Modernes Software-Engineering

Entwurf und Entwicklung von Softwareprodukten

Ian Sommerville

Pearson

EXTRAS ONLINE

Das ständige und zunehmende Risiko der Cyberkriminalität bedeutet, dass alle Produktentwickler Sicherheitsmaßnahmen in ihre Software integrieren müssen. Sicherheit ist für die Produktentwicklung so wichtig, dass ich diesem Thema ein eigenes Kapitel (▶ Kapitel 7) widme. Sie können ein sicheres System erhalten, indem Sie den Systemschutz als eine Reihe von Schichten gestalten (▶ Abbildung 4.5). Ein Angreifer muss alle diese Schichten durchdringen, bevor das System kompromittiert wird. Zu den Ebenen können Systemauthentifizierungsebenen, eine separate Authentifizierungsschicht für kritische Funktionen, eine Verschlüsselungsschicht usw. gehören. Architektonisch können Sie jede dieser Schichten als separate Komponente implementieren, sodass, wenn ein Angreifer eine dieser Komponenten gefährdet, die anderen Schichten intakt bleiben.

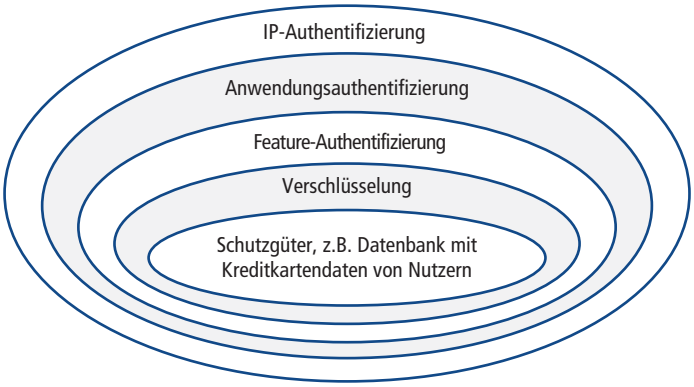

Abbildung 4.5: Authentifizierungsebenen

Leider gibt es Nachteile bei der Verwendung mehrerer Authentifizierungsebenen. Ein mehrschichtiger Sicherheitsansatz beeinflusst die Benutzerfreundlichkeit der Software. Benutzer müssen sich Informationen wie Passwörter merken, die benötigt werden, um in eine Sicherheitsschicht vorzudringen. Die Interaktion mit dem System wird durch die Sicherheitsfunktionen zwangsläufig verlangsamt. Viele Benutzer empfinden dies als lästig und suchen oft nach Möglichkeiten, dies zu umgehen, damit sie sich nicht erneut authentifizieren müssen, um auf Systemfunktionen oder Daten zuzugreifen.

Viele Sicherheitsverletzungen entstehen, weil sich die Benutzer nicht an Sicherheitsstandards halten, indem sie z.B. leicht zu erratende Passwörter auswählen, Passwörter mehrfach benutzen und Systeme ohne Abmelden verlassen. Dies passiert, weil die Anwender frustriert sind von Systemsicherheitsfunktionen, die schwer zu bedienen sind oder den Zugriff auf das System und seine Daten verlangsamen. Um dies zu vermeiden, benötigen Sie eine Architektur, die nicht zu viele Sicherheitsebenen hat, dem Benutzer keine unnötige Sicherheit aufzwingt und nach Möglichkeit Hilfskomponenten bereitstellt, die die Benutzer entlasten.

Die Verfügbarkeit eines Systems ist ein Maß für die Dauer der Betriebszeit dieses Systems. Sie wird normalerweise als Prozentsatz der Zeit ausgedrückt, die ein System für die Bereitstellung von Benutzerdiensten zur Verfügung steht. Eine Verfügbarkeit von 99,9% in einem System, das ständig verfügbar sein soll, bedeutet daher, dass das System für 86313 Sekunden von 86400 Sekunden an einem Tag verfügbar sein sollte. Die Verfügbarkeit ist besonders wichtig bei Unternehmensprodukten, wie z.B. Produkte für die Finanzindustrie, wo ein Betrieb rund um die Uhr erwartet wird.

Architektonisch verbessern Sie die Verfügbarkeit, indem Sie redundante Komponenten in einem System vorsehen. Um die Redundanz zu nutzen, binden Sie Sensorkomponenten ein, die einen Ausfall erkennen, sowie Schaltkomponenten, die den Betrieb auf eine redundante Komponente umstellen, wenn ein Ausfall erkannt wird. Das Problem hierbei ist, dass die Implementierung dieser zusätzlichen Komponenten Zeit in Anspruch nimmt und die Kosten für die Systementwicklung erhöht. Das System wird dadurch komplexer und somit steigt die Gefahr, Fehler und Schwachstellen einzuführen. Aus diesem Grund vermeiden die meisten Produktsoftwareprogramme ein Umschalten zwischen Komponenten bei einem Systemausfall. Wie ich in ▶ Kapitel 8 erkläre, können Sie mit zuverlässigen Programmiertechniken die Wahrscheinlichkeit von Systemausfällen reduzieren.

Sobald Sie sich für die wichtigsten Qualitätsmerkmale Ihrer Software entschieden haben, müssen Sie sich drei Fragen zum architektonischen Entwurf Ihres Produkts stellen:

1. Wie sollte das System als eine Reihe von Architekturkomponenten strukturiert sein, wobei jede dieser Komponenten eine Teilmenge der gesamten Systemfunktionalität bereitstellt? Diese Struktur sollte die Systemsicherheit, Zuverlässigkeit und Leistung bieten, die Sie benötigen.

2. Wie sollten diese architektonischen Komponenten verteilt werden und wie sollen sie miteinander kommunizieren?

3. Welche Technologien sollten beim Aufbau des Systems eingesetzt werden und welche Komponenten sollten wiederverwendet werden?

Ich gehe auf diese drei Fragen in den verbleibenden Abschnitten dieses Kapitels ein.

Architekturbeschreibungen in der Produktentwicklung bilden die Grundlage für das Entwicklerteam, um die Organisation des Systems zu diskutieren. Darüber hinaus haben sie die wichtige Aufgabe, das gemeinsame Verständnis dessen festzulegen, was zu entwickeln ist und welche Annahmen beim Entwurf der Software getroffen wurden. Das endgültige System kann vom ursprünglichen Architekturmodell abweichen – es ist also kein zuverlässiger Weg, die ausgelieferte Software zu dokumentieren.

Informelle Diagramme mit Symbolen zur Darstellung von Entitäten, mit Linien zur Darstellung von Beziehungen und mit erklärendem Text sind meines Erachtens der beste Weg, um Architekturen von Softwareprodukten zu beschreiben und Informationen darüber auszutauschen. Jeder kann am Designprozess teilnehmen. Man kann informelle Diagramme schnell und ohne den Einsatz spezieller Softwarewerkzeuge zeichnen und ändern. Informelle Notationen sind flexibel, sodass Sie unvorhergesehene Änderungen leicht vornehmen können. Neue Mitarbeiter, die einem Team beitreten, können sie ohne Spezialkenntnisse verstehen.

Das Hauptproblem bei informellen Modellen ist, dass diese mehrdeutig sind und nicht automatisch auf Auslassungen und Inkonsistenzen überprüft werden können. Wenn Sie formalere Ansätze verwenden, die auf Architekturbeschreibungssprachen (*Architectural Description Language*, ADL) oder der Unified Modeling Language (UML) basieren, reduzieren Sie Mehrdeutigkeiten und können Prüfwerkzeuge verwenden. Meine Erfahrung ist jedoch, dass formale Notationen dem kreativen Entwurfsprozess im Weg stehen. Sie schränken die Ausdrucksmöglichkeiten ein und verlangen, dass jeder sie versteht, bevor er am Designprozess teilnehmen kann.

4.3 Zerlegung des Systems

Die Idee der Abstraktion ist grundlegend für jedes Softwaredesign. Abstraktion im Softwaredesign bedeutet, dass Sie sich auf die wesentlichen Elemente eines Systems oder einer Softwarekomponente konzentrieren, ohne sich um die jeweiligen Details zu kümmern. Auf der architektonischen Ebene sollte Ihre Aufmerksamkeit auf größeren Komponenten liegen. Eine Zerlegung beinhaltet die Analyse dieser größeren Komponenten und deren Darstellung als eine Reihe von kleineren Komponenten.

▶ Abbildung 4.6 zeigt beispielsweise schematisch die Architektur eines Produkts, mit dem ich vor einigen Jahren beschäftigt war. Dieses System wurde für den Einsatz in Bibliotheken entwickelt und ermöglichte den Benutzern den Zugriff auf Dokumente, die in einer Reihe von privaten Datenbanken gespeichert waren, wie beispielsweise Rechts- und Patentdatenbanken. Der Zugang zu diesen Dokumenten war kostenpflichtig. Das System musste die Rechte an diesen Dokumenten verwalten sowie die Gebühren für den Zugang berechnen und einziehen.

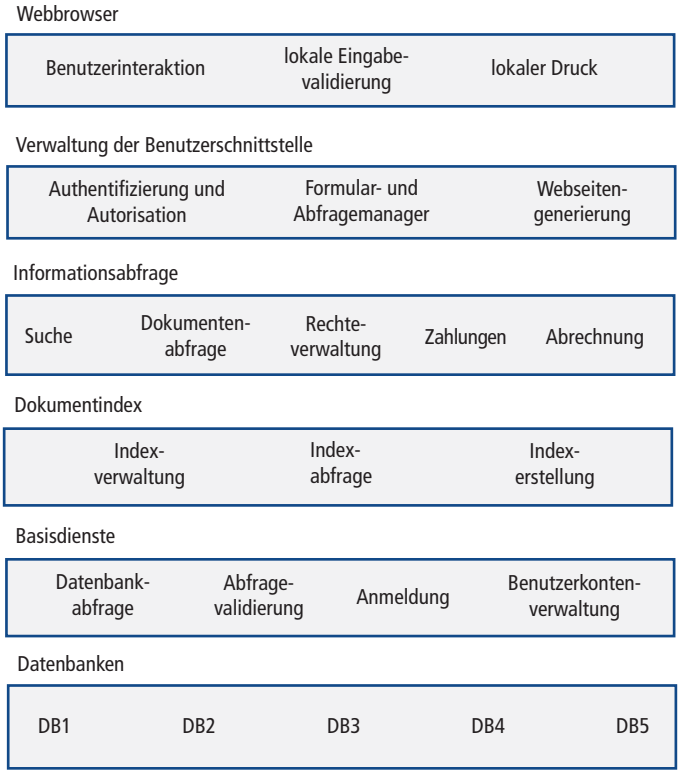

Abbildung 4.6: Architekturmodell eines Dokumentenabfragesystems

In diesem Diagramm enthält jede Schicht im System eine Reihe von logisch zusammenhängenden Komponenten. Informelle Schichtmodelle wie in ▶ Abbildung 4.6 werden häufig verwendet, um zu zeigen, wie ein System in Komponenten zerlegt wird, wobei jede Komponente eine bedeutende Systemfunktionalität bietet.

Webbasierte und mobile Systeme sind ereignisbasierte Systeme. Ein Ereignis der Benutzeroberfläche, wie beispielsweise ein Mausklick, löst die Aktionen aus, die der Benutzer durch seine Auswahl vorgegeben hat. Das bedeutet, der Kontrollfluss verläuft in einem mehrschichtigen System von oben nach unten. Benutzerereignisse in den höheren Schichten lösen Aktionen in dieser Schicht aus, die wiederum Ereignisse in den unteren Schichten auslösen. Im Gegensatz dazu verlaufen die meisten Informationsflüsse im System von unten nach oben. Informationen werden auf den unteren Ebenen erstellt, in die Zwischenebenen transformiert und schließlich an die Benutzer auf der obersten Ebene ausgegeben.
Oftmals herrscht Verwirrung über die Architekturterminologie bezüglich Wörtern wie „Service“, „Komponente“ und „Modul“. Es gibt keine standardisierten, allgemein akzeptierten Definitionen dieser Begriffe, aber ich versuche, sie in diesem Kapitel und an anderer Stelle im Buch konsequent zu verwenden:

1. Ein *Service* oder *Dienst* ist eine zusammenhängende Einheit von Funktionalität. Dies kann auf den verschiedenen Ebenen des Systems unterschiedliche Dinge bedeuten. So kann beispielsweise ein System einen E-Mail-Service anbieten und dieser E-Mail-Service selbst kann Dienste zum Erstellen, Senden, Lesen und Speichern von E-Mails beinhalten.

2. Eine *Komponente* ist eine benannte Softwareeinheit, die einen oder mehrere Dienste für andere Softwarekomponenten oder für die Endbenutzer der Software anbietet. Bei Verwendung durch andere Komponenten werden diese Dienste über eine API angesprochen. Komponenten können mehrere andere Komponenten verwenden, um ihre Dienste zu implementieren.

3. Ein *Modul* ist ein benannter Satz von Komponenten. Die Komponenten in einem Modul sollten etwas gemeinsam haben. So können sie beispielsweise eine Reihe von verwandten Diensten bereitstellen.

Komplexität in einer Systemarchitektur entsteht durch die Anzahl und die Art der Beziehungen zwischen den Komponenten in diesem System. Darauf gehe ich in ▶ Kapitel 8 näher ein. Wenn Sie ein Programm ändern, müssen Sie diese Beziehungen verstehen, um zu wissen, wie sich Änderungen an einer Komponente auf andere Komponenten auswirken. Bei der Zerlegung eines Systems in Komponenten sollten Sie es möglichst vermeiden, unnötige Komplexität in die Software zu bringen.
Komponenten haben unterschiedliche Arten von Beziehungen zu anderen Komponenten (▶ Abbildung 4.7). Aufgrund dieser Beziehungen müssen Sie, wenn Sie eine Änderung an einer Komponente vornehmen, oft Änderungen an mehreren anderen Komponenten vornehmen.
▶ Abbildung 4.7 zeigt vier Arten von Komponentenbeziehungen:

1. *ist-Teil-von:* Eine Komponente ist Teil einer anderen Komponente. So kann beispielsweise eine Funktion oder Methode Teil eines Objekts sein.

2. *benutzt:* Eine Komponente verwendet die Funktionalität, die von einer anderen Komponente bereitgestellt wird.

3. *ist-zusammen-mit:* Eine Komponente wird im gleichen Modul oder Objekt wie eine andere Komponente definiert.

4. *teilt-Daten-mit:* Eine Komponente teilt Daten mit einer anderen Komponente.

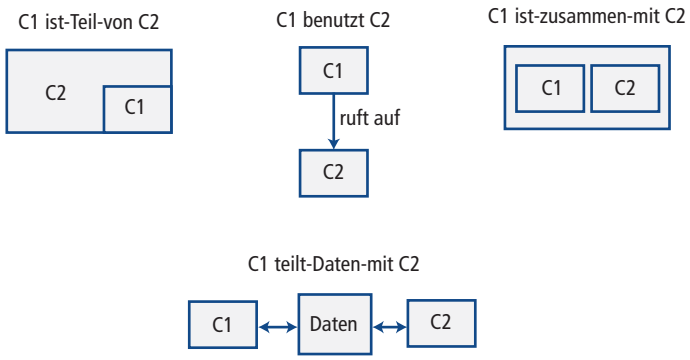

Abbildung 4.7: Beispiele für Komponentenbeziehungen

Mit zunehmender Anzahl der Komponenten steigt auch die Anzahl der Beziehungen – und zwar in der Regel im stärkeren Maße. Aus diesem Grund sind große Systeme komplexer als kleine Systeme. Es ist nicht zu vermeiden, dass die Komplexität mit der Größe der Software zunimmt. Sie können jedoch die Komplexität der Architektur durch zwei Maßnahmen kontrollieren:

1. *Beziehungen lokalisieren:* Wenn es z.B. Beziehungen zwischen den Komponenten A und B gibt, dann sind diese leichter zu verstehen, wenn A und B im selben Modul definiert sind. Sie sollten logische Komponentengruppierungen identifizieren (wie die Schichten in einer mehrschichtigen Architektur), deren Beziehungen sich hauptsächlich innerhalb der jeweiligen Komponentengruppe abspielen.

2. *Gemeinsame Abhängigkeiten reduzieren:* Wo die Komponenten A und B von einer anderen Komponente oder von Daten abhängen, steigt die Komplexität, da Änderungen an der gemeinsamen Komponente bedeuten, dass Sie verstehen müssen, wie sich diese Änderungen auf A und B auswirken. Verwenden Sie deshalb möglichst immer lokale Daten und vermeiden Sie den Datenaustausch.

Drei allgemeine Entwurfsrichtlinien helfen, die Komplexität zu kontrollieren (▶ Abbildung 4.8).

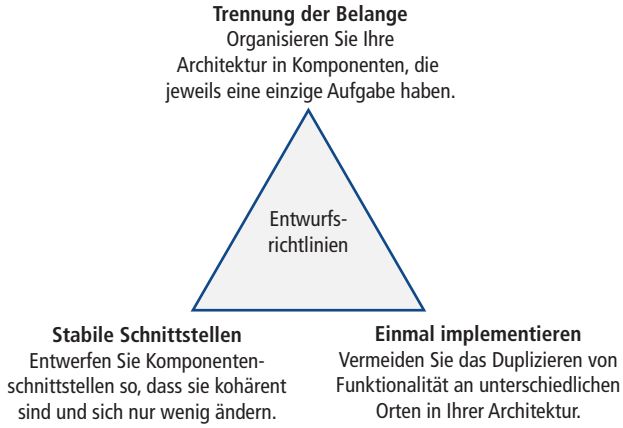

Abbildung 4.8: Architektonische Entwurfsrichtlinien

Die Richtlinie „Trennung der Belange“ (*Separation of Concerns*, SoC) empfiehlt, relevante architektonische Belange bzw. Zuständigkeiten in Gruppen von verwandten Funktionen zu identifizieren. Beispiele für architektonische Zuständigkeiten sind Benutzerinteraktion, Authentifizierung, Systemüberwachung und Datenbankmanagement. Im Idealfall sollten Sie für jede Zuständigkeit die entsprechenden Komponenten oder Gruppierungen von Komponenten in Ihrer Architektur identifizieren können. Auf einer niedrigeren Ebene bedeutet die Trennung der Belange, dass jede Komponente idealerweise nur für eine einzige Aufgabe zuständig sein sollte. Ich werde auf die Trennung der Belange in ▶ Kapitel 8 ausführlicher eingehen.

Die Richtlinie „Einmal implementieren“ empfiehlt, die Funktionalität in Ihrer Softwarearchitektur nicht zu duplizieren. Dies ist wichtig, da Duplikate bei Änderungen zu Problemen führen können. Wenn Sie feststellen, dass mehr als eine Architekturkomponente den gleichen oder einen ähnlichen Service benötigt beziehungsweise bereitstellt, dann sollten Sie Ihre Architektur reorganisieren, um Duplikate zu vermeiden.

Entwerfen und implementieren Sie niemals Software, bei der Komponenten die Implementierung anderer Komponenten kennen und sich auf diese verlassen. Implementierungsabhängigkeiten bedeuten, dass bei der Änderung einer Komponente auch die Komponenten geändert werden müssen, die auf deren Implementierung basieren. Implementierungsdetails sollten hinter einer Komponentenschnittstelle (API) verborgen sein.

Die Richtlinie „Stabile Schnittstellen“ ist wichtig, damit Komponenten, die eine Schnittstelle verwenden, nicht geändert werden müssen, weil sich die Schnittstelle ändert.

Mehrschichtige Architekturen, wie die in ▶ Abbildung 4.6 dargestellte Architektur des Dokumentenabfragesystems, basieren auf diesen allgemeinen Entwurfsrichtlinien:

1. Jede Schicht ist ein Zuständigkeitsbereich und wird getrennt von anderen Schichten betrachtet. Die oberste Schicht befasst sich mit der Benutzerinteraktion, die nächste Schicht mit der Verwaltung der Benutzeroberfläche, die dritte Schicht mit Informationsabruf und so weiter.

2. Innerhalb jeder Schicht sind die Komponenten unabhängig und überschneiden sich nicht in der Funktionalität. Die unteren Schichten enthalten Komponenten, die allgemeine Funktionalität bieten, sodass diese nicht in den Komponenten auf einer höheren Ebene repliziert werden müssen.

3. Das Architekturmodell ist ein abstraktes Modell, das keine Implementierungsinformationen enthält. Im Idealfall sollten z.B. Komponenten auf Ebene X nur mit den APIs der Komponenten auf Ebene X−1 interagieren. Das heißt, Interaktionen sollten zwischen zwei Schichten und nicht innerhalb einer Schicht stattfinden. In der Praxis ist dies jedoch ohne Codeduplizierung oft nicht möglich. Die unteren Schichten des Ebenenstapels bieten grundlegende Dienste, die eventuell von Komponenten benötigt werden, die sich nicht in der unmittelbaren Schicht darüber befinden. Es ist nicht sinnvoll, zusätzliche Komponenten in einer höheren Schicht hinzuzufügen, wenn diese nur für den Zugriff auf untergeordnete Komponenten verwendet werden.

Mehrschichtige Modelle sind informell und leicht zu zeichnen und zu verstehen. Sie können auf einem Whiteboard gezeichnet werden, sodass das gesamte Team die Zerlegung des Systems sehen kann. In einem Schichtmodell sollten Komponenten in unteren Schichten niemals von übergeordneten Komponenten abhängig sein. Alle Abhän-

gigkeiten sollten sich auf untergeordnete Komponenten beziehen. Das bedeutet, wenn Sie eine Komponente auf der Ebene X im Stapel ändern, sollten Sie keine Änderungen an Komponenten auf niedrigeren Ebenen im Stapel vornehmen müssen. Sie müssen nur die Auswirkungen der Änderung auf Komponenten auf höheren Schichten berücksichtigen.
Die Schichten im Architekturmodell sind keine Komponenten oder Module, sondern lediglich logische Gruppierungen von Komponenten. Sie sind relevant, wenn Sie das System entwerfen, aber in der Systemimplementierung können Sie diese Schichten normalerweise nicht identifizieren.
Die allgemeine Idee, Komplexität zu kontrollieren, indem man Zuständigkeiten innerhalb einer einzigen Schicht einer Architektur verortet, ist verlockend. Wenn Sie dies umsetzen können, müssen Sie keine Komponenten in anderen Ebenen ändern, wenn Komponenten in einer Schicht geändert werden. Leider gibt es zwei Gründe, warum das Lokalisieren von Zuständigkeiten nicht immer möglich ist:

1. Aus praktischen Gründen der Benutzerfreundlichkeit und Effizienz kann es notwendig sein, die Funktionalität so zu unterteilen, dass sie in verschiedenen Schichten implementiert ist.

2. Einige Zuständigkeiten sind bereichsübergreifend und müssen auf jeder Schicht im Stapel berücksichtigt werden (*Cross-Cutting Concerns*, CCC).

Ein Beispiel für das Problem der praktischen Trennung von Belangen sehen Sie in ▶ Abbildung 4.6. Die oberste Schicht beinhaltet „lokale Eingabevalidierung“ und die fünfte Schicht im Stapel beinhaltet „Abfragevalidierung“. Die Aufgabe „Validierung“ ist nicht in einer einzigen untergeordneten Serverkomponente implementiert, weil dies wahrscheinlich zu unnötig viel Netzwerkverkehr führen würde.
Wenn die Validierung von Benutzerdaten als Server- statt Browseroperation angelegt ist, so erfordert dies eine Netzwerktransaktion für jedes Formularfeld. Offensichtlich verlangsamt dies das System. Daher ist es sinnvoll, eine lokale Eingabeprüfung, z.B. eine Datumsprüfung, im Browser oder in der mobilen App des Benutzers zu implementieren. Für einige Überprüfungen sind jedoch eventuell Kenntnisse der Datenbankstruktur oder der Berechtigungen eines Benutzers erforderlich, die nur durchgeführt werden können, wenn das gesamte Formular ausgefüllt ist. Wie ich in ▶ Kapitel 7 erläutere, sollte die Überprüfung sicherheitskritischer Felder auch eine serverseitige Operation sein.

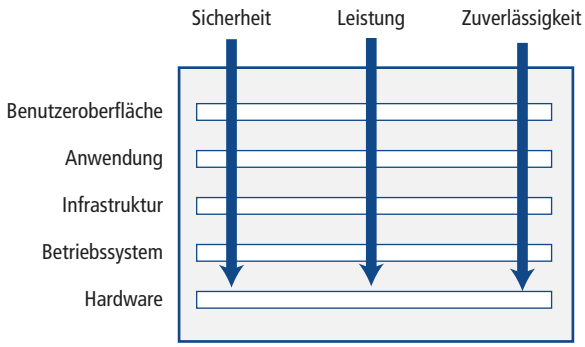

Abbildung 4.9: Cross-Cutting Concerns

Cross-Cutting Concerns (CCC) sind systemische Aufgaben, d.h., sie betreffen das gesamte System. In einer mehrschichtigen Architektur wirken sich CCCs auf alle Ebenen des Systems aus ebenso wie auf die Art und Weise, wie Menschen das System nutzen. ▶ Abbildung 4.9 zeigt drei CCCs – Sicherheit, Leistung und Zuverlässigkeit –, die für Softwareprodukte wichtig sind.
CCCs unterscheiden sich völlig von den funktionalen Zuständigkeiten, die durch Schichten in einer Softwarearchitektur dargestellt werden. Jede Schicht muss die CCCs berücksichtigen und aufgrund dieser Zuständigkeiten kommt es zwangsläufig zu Wechselwirkungen zwischen den Schichten. Diese CCCs erschweren es, die Systemsicherheit nach der Entwicklung zu verbessern. ▶ Tabelle 4.5 erklärt, warum Sicherheit nicht in einer einzelnen Komponente oder Schicht lokalisiert werden kann.

Sicherheitsarchitektur
Auf verschiedenen Schichten werden unterschiedliche Technologien eingesetzt, wie z.B. eine SQL-Datenbank oder ein Firefox-Browser. Angreifer können versuchen, Schwachstellen in diesen Technologien zu nutzen, um sich Zugang zu verschaffen. Daher benötigen Sie Schutz vor Angriffen auf jeder Ebene sowie Schutz auf niedrigeren Ebenen im System vor erfolgreichen Angriffen, die auf höheren Ebenen stattgefunden haben. Wenn es in einem System nur eine einzige Sicherheitskomponente gibt, stellt dies eine kritische Systemschwachstelle dar. Falls alle Sicherheitskontrollen durch diese Komponente laufen und diese nicht mehr ordnungsgemäß funktioniert oder durch einem Angriff kompromittiert ist, dann haben Sie keine zuverlässige Sicherheit in Ihrem System. Indem Sicherheit auf allen Schichten implementiert ist, wird Ihr System widerstandsfähiger gegen Angriffe und Softwarefehler (erinnern Sie sich an das *Rogue One*-Beispiel weiter vorne im Kapitel).

Tabelle 4.5: Sicherheit als CCC

Nehmen wir an, Sie sind ein Softwarearchitekt und wollen Ihr System in einer Reihe von Schichten organisieren, um die Komplexität zu kontrollieren. Sie stehen dann vor der allgemeinen Frage „Wo fange ich an?“ Glücklicherweise haben viele Softwareprodukte, die über das Web ausgeliefert werden, eine weit verbreitete Schichtenstruktur, die Sie als Ausgangspunkt für Ihr Design verwenden können. Diese allgemeine Struktur ist in ▶ Abbildung 4.10 dargestellt. Die Funktionalität der Schichten in dieser generischen Schichtenarchitektur ist in ▶ Tabelle 4.6 erläutert.

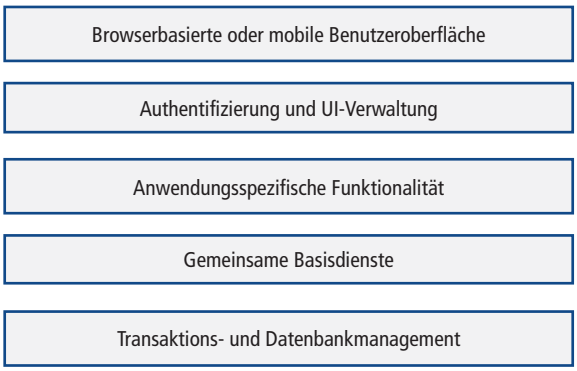

Abbildung 4.10: Eine allgemeine Schichtenarchitektur für eine webbasierte Anwendung

Schicht	Erläuterung
Browserbasierte oder mobile Benutzeroberfläche	Eine Webbrowser-Systemschnittstelle, in der häufig HTML-Formulare verwendet werden, um Benutzereingaben zu erfassen. JavaScript-Komponenten für lokale Aktionen, wie z. B. die Eingabevalidierung, sollten ebenfalls auf dieser Ebene integriert werden. Alternativ kann auch eine mobile Schnittstelle als App implementiert werden.
Authentifizierung und UI-Verwaltung	Eine Verwaltungsschicht für die Benutzeroberfläche, die Komponenten für die Benutzerauthentifizierung und die Erstellung von Webseiten beinhalten kann.
Anwendungsspezifische Funktionalität	Eine „Anwendungsschicht", die die Funktionalität der Anwendung bereitstellt. Manchmal kann dies auf mehr als eine Ebene erweitert werden.
Gemeinsame Basisdienste	Eine Schicht für gemeinsame Dienste, die Komponenten enthält, die Dienste bereitstellen, die von den Komponenten der Anwendungsschicht verwendet werden.
Datenbank- und Transaktionsmanagement	Eine Datenbankschicht, die Dienste wie Transaktionsmanagement und Wiederherstellung bereitstellt. Wenn Ihre Anwendung keine Datenbank verwendet, ist diese Schicht möglicherweise nicht erforderlich.

Tabelle 4.6: Funktionalität verschiedener Schichten einer webbasierten Anwendung

Für webbasierte Anwendungen können die in ▶ Abbildung 4.10 dargestellten Schichten der Ausgangspunkt für Ihren Zerlegungsprozess sein. In der ersten Phase wird darüber nachgedacht, ob dieses Fünf-Schichten-Modell das richtige ist oder ob Sie mehr oder weniger Schichten benötigen. Ihr Ziel sollte es sein, dass Schichten logisch zusammenhängend sind, sodass alle Komponenten einer Schicht etwas gemeinsam haben. Dies kann bedeuten, dass Sie für Ihre anwendungsspezifische Funktionalität eine oder mehrere zusätzliche Schichten benötigen. Manchmal möchten Sie vielleicht eine Authentifizierung in einer separaten Schicht haben und manchmal ist es sinnvoll, gemeinsame Dienste innerhalb der Schicht für das Datenbankmanagement zu integrieren.
Sobald Sie herausgefunden haben, wie viele Schichten Ihr System benötigt, können Sie beginnen, diese Schichten zu füllen. Meiner Erfahrung nach ist der beste Weg dabei, das gesamte Team einzubeziehen und verschiedene Zerlegungen auszuprobieren, um deren Vor- und Nachteile zu verstehen. Dies ist ein Prozess von Versuch und Irrtum – man hört auf, wenn man ein Ergebnis hat, das nach einer praktikablen Zerlegungsarchitektur aussieht.
Die Diskussion über die Systemzerlegung kann von grundlegenden Prinzipien gesteuert werden, die für den Entwurf Ihres Anwendungssystems gelten sollten. Mithilfe dieser Prinzipien werden Ziele abgesteckt, die Sie erreichen möchten. Anschließend können Sie die Entscheidungen über die Architekturplanung anhand dieser Ziele bewerten. ▶ Tabelle 4.7 zeigt zum Beispiel die Prinzipien, die wir bei dem Entwurf der iLearn-Systemarchitektur für am wichtigsten hielten.

Prinzip	Erläuterung
Austauschbarkeit	Es sollte möglich sein, dass Benutzer Anwendungen im System durch Alternativen ersetzen und neue Anwendungen hinzufügen können. Daher sollten die in der Liste enthaltenen Anwendungen nicht fest im System verankert sein.
Erweiterbarkeit	Es sollte möglich sein, dass Benutzer oder Systemadministratoren eigene Versionen des Systems erstellen können, die das „Standardsystem" erweitern oder einschränken.
Altersgerecht	Alternative Benutzeroberflächen sollten unterstützt werden, damit altersgerechte Schnittstellen für Schüler der unterschiedlichen Schulstufen geschaffen werden können.
Programmierbarkeit	Es sollte für den Benutzer einfach sein, seine eigenen Anwendungen zu erstellen, indem er bestehende Anwendungen im System verknüpft.
Minimaler Arbeitsaufwand	Benutzer, die das System nicht ändern möchten, sollten keine zusätzlichen Arbeiten durchführen müssen, damit andere Benutzer Änderungen vornehmen können.

Tabelle 4.7: Architektonische Entwurfsprinzipien von iLearn

Unser Ziel bei der Entwicklung des iLearn-Systems war es, ein anpassungsfähiges, universelles System zu schaffen, das leicht aktualisiert werden konnte, wenn neue Lernwerkzeuge verfügbar wurden. Das heißt, es musste möglich sein, Komponenten und Services im System zu ändern und zu ersetzen (Prinzip 1 und 2). Da das Alter der potenziellen Systembenutzer von 3 bis 18 Jahren reichte, mussten wir altersgerechte Benutzeroberflächen bereitstellen und die Auswahl einer Schnittstelle erleichtern (Prinzip 3). Prinzip 4 trägt ebenfalls zur Systemanpassung bei und Prinzip 5 wurde aufgenommen, um sicherzustellen, dass diese Anpassungsfähigkeit keine negativen Auswirkungen auf Benutzer hat, die sie nicht benötigen.

Leider kann Prinzip 1 manchmal im Widerspruch zu Prinzip 4 stehen. Wenn Sie Benutzern erlauben, neue Funktionen durch Kombinieren von Anwendungen zu erstellen, funktionieren diese kombinierten Anwendungen möglicherweise nicht, wenn eine oder mehrere der elementaren Anwendungen ersetzt werden. Mit dieser Art von Konflikten muss man sich häufig beim Architekturentwurf auseinandersetzen.

Diese Prinzipien führten uns zu der architektonischen Entwurfsentscheidung, dass das iLearn-System serviceorientiert sein sollte. Jede Komponente im System ist ein Service. Jeder Service ist potenziell ersetzbar und neue Dienste können durch die Kombination bestehender Dienste erstellt werden. Für Schüler unterschiedlichen Alters können verschiedene Services mit vergleichbarer Funktionalität angeboten werden.

Die Nutzung von Diensten bedeutet, dass der von mir oben identifizierte potenzielle Konflikt weitgehend vermeidbar ist. Wenn ein neuer Service unter Verwendung eines bestehenden Dienstes erstellt wird und anschließend andere Benutzer eine Alternative einbringen möchten, können sie dies tun. Der ältere Service kann im System beibehalten werden, sodass Benutzer dieses Dienstes keinen größeren Aufwand betreiben müssen, weil ein neuerer Dienst eingeführt wurde.

Wir gingen davon aus, dass nur ein kleiner Teil der Anwender an der Programmierung eigener Systemversionen interessiert ist. Daher haben wir uns entschieden, einen Standardsatz von Anwendungsdiensten anzubieten, der einen gewissen Grad an Integration

mit anderen Diensten aufweist. Wir haben erwartet, dass sich die meisten Anwender auf diese Services verlassen und sie nicht ersetzen wollten. Integrierte Anwendungsdienste wie Blogging- und Wiki-Services könnten so konzipiert sein, dass sie Informationen austauschen und übliche gemeinsame Dienste nutzen. Einige Benutzer möchten vielleicht andere Services in ihre Umgebung einbringen, daher haben wir auch Dienste zugelassen, die nicht eng mit anderen Systemdiensten integriert waren.
Wir haben uns entschieden, drei Arten der Integration von Anwendungsdiensten zu unterstützen:

1. *Vollständige Integration:* Services wissen von anderen Diensten und können über ihre APIs mit diesen kommunizieren. Services können Systemdienste und eine oder mehrere Datenbanken gemeinsam nutzen. Ein Beispiel für einen vollständig integrierten Service ist ein speziell programmierter Authentifizierungsdienst, der die Anmeldeinformationen von Systembenutzern überprüft.

2. *Teilintegration:* Dienste können Servicekomponenten und Datenbanken gemeinsam nutzen, wissen aber nichts von anderen Anwendungsdiensten und können nicht direkt mit diesen kommunizieren. Ein Beispiel für einen teilintegrierten Dienst ist ein Wordpress-Service, bei dem das Wordpress-System geändert wurde, um die standardmäßigen Authentifizierungs- und Speicherdienste im System zu verwenden. Office 365, das mit lokalen Authentifizierungssystemen integriert werden kann, ist ein weiteres Beispiel für einen teilintegrierten Service, den wir in das iLearn-System aufgenommen haben.

3. *Unabhängig:* Diese Dienste verwenden keine gemeinsamen Systemdienste oder Datenbanken und sie wissen auch nichts von anderen Diensten im System. Sie können durch jeden anderen vergleichbaren Service ersetzt werden. Ein Beispiel für einen unabhängigen Service ist ein Fotoverwaltungssystem, das seine eigenen Daten verwaltet.

Das von uns entworfene Schichtmodell für das iLearn-System ist in ▶ Abbildung 4.11 dargestellt. Um die „Austauschbarkeit“ von Anwendungen zu unterstützen, haben wir das System nicht um eine gemeinsame Datenbank herum aufgebaut. Wir sind jedoch davon ausgegangen, dass vollständig integrierte Anwendungen gemeinsame Services wie Speicherung und Authentifizierung nutzen würden.

Copyright