Unified Modeling Language
Rational Rose

Sergej Schwenk
Oktober 1999

Inhaltsverzeichnis

0. Motivation

Die Entwicklung echtzeitfähiger Software in Fahrzeuganwendungen ist im Moment noch über weite Strecken durch funktionsorientierte Denkansätze geprägt. Die steigende Komplexität dieser Systeme zwingt zu einem Umdenken bei der Auswahl von Entwicklungsmethoden. In den letzten paar Jahren hat sich die objektorientierte Technologie in vielen Bereichen der Informatik als Methode durchgesetzt, die Komplexität vieler verschiedener Systeme zu überwinden. In diesem Beitrag erfolgt eine einführende Darstellung von Unified Modeling Language (**UML** [1]) und **Rational Rose** [2], die in der letzten Zeit immer häufiger bei der objektorientierten Softwareentwicklung eingesetzt werden.

1. Historie

In den 80er Jahren verließen Objekte die Forschungslaboratorien und bewährten sich auch in der Praxis. Smalltalk stabilisierte sich zu einer von Entwicklern benutzbaren Plattform, und C++ wurde entworfen. Wie viele Entwicklungen im Softwarebereich wurde auch die Objektidee zuerst durch Programmiersprachen verbreitet.

In den späten 80er und frühen 90er Jahren waren immer mehr Leute der Ansicht, daß frühere sehr populäre Techniken der industriellen Softwareentwicklung, die den Entwicklern dabei halfen, Analyse und Entwurf gut zu bewerkstelligen, natürlich genauso wichtig in der objektorientierten Entwicklung sind. Es kam zu einer Welle von objektorientierten Analyse- und Entwurfsmethoden , die regelrecht die Methodenszene überfluteten. Die Abbildung 1 zeigt nur einen kleinen Ausschnitt der Entwicklung einiger auf dem Markt verfügbaren Methoden in den letzten Jahren.

[1] eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme.

[2] Tool für objektorientierte Analyse und Design (OOA / OOD). Unterstützt unter anderem UML Notation, Codegenerierung (Java, C++, Ada), Reverse Engineering.

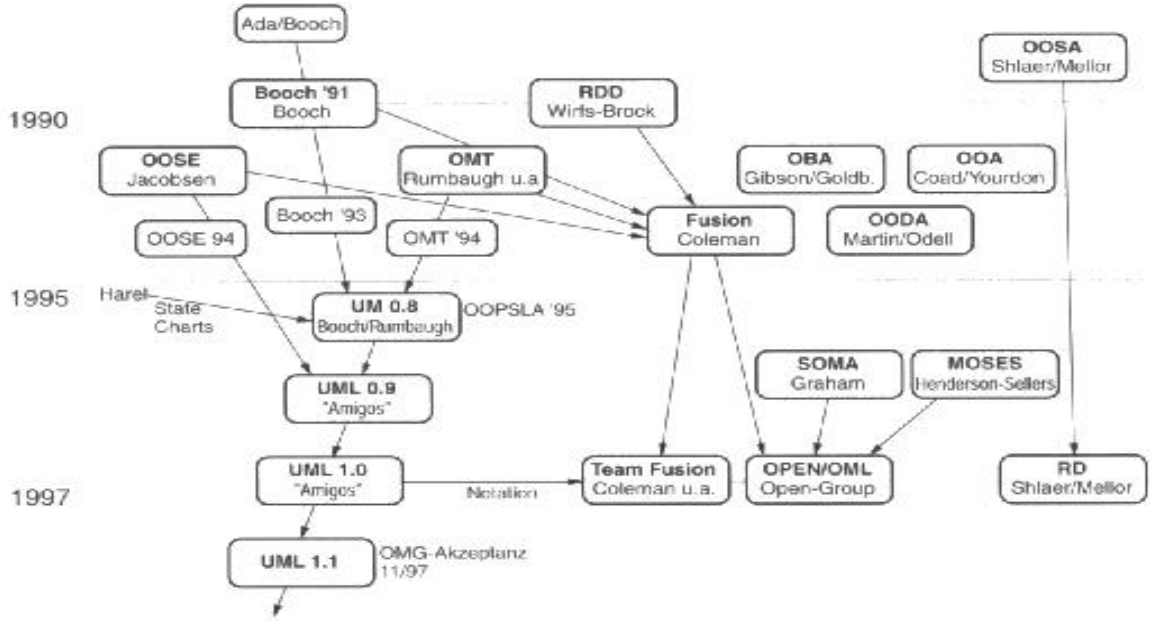

Abbildung 1 Methodenszene 1990-1997 [Oestereich98]

Die Methodenszene war sehr vielfältig und die einzelnen Methoden standen in harter Konkurrenz untereinander. Einige Standardisierungsversuche verliefen zunächst erfolglos. 1994 wechselte Jim Rumbaugh von General Electric zu Rational Software, und entwickelte zusammen mit Grady Booch eine zusammengeführte Methode der beiden. Diese wurde bei OOPSLA 95 als Version 0.8 UM (Unified Method) der Öffentlichkeit präsentiert. Grady Booch und Jim Rumbaugh proklamierten „the methods war is over – we won“. Damit deklarierten sie eigentlich, daß sie die Standardisierung auf die Microsoft – Art erreichen wollen. Noch im selben Jahr kaufte Rational Software die Firma Objectory, und Ivar Jacobson bereicherte das Unified Team. Die weiterhin als **„drei Amigos“** bezeichnete Gruppe arbeitete weiter an ihrer Methode. Diese wurde dann 1996 unter dem neuen Namen **Unified Modeling Language (UML)** in der Version 0.9 präsentiert. Später folgte auch die Version 1.0 von UML. Im Jahre 1997 wurden von **OMG**[1] (Object Managment Group) verschiedene Methodenvorschläge eingesammelt, um letztendlich nur eine Methode als Standard anzuerkennen.

Die UML in der Version 1.1 wurde von der OMG als Standard akzeptiert, alle anderen Gegenvorschläge wurden zurückgezogen. Die Weiterentwicklung der UML wird derzeit durch die OMG betrieben.

[1] Eine Gruppe aus 11 Unternehmen, die im April 1989 gegründet wurde. Als "non - for - profit corporation" entwickelt OMG die unabhängige Spezifikationen für die Softwareindustrie. Hauptziel des Konsortiums ist die Angleichung sämtlicher weltweit verbreiteter Spezifikationen. Aktuell sind rund 800 Mitarbeiter bei OMG beschäftigt.

2. Der Entwicklungsprozeß

Prozesse dienen im allgemeinen der Herstellung von Dingen, hier speziell von Softwaresystemen. Ein Prozeß beschreibt in etwa die Vorgehensweise bei der Entwicklung eines Softwaresystems. Dafür gibt es verschiedene Modelle, die sich in der Praxis mehr oder weniger gut bewährt haben. Einen Prozeß zu entwickeln, der auf alle realen Prozesse paßt, ist unmöglich. Deshalb wurde der Entwicklungsprozeß als solcher aus der UML ausgeklammert[1]. UML ist unabhängig vom Prozeß und kann mit jeden Prozeß benutzt werden. Dazu müssen jedoch alle Vorgehensweisen und Notationen von konkreten Prozessen mit der UML Notation darstellbar sein. Die UML - Notation stellt nur ein Rahmenwerk (Umgebung) zur Verfügung, mit der sämtliche Prozesse beschrieben werden können. Grundlage für dieses Prozeßrahmenwerk bildet die Beschreibung der OOSE – Methode von Jacobson und der darin beschriebene Prozeß „Objectory“. Das Prozeßrahmenwerk (engl. **process framework**) der UML wird auch **Rational -Objectory-Process** genannt. Es schließt die gemeinsamen Elemente der Prozesse von OOSE , BOOCH und OMT Methoden ein, läßt aber auch genügend Spielraum, um die für das konkrete Projekt geeigneten Techniken einzusetzen.

In diesem Vortrag greife ich auf die Darstellung des Entwicklungsprozesses in [Fowler98] zurück. Diese Darstellung des Prozesses stellt eine etwas vereinfachte Form des Objectory-Prozesses der Firma Rational [Rational] dar. Dieser Prozeß ist ein **iterativer und inkrementeller** Entwicklungsprozeß, in dem die Software nicht in einem großen Schlag am Ende des Projekts herausgegeben wird, sondern in Teilen entwickelt und freigegeben wird.

Die Abbildung 3 zeigt die Phasen des Prozesses.

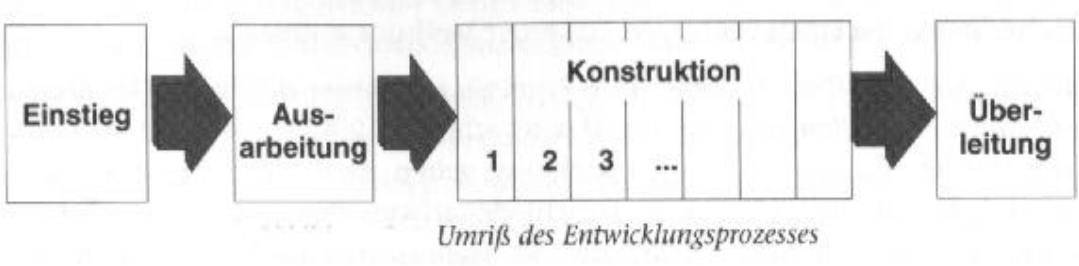

Abbildung 2 Der Entwicklungsprozeß [Fowler98] , S.28

- Die erste Phase stellt der **Einstieg (engl. inception)** dar. Während dieser Phase begründet man die geschäftlichen Absichten für das Projekt und entscheidet über den vom Projekt abgedeckten Bereich. Hier erhält man auch die Bestätigung des finanziellen Trägers des

[1] UML wird als Modellierungssprache bezeichnet, die überwiegend aus einer grafischen Notation besteht. Im Gegensatz dazu existieren in der objektorientierten Technologie auch **die Methoden**. Methoden enthalten eine Modellierungssprache und einen beschriebenen Entwicklungsprozeß.

Projektes, weiter voranschreiten zu können. Es wird dabei also unter anderem abgeschätzt, wie viel das Projekt kosten und andererseits erbringen wird.

- Während der **Ausarbeitung (engl. elaboration)** sammelt man detailliertere Anforderungen, führt Analyse und Entwurf auf hoher Ebene durch, um eine grundlegende Architektur einzuführen, und erstellt einen Plan für die Konstruktion. Erfahrungsgemäß beansprucht diese Phase etwa 20 Prozent der gesamten Projektdauer.
  Am Anfang dieser Phase werden jedoch zunächst Risiken identifiziert und zu deren Begrenzung Lösungsansätze entwickelt. Man unterscheidet nach [Fowler98] folgende Risikokategorien:
  - **Anforderungsrisiken** (Anforderungen müssen sorgfältig erfaßt werden, da sonst die Gefahr ein falsches System zu bauen besteht)
  - **Technologische Risiken** (u.a. ungeeigneter Einsatz neuer Technologien, Nichtkompatibilität eingesetzter Einzeltechnologien, Lücken in der Toolkette zur Unterstützung der Entwicklung etc.)
  - **Kompetenzrisiken** (Unerfahrenheit der Entwicklungsmannschaft)

  Ein wichtiges Ergebnis der Ausarbeitung ist eine **grundlegende Architektur (engl. baseline architecture)** für das System.
  Diese Architektur besteht aus
  - einer Liste von Anwendungsfällen, die die Anforderungen wiedergeben
  - das Problembereichmodell, stellt so zu sagen eine Beschreibung des letztendlich zu implementierenden Systems dar (z.B. die Krankenhausstruktur bei Softwaresystemen für die Patienetenverwaltung). Dient als Ausgangspunkt für die zentralen Klassen des Problembereichs.
  - die technologische Plattform, die die wichtigen Teile der Implementierungsbasis und ihre Verknüpfung beschreibt.

  Am Ende der Ausarbeitungsphase entsteht ein **Plan**, der eine Folge von Iterationen für die Konstruktion festlegt und den einzelnen Iterationen Anwendungsfälle zuordnet.
- Auf Basis der Anforderungen und des Architekturentwurfs wird in der **Konstruktion (engl. construction)** die Softwarearchitektur schrittweise (iterativ und inkrementell) verfeinert und implementiert. Jede Iteration der Konstruktionsphase ist dabei ein eigenes Miniprojekt. Man macht Analyse, Entwurf, Programmierung, Testen und Integration für jeden Anwendungsfall durch, der der Iteration zugewiesen ist.
- Die abschließende **Überleitungsphase** (**engl. transition phase**) enthält die Betatests, Verbesserung der Performanz und Benutzertraining.

3. UML

Die Unified Modeling Language (UML) ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme. Die UML berücksichtigt die gestiegenen Anforderungen bezüglich der Komplexität heutiger Systeme, deckt ein breites Spektrum von Anwendungsgebieten ab und eignet sich für konkurrierende, verteilte, zeitkritische eingebettete Systeme uvm. Die UML ist heute in ihrer Fassung 1.3 zum Standard von der OMG (Object Managment Group) [OMG] erklärt worden. Mein Vortrag bezieht sich deswegen hauptsächlich auf diese Version von UML.

Die UML enthält eine Vielzahl von Modellelementen und – Details. Um den Einstieg ins Thema zu erleichtern, werden spezielle sowie in der Praxis weniger bedeutende Elemente der UML hier nicht behandelt. Insbesondere werden die Aspekte des Metamodells nicht thematisiert.

Unified Modeling Language unterstützt die Softwareentwicklung durch eine Vielzahl von Notationselementen und Diagrammen , die ein System durch unterschiedliche Sichten beschreiben. Diese werden nun schrittweise näher erläutert. Zum besseren Verständnis wird durchgängig ein Beispiel **[DSR99]** vorgestellt und in jedem Abschnitt erweitert. In dem Beispiel handelt es sich um die Entwicklung eines digitalen Aufnahmegerätes für Sprachnachrichten (digital sound recorder).

Es werden folgende Anforderungen an das Gerät gestellt:

- Die Aufnahme und das Abspielen der Nachrichten soll unterstützt werden
- Nachrichten werden über eingebautes Mikrophon aufgenommen, und im digitalen Speicher abgelegt. Das Abspielen erfolgt über den eingebauten Lautsprecher.
- Es sollen bis zu 10 Nachrichten in unterschiedlichen „Slots“ im Speicher abgelegt werden können. Die Länge der Nachrichten ist durch den verfügbaren Speicherplatz begrenzt. Jede Nachricht kann abgespielt oder gelöscht werden.
- Es soll klein, tragbar, einfach zu benutzen und Batteriebetrieben sein.

Besondere Eigenschaften

- Einfache Handhabung durch on screen Menü
- Alarm Uhr mit dem Jahr 2000 fähigen Kalender
- LCD Display zur Darstellung der aktuellen Zeit, dem Datum, und der Menüführung
- Batterie Indikator und Stand By Modus
- Akzeptable Soundqualität (hier 6kHz mit 8 bit per sample)

Die Abbildung 3 zeigt eine mögliche Ausführung des Gerätes

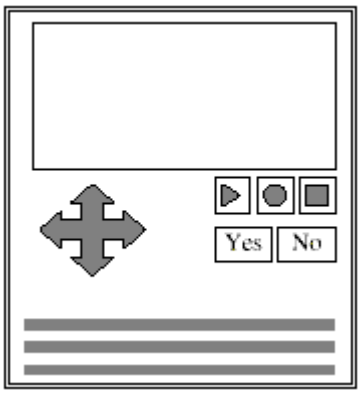

Abbildung 3 Frontansicht des digitalen Aufnahmegerätes

3.1 Anwendungsfalldiagramme

Bevor wir uns mit dem eigentlichen Diagramm beschäftigen, sollte zunächst der Begriff eines Anwendungsfalls geklärt werden. In der Essenz ist **ein Anwendungsfall (engl. use case)** eine **typische Interaktion** zwischen einem Benutzer und einem Computersystem (z.B. das Ausfüllen einer Eingabemaske eines Dialogfeldes ist ein Anwendungsfall). Typischerweise werden die Anwendungsfälle mittels einiger Diskussionen mit den späteren Anwendern und / oder mit den Experten ermittelt. Ein Anwendungsfall wird stets durch einen **Akteur[1] (engl. actor)** initiiert. Die **wichtigste Eigenschaft** der Akteure in der Interaktion ist eine **definierte Rolle[2]**. Beim Umgang mit den Akteuren ist es wichtig in Rollen zu denken, statt z.B. an Menschen, die im Anwendungsfall die Rolle übernehmen. Ein einzelner Akteur kann viele Anwendungsfälle ausführen, und ein Anwendungsfall kann auch von verschiedenen Akteuren ausgeführt werden. Man sollte sich im klaren sein, daß ein Akteur keineswegs aus Fleisch und Blut sein muß. Ein Akteur kann auch ein externes System sein, daß Informationen vom betrachteten System benötigt. Die Beziehungen zwischen den Akteuren und Anwendungsfällen werden durch A**nwendungsfalldiagramme (engl. Use – Case – Diagram)** aufgezeigt.

Ein Anwendungsfall wird durch eine Ellipse und ein Akteur durch ein Strichmännchen dargestellt. Als Verbindung zwischen einem Akteur und einem Anwendungsfall wird eine einfache Linie[3] benutzt. Weiterhin unterscheidet man noch drei Arten von Beziehungen zwischen Anwendungsfällen:

- Mit der **<<include>> Beziehung** (ersetzt die **<<uses>>** Beziehung aus UML 1.1) läßt sich darstellen, daß innerhalb eines Anwendungsfalles ein anderer Anwendungsfall

[1] eine außerhalb des Systems liegende Klasse, die an der in einem Anwendungsfall beschriebenen Interaktion mit dem System beteiligt ist.

[2] Hilfsfrage: (Als was) oder (als wer) ist ein Akteur an dem Anwendungsfall beteiligt ?

[3] Die Interaktion ist dann in beide Richtungen erlaubt. Zur Einschränkung kann der Richtungspfeil verwendet werden.

vorkommt. Man erspart sich das Kopieren des gleichen Verhaltens in den verschiedenen Anwendungsfällen (Redundanz wird vermieden)

- Mit der **<<extend>> Beziehung** hingegen läßt sich ausdrücken, daß ein Anwendungsfall unter bestimmten Umständen bzw. an einer bestimmter Stelle (engl. extention Point) durch einen anderen erweitert wird. Bei der UML Notation 1.3 zeigt die Erweiterung eines Anwendungsfalls auf den zu erweiternden Anwendungsfall. Um die Sache noch übersichtlicher zu machen, wird zusätzlich der **extension point** an der Stelle angegeben, wo der erweiternde Fall eingebunden werden soll.
- Mit der **Generalisierung** können Unter–Anwendungsfälle von den Ober-Anwendungsfällen Verhalten und Bedeutung erben, analog zur Generalisierung zwischen den Klassen.

Die Abbildung 4 zeigt die verwendete Notation

Anwendungsfalldiagramm

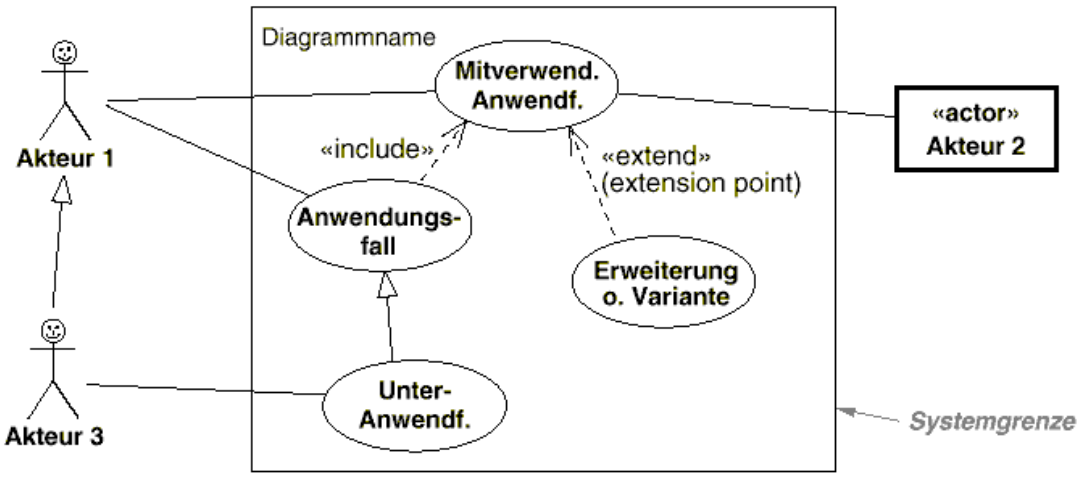

Abbildung 4 Elemente der Anwendungsfalldiagramme

Abschließend sollte vermerkt werden, daß die Stereotypen <<include>> und <<extend>> nützliche aber auch entbehrliche Modellkonstrukte der UML darstellen, die häufig dazu verleiten, die Anwendungsfälle haarklein funktional zu zerlegen.

Kommen wir auf den Beispiel des digitalen Aufnahmegerätes zurück. Wir können z.B. den Benutzer als einen Akteur identifizieren, der mit den Anwendungsfällen zum Aufnehmen der Nachricht, Abspielen der Nachricht, Löschen der Nachricht, Setzen der Uhrzeit, Setzen der Alarmzeit und einfaches Ablesen der Uhrzeit und des Datums vom Display interagieren kann. Abbildung 5 verdeutlicht den Sachverhalt.

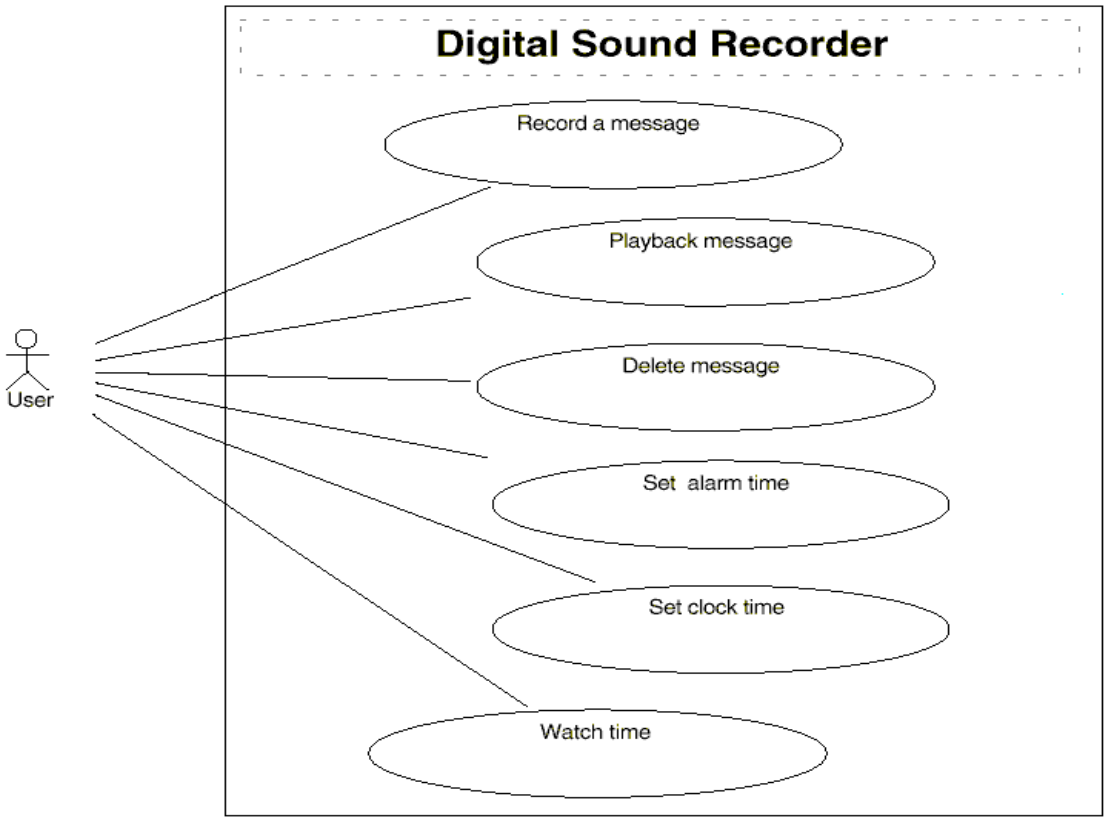

Abbildung 5 Anwendungsfalldiagramm eines digitalen Aufnahmegerätes

3.2 Klassendiagramme

3.2.1 Basiselemente

- **Klasse** enthält die Beschreibung der Struktur und des Verhaltens von Objekten, die sie erzeugt oder mit ihr erzeugt werden können. Sie enthält die Definition der Attribute, Operationen und der Semantik für eine Menge von Objekten. Eine Klasse wird durch Rechtecke mit dem Klassennamen dargestellt. Diese Rechtecke werden in drei Rubriken (Klassenname, Attribute, Operationen) jeweils durch horizontale Linien aufgeteilt.
  Die Abbildung 6 zeigt die grafische Notation.

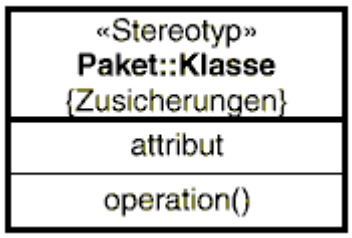

Abbildung 6 UML Notation der Klassen

Es können auch parametrisierbare und abstrakte Klassen dargestellt werden. Eine **parametrisierbare Klasse** ist eine mit generischen formalen Parametern versehene

Schablone, mit der gewöhnliche nicht generische Klassen erzeugt werden können. Bei der grafischen Notation werden zusätzlich die Parameter im gestrichelten Rechteck rechts oben mitangegeben. Die Verfeinerung erfolgt mittels einer **<<bind>> Beziehung** zur parametrisierbaren Klasse (Template) wie in der Abbildung 6 zu sehen ist.

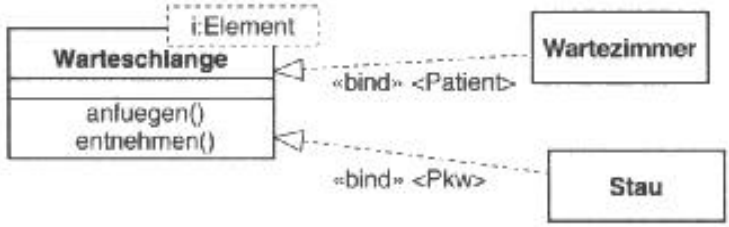

Abbildung 7 Parametrisierbare Klasse

Abstrakte Klassen (es werden keine Objektexemplare von der Klasse erzeugt, und die abstrakte Klasse ist bewußt dazu da, die Basis für weitere Unterklassen zu bilden) werden durch den Eigenschaftswert **{abstrakt}** unter dem Klassennamen dargestellt

- **Objekt** ist eine im laufenden System konkret vorhandene und agierende Einheit. Jedes Objekt ist Exemplar bzw. Instanz einer Klasse. Grafische Notation zeigt die Abbildung 8.

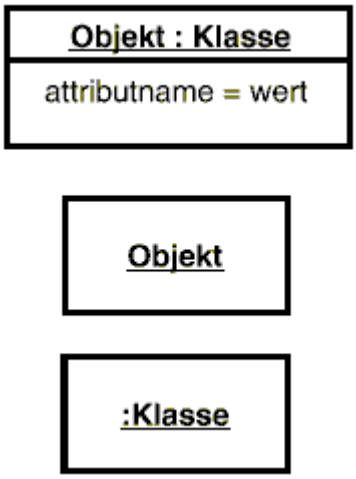

Abbildung 8 UML Notation der Objekte

- **Attribut** ist ein (Daten) –Element, das in jedem Objekt einer Klasse enthalten ist, und von jedem Objekt mit einem individuellen Wert repräsentiert wird. Man unterscheidet auch noch zwischen den **abgeleiteten Attributen** (Attribute dessen Wert durch eine Berechnungsvorschrift ermittelt wird und nicht direkt änderbar ist) und **Klassenattributen (**Attribute die zu einer Klasse gehören, damit können alle Objekte der Klasse auf dieses gemeinsame Attribut zugreifen; z.B. um die Objekte einer Klasse beim Erzeugen zu numerieren etc.). **Abgeleitete Attribute** werden durch das vorangestelltes Zeichen „/“ gekennzeichnet. **Klassenattribute** werden unterstrichen dargestellt

- **Operationen** werden durch ihre Signatur beschrieben (Operationsname, Parameter , ggf. Rückgabetyp). Man kann auch die **abstrakten Operationen** formulieren, deren Implementierung erst in einer Unterklasse stattfindet.

  Abstrakte Operationen werden entweder kursiv geschrieben oder durch den Eigenschaftswert **{abstrakt}** markiert.

- **Schnittstelle , Schnittstellenklasse (engl. interface, interface class)**

  Schnittstellen beschreiben einen ausgewählten Teil des extern sichtbaren Verhaltens von Modellelementen (Klassen oder Komponenten). Schnittstellenklassen sind abstrakte Klassen (genauer Typen), die ausschließlich abstrakte Operationen definieren. Schnittstellenklassen werden wie normale Klassen mit dem Stereotyp **<<interface>>** notiert. Die Operationen müssen nicht mehr als abstrakt gekennzeichnet werden, da es bei Schnittstellen zwingend vorausgesetzt ist. Schnittstellen können auch andere Schnittstellen erweitern, dazu wird das Stereotyp **<<extend>>** benutzt (siehe Abb. 9). Die Implementierung der Schnittstelle wird durch das Stereotyp **<<implement>>** angedeutet.

  Die folgende Abbildung 9 zeigt die Klasse String, die eine Schnittstelle Sortierbar implementiert. Diese Implementierung wird auch von der Klasse Sortierte Stringliste genutzt, was durch die Abhängigkeitsbeziehung notiert ist. Das heißt, die Klasse Sortierte Stringliste nutzt die Eigenschaften von String, die in der Schnittstelle Sortierbar definiert sind. Die Definition der Schnittstellen ist hilfreich, um die Kopplung zwischen Klassen zu explizieren und zu reduzieren. Im Beispiel ist die Schnittstellennutzerin Sortierte Stringliste nur von zwei speziellen Operationen der Klasse String abhängig. Alle anderen Operationen der klasse String könnten ohne Beeinträchtigung der Sortierbarkeit verändert werden. Eine Information, die man sonst nur durch intensives Studium der Klasse Sortierte Stringliste gewonnen hätte.

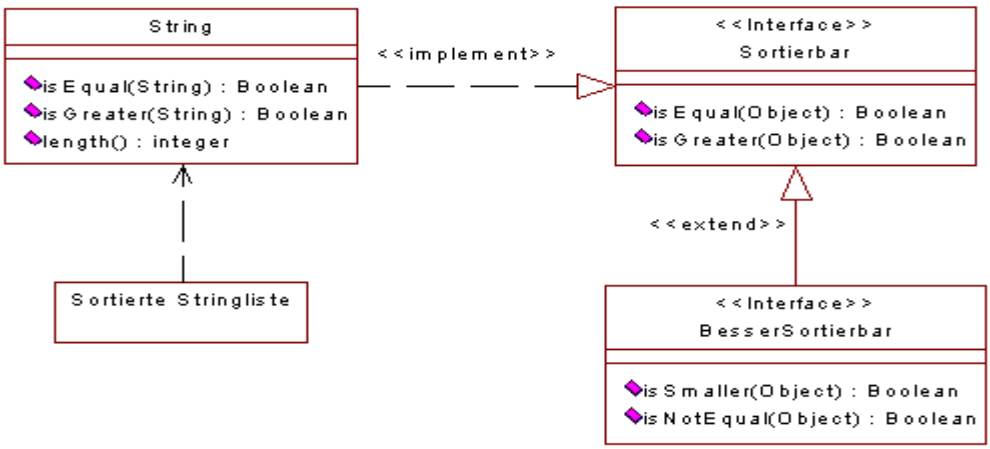

Abbildung 9 Schnittstellenklassen

- **Zusicherung (engl. constraint)** ist ein Ausdruck, der die möglichen Inhalte, Zustände oder die Semantik eines Modellelementes einschränkt und der stets erfüllt sein muß. Zusicherungen können in Form von Stereotypen, formalen OCL (Object Constraint Language) Ausdrücken, freie Formulierung (Notiz), Abhängigkeitsbeziehung angegeben werden. Zusicherungen in Form reiner boolescher Ausdrücke werden auch **assertions** genannt.

  Zusicherungen werden in geschweifte Klammern gefaßt **{Zusicherung}.**

  Beispiele findet man in der Abbildung 10.
- **Eigenschaftswert (engl. Property String)**

  Eigenschaftswerte sind sprach – und werkzeugspezifische Schlüsselwort – Wert – Paare, die die Semantik einzelner Modellelemente um spezielle charakteristische Eigenschaften erweitern. Eigenschaftswerte bestehen aus einem Schlüsselwort und einem Wert und werden einzeln oder als Aufzählung in geschweifte Klammern gesetzt. Zusicherungen und Eigenschaftswerte überlappen sich in ihrer Verwendung etwas. Eigenschaftswerte können nicht frei formuliert werden, sondern sind spezifische Schlüsselwort-Wert – Paare. Sie beeinflussen im Gegensatz zu frei formulierten Zusicherungen in den meisten Fällen direkt die Codegenerierung. Falls statt einer Zusicherung also ein entsprechender Eigenschaftswert definiert werden kann, ist dies im Hinblick auf die präzisere Bedeutung und die Code-Generierung vorzuziehen. Beispiele findet man in der Abbildung 10.
- **Stereotypen** sind projekt- , unternehmens- oder methodenspezifische Erweiterungen vorhandener Modellelemente von UML. Stereotypen ermöglichen eine **mentale und ggf. visuelle Unterscheidung** und geben Hinweise auf die **Art der Verwendung**, auf den Bezug zur vorhandenen Anwendungsarchitektur, der Entwicklungsumgebung usw. Die Stereotypen sind von den Eigenschaftswerten zu unterscheiden, da sie im Gegensatz zu Eigenschaftswerten das Metamodel von UML um ein neues Element erweitern.

  Notation: **<<stereotyp>>** oder durch spezielle Symbole z.B. **Akteur**

  Selbstkreierte und werkzeugspezifische Stereotypen sind insofern problematisch, als daß damit die Standardisierung der UML hintergangen wird. Beispiele findet man in der Abbildung 10.

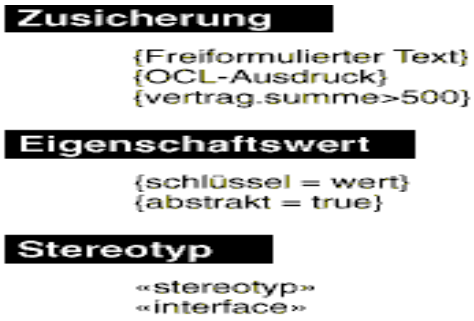

Abbildung 10 Zusicherungen, Eigenschaftswerte, Stereotypen

- **Notiz (engl. note)**
  Notizen sind Anmerkungen zu Klassen, Attributen, Operationen, Beziehungen usw. Sie werden durch Rechtecke dargestellt, bei denen eine Ecke geknickt ist.
- **Paket (engl. package)**

  Pakete sind Ansammlungen von Modellelementen beliebigen Typs, mit denen das Gesamtmodell in kleinere überschaubare Einheiten gegliedert wird. Ein Paket wird in Form eines Aktenregisters dargestellt. Innerhalb des Symbols steht der Name des Paketes. Werden innerhalb des Symbols Modellelemente angezeigt (Klassen oder verschachtelte Pakete), steht der Name auf der Registerlasche.

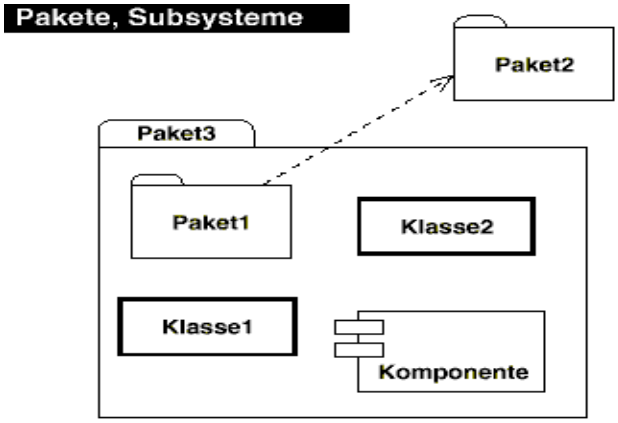

Abbildung 11 UML Notation der Pakete

3.2.2 Beziehungselemente

- **Generalisierung / Spezialisierung** (vergleichbar mit der Vererbung)

  Generalisierung und Spezialisierung sind Abstraktionsprinzipien zur hierarchischen Strukturierung der Semantik eines Modells. Es werden also Eigenschaften hierarchisch gegliedert, d.h. Eigenschaften allgemeinerer Bedeutung werden allgemeineren Klassen **(Oberklassen)** zugeordnet und speziellere Eigenschaften werden Klassen zugeordnet, die den allgemeineren untergeordnet sind **(Unterklassen).** Die Unterscheidung in Ober- und Unterklassen erfolgt häufig aufgrund eines Diskriminators. Unter **Diskriminator** versteht

man ein Unterscheidungsmerkmal bzw. ein Charakteristikum. Die Vererbungsbeziehung wird mit einem großen, nicht ausgefüllten Pfeil dargestellt, wobei der Pfeil von der Unterklasse zu der Oberklasse zeigt(siehe Abbildung 12).

Vererbung

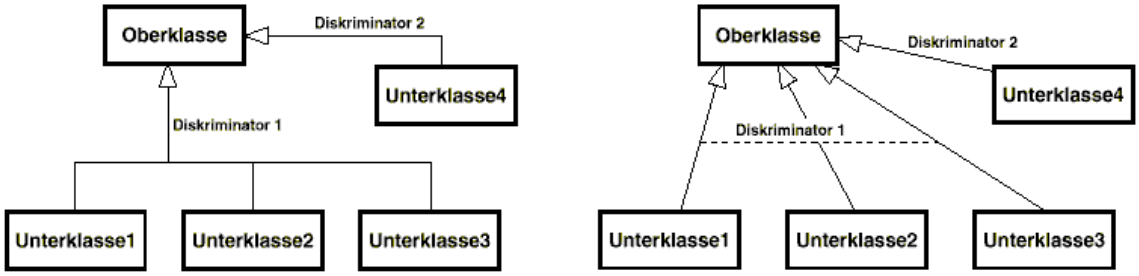

Abbildung 12 Generalisierung / Spezialisierung

- **Mehrfachvererbung**
  Bei der Mehrfachvererbung kann eine Klasse mehr als eine Oberklasse besitzen. Statt von der Klassenhierarchie kann man in diesem Fall auch von einer Klassenheterarchie sprechen. Die Besonderheit der Mehrfachvererbung lassen sich mit dem folgendem Bild erklären:

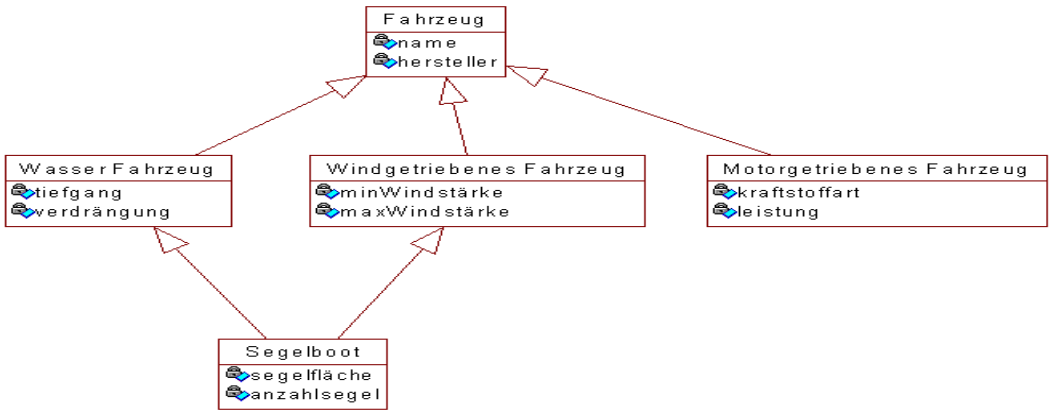

Abbildung 13 Mehrfachvererbung

Die Klasse Segelboot erbt die Eigenschaften ihrer Oberklassen, so daß die Eigenschaften der Klasse Fahrzeug über zwei verschiedene Wege (also doppelt) geerbt werden. In der UML sieht man deswegen zwei besondere Zusicherungen vor. Standardmäßig sind **Vererbungsbeziehungen** immer **{disjoint},** dies führt zu den doppelten Attributen (WasserFahrzeug.Name,windgetriebenesFahrzeug.Name und WasserFahrzeug.Hersteller, WindgetriebenesFahrzeug.Hersteller) bei der Klasse Segelboot. Durch die andere Zusicherung **{overlapping}** werden die Attribute einmal geerbt.

- **Assoziation / Links**

  Eine Assoziation beschreibt eine Verbindung (Beziehung) zwischen Klassen. Die konkrete Beziehung zwischen zwei Objekten dieser Klassen wird Objektverbindung **(engl. link)** genannt. Objektverbindungen sind also die Instanzen einer Assoziation. In der UML Notation werden die Beziehungen durch eine Linie zwischen den beteiligten Klassen dargestellt. An den jeweiligen Enden kann die Kardinalität der Beziehung angegeben werden. Kardinalität einer Assoziation gibt an, mit wievielen Objekten der gegenüberliegenden Klasse ein Objekt assoziiert sein kann. Wenn diese Zahl variabel ist, wird die Bandbreite, d.h. Minimum und Maximum angegeben. Liegt das Minimum bei 0, bedeutet es, daß die Beziehung optional ist. Jede Assoziation kann mit einem Namen versehen werden.

  Zusätzlich können auch Rollennamen angegeben werden, die beschreiben, wie das Objekt durch das in der Assoziation gegenüberliegende Objekt gesehen wird. Die Abbildung 14 zeigt die grafische Assoziation.

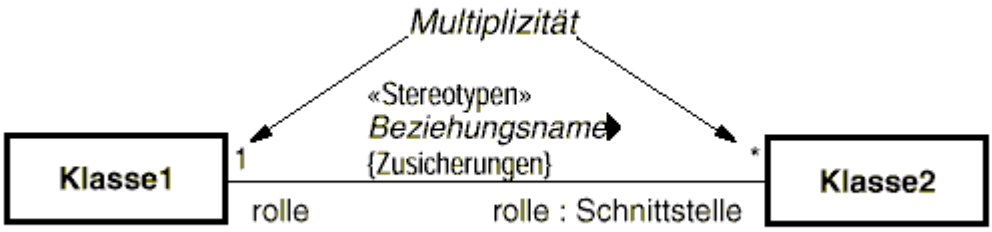

Abbildung 14 Assoziation

- **Aggregation**

  Eine Aggregation ist eine Assoziation, deren beteiligte Klassen eine Ganzes – Teile – Hierarchie darstellen. Unter Aggregation versteht man also die Zusammensetzung eines Objektes aus einer Menge von Einzelteilen. Kennzeichnend für die Aggregationen ist, daß das Ganze Aufgaben stellvertretend für seine Teile wahrnimmt und dann an die Einzelteile weiterleitet. Damit übernimmt eine Klasse (das Aggregat) eine besondere Rolle der Verantwortung und Führung. Eine Aggregation wird wie eine Assoziation als Linie zwischen zwei Klassen dargestellt und zusätzlich mit einer kleinen Raute auf der Seite des Aggregats (des Ganzen) versehen (siehe Abb. 15).

- **Komposition**

  Eine Komposition ist eine strenge Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig[1] sind. Wichtige Unterschiede zur Aggregation: Die Kardinalität auf der

[1] Die Teile können nicht ohne des Ganzen existieren, daher auch die Existenzabhängigkeit !

Seite des Aggregats kann nur 1 sein, da jedes Teil nur Teil eines Kompositionsobjektes ist (wegen Existenzabhängigkeit). Außerdem werden beim Löschen des Ganzen zuerst alle Teilobjekte vernichtet, das wieder auf die Existenzabhängigkeit der Teile zurückzuführen ist. Die Notation der Komposition entspricht der Aggregation mit der ausgefüllten Raute auf der Seite des Aggregats.

Grafische Notation für die Aggregation und Assoziation wird auf der Abbildung 15 dargestellt.

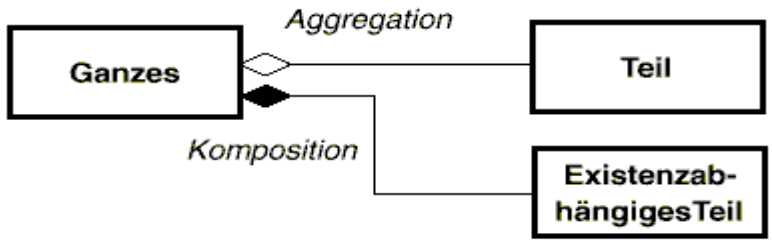

Abbildung 15 Aggregation und Komposition

- **Abhängigkeitsbeziehung (engl. Dependency)**

  Eine Abhängigkeit ist eine Beziehung zwischen zwei Modellelementen, die zeigt, daß eine Änderung in dem einen (unabhängigen) Element eine Änderung in dem anderen (abhängigen) Element notwendig macht. Dargestellt wird eine Abhängigkeit durch einen gestrichelten Pfeil, wobei der Pfeil vom abhängigen auf das unabhängige Element zeigt.

  Grafische Notation:

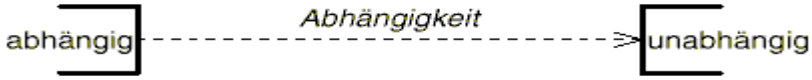

Abbildung 16 Abhängigkeitsbeziehung

Wir können nun an dieser Stelle das Beispiel vom digitalen Aufnahmegerät etwas erweitern, indem wir zunächst eine grobe Klassenhierarchie für das System beschreiben (siehe Abb. 17).

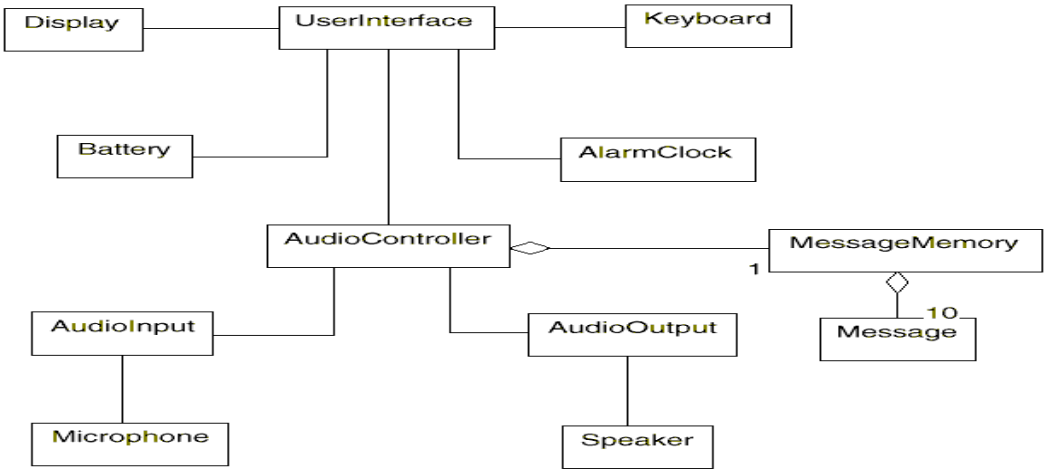

Abbildung 17 Klassendiagramm des digitalen Aufnahmegerätes

Der Benutzer interagiert mit dem System über den Bildschirm (Display) und über die Tastatur (Keyboard). Beide Klassen (**Display** und **Keyboard**) werden jedoch sehr einfach gehalten, so daß **eine zentrale Klasse Userinterface** definiert wird, die sämtliche Interaktionen managt. Weiterhin existieren die Klassen **Battery** und **Alarmclock**, die jeweils den Status der Batterie und die Uhr des Gerätes überwachen. Die Klasse **AudioController** unterstützt die Aufnahme (über die Klassen **AudioInput** und **Microphone**) und Wiedergabe (über die Klassen **AudioOutput** und **Speaker**) der Nachrichten. Die Nachrichten werden in dem Speicher gehalten, der durch die Klasse **MessageMemory** verwaltet wird. In dem Klassendiagramm sind zwei **Aggregationen** enthalten. Zuerst ist **die genau eine** Klasse MessageMemory ein Teil der AudioController Klasse. Weiterhin müssen durch die Anforderungen an das System genau **10 Messages** in MessageMemory verwaltet werden. Sämtliche Klassen die miteinander kommunizieren, werden durch **Assoziationen** miteinander verbunden.

Wir können das System auch unter einem anderen Gesichtspunkt modellieren. Dazu zerlegen wir es zuerst in mehrere Subsysteme. Die Abbildung 18 zeigt den Sachverhalt.

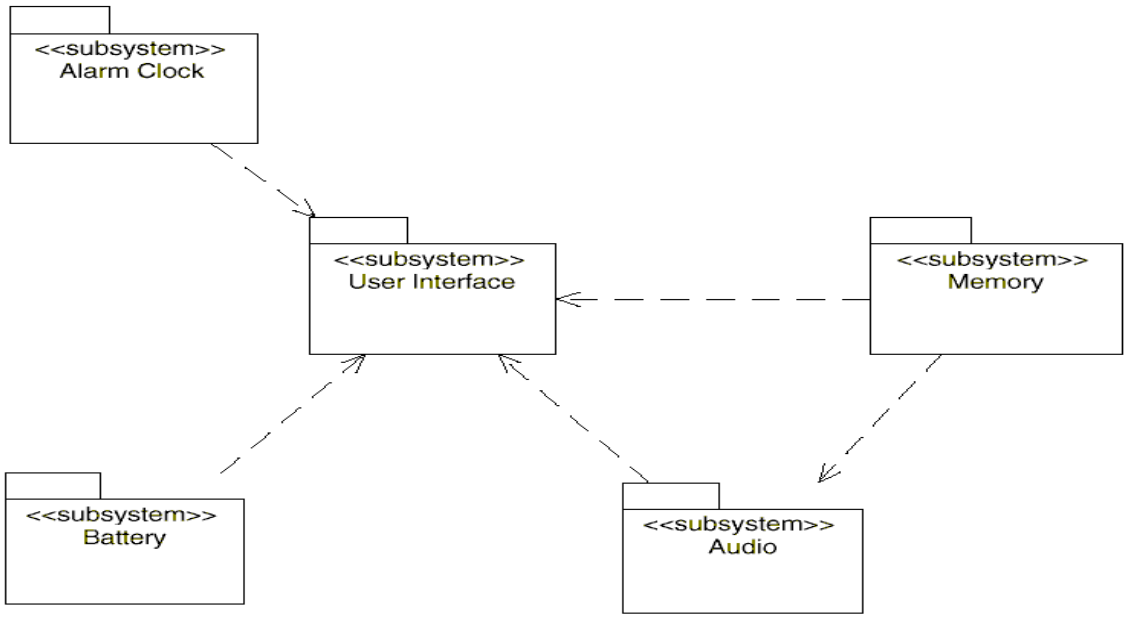

Abbildung 18 Paketdiagramm des digitalen Aufnahmegerätes

Wir haben an dieser Stelle auch ein **eigenes Stereotyp <<subsystem>>** definiert. Damit nutzten wir die Möglichkeit von UML eigene projektbezogene Elemente einzuführen. Dies trägt an dieser Stelle dem besseren Verständnis des Diagramms bei. Weiterhin sind die einzelnen Elemente als Pakete aufgefaßt, da wir voraussetzen, daß jedes Subsystem aus mehreren Klassen besteht. Da die einzelnen Subsysteme miteinander auch kommunizieren, werden die Abhängigkeitsbeziehungen zwischen den einzelnen Paketen definiert. Man wird

an dieser Stelle auch erkennen können, daß genau ein Paket (hier User Interface) die zentrale Rolle in der Interaktion einnimmt, da sämtliche Subsysteme davon abhängig sind. Nun werden im nächsten Schritt für jedes Subsystem Klassendiagramme entwickelt. An dieser Stelle möchte ich Beispielhaft nur die Entwicklung des Subsystems User Interface vorstellen. Die Abbildung 19 stellt es dar.

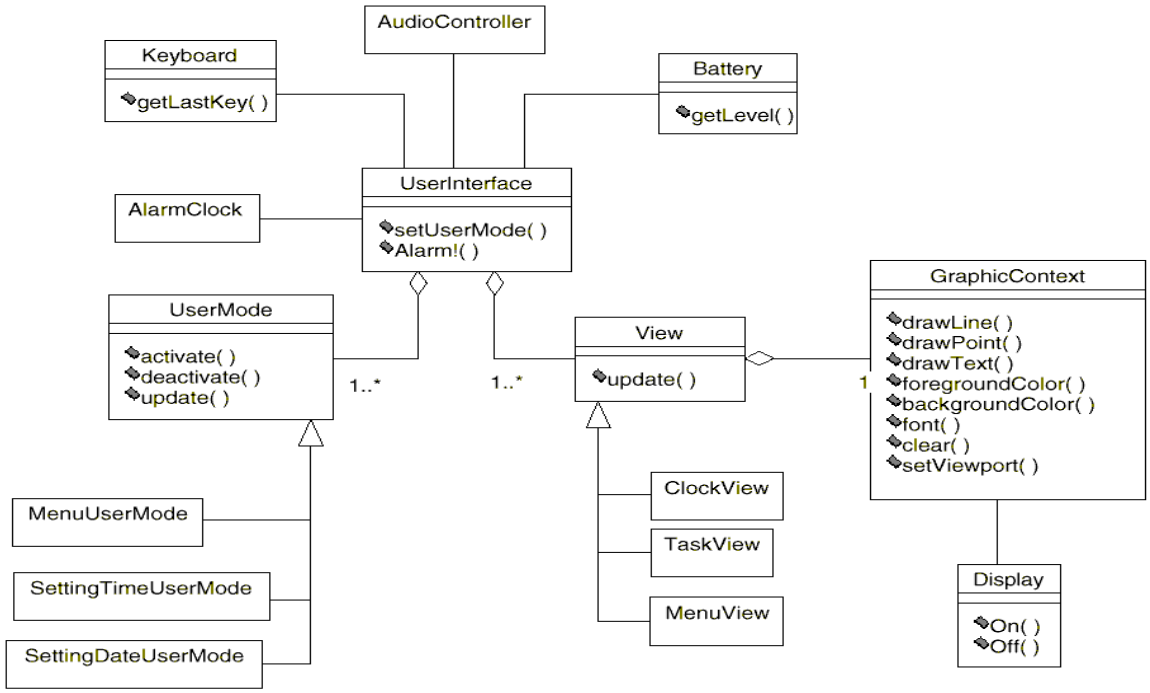

Abbildung 19 Klassendiagramm des Subsystems User Interface

Die zentrale Klasse des Subsystems User Interface ist die fast gleichnamige Klasse **UserInterface**. Diese Klasse empfängt die Eingaben der Tastatur (**Keyboard** Klasse) und gibt bestimmte Meldungen auf dem Bildschirm. Für die Ausgabe wird die Klasse **GraphicContext** benutzt, die einige Standardoperationen hierzu anbietet. Es existiert genau ein GraphicContext, das Teil der Klasse **View** ist (**Aggregation**). Weiterhin werden durch **Spezialisierung** drei zusätzliche Klassen (**Clockview**, **Taskview**, **Menuview**) definiert, die für die Darstellung der Uhrzeit, der gerade aktiven Aufgabe und Menü verantwortlich sind. Um es einfacher zu Handhaben wird jedem GraphicContext ein rechteckiger Bereich auf dem Bildschirm reserviert. Die Klasse UserInterface empfängt auch die Nachrichten von den Klassen **Battery** und **AlarmClock** und leitet gegebenenfalls die Reaktion darauf ein. Die Klasse **UserMode** managt die verschiedenen Betriebsmodi des Gerätes. Letztendlich verfügt die Klasse UserInterface auch eine Assoziation zu der Klasse **AudioController**, die aber in dem anderem Subsystem definiert ist. Hier sieht man also den eigentlichen Grund für die Abhängigkeit des Subsystems Audio vom Subsystem User Interface.

3.3 Verhaltensdiagramme

3.3.1 Aktivitätsdiagramm

Aktivitätsdiagramme beschreiben die Ablaufmöglichkeiten eines Systems mit Hilfe von Aktivitäten. Eine Aktivität ist ein einzelner Schritt in einem Verarbeitungsablauf. Sie ist ein Zustand mit einer internen Aktion und mindestens einer ausgehenden Transition. Die ausgehende Transition impliziert den Abschluß der internen Aktion. Eine Aktivität kann mehrere ausgehende Transitionen haben, wenn diese durch Bedingungen unterschieden werden können. Aktivitäten können Bestandteil von Zustandsdiagrammen sein, gewöhnlich werden sie jedoch in eigenen Aktivitätsdiagrammen verwendet. Dargestellt wird eine Aktivität durch eine Figur mit gerader Ober- und Unterseite und konvex geformten Seiten. Die Figur enthält eine Aktionsbeschreibung, die ein Name, eine frei formulierbare Beschreibung, Pseudocode oder Programmiersprachencode sein kann. Eingehende Transitionen lösen eine Aktivität aus. Existieren mehrere eingehende Transitionen, so kann jede dieser Transitionen unabhängig von den anderen die Aktivität auslösen. Die Abbildung 20 verdeutlicht den Sachverhalt.

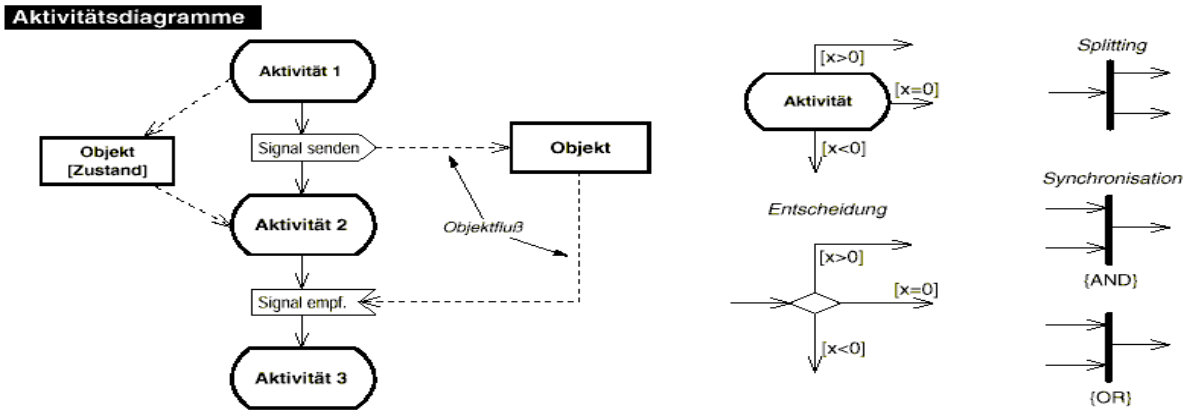

Abbildung 20 Aktivitätsdiagramm

Die ausgehenden Transitionen können mit Bedingungen (boolesche Ausdrücke) in eckigen Klammern versehen werden. Es besteht auch die Möglichkeit die Transitionen zu synchronisieren.

Häufig bewirken Aktivitäten die Änderung eines Objektzustandes. Objektzustände werden durch Rechtecke dargestellt, die den Namen des Objektes und in eckigen Klammern den Objektzustand enthalten

3.3.2 Kollaborationsdiagramm

Ein Kollaborationsdiagramm (engl. Collaboration diagram, Zusammenarbeitsdiagramm) zeigt eine Menge von Interaktionen zwischen ausgewählten Objekten in einer bestimmten begrenzten Situation (Kontext) unter Betonung der Beziehungen zwischen den Objekten. Der

Verlauf der Kommunikation zwischen den Objekten wird dabei durch die Numerierung der Nachrichten verdeutlicht. Damit zwei Objekte miteinander kommunizieren können, muß der Sender einer Nachricht eine Referenz auf das Empfängerobjekt haben, d.h. eine Assoziation zu diesem. Zwischen den Objekten werden also Assoziationslinien gezeichnet, auf denen dann die Nachrichten notiert werden. Ein kleiner Pfeil zeigt jeweils die Richtung der Nachricht vom Sender zum Empfänger.

Kollaborationsdiagramme

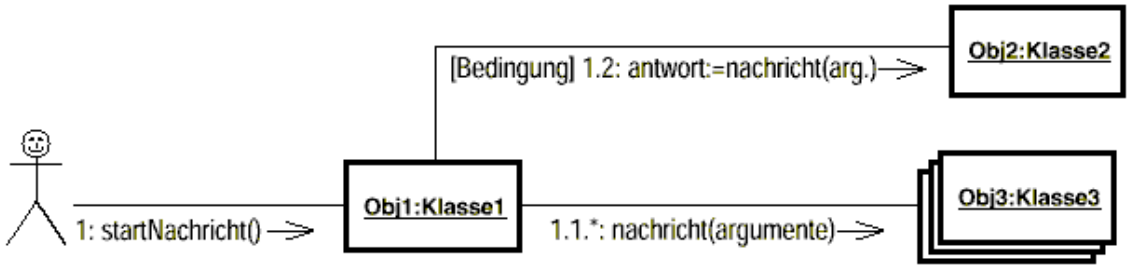

Abbildung 21 Kollaborationsdiagramm

Folgende Syntax liegt der Nachrichtenbezeichnung zugrunde:

Vorgänger – Bedingung Sequenzausdruck Antwort := Nachrichtenname (Parameterliste)

- **Vorgänger – Bedingung** : eine Aufzählung der Sequenznummern anderer Nachrichten, die bereits gesendet sein müssen, bevor diese Nachricht gesendet werden darf. Damit kann eine Synchronisation durchgeführt werden. Die Sequenznummern werden durch Komma getrennt aufgelistet und mit einem „/“ abgeschlossen.
- **Sequenzausdruck**: Numerierung der Nachrichten z.B. 2.2.1.0 Der Punkt gibt so zu sagen die Untersequenznummer an. Wiederholtes Senden von Nachrichten kann durch ein Sternchen „* “ gekennzeichnet werden. Um die Iteration näher zu beschreiben, kann in eckigen Klammern eine entsprechende Angabe in Pseudocode oder in der verwendeten Programmiersprache erfolgen (z.B. 1.2. * [i:=1..n])
- **Antwort:** Die von der Nachricht gelieferte Antwort kann mit dem Namen versehen werden. Dieser Name kann in den anderen Nachrichten als Argument verwendet werden.
- **Nachrichtenname (Parameterliste) :** Name der Nachricht gleichlautend zu entsprechenden Operation.

Wir erweitern nun an dieser Stelle das Beispiel des digitalen Aufnahmegerätes um ein Kollaborationsdiagramm. Die Abbildung 22 zeigt die Interaktion der Objekte bei der Aufnahme eines Audioblockes .

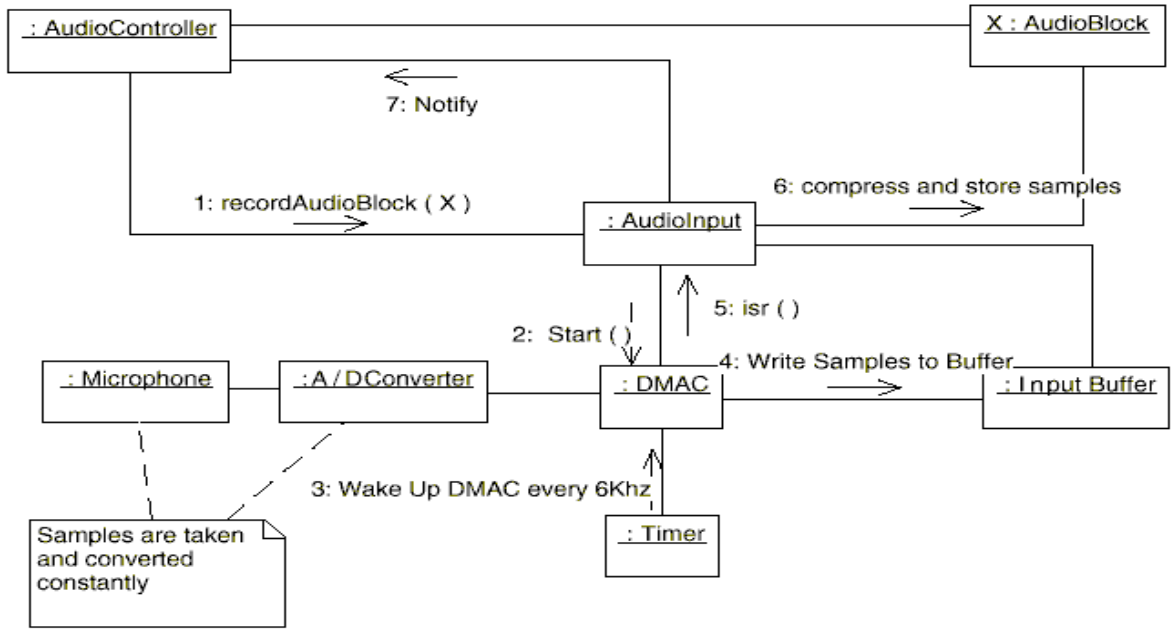

Abbildung 22 Kollaborationsdiagramm der Aufnahme eines Audioblockes

Die Aufnahme erfolgt durch periodisches Abtasten des Signals vom Mikrophon (**Klasse Microphone**) mit einer Frequenz von 6kHz (**Klasse Timer**). An dieser Stelle wird man mit einem Problem konfrontiert eine große Menge an Daten vom Mikrophon durch den **AudioInput** an den **AudioBlock** weiterzuleiten, ohne den Prozessor für die Zeit des Transfer zu beanspruchen. Als Lösung bietet sich an dieser Stelle der Einsatz von DMAC **(direct memory access chanel)** an. **DMAC** kann mehrere Bytes mit einem „Rutsch“ von einer Stelle zu der anderen übertragen, ohne den Prozessor dazu zu benutzen. Die Übertragung muß lediglich durch Interruptroutine (**ISR**) gesteuert werden. Der Ablauf der Aufnahme läßt sich nun leicht nachvollziehen. **AudioController** möchte einen **AudioBlock X** aufnehmen und schickt die entsprechende Nachricht an den AudioInput. AudioInput leitet die Nachricht Start() an den DMAC-Kanal, der 6000 mal pro Sekunde vom Timer geweckt wird und Daten an InputBuffer weitergibt. Mit Hilfe der Interruptroutine **isr()** wird weiterer Ablauf gesteuert. Die Daten werden komprimiert und in dem Audioblock X abgespeichert. Ist der Vorgang abgeschlossen, wird AudioController vom AudioInput entsprechend unterrichtet.

3.3.3 Sequenzdiagramm

Ein Sequenzdiagramm (engl. **Sequence diagram**) zeigt im Grunde die gleichen Sachverhalte wie ein Kollaborationsdiagramm, jedoch aus einer anderen Perspektive. Bei Kollaborationsdiagrammen steht die Zusammenarbeit der Objekte im Vordergrund. Der zeitliche Verlauf der Kommunikation zwischen Objekten wird durch Numerierung der Nachrichten angedeutet. Bei **Sequenzdiagrammen** steht der **zeitliche Verlauf** der Nachrichten **im Vordergrund**. Die Objekte werden lediglich durch gestrichelte senkrechte

Lebenslinien dargestellt. Oberhalb der gestrichelten Linie steht der Name bzw. das Objektsymbol. Die Zeitachse verläuft von oben nach unten. Die Nachrichten werden als waagerechte Pfeile zwischen den Objekt - Linien gezeichnet (siehe Abbildung 23).

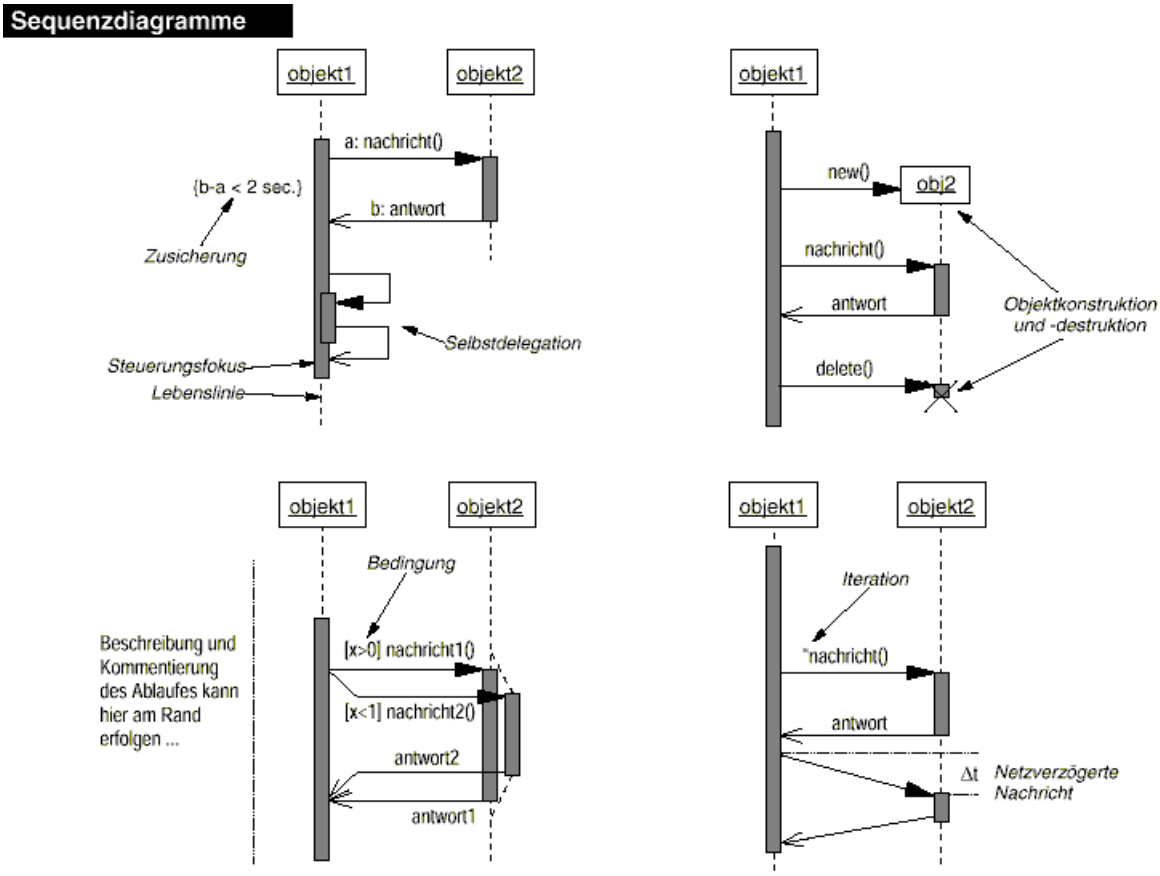

Abbildung 23 Sequenzdiagramm

Die Überlagerung der gestrichelten Lebenslinien durch breite , nicht ausgefüllte (oder graue) senkrechte Balken, symbolisiert den Steuerungsfokus. Der Steuerungsfokus gibt an, welches Objekt gerade die Programmkontrolle besitzt, d.h. welches Objekt gerade aktiv ist. Am linken und rechten Rand können frei formulierte Erläuterungen und Zeitanforderungen notiert werden. Das Erzeugen eines neuen Objektes wird durch eine Nachricht, die auf ein Objektsymbol trifft, angezeigt. Die Destruktion eines Objektes wird durch ein Kreuz am Ende des Steuerungsfokus angedeutet. Sonst sind Bedingungen, Zusicherungen und Iterationen genauso wie bei Kollaborationsdiagrammen erlaubt.

Auch an dieser Stelle bietet es sich an, das Beispiel des digitalen Aufnahmegerätes auszubauen. Die Abbildung 24 zeigt das Sequenzdiagramm für das Betreten und Verlassen des **stand-by** Betriebsmodus.

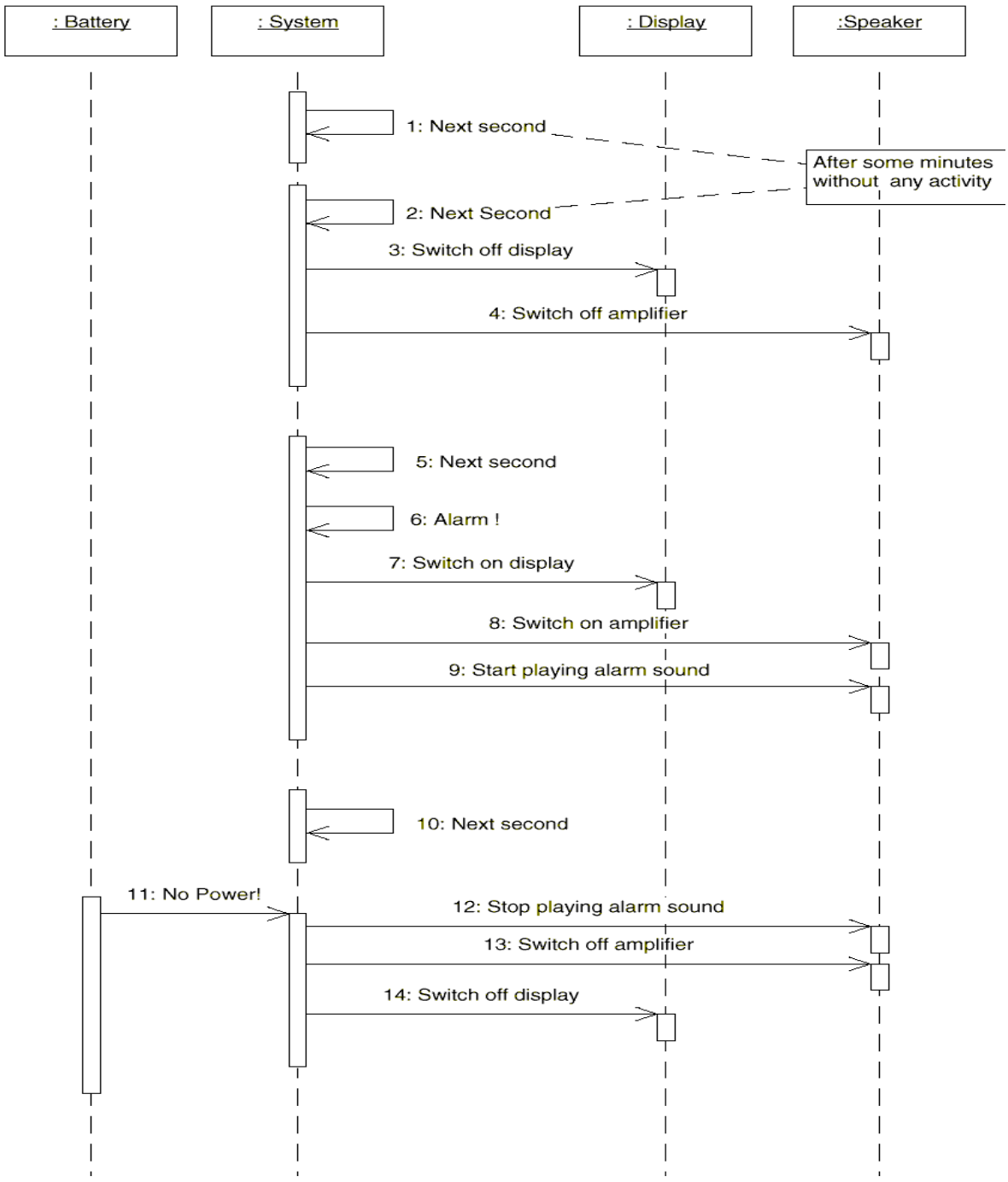

Abbildung 24 Sequenzdiagramm [Stand By Modus]

Der Ablauf sieht dann folgendermaßen aus: System wird einige Minuten ohne irgendeine Aktivität betrieben, der stand by Modus wird ausgelöst, Display und Lautsprecher Verstärker werden abgeschaltet. Nun kommt es zu einem voreingestelltem Alarm, was das Verlassen des stand by Modus bewirkt, Display und Verstärker für Lautsprecher werden aktiviert, Alarm Melodie wird gespielt. Die Batterie meldet nun „No Power“ , und das System stellt das Abspielen der Melodie ein, schaltet Display und Verstärker für Lautsprecher aus.

3.3.4 Zustandsdiagramm

Ein Zustandsdiagramm **(engl. state diagram)** zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Zustandsänderungen stattfinden. Ein Zustandsdiagramm beschreibt eine hypothetische Maschine (Endlicher Automat) und besteht aus:

- Einer endlichen, nicht – leeren Menge von Zuständen
- Einer endlichen, nicht – leeren Menge von Ereignissen
- Funktionen, die den Übergang von einem Zustand in den nächsten beschreiben
- Einem Anfangszustand
- Einer Menge von Endzuständen

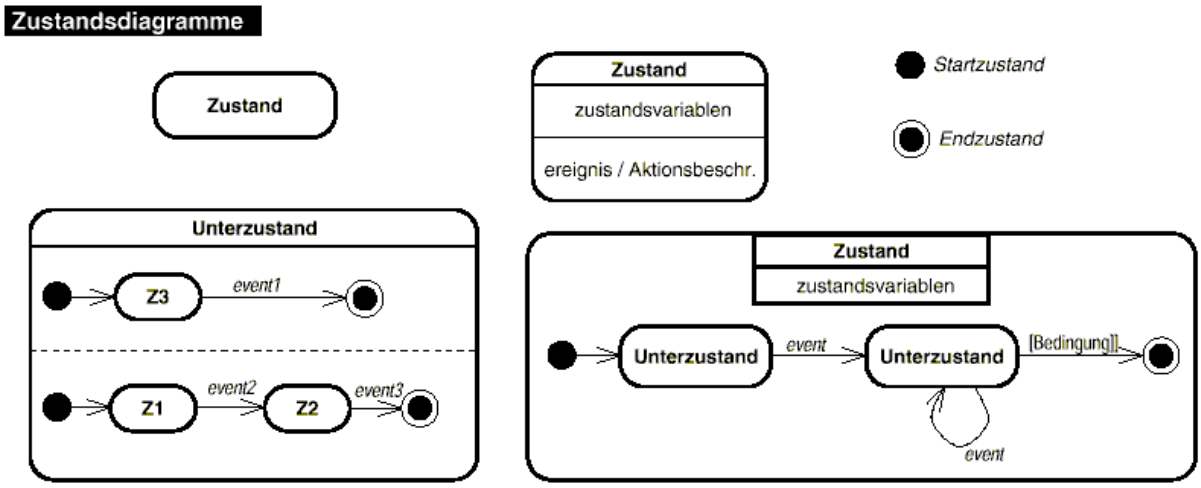

Abbildung 25 Zustandsdiagramm

Ein Zustand gehört zu einer Klasse und stellt eine Abstraktion bzw. Zusammenfassung einer Menge von möglichen Attributwerten dar, die die Objekte dieser Klasse einnehmen können. Nicht jede Änderung eines Attributwertes wird hier als Zustandsänderung angesehen. Darin besteht gerade die Abstraktion, nur solche Ereignisse zu berücksichtigen, die das Verhalten des Objektes maßgeblich beeinflussen. Damit kann man auch einen Zustand als Zeitspanne zwischen zwei Ereignissen ansehen.

Man unterscheidet zwei besondere Zustandstypen: **Startzustand** (zu einem Startzustand kann kein Übergang stattfinden) und **Endzustand** (von einem Endzustand führt kein Ereignis mehr weg).

Zustände können einen **eindeutigen Namen** enthalten **oder anonyme** Zustände sein. Es ist jedoch stets zu beachten, daß anonyme Zustände grundsätzlich voneinander verschieden sind.

Jeder Zustand kann eine Menge von **Zustandsvariablen** enthalten. Die Menge der Zustandsvariablen ist eine Untermenge der Attribute der Klasse, die zur Beschreibung bzw. Identifikation des Zustandes bedeutsam ist.

Übergänge von einem Zustand zum nächsten werden durch **Ereignisse** ausgelöst, die aus einem Namen und einer Liste möglicher Argumente bestehen. Ein Zustand kann **Bedingungen** an diese Ereignisse knüpfen, die erfüllt sein müssen, damit der Zustand durch dieses Ereignis eingenommen werden kann. Ereignisse können Aktionen innerhalb des Zustandes auslösen. Drei Auslöser sind in der UML vordefiniert:

- Entry :löst automatisch beim Eintritt in einen Zustand aus
- Exit :löst automatisch beim Verlassen eines Zustandes aus
- Do :wird immer wieder ausgelöst, solange der Zustand aktiv ist.

Zustände werden durch abgerundete Rechtecke notiert, die optional durch horizontale Linien in bis zu drei Bereiche geteilt werden.

Zustandsvariablen werden wie folgt notiert:

Variable : Klasse =Initialwert {Merkmal} {Zusicherung}

Zustände können in weitere, entweder **sequenzielle** oder **parallele Unterzustände** geschachtelt sein. Für die Darstellung paralleler Unterzustände wird der Zustand durch gestrichelte Linien unterteilt.

Auch hier erweitern wir das Beispiel des digitalen Aufnahmegerätes um ein weiteres Diagramm. Abbildung 26 zeigt das Zustandsdiagramm für den Fall des Setzen der Zeit.

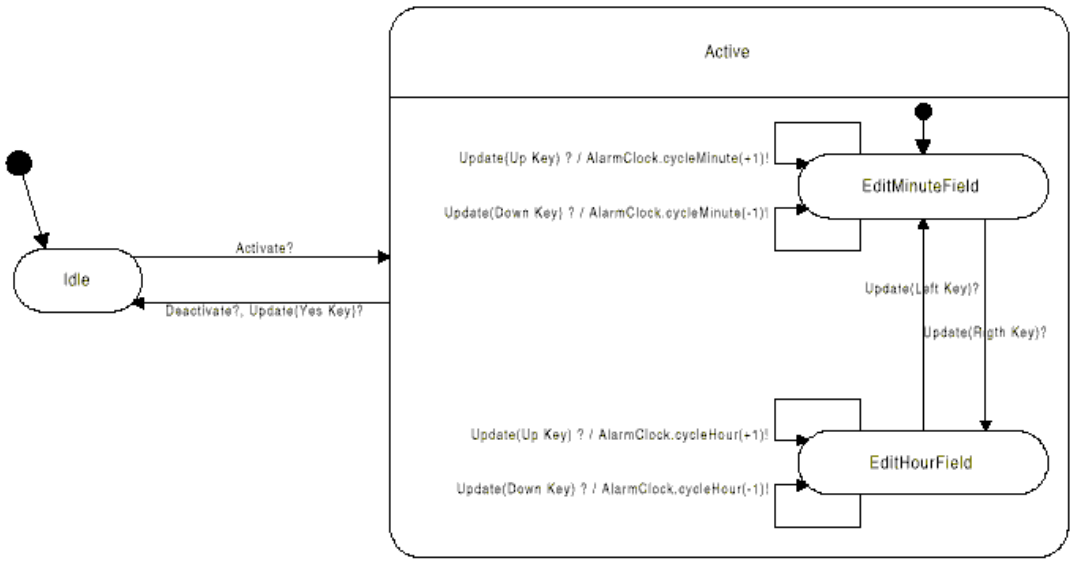

Abbildung 26 Zustandsdiagramm zum Setzen der Zeit

Wir befinden uns zuerst in einem **Idle** Zustand. Durch Aktivieren des Zeitsetzen – Modus über die Menüführung, gelangen wir in den **Active** Zustand. Hier kann durch die Pfeile nach oben und unten das aktive Feld verändert werden.Durch die Pfeile links und rechts wird das aktive Feld gewechselt. Durch Yes – Taste wird wieder in den Idle Zustand zurückgekehrt.

3.4 Implementierungsdiagramme

3.4.1 Komponentendiagramm

Eine Komponente stellt ein physisches Stück Programmcode dar, entweder als Quellcode, Binärcode, DLL oder ausführbares Programm. Komponentendiagramme **(engl. component diagram)** zeigen die Beziehungen der Komponenten untereinander.

In der Praxis sind Komponenten (physische Sicht) den Paketen(logische Sicht) sehr ähnlich: sie definieren Grenzen, sie gruppieren und gliedern eine Menge einzelner Elemente. Komponenten können über Schnittstellen verfügen. Die Anwendung der Komponentendiagrammen kommt spätestens mit dem Beginn der Realisierung, wenn je nach der verwendeten Programmiersprache festgelegt werden muß, in welchen Dateien der Programmcode zu den einzelnen Klassen usw. stehen soll.

Eine Komponente wird als Rechteck notiert, das am linken Rand zwei kleine Rechtecke trägt. Innerhalb der Komponente wird der Name der Komponente und ggf. ihr Typ beschrieben. Außerdem können in der Komponente wiederum weitere Elemente (Objekte, Komponenten, Knoten) enthalten sein.

Komponentendiagramme

Abbildung 27 Komponentendiagramm

3.4.2 Verteilungsdiagramm

Ein Knoten ist ein zur Laufzeit physisch vorhandenes Objekt, das über Rechenleistung bzw. Speicher verfügt, also Computer (Prozessoren), Geräte usw.

Verteilungsdiagramme **(engl. deployment diagram)** zeigen, welche Komponenten und Objekte auf welchen Knoten (Prozessen, Computer) laufen, d.h. wie diese konfiguriert sind und welche Kommunikationsbeziehungen dort bestehen.

Dargestellt werden Knoten durch Quader. Knoten , die miteinander kommunizieren, d.h. entsprechende Verbindungen unterhalten, werden durch Assoziationslinien miteinander verbunden. Innerhalb der Quader können optional Komponenten oder Laufzeitobjekte (Prozesse) plaziert werden. Auch Schnittstellen und Abhängigkeitsbeziehungen zwischen diesen Elementen sind zulässig.

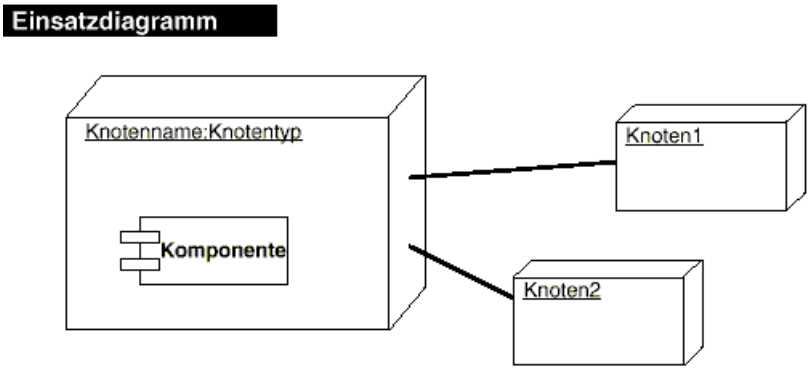

Abbildung 28 Verteilungsdiagramm

Um zum letzten mal auf den Beispiel zurückzukommen, stellt die nachfolgende Abbildung 29 das Verteilungsdiagramm des digitalen Aufnahmegerätes vor.

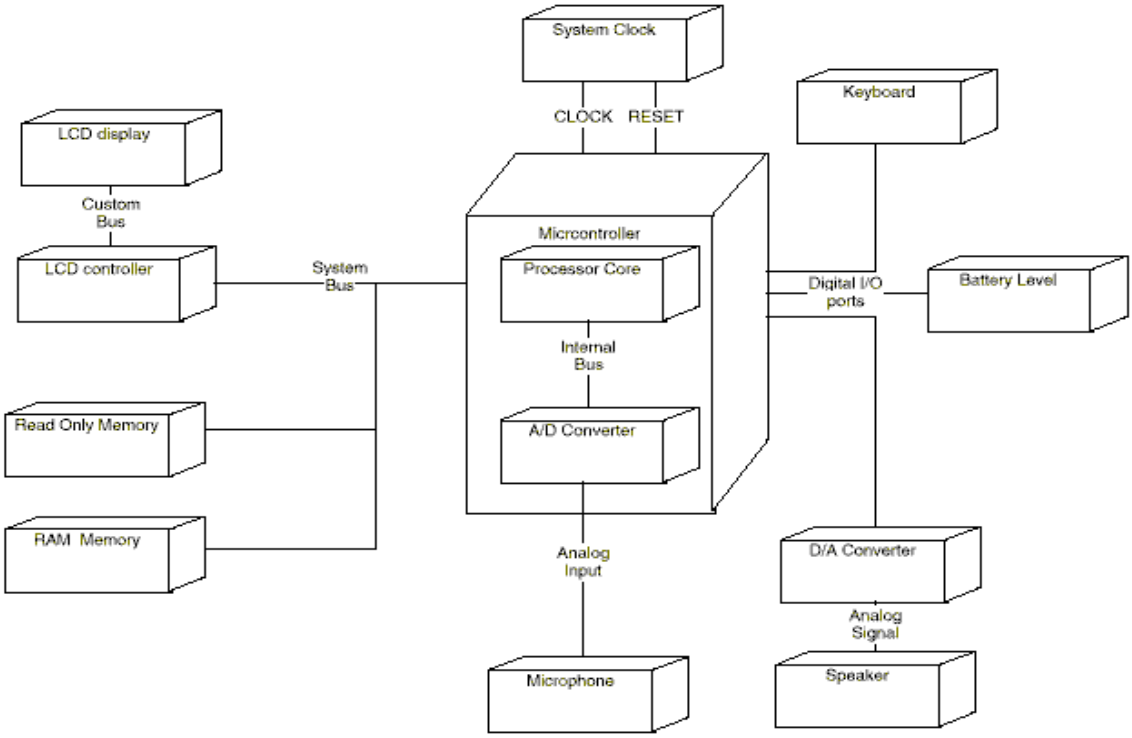

Abbildung 29 Verteilungsdiagramm des digitalen Aufnahmegerätes

4. Rational Rose

Die Firma Rational entwickelt und vertreibt seit einigen Jahren Rational Rose als ein Tool für Objekt Orientierte Analyse (OOA) und Objekt Orientiertes Design (OOD). Insbesondere stellt das Tool die Möglichkeit mit der UML zu modellieren und anschließend einen Code in den Programmiersprachen Java, C++, Ada etc. zu erstellen. Mit dem integrierten Analyser läßt sich auch das Reengineering betreiben. So ist es möglich den veränderten C++ Code wieder in ein Modell umzuwandeln. Die nachfolgende Abbildung zeigt die Bedienungsoberfläche von Rational Rose nach dem Start.

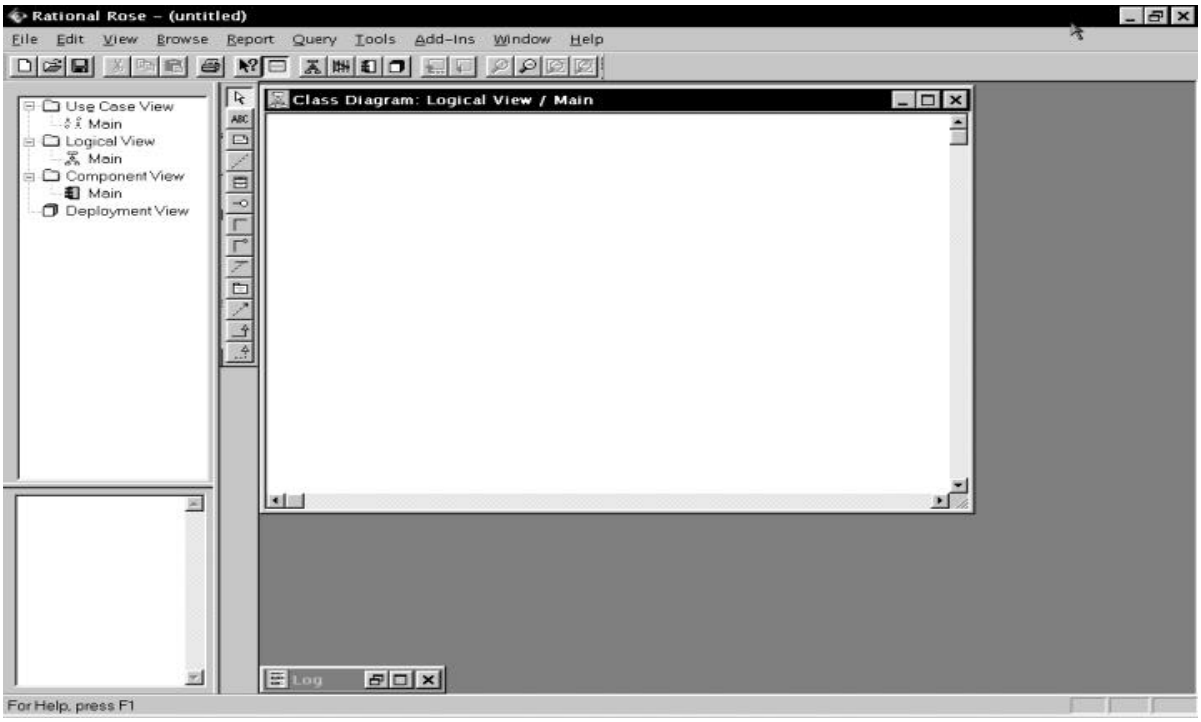

Abbildung 30 Rational Rose

Man unterscheidet in Rational Rose zwischen vier verschiedenen Sichten.

- Die Anwendungsfallsicht (Use – Case View) erlaubt das modellieren der Zusammenhänge verschiedener Anwendungsfälle.

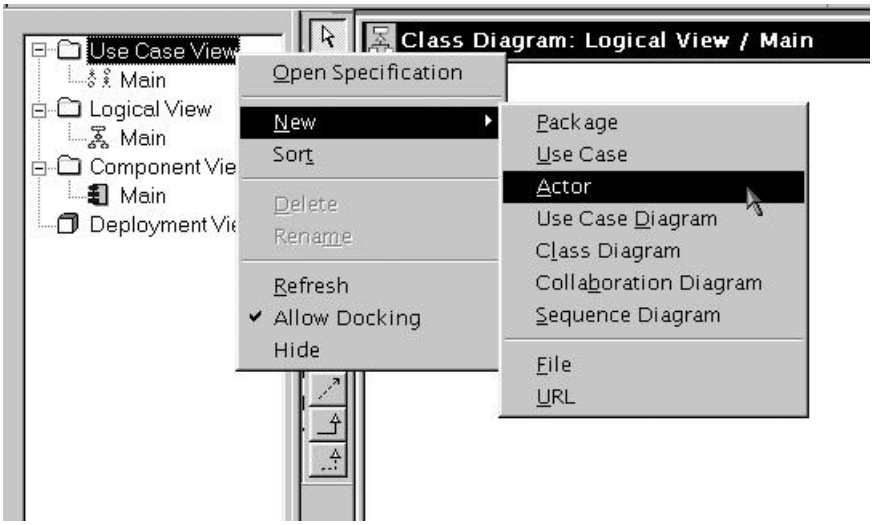

Abbildung 31 Use Case View

- Die logische Sicht (Logical View) erlaubt die Beschreibung der Funktionalität des Systems auf der Basis der Klassendiagramme

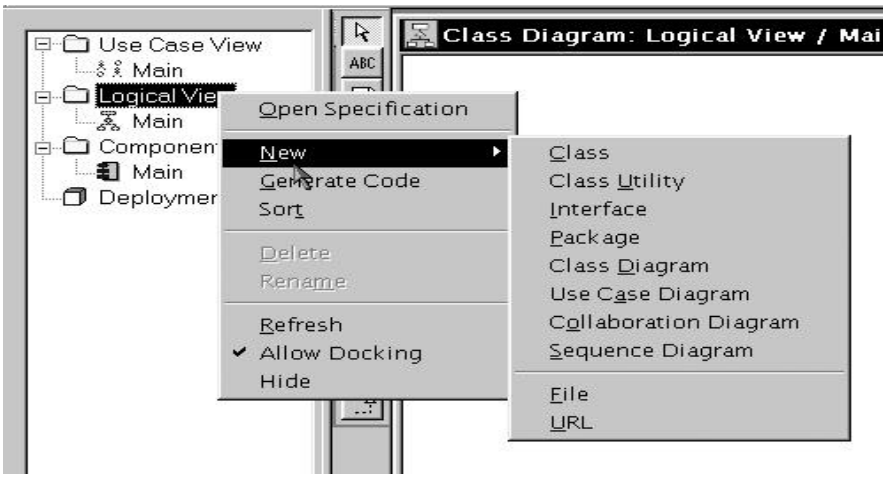

Abbildung 32 Logical View

- Component View beschreibt die Modulare Organisation der Software (z.B. Dateien)

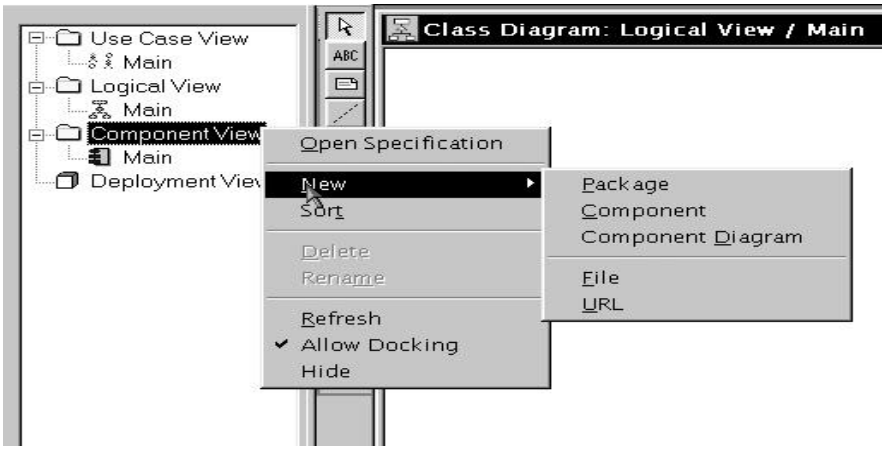

Abbildung 33 Component View

- Deployment View beschreibt die Knotenstruktur der Software zur Laufzeit (z.B. welche Ressourcen die einzelnen Knoten benutzen / belegen)

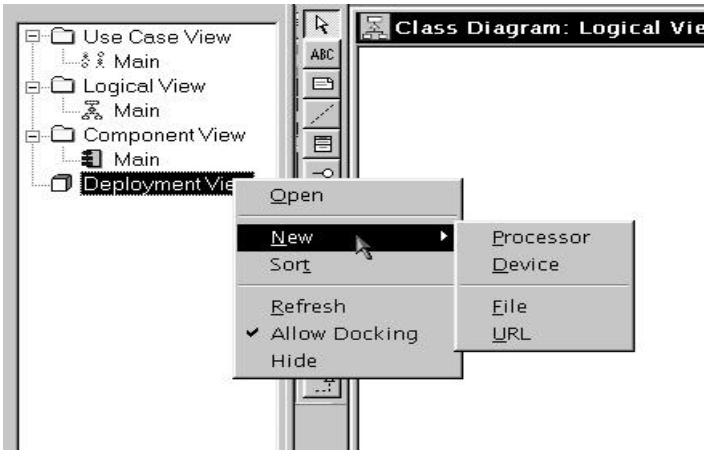

Abbildung 34 Deployment View

Es ist wichtig an dieser Stelle anzumerken, daß für die Code – Generierung nur die Logische Sicht herangezogen wird. Die anderen Sichten bieten eher die Ergänzung des Modells zum besseren Verständnis des Systems.

5. Vor – und Nachteile von UML / Rose

Aus meiner persönlichen Sicht liegen die wesentlichen Vorteile bei dem Einsatz von UML in der breiten Unterstützung sämtlicher objektorientierter Grundsätze. Man kann an vielen Stellen ein und dasselbe Problem auf unterschiedliche Art und Weise lösen, ohne sich dabei durch die Vorgaben der Sprache eingeengt zu fühlen. Die Sprache läßt also den Softwareentwicklern den Freiraum, nach eigenen Vorstellungen zu modellieren. Besonders Vorteilhaft erscheint die Tatsache, daß UML auch die Erweiterungsmöglichkeiten durch eigene Sprachkonstrukte vorsieht (Metamodell), und damit wirklich jedem das Recht anbietet, den eigenen Notationsrahmen zu erschaffen. Diese Möglichkeit sollte aber nur in äußersten Notfällen (wenn es nicht anders geht) verwendet werden, da man sonst der Gefahr, sich vom Standard zu entfernen, entgegenläuft .

Weiterer Vorteil liegt in der Standardisierung der UML. Die UML bietet eine fast perfekte Grundlage für die Kommunikation der verschiedenen Entwicklungsteams miteinander. Diagramme können nun nicht nur von den Fachleuten der Software –Firma , sondern auch von den außerhalb stehenden verstanden und verbessert werden. Lästiges Einarbeiten in die verschiedenen Notationen entfällt, falls sich alle grundsätzlich an die UML halten.

Weiterer Vorteil liegt in der zunehmenden Verbreitung der Unified Modeling Language. Die exakten Zahlen sind zwar noch schwer abzuschätzen, es zeichnet sich jedoch ein wachsender Trend für den Einsatz der UML ab.

Die Nachteile lassen sich nun wiederum aus dem Umfang der Sprache ableiten. UML ist sehr vielfältig, so daß auch am Anfang sehr viel Aufwand für das Aneignen und Verstehen sämtlicher Sprachkonstrukte aufgebracht werden muß. Außerdem wird man in der Regel feststellen, daß viele Elemente sehr selten zum Einsatz kommen und damit eher als Ballast der Sprache angesehen werden. So habe ich in der Darstellung der UML Notation auf die Erläuterung der OCL Konstrukte (Object Constraint Language) verzichtet, obwohl OCL als formale Sprache zur Erweiterung der Semantik der UML Notation herangezogen werden kann. Damit lassen sich zwar Zusicherungen, Invarianten, Vor –und Nachbedingungen, Navigationspfade etc. kurz und aussagekräftig darstellen, aber man kann diese im begrenzten Maße auch durch UML Basiselemente beschreiben.

Speziell für die Entwicklung der Software für eingebettete Systeme läßt sich anmerken, daß UML eine objektorientierte Modellierungssprache ist. Sämtliche Konzepte von UML können nur dann ausgenutzt werden, wenn wirklich objektorientiert programmiert wird. Es macht wenig Sinn, objektorientiert zu modellieren , wenn anschließend kein objektorientierter Code verwendet wird. Steht kein objektorientierter Compiler zur Verfügung, sollte man sich über

den Einsatz anderer Werkzeuge und Spezifikationssprachen Gedanken machen. Man kann zwar im begrenzten Maße sämtliche objektorientierte Ansätze in den „strukturierten“ Code konvertieren (also z.B. von C++ nach C überführen), das Ergebnis läßt jedoch zu wünschen übrig, insbesondere leidet die Code- Qualität / Lesbarkeit darunter.
Beim Modellieren mit der UML kann man sämtliche Ergebnisse theoretisch einfach auf das Papier bringen und immer wieder damit arbeiten. Praktisch stößt man jedoch bei größeren Projekten sehr schnell an die Grenzen. Durch ständiges Ändern der Modelle wird das Aktualisieren der Ergebnisse schnell zu einem Horror–Vorhaben. An dieser Stelle kommen die verschiedenen Werkzeuge und Tools zum Einsatz, die uns ja gerade beim Entwickeln der Software mit allen möglichen Mitteln unterstützen sollen. Rational Rose als Tool bietet den Vorteil, sämtliche Elemente der UML in verschiedenen Diagrammen übersichtlich darzustellen. Rose unterstützt vor allem den Entwicklungsprozeß durch die Möglichkeit der Codegenerierung und des Reengineering. Meiner Meinung nach bietet das Tool eine gute Unterstützung beim Modellieren mit der UML. Vorteilhaft ist der Einsatz von virtuellen Bildschirmen, die auch bei größeren Projekten für klare Übersicht sorgen. Rose erlaubt das Dokumentieren sämtlicher Elemente, diese Dokumentation steht jedoch auf einem Ausdruck aus Platzgründen nicht zur Verfügung.
Rose unterstützt Reengineering mit Hilfe des integrierten Analysers. Falls dieser allerdings aufgrund bestimmter Compiler - spezifischer Gegebenheiten etc. beim Interpretieren des Codes versagt, ist der manuelle Eingriff notwendig.

Fazit:

- Der Einsatz von Rational Rose ist empfehlenswert, auch wenn man an einigen Stellen an die Grenzen des Werkzeuges stößt. Die Einarbeitung in das Werkzeug gestaltet sich leicht, falls man mit UML und dem Entwicklungsprozeß vertraut ist.
- Der Einsatz der UML–Sprache ist bei einer objektorientierten Vorgehensweise sehr nützlich, erfordert aber relativ langes Einarbeiten in die Sprache. Bei einer, nicht objektorientierten Vorgehensweise, werden die Vorteile von UML dagegen nicht richtig genutzt, so daß andere Spezifikationssprachen an dieser Stelle eventuell besser geeignet sind.

6. Literaturhinweise

[Booch95] Grady Booch
Object – oriented Analysis and Design
Benjamin / Cummings Publishing Company
Deutsche Übersetzung Judith Muhr, Drachselried
Addison – Wesley Publishing Company , 1995
Objektorientierte Analyse und Design

[Burkhard] Rainer Burkhard
UML – Unified Modeling Language
Objektorientierte Modellierung für die Praxis
Addison – Wesley – Longman , 1997

[DSR99] Digital Sound Recorder: A case study on designing embedded systems using the UML notation
Technical Report No 234
Ivan Porres Paltor, Johan Lilius
Turku Centre for Computer Science , Finnland

[Fowler98] Martin Fowler, Kendall Scott
UML Distilled. Applying the Standard Object Modeling Language.
Addison Wesley Longman , Inc.
Deutsche Übersetzung:
Arnulf Mester,Michael Sczittnick, Günter Graw
UML konzentriert

[OMG] www.omg.org

[Oestereich98] Bernd Oestereich
Objektorientierte Softwareentwicklung
Analyse und Design mit der Unified Modeling Language
4.Auflage, R.Oldenbourg Verlag München Wien 1998

[Rational] www.rational.com

[Rumbaugh91] Rumbaugh, J Object Oriented Modelling and Design
Prentice-Hall, Englewood Cliffs, 1991

[Wahl] Günter Wahl
UML kompakt , Erschienen im Objektspektrum 2/1998