Code Complete

Deutsche Ausgabe

Bearbeitet von
Steve McConnell

1. Auflage 2005. Buch. 940 S. Hardcover
ISBN 978 3 86063 593 3
Gewicht: 1631 g

Kapitel 2

Metaphern als Hilfsmittel für die Softwareentwicklung

cc2e.com/0278

Inhalt

Verwandte Themen

- Heuristik beim Entwurf: »Der Entwurf als heuristischer Prozess« in Abschnitt 5.1

Die Sprache der Informatik ist ausgesprochen farbig und bildhaft. Wo sonst treffen Sie in einem sterilen, klimatisierten Raum auf Viren, trojanische Pferde, Würmer, Briefbomben, Abstürze, fatale Fehler oder – höchst interessant – auf Twisted Sex Changers?

[Leider hat die Übertragung ins Deutsche dieser heiteren und kräftigen Sprache vieles von ihrem Glanz genommen. Hinter einem deutschen Ohr entstehen zum Beispiel bei Erwähnung des Wortes »Debugger« je nach Vorkenntnis dessen Besitzers Assoziationen wie: Gurumystik, »Heißt es nicht: *Der* Bagger?« oder eben »Werkzeug, mit dem sich logische Fehler kompilierter Programme finden lassen«. Diese Fehler heißen in der Computerwelt *Bugs* – in Anlehnung an die lästigen kleinen Wanzen, Kakerlaken und Flöhe, denen ja ansonsten auch nur schwer beizukommen ist. Am besten, man ruft dazu einen Fachmann, nämlich den Kammerjäger oder Ent-Wanzer, sprich: den De-Bugger! Und schon könnte das fremd und bedrohlich im Deutschen herumstehende Wort Debugger einen freundlichen Stallgeruch haben, könnte vom Sockel der Computermystifizierung steigen, wenn sich nicht vor Urzeiten ein Übersetzer anders entschieden hätte. Oder kommen wir auf die bereits erwähnten Twisted Sex Changers zurück – damit sind kleine Adapter gemeint, die Computerstecker in Computerbuchsen (und umgekehrt) verwandeln. Der subversive Spaß, in diesem nüchtern-technischen Zusammenhang sofort an Tim Currys glanzvollen Auftritt in der Rocky Horror Picture Show denken zu müssen, geht im Deutschen leider unter. *Anm. d. Übers.*]

Diese sehr plastischen Metaphern beschreiben Softwarephänomene und sollen die Hintergründe der Softwareentwicklung erhellen. Genauso anschauliche Metaphern werden auch für die Beschreibung anderer Vorgänge verwendet. Sie können damit Ihr Verständnis für den Prozess der Softwareentwicklung verbessern.

Die Diskussion von Metaphern in diesem Kapitel ist nicht unbedingt erforderlich, um dieses Buch zu verstehen. Wenn Sie jedoch ein klareres Bild von der

Softwareentwicklung bekommen wollen, dann sollten Sie die folgenden Seiten nicht überspringen.

2.1 Die Bedeutung von Metaphern

Wichtige Entwicklungen entstehen oft aus Analogien. Indem Sie einen Fakt, über den Sie nur wenig wissen, mit einem anderen, gut bekannten Sachverhalt vergleichen, können Sie Schlussfolgerungen ziehen, die zu einem besseren Verständnis des ersteren Fakts führen. Man nennt dies auch Modellierung.

Die Geschichte der Naturwissenschaften kennt viele Entdeckungen, die erst durch Metaphern möglich wurden. So erschien dem Chemiker Kekulé im Traum eine Schlange, die sich selbst in den Schwanz biss. Am nächsten Morgen war ihm klar, dass sich die molekulare Struktur von Benzol mit einem solchen Ring beschreiben lässt, was er experimentell nachweisen konnte (Barbour 1966).

Die kinetische Gastheorie wurde in Analogie zum Billardspiel entwickelt. Man ging davon aus, dass Gasmoleküle genau wie Billardkugeln eine Masse besitzen und elastisch zusammenstoßen. Dieser Ansatz erwies sich als so gelungen, dass viele weitere Theoreme von ihm abgeleitet werden konnten.

Die Wellentheorie des Lichts wurde größtenteils anhand der Ähnlichkeiten von Licht und Schall entwickelt. Beide haben eine Amplitude (Helligkeit, Lautstärke), eine Frequenz (Farbe, Tonhöhe) und noch einige andere gemeinsame Eigenschaften. Die Ähnlichkeiten zwischen Licht und Schall waren so offensichtlich, dass die Wissenschaftler lange Zeit nach einem Medium suchten, das die Lichtwellen weiterleitet – Schallwellen benötigen ja auch ein Medium, etwa Luft. Sie gaben diesem Medium sogar einen Namen, nämlich »Äther«, konnten es jedoch nie nachweisen. In diesem Fall hatten die sonst so fruchtbaren Analogiebetrachtungen in die Irre geführt.

Allgemein kann gesagt werden, dass solche Modelle anschaulich sind und komplexe Zusammenhänge als Ganzes repräsentieren. Modelle führen auf Eigenschaften, Beziehungen und weitere zu untersuchende Gebiete, was jedoch auch schief gehen kann. Als die Wissenschaftler nach dem Äther suchten, hatten sie ihr Modell (beziehungsweise die Analogiebetrachtungen) überstrapaziert.

Natürlich gibt es gute und schlechte Metaphern. Eine gute Metapher ist vor allem einfach, verträgt sich mit anderen, bereits etablierten Metaphern und erklärt viele experimentell nachgewiesene Fakten und andere beobachtete Phänomene.

Nehmen wir als Beispiel einen schweren Stein, der an einem Faden hin und her schwingt. Vor Galileo ging die aristotelische Betrachtungsweise davon aus, dass ein Körper sich natürlich von einer höheren Position zu einer niedrigeren Position bewegt, wo er zur Ruhe kommt. Für die darauf folgende Aufwärtsbewegung war keine Erklärung zu Hand, sodass dieser Vorgang unter aristotelischer Sicht als »kompliziertes Fallen« bezeichnet wurde, das letzten Endes ja doch zum Stillstand in der niedrigsten Position führt. Galileo hingegen sah ein Pendel, das immer wieder dieselbe Bahn beschreibt und dessen Ausschlag sich nur durch die Reibungsverluste verringert.

Beide Modelle führen zu verschiedenen Schlussfolgerungen. Der aristotelische Wissenschaftler, der den hin und her schwingenden Stein vor allem als ein fallendes Objekt sah, maß das Gewicht des Steins, seine Höhe und die Zeit, nach der das Pendel stillstand. Für Galileos Modell waren andere Faktoren wichtiger: wiederum das Gewicht, nun aber auch die Fadenlänge des Pendels, der Ausschlagwinkel und die Zeit pro Schwingung. So konnte Galileo Naturgesetze entdecken, die der aristotelischen Betrachtung verborgen blieben, da deren Modell sich auf andere Phänomene konzentrierte und in eine andere Richtung führte.

Genau wie in den Naturwissenschaften können Metaphern auch Phänomene der Softwareentwicklung erhellen. Charles Bachmann beschrieb 1973 in seiner Rede anlässlich der Verleihung des Turing-Preises den Übergang vom geozentrischen zum heliozentrischen Weltbild. Die Auffassung des Ptolemäus, dass die Erde der Mittelpunkt des Sonnensystems sei, blieb 1400 Jahre lang unangefochten. 1543 behauptete Kopernikus, dass sich nicht die Sonne um die Erde drehe, sondern die Erde um die Sonne. Dieser Wechsel der Denkmodelle führte zur Entdeckung weiterer Planeten und zu der Erkenntnis, dass der Mond kein Planet, sondern ein Satellit der Erde ist, und letztendlich zu einer Neubewertung des Platzes der Menschheit im Universum.

Metaphern haben unschätzbare Vorteile. Ihr Verhalten ist vorhersehbar und jedermann verständlich. Unnötige Erklärungen und Missverständnisse fallen weg. Fakten lassen sich schneller erlernen und lehren. Metaphern trennen Wesentliches von Unwesentlichem und führen so zu abstrakten Konzepten, die ein Denken auf höherer Ebene ermöglichen und Fehler auf niedrigerer Ebene vermeiden.

Fernando J. Corbató

Bachmann verglich diesen Paradigmenwechsel mit den Veränderungen, die in den frühen 1970er Jahren in der Computerprogrammierung stattfanden. Zu diesem Zeitpunkt, 1973, ging die Datenverarbeitung von einer computerzentrierten zu einer datenbankzentrierten Sicht über. Bachmann wies darauf hin, dass die Altvorderen der EDV Daten als eine Kette von Lochkarten ansahen, die durch den Computer flossen – das computerzentrierte Weltbild. Nunmehr konzentrierte man sich auf eine Ansammlung von Daten, die eben durch einen Computer bearbeitet wurden – der Mittelpunkt des Weltbilds war nicht mehr der Computer, sondern die Datenbank.

Heute fiele es schwer sich vorzustellen, dass sich die Sonne um die Erde bewegt. Ebenso würde die Idee, dass Daten aus einer Kette von Lochkarten bestünden, auf Befremden stoßen. Nachdem eine alte Theorie über Bord geworfen wurde, ist es kaum vorstellbar, dass irgendjemand einmal an sie geglaubt hat. Umgekehrt gilt natürlich, dass sich die Verfechter einer alten Theorie bei Erscheinen einer neuen Theorie mit Klauen und Zähnen gegen solche »absurden Hirngespinste« wehren.

Die geozentrische Weltsicht der Astronomen, die auch nach Erscheinen der neuen Theorie an ihr festhielten, erwies sich mehr und mehr als hinderlich. Ebenso behinderte die computerzentrierte Sichtweise ihre Anhänger immer stärker, je mehr sich die datenbankzentrierte Theorie durchsetzte.

Man sollte nicht der Versuchung erliegen, die Kraft von Metaphern zu trivialisieren, also zu sagen: »Klar, die neue Theorie ist viel nützlicher. Die alte Theorie war falsch!«. Dies ist eine zwar verständliche, doch zu schlichte Reaktion. Die Geschichte der Wissenschaft besteht nicht aus einer Reihe von Momenten, an denen von einer »falschen« Theorie auf eine »richtige« Theorie umgeschaltet wurde, sondern aus Übergängen von »schlechteren« Theorien zu »besseren«, das

heißt zu Theorien, die mehr Phänomene erklären können, die aber andererseits bestimmte Bereiche nicht mehr so klar und verständlich abbilden.

Tatsächlich behaupten sich einige Modelle, die eindeutig durch neuere Theorien entkräftet sind, auch heutzutage. Auch nachdem sich herausgestellt hat, dass die newtonsche, klassische Physik eine Untermenge der einsteinschen Theorie ist, benutzen Ingenieure für die meisten Aufgaben noch immer das newtonsche Formelgebäude.

Softwareentwicklung ist eine im Vergleich zu anderen Wissenschaften junge Disziplin und verfügt im Gegensatz zur newtonschen Physik über keine unerschütterlichen Standardmetaphern. Vielmehr tummeln sich hier viele Metaphern, die noch nicht völlig ausgereift sind, die sich ergänzen und auch widersprechen. Je besser Sie sich in diesen Metaphern zurechtfinden, desto besser verstehen Sie die Softwareentwicklung.

2.2 Das Verwenden von Softwaremetaphern


WICHTIG

Sie sollten eine Softwaremetapher eher als einen Suchscheinwerfer denn als eine Straßenkarte betrachten: Sie sagt Ihnen nicht, wo die Antwort liegt, sondern hilft Ihnen bei der Suchen danach. Eine Metapher ist also weniger ein Algorithmus, sondern eher eine heuristische Anleitung.

Bei einem Algorithmus handelt es sich um einen Satz festgelegter und kommentierter Anleitungen für die Ausführung einer bestimmten Aufgabe. Ein Algorithmus ist vorhersehbar, deterministisch und unveränderlich. Er sagt Ihnen, wie Sie von A nach B gelangen – Abstecher nach C, D, E, eine Kaffeepause oder eine verträumte Viertelstunde im Garten sind nicht vorgesehen.

Eine heuristische Technik hilft Ihnen, eine Antwort zu suchen. Die Ergebnisse sind nicht vorhersehbar, da Sie zwar wissen, wie Sie vorgehen müssen, nicht jedoch, was Sie erwartet. Sie erfahren nicht, wie Sie von A nach B gelangen, ja vielleicht nicht einmal, wo die Punkte A und B überhaupt liegen. Eine heuristische Technik ist sozusagen ein Algorithmus im Clownskostüm: Sie ist kaum vorhersehbar, macht mehr Spaß, und es gibt keine zehnjährige Garantie gegen Durchrosten.

So könnte ein Algorithmus aussehen, um zu einem bestimmten Haus zu fahren: Highway 167 südwärts in Richtung Puyallup. Ausfahrt South Hill Mall. 4,5 Meilen bergan. An der Ampel beim Supermarkt rechts abbiegen, dann die erste Querstraße nach links. Fahren Sie in die Einfahrt des großen, dunklen Hauses linker Hand, 714 North Cedar.

Querverweis Details zur Verwendung von Heuristik beim Softwareentwurf finden Sie in »Der Entwurf als heuristischer Prozess« in Abschnitt 5.1.

Und so sieht ein adäquates heuristisches Verfahren aus: Nehmen Sie den letzten Brief, den wir Ihnen geschickt haben, und fahren Sie zu der Stadt, die auf dem Absender vermerkt ist. In der Stadt fragen Sie einfach jemanden nach uns. Jeder kennt uns hier, man wird Ihnen sicher gern helfen. Sollte das nicht klappen, so rufen Sie uns einfach von einer Telefonzelle aus an, wir holen Sie dann ab.

Die Übergänge von Algorithmen zu heuristischen Verfahren sind fließend, beide Begriffe überschneiden sich. In diesem Buch sollen beide Ausdrücke dadurch unterschieden werden, wie direkt beziehungsweise indirekt sie sind: Ein Algo-

rithmus gibt Ihnen direkte Anweisungen. Ein heuristisches Verfahren sagt Ihnen, wie Sie die Anweisungen auf eigene Faust entdecken können oder zumindest, wo Sie sie finden.

Es wäre sicherlich schön, wenn Sie alle Ihre Programmieraufgaben mit Hilfe von Algorithmen lösen könnten. Das Programmieren fiele leichter und die Ergebnisse wären eher vorhersehbar. Die Wissenschaft vom Programmieren hat jedoch noch nicht dieses Stadium der Reife erreicht und wird dies vielleicht auch niemals tun. Der anspruchsvollste Teil einer Programmieraufgabe ist das Erstellen eines Konzepts für das Problem. Viele Programmierfehler sind auf das Konzept zurückzuführen. Da jedem Programm ein eigenes, einzigartiges Konzept zugrunde liegt, ist es schwierig oder gar unmöglich, einen allgemein gültigen Satz von Anleitungen zu entwickeln, die in jedem Fall zu einer Lösung führen. Es ist daher sowohl wichtig zu wissen, wie Probleme allgemein angepackt werden, als auch bestimmte Lösungen für bestimmte Probleme zu kennen.

Wie helfen Ihnen hierbei die Softwaremetaphern? Sie sollen Ihnen ein Verständnis für Programmierprobleme und -prozesse vermitteln. Metaphern sollen Denkhilfen und Verbesserungsvorschläge sein. Zwar werden Sie kaum in der Lage sein, sich eine Zeile Quelltext anzusehen und sofort zu sagen, welche der in diesem Kapitel beschriebenen Metaphern in dieser Zeile verletzt werden. Im Laufe der Zeit werden Ihnen die Metaphern jedoch ein Gefühl für die Programmierung geben, und Sie werden schneller und besser programmieren als Entwickler, die ohne Metaphern arbeiten.

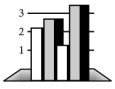
2.3 Gebräuchliche Softwaremetaphern

Für die Softwareentwicklung gibt es verwirrend viele Metaphern. David Gries sagt, das Schreiben von Software sei eine Wissenschaft (1981). Donald Knuth hält es für eine Kunstform (1998). Watts Humphrey beschreibt es als Prozess (1989). P. J. Plauger und Kent Beck sagen beide, es gleiche dem Fahren eines Autos, sie kommen aber zu praktisch gegensätzlichen Schlussfolgerungen (Plauger 1993, Beck 2000). Alistair Cockburn behauptet, es sei ein Spiel (2002). Für Eric Raymond ist es ein Basar (2000). Andy Hunt und Dave Thomas fühlen sich an Gartenbau erinnert. Paul Heckel vergleicht das Schreiben von Software mit dem Verfilmen von *Schneewittchen und die sieben Zwerge* (1994). Fred Brooks sagt, beim Schreiben von Software fühle er sich wie ein Landwirt, wie ein Werwolfjäger oder als ertränke er zusammen mit den Dinosauriern im prähistorischen Sumpf (1995). Welches sind nun die besten Metaphern?

Der Software-Autor: Quelltext schreiben

Die simpelste Metapher für die Softwareentwicklung stammt von dem Ausdruck »Quelltext schreiben« ab. Diese Metapher verleitet zu der Annahme, dass ein Programm genau wie ein Brief an die Großmutter geschrieben wird – man setzt sich mit Stift und Papier hin und schreibt den Text von Anfang bis Ende. Exakte Planung ist überflüssig, da Ihnen beim Schreiben einfällt, was Sie mitteilen wollen.

Viele Vorstellungen beruhen auf dieser Metapher. Jon Bentley meint, ein Programm sollte so geschrieben sein, dass man es wie einen guten Roman genießen kann – mit einer guten Zigarre entspannt zurückgelehnt im Ohrensessel, auf dem Tisch steht ein Glas alten Weinbrands, das Kaminfeuer prasselt, und der Lieblingsjagdhund wärmt einem die Füße. Brian Kernighan und P. J. Plauger gaben ihrem Buch über Programmierstile den Namen *The Elements of Programming Style* (1978) in Anlehnung an die Stilfibel für Schriftsteller, *The Elements of Style* (Strunk und White 2000). Programmierer sprechen oft von der »Lesbarkeit« eines Programms.



FAKTEN

Für die Arbeit Einzelner oder für kleinere Projekte stimmt diese Metapher auch. Ansonsten ist sie nicht ausreichend, um den Prozess der Softwareentwicklung adäquat zu beschreiben. Das Schreiben wird meist von einer Person allein verrichtet; an einem Softwareprojekt arbeiten hingegen viele Leute mit unterschiedlicher Verantwortung. Wenn Sie mit einem Brief fertig sind, packen Sie ihn in einen Umschlag und werfen ihn in den Briefkasten. Änderungen sind dann nicht mehr möglich. Software kann jedoch sehr wohl nachträglich verändert werden, und meist ist Software niemals endgültig fertig. Bis zu 90 Prozent des Entwicklungsaufwands eines typischen Softwaresystems fallen nach seiner ersten veröffentlichten Version an, durchschnittlich sind es zwei Drittel (Pigoski 1997). Beim Schreiben eines Briefs oder Buchs kommt es vorrangig auf Originalität an. Bei der Softwareentwicklung werden hingegen oft Entwurfsideen, Quelltext und Testfälle vorangegangener Projekte wieder verwendet, um die Effizienz zu erhöhen. Alles in allem greift die Metapher des »Schreibens« von Software zu kurz – ein solcher Hauruckstil kann nicht gut gehen.

Rechnen Sie fest damit, dass Sie eine Version wegwerfen müssen. Auf die eine oder andere Art kommt es immer dazu.

Fred Brooks

Wenn Sie fest eingeplant haben, eine Version wegzuwerfen, werden es sicher zwei werden.

Craig Zerouni

Leider ist diese Metapher auch in einem der populärsten Softwarebücher der Welt weiterverbreitet worden, nämlich von Fred Brooks' *Vom Mythos des Mann-Monats* (*The Mythical Man-Month,* 1995). Darin heißt es: »Rechnen Sie immer damit, dass Sie eine Version wegwerfen müssen. Auf die eine oder andere Art kommt es immer dazu.« Das beschwört das Bild lauter weggeworfener halb fertiger Manuskripte herauf (Abbildung 2.1).

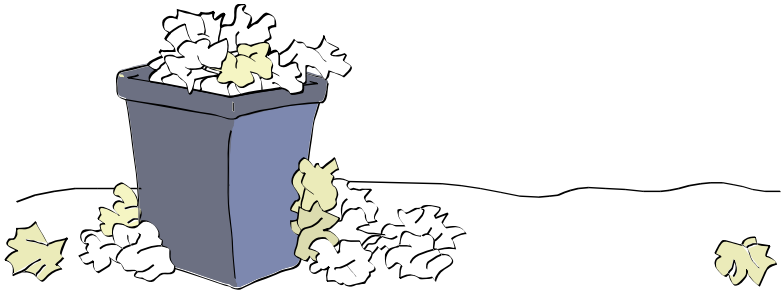


Abbildung 2.1: Die Schreibmetapher verleitet zu der Annahme, die Softwareentwicklung sei ein Prozess, der eher auf vielen Versuchen beruht als auf sorgfältiger Planung

Dieses Konzept mag ja noch ganz gut sein, wenn Sie einen netten Geburtstagsgruß an Ihre Tante schicken. Sie wären allerdings nicht gut beraten, wenn Sie diesen Rat auch bei Projekten beherzigen würden, wo große Softwaresysteme

bereits so viel wie ein zehnstöckiges Bürogebäude oder ein Kreuzfahrtschiff kosten. Wenn Sie eine eigene Schießbude haben, können Sie üben, so viel Sie wollen, bis Sie das Röhrchen mit der schönsten Blume fast im Schlaf treffen. Sollte dies nicht der Fall sein, müssen Sie Ihrer Angebeteten wohl oder übel die Blume im ersten Anlauf holen – oder Sie üben vorher drei, vier Mal, damit Sie sich nicht blamieren, wenn es ernst wird. Andere Metaphern sind sicher besser geeignet, um die Entwicklung von Software zu illustrieren.

Der Landwirt: Software anbauen

Die nächste Metapher steht im Gegensatz zur starren Schreibmetapher. Betrachten Sie die Softwareentwicklung wie ein Landwirt: Sie säen, jäten, gießen und so weiter, auf dass die einzelnen Pflanzen wachsen und gedeihen und am Ende eine gute Ernte herauskomme. Genauso entwerfen Sie ein Programmelement, schreiben den Quelltext, testen und verbessern es, bis Sie es schließlich in das System einbauen. Dann kommt das nächste Element. Auf diese Weise wächst das System in kleinen Schritten, und der Ärger mit Fehlfunktionen hält sich in Grenzen.

WICHTIG

Manchmal wird ein gutes Verfahren mit einer unpassenden Metapher beschrieben. So auch in diesem Fall. Versuchen Sie in solchen Fällen, eine bessere Metapher zu finden! Das schrittweise Verfahren ist in Ordnung, die Landwirtmetapher leider misslungen.

Weiterführende Literatur
Eine andere Metapher aus der Landwirtschaft, die sich auf die Pflege von Software bezieht, finden Sie im Kapitel »On the Origins of Designer Intuition« in *Rethinking Systems Analysis and Design* (Weinberg, 1988).

Zugegeben, der Gedanke, Software Schritt für Schritt aufzubauen, erinnert entfernt an das Wachstum von Pflanzen auf Feldern. Das war's dann aber auch schon, diese Metapher kann daher durch bessere, später beschriebene Metaphern ersetzt werden. Diese Metapher könnte Ihnen genauso gut nahe legen, den Systementwurf zu düngen, den Ertrag des Codes durch pfiffige Felderverwaltung zu verbessern und am Ende den Code zu ernten (Abbildung 2.2). Oder sollten Sie besser das Land für ein Jahr stilllegen, damit sich der Stickstoffgehalt der Festplatte stabilisiert?

Der eigentliche Schwachpunkt dieser Metapher liegt jedoch in der Unterstellung, Sie hätten keinen direkten Einfluss auf die Entwicklung der Ernte beziehungsweise der Software: Sie bringen also die Codesaat im Frühling aus, und wenn das Landwirtschaftsministerium und das Wetter mitspielen, haben Sie im Herbst allen Grund für ein Erntedankfest.

Abbildung 2.2: Die Landwirtmetapher eignet sich nur ungenügend für die Darstellung der Softwareentwicklung

Austernzucht: Das wachsende System

Manchmal ist mit dem »Anbau« von Software eher die »Zunahme« von Software gemeint. Beide Metaphern ähneln einander, die letztere ist jedoch einleuchtender. Unter »Zunahme« ist hier also der Größenzuwachs zu verstehen, der durch schrittweises Hinzufügen entsteht, genauso wie in einer Auster eine Perle entsteht oder wie sich Sediment am Meeresboden absetzt und so neues Land bildet.

Querverweis Details zur Anwendung schrittweiser Strategien auf die Systemintegration finden Sie in Abschnitt 29.2, »Integrationsfrequenz – synchron oder inkrementell?«.

Sie müssen also lernen, wie Sie Ihre Software in kleinen Schritten aufbauen oder (mit anderen Worten) »schrittweise«, »inkrementell« aufbauen. Andere Bezeichnungen in diesem Zusammenhang sind »iterativ«, »adaptiv«, und »evolutionär«. Inkrementelle Verfahren bei Entwurf, Softwarebau und Test bilden einige der leistungsfähigsten Konzepte, die für die Softwareentwicklung zur Verfügung stehen.

Bei der inkrementellen Entwicklung erzeugen Sie zunächst die allereinfachste, aber lauffähige Version Ihres Programms. Vergessen Sie realistische Ein- und Ausgaben oder Berechnungen – alles, was Sie in diesem Stadium brauchen, ist ein Skelett, das Sie anschließend nach und nach mit Leben füllen. Wahrscheinlich muss dieses Minimalprogramm zunächst Dummyklassen aufrufen, die für die grundlegenden, von Ihnen festgelegten Programmfunktionen stehen. Dieser erste Schritt der Programmentwicklung ist wie das Sandkorn, um das herum in der Auster die Perle wächst.

Nachdem das Skelett steht, kommen nach und nach Muskeln und Haut hinzu. Schritt für Schritt tauschen Sie die Dummyklassen durch die eigentlichen Klassen aus. Ihr Programm arbeitet dann zum Beispiel nicht mehr mit irgendwelchen Testdaten, sondern mit richtigen, vom Benutzer eingegebenen Daten. Und das Programm täuscht nicht mehr vor, Ausgaben zu erzeugen, sondern Sie fügen den Code ein, der die korrekten Ausgaben berechnet. Sie fügen Schritt für Schritt kleine Codestücke hinzu, bis Sie ein voll funktionsfähiges System haben.

Die Belege für die Wirksamkeit dieses Ansatzes sind beeindruckend. Selbst Fred Brooks, der 1975 in seinem Klassiker *Vom Mythos des Mann-Monats* vorschlug, eine Version zum Wegwerfen einzuplanen, bestätigte später, dass nichts im Jahrzehnt nach dem Erscheinen des Buchs seine Arbeitsweise und vor allem deren Effizienz so verändert hat wie der Ansatz der inkrementellen Entwicklung (1995). Tom Gilb schlug mit seinem bahnbrechenden Buch *Principles of Software Engineering Management* (1988) in dieselbe Bresche. Darin stellte er das Prinzip der »evolutionären Fertigung« (Evolutionary Delivery) vor und bereitete den Boden für viele moderne Ansätze im Umfeld des so genannten »Agile Programming«. Zahlreiche aktuelle Methodenlehren basieren auf diese Idee (Beck 2000, Cockburn 2002, Highsmith 2002, Reifer 2002, Martin 2003, Larman 2004).

Die Stärke der Austernmetapher besteht darin, dass sie nicht zu viel verspricht. Sie lässt sich nicht beliebig – wie die Landwirtmetapher – bis in völlig blödsinnige Analogien ausdehnen. Eine Auster, in der eine Perle heranwächst, ist ein gutes Bild für die inkrementelle Entwicklung.

Ein Haus errichten: Software bauen

Die Vorstellung, Software zu »bauen«, hat viel mehr mit der Realität zu tun, als etwa Software zu »schreiben« oder »anzubauen«, und verträgt sich gut mit der Vorstellung eines inkrementellen Zuwachses, das heißt der Austernmetapher. Wie beim Bau eines Hauses durchlaufen Sie Phasen wie Entwurf, Planung, Vorbereitung und Ausführung – Ihnen fallen bestimmt weitere Parallelen ein.

Um einen 1 Meter hohen Turm zu bauen, benötigen Sie eine ebene Unterlage, eine ruhige Hand und zehn unbeschädigte Bierdosen. Um einen Turm zu bauen, der 100 Mal so hoch ist, reichen 1000 Bierdosen nicht ganz aus – hier müssen Sie völlig anders planen und bauen.

Wenn Sie ein einfaches Gebäude erstellen wollen, beispielsweise eine Hundehütte, fahren Sie in den nächsten Heimwerkermarkt, kaufen Holz und Nägel, und bevor der Tag zu Ende ist, hat Fido ein neues Zuhause. Wenn Sie die Tür vergessen (Abbildung 2.3) oder andere Fehler machen, ist das kein großes Problem. Sie können den Fehler entweder an Ort und Stelle beheben oder einfach von vorn anfangen; das kostet Sie vielleicht ein paar Stunden. Auch an kleine Softwareprojekte kann man auf diese lässige Weise herangehen. Wenn sich bei einem Programm von 1000 Zeilen Quelltext herausstellt, dass der Ansatz nicht stimmt, können Sie das Programm refaktorieren oder gleich neu anfangen, der Zeitverlust ist auf jeden Fall nicht bedeutend.

Abbildung 2.3: Fehler bei einfachen Gebäuden kosten etwas Zeit und sind vielleicht ein wenig peinlich

Wenn es statt einer Hundehütte ein richtiges Haus sein soll, ist die Bautechnologie wesentlich komplexer, und auch Fehler in der Konstruktion oder im Entwurf wirken sich erheblich ernster aus. Zunächst müssen Sie entscheiden, was für ein Haus Sie bauen wollen – analog zur Problemdefinition bei der Entwicklung von Software. Anschließend müssen Sie zusammen mit dem Architekten einen allgemeinen Entwurf ausarbeiten und genehmigen lassen. Diese Phase ähnelt dem Entwurf der Softwarearchitektur. Dann fertigen Sie detaillierte Pläne an und nehmen ein Bauunternehmen unter Vertrag – was den detaillierten Ausarbeitungen eines Softwareentwurfs entspricht. Nun wird das Bauland erschlossen, das Fundament gelegt, das Haus hochgezogen und verputzt, worauf Strom- und Wasserleitungen verlegt werden. Diese Phase nennt der Programmierer »Softwarebau«.

Wenn die gröbste Arbeit erledigt ist, kommen Maler, Garten- und Innenarchitekt zum Zug – dies entspricht dem Optimieren von Software. Der gesamte Bauprozess wird von der Bauaufsicht begleitet und überwacht, was den Softwareüberprüfungen und -untersuchungen entspricht.

Die Größenordnung der Aufgabe und die damit einhergehende Komplexität wirken sich sowohl auf den Hausbau als auch auf die Programmierung von Software aus. Die Baumaterialien für ein Haus sind natürlich teurer als die für eine Hundehütte, der wichtigste Kostenfaktor ist jedoch die menschliche Arbeit. Eine Wand herauszureißen und sie um 2 Meter zu versetzen, ist nicht so teuer wegen der paar Ziegel, sondern weil Sie die Handwerker dafür bezahlen müssen. Daher sollten Sie so sorgfältig wie möglich vorausplanen (Abbildung 2.4), um vermeidbare Fehler auch wirklich zu vermeiden. Bei Softwareprodukten sind die eingesetzten »Baumaterialien« sogar noch billiger, Arbeit hat aber auch hier ihren Preis. Das Format eines Berichts zu ändern, kann genauso teuer werden, wie eine Wand zu versetzen, da in beiden Fällen die aufgewendete Arbeitszeit der wichtigste Kostenfaktor ist.

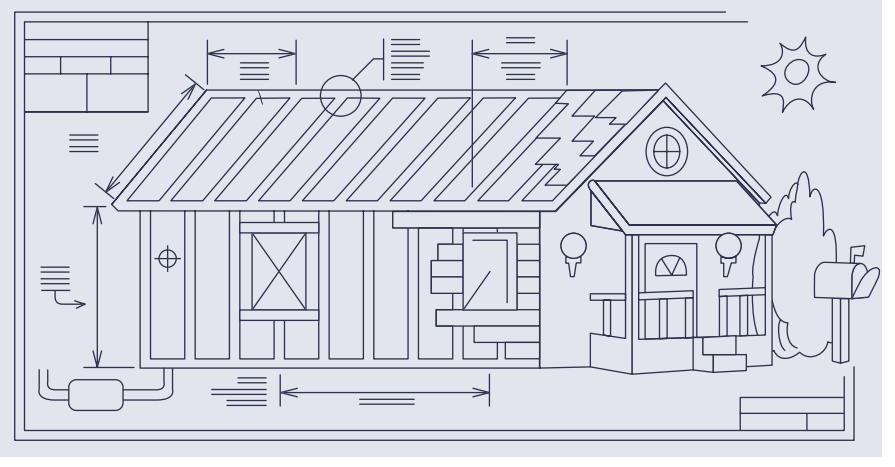


Abbildung 2.4: Kompliziertere Gebäude erfordern eine sorgfältigere Planung

Wir wollen nach weiteren Gemeinsamkeiten suchen. Beim Hausbau werden Sie sicher nicht versuchen, Dinge zu bauen, die Sie auch fix und fertig kaufen können, so zum Beispiel Waschmaschine, Geschirrspüler oder Kühlschrank. Nur wenn Sie ein Schraubergenie sind, könnte es für Sie reizvoll sein, diese Gegenstände selbst zu bauen. Sie werden wahrscheinlich auch fertige Türen, Fenster oder Badezimmerarmaturen kaufen. Genauso gehen Sie vor, wenn Sie ein Softwaresystem bauen. Sie werden das meiste in Hochsprachen schreiben und so weit wie möglich auf eigene, systemnahe Assemblerroutinen verzichten. Sie werden wahrscheinlich vorgefertigte Bibliotheken mit Containerklassen, Klassen für wissenschaftliche Berechnungen, für die Bildschirmausgabe oder für die Verwaltung von Datenbanken verwenden. Es ist im Allgemeinen nicht sinnvoll, Code neu zu schreiben, den Sie – bereits ausgetestet und optimiert – auch kaufen können.

Wenn Sie allerdings ein besonders schickes Haus mit einer ganz besonderen Innenausstattung im Sinn haben, kann es sein, dass die Einbauschränke extra angefertigt werden müssen. Und wer weiß, vielleicht bevorzugen Sie weltweit einzigartige Designerfenster. Wieder gibt es Parallelen zur Softwareentwicklung. Wenn Sie ein Spitzenprogramm entwickeln, kann es nötig werden, eigene wissenschaftliche Funktionen zu erstellen, wegen der Genauigkeit oder der Geschwindigkeit. Vielleicht erstellen Sie auch eigene Containerklassen, Klassen für Benutzeroberfläche oder Datenbankzugriff, um dem System eine nahtlose und einheitliche Benutzerschnittstelle zu geben.

Doch die Ähnlichkeiten reichen weiter. Bei beiden Tätigkeiten müssen Fehler ähnlich bezahlt werden; beide erfordern ein sorgfältiges Projektmanagement. Wenn Software in der falschen Reihenfolge gebaut wird, kann ein Monster entstehen, das überall Schwierigkeiten macht – beim Schreiben des Quelltextes, beim Testen, beim Debugging – und das fröhlich Ihre Zeit auffrisst. Das Projekt kann sogar ganz und gar auseinander brechen, wenn die Beiträge der einzelnen Programmierer so komplex sind, dass der Überblick beim Zusammenbau des Gesamtsystems verloren geht.

Sorgfältige Planung muss nicht extrem detailliert oder gar übertrieben sein. Sie können den Rohbau planen und später entscheiden, ob Parkett oder Teppichboden verlegt wird, in welcher Farbe die Wände gestrichen werden, wie das Hausdach gedeckt wird und so weiter. Ist ein Projekt gut geplant, können Sie später einfacher die Details ändern. Je mehr Erfahrung Sie mit der Art Software haben, die Sie bauen, desto mehr Details können Sie außen vor lassen. Sie sollten nur sicherstellen, dass die Planung so weit durchgeführt wurde, dass später keine größeren Probleme deswegen auftreten.

Die Baumetapher hilft auch zu erklären, warum unterschiedliche Softwareprojekte von unterschiedlichen Entwicklungsansätzen profitieren. Beim Gebäudebau kommen unterschiedliche Stufen von Planung, Entwurf und Qualitätssicherung zum Einsatz, je nachdem, ob Sie ein Lagerhaus, einen Geräteschuppen, ein Krankenhaus oder ein Atomkraftwerk bauen. Aber auch beim Bau von Schule, Hochhaus oder Einfamilienhaus werden Sie jeweils andere Ansätze verwenden. Bei der Softwareentwicklung wenden Sie in vielen Fällen flexible Ansätze an, aber manchmal brauchen Sie formelle, schwerfällige Verfahren, um Sicherheitsvorgaben oder andere Ziele zu erreichen.

Änderungen an der Software führen auf eine weitere Gemeinsamkeit mit dem Hausbau. Eine Trennwand lässt sich eher versetzen als eine tragende Wand. Genauso sind grundlegende Änderungen an der Struktur eines Programms teurer als die Veränderung nebensächlicher Details.

Auch bei sehr großen Projekten lassen sich Parallelen zwischen Hausbau und Softwareproduktion ziehen. Da die Strafen für ein Versagen solcher großen Projekte erheblich sind, muss deren Struktur »over-engineered«, das heißt mehrfach übersichert sein. Pläne werden sorgfältigst ausgearbeitet und überprüft. Sicherheitsreserven werden eingebaut, denn es ist bestimmt sinnvoller, 10 Prozent mehr Geld für stärkeres Material auszugeben, als dass der Wolkenkratzer eines schönen

Tages umfällt. Große Bedeutung hat auch die Zeitplanung. Beim Bau des Empire State Building hatte jeder Lastkraftwagen einen Zeitplan, der auf die Viertelstunde genau war. Kam ein Wagen zu spät, verzögerte sich das gesamte Projekt.

Ebenso erfordern sehr umfangreiche Softwareprojekte eine ganz andere Größenordnung an Planung als einfach »nur« große Projekte. Capers Jones veranschlagt zum Beispiel für ein System mit 750.000 Zeilen Quelltext im Durchschnitt 69 *unterschiedliche Arten* der Dokumentation (1998). Die Spezifikation der Anforderungen für ein solches System umfasst normalerweise 4000 bis 5000 Seiten, und die Entwurfsdokumentation kann sogar das Zwei- bis Dreifache dieses Umfangs erreichen. Ein einzelner Programmierer kann den Entwurf eines solchen Projekts wahrscheinlich nicht mehr vollständig erfassen, geschweige denn lesen. Hier ist es angebracht, die Vorbereitung des Projekts auszudehnen.

Wir wagen uns an Softwareprojekte, die sich durchaus mit dem Bau des Empire State Building vergleichen lassen. Projekte dieser Größenordnung verlangen vergleichbare Managementmethoden.

Weiterführende Literatur Einige gute Anmerkungen zum Erweitern der Baumetapher finden Sie in »What Supports the Roof?« (Starr 2003).

Wir könnten die Analogie zum Hausbau noch in viele andere Richtungen verfolgen – diese Metapher ist sicherlich die gelungenste unter den hier vorgestellten. Die Softwareentwicklung hat viele Begriffe aus dem Bauwesen entlehnt: Da gibt es die Softwarearchitektur, den Bau, im Englischen sind viele weitere üblich: *scaffolding, foundation classes, tearing code apart* und so weiter. Ihnen werden bestimmt weitere Ausdrücke dieser Art begegnen.

Softwaretechniken: Der geistige Werkzeugkasten

Entwickler erstklassiger Software haben Jahre damit verbracht, Dutzende von Verfahren, Kniffen und Tricks auszuarbeiten. Hierbei handelt es sich nicht um Kochrezepte, sondern um analytische Werkzeuge. Ein guter Handwerker weiß, welche Werkzeuge er für seine Arbeit braucht und wie er sie einsetzen muss. Für einen guten Programmierer gilt das Gleiche. Je mehr Sie in die Programmierung einsteigen, umso mehr analytische Werkzeuge sammeln Sie in Ihrem Kopf, zusammen mit dem Wissen über deren richtige Anwendung.

Querverweis Details zum Auswählen und Kombinieren von Entwurfsmethoden finden Sie in Abschnitt 5.3, »Bausteine entwerfen: Heuristik«.

Wahrscheinlich kennen auch Sie Softwareconsultants, die ihre Lieblingsverfahren als einzig wahre Lehre predigen und alle anderen Methoden strikt ablehnen. Wenn Sie sich auf deren Heilslehre ganz und gar einlassen, legen Sie sich Scheuklappen an. Sie laufen dann Gefahr, andere, möglicherweise für Ihr momentanes Problem besser geeignete Methoden zu übersehen. Darum denken Sie sich bitte Ihr Gehirn als einen Werkzeugkasten, in dem jedes Werkzeug seine Daseinsberechtigung und seinen wohlgeordneten Platz hat.

Metaphern kombinieren

Da Metaphern eher einen heuristischen denn einen algorithmischen Charakter haben, brauchen sie sich nicht gegenseitig auszuschließen. Wenn Ihnen Hausbau und Austernzucht gleichermaßen zusagen – bitte. Verwenden Sie den Begriff »Schreiben«, wenn Sie mögen, jagen Sie Werwölfe oder ertrinken Sie zusammen mit Dinosauriern im prähistorischen Sumpf. Verwenden Sie die Metapher oder

Kombination aus Metaphern, die Ihr Denken anregt oder in Ihrem Team gut ankommt.

Aber Vorsicht, denn Metaphern sind ein zweischneidiges Schwert. Einerseits müssen Sie die Metaphern ausbauen, um von den heuristischen Anregungen zu profitieren. Leicht können Sie eine Metapher aber überstrapazieren oder in die falsche Richtung ausbauen. Metaphern können genau wie Werkzeuge falsch eingesetzt werden. Ordnen Sie die Metaphern richtig ein, dann werden sie zu einem wertvollen Element Ihres geistigen Werkzeugkastens.

Weiterführende Literatur

cc2e.com/0285

Unter den allgemeinen Büchern über Metaphern, Modelle und Paradigmen ragen folgende heraus:

Kuhn, Thomas S.: *Die Struktur wissenschaftlicher Revolutionen,* Frankfurt/M.: Suhrkamp, 2003. Kuhns Buch über das Entstehen, den Aufstieg und das Vergehen (eigentlich die Verdrängung durch andere) wissenschaftlicher Theorien in einem darwinistischen Kreislauf sorgte bei seiner Veröffentlichung im Jahr 1962 unter den Wissenschaftsphilosophen für Furore. Das Buch ist klar und kurz und enthält viele interessante Beispiele für Aufstieg und Fall von Metaphern, Modellen und Paradigmen in der Wissenschaft.

Floyd, Robert W.: »The Paradigms of Programming«, Rede anlässlich der Verleihung des Turing-Preises, 1978. *Communications of the ACM,* August 1979, 455–460. In diesem faszinierenden Text werden Modelle für die Softwareentwicklung diskutiert und Kuhns Ideen auf das Gebiet der Software angewandt.

Zusammenfassung

- Metaphern tragen heuristischen Charakter; sie sind keine Algorithmen. Metaphern wirken daher ein wenig verschwommen.
- Metaphern helfen Ihnen, den Prozess der Softwareentwicklung zu verstehen, indem sie Beziehungen zu anderen, wohlbekannten Tätigkeiten herstellen.
- Es gibt gute und weniger geeignete Metaphern.
- Die Produktion von Software ähnelt dem Bau eines Hauses: Diese Metapher weist auf die Bedeutung sorgfältiger Planung und auf die Unterschiede zwischen kleinen und großen Projekten hin.
- Stellen Sie sich Methoden der Softwareentwicklung als Werkzeuge in einem Werkzeugkasten vor: Jeder Programmierer hat viele Werkzeuge, denn es gibt kein Werkzeug, das für alle Arbeiten geeignet ist. Ein guter Programmierer wählt das jeweils passende Werkzeug.
- Metaphern schließen sich nicht gegenseitig aus. Verwenden Sie die Kombination von Metaphern, die Ihnen den größten Nutzen bringt.