Modelle der Software-Entwicklung - Abbilder oder Vorbilder ?

Jochen Ludewig, Institut für Informatik, Universität Stuttgart
Azenbergstr. 12, 7000 Stgt 1, ludewig@ifi.informatik.uni-stuttgart.dbp.de

Der nachfolgende Beitrag ist aus der Antrittsvorlesung entstanden, die ich am 28. Juni 1989 hier an der Universität Stuttgart gehalten habe. Ich gehe von der Problemstellung aus, wie sie sich heute aus meiner Sicht im Fach Software Engineering bietet. Anschließend gehe ich auf das Problem der Modellbildung ein und skizziere schließlich ein Projekt, das unsere Abteilung in den nächsten Jahren bearbeiten wird.

Wir suchen die Zusammenarbeit mit allen, die an der Thematik (Quantifizierung und Simulation der Software-Entwicklung) interessiert sind.

"Software Engineering" als Idee, und als Fach

In den ersten Jahren der Informatik war die Hardware das Nadelöhr, durch das alle Software gezwängt werden mußte. Alle Beschränkungen waren offensichtlich auf die Probleme mit den Rechnern zurückzuführen, vor allem auf die Knappheit des Speichers und die langsame Verarbeitung. Erst als diese Grenzen in den 60er Jahren zurückwichen, wurde deutlich, daß auch die Software Schwierigkeiten bereitet, daß wir also nicht alles machen können, was theoretisch offensichtlich möglich wäre. Dieses Phänomen wurde als **Software-Krise** bezeichnet. Es war etwa so, als ob ein Kind zunächst nur ein Dutzend Karten besitzt, um daraus Kartenhäuser zu errichten: Es meint, daß es nur die Zahl daran hindere, beliebig hoch zu bauen.

Software Engineering entstand vor gut zwanzig Jahren als provokative Wortschöpfung. Es war eine Idee, die gegen die Software-Krise gestellt wurde, mit der Botschaft: Macht es doch wie die Ingenieure, also die Brückenbauer, die Entwickler elektronischer Geräte, die Maschinen-Konstrukteure, die mit ihren Produkten im großen und ganzen recht erfolgreich sind. In diesem Falle wurde also ein neuer, noch sehr vager Begriff geschaffen, der uns als flatternde Fahne vorhergetragen wurde und den Weg wies (vgl. Ludewig, 1987).

"Software Engineering" bezeichnet damit den Versuch, Prinzipien der etablierten Ingenieurdisziplinen auf die Arbeit mit Software zu übertragen. Dabei steht jeder Beteiligte, Wissenschaftler oder Praktiker, vor dem Problem, einen Prozeß verbessern zu wollen, der bislang nur unzureichend verstanden ist und durch Änderungen der technischen Voraussetzungen, durch neue Erkenntnisse und auch durch Modeströmungen ständigem Wandel unterliegt.

Die "Entdeckung" dieses Gebiets hatte zur Folge, daß dafür auch Lehrstühle eingerichtet wurden. An einigen Hochschulen haben zudem die Forscher ihre Tätigkeit in diese Richtung verlagert[1]. Software Engineering ist damit ein neues Fach, das mit anderen konkurriert, also der Legitimation bedarf: Es muß *sinnvoll* und *möglich* sein, darin zu arbeiten.

Geht man von der Parallele zu anderen Fächern aus, so scheint der Sinn zweifelhaft; niemand käme auf die Idee, in der Elektrotechnik ein Fach "Electrical Engineering" (oder einfach "Elektrotechnik") einzurichten. Die betreffenden Lehrstuhlinhaber würden sich (zu Recht!) mit dem Hinweis wehren, sie *alle* machten das schließlich[2], so wie jeder Mediziner Hygiene oder Beratung der Patienten als Teil seiner Arbeit auffaßt.

Dieser Einwand gilt in der Informatik nur darum nicht, weil er offensichtlich durch die Realität widerlegt wird: Wer kennt schon die Normen, wer kümmert sich um die Standardisierung der Notationen oder gar der Verfahren, wer beurteilt die Methoden und Werkzeuge danach, wie und mit welcher Wirkung sie sich in der Praxis anwenden lassen? Aus diesem Grunde gibt es offensichtlich Bedarf für eine Avantgarde der Normalität, die in der Hochschule das Selbstverständliche fordert und lehrt und es den Praktikern vormacht. Wenn die Informatik eines Tages eine normale technische Disziplin geworden ist, werden wir uns auf das beschränken können, was in anderen Disziplinen "Verfahrenstechnik" heißt; dann ist wohl auch eine weniger globale Bezeichnung des Fachs sinnvoll.

Wie das Perpetuum Mobile zeigt, beweist der Bedarf noch nicht die Realisierbarkeit. Tatsächlich ist zu beobachten, daß Leute, die auf Software Engineering-Stellen berufen wurden, sich nach einigen Jahren wieder mit Themen befassen, die ohne Bruch auch in "klassische" Fächer wie Formale Sprachen, Compilerbau oder Graphentheorie paßten, oder daß sie ein akademisches Software-Haus betreiben, oder daß sie sich ganz auf die grundsätzliche Kritik konzentrieren[3]. Mir liegt es fern, diese Arbeiten als solche geringzuschätzen. *Software Engineering* muß aber nach meiner Meinung seine Position im Niemandsland halten, sonst geht die Chance der Vermittlung verloren. Vermittlung: das wäre wohl zuallererst der Brückenschlag zwischen Hochschule und Industrie, oder etwas vereinfacht: zwischen Theorie und Praxis.

---

[1] Leider gibt es meines Wissens keine Übersicht der betreffenden Lehrstühle und ähnlicher Forschungseinrichtungen im deutsprachigen Raum. Eine solche Liste wäre für die Zusammenarbeit nützlich.

[2] Mit genau dieser Begründung widersprach Prof. Wirth 1986 meinem Vorschlag, eine Abteilung im Institut für Informatik der ETH Zürich mit "Software Engineering" zu bezeichnen.

[3] Zitat aus dem Gespräch mit einem Kollegen von einer anderen Hochschule: "Ach, Sie machen Software Engineering. Dafür haben wir auch schon ein paar Leute berufen, aber die haben dann alle 'was anderes gemacht."

Die Informatik-Forschung hat sich bislang in der Regel auf den Standpunkt gestellt, daß die *Praxis* des Software Engineerings nicht als Ausgangspunkt für Verbesserungen betrachtet werden kann, weil sie von der idealen Situation, in der Software planmäßig, zu angemessenen Kosten und in definierter Qualität entwickelt würde, allzu weit entfernt ist. Daraus leitet man ab, daß die Praxis nicht beobachtet, sondern nur beeinflußt werden müsse[4].

Die Praktiker halten dem entgegen, daß die Ansätze der Hochschulen weltfremd seien, daß also vieles vorgeschlagen werde, was einfach nicht anwendbar sei. Sie wünschen Verbesserungen, die sich durch konzeptionell einfache, "schmerzlose" Maßnahmen realisieren ließen, beispielsweise durch die Einführung neuer Software-Werkzeuge.

Keiner der beiden Ansätze hat in der Vergangenheit befriedigende Ergebnisse geliefert. Die Praktiker erzeugen und unterhalten zwar mit Erfolg große Software-Systeme, bekommen aber den Prozeß und seine Kosten nicht in den Griff. Die Theoretiker weisen die Richtigkeit ihrer Hypothesen mehr oder minder schlüssig nach, legen aber Prämissen zugrunde, die eine praktische Überprüfung schwer machen oder ausschließen. Damit ergibt sich eine Situation, in der sich die Vertreter der Praxis und der Theorie mit gewissem Recht gegenseitig für inkompetent erklären und in "splendid isolation" *nebeneinander* leben, statt *voneinander* zu profitieren.

Die besondere Schwierigkeit unseres Fachs scheint darin zu liegen, daß wir zwar eine Reihe allzu bekannter Symptome sehen (Software ist oft zu teuer, zu schlecht und nicht genügend präzise planbar), aber kein zentrales Problem[5]. Es liegt daher nahe, ein einzelnes Problem herauszugreifen und zu behaupten, dies allein sei schuld: *"Ihr müßt nur alles formalisieren /* oder *mit dem Benutzer kooperieren /* oder *nicht-prozedurale Sprachen einsetzen /* oder *... /, dann werden eure Probleme verschwinden."* Ich leite daraus meine These ab:

> *Das zentrale Problem des Software Engineerings ist, daß es im Software Engineering kein zentrales Problem gibt, sondern viele verfilzte Einzelprobleme, die einen außerordentlich zähen Verband bilden. Eine Verbesserung ist nur durch viele verschiedene Einzelmaßnahmen zu erzielen. Diese lassen sich nur durchsetzen, wenn ihr Risiko gering, ihr Nutzen aber nachweisbar oder wenigstens sehr plausibel ist.*

---

[4]Die Forschung entwickelt daher Modelle mit Vorbild-Charakter (siehe unten), beispielsweise das Modell der formalen Spezifikation und Verifikation.

[5]Es sind, in der Sprache der Mediziner, unspezifische Symptome, die wenig Hilfe für die Diagnostik leisten.

Modelle

Techniker haben seit Jahrtausenden mit Modellen gearbeitet, und in der Informatik benutzen wir - in aller Regel wohl ohne Reflektion - das Wort in Verbindungen wie "Life Cycle Model". Ich will daher zunächst den Modellbegriff in seiner Bedeutung für die Informatik beleuchten. Ausführliche Darstellungen finden sich bei Stachowiak (1973) und Luft (1984).

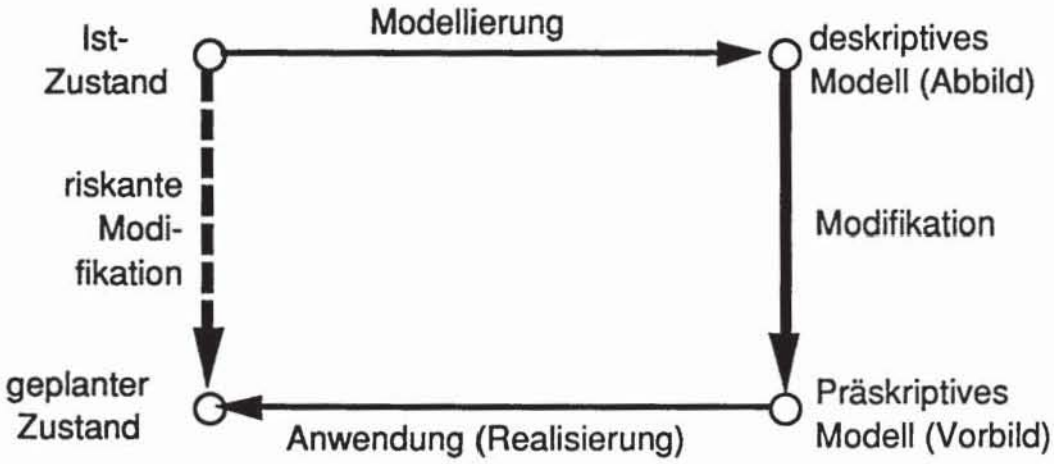
Modelle sind entweder *Abbilder* von etwas oder *Vorbilder* für etwas. Ihnen sind in allen Fällen drei Merkmale zueigen:
das **Abbildungsmerkmal**
Zum Modell gibt es das Original, ein Gegenstück, das wirklich vorhanden, geplant oder fiktiv sein kann.
das **Verkürzungsmerkmal**
Ein Modell erfaßt im allgemeinen nicht alle Attribute des Originals, sondern nur solche, die seinem Schöpfer relevant erscheinen.
das **pragmatische Merkmal**
Modelle können unter bestimmten Bedingungen und bezüglich bestimmter Fragestellungen das Original ersetzen.

Stachowiak (1973, S. 139 f.) beschreibt die wesentliche Funktion der Modelle (verkürztes Zitat):

> *... Dabei ist bei allen Modellierungen, die zum Informationsgewinn über das Original führen sollen, die folgende Vorgehensweise zu beobachten. Das Original wird in sein Modell abgebildet, wobei zumeist zahlreiche Originalattribute fortgelassen und oft Modellattribute neu eingeführt werden. (...) Mittels zielgerichteter Modelloperationen wird dann das ursprüngliche Modell in ein verändertes übergeführt. Ist die attributenmäßige Original-Modell-Zuordnung umkehrbar eindeutig, so sind den modellseitigen Operationen bestimmte originalseitige zugeordnet, und die faktisch-operative Überführung des ursprünglichen in das veränderte Modell zieht eine wohlbestimmte hypothetisch-operative Überführung des ursprünglichen Originals in ein verändertes nach sich. (...)*
>
> *Der Gewinn dieser Vorgehensweise liegt auf der Hand: Modell-seitig gewonnene Einsichten und Fertigkeiten lassen sich - bei Erfülltsein gewisser Transferierungskriterien - auf das Original übertragen, der Modellbildner gewinnt neue Kenntnisse über das modellierte Original, er bekommt dieses besser als bisher in den Griff, kann es auf neue Weise zweckdienlich umgestalten oder als verbessertes Hilfsmittel für neue Aktionen verwenden.*

Modelle werden also immer dann eingesetzt, wenn eine anscheinend sinnvolle Änderung in der Realität zu riskant oder zu schwierig ist, um sie direkt herbeizuführen. Beispielsweise ist die Erstellung eines Gebäudes heute zwar technisch in der Regel nicht besonders schwierig, doch besteht das Risiko, daß es dem Kunden dann nicht gefällt oder die Landschaft verschandelt. Aus diesem Grunde wird ein Modell angefertigt, das diese

Risiken zu vermindern hilft. Wir gehen also nicht direkt vom Ist-Zustand in den geplanten Zustand über, sondern zunächst in das deskriptive Modell (im Falle des Gebäudes ein Modell der Umgebung, also der Landschaft) und von dort zum präskriptiven Modell (Landschaft mit dem geplanten Gebäude). Erscheint dieses - u.U. nach mehreren Versuchen - akzeptabel, so wird der Schritt zurück in die Realität vollzogen (das Haus wird gebaut).



Wenigstens zwei Modelle sind heute im Bereich des Software Engineerings in aller Munde, das Life Cycle Model und das Constructive Cost Model (COCOMO). Beide wurden von B.W. Boehm (1973, 1980) bekannt gemacht. Gerade im Bereich der Kostenschätzung gibt es weitere bekannte Modelle, beispielsweise als Grundlage der Function Point-Methode.

Für die Bearbeitung von Software sind in den letzten Jahren zahlreiche weitere Modelle entstanden (vgl. Agresti, 1986), die hier zusammen mit den "klassischen" ohne Anspruch auf Vollständigkeit aufgelistet sind. Dazu sind allerdings einige Bemerkungen vorauszuschicken:

- Die Terminologie auf diesem Gebiet ist sehr schwankend, vor allem das Wort "Prototyp" wird für nahezu jede Bedeutung verwendet[6].
- Die Ansätze stehen nicht alternativ nebeneinander, sondern werden in der Praxis kombiniert.
- Die zitierten Literaturquellen sind nur teilweise die Originalarbeiten zu den Ansätzen, bieten aber jeweils einen Einstieg!
- Es gibt hier keine "richtige" Reihenfolge; ich habe versucht, an den Beginn die alten, einfachen Modelle zu setzen, es folgen verschiedene Formen von Prototyping im weiteren Sinne, dann die Baustein-orientierten, die formalen und schließlich die kombinierten Ansätze.

---

[6]"Prototyp" ist für *Software* ein unglückliches Wort, denn es bezeichnet in der fertigenden Industrie ein Produkt-Muster, dessen Stückkosten zwar relativ hoch(!) sind, das aber keine Investitionen für eine Serienfertigung beansprucht. Der Prototyp eines Autos kommt natürlich *teurer* als der Wagen aus der Serie.

Code and Fix

Codierung, unsystematischer Test, Fehlersuche und Korrektur als einzige identifizierbare Tätigkeiten

Der traditionelle Ansatz, noch heute weit verbreitet; in jüngster Zeit oft mit "evolutionäre Entwicklung" oder "Prototyping" umschrieben

Wasserfallmodell (Royce, 1970)

Aktivitäten, als (zyklischer) Graph durch Teilergebnisse gekoppelt

Die erste Alternative zum Code and Fix. Relativ einfach und verständlich

Kostenmodell (Frühauf, Ludewig, Sandmayr, 1988)

Phasen, durch Meilensteine zyklenfrei gekoppelt

Der klassische Ansatz des Managements, nicht zu verwechseln mit dem Wasserfallmodell

Treppenmodell (Rzevski, 1984; Frühauf, Jeppesen, 1986):

Kern- oder Pilotsystem, das in vorgeplanten Schritten ausgebaut wird

Der Kompromiß, wenn ein komplexes Produkt schnell auf den Markt oder zum Kunden soll

Evolutionäre Entwicklung (Basili, Turner, 1975, Hewitt, 1980; vgl. auch Brooks, 1987)

Nullversion mit beschränkter Leistung, die sich, gesteuert durch die Erfahrungen bei der Anwendung, organisch weiterentwickelt

Die geeignete Lösung, wenn Entwickler und Kunde viel Kontakt haben können und der Kunde nicht in der Lage ist, die Anforderungen explizit zu formulieren

Vorversion (Brooks, 1975)

Brauchbares Produkt geringer Qualität, das vorübergehend eingesetzt, im Laufe der weiteren Entwicklung aber praktisch ganz ersetzt wird.

Ein in der Industrie oft übliches Vorgehen, nicht durch bewußte Entscheidung, sondern durch den Druck der Rahmenbedingungen

Prototyp (Mock Up, Attrappe) (Boehm, Gray, Seewald, 1984; Budde et al., 1984; Diederich, Milton, 1987)

Spielversion, die nicht als Produkt eingesetzt werden kann, sondern explizit der Anforderungsklärung dient

Ein Vorgehen, das in der Literatur wesentlich häufiger vorkommt als in der Praxis

Simulationsmodell (Alford, 1985)

System, das die Voraussage von Leistungsmerkmalen gestattet

vor allem bei Anwendungen, in denen die Antwortzeiten kritisch sind, beispielsweise in Kommunikationssystemen, im Betriebssystemkern, bei Echtzeit- Anwendungen

Generierungsmodell (Cheatham, 1983)

Entwicklung einer abstrakten Lösung und (parallel) eines problem- und zielmaschinenspezifischen Generators

Besonders gut geeignet zur Entwicklung von Programmfamilien mit Varianten der abstrakten Lösung und Varianten der Transformatoren

Universelle Generatoren (4GL) (Gutzwiller, Österle, 1988)

Lösung innerhalb eines vorgegebenen (relativ engen) Schemas, daraus Generierung mit einem Compiler

4GL, die bevorzugte Lösung der "Endanwender"

Baustein-Ansatz (IEEE, 1987; Trasz, 1988)

Programmentwicklung mit dem vorrangigen Ziel, vorhandene Bausteine zu nutzen oder wiederverwendbare Bausteine zu entwickeln

Der Ansatz für Organisationen, die immer wieder ähnliche (Teil-)Aufgaben lösen

Ausbaubarer Ausschnitt (nach Praxisberichten)
Die Entwicklung beginnt mit einem bestimmten Lösungsausschnitt
das typische Vorgehen bei Systemen, die mit Masken arbeiten, wenn ein Masken-Editor zur Verfügung steht

Schrittweise Vervollständigung (Ludewig et al., 1987)
Ein Prototyp wird sukzessive in ein Produkt überführt
experimenteller Ansatz in der Arbeit der Abteilung Software Engineering in Stuttgart

Halbformale Spezifikation (Ludewig, 1989)
Möglichst formale Fassung der Spezifikation, dann konventionelle Entwicklung
Pragmatischer Kompromiß zwischen Formalität und Praktikabilität

Formale Spezifikation und Verifikation (Gries, 1987)
Formale Spezifikation und Implementierung, dann Verifikation
Der akademische Ansatz der 70'er Jahre

Transformationsansatz (Bauer et al., 1985)
Formale Spezifikation, dann korrektheitserhaltende Transformationen
Ein akademischer Ansatz für Leute mit guter formaler Ausbildung (Beispiel: CIP)

Cleanroom Development (Mills, 1987)
Programmentwicklung mit höchstem Aufwand zur Fehlervermeidung, praktisch ohne Testaufwand
Ein radikaler Ansatz im Sinne der Software-Qualitätssicherung

Spiralenmodell (Boehm, 1988)
Vereinigung verschiedener Konzepte, wobei für jeden Schritt der optimale Ansatz gewählt werden kann.
Die Kombination der verschiedenen Ansätze zur Minimierung des Risikos

Betrachtet man diese Ansätze unter dem Aspekt des Modell-Begriffs, so fällt folgendes auf:

- Das Original ist in der Regel nicht beschrieben (aus welchen Projekten ist das Modell abgeleitet?). Wir sind damit in einer Situation, wo Uhrmacher und Schiffbauer so tun, als ob sie die gleichen Probleme hätten, indem sie jeweils allgemeingültige Regeln für die mechanische Konstruktion aufstellen.
- Die Zielsetzung der (im Modell steckenden) Modifikation des Prozesses (z.B. der Formalisierung anstelle eines Vorgehens mit informalen Notationen) ist nicht definiert. Damit bleibt auch unklar, ob das Ziel erreicht wird. Qualitative Aussagen über den relativen Erfolg (Davis, Bersoff, Comer, 1988) sind ohne praktischen Nutzen.
- Die Rücktransformation in das Original (d.h. die Übertragung des Modells in die Praxis) wird nicht als integraler Teil der Arbeit aufgefaßt, die darin verborgenen Probleme bleiben daher unentdeckt, und der Wert des Modells kann nicht beurteilt werden.

Es entsteht damit der Eindruck, daß wir in der Informatik die Welt der Modelle von der Realität abgekoppelt haben und auf die Verbindung beider Welten keinen besonderen Wert legen.

G O S E - the Game Of Software Engineering

Software ist abstrakt, und Methoden sind abstrakt, Methoden des Software Engineerings sind darum doppelt abstrakt. Diese Tatsache erschwert die Ausbildung außerordentlich. Hinzu kommt, daß Software Engineering nur unter realistischen Randbedingungen möglich ist; an einer Hochschule können diese nicht, auch nicht annähernd, geschaffen werden. Außerhalb der Hochschulen wären zwar die Randbedingungen gegeben, doch fehlt dort in der Regel der zeitliche, personelle und finanzielle Spielraum, um zu experimentieren, Alternativen zum gewohnten Vorgehen durchzuspielen.

in dieser Lage ist eine Simulation sehr attraktiv: Sie gestattet es, in kurzer Zeit (z.B. in einigen Stunden oder wenigen Tagen) Projekte durchzuspielen, die in der Praxis Monate oder Jahre brauchen und erheblichen Aufwand erfordern. Parameter wie etwa der Aufwand für die Qualitätssicherung oder die Verfügbarkeit von Werkzeugen können variiert und auf ihre Auswirkungen untersucht werden.

In der Abteilung Software Engineering soll ein solches Simulationspaket in Form eines Computerspiels (GOSE) entwickelt werden. Wie bei Adventure, Dungeon usw. spielt eine Person oder eine Gruppe gegen den Rechner. Sie übernimmt im Rahmen eines (simulierten) Software-Projekts die Rolle des Projektleiters. Ziel des Spiels ist es, das Projekt erfolgreich durchzuführen und abzuschließen.

Wir verfolgen damit mehrere Ziele:

- Sammlung und Zusammenstellung der heute bekannten oder vermuteten Gesetzmäßigkeiten im Software Engineering
- Zusammenarbeit mit Leuten in Firmen und Hochschulen, die zu diesen Kenntnissen beitragen können
- Projekttraining für alle Mitarbeiter durch die Arbeit *an* GOSE
- Bereitstellung eines Mittels für die Ausbildung (die Arbeit *mit* GOSE)

Konzeption des Spiels

GOSE wird konzipiert nach dem Vorbild von Adventure, d.h. der Spieler muß selbst seine Aktivitäten wählen, er wird dabei nicht geführt, so wie es auch in realen Projekten der Fall ist. Aber auch wenn er nichts tut, läuft wie bei einem Flugsimulator die Zeit weiter, die Budgets verrinnen, und es treten - überwiegend stochastisch - Störungen auf, beispielsweise Kündigungen der Mitarbeiter, Rechnerausfälle, Lieferverzögerungen. Spiele dieser Art sind in der letzten Zeit sehr populär geworden, z.B. als Lernmittel für Studenten der Betriebswirtschaft oder als Simulator für Reviews (siehe McKeeman, 1989).

Wie in einem realen Prozeß sind die Zustandsgrößen nicht direkt sichtbar; es liegt im Ermessen des Spielers (d.h. des zunächst schlecht informierten Projektleiters), welchen Aufwand er für ihre Beobachtung und Bewertung treiben will.

Alle Ereignisse und Zwischenzustände werden intern protokolliert. Diese Aufzeichnungen können später zu einer Analyse des Projektverlaufs herangezogen werden (vor allem interessant, wenn der Spieler auch seine Schätzungen aufzeichnet). Sie können auch verwendet werden, um die Uhr zurückzustellen und eine andere Variante durchzuspielen.

Besondere Bedeutung hat die integrierte Regelsammlung: Der Spieler kann jederzeit nachsehen, welche Möglichkeiten des Handelns in der Literatur vorgeschlagen werden, und er kann sich, wenn er will, entsprechend verhalten. Beispielsweise kann er beschließen, den Aufwand für Qualitätssicherung durch Einführung von Review zu erhöhen; die Zustandsgrößen des Entwicklungsprozesses werden entsprechend reagieren (in diesem Fall durch die zusätzliche Belastung der Mitarbeiter negativ, durch höhere Qualität positiv).

Das GOSE-Projekt

Unser (reales) Projekt besteht aus folgenden Aktivitäten, die sich wechselseitig beeinflussen:

- Sammlung von Hypothesen über die Software-Bearbeitung
- Modellierung der Praxis (Aufbau eines Simulators)
- Anwendung des Modells zu seiner Überprüfung und Verbesserung
- Anwendung des Modells in der Ausbildung

Unter den gegebenen Bedingungen wählen wir einen evolutionären Ansatz. Nachdem in einem ersten Schritt die wichtigsten Merkmale festgelegt sind, wird ein einfaches Modell implementiert. Dieses Modell wird vermutlich mehr Ablehnung als Zustimmung hervorrufen, aber die Kritik kann zur Verbesserung und zum Ausbau führen.

Für unser Vorgehen haben wir festgelegt:

1. Zunächst wird nur die Konzeptionsphase und die Pilotimplementierung geplant, weitere Planung folgt, wenn die ersten Ziele erreicht sind.
2. Das GOSE-Projekt besteht aus den sechs Mitarbeitern der Abteilung Software Engineering sowie Diplomanden. Einer der Mitarbeiter ist Projektleiter, diese Rolle rotiert. Ich wirke beratend mit.
3. Implementiert wird in Ada auf Workstations; eine Portierung auf kleinere Maschinen (z.B. PC, Macintosh) soll offengehalten werden.
4. Die Entwicklungssprache ist Deutsch, die Bediensprache zunächst ebenfalls, sollte aber leicht änderbar sein.

5. Der Informationsaustausch mit externen Stellen wird gefördert, eine Förderung des Projekts aus öffentlichen oder privaten Mitteln wird angestrebt.
6. Die Konzeptionsphase hat im Juni 1989 begonnen, die Pilotimplementierung wird vom September 1989 an entwickelt. Sie sollte Ende Februar 1990 lauffähig sein.

Die Entwicklung von GOSE ist offensichtlich keine leichte Aufgabe. Abgesehen von dem beträchtlichen Aufwand, der allein in die Implementierung fließen wird, enthält die Modellierung eine Reihe ungelöster Probleme. Es handelt sich also um echte Forschungsarbeit.

Praxis für Theoretiker, Theorie für Praktiker

Modelle dienen in vielen Fällen dazu, den unerwünschten, aber in der Praxis zum Glück seltenen Spezialfall zu üben (z.B. Flugsimulatoren). Im Software Engineering liegen die Dinge heute anders: Der Praktiker kennt aus der täglichen Erfahrung die unerwünschte Situation besser, als ihm lieb ist. Was ihm fehlt, ist die Gewißheit, daß es auch möglich ist, Software auf reguläre Weise zu entwickeln und zu bearbeiten, also planmäßig, innerhalb definierter Grenzen für Kosten und Zeit. Die Theorie hat ihm diese Gewißheit bis heute nicht geben können; sie ist dabei nicht zuletzt an ihrer mangelnden Bereitschaft (oder Fähigkeit) gescheitert, das Ideal vorzuleben.

Mit dem Projekt GOSE wollen wir daher zwei Fliegen mit einer Klappe schlagen:

- Wir führen selbst ein Projekt durch und unterwerfen uns dessen Schwierigkeiten. Und wir werden dabei auch sehr kritisch gegen unsere eigene Arbeit sein, also unsere Hypothesen im Selbstversuch prüfen. Beispielsweise werden wir ein Projekt-Tagebuch führen, um die Möglichkeiten des Selbstbetrugs zu begrenzen.
- Wir stellen, soweit wir erfolgreich sind, dem Praktiker ein Mittel zur Verfügung, mit dessen Hilfe er vorteilhafte Wege zu besserem Software Engineering ohne Risiko praktisch ausprobieren kann[7].

Natürlich ist dieses Mittel auch sehr attraktiv für Studenten, die sich, was die Erfahrungen angeht, von den Praktikern nur dadurch unterscheiden, daß sie bis zur Assimilation im Beruf unbeirrbar am Glauben festhalten, in der Praxis sei alles nicht so schlimm, wie wir es in der Hochschule behaupten.

---

[7]Man könnte dieses Spiel mit den Wohngemeinschaften vergleichen, in denen ehemalige Drogenabhängige auf ein normales Leben vorbereitet werden. Hier wie dort muß Normalität mühsam erlernt werden.

Literaturangaben

Agresti, W.W. (1986): **New Paradigms for Software Development.** IEEE Tutorial, Order No. FJ707.

Alford, M. (1985): SREM at the age of eight: The distributed computing design system. **IEEE COMPUTER, 18,** 4, 36-46.

Basili, V.R., A.J. Turner (1975): Iterative Enhancement: a practical technique for software development. **IEEE Trans. on Softw. Eng., SE-1,** 4, 390-396.

Bauer, F.L, et al (CIP Language Group) (1985): **The Munich Project CIP, Vol.1:The Wide Spectrum Language CIP-L.** Lecture Notes in Computer Science Vol. 183, Berlin, Heidelberg, New York, Springer.

Boehm, B.W. (1973): Software and its impact: a quantitative assessment. **DATAMATION, 19,** 5, 48-59.

Boehm, B.W. (1980): **Software Engineering Economics.** Prentice Hall, Englew. Cliffs, NJ.

Boehm, B.W., T.E. Gray, T. Seewald (1984): Prototyping vs. specifying: a multiproject experiment. **IEEE Trans. on Software Engineering, SE-10,** 290-303.

Boehm, B.W. (1988): A spiral model of software development and enhancement. **IEEE COMPUTER, 21,** 5, 61-72.

Brooks, F.P. Jr. (1975): **The Mythical Man Month.** Addison Wesley, Reading, MA.

Brooks, F.P. Jr. (1987): People are our most important product. in Gibbs, Fairley (eds.): **Software Engineering Education.** Springer Verlag, New York etc., pp.1-15.

Budde, R., K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (eds.) (1984): **Approaches to Prototyping.** Springer-Verlag, Berlin etc.

Budde, R., Kuhlenkamp, K., Sylla, K.H. (1987): Konzepte des Prototyping. in Requirements Engineering´87, **GMD-Studien Nr. 121,** GMD, Sankt Augustin, pp 75-94.

Cheatham, T.E. (1983): Reusability through program transformations. **Workshop on Reusability in Programming,** Newport, RI, September 1983, pp. 122-128.

Davis, A.M., E.H. Bersoff, E.R. Comer (1988): A strategy for comparing alternative software development life cycle models. **IEEE Trans. on Software Engineering, SE-14,** 10, 1453-1461.

Diederich, L.G., Milton, J. (1987): Experimental Prototyping in Smalltalk-80. **IEEE Software, 4,** 3 (May 1987), 50-64.

Frühauf, K., Jeppesen, K.I. (1986): Software Development, the staircase approach. **1st IFAC-Workshop on Experience with Management of Software Projects,** Pergamon Press, London.

Frühauf, K., J. Ludewig, H. Sandmayr (1988): **Software-Projektmanagement und -Qualitätssicherung.** Teubner, Stuttgart, und vdf, Zürich.

Gries, D. (1987): **The science of programming.** Springer, New York usw., 4th print

Hewitt, C. (1980): Evolutionary programming. in Freeman, Lewis (eds.): **Software Engineering.** Academic Press. 133-147.

IEEE (1987): Tools making reuse a reality. **IEEE Software, 4**, 4 (July 1987), Themenheft "Reuse", 6-72.

Ludewig, J. (1987): Software Engineering: Computer-Programme als technische Produkte. **TECHNISCHE RUNDSCHAU**, Bern, Heft 7, 1987, 50-57.

Ludewig, J. Färberböck, H., Lichter, H., Matheis, H., Wallmüller, E. (1987): Software-Entwicklung durch schrittweise Komplettierung. in **Requirements Engineering'87**, GMD-Studien Nr. 121, GMD, pp. 113-124, Sankt Augustin.

Ludewig, J. (1989): Languages, methods, and tools for software specification. in J. Zalewski, W. Ehrenberger: **Hardware and Software for Real Time Process Control**. North Holland, Amsterdam etc., 225-256.

Luft, A.L. (1984): Zur Bedeutung von Modellen und Modellierungsschritten in der Softwaretechnik. **Angewandte Informatik**, Heft 5 (1984), 189-196.

McKeeman, W.M. (1989): Graduation Talk at Wang Institute. **IEEE COMPUTER, 22**, 5, 78-80.

Mills, H.D., et al. (1987): Cleanroom process - A software engineering approach to statistical quality control. **IEEE Software, 4**, 5 (September 1987), 19-25.

Gutzwiller, Th., H. Österle (Hrsg.) (1988): **Anleitung zu einer praxisorientierten Software-Enwicklungsumgebung**. Band 2: Entwicklungssysteme und 4.-Generation-Sprachen. Angewandte InformationsTechnik Verlags GmbH, Halbergmoos.

Royce, W.W. (1970): Managing the development of large software systems. **IEEE WESCON**, August 1970, pp.1-9. Nachgedruckt in Proc. of the 9th ICSE (IEEE), pp.328-338.

Rzevski, G. (1984): Prototypes versus pilot systems: strategies for evolutionary information system development. in Budde et al. (1984), pp.356-367.

Stachowiak, H. (1973): **Allgemeine Modelltheorie**. Springer-Verlag, Wien etc.

Tracz, W. (1988): **Software Reuse - Emerging Technology**. IEEE Tutorial Order No. FJ846.