Basiswissen Softwarearchitektur

Verstehen, entwerfen, wiederverwenden

von
Torsten Posch, Klaus Birken, Michael Gerdom

3., akt. u. erw. Aufl.

Thematische Gliederung:

Software Engineering

dpunkt.verlag 2011

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 89864 736 6

Inhaltsverzeichnis: Basiswissen Softwarearchitektur – Posch / Birken / Gerdom

10 Softwarearchitektur im industriellen Maßstab

Ich kann freilich nicht sagen,
ob es besser werden wird, wenn es anders wird;
aber so viel kann ich sagen,
es muß anders werden, wenn es gut werden soll.

Georg Christoph Lichtenberg

Aktuelle Paradigmen in der Softwareentwicklung wie objektorientierte oder komponentenbasierte Entwicklung haben uns in die Lage versetzt, Systeme von großer Komplexität zu bauen. Trotz einiger Erfolge hat die Softwareentwicklung jedoch bisher nicht den Zustand erreicht, bei dem man von einer »industriellen Vorgehensweise« sprechen könnte. Dies zeigt sich u.a. bei der Entwicklung von Produktfamilien, komplexen verteilten Systemen und beim heute erreichten, enttäuschenden Grad an Wiederverwendung. Die Entwickler von Gütern in den weniger dynamischen Technologien wie Werkzeugmaschinen, Autos oder PCs sind der Softwarewelt um einiges voraus.

In diesem Kapitel gehen wir auf die Thematik der Softwareentwicklung im industriellen Maßstab und den speziellen Beitrag der Softwarearchitektur ein. Dabei werden zunächst die chronischen Probleme der heutigen Softwareentwicklung aufgezeigt. Danach werden zwei zukunftsweisende Innovationsfelder erläutert, welche die Überwindung dieser Probleme ermöglichen sollen. Schließlich wird als theoretischer Hintergrund der Zusammenhang von Komplexität und der Abstraktionslücke erörtert.

Es gibt bereits zukunftsweisende Ansätze zur systematischen Wiederverwendung, die sich aus den obigen Ideen ableiten lassen. Diese werden zum Schluss dieses Kapitels aufgelistet und in den beiden verbleibenden Kapiteln des Buchs vertieft.

10.1 Chronische Probleme der heutigen Softwareentwicklung

In den bisherigen Kapiteln dieses Buchs haben wir uns mit allen notwendigen Aspekten der Disziplin Softwarearchitektur beschäftigt. Dies reichte von Entwurf und Dokumentation bis hin zu Bewertung und Vorgehensmodellen. Die Sichtweise war dabei stets auf ein einzelnes Projekt konzentriert. Im Kontext von softwareentwickelnden Unternehmen müssen wir den Blickwinkel jedoch erweitern: Es werden kontinuierlich immer neue Systeme und diese in aufeinanderfolgenden Versionen entwickelt. Dies resultiert aus immer neuen Anforderungen an die Produkte und letztlich aus der Weiterentwicklung von Märkten und Technologien insgesamt.

Mehrere verwandte Softwaresysteme

Der Softwarearchitekt hat es also nicht nur mit einzelnen Projekten, sondern mit mehreren, verwandten oder zumindest ähnlichen Softwaresystemen zu tun. Natürlich kommen bei dieser globalen Sicht auch neue Fragestellungen dazu: Zentral ist die Frage, welche Probleme der heutigen Softwareentwicklung uns hindern, in dem oben beschriebenen, umfassenderen Rahmen zu denken. In diesem Abschnitt gehen wir auf drei Problemfelder näher ein, die in der Praxis so allgegenwärtig sind, dass wir sie als chronisch einstufen können. Der Softwarearchitekt wird zukünftig wichtige Beiträge leisten, um diese chronischen Probleme zu überwinden.

10.1.1 Unnötige Freiheitsgrade bei Sprachen und Tools

Universell einsetzbare Sprachen

Heute arbeiten Architekten, Komponentendesigner und Entwickler größtenteils mit Sprachen und Tools, die ihnen sehr viele Freiheitsgrade gestatten. Beispielsweise sind C++ oder Java universell einsetzbare Programmiersprachen, mit denen jedes denkbare Softwareproblem gelöst werden kann. In der Informatik gibt es für diese Allgemeinheit den Begriff der *Turing-Vollständigkeit*.

Zu viele Freiheiten, zu viele Fehlermöglichkeiten

Vergleicht man nun Applikationen, die aus der gleichen Problemdomäne kommen, stellt man fest, dass sich Konzepte, Mechanismen, Algorithmen, Lösungen usw. von Applikation zu Applikation immer wiederholen. Die von den Sprachen angebotene Allgemeinheit ist dafür unnötig und oft sogar störend. Das Fehlerpotenzial ist so groß wie die Freiheiten, die durch die Sprachen geboten werden. Geeignete, domänenspezifische Einschränkungen könnten hier helfen, die Produktivität der Entwickler zu steigern.

Als Beispiel wollen wir die Entwicklung grafischer Bedienoberflächen betrachten. Der Entwickler will die Applikationslogik an die Ereignisse aus der grafischen Benutzerschnittstelle anbinden; dazu ist

es nicht nötig, dass er direkten Zugriff auf die Mechanismen der Speicherverwaltung bekommt, z.B. überladene new-Operatoren und Pointerarithmetik in C++. Vielmehr möchte er auf Basis eines geeigneten Frameworks produktiv entwickeln und von Fehlern möglichst abgehalten werden.

Von Assembler zu 3GL

Als Analogie ist es nützlich, sich den Schritt von Assemblercode zu den heutigen 3GL-Sprachen wie C++ oder Java zu vergegenwärtigen. Bereits dieser Schritt hat die Produktivität immens verbessert, die Fehlerwahrscheinlichkeit erheblich eingeschränkt und dabei Wartung und Weiterentwicklung großer Softwarepakete erst möglich gemacht. Dabei ging es nicht einfach darum, eine Programmiersprache (Assembler) durch eine andere (3GL) zu ersetzen; vielmehr wurde das Denken und Arbeiten der Entwickler auf eine höhere Abstraktionsebene gehoben. Gleichzeitig konnte der Entwickler nicht mehr jede einzelne Instruktion des Prozessors separat benutzen; diese Freiheiten waren aber auch nicht notwendig.

Gründe für die Dominanz der universell einsetzbaren Sprachen

Modelle statt Sprachen

Die universell einsetzbaren Sprachen (engl. *general purpose languages*) haben zwar die obigen Nachteile, werden heute jedoch in den meisten Softwareprojekten eingesetzt. Generell würde eine höhere Abstraktionsebene bedeuten, mit *Modellen* zu arbeiten. Geeignete Ansätze für die Entwicklung auf Basis von Modellen gibt es zwar heute schon, diese bringen aber noch Probleme mit sich, die sich meist aus der Historie erklären lassen. Vor allem sind dies die folgenden Probleme [Greenfield04, S. 116]:

- Für die Arbeit mit Modellen werden *Modellierungssprachen* benötigt. Prominentestes Beispiel ist hier UML (vgl. Kapitel 6, »Dokumentation«). Der Schwerpunkt von UML liegt jedoch traditionell beim informellen Design. Für den Einsatz als Grundlage einer modellgetriebenen Entwicklung wird eine präzise Semantik benötigt, die erst mit UML 2.0 allmählich verfügbar ist. Darüber hinaus hat UML einen ähnlichen Anspruch auf universelle Einsetzbarkeit wie die 3GL-Sprachen und ist somit auch ähnlichen Problemen ausgesetzt (z.B. unnötige Freiheitsgrade).
- Den Schritt vom C++-Programm zum Assemblercode bewältigen Compiler; den Schritt vom Modell zum Programmcode müssen Codegeneratoren leisten. Diese hatten in der Vergangenheit bis heute einige wesentliche Schwächen. Der generierte Code nimmt

oft zu wenig Rücksicht auf die Besonderheiten der Zielplattform; das Ergebnis sind ineffiziente Systeme. Nicht zuletzt dadurch können Benutzer von Generatoren das Vertrauen in diese Technologie verlieren.

10.1.2 Schwerpunkt auf Einzelprojekten

Obwohl es zu fast jedem Softwareprodukt viele ähnliche Produkte gibt, wird doch ein Softwareprodukt meistens so entwickelt, als ob es nur dieses eine gäbe. Dies gilt bereits für die Planung von Softwareprojekten: Ziel ist oft die Lieferung einer Version eines bestimmten Produkts, obwohl sich durch die Berücksichtigung von mehreren ähnlichen Produkten und mehreren aufeinanderfolgenden Versionen ein höherer Gewinn erzielen lassen würde.

Produktfamilien

Im Rahmen einer industriellen Softwareentwicklung muss sich die Betrachtung immer über mehrere Produkte bzw. über Produktfamilien erstrecken. Betroffene Inhalte sind nicht nur wiederverwendbare Komponenten (siehe dazu auch den nächsten Abschnitt), sondern auch Anforderungen, Architektur- und Entwurfsmuster, Checklisten, Testfälle und nicht zuletzt Prozesse.

Gerade im Bereich der Entwicklungsprozesse lässt sich dieses Problem exemplarisch ablesen: Prominente Vertreter wie eXtreme Programming (XP) oder Unified Process (UP) gehen von einem immer neuen Satz von Anforderungen und einer unabhängigen Entwicklung aus. Wiederverwendbare Komponenten oder andere Güter werden nicht einbezogen [Greenfield04, S. 121].

Einzelprojektsicht hemmt Wiederverwendung.

Bei dieser Art der Entwicklung von Einzelprojekten fehlt es am Kontext, um Möglichkeiten zur Wiederverwendung zu erkennen. Die Organisation muss sich ständig Wissen und Artefakte neu erarbeiten, die eigentlich schon verfügbar wären. Nur sporadisch und eher zufällig wird es Wiederverwendung geben.

10.1.3 Ungenügendes Zusammenspiel von Komponenten

Komponentenbasierte Entwicklung greift nicht.

Ein Hauptziel der objektorientierten Entwicklung ist es, Komponenten und Systeme aus bereits existierenden, fertigen Klassen und Objekten zusammenzusetzen. In vielen anderen Industrien (z. B. PC-Hardware) war und ist dies der Schlüssel zu wirtschaftlichem und technologischem Erfolg. Doch weder auf Basis der Objektorientierung noch mittels der komponentenbasierten Entwicklungsmethodik konnte dieses Ziel bislang realisiert werden. Stattdessen entstehen nach Abschluss eines Projektzeitraums meist monolithische Systeme, weshalb das Pro-

blemfeld von Greenfield und Short als *monolithic construction* bezeichnet wird [Greenfield04, S. 110].

Man weiß mittlerweile, dass Klassen und Objekte zu feingranular sind, um daraus komplexe Systeme zusammenzusetzen. Aber was sind die Gründe dafür, dass es mit der gröberen Granularität von Komponenten nicht möglich ist, dass diese als Bauklötze zum Zusammenbau von Softwareprodukten verwendet werden können? Nach [Greenfield04] ist dies ein Symptom für eine Reihe fundamentalerer Probleme, die in Abbildung 10–1 zusammengefasst dargestellt sind. In den folgenden Abschnitten werden diese Probleme genauer erläutert.

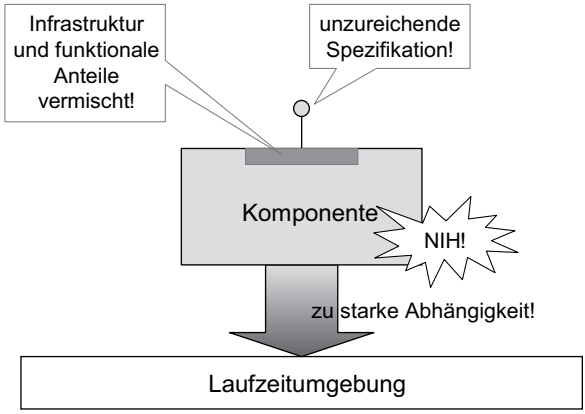

Abb. 10–1 Vier fundamentale Probleme beim Entwurf von wiederverwendbaren Komponenten

Zu starke Abhängigkeit von der Laufzeitumgebung

Komponenten brauchen eine passende Laufzeitumgebung, oft auch als *Plattform* bezeichnet. Heutige Komponentenplattformen (z.B. Java RMI oder .NET) arbeiten meist nicht zusammen, so dass auch die jeweiligen Komponenten nicht interoperabel sind. Die Komponenten sind eng an ihre Plattform gebunden. Will man z.B. einen Geschäftsprozess automatisieren, der über Abteilungs- oder sogar Unternehmensgrenzen geht, ist das bei unterschiedlichen Plattformtechnologien in den einzelnen Abteilungen bzw. Unternehmen nur schwer möglich. Vielfach sind Komponenten sogar an bestimmte Versionen ihrer Plattform gebunden; dies verschärft das Problem noch.

Unzureichende Spezifikation der Komponenten

Umfassende Spezifikation von Schnittstellen

Komponenten können nur dann außerhalb ihres Entwicklungskontexts eingesetzt werden, wenn sie eine aussagekräftige Spezifikation mitbringen. In Kapitel 5 wurden bereits die wichtigsten Bestandteile einer solchen Spezifikation erläutert; dazu gehören u.a. die syntaktische und semantische Beschreibung der implementierten und benötigten Schnittstellen, der Ressourcenbedarf der Komponente und Aussagen zu ihrer Performance.

Fehlen diese Angaben oder Teile davon, müssen bei der Verwendung der Komponente Annahmen gemacht werden. Diese Annahmen treffen oft nicht zu und führen dann zu schwer lokalisierbaren Fehlern auf verschiedenen Abstraktionsebenen. Beispiele für solche Annahmen sind:

- *Annahmen über Komponenten*
  Welche Dienste werden benötigt (z.B. Persistenz)? Kontrolliert die Komponente aktiv die Ausführung? Wer kontrolliert bestimmte Datenstrukturen?
- *Annahmen über Komponentenbeziehungen*
  Wie interagieren die Komponenten (z.B. asynchron oder synchron)? Welche Daten werden ausgetauscht? Wie werden die Daten ausgetauscht (z.B. Nachrichten oder Streams)?
- *Architektur-Annahmen*
  Interagieren Komponenten direkt oder über einen Broker bzw. Mediator? Welche Tools werden eingesetzt (z.B. Generierung aus Schnittstellenbeschreibungen)?

Übereifrige Kapselung

Komponente klebt oft an ihrer Plattform.

Gut gekapselte Komponenten lassen sich besser wiederverwenden. Die Kehrseite dieses Glaubenssatzes ist, dass die funktionalen Anteile der Schnittstellen einer Komponente oft stark mit dem Infrastrukturanteil vermischt werden. Dies geschieht z.B. dadurch, dass aus einer abstrakten Schnittstellenbeschreibung (z.B. in IDL oder bestimmten XML-Formaten) Quellcode in Form von Stubs (d.h. leeren Methodenrümpfen) generiert wird, der dann manuell mit Leben gefüllt werden muss.

Sollen im Rahmen eines Refactorings Funktionalitäten zwischen Komponenten verlagert werden, kommt man in Schwierigkeiten: Die funktionalen Codeanteile sind von nichtfunktionalen Anteilen durchdrungen und deshalb eng an ihre Komponente gebunden. Die Infrastrukturanteile der Komponente müssen manuell portiert werden.

Not-invented-here-Syndrom

Es ist auf Dauer ökonomischer, bereits bestehende, allerdings nicht 100 % passende Komponenten wiederzuverwenden, als genau passende Komponenten von Grund auf neu zu entwickeln.

Gründe für not invented here

Leider lehrt uns die Praxis oft genug, dass scheinbar das Gegenteil dieser Aussage gilt. Aus schlechten Erfahrungen mit zugelieferten Komponenten entscheiden sich viele Architekten und Projektleiter für Neuentwicklungen, obwohl es halbwegs passende Softwarebausteine auf dem Markt gegeben hätte. Dieses Phänomen wird auch als NIH-Syndrom (engl. *not invented here*) bezeichnet. Die Ursache für dieses Problem liegt in der mangelnden Reife der Kunden-Lieferanten-Beziehung; dies betrifft z. B. die Güte der Spezifikationen oder die mangelnde Einhaltung von Terminen. Nicht zuletzt scheint es für den einzelnen Entwickler angenehmer zu sein, neue Software auf der grünen Wiese zu erstellen, als bestehende Komponenten mit komplexen Schnittstellen zu verstehen und korrekt einzusetzen.

10.2 Bahnbrechende Innovationen

Einer Softwarearchitektur und -entwicklung im industriellen Maßstab stehen also die im vorherigen Abschnitt beschriebenen chronischen Probleme gegenüber. Es gibt jedoch bereits heute eine Reihe von Innovationen, die bei der Überwindung dieser Probleme helfen können. Diese bahnbrechenden Innovationen tragen wesentlich zu den Technologien bei, die in den verbleibenden Kapiteln dieses Buchs dargestellt werden.

Im Folgenden werden zwei Innovationsfelder vorgestellt, die gemeinsam den Weg hin zu einer industriellen Softwareentwicklung vorzeichnen: systematische Wiederverwendung und modellgetriebene Entwicklung.

10.2.1 Innovationsfeld 1: Systematische Wiederverwendung

Gemeinsamer Kontext erleichtert Wiederverwendung.

Es ist nahezu unmöglich, Komponenten zu entwickeln, die sich in beliebigem Kontext nutzen lassen. Bei jeder Softwarekomponente gibt es zum einen eine starke Abhängigkeit zur Plattform, in der die Komponente lebt; zum anderen gibt es einen engen Bezug zur jeweiligen Problemdomäne, zu deren Lösung die Komponente beiträgt [Greenfield04, S. 125f]. Legt man Problemdomäne und Plattform fest, so ist das Problem der Wiederverwendbarkeit einfacher lösbar. Die Plattform-Festlegung ist innerhalb einer Organisation ohne Schwierigkeiten möglich, z. B. innerhalb einer .NET- oder Webservice-Welt.

Produktfamilien

Zur Bestimmung der Problemdomäne trägt der Begriff der *Software-Produktfamilien* bei. Produkte einer Familie haben viele gemeinsame und einige spezifische Features. Bereits 1976 beschrieb David L. Parnas diesen für die Wiederverwendung zentralen Begriff (abgedruckt in [Hoffman00, S. 193ff]). Komponenten innerhalb einer Produktfamilie haben einen generischen Anteil, der die gemeinsamen Features adressiert, sowie einen konfigurierbaren Anteil, über den die spezifischen Features abgedeckt werden. Die Schnittstellen zwischen den Komponenten leben ebenfalls im Kontext der Produktfamilie; dies erleichtert die Zusammenarbeit von Komponenten. Eine systematische Wiederverwendung wird somit möglich.

Softwareproduktlinien

Produktlinien

Mittels Softwareproduktlinien wird diese systematische Wiederverwendung ausgenutzt und in die Praxis umgesetzt. In einer Softwareproduktlinie werden u.a. Anforderungen, Architekturen, Komponenten, Tests und Frameworks mit dem Ziel wiederverwendet, die Produkte einer Produktfamilie mit größtmöglicher Effizienz zusammenzubauen. Dafür besteht die Produktlinie im Wesentlichen aus zwei Teilen:

- Entwicklung gemeinsamer Güter
  (z.B. Anforderungen, Komponenten, Tests, Dokumentation)
- Entwicklung von Produkten

Der Produktlinienansatz überwindet zwei der oben dargestellten chronischen Probleme: Es wird nicht mehr nur ein einzelnes Softwareprodukt pro Projekt betrachtet; außerdem bietet die Produktlinie einen Rahmen, in dem Komponenten erfolgreich zusammenspielen können. Im nächsten Kapitel werden wir daher Softwareproduktlinien detailliert betrachten.

10.2.2 Innovationsfeld 2: Modellgetriebene Entwicklung

Modellbasiert ...

Der Softwarearchitekt ist bereits heute den Umgang mit Modellen gewöhnt, v.a. durch die Erarbeitung seiner Architekturen auf Basis von UML-Modellen. Die Modelle werden dabei hauptsächlich zur Dokumentation eingesetzt; die realen Systeme werden gemäß dieser Modelle implementiert. Diese Art der Entwicklung wird in der Literatur als *modellbasiert* bezeichnet [Stahl07]. Modelle und Implementierung haben dabei allerdings keine formale Verbindung und drohen daher ständig auseinanderzulaufen und damit inkonsistent zu werden.

... vs. modellgetrieben

Bei der modellgetriebenen Entwicklung (engl. *model-driven development*) wird die Implementierung durch einen Generator aus dem Modell automatisch erzeugt. Das Modell wird so selbst zu Quellcode, allerdings auf einer höheren Abstraktionsebene als der Quellcode in einer Programmiersprache wie Java oder C. Modellgetriebene Entwicklung adressiert somit das oben beschriebene chronische Problem der unnötigen Freiheitsgrade von Sprachen und Tools. Der Entwickler arbeitet mit dem Modell auf einer höheren Abstraktionsebene als bisher; die mühsame Handarbeit wird von Generatoren erledigt.

Metamodelle und domänenspezifische Sprachen

Ein wichtiger Aspekt bei der modellgetriebenen Entwicklung ist die Definition dieser Abstraktionsebene. Dazu muss beschrieben werden, wie Modelle für ein bestimmtes Anwendungsgebiet aussehen dürfen. Dies geschieht über *Metamodelle* und *domänenspezifische Sprachen*. Ein Metamodell beschreibt die mögliche Struktur von Modellen [Stahl07, S. 91]. Es beschreibt damit eine Modellierungssprache, mit der sich Modelle eines bestimmten Anwendungsgebiets ausdrücken lassen. Diese Sprache wird in der Literatur domänenspezifische Sprache (engl. *domain-specific language* oder DSL) genannt.

In Abbildung 10–2 wird die Idee der modellgetriebenen Entwicklung schematisch dargestellt. Modelle im Rahmen der modellgetriebenen Entwicklung müssen übrigens nicht unbedingt grafisch dargestellt werden; textuelle Modelle (z.B. als domänenspezifische Sprachen oder in Form von XML-Dateien) sind ebenfalls möglich.

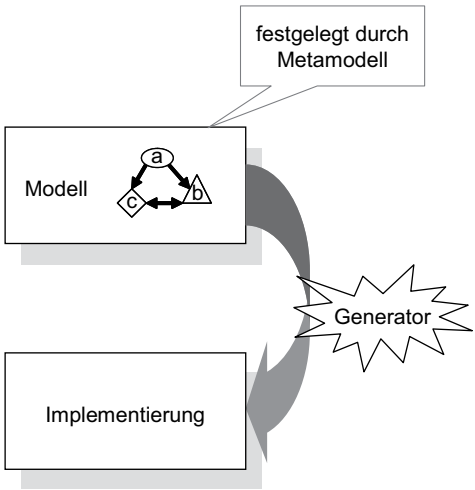

Abb. 10–2 Modellgetriebene Entwicklung: Aus dem Modell wird durch einen Generator eine ausführbare Repräsentation (Implementierung) erzeugt.

10.3 Komplexität und die Abstraktionslücke

Der Softwarearchitekt hat im oben beschriebenen Umfeld von mehreren, oft verwandten Softwareprojekten verschiedene Aufgaben zu bewältigen. Die meisten dieser Aufgaben haben mit Komplexität und Abstraktion zu tun. Zum Abschluss dieses Kapitels betrachten wir daher den Zusammenhang der Begriffe Komplexität und Abstraktion. Dies bildet gleichzeitig den theoretischen Unterbau zur Anwendung der im vorigen Abschnitt erklärten Innovationen.

10.3.1 Arten von Komplexität in der Softwareentwicklung

Komplexität ist ein Maß für die Schwierigkeit, ein bestimmtes Problem zu lösen. Bereits in den 70er Jahren wurde Komplexität als eine große Herausforderung in der Softwareentwicklung erkannt, dies ist bis heute so geblieben. Die Methoden und Technologien wurden zwar ständig verbessert, die Anforderungen an die Systeme sind jedoch gleichzeitig gewachsen. [Greenfield04, S. 35]. G. F. Leibniz formulierte es so: »Omne possibile exigit existere.« (Alles, was möglich ist, strebt nach Existenz.)

Essenzielle und versehentliche Komplexität

In der Softwareentwicklung lassen sich zwei grundsätzlich verschiedene Arten von Komplexität unterscheiden:

- *Essenzielle* Komplexität (engl. *essential complexity*) bezeichnet die Schwierigkeit eines zu lösenden Problems. Sie kann nicht reduziert oder vermieden werden, da es per definitionem keine »einfache« Lösung für das Problem gibt. Die essenzielle Komplexität eines Softwareprodukts ist eine Funktion seiner Features und der Abhängigkeiten zwischen diesen Features [Greenfield04, S. 36].
- *Versehentliche* Komplexität (engl. *accidental complexity*) ist eine ungewollte Eigenschaft einer *Problemlösung*, nicht des Problems selbst. Sie entsteht durch fehlerhafte Methoden oder Tools, durch falsche Planung oder durch ein falsch verstandenes Problem. Beispielsweise würde die Implementierung einer webbasierten Anwendung in der Programmiersprache C zu einem großen Maß an versehentlicher Komplexität führen.

Reduktion von Komplexität

Versehentliche Komplexität kann oft verringert oder gar vermieden werden, essenzielle Komplexität niemals. Damit ist versehentliche Komplexität ein vielversprechender Angriffspunkt für Optimierungen und Vereinfachungen.

Abbildung der Anforderungen auf die Lösung

Viele Anforderungen eines Softwaresystems lassen sich nicht innerhalb einer einzelnen Komponente umsetzen, sondern wirken sich auf mehrere Komponenten oder Module des Systems aus. Beispiel: In einer Enterprise-Anwendung wirkt sich jede Klasse des fachlichen Modells auf die Benutzerschnittstelle, die Fachlogik und die Persistenzschicht aus. Die logische Struktur der Anforderungen lässt sich also nicht 1:1 auf die Architektur der Lösung abbilden. Dies erzeugt zusätzliche querschnittliche Abhängigkeiten, die nicht im Problem, sondern in der Art der Lösung begründet liegen. Diese querschnittlichen Abhängigkeiten sind eine häufige Ursache für versehentliche Komplexität.

10.3.2 Die Abstraktionslücke

Die Zielsysteme der Softwareentwicklung sind letztlich Prozessoren und damit Maschinen, die konkrete Folgen von Anweisungen sequenziell ausführen können. Die Anforderungen, die ein Softwaresystem definieren, stellen dagegen eine abstrakte Beschreibung des Systems dar. In der Softwareentwicklung müssen wir nun den Sprung von den Anforderungen hin zu den ausführbaren Systemen schaffen. Dabei müssen die Anforderungen so lange mit konkreten Details »angereichert« werden, bis lauffähige Artefakte (z.B. Java-Code) vorliegen.

Abstraktionslücke oder semantische Lücke

Wie im vorigen Abschnitt dargestellt, gibt es zwischen der Ebene der Anforderungen und dem konkreten, ausführbaren System eine konzeptuelle Lücke, die es zu überwinden gilt. Diese Lücke wird in der Literatur semantische Lücke oder auch Abstraktionslücke genannt [Greenfield04, S. 41]. Abbildung 10–3 illustriert diesen Zusammenhang.

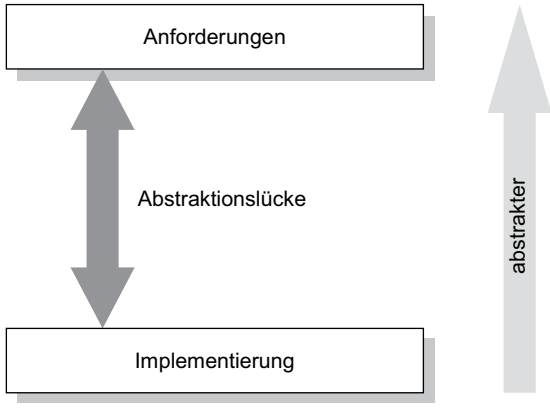

Abb. 10–3 Die Abstraktionslücke zwischen Anforderungen und Implementierung eines Systems

10.3.3 Verkleinern der Abstraktionslücke

Überwinden der Abstraktionslücke

Wie lässt sich nun die Abstraktionslücke überwinden? Dazu ziehen wir nochmals das Beispiel des Compilers heran, da sich diese Technologie seit vielen Jahrzehnten bewährt hat. Der Compiler definiert eine Programmiersprache und damit eine neue Abstraktionsebene, die signifikant über der vorgegebenen Ebene der ausführbaren Instruktionen des Prozessors liegt. Die Abstraktionslücke wird also durch den Compiler bzw. durch die von ihm definierte Abstraktionsebene verkleinert. Der Entwickler kommt durch die Verwendung von neuen Konzepten wie z.B. komplexen Datentypen, Polymorphie oder Kapselung näher an die Ebene der Anforderungen heran.

Beispiel: modellgetriebene Entwicklung

Die Abstraktionslücke lässt sich aber noch weiter verkleinern, dabei helfen Technologien wie Frameworks, Plattformen, Patterns oder Generierungswerkzeuge. Im Abschnitt zur modellgetriebenen Entwicklung haben wir bereits einen dieser wichtigen Ansätze kurz beschrieben. Die verschiedenen Abstraktionen sollten aufeinander aufbauen oder einander ergänzen. Beispiel: Ein Generator bildet Modelle einer höheren Abstraktionsebene auf eine virtuelle Maschine ab, die durch ein Framework zur Verfügung gestellt wird. Das Framework ist wiederum in einer bestimmten Programmiersprache implementiert, die von einem Compiler auf die Zielplattform abgebildet wird. Abbildung 10–4 zeigt dieses Beispiel schematisch.

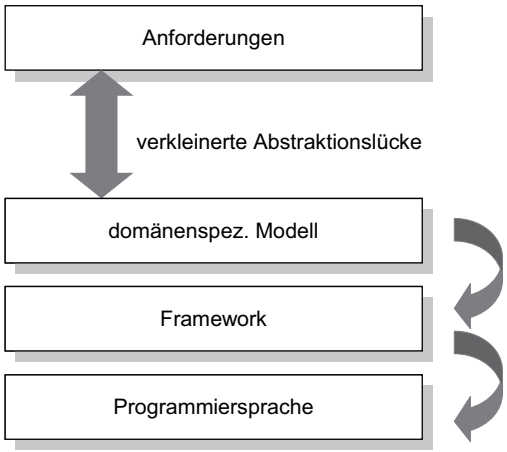

Abb. 10–4 Beispielhafte Möglichkeiten zur Verkleinerung der Abstraktionslücke

Rolle des Softwarearchitekten

Der Softwarearchitekt gibt die Abstraktionsebenen vor, indem er Tools und Technologien auswählt und deren Zusammenspiel definiert. Von der Auswahl der geeigneten Abstraktionen hängt es ab, ob die Kom-

plexität angemessen bewältigt werden kann. Durch die Wiederverwendung von Abstraktionen wirken diese auch im industriellen Maßstab. Die weiter oben beschriebenen Innovationen bilden einen Rahmen zum Auffinden, Formulieren und Einordnen dieser Abstraktionen.

10.4 Zusammenfassung

Das zurückliegende Kapitel hat die Basis für eine Beschäftigung mit den Anforderungen der industriellen Softwareentwicklung gelegt. Am Anfang des Kapitels lag die Erkenntnis, dass es heute einige chronische Probleme in der Softwareentwicklung gibt, die einer produktiven Softwareentwicklung im industriellen Maßstab entgegenstehen, z. B. die getrennte Entwicklung einzelner Softwareprodukte oder das ungenügende Zusammenspiel von Komponenten. Es gibt jedoch bereits heute Innovationen, die diese chronischen Probleme heilen können: Systematische Wiederverwendung und modellgetriebene Entwicklung sind wichtige Stichworte.

Der Schlüssel zu diesen Innovationen ist die Überwindung der Abstraktionslücke. Dies geschieht durch die Definition geeigneter Abstraktionsebenen als Zwischenstufen des Schrittes von den Anforderungen zur Implementierung. In der Praxis geschieht dies durch die Zusammenarbeit der Technologien Plattformen, Modelle, Patterns, Frameworks und Tools.

Zukunftsweisende Ansätze für den industriellen Maßstab

Zukunftsweisende Ansätze, siehe Folgekapitel

Welche Ansätze bringen nun diese Innovationen so zusammen, dass sie im industriellen Maßstab erfolgreich sind? Im Verlauf des Kapitels wurden diese bereits genannt:

- *Produktlinien für Software*
  Der Produktlinien-Ansatz ist bereits in der Industrie etabliert und hat zum Erfolg einiger prominenter Unternehmen entscheidend beigetragen. In Kapitel 11 wollen wir uns daher ausschließlich dem Ansatz der Softwareproduktlinien widmen.
- *Modellgetriebene Entwicklung und MDA*
  Modellgetriebene Entwicklung ist eines der oben genannten Innovationsfelder. Model-driven Architecture (MDA) ist ein Standard der OMG, der die modellgetriebene Entwicklung auf der Basis von UML umsetzt. MDA hält aktuell Einzug in unzählige Softwareprojekte in der Industrie. Kapitel 12 beschreibt den MDA-Ansatz im Detail.

- *Software Factories*
  Dieser noch recht neue, hauptsächlich von Microsoft vertretene Ansatz bringt Produktlinien, modellgetriebene Entwicklung, Baukastenentwicklung, agile Methoden und andere wichtige Innovationen zusammen. Er zielt auf einen generellen Innovationssprung ausgehend von der Objektorientierung hin zu einer echten industriellen Softwareentwicklung. Die detaillierte Beschäftigung mit diesem umfangreichen Thema würde den Rahmen dieses Buches sprengen; daher sei an dieser Stelle lediglich auf [Greenfield04] verwiesen.

Die beiden letzten Kapitel dieses Buchs sind also den Ansätzen Softwareproduktlinien und MDA gewidmet. Dies sind zwei Bereiche, die in der nächsten Dekade die Softwareentwicklung im industriellen Maßstab bestimmen und einen wesentlichen Einfluss auf die Produktivität und den Erfolg der Branche haben werden.

Die Rolle des Softwarearchitekten

Der Softwarearchitekt legt die Abstraktionsebenen fest, wählt beteiligte Technologien aus und entwirft Referenz- oder Produktlinienarchitekturen. Er besetzt also eine zentrale Rolle, sowohl im Rahmen einer Produktlinie als auch bei modellgetriebenen Ansätzen. Deshalb ist es für Softwarearchitekten unerlässlich, sich mit diesen Paradigmen zu beschäftigen. Die beiden folgenden Kapitel sollen hier Ausgangspunkt und Hilfestellung sein.