Modelle im Software Engineering – eine Einführung und Kritik

Jochen Ludewig

Institut für Informatik der Universität Stuttgart
Breitwiesenstr. 20-22, D-70565 Stuttgart
Tel. 0711-7816-354, Fax 0711-7816-380
ludewig@informatik.uni-stuttgart.de

Zusammenfassung: Der Modellbegriff ist fundamental für das Software Engineering. Er wird im folgenden Beitrag definiert und unter verschiedenen Aspekten differenziert. Die wichtigsten Modell-Arten werden an Beispielen vorgestellt. Im zweiten Teil werden einige Probleme diskutiert, die bei der Verwendung von Modellen auftreten. Das SESAM-System, das in der Universität Stuttgart entwickelt wurde, dient als Beispiel für komplexe Modelle.

1 Rechtfertigungsversuch

Wer in einer einführenden Vorlesung an der Hochschule über Modelle spricht, kann sicher sein, den Studierenden etwas sehr Nützliches zu bringen. Wer aber über dasselbe Thema auf einer Tagung vorträgt, deren Teilnehmer das aktive Interesse am Modellbegriff verbindet, muss sich rechtfertigen. Hier also ein Versuch:
Wir alle arbeiten ständig mit Modellen, bewusst und – weit öfter – unbewusst. Der Modellbegriff ist so fundamental, dass er in der Regel weder explizit eingeführt noch abgegrenzt, d.h. definiert wird. In bestimmten Zusammenhängen wird oft von Modellen gesprochen, allerdings meist ohne Begründung.
Indem wir den Modellbegriff selbst zum Gegenstand der Betrachtung machen, rüsten wir uns für einen rationalen Umgang mit Modellen aus und verbessern die Fähigkeiten,

- Modelle zu erkennen, wenn wir sie sehen oder benutzen,
- Modelle hinsichtlich ihrer Eigenschaften und ihrer Mächtigkeit zu analysieren,
- klar zu unterscheiden zwischen Modell und Original,
- wo nötig, gezielt und mit der notwendigen Umsicht neue Modelle zu schaffen.

Dazu soll nachfolgend ein bescheidener Beitrag geleistet werden.

2 Modelle als Lebensmittel

Modellierung ist das uns angeborene Verfahren, das komplexe Universum auf eine überschaubare Welt zu reduzieren. Indem wir sichtbare und unsichtbare Phänomene auf *Begriffe* abbilden und nur noch mit diesen umgehen, wird die Gesamtzahl der zu betrachtenden Gegenstände beherrschbar, und wir werden in die Lage versetzt, *Erfahrungen* zu

sammeln, *generische Urteile* (mit negativer Konnotation als *Vorurteile* bezeichnet) quasi auf Vorrat zu fällen und allgemein Strategien zu entwickeln, um mit der realen Welt zurecht zu kommen.
Damit bildet die Fähigkeit zur Modellierung, zur Abstraktion und zur Konkretisierung, einen zentralen Teil unserer menschlichen Grundausrüstung. Anders als angeborene Instinkte und Reflexe versetzen uns die Modelle in die Lage, zu *lernen* und die Welt in jeder Hinsicht zu erobern. Die für den Menschen charakteristische Fähigkeit zur *Reflektion* ist unmittelbar daran geknüpft.
In der *Abstraktion*, die mit der Schaffung von Modellen stets verbunden ist, liegt ihre Stärke: Das Modell gilt nicht nur für einen einzigen konkreten Gegenstand, für ein einziges Phänomen, sondern für mehrere, unbegrenzt viele, für eine *Klasse* von Gegenständen. Wer in den Änderungen des Wasserstandes den Rhythmus von Ebbe und Flut erkennt, also ein Modell entwickelt, kann sich darauf einstellen. Wer die Beobachtung, dass ein ganz bestimmtes Objekt schnell, stark und gefährlich ist, auf alle ähnlichen Wesen überträgt, hat bessere Chancen, sich zu schützen.

Während wir leben, also agieren und reagieren, setzen wir fortwährend Modelle ein. Das geschieht in aller Regel unbewusst. Anders in der *Wissenschaft*: Hier wird die Entwicklung von Modellen thematisiert und zum eigentlichen Zweck der Forschung erhoben. Das Resultat einer Forschung ist in jedem Falle ein Modell, eine *Theorie*. Je stärker sich die Anwendung dieser Theorie auf die Welt auswirkt, sei es durch physische Konsequenzen bis hin zur Atombombe, sei es durch Änderungen des Weltbildes im engeren oder im weiteren Sinne (heliozentrisches System, Evolutionslehre), um so höher wird die Bedeutung der Forschung eingeschätzt.
Eine ähnliche Rolle spielt die Modellierung in der *Technik*. Hier unterstützt das Modell die Entwicklung von *Artefakten*, sei es die Theorie der Mechanik beim Brückenbau oder die Spezifikation bei der Software-Entwicklung. Bei der Verbreitung von Technik sind auch Modelle in Form von Schnittstellen-Definitionen von sehr großer Bedeutung.
Beim Umgang mit Modellen ist es wichtig, Modelle sinnvoll zu wählen, sich des Unterschieds zwischen Modell und Original bewusst zu sein und die Möglichkeiten und Grenzen der Modelle realistisch einzuschätzen.

3 Begriffsbestimmung

Das Wort „Modell“ ist laut Kluge, „Etymologisches Wörterbuch der deutschen Sprache“, die moderne, vom italienischen *modello* geprägte Variante des älteren „Model“ (maskulin), das aus dem lateinischen *modulus*, in der Bedeutung *Maß, Regel, Form, Muster, Vorbild* entstanden ist. Der Sinn hat sich also kaum geändert.
Modelle sehen wir laufend, beispielsweise Modelleisenbahnen, Schaufensterpuppen, Landkarten, Architekturmodelle, im Software-Bereich Prozessmodelle, Entwurfsmuster, Klassendiagramme. Neben diesen offensichtlichen Modellen gibt es noch eine weit größere Zahl, die uns wie Projektpläne, Spezifikationen und Entwürfe, Metriken und Sitzungsprotokolle kaum als Modelle erscheinen.

3.1 Die Modellmerkmale

Nach Stachowiak [St1993] sind jedem Modell zwingend drei *Merkmale* zueigen, sonst ist es kein Modell:

- **Abbildungsmerkmal:** Zum Modell gibt es das Original; ein Modell ist niemals eigenständig, ohne Original kein Modell.
- **Verkürzungsmerkmal:** Ein Modell gibt nicht alle Attribute des Originals wieder, sondern nur eine echte Teilmenge.
- **Pragmatisches Merkmal:** Ein Modell hat den Zweck, unter bestimmten Bedingungen und bezüglich bestimmter Fragestellungen das Original zu ersetzen.

Durch das Abbildungsmerkmal ist nicht impliziert, dass das Original wirklich existiert, es kann auch geplant, vermutet oder der Fantasie entsprungen sein. So ist ein Gartenzwerg das Modell eines gedachten Wesens, der Stadtplan von Troja spiegelt eine umstrittene wissenschaftliche Hypothese. Fast jeder Roman oder Kinofilm ist das Modell einer erdachten Realität. Die Kostenschätzung eines Softwareprojekts ist ein spekulatives Modell des Projektverlaufs.
Ein Modell kann das Original eines anderen Modells sein, wir haben *kaskadierte Modelle* vor uns, etwa, wenn ein Gemälde ein Atelier mit Gemälden zeigt. In der Software ist die Kaskade von der Spezifikation zum Code und weiter zur Programmausführung ein prominentes Beispiel.
Das Verkürzungsmerkmal bedeutet minimal, dass die Identität verlorengeht, wo sie überhaupt besteht (nämlich bei materiellen Originalen). Was zunächst als Verlust erscheint, ist in der Regel die Stärke des Modells: Durch die Verkürzung wird es überschaubar, handlich, brauchbar im Sinne des pragmatischen Merkmals.
Das pragmatische Merkmal gibt dem Modell seinen Sinn. Weil wir nicht unmittelbar auf das Original zugreifen wollen oder können, verwenden wir an seiner Stelle das Modell. Das gilt für das Plüschtier ebenso wie für die Urknalltheorie.

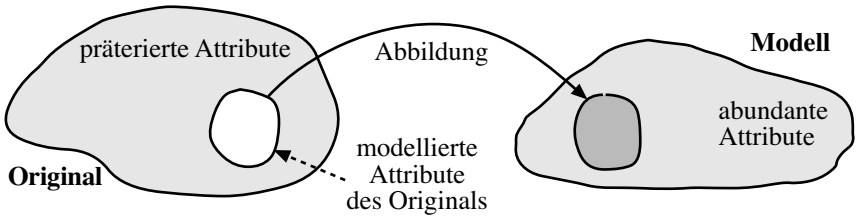

Abb. 1: Original und Modell nach Stachowiak

Abb. 1 zeigt den Zusammenhang zwischen Original und Modell. Man beachte, dass die Modelle den Originalen nicht äußerlich ähnlich sein müssen, wie es bei einem Modellauto der Fall ist. Die abgebildeten Attribute können im Modell auf verschiedenste Weise erscheinen, z.B. Farben als Grauwerte (im Foto), physikalische Eigenschaften als Zahlen (Messwerte), Empfindungen als Farben oder Klänge (in der Kunst), Laufzeiten von Programmen als Balken in Diagrammen.

Durch Verkürzung fallen die *präterierten* (d.h. *übergangenen*) Attribute weg; das Modell weist stattdessen *abundante* (*reichliche* im Sinne von *überflüssige*) Attribute auf, die nichts mit dem Original zu tun haben. So entfallen auf einem Foto die meisten Attribute der fotografierten Motive; das Material des Fotopapiers ist nicht vom Motiv beeinflusst. Viele Details eines Programms sind in der Z-Spezifikation nicht enthalten, dagegen die syntaktischen Merkmale einer Notation (Z), die für das Programm ohne Bedeutung ist. Die in Abb. 1 angegebene Terminologie wird auch dann beibehalten, wenn das Original *nach* dem Modell entsteht; die präterierten Attribute gehören immer zum Original, die abundanten zum Modell.

Betrachten wir als typisches Modell einen Personalausweis:

> Der Ausweis ist für eine Person ausgestellt (Abbildungsmerkmal). Bei einem gefälschten Ausweis ist diese Person fiktiv. Die meisten Eigenschaften des Menschen, z.B. sein Körpergewicht, seine kulinarischen Präferenzen oder seine Kindheitserinnerungen erscheinen auf dem Personalausweis nicht (Verkürzungsmerkmal); nur so lässt sich mit akzeptablem Aufwand ein handlicher Ausweis realisieren. Die Körpergröße ist durch einen Messwert dargestellt, sein Wohnsitz durch eine Adresse. Die Ausweisnummer ist ihrerseits geeignet, den Ausweis zu identifizieren (Modell eines Modells). Anhand des Ausweises können verschiedene Personen Feststellungen treffen, ohne auf den Menschen insgesamt zuzugreifen, beispielsweise lässt sich klären, ob er bestimmte Rechte hat (pragmatisches Merkmal).

3.2 Verwandte Begriffe

Viele geläufige Begriffe sind mit dem des Modells eng verwandt (Abb. 2); nachfolgend wird diskutiert, wieweit wir sie als Modelle akzeptieren wollen.

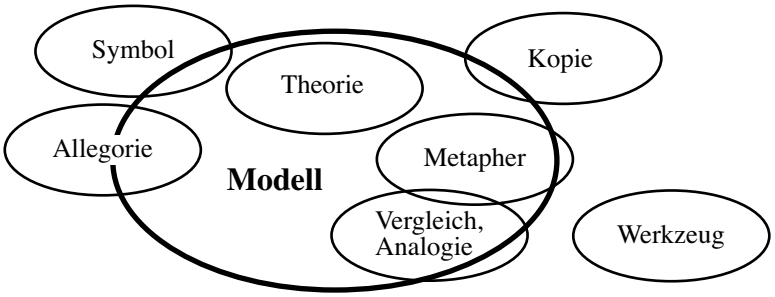

Abb. 2: „Modell“ und verwandte Begriffe

Theorie: Eine Theorie, z.B. die der Sprachklassen von Chomsky, ist das Modell eines Phänomens. Alle drei Merkmale sind offensichtlich vorhanden. Darum bilden die Theorien eine spezielle Klasse der Modelle.

Theorien sind nicht notwendig formal. Auch Brook‘s Law „Adding manpower to a late project makes it even later“ ist eine Theorie.

Metapher: Eine Metapher wie die des Computer-Virus hat ein Original (den schädlichen Code), gibt von diesem keine Details wieder und dient dazu, über das Original zu kommunizieren, etwa in einer Nachrichtensendung. Die Metapher ist also ein Modell. Beim Umgang mit Software sind Metaphern extrem wichtig, weil sie und nur sie uns gestatten, all die Objekte und Vorgänge, die sich unserer Vorstellung verweigern, durch Metaphern zu fassen. Das beginnt mit dem Wort *Software* selbst und setzt sich in nahezu allen Wörtern der Fachsprache und des Jargons fort (beispielsweise *Sprung*, *Schleife*, *Absturz*, *Bug*, *top-down*, *Speicher* und *Memory*, *Wartung*, *Fire Wall*, *Kanal*, *Portal*, *Protokoll, Botschaft*, *Vererbung*, *Lebensdauer* usw. usw.). Die meisten dieser Metaphern sind vermutlich spontan entstanden. Andere wurden bewusst eingeführt, beispielsweise die *Schreibtischoberfläche*, der *Mülleimer*, die *Transaktion*, der *Handshake* usw. *Icons*, wie wir sie ständig sehen und verwenden, sind ebenfalls Metaphern, sie repräsentieren Mechanismen, die wir „anklicken" können.

Die Metaphern der objektorientierten Programmierung, vor allem das Objekt und die Vererbung, zeigen den Pferdefuß der Metaphern: Uns wird Einfachheit und Verständlichkeit suggeriert, wo Dinge weder einfach noch verständlich sind. Die ursprüngliche Bedeutung der Vererbung ist eben an Objekte geknüpft, nicht an Klassen. Und Objekte der realen Welt bilden Klassen, nicht umgekehrt, wie es in der objektorientierten Welt der Fall ist.

Die mit der Metapher verwandte Allegorie, eine Gestalt wie Justitia, die etwas abstraktes oder jedenfalls kaum fassbares repräsentiert, spielt in der Software keine Rolle.

Vergleich, Analogie: Wie die Metapher bedient sich die Analogie eines schon zuvor bekannten Phänomens oder Gegenstands, das Modell wird also nicht neu geschaffen, sondern etwas vorhandenes wird als Modell umgedeutet. Anders als bei der Metapher bleibt das Modellhafte dabei bewusst. Die Erklärung eines Monte-Carlo-Algorithmus durch die Zählung von Schneeflocken, die auf ein Koordinatensystem fallen, ist eine solche Analogie. Probleme gibt es dabei, wenn gerade auf die (typisch ins Auge fallenden) abundanten Attribute Bezug genommen wird. Der „Vergleich hinkt".

Symbol: Beim Symbol sind Abbildungsmerkmal und pragmatisches Merkmal gegeben. Die Verkürzung ist aber radikal, d.h. im Symbol bleibt nichts oder fast nichts vom Original übrig. Flaggen sind typische Symbole, einige (wie die der USA) enthalten noch Hinweise auf den Staat (in diesem Fall durch die Zahl der Sterne). Trotzdem ist die Wiedergabe von Merkmalen des Originals für das Symbol ohne Bedeutung. Darum werden Symbole nicht als Modelle eingestuft.

Kopie: Die bloße Herstellung einer Kopie schafft ein Objekt, das sich als Modell sehr gut eignet, aber es muss erst noch durch eine entsprechende Zweckbestimmung dazu werden. Die Sicherheitskopie einer Harddisk ist sicher ein Modell, denn sie erfüllt alle Kriterien; das pragmatische Merkmal entsteht durch die Verwendung der Daten bei Verlust der Originaldaten. Wer dagegen auf einer Kopierfräse eine Schnitzerei vervielfältigt, arbeitet mit einem präskriptiven Modell und stellt die Originale her.

Werkzeug: Viele, vielleicht alle Werkzeuge sind ursprünglich dadurch entstanden, dass bekannte Hilfsmittel nachgebildet und verbessert wurden. Der Hammer ist eine verbesserte Kopie der menschlichen Faust. Trotzdem werden Werkzeuge hier nicht

als Modelle betrachtet, weil sie sich völlig vom Original lösen können. Wer einen Hammer benutzt, muss den Zusammenhang mit der Faust nicht kennen. Bei einem modernen Rechner ist auch mit viel Fantasie nicht mehr erkennbar, was das Original gewesen sein könnte.

4 Taxonomie der Modelle

4.1 Deskriptive und präskriptive Modelle

Die Modelle im engeren Sinne (ohne Begriffe und Metaphern) können wir nach verschiedenen Kriterien klassifizieren. Sie sind entweder *Abbilder von* etwas oder *Vorbilder für* etwas; entsprechend werden sie als *deskriptive* (d.h. beschreibende) bzw. *präskriptive* (d.h. vorschreibende) Modelle bezeichnet. Eine Modelleisenbahn ist deskriptiv; eine technische Zeichnung, nach der ein Mechaniker arbeitet, ist präskriptiv. Wenn auf dem Foto eines Hauses ein geplanter Umbau skizziert wird, geht das eine in das andere über (*transiente Modelle*). Den Modellen kann man im allgemeinen nicht ansehen, ob sie deskriptiv oder präskriptiv sind; eine Modelleisenbahn könnte auch die Vorgabe für eine zu bauende Bahn sein, eine Konstruktionszeichnung kann bei der Produktspionage entstehen.
Es liegt nahe zu vermuten, dass Abbilder stets *nach*, Vorbilder stets *vor* dem Original existieren. Tatsächlich ist das aber nicht immer so. *Prognostische Modelle* liefern Aussagen über *zukünftige* Objekte und Phänomene. Typische Beispiele sind Wahlprognosen und Kostenschätzungen. Diese Modelle sind deskriptiv, auch wenn vorerst noch nicht greifbar ist, was sie beschreiben. Trotzdem hängt das Modell am Original, nicht umgekehrt, das Original (die tatsächlichen Kosten) wird nicht vom Modell (der Kostenschätzung) beeinflusst. Dagegen ist es unmöglich, dass ein präskriptives Modell *nach* dem Original entsteht. Als Ausnahme mag der Fall erscheinen, wenn durch Reverse Engineering auf der Grundlage des Codes eine Anforderungsspezifikation rekonstruiert wird. Aber natürlich kann das nur gelingen, wenn weitere Informationsquellen zur Verfügung stehen.

Anders als die Kosten*schätzung* bildet eine vorgegebene Kosten*schranke* ein präskriptives Modell. Allerdings kann sich dieses als unvereinbar, inkonsistent mit anderen präskriptiven Modellen erweisen, vor allem mit der Spezifikation. In diesem Falle wird das Produkt die Kostenschranke oder die Spezifikation verletzen.
Deskriptive Modelle haben in der Regel den Zweck, den Zugriff auf bestimmte Informationen so einfach und schnell wie möglich zu machen; ein Organigramm erspart die Erkundung der Personalstrukturen, die Metrik der Programmlänge (LOC) erlaubt eine rasche Beurteilung der Größe eines Systems. Bei den prognostischen Modellen geht es darum, den Zugriff überhaupt zu ermöglichen.
Natürlich hat jedes präskriptive Modell einen Ursprung, den es deskriptiv repräsentiert. Beispielsweise repräsentiert der Dokumentationsplan die Vorstellungen des Projektleiters oder die Forderungen des Projekthandbuchs. Nach erfolgreicher Umsetzung können die präskriptiven Modelle als deskriptive Modelle dienen. Wenn die Dokumentation so entstanden ist, wie im Dokumentationsplan gefordert war, dann ist dieser Plan anschließend das Inhaltsverzeichnis der Dokumentation.

4.2 Der Zweck der Modelle

Auch nach ihrem *Zweck* können wir die Modelle sortieren:

- **Dokumentation** entsteht, wenn Daten im weitesten Sinne aus den vorhandenen generiert und archiviert werden. Sie ist also deskriptiv. Wir unterscheiden
  - *Metriken*, sowohl Produkt-Metriken wie Seitenanzahl der Spezifikation, Anzahl der Klassen, Programmlänge, Speicherbedarf, Antwortzeiten, als auch Projekt- und Prozessmetriken wie Dauer und Kosten der Entwicklung, Zahl der entdeckten Fehler, Zahl der Entwickler.
  - *Aufzeichnungen* (Protokolle) aller Art beispielsweise über durchgeführte Prüfungen und Installationen; eine herausragende Rolle spielen die Aufzeichnungen, die bei der Anforderungsanalyse entstehen. Sie werden zum *Lastenheft* verdichtet, das ein Modell der Kunden-Anforderungen darstellt.
  - *Kurzbeschreibungen* in Software-Datenbanken, Angeboten und Prospekten
- **Vorgaben** liefern Information darüber, wie ein Artefakt oder der Vorgang seiner Bearbeitung beschaffen sein soll (Spezifikation, Entwicklungsplan, Installationsanleitung). Sie sind natürlich präskriptiv.
- **Explorative Modelle** (Abb. 3) sind transient. Sie werden entwickelt, wenn die Folgen einer möglichen Änderung der Realität beurteilt werden sollen. Die Änderung wird darum versuchsweise im Modell durchgeführt (Modelle im Städtebau).
- **Spiel- und Lehrmodelle** (deskriptiv) treten aus ethischen oder praktischen Gründen an die Stelle des Originals (Modell der menschlichen Anatomie, Flugsimulator). Typisch ist eine partielle äußerliche Ähnlichkeit mit dem Original.
- **Formale** (mathematische) **Modelle** (z.B. die Formeln der Physik und der Chemie) sind ebenfalls deskriptiv, haben aber kaum optische Ähnlichkeit mit den Originalen. Sie eröffnen die Möglichkeit, reale Situationen und Vorgänge formal darzustellen, Hypothesen zu begründen. In anderen, komplexeren Modellen (Flugsimulator) sind sie oft enthalten.

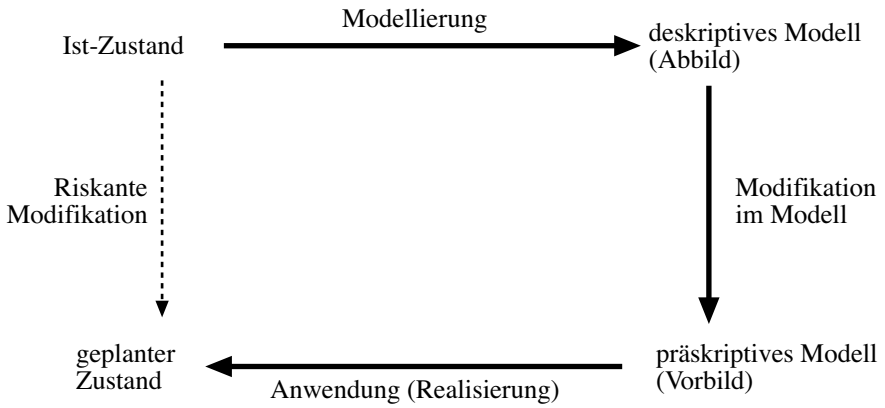

Abb. 3: Anwendung eines explorativen Modells

5 Modelle im Software Engineering

Im Software Engineering begegnen uns Modelle auf verschiedenen Ebenen und unter verschiedenen Bezeichnungen. Einige dieser Modell-Klassen werden nachfolgend diskutiert.
Während die Software-Entwickler *konkrete* Modelle schaffen (siehe unten), befasst sich das *Forschungsgebiet* Software Engineering typisch mit Notationen und Methoden, die zur Entwicklung solcher Modelle dienen. Beispiele sind Endliche Automaten und State Charts, Petri-Netze und Datenflussdiagramme. CASE-Tools sind Werkzeuge zur Erstellung, Speicherung, Prüfung und Bearbeitung von Software-Modellen.

5.1 Präskriptive Modelle für das Software Engineering

Wo im Software Engineering explizit von Modellen gesprochen wird, handelt es sich meist um präskriptive Modelle, beispielsweise

- *Prozessreife-Modelle*: CMM, Bootstrap, SPICE (Jede Stufe ist *ein* Modell.)
- *Ablaufmodelle*: Phasenplan, evolutionäre Entwicklung
- *Prozessmodelle*: Cleanroom, XP
- *Modelle der Informationsflüsse im interessierenden Weltausschnitt:* Datenflussdiagramme (z.B. in SADT)
- *Entwurfsmodelle*: Klassendiagramme
- *Interaktionsmodelle*: Use-Cases, Interaktionsdiagramme
- *Teilkonstruktionsmodelle*: Entwurfsmuster

Die Anforderungsspezifikation *in statu nascendi* ist ein typisches transientes Modell: Nachdem in der Ist-Analyse ein Bild dessen entstanden ist, was vorgefunden wurde, entsteht in der Soll-Analyse ein Bild dessen, was werden soll.

5.2 Die Dokumentenkette als Modell-Kaskade

In einer systematischen Software-Entwicklung nach dem Wasserfall-Modell entstehen nacheinander Dokumente, die Modell-Eigenschaften aufweisen: Die Planungsdokumente (Phasen-, Dokumentations- und Prüfplan) sind offensichtlich präskriptiv bezüglich Organisation usw. Die Spezifikation (das Pflichtenheft) ist janusköpfig, ein deskriptives Modell der Anforderungen, ein präskriptives der Realisierung. Diese Doppelrolle kennzeichnet ihre zentrale Stellung in der Software-Entwicklung. Die Kaskade geht dann über System- und Feinentwurf bis zum Code und zur Ausführung. Testdaten und Benutzungshandbuch sind Modelle der Spezifikation und können diese teilweise ersetzen. Die Metriken sind Modelle der Software oder ihres Entwicklungsprozesses.
Ein hartes Problem des Software Engineerings besteht in der Schwierigkeit, die Dokumentenkette inhaltlich zu erhalten. Wir haben bis heute keine überzeugende Technik, Änderungen des einen Dokuments im anderen nachzuvollziehen (Tracing). Beim Vorwärts-Tracing machen uns die präterierten Attribute Probleme: Der Code ist mehr als die Spezifikation. Beim Rückwärts-Tracing fehlen uns die abundanten: Der Code sagt nichts über die Absichten und Ziele des Kunden aus.

5.3 Die Software als Modell eines Weltausschnitts

> In many systems the machine embodies a *model* or a *simulation* of some part of the world. (...) The purpose of such a model is to provide efficient and convenient access to information about the world. By capturing states and events of the world and using them to build and maintain the model we provide ourselves with a stored information asset that we can exploit later when information is needed but would be harder or more expensive to acquire directly." (M. Jackson in [Ja1995]).

Zu den populären Weisheiten unseres Fachs gehört die Feststellung, dass viele (oder alle) Softwaresysteme das Modell eines Weltausschnitts darstellen. Stimmt das?
Wie im Software Engineering üblich schauen wir zunächst bei den anderen Ingenieuren nach, ob auch sie vor allem Modelle liefern. In der Regel tun sie das nicht, was sie herstellen, sind meist „nur" Werkzeuge. Das Auto modelliert nichts, was unseren Vorfahren bekannt war. Ein Haus ist nicht das Modell des uns fehlenden Fells oder das der Höhle. Die Technik stellt also im allgemeinen keine Modelle zur Verfügung, sondern Hilfsmittel (Werkzeuge im allgemeinsten Sinne), die ihre Besitzer befähigen (sollen), Ziele leichter, billiger, angenehmer zu erreichen, als es ohne die Hilfsmittel möglich wäre. Und inzwischen gibt es sehr viel Technik, die uns völlig neue Ziele eröffnet; denn zu Fuß kommt man nie zum Mond, und auch durch konzentriertes Schauen kann man keine Bakterien sehen. Technik erleichtert und ermöglicht das, ohne Modelle! Das Symbol der Technik, das Rad, ist kein Modell.
Natürlich gilt auch für die Systeme der Softwaretechnik, dass sie Tätigkeiten erleichtern und ermöglichen. Aber wir erklären die Artefakte mit den Begriffen der realen Welt: „Hier sind die Kunden gespeichert, dort werden die Bestellungen registriert, hier werden Vorgänge archiviert, dort Informationen über säumige Zahler eingeholt." Das Rechnersystem als Puppenhaus, in dem unsere Welt nach- oder vorgespielt wird.

Woran liegt diese Besonderheit der Informatik-Systeme?
Zwei Gründe stehen hinter der starken Modell-Betonung in der Software:

- Rechner sind von Beginn an Modelle des erinnernden, denkenden und entscheidenden Menschen. Aus diesem Grunde eignen sie sich besonders gut, um menschliche Tätigkeiten und Organisationen, z.B. das Schachspielen, eine Auskunftsstelle oder ein Warenlager, zu simulieren. Dieser Gedanke ist viel älter als die Informatik, er taucht schon vor Jahrhunderten in Märchen auf, auch in Betrügereien (Schach-Automat).
- Wenn wir aus den Informatik-Komponenten etwas Neues schaffen, dann reicht unsere Fantasie nicht aus, um etwas *wirklich* Neues zu ersinnen, wir richten uns nach dem Vorbild von Artefakten, die uns bekannt sind. Gerade weil uns in der Welt der Rechner jegliche Vorstellungskraft fehlt, bleiben wir ganz in der Welt unserer Anschauungen. In der objektorientierten Programmierung wurde dieses Prinzip zur Leitidee erhoben, aber es erscheint bereits früher, am klarsten in JSD (Jackson System Development [Ca1986]): Nachdem in der Modellierungsphase der interessierende Weltausschnitt modelliert wurde, werden die Modelle in der Vernetzungsphase verknüpft. Dabei können Funktionen wie Merken, Suchen und Entscheiden an Algorithmen delegiert werden, so dass ein Software-System entsteht.

Die strukturähnliche Modellierung der Welt in der Software hat einen wichtigen Vorteil, auf den ebenfalls Jackson schon vor einem Vierteljahrhundert hingewiesen hat: Da sich die reale Welt verändert, muss auch die Software verändert werden. Diese Änderung ist um so einfacher, je ähnlicher sich beide strukturell sind. Wenn beispielsweise in der realen Welt eine Abteilung eines Unternehmens verkauft wird, dann kann die Unternehmenssoftware leicht angepasst werden, indem das dieser Abteilung entsprechende Subsystem entfernt wird. Das Softwaresystem ist um so wartungsfreundlicher (im Hinblick auf die adaptive Wartung), je ähnlicher es strukturell der Realität ist.
Andererseits bedeutet die Beschränkung auf Modelle eine drastische Limitierung unserer Möglichkeiten. Die anderen Ingenieure waren erst dann in der Lage, wirklich revolutionäre Erfindungen zu machen, als sie sich von den Modellen gelöst hatten. Rad und Radio, Hochofen und Hochfrequenz waren absolut neu. In einigen Mikroproblemen hat es auch die Informatik geschafft: Quicksort und Heapsort, zwei überraschend leistungsfähige Sortieralgorithmen, sind nicht (wie Selection Sort und Insertion Sort) dem menschlichen Vorgehen abgeschaut. Die schnelle Fourier-Analyse erscheint dem uneingeweihten Betrachter als pure Zauberei, nicht nachvollziehbar, aber rasend schnell. In den genannten Fällen war die mathematische Formulierung des Problems entscheidend: Formale Manipulation überholt den „gesunden Ingenieur-Verstand“. Die Frage, ob dieser Ansatz irgendwann auch für komplexe Probleme anwendbar wird, lässt sich auf absehbare Zeit nicht seriös beantworten.
Bei den allerersten Freileitungen zur Stromübertragung hat man scharfe Richtungsänderungen vermieden, um die Verluste gering zu halten. Das naheliegende Modell des Transportsystems war irreführend. Es wurde verworfen, als die Leute Theorien entwickelt oder kennengelernt hatten, die zwar weniger anschaulich, aber sehr zuverlässig die elektrischen Vorgänge erklärten. So weit sind wir bei der Software-Entwicklung noch nicht.

5.4 Benutzungs- und Architektur-Metaphern

> Each XP software project is guided by a single overarching metaphor. (...) Sometimes the metaphor needs a little explanation, like saying the computer should appear as a desktop, or that pension calculation is like a spreadsheet. (...) As development proceeds and the metaphor matures, the whole team will find new inspiration from examining the metaphor. (...) The metaphor in XP replaces much of what other people call “architecture.” Kent Beck [Be1999], p. 56

Die Metapher spielt in XP (Extreme Programming) eine zentrale Rolle. Beck scheint überzeugt, dass die Architektur und die Vorstellung der Benutzer kongruent sind: Was dem Benutzer als Schreibtisch erscheint, ist auch für den Entwickler ein Schreibtisch.
Jeder Programmierer weiß, dass das nicht stimmt. Die Mülleimer-Metapher auf dem Bildschirm ist keine Architektur, sondern eine Verständnis-Hilfe für den Benutzer. Wir legen ja auch gar keinen Wert darauf, dass Dokumente, die wir zurückholen, verschmutzt, zerknüllt und ungeordnet sind. Und natürlich erwarten wir nicht, dass wirklich in einen speziellen Speicher bewegt wird, was wir wegwerfen.
Sicher ist, dass der *Benutzer* eine Metapher braucht, um mit dem System zurechtzukom-

men. Diese Metapher sollte konsistent umgesetzt sein, also nicht Löcher aufweisen wie eine beschädigte Theaterkulisse. Wer in einer Hochsprache programmiert, will keine Meldungen des Systems mit Fehlercodes und Adressen der Hardware-Ebene erhalten, denn die gibt es in seiner Metapher nicht. Wer eine Mail verschickt hat und zurückbekommt, weil sie nicht zugestellt werden konnte, verlässt sich darauf, dass sie tatsächlich nicht angekommen ist (was oft nicht stimmt, so dass wir die gleiche Mail vielfach erhalten). Wer einen Text auf dem Bildschirm formatiert, ist verblüfft, wenn sein Werk nicht genau gleich aus dem Drucker läuft.
Ob die Software-Architektur eine Metapher braucht, muss diskutiert werden. Das Vorbild des Bauwesens deutet eher darauf hin, dass solche Metaphern schädlich wären. Denn in der Konstruktion können nicht einfach kleine Objekte zu großen aufgeblasen werden, eine Talbrücke ist nicht nach der Metapher eines Holzstegs konstruiert. WYSIWYG ist eine Metapher für den Benutzer, nicht für den Architekten. Für ihn ist die Metapher Teil der Anforderungen, seine Architektur muss sie herstellen. Wer eine elektronische Lenkung („drive by wire“) realisiert, muss – wenigstens in den nächsten dreißig Jahren – dafür sorgen, dass diese ganz ähnlich wie eine konventionelle reagiert. Er sollte aber auf keinen Fall die Architektur der konventionellen übernehmen.
Kurz: Wer einen guten Weihnachtsmann abgeben will, sollte selbst besser nicht an den Weihnachtsmann glauben.

6 Risiken beim Umgang mit Modellen

Beim Umgang mit Modellen gehen zwei Dinge immer wieder schief:

- Der Anwender des Modells hält das Modell für das Original.
- Aus dem Modell werden Schlüsse gezogen, die nicht gezogen werden dürfen, weil die relevanten Attribute präteriert, also nicht abgebildet wurden.

Beide Effekte sind oft miteinander vermischt, so dass sie hier nicht getrennt werden. Wer das Modell als Original anerkennt, hat die Existenz präterierter Attribute vergessen.
Modelle sind keine Originale, können und sollen aber entsprechend dem pragmatischen Merkmal die Originale unter bestimmten Fragestellungen ersetzen. Gute Modelle kommen dem Original so nahe, dass die Unterscheidung vergessen wird. Dann ist die Falle zugeschnappt.

6.1 Das Modell als Weltbild

Die Welt ist nun mal objektorientiert.
Ein Informatik-Professor, 1992

Besonders oft findet man dieses Problem als *deformation professionelle*, d.h. als berufsbedingte Verzerrung der Sicht. Der Mediziner teilt die Patienten nach ihren Krankheiten ein; schlimm für ihn, wenn er in der Freizeit nicht davon loskommt. Der Logiker lehrt seine Studenten das *tertium no datur*, die einfache, aber ganz und gar weltferne zweiwertige Logik, nach deren Regeln die Erziehung seiner Kinder fast sicher scheitern wird. Und der Software-Entwickler ist so begeistert von der objektorientierten Programmie-

rung, dass er allen Ernstes die reale Welt zu einer Instanz seines objektorientierten Klassen-Weltbildes erklärt. Dieses Weltbild wird dann dogmatisch vertreten und verteidigt. Das ist unerfreulich und oft lächerlich. Gerade jede universitäre Lehre muss immer wieder klar machen, was die Errungenschaft der Moderne gegenüber dem Mittelalter ausmacht: Nur die Beobachtung, möglichst oft und von möglichst vielen Menschen, ist ein geeignetes Verfahren, um der Realität auf die Spur zu kommen. Wer einfach etwas postuliert, weil es so schön in seine Modelle, das heißt in sein Weltbild passt, agiert nicht in der Wissenschaft, sondern in der Religion.

6.2 Beispiele aus der Praxis

In der Praxis ist dieser Effekt meist weniger auffällig und darum schädlicher. Wenn ein Software-Analytiker eine Spezifikation verfasst, erzeugt er ein Modell der Anforderungen, die er dem Kunden abgerungen hat. Oft waren seine Gesprächspartner weder repräsentativ noch kompetent, sie waren auch nicht in der Lage zu prüfen, ob die Spezifikation ihre Vorstellungen korrekt spiegelt, haben aber, um die Sache abzuschließen, unterschrieben. Wenn später der Entwickler glaubt, die Anforderungen des Kunden zu sehen, irrt er: Er kennt nur das *Bild*, das sich sein Kollege von den Kundenanforderungen gemacht hat.
Eine besondere Magie haben Wörter: Indem man die Zahl der Verzweigungen im Programm als Komplexitätsmaß bezeichnet, bringt man viele Menschen dazu, diese Zahl als Komplexität zu akzeptieren; ähnlich ist es bei der Produktivität.
Testergebnisse werden in der Praxis besserem Wissen zum Trotz als Modelle der Qualität betrachtet, die Tatsache, dass n Fehler gefunden wurden, wird so interpretiert, dass (vor der Korrektur) n Fehler vorhanden waren. Dieser banale Fehler ist weit verbreitet.
Damit verbunden ist ein weiter reichendes Problem: Während Inkorrektheiten wenigstens mit einer minimalen Wahrscheinlichkeit im Testresultat auffallen, ist das bei unleserlichem Code, fehlender Portabilität und anderen, ähnlichen Mängeln sicher nicht der Fall. Eine Qualitätssicherung, die sich fast oder ganz ausschließlich auf den Test stützt (und das ist in weiten Teilen der Wirtschaft der Fall), muss unvermeidlich ein falsches Bild der Qualität entwickeln und damit den Prozess höchst unglücklich beeinflussen.

Die Aussagen dieses Abschnitts können auch konstruktiv formuliert werden: Wer Verbesserungen im Software Engineering anstrebt, muss vorrangig Modelle entwickeln, die die relevanten Eigenschaften der Software sinnvoll wiedergeben. Er muss zweitens dafür sorgen, dass diese Modelle anerkannt und eingesetzt werden. Und er muss schließlich sicherstellen, dass erfolgreiche Arbeit im Sinne dieser Modelle sichtbare Vorteile hat.
Noch eine Stufe konkreter:

- Wir brauchen Metriken für die Software-Qualität, die anders als die von Halstead und McCabe zu sinnvollen Aussagen führen.
- Wir müssen diese Metriken in möglichst weiten Bereichen einsetzen.
- Wir müssen dafür sorgen, dass nicht der Desperado befördert wird, der ein Projekt (unter schweren Verlusten an Funktionalität und Qualität) „gerettet“ hat, nachdem er vorher Mängel und Risiken übersehen hatte, sondern seine Kollegin, deren Projekt keiner Rettung bedurfte, weil es stets und stets erkennbar im grünen Bereich lag.

7 Erfahrungen mit einem großen Modell

Das SESAM-System (Software-Engineering-Simulation durch animierte Modelle) wird seit etwa 1990 in Stuttgart entwickelt. Viele der Beobachtungen, die in diesem Beitrag wiedergegeben sind, wurden in diesem Projekt gemacht.
SESAM soll hier nur soweit vorgestellt werden, wie es für einige Aspekte der Modelle notwendig ist. Wesentlich mehr Information über SESAM ist verfügbar [Ge2002].
SESAM ist ein umfangreiches Software-System, ein generischer Simulator, der durch ein ähnlich umfangreiches Modell zu einem konkreten Simulator wird. Dieser erlaubt es einem Benutzer (dem „Spieler“), die Rolle eines Software-Projektleiters zu übernehmen und im Zeitraffer ein Projekt durchzuführen (Abb. 4). Am Ende bekommt er wie bei anderen Computer-Spielen eine Bewertung seiner Leistung.
SESAM ist unzulänglich. Eine ganze Reihe von Effekten ist aus technischen Gründen nicht oder unbefriedigend dargestellt, und wir müssen die Spieler auf Schwachpunkte auf-

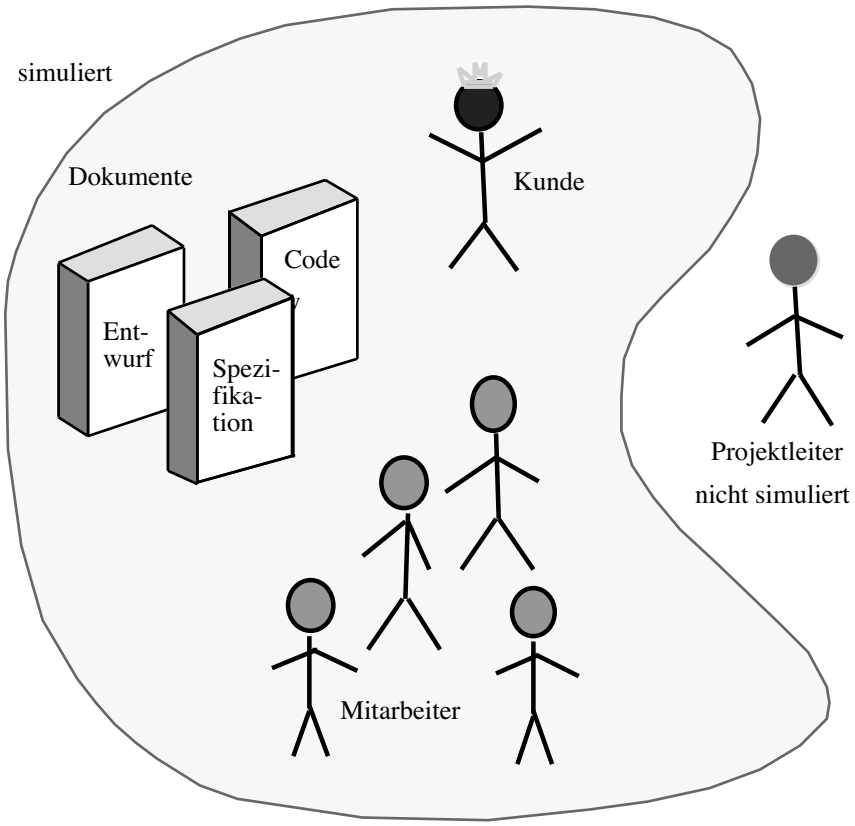

Abb. 4: Open-Loop-Simulation in SESAM

merksam machen, bevor wir sie auf das System loslassen. Das ist aber kein Problem, im Gegenteil: Verfremdungseffekte, die den Spielern immer wieder zeigen, dass sie nicht in der Realität sind, müssten systematisch eingebaut werden, wenn nicht die primitive Benutzungsschnittstelle und die Ausführung im Zeitraffer die Spielsituation jederzeit klarstellten.
Trotzdem lassen sich die Studierenden auf das Modell ein, entwickeln sportlichen Ehrgeiz und Konkurrenzdenken, Motivationen, die sehr nützlich sind, um Kenntnisse zu vermitteln, die ohne solche Hilfen sehr trocken und theoretisch erscheinen. SESAM erweist sich tatsächlich auch in einem negativen Sinne als realitätsnah: So wenig, wie ein Projektleiter durch ein gescheitertes Projekt klug wird, so wenig lernt auch ein Spieler, der sein virtuelles Projekt in den Sand gesetzt hat [Dr2000]. Wir müssen also den Umgang mit dem Modell ergänzen durch traditionellen Unterricht, am besten anhand der Analysen, die aufgrund der Spielverläufe angefertigt werden [Ma2002].

7.1 Schwierigkeiten bei der Entwicklung der SESAM-Modelle

SESAM verwendet zwei verschiedene Modelle, beide in zwei verschiedenen Repräsentationen (Abb. 5):

- Das Modell des simulierten Projekts, das sog. *Zustandsmodell*, wird vom Tutor initialisiert. Eine interne Darstellung dieses Modells, ein attributierter Graph, wird während des Spiels ständig verwendet und verändert. Davon sind nicht nur Attribut-

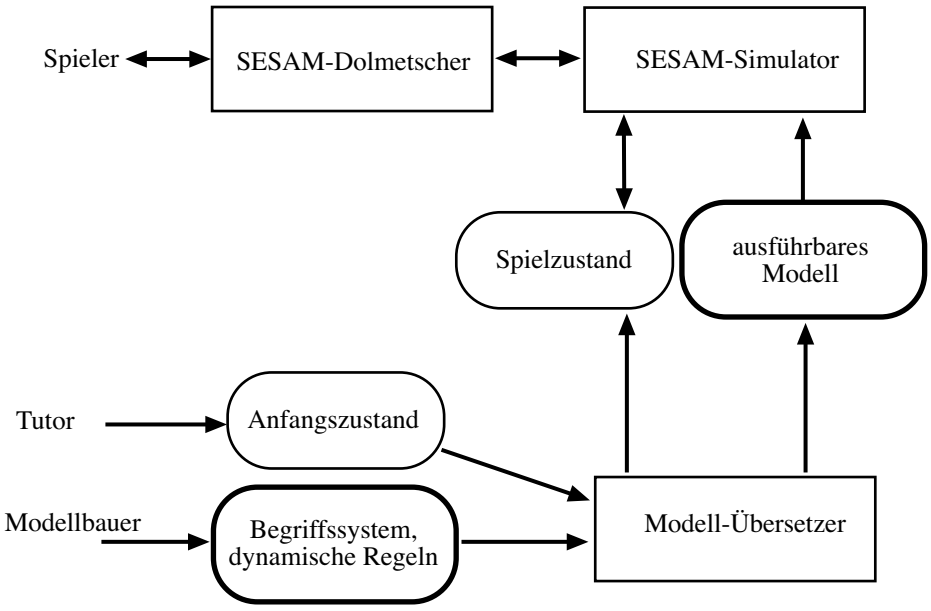

Abb. 5: Vereinfachte SESAM-Architektur;
Modelle in abgerundeten Kästen; Zustandsmodelle mager, Ablaufmodelle fett

werte (etwa der Umfang oder die Fehlerzahl eines Dokuments), sondern auch Strukturen (die Zuordnung eines Mitarbeiters zu einer bestimmten Teilaufgabe) betroffen.

- Das Modell der Mechanismen, die in den Projekten wirksam sind, das sog. *Ablaufmodell*, wird vom Modellbauer geschaffen. Es zerfällt weiter in das *statische Modell* der Begriffe (Entitäten und Relationen) und das *dynamische Modell* der Veränderungen (Regeln). Dem Zustandsmodell liegt das statische Modell zugrunde. Das dynamische Modell repräsentiert quasi die innere Logik des Softwareprojekt-Managements. Dieses Modell wird aus Effizienzgründen in eine interne Form überführt und dann im Zuge der Simulation laufend interpretiert.

Während die Erstellung eines Zustandsmodells eine relativ einfache Sache ist, braucht der Modellbauer für ein Ablaufmodell mindestens Monate. Darum meinen wir immer dieses, wenn wir vom *Modell* sprechen.

Das Ablaufmodell repräsentiert eine empirisch fundierte Theorie der Projektabläufe. Seine Entwicklung ist so schwierig, weil empirische Daten in der Literatur sehr rar sind und in der Praxis nur mit extremem Aufwand gesammelt werden können. Je mehr Elemente spekulativ eingesetzt werden, um so wichtiger wird die Validierung des Modells, die auf die selben Probleme stößt. Für das bisher reifste Modell [Dr2000] wurde die Validierung mit sehr großem Aufwand geleistet. Die neueren Modelle sind aus diesem sog. QA-Modell (für *quality assurance*, dem Schwerpunkt des Modells) entstanden.

Weiterhin ist sich niemand sicher, welcher Weltausschnitt modelliert werden muss, um den richtigen Kompromiss aus Einfachheit und Realitätsnähe zu treffen. Kann man bei simulierten Mitarbeitern auf deren Privatleben einfach verzichten? Und schließlich zeigen Erfahrungen, dass die Komplexität der Modelle nicht nur durch die Rechenzeit, sondern auch durch das Verständnis der Spieler begrenzt ist.

Der letzte Punkt hat zu einer Revision der Ziele geführt: Erschien es früher attraktiv, den Spielern eine möglichst vertrackte Aufgabenstellung zu geben, die sie zwingt, alle ihre Fähigkeiten zu nutzen, halten wir heute eine recht simple Ausgangslage für vorteilhaft. Es schadet auch nichts, den zuvor erfolglosen Spielern genau zu erklären, was sie tun sollten, um erfolgreich zu sein. Diese Aussage kann man meines Erachtens verallgemeinern: Wir müssen in der Lehre dafür sorgen, dass unseren Studierenden lösbare Aufgaben gestellt sind, dass sie diese auch praktisch lösen und dass ihnen Mängel ihrer Lösungen erklärt werden. Wenn wir davon überzeugt sind, dass gutes Software Engineering besser ist als schlechtes Software Engineering, dann müssen wir das auch vorleben und nachvollziehbar machen.

7.2 Deskriptive versus präskriptive Modelle in der Forschung

Ordnen wir das SESAM-Modell in die anfangs entwickelte Taxonomie ein: Es ist ein Spiel, also ein deskriptives Modell der Realität. So harmlos diese Feststellung klingt, so unüberschaubar waren die Folgen für die Entwickler: Ein deskriptives Modell muss den modellierten Weltausschnitt möglichst unter allen Umständen möglichst perfekt darstellen. Kommt der Spieler auf die überraschende Idee, seine Mitarbeiter mit dem Test zu beauftragen, bevor die Entwicklung begonnen hat (XP), dann wird in der Realität dadurch nicht die Welt einstürzen. Also muss auch SESAM leidlich sinnvoll reagieren. Wird ein

Mitarbeiter krank, der eigentlich gerade kündigen wollte, so nimmt er vermutlich erst die Krankheitspause mit, bevor er geht. SESAM muss das „wissen“. Die Menge der Möglichkeiten, die sich damit auftun, ist riesig groß. Das macht die Entwicklung eines Modells für SESAM extrem aufwändig.

SESAM ist ein Exot unter den Forschungsprojekten: Die allermeisten anderen Arbeiten befassen sich nicht mit deskriptiven, sondern mit präskriptiven Modellen. Drastisch gesagt: „Es interessiert uns nicht, wie es ist, wir sagen euch stattdessen, wie es sein sollte.“ Das wäre sinnvoll, wenn die Ideen umgesetzt würden, so dass der eine gute unter den 99 untauglichen Vorschlägen zum Vorschein käme. Ganz überwiegend geschieht das aber nicht. Vielmehr sind die Arbeiten schon so angelegt, dass eine Umsetzung praktisch ausgeschlossen ist. So entstehen Dissertationen und Doktortitel, sonst nichts.

Die Verfasser sind darum keine Ignoranten, im Gegenteil: Sie schätzen die Lage und ihre Möglichkeiten richtig ein. Präskriptive Modelle sind leicht zu bauen, und darum sind sie seit Jahrzehnten bei den Forschern beliebt. Waren es früher neue Programmiersprachen, die man sich ausdachte, so traten später neue Werkzeuge, wieder später neue Prozessmodelle an diese Stelle. Die allermeisten dieser Sprachen und Prozessmodelle wurden nie eingesetzt, und kaum ein Werkzeug wurde implementiert. Darum blieb auch unbekannt, wie tauglich diese Ideen waren.

Natürlich haben die Doktoranden dabei eine Menge gelernt, und das ist wichtig. Aber es bringt das Fach nicht weiter. Es wäre sehr schön, wenn das Gebiet „Software Engineering“ durch jede Dissertation um ein Epsilon schöner, größer oder sicherer würde. Dazu tragen Arbeiten bei, die die Landschaft, in der wir leben, erkunden und verändern, nicht solche, die neue Landschaften *postulieren*. Was fehlt, sind solide empirische Arbeiten, die uns deskriptive Modelle der Realität geben. Damit solche entstehen, muss das Modell der Forschung verändert werden: Wer etwas Neues ausdenkt, muss verpflichtet sein, das Neue ehrlich zu erproben und seine Resultate zu präsentieren. Das sollte selbstverständlich sein. Wer nichts Neues ausdenkt, sondern „nur“ bislang Vermutetes empirisch überprüft, leistet eine unspektakuläre, aber höchst wertvolle Arbeit, die mehr Achtung verdient.

Literaturverzeichnis

[Be1999] Beck, K.: Extreme Programming. Addison-Wesley, Reading, Mass., 1999.

[Ca1986] Cameron, J.R. (1986): An overview of JSD. IEEE Trans. on Softw. Eng., SE-12, 2, 222-240.

[Dr2000] Drappa, A.: Quantitative Modellierung von Softwareprojekten. Dissertation, Universität Stuttgart. Shaker-Verlag Aachen, 2000.

[Ge2002] Georgescu, A.: Web-Seiten zu SESAM unter http://www.informatik.uni-stuttgart.de/ifi/se/research/sesam

[Ja1995] Jackson, M.: The world and the machine. Proc. of the 17th ICSE, Seattle, ACM 1995, 283-292.

[Ma2002] Mandl-Striegnitz, P.: Dissertation, Universität Stuttgart, erscheint 2002.

[St1993] Stachowiak, H.: Allgemeine Modelltheorie. Springer-Verlag, Wien etc., 1973.