

Designprinzipien SOLID -

Einleitungstext Clean Architecture

Gute Softwaresysteme bauen auf *Clean Code*, also sauberem Code auf. Einerseits spielt die Architektur eines Softwarekonstrukts keine große Rolle, wenn die einzelnen Bausteine nicht von guter Qualität sind – andererseits kann man aber durchaus auch mit qualitativ hochwertigen Bausteinen erhebliches Chaos anrichten. Und genau hier kommen die sogenannten *SOLID-Prinzipien* (Single-Responsibility-Prinzip, Open-Closed-Prinzip, Liskov'sches Substitutionsprinzip, Interface-Segregation-Prinzip, Dependency-Inversion-Prinzip) zum Tragen.

Die SOLID-Prinzipien geben vor, wie wir unsere Funktionen und Datenstrukturen in Klassen anordnen und wie diese Klassen miteinander verbunden werden. Die Verwendung des Begriffs »Klasse« soll an dieser Stelle allerdings keineswegs bedeuten, dass die Prinzipien nur auf objektorientierte Software angewendet werden sollten – eine Klasse ist einfach eine gekoppelte Gruppierung von Funktionen und Daten. Jedes Softwaresystem besitzt solche Gruppierungen, ob sie nun als Klassen bezeichnet werden oder nicht. Und genau dafür gelten die SOLID-Prinzipien.

Das Ziel dabei ist die Erzeugung von mittelschichtigen Softwarestrukturen, die

- Modifikationen tolerieren,
- leicht nachzuvollziehen sind und
- die Basis der Komponenten bilden, die in vielen Softwaresystemen eingesetzt werden können.

Der Begriff »mittelschichtig« bezieht sich auf den Umstand, dass die Prinzipien von Programmierern angewendet werden, die auf der Modulebene arbeiten. Sie werden unmittelbar oberhalb des Codes angewendet und helfen bei der Definition der Arten von Softwarestrukturen, die in Modulen und Komponenten eingesetzt werden.

Ebenso wie sich mit qualitativ hochwertigen Bausteinen ein beachtliches Chaos anrichten lässt, ist es aber ebenso möglich, ein systemweites Chaos mit gut designten mittelschichtigen Komponenten zu verursachen. Deshalb werden wir uns nach der Betrachtung der SOLID-Prinzipien im weiteren Verlauf auch mit ihren Gegenstücken in der Komponentenwelt sowie anschließend mit den Prinzipien der übergeordneten hochschichtigen Architektur befassen.

Die Geschichte der SOLID-Prinzipien reicht lange zurück. Ich persönlich begann in den späten 1980er-Jahren, sie zu ergründen – während einer Diskussion mit anderen Leuten auf USENET (einer frühen Form von Facebook). Im Laufe der Jahre haben sich die Prinzipien verschoben und verändert. Einige wurden verworfen, andere verschmolzen, und auch neue wurden hinzugefügt. Die endgültige Zusammenstellung verfestigte sich erst in den frühen 2000er-Jahren, wenngleich ich sie hier in einer anderen Reihenfolge aufgeführt habe.

Etwa um das Jahr 2004 schickte mir Michael Feathers eine E-Mail, in der er mich darauf aufmerksam machte, dass die Anfangsbuchstaben der Prinzipien, das englische Wort *SOLID* (zu Deutsch »solide, stabil«) ergeben würden, wenn ich sie umordnete – und so wurde der Begriff »SOLID-Prinzipien« geboren.

Quelle: Clean Architecture. Robert C. Martin, S. 82f.

SRP: Single Responsibility-Prinzip**Ihre Aufgabe:**

Betrachten Sie die Erklärung zum Single Responsibility-Prinzip auf dem bereitgestellten [Plakat](#).

1. Leiten Sie aus der richtigen Darstellung ein passendes UML-Diagramm ab und notieren Sie dieses im nachfolgenden Feld:



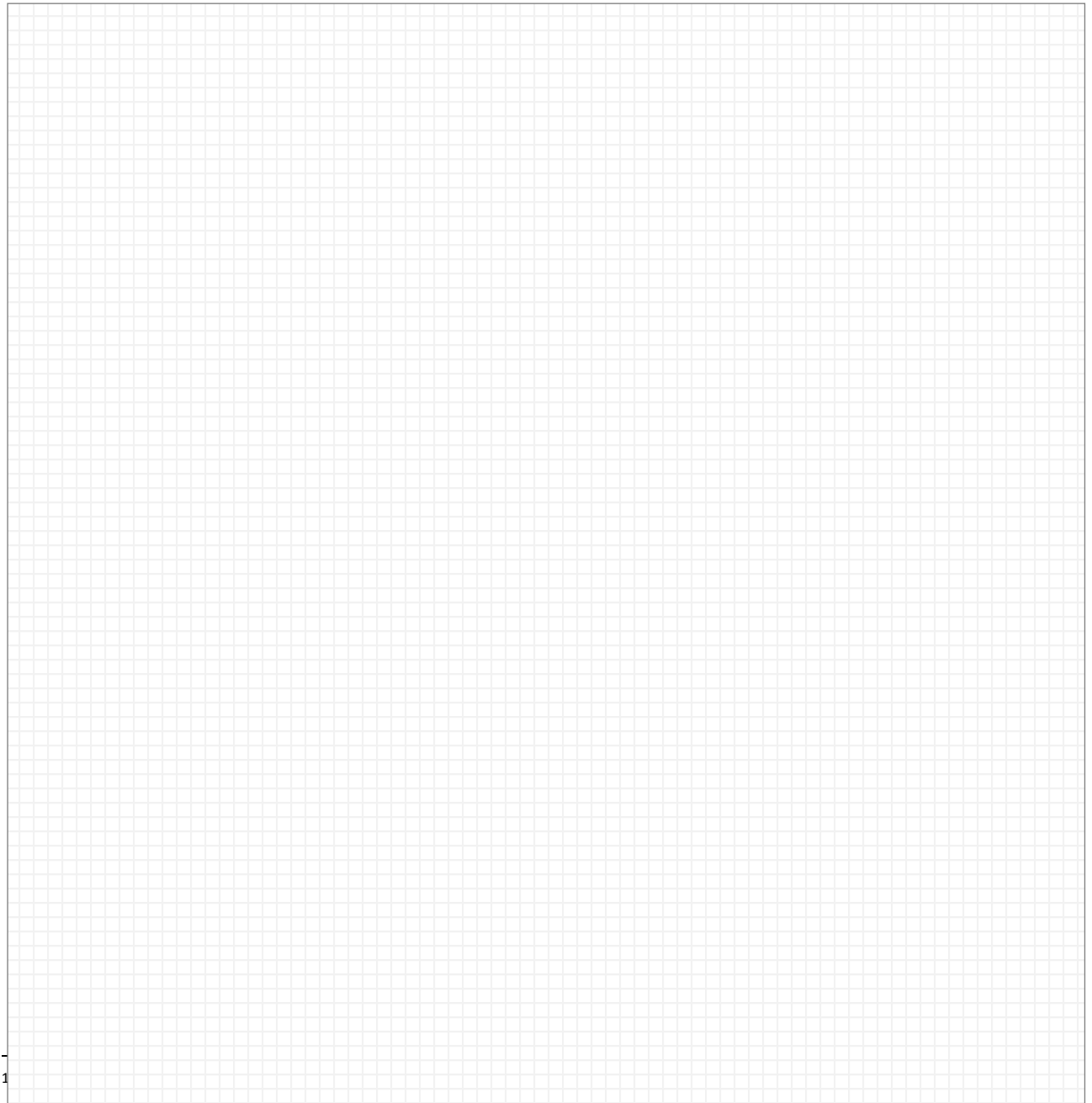
2. Lesen Sie nun den bereitgestellten Theorietext zum Single Responsibility-Prinzip(im Anhang) durch und notieren Sie stichpunktartig im nachfolgenden Feld, welche zwei Probleme bei der Nichteinhaltung des Prinzips auftreten können.



OCP: Open-Closed-Prinzip**Ihre Aufgabe:**

1. Betrachten Sie die Erklärung zum OC-Prinzip auf dem bereitgestellten [Plakat](#).
2. Laden Sie sich den auf [GITHUB](#) bereitgestellten Code aus dem Ordner SOLID/OC/wrong¹ herunter. Erstellen Sie nun basierend auf dem korrekten UML-Diagramm des Plakates eine korrekte Umsetzung des Beispiels in der Programmiersprache Ihrer Wahl.
3. Vergleichen Sie anschließend Ihre Lösung mit der Musterlösung auf [GITHUB](#) unter SOLID/OC/correct

Für Ihre Stichpunkte:



LSP: Liskov'sche Substitutionsprinzip**Ihre Aufgabe:**

1. Betrachten Sie die Erklärung zum LS-Prinzip auf dem bereitgestellten [Plakat](#).
2. Laden Sie sich den auf [GITHUB](#) bereitgestellten Code aus dem Ordner SOLID/LS/wrong herunter. Erstellen Sie nun basierend auf dem korrekten UML-Diagramm des Plakates eine korrekte Umsetzung des Beispiels in der Programmiersprache Ihrer Wahl.
3. Vergleichen Sie anschließend Ihre Lösung mit der Musterlösung auf [GITHUB](#) unter SOLID/LS/correct

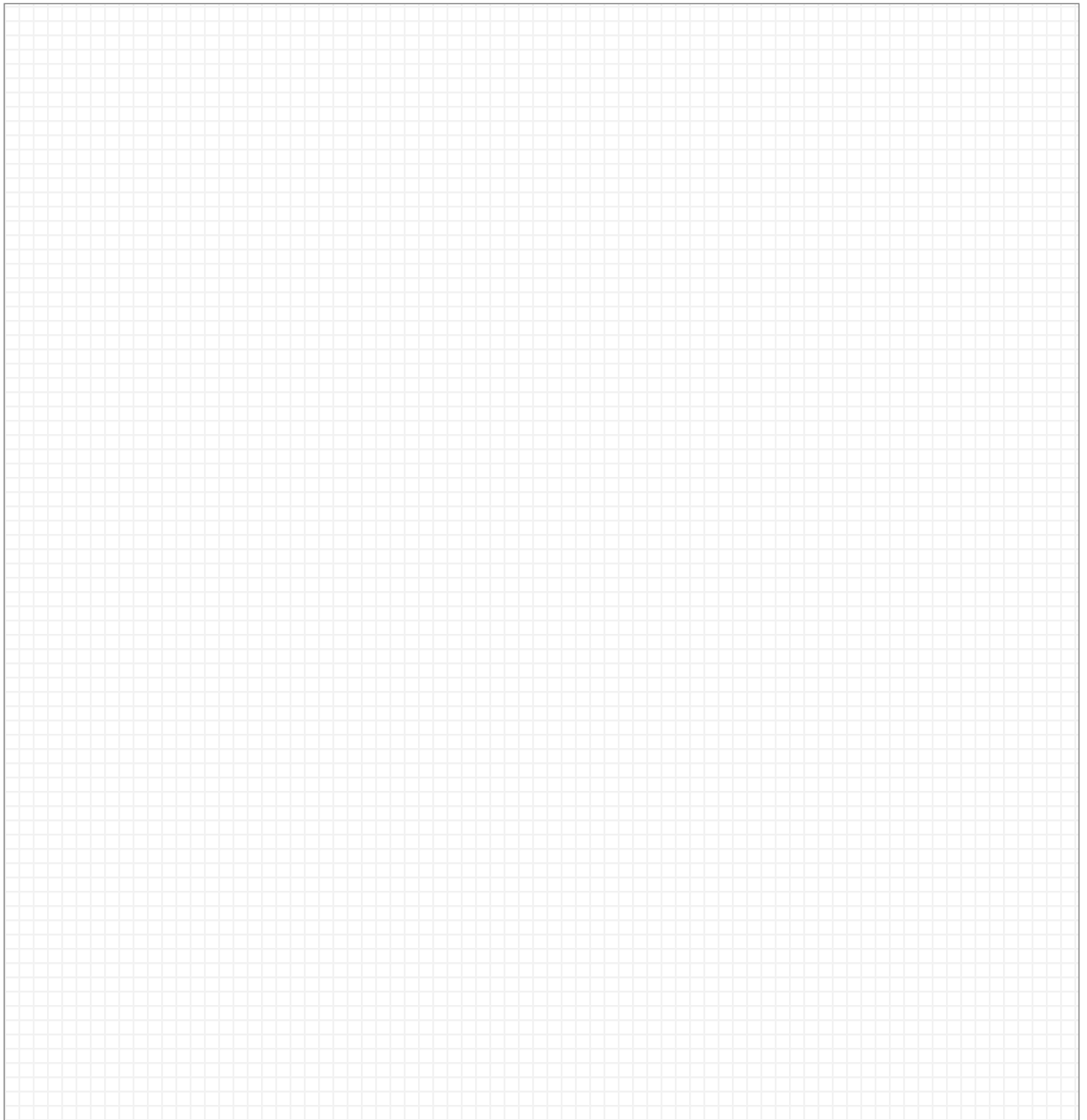
Für Ihre Stichpunkte:



ISP: Interface-Segregation-Prinzip**Ihre Aufgabe:**

1. Betrachten Sie die Erklärung zum IS-Prinzip auf dem bereitgestellten [Plakat](#).
2. Laden Sie sich den auf [GITHUB](#) bereitgestellten Code aus dem Ordner SOLID/IS/wrong herunter. Erstellen Sie nun basierend auf dem korrekten UML-Diagramm des Plakates eine korrekte Umsetzung des Beispiels in der Programmiersprache Ihrer Wahl.
3. Vergleichen Sie anschließend Ihre Lösung mit der Musterlösung auf [GITHUB](#) unter SOLID/IS/correct

Für Ihre Stichpunkte:



DIP: Dependency-Inversion-Prinzip**Ihre Aufgabe:**

1. Betrachten Sie die Erklärung zum DI-Prinzip auf dem bereitgestellten [Plakat](#).
2. Leiten Sie aus der richtigen Darstellung ein passendes UML-Diagramm ab und notieren Sie dieses im nachfolgenden Feld:



3. Lesen Sie nun den bereitgestellten Theorietext zum DI-Prinzip durch und notieren Sie stichpunktartig im nachfolgenden Feld, in welchem Fall Abhängigkeiten zu vermeiden sind.



4. Notieren Sie auch, inwiefern das Pattern der Abstract Factory hier einen passenden Lösungsansatz bietet.



Quelle: Clean Architecture. Robert C. Martin, S. 85ff.

SRP: Das Single-Responsibility-Prinzip



Das *Single-Responsibility-Prinzip* (SRP) dürfte das von allen SOLID-Prinzipien am meisten Missverständene sein. Einer der Gründe dafür ist vermutlich seine besonders unpassende Bezeichnung (»Prinzip der eindeutigen Verantwortlichkeit«), die Programmierer allzu leicht zu der Annahme verleitet, dass jedes Modul ausschließlich eine einzige Aufgabe erfüllen sollte.

Und tatsächlich gibt es auch eine solche Richtlinie, die da lautet: Eine *Funktion* sollte immer nur eine bestimmte, und nur diese eine Aufgabe haben. Dieser Grundsatz wird beim Refactoring umfangreicher Funktionen in kleinere Funktio-

nen angewendet, und zwar auf den niedrigsten Ebenen. Er gehört jedoch *nicht* zu den SOLID-Prinzipien – mit anderen Worten: Das ist *nicht*, worum es beim SRP geht.

Traditionell lautet die Beschreibung des SRPs wie folgt:

Es sollte nie mehr als einen Grund geben, eine Klasse zu modifizieren.¹

Softwaresysteme werden in erster Linie modifiziert, um User und Stakeholder zufriedenzustellen. Sie sind der eigentliche Grund für eine Anpassung der Art, um die es bei diesem Prinzip geht. Man könnte die Beschreibung also auch folgendermaßen umformulieren:

Ein Modul sollte für einen, und nur einen, User oder Stakeholder verantwortlich sein.

Die Aussage »für einen User oder Stakeholder« ist hier allerdings nicht wirklich zutreffend, denn es ist sehr wahrscheinlich, dass ein und dieselbe Systemänderung gleich von mehreren Usern oder Stakeholdern gewünscht wird. Es handelt sich daher eher um Gruppen, die wir nachstehend mit dem Sammelbegriff »Akteur« bezeichnen.

Die finale Version unserer SRP-Beschreibung muss somit lauten:

Ein Modul sollte für einen, und nur einen, Akteur verantwortlich sein.

Doch was ist denn eigentlich mit dem Begriff »Modul« gemeint? Die einfachste Antwort auf diese Frage ist: eine Quelldatei. In den meisten Fällen funktioniert diese Definition wunderbar. Einige Sprachen und Entwicklungsumgebungen nutzen jedoch keine Quelldateien in ihrem Code. In diesen Fällen versteht man unter einem Modul einfach einen zusammenhängenden Satz von Funktionen und Datenstrukturen. Ausschlaggebend für das SRP ist hierbei das Wort »zusammenhängend«: Der Zusammenhalt (die *Kohäsion*) des Moduls ist die treibende Kraft, die den für einen einzelnen Akteur verantwortlichen Code bündelt.

Am besten lässt sich das Single-Responsibility-Prinzip veranschaulichen, indem man die Symptome der möglichen Verstöße gegen Selbiges betrachtet.

7.1 Symptom 1: Versehentliche Duplizierung

Mein Lieblingsbeispiel ist die `Employee`-Klasse einer Anwendung für Lohn- und Gehaltsabrechnungen. Sie verfügt über drei Methoden: `calculatePay()`, `reportHours()` und `save()` (siehe Abbildung 7.1).

¹ *Agile Software Development, Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

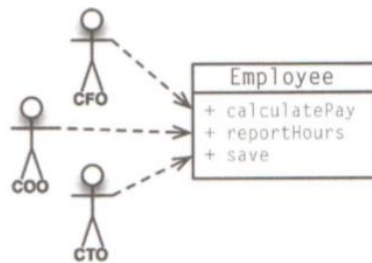


Abb. 7.1: Die Klasse Employee

Diese Klasse verstößt gegen das SRP, weil die zugehörigen drei Methoden drei sehr verschiedenen Akteuren gegenüber verantwortlich sind:

- Die Methode `calculatePay()` ist durch die Buchhaltung spezifiziert, die dem CFO (Finanzvorstand) Bericht erstattet.
- Die Methode `reportHours()` ist durch die Personalabteilung spezifiziert und wird von dieser genutzt, um dem COO (Geschäftsleitung) Bericht zu erstatten.
- Und die Methode `save()` ist durch die Datenbankadministratoren (DBAs) spezifiziert, die dem CTO (Technischen Leiter) berichten.

Dadurch, dass der Quellcode für diese drei Methoden in einer einzigen `Employee`-Klasse abgelegt wurde, haben die Softwareentwickler jeden der genannten Akteure mit den anderen verbunden. Eine derartige Koppelung kann dann beispielsweise zur Folge haben, dass sich vom CFO-Team durchgeführte Aktionen auf etwas auswirken, von dem das COO-Team abhängig ist.

Nehmen wir einmal an, dass sich die Funktion `calculatePay()` und die Funktion `reportHours()` einen gemeinsamen Algorithmus für die Berechnung von unbezahlten Überstunden teilen. Gehen wir außerdem davon aus, dass die Programmierer, die sorgfältig darauf bedacht sind, Codeduplizierungen zu vermeiden, diesen Algorithmus in einer Funktion namens `regularHours()` verwenden (siehe Abbildung 7.2).

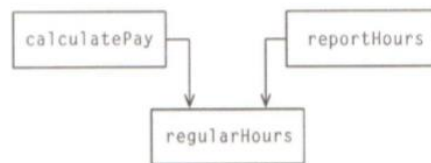


Abb. 7.2: Geteilter Code

Angenommen, das CFO-Team entscheidet nun, dass die Art und Weise der Berechnung der unbezahlten Überstunden optimiert werden muss. Im Gegensatz dazu möchte das COO-Team in der Personalabteilung diese Optimierung jedoch nicht, weil sie die unbezahlten Überstunden für einen anderen Zweck nutzen.

Ein Softwareentwickler wird beauftragt, die Anpassung vorzunehmen und stellt fest, dass die Funktion `regularHours()` praktischerweise von der `calculatePay()`-Methode aufgerufen wird. Allerdings fällt diesem Entwickler nicht auf, dass sie darüber hinaus auch von der Funktion `reportHours()` aufgerufen wird.

Er nimmt also die angeforderte Anpassung vor und testet sie sorgfältig. Das CFO-Team aus der Buchhaltung validiert, dass die neue Funktion wie gewünscht funktioniert, und das System wird deployt.

Allerdings hat das COO-Team keine Kenntnis davon. Die Personalabteilung verwendet weiterhin die Berichte, die über die Funktion `reportHours()` generiert werden – doch diese enthalten nun falsche Zahlen. Schließlich wird das Problem aufgedeckt und der COO ist stinksauer, weil ihn die verfälschten Daten Millionen von Euro seines Budgets kosten.

Wir alle haben solche Situationen schon selbst erlebt. Probleme wie dieses treten auf, weil Code, von dem verschiedene Akteure abhängig sind, in zu große Nähe zueinander rückt. Deshalb besagt das SRP, dass *Code, von dem verschiedene Akteure abhängig sind, separiert werden muss*.

7.2 Symptom 2: Merges

Es ist nicht allzu schwer nachzuvollziehen, dass das *Mergen*, also das »Abgleichen und Zusammenführen« unterschiedlicher Versionen von Quelldateien, die viele verschiedene Methoden enthalten, gängige Praxis ist. Das gilt insbesondere dann, wenn die betreffenden Methoden für diverse Akteure verantwortlich sind.

Nehmen wir beispielsweise an, die Datenbankadministratoren des CTO-Teams beschließen, dass eine simple schematische Änderung an der `Employee`-Tabelle der Datenbank vorgenommen werden soll. Außerdem entscheiden die Personalmitarbeiter des COO-Teams, dass die Formatierung der Stundenberichte modifiziert werden muss.

Zwei verschiedene Programmierer, womöglich von zwei unterschiedlichen Teams, prüfen nun die `Employee`-Klasse und beginnen, Änderungen daran vorzunehmen. Aber leider kollidieren ihre jeweiligen Anpassungen – die Folge ist dann ein Merge.

Wie Sie sicher schon wissen, sind Merges eine riskante Angelegenheit. Die Tools, die uns heutzutage zur Verfügung stehen, sind ziemlich gut, dennoch ist keins von ihnen jedem Merge-Fall gewachsen – im Endeffekt bleibt immer ein Restrisiko.

In unserem Beispiel bedeutet der Merge sowohl für den CTO als auch für den COO ein Risiko. Und es ist auch nicht ausgeschlossen, dass der CFO ebenfalls davon betroffen wäre.

Es gibt noch viele weitere Symptome, die wir untersuchen könnten, sie alle haben jedoch gemeinsam, dass sie mit mehreren Leuten zu tun haben, die dieselbe Quelldatei aus unterschiedlichen Gründen ändern.

Deshalb noch einmal: Der richtige Weg, um dieses Problem zu vermeiden, ist die *Separierung von Code, der von verschiedenen Akteuren genutzt wird*.

7.3 Lösungen

Zur Lösung dieser Problematik stehen zahlreiche Möglichkeiten zur Verfügung, wobei in allen Fällen die Funktionen in unterschiedliche Klassen verschoben werden.

Der vielleicht offensichtlichste Lösungsweg besteht darin, die Daten von den Funktionen zu trennen. In unserem Beispiel haben alle drei Klassen Zugriff auf die Klasse `EmployeeData`, die eine einfache Datenstruktur ohne Methoden enthält (siehe Abbildung 7.3). Jede dieser Klassen hält lediglich den für ihre spezifische Funktion notwendigen Quellcode vor. Es ist den drei Klassen nicht erlaubt, Kenntnis voneinander zu haben – so werden versehentliche Duplizierungen vermieden.

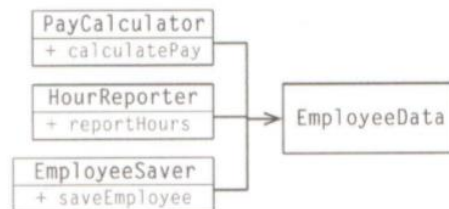


Abb. 7.3: Die drei Klassen haben keine Kenntnis voneinander.

Der Nachteil dieser Lösung ist, dass die Entwickler nun drei Klassen haben, die sie instanziierten und nachverfolgen müssen. Ein gebräuchlicher Ausweg aus diesem Dilemma ist die Verwendung des Patterns *Facade (Fassade)* (siehe Abbildung 7.4).

Die Klasse `EmployeeFacade` enthält nur sehr wenig Code. Sie ist für die Instanziierung und Delegierung der Klassen mitsamt den darin enthaltenen Funktionen verantwortlich.

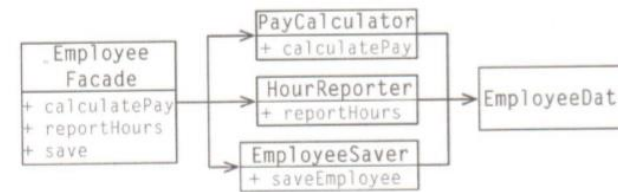


Abb. 7.4: Das Pattern *Facade (Fassade)*

Manche Softwareentwickler bevorzugen es, die wichtigsten Geschäftsregeln näher bei den Daten zu halten. Das kann dadurch realisiert werden, dass die wichtigste Methode in der ursprünglichen `Employee`-Klasse beibehalten und diese dann als *Facade (Fassade)* für untergeordnete Funktionen genutzt wird (siehe Abbildung 7.5).

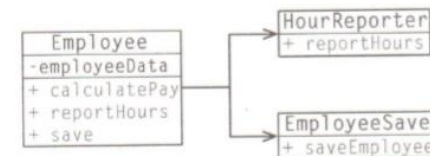


Abb. 7.5: Die wichtigste Methode wird in der ursprünglichen `Employee`-Klasse belassen und als *Facade (Fassade)* für untergeordnete Funktionen genutzt.

Nun könnten Sie gegen diese Lösungswege einwenden, dass jede Klasse nur eine Funktion enthalten würde – doch das dürfte wohl eher nicht zutreffen. Die Anzahl der zur Berechnung der Gehälter, für die Erstellung eines Berichts oder zum Speichern der Daten benötigten Funktionen ist in allen diesen Fällen sehr wahrscheinlich hoch. Jede dieser Klassen würde zahlreiche *private* Methoden beinhalten.

Alle Klassen, die solch eine Familie von Methoden enthalten, bilden einen *Scope* bzw. Anwendungsbereich. Außerhalb dieses Bereichs ist die Existenz der privaten Mitglieder der Familie nicht bekannt.

7.4 Fazit

Beim *Single-Responsibility-Prinzip* geht es um Funktionen und Klassen – darüber hinaus tritt es in einer abgewandelten Form aber auch auf zwei weiteren Ebenen in Erscheinung: Auf der Ebene der Komponenten wird es zum *Common-Closure-Prinzip*, auf der Ebene der Architektur wird es zum *Axis-of-Change-Modell* zur Errichtung architektonischer Grenzen. Diese Konzepte werden in den nachfolgenden Kapiteln betrachtet.

DIP: Das Dependency-Inversion-Prinzip

Quelle: Clean Architecture. Robert C. Martin, S. 107ff.



Das *Dependency-Inversion-Prinzip* (DIP) macht deutlich, dass Systeme, in denen sich Quellcode-Abhängigkeiten ausschließlich auf Abstraktionen beziehen statt auf Konkretionen, am flexibelsten sind.

In einer statisch typisierten Sprache wie Java bedeutet dies, dass sich die Anweisungen `use`, `import` und `include` nur auf Quellmodule beziehen sollten, die Schnittstellen, abstrakte Klassen oder andere Formen von abstrakten Deklarationen enthalten. Dagegen sollten keinerlei Abhängigkeiten zu konkreten Modulen bestehen.

Dasselbe gilt auch für dynamisch typisierte Sprachen wie Ruby und Python: Quellcode-Abhängigkeiten sollten auch hier keine konkreten Module referenzieren. Allerdings ist es in diesen Programmiersprachen ein bisschen schwieriger zu definieren, was ein konkretes Modul ist – grundsätzlich ist darunter jedes Modul zu verstehen, dessen aufgerufene Funktionen implementiert werden.

Dieses Konzept als Regel aufzufassen, ist jedoch sicherlich illusorisch, weil Softwaresysteme natürlich auch von vielen konkreten Entitäten abhängig sein müssen. Beispielsweise ist die `String`-Klasse in Java konkret – und zu versuchen, sie zwingend abstrakt zu gestalten, wäre unrealistisch. Die Quellcode-Abhängigkeit von dem konkreten `java.lang.String`-Objekt kann und sollte nicht vermieden werden.

Hinzu kommt, dass die Klasse `String` sehr stabil ist: Sie wird nur selten modifiziert und zudem engmaschig kontrolliert. Im Allgemeinen brauchen sich Pro-

grammierer und Softwarearchitekten keine Gedanken über häufige, willkürliche Änderungen an dieser Klasse zu machen.

Deshalb neigen wir dazu, den stabilen Hintergrund des Betriebssystems und der Plattformentitäten im Zusammenhang mit dem DIP zu ignorieren. Wir tolerieren konkrete Abhängigkeiten dieser Art, weil wir uns darauf verlassen können, dass sie sich nicht ändern.

Anders verhält sich das hingegen mit konkreten, flüchtigen *volatile*-Elementen unseres Systems – hier gilt es, Abhängigkeiten zu vermeiden, denn diese Module werden aktiv entwickelt und unterliegen daher häufigen Anpassungen und Modifikationen.

11.1 Stabile Abstraktionen

Jede Änderung an einer abstrakten Schnittstelle hat zugleich auch eine Modifikation ihrer konkreten Implementierungen zur Folge. Umgekehrt gehen Anpassungen konkreter Implementierungen dagegen nicht immer mit Änderungen an den Schnittstellen einher, von denen sie implementiert wurden – in der Praxis ist das sogar eher unüblich. Insofern sind Schnittstellen weniger flüchtig als Implementierungen.

Tatsächlich achten gute Softwaredesigner und -architekten bei ihrer Arbeit sehr darauf, die Flüchtigkeit von Schnittstellen zu reduzieren. Zu diesem Zweck suchen sie nach Möglichkeiten, um die Funktionalität der Implementierungen ohne die Notwendigkeit entsprechender Anpassungen der Schnittstellen erweitern zu können. Diese Vorgehensweise wird als »Software Design 101« bezeichnet.

Im Umkehrschluss bedeutet dies, dass Softwarearchitekturen genau dann stabil sind, wenn sie Abhängigkeiten von flüchtigen Konkretionen vermeiden und stattdessen dem Einsatz stabiler abstrakter Schnittstellen den Vorzug geben. Und das lässt sich durch die Anwendung einer Reihe von sehr spezifischen Programmierpraktiken erreichen:

- **Referenzieren Sie keine flüchtigen konkreten Klassen.** Verwenden Sie stattdessen Referenzierungen auf abstrakte Schnittstellen. Diese Richtlinie gilt für alle Programmiersprachen, ob statisch oder dynamisch typisiert. Zudem unterwirft sie die Erzeugung von Objekten strikten Einschränkungen und forciert allgemein den Einsatz des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*).
- **Nutzen Sie keine Ableitungen von flüchtigen konkreten Klassen.** Das ist im Grunde genommen eine logische Folge der vorherigen Richtlinie, soll aber an dieser Stelle dennoch besondere Erwähnung finden. In statisch typisierten Sprachen stellt die Vererbung die stärkste und strikteste aller Quellcode-Beziehungen dar – und dementsprechend sollte sie auch mit besonderer Sorgfalt angewendet werden. In dynamisch typisierten Sprachen ist die Vererbung ein

geringfügigeres Problem, es existiert aber immer noch eine Abhängigkeit – und eine vorsichtige Vorgehensweise ist stets die klügste Wahl.

- **Überschreiben Sie keine konkreten Funktionen.** Konkrete Funktionen bedingen häufig Quellcode-Abhängigkeiten. Wenn Sie diese Funktionen überschreiben, heben Sie die zugehörigen Abhängigkeiten damit nicht auf – faktisch *vererben* Sie sie. Um solche Abhängigkeiten zu verwalten, sollten Sie die betreffende Funktion abstrakt gestalten und mehrere Implementierungen erzeugen.
- **Erwähnen Sie konkrete und flüchtige Elemente zu keinem Zeitpunkt namentlich.** Hierbei handelt es sich im Kern um eine nochmalige Darlegung des Prinzips selbst.

11.2 Factories

Um den vorerwähnten Richtlinien zu entsprechen, muss der Erzeugung flüchtiger konkreter Objekte eine besondere Behandlung zukommen. Diese Vorsichtsmaßnahme ist deshalb erforderlich, weil zur Erstellung eines Objekts in buchstäblich allen Programmiersprachen eine Quellcode-Abhängigkeit von der konkreten Definition des betreffenden Objekts notwendig ist. In den meisten objektorientierten Sprachen wie Java würde man zur Verwaltung dieser unerwünschten Abhängigkeit das Design Pattern *Abstract Factory* (*Abstrakte Fabrik*) anwenden.

Die schematische Darstellung in Abbildung 11.1 veranschaulicht den strukturellen Aufbau dieses Konzepts. Die Klasse *Application* nutzt *ConcreteImpl* über die *Service*-Schnittstelle. Allerdings muss *Application* irgendwie Instanzen von *ConcreteImpl* erzeugen. Um dies zu erreichen, ohne eine Quellcode-Abhängigkeit von *ConcreteImpl* herzustellen, ruft *Application* die Methode *makeSvc* der Schnittstelle *ServiceFactory* auf. Diese Methode wird wiederum von der Klasse *ServiceFactoryImpl* implementiert, die von *ServiceFactory* abgeleitet ist. Und diese Implementierung instanziiert schließlich die Klasse *ConcreteImpl* und gibt sie als *Service* zurück.

Die geschwungene Linie in Abbildung 11.1 kennzeichnet eine architektonische Grenze: Sie trennt das Abstrakte von dem Konkreten. Alle Quellcode-Abhängigkeiten kreuzen diese Linie und zeigen in dieselbe Richtung – zur abstrakten Seite.

Sie unterteilt das System in zwei Komponenten: Die eine ist abstrakt, die andere konkret. Die abstrakte Komponente enthält alle übergeordneten Geschäftsregeln der Anwendung. Die konkrete Komponente enthält alle Implementierungsdetails, die diese Geschäftsregeln manipulieren.

Beachten Sie hierbei, dass der Kontrollfluss die geschwungene Linie in die entgegengesetzte Richtung der Quellcode-Abhängigkeiten kreuzt. Letztere sind gegen den Kontrollfluss invertiert – weshalb dieses Prinzip als *Dependency Inversion* (zu Deutsch »Abhängigkeitsumkehr«) bezeichnet wird.

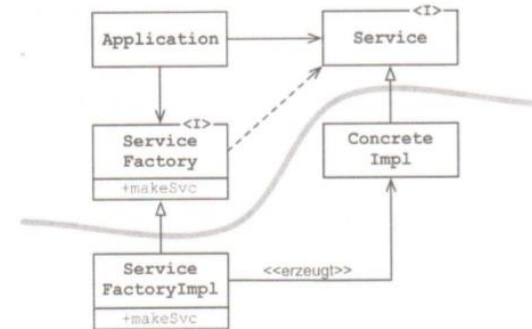


Abb. 11.1: Anwendung des Design Patterns *Abstract Factory* (*Abstrakte Fabrik*) zur Verwaltung der Abhängigkeit

11.3 Konkrete Komponenten

Die konkrete Komponente in Abbildung 11.1 enthält eine einzelne Abhängigkeit und verstößt damit gegen das DIP. Das ist ein typischer Fall. DIP-Verstöße lassen sich nicht vollständig beseitigen, aber sie können in einer kleineren Anzahl von konkreten Komponenten gesammelt und vom Rest des Systems getrennt gehalten werden.

Die meisten Systeme enthalten mindestens eine solche konkrete Komponente – die oftmals mit *main*¹ bezeichnet ist, weil sie die *main*-Funktion umfasst. In dem in Abbildung 11.1 gezeigten Beispielfall würde die *main*-Funktion die Klasse *ServiceFactoryImpl* instanziiieren und diese Instanz dann in einer globalen Variablen vom Typ *ServiceFactory* platzieren. Der Zugriff von *Application* auf die Factory würde somit über diese globale Variable erfolgen.

11.4 Fazit

Wenn wir uns im weiteren Verlauf dieses Buches den höheren Architekturprinzipien zuwenden, werden wir dem DIP immer wieder begegnen. Es ist das offensichtlichste Organisationsprinzip der noch folgenden Architekturschemata. Die geschwungene Linie in Abbildung 11.1 beschreibt die architektonischen Grenzen in den späteren Kapiteln. Die Art und Weise, in der die Abhängigkeiten diese Linie in eine Richtung kreuzen – hin zu den abstrakteren Entitäten –, wird später noch in einer neuen Regel erfasst, die als *Abhängigkeitsregel* (engl. *Dependency Rule*) bezeichnet wird.

¹ Mit anderen Worten die Funktion, die beim ersten Start der Anwendung vom Betriebssystem aufgerufen wird.