TeachSWT 2002 – Anleitung für den Assistenten und Erfahrungsbericht

M E Leypold

19. Dezember 2002

Inhaltsverzeichnis

1 Was ist TeachSWT?

TeachSWT ist eine Sammlung von Handouts, Fallstudien und Übungen zu Themen der Softwaretechnik. Sie entstand (in Fragmenten) im Verlauf des Sommersemesters während der Übungen zur Vorlesung und wurde in den folgenden Monaten erweitert und in eine zum Release geeignete Form gebracht. TeachSWT ist in Quellform unter der *Gnu Free Documentation License* für jedermann verfügbar, so dass es entsprechend den jeweiligen Anforderungen auch möglich ist, nur Teile zu übernehmen, oder die Texte zu erweitern oder abzukürzen.

TeachSWT ist aus dem Problem heraus entstanden, dass uns für die Softwaretechnikvorlesung keine Fallbeispiele zur Verfügung standen, die zum einen – hoffentlich überzeugend – Wert und Anwendung verschiedener Methoden illustrieren, zum anderen aber klein genug sind, um in der Vorlesung oder als Übungsaufgaben behandelt zu werden. TeachSWT schließt – zumindest für meine Bedürfnisse – diese Lücke. Dabei ist die Reihenfolge und Darstellung des Materials in einigen Punkten leider nicht ganz optimal – ich werde auf die Gründe noch zu sprechen kommen – aber als Steinbruch für Material sollte es allemal taugen.

2 Für wen ist diese Anleitung?

Dieses Handbuch ist zum einen ein Überblick über das vorhandene Material, ein Führer über die Zusammenhänge zwischen den einzelnen Themen, zum anderen ein Erfahrungsbericht, wo es zu „Missverständnissen“ mit den Teilnehmern über die Aufgabenstellung kam, unsere Ziele nicht erreicht wurden , und wo wir jetzt Verbesserungsbedarf erkennen.

In diesem Sinne ist dieses Handbuch zuerst einmal eine Handreichung für den Assistenten oder Mitarbeiter, der die Übung betreut, zum anderen aber auch für den Dozenten, damit die Vorlesung so gestaltet werden kann, dass sie die Themen für die Übungen abwirft. Dazu können die Lösungsskizzen – soweit die Aufgaben nicht oder nicht genau so als Übungen gestellt werden – auch Anschauungsmaterial für und in der Vorlesung liefern.

3 Überblick über das Material

3.1 Themenfolge

Schwerpunkt: Technische Probleme der Programmierung. Der Themenbereich der Übungsblätter ist beschränkte auf die technischen Aspekte der Programmierung. Wirtschaftliche (Kostenschätzung und -effizienz) sowie soziale Aspekte (Teamarbeit, Arbeitsverteilung und -organisation, „Soft Skills“ (whatever . . .)) blieben ausgeschlossen, da sie sich in einem Hausaufgabensystem schlecht behandeln lassen.

Zu den technischen[1] Aspekten der Programmierung gehört die systematische[2] Konstruktion von Programmen. Hier müssen u. a. die folgenden Fragen beantwortet werden:

- Was sind Systemteile?
- Wie beschreibe ich Systeme und Systemteile?
- Wie setze ich Teile zu einem System zusammen und wie zerlege ich ein System in Teile?
- Von welchen Kriterien lasse ich mich dabei leiten?

Aus diesen Fragen ergeben sich weitere, wie etwa die nach der Verbindung zwischen Spezifikation und Implementation. Summa summarum könnte man die Übungen zur Softwaretechnik einen *Programmierkurs für Nicht-Ganz-Anfänger* nennen.

In der Praxis des Sommersemesters 2002 haben sich dann übrigens bei den Teilnehmern schwere „handwerkliche“ Defizite in der Beherrschung von Programmiersprachen herausgestellt – selbst bei dem Kreis von Teilnehmern, die dem Vernehmen nach C++ bereits erlernt hatten – so dass wir in den meisten Aufgabenblättern dann auf der beschreibenden Seite blieben und es vermieden[3], Implementationsaufgaben zustellen.

Runden 1 bis 4: Systembegriff und modellbasierte Spezifikation. Die Runden 1 bis 4 der Übungen beschäftigen sich mit der Beschreibung und Implementation von Systemen und Systemteilen. Zentral ist dabei der Gedanke, dass Systemteile (möglicherweise) Zustand enthalten und es genügt, Einflüsse und Reaktionen im Auge zu behalten, die die Systemgrenze überqueren[4], mit einem Wort: Ein- und Ausgaben.

Um ein System oder ein Subsystem (einen Systemteil) zu beschreiben, genügt es, seine „Antwortfunktion“ zu nennen – d. h. wie sich der Zustand als Resultat einer Eingabe ändert, und welche Ausgabe dabei produziert wird. Dies ist im Wesentlichen das Thema von Runde 1, ausgeführt an einfachen Beispielen: Einem assoziativen Array und einem FIFO. Traditionell zerfallen die Verfahren, mit denen die Reaktion eines Systems beschrieben werden kann, in 2 Klassen: Die *algebraischen* Spezifikationsverfahren (z. B. Larch),

[1] Ich verwende das Wort *technisch* hier in der ursprünglichen des griechischen Wortes τεχνη, das für *Kunst*, insbesondere *handwerkliche Kunst* steht. Es geht also um Kenntnis des „Materials“ aus dem Programme zusammengesetzt sind, und um die Fähigkeit, dieses Material gezielt zum Endprodukt zu verarbeiten. τεχνη ist die *Fachkenntnis.*

Die handwerkliche Assoziation ist gewollt und schließt Kreativität nicht aus. Das Beispiel des Schreiners ist hier ganz aufschlussreich: Planung und Definition des Endproduktes ist die Gelegenheit, bei der Kreativität in hohem Maße gestaltend wirkt. Bei der Anfertigung des Produktes gibt es dagegen Regeln guter Praxis, an die sich ein guter Tischler auch halten wird. Darin, wie diese Regeln umgesetzt werden, scheiden sich aber auch gute von schlechten Schreinern – dies ist zwar auch, aber nicht nur eine Frage vorhandener manueller Geschicklichkeit. Es ist vielmehr die Fähigkeit, mit unvermutet auftretenden Problemen oder individuellen Eigenschaften des Materials (z. B. Maserung oder Astlöchern) umgehen zu können und den Freiraum der Regeln so zu nutzen, dass das Endprodukt entsprechend profitiert. Eben das erfordert wieder Kreativität – im Kleinen.

[2] *Systematisch* bedeutet, dass es ein System gibt, nach dem vorgegangen wird, d. h., dass – eventuell auch nur lose definierte – Regeln existieren, denen die Einzelschritte der Entwicklung unterworfen werden, und die dazu dienen können, Entscheidungen, im nötigen Fall auch *post factum*, zu rechtfertigen.

[3] Dies ist ein gutes Beispiel dafür, wie die Softwaretechnik auf bereits vorhandene Kenntnisse aus anderen Fachbereichen der Informatik und manchmal sogar über die Informatik hinaus angewiesen ist. Alles in allem kann man sich dem Programmieren im Größeren und im Großen erst widmen, wenn das Programmieren im Kleinen kein Problem mehr darstellt, und wenn auch gewisse Kenntnisse von Fachbereichen vorhanden sind, die Gegenstand von Projekten und Fallstudien sein können.

Die syntaktische und konzeptionelle Beherrschung einer Programmiersprache muss vorausgesetzt werden. ideal und in hohem Maße wünschenswert wäre es, wenn die Teilnehmer bereits mehrere Sprachen beherrschen, so dass sprachunabhängiges Denken bereits Einzug gehalten hat.

[4] Dies ist die Essenz des Systemgedankens, einen Teil der Welt abzugrenzen, so dass man sich auf die Systemgrenzen konzentrieren und über den Systeminhalt abstrahieren kann.

charakterisieren nur das algebraische Verhältnis der Operationen untereinander[5], während in den *modellorientierten* Verfahren (wie VDM-SL oder Z) explizit die Wertemengen des Zustandes und der Ein- und Ausgabe konstruiert werden. Runde 1 führt VDM-SL als Spezifikationssprache ein. (Im Folgenden wird diese Wertemenge zuweilen als *Datenmodell* – in Unterscheidung von der *Spezifikation der Operationen* bezeichnet.)

Modellbasierte Spezifikationssprachen treffen ihre Aussagen in Form einer Analogie: Das beschriebene System verhält sich *so wie* das Modell, wenn bestimmte Entsprechungen angenommen werden zwischen den Zuständen des Modells und denen des beschriebenen Systems. Die Entsprechung wird hergestellt durch eine (bezüglich der Operationen homomorphe) Abbildung zwischen Modellen und beschriebenem System – der *Retrieve*-Funktion. Diese muss bestimmte Eigenschaften haben. Die Entsprechung zwischen Modell und Implementation, die Beschreibung von Modulen (Implementationseinheiten) durch Schnittstellen, sowie modulare Programmierung in C/C++ sind die Themen von Runde 2.

Runde 3 spricht schließlich die Behandlung von Speicher (allgemeiner Referenzen auf zustandsbehaftete Ressourcen) und von „Seiteneffekten“ in den modellbasierten Spezifikationsverfahren an[6]. Dazu gehören auch Ausführungen über den Zusammenhang zwischen der Signatur von Operationen, und der Schnittstelle von Prozeduren, welche die Operationen implementieren. Es wird dargelegt, dass bestehenden Freiheiten vor allem in der Festlegung der Speicherstrategie (Pass-by-Value versus Pass-by-Reference) liegen, und wie objektorientierte Methodenaufrufe dazu in Bezug zu setzen sind.

Für die Aufgaben der Runden 1 und 2 wurde bewusst ein sehr einfaches Beispiel ausgewählt, damit sich die Teilnehmer auf das Erlernen von VDM-SL und grundlegender Elemente der modellbasierten Spezifikation konzentrieren können. Runde 3 führt das erste nichttriviale Beispiel modellbasierter Spezifikation ein: einen Heap. Dieser kann stellvertretend für eine ganze Anzahl ähnlicher Strukturen gesehen werden (Datenbank, Dateisystem, assoziatives Array …), denen allen gemeinsam ist, dass Daten darin abgelegt werden, und Schlüssel (Referenzen!) zur Wiedererlangung der gespeicherten Daten verwendet werden müssen.

Runde 4 schließlich beschließt den Reigen der VDM-SL-Themen mit einer Aufgabe (Nachbarschaft von Theatersitzen und die Allokation zusammenhängender Sitzgruppen) die praktisch nur lösbar ist, wenn man jeden Versuch, an die Darstellung für Implementationszwecke (Listen, Array, was auch immer) zu denken, vollständig aufgibt und über die Topologie der Aufstellung der Theatersitze abstrahiert.

Die Runden 1 bis 4 haben im Wesentlichen modellbasiertes Spezifizieren zum Thema (Das ist hier die Antwort auf: Wie spezifiziert man Systemteile?). Thematisch darin eingebettet sind jeweils Exkurse zum modularen Programmieren (Runde 2), besonders sichtbar in Form der Auslassungen über Namensräume von Modulen (TeachSWT@Tü, 2002u) und einer Aufgabe zu Make – dem bekannten Werkzeug zur modulweisen inkrementellen Übersetzung (Runde 4). Mit Runde 4 sind die VDM-SL-Aufgaben jedoch abgeschlossen.

Runde 5: Feinentwurf. Die Runden 1 bis 4 haben sich mit der Beschreibung *einzelner* Systemteile beschäftigt. Das Thema von Runde 5 ist der Fein-Entwurf, das heisst, zum einen die Zusammensetzung eines Systems aus vielen Teilen, zum anderen reden wir hier bereits von Teilen der Implementation (Module). Die Methode, durch die sich die Lösungsskizze auszeichnet, ist die Komposition des Systems aus abstrakten Datentypen. Als Beispiel dient ein Disassembler für einen 8-Bit-Mikroprozessor. Ein System dieser Größenordnung vollständig mit formalen Methoden zu beschreiben, ist zum einen ein großer Aufwand, zum anderen wahrscheinlich auch ein Overkill. Stattdessen wird in Runde 5 die Spezifikation von Modulen in natürlicher Sprache propagiert – wobei man

[5] Daraus können dann auch geeignete Trägermengen konstruiert werden.

[6] Nicht alles ist eine Funktion, aber alles kann durch eine Funktion beschrieben werden.

sich, um vollständig zu sein und eine gewisse Präzision zu erhalten am „Geist der formalen Spezifikation“ orientieren sollte, d. h. *Eingabe*, *Zustände* und *Ausgabe* im Auge behalten und Operationen um (abstrakte) Datentypen gruppieren sollte. Hier wird auch das Thema von Ressource-Referenzen wieder aufgenommen in Form der Frage nach den Verantwortlichkeiten für Speicher, auf den Zeiger zeigen, die Modulgrenzen überqueren. Da in dieser Runde Module als Systemteile in Erscheinung treten, werden hier nochmals die Fragen der modularen Programmierung in C/C++ berührt (u. a. in der Lösungsskizze die Kapselung der Implementation in einer idealerweise *opaken* Typdefinition).

Runden 6 und 7: Objektorientierung. Die Runden 6 und 7 haben objektorientierte Analyse und Entwurf zum Gegenstand. Diese sind in vielerlei Hinsicht anders als die klassischen strukturierten (modularen) Methoden[7]: Während in den letzteren das System hierarchisch zusammengesetzt (Bottom-Up) oder zergliedert (Top-Down) wird, zeichnen sich die objektorientierten Methoden durch Hierarchielosigkeit aus. Stattdessen werden immer Teilansichten und Ausschnitte beschrieben, die dann additiv zusammengenommen das Gesamtsystem beschreiben. Dementsprechend ist die zeitliche Trennung zwischen dem Entwurf der Teile und ihrer Zusammensetzung in der objektorientierten Welt auch nicht haltbar – vielmehr wird zu fast jedem Zeitpunkt der Systemteil (Objekt) in einem Kontext gezeigt, in den er eingebettet ist (Assoziationen, siehe Klassendiagramme). Stattdessen ist es in der Objektorientierung üblich, zwischen der Modellierung statische und dynamischer Aspekte zu trennen[8]. Erstere ist Gegenstand von Runde 6, letztere wird in Runde 7 behandelt.

Runde 8: Strukturierte Analyse. Runde 8 versucht anhand eines nichttrivialen Beispiels (des Disassemblers aus Runde 5) den Teilnehmern mit der strukturierten Analyse ein Werkzeug zur Klärung von Datenzusammenhängen an die Hand zu geben.

Das Aufgabenblatt zu Runde 8 stellt in Wahrheit drei Aufgaben zur Auswahl. Diese Wahlfreiheit war als Bonus für fortgeschrittene Teilnehmer gedacht. Zwei davon sind auf Design-Patterns hin orientiert. In der Lösungsskizze ist nur die dritte Aufgabe – die zur strukturierten Analyse – ausgeführt.

3.2 Verzeichnis aller Dokumente

Ich gebe hier in logischer Reihenfolge – d. h. geordnet nach Runden, deren jede einem Aufgabenblatt mit dazugehörigen Handouts und Lösungsskizzen entspricht – eine vollständige Liste aller Dokumente und Quelltexte an, die im Sommer 2002 für die Vorlesung *Softwaretechnik* ausgegeben oder für den internen Gebrauch angefertigt wurden. All diese Dokumente und Quelltexte sind auch in TeachSWT enthalten (mit Ausnahme des Accountantrags). Die meisten Lösungsskizzen wurden allerdings erst nach Ende des Semester fertiggestellt und zugänglich gemacht[9].

Literatur zu den Themen der einzelnen Übungsrunden ist jeweils in den Aufgabenstellungen und in den Lösungsskizzen zitiert.

[7] Auch wenn natürlich zentrale Paradigmen – Abstraktion, Geheimnisprinzip, Design-by-Contract – auch hier eine Schlüsselstellung einnehmen.

[8] In der modellbasierten Spezifikation – präziser: der funktionalen Spezifikation – würde die statische Modellierung (das Datenmodell alleine) wenig Information geben. In der objektorientierten Welt zählt auch die Beschreibung von Methoden zur statischen Modellierung, so dass Klassendiagramme bereits recht viel Information über das beschriebene System bereitstellen.

[9] Das Prinzip der Lösung und Skizzen von Teillösungen wurden aber häufig in den Tutorien besprochen. Vielen Dank hier auch an die Tutoriumsteilnehmer, deren Feedback mir beim Erstellen der Lösungsskizzen und beim Verständnis der didaktischen Probleme sehr geholfen hat.

Konzeptpapiere und Übersichten:

- *Softwaretechnik 2002 – Konzept zur Durchführung der Übungen* (Leypold, 2002)
- *Softwaretechnik 2002 – Anleitung für den Assistenten und Erfahrungsbericht* (TeachSWT@Tü, 2002x)

Zum Semesterbeginn: Zu Beginn des Semesters wurde an alle Teilnehmer eine Beschreibung des geplanten Ablaufs der Übungen ausgegeben. Dies erspart viele Wiederholungen, da nicht wenige Teilnehmer erst in der zweiten und dritten Woche die Lehrveranstaltung zum ersten Male besuchen. Darüberhinaus ist es sinnvoll, den Teilnehmern etwas an die Hand zu geben, worin sie Regeln, Kriterien und Termine nachlesen können. Dies beugt Missverständnissen vor, und erspart viele Sonderregelungen, die dann leider doch nötig werden.

- *Ablauf der Übungen und Scheinkriterien* (TeachSWT@Tü, 2002a)
- *Accountantrag*

Runde 1: Systembeschreibung durch Modellierung von Eingabe, Zustand, Ausgabe; Einführung der funktionalen Spezifikationsverfahren; Einführung in VDM-SL.

- *VDMTools am WSI* (TeachSWT@Tü, 2002y)
- *Spezifikation eines assoziativen Arrays in VDM-SL* (TeachSWT@Tü, 2002w)
- *VDM-SL-Beispiele aus dem Editor EFASS* (efass, 2002c)
- *Übungsblatt 1: Spezifikation eines FiFo in VDM-SL* (TeachSWT@Tü, 2002c)
- *Lösungsskizze 1: Spezifikation eines FIFO* (TeachSWT@Tü, 2002m)

Runde 2: Module und Modulschnittstellen (Fein-Entwurf), modulares Programmieren in C++; Module und Namensräume; Implementation; Entsprechung zwischen Implementation und Spezifikation (Retrieve-Funktionen); Implementationsspezifische Beschränkungen.

- *Module und Namensräume* (TeachSWT@Tü, 2002u)
- *Von der Spezifikation zur Implementation* (TeachSWT@Tü, 2002z)
- *Übungsblatt 2: Implementation eines FiFo* (TeachSWT@Tü, 2002d)
- *Quelltext für eine Prioritätswarteschlange in Modula-2* (SRC-prioqueue, 2002)
- *Lösungsskizze 2: Implementation eines FiFo* (TeachSWT@Tü, 2002k)
- *Quelltext für einen FIFO in C++* (SRC-fifo, 2002)

Runde 3: Speicher, Referenzen und Seiteneffekte in der funktionalen Spezifikation; Freiheiten beim Umsetzen von Schnittstellen (Pass-by-Reference versus Pass-by-Value); das erste komplexere VDM-SL-Beispiel.

- *Spezifikation durch Funktionen und die Behandlung von Speicher* (TeachSWT@Tü, 2002v)
- *Übungsblatt 3: Modellierung eines Heap in VDM-SL* (TeachSWT@Tü, 2002e)
- *Lösungsskizze 3: Spezifikation eines Heap in VDM-SL* (TeachSWT@Tü, 2002o)

Runde 4: Ein schwieriges VDM-SL-Beispiel, für das Abstraktion unverzichtbar ist; Inkrementelles Übersetzen mit Make.

- *Übungsblatt 4: Theatersitze, Make* (TeachSWT@Tü, 2002f)
- *Quelltext für einen FIFO in C++* (SRC-fifo, 2002), ohne Makefiles, Material!
- *Lösungsskizze 4: Saalbestuhlung, Makefile* (TeachSWT@Tü, 2002p)

Runde 5: Natürlichsprachige Beschreibungen von Schnittstellen; Feinentwurf (in C/C++); Systemkomposition aus abstrakten Datentypen; Opake Kapselung von Implementation in C/C++; Verantwortlichkeiten für Speicher.

- *Das EVA-Paradigma* (TeachSWT@Tü, 2002l)
- *Beschreibung von Modulen in natürliche Sprache* (TeachSWT@Tü, 2002b)
- *Übungsblatt 5: Entwurf eines Disassemblers* (TeachSWT@Tü, 2002g)
- *Modulschnittstellen (Headerdateien) für einen 8-Bit-Disassembler in C++* (SRC-disas65, 2002)
- *Lösungsskizze 5: Entwurf eines Disassembler* (TeachSWT@Tü, 2002q)

Runde 6: Statische objektorientierte Modellierung; Klassendiagramme.

- *Übungsblatt 6: Statische OO-Modelle* (TeachSWT@Tü, 2002h)
- *Lösungsskizze 6: Statische OO-Modelle mit UML* (TeachSWT@Tü, 2002r)

Runde 7: Dynamische Objektorientierung; Ablaufdiagramme; Zustandsdiagramme.

- *Übungsblatt 7: Dynamische Modellierung mit UML* (TeachSWT@Tü, 2002i)
- *Lösungsskizze 7: Dynamische Modellierung mit UML* (TeachSWT@Tü, 2002s)

Runde 8: Strukturierte Analyse.

- *Übungsblatt 8: Strukturierte Analyse, MVC-Pattern* (TeachSWT@Tü, 2002j)
- *Lösungsskizze 8a: Strukturierte Analyse* (TeachSWT@Tü, 2002t)

Weiteres Material: EFASS ist ein kleiner, sehr einfacher Editor, der als Fallstudie für Spezifikation und modulare Implementation in C/C++ entwickelt wurde. Teile aus der EFASS-Spezifikation wurden in der Vorlesung gezeigt, und als Handout ausgegeben. Als Referenz stand der EFASS-Quelltext während des Semesters zum Download zur Verfügung.

- *EFASS (An Editor for A Small System) – Fallstudie zu modularem Entwurf und modularer Programmierung für die Softwaretechnikvorlesung 2002 in Tübingen* (efass, 2002b)
- *EFASS (An Editor for A Small System) – Beschreibung und VDM-SL-Spezifikation* (efass, 2002a)

3.3 Fallstudien und Quelltexte

TeachSWT enthält – eingebettet in Übungen, Handouts und Lösungskizzen eine Reihe von Fallstudien. Eine Fallstudie ist jeweils ein Problem der Anwendung bzw. ein Softwareprodukt, auf das einzelne Methoden der Softwaretechnik angewandt werden. Um die Suche nach Fallbeispielen zu erleichtern, folgt nun eine Übersicht über die Fallstudien (Thema, angewandte Methoden, zugehörige Handouts, Lösungen und Aufgabenstellungen, Grad der Ausarbeitung):

- In Runde 1 und 2 wird ein **FIFO** spezifiziert (TeachSWT@Tü, 2002c,m) und in modularer Weise implementiert (TeachSWT@Tü, 2002d,n; SRC-fifo, 2002). Der Zusammenhang zwischen Spezifikation und Implementation wird mittels der Abstraktionsfunktion hergestellt. Aus der Abstraktionsfunktion werden Beschränkungen der Implementation abgeleitet und die Spezifikation entsprechend korrigiert und eine Behandlung für Sonderfälle hinzugefügt (TeachSWT@Tü, 2002n).

  Der FIFO ist ein einfaches Beispiel für Spezifikation in VDM-SL un modulare Implementation in C/C++. Zudem ist ein FIFO übersichtlich genug dafür, dass das in *Von der Spezifikation zur Implementation* (TeachSWT@Tü, 2002z) skizzierte Verfahren um Spezifikation und Implementation miteinander zu verbinden und dadurch die Implementation zu prüfen, vollständig durchgeführt werden kann.

- **DisAs65** ist ein einfacher Disassembler für einen 8-Bit-Mikroprozessor. Für DisAs65 wurde eine strukturierte Analyse (TeachSWT@Tü, 2002j,t) angefertigt, sowie ein modularer Fein-Entwurf (TeachSWT@Tü, 2002g,q; SRC-fifo, 2002), d. h. das Programm wurde in Module gegliedert, die Signatur der Module wurde mit sprachspezifischen Mitteln (hier: Headerdateien) und ihre datenverarbeitende Leistung in natürlicher Sprache beschrieben.

  DisAs65 wurde nicht implementiert, lediglich die Headerdateien (SRC-disas65, 2002) wurden durch den Compiler geprüft.

- **EFASS** ist ein einfacher, vollständig ausimplementierter Editor in C/C++ (efass, 2002b) von dem Teile in VDM-SL spezifiziert wurden (efass, 2002c,a) EFASS ist ein anspruchsvolleres Beispiel für die Spezifikation einer realen Implementation in VDM-SL, für modulare Implementation in C/C++ und die Gliederung von Applikationen mittels abstrakter Datentypen. EFASS stellt dabei sowohl Beispiele für abstrakte Datentypen (eigentlich: *opake Datentypen*, d. h. Typen, von denen die Sichtbarkeit der Struktur auf ein Modul beschränkt – also gekapselt – ist), als auch Beispiele für abstrakte Datenobjekte (nach Balzert (1996), eigentlich: *opaker Zustand*, also Zustand, der in ein Modul gekapselt ist) zu Verfügung.

  Weiterhin illustriert die EFASS-Spezifikation, wie die Systemumgebung, in der ein Produkt später ausgeführt werden soll, beschrieben werden kann.

- Ein **Verkaufssystem für Theaterkarten** (Theaterkartenautomaten) war Gegenstand einer Simulation in den Tutorien. Dabei wurde ein Objektmodell des Datenbestandes des Systems (strategische Objekte) erstellt. Darüberhinaus liefert dieses Beispiel die Grundlage für die Aufgabenstellung in Runde 4, (TeachSWT@Tü, 2002f,p), und schließlich wurden dynamische Aspekte des Verkaufssystems in Runde 7 (TeachSWT@Tü, 2002i,s), modelliert.

  Zwischen der Anwendung objektorientierter Methoden und der Aufgabenstellung in Runde 4 existiert kein besonderer Zusammenhang. Diese Fallstudie ist sehr unvollständig ausgeführt, verspricht aber viel für die Arbeit mit Studenten, da das Thema einfach genug ist, dass keine besonderen Fachkenntnisse eingebracht werden

müssen, zum anderen aber auch das System komplex genug ist, dass eine Modellierung lohnt, und sei es nur in Teilaspekten, um ein gemeinsames Vokabular für das Entwicklungsteam zu schaffen.

- In Runde 6 wurde ein statisches OO-Modell für den Kern eines **Webshop** erstellt (TeachSWT@Tü, 2002h,r). Dieses könnte noch entsprechend erweitert, verfeinert oder in Teilansichten zerlegt werden.

- Wenn man möchte, kann auch die Spezifikation des **Heap** aus Runde 3 (TeachSWT@Tü, 2002e,o), zu den Fallstudien gezählt werden: Dieses Beispiel kann dahingehend ausgebaut werden, dass der Heap auf eine endliche Größe beschränkt wird (es ist dann möglich, dass die Allokation scheitert) und der Mechanismus der Speicherbelegung explizit beschrieben wird. Beide Modelle wären dann durch eine Retrieve-Funktion miteinander zu verbinden.

4 Bewertungsschlüssel

Ich werde im Folgenden die Bewertungskriterien für die Übungsblätter, sowie die auf Einzelaufgaben oder -aspekte vergebenen Punkte angeben.

Ich möchte dabei betonen, dass es sich dabei nicht um fertige Korrekturschemata handelt. Ursprünglich war geplant (siehe Leypold, 2002) die abgegebenen Lösungen vor den Testaten zu sichten, um den Umfang zu schätzen, in dem die Aufgaben bearbeitet worden waren, und in den Testaten im Wesentlichen nur die Fähigkeit der Teilnehmer, ihre Lösung kurz und knapp anhand der Unterlagen zu skizzieren, zu bewerten. Dieser Plan war so nicht durchführbar, da sich 10-15 min Testat (für bis zu 3 Blätter) als zu kurz herausstellten, nicht zuletzt wegen der teilweise vollständig konfusen Lösungen. So wurde es nötig, solche Lösungen im Vorfeld genauer anzusehen, um herauszufinden, ob und wo darin das eine oder andere Körnchen Sinn verborgen war. Daraus entwickelte sich dann die Praxis, die abgegebenen Lösungsversuche immer vorher zu untersuchen, vorzubewerten und eventuell im Testat Möglichkeiten zur Verbesserung zu lassen.

Nach dem ursprünglichen Konzept hätten unverständliche Abgaben schlicht als *falsch* bewertet werden sollen – ebenso Testate, in denen es dem Teilnehmer nicht möglich war, eine sinnvolle Erklärung abzugeben. Dies ließ sich aber aus Gründen der relativen Gerechtigkeit im Vergleich mit anderen Vorlesungen nicht durchhalten: Zuviele Teilnehmer hätten den Kurs nicht bestanden[10]. Aus denselben Gründen wurden die späteren Blätter wesentlich großzügiger bewertet als die ersten[11].

[10] Dies ist – auch wenn lieber darüber geschwiegen wird – die im Lehrbetrieb übliche Praxis. Ich möchte hier allerdings die kritische Frage stellen, welchen Sinn es macht, Informatiker, ob im Haupt- oder Nebenfach, auszubilden, die weder verständliche Programme noch Programmentwürfe verfassen können, selbst nicht für einen FIFO, noch das, was sie da vollbracht haben, zusammenhängend referieren können. Sollte es nicht gewisse absolute Maßstäbe geben, die auch durch kollektives Nicht-Können oder -Wissen auf der Teilnehmerseite nicht verrückbar sind? Und um möglichen Entgegnungen, das käme hier und dort („bei uns“) aber nicht vor, gleich vorzubeugen: In der Regel kann das derjenige, der das äußert, gar nicht wissen, da die übliche Praxis der Gemeinschaftsabgabe jede Erkenntnis auf der Dozenten- oder Betreuerseite über den Kenntnisstand der Teilnehmer effizient verhindert.

Es geht mir auch gar nicht daum moralische Schuld für diesen Zustand auf Seiten der Teilnehmer oder der Lehrenden zu finden, sondern um die Tatsache, dass der Lehrbetrieb objektiv nicht funktioniert (und das genannte Unvermögen unserer Teilnehmer ist nur ein oberflächliches Symptom für ein sehr tiefliegendes und umfangreiches Problem). Woran immer das liegen mag: Es wird höchste Zeit etwas dagegen zu tun, anstatt das Problem ständig „woanders“ zu lokalisieren.

[11] Das fiel schon deshalb leichter, weil bei den halbformalen Methoden der Gradient zwischen richtig und vollständig falsch weniger steil ist als bei den formalen. Bei den Letzteren tendieren (in meiner Erfahrung) Lösungen, die nicht vollständig richtig sind, sehr leicht dazu, nun vollständig konfus zu werden. Wenn z. B. ein Datenmodell unangemessen gewählt wird, ist es sehr selten noch möglich, die Operationen mit vertretbarem Aufwand zu formulieren. Das wurde in den Abgaben der Teilnehmer zu den Blättern 3 und 4 besonders deutlich sichtbar.

Die Kriterien zur Bewertung entstanden folgendermaßen: Vor Beginn der Korrektur wurden die Abgaben oberflächlich gesichtet. Das vermittelte einen Eindruck, welche Fehler typischerweise auftraten. An diesen Fehlern wurden die Bewertungskriterien ausgerichtet – d. h. die Bewertung erfolgte häufig entlang Kategorien, in denen die Fehler gemacht wurden. Das bedeutet unter anderem, dass die hier wiedergegebenen Kriterien u. U. für ein anderes Teilnehmerspektrum angepasst werden müssten.

Darüberhinaus fließen in ein Korrektur immer wieder Stilfragen ein – wenn z. B. eine Spezifikation „im Prinzip“ richtig ist, aber leider unverständlich ausgedrückt ist, etwa dass die verwendeten Namen nichts bedeuten oder gar etwas anderes suggerieren, als das, was die benannten Konstrukte darstellen, dann muss dies eigentlich als Fehler gelten.

Aus all diesen Gründen sollten die folgenden Bewertungskriterien ehe als eine Dokumentation dessen angesehen werden, wie im Sommer 2002 bewertet wurde, denn als die Vorgabe von Bewertungsmaßstäben für zukünftige Einsätze von TeachSWT.

Bei allen Blätter bis auf Blatt 7 konnten maximal 100 Punkte erzielt werden. Blatt 7 wurde mit maximal 120 Punkten bewertet. Als 100 % galt die Gesamtpunktzahl von 800 Punkten (d. h. Blatt 7 wurde für diesen Zweck mit nominell 100 Punkten gezählt, die überschüssigen 20 Punkte galten als Bonus). Damit lag die Scheingrenze bei 320 Punkten.

Blatt 1 – *Spezifikation eines FiFo in VDM-SL*: Vier Operationen waren zu spezifizieren. Eine besondere Schwierigkeit scheint es bedeutet zu haben, die Signatur der Operation (die Funktionsköpfe bzw.-typen) richtig hinzubekommen. Deshalb wurde für die richtige Funktionsköpfe (aller 4 Operationen zusammen) 20 Punkte verteilt, sowie für die richtige Spezifikation (Vor- und Nachbedingungen) der 4 Operationen je wiederum 20 Punkte.

Signaturen der 4 Operationen:	20
Spezifikation von *init*():	20
Spezifikation von *insert*():	20
Spezifikation von *extract*():	20
Spezifikation von *isEmpty*():	20

Für die folgenden Fehler wurden (pro Vorkommen) jeweils zwischen 5 und 10 Punkte abgezogen:

- Falsche Funktions-Signatur (meist Fehlen des Resultatparameters für den FIFO-Zustand).
- Fehlende oder unvollständige Vorbedingung.
- Fehlende oder unvollständige Nachbedingung.
- Sinnentstellende syntaktische oder semantische Fehler wie die Nachbedingung $f = \mathrm{tl}\, f$.

Blatt 2 – *Implementation eines FiFo*: Blatt 2 bestand aus zwei Teilaufgaben: Der Implementation eines FIFO in C++ und der Erklärung eines Namenskonfliktes durch pauschalen, unqualifizierten Import in Java. Die erste Teilaufgabe wiederum bestand aus zwei Schwerpunkten: Der Implementation selbst und der Prüfung der Implementation durch Angabe einer Abstraktionsfunktion und nötigenfalls der Korrektur der Spezifikation.

Für die FIFO-Implementation wurden 40 Punkte vergeben, dabei wurde schwerpunktmäßig bewertet, wieweit die Implementation modular gestaltet war, d. h. für Headerdateien, deren

Bei den halbformalen Methoden vermittelt jedoch auch ein Diagramm, in dem einiges fehlt (etwa Constraints) verhältnismäßig viel sinnvolle Information.

Minimalität (nicht zuviel exportiert) und für die Trennung der Namensräume wurden je 10 Punkte vergeben, die restlichen 10 Punkte für eine (mehr oder weniger funktionierende) Implementation:

Implementation:	10
Vorhandensein von Headerdateien:	10
Minimalität der Headerdateien:	10
Trennung in Namensräume:	10

Für die Abstraktionsfunktion wurden 15 Punkte vergeben, für das Erkennen der Implementationsgrenzen, und die korrigierte Spezifikation jeweils 10 Punkte:

Abstraktionsfunktion:	15
Implementationsgrenzen:	10
Korrigierte Spezifikation:	10

Teilaufgabe 2 (Pauschaler Import) wurde mit 25 Punkten bewertet. Dabei wurde sehr großzügig korrigiert, d. h. im Wesentlichen der zugrundeliegende richtige Gedanke gewertet, auch wenn vielen Teilnehmern die Terminologie stark durcheinandergeriet.

Die erste Teilaufgabe dieses Aufgabenblatts war im Verhältnis zu anderen Aufgabenblättern sicher zu schwierig bzw. unterbewertet.

Blatt 3 – *Modellierung eines Heap in VDM-SL*: In Blatt 3 sollte ein Heap modelliert werden (Teilaufgabe 1), dann ein Nullzeiger eingeführt werden (Teilaufgabe 2) und schließlich das Verhalten eines Programmfragments anhand der Spezifikation beurteilt werden (Teilaufgabe 3).

Teilaufgabe 1 (Modellierung von Heap und Zeiger) wurde mit insgesamt 70 Punkten bewertet, dabei wurden 20 Punkte für ein richtiges (oder einsichtiges) Datenmodell vergeben, je 10 Punkte für die zu spezifizierenden Operationen auf dem Heap, sowie 10 Punkte für die Kontrollfrage (Spezifikation von *swap*()).

Datenmodell:	20
malloc():	10
free():	10
access():	10
store():	10
swap():	10

Für Teilaufgabe 2 (Einführung eines Nullzeigers) wurden 15 Punkte vergeben. Dabei gab es 5 Punkte Abzug für mindere Fehler (Nullzeiger als Nachbedingung von *malloc*(), dadurch „zu großes“ Datenmodell).

Für Teilaufgabe 3 (Eindeutigkeit bzw. Determinismus der Ergebnisse der Allokation) gab es 15 Punkte. Hier wäre eigentlich ein Beweis erwünscht gewesen (siehe Lösungsskizze), diesen blieben die meisten Teilnehmer jedoch schuldig.

Blatt 4 – *Theatersitze, Make*: Blatt 4 bestand aus zwei Teilaufgaben: Die Modellierung der Topologie einer Stuhlaufstellung in einem Theatersaal und die Spezifikation einer Operation, die ein Gruppe zusammenhängender Stühle belegt, sowie die Erstellung eines Makefile für ein gegebenes Programm (die FIFO-Implementation aus Blatt 2).

Für die erste Teilaufgabe wurden 65 Punkte vergeben, davon wurden 20 für das Datenmodell vergeben (also: die Art, wie die Topologie beschrieben wurde), 30 für das Prädikat

sind-nebeneinander, das angibt, ob eine gegebene Menge von Stühlen zusammenhängt, die restlichen 15 Punkte für die Belegungsoperation *suche-plaetze*() selbst, die dann entsprechend einfach anzugeben war:

Datenmodell:	20
sind-nebeneinander:	30
suche-plaetze:	15

Für das Makefile (Teilaufgabe 2) wurden 35 Punkte vergeben. Das Makefile enthält im Wesentlichen fünf Regeln (Je eine für die drei Objektdateien, für eine ausführbare Datei und für das Phony-Target *all*). Außerdem waren drei Kontrollfragen zu beantworten, nämlich, was jeweils neu übersetzt werden würde, wenn drei gegebene Dateien modifiziert worden wären. Für jede falsche oder unvollständige Regel und für jede falsche Antwort auf die Kontrollfragen wurden je 5 Punkte abgezogen (jedoch natürlich maximal 35 insgesamt).

Blatt 5 – *Entwurf eines Disassemblers*: In Blatt 5 sollte ein Disassembler entworfen werden, indem die einzelnen Module durch Headerdateien und natürliche Sprache beschrieben werden. Zudem sollten die Benutzungsbeziehungen der Module durch ein Diagramm illustriert werden.

Punkte wurden jeweils in den folgenden Aspekten vergeben:

- **Benutzungsbeziehungen** – Darstellung der Benutzungsbeziehungen zwischen den Modulen.
- **Schnittstellenbeschreibungen** – Beschreibung der Schnittstellen der Module in natürlicher Sprache.
- **Headerdateien** – Vorhandensein, Korrektheit und Vollständigkeit der Headerdateien für die einzelnen Module.
- **Abschätzung der Änderungen** – Abschätzung von Konsequenzen der Änderungen der Spezifikation (siehe Aufgabenstellung).
- **Modularität** – Kapselung und Entkoppelung der Module.

In jedem Aspekt wurde unterschieden, ob der jeweilige Aspekt zum einen formal richtig ausgeführt wurde (also zum Beispiel ob Headerdateien vorhanden waren, diese richtig strukturiert (ifdef-Guards), benannt und verwendet waren), zum anderen, ob diese Aspekte für das gegebene Problem inhaltlich sinnvoll gestaltet waren (z. B. ob die Funktionen in den Headerdateien eine sinnvolle Aufgabe im Disassembler erfüllen konnten).

In diesen beiden Dimensionen wurden die Punkte wie folgt vergeben:

	formal	inhaltl.
Benutzungsbeziehungen	5	5
Beschreibung der Modulschnittstellen	15	15
Headerdateien	10	10
Abschätzung der Änderungen	10	10
Modularität	–	10

Es muss kritisch angemerkt werden, dass damit bereits pro-forma-Anstrengungen relativ hohe Punktzahlen erzielen hätten können.

Blatt 6 – *Statische OO-Modelle*: Blatt 6 bestand aus einer Teilaufgabe (A) mit einfachen Fingerübungen in der grundlegende Aspekte der statischen Objektmodellierung an minimalsten Beispielen abgefragt wurden, sowie der Aufgabe (B), ein Datenmodell des Kerns eines Webshopsystems zu erstellen.

Für die Fingerübungen (Teilaufgabe A) wurden insgesamt 30 Punkte vergeben, d. h. 5 Punkte pro Fingerübung.

Für die Bewertung des Webshops-Modells wurden insgesamt 70 Punkte auf unterschiedliche Aspekte vergeben, wiederum, wie schon in Blatt 5, nach formalen und inhaltlichen Kriterien getrennt:

	formal	inhaltl.
Klassen	10	10
korrekte Notation	10	10
Constraints, Nebenbedingungen	10	10
Verständlichkeit	–	10

Blatt 7 – *Dynamische Modellierung mit UML*: Auch Blatt 7 bestand aus einer Teilaufgabe mit Fingerübungen (Teilaufgabe A) und eine Aufgabe (B) in der diesmal einzelne dynamische Aspekte eines Theaterkartenverkaufssystems beschrieben werden sollten.

Wiederum wurden auf die Fingerübungen (Teilaufgabe A) insgesamt 60 Punkte vergeben, d. h. 20 Punkte pro Fingerübung.

Teilaufgabe A.1 (Lebenslauf Warenkorb)	20
Teilaufgabe A.2 (Ablaufdiagramm Webshopbesuch)	20
Teilaufgabe A.3 (Lebenslauf Bestellung)	20

Auf Teilaufgabe 2 (Dynamische Modelle zum Theaterkarten-Verkaufsterminal) wurden insgesamt 50 Punkte vergeben, die auf die einzelnen Diagramme wie folgt verteilt wurden:

Teilaufgabe B.1 (Lebenslauf eines Platzes)	10
Teilaufgabe B.2 (Bedienerführung beim Kartenverkauf)	10
Teilaufgabe B.3 (Verfeinerung: Bedienerführung Bezahlung)	20
Teilaufgabe B.4 (Verfeinerung: Vorbelegung von Sitzen)	20

Punkte wurden für Notationsfehler (vor allem in Teilaufgabe A) und inhaltliche Fehler abgezogen.

Blatt 8 – *Strukturierte Analyse, MVC-Pattern*: Für die strukturierte Analyse wurden Punktabzüge zwischen 10 und 20 Punkten, je nach Schwere, für die folgenden Fehler verhängt:

- Notationsfehler in Datenflussdiagrammen oder Data-Dictionaries.
- Logische Fehler in Datenflussdiagrammen, d. h. der spezifizierte Prozess kann eigentlich mit den gegebenen Eingabe die Ausgaben nicht erzeugen.
- Bedeutungsunklarheiten, d. h. ungenaue Bezeichnungen für Prozesse und Datenelemente, deren Bedeutung auch aus den Minispecs nicht sinnvoll extrapoliert werden kann.
- Zu wenig durchdetaillierte Diagramme. Die Grenze lag hier bei etwa 3-4 Datenflussdiagrammen.

Allgemeine Bemerkung zur Bewertung: Die letzten Blätter (6,7,8) wurden relativ großzügig bewertet, einmal, weil UML leider zum Teil eine recht vage Angelegenheit ist, zum anderen wegen der überwältigenden schlechten Praxis, die sich auch in der Literatur findet (etwa das Nichtangeben von Constraints), die es wirklich schwierig machen, die Lösungen von Teilnehmern abzuwerten, wenn sie dieselben Auslassungen enthalten.

5 Probleme, die im Verlauf des Kurses auftraten

5.1 Einleitung und Übersicht

Ich möchte in den folgenden Abschnitten im Detail auf die Probleme eingehen, die sich bei der Bearbeitung der Aufgaben durch die Teilnehmer ergaben. Wie wir relativ früh im Semester feststellen konnten, wiederholten sich bestimmte Fehler in den angegebenen Lösungen relativ häufig. Die Ursachen dafür sind unterschiedlich: Mangelnde Kenntnisse und Mitarbeit der Teilnehmer kommen ebenso in Frage, wie ungenügende Anleitung und mißverständliche Aufgabenstellungen.

Ich werde im Folgenden die typischen Fehler aufzählen, wo möglich klassifizieren, Vermutungen über ihre Ursache anstellen und Empfehlungen geben, wie Ihr Auftreten künftig vermieden werden kann, beispielsweise durch geänderte Aufgabestellung oder Abfolge des Stoffes, oder vertiefte Anleitung und verbesserte Unterweisung in bestimmten Teilthemen. Ich will damit nicht implizieren, dass sich alle vorgefundenen Fehler bestimmten Klassen oder typischen Fehlermustern hätten zuordnen lassen: Viele Fehler waren in ihrer Art einzig und sehr individuell, zuweilen blieb der abgegebene Text auch vollständig uninterpretierbar.

Ich werde nun zuerst Probleme, welche typisch waren für einzelne Blätter (mit einem Wort: themenspezifische Probleme) Blatt für Blatt durchgehen. Dann werde ich zum Abschluss kurz einige Probleme ansprechen, die der Übungsbetrieb als solcher aufwarf.

5.2 Blatt 1 – *Spezifikation eines FiFo in VDM-SL*

In Blatt 1 sollte ein FIFO in VDM-SL beschrieben werden. Den Teilnehmern unterliefen typischerweise die folgenden Fehler:

1. Der Zustand des FIFO wurde nur in der Liste der Eingabeparameter einer Operation aufgeführt, nicht jedoch unter den Resultaten. Beispielsweise wurde statt

1.0 $enqueue\,(f : fifo, j : job)\; f' : fifo$
.1 post ...

spezifiziert

2.0 $enqueue\,(f : fifo, j : job)$ **Falsch!**
.1 post ...

2. VDM-SL wurde auch als imperative Programmiersprache missverstanden. So wurden zum Beispiel Nachbedingungen der folgenden Art angegeben:

```
3.0  extract(f : fifo) j : job
.1   pre     ...
.2   post    ...   ∧
.3        f = tl f ;
```

⇒ Den beiden eben gezeigten Fehlern liegt offensichtlich die gleiche fehlerhafte Auffassung zugrunde, namentlich die, VDM-SL wäre eine *Programmiersprache*, ja sogar eine *imperative* Programmiersprache, in der Namen als Namen für Speicherplätze dienen. Dies wurde auch in Gesprächen mehrfach so geäussert. In Wirklichkeit ist (zumindest für Zwecke dieser Lehrveranstaltung) VDM-SL eine *Notation für Aussagen*, die darin vorkommenden Identifizierer sind Namen für Werte oder Resultate.

Abhilfe könnte es hier schaffen, diese beiden unterschiedlichen Konzepte nocheinmmal gezielt anzusprechen, als verschieden zu betonen und nochmals auf den Funktionsbegriff der Mathematik hinzuweisen. Langfristig und etwas globaler könnte auch das gezielte Einbringen mindestens einer funktionalen Programmiersprache ins Grundstudium helfen.

Beide Fehler – d. h. allgemein dass Missverständnis, VDM-SL wäre eine imperativ aufzufassende Programmiersprache – fanden sich auch noch in den Abgaben für spätere Aufgabenblätter (bis Runde 4): Zum Teil dürfte sich das damit entschulden lassen, dass sich die Teilnehmer nicht über ihre Fehlauffassung im Klaren waren, da die Testate über die ersten Blätter erst nach dem zweiten oder dritten Blatt abgelegt wurden, d. h. in der Regel nach dem Abgabetermin der vierten Runde. Eine Verbesserung des Feedbacks zu den Teilnehmern könnte hier helfen.

Andererseits – und dies relativiert die Hoffnung, eine Verbesserung des Ablaufes in dieser Hinsicht könnte diese Probleme beheben – können die Teilnehmer, bei denen die erläuterten Fehler auftraten, auch das ausgegebene Beispiel (TeachSWT@Tü, 2002w) nicht verstanden haben, möglicherweise blieb es sogar ungelesen. Dort nämlich kommt der Typ *repository* – der Zustand des Arrays – sowohl in der Ein- als auch in der Ausgabe vor. Im diesem Beispiel wird zudem (in Abschnitt 5) auf die Behandlung von Zustand in der funktionalen Spezifikation hingewiesen.

3. Einige Teilnehmer verfassten Spezifikationen für Strukturen, die nicht zu FIFOs äquivalent waren, z. B. Druckerwarteschlangen in die Aufträge mit einer durch den Klienten (!) erzeugten ID eingestellt werden, oder ähnliches.

4. Andere Teilnehmer wiederum versuchten, im ausgegebenen Beispiel (das assoziative Array, TeachSWT@Tü, 2002w) die Namen und Parameter umzubenennen und das Resultat als FIFO auszugeben.

⇒ Bei diesen Fehlern habe ich zuzeiten wirklich gefragt, ob ich nicht eigentlich halluziniere. Spätere Aufgabenblätter haben mich schließlich davon überzeugt, dass diese Versuche durch pro-Forma-Abgaben Punkte zu erwirtschaften, nicht selten sind, aber nicht die Mehrheit stellen.

Maßnahmen zur Abhilfe kann ich hier nicht empfehlen: Ein Teil der Teilnehmer scheint wirklich nicht gewusst zu haben, was ein FIFO ist – wenn dem tatsächlich so war, muss hier ein weiteres didaktische Problem diagnostiziert werden – ein anderer Teil scheint die Aufgabe nicht gelesen zu haben, die dritten wiederum haben es „einfach mal probiert – das würdest Du auch so machen, wenn Du am Tag vor der Abgabe nicht fertig

wirst.“ Obwohl ich diese Argumentation für fragwürdig halte, möchte ich kurz auf die Ausführungen zum Abschluss dieses Kapitels (Seite 36) hinweisen: Diese Einstellung ist ein *kulturelles* Problem, d. h. vielschichtig, multikausal, und historisch an den deutschen Hochschulen gewachsen. Die Suche nach einfachen Ursachen („die Studenten sind minder begabt“) und Lösungen („mehr Kontrolle“) muss da nicht nur erfolglos bleiben, sondern könnte auch leicht das Problem noch verschärfen, da die Lösungsversuche der Vergangenheit („mehr Kontrolle“) die Ursachen („mehr Druck“) für die Problemlage der Gegenwart (Unterschleifversuche, Orientierung auf Punkte, statt auf erworbenes Wissen) stellen.

5. Weiterhin wurde in unerwünschter Weise von Teilen der VDM-SL-Notation Gebrauch gemacht, einmal von der direkten Definition – was natürlich auch wieder die Ähnlichkeit zu Programmiersprachen betont – das andere Mal von der Operationensyntax.

⇒ Abhilfe muss hier geschaffen werden, indem – etwa in der Aufgabenstellung – nochmals darauf hingewiesen wird, dass für Zwecke dieses Kurses nur eine Untermenge der Notation verwendet werden soll – d. h. Operationen sollen funktional und durch Vor- und Nachbedingungen spezifiziert werden – und besonders, welche Notationselemente nicht verwendet werden sollen, obwohl sie im Handbuch stehen.

6. Einige Teilnehmer simulierten den Sequenzdatentyp „zu Fuß“, d. h. durch endliche Abbildungen, oder schrieben sich eigene Funktionen zum Aneinanderfügen von Sequenzen.

⇒ Auch wenn dies, wenn es richtig gemacht wird (was aber meist leider doch nicht der Fall war), nichts schadet, so sollten die Teilnehmer doch nochmals explizit auf den bereits vorhandenen Reichtum an Notation in VDM-SL hingewiesen werden. Während funktionale Sprachen als primitive Operationen auf Listen aus Darstellungs- und Effizienzgründen gerade mal *cons*, *head* und *tail* zur Verfügung stellen, spielen diese Erwägungen für eine Spezifikationssprache keine Rolle.

7. Die Aufforderung zum Kommentieren der erarbeiteten Lösung hat wenig gebracht: Die Kommentare geben meist nur in langen Worten wieder, was entweder schon offensichtlich ist, oder sowieso in der Aufgabenstellung steht.

Moral: Auch Kommentieren will gelernt sein – nicht nur in VDM-SL. Dem sollte man im Kurs vielleicht sogar einen eigenen Abschnitt widmen.

5.3 Blatt 2 – *Implementation eines FiFo*

Für Blatt 2 waren zwei Teilaufgaben anzufertigen: Zum einen sollte der FIFO aus der ersten Runde implementiert werden, die Implementation gegen die Spezifikation geprüft und gegebenenfalls die Spezifikation entsprechend den Beschränkungen der Implementation korrigiert werden. Zum anderen war ein Namenskonflikt in einem Szenario mit pauschalem unqualifizierten Import zu erklären und Schlussfolgerungen aus diesem Vorfall zu ziehen.

Zu meinem Erstaunen wurde von vielen Teilnehmern entweder die eine oder die andere Teilaufgabe nicht angefertigt. Ich nehme an, dass diejenigen, die die Implementationsaufgabe nicht ausführten, zuwenig C++ zu beherrschen glaubten. Die andere Teilaufgabe wurde dem Vernehmen nach nicht bearbeitet, weil die betreffenden Teilnehmer die Erklärungen

im Handout *Module und Namensräume* (TeachSWT@Tü, 2002u) mangels Vertrautheit mit Übersetzungs- und Linkprozessen nicht nachvollziehen konnten[12].

5.3.1 Erste Teilaufgabe: FIFO-Implementation

Eine häufige Klasse von Fehlern, die in den Implementationen der Teilnehmer auftraten, drehen sich um Verstöße gegen Prinzipien modularer Programmierung – soweit solche für C/C++ formuliert werden können.

1. Sehr häufig wurde vergessen, die Übersetzungseinheiten in Namensräume einzuschließen. Stattdessen wurden manchmal Klassen eingeführt – aber das genügt nicht[13].

2. Manchmal wurden keine Headerdateien für die Module verfasst, stattdessen wurden die Deklarationen in den geschriebenen Modulen wiederholt. Der bizarrste Fall war der, dass die Implementation vollständig in den Headerdateien untergebracht wurde: In Form von Inline-Methoden (What C++ can do for you …).

⇒ EFASS stand als Beispiel für die modulare Strukturierung von C/C++-Quelltext zur Verfügung. Einige Teilnehmer waren erfolgreich darin, die entscheidenden Punkte zu verstehen und auf die vorliegende Aufgabe zu übertragen. Es muss aber zugegeben werden, dass dieses Thema in der Vorlesung nicht sehr lange oder explizit behandelt wurde. Deshalb kann es, um die erläuterten Fehler abzustellen, nicht schaden, mit den Teilnehmern ein Beispiel für modulare Implementation in C/C++ im Detail durchzugehen, auf die obigen Fallstricke explizit hinzuweisen und zu erklären, warum bestimmte Implementationsstrategien eine schlechte Idee sind (auch wenn C/C++ sie erlaubt und sie recht weit verbreitet sind).

3. Ein häufiges Problem war, dass Teilnehmer über keine Handhabe zu verfügen schienen, die Entscheidung für *Pass-by-Value* versus *Pass-by-Reference* systematisch zu treffen. So wurde zum Beispiel das Folgende definiert:

```
class queue{
  int  blen;
  int  count;
  job* jobs;

  // ...
}
```

[12] Auch hier kann ich nur nochmals darauf hinweisen, dass für eine erfolgreiche Durchführung eines Softwaretechnik-Kurses eine gewisse Erfahrung in der praktischen Programmierung – und zwar am besten nicht in einer IDE – vorausgesetzt werden muss. Softwaretechnik ist kein Programmierkurs, sie muss thematisch auf einem solchen aufbauen.

[13] Entsprechend dem, was in *Module und Namensräume* (TeachSWT@Tü, 2002u) ausgeführt ist, geht es darum, die von jedem Modul zur Linkzeit *in den globalen Linkzeitnamensraum* eingebrachten Namen soweit zu regulieren, dass keine Namenskonflikte auftreten *können*, also ein Benennungssystem zu finden, das Namenskonflikte verhindert. Dies funktioniert ohne eine zentrale Instanz zur Vergabe von Namen am Besten, wenn die exportierten Symbole den Modulnamen mit sich tragen, da die Modulbezeichner aus Prinzip eindeutig sein müssen (ob deren Namensraum nun hierarchisch ist – wie in Java – oder flach – wie in Modula-2 oder OCaml). Diesem Zweck kann in C++ das *namespace*-Konstrukt dienstbar gemacht werden. Dagegen funktioniert der Export einer Klasse (ohne herumgewickelten Namensraum) nur dann in diesem Sinn, wenn (a) nur genau eine Klasse exportiert wird, und (b) diese den Namen des Moduls trägt. Dürfte man gegen das zweite Kriterium verstossen, könnten z. B. ein Drucker-Modul und ein Scheduler-Modul beide unqualifiziert (d. h. ohne Voranstellung des Modulnamens) eine Klasse mit dem Namen *queue* exportieren.

```
queue queue::enqueue(queue q; job j)    // sic !
{
  // ...

  jobs[count]=j;
  count++;

  // ...

  return q;
}
```

Dies wurde dann so aufgerufen:

```
q = enqueue(q,j);
```

Dies suggeriert funktionalen Stil und funktioniert so, wie hier aufgeschrieben, sogar, da die temporäre Kopie *eines Teils des Zustandes* des Objekts wieder in die ursprüngliche Variable zurückgeschrieben wird, aber

```
q2 = enqueue(q1,j);   // q1 now invalid
q3 = enqueue(q1,j);   // q2 now invalid, potential memory leak
```

funktioniert schon nicht mehr. Es ist natürlich auch unsinnig, den Zustand eines *Containers* zu kopieren, also selbst ein oberflächlich korrekt erscheinendes *deep copy* – etwa durch eine entsprechendes Definition der Assignment-Methode – wäre hier sinnwidrig und ineffizient.

⇒ Abhilfe könnte her dadurch geschaffen werden, dass die Thematik *Pass-by-Value* versus *Pass-by-Reference* aus dem Handout *Spezifikation durch Funktionen und die Behandlung von Speicher* (TeachSWT@Tü, 2002v) früher in den Kurs eingebracht wird, und vor allem die Teilnehmer auf die Gefahren des *Representation-Sharing*, das berührt auch die Frage danach, wann *deep copy* und wann *shallow copy* zum Einsatz kommen sollten, explizit hinzuweisen. Die Abstraktionsfunktion kann dabei wertvolle Dienste leisten, weil mit ihr beschreibbar ist, wenn sich der scheinbare Inhalt einer Variablen als Seiteneffekt einer Operation auf einer ganz anderen Variablen plötzlich ändert, d. h. die durchgeführte Operation *nicht mehr die spezifizierte Signatur hat*, da sie nun zwei neue Werte zum Resultat hat, und zudem auch keine Chance besteht, die Invarianten der Darstellung des implizit veränderten Wertes zu erhalten.

4. Die Implementation einiger Teilnehmer schrieb im Ausnahmefall (voller FIFO) eine Warnung auf die Standardausgabe, veränderte den FIFO nicht, gab aber auch keine Fehlerindikation zur Klientenprozedur zurück.

⇒ Es steht natürlich außer Frage, dass „wildes“, d. h. unspezifiziertes Schreiben in die Standardausgabe für kein Modul außer den designierten Ein- und Ausgabemodulen in Frage kommt.

Was das die unterschlagene Fehlerindikation betrifft, könnte es helfen, die Rolle des Kontraktes für den Versuch, die Korrektheit des Klientenmoduls zu beweisen, zu betonen. So bleibt ohne ein Fehlerflag bei folgendem Quelltextfragment der Klientenprozedur

```
queue::enqueue(&q,&j)
          //
          //          [1]  Is j in q? You can't know ...

          // ...
```

bei Punkt [1] keine *logische* Handhabe mehr, irgendwelche Zusicherungen darüber zu machen, ob der Job *j* sich nun in der Warteschlange befindet oder nicht. Dagegen ist es in

```
if (OK=queue::enqueue(&q,&j)){
          //
          // [2] contract correlates OK and state of q after
          // call to enqueue. OK is true when we arrive here,
          // so we _conclude_ that now j must be in q.

          // ...
} else {
          //
          // [3] contract correlates etc. ...
          // OK is false when we arrive here, so we conclude
          // that now j can't be in q.

          // ...
}
```

tatsächlich möglich, in [2] zu garantieren, dass der Job im FIFO ist.

5. Einige jener Teilnehmer, die – entgegen unserer didaktischen, wohlgemeinten Absicht – in dieser Runde auf die Verwendung von Klassen bestanden, produzierten Module, die genau eine Klasse mit einer Methode enthielten, und jeweils einer Operation entsprachen. Also wurde z. B. ein Modul *PushJob* mit der Headerdatei *PushJob.h* definiert, dass die Definition

```
class PushJob{
  PushJob(FiFo f, Job j);
}
```

enthielt. Hier schweigt des Sänger's Höflichkeit ...

6. Die Aufgabenstellung zu Abstraktions- und Retrievefunktionen wurde zumeist nicht bearbeitet.

Schlussfolgerung: Es muss zugegeben werden, dass diese Teilaufgabe bei Weitem zu umfangreich war. In Anbetracht der bereits geschilderten Probleme würde es wohl nicht schaden, wenn das Thema über mehrere Aufgabenblätter verteilt würde, und im Vorfeld jeder Schritt anhand eines anderen Beispiels (etwa anhand eines Stacks oder eines assoziativen Arrays) im Detail durchexerziert würde.

5.3.2 Zweite Teilaufgabe: Pauschaler unqualifizierter Import

1. In dieser Teilaufgabe scheinen die Teilnehmer einigermaßen verwirrt über die Frage gewesen zu sein, welche Art von Konstrukten in Java exportiert wird: Es wurde da von

Prozeduren, von *Packages* und von *Methoden* gesprochen. Tatsächlich ist die einzelne Einheit, welche exportiert wird, die *Klasse*.

⇒ Im Kontext der im Handout *Module und Namensräume* (TeachSWT@Tü, 2002u) angesprochenen Themen wäre es vielleicht nützlich, die in den verschiedenen Sprachen vorhandenen Namensräume (in Java *Packages*, in Modula oder OCaml *Module*), die Einheiten der Übersetzung (in Java *Klassen*, in Modula oder OCaml *Module*), und die Einheiten der Implementation und des Exports (in Java *Klassen*, in Modula oder OCaml *Prozeduren* und *Typen*) in Übersicht und im Vergleich zu besprechen.

2. Wie aus dem vorangegangenen Problem schon zu erwarten, waren manche Lösungsvorschläge für Java unangemessen: Zum Teil wurde vorgeschlagen, die Klassen des Package „in einen Namespace zu kapseln“[14], zu, Teil wurde empfohlen, allen Namen im Package mit den Namen des Package voranzustellen[15].

3. Letztlich und endlich waren die zu dieser Aufgabe formulierten Stellungnahmen häufig und reichlich mit leeren Floskeln angereichert, wohl in der Hoffnung, durch Fallenlassen der richtigen Stichworte möglichst viele Punkte einfahren zu können.

5.4 Blatt 3 – *Modellierung eines Heap in VDM-SL*

In Runde 3 sollte mit VDM-SL die Schnittstelle eines unendlichen Memory-Heap beschrieben werden. Diese Aufgabe war die erste ernsthafte Modellierung, welche die Teilnehmer selbst durchführen sollten.

1. Der bei Weitem häufigste Fehler bestand in der Wahl bizarrer – d. h. vollkommen ungeeigneter – Datenmodelle. Ein Teil dieser Modelle entstand wohl aus dem Versuch, *entgegen dem, was in der Anleitung stand*, doch zu versuchen, die Allokation aus dem unterliegenden linearen Systemspeicher explizit zu beschreiben.

⇒ Für Abhilfe könnte hier zweierlei sorgen: Einmal, dass noch deutlicher betont wird, dass die Organisation der zur Verfügung gestellten Speicherblöcke aus dem unterliegenden Systemspeicher nicht beschrieben werden soll, ja, dass es sich dabei um eine Aufgabe handeln würde, die wesentlich schwieriger und umfangreicher wäre. Zum anderen könnte darauf hingewiesen werden, dass zwischen einem Heap und einem assoziativen Array große Ähnlichkeiten bestehen: Beide dienen dazu, Daten aufzubewahren, und die Daten werden unter Verwendung entsprechender Schlüssel (nur dass diese beim Heap *Zeiger* genannt werden) eingeschrieben und können mit diesen Schlüsseln extrahiert werden.[16]

2. Bei denen, die auf ein plausibles Datenmodell verfallen waren, war der häufigste Fehler der, die Nachbedingungen nicht ausreichend streng zu fassen. In fast allen diesen Fällen wurde zwar die Modifikation des adressierten Blocks korrekt beschrieben, jedoch vergessen, zu fixieren, dass sich die restlichen Daten des Heap nicht ändern.

[14] Ich vermute hier, dass dieser Satz als Floskel irgendwo, möglicherweise aus dem Handout, abgeschrieben wurde. In Java *sind* die Packages der Namensraum.

[15] Auch dieser Vorschlag wurde wohl einfach ohne tieferes Verständnis aus dem genannten Handout übernommen.

[16] Es besteht, wenn man diesen Hinweis gibt, natürlich die Gefahr, dass die Teilnehmer, die es sich einfach machen wollen, versuchen, die Spezifikation des Heap durch Kopieren und Abwandeln des ausgegebenen Beispiels zum assoziativen Array zu erhalten – und dabei die Hälfte vergessen.

⇒ Dieser Fehler unterläuft mir selbst oft. Ich halte das für ein psychologisches Problem: Man ist so auf den eigentlichen Nutzwert der Operation konzentriert, dass man vergisst, die restlichen Freiheitsgrade zu eliminieren. Besonders häufig geschieht das in Fällen, in denen ein Selektor (Schlüssel, Index, Name, Zeiger, Adresse) dazu verwendet wird, einen Teil einer zusammengesetzten Struktur (Array, Dateisystem, Heap, Speicher) zum Update auszuwählen.

Hier hilft wohl nichts weiter, als entweder die Aufmerksamkeit für genau diesen Effekt zu schärfen, oder aber zu lehren, auch Eindeutigkeitsbeweise für Resultate von Operationen zu führen[17].

3. Manchmal wurde der *Nullzeiger* mit einem Zeiger *auf* einen Block der Länge 0 verwechselt.

4. Vielfach wurde die Ungültigkeit des Nullzeigers nicht in der Invariante des Heapzustandes verankert, sondern durch die Nachbedingung von *malloc* impliziert.

⇒ Der Gedankengang war – wie ich bei Diskussionen mit den Teilnehmern feststellen konnte – dass, wenn *malloc* den Zeiger nie zurückliefert, dieser auch nie auf einen gültigen Block verweisen kann. Dieser Gedankengang ist zwar prinzipiell richtig, gehört aber doch mehr der Welt der algebraischen Spezifikation an, wo die Operationen den Zustandsraum erzeugen. In der modellorientierten Spezifikation dagegen existiert das Datenmodell vor den Operationen, d. h. im vorliegenden Fall endet man mit einem zu großen Datenmodell: Dieses beschreibt Heapzustände, die auch durch beliebige Folgen von Operationen nicht erreicht werden können[18].

Auf diese Thematik muss wohl auch nochmal explizit hingewiesen werden: Datenmodelle müssen im angedeuteten Sinn minimal sein.

5. Ein letztes Problem bei dieser Aufgabe (aber auch bei anderen) scheint schließlich in Schwierigkeiten bestanden zu haben, Konzepte präzise sprachlich zu fassen und in mathematische Notation zu übertragen. Als Beispiel möge das Folgende genügen: Ich erinnere mich, dass auf der Helpdesk-Mailingliste ein Teilnehmer darauf bestand, dass *„ein Pointer* eine Länge *hat*“. Gemeint war die Länge des Blocks auf den der Zeiger verweist.

⇒ Ganz allgemein gesprochen verwundern einige der Probleme, die sich bei der Bearbeitung dieses Blattes ergaben, nicht, wenn man sich die Probleme der ersten beiden Blätter vor Augen führt, wo auch schon eine gewisse Verwirrung um Speicher und Referenzen

[17] Das ist sicher aufwendig. Nichtsdestotrotz muss man sich überlegen, welchen Nutzen eine formale Spezifikation bringt, wenn damit nicht ein Instrumentarium einhergeht, Defekte zu erkennen und auszumerzen. Ein Korrektheitsbeweis zeigt, dass ein Teil einer Spezifikation (also etwa eine Verfeinerung) einen anderen korrekt implementiert. Er beweist leider nicht, dass die ursprünglich Spezifikation sachlich richtig – also dem Problem angemessen war. *Validation conjectures*, d. h. aus dem fachlichen Sachverhalt innerhalb des Problembereichs formulierte Behauptungen über das spezifizierte System, welche nach VDM-SL übersetzt werden, können hier helfen. Können die *validation conjectures* aus der Spezifikation widerlegt werden, erfüllt das spezifizierte System die „naiven“ Erwartungen nicht, und man muss sich fragen, ob die Spezifikation dem Problem nicht angemessen ist oder ob die Erwartungen selbst widersprüchlich sind. Eine mögliche *validation conjecture* für die Nachbedingung des Heap besteht in der Antwort auf die Frage: Welche Datenelemente des Heaps ändern sich maximal? *Validation conjectures* erfordern natürlich die Einführung von Beweistechniken, aber das ist u. U. auch aus didaktischen Gründen gar nicht falsch: Zuviele Teilnehmer haben nach meinem Eindruck VDM-SL lediglich als eine kuriose Notation gesehen.

[18] Nach den Regeln müssen dabei aber trotzdem alle Beweise über die gesamte Zustandsmenge geführt werden: Es ist klar, dass hier ein ganzes Nest von Problemen verborgen liegt.

bzw. Zeiger auftrat. *Ein bemerkenswerter Bruchteil der Teilnehmer schien mit dem Begriff* Zeiger *nichts anfangen zu können.*

5.5 Blatt 4 – *Theatersitze, Make*

Blatt 4 enthielt wieder 2 Teilaufgaben: Einmal sollte eine – verhältnismäßig abstrakte – Beschreibung einer Operation zum Belegen zusammenhängender Gruppen von Theatersitzen erstellt werden, zum anderen ein Makefile für ein Programm mit drei Modulen (es handelte sich dabei um die Implementation aus Aufgabe 2).

5.5.1 Erste Teilaufgabe: Saalbestuhlung

Eigentlich war die erste Teilaufgabe eine Verlegenheitslösung, da zu diesem Zeitpunkt der Stoff etwas knapp wurde, und die Vorlesung noch nicht weit genug fortgeschritten war für die Aufgabe, die dann in Runde 5 gestellt wurde. Trotzdem neige ich jetzt dazu, gerade dieser Aufgabe eine gewisse Wichtigkeit zuzuschreiben, da sie die Notwendigkeit für und die Vorteile einer abstrakten Beschreibung essentieller Teile des Problems (hier: der Topologie der Bestuhlung) vor Augen führt: Ohne diese Reduktion auf das Essentielle war eine übersichtliche Lösung der Aufgabe nicht möglich. Die bei dieser Aufgabe mehrheitlich begangenen Fehler bestehen vor allem in einer unangemessenen, unnötig komplizierten oder detaillierten, oder schlicht semantisch leeren Definition des Datenmodells. Wenn von solchen Modellen ausgegangen wurde, musste dann auch die Spezifikation der Operationen fast zwangsläufig an deren Umständlichkeit scheitern.

1. Viele versuchten (trotzdem die Aufgabenstellung gerade das Gegenteil suggeriert) die Bestuhlung als ein rechteckiges Array von Stühlen, d. h. als Folge von Folgen zu beschreiben. Meistens waren es genau diese Teilnehmer, die zudem zum Typ *stuhl** (Folge von *stuhl*) als Ergebnistyp für verschiedene Operationen griffen, anstatt *stuhl*-set (Menge von *stuhl*) für diesen Zweck einzusetzen. Aus beidem ist nicht verwunderlich, dass die Spezifikationen dann häufig explizite Iterationen (bzw. Rekursionen) über die Elemente dieser Folgen enthielten, mit allen Komplikationen, die dies mit sich brachte.

$\Rightarrow$ Als Abhilfe kann ich mir vorstellen, die Teilnehmer künftig detaillierter zum Lösungsweg hinzuleiten. Die aktuelle Aufgabenstellung ist in dieser Hinsicht noch recht vage. Insbesondere sollte explizit angeleitet werden, dass es einmal darum geht, den Begriff der Nachbarschaft über die Anschauung eines 2-dimensionalen Punktgitters hinaus zu verallgemeinern, dahingehend, dass *Nachbarschaft* eine Relation zwischen je zwei Stühlen ist.[19]

Bezüglich des Einsatzes von Listen statt Mengen als Ergebnistypen – und den dadurch hineingebrachten Iterationen über die Liste, das sind explizite Beschreibungen sequentieller Verarbeitungsprozesse – könnte es nützen, mit den Teilnehmern einige pragmatische Aspekte des Einsatzes verschiedener Typen als Modelle zu erarbeiten (also: wann welche Typen sinnvoll eingesetzt werde). Man könnte unter anderem anhand eines Beispiels darauf hinweisen, wie eine Liste eine Reihenfolge impliziert, also unter bestimmten Umständen *zuviel* Information enthält, und insbesondere darauf, dass, während für eine Listendarstellung eine sequentielle Suche nötig ist, um Elemente mit bestimmten Eigenschaften zu fin-

[19] Die Beschreibung einer Topologie dieser Art ist natürlich kein softwaretechnisches Lernziel, sondern eher ein mathematisches Problem. Was hier jedoch gelernt werden kann, ist, wie die richtige Beschreibung des Gegenstandes die Problemlösung stark vereinfacht. An dieser Fähigkeit und an einem ausreichenden mathematischen und natürlichsprachigen Vokabular zur Beschreibung von Gegenständen und Problemen mangelte es meines Erachtens der Mehrheit der Teilnehmer. Und das wird sich – die ungute Trennung von Theorie und Praxis an deutschen Hochschulen mitbedenkend – auch bis zum Diplom wenig ändern.

den, bei Mengen der Existenzquantor – in einem gewissen Sinn – dieselbe Funktion auf einfachere Art erfüllen kann[20].

2. Eine weitere Klasse fehlerhafter Datenmodelle wurde von Teilnehmern verfasst, die wohl im Prinzip den Ansatz, Nachbarschaft durch eine Relation zwischen Stühlen zu beschreiben, richtig erkannt hatten. Leider versuchten sie diese Relation in den Typ Stuhl zu integrieren und formulierten:

```
types

  stuhl ::
     rechter_nachbar : stuhl
     linker_nachbar  : stuhl
          ...                ;
```

Dies ist in VDM-SL sinnlos, auch wenn das vom Typprüfer nicht erkannt wird, da die induktive Definition der Wertemenge von *stuhl* damit keinen Induktionsanfang hat.

⇒ Es scheint mir, als ob hier objektorientiertes Denken eingeflossen wäre: Ich bin der Ansicht, dass hier Werte, nämlich die Teile des *composite*-Typen *stuhl* mit Referenzen auf Objekte verwechselt worden sind. [21]

Hier müssen die Teilnehmer darauf hingewiesen werden, dass ein *Typ* (in den meisten Sprachen) einer *Wertemenge* entspricht. Dagegen sind – und das wird hier recht gut sichtbar – *Klassen* ganz andere Konzepte: Eine Klasse trägt Informationen über Ihren Kontext (Assoziationen) immer mit sich. [22]

Stellt man in Rechnung, wie sehr die Möglichkeit, den Rest der Aufgabe zu lösen, von der geschickten Wahl des Datenmodells abhängt, so empfiehlt es sich, die Aufgabe künftig auf zwei oder mehr Übungsrunden zu verteilen, wobei z. B. in der ersten das Datenmodell erarbeitet und in der zweiten die Operationen spezifiziert würden.

Überhaupt sollte wohl der Erarbeitung von Datenmodellen – eventuell aus natürlichsprachigen Beschreibungen – unabhängig[23] von Operationen[24] eine größere Aufmerksamkeit geschenkt werden. Das lehren auch die Erfahrungen aus den vorangegangenen Blättern.

⇒ Zum Abschluss der Diskussion dieser Teilaufgabe noch eine Anekdote aus dem real existierenden Übungsbetrieb: In einer der Abgaben fand sich (sinngemäß) die etwas schnippische Bemerkung: „Was uns der Auto hier mit so vielen Worten sagen will, ist, dass hier

[20] Eben dann, wenn man keine Ordnung und keine mehrfache vorkommenden Elemente benötigt.

[21] Ich stelle mir vor, dass denjenigen, die nur Java beherrschten, diese Sprache ein Stolperstein war, da sie den Unterschied zwischen Referenzen und Werten syntaktisch vollkommen verwischt.

[22] Ob man damit glücklich ist, ist eine ganz andere Frage. Was hier klar wird, ist, dass die Relationen, in denen ein Objekt einer Klasse stehen kann, zum Typ der Klasse gehören (siehe Runden 6 und 7), während die Relationen in denen z. B. ein *stuhl* in der vorliegenden Aufgabenstellung vorkommt, mit dem Typ *stuhl* nichts zu tun haben. Alle Versuche, Interpretationen mit Wertemengen und Funktionen bzw. Relationen auf objektorientierte Modelle zurückzufitten, müssen deshalb konzeptionell unbefriedigend bleiben, auch wenn sie vielleicht sogar mathematisch korrekt sind: Die Begriffswelt klassischer funktionaler Spezifikation und die der Objektmodellierung sind im Grunde nicht miteinander vergleichbar. Hier hat wirklich ein Paradigmenwechsel stattgefunden. Das Wesen eines echten Paradigmas (EVA gehört da eigentlich nicht dazu) ist es, das Denken *total*, d. h. von Grund auf, *zu verändern*, nicht, es lediglich zu ergänzen.

[23] Auch wenn dies *semantisch*, d. h. für die Bedeutung des Systemmodells, widersinnig scheint, macht das didaktisch durchaus Sinn, da solche Datenmodelle durch ihre Syntax und die gewählten Bezeichner meist eine pragmatische Bedeutung suggerieren, von der man sich leiten lassen kann.

[24] Aber natürlich inklusive der Attributfunktionen. Es scheint mir didaktisch günstig zu Anfang zwischen Attributen (des beschriebenen Gegenstandes) und Operationen auf oder mit dem Gegenstand zu trennen, auch wenn die Unterscheidung später wieder etwas verwischt wird.

die Stühle einen Graphen bilden, wobei ein Knoten einem Stuhl entspricht, eine Kante die Nachbarschaft symbolisiert, und dass ein zusammenhängender Teilgraph gefunden werden muss“.

Was für den Teilnehmer ein Manko war, nämlich, dass dies nicht so in der Aufgabenstellung stand, sondern gewissermaßen „verkleidet“ wurde, ist, so denke ich, in Wahrheit ein Charakteristikum der Softwaretechnik: Was viele nicht erkannten, war, dass Softwaretechnik zu einem großen Teil die Kunst ist, in einer für den Anwendungsbereich spezifischen Formulierung (Bestuhlung, Nachbarschaft) die dahinterstehende programmtechnische oder mathematische Struktur (Graphen) zu sehen. Dagegen ist vom Anwender nicht nur nicht zu erwarten, dass er in Stacks, Graphen und Protokollen über sein Problem spricht, sondern auch gefährlich, da der Laie sicher keine angemessene Lösung vorschlagen wird.[25]

5.5.2 Zweite Teilaufgabe: Makefile

1. Am Häufigsten wurden bei der Erstellung des Makefile die Edukte (Makejargon: *Dependencies*) in den Regeln vergessen. Besonders gerne wurden „indirekt“ benutzte Headerdateien, wie z. B. *jobs.hh* bei der Übersetzung von *queues.cc*, vergessen.

⇒ Abhilfe könnte es hier schaffen, den vollständigen Übersetzungsprozess für ein C/C++-Projekt aus mehreren Modulen graphisch zu veranschaulichen (Compiler und Präprozessor wären dabei getrennt darzustellen) um klarzustellen, dass jede Eingabedatei als Edukt aufzuführen ist, auch, wenn sie im Übersetzungsskript (Makejargon: *Action*) nicht explizit genannt wird. Eventuell ließe sich die Aufgabe um eine Teilaufgabe ergänzen, in der eine solche Illustration für den in der Aufgabe als Arbeitsmaterial dienenden Quelltext eingefordert wird.

2. Einige Teilnehmer testeten ihre Makefiles ohne die eingebauten Regeln mit dem Flag *-r* ausser Kraft zu setzen und konnten dementsprechend vergessene Regeln nicht bemerken.

3. Zum Teil war ein gewisses Unverständnis zum einen für den Sinn, zum anderen für die Umsetzung von Make-Regeln mit administrativer Funktion (Makejargon: *Phony-Targets*) festzustellen. Dies wurde auch so mehrfach im Testat geäussert. Einige Teilnehmer implementierten *all* als ein Target ohne Edukte, in dessen *Actions* sie alle Übersetzungsbefehle nochmal aufführten.

5.6 Blatt 5 – *Entwurf eines Disassemblers*

In Runde 5 sollten die Teilnehmer einen Disassembler im Detail entwerfen, d. h. es sollten Module festgelegt und sowohl durch Headerdateien syntaktisch, als auch in ihrer Leistung mittels natürlicher Sprache beschreiben werden.

Es handelt sich dabei um eine relative freie Aufgabe, für die keine Schritt-für-Schritt-Anleitung gegeben werden kann. Sie erfordert einen gewissen Überblick, sowohl über das entstehende Produkt als auch die verschiedenen Aspekte des Entwurfs- und Spezifikationsprozesses, die in den vorhergehenden Runden behandelt wurden. Zudem muss ein gewisses Gefühl für die Ästhetik der entstehenden Struktur mitgebracht werden.

Vor allem deshalb lässt sich für die nun folgenden Fehler auch kein schnelles Rezept ihrer

[25] Sondern zynisch gesagt, etwas, das so aussieht, „wie sich Fritzchen vorstellt, dass Computer funktionieren“.

Beseitigung angeben.[26] Vielmehr werde ich mich meist darauf beschränken müssen, festzustellen, wann und wo mich Lösungen verwirrt, oder unbefriedigt zurückgelassen haben.

Möglicherweise hilft es, mit den Teilnehmern im Vorfeld dieser Aufgabe gute Entwürfe (d. h. Entwürfe, die die Organisatoren des Kurses nach den Kriterien des Kurses als typisch empfinden) und auch die Frage nach der Ästhetik von Programmen und Entwürfen zu thematisieren (z. B. übersichtliche und sinngebende Benutzungsbeziehungen von Modulen).

1. Viele der abgegebenen Entwürfe wiesen eine wenig aussagekräftige Gliederung in Module auf[27]. Deren Verantwortlichkeiten blieben dabei vollkommen im Dunkeln. In Konsequenz konnten dann die häufig die Kontrollfragen, welche sich darum drehten, welche Programmteile geändert werden müssten, wenn sich bestimmte Anforderungen ändern, nicht beantwortet werden.

⇒ Möglicherweise hilft es hier, *Kochrezepte*[28] zum Finden von ADTs und deren Gruppierung zu Modulen zu vertiefen. Ich selbst habe damit – mit der Zergliederung von Systemen anhand mehr oder weniger abstrakter Datentypen, aber auf jeden Fall anhand der Datenstruktur – immer recht gute Erfahrungen gemacht. Ob sich aber andererseits jede Applikation geschickt in Datentypen zerlegen lässt, scheint mir nicht gesichert. Der Versuch rentiert aber allemal.

2. Einige der abgegebenen Graphen der gegenseitigen Benutzung der Module wiesen Zyklen auf. Da, soweit ich mich später noch durch Stichproben überzeugen konnte, in diesen Fällen keine Headerdateien erstellt worden waren, lässt sich schwer sagen, *was* eigentlich beim Zeichnen dieser Graphen gedacht wurde, d. h. welches Missverständnis oder welche Fehlauffassung letzten Endes zu solchen Graphen geführt hat. Ich vermute aber, dass nicht Importbeziehungen dargestellt wurden, sondern möglicherweise (unbeschriftete) Datenflüsse.

3. Einige Architekturen waren so angelegt, dass implizit während der Ausgabe disassembliert wurde, d. h. einem Druckmodul wurden *Opcode* und *Operand* in einem Prozeduraufruf übergeben, diese von der Prozedur in einen String übersetzt und dieser an den Druckertreiber übermittelt.

⇒ Eine solche Architektur muss sehr kritisch betrachtet werden, da sie darauf zählt, dass sich jede Instruktion für sich genommen disassemblieren lässt. Wenn z. B. symbolische Sprungziele generiert werden sollen, wäre dies (ohne eine kompliziertes Mitführen von

[26] Wie „beseitigt“ man d. h. Unmusikalität? Ein Gefühl für Ästhetik von Programmen, Systemarchitektur, Konzepten, mathematischen Beweisen usw. lässt sich wohl *trainieren*, aber schwerlich durch einen schrittweises Besprechen von Regeln (oder Style-Guides) erzwingen bzw. durch einen Prozess rationaler Einsicht erreichen. In jedem Fall kann man wahrscheinlich ausgehen, dass die Entwicklung musikalischen, malerischen, mathematischen oder eben auch programmiererischen Talents (das schließt für mich jetzt Entwurf, eben die ganze Softwaretechnik, mit ein) nicht ohne ein „Sich-Darauf-Einlassen“, ohne intensives Mitgehen des Lernenden (hier sage ich lieber: des sich Bildenden) zu erreichen ist: Rein *reaktiv* ist da nichts zu machen. Aus diesem Grund kann ein Herausholen von Regeln, wie z. B. Entwurf zu machen ist, eine Verbalisierung des Vorgehens in kleinen Schritten, nur dem helfen, der bereits motiviert ist, d. h. es kann ihm geholfen werden sich zu verbessern.

[27] Das betrifft sowohl Anzahl, Benutzungstopologie, als auch Benennung der Module. Recht üblich waren Entwürfe, in denen ein Modul mit der Bezeichnung „Main“ drei Untermodule kontrollierte: „Eingabe“, „Disassemblierung“ und „Ausgabe“.

[28] Es dürfte sich dabei unter anderem um etwa fünf bis zehn Faustregeln handeln, die aber alle keinen strikte Charakter haben, deshalb die Bezeichnung *Kochrezept*. Da sich ADTs und Klassen nicht ganz unähnlich sind, dürften diese Rezepte in leicht abgewandelter Form auch für das Finden von Klassen recht nützlich sein.

akkumulierter Information und Pufferung der generierten Daten im scheinbaren Ausgabemodul) nicht mehr so einfach möglich. Die Anwendung des EVA-Prinzips hilft, diese Art von Strukturfehlern zu vermeiden.

4. Die natürlichsprachigen Beschreibungen der Modulschnittstellen empfand ich häufig als unpräzise und die verwendeten Begriffe als schwammig.

Manchmal wurden statt der datenverarbeitenden Leistung einer Prozedur (das ist das Verhältnis von Ein- zu Ausgabe …) die Operationen beschrieben, die diese Prozedur vollzieht, um diese Leistungen zu erbringen, z. B. mit Sätzen der Art „*X* sendet *Y* an *Z*“ (was immer „senden“ auch mit einem Disassembler zu tun haben mag).

⇒ Ich kann mir vorstellen, dass es hier helfen würde, die natürlichsprachige Spezifikation zuerst mittels eines „Fragebogens“ zu üben, der etwa Fragen der folgenden Art enthält:

- Kapselt das Modul Zustand?
- Ist dieser Zustand programm-intern oder -extern?
- Aus welchen Teilen besteht dieser Zustand?
- Welches sind die Eingabedaten der Prozedur?
- Von welchen weiteren Faktoren (Zuständen!) hängt das Resultat der Prozedur ab?
- Wie werden Eingabedaten an eine Prozedur übergeben (hier sollte man ruhig mit dem traditionellen Ausführungsmodell von Stack und Speicher für prozedurale Sprachen arbeiten).
- usw.

Einige Autoren aus der Epoche der strukturierten Methoden (McMenamin und Palmer, 1984; Page-Jones, 1988) referieren die Idee eines *structured English*, das ist ein stark ritualisiertes und stereotypisiertes, eingeschränktes Englisch für die Beschreibung von Algorithmen und Verfahren in (quasi-) natürlicher Sprache. Ich möchte vorschlagen, in analoger Weise mit den Teilnehmern ein natürlichsprachiges (deutsches oder IT-deutsches) Standardvokabular zu erarbeiten und zu definieren – zum einen, um einen Pool von Wörtern und Phrasen zur Verfügung zu stellen, aus dem geschöpft werden kann[29], zum anderen, um den Sinn für präzise Beschreibungen in natürlicher Sprache zu schärfen. Dabei wären unter anderem auch Begriffe wie *Wert*, *Referenz*, *Pass-by-Value*, *Pass-by-Reference*, *Update*, *Überschreiben*, *Parameter*, *Datenfluss*, *Aufruf* , *übergeben*, usw. zu klären und im Kontext kleiner Beispiele zu exerzieren.[30]

[29] Das Schöne wäre, dass ein Verb wie „senden“ in diesem Pool nicht vorkäme, also vom Autor eines Dokuments vor Gebrauch (irgendwie) definiert werden müsste.

[30] Das mag leicht lächerlich erscheinen, ich habe jedoch bereits im vorangegangenen Text erwähnt, dass auch das präzise, natürlichsprachige Beschreiben etwas ist, das gelernt und gepflegt werden will. Weiterhin habe ich hier die Vermutung, dass sich die Begriffswelt, in der sich Informatikstudenten bewegen in den letzten Jahren stark geändert hat, so dass man, was vor 10 Jahren noch selbstverständlich war, heute nicht mehr einfach voraussetzen kann (das hat, ich möchte das betonen, nichts mit einer Absenkung des Niveaus zu tun, sondern zuerst einmal mit einer Verschiebung von Schwerpunkten, was dann zu einem laufenden Hindernis für die Kommunikation zwischen der Generation der Lehrenden und der Lernenden wird): Für einen frühere Generation hat sich der Erstkontakt mit dem Computer auf dem Niveau einfacher Systeme etwa von der Komplexität eines Mikroprozessors mit wenigen Peripheriebausteinen abgespielt. Diese Generation ist mit den modernen Sprachen und Paradigmen (funktionale Programmierung, Objektorientierung) gewachsen und das prinzipielle Wissen um die unterliegenden Strukturen bildet den kulturellen Untergrund. Die jüngere Generation dagegen hat diese Technologien bereits fertig vorgefunden und muss sich nun durch die Schichten hoher Abstraktion im Verständnis nach unten durcharbeiten, d. h. diese modernen Konzepte (Objekte, Botschaften) erst in primitivere Konzepte zerlegen

7. Die schiere Menge des zum Teil ungegliederten Textes, den manche Teilnehmer abgaben, war in einigen Fällen nicht lesbar, und es bestand mangels Strukturierung bzw. Gliederung auch keine Möglichkeit, sich einen Überblick zu verschaffen.

⇒ Es handelt sich hierbei wohl um allgemeine Schwierigkeiten damit, Informationen gegliedert und in übersichtlicher Form zu präsentieren. Ich habe in Praktika der Physik gute Erfahrungen damit gemacht, klare Richtlinien zur Form eines Berichtes zu geben und dies mit einem Beispiel zu illustrieren. Ich würde auch für die Übungen zur Softwaretechnik vorschlagen, mit den Teilnehmern eine Standardgliederung zur Präsentation eines modularen Feinentwurfs zu erarbeiten. Es schadet nichts, hier den Spieß umzudrehen, und darauf aufmerksam zu machen, dass es sich hier um eine Publikation bzw. Repräsentation einer vom Teilnehmer erbrachten Leistung handelt, und dass es letzten Endes im Interesse des Autors liegt, diese so übersichtlich und eingängig als möglich darzustellen.

Dagegen ist es nicht der Auftrag des Betreuers, aus einer ungeeigneten Darstellung Körnchen von Richtigkeit herauszulesen. Im „wirklichen Leben“ würde der Auftraggeber wohl den Entwurf in den Papierkorb befördern und das Projekt an jemand anderen vergeben[31]. Aus dieser Motivation – Übersichtlichkeit und geschickte Repräsentation – müsste man die erarbeitete Gliederung auch begründen.

7. Manche Teilnehmer gaben handgeschriebene unvollständige und syntaktisch falsche Headerdateien ab.

⇒ Ich habe den Verdacht, es könnte sich dabei eher um Versuche gehandelt haben, zwischendurch in bester Schulmanier[32] „mal schnell“ Abgaben anderer Teilnehmer zu kopieren, als um ernsthafte Entwicklungsarbeit. Trotzdem schadet es nicht, die Teilnehmer auf die Möglichkeit und Notwendigkeit hinzuweisen, die Headerdateien vom Compiler syntaktisch und im Bezug auf Typen prüfen zu lassen (siehe TeachSWT@Tü, 2002q, Abschnitt 5.1:*Bemerkungen zum Vorgehen*).

8. Einige Teilnehmer scheinen „binäre“ Daten und die zu ihrer Notation verwendete hexadezimal Darstellung durcheinandergebracht zu haben: Jedenfalls wimmelt es in deren Abgaben von Identifizierern wie „*hexheader*“ und „*hexcode*“.

9. Letztlich und endlich waren viele der abgegebenen Erklärungen und Beschreibungen von allgemeinem „Gobbledigok“ durchsetzt. Ich gebe im Folgenden einige Perlen wieder:

(Referenzen, Zeiger, Speicher).

Für die jüngere Generation ist das kein selbstverständliches, von Anfang an vorhandenes Wissen, und muss deshalb explizit gelehrt werden. Es ist aber unverzichtbar, dass dies geschieht, da die moderneren Programmierideologien oft Konglomerate primitiverer Konzepte sind (und nicht Abstraktionen, die für sich verstanden werden können). Beispielsweise muss man, um *Objekte* zu verstehen, die Begriffe *Zustand*, *Speicher*, *Referenz* und *Prozedur* verinnerlicht haben. Dies ist schwierig für Leute, die Prozeduren Zeit ihres Lebens nur als *Methoden* kennengelernt haben und als die einzige Art Zustand zu speichern, die Kapselung in ein Objekt.

[31] Das heisst, ich will hier fair sein: Ich habe auch im „wirklichen Leben“ sogenannte Fachkonzepte gesehen, die an Sabotage grenzten – nur dass niemand von den Beteiligten das zugeben konnte oder wollte. Der aktuelle Stand der industriellen Praxis in diesem Bereich berechtigt zu Pessimismus.

[32] Ich bitte darum, hier keine überflüssige Paranoia zu vermuten. Eine gewisse Kopieraktivität war in der Tat immer in den Pausen der Vorlesung zu beobachten und hat mich sehr an die Vorgänge auf der Kellertreppe meiner ehemaligen Schule erinnert. Eigentlich ist es lächerlich, die zukünftigen Experten bei solchen Manövern zu beobachten. Verfolgt haben wir diese allerdings nicht – weniger aus Toleranz, als vielmehr in der Erkenntnis, dass dies einen hochschulweiten Konsens bezüglich einer Etikette zur „akademischen Ehrlichkeit“ voraussetzen würde. Wüsste man, dass jeder das, was er abgegeben hat, auch wirklich selbst angefertigt hat, könnte man die Messlatten auch leicht etwas absenken und hätte im Großen und Ganzen doch an Qualität gewonnen.

- „Nachbedingung: Zwei neue Parameter ersetzen den Adressmodus.“
- „Gibt ein transform [sic!] zurück, das aus t entstanden ist, wobei t my_bin_header neu auf bin gesetzt wird“
- Interessanterweise findet sich dann eine Seite weiter der folgende Text: „Gibt ein transform zurück, das aus t entstanden ist, wobei t my_data_header neu auf data gesetzt wird“. Gleiche Wiederholungen finden sich für *table* und *code* statt *data*.

⇒ Blatt 5 war im Nachhinein gesehen sicher zu umfangreich. Eine schrittweise Heranführung über mehrere Runde (Essentielle Systemdaten identifizieren, Abstrakte Datentypen identifizieren, Modulstruktur entwerfen, usw.) hätte vielleicht den Teilnehmern zu besseren Abgaben verhelfen können.

Nichtsdestotrotz muss der Erwerb der Fähigkeit, Systeme von mindestens der Komplexität und Größenordnung dieses Disassemblers selbst planen, entwerfen und umsetzen zu können, ein wichtiges Lehrziel der Veranstaltung bleiben. Den Aufwand für eine Analyse (wie in Runde 8: TeachSWT@Tü, 2002j,t), einen Entwurf im gezeigten Detaillierungsgrad (wie in Runde 5: TeachSWT@Tü, 2002g,q) und anschließender Implementation würde ich allerdings durchaus auf 80-120 h, wenn nicht gar mehr, schätzen. Dies entspricht dem *gesamten* Zeitkontingent, das Studenten im optimistischsten Fall für die Vorlesung zur Verfügung halten.

Es stellt sich hier also wieder die Frage, inwieweit Softwaretechnik überhaupt an realistischen, praktischen Beispielen erfahrbar gemacht werden kann. Zudem wirkt die Anwendung der Methoden der Softwaretechnik auf zu einfache Beispiele (FIFO, Stack) künstlich – es wird mit Kanonen auf Spatzen geschossen – und ist deswegen, außer zu Demonstrationszwecken vor dem Ernstfall, von zweifelhaftem didaktischen Wert.

5.7 Blatt 6 – *Statische OO-Modelle*

Die Aufgabenstellung von Blatt 6 umfasste sechs sogenannte Fingerübungen, d. h. Aufgaben für kleine Klassendiagramme mit ein bis zwei Klassen, und den Auftrag, ein Objektmodell für die wesentlichen Klassen eines Webshops zu erstellen (also die Geschäftsklassen oder strategischen Klassen zu finden).

5.7.1 Erste Teilaufgabe: Fingerübungen

1. Der häufigste in dieser Aufgabe begangene Fehler waren Verwechslungen von Assoziationen und Aggregationen. So wurden Aggregationen notiert, wo Assoziationen richtig gewesen wären (also: „Ein Konto besteht aus Kunden“) und umgekehrt.

⇒ Die Unterscheidung, wann Aggregationen und wann Assoziation zum Einsatz kommen sollten, ist, zugegeben, nicht ganz einfach. Nach meinem Dafürhalten hilft das in der handelsüblichen Literatur zur Objektorientierung verbreitete „ontologische“ Argumentieren wenig. Dabei wird versucht, über Verhältnisse der „realen“[33] Welt argumentierend die bewusste Unterscheidung zu treffen, also mittels Sätzen der Art „Ein Haus besteht aus Zimmern“ und so weiter. Die Anzahl der als „wahr“ oder akzeptabel angenommenen Sätze über die Verhältnisse der Dinge der „realen“ Welt ist jedoch zu vielfältig, um eine klare Richtschnur für die im Modell zu treffenden Festlegungen zu geben. Stattdessen benötigt man ein Kriterium, das einen prinzipiellen – qualitativen – Unterschied von Aggregation

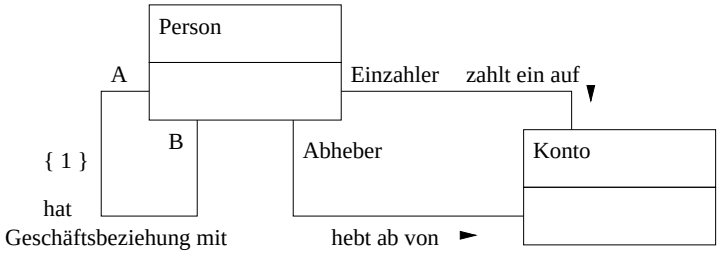
[33] Anführungszeichen hier wegen des konstruktivistischen Vorbehalts :-). Gemeint ist die Sicht der Welt, in der sich unser Alltagsleben abspielt.

und Assoziation *innerhalb des Modells* und mit der Begriffswelt der Modellierung festlegt. Diese Begriffswelt redet über Botschaften, Zustände, Operationen, Constraints und ähnliches. Ich schlage hier vor, dass eine Aggregation (genau?) dann vorliegt, wenn die untergeordneten Objekte vom übergeordneten verwendet werden, um dessen Zustand darzustellen – oder anders gesagt: Wenn sich die Attribute des übergeordneten Objektes aus denen der untergeordneten berechnen lassen, bzw. entscheidend durch diese eingeschränkt werden[34].

Da, wie erläutert, die Literatur bei diesem Thema meines Erachtens gegen die Regeln verstösst, und plötzlich einen Rückgriff auf Wissen über die Welt fordert, ist eine klare Absprache mit den Teilnehmern über den Einsatz von *Aggregation* und *Assoziation* notwendig. *Aggregation* und *Komposition* andererseits sollten nicht unterschieden werden (siehe dazu Lösungskizze 6: TeachSWT@Tü, 2002r). Die Teilnehmer sollten angeleitet werden, zuerst alle Beziehungen als Assoziationen zu modellieren, und erst nach der Ermittlung aller wesentlichen Attribute alle Assoziationen daraufhin zu überprüfen, ob diese nicht – entsprechend obigem Kriterium (oder anderen, die noch zu erarbeiten wären) – in Aggregationen umgewandelt werden sollen.

Der Erfolg dieses Vorgehens hängt, wie auch die erfolgreiche Vermeidung der im weiteren angesprochenen Fehlerarten, davon ab, dass Teilaspekte der Diagramme in einer ganz bestimmten Reihenfolge erarbeitet werden. Hier sollte man darüber nachdenken, ob es nicht sinnvoll wäre, den Teilnehmern einen auf die Vorlesung abgestimmten Vorgehensplan – inklusive Prüflisten und Testfragen für die OO-Modellierung – an die Hand zu geben. Bezüglich der Verwertbarkeit der in der Literatur gegebenen Vorgehenspläne für OOA und OOD (Balzert, 1996; Yourdon, 1996) hege ich Zweifel: Diese halte ich entweder für zu lang (-weilig) oder bereits überholt.

2. Auffällig war, dass ein guter Anteil der Teilnehmer in der sechsten und letzten Fingerübung statt einer zur Klasse *Person* zurückführenden Assoziation „hat Geschäftsbeziehung mit“, eventuell mit entsprechenden Rollenbezeichnungen, etwa so

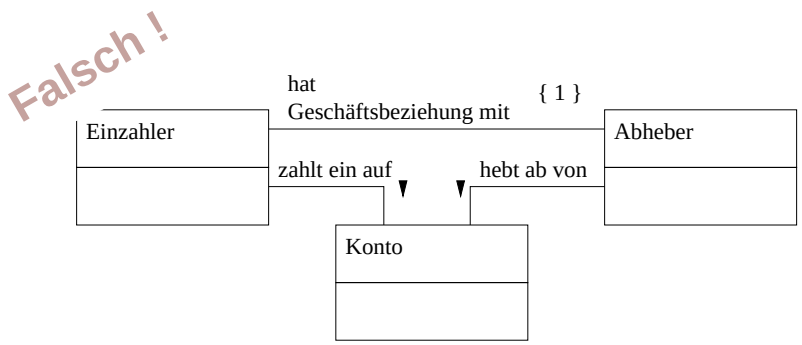


zwei separate Klassen *Einzahler* und *Abheber* modellierte:

[34] Zwei Beispiele: Im „traditionellen“ ontologisch argumentierten Fall „Ein Auto hat Räder“ würde ich für eine Modellierung durch Assoziation plädieren. Räder können abgenommen werden, an andere Autos montiert werden, für den nächsten Winter aufgehoben werden, und so weiter. Sollte es doch in irgendeinem Ausschnitt des Problems so sein, dass man eine fixe Zuordnung haben will, d. h. jedes Auto genau vier Räder haben soll und jedes Rad genau einem Auto zugeordnet sein soll, so formuliert man einfach entsprechende Kardinalitäten für die Assoziation. Das beweist einmal mehr, wie irreführend die ontologischen Argumente sein können.

Dagegen würde ich den Fall „Ein Text besteht aus Zeilen“ wirklich als Aggregation modellieren, da der Zustand des Textes im Wesentlichen in den Zeilen gespeichert ist, aus dem er besteht. Das Textobjekt abstrahiert also über die Zeilenkollektion und fügt dem vielleicht noch ein bisschen Zustand hinzu. In ähnlicher Weise abstrahiert (summiert) ein Kontoobjekt über die Geschichte der Einzelbuchungen: Der Kontostand kann nicht als unabhängige Variable begriffen werden. Auch hier halte ich Aggregation für die angemessene Wahl zur Modellierung.

Man sieht, dass die Entscheidung zwischen Aggregation und Assoziation dem Bereich des Feinentwurfs zuzurechnen ist, da hier bereits Implementationsfragen aufgeworfen werden.



⇒ Hier muss man zum einen den Gedanken der unterschiedlichen *Rollen* in denen Objekte agieren können (unabhängig von der Polymorphie, die üblicherweise mit der Vererbung einhergeht), noch tiefer verankert werden. Zum anderen liegt auch der Verdacht nahe, dass denen, die den illustrierten Fehler begingen, nicht ganz klar war, dass die „Kästen“ des Klassendiagramms eben *Klassen* und keinen einzelnen *Objekte* (*Instanzen*) repräsentieren. Ein Klassendiagramm entspricht also einer Beschreibung einer möglicherweise unendlichen Menge von Objektszenarien (für die es meines Wissens keine UML-Notation gibt). Es könnte helfen, eine separate Notation für Objektszenarien einzuführen und die Frage der Entsprechung zu gegebenen Klassendiagrammen (eventuell in Form einer Aufgabe: „Welche der folgenden Objektszenarien werden durch das Klassendiagramm beschrieben?“) explizit zu thematisieren.

3. Viele der Constraints waren ungenügend oder aussagelos („*X* muss konsistent sein“ – das war zwar ein wörtliches Zitat aus der Aufgabenstellung, hätte aber noch elaboriert werden sollen[35]).

⇒ Auch wenn ich hier keine Vorstellung habe, wie sich dies in den Übungsbetrieb integrieren lässt, so könnte es vielleicht helfen, Spezifikation als Teil eines Argumentationsspiels aufzufassen: Die gegnerische Partei punktet, wenn sie ein (unsinniges) Szenario vorlegen kann, das durch die Spezifikation nicht ausgeschlossen wird.[36].

Über alle Aufgaben hinweg schienen die Teilnehmer dazu zu tendieren, zu allgemeine Klassen von Systemen zu spezifizieren, statt im Zweifel soweit als möglich einzuschränken. Typische Fragestellung im Tutorium: „Soll das System nicht auch noch … können?“.

Den Teilnehmern müssten zum einen die Augen geöffnet werden für die Gefahren dieses Vorgehen: Entweder muss in allen Programmteilen ein viel zu weites Spektrum von Fällen behandelt werden, oder der einzelnen Programmierer entscheidet nach seinem Gutdünken, welche Fälle nach seiner Auffassung der Sachlage nicht vorkommen können, da sie unsinnig seien. Zum anderen sollten die Teilnehmer darauf hingewesen werden, dass eine einschränkende Spezifikation in späteren Phasen der Entwicklung viel Arbeit (und Geld) sparen kann, da sie klar sagt, was *nicht* implementiert werden soll.

[35] Überhaupt habe ich eine allgemeine Vorliebe der Teilnehmer für wichtig klingende, aber allgemeine Worte und Redewendungen, wie „effizient“, „korrekt“ (im Bezug auf welchen Maßstab?), „konsistent“, „richtig“, „abstrakt“, „entsprechend“, „alle erdenklichen“, „muss vorhanden sein“ (wo?), „fehlerfrei“, „stabil“, „robust“ usw.. bemerkt. Ich gebe offen zu, dass dies zum Teil auch ein Problem des vorliegenden Textes ist, möchte das aber dennoch zum Anlass nehmen, zu fragen, ob man hier nicht viel bewusster auf einen extrem konservativen, d. h. vorsichtigen Sprachgebrauch hinwirken sollte. Die Branche produziert schon genug heiße Luft.

[36] Ich erinnere mich vage, dass es in der Wissenschaftstheorie ähnliche Ansätze gibt, Logik als Dialog zu beschreiben und Beweise als Rezepte zur Erzeugung von Entgegnungen auf In-Frage-Stellungen der Behauptungen.

5.7.2 Zweite Teilaufgabe: Webshop

Die bei der Modellierung des Webshop begangenen Fehler lassen sich grob einteilen in formale, methodische und prinzipielle.

1. Zu den prinzipiellen Fehlern gehört, dass einige wenige Teilnehmer statt Assoziationen offensichtlich *Datenflüsse* oder *Kontrollflüsse* eingezeichnet haben.

⇒ Ohne genau Nachforschungen, wie es zu diesem „Missverständnis“ kommen konnte, lässt sich über die Ursachen dieses Fehlers wenig sagen. Auffällig ist jedoch, dass sich dieses Thema wiederholt: Wo Kästen und sie verbindende Linien gezeichnet wurden, schlichen sich auch in anderen Aufgaben immer wieder Datenflüsse oder Kontrollflüsse ein.

2. In einigen Fällen wurden Methoden in der falschen Klasse untergebracht, z. B.

$$Abteilung.put\_in\_Warenkorb(Artikel) \quad .$$

⇒ Hier könnte es wiederum helfen, an Beispielen und unter Umständen mit Hilfe einer Prüfliste, Kriterien für die Lokalisierung von Methoden zu vertiefen. Deren Wichtigstes dürfte sein: Verändert eine Operation *m* den Zustand eines Objektes aus der Klasse *K*, und sonst nichts, so muss *m* in der Klasse *K* lokalisiert werden (Kapselung!).

3. Viele Teilnehmer führten Klassen wie *Rechnungsabteilung*, *Buchhaltung*, *Liefersystem* und so weiter, ein.

⇒ Hier wurde wohl die Verb-Substantiv-Analyse zu naiv angegangen. Zum einen muß zum Nutzen dieser Teilnehmer vertieft werden, dass nicht alle Substantive automatisch Klassen sind, zum anderen müssen auch möglichst klare Kriterien erarbeitet werden, wann ein *Kandidat für eine Klasse* dann auch wirklich eine Klasse werden darf.[37]

Schließlich könnte es helfen, Use-Case-Diagramme (eventuell im Catalysis-Stil) einzuführen, um auch die menschlichen Akteure im Modell unterbringen zu können. Dies würde vielleicht den Drang aus jedem Substantiv auf Biegen und Brechen eine Klasse zu machen, etwas dämpfen.

4. Häufig wurden Klassen ohne Attribute oder Methoden eingeführt.

⇒ Ich halte dies zwar in Einzeldiagrammen für prinzipiell zulässig (siehe Lösungsskizze 6 TeachSWT@Tü, 2002r), wenn die Attribute der Klasse im gezeigten Ausschnitt keine Rolle spielen, aber in irgendeinem Teildiagramm einer Spezifikation (die ja aus vielen Diagrammen besteht, in denen diese Klasse wiederholt auftreten kann) sollte eine Klasse durchaus wohldefinierte Attribute besitzen. Hat sie das nicht, dann handelt es sich nur um eine lose Sammlung von Prozeduren, die keinen Zustand kapseln, und vermutlich anderswo besser untergebracht wären (nämlich in einer der Klassen, die als Parameter auftreten). Hierauf zu achten hätte den betreffenden Teilnehmern helfen können, Klassen wie *Rechnungsabteilung* und *Liefersystem* nochmal zu hinterfragen und gegebenenfalls zu vermeiden.

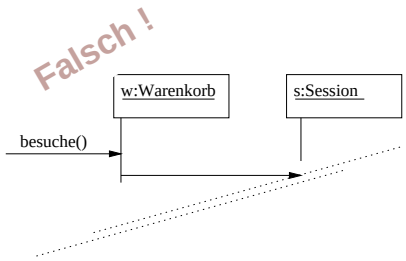
[37] Also auch hier wieder das Thema einer Prüfliste. Die Gefahr, die ich hierin sehe, ist allerdings, dass die Teilnehmer dann vollständig aufhören mitzudenken und die Prüfliste – die nie vollständig sein kann – als einziges Kriterium heranziehen.

5.8 Blatt 7 – *Dynamische Modellierung mit UML*

Blatt 7 war der dynamischen objektorientierten Modellierung gewidmet. Als Fingerübung sollten zuerst zwei einfache Zustandsdiagramme – und zwar der Lebenslauf eines Warenkorbes bzw. einer Bestellung im Webshopsystem von Runde 6 (TeachSWT@Tü, 2002r) – erstellt werden (erste Teilaufgabe). Dann sollte die Interaktion eines Theaterkartenautomaten mit den Benutzern beschrieben und schrittweise verfeinert werden werden (zweite Teilaufgabe).

Da in beiden Teilaufgaben sehr ähnliche Fehler begangen wurden, werde ich hier nicht zwischen den Teilaufgaben differenzieren, sondern die Fehler aus beiden gemeinsam abhandeln:

1. Einige Teilnehmer beschrifteten das erste (initiale) Nachrichtenereignis des Ablaufdiagramms mit einer unsinnigen Botschaft, die keiner Methode des empfangenden Objektes entsprach.



⇒ Ich nehme an, dass hier richtig erkannt wurde, dass die Kausalkette von Ereignissen, die das Ablaufdiagramm beschreibt, ausserhalb des Objektmodells ihren Anfang nehmen muss. Aus Gesprächen mit den Teilnehmern schließe ich aber, dass die Anbindung des in Runde 6 bechriebenen „Kerns“ des Webshops an den Webserver-Mechanismus unklar geblieben ist. Es muss letzten Endes ein Mechanismus existieren, der einen „Klick“ auf eine URL in eine Botschaft an ein Objekt umwandelt. Genau das ist aber für manche Teilnehmer unklar geblieben.

2. Einige Teilnehmer scheinen Zustandsdiagramme als Daten- oder Kontrollflussdiagramme aufgefasst oder missbraucht zu haben.

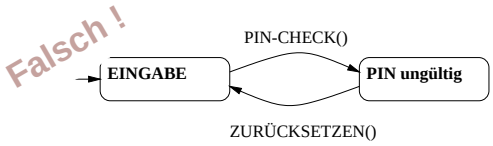
⇒ Kontrollfluss, Datenfluss, Assoziationen usw. beschreiben jeweils verschiedene Aspekte des Systems, z. B. zeitliche Abfolgen von Tätigkeiten, Abhängigkeiten von Daten oder ähnliches. In der Literatur werden diese Aspekte selten im Vergleich behandelt, so dass sich ein Lernender in diesem Bereich schlecht Kriterien zur Abgrenzung erarbeiten kann. Hier könnte es helfen, anhand *eines einzigen* Beispiels, das jedoch strukturiert genug sein muss, um interessant zu sein[38] alle Aspekte der Beschreibung zu illustrieren (Klassendiagramm, Lebenslauf, Datenfluss usw.), um das komplementäre Verhältnis, das diese zueinander haben, sichtbar zu machen.

Ganz allgemein müsste man jedoch darüber nachdenken, wie man den Teilnehmern hier zu einem *differenzierten* Vokabular verhelfen kann (dessen Fehlen ja auch schon in den vorangegangenen Aufgaben ein Problem war), das es erst ermöglichen würde, Linien zwischen Kästen anders als als „Übergabe von X“ oder „Aufruf von Y“ zu interpretieren.

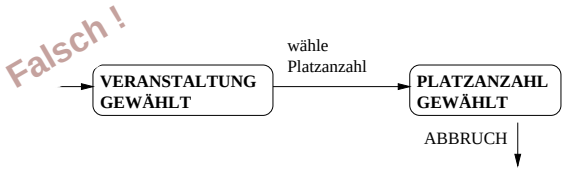
[38] Vielleicht würde sich ein Textobjekt aus einem Texteditor dafür eignen.

3. Häufig wurden Knoten von Zustandsdiagrammen dazu eingesetzt (in meinen Augen: missbraucht), entweder im Stil von Flussdiagrammen Bedingungen zu prüfen, etwa so

oder gar das Ergebnis einer Prüfung zu notieren:

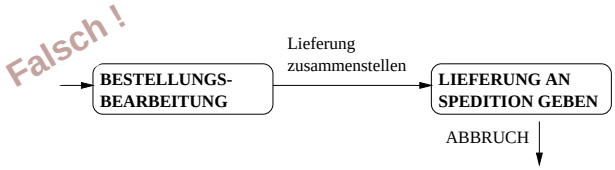


4. Sehr viele Teilnehmer vertauschten in den Zustandsdiagrammen Knoten (Prozesse, Zustände, Vorgänge) und Kanten (Bedingungen, Ereignisse), in etwa so:



Eigentlich soll es ja der Prozess der Platzwahl sein, der abgebrochen werden können soll – „wähle Platzanzahl“ kann also unmöglich an einer Kante stehen.

5. In analoger Weise wurden häufig Prozesse an Kanten geschrieben (hier für den Lebenslauf einer Bestellung):



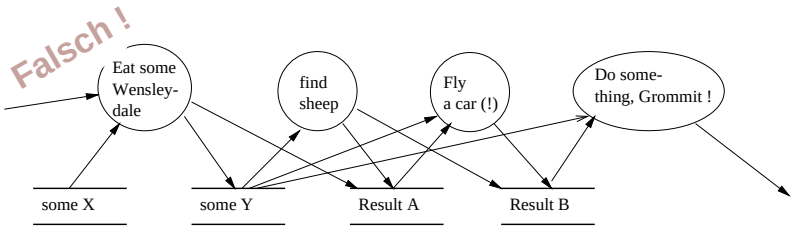
Hier liegt der Fehler darin, dass „Lieferung zusammenstellen“ weder ein externes Ereignis ist, noch als Resultat des Prozesses gesehen werden kann, von dem die Kante ausgeht. Weiterhin ist „Lieferung an Spedition geben“ kein Zustand der Bestellung und schon gar nicht des Objektes, das den Ablauf der Abwicklung einer Bestellung im Rechner nachvollzieht. Es kann allerdings sein, dass es sich dabei nur um ein Problem in der Formulierung handeln. Jedoch selbst das solche Diagramme unlesbar, auch wenn sie – abgesehen von den irreführenden Beschriftungen – logisch äquivalent zu einer richtigen Lösung wären.

⇒ Letzten Endes scheinen viele Teilnehmer hier an der Frage gescheitert zu sein, was ein Zustand (Knoten) und was ein Übergang (Kante) ist. Die Lösungsskizze (TeachSWT@Tü, 2002s) enthält dazu weitere Ausführungen. Die Frage ist zugegeben nicht ganz einfach zu beantworten, wenn man sich nicht ganz auf die suggestive Kraft von Pfeilen, Kästen und Andeutungen durch die Beschriftung verlassen will. Hier ist eine genauere Erklärung für die Teilnehmer nötig.

5.9 Blatt 8 – *Strukturierte Analyse, MVC-Pattern*

In Runde 8 wurden drei Aufgaben zur Auswahl gestellt. Zwei davon bezogen sich auf objektorientierte Entwurfsmuster und waren als eine Gelegenheit für Interessierte und Fortgeschrittene gedacht, ihr Können zu demonstrieren. Die Mehrheit bearbeitete allerdings die Aufgabe zur strukturierten Analyse, auf die ich mich (wie in der Lösungsskizze) für die folgenden Ausführungen beschränken will.

1. Zu den skurrileren Schwächen, die Abgaben zu dieser Aufgabe aufwiesen, gehörte die Einführung einer Unzahl von Speichern, wobei viele Prozesse auf die Mehrzahl aller Speicher eines Diagramms zugriffen:



⇒ Die einschlägige Literatur (McMenamin und Palmer, 1984; Page-Jones, 1988) weist darauf hin, dass es bei der Erstellung von Datenflussdiagrammen in fast beliebiger Weise möglich ist, Speicher einzuführen oder zu eliminieren. Ich stehe allerdings auf dem Standpunkt, dass Speicherelemente meist von der fundamentalen Struktur der stattfindenden Datenverarbeitung *ablenken*. Man sollte den Studenten nahelegen, soweit als möglich ohne Speicherelemente auszukommen. Es gibt (nur) einige wenige Gelegenheiten, zu denen ich auch die Notwendigkeit sehe, Speicherelemente in der strukturierten Analyse zu verwenden, nämlich,

- wenn der Speicher als solcher bereits vor dem zu erstellenden System existiert (z. B. eine Datenbank, die bereits vorhanden ist, und aus der das System wird Daten entnehmen müssen), und
- wenn die Verarbeitungsprozesse ein starkes Element von „Update“ beinhalten, bzw. in einer großen Struktur Teile ersetzt werden. Man wird also statt der linken lieber die Notation der rechten Figur wählen.

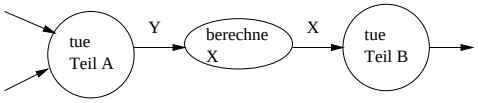
- Wenn es sich um Datenbanken, Dateisysteme, Warteschlangen, generell um *Datencontainer* handelt, die, da problemspezifisch, beschrieben werden sollen, also die Art der Speicherung und die Zugriffsmethode, -ordnung oder -strategie eine Rolle spielt.

Dagegen ist es unsinnig, für jedes ausgetauschte Datum einen Speicher im Stil einer Variable vorzusehen.

2. Viele Teilnehmer führten unnötige Sequenzierungen von Prozessen ein. Dass solche Defekte in der strukturierten Analyse eingeführt werden können, ist bekannt, für Beispiele und Erklärungen möchte ich auf das Skript zur Vorlesung (Klaeren, 2002) verweisen.

3. Häufig mutierten Datenflussdiagramme zu Darstellungen von Kontrollflüssen. Beispiele sind die folgenden Diagramme:

Das rechte Diagramm sollte wohl besser so umgeschrieben werden:



⇒ Bezüglich dieser Fehler muss bei den Teilnehmern wohl vertieft werden, dass der Kontrollfluss – also auch, wann und unter welchen Umständen welche Datenflüsse *ausgelöst* werden – nur in der Minispezifikation festgehalten werden kann. Da ich mir vorstellen kann, dass ein Teil der Teilnehmer damit überfordert war, zum einen einen Datenfluss zu analysieren und zu erstellen, zum anderen aber auch gleichzeitig wohlgeformte Diagramme zu erstellen, empfiehlt sich vielleicht eine Vorübung, in der sich die Frage nach dem Kontrollfluss gar nicht stellt: Man könnte zum Beispiel den Datenfluss einer mathematischen Formel, wie etwa der Lösungsformel für quadratische Gleichungen, als ein Datenflussdiagramm darstellen lassen.

4. Die Benennung der primitive Daten (d. h. der Elemente, die im Datenwörterbuch nicht in Komponenten aufgelöst wurden) war nicht selten unklar und irreführend.

⇒ Ich vermute, dass es das beste sein wird, die Teilnehmer dazu anzuhalten, primitive Elemente in natürlicher Sprache zu beschreiben, und zwar – wo möglich – in ihrer Semantik (also „Die *Kundennummer* bezeichnet einen Kunden der Firma in eindeutiger Weise und ist zur Zeit eine Folge von mindestens 5 und höchstens 10 Ziffern ohne führende Nullen“) statt ausschließlich in ihrer Darstellung (wie „Eine *Kundennummer* ist ein Zeichenkette“).

5.10 Allgemeine Probleme

Die letzten Abschnitte behandelten Fehlerklassen, welche für die Einzelthemen der jeweiligen Runde spezifisch waren. Ich möchte nun im Folgenden einige allgemeinere Probleme ansprechen, welche über alle Runden bemerkt werden konnten, bzw. die Organisation und Struktur der Übungen als Ganzes betreffen.

1. Ganz allgemein haben wir den Aufwand, den die Übungsblätter den Teilnehmern verursachen würden, stark unterschätzt.

⇒ Das soll nicht bedeuten, dass es nicht möglich gewesen wäre, die Aufgaben mit weniger Aufwand zu erledigen. Zum Teil war dieser erhöhte Aufwand auch auf falsche Lösungsansätze zurückzuführen (etwa die Verwendung von Folgen statt Mengen), die dann in viel komplizierteren Lösungen resultierten, als das von uns beabsichtigt war. Man muss aber sehen, dass dies dennoch viel Arbeit für die jeweiligen Teilnehmer bedeutet hat, so unnötig sie auch im Einzelfall gewesen sein mag.

Bei Blatt 2 und Blatt 5 gab es sicher Planungsfehler bezüglich des zu erwartenden Umfangs der Lösung auf unserer Seite. Ich möchte das aber insofern relativieren, als die Bearbeitungszeit für Blatt 2 auf zwei Wochen verlängert wurde, und man in Blatt 5 auch mit wesentlich weniger umfangreichen Lösungen als der vorgestellten hohe Punktzahlen erwirtschaften konnte.

Im allgemeinen waren die Aufgaben recht monolithisch angelegt, d. h. die erfolgreiche Durchführung späterer Lösungsschritte hing in hohem Maß davon ab, dass zuvor ein bestimmter, „richtiger“ Ansatz erstellt bzw. frühere Schritte der Aufgabe erfolgreich bearbeitet wurden. Dies ist natürlich ein Problem, da man auf diese Art sehr leicht entweder sehr erfolgreich ist, oder gleich sehr viele Punkte verliert, und wenig Möglichkeit hat zwischendurch zu korrigieren.[39] Abhilfe könnte hier schaffen, die Aufgaben jeweils in mehrere Teile zu zerlegen und über mehrere Aufgabenblätter zu verteilen. Eine andere Maßnahme könnte sein, den Aufgaben Fingerübungen (im Stil der in Runde 6 und 7 gestellten) vorangehen zu lassen, die nochmals die feinen Details an minimalen Beispielen exerzieren: Die Fingerübungen aus Runde 6 und 7 wurden nämlich verhältnismäßig erfolgreich bearbeitet.

In einem gewissen Umfang wurden wir auch immer wieder von Defiziten auf der Seite der Teilnehmer überrascht. Die Verteilung der im Sommer 2002 gestellten Aufgaben auf mehrere Blätter könnte diese Überraschung etwas mindern helfen, da dann Betreuer und Dozenten etwas feinergranulares Feedback erhielten.

2. Es schien ein generelles, weit verbreitetes, Unvermögen vorzuliegen, mathematische Notation zu verstehen oder zu schreiben. Damit Hand in Hand ging eine gewisse Distanz zur mathematischen Denkweise: Sowohl der mathematische Funktionsbegriff, als auch die Verwendung von Symbolen für Werte schien vielen Teilnehmern fremd zu sein. Funktionen wurden (kontinuierlich) als Prozeduren missverstanden, symbolische Bezeichnungen als Variablen.

3. Fast alle Teilnehmer taten sich mit der Bildung syntaktischer Abstraktionen schwer – darunter verstehe ich, dass wiederholende Strukturen einen symbolischen (und nach Möglichkeit Bedeutung suggerierenden) Namen erhalten und dann nur noch anhand des Namens zitiert werden.

⇒ Die letzten beiden Probleme könnten vielleicht darauf zurückgeführt werden, dass etwa 40 % der Teilnehmer Nebenfachinformatiker waren, für die genau diese eine Vorlesung (durch einen unvermuteten Schnörkel der Prüfungsordnung) eine Pflichtvorlesung war, was es natürlich verunmöglichte, entsprechend Talent und Neigung, auf andere Vorlesungen der praktischen Informatik auszuweichen. Diese Situation hat sich mittlerweile geändert, so dass man gespannt sein kann, wie ein freiwilliges Publikum in dieser Hinsicht beschaffen sein wird.[40]

[39] Nun ja, eigentlich gab es ja die Helpdesk-Mailingliste, und Hausaufgaben-Hilfe in den Tutorien. Beide Angebote wurden allerdings nur von einer Minderheit wahrgenommen.

[40] In meiner persönlichen Erfahrung muss ich allerdings sagen, dass gerade ein bestimmter Bruchteil der Nebenfächler sehr engagiert und kompetent mitgearbeitet hat. Ausserdem wäre es meines Erachtens seltsam, bezüglich der *mathematischen Grundbildung* für Nebenfachinformatiker die Maßstäbe prinzipiell niedriger anzusetzen. Als *Entschuldigung* für die fast schon katastrophale Situation in dieser Hinsicht taugt also der Verweis auf

Es wäre schön, man könnte sagen, dass die Teilnehmer im Ausgleich für die fehlenden mathematischen (sagen wir: theoretischen) Kenntnisse und Fertigkeiten mehr in der Praxis beheimatet gewesen wären. Leider sprechen die Schwierigkeiten mit der Implementation in Blatt 2 und dem Makefile in Blatt 4 eine andere Sprache. [41]

4. Ein Teil der Probleme, die die Teilnehmer mit den Übungen hatten, lag sicher daran, dass (wie ich mich in jeder Vorlesung selbst überzeugen konnte) die Vorlesung nur teilweise besucht wurde (gezählt 10 bis 20 Besuchern der Vorlesung standen bis zum Ende konstant 30 Abgaben gegenüber) und dass Hilfsangebote (wie die Helpdesk-Mailingliste) nur sehr sparsam, meist in letzte Stunde (in der Nacht vor dem Abgabetermin) wahrgenommen wurden. Dazu passt, dass sich viele Teilnehmer nur sehr schwer selbst helfen konnten: Gerade mal eine Teilnehmerin ging soweit, sich über relevante C++-Konstrukte aus der Literatur zu informieren.

⇒ Gerade bezüglich des Besuches der Vorlesung fielen tatsächlich auch Äusserungen, die Vorlesung hätte ja keinerlei Bedeutung für und keinen Zusammenhang mit der Übung. In der Tat sehe ich das anders, und verstehe nicht ganz, wie das jemand beurteilen will, der die Vorlesung nur zu Anfang besucht hat, aber trotzdem ist hier wohl noch einige Überzeugungsarbeit zu leisten. Ausführliches Besprechen von Fallbeispielen (woher die Zeit nehmen?) in der Vorlesung könnte vielleicht helfen.

5. Zuguterletzt muss auch noch das Problem der Form angesprochen werden, in dem die Abgaben erfolgten: Nicht selten handelte es sich um verschmierte, wild mit einem stumpfen Bleistift (!) bekritzelte Blätter, die zum Teil bereits auf der Rückseite beschriebenes Altpapier waren, manchmal nicht einmal im A4-Format, manchmal ungeordnet, unnummeriert und ohne Überschriften.

6. Mehrfach erwähnt wurden bereits Versuche, weißes Rauschen, Mock-Up oder Plagiate als Lösungen abzugeben.

⇒ Ich möchte gerade auf dem letzten leidigen Thema nicht mehr allzu intensiv herumreiten. Ich will lediglich zu bedenken geben, dass es sich dabei, ebenso wie bei der mangelnden Inanspruchnahme von Hilfsangeboten, dem streckenweise dünnen Besuch der Vorlesung und der Form der Abgaben um ein *kulturelles* Problem handelt, d. h. um eine überlieferte und mittlerweile über Jahrzehnte eingeschliffene Verhaltenstradition, in diesem Fall zwischen zwei Parteien – dem Lehrpersonal der Institution Universität und den Studenten – eine Tradition von „Spicken“, „Kopieren“[42] und Wegschauen, eine Tradition, die mittlerweile sogar notwendig ist, um das System (Betreuungsverhältnisse von 1 : 10 bis 1 : 100 je nach Veranstaltung und Fachbereich an deutschen Universitäten) aufrechtzuerhalten. Es handelt sich *nicht* um einen „Mangel“ am „studentischen Material“, wie einige, die lieber strikter Aussuchen und Auslesen würden, uns gerne verkaufen würden. Umdenken ist hier

den Nebenfächlerstatus nicht.

[41] Darüberhinaus kann ich die Trennung zwischen Theorie und Praxis, wie sie so gerne in Industrie und der akademischen Gemeinschaft aus unterschiedlichen Interessen stilisiert wird, so nicht sehen: Programmieren ohne die Fähigkeit das platonische Ideal (die Idealisierung des betreffenden Prozesses) zu sehen, ohne die Fähigkeit über die dahinterliegenden Prinzipien zu sprechen, führt vielleicht zu gerade mal funktionierender, aber sicher nicht zu sehr anspruchsvoller Software – wenn sie auch gerade deswegen recht umfangreich sein mag. Andersherum ist Theorie ohne die Herausforderungen der Praxis unfruchtbar: Wie man z. B. der entsprechenden Literatur entnehmen kann, haben sich große mathematische Disziplinen (Funktionalanalysis, Stochastik, Maßtheorie ...) aus und mit den Herausforderungen der Praxis entwickelt.

[42] Allein die Existenz dieser – euphemistischen – Termini belegt schon den kulturellen Charakter dieser Handlungsmuster.

angesagt, Förderung statt Auslese, Motivation statt Bestrafung oder Gleichgültigkeit und zwar auf breiter Front, nicht nur in einer einzelnen[43] Vorlesung.

Handeln müssen diejenigen, die die Ressourcen und die Bedingungen soweit kontrollieren, d. h. die Hochschule.

6 Anmerkungen in letzter Minute

Es folgen einige Hinweise, Informationen und Ergänzungen, die quasi erst nach Redaktionsschluss gefunden wurden, und deshalb in die relevanten Text nicht mehr mit eingearbeitet wurden:

- In der Lösungsskizze 7 (TeachSWT@Tü, 2002s) behaupte ich, dass die Literatur wenig zur Bedeutung von Zustands- und Ereignisdiagrammen zu sagen weiß. Inzwischen habe ich das Buch *Object-oriented Modeling and Design* (Rumbaugh u. a., 1991) gefunden. Hierin finden sich auch detaillierte Erklärungen von Zustandsdiagrammen.[44]

  Das genannte Buch ist übrigens ganz allgemein zu empfehlen, auch wenn es sich auf die OMT-Notation und -Methode stützt, welche älter ist als die UML. Der Schwerpunkt liegt auf der Modellierung und Beschreibung von Systemen, nicht auf der automatischen Übersetzbarkeit des erzeugten Modells in ein ausführbares Programm (eine Sünde, die UML meines Erachtens zu stark prägt).

  Rumbaugh u. a. (1991) vertreten ein breites Spektrum von Methoden, z. B. werden auch Datenflussdiagramme eingesetzt, wo dies angemessen ist. Ich halte dies für einen fruchtbareren Zugang, als jedes Problem mit Gewalt durch die Schablone objektorientierter Sichtweise zu pressen („the golden hammer approach“). Aus diesem Grund kann ich das Buch zusammen mit denen von Graham u. a. (2001), Martin und Odell (1992), Meyer (1990) und Liskov und Guttag (2001) als Grundstock für eine Handbibliothek zur Objektorientierung sehr empfehlen.

  Vor vielen anderen Büchern muss dagegen nochmals ausdrücklich gewarnt werden, selbst wenn die Autoren entsprechende akademische Titel tragen, ganz zu schweigen von Büchern, die bereits im Titel suggerieren, um Objektorientierung zu erlernen, genüge es, „mal schnell“ UML anzulesen.

[43] (Sub-) Kulturelle Traditionen ändern sich nicht aufgrund punktueller Konflikte, die wirken im Gegenteil eher identitätsstiftend, denke ich.

[44]

7 Danksagungen und Entschuldigungen

Entschuldigen möchte ich bei den Lesern dieser Texte dafür, dass das Material nicht immer die optimale Reihenfolge und Darlegungsweise gefunden hat. Dies ist bedingt durch die Entstehungsgeschichte, durch die Erklärungen zum Teil in späteren Dokumenten nachgereicht werden, und zwar zu Themen wo wir aus den Erfahrungen mit den zeitlich vorangegangene Übungen Erkenntnisschwierigkeiten vermuteten. In einem Lehrbuch würde vieles sicher in anderer Reihenfolge, d. h. mit den Grundlagen beginnend, dargelegt werden.

Entschuldigen möchte ich mich weiterhin bei den Studenten des Semesters 2002, für die die Übungen zur Vorlesung sicher zum Teil sehr hart waren – wenn ich auch nicht glaube, dass wir unangemessen viel verlangt haben, so haben die Teilnehmer sicher unter dem plötzlich veränderten Anforderungsklima gelitten.

Gleichzeitig möchte ich mich bedanken bei den Teilnehmern der Tutorien, deren Fragen mir zum einen sehr geholfen haben, zu erkennen, wo wir besser und deutlicher erklären müssen, wo unter Umständen Verständnisschwierigkeiten auftreten, und deren zum Teil sehr gute Lösungen und Gedanken mich trotz anderer, pessimistisch stimmender Begegnungen, davon überzeugt haben, dass noch nicht alles verloren ist.

Besonders bedanken möchte ich mich bei meinem Chef, Professor Herbert Klaeren, für die Gelegenheit, endlich einmal das zu tun, was ich schon seit Jahren als unumgänglich angemahnt habe, nämlich die Sicherung von Ergebnissen aus der Lehre des vorangegangenen Semesters, und für seine Geduld beim wiederholten Überschreiten der von mir geplanten und angekündigten Termine. Ich hoffe, das Ergebnis ist es für künftige Softwaretechnikvorlesungen in Tübingen auch wert.

Schließlich möchte ich mich bedanken bei unserem studentischen Mitarbeiter Herrn Caspar Bothmer, für die Hilfe zum einen, zum anderen aber für die vielen fruchtbaren und anwendungsbezogenen fachlichen Gespräche.

Last not least gilt mein Dank Herrn Eckart Göhler, dessen Programmiertalent und dessen Gedanken über Programmierung ich sehr hoch schätze. Ohne den laufenden Diskurs mit ihm hätten viele kluge Gedanken in diesen Texten nicht das Licht der Welt erblickt.

Literatur

[Balzert 1996] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Entwicklung*. Spektrum, Akademischer Verlag, 1996 (Lehrbücher der Informatik). – m. CD-ROM.. – ISBN 3-8274-0042-2

[Bird und Wadler 1997] BIRD, Richard ; WADLER, Philip: *Introduction to functional programming*. Prentice Hall, 1997 (Prentice-Hall International series in computer science). – ISBN 0-13-484197-2, 0-13-484189-1

[efass 2002a] LEYPOLD, M E.: EFASS (An Editor for A Small System) – Beschreibung und VDM-SL-Spezifikation. In: (efass, 2002b). – Quelle (Hauptdokument) `doc/vdm.tex`

[efass 2002b] Wilhelm Schickart Institut (Veranst.): *EFASS (An Editor for A Small System) – Fallstudie zu modularem Entwurf und modularer Programmierung für die Softwaretechnikvorlesung 2002 in Tübingen*. Sand 13, D 72076 Tübingen, 2002. (Materialien zur Softwaretechnik). – Quelltext mit Dokumentation, URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `efass-tue-2002-a.tar.gz`

[efass 2002c] LEYPOLD, M E.: VDM-SL-Beispiele aus dem Editor EFASS. In: (TeachSWT@Tü, 2002k). – Quelle `handouts/vdm-samples-efass.tex`

[Graham u. a. 2001] GRAHAM, Ian ; O'CALLAGHAN, Alan J. ; WILLS, Alan C.: *Object-oriented methods: principles & practice*. Erste Auflage. Addison-Wesley, 2001 (The Addison-Wesley object technology series). – ISBN 0-201-61913-X

[Jones 1992] JONES, Cliff B.: *Systematic software development using VDM*. Zweite Auflage. Prentice Hall, 1992 (Prentice-Hall International series in computer science). – ISBN 0-13-880733-7

[Klaeren 2002] KLAEREN, Prof. Dr. H.: *Vorlesungsmanuskript Softwaretechnik*. Publiziert auf den Webseiten zur Vorlesung. 2002. – URL `http://www-pu.informatik.uni-tuebingen.de/users/klaeren/swt/`

[Leypold 2002] LEYPOLD, M E.: Softwaretechnik 2002 – Konzept zur Durchführung der Übungen. Siehe (TeachSWT@Tü, 2002k). – `essays/konzept.tex`

[Liskov und Guttag 2001] LISKOV, Barbara ; GUTTAG, John: *Program Development in Java: Abstraction, Specification and Object-oriented Design*. Addison-Wesley, 2001. – ISBN 0-201-65768-6

[Martin und Odell 1992] MARTIN, James ; ODELL, James J.: *Object-oriented analysis and design*. Prentice Hall, 1992. – ISBN 0-13-630245-9

[McMenamin und Palmer 1984] MCMENAMIN, Stephen M. ; PALMER, John F.: *Essential systems analysis*. Yourdon Press, 1984 (Yourdon computing series). – ISBN 0-13-287905-0

[Meyer 1990] MEYER, Bertrand: *Object oriented software construction*. Prentice Hall, 1990 (Prentice Hall International series in computer science). – ISBN 0-13-629031-0

[Page-Jones 1988] PAGE-JONES, Meilir: *The practical guide to structured systems design*. Yourdon Press, 1988 (Yourdon Press computing series). – ISBN 0-13-690777-6

[Rumbaugh u. a. 1991] RUMBAUGH, James ; BLAHA, Michael ; PREMERLANI, William ; EDDY, Frederick ; LORENSEN, William: *Object-oriented modeling and design*. Prentice Hall, 1991. – ISBN 0-13-629841-9

[SRC-disas65 2002] Wilhelm Schickart Institut (Veranst.): *Modulschnittstellen (Headerdateien) für einen 8-Bit-Disassembler in C++ – Begleitender Quelltext zur Softwaretechnikvorlesung 2002 in Tübingen*. 2002 (Materialien zur Softwaretechnik). – Als Tape-Archive gepackter Quelltext, URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `disas65-2002-10-22-a.tar.gz`

[SRC-fifo 2002] Wilhelm Schickart Institut (Veranst.): *Quelltext für einen FIFO in C++ – Begleitender Quelltext zur Softwaretechnikvorlesung 2002 in Tübingen*. 2002 (Materialien zur Softwaretechnik). – Als Tape-Archive gepackter Quelltext, URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `FiFo-2002-08-06.tar.gz`

[SRC-prioqueue 2002] Wilhelm Schickart Institut (Veranst.): *Quelltext für eine Prioritätswarteschlange in Modula-2 – Begleitender Quelltext zur Softwaretechnikvorlesung 2002 in Tübingen*. 2002 (Materialien zur Softwaretechnik). – Als Tape-Archive gepackter Quelltext, URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `PrintQueue-2002-05-08.tgz`

[TeachSWT@Tü 2002a] LEYPOLD, M E.: Ablauf der Übungen und Scheinkriterien. In: (TeachSWT@Tü, 2002k). – Quelle `handouts/uebungen-anleitung.tex`

[TeachSWT@Tü 2002b] LEYPOLD, M E.: Beschreibung von Modulen in natürliche Sprache. In: (TeachSWT@Tü, 2002k). – `handouts/module-description.tex`

[TeachSWT@Tü 2002c] LEYPOLD, M E.: Übungsblatt 1: Spezifikation eines FiFo in VDM-SL. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-01.tex`

[TeachSWT@Tü 2002d] LEYPOLD, M E.: Übungsblatt 2: Implementation eines FiFo. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-02.tex`

[TeachSWT@Tü 2002e] LEYPOLD, M E.: Übungsblatt 3: Modellierung eines Heap in VDM-SL. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-03.tex`

[TeachSWT@Tü 2002f] LEYPOLD, M E.: Übungsblatt 4: Theatersitze, Make. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-04.tex`

[TeachSWT@Tü 2002g] LEYPOLD, M E.: Übungsblatt 5: Entwurf eines Disassemblers. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-05.tex`

[TeachSWT@Tü 2002h] LEYPOLD, M E.: Übungsblatt 6: Statische OO-Modelle. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-06.tex`

[TeachSWT@Tü 2002i] LEYPOLD, M E.: Übungsblatt 7: Dynamische Modellierung mit UML. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-07.tex`

[TeachSWT@Tü 2002j] LEYPOLD, M E.: Übungsblatt 8: Strukturierte Analyse, MVC-Pattern. In: (TeachSWT@Tü, 2002k). – `handouts/uebung-08.tex`

[TeachSWT@Tü 2002k] Wilhelm Schickart Institut (Veranst.): *Übungsmaterial, Fallbeispiele und Ergänzungen zur Softwaretechnikvorlesung 2002 in Tübingen*. Sand 13, D 72076 Tübingen, 2002. (Materialien zur Softwaretechnik). – URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `teachswt-tue-2002-a.tar.gz`

[TeachSWT@Tü 2002l] LEYPOLD, M E.: Das EVA-Paradigma. In: (TeachSWT@Tü, 2002k). – `handouts/eva-paradigma.tex`

[TeachSWT@Tü 2002m] LEYPOLD, M E.: Lösungsskizze 1: Spezifikation eines FIFO. In: (TeachSWT@Tü, 2002k). – Quelle `handouts/loesung-01.vdm`

[TeachSWT@Tü 2002n] LEYPOLD, M E.: Lösungsskizze 2: Implementation eines FiFo. In: (TeachSWT@Tü, 2002k). – handouts/loesung-02.vdm

[TeachSWT@Tü 2002o] LEYPOLD, M E.: Lösungsskizze 3: Spezifikation eines Heap in VDM-SL. In: (TeachSWT@Tü, 2002k). – handouts/loesung-03.vdm

[TeachSWT@Tü 2002p] LEYPOLD, M E.: Lösungsskizze 4: Saalbestuhlung, Makefile. In: (TeachSWT@Tü, 2002k). – handouts/loesung-04.vdm

[TeachSWT@Tü 2002q] LEYPOLD, M E.: Lösungsskizze 5: Entwurf eines Disassembler. In: (TeachSWT@Tü, 2002k). – handouts/loesung-05.vdm

[TeachSWT@Tü 2002r] LEYPOLD, M E.: Lösungsskizze 6: Statische OO-Modelle mit UML. In: (TeachSWT@Tü, 2002k). – handouts/loesung-06.tex

[TeachSWT@Tü 2002s] LEYPOLD, M E.: Lösungsskizze 7: Dynamische Modellierung mit UML. In: (TeachSWT@Tü, 2002k). – handouts/uebung-07.tex

[TeachSWT@Tü 2002t] LEYPOLD, M E.: Lösungsskizze 8a: Strukturierte Analyse. In: (TeachSWT@Tü, 2002k). – handouts/loesung-08.tex

[TeachSWT@Tü 2002u] LEYPOLD, M E.: Module und Namensräume. In: (TeachSWT@Tü, 2002k). – Quelle handouts/modules-and-namespaces.tex

[TeachSWT@Tü 2002v] LEYPOLD, M E.: Spezifikation durch Funktionen und die Behandlung von Speicher. In: (TeachSWT@Tü, 2002k). – handouts/functionale-spec-und-speicher.vdm

[TeachSWT@Tü 2002w] LEYPOLD, M E.: Spezifikation eines assoziativen Arrays in VDM-SL. In: (TeachSWT@Tü, 2002k). – Quelle handouts/example-vdm-spec.vdm

[TeachSWT@Tü 2002x] LEYPOLD, M E.: TeachSWT 2002 – Anleitung für den Assistenten und Erfahrungsbericht. In: (TeachSWT@Tü, 2002k). – essays/handbuch.tex

[TeachSWT@Tü 2002y] LEYPOLD, M E.: VDMTools am WSI. In: (TeachSWT@Tü, 2002k). – Quelle handouts/anleitung-vdmtools.tex

[TeachSWT@Tü 2002z] LEYPOLD, M E.: Von der Spezifikation zur Implementation. In: (TeachSWT@Tü, 2002k). – handouts/example-specification-to-implementation.vdm

[Yourdon 1996] YOURDON, Edward: *Mainstream objects*. Prentice Hall, 1996. – ISBN 3-8272-9517-3