1 Softwarearchitektur als Herausforderung

Software ist allgegenwärtig: Fliegen, Telefonieren, Überweisen, Autofahren – all das wäre ohne Software unmöglich. Wie hat die Welt funktioniert, als es noch keine Software gab?

Man sollte meinen, dass Softwareingenieure inzwischen gelernt haben, wie man Software baut. Aber sie können es noch immer nicht: Softwareprojekte dauern zu lange, kosten zu viel, scheitern zu oft, und selbst wenn sie das Ziel der Inbetriebnahme erreichen, ist das Ergebnis zu oft mangelhaft: Bei der Inbetriebnahme sind Probleme wie mangelnde Performance oder mangelnde Stabilität die Regel, nicht die Ausnahme.

Was wir uns wünschen sind Effizienz und Sicherheit in der Projektdurchführung und hohe Qualität des fertigen Systems. Dabei helfen uns viele Ansätze: Neue Verfahren der Anforderungsanalyse, moderne Vorgehensmodelle, sorgfältige Qualitätssicherung, Wiederverwendung von Software, um nur einige zu nennen.

Ein ganz entscheidender Erfolgsfaktor ist dabei die Softwarearchitektur, das Thema dieses Buches. Es geht um die Frage, wie man die Millionen Programmzeilen großer Systeme so strukturiert, dass im Ergebnis die gewünschte Qualität erreicht wird: Wartbarkeit, Flexibilität, Performance und andere Eigenschaften. Dabei gilt Softwarearchitektur oft als akademisches Thema: Hauptsache, die Software läuft, die Architektur ist nicht so wichtig. Aber in Wirklichkeit ist gute Softwarearchitektur die Voraussetzung dafür, dass andere Erfolgsfaktoren überhaupt zum Tragen kommen – das zeigen die folgenden Überlegungen:

Anforderungsanalyse

Nicht selten stellt man bei der Systemeinführung fest, dass man das falsche System gebaut hat. Hier helfen neuere Verfahren der Anforderungsanalyse (etwa Volere, siehe [Robertson & Robertson 2000]). Sie gestatten es, die Anforderungen unterschiedlicher Interessengruppen strukturiert aufzunehmen und zu gewichten. Die Anforderungsanalyse ist ein Prozess, der das Projekt während seiner ganzen Laufzeit begleitet; jeder weiß, dass sich die Anforderungen laufend

ändern, egal wie gut sie aufgeschrieben sind. Sich ändernde Anforderungen beeinflussen den Projektplan, das Budget und vor allem die Architektur der Software. Aber auch die beste Änderungsverwaltung läuft in die Leere, wenn sich die Software den gewünschten Änderungen widersetzt – und deshalb ist gute Softwarearchitektur die Voraussetzung dafür, dass erhobene Anforderungen in die Software eingebaut werden können.

Vorgehensmodelle

Softwareprojekte sind schnell, wenn es gelingt, die Arbeit zu parallelisieren, und sie werden billiger, wenn die Arbeit wenigstens teilweise in Ländern mit niedrigerem Lohnniveau durchgeführt wird. Aber bevor man erste Budgetüberlegungen mit niedrigen Stundensätzen anstellt, sollte man überlegen, wie man die Arbeit zu Hause (*on shore*) und in der Ferne (*off shore*) sinnvoll aufteilt. Das Ziel ist klar: Jedes Teilteam befasst sich mit einer wohl definierten Aufgabe; die Schnittstellen zwischen den Teams sind eindeutig festgelegt; jedes Team weiß genau, was es zu liefern hat und auf welchen Ergebnissen es aufbaut. Das alles funktioniert nur unter der Voraussetzung, dass die Softwarearchitektur eine solche Aufteilung überhaupt zulässt. Wenn sie es nicht tut, sind die schönsten Projektpläne Makulatur.

Qualitätssicherung

Qualität entsteht nicht von alleine, sondern sie muss gesichert werden. Die Qualität des Gesamtsystems ergibt sich aus der Qualität seiner Bestandteile, und man kann die Qualität des Ganzen nur sichern, indem man die Qualität der Bestandteile überprüft. Auch das kann nur glücken, wenn das Gesamtsystem in vernünftig gestaltete Komponenten aufgeteilt ist.

Wiederverwendung

Man wird schneller fertig, wenn man vorhandene Software wieder verwendet, anstatt sie jedes Mal neu zu schreiben – das leuchtet ein. Aber trotzdem ist der Produktivitätsgewinn durch Wiederverwendung bis heute außerordentlich gering, wenn er überhaupt messbar ist. Der Grund dafür liegt darin, dass sich Software zur Wiederverwendung nur dann eignet, wenn sie den Prinzipien guter Softwarearchitektur folgt: klare, einfache Schnittstellen und minimale Abhängigkeiten. Aber damit tun wir uns schwer: Oft sind die Schnittstellen zu kompliziert, die Abhängigkeiten verborgen – und so ist Wiederverwendung ausgeschlossen.

Diese Überlegungen zeigen, dass Softwarearchitektur kein akademisches Thema ist, sondern dass Softwarearchitektur trotz der unverbindlichen Ästhetik, die im Namen mitschwingt, eine notwendige (leider keine hinreichende) Voraussetzung ist für effiziente und sichere Projektabwicklung. Es gibt eigentlich nur einen,

allerdings entscheidenden Erfolgsfaktor, der wirklich nichts mit Softwarearchitektur zu tun hat, nämlich:

Beauftragung mit Augenmaß

Licht hat eine feste, nicht beeinflussbare Geschwindigkeit, und kein Mensch wird jemals 100 m in sieben Sekunden laufen. Das ist allgemein bekannt – aber bei der Beauftragung und Planung von Softwareprojekten vergessen wir mitunter die Grenzen der Machbarkeit: Für jedes System mit gegebenem Funktionsumfang gibt es eine minimale Projektdauer und ein minimales Budget, das kein Projektteam dieser Welt zu unterschreiten in der Lage ist. Wer aus Naivität, Konkurrenzdruck oder allgemeinem Wunschdenken heraus ein Projekt mit unrealistischen Rahmenbedingungen unternimmt, der hat von vornherein verloren: Strukturiertes Vorgehen, klare Softwarearchitektur, sorgfältige Qualitätssicherung – die hehren Prinzipien des Software-Engineering entfalten ihre segensreiche Wirkung nur diesseits der Grenze der Machbarkeit, jenseits herrscht Chaos, ganz egal wie man vorgeht. Leider ist bisher niemand in der Lage, die Grenze der Machbarkeit genau anzugeben, aber oft würde es genügen, die bekannten Schätzverfahren sorgfältig anzuwenden, um zu erkennen, dass ein Projekt in der geforderten Zeit mit dem geplanten Budget einfach nicht funktionieren kann.

Für Projekte, die diesseits der Machbarkeitsgrenze operieren, ist eine gute Softwarearchitektur zwar noch nicht die Garantie für den Projekterfolg, aber immerhin eine solide Grundlage.

1.1 Was ist Softwarearchitektur?

Softwarearchitektur ist offenbar wichtig, aber was kann man sich darunter vorstellen? Die Architektur eines Softwaresystems hat mit der Architektur eines Gebäudes wenig gemeinsam: Die Architektur eines Gebäudes ist für jeden sichtbar, auch der Laie kann sich ein Urteil bilden. Wir würden aber Stromversorgung, Abwasserleitungen und Fahrstuhlschächte nicht als Teil der Architektur betrachten.

Die Architektur eines Softwaresystems ist umfassend; dazu gehört einfach alles: die Dialoge, Stapelverarbeitung, alle Softwarekomponenten und deren Zusammenspiel im kleinsten Detail. Die folgende Definition von Softwarearchitektur ist dank ihrer Allgemeinheit weitgehend akzeptiert. Sie stammt von [Bass et al. 1998] und wird oft zitiert (z.B. in [Bosch 2000] und [Starke 2002]):

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.*

Das ist in dieser Allgemeinheit natürlich richtig. Allerdings wäre der Begriff »Softwarestruktur« treffender und etwas weniger hochtrabend. Trotzdem bleiben wir – der Mehrheit folgend – bei dem nicht ganz glücklichen Begriff der »Softwarearchitektur«.

1.2 Quasar: Qualitätssoftwarearchitektur

Softwarearchitektur hat unmittelbare Auswirkungen auf Qualität und Kosten eines Systems. Qualitätssoftwarearchitektur – *Quasar* – ist das Thema dieses Buches. Quasar definiert ein Verständnis von Softwarequalität. Quasar versucht, besonders wichtige Regeln und Mechanismen der Softwaretechnik verständlich zu beschreiben, zu präzisieren und zum Standard zu erklären. Dies geschieht auf vier Ebenen:

- *Ideen und Konzepte:* Dies sind die Grundprinzipien der Softwaretechnik: Trennung der Zuständigkeiten, Denken in Schnittstellen und Komponenten. Diese Ideen sind uralt und wenigstens als Lippenbekenntnis weltweit akzeptiert, aber in der Praxis offenbart wohl jedes Review Verstöße gegen diese Prinzipien.
- *Begriffe:* Die Informatik krankt an vielen unklaren oder mehrdeutigen Begriffen. In diesem Buch definieren wir eine Reihe technikneutraler Begriffe, mit denen wir über Softwarearchitektur sprechen können, ohne uns auf EJB[1] oder CORBA[2] festzulegen.
- *Standardarchitektur und Standardschnittstellen:* Die meisten Datenbankzugriffsschichten haben ähnliche Schnittstellen. Quasar definiert für verschiedene Themen (wie den Datenbankzugriff) Standardschnittstellen mit Syntax und Semantik, auf denen jedes Projekt aufsetzen kann.
- *Standardkomponenten:* In einigen Bereichen (u.a. Persistenz, Autorisierung) wurden bei sd&m Quasar-konforme Komponenten entwickelt, die allen Projekten zur Verfügung stehen. Aber dabei steht immer die Schnittstelle im Vordergrund und nicht die Implementierung: Wenn eines Tages eine Autorisierungskomponente auf dem Markt erscheint, die signifikant besser ist als die eigene, dann werden wir auf das bessere Produkt wechseln, aber weil dies hinter definierten Schnittstellen passiert, ist die Anwendungssoftware nur in geringem Maß betroffen.

Aber der wichtigste Gedanke hinter all diesen Ebenen ist die *Einfachheit*. Quasar ist nicht gedacht als theoretischer Ballast, der den Softwarearchitekten ein schlechtes Gewissen beschert, sondern als Hilfe bei der Frage, wie man gegebene Anforderungen so einfach wie möglich in Software umsetzt. Natürlich kann man

1. Enterprise Java Beans
2. Common Object Request Broker Architecture

jedes Prinzip zu Tode reiten: Wer sogar die String-Klasse hinter einem Interface versteckt, der tut des Guten zu viel.

1.3 Der Fahrplan

Alle vier Ebenen sind Thema dieses Buches. Dabei ist stets unser Anliegen, weder im Allgemeinen hängen zu bleiben noch in den Details eines technischen API zu ertrinken. Die Aussagen und Vorschläge dieses Buches sind so weit wie möglich mit Programmbeispielen belegt, denn das Programm ist die letzte Ebene der Konkretion; genauer geht es nicht. Die Programmiersprache ist Java, denn Java ist weit verbreitet, einfach zu verstehen und trotz mancher Mängel eine brauchbare, elegante Sprache. Wir setzen voraus, dass der Leser Java wenigstens passiv beherrscht. Lesern mit Java-Nachholbedarf empfehlen wir das originale Werk über Java (siehe [Arnold et al. 2000]).

Obwohl alle Programmbeispiele in Java formuliert sind, ist dies kein Buch über Java, und seine Aussagen sind keineswegs auf Java beschränkt. Java ist das sprachliche Vehikel, um dem Leser die Botschaft eindeutig und unmissverständlich zu vermitteln – wir hätten genauso gut C#, Python, Eiffel oder irgendeine andere moderne Sprache wählen können, aber eine mussten wir nehmen, genauso wie das Buch, das Sie in Händen halten, in Deutsch vorliegt und nicht in Englisch oder Französisch.

Die Aussagen dieses Buches gelten mit geringen Einschränkungen für alle Programmiersprachen von Assembler über Cobol und C bis hin zu Java, C# und Haskell, denn die Grundfragen der Softwarearchitektur stellen sich in jedem System dieser Welt, und die Antworten hängen nur in geringem Umfang von der eingesetzten Sprache ab. Im Zentrum des Buches stehen *Schnittstellen und Komponenten*, und dieses Konzept ist unabhängig von der Objektorientierung: Schnittstellen und Komponenten lassen sich in jeder Sprache realisieren – in Cobol mit Mühe, in objektorientierten Sprachen recht gut. Aber man kann auch und gerade in objektorientierten Sprachen grässliche unwartbare Monolithen von vollständig vernetzten Klassenstrukturen zustande bringen, und davor schützen auch die Muster nicht – wohl aber Quasar, das ist jedenfalls der Anspruch.

Die ersten sieben Kapitel dieses Buches befassen sich mit Softwarearchitektur im Allgemeinen: Sie gelten für jede Art von Softwareentwicklung. Im zweiten Teil, das sind die Kapitel acht bis zehn, geht es um *Informationssysteme*. Ein Informationssystem oder kurz *System* ist ein sinnvolles Ganzes, das der Anwender als technische und fachliche Einheit begreift. Wir meinen damit sowohl betriebliche Informationssysteme (die also nur von Mitarbeitern eines Unternehmens genutzt werden) als auch überbetriebliche Informationssysteme (das schließt B2B, B2C, C2B ein). Hier der weitere Fahrplan:

1.3.1 Klassen und Schnittstellen (Kapitel 2)

Wir betrachten Klassen und Schnittstellen am Beispiel der Java-Behälter, anhand derer sich das Denken in Schnittstellen und Klassen in fast idealer Weise vorführen lässt. Wir empfehlen Kapitel 2 auch und vielleicht gerade den Java-Experten, denn es extrahiert aus den Java-Behältern viele Konzepte für den Softwareentwurf im Kleinen, die sich – wie Kapitel 3 zeigen wird – hervorragend auf den Softwareentwurf im Großen übertragen lassen. Kapitel 2 ist nicht gedacht als Crashkurs der JRE[3]-Grundlagen, obwohl es als Nebeneffekt auch diesen Zweck erfüllen könnte.

1.3.2 Komponenten und Schnittstellen (Kapitel 3)

Kapitel 3 bildet die Grundlage für den Rest des Buches: Dort definieren wir die drei wichtigsten Begriffe, nämlich *Schnittstelle*, *Komponente* und *Komposition*. Komponenten und Schnittstellen sind die Einheiten, in denen der Softwarearchitekt denkt. Wir erklären, was Komponenten sind und wie man große Komponenten aus kleinen Komponenten komponiert.

1.3.3 Softwarekategorien – wie findet man Komponenten? (Kapitel 4)

Neben dem Denken in Schnittstellen und Komponenten ist die Kontrolle der Abhängigkeiten die wichtigste Richtschnur des Softwarearchitekten. Die Vermeidung von Abhängigkeiten ist ein täglicher Kampf gegen eine siebenköpfige Hydra, deren Köpfe schneller nachwachsen, als man schauen kann: Es gibt die Abhängigkeiten vom Betriebssystem, von globalen Systemvariablen, von der eingesetzten Technik, von Nachbarsystemen, von der Verzeichnisstruktur, von anderen installierten Produkten, von eingekauften Komponenten und nicht zuletzt die Abhängigkeiten der selbst entworfenen Komponenten untereinander. Beim Finden von Komponenten und bei der Kontrolle von Abhängigkeiten sind die *Softwarekategorien* eine wichtige Hilfe. Dabei kommt der Trennung von Anwendung (A) und Technik (T) eine besondere Bedeutung zu. Softwarekategorien heißen umgangssprachlich auch *Blutgruppen*, und in der Tat gibt es da einige Analogien.

1.3.4 Fehler und Ausnahmen – Rechte und Pflichten (Kapitel 5)

Obwohl Fehler und Ausnahmen allgegenwärtig sind, werden sie in der Literatur stiefmütterlich behandelt. Programmierlehrbücher verzichten »aus Platzgründen« fast durchgängig auf jede Art von Fehler- bzw. Ausnahmebehandlung. Auf der Ebene der Programmiersprachen gibt es tief schürfende Diskussionen über

3. Java Runtime Environment

sinnvolle und weniger sinnvolle Mechanismen zur Ausnahmebehandlung, aber wir vermissen in der Literatur eine praxisnahe Auseinandersetzung mit der brennenden Frage, was passieren soll, wenn etwas passiert, das nicht passieren darf. Die meisten Systeme stürzen im besten Fall einfach ab, oft aber verursachen sie noch in der Agonie schlimme Fehler, die schwer zu analysieren und noch schwerer zu beheben sind. Bei sicherheitskritischer Software begegnet uns oft ein umgekehrtes Phänomen: Die angestrebte Zuverlässigkeit wird mit roher Kraft erzwungen. Man prüft Daten vielfach, man legt Funktionen mehrfach aus – aber die wirklich kritischen Punkte wurden übersehen, und wenn der Notfall eintritt, versagen alle parallelen Einheiten gleichzeitig. Der Umgang mit Notfällen, also mit Situationen, die eigentlich nicht vorkommen dürften, aber trotzdem gelegentlich vorkommen, ist ein Schwachpunkt in allen Projekten, in die wir Einblick hatten. Kapitel 5 befasst sich mit Fehlern, Ausnahmen und Notfällen und schlägt eine komponentenorientierte Strategie zu deren Behandlung vor.

1.3.5 Wie spezifiziert man Schnittstellen? (Kapitel 6)

Das ist eine der wichtigsten und bisher leider ungelösten Fragen der Softwareerstellung überhaupt. Nur die genaue Kenntnis der *Semantik* einer Schnittstelle versetzt uns in die Lage, diese sachgemäß zu verwenden, aber allzu oft sind die Schnittstellenbeschreibungen vage, ungenau oder einfach nicht vorhanden. Formale Spezifikationsmethoden sind zwar präzise, aber für die Praxis meistens zu kompliziert. Wir schlagen vor, ausgehend von der Idee des *Programming by Contract* die Semantik von Methoden mit wenigen einfachen Hilfsmitteln zu beschreiben: Vor- und Nachbedingungen, Invarianten, Zustandsmodelle und Testfälle. Die Spezifikationssprache heißt QSL (Quasar Specification Language). Wir erläutern sie in Kapitel 6 an einem einfachen Beispiel.

Im Anhang A spezifizieren wir eine realitätsnahe, an *QuasarAuthorization* angelehnte Autorisierungskomponente mit QSL. Diese Komponente ist hinreichend kompliziert, um die Tücken der Spezifikation zu illustrieren, aber doch so einfach, dass man sie mit ihren wesentlichen Elementen in einem kurzen Anhang darstellen kann. Das Beispiel zeigt, dass eine QSL-Spezifikation erstens nützlich ist und zweitens nur einen vertretbaren Aufwand erfordert.

1.3.6 Softwarearchitekturen (Kapitel 7)

Eine der wichtigsten Aussagen dieses Buches ist die Trennung von Anwendung und Technik. Diese beeinflusst die Architektur des Gesamtsystems: Es gibt die Architektur der Anwendung (A-Architektur) und die der Technik (T-Architektur), die sich weitgehend unabhängig voneinander entwickeln lassen.

1.3.7 Anwendungskern und Anwendungskomponenten (Kapitel 8)

Dieses Kapitel befasst sich mit der A-Architektur: Wie zerlegt man eine Anwendung in sinnvolle Komponenten; welche Schnittstellen exportieren sie und welche Schnittstellen importieren sie?

1.3.8 Pools, Persistenz und Transaktionen (Kapitel 9)

Persistenz ist ein zentrales Thema für jedes betriebliche Informationssystem: Datenbankzugriffe beeinflussen die Performance maßgeblich; und auch die Wartbarkeit eines Systems hängt stark davon ab, ob überhaupt und wie gut Datenbankzugriffe gekapselt sind. In diesem Kapitel befassen wir uns auch mit JDO[4] und mit *QuasarPersistence*, einem bei sd&m entwickelten Open-Source-Produkt.

1.3.9 GUI-Architektur (Kapitel 10)

Während bei den Zugriffsschichten eine gewisse Konvergenz festzustellen ist, gibt es bei den GUI-Bibliotheken noch einen erheblichen Wildwuchs. In diesem Kapitel definieren wir einige Grundlagen der GUI-Architektur und schaffen vor allem eine einheitliche Begriffswelt.

1.4 Quasar – Hilfe oder Korsett?

Manche Kollegen betrachten Quasar mit Zurückhaltung, weil sie befürchten, in ein enges, unbequemes Korsett gesteckt zu werden. Gelegentlich ist auch zu hören: »Die Quasar-Ideen klingen ja ganz vernünftig, aber meine Projekte sind völlig anders, da würde das alles nicht funktionieren.« Diese Aussage ist in ihrer Allgemeinheit schon deshalb schief, weil unter dem Quasar-Dach viele Konzepte unterschiedlicher *Verbindlichkeit* zusammengetragen wurden. Deshalb sind einige Bemerkungen über die Verbindlichkeit von Quasar angebracht.

Alle Kapitel (mit Ausnahme dieser Einführung) enthalten eine kurze Zusammenfassung mit den wichtigsten Regeln und Hinweisen. Dabei unterscheiden wir Vorgaben, die nach unserer Meinung in jedem Fall einzuhalten sind, und eher konstruktive Hinweise und Hilfen, die der Leser annimmt oder auch nicht. Beispiel: Im Kapitel 5 ist die Trennung der Begriffe *Fehler* und *Ausnahmen* essenziell; der erste Teil des Kapitels befasst sich mit dieser Unterscheidung. Es folgt der Vorschlag, Ausnahmen im Rahmen einer speziellen Konstruktion zu behandeln (Sicherheitsfassaden und D&R-Experten, siehe Abschnitt 5.4) – dieser Vorschlag

4. Java Data Objects, siehe [Roos 2003].

ist als Hilfe gedacht, nicht als Zwang, genau diese Konstruktion in jedem Detail zu übernehmen.

Die Aussagen der Kapitel 3 bis 6 gelten für jedes Softwaresystem: Wir glauben, dass man ohne eine solide Vorstellung von Komponenten und Schnittstellen kein vernünftiges System bauen kann – das gilt gleichermaßen für Informationssysteme, Gerätetreiber und Software im Automobil. Die Softwarekategorien (Kapitel 4) sind ein neuer Beitrag, der aber kein Ballast sein soll, sondern eine Hilfe beim Finden von Komponenten. Kapitel 4 zeigt, wie jedes Projekt seine eigenen Softwarekategorien definiert – geschickt gewählte Kategorien erleichtern den Softwareentwurf.

Kapitel 6 enthält zwei Botschaften. Erstens: Spezifiziere die Semantik der Schnittstellen so genau wie möglich. Zweitens: Hüte dich vor Seiteneffekten. Die Spezifikationssprache QSL ist als konstruktive Hilfe gedacht – mit QSL kann man bestimmt besser spezifizieren als in Prosa, aber wer meint, dass er mit Prosa zurechtkommt, darf es gern versuchen.

Kapitel 7 beschreibt eine Begriffswelt von verschiedenen Architekturen, die sich in den letzten Jahren bei sd&m rasch durchgesetzt hat. Die dort propagierte Trennung von Anwendungs- und Technikarchitektur ist derjenige Bestandteil von Quasar, der am häufigsten kontrovers diskutiert wird. Hierzu zwei Anmerkungen:

- Oberstes Ziel von Quasar ist Einfachheit und Klarheit des Entwurfs. Uns ist kein Beispiel bekannt, wo die Trennung von Anwendung und Technik mit diesem Ziel kollidiert, jedenfalls nicht, wenn man etwas nachdenkt.
- Die Trennung von Anwendung und Technik schadet der Performance nicht; im schlimmsten Fall kann man Methodenaufrufe durch Makros ersetzen. Datenabstraktion wurde erfunden und zum ersten Mal eingesetzt im A7-Projekt (siehe [Parnas 1972]) – unter technischen Restriktionen, die wir uns heute kaum noch vorstellen können.

Die Standardarchitektur des Anwendungskerns (Kapitel 8) bezieht sich auf komplexe Informationssysteme – und dort gilt die Standardarchitektur. Sie gilt in dieser Form nicht für andere Arten von Softwaresystemen: Gerätetreiber und Herzschrittmacher haben ihre eigenen Standardarchitekturen, aber die sind nicht Thema dieses Buches. Für die Kapitel 9 und 10 gilt sinngemäß dasselbe: Die dort definierten Standardarchitekturen für den Datenbankzugriff und für GUI-Oberflächen haben in dieser Form schon oft funktioniert und sind deshalb ein guter Ausgangspunkt für jeden, der so etwas neu programmieren will oder der ein vorhandenes Produkt untersucht.

1.5 Warum brauchen wir Quasar?

Es gibt doch J2EE[5], CCM[6] und anderes. Dazu folgende Antworten:

- Die Schnittstellen von Quasar sind wesentlich anwendungsnäher als die oben genannten technisch orientierten Schnittstellen. Quasar definiert einen Puffer zwischen der Anwendung und der Technik.
- Quasar lässt uns die Freiheit, diejenigen Produkte zu nutzen, die uns gefallen (oder auf denen der Auftraggeber besteht), und überall dort eigene Software zu verwenden, wo es keine brauchbaren Produkte gibt, oder wo wir etwas Besseres haben. Quasar ist keine Konkurrenzveranstaltung zu J2EE, sondern lässt uns das Beste davon nutzen.
- Quasar macht uns unabhängig von den schnellen und nicht beeinflussbaren Innovationszyklen im technischen Bereich (Wie sieht EJB in vier Jahren aus?).
- Quasar macht uns unabhängig von teuren, schwerfälligen und in der Weiterentwicklung nicht beeinflussbaren Produkten: Weil wir Produkte als Komponenten begreifen, die hinter Schnittstellen agieren, ist der Austausch von Produkten zwar nicht gratis, aber machbar.
- Quasar-Ideen und -Schnittstellen gelten sprachübergreifend, und zwar ohne Einschränkung für Java, C++ und C#, mit geringen Einschränkungen für C und mit größeren für Cobol.
- Quasar-Prinzipien helfen uns vor allem in Nicht-Standardsituationen, die ein Produkt niemals abdecken wird (etwa beim Zugriff auf Nachbarsysteme). Man kann keine Zugriffsschicht kaufen, die Änderungen gleichzeitig in eine relationale Datenbank und in ein Nachbarsystem schreibt.

1.5.1 Argumente gegen Quasar

Zwei Argumente gegen Quasar versuchen wir schon an dieser Stelle zu entkräften:

- *Quasar schadet der Performance*. Zusätzliche Schichten kosten tatsächlich ein paar Maschinenzyklen, aber das ist heute kein Problem; an dieser Front wird der Performance-Krieg nicht gewonnen. Viel wichtiger sind die Steuerungsmöglichkeiten, die ein nach Quasar-Ideen gebautes System bietet. Wir betrachten als Beispiel den Datenbankzugriff: Systeme sind dann langsam, wenn sie zum falschen Zeitpunkt zu viele Daten lesen. Wer den Quasar-Ideen folgt, der behält die vollständige Kontrolle über diesen Zeitpunkt und verfügt damit über viele Tuningmöglichkeiten.
- *Quasar schränkt die Menge der nutzbaren Funktionen ein*. Dies ist in doppelter Hinsicht falsch:

5. Java Enterprise Edition, siehe [J2EE].
6. CORBA Component Model, siehe [CCM].

- Wir bleiben beim Beispiel der Zugriffsschicht: Gerade weil die harten SQL-Anweisungen an einer Stelle konzentriert sind, kann man ohne Bedenken alle Spezialitäten des gegebenen DBMS verwenden. Man hat also mehr Funktionen, nicht weniger.
- Aus Anwendungssicht sind die technischen Schnittstellen viel zu kompliziert. Keine einzelne Anwendung braucht alle sechs Transaktionsmodi von EJB. Deshalb definiert Quasar eine einfache Transaktionsschnittstelle mit den Operationen *commit* und *rollback*, und erst in einer technischen Komponente wird entschieden, welcher Transaktionsmodus zur Anwendung kommt.

1.5.2 Quasar und Muster

Muster sind *Design in the Small*, Quasar ist *Design in the Large*. Quasar verwendet zahlreiche Muster: Die verschiedenen *Fabrikmuster*, der *Adapter* und die *Fassade* kommen laufend vor und werden meistens nicht eigens erwähnt. Manche Quasar-Elemente könnte man als Muster aufbereiten, aber das hat geringe Priorität (siehe [Gamma et al. 1995], [Schmidt et al. 2002]).

1.5.3 Quasar und Wissenschaft

Quasar ist aus wissenschaftlicher Sicht unfertig: Unsere Definitionen sind weitgehend informell; viele Konzepte und Ideen sind nur angedeutet, aber noch nicht in der Tiefe beschrieben. So ist Quasar auch eine Agenda für weitere Forschung, die wir aber nicht aus der Praxis heraus leisten können, sondern lieber den dafür vorgesehenen Forschungseinrichtungen überlassen. Als Themen bieten sich (mindestens) an:

- Komponenten und Schnittstellen (Kapitel 3)
- Softwarekategorien (Kapitel 4)
- Behandlung von Fehlern und Ausnahmen (Kapitel 5)
- Spezifikation von Schnittstellen (Kapitel 6)
- Aufteilung der Softwarearchitektur in A-, T- und TI-Architektur (Kapitel 7).

Die Kluft zwischen Praxis und Theorie ist auch heute noch sehr breit. Wer baut die Brücke? Wir betrachten Quasar als einen praxisseitigen Brückenkopf. Auf der Seite der Theorie sehen wir Arbeiten wie [Broy & Stolen 2001], wo ohne Anspruch auf unmittelbaren Praxisbezug ein rigides Gedankengebäude errichtet wird. Wird es gelingen, die beiden Welten zu verbinden?

1.5.4 Quasar und UML

UML 2.0 kennt 27 Pfeiltypen. In diesem Buch verwenden wir nur drei davon (*ruft auf*, *implementiert* und *instanziiert*), dazu den Kasten für Klasse/Komponente und einen Kreis (Lollypop) für Schnittstelle (siehe Abb. 1–1). UML-Klassenmodelle werden in der üblichen Weise eingesetzt.

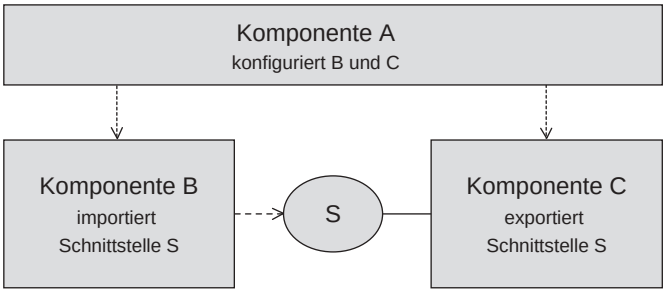

Abb. 1–1 *Konfiguration, Importeur und Exporteur*

Mehr zu Komponenten und Schnittstellen steht in den Kapiteln 2 und 3.

1.6 Was sagen andere?

Die Suche nach dem Schlagwort »Softwarearchitektur« liefert in Amazon über fünfzig Bücher, die in den letzten drei Jahren erschienen sind. Sie lassen sich grob in vier Kategorien unterteilen:

1. Theoretische Arbeiten (z.B. [Broy & Stolen 2001] und andere) werden in der Praxis bisher wenig wahrgenommen.
2. Die Muster-Bücher (etwa [Gamma et al. 1995], [Schmidt et al. 2002], [Shaw & Garlan 1996]). Muster gehören zu den wichtigen Errungenschaften der Softwaretechnik der letzten Dekade. Sie helfen vor allem beim Softwareentwurf im Kleinen; sie sind Mosaiksteine, die der Softwarearchitekt zu einem sinnvollen Ganzen zusammenfügt. Die wenigen Muster für Softwarearchitektur im Großen bewegen sich auf einer solchen Ebene der Allgemeinheit, dass der Nutzen für das einzelne Projekt reduziert ist. So beschreiben [Shaw & Garlan 1996] das Schichtenmodell so allgemein, dass sich die sieben ISO-Schichten, die Schichten eines Informationssystems und die Schalen eines Betriebssystems als Spezialfälle ergeben. Diese Darstellung ist von theoretischem Interesse, aber von geringem Nutzen im Projekt. Im Übrigen ist die Anzahl der verwendeten Muster oft nur schwach, manchmal auch negativ korreliert mit der Qualität der Architektur. Viele Projekte verwenden gerade die kleinen Muster unsachgemäß und unüberlegt (vor allem die Muster

Singleton und *Besucher* spielen hier eine unrühmliche Rolle) und machen so die Architektur kaputt.

3. Die Technik-Bücher. Viele Bücher, die den Begriff »Architektur« im Titel führen, erklären in Wirklichkeit eine bestimmte Technik wie EJB oder CORBA. Es ist zwar verdienstvoll, diese komplizierten technischen APIs verständlich aufzubereiten, aber ein großes Problem liegt woanders: Jedes einzelne Projekt braucht nur einen Bruchteil der überwältigenden Funktionsfülle, und es würde gut daran tun, die Verwendung spezieller Techniken weitgehend zu verbergen und in geeignete Adapter auszulagern. Das erzählen die Technik-Bücher aber gerade nicht, sondern im Gegenteil: Viele rufen dazu auf, die jeweilige Technik möglichst breit und flächendeckend einzusetzen. Aber das ist naiv und hat schon viele Projekte in Schwierigkeiten gebracht. An dieser Stelle sind übrigens auch die Berater der Herstellerfirmen zu nennen, die zwar die Technik in der Regel gut beherrschen, aber deren Ratschläge für die konkrete Nutzung die Abhängigkeiten von der Technik nicht selten maximieren, anstatt sie zu minimieren. Insofern dienen sie mehr den Interessen der Hersteller als denen des Projekts.
4. Die Bücher der guten Ratschläge. Diese enthalten viele wertvolle Informationen über Architektursichten, Strategien zum Finden einer guten Architektur und anderes, aber sie lassen den Leser zurück mit guten Ratschlägen der Form: »Identifizieren Sie Projektrisiken«, »Konzentrieren Sie sich auf die Schnittstellen«, »Entwerfen Sie nach Verantwortlichkeiten«. Dagegen lässt sich nicht das Geringste einwenden, aber wie geht es denn wirklich? Gern würden wir uns auf Schnittstellen konzentrieren, aber was ist da genau zu tun, was sind Schnittstellen überhaupt? Was kann man alles falsch machen und welche Konsequenzen hätte das?

Zwischen den allgemeinen Ratschlägen der Kategorie (4) und den Technik-Büchern der Kategorie (3) besteht eine Kluft, die von den Muster-Büchern nur teilweise gefüllt wird. Auch dieses Buch wird die Kluft nicht gänzlich schließen, aber es zeigt eine gangbare Brücke, die beide Welten verbindet.