

Vorlesungsmodul Softwaretechnik 1

- VorlMod SoftwareTk1 -

Matthias Ansorg

1. Oktober 2001 bis 26. Mai 2003

Zusammenfassung

Studentische Mitschrift zur Vorlesung Softwaretechnik 1 bei Prof. Letschert (Wintersemester 2001/2001) im Studiengang Informatik an der FH Gießen-Friedberg, Studienort Gießen. Zur Klausurvorbereitung sind ergänzend zu dieser Mitschrift die Lösungen der restlichen Übungsaufgaben und die Dokumente zur Vorlesung von Professor Letschert nötig (bisher [20, 21, 22, 4, 5, 6, 7, 8, 9, 10, 15, 16, 17, 18, 19,]). Allerdings hätte zum Bestehen der Klausur vom 2001-01-25 ohne weiteres auch nur diese Mitschrift und [13] genügt, sofern man die Übungsaufgaben konnte.

- **Bezugsquelle:** Die vorliegende studentische Mitschrift steht im Internet zum Download bereit: <http://homepages.fh-giessen.de/~hg12117/index.html>. Wenn sie vollständig ist, kann sie auch über die Skriptsammlung der Fachschaft Informatik der FH downgeloadet werden.
- **Lizenz:** Diese studentische Mitschrift ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der verwendeten Quellen zu beachten.
- **Korrekturen:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg, ansis@gmx.de.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX (graphisches Frontend zu L^AT_EX) unter Linux erstellt und als PDF-Datei exportiert.
- **Dozent:** Prof. Letschert.
- **Klausur:** Es dürfen alle Unterlagen benutzt werden; man muss also nichts auswendig können. Wer kein UML-Buch hat, sollte sich für die Klausur eine kleine UML-Referenz erstellen oder besorgen, um darin die Notation nachzuschlagen; eine gute Idee ist das kostenlose Dokument [13]. Klausurstruktur nach Angaben von Prof. Letschert:
 - Man sollte die wichtigen Begriffe der Softwaretechnik vernünftig erklären können, z.B. »Was ist die Analysephase?«. Siehe [15, Aufg. 3,4]. Die Antworten zu diesem ersten Übungsblatt muss man jedoch nicht auswendig lernen, da man Unterlagen verwenden darf.
 - »Definieren Sie Aktoren und Anwendungsfälle für folgende Problemstellung: [...]« (Beispielaufgabe).
 - »Entwerfen Sie ein angepasstes Wasserfallmodell für folgende Aufgabenstellung und geben Sie die Argumente für ihr Verfahren im Gegensatz zum konventionellen Wasserfallmodell an.« (Beispielaufgabe)
 - Grundbegriffe der Komponentenerlegung an einem kleinen Problem anwenden können: nach Wissen und Können in Teilkomponenten zerlegen, Schnittstellen definieren. Diese Aufgaben sind weniger kompliziert als die schwierigen Beispiele in der Vorlesung. Es gibt mögliche Variationen der Antworten.
 - »Erläutern Sie folgendes UML-Diagramm: [...]« (Beispielaufgabe)
 - »Setzen Sie folgendes einfache UML-Diagramme in C++ um: [...]« (Beispielaufgabe). Dies kann sich auf die Umsetzung eines Komponentendiagramms in eine Dateistruktur aus .h- und .c-Dateien und Sourcecode beziehen, auf die Umsetzung eines Klassentemplates.
 - »Modellieren sie folgende einfache Aufgabenstellung in einem geeigneten UML-Diagramm« (Beispielaufgabe, für einfache Probleme wie das Mutter-Kind-Problem). Man sollte die UML-Notation beherrschen und bei der Modellierung eine ggf. nötige Konzepttrennung durchführen und die Interpretation von Relationen in Tabellen verstehen. Konzepttrennung ist z.B. die Trennung in einen geometrischen Ausdruck (einen **string**) und einen geometrischen Wert (die Objekte, die man dann abspeichert).

- Umsetzung von einer Softwarestruktur in Quelldateien (Komponentenaufteilung in `.c` und `.h`), ein Makefile dazu schreiben. Es ist also Stoff von ProgII notwendig, denn SoftwareTk1 hängt damit recht eng zusammen.
- »Muss eine Methode `void push(int)`, zu der in einem Kommentar die Vorbedingung `//PRE: Stack ist nicht voll` angegeben ist, selbst überprüfen, ob der Stack voll ist? Wenn nein, wer muss dann dafür sorgen?«
- Wichtig zum Lernen für die Klausur sind die Übungsaufgaben, besonders die letzten Übungsblätter.
- Unerscheiden: `error`, `fault`, `failure`
- Unterscheidung: `black-box Test`, `white-box Test`.
- Tests mit Äquivalenzklassen und Pfadüberdeckung an einem einfachen Programm entwickeln.
- Geben Sie ein UML-Diagramm und eine Implementierung in C++ an für eine Warteschlange (FIFO-Prinzip) im Sinne einer UML-Komponente.
- Komponentenimplementierung heißt in der Klausur: welche Dateien haben welchen Inhalt. Die Algorithmik ist dabei eher nebensächlich.

- **Verwendete Quellen:** .

Inhaltsverzeichnis

1	Entwicklung der Softwaretechnik	4
1.1	Definition	4
1.2	Softwareproduktion ist Industrie	4
1.3	Softwareproduktion ist anders	5
1.4	Die Softwarekrise	5
2	Das Wasserfallmodell	6
2.1	Analyse	6
2.1.1	Planungsphase	6
2.1.2	Anforderungsanalyse und Analysemodell	6
2.2	Entwurf	7
2.2.1	Grobentwurf	7
2.2.2	Feinentwurf	9
2.3	Implementierung	9
2.4	Test und Integration	9
2.4.1	Verifikation	9
2.4.2	Validierung	11
2.4.3	Konfigurationsmanagement	11
2.4.4	Tools zum Konfigurationsmanagement	11
2.5	Meilensteine (Zusatz)	12
2.6	Projekte (Zusatz)	12
3	Strukturierte Programmierung	14
3.1	Was ist strukturierte Programmierung?	14
3.2	Spezifikation sequentieller Programme	14
3.3	Das Hoare-Kalkül	15
3.4	Strukturierte Programmierung in der Praxis	16
3.5	Spezifikation objektorientierter Programme	16
4	UML-Diagramme	18
4.1	Einsatz von UML	18
4.2	Anwendungsfalldiagramme	18
4.3	Aktivitätsdiagramme	19
4.4	Komponentendiagramme	19
4.4.1	Komponentenbegriff der UML	19
4.4.2	Komponente i.S.v. UML versus Komponente i.S.v. OOP (Beispiel)	19
4.4.3	Schnittstellen in UML	21
4.5	Klassendiagramme und Objektdiagramme	22

4.5.1	Klasse	23
4.5.2	Objekt	24
4.5.3	Assoziation	24
4.5.4	Abhängigkeit	24
4.5.5	Aggregation	25
4.5.6	Komposition	25
4.5.7	Kardinalität	26
4.5.8	abstrakte Basisklasse	26
4.5.9	Vererbung	26
4.5.10	Schnittstelle	28
4.5.11	Template	28
4.6	Sequenzdiagramme	28
4.6.1	Beispiel: Ausdrucksanalyse mit Bäumen	29
5	Glossar	30
6	Durchgängiges Fallbeispiel zum Wasserfallmodell	33
6.1	Analyse	33
6.1.1	Planungsphase	33
6.1.2	Anforderungsanalyse und Analysemodell	33
6.2	Entwurf	35
6.2.1	Grobentwurf	35
6.2.2	Feinentwurf	38
6.3	Implementierung	38
6.3.1	Komponente internSpeicher	38
6.3.2	Komponente Syntax (Prototyp)	39
7	Übungen und Lösungen	41
7.1	Blatt 4 Aufgabe 1	41
7.2	Blatt 4 Aufgabe 2	42
7.3	Blatt 4 Aufgabe 3	44
7.4	Blatt 4 Aufgabe 4	45
7.5	Blatt 4 Aufgabe 5	47
7.6	Blatt 4 Aufgabe 6	49
7.7	Komponentenaufteilung: System zur Auswertung geklammerter Ausdrücke	50

Abbildungsverzeichnis

1	Architekturmuster einer einfachen unverteilter interaktiven Anwendung	8
2	Reale Organisation der Softwareproduktion	13
3	Skizze der Beziehung zwischen konkretem und abstraktem push	18
4	Die Schnittstelle »Auto«	22
5	Synonyme Möglichkeiten zur Darstellung eines Interface in UML	23
6	Notation von Klassen in UML	24
7	Notation von Objekten in UML	24
8	»Aggregation zwischen Meier und Bein«	25
9	»Komposition zwischen Meier und Bein«	26
10		26
11		27
12		27
13	UML-Diagramm eines Klassentemplate	28
14	Ausdrucksauswertung in Bäumen, Variante 1.	29
15	Ausdrucksauswertung in Bäumen, Variante 2.	30
16	Datenflussdiagramm: Notation an einem Beispiel	34
17	Konzeptionelles Modell zu Hausübung 4 (Klassendiagramm)	35
18	Aktivitätsdiagramm zu Hausübung 4	35
19	Komponentenaufteilung in UML	37

20	Klassendiagramm der Interface-Klassen zur Stack-Komponente	40
21	Vollständig geklammerte binäre Ausdrücke in einem SyntaxBaum	40
22	Sequenzdiagramm zu Übungsblatt 4 Aufgabe 1	42
23	Warteschlange als UML-Komponente	45
24	Klassenstruktur zu Übungsblatt 4, Aufgabe 5	49
25	50
26	50
27	50
28	51
29	52

1 Entwicklung der Softwaretechnik

1.1 Definition

»Softwaretechnik ist die rationale Gestaltung aller organisatorischen und technischen Aspekte des Prozesses der Softwareproduktion.« [4]. Es geht darum, Software industriell und ingenieurmäßig zu produzieren: der Unterschied, den hobbymäßiges Programmieren von Softwareproduktion gegen Bezahlung unterscheidet. Softwaretechnik ist in der Industrie weit wichtiger als die im Studium gelernte Mathematik. Denn wenn in der Industrie überhaupt Mathematik gebraucht wird, so ist sie deutlich komplizierter als die im Studium behandelte.

Manche bezeichnen Softwaretechnik als »Laberfach«. An den behandelten Übungsaufgaben zeigt sich: zu vielen gibt es keine eindeutige Lösung, man muss ein Problembewusstsein lernen.

1.2 Softwareproduktion ist Industrie

Eine unterschätzte Industrie. Nach einer Untersuchung von 1994 über die Berufssituation von Informatikern gab es 1994 25000 arbeitende Informatiker und 60000 Informatik-Studenten. Personalchefs sagten zu diesem Zeitpunkt, dass man »richtige« Informatiker nicht brauche, nur Anwendungsinformatiker. Die Folge war, dass die Zahl der Informatikstudenten drastisch sank. Dies war eine völlige Fehleinschätzung des zukünftigen Bedarfs an Informatikern und der Entwicklung der Informatik; man unterschätzte die Schwierigkeit der Softwareproduktion. Dies lag daran, dass es fast keine Informatiker in Führungspositionen gab. Weiter sagte die Studie, dass die Aufstiegschancen für Informatiker schlecht waren. Derzeit wird bzgl. des Bedarfs an Informatik in der Gegenrichtung übertrieben . . .

Die aktuelle Situation. Eine aktuelle Studie mit Datenstand 2000 besagt (genauere Darstellung in [4]):

- Es gibt in Deutschland 19200 Unternehmen, die sich mit Entwicklung und Anpassung von Software beschäftigen, davon 10000 Softwarehersteller.
- Der Umsatz mit Softwareproduktion und -anpassung beträgt $55 \cdot 10^9$ DM (1,4% des BIP), demgegenüber der der Landwirtschaft $42 \cdot 10^9$ DM.
- Produkte
 - 55% der Betriebe stellen betriebswirtschaftliche Software her oder passen sie an (SAP-Systeme);
 - 46% der Betriebe beschäftigen sich mit Internet und Multimedia (Daten aus dem »dot com«-Boom von 2000; seither gab es zwar viele Pleiten, jedoch ist die Situation auf dem Markt nicht so schlecht wie die entsprechenden Aktionkurse).
 - 33% erstellen technische Software;
 - 16% erstellen Systemsoftware.
- Standorte: viele Firmen produzieren Software im In- und Ausland oder nur im Ausland. Wichtig für deutsche Informatiker sind deshalb:
 - Fremdsprachenkenntnisse
 - »interkulturelle Kommunikationsfähigkeiten«

– Führungsfähigkeiten und Projektmanagement zur Leitung der ausländischen Softwareproduktion.

- Im Jahr 2000 arbeiteten 177000 Personen in der Softwareproduktion, man erwartete einen Zuwachs von 180000 bis 2005. 2001 wurden ungefähr soviele Greencards verteilt wie es Informatikabsolventen gab. 36% der Firmen der primären Branche und der sekundären Branche wollten FH-Absolventen haben. Man sucht jedoch auch (aber weniger) Uni-Absolventen, die die Probleme abstrakt erfassen können. Eine Konkurrenz für FH-Absolventen sind Abgänger von Berufsakademien. 20% der Firmen wollen ihre Arbeiter selbst ausbilden und setzen gar keine Abschlüsse voraus.

Schlussfolgerungen.

- Die Industrieinformatik braucht zunehmend in Informatik ausgebildete Führungskräfte.
- Die Struktur der Nachfrage nach Informatikern ist standortspezifisch für Deutschland: die Stärken der deutschen Industrie sind Maschinenbau und Chemie in mittelständischen Unternehmen und der Dienstleistungssektor. Entsprechend sind die Tätigkeiten der Informatiker: Anpassung von Software und Kundenberatung im Dienstleistungssektor, Produktion technischer Software als Teil des Produktes (d.h. eingebettete Systeme, SPS) im Maschinenbau.
- Personalkosten: Sie sind in Deutschland extrem hoch. Deshalb wird Software rationell verteilt in verstreuten Teams entwickelt. Dazu sind Kenntnisse über Komponententechnologie und Standards nötig (CORBA¹, DirectX). Dies entspricht der generellen Tendenz, Software auf einem relativ hohen Abstraktionsniveau zu entwickeln.

1.3 Softwareproduktion ist anders

Sie muss rationell und technisch hochwertig sein, wie die Produktion aller materiellen Güter. Jedoch ist sie irgendwie anders: Software ist immateriell, d.h. die Reproduktionskosten sind nahe null. Die Kosten für Software bestehen also rein in den Kosten für ihre Entwicklung. Softwareproduktion ist auch heute noch nicht wirklich ingenieurmäßig, d.h. sie ist »unprofessionell«: es besteht (fast) keine Trennung / Arbeitsteilung zwischen Handwerker, Ingenieur und Techniker, wie z.B. beim Ingenieurbau. Dies ist ein massives Problem der Informatik, denn es verunmöglicht eine angepasste Ausbildung. Es liegt daran, dass die Mitteilung, wie ein Problem im Programm gelöst werden soll, fast nur durch die Programmiersprache selbst möglich ist.

1.4 Die Softwarekrise

Die unter Kapitel 1.3 beschriebene Eigenart der Softwareproduktion führte 1964 in Garmisch-Patenkirchen zur sogenannten »Softwarekrise«. Das heißt, auf einer Konferenz gelangte man zum gemeinsamen Ergebnis, dass die bisherige Softwareproduktion so nicht weitergehen könne.

Die mit der »Softwarekrise« adressierten Probleme:

- Um 1960 wurde Programmieren als eine Kunst angesehen, d.h. jeder arbeitete nach seinem eigenen kreativen Verfahren. Jedoch müssen Programme im Gegensatz zu Kunstwerken auch funktionieren, und das taten sie oft nicht.
- Softwareentwicklung funktionierte einfach nicht: Die Programme wurden zu spät fertig und taten dann immer noch nicht das, was sie tun sollten.

Die Ursachen der Softwarekrise:

- Es gab keine richtige Ausbildung von Programmierern.
- Software wurde nach Art von Hobbyprogrammierern entwickelt.

Die Antworten auf die Softwarekrise:

- Industrielle Softwareproduktion: Die Softwaretechnik entstand und führte Management-Techniken wie das Wasserfall-Modell ein, um die geforderte Professionalisierung der Software zu erreichen. Zu diesem Ansatz vergleiche Kapitel 2.
- Wissenschaftliche Softwareproduktion: Dieser Ansatz wurde von Dijkstra begründet. Vergleiche dazu Kapitel 3.

¹Common Object Request Broker Architecture

2 Das Wasserfallmodell

Das Wasserfall-Modell versucht einfach, die Teilschritte der Softwareproduktion in einer logischen Abfolge (»wie ein Wasserfall«) zu definieren. Es ist jedoch nur wenig realistisch, weil z.B. ein nicht realisierbarer Entwurf erstellt werden kann. Das Wasserfallmodell wurde vielfältig verfeinert (z.B. feinere Phasenteilung) und es wurden zyklische Modelle entwickelt (mit Rückkopplungen: Wiederholung eines fehlgeschlagenen Schrittes).

Neben dem Wasserfallmodell entwickelten sich andere Ansätze wie die Entwicklung über phasenlos entwickelte Prototypen oder das extreme programming (sehr qualifizierte Leute einfach machen lassen). In der realen Softwareproduktion wird immer die ein oder andere Form des Wasserfallmodells verwendet. Die Phasen des Wasserfallmodells sind:

2.1 Analyse

Auch »Analysephase« genannt. Es ist nicht unwahrscheinlich, dass begabte Informatiker(-innen) später hauptsächlich in der Analyse arbeiten. Die Analyse eines größeren Projektes kann 7 Informatiker 1 Jahr beschäftigen und 4 Millionen Mark kosten.

In diesem ersten Schritt überlegt man, was zu tun ist, worin das Problem eigentlich besteht. Zur Problemdarstellung nennt die Literatur viele verschiedene Möglichkeiten. Wichtiger als die Wahl der optimalen Methode ist es jedoch, tatsächlich etwas zu tun. Die Analyse wird weiter untergliedert:

2.1.1 Planungsphase

Dies ist die Zieldefinition des Softwareprojektes, an dessen Ende das Softwareprojekt hinsichtlich Terminen, Ressourcen, Kosten und Gewinn als Projekt planbar (siehe Kapitel 2.6) und präsentierbar ist.

1. Produkte und Prozesse.

- Welches Produkt soll entstehen?
- In welchen Handlungsablauf (»Prozess«) wird das fertige Produkt eingebunden sein? Das heißt: Wer wird das Programm zu welchem Zweck benutzen?
- Wie soll das Produkt in diesen Handlungsablauf integriert werden?

2. Personal- und Verantwortlichkeitsplanung.

3. Wirtschaftlichkeitsbetrachtung: lohnt sich das alles überhaupt? Mit welchem Aufwand soll das Ziel erreicht werden? Zur Wirtschaftlichkeitsbetrachtung gehört auch der angestrebte Abgabetermin.

2.1.2 Anforderungsanalyse und Analysemodell

1. Anforderungsanalyse.

Worin besteht genau das Problem und was sind die Anforderungen an die zu findende Lösung?

2. Analysemodell.

Ein Modell ist immer eine Beschreibung eines Produktes, die nicht selbst schon das Produkt ist. In der Architektur sind dies z.B. Papiermodelle und technische Zeichnungen, in der Informatik z.B. Diagramme und Text, aber eben nie ausführbarer Quellcode. Das Analysemodell dient dazu, dem Kunden zu demonstrieren, was durchgeführt werden soll (es hilft zur Kommunikation mit dem Kunden), und als Planungsunterlage für eigene Zwecke, z.B. um die Komplexität des Systems zu erfassen.

Was sind Softwaremodelle und wie werden sie erstellt? Die folgenden Arten von Softwaremodellen basieren auf einfachen logischen Überlegungen. Sie bilden zusammen das Analysemodell, d.i. die Repräsentation der Software; ihre Entwicklung ist interdependent, d.h. sie werden in einem gemeinsamen Prozess entwickelt und verfeinert. Arten von Softwaremodellen:

Aktoren und Anwendungsfälle (»actors and usecases«). Ein sehr früh erstelltes Modell der Software. Weitere Beschreibung siehe Kapitel 4.2.

Modell der Benutzeroberfläche. Ein sehr früh erstelltes Modell der Software. Zusammen mit dem Modell der »Aktoren und Anwendungsfälle« ergibt sich die vollständige Modellierung des Verhaltens des Programms von außen.

Funktionale Datendekomposition. Welche Daten werden von welchen Prozessen bearbeitet und welche Daten werden gespeichert? Notation hierfür ist das Datenflussdiagramm (nicht das Flussdiagramm, denn es geht nicht um Verarbeitungsschritte, sondern um die Daten, die verarbeitet und gespeichert werden). Beispiel siehe Abbildung 16.

Datendiktionär. Die möglichst exakte, aber allgemeinverständliche Definition aller verwendeten Begriffe (Arten von Daten), sozusagen das »Vokabular des Programms«. Dabei soll das Anwendungswissen, das hinter dem Programm steht, mit den verwendeten Daten in Beziehung gebracht werden. Der Detaillierungsgrad kann unterschiedlich sein. Die Erstellung eines Datendiktionärs dient dem Softwareentwickler dazu, sein bisheriges Wissen über das Produkt zusammenzutragen und fehlendes Wissen zu erkennen und dann zu recherchieren (z.B. in Literatur oder beim Kunden).

Klassendiagramm. Bei Klassendiagrammen geht es darum, die inhaltlichen Konzepte festzustellen und ein Konzept der Implementierung zu entwickeln. Klassendiagramme sind ein spätes Modell, das die Software schon im Innern beschreibt. Es ist eine Fortsetzung des Datendiktionärs, das die dort vorgestellten Konzepte verknüpft. Das Klassendiagramm muss nicht die später zu implementierenden Klassen enthalten, sondern untersucht inhaltliche Konzepte in Klassenform. Bei der Entwicklung des Klassendiagramms müssen bereits Entwurfsentscheidungen getroffen werden, z.B. ob eine Variable eindeutig durch einen Namen oder durch mehrere Namen referenziert werden kann. Die Notation erfolgt in UML - zur Beschreibung der UML-Klassendiagramme siehe Kapitel 4.5.

Methode von Abbot: Um die relevanten Konzepte in Klassenform modellieren zu können, muss man sie erst einmal finden. Dazu analysiert man die bisherigen Analysedokumente: jedes Substantiv ist potentiell eine Klasse, jedes zugehörige Adjektiv ist potentiell ein Attribut, jedes Verb ist potentiell eine Methode.

2.2 Entwurf

Die Überlegung, wie etwas zu tun ist: das Design, das Konzept der Implementierung. Man zieht dabei anders als bei der Analyse die Art der Implementation auf einer Rechnerarchitektur in Betracht. Die Entwurfsphase wird weiter gegliedert:

2.2.1 Grobentwurf

Der Grobentwurf ist die Zergliederung des Gesamtsystems: die Aufteilung der Software in Komponenten. Definition: »Eine Komponente ist eine ausführbare und austauschbare Softwareeinheit mit definierten Schnittstellen und eigener Identität.« [13]. Sie ist also kein eigenständiger Prozess des Betriebssystems (kein eigenes Programm!), sondern eine Sammlung von thematisch abgegrenztem Sourcecode. Sie kann z.B. als `namespace` implementiert werden. Die Komponentenaufteilung hat das Ziel, Module für einzelne Teams zu erhalten, die unabhängig voneinander entwickelt werden können. Verfahren der Aufteilung:

1. Architekturmuster.

Da die Komponentenaufteilung recht kompliziert ist, richtet man sich nach bereits früher verwandten, bewährten Architekturmustern. Man nutzt Erfahrungen, wie man die zu erstellende Art von Anwendung eben unterteilt. Das typische Architekturmuster für eine »einfache unverteilter² interaktive Anwendung« zeigt das Komponentendiagramm in Abbildung 1. Dabei bedeuten:

Präsentation Die Benutzeroberfläche, unabhängig von der darunterliegenden Algorithmik. Die Komponenten »Präsentation« und »Geschäftslogik« kommunizieren durch Ein- und Ausgabe, d.h. durch `strings` ohne Rücksicht auf die Bedeutung.

Geschäftslogik Die Komponente Geschäftslogik kann Anweisungen an die Komponente Allgemeine Dienste machen, z.B. »speichere Vektor a unter b «. Sie ist weiter unterteilt:

workflow Die Arbeitsabläufe: die Art, wie was wann sinnvollerweise hintereinander ausgeführt wird. Diese Komponente steuert also die Interaktion mit dem Benutzer. Sie soll verhindern, dass sich die anderen Komponenten gegenseitig direkt aufrufen, indem sie diesen Ablauf global steuert.

Fachkonzepte Weiß etwas über die zugrundeliegenden fachlichen Strukturen (z.B. Vektoren, Geraden) und kann damit umgehen.

²Im Gegensatz zur unverteilter Anwendung sind verteilte Anwendungen solche, die auf einem client-server Modell basieren.

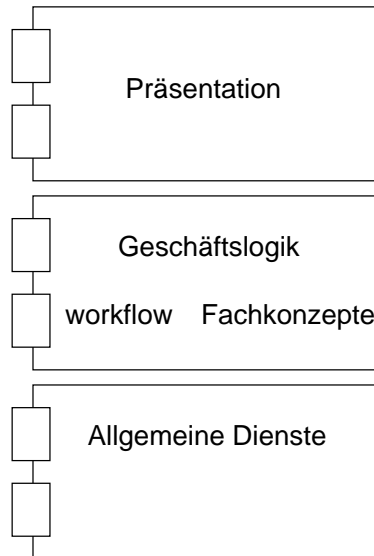


Abbildung 1: Architekturmuster einer einfachen unverteilter interaktiven Anwendung

Allgemeine Dienste Z. B. »Drucken« und »Speichern«.

2. Aktivitätsdiagramm.

Das Architekturmuster ist der erste Ansatz zur Modularisierung der Software; in einem Aktivitätsdiagramm wird nun überprüft, ob dieser Ansatz tatsächlich zur gegebenen Problemstellung passt, und er wird verfeinert. So kann man die inhaltlichen Konzepte in eine passende »Schwimmbahn der Objekte« einordnen und so thematisch kohärente Module bilden, man kann tief untergliederte Aktivitäten als eigenes Kompetenzgebiet auffassen und dafür eine neue Schwimmbahn erstellen.

3. Eigentliche Modularisierung.

Das Architekturmuster und das Aktivitätsdiagramm sind die Quellen, aus denen man die Modularisierung der Software entwickelt. Komponenten sind black boxes, die von den jeweiligen Arbeitsgruppen ausgearbeitet werden sollen. Während der Komponentenaufteilung muss man nicht wissen, wie sie intern funktionieren sollen. Auch wenn man alleine programmiert, ist die Komponentenaufteilung im Sinne der Gliederung in unabhängige Teilaufgaben sinnvoll, denn man muss sich dann nur mit einem Teil gleichzeitig beschäftigen. Bei der Komponentenaufteilung geht man vom gewählten Architekturmuster aus und teilt feiner in Komponenten ein entsprechend folgenden Prinzipien; man überprüft gleichzeitig, ob das gewählte Architekturmuster überhaupt geeignet ist:

Hohe Kohäsion. Die Modularisierung der Software erfolgt nach dem Verantwortungsprinzip (responsibility). Das heißt: In einem Modul werden Konzepte zusammengefasst, die sich mit einem Thema befassen, die also mit einer bestimmten Menge Können und Wissen (zusammen: Verantwortung) bearbeitet werden können. Nun soll sich ein Modul möglichst nur mit einem Thema beschäftigen (man sagt »eine hohe Kohäsion (Zusammenhalt) ist angestrebt«), denn nur dadurch kann erreicht werden, dass in der zugehörigen Entwicklergruppe möglichst gering qualifizierte Mitarbeiter arbeiten können: sie müssen nur auf das Gebiet spezialisiert sein, mit dem sich die Komponente beschäftigt. Und andersherum: jeder Mitarbeiter wird entsprechend seiner Qualifikation einer entsprechenden Entwicklergruppe zugewiesen, so dass er der für diese Komponente übernommenen Verantwortung auch gerecht werden kann.

Beispiel: Kann die Komponente grafische Oberflächen erzeugen, so hat sie die Verantwortung für die Kommunikation mit dem Benutzer.

Geringe Kopplung. Zwischen den Komponenten sollen möglichst wenige geringe Abhängigkeiten bestehen. Dadurch soll erreicht werden, dass erforderliche Änderungen nur lokal in der Komponente bleiben und nicht Änderungen in anderen Komponenten erforderlich machen.

Die Erstellung eines Aktivitätsdiagramms vor der Modularisierung führt u.U. zu Problemen: Aktivitätsdiagramme führen zu einer funktionsorientierten Denkweise, im Fallbeispiel (Kapitel 6) im Aktivitätsdiagramm

gramm (Abbildung 18): zuerst die Zuweisung prüfen und dann die Zuweisung auswerten. Der Versuch, das in Komponenten umzusetzen, scheitert daran, dass keine Kohärenz möglich ist: die Auswertung einer Zuweisung ist nur eine kleine Erweiterung ihrer Prüfung, die Komponente Wert müsste also dasselbe können wie die Komponente Syntax (vgl. Diskussion der Schnittstellen im Fallbeispiel, Kapitel 6).

Die extrem erfolgreiche obektorientierte Denkweise sagt aber: trenne in Kompetenzbereiche, nicht in Aktivitäten, d.i. trenne nach dem Verantwortungsprinzip. Wer top-down vorgeht und so Aktivitäten und Teilkaktivitäten gliedert, kommt nicht zu einer Einteilung in Kompetenzblöcke. Stattdessen gehe man bottom-up vor: man gruppiert alle vorhandenen elementaren Dinge thematisch nach Kompetenzbereichen (man »baut sich Werkzeuge«, das sind Klassen und Komponenten) und löst mit diesen das Problem. Statt dass man erst das Problem löst und dann überlegt, welche Werkzeuge man dazu braucht.

4. Schnittstellen der Komponenten.

Nachdem man nun eine Komponentenaufteilung hat, muss man die Schnittstellen zwischen den Komponenten definieren, d.h. die Struktur der Daten, die zwischen den Komponenten wandern. Die Schnittstelle besteht nicht aus der ganzen Implementierung der Funktionen, sondern nur aus den Deklarationen der Funktionen; diese Schnittstellendefinition durch die Art der Funktionsaufrufe reicht aus, um mit der Schnittstelle zu arbeiten, und ist vor dem Beginn der Entwicklung nötig, damit anschließend die Komponenten unabhängig voneinander bearbeitet und programmiert werden können.

Zur genaueren Diskussion der Art und Implementierung von Schnittstellen und den zugehörigen Darstellungsmöglichkeiten in UML siehe Kapitel 4.4.

2.2.2 Feinentwurf

Während der Grobentwurf das System auf der Detailebene von Komponenten strukturiert, geht der Feinentwurf weiter ins Detail und untersucht Klassen, Funktionen, Methoden usw.

2.3 Implementierung

Die Realisierung der gefundenen Lösung. Die Umsetzung von UML-Konzepten in Sourcecode richtet sich nach den dafür vorgesehenen Möglichkeiten der jeweiligen Programmiersprache. Beispiel zur Implementierung von Komponenten in C++ siehe Kapitel 6.3.1.

2.4 Test und Integration

2.4.1 Verifikation

Die Verifikation soll die Frage beantworten: »Machen wir es richtig?«. Dazu prüft man, ob man mit der Realisierung das Gesetzte Ziel (Spezifikation) erreicht hat. Zur Spezifikation gehören:

- formale Spezifikation (Vor- und Nachbedingungen)
- Pflichtenheft, Lastenheft: erfüllt der Entwurf diese Forderungen?
- Entwurf: erfüllt die Implementierung den Entwurf?

statische Methoden der Verifikation. Bei diesen Methoden wird das erstellte Programm nicht ausgeführt!

Reviews. Diese Methoden haben eine große Bedeutung. Sie sind die Analyse von Dokumenten und heißen »Review« (»Begutachtung«). Es ist das Standardverfahren der Verifikation in der praktischen Softwaretechnik.

Bei Code-Reviews verteilt der Programmierer seinen Code an Teammitglieder und Chef; die anschließende Diskussion soll Schwächen und Fehler im Code aufdecken. Vorher gibt es Reviews von Entwurfsdokumenten. Alle wichtigen Dokumente werden durch Reviews gemeinsam begutachtet.

formale Verifikation. Sicherheitskritische Software sollte möglicherweise formal bewiesen werden.

dynamische Methoden der Verifikation. Das sind Tests. Ein Test dient dazu, Fehler zu finden. Der Fehlerbegriff wird im Englischen genauer unterschieden:

error (auch: Fehler) Etwas Schlechtes, das nicht dem Ideal entspricht. Dieser allgemeine Begriff kann sich auf alle Arten von Fehlern beziehen.

fault (auch: bug, Defekt, Fehler). Das Kaputte in einer Sache, d.h. der semantische Fehler in einem Programm selbst.

failure (auch: Ausfall, Fehler). Das beobachtbare Fehlverhalten eines Programms. Ein Programm mit fault liefert nicht immer einen failure!

Man versucht, mit Tests die Bugs im Programm zu finden. Man hofft dass Defekte (bugs, faults) zu beobachtbarem Fehlverhalten (failure, Ausfall) führen. Bei einem Testfall erwartet man zu einer bestimmten Eingabe eine bestimmte Ausgabe oder Verhalten. Viele Testfälle sind prinzipiell gut, jedoch braucht man auch eine Testfallsystematik: man konstruiert möglichst solche Testfälle, die bei möglichst wenig verschiedenen Tests möglichst viele Bugs aufdecken. Wenn ein Bug durch einen failure aufgedeckt wurde, folgt das Debugging, die Beseitigung des Defekts.

Testverfahren Die Kunst ist, gute Testfälle zu konstruieren. Es gibt verschiedene Verfahren:

black box-Verfahren Die Testfälle werden aus der Spezifikation erzeugt - der Konstrukteur der Testfälle kennt das Programm nicht, sondern nur das, was es können muss, er sieht es als black box. Ein Entwickler kann also nicht gut solche Testfälle für seinen Code entwickeln, weil er ggf. seine eigenen Erweiterungen der Spezifikation berücksichtigt. Die Testfälle sollen alle Klassen an möglichen Eingaben abdecken: man bildet gemäß eigener Intuition Äquivalenzklassen und testet sie. Grundlage sind gemeinsame Eigenschaften der Eingaben.

Beispiele für Äquivalenzklassen in einer Funktion zur Berechnung des Maximums aus 3 Zahlen:

- aufsteigend sortierte Eingabe
- absteigend sortierte Eingabe
- Maximum in der Mitte
- alle Zahlen gleich

white box-Verfahren Die Testfälle werden mit Kenntnis des Programmcodes entwickelt, und zwar so, dass möglichst alle Verzweigungen im Code abgedeckt werden. So soll jede Codesequenz einmal abgearbeitet werden (es soll eine Pfadüberdeckung stattfinden). Bei Schleifen und Rekursionen gibt es nie eine vollständige Pfadüberdeckung.

gray box-Verfahren. Gemischte Verfahren.

Teststrategien

Regressionstests Große Softwareprodukte beginnen mit einem Prototyp, der dann um Funktionalität erweitert wird. Regressionstests bedeuten, dass man alte Testfälle durchläuft: funktionieren die alten Tests auch mit der neuen Funktionalität? Es gibt Tools, die dies automatisch durchführen.

Komponententests Test einer einzelnen Komponente.

Integrationstest Test der Zusammenarbeit mehrerer Komponenten über ihre Schnittstellen.

Systemtest Test des ganzen Systems. Danach hält man das System für auslieferbar.

Akzeptanztest Test mit Kunden: was halten die Kunden von einem Programm, das die Entwickler für fertig halten?

alpha-Test Entwickler und Kunden testen.

beta-Test Nur die Kunden testen.

Testplan³ Der organisatorische Teil des Testens. Ein solcher Plan muss enthalten:

- Wer muss was wann testen?
- Wer entscheidet, was überhaupt ein Fehler ist? Dies soll verhindern, dass Tester den Entwicklern das Leben schwer machen, indem sie kleine Insuffizienzen als Fehler der Hierachiestufe »emergency« einstufen.
- Wie werden Fehler beseitigt? Am besten natürlich mit einer change-request-Datenbank.

2.4.2 Validierung

Hier wird überprüft: Machen wir das Richtige? Haben wir überhaupt ein gültiges Ziel, oder versuchen wir etwas zu erreichen, das der Kunde überhaupt nicht will?

2.4.3 Konfigurationsmanagement

Softwareprodukte (»Systeme«) unterscheiden sich nach Varianten. Aus den verschiedenen Arten von Varianten ergeben sich die Aspekte des Konfigurationsmanagement:

Modifikationskontrolle. Management der verschiedenen Reifegrade der Software, d.h. der zeitlichen Varianten mit Versionsnummern. Dabei werden Änderungswünsche der Kunden in entsprechende Releases und ihre Varianten umgesetzt.

Variantenkontrolle. Management der verschiedenen Ausbaustufen des Systems, d.h. seiner Varianten. Die Varianten unterscheiden sich in ihrem Funktionsumfang für verschiedene Kunden, z.B. »enterprise edition« und »personal edition«.

Systemproduktion. Auch die einzelnen Komponenten eines Systems liegen in Varianten vor. Systemproduktion bedeutet nun die Erstellung eines lauffähigen Systemreleases aus den einzelnen Komponenten, u.a. durch Zusammenbinden: `g++ -o main main.c komp1.o [...]kompn.o`. Aufgrund der sich stets ändernden Compilerversionen kann dies eine anspruchsvolle Aufgabe werden. Es empfiehlt sich daher, für jede Komponente dokumentieren zu lassen, mit welcher Compilerversion und welchen Switches sie produziert werden kann und all diese Werkzeuge stets in lauffähiger Version vorrätig zu halten.

2.4.4 Tools zum Konfigurationsmanagement

make Ein Werkzeug zur Systemproduktion, das in vielen, teilweise sehr mächtigen Varianten existiert.

RCS (»revision control system«) Mit solch einem Sourcecode-Verwaltungssystem kann man die die Änderungen an SourceCode rückverfolgen und rückgängig machen und Varianten und Versionen von Sourcecode verwalten.

Datenbanken Meist verwendet eine Firma hier ihre eigene proprietäre Lösung. Sie enthalten:

- Designdokumente, z.B. Komponentenspezifikationen
- UML-Diagramme
- Änderungsanforderungen (change request database).

CASE-Tools (»computer aided software engineering«) Anwendungsbereiche:

- Zeichnen von UML-Diagrammen
- Code aus (UML-)Diagrammen erzeugen

³Näheres siehe Vorlesungen »Projektmanagement«, »Qualitätssicherung«

2.5 Meilensteine (Zusatz)

Die Projektleiter müssen kontrollieren können, ob ihr Projekt nach Plan läuft oder vielleicht abgebrochen werden muss. Dazu werden im Wasserfallmodell folgende Meilensteile festgesetzt, das sind Zeitpunkte, zu denen ein definierter Projektteil fertiggestellt sein muss. Man kann auf einzelne Meilensteine wie das Lastenheft verzichten oder beliebige weitere festsetzen.

1. Analyse

Meilensteine:

- (a) Lastenheft: nach Planungsphase und Anforderungsanalyse. Dieses erste Dokument definiert (im Unterschied zum Pflichtenheft) das Projektziel. Das Lastenheft ist in einer Sprache formuliert, in der es Entscheidungsträger verstehen können; es wird meist in einer kurzen Präsentation vorgestellt. Anhand des Lastenheftes wird entschieden, ob die genaue Analyse überhaupt sinnvoll ist, an deren Ende das Pflichtenheft stehen würde.
- (b) Pflichtenheft: (auch »Analysedokument«, »Anforderungsspezifikation« o.ä.). Dieses Dokument enthält die komplette Analyse mit allen Analysemodellen, d.h. es definiert genau, was das Produkt können wird. Das Pflichtenheft ist gleichzeitig Grundlage zum Vertrag mit dem Kunden.

Das Lastenheft ist die frühe Niederschrift dessen, was zu tun ist (also mit wenig Wissen und Aufwand), das Pflichtenheft ist die späte Niederschrift dessen, was zu tun ist (also mit mehr Wissen und Aufwand).

2. Entwurf

Meilenstein: Entwurfsdokument

3. Implementierung

Meilenstein: lauffähiger Code

4. Test

2.6 Projekte (Zusatz)

Abbildung 2 zeigt die Organisation von Softwareprojekten in einer real existierenden Firma. Erläuterungen:

Produktmanager Arbeitet im Außendienst, d.h. er besucht potentielle Kunden und liefert dadurch Ideen für Produkte.

SystemTeam Es entscheidet, welche Ideen des Produktmanagers umgesetzt werden und definiert daraus durch grobe Analysedokumente die Projekte.

Projektteam Die hier arbeitenden »einfachen Softwareentwickler« schreiben Codebrocken nach dem Entwurfsdokument, oft ohne zu wissen an welchem Produkt sie überhaupt arbeiten. Entwickler aus dem Anwendungssoftware-Entwicklungsteam sind zu 80% nicht Informatiker, sondern angelernte oder umgeschulte Mitarbeiter.

SoftwareTeam Es verwaltet die erstellten Codebrocken. Hier und in der im SystemSoftware-Entwicklungsteam werden vorwiegend Diplom-Informatiker (FH) eingesetzt.

Bestandteile eines Projektes (am Beispiel von »Programmieren 2, Hausübung 4«, siehe Kapitel 6)

Projektname

Projektteilnehmer

Lastenheft. Es entspricht dem Meilenstein »Lastenheft« und definiert das Projektziel. Im Beispiel ist das Projektziel die pünktliche Abgabe der Hausübung. Man entscheidet in diesem Fall selbst, ob das Projekt durchgeführt werden soll.

Durchführbarkeitsstudie. Man schätzt ab, ob sich die Durchführung des Projektes lohnt, ob sie mit den verfügbaren Ressourcen überhaupt machbar ist.

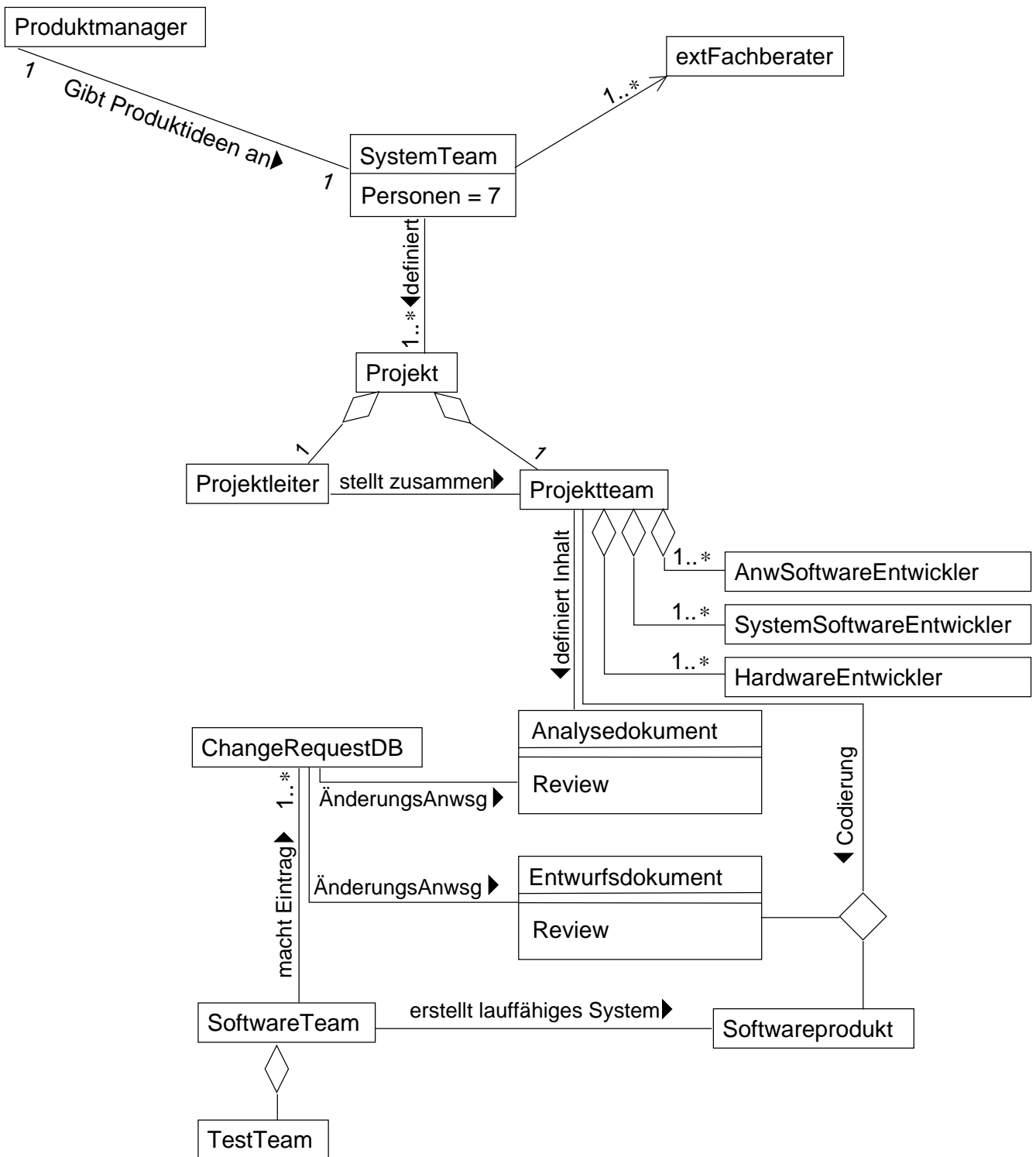


Abbildung 2: Reale Organisation der Softwareproduktion

Kosten / Nutzen. Im Beispiel sind die Kosten hauptsächlich Freizeit. Geschätzter Zeiteinsatz: 72h. Nutzen: Bonuspunkte zur Klausur. Der Aufwand wird in Personentagen (*PT*) zu je 8 Arbeitsstunden angegeben.

Ressourcen

Personal

Art der Kooperation

Meilensteine. Im Beispiel gibt es mehrere Alternativen, Meilensteine zu setzen und so die verfügbare Zeit zu planen:

- das klassische Wasserfallmodell mit sehr spät lauffähigem Code, dafür guter Analyse, aber dem Risiko, dass die Implementierung fehlschlägt.
- die Entwicklung selbständiger Module als eigene Teilprojekte, nämlich entsprechend den aufeinander aufbauenden Aufgabenstellungen der Hausübungen 1-4. Dadurch entsteht früh lauffähiger Code und das durch den ungewohnten Umgang mit komplexen Programmieretechniken bestehende Risiko wird minimiert. Die einzelnen Teilprojekte würden jeweils wieder mit dem Wasserfallmodell bearbeitet.

Das Risiko dieser Methode ist eine möglicherweise unzureichende Analyse, so dass das Falsche implementiert wird oder die Einzelteile nicht zueinander passen.

Risiko. Das Beispielprojekt ist mit hohem Risiko behaftet bzgl. dem Einsatz der komplexen Technologie, die für diese einfache Aufgabenstellung verwendet wird.

3 Strukturierte Programmierung

Auch »systematisches Programmieren« genannt. Begonnen hat dieses Thema mit dem Artikel des Holländers Dijkstra »GOTO considered harmful« (1968). Danach disqualifiziere die Verwendung von GOTO einen Programmierer. Stattdessen forderte er als Antwort auf die Softwarekrise eine wissenschaftliche Art der Programmierung.

3.1 Was ist strukturierte Programmierung?

Diese von Dijkstra geforderte und begründete »wissenschaftliche Softwareproduktion« besteht aus zwei Elementen:

Erstellen einer Spezifikation. Wann ist eine Software korrekt? Dijkstra sagte: Testen kann nur die Anwesenheit von Fehlern zeigen, aber nicht deren Abwesenheit. Stattdessen definierte er die Korrektheit wissenschaftlich: Die Software ist korrekt, wenn sie ihrer Spezifikation genügt.

Entwickeln nach Spezifikation. Dijkstra forderte weiter: Software sollte systematisch entsprechend ihrer Spezifikation entwickelt werden.

3.2 Spezifikation sequentieller Programme

»Ein sequentielles Programm ist etwas, das die Belegung von Variablen verändert«. Es muss theoretisch geklärt werden:

- Was ist eine Spezifikation?
- Wann erfüllt ein Programm seine Spezifikation?

Der Begriff »Spezifikation« war vor der Einführung objektorientierter Programmierung, d.h. für sequentielle Programme, recht einfach zu definieren.

- Sei die Vorbedingung Q . Sie grenzt die Pflicht des Programmes ein.
- Die Sequenz S (das Programm) verändert die Variablenbelegung, die der Vorbedingung Q entspricht.
- Die Nachbedingung R ist das Ergebnis dieser Veränderung.

Dann ist die Spezifikation der Sequenz S : Wenn die Vorbedingung erfüllt ist, so muss sie die Variablenbedingung so ändern, dass sie der Nachbedingung genügt. Wenn die Vorbedingung nicht erfüllt ist, so muss sie keiner Anforderung genügen.

Zusicherungen sind Aussagen über Variablenbelegungen, d.h. Vor- und Nachbedingung sind Zusicherungen.

3.3 Das Hoare-Kalkül

Nach der theoretischen Definition von »korrekter Software« und ihrer »Spezifikation« wurde natürlich versucht, ein Beweisverfahren zu finden, mit dem man beweisen kann, dass eine Sequenz ihrer Spezifikation entspricht, also korrekt ist. Im allgemeinen haben alle Anweisungen ja eine bestimmte, rechnerisch nachvollziehbare Wirkung. Der Ire Hoare entwickelte, unter Mitarbeit des Pascal-Erfinders Wirth, mit dem Hoare-Kalkül ein entsprechendes Beweissystem.

Dieser Beweis erfordert bei Schleifen das Finden der Schleifeninvariante, d.h. hier benötigt er Kreativität. Die Schleifeninvariante ist eine wesentliche Beziehung zwischen in der Schleife veränderten Variablen, die sich während der Schleifendurchgänge nicht ändert. Der Beweis von Schleifen entspricht dem Beweis über vollständige Induktion! Eine Schleife ist korrekt, wenn:

1. Die Schleifeninvariante bei Eintritt in die Schleife gilt (d.h. vor dem ersten Durchlauf). Dies entspricht der Induktionsverankerung.
2. Die Schleife die Schleifeninvariante erhält. Dies kann aus den Werten des vorigen Schleifendurchgangs berechnet werden (vgl. Beispiel). Dies entspricht dem Induktionsschritt. Um zu beweisen, dass die Schleife ihre Invariante nicht erhält, genügt ein Gegenbeispiel; das bedeutet dann im Allgemeinen, dass die Sequenz falsch ist, es kann aber auch bedeuten, dass man die falsche Schleifeninvariante gewählt hat!
3. Aus der Invariante und der Verneinung der Schleifenbedingung die Nachbedingung gefolgert werden kann.

Beispiel: Fakultätsberechnung

- Vorbedingung Q : $n \geq 0$.
- Sequenz S .

```
//n>=0
int f=1;
int i=0;
while (i<n) { //INV: f=i!
    ++i;
    f=f*i;
} // f=i! && !(i<n)
// n>=0, f=n!
```

- Nachbedingung R : $f = n!$.

Entspricht S der in Vorbedingung Q und Nachbedingung R festgelegten Spezifikation? Verwendung des Hoare-Kalküls zum Beweis der Richtigkeit der Schleife:

1. Gilt die Schleifeninvariante bei Eintritt in die Schleife? Ja, denn $1 = 0!$.
2. Erhält die Schleife die Schleifeninvariante?
 - Zu Beginn eines beliebigen Schleifendurchlaufs gelte die Schleifeninvariante, d.h. $i = i_0$, $f = f_0 = i_0!$.
 - Es werden die Anweisungen `++i; f=f*i;` ausgeführt.
 - Danach gilt mathematisch:

$$\begin{aligned} i &= i_0 + 1 \\ f &= f_0 \cdot i \\ &= i_0! \cdot (i_0 + 1) \\ &= (i_0 + 1)! \\ &= i! \end{aligned}$$

Durch Einsetzen der alten Werte in die Anweisungen und Zusammenfassung zu Ausdrücken mit neuen Werten konnte mathematisch gezeigt werden, dass die Schleife ihre Invariante erhält.

3. Kann aus der Verneinung der Schleifenbedingung und aus der Schleifeninvariante die Nachbedingung gefolgert werden? Ja:

$$\begin{aligned} \overline{(i < n)} \quad \wedge \quad f = i! \\ \Rightarrow i = n \quad \wedge \quad f = i! \\ \Rightarrow f = n! \end{aligned}$$

Dabei wird der logische Schluss $\overline{(i < n)} \Rightarrow i = n$ verwendet; er ist unter der gegebenen Voraussetzung zulässig, dass i stets um 1 wächst und so irgendwann auf den ganzzahligen Wert n trifft.

3.4 Strukturierte Programmierung in der Praxis

Auswirkungen der Strukturierten Programmierung auf die Praxis:

- **assert** -Anweisung in Programmiersprachen, um prüfen zu können, ob Vor- und Nachbedingungen gelten.
- Konventionen für Kommentare: man gibt bei einer Funktion (jedoch nicht bei einer einfachen Sequenz S) Vor- und Nachbedingung an: unter welchen Bedingungen sie korrekt arbeitet und welches Ergebnis sie dann liefert.
- Schrittweise Verfeinerung als Programmentwicklungsmethode (sog. »top-down«-Entwicklung, auch selbst als »Strukturierte Programmierung« bezeichnet). Das heißt: rekursive Problemzerlegung eines schwierigen Problems in einfachere Probleme, bis das Problem trivial ist. In der Praxis kann man so komplizierte Schleifen entwickeln und weiß, dass sie aufgrund des verwandten Hoare-Kalküls korrekt sind, ohne zu testen. Am Beispiel der Fakultätsberechnung:
 - Großes Problem: die Programmspezifikation aus Vorbedingung ($n \geq 0$) und Nachbedingung ($f = n!$, n unverändert).
 - Erste Zerlegung in Teilprobleme:
 - * Berechnung von $f = i!$ aus $f = (i - 1)!$ in einer Schleife, unter Erhalt der Schleifeninvariante
 - * Die Initialwerte der Schleife müssen die Schleifeninvariante erfüllen.
 - * Aus der Abbruchbedingung der Schleife und der Verneinung der Schleifeninvariante soll die Nachbedingung folgen.
 - Zweite Zerlegung in Teilprobleme: Das vollständige Programm.

3.5 Spezifikation objektorientierter Programme

Kann man diese Überlegungen zur Korrektheit von Software auf die moderne, objektorientierte Softwareentwicklung übertragen? Dazu prägte der Franzose Meyer⁴ den Begriff »Design by Contract« (Entwurf durch Vertrag): es gebe zwei Objekte, den Kunden (Benutzer, benutzendes Objekt) und den Dienstleister (benutztes Objekt). Der Vertrag ist nun: Unter bestimmten Bedingungen (der Vorbedingung P), die der Kunde erfüllen muss, wird der Dienstleister andere Bedingungen (die Nachbedingung Q) garantieren. Die Vor- und Nachbedingungen müssen für die einzelnen Methoden aufgliedert werden.

Es macht also keinen Sinn, ein Programmstück (eine Klasse) nur dann als korrekt anzusehen, wenn es (sie) unter allen Bedingungen fehlerlos arbeitet⁵. Würden nämlich alle möglichen Fehlbedingungen abgefangen, ergeben sich vor und nach dem Aufrufe eines Programmstücks redundante Überprüfungen. Stattdessen sind Fehlbedingungen dann möglich, wenn man sich nicht an die Vorbedingung hält.

Beispiel: Spezifikation einer Klasse `stack`

- Vorbedingungen
 - `top` nur bei nicht leerem Stack
 - `pop` nur bei nicht leerem Stack

⁴u.a. Erfinder der Programmiersprache Eiffel

⁵Für ein Gesamtprogramm sollte dies jedoch gelten: Freiheit von Abstürzen.

- push nur bei nicht vollem Stack

- Klassendefinition

```
// A: a[0..top] ist der dargestellte Stack, a[0] ist unten
// INV: -1<=top<size
template <class T, unsigned int size>
class stack {
public:
    stack() : top(-1) {}
    int top() {};
    void push(T x){
        //PRE: Stack nicht voll, d.h. top<size-1
        ++top;
        a[top]=x;
    } //POST: Stack hat x als oberstes Element, d.h. a[top]=x
    void pop() {};
private:
    int top;
    T a[size];
};
```

- Nachbedingungen

- top: liefert a[top], top wird top-1
- push(x): Der Stack hat x als oberstes Element
- pop: top ist um 1 kleiner

Abstraktionsfunktion und Klasseninvariante Programmieren ist Modellbildung: es gibt den Raum der abstrakten Objekte (echte, real existierende, mit dem Programmcode gemeinte Objekte wie z.B. echte Stacks) und den Raum der konkreten Objekte (die aus Feldern und Integern und Werten bestehenden implementierten Stacks).

Beim Entwurf eines Programms muss man nun entscheiden, wie ein abstraktes Objekt modelliert (»abgebildet«) werden soll. Zum Beispiel kann man festlegen, dass die Belegung der Attribute eines Stack `top`, `a[size]` meint, dass `top` auf den ersten freien Platz des Stack zeigt ($0 \leq top \leq size$) oder auf den letzten belegten Platz ($-1 \leq top \leq size - 1$). Ebenso muss man entscheiden, welches das unterste Element im Stack sein soll und auf welcher Seite des Feldes (bezogen auf `top`) der Stack liegt. Zusammen ist das die Abstraktionsfunktion `A`, die vom Konkreten zum Abstrakten vermittelt dadurch, dass sie jedem konkreten Stack einen abstrakten Stack zuordnet. Weil die Abstraktionsfunktion also die Essenz des Entwurfs ist, dokumentiert man sie durch einen Kommentar zur Klasse (siehe Beispiel `class stack`).

Aus der Abstraktionsfunktion wird dann die Klasseninvariante abgeleitet, das ist die Bedingung, die alle nach der Abstraktionsfunktion möglichen Belegungen der Attribute gemeinsam erfüllen (siehe Beispiel `class stack`). Sie muss vor und nach der Ausführung jeder Methode gelten und ist damit die Bedingung für das Funktionieren des Objektes. Hierüber ist analog zur Schleifeninvariante der Beweis der Korrektheit einer Klasse möglich:

1. Die Klasseninvariante muss bei Eintritt in die Methode gelten. Die Methode setzt dies ungeprüft voraus.
2. Die Methode muss die Klasseninvariante erhalten.

Bezogen auf abstrakte und konkrete Zustände heißt das (vgl. Abbildung 3): Die konkrete Funktion `pushK` ist eine korrekte Implementierung der abstrakten Funktion `pushA` genau dann, wenn zum gleichen Ziel führen:

- Der Weg vom konkreten Zustand zu seinem abstrakten Zustand und dann über die abstrakte Funktion `pushA`.
- Der Weg über die konkrete Funktion `pushK` zum konkreten und dann zu dessen abstrakten Zustand.

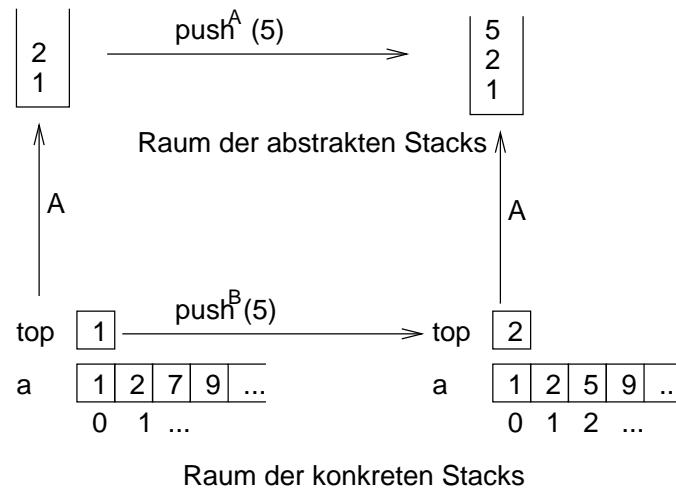


Abbildung 3: Skizze der Beziehung zwischen konkretem und abstraktem push

4 UML-Diagramme

Dieses Kapitel ist keine vollständige Referenz zu der in der Vorlesung behandelten Notation! Als solche verwende man das aktuelle und frei kopierbare Dokument [13]. Hier werden nur einige Anmerkungen gemacht.

4.1 Einsatz von UML

- Illustration und Kommentierung von vorhandenem SourceCode. Vereinfacht spätere Änderungen.
- »Höheres Programmieren«, d.i. übergeordnete Softwareentwicklung auf abstraktem Niveau.

4.2 Anwendungsfalldiagramme

Das Modell der »Aktoren und Anwendungsfälle« (siehe Kapitel 2.1.2) ist die ganz praktische Untersuchung, wer das zu ertellende System wie benutzen wird. Ein Aktor ist eine Person oder ein Ding, die / das das System in einer Rolle benutzt. Ein Anwendungsfall ist ein Benutzungsszenario. Modelldarstellung:

- Konzept der Programmoberfläche und Beschreibung ihres Verhaltens. Für einfache Fälle geeignet. Bei komplizierten Fällen ist die Gliederung in »business usecases«, »technical usecases« und dann das Oberflächendesign sinnvoll (vgl. die Vorlesung Systemanalyse).
- Aufzählender, allgemeinverständlicher Text. Dabei können die Anwendungsfälle hierarchisch gegliedert werden. Für einfache Fälle geeignet.
- UML-Anwendungsfalldiagramm. Auch für komplizierte Fälle geeignet. Aktoren werden durch einen stilisierten Menschen dargestellt, Anwendungsfälle durch ein Oval mit Text. Solche graphische Darstellung ist aber nicht unbedingt sinnvoll, weil die Darstellung weniger wichtig ist als die Überlegungen.

Beispiel: Für ein System zur Pfandrückgabe für Getränkeverpackungen gibt es vier Aktoren mit passenden Anwendungsfällen:

- Benutzer, der Pfand bekommt
- Benutzer, der die Pfandwerte definiert
- Eigentümer, der das Geld bekommt
- Benutzer, der die leeren Verpackungen bekommt

4.3 Aktivitätsdiagramme

Die Schwimmbahnen der Aktivitäten (wie in Abbildung 18) sind eine frühe Phase der Komponentendefinition: jede Schwimmbahn enthält ein Kompetenzfeld. Die Ausarbeitung erfolgt dann in einem Komponentendiagramm. Aktivitätsdiagramme werden in den frühen Phasen der Softwareentwicklung benutzt. Beispiel siehe Abbildung 18, Notation siehe Tabelle 1.

Mit Aktivitätsdiagrammen wird der Informationsfluss zwischen den Komponenten (das sind die Aktivitäten in einer Schwimmbahn) im Verlauf der Zeit betrachtet.⁶ Der Informationsfluss zwischen Aktivitäten wird durch gestrichelte Linien dargestellt. Die Art der übergebenen Information wird durch Stereotype dargestellt, z.B. meint die Stereotype entity (»Einheit«), dass das übergebene Objekt im wesentlichen Informationen trägt, also nur Wissen und kein Können. Durch Pfeile wird dargestellt, dass eine Aktivität aufgehoben wird und dabei in eine oder mehrere durch sie angestoßene Aktivitäten führt.

4.4 Komponentendiagramme

Ein Komponentendiagramm stellt die Organisation und die Abhängigkeiten der Komponenten dar. Es wird u.a. im Grobentwurf (Komponentenaufteilung der Software) benötigt.

4.4.1 Komponentenbegriff der UML

Eine UML-Komponente repräsentiert ein Softwarestück mit dem Namen der Komponente, das einen internen Zustand hat und über eine Serie von freien Funktionen (Interface-Operationen) benutzt wird⁷. Diese Funktionen, die die Komponente anderen Softwareteilen zur Verfügung stellt (sog. »Schnittstelle«) definieren die Funktionalität der Komponente.

Eine Komponente im Sinne von UML ist also ein abstraktes Objekt, d.h. UML hat einen sehr eingeschränkten Komponentenbegriff.

Zu jedem Objekt eines UML-Objektdiagramms gibt es eine eindeutig bestimmte Klasse in einem UML-Klassendiagramm. Auch zwischen Komponentendiagrammen und Klassendiagrammen gibt es einen Zusammenhang: Eine Komponente können wir ansehen als »(abstraktes) Objekt, das die einzige Instanz seiner Klasse ist«, so dass also immer die Darstellung der Klassen impliziert wird.

4.4.2 Komponente i.S.v. UML versus Komponente i.S.v. OOP (Beispiel)

Komponente im Sinne von UML Das Interface wird in einer `.h`-Datei definiert:

```
//stack.h
#ifndef STACK_I_H
#define STACK_I_H
void push(int);
void pop();
int top();
#endif
```

Die zugehörige `.c`-Datei definiert die in `stack.h` deklarierten Interface-Funktionen:

```
//stack.c
#include<stack.h>
int i=1; // 1 ist Top von Stack
int a[1000]; // die Elemente
int top() {return a[i];}
//...
```

Hier zeigt sich, dass UML-Komponenten abstrakte Objekte sind: `stack.c` implementiert keine Klasse (also keinen Typ von Dingen), sondern ein einziges Ding. In einem Programm, das den Stack benutzt, kann es daher nur diesen einen Stack geben, man kann nicht mehrere Instanzen erzeugen wie es bei Klassen möglich ist. Wollte man mehrere Stacks in einem Programm verwenden, müsste man eine weitere `.h`-Datei anlegen, in der die zur Verfügung gestellten Funktionen anders heißen. »Stack« ist hier also nur im logischen Sinne

⁶In Aktivitätsdiagrammen können die Schwimmbahn der Aktivitäten und die vertikale Zeitachse auch fehlen; dies ist sogar der Normalfall.

⁷Ursprünglich dienten die beiden Rechtecke in einem Komponentendiagramm dazu, die Interface-Funktionen darzustellen.

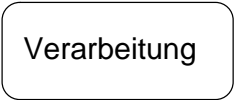
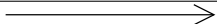


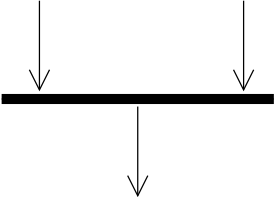
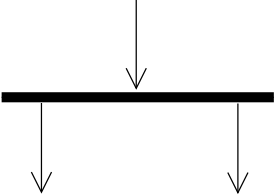
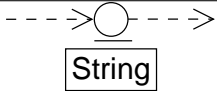
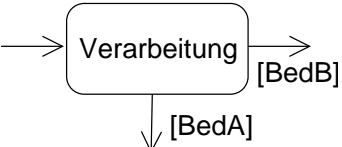
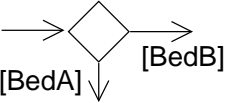
	eine Aktivität
	anstoßen einer anderen Aktivität
	Programmbeginn
	Programmende
	Splitting: Parallelverzweigung von Aktivitäten; tue beides, ohne festgelegte Reihenfolge.
	Synchronisation: Zusammenführung nach vorherigem Splitting
	Übergabe eines Strings zwischen Aktivitäten. Die Art des übergebenen Objektes wird durch Stereotypen gekennzeichnet, hier durch die Stereotype Entity (Einheit).
	Entscheidung, erste Möglichkeit. Dient der Darstellung einer Verzweigung im Kontrollfluss.
	Entscheidung, zweite Möglichkeit.

Tabelle 1: Notation von UML-Aktivitätsdiagrammen

ein Objekt (ein »abstraktes Objekt«, d.h. ein simuliertes Objekt), kein echtes Objekt im Sinne von OOP. UML-Komponentendiagramme meinen immer (!) ein Stück fertige Software mit einem abstrakten Objekt als Interface. In UML kann nicht ausgedrückt werden, dass ein Objekt eine Klasse als Interface zur Verfügung stellt⁸.

So wurden Objekte in den 50 Jahren vor Entwicklung der OOP-Sprachen programmiert. Warum benutzt UML diese überholten »abstrakten« Komponenten für seine Komponentendiagramme? Hauptsächlich deshalb, weil die Industriestandards (COM⁹, CORBA¹⁰) auch noch diesem »naiven« Komponentenbegriff folgen. Diese Standards basieren darauf, die binären Aufrufkonventionen von Funktionen festzuschreiben und so die Software in abstrakten Komponenten austauschbar zu machen. Ein weiterer, nicht derart trivialer Standard für Komponenten sind »JavaBeans«.

Komponente im Sinne der OOP Heute würde man unter Verwendung von OOP eine Komponente formulieren, die den Typ »Stack« exportiert, nicht nur eine Instanz. Vom Typ Stack kann man nun beliebig viele Instanzen erzeugen.

```
//stack.h
class Stack {
    public:
        void push(int);
        void pop();
        int top();
    private:
        //...
};
// stack.c (Methodendefinitionen)
stack::push(int x) {
    //...
}
```

Diese Art Komponente, die einen Typ »Stack« exportiert, kann in UML nicht als Komponente in einem Komponentendiagramm dargestellt werden, sondern nur in einem in einem Klassendiagramm. Ein möglicher, aber nicht wirklich objektorientierter Kompromiss ist, die in der `stack.h`-Datei einer Stack-Komponente im Sinne der UML exportierten freien Funktionen in der `stack.c`-Datei durch den Aufruf der entsprechenden Funktionen eines Objekts zu definieren. Das Objekt muss die einzige Instanz einer in dieser `stack.c`-Datei definierten Klasse `stack` sein.

4.4.3 Schnittstellen in UML

Wie oben dargestellt, sind die Schnittstellen von UML-Komponenten freie Funktionen. Auch in der Programmierung verwendet man ja oft freie Funktionen zur Realisierung von Schnittstellen. Nun können freie Funktionen aber nicht in UML-Klassendiagrammen dargestellt werden, trotz dass Komponentendiagramme genau dies meinen¹¹. Stattdessen müssen die Funktionen in das »objektorientierte Paradigma¹²« eingepasst werden: sie werden UML zuliebe in neu zu definierende Klassen gefasst.

Diese Interface-Klassen sind in UML Klassen vom Stereotyp <<interface>> und können kurz durch den Stereotyp für »Klasse« (einen Kreis) dargestellt werden (siehe Abbildung 5). Stereotype geben spezielle Arten von Klassen an. Interface-Klassen unterscheiden sich dadurch von normalen Klassen, dass sie nur deklariert, aber nicht implementiert werden. Warum das?

Die Komponente, die ein Interface exportiert, wird in UML als Implementierung (Spezialfall) des Interface angesehen. Der entsprechende Pfeil in UML (»implementiert«-Pfeil, auch »realizes«-Pfeil) ist daher dem Vererbungspfeil nachempfunden: Er weist stets von der Komponente zur Schnittstellenklasse, vom Besonderen zum Allgemeinen, vom Konkreten zum Abstrakten, und deutet damit an, dass die Komponente (die Implementierung) ein Sonderfall der Schnittstelle ist. So sind z.B. Pkw und Lkw Spezialfälle des aus Lenkrad und Pedalen bestehenden Interfaces. Wenn wie hier zwei Klassen dieselbe Schnittstelle implementieren, bringt dies

⁸Deshalb muss UML im professionellen Einsatz erweitert werden ...

⁹Common Object Model

¹⁰Common Object Request Broker Architecture

¹¹»UML treibt OOP bis zum Exzess.«

¹²meint »beispielhaftes Denkmuster«

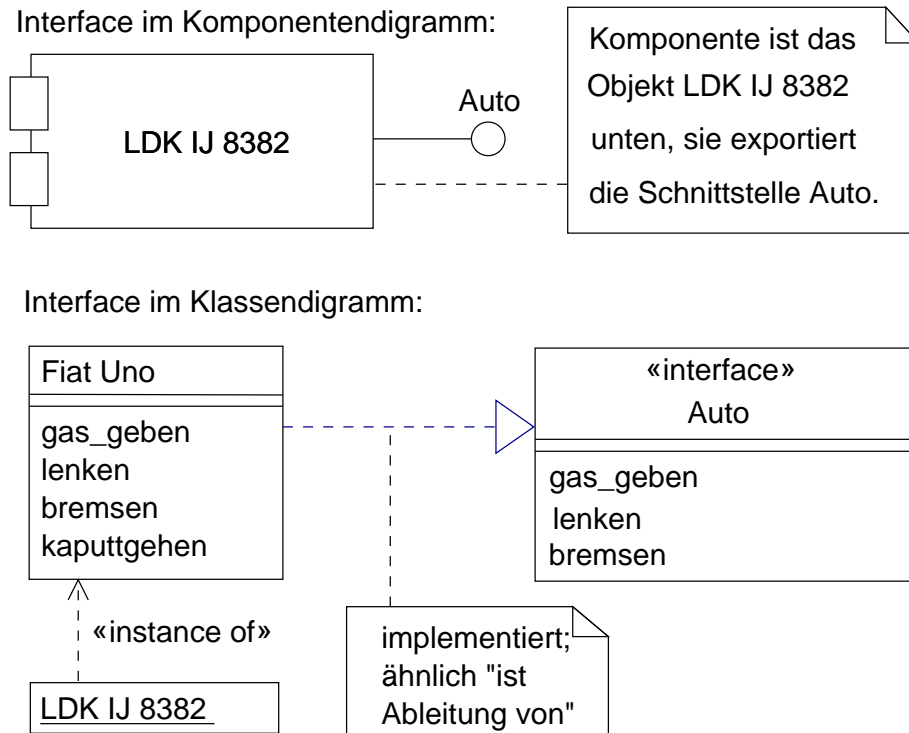


Abbildung 4: Die Schnittstelle »Auto«

zum Ausdruck, dass man mit beiden Klassen dasselbe tun kann, auch wenn die Klassen intern unterschiedlich funktionieren.

An Abbildung 4 wird deutlich: das »Interface«-Zeichen im UML-Komponentendiagramm ist (nach Einpassung ins objektorientierte Paradigma ...) eine kurze Notation für eine `<<interface>>`-Klasse, die Methoden exportiert, aber nicht definiert. Diese Oberklasse stellt also nur das Konzept zur Verfügung, sie hat auch keinen `private`-Teil. Von ihr können aufgrund der fehlenden Definitionen keine Instanzen gebildet werden, sondern nur Ableitungen. Diese abgeleiteten Klassen implementieren das Interface, d.h. sie müssen die in der Interface-Klasse deklarierten Methoden anbieten (die Oberklasse ist ihre »Schnittmenge« und ihre Instanz ist eine Komponente i.S.v. UML).

Solche Interface-Klassen, die keine Instanzen haben können, werden in C++ wie folgt geschrieben:

```
class Auto{
public:
    virtual void gasgeben()=0;
    virtual void bremsen()=0;
    virtual void lenken()=0;
};
```

Durch =0 wird angegeben, dass die Methoden nicht definiert werden. Die Klasse Auto ist eine sog. »abstrakte Basisklasse« oder »Schnittstelle«, in Java »Interface«. Von ihr gibt es keine Instanzen: es gibt nichts, was nur Auto ist, ohne zusätzlich ein bestimmtes Modell zu sein.

4.5 Klassendiagramme und Objektdiagramme

Das Klassendiagramm stellt inhaltliche Konzepte in Klassenform oder eine zu implementierende Klassenstruktur dar und untersucht die Beziehungen der Klassen / Konzepte untereinander. Es geht um die Fragen: Welche Konzepte der Anwendungswelt sind für die Problemstellung relevant, welche Beziehungen gelten zwischen ihnen, welche Attribute haben sie? Die UML-Diagrammart »Klassendiagramm« und »Objektdiagramm« können gemischt werden. Problematisch ist dabei, dass sowohl eine Klasse als auch ein Objekt durch ein Viereck dargestellt werden. Die Titel von Objekten werden zur Unterscheidung daher unterstrichen dargestellt. Auch wenn dies nicht gemacht wird, kann man Klassen (das sind Typen) und ihre Instanzen (die Objekte)

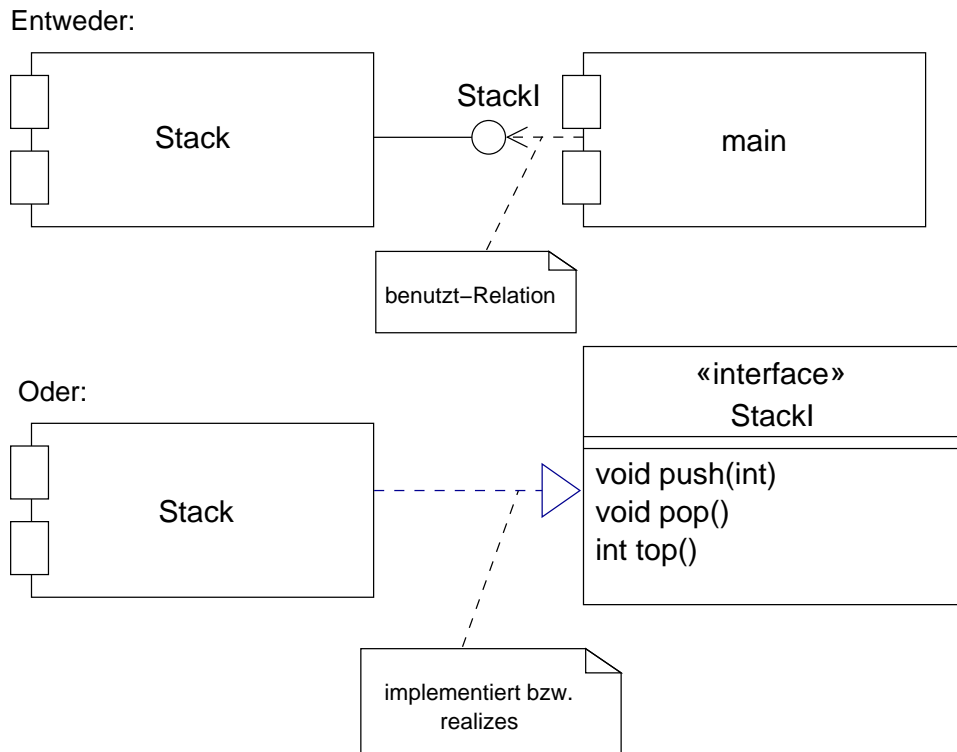


Abbildung 5: Synonyme Möglichkeiten zur Darstellung eines Interface in UML

unterscheiden: In UML gibt es keine anonymen Typen, d.h. jede Klasse hat einen Namen. Und bei jedem Objekt wird die Klasse durch »:<klassenname>« angegeben. Typen und Templates wiederum unterscheiden sich in der Symbolik durch den Kasten mit der Template-Parameterklasse.

Klassendefinitionen in der Softwaretechnik sind immer implizit unvollständig, angelegt auf spätere Erweiterung. Das Klassendiagramm ist der erste formale Beginn in der Notation des Programms; hier werden Beziehungen zwischen Klassen ausgedrückt, die aber im Verlauf von Entwurf und Codierung geändert und genauer spezifiziert werden (z.B. »benutzt als Komponente« statt »benutzt«), so dass das Klassendiagramm nie ganz fertig wird.

4.5.1 Klasse

In ein Symbol für Klasse in UML werden nur die momentan relevanten Informationen aufgenommen (vergleiche Abbildung 6):

- Klassenname (erstes Feld). Die in einem Klassendiagramm dargestellten Konzepte müssen benannt werden!
- ggf. Attribute (zweites Feld). Attribute sind die Datenkomponenten (»Eigenschaften«) der Klasse. Ob man etwas als Attribut oder abhängige Klasse (Komposition / Aggregation) realisiert, ist eine eigentlich freie Entscheidung; man trifft sie danach, ob die darzustellenden Eigenschaften ein wichtiges (anderswo referenziertes) Konzept im Klassendiagramm sind oder eine Eigenschaft, die nur zu dieser Klasse gehört.
- ggf. Methoden (drittes Feld). Methoden sind die Operationen (services), die man auf Objekte dieser Klasse anwenden kann.
 - + vor Methoden heißt: öffentliche Methoden. Diese können auch von Objekten anderer Klassen aktiviert werden.
 - – vor Methoden heißt: private Methoden. Diese können nur von Objekten derselben Klasse aktiviert werden.

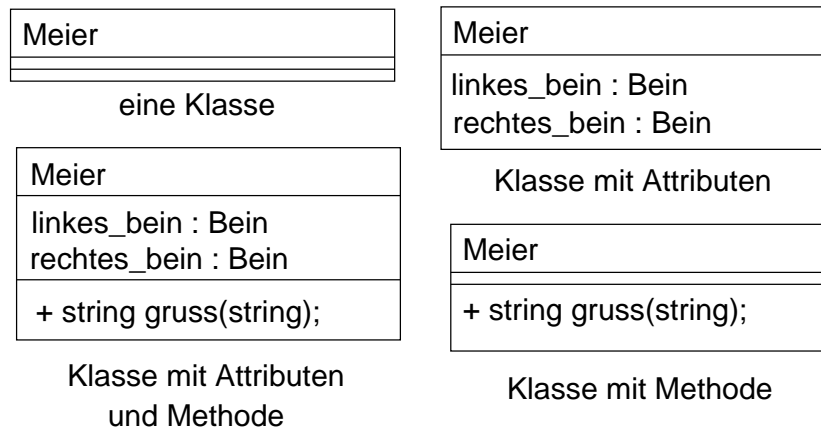


Abbildung 6: Notation von Klassen in UML

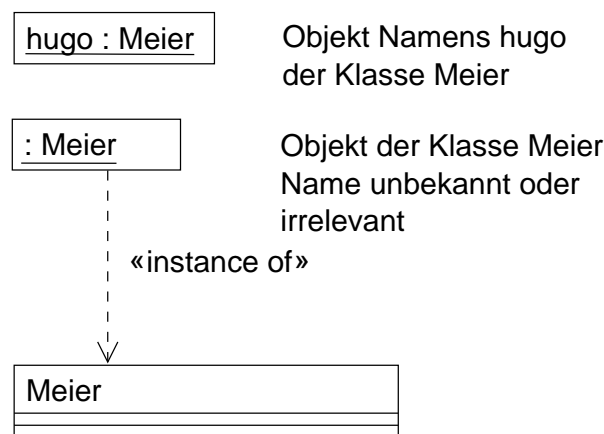


Abbildung 7: Notation von Objekten in UML

- ggf. Verantwortlichkeiten (viertes Feld). Diese natürlichsprachlichen Angaben sind zwar unpräzise, repräsentieren aber das in der Objektmodellierung sehr wichtige »Verantwortungsprinzip«. Auch in Komponentendiagrammen sind diese Angaben möglich.

4.5.2 Objekt

Klassen sind nur Typen von Daten. Objekte sind ihre Instanzen. Sie werden in UML dargestellt wie in Abbildung 7. Auch Objekte können implizit unvollständig sein, z.B. kann ihr Name fehlen. Durch einen gestrichelten Pfeil mit der Beschriftung `<<instance of>>` kann dargestellt werden, von welcher Klasse ein Objekt eine Instanz. Weil der Titel eines Objektes aber immer mit der Angabe `:<klasse>` endet, ist dieser Pfeil eigentlich unnötig.

4.5.3 Assoziation

Notation siehe [13, S. 2]. Eine Assoziation ist die allgemeinste Form einer Beziehung zwischen Objekten, dargestellt als Beziehung zwischen den Klassen, zu denen sie gehören. Zwei Klassen, die in einer Beziehung zueinander stehen, sind assoziiert. Beispiel: Ein Kunde mietet ein Auto. »mieten« ist der Beziehungsname der Assoziation.

4.5.4 Abhängigkeit

In der Vorlesung als »Benutzt-Relation« bezeichnet. Zur Notation siehe [13, S.2] unter der Bezeichnung »Abhängigkeit«. Die Relation »A benutzt B« heißt nur, dass in der Definition der Klasse A die Definition der Klasse

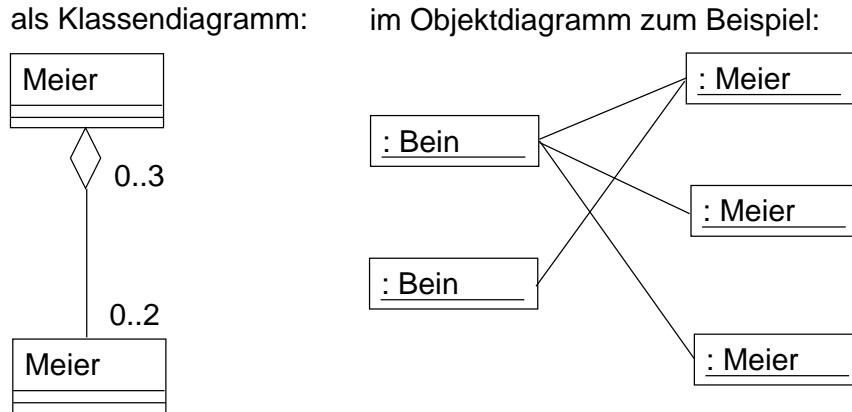


Abbildung 8: »Aggregation zwischen Meier und Bein«

B benutzt irgendwie benutzt wird, z.B. als Typ eines Parameters, dass also Klasse A abhängig von Klasse B ist. Auswirkung: Eine Änderung in der Definition der Klasse B kann eine Änderung in der Definition von Klasse A bedeuten.

Wenn eine Klasse eine andere tatsächlich als Komponente benutzen soll, verwendet man eine genauere Notation, nämlich die Aggregation oder sogar Komposition. Dies spezialisiert die bestehende Abhängigkeit.

4.5.5 Aggregation

Notation siehe [13, S.2] unter »Aggregation«. Dies ist eine besondere »benutzt«-Relation, nämlich »benutzt als Komponente« bzw. »besteht aus Komponente«, wobei die Existenz der benutzen Klasse (der Komponente) auch allein sinnvoll möglich ist. Eine Aggregation kann wie die Assoziation mit einem Namen versehen werden!

Bei der Aggregation können beliebige Kardinalitäten gewählt werden. Die Aggregation wird mit Hilfe eines Bezuges (Zeiger *, Indexnummer, andere Art Referenz) implementiert. Dadurch wird ausgedrückt, dass die Aggregation ein Bezug zu unabhängigen, selbst existierenden (»entkoppelten«) Objekten ist. Ein etwas seltsames Beispiel: Ein Objekt der Klasse Meier benutzt zwei Objekte der Klasse Bein; aufgrund der Aggregation können auch zwei Meiers dasselbe Bein (vielleicht ein Holzbein) benutzen, d.h. die Beine sind austauschbar und können auch für sich, ohne einen Meier, existieren. In C++:

```

class Meier {
private:
    Bein *[2] meineBeine;
}
  
```

Und als UML-Diagramm: (siehe Abbildung 8).

4.5.6 Komposition

Notation siehe [13, S.2] unter »Komposition«. Dies ist eine besondere »benutzt«-Relation, nämlich »benutzt als Komponente« bzw. »besteht aus Komponente«, wobei die Existenz der benutzen Klasse (der Komponente) allein, ohne als Komponente benutzt zu werden, keinen Sinn macht. Wird das Objekt der benutzenden Klasse gelöscht, so verschwinden daher auch die Objekte der existenzabhängigen Teile. Die Komposition entspricht der Verwendung von Attributen.

Nach üblicher Interpretation geht man davon aus, dass eine Klasse nur von einer anderen durch Komposition benutzt werden kann. Entsprechend setzt man die Kardinalitäten.

Ob man Komposition oder Aggregation verwendet, ist von der Art der Anwendung abhängig, nämlich davon, ob die benutzte Klasse sinnvollerweise allein existieren kann. Das unter »Aggregation« angeführte Beispiel kann entsprechend modifiziert werden. In C++:

```

class Meier {
private:
    Bein meineBeine[2];
}
  
```

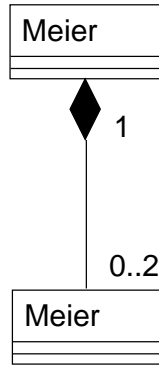


Abbildung 9: »Komposition zwischen Meier und Bein«

Abbildung 10:

```

    int aktiveBeine; //wieviele »gehen« noch
}

```

Und als UML-Diagramm (siehe Abbildung 9).

4.5.7 Kardinalität

Sie gibt für eine Assoziation an, wieviele Objekte einander zugeordnet sind. Notation siehe [13, S.2] unter »Multiplizität«.

4.5.8 abstrakte Basisklasse

Eine abstrakte Basisklasse ist eine Klasse, von der nur Ableitungen, aber keine Instanzen erzeugt werden können.

C++ Sie muss mindestens eine rein virtuelle Methode (**pure virtual**) haben. Im Sourcecode schreibt man dafür z.B.:

```
virtual void push(int)=0;
```

Eine nur virtuelle, aber nicht rein virtuelle Methode dagegen wäre z.B.:

```
virtual void push(int);
```

UML Notation durch einen kursiv gesetzte Namen der Klasse und der rein virtuellen Funktionen (wie in [13, S.1]) oder durch ergänzen von **«abstract»**¹³ zu diesen Namen. In UML können nur rein virtuelle Methoden ausgedrückt werden, nicht aber bloß virtuelle Methoden, die im Gegensatz zu rein virtuellen Methoden überschreibbar sind.

4.5.9 Vererbung

»Jeder Programmierer ist ein Angestellter, jede Kuh ist ein Säugetier«. Es kann kein Objekt geben, das nur ein Säugetier ist, und nicht weiter spezialisiert; es kann jedoch einen Angestellten Meier geben, der nicht weiter spezialisiert ist.

Hier geht es nicht um die Wirklichkeit, sondern um deren Modellierung durch Vererbung. Das Konzept der Ableitung (»Vererbung«) kann nämlich in zwei Arten benutzt werden:

Kategorisierung. die real existierenden Dinge mit ihren speziellen Typen einteilen in abstrakte Kategorien. Es gibt keine real existierende »Klasse Säugetier«, sondern es ist ein konzeptionelles Konstrukt zur Einteilung von Tieren. Erst am Ende dieser Ableitungen stehen Konzepte, die konkrete Instanzen haben

¹³oder nach [3] auch {abstract}, was jedoch kein offizielles UML ist.

Abbildung 11:

Abbildung 12:

können (wie hier Else:Kuh), es gibt keine Instanzen von Säugetier. Die Einteilung der Realität in Konzepte richtet sich nach bestimmten Eigenschaften, ist aber eigentlich willkürlich.

Dies ist: Man hat eine Menge von real existierenden Typen und baut darüber abstrakte Übertypen. Abstrakt heißt: diese Übertypen können keine Instanzen haben. Es ist eine Bottom-Up-Methode: von vielen Dingen ausgehen und daraus Kategorien bilden, um den Umgang mit den Dingen zu erleichtern. Die Kategorien heißen »abstrakte Basisklassen«, d.h. sie sind nicht instanzierbar und werden zur Kategorisierung eingesetzt.

Der Name von abstrakten Basisklassen muss in UML kursiv geschrieben werden, ggf. ergänzt durch ein <<abstrakt>> über dem Namen.

Allgemeine und spezielle Varianten unterscheiden. Dies wurde in Abbildung 10 ausgedrückt: Programmierer sind eine besondere Art von Angestellten, es gibt außer auch Angestellte, die weder Programmierer noch Manager sind. Von all diesen Klassen können Instanzen gebildet werden!

Natürlich sind Mischformen dieser beiden Arten von Vererbung möglich. Aber man sollte sich bei der Programmierung stets fragen, welche Art der Vererbung man nun benötigt. Am Beispiel der Hausübung 1 von Prog2: hier wurden OperatorAusdruck, GeradenAusdruck und VektorAusdruck zu einer Kategorie (abstrakten Basisklasse) »Ausdruck« zusammengefasst; sie haben keine gemeinsamen Attribute, sondern allein die Eigenschaft gemeinsam, dass sie einmal eingelesen werden müssen und dies gleichartig für alle Ausdrücke geschehen soll (Ausdruck a; cin >> a;). Eine weitere Gemeinsamkeit ist, dass sie alle Unterausdrücke eines Operatorausdrucks sein können. Ebenso für die Zusammenfassung von Gerade, Vektor und Punkt zu einer abstrakten Basisklasse »Wert«: sie haben keine gemeinsamen Attribute, sondern werden als Werte manipuliert und gespeichert.

Wenn man Probleme hat, sich eine Instanz einer Klasse an sich (z.B. »Wert«) vorzustellen, ist das ein Indiz dafür, dass es eine abstrakte Basisklasse ist.

Man kann nicht sagen, dass etwas eine Instanz einer allgemeinen Variante ist (statt dass die betreffende Oberklasse eine abstrakte Basisklasse ist), weil man einfach noch nicht weiß, zu welcher abgeleiteten Klasse das Objekt gehört - es ist unabhängig vom eigenen Wissen trotzdem schon eine Instanz einer abgeleiteten Klasse statt einer allgemeinen Variante.

Die Modellierung mit Vererbung nach »allgemeinen und speziellen Varianten« ist einfacher als mit abstrakten Basisklassen, denn hier müssen nicht alle Eigenschaften einem bestimmten abgeleiteten Typ zugeordnet sein. Die Modellierung mit »abstrakten Basisklassen« ist jedoch im Regelfall übersichtlicher und deshalb empfehlenswert: es sollte nichts geben, was nicht einem Teilbereich (einer abgeleiteten Klasse) zugeordnet werden kann (Modellierung in UML: Siehe Abbildung 11). Nur wenn es sehr viele Typen gleicher Art und nur wenige mit Ausnahmen gibt, so ist die Modellierung mit allgemeinen und speziellen Varianten besser.

Nun muss man unabhängig von der gewählten Art der Modellierung die Gemeinsamkeiten der Typen entdecken. Oft gibt es keine offensichtlichen Gemeinsamkeiten (im Gegensatz zu Lehrbüchern zur Objektmodellierung), wie im Beispiel oben aus Prog2.

Man kann Ausdrücke auch ohne Vererbung analysieren, indem man sagt: ein Ausdruck besteht stets aus einem Operator und einem Wert und zwei Verweisen auf andere Ausdrücke; alle diese Elemente können ggf. undefiniert sein. Dies ist programmiertechnisch jedoch unschön: denn alle Knoten sind maximal groß, enthalten oft undefinierte sinnlose Werte. Warum musst das so gemacht werden? Weil die Klassen so verwendet werden, dass sie über Zeiger angesprochen werden; am Zeiger aber kann man nicht erkennen, welche Werte definiert und undefiniert sind. Beispiel siehe Abbildung 12. Die einzige praktische Gemeinsamkeit ist also, dass es Zeiger gibt, die auf die Objekte zeigen, ohne dass man weiß, welche Untersorte es ist. Die Oberklasse wurde also eingeführt, damit Zeiger darauf zeigen können, ohne den Untertyp zu kennen.

Fragen zur Modellierung der Vererbung

- welche Art der Vererbung? Abstrakte Basisklasse?
- Welche Gemeinsamkeiten gibt es?

– In der Verwendung

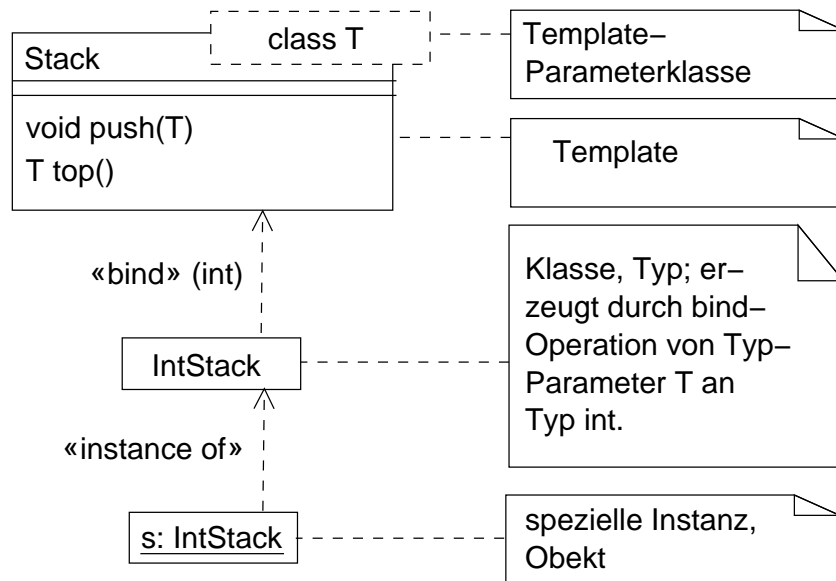


Abbildung 13: UML-Diagramm eines Klassentemplate

- * Zeiger sollen auf die Elemente zeigen können
- * Gemeinsame Verarbeitung, Speicherung
- In Attributen (einfacherer Fall)

4.5.10 Schnittstelle

Schnittstellen (Interfaces) sind Klassen mit nur rein virtuellen Methoden, in UML dargestellt durch `«interface»` im Kopf des Klassensymbols.

4.5.11 Template

Gegeben ist das UML-Diagramm eines Klassentemplate (Abbildung 13). Übersetzung nach C++:

```
template<class T>
class Stack{
public:
    // ...
    void push(T);
    T top();
private:
    // ...
};

typedef Stack<int> IntStack;
IntStack s;
// mit anonyem Typ auch »Stack<int> s;<, in UML aber nicht darstellbar
```

4.6 Sequenzdiagramme

In Sequenzdiagrammen kann genauer als in Aktivitätsdiagrammen ausgedrückt werden, welches Objekt was tut. Sie stellen die Interaktion von Objekten dar, Aktivitätsdiagramme dagegen die Aktivität von Komponenten. Notation siehe [13, S. 3]. Erläuterungen

- Objekte werden in der horizontalen Achse aufgetragen

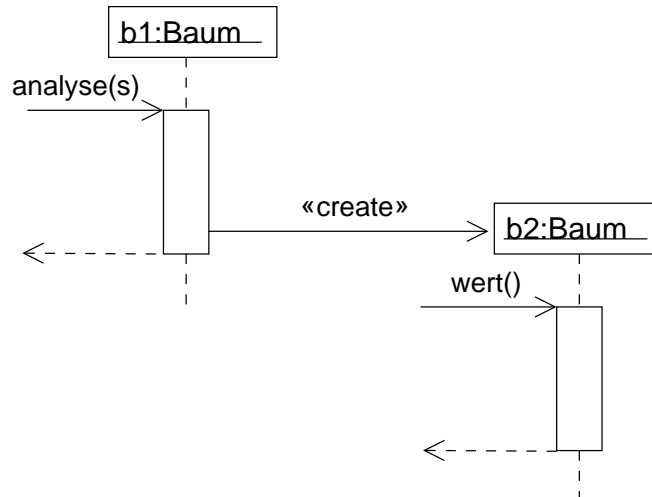


Abbildung 14: Ausdruckauswertung in Bäumen, Variante 1.

- die Zeit wird in der vertikalen Achse aufgetragen. Werden Nachrichten und Antworten als horizontale Pfeile eingezeichnet, geht man also von einer Zeitverzögerung $\Delta t = 0$ aus. Durch diagonale Pfeile kann eine Netzverzögerung der Nachrichten und Antworten angezeigt werden.
- Aktivitäten sind die Laufzeit einer Methode eines Objektes und werden auf der Lebenslinie des Objektes als längliche Kästen gezeichnet. Sie werden durch andere Objekte aufgerufen, dargestellt durch einen Pfeil, der mit dem Methodennamen und den Argumenten überschrieben ist.
- Endet eine Methode, so wird die Rückkehr (Antwort) zum aufrufenden Objekt durch einen gestichelten Pfeil angezeigt.
- In spitzen Klammern stehen Meta-Informationen, das sind Bemerkungen, was ein Symbol bedeutet, das von der üblichen Bedeutung abweicht. Zum Beispiel heißt ein Pfeil mit dem Zusatz `<<create>>`, dass ein Objekt von einem anderen neu erzeugt wird. `create` ist dabei kein festgelegtes Schlüsselwort, man hätte auch `<<new>>` o.ä. wählen können.

4.6.1 Beispiel: Ausdrucksanalyse mit Bäumen

Aufgabenstellung. Die Klasse `Baum` hat eine Methode `analyse`; wenn man `analyse` aufruft, dann erzeugt es einen Baum; ein Baum hat eine Methode `wert`, die den Wert des ausgewerteten Ausdrucksbaums liefert.

Umsetzung in ein Sequenzdiagramm. Bei der Umsetzung muss die gewöhnlich unpräzise sprachliche Aufgabenstellung entsprechend den Erfordernissen präzisiert werden. Lösung siehe Abbildung 14.

Umsetzung in C++.

```

string s;
cin >> s;
Baum b1;
Baum b2=b1.analyse(s);
// besser mit static-Methode: b1=Baum::analyse(s);
wert w=b2.wert();
  
```

In UML gibt es im Gegensatz zu C++ keine statischen Methoden, d.h. jede Methode muss einem Objekt zugeordnet sein. Die oben auskommentierte Optimierung ist also in einem UML-Sequenzdiagramm nicht darstellbar.

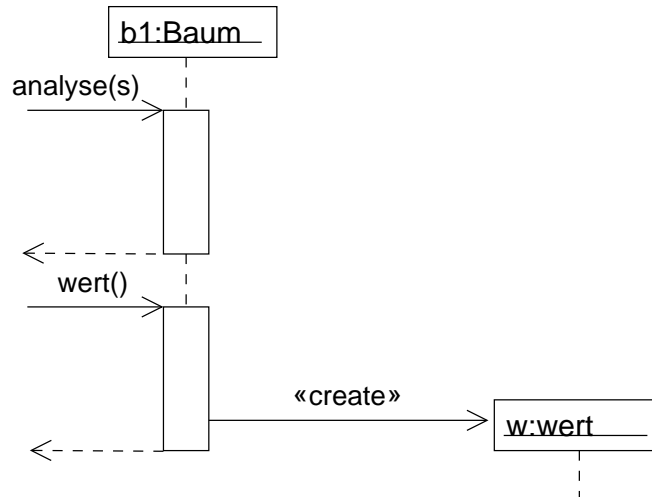


Abbildung 15: Ausdrucksauswertung in Bäumen, Variante 2.

Variation. Um nicht wie bisher zwei Objekte der Klasse Baum erzeugen zu müssen, kann man auch ein Objekt vom Typ Baum die Baumstruktur im eigenen Objekt aus dem zu analysierenden String erzeugen lassen. In UML dargestellt in Abbildung 15. In C++:

```

string s;
cin >> s;
Baum b1;
// b1 analysiert den Ausdruck und speichert den Ausdruckbaum
// in sich selbst:
b1.analyse(s);
// Beispiel, den Wert herauszuholen:
wert w=b1.wert();
  
```

5 Glossar

abstrakte Klasse: Ein technischer Begriff. »Abstrakt« meint hier nicht wie sonst in der Softwaretechnik »simuliert«, sondern eine Klasse, von der man keine Instanzen bilden kann, weil nicht alle Methoden definiert sind.

abstrakter Datentyp (ADT) Ein virtueller Datentyp, der nicht konkret realisiert ist - er existiert nicht in der Programmiersprache, aber in der Phantasie des Programmiers. ¹⁴ Zu unterscheiden von einem abstrakten Objekt, von dem im Gegensatz zu Typen keine Instanzen erzeugt werden können. Ein abstraktes Objekt ist von einem abstrakten Datentyp! Da die Zuweisung auf abstrakten Objekten nicht definiert ist (wie auch? Es kann je nur eine Instanz geben!), entsprechen die zugehörigen abstrakten Datentypen einer zustands- / bzw. variablenorientierten Definition (siehe »zustandsorientierte Klassendefinition«). Beispiel: Ein abstrakter Datentyp Stack (d.h. ohne eine Klasse als konkreten Datentyp zu definieren). Als Typ ist er das Modell eines in der realen Welt existierenden Stacks, hier modelliert durch eine Abbildung auf ein einfaches Konstrukt von Zeigern und Operationen. Zur Verbesserung der Modellbildung unterstützen Programmiersprachen das Klassenkonzept, d.h. einen echten Typ »Stack«, der durch den hier verwandten Zeiger nur simuliert wurde (daher abstrakter Typ).

```

//stack.h
struct StackRep;
typedef StackRep *Stack;
int top(Stack);
  
```

¹⁴Manche meinen mit »abstrakter Datentyp« die Realität selbst, die Grundlage für die Modellbildung durch Typdefinition. Dies ist jedoch falsch.

```

void push (Stack);
void push(Stack, int);
void pop(Stack);

// stack.c (Implementierung)
#include <stack.h>
struct StackRep {
    Knoten *K;
    int v;
}
// ...

// stackuse.c (Benutzung)
#include <stack.h>
// ...
Stack s1,s2;
push(s1,5);
push(s2,7);
int i=pop(s2);

```

abstraktes Objekt Ein \rightarrow Objekt ist das Exemplar eines Typs (der Klasse). Im Gegensatz dazu ist ein abstraktes Objekt nicht von einem Typ instanziiert, sondern simuliert nur ein Objekt - es hat keinen Namen wie ein richtiges Objekt, unter dem es angesprochen werden könnte, und es können keine weiteren Instanzen gebildet werden, d.h. ein abstraktes Objekt ist ein »Objekt ohne Klasse«, es existiert stets nur in einer Instanz. Das abstrakte Objekt existiert also nicht real in der Software als Objekt, sondern wird im logischen Sinne z.B. durch eine Sammlung von Funktionen simuliert (das meint »abstrakt«); damit entspricht es der »Komponente« im Sinne von COM, CORBA, UML. Auch der Typ dieses abstrakten Objektes existiert nicht real, sondern ist ein abstrakter Datentyp.

coding conventions (Codierkonventionen) Bezeichnen die übliche Art der Implementierung von Problemstellungen in einer Programmiersprache und übliche Schreibweisen, die jedoch nicht durch den Sprachstandard genormt sind. Beispiele:

Dateinamen Wenn eine .h-Datei nur eine Klasse enthält, benennt man sie nach dem Klassennamen.

Komponentenaufteilung Die Sprache C++ stellt selbst keine Möglichkeit bereit, Komponenten zu definieren, es gibt also kein Äquivalent zu einem PACKAGE von ADA oder einer Unit und Interface in TurboPascal. Hier behilft man sich mit Codierrichtlinien, indem man festlegt, was eine Komponente darstellen soll:

- die Schnittstelle (die öffentlich benutzbaren (»exportierten«) Bestandteile der Komponente) steht in der .h-Datei. Sie soll nur die Dinge enthalten, die der Compiler wissen muss, um die Benutzung der Komponente zu ermöglichen, die deshalb an verschiedenen Stellen im Sourcecode durch Inklusion sichtbar gemacht werden.
- die Implementierung (die privaten, internen Bestandteile der Komponente) steht in der gleich benannten .c-Datei (Internes, typischerweise die Implementierung der Komponente). So trennt man also eine Komponente in Einheiten der getrennten Übersetzung.

Inklusion Inklusion von .c-Dateien ist per Codierrichtlinie verboten!

Datenstruktur¹⁵ Reale, abstrakte Werte werden durch Programmvariablen modelliert. Die Datenstruktur ist nun die Abbildung dieser Variablen auf Werten die gemeinten Werte.

Entwicklungsorganisation Organisatorischer Rahmen der Softwareentwicklung, z.B. das Wasserfallmodell.

freie Funktion (free function) Sie werden meist außerhalb einer Klasse auf beliebigen Datentypen definiert, jedoch kann auch eine Klasse freie Funktionen definieren. Siehe auch \rightarrow Methode.

Inklusionswächter Inklusionswächter in Header-Dateien verhindern die Mehrfach-Inkludierung einer Header-Datei im Programm, denn das würde zu Fehlern führen. Das Argument der #ifndef- und der #define-Anweisung muss dem Headerdateinamen eindeutig zuzuordnen sein, ansonsten kann dieser String frei gewählt werden. Nach einer Kodierkonvention schreibt man solche Preprozessorconstanten aber immer groß. Beispiel:

```

//kompl.h
//wenn noch nicht definiert: ...
#ifndef KOMP_1_H
//...dann definiere
#define KOMP_1_H
//Inhalt der Header-Datei
#endif
//Ende der Definition

```

Für die `#include`-Anweisung zur Einbindung einer solchen Header-Datei gibt es zwei Möglichkeiten:

- suche in den Standard-Include-Verzeichnissen. Mit Hilfe der Compileroption `-I` können weitere solche Verzeichnisse angegeben werden.

```
#include<stack.h>
```

- suche im aktuellen Verzeichnis

```
#include"stack.h"
```

Klasse In C++ ist jede Klasse ein \rightarrow Typ; aber nur die Typen sind Klassen, die mit dem Klassnkonstrukt (in C++ `class`) definiert wurden. Denn natürlich sind z.B. die Typen `int`, `float` keine Klassen! Der Besitz mindestens einer Methode ist für Klassen keine definierende Eigenschaft! Es gibt auch Gegenbeispiele! Jedoch waren Klassen (und Records `struct` als deren Variante) natürlich die ersten Typen mit der Möglichkeit, darauf Methoden zu definieren.

Auf abstrakter Ebene der Softwaretechnik werden Klassen und Typen leider sehr oft miteinander identifiziert.

Komponente (auch: Modul, Package). Komponenten sind Sourcecode-Kollektionen (z.B. mehrere Klassen) und können daher nicht als »Klasse« in C++ implementiert werden; stattdessen behilft man sich mit \rightarrow coding conventions. Definierende Eigenschaften einer Komponente im allgemeinsten Sinne:

- ein austauschbares Teilsystem des Gesamtsystems
- stellt eine bestimmte Funktionalität zur Verfügung (eine Komponente bearbeitet eine inhaltliche Aufgabenstellung)
- hat eine exakt definierte öffentliche Schnittstelle
- ist als Arbeits- und Auftragseinheit abgegrenzt
- bildet auch in der Implementierung eine Einheit, z.B. Einheit der getrennten Übersetzung, Objektbibliothek, eigenständiges Programm.

konkreter Datentyp Ein vordefinierter Datentyp (wie `int`, `float`); oder eine Klasse, die wertorientiert definiert wurde. Nur die Klasse bietet die Möglichkeit der wertorientierten Definition, so dass also jeder selbst definierte konkrete Datentyp eine Klasse ist. Durch die wertorientierte Definition ist mit diesen Klassen alles möglich, was es mit vordefinierten konkreten Datentypen möglich ist (z.B. Zuweisung, ermöglicht durch Klassenkonstruktoren).

Lebenszyklusmodell (lifecycle model) Andere Bezeichnung für das Wasserfallmodell (Royce, ca. 1970). Der Gedanke ist, dass eine Software während ihrer Produktion und ihres Betriebs verschiedene Phasen der Entwicklung und Reifung durchläuft.

Methode (member function) Es sind Funktionen, die an einem Objekt hängen, das vom Typ Klasse `class` (oder `struct` als Sonderfall) ist. Siehe auch \rightarrow freie Funktion.

Objekt Eine Instanz (auch: »Exemplar«) einer Klasse. Zu einem Objekt gehören die Methoden und der Zustand (d.h. die Attribute mit ihrer aktuellen Belegung).

software engineering Der englische Fachbegriff für »Softwaretechnik«.

Software-Produktions-Prozess Die Realisierung eines Software-Produktionsverfahrens, d.h. einer \rightarrow Entwicklungsorganisa

Software-Prozess \rightarrow Software-Produktions-Prozess

Subsystem Eine Komponente, die aus Unterkomponenten besteht, wird oft Subsystem genannt. Ein »System« (die Software) besteht dann also aus Subsystemen und diese aus Komponenten.

Systemmodellierung Ein anderer Begriff für »Analysemodell«, eine Teilphase im Wasserfallmodell.

Typ (auch: »Datentyp«) Zu einem Typ gehört eine Menge von Werten und Operationen auf ihnen. Er kann ein »→konkreter Datentyp« oder ein »→abstrakter Datentyp« sein. Typen sind Dinge wie z.B. Klassen, die nicht benutzt werden können, ohne eine Instanz davon zu erzeugen.

wertorientierte Klassendefinition Das Gegenteil ist die (einfachere) →zustandsorientierte Klassendefinition. Am Beispiel der Implementierung eines Stack:

- Bei einem konkreten Datentyp `stack` (d.h. bei wertorientierter Klassendefinition) ist ein Stack der Wert einer Variablen vom Typ `stack`. Dieser Wert kann also anderen Variablen vom Typ `stack` zugewiesen werden.
- Bei einem nicht konkreten Datentyp `stack` (d.h. bei zustands- / variablenorientierter Klassendefinition) ist ein Stack die Variable vom Typ `stack` selbst. Ihre Werte sind nur Zahlen (allgemein: Stackelemente) und kein Stack selbst. Deshalb ist die Zuweisung eines Stack an eine andere Variable vom Typ `stack` nicht mit dem Zuweisungsoperator möglich, sondern muss durch Zugriffe mit den Operationen `push` und `pop` simuliert werden.

zustandsorientierte Klassendefinition Siehe →wertorientierte Klassendefinition.

6 Durchgängiges Fallbeispiel zum Wasserfallmodell¹⁶

6.1 Analyse

6.1.1 Planungsphase

1. Produkte und Prozesse.

- Welches Produkt soll entstehen? Ein Programm, nämlich die fertige Hausübung 4 von Programmieren 2.
- In welchen Handlungsablauf (»Prozess«) wird das fertige Produkt eingebunden sein?
- Wie soll das Produkt in diesen Handlungsablauf integriert werden?

2. Personal- und Verantwortlichkeitsplanung. Personal ist die eigene Arbeitskraft, dazu die Arbeitskraft der Tutoren und ggf. der weiteren Mitglieder der Arbeitsgruppe. Was können die Arbeitskräfte, mit denen man zusammenarbeitet (Qualifikationen), also hier auch: was kann ich selbst? Wer will mit mir zusammenarbeiten? Wann ist wer verfügbar? Was erwarte ich von wem genau? Wieviel Prozent ihrer Arbeitskraft wollen diese Personen einsetzen?

3. Wirtschaftlichkeitsbetrachtung. Annahme: Am Ende der Planungsphase wurde entschieden, das Projekt durchzuführen, da es wirtschaftlich ist.

Zieldefinition ist also: die fertig abgegebene Hausübung 4.

6.1.2 Anforderungsanalyse und Analysemodell

1. Anforderungsanalyse.

Die Aufgabe erfordert die Erstellung einer Anwendung zur Auswertung geometrischer Zuweisungen $Variable = Ausdruck$. Ausdruck ist ein Punkt, eine Gerade, der Schnittpunkt zweier Geraden, die Frage ob zwei Geraden parallel sind.

2. Analysemodell.

Aktoren und Anwendungsfälle.

- Aktoren: der Endbenutzer als einziger Akteur

¹⁶Anmerkung: Dies ist keine Musterlösung, sondern eher eine unvollständige Notizsammlung!

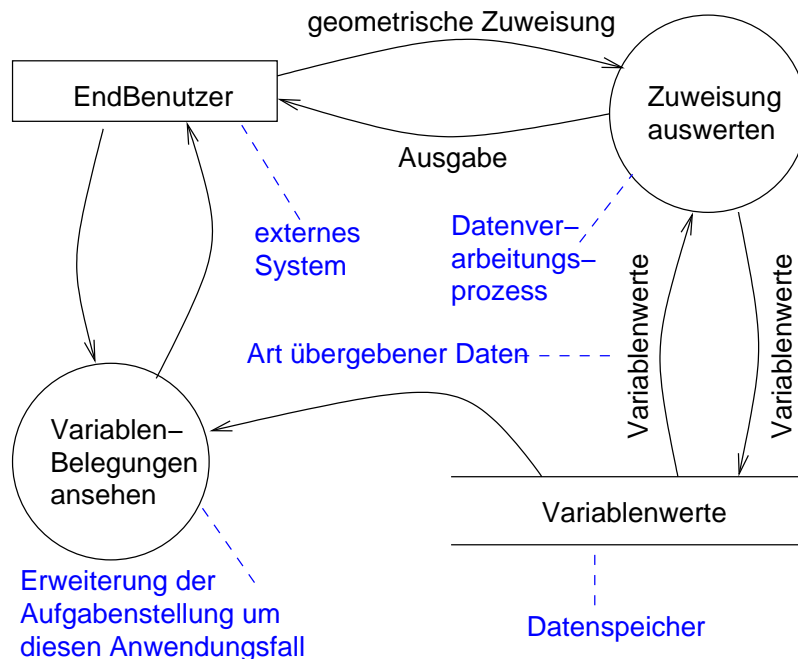


Abbildung 16: Datenflussdiagramm: Notation an einem Beispiel

- Anwendungsfälle: wie wird das System benutzt? Anwendungsfall: Eingabe eines Ausdrucks im Feld A (siehe Zeichnung 1). Jetzt muss überlegt werden, wie das System sinnvollerweise reagieren soll: Es stellt dem Benutzer ein Fenster zur Verfügung (siehe Zeichnung 1). Es prüft den eingegebenen Ausdruck, gibt entsprechend einen Fehler (Sonderfall als Teil des Anwendungsfalls) aus oder berechnet und speichert den Wert des Ausdrucks und gibt ihn gleichzeitig im Feld B (siehe Zeichnung 1) aus.
Sonderfälle: der Ausdruck ist nicht korrekt:
 - syntaktischer Fehler im Ausdruck. Überlegen: Reaktion des Systems?
 - Verwendung undefinierter Variable im Ausdruck. Überlegen: Reaktion des Systems?

Modell der Benutzeroberfläche.

Funktionale Darendekomposition. Welche Daten müssen gespeichert werden? Variablen und ihre Werte. Datenflussdiagramm siehe Abbildung 16.

Datendiktionär.

geometrische Zuweisungen: Text, der eingegeben und verarbeitet wird. Definiert durch folgende Grammatik: (siehe [23], Übungen 2 und 3).

Variable: Wort, dem ein Wert zugeordnet ist. Weiter wäre der Syntax von Variablen zu definieren.

Wort: Aus einem der Typen Punkt, Vektor, Gerade.

Punkt: definiert durch einen Ortsvektor in R^3 .

Ortsvektor: Ein Vektor vom Ursprung zu einem Punkt.

Vektor: Er ist gegeben durch seine Koordinaten, dargestellt durch drei float-Werte.

Gerade: dargestellt in Punkt-Richtungsform.

Schnittpunkt: ...

Parallelen: ...

Gleichheit: ...

Klassendiagramm. Punkte können dargestellt werden durch einen Ortsvektor. Siehe Abbildung 17.

Abbildung 17: Konzeptionelles Modell zu Hausübung 4 (Klassendiagramm)

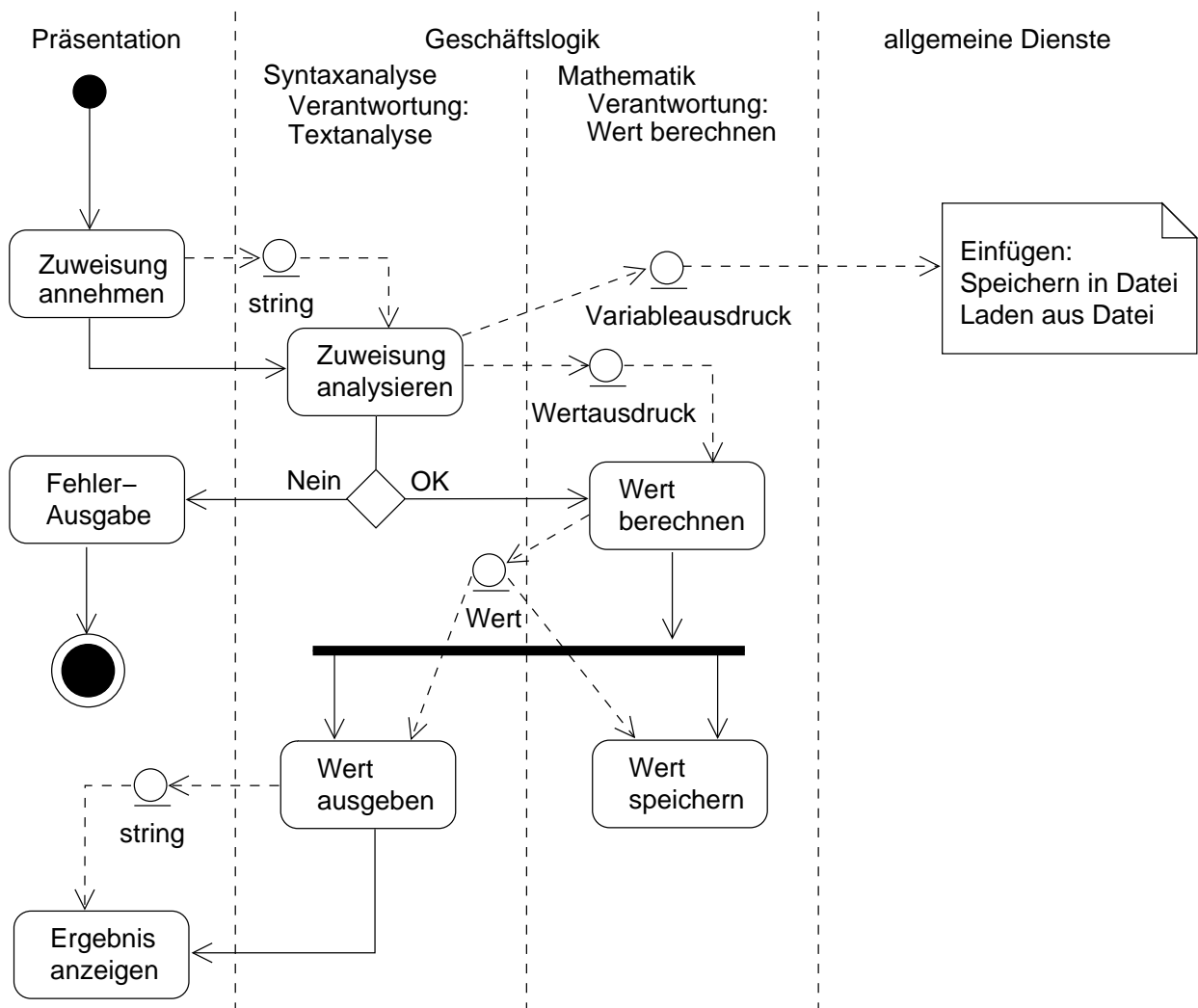


Abbildung 18: Aktivitätsdiagramm zu Hausübung 4

6.2 Entwurf

6.2.1 Grobentwurf

1. Architekturmuster.
Es wird das typische Architekturmuster einer einfachen unverteilter interaktiven Anwendung gewählt.
2. Aktivitätsdiagramm.
3. Eigentliche Modularisierung.
Verantwortlichkeiten (aus Können und Wissen, also aufgrund der Qualifikation) der auf dem gewählten Architekturmuster und dem Aktivitätsdiagramm basierenden Komponentenaufteilung (Komponentendiagramm vgl. 19):

Präsentation. Geforderte Qualifikation: graphische Benutzeroberflächen programmieren können. Verantwortlichkeiten:

- Management der Benutzeroberfläche. Der Sourcecode wird eine ähnliche Gestalt haben wie:

```
s1=eingabefeld.text;
string s2=auswerte(s1);
```

```
ausgabefeld.setText(s2);
```

- Eingabe: Eine Zuweisung annehmen, als **string** an die Komponente »Geschäftslogik« übergeben
- Ausgabe: Einen Wert (Fehler oder Ergebnis) als **string** von der Komponente »Geschäftslogik« übernehmen und ausgeben.

Geschäftslogik.

workflow. Verantwortlichkeiten:

- Zuweisung in Variablenausdruck und Wertausdruck spalten.
- Wert nach Berechnung unter der entsprechenden Variable speichern.

Syntax. Verantwortlichkeiten: Konvertierung von **strings** in verzeigerte Strukturen **wertausdruck**.
Qualifikationen:

- Analyse von Strings
- Syntax geometrischer Zuweisungen kennen

Wert. Beherrscht alle Mathematik zur Berechnung von Werten aus Wertausdrücken. Qualifikationen:

- Datenstrukturen zum Abspeichern von Variablen und ihren Werten kennen.
- geometrische Operationen beherrschen

Allgemeine Dienste.

intern Speichern Verantwortung: verwaltet die aktuelle Variablenbelegung durch Speicherung im Hauptspeicher.

in Datei Speichern¹⁷ Verantwortung: Speicherung einer Variablenbelegung in einer Datei. Nötige Qualifikationen der Entwickler: Dateimanagement können; wissen, wie man komplizierte Datenstrukturen in Dateien speichert.

4. Schnittstellen der Komponenten.

Präsentation / workflow. Zwischen diesen beiden Komponenten werden die eingegebenen / auszugehenden Strings ausgetauscht. Wie kann dieser Datentransfer von der Komponente **workflow** organisiert werden? Beispiele:

- Die Komponente **workflow** prüft alle 35ms, ob in einem Textfeld ein String eingegeben wurde, und wertet ihn aus. Das wäre natürlich sehr schlechter »Programmierstil« ...
- Wenn in der Komponente **Präsentation** nach Eingabe eines Strings der Knopf »Auswerten« gedrückt wurde, ruft die Komponente **Präsentation** die Interface-Funktion **string auswertung(string)** der Komponente **workflow** auf¹⁸; ihren Rückgabewert setzt sie mit einem Methodenaufruf in das Objekt **Ausgabefeld**, in Delphi z.B. mit der Methode **ausgabefeld.setText(string)**. Es besteht stets eine Kopplung zwischen den Elementen der graphischen Benutzeroberfläche und entsprechenden Variablen.

workflow / Syntax und workflow / Wert. Wieder gibt es mehrere Möglichkeiten der Schnittstellendefinition, die sich in ihrer Qualität unterscheiden:

- Entweder werden Objekte der Klassen **Vektor**, **Gerade** usw. übergeben. Dazu müssten jedoch auch die Komponenten **Syntax** und **workflow** die Mathematik und die entsprechenden Klassendefinitionen kennen, was eigentlich rein zur Komponente **Wert** gehören sollte. Diese Aufteilung ist also ungünstig, da sie der Kohäsion der Komponentenaufteilung widerspricht.
- Oder es werden Strings übergeben. Die Komponente **Syntax** würde den String nur auf syntaktische Korrektheit überprüfen und zurückgeben, worauf **workflow** ihn an die Komponente **Wert** übergibt. Diese muss dann wie auch die Komponente **Syntax** die Syntax der Strings kennen und würde zur Auswertung dieselben Schritte durchführen, die **Syntax** zur Überprüfung durchgeführt hat. Wiederum ist diese Aufteilung ungünstig, da sie der Kohäsion der Komponentenaufteilung widerspricht.

¹⁸d.h. die Interface-Funktion ist in dieser Komponente definiert.

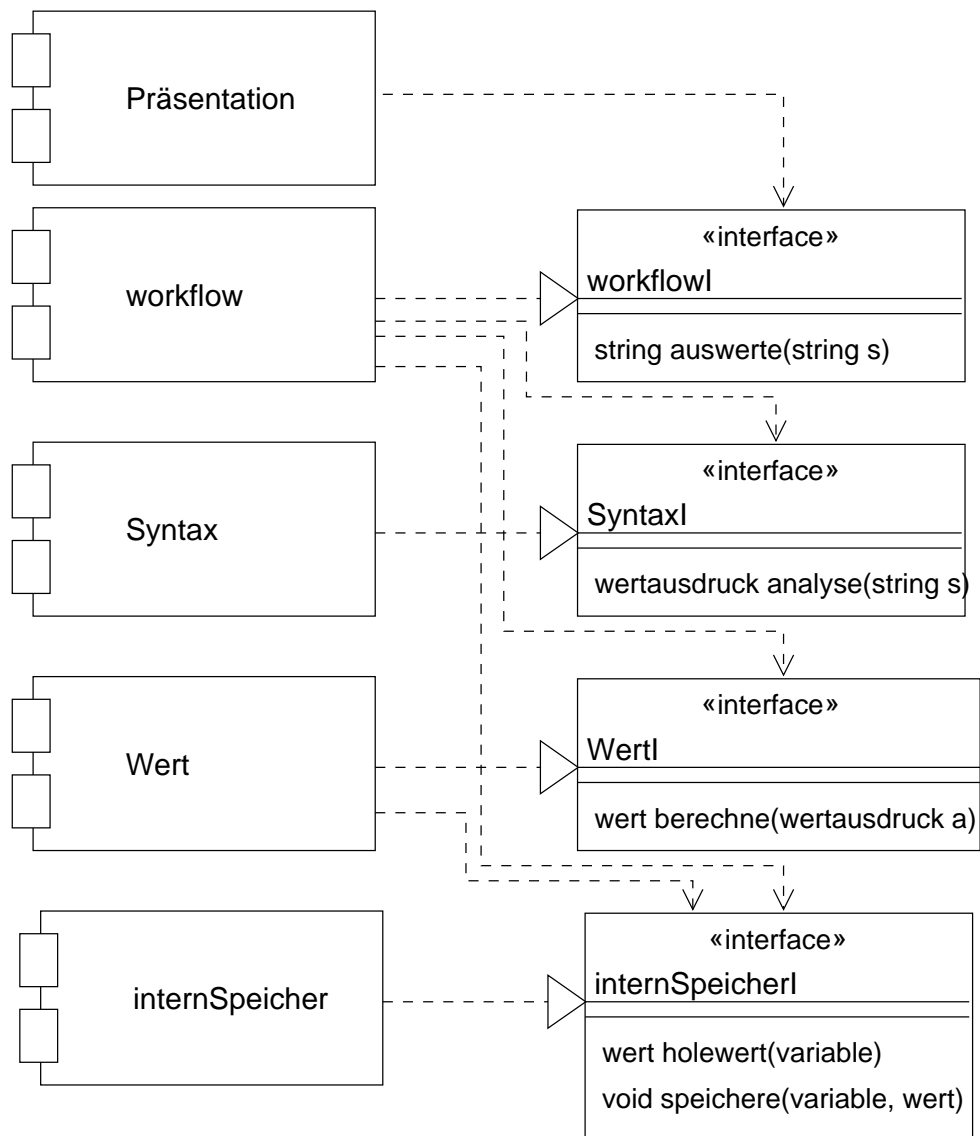


Abbildung 19: Komponentenaufteilung in UML

- Was ist die Lösung dieses Problems? Die Schnittstelle werde über »Strings in aufbereitete Form« realisiert, bei denen die Operationen wie Schnittpunktberechnung noch nicht durchgeführt wurden. Diese aufbereitete Form soll von Wert leichter weiterzuverarbeiten sein als Strings, ohne Kohäsionsverlust. Diese aufbereitete Form entspricht der in der vorhergehenden Konzeptanalyse (siehe Klassendiagramm in Abbildung 17) gefundenen Darstellung einer geometrischen Zuweisung als Variablenausdruck und Wertausdruck.

Die Schnittstellen `workflow / Syntax` und `workflow / Wert` enthalten also eine Klasse `Zuweisung`, die dieses Konzept implementiert. Solch eine große Schnittstelle ist eigentlich schlecht, aber die Zusammenfassung von `Wert` und `Syntax` zu einer großen Komponente wäre noch schlechter (da dann nur noch von Hackern beherrschbar). Die Implementierung der Klasse `Zuweisung` geschieht am besten in der Komponente `Syntax`, da ja dort die Entwickler arbeiten, die Strings analysieren können.

Wert / internSpeicher `Wert` übergibt mit `wert holewert(variable)` eine Variable und erhält dafür ein Objekt vom Typ `wert` zurück. Diese Schnittstelle benötigt also die Definition der Klasse `Wert` (vgl. Abbildung 17), die am besten in der Komponente `Wert` implementiert wird, da ja dort die Entwickler arbeiten, die die Mathematik der Anwendung kennen.

Zur Implementierung von der Klasse `Wert` benötigt man Verebung (»Wert ist ein Punkt oder ein Vektor oder eine Gerade ... «), zur Implementierung von `Zuweisung` Zeiger (aufgrund rekursiver Definition der eingegebenen `Zuweisung`).

6.2.2 Feinentwurf

6.3 Implementierung

Jede Komponente wird in einer eigenen `.c`-Datei implementiert; es gibt also: `praesentation.c`, `workflow.c`, `syntax.c`, `wert.c`, `internspeicher.c`.

Jede Schnittstelle (d.h. ihre Funktionen bzw. Methoden) wird in einer zugehörigen `.h`-Datei deklariert; es gibt also: `workflow.h`, `syntax.h`, `wert.h`, `internspeicher.h`. Die Implementierung erfolgt in der zugehörigen `.c`-Datei. Unter Verwendung freier Funktionen statt der in UML nötigen Klassen z.B.:

```
//workflow.h
string auswerte(string);
```

Jede Komponente, die diese Schnittstelle benutzt, muss die Deklaration der Schnittstelle inkludieren:

```
//praesentation.c
#include <workflow.h>
```

Schnittstellen werden gewöhnlich aber nicht als freie Funktionen, sondern als abstrakte Basisklasse mit entsprechenden Methoden definiert.

6.3.1 Komponente internSpeicher

Schnittstellendefinition:

```
//internspeicher.h
#include <wert.h>
#include <variable.h>
wert holewert(variable);
void speichere(variable,wert);
```

Dass diese Schnittstelle von der Komponente `Wert` abhängt (wo der Datentyp `wert` definiert wird) und außerdem von der Datei `variable.h` (wo der Datentyp `variable` definiert wird), kann im UML-Komponentendiagramm (Abbildung 19) nicht ausgedrückt werden.

Schnittstellenimplementierung:

```
//internspeicher.c
#include<internspeicher.h>
wert sp[1000];
```

```

wert holewert(variable){
    //...
}
void speichere(variable,wert){
    //...
}

```

Diese Schnittstelle implementiert ein abstraktes Objekt, aber keinen Typ, von dem mehrere Instanzen möglich wären. Die objektorientierte Implementierung sieht anders aus: die Interface-Klasse `internSpeicherI` ist eine abstrakte Basisklasse; die Klasse `internSpeicher` der Komponente ist eine Ableitung dieser abstrakten Basisklasse, die ihre Methoden implementiert (und also diese Schnittstelle hat) und weitere Dinge im `private`-Teil hinzufügen kann. Schnittstellendefinition:

```

//internspeicher.h
class internspeicher:internspeicherI {
public:
    wert holewert(variable);
    void speichere(variable,wert);
private:
    wert sp[1000];
};

```

6.3.2 Komponente Syntax (Prototyp)

Die Aufgabenstellung wird hier dahingehend vereinfacht, dass die eingegebenen Strings vollständig geklammerte binäre Ausdrücke aus ganzen Zahlen und den Operationen `+` und `·` sein sollen, also z.B. $((2 + 3) \cdot 5)$. Wie bei der eigentlichen Problemstellung gibt es zur Auswertung die Komponenten `Syntax` und `Wert`, die von einer Komponente `workflow` gesteuert werden (vgl. Abbildung 19).

Die einzige Änderung ist die Verwendung des Datentyps `Ausdruck*` als Rückgabewert, trotz dass Zeiger in Interfaces eigentlich vermieden werden sollen.

Schnittstellendeklaration bei Implementierung der Komponente als abstraktes Objekt:

```

//syntax.h
#ifdef ...
...
#include<string>
class Ausdruck;
// Vorausdeklaration, damit der Compiler gleich "Ausdruck*"
// akzeptiert, ohne zu wissen, was das ist. Alternativ waere
// #include<ausdruck.h> moeglich.
namespace SyntaxNS {
    Ausdruck* analyse(string);
}
#endif

```

Benutzung der Komponente `Syntax` durch die Komponente `workflow` z.B.:

```

//workflow.c
#include<syntax.h>
#include<ausdruck.h>
using namespace SyntaxNS;
int main () {
    Ausdruck *a;
    string s;
    cin >> s;
    a=analyse(s);
}

```

Wie soll nun eine Variable vom Typ `Ausdruck` aufgebaut sein (gleichzeitig Prototyp des Typen `wertausdruck`), in die die eingegebenen Strings transformiert werden? Sie bestehe aus einer verzeigerten Baumstruktur von

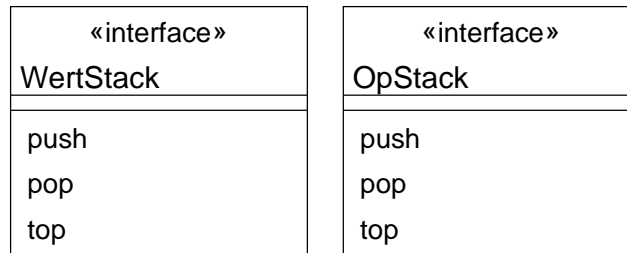


Abbildung 20: Klassendiagramm der Interface-Klassen zur Stack-Komponente

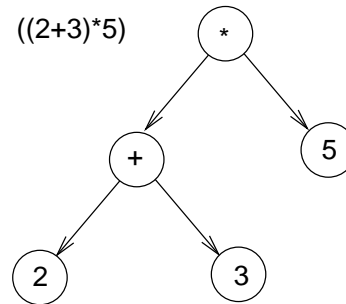


Abbildung 21: Vollständig geklammerte binäre Ausdrücke in einem Syntaxbaum

Operanden (Baumknoten) und Argumenten (siehe Abbildung 21). Die Schnittstellenfunktion `Ausdruck* analyse(string)`; kann die Transformation von Strings in solche Ausdrücke rekursiv oder mit Stacks durchführen. Im folgenden werden zur Implementierung in `syntax.c` Stacks verwendet:

```

#include<syntax.h>
#include<ausdruck.h>
#include<WertStack.h>
#include<OpStack.h>
// Benutzung: ws_push() und os_push()
  
```

Die beiden Stacks Wertstack und Operandenstack wurden als abstrakte Objekte implementiert und müssen daher einzeln und unter anderem Namen inkludiert werden, auch ihre Zugriffsfunktionen müssen andere Namen haben: `ws_push()` bzw. `os_push()`. Besser wäre natürlich wieder die objektorientierte Lösung: Eine Komponente Stack stellt zwei Interfaces mit Klassen zur Verfügung, eine Klasse für einen Wertstack und eine für einen Operandenstack. Bei Benutzung werden Instanzen dieser Klassen erzeugt:

```

#include<syntax.h>
#include<ausdruck.h>
#include<WertStack.h>
#include<OpStack.h>
WertStack ws;
OpStack os;
// Benutzung: ws.push() und os.push()
  
```

Weil UML-Komponenten aber stets abstrakte Objekte sind, können Klassen als Interfaces nicht in einem UML-Komponentendiagramm dargestellt werden. Diese Interface-Klassen gehören zu keiner Komponente und werden in einem geeigneteren Klassendiagramm dargestellt (siehe Abbildung 20).

Entwurf eines Algorithmus für die Methode `Ausdruck* analyse(string)`; zur Transformierung eines Strings in einen solchen Ausdrucksbaum mit Hilfe der Wert- und OperandenStacks:

- öffnende Klammern ignorieren
- Werte auf den WertStack legen, Operanden auf den OperandenStack legen

- bei schließender Klammer die beiden obersten Werte und den obersten Operanden mit Zeigern zu einem Knoten verbinden, dessen Komponenten vom Datentyp Ausdruck sind, und den Zeiger auf den Knoten wieder auf den WertStack ablegen.

Am Ende liegt auf dem Wertstapel ein vollständiger SyntaxBaum aus Objekten vom Typ Ausdruck, die mit Zeigern untereinander verbunden sind und worin alle Teilbäume (Knoten), die einmal auf dem Wertstapel gelegen haben, enthalten sind. Die Definition des Datentyps Ausdruck ist schwierig, denn er muss Zahlen und Operanden enthalten können.

7 Übungen und Lösungen

7.1 Blatt 4 Aufgabe 1

1. Geben sie informal ein Beispiel für

- zu hohe Kopplung: das sind zu hohe Abhängigkeiten der Klassen untereinander. Beispiel: Wenn viele Klassen miteinander befreundet sein müssen, damit eine Problemstellung gelöst werden kann. Befreundet heißt: eine Klasse kennt die Interna einer befreundeten Klasse. Hohe Kopplung heißt, Dinge von anderen Klassen wissen zu müssen, die zu sehr ins Detail gehen (wozu friend-Klassen deklariert werden müssen), d.h. wenn man zu viele Dinge auf einmal wissen muss. Beispiel: wenn man, um Geraden zu definieren, wissen muss, wie die Vektoren in der benutzen Klasse vector intern dargestellt sind.
- zu geringe Kohäsion: eine zu geringe thematische Zusammengehörigkeit. Das passiert, wenn man in eine Einheit (in eine Komponente) Dinge hineinpackt, die inhaltlich nicht dazugehören. Beispiel: Wenn eine Komponente syntaxanalyse gleichzeitig eine Primzahlprüfung durchführen kann.
- Verletzung des Geheimnisprinzips: Wenn man in einer Komponente Wissen über die Implementierung (die Interna) einer anderen Komponente nutzt, das man eigentlich nicht wissen dürfte; d.i. wenn man seine Komponente von mehr abhängig macht als die Informationen, die die Schnittstelle zur Verfügung stellt. Beispiel:

```
struct menge {
    menge ();
    void einfuege (int);
    bool enthalten ();
    int a[100];
    int i;
};
```

Wenn nun ein Benutzer nicht über die Methode m.einfuege(5); ein Element einfügt, sondern direkt über m.a[i]=5; ++m.i;, so ist das eine Verletzung des Geheimnisprinzips. In C++ wurde public und private entwickelt, um die Verletzung des Geheimnisprinzips in der Sprache zu verhindern. Werden diese Möglichkeiten jedoch nicht richtig eingesetzt (nicht dem Gedanken der Implementierung entsprechend), so handelt es sich trotzdem noch um eine Verletzung des Geheimnisprinzips.

- Mit welcher Art von UML-Diagramm lässt sich folgende informale Beschreibung darstellen:
»Zuerst erzeugt ein Objekt c der Klasse client ein Objekt der Klasse (»vom Typ«) transaction. Dann ruft c die Methode setaction mit den Parametern a und b dieses neuen Objekts der Klasse transaction auf. Diese Methode setaction aktiviert zuerst die Methode setval(c,d), die zum Objekt p der Klasse proxy gehört, und dann die gleiche Methode mit den Parametern e und f.« Zur Darstellung eignet sich ein Sequenzdiagramm.
- Stellen Sie den Sachverhalt mit der vorgeschlagenen Art von Diagramm dar. Siehe Abbildung 22.
- Erläutern Sie die Gemeinsamkeiten und die Unterschiede von Aktivitäts- und Sequenzdiagramm in UML.

Gemeinsamkeiten: Beides sind dynamische Modelle der Software.

Unterschiede: Das Aktivitätsdiagramm stellt die Aktivität von Komponenten durch Zustände und ihre Übergänge dar, das Sequenzdiagramm dagegen die Interaktion von Objekten in einem Szenario. In einem Aktivitätsdiagramm können Bedingungen und Schleifen formuliert werden, in einem Sequenzdiagramm kann nur der Programmablauf bei einer konkreten Bedingung oder Schleife angegeben werden.

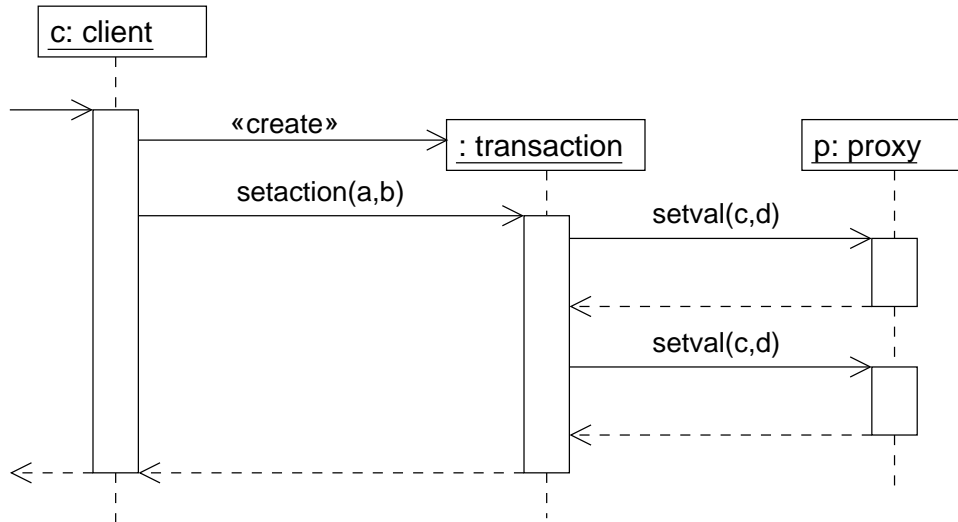


Abbildung 22: Sequenzdiagramm zu Übungsblatt 4 Aufgabe 1

7.2 Blatt 4 Aufgabe 2

Betrachten Sie folgenden ADA-Code eines »Package«:

```

// Schnittstelle
PACKAGE STACK IS
  PROCEDURE push (e: INTEGER);
  // ...
END STACK;

// Implementierung
PACKAGE BODY STACK IS
  a: ARRAY(1..100) OF INTEGER;
  index: RANGE 0..100:=0;
  PROCEDURE push (e: INTEGER) IS
    BEGIN ... END;
  // ....
END STACK;
  
```

1. Was wird hier implementiert? Ein Datentyp, eine Klasse, ein Objekt,...?
Was beschreibt dieses Stück ADA? Eine Klassendefinition (d.i. eine Art Typdefinition) oder eine Objektdefinition? Eine Klassendefinition müsste durch eine Ableitung von dieser Klasse benutzt werden, z.B.:

```

s: STACK;
s.push(5); // Methodenaufruf push des Objektes s vom Typ der
           // Klasse STACK.
  
```

Es könnte sich auch um eine Objektdefinition handeln. Wie würde es dann benutzt werden? Wenn es nur ein Objekt vom Typ STACK gibt, ist bereits folgender Aufruf eindeutig:

```

// ggf. USING PACKAGE STACK;
push(5); // Methodenaufruf
  
```

Solche Objekte, die ohne Ableitung von einer Klasse definiert werden, heißen »abstrakte Objekte« (siehe oben). Tatsächlich meint die ADA-Syntax solche abstrakten Objekte.

Ein astraktes Objekt hat einen Zustand (aktuelle Variablenbelegung, hier Initialisierung des index mit 0); eine Klasse dagegen, von der »richtige« Objekte abgeleitet werden, ist zustandslos. Ein Objekt ist stets dadurch charakterisiert, dass es einen änderbaren inneren Zustand hat.

Wäre es möglich, mit ADA ein PACKAGE zu implementieren, das kein abstraktes Objekt, sondern ein Typ ist? Ja, z.B. eine Serie von mathematischen Funktionen, denn das impliziert noch keinen eigenen inneren Zustand. Beispiel:

```
// Schnittstelle
PACKAGE Math IS
  PROCEDURE Sinus (t: FLOAT);
  // ...
END Math;

// Implementierung
PACKAGE BODY Math is
  // ...
END Math;
```

2. Geben Sie

(a) Eine entsprechende Definition in C++ an.

Was wäre die Umsetzung des ADA PACKAGE STACK in C++? Realisiert werden müssen folgende Eigenschaften:

- Trennung von Schnittstelle und Implementierung
- Realisierung eines abstrakten Objektes
- eine Einheit der getrennten Übersetzung, wobei die abhängige (benutzende) Übersetzungseinheit nur die Schnittstelle zur Übersetzung benötigt

Ein genau entsprechendes Konzept gibt es in der Sprache C++ nicht:

- Bei einem namespace besteht keine Trennung in Schnittstelle und Implementierung.
- Das Klassenkonzept sieht keine abstrakten Objekte vor. Ein ADA-PACKAGE ist aber auch keine Klasse, da es kein Typ, sondern ein abstrates Objekt ist.
- Man verwendet zur Umsetzung in C++ daher coding conventions, d.i. die »übliche Art der Implementierung«. Hier geschieht dies durch eine Aufteilung in .c-Datei und .h-Datei, um Einheiten der getrennten Übersetzung zu erhalten. Bei ADA sind solche coding conventions unnötig, da für abstrakte Objekte ja wie auch in anderen Programmiersprachen das offizielle Konstrukt PACKAGE zur Verfügung steht.

```
// stack.h
#ifndef STACK_H
#define STACK_H
void push(int e);
int pop();
#endif
// stack.c
#include<stack.h>
int space[10];
int index=0;
void push(int e) {
  if (index<10) {
    ++index;
    space[index]=e;
  }
  else
    //...
};
int pop() {
  //...
};

// main.c (benutzender Code)
```

```
#include<stack.h>
push(5);
```

(b) Ein adäquates UML-Diagramm an.

Das Konzept »Package« der UML ist eine beliebige Sammlung von Definitionen, das Konzept »Komponente« der UML dagegen ist semantisch höhergradig, denn eine Komponente hat immer eine definierte Schnittstelle. Ein ADA-Package hat eine definierte Schnittstelle und entspricht deshalb einer UML-Komponente.

3. ADA Packages sind Einheiten der getrennten Übersetzung. Was kann in C++ getrennt übersetzt werden? Kann Ihre C++ Variante des ADA Package getrennt übersetzt werden?

In C++ wird jede Datei als Einheit der getrennten Übersetzung verwendet. Entsprechend kann auch die oben dargestellte C++ Variante des ADA Package getrennt übersetzt werden, weil sie in einer eigenen Datei codiert wurde.

4. Welche Unterstützung bietet ADA zur Realisierung des Geheimnisprinzips an, wie ist es in Ihrer C++ Variante umgesetzt?

Jedes ADA Package besitzt eine definierte Schnittstelle zur Benutzung; im Beispiel werden zum Zugriff auf den Stack die Prozeduren `push` und `pop` bereitgestellt, der Nutzer kann (oder sollte) nicht direkt auf die im `PACKAGE BODY Stack` definierten Variablen `space` und `index` zugreifen. Obige C++ Variante setzt die Schnittstelle des ADA Package in eine `.h`-Datei um, den `PACKAGE BODY` aber in einer `.c`-Datei. So wird auch vermieden, dass der Benutzer auf mehr zugreifen kann, als ihm in der Schnittstelle bereitgestellt wird. Etwas wie `#include<stack.c>` ist entsprechend den coding conventions strikt untersagt und wäre eine Verletzung des Geheimnisprinzips.

5. Welche Rolle spielt das Geheimnisprinzip in UML, wie kann dort Schnittstelle und Implementierung unterschieden werden?

In UML gibt es das Geheimnisprinzip sowohl bei abstrakten Objekten (Komponenten, auf die nur über eine Schnittstelle zugegriffen werden kann) als auch bei »richtigen« Objekten, denn deren Klassen besitzen einen `private`-Teil, in dem zu versteckende Information steht. Möglichkeiten zur Unterscheidung von Schnittstelle und Implementierung in UML:

- in Komponentendiagrammen: Schnittstelle durch ihren Stereotypen (einen Kreis) darstellen, Implementierung geschieht in der zugehörigen Komponente.
- in Klassendiagrammen: die Schnittstelle einer Klasse besteht aus ihren als `public` gekennzeichneten Methoden und Attributen. Die Schnittstelle einer Klasse und die Tatsache, dass mehrere Klassen die gleiche Schnittstelle implementieren, kann durch eine Klasse vom Stereotyp `<<interface>>` (mit abstrakten Methoden!) und die `realize`-Assoziation dargestellt werden (vgl. [13, S.1]).

7.3 Blatt 4 Aufgabe 3

»In einem Programmiererteam schreibt einer der Entwickler als Bestandteil eines umfangreichen Systems eine Komponente `Komplex` zur Implementierung komplexer Zahlen. Diese Komponente wird an verschiedenen Stellen benutzt. Von einem Tag zum anderen funktioniert die Gesamtsoftware nicht mehr in allen Situationen richtig. Umfangreiche Debugsitzungen der Teamchefin bringen zu Tage, dass der gerade mit unbekanntem Ziel in Urlaub verreiste Entwickler und Betreuer von `Komplex` zur Effizienzsteigerung die interne Darstellung von kartesischen in Polarkoordinaten umgestellt hat. Das korrekte Funktionieren einer anderen Komponente beruht jedoch auf der kartesischen Darstellung der komplexen Zahlen. Wer hat hier welchen Fehler gemacht und muss darum bei der nächsten Gehaltsrunde auf einen Zuwachs verzichten: Der Implementierer der Komponente `Komplex`, der sie ohne Benachrichtigung ändert, der Implementierer der Komponente, der `Komplex` benutzt, oder die ChefIn?«

- Der Programmierer von `Komplex` hat Schuld, wenn er entgegen dem Softwareentwurf eine Methode nicht als geheim gekennzeichnet hat, die dann der Programmierer der benutzenden Komponente benutzt hat. Diese Kennzeichnung erfolgt durch geschützte Methoden (`private` in C++) oder durch entsprechende Dokumentation in Sprachen, in denen dies nicht möglich ist. Dieser Fehler ist eine Verletzung des Geheimnisprinzips.
- Der Programmierer der benutzenden Komponente ist Schuld, wenn er eine als geheim gekennzeichnete Methode benutzt hat. Solch eine Benutzung ist ja möglich, wenn die Sprache selbst keine Möglichkeit

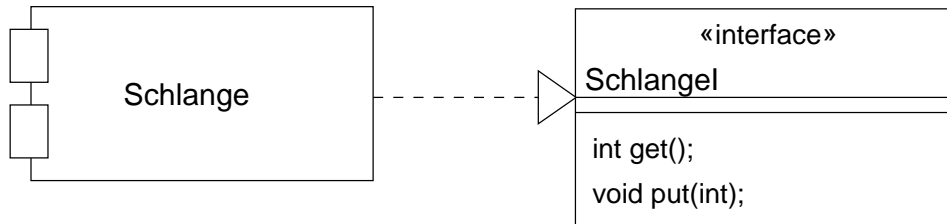


Abbildung 23: Warteschlange als UML-Komponente

bietet, Methoden zu schützen (wie z.B. C). Dieser Fehler ist eine Verletzung des Geheimnisprinzips, denn die benutzenden Komponenten hingen von Interna der Komponente komplex ab, die sie eigentlich nichts angehen dürften.

- Die Chefin ist schuld, wenn die Schnittstelle im Entwurf falsch definiert wurde oder wenn überhaupt keine bindenden Entwurfsdokumente existieren.

7.4 Blatt 4 Aufgabe 4

Geben Sie jeweils ein geeignetes UML-Diagramm und eine Implementierung in C++ (entsprechend den üblichen Konventionen) an für eine Warteschlange (FIFO Prinzip):

1. als Komponente im Sinne von UML Komponenten

Man muss entscheiden, ob man die Warteschlange auf einen bestimmten Typ festlegt oder für alle Typen offenlässt (d.h. ein Template definiert). Da UML-Komponenten aber keine Typen, sondern abstrakte Objekte sind, können sie natürlich auch keine Templates sein.

UML-Diagramm zu dieser Problemstellung siehe Abbildung 23. Die Implementierung in C++ ist im logischen Sinne ein abstraktes Objekt (die .c-Datei), das vom Typ einer abstrakten Basisklasse (definiert durch die .h-Datei) ist:

```
//schlange.h
#ifndef SCHLANGE_H
#define SCHLANGE_H
int get();
void put(int);
#endif

//schlange.c
#include<schlange.h>
#include<list>
list<int> s; // int-Liste; aus dem Listentemplate der STL
int get() {
    int temp=s.front();
    s.pop.front();
    return temp;
};
void put(int x) {
    s.push_back(x);
};
```

2. als Komponente die ein abstraktes Objekt realisiert

Dies entspricht obiger Implementierung als UML-Komponente, denn UML-Komponenten sind abstrakte Objekte. Abstrakte Objekte können in UML nur durch Komponentendiagramme dargestellt werden, ihre Implementierung ist jedoch auf verschiedene Arten möglich (CORBA, namespace, .c- und .h-Datei usw.).

3. als Komponente, die den Zugriff auf ein Objekt einer Klasse Warteschlange ermöglicht
- Das Abstraktum »Warteschlange« kann auf verschiedene Arten implementiert werden, sei es als UML-Diagramm oder C++-Code, wobei der C++-Code eine Implementierung des UML-Diagramms sein kann, aber nicht sein muss. Aufgrund des sehr eingeschränkten Komponentenbegriffs der UML ist die sofortige Implementierung in C++, ohne die Grundlage eines UML-Diagramms (d.h. die Hacker-Methode) hier günstiger. Der Benutzer darf nur auf ein einziges Objekt der Klasse `SCHLANGE` zugreifen können; deshalb gehört die Typdefinition der Klasse `SCHLANGE` nicht zur Schnittstelle `schlange.h`, sonst könnte sich der Benutzer nämlich weitere Objekte der Klasse `SCHLANGE` anlegen.

```
//schlange.h
put(int x);
get(int &x);

//schlange.c
class SCHLANGE {
public:
    void put(int) {
        //...
    };
    int get() {
        //...
    };
private:
    //...
};
SCHLANGE s;
void put(int x) {
    s.put(x);
}
int get() {
    return s.get();
}
```

Diese Implementierung kann nicht durch ein einziges UML-Diagramm hinreichend beschrieben werden. Entweder stellt man die Komponente so dar, wie sie sich nach außen verhält, als abstraktes Objekt (entspricht Abbildung 23) und kann damit die innere Klassenstruktur der Komponente nicht ausdrücken. Oder man zeichnet ein Klassendiagramm und kann dann nicht ausdrücken, dass die Komponente ein abstraktes Objekt ist. Diese beiden Diagrammartentypen dürfen nach offizieller UML-Syntax nicht kombiniert werden¹⁹, denn man befindet sich laut UML-Philosophie auf unterschiedlichen Detail-Ebenen der Diskussion über Software. Beide Diagramme wären eine mögliche und richtige Lösung in der Klausur!

4. als Komponente, die eine Klasse Warteschlange exportiert
- Exportieren: Diese Komponente soll anderen Komponenten die Klassendefinition von Warteschlange zur Verfügung stellen. In einem UML-Klassendiagramm kann man darstellen, dass eine Klasse Warteschlange existiert; weil aber UML-Komponenten abstrakte Objekte sind, kann man in einem UML-Komponentendiagramm nicht darstellen, dass eine Komponente einen Typ als Schnittstelle bereitstellt. Die Implementierung der Komponente in C++ erfolgt entsprechend der üblichen coding conventions (siehe Glossar):

```
//schlange.h - enthaelt Klassendefinition
#ifndef SCHLANGE_H
#define SCHLANGE_H
#include<list>
class schlange {
public:
    int get();
};
```

¹⁹Also z.B. keine Kompositions-Relation zwischen Komponente und Klasse, denn beide Symbole können nicht in einem Diagramm vorkommen!

```

        void put(int);
    private:
        // ...
        list<int> s;
    }
#endif

//schlange.c - enthaelt Methodendefinitionen
#include<schlange.h>
// list<int> s; (hier nicht!)
int schlange::get() {
    // ...
}
void schlange::put(int x) {
    // ...
}

```

Die Liste ist interner Bestandteil dieser Klasse; trotz dass sie also nicht zur Schnittstelle gehört, muss sie in der .h-Datei definiert werden, damit jedes Objekt eine eigene Liste hat statt dass es eine gemeinsame Liste gibt. Weil dieses Element den Benutzer nichts angeht, wird es als `private` gekennzeichnet.

- als Komponente, die eine Methode zur Erzeugung von Warteschlangen exportiert, aber es den Benutzern nicht erlaubt, selbständig (mit Hilfe des Warteschlangen-Typs) Warteschlangen zu erzeugen. Eine Klasse, von der man keine Instanzen erzeugen kann, kann z.B. implementiert werden, indem man alle Konstruktoren unter `private:` stellt. Dieses sog. `factory-pattern` wurde hier nicht behandelt vergleiche aber [3].

7.5 Blatt 4 Aufgabe 5

- Implementieren Sie eine Komponente, die eine Erzeugungsfunktion für Warteschlangen exportiert. Die Warteschlange soll mit Hilfe einer (ebenfalls selbst implementierten) allgemeinen Listenkomponente implementiert werden.

```

//schlange.h
#ifndef SCHLANGE_H
#define SCHLANGE_H
#include<list.h>
class Schlange {
    public:
        void put(int);
        int get();
        Schlange create() {
            return Schlange();
        };
    private:
        Schlange();
        List<int> l;
}
#endif

//schlange.c
#include<schlange.h>
void Schlange::put(int x) {
    l.insert(x);
}
int Schlange::get() {
    int temp=l.getfront();
    l.erase(temp);
    return temp;
}

```

```

}

//list.h
#ifndef LIST_H
#define LIST_H
template <class T>
class List {
public:
    List ();
    ~List ();
    List (const List &);
    List & operator= (const List &);
    void insert (T i);
    void erase (T i);
    void eraseAll (T i);
    int isIn (T i);
    T getfront ();
private:
    class Node {
public:
        Node ();
        Node (T, Node *);
        ~Node ();
        T v;
        Node * next;
    };
    Node *head;
    void erase_r (T i, Node * &);
};
#endif

//list.c
#include<list.h>
//...

```

2. Geben sie eine geeignete Verzeichnis- und Dateistruktur an

```

./include/schlange.h
./include/list.h
./schlange.c
./list.c

```

3. Schreiben sie eine Makedatei, die die Warteschlange als Objektbibliothek erzeugt.

```

//Makefile
schlange.so.1: schlange.o list.o
    g++ -o schlange.so.1 schlange.o list.o

schlange.o: schlange.c schlange.h list.h
    g++ -o schlange.o -I ./include -c schlange.c

list.o: list.c list.h
    g++ -o list.o -I ./include -c list.c

```

4. Welche Include-Datei wird mit der Bibliothek ausgeliefert?
 Nur `Schlange.h`, denn um gegen die Objektbibliothek zu binden, muss man nicht die Schnittstelle der Komponente `list` kennen, die von der Komponente `schlange` benutzt wird.
5. Geben sie ein UML-Diagramm an, das die Klassenstruktur ihrer Implementierung darlegt. (Siehe Abbildung).

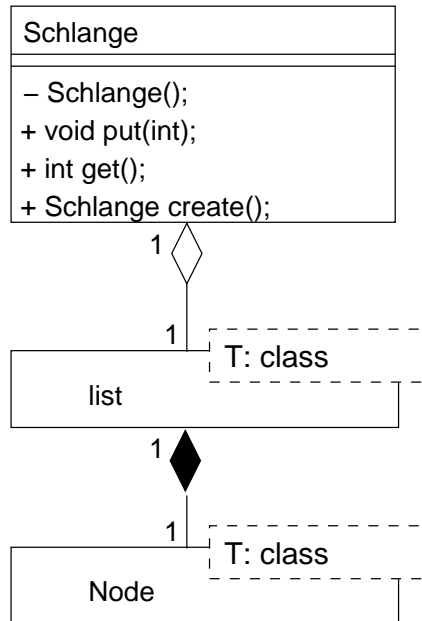


Abbildung 24: Klassenstruktur zu Übungsblatt 4, Aufgabe 5

6. Geben Sie geeignetes Sequenzdiagramm zu Ihrer Implementierung an und erläutern Sie dessen Inhalt und Aussage.

7.6 Blatt 4 Aufgabe 6

Die Aufgabe, anderer Leute SourceCode zu lesen, ist eine wichtige Aufgabe eines Softwareentwicklers. Hier soll UML zur Illustrierung der gewonnenen Erkenntnisse beim Lesen des Sourcecodes eingesetzt werden. Außerdem soll versucht werden, Muster im Programm zu erkennen, d.h. wie Dinge nach üblicher Art implementiert wurden.

1. Hier werden die Klassen Knoten, Wertknoten, OpKnoten im privaten Teil der Klasse Ausdruck definiert. Diese geschachtelte Definition ist in UML nicht ausdrückbar, hier müssen die Klassen nebeneinander gezeichnet werden.
Im Normalfall werden Zeiger auf eine Klasse als nicht ausgefüllte Raute dargestellt, d.h. die Klasse, auf die ein Zeiger zeigt, kann auch ohne die benutzende Klasse existieren. Dies ist jedoch nicht ohne Ausnahme, z.B. wenn die benutzte Klasse im private-Teil der benutzenden Klasse definiert ist, wie in dieser Aufgabe. Man muss also stets noch untersuchen, ob die über Zeiger benutzte Klasse auch sinnvollerweise alleine existieren könnte.
2. UML-Diagramm siehe Abbildung 26.
3. Sequenzdiagramm zu »Ausdruckswert berechnen«: Es erfolgt ein Zugriff auf ein Objekt der Klasse Ausdruck, dann auf deren Knoten und rekursiv absteigend erfolgt hier die Berechnung. In Sequenzdiagrammen kann die Rekursion nicht allgemein angegeben werden, sondern nur für einen speziellen Fall an gegebenen Objekten vom Typ Knoten und ihres Inhalts.
4. Es gibt zwei Familien (Konten, Werte), die beide nach demselben Schema definiert sind, nämlich: es gibt jeweils eine Umschlagklasse (Ausdruck, Wert), als deren Bestandteil jeweils eine abstrakte Basisklasse (Knoten, Zahl) und darunter die realen Varianten (Wertknoten, Opknoten bzw. NZahl, Rational). Die Umschlagklassen enthalten jeweils die Lese- und Schreiboperationen; da sie als freie friend-Funktionen realisiert sind (operator>>, operator<<) können sie im UML-Diagramm nicht dargestellt werden. In der abstrakten Basisklasse ist jeweils eine clone-Funktion zum Kopieren implementiert. Die Umschlagklasse enthält einen Zeiger (Knoten * ak bzw. Zahl * z), der auf ein Objekt zeigt, das von der abstrakten Basisklasse abgeleitet wurde. Wie kann man nun eine Kopie der Umschlagklasse und des Objektes, auf das sie zeigt, erzeugen, trotz dass man nicht auf die Konstruktoren der von der abstrakten

Abbildung 25:

Abbildung 26:

Basisklasse abgeleiteten Objekte zugreifen kann, weil man den Namen ihrer Konstruktoren nicht kennt, da man ja nicht den Typ der Ableitung kennt? Man verwende eine Funktion clone, die in allen abgeleiteten Klassen gleich heißt und entsprechend einem Konstruktor wirkt.
Die Programmierung nach solchen Mustern ist wichtig! Siehe [Abbildung 25](#).

7.7 Komponentenaufteilung: System zur Auswertung geklammerter Ausdrücke

Es sollen Ausdrücke der Form $((4+2)*4)$ ausgewertet werden. Dies kann geschehen durch einen Wert- und einen Operandenstapel; am Ende liegt auf dem Wertstapel das Ergebnis. Wie könnte die Komponentenaufteilung zu diesem System aussehen? Siehe [Abbildung 27](#). Algorithmus zur Auswertung in Pseudocode

```
while(nochZeichenDa){
    z=nächstesZeichen;
    case z
        '(' -> break;
        ziffer -> push(Wertstapel);
        op -> push(Opstapel);
        ')' -> w1=top(Wertstapel), pop(Wertstapel);
            w2=top(Wertstapel), pop(Wertstapel);
            op=top(opStapel), pop(Opstapel);
            push(Wertstapel op(w1,w2));
    }
```

Die Schnittstelle zur äußersten Komponente main sollte sinnvollerweise einfach sein, um vom Kunden benutzt werden zu können. Es bieten sich strings an, die den zu berechnenden Ausdruck bzw. das Ergebnis enthalten. Also wird main.c zum Beispiel:

```
#include <komp1.h>
int main()
    string s;
    int w;
    // berechne() ist eine Funktion der Komponente komp1
    w=berechne(s);
}
```

Die Schnittstelle wird dann definiert in der Datei komp1.h:

```
#ifndef KOMP_1_H
#define KOMP_1_H
#include<strings>
int berechne(std::string);
#endif
```

Zu dieser Schnittstelle gehört nun ein entsprechendes UML-Diagramm. Siehe [28](#) Der Komponente komp1 können nun entsprechend dem Algorithmus die Hilfskomponenten OpStapel und WertStapel zur Verfügung gestellt werden. Verantwortlichkeiten der Komponenten: komp1 implementiert den obigen Algorithmus, d.h. sie durchläuft den String und erkennt die Klammerstruktur. Die entsprechende C-Datei ist dann komp1.c:

Abbildung 27:

Abbildung 28:

```
#include<komp1.h>
#include<opstapel.h>
#include<wertstapel.h>
//es folgt der Algorithmus
int berechne(std::string s){
//...
// berechne ruft die Komponenten opstapel und wertstapel über
// das Interface topOp() usw., d.h. über Funktionsaufrufe auf
}
```

Gesamtbild der Komponentenaufteilung siehe [29](#).

Die Datei opstapel.h:

```
#ifndef ...
// das Interface ist keine Klasse, sondern ein Objekt:
// es besteht nur aus freien Funktionen
char topOp();
void popOp();
void pushOp(char);
#endif
```

Die Datei opstapel.c:

```
#include<opstapel.h>
// Implementierung mit Hilfe der Standardbibliothek:
// (man programmiert stets so wenig wie möglich, d.i. man
// verwendet möglichst nur Bibliotheken)
#include<stack>
stack<char> stOp;

char topOp(){
    return stOp.top;
}
// und die anderen Funktionen.
```

Die Datei wertstapel.h:

```
#ifndef ...
int topWst();
void popWst();
void pushWst(int);
#endif
```

Die Datei wertstapel.c:

```
#include<wertstapel.h>
#include<stack>
stack<int> stWst;

void pushWst(int x){
    stWst.push(x);
// und die anderen Funktionen
}
```

Hier wurden Komponenten im Sinne von UML erzeugt, d.h. abstrakte Objekte.

Literatur

- [1] Sommerville. Sein Buch zu Softwaretechnik ist für den späteren realen Einsatz empfehlenswert. Es ist im Bestand der Bibliothek der FH Gießen-Friedberg.
- [2] »UML@work«. Ein deutsches Buch zur UML, der akzeptierten graphischen Darstellung in der Informatik. Es ist im Bestand der Bibliothek der FH Gießen-Friedberg vorhanden.
- [3] Bernd Oestereich: »Objektorientierte Softwareentwicklung«, 5. Auflage. Das deutsche Standardwerk der objektorientierten Softwareentwicklung. Daher als Literatur zur Vorlesung »Softwaretechnik 1« empfohlen. Es ist im Bestand der Bibliothek der FH Gießen-Friedberg vorhanden. Weitere Informationen zu diesem Werk und eine Leseprobe von 116 Seiten finden sich auf der Homepage des Verlages <http://www.oose.de>.
- [4] Thomas Letschert: »SWT-I; Einführung: Software als Industrieprodukt«. Erstes Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-1.pdf>.
- [5] Thomas Letschert: »Analyse: Planung, Anforderungsanalyse«. Zweites Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-2.pdf>.
- [6] Thomas Letschert: »Analyse: Konzeptionelles Modell«. Drittes Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-3.pdf>.
- [7] Thomas Letschert: »Entwurf: Grobentwurf«. Viertes Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-4.pdf>.
- [8] Thomas Letschert: »Entwurf: Komponenten, Definition und Implementierung«. Fünftes Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-5.pdf>.
- [9] Thomas Letschert: »Strukturiertes oder Systematisches Programmieren«. Sechstes Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-6.pdf>.
- [10] Thomas Letschert: »Test und SW-Management«. Siebtes Skriptstück zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/swt-7.pdf>.
- [11] Skript zur Vorlesung »Softwaretechnik 1« bei Prof. Franzen, dem »ersten Softwaretechniker der FH Gießen-Friedberg«. Erhältlich unter <http://hera.mni.fh-giessen.de/~hg7132/swt/swt-1.html>.
- [12] Skript zur Vorlesung »Softwaretechnik 1« bei Prof. Kaufmann, dem »ersten Systemanalytiker der FH Gießen-Friedberg«. Erhältlich unter <http://wi.mni.fh-giessen.de/veranstaltungen/kaufmann/systemtechnik.htm>.
- [13] Notationsübersicht und Glossar zur UML. Lizenz public domain. Dieses Dokument ist ein Teil des Werkes [3]. Download auf <http://www.oose.de/uml>.
- [14] »Glossar über alle Themengebiete«. Enthält u.a. die Einträge des Glossars in [13], außerdem ein Glossar zur objektorientierten Programmierung und weitere Glossare. Quelle <http://www.oose.de/glossar>.
- [15] Thomas Letschert: »Übung 1 - SWT I«. Erstes Übungsblatt zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/uebung-1.pdf>.

- [16] Thomas Letschert: »Übung 2 - SWT I«. Zweites Übungsblatt zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/uebung-2.pdf>.
- [17] Thomas Letschert: »Übung 3 - SWT I«. Drittes Übungsblatt zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/uebung-3.pdf>.
- [18] Thomas Letschert: »Übung 4 - SWT I«. Viertes Übungsblatt zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/uebung-4.pdf>.
- [19] Thomas Letschert: »Übung 5 - SWT I«. Viertes Übungsblatt zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/uebung-5.pdf>.
- [20] Thomas Letschert: »Beispiel: Aufgabenstellung«. Zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/Aufgabenstellung.pdf>.
- [21] Thomas Letschert: »Beispiel: Aktoren und Anwendungsfälle, Obefläche und Datendiktionär«. Zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/Anwendungsfaelle.pdf>.
- [22] Thomas Letschert: »Beispiel: Konzeptionelles Modell als Klassendiagramm«. Zur Vorlesung »Softwaretechnik 1« im Wintersemester 2001/2002 an der FH Gießen-Friedberg, Studienort Gießen. Erhältlich unter Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/SWT-1/konzept.pdf>.
- [23] Thomas Letschert: »Hausübungen zu Programmierung II (C++) Wintersemester 2001/2002«, 2001-09-25. Ursprüngliche Adresse auf der Homepage von Professor Letschert <http://homepages.fh-giessen.de/~hg51/> ist nicht mehr aktuell, daher zur Verfügung gestellt unter <http://matthias.ansorgs.de/InformatikDiplom/Modul.SoftwareTk1.Letschert/Prog2.HausUebg.pdf>.