdings stehen weitere Schnittstellen zur Verfügung, mit denen sich zentrale Eigenschaften von Komponenten andeuten lassen. Auch hier gilt, dass die Schnittstellen äußerst schlank, aber mit weitreichenden Verträgen verwoben sind.

Protokollierung, Initialisierung und Verwurf

Bei vielen Komponenten ist eine fortlaufende Protokollierung der in ihnen ablaufenden Vorgänge hilfreich, wenn nicht sogar zwingend erforderlich. Dies gilt insbesondere für Komponenten, die vollständige Dienste zur Verfügung stellen, wie z.B. einen Web-Server. Im Avalon-Framework gibt es für diesen Zweck die Schnittstelle LogEnabled, wobei die Protokollierung vertragsgemäß durch sogenannte Logger-Objekte erfolgt.

Wenn eine Komponente nach ihrer Konfiguration noch weiterer Initialisierungsschritte bedarf, so sollte die Schnittstelle Initializable verwendet werden. Beispielsweise könnte ein Gerätetreiber, dessen Konfiguration abgeschlossen ist, an dieser Stelle die Verbindung zum entsprechenden Gerät überprüfen bzw. herstellen. Die einzige in dieser Schnittstelle geforderte Methode heißt initialize und hat keine Argumente, was auch naheliegt, da die Komponente zu diesem Zeitpunkt als vollständig konfiguriert gilt.

Dual zum Initialisieren verläuft das Verwerfen von Komponenten. Die zugehörige Schnittstelle heißt Disposable und deklariert die argumentfreie Methode dispose. Das explizite Verwerfen ist insbesondere dann sinnvoll, wenn die Komponente Resourcen belegt, die nach ihrer Zerstörung nicht automatisch freigegeben würden.

Konfiguration

In vielen Fällen reicht es aus, die Konfiguration einer Komponente anhand von Paaren aus Parameternamen und -werten durchführen zu können. Für diesen Zweck kann die Schnittstelle Parameterizable verwendet werden. Diese fordert eine Methode parameterize mit einem entsprechenden Argument.

Bei Komponenten, deren Konfiguration komplexer ausfällt, kann alternativ – aber nicht gleichzeitig – die Schnittstelle Configurable eingesetzt werden. Analog zum obigen Fall wird dort eine Methode configure deklariert, die als Argument ein Objekt vom Typ Configuration erhält. Der wesentliche Unterschied zu Parameters besteht darin, dass Configuration lediglich eine Schnitt-

stelle ist. Im Wesentlichen dient sie also nur zur Standardisierung der Namen der Methoden, mit denen die Konfigurationsdaten abgefragt und manipuliert werden können.

Container und Komponenten

Wie bei den oben angesprochenen Konfigurationsdaten sollte es auch im Allgemeinen möglich sein, komplexe Daten, wie sie z.B. in Container-Objekten vorliegen, an Komponenten zu übergeben. Im Avalon-Komponentenmodell wird ein solches Datenbündel als *Kontext* bezeichnet. Die zugehörige Schnittstelle trägt den Namen Context und deklariert eine get-Methode, über die das Extrahieren von Kontextinformationen anhand von Schlüsseln ermöglicht werden soll. Die einzige konkrete Vertragsbindung ist hierbei die, dass der Kontext nur gelesen, aber nicht verändert werden darf. Komponenten, die einen Kontext aufnehmen können, sollten die Schnittstelle Contextualizable implementieren. Dazu muss die Methode contextualize definiert werden, die als Argument den einzubettenden Kontext erhält.

Andererseits sind auch Objekte denkbar, die in dem Sinne Container darstellen, dass sie aus anderen Komponenten zusammengesetzt werden. Für diesen Fall stellt das Avalon-Framework die Schnittstelle Composable zur Verfügung. Eine solche Konstruktion wäre mit dem bekannten Entwurfsmuster Kompositum (*composite*) bedingt vergleichbar, weil die Komposition rekursiv fortführbar ist. Allerdings existiert ein gravierender Unterschied, da nämlich der Container nicht zwangsweise selbst eine Komponente sein muss.

Unbedingt zu beachten ist, dass die Adressierung der enthaltenen Komponenten innerhalb des ComponentManager-Objekts ausschließlich über deren Rollen verläuft. Deshalb sollten nur Komponenten mit paarweise verschiedenen Rollen aufgenommen werden. Wenn hingegen mehrere Komponenten der gleichen Rolle zusammengefasst werden müssen, so sollten sie stattdessen durch einen ComponentSelector verwaltet werden. Dann wird die Adressierung über Schlüsselobjekte (*hints*, also Hinweise auf die Identität der gewünschten Komponente) beliebigen Typs vorgenommen.

Wie auch bei der Konfiguration zeigen sich hier sehr deutlich die Auswirkungen des Entwurfsprinzip *inversion of control*. So ist es etwa nicht die Aufgabe eines Composable-Objekts, die in ihm enthaltenen Komponenten festzulegen. Vielmehr übergibt der Aufrufer ein Manager-Objekt, für das allein er zu entscheiden hat, welche Komponenten es enthalten soll. Das Verhalten des Composable-Objekts wird also ganz wesentlich vom Aufrufer kontrolliert.

Nebenläufigkeit

Nebenläufige Prozesse werden im Avalon-Framework aus zweierlei Blickwinkeln berücksichtigt. Einerseits kann eine Komponente nebenläufiger Natur sein, was insbesondere dann gilt, wenn die Komponente zur Kapselung eines ganzen Dienstes verwendet wird. Für derartige Komponenten gibt es die Schnittstelle Startable, in der lediglich die beiden argumentfreien Methoden start und stop deklariert werden. Der mit diesen Methoden verbundene Vertrag ist einfach, aber streng: Die Komponente darf nur einmal, und zwar nach ihrer Initialisierung, gestartet werden; Danach darf sie höchstens einmal gestoppt werden, woraufhin sie nur noch verworfen werden dürfte.

Soll die Arbeit der Komponente zwischenzeitlich unterbrochen werden können, so bietet sich die zusätzliche oder auch alleinige Verwendung der Schnittstelle Suspendable an. Analog zu start und stop fordert sie die beiden argumentlosen Methoden resume und suspend. Der Unterschied ist, dass diese beliebig oft aufgerufen werden dürfen, allerdings nur zwischen der Initialisierung und dem Verwurf der Komponente.

Andererseits kann es erforderlich sein, den parallelen Zugriff auf eine Komponente durch nebenläufige Prozesse zu regulieren, wie dies z.B. durch das Attribut synchronized für Methoden in Java möglich ist. Hierzu gibt es im Avalon-Framework sogenannte Markierungsschnittstellen, die sich – genau wie im Falle von Component – dadurch auszeichnen, dass sie keine Methoden deklarieren. Das Java-Typsystem wird hier also nur dazu verwendet, bestimmte Informationen über das Verhalten der Komponenten unter nebenläufigen Prozessen festzuhalten.

19.2.2 Der Lebenszyklus

Ein ganz wesentlicher Bestandteil der Verträge im Avalon-Framework besteht in der Regulierung der zeitlichen Abfolge von Methodenaufrufen. Während einige der damit verbundenen Forderungen naheliegend sind, wurden andere mit dem Ziel aufgestellt, eine höhere Transparenz und die Gültigkeit bestimmter Eigenschaften, die das Programmieren von Komponenten erleichtern, zu erreichen. So ist beispielsweise leicht einzusehen, dass das Verwerfen einer Komponente der letzte Aufruf einer Methode dieser Komponente sein muss, aber nicht unmittelbar klar, wieso die Komposition grundsätzlich vor der Konfiguration erfolgen soll.

Der Lebenszyklus unterteilt sich in drei Phasen: die Initialisierung, nach deren Abschluss eine vollständig konfigurierte und arbeitsfähige Komponente vorliegen soll, die „Dienstzeit“, in der die Komponente genutzt werden kann, und die Destruktion, nach der die Komponente nicht mehr zur Verfügung steht.

Die Initialisierung darf nur einmal durchlaufen werden. Der zugehörige Vertrag besagt hierbei unter anderem, dass eine eventuelle Protokollierung sofort nach der Instanziierung einer Komponente aktiviert werden muss, dass die Initialisierung erst nach der vollständigen Konfiguration erfolgen darf und dass das Starten von komponenteneigenen Prozessen immer der letzte Schritt dieser Phase zu sein hat.

Für die Dienstzeit fällt die Vertragsbindung etwas schwächer aus. Die zentrale Forderung lautet hierbei, dass nebenläufige Prozesse von Komponenten, die die Schnittstelle Suspendable implementieren, unterbrochen werden müssen, bevor eine Rekonfiguration vorgenommen werden darf.

Die Destruktion ist die einfachste der drei Phasen des Lebenszyklus. Hier sind lediglich das Einstellen der Arbeit und das Freigeben der Resourcen anzustoßen, sofern die entsprechenden Schnittstellen, Startable und Disposable, überhaupt eingesetzt wurden.

19.2.3 Beispielprogramm

Das in diesem Abschnitt verwendete Beispiel ist eng an den Themenbereich der Projektgruppe HyCop angelehnt. Betrachtet wird die Standansicht für ein Fahrzeugcockpit, die auf verschiedene Anzeigeelemente zurückgreift und deshalb als Composable implementiert wird.

Jedes Anzeigeelement muss dabei als Avalon-Komponente aufgefasst werden, damit es in die Standansicht aufgenommen werden kann. Dies ist aber unproblematisch, da diese Einstufung ohnehin naheliegt. Die zugehörige Rolle ergibt sich trivial, da ein Anzeigeelement einfach ein HyCop-Anzeigeelement darstellt. In der aus diesen Überlegungen resultierenden Schnittstelle soll weiterhin gefordert werden, dass jedes Anzeigeelement eine draw-Methode hat:

```
import org.apache.avalon.framework.component.*;

public interface Anzeigeelement extends Component
{
  String ROLE = "de.uni-dortmund.cs.ls10.hycop.anzeigeelement";

  public void draw ();
}
```

Nun können Arbeitsschnittstellen für die konkreten Anzeigeelemente angegeben werden, beispielsweise Nachrichtenfenster, Routenplaner und Statusanzeige. Dabei wird jeweils die obige Schnittstelle erweitert, wie es in der objektorientierten Programmierung üblich ist, wenn eine „*is a*“-Relation vorliegt – ein Nachrichtenfenster *ist ein* Anzeigeelement. Zusätzlich werden die Rollen konkretisiert, was z.B. für das Nachrichtenfenster zu der folgenden Arbeitsschnittstelle führt:

```
public interface Nachrichtenfenster extends Anzeigeelement
{
  String ROLE = Anzeigeelement.ROLE + ".nachrichtenfenster";
}
```

Zu einer Arbeitsschnittstelle gehört bei Avalon immer auch eine sie implementierende Klasse. Der Einfachheit halber soll hier angenommen werden, dass es für die drei genannten Schnittstellen die Klassen DebugNachrichtenfenster, DebugRoutenplaner und DebugStatusanzeige gibt, die nur Testzwecken dienen und später durch vollständige Implementierungen ersetzt werden sollen. Beispielsweise könnte die erstgenannte Klasse wie folgt aussehen:

```
import org.apache.avalon.framework.component.*;

class DebugNachrichtenfenster implements Nachrichtenfenster
{
  DebugNachrichtenfenster () {}

  public void draw () {
    System.out.println(this.ROLE + " (DEBUG)");
  }
}
```

Die Standansicht ist ein Composable und muss deshalb sowohl einen Komponentenmanager als auch eine compose-Methode haben. Da diese nur einmal aufgerufen werden darf, wird sie so

implementiert, dass weitere Aufrufe keinen Effekt haben; Der beim ersten Aufruf übergebenene Komponentenmanager wird dann einfach beibehalten.

Die Methode checkItem erhält als Argument eine Rolle und überprüft, ob die Standansicht eine Komponente enthält, die diese Rolle übernimmt. Wenn dies der Fall ist, wird die zugehörige draw-Methode aufgerufen, und sonst eine Warnung ausgegeben. Zum Überprüfen der Vollständigkeit wird weiterhin die Methode checkAll definiert, in der checkItem-Aufrufe für alle Anzeigeelemente durchlaufen werden.

Alle Methoden der Klasse Standansicht können eine ComponentException werfen, da dies für compose und lookup bereits im Avalon-Framework festgelegt ist.

```
import org.apache.avalon.framework.component.*;

class Standansicht implements Composable
{
  ComponentManager manager;

  Standansicht () {}

  public void compose (ComponentManager managerNew)
    throws ComponentException
  {
    if (manager == null) manager = managerNew;
  }

  public void checkItem (String role)
    throws ComponentException
  {
    if (manager.hasComponent(role))
    {
      Anzeigeelement anzeige =
        (Anzeigeelement)manager.lookup(role);
      anzeige.draw();
    } else System.out.println("Warnung: Ansicht ohne " + role);
  }

  public void checkAll ()
    throws ComponentException
  {
    checkItem (Nachrichtenfenster.ROLE);
    checkItem (Statusanzeige.ROLE);
    checkItem (Routenplaner.ROLE);
  }
}
```

An dieser Stelle muss angemerkt werden, dass die hier verwendete Konstruktion etwas anders aussähe, wenn mehrere Ansichten zu beachten wären. Dann nämlich wäre es sinnvoller, eine

abstrakte Klasse Ansicht hinzuzunehmen, die neben der Schnittstelle Composable auch die Schnittstelle Component implementiert. Auf diese Weise könnten die verschiedenen aus dieser abstrakten Klasse abgeleiteten Ansichten wieder als Komponenten erachtet werden, womit deren Handhabung einfacher würde.

Andererseits bricht die oben vorgestellte Methode checkAll mit dem Prinzip der *inversion of control*, da sie die Informationen über benötigte Anzeigeelemente innerhalb der Klasse Standansicht bündelt. Bei einer ernsthaften Implementierung wäre es offenbar eher angebracht, den Aufrufer bzw. Benutzer der Ansicht entscheiden zu lassen, welche Anzeigeelemente er in ihr erwartet. Im Rahmen des hier betrachteten Beispiels wird darauf aber verzichtet, da das Augenmerk allein auf der Verwendung der Schnittstelle Composable liegt.

Abschließend wird noch die Klasse ComposableDemo eingeführt, die das Hauptprogramm liefert. Dort werden zunächst eine Standansicht und ein Komponentenmanager instanziiert. Danach wird der Manager mit zwei der drei oben betrachteten Anzeigeelemente aufgefüllt und dann über die compose-Methode an die Standansicht übergeben. Zuletzt erfolgt ein Aufruf der checkAll-Methode.

```
import org.apache.avalon.framework.component.*;

class ComposableDemo
{
  public static void main(String argv[])
  {
    Standansicht            ansicht =
      new Standansicht();
    DefaultComponentManager manager =
      new DefaultComponentManager();

    manager.put(Nachrichtenfenster.ROLE,
                new DebugNachrichtenfenster());
    manager.put(Statusanzeige.ROLE,
                new DebugStatusanzeige());

    try {
      ansicht.compose(manager);
      ansicht.checkAll();
    }
    catch (ComponentException ce)
    {
      System.out.println("Fehler bei Komposition der Ansicht.");
    }
  }
}
```

Da dem Komponentenmanager keine Komponente übergeben wird, die die Rolle des Routenplaners einnehmen kann, liefert das Programm die folgende Meldung (gekürzt):

```
...hycop.anzeigeelement.nachrichtenfenster (DEBUG)
...hycop.anzeigeelement.statusanzeige (DEBUG)
Warnung: Ansicht ohne ...hycop.anzeigeelement.routenplaner
```

19.3 Fazit

An dieser Stelle soll der Übergang von einer weitestgehend bewertungsfreien Beschreibung des Avalon-Komponentenmodells zu einer konkreten Beurteilung erfolgen. Dazu werden zunächst die objektiv ermittelbaren Vor- und Nachteile herausgearbeitet, während die abschließende Bewertung durchaus auch dem Einfluss subjektiver Eindrücke unterliegt.

19.3.1 Vorteile

Die eindeutige Zuordnung zwischen der Arbeitsschnittstelle, der Rolle und der Implementierung einer Komponente kann die Transparenz erhöhen. Eine besonders angenehme Konsequenz ist die problemlose Austauschbarkeit: Soll die Implementierung einer Komponente durch eine andere ersetzt werden, so geschieht dies unter Beibehaltung sowohl der Arbeitsschnittstelle als auch der zugehörigen Rolle. Wenn die neue Implementierung alle Verträge beachtet, sollte der Anpassungsaufwand gering ausfallen.

Das Vertragssystem rund um den Lebenszyklus von Komponenten reduziert die Anzahl möglicher Aufrufabfolgen, so dass weniger Ausnahmefälle abgefangen werden müssen, während mehr Annahmen über die Arbeitsweise auch unbekannter Komponenten möglich werden. Dies bedeutet also einen weiteren Zuwachs an Transparenz, der die Wartung eines komplexen Systems und die Integration externer, z.B. aus Fremdfertigung oder Standardbibliotheken stammender Komponenten erleichtert.

Generell gilt, dass bei der Entwicklung großer Systeme die Komponentenorientierung eine konsequente und hilfreiche Fortsetzung der Objektorientierung darstellt. Der Entwurfsprozess wird weiter systematisiert, da er entlang der Hierarchie durchgeführt werden kann. Die Frage, welche Komponenten das System benötigt, lässt sich oft relativ schnell anhand der Anforderungen beantworten. Nachdem der Zusammenhang zwischen den Komponenten geklärt ist, können diese dann im Einzelnen, und zwar objektorientiert, entworfen werden. Eine wertvolle Auswirkung einer solchen Vorgehensweise beim Entwurf ist, dass mit den Komponenten relativ schnell eigenständige Bausteine herausgearbeitet werden können, deren interner Entwurf und deren spätere Implementierung dann frühzeitig von einer eigenen Entwicklergruppe übernommen werden können.

19.3.2 Nachteile

Bei der konsequenten Verwendung des Avalon-Frameworks kommt es zu einem Zuwachs an Typinformationen. Dies liegt einerseits an den zahlreichen Schnittstellen für spezielle Komponententypen (wie z.B. Contextualizable oder Configurable), andererseits aber auch an den Arbeitsschnittstellen für die einzelnen Komponenten. Es ist durchaus denkbar, dass eine weitverzweigende Hierarchie unübersichtlich werden kann, so dass die oben erwähnten Transpa-

renzgewinne zumindest für Entwickler, die das Avalon-Framework nicht vollständig kennen, wieder verloren gehen.

Weiterhin kann das Avalon-Framework natürlich nur Abläufe und Strukturen standardisieren, bei denen dies keine Einschränkung der Anwendbarkeit zur Folge hat. So ist beispielsweise klar, dass es konfigurierbare Komponenten gibt und dass sich die zugehörigen Konfigurationsdaten in Configuration-Objekten festhalten lassen, wohingegen eine Festlegung bezüglich der Struktur dieser Daten wenig sinnvoll wäre. Avalon vereinfacht also nur einige wenige Aufgaben, die sich generell vereinfachen lassen. Deshalb kann es bei der Konstruktion großer Systeme eventuell von Vorteil sein, statt des Avalon-Frameworks einfach eine auf die speziellen Anforderungen des jeweiligen Systems zugeschnittene Architektur vorzuziehen, oder die Vorgaben, die Avalon macht, wesentlich zu ergänzen.

19.3.3 Abschließende Beurteilung

Das Avalon-Framework liefert ein gut strukturiertes, allgemein anwendbares und in vielen Punkten komfortables Komponentenmodell. Dies hat nicht nur positive Auswirkungen auf den entstehenden Programmcode, der transparenter ausfällt, sondern auch auf den Entwurfsprozess, der durch die zusätzliche Abstraktionsebene der Komponentenorientierung besser hierarchisierbar wird. Andererseits ist das Modell selbstverständlich an eine allgemeine Auslegung und beschränkte Wirkungsbereiche gebunden. Die Konstruktion eines komplexen Systems wird mit dem Avalon-Framework einfacher, aber deshalb natürlich nicht unbedingt einfach.

Kapitel 20

Das JavaBeans™-Komponentenmodell

Autoren: *Rafael Hosenberg und Yue Zhang*

Dieser Abschnitt gibt einen Einblick in das Java-Komponentenmodell JavaBeans™. Dazu wird zunächst das allgemeine Komponentenmodell erläutert. Ausgehend von den Eigenschaften, die ein Komponentenmodell kennzeichnet, werden diese Eigenschaften die auch das JavaBeans™-Modell kennzeichnen aufgezeigt. Neben dem Komponentenmodell bildet die Arbeit mit JavaBeans™ in der Praxis einen Schwerpunkt. Dazu werden einfache Beispiele vorgestellt.

20.1 Komponentenmodell

Eine Software-Komponente besitzt gewisse Eigenschaften, die in einem Modell beschrieben werden. Ein solches Modell nennt man Komponentenmodell. Im Allgemeinen kennzeichnet sich ein Komponentenmodell durch folgende vier Eigenschaften aus. Jedes Komponentenmodell besitzt eine Architektur. Diese beschreibt das Komponentenmodell und gibt diesem eine individuelle Charakteristik. Eine solche Architektur ist Grundlage einer jeden API (application programming interface), die die Entwicklung und Manipulation von JavaBeans™ unterstützt. Die Architektur eines Komponentenmodells wird im wesentlichen durch Konventionen bestimmt. In Java werden diese als *Design Pattern* bezeichnet. Von besonderer Bedeutung ist die Strukturangabe der Komponenten und die Strukturangabe der Container. Desweiteren stellen die Container Methoden bereit um mehrere Komponenten zu verwalten. Was ein Container im einzelnen darstellt, darauf wird in Kapitel 20.1.1 eingegangen.

20.1.1 Komponenten

Komponenten sind Softwarebausteine, siehe [67]. Sie verhalten sich im Vergleich zu anderen Softwarebestandteilen wie eine *Blackbox*. Von außen ist nicht ersichtlich was im Innern der Blackbox passiert. Sie kommuniziert über eine normierte Schnittstelle. Diese Schnittstelle besteht aus drei voneinander unabhängigen Bestandteilen. Eine Blackboxkomponente kann über *Eigenschaften* (*properties*), *Methoden* (*methods*) und über *Ereignisse* (*events*) mit anderen Softwarebausteinen kommunizieren. Jede Komponenteninstanz befindet sich in einem Zustand.

Dieser Zustand wird durch die Eigenschaftswerte bestimmt. In Abhängigkeit von Ihrem Zustand wird ihr dynamisches Verhalten durch Ereignisse oder Methodenaufrufe in Gang gesetzt. Damit aus einzelnen Komponenten größere Anwendungen entstehen können werden mehrere Komponenten in einer *Applikation* zusammengefasst. Ein sogenannter *Container* übernimmt die Verwaltung der Komponenten und enthält auch die Komponenten selbst.

20.1.2 Komponentenmodellkennzeichen

Ein Komponentenmodell wird maßgeblich durch folgende Kennzeichen bestimmt, siehe [40] Kapitel 1. Dazu gehören *Customization*, *Introspection*, *Persistence*, *Event handling*, *Component packaging* und *Distributed Computing*.

Customization

Unter Customization versteht man die Veränderbarkeit von Komponenten. Man kann deren äußere Form selbst gestalten. Zudem ist es möglich, dass man das Verhalten steuern kann. Man kann die Aktionen der Komponenten individuell und unabhängig voneinander definieren.

Introspection

Das Kennzeichen der *Introspection* sichert einer Komponente, dass diese von außen von sogenannten *builder tools* analysiert werden kann. Dabei werden die Merkmale der Komponente bestimmt. Diese sind die Eigenschaften, Methoden und Ereignisse. Gespeichert werden diese Merkmale in der *BeanInfoklasse*.

Persistence

Durch *Persistence* ist es einer Komponente möglich Veränderungen zu speichern. Da Komponenten zustandsbehaftet sind wird es möglich, dass ein bestimmter Zustand wieder geladen werden kann.

Event handling

Ein weiteres wichtiges Kennzeichen eines Komponentenmodells ist das *Event handling*. Dadurch erhalten Komponenten die Fähigkeit agieren zu können. Dies geschieht durch Verschicken von Nachrichten und Empfangen von Nachrichten.

Component packaging

Mit dem *Component packaging* werden all die Funktionen eines Komponentenmodells zusammengefaßt, die dazu nötig sind alle Komponenten oder Teile zu einer Applikation zusammenzufassen. Dazu gehören auch Grafiken oder Musikdateien. Zusätzlich wird eine Laufzeitun-

terstützung dem Packet hinzugefügt. Diese sichert unter anderem, dass das Programm auf verschiedenen Systemen oder Virtuellen Maschinen lauffähig ist.

Distributed Computing

Das Kennzeichen des *Distributed Computing* eines Komponentenmodells beschreibt die Möglichkeit der Komponenten auf verschiedenen Rechnern arbeiten zu können. Dies macht auch eine gewisse Netzwerkkommunikation nötig, welche auch durch Funktionen unterstützt wird. Die Kommunikation erfordert Sicherheitselemente wie zum Beispiel Benutzerkennungen und die Passwortverwendung. Auch diese Funktionen sind Teil des Distributed Computings.

20.1.3 Vorteile eines Komponentenmodells

Ein besonderer Vorteil den Komponenten eines Komponentenmodells haben ist deren *Wiederverwendbarkeit* (Modularisierung). Dabei spielt die Schnittstelle die Rolle eines Schlüssels über die Komponenten miteinander kommunizieren. Ein weiterer Vorteil ist die Möglichkeit der unabhängigen Entwicklung. Die Komponenten können unabhängig voneinander programmiert werden. Wenn im builder tool die Komponenten zusammengesetzt werden zur Applikation ist den Entwicklern nicht bekannt wie diese im Detail im Inneren funktionieren (Blackbox). Nur die Schnittstellenkenntnisse reichen aus um die Komponenten zu einer Applikation zu komponieren. Diese normierte Schnittstelle gibt den Komponenten die Eigenschaft der leichten *Portierbarkeit*. Diese Portierbarkeit zeichnet sich dadurch aus, dass man Komponenten leicht kombinieren kann oder austauschen kann und das Komponenten oder mehrere Komponenten (Applikation) auch auf verschiedenen Systemen laufen lassen kann.

Die Vorteile des Komponentenmodells JavaBeans™ liegen nicht nur in dem Komponentenmodell selbst. *Java* als Grundlage für ein Komponentenmodell besitzt besondere Eigenschaften. Die *Objektorientiertheit*, die Java™ Kennzeichnet gibt dem Komponentenmodell JavaBeans™ diese wichtige Eigenschaft. Zudem profitiert das Komponentenmodell JavaBeans™ von weiteren positiven Eigenschaften der Sprache Java™ wie zum Beispiel der großen Klassenbibliothek oder des Sicherheitsmodells.

20.1.4 Komponentenmodelle

- Visual Basic Extension (VBX)
- ActiveX AND DCOM
- CORBA AND IDL
- OpenDoc
- Enterprise JavaBeans™
- Avalon

20.2 JavaBeans™

Die JavaBeans™ Spezifikation (siehe [28]) enthält eine Definition der JavaBeans™, die das Konzept kurz und prägnant wiedergibt: „A Java Bean is a reusable software component that can be manipulated visually in a builder tool. "In dieser Definition stecken zwei wichtige Eigenschaften. Zum einen, dass ein Java Bean eine wiederverwendbare Softwarekomponente darstellt und zum Anderen, dass diese Softwarekomponente in einer visuellen Entwicklungsumgebung bearbeitet werden kann.

20.2.1 JavaBeans™ Kurzgeschichte

- 1995 Java von Sun eingeführt
- 1996 Komponentenmodell JavaBeans™
- 1996 Dez. JDK 1.1, BDK kurz danach
- Später ActiveX-Unterstuützung (bridge)
- 1999 Enterprise Beans™ für Unternehmensanwendungen

20.2.2 JavaBeans™ Entwicklungszyklus

Jede Anwendungsentwicklung mit JavaBeans™ beginnt mit der unabhängigen Programmierung der Komponenten. Diese Komponenten werden separat compiliert (*Compilephase*). Im nächsten Schritt (*Designphase*) werden die Komponenten in einer *integrierten Java Entwicklungsumgebung* (*Java-IDE*) in der grafischen Darstellung angepasst. Anschließend werden die sichtbaren oder unsichtbaren Komponenten je nach Sinnzusammenhang verbunden. Darauf schließt sich die *Buildphase* an. In dieser Phase wird eine Datei erstellt, die Layout und Interaktionen beschreibt. Diese Phase ist beendet sobald der vollständige Java byte code erzeugt wurde. Dann kann die Applikation durch den Befehl „Run java applikation" in einer Virtuellen Maschine gestartet werden.

20.2.3 Java Bean oder Java-Klasse?

Es ist nicht zulässig Java Beans mit Java-Klassen zu vergleichen. Diese beiden Modelle stellen jeweils andere Abstraktionsebenen dar. Darum variieren ihre Einsatzgebiete. Da sie aber sehr viel gemeinsam haben ist es interessant ihre Unterschiede herauszustellen. Der wesentliche Unterschied besteht in der Unterstützung von Introspection. Introspection ist ein wesentlicher Teil von JavaBeans™, der in Java-Klassen nicht vorgesehen ist. Tools sind bei JavaBeans™ in der Lage die Schnittstellen eines Beans zu erkennen (Design Pattern). Und Java Beans von sich aus haben die Eigenschaft ihr Verhalten durch Schnittstellen (Properties, Methods, Events) zu veröffentlichen. Dies sind Eigenschaften der Komponentenbasierten Softwareentwicklung. Es gibt jedoch Situationen in denen Klassen besser geeignet sind. Zum Beispiel für SQL-Anwendungen oder für den Zugriff auf Relationale Datenbanken JDBC (Java Database Connectivity - API).

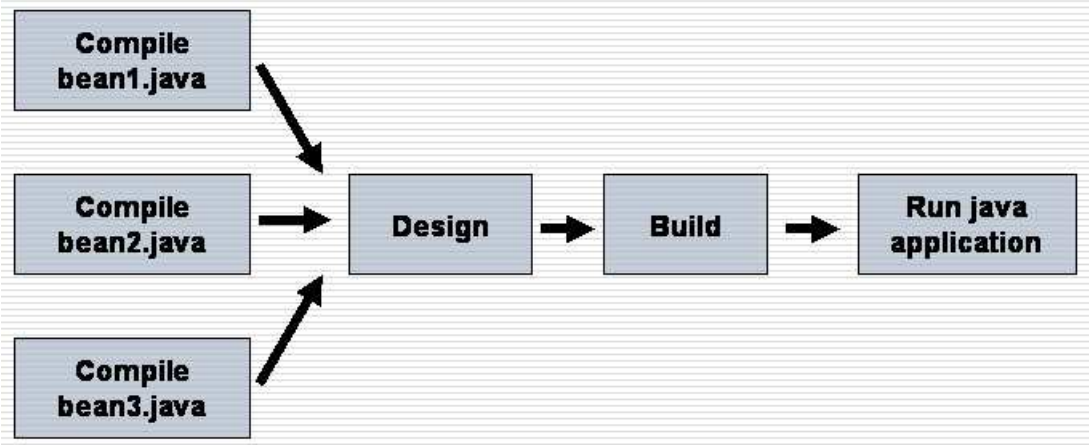

Abbildung 20.1: JavaBeans der Entwicklungszyklus

20.3 JavaBeans™ Kennzeichen

20.3.1 Persistent Storage

Mit Hilfe der *Persistent Storage* kann man Komponenten oder Teile dieser speichern. Insgesamt gibt es drei Möglichkeiten der Speicherung. Eine Möglichkeit ist die Verwendung der readObject- und writeObject-Methoden. Mit diesen beiden Methoden kann man Java-Klassen explizit speichern und laden. Mit Hilfe der *Serialization/Deserialization* ist es Möglich die Speicherung zu automatisieren. Dabei wird ein bytestream erzeugt, der in eine Datei oder über ein Netzwerk umgeleitet werden kann. Nur Objekte, die das Interface java.io.Serializable implementiert haben können so gespeichert werden.

```
public class Button implements java.io.Serializable
{
  private String theName; // property
}
```

Nach Konvention werden in einem builder tool veränderte Beans mit dem .ser Suffix gesichert. Es wird jeweils ein Bean in einer .ser Datei gespeichert wobei ser füer serialisierte Beans steht. Eine weitere Methode Beans zu speichern wird mit *Externalization* bezeichnet. Hierbei wird dem Objekt die vollständige Kontrolle über den Schreib- und Lesevorgang gegeben. Das heißt, dass das Objekt selbst über Art, Struktur und Format der Daten die auf dem Stream abgelegt werden entscheidet. In diesem Fall muß das Interface java.io.Externalizable implementiert werden. Programme, die Objekte speichern, haben keinen Einfluß über die serialisierte oder externalisierte Speicherung. Dies legen die Objekte selbst fest.

20.3.2 Packaging

Um Java Beans verwenden zu können werden diese in einem Packet zusammengefaßt. Ein solches Packet nennt man *JAR-Archiv*. Dieses wird im ZIP-Format abgespeichert. Das Archiv kann eine *MANIFEST* Datei enthalten die nähere Informationen bezüglich des Inhalts des Packets enthält. Ein JAR Archiv das Beans enthält muß eine MANIFEST Datei enthalten. Der Inhalt eines JAR Archivs besteht aus .class Dateien, serialisierten Beans, Hilfe Dateien in HTML, und resourcen (Bilder, Musik, Text).

20.4 Drei wichtige Kennzeichen von JavaBeans™

In der Praxis kennzeichnen drei Merkmale, nämlich **properties**, **methods** und **events**, ein Java Bean. Wenn diese drei Merkmale existieren, dann ist ein Java-Programm ein Bean.

20.4.1 Properties

Definition

Die genaue Definition eines property lautet, siehe [28] Kapitel 7: **„Properties are discrete, named attributes of a Java Bean that can effect its appearance or its behaviour.“** *Größe* ist zum Beispiel ein property des Java Bean *Fenster*. Wenn der Wert des property *Größe* verändert wird, dann wird sich die Größe des *Fensters* auf dem Bildschirm auch verändern.

Zwei Arten von properties

Für JavaBeans™ gibt es zwei Arten von properties, *single value property* und *indexed property*. Ein *single value property* ist ein property, das einen einzigen Wert hat. Und ein *indexed property* ist ein property, das einem Array entspricht.

Beispiel

Hier sind zwei Java-Klassen mit ihren *properties* aufgeführt:

```
class Sensor
{
  private string Name; //single value property.
}

class Monitor
{
  private Color[]  theSignals; //indexed property
}
```

Design Pattern

Für *single value properties* müssen zwei bestimmte Methoden im Bean existieren, damit andere Komponenten auf das *property* zugreifen können. Das *Design Pattern* sieht wie folgt aus:

1. void setProperty(PropertyType newValue);
2. PropertyType getProperty();

Für *indexed properties* müssen vier Methoden in dem Bean existieren, damit andere Komponenten auf das Array als eine Einheit oder einem Wert des Arrays durch einen Index zugreifen können.

1. void setProperty(PropertyType[] newValue);
2. PropertyType[] getProperty();
3. void setProperty(int index, PropertyType newValue);
4. PropertyType getProperty(int index);

Sonstige Merkmale

Es gibt noch einige wichtige Merkmale eines properties. Diese werden hier mit der Definition aufgeführt.

- Abhängige properties, siehe [28] Kapitel 7
  „They allow other components to bind special behaviour to property changes.“
- Eingeschränkte properties, siehe [28] Kapitel 7
  „Sometimes when a property change occurs some other bean may wish to validate the change and reject it if it is inappropriate. We refer to properties that undergo this kind of checking as constrained properties.“

20.4.2 Methods

Definition

Die Definition einer Methode lautet wie folgt, siehe [28] Kapitel 2: **„The Methods a Java Bean exports are just normal Java methods which can be called from other components or a scripting environment.“**

Wird eine Methode einer Java-Klasse als public definiert, ist sie nach der Definition schon eine Methode für ein Java Bean. Im Wesentlichen gibt es keine anderen Beschränkungen für eine Methode eines JavaBeans™ außer dem Zugriffsrecht.

Beispiel

```
public class Sensor
{
  private String theName;  //property

  public void setType(String newType)  // method for property
  {
    theName = newType;
  }

  public String getType()
  {
    return theName;
  }

  public void measure()  //method
  {
    ...
  }
}
```

20.4.3 Events

Definition

Die genaue Definition für Events ist, siehe [28] Kapitel 6: **„Events are a mechanism for propagating state change notification between a source object and one or more target listener objects.“** Hier sind Events Mechanismen für asynchrone Übertragungen von Informationen.

Mechanismus

Der Mechanismus eines Events hat eine asynchrone Struktur. Neben der **Quellkomponente** und dem **Zuhörer** sind auch der **Event** selbst und eventuell auch ein **Adapter** wichtige Objekte um den Mechanismus auszuführen. Am Anfang muß sich jeder Zuhörer in der Quellkomponente registrieren. Dann weißt die Quellkomponente den Zuhörern die Informationen zu. Falls ein Ereignis passiert, sendet die Quellkomponente an alle Zuhörer ein Event mit der entsprechenden Information. Im Fall eines Event Adapters werden alle Events zuerst zu diesem Adapter gesandt. Hier ist das Senden-/ und Empfangensverfahren asynchron.

Event State Objekt

Ein **Event** enthält die Information des Ereignisses. Nach der Definition aus der Spezifikation für JavaBeans™ soll jedes Event State Objekt von der Klasse java.util.EventObject erben. Der

Klassenname soll mit dem Suffix „Event“ enden.
Hier ist das TempoMeasuredEvent als Beispiel aufgeführt:

```
public class TempoMeasuredEvent extends java.util.EventObject
{
  private int theTempo;

  public int getTempo()
  {
    return theTempo;
  }
}
```

EventListener Interface

Jeder Zuhörer muß ein Interface implementieren, das von java.util.EventListener erbt. Das Interface integriert alle Methoden, die auf dem gleichen Event reagieren können. In diesem Interface werden solche Methoden als abstrakte Methoden definiert. Jeder Zuhörer, der das mit dem Interface entsprechende Event empfangen möchte, muß solche Methoden realisieren. In dem folgenden Beispiel wird ein Event, ein Interface und ein Zuhörer mit dem entsprechenden Interface verwendet.

```
class TempoMeasuredEvent extends java.util.EventObject {}

interface TempoMeasuredListener extends java.util.EventListener
{
  void tempoIsMeasured();
}

class Monitor implements TempoMeasuredListener
{
  void tempoIsMeasured()  // method for Interface
  {
    ...
  }

  private Color[] theSignals; // property

  void setSignals( Color[] newSignals )
  {
    theSignals=newSignals;  // method for property
  }

  Color[] getSignals()
  {
```

```
    return theSignals;
  }

  public void setSignal( int index, Color newSignal)
  {
    theSignals[index] = newSignal;  // method for property
  }

  public Color getSignal( int index)
  {
    return theSignals[index];
  }
}
```

Quellkomponente

Die Quellkomponente ist ein Java Bean. Die Komponente ist zuständig für die Übertragung des Ereignisses. In erster Linie muß sie zwei Methoden haben, um die Zuhörer registrieren und abmelden zu können. Die Komponente kann auch die Relation zwischen Quellkomponente und dem Zuhörer dynamisch manipulieren. Für mehrfache Registrierungen soll die Quellkomponente das entsprechende Verhalten selbst definieren. Falls ein Ereignis passiert, wird die Quellkomponente allen registrierten Zuhörern ein Event schicken. Das entsprechende Beispiel sieht wie folgt aus:

```
public class Sensor  // method for source
{
  public void addTempoMeasuredListener(TempMeasuredListener l)
  {
    ...
  }

  public void removeTempMeasuredListener(TempMeasuredListener l)
  {
    ...
  }

  private String theName;  // property

  public void setType(String newType)  // method for property
  {
    theName = newType;
  }

  public String getType()
  {
    return theName;
```

```
  }

  public void measure()  // method
  {
    ...
  }
}
```

Event Adapter

Manchmal können die Zuhörer das Interface direkt nicht implementieren. Das heißt das Extrafunktionen zwischen Quellkomponente und Zuhörer realisiert werden. Dann wird ein **Adapter** gebraucht. Ein solcher Adapter implementiert das Interface und verbindet die Quellen und die Zuhörer. Wenn ein Ereignis passiert, empfängt der Adapter das entsprechende Event. Danach entscheidet der Adapter, welche Methoden der Zuhörer aufgerufen werden.

Zwei Arten eines Adapters sind möglich. Eine Form ist der *einfache Adapter*. Das heißt, jedes Interface wird mit einem entsprechenden Adapter versehen. Wenn es drei Typen von Events gibt, dann werden drei unterschiedliche Adapter definiert. Jedes Event wird zuerst von einem Adapter empfangen und dann zum Zuhörer geschickt. Die zweite Form eines Adapters ist der *generic Adapter*. Man definiert für eine bestimmte Anzahl an Zuhörern einen Adapter. Alle Events werden zu erst zu diesem Adapter geschickt. Der Adapter wird nach dem Empfang der Ereignisse entscheiden, welche Zuhörer benachrichtigt werden und welche Methoden aufgerufen werden sollen.

20.4.4 Zusammenfassung

Die drei wichtigen Merkmale von JavaBeans™, nämlich **properties**, **methods** und **events** sind die Basis der JavaBeans™. In der Praxis haben wir mit einem Java-Programm schon einen Bean, wenn die drei Eigenschaften existieren. Grundsätzlich sind für properties zwei Methoden zu definieren um ein property verändern zu können. Die Methode eines Beans unterscheidet sich nicht von einer public Java-Methode. Schließlich ist ein Event ein Mechanismus, der es erlaubt, das Zuhörer auf Ereignisse asynchron reagieren können.

Neben diesen drei wichtigen Eigenschaften besitzen JavaBeans™ noch weitere Eigenschaften. Da sich diese Arbeit nur auf das Komponentenmodell JavaBeans™ bezieht ist die Vorstellung von Eigenschaften nur auf die bezogen, die ein Komponentenmodell im wesentlichen kennzeichnen. Im folgenden werden Beispiele vorgestellt, an denen deutlich wird, wie man das Beans Development Kit 1.0 verwenden kann um damit einfache Applikationen zu gestalten, siehe [25]. Die einfache Komponierbarkeit wird dabei durch das Komponentenmodell JavaBeans™ sichergestellt.

20.5 JavaBeans™ Beispiele

20.5.1 JavaBeans™ Beispiel Juggler

Dieses Beispiel zeigt das JugglerBean. Anhand des Beispiels wird gezeigt, wie man Eigenschaften eines Beans beeinflussen kann und wie sich dies direkt auf die grafische Repräsentation auswirkt. Zudem wird gezeigt, wie Aktionen in einem Bean mit Hilfe der Entwicklungsumgebung definiert werden können.

20.5.2 JavaBeans™ Beispiel RectBean

Anhand des Beispiels wird gezeigt, wie Eigenschaften durch wenig Aufwand gesetzt werden können und wie das resultierende Bean in der BeanBox dargestellt wird.

```
import.java.awt.*;
import.java.io.Serializable;

public class RectBean extends Canvas implements Serializable
{
  public RectBean()
  {
    resize(60,40);
  }

  public void paint(Graphics g)
  {
    g.setColor(Color,blue);
    g.fillRect(20,5,20,30);
  }
}
```

20.6 Fazit

Das Komponentenmodell JavaBeans™ unterstützt Entwickler durch seine besonderen Eigenschaften. Es stellt kein vollständig anwendbares Entwicklungssystem dar. Diese Aufgabe übernehmen sogenannte Java-IDEs (builder tools). Es bietet allerdings durch seine Struktur und durch seine Merkmale eine besondere Unterstützung für den Entwicklungsprozess in Java. Das Modell vereinfacht Entwicklern oder Designern die Arbeit an einem gemeinsamen Projekt. Der nötige Kommunikationsbedarf, der Entwickler untereinander, wird durch das Modell auf ein kleines und überschaubares Maß reduziert. Der nicht unerhebliche Effekt liegt darin, dass Entwickler unabhängig voneinander an Komponenten arbeiten können (Modularisierung). Ohne das Gesamtsystem zu kennen sind sie, durch Anwendung des Paradigmas JavaBeans™, in der Lage eigene Komponenten, nur durch Schnittstellenkenntnisse, zu entwickeln (Blackbox). Da sich durch diese Technik die Verbindung von Komponenten einfach über Schnittstellen

realisieren läßt lassen sich die einzelnen Bestandteile (Komponenten) am Ende des Entwicklungsprozesses leicht zu einer Applikation zusammenstellen. Wenn man die Schnittstellen zusätzlich normiert kann man mehrere Komponenten entwickeln, die ein und dieselbe Aufgabe übernehmen. Diese Komponenten sind so leicht auszutauschen. Die Möglichkeit der Wiederverwendung dieser Komponenten wird dadurch verbessert. Diese können dann auch, durch Java unterstützt, auf anderen Systemen eingesetzt werden die dieselben Schnittstellen bieten.

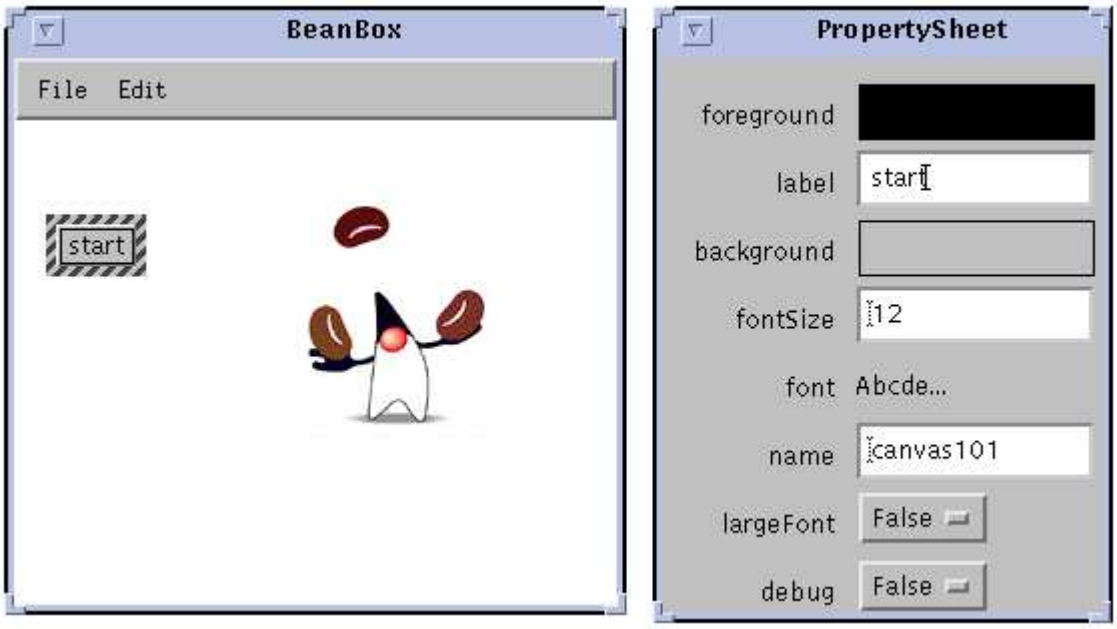

Abbildung 20.2: Beispiel für Beans Juggler

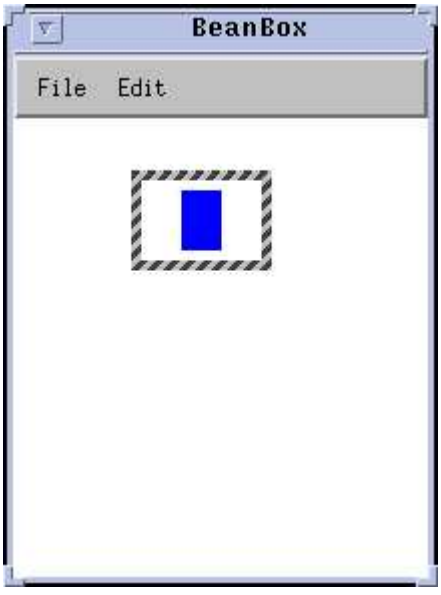

Abbildung 20.3: Beispiel für Beans RectBean

Kapitel 21

Infobus und Java Message Service

Autoren: *Ulf Schellbach*
Oliver Szymanski

Thema dieses Abschnitts ist die Behandlung des Problems des Datenaustausches zwischen Komponenten. Dazu werden zwei spezielle Lösungen vorgestellt. Zum Einen Infobus als Lösung innerhalb einer Java Virtual Machine. Zum Anderen Java Message Service als Lösung in verteilten Systemen.

21.1 Infobus

Infobus wurde von Lotus und Sun entwickelt. Infobus beruht auf der Idee, einen Hardwarebus durch Software zu emulieren. Komponenten können sich an den Bus anmelden und Daten darüber versenden und empfangen. Dabei unterstützt der virtuelle Bus die Kommunikation innerhalb nur einer Java Virtual Machine, wobei durch Erweiterungen von Infobus auch Kommunikation in verteilten Systemen ermöglicht wird.

21.1.1 Infobus Schema

Die Kommunikation über einen virtuellen Infobus findet innerhalb einer Java Virtual Machine statt. Dabei ist der Infobus ein Objekt und stellt Methoden zum An- und Abmelden von Komponenten, Veröffentlichen und Abfragen von Daten zur Verfügung. Komponenten, welche sich am Infobus anmelden, können in drei verschiedenen Rollen auftreten, nämlich als Data Producer, Data Consumer und Data Controller.

Data Producer stellen Daten über den Bus zur Verfügung. Data Consumer empfangen Daten über den Bus. Data Controller regulieren und überwachen den Datenfluss auf dem Bus. Alle Daten, die über den Bus versendet werden, werden erst an die Controller geleitet, welche sie filtern oder Statistiken über den Kommunikationsverlauf führen können.

21.1.2 Infobus Methoden und ihre Verwendung

Der Infobus stellt folgende Methoden zur Verfügung:

infobus.join(...) dient dazu sich am Infobus anzumelden. Jede Komponenten, die auf den Infobus zugreifen möchte, muss sich mit dieser Methode dem Infobus bekanntmachen.

infobus.leave(...) dient dazu sich vom Infobus abzumelden.

infobus.addDataProducer(...) dient dazu einen Data Producer als Listener dem Infobus hinzuzufügen. Dies ermöglicht Data Consumern gezielt Datenanfragen zu stellen, die den angemeldeten Data Producer Listenern zugestellt werden.

infobus.addDataConsumer(...) dient dazu einen Data Consumer als Listener dem Infobus hinzuzufügen. Jeder angemeldete Data Consumer Listener wird benachrichtigt, wenn neue Daten auf dem Infobus veröffentlicht werden.

infobus.fireItemAvailable(...) wird zur Veröffentlichung neuer Daten im Infobus verwendet.

infobus.findDataItem(...) wird zur gezielten Datenanfrage verwendet. Diese Anfrage wird an alle angemeldeten Data Producer Listener weitergeleitet. Jeder Data Producer entscheidet dann selbst ob er die Anfrage bearbeitet und evtl. neue Daten veröffentlicht.

consumer.dataItemAvailable(...) wird vom Infobus bei allen angemeldeten Data Consumer Listenern aufgerufen, sobald neue Daten auf dem Bus zur Verfügung stehen.

producer.dataItemRequested wird vom Infobus bei allen angemeldeten Data Producer Listenern aufgerufen, sobald ein Consumer gezielt Daten anfordert.

Die Kommunikation über den Infobus verläuft nach folgendem Schema. Data Producer und Data Consumer melden sich beim Infobus an und fügen ihm danach entsprechende Listener hinzu. Ein Data Producer kann nun Daten veröffentlichen, die den Data Consumer Listenern zugestellt werden. Ein Data Consumer kann Datenanfragen stellen, die vom Infobus an die angemeldeten Data Producer Listener weitergeleitet werden. Diese entscheiden selbständig, ob sie die Anfragen behandeln.

21.1.3 Infobuserweiterung für verteilte Systeme

Zur Erweiterung der Infobustechnologie auf mehrere Java Virtual Machines, die untereinander durch ein Netzwerk verbunden sind, bedient man sich sogenannter Repeater. Ein Repeater ist an einem Infobus sowohl als Data Consumer, als auch als Data Producer angemeldet. Alle Daten die er in seiner Rolle als Consumer empfängt, versendet er über das Netzwerk an alle ihm bekannten Repeater. Daten die ein Repeater von anderen Repeater empfängt, veröffentlicht er auf dem ihm zugehörigen Infobus.

21.2 Java Message Service (JMS)

Der Java Message Service ist ein Nachrichtendienst, der aus einer Message Oriented Middleware und einer API besteht. Die Message Oriented Middleware ist ein Server, welcher Nachrichten verwaltet und die API beinhaltet Schnittstellen zum Senden und Empfangen von Nachrichten.

Sowohl Server als auch die API gibt es in verschiedenen Implementierungen unterschiedlicher Hersteller (IBM, Sun, Bea, ExoLab).

21.2.1 JMS Nachrichtenmodelle

In JMS werden zwei Nachrichtenmodelle unterstützt. Das Publish-And-Subscribe Modell erlaubt mehreren Produzenten, Nachrichten zu einem bestimmten Thema (Topic) zu veröffentlichen, welche allen zu dem Thema angemeldeten Konsumenten zugestellt werden. Hierbei werden die Produzenten in Anlehnung an dieses Verhalten Publisher und die Konsumenten Subscriber genannt.

Das Point-To-Point Modell ermöglicht mehreren Produzenten, Nachrichten in eine Warteschlange (Queue) zu stellen. Eine unbestimmte Anzahl von Konsumenten kann sich an der Warteschlange anmelden. Allerdings wird jede Nachricht von höchstens einem Konsumenten empfangen. In diesem Modell werden die Produzenten Sender und die Konsumenten Receiver genannt.

21.2.2 JMS API

Die JMS API beinhaltet die Komponenten ConnectionFactory, Connection, Session, Destination, Message, MessageProducer und MessageConsumer.

ConnectionFactory Die ConnectionFactory wird von Produzenten und Konsumenten zur Erzeugung einer speziellen Verbindung zu einem JMS Server genutzt. Diese Verbindung wird in einem Connection Objekt gekapselt.

Connection Das Connection Objekt repräsentiert einen einzelnen Kommunikationskanal zu einem JMS Server und ist eine Netzwerkverbindung. Die Art der Netzwerkverbindung ist abhängig vom Hersteller des JMS Servers und davon, welches Netzwerk man benutzt.

Session Alle Kommunikation über eine Verbindung geschieht im Rahmen einer Session die mit Hilfe des Connection Objektes erzeugt wird. Eine Session kann transaktionsbasiert (transacted) oder nicht transaktionsbasiert erzeugt werden.

Eine Session im Transaktionskontext puffert zu sendende Nachrichten und verschickt sie erst, bis ein expliziter Aufruf der Methode commit erfolgt. Der Aufruf von commit bewirkt, dass alle zwischengespeicherten Nachrichten als atomare Einheit versendet werden. Erolgt ein Aufruf der Methode rollback werden alle zwischengespeicherten Nachrichten verworfen und die Session kehrt in den Zustand nach dem letzten Aufruf von commit zurück.

Eine nicht transaktionsbasierte Session kann in einem von drei Modi erzeugt werden. AUTO_ACKNOWLEDGE bewirkt dass die Session den Empfang von Nachrichten automatisch bestätigt. CLIENT_ACKNOWLEDGE überträgt die Verantwortung für die Bestätigung des Nachrichtenempfangs den einzelnen Konsumenten. Dagegen führt der Modus DUPS_OK_ACKNOWLEDGE dazu, dass die Session die Auslieferung der Nachricht automatisch bestätigt. Dabei wird der Empfang von Duplikaten beim Konsumenten in Kauf genommen.

Destination Die Destination repäsentiert eine Zieladresse, also z.B. ein Topic oder eine Queue.

Message Ein Message Objekt repräsentiert eine Nachricht in JMS. Sie besteht aus den Teilen Header, Properpties und Body. Der Header beinhaltet Kontrollinformationen zur Weiterleitung und zur Identifikation der Nachricht. Die Properties sind optionale Applikationsspezifische Eigenschaften der Nachricht, welche die unterschiedlichen Nachrichtentypen unterstützen.

JMS unterstützt durch Erweiterung des Message Interfaces folgende Nachrichtentypen: BytesMessage, StreamMessage, ObjectMessage, MapMessage und TextMessage.

Mit Hilfe der Methode acknowledge eines Message Objektes können Konsumenten den Empfang der speziellen Nachricht bestätigen.

MessageProducer Zum Versenden einer Nachricht an eine Zieladresse muss zunächst mit Hilfe des Session Objektes ein MessageProducer erzeugt werden. Das MessageProducer Objekt wird benutzt um Nachrichten zu erzeugen.

MessageConsumer Ein MessageConsumer Objekt wird mit Hilfe eines Session Objektes erzeugt, wenn Bedarf für den Empfang von Nachrichten besteht. MessageConsumer sind mit einer bestimmten Zieladresse verbunden, von der sie Nachrichten empfangen. Es gibt zwei Möglichkeiten des Nachrichtenempfanges. Zum Einen kann ein Konsument einen Listener anmelden, was dazu führt, dass er beim Vorliegen neuer Nachrichten darüber informiert wird. Diese Art des Nachrichtenempfanges ist asynchron. Zum Anderen kann ein Konsument mittels verschiedener receive Methoden beim Server anfragen, ob neue Nachrichten vorliegen und diese gegebenenfalls empfangen. Diese Art des Nachrichtenempfanges ist synchron.

Zu jeder dieser Komponenten außer der ConnectionFactory gibt es Spezialisierungen für Topics und für Queues (Publisher/Subscriber und Sender/Receiver) um beide JMS Nachrichtenmodelle zu unterstützen.

21.2.3 Schritte bei der Nutzung von JMS

Das Folgende zeigt die Schritte, welche bei der Benutzung von JMS erforderlich sind.

1. Factory über JNDI Context finden
2. Connection anlegen
3. Session anlegen

4. Destination anlegen bzw. finden
5. Producer und Consumer erzeugen
6. Optional einen Message Listener beim Konsumenten setzen
7. Nachricht erzeugen und versenden
8. Nachricht über den Message Listener empfangen (asynchron) oder durch explizenten Aufruf einer receive Methode (synchron)

21.3 Fazit

Infobus ist beschränkt auf eine Java Virtual Machine, aber erweiterbar auf verteilte Systeme. Sowohl asynchrone als auch synchrone Kommunikation über den Bus werden unterstützt. Der Bus läuft in derselben Java Virtual Machine wie die Applikation. Innerhalb von lokalen Anwendungen bietet Infobus schnellere Kommunikationsmöglichkeiten als JMS.

JMS unterstützt asynchrone und synchrone Kommunikation in verteilte Systemen. Ein separater Server ist erforderlich, was Overhead in lokalen Anwendungen zur Folge hat. JMS hat im Vergleich zu den Repeatern bei Infobus in verteilten Systemen einen Geschwindigkeitsvorteil und bietet mehr Funktionalität.

Daher ist es durchaus sinnvoll beide Systeme parallel zu nutzen, abhängig davon, ob sich die Kommunikationspartner innerhalb einer Java Virtual Machine befinden, oder sie verteilt sind.

Kapitel 22

eXtreme Programming

Autoren: *Markus Niehammer*
Bastian Krol

Wenn man die Softwareentwicklung mit anderen Ingenieursdisziplinen vergleicht, stellt man fest, dass viele Softwareprojekte den ursprünglich geplanten Zeit- und Kostenrahmen erheblich überschreiten. Ein nicht unwesentlicher Teil der Projekte wird gar nicht erst zum Abschluss gebracht, sondern scheitert, nachdem bereits erhebliche Mittel investiert wurden.

Es stellt sich also die Frage, ob man Software nicht wesentlich schneller entwickeln kann, als dies heute üblich ist. Wird eine Software unter allen Umständen ihren Anforderungen gerecht und weist den erforderlichen Qualitätsstandard auf? Wie ein Softwareprojekt abläuft und ob es erfolgreich ist, hängt auch vom angewendeten Entwicklungsprozess ab. Der Unified Process [5] und das Spiralmodell sind Beispiele für solche Entwicklungsprozesse.

Dem gegenüber steht der von Kent Beck vorgeschlagene Entwicklungsprozess eXtreme Programming (XP) (siehe [10]). Beck stellt hier einen leichtgewichtigen Prozess vor, der sich in vielen Aspekten deutlich von anderen Vorgehensmodellen unterscheidet.

22.1 Überblick

Einer der Grundgedanken von XP ist der Folgende: Änderungen ergeben sich während des Projektverlaufs immer, unabhängig davon wie sorgfältig man vorausplant. Insbesondere ist es wahrscheinlich, dass man Arbeit umsonst macht, wenn man viel Zeit in die Planung investiert. Diese Planungsarbeit kann dann von einer grundlegenden Veränderung (z.B. geänderten Anfordungen des Kunden) zunichte gemacht werden.

Aus dieser Überlegung resultiert folgendes Vorgehen: Anstatt viel Zeit in die Planung zu investieren, fängt das Team sofort mit der eigentlichen Arbeit, dem Programmieren, an. Damit diese am Rapid Prototyping angelehnte Vorgehensweise nicht völlig unkoodiniert verläuft, wird ein Metapher für das zu entwickelnde System benutzt (siehe Abschnitte 22.5.4). Im weiteren Verlauf des Projektes wird das anfänglich noch vage Design dauernd angepasst und der Code wird immer wieder durch Refactoring vereinfacht (siehe Abschnitt 22.5.8), ansonsten

würde sich schnell eine völlig undurchschaubare Codebasis entwickeln. Damit beim Refactoring nichts zerstört wird und die Hemmschwelle für das Refactoring gesenkt wird, gibt es für jede nichttriviale Methode einen Test. Diese Tests werden im Laufe des Projekts ständig ausgeführt (siehe Abschnitt 22.5.6). Da die sonst übliche, aufführliche Analysephase entfällt und auch keine ausführliche Dokumentation erstellt wird, ist ein einfaches Design unabdingbar, um stets den Überblick über den Entwurf zu behalten (siehe Abschnitt 22.5.5).

22.2 Voraussetzungen

Damit der im Abschnitt 22.1 skizzierte Prozess funktionieren kann, müssen Änderungen im Entwurf schnell und einfach in das Projekt eingearbeitet werden können. Liegt wie im Spiralmodell vor der Implementierung bereits ein vollständiges Design vor, so können Änderungen ein komplettes Überarbeiten des Entwurfs erfordern und damit das Projekt in ein sehr frühes Stadium zurückwerfen. Einer der Kerngedanken von XP ist es daher, dass Änderungen zum einen nicht vermieden werden können und sollen. Gleichzeitig müssen diese nicht notwendigerweise teuer sein (sowohl finanziell als bzgl. Zeit und Aufwand).

Der Verlauf der Kosten für Änderungen in herkömmlichen Entwicklungsprozessen ist in Abbildung 22.1 dargestellt. Zum einen kann man nun fragen, ob es möglich ist, Projekte so abzuwickeln, dass die Kurve der Kosten für eine Entwurfsänderung wesentlich flacher verläuft. Ein Ansatz, dies zu erreichen, ist XP. Andersherum stellt sich aber auch die Frage, ob mit XP wirklich eine solche Aufwandskurve erzeilt werden kann.

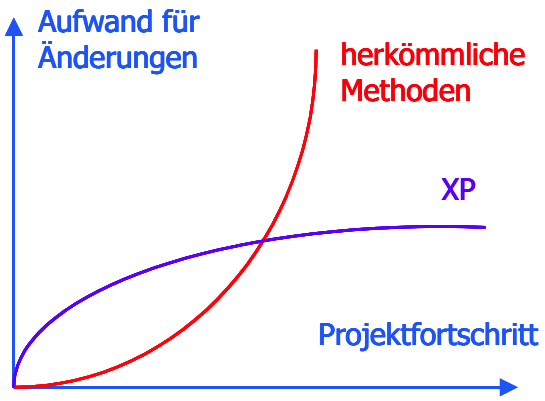

Abbildung 22.1: Aufwandskurven herkömmlicher Methoden im Vergleich zu XP

Weitere Vorausetzungen für ein erfolgreiches XP-Projekt sind

- Die Mitarbeit des Kunden (siehe Abschnitte 22.5.1 und 22.5.3)
- Eine überschaubare Teamgröße (von etwa 10 bis 20 Entwicklern).
- Ein entsprechen ausgestatteter Raum, in dem alle Entwickler gleichzeitig arbeiten können.

22.2.1 Die vier Variablen

XP geht davon aus, dass ein Projekt im wesentlichen von vier Variablen bestimmt wird. Diese sind Qualität, Kosten, Zeit und Umfang. Der Kunde bzw. der Manager des Projekts kann höchstens drei davon vorgeben, die vierte wird dadurch implizit mit festgelegt [45]. Wenn zum Beispiel die gewünschte Qualität sowie der Kosten- und der Zeitrahmen feststehen, ergibt sich der erreichbare Funktionsumfang der zu entwickelnden Anwendung automatisch.

22.3 Die vier zentralen Werte bei XP

XP basiert auf einer engen Zusammenarbeit aller Projektbeteiligten. Jeder im Team trägt die Verantwortung für das gesamte Projekt, was von allen ein hohes Maß Disziplin verlangt.

Einfachheit Die Qualität einer Software lässt sich nicht etwa daran messen, wie komplex und umfangreich sie ist. Ein schlankes, einfaches und übersichtliches Produkt, welches genau den Anforderungen gerecht wird, dient den Erfordernisen des kunden sicherlich eher als ein Produkt mit vielen versteckten Funktionen, die sowieso keiner nutzt. Einfachheit wird beim XP aber nicht nur für das Produkt, sondern auch beim Entwicklungsprozess selbst angestrebt. Das garantieren die im folgenden vorgestellten Aktivitäten, Praktiken und Techniken (Abschnitt 22.5).

Kommunikation Kommunikation ist bei jedem Entwicklungsporzess unerlässlich. Zum einen gilt dies natürlich für die Kommunikaton zwischen den Entwicklern. Beim XP sollte aber auch der Kontakt zum Management und vor allem zum Kunden gesucht und gepflegt werden. Dies sichert, dass alle Beteiligten stets vom aktuellen Stand des Projektes wissen.

Feedback Wird der Kunde aktiv in den Entwicklungsprozess eingebunden, so steigert das ganz offensichtlich die Qualität des Produktes. Es ist dabei wichtig, dass der Kunde stets und unmittelbar Kritiken, Verbesserungen und Änderungswünsche einbringt. Diese können dann von den Entwicklern unmittelbar berücksichtigt werden.

Mut Viele Dinge beim XP erfordern, dass die Beteiligten eine Menge Mut aufbringen. Einfachheit erfordert Mut, weil Dinge eventuell als zu einfach angesehen werden könnten. Jeder im Team muss sich mit Kritiken auseinander setzen und auch den Mut aufbringen, andere zu kritisieren.

22.4 Die zentralen XP-Aktivitäten

Im Entwicklungsprozess treten für alle Beteiligten eine Vielzahl von Aktivitäten auf. Es ist entscheidend für den erfolgreichen Verlauf des Projektes, dass man sich dabei auf das Wesentliche konzentriert.

Die vier zentralen Aktivitäten beim eXtreme Programming sind:

Programmieren Die Kernaufgabe des Projektes ist es, eine Anwendung fertigzustellen. Daher rückt XP das Programmieren wieder mehr in den Vordergrund, als dies vielleicht bei anderen Entwicklungsmethoden der Fall ist.

Testen Wenn das ganze System ständig automatisiert getestet wird, kann man sich (einigermaßen) sicher sein, dass trotz Änderungen nach wie vor noch alles funktioniert.

Zuhören Diese Aktivität unterstreicht die Wichtigkeit der Kommunikation, sowohl innerhalb des Entwicklerteams als auch zwischen Kunden, Managern und Entwicklern (siehe auch Abschnitt 22.3).

Designen Das Design der Anwendung wird am Anfang des Projektes nur vage herausgearbeitet (siehe Abschnitt 22.5.4) und dann ständig verfeinert (siehe Abschnitt 22.5.8).

Diese Aktivitäten laufen nicht etwa phasenweise sequentiell ab, wie zum Beispiel der Entwurf, das Programmieren und Testen bei anderen Entwicklungsmethoden. Vielmehr werden alle vier Aktivitäten grundsätzlich gleichzeitig ausgeführt. Außerdem sind die Zyklen beim XP gegenüber konventionellen Entwickungsprozessen wesentlich kürzer, d.h. es gibt mehr Iterationen, in die das Projekt gegliedert ist (siehe Abbildung 22.2). Dies ist ein Ansatz zum erreichen der flacheren Aufwandskurve aus Abbildung 22.1, denn man befindet sich praktisch ständig im Entwurfsprozess. Außerdem wird das Projekt mit jeder Iteration auch immer wieder neu überdacht und bewertet.

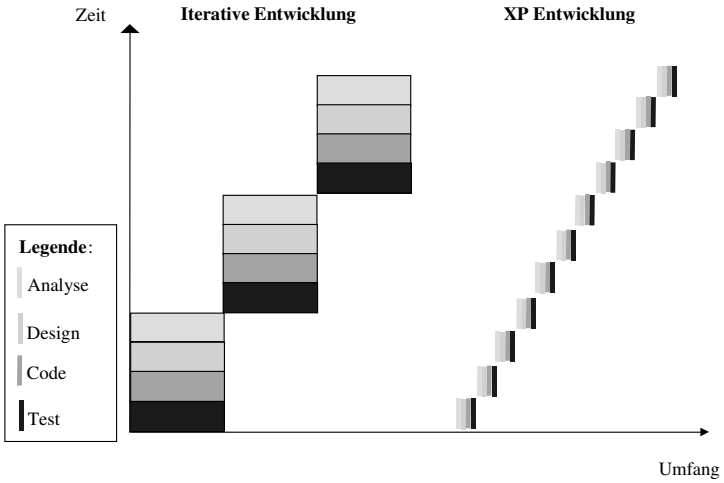

Abbildung 22.2: Entwicklungsphasen herkömmlicher Prozesse im Vergleich zu XP [26]

22.5 Die zwölf XP-Techniken

Unter Beachtung der bisher vorgestellten Werte und Anwendung der Aktivitäten soll der Entwicklungsprozess nach einem bestimmten Muster vorgehen.

Beck gibt zwölf Techniken an, die ein Entwicklerteam in einem XP-Projekt anwenden sollte. Manche dieser Techniken sind auch für sich allein genommen eine „good practice“, andere

wiederum machen isoliert wenig Sinn. Der Punkt ist, dass sich diese Techniken gegenseitig unterstützen und so in ihrer Wirkung verstärken sollen.

In diesem Zusammenhang erklärt sich auch der etwas reißerisch anmutende Name eXtreme Programming. Die Idee ist, alle guten Praktiken, die in Softwareprojekten angewendet werden können, zu sammeln. Jede einzelne Praktik wird dann konsequent genutzt und ihre Anwendung ins Extreme gesteigert.

Die Techniken stehen beim XP in einem engen Zusammenhang. Zum Beispiel wird die Software durch ständiges Refactoring (22.5.8) stets einfach gehalten (22.5.5), das Testen (22.5.6) sichert während der Integration (22.5.7) die Qualität und Korrektheit. Alle Zussammenhänge zu überschauen ist dabei fast unmöglich, was die Komplexität der Abbildung 22.3 deutlich zeigt.

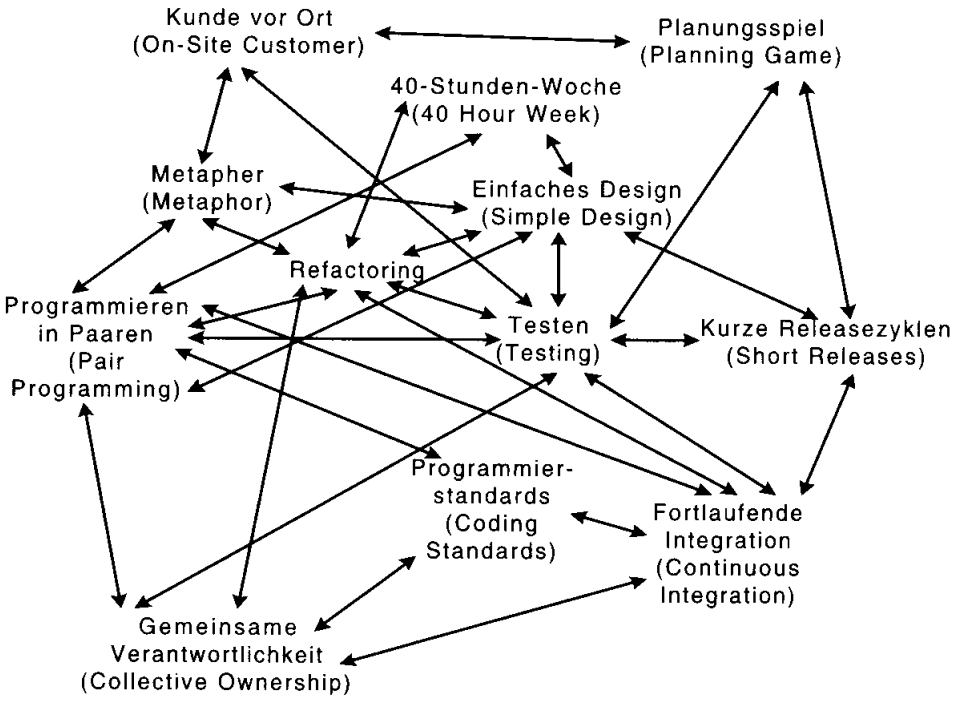

Abbildung 22.3: Zusammenhänge zwischen den einzelnen XP-Techniken [6]

22.5.1 Planungsspiel

Ein XP-Projekt wird in Releases aufgeteilt. Jedes Release hat einen Umfang von ein bis drei Monaten, das erste Release hat evtl. einen längeren Zeitrahmen von vier bis sechs Monaten.

Vor einem neuen Release wird der Umfang und der Zeitrahmen für selbiges im Planungsspiel zwischen den Kunden und den Entwicklern ausgehandelt. Der Kunde schreibt hierfür User

Stories auf, die jeweils eine zu implementierende Funktion der Software aus seiner Sicht beschreiben. Die Entwickler schätzen den Aufwand für die einzelnen Stories ab. Daraufhin teilt der Kunde die Stories ind verschiedene Prioritäten ein (z.B. *must have*, *costly to loose* und *nice to have*). Wenn der Zeitrahmen vorgegeben ist, ergibt sich daraus der Umfang des nächsten Releases. Wird der Umfang des nächsten Releases vorgegeben ergibt sich daraus der Termin für seine Fertigstellung (siehe auch Abschnitt 22.2.1).

Die Entscheidung, welche Features wichtig sind, wird vom Kunden getroffen und nicht vom Entwickler, der mit dieser Entscheidung überfordert sein könnte.

22.5.2 Kurze Releasezyklen

Nach jeweils ein bis drei Monaten (also einem Release) wird das System an die Kunden ausgeliefert. Diese können den Entwicklern dann sofort Feedback geben. Dadurch wird verhindert, dass das Entwicklerteam über Monate oder Jahre hinweg das falsche System entwickelt. Dies bedeutet, dass ein Release eine für den Kunden sinnvolle Einheit darstellen muss, d.h. jedes Release bietet dem Kunden mehr Funktionalität als das vorherige.

Mit dem in Abschnitt 22.5.1 erwähnten Planungsspiel wird der Umfang für jedes Release wieder neu ausgehandelt.

22.5.3 Kunde vor Ort

Die Firma des Kunden stellt dem Entwicklerteam idealerweise einen Mitarbeiter als Vollzeitkraft zur Verfügung. Dieser ist als Ansprechpartner für die Entwickler ständig erreichbar, weiterhin spezifiziert er funktionale Tests (siehe Abschnitt 22.5.6). Diese Forderung erscheint sicherlich vielen Kunden ungewöhnlich und ist evtl. schwierig durchzusetzen.

22.5.4 Metapher

Da das Design der Anwendung nicht vorab in einer breit angelegten Analysephase erarbeitet wird, hilft eine vage Idee der Systemarchitektur den Entwicklern, ihre Implementierungsversuche zu koordinieren. Dieser Metapher, wie auch das Design der Anwendung, entwickelt sich im Verlauf des Projektes ständig weiter.

22.5.5 Einfaches Design

Damit das Design trotz häufiger Änderungen immer überschaubar bleibt, muss man verstärkt darauf achten, es nicht unnötig zu verkomplizieren (siehe auch Abschnitt 22.6). Das Design ist genau dann richtig, wenn

- alle Tests laufen,
- keine Logik dupliziert wird,
- der Code selbsterklärend ist,

- möglichst wenig Klassen und Methoden vorhanden sind.

22.5.6 Testen

Es gibt zwei Arten von Tests, nämlich Unit-Tests und funktionale Tests (auch Akzeptanztests genannt). Bevor man eine Task implementiert, schreibt man einen Unit-Test. Dieser testet jede Methode der geplanten Klasse und dient damit auch gleichzeitig als ad hoc Schnittstellenspezifikation. Danach implementiert man solange, bis die Tests laufen.

Funktionale Tests werden vom Kunden (evtl. in Zusammenarbeit mit einem Entwickler) erstellt und testen jeweils eine Funktion der Anwendung aus Sicht des Kunden.

22.5.7 Fortlaufende Integration

Der Code wird stündlich bis täglich integriert, dabei integriert immer nur jeweils ein Entwicklerpaar eine Task. Nach der Integration werden alle Unit-Tests und alle funktionalen Tests gestartet. Falls dann ein Test fehlschlägt, ist die Fehlerquelle (hoffentlich) leicht auszumachen.

22.5.8 Refactoring

Das Design der Anwendung wird ständiges überarbeitet und angepasst. Sobald ein Entwicklerpaar eine Möglichkeit sieht, etwas zu vereinfachen, sollte dies auch geschehen. Die Tests helfen dabei, vorhandene Funktionalität nicht zu zerstören.

22.5.9 Programmieren in Paaren

Es wird immer zu zweit programmiert, das heißt vor einem Rechner sitzen jeweils zwei Programmierer. Der eine implementiert, während der andere einen sofortige Code-Review vornimmt. Die Paarungen ändern sich häufig, so dass sich Wissen über bestimmte Teile des Codes schnell im Team verbreitet. Das hat zur Folge, dass kein Code existiert, mit dem sich nur eine einzige Person auskennt.

22.5.10 Quellcode ist Gemeinschaftseigentum

Damit die in Abschnitt 22.5.8 bezüglich des Refactoring angegebenen Vorgaben umgesetzt werden können, ist der Quellcode Gemeinschaftseigentum. Das bedeutet, dass grundsätzlich jeder das Recht hat, alles zu ändern. Damit übernimmt dann auch jeder Entwickler Verantwortung für das ganze System und nicht nur für „seinen“ Teil. Wichtige Voraussetzung dazu ist auch das Programmieren in Paaren (Abschnitt 22.5.9).

Andererseits bedeutet dies natürlich nicht, dass man mit dem eigentlichen Urheber des Codes grundsätzlich keine Rücksprache zu halten braucht, bevor man etwas ändert (siehe auch Abschnitt 22.3, Kommunikation).

22.5.11 Programmierstandards

Wenn jeder Entwickler prinzipiell jede Programmzeile bearbeiten kann, wie in Abschnitt 22.5.10 beschrieben, sind Programmierstandards unabdingbar. Dazu gehören beispielsweise Code-Layout und Namenskonventionen.

22.5.12 Vierzigstundenwoche

Laut Beck sollte ein XP-Team nie mehr als eine Woche mit Überstunden in Folge machen, da ausgeruhte Programmierer bessere Leistungen bringen. Kurzfristig kann man mit Überstunden sicherlich die Projektgeschwindigkeit anheben. Wenn Überstunden allerdings zum Dauerzustand werden, leidet die Qualität des Codes darunter erheblich, was das Projekt insgesamt sogar verlangsamen kann. Ebenso wichtig ist es, dass die Entwickler sich während ihrer Arbeitszeit auf das ein Projekt konzentrieren und nicht etwa an mehreren Projekten gleichzeitig arbeiten.

22.6 XP-Prinzipien

Dtsttcpw und Develop for today Zwei Dogmen der XP-Gemeinde sind „do the simplest thing that could possibly work“ und „Develop for today“. Das bedeutet, dass man dem Drang, die allgemeingültigste Lösung zu implementieren nicht nachgehen soll. Stattdessen sollte man sich auf die Anforderungen konzentrieren, die zum gegenwärtigen Zeitpunkt bekannt und klar definiert sind. Wenn man eine Lösung zu entwickelt, die auch zukünftige Probleme behandelt, von denen man nicht sicher weiß, ob sie überhaupt auftreten, verschwendet man gegebenenfalls Zeit und Arbeit.

Verantwortung übernehmen Üblicherweise weist ein übergeordneter Manager den einzelnen Entwicklern Verantwortlichkeiten zu. Es kann jedoch vorkommen, dass diese sich damit überfordert oder unwohl fühlen. XP rät dazu, dieses Problem anders herum anzugehen, in dem die Projektteilnehmer Verantwortlichkeiten für bestimmte Aufgaben aus ihrer eigenen Initiative heraus übernehmen.

Mit leichtem Gepäck reisen Es wird keine umfangreiche Dokumentation erstellt. Eine solche tendiert ohnenhin dazu, nach wenigen Wochen veraltet zu sein. Durch das Programmieren in Paaren sollte es für jeden Programmteil mehrere Leute geben, die sich damit auskennen. Daher ist eine ausführliche Dokumentation nicht mehr unbedingt erforderlich.

Gegebenheiten anpassen XP ist nicht dogmatisch. Jedes Entwicklerteam muss den Prozess auf sich und seine speziellen Gegebenheiten anpassen. Wenn das Team feststellt, dass eine bestimmte Vorgehensweise so nicht funktioniert, wird diese modifiziert oder abgeschafft. Dieser Anpassungsprozess kann evtl. das ganze Projekt über andauern.

Geringe Investition Sollte ein Projekt scheitern oder aus anderen Gründen aufgegeben werden, dann ist es sinnvoll, wenn nicht allzuviel in das Projekt investiert wurde. Vor allem

am Anfang eines Projektes sollte daher auf geringe Investitionen geachtet werden, nicht nur finanziell.

Auf Sieg spielen XP-Entwickler sollen offensiv an eine Aufgabe herangehen. Viele sind darauf bedacht, möglichst keine Fehler in eine Software einzubauen. Allerdings lassen sich Fehler auch bei größter Sorgfalt nicht vermeiden. Daher ist keine übertriebene Vorsicht, sondern der Mut zu Experimenten gefragt. Damit keine Fehler in das Projekt integriert werden (vgl. Abschnitt 22.5.7), ist das Testen (22.5.6) unabdingbar.

22.7 Fazit

Aufgrund der engen Zusammenhänge zwischen den einzelnen XP-Techniken ist fraglich, ob XP auch erfolgreich sein kann, wenn nur ein Teil der Techniken angewandt wird. Beck stellt mit XP einen sehr idealistischen Ansatz vor, dessen Realisierung selbst eingespielte Entwicklerteams vor eine erhebliche Herausforderung stellt. Bei der Einführung von XP sollte daher zum einen schrittweise vorgegangen werden, d.h. die Techniken sollen nach und nach eingeführt werden. Außerdem sollte der Entwicklungsprozess ständig auf das Entwicklerteam angepasst werden. Ob die Anwendung aller XP-Techniken im Sinne von Beck möglich ist, bleibt fraglich und sicherlich stark von den Möglichkeiten der am Projekt beteiligten Parteien abhängig.

Speziell in der Projektgruppe sind einige Techniken nicht oder nur unter Schwierigkeiten zu verwirklichen. Welche Auswirkungen XP auf den Entwicklungsprozess und auf das Produkt selbst sowie auch auf die PG-Teilnehmer und die Betreuer hat, wird sich im Laufe der Implementierungsphase zeigen.

Kapitel 23

Testen bei XP mit JUnit

Autoren: *Stefan Borggraefe, Tobias Wolf*

In diesem Kapitel wird das Softwaretesten mittels des Test First-Ansatzes vorgestellt. Zunächst wird im Abschnitt 23.1 die Rolle des Testens bei XP erläutert und kurz angeschnitten, was XP überhaupt ist. Im Abschnitt 23.2.1 werden einige Techniken des Testens erklärt. Beim Testen ist es besonders wichtig, Randfälle zu berücksichtigen, da durch das Testen mehrerer Randfälle häufig die Fälle dazwischen mitabgedeckt werden. Dies wird im Abschnitt 23.2.2 genauer erörtert.

Refactoring (s. Abschnitt 23.2.3) ist eine Technik, die darauf abzielt, die Verständlichkeit und Wartbarkeit von Code zu verbessern, ohne sein äußeres Verhalten zu ändern und neue Fehler einzubauen. Um diese Technik mit einem vertretbaren Risiko anwenden zu können, muss eine ausreichende Anzahl Tests vorhanden sein, die nach jeder Änderung ausgeführt werden. Im Falle von Java können solche Tests beispielsweise mit JUnit (s. Abschnitt 23.3) erstellt werden. Eine Erweiterung von JUnit für das Testen grafischer Benutzeroberflächen ist Abbot, welche in Abschnitt 23.3.3 vorgestellt wird.

23.1 Einleitung

Die Entwicklungsmethode eXtreme Programming (XP, siehe [10]) beschreitet in vielen Bereichen der Softwareentwicklung neue Wege. Eine unscharfe Metapher wird einer aufwändigen Designphase vorgezogen. Mit dieser Metapher vor Augen können die Entwickler sofort mit der Implementierung beginnen. Dabei soll nicht weit in die Zukunft geplant werden, sondern in kleinen Schritten die gerade aktuellen Aufgaben erledigt werden, da davon ausgegangen wird, dass die Anforderungen in fernerer Zukunft ohnehin starken Veränderungen unterworfen sein werden.

Damit dieses Vorgehen nicht ins Chaos führt, gibt es insgesamt zwölf Prinzipien, die den Prozess koordinieren. Testen stellt hierbei eine der zentralsten Tätigkeiten dar. Immer wenn ein kleiner Schritt fertiggestellt wurde, soll getestet werden um die Korrektheit des Schrittes zu ermitteln.

Beim Test First-Ansatz (siehe [12]) wird das Entwickeln von Tests als erster Punkt innerhalb eines solchen Schrittes durchgeführt – es werden zuerst die Tests und danach der zu implementierende Code geschrieben. Die zu erreichenden Ziele lassen sich durch eine Steigerung der Qualität der Software und eine Steuerung des Designs formulieren. Das Design soll hierbei auf Testbarkeit und Einfachheit optimiert werden. Das Vorgehen lässt sich durch folgende Schritte beschreiben:

- Es entsteht zuerst der Test, dann der eigentlich zu implementierende Code.
- Zunächst schlagen alle Tests fehl, da der zu testende Code noch nicht existiert. Im Verlauf der Implementierung funktionieren immer mehr Tests.
- Laufen alle Tests, sollte der eigentliche Code funktionieren und fertig sein.
- Laufen alle Tests, es werden aber trotzdem noch Fehler im Code gefunden, war die Testabdeckung nicht hoch genug. Es müssen noch weitere Tests geschrieben werden, die die aufgetretenen Fehler abdecken und der eigentliche Code so lange weiterentwickelt werden, bis diese Tests ebenfalls durchlaufen.
- Es wird in „Mikroiterationen“ entwickelt, jede Iteration dauert maximal 10 Minuten.

Die Vorteile bei dieser Methode liegen u.a. darin, dass am Ende jedes Stück Code getestet worden ist und das Design der Software durch die Tests bestimmt wird.

23.2 Allgemeine Testkonzepte

23.2.1 White und Black Box Testen

Wenn man einen Test entwirft, kann man nach unterschiedlichen Methoden vorgehen. Beim White Box-Testen kann man sich die zu testende Unit als einen transparenten Kasten vorstellen. Man hat also vollen Zugriff auf die Implementierung. Daher kann man sich bei der Erstellung der Testfälle an den Kontroll- und Datenflüssen orientieren. Der Datenfluss wird durch Erzeugen und Zerstören von Variablen und Zuweisungen an Variablen bestimmt. Ein Kontrollfluss eines Programmes wird durch seine Anweisungen und Verzweigungen definiert. Je nachdem, wie viele Kontrollflüsse ausgeführt werden, können verschiedene Grade von Testabdeckungen identifiziert werden:

C_0: Jede Anweisung wird mindestens einmal ausgeführt.

Dieses Kriterium ist recht schwach, da zum Beispiel logische Ausdrücke nicht unbedingt vollständig überprüft werden.

C_1: Alle Segmente werde mindestens einmal ausgeführt.

C_{1+}: Wie C_1, Schleifen werden mit Extremwerten getestet.

C_{1p}: Alle Segmente werden getestet, logische Ausdrücke werden so getestet, dass jede logische Bedingung einmal getestet wird.

Die Kriterien C_1,C_{1+} und C_{1p} sollten bei jedem White Box-Test zu erreichen sein.

C_2: Alle Segmente werden getestet. Schleifen müssen so getestet werden, dass sie 0-mal, mit wenigen Iteration und mit hohen Iterationen ausgeführt werden

Dieses Kriterium kann unter Umständen zu einer sehr hohen Zahl von Testfällen führen. Dafür bietet es hohe Chancen, Fehler im Kontrollfluss zu finden.

C_{ik}: Alle Segmente werden mindestens einmal getestet. Auftretende Schleifen werden mit allen Kombinationen von i und k getestet.

C_t: Es sollen alle möglichen Kombinationen von Pfaden getestet werden.

Die Kriterien C_{ik} und C_t sind fast immer nicht zu erfüllen, da die Anzahl der nötigen Testfälle sehr schnell gegen Unendlich geht.

Beim White Box-Testen wird vor allem die Implementierung eines Moduls getestet. Dies hat zur Folge, dass nicht gewährleistet ist, dass die Spezifikation richtig umgesetzt worden ist. Es können auch zusätzliche Funktionen implementiert worden sein, die nicht in der Spezifikation vorkommen. Solange das Modul in sich logisch und konsistent ist, werden keine Fehler entdeckt.

An dieser Stelle tritt der Black Box-Test auf die Bühne. Für einen Test nach dem Black Box-Schema ist ein Modul ein schwarzer Kasten. Er kann also nicht auf die jeweilige Implementierung schauen und ist auf die nach außen sichtbare Schnittstelle angewiesen. Zum anderen muss man die Spezifikation heranziehen, um zu erfahren, welche Funktion das Modul erfüllen soll. Wir ziehen uns also von der Entwicklerrolle auf eine Benutzerrolle zurück und erwarten nur, dass das Modul die spezifizierte Funktion korrekt erfüllt. Wie sie das macht, ist für den Test egal.

Der Ursprung dieser beiden Testarten liegt vor der Einführung von XP (siehe [10]) und dem Test First-Ansatz. Trotzdem werden sie beim Test First-Ansatz auf eine recht intuitive Art benutzt. Da zuerst die Tests entwickelt werden, wird automatisch eine recht hohe Testüberdeckung erreicht. Außerdem bleibt man immer sehr nahe an der Spezifikation, da die Tests auf dieser basieren.

23.2.2 Grenzfälle

Wenn ein Test geschrieben wird, muss irgendwann überlegt werden mit welchen Werten getestet werden soll. Gerade an die Ränder des erlaubten Wertebereichs bilden besondere Fehlerquellen. Daher sollten sie näher in Augenschein genommen werden. Um für eine bestimmte Stelle im Modul den erlaubten Wertebereich zu bestimmen, muss die Implementierung des Moduls analysiert werden. Als zu testende Größen kommen Eingabeparameter, Größe von Eingabedateien, Anzahl der Aufrufe einer Funktion, Variablen in logischen Ausdrücken usw. in Betracht. Sind die Grenzbereiche ausreichend durch Testfälle abgedeckt, sollten weitere Testfälle entwickelt werden. Diese gehen dann noch einen Schritt weiter, überschreiten die Grenzen oder übergeben absichtlich falsche an die zu testenden Klassen. Dieses entspricht der

Philsophie mit einem Tests nicht die Korrektheit, also das Nicht-Vorhandensein von Fehlern zu beweisen, sondern das Gegenteil.

Dieses Vorhaben beim Test First-Ansatz umzusetzen, klingt zunächst nicht einfach. Schließlich gibt es noch keinen Code gegen den angekämpft werden kann. Im Gegenteil, der Code wird so geschrieben, dass er den Anforderungen des Tests genügt. Und hier schließt sich wieder der Kreis. Je härter die Tests ausgelegt sind, je besser aus der Spezifikation die Grenzfälle, erlaubten und nicht erlaubten Werte herausgelesen werden, desto besser wird auch der fertige Code.

23.2.3 Refactoring

Als Refactoring bezeichnet man Änderungen an einem Softwaresystem, die das das externe Verhalten nicht verändern, aber die Verständlichkeit und Wartbarkeit des Codes verbessern ([31], S.53). Um sicherzustellen, dass sich das externe Verhalten durch ein Refactoring wirklich nicht ändert und weiterhin die gewünschte Funktionalität gegeben ist, stellen automatisierte Tests ein wichtiges Werkzeug dar.

Es können folgende Maßnahmen identifiziert werden, die im Folge eines Refactoring durchgeführt werden können:

- Umbenennung: Werden Methoden, Klassen oder Packages umbenannt, so müssen auch die entsprechenden Tests umbenannt werden.
- Entfernen von Parametern: Dann müssen auch die entsprechenden Testfälle angepasst werden. Manchmal fallen dann Testfälle weg, wenn die gerade diesen bestimmten Parameter überprüfen.
- Hinzufügen von Parametern: Die Methodenaufrufe der Tests muss geändert werden und eventuell müssen auch TestSuites angepasst werden um den neuen Parameter zu berücksichtigen.
- Extraktion einer Klasse: Die entsprechenden Tests müssen mitverschoben werden. Zusätzlich kommen Interaktionstest zwischen der neuen und der ursprünglichen Klasse hinzu.
- Einverleibung einer Klasse: Auch hier müssen Tests verschoben werden, allerdings können hier Tests überflüssig werden.
- Verschieben einer öffentlichen Methode: dies hat ähnliche Auswirkungen wie die Extraktion einer Klasse.
- Änderung der Implementierung: die hat keine Auswirkung an Black-Box-Test, allerdings müssen alle implementierungsabhängige Testfälle neu überdacht werden.

Beim Refactoring ist ein Vorgehen in möglichst kleinen Schritten ratsam. Nachdem der Code umgebaut wurde, muss überprüft werden welche Tests weiterhin Gültigkeit besitzen und welche Tests geändert oder verschoben werden müssen. Dabei erfolgt nach jedem abgeschlossenen

Teilschritt ein Start der verbliebenen Tests um die Korrektheit der bisherigen Schritte zu überprüfen. Ist das Refactoring abgeschlossen wird der Programmierer abermals zum Tester und überprüft alle verbleibenden Testfälle auf Adäquatheit und Redundanz. Im Laufe des Softwareentwicklungsprozesses werde immer wieder größere Systemänderungen vorkommen. Diese können zur Folge haben, dass einige Tests entfernt werden müssen, um durch neue ersetzt zu werden. Aber natürlich kann auch diesem Umstand ein positiver Aspekt abgezwungen werden, da immer wieder die Annahmen der Vergangenheit neu überdacht werden müssen.

23.2.4 Unit- und Integrationstests

Ein Unittest testet eine in sich geschlossene Einheit des Softwaresystems. Andere Einheiten die die zu testende Einheit aufrufen, oder von dieser aufgerufen werden, werden durch *Stellvertreter* und *Treiber* ersetzt.

Stellvertreter werden von der zu testenden Einheit benutzt und liefern meistens nur Standardwerte zurück oder melden den Zeitpunkt ihres Aufrufs. Damit kann überprüft werden, ob Werte richtig weiterverwendet werden oder zum richtigen Zeitpunkt bestimmte Funktionen aufgerufen werden. *Treiber* versorgen die zu testende Klasse mit Werten und Methodenaufrufen. Damit wird in erster Linie überprüft, ob die implementierten Methoden in erwarteter Weise arbeiten.

Sind eine Reihe von Einheiten implementiert und getestet worden, werden Integrationstests durchgeführt. Diese testen hauptsächlich die Schnittstellen und die Interaktion zwischen diesen Einheiten. Wird eine Menge von Einheiten wieder als eine Einheit betrachtet, gelangt man schnell zu Unit Tests. Mit Fortschreiten der Implementierung werden Integrationstest dieser großen Einheiten notwendig.

23.3 Unit-Tests unter Java mit JUnit

JUnit (siehe [11]) ist eine überschaubares Framework zur Unterstützung von Unit-Tests in Java. Erich Gamma und Kent Beck, die beiden Autoren von JUnit, waren zum einen sehr von dem Nutzen von Unit-Tests überzeugt, mussten aber andererseits feststellen, dass die meisten Entwickler das Schreiben von Tests eher als unnötigen Zusatzaufwand empfinden. Daher war eines der zentralen Designziele von JUnit, es so einfach wie möglich zu machen, Unittests zu schreiben, um diesen Zusatzaufwand zu reduzieren.

Durch diese Einfachheit auf der einen und der Tatsache, dass es sich um ein kostenloses Open Source-Programm handelt, hat JUnit inzwischen eine weite Verbreitung unter Java-Entwicklern gefunden und ist inzwischen in zahlreiche Entwicklungsumgebungen integriert.

23.3.1 Aufbau des JUnit-Frameworks

Das JUnit-Framework besteht aus etwa zwei Dutzend Klassen, von denen die wichtigsten durch das Klassendiagramm in Abbildung 23.1 dargestellt werden. Im Grunde ist die Aufgabe von JUnit eine sehr simple: führe einige Tests aus und zeige die Ergebnisse an. Da JUnit dar-

überhinaus jedoch auch als ein flexibles und erweiterungsfähiges Framework für das Erstellen von Tests entworfen wurde, ist die Struktur im Vergleich zu der einfachen Aufgabe relativ komplex. Die folgende Beschreibung von JUnit ist angelehnt an [35].

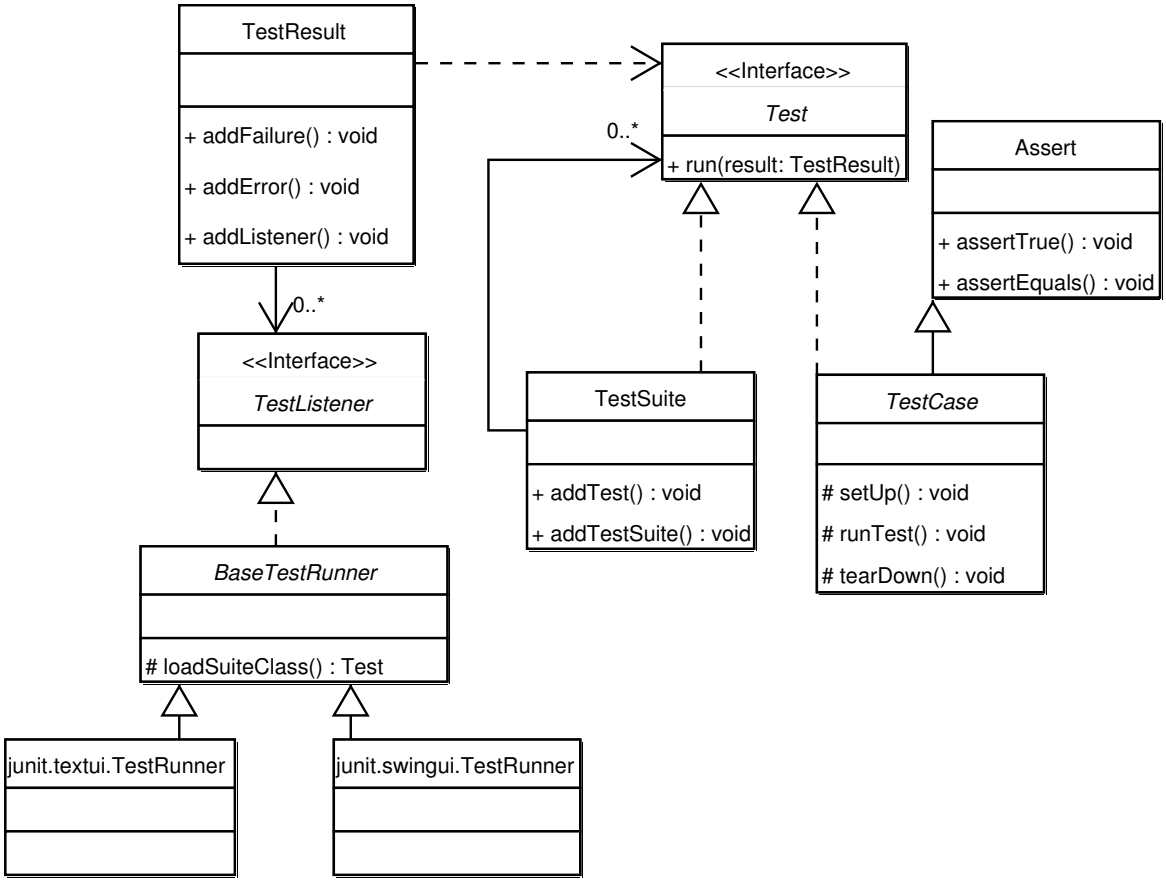

Abbildung 23.1: Klassendiagramm der zentralen Klassen des JUnit-Framework

Um einen Test mit Hilfe von JUnit zu schreiben, muss man die Klasse TestCase erweitern und die Methode runTest() überschreiben. In dieser Methode kann beliebiger Java-Code ausgeführt werden. Außerdem stehen einige spezielle Methoden zum Überprüfen von Bedingungen, die erfüllt sein müssen zur Verfügung. Solche Bedingungen werden neudeutsch „Assertions“ genannt und dementsprechend fangen auch die Namen der Methoden zum Abprüfen der Bedingungen mit dem Wort assert... an. Eine typische JUnit-Assertion sieht folgendermaßen aus:

```
assertTrue("Der Öldruck ist zu hoch.", getOeldruck() < MAX_OELDRUCK);
```

Ist eine in einer Assertion geforderte Bedingung nicht erfüllt, wird ein AssertionFailedError geworfen und der Test, in der dieses Ereignis aufgetreten ist gilt als fehlgeschlagen (als „Failure“). Möchte man einen Test unabhängig von einem assert-Befehl fehlschlagen lassen, kann man sich der Methode fail() bedienen. Damit kann man beispielsweise überprüfen, ob Exceptions so wie erwartet auftreten:

```
try
{
  File datei = new File("Nicht-existierende-Datei.txt");
  fail("Erwartete IOException ist nicht aufgetreten.");
}
catch (IOException e) { }
```

Bei einem Failure handelt es sich um eine Art Sollbruchstelle. Sie werden dort erzeugt, wo der Autor eines Tests mit einem möglichen Fehlverhalten der Software gerechnet hat. Treten nicht vorgesehene Fehler auf, werden diese von JUnit als „Error“ vermerkt.

Weiterhin hat man die Möglichkeit die beiden Methoden setUp() und tearDown() zu überschreiben. In der Methode setUp() sollte dabei Code zum Aufsetzen einer Testumgebung enthalten sein, beispielsweise das initialisieren bestimmter Datenstrukturen oder Ressourcen. Diese Testumgebung wird auch „Fixture“ genannt. Die Methode tearDown() hat die entgegengesetzte Aufgabe und kann zum Aufräumen nach dem Ausführen von Testmethoden benutzt werden. Das Ausführen dieser Methoden in der richtigen Reihenfolge übernimmt die Template-Methode ([36], S.366) run(), die vom Prinzip her folgendermaßen aussieht:

```
public void run(TestResult result)
{
  setUp();
  runTest();
  tearDown();
}
```

Die drei aufgerufenen Methoden können von Unterklassen nach Bedarf überschrieben werden.

Wäre dies jedoch die einzige Möglichkeit zum Programmieren von Tests, wäre man gezwungen, für jeden einzelnen Test eine neue Klasse zu erstellen, die von TestCase erbt und zumindest die Methode runTest() überschreibt. Das würde bei einer guten Testabdeckung schnell unübersichtlich und unkomfortabel werden. Da es eines der Design-Ziele von JUnit war, den Entwickler möglichst gut beim Schreiben von Tests zu unterstützen, gibt es noch eine weitere Möglichkeit, die zu befürchtende Klasseninflation zu verhindert. Dabei erbt man weiterhin von TestCase und kann setUp() und tearDown() wie gewohnt überschreiben. Statt nun aber die Methode runTest() zu überschreiben, erstellt man bei dieser Möglichkeit beliebig viele Methoden programmieren, die einen Methodennamen besitzen, der mit der Zeichenfolge „test“ beginnt, die eine leere Parameterliste besitzen und void als Rückgabetyp haben. Beim Programmieren dieser Methoden geht man einfach davon aus, dass sie genauso ausgeführt werden wie eine runTest()-Methode, also eingerahmt von einem setUp() und einem tearDown()-Aufruf. Wie das intern realisiert ist, wird im nächsten Abschnitt näher erläutert.

Die Klasse TestSuite sorgt dafür, dass dies wirklich so passiert. Hauptsächlich dient sie dazu, mehrere Tests zusammenzufassen und gemeinsam ausführen zu können. Es handelt es sich dabei um ein Kompositum ([36], S.239) von Tests, ein Interface, das sowohl von TestCase als auch von TestSuite implementiert wird. Um einer TestSuite Tests hinzuzufügen, gibt es zwei Möglichkeiten: Zum einen kann durch die Methode addTest(Test test) der Klasse TestSuite ein einzelner Test einer Suite beigefügt werden. Zum anderen kann die Methode addTestSuite(Class

testCaseClass) benutzt werden. Diese extrahiert mit Hilfe von Reflection aus einer TestCase-Klasse sämtliche Methoden, die mit der Zeichenfolge „test“ beginnen, eine leere Parameterliste besitzen und void als Rückgabetyp haben. Beim Ausführen ihrer run()-Methode sorgt die Klasse TestSuite dafür, dass die einzelnen Tests, die zu ihr gehören, einer nach den anderen abgearbeitet und die mit „test“-beginnenden Methoden in TestCases genau in der Art wie am Ende des letzten Absatzes gefordert ausgeführt werden.

Wenn man nun einige Tests programmiert hat, ist man natürlich auch an den Ergebnissen interessiert. Grundsätzlich gilt dabei, dass ein funktionierender Test keinerlei Ausgaben machen sollte und ein fehlgeschlagener Test möglichst präzise und knapp die Ursache des Fehlschlags darstellen sollte. Durch Beachtung dieses Prinzips, ist ein Entwickler nicht dazu gezwungen, sich bildschirmweise Ausgaben anzusehen, nur um letztlich festzustellen, dass alle Tests erfolgreich verlaufen sind. Bei JUnit werden die Ergebnisse von Tests in einem TestResult-Objekt gesammelt, dass als Parameter an alle von einer TestSuite ausgeführten run()-Methoden übergeben wird. Dieses TestResult-Objekt „besucht“ also sozusagen die einzelnen Tests — ein Beispiel für das Collecting Parameter Pattern ([9], S. 75). Im Falle eines erfolgreich verlaufenen Tests passiert nicht viel mehr, als dass ein Zähler im TestResult-Objekt um Eins erhöht wird. Schlägt ein assert()-Befehl fehl, wird eine AssertionFailedException geworfen und dem TestResult-Objekt als Failure hinzugefügt. Alle anderen Exceptions, die auftreten sind auch im Falle eines fehlgeschlagenen Tests nicht erwartet und werden dem TestResult-Objekt als „Error“ hinzugefügt.

Nun wäre alles beisammen, was zum Ausführen von Tests benötigt wird, allerdings fehlt noch die Möglichkeit, sich die im TestResult-Objekt gesammelten Ergebnisse anzeigen zu lassen. Dazu gibt es bei JUnit sog. TestRunner, die alle von der Klasse BaseTestRunner erben. Bei JUnit fest enthalten sind drei Unterklassen, die eine text-, eine AWT- und eine Swing-basierte Oberfläche zum Anzeigen von Testergebnissen enthalten. Die Klasse BaseTestRunner implementiert die Schnittstelle TestListener und kann sich damit an einem TestResult-Objekt registrieren. Die Methode startTest() von TestResult informiert sämtliche registrierte Listener darüber, dass ein neuer Test begonnen hat. Dadurch werden die TestRunner in die Lage versetzt, den Fortschritt während der Abarbeitung einer Anzahl von Tests kontinuierlich anzuzeigen.

Nach diesen genaueren Erläuterungen, hier noch einmal die oben gezeigte Template-Methode der Klasse TestCase zum Ausführen eines Tests, die nun bis aus einige Details mit dem JUnit-Quelltext übereinstimmt und die bisherigen Erklärungen noch einmal gut zusammenfasst:

```
public void run(TestResult result)
{
  // Nächster Test -> Listener informieren
  result.startTest(this);

  setUp();

  try
  {
    runTest();
  }
  catch (AssertionFailedError e)
```

```
    {
      result.addFailure(this, e);
    }
    catch (Throwable t)
    {
      result.addError(this, t);
    }
    finally
    {
      tearDown();
    }
  }
```

JUnit ist hauptsächlich für das Erstellen von Black Box-Tests ausgelegt. White Box-Tests werden mit der Erweiterung JUnitX möglich (kostenlos erhältlich unter [41]).

23.3.2 Entwickeln von Tests

Nachdem im letzten Abschnitt ein kurzer Überblick über den Aufbau von JUnit gegeben wurde, soll nun noch einmal erläutert werden, wie typischerweise ein Test mittels JUnit entwickelt werden sollte.

Wird der Test First-Ansatz verwendet, strebt man an die Tests möglichst früh zu erstellen. Es ist allerdings erforderlich, dass zumindest die Schnittstellen der zu testenden Klassen schon vorhanden sind, bevor mit dem erstellen eines Tests mit JUnit begonnen werden kann. Der Grund dafür ist, dass ein Test Methoden der zu testenden Klassen aufrufen muss. Sind diese noch nicht vorhanden, ist es nicht möglich zu kompilieren.

Hat man nun also zumindest die Schnittstellen der zu testenden Klassen vorliegen, kann damit begonnen werden, den Test zu entwickeln. Dazu wird die Klasse TestCase erweitert. Für das Aufsetzen einer Testumgebung überschreibt man die Methode setUp(). Auf dieser Testumgebung können dann Operationen während der Tests ausgeführt werden.

Für die eigentlichen Tests überschreibt man entweder die Methode runTest() oder erstellt eine oder mehrere Methoden, die mit der Zeichenkette „test“ im Methodennamen beginnen. Innerhalb dieser Methoden führt man Operationen auf der in setUp() erstellten Testumgebung auf und überprüft die Ergebnisse davon mit den verschiedenen assert...()-Methoden. Durch die im Abschnitt 23.3.1 erläuterten Mechanismen wird vor jedem Test die Methode setUp() ausgeführt und somit jeweils eine frische Testumgebung erstellt. Mit der Methode tearDown() verhält es sich ähnlich. In ihr kann Code untergebracht werden, der nach jedem Test ausgeführt werden soll.

Es ist empfehlenswert, die Testklasse einer zu testenden Klasse im gleichen Paket unterzubringen, damit Zugriff auf paketprivate Methoden und Attribute der zu testenden Klassen besteht. Um die Tests eines Projekts zusammenzufassen, ist es außerdem sinnvoll, pro Paket eine TestSuite zu definieren und diese Suites wiederum zu größeren Suites und schließlich zu einer, die sämtliche Tests des Projekts enthält zusammenzufassen. Diese TestSuite kann dann z.B. per Konvention vor jedem Einchecken von neuem Code in eine Versionsverwaltung von

den Entwicklern ausgeführt oder noch besser durch ein Continuous Integration-Tool (siehe [32] automatisiert werden.

23.3.3 GUI-Tests mit der JUnit-Erweiterung Abbot

Für das Schreiben von GUI-Tests bietet JUnit von Haus aus leider keine spezielle Unterstützung. Glücklicherweise ist JUnit als erweiterbares Framework ausgelegt und so gibt es unter anderem mehrere Erweiterungen für das effiziente Entwickeln von Oberflächen-Tests. Eine davon ist Abbot (erhältlich unter [80]).

Der Name ist eine Abkürzung für „a better bot“ und spielt auf die Klasse java.awt.Robot der Standard-API an. Dabei handelt es sich um eine Klasse zum Fernsteuern von GUI-Operationen. Abbot enthält ebenfalls diese Funktionalität und erhebt mit seinem Namen den Anspruch, diese besser umzusetzen. Der Hauptvorteil von Abbot liegt dabei neben der JUnit-Integration darin, dass es Fernsteuerungskommandos auf einer höheren semantischen Ebene versteht. Während der AWT-Robot lediglich Kommandos der Art „Gehe zu Punkt 100,100, Klicke links, usw.“ versteht, kann Abbot mit zahlreichen Oberflächen-Elementen direkt umgehen und Befehle wie „Wähle Punkt ‚Blau' aus der JComboBox ‚Farbauswahl'.“ verarbeiten. Um dies zu erreichen, bietet Abbot zahlreiche Klassen (sog. „Tester“), um Aktionen auf spezielle GUI-Komponenten auszuführen. Mit der Methode ComponentTester.getTester(Component c) kann man ein passendes Tester-Objekt für ein GUI-Objekt erhalten. Auf dem Tester-Objekt können dann durch das betreffende GUI-Objekt unterstützte Aktionen ausgeführt werden. Beispielsweise bietet die Klasse JListTester Methoden JList-spezifische Methoden wie actionScrollCellToVisible(), actionSelectIndex(), actionSelectValue() und getContents().

Neben den speziellen Klassen zum Fernsteuern von GUI-Elementen enthält Abbot außerdem noch ein Tool zum *Aufzeichnen* einer Reihe von Benutzereingaben. Diese Aufzeichnung kann dann in eine XML-Datei abgespeichert werden und als Teil eines Tests abgespielt werden. Um diese Funktionalität in JUnit zu integrieren, bringt Abbot Erweiterungen der Klassen TestCase und TestSuite mit, die die Namen ScriptTestCase und ScriptTestSuite tragen.

Das Erstellen der Tests funktioniert prinzipiell genau so wie auch ohne die Abbot-Erweiterung. Die Methoden setUp() und tearDown() werden wie zuvor genutzt. Die eigentlichen Testmethoden überprüfen mit den assert...()-Befehlen die Ergebnisse der ausgeführten Operationen, wobei es sich dabei nun auch um Interaktionen mit einer grafischen Benutzeroberfläche handeln kann. Dadurch, dass Abbot TestCase und TestSuite erweitert, lassen sich die GUI-Tests ohne Probleme mit den Standard-JUnit-Tests in einem Projekt integrieren.

23.4 Fazit

Der Test First-Ansatz soll Entwickler dazu anregen, sich zunächst zu überlegen was genau implementiert werden soll. Durch die Erstellung eines Testrahmens können viele Stolpersteine und kritische Stellen im späteren Code im vorraus erkannt werden. Es mag nach einer Menge zusätzlichem Aufwand aussehen, und in der Anfangsphase sicherlich länger dauern. Diese Zeit wird aber später eingespart, da aufwändige Debugging-Phasen schneller zum Ziel führen oder ganz wegfallen. Wichtig ist, sich selber zu bremsen und die kleinen Iterationschritte

einzuhalten. JUnit ist das Tool der Wahl, wenn es um das Erstellen von Unit-Tests unter Java geht.

Teil IV

Einsatz von XP

Kapitel 24

Einsatz von XP

Autor: *Markus Niehammer*

24.1 Der Weg zum eXtreme Programming

Die Entscheidung, in der Implementierungsphase XP als einen neuen, unkonventionellen Entwicklungsprozess einzusetzen, mag zunächst recht riskant erscheinen. Andererseits ist es eine große Herausforderung, diese neue Art der Softwareentwicklung zu erproben und ein XP-Projekt erfolgreich zu Ende zu bringen. Zunächst hat es natürlich naheliegende Vorteile, auf konventionelle und erprobte Entwicklungsprozesse zurückzugreifen, die den Erfolg eines Projekts zu garantieren scheinen. Auch die enorme Planungs-, Analyse- und Entwurfsarbeit, die bereits durchgeführt wurde, verspricht nach einem bekannten Modell eine einfache Implementierung. Mit dem Einsatz von XP wird ein Großteil dieser Vorarbeiten ausgeklammert, was von allen Beteiligten eine Menge Mut erfordert. Das Projekt startet von Grund auf neu und entwickelt sich direkt während der Implementierung. Zwar kann der bereits fertige Entwurf die Orientierung erleichtern und eine Art Leitfaden darstellen, doch lässt es sich nicht vermeiden, vieles einfach zu verwerfen und im XP-Prozess neu zu gestalten.

Das eXtreme Programming stellt hohe Anforderungen an das Entwicklerteam. Auch stellt sich die Frage, ob und wie die verschiedenen Techniken des XP eingesetzt werden können. Möglicherweise können Vorarbeiten nach und nach in das XP-Modell eingebaut werden, so dass sich der Entwicklungsprozess selbst an die konkrete Aufgabe ständig anpasst. XP-Techniken können abgeändert, ergänzt oder einfach übergangen werden, wenn sich herausstellt, dass eine Technik eher hinderlich ist, als dass sie bei der Implementierung hilft. Wie im einzelnen die XP-Techniken bei der Entwicklung von HyCop zur Anwendung kommen und welche Gründe für oder gegen bestimmte Techniken sprechen, stellt dieses Kapitel dar.

24.2 Rollenverteilung und Ziele

Die klassischen XP-Rollen Kunde, Programmierer und Manager lassen sich in der Projektgruppe zunächst nicht so leicht wiederfinden. Die PG-Betreuer sichern zwar die Qualität des

Produktes, indem sie die Teilnehmer auf einen erfolgreichen Abschluß hin zu lenken versuchen. Damit nehmen sie einen Teil der Manager-Rolle ein. Die PG-Teilnehmer selbst allerdings sind zu Beginn gleichberechtigt, so dass ein Kunde im Sinn des eXtreme Programming nicht vorhanden ist. Da die eigentliche Programmieraufgabe im groben von den PG-Betreuern und detaillierter durch die Vorarbeiten im ersten PG-Semester vorgegeben wurde, fehlt ein konkreter Auftraggeber. Es wird daher beschlossen, den Kunden im XP-Prozess durch wechselnde Paare von PG-Teilnehmern zu simulieren. Pro Releasezyklus sollen zwei Personen aus dem Teilnehmerfeld ausgewählt werden, die die Aufgaben des Kunden wahrnehmen. Die übrigen Teilnehmer bilden im Sinn von XP wechselnde Entwicklerpaare. Ein Teil der Rolle des Managers fällt wie oben beschrieben den Betreuern zu.

Die vier konkurrierenden Ziele bei der Softwareentwicklung Qualität, Kosten, Zeit und Umfang spielen in der Implementierungsphase eine untergeordnete Rolle. Zwar werden mit den geplanten Releasezyklen Umfang und Zeit in einem gewissen Rahmen festgelegt, doch sind diese Parameter nicht unbedingt als so kritisch für den Projekterfolg einzustufen wie in einem industiellen Projekt. Der finanzielle Aspekt fällt in unserem Fall vollkommen weg, die zu erreichende Qualität des Produktes wird nicht von vornherein eingefordert, sondern soll sich von selbst ergeben.

24.3 Ressourcen

Natürlich stehen in einer Lehrveranstaltung nicht die Ressourcen zur Verfügung, die man in einem industriellen Projekt erwarten würde. Wie bereits erwähnt, bleiben die Ziele Produktqualität und Kosten völlig außen vor, was die Verfügbarkeit an Hardware, (kommerziellen) Programmiertools oder etwa zusätzlichen Fachkräften begrenzt. In der Implementierungsphase steht jedem Entwicklerpaar ein Computer zur Verfügung. Was die Software betrifft, so wird auf frei erhältliche Programme gesetzt, deren Verfügbarkeit kein Problem darstellt. Alle Arbeitsplätze befinden sich einem Raum, was die Kommunikation unter den Programmierern enorm erleichtert.

Da die PG-Teilnehmer natürlich noch andere Veranstaltungen neben der Projektgruppe besuchen wollen, stehen sie nicht in Vollzeit und nicht exklusiv für das Projekt zur Verfügung. Die wöchentliche Arbeitszeit ist daher für alle beschränkt. Termine zu finden, an denen im Sinn von XP alle Programmierer-Paare und Kunden auch außerhalb der PG-Sitzungen zusammenkommen, scheint zunächst schwierig. Es wird sich herausstellen, ob eine gemeinsame Arbeit zur selben Zeit am selben Ort möglich ist.

24.4 Einsatz von XP-Techniken

Die Empfehlungen zum Einsatz von XP aus der Literatur und aus Dokumentationen abgeschlossener XP-Projekte besagen, dass ein Projekt umso erfolgreicher wird, je mehr XP-Techniken realisiert werden können. Allerdings ist es möglich, die Techniken nach und nach einzuführen und natürlich auch den eigenen Bedürfnissen anzupassen. Im folgenden wird beschrieben, wie die einzelnen Techniken zum Einsatz kommen sollen. Es werden besonders die Abweichungen zum XP-Modell dargestellt.

24.4.1 Releases

Das Projekt wird in drei Releases unterteilt. Im ersten Release wird zunächst ein Prototyp von HyCop entwickelt, der die grundlegende Funktionalität unterstützt. Im zweiten und dritten Release soll der Funktionsumfang von HyCop erweitert und das Design verbessert werden.

24.4.2 Planungsspiel und Kunde vor Ort

Zu Beginn eines Releases findet das Planungsspiel ganz im Sinn von XP statt. Die Entwickler fertigen aus den Stories der (simulierten) Kunden feiner gegliederte Tasks an, die wie auch die Stories auf Karteikarten notiert werden. Die Task-Karten werden nach und nach abgearbeitet, wobei auf die Priorisierung durch die Kunden und auf den Aufwand geachtet werden soll. Der Aufwand mag aus Zeitgründen dabei oft höhere Priorität haben. Die ständige Anwesenheit des Kunden kann sich natürlich als enorm hilfreich bei Fragen oder akuten Änderungswünschen herausstellen.

24.4.3 Pair-Programming

Das Pair-Programming lässt sich von allen XP-Techniken wohl am einfachsten realisieren. Es soll dabei nicht konkret geplant werden, welche Paare zusammen programmieren; je nachdem, wer gerade anwesend ist, sollen immer neue Paare entstehen. Die Vorteile daraus sind im Hinblick auf das Gemeinschaftseigentum des Quelltextes und den Truck-Factor[1] offensichtlich.

24.4.4 Programmierstandards und einfaches Design

Die eingesetzten Entwicklungs-Tools erleichtern den Umgang mit Programmierstandards enorm. Hilfsmittel wie die automatische Formatierung des Quellcodes fördern zudem die Übersichtlichkeit und sorgen für ein einfaches Design.

24.4.5 Integration, Refactoring und Quellcode-Eigentum

Die fortlaufende Integration wird durch eine Versionsverwaltung unterstützt und so erheblich erleichtert. Durch oft wechselnde Programmierpaare soll ein ständiges Überarbeiten und ein großer Bekanntheitsgrad aller Quelltexte gegeben sein.

24.4.6 Testen

Das Testen stellt wohl das größte Problem dar und beschränkt sich im Verlauf des Projektes auf das funktionale Testen. Es werden zwar einige Komponententests geschrieben, doch macht die Einfachheit der einzelnen Projektteile das Fortführen dieser Art von Tests wohl schnell unnötig. Die funktionalen Tests dagegen, die auch von den Kunden entwickelt und durchgeführt

[1] Der Truckfactor gibt die Wahrscheinlichkeit an, dass einer der Entwickler von einem Truck überfahren wird (oder auf eine andere Weise ausfällt), und dass dadurch das Projekt scheitert

werden sollen, sind für die Qualitäts- und Funktionssicherung entscheidend. Dadurch, dass die Kunden funktionale Tests unabhängig von der eigentlichen Implementierung entwickeln, bleibt der Zweck des Test-First-Ansatzes aus dem XP-Modell erhalten. Die Entwicklung ist genau dann als erfolgreich einzustufen, wenn alle Tests erfolgreich absolviert wurden.

24.5 Besonderheiten

Gute Vorkenntnisse der Entwickler können das Projekt gerade in der Anfangsphase entscheidend voranbringen. Entwickler, die Erfahrung in bestimmten Bereichen haben, können sich gezielt an Stellen in das Projekt einbringen, wo sie benötigt werden. Das Pair-Programming mit häufig wechselnden Paaren stellt dabei sicher, dass alle Entwickler zügig auf einen einheitlichen Kenntnisstand kommen können.

Während der Prototyp-Entwicklung zu Beginn der Implementierungsphase soll sich herausstellen, ob der gewählte Entwicklungsprozess (XP) geeignet ist. Außerdem lassen sich anhand dieses Prototyps bereits mögliche Schwierigkeiten und Risiken identifizieren, und alle Entwickler bekommen eine konkretere Vorstellung vom Projekt (Systemmetapher). Sollte die Prototyp-Entwicklung fehlschlagen, so ist noch nicht viel Aufwand investiert worden und es fällt leicht, den Entwicklungsprozess zu ändern oder noch einmal von vorne zu beginnen.

24.6 Erwartungen

Die Atmosphäre innerhalb des Entwicklerteams, der Kontakt zu den Kunden und der Einsatz jedes einzelnen ist entscheidend für den Projekterfolg. Daher sollten die Entwickler sich stets an die vier Werte im eXtreme Programming Einfachheit, Kommunikation, Feedback und Mut halten. Offensichtlich fällt dies hier wesentlich leichter als in industriellen Entwicklerteams, da das Risiko begrenzt ist – vor allem finanziell und zeitlich. Zu guter Kommunikation und Feedback sind keine Hemmschwellen zu überwinden, da die Kunden nur simuliert werden und obendrein noch wechseln sollen.

Um den Verlauf des XP-Prozesses einfacher verfolgen zu können, soll die gesamte Arbeit in Form eines XP-Tagebuchs protokolliert werden. Gleichzeitig soll daraus eine Dokumentation über unsere Arbeitsweise entstehen, in der deutlich wird, inwieweit unsere Erwartungen und Planungen während der Implementierungsphase verwirklicht werden können. Außerdem wird durch das XP-Tagebuch eine Analyse des Vorgehens im Nachhinein wesentlich unterstützt.

Teil V

Das XP-Tagebuch

Kapitel 25

HyCop-Tagebuch

Autoren: *Leonore Brinker*
Evgenij Golkov
Rafael Hosenberg
Ulf Schellbach
Tobias Wolf

25.1 Einleitung

In diesem Kapitel werden die Entwicklungszyklen und ihre Motivation nach dem XP-Entwicklungsschema zwischen den mit jedem Release (siehe Abschnitt 24.4) wechselnden Kunden und Entwicklern (siehe Abschnitt 24.1) dokumentiert. Hierbei soll ein Einblick in den Einsatz von XP bei der Entwicklung von HyCop (siehe auch Abschnitt 24.3) gegeben sowie Abweichungen aufgezeigt werden.

Das Tagebuch wurde von den Kunden geführt und laufend aktualisiert. Hierbei wurden vor allem zu Beginn eines Entwicklungszyklus im Rahmen des Planungsspieles (siehe Abschnitt 24.4.1) *Stories* und *Tasks* festgehalten, um den Entwicklungsprozess zu dokumentieren.

Zu jeder Story ist eine Priorität notiert. Diese stammt aus einem Bereich zwischen 1 und 15, wobei 1 die höchste und 15 die niedrigste Priorität darstellt. Die Entwickler erstellten aus den Stories sogenannte Tasks.

Für den Endbericht wurde das Tagebuch um Fazits zu den einzelnen Zyklen erweitert. Darüber hinaus wurden die Zeitangaben zu den einzelnen Tasks aus dem Dokument entfernt, da leider keine konsequente Protokollierung stattgefunden hat. Die Struktur wurde für alle drei Releases angeglichen und die zunächst als Stichwortsammlung geschriebenen Tasks kurz ausformuliert. Außerdem wurde durch Referenzen auf Kapitel 24 „Einsatz von XP“ auf den Einsatz von eXtreme Programming in HyCop eingegangen und abschließend ein übergreifendes Fazit verfasst.

25.2 Erstes Release

Zunächst wird mit der Erstellung eines prototypischen Cockpits begonnen (siehe auch Abschnitt 24.4). Dieses soll aus zwei Anzeigen, der Tankanzeige und einer Warnleuchte, bestehen.

25.2.1 Stories

Im ersten Release wurden den Entwicklern von den Kunden folgende Stories als Anforderungen vorgelegt:

Fahrercockpit

Das Fahrercockpit soll Anzeigen darstellen können. Es soll um beliebige Anzeigen erweiterbar sein. Das Cockpit soll bei Tag und Nacht gut lesbar sein. Diese Story hat Priorität 1.

Tankanzeige

Die Tankanzeige wird im Cockpit angezeigt und nimmt Daten entgegen, die ein vorhandener Sensor periodisch liefert. Sie zeigt während des Betriebes stets den aktuellen Füllstand des Tanks an. Die graphische Darstellung soll wie auf Abbildung 25.1 dargestellt aussehen. Diese Story hat Priorität 1.

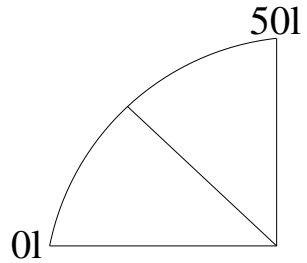

Abbildung 25.1: Tankanzeige

Tankwarnleuchte

Die Warnleuchte leuchtet auf, wenn der Tankfüllstand einen kritischen Bereich (z.B. 5l) unterschreitet. Sie muss im Cockpit angezeigt werden. Graphisch kann sie beispielsweise als Zapfsäule realisiert werden. Diese Story hat Priorität 5.

Standard- und Komfortcockpit

Es sollen zwei unterschiedliche Ansichten erstellt werden. Das Standardcockpit soll Tachometer, Drehzahlmesser, Kontrollleuchten bei Aktivität und bei Defekten, Kilometerzähler,

Tankanzeige und Warnleuchte enthalten. Das Komfortcockpit soll aus allen Elementen des Standardcockpits bestehen und zusätzlich einen Routenplaner, die Möglichkeit eine Tankstelle zu suchen sowie Checkliste und eine Oberfläche für den Zugriff auf Unterhaltungsmedien erhalten. Im Standardcockpit soll der Drehzahlmesser links vom Tacho stehen, während es im Komfortcockpit rechts stehen sollte. Diese Story hat Priorität 1.

Tachometer

Das Tachometer soll die aktuelle Geschwindigkeit des Fahrzeugs anzeigen. Gleichzeitig soll die aktuell geltende Geschwindigkeitsbegrenzung angezeigt werden, zum Beispiel als roter Strich in der Skala. Diese Story hat Priorität 1.

Drehzahlmesser

Sobald der Motor gestartet wird, soll der Drehzahlmesser sichtbar sein. Er soll stets die richtige Drehzahl anzeigen. Diese Story hat Priorität 3.

Kontrollleuchten bei Aktivität

Wenn eine der Komponenten Handbremsenwarnleuchte, Standleuchte, Abblendlicht, Fernlicht, Blinker, Nebelscheinwerfer, Nebelschlußleuchte, Rückfahrscheinwerfer oder Rücklicht aktiviert ist, soll die entsprechende Kontrollleuchte aktiviert werden. Diese Story hat Priorität 3.

Tankstellensuche

Es soll eine Tankstellensuche realisiert werden. Nach der Meldung „Tankfüllstand knapp“ entscheidet der Kunde, eine Tankstelle aufzusuchen. Es soll eine Liste von Tankstellen angezeigt werden, aus der der Fahrer eine Tankstelle auswählen kann. Hierbei sollen auf Wunsch auch detailliertere Informationen über das Angebot der einzelnen Tankstellen aufgelistet werden. Diese Story hat Priorität 4.

Routenanzeige

Eine weitere Story ist die Anzeige der Route im Routenplaner. Diese Story hat Priorität 4.

Kilometerzähler

Es sollen zurückgelegte Entfernungen im Kilometerzähler angezeigt werden können. Hierbei soll der Fahrer drei unterschiedliche Zähler zur Verfügung haben, die zum Einen die übliche, nicht veränderbare gesamte Laufleistung des Fahrzeugs sowie die Reisekilometer anzeigen, die vom Benutzer beliebig zurücksetzbar sind. Zum Anderen soll es eine Anzeige der Tageskilometer geben, die alle 24 Stunden zurückgesetzt wird. Diese Story hat Priorität 5.

Kontrollleuchten bei Defekt

Bei Nichtfunktionstüchtigkeit soll die entsprechende Kontrollleuchte für die Komponenten Batterie, ABS, ASR sowie Airbag angezeigt werden. Diese Story hat Priorität 5.

Parkplatzsuche

Weiterhin soll eine Parkplatzsuche implementiert werden. Es soll auf Wunsch eine Parkplatzliste angezeigt werden, die nach Entfernung, Kosten oder Anzahl noch vorhandener Stellplätze sortiert werden kann. Bei Auswahl eines Rastplatzes soll dieser in die Route integriert und danach die Route im Routenplaner angezeigt werden. Diese Story hat Priortät 7.

25.2.2 Abgelehnte Stories

Folgende User Stories wurden von den Entwicklern zurückgestellt:

Alternative Darstellung der Tankwarnleuchte

Um auf unterschiedliche Gewohnheiten der Fahrer einzugehen, soll eine alternative Darstellung der Tankwarnleuchte eingestellt werden können. Diese Story hat Priorität 5.

Routenplaner

Diese Story betrifft die Anzeige der Route in ausreichender (fester) Größe in einem Fenster. Darüber hinaus sollte eine Route geplant werden können, indem der Zielort eingegeben wird und zwei Zwischenpunkte ausgewählt werden. In der Anzeige der Route soll man zwischen verschiedenen Detailstufen wechseln und auf der Route mit einem Bedienteil navigieren können. Diese Story hat Priorität 1.

Staumelder

Es soll ein Staumelder entwickelt werden, der auf Staus aufmerksam macht, die auf der geplanten Route liegen. Der Nutzer soll entscheiden können, ob der Stau umfahren und die Route entsprechend neu berechnet werden soll. Diese Story hat Priorität 3.

Einparkhilfe

Wird das Auto eingeparkt, so sollen Anzeigen auf Hindernisse vor und hinter dem Fahrzeug aufmerksam machen, der jeweilige Abstand sollte angezeigt werden. Wird der Abstand gefährlich gering, so sollte dies entsprechend angezeigt werden. Diese Story hat Priorität 5.

Unterhaltungsmedien

Es soll der Zugriff auf die verfügbaren Unterhaltungsmedien im Fahrzeug unter einer einheitlichen Oberfläche ermöglicht werden. Zunächst sollte es möglich sein, Radio und CD zu hören. Weitere Medien sollten später realisiert werden. Diese Story hat Priorität 6.

Checkliste

Vor dem Beginn einer Fahrt sollen Gurtpflicht, Reifendruck, Scheinwerfer und Türen überprüft werden. Bei Fehlern soll eine Warnmeldung angezeigt werden. Diese Story hat Priorität 8.

25.2.3 Tasks

Aus den Stories wurden von den Entwicklern folgende Tasks erstellt:

1. Klassen für Kontrollleuchten erstellen; hierbei wird als Namenskonvention *.warninglight festgelegt
2. Eventsystem entwickeln, vorerst nur Sensoradapter für Tankanzeigen
3. Sensoradapter mit Test (liefert periodisch Daten) programmieren, Füllstand an Tankanzeige liefern
4. Bild für Tankanzeige malen → tankanzeige.gif; Bild für Warnleuchte malen → tankwarnleuchte.gif
5. tankanzeige.gif in Swingfenster einfügen, auf dem Monitor im Fenster der Fahreransicht anzeigen
6. Tankfüllstand in der Tankanzeige visualisieren lassen
7. tankwarnleuchte.gif in die Fahreransicht einfügen
8. Warnleuchte ein-/ausblenden bei Füllstand 5l
9. alternative Darstellung der Tankanzeige tankanzeige2.gif malen
10. ein Fenster (<800*600) als Vielfaches von 32x32 erstellen, in dem eine Positionierung und Gruppierung der enthaltenen Elemente möglich ist
11. Fahreransicht als Swingkomponente programmieren
12. Komfort- und Standardansicht erstellen
13. Implementierung einer Tankstellensuchmaske mit Sortierkriterien und Tankstellenliste
14. Implementierung einer Parkplatzsuchmaske mit Sortierkriterien
15. Implementierung eines Links zwischen Parkplatz-(bzw. Tankstellenliste) und Routenplaner, der die Integration des ausgewählten Parkplatzes bzw. der ausgewählten Tankstelle in die Route veranlasst

16. Erstellung eines Links zwischen der Tankstellensuchmaske und der Tankwarnleuchte, sodass die Tankstellensuchmaske eingeblendet wird, sobald die Tankwarnleuchte aufleuchtet

Das grundlegende Eventsystem wurde durch einen allgemeinen Nachrichten-Mechanismus realisiert, der im Paket de.ls10.hycop.messaging gebündelt wird. Generell besteht eine Nachricht aus einer Zieladresse und einem Inhalt. Als Zieladressen sind bisher nur Objekte vorgesehen, die über eine update-Methode verfügen, mit deren Hilfe der Inhalt an das Objekt übergeben wird. Die zugehörige Schnittstelle heißt Updatable. Objekte, die Nachrichten beziehen wollen, können sich bei einem Nachrichtenerzeuger als Empfänger anmelden. Die hierfür benötigte subscribe-Methode ist in der MessageGenerator-Schnittstelle deklariert. Als Nachrichtenerzeuger werden hauptsächlich SensorAdapter-Objekte auftreten. Voraussichtlich wird es für jeden Sensor einen eigenen SensorAdapter geben.

Im Paket de.ls10.hycop.simulation wurde ein Simulator für den Tanksensor implementiert, der fortlaufend Messwerte von 50.0 bis 1.0 liefert. Es wurde ein Test mit einem primitiven Anzeigeelement durchgeführt, bei dem das Eventsystem funktionierte.

Den Entwicklern ist aufgefallen, dass es sinnvoll ist, den Drehzahlmesser ständig anzuzeigen. Nach Zustimmung der Kunden wurde die betreffende Storycard dahingehend abgeändert. Diese Aufgabe wurde sofort erledigt.

25.2.4 Funktionale Tests der Kunden

Die Konzentration auf GUI-Elemente in der Entwicklung macht es erforderlich, sich an diese Tatsache angepasste Testlösungen auszudenken. Die Tests (siehe auch Abschnitt 24.4.5) wurden zunächst vollständig von den Entwicklern erstellt. Da jedoch Zweifel aufkamen bezüglich der Überzeugungskraft von Tests, die von Entwicklern für ihr eigenes Produkt gemacht werden, sollen die Tests zukünftig durch die Kunden erstellt werden. Hierfür wurde ihnen eine Schnittstelle zu den einzelnen Cockpitelementen zur Verfügung gestellt.

Es gab noch Schwierigkeiten, den Test-First-Ansatz (siehe auch Abschnitt 24.4.5) umzusetzen. Ein Grund dafür war die Schwierigkeit, die Schnittstelle im Vorraus zu erahnen und dafür Tests zu schreiben. Darum wurden die Tests erst nach Fertigstellung der Klassen erstellt.

Die implementierten Stories Tankanzeige und Tankwarnleuchte wurden durch visuelle Inspektion getestet. Die Tankanzeige und die Warnleuchte funktionierten einwandfrei. Einziger Verbesserungsvorschlag war eine Änderung der Tanknadel zu einem gefüllten Polygon.

25.2.5 Fazit

In der ersten Woche entschieden sich die Kunden bewusst dafür, den Entwicklern nur eine geringe Anzahl an Stories vorzulegen. Dies führte zu einer geringen Auswahlmöglichkeit für die Entwickler, was als unvorteilhaft empfunden wurde. Im weiteren Verlauf soll daher eine größere Anzahl an Stories entwickelt werden.

Da die Entwickler mit den in der ersten Woche gestellten Aufgaben recht schnell fertig wurden, wurden mehr zu bearbeitende Aufgaben von den Kunden erwartet. Das Pair-Programming

(siehe auch Abschnitt 24.4.2) klappte in der ersten Woche noch nicht so gut. Es gab auch Code, der nur von einem Entwickler erzeugt wurde. Darum wurde beschlossen, dass zumindest ein Code-Review gemacht werden sollte. Kleinere Korrekturen am Code dürfen auch alleine vorgenommen werden. Im weiteren Verlauf haben die Entwickler mit dem Pair-Programming positive Erfahrungen gemacht. Die Paare wechselten dynamisch und Fehler wurden schneller entdeckt. Auch wurde die Storyanzahl in der zweiten Iteration von den Entwicklern als angemessen beurteilt.

Die Entwickler realisierten einzelne Tasks recht zügig. Dabei vergaßen sie häufig, die dafür benötigte Zeit festzuhalten. Die bisherigen Zeitangaben sind darüberhinaus kaum vergleichbar, da jeder Entwickler sein ganz persönliches Zeitmaß hat. Einige rechnen in Minuten und andere wiederum in Tagen. Es wird angestrebt, ein einheitliches Zeitmaß zu finden.

Das Ziel des ersten Releases, das in der Fertigstellung einer vorläufigen Standardansicht und Komfortansicht bestand, wurde in dieser Woche erreicht. Die Abnahme des ersten Releases durch die Manager ist positiv verlaufen. Die geforderte Funktionalität (einer Standardansicht und einer Komfortansicht) mit den oben genannten Medienobjekten wurde erreicht und sogar durch nicht geforderte zusätzliche Elemente bereichert (XML-Parser). Die Testumgebung für die funktionalen Tests wurde von den Kunden ebenso fristgerecht fertiggestellt. Die Präsentation der beiden Ansichten und der funktionalen Tests konnte ohne Fehler durchgeführt werden. Damit ist dieses Release abgeschlossen.

25.3 Zweites Release

Hauptziel dieses Releases ist es, ein Tool zu erstellen, das dem Benutzer das Konfigurieren von Cockpitansichten erlaubt (siehe auch Abschnitt 24.4). Darüberhinaus soll ein parallel zum Fahrercockpit laufendes Beifahrercockpit erstellt und der Zugriff auf Medien wie den CD-Player und das Radio über das Cockpit ermöglicht werden.

25.3.1 Stories

Am Dienstag wurden den Entwicklern von den neuen Kunden folgende Anforderungen für das zweite Release vorgelegt:

Generisches Cockpit

Hauptanforderung ist die Erstellung eines generischen Cockpits. Die Ansichten sollen bei abgeschaltetem Motor mit einem geeigneten Tool konfiguriert werden können. Es soll vordefinierte Ansichten geben, die unverändert bleiben. Benutzerdefinierte Ansichten sollen auf der Basis existierender Ansichten erstellt werden können. Folgende Möglichkeiten zur Bearbeitung von Ansichten sollen gegeben sein:

- variable Positionierung von Cockpitelementen

- variable Farbgestaltung von Hintergrund und Cockpitelementen (nach Absprache mit den Entwicklern einigten wir uns auf 3 Farbschemata, die zur Wahl stehen sollen)
- Wahlmöglichkeiten zwischen verschieden gestalteten Anzeige- und Bedienelementen desselben Typs

Dabei ist auf Usability in Form von Pflichtelementen und -farben, Sichtbarkeit, Zweckmäßigkeit, geeigneten Farbkontrasten etc. zu achten. Diese Story hat Priorität 1.

Fahrer- und Beifahrercockpit

Es sollen ein Fahrer- und ein Beifahrercockpit realisiert werden. Beide sollen gleichzeitig anzeigbar und bedienbar sein, sowie getrennt voneinander konfiguriert werden können. Diese Story hat Priorität 2.

Oberfläche für Unterhaltungsmedien

Es soll eine geeignete Oberfläche zum Zugriff auf Unterhaltungsmedien entwickelt werden. Diese Oberfläche soll dem Benutzer zunächst nur den Zugang zu einem CD-Player und zum Radio geben. Dabei sollen die gängigen Funktionen (Play, Stop, Rew, Skip, Tuning,...) zur Verfügung stehen. Diese Story hat Priorität 3.

25.3.2 Usability

Zunächst werden die Cockpitelemente Tachometer, Blinker, Warnblinkanlage, Fernlicht, Abblendlicht, Tankfüllstand, Motortemperaturanzeige und alle Warnleuchten als Pflichtelemente deklariert. Darüber hinaus werden Pflichtfarbenden für die Medienobjekte Blinker (grün), Fernlicht (blau), Warnleuchten, die Elektronik betreffen (gelb) und sonstige Warnleuchten (rot) festgelegt. Die Möglichkeit der Positionierung von Medienobjekten in einer Ansicht soll auf folgende Weise eingeschränkt sein: Medienobjekte dürfen einander nicht überlappen. Der linke Blinker muss links vom rechten Blinker und beide auf der gleichen Höhe liegen. Die Warnleuchten sollten fest positioniert und für die Kontrollleuchten bei Aktivität eine Gruppierung vorgegeben sein. Alle ausgewählten Leuchten einer Gruppe sollen in einen Container eingezeichnet und innerhalb dieses Containers umpositioniert werden können. Das heißt, dass nicht die einzelnen Kontrollleuchten auf dem Cockpit positioniert werden sollen, sondern die Container. Diesem Konzept zufolge gehören beispielsweise die Leuchten für das Parklicht, das Abblendlicht vorne, das Abblendlicht hinten, das Fernlicht, den Rückwärtsgang, den Nebelscheinwerfer und die Nebelschlussleuchte in einen gemeinsamen Container.

25.3.3 Abgelehnte Stories

Es wurden alle Stories angenommen.

25.3.4 Tasks

Aus den Stories wurden folgende Tasks erstellt:

1. Staumeldungen im Nachrichtenfenster anzeigen lassen
2. RMI in das Messagingsystem einbauen
3. Radio Tuner; Sounds, Streams auf Soundkarte ausgeben
4. Erstellung eines Ansichteneditors, der das Positionieren von Cockpitelementen mittels drag and drop unterstützt
5. Anforderungen an Usability definieren
6. Beifahreransicht erstellen
7. Die Entwicklung von Graphiken für die Bedienelemente der Multimediaoberfläche

Zusätzlich wurden Tasks zu der im ersten Release zurückgestellten Story „Routenplaner“ erstellt:

1. Erstellung eines Eingabedialogs für den Routenplaner
2. Zeichnen und Einbinden von Knöpfen für das Zoomen und Navigieren auf der vom Routenplaner gezeigten Landkarte

Zudem wird eine Task in Angriff genommen, die durch keine von den Kunden gestellte Anforderung motiviert ist, die jedoch für die Simulation eines echten Routenplaners als unerläßlich erscheint. Dabei handelt es sich um die Entwicklung einer Blackbox für den Routenplaner. Sie soll ein GPS-System simulieren, welches zur Berechnung der Route dient.

25.3.5 Funktionale Tests

Um für die funktionalen Tests auch komplexe Funktionsabläufe wiederholt simulieren zu können wurde hierfür eigens eine Anwendung entwickelt. Mit Hilfe dieser Anwendung wurden einige Fehler in der Implementierung aufgedeckt. Die Sensoradaptoren sensoradapter.seatbelt, sensoradapter.doorsnotclosed, sensoradapter.defectlights und sensoradapter.tirepressure für Leuchten und Anzeigeinstrumente, tauchen in keiner Ansicht auf.

Es fehlt eine Warnleuchte für zu niedrigen Öldruck und bei der Öldruckanzeige fehlen die Maßeinheiten. Darüber hinaus funktionerte die Kontrollleuchte failure.asr für die ASR (Anti-Schlupf-Regelung) nicht. Dieser Fehler ist behoben worden.

Die Behandlung ungültiger Werte von Anzeigeinstrumenten ist bisher nicht zufriedenstellend. Insbesondere können sämtliche Kilometerzähler negative Werte und die Uhr Zeiten von -99:99 bis 99:99 anzeigen, was den für eine Uhr üblichen Wertebereich deutlich übersteigt.

25.3.6 Fazit

Die Anwendung ist um RMI erweitert worden. Dadurch besteht die Möglichkeit, die Hardware vom Cockpit zu trennen, was unsere Simulation der Realitität ein Stück näher bringt. Außerdem erlaubt es, nun mehrere Cockpits parallel laufen zu lassen.

Bei den Arbeiten am Ansichteneditor wurde schnell deutlich, dass es sich hierbei, um eine Aufgabe von großem Umfang handelt. Es treten Schwierigkeiten mit dem Flowlayout der Instrumentenauswahlleiste auf, die durch Implementierung eines neuen Layouts gelöst werden. Die *drag&drop*-Funktionalität des Editors, sowie das Laden und Speichern von Ansichten wird entwickelt. Auch hierbei treten Probleme auf, die mit der gegebenen Java-API zusammenhängen und nicht gut nachvollziehbar sind: Wenn ein Instrument verschoben wird, führt das anschließende Einfügen in den umliegenden Container dazu, dass alle Einträge der Matrix, die auf dieses Instrument verweisen, gelöscht, also die entsprechenden Zeiger auf null gesetzt werden. Das Problem läßt sich umgehen, indem das Einfügen in den Container vor dem Aktualisieren der Matrix vorgenommen wird.

In der PG-Sitzung ist eine neue Konzeption des Ansichteneditors festgelegt worden. Zur Gruppierung der Instrumente und Anzeigen wird festgelegt, dass die Hersteller eine feste Aufteilung in Bereiche mit zulässigen Medienobjekten vorgeben, in denen der Nutzer später einzelne Instrumente anordnen kann. Die Positionen kritischer Warnleuchten werden fest vorgegeben. Es werden also zwei Editoren benötigt, je einer für die Nutzer und die Hersteller. Es soll aber zunächst nur der Editor für die Nutzer realisiert werden.

Die Abnahme des zweiten Releases war erfolgreich.

25.4 Drittes Release

Das wichtigste Ziel dieses dritten Releases ist die Realisierung von automatischen Ansichtenwechseln, gesteuert durch GPS-Koordinaten. Die Entwicklungsarbeiten werden ab dem 06.01.2003 fortgesetzt.

25.4.1 Stories

Die neuen Kunden legten den Entwicklern folgende Anforderungen für das dritte Release vor:

Ausstellungsansicht

Es soll eine Ausstellungsansicht zur Demonstration der Funktionen des Cockpits anhand der Storysimulation erstellt werden. Ein wichtiger Bestandteil der Storysimulation soll sein, dass dem Betrachter Informationen zur Erklärung der Simulationsabläufe bzw. der Cockpitfunktionen angezeigt werden, wobei natürlich nichts Selbsterklärendes zu kommentieren ist. Diese Story hat Priorität 4.

Onlinehilfe

Es soll eine Onlinehilfe implementiert werden. Es soll die Möglichkeit zum Wechsel in einen Hilfemodus geben, in dem auf Wunsch detaillierte Informationen zur Cockpitbenutzung bzw. zur Funktion der einzelnen Anzeigen geliefert werden. Vorschlag der Kunden ist die Realisierung durch Tooltips. Diese Story hat Priorität 3.

Werkstattansicht

Es soll eine Werkstattansicht entwickelt werden, welche neben einer erweiterten Checkliste, die detaillierter und visuell aussagekräftiger ausfallen sollte als bisher, auch Reparatur- und Inspektionsdaten anzeigt. Sie soll nur manuell erreichbar sein und die Möglichkeit zum Rufen eines Pannendienstes bzw. zum Suchen einer Werkstatt geben. Die Medienobjekte Geschwindigkeitsanzeige, Drehzahlmesser, Routenplaner, Abstandswarner, Unterhaltungsmedien, Tages- und Reisekilometerzähler sowie Innen- und Außentemperaturanzeigen müssen nicht Bestandteil dieser Ansicht sein. Die Werkstattansicht soll nur Herstellerkonfigurierbar sein. Diese Story hat Priorität 2.

Automatischer Ansichtenwechsel

Es soll die Möglichkeit zum Wechsel in einen Modus geben, in dem, gesteuert durch Auswertung von GPS-Koordinaten, automatische Ansichtenwechsel erfolgen. Beim Ansichtenwechsel müssen die aktuellen Daten erhalten bleiben. Diese Story erhält die Priorität 1.

Installationsprogramm und Benutzerhandbuch

Es sollen ein Installationsprogramm und ein Benutzerhandbuch erstellt werden. Diese Story hat Priorität 1.

25.4.2 Abgelehnte Stories

Folgende Stories wurden von den Entwicklern zurückgewiesen:

Herstellereditor

Es soll ein Ansichteneditor speziell für Autohersteller erstellt werden. Er soll die Möglichkeit der Festlegung von Bereichen in Ansichten geben, sodass für jeden Bereich eine Auswahl von darin platzierbaren Medienobjekten bestimmt werden kann. Diese Story hat Priorität 7.

25.4.3 Tasks

1. Schreiben eines Benutzerhandbuches

2. Fertigstellung eines Installationsprogramms
3. Entwicklung eines Buttons, mit dessen Hilfe der automatische Ansichtenwechsel während der Fahrt an- und abschaltbar ist
4. Erstellung eines Sensoradapters für das An- und Abschalten des automatischen Ansichtenwechsels (analog zum Rechtsklick-Menü)
5. Implementierung entsprechender Medienobjekte zu Anzeige, Eingabe und Speicherung von Inspektions- und Reparaturdaten
6. Bilder für die Werkstattansicht malen
7. Die verschiedenen Ansichtstypen auf read-only schaltbar und die Werkstattansicht nur manuell aufrufbar machen
8. Ertellung einer Pannendienstliste und eines Buttons zum Rufen eines Pannendienstes
9. Entfernen der Größenangaben der einzelnen Instrumente aus den XML-Views
10. Werkstattansichtenfenster erstellen
11. Entwicklung einer ausführlicheren Checkliste für die Werkstattansicht mitsamt eines Trigger-Buttons zum Auslösen eines Durchlaufes derselben
12. XML-Dateien um Tooltips als Onlinehilfe erweitern und diese evtl. auslagern
13. Tooltips mittels Klick auf einen Button ein- und ausschaltbar zu machen
14. Exceptions im Staumelder abfangen und im Nachrichtenfenster ausgeben
15. GPS-Daten für die Simulation sammeln
16. Implementierung einer XML- (Zeitpunkt- und Ort-) basierten Simulation der zu Beginn des ersten Semesters entworfenen Urlaubsfahrt mit HyCop

25.4.4 Funktionale Tests

Die funktionalen Tests wurden um die zu den Ansichten hinzugekommenen Medienobjekte erweitert. Als umfangreiche Testfolge wurde die Storysimulation realisiert, die komplexe Zusammenhänge verschiedenster Cockpitelemente benötigt und so einen weitreichenden funktionalen Test darstellt.

25.4.5 Fazit

Kleinere, im Rahmen der Präsentation des zweiten Releases entdeckte Fehler wurden zu Beginn des dritten Releases zunächst behoben.

Ursprünglich war geplant, das Cockpit so zu implementieren, dass zuletzt die im ersten Semester entworfene Story simuliert werden kann. Einige dafür notwendige Elemente wurden jedoch aus zeitlichen Gründen nicht realisiert. Ihre Ergänzung stellt allerdings keine große

Herausforderung dar, da ähnliche Funktionen bzw. Medienobjekte bereits an anderen Stellen im Cockpit existieren. Es wurde beispielsweise keine Hotelsuchmaske implementiert. Dafür gibt es jedoch eine Parkplatzsuchmaske mit analoger Funktionalität.

Der Vorsatz, bei diesem Release nicht wieder in letzter Minute Änderungen vorzunehmen, wurde leider nicht eingehalten, so dass auch bei diesem Release die letzte Stunde einige Überraschungen brachte. Es kam jedoch nicht zu einer Katastrophe. Das dritte Release wurde zwei Wochen früher fertiggestellt als ursprünglich geplant und am 21.01.2003 erfolgreich präsentiert.

25.5 Übergreifendes Fazit

Insgesamt ist die Protokollierung der geschätzten und tatsächlichen Zeiten nur sehr lückenhaft geschehen, so dass diese Angaben im Tagebuch leider fehlen. Die zeitliche Einschätzung der Stories war jedoch so gut, dass die Releases immer termingerecht fertiggestellt wurden.

Es wurde zunächst nicht sehr ordentlich mit den Taskcards umgegangen. Zumeist lagen sie neben den Rechnern, an denen zuletzt gearbeitet wurde. Es gab keinen definierten Stapel fertiggestellter Tasks. Im PG-Raum wurden daher zwei Ablagen für die Stories und die Tasks vorbereitet. In die orangene Ablage kamen die nicht abgearbeiteten Storycards und Taskcards, in die grüne Ablage die fertigen Storycards und Taskcards.

Es hat sich als sehr hilfreich erwiesen, dass einer der Kunden aus dem zweiten Release auch im dritten Release Kunde war. So konnte die Kundenarbeit von Anfang an besser erledigt werden.

Teil VI

Dokumentation

Kapitel 26

Benutzung von HyCop

Autoren: *Stefan Borggraefe*
Tobias Wolf

In diesem Abschnitt wird die Benutzung von HyCop durch den Endbenutzer erläutert. HyCop besteht aus zwei Teilen: dem eigentlichen Cockpit mit seinen verschiedenen Ansichten (siehe Abschnitt 26.2) und dem Ansichteneditor, in dem die Ansichten verändert werden können (in Abschnitt 26.3 beschrieben). Zunächst wird jedoch die Installation von HyCop erläutert.

26.1 Installation

HyCop benötigt eine Java Runtime Environment der Version 1.4 oder höher.

HyCop wird als eine .tar.gz-Datei verteilt. Diese Datei ist zu entpacken, so dass man ein Verzeichnis erhält, dass unter anderem die Dateien cockpit.bat, cockpit.sh, editor.bat und editor.sh enthält. Durch das Ausführen dieser Dateien kann das Cockpit bzw. der Ansichteneditor gestartet werden. Unter Windows sind dabei die *.bat, unter allen anderen Plattformen die *.sh-Dateien zu verwenden.

26.2 Das Cockpit

Nach dem Starten des Cockpits sieht man eine Reihe von Instrumenten, die zusammen die Standansicht bilden. Es gibt reine Anzeigeinstrumente, so z.B. Tachometer und Zuladungsanzeige, aber auch Bedienelemente, mit denen man interagieren kann. Beispielsweise lassen sich die Blinker durch einen Klick an- und abschalten. Verweilt man mit der Maus auf einem Instrument, erscheint eine Erklärung zu diesem Instrument. Diese Hilfe kann auch durch einen Klick auf den Knopf mit dem Fragezeichen in der Sprechblase ab- und wieder angeschaltet werden.

Im Folgenden werden das Popup-Menü, mit dem der Benutzer zwischen den verschiedenen Ansichten umschalten sowie Simulationen starten kann, und das Nachrichtenfenster, das alle

Meldungen an den Benutzer aufnimmt, näher beschrieben.

26.2.1 Das Popup-Menü

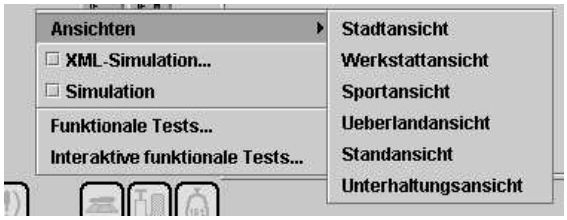

Abbildung 26.1: Popup-Menü des Cockpits

Über die rechte Maustaste ist das auf Abbildung 26.1 zu sehende Popup-Menü erreichbar, über das in andere Ansichten gewechselt werden kann. Jede Ansicht stellt für bestimmte Situationen die passenden Instrumente dar. Es stehen folgende Ansichten zur Verfügung:

- Standansicht (Abbildung 26.3
- Stadtansicht (Abbildung 26.7)
- Sportansicht (Abbildung 26.2)
- Unterhaltungsansicht (Abbildung 26.4)
- Überlandansicht (Abbildung 26.5)
- Werkstattansicht (Abbildung 26.6)

Diese Ansichten werden auch während der Fahrt zu passenden Zeitpunkten automatisch angezeigt. Diese automatischen Wechsel kann man auch unterbinden, indem man den Knopf „Ansichtenwechselschalter“ drückt. Verlassen wir beispielsweise die Stadt, so wird von der Stadt- in die Überlandansicht gewechselt. Weiterhin wird dafür gesorgt, dass die Unterhaltungsansicht während der Fahrt für den Fahrer nicht verfügbar ist, da diese Ansicht für das Fahren unverzichtbare Instrumente nicht enthält.

Da es HyCop an echter Fahrzeug-Hardware mangelt, ist nur eine simulierte Fahrt möglich. Über das Popup-Menü sind zwei Arten von Simulationen erreichbar, über XML-Dateien definierte (Menüpunkt „XML-Simulation...“) und eine festverdrahtete Simulation (Menüpunkt „Simulation“). Die festverdrahtete Simulation startet direkt nach Anwahl des Menüpunkts und läuft so lange, bis der (natürlich ebenfalls simulierte) Tank leer ist; der Durchlauf dauert einige Minuten. Bei der XML-Simulation muss vor dem Start eine entsprechende XML-Datei geladen werden. Es werden bereits zwei XML-Simulationen mitgeliefert, eine kürzere und eine ausführlichere, die die in Kapitel 13 beschriebene Story wiedergibt. Die Simulationen können durch nochmalige Anwahl des entsprechenden Menüpunktes wieder angehalten werden.

Über die Menüpunkte „Interaktive funktionale Tests...“ und „Funktionale Tests...“ können die einzelnen Instrumente von HyCop mit Werten versorgt und so getestet werden.

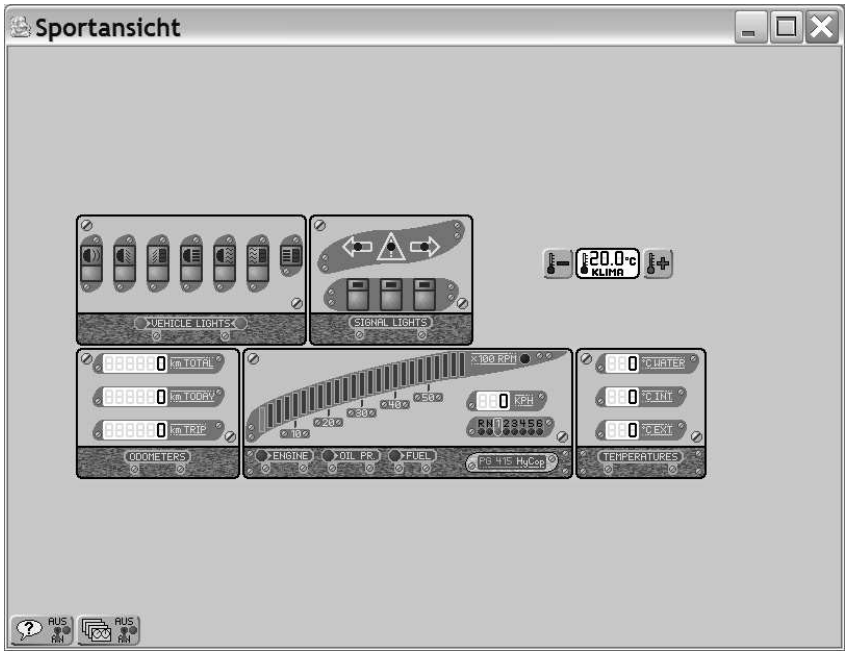

Abbildung 26.2: Sportansicht

26.2.2 Das Nachrichtenfenster

Alle wichtigen Informationen für den Fahrer, wie z.B. Stau- und Diagnosemeldungen, werden dem Fahrer im Nachrichtenfenster präsentiert. Auf Abbildung 26.7 ist es rechts unten im Cockpit zu sehen.

Im Laufe einer Fahrt können im Bereich des Nachrichtenfensters auch noch weitere Informationen auftauchen:

- Die Tankstellenliste erscheint, wenn der Tankfüllstand einen kritischen Wert erreicht hat oder wenn auf die Tankanzeige geklickt wird. In der Tankstellenliste sind alle Tankstellen der Umgebung aufgelistet.
- Klickt man auf einen Eintrag in der Tankstellenliste, erscheint eine detaillierte Auskunft über die jeweilige Tankstelle.
- Im Falle einer Panne erscheint im Bereich des Nachrichtenfensters eine Auflistung der verfügbaren Pannendienste. Der Benutzer kann dann den Pannendienst seiner Wahl rufen lassen, indem er den entsprechenden Eintrag anklickt.

26.3 Der Ansichteneditor

Über den Ansichteneditor (siehe Abbildung 26.8 können alle Ansichten verändert werden. Das Fenster ist in zwei Bereiche aufgeteilt. Im linken Bereich befindet sich eine Palette mit

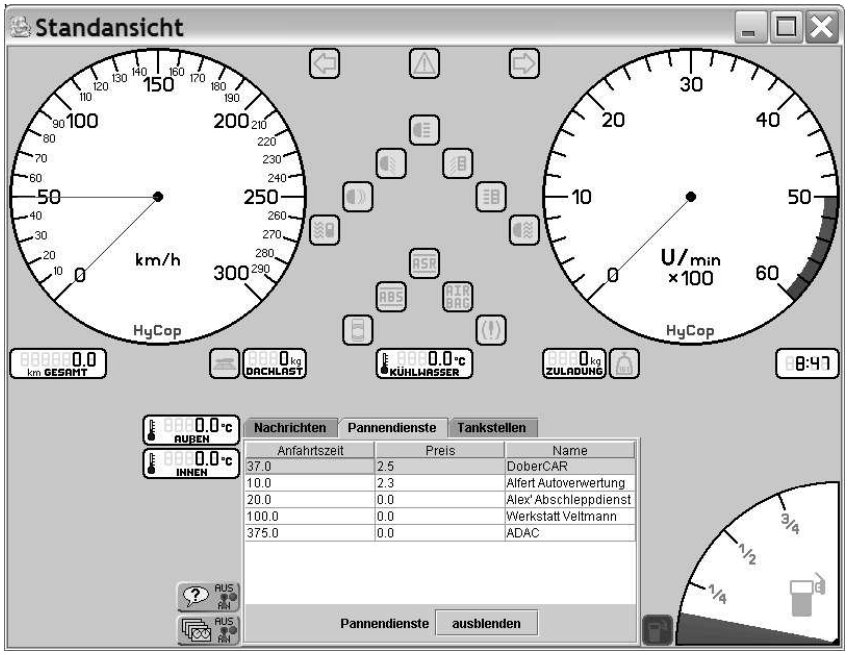

Abbildung 26.3: Standansicht

Instrumenten, die in der derzeit bearbeiteten Ansicht noch nicht verwendet wurden. Im rechten Teil sieht man immer die zur Zeit ausgewählte Ansicht.

Die Bedienung geschieht zum größten Teil per Drag & Drop. Alle Instrumente können mit der Maus hin- und hergeschoben werden, sowohl innerhalb einer Ansicht als auch von der Instrumentenpalette in die Ansicht und zurück. Hat man ein Instrument in die Ansicht geschoben, verschwindet es aus der Intrumentenpalette. Die Konsequenz daraus ist, das jedes Instrument nur maximal einmal in einer Ansicht vorhanden sein kann.

Es gibt Instrumente, die in einer Ansicht verpflichtend vorhanden sein müssen. Diese werden mit einer roten Umrandung dargestellt. Ist ein solches Instrument nicht in der Ansicht vorhanden, kann sie nicht gespeichert werden. In diesem Fall bekommt man eine Meldung, die das Fehlen des entsprechenden Instruments moniert. Über den Menüpunkt „Bearbeiten|Fehlende Instrumente“ kann auch jederzeit während der Bearbeitung einer Ansicht eine Auflistung der fehlenden Instrumente aufgerufen werden.

Weiterhin gibt es Instrumente, die sich innerhalb des Cockpits nur an bestimmten Positionen befinden dürfen. Wenn man ein Instrument in einen unerlaubten Bereich mit der Maus verschieben möchte, wird durch einen roten Rahmen um das Instrument gekennzeichnet, dass diese Aktion nicht erlaubt ist.

26.3.1 Die Menüs des Ansichteneditors

Der Ansichteneditor enthält zwei Menüs, die im Folgenden kurz beschrieben werden. Im Menü „Datei“, vergleiche Abbildung 26.9, kann unter „Ansicht bearbeiten“ die zu bearbeitende

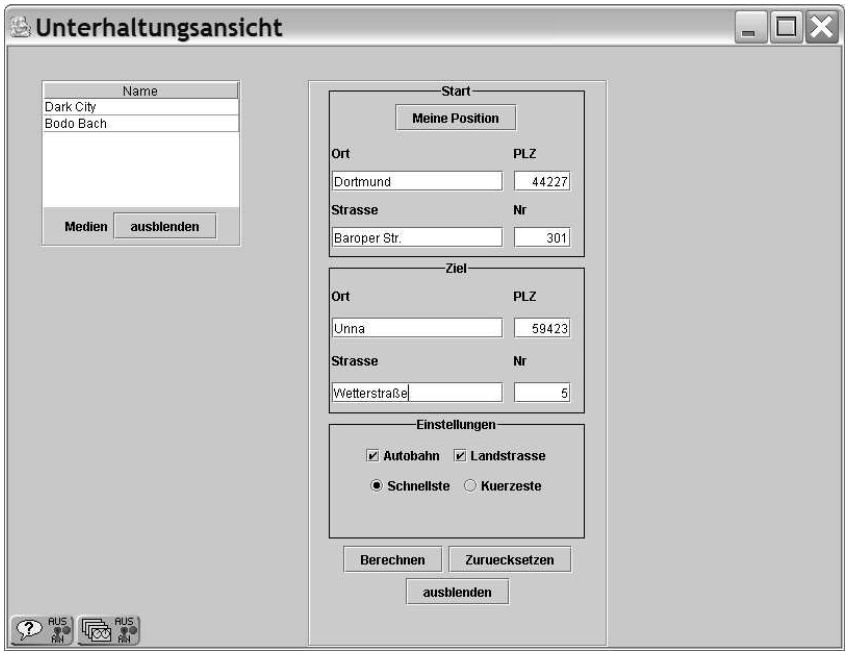

Abbildung 26.4: Unterhaltungsansicht

Ansicht gewählt werden. Mittels des Menüpunkts „Speichern“ kann die aktuelle Ansicht gespeichert werden. Über „Beenden“ wird der Ansichteneditor geschlossen.

Das Menü „Bearbeiten“, auf Abbildung 26.10 zu sehen, enthält einen Menüpunkt „Hintergrundfarbe ändern...“, mit dem die aktuell bearbeitete Ansicht mit einer auszuwählenden Farbe hinterlegt werden kann. Durch „Dimension ändern...“ kann die Größe der Ansicht geändert werden. In dem durch Anwahl dieses Menüpunktes dargestellten Fenster können durch das Verschieben von zwei Reglern Breite und Höhe eingestellt werden. Dabei ist zu beachten, dass das Cockpit nur dann verkleinert werden kann, wenn in dem Bereich, der durch diese Aktion abgeschnitten wird, kein Instrument liegt. Weiterhin ist die Größe des Cockpits in der gegebenen Voreinstellung auf maximal 25×18 Blöcke beschränkt. Beschränkungen der Mindestgröße ergeben sich implizit durch die Forderung nach bestimmten Instrumenten.

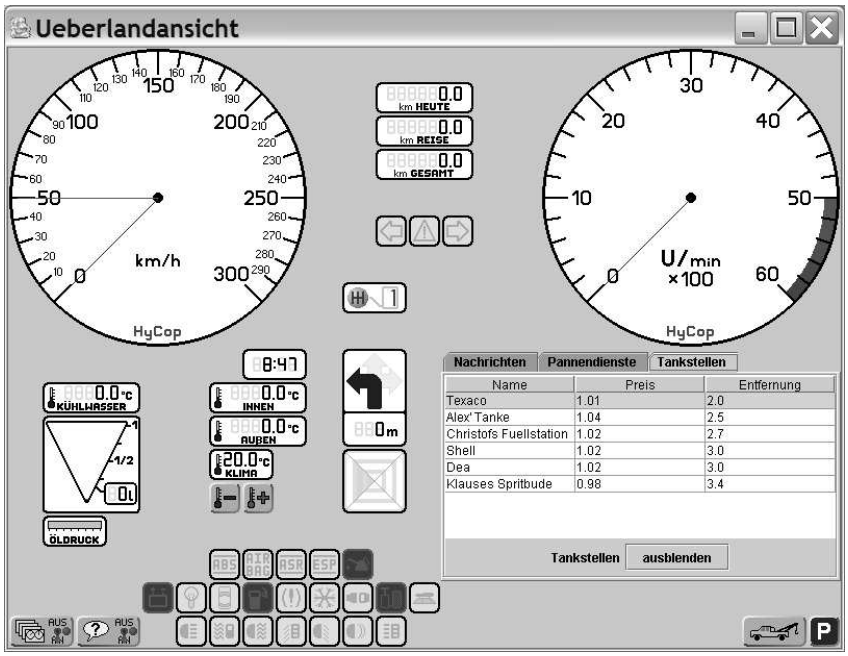

Abbildung 26.5: Überlandansicht

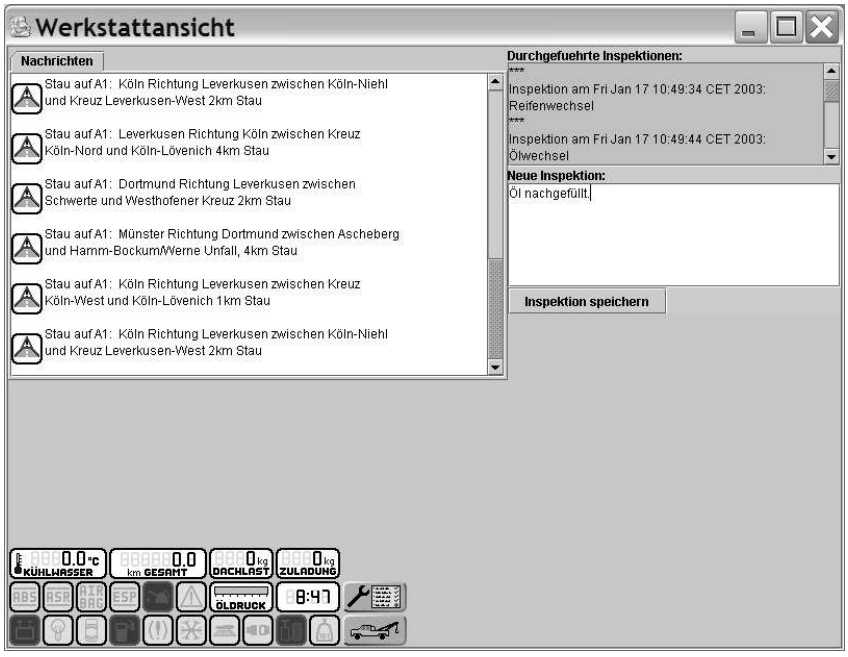

Abbildung 26.6: Werkstattansicht

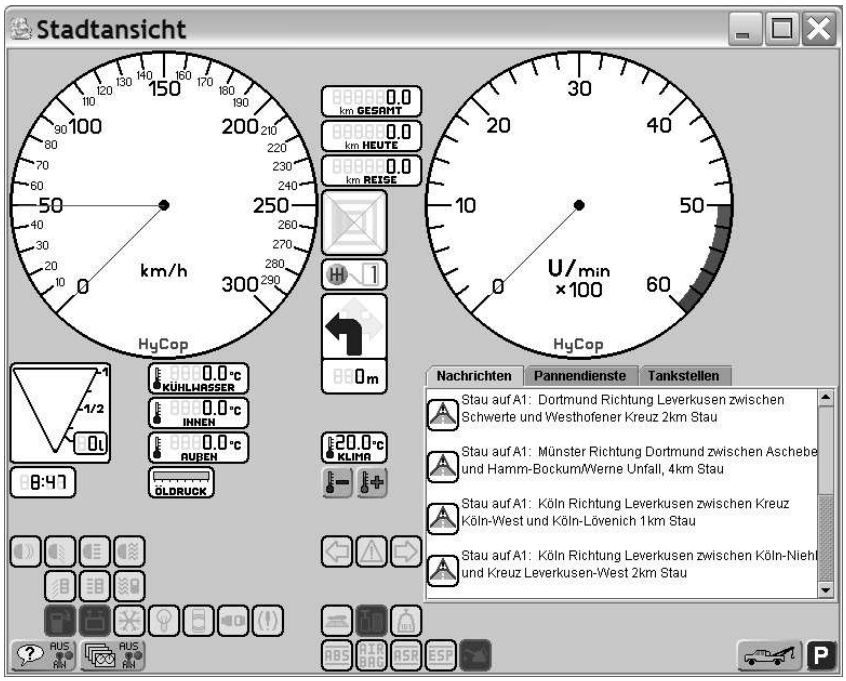

Abbildung 26.7: Nachrichtenfenster

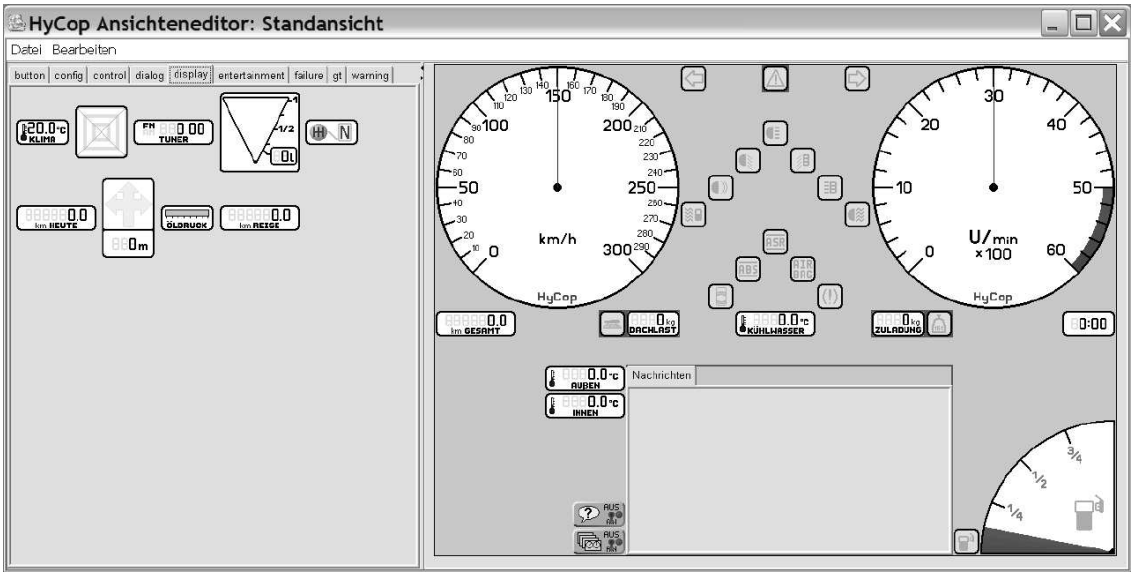

Abbildung 26.8: Der Ansichteneditor

Abbildung 26.9: Menü Datei des Ansichteneditors

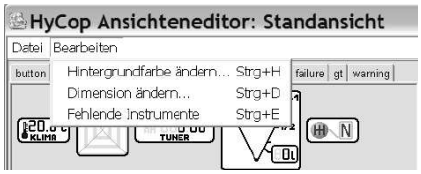

Abbildung 26.10: Menü Bearbeiten des Ansichteneditors

Kapitel 27

Die Struktur der HyCop-Software

Autoren: *Stefan Borggraefe*
Bastian Krol
Daniel Mölle
Oliver Szymanski

Dieses Kapitel beinhaltet eine Beschreibung des konzeptionellen und strukturellen Aufbaus der HyCop-Software. Dabei soll nicht nur auf die Paket- und Klassenstruktur, sondern auch auf die konkrete Arbeitsweise der Systemkomponenten eingegangen werden; dies umfasst eine Erläuterung der Aufgabenverteilung, der bei der Ausführung des HyCop-Programms stattfindenden Abläufe und der Konfiguration des Cockpits.

27.1 Zentrale Konzepte der Implementierung

Zentrale Klassen und Schnittstellen der Implementierung sind die Klassen, die die Laufzeitumgebung (Klasse RuntimeEnvironment), Umgebungen (Klasse Environment) und Ansichten (Klasse View) kapseln. Die Zusammenhänge zwischen diesen Klassen sind in Abbildung 27.1 dargestellt.

Die Laufzeitumgebung ist der Kern der HyCop-Software. Beim Starten der Laufzeitumgebung werden sämtliche für das System relevanten Konfigurationsdateien eingelesen, die allesamt im XML-Format vorliegen. Anhand dieser Konfigurationsdateien werden die Objekte, aus denen sich das Cockpit zusammensetzt, erzeugt. Zu nennen sind hierbei die einzelnen Medienobjekte (Schnittstelle MediaObject), die Sensoradapter (Schnittstelle SensorAdapter) und die Objekte, die die Verknüpfungsstruktur bilden (Klasse Link).

Eine Umgebung beschreibt die Eigenschaften eines Sitzplatzes, insbesondere die Größe des Displays und die Menge der erlaubten Ansichten. Eine Ansicht beschreibt eine Menge von Medienobjekten und ihre Anordnung. Medienobjekte sind dabei alle Anzeigen und Bedienelemente.

Die Sensoradapter stellen die Schnittstelle des Cockpits zu den Sensoren der Fahrzeugelektronik dar, sie können von einem (realen oder simulierten) Sensor mit Daten versorgt wer-

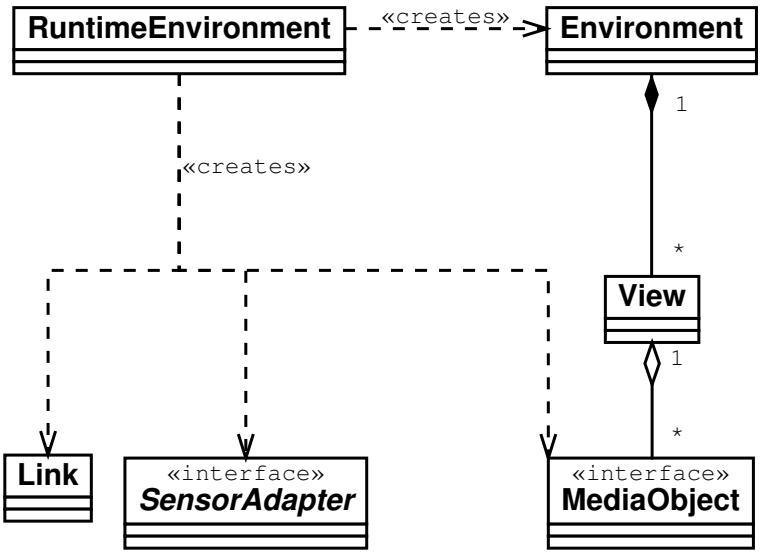

Abbildung 27.1: Umgebungen und Ansichten

den. Wenn ein Medienobjekt Daten aus der Außenwelt empfangen können soll, muss es die Schnittstelle Updatable (vgl. Abbildung 27.2) implementieren. Sie können sich dann bei dem entsprechenden Exemplar der Klasse SensorAdapter durch dessen subscribe-Methode anmelden. Die Methode getData der SensorAdapter dient dazu, die aktuellen Daten, die von einem Sensor geliefert werden, zu erfragen. Dies wird im Normalfall innerhalb der Implementierung der update-Methode eines Medienobjekts passieren.

Auch SensorAdapter implementiert die Schnittstelle Updatable. Die Implementierung der Methode update wird dafür verwendet, neue Daten von Sensoren zu empfangen und sie an alle angemeldeten Medienobjekte zu propagieren.

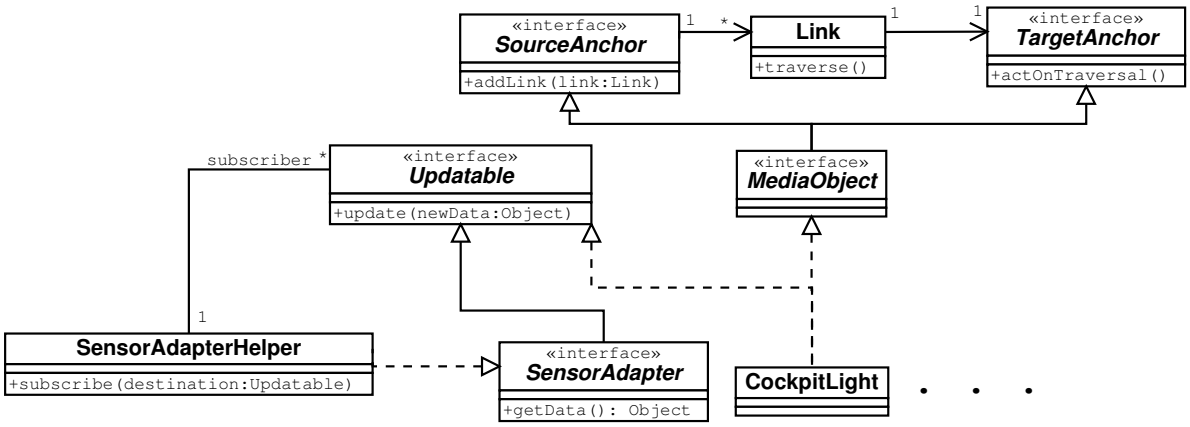

Abbildung 27.2: Sensor- und Linkkonzept

Jeder Link hat genau ein Zielobjekt (Schnittstelle TargetAnchor) und jedes Quellobjekt (Schnitt-

stelle SourceAnchor) kann beliebig viele Links zu anderen Medienobjekten besitzen. Diese Struktur ist in Abbildung 27.2 zu erkennen. Weiterhin implementieren alle Medienobjekte diese beiden Schnittstellen, so dass sie sowohl Ziel als auch Quelle eines Links sein können. Als Beispiel ist in der Abbildung die Klasse CockpitLight dargestellt. Das Linkkonzept wird ausführlicher im Abschnitt 27.6.2 besprochen.

27.2 Überblick über die Struktur der Software

Der grundsätzliche Aufbau und die Interaktion innerhalb des HyCop-Systems ist in Abbildung 27.3 dargestellt. Wie bereits in Abschnitt 27.1 beschrieben, liest die Laufzeitumgebung beim Start des Cockpits die Konfigurationsdateien ein. Daraufhin wird das Cockpit initialisiert und die benötigten Ansichten zur Verfügung gestellt.

Die hierfür benötigten Objekte (z.B. Medienobjekte und Sensoren) werden per *Reflection* erzeugt. Daher ist eine zentraler Mechansimus erforderlich, der bei Bedarf eine Referenz auf ein solches Objekt liefert. Diese Aufgabe übernimmt das *Singleton* ComponentMap, das zentral alle Medienobjekte und Sensoren verwaltet und anderen Klassen zugänglich macht.

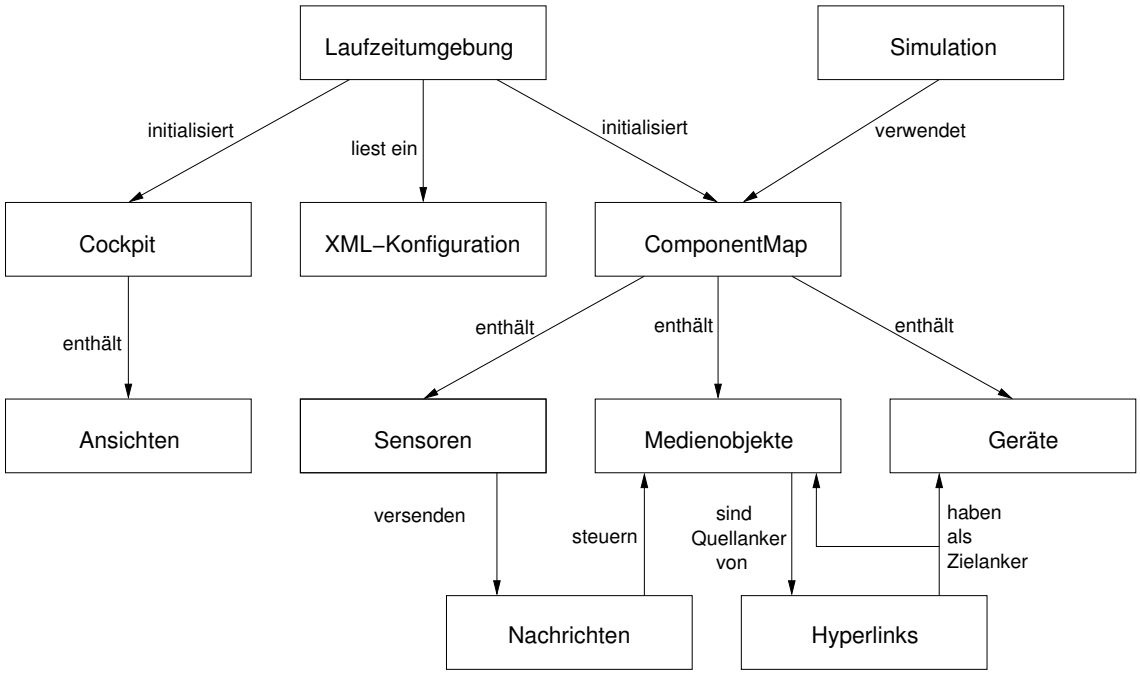

Abbildung 27.3: Übersicht über das HyCop-System

Zusätzlich zu den in Abschnitt 27.1 beschriebenen Sensoren existieren noch Geräte. Damit sind alle technischen Steuergeräte, wie z.B. das Navigationssystem oder die Lichtanlage, gemeint. Auch die Geräte sind über die ComponentMap zugänglich.

Regen Gebrauch von der ComponentMap macht die Simulation, die Daten an die Sensoren schickt, auf die das Cockpit dann entsprechend reagiert. So kann z.B. im Rahmen der Si-

mulation die Zunahme der Außentemperatur gemeldet werden. Stellt ein Cockpit gerade ein Medienobjekt dar, das die Außentemperatur visualisiert, wird die Darstellung dementsprechend aktualisiert.

Damit diese Benachrichtigung von Medienobjekten funktionieren kann, werden Nachrichten von Sensoren an Medienobjekte verschickt. Weiterhin kann der Benutzer durch die Interaktion mit Medienobjekten die Traversierung von Hyperlinks auslösen. Das kann sich dann auf andere Medienobjekte oder auf Geräte auswirken.

27.3 Komponenten bei HyCop

Der Kern der HyCop-Software ist die Laufzeitumgebung, über die sowohl das Cockpit als auch die Kommunikationsschnittstellen der Fahrzeug-Hardware gestartet werden können. Für beide ist der Komponentenbegriff von zentraler Bedeutung. Die Komponenten des Cockpits sind die verschiedenen Medienobjekte, also Instrumente oder Dialoge, während die Komponenten der Hardwareschnittstelle Sensoradapter oder Adapter zu physikalischen Geräten sind, zum Beispiel Temperaturfühler bzw. Steuergeräte für die Fahrzeugbeleuchtung.

Komponenten werden nicht etwa von der Laufzeitumgebung vorgegeben, sondern anhand der Konfigurationsdateien mediabobjects.xml, cockpit.xml und hardware.xml erzeugt und parametrisiert (siehe Abschnitt 28.1). Das Gleiche gilt für etwaige Daten- und Kontrollflüsse zwischen den Komponenten. Diese Konzeption hat zwei wesentliche Konsequenzen. Einerseits werden praktisch keine Annahmen über das Cockpit und die Fahrzeughardware gemacht, so dass ein sehr hoher Grad an Flexibilität besteht. HyCop ist deshalb nicht auf den Einsatz als Fahrzeugcockpit beschränkt – es könnte mit einer geeigneten Konfiguration genauso gut zur Überwachung beliebiger technischer Anlagen, als Wetterstation oder als Assistenzsystem bei medizinischen Operationen eingesetzt werden. Hierzu müssten die grafischen Repräsentationen der benötigten Instrumente und die Verknüpfungsstruktur angepasst werden. Dies ist mit Hilfe der Konfigurationsdateien möglich. Andererseits führt der Verzicht auf einschränkende Annahmen dazu, dass die Komponenten keinen festgelegten Verwendungszweck haben, solange keine Ankopplung an echte Sensoren und Geräte oder eine entsprechende Simulation erfolgt. Bei einem praktischen Einsatz von HyCop als Fahrzeugcockpit müsste deshalb vor allem die Hardware-Konfiguration vom Hersteller vorgenommen und vor den Anwendern verborgen werden.

27.4 Übersicht über die Paketstruktur

Ausgangspunkt der folgenden Erörterungen ist die Pakethierarchie inner- und unterhalb von de.ls10.hycop, in der alle von der Projektgruppe entwickelten Klassen liegen. Der Aufbau der Paketstruktur ist der übliche: Während die zentralen Klassen, z.B. das Hauptprogramm, und die wichtigsten Schnittstellen in diesem Hauptpaket untergebracht sind, gibt es für spezielle Aufgabenbereiche verschiedene Unterpakete, wie es der Grundsatz des *separation of concerns* nahelegt. Viele dieser Unterpakete kapseln HyCop-spezifische Mechanismen und Werkzeuge, die von der Laufzeitumgebung des Cockpits genutzt werden. Ihre Aufzählung erfolgt in Tabelle 27.1. Die übrigen Unterpakete werden in Tabelle 27.2 aufgelistet.

Unterpaket	*Aufgabenbereich*
checklist	Speicherung und Durchlauf von Checklisten
gui	Darstellung graphischer Cockpit-Elemente
hardware	Kapselung der Fahrzeug-Hardware
link	Repräsentation von Hyperlinks
messaging	Versand von Nachrichten
simulation	Simulation der Fahrzeugsensorik

Tabelle 27.1: HyCop-spezifische Unterpakete

Unterpaket	*Aufgabenbereich*
dom	Auslesen von XML-Dateien über DOM/SAX
evaluator	Auswertung und Verwaltung von Ausdrücken
functionaltest	Kapselung und Durchführung funktionaler Tests
util	Bereitstellung diverser Hilfswerkzeuge
viewedit	Bearbeitung von Cockpit-Ansichten (Masken)

Tabelle 27.2: Sonstige Unterpakete

27.5 Komponentenverwaltung

Zur Verwaltung der Komponenten, z.B. Sensoradapter, Anzeige- und Bedienelemente, gibt es die Klasse ComponentMap. Sämtliche Komponenten des HyCop-Systems werden von der HyCop-Laufzeitumgebung in der ComponentMap abgelegt, die als *Singleton* vorliegt. Zur Speicherung der Komponenten wird hierbei eine HashMap (java.util.HashMap) verwendet, wobei die grundlegenden Zugriffsoperationen, vor allem also das Einfügen und Abfragen von Komponenten, selbstverständlich zur Verfügung stehen.

Die ComponentMap leistet aber weitaus mehr als nur die Verwaltung von Komponenten. Sie kapselt nämlich den Mechanismus, der eine verteilte Ausführung über Java RMI ermöglicht. Auf diese Weise können mehrere Laufzeitumgebungen auf verschiedenen Systemen gestartet werden, die dann miteinander kommunizieren. Insbesondere erlaubt dies eine Trennung von Hardware bzw. Simulation und Cockpit. Die Vorgehensweise ist dabei die, dass die verschiedenen Objekte der Klasse ComponentMap miteinander verkoppelt werden. Die Kommunikation zwischen diesen Komponenten geschieht über Java RMI. Bei einer Anfrage nach einem Objekt an die ComponentMap prüft diese, ob das angefragte Objekt lokal vorliegt. Wenn dies der Fall ist, wird das Objekt zurückgeliefert. Ansonsten prüft die ComponentMap, ob sich das Objekt in einer der gekoppelten ComponentMaps befindet. Ist dies der Fall, wird ein Proxy-Objekt generiert, welches alle Schnittstellen des nachgefragten Objektes implementiert und bei Methodenaufrufen diese über RMI transparent an das entfernte Objekt delegiert.

Der Vorteil bei dieser Vorgehensweise ist, dass man beliebige Objekte einfach über das Ablegen in der ComponentMap verteilt zugänglich machen kann, ohne sich selbst über die Java RMI-Implementierung eines Remote-Objektes zu kümmern. Ebenso kann man nachgefragte Objekte wie ein lokales benutzen.

27.6 Daten- und Kontrollflüsse

Für die Kommunikation zwischen den HyCop-Komponenten gibt es verschiedene Konzepte, da sich auch die Arten der Kommunikation wesentlich voneinander unterscheiden:

- **Datenflüsse von Sensoradaptern zu Medienobjekten**
  Diese Art der Kommunikation ist im Kontext eines Fahrzeugcockpits die wichtigste und wurde deshalb auch zuerst implementiert. Der Datenfluss ist notwendig, wenn ein Sensoradapter neue Daten empfangen hat und die angemeldeten Medienobjekte darüber informieren möchte. Zum An- und Abmelden eines Medienobjektes am Sensoradapter wird das *Observer*-Entwurfsmuster verwendet, für den Datenfluss zwischen den Komponenten dient das messaging-Paket (siehe Abschnitt 27.6.1).
- **Daten- und Kontrollflüsse von Medienobjekten**
  Diese Art von Kommunikation umfasst sowohl die Ansteuerung von Fahrzeug-Hardware als auch die Manipulation anderer Medienobjekte. Dem hypermedialen Design entsprechend werden hierbei Hyperlinks verwendet, die als eigenständige Objekte existieren. Die zugehörigen Klassen befinden sich im link-Paket (siehe Abschnitt 27.6.2).
- **Anfragen an Sensoradapter**
  Unter bestimmten Umständen ist es notwendig, den zuletzt gesendeten Datenwert eines Sensoradapters zu erfragen; dies gilt zum Beispiel beim Durchgehen einer Checkliste. Auch dafür besitzt die Klasse SensorAdapter entsprechende Methoden.

Bezüglich der Kommunikation sind die verschiedenen Medienobjekte, also alle Instrumente und Dialoge, gleichberechtigt. Sie können Daten über das Messaging-System empfangen und sowohl Quell- als auch Zielanker von Hyperlinks sein. Bei der Hardware-Schnittstelle gilt dies jedoch nicht. Ein Sensoradapter (Klasse SensorAdapter) liefert Daten an Medienobjekte, wird aber ausschließlich durch die Hardware-Umgebung (reale oder simulierte Geräte und Sensoren) mit Daten versorgt. Geräte (Paket hardware) hingegen treten nur als Zielanker von Hyperlinks auf, deren Quellanker Medienobjekte sind. Außerdem ist jedem Gerät ein Sensoradapter zugeordnet, dem das Gerät seine Zustandsänderungen mitteilt.

27.6.1 Das Messaging-System

Das messaging-Paket kapselt Klassen zum Nachrichtenaustausch zwischen Komponenten, erlaubt Nachrichten zu generieren, Empfänger und Sender anzugeben und die Nachricht zu übertragen. Dabei ist für den Benutzer des Paketes transparent, wie diese Nachrichten übertragen werden. Innerhalb der messaging-Paketes wird abhängig von der Zieladresse entschieden, welche Übertragungsart (einfacher Methodenaufruf, Remote Method Invocation, Java Message System, etc.) genutzt wird.

Eine Nachricht wird definiert durch die Schnittstelle Message und besteht aus einem Nachrichteninhalt und einer Empfängeradresse. Außerdem besitzt jede Nachricht eine Methode send, um sie nach dem Setzen des Empfängers und des Inhaltes zu verschicken. Es gibt für verschiedene Übertragungsarten unterschiedliche Implementationen dieser Schnittstelle, die sich vor allem in der Implementierung der send-Methode unterscheiden.

Zum Erzeugen von Nachrichten gibt es die Klasse MessageFactory, welche nach dem *Factory*-Entwurfsmuster Nachrichten erzeugt. Dabei wird den statischen getNewMessage-Methoden die Empfängeradresse und optional der Nachrichteninhalt übergeben und eine neue Nachricht erstellt und zurückgeliefert. Beim Erstellen der neuen Nachricht entscheidet die MessageFactory je nach übergebenem Empfängeradresse, welche Implementation einer Nachricht sie wählt. Wenn es sich bei der Empfängeradresse z.B. um ein Objekt handelt, welches die Schnittstelle Updatable implementiert, die eine Methode update definiert, so wird eine Nachricht als Instanz der Klasse UpdateMethodMessage generiert. Eine Nachricht dieser Klasse ruft beim Ausführen ihrer send-Methode die entsprechende Update-Methode des Empfängers auf.

Die Schnittstelle MessageGenerator definiert die Methoden subscribe, unsubscribe und sendToAllSubscribers, die für das *Observer*-Entwurfsmuster benötigt werden. Dies ermöglicht es leicht, eine Vielzahl angemeldeter Komponenten bei Bedarf zu benachrichtigen. Die Klasse MessageGeneratorHelper enthält eine Standardimplementation der Methoden subscribe und unsubscribe und kann zur leichteren Implementation der Schnittstelle MessageGenerator genutzt werden.

27.6.2 Hyperlinks

Ein wichtiger Aspekt des hypermedialen Cockpits sind die Hyperlinks, die einzelne Medienobjekte miteinander verbinden. Im link-Paket befinden sich die Klassen und Interfaces, die dieses Konzept unterstützen. Links werden als eigenständige Objekte behandelt, und zwar als Instanzen der Klasse Link. Die Objekte, die verlinkt werden sollen, müssen das Interface SourceAnchor bzw. TargetAnchor erfüllen, je nachdem ob sie Quell- oder Zielanker eines Links sein sollen (siehe Abbildung 27.4). Ein Link hat im Wesentlichen drei Attribute, nämlich die Strings event und action, sowie den TargetAnchor target.

Jede Quelle verfügt über eine Liste von Links, zu der weitere Links mit der Methode addLink() hinzugefügt werden können. Das Traversieren eines Links läuft nun folgendermaßen ab (siehe auch Abbildung 27.5): Wenn in der Quelle ein Ereignis auftritt, durchläuft diese ihre Liste mit Links. Jeder Link, dessen event mit dem aufgetretenen Ereignis übereinstimmt, wird mittels eines Aufrufs der Methode traverse() traversiert. Daraufhin ruft der Link auf dem Zielanker die Methode actOnTraversal(action, parameterMap) auf.

Das Interface MediaObject aus dem Paket gui erbt sowohl von SourceAnchor als auch von TargetAnchor, so dass alle Medienobjekte per Definition Quell- und Zielanker eines Links sein können. Medienobjekte können also in den Konfigurationsdateien (siehe Abschnitt 28.1.2) beliebig verknüpft werden.

Allerdings können nicht nur Medienobjekte mit anderen Medienobjekten verlinkt werden, auch in anderen Bereichen hat sich das Linkkonzept als hilfreich erwiesen. So sind zum Beispiel einige Medienobjekte auch mit HardwareDevices (siehe Abschnitt 27.8) verknüpft, wobei die Klasse HardwareDevices nur das Interface TargetAnchor zu implementieren braucht, nicht aber das Interface SourceAnchor, da Devices nie Quellanker eines Links sind.

Weiterhin hat sich herausgestellt, dass es in einigen Fällen Sinn macht, wenn Links parametrisierbar sind. Ein Beispiel dafür ist der Link von der Tankstellenliste auf den Tankstellendetaildialog, der vor dem Traversieren mit der ausgewählten Tankstelle parametrisiert wird.

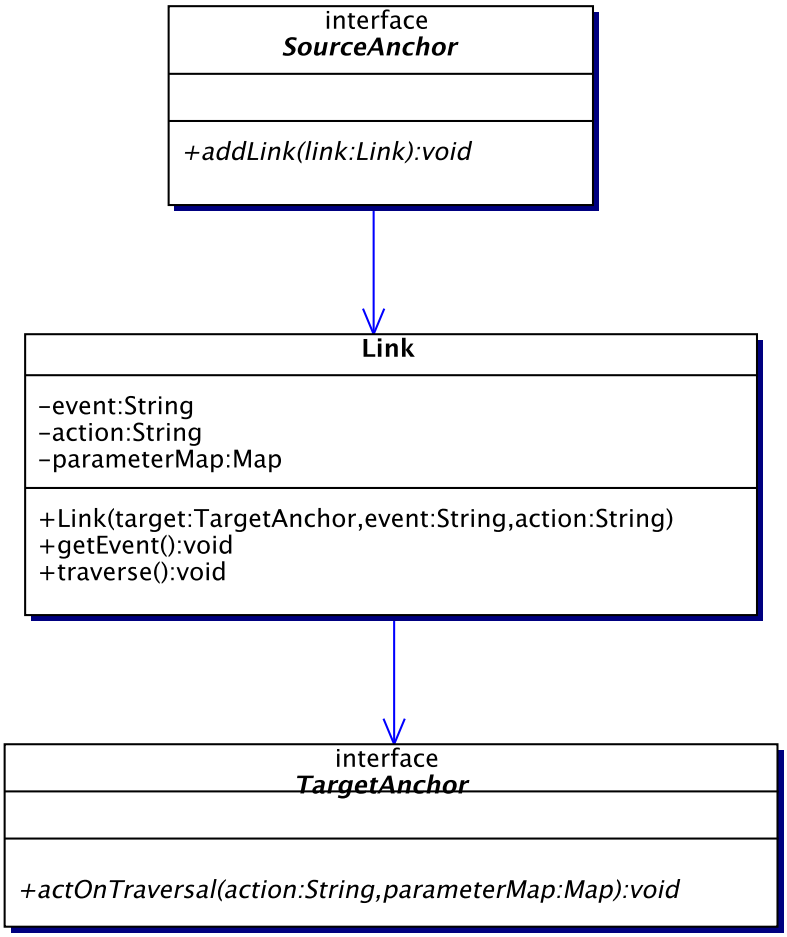

Abbildung 27.4: Klassendiagramm zum Linkkonzept

Diesem Zweck dient das Attribut parameterMap der Klasse Link, welches beim Traversieren an den Zielanker übergeben wird.

27.7 Graphische Oberfläche

Die graphische Oberfläche des Hycop-Cockpit besteht aus Medienobjekten wie der Geschwindigkeitsanzeige, dem Drehzahlmesser, der Tankfüllstandsanzeige oder der Tankwarnleuchte, welche Nachrichten von Sensoradaptern empfangen und untereinander über Links verbunden werden können. Das Paket für Medienobjekte ist de.ls10.hycop.gui und die Schnittstelle, die alle Medienobjekte implementieren, ist MediaObject. Diese Schnittstelle gibt an, dass Medienobjekte sowohl Quelle als auch Ziel von Links sein können. Sie ermöglicht das Hinzufügen eines Listeners für Mausereignisse und das Setzen und Abfragen einer Beschreibung für das Medienobjekt.

Die Klasse MediaObjectGroup ermöglicht es, mehrere Medienobjekte zu einer Gruppe zusammenzufügen. Durch sie kann ein neues Medienobjekt aus anderen Medienobjekten zusammen-

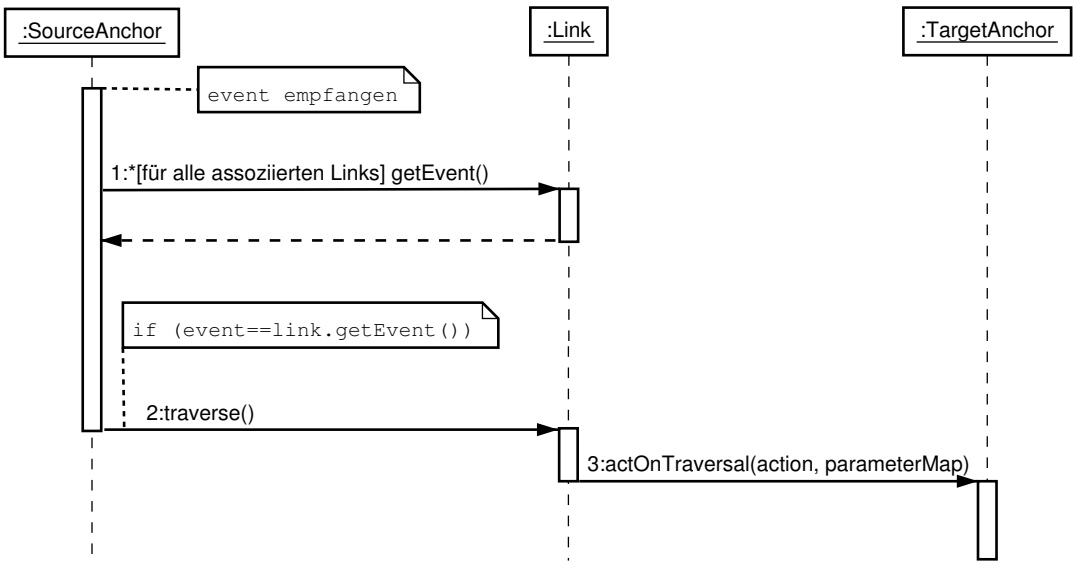

Abbildung 27.5: Sequenzdiagramm zum Linkkonzept

gebaut und dann als Gruppe gehandhabt werden. Ein rekursiver Aufbau von Medienobjekten ist jedoch nicht möglich. Zur Anwendung kommt diese Klasse vor allem im Ansichteneditor. In diesem kann eine Medienobjektgruppe, die aus zwei Elementen wie zum Beispiel dem linken und rechten Blinker besteht, als ein einziges Element im Cockpit platziert werden.

Das Cockpit kann aus Medienobjekten aufgebaut werden. Es sind bereits einige Medienobjektklassen implementiert. Diese können entweder direkt benutzt werden, oder man leitet von ihnen Unterklassen ab, um neue Arten von Medienobjekten zu definieren. Die existierenden Klassen sind dabei thematisch nach einfachen Leuchten, Instrumenten, Dialogen und Unterhaltungsmedien in Pakete gegliedert.

27.8 Emulation der Hardware

Alle Messdaten, die in einem echten Fahrzeug an der Sensorik oder in einer Simulation anfallen, werden von Sensoradaptern empfangen und auf geeignete Weise (siehe Abschnitt 27.6.1) an alle Interessenten weitergeleitet. Sie bilden also sozusagen die Schnittstelle des HyCop-Systems zur Außenwelt.

Da Sensorereignisse nicht nur von außen kommen, sondern auch von den Insassen durch Bedienen der Instrumente ausgelöst werden können, müssen Hardware-Bauteile emuliert werden. Diese Emulation wird vom Paket hardware übernommen. Jedes Exemplar der Klasse HardwareDevice stellt ein emuliertes Gerät dar und kapselt den Zustand desselben. Ein HardwareDevice kann mit einem Sensoradapter verbunden werden, der dann bei Änderungen des Zustandes informiert wird. Da z.B. kein realer Tank existiert, wird bei der Simulation eine Instanz der Klasse HardwareDevice angelegt, welche mit einem Sensoradapter für den Tankfüllstand verbunden werden kann, der dann den Zustand des Tankes überwacht und dem Hycop-Laufzeitsystem zur Verfügung stellt.

Zum Überwachen des Fahrzeugzustands existieren sogenannte Checklisten. Der Einsatz derselben wird in Abschnitt 28.1.3 beschrieben.

27.9 Simulation

Das Paket simulation erlaubt den Ablauf von in XML kodierten Simulationen; die Notation wird in Abschnitt 28.1.7 vorgestellt. Dabei wird im Wesentlichen zwischen zwei Typen von Ereignissen unterschieden, nämlich diskreten und fortlaufenden. Diskrete Ereignisse sind immer nur momentan, treten also zu einem festen Zeitpunkt punktuell auf, während fortlaufende Ereignisse die Veränderung eines Wertes über eine gegebene Zeitspanne hinweg darstellen.

27.9.1 Leuchten

Das Paket de.ls10.hycop.gui.cockpitlights enthält die Klassen für Kontroll-, Warn- und Fehlerleuchten. Diese können ein- oder ausgeschaltet sein und sind somit eher triviale Instrumente. Sie können z.B. zum Anzeigen des Zustandes von Blinker oder Scheinwerfer und für die Tankfüllstandswarnung genutzt werden. Jede Klasse des Pakets implementiert die Schnittstelle Updatable und reagiert in der Methode update unterschiedlich. Je nachdem, welche Werte als Parameter für die Methode update erwartet werden, nutzt man z.B. die Klasse BooleanCockpitLight, DoubleCockpitLight, und so weiter. Objekte der Klasse BooleanCockpitLight erwarten also Boolean-Objekte als Parameter und aktivieren sich bei dem Wert true und deaktivieren sich bei false. Objekte der anderen Klassen verwalten einen Minimal- und Maximalwert und aktivieren sich, wenn der übergebene Parameterwert außerhalb dieses Bereiches liegt, sonst deaktivieren sie sich.

Von der Klasse BooleanCockpitLight gibt es die Spezialisierung BlinkingBooleanCockpitLight. Im aktivierten Zustand schalten sich die entsprechenden Instrumente in konfigurierbaren Zeitabständen an und aus.

27.9.2 Instrumente

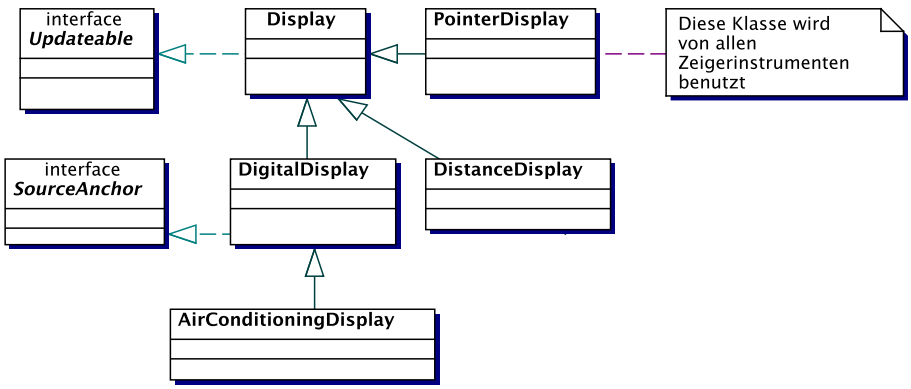

Abbildung 27.6: Instrumente

Das Paket de.ls10.hycop.gui.instruments beinhaltet Klassen für Instrumente (s. Abb. 27.6). Instrumente sind im Prinzip im Aufbau ähnlich den Kontrollleuchten, und auch ihre Klassen implementieren die Schnittstelle Updatable. Die in der Vererbungshierarchie oberste Klasse für Instrumente ist die Klasse Display. Sie wird beim Definieren spezieller Instrumenttypen entsprechend überschrieben. Je nach Instrument wird als Parameter der update-Methode ein Objekt eines bestimmten Typs erwartet. Dieser Parameter wird dann als Information aufgefasst, welche das Instrument im Cockpit anzeigen soll. Je nach Instrumenttyp werden also unterschiedliche Werte erwartet und unterschiedlich angezeigt, bei nicht erwarteten Werten wird ein Fehler ausgelöst. Instrumente besitzen unterschiedliche Parameter, mit denen sie über die Methode setParameter konfiguriert werden können. Diese Konfiguration geschieht durch das HyCop-Laufzeitsystem, analog zum Instanziieren sämtlicher Cockpitkomponenten dynamisch über die Konfigurationsdateien.

Obwohl Instrumente hauptsächlich zum Darstellen von Informationen gedacht sind, können sie bei Bedarf auf Mausereignisse reagieren. So kann beispielsweise ein Klick auf den Reisekilometerzähler bewirken, dass dieser auf 0 km zurückgesetzt wird.

Einige Instrumenttypen sind bereits implementiert. Es ist jedoch möglich, beliebige neue Instrumente zu entwickeln. Die Klasse PointerDisplay implementiert Zeigerinstrumente und kann z.B. für analoge Drehzahlmesser oder Geschwindigkeitsmesser genutzt werden. Die Klasse DigitalDisplay implementiert digitale Instrumente und kann z.B. für das digitale Anzeigen der Geschwindigkeit oder des Kilometerstandes genutzt werden. Es gibt viele weitere, speziellere Klassen.

27.9.3 Dialoge

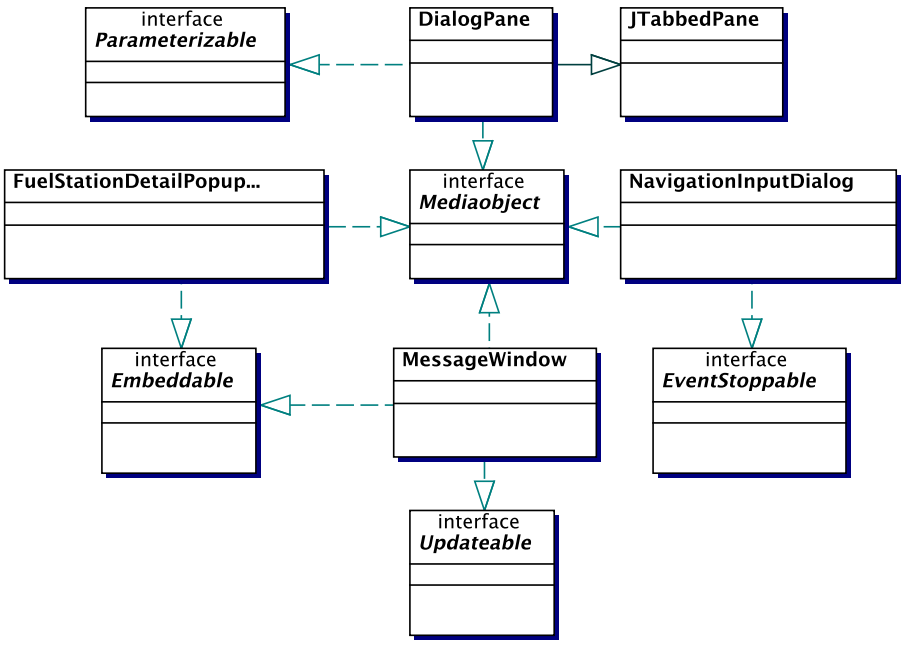

Abbildung 27.7: Dialoge

Das Paket gui.dialogs beinhaltet Klassen für Dialoge (s. Abb. 27.7). Dialoge unterscheiden sich von Instrumenten dadurch, dass sie eher der Interaktion mit dem Benutzer zugeordnet sind. Dazu gehören also Eingabedialoge und Auswahlmöglichkeiten, wie z.B. der Routenplanereingabedialog.

Um die Anzeige von Dialogen in Registerkarten zu realisieren gibt es die Klasse DialogPane. Sie ist ein Container für andere Dialoge, die die Schnittstelle Embeddable implementieren, wie z.B. das Nachrichtenfenster und die Tankstellenliste. Die einzelnen Komponenten, die einer Instanz der Klasse DialogPane zugeordnet sind, werden in einzelnen Registerkarten angezeigt. Der Benutzer kann zwischen den Registerkarten wechseln.

Für die Aufnahme und Anzeige von Nachrichten wir die Klasse MessageWindow benötigt. Nachrichten bestehen aus einem Text und haben einen bestimmten Typ. Über den Typ wird der Nachricht im Nachrichtenfenster ein Symbol zugeordnet, welches zusätzlich zum Text im Nachrichtentext angezeigt wird.

Um eine grössere Anzahl an Nachrichten auf beschränktem Platz verfügbar zu machen, gibt es die Klasse ScrollableMessageWindow. Sie zeigt ein MessageWindow in einem scrollbaren Bereich an, so dass der Benutzer bei Bedarf zu älteren Nachrichten zurückblättern kann.

Zuletzt ist die Klasse NavigationInputDialog als spezieller Eingabedialog für den Routenplaner realisiert worden.

27.10 Hilfswerkzeuge

In diesem Abschnitt werden Hilfsklassen beschrieben, die an verschiedenen Stellen benötigt werden, aber nicht zentraler Bestandteil der HyCop-Software sind.

27.10.1 Der Ausdrucksauswerter (evaluator)

In diesem Paket werden Klassen zur Auswertung von ganzzahligen und booleschen Ausdrücken zusammengefasst. Die Überprüfung solcher Ausdrücke wird im Ansichteneditor benötigt. Dort sind Bedingungen (Constraints) zu überprüfen, mit denen z.B. sinnlose Instrumentenanordnungen verboten werden können.

27.10.2 DOM-Hilfsklassen

Das Paket dom beinhaltet die Klasse DOMDocument. Diese wird für das Einlesen und Verarbeiten von XML-Dokumenten benutzt werden. Die Klasse stellt Methoden zum Abfragen von Elementknoten und von Attributen zur Verfügung. Die Klasse DOMErrorHandler stellt eine Standardimplementation für eine Behandlung von XML-Parser-Fehlern dar, welche beim Einlesen eines nicht konformen XML-Dokumentes auftreten.

27.10.3 Das util-Paket

Dieses Paket enthält einige Hilfsklassen, die z.B. zum Zeichnen von Digitalziffern und zum Vergleichen komplexer Objekte benutzt werden. Darüber hinaus sind Klassen zur Beschreibung der Eigenschaften der verfügbaren Tankstellen, Parkplätze und Abschleppdienste vorhanden.

Kapitel 28

Konfiguration der HyCop-Software

Autoren: *Daniel Mölle*
Markus Niehammer

Das folgende Kapitel beschreibt die Konfiguration der HyCop-Software. Dem Fahrzeughersteller obliegt es, die grundlegenden Eigenschaften der Fahrzeug-Hardware und die verfügbaren Ansichten anzugeben. Der Benutzer hingegen kann die vordefinierten Ansichten mit Hilfe des Ansichteneditors an die individuellen Bedürfnisse anpassen, wobei allerdings einschränkende Vorgaben des Herstellers eingehalten werden müssen. Auf diese Weise kann beispielsweise ausgeschlossen werden, dass der Fahrer eine Ansicht erzeugt, die der Straßenverkehrs(zulassungs)ordnung widerspricht.

28.1 Konfiguration durch den Hersteller

28.1.1 Fahrzeugaufbau

Die grundlegenden Eigenschaften des Fahrzeugs werden in der Datei vehicle.xml angegeben. Dazu gehören primitive Eigenschaften, die als Name/Wert-Paare angegeben werden können, z.B. benötigter Kraftstofftyp, Anzahl der Sitzpläte etc.:

```
<property name="Kraftstoff" value="Diesel"/>
<property name="Sitzplaetze" value="2"/>
```

In Umgebungen werden die einzelnen Hardwarekomponenten und Sitzplätze beschrieben. Diese sind weitergehend konfigurierbar durch einen Namen und eine Konfigurationsdatei, die per Attribut angegeben wird. Einer Umgebung können ein Display und verschiedene Ansichten, von denen das Display zu jedem Zeitpunkt genau eine zeigt, zugewiesen werden, wobei die Größe des Displays und eine Standardansicht, die das Display stets zu Beginn zeigen soll, festgelegt werden. Die Umgebung für den Fahrersitz könnte beispielsweise wie folgt aussehen:

```
<environment name="driver" config="cockpit.xml">
```

```
  <display width="25" height="18" defaultview="Standansicht"/>
  <viewtype name="Standansicht"/>
  <viewtype name="Stadtansicht"/>
  <viewtype name="Ueberlandansicht"/>
  <viewtype name="Sportansicht"/>
  <viewtype name="Unterhaltungsansicht" critical="yes"/>
</environment>
```

Der Fahrersitz hat die Bezeichnung driver und wird in der Datei config.xml näher beschrieben. Er verfügt über ein Display von 25×18 Blöcken, wobei ein Block eine atomare Größeneinheit ist und 32×32 Pixeln entspricht (Größe einer Kontrollleuchte im mitgelieferten Demo-Design). Als Standardansicht für den Fahrer wird die Standansicht angegeben, die beim Start der Umgebung erscheint. Außerdem werden noch Stadt-, Überland-, Sport- und Unterhaltungsansicht zur Verfügung gestellt. Die Unterhaltungsansicht hat das Attribut critical , welches aussagt, dass diese Ansicht nicht angezeigt werden darf, wenn der Motor an ist.

Die XML-Dateien, die die einzelnen Ansichten beschreiben, werden automatisch vom Editor erstellt und bedürfen daher keiner weiteren Dokumentation.

28.1.2 Konfiguration der Umgebung

Für jede der in der Datei vehicle.xml angegebenen Umgebungen existiert eine weitere Datei mit detaillierteren Einstellungen. Der Name der Datei entspricht dem in vehicle.xml angegebenen Attribut config. Weiterhin gibt es noch eine Datei mediaobjects.xml mit globalen Einstellungen, die für alle Umgebungen gelten. Diese sind zum Beispiel feste Verknüpfungsstrukturen zwischen Bestandteilen des Cockpits oder überall verfügbare Medienobjekte.

Die Dateien definieren neben den in einer Umgebung verfügbaren Medienobjekten (Instrumente) auch Verknüpfungen zwischen Objekten und Bindungen an Sensoradapter, die Cockpitelemente mit Hardwaredaten versorgen. Es ist möglich, Medienobjekte aus dieser globalen Datei in den einzelnen Umgebungen zu überschreiben. Verknüpfungen und Bindungen hingegen sind fest und können in den einzelnen Umgebungen nur hinzugefügt werden.

Medienobjekte

Ein Medienobjekt hat einen Namen, eine zugehörige Java-Klasse und eine Beschreibung. Es können beliebig viele Parameter als Name/Wert-Paare angegeben werden. Welche Parameter eine Klasse unterstützt, kann ihrer Methode getSupportedParameterNames() entnommen werden.

Medienobjekte werden über ihren Namen kategorisiert. Der Name setzt sich zusammen aus der Kategorie, gefolgt von einem Punkt und dem Namen des Instruments. Es können beliebige Kategorien und Namen benutzt werden, der Punkt zur Trennung ist aber in jedem Fall erforderlich.

```
<instrument name="display.tripodometer"
```

```
      class="de.ls10.hycop.gui.instruments.DigitalDisplay"
      description="Reisekilometerzähler">
  <parameter name="image" value="icons/tripOdometer.png"/>
  <parameter name="leftdigitx" value="12"/>
  <parameter name="leftdigity" value="5"/>
  <parameter name="digitsbeforedecimalpoint" value="6"/>
  <parameter name="digitsafterdecimalpoint" value="1"/>
  <parameter name="minvalue" value="0.0"/>
</instrument>
```

Im obigen Beispiel wird ein Medienobjekt der Kategorie display und des Namens tripodometer definiert. Als Java-Klasse wird hierfür die Klasse DigitalDisplay benutzt, die das Anzeigen von Zahlen in Digitaldarstellung ermöglicht. Als Parameter erhält das Instrument eine Bilddatei mit der Hintergrundgrafik und einige Angaben zu Positionierung, Format und Wertebereich der auszugebenden Daten.

Instrumente können zu Gruppen zusammengefasst werden, die dann zusammen als ein Superinstrument erscheinen. Dieses tritt dann sowohl im Ansichteneditor wie auch im Cockpit selbst nur als ein zusammenhängendes Instrument auf, die einzelnen Teile sind nicht mehr erkennbar, werden jedoch bei Verknüpfung und Bindungen einzeln angegeben.

```
<group name="gt.signal-panel"
       description="GT-Blinkerleiste">
  <item x="0" y="0" width="1" height="1">
    <instrument name="gt.button.turnsignalleft"
          class="de.ls10.hycop.gui.cockpitlights.BooleanCockpitLight"
          description="GT-Blinkerschalter links">
      <parameter name="image-on" value="icons/gt/flipbutton-on.png"/>
      <parameter name="image-off" value="icons/gt/flipbutton-off.png"/>
    </instrument>
  </item>

  <item x="1" y="0" width="1" height="1">
    <instrument name="gt.button.turnsignalright"
          class="de.ls10.hycop.gui.cockpitlights.BooleanCockpitLight"
          description="GT-Blinkerschalter rechts">
      <parameter name="image-on" value="icons/gt/flipbutton-on.png"/>
      <parameter name="image-off" value="icons/gt/flipbutton-off.png"/>
    </instrument>
  </item>
</group>
```

Das Instrument gt.signal-panel (Kategorie gt, Name signal-panel) besteht aus den beiden Teilinstrumenten button.turnsignalleft und button.turnsignalright, die wie oben beschrieben definiert und parametrisiert werden. Die Verwendung von Subkategorien (hier .button) ist dabei optional und bleibt ohne Auswirkung. Innerhalb einer Gruppierung wird weiterhin die relative Position und die Größe jedes Teilinstruments angegeben.

Verknüpfungen

Es können zwei Arten von Verknüpfungen identifiziert werden, nämlich solche zwischen einem Medienobjekt und einem Sensoradapter und solche zwischen zwei Medienobjekten. Erstere können als Bindungen und Letztere als Links bezeichnet werden. Die Bindung von Medienobjekten an Sensoradapter erfolgt durch die Angabe der beteiligten Komponenten im Abschnitt bindings. Dabei fungiert das Medienobjekt als Empfänger, der Sensoradapter als Sender von Daten. Eine Bindung wird über das XML-Element subscription angegeben. Dabei wird als Attribut listener das Medienobjekt und als Attribut generator der Sensoradapter jeweils über den Namen angegeben.

```
<subscription listener="display.tripodometer"
        generator="sensoradapter.tripodometer" />
<subscription listener="display.enginespeed"
        generator="sensoradapter.enginespeed" />
```

Hier wird der Reisekilometerzähler mit dem sensoradapter.tripodometer verbunden, der Geschwindigkeitsmesser mit dem sensoradapter.enginespeed.

Abbildung 28.1: Link

Bei Links zwischen Medienobjekten (siehe Abbildung 28.1) werden Quelle (source) und Ziel (target) angegeben. event bestimmt, wann eine Vernüpfung verfolgt werden soll, d.h. welches Ereignis in der source-Klasse eintreten muss. action beschreibt, wie das Ziel des Links auf das Traversiern desselbigen reagieren soll. Welche Aktionen eine Klasse beherrscht kann jeweils der Methode actOnTraversal entnommen werden. Welche Ereignisse eine Linkverfolgung auslösen, wird ebenfalls in der entsprechenden Klasse festgelegt. Es sind beliebig viele Links, Ereignisse und Aktionen möglich.

```
<link source="button.hazardsignal"
      target="device.hazardsignal"
      event="touched"
      action="toggle"/>
<link source="button.hazardsignal"
      target="device.turnsignalright"
      event="touched"
      action="off"/>
```

Beim Betätigen des Schalters für die Warnblinker wird die Warnblinkanlage um- und der rechte Blinker ausgeschaltet.

```
<link source="display.fuel"
      target="display.fuelstationslist"
      event="touched"
      action="show"/>
```

Beim Berühren der Tankanzeige wird eine Liste mit Tankstellen eingeblendet.

28.1.3 Checklisten

Die Idee, Checklisten einzusetzen, geht auf die ersten Überlegungen zum HyCop-Szenario zurück. Benötigt wird ein Mechanismus, der durch einen bestimmten Vorgang ausgelöst wird und dann für verschiedene Sensoren überprüft, ob sie unkritische Werte liefern. Ursprünglich sollte das Anlassen des Motors als Auslöser dienen, und es sollten alle für die Fahrbereitschaft relevanten Elemente des Fahrzeugstatus geprüft werden. Aufgrund der hohen Konfigurierbarkeit der Hardware hat es sich aber als sinnvoll herausgestellt, auch Checklisten flexibler zu halten. Die entsprechenden Klassen sind im Paket checklist gebündelt.

Um diese Flexibilität zu ermöglichen, werden Checklisten (Klasse Checklist) in den HyCop-Konfigurationsdateien definiert. Sie werden innerhalb des XML-Elementes checklists angegeben. Dabei wird ihnen neben einem Namen, einem auslösendem Sensoradapter und die Komponentenbezeichnung eines Medienobjekts auch eine beliebige Zahl von Tests (Klassen CheckItem und CheckItemComparable) zugewiesen. Es folgt ein Beispiel einer Checkliste, welche zwei Test beinhaltet.

```
<checklist name="standard"
           adapter="sensoradapter.engineon"
           target="display.messages">
    <check adapter="sensoradapter.battery" type="java.lang.Double"
           testtype="greater" testvalue="1.0"
           failmessage="Kritischer Batteriestand." />
    <check adapter="sensoradapter.abs" type="java.lang.Boolean"
           testtype="equals" testvalue="false"
           failmessage="ABS-Ausfall." />
</checklist>
```

Der Name dient der Kennzeichnung der Checkliste. Dadurch können verschiedene Checklisten unterschiedliche Arten von Fehlermeldungen erzeugen, auch wenn sie denselben Test durchführen. Dies wird in HyCop benutzt um zwei verschiedene Detailstufen, eine ausführliche in der Werkstattansicht und eine weniger ausführliche in der Fahreransicht, zu erzeugen.

Als Auslöser kommen alle Sensoradapter in Frage, die Boolesche Werte liefern. Die Checklisten müssen dann an diese Sensoradapter gebunden werden. Immer dann, wenn dieser Wert

von false auf true umspringt, wird die Checkliste durchlaufen. So wird zum Beispiel die Standardcheckliste in der normalen Konfiguration jedesmal abgearbeitet, wenn der Motor angelassen wird. Auf dieselbe Weise ließe sich beispielsweise eine Checkliste angeben, die immer dann, wenn die Bodenfrostwarnung aktiviert wird, überprüft, ob der Reifendruck in einem für Glatteis geeigneten Wertebereich liegt.

Die zu einer Checkliste gehörenden Tests lassen sich in zwei Typen unterteilen, Gleichheitstests (Klasse CheckItem) und Schrankentests (Klasse CheckItemComparable). Ein typischer Gleichheitstest wird wie folgt definiert:

```
<check adapter="sensoradapter.abs"
       type="java.lang.Boolean"
       testtype="equals"
       testvalue="false"
       failmessage="ABS-Ausfall." />
```

In diesem beispielhaften Fall wird also der ABS-Sensoradapter getestet. Dieser liefert true, wenn das ABS-System ausgefallen ist. Als Rückgabe wird ein Boolescher Wert erwartet, der auf Gleichheit mit dem Wert false zu prüfen ist. Wenn der Sensoradapter zuletzt diesen Wert verschickt hat, gilt der Test als erfolgreich. Andernfalls wird die angegebene Fehlermeldung ausgegeben.

Schrankentests verlaufen analog, überprüfen aber die Einhaltung von Schranken. So ist der folgende Test genau dann erfolgreich, wenn der Batterie-Sensoradapter zuletzt einen Double-Wert über $10,0$ gemeldet hat:

```
<check adapter="sensoradapter.battery"
       type="java.lang.Double"
       testtype="greater"
       testvalue="10.0"
       failmessage="Kritischer Batteriestand." />
```

Beim Versand der Fehlermeldung wird wie folgt vorgegangen: Wenn es ein Medienobjekt mit der angegebenen Komponentenbezeichnung gibt, wird die Meldung an dieses verschickt; andernfalls wird die Meldung auf der Konsole ausgegeben.

28.1.4 Hardware

Bezüglich der Fahrzeug-Hardware wird zwischen einfachen Sensoradaptern und eventuell komplexeren Geräten, wie zum Beispiel Steuergeräten oder einem Navigationssystem, unterschieden. Ein Sensoradapter dient als Schnittstelle zwischen Cockpit und Außenwelt, an der entweder simulierte oder reale Sensoren angelegt werden können. Über das messaging-System können sich Instrumente über Änderungen an den Sensoradaptern informieren lassen, so dass ihnen immer die aktuellen Messdaten zur Verfügung stehen.

Bei der Konfiguration eines Adapters wird lediglich sein Bezeichner festgelegt. Ein Sensoradapter wird dabei aus der gleichnamigen Klasse de.ls10.hycop.SensorAdapter instanziiert. Die

Bedeutung der über ihn verschickten Daten wird erst durch die angeschlossenen Geräte und Instrumente festgelegt. Der folgende, beispielhafte Ausschnitt aus der Datei hardware.xml verdeutlicht die Syntax.

```
<adapters>
  <sensoradapter name="sensoradapter.speed" />
  <sensoradapter name="sensoradapter.enginespeed" />
</adapters>
```

Im Gegensatz zu Sensoradaptern werden Geräte (-schnittstellen) aus verschiedenen Klassen gebildet. Außerdem wird jedem Gerät ein Sensoradapter zugeordnet, über den es mit den Instrumenten kommunizieren kann. Das Gerät für den Reisekilometerzähler wird beispielsweise wie folgt konfiguriert:

```
<hardware>
  <device name="device.tripodometer"
          class="de.ls10.hycop.hardware.DoubleDevice"
          adapter="sensoradapter.tripodometer" />
</hardware>
```

Die Zuweisung des eindeutigen Bezeichners device.tripodometer ist deshalb wichtig, weil das Gerät als mögliches Ziel eines Hyperlinks addressierbar sein muss. Dies kann der Fall sein, wenn man im Cockpit etwas auslösen möchte, was den Zustand eines Gerätes verändert (z.B. Kilometerstand zurücksetzen). Das Gerät ist vom Typ DoubleDevice, hat also als inneren Zustand im Wesentlichen lediglich eine reelle Zahl, in diesem Fall natürlich den Kilometerstand. Angeschlossen wird das Gerät an den Reisekilometer-Sensoradapter. Auf diese Weise ist es nun möglich, das Gerät als Zielanker eines Hyperlinks zu verwenden, dessen Traversierung eine Änderung des Kilometerstandes bewirkt. Wenn in der Cockpit-Konfiguration noch der folgende Eintrag eingefügt wird, führt das Berühren des Reisekilometerzählers zu dessen Rücksetzung auf den Wert 0:

```
<link source="display.tripodometer"
      target="device.tripodometer"
      event="touched"
      action="0" />
```

28.1.5 Ansichten und Layout-Vorgaben

Die verfügbaren Ansichten werden über die Datei viewtypes.xml konfiguriert. Zu einer Ansicht wird ein Bezeichner, ein Dateiname und optional der Parameter readonly="yes" angegeben. Letzterer bewirkt, dass die Ansicht im Ansichteneditor nicht bearbeitet werden kann.

Zu jeder Ansicht können bestimmte Rahmenbedingungen vorgegeben werden, die der Benutzer beim Verändern der Ansicht nicht verletzen darf. Dazu gehören Platzierungsregeln, Existenzforderungen sowie freie Nebenbedingungen in der Form Boolescher Ausdrücke.

Für die Angabe von Platzierungsregeln kann eine Ansicht in verschiedene Bereiche aufgeteilt werden, die aus Rechtecken zusammengesetzt werden. Der verbleibende Platz wird ebenfalls als Bereich aufgefasst und im Folgenden als Rest bezeichnet. Jedem Bereich lassen sich verschiedene Medienobjekte zuweisen, die dort platziert werden dürfen oder müssen. Die Standardvorgabe für jedes Instrument ist, dass es optional ist und in den Restbereich gehört.

Die Zuweisung von Medienobjekten kann auf zweierlei Weise geschehen. Bei der expliziten Zuweisung wird das Medienobjekt konkret benannt und einem Bereich als optional oder erforderlich zugewiesen. Bei der Gruppenzuweisung hingegen werden alle Medienobjekte, deren Namen mit dem angegebenen Präfix beginnen und die nicht explizit behandelt werden, einem Bereich als optional oder erforderlich zugewiesen. Die Konfiguration soll an dem folgenden Beispiel exemplifiziert werden:

```
<viewtypes>
  <viewtype name="Beispielansicht"
            file="views/view.example.xml"
            readonly="yes">
    <area name="Leuchten">
      <rectangle x1="0"  y1="0"  x2="3"  y2="17" />
      <rectangle x1="0"  y1="15" x2="24" y1="17" />
      <rectangle x1="22" y1="0"  x2="24" y2="17" />

      <objects prefix="control." required="no" />
      <objects prefix="warning." required="yes" />
      <objects prefix="failure." required="yes" />

      <object name="control.hazardsignal" required="yes" />
      <object name="control.parkinglight" required="yes" />
    </area>
    <rest>
      <object name="display.distance" required="yes" />
    </rest>
  </viewtype>
</viewtypes>
```

Die hierdurch definierte Beispielansicht besitzt zwei Bereiche. Der erste Bereich besteht aus drei Rechtecken, die jeweils den linken, unteren und rechten Rand des Cockpits umfassen. Alle Kontroll-, Warn- bzw. Fehlerleuchten (Präfices control., warning. bzw. failure.) werden diesem Bereich zugeordnet, wobei die diversen Kontrollleuchten optional, die Leuchten der anderen beiden Kategorien aber erforderlich sind. Allerdings werden zwei spezielle Kontrollleuchten, nämlich die für die Warnblinkanlage und das Parklicht, explizit als erforderlich angegeben. Ihre Optionalität entfällt dadurch, weil explizite Zuweisungen Werte aus einer Gruppenzuweisung überschreiben.

Alle anderen Medienobjekte gehören gemäß der Voreinstellung in den zweiten Bereich (rest) und sind optional. Dabei wird in der obigen Konfiguration der Abstandswarner als erforderlich ausgezeichnet, wodurch die Einstellungen für dieses konkrete Medienobjekt erwartungsgemäß überschrieben werden.

28.1.6 Freie Nebenbedingungen

Die Notwendigkeit, weitere Nebenbedingungen stellen zu können, ergibt sich aus den realen Anforderungen an Fahrzeug-Cockpits. Eines der intuitivsten Beispiele hierfür ist die Forderung, dass die Kontrollleuchte für den linken Blinker links von der Kontrollleuchte für den rechten Blinker liegen sollte. Da sich die für einen praktischen Einsatz sinnvollen Forderungen ohne das entsprechende Know-How nur schwer abschätzen lassen, wird bei HyCop ein relativ allgemeiner Constraint-Prüfer eingesetzt, der auf Booleschen Ausdrücken operiert.

Neben den nullären Konstruktoren true und false werden auch zahlreiche Boolesche Operatoren unterstützt. Dazu zählen die üblichen Operatoren not, and, or und imply für die Negation eines Ausdrucks, die Konjunktion bzw. Disjunktion über Listen von Ausdrücken und die Implikation über zwei Ausdrücken. Weiterhin gibt es die Vergleichsoperatoren equals, less und greater, die jeweils über zwei ganzzahligen Ausdrücken definiert sind. Eine Besonderheit stellt das Element boolproperty dar, mit dem Boolesche Attribute von Medienobjekten abgefragt werden können. Ein Ausdruck dieses Typs stellt also eine Boolesche Variable dar.

Analog dazu dient das Element intproperty der Abfrage ganzzahliger Attribute von Medienobjekten, beispielsweise der Position des Instruments auf einer bestimmten Achse. Die Besonderheit hierbei ist, dass sich die Position natürlich nicht feststellen lässt, wenn das Instrument gar nicht in die Ansicht aufgenommen wurde. Ein Constraint, der Ausdrücke des Typs intproperty enthält, die sich aus dem oben genannten Grund nicht auswerten lassen, wird dabei automatisch als erfüllt angesehen. Wenn dieses Verhalten nicht erwünscht ist, können die Instrumente einfach als erforderlich deklariert oder ihre Existenz in der Ansicht über boolproperty-Ausdrücke eingefordert werden. Mit intconstant schließlich lassen sich ganzzahlige Konstanten einfügen.

Die Nebenbedingungen zu einem Ansichtstyp werden jeweils unterhalb des zugehörigen rest-Elements angegeben. Zu einer Bedingung gehört neben dem Booleschen Ausdruck auch eine Warnmeldung, die ausgegeben werden soll, wenn die Bedingung als verletzt erkannt wird. Im Folgenden sollen drei Beispiele vorgestellt werden.

```
<constraint failMessage="Parkplatzsuche erfordert Dialogkarteikasten">
  <imply>
    <boolproperty object="button.parking" attribute="included" />
    <boolproperty object="dialog.dialogpane" attribute="included" />
  </imply>
</constraint>
```

Dieser Constraint besagt, dass eine Verwendung des Knopfs für die Parkplatzsuche die Existenz des Dialogkarteikastens implizieren soll. Die Bedingung ist deshalb sinnvoll, weil die Liste der gefundenen Parkplätze in diesem Karteikasten angezeigt wird.

```
<constraint failMessage="Illegale Tacho-Platzierung">
  <or>
    <less>
      <intproperty object="display.speed" attribute="xpos" />
```

```
      <intconstant value="5" />
    </less>
    <equals>
      <intproperty object="display.speed" attribute="ypos" />
      <intproperty object="display.enginespeed" attribute="ypos" />
    </equals>
  </or>
</constraint>
```

Dieser Constraint demonstriert die Verwendung ganzzahliger Ausdrücke. Die Bedingung ist erfüllt, wenn die Geschwindigkeitsanzeige entweder im linken Randbereich der Ansicht ($x < 5$) oder auf derselben Höhe wie der Drehzahlmesser liegt. Allerdings ist die Bedingung auch dann erfüllt, wenn mindestens eines der beiden Instrumente gar nicht in der Ansicht enthalten ist. Um dieses Verhalten abzustellen, könnten beide als erforderlich deklariert werden.

Im Rahmen des letzten Beispiels sollen Besonderheiten erläutert werden, die sich durch die Auswertungsreihenfolge ergeben. Die Argumente von Implikationen, Konjunktionen und Disjunktionen werden in der gegebenen Reihenfolge durchlaufen, wobei die Auswertung der Argumente unter Umständen frühzeitig abgebrochen werden kann. So liefert beispielsweise die Implikation das Ergebnis true, wenn das erste Argument als false bewertet wird, ohne das zweite Argument zu beachten. Wenn bei der Auswertung einer Konjunktion ein Argument den Wert false hat, wird sofort false zurückgeliefert. Dual gilt dies für die Disjunktion.

Im letzten Beispiel soll eine robuste Version des obigen Constraints angegeben werden, die genau dann erfüllt sein soll, wenn mindestens eine der folgenden Bedingungen gilt:

1. Es gibt keine Geschwindigkeitsanzeige in der Ansicht.
2. Es gibt eine Geschwindigkeitsanzeige am linken Rand der Ansicht.
3. Es gibt sowohl eine Geschwindigkeitsanzeige als auch einen Drehzahlmesser, und diese befinden sich auf derselben Höhe.

Der Unterschied zum obigen Beispiel ist also der, dass ein Fehlen des Drehzahlmessers nicht automatisch zum Erfüllen der Bedingung führt.

```
<constraint failMessage="Illegale Tacho-Platzierung (2)">
  <or>
    <less>
      <intproperty object="display.speed" attribute="xpos" />
      <intconstant value="5" />
    </less>
    <and>
      <boolproperty object="display.enginespeed" attribute="included" />
      <equals>
        <intproperty object="display.speed" attribute="ypos" />
        <intproperty object="display.enginespeed" attribute="ypos" />
```

```
      </equals>
    </and>
  </or>
</constraint>
```

Dieser Constraint leistet das Gewünschte, weil das Fehlen des Drehzahlmessers frühzeitig erkannt und die Auswertung der Konjunktion deshalb abgebrochen würde, bevor es zu einer Abfrage seiner Y-Position kommen kann.

28.1.7 XML-Simulation

Die XML-Simulation dient dazu, die Funktionalität des Fahrercockpits in möglichst realitätsnaher Weise zu testen. Als Grundlage dafür diente uns die bereits im 1. Semester verfasste Story. Wir haben mittels einfacher XML-Elemente die verschiedenen Sensoradapter des Cockpits derart mit Werten versorgt, dass dadurch die verschiedenen in der Story vorkommenden Abläufe anhand der Anzeigen des Cockpits nachempfunden werden können. Dazu wurden ausschließlich die folgenden drei Typen von XML-Tags verwendet, für die hier konkrete Beispiele gegeben sind:

```
<setvalue start="100"
        adapter="sensoradapter.time"
        type="double"
        value="36000" />

<slidevalue start="200"
        stop="300"
        adapter="sensoradapter.fuel"
        type="double"
        from="88.5"
        to="46.4" />

<accelerateto start="200"
        stop="250"
        to="50" />
```

setvalue setzt den Wert eines Sensoradapters, slidevalue läßt den Wert eines Sensoradapters von einem durch die property from gegebenen Wert zu einem durch die property to spezifizierten Wert gleiten und das Makro-Tag accelerateto verändert die Werte der Sensoradapter für den Tachometer, den Drehzahlmesser und die Gangschaltung derart, dass eine Geschwindigkeitsänderung vom aktuellen Wert zu dem durch die property to spezifizierten Wert simuliert wird. Die Kernidee, welche es uns ermöglicht, einen Ablauf in der Zeit zu simulieren, besteht nun darin, für jede Wertänderung einen genauen Zeitpunkt oder eine genaue Zeitspanne festzulegen. Dies leisten die properties start und stop. Die Simulation beginnt zum Zeitpunkt 0, von dem aus in Zehntelsekundenschritten vorwärts gezählt wird. start definiert diejenige Zehntelsekunde nach 0, zu der eine entsprechende Veränderung stattfinden bzw. beginnen soll, und

`stop` definiert in analoger Weise die Zehntelsekunde nach 0, zu der die Veränderung enden soll, sofern sie eine zeitliche Ausdehnung hat.

Teil VII

Fazit

Kapitel 29

Fazit

Autoren: *Daniel Mölle*
Bastian Krol
Rafael Hosenberg
Stefan Borggraefe

29.1 Zusammenfassung

Es wurde ein Programmpaket entwickelt, mit dem grafische Benutzeroberflächen in XML konfiguriert und als Hyperdokument repräsentiert werden. Die erste Hälfte des Projektzeitraums wurde für eine intensive Anforderungsanlyse verwendet, die zweite Hälfte stand für die Implementierung zur Verfügung.

Während die Funktionalität jeder eigenen Kategorie von GUI-Elementen (wie Leuchten, Rundinstrumente oder Nachrichtenfenster) nach wie vor implementiert werden muss, können die grafische Repräsentation und die funktionalen Verknüpfungen jedes einzelnen GUI-Elements ohne Zugriff auf den Quelltext geändert werden. Für die Kommunikation des Systems mit der Außenwelt, also Sensoren und steuerbaren Geräten, stehen einfache Schnittstellen zur Verfügung, so dass Simulation und reale Anwendung für die Software nicht unterscheidbar sind.

In Ermangelung echter Hardware wurde ein vollwertiges Simulationssystem entwickelt, das die Wiedergabe von in XML festgelegten Simulationsläufen gestattet. Da aus solchen Läufen heraus alle Sensoren und Geräte ansprechbar sind, kann das Simulationspaket auch für funktionale Tests oder zur Vorführung des Systems verwendet werden.

Die Konfiguration wurde am Beispiel des Cockpits für Automobile ausgerichtet. Dabei wurden eine Vielzahl an Ansichten, Instrumenten, Sensoren und Steuergeräten berücksichtigt, so dass mehrere tausend Zeilen XML-Code und über 150 Einzelgrafiken erzeugt werden mussten.

29.2 Beurteilung

Die Kernaufgabe der Projektgruppe bestand in der Entwicklung eines GUI-Systems unter der Beachtung von Designtechniken, die sonst eher bei der Konstruktion hypermedialer Systeme zum Einsatz kommen; insbesondere sind hiermit die Konzepte gemeint, die der Sprache DoDL zugrunde liegen. Der Nachweis, dass dieses Ziel erreicht wurde, fällt ausgesprochen leicht: die HyCop-Software stellt ein System zur Verfügung, welches gundlegend Medienobjekte und Hyperlinks verwaltet und welche die Basis bilden, auf der der Anwender mit Hilfe von HyCop hochflexibel Hyperdokumente definiert. Diese Hyperdokumente stellen die Ansichten dar, die im HyCop-Fenster angezeigt werden und zwischen welchen beliebig hin- und hergeschaltet werden kann. Jedes GUI-Element wird durch ein Medienobjekt repräsentiert. Hyperlinks, die selbst eigenständige Objekte sind, dienen der Darstellung von Kontrollflüssen.

Allerdings bestehen auch Abweichungen von DoDL-typischen Sichtweisen. So sind Medienobjekte nicht beliebig verschachtelbar; nur Objektgruppen und Dialogkarteikästen können andere Medienobjekte aufnehmen. Deswegen werden zur Speicherung der Medienobjekte fast ausschließlich Listen und *maps*, aber keine baumartigen Strukturen verwendet. Gegen eine Verwendung des eher relativen DoDL-Positionierungskonzepts spricht weiterhin, dass eine absolute Positionierung jedes einzelnen Elementes möglich sein muss.

Eine Nebenaufgabe war die Erprobung eines unkonventionellen Entwicklungsprozesses. Die Anforderungsanalyse wurde um eine Betrachtung der Medienobjekte und ihrer Verknüpfungen erweitert, die Implementierung hingegen auf *eXtreme Programming* ausgerichtet. Insbesondere aufgrund der letztgenannten Entscheidung bewahrheitete sich die bereits im Fazit zum ersten Semester erwähnte Befürchtung, dass einige der in der Anforderungsanalyse erarbeiteten Ergebnisse bei der Implementierung überhaupt keine Rolle mehr spielen könnten. Es wurden viele Strukturentscheidungen neu getroffen, da während der Programmentwicklung bessere Alternativen sichtbar wurden. So kam beispielsweise den Zustandsübergangsdiagrammen nur wenig Bedeutung zu. Triviale Zustände wie „sichtbar“ und „nicht sichtbar“ hätten sich ohnehin ergeben; andererseits mussten die sehr detaillierten Vorgaben zur Funktionsweise einzelner Instrumente verworfen werden, weil sich andere Konzeptionen aufdrängten. Außerdem führte die starke Parametrisierbarkeit der einzelnen Instrumenttypen zu einer rapiden Vergrößerung ihres Zustandsraums. Ähnliches gilt für die Medienobjektdiagramme: Die in ihnen festgelegten Daten- und Kontrollflüsse zwischen den Medienobjekten, aber auch die exakte Vorgabe der zu beachtenden Medienobjekte waren für die Implementierung nur selten relevant. Sowohl die Objekte als auch die Daten- und Kontrollflüsse sind nämlich hochgradig konfigurabel – und HyCop ist somit viel allgemeiner anwendbar, als es die Medienobjektdiagramme überhaupt nahelegen. Weiterhin existieren viele indirekte Zusammenhänge zwischen Medienobjekten, die erst bei der Betrachtung des Zusammenspiels von Instrumenten und Fahrzeughardware sichtbar werden und die somit in den Diagrammen gar nicht darstellbar gewesen wären.

Bezüglich des Entwicklungsprozesses ist also festzuhalten, dass die im *eXtreme Programming* gestellte Forderung, auf eine umfassende Anforderungsanalyse zu verzichten, aus unserer Sicht berechtigt ist. Tatsächlich haben die im Rahmen der inkrementellen Entwicklung neu gefällten Strukturentscheidungen praktisch immer positive Auswirkungen auf das Projekt gehabt, vor allem auf die Flexibilität und die konzeptionelle „Sauberkeit“. Eine etwas andere Perspektive, die man nicht außer acht lassen sollte ist die, dass gerade die vorhergehende Anforderungsanalyse die Teilnehmer erst in die Lage versetzte, während der Implementierung durch genügend

Sachkenntnisse und Überblick, gute Entscheidungen zu treffen.

Was den Prozess des eXtreme Programming im Detail betrifft, konnten trotz einer gewissen Skepsis alle Erwartungen, die wir zu Beginn der Implementierung hatten, erfüllt werden. Die einzelnen XP-Techniken konnten weitgehend den Planungen entsprechend angewendet werden und die Vorteile, die für uns durch den Einsatz von XP entstanden sind haben sich auch auf andere Tätigkeiten im Rahmen der Projektgruppe ausgewirkt. So wurde z.B. auch der Endbericht zum größten Teil unter Einsatz von Pair-„Programming“ erstellt.

Das Projekt hat gezeigt, dass die hypermediale Beschreibung von GUIs nicht nur grundsätzlich möglich, sondern auch mit angenehmen Konsequenzen verbunden ist. Aufgrund der allgemeinen Auslegung ließ sich ein erstaunlich hoher Anteil der Cockpit-Funktionalität allein durch die Anpassung von Konfigurationsdateien realisieren. So lassen sich auch weitere Medienobjekte, beispielsweise zur Kommunikation (Telefon, Internet, etc.) einfach in das bestehende System integrieren. Es muss allerdings für jedes Medienobjekt mit spezieller Funktionalität nach wie vor eine entsprechende Klasse existieren. Es stellt sich also die Frage, ob hier ein guter *trade-off* zwischen Flexibilität der Software und Einfachheit der Konfiguration vorliegt, oder ob eine weitere Erhöhung der Konfigurierbarkeit zu noch besseren Ergebnissen führen kann. Wir behaupten Ersteres – und dies mit der einfachen Begründung, dass eine weitere, nicht-marginale Steigerung der Ausdruckskraft der verwendeten Konfigurationssprache dazu führen muss, dass in den Konfigurationsdateien auch Programmcode eingebettet werden kann. Das aber kann nicht sinnvoll sein, weil wir an konfigurierbaren GUIs sowie einer Anlehnung an Hypermedia-Modelle und eben nicht an der Entwicklung einer neuen Interpretersprache interessiert sind.

Eine gänzlich andere Abschätzung ergibt sich bezüglich der potenziellen Anwendungsbereiche der HyCop-Software. Auch dies ist eine Frage der Abwägung zwischen zwei Extremen: Ist das System so speziell, dass es ohnehin nur das bisher Vorgesehene leisten kann – nämlich ein Fahrzeugcockpit zu *simulieren* – oder ist es vielleicht so allgemein gehalten, dass es für kein konkretes Gebiet mehr praktisch einsetzbar ist? Der Umstand, dass bisher immer nur die Anwendung als Autocockpit betrachtet wurde, ist lediglich ein Scheinargument für den ersten Fall, da dies im Wesentlichen nur Auswirkungen auf die Hintergrundgrafiken der Instrumente hat, und diese sind selbstverständlich austauschbar. Erst bei einem Wechsel in eine völlig andere Anwendungsdomäne, in der die GUI-Elemente nicht mehr als Instrumente aufgefasst werden können, müssten viele neue Klassen geschrieben werden. Andererseits, und dies spricht eher für den zweiten Fall, stand immer die Umsetzung der grundlegenden Entscheidung im Vordergrund, die GUI als Hyperdokument aufzufassen und dabei streng objektorientiert vorzugehen. Dies hat zum Ergebnis, dass die HyCop-Software sehr portabel und enorm konfigurierbar, aber weder hochperformant noch leicht verifizierbar ist. Vor einem Einsatz in einer speziellen Umgebung, beispielsweise einem realen Fahrzeug, wäre eine Reimplementierung durchzuführen, bei der die Software auf ein geeignetes *embedded system* zugeschnitten wird. Dabei müsste neben der Korrektheit der Programme auch immer die Korrektheit der Konfiguration nachgewiesen werden, um ein Fehlverhalten ausschließen zu können.

29.3 Ausblick

Im letzten Abschnitt wurde bereits die Notwendigkeit einer automatisierten Überprüfung der Konfiguration angesprochen. Während die syntaktische Korrektheit durch die Verwendung von XML-DTDs bereits sichergestellt wird, ist darüber hinaus eine semantische Verfikation im Sinne eines Model-Checkings eine wünschenswerte Erweiterung. Es wäre z.B. gut denkbar, die definierten Links und beteiligten Medienobjekte sowie Geräte in eine Kripke-Struktur abzubilden, so dass Aussagen über Zustände und Übergänge automatisch überprüfbar werden.

Bisher werden durch HyCop nur unidirektionale 1 : 1-Links unterstützt. Eine zukünftige Erweiterung zu bidirektionalen $n : n$-Links könnte die Konfiguration vereinfachen und flexibler machen, weil starke Zusammenhänge zwischen mehreren Medienobjekten kompakt notiert werden könnten.

Ein weiterer Ansatz für die Fortentwicklung von HyCop könnte die Ausimplentierung der derzeit nur als *blackboxes* vorhandenen Kommunikationsschnittstellen sein. Darunter fallen beispielsweise das Navigationssystem, Einkauf- und Bezahldialoge sowie der Pannendienstruf. Auch die Kommunikation über das Internet und das Beziehen von Unterhaltungsmedien könnten noch deutlich erweitert werden.

29.4 Ein Fazit aus Sicht der Betreuer

Autoren: *Klaus Alfert*
Alexander Fronk
Christof Veltmann

Aus Sicht der Betreuer ist die Arbeit mit der Projektgruppe HyCop angenehm gewesen, man kann mit Fug und Recht sagen, die Arbeit hat Spaß gemacht. Besonders hervorheben muss man die große Selbständigkeit der PG-Mitglieder, die sich insbesondere in der Gruppendynamik zeigt. Es war nur sehr selten notwendig als Betreuer eingreifen zu müssen, um die PG auf Kurs zu halten, da die PG mit hohen Engagement ihre Arbeit selbst geplant und gestaltet hat. Auch die unkomplizierte und selbstverständliche Aufnahme des jüngsten PG-Mitgliedes Jonas, der erst während der PG geboren wurde, und die flexible und effektiven Integration seiner Mutter, zeugt von diesem angenehmen Verhalten.

Die Realisierung des HyCop-Systems ist gelungen. Sie ist technisch durchdacht und überlegt realisiert worden und zeigt, dass der Perspektivwechsel von klassischer GUI-Entwicklung zur Modellierung von Hyperdokumenten auch technisch sinnvoll umsetzbar ist. Die Einhaltung der selbstgesetzten Deadlines während der Implementierung spricht dafür, dass die PG in der Lage ist, ihre Arbeitsaufwände und ihr Arbeitsvermögen gut einzuschätzen. Insofern war die Durchführung der Implementierung nach XP ein erfolgreiches Experiment.

Weniger glücklich war XP bezüglich der Dokumentation der Architektur und des Entwurfes, wobei dies systemimanent für diese Vorgehensweise ist. Um die im Rahmen einer PG notwendige Darstellung des Entwurfs trotzdem mit XP ohne all zu großen Aufwand realisieren zu können, bietet es sich an, zusätzliche Berichte in den Prozess zu integrieren. So sollten sowohl die Kunden als auch die Entwickler nach ihren Verhandlungsrunden das Verhandlungsergebnis

der Geschäftsführung berichten und dokumentieren. Die Kunden legen dar, welche Stories akzeptiert wurden und welche nicht, die Entwickler erläutern die Zerlegung der Stories in Tasks und stellen den damit verbunden Entwurf dar. In den wöchentlichen Sitzungen kann dann über die Evolution des Entwurfes berichtet werden, zum Release hin kann man zusätzlich den Originalentwurf und dessen endgültige Form gegenüberstellen. Auf diese Weise wird der Entwurf immer wieder in eher zwangloser Form expliziert und man kann beim Schreiben des Endberichtes darauf zurückgreifen. Eine nächste PG wird zeigen müssen, ob dieser angepasste XP-Prozess erfolgreich sein wird.

Literaturverzeichnis

[1] AMI-C. AMI-C Spec 1001-0-0: Architecture Specification - Release 1.
URL http://www.ami-c.org/specs/AMI-C_SPEC_1001.0.0_(Architecture).zip, 2001.
Zugegriffen am 10.07.2002.

[2] AMI-C. AMI-C Spec 1003-0-0: Use Cases - Release 1.
URL http://www.ami-c.org/specs/AMI-C_SPEC_1003.0.0_(AMI-C_Use_Cases).zip, 2001.
Zugegriffen am 10.07.2002.

[3] AMI-C. AMI-C Spec 3001-0-0: Vehicle Interface Specification - Release 1.
URL http://www.ami-c.org/specs/AMI-C_SPEC_3001.0.0_(Vehicle_Interface).zip, 2001.
Zugegriffen am 10.07.2002.

[4] AMI-C. AMI-C Spec 3002-0-0: Common Message Set Specification - Release 1.
URL http://www.ami-c.org/specs/AMI-C_SPEC_3002.0.0_(Common_Message_Set).zip, 2001.
Zugegriffen am 10.07.2002.

[5] Ivar Jacobson Grady Booch James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1998.

[6] Martin Lippert Henning Wolf Stefan Rook. *Software entwickeln mit eXtreme Programming*. dpunkt-Verlag, 2002.

[7] Trolltech AS. Qt/Embedded - The C++ Embedded GUI Application Developer's Toolskit Technical Overview, 2001.

[8] Baader and F. und T. Nipkow. Term Rewriting and All That. *Cambridge University Press*, 1998.

[9] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, paperback edition, 1996.

[10] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[11] Kent Beck and Erich Gamma. JUnit.
URL http://www.junit.org.
Zugegriffen am 13.12.2002.

[12] Kent Beck and Erich Gamma. Junit test infected: Programmers love writing tests.
URL http://junit.sourceforge.net/doc/testinfected/testing.htm.
Zugegriffen am 14.01.2003.

[13] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000.
URL http://www.w3c.org/TR/2000/REC-xml-20001006, 2000.
Zugegriffen am 05.04.2002.

[14] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. XML Schema: Formal Description, W3C Working Draft, 20 March 2001.
URL http://www.w3c.org/TR/2001/WD-xmlschema-formal-20010320, 2001.
Zugegriffen am 05.04.2002.

[15] J.F. Burton. Evaluating HyTime: An Examination and Implementation Experience. *Hypertext '96: The seventh ACM conference on Hypertext, S. 105-115*, 1996.

[16] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Patternorientierte Softwarearchitektur. Ein Pattern-System.* Addison Wesley, München, 1. korr nachdruck edition, 2000.

[17] Mary Campione and Kathy Walrath. The Java Tutorial - A Visual Index of the Swing Components.
URL http://java.sun.com/docs/books/tutorial/uiswing/components/components.html.
Zugegriffen am 30.04.2002.

[18] L.A. Carr, W. Barron, H.C. Davis, and W. Hall. Evaluating HyTime: An Examination and Implementation Experience. *Electronic Publishing 7 (3), S. 163-178*, 1994.

[19] Dan Connolly. Web Naming and Addressing Overview (URIs, URLs, ...).
URL http://www.w3c.org/Addressing, 2001.
Zugegriffen am 05.04.2002.

[20] D. D. Cowan and C. J. P. Lucena. Abstract Data Views, an interface specification concept to enhance design for reuse. *IEEE Transactions on Software Engineering*, 21(No. 3), March 1995.

[21] DaimlerChrysler. DaimlerChrysler IT Cruiser Telematics Concept, 2001.

[22] James Clarkand Steve DeRose. XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999.
URL http://www.w3c.org/TR/1999/REC-xpath-19991116, 1999.
Zugegriffen am 05.04.2002.

[23] Steve DeRose, Eve Maler, and Ron Daniel Jr. XML Pointer Language (XPointer) Version 1.0, W3C Candiate Recommendation 11 September 2001.
URL http://www.w3c.org/TR/2001/CR-xptr-20010911, 2001.
Zugegriffen am 05.04.2002.

[24] Steve DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) Version 1.0, W3C Recommendation 27 June 2001.

URL http://www.w3c.org/TR/2001/REC-xlink-20010627, 2000.
Zugegriffen am 05.04.2002.

[25] Alden DeSoto. *Using the Beans Development Kit 1.0.* Sun Microsystems, 1997.

[26] Jutta Eckstein. eXtreme Programming - Ein leichtgewichtiger Software-Entwicklungsprozess.
URL http://ourworld.compuserve.com/homepages/jutta_eckstein/, 2000.
Zugegriffen am 07.11.2002.

[27] Robert Eckstein, Marc Loay, and Dave Wood. *Java Swing.* O´Reilly & Associates, Inc., Sebastapol, CA, first edition edition, December 1998.

[28] Graham Hamilton (Editor). *JavaBeans.* Sun Microsystems, 1997.

[29] embedded com.
URL http://www.embed.com.cn/. Zugegriffen am 01.04.2002.

[30] Berin Loritsch et al. Developing with Apache Avalon.
URL http://jakarta.apache.org/avalon/developing/index.html, 2001.
Zugegriffen am 08.04.2003.

[31] Martin Fowler. *Refactoring - Improving the Desing of Existing Code.* Object Technologie Series. Addison Wesley, 2000.

[32] Martin Fowler and Matthew Foemmel. Continuous integration.
URL http://www.martinfowler.com/articles/continuousIntegration.html.
Zugegriffen am 14.01.2003.

[33] Martin Fowler and Kendall Scott. *UML konzentriert.* Addison-Wesley, 2000.

[34] Alexander Fronk. *Algebraische Semantik einer objektorientierten Sprache zur Spezifikation von Hyperdokumenten.* Shaker Verlag, 2001.

[35] Erich Gamma and Kent Beck. JUnit - A Cook's Tour.
URL http://junit.sourceforge.net/doc/cookstour/cookstour.htm.
Zugegriffen am 13.12.2002.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of reuseable Object-Oriented Software.* Addison Wesley, first edition, December 1994.

[37] Franca Garzotto, Paolo Paolini, and Daniel Schwabe. HDM - A Model-Based Approach to Hypertext Application Design. *ACM Transitions on Information Systems*, 11(No. 1):1–26, January 1993.

[38] Kaj Grønbæk and Randall H. Trigg. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2):30–39, Februar 1994.

[39] Frank Halasz and Mayer Schwartz. The Dexter Hypertext Reference Model. *Proceedings of the Hypertext Workshop, NIST Special Publication*, 500-178:95–133, März 1990. National Institute of Standards and Technology, Gaithersburgh, Md,USA, Jan. 16-18 1990.

[40] Elliotte Rusty Harold. *JavaBeans*. IDG Books Worldwide, 1998.

[41] Andreas Heilwagen. JUnitX.
URL http://www.extreme-java.de/junitx/.
Zugegriffen am 13.12.2002.

[42] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 2 Core Specification Version 1.0, W3C Recommendation 13 November, 2000.
URL http://www.w3c.org/TR/2000/REC-DOM-Level-2-Core-20001113, 2000.
Zugegriffen am 05.04.2002.

[43] IBM.
URL http://www.embedded.oti.com.
Zugegriffen am 01.04.2002.

[44] Sun Microsystems Inc. Automotive Infotainment and Java TM Technology: Blueprint for the Future, 2001.

[45] Ron Jeffries. The king's dinner.
URL http://www.xprogramming.com/xpmag/kings_dinner.htm, 1999.
Zugegriffen am 07.11.2002.

[46] W. Kim. Advanced Database systems. *ACM Press*, 1994.

[47] Rob Kremer. Z Specification of the Dexter Hypertext Reference Model.
URL http://pages.cpsc.ucalgary.ca/ kremer/dexter/index.html, 2002.
Zugegriffen am 18.07.2002.

[48] M. Langerhuizen and W. Kraaijvanger. Linking in HyTime, Second Edition.
URL http://hagen.let.rug.nl/hypertext/hytime2.html, 1998.
Zugegriffen am 10.05.2002.

[49] Embedded Linux.
URL http://www.embeddedlinux.com.
Zugegriffen am 01.04.2002.

[50] Xiao Haiqiao Lü Jinjian.
URL http://www.bol-system.com/MCU/990609-1.htm.
Zugegriffen am 01.04.2002.

[51] Xiao Haiqiao Lü Jinjian.
URL http://www.bol-system.com/MCU/990609-2.htm.
Zugegriffen am 01.04.2002.

[52] Peter Marwedel. *Introduction to Embedded Systems*. Universität Dortmund, 2001.

[53] Brett McLaughlin. *Java und XML*. O'Reilly Verlag, 2000.

[54] David Megginson. SAX, official website for SAX.
URL http://www.saxproject.org, 2000.
Zugegriffen am 05.04.2002.

[55] Microsoft.
URL http://www.microsoft.com/automotive/default.htm.
Zugegriffen am 01.04.2002.

[56] Microsoft. Microsoft Windows CE for Automotive 3.5 - Technology Primer, 2001.

[57] S.R. Newcomb, N.A. Kipp, and V.T. Newcomb. The 'HyTime" Hypermedia/Time-Based Document Structuring Language. *Communications of the ACM 34 (11), S. 67-83*, 1991.

[58] Jakob Nielsen. *Usability Engineering.* Academic Press, 1993.

[59] Jakob Nielsen. *Multimedia and Hypertext.* Academic Press, 1995.

[60] Jakob Nielsen. *Designing Web Usability.* New Riders Publ., 2000.

[61] Jakob Nielsen. Heuristic Evaluation.
URL http://www.useit.com/papers/heuristic, 2002.
Zugegriffen am 20.03.2002.

[62] Jakob Nielsen. Heuristic Evaluation - How to Conduct a Heuristic Evaluation.
URL http://www.useit.com/papers/heuristic/heuristic_evaluation.html, 2002.
Zugegriffen am 20.03.2002.

[63] Jakob Nielsen. Heuristic Evaluation - Severity Ratings for Usability Problems.
URL http://www.useit.com/papers/heuristic/severityrating.html, 2002.
Zugegriffen am 20.03.2002.

[64] Jakob Nielsen. Heuristic Evaluation - Ten Usability Heuristics.
URL http://www.useit.com/papers/heuristic/heuristic_list.html, 2002.
Zugegriffen am 20.03.2002.

[65] Jakob Nielsen. Heuristic Evaluation - Usability Problems found by Heuristic Evaluation.
URL http://www.useit.com/papers/heuristic/usability_problems.html, 2002.
Zugegriffen am 20.03.2002.

[66] C. Oberscheid. SGML, HyTime und DSSSL.
URL http://www.cg.cs.tu-bs.de/oberscheid, 1996.
Zugegriffen am 10.05.2002.

[67] Claudia Piemont. *Komponenten in Java.* dpunkt.verlag, 1999.

[68] QNX.
URL http://www.qnx.com/products/realtimeplatform/index.html.
Zugegriffen am 01.04.2002.

[69] Martin Riegel. Beurteilung multimedialer Anwendungen und Systeme. *Berichte des German Chapter of the ACM, 49, Software-Ergonomie*, 1997.

[70] Jeffrey Rubin. *Handbook of Usability Testing.* Jahn Wiley & Sons, 1994.

[71] Daniel Schwabe and Gustavo Rossi. Building Hypermedia Applications as Navigational Views of Information Models. In *Proc. of the 28th. Hawaii International Conference on System Sciences*, volume 3, pages 231–240, Januar 1995.

[72] Daniel Schwabe and Gustavo Rossi. The Object-Oriented Hypermedia Design Model. *Communications of the ACM*, 98(No. 8), August 1995.

[73] Daniel Schwabe and Gustavo Rossi. *An Object Oriented Approach to Web-Based Application Design*, chapter 4. Wiley and Sons, New York, 1998.

[74] Daniel Schwabe and Gustavo Rossi. Developing hypermedia applications using OOHDM. *Workshop on Hypermedia Development Processes, Methods and Models, Hypertext '98, Pittsburgh, USA*, 1998.

[75] P. Seyer. *Understanding hypertext: concepts and applications.* Windcrest/MacGraw-Hill, 1991.

[76] MontVista Software. Embedded Linux and Java TM Technology - Made for Each Other, 2001.

[77] C.M. Sperberg-McQueen and L. Burnard. A Gentle Introduction to SGML.
URL http://www.uic.edu/orgs/tei/sgml/teip3sg/index.html, 1995.
Zugegriffen am 10.05.2002.

[78] Trolltech.
URL http://www.trolltech.com/products/qt/examples.html.
Zugegriffen am 01.04.2002.

[79] Trolltech.
URL http://www.trolltech.com/products/Embedded/index.html.
Zugegriffen am 01.04.2002.

[80] Timothy Wall. Abbot.
URL http://abbot.sourceforge.net.
Zugegriffen am 13.12.2002.

[81] Frank Westphal.
URL http://www.frankwestphal.de.
Zugegriffen am 07.11.2002.

[82] Windriver.
URL http://www.windriver.com/products/html/vxwks5x.html.
Zugegriffen am 01.04.2002.

/99/ T. Bühren, M. Cakir, E. Can, A. Dombrowski, G. Geist, V. Gruhn, M. Gürgrn, S. Handschumacher, M. Heller, C. Lüer, D. Peters, G. Vollmer, U. Wellen, J. von Werne
Endbericht der Projektgruppe eCCo (PG 315)
Electronic Commerce in der Versicherungsbranche
Beispielhafte Unterstützung verteilter Geschäftsprozesse
Februar 1999

/100/ A. Fronk, J. Pleumann,
Der DoDL-Compiler
August 1999

/101/ K. Alfert, E.-E. Doberkat, C. Kopka
Towards Constructing a Flexible Multimedia Environment for Teaching the History of Art
September 1999

/102/ E.-E. Doberkat
An Note on a Categorial Semantics for ER-Models
November 1999

/103/ Christoph Begall, Matthias Dorka, Adil Kassabi, Wilhelm Leibel, Sebastian Linz, Sascha Lüdecke, Andreas Schröder, Jens Schröder, Sebastian Schütte, Thomas Sparenberg, Christian Stücke, Martin Uebing, Klaus Alfert, Alexander Fronk, Ernst-Erich Doberkat
Abschlußbericht der Projektgruppe PG-HEU (326)
Oktober 1999

/104/ Corina Kopka
Ein Vorgehensmodell für die Entwicklung multimedialer Lernsysteme
März 2000

/105/ Stefan Austen, Wahid Bashirazad, Matthais Book, Traugott Dittmann, Bernhard Flechtker, Hassan Ghane, Stefan Göbel, Chris Haase, Christian Leifkes, Martin Mocker, Stefan Puls, Carsten Seidel, Volker Gruhn, Lothar Schöpe, Ursula Wellen
Zwischenbericht der Projektgruppe IPSI
April 2000

/106/ Ernst-Erich Doberkat
Die Hofzwerge — Ein kurzes Tutorium zur objektorientierten Modellierung
September 2000

/107/ Leonid Abelev, Carsten Brockmann, Pedro Calado, Michael Damatow, Michael Heinrichs, Oliver Kowalke, Daniel Link, Holger Lümkemann, Thorsten Niedzwetzki, Martin Otten, Michael Rittinghaus, Gerrit Rothmaier
Volker Gruhn, Ursula Wellen
Zwischenbericht der Projektgruppe Palermo
November 2000

/108/ Stefan Austen, Wahid Bashirazad, Matthais Book, Traugott Dittmann, Bernhard Flechtker, Hassan Ghane, Stefan Göbel, Chris Haase, Christian Leifkes, Martin Mocker, Stefan Puls, Carsten Seidel, Volker Gruhn, Lothar Schöpe, Ursula Wellen
Endbericht der Projektgruppe IPSI
Februar 2001

/109/ Leonid Abelev, Carsten Brockmann, Pedro Calado, Michael Damatow, Michael Heinrichs, Oliver Kowalke, Daniel Link, Holger Lümkemann, Thorsten Niedzwetzki, Martin Otten, Michael Rittinghaus, Gerrit Rothmaier
Volker Gruhn, Ursula Wellen
Zwischenbericht der Projektgruppe Palermo
Februar 2001

/110/ Eugenio G. Omodeo, Ernst-Erich Doberkat
Algebraic semantics of ER-models from the standpoint of map calculus.
Part I: Static view
März 2001

/111/ Ernst-Erich Doberkat
An Architecture for a System of Mobile Agents
März 2001

/112/ Corina Kopka, Ursula Wellen
Development of a Software Production Process Model for Multimedia CAL Systems by Applying Process Landscaping
April 2001

/113/ Ernst-Erich Doberkat
The Converse of a Probabilistic Relation
Oktober 2002

/114/ Ernst-Erich Doberkat, Eugenio G. Omodeo
Algebraic semantics of ER-models in the context of the calculus of relations.
Part II: Dynamic view
Juli 2001

/115/ Volker Gruhn, Lothar Schöpe (Eds.)
Unterstützung von verteilten Softwareentwicklungsprozessen durch integrierte Planungs-, Workflow- und Groupware-Ansätze
September 2001

/116/ Ernst-Erich Doberkat
The Demonic Product of Probabilistic Relations
September 2001

/117/ Klaus Alfert, Alexander Fronk, Frank Engelen
Experiences in 3-Dimensional Visualization of Java Class Relations
September 2001

/118/ Ernst-Erich Doberkat
The Hierarchical Refinement of Probabilistic Relations
November 2001

/119/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer, Ingo Röpling,
Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Intermediate Report
November 2001

/120/ Volker Gruhn, Ursula Wellen
Autonomies in a Software Process Landscape
Januar 2002

/121/ Ernst-Erich Doberkat, Gregor Engels (Hrsg.)
Ergebnisbericht des Jahres 2001
des Projektes “MuSofT – Multimedia in der SoftwareTechnik”
Februrar 2002

/122/ Ernst-Erich Doberkat, Gregor Engels, Jan Hendrik Hausmann, Mark Lohmann, Christof Veltmann
Anforderungen an eine eLearning-Plattform — Innovation und Integration —
April 2002

/123/ Ernst-Erich Doberkat
Pipes and Filters: Modelling a Software Architecture Through Relations
Juni 2002

/124/ Volker Gruhn, Lothar Schöpe
Integration von Legacy-Systemen mit Eletronic Commerce Anwendungen
Juni 2002

/125/ Ernst-Erich Doberkat
A Remark on A. Edalat’s Paper *Semi-Pullbacks and Bisimulations in Categories of Markov-Processes*
Juli 2002

/126/ Alexander Fronk
Towards the algebraic analysis of hyperlink structures
August 2002

/127/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer
Ingo Röpling, Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Final Report
August 2002

/128/ Timo Albert, Zahir Amiri, Dino Hasanbegovic, Narcisse Kemogne Kamdem,
Christian Kotthoff, Dennis Müller, Matthias Niggemeier, Andre Pavlenko, Stefan Pinschke,
Alireza Salemi, Bastian Schlich, Alexander Schmitz
Volker Gruhn, Lothar Schöpe, Ursula Wellen
Zwischenbericht der Projektgruppe Com42Bill (PG 411)
September 2002

/129/ Alexander Fronk
An Approach to Algebraic Semantics of Object-Oriented Languages
Oktober 2002

/130/ Ernst-Erich Doberkat
Semi-Pullbacks and Bisimulations in Categories of Stochastic Relations
November 2002

/131 Yalda Ariana, Oliver Effner, Marcel Gleis, Martin Krzysiak,
Jens Lauert, Thomas Louis, Carsten Röttgers, Kai Schwaighofer,
Martin Testrot, Uwe Ulrich, Xingguang Yuan
Prof. Dr. Volker Gruhn, Sami Beydeda
Endbericht der PG nightshift:
Dokumentation der verteilten Geschäftsprozesse im FBI und Umsetzung von Teilen dieser Prozesse im Rahmen eines FBI-Intranets basierend auf WAP- und Java-Technologie
Februar 2003

/132/ Ernst-Erich Doberkat, Eugenio G. Omodeo
ER Modelling from First Relational Principles
Februar 2003

/133/ Klaus Alfert, Ernst-Erich Doberkat, Gregor Engels (Hrsg.)
Ergebnisbericht des Jahres 2002 des Projektes “MuSofT – Multimedia in der SoftwareTechnik”
März 2003

/134/ Ernst-Erich Doberkat
Tracing Relations Probabilistically
März 2003

/135/ Timo Albert, Zahir Amiri, Dino Hasanbegovic, Narcisse Kemogne Kamdem,
Christian Kotthoff, Dennis Müller, Matthias Niggemeier,
Andre Pavlenko, Alireza Salemi,Bastian Schlich, Alexander Schmitz,
Volker Gruhn, Lothar Schöpe, Ursula Wellen
Endbericht der Projektgruppe Com42Bill (PG 411)
März 2003

/136/ Klaus Alfert
Vitruv: Specifying Temporal Aspects of Multimedia Presentations —
A Transformational Approach based on Intervals
April 2003

/137/ Klaus Alfert, Jörg Pleumann, Jens Schröder
A Framework for Lightweight Object-Oriented Design Tools
April 2003

/138/ K. Alfert, A. Fronk , Ch. Veltmann (Hrsg.)
Stefan Borggraefe, Leonore Brinker, Evgenij Golkov, Rafael Hosenberg, Bastian Krol, Daniel Mölle,
Markus Niehammer, Ulf Schellbach, Oliver Szymanski, Tobias Wolf, Yue Zhang
Endbericht der Projektgruppe 415: Konzeption und Implementierung eines digitalen und hypermedialen Automobilcockpits (HyCop)
Mai 2003