

# Der Catalysis–Approach

Simone Everding, Alex Wiesmaier

Revision 31. Januar 2000

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>4</b>
1.1	Anforderungen an Software . . . . .	4
1.2	Phasen der Software-Entwicklung . . . . .	5
1.3	Vorgehensmodelle . . . . .	5
1.3.1	Wasserfallmodell . . . . .	5
1.3.2	V-Modell . . . . .	6
1.3.3	Spiralmodell . . . . .	7
<b>2</b>	<b>Entwicklungsschichten</b>	<b>8</b>
2.1	Einleitung . . . . .	8
2.2	UML & Running-Example . . . . .	8
2.2.1	Use-Case-Diagram . . . . .	9
2.2.2	Sequence-Diagram . . . . .	10
2.3	Business-Model . . . . .	11
2.4	Requirements-Specification . . . . .	12
2.5	Component-Design . . . . .	13
2.6	Object-Design . . . . .	15
<b>3</b>	<b>Grundprinzipien</b>	<b>17</b>
3.1	Abstraktion . . . . .	17
3.2	Präzision . . . . .	17
3.3	Komponenten . . . . .	18
<b>4</b>	<b>Resümee</b>	<b>19</b>
4.1	Vorteile des Catalysis-Approach . . . . .	19
4.1.1	Komponentenbasierte Entwicklung . . . . .	19
4.1.2	High-integrity Entwicklung . . . . .	19
4.1.3	Objekt-orientierte Entwicklung . . . . .	19
4.1.4	Reengineering . . . . .	20
4.2	Fazit . . . . .	20

# Abbildungsverzeichnis

1.1	Wasserfallmodell . . . . .	5
1.2	V-Modell . . . . .	6
1.3	Spiralmodell . . . . .	7
2.1	Use-Case-Diagram zum Running-Example . . . . .	9
2.2	Sequence-Diagram zum Running-Example . . . . .	10
2.3	Use-Case-Diagram zum Business-Model . . . . .	11
2.4	Use-Case-Diagram zur Requirements-Specification . . . . .	12
2.5	Use-Case-Diagram zum Component-Design . . . . .	13
2.6	Sequence-Diagram zum Component-Design . . . . .	14
2.7	Use-Case-Diagram zum Object-Design . . . . .	15
2.8	Sequence-Diagram zum Object-Design: Actor Verkauf . . . . .	16
2.9	Sequence-Diagram zum Object-Design: Actor Lehrer . . . . .	16

# Kapitel 1

## Einleitung

Seit den 60er Jahren versucht man, die Software-Krise mit Hilfe von systematischem, ingenieurmäßigem Vorgehen (Software Engineering) zu beenden. Es wurde z.B. die Software-Entwicklung in verschiedene Phasen unterteilt und verschiedene Vorgehensweisen entwickelt, um die Anforderungen an Software zu erfüllen.

### 1.1 Anforderungen an Software

Die wichtigste Eigenschaft von Software ist die Integrität, d.h. Vollständigkeit und Korrektheit.

Die zunehmende Komplexität der Software erzwingt Team-Entwicklung, da eine Person allein zu lange für die Entwicklung brauchen würde. Damit eine Aufgabe von einem Team bearbeitet werden kann, muß eine klare Trennung der Aufgabenteile vorgenommen werden. Ebenso müssen Abhängigkeiten, Konventionen und Schnittstellen klar definiert werden.

Die letzte hier genannte Anforderung ist die Flexibilität. Einerseits auf die schnelle Änder- und Anpassbarkeit bezogen, andererseits auf die Variantentechnik, die in anderen Bereichen (z.B. in der Autoindustrie) längst nicht mehr wegzudenken ist. In der Software-Entwicklung sind sie noch nicht so weit verbreitet, dabei könnte man damit z.B. Versionen für verschiedene Länder mit unterschiedlichen Rechtsgrundlagen umsetzen.

## 1.2 Phasen der Software-Entwicklung

Die Software-Entwicklung wird in vier Phasen eingeteilt: Analyse, Design, Implementierung, Test.

- Analyse — Umfaßt die Spezifikation des Außenverhaltens (Was soll das System machen?)
- Design — Legt die interne Systemarchitektur für das gewünschte Außenverhalten fest (Wie soll das System die Spezifikation umsetzen?)
- Implementierung — Realisiert die erstellte Spezifikation
- Test — Überprüft die Funktionalität der Implementierung auf Korrektheit und Vollständigkeit

## 1.3 Vorgehensmodelle

Drei bekannte Vorgehensmodelle werden nun kurz vorgestellt.

### 1.3.1 Wasserfallmodell

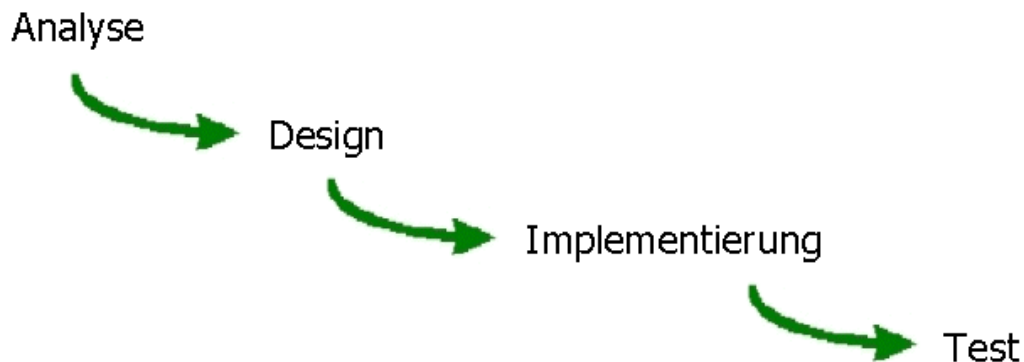


Abbildung 1.1: Wasserfallmodell

Die vier Phasen werden nacheinander durchgeführt, d.h. man muß in den einzelnen Phasen alle Aspekte berücksichtigen.

Ein großer Nachteil dieses Modells ist, daß Fehler, die in der Analysephase bzw. Designphase gemacht werden, erst in der Testphase aufgedeckt werden. Zu diesem Zeitpunkt ist es damit kaum noch möglich, vorhandene Kosten- und Zeitpläne einzuhalten.

### 1.3.2 V-Modell

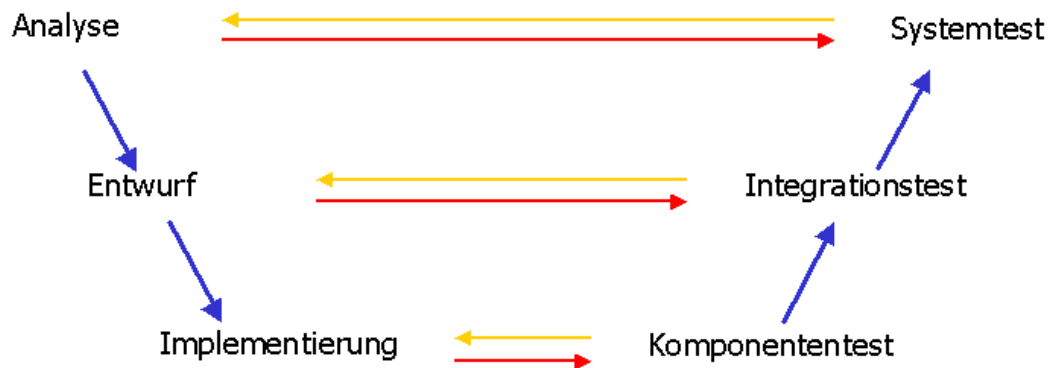


Abbildung 1.2: V-Modell

Im Unterschied zum Wasserfallmodell wird beim V-Modell die Testphase in drei Ebenen aufgeteilt: Komponenten-, Integrations- und Systemtests. Dabei gehen die Testfälle aus den korrespondierenden Phasen hervor:

- Analyse — Systemtest
- Design — Integrationstest
- Implementierung — Komponententest

Zu diesem Modell muß bemerkt werden, daß die Testreihenfolge eigentlich falsch ist, denn wie schon beim Wasserfallmodell werden auch hier die Fehler der frühen Phasen erst ganz am Ende entdeckt, wenn es für eine schnelle Korrektur zu spät ist.

### 1.3.3 Spiralmodell

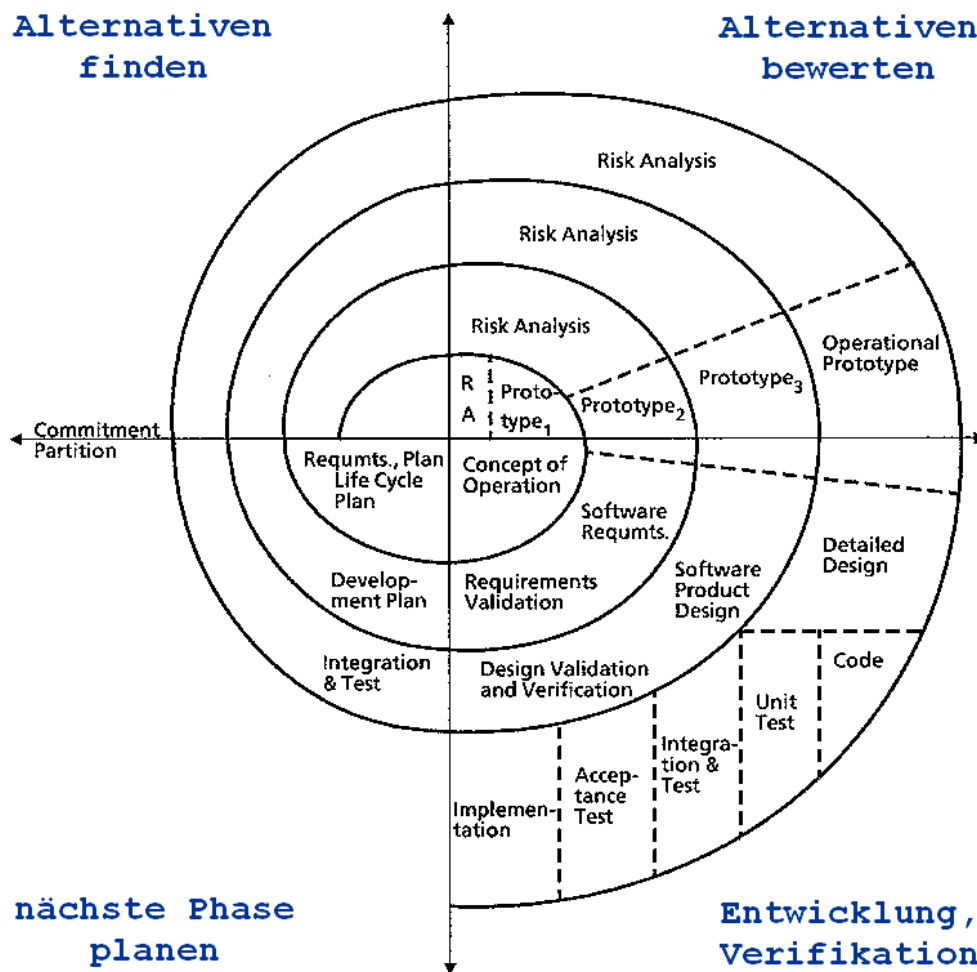


Abbildung 1.3: Spiralmodell

Das Spiralmodell von B. Böhm basiert darauf, daß in jeder „Spiralrunde“ nur ein Aufgabenteil bearbeitet wird. Dabei werden für den zu bearbeitenden Aufgabenteil zunächst Lösungsalternativen gesucht, dann werden diese bewertet und in einen Prototypen umgesetzt, der evtl. dem Kunden vorgeführt werden kann. Danach erfolgt die eigentliche Entwicklung und deren Verifikation. Am Ende der Bearbeitung eines Aufgabenteils steht schließlich das Planen der nächsten „Runde“.

Vorteilhaft ist bei diesem Ansatz, daß gut überschaubare Aufgabenteile statt der Gesamtaufgabe bearbeitet werden. Sollten dabei Fehler auftreten, werden diese frühzeitig bemerkt, so daß durch Korrektur dieser verhältnismäßig kleinen Fehler größere Folgefehler vermieden werden können.

Allerdings ist für die sinnvolle Teilung der Gesamtaufgabe ein guter Überblick nötig.

# Kapitel 2

## Entwicklungsschichten

### 2.1 Einleitung

Der Catalysis-Approach ist ein Vorgehensmodell, welches den Entwickler auf dem Weg von der Analyse zur Implementierung führt. Dabei wird auf die wohl mächtigsten Werkzeuge der Softwareentwicklung — Abstraktion und Präzision — zurückgegriffen. In diesem Kapitel werden die vier häufigsten Abstraktionsstufen (Business-Model, Requirements-Specification, Component-Design und Object-Design) vorgestellt. Je nach Situation kann es in der Praxis vorkommen, daß mehr oder weniger Schichten benutzt werden.

### 2.2 UML & Running-Example

Als Notation schlägt der Catalysis-Approach UML vor. Dies ist eine graphische Sprache zur Beschreibung von Zuständen, Vorgängen und Abhängigkeiten, die von „den drei Amigos“ Booch, Jacobson und Rumbaugh entwickelt wurde. Die beiden wichtigsten UML-Komponenten (Use-Case-Diagramm und Sequence-Diagramm) werden hier zusammen mit dem fortlaufendem Beispiel, welches dieses Kapitel von Anfang bis Ende durchzieht, erklärt.

Zunächst gehen wir kurz auf eben erwähntes Beispiel ein. Für eine Seminar Firma soll eine Verwaltungssoftware erstellt werden. Die von den Kunden beantragten Kurse werden durchgeführt, wenn qualifizierte Lehrer zur Verfügung stehen. Die Qualifikation der Lehrer beruht auf Examen die sie ablegen müssen und auf den Resultaten bereits von ihnen gehaltener Kurse. Die zu erstellende Anwendung muß mit dem existierenden Urlaubsplaner und der vorhandenen Kundendatenbank zusammenarbeiten.

### 2.2.1 Use-Case-Diagram

Abbildung 2.1 zeigt das vorgestellte Beispiel als Use-Case-Diagramm. Solche Diagramme dienen dazu, Beziehungen und Eigenschaften in komplexen Systemen zu veranschaulichen. Man kann hier verschiedene Komponenten identifizieren. Die Strichmännchen heißen *Actors* und symbolisieren Rollen, die von Personen, Objekten oder ganzen Systemen eingenommen werden. Actors sind die Komponenten, die etwas tun. Die Ovale werden *Use Cases* genannt. Sie zeigen an, was getan wird und wer daran beteiligt ist. Die Quadrate sind die *Classes*. Sie stehen für passive Instanzen. Dahinter können abstrakte Begriffe oder komplexe Systeme stehen, die nicht näher erklärt werden sollen. Die Striche zwischen den Komponenten zeigen an, daß diese in Beziehung zueinander stehen. Grundsätzlich gilt für alle UML-Diagramme, daß Kommentare zur Erläuterung eingefügt werden sollten.

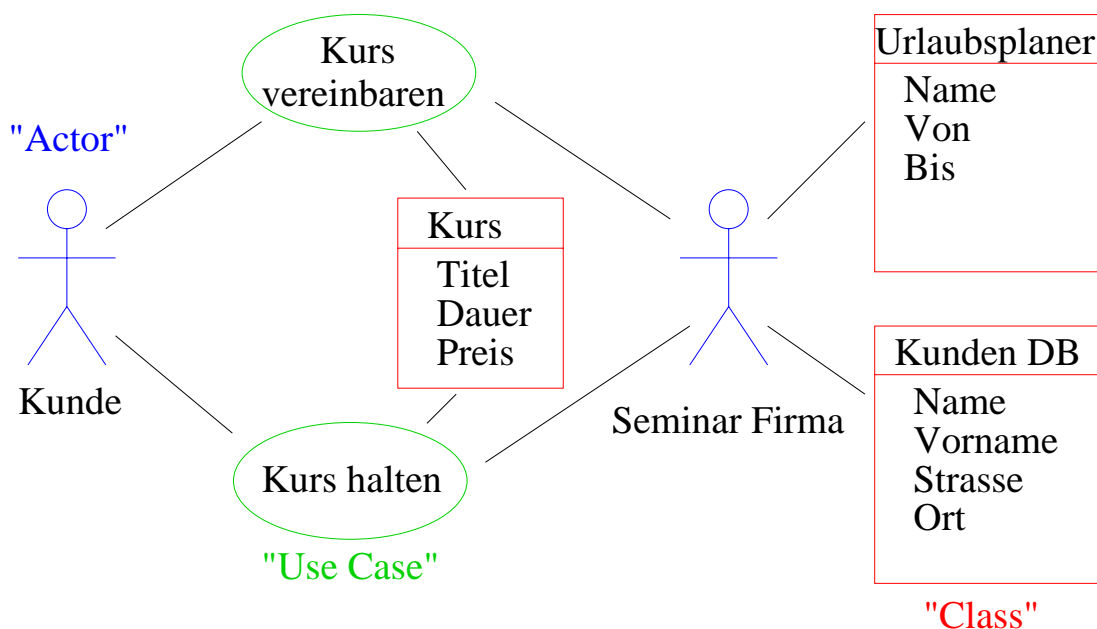


Abbildung 2.1: Use-Case-Diagramm zum Running-Example

Im Diagramm sieht man, daß der Actor Kunde an den Use-Cases Kurs vereinbaren und Kurs halten beteiligt ist. Diese beiden stehen mit der Class Kurs in Beziehung, welche die Attribute Titel, Dauer und Preis besitzt. Ebenso steht der Actor Seminar Firma mit beiden genannten Use-Cases in Verbindung. Er benutzt zur Bewältigung seiner Aufgaben zum einen die Class Urlaubsplaner mit den Attributen Name, Von und Bis, und zum anderen die Class Kunden DB mit den Attributen Name, Vorname, Straße und Ort.

### 2.2.2 Sequence-Diagram

Wie das zum Beispiel passende Sequence-Diagram aussieht (aussehen kann), zeigt Abbildung 2.2. Zweck dieses Diagrammes ist die Veranschaulichung zeitlicher Abfolgen. Wieder kann man verschiedene Komponenten identifizieren. Die senkrechten Balken stellen die interagierenden *Instances* dar. Wie man leicht bemerkt, sind dies die **Actors** und **Classes** (Ausnahme: Kurs) des vorhergehenden Use-Case-Diagramms. Der Balken ganz links, der als Mauer ausgeprägt ist, stellt die *Systemborder* dieses Systems dar. Dies bedeutet, sie ist die Schnittstelle (und damit Anfang und Ende) des in diesem Diagramm beschriebenen Vorgangs. Das Mauersymbol ist nicht Teil des UML-Standards, sondern wurde zur Verdeutlichung eingeführt. Die Pfeile zwischen den Instanzen sind *Actions*, die zwischen diesen ausgeführt werden. Die Zeitachse verläuft von oben nach unten. Folgt man den Pfeilen (links oben beginnend) durch das Diagramm, so folgt man genau dem zeitlichen Ablauf des Vorgangs.

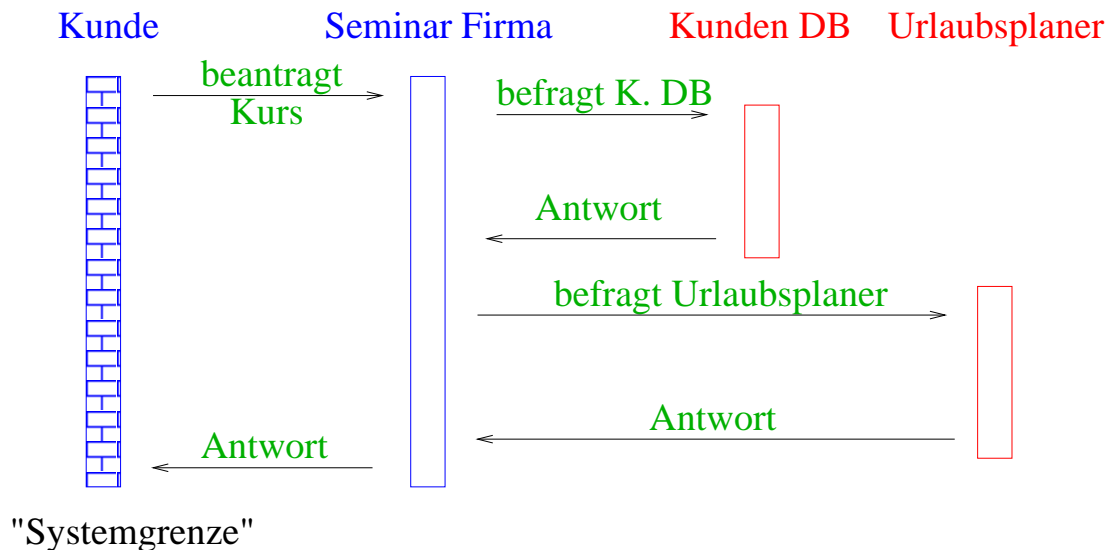


Abbildung 2.2: Sequence-Diagram zum Running-Example

Hier ist die Instanz Kunde die Systemgrenze. Diese löst zu Beginn die Aktion *beantragt Kurs* gegenüber der Instanz *Seminar Firma* aus. Letztere startet die Aktion *befragt K.DB* mit der Instanz *Kunden DB*. Nach Erhalt der Aktion *Antwort* löst sie die Aktion *befragt Urlaubsplaner* auf der Instanz *Urlaubsplaner* aus. Das Auslösen der Aktion *Antwort* veranlaßt die Instanz *Seminar Firma* schließlich die Aktion *Antwort* gegenüber der Instanz *Kunde* zu starten.

## 2.3 Business-Model

Die erste Stufe des Catalysis-Approach wird manchmal auch als *Domain* oder *Essential-Model* bezeichnet. Sie beschreibt die Welt des Benutzers ohne jegliche Notation aus dem Bereich des Informatikers. Ziel ist es, zu beschreiben, wie sich das (zu entwickelnde) System in die Welt des Benutzers einfügt. Dieser ist auch der alleinige Spezifizierer des Modells, während der Informatiker durch gezielte Fragen versucht sich — und auch dem Benutzer — klar zu machen, was erwartet wird. Typische Fragen des Entwicklers an den Benutzer in dieser Phase sind: „Was machen Sie?“ und „Mit wem interagieren Sie dabei?“

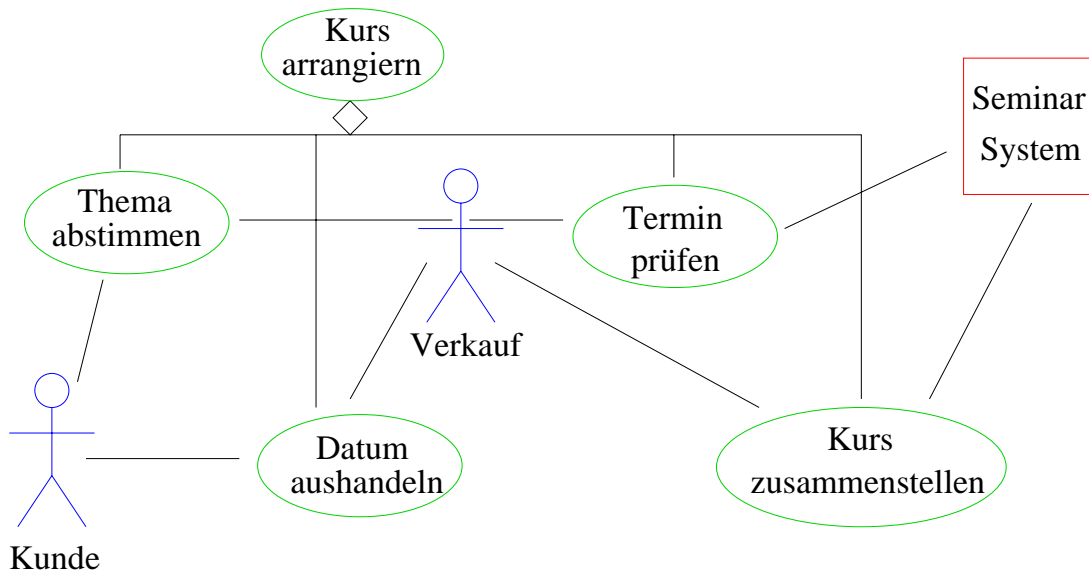


Abbildung 2.3: Use-Case-Diagramm zum Business-Model

Im Beispiel (Abbildung 2.3) stellt ein Verkäufer dar, wie das Arrangieren eines Kurses abläuft, und wo er dabei von dem neuen System unterstützt werden soll: „Um einen Kurs zu arrangieren, muß ich zunächst mit dem Kunden ein Thema abstimmen. Danach wird das Datum ausgehandelt, an dem der Kurs stattfinden soll. Um dies tun zu können, muß ich mögliche Termine prüfen, dabei soll mich das Seminar System unterstützen. Ist alles geklärt, so ist es meine Aufgabe, dafür zu sorgen, daß der Kurs zusammengestellt wird. Auch dabei möchte ich die Hilfe des neuen Systems in Anspruch nehmen.“

## 2.4 Requirements–Specification

Hier geht es darum, die Situation der Anwendung zu beschreiben. Auch hier wird keinerlei (aus Sicht des Benutzers) fachfremde Sprache verwendet. Ziel ist es, zu beschreiben, wie sich die Benutzer in die Welt der Applikation einfügen. Wieder ist der Benutzer der Spezifizierer, jedoch stellt der Entwickler gezieltere Fragen und zeigt Grenzen auf. Typische Fragen sind: „Was muß das Programm leisten?“ und „Wer benutzt es?“

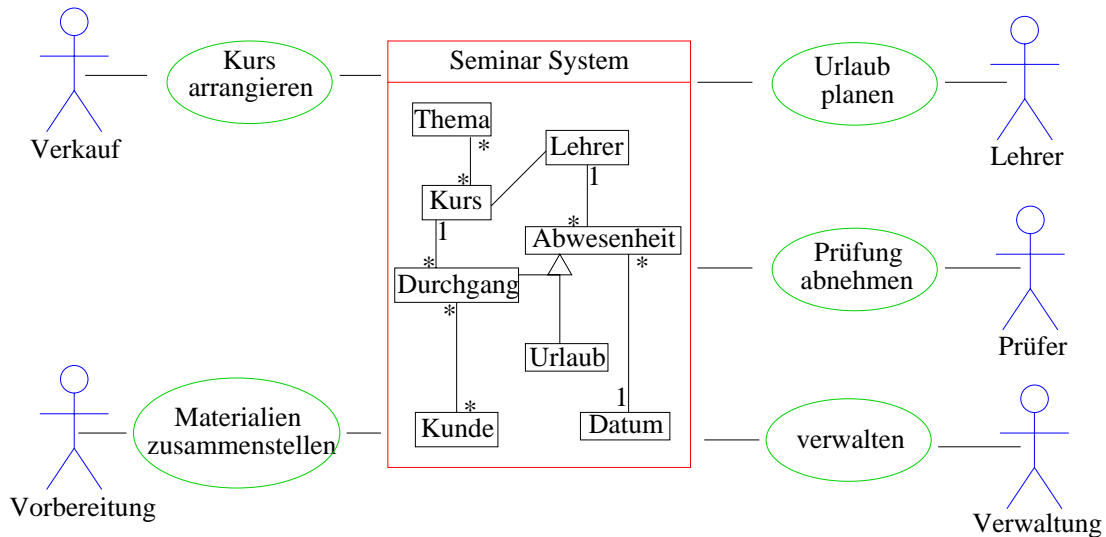


Abbildung 2.4: Use–Case–Diagram zur Requirements–Specification

Abbildung 2.4 zeigt das zugehörige Beispiel. Das System wird vom Verkauf benutzt, um seine Kurse zu arrangieren. Die Vorbereitung bedient sich des Systems, um die zu einem Kurs gehörigen Materialien zusammenzustellen. Die Lehrer planen damit ihre Termine und den Urlaub. Dem Prüfer ist es bei der Abnahme der Prüfungen behilflich. Auch die Verwaltung greift auf das System zu, um die anfallenden Arbeiten zu tätigen. Innerhalb des Systems sind die Beziehungen der einzelnen zu erledigenden Aufgaben bereits im Groben beschrieben.

## 2.5 Component-Design

Zentrum dieser Schicht sind die Komponenten. Es geht darum herauszufinden, welche Funktionalitäten sich in Komponenten kapseln lassen, und wie verschiedene Komponenten zusammen arbeiten. Um verschiedene Versionen eines Programmes zu erstellen ist es nötig, daß die Komponenten austauschbar sind. Diese Stufe spezifiziert der Entwickler in Informatikersprache. Die Ergebnisse werden dem Benutzer (in seiner Sprache) vorgelegt, und mit ihm diskutiert.

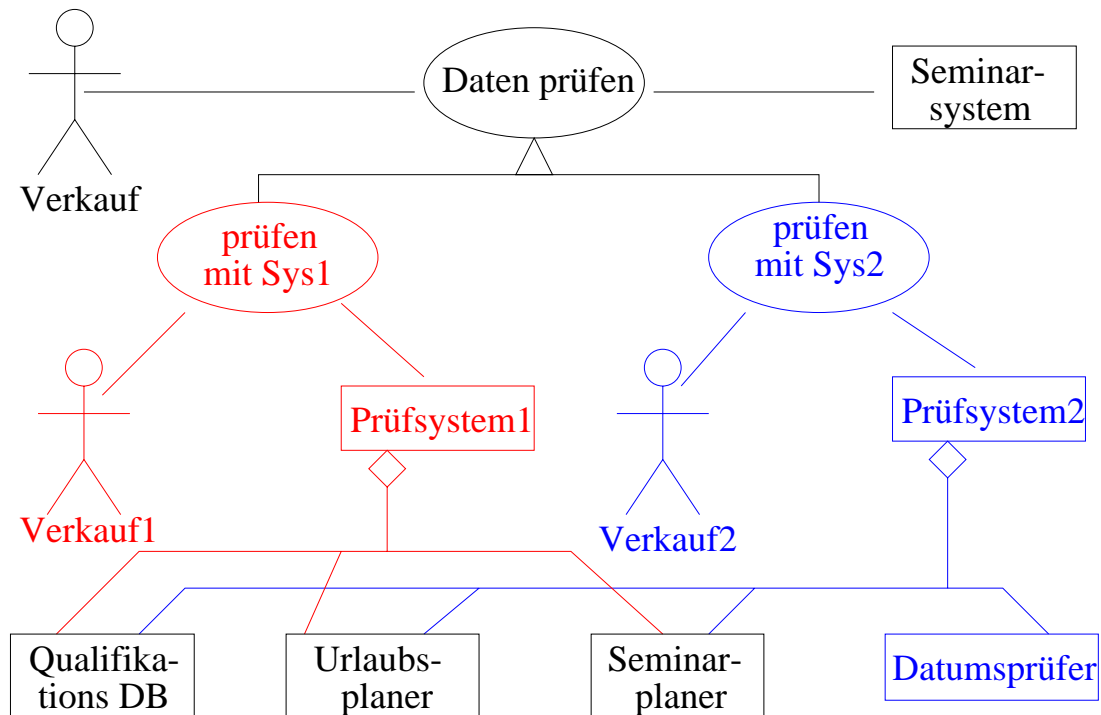


Abbildung 2.5: Use-Case-Diagramm zum Component-Design

In Abbildung 2.5 sehen wir das Use-Case-Diagramm zum Beispiel. Die Software soll in zwei Versionen ausgeliefert werden. Eine **einfache Variante** (alle Komponenten deren Name mit 1 endet), bei der Verkauf die Ergebnisse aus den Anfragen bei der Qualifikations DB, dem Urlaubsplaner und dem Seminarplaner selbst vergleichen muß. Des weiteren existiert eine **komplexe Variante** (alle Komponenten deren Name mit 2 endet und der Datumsprüfer), bei der der Datumsprüfer den Vergleich übernimmt. Es ist leicht zu sehen, daß alle übrigen Komponenten von beiden Versionen gleichermaßen benutzt werden.

Verkauf                      **Datumsprüfer**                      Quali DB                      Urlaubsplaner                      Seminarplaner

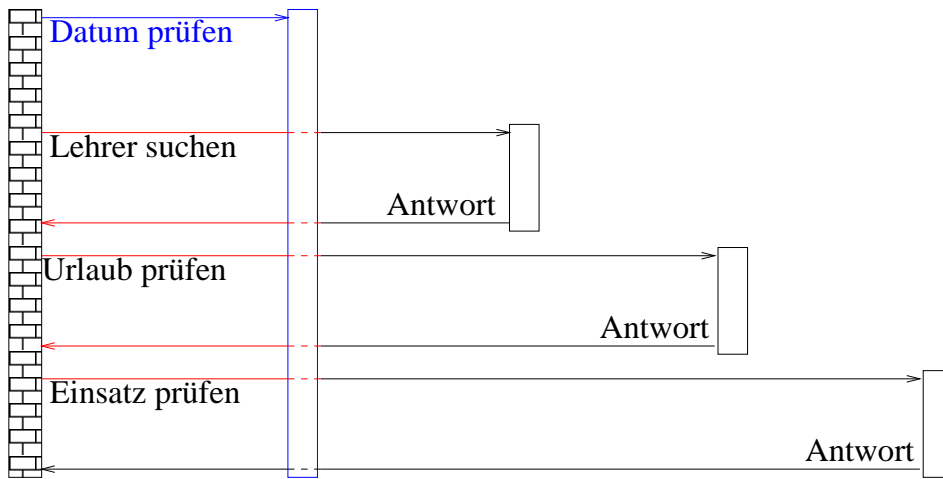


Abbildung 2.6: Sequence-Diagramm zum Component-Design

Um den Unterschied zwischen den Varianten zu verdeutlichen, sind die beiden zugehörigen Sequenzdiagramme zu einem (Abbildung 2.6) zusammengefasst. Die Instanzen Verkauf, Quali DB, Urlaubsplaner und Seminarplaner gehören zu beiden Versionen. Die **einfache Variante** arbeitet ohne die Instanz Datumsprüfer und der Verkauf muß nacheinander die Aktionen Lehrer suchen, Urlaub prüfen und Einsatz prüfen ausführen und die Ergebnisse vergleichen. In der **komplexen Variante** muß lediglich die Aktion Datum prüfen auf der Instanz Datumsprüfer aufgerufen werden, und dieser übernimmt die anderen Aufrufe und den Vergleich.

## 2.6 Object-Design

Dies ist die abschließende Schicht. In ihr wird die geforderte Funktionalität auf Codeebene beschrieben. Die Beschreibung soll so formal sein, daß entsprechende Werkzeuge die Diagramme in Coderümpfe in der Zielsprache umwandeln können. In diesen Hüllen liegen die Klassen- und Methodensignaturen dann bereits vollständig vor. Diese Ebene ist alleiniges Terrain des Entwicklers.

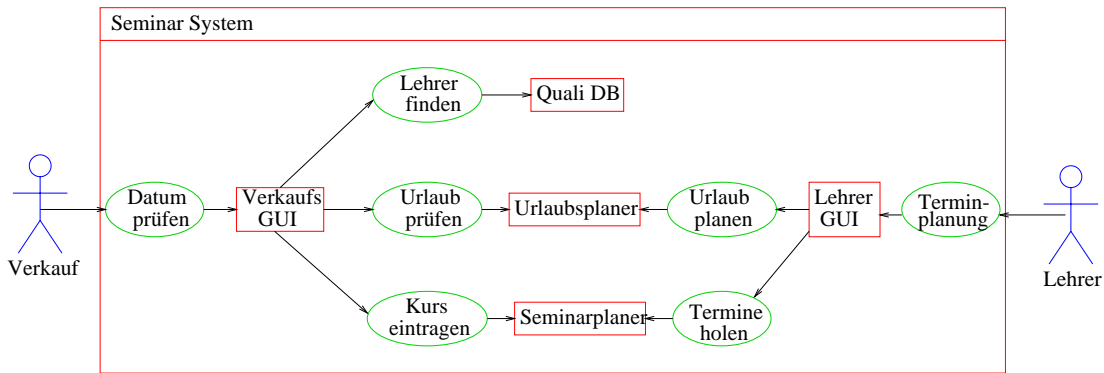


Abbildung 2.7: Use-Case-Diagramm zum Object-Design

Abbildung 2.7 zeigt das Use-Case-Diagramm in dem die Benutzung des Systems durch den Verkauf und den Lehrer dargestellt ist. Der Verkauf bekommt beim Datum prüfen die Verkaufs GUI angezeigt. Zu Beginn muß er einen Lehrer finden. Dazu greift das System auf die QualiDB zu, während zum Urlaub prüfen der Urlaubsplaner konsultiert wird. Beim Kurs eintragen wird der Seminarplaner benutzt. Ein Lehrer hingegen bekommt die Lehrer GUI angezeigt. Dort kann er seinen Urlaub planen, wozu das System den Urlaubsplaner verwendet, oder seine Termine holen, was zur Benutzung des Seminarplaners führt.

Verkäufer      Verkaufs GUI      QualiDB      Urlaubsplaner      Seminarplaner

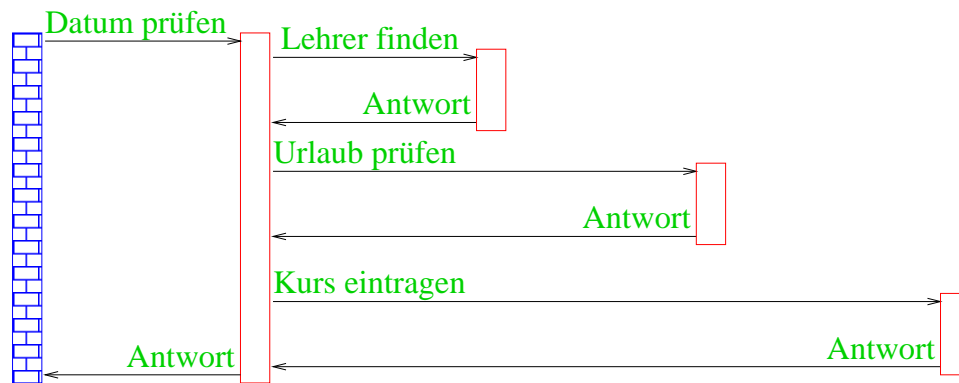


Abbildung 2.8: Sequence-Diagramm zum Object-Design: Actor Verkauf

Das zum Verkauf gehörige Sequence-Diagramm ist in Abbildung 2.8 zu sehen. Will der Verkauf ein Datum prüfen, muß er über die Verkaufs GUI in der QualiDB einen Lehrer finden, danach im Urlaubsplaner einen eventuellen Urlaub prüfen und schließlich im Seminarplaner den Kurs eintragen.

Lehrer      Lehrer GUI      Urlaubsplaner      Seminarplaner

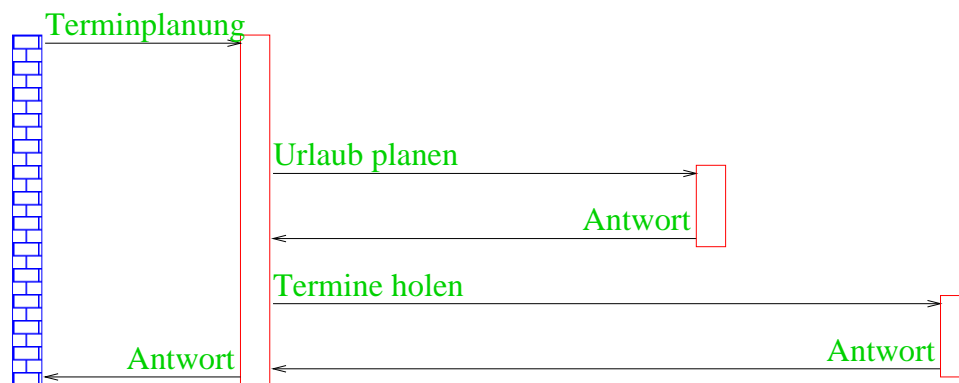


Abbildung 2.9: Sequence-Diagramm zum Object-Design: Actor Lehrer

In Abbildung 2.9 sehen wir das zum Lehrer gehörige Sequence-Diagramm. Über die Lehrer GUI macht er seine Terminplanung. Zunächst möchte er seinen Urlaub planen, dazu benutzt das System den Urlaubsplaner. Will er seine Termine holen, so wird dazu der Seminarplaner benutzt.

# Kapitel 3

## Grundprinzipien

Der im vorhergehenden Kapitel vorgestellte Catalysis–Approach basiert auf den Prinzipien Abstraktion, Präzision und Komponenten.

### 3.1 Abstraktion

Abstraktion bedeutet, daß nur relevante Aspekte in einer Beschreibung Beachtung finden, alle anderen Aspekte bleiben unbeachtet. Trotz der häufig anzutreffenden Assoziation von abstrakt mit esoterisch, akademisch oder einfach nicht anwendbar, spielt die Abstraktion eine wichtige Rolle im Catalysis–Approach. Erst die Anwendung der Abstraktion ermöglicht es, die weitreichenden Anforderungen überschaubar und die zu treffenden Architekturentscheidungen behandelbar zu machen. Ohne Abstraktion gäbe es keine Modelle auf den verschiedenen Ebenen von den Business–Rules bis zum Code. Ebenso würde die methodische Verfeinerung fehlen.

Insgesamt ermöglicht es die Abstraktion, mit den Sachen anzufangen, die zuerst erledigt werden müssen.

### 3.2 Präzision

Da die natürliche Sprache und auch ad–hoc–Diagramme nicht präzise sind, Code aber schon, muß man von Anfang an Inkonsistenzen und Zweideutigkeiten vermeiden. Alle Spezifikationen müssen präzise genug sein, um (im Fehlerfall) widerlegbar zu sein. Nur durch Präzision kann erreicht werden, daß Anforderungen durch alle Modellebenen verfolgbar sind. Auch der Einsatz von Tools auf der semantischen Ebene ist nur auf der Basis einer präzisen Spezifikation möglich.

### **3.3 Komponenten**

Der Catalysis-Approach unterstützt den Entwickler beim Design von Komponenten und deren Komposition.

Durch das Benutzen von Komponenten wird die Software flexibler und man vermeidet Fehler, wenn man auf bereits existierende (und getestete) Komponenten zurückgreift. Zusätzlich wird die Entwicklung von Software beschleunigt. Dabei kann man nicht nur Klassen, sondern auch Frameworks, Patterns und Spezifikationen als Komponenten wiederbenutzen.

Der Gewinn aus dem Design-Aufwand wird durch Wiederverwendung von Komponenten noch gesteigert.

# Kapitel 4

## Resümee

Hier soll noch einmal deutlich gemacht werden, wo die Vorteile des Catalysis–Approach liegen und was wir von diesem Ansatz halten.

### 4.1 Vorteile des Catalysis–Approach

Um zu sagen, wo die Vorteile des Catalysis–Approach liegen, muß man zwischen den verschiedenen Entwicklungsarten unterscheiden.

#### 4.1.1 Komponentenbasierte Entwicklung

Der Catalysis–Approach unterstützt bei der präzisen Definition von Schnittstellen, die trotz der Präzision immer noch implementierungsunabhängig sind. Desweiteren wird erklärt, wie man eine Komponentenarchitektur und die Verbindungen zwischen Komponenten konstruieren sollte. Und schließlich hilft er bei der Sicherstellung, daß eine Komponente die geforderten Schnittstellen besitzt, d.h. zu den vorher definierten Komponentenverbindungen paßt.

#### 4.1.2 High–integrity Entwicklung

Der Catalysis–Approach verhilft zu einer präzisen, abstrakten Spezifikation. Er ermöglicht die eindeutige Nachvollziehbarkeit von den Business–Goals zum Programmcode.

#### 4.1.3 Objekt–orientierte Entwicklung

Im Catalysis–Approach wird eine use-case-basierte Technik für das Vorgehen vom Business–Modell zum objekt-orientierten Code benutzt.

### **4.1.4 Reengineering**

Mit dem Catalysis–Approach erhält man Techniken zum Verstehen existierender Software und zum Entwickeln neuer Software aus der vorhandenen.

## **4.2 Fazit**

Durch die im Catalysis–Approach benutzte Technik der Modell–Ebenen erhält der Informatiker eine präzise Spezifikation der Vorgehensweise. Durch die allgemeine Verständlichkeit der ersten Ebenen, dort wird keinerlei „Informatikersprache“ verwendet, kann der Benutzer in die Analyse und das Design miteinbezogen werden und von Anfang an das Produkt mitgestalten.

Aus den eben aufgeführten Gründen sind wir der Meinung, daß der Catalysis–Approach ein gelungenes Vorgehensmodell ist.

# Literaturverzeichnis

- [BJR97a] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language, Notation Guide*. 1997.
- [BJR97b] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language, Semantics*. 1997.
- [DW98] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
- [FS97] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.
- [Hen99] W. Henhapl. *Vorlesungsunterlagen Software Engineering*. TU Darmstadt, FG Praktische Informatik, 1999.

# Index

- Abstraktion, 8, 17
- Action, 10
- Actor, 9
- Analyse, 5, 6, 20
- Anforderungen an Software, 4
  
- Booch, 8
- Business-Model, 8, 11
  
- Catalysis-Approach, 8
- Class, 9
- Component-Design, 8, 13
  
- Design, 5, 6, 20
- die drei Amigos, 8
- Domain, 11
  
- Einleitung, 4
- Entwicklungsschichten, 8
- Essential-Model, 11
  
- Fazit, 20
- Flexibilität, 4
  
- Grundprinzipien, 17
  
- High-integrity Entwicklung, 19
  
- Implementierung, 5, 6
- Instance, 10
- Integrationstest, 6
- Integrität, 4
  
- Jacobson, 8
  
- Komponenten, 17, 18
- Komponentenarchitektur, 19
- Komponentenbasierte Entwicklung, 19
- Komponententest, 6
  
- Korrektheit, 4
  
- Notation, 8
  
- Object-Design, 8, 15
- Objekt-orientierte Entwicklung, 19
  
- Phasen der Software-Entwicklung, 5
- Präzision, 8, 17
- Prototyp, 7
  
- Reengineering, 20
- Requirements-Specification, 8, 12
- Resümee, 19
- Rollen, 9
- Rumbaugh, 8
- Running-Example, 8
  
- Sequence-Diagram, 8, 10
- Spiralmodell, 7
- Systemborder, 10
- Systemtest, 6
  
- Team-Entwicklung, 4
- Test, 5
  
- UML, 8
- Use-Case, 9
- Use-Case-Diagram, 8-10
  
- V-Modell, 6
- Verifikation, 7
- Vollständigkeit, 4
- Vorgehensmodelle, 5
- Vorteile des Catalysis-Approach, 19
  
- Wasserfallmodell, 5
  
- Zielsprache, 15