4 5C Design

So grease up your baby for the ball on the hill, polish them rockets now and swallow those pills.
Monster Magnet

Bevor wir zu den Details des 5C-Design-Modells kommen, das ich im Februar 2019 als Architektur-Spicker (*https://www.architektur-spicker.de*) erstmals veröffentlicht habe, möchte ich noch ein paar Worte zu meiner diesbezüglichen Motivation verlieren. Einerseits hatte ich in Abschnitt 2.5 erklärt, warum ich die SOLID-Designprinzipien als nicht mehr zeitgemäß betrachte. Im Vorwort habe ich meine Verwunderung über den Microservice-Hype bereits angedeutet. Daraus entstand mein Wunsch nach einer Prinzipiensammlung, die einen modernen Ersatz für SOLID darstellt. Dabei hatte ich mir das Ziel gesetzt, dass man daraus alles ableiten kann, was heutzutage als zeitgemäße Architektur angesehen wird. Hierzu zählt natürlich auch der Microservice-Architekturstil. Allerdings sollte es nicht mehr nötig sein, konkret einem Hype zu folgen. Mein Wunsch war, dass man mit 5C relativ einfach in der Lage sein würde, die passenden Muster der Zerlegung für eine Softwarearchitektur zu finden. Sie können 5C als eine Sammlung von Prinzipien sehen oder auch als eine Art Vorgehensmodell, bei dem Sie eben die Zerlegung in diesen fünf Schritten festlegen.

Das Ziel ist, Entwicklern von Software dauerhaft die Kontrolle über ihr Produkt zu gewährleisten. Ein effizientes Design nach 5C ist nichts, was ein Compiler oder ein Interpreter brauchen würde. Diesen wäre es nämlich vollkommen egal, wie gut strukturiert eine Software ist. 5C ist vielmehr ein Modell, um komplexe Software effizient in ihre einzelnen Teilbereiche zu zerlegen, sodass diese vom menschlichen Verstand noch gut erfasst und weiterentwickelt werden können. Dass dies ein großes Thema ist, beweisen die vielen Legacy-Systeme, welche man rund um den Globus in vielen großen Unternehmen heutzutage findet. Ein Legacy-System zeichnet sich dadurch aus, dass in gewisser Weise ein Kontrollverlust durch die Wartungsmannschaft droht. Dieser kündigt sich meist frühzeitig an durch Fehleranfälligkeit in der Wartungstätigkeit oder auch sinkende Produktivität.

Über Flexibilität

Einmal hatte mir ein Architekt erklärt, er erreicht bei Green-Field-Projekten immer dadurch eine maximale Flexibilität, indem er der Entwicklungsmannschaft keinerlei Einschränkungen bei deren Arbeit auferlegt. Würde er deren Arbeit einschränken, würden dem Team schließlich weniger Handlungsoptionen zur Verfügung stehen und somit wären sie – per definitionem – weniger flexibel. Diese Aussage stand sofort in einer gewissen Diskrepanz zu meinem Bauchgefühl. Überraschend war für mich, dass es gar nicht so einfach

war, dies argumentativ zu widerlegen. Tatsächlich scheint es so, dass ein nachhaltiger Architekturentwurf zuerst immer einschränkend wirken muss. Ich werde als erfahrener Architekt am Anfang dem Team immer Handlungsoptionen nehmen, indem ich gewisse Vorgaben mache. Das Ziel sollte dabei aber natürlich sein, dem Team die Kontrolle über die Software dauerhaft zu gewährleisten. Durch diese spätere Kontrolle wiederum ergeben sich Handlungsoptionen, und diese Flexibilität ist es, welche wir durch 5C auf Dauer haben möchten. Tatsache ist aber, dass wir dafür zunächst investieren müssen. Anfangs werden wir uns beschränken, wenn wir etwas wie 5C befolgen. Betrachten wir 5C also als eine Investition in die Zukunft, die allerdings tatsächlich nicht immer angebracht ist. In seltenen Fällen mag es Systeme geben, für die spätere Wartbarkeit gar nicht erst nötig ist, weil beispielsweise nur eine kurze Einsatzzeit geplant ist. In den allermeisten Fällen ist es aber nicht wünschenswert, einen späteren Kontrollverlust von Anfang an einzuplanen.

No Silver Bullet

1986 veröffentlichte ein gewisser Fred Brooks, seines Zeichens Gewinner des Turing Awards, einen vielbeachteten Artikel namens „No Silver Bullet – Essence and Accident in Software Engineering“ [Bro86]. Darin argumentiert er, dass es immer so etwas wie essenzielle Komplexität gibt, welche dem Problem, das es zu lösen gilt, innewohnt. Egal wie gut Ihr Entwurf für eine Software wird, die ein bestimmtes Problem löst, sie wird niemals einfacher sein können als diese Komplexität, die der Herausforderung selbst bereits innewohnt. Oft kommt es aber zu Komplexität, die im Zuge der Lösung erst entsteht. Diese nennt er „accidental“, also versehentlich. 5C ist ein Modell, das helfen soll, diese unbeabsichtigte Komplexität so gut es geht im Zaum zu halten.

Bevor wir ins Detail gehen, vorab noch ein Überblick über das 5C-Modell:

1. **CUT – Richtig schneiden:** Je unabhängiger ein Baustein ist, desto einfacher wird er zu handhaben sein. Schneide Bausteine so, dass sie sich möglichst gut voneinander abgrenzen.
2. **CONCEAL – Verbergen:** Verbirg so viel der internen Struktur eines Bausteins und der Art der Umsetzung vor der Außenwelt wie möglich.
3. **CONTRACT – Schnittstelle festlegen:** Entwerfe Schnittstellen so, dass eine möglichst reibungslose Interaktion zwischen dem Baustein und seinen Consumern möglich ist.
4. **CONNECT – Verbinden:** Durch Verwendung einer Schnittstelle eines anderen Bausteins kommt es immer zu Abhängigkeiten. Plane explizit, zwischen welchen Bausteinen es welche Art von Abhängigkeit geben soll.
5. **CONSTRUCT – Aufbauen:** Eine Bausteinstruktur kann auf einer Ebene selbst wieder unübersichtlich werden. Baue Systeme höherer Komplexität durch Zusammenfassen von Bausteinen einer Ebene zu einem neuen Baustein einer nächsthöheren Ebene. Dabei sind weiterhin dieselben Handlungsmaximen anzuwenden.

4.1 CUT – Richtig schneiden

- **Definition:** Je unabhängiger ein Baustein ist, desto einfacher wird er zu handhaben sein. Schneide Bausteine so, dass sie sich möglichst gut voneinander abgrenzen.
- **Siehe auch:** P02-SoC, P03-Cohesion, P04-SLA, P06-IOSP, A02-GodObj
- **Messung durch:** Zyklomatische Komplexität nach McCabe (Abschnitt 5.1.2), Relational-Cohesion (Abschnitt 5.1.4.1), LCOM (Abschnitt 5.1.4.2), ACD (Abschnitt 5.1.6.3)
- **Später:** P05-DumbPipe, Domain-Driven-Design (Kapitel 6), Conway's Law (Abschnitt 8.1)

Zunächst geht es einmal darum, Grenzen zwischen den einzelnen Bausteinen an den richtigen Stellen zu ziehen. Je unabhängiger die einzelnen Bausteine voneinander sind, desto einfacher wird später das gesamte System zu handhaben sein. Die Kunst ist, die Stellen zu identifizieren, wo sich Bausteine idealerweise voneinander abgrenzen und entkoppeln lassen. Einige Indizien dafür haben wir in den ersten Kapiteln dieses Buchs bereits kennengelernt. Diese lassen sich teilweise als Ziele sehen, aber andererseits auch als Richtlinien, welche man von Beginn an dabei beachten sollte. Und zwar:

- Der so abgetrennte Baustein macht nur eine Sache. Es braucht nur wenige Worte, um die Aufgabe des Bausteins zu beschreiben.
- Der Baustein ist nicht zu groß, sodass er noch am Stück verstanden werden kann, oder er zerlegt sich selbst wiederum in weitere Bausteine, die das ermöglichen.
- Der so gebildete Baustein kann seine Aufgabe möglichst autonom erledigen.
- Die Aufgabe des Bausteins lässt sich anhand seiner ein- und ausgehenden Schnittstellen gut beschreiben und klar definieren.
- Der Baustein ist gut testbar an seinen ein- und ausgehenden Schnittstellen. Der Aufwand, andere Bausteine für einen isolierten Test zu mocken, hält sich in Grenzen.
- Der Baustein ist kohäsiv, d. h., er hat einen größeren Zusammenhalt nach innen als nach außen.
- Der Grad an Kopplung mit anderen Bausteinen auf derselben Abstraktionsebene ist angemessen gering.
- Innerhalb des abgetrennten Bausteins gibt es keine Teilbereiche, die nichts miteinander zu tun haben, denn sonst würde sich eine weitere Abtrennung anbieten.
- Mögliche einzelne Änderungswünsche sind an vergleichsweise wenigen, im Idealfall nur einem der Bausteine durchführbar.
- Die Gefahr, dass andere Bausteine durch Änderungen an diesem Baustein destabilisiert werden könnten, ist möglichst gering.

4.1.1 Vertikale vs. Horizontale Abtrennung

Weit verbreitet sind zwei grundsätzliche Ansätze, um Bausteine abzutrennen. Der eine trennt nach Fachlichkeit und bildet sogenannte Vertikale Strukturen, der andere nach Technik, was üblicherweise als Horizontale Struktur oder Layer (bzw. Schichten) dargestellt wird. Die Ansätze besitzen verschiedene Vor- und Nachteile, wobei man sagen muss, dass üblicherweise die Eigenständigkeit der dadurch abgetrennten Bausteine höher ist, wenn nach Fachlichkeit vertikal abgetrennt wird. Betrachten wir dazu zunächst die horizontale Schichtenstruktur in Bild 4.1. Wenn Sie hier einem Feature ein neues Feld hinzufügen möchten, so wird das üblicherweise eine Kaskade von Änderungen in den anderen Schichten nach sich ziehen. Zuerst wird das Feld in der DB addiert, danach in der Persistenzschicht, dann in der Serviceschicht und zuletzt im Userinterface. Daran sieht man, wie wenig eigenständig die einzelnen Strukturen sind.

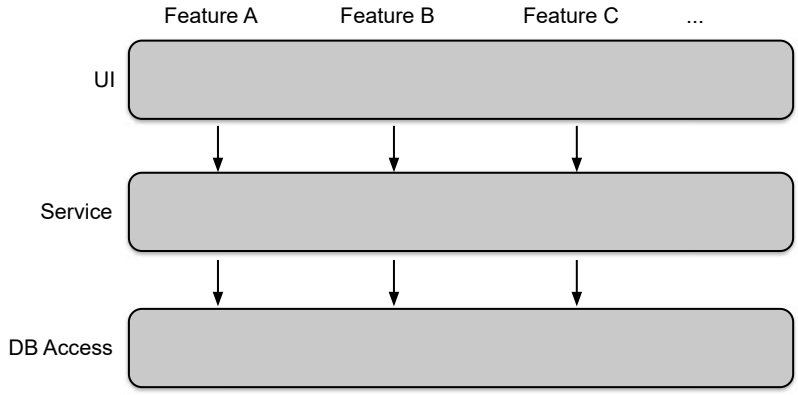


Bild 4.1 Typisches Beispiel für eine horizontale Schichtenarchitektur

Dabei darf man allerdings nicht die Vorteile übersehen, die so eine Architektur zweifelsohne hat. Querschnittliche Themen werden dadurch einfacher zu handhaben sein. Wenn Sie eine gewisse Konsistenz bei der Umsetzung von Mustern im UI oder in der Persistenz haben wollen, ist dies mit so einer Form der Struktur bestimmt einfacher zu erreichen. Auch bedeuten Layer nicht, dass es keinerlei fachliche Strukturen gibt, da diese ja als Substrukturen innerhalb der einzelnen Schichten sehr wohl noch gebildet werden können. Trotzdem sollte man zumindest darüber nachdenken, vertikale (und somit primär fachliche) Strukturen zu bilden, wie sie in Bild 4.2 dargestellt sind. Die Vor- und Nachteile hier sind das genaue Gegenteil zur Schichtenarchitektur aus Bild 4.1.

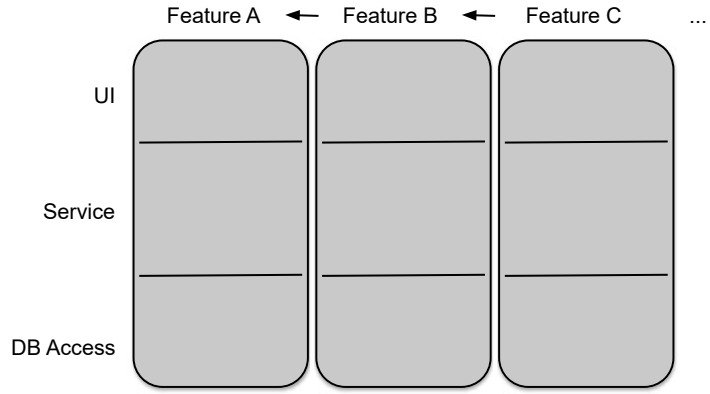
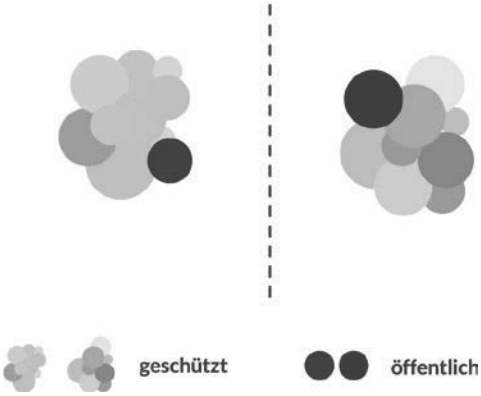


Bild 4.2 Die Alternative zur Schichtenarchitektur ist jede Form der vertikalen fachlichen Strukturierung.

Dass es viele Anwendungsfälle für vertikale bzw. horizontale Architekturen gibt, sollte nicht davon ablenken, dass prinzipiell natürlich jede Form von Modularisierung gewählt werden kann. Also auch solche, die nicht in dieses streng zweidimensionale Muster aus Fachlichkeit und Technik übertragen werden können.

4.2 CONCEAL – Verbergen



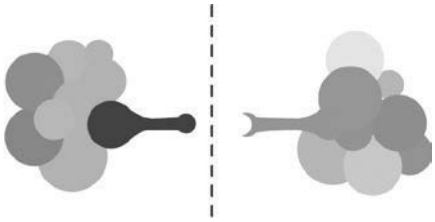
- **Definition:** Verbirg so viel der internen Struktur eines Bausteins und der Art der Umsetzung vor der Außenwelt wie möglich.
- **Siehe auch:** P07-Hide, P08-Demeter
- **Messung durch:** Visibility-Metriken (5.1.3), Attribute- bzw. Method-Hiding-Factor (Abschnitt 5.1.8.1)
- **Später:** A07-MaxReuse

Naiverweise könnte man meinen, jede Logik, welche man z. B. in Form einer Methode kapselt, sollte der Außenwelt zur Verfügung gestellt werden. Schließlich kann man doch nicht wissen, ob diese Logik später nicht noch jemand anders brauchen wird, oder? Wenn dieser jemand sie dann später sucht, steht sie ihm schon zur Verfügung und er kann sie wieder-

verwenden. Dies ist allerdings keine gute Idee. In so einem Fall sollte immer hinterfragt werden, warum ein anderer Baustein diese Logik benötigt, weil dies nicht selten ein Indiz für einen schlecht erfolgten CUT ist. Im Zweifelsfall ist es immer besser, Substrukturen vor der Außenwelt zu verbergen. Wenn sie verborgen sind, kann es nämlich zu gar keinen Abhängigkeiten anderer Bausteine kommen. Damit bleibt dieser Aspekt dauerhaft flexibel und man kann ihn jederzeit ändern oder löschen, ohne dass die Welt außerhalb davon betroffen wäre. Das Ziel von CONCEAL ist also die Verhinderung von Abhängigkeiten, die vermutlich gar nicht gewünscht sind. Falls man später bemerkt, dass ein Aspekt doch auch außerhalb benötigt wird, so kann er immer noch sichtbar gemacht und zur Verfügung gestellt werden.

In Kapitel 7 werden wir die sogenannten Distributed-Systems oder Verteilten Systeme kennenlernen. Ein solcher Entwurf besitzt viele Vor- bzw. Nachteile gegenüber jeder Form von Deployment-Monolithen, auf die wir dort dann noch näher eingehen werden. Gerne wird argumentiert, dass man dies alleine schon wegen des Information-Hiding-Aspekts tun sollte. Wenn ein Implementierungsdetail hinter dem Netzwerk in einer anderen Deployment-Unit vor dem Consumer verborgen ist, so ist dies zweifelsohne eine Form von Information Hiding, allerdings ist es als Mittel dafür keineswegs alternativlos, auch wenn Anhänger des Microservice-Hypes dies gerne so sehen möchten. Tatsächlich bilden die Bordmittel der Programmiersprachen, wie private Definitionen in Klassen oder auch package-protection und JPMS-Module in Java in Kombination mit Toolunterstützung und Metriken, hervorragende Alternativen (siehe Abschnitt 3.6).

4.3 CONTRACT – Schnittstelle festlegen



- **Definition:** Entwerfe Schnittstellen so, dass eine möglichst reibungslose Interaktion zwischen dem Baustein und seinen Consumern möglich ist.
- **Siehe auch:** P09-Open, P10-DbC, P11-Liskov, P12-InSeg, P13-AoI, A01-MisInh
- **Messung durch:** Weighted Interface Complexity (siehe Abschnitt 5.1.10)
- **Später:** P14-Tolerant, P15-Robust, A05-Cannonical

Um Funktionalität anderen Bausteinen zur Verfügung zu stellen, sollte ein Baustein immer eine dedizierte Schnittstelle anbieten. Der erste Adressat einer Schnittstelle ist dabei immer der Entwickler des Bausteins, der später der Consumer der Schnittstelle sein wird. Beim Entwurf sollte man daher berücksichtigen, dass eine Schnittstelle in erster Linie für Menschen entworfen wird und erst in zweiter Linie für die Technik. Schnittstellen sind außerdem neuralgische Punkte in einer Architektur, wo es nicht selten zu Fehlersituationen kommt. Eine Schnittstelle, die von Entwicklern einfach gut verstanden werden kann, führt

üblicherweise auch seltener zu solchen Fehlersituationen. Im Idealfall erfüllt sie folgende Kriterien:

- Sie ist **ausreichend definiert**. Bei numerischen Werten ist z. B. klar, was der Inhalt bedeutet. Bei Datums- oder Zeitfeldern ist das Format klar definiert.
- Sie ist selbsterklärend oder **ausreichend dokumentiert**. Die diversen Bezeichnungen werden passend gewählt. Man wird (z. B. durch Hypermedia) von einem Punkt zum anderen geführt. Oder es gibt eine ausreichende Dokumentation.
- **Konventionen werden eingehalten**. Solche kann es einerseits auf fachlicher, andererseits auf technischer Ebene geben. Sie können weltweit gelten, für die jeweilige Domäne, nur im Unternehmen oder innerhalb dieser Schnittstelle. Das Risiko für Überraschungen wird möglichst minimiert [Lau19].
- Durch ihre Verwendung kann der **Baustein nicht destabilisiert** werden. Durch Missbrauch, wie z. B. Abfragen von zu vielen Daten, wird der Betrieb des Bausteins nicht gefährdet.
- Sie sollte **angemessen schmal** (P07-Hide) bzw. entsprechend der unterschiedlichen Verantwortlichkeiten aufgeteilt sein (P12-InSeg).

Hyrum's Gesetz

Egal, wie gut Sie alle Aspekte einer Schnittstelle dokumentieren, es wird auch immer informelle Aspekte einer Schnittstelle geben. Es werden außerdem früher oder später Consumer Abhängigkeiten zu diesen Aspekten entwickeln [Lau19]. Das folgende Beispiel soll dies illustrieren: Angenommen, eine Schnittstelle liefert alle Adressen einer Person, wobei diese verschiedene Bedeutungen haben, wie den Hauptwohnsitz, Nebenwohnsitze oder die Büroadresse. Der Baustein verhält sich so, dass er anfangs immer den Hauptwohnsitz an erster Stelle in der Liste liefert. Wenn nun ein Consumer den Hauptwohnsitz ermittelt, indem er das erste Element aus der Liste entnimmt, wird dies so lange gut gehen, bis der Provider sein Verhalten diesbezüglich ändert. Die formelle Schnittstellendefinition wird dies vermutlich zulassen. Ein gewisser Hyrum Wright hat dies einmal ganz gut auf den Punkt gebracht, als er sein Gesetz (L01-Hyrum) formulierte:

> *With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.*

4.3.1 Antipattern

4.3.1.1 Kanonisches Schnittstellenmodell

Bereits oft gescheitert und trotzdem immer wieder probiert, ist der Versuch, die Formate, mit denen Module Daten austauschen, über die gesamte Systemlandschaft hinweg zu vereinheitlichen. Ein Unterfangen, welches in seiner Komplexität dabei unterschätzt wird, weil es durch die Annahme dieser global definierten Datenformate Kopplung zwischen allen Modulen und Bausteinen einer Systemlandschaft herstellt. Damit wäre dann das Prinzip der losen Kopplung auf drastische Art und Weise verletzt. Oder um Eric Evans, seines Zeichens Erfinder des Domain Driven Design, zu zitieren [Eva03]:

Total unification of the domain model for a large system will not be feasible or cost-effective.

Auch wenn man den initialen Wurf eines solchen Modells hinbekommen sollte, bedeutet es auch in weiterer Folge bei jeder Änderung an einem der dabei angebundenen Module enorme Aufwände zur Abstimmung. Nach unzähligen Diskussionen, welche ich in meiner Karriere schon zu dem Thema geführt habe, scheint es mir hauptsächlich zwei Gründe zu geben, warum dies manchmal nach wie vor irrtümlicherweise für sinnvoll erachtet wird:

- Es wird als notwendiges Mittel zum Zweck gesehen, welches es den Services ermöglichen soll, miteinander zu kommunizieren.
- Durch eine prinzipielle Adapterschicht zu einem kanonischen Modell werden die einzelnen konkreten Service-Implementierungen angeblich komplett austauschbar.

Zunächst möchte ich auf das erste Argument eingehen. Diesem liegt die irrtümliche Annahme zugrunde, dass einzelne Business-Themen auf mehrere Services verteilt zu implementieren seien. Stattdessen sollte man aber versuchen, jedes der fachlichen Themen (also Subdomänen im Sinne des DDD – siehe Kapitel 6) in möglichst einem Service zu kapseln. Wenn Sie das hinbekommen, so modelliert jedes Modul (oder Service) den jeweiligen Teil der Problemdomäne, für den es zuständig ist. Es bleiben allerdings immer manche Aspekte, die von mehr als einem Modul bearbeitet werden. Hier ist es trotzdem oft besser, wenn jedes Modul diese für sich selbst modelliert, und zwar aus zwei Gründen: einerseits, weil die einzelnen Subdomänen oft unterschiedliche Sichtweisen auf diese geteilten Aspekte haben, und andererseits, um Abstimmungsaufwände zwischen den einzelnen Teams, die die Module entwickeln, zu vermeiden.

Argument Nr. 2 für kanonische Modelle setzt ein Szenario voraus, wie es in Bild 4.3 dargestellt ist. Die Idee ist, dass die einzelnen Consumer auf den konkreten Provider immer über die globale Adapterschicht zugreifen, welche das konkrete Schnittstellenmodell A des Providers immer auf das kanonische Modell X konvertiert. Wenn man nun den Provider durch einen anderen austauschen möchte, der dann stattdessen ein anderes Schnittstellenmodell B liefern würde, so ist nichts weiter zu tun, als den Adapter für diesen Provider in der globalen Schicht auszuwechseln, welcher dann ein Mapping von B auf X macht, anstatt wie zuvor von A auf X. Die einzelnen Consumer wären davon demnach nicht betroffen. Soweit die Theorie. Valide ist dieses Argument allerdings keineswegs. Einerseits ist nicht gesagt, dass der neue Provider, was die Schnittstelle (wie Modell und Abläufe) angeht, überhaupt prinzipiell kompatibel ist. Und auch wenn dem so ist, macht es keinen Sinn, den Adapter vorwegzunehmen. Wenn es wirklich zu einem solchen gefürchteten Austausch des Providers kommt, so kann man später immer noch einen Adapter bauen, welcher das Mapping von B zu A übernimmt. Und diese Modelle müssen im gegebenen Fall zueinander konvertibel sein, denn wenn der Umweg B->X->A möglich ist, so ist natürlich im Endeffekt die Übersetzung von B zu A ebenfalls möglich (und sei es mittels eines Umwegs über X). Es handelt sich hierbei übrigens um einen Irrtum der Kategorie A03-MagicAbstract.

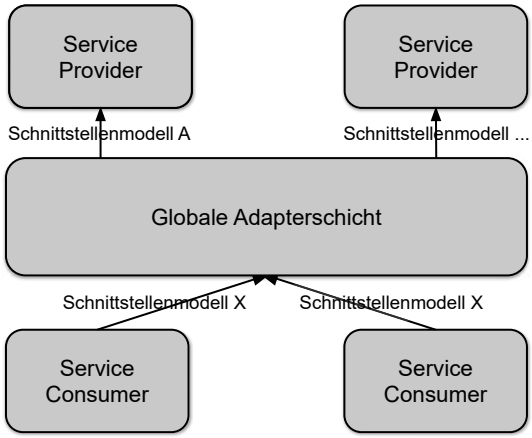


Bild 4.3 Das theoretische Einsatzszenario für ein kanonisches Modell

4.3.1.2 Versionierung

Verteilte Systeme bringen früher oder später immer die folgende Herausforderung mit sich: Schnittstellen müssen geändert werden, weil der Service, der diese anbietet, weiterentwickelt wird. Dabei stellt sich die Frage, ob und wie man auf die jeweiligen Consumer der eigenen API Rücksicht nimmt. Schließlich wurden die Consumer mit der alten Version des zu ändernden Providers getestet. Früher war es daher üblich, die Änderungen am Provider eine Zeitlang parallel zur alten Version anzubieten, bis alle Consumer von der alten auf die neue Version umgestellt (und getestet) wurden. Meist lief es aber so ab, wie in dem folgenden Beispiel:

- Modul A, aktuell in Version 1.1.0 deployed, stellt Erweiterungen online. Eine „Impact Analysis“ ergibt, dass davon potenziell die beiden Consumer B und C betroffen wären. Daher wird die Änderung als Version 1.2.0 parallel zur alten Version deployed. Die Teams von B und C unterliegen der Unternehmensrichtlinie, nach der sie innerhalb eines Quartals die Umstellung auf die neue Version machen müssen.
- Während Modul C bald auf die neue Version 1.2.0 von Modul A umstellt, sind die Benutzer von Modul B mit dessen Stabilität unzufrieden und somit wird hier alles darangesetzt, diese Probleme in den Griff zu bekommen. Irgendwelche Richtlinien der IT-Abteilung sind dem Kunden egal und daher ist das Team B die nächsten Monate mit dem Refactoring beschäftigt.
- Inzwischen müssen aber neue Features von Modul A raus und somit erstellt man eine weitere Version 1.3.0.
- Team B bemerkt inzwischen, dass ihre Probleme teilweise von einem Bug in Version 1.1.0 des Moduls A verursacht werden, und um den Betrieb nicht zu gefährden, wird kurzerhand eine neue Version von Modul A deployed, nämlich 1.1.1.

Nun laufen inzwischen drei Varianten von Modul A, nämlich 1.1.1, 1.2.0 und 1.3.0. Stellen Sie sich das Spiel mit einer Systemlandschaft aus 100 Modulen (bzw. Services) oder mehr über einen Zeitraum von mehreren Jahren vor. Sie sehen also, dass es keine gute Idee ist, Modulschnittstellen gezielt in verschiedenen Versionen anzubieten. Viel besser ist es statt-

dessen, eine neue Version einer Schnittstelle als rückwärtskompatiblen Evolutionsschritt zu deployen. Um das ohne schlaflose Nächte hinzubekommen, bieten sich die Pattern an, welche ich Ihnen später noch vorstellen werde, beispielsweise das Consumer Driven Contract Testing (Abschnitt 4.3.2.1) oder auch der Robustheitsgrundsatz (Abschnitt 4.3.2.2). Eine weitere Alternative ist es, jedesmal aufwendige System-Integrationstests durchzuführen. Auf das parallele Anbieten unterschiedlicher Versionen einer Schnittstelle sollte man nur zurückgreifen, wenn es keine andere Möglichkeit gibt. Und wenn man es tut, sollte man es zeitlich so gut es eben geht begrenzen.

Das beste Mittel, um der Problematik der sich ändernden externen Schnittstellen zu begegnen, ist nach wie vor das gute alte Information-Hiding (P07-Hide). Dinge, die gar nicht nach außen hin wirksam werden, können geändert werden, ohne dass es eine Auswirkung auf eine der externen Schnittstellen hat.

4.3.2 Pattern

4.3.2.1 Consumer Driven Contract Testing

Kommen wir nochmal auf das Dilemma aus Abschnitt 4.3.1.2 zu sprechen. Ein Modul oder Service möchte seine Schnittstelle ändern und wir möchten mit möglichst viel Selbstvertrauen und ohne den Betrieb zu gefährden die neue Version releasen. Das möglichst auch noch ohne große Aufwände zur Abstimmung dieser Änderung. Eine Möglichkeit ist natürlich, auf System-Integrationstests zu setzen, die aber teuer, aufwendig und teilweise schwierig durchzuführen sind. Eine einfachere Alternative dafür stellen die sogenannten Consumer Driven Contract Tests dar [Nyg07] (Bild 4.4). Die Idee ist, dass jeder Consumer dem Modul, welches er benutzt, auch einen Testfall zur Verfügung stellt. Dieser Testfall deckt die Erwartungen des Consumers an diesen Provider ab. Man dreht damit quasi die Richtung der Verantwortung um und überlässt den Consumern die Beweisführung, dass eine neue Version des Providers problemlos in Produktion gehen darf. Mit so einem Testfall

- sagt der Consumer: Wenn dieser Testfall durchläuft, so sind alle Erwartungen, welche ich an dich habe, erfüllt. Du kannst dann beruhigt jederzeit eine neue Release rausbringen;
- sagt der Provider: Wenn du als Consumer meine Schnittstelle verwenden möchtest, so darfst du das gerne tun. Ich verlange dafür allerdings von dir einen solchen Consumer Driven Contract Test, sodass meine Weiterentwicklung davon nicht gebremst wird.

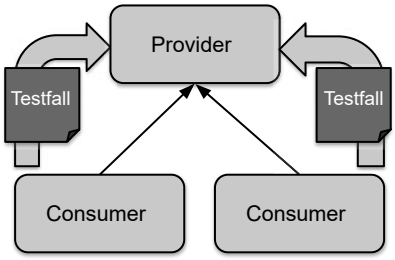


Bild 4.4
Consumer Driven Contract Testing zur organisatorischen Entkopplung der Entwicklung einzelner Services

4.3.2.2 Postel's Law/Robustheitsgrundsatz (Tolerant Reader)

Betrachten wir nochmal dieselbe klassische Herausforderung von Änderungen an Schnittstellen aus dem vorigen Abschnitt über Consumer Driven Contract Testing. Ein Provider möchte mit einer Änderung seiner Schnittstelle produktiv gehen. Wenn die betroffenen Consumer dabei in der Interpretation der Antwort des Providers nicht so streng sind, wird Ihnen die Weiterentwicklung des Providers ungleich leichter fallen. Dieser Grundsatz ist auch bekannt als das Tolerant Reader Pattern (P14-Tolerant). Ein Beispiel wäre eine Schnittstelle, die eine gewünschte Anzahl an Treffern als Input liefert. Die maximale Anzahl beträgt 50. Wenn ein Consumer nun gerne 100 Einträge haben möchte, kann man sich darauf beschränken, ein Warning mitzugeben anstatt eines Errors und die maximale Anzahl an 50 Treffern zu liefern.

Wenn dazu noch der Absender einer Nachricht sehr genau darauf achtet, was er absendet, um den von ihm garantierten Schnittstellen-Vertrag auch immer exakt einzuhalten, dann ergibt das in Summe das sogenannte Robustness Principle (P15-Robust) [Pos89]. Um bei unserem Beispiel zu bleiben, würde sich jeder Consumer bemühen, bei numerischen Werten sich immer innerhalb der definierten Grenzwerte zu bewegen, wenn er die Schnittstelle eines Providers benutzt (siehe Bild 4.5).

Ich denke allerdings, dass man dieses Prinzip nur abhängig von den definierten Qualitätszielen eines Produkts wählen sollte. Diese hatten wir in Abschnitt 1.2 kennengelernt. Wenn hohe funktionale Korrektheit höher bewertet ist als Verfügbarkeit und Wartbarkeit es sind, dann ist es vielleicht sogar ratsamer, als Provider einer Schnittstelle im Zweifelsfall eher mit einem Fehler zu reagieren. Nach dem Motto: besser abbrechen als etwas u. U. Fehlerhaftes zu tun.

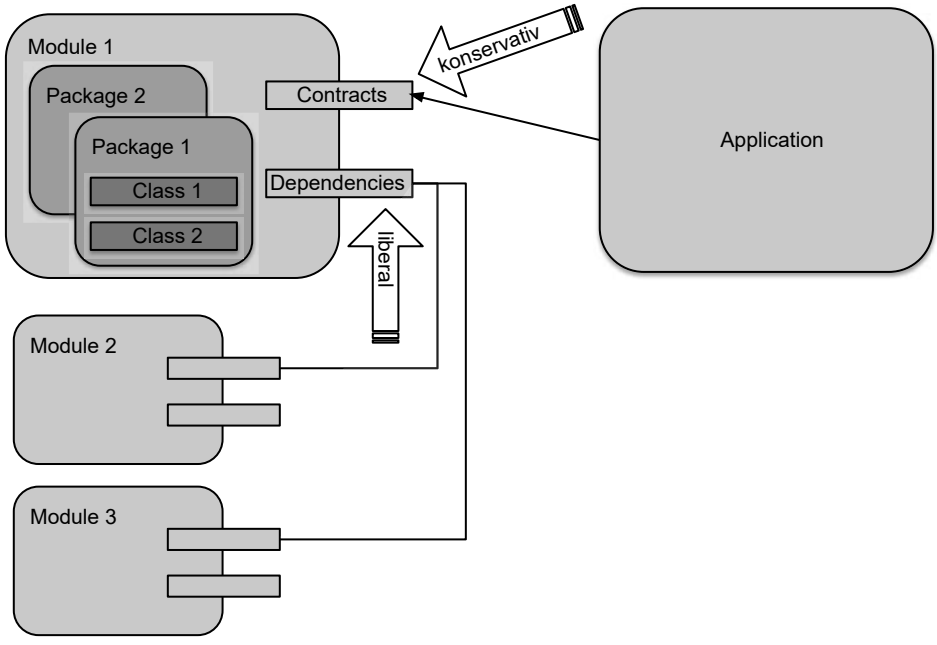
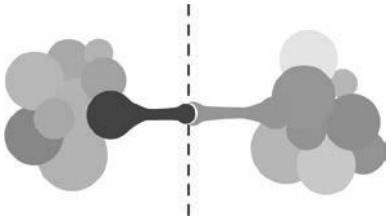


Bild 4.5 Postel's Law illustriert

4.4 CONNECT – Verbinden



- **Definition:** Durch Verwendung einer Schnittstelle eines anderen Bausteins kommt es immer zu Abhängigkeiten. Plane explizit, zwischen welchen Bausteinen es welche Art von Abhängigkeit geben soll.
- **Siehe auch**: P17-DepInv, P18-Loose, A04-Cyclic
- **Messung durch:**
  Relative Cyclicity (Abschnitt 5.1.7),
  John-Lakos Metriken (Abschnitt 5.1.6),
  NCCD (Abschnitt 5.1.6.5)
- **Später:** Don't repeat yourself (P16-DRY)

Sobald wir Schnittstellen definiert haben, können wir die im ersten Schritt abgetrennten Bausteine wieder miteinander verbinden. Der Grund dafür ist irgendeine Form der Interaktion, die nötig ist. So möchte der Consumer einer Schnittstelle beispielsweise eine Verarbeitung beim Provider anstoßen. Oft ist der Hintergrund einer solchen Verbindung aber einfach der Wunsch, gewisse Logik des Providers im Consumer wiederzuverwenden, weil man Codeduplikation (P16-DRY) vermeiden möchte.

Aber: Durch eine solche Verbindung kommt es immer zu Abhängigkeiten zwischen den beiden verbundenen Bausteinen. Manchmal kann es daher sein, dass die Redundanz billiger bzw. mit weniger Nachteilen verbunden ist als die Kopplung durch die Verbindung der Bausteine (T04-DRYvsIndependence). In solchen Fällen kann eine Redundanz die bessere Alternative darstellen.

In welchem Ausmaß ist man abhängig?

Bei Abhängigkeiten gibt es gewaltige Unterschiede. Manche wiegen schwer, andere sind wiederum weniger problematisch. Achten Sie beim Entwurf einer Softwarearchitektur vor allem auf die schwerwiegenden Abhängigkeiten und versuchen Sie, diese zu minimieren! Wie problematisch eine Abhängigkeit ist, ergibt sich aus einer Kombination der folgenden Faktoren:

- Ist die Abhängigkeit in irgendeiner Form einschränkend? Gibt es Dinge, die dadurch verunmöglicht werden?
- Wie viel Aufwand würde es bedeuten, etwas von dem man abhängig ist (wie eine Technologie oder ein Baustein als Provider einer Schnittstelle) durch etwas anderes auszutauschen?
- Wie viel Risiko bedeutet es, die Abhängigkeit wieder loszuwerden? Lassen sich die Tests, die im Zuge einer Ablöse nötig wären überhaupt gut durchführen? Wenn nicht, wäre es ein Indiz für ein hohes Risiko einer potenziell einmal nötigen Ablöse.
- Wie wahrscheinlich ist es, dass eine solche Änderung notwendig sein wird im Rahmen einer realistischen Lebensspanne des Systems? Dass man einen Technologiestandard, für den es verschiedene Implementierungen gibt, wird ersetzen müssen, ist z. B. weni-

ger wahrscheinlich, als eine proprietäre Technologie, die wenig verbreitet ist. Bausteine, die man nicht selber entwickelt und damit nicht unter Kontrolle hat, sollten ebenfalls kritischer betrachtet werden.

Bei den im folgenden Abschnitt 4.4.1 vorgestellten Punkten handelt es sich um die möglichen Abhängigkeiten, die durch eine Interaktion zwischen zwei Bausteinen entstehen können.

4.4.1 Formen der Kopplung

Laufzeitumgebung, Ausführungsort

Die andere Komponente muss auf derselben Maschine laufen, damit die Integration möglich ist. In so einem Fall wäre es schwierig, Fehler, welche sich auf den stabilen Betrieb der Laufzeitumgebung auswirken (Verbrauch von zu viel Speicher oder CPU), von den anderen Komponenten zu isolieren. Es bestünde dann die Gefahr, dass das gesamte System in Mitleidenschaft gezogen wird. Im nicht so extremen Fall gibt es gewisse andere Formen der Einschränkung des operativen Systems, die im Fall einer Wiederverwendung für die andere Komponente gelten würden. So kann es sein, dass die andere Komponente einfach nur im selben Rechenzentrum betrieben werden muss, was weniger einschränkend wäre.

Technologie

Einschränkungen bei der Technologiewahl der angebundenen Komponente. Dabei kann es sich um Einschränkungen handeln, die noch einen gewissen Spielraum zulassen (wie alle Systeme, welche eine REST API konsumieren können). Im Extremfall würde die andere Komponente genau dieselbe Technologie verwenden müssen.

Zeit

Die andere Komponente kann beispielsweise zu gewissen Zeitpunkten nicht angesprochen werden, bietet aber keine asynchrone Variante ihrer Schnittstelle an. Wenn die aufrufende Komponente die fremde Schnittstelle benötigt, um mit einer Verarbeitung fortfahren zu können, wäre sie somit für diese Zeit blockiert.

Daten und Formate

Dabei kann es bei der Kommunikation gewisse allgemeine Einschränkungen geben in Bezug auf die möglichen Datenformate, die geparst und verstanden werden, beispielsweise Datumsformate oder Header, die gesetzt werden müssen. Hauptsächlich bestehen diese Kopplungen aber einfach aufgrund des konkreten Datenformats der Schnittstelle, die verwendet wird. Wenn das fremde Schnittstellenformat außerdem noch überall in der eigenen Komponente zur Anwendung kommt, wird so eine Kopplung sogar noch stärker. Schließlich würde dann auch der Aufwand, die konkrete Schnittstelle durch eine andere zu ersetzen, entsprechend ansteigen. Durch Integration mit dem Composite-UI-Pattern (Abschnitt 4.4.1.1) oder Verwendung des Tolerant-Reader-Musters (Abschnitt 4.3.2.2) kann man diese Form der Kopplung teilweise abschwächen.

Wenn Sie zwei Komponenten integrieren, so tun Sie das immer auf die Art und Weise, welche die geringstmögliche Kopplung bedeutet (P18-Loose). Bei hohem Abstraktionsgrad wie der Makro-Architektur ist dies besonders wichtig! In der Mikro-Architektur oder Designebene kann dagegen auch ein hoher Grad an Kopplung (wie durch Code Reuse) noch akzeptabel sein.

Werfen wir einen Blick auf unterschiedliche Möglichkeiten zur Kopplung zweier Bausteine, die diese vier Formen der Kopplung beispielhaft näher erläutern.

4.4.1.1 Integrationsformen

Code Reuse

Grad der Kopplung: sehr groß (Laufzeitumgebung, Technologie, Zeit, Datenformate)

Davon sprechen wir, wenn z. B. in Java direkt eine Methode einer anderen Klasse aufgerufen und die andere Komponente als .jar-File wiederverwendet wird. Das ist die stärkste Form der Kopplung und sollte nur in der Mikro-Architekturebene Anwendung finden bzw. beim Bau von Deployment-Monolithen.

Datenbankintegration – gemeinsames Datenmodell

Grad der Kopplung: groß (Technologie, Datenformate)

Dabei greifen verschiedene Systeme direkt auf dieselben Tabellen einer Datenbank zu (Bild 4.6). Wir sind dabei zwar zeitlich voneinander entkoppelt, treffen aber sonst alle möglichen Annahmen. So müssen alle beteiligten Systeme mit dieser Datenbanktechnologie kommunizieren können und haben auch Abhängigkeiten zum konkreten Datenmodell.

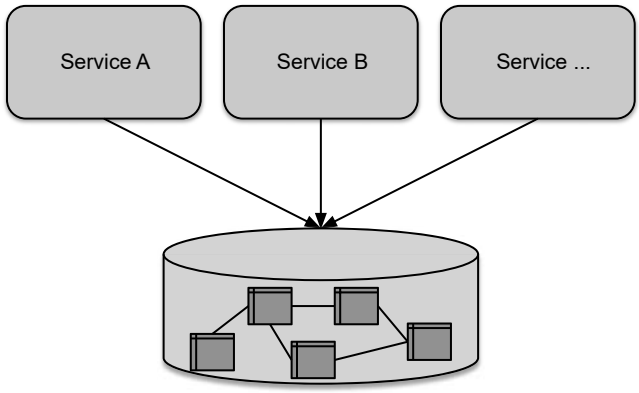


Bild 4.6
Unterschiedliche Services teilen sich dasselbe Datenmodell in derselben Datenbank.

Datenbankintegration – selbe Datenbank, unterschiedliche Datenmodelle

Grad der Kopplung: mittel (Technologie)

Eine alternative Form der Datenbankintegration, bei der sich die einzelnen Services nicht mehr dasselbe Datenmodell teilen, liegt darin, wenn zwar noch dieselbe Datenbank genützt

wird, aber jeder dort sein eigenes Datenmodell besitzt (Bild 4.7). Dadurch ist es einfach, Konsistenz zwischen den Services durch atomare Transaktionen zu gewährleisten. Allerdings wird man mit dieser Form der Integration kaum auskommen, weil auf diese Art und Weise alleine ein Service keine Aktionen in einem anderen Service auslösen kann, wodurch es bei der Datenbankintegration meist noch zu anderen Formen der Interaktion kommt (wie Code-Reuse oder RPC).

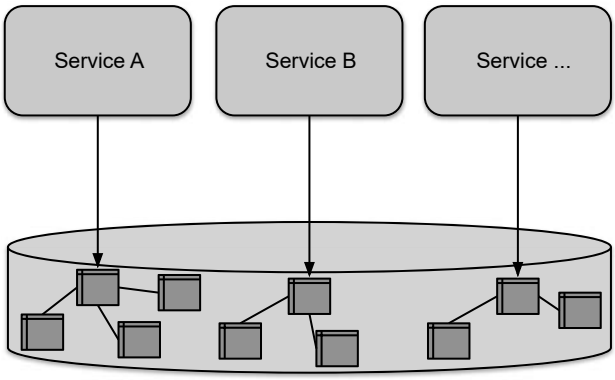


Bild 4.7
Unterschiedliche Services benutzen verschiedene Datenmodelle in derselben Datenbank.

Synchroner Remote Procedure Call

Grad der Kopplung: mittel (Zeit, Datenformate)

Bei Remote Procedure Calls (RPC) kommuniziert eine Komponente mit einer anderen durch eine mehr (SOAP) oder weniger (REST) standardisierte Schnittstelle über das Netzwerk. Hier muss die andere Komponente nach wie vor zur selben Zeit verfügbar sein (Annahme: Zeit), dafür ist es uns egal, mit welcher Technologie die Gegenstelle gebaut wurde. Völlig los werden Sie übrigens die technologische Kopplung mit einem synchronen RPC ebenfalls nicht. So gibt es in Java die sogenannte Remote Method Invocation, kurz RMI, die zwar einen synchronen Call über das Netzwerk ermöglicht, aber immer noch sehr einschränkend auf die Gegenstelle wirkt. Sie können sich dabei weitestgehend die Hardwareplattform des Services aussuchen, sind aber eingeschränkt auf die Java-Technologie, die darauf lauffähig sein muss. Bei anderen Formen des synchronen RPC ist diese Einschränkung in irgendeiner Form eigentlich fast immer ebenfalls vorhanden, aber nicht gleich so offensichtlich. Wenn Sie eine SOAP oder REST API zur Verfügung stellen, unterliegen die potenziellen Consumer nach wie vor der technologischen Einschränkung, dass sie diese Art von Kommunikation beherrschen müssen. Bei einem Cobol-Entwickler, welcher Code für einen Mainframe schreibt, wird man mit einer WSDL wenig Euphorie auslösen. Üblicherweise behilft man sich dann mit einem Message-Bus, welcher auch die zeitliche Kopplung entfernen kann.

Datenreplikation

Grad der Kopplung: gering bis mittel (Datenformate)

Daten werden nicht wie beim RPC per Remote-Schnittstelle geholt, sondern asynchron repliziert. Damit wäre die zeitliche Kopplung entfernt. Der Aufwand lohnt sich aber meiner

Meinung nach nur, wenn es einen triftigen Grund für die Aufgabe der zeitlichen Kopplung gibt. So kann z. B. der Anspruch an ein System, was die Verfügbarkeit angeht, so hoch sein, dass man sich entschließt, die Daten redundant zu halten. Außerdem kann man damit nur Daten einer anderen Komponente wiederverwenden, aber keinerlei Logik, wie eine Berechnung von was auch immer. Zur Umsetzung gibt es verschiedene Möglichkeiten, wie ETL (Extract, Transform, Load) oder Transaction-Log Tailing bzw. CDC (Change-Data-Capture).

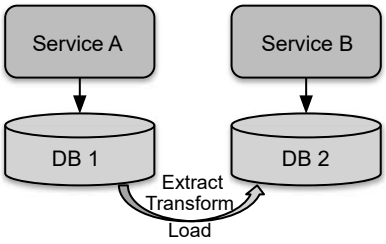


Bild 4.8 Datenreplikation

Messaging

Grad der Kopplung: gering (Datenformate)

Während Remote Procedure Calls darüber definiert sind, dass sie synchron erfolgen, geht es beim Messaging um asynchrone Kommunikation. Man reduziert die Abhängigkeit zwischen Sender und Empfänger um die zeitliche Komponente. Das bedeutet, dass Sender und Empfänger nicht gleichzeitig online sein müssen, es wird also keine zeitliche Annahme getroffen. Prinzipiell unterscheidet man dabei zwischen Message-Bus und Message-Broker. Während ein Bus agnostisch darüber ist, wer Nachrichten publiziert und wer welche konsumiert, ist ein Broker darüber hinaus für das Routing der Nachrichten zu den konkreten Empfängern zuständig [Bro19]. Ein Broker implementiert dabei wenigstens die folgenden beiden Routing-Muster:

Command

Hinter Commands steht der Wunsch des Absenders, eine bestimmte Aktion auszuführen. Ein Message-Broker ist dabei als dezidierte Indirektion dafür zuständig, einen solchen Command an den dafür zuständigen Empfänger weiterzuleiten. Dabei gibt es üblicherweise genau einen Empfänger der Nachricht, aber durchaus verschiedene Endpoints, die einen solchen Command erstellen dürfen. In manchen Fällen kann der konkrete Empfänger auch mittels eines gewissen Regelwerks von der Infrastruktur ermittelt werden. So gibt es in RabbitMQ die Möglichkeit, den Empfänger abhängig von einem sogenannten Topic der Nachricht auszuwählen (Topic Based Routing).

Event

Bei Events informiert ein Absender darüber, dass ein bestimmtes Ereignis stattgefunden hat. Interessierte Empfänger können solche Nachrichten abonnieren und werden dann bei Auftreten eines der abonnierten Ereignisse informiert. Der Absender ist dabei agnostisch darüber, wer und wie viele Empfänger seine Nachrichten abonniert haben. Es gibt dabei üblicherweise immer genau einen Absender einer solchen Nachricht, während sie von mehreren Endpoints empfangen werden kann.

Composite-UI

Grad der Kopplung: Sehr gering (Zeit)

Eine sehr elegante Art und Weise, um Systeme zu integrieren, bietet das User-Interface selbst. Wenn es für den jeweiligen Anwendungsfall sinnvoll ist, sollten Sie immer bestrebt sein, die Kopplung in der UI-Ebene herzustellen. Wobei das prinzipiell auf dem Server oder am Client und somit meist im Browser erfolgen kann. Natürlich kann man es aber mit dieser Form der Optimierung auch übertreiben. Es kann sein, dass eine UI-Integration komplizierter ist als ein synchroner RPC, einfach weil beispielsweise die Aufwände zur Angleichung der User-Interfaces (wie der DOM-Struktur und der CSS-Klassen bei einer Webapplikation) größer sind, als sie es für einen synchronen RPC wären. Wie so vieles, ist also auch das nur eine Abwägung des Pros gegen das Contra.

Microfrontends

Aktuell gibt es als Pendant zu den Microservices den Microfrontend-Hype. Dabei werden verschiedene Möglichkeiten diskutiert, wie man ein größeres Webportal aus mehreren kleineren Einheiten zusammenbauen kann, die selber wiederum mit unterschiedlichen Technologien gebaut werden können. Dafür gibt es unterschiedliche Möglichkeit der technologieneutralen Integration:

- Web-Technologien: Die einzelnen Seiten werden über Links verknüpft oder eine Seite holt sich einen Teil des DOM von einem Service per GET-Request und bettet es in die eigene Seite ein. Siehe dazu auch *https://roca-style.org*.
- Serverseitige Integration: Die Einzelteile werden vom Server zu einer Page zusammengebaut. Dafür gibt es Webstandards wie Edge-Side-Includes (ESI, siehe: *https://www.w3.org/TR/esi-lang*) bzw. Server-Side Includes (SSI, siehe: *https://www.w3.org/Jigsaw/Doc/User/SSI.html*).
- Die Komposition erfolgt im Browser auf der jeweiligen Page. Durch die Möglichkeit der Trennung verschiedener JavaScript-Teile durch die ES-Module (Abschnitt 3.6.1.1) können diese einzelnen Module auf unterschiedliche UI Frameworks und Libraries aufbauen.

4.4.2 Antipattern

4.4.2.1 Kaskadierende Abhängigkeiten

Besonders teuflisch sind sogenannte kaskadierende Abhängigkeiten (A06-Cascades). Dabei handelt es sich um dieselbe Art von Abhängigkeit entlang einer Kette von transitiven Abhängigkeiten. Wenn Ihr Entwurf so etwas aufweist, so wird der Aspekt, der kaskadierend transitiv von Baustein zu Baustein weitergegeben wird, später extrem schwierig zu ändern sein. Dabei kann es sich um jeden der Aspekte aus Abschnitt 4.4.1 handeln. In einem Deployment-Monolithen, wo jede Modulkopplung mit Mitteln der Technologie erfolgt und es somit bei jeder Modulverbindung zur selben technologischen Abhängigkeit kommt, werden Sie irgendwann Probleme haben, die jeweilige Technologie später einmal loszuwerden. Dasselbe Problem können Sie allerdings auch mit Daten- und Schnittstellenformaten bekommen, so wie in Bild 4.9 dargestellt.

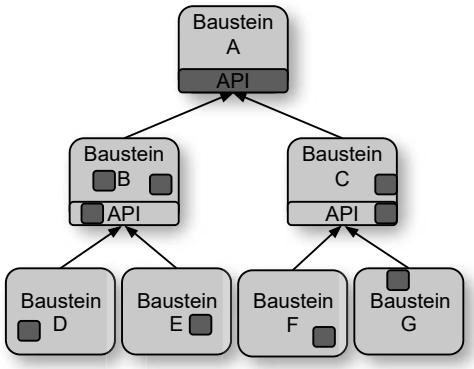


Bild 4.9 Das Schnittstellenmodell pflanzt sich fort.

In diesem Beispiel verwenden die Bausteine B und C das Schnittstellenmodell von Baustein A, welches sie wiederverwenden, selber weiter. Teile davon kommen auch in deren Schnittstellen wiederum vor, welche sie anderen Bausteinen anbieten. Damit ist das Schnittstellenmodell von Baustein A in den Bausteinen zu finden, welche wiederum Consumer der Consumer von Baustein A sind, und hat sich kaskadierend über transitive Abhängigkeiten fortgepflanzt. Eine später einmal notwendige Änderung am Schnittstellenmodell von Baustein A wird entsprechend aufwendig werden, da sie Auswirkungen auf so ziemlich das gesamte System haben wird!

4.4.2.2 Maximierung des Reuse

Zugegeben, es klingt verlockend, möglichst viele Dinge wiederzuverwenden. Schließlich bedeutet das in der Theorie, dass man sich viel redundante Entwicklung erspart [Mur10]. Wie kann es also sein, dass ich mir erlaube, dies hier als Antipattern anzuführen (A07-MaxReuse)? Vielleicht ist dies auch etwas übertrieben. Hauptsächlich möchte ich aufzeigen, dass Reuse nicht das ultimative Ziel einer Architektur sein kann, auch wenn diese Denkweise doch recht verbreitet ist. Der Wunsch nach möglichst viel Wiederverwendung gleicht nämlich dem Versuch, möglichst viele externe Abhängigkeiten zu haben. Und externe Abhängigkeiten sind doch wieder eher die Herausforderung der Softwarearchitektur und auf keinen Fall ihr ultimatives Ziel, da man dadurch das Prinzip der möglichst losen Kopplung (P18-Loose) verletzen würde.

4.4.3 Pattern

4.4.3.1 „Umgekehrte“ Architekturen

In letzter Zeit sind einige Architekturstile populär geworden, die alle gemeinsam haben, dass die Businesslogik sich dabei in der Kette der Abhängigkeiten ganz unten befindet. Dort wird also nicht, wie früher üblich, die Businesslogik über technischen Schichten wie dem Persistenzlayer abgebildet. Beispiele für solch moderne Architekturstile wären Robert Martin’s Clean Architecture [Mar17], Boundary-Control-Entity oder Hexagonale Architek-

tur. Das Ziel ist, die Businesslogik frei von Abhängigkeiten (durch ausgehende Connections) zu technologischen Aspekten (wie Datenbanken oder Messaging-Systemen) zu haben (siehe Bild 4.10). Dadurch können technische Aspekte getauscht, erweitert oder verändert werden, ohne dass im Idealfall die Businesslogik von solchen Änderungen betroffen wäre.

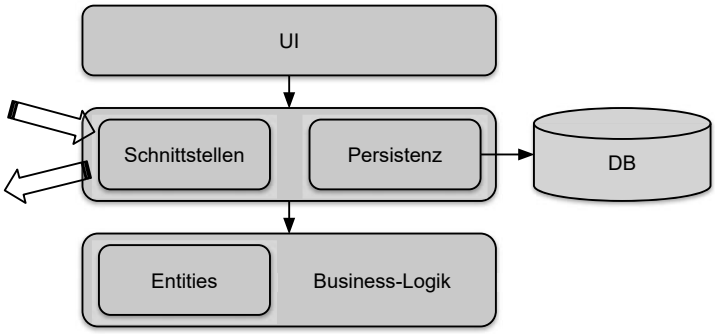


Bild 4.10 Die Business-Logik hat hier keine Abhängigkeiten zu irgendwelchen Technologien.

Das, was „umgekehrte" Architekturen mit der Business-Logik machen, können Sie sich jederzeit beim Entwurf einer Architektur zunutze machen. Teile, die möglichst wenig von anderen Komponenten beeinflusst werden sollen, weil sie beispielsweise besonders wichtig für den Unternehmenserfolg sind, können mit möglichst wenig ausgehenden Abhängigkeiten entworfen werden. Dadurch werden sie in der Wartung stabil. Zur Not helfen Ihnen die Techniken zur Umkehr von Abhängigkeiten aus Abschnitt 3.3.1 (P17-DepInv).

4.4.3.2 Entwurf für Ersetzbarkeit

Überlegen Sie sich bei jedem Baustein, den Sie erzeugen, und bei jeder Technologie, die sie entwickeln, wie Sie diese gegebenenfalls wieder loswerden können. Dazu ist das genaue Gegenteil eines hohen Grads an Wiederverwendung hilfreich, nämlich eine möglichst geringe Anzahl eingehender Abhängigkeiten, oder mit anderen Worten eben eine lose Kopplung. Wie wichtig das ist, darf man nicht unterschätzen. Ich habe jedenfalls noch nie ein Unternehmen erlebt, das in Schwierigkeiten kam, weil manche Funktionalitäten redundant entwickelt wurden. Meistens kommen gewachsene IT-Landschaften in Probleme, wenn Komponenten oder Systeme, welche Schwierigkeiten welcher Art auch immer machen oder deren Betrieb einfach nur teuer wird, nicht mehr einfach ausgetauscht werden können.

4.4.3.3 Circuit Breaker

Vielleicht kennen Sie ja die folgende Situation aus der eigenen leidvollen Erfahrung: Ein Modul in einem verteilten System (genannt Service) bekommt im Betrieb Probleme, die Antwortzeiten steigen immer weiter, bis er schließlich überhaupt nicht mehr antworten kann. Es kommt zu Time-Outs, die aber leider die jeweiligen Consumer dieses Services

nicht davon abhalten, weiterhin Anfragen zu schicken. Diese weiteren Anfragen zwingen dann das System endgültig in die Knie und manchmal auch die Service Consumer mit dazu. Dabei wäre das ganz einfach zu verhindern gewesen, und zwar durch das sogenannte Circuit Breaker Pattern [Nyg07]. Der Consumer ruft den Provider dann nicht mehr direkt auf, sondern jeweils im Umweg über eine „Sicherung“ (englisch: Circuit Breaker). Bemerkt diese Sicherung Probleme im Antwortverhalten des Providers, sendet sie eine Weile keine weiteren Anfragen des Consumers ab und erst nach einer Weile versucht sie es dann wieder. Wenn der Provider wieder antwortet, gehen die Anfragen wieder wie gewohnt dorthin (Bild 4.11). Dadurch erreichen wir Folgendes: Von der jeweiligen Kopplung zwischen zwei interagierenden Services wird die zeitliche Komponente entschieden abgeschwächt. Somit ist der Consumer weiterhin zumindest eingeschränkt lauffähig, auch wenn der Provider ausfallen sollte. Dadurch können sich Probleme auch nicht so einfach weiter ausbreiten und der Provider bekommt die Möglichkeit, sich wieder zu erholen.

Als Fallback-Strategie kann eine Default-Antwort definiert werden, die der Circuit Breaker liefert, falls der Provider offline ist, oder er schaltet auf einen anderen Service Provider oder einen internen Cache um, falls es eine solche alternative Quelle geben sollte.

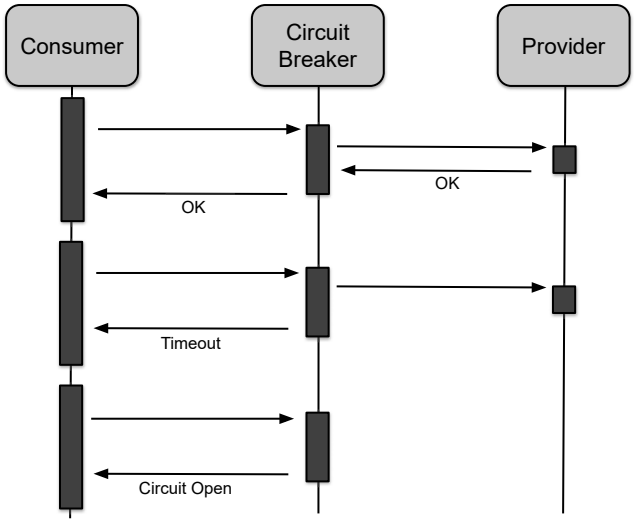


Bild 4.11 Circuit Breaker, in Anlehnung an die UML

4.4.3.4 Bulkhead

Das Bulkhead Pattern [Nyg07] soll, genauso wie ein Circuit Breaker, die Stabilität des Gesamtsystems verbessern. Die Idee ist die gleiche, wie bei Schotten eines Schiffs. Läuft ein Schiff an einer Stelle leck, so werden andere Bereiche abgeschottet, sodass diese nicht auch mit Wasser volllaufen können. In der IT können solche Schotten auf unterschiedliche Art und Weise umgesetzt werden. Es könnten beispielsweise Teile der Systemlandschaft auf jeweils eigenen Server- und Netzwerkteilbereichen im Rechenzentrum betrieben werden (Bild 4.12). Fällt ein Teilnetzwerk aus, so kann der Rest des Systems durch die gegebene Isolation unter Umständen noch weiterverwendet werden.

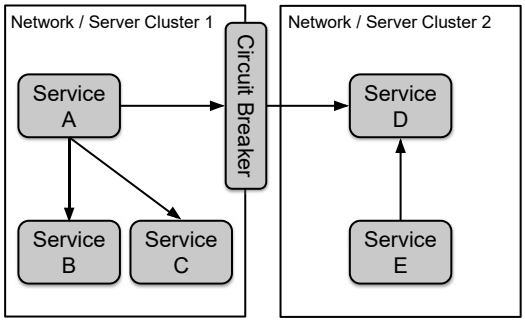
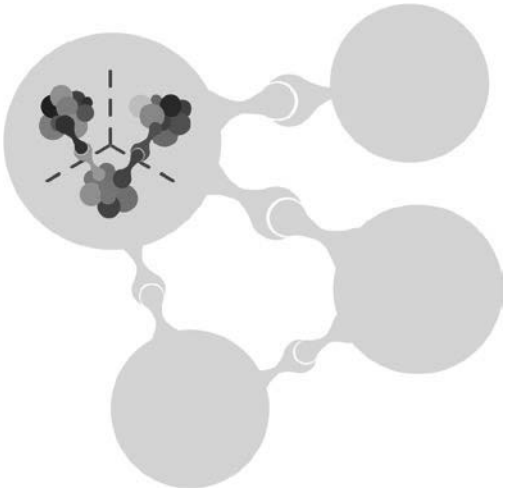


Bild 4.12 Die Systemlandschaft läuft in zwei voneinander abgegrenzten Teilbereichen. Fällt eine aus, so kann mit der anderen immer noch gearbeitet werden.

4.5 CONSTRUCT – Aufbauen



- **Definition:** Eine Bausteinstruktur kann auf einer Ebene selbst wieder unübersichtlich werden. Baue Systeme höherer Komplexität durch Zusammenfassen von Bausteinen einer Ebene zu einem neuen Baustein einer nächsthöheren Ebene. Dabei sind weiterhin dieselben Handlungsmaximen anzuwenden.
- **Messung durch:** Höchster oder durchschnittlicher ACD (Abschnitt 5.1.6.3)

Praktisch jede komplexe Software bietet irgendeine Form von hierarchischer Struktur. Meist bieten die technischen Plattformen bereits bestimmte Möglichkeiten, Code hierarchisch zu strukturieren. In Bild 4.13 sehen Sie die Möglichkeiten, die die Java-Plattform von Haus aus mitbringt. Methoden werden gemeinsam in Klassen gekapselt. Diese wiederum in Packages. Packages können dann noch zu JPMS-Modulen zusammengefasst werden.

Bei entsprechend großem Umfang der Lösung kann dabei nach wie vor Folgendes passieren: Die schiere Anzahl der Module wächst Ihnen über den Kopf und das gesamte System ist im Endeffekt zunächst wieder unübersichtlich. Was kann man in so einem Fall tun? Eine schlechte Idee wäre, mehr Logik in die einzelnen Module zu geben, sodass die Gesamtzahl an Modulen wieder überschaubar wird. Dadurch würde nur Komplexität in die Module hineinverlagert. Die Alternative ist, Strukturen auf der nächsthöheren Abstraktionsebene zu

bilden. Wir zoomen aus unserer Architektur heraus und bauen dann modulare Strukturen nach denselben Regeln, nach denen wir schon die Module auf der unteren Abstraktionsebene entworfen hatten. Während dies den Vorgang eher bottom-up beschreibt, so ist dasselbe Ergebnis natürlich auch top-down zu erreichen. Dabei handelt es sich um den sogenannten hierarchischen Ansatz zum Aufbau einer Architektur.

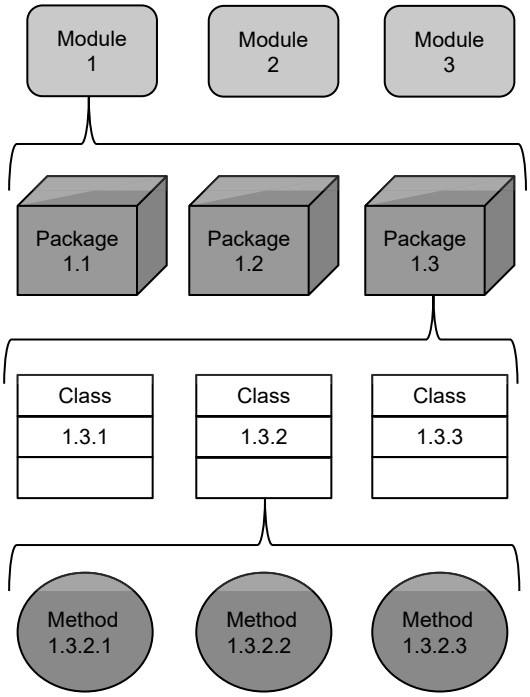


Bild 4.13 Hierarchische Zerlegung von Java-Code

4.5.1 Anitpattern

4.5.1.1 Paradigmenwechsel

Anhänger klassischer Enterprise-Architektur vertreten teilweise die Meinung, dass ab einem gewissen Komplexitätsgrad andere Regeln der Softwarearchitektur gelten würden. Angeblich steht dann nicht mehr ein möglichst hoher Grad an Kapselung im Vordergrund, sondern eine Maximierung der Wiederverwendbarkeit (A07-MaxReuse oder siehe Abschnitt 4.4.2.2). Diesen Paradigmenwechsel gibt es allerdings nicht. In sehr komplexen Architekturen, wie typischen Unternehmensarchitekturen, verändert sich der Zugang zur Architekturarbeit, worauf wir in Kapitel 8 noch eingehen werden. Es handelt sich aber im Wesentlichen immer noch um dieselbe Aufgabe, die dann zu skalieren ist.

4.5.2 Pattern

4.5.2.1 Whole-Part

Ein fast schon in Vergessenheit geratenes Muster, das dabei hilfreich sein kann, ist das Whole-Part-Pattern. Vorgestellt wurde es in einem zeitlosen Klassiker der Softwarearchitektur namens „Pattern-Oriented Software Architecture“ [Bus96] aus dem Jahr 1996. Als Beispiel dient uns der Motor eines Autos. Er besteht aus vielen Einzelteilen, wie dem Kolben, der Kurbelwelle und den Zündkerzen. Der Motor selbst ist aber wiederum nur ein Teil des Autos, das außerdem noch aus Dingen wie dem Lenkrad, der Karosserie, den Rädern und dem Auspuff besteht (Bild 4.14). Vor dem Fahrer bleibt diese Komplexität weitestgehend verborgen. Er dreht den Schlüssel um, beschleunigt mit dem Gaspedal und beeinflusst mit dem Lenkrad die Fahrtrichtung. Einmal im Jahr kommt das Auto in die Werkstätte zur Wartung. Dass dort der Filter der Klimaanlage genauso gewechselt wird wie das Motoröl, kann dem Fahrer egal sein. Die Abstraktion „Auto“ verbirgt all diese Komplexität vor seinem Nutzer so gut es geht.

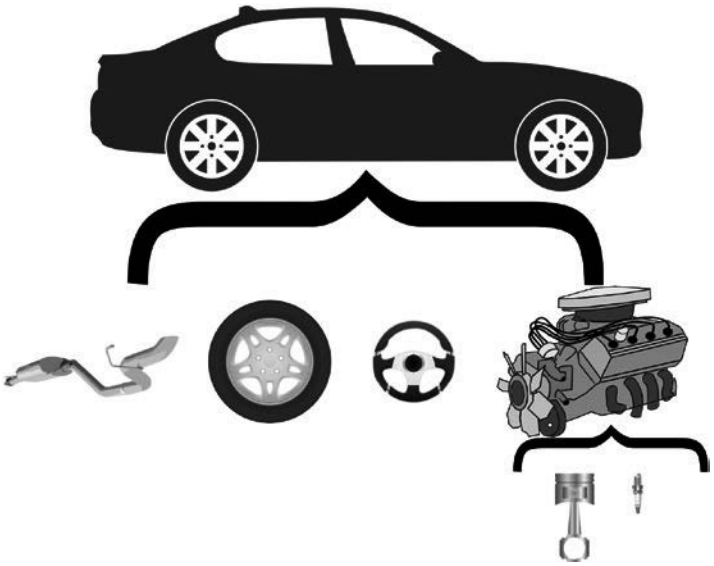


Bild 4.14 Das Auto besteht aus Einzelteilen, die wiederum aus Einzelteilen bestehen.

Das Beispiel mit dem Auto könnte man sogar noch weiterdenken. Ein Robotertaxi-Service könnte eine weitere Abstraktionsebene darüber darstellen. Eine Serversoftware weiß, wo sich die einzelnen selbstfahrenden Taxis gerade befinden. Wenn ein Kunde per App eines der Taxis anfordert, wird die Software des nächstbesten Robotertaxis dieses zum Kunden bewegen und ihn in weiterer Folge an sein gewünschtes Ziel bringen. Das Auto, die Software darin, die Software am Server und die App am Handy des Kunden bilden dann in Summe eine weitere Abstraktionsebene, die die Mobilität für den Kunden noch weiter vereinfacht.

4.5.2.2 Simon Brown's C4-Modell

Beim C4-Modell von Simon Brown (siehe: *https://c4model.com*) handelt es sich primär um einen Standard-Blueprint für hierarchische Softwarearchitekturen. Es gibt vier Ebenen, auf denen eine Zerlegung betrachtet werden kann. Doch damit nicht genug. Simon hat außerdem eine ausgesprochen einfach zu verstehende Alternative zur UML geschaffen und mit structurizr (*https://structurizr.com*) ein Tool, das Architekturen nach dem C4-Modell mit dieser Alternative dokumentieren kann. Um dieses Modell näher zu erläutern, hier ein Beispiel für diese vier Ebenen von top zu bottom, dokumentiert mit der spezifischen Notation:

Context

In der Context-Sicht betrachten wir die gesamte Systemlandschaft, oder das konkrete System mit den Akteuren, die es umgeben. Bei diesen handelt es sich um Personen (also Nutzer des Systems) und Fremdsysteme, die damit interagieren. Unser Beispiel handelt von einem Internet-Banking-System (siehe Bild 4.15). Die Bedeutung eines Pfeils in einem solchen Diagramm ist dabei nicht von der Notation her vorgegeben, sondern wird jeweils durch eine Beschreibung näher erläutert.

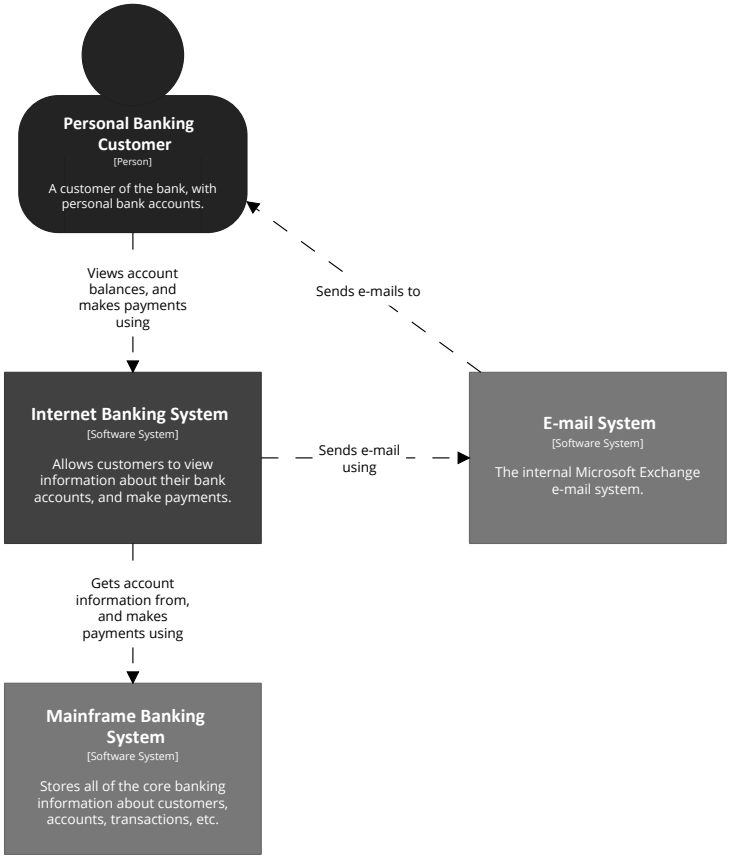


Bild 4.15 System-Context-Diagramm für das Internet-Banking-System

Container

Eine Ebene tiefer blicken wir auf einzelne Applikationen oder Subsysteme (Bild 4.16). Wir sehen, wie sich das Internet-Banking-System weiter in seine Einzelteile zerlegt.

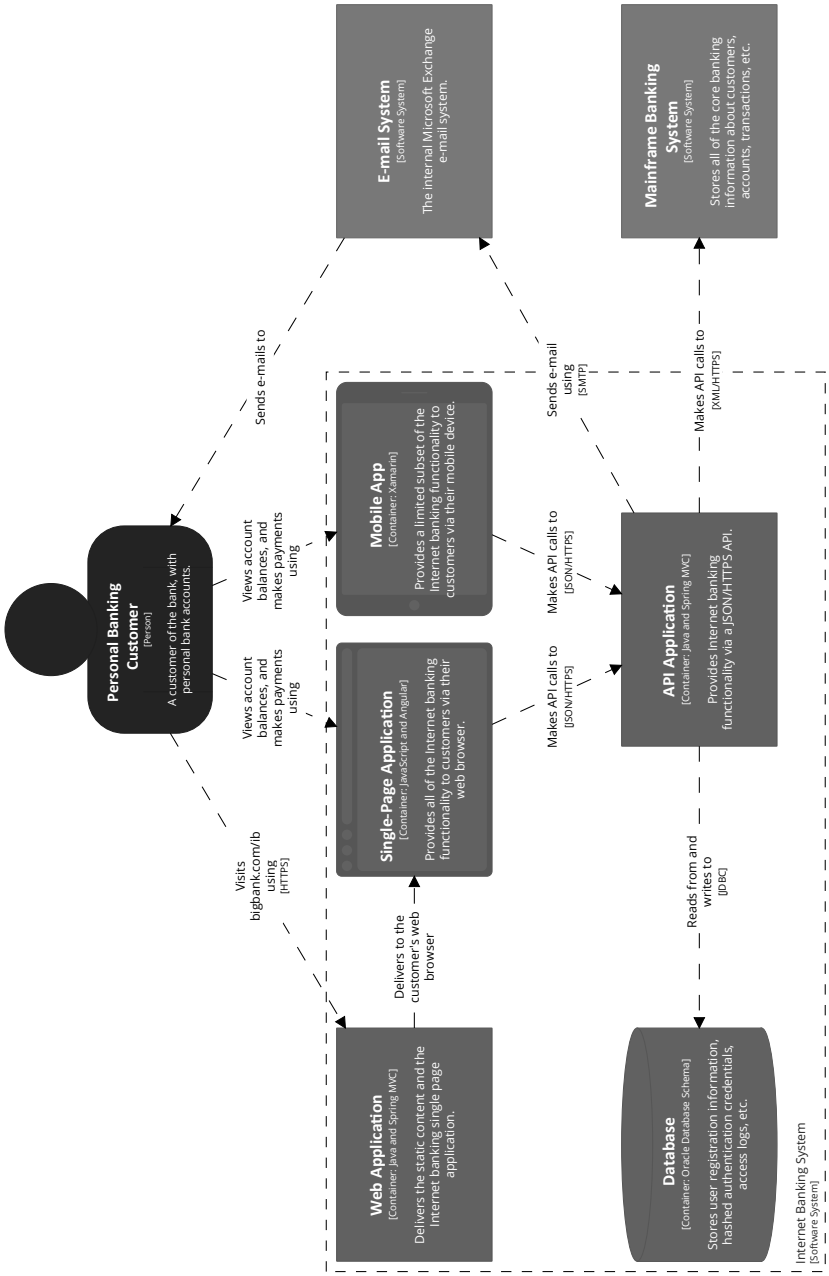


Bild 4.16 Ein Blick in das Internet-Banking-System offenbart die einzelnen Module, aus denen es besteht.

Components

Ein näherer Blick auf das Subsystem API-Application (Bild 4.17) offenbart die einzelnen Module, aus denen diese besteht.

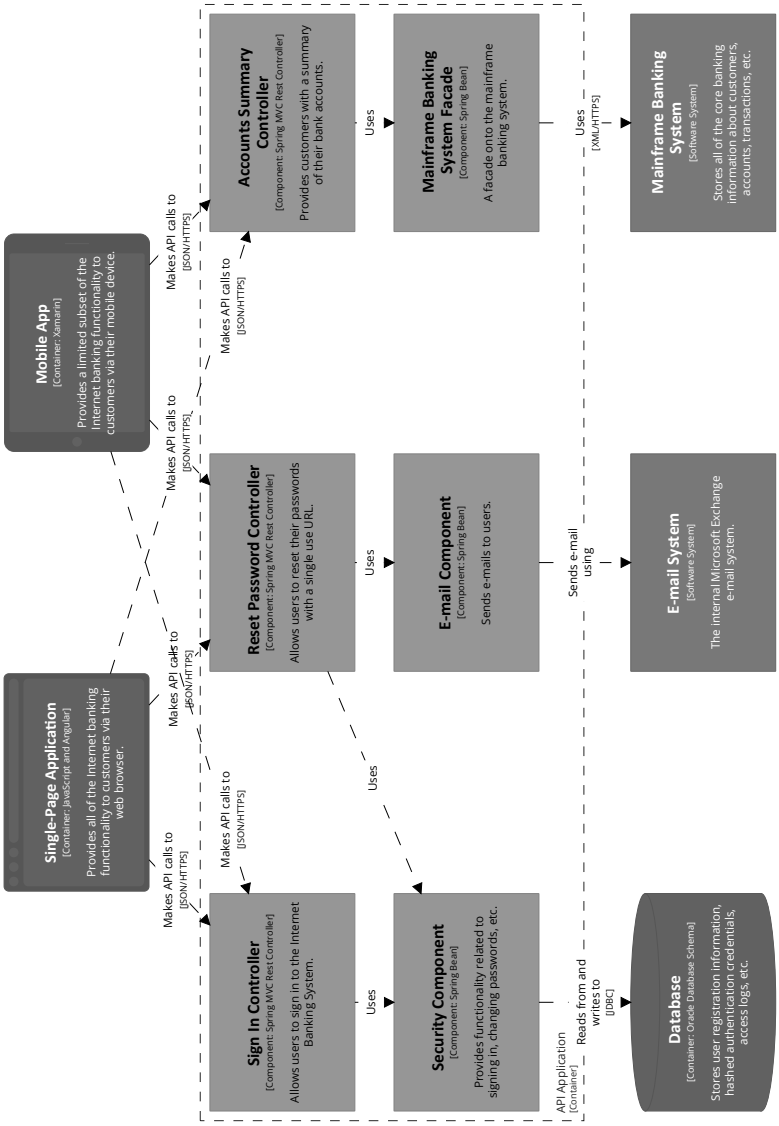


Bild 4.17 Die Module, aus denen die API-Application besteht

Code/Klassen

Üblicherweise bilden Klassen schließlich die Strukturen auf der untersten Ebene einer Architektur. Hier wird empfohlen, diese nicht mit einem Tool oder in einer eigens erstellten Grafik abzubilden, da dies zu detailliert wäre und sich Konzepte auf dieser Ebene zu schnell ändern. Besser lassen sich Abbilder dieser Strukturen bei Bedarf mit der IDE erzeugen.