Softwareentwicklung — eine Ingenieursdisziplin!(?)

Bernhard K. Aichernig
Peter Lucas

Ordinariat für Softwaretechnologie, TU-Graz

Juni 1998

1 Es ist nicht so einfach wie es scheint

Programmieren macht Spaß und ist auch zunächst relativ leicht zu erlernen. Eine Methode, die rasch zum Ziel führt, ist, die Beispiele in einem guten Lehrbuch nachzuvollziehen. Bald hat man genügend Intuition entwickelt, um aus den vorhandenen Mustern eigene Programme zu entwickeln. Intensive Erfolgserlebnisse stellen sich ein. Die Maschine führt meine Anordnungen minutiös und prompt aus[1]!

Diese Anfangserfolge sind verführerisch. Die Methode, Lösungen kleiner Probleme intuitiv mit den Mitteln einer Programmiersprache in eine exekutierbare Form zu bringen, ist nicht auf die Entwicklung größerer Softwaresysteme skalierbar. Auch die Methode, kleine Programme solange durch Modifikationen und Zusätze wachsen zu lassen, bis sie gegebenen Anforderungen entsprechen, ist bei größeren Vorhaben nicht zielführend, vor allem dann nicht, wenn harte Spezifikationen der zu entwickelnden Software vorliegen. Dies war die Erfahrung in den frühen 60er Jahren als an den einschlägigen Universitätsinstituten an Algol60-Compilern gearbeitet wurde. Eine solche Gruppe war die Forschungsgruppe Wien unter der Leitung von Prof. Heinz Zemanek.

Der Compilerbau eignet sich gut als Beispiel, um den Reifeprozeß einer neuen Technologie zu demonstrieren. Es ist eines der ureigenen Gebiete der Informatik, weil hier nicht auf bekannte Techniken zurückgegriffen werden konnte. Man kann also an diesem Beispiel über einen kurzen Zeitraum hinweg die gesamte Entwicklung einer neuen Softwaretechnologie verfolgen.

Die Pionierphase

Es liegt ein Problem in der Luft. Ist es wirklich notwendig, Algorithmen in kleinste atomare Instruktionen zu zerlegen, um sie dem Computer schmackhaft zu machen? Eine Gruppe von Visionären nimmt sich des Problems an. Die Bezeichnung "Visionär" ist hier mit Bedacht gewählt, denn die Idee höherer Programmiersprachen hatte bis zur allgemeinen Akzeptanz viele Hürden zu überwinden. Es wurde sowohl die Sinnhaftigkeit höherer Programmiersprachen vehement angezweifelt, als auch die technische Durchführbarkeit der automatischen Übersetzung in die Maschinensprache, vom psychologischen Widerstand der Programmierer ganz zu schweigen, die fürchteten, ihre Jobs zu verlieren. Die ersten Compiler entstanden in einem Zusammenspiel des Suchens nach Sprachkonstrukten und Übersetzungsmethoden. Fortran, das Resultat eines Projektes der IBM unter der Leitung von John Backus, war ein Resultat dieses Prozesses und letztlich ein Erfolg, der bis heute anhält. Das Projekt wurde von der Firma unterstützt, weil ein Manager an die Kreativität von John Backus glaubte. Es war nicht das Resultat einer Marktanalyse, folgte keinem genormten Vorgehensmodell, keiner genauen Spezifikation, es gab nur das Problem und die Idee zur Lösung. Was treibt ein solches Projekt voran? Die Motivation auf Seiten der Erfinder ist das Erlebnis der Innovation und auf Seiten des Managements, der Wille den Konkurenzkampf des Unternehmens zu bestehen, aus dem die Notwendigkeit solche Innovation zu fördern folgt.

Die-Sturm-und-Drang-Periode

Ist die Realisierbarkeit gezeigt und ist Aussicht auf eine entsprechende industrielle Relevanz, folgt eine Periode gesteigerter Aktivitäten in den Forschungslabors der Industrie und der Universitäten. Algol60 entstand, nachdem die technische Möglichkeit bereits nachgewiesen war. Eine Gruppe von Fachleuten, teils Professoren, teils aus der Industrie, wollte eine Standardsprache schaffen, die so genau und maschinenunabhängig definiert ist, daß Programme zwischen Institutionen ausgetauscht werden können und ohne diese zu adaptieren, auf einer beliebigen Maschine exekutiert werden können.

Das Ziel war ehrgeizig. Die Sprache sollte wirklich maschinenunabhängig sein[2]. Diese Zielsetzung macht eine abstrakte

[1] Bei größeren Programmen wird dies dann gelegentlich zur schmerzlichen Erfahrung, zur Reflexion der eigenen Unzulänglichkeit.

[2] In der heute üblichen Terminologie würde man sagen "plattformunabhängig". DieZielsetzung von Java ist sehr ähnlich.

Spezifikation der Sprache notwendig, die von Implementierungen auf verschiedensten Maschinen erfüllt werden kann.

Es wurde versucht Entwurfsprinzipien zu realisieren, insbesondere Orthogonalität, d.h. möglichst komplementäre Konzepte, die frei kombinierbar sind. Dabei wurde die Sprache keineswegs auf jene Konstruktionen eingeschränkt, deren mechanische Übersetzbarkeit, bekannt war.

Die Situation in dieser Periode war sehr verschieden, von der Zeit der Pioniere.

1. Es lag eine Spezifikation der Sprache vor, der Algol60-Bericht. D.h. wer beitragen wollte, mußte sich an diese Spezifikation halten.

2. Die Idee einer höheren Programmiersprache war zwar keineswegs allgemein als nützlich akzeptiert, die Akzeptanz war aber breit genug, um ein neues Forschungsgebiet zu etablieren.

3. Die Herausforderungen der neuen Sprache wurden weithin angenommen und regten viele Forschungsprojekte an.

Die Spezifikation war leider nicht so eindeutig, wie man dies zunächst glaubte. Aus dieser Erfahrung entstand der Wunsch, die Methoden der Sprachdefinition zu verbessern. Auch die Implementierung der Sprache stellte sich als nicht trivial heraus, was ebenfalls zu regen Forschungsaktivitäten führte.

Die Mathematisierung des Gebietes

Ein wesentlicher Beitrag von Algol60 war eine Methode zur Definition der Syntax von Sprachen, die unter der Bezeichnung BNF (Backus Normalform) bekannt wurde. Diese Methode gab den Anstoß zur Entwicklung der mathematischen Grundlagen des Compilerbaus. Unmittelbar nach der Veröffentlichung dieser formalen Definitionsmethode entstand eine Theorie der syntaktischen und lexikalischen Analyse. Zunächst war es möglich, auf Grund dieser Theorie die Phasen eines Compilers, die der lexikalischen und syntaktischen Analyse dienen, systematisch zu konstruieren. Dies führte zu sehr eleganten, allgemeinen Lösungen. Vergleichsweise erscheinen die zunächst intuitiv und für den Spezialfall von Fortran entwickelten, frühen Compiler unbeholfen. Damit soll die Pionierleistung von Fortran in keiner Weise geschmälert werden. Es war ja auch John Backus, der Vater von Fortran, der nach der Entwicklung von Fortran, aus dieser Erfahrung heraus, die Methode zur Formalisierung der Syntax vorschlug. Auf Grund dieser Theorie war es später sogar möglich, die Programme zur lexikalischen und syntaktischen Analyse automatisch aus einer gegebenen Definition der Syntax einer Sprache zu generieren. Keine Programmiersprache wird heute kreiert ohne die Syntax in BNF oder einer ihrer Variationen zu definieren. Keine neue Compilerentwicklung würde heute auf diese theoretischen Grundlagen und deren praktische Anwendungen verzichten. Man würde auch keinem Softwareingenieur, der dieses Spezialwissen nicht besitzt, eine solche Entwicklung anvertrauen.

In der Definition von Algol60 wurde auch der Versuch gemacht, die Semantik (Bedeutung) der Sprache streng festzulegen. Dies geschah zum Teil durch geeignete Ersetzungsregeln und Äquivalenzen. Die Definition der Semantik war aber weit weniger rigoros als die Definition der Syntax. Zur Enttäuschung der Urheber von Algol60 wurden auch nach Fertigstellung der Referenzdokumentation einige Unklarheiten, Mehrdeutigkeiten und Fehler entdeckt, die eine revidierte Version notwendig machten. Die Definition der Semantik einer Sprache war auch ein viel schwierigeres Problem. Erst viele Jahre nach dem ersten Algol60 Bericht zeichnete sich eine kohärente und brauchbare theoretische Grundlage und Definitionsmethode zur Semantik von Programmiersprachen ab. Die Forschungsgruppe um Zemanek, die 1961 zum IBM Labor Wien avancierte, war an dieser Formalisierung der Semantik und deren Anwendung im Compilerbau wesentlich beteiligt. Die aus dem Wiener Labor hervorgegangene Methodik ist heute unter dem Namen VDM (Vienna Development Method) bekannt. Obwohl VDM ursprünglich, wie oben angedeutet wurde, aus der Sprachdefinition und dem Compilerbau hervorging, ist VDM heute nicht auf dieses enge Gebiet beschränkt. Viele der Prinzipien, die zunächst nur für den Compilerbau gedacht waren, sind auf Softwareentwicklung im allgemeinen anwendbar. Im Abschnitt 3 wird VDM an einem Beispiel demonstriert.

Das etablierte Gebiet

Heute ist der Compilerbau ein relativ reifes Gebiet. Mindestens für konventionelle Programmiersprachen gibt es etablierte Methoden der Definition und Implementierung. Lexikalische Analyse, Syntaxanalyse, Optimierung, Registerzuweisung, Codegenerierung, besitzen ein Arsenal an theoretisch fundierten in der Praxis bewährten Methoden.

Compilerbau kann man lehren und lernen. Es ist mehr oder weniger klar, was in einer Lehrveranstaltung über Compilerbau vorkommen muß. Es ist klar, was die Qualifikationen eines auf Compilerbau spezialisierten Softwareingenieurs sind.

Es soll hier nicht der Eindruck erweckt werden, daß das Gebiet der Programmiersprachen und des Compilerbaus abgeschlossen sind. Es entstehen nach wie vor neue, konventionelle und unkonventionelle Programmiersprachen. Es werden die bestehenden Compilermethoden verbessert. Neue Hardwarearchitekturen erfordern entsprechende Anpassungen der Compilertechniken. Wie auf den meisten technischen Gebieten gibt es Fortschritte. Aber gemessen an den 60er-Jahren ist es ruhiger geworden.

Das bedeutet nicht, daß dieser Zustand so bleiben muß. Revolutionen sind möglich.

2 Vom Hacker zum Software-Ingenieur

In welcher Phase der Entwicklung steht Software-Engineering?

Der Software-Ingenieur unterscheidet sich von Hacker etwa so, wie sich der Elektrotechniker vom Radiobastler unterscheidet. Dabei soll hier weder der Radiobastler noch der Hacker abgewertet werden. Der Ingenieur benützt wissenschaftlich fundierte Methoden und hat die zum Verständnis dieser Methoden erforderliche Grundlagenausbildung.

In der Durchführung eines Projekts hält sich der Ingenieur an etablierte oder genormte Vorgangsweisen. Im Falle des Versagens eines Projektes oder Produktes, muß die angewandte Vorgehensweise und die angewandten Konstruktionsmethoden nachvollziehbar sein. Auch müssen die Kosten eines Vorhabens und der Fertigstellungstermin abschätzbar sein.

Software-Engineering ist derzeit mehr Wunsch als Realität von einigen Teilgebieten abgesehen, fehlen die genormten Vorgehensmodelle, die etablierten Methoden und Mittel für eine zuverlässige Planung. Software-Engineering insgesamt ist daher sicher nicht als etabliertes Gebiet und ausgereifte Ingenieursdisziplin zu sehen. Dessen ungeachtet werden aber große Softwaresysteme gebaut. Die Konsequenzen zeigen sich in der Ausbeute und in manchmal spektakulärem Versagen. Studien haben gezeigt, daß drei-viertel aller großen Systeme entweder nicht wie beabsichtigt funktionieren oder überhaupt nicht benützt werden. Einige der Spektakulären Versager – weithin publiziert – ist das Gepäckssystem des Flughafens von Denver, das neun Monate nachdem der Flughafen hätte eröffnet werden sollen, immer noch nicht funktionierte. Ein anderes Beispiel ist das neue Flugverkehrskontrollsystem, das in den US geplant war und nach vielen Millionen Dollar Entwicklungskosten großteils wieder aufgegeben wurde.

Mathematische Basis

Es ist nicht leicht vorherzusehen, wohin die Entwicklung die im entstehen begriffene Disziplin des Software-Engineerings treiben wird. Die mathematische Basis, die sich abzeichnet, ist verschieden von den traditionellen Ingenieursdisziplinen. Es ist hauptsächlich die sogenannte diskrete Mathematik, die gebraucht wird. Dazu rechnet man: die Aussagen- und Prädikatenlogik, die Mengenlehre, abstrakte Algebren, den Lambdakalkül, usw.

Wie schon am Beispiel der Programmiersprachen diskutiert wurde, ist eines der Hauptprobleme im Entwurf von Softwaresystemen, die Spezifikation des Systems unabhängig von einer speziellen Implementierung. D.h. es muß spezifiziert werden, was das System leisten soll, ohne im Detail zu sagen, wie der Effekt erreicht wird. Eine Methode dies zu bewerkstelligen, sind die sogenannten Vor- und Nachbedingungen (pre- und post-condition). Die Vorbedingung schränkt den Anfangszustand des Systems ein. Die Nachbedingung definiert die Relation von Anfangs und Endzustand. Zur Formulierung dieser Bedingungen werden Prädikatlogische Ausdrücke verwendet.

VDM ist eine modellorientierte Methode. Ein System wird modelliert, indem man die möglichen Zustände des Systems, möglichst abstrakt in Form mathematischer Objekte (Mengen, Abbildungen, usw.) darstellt. Die auf diese Zustände anwendbaren Funktionen werden dann entweder implizit mit Hilfe von Vor- und Nachbedingungen spezifiziert oder explizit definiert.

Der Lambdakalkalkül ist eine ausgezeichnete Basis, zur theoretischen Untersuchung der Semantik von Programmiersprachen. Dies wurde schon in den frühen 60er-Jahren erkannt. Vor allem funktionale Sprachen sind eigentlich nur notationelle Varianten des Lambdakalküls.

Genormte Vorgangsweisen

Unter Vorgangsweise versteht man die Phasen, die eine Softwareentwicklung durchlaufen muß. Üblicherweise beginnt dieser Prozeß mit einem Anforderungsdokument, durchläuft dann einige Entwurfsphasen bis zur Kodierung, einige Testphasen bis zur Übergabe des Produktes, gefolgt von der Wartung. Die Schwierigkeit des derzeitigen Standes liegt nicht darin, daß es keine standardisierten Vorgehensmodelle gibt. Es gibt zu viele. Die großen Firmen und Organisationen haben jeweils ihre eigenen Normen. Als Beispiel sei der ESA (European Space Agency) erwähnt. Dieser Standard wie auch andere legt fest, welche Dokumente in jeder Phase zu erstellen sind und welche Prüfungen durchzuführen sind, damit die nächste Phase in Angriff genommen werden kann. Die Darstellungsmethoden und Prüfmethoden werden aber großteils dem Entwickler überlassen. Das ist nicht überraschend, weil ja über die Darstellung des Produktes in den verschiedenen Entwurfsphasen kein Konsens besteht. VDM ist ein Ansatz zur Mathematisierung der Darstellungen, Prüfungen und der Entwicklungsschritte.

3 Mathematisch basiertes Vorgehen

Definieren wir ein Problem als das Delta zwischen einem Ist-Zustand und einem gewünschten Soll-Zustand, so ist die Aufgabe des Ingenieurs dieses Delta unter Einsatz von Technik gegen Null streben zu lassen. Genormte Vorgehensmodelle weisen dabei den Weg von einer Problemstellung zu einer technischen Lösung und grenzen die Ingenieursdisziplinen so von der reinen Wissenschaft ab. Doch ist es wiederum die Anwendung von wissenschaftlicher Erkenntnis und Mathematik, welche den Ingenieur vom Bastler unterscheiden.

Aus oben Gesagtem ergibt sich die Notwendigkeit die Softwareentwicklung und deren Vorgehensmodelle auf ein mathematisches Fundament zu stellen. Um dies zu erreichen, ist es schon in der Ausbildung des Software-Ingenieurs erforderlich,

Es ist die Steuerungslogik eines Liftes in einem Gebäude mit zehn Stockwerken zu implementieren. Der Lift soll sich nach folgenden Regeln verhalten:

R1 Der Lift hat eine Menge von Knöpfen in der Kabine, für jeden Stock einen. Diese leuchten, wenn sie gedrückt werden und veranlassen den Lift im entsprechenden Stockwerk anzuhalten. Sobald der Lift in einem Stockwerk hält, erlischt der entsprechende Knopf.

R2 Jedes Stockwerk, außer das Erdgeschoß und das Letzte, hat zwei Knöpfe, einen um eine Anforderung hochzufahren und einen um eine Anforderung hinunterzufahren zu aktivieren. Die Knöpfe leuchten, wenn sie gedrückt werden. Die Knöpfe erlöschen, wenn der Lift im Stockwerk hält und dann entweder in die gewünschte Richtung fährt oder aber ohne weitere Anforderungen verweilt.

R3 Hat ein Lift keine Anforderungen, bleibt er mit geschlossenen Türen im Stockwerk und wartet auf Anforderungen.

R4 Alle von Stockwerken getätigten Anforderungen sollen behandelt werden, wobei alle Stockwerke die gleiche Priorität haben.

R5 Alle im Lift getätigten Anforderungen sollen behandelt werden, wobei alle Stockwerke die gleiche Priorität haben.

Abbildung 1: User Requirements einer Liftsteuerung

die Barrieren zwischen Theorie und Praxis aufzuheben. Wie oben gezeigt, kann hier der Compilerbau als Vorbild bezeichnet werden, wo die Theorie zu Werkzeugen führte, die das Entwickeln eines Compilers von einer Wissenschaft zur Ingenieursdisziplin machten: Noch in den 70er Jahren konnte man durch das Entwickeln eines Compilers zum *doctor technicae* graduieren.

Leider hat dieses Verschmelzen angewandter Mathematik und methodischen Vorgehens in der allgemeinen Softwareentwicklung nur im begrenzten Maße Einzug gehalten, obwohl die notwendigen Techniken schon lange bekannt sind.

Im folgenden wird ein Einblick in die Anwendungsmöglichkeiten diskreter Mathematik gegeben, welche die Qualität des Entwicklungsprozesses erheblich steigert. Das Vorgehensmodell der European Space Agency (ESA) [6] und ein kleines Beispiel einer Liftsteuerung sollen dem Diskurs zugrunde liegen.

Ein Problem wird formuliert

In der User Requirements Phase am Beginn eines Software-Projekts steht die Idee eines Problems, welches mittels Computer gelöst werden soll. Diese Idee wird verfeinert und im Anforderungsdokument festgehalten. Leider enthält das in Prosa gehaltene Dokument oft Mehrdeutigkeiten, Widerspruech oder unvollständige Angaben. Daher ist es die Aufgabe der Ingenieure, dem Auftraggeber bei der präzisen Erstellung der Anforderungen zu helfen und etwaige Interpretationsfragen zu klären. Dies wird durch das Erstellen eines logischen Modells der Software-Requirements in der nächsten Phase des Projekts erreicht. Abbildung 1 zeigt die User Requirements einer Liftsteuerung, welche im folgenden als Beispiel dienen soll.

Ein Problem wird modelliert

In der Software Requirements (SR) Phase werden die User Requirements analysiert und die Anforderungen an die Software so komplett, konsistent und korrekt wie möglich definiert. Um dies zu erreichen, empfiehlt der ESA-Standard ein logisches Modell zu erstellen. Nun geben populäre graphische Modellierungsmethoden einen guten Überblick der Abhängigkeiten im Modell, doch fehlt ihnen meist die eindeutige Semantik. Das hat zur Folge, daß die Konsistenzprüfung der graphischen Modelle nicht möglich bzw. mechanisierbar ist. Wie in anderen Ingenieursdisziplinen üblich, sollten folglich mathematische Modelle [5] die Basis einer Spezifikation bilden, und graphische Notationen der Veranschaulichung sowie der Kommunikation mit dem Auftraggeber dienen. Für den Ingenieur, der aus dem Modell eine Implementierung ableiten muß, gilt daher der Leitsatz:

> Ein Bild sagt mehr als tausend Worte.
> Eine Formel mehr als tausend Bilder.

Eine dem Denken des SW-Ingenieurs sehr entsprechende Methode der mathematischen Modellbildung, ist das Spezifizieren einer abstrakten Zustandsmaschine (*Abstract State Machine*) [1, 8]. Dabei wird ein System durch seinen Zustand, seine Zustandsinvariante und die Operationen auf dem Zustand abstrakt beschrieben. Die Abstraktion des Zustandsmodells wird durch das Verwenden von mathematischen Objekten wie Mengen, Folgen und Tabellen (endliche Abbildungen) erreicht. Die Operationen definiert man als Relation zwischen Eingabeparametern, Ausgabeparametern, dem alten und dem neuen Zustand.

Statisches Modell

Abbildung 2 zeigt das Zustandsmodell der Liftsteuerung. Man beginnt mit der Definition der zur Modellierung benötigten Typen. Der Typ *Direction* definiert die zwei Bewegungsrichtungen des Liftes, *UP* oder *DOWN*. Stockwerke (*Floor*) werden durch natürliche Zahlen repräsentiert. Der Anforderung *R2* entsprechend, werden die Rufknöpfe *InReq* als Relation zwischen dem Stockwerk und der gewünschten Fahrtrichtung dargestellt.

Wie in Programmen üblich, wird der Zustand des Lifts als eine endliche Menge von Variablen repräsentiert, wobei hier abstrakte Datentypen aus der Mathematik, dem statischen Modell eine eindeutige und implementierungsunabhängige Semantik geben:

Ein Bool'scher Wert *moving* gibt an, ob der Aufzug fährt (*true*) oder steht. Die zweite Variable *actfloor* definiert das aktuelle Stockwerk in dem sich die Kabine befindet. Die Richtung in die der Aufzug unterwegs ist wird durch *dir* repräsentiert. Und schließlich die Mengen der Knöpfe des Liftes in den Stockwerken (*ins*) und in der Aufzugskabine (*outs*). Wäre eine Ordnung der Liftknöpfe (Prioritäten) gefordert, müßte

types

$$
\begin{array}{ll}
1.0 & \mathit{Direction} = \text{UP} \mid \text{DOWN}; \\
2.0 & \mathit{Floor} = \mathbb{N}; \\
3.0 & \mathit{InReq} = \mathit{Floor} \times \mathit{Direction} \\
4.0 & \mathsf{state}\ \mathit{Lift}\ \mathsf{of} \\
.1 & \quad \mathit{moving} : \mathbb{B} \\
.2 & \quad \mathit{actfloor} : \mathit{Floor} \\
.3 & \quad \mathit{dir} : \mathit{Direction} \\
.4 & \quad \mathit{ins} : \mathit{InReq}\text{-}\mathsf{set} \\
.5 & \quad \mathit{outs} : \mathit{Floor}\text{-}\mathsf{set} \\
.6 & \mathsf{end}
\end{array}
$$

Abbildung 2: Zustandsvariablen einer Liftsteuerung.

man die Knöpfe als Folgen modellieren. Diese kleine Überlegung zeigt den Vorteil eines mathematischen Modells auf: Der Vorgang der Modellbildung, hier die Auswahl der geeigneten Repräsentation, wirft konkrete Fragen an die Anforderungen der Software auf, die, wenn nicht in den User-Requirements beantwortet, mit dem Auftraggeber abgeklärt werden müssen.

Der kritische Leser wird einwenden, daß unser bisheriges Zustandsmodell sehr allgemein ist und den Variablendefinitionen in Programmiersprachen sehr ähnlich: Stockwerke werden z.B. als unendliche Menge der natürlichen Zahlen definiert, die Einschränkung der Knöpfe im ersten und letztem Stockwerk in **R2** bleibt unberücksichtigt, u.s.w. Diese fehlenden Bedingungen können mittels Prädikaten, also Funktionen auf Wahrheitswerte, formuliert werden. Da diese Prädikate die immer geltenden Eigenschaften der Daten in einem System definieren, werden sie Dateninvarianten genannt. Abbildung 3 zeigt die Dateninvarianten des Liftes:

Die Invariante der Daten vom Typ *Floor* ist eine Funktion der Stockwerke auf einen Wahrheitswert. Ein Stockwerk soll in der Menge der natürlichen Zahlen zwischen den Konstanten *ground* und *top* liegen. Die Invariante *inv-InReq* formuliert die Ausnahme, daß das Erdgeschoß (es gibt keinen Keller) und der letzte Stock jeweils nur einen Knopf haben. Die Invariante des Lifts definiert die geltende Beziehung zwischen den Zustandsvariablen: Ein stehender Lift impliziert, daß das aktuelle Stockwerk weder in der Menge der Ausstiegswünsche, noch in den Einstiegswünschen enthalten ist.

values

$$
\begin{array}{ll}
5.0 & \mathit{ground} = 0; \\
6.0 & \mathit{top} = 10
\end{array}
$$

functions

$$
\begin{array}{ll}
7.0 & \mathsf{inv}\text{-}\mathit{Floor} : \mathit{Floor} \to \mathbb{B} \\
.1 & \mathsf{inv}\text{-}\mathit{Floor}\,(f) \triangleq \\
.2 & \quad f \in \{\mathit{ground}, \ldots, \mathit{top}\}; \\
8.0 & \mathsf{inv}\text{-}\mathit{InReq} : \mathit{InReq} \to \mathbb{B} \\
.1 & \mathsf{inv}\text{-}\mathit{InReq}\,(\mathit{ir}) \triangleq \\
.2 & \quad \mathit{ir} \neq \mathsf{mk}\text{-}(\mathit{ground}, \text{DOWN}) \wedge \\
.3 & \quad \mathit{ir} \neq \mathsf{mk}\text{-}(\mathit{top}, \text{UP}); \\
9.0 & \mathsf{inv}\text{-}\mathit{Lift} : \mathit{Lift} \to \mathbb{B} \\
.1 & \mathsf{inv}\text{-}\mathit{Lift}\,(l) \triangleq \\
.2 & \quad \neg\, l.\mathit{moving} \;\Rightarrow\; (l.\mathit{actfloor} \notin l.\mathit{outs} \wedge \\
.3 & \quad \{\mathsf{mk}\text{-}(l.\mathit{actfloor}, \text{UP}), \mathsf{mk}\text{-}(l.\mathit{actfloor}, \text{DOWN})\} \cap l.\mathit{ins} = \\
.4 & \quad \{\})
\end{array}
$$

Abbildung 3: Dateninvarianten einer Liftsteuerung.

Dynamisches Modell

Für die abstrakte Spezifikation der Funktionalität ist es wesentlich, nicht das *Wie* sondern das *Was* zu spezifizieren. Eine Möglichkeit ist die Verwendung impliziter Spezifikationen, welche die Wirkung einer Operation als Relation zwischen Input und Output beschreiben.

Das Verhalten des Lifts definiert sich durch das Drücken der Knöpfe (Ereignisse) und das Fahren selbst. In Abbildung 4 ist das Drücken der Knöpfe in der Aufzugskabine definiert. Die Operation *request-floor* nimmt das gewünschte Stockwerk als Parameter, greift auf die (*ext*ernen) Zustandsvariablen *moving* und *actfloor* lesend zu und verändert den Zustand von *outs*. Eine Vorbedingung (pre-condition) definiert jene Bedingung, welche vor Ausführung der Operation gelten muß, damit diese definiert ist. Hier soll es nicht möglich sein, ein Stockwerk zu wählen, in dem der Lift gerade hält. Die Nachbedingung beschreibt **was** nach der Ausführung gelten muß: In diesem Fall soll das gewünschte Stockwerk der Menge der Stockwerke in denen angehalten werden soll hinzugefügt werden. Der Haken über der Zustandsvariable kennzeichnet den Zustand vor dem Ausführen von *request-floor*.

Das Fahren des Liftes wird durch die Kommandos an den Antrieb modelliert. Abbildung 5 zeigt exemplarisch die Definition des Anhaltens im Aufwärtsfahren. Zwei Prädikate des Liftes helfen die Vorbedingung der Operation zu definieren: *attracted-up* ist wahr, wenn die Menge der höheren Stockwerke, in denen anzuhalten ist, ungleich der leeren Menge ist.

operations

$$
\begin{array}{ll}
10.0 & \mathit{request\text{-}floor}\,(f : \mathit{Floor}) \\
.1 & \mathsf{ext}\ \mathsf{rd}\ \mathit{moving} \\
.2 & \quad\ \mathsf{rd}\ \mathit{actfloor} \\
.3 & \quad\ \mathsf{wr}\ \mathit{outs} \\
.4 & \mathsf{pre}\ \neg\,\mathit{moving} \;\Rightarrow\; \mathit{actfloor} \neq f \\
.5 & \mathsf{post}\ \mathit{outs} = \overleftarrow{\mathit{outs}} \cup \{f\}\ ;
\end{array}
$$

Abbildung 4: Drücken der Knöpfe im Lift.

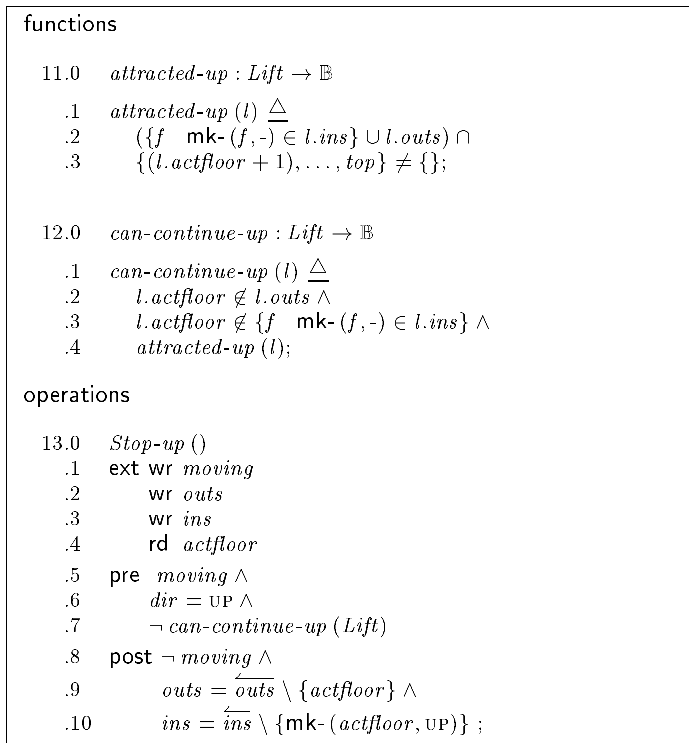

Abbildung 5: Anhalten des Lifts im Aufwärtsfahren.

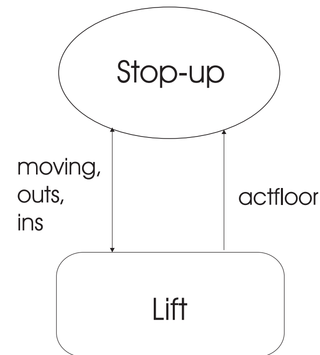

Abbildung 6: Datenflußdiagramm des Anhaltens.

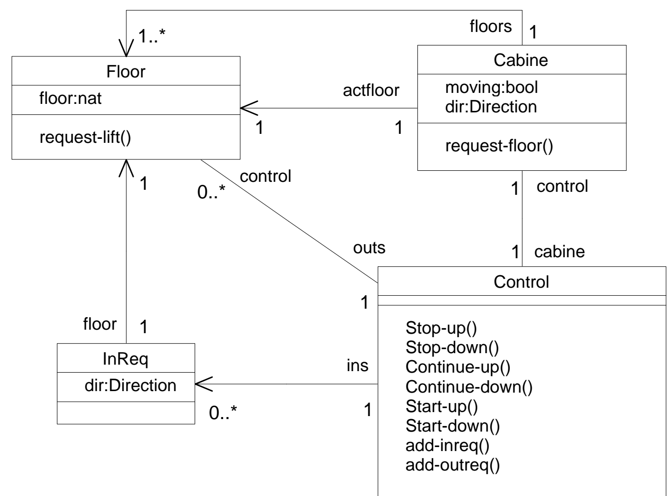

Abbildung 7: Klassendiagramm des Liftes in UML.

Die Funktion *can-continue-up* definiert ob der Lift seine Fahrt ohne im aktuellen Stockwerk anzuhalten nach oben fortsetzten kann. Es darf dazu weder ein Ausstiegs- noch Einstiegswunsch für das aktuelle Stockwerk vorliegen und *attracted-up* muß gelten. Die Operation *Stop-up* läßt sich dann sehr einfach modellieren.

Ein Problem wird visualisiert

Es ist wichtig zu betonen, daß mathematische Notationen keine Alternative, sondern eine Ergänzung zu 'klassischen' Methoden sind. Graphische Darstellungen helfen einen Überblick über größere Spezifikationen zu gewinnen und dienen der Kommunikation mit dem Kunden. Als Beispiel einer möglichen Kombination mit Strukturierter Analyse [12] zeigt Abbildung 6 die Operation *stop-up* in Form eines Datenflußdiagramms. Die Beschriftung der Datenflußpfeile verweisen auf das mathematische Modell und geben so dem Diagramm eine eindeutige Semantik.

Ein Problem wird auf-, ge- und verteilt

Sind die Anforderungen an die Software modelliert, so gilt es in der Architektur-Design Phase das Problem zu strukturieren und in logische bzw. physische Komponenten aufzuteilen. Ist das Problem sehr Komplex, so kann die Aufteilung bereits in der ersten Phase der Modellbildung stattfinden.

Designed man objektorientiert, wird der Systemzustand und die Operationen auf Klassen aufgeteilt. Abbildung 7 zeigt ein mögliches objektorientiertes Design des Liftes in UML. Die Pfeile beschreiben die Assoziationen zwischen den Klassen [7] . Die impliziten Spezifikationen der Operationen einer Klasse bilden eine präzise und vollständige Schnittstellenbeschreibung der gekapselten Datentypen, was die (Wieder-) Verwendbarkeit erheblich erleichtert und sicherer macht. Statt zur Klärung von Detailfragen die Implementierung verstehen zu müssen, was einem gedanklichen Simulieren der Zustandsmaschine entspricht, genügt ein Blick auf die Signaturen, Vor- und Nachbedingungen der Operationen (Methoden).

Ist eine Klasse korrekt implementiert, so verhindert die Kapselung des Zustands eine Verletzung der Dateninvariante, wenn die Vorbedingungen der Methoden eingehalten werden. Für den Fall einer nicht erfüllten Vorbedingung, sollte man in dieser Prozeßphase eine entsprechende *Exception-*

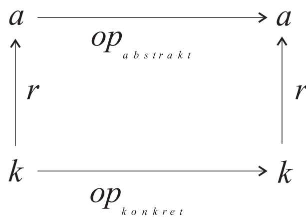

Abbildung 8: Abstraktion als kommutierendes Diagramm.

Generierung definieren (defensive Programmierung).

Detailiertes Design und Implementierung

Die Entwicklung von Software ist ein Prozeß der Konkretisierung. Ausgehend von einer Problemstellung wird diese zuerst abstrakt spezifiziert und schrittweise verfeinert, bis man schließlich zu einer konkreten Implementierung gelangt. Eine Verfeinerung kann daher der Schritt von einer impliziten zu einer expliziten Spezifikation sein, oder die Anwendung effizienterer Datenstrukturen bzw. Algorithmen oder einfach das Hinzufügen von Informationen zu einem Modell.

Es wird den Leser nicht überraschen, daß auch dieses Verfeinern von Modellen hin zu einer Implementierung mathematisch formulierbar ist. In unserem mathematisch basierten Vorgehen sind sowohl Spezifikationen als auch Implementationen mathematische Objekte, genauer gesagt, Algebren: Die verwendeten Datentypen stellen die Grundbereiche (Sorten) dar, und Operationen, als Funktion von einem Zustand auf einen anderen Zustand interpretiert, entsprechen den Operatoren der Algebra. Die Beziehung zwischen abstrakten und konkreteren Modellen läßt sich als Abbildung von konkreten Operationen auf abstraktere Operationen darstellen. Dies wiederum erlaubt es die korrekte Verfeinerung (Implementierung) eines abstrakten Modells als Homomorphisums zu definieren:

Es sei $op_a : a \to a$ eine abstrakte Operation von a nach a, und $op_k : k \to k$ eine konkrete Operation von k nach k. Weiters existiere eine Funktion $r : k \to a$. Dann ist op_k eine korrekte Verfeinerung von op_a, wenn r ein Homomorphismus ist, also gilt:

$$\forall k \cdot op_a(r(k)) = r(op_k(k))$$

Abbildung 8 zeigt diese Beziehung als kommutierendes Diagramm. Die Anwendung obiger Beziehung ist zweierlei: (1) Durch die Definition von r läßt sich die korrekte Verfeinerung von der Spezifikation zur Implementierung schrittweise formal beweisen. (2) Moderne Werkzeuge implementieren die Inverse von r als Code-Generator von expliziten, implementierungsnahen Spezifikationen nach Programmiersprachen, wie ADA, C++ oder Java.

Validierung und Verifikation

Das Verwenden mathematischer Modelle ermöglicht die automatisierte Validierung und Verifikation in den frühen Designphasen. Dies hat zur Folge, daß inkonsistente Requirements und Designfehler bereits sehr früh entdeckt werden, was den Aufwand der Modellbildung rechtfertigt: Spät entdeckte Fehler sind die teuersten!

Die mathematische Notation erlaubt es formale Beweise über Systemeigenschaften zu führen, wobei heutige Beweiswerkzeuge (Theorem Prover) bereits einen hohen Grad an Automatisierung erreicht haben [11, 9]. Dabei wird das Modell axiomatisiert, wobei die Vorbedingungen den Prämissen und die Nachbedingungen den Konklusionen logischer Schlußregeln entsprechen. Die zu beweisenden Eigenschaften werden dann als Theoreme formuliert und mit den gegebenen Axiomen bewiesen [3, 2].

Automatisches Black-Box Testen wird erst durch das mathematische Modell möglich: Tester nennen daher das Formalisieren eines Modells, es testbar machen [10]. Hier beschreiben Vorbedingungen und Dateninvarianten die gültigen Testfälle, und Nachbedingungen bilden das Orakel, welches über den Erfolg eines Tests Auskunft gibt.

4 Quo vadis?

In diesem Artikel wurde am Beispiel des Compilerbaus der Reifeprozeß einer Ingenieursdisziplin veranschaulicht. Weiters wurde die Anwendung der im Compilerbau wurzelnden mathematischen Techniken im Softwareprozeß skizziert. Es stellt sich nun die Frage, wo die Softwareentwicklung heute steht und wohin sie sich entwickelt. Den ersten Teil der Frage beantworten wir mit: “Am Übergang der Sturm-und-Drang Periode zur Mathematisierung”. Die Frage nach dem Wohin, ist nicht die Frage ob es zu einer Mathematisierung kommt, sondern die Frage welche Techniken sich durchsetzen werden. Dies wird von dem Faktorentripel (verfügbare Werkzeuge, Ausbildung, Akzeptanz), eine Relation, abhängen. Die hier gezeigte Notation und Methodik wird auf jeden Fall an diesem Wettstreit der Methoden teilnehmen: Die *Vienna Development Method* (VDM).

Literatur

[1] J.-R. Abrial. *The B-Book, Assigning programs to meanings.* Cambridge University Press, 1996. ISBN 0521-49619-5 (hardback).

[2] Bernhard K. Aichernig and Peter Gorm Larsen. A proof obligation generator for VDM-SL. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer*

Science, Graz, Austria, September 1997. Technical University Graz, Springer.

[3] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[4] Formal Methods Europe. FME Hub. WWW at URL http://www.cs.tcd.ie/FME/.

[5] John S. Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development.* Cambridge University Press, 1998. ISBN 0-521-62348-0.

[6] ESA Board for Software Standardisation and Control (BSSC). Esa software enineering standards. Technical Report ESA PSS05-0 Issue 2, European Space Agency, Februar 1991.

[7] Martin Fowler and Kendal Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997. ISBN 0-201-32563-2.

[8] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[9] NASA. Formal methods, specification and verification guidebook for software and computer systems, volume II: A practitioner's companion. Technical Report NASA-GB-001-97, NASA, Washington, DC 20546, USA, May 1997. Available from http://eis.jpl.nasa.gov/quality/Formal_Methods/.

[10] Robert M. Poston. *Automating Specification-Based Testing.* IEEE Computer Society Press, 1996. ISBN 0-8186-7531-4.

[11] SRI. The PVS verification system. WWW at URL http://www.csl.sri.com/pvs.html.

[12] Edward Yourdon. *Modern Structured Analysis.* Yourdon Press Computing Series. Prentice Hall, 1989. ISBN 0-13-598624-9.