SESAM

Software-Engineering-Simulation durch animierte Modelle

Jochen Ludewig (Hrsg.)

Inhalt

Teil 1	Jochen Ludewig	Die Abteilung Software Engineering — ein privater Rückblick	3
Teil 2	Jochen Ludewig	SESAM: Grundidee und Überblick	5
Teil 3	Kurt Schneider	SESAM: Die konzeptionelle Basis	7
Teil 4	Jinhua Li	SESAM als Simulator	21
Teil 5	Anke Drappa	SESAM und die Realität	27
Teil 6	Marcus Deininger	SESAM und die Lehre	35
Teil 7	Jürgen Schwille	SESAM und vis-A-vis	49
Teil 8	Horst Lichter	Software Engineering objektorientiert — eine Herausforderung für die Praxis	55
Teil 9	Jochen Ludewig	Software Engineering in der Universität	61

Teil 1

Die Abteilung Software Engineering — ein privater Rückblick

Jochen Ludewig

1. Ab urbe condita

Die Abteilung Software Engineering entstand als eine der letzten in der Stuttgarter Informatik, der Lehrstuhl wurde Ende 1986 erstmalig ausgeschrieben. Der Name der neuen Abteilung fand weniger Zustimmung als ihr Arbeitsgebiet, und mit dem Senatsbeschluß zur Ausschreibung war die Aufforderung an die Kommission verbunden, einen anderen, besseren Namen zu suchen. Tatsächlich blieb es beim Namen „Software Engineering“. Wir haben uns daran gewöhnt, und wenn mir auch eine deutsche Bezeichnung lieber wäre, so ziehe ich diesen klaren Anglizismus doch allen Mischwörtern wie „Software-Technologie“ oder „Software-Technik“ vor.

Nachdem ich den Ruf nach Ende Juli 1988 angenommen hatte, gelang es in der kurzen Frist von zwei Monaten, die Berufungsurkunde nicht nur auszustellen, sondern auch vom Ministerpräsidenten unterzeichnen zu lassen. Damit stand dem Dienstantritt am 1. Oktober 1988 nichts mehr im Wege.

Der Start fiel mit dem Ausscheiden Prof. Barths zusammen. So wurden wir die Erben der Abt. Programmiersprachen, sowohl bei den Pflichten (Einführungsvorlesung I und II) als auch bei den Privilegien (Räume im Gebäude Azenbergstraße und vor allem die Betreuung durch Frau Günthör).

Da Frau Günthörs Zuordnung am Anfang noch nicht klar war, gab es zunächst nur einen Mitarbeiter, Horst Lichter, der mutig (oder desperat) genug war, um den Wechsel von Zürich nach Stuttgart mitzumachen.

Das erste Jahr war geprägt durch die Rekrutierung weiterer Mitarbeiter und durch den Papierkrieg zum Zwecke der Rechnerbeschaffung. 18 Monate nach dem Start waren alle Stellen besetzt, die Gruppe hatte fast die gleiche Zusammensetzung wie heute:

Ursula Günthör, die auch Mitglied der Abteilung Programmiersrprachen ist, im Sekretariat.

Angela Georgescu und **Max Schneider** auf den Programmiererstellen.

Horst Lichter, **Marcus Deininger**, **Kurt Schneider**, **Jürgen Schwille** als wissenschaftliche Mitarbeiter; der dienstälteste hat inzwischen seine Stelle geräumt und damit Platz gemacht für **Anke Drappa**.

Thomas Bassler war 16 Monate ab Herbst 1991 wissenschaftlicher Mitarbeiter der Abteilung. Heute sind zwei Doktoranden als Stipendiaten bei uns, **Helga Hoff** und **Jinhua Li**.

Die Rechnerauswahl war zugunsten von DEC-Workstations entschieden worden; da wir unser Geld nur zögerlich ausgaben, vermehrte sich sein Wert wundersam, und wir haben schließlich eine leistungsfähige, zuverlässige und vermutlich noch für einige Jahre brauchbare Rechnerkonfiguration bekommen. Einzig die Software macht uns Sorgen, da wir mit den MIPS-Prozessoren an einem Gleis sitzen, das von DEC nicht mehr bedient wird.

In das erste Jahr fällt auch die Idee zum Projekt SESAM; darauf werde ich im zweiten Teil dieses Beitrags näher eingehen.

Die folgenden Jahre waren in keiner Weise spektakulär. Ämter wie das des Geschäftsführenden Institutsdirektors ließen sich nicht vermeiden, eher undankbare Lehrveranstaltungen wie die Einführung in die Informatik III auch nicht. Der IVS und die verwaiste Abteilung Programmiersprachen erzeugten auch eine gewisse Belastung, inzwischen sind zum Glück beide unter kompetenter Führung (Proff. Claus und Plödereder).

Die Mitarbeiter wurden kompetenter und selbständiger, so daß es immer riskanter wurde, ihnen leichtfertig zu widersprechen. Verschiedene Engagements in der Industrie zur Schulung und zur Kooperation waren nicht nur eine willkommene Möglichkeit, den Hypothekarzins-Drachen niederzukämpfen, sondern hielten auch das Gefühl für das in der Praxis Notwendige und Mögliche wach.

Als im Wintersemester 92/93 Ämter und Lehrveranstaltungen die offene Flanke meiner organisatorischen Immunschwäche nutzten, um mich endgültig ins Chaos zu stürzen, konnte nur noch die vorübergehende Flucht helfen („Forschungssemester“). Sie führte mich ins schöne Land Ontario, und sie wäre ganz und gar erfolgreich gewesen, hätte ich nicht meine temporäre e-mail-Adresse in Stuttgart hinterlassen.

2. Hochschullehrer: Ein Anlern-Beruf

Hochschullehrer und Politiker haben gemeinsam, daß sie zunächst eine sehr strenge Auswahl überstehen müssen, um ins Amt zu kommen. Anschließend stehen sie in aller Regel vor Problemen, die mit den Kriterien dieser Auswahl nur wenig zu tun haben.

Bei der Bewerbung um eine Hochschullehrerstelle muß man unter Beweis stellen, daß man auch schwierige wissenschaftliche Probleme lösen kann und in der Lage ist, die Resultate vorzutragen und zu publizieren. Neben der fachlichen Ausrichtung entsprechend der Widmung ist dies das wesentliche Kriterium.

Als Hochschullehrer steht man vor Aufgaben, die eine Art wissenschaftlichen Herkules erfordern: Natürlich soll man auch weiterhin und mit Erfolg wissenschaftlich aktiv sein; daneben soll man einen gelegentlich spröden, auch für den Dozenten nicht immer begeisternden Stoff für Studenten aufbereiten, die nicht stets und sämtlich auf das Wissen scharf sind. Man soll Mitarbeiter auswählen und Doktoranden so führen, daß sie die entscheidende Hilfe erhalten, um nach einigen Jahren den Sprung über die Hürde zu schaffen.

Neben diesen im ganzen erfreulichen Aufgaben gibt es andere, die jedenfalls nur selten erfreulich, oft höchst unerfreulich sind. Als Mitglied des Institutsvorstands oder gar als Geschäftsführender Direktor hat man in den Bereichen Personal und Haushalt Aufgaben, die einfach nicht befriedigend zu lösen sind. Personalprobleme entstehen unter den Rahmenbedingungen der Universität (Führungsdefizit, BAT und Beamtenrecht) unausweichlich wie die Leberzirrhose des Wirts, und ihre Behandlung ist ebenso erfolgreich. Die Unsicherheiten und Irrationalitäten der öffentlichen Haushalte sind ein ständiger Quell für Irritationen, vorsorgliche Diskussionen und Anträge, Leerlauf eben.

Kann man für diese Schwierigkeiten irgend jemanden verantwortlich machen? Zum Teil. Die Ausstatung der Institute mit Geld ließe sich nach meiner Meinung deutlich effizienter und rascher organisieren, vor allem, indem man die Mittel den Instituten zur freien Verfügung überträgt, ohne Bindung an Titel und Haushaltsjahre. Schon die Umstellung vom Kalenderjahr auf das Studienjahr wäre eine Verbesserung. Im übrigen leiden wir unter den impliziten politischen Vorgaben, die niemand formulieren will, aber jeder in irgendeiner unbestimmten Form im Kopf hat, schwankend je nach politischer Strömung und Konjunktur. Was soll die Universität denn leisten? Hochwertige Lehre oder Drittmittelforschung? Buchstabengetreue Exekution der Gesetze und Bestimmungen oder Nägel mit Köpfen? Ausschluß oder Auszeichnung derer, die Studentenausweis *und* festen Job haben? Arbeit zum Wohle des Landes oder möglichst viele Publikationen? Throughput oder Qualität? Ausbildung der Massen oder Bildung einer Elite?

Das „oder“ ist hier, zugegeben, nicht wirklich exklusiv, aber kaum ein Hochschullehrer ist nach meiner Erfahrung und Beobachtung in der Lage, *alle* Anforderungen zu erfüllen. Er muß also Prioritäten setzen. Wenn wenigstens die Gemeinschaft der Lehrenden einen — natürlich nicht punktförmigen — Konsens erreichte, dann hätten sie doch eine Position, von der aus man werten und urteilen könnte. Unter den gegebenen Bedingungen geht jegliche Stellungnahme, z.B. zu einer Mittel- oder Stundenkürzung, von Prämissen aus, über die nie Einigkeit bestanden hat.

Meine Zeit in der Industrie und an der ETH Zürich brachte einige interessante und bis heute nachwirkende Kurse mit, z.B. über Personalführung und über Hochschuldidaktik. Jeder Student muß im Studium bestimmte Nachweise erbringen, teilweise als Vorleistung (Praktika, Latinum). Ich sehe noch immer nicht ein, warum der Hochschullehrer nicht als Teil seiner Bewerbung, notfalls nachträglich, Scheine über die erfolgreiche Teilnahme an bestimmten Kursen vorlegen muß, z.B. über Didaktik oder über öffentliche Haushalte.

Teil 2

SESAM: Grundidee und Überblick

Jochen Ludewig

1. Der Hintergrund

Als 1989 die Gruppe langsam auf ihre Sollgröße wuchs, stellte sich die Frage, an welchem Thema sie denn arbeiten sollte. Naheliegende, wenn auch keineswegs ganz bewußte Randbedingungen waren:

- Das Thema soll praxisnah sein, also nicht von einem unsinnig vereinfachten Bild der Praxis ausgehen, und Resultate liefern, die in der Praxis eingesetzt werden können.
- Es soll die spezifische Stärke der Hochschule zur Geltung bringen, nämlich die Freiheit von kurzfristigen Rentabilitätsüberlegungen und die Freiheit zur firmenübergreifenden Forschung.
- Es soll Raum für viele miteinander verbundene Arbeiten bieten und in absehbarer Zeit nicht zu erschöpfen sein.
- Es soll den Doktoranden eine auch im Hinblick auf ihre spätere Arbeit nützliche Erfahrung verschaffen.
- Und es soll allen Beteiligten Spaß machen.

So kam mir die Idee, den Prozeß der Software-Entwicklung in einem Computer-Spiel zu simulieren. „Auslöser“ war ein Zeitungsartikel über Ökolopoly von F. Vester; viele Erinnerungen und Erfahrungen, beginnend mit meiner Diplomarbeit (ein Simulationssystem, 1973), auch frühe Spiele auf der VAX („Dungeon“), bildeten den Hintergrund.

2. Die Zielsetzung

Was soll SESAM? Das Projekt („Software-Engineering-Simulation durch animierte Modelle“) hat das Ziel, ein Software-System zu schaffen, das auf einer Workstation läuft und von einer Person, dem *Spieler*, bedient wird. Der Spieler wird durch die Mitteilungen des SESAM-Systems mit Informationen über ein Software-Projekt versehen. Er kann den Verlauf dieses Projekts in ähnlicher Weise beeinflussen wie bei realen Projekten der Projektleiter und es dadurch mehr oder minder gelingen oder scheitern lassen.

Bei der Ausbildung im Flugsimulator findet keine reale Flugbewegung statt, nur die Daten einer simulierten Bewegung werden erzeugt. Ebenso entsteht beim Spiel mit SESAM keine Software, nur Daten der Software, z.B. quantitative und qualitative Merkmale ihrer Komponenten, werden aus dem Spielverlauf berechnet.

SESAM kann aus verschiedenen Perspektiven beschrieben und präsentiert werden:

A. Der Blickpunkt des Spielers ist klar und lädt zur Identifikation ein („das möcht‘ ich auch spielen“).

B. Aus der Sicht des forschenden Ingenieurs stellt SESAM, wenn es im Spiel ein plausibles Verhalten zeigt, eine kompakte Codifizierung wichtiger Gesetzmäßigkeiten im Software Engineering dar.

(A) SESAM als Spiel

Wenn uns ein solches System zur Verfügung steht, dann können wir auch Dinge unterrichten, die in der traditionellen Weise so gut wie gar nicht zu vermitteln sind. Denn die reale Welt der Software-Bearbeitung ist im Sinne der reinen Lehre überwiegend durch „Schmutzeffekte“ geprägt. Alle möglichen persönlichen Beweggründe, überraschende Ereignisse und zufällige, aber kaum zu ändernde Randbedingungen prägen die Resultate oft stärker als rationale Entscheidungen.

Solche Effekte können wir simulieren und damit erfahrbar machen. SESAM ist also ohne Frage attraktiv als Lehrmittel, und das nicht nur an der Hochschule, sondern überall, wo Software entwickelt wird.

(B) SESAM als Modell

Über Software Engineering gibt es seit einigen Jahren eine ganze Reihe teilweise dicker Bücher. Man sollte also meinen, daß eine ganze Menge Wissen verfügbar sein müßte. Das gilt aber nur mit erheblichen Einschränkungen. Wir wissen heute vor allem, wie man *nicht* vorgehen sollte, und wir können unsere Erfahrungen in Aussagen kleiden, die nach ihrer Präzision eher der Medizin des 19. Jahrhunderts denn der Physik des 20. ähneln.

In diesem Sinne ist SESAM das Modell, das die „Bauernregeln“ des Software Engineerings präzisiert und quantifiziert, so daß aus konkreten Daten erstmals konkrete Schlüsse gezogen werden können. Hier liegt der wissenschaftliche Reiz und die akademische Herausforderung des Projekts.

3. Der Stand mit SESAM-1

Darum haben wir uns bislang kaum mit Fragen befaßt, die bei der Entwicklung eines Abenteuerspiels auf dem Rechner scheinbar vorrangig sind, beispielsweise mit der möglichst luxuriösen Benutzerschnittstelle. Die meiste Zeit ist damit vergangen, die

abstrakte Architektur des Systems zu schaffen. Damit ist die Beziehung zwischen Modell und Simulator gemeint.

Wir arbeiten an einem ausführbaren Modell, von dem wir wissen, daß es alles andere als perfekt ist. Wir brauchen darum einen Simulator, der durch das Modell quasi parametrisiert wird. Nur so ist eine schnelle Evolution der Modelle möglich. Aber das Prinzip der Parametrisierung schränkt die Freiheit des Modell-Schöpfers ein. Was *nicht* durch Parameter gesetzt werden kann, das ist durch den Simulator vorgegeben und nur mit sehr großem Aufwand änderbar. Das nach drei Prototypen jetzt fertiggestellte System SESAM-1 ist nach unserer Einschätzung für einige Jahre als Werkzeug tragfähig. Die Entwicklung der Modelle ist zurückgeblieben, denn bislang hatten wir keine Möglichkeit, neue Modelle in kurzer Zeit zu formalisieren und zu erproben.

Wir stehen also bei SESAM heute an einem Meilenstein: Wir zeigen das Werkzeug und seine Komponenten als wichtiges Halbfabrikat; in den nächsten Monaten und Jahren werden wir vor allem die Evolution des Modells forcieren.

4. Die Modellierung des Unbekannten

Bei der Konzeption von SESAM standen wir immer wieder vor einem Dilemma: Simulieren kann man alles, was man gut verstanden hat. Der Prozeß der Software-Entwicklung ist aber keineswegs gut verstanden, und so ist er auch der Formalisierung, damit der Simulation entzogen. Andererseits ist diese Situation in der Wissenaschaft nicht ungewöhnlich: Gerade durch eine – unzulängliche – Präzisierung des Problems wird sichtbar, wie eine bessere aussehen könnte.

Hier ist ein Zitat aufschlußreich: Hj. Siegenthaler, Prof. für Wirtschaftsgeschichte am Sozioökonomischen Seminar der Universität Zürich, schreibt in der NZZ vom 16.10.1993 in einem Artikel zur Vergabe des Nobelpreises für Wirtschaftswissenschaften („Neuer Blick in die Geschichte: Die innovativen Ansätze Robert Fogels und Douglass C. Norths“):

> Fogel *quantifiziert* systematisch alle Feststellungen, die er zur Begründung seiner Vorstellungen trifft. Er tut dies auch dort, wo die Datenlage seinen Quantifizierungsversuchen nicht eben entgegenkommt. Dabei rückt er die Bedeutung statistischer Verfahren aus dem Zwielicht zweifelhafter Wahrheitsansprüche sehr entschieden heraus: Quantifizierung begründet keine Wahrheit, aber wer auf sie verzichtet, schreckt davor zurück, sich einer immerhin *kritisierbaren «Wahrheit»* überhaupt zu stellen.

Dies ist auch unser Ansatz: Wir wissen sehr wenig über die Zusammenhänge im Software Engineering. Darum simulieren wir sie.

5. Überblick zu den Beiträgen

Aus der Sicht des Spielers ist SESAM einfach ein großes, komplexes System. Aus der Sicht der Entwickler zerfällt es in viele Komponenten, deren Abgrenzung uns beträchtliche Mühe gemacht hat und die wir heute als ein wesentliches Resultat unserer Arbeit betrachten.

Die folgende schematische Darstellung zeigt die Gliederung und die Themen der Beiträge.

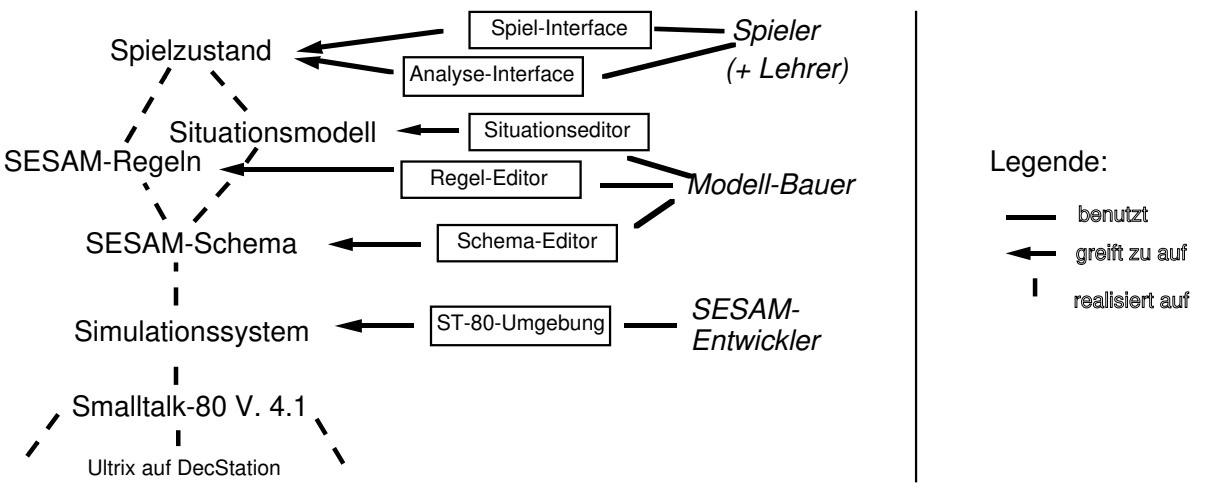
Kurt Schneider befaßt sich mit der Gliederung selbst, dann vor allem mit dem Zusammenspiel von Schema, Regeln und Situationsmodell.

Jinhua Li geht auf die Realisierung des Simulationssystems ein.

Anke Drappa diskutiert die verschiedenen Möglichkeiten, Informationen für den Modellbauer zu beschaffen, also die Quellen für ein Modell.

Marcus Deininger betrachtet SESAM aus der Sicht des Lehrers, also den Einsatz von SESAM.

Jürgen Schwille präsentiert einen wichtigen Implementierungsaspekt, den generischen Editor vis-A-vis, der den drei Editoren für den Modell-Bauer zugrundeliegt.



Teil 3

SESAM – die konzeptionelle Basis

Kurt Schneider

Zusammenfassung

In diesem Beitrag werden die grundlegenden Konzepte von SESAM vorgestellt. Zunächst gehe ich noch einmal kurz auf die Idee von SESAM ein und grenze sie von Prozeßmodellierung ab: In SESAM wird deskriptiv (beschreibend) modelliert, nicht normativ (vorschreibend).

Im zweiten Kapitel werden grundsätzliche Entscheidungen über SESAM vorgetragen. Thesenartig wird jeweils eine Eigenschaft von SESAM in den Raum gestellt. Anschließend wird sie diskutiert, die Konzepte, die diese Eigenschaft hervorbringen, werden genannt.

Eine zentrale Rolle in SESAM nimmt die Modellbildung ein. Für SESAM wurde eine neuer Modellierungsansatz entwickelt, der hier als "Effektorientierte Modellierung" bezeichnet wird. Die Grundzüge dieses Ansatzes werden im dritten Kapitel erörtert.

Das vierte Kapitel geht kurz auf die Architektur des SESAM-Programmsystems ein. Es zeigt sich, daß diese Architektur den konzeptionellen Aufbau von SESAM-Modellen widerspiegelt. Zum Abschluß gebe ich einen kurzen Überblick über den gegenwärtigen Entwicklungsstand von SESAM.

Dieser Beitrag liefert einerseits die konzeptionelle Grundlage, auf der die folgenden Aufsätze basieren. Andererseits gibt der Beitrag einen groben Überblick über die Themen, die in den anderen Beiträgen im einzelnen behandelt werden.

1. Die Idee: Deskriptive Modelle

In SESAM werden Software-Projekte simuliert. Dazu müssen sie modelliert werden – und zwar so, wie sie sind: deskriptiv.

Auch in anderer Bedeutung wird mitunter von „Modellierung“ oder „Simulation“ von Software-Prozessen gesprochen. Zunächst sollen zwei grundsätzliche Feststellungen getroffen werden, die die Position von SESAM charakterisieren: SESAM leitet niemanden an, und SESAM ist kein Programmgenerator.

1.1 SESAM versus Prozeßmodellierung: Abbild oder Vorbild?

Mit SESAM sollen Software-Projekte modelliert werden. Die Projekte werden so modelliert, wie sie *sind* – nicht wie sie sein *sollten*. Ludewig (1989) hat zwischen Modellen mit Abbildcharakter und solchen mit Vorbildcharakter unterschieden. SESAM-Modelle sind Abbilder realer Projekte. Alle Fehlentwicklungen, überraschenden Störungen und menschlichen Insuffizienzen werden nach Möglichkeit modelliert. So kann ein angehender Projektleiter bzw. Spieler an den simulierten Projekten realistische Erfahrungen sammeln.

Es gibt seit einigen Jahren eine Bewegung im Software Engineering, die als Prozeßmodellierung (process modeling) bezeichnet wird; Curtis *et al.* (1992) geben einen guten Überblick über dieses Gebiet. In der Prozeßmodellierung werden die Abläufe in Software-Projekten ebenfalls modelliert – aber die Modelle haben Vorbildcharakter. Einschlägige Systeme wie MARVEL (Kaiser *et al.*, 1993) oder Merlin (Peuschel/Schäfer, 1992) unterstützen die Ausführung von Prozeßmodellen. Wird ein Prozeßmodell ausgeführt, so läuft im Prinzip ein Programm ab: das Modell. Nur werden nicht alle Teile des Programms vom Computer ausgeführt. An vielen Stellen werden menschliche Bearbeiter „wie Unterprogramme aufgerufen“ und zu einer Leistung veranlaßt. Dann müssen sie z.B. zu einer Systemspezifikation einen Modulentwurf erstellen. Sind sie damit fertig, melden sie es dem System. Entsprechend der im Modell festgelegten Abhängigkeiten wird dann der Modulentwurf an einen anderen Bearbeiter zur Kontrolle oder zur Implementierung weitergereicht. In Prozeßmodellen ist also festgelegt, welche Aktivitäten-Reihenfolge einzuhalten und welche Bedingungen zu beachten sind. Animierte Prozeßmodelle leiten ein Software-Projekt an, indem sie die Tätigkeiten der Entwickler koordinieren. Die Modelle werden zu diesem Zweck interpretiert. Ist der Prozeß ungünstig modelliert, leitet er die Entwickler schlecht an. Mit der Prozeßmodellierung will man Software-Projektleiter entlasten, indem man ihnen die Koordination aus der Hand nimmt.

SESAM ist in diesem Sinn *kein* Ansatz zur Prozeßmodellierung: Projektleiter lernen an SESAM, Projekte besser zu leiten, Fehlentwicklungen schneller zu erkennen; aber sie bleiben Projektleiter, mit allen ihren Aufgaben. SESAM wird einem Projektleiter als "Abenteuerspiel" präsentiert, in dem

er sich bewähren muß. Aufgrund des komplizierten, quantitativen Modells vereint SESAM Elemente von Planspielen und Adventure Games, was in Schneider (1993a) ausgeführt wird. SESAM-Modelle sind Abbilder, Prozeßmodelle sind Vorbilder.

1.2 Reale und simulierte Projekte

In SESAM werden Software-Projekte simuliert. Dazu werden Dokumente, Code und beteiligte Personen modelliert. Das Modell einer Person ist – grob gesprochen – ein Smalltalk-80 - Objekt mit einigen Variablen wie Name, Monatsgehalt, analytische und synthetische Fähigkeiten. Natürlich erfaßt dieses Modell nur sehr wenige Facetten einer Person, es vereinfacht und abstrahiert stark. Nur diejenigen Merkmale sind im Modell berücksichtigt, die für Software-Projekte am wichtigsten sind. Die meisten Eigenschaften realer Personen tauchen im Modell überhaupt nicht auf, sie werden weggelassen.

Ebenso wie Modelle von Personen natürlich keine wirklichen Personen sind, sind auch Modelle von Dokumenten keine Dokumente, und Modelle von Programmcode sind kein Programmcode. Das Modell besteht nur aus Vertretern für Personen, Dokumenten und Code. Jeder dieser Vertreter hat einige Attribute, die ihn charakterisieren.

Stachowiak (1972) hat die Modellabbildung wie in Abb. 1 dargestellt: Von einem realen Projekt werden nur wenige Eigenschaften für relevant erachtet, die meisten sind irrelevant und werden einfach weggelassen. Die relevanten werden ins Modell *abgebildet* – und nicht einfach *übernommen*. Aus den komplizierten Fähigkeiten einer Person wird das Attribut "analytische Fähigkeiten" mit dem Wert 1,1 kondensiert. Das soll bedeuten: Die Person ist um 10% begabter als der Durchschnitt. Der Inhalt von Dokumenten wird ebenfalls *abgebildet*: Auf eine Menge von Software-Quanten. Jeder Software-Quant hat eine individuelle, aber anonyme Identität. Er steht für irgendeine Anforderung des Kunden. Die Information, um welche Anforderung es sich dabei handelt, ist durch die Modellabbildung verloren gegangen: Für den Projektleiter ist in erster Linie wichtig, wie viele Anforderungen ein Kunde hat – nicht, wie diese lauten. Jeder Software-Quant macht gleich viel Aufwand. Marcus Deininger geht auf die Idee der Software-Quanten ein.

Modellierte Dokumente haben also keinen für reale Menschen lesbaren Inhalt. Das gilt auch für Modell-Code; er ist weder lesbar noch ablauffähig: schließlich handelt es sich eben nicht um Code, sondern um ein Modell von Code. SESAM ist kein Programmgenerator. Alle relevanten Projektbestandteile werden bei der Modellbildung auf einfache Objekte mit einigen wenigen Instanzvariablen-Werten abgebildet.

2. Grundkonzepte

SESAM zeichnet sich durch eine Reihe charakterisitischer Eigenschaften aus. Diese werden zu Beginn der folgenden Unterabschnitte thesenartig angegeben und dann erläutert. Schließlich wird zu jeder Eigenschaft angegeben, durch welche Konzepte sie in SESAM erreicht wird.

2.1 Der Zweck von SESAM-Modellen

> SESAM ist ein Lehrspiel für Projektleiter; es ist aber auch ein Forschungswerkzeug für Software Engineering.

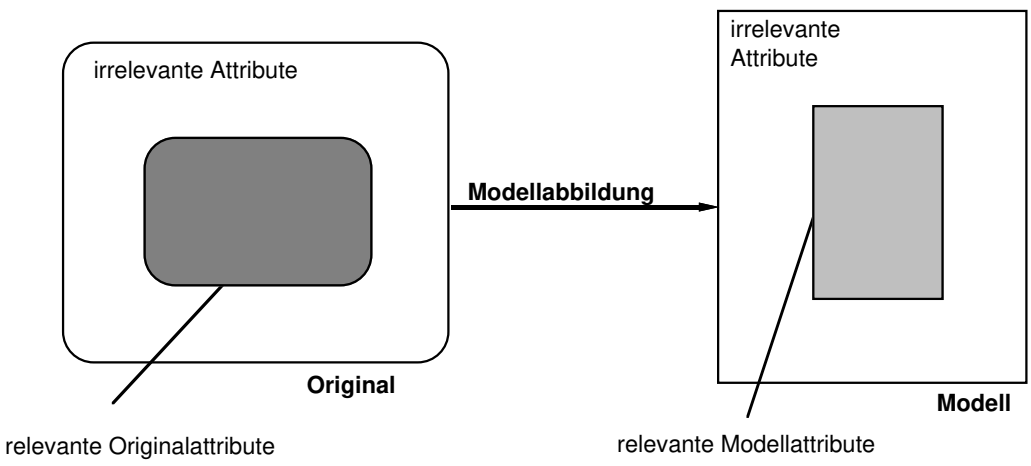


Abb. 1: Bei der Modellabbildung können irrelevante Attribute weggelassen werden

Die Vision eines computergestützten Adventure-Games für Software-Projektleiter hört sich zunächst ganz einfach an. Aber was sind die relevanten Aspekte eines Software-Projekts? Welche können weggelassen werden? Offensichtlich muß man zuerst klären, was man mit SESAM-Modellen genau erreichen will. Dann kann man entscheiden, welche Aspekte für diesen Zweck relevant sind.

Konzepte

- Wir wollen mit SESAM Abhängigkeiten, Beobachtungen und dynamische Effekte sammeln, die den Erfolg von Software-Projekten wesentlich beeinflussen.
- Sie werden in einer eigenen Notation einheitlich dargestellt und in einem Modell integriert.
- Das Modell ist anschaulich für Menschen und zugleich animierbar für den Computer.
- Das Modell kann als Adventure Game präsentiert werden.
- Durch Experimente mit dem Modell sollen *Projektleiter* lernen, reale Projekte besser zu führen (Abb. 2, kleine Schleife). *Wir* wollen dabei lernen, wovon die Entwicklung von Software-Projekten wirklich abhängt (Abb. 2, große Schleife).

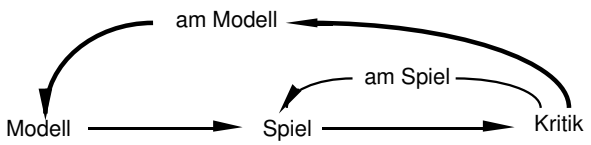


Abb. 2: Lernen durch Kritik

2.2 Lernziele

> Spieler sollen mit SESAM *bestimmte* Dinge lernen können. Darauf müssen die Modelle zugeschnitten sein.

An einem simulierten Projekt kann man nicht alles erfahren und lernen, was man in realen Projekte erfahren könnte. Schließlich sind viele Eigenschaften als irrelevant weggelassen worden. Man muß sich klar machen, welche Einsichten und Erfahrungen am Modell möglich sein sollen. Die dazu nötigen Aspekte darf man dann nicht weglassen: Sie sind relevant.

Lernziele von SESAM

Spieler sollen durch den Umgang mit SESAM einige Einsichten vermittelt bekommen:

- In Software-Projekten gibt es viele, komplizierte Zusammenhänge, die man auf den ersten Blick nicht erkennen kann. Projektdauer, -preis und Produktqualität hängen beispielsweie untereinander und vom Betriebsklima ab. Das Betriebsklima hängt wiederum vom Zeitdruck ab. Wenn der Projektleiter eine Anordnung trifft, erreicht er damit nicht nur die gewünschte Änderung: Viele Seiten- und Folgeeffekte können ausgelöst werden und die angestrebte Verbesserung sogar wieder zunichte machen.
- Software ist immateriell. Es ist für den Projektleiter schwierig einzuschätzen, wie weit das Projekt schon gediehen ist; auch das Betriebsklima kann er nicht messen, sondern nur vage einschätzen. Nur wenn er ausdrücklich Maßnahmen zur Fortschrittskontrolle (wie Software-Metriken) einsetzt, erhält er etwas mehr Informationen. Das kostet aber Zeit und Geld.
- Die meisten Entscheidungen in Software-Projekten muß der Projektleiter fällen, obwohl ihm viele Planungsgrundlagen fehlen: Er weiß nicht, wann seine Mitarbeiter krank werden; er weiß nicht, wie weit sie bereits gekommen sind und wie gut die Qualität des entstandenen Software-Produkts ist. Er muß unter Unsicherheit entscheiden.
- Wenn ein Projektleiter nicht plant, weiß er auch über die angestrebte Entwicklung seines Projekts nicht Bescheid. Um das zu verhindern, muß er planen; und zwar mit Puffern, um den zahlreichen Unsicherheiten Rechnung zu tragen. Tut er es nicht, gerät seine Projektleitung immer mehr zu reaktivem Krisenmanagement.
- Es kommt in Software-Projekten darauf an, das Richtige zu tun. Es kommt aber auch darauf an, das Richtige angemessen zu dosieren: So reicht es nicht aus, Anforderungsanalyse überhaupt durchzuführen. Ihr muß statt dessen angemesse Zeit und genügend Aufwand gewidmet werden.

Konzepte

SESAM-Spieler sollen diese Erkenntnisse sammeln können. Die Modelle weisen die entsprechenden Eigenschaften auf:

- Sie sind quantitativ.
- Sie sind dynamisch, lassen also die Auswirkungen versteckter Zusammenhänge teilweise erst im Laufe der Zeit sichtbar werden.
- Sie sind interaktiv und bieten dem Spieler immer wieder Eingriffsmöglichkeiten, mit denen er den Fortgang des simulierten Projekts beeinflussen kann.
- Der Spieler kann das simulierte Projekt aktiv leiten, er kann es aber auch vernachlässigen. Dann gerät es mehr und mehr außer Kontrolle.
- Mögliche Projektleiteraktionen und Modelle sind feingranular und konkret; der Spieler kann mit ähnlichen Aktionen ins simulierte Projekt eingreifen, wie ein realer Projektleiter in sein Projekt. Insbesondere muß er mit einzelnen, indivi-

duellen (simulierten) Mitarbeitern umgehen, über ihre Aktivitäten nachdenken und den Überblick über die entstehenden (simulierten) Dokumente behalten.

2.3 Einbettung von SESAM in die Projektmitarbeiter-Ausbildung

> Spielen am Modell allein genügt nicht. Es ist nur komplementär zu praktischer Erfahrung und theoretischem Unterricht einzusetzen und muß in ein Ausbildungs-Gesamtkonzept integriert werden.

Die genannten Lernziele decken nicht alle Fertigkeiten ab, die ein Projektleiter haben muß. Mitarbeiterführung wird nur in sehr groben Zügen abgedeckt, Gesprächsführung kann man z.B. nicht im Spiel lernen. Auch reicht es nicht aus, ein oder zwei Mal zu spielen, um die oben genannten Lernziele zu erreichen. Dazu müssen die Spieler vielmehr zusammen mit einem Betreuer die simulierten Projekte nachvollziehen und besprechen. Man braucht also ein didaktisches Konzept für SESAM. SESAM muß seinen Teil dazutun, indem es die technischen Voraussetzungen bietet, um Spiele wiederholen und kritisch diskutieren zu können.

Konzepte

- In SESAM werden alle Simulationsläufe bzw. Spiele aufgezeichnet und können beliebig oft wiederholt werden. Dabei treten auch "überraschende" Ereignisse immer wieder zur selben Modellzeit ein.
- Es gibt verschiedene Simulationsmodi: Im Spielmodus hat der Spieler nur sehr wenige Informationen (vgl. die Lernziele). Im Analyse-Modus bekommt er zusätzliche Informationen, einige versteckte Zusammenhänge werden aufgedeckt. Im Betreuer-Modus kann man alle im Modell vorhandenen Größen beobachten und sieht genau, was sich im Modell tut.
- In der Ausbildung sollten sich SESAM-Spielphasen mit Lehreinheiten abwechseln. Erst durch Rückmeldungen und Kritik können viele Spieler ihre eigenen Fehler erkennen. Dazu kann man im Analyse- und schließlich im Betreuermodus immer mehr Informationen zeigen, damit schrittweise die Dynamik des simulierten Projekts erschließen.

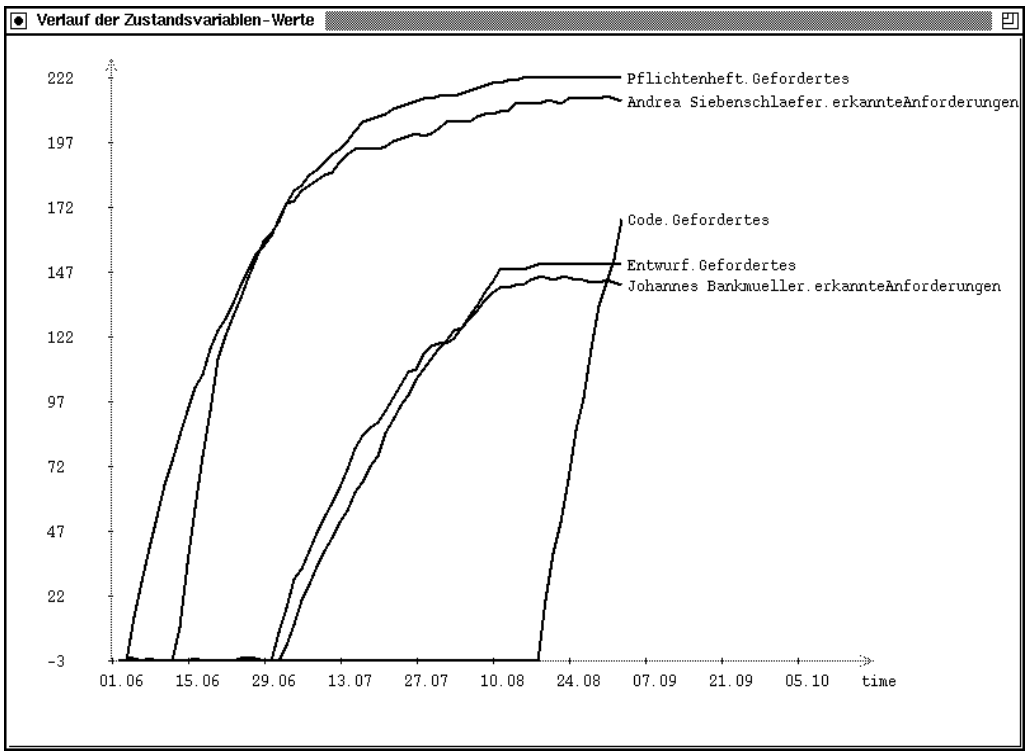


Abb. 3: Analyse eines Simulationslaufs durch Studium der Attributwertverläufe

2.4 Modellierungsansatz statt festem Modell

> Es gibt in SESAM nicht "das Modell" von Software-Entwicklung schlechthin, sondern viele konkurrierende und einander ergänzende Modelle.

So wie es nicht nur eine Art von Software-Projekten gibt, kann es nicht ein einziges, immer gleiches Modell aller dieser Projekte geben. Auftragsprojekte laufen anders ab als in-house-Projekte; Verwaltungssoftware wird anders erstellt, getestet und gewartet als Echtzeitsoftware. Jede Firma hat ihre eignen Randbedingungen, Standards und Vorgehensmodelle. Die individuellen Mitarbeiter entscheiden über Erfolg oder Mißerfolg gerade kleiner Projekte.

Es gibt verblüffenderweise trotzdem Ansätze, um mit einem Abbild-Modell Aussagen zu gewinnen, die auf *alle* (oder zumindest sehr viele) Software-Projekte anwendbar sein wollen (Abdel-Hamid, 1991). Das dort verwendete Modell wurde mit einem NASA-Projekt abgeglichen.

In SESAM gehen wir einen anderen Weg. Denn selbst, wenn man ein *einziges*, konkretes Projekt modelliert, werden verschiedene Modellierer zu ganz verschiedenen Ergebnissen kommen. Darin drückt sich der Mangel an allgemein anerkannten Erkenntnissen über Zusammenhänge und dynamische Effekte in Software-Projekten aus. Jeder Modellierer ist weitgehend auf eigene Intuition und Erfahrung angewiesen; er muß viele Hypothesen in sein Modell einbauen. Dabei wird er Fehler machen, Hypothesen werden sich als inkonsistent oder schlicht falsch herausstellen. Man muß mit ständigen Modellveränderungen rechnen.

Konzepte

- In SESAM ist nicht *ein* Modell fest implementiert, sondern es steht ein ganzer Modellierungsansatz zur Verfügung: ein Modell-Baukasten, mit dem man schnell und einfach Modelle erstellen, verändern und anpassen kann.
- Daher sollte man unterscheiden zwischen dem modellunabhängigen SESAM-Programmsystem und den SESAM-Modellen, die damit erstellt, verwaltet und animiert werden.

Im folgenden Abschnitt werden Konsequenzen aus dieser Entscheidung für einen Modell-Baukasten gezogen: Wenn Modelle ständig geändert werden sollen, muß das leicht und einfach gehen.

2.5 Flexible Modelle

> SESAM-Modelle laden zu Diskussion und Modell-Änderungen geradezu ein.

Der Modell-Baukasten gibt die Bausteine vor, aus denen Modelle bestehen. Sie sind so gestaltet, daß sie einfach auszuwechseln und zu verändern sind. Dazu werden die Modelle aus Teilen aufgebaut, die jeweils weitgehend unabhängig voneinander bearbeitet werden können. Nicht einmal das Repertoire der im Modell vorkommenden Objekte ist durch den Modellierungsansatz vorgegeben: Der Modellierer soll selbst entscheiden, welche Komponenten in seinen Modellen vorkommen, welche charakteristischen Eigenschaften sie haben und wie sie sich dynamisch verhalten. Besonders zur Darstellung der Dynamik muß man sich etwas einfallen lassen: Modelle bestehen nicht einfach aus Programmstücken; damit könnte man zwar bequem dynamische Veränderungen darstellen. Sie wären aber nur sehr schwer zu verstehen und zu warten. Modellierer sollen nicht Programmierer sein. Die Struktur der Modelle wird daher graphisch notiert, nur Details als Text angegeben.

Konzepte

- In einem Entity-Relationship-Schema wird festgelegt, welche Objekte und Beziehungen als rele-

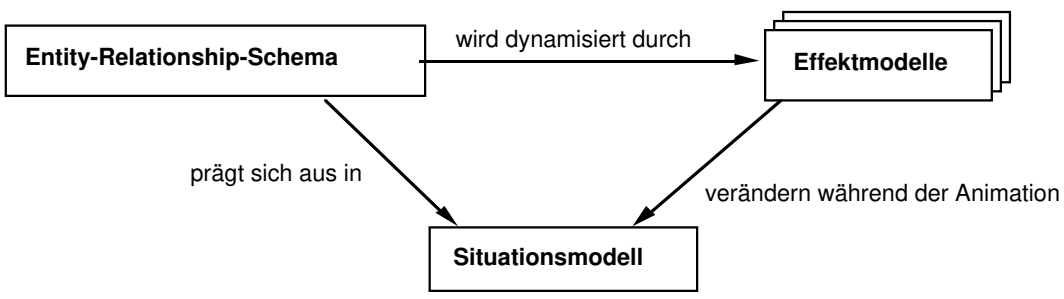


Abb. 4: Schema, Situationsmodell und Regeln

vant erachtet werden. Der Modellierer braucht nur das Schema zu ändern, um das Repertoire an verfügbaren Komponenten zu verändern.

- Die Ausgangssituation eines Projekts wird modelliert, indem eine Ausprägung des Schemas erstellt wird. Darin sind als Objekte z.B. einzelne Mitarbeiter, Dokumente und Computer enthalten. Dazu kommen vielerlei Beziehungen, in denen diese Objekte zu Beginn des Projekts stehen.
- Die Dynamik – als besonders komplizierter Modellteil – wird in Effekte zerlegt, die mit einer besonderen Art von Regeln (SESAM-Regeln) dargestellt werden (siehe unten).

Abb. 4 zeigt die Teile eines dynamischen Modells im Überblick: Zuerst wird in einem Schema festgelegt, welche Komponentenarten es gibt. Dann wird eine Ausgangssituation als Ausprägung des Schemas aufgebaut. Die Dynamik wird durch eine Menge von Regeln beschrieben.

Will man einen Aspekt der Dynamik ändern, kann man einzelne Regeln austauschen oder verändern. Die Interpretation und Kombination aller Regeln übernimmt das SESAM-Programmsystem. Dadurch kann sich der Modellierer ganz auf inhaltliche Fragen seines Modells konzentrieren. Er braucht überhaupt nicht zu programmieren. Damit sind die Voraussetzungen für eine schnelle Überarbeitung von Modellen geschaffen.

2.6 SESAM in anderen Einsatzbereichen

> SESAM ist durch seine Flexibilität ein universell einsetzbarer Modellierungsansatz. Es ist nicht auf Software-Projekt - Simulation beschränkt.

Wenn im folgenden Ausschnitte aus einem SESAM-Modell gezeigt werden, so haben sie Beispielcharakter: Man kann mit SESAM auch ganz andere Projekte, und sogar ganz andere Themengebiete modellieren als Software-Projekte: Man zeichnet ein Schema dieses anderen Wirklichkeitsausschnitts, bildet eine Anfangssituation und legt in Regeln die Dynamik fest. Das funktioniert in vielen Bereichen z.B. der Biologie, der Soziologie – überall dort, wo *Effekte* als Träger der Dynamik identifiziert werden können und eine sinnvolle Granularität bilden (siehe Kapitel 3).

Bei der Entwicklung von SESAM als Projekt-

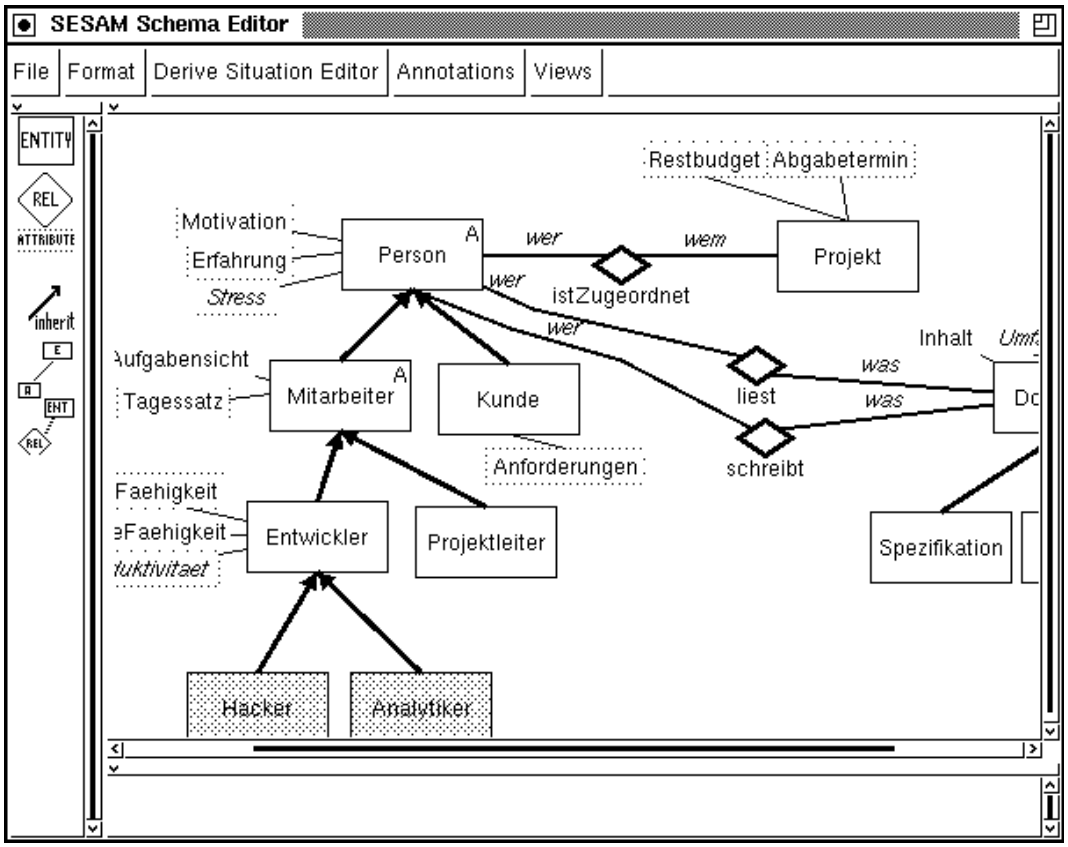


Abb. 5: Ein Schema im Schema-Editor

Simulator war diese allgemeine Einsetzbarkeit nicht unbedingt angestrebt worden. Die letztlich eingesetzten Konzepte sind aber keineswegs auf dieses eine Anwendungsgebiet eingeschränkt.

2.7 Verständliche Notation und Werkzeuge für Modellbildung

> Modelle werden so notiert, daß sie sowohl für menschliche Betrachter als auch für das Simulationswerkzeug verständlich sind.

Derzeit gibt es kaum allgemein anerkannte, quantitative Beobachtungen über Software-Projekte. In jedem quantitativen Modell stecken daher zahlreiche Hypothesen. Es ist in diesem Stadium der Forschung unverzichtbar, die Hypothesen explizit nennen zu können. Modelle können nur verbessert werden, wenn ganz klar ist, welche Annahmen darin stecken. Deshalb ist die Darstellung der Modelle so wichtig.

Es ist nicht akzeptabel, ein Modell zwar graphisch zu zeichnen, es dann aber noch einmal manuell in eine lineare Notation transformieren zu müssen. Dieser Vorgang wäre nicht nur zu aufwendig, er würde beinahe sicher auch zu Fehlern und Inkonsistenzen führen. Dadurch würde die Validierung zusätzlich erheblich erschwert.

Konzepte

- Modelle sind modularisiert: Ein dynamisches Modell besteht aus einem Schema, einem Situationsmodell und einer Menge von Regeln. Jeder dieser Teile ist für sich genommen überschaubar. Ferner existieren einfache Prinzipien, wie die Teile zusammenwirken.
- Jede Regel steht für einen Effekt oder einen Zusammenhang im Projekt. Ein dynamischer Effekt ist eine logisch zusammengehörige Menge von Abhängigkeiten, Einflüssen und Veränderungen an den Attributen mehrerer beteiligter Komponenten (siehe Kapitel 4).
- Es gibt eine halb-graphische Notation für alle Teile von dynamischen Modellen. Strukturen werden graphisch notiert, Details durch Formeln oder textuelle Annotationen.
- Es gibt halb-graphische Editoren für Schema, Situationsmodell und Regeln. Durch Veränderung von Diagrammen werden intern computerinterpretierbare Modelle verändert. Die Diagramme sind für Menschen verständlich; die in-

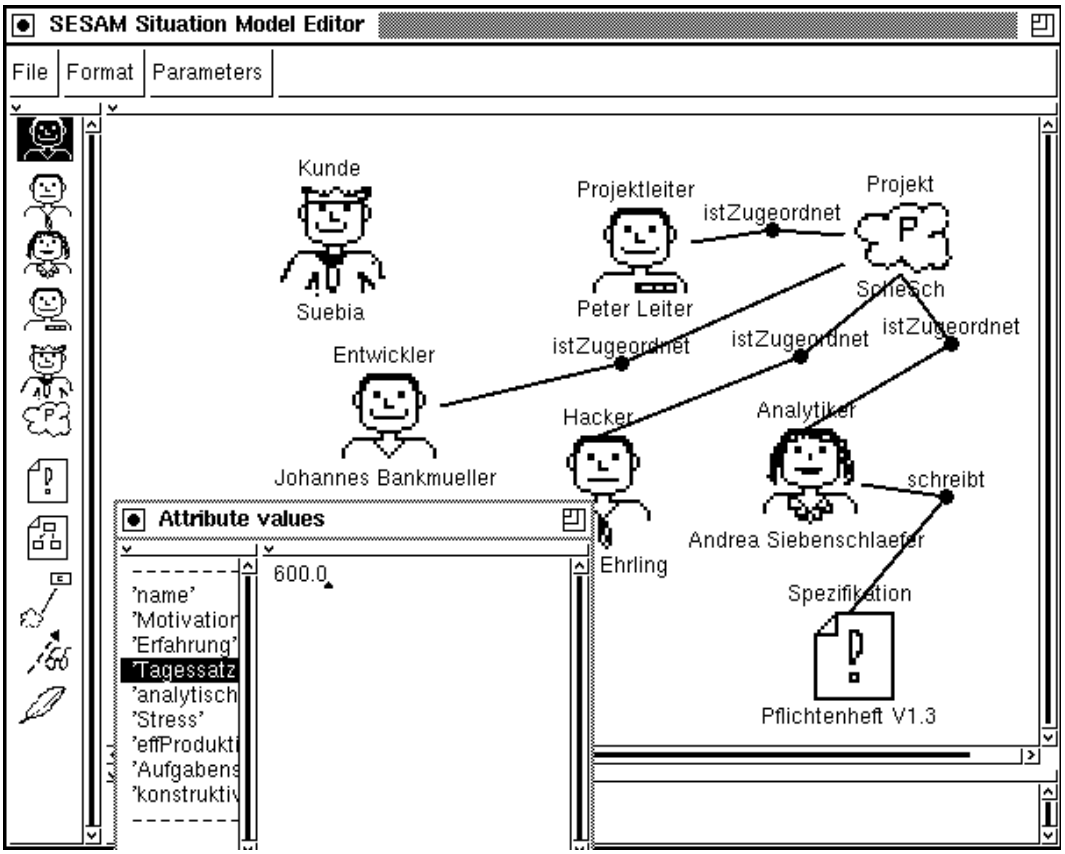


Abb. 6: Ein Situationsmodell im graphischen Editor

terne Darstellung ist unmittelbar animierbar.

- Effekte sind eine Gliederungseinheit, die den Blick von Modellierern und Projektleitern auf das für sie Wesentliche lenken: wiedererkennbare, logisch zusammengehörige Komplexe von Veränderungen.
- Die graphisch repräsentierte Struktur ist leicht verständlich. Damit sind die Grundideen eines Modells mit wenig Aufwand vermittelbar. Für quantitative Details muß man sich dann näher mit den Modellen beschäftigen.

2.7 Die Spieloberfläche: Einfach und klar

> Während der Modellaufbau graphisch unterstützt wird, ist die Spieloberfläche bewußt einfach gehalten: Eine reine Textschnittstelle.

Die meisten der bisherigen Konzepte bezogen sich auf den Aspekt des Modellaufbaus. Im Spiel werden Modelle eingesetzt. Während der Modellaufbau durch eine Familie graphischer Editoren unterstützt wird, wirkt die Spieloberfläche eher spartanisch: Der Projektleiter sucht seine Aktionen aus einem sehr vollen, überladenen Menü aus, gibt Parameter an und erhält Rückmeldungen in einem Textfenster. Von den Modellen ist im Spiel nicht viel zu sehen. Sie laufen völlig im Verborgenen ab.

Die Menüauswahl des Projektleiters geht als Ereignis ins Modell ein, das die Modellentwicklung beeinflussen kann. Über die Entwicklung wird der Projektleiter nur durch kurze Texte unterrichtet, die von den Regeln ins Textfenster geschrieben werden. Abb. 8 zeigt einen Ausschnitt daraus.

Die Interaktion ist absichtlich so einfach ausgefallen. Als Eingabe für Projektleiteraktionen wäre eigentlich Freitext am besten geeignet: Der Projektleiter/Spieler müßte selbst und völlig ohne Anleitung oder Hilfe seine Anordnungen formulieren. Nichts würde ihm einen Hinweis geben, was er jeweils tun sollte – oder auch nur, was er überhaupt tun kann. Allerdings ist Freitext außerordentlich schwer zu interpretieren. Das überladene, absichtlich unübersichtliche Menü stellt einen Kompromiß dar: Es leitet nur wenig an, erleichtert aber die Interpretation von Aktionen ungemein.

Eine graphische Spieloberfläche, wie man sie von Adventure Games wie „Larry Leasure in the Land of Lounge Lizzards“ her kennt, habe ich ausprobiert. Inzwischen bin ich aber der Überzeugung, daß die Graphik SESAM eher schadet als nützt: Die Spieler glauben, mit einem Blick erfassen zu können, wodurch eine Situation gekennzeichnet ist. Wenn sie die Situation nicht sehen, ist diese Gefahr kleiner, es bleibt die nagende Ungewißheit, ob man nicht noch etwas vergessen hat. So soll es sein.

3. Effekt-orientierte Modellierung

Ein eigenes Kapitel ist dem zentralen Modellierungskonzept in SESAM gewidmet: Der Zergliederung eines dynamischen Modells in sogenannte Effekte und Abhängigkeiten.

3.1 Was ist ein Effekt?

Die Dynamik eines SESAM-Modells wird dargestellt

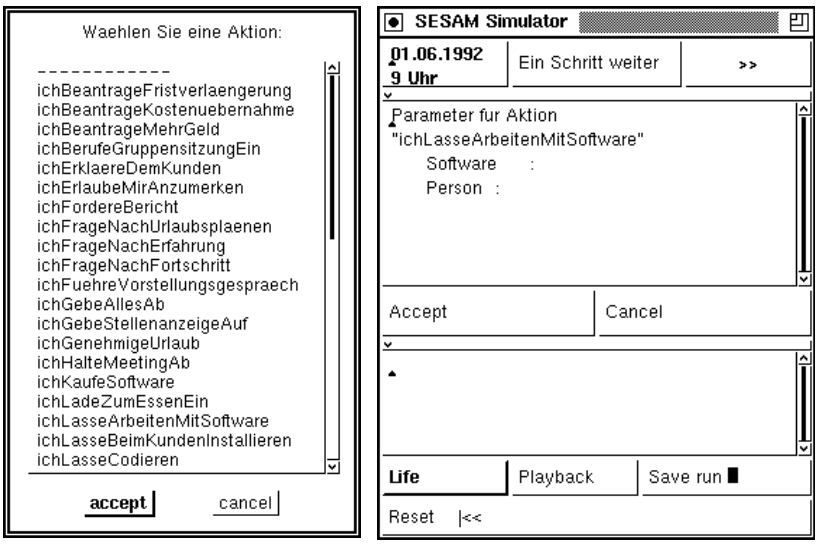


Abb. 7: Menüauswahl und Simulationssteuerfenster

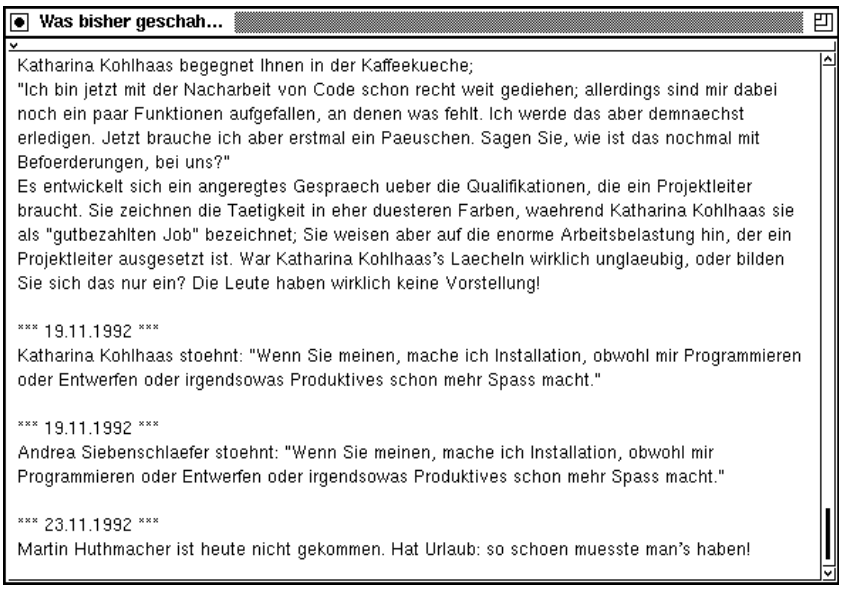


Abb. 8: Das Rückmeldungsfenster

als ein System von Abhängigkeiten und Effekten. Eine der schwierigsten Fragen in SESAM war die Modellierung der Dynamik. In SESAM wird die Dynamik als System sich überlagernder und gegenseitig auslösender Effekte und Abhängigkeiten aufgefaßt. Das Konzept der Effektmodelle ist eine Verallgemeinerung sowohl der objekt-orientierten Modellierung, als auch der aktivitätsorientierten Simulation.

Objekt-orientierte Modellierung

In einem streng objekt-orientieren Modell würde man für alle beteiligten Klassen aus dem Schema festlegen, welches Verhalten ihre Ausprägungen zeigen. Die Dynamik des Gesamtmodells ergibt sich durch das Zusammenwirken der dynamischen Objekte; sie schicken sich gegenseitig Nachrichten zu und fordern sich damit zu Aktivitäten (Verhalten) auf. In diesem Paradigma ist die Dynamik auf alle Objekte und innerhalb der Objekte auf die „Methoden" verteilt. Innerhalb einer Methode (Verhaltenseinheit) legt man genau fest, was mit dem betreffenden Objekt geschieht. Veränderungen an anderen Objekten sieht man dagegen nicht: Ihnen werden Nachrichten geschickt, jedes Objekt kapselt aber alle Änderungen an seinem Zustand und verbirgt sie nach außen. In Schneider (1993) wird SESAM von rein objekt-orientierter Modellierung abgegrenzt.

Aktivitäts-orientierte Simulation

Die aktivitätsorientierte Simulation ist eine Form ereignisorientierter bzw. diskreter Simulation (Page, 1991). In der aktivitätsorientierten Simulation ist die Dynamik eines Systems auf eine Menge von Aktivitäten verteilt. Zu jeder Aktivität gehört eine Ausführungsbedingung. Die Aktivität ist selbst dafür „verantwortlich", ständig zu prüfen, ob diese Bedingung erfüllt ist. Dann wird die Aktivität ausgeführt. Dies resultiert in einer diskreten Zustandsänderung an beliebigen Modellkomponenten. Die Wirkung der Aktivität wird also nicht über „Objekte" verstreut, sondern ist an einer Stelle für alle Objekte gemeinsam festgelegt., die von einer Aktivität betroffen sind: in der Aktivitätsbeschreibung.

Effekt-orientierte Modellierung und Simulation in SESAM

Die Gliederungseinheit der Dynamik in SESAM ist der Effekt. Ein Effekt ist eine Klasse von Begebenheiten, die unter typischen Bedingungen auftreten. Ergibt sich in einem Software-Projekt eine Konstellation, die den Effekt auslöst, so findet eine solche Begebenheit statt. Die Begebenheit verändert den Zustand der am Effekt beteiligten Objekte – der Objekte, die in der Konstellation vorkommen. Die Konstellation kann ein beliebiges Situationsmuster aus dem Schema sein. In einem Effektmodell sind alle Zustandsänderungen an allen Objekten zusammengefaßt, die unter den Effekt fallen. Nicht das Objekt, nicht unbedingt eine bewußte Aktivität, sondern Effekte sind Träger von Dynamik. Anders als in der aktivitätsorientierten Simulation kann die Zustandsänderung auch kontinuierlicher Natur sein: Durch Differenzengleichungen werden allmähliche Attributwertänderungen beschrieben. Sie wirken sich erst aus, wenn Modellzeit verstreicht.

Effekte können sich gegenseitig auslösen. Ein Effekt soll eine wiedererkennbare Einheit sein, die ein Projektleiter auch in realen Projekten bemerken kann. Seine Aufgaben ist es, die jeweils wirksamen

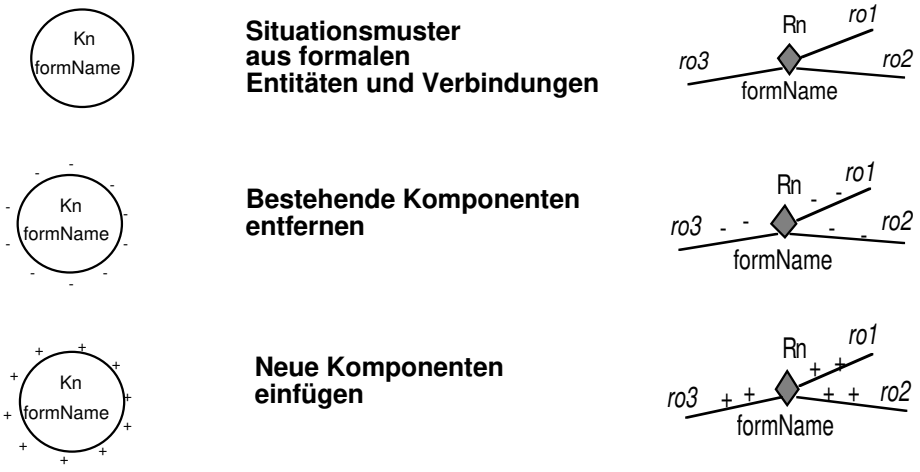


Abb. 9: Prinzip der Notation für Strukturänderungen

Effekt zu erkennen und ihre überlagerte Wirkung abzuschätzen. Das kann er in SESAM lernen. Er muß es meiner Überzeugung nach auch in realen Projekten ständig tun.

Effekt-orientierte Modellierung prägt der Wirklichkeit eine typische Struktur auf; überall sieht man Effekte, komplementiert von Abhängigkeiten. Projektleiter müssen lernen, durch ihre Aktionen indirekt mit Effekten zu jonglieren: wünschenswerte hervorzurufen, schädliche zu erkennen und zu unterbinden. Ungeübte Projektleiter können zunächst nur auf wenige grundlegende Effekte achten. Mit mehr Übung wächst das Repertoire, man hat das Projekt immer besser im Griff. Die effekt-orientierte Sicht der Welt ist der Projektleitertätigkeit angemessen.

3.2 Prinzip: Graphmanipulationsregeln als Effektmodelle

Den Kern eines dynamischen Modells bilden die Regeln, mit denen die Dynamik modelliert wird. Jede Regel stellt einen Effekt oder eine Abhängigkeit dar. Hier kann nur das Prinzip der Regeln vorgestellt werden.

Eine Regel besteht üblicherweise aus einem Bedingungs- und einem Aktionsteil. Im Bedingungsteil ist festgelegt, unter welchen Umständen die Regel anwendbar ist. Im Aktionsteil wird beschrieben, was dann geschieht.

Der Zustand eines SESAM-Modells wird als Graph interpretiert, bestehend aus Projektbeteiligten Objekten (Knoten) und Beziehungen dazwischen (Kanten). Ein Situationsmodell beschreibt einen Zustand. Auf dem Diagramm ist daher ein Graph zu sehen (vgl. Abb. 6).

SESAM-Regeln sind eine spezielle Variante von Graphmanipulationsregeln: Bedingungs- und Aktionsteil bestehen aus Graphmustern. Wird im Situationsmodell ein Teilgraph gefunden, der mit dem Graphmuster des Regel-Bedingungsteils übereinstimmt, so wird das Graphmuster mit diesem Teilgraphen gebunden. Der Teilgraph wird nun ersetzt durch einen anderen Teilgraph, der dem Aktionsteil-Muster entspricht. Dieses Muster wird dann anstelle des Bedingungsteil-Musters in das Situationsmodell eingebettet. Verschiedene Formen von Graph-Grammatiken werden unter anderem in Schneider (1977) und Göttler (1988) detailliert beschrieben.

3.3 Eine Notation für Effekte

Die Idee, Zustandsänderungen als Graphmanipulationsregeln darzustellen, entfaltet ihre Stärken nur dann vollständig, wenn die Veränderungen auch weitgehend graphisch notiert werden. Wenn man von einigen syntaktischen Details absieht, hat die graphische Notation folgende Bestandteile: Ein Situationsmuster wird als Graph aus formalen Komponenten (Kreisen und Verbindungslinien mit Raute in Abb. 9) dargestellt. Veränderungen werden durch + bzw. - Zeichen an formalen Komponenten ausgedrückt: mit "-" markierte Komponenten werden gelöscht, für jede "+" markierte Komponente wird ein neues Objekt oder eine neue Beziehung in den Situationsgraphen eingefügt.

Dieses Konzept ist in SESAM verbunden mit der Möglichkeit, Attributwerte von gebundenen Teilgraphen zu verändern. Der Bedingungsteil besteht also weiterhin aus einem Situationsmuster, das an einen Teilgraphen des Situationsmodells gebunden wird. Im Aktionsteil der Regel wird angegeben, wie sich die Attributwerte ändern. Die Änderungen vollziehen sich entweder auf einen Schlag (Zuweisungen) oder allmählich (Differenzengleichungen). Die Zugehörigkeit eines Attributs zu einer (formalen) Komponente wird wie in Abb. 10, links, ausgedrückt.

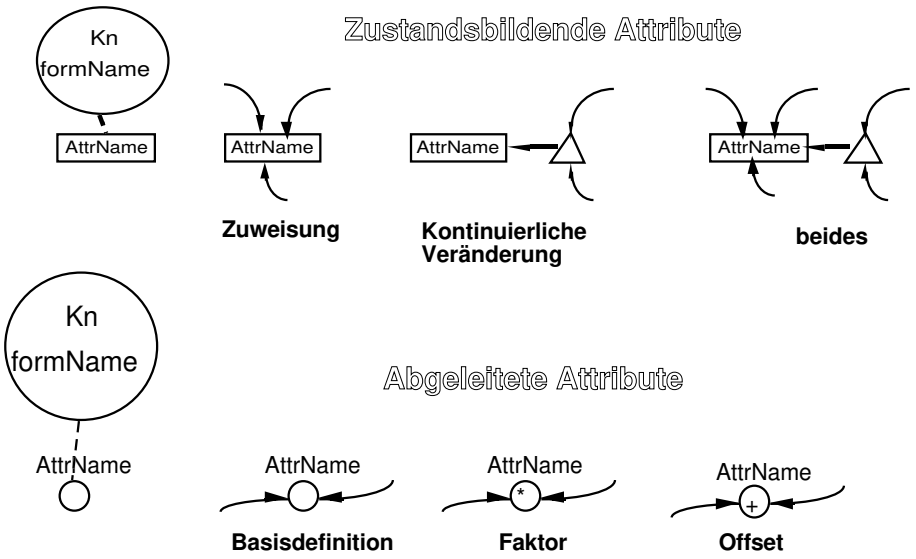


Abb. 10: Attributwertänderungen (Übersicht)

In einer SESAM-Regel können verschiedene Aktionen kombiniert werden:

- Graphstruktur wird verändert,
- Attributwerte verändern sich schlagartig,
- Attributwerte verändern sich über einen Zeitraum hinweg allmählich.

Außerdem gibt es in SESAM das Konzept diskreter Ereignisse, die wie bei herkömmlicher ereignis-orientierter Simulation (z.B. Page, 1991) auftreten können. Beispiele: Mitarbeiter wird krank, Kunde taucht auf.

4. Architektur

Die genannten Konzepte sind im SESAM-Programmsystem implementiert. Die Architektur des Systems wird skizziert. In dieser Archtiektur spiegeln sich die Konzepte.

4.1 Teilsysteme von SESAM

Das SESAM-Programmsystem besteht aus drei Teilsystemen:

- Modellaufbau-Teilsystem mit drei halb-graphischen Editoren zur Entwicklung, Verwaltung und Veränderung von Schema, Situationsmodell und Regeln.
- Konfigurations- und Analyseteilsystem für die Kombination der Modellteile zu einem vollständigen dynamischen Modell. In diesem Teilsystem wird je ein Schema, ein Situationsmodell und eine dazu passende Menge von Regeln ausgesucht. Das Teilsystem kombiniert diese Teile dann automatisch.
- Modellanimations-Teilsystem zur eigentlichen Simulation. In diesem Teilsystem wird das Modell animiert, verschiedene Animationsmodi stehen zur Verfügung: Spielmodus, Analysemodus und Betreuermodus. Je nach ausgewähltem Modus sind mehr oder weniger Informationen und Optionen zugänglich.

4.2 Schichtenstruktur von SESAM

SESAM ist in Smalltalk-80, Version 4.1 implementiert. Diese Smalltalk-Version ist auf PCs, Mac-Intoshs und einer großen Zahl von Workstations verfügbar; SESAM ist ohne jede Anpassung auf allen diesen Plattformen ablauffähig. Derzeit umfaßt SESAM rund 250 Smalltalk-Klassen mit über 4000 Methoden.

Abb. 11: Erwartetes (links) und ausgelöstes Ereignis (rechts)

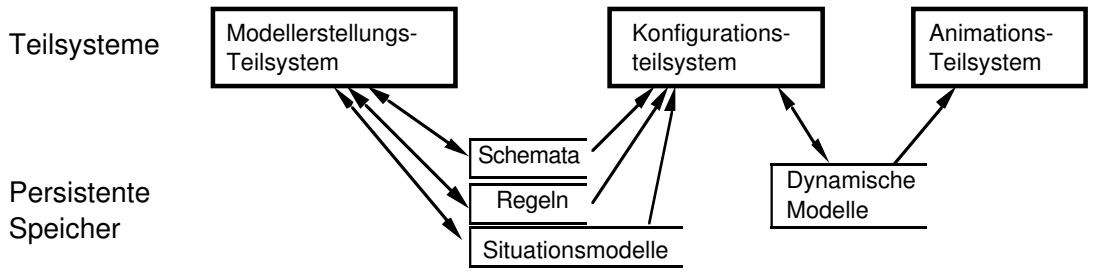


Abb. 12: Die drei SESAM-Teilsysteme, verbunden durch persistente Speicher

Im Modellaufbau-Teilsystem wird der generische graphische Editor vis-A-vis eingesetzt, um die drei halb-graphischen Editoren zu realisieren. vis-A-vis umfaßt rund 75 Smalltalk-Klassen und bietet die Basisfunktionen eines graphischen Editors. Die Spezifika von Schema, Situationsmodell und Regeln werden in den Rahmen von vis-A-vis eingebracht und ergeben so drei halb-graphische Editoren. vis-A-vis ist in Lichter/Schneider (1993) und Lichter/ Schneider (1993a) beschrieben. Jürgen Schwille geht näher auf vis-A-vis ein.

5. Wo steht SESAM heute?

Die oben vorgestellten Konzepte sind im SESAM-Programmsystem umgesetzt. Seit inzwischen eineinhalb Jahren haben wir auch ein Modell, mit dem man ein Projekt von rund sechs Monaten Dauer innerhalb weniger Stunden bzw. Tage im Rahmen einer Lehrveranstaltung simulieren kann. Das Modell war im Wintersemester 1992/93 bereits manuell animiert worden: Damals stand noch kein Simulator zur Verfügung. Die Animation war sehr mühsam und erforderte mehrere Manntage pro Simulationsschritt. Über das Modell und die Lehrveranstaltung, in der wir es eingesetzt haben, berichten wir in Deininger/ Schneider (1994).

Grundlage des Modells ist die Metapher der Stillen Post: In der Software-Entwicklung muß Information vom Kunden in die Spezifikation, weiter in den Entwurf und schließlich in den Code transportiert werden. Dabei kommt es zu Mißverständnissen und Fehlern. Die Aufgabe des Projektleiters ist es, seinen Teil zu einer möglichst unbeschadeten Informationsweitergabe zwischen den Projektbeteiligten und den Dokumenten beizutragen. Er muß die Rahmenbedingungen schaffen und geeignete Anordnungen treffen, so daß die Information möglichst wenig entstellt wird. Die Metapher wird in Schneider (1994) ausführlich beschrieben. Repräsentant einer "Informationseinheit" ist ein Software-Quant.

Ein derartiges Modell ist im Wintersemester 1993/94 im SESAM-Programmsystem aufgebaut worden. Es wurde an vier Studenten erprobt, die interaktiv Projektleiter spielten. In beiden Lehrveranstaltungen erwies sich SESAM als brauchbar.

Erkannte Schwächen sind in einer neuen Version des Modells teilweise beseitigt; die Modellevolution (Abb. 2) hat begonnen und führt zu immer besseren Modellen. Das SESAM-Programmsystem und die zugrundeliegenden Konzepte sind dagegen weitgehend stabil und unverändert geblieben. An der Benutzeroberfläche ist noch einiges zu tun, das Programm ist jedoch inzwischen durchaus einsetzbar.

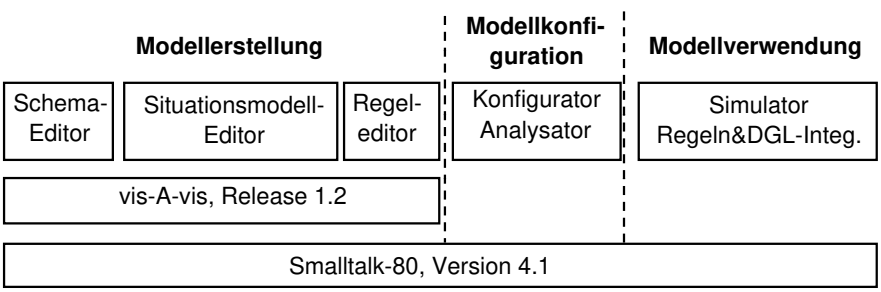


Abb. 13: vis-A-vis als Hilfsschicht im Modellerstellungs-Teilsystem

Literatur

Abdel-Hamid, T. K.; Madnick, S. (1991): **Software Project Dynamics**; Prentice Hall, Englewood Cliffs, NJ

Curtis, B; Kellner, M. I.; Over, J. (1992): Process Modeling; Communications of the ACM, Sept. 1992, vol. 35, no. 9

Deininger, M.; Schneider, K. (1994): Teaching Software Project Management by Simulation - Experiences with a Comprehensive Model; **Proc. of the Conference on Software Engineering Education (CSEE), Austin, Texas, Jan. 1994**

Göttler, H. (1988): Graph-Grammatiken in der Softwaretechnik; Springer, Berlin

Kaiser, G. T. E.; Popovich, S. S.; Ben-Shaul, I. Z. (1993): A Bi-Level Language for Software Process Modeling; **Proc. of the International Conference on Software Engineering (ICSE-15)**, IEEE Comp. Soc. Press

Lichter, H.; Schneider, K. (1993): vis-A-vis: An Object-Oriented Application Framework for Graphical Design Tools; **Proc. of the IFIP Workshop on Interfaces in Industrial Systems for Production and Engineering; Darmstadt, Germany, March 15-17, 1993**; Elsevier

Lichter, H.; Schneider, K. (1993a): vis-A-vis: Ein objektorientiertes Application Framework für graphische Entwurfswerkzeuge; in Mayr, H.C; Wagner, R. (Hrsg.): **Objektorientierte Methoden für Informationssysteme**; Springer, Informatik aktuell

Ludewig, J. (1989): Modelle der Software-Entwicklung: Abbilder oder Vorbilder? **Softwaretechnik Trends, Band 9, Heft 3, Okt. 1989**

Ludewig, J; Bassler, Th.; Deininger, M.; Schneider, K.; Schwille, J. (1992): SESAM - Simulating Software Projects; **Proceedings of the Software Engineering and Knowledge Engineering (SEKE) Conference**, Capri, Italy

Page, B. (1991): **Diskrete Simulation - Eine Einführung mit Modula 2**; Springer, Berlin

Peuschel, B.; Schäfer, W. (1992): concepts and Implementation of a Rule-based Process Engine; **Proc. of the International Conference on Software Engineering (ICSE-14)**, ACM

Schneider, H.-J. (1977): **Graph Grammers**; Lecture Notes in Computer Science 56, pp 314-331

Schneider, K. (1993): SESAM-zwischen Planspiel und Adventure Game; Tagungsband zur 5. Fachtagung Informatik und Schule ´93; Springer, Informatik aktuell "**Informatik als Schlüssel zur Qualifikation**"

Schneider, K. (1993a): Object-Oriented Simulation of the Software Development Process in SESAM; **Proc. of the Object-Oriented Simulation Conference (OOS´93)**, part of the Western Simulation Multiconference, San Diego; SCS Society for Computer Simulation

Schneider, K. (1994): Komm, wir spielen Projektleiter!; Tagungsband zum 3. Workshop SEUH (Hußmann/Paech, Hrsg.: **Software Engineering im Unterricht der Hochschulen'94**); Teubner, Stuttgart

Stachowiak, H. (1972): Allgemeine Modelltheorie; Springer Verlag, Wien, New York

Teil 4

SESAM als Simulator

Jinhua Li

1. Einleitung

SESAM ist ein Simulationssystem, in dem Simulationstechniken verwendet werden, um Software-Entwicklungen zu modellieren. Dieser Artikel behandelt den SESAM-Simulator, ein Teilsystem von SESAM, das die Animation verschiedener Software-Projekt-Modelle (SP-Modelle) unterstützt. Die grundlegende Kenntnis von SESAM und Software-Projekt-Modellbildungen aus der Einführung von Ludewig und den konzeptionellen Grundlagen von Schneider werden für den vorliegenden Aufsatz vorausgesetzt.

Zuerst werden einige in diesem Artikel benutzte Begriffe kurz erklärt, damit man die Diskussionen in den folgenden Abschnitten besser verstehen kann. Genaue Definitionen und ausführliche Auseinandersetzungen dieser und anderer relevanten Konzepte finden sich in anderen SESAM-Dokumenten und in der Literatur.

Unter *Systemsimulation* versteht man eine Methode zur Lösung von Problemen, bei der man die Änderungen eines dynamischen Systemmodells über der Zeit verfolgt (Gordon, 1969). Es wird zwischen *stetigen Simulationen*, die stetige Systeme modellieren, und *diskreten Simulationen*, die diskrete Systeme modellieren, unterschieden. Durchführung von Systemsimulationen besteht aus ein Reihe von grundlegenden Schritten, die sich im wesentlichen in drei Bereiche einordnen lassen: *Modellbildung* (z.B. Problemdefinition, Modellentwurf und -implementation), *Modell- oder Simulationsexperimente* (z.B. Planung und Ausführung von Modellexperimenten) und *Ergebnisanalyse* (z.B. Bewertung von Ergebnissen). Ein rechnerunterstütztes Simulationssystem ist nach Page (1991) ein Softwaresystem, das die Bearbeitung der drei Aufgabenbereiche Modell-bildung, Durchführung von Simulationsexperi-menten und Ergebnisanalyse im Rahmen einer Simulationsstudie unterstützt. In den drei genannten Bereichen sind jeweils die folgenden Funktionen zu erfüllen:

Modellbildung
- Eingabe und Modifikation von Modellen
- Speicherung von Modellen
- Zugriff und Verknüpfung gespeicherter Modelle

Durchführung von Simulationsexperimenten
- Festlegung von Eingabedaten für Experimente
- Start und Ausführung von Simulationsläufen
- Speicherung der Ergebnisse

Ergebnisanalyse
- Zugriff auf gespeicherte Ergebnisse
- Auswahl der zu analysierenden Ergebnisse
- Präsentation der Ergebnisse

Das Teilsystem eines Simulationssystems, das für Simulationsexperimente eingesetzte lauffähige Computerprogramm, wird in der Literatur gelegentlich als *Softwaresimulator* oder kurz *Simulator* bezeichnet (vgl. z.B. Schmidt 1985). Dieser Teil von SESAM wird in diesem Artikel als *SESAM-Simulator* bezeichnet und vorgestellt.

Der folgende Abschnitt behandelt Modelldarstellungen in SESAM-System, die der Bildung von SP-Modellen dienen. Im dritten Abschnitt wird der SESAM-Simulator aus Sicht des Benutzers betrachtet, d.h. wie der Spieler durch SESAM ein Simulationsexperiment (einen Software-Entwick-lungsprozeß) durchführt. Diese wird uns zum Begriff des interaktiven Simulationssystems führen. Der Aufbau und die Mechanismen des SESAM-Simulators, der eigentlich die Animation von SP-Modellen zeitlich vorantreibt, erläutert Abschnitt 4. Der Artikel endet mit einigen Schlußbemerkungen zum Thema Simulation in SESAM.

2. Darstellungen von Software-Projekt-Modellen

Software-Projekt-Modelle im SESAM-System umfassen alle wichtigen in realen Softwareent-wicklungen auftretenden Elemente (z.B. Aufträge, Werkzeuge, Personal und Budget) und Beziehungen (z.B. ein Mitarbeiter erzeugt oder liest ein Dokument). Sie werden in SESAM in zwei Kategorien eingeordnet: Entitäten und Beziehungen. Entitäten lassen sich wieder durch Attribute beschreiben. Ein Mitarbeiter in SESAM wird z.B. als eine Entität modelliert, die durch Attribute wie Qualifikation, Motivation und Fähigkeit zur Programmierung charakterisiert wird. Das im Simulationsverlauf entstandene Produkt wird auch durch eine Reihe von Attributen (wie den Umfang und Qualität einer Software) enthaltender Entitäten beschrieben. Eine Beziehung verbindet mindestens zwei Entitäten, z.B. ein Mitarbeiter diskutiert mit dem Kunden über die Softwareanforderungen.

Schwieriger als bei anderen Simulationen sind bei SESAM einerseits die konzeptionelle Modell-bildung von SP, die die Realität sinnvoll und adäquat nachbildet, und andererseits die konkrete Darstellung ihrer

Attributwerte, die sowohl genaue Werte (Zeit und Budget) oder unscharfe Werte (Motivation und Leistung eines Mitarbeiters) sein können, als auch stetig verändern (Qualität eines Softwareprodukts und Entwurfsfähigkeit eines Mitarbeiters) oder sprunghaft verändern (Anzahlsänderungen von Mitarbeiter und Budget) können.

Um SP-Modelle genau nachzubilden, wird in SESAM ein SP-Modell in drei Schichte beschrieben: Das attributierte Entity-Relationship-Schema führt die Entitäten (mit ihren Attributen) und ihre möglichen Beziehungen ein. Die Anfangssituation, mit der der Spieler ein Simulationsexperiment durchführt, ist eine spezielle Ausprägung dieses Schemas. Die Änderungen der Situation (des Szenarios) werden durch Regeln, die sog. Effektmodelle, beschrieben. Allmähliche Änderungen der Attributwerte sind nach dem Konzept von „System Dynamics“ definiert; sprunghafte Änderungen aufgrund von Ereignissen (Einführung oder Entfernung von Entitäten und Beziehungen) sind durch eine attributierte Graph-Grammatik vorgegeben. Zur Erzeugung und Handhabung der drei Schichten stehen in SESAM entsprechend jeweils drei Graphen-Editoren zur Unterstützung, des universellen Werkzeugs „vis-A-vis“ (Lichter, Schneider, 1993) zur Verfügung.

3. SESAM-Simulator I: benutzer-orientierte Simulation

In diesem Abschnitt wird der SESAM-Simulator aus Sicht des SESAM-Spielers erläutert. Zu Beginn eines Simulationsexperiments steht dem Spieler eine Ausgangslage (das Szenario) zur Verfügung, die ein simuliertes Softwareprojekt repräsentiert und alle wesentlichen Informationen und Gegenstände für ein Projekt enthält, z.B. Budget, Kunde, Betriebsmittel (z.B. Räume und Werkzeuge) und eine Gruppe von Mitarbeitern. Sie wird in SESAM graphisch dargestellt und bildet den Anfangszustand einer Simulation.

Ziel des Spielers ist es, durch seine Aktionen mit den Unterstützungen des SESAM-Simulators das modellierte Projekt erfolgreich zu Ende zu bringen. Er übt seinen Einfluß auf die Simulationsläufe über eine Benutzeroberfläche in Form von Aktionen, also Spieleraktionen, aus. Abbildung 1 veranschaulicht diese Beziehung zwischen dem SESAM-Simulator und dem Spieler.

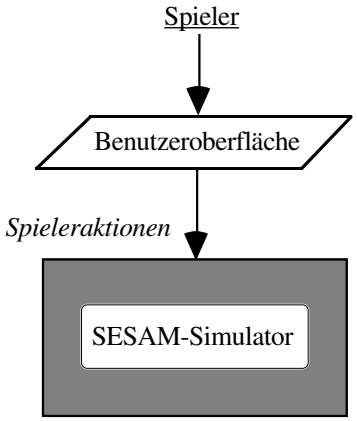


Abb. 1: Simulationsläufe durch die Spieleraktion

Zur Verfügung stehen dem Spieler eine Reihe von vordefinierten möglichen Spieleraktionen, mit denen er ein simuliertes Projekt leiten kann. Der Spieler muß allerdings die Aktionen auswählen, ihre Durchführungen ordnen und Entscheidungen treffen, wenn etwas passiert ist, d.h. ein Ereignis zufällig entsteht. Die Spieleraktionen entsprechen im Prinzip allem, was ein Projektleiter in der Praxis tun kann. Beispielsweise kann er

- Mitarbeiter einstellen oder entlassen,
- Arbeitsgruppen bilden und Aufgaben erteilen,
- entscheiden, ob nötige Werkzeuge erzeugt oder gekauft werden,
- beim Vorgesetzten sein Budget verteidigen,
- zu einer Besprechung zusammenkommen.

Außerdem ist dem Spieler möglich, die Simulationsläufe zu beenden, zu unterbrechen oder wiederaufzunehmen sowie sich die aktuellen Informationen seines Projekts zu geben lassen.

Der SESAM-Simulator verarbeitet solche Spieleraktionen, wodurch sich der Systemzustand ständig verändert. Der SESAM-Simulator kann z.B. (vgl. Tabelle 1)

- Entitäten in das System einfügen oder entfernen,
- Beziehungen zwischen existierenden Entitäten einrichten
- Attributwerte in einem Systemzustand verändern oder
- Ereignisse erzeugen (z.B. Mitarbeiter E ist kranken und muß eine Woche zu Hause bleiben) oder behandelt (z.B. muß der SESAM-Simulator auf dem Ereignis, daß Mitarbeiter E nicht zur Arbeit kommen kann, aber mit dem Kunden einen Termin ausgemacht hat, dadurch reagieren, daß er entweder eine Aktion auslöst oder den Spieler eine Entscheidung treffen läßt)

Spieleraktion	Auswirkungen im SESAM-Simulator
Erzeugen einer Modulspezifikation	Einfügen der Entität in dem Systemzustand
Entlassung eines Mitarbeiters	Löschung der Entität in dem Systemzustand
Mitarbeiter A beschäftigt sich mit Teilsystem B nach Spezifikation C	Übernahme der entsprechenden Entitäten in den Systemzustand und Änderungen der entsprechenden Attributwerte über der Zeit
Mitarbeiter D diskutiert mit Kunden K über die Systemanforderungen	ständige Änderung der Attributwerte, z.B. des Verständnisses von Mitarbeiter D für das Projekt und Funktion-Points der simulierten Software-Spezifikation

Tab. 1: Einige Spieleraktionen und entsprechende Auswirkungen im SESAM-Simulator

Außerdem verwaltet der SESAM-Simulator die simulierte Uhr, in dem er die Simulationszeit nach jeder Aktion des Spielers weitersetzt.

Außerhalb des SESAM-Systems können nur die Auswirkungen des Spielers auf das Systemverhalten beobachtet werden. Der Spieler bestimmt den Verlauf eines Simulationsexperiments und spielt damit die zentrale Rolle bei SESAM-Simulationen. SESAM wird deshalb als interaktives Simulations-system oder benutzer-orientiertes Simulationssystem bezeichnet.

Die Abläufe im SESAM-Simulator während einer Simulation lassen sich allerdings nicht direkt außerhalb erkennen. Der SESAM-Simulator reagiert auf jede Spieleraktion, indem er den Systemzustand ständig verändert und damit die Simulation um eine Simulationszeiteinheit (Δt) fortführt. Wie der SESAM-Simulator arbeitet und was innerhalb des SESAM-Simulators passiert, wird in dem nächsten Abschnitt ausführlich beschrieben.

4. SESAM-Simulator II: Aufbau und Mechanismen

Die gegenwärtige Version des SESAM-Simulators besteht aus zwei Modulen: einem Ereignis-Simulator (wird kurz als Simulator bezeichnet) zur Animation von zufälligen oder anschließend vom Spieler ausgelösten Ereignissen, und ein Regelanwender (RA) , der u.a. die zeitkonsumierenden Vorgänge von SP-Modellen simuliert. Um den Ablauf beider Module zu synchronisieren, wird in SESAM der RA dem Ereignis-Simulator untergeordnet (Kiehne, 1993). Dies führt dazu, daß der Ereignis-Simulator die Simulation zeitlich vorantreibt. SESAM ist ein gemischtes Simulationssystem in dem Sinne, daß in SESAM einerseits von einem Zeitpunkt zum nächsten Zeitpunkt aufgrund von Ereignissen um die Simulationsschrittweite (Δt) fortgeschritten wird, andererseits sich das Systemverhalten in einer gewissen Situation innerhalb jeder simulierten Zeitspanne (Δt) quasi-kontinuierlich verändert. Diesen Zeitverlauf und die Zustandsveränderungen zwischen zwei simulierten Zeitpunkten kann man jedoch außerhalb des SESAM-Simulators nicht beobachten.

Abbildung 2 zeigt die Komponenten des SESAM-Simulators, und ihre Beziehung mit anderen Komponenten, die zum Systemverhalten zusammen beitragen.

Systemzustand und Spieleraktionen, die eigentlich nicht zum SESAM-Simulator gehören und dennoch enge Zusammenhänge mit dem Systemverhalten in einem Simulationslauf haben, werden ebenso in diesem Abschnitt erläutert. Der Systemzustand wird in SESAM als ein Spielzustand und ein erweitertes System-Dynamics-Modell (SD-Modell) repräsentiert. Ereignisse signalisieren, daß etwas passiert ist. In SESAM repräsentieren sie Zustandsänderungen und beinhalten keine Information über den aktuellen Zustand selbst, sondern machen Aussagen über seine Entwicklung. Es ist dennoch praktisch, Ereignisse als Teil des Zustands anzusehen, denn so kann man anhand der eingetretenen Ereignisse erkennen, was in der Vergangenheit mit dem Zustand passiert ist. Ereignisse sind in SESAM entweder exogene Ereignisse, die vom Spieler ausgelöst werden, oder endogene Ereignisse, die durch den RA zufällig erzeugt werden. Spieleraktionen, mit denen der Spieler seinen Einfluß auf die Simulation ausübt, werden im SESAM-Simulator als exogene Ereignisse angesehen.

Der Ereignis-Simulator stellt die Drehscheibe der Simulation dar. Er kommuniziert mit allen in einem Simulationslauf mitwirkenden Komponenten, nämlich den RA, den Spieleraktionen, den Systemzustand usw. Die Hauptaufgabe vom Ereignis-Simulator besteht darin, daß er

- alle Ereignisse speichert, verwaltet und manipuliert (z.B. er nimmt die Spieleraktionen in Empfang und vermerkt alle Ereignisse als eingetreten, deren Eintrittszeitpunkt erreicht oder Überschritten ist),
- einen Teil von Attributwerten im SD-Modell berechnet und damit auch den Systemzustand konsistent macht,
- die Simulationsergebnisse sammelt und verwaltet,
- die Routinearbeiten wie Setzen von Simulationsparametern (z.B. Anfangs- und Endzeit), Start der Simulation und Vormerkung von Unterbrechungs-zeitpunkten zur Verführung stellt und
- die Simulationszeit, die während einer Simulation vergehende Zeit, sprunghaft fortschaltet.

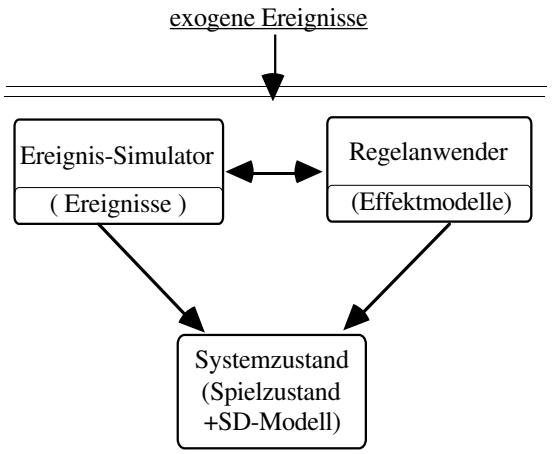


Abb. 2: Logische Struktur des SESAM-Simulators

Der Regelanwender, ähnlich einem Regelsystem, stellt die Art und Weise des Systemverhaltens dar, d.h. wie sich der Systemzustand ändert. Effektmodelle, die der Wissenbasis eines Regelsystems entsprechen, sind die bearbeiteten Hypothesen im Software Engineering. Der RA beschreibt, wie die Regeln in einer Situation angewendet werden können, und bestimmt, ob das Systemverhalten schon zu einem stabilen Zustand erreicht ist, so daß keine Regeln mehr verwendbar sind. Unter einem stabilen Zustand versteht man einen Zustand, in dem alle Strukturen stimmig (widersprechen keiner Regeln) sind und alle Werte auf denselben Zeitpunkt und aufeinander bezogen sind. Aber der RA unterscheidet sich von gewöhnlichen Regelsystem so, daß in SESAM nicht nach einem Lösungszustand gesucht, sondern das Verhalten von Softwareprojekt nachgebildet wird. Die Aufgabe der Problemlösung kommt dabei dem Spieler, nicht dem SESAM-System zu.

Der RA führt die durch Ereignisse ausgelösten Aktionen aus und verändert quasi-kontinuierlich den Systemzustand zwischen zwei Simulationszeitpunkte dadurch, daß er nach Ereignissen und Regeln

- Attributwerte ändert,
- Attribute, Entitäten und Beziehungen im Spielzustand erzeugt oder löscht und
- Ereignisse zufällig erzeugt.

Simulator und RA stellen gemeinsam die Antriebsfeder des SESAM-Systems dar. Sie arbeiten Hand in Hand und verändern den Systemzustand gemäß den verwendeten Effektmodelle und aufgetauchten Ereignissen. Änderungen der Systemzustände mit der Zeit, d.h. die Animation eines Software-Prozesses, sind möglich durch

- die entweder vom Spieler ausgelösten oder zufällig eingetreten Ereignissen oder
- das Überschreiten von Schwellwerten definiert sein können.

Das Zusammenwirken von Spieler und SESAM-Simulator

Um das Zusammenwirken von Spieler, Regelanwender und Ereignis-Simulator klar zu machen, nehmen wir ein einfaches Beispielszenario in Anlehnung an Krause (1993), das in Abbildung 3 anschaulich dargestellt ist. Fangen wir damit an, daß in einer bestimmten Situation zu einem Zustand 1 der Spieler einen Mitarbeiter des Projekts zum Kunden geschickt hat. Diese Spieleraktion fügt in den Zustand ein Ereignis ein, das sofort bearbeitet wird. Die Kontrolle wird nach Ausführung dieser Spieleraktion an den Regelanwender übergeben (die Übergabe der Kontrolle wird durch vertikale Linien mit Pfeilen angedeutet), der nun prüft, ob er aufgrund der Effektmodellen Aktionen ausführen soll. Wir nehmen für den Zustand 1 an, daß dies der Fall ist. Eine Regel reagiert auf die Spieleraktion und entfernt den Mitarbeiter aus dem Raum, in dem er sich gerade befindet und richtet eine Beziehung (sprichtMit) zwischen ihm und dem Kunden ein. Zugleich beeinflußt sie den Zustand des Mitarbeiters durch die Besprechung mit dem Kunden. Diese Aktionen führen zum Zustand 2, in dem die Attribute des Systems, z.B. das Verständis dieses Mitarbeiters für das Projekt, beeinflußt werden und deshalb der Zustand inkonsistent wird. Berechnung der Attributwerte ist nun die Aufgabe des Simulators, der die Kontrolle übernimmt, um den Zustand logisch konsistent zu machen. Jetzt ist der Zustand 3 erreicht.

Aufgrund der veränderten Attribute könnte nun eventuell eine Regel anwendbar sein. Die Kontrolle geht darum wieder an den RA zurück. Dieser tut wieder das gleiche wie in Zustand 1: Er wendet die Effektmodelle auf den veränderten Zustand an. Werden dabei Aktionen ausgeführt, so wird der nächste Zustand, Zustand 4, wieder dem Simulator überlassen, der die Attribute berechnet. So kann die

Kontrolle zwischen dem Simulator und dem RA mehrfach hin und her gehen, bis der RA keine Regeln mehr anwenden kann und ein stabiler Zustand gelangt. In Abbildung 3-2 stellt Zustand 5 einen solchen stabilen Zustand dar, mit ihm nun Zustand 6 identisch ist.

Bis zu diesem Punkt ist die Simulationszeit noch nicht fortgeschritten. Dieser Abschnitt der Simulation wird *Phase 1 der Simulation* genannt. Nun kann der Simulator also einen Simulationsschritt ausführen und die Simulationszeit um ein Δt weiterschalten. Dies heißt Phase 2 der Simulation oder *Simulationsschritt*. Damit ist Zustand 7 erreicht und der Spieler kann wiederum eine Aktion ausführen. Denkbar ist, daß während der Phase 1 der Simulation eine Situation eintritt, die unbedingt einen Spielereingriff braucht. In diesem Fall muß die Kontrolle an den Spieler gegeben werden. Zustände 8 und 10 stellen einen solchen Simulationsverlauf dar: Zustand 9 verlangt nach einer Spielerentscheidung; bevor die Kontrolle an den Spieler geht, macht der Simulator den Zustand noch konsistent.

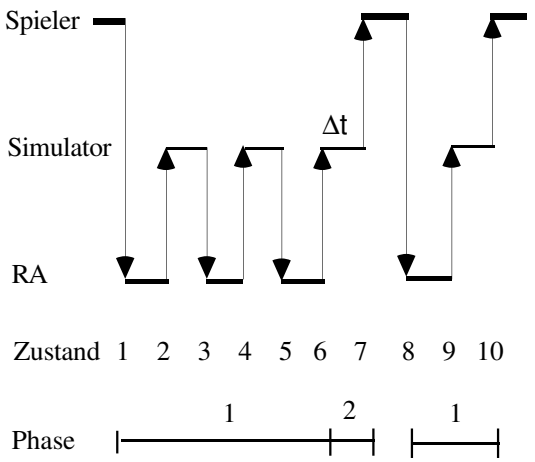


Abb. 3: Zusammenspiel von Spieler, RA und Simulator

5. Schlußbemerkungen

Eine entschiedene Rolle spielt in Simulationssystemen und in SESAM Zeit, von der u.A. die Qualität eines Simulators abhängt. In SESAM-Simulator geht es um zwei Aspekte von Simulationszeit. Erstens muß die Entscheidung über das kleinste Zeiteinheit, auch Simulationsschrittweite genannt, getroffen werden, um die praktischen Situationen von Software-Entwicklungen getreu simulieren zu können. Zweitens weil es in SESAM zwei Modulen, der Ereignis-Simulator und der Regelanwender, gibt, ist es erforderlich und auch möglich, die Schrittweiten in beiden Teilen anzupassen. Außerdem ist im Sinne vom Abenteuerspiel, daß der Spieler den Spielverlauf möglicherweise schnell durchführt, eine Erweiterung von SESAM zweckmäßig. Dieser schwierige Aspekt eines Abenteuerspiels ist gekennzeichnet durch die Realzeit, die tatsächlich in der realen Welt vergeht, und die CPU-Zeit, die einer Spieler bei der Simulation auf einem Rechner benützt.

Der Regelanwender ist im wesentlichen ein Regelsystem, das aus drei Komponenten besteht: Effektmodell (Wissensbasis), Konfliktlösungsstrategie und Regelanwendungsverfahren (Kontrolleinheit) sowie Systemzustand (Datenbasis). Im gegenwärtigen SESAM ist jedoch noch keine Konfliktlösungsstrategien fertiggestellt. Bei Durchführung von Simulationsexperimenten verwendeten statistischen Verfahren (z.B. Verteilungsfunktionen wie die Normalverteilung oder Erlangverteilung) sind nun noch nicht zufrieden in Simulationsmodellen eingebracht.

Gekennzeichnet ist SESAM in bezug auf Simulationstechniken durch:

- interaktive Simulationsablaufsteuerung
- Anwendung von regelbasierten Techniken in Systemsimulationen
- Erweiterung und Anwendung der formalen Darstellung programmierter attributierten Graph-Grammatik mit objektorientierten Ansätze
- Erweiterung und Anwendung der Systemtheorie System Dynamics bei Modellierung und Simulierung von Softwareprozessen
- Erweiterung und Anwendung von zeitgesteuerten und ereignisorientierten Simulationsmethoden in einem Simulationssystem
- Repräsentation und Handhabung von unscharfen Eigenschaften und Wissen in Software Engineering und in Simulationssystemen
- interaktive graphische Modellbildung in Systemsimulationen.

Literatur

Gordon, G. (1969): **System Simulation**. Prentice-Hall, Inc. 1969.

Kiehne, K. (1993): **Entwurf und Implementierung eines Simulator-Moduls für SESAM**. Diplomarbeit 870, Fakultät Informatik, Universität Stuttgart, 1993.

Krause, M. (1993): **Entwicklung eines regelbasierten Baukasten zur Verhaltensmodellierung in SESAM**. Diplomarbeit 994, Fakultät Informatik, Universität Stuttgart, 1993.

Lichter, H.; Schneider, K. (1993): vis-A-vis: Ein objekorientiertes Application Framework für graphische Entwurfswerkeuge. in Mayr, H.C.; Wagner, R. (Hrsg.) **Objektorientierte Methoden für Informationssysteme (EMISA-Tagung)**; Springer, Informatik aktuell, pp. 187-207, 1993.

Page, B. (1991): **Diskrete Simulation - Eine Einführung mit Modula-2**. Springer 1991.

Schmidt, B. (1985): Systemanalyse und Modell-aufbau. **Grundlagen der Simulationstechnik.** Fachberichte Simulation 1. Berlin: Springer 1985.

Teil 5

SESAM und die Realität

Anke Drappa

Zusammenfassung

SESAM ist ein objekt-orientiertes Werkzeug, mit dem Modelle für Software-Projekte in einer graphischen Notation beschrieben und anschließend animiert werden können.

In diesem Beitrag wird untersucht, wie sinnvolle, d.h. insbesondere realitätsnahe Modelle von Software-Projekten entwickelt werden können und wodurch sie sich auszeichnen. Dafür werden die Ergebnisse verschiedener Arbeiten, die in den letzten Jahren in der Abteilung Software Engineering durchgeführt worden sind, vorgestellt und die dabei gewählten Ansätze diskutiert.

1. Einführung

SESAM ist ein Simulator für Software-Projekte, der im Rahmen eines Forschungsprojekts der Abteilung Software Engineering an der Universität Stuttgart entwickelt worden ist.

In SESAM können Modelle von Software-Entwicklungsprojekten in einer überwiegend graphischen Notation beschrieben und anschließend "durchgespielt" werden. Der Spieler übernimmt dabei die Rolle des Projektleiters und veranlaßt alle für die Projektdurchführung notwendigen (simulierten) Tätigkeiten, wie z.B. Einstellen von Mitarbeitern, Zuteilen von Aufgaben oder Beschaffen von Werkzeugen.

In der Art eines Adventure Games erhält der Spieler Informationen und Reaktionen vom System, die seine weiteren Aktivitäten im simulierten Projekt bestimmen. Am Ende des Spiels kann der Verlauf des Software-Projekts sichtbar gemacht und analysiert werden. Der Spieler erfährt, welche Aktionen sich günstig und welche sich eher negativ auf das simulierte Projekt ausgewirkt haben.

Hier steht weniger das *Werkzeug* SESAM als vielmehr die Problematik der Modellierung von Software-Projekten im Vordergrund. Ziel der Modellierung ist, dem Praktiker ein realitätsnahes Abbild seiner Wirklichkeit anzubieten. Es wird untersucht, welche Schwierigkeiten bei der Modellierung bestehen und wie die Abbildung der Realität in ein Modell durch empirische Daten gestützt werden kann.

In der Abteilung Software Engineering sind mehrere Arbeiten durchgeführt worden mit dem Ziel, Gesetzmäßigkeiten bei der Abwicklung von Software-Projekten zu erkennen. In diesen Arbeiten wurden sowohl die vorhandenen Literaturquellen ausgewertet als auch reale Projektdaten erhoben und analysiert. In diesem Beitrag werden Inhalt und Ergebnisse der Arbeiten vorgestellt. Darüber hinaus wird diskutiert, inwieweit die gewählten Ansätze die Modellbildung unterstützen können.

Während die Hypothesensammlung (Utz, 1992) und die Erhebung von Metriken in Software-Projekten (Drappa, 1993) unabhängig von SESAM durchgeführt wurden, ist in Feest (1993) das bisher in SESAM vorhandene Modell als Basis für die Untersuchungen verwendet worden. Zunächst sollen der grundsätzliche Aufbau der Modelle sowie die Grundidee des in SESAM verfügbaren Modells kurz erläutert werden.

Komponenten von SESAM-Modellen

Modelle in SESAM bestehen grundsätzlich aus drei Komponenten, dem Schema, der Startsituation und den Regeln.

Im Schema werden die benötigten Entitäts- und Relationstypen definiert und mit den zu ihrer Charakterisierung erforderlichen Attributen versehen. Das Schema liefert im Prinzip eine abstrakte Beschreibung für alle im Modell berücksichtigten Objekte und Beziehungen der realen Welt.

Im Verlauf des Spiels werden von diesen Entitäts- und Relationstypen Ausprägungen (also Entitäten und Relationen) erzeugt und deren Attribute mit konkreten Werten belegt. In der Startsituation wird der Anfangszustand der Simulation definiert, d.h. es werden alle im Spiel bereits verfügbaren Entitäten und Relationen mit den entsprechenden Attributwerten erzeugt.

Die letzte Modellkomponente, die Menge von Regeln, dient der Änderung des Spielzustands während der Simulation. Jede Regel übt gewisse Effekte auf den Spielzustand aus, wie z.B. das Erzeugen oder Löschen von Entitäten bzw. Relationen oder das Ändern von Attributwerten.

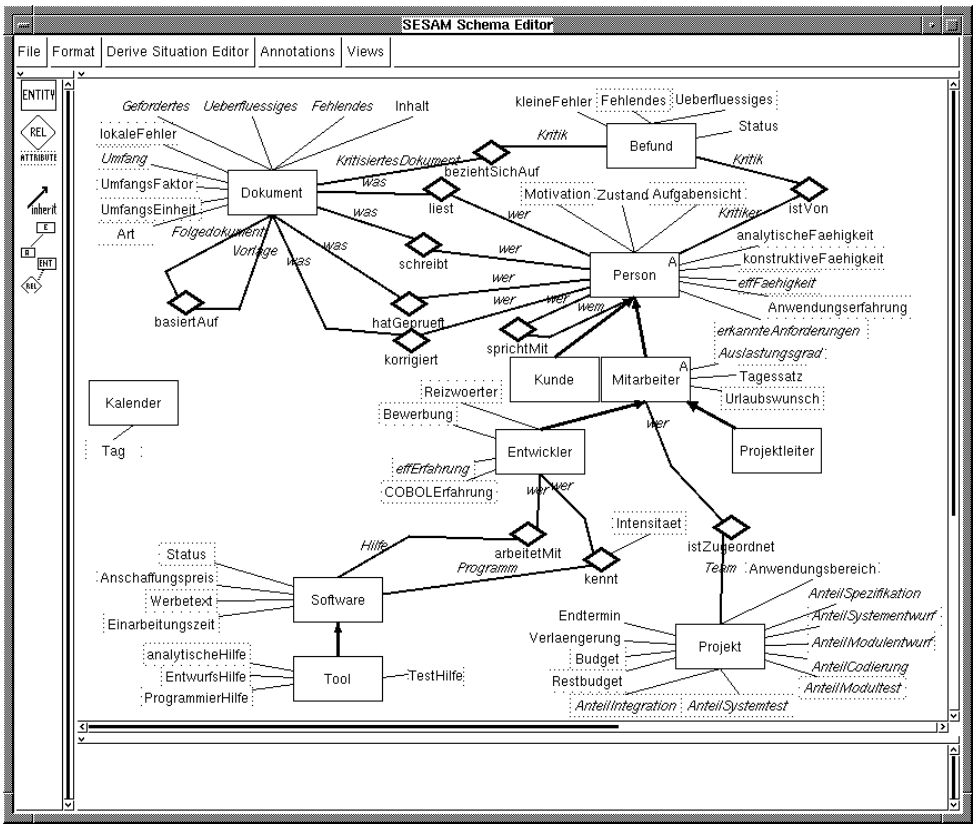


Abb. 1: Schema im derzeitigen SESAM-Modell

Bisheriges Modell

Das derzeit in SESAM vorhandene Modell ist konzeptionell noch sehr einfach. Auf der Schemaebene werden im wesentlichen Dokumente, Personen (mit verschiedenen Subentitäten), Werkzeuge sowie das Projekt selbst berücksichtigt. Die wichtigsten Beziehungen bestehen zwischen den Entitätstypen Dokument und Person, d.h. Personen lesen, schreiben oder prüfen Dokumente (vgl. Abbildung 1).

Die Mitarbeiter werden durch ihre (konstruktiven und analytischen) Fähigkeiten sowie durch ihre Erfahrung charakterisiert. Die Dokumente werden hauptsächlich durch die Attribute Umfang und Inhalt modelliert, wobei die Beschreibung des Inhalts durch die Attribute Gefordertes, Überflüssiges und Fehlendes detailliert wird.

Ein Problem der Modellierung besteht darin, den Attributen "aussagekräftige" Werte zuzuordnen. Bei den Attributen der Entität Person und den Attributen ihrer Subentitäten wurden im wesentlichen Multiplikatoren verwendet, die mit den aus COCOMO bekannten Anpassungsfaktoren (z.B. ACAP, PCAP) vergleichbar sind. Das Attribut Fähigkeit erhält bei einer Person mit durchschnittlichen Fähigkeiten beispielsweise den Wert 1, während entsprechend geringere oder höhere Fähigkeiten durch Werte kleiner bzw. größer 1 modelliert werden.

Die wesentliche Idee des Modells liegt jedoch in der Art der Modellierung des Inhalts und der Qualität der Dokumente. Dafür wurden die sogenannten Software-Quanten entwickelt, die für kleinste, nicht weiter teilbare Aufgabeneinheiten stehen. Software-Quanten sind voneinander unterscheidbar, werden aber inhaltlich nicht näher spezifiziert.

Der in dem Projekt zu entwickelnden Software wird eine bestimmte Menge an erforderlichen Software-Quanten zugeordnet. Die Idee ist nun, Inhalt und Qualität der Dokumente durch die Menge der ihnen zugeordneten Software-Quanten zu beschreiben. Angenommen, die zu erstellende Anwendung erfordert 234 Software-Quanten. Dann müßten in der Spezifikation idealerweise ebenfalls 234 Software-Quanten enthalten sein, wenn nach dem Wünschen des Kunden spezifiziert wurde. Tatsächlich werden aber oft nicht alle geforderten Software-Quanten spezifiziert, sondern einige fehlen oder überflüssige eingeführt werden. Je mehr dieser Software-Quanten nun in die Spezifikation "hinübergerettet" werden, desto höher ist die Qualität des entstandenen Dokuments.

Im Modell werden also Regeln bereitgestellt, die das Fließen dieser Software-Quanten beschreiben. Abhängig von dem in das Dokument investierten Bearbeitungsaufwand und von den Fähigkeiten der Bearbeiter wächst die Menge der Software-Quanten, die den Inhalt des Dokuments modelliert. Je länger an dem Dokument gearbeitet wird, desto weniger Quanten werden fehlen und desto mehr überflüssige werden eingeführt. Dabei werden die Ergebnisse umso besser sein, je höher die zuständigen Mitarbeiter qualifiziert sind. Nach dem gleichen Prinzip wird auch das Fließen der Software-Quanten von der Spezifikation in den Entwurf und vom Entwurf in den Code modelliert.

Die Entität Projekt beschreibt wesentliche Aspekte des durchzuführenden Software-Projekts, wie z.B. den geplanten Endtermin, das verfügbare Budget oder den Anteil einzelner Phasen am Gesamtaufwand.

In diesem Stadium des Forschungsprojekts ist das Modell weit davon entfernt, vollständig und wirklichkeitsgetreu zu sein, obwohl wesentliche Aspekte durchaus realistisch modelliert worden sind (vgl. Kapitel 3). Es soll hauptsächlich als Grundlage für die hier präsentierten Überlegungen betrachtet werden. Es geht also nicht nur darum, *dieses* Modell zu kritisieren, sondern es muß festgestellt werden, wodurch sich gültige Modelle auszeichnen und auf welcher Basis diese erstellt werden können. Diese Aspekte sollten durch die Diplomarbeiten untersucht werden, die im folgenden vorgestellt werden.

2. Sammlung von Hypothesen

Die erste der folgenden drei Arbeiten wurde im Jahr 1991 durchgeführt. Zu dieser Zeit befand sich das Forschungsprojekt SESAM noch im Anfangsstadium, diese Arbeit diente auch der Klärung der Anforderungen an ein Programmsystem zur Simulation.

Ziele der Arbeit

In der Diplomarbeit sollten Hypothesen über Software-Projekte gesammelt und für SESAM formuliert, klassifiziert und quantifiziert werden. Die Grundlage für die Sammlung bildete die vorhandene Software Engineering-Literatur.

Ziel der Diplomarbeit war die Gewinnung von Aussagen über Software-Projekte, welche die darin gültigen Zusammenhänge beschreiben. Durch die Definition einer einheitlichen Repräsentationsform und die Entwicklung eines Klassifikationsschemas sollten die Hypothesen gruppiert und ähnliche Hypothesen identifiziert werden.

Diese Hypothesen-Sammlung sollte eine erste Grundlage für ein Modell des Software-Entwicklungsprozesses darstellen. Die vorhandenen Hypothesen können auf dieser Basis schrittweise verifiziert, falsifiziert oder angepaßt werden.

Ergebnisse der Arbeit

In der Arbeit ist zunächst ein Schema entwickelt worden, das den Aufbau von Hypothesen beschreibt. Danach verwendet eine Hypothese Attribute von Eingabeobjekten und verändert (hier) ein Attribut eines Zielobjekts. Die sogenannte Auslösebedingung enthält Informationen darüber, wann eine Hypothese gültig ist (z.B. wenn ein bestimmtes Ereignis eingetreten ist).

Darüber hinaus ist ein Schema zur Klassifikation der Hypothesen nach den Zielobjekten, über die sie Aussgen machen, erarbeitet worden; z.B. gibt es Klassen wie Mitarbeiter, Kunde, Sourcecode oder Testdokument.

Die in der Literatur gefundenen Hypothesen sind zusammengestellt und in das Klassifikationsschema eingeordnet worden. Dabei ist eine umfangreiche Sammlung mit insgesamt 273 Hypothesen entstanden. Es war zu beobachten, daß die Klassenbelegung sehr inhomogen war; sechs von sechzehn Klassen enthielten keine Hypothesen, während andere Klassen, z.B. Mitarbeiter oder Sourcecode, mit 66 bzw. 60 Hypothesen sehr stark belegt waren.

Die folgenden Beispielhypothesen sollen die Ergebnisse der Arbeit veranschaulichen (Utz, 1992):

- Je größer eine Organisation ist, desto niedriger ist die Produktivität. (1)
- Wenn wiederverwendbarer Code eingesetzt wird, dann steigt die Qualität von Modulen. (2)
- Der Aufwand für ein Software-Projekt ist von der Programmgröße abhängig und beträgt im Modus semidetached $E = 3.2 (KDSI)^{1.05}$ MM. (3)
- Fehlerbeseitigungsaktivitäten erfordern im Durchschnitt 30 % des gesamten Entwicklungsaufwands. (4)
- Die Produktivität für "leichte" Funktionen beträgt 20 LOC/Tag, für "mittlere" Funktionen 10 LOC/Tag und für "schwere" Funktionen 5 LOC/Tag (LOC = Lines of Code). (5)
- Bei Kontrollprogrammen (z.B. Betriebssystemen) beträgt die Produktivität 600 LOC pro Mitarbeiter und Jahr, bei Übersetzern beträgt sie ca. 2200 LOC pro Mitarbeiter und Jahr. (6)

Viele in der Arbeit gesammelten Hypothesen beschreiben einen rein qualitativen Zusammenhang zwischen verschiedenen Attributen. Die beiden ersten der oben genannten Hypothesen geben z.B. einen Zusammenhang zwischen der Organisationsgröße und der Produktivität bzw. zwischen wiederverwendbarem Code und der Modulqualität an. Andere Hypothesen enthalten zwar quantitative Angaben, sind aber dennoch sehr allgemein formuliert. Es ist

beispielsweise nicht klar, welche der in einem Projekt durchgeführten Tätigkeiten zu den Fehlerbeseitigungsaktivitäten zu rechnen sind (vgl. Hypothese 4). In weiteren Hypothesen sind exakte Angaben enthalten, ihre Aussagen weichen aber von den Untersuchungsergebnissen anderer Autoren teilweise ab (vgl. Hypothesen 5 und 6).

Diskussion des Ansatzes

Obwohl die Hypothesensammlung interessante Einblicke in die Praxis der Software-Entwicklung gewährt, ist sie als Grundlage für die Modellierung von Software-Projekten in SESAM nur mit einigen Einschränkungen verwendbar. In der Literatur waren überwiegend qualitative und nicht – wie erhofft – quantitative Aussagen zu finden. Qualitative Aussagen sind aber nicht oder nur schlecht für die Simulation zu verwenden. Zum einen müssen die Zusammenhänge zwischen den Attributen exakt quantifiziert sein, damit eine entsprechende Regel formuliert und in der Simulation eingesetzt werden kann. Zum anderen muß festgelegt werden, wie die Attribute gemessen werden können, durch welche Metrik ihnen also ein Wert zugeordnet wird.

Betrachtet man beispielsweise die im letzten Abschnitt genannte Hypothese "Je größer eine Organisation ist, desto niedriger ist die Produktivität", so wird deutlich, daß zwar irgendein, aber sicher kein quantitativer Zusammenhang zwischen den Attributen Organisationsgröße und Produktivität postuliert wird. Während bei dieser Hypothese wenigstens noch vorstellbar ist, wie die Attribute bewertet werden könnten (Organisationsgröße durch die Anzahl der Mitarbeiter und Produktivität durch LOC pro Tag), ist beispielsweise für das Attribut Modulqualität bisher keine einheitliche, validierte Meßvorschrift bekannt.

Ein weiteres Problem bei der Sammlung von Hypothesen aus der Literatur besteht darin, daß der Kontext, in dem die jeweiligen Untersuchungen stattgefunden haben, verloren geht. Bestimmte Zusammenhänge gelten z.B. nur unter besonderen Randbedingungen oder speziellen Voraussetzungen. Wegen der Reduktion der durchgeführten Untersuchungen auf die einzelnen Aussagen werden solche Einschränkungen u.U. nicht mehr berücksichtigt. Dies ist auch eine mögliche Erklärung für die z.T. stark voneinander abweichenden Aussagen über denselben Aspekt der Software-Entwicklung (wie z.B. die Produktivität), die in der Hypothesensammlung enthalten sind.

Trotz der genannten Schwierigkeiten und Gefahren trägt aber der vorgestellte Ansatz wesentlich zur Konservierung und zur Konsolidierung bisher erzielter Forschungsergebnisse bei. Darüber hinaus kann mit diesem Ansatz eine erste Grundlage für die Modellierung geschaffen werden. Die so entstandenen Modelle müssen dann schrittweise verbessert und verfeinert werden.

3. Validierung des SESAM-Modells

Während die im letzten Kapitel vorgestellte Arbeit die Grundlagen für die Modellierung von Software-Projekten schaffen sollte und lange vor der ersten Implementierung des SESAM-Systems stattfand, ist in der jetzt diskutierten Diplomarbeit das bisher in SESAM enthaltene Modell als Ausgangsbasis verwendet worden.

Ziele der Arbeit

In dieser Arbeit sollten die Regeln des in Kapitel 1 skizzierten SESAM-Modells überprüft, also auf Vollständigkeit, Genauigkeit und besonders auf Realitätsnähe hin untersucht werden. Dazu sollten – ähnlich wie in der letzten Arbeit – eine Literaturanalyse durchgeführt und die relevanten Daten zusammengestellt und verdichtet werden. Die sich aus den Daten ergebenden Aussagen sollten mit den SESAM-Regeln verglichen werden.

Das Literaturstudium wurde durch eine Praktikerbefragung ergänzt, um diejenigen Regeln validieren zu können, für deren Prüfung in der Literatur keine Daten vorhanden waren.

Ergebnisse der Arbeit

Die Ergebnisse dieser Arbeit sind durchaus ermutigend für die Zukunft. Im Rahmen der Literaturanalyse sind ingesamt 14 Quellen untersucht worden. Sie ist nach verschiedenen Aspekten der Software-Entwicklung gegliedert (z.B. Fehler, Aufwandsschätzung oder Produktivität der Mitarbeiter) und liefert eine Fülle quantitativer Aussagen. Angaben, die von den Autoren nicht durch Zahlen belegt worden sind, wurden in dieser Arbeit nicht berücksichtigt.

Im Rahmen der Primäranalyse, also der Befragung, sind elf Praktiker aus acht verschiedenen Unternehmen interviewt worden, wobei ein auf der Basis der empirischen Sozialforschung ausgearbeiteter Fragebogen eingesetzt wurde. Obwohl die resultierende Datenmenge relativ klein ist, konnte durch spezifische Fragen zum ersten Mal die "Realitätsnähe" des SESAM-Modells geprüft werden.

Die auf der Basis der Analyse durchgeführte Validierung ergab, daß das SESAM-Modell im großen und ganzen zutreffend ist, jedoch an einigen Stellen erweitert und verbessert werden muß. Kurz zusammengefaßt stellte Feest (1993) folgendes fest:

- Die Kosten für Mitarbeiter und Stellenanzeigen sind im Modell zu niedrig angesetzt worden.

- Die im Modell angenommene Zahl der Fehler, die durch die Bearbeitung von den Mitarbeitern in die Dokumente eingefügt werden, ist zu hoch.
- Die Produktivität eines simulierten Mitarbeiters, gemessen in LOC/Tag, muß deutlich reduziert werden.
- Die Kommunikation der Mitarbeiter ist zu wenig berücksichtigt worden.
- Die Motivation der simulierten Mitarbeiter ist zu grob modelliert worden.
- Die Mitarbeiter benötigen neben Fähigkeit, Erfahrung und Motivation weitere Persönlichkeitsmerkmale, damit die von ihnen erbrachte Leistung realistischer modelliert werden kann. Nach Meinung des Autors fehlen z.B. Merkmale wie Kooperations- und Kommunikationsfähigkeit oder Selbständigkeit.
- Die simulierten Mitarbeiter werden in bezug auf Fähigkeiten und Erfahrung zu schlecht bewertet. Im SESAM-Modell wird jedem Mitarbeiter eine Analyse-, Entwurfs- und Programmiererfahrung zugeordnet. Zur Prüfung dieser Werte wurden im Interview Fragen in bezug auf die Fähigkeiten der Mitarbeiter der jeweiligen Unternehmen eingebaut. Die Antworten wurden in den im Modell verwendeten Wertebereich transformiert, um sie mit den im Modell verwendeten Daten vergleichen zu können. Das Ergebnis des Vergleichs wird in Abbildung 2 gezeigt.

Abb. 2: Vergleich der Fähigkeiten der Mitarbeiter

Es ist zu erkennen, daß die Fähigkeiten der SESAM-Mitarbeiter niedrigere Werte aufweisen als die der "echten" Mitarbeiter in Software-Projekten.

Diskussion des Ansatzes

Generell sind durch die beschriebene Vorgehensweise in diesem Projektstadium die erwarteten Ergebnisse erzielt worden. Dennoch sind auch zu diesem Ansatz einige Bemerkungen notwendig.

Der Kritikpunkt, daß durch die Literaturanalyse die Daten oft aus ihrem ursprünglichen Zusammenhang gerissen und die Aussagen, die von den Autoren aus den Daten abgeleitet werden, häufig recht spekulativ sind, gilt leider auch für den hier beschriebenen Ansatz.

Darüber hinaus war zu beobachten, daß sowohl die Daten aus der Literatur als auch die Angaben der Praktiker sehr stark streuen. Die in dieser Arbeit verwendete Datenmenge war aber noch sehr klein, so daß auch eine statistische Auswertung keine gültigen Resultate gewährleisten kann. Außerdem ist bei der Durchführung der Interviews deutlich geworden, daß vielfach in den Unternehmen keine konkreten Daten vorlagen, sondern die angegebenen Werte grob geschätzt worden sind, so daß einige Zweifel hinsichtlich der Zuverlässigkeit der Daten angebracht sind.

Trotz der genannten Probleme konnte durch die Arbeit aber gezeigt werden, daß es mit diesem Ansatz grundsätzlich möglich ist, die SESAM-Modelle zu validieren. Positiv ist auch, daß Kenntnisse einer anderen Fachrichtung für eine empirische Untersuchung in der Informatik genutzt worden sind.

4. Untersuchung realer Software-Projekte

Die letzte, hier präsentierte Arbeit wurde ebenfalls unabhängig von SESAM durchgeführt. Es handelt sich dabei um eine rein empirische Untersuchung der "Realität der Software-Erstellung" in einer konkreten industriellen Entwicklungsumgebung.

Diese Arbeit bildet gewissermaßen das Gegenstück zu der in Kapitel 2 beschriebenen, während die in Kapitel 3 vorgestellte Arbeit beide Ansätze vereint.

Ziele der Arbeit

Ziel der Arbeit war, in einer "echten" Software-Entwicklungsumgebung Daten zu erheben, um Aussagen über die beobachteten Projekte treffen zu können und Gesetzmäßigkeiten zu erkennen.

Dafür sollte zunächst bestimmt werden, welche Attribute welcher Objekte des Prozesses überhaupt für seine Beurteilung relevant sind und durch welche Metriken die Attribute möglicherweise bewertet werden können.

Auf dieser Grundlage sollten die konkreten Projektdaten erhoben und für eine Analyse des Entwicklungsprozesses eingesetzt werden.

Ergebnisse der Arbeit

Im Rahmen der Arbeit wurde ein Schema mit den wichtigen Objekten und Beziehungen und den sie charakterisierenden Attributen erstellt. Für die Attribute ist definiert worden, durch welche Metriken sie bewertet werden können.

Das Schema ist als Anleitung für die eigentliche Datenerhebung im Unternehmen verwendet worden. In der Abteilung wurden elf (überwiegend abgeschlossene) Projekte untersucht, in denen von meist einem Bearbeiter kleine Datenbank-Applikationen erstellt worden sind.

Die resultierenden Daten wurden zunächst für die Beschreibung der einzelnen Projekte verwendet und dann verdichtet, um damit die "typische Vorgehensweise" bei der Software-Entwicklung in der Abteilung zu beschreiben.

Die Analyse der kumulierten Projektdaten hat u.a. folgendes ergeben (Drappa, 1993):

- Für die Programmierung sind sehr hohe Sprachen und leistungsfähige Werkzeuge (Code-Generatoren) eingesetzt worden; es gab jedoch in keiner anderen Phase Unterstützung durch Methoden oder Tools.
- Es ist praktisch keine Dokumentation erstellt worden, auch keine Benutzerhandbücher für die Applikationen.
- Es waren keine Richtlinien zum Vorgehen bei der Software-Entwicklung vorhanden.
- Das gesamte Management der Projekte wurde sehr informal betrieben; es gab kaum Unterlagen zur Projektplanung und daher auch wenig Termin- und Fortschrittskontrollen.
- Der Aufwand entfiel trotz der in der Implementierung verwendeten Hilfsmittel hauptsächlich auf die Codierphase.

Die Datenerhebung ermöglichte auch die Ableitung einiger Gesetzmäßigkeiten für die untersuchte Abteilung. Durch die Erfassung des in die Applikationen investierten Aufwands und die Analyse des dafür erzeugten Codes konnten beispielsweise Aussagen über die durchschnittliche Produktivität der Mitarbeiter der Abteilung getroffen werden. Dabei wurde der (sehr hohe) Wert von fast 2000 LOC/Mitarbeitermonat berechnet, der durch den Einsatz eines Code-Generators und weitere spezielle Randbedingungen in der Abteilung erklärt werden konnte.

Auf der Grundlage der Datenauswertung wurden schließlich Vorschläge für Veränderungen des Entwicklungsprozesses ausgearbeitet und ein Metrikprogramm zur zukünftigen Steuerung und Analyse der Software-Projekte des Unternehmens erstellt.

Diskussion des Ansatzes

Die Erhebung und Verwendung echter Projektdaten für die Modellierung in SESAM ist prinzipiell ein wichtiger Ansatz, allerdings sind auch hier einige Schwierigkeiten aufgetreten.

In dem untersuchten Unternehmen war eine sehr spezielle Entwicklungsumgebung vorhanden (Einsatz von 4-GL, Code-Generator), so daß nur wenige der bekannten Metriken, insbesondere für die Bewertung des Codes, einsetzbar waren. Um überhaupt zu Aussagen zu kommen, mußten z.T. neue Maße definiert werden.

Daten über den Software-Entwicklungsprozeß waren in der Abteilung so gut wie nicht verfügbar. Deshalb mußte ebenfalls die Befragungstechnik eingesetzt werden, wobei die Bearbeiter die Werte rückblickend geschätzt haben. Die resultierenden Daten sind daher teilweise widersprüchlich und ungenau gewesen.

Durch die spezielle Entwicklungsumgebung sind darüber hinaus die Aussagen kaum verallgemeinerbar und damit auf andere Umgebungen übertragbar gewesen. Diese Erkenntnis hat allerdings auch zu der Einsicht geführt, daß nicht das *typische* Software-Projekt modelliert werden kann, sondern verschiedene Umgebungen auch verschiedene Modelle mit jeweils spezifischen Regeln erfordern.

5. Fazit und Ausblick

Modelle sind grundsätzlich entweder Abbilder von etwas oder Vorbilder für etwas. Nach Ludewig (1989) weisen Modelle immer die folgenden drei Merkmale auf:

- das Abbildungsmerkmal, d.h. zum Modell gibt es ein Original, das wirklich vorhanden, geplant oder fiktiv sein kann,
- das Verkürzungsmerkmal, d.h. daß im allgemeinen nicht alle Attribute eines Originals, sondern nur die für einen bestimmten Zweck relevanten erfaßt werden, sowie
- das pragmatische Merkmal, d.h. das Modell kann unter bestimmten Bedingungen und für bestimmte Fragestellungen das Original ersetzen.

Bezogen auf die hier diskutierten Modelle von Software-Projekten in SESAM, hat sich durch die Arbeiten deutlich gezeigt, worin die Problematik der Modellbildung speziell besteht.

- Es gibt nicht *das* eine Original, nicht das Software-Projekt für das ein Modell erstellt und anschließend validiert werden kann (Abbildungsmerkmal). Vielmehr hat sich gezeigt, daß zumindest jede Entwicklungsumgebung ein eigenes Modell erfordert, in dem die spezifischen Gegebenheiten berücksichtigt werden.

- Es ist bisher nicht klar, welches die wesentlichen Faktoren sind, die ein Projekt beeinflussen. Daher ist es schwierig, diejenigen Attribute des Originals (des Software-Projekts) zu identifizieren, die auch im Modell enthalten sein müssen, damit die wesentlichen Aspekte der Realität in das Modell abgebildet werden können (Verkürzungsmerkmal).
- Es ist schwierig zu entscheiden, wann ein Modell detailliert genug ist, um seinen Zweck, das Sammeln von Projekterfahrung für eine bestimmte Kategorie von Software-Projekten, erfüllen zu können (pragmatisches Merkmal).

Die genannten Probleme sollen durch weitere, vornehmlich emprische Arbeiten, zumindest in Ansätzen gelöst werden. Dabei sind im wesentlichen zwei Vorgehensweisen denkbar. Auf der Grundlage der in einer speziellen Entwicklungsumgebung erhobenen Daten könnten unternehmensspezifische Modelle erstellt werden, die dann von den Mitarbeitern des Unternehmens durch Benutzung erprobt werden. Dadurch könnten die Modelle schrittweise verfeinert und an die Realität angepaßt werden.

Um sich aber nicht nur unternehmensspezifische Modelle zu konzentrieren, wäre es darüber hinaus denkbar, umfassende Untersuchungen in vielen verschiedenen Unternehmen durchzuführen. Auf diese Weise könnten wahrscheinlich allgemeingültige Regeln identifiziert werden, die schließlich die Grundlage aller SESAM-Modelle bilden und nur noch um unternehmensspezifische Regeln ergänzt werden müßten.

Literatur

Deininger, M., Schneider, K. (1994): Teaching Project Management by Simulation. **Proceedings of the 7th Conference on Software Engineering and Education (CSEE)**, San Antonio, Januar 1994, pp. 227-242.

Drappa, A. (1993): **Konzeption und Einführung eines Metrikprogramms in einem Software-Projekt.** Diplomarbeit, Univ. Stuttgart, 1993.

Feest, R. (1993): **Validierung von SESAM-Modellen anhand von Aufwandsschätzverfahren für Software-Projekte und von Praktikerbefragungen.** Diplomarbeit, Univ. Stuttgart, 1993.

Ludewig, J. (1989): Modelle der Software-Entwicklung - Abbilder oder Vorbilder? **Softwaretechnik-Trends**, Oktober 1989, pp. 1-12.

Schneider, K. (1993): Object-Oriented Simulation of the Software Development Process in SESAM. **Proceedings of the Object-Oriented Simulation Conference (OOS ´93)**, Teil der Western Simulation Multiconference, San Diego, Januar 1993.

Utz, A. (1992): **Sammlung und Darstellung von Hypothesen über Software-Projekte.** Diplomarbeit, Univ. Stuttgart, 1992.

Teil 6

SESAM und die Lehre

Marcus Deininger

Zusammenfassung

In der Software Engineering-Ausbildung haben Studenten selten die Gelegenheit, Erfahrungen bei der Leitung von Projekten zu sammeln – dies ist ein schwerwiegender Mangel in der Ausbildung zum „Software Ingenieur“. In der Software Engineering-Ausbildung erschwert das Fehlen der Erfahrung sehr oft das Verständnis für die Probleme der Software-Entwicklung, im späteren Berufsleben könnten viele Einstiegsschwierigkeiten vermieden werden.

In diesem Beitrag wird gezeigt, wie die Software Engineering-Ausbildung mit Hilfe von Simulation unterstützt werden kann. Studenten durften ein simuliertes Projekt führen. Um willkürliche Effekte auszuschalten und zu nachvollziehbaren Ergebnissen zu gelangen, wurde für die Simulation ein sehr einfaches, mathematisches Simulationsmodell gewählt. Im Verlauf der Simulation stellte sich heraus, daß dieses Modell mehr als ausreichend war: Viele bekannte Projekt-Effekte stellten sich während der Simulation ein. Die „Projekt Manager“ machten typische Fehler, die zu plausiblen Effekten im Projekt führten. Am Ende der Projekte hatten Studenten ein besseres Verständnis für die Aufgaben eines Projektleiters, die Tutoren ein besseres Verständnis für die Simulation von Software-Projekten.

Die Simulation war eine Phase des Projekts SESAM. SESAM soll einen allgemeinen Rahmen zur Simulation von Software-Projekten liefern. Um den Rahmen mit einem Modell zu füllen, wurde das in diesem Beitrag beschriebene Modell entwickelt und zunächst „trocken“, d.h. ohne Simulator, in einem Fachpraktikum durchgespielt.

1. Einleitung

In der Software Engineering-Ausbildung an einer Universität stehen wir vor folgendem Problem: Wir lehren die Grundlagen des Software Engineerings; Studenten haben aber praktisch keine Gelegenheit, dieses Wissen in einem echten Projekt anzuwenden. Sie können selten während ihres Studiums an einem Projekt teilzunehmen und wenn, dann nicht als Projektleiter.

In diesem Betrag beschreibe ich die Erfahrungen, die wir während eines Fachpraktikums an unserer Abteilung mit der *Simulation* von Software-Projekten gesammelt haben. Dieser Ansatz erlaubt uns, Projektleiter-Erfahrungen innerhalb der beschränkten Möglichkeiten der Universität zu vermitteln. Statt eines echten Projekts waren Studenten Leiter eines simulierten Projekts, das sie erfolgreich zu Ende bringen sollten. Das zugrunde liegende Simulationsmodell war sehr einfach, erlaubte aber trotzdem den gesamten Software-Entwicklungsprozeß abzudecken.

2. Das Fachpraktikum innerhalb des Projekts SESAM

Das Fachpraktikum, das in diesem Beitrag beschrieben ist, war eine Phase unseres Projekts SESAM (**S**oftware **E**ngineering **S**imulation durch **A**nimierte **M**odelle). Wir haben über SESAM in Ludewig et al. (1992), Schneider (1993), Schneider (1993a), Deininger/Schneider (1994) und Schneider/Deininger (1994) ausführlich berichtet, ich werde deshalb nur einen kurzen Überblick geben und das Fachpraktikum einordnen.

2.1 Das Projekt SESAM

SESAM soll einen allgemeinen Rahmen zur Simulation von Software-Projekten liefern. Teilziele von SESAM sind:

- Dynamische und animierte Modelle von Software-Entwicklungsprojekten.
- Graphische Editoren, um diese Modelle zu formulieren, zu speichern und zu ändern. (SESAM-Modelle müssen leicht änderbar sein, um leicht an neue Situationen angepaßt werden zu können.)
- Geeignete Simulationswerkzeuge für diese Modelle.
- Ein graphisches interaktives Abenteuerspiel, das es Studenten ermöglicht, ein Projekt ohne Tutor durchzuspielen.

Bis zum Sommer 1992 waren drei Prototypen fertiggestellt, mit denen wir verschiedene Aspekte der Simulation untersucht haben. Nächstes Ziel war die Entwicklung eines *Modells*, das zur Simulation eingesetzt werden konnte. Da zu diesem Zeitpunkt Notation und Mächtigkeit der Modelle noch offen war, machte es keinen Sinn dafür einen weiteren Simulator-Prototyp zu entwickeln. Statt dessen haben wir uns entschlossen, zunächst ein sehr einfaches

Modell „trocken“, d.h. ohne Simulator, in einem Fachpraktikum durchzuspielen.

Das Simulationsmodell war erfolgreicher als wir es zunächst erwartet hatten:

- Es wurden eine ganze Reihe typischer Projekteffekte evoziert, die auch in wirklichen Projekten dem Projektleiter das Leben schwer machen.
- Die Studenten gewannen einen Eindruck von den Sorgen und Nöte eines Projektleiters, die Tutoren – und damit die Abteilung Software Engineering – gewannen Erfahrung in der Modellierung und Simulation von Software-Projekten.

Damit haben wir unser ursprüngliches Ziel, nämlich den prinzipiellen Aufbau geeigneter Simulationsmodelle für SESAM, erreicht. Darüber hinaus haben wir mit diesem einfachen Modell eine Möglichkeit geschaffen, solche Simulationen auch von Hand durchzuführen und erfolgreich im Unterricht einzusetzen.

2.2 Verwandte Arbeiten

Die Simulation von Software-Projekten ist keine neue Idee. Abdel-Hamid(1991) beschreibt eine Software-Projekt-Simulation, die auf System Dynamics basiert. Anders als unser Modell ist dieses Modell eine „closed loop simulation“, d.h. auch die Projekt-Manager werden simuliert. Der Benutzer der Simulation beobachtet nur den Simulationsverlauf, kann aber nicht mit dem System interagieren. Zu Beginn wird der Anfangszustand eingegeben, danach simuliert der Simulator den Projektverlauf, das Ergebnis der Simulation kann am Ende des Laufs begutachtet werden. Im Gegensatz dazu haben wir eine „open loop simulation“: in der Simulationsumgebung können die Spieler während der Simulation Entscheidungen treffen und eingreifen.

McKeeman (1989) berichtet über ein Tutor-Programm zur Schulung von Software-Entwicklern. Das Programm hilft Entwicklern die Prinzipien von Reviews zu lernen. Das Programm hat Spielcharakter, wie unsere Simulation, beschränkt sich aber auf einen kleinen Ausschnitt des Entwicklungsprozesses.

Einen Schritt weiter, aber auf einem ganz anderen Gebiet, geht Vester (197). Er hat ein Spiel namens „Ökolopoly“ entwickelt. Der Spieler ist Präsident eines fiktiven Landes und hat die Aufgabe, die ökonomischen und ökologischen Probleme dieses Landes zu lösen. Dazu stehen ihm eine Reihe von Entscheidungsmöglichkeiten und Zügen zur Verfügung. Auch hier sind die Abhängigkeiten mit Hilfe von System Dynamics modelliert. Das Grundprinzip, also die offene Schleife und ein interagierender Spieler, ist dasselbe wie unseres.

3. Software Engineering-Ausbildung an der Universität Stuttgart

Die Software Engineering-Ausbildung an der Universität Stuttgart zielt darauf ab, Studenten das nötige Wissen zu vermitteln, damit sie später erfolgreich an Software-Projekten teilnehmen oder diese leiten können. Dieses Ziel verfolgten wir zunächst mit den üblichen Mitteln: Auf der Basis der Bücher von Fairley (1986) und Sommerville (1989) haben wir eine Vorlesung angeboten, die die Grundlagen des Software Engineerings vermittelt. In dieser Vorlesung werden zunächst die grundlegenden Prinzipien des Software Engineerings diskutiert; Themen sind u.a. Projekt-Management, der Software-Life-Cycle, Qualitätssicherung und Software-Metriken. Im zweiten Teil der Vorlesung werden, entlang der Phasen des klassischen Wasserfall-Modells, die wesentlichen Aktivitäten, Methoden, Notationen und Ergebnisse dieser Phasen diskutiert. Am Ende der Vorlesung sollten die Studenten die wesentlichen Konzepte des Software Engineerings kennen.

Diese Vorlesung wird durch begleitende Übungen ergänzt. In diesen Übungen werden die in der Vorlesung vorgestellten Konzepte in Fall-Studien und Beispielen eingesetzt. Allerdings können mit solchen Übungen nur einzelne Schwerpunkte vertieft werden, sie sind kein Ersatz für die Mitarbeit in einem Projekt. Dieser Mangel führte dazu, daß wir die folgenden Praktika angeboten oder bestehende Praktika nach unseren Vorstellungen umgestaltet haben:

- **Software-Praktikum.** In einer Gruppe von 3-4 Personen führen Studenten ein kleines, aber vollständiges Projekt durch. Das Projekt beginnt mit einer Aufgabenstellung, zu der nacheinander Spezifikation, Entwurf und Code entwickelt werden müssen. Das Projekt dauert ca. ein halbes Jahr. Gruppenleiter sind Mitarbeiter unserer Abteilung. Am Ende haben die Studenten einen vollständigen Software-Life-Cycle mitgemacht und zum ersten Mal Erfahrungen in Projektarbeit gemacht.
- **Fachpraktikum „Software Engineering“.** Im Software-Praktikum können Studenten ein Projekt vom Standpunkt des Analytikers und Programmierers miterleben. Im Fachpraktikum „Software Engineering“ erhalten sie zusätzlich einige Managementaufgaben: neben der Aufgabenstellung bekommen sie einen Zeitplan, der ihnen die Phasen und Meilensteine vorgibt. Die Durchführung des Projekts im Rahmen dises Plans liegt ganz in ihren Händen, insbesondere die Planung und Durchführung von Reviews, die Aufteilung in Arbeitspakete oder die Festlegung von Schnittstellen.

Um den Schwerpunkt auf die Planung und Organisation legen zu können, sind die Aufgaben, die bearbeitet werden, eher einfach. Allerdings werden nach jedem Meilenstein die Projektergebnisse (Spezifikation, Entwurf und Module) zyklisch zur nächsten Gruppe weitergegeben. Dies führt dazu, daß die Studenten auch mit fremden Dokumenten arbeiten müssen, was ein gutes Gefühl für die Qualität der Dokumente schafft.

Am Ende der Projekte haben die Studenten eine ganze Reihe von Problemen, sowohl aus Sicht der Entwickler als auch aus Sicht des Leiters, kennengelernt: knappe Zeitpläne, die Auswirkungen fehlerhafter Spezifikationen, die Effekte unzureichender Schnittstellendefinitionen, usw.

- **Fachpraktikum „Projekt Management".** Hauptnachteil der vorigen Praktika (aus Sicht unserer Abteilung) waren zwei sich widersprechende Ziele: Studenten sollten einerseits Erfahrungen als Projekt-Mitarbeiter sammeln (was sie nach dem Software-Praktikum bereits haben), andererseits sollten sie Erfahrungen als Projektleiter sammeln (etwas, was sie noch nie zuvor gemacht haben). Um beide Erfahrungen zu ermöglichen, waren die Projekte einfach. Allerdings evozierten solche Projekte nicht alle von uns gewünschten Effekte. Wir haben uns deshalb entschlossen, ein neues Praktikum ins Leben zu rufen, bei dem sich die Studenten vollständig auf das Projekt-Management konzentrieren können, aber keine Entwicklungsarbeit leisten sollten – die sollte von anderen gemacht werden. Damit sollte es möglich sein größere und komplexere Projekte zu bearbeiten.

4. Randbedingungen des Fachpraktikums

4.1 Ziele des Fachpraktikums

Das Hauptziel des Fachpraktikums war, den Studenten „echte" Projektleiter-Erfahrungen zu vermitteln, ohne daß sie irgendwelche Entwicklertätigkeiten durchführen mußten. Sie sollten soviel wie möglich an Software Engineering-Wissen einbringen und einsetzten können, insbesondere sollten sie die folgenden Effekte kennenlernen:

- Planung ist unerläßlich für ein Software-Projekt! Ein fehlendes oder mangelhaftes Prozeßmodell führt zu einem mangelhaften Produkt. Die Studenten sollten lernen, zu planen und ihre Planung dem Projektverlauf anzupassen.
- Quantität ist nicht Qualität! Erhöhter Aufwand führt zu größeren Produkten, aber nicht notwendigerweise zu mehr Qualität. Die Produkt-Qualität wird sich nur verbessern, wenn ausdrücklich Qualitätssicherung betrieben wird. Fehlende Qualitätssicherung führt zu Inkonsistenzen zwischen den Dokumenten der einzelnen Phasen.
- Die beste Möglichkeit der Qualitätssicherung (vor allem in den frühen Phasen) sind Reviews! Allerdings müssen Reviews gut vorbereitet werden, um erfolgreich zu sein. Erfolgreiche Reviews finden viele Fehler.
- Die Wünsche des Kunden sollen erfüllt werden und nicht, was sich die Entwickler an Stelle des Kunden wünschen würden!
- Die Fähigkeiten der Entwickler sollen realistisch eingeschätzt werden – sie können keine Wunder vollbringen!
- Und schließlich: Der Projektleiter ist *allein* für den Erfolg oder Mißerfolg des Projekts verantwortlich und niemand sonst!

Veranstaltung	Teilnehmer	Aufwand der Studenten
Vorlesung „Software Engineering"	120 Studenten, 5. - 7. Semester	15 Wochen, 4 h / Woche
Übungen zur Vorlesung „Software Engineering"	120 Studenten, 5. - 7. Semester	Begleitend zur Vorlesung, zweiwöchentlich, jeweils 2 h
Software Praktikum	ca 10 Gruppen à 3-4 Studenten, 3. Semester	20 Wochen, ca. 4 h / Woche
Fachpraktikum „Software Engineering"	ca 6 Gruppen à 3-4 Studenten, 7. Semester	12 Wochen, ca. 8 h / Woche
Fachpraktikum „Projekt Management"	ca 5 Gruppen à 2 Studenten, 7. Semester	12 Wochen, ca. 8 h / Woche

Tab. 1: Überblick über die Lehraktivitäten unserer Abteilung. (Nicht enthalten sind Vertiefungsvorlesungen zu speziellen Gebieten des Software Engineerings)

4.2 Organisatorischer Rahmen

Das Fachpraktikum fand während eines Sommersemesters statt, damit standen zwölf Wochen zur Verfügung. Für Studenten ist ein Aufwand von acht Stunden pro Woche für eine Veranstaltung dieser Art vorgesehen.

Dabei sollten die folgenden Rahmenbedingungen eingehalten werden:

- Alle Teilnehmer sollten sich ausschließlich auf die Projektleitung konzentrieren können; es sollte kein Aufwand für Software-Entwicklung getrieben werden.
- Die Projektleiter sollten innerhalb der zur Verfügung stehenden zwölf Wochen alle wesentlichen Erfahrungen eines Projektleiters machen.

  Für ein reales Projekt waren diese Forderungen nicht vereinbar.
- Ein reales Projekt, das in zwölf Wochen durchführbar ist, ist zu klein, um die von uns gewünschten Probleme zu verursachen.
- Ein größeres Projekt würde innerhalb der zwölf Wochen nicht über die frühen Phasen hinauskommen und damit ebenfalls nur einen Teil der Probleme verursachen.
- Ein echtes Projekt zu Übungszwecken stand nicht zur Verfügung, und würde von uns nicht genügend kontrolliert werden können.
- In einem echten Projekt müßte jemand die Entwicklerarbeit tun.
- Wir hatten ber weder die Zeit noch die Mittel, echte Projekte für dieses Praktikum durchzuführen.

Um innerhalb dieser Bedingungen zu arbeiten, haben wir uns entschlossen ein Projekt zu *simulieren*. Eine Simulation erfüllt alle die oben aufgestellten Forderungen. Die Simulation sollte offen sein, d.h. die Spieler sollten während der Simulation ins Geschehen eingreifen können. Simuliert werden sollten alle Entwicklungsaktivitäten (und damit natürlich auch die Entwickler) und die entstehenden Ergebnisse (also die Software). Die Studenten sollten lediglich als Manager in das Projekt eingreifen können und ihren Mitarbeitern Anweisungen erteilen können.

5. Elemente der Simulation

5.1 Das simulierte Projekt

Alle Projektleiter wurden unabhängig voneinander dem gleichen Projekt zugeteilt. Sie sollten für einen externen Kunden ein Scheckschreibungsprogramm entwickeln. Zu Beginn erhielten die Projektleiter eine vierseitige Analyse über das Projekt. Diese Analyse beschrieb die wesentlichen funktionalen Anforderungen des zu entwickelnden Produkts. Die Analyse war das einzige echte Stück Software, das die Spieler zu sehen bekamen – die andere Software, wie z. B. der Code wurde nur simuliert.

Die Analyse entstammte einem Buch über Kostenschätzungsmethoden (Knöll 1990). Sie war ein Beispiel für die Anwendung der Function Point-Methode. Diese Methode lieferte für unser Beispielprojekt 234 Function Points, einen erwarteten Aufwand von 17 Mitarbeiter Monaten und eine Dauer von 7 Monaten. Keine dieser Informationen wurde den Spielern gegeben. Die Spieler erhielten einen knapperen Zeitplan: Das Projekt sollte in 6 Monaten mit einem geplanten Budget von 250.000.- DM abgeschlossen werden. Insgesamt war ein Festpreis von 400.000,- DM für das Produkt ausgemacht. Diese Vorgaben waren von einem „anonymen“ Management gesetzt worden. Sowohl Zeit als auch Budget waren von vorne herein etwas zu knapp gesetzt, um die Spieler im Simulationsverlauf zu Kompromissen zu zwingen.

5.2 Der Simulationsrahmen

Die Simulation mußte innerhalb von zwölf Wochen durchgeführt werden. Die Studenten konnten pro Woche einen Zug machen. Ein Zug umfaßte die die folgenden Bestandteile:

- Eine Liste aller Aktionen (einschließlich ihrer Dauer), die in dem simulierten Projekt durchgeführt werden sollten,
- Erläuterungen zu den Aktionen,
- die erwarteten Ergebnisse der Aktionen
- und alle Dokumente, die eine echter Projektleiter im entsprechenden Projekt erstellen würde, wie Zeitpläne, Mitarbeiterbewertungen, usw.

Im Gegenzug erhielten die „Manager“ zwei Tage später als Ergebnis alle Informationen und Dokumente, die ein echter Projektleiter an ihrer Stelle erhalten würde; insbesondere wurden ihnen Kommentare und Bemerkungen ihrer simulierten Mitarbeiter mitgeteilt – aber keine Software!

Zu Beginn der Simulation war von uns noch nicht festgelegt worden, welche Aktionen durchgeführt werden dürfen. Jede echte Projektleitertätigkeit sollte erlaubt sein, verboten waren lediglich „Wunder“, wie „Ich stelle einen Assistenten ein, der das Projekt führen wird“. Trotz dieser unbeschränkten Möglichkeiten, wurden tatsächlich nur die folgenden Aktionen durchgeführt:

- Einer oder mehrere der simulierten Projektmitarbeiter sollen eine Aufgabe ausführen. Mögliche Aufgaben waren:
  - Anforderungssanalyse schreiben
  - Architekturentwurf schreiben
  - Modulentwurf schreiben
  - Modul codieren

  - Benutzerhandbuch schreiben
  - Modultest vorbereiten und ausführen
  - Integrationstest vorbereiten und ausführen
  - Systemtest vorbereiten und ausführen
  - Review vorbereiten
  - Dokument reviewen
  - die in Tests oder Reviews gefundene Fehler korrigieren
- Der Kunde soll in eine Aktion einbezogen werden (an einem Review teilnehmen, an einer Besprechung teilnehmen, an einem Geschäftsessen teilnehmen, usw.).
- Informationen über verfügbare Schulungen erfragen.
- Mitarbeiter zu einer der Schulungen schicken.
- Informationen über verfügbare Software-Werkzeuge erfragen.
- Ein Werkzeug kaufen.
- Einen Berater konsultieren.
- Eine Stelle ausschreiben.
- Einen Bewerber einstellen oder einen Mitarbeiter entlassen.
- Mit dem Management oder dem Kunden ein höheres Budget oder Verschiebung des Projektendes aushandeln.
- Eine Reihe von Social Events, von einer Einladung zum Essen bis zu einer Wochenendreise für alle Mitarbeiter. (Diese Aktionen verbesserten vor allem die Motivation der Mitarbeiter, hatten aber keinen direkten Einfluß auf en Fortgang der Simulation.)

Im Gegenzug erhielten die Spieler folgende Informationen:

- Alle Aktivitäten hatten zunächst ein quantitatives Ergebnis. Den Spielern wurde mitgeteilt, wie viele Seiten Spezifikation, wie viele Zeilen Code geschrieben wurden, wie viele Fehler in einem Review gefunden wurden. Da sie *keine* weitere Software erhielten, mußten sie ihre Entscheidungen allein auf diesen Projektdaten begründen.
- Auf Anfragen erhielten die Spieler Prospekte für Schulungen. Es gab Prospekte zu den folgenden Schulungsthemen: COBOL, Ada, Test, Strukturierte Programmierung, Structured Design und Kostenschätzung. Die Prospekte wurden nur verschickt, falls ein Spieler eine entsprechende Anforderung machte. Wurde einer der simulierten Mitarbeiter auf eine Schulung geschickt, verbesserten sich seine Fähigkeiten in dem entsprechenden Gebiet.
- Auf Anfrage wurden Prospekte für Software-Werkzeuge zurückgegeben: drei verschiedene CASE-Tools, ein Test-Werkzeug, eine Standard-Datenbank, Ada- und COBOL-Compiler. So wie bei den Schulungen erhöht ein Werkzeug die Produktivität – nach einer angemessenen Schulungszeit.
- In einer persönlichen Beratung konnten sich die Manager selbst beraten lassen (alle anderen Schulungen waren nur für die simulierten Mitarbeiter). Die Berater wurden von uns dargestellt. In der Beratung konnten die Spieler Hinweise auf den Spielverlauf bekommen – natürlich zu einem angemessenen (simulierten) Preis.
- Bewerbungsschreiben von Stellenanwärtern.
- Die Spieler wurden immer über alle aktuell angefallenen Kosten informiert, sie mußten aber selbst die Kosten verfolgen und den Überbllick behalten.

Alle diese Informationen wurden immer in einen kurzen Text gepackt, um so der Simulation einen lebendigeren Anstrich zu geben.

6. Das Simulationsmodell

Als Basis für die Simulation wurde ein einfaches mathematisches Modell verwendet. Dieses Modell wurde manuell, nur unterstützt von einem Spreadsheet-Programm ausgeführt. Ein solches Modell erlaubte nachvollziehbare und objektive Reaktionen, die konsistent über alle Simulationen waren. Das Modell sollte so einfach wie möglich sein, nur so würde das Ergebnis überprüfbar bleiben. Eine solche Überprüfung ist die Voraussetzung für Validierung und Vermeidung von Berechnungsfehlern.

Ausgangspunkt für das Modell waren die folgenden Zielkriterien für ein erfolgreiches Software-Produkt. Nach Frühauf *et al.* (1988) ist ein Projekt erfolgreich, wenn

- das Projekt in der vorgegebenen Zeit durchgeführt wird,
- das Projekt mit der vorgegebenen Budget durchgeführt wird,
- am Ende sowohl Kunde als auch Mitarbeiter zufrieden sind und
- das fertige Produkt die geforderte Qualität besitzt.

6.1 Das statische Simulationsmodell

Diese Ziele wurden in unserem Modell folgendermaßen repräsentiert:

Zeit und Budget

Die Kosten aller Aktionen waren zuvor festgelegt worden. Während der Simulation mußten Zeit und Geld einfach aufaddiert werden.

Motivation und Zufriedenheit

Alle Aktionen hatten Auswirkungen auf die Motivation und Zufriedenheit von Projekt-Mitarbeitern und Kunde. Die Motivation wurde durch sog. „Motivations-Punkte“ gezählt. Jeder Mitarbeiter und der Kunde hatten ein eigenes Motivations-Konto, auf das einfach Punkte aufaddiert oder abgezogen wurden.

Produktqualität

Während die Simulation von Zeit, Geld und Motivation relativ einfach war, standen wir bei der Verfolgung der Produktqualität vor einem großen Problem. Da wir keine echten Dokumente erzeugen wollten, mußten wir den Inhalt der Dokumente simulieren.

Entstehungszeitpunkt, aber geben keinen Hinweis auf die Natur der Fehler.

Wir mußten deshalb eine neue Metrik zur Beschreibung des Dokumenteninhalts entwickeln. Dabei gingen wir von folgendem Modell aus: Wesentlich für ein Software-Projekt ist es, daß die ursprünglichen Anforderungen des Kunden über die Spezifikation und den Entwurf in das endgültige Produkt übertragen werden. Bei dieser Übertragung dürfen keine Anforderungen verloren gehen, es dürfen aber auch keine unnötigen Anforderungen hinzugefügt werden. Fehlende Anforderungen führen zur Unzufriedenheit des Kunden, unnötige Anforderungen verursachen unnötigen Aufwand.

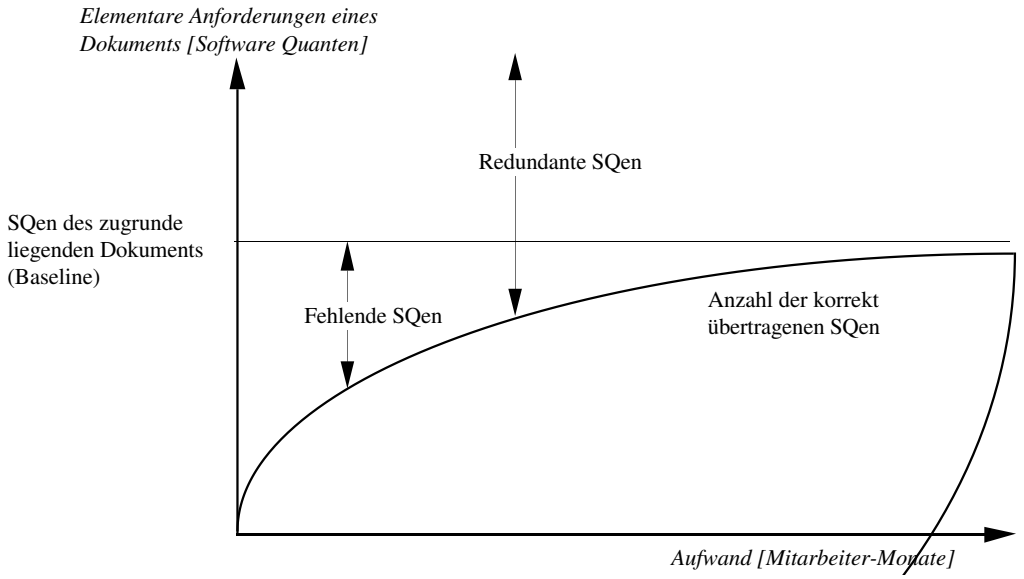


Abb. 1: Zusammenhang zwischen Aufwand und korrekt übertragenen, insgesamt erzielten, fehlenden und redundanten Software-Quanten

Alle Metriken, die wir dazu untersucht haben, messen nur den Umfang oder zählen die Fehler auf die eine oder andere Weise: COCOMO (Boehm 1981) schätzt die Anzahl der Instruktionen im späteren Produkt, Albrecht (1983) zählt die Function Points der Anforderungsanalyse. Die Zyklomatische Komplexität (McCabe 1976) zählt die Anzahl der Verzweigungen im Quellcode. Die Metriken, die im IEEE Standard 1045 (IEEE 1993) verzeichnet sind, zählen die Dokumenten-Seiten oder das Verhältnis von Seiten zu Graphiken in nicht-formalen Dokumenten. Die Fehler-Metriken des IEEE Standards 982.1 (IEEE 1988) zählen die Fehler klassifiziert nach ihrem

Diese Vorstellung ermöglicht uns allgemein die Qualität von Dokumenten, aber auch ihre quantitative Evolution zu modellieren:

- Ein Dokument besteht aus einer Menge von elementaren Anforderungen; diese Anforderungen werden durch sog. „Software-Quanten“ (SQen) dargestellt. Ein Software-Quant repräsentiert eine nicht mehr teilbare, elementare Information einer Software.
- Der Inhalt *jedes* Dokuments kann durch Software-Quanten modelliert werden. Dabei „manifestieren“ sie sich in jedem Dokument anders: in der Spezifikation beispielsweise in einem Satz, im

Entwurf durch eine Graphik, im Code schließlich durch einige Zeilen Programmcode.

- Wird ein Dokument geschrieben, so werden die Software-Quanten des zugrunde liegenden Dokuments in das neue Dokument übertragen. Wenn also ein Designer den Architekturentwurf schreibt, so muß er alle Anforderungen (alle Quanten) der Spezifikation in den Entwurf übertragen, nicht mehr und nicht weniger.
- Jede Übertragung führt zu Störungen: einige der Quanten der ursprünglichen Anforderungen gehen verloren, andere Quanten kommen unnötigerweise z.B. durch Mißverständnisse hinzu.
- Die Quanten sind unterscheidbar. Dadurch kann für jedes Dokument festgestellt werden, welche Quanten aus dem Vorgängerdokument übernommen worden sind und welche hinzukamen.
- Der Einfachheit halber haben wir als ersten Ansatz, die Anzahl der Quanten der Analyse (also das, was der Kunde ursprünglich wollte) mit der Anzahl der Function-Points gleichgesetzt (also 234 SQen für unser Projekt).

6.2 Das dynamische Simulationsmodell

Abbildung 1 zeigt den prinzipiellen Verlauf der Software-Quanten in einem Dokument bezüglich dem Aufand der für das Dokument erbracht wird.

Die Anzahl aller korrekt übertragbaren Software-Quanten wird natürlich durch den Inhalt des vorigen Dokuments bestimmt. Das vorige Dokument ist die Referenz für das aktuelle Dokument. Die Übertragungsfunktion hat die Form einer negativen Exponential-Funktion über den Aufwand, der für das Dokument eingesetzt wird. Zusammen mit den korrekten Quanten werden redundante eingeführt. Redundante Quanten sind als Differenz zwischen allen und den korrekten Quanten definiert. Die Gesamtzahl der Quanten wird mit Hilfe von COCOMO (1981) bestimmt. Die fehlenden Quanten sind definiert durch die Differenz zwischen allen Quanten des Vorgängerdokuments und den richtig übertragenen.

Vom Blickwinkel der nachfolgenden Dokumente gibt es keinen Unterschied mehr zwischen korrekten und redundanten Quanten: Was im Entwurf redundant war und was verlangt, ist für den Programmierer nicht mehr unterscheidbar – er implementiert alles, was entworfen ist;. Die Konsequenz diese Effekts ist durch die Badewannenkurve beschrieben, die in Abbildung 2 zu sehen ist: Fehler (die redundanten und fehlenden Quanten unseres Modells) können nur entdeckt werden, wenn ein Dokument gegen sein Vorgängerdokument geprüft wird – in späteren Phasen sind sie nicht mehr unterscheidbar. D. h. Fehler können nur in der Phase, in der sie gemacht wurden oder in der entsprechenden Testphase, aber in keiner der folgenden Entwicklungsphasen, gefunden werden!

Wir haben ebenfalls ein einfaches Modell von Schreib- und Codierfehlern eingeführt: Diese Fehler werden direkt aus der Anzahl der Software-Quanten abgeleitet. Im Gegensatz zu den fehlenden oder redundanten Quanten, haben diese Fehler für uns keine Seiteneffekte – sie repräsentieren einfache Syntax- oder nur Schreibfehler.

6.3 Ausführung der Simulation

Die Spieler haben keine Informationen über unser internes Modell erhalten, d.h. über die korrekten, fehlenden und redundanten Software-Quanten – sie erhielten nur die Anzahl der Seiten eines Dokuments oder die Anzahl der Codezeilen in einem Modul. Alle diese Größen wurden direkt aus der Gesamtzahl der Quanten abgeleitet.

Ohne Qualitätssicherung gehen mehr und mehr der ursprünglich verlangten Quanten verloren, während immer mehr redundante eingeführt werden. Unsere Übertragungsfunktion ist so parametrisiert, daß ein Projekt, das auf Basis von COCOMO geplant ist, in jeder Phase 80% der korrekten Quanten des Vor-

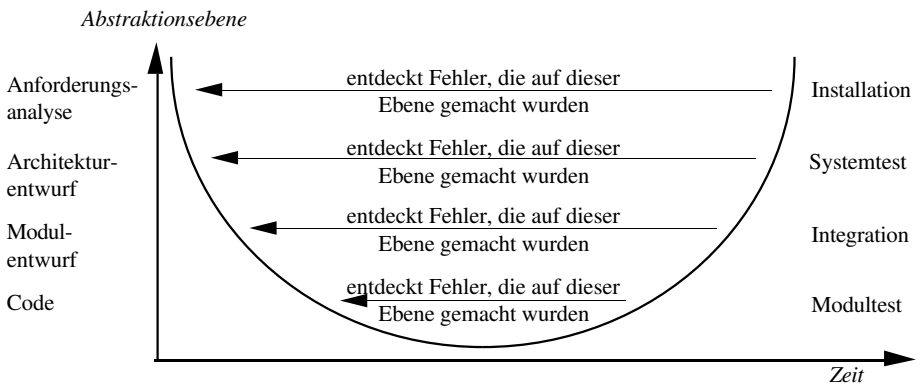


Abb. 2: Die Badewannenkurve

gängerdokuments in das nachfolgende überführt und zusätzlich noch eine Anzahl redundanter Quanten. (Wir unterstellen hier – willkürlich – eine Pareto-Verteilung.) In einem nominalen COCOMO-Projekt *ohne* Qualitätssicherung werden damit 80% der ursprünglichen Benutzeranforderungen in die Spezifikation übertragen, 64% (80% von 80%) werden in den Architekturentwurf übertragen. 51% (80% von 64%) gelangen in den Modulentwurf. Und schließlich gelangen nur 41% (80% von 51%) in den Programmcode. D. h. der Code enthält nur 41% der ursprünglich geforderten Eigenschaften. Trotz dieses dramatischen Schwunds, hat der Code die ursprünglich geforderte *Größe* , die unterwegs verlorenen Quanten wurden (für die Entwickler) unbemerkt durch redundante ersetzt.

Wie konnten die Spieler diesen Effekt verhindern? Zuallererst natürlich indem sie mehr Aufwand treiben ließen – dies führte dazu, daß sich die Anzahl der korrekten Quanten erhöhte, gleichzeitig erhöhte sich aber auch die Anzahl der redundanten Quanten und damit die Gesamtzahl der Quanten. Dies führt dazu, daß die Folgedokumente immer größer werden. Eine solche Politik würde in einem riesigen und dramatisch aufwendigen (und damit verspäteten) Produkt enden.

Der zweite Weg besteht darin, Qualitätssicherung zu betreiben. Im Fall unseres Modells vor allem durch Reviewing von Dokumenten. Ein gut vorbereitetes Review entdeckt 60% aller Fehler in einem Dokument, also 60% aller fehlenden und 60% aller redundanten Quanten. Die entdeckten redundanten Quanten können einfach und ohne Aufwand entfernt werden, die fehlenden Quanten müssen nachgearbeitet werden.

7. Ein Beispielzug

In diesem Kapitel zeige ich einen Beispielzug aus dem Fachpraktikum, um einen Eindruck vom Ablauf der Simulation zu vermitteln. Der Zug ist ein Ausschnitt aus dem siebten (von zwölf) Zügen der Gruppe 2 (der Gewinnergruppe).

7.1 Projektzustand zu Beginn des Zugs

Zu Beginn des siebten Zugs hatte das Projekt folgenden Zustand:

- Das Projekt begann am 1. Juni 1992. Das aktuelle Datum ist der 7. September 1992. Das Projekt hat ein Gesamtbudget von 250.000,- DM von denen bis jetzt 123 830,- DM ausgegeben sind.
- Der Projektleiter hat bisher drei Mitarbeiter eingestellt: Frau Siebenschläfer, Herrn Bankmüller und Frau Schmidt.
- Der Projektleiter hat bisher einige Informationen über Schulungen erhalten, darunter auch Unterlagen über MENTOR-5, ein Kurs über Software-Test-Methoden.
- Die Anforderungsanalyse hat einen Aufwand von 18 Mitarbeiter-Tagen benötigt und ist jetzt abgeschlossen. Die Spezifikation enthält 225 korrekte Software Quanten (von 234 möglichen) und 10 redundante Quanten. 68 der 225 Quanten wurden aufgrund ausführlicher Reviews erzielt.
- Im Augenblick werden gerade Architektur- und Modulentwurf durchgeführt. Der Architekturentwurf hat bisher 36 Mitarbeiter-Tage benötigt und ist inzwischen 143 Seiten dick. Er enthält 187 korrekte und 3 redundante Quanten. Der Modulentwurf hat bisher 20 Mitarbeiter-Tage gedauert, der bisher entstandene Entwurf umfaßt 112 Seiten und enthält 70 korrekte und 4 redundante Software-Quanten.
- Im letzten Zug wurde ein Review des Architekturentwurfs durchgeführt, das 31 mittlere Fehler (unser Ausdruck für fehlende oder redundante Quanten) und 37 einfache Fehler (unser Ausdruck für Schreibfehler) entdeckte. Aufgrund dieser Befunde wurde das Dokument mit der Auflage von Nacharbeiten akzeptiert.

Wie schon zuvor erläutert wurde nur ein Teil dieser Informationen den Spielern gegeben.

7.2 Der Zug

Der siebte Zug besteht aus den in Tabelle 2 gezeigten Aktionen.

Beigefügt war außerdem eine detaillierte Beschreibung der Arbeitspakete, die in diesem Zug ausgeführt werden sollten, sowie eine überarbeitete Fassung des Projektplans.

7.3 Reaktionen auf den Zug

Gruppe 2 erhielt folgende Antwort auf den Zug:

> „Das Management ist sehr besorgt über die Verzögerungen ihres Projekts. Sie werden gefragt, wie Sie unter diesen Umständen das Projekt zu Ende bringen werden und wie Sie den neuen Abgabetermin am 19. Dezember einhalten wollen. Die Aufstockung des Budgets wird vorläufig abgelehnt, eine endgültige Entscheidung wird nächsten Monat getroffen.
>
> Der Kunde ist nicht sehr davon angetan, daß Sie den Abnahmetermin auf den 19. Dezember verschieben wollen, akzeptiert aber unter der Bedingung, daß keine weiteren Verzögerungen mehr stattfinden.
>
> Herr Bankmüller korrigiert den Architekturentwurf, Frau Siebenschläfer und Frau Schmidt arbeiten am weiter am Modulentwurf, der inzwischen einen Umfang von 177 Seiten hat.

Nr.	Aktion	Betroffene	Beginn/Dauer
1	Wir geben dem Management einen Überblick über die allgemeine Projektsituation. Aufgrund des erwarteten weiteren Projektverlaufs bitten wir um eine Aufstockung des Budgets um 60.000,- DM.	Projektleiter Management	7. September/ 1 Tag
2	Wir verhandeln mit dem Kunden eine Verschiebung des Projektendes auf den 19. Dezember 1992 aus.	Projektleiter Kunde	8. September/ 1 Tag
3	Herr Bankmüller soll die im Review des Architekturentwurfs gefundenen Fehler korrigieren.	Herr Bankmüller	7. September/ 5 Tage
4	Frau Siebenschläfer und Frau Schmidt sollen die Arbeit am Modulentwurf fortsetzen.	Frau Siebenschläfer Frau Schmidt	7. September/ 5 Tage
5	Frau Siebenschläfer und Frau Schmidt sollen die Arbeit am Modulentwurf fortsetzen.	Frau Siebenschläfer Frau Schmidt	14. September/ 3 Tage
6	Wir gratulieren Frau Siebenschläfer zum Geburtstag, schenken ihr einen Blumenstrauß und laden alle zu einem kleinen Imbiß ein.	alle Mitarbeiter	14. September/ 30 Minuten
7	Wir schicken Herrn Bankmüller zur MENTOR-5-Schulung.	Herr Bankmüller	14. September/ 3 Tage
8	Frau Siebenschläfer, Frau Schmidt und Herr Bankmüller sollen sich auf das Review des Modulentwurfs vorbereiten.	Frau Siebenschläfer Frau Schmidt Herr Bankmüller	18. September/ 2 Tage
9	*weitere Aktionen*	…	…

Tab. 2: Ausschnitt aus dem Beispielzug

Frau Siebenschläfer ist sehr erfreut über den Blumenstrauß (60,- DM). Der Imbiß kostet sie 40,- DM.

Herr Bankmüller kehrt hoch motiviert und voller neuer Ideen von der MENTOR-5-Schulung zurück.

…“

Diese Resultate hatten die folgenden, internen Konsequenzen:

- Das Budget wird beim nächsten Zug auf 310.000,- DM erhöht werden.
- Das geplante Projektende wird auf den 19. Dezember 1992 verschoben.
- Die Motivation des Kunden wird um einen Punkt gesenkt.
- Der Architekturentwurf umfaßt nun 219 korrekte und 2 redundante Software-Quanten.
- Der Modulentwurf umfaßt nun 142 korrekte und 14 redundante Software-Quanten.
- Die Motivation von Frau Siebenschläfer steigt nach ihrem Geburtstag um einen Punkt.
- Die Testfähigkeiten von Herrn Bankmüller steigen nach der Schulung um eine Einheit. (Die Fähigkeiten wurden mit Hilfe der COCOMO-Einflußfaktoren in Boehm (1981) modelliert.)
- Die Motivation von Herrn Bankmüller steigt nach der Schulung um einen Punkt.
- …
- Gehälter (einschließlich Steuern und Versicherungskosten) für die vergangenen vier Wochen:
  - Projektleiter: DM 16.000,-
  - Frau Siebenschläfer: DM 12.000,-
  - Herr Bankmüller: DM 12.000,-
  - Frau Schmidt: DM 8.000,-
- Andere Kosten:
  - MENTOR-5 -Schulung: DM 1.470,-
  - Blumenstrauß und Imbiß: DM 100,-
- Das aktuelle Projektdatum ist der 5. Oktober 1992.
- Die bisher aufgelaufenen Projektkosten betragen 173.400,- DM.

8. Der Verlauf der Projekte

Das Fachpraktikum begann im Oktober 1992. Es nahmen 5 Gruppen Teil, pro Gruppe zwei Studenten. Die Studenten einer Gruppe spielten zusammen und verkörperten jeweils einen Projektleiter. Alle Gruppen waren erfolgreich (im Sinne des Fachpraktikums – nicht unbedingt im Sinne des simulierten Projekts!) und lieferten ein „vollständiges System“ innerhalb von zwölf Zügen ab. Keine Gruppe kam mit weniger als zwölf Zügen aus.

8.1 Gesamtergebnisse

Alle Gruppen arbeiteten gemäß dem Wasserfallmodell – sie begannen mit der Anforderungsanalyse und endeten bei der Integration und Abnahme – obwohl kein Modell vorgeschrieben war. Während des Projekts wurde von allen Gruppen ein Projektplan abgeliefert. Für die Projektplanung wurde von allen Gruppen mehr oder weniger COCOMO eingesetzt, eine Gruppe führte sogar eine Function Point-Analyse durch, konnte aber mit den ermittelten Function Points nichts weiter anfangen.

Gruppen, die eine frühzeitige Qualitätssicherung durchführten, gelangten zu besseren Ergebnissen, als Gruppen die nur sehr spät oder gar keine Qualitätssicherung hatten. Alle Gruppen haben sich während des Praktikums sehr stark mit ihren Projekten identifiziert, diese Identifikation ging soweit, daß einige Spieler in echte persönliche Probleme gestürzt wurden, als ihr Projekt ins Schlingern kam.

8.2 Bestimmung des Gewinners

Um den Gewinner zu bestimmen, haben wir die Gruppen nach den Kriterien in Tabelle 3 bewertet. Tabelle 3 enthält außerdem eine Gewichtungsfunktion, um Konflikte zwischen widerstreitenden Zielen zu lösen.

8.3 Ergebnisse der einzelnen Gruppen

Abb. 3 gibt eine Überblick über die Ergebnisse der fünf Gruppen.

Die obere Hälfte der Abbildung zeigt die erfüllten Kundenwünsche im fertigen Produkt (die korrekten Software-Quanten), wobei maximal 234 Quanten erreichbar waren. Diese Werte sollten so hoch wie möglich sein – das war das Hauptkriterium bei der Auswertung der Projektergebnisse. Die untere Hälfte zeigt die nicht gewünschten, aber trotzdem realisierten Eigenschaften (in redundanten Software-Quanten), die Budgetüberschreitung in 1000 DM und die Zeitüberschreitung gegenüber dem geplanten Abgabetermin in Tagen. Diese Werte sollten natürlich so niedrig wie möglich sein.

- Gruppe 2 erzielte die besten Ergebnisse. Sie zeigten keine besonders aufregenden Ergebnisse in den einzelnen Phasen, aber lieferten zu Beginn einen vollständigen 15-seitigen (!) Projektplan ab,

Kategorie	**sehr gut**	**gut**	**mittel**	**schlecht**	Gewicht
Projekt Ende	4. Dez. 1992 oder früher	12. Dez. 1992	26. Dez. 1992	8. Jan. 1993 oder später	5
Budget (DM)	280.000,- oder weniger	320.000,-	360.000,-	400.000,- oder mehr	5
fehlende SQen verglichen mit der ursprüngl. Analyse (die 234 SQen enthielt)	5 oder weniger	10	15	20 oder mehr	Anf. Analyse: 3 Architekturentw.: 3 Modulentw.: 3 Code: 5
redundante SQen verglichen mit der ursprüngl. Analyse	5 oder weniger	10	20	40 oder mehr	Anf. Analyse: 3 Architekturentw.: 3 Modulentw.: 3 Code: 5
Code-Fehler	3 oder weniger	6	10	15 oder mehr	5
Durchschnittl. Motivation der Projektmitarbeiter	2 oder mehr	1	0	-1 oder weniger	1
Motivation des Kunden	2 oder mehr	1	0	-1 oder weniger	1
Punkte	4 Punkte	3 Punkte	2 Punkte	1 Punkt	

Tab. 3: Gewichtungsfunktion zur Ermittlung der Simulationsergebnisse

der dauernd aktualisiert wurde. In den meisten Kriterien kamen sie nur auf den zweiten Platz – bis auf die erfüllten Kundenwünsche, hier waren sie am besten. Das Geheimnis ihres Erfolgs war eine vernünftige Planung, an die sie sich über das ganze Projekt hin gehalten haben. Diese Gruppe (und auch Gruppe 5) hat erwogen, eine Standard-Datenbank zu kaufen, um die geforderte Aufgabe zu erledigen. Beide Gruppen haben die Datenbank nicht gekauft, da sie zu teuer war.

- Diese Gruppe wurde von Gruppe 4 gefolgt. Sie hatten zwar bessere Ergebnisse bei Budget und Termin, aber weniger Kundenwünsche erfüllt. Die Gruppe beendete alle Aktivitäten zu früh, bevor sie überhaupt die Chance hatten, die geforderte Qualität zu erreichen – allerdings hatten sie dadurch auch nicht die Chance viele Fehler zu machen. Dies führte zu einem eher kleinen System ohne viele redundante Software-Quanten und geringen Zeit- und Budgetüberschreitungen.
- Gruppe 5 kam auf den 3. Platz. Sie entwickelten zwar fast das gleiche Produkt wie Gruppe 4 – im Sinne der Software-Quanten – aber sie benötigten wesentlich mehr Zeit und Geld. Ursache des schlechten Abschneidens (wie auch bei Gruppe 3) war der viel zu große Aufwand in der Entwicklung und der zu geringe Aufwand in der Prüfung der Ergebnisse. Dies führte dazu, daß diese Gruppe und Gruppe 3 riesige Produkte mit geringer Qualität und große Termin- und Budgetüberziehungen hatten. Interesanterweise haben sich beide Gruppen dazu entschlossen, in der Mitte der Entwicklung einen Teil ihrer zu großen Produkte zu streichen – welcher Teil gestrichen werden sollte, wurde allerdings nicht geplant, so daß der Streichung neben den redundanten auch eine Reihe korrekter Software-Quanten zum Opfer fielen. Beide Gruppen hatten offensichtliche Schwierigkeiten, ein Projekt zu planen und sich an die Planung zu halten.
- Auf den 4. Platz kam Gruppe 1. Diese Gruppe hat ein System entwickelt, ohne sich dabei allzu sehr um Qualitätssicherung zu kümmern. Statt der geforderten Eigenschaften hatte das Produkt eine große Anzahl redundanter Quanten. Dies war die einzige Gruppe die beinahe innerhalb der vorgegebenen Zeit und des vorgegebenen Budgets war – sie hörten einfach auf als Zeit und Geld zu Ende waren
- Gruppe 3 kam auf den letzten Platz. Diese Gruppe konnte nicht vernünftig planen und machte die gleichen Fehler wie Gruppe 5, kam dabei aber noch mehr ins Schlingern. Man kann ihre Anstrengungen an den größten Budgetüberschreitung ablesen, die den größten Aufwand anzeigt – diese Anstrengungen führten aber zu einem Produkt, das noch schlechter als das der Gruppe 1 war.

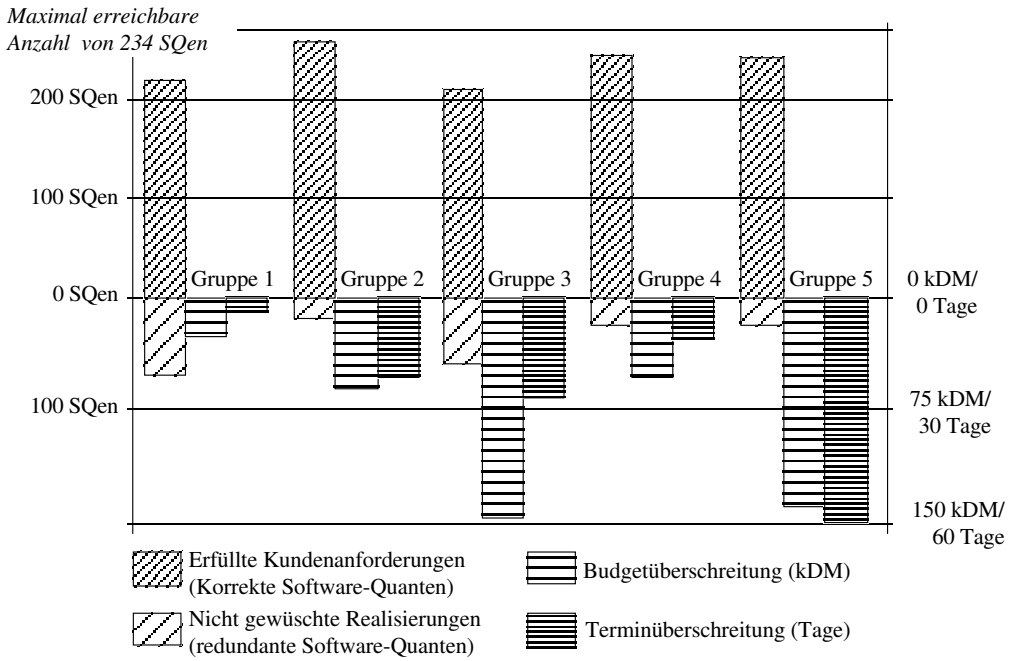


Abb. 3: Die Resultate der Projekte

Tabelle 4 zeigt die Einzelergebnisse der Projekte.

9. Erfahrungen mit der Simulation von Software-Projekten

Am Ende des Fachpraktikums wurden zwei Besprechungen mit allen Spielern durchgeführt, in der die Ergebnisse diskutiert und das Modell vorgestellt wurde. Dabei wurden zahlreiche Probleme identifiziert und das Modell kritisiert. Insgesamt schätzen wir den Verlauf des Fachpraktikums aber erfolgreich ein:

- Wir konnten ein plausibles Projektverhalten simulieren, das sich den Aktionen unserer fünf „Projektleiter“ anpaßte.
- Wir konnten ein relativ einfaches Simulationsmodell einsetzen, das über die ganze Simulation hinweg unverändert blieb.
- Die Studenten berichteten über Erfahrungen, wie man sie in echten Projekten bei mangelhafter Planung oder unzureichender Qualitätssicherung erleben kann.
- Die post-mortem Analyse war ergiebiger als in wirklichen Projekten und führte zu einem besseren Verständnis der Vorgänge in den simulierten Projekten.
- Der Aufwand für die Simulation war hoch, konnte aber dank des einfachen Modells bewältigt werden.
- Die Simulation beeinflußte das Verhalten der Studenten nicht dramatisch: keiner hatte das Gefühl, daß unrealistische Effekte auftraten.

Gruppe	**1**	**2**	**3**	**4**	**5**
Projektende (alle Projekte begannen am 1. Juni 1992)	4. Dez. 1992	21. Dez. 1992	27. Dez. 1992	11. Dez. 1992	29. Jan. 1993
Budget (DM)	276.660,-	310.596,-	398.616,-	303.615,-	391.508,-
Korrekte Software-Quanten (kSQen), fehlende Software-Quanten (fSQen) und redundante Software-Quanten (rSQen) in der *Anforderungsanlyse* verglichen mit den ursprüngl. Anforderungen	215 kSQen 29 fSQen 15 rSQen	225 kSQen 9 fSQen 10 rSQen	228 kSQen 6 fSQen 50 rSQen	214 kSQen 20 fSQen 23 rSQen	213 kSQen 21 fSQen 8 rSQen
Korrekte Software-Quanten, fehlende Software-Quanten und redundante Software-Quanten im *Architektiurentwurf* verglichen mit den ursprüngl. Anforderungen	198 kSQen 36 fSQen 51 rSQen	210 kSQen 24 fSQen 11 rSQen	217 kSQen 17 fSQen 70 rSQen	214 kSQen 20 fSQen 34 rSQen	202 kSQen 32 fSQen 24 rSQen
Korrekte Software-Quanten, fehlende Software-Quanten und redundante Software-Quanten im *Modulentwurf* verglichen mit den ursprüngl. Anforderungen	166 kSQen 68 fSQen 48 rSQen	200 kSQen 34 fSQen 15 rSQen	164 kSQen 70 fSQen 63 rSQen	198 kSQen 36 fSQen 39 rSQen	202 kSQen 32 fSQen 24 rSQen
Korrekte Software-Quanten, fehlende Software-Quanten und redundante Software-Quanten im *fertigen Produkt* verglichen mit den ursprüngl. Anforderungen	186 kSQen 48 fSQen 66 rSQen	215 kSQen 19 fSQen 19 rSQen	180 kSQen 54 fSQen 58 rSQen	212 kSQen 22 fSQen 24 rSQen	212 kSQen 22 fSQen 25 rSQen
Code-Fehler	44 Fehler	0 Fehler	3 Fehler	10 Fehler	1 Fehler
Durchschnittliche Motivation der Projektmitarbeiter	0,75	3,3	0,2	0,75	2,8
Motivation des Kunden	-3	-1	0	-2	-1
Wertung	**48**	**95**	**48**	**66**	**54**

Tab. 4: Die Resultate der Projekte

Ein wichtiger Erfolgsfaktor für die Simulation war die lebendige Präsentation. Sobald sich die Studenten mit dem Projekt identifizieren konnten, haben sie sich auch wie Projektleiter verhalten.

Die meisten Spieler kritisierten, daß sie keine echten Dukumente und keine informativen Antworten von ihren simulierten Mitarbeitern erhielten:

- „Wenn ich das Dokument gesehen hätte, würde ich wissen, ob es die Anforderungen des Kunden erfüllt." Der Blindflug war eine negative Erfahrung – aber eine Sache mit der sie auch in einem echten Projekt umgehen müssen.
- „In Wirklichkeit sind meine Mitarbeiter aktiver, sie erzählen mehr über den Projektfortschritt." Nein, das tun sie nicht! Aber die Aussage zeigt, wie wichtig eine lebendige Simulation ist. Die nackten Zahlen in natürliche Sprache zu packen, ist ein guter Ansatz dazu, der aber noch weiter entwickelt werden muß.
- „Es muß Unterstrukturen für die Dokumente geben. Man kann keine ganze Spezifikation in einem einzigen Review prüfen." Das ist richtig und wird in dem nächsten Modell korrigiert werden.

Die folgenden Bestandteile haben sich bewährt und können beibehalten werden:

- Das einfache Modell war prinzipiell ausreichend. Lediglich die manuelle Simulation war zu mühsam. Mit der Fertigstellung des SESAM-Simulators sollte dieses Problem erledigt sein.
- Die Granularität der Züge auf Tagesebene war ausreichend, sie bewahrte die Spieler davor, sich in Details zu verlieren. Vorstellbar ist aber auch eine feinere Granularität.
- Die von uns für die Simulation bereitgestellten Aktionen, Programmierer, Schulungen und Werkzeuge waren ausreichend.

Zusammenfassend kann sagen, daß für den Einsatz im Unterricht bereits dieses einfache Modell ausreicht, um die Brücke zwischen Software Engineering-Ausbildung und Praxis zu schlagen: Es gibt Studenten die Gelegenheit, Erfahrungen als Projektleiter zu sammeln!

Literatur

Abdel-Hamid, T. K. (1991): **Software Project Dynamics - An Integrated Approach.** Prentice Hall, Englewood Cliffs, New Jersey.

Albrecht, A. J., J. E. Gaffney (1983): Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. **IEEE Transactions on Software Engineering, SE-9,** Nov. 83, pp. 639-648.

Boehm, B. W. (1981): **Software Engineering Economics**. Prentice Hall, Englewood Cliffs, New Jersey.

Deininger, M., K. Schneider (1994): Teaching Software Project Management by Simulation. **Proceedings of the 7th Conference on Software Engineering and Education (CSEE)**, San Antonio, Januar 1994, pp. 227-242.

Fairley, R. (1985): **Software Engineering Concepts.** McGraw-Hill, New York.

Frühauf, K., J. Ludewig, H. Sandmayr (1988): **Software-Projektmanagement und -Qualitätssicherung.** Verlag der Fachvereine an der Schweizerischen Hochschulen und Techniken, Zürich.

IEEE (1989): **Standard Dictionary of Measures to Produce Reliable Software**. IEEE Std 982.1-1988.

IEEE (1993): **Standard for Software Productivity Metrics**. IEEE Std. 1045.

Knöll, H.-D., J. Busse (1991): **Aufwandsschätzung von Software-Projekten in der Praxis: Methoden, Werkzeugeinsatz, Fallbeispiele**. (Reihe Angewandte Informatik Bd. 8), BI-Wissenschaftsverlag, Mannheim, Wien, Zürich.

Ludewig, J., Th. Bassler, M. Deininger, K. Schneider, J. Schwille (1992): SESAM - Simulating Software Projects. **Proceedings of the Software Engineering and Knowledge Engineering Conference (SEKE '92)**, Capri, Mai 1992, pp. 608-615.

McCabe, T. J. (1976): A Complexity Measure. **IEEE Transactions on Software Engineering, SE-2**, pp. 308-320.

McKeeman, W. M. (1989): Graduation Talk at Wang Institute. **IEEE Computer, Vol. 22**, No. 5, pp. 78-80.

Schneider, K. (1993): Object-Oriented Simulation of the Software Development Process in SESAM. **Proceedings of the Object-Oriented Simulation Conference (OOS '93)**, Teil der Western Simulation Multiconference, San Diego, Januar 1993.

Schneider, K. (1993a): **SESAM-Zwischen Planspiel und Adventure Game**. „Informatik und Schule '93", Koblenz, Oktober 1993.

Schneider, K., M. Deininger (1994): An Overview of the SESAM Project. Erscheint in den **Proceedings of the GMD-Metrics Workshop**.

Sommerville, I. (1989): **Software Engineering**. 3rd Edition, Addison Wessley, Workingham, England.

Vester, F. (1987): **Ökolopoly - Ein kybernetisches Umweltspiel.** Otto Maier Verlag, Ravensburg, Germany.

Teil 7

SESAM und vis-A-vis

Jürgen Schwille

Zusammenfassung

SESAM-Modelle werden mittels verschiedener graphischer Notationen beschrieben. Für jede graphische Notation stellt SESAM einen graphischen Editor bereit. Dieser Artikel beschreibt das Werkzeug vis-A-vis, welches hinter diesen Editoren steht, seinen Einsatz in SESAM und das Konzept der Integritätsbedingungen, das eine mögliche Weiterentwicklung von vis-A-vis zur einfacheren Realisierung graphischer Editoren darstellt.

1. Einführung

SESAM enthält verschiedene graphische Notationen zur Beschreibung von Modellen, z. B. eine Notation zur Beschreibung der Elemente eines Software-Projekts und deren Beziehungen (Abbildung 1), zur Beschreibung von konkreten Projektsituationen (Abbildung 2) und zur Beschreibung von Regeln, die angegeben, wie aus einer Projektsituation eine neue Situation hervorgeht (Abbildung 3).

Schema-, Situations- und Regeleditor werden von einem Modellbauer verwendet, um sein Modell einer Software-Entwicklung zu beschreiben. Der Modellbauer legt hierfür sein Projektschema fest,

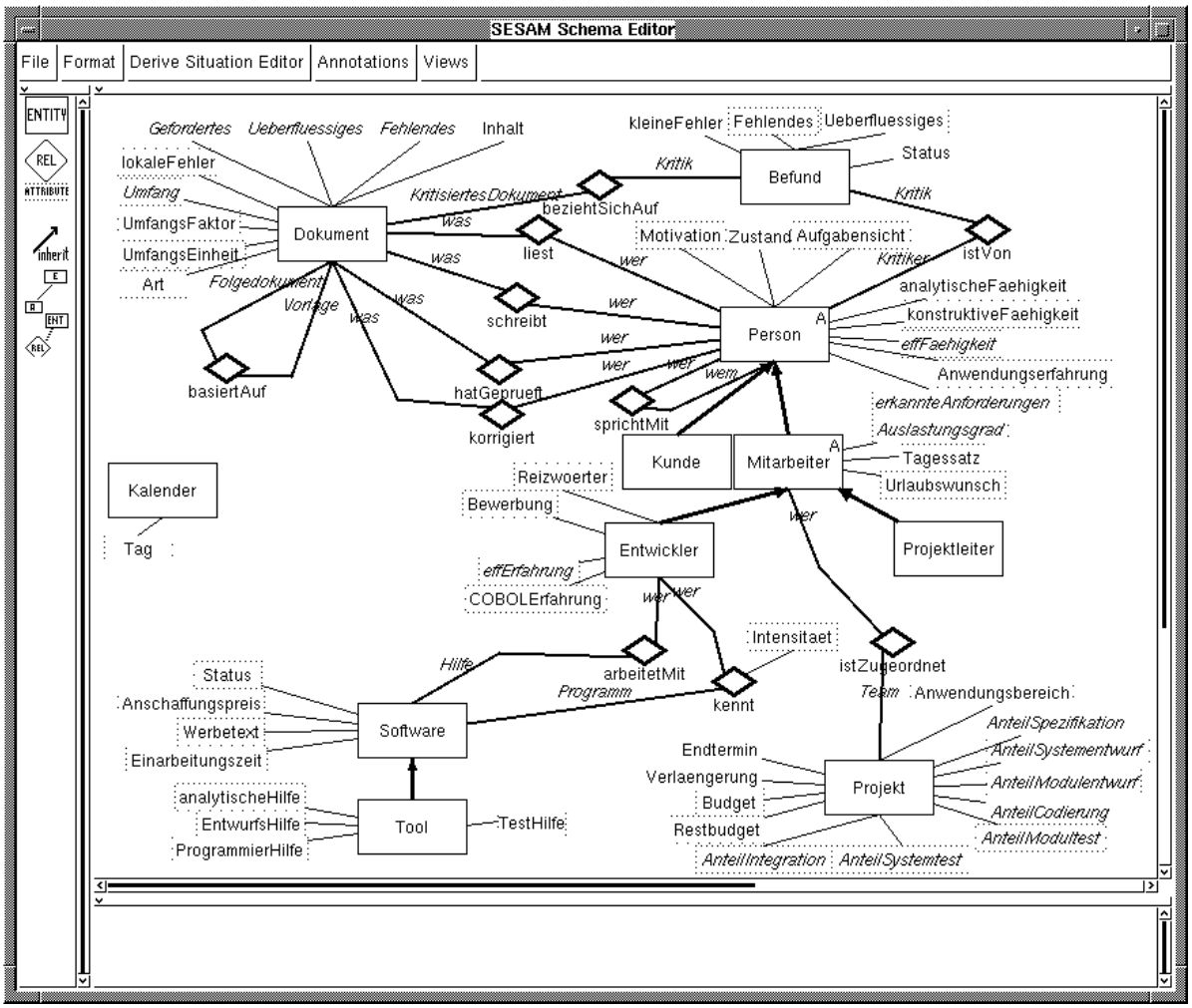


Abb. 1: Der Schemaeditor

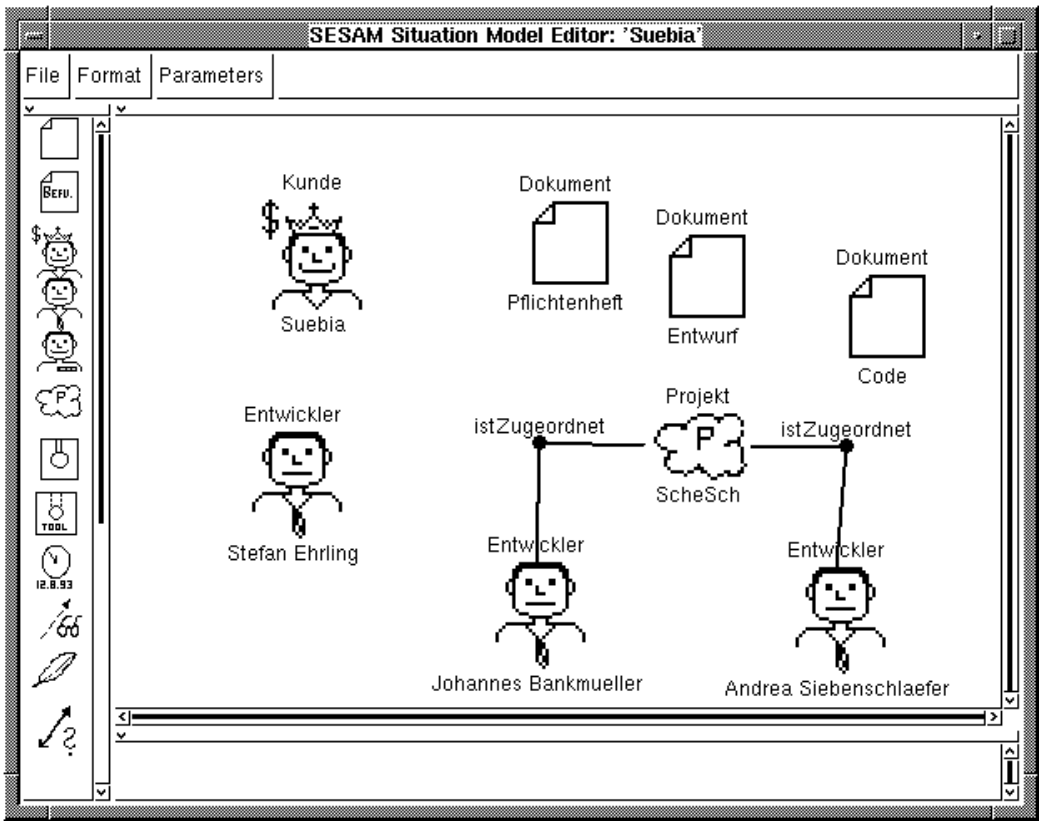


Abb. 2: Der Situationseditor

beschreibt eine Anfangssituation des zu simulierenden Projekts und gibt die Regeln an, nach denen sich sein Projektmodell verhalten soll. Das in diesem Band verwendete SESAM-Beispielmodell umfaßt insgesamt 30 Regeln, um ein Beispiel für die notwendige Zahl von Regeln zu geben.

Der Modellbauer kann, z. B. aufgrund von Erkenntnissen, die er aus der Simulation seines Projektmodells gewonnen hat, sein Modell verändern und anschließend neu simulieren. Die graphische Darstellung seiner Modelle erlauben einem Modellbauer, notwendige Modelländerungen schnell durchzuführen und seine Modelle übersichtlich und kompakt zu halten. SESAM bietet komfortable Möglichkeiten zur Bearbeitung von Modellen. Modelle müssen nicht umständlich textuell beschrieben werden, sondern können graphisch erstellt und geändert werden, was die Arbeit des Modellbauers wesentlich erleichtert.

Die hier gezeigten graphischen Editoren haben sehr viele Gemeinsamkeiten: In jedem Editor können graphische Elemente eingefügt, gelöscht und positioniert werden, jede Notation legt bestimmte graphische Repräsentationen ihrer Elemente fest, jede Notation definiert mögliche Verbindungen zwischen den Elementen.

Diese Vielzahl von Gemeinsamkeiten bildete die Motivation für die Erstellung von vis-A-vis. vis-A-vis wird von seinen Entwicklern als „Application Framework“ bezeichnet, das Bausteine zur Erstellung von graphischen Editoren enthält. Alle gezeigten Editoren bauen auf vis-A-vis auf, d. h. verwenden die von vis-A-vis bereitgestellten Bausteine, was sich z. B. in einer einheitlichen Benutzeroberfläche widerspiegelt (vgl. Abbildungen 1 bis 3).

Dieser Artikel soll einerseits die Verbindung von vis-A-vis und SESAM aufzeigen, andererseits einen kurzen Überblick über die Möglichkeiten von vis-A-vis geben (Abschnitt 2). Abschnitt 3 geht über vis-A-vis hinaus und zeigt, wie die graphischen Notationen inhärenten Integritätsbedingungen mittels Prädikatenlogik formuliert werden können.

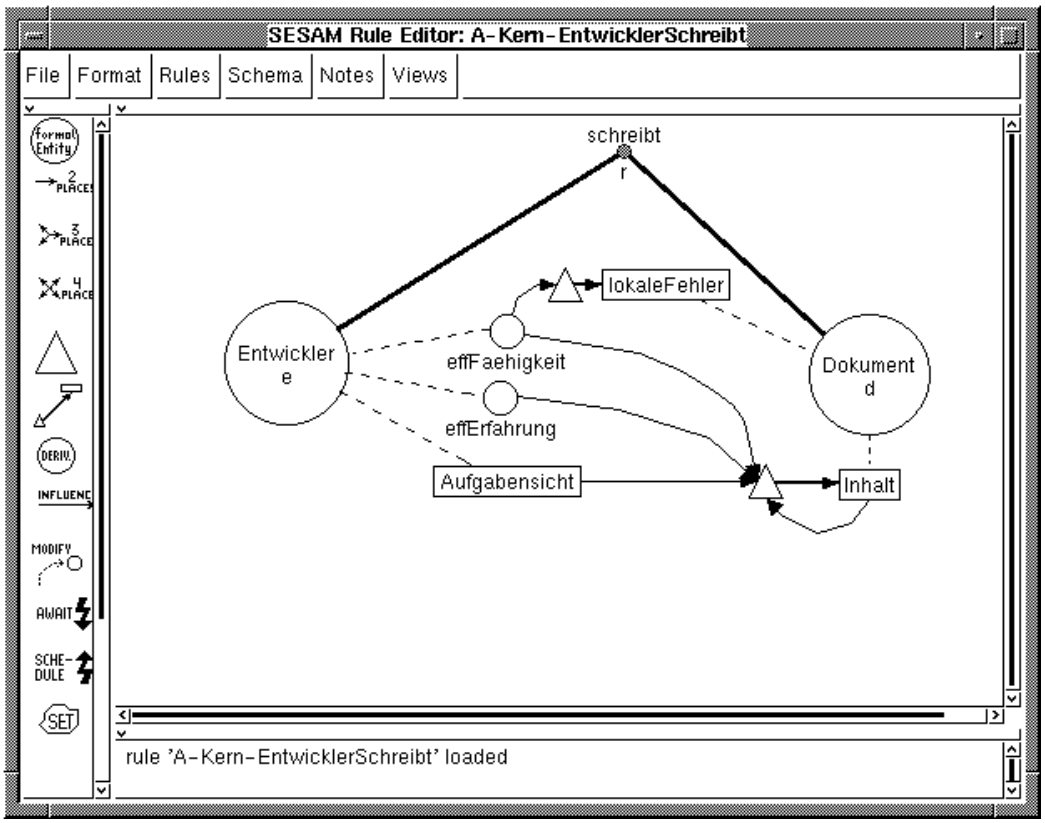


Abb. 3: Der Regeleditor

2. vis-A-vis

Dieser Abschnitt beschreibt die wesentlichen Merkmale von vis-A-vis. Lichter (1993) und Lichter (1993a) beschreiben vis-A-vis im Detail.

2.1 Anwendungsbereich von vis-A-vis

Graphische Notationen werden nicht nur in SESAM verwendet. Überall im Software Engineering finden sich graphische Notationen, z. B. in SA von DeMarco, in JSD von Jackson und Cameron sowie in OMT von Rumbaugh, um nur einige Notationen zu nennen. Auch außerhalb des Software Engineering findet man viele Beispiele, wie etwa Petrinetze, endliche Automaten oder Flußdiagramme. Der Grund für diese Vielzahl graphischer Notationen ist einfach: Graphische Notationen erlauben es, komplexe Zusammenhänge übersichtlich darzustellen.

Einen graphischen Editor für eine graphische Notation „from scratch“ zu erstellen ist eine schwierige und aufwendige Aufgabe. vis-A-vis erleichtert diese Aufgabe wesentlich durch seine Klassenbibliothek, in der der „Werkzeugbauer“ viele fertige Bausteine für seinen Editor vorfindet. Anders als z. B. in Gandalf (Habermann, 1986) geht es bei vis-A-vis nicht darum, aus der textuellen Beschreibung einer Notation in Form einer Grammatik einen fertigen Editor zu generieren, sondern vis-A-vis unterstützt einen Werkzeugbauer durch eine umfangreiche Bibliothek von fertigen Editorbausteinen.

2.2 Leistungen von vis-A-vis

vis-A-vis bietet einem Benutzer eine einheitliche Benutzeroberfläche, eine umfangreiche Klassenbibliothek und eine Standardarchitektur. Diese drei Leistungen werden im folgenden kurz erläutert.

Abbildungen 1 bis 3 zeigen die Benutzeroberfläche, die allen vis-A-vis-Editoren zugrunde liegt. Ein vis-A-vis-Fenster besteht aus einer *Menüleiste*, in der einerseits Standardoperationen wie Speichern und Laden eines Diagramms, andererseits auch editorspezifische Kommandos enthalten sind, einer *Palette*, die die Symbole der graphischen Notation zeigt, einem *Zeichenfenster* für die Erstellung eines Diagramms aus den verfügbaren

Symbolen und einem *Textfenster* für Meldungen an den Benutzer.

Die vis-A-vis-Klassenbibliothek stellt eine Reihe von Grundvisualisierungsformen zur Verfügung, aus denen der Werkzeugbauer seine eigenen graphischen Symbole zusammensetzen kann. *vis-A-vis-Konnektoren* verbinden die semantischen Objekte einer Notation, bei einem Petrinetz-Editor sind dies Stellen und Transitionen sowie ihre zugehörigen Operationen wie z. B. das Setzen einer Marke auf eine Stelle, mit ihrer graphischen Darstellung.

vis-A-vis gibt Werkzeugbauern eine Standardarchitektur vor. Durch Spezialisierung der vorhandenen vis-A-vis-Klassen erstellt ein Werkzeugbauer einen neuen Editor, d. h. die Architektur des neuen Editors ist der vis-A-vis-Architektur untergeordnet.

2.3 Verwendung von vis-A-vis

Lichter (1993) schlägt folgende Vorgehensweise vor, um einen neuen vis-A-vis-Editor zu erstellen:

1. **Semantische Objekte der Anwendung identifizieren und implementieren**

Der Werkzeugbauer muß die Bestandteile seiner graphischen Notation, die semantischen Objekte, identifizieren und ihr Verhalten realisieren. Dies geschieht völlig unabhängig von vis-A-vis.

2. **Grundsymbole für die semantischen Objekte festlegen**

Für die graphische Darstellung der Symbole einer Notation stellt vis-A-vis eine umfangreiche Bibliothek zur Verfügung, aus der die passende Darstellung für ein Symbol ausgewählt wird oder anhand der vorhandenen Grundvisualisierungsformen zusammengesetzt wird.

3. **Festlegen, welche Aspekte des semantischen Objekts wie visualisiert werden sollen**

In diesem Schritt wird die in Schritt 1 erstellte Realisierung eines semantischen Objekts mit der in Schritt 2 erstellten graphischen Darstellung des Objekts verknüpft. Hierfür werden die vorhandenen vis-A-vis-Konnektoren verwendet.

4. **Unterklasse der vis-A-vis-Werkzeugklasse erstellen**

Die in den Abbildungen 1 bis 3 gezeigte Benutzeroberfläche kann um editorspezifische Menüs erweitert werden, indem das vorhandene vis-A-vis-"Basiswerkzeug" spezialisiert wird (vgl. die unterschiedlichen Menüs der einzelnen Editoren).

2.4 Stand von vis-A-vis

Die Vielzahl der mit vis-A-vis erstellten Editoren innerhalb und außerhalb von SESAM zeigt, daß vis-A-vis ein vielseitig einsetzbares Werkzeug ist, das die Erstellung graphischer Editoren wesentlich erleichtert.

Geplante Erweiterungen von vis-A-vis sind einerseits die Speicherung der mit vis-A-vis erstellten Modelle in der objektorientierten Datenbank GemStone und andererseits die Entwicklung von Meta-Werkzeugen zur Verkürzung der reinen Programmierarbeit bei der Erstellung eines vis-A-vis-Editors.

3. Integritätsbedingungen

Modelle, die mittels graphischer Notationen beschrieben werden, müssen bestimmten Bedingungen genügen, damit diese Modelle bzgl. der verwendeten Notation korrekt sind. Diese Bedingungen werden hier als *Integritätsbedingungen* bezeichnet.

Integritätsbedingungen beschreiben, welche Elemente einer graphischen Notation wie kombiniert werden dürfen. In Abbildung 1 ist jedes Attribut, dargestellt durch ein gepunktetes Rechteck, mit genau einem Entitätstyp, dargestellt durch ein Rechteck mit durchgezogenen Linien, verbunden. Ein Attribut, daß keinem oder mehr als einem Entitätstyp zugeordnet ist, verletzt diese Bedingung und macht das zugehörige Diagramm ungültig.

Zwei Arten von Integritätsbedingungen lassen sich unterscheiden: Integritätsbedingungen können entweder *notationsbezogen* oder *benutzerdefiniert* sein. Die Integritätsbedingung des vorangegangenen Absatzes ist notationsbezogen. Ein Beispiel für eine benutzerdefinierte Integritätsbedingung ist, daß ein bestimmter Mitarbeiter nur maximal einem Projekt zugeordnet sein darf (vgl. Relation istZugeordnet in Abbildung 1 und 2). Benutzerdefinierte Integritätsbedingungen werden nicht durch die Notation impliziert, sondern durch den Modellbauer festgelegt.

Beide Arten von Integritätsbedingungen treten in graphischen Notationen in unerwartet großer Zahl auf. Für eine vom Autor spezifizierte Entity-Relationship-Notation, die der in Abbildung 1 gezeigten Notation sehr ähnlich ist, wurden rund 70 verschiedene notationsbezogene Integritätsbedingungen gefunden. Für ein in dieser Notation beschriebenes Schema wurden bisher rund 30 benutzerdefinierte Integritätsbedingungen aufgestellt, wobei diese Zahl noch stark steigen wird, da die Entwicklung des Schemas noch lange nicht abgeschlossen ist.

vis-A-vis unterstützt die Formulierung von Integritätsbedingungen bisher nur rudimentär. Integritätsbedingungen müssen „festverdrahtet" reali-

siert werden, wobei vis-A-vis hierfür einige mengenorientierte Anfragen wie das Durchlaufen aller Elemente eines Diagramms zur Verfügung stellt. Was der Modellbauer sich dagegen wünscht ist eine Notation, in der er die Integritätsbedingungen seiner Modelle nur spezifiziert, die Überprüfung der Bedingungen jedoch nicht realisieren muß.

Eine Möglichkeit zur Formulierung von Integritätsbedingungen bietet die Prädikatenlogik (vgl. Westfechtel, 1991, und Wiebe, 1990), mit der erste positive Erfahrungen gemacht wurden. Von den oben zitierten 70 notationsbezogenen Integritätsbedingungen konnten fast 60 Bedingungen mittels Prädikatenlogik erster Ordnung nach dem Ansatz von Wiebe formuliert werden, d. h. nur noch ein kleiner Teil von Integritätsbedingungen muß hartverdrahtet in den Editor eingebaut werden, was den Aufwand zur Erstellung eines Editors beträchtlich reduziert und den Editor selbst wesentlich übersichtlicher macht, da der Modellbauer gezwungen ist, Code zur „normalen“ Diagrammbearbeitung von Code zur „Fehlerbearbeitung“, sprich Code für die Integritätsbedingungen, strikt zu trennen.

Ohne hier auf die Details prädikatenlogikbasierter Integritätsbedingungen einzugehen wird jetzt noch abschließend auf die Einbettung von Integritätsbedingungen in eine graphische Notation eingegangen. Benutzerdefinierte Integritätsbedingungen werden an das jeweilige Element eines Diagramms angehängt. Die oben beschriebene Bedingung, daß ein Mitarbeiter nur maximal einem Projekt zugeordnet sein darf, wird in Abbildung 1 am bestem beim Entitätstyp Mitarbeiter definiert. Diese Bedingung wird aus Gründen der Übersichtlichkeit nicht im Diagramm sichtbar sein. Einen Mitarbeiter im Situationseditor mehr als einem Projekt zuzuordnen, wird durch die obige Bedingung verhindert. (Diese Bedingung könnte mit den in vielen Entity-Relationship-Notationen enthaltenen Kardinalitäten formuliert werden. Für andere benutzerdefinierte Integritätsbedingungen reichen Kardinalitäten jedoch nicht aus.) Notationsbezogene Integritätsbedingungen können nicht mehr einem einzelnen Element zugeordnet werden, sondern gelten für mehrere Elemente eines Diagramms: Daß ein Attribut stets genau einem Entitätstyp zugeordnet ist, muß für alle Attribute gelten.

Die hier vorgestellten Überlegungen bzgl. Integritätsbedingungen werden derzeit noch vom Autor untersucht und dürfen deshalb keinesfalls als abgeschlossen angesehen werden. Dieser Artikel sollte nur die Problematik der Integritätsbedingungen klären und eine mögliche Lösung aufzeigen. Ob dieser Weg auch wirklich zum Erfolg führt, muß erst noch untersucht werden.

Literatur

Habermann, A.N., D. Notkin (1986): Gandalf. Software Development Environments. **IEEE Transactions on Software Engineering, 12(12)**, Dezember 1986. S. 1117-1127.

Lichter, H., K. Schneider (1993): vis-A-vis: Ein objektorientiertes Application Framework für graphische Entwurfswerkzeuge. In **H.C. Mayr und R. Wagner (Hrsg.): Objektorientierte Methoden für Informationssysteme**. Springer, Informatik aktuell.

Lichter, H., K. Schneider (1993a): vis-A-vis: An Object-Oriented Application Framework for Graphical Design Tools. **Proc. of the IFIP Workshop on Interfaces in Industrial Systems for Production and Engineering**. Darmstadt, 15.-17.03.93. Elsevier

Westfechtel, B. (1991): **Revisions- und Konsistenzkontrolle in einer integrierten Software-Entwicklungsumgebung**. Informatik-Fachbericht, Nr. 280. Springer-Verlag, Berlin.

Wiebe, D. (1990): **Generic Software Configuration Management: Theory and Design**. University of Washington, Dept. of Computer Science, PhD Thesis, TR 90-07-03.

Teil 8

Software Engineering objektorientiert — eine Herausforderung für die Praxis

Horst Lichter, Schweizerische Bankgesellschaft Zürich

Zusammenfassung

In diesem Beitrag formuliere ich am Anfang einige wesentliche Gründe, warum die objektorientierte Technologie (OO-Technologie) in der industriellen Software-Entwicklung eingesetzt wird oder werden wird. Anschließend werden Randbedingungen und Einflußfaktoren vorgestellt und diskutiert, die berücksichtigt werden müssen, wenn die OO-Technologie in einem Unternehmen eingeführt werden soll. Darauf aufbauend wird am Beispiel der Schweizerischen Bankgesellschaft (SBG) erläutert, wie dies organisatorisch durchgeführt werden kann.

1. Warum Objektorientierung? – Einige Gründe

Es gibt verschiedene Gründe, warum die OO-Technologie zur Zeit in vielen industriellen Software-Unternehmen eingeführt oder zum Teil bereits eingesetzt wird. Es sollen nur einige genannt werden:

- Der Aspekt „time to market“ wird auch oder ist gerade auch bei Software-Produkten immer wichtiger. Je schneller und kostengünstiger neue Produkte erstellt oder existierende den Marktanforderungen angepaßt werden können, je besser kann man sich im Markt behaupten.

  Man hofft, mit den Mitteln der OO-Technologie – insbesondere durch die Wiederverwendung von Bausteinen – kürzere Entwicklungszeiten zu erzielen.
- In den Entwicklungsabteilungen großer Firmen werden mehr und mehr Ressourcen verbraucht, um die bestehenden operativen Systeme zu warten und zu pflegen. Schuld daran sind unter anderem eine extrem hohe Integration der einzelnen Anwendungen; eine saubere Schnittstellenarchitektur zwischen den Anwendungen fehlt. Änderungen können dementsprechend nicht lokal begrenzt ausgeführt werden, sondern schlagen in das gesamte Netz der Anwendungen durch.

  Hier hofft man, mit den Mitteln der Datenkapselung Bausteine mit hohem inneren Zusammenhalt konstruieren zu können (Datenstrukturen und deren Operationen), die über exakt definierte Schnittstellen miteinander verbunden sind.
- Ein weiteres Manko der konventionellen Software-Entwicklung, das sich sowohl in den Entwicklungszeiten als auch im Wartungsaufwand niederschlägt, besteht darin, daß nur sehr selten Bausteine wiederverwendet werden, wenn neue „ähnliche“ Anwendungen erstellt werden müssen.

  Hier hofft man, spezielle Klassenbibliotheken oder sogar Frameworks erstellen und einsetzen zu können, die einen großen Teil der immer wieder benötigten Funktionalität einer Anwendungsklasse zur Verfügung stellen.

Die Gründe, warum die OO-Technologie in der industriellen Software-Entwicklung eingesetzt werden soll, können zusammenfassend auf den folgenden Nenner gebracht werden: Software soll *schneller*, *billiger* und *qualitativ* hochwertiger erstellt werden.

Die Ansprüche, denen die OO-Technologie gerecht werden muß, sind demnach sehr hoch. Die in sie gesetzten Hoffnungen ruhen vor allem auf den Techniken Datenkapselung mit sauberer Schnittstellenarchitektur und Vererbung als Mittel zur Wiederverwendung. Daß die Hoffnungen, die hinter den genannten Gründen stehen, nicht unberechtigt sind, zeigen Beispiele industrieller Software-Entwicklungen (siehe z.B. Bürkle, 1992)

2. Einführung der OO-Technologie

In Kilberth et al. (1993) wird die OO-Technologie und ihre Einführung unter verschiedenen Gesichtspunkten – technisch, organisatorisch und wirtschaftlich – beschrieben. Ich möchte besonders auf die folgenden Aspekte eingehen: Welche Randbedingungen sind bei der Einführung der OO-Technologie zu beachten und welche Konsequenzen müssen daraus gezogen werden. Die dazu gemachten Aussagen gelten zum Teil nicht nur speziell für die Einführung der OO-Technologie, sondern gelten generell, wenn eine neue Technologie eingeführt werden soll.

Die Einführung der OO-Technologie stellt in verschiedener Hinsicht ein Risiko dar: Auf der einen Seite birgt die Technologie selbst erhebliche Risiken in sich, auf der anderen Seite kann der Schaden, der dadurch entstehen kann, daß die Technologie ungeplant und unsystematisch eingeführt wurde, ebenfalls

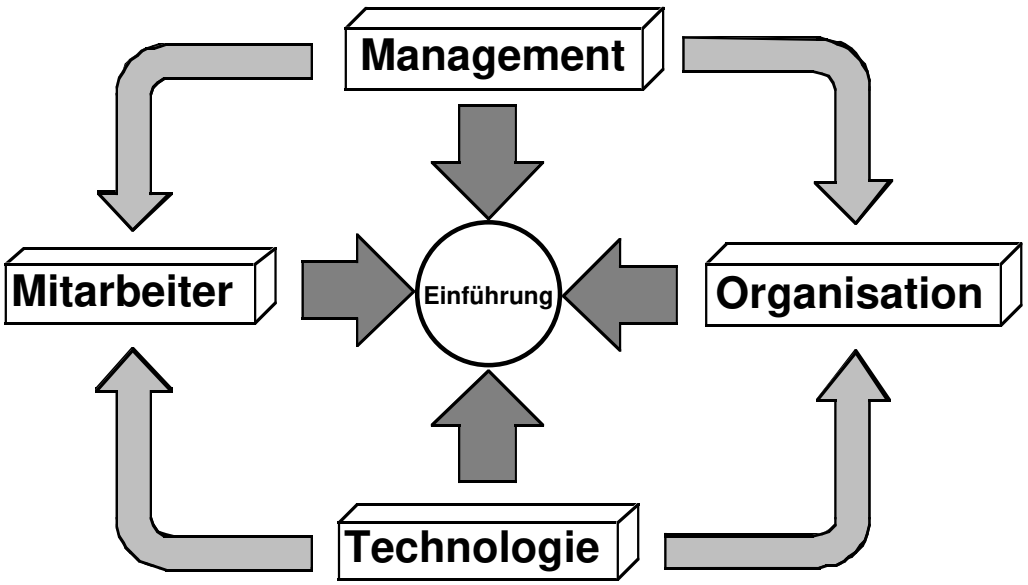


Abb. 1: Einflußfaktoren

erheblich sein. Die zuletzt genannte Risikogruppe wird gemildert, wenn die Einführung sinnvoll geplant und systematisch durchgeführt wird. Die nachfolgende Abbildung zeigt wesentliche Einflußfaktoren und Randbedingungen, die zu beachten sind.

Das *Management* muß die Einführung der OO-Technologie vollumfänglich tragen und mitverantworten. In diesem Zusammenhang müssen dem Management die Chancen, aber auch die Risiken, die in der OO-Technologie liegen, bekannt sein. Letzteres ist – bei der Flut von Lobpreisungen – besonders wichtig. Weiterhin muß dem Management klar sein, welcher Aufwand zu leisten ist, damit ein konsolidiertes OO-Engineering in der Unternehmung entstehen kann; ihm muß klar sein, daß sich die vorhandenen Potentiale der OO-Technologie sowie ein messbarer Nutzen in Form von projektübergreifender Wiederverwendung nicht kurzfristig einstellen werden, sondern erst mittelfristig zu erwarten sind. Kurz gesagt: Die Einführung der OO-Technologie muß "Chefsache" sein.

Die *Mitarbeiter* müssen die OO-Technologie in ihren Projekten umsetzen. Damit dies erfolgreich möglich sein kann, müssen sie motiviert werden, Neues zu lernen. Da dieses mit nicht unerheblichem Aufwand für jeden einzelnen Mitarbeiter verbunden ist, ist dies nicht immer einfach („Wir machen das doch schon 15 Jahre so und die Anwendungen laufen doch prima“). Wird der Aufwand von den Mitarbeitern investiert, so muß dieser „belohnt“ werden. Dies kann in extrinsischer oder intrinsischer Form geschehen. Da die OO-Technologie nicht in einer „softwarekulturfreien“ Umgebung eingeführt wird, die häufig durch die traditionelle Host-Entwicklung geprägt ist, muß verhindert werden, daß eine sich gegenseitig hemmende Zwei-Welten-Kultur entsteht. Die eine – alte – Welt darf nicht als die ewiggestrige, die zweite - die OO-Welt – nicht als die allein-seligmachende Welt dargestellt werden. Dazu muß, und dies ist Sache des Managements, der Stellenwert beider Technologien klar formuliert sein und das Mit- und Nebeneinander der Technologien deutlich gemacht werden.

In diesem Zusammenhang sei am Rande erwähnt, daß die Umsetzung der OO-Technologie nicht – wie ab und an fälschlich zu lesen oder zu hören ist – dazu führt, daß die vorhandenen Kenntnisse der Mitarbeiter nichts mehr wert sind und nicht mehr gebraucht werden. Im Gegenteil bilden diese doch die Basis und den Grundstock, um die neuen Konzepte der OO-Technologie zu schulen und ihre Vorteile gegenüber älteren bekannten Konzepten zu erläutern.

Die *Organisationsform* der Projekte muß unter Umständen an die Bedürfnisse der OO-Technologie angepaßt werden. In Kilberth et al. (1993) wird festgestellt, daß sich in objektorientierten Projekten gezeigt hat, daß ein objektorientiertes Anwendungssystem nicht völlig unabhängig von der Organisationsform entwickelt werden kann. Die vorhandene Aufteilung in Geschäftsbereiche mit ihren jeweiligen Zuständigkeiten kann sich gelegentlich als sperrig erweisen. So ist etwa zu berücksichtigen, daß anwendungsnah arbeitende Systemanalytiker oder DV-Berater viel stärker als vorher üblich in ein Entwicklungsprojekt integriert werden müssen. Dies wirft dort Probleme auf, wo fachliche Verantwortung und personelle Zuständigkeit, bedingt durch die Organisationsstruktur, auseinanderfallen. Als Lösung

empfiehlt sich meist ein Matrixmanagement, das Mitarbeiter aus ver schiedenen Abteilungen für den Ablauf eines Projektes fachlich einer Projektleitung unterstellt.

Soll objektorientierte Entwicklung nicht nur in einem einzelnen Projekt, sondern in zeitlich und thematisch parallelen Projekten erfolgen, dann erfordert das sowohl umfassende Werkzeugunterstützung als auch personelle und organisatorische Voraussetzungen. Von herausragender Bedeutung für die mittelfristige Realisierung der Aspekte Flexibilität, Wiederverwendbarkeit und Offenheit ist dabei, daß die in den entstehenden Klassenbibliotheken und Frameworks dokumentierten Konzepte kontinuierlich gepflegt und weiterentwickelt werden. Dazu muß nicht nur bekannt sein, wie OO-Programmtexte technisch zu verwalten sind, sondern dies bedeutet auch, daß kontinuierlich Wissen und Erfahrung weitergegeben werden muß.

Nicht zuletzt müssen die Elemente der einzuführenden *OO-Technologie* bekannt und auf die Bedürfnisse der Projekte abgestimmt sein. Da es zur Zeit eine Vielzahl von methodischen Ansätzen im Bereich der objektorientierten Analyse und des Entwurfs gibt, verschiedene Sprachen- und Werkzeugalternativen existieren, müssen die Elemente ausgewählt und zusammengestellt werden, die den Anforderungen der ersten OO-Projekte am besten gerecht werden. Bei der Auswahl dieser Projekte muß darauf geachtet werden, daß diese von der Aufgabenstellung besonders geeignet sind, um sie in der OO-Technologie durchzuführen. Positive Erfahrungen existieren zur Zeit im Bereich der interaktiven Auskunfts- und Beratungssysteme.

3. OO-Technologie bei der SBG - Strategie der Einführung

In diesem Abschnitt möchte ich kurz beschreiben, welche Strategie die SBG umsetzen will, um die OO-Technologie erfolgreich einzuführen und zu etablieren.

In der SBG existiert bereits seit ca. fünf Jahren in Form des eigenen Informatik-Forschungslabors UBILAB eine Keimzelle für die OO-Technologie. In den letzten drei Jahren wurden neben den doch eher Labor-orientierten Arbeiten des UBILAB´s einzelne Projekte in OO-Technologie entwickelt. Erst in diesem Jahr ist jedoch geplant, die OO-Technologie systematisch einzuführen

Marty (1994) beschreibt die folgenden Aspekte für die Einführung der OO-Technologie bei der SBG:

- Es ist ein gradueller Übergang hin zur OO-Technologie geplant, der mindestens fünf Jahre dauern wird.
- Die Einführung soll in kleinen nachvollziehbaren Schritten durchgeführt werden.
- Erste Erfahrungen sollen an speziellen Pilotprojekten erhalten werden.
- Um eigene SBG-angepaßte Klassenbibliotheken oder Frameworks zu erstellen, soll verstärkt auf zugekauften „Halb-Produkten“ aufgebaut werden.

3.1 Die OO-Gruppe - Selbstverständnis und Ziele

In der SBG wird zur Zeit eine Gruppe aufgebaut, die mit der Einführung der OO-Technologie und deren Umsetzung in den Projekten betraut sein wird. Die Etablierung dieser OO-Gruppe ist letztlich eine Konsequenz der oben für die OO-Technologie beschriebenen organisatorischen Voraussetzungen. Das Selbstverständnis und die Aufgaben der OO-Gruppe werden nachfolgend erläutert.

Die OO-Gruppe ist ein Anbieter von Dienstleistungen.

Die OO-Gruppe versteht sich als ein Dienstleistungsanbieter. Die angebotenen Dienstleistungen müssen den Erfordernissen der OO-durchgeführten Projekte entsprechen. In diesem Zusammenhang ist es wichtig, daß die Projekte einen direkten meßbaren Nutzen haben, wenn sie auf die Dienste der OO-Gruppe zurückgreifen. Damit dies erreicht werden kann, müssen die angebotenen Dienstleistungen bekannt sein und mehrheitlich akzeptiert werden.

Die OO-Gruppe ist der Synergiepunkt für die OO-Technologie.

Die OO-Gruppe ist die zentrale Stelle, die Ergebnisse, die in verschiedenen Projekten erarbeitet werden, aufbereitet, pflegt und anderen Projekten zur Verfügung stellt. Hierzu zählen insbesondere die eingesetzten Klassenbibliotheken, aber auch die verwendeten Werkzeugumgebungen. Die Chancen, die die OO-Technologie im Bereich Wiederverwendung bietet, sollen dadurch aktiv genutzt und zielführend umgesetzt werden.

Die OO-Gruppe arbeitet innovations-orientiert.

Die OO-Technologie ist noch lange nicht ausgereift, sondern unterliegt ständiger Veränderungen und Verbesserungen. Dieser Tatsache muß die OO-Gruppe dadurch Rechnung tragen, daß sie die sich stabilisierenden Neuerungen erkennt, bewertet und eventuell umsetzt. Die OO-Gruppe beobachtet dazu u.a. den sich rasch entwickelnden Markt für Methoden, Werkzeuge, Klassenbibliotheken und Frameworks.

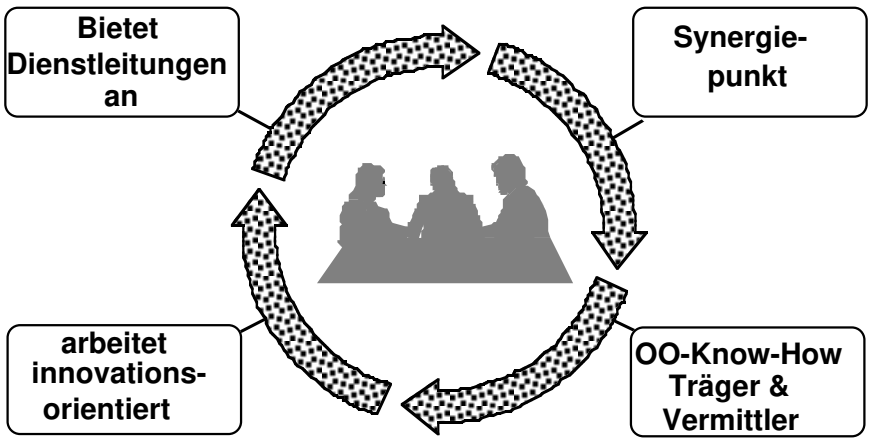


Abb. 2: Selbstverständnis der OO-Gruppe

Die OO-Gruppe ist der zentrale Know-How-Träger der OO-Technologie.

Damit die OO-Technologie systematisch eingeführt, verbreitet und stabilisiert werden kann, muß es eine zentrale Stelle geben, in der das eingesetzte Know-How zusammengefasst ist. Die OO-Gruppe muß alle aktuell verwendeten Sprachen, Methoden und Werkzeuge, sowie die praxis-orientierten QS-Maßnahmen kennen, die speziell auf OO-Projekte zugeschnitten sind. Sie muß weiterhin in der Lage sein, dieses Wissen angemessen zu vermitteln.

3.2 Aufgabenfelder der OO-Gruppe

Aus den oben genannten Zielen lassen sich die folgenden zentralen Aufgabenbereiche der OO-Gruppe identifizieren:

A. Zeitliche begrenzte Mitarbeit in OO-Projekten

Die Arbeit der OO-Gruppe darf kein Selbstzweck sein. Im Sinn eines Dienstleistungsanbieters können deshalb die Mitarbeiter der OO-Gruppe in einem zeitliche beschränkten Maße in OO-durchgeführten Projekten mitarbeiten. Dies führt auf der einen Seite dazu, daß die OO-Gruppe ständig mit den Problemen und der Entwicklungssituation der Projekte vertraut bleibt und dementsprechend darauf reagieren kann. Auf der anderen Seite ist dies die Voraussetzung, um Gemeinsamkeiten im Sinn der Synergie und Wiederverwendung zu entdecken und umzusetzen.

Aus heutiger Sicht werden die folgenden Schwerpunkte beim Einsatz in einem Projekt gesehen:

- Unterstützung bei der Einarbeitung in die verwendete Entwicklungsumgebung, Sprache und Methode.
- Vermitteln der eingesetzten Klassenbibliotheken.
- Mitarbeit und Beratung im Bereich Analyse, Design- und Implementierung.
- Anstoß von QS-Aktivitäten.

B. Pflege der vorhandenen und entstehenden wiederverwendbaren Bausteine

Damit projektübergreifend nutzbare Klassenbibliotheken entstehen können, muß es eine Stelle geben, die sich intensiv mit den bestehenden eingekauften Klassenbibliotheken und mit den in den Projekten entwickelten Klassen auseinandersetzt. Dies können die einzelnen Projekte aus Termin- und Kostengründen nicht leisten. Die OO-Gruppe hat in diesem Zusammenhang die Aufgabe, in sich konsistente, aktuelle und wiederverwendbare Klassen in Form von Bibliotheken oder sogar Frameworks zur Verfügung zu stellen.

Damit dieses Ziel erreicht werden kann, müssen seitens der OO-Gruppe die folgenden Aktivitäten durchgeführt werden:

- Die vorhandenen Klassen müssen den Projekten vermittelt werden. Beim Entwurf neuer Klassen muß darauf geachtet werden, daß die existierenden Klassen sinngemäß verwendet und eingesetzt werden.
- Entstehen in einem Projekt neue Klassen, die nicht problemspezifisch, sondern genereller Natur sind, so müssen diese in die Klassenbibliothek aufgenommen werden. In der Regel müssen solche Klassen aber einem Redesign unterzogen werden. Weiterhin kann es notwendig sein, daß existierende Klassen verändert und aktualisiert werden müssen, wenn neue Klasse in eine Bibliothek aufgenommen werden. Dies ist die Aufgabe der OO-Gruppe.

C. Aufbau eines OO-Engineerings

Die OO-Technologie ist nur auf der Basis eines entsprechenden Software-Engineerings umsetzbar. Dazu zählen alle Aspekte, die für den Entwicklungsprozeß relevant sind. Die OO-Gruppe muß in diesem Zusammenhang einen OO-Engineering-Ansatz erarbeiten, definieren und in die Projekte einfließen lassen. Dieser Prozeß wird iterativ sein, da sich das Engineering nicht ad hoc abschließend definieren läßt, sondern sich aufgrund der in den Projekten gemachten Erfahrungen anpassen und verändern wird.

Es lassen sich die folgenden groben Themengebiete im Bereich OO-Engineering identifizieren:

- Erarbeiten einer praxisnahen und praxisgerechten Qualitätssicherung für OO-Projekte (Reviews, Metriken, Testverfahren etc.).
- Erarbeiten von Standards und Konventionen (Programmierrichtlinien, Dokumentationsstandards, Bezeichnerkonzept etc.)
- Bedingt durch den Einsatz integrierter OO-Entwicklungsumgebungen wie etwa Smalltalk muß ein Konzept für ein geeignetes Versions- und Konfigurationsmanagement erarbeitet und umgesetzt werden.

D. Erarbeiten eines OO-angepaßten Schulungskonzeptes

Damit die objektorientierte Technologie in den Projekten umgesetzt werden kann, müssen die Projektmitglieder entsprechend geschult sein. Dabei ist zu beachten, daß die einzelnen Mitarbeiter einen unterschiedlichen Wissenstand haben. Weiterhin müssen die zu schulenden Inhalte so gewählt sein, daß sie den Anforderungen der Projekte genügen. Dies kann durch ein in sich abgestimmtes Schulungskonzept erreicht werden.

Die Aufgabe der OO-Gruppe besteht in diesem Zusammenhang darin, die Inhalte und die Reihenfolge der aufeinander aufbauenden Kurse mitzugestalten. Das Schulungskonzept kann grob in zwei Kategorien gegliedert werden. Die erste Kategorie enthält Kurse, die das der OO-Technologie zugrundeliegende softwaretechnische Basiswissen vermitteln. Dazu zählt im einzelnen das Modulkonzept, das Geheimnisprinzip, die Datenkapselung, Abstrakte Datentypen, Generizität, Polymorphie, Typsysteme, dynamisches Binden, evolutionäre Prozeßmodelle und Prototyping. In der zweiten Kategorie sind Kurse enthalten, die spezielle Themen die Objektorientierung vermitteln. Folgende Kurse sind denkbar:

- Überblick über die OO-Software-Entwicklung
- Objektorientiertes Programmieren
- Qualitätssicherungsmaßnahmen in OO-Projekten
- Objektorientierter Systementwurf (Methoden, Software-Architekturen)
- Objektorientierte Frameworks – Konzepte und Designmuster

E. Aufbau und Unterhalt des OO-Entwicklungs-Environments

Die OO-Software-Entwicklung findet in zunehmendem Maße auf speziellen, teils integrierten Entwicklungsumgebungen (Smalltalk, Visual C++) statt und wird durch speziell geeignete Werkzeuge unterstützt. Die Auswahl, das Know-How und die Pflege der eingesetzten Werkzeuge sollte sinnvollerweise an einem Ort zusammengefaßt sein.

Durch den Mix von Projektarbeit und Innovationstätigkeit der OO-Gruppe können die Werkzeuge so ausgewählt und notfalls adaptiert werden, daß sie den Erfordernissen der Projekte und den Erfordernissen des OO-Engineering-Ansatzes entsprechen.

3.3 Bewertung

In diesem Bericht ist der aktuelle Stand des Denkens dokumentiert, der bzgl. der Einführung der OO-Technologie in der SBG vorhanden ist. Da wir zur Zeit erst in der Initialphase der Einführung sind, können keine Erfahrungen berichtet werden. Wir glauben jedoch, mit der gewählten Vorgehensweise folgende Ziele erreichen zu können:

- Die Technologie kann geplant, schrittweise, systematisch und nachvollziehbar eingeführt werden.
- Die vorhandenen noch knappen OO-Ressourcen werden in Form der OO-Gruppe zusammengefaßt, so daß eine kritische Masse entstehen kann, die notwendig ist, damit Erfolge erzielt werden können.
- Durch die Arbeit der OO-Gruppe besteht die Möglichkeit, eine unternehmensweite OO-Basis zu bilden. Dazu zählt neben den Klassenbibliotheken insbesondere auch das Know-How im Umgang mit OO-Projekten.
- Die OO-Technologie soll nicht überall und nicht zu jedem Preis eingesetzt werden. Es sollen die Projekte in dieser Technologie realisiert werden, die besonders dafür geeignet sind. Die OO-Technologie wird nicht zum Selbstzweck, sondern als „Nutzengenerator“ eingesetzt.

Literatur

Kilberth, K., G. Gryczan, H. Züllighoven (1993): **Objektorientierte Anwendungsentwicklung: Konzepte, Strategien, Erfahrungen**. Vieweg, Wiesbaden.

Bürkle, U., G. Gryczan, H. Züllighoven (1992): Erfahrungen mit der objektorientierten Vorgehensweise in einem Bankenprojekt. **Informatik-Spektrum 15**, Heft 5, 273-381.

Marty, R (1994): Klassische Entwicklungstechnologien ungenügend. **Computer Woche Extra** 1, Februar 1994, 38-40.

Teil 9

Software Engineering in der Universität

Jochen Ludewig

Dem Ingenieur ist nichts zu schwer,
Er türmt die Böschung in die Luft,
Er wühlt als Maulwurf in der Gruft,
Kein Hindernis ist ihm zu groß,
Er geht drauf los.

Heinrich Seidel, 1842 – 1906

Zusammenfassung

Die Rolle derer, die in einer Universität das Fach Software Engineering vertreten, ist nach wie vor unklar und schwierig. Die Gründe liegen im Gebiet selbst, in der traditionellen Struktur der Hochschule, in der Erwartung der Umgebung und in der Praxis außerhalb der Universitäten.

Infolgedessen muß ich als Hochschullehrer dieses Gebietes die Fragen nach Zielen, Grenzen, Maßstäben und Erfolgskriterien immer wieder selbst stellen und beantworten. Das sind die Themen dieses Beitrags. Er knüpft an Überlegungen, die ich in meinen Antrittsvorlesungen in Zürich und Stuttgart entwikkelt hatte (Ludewig, 1986, 1989).

1. Schwierigkeiten mit dem Begriff Software Engineering

Laut IEEE Std. 610.12 (1990) (Standard Glossary of Software Engineering Terminology) ist

> **software engineering**
> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
> (2) The study of approaches as in (1).

So nett diese Definition auf den ersten Blick aussieht, sie hilft uns kaum. Die darin enthaltene Wertung läßt erkennen, daß damit mehr ein *Programm* als eine *Definition* gemeint ist. Aber auch bei jedem anderen Versuch, den Begriff sinnvoll zu fassen, begegnen uns verschiedene Schwierigkeiten:

(1) *Das* Problem des Software Engineerings fehlt.
(2) Über den Zweck besteht kein Konsens.
(3) Software Engineering ist ein riesiges Gebiet.
(4) Software Engineering ist keine Disziplin, die klar von den anderen abgegrenzt werden kann
(5) Software Engineering ist an der Hochschule kaum möglich, in der Praxis kaum üblich.

1.1 Kein Problem, viele Probleme

Das zentrale Problem des Software Engineerings ist, daß es im Software Engineering kein zentrales Problem gibt, sondern viele verfilzte Einzelprobleme... (aus Ludewig, 1989)

In den siebziger Jahren war es üblich, die Schwierigkeiten mit der Software auf irgendeinen „offensichtlichen" Mangel zurückzuführen, z.B. die falsche Programmiersprache oder das Fehlen einer formalen Spezifikation. Das war eine Form des Wunderglaubens, tatsächlich gibt es den einen Drachen nicht, dessen Tod uns erlöst. Die Hoffnung auf die Freikugel (Brooks, 1987) setzt den Glauben an das Monster voraus. Wer mit Mücken kämpft, kann mit Freikugeln nichts anfangen.

1.2 Ein Gebiet ohne klaren Zweck

Auch der Zweck des Software Software Engineerings ist nicht klar; ich habe mich entschieden, als (einzigen) Zweck die Kostenminimierung zu akzeptieren, aber ich rechne dabei nicht mit großer Zustimmung.

1.3 Ein riesiges Gebiet

Alles, was mit der Entwicklung oder Bearbeitung von Software zu tun hat, fällt in die Zuständigkeit des Software Engineerings; das ist viel. Darum kann jeder Einzelne nur einen Ausschnitt kennen und vertreten. Die Gemeinsamkeiten dieser Leute sind entsprechend gering, wie die seit 1992 laufende Tagung „Software Engineering im Unterricht der Hochschulen" (Ludewig, Schneider, 1992) deutlich zeigt. Das „Arbeitsgebiet Software Engineering" ist eine Fiktion, so wie das „Heimatland Afrika" eine Fiktion wäre.

1.4 Ein Gebiet in vielen Gebieten

Baumstrukturen schaffen klare, disjunkte Subsysteme, in denen alles seinen bestimmten Platz hat. Auch Universitäten haben mit ihren Fakultäten, Instituten und Lehrstühlen eine Baumstruktur, und es ist praktisch, daß diese Struktur weitgehend auch im fachlichen gilt. Das Software Engineering jedoch liegt quer: Wer ein Programm schreibt, betreibt (vielleicht, je nach Definition) Software Engineering, und was der Software-Ingenieur realisiert, gehört stets *auch* zu einem (anderen) Spezialgebiet der Informatik.

Unser Arbeitsgebiet überlappt also mit allen anderen Arbeitsgebieten, und *jeder* Informatik-Dozent unterrichtet Software Engineering, auf seine Art.

Wir haben damit das alte Dilemma derer, die amtlich ein populäres Thema vertreten (wie auch Pfarrer, Deutschlehrer, Berufsberater): Entweder wir haben die gleiche Position wie alle unsere nebenberuflichen Kollegen (dann wären wir überflüssig), oder wir haben eine besondere (dann stehen unsere Chancen, einen Effekt zu erzielen, schlecht).

1.5 Software Engineering? Nein danke!

Akzeptiert man, daß das Ziel des Software Engineerings die Kostenminimierung ist, dann sind die Rahmenbedingungen an der Hochschule ungünstig. Denn die Studenten kennen zwar meist den Preis einer Speichererweiterung, aber nicht den Wert geistiger Arbeit. Was sie nicht bezahlt bekommen, kann nicht wertvoll sein.

Aber auch die übrigen Angehörigen der Hochschule, Mitarbeiter, Professoren und Verwaltung, denken nicht in Kosten. Wir haben Stellen (die sind gratis) und Mittel (meist überraschend gegen Jahresende), aber keine Kosten. Die Universität ist in diesem Sinne das letzte Refugium des real existierenden Sozialismus. Wir schulen darum eine Denkweise, die die Kosten als Angelpunkt hat, unvermeidlich so, wie die Parteihochschule in Moskau vermutlich den Außenhandel geschult hat.

Es ist daher naheliegend, den Blick hoffnungsvoll in die „freie Wirtschaft“ zu richten. Was sehen wir? Fast überall guten Willen und sinnvolle Ansätze, aber keinen langen Atem, um die „offiziellen“ Ziele auch im Auge zu behalten. Solange keine zusammenhängende Kostenrechnung für Software auf dem Tisch liegt, wird es unmöglich sein, Änderungen vorzunehmen, die fast jeder für sinnvoll und rentabel hält. Beispielsweise sollte der Aufwand in den frühen Phasen der Software-Entwicklung erhöht werden, damit die Wartung billiger wird. Was sich nicht im nächsten Quartalsabschluß vorteilhaft auswirkt und keinen simplen Effekt auf Kosten und Erträge hat, ist chancenlos.

Die Trägheit der Software-Leute kommt hinzu, und kaum ein Management nimmt es auf sich, von ihnen hart zu fordern, was jeder Ingenieur selbstverständlich liefert, nämlich Resultate nach dem Stand der Technik, innerhalb der vorgesehenen Zeit und zu den vorher geschätzten Kosten.

Darum können wir unseren Studenten und Mitarbeitern in der Praxis kaum Vorbilder zeigen: Die Praxis ist nicht vorbildlich. Wer daran zweifelt, sollte den Versuch machen, dort vorbildliche Software oder Projekt-Dokumentationen als Anschauungsmaterial zu erhalten. Wir haben es mehrfach probiert und sind uns sicher: Das ist entgegen naheliegenden Vermutungen kein Problem der Geheimhaltung.

2. Ziele und Grenzen

Die vorstehende Diskussion zeigt, daß wir im Software Engineering zunächst den Boden ebnen müssen, auf dem wir dann die eigentliche Arbeit leisten können. Sie zeigt auch, daß wir, die Software-Engineering-Professionals, uns langfristig überflüssig machen sollten. Eine Ingenieur-Disziplin, die ihren Namen verdient, braucht keine Spezialisten zur Pflege des Ingenieur-Gedankens.

Was können und sollen wir, solange wir noch nicht überflüssig sind, leisten?

In der Forschung sollten wir versuchen, die Methoden und Mittel bereitzustellen, die eine rationale Bewertung der Methoden und Mittel in der Informatik unterstützen. Erst wenn wir in der Lage sind, den Wert zweier CASE-Tools sinnvoll zu vergleichen, den Aufwand und Nutzen einer Neuimplementierung dem der weiteren Wartung gegenüberzustellen, den Einfluß der Programmiersprache zu quantifizieren, dann haben wir die Grundlagen für ein rationales Software Engineering. Metriken, allgemeiner Bewertungsverfahren sind also nach meiner Einschätzung die Schlüsseltechnologie.

In der Lehre sollten wir unsere Hörer darauf vorbereiten, ihnen wie auch immer glaubhaft machen, daß es ein Leben *nach* der (heutigen) Praxis gibt, daß man Software also auch ganz anders machen kann. Nach wie vor gilt mein Programm, eine SE-Guerilla auszubilden, die auch in einer dem Software Engineering feindlichen Umgebung überleben kann, ohne zum „Feind“ überzulaufen.

Und in der Hochschule selbst müssen wir weiterhin um unsere Kollegen als Verbündete werben und die Rolle spielen, die auch die Fachleute für Hygiene in den Universitätskliniken haben: *Das* besonders überzeugend tun, was *alle* tun sollten.

Quellen und Referenzen

Brooks, F.P., Jr. (1987): No silver bullet - essence and accidents of software engineering. **IEEE COMPUTER 20**, 4, 10-19.

Ludewig, J. (1986): Software Engineering: Computer-Programme als technische Produkte. Antrittsvorlesung an der ETH Zürich, Mai 1986. **Technische Rundschau 79** (1987), Heft 7, 50-57.

Ludewig, J. (1989): Modelle der Software-Entwicklung: Abbilder oder Vorbilder? Antrittsvorlesung an der Universität Stuttgart, Juni 1989. **Softwaretechnik-Trends**, **9**, 3 (Okt. 1989), 1-12.

Ludewig, J., K. Schneider (Hrsg.) (1992): **SEUH (Software Engineering im Unterricht der Hochschulen)**. Berichte des German Chapter of the ACM, Band 37, Teubner, Stuttgart.