Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

# Verbesserung und Evolution von Architekturen

Bearbeiter 1: Tobias Thierbach
Bearbeiter 2: Erik Kierspel
Bearbeiter 3: Annika Kremer

Gruppe: Thema 8

Software-Architekturen Ausarbeitung

Trier, 17.07.2020

# Kurzfassung

Im Rahmen dieser Ausarbeitung wird zunächst vorgestellt, wie sich Änderungen anhand von Ausmaß, Dimension und Zeit einordnen lassen. Danach wird die Evolutionsperspektive auf die Softwarearchitektur erläutert. Hierbei wird zunächst gezeigt, wie sich die Evolutionsperspektive auf verschiedenen Viewpoints, wie z.B. den Kontext, den Funktionalen, den Informations, den Concurrency und den Development Viewpoint anwenden lässt. Daraufhin werden die Concerns der Evolutionsperspektive erklärt. Im Anschluss werden die Aktivitäten zur Anwendung der Evolutionsperspektive anhand eines Aktivitätsdiagramms vorgestellt. Hierbei werden die vier Schritte Anforderungen bestimmen, aktuellen Stand bestimmen, Tradeoffs abwägen und Architekturtaktiken anwenden näher erläutert. AIM42 liefert dazu eine Sammlung bestehender Praktiken, mit deren Hilfe Software systematisch verbessert werden kann. Hier werden die drei Phasen von AIM42, sowie phasenübergreifende Praktiken beschrieben. Nach dem allgemeinen Vorgehen wird eine Auswahl von konkreten Architectural Tactics vorgestellt, die zur Evolution von Software-Architekturen angewandt werden könnten. Hierbei werden die etablierten SOLID-Prinzipien im Bezug auf Evolution von Softwarearchitekturen präsentiert und erläutert sowie Möglichkeiten zur lokalen Eingrenzung von Änderungsauswirkungen vorgestellt. Als weitere Taktiken wird gezeigt, wie sich Interfaces erweiterbar gestalten lassen, Variation Points einbauen und Extension Points nutzen lassen. Außerdem werden Metamodelle behandelt, ein Architekturstil, der auf Evolution ausgelegt ist. Den Abschluss der Architectural Tactics bilden Methoden, Änderungen am Code so sicher wie möglich umzusetzen. Zu allen Taktiken werden anschauliche Beispiele gezeigt. Abschließend wird auf verschiedene Probleme und Fallen eingegangen. Dabei wird erklärt wie diese zustande kommen können, welche Auswirkungen sie besitzen und wie die Eintrittswahrscheinlichkeit reduzierbar ist.

# Inhaltsverzeichnis

# 1

# Architekturtaktiken

Eine Architekturtaktik ist eine etablierte und bewährte Vorgehensweise, die verwendet werden kann, um eine bestimmte Qualitätseigenschaft zu erreichen ([NR12], S.48, Z.11-12).
In diesem Kapitel werden sieben Taktiken zur Erfüllung der für Softwarearchitektur-Evolution wichtigen Qualitätseigenschaften (Concerns) identifiziert und beschrieben.

## 1.1 Begriffe

In diesem Abschnitt werden zunächst die zum Verständnis dieses Kapitel notwendigen Begriffe erläutert.

*Modul*

Ein Modul ist ein zusammenhängender Satz von Funktionen und Datenstrukturen ([Mar18], S.86). Im Falle der objektorientierten Programmierung ist ein Modul beispielsweise eine Klasse oder ein Interface.

*Komponente*

Komponenten sind als Deployment-Einheiten zu verstehen, d.h. „sie repräsentieren die kleinsten Entitäten, die als Teil eines Systems deployt werden können“([Mar18], S.113, Z.2-3). Es ist bei gutem Komponentendesign möglich, Komponenten unabhängig voneinander zu entwickeln und zu deployen, beispielsweise als Plug-ins im .jar oder .exe Format ([Mar18], S.113). Diese Unabhängigkeit erleichtert den Umgang mit Änderungen enorm, da Entwickler entscheiden können, wann geänderte Komponenten integriert werden ([Mar18], S.131).

*Abhängigkeit*

Wenn ein Modul ein anderes Modul verwendet, ist dies eine Quellcode-Abhängigkeit, kurz Abhängigkeit, vom verwendeten Modul. Jede Objekterzeugung stellt eine Abhängigkeit von der konkreten Definition des Objektes dar ([Mar18], S.109). Die stärkste und strengste Abhängigkeit stellt Vererbung dar ([Mar18], S.108).

*Design-Pattern*

Ein Design-Pattern dokumentiert eine oft wiederkehrende und etablierte Struktur von miteinander verbundenen Design-Elementen, die ein generelles Design Problem in einem bestimmten Kontext löst ([NR12], S.165, Z.4-6). Design-Patterns lösen demnach bestimmte Probleme in einem bestimmten Kontext.

*Designprinzip*

Ein Software-Designprinzip ist eine umfassende und fundamentale Doktrin oder Regel, welche die Entwicklung von qualitativen Software Designs leitet ([Kan03], S.1, Z.64-66). Designprinzipien sind allgemeiner als Patterns, d.h. sie sind nicht an bestimmte Probleme oder Kontexte gebunden und unabhängig von Implementierungsdetails.

## 1.2 Designprinzipien

Die erste hier vorgestellte Architekturtaktik besteht darin, etablierte Designprinzipien zu verwenden. Sie helfen, Auswirkungen von Änderungen einzuschränken ([NR12], S.553) sowie die Komplexität des Systems zu verringern. Eine verständliche und gut strukturierte Architektur lässt sich einfacher erweitern und verbessern.

### 1.2.1 Die SOLID-Prinzipien

Bei den SOLID-Prinzipien handelt es sich um Designprinzipien, welche sich auf die Modulebene beziehen. Der Begriff „SOLID “ist ein Akronym für die einzelnen Prinzipien, wobei der Begriff um 2004 von Robert C. Martin geprägt wurde. Ihm ist die Zusammenstellung der Prinzipien in ihrer heutigen Form zu verdanken. Die Prinzipien selbst reichen jedoch weitaus länger zurück, wennngleich sie sich im Laufe der Jahre immer wieder verändert haben. Die SOLID-Prinzipien gelten nicht nur für objektorientierte Programmierung ([Mar18], S.82-83).
Ziel der SOLID-Prinzipien ist es, Module so zu gestalten und zu organisieren, dass diese Änderungen tolerieren und leicht nachvollziehbar sind. Damit wird Änderbarkeit bereits auf Modulebene unterstützt und es wird das Fundament für gut struktuierte Komponenten gelegt.

#### Das Single Responsibility Princip (SRP)

Das Single-Responsibility-Prinzip wird aufgrund des Namens oft missverstanden, mit der Annahme, dass jedes Modul nur eine Aufgabe haben soll. Dies ist jedoch das Separation of Concerns Prinzip.
Das Single-Responsibility-Prinzip lässt sich eindeutiger formulieren: „Ein Modul sollte für einen, und nur einen, Akteur verantwortlich sein. “([Mar18], S.86, Z.18)
Das heißt, dass „Code, von dem verschiedene Akteure abhängen, separiert werden

muss “([Mar18], S.88, Z.19-20).
„Akteur“ist hierbei ein Sammelbegriff für Gruppen, die gemeinsame Änderungsinteressen haben. Das kann ein User oder Stakeholder sein, aber auch mehrere. Das SRP soll verhindern, dass Änderungen für einen Akteur sich unbeabsichtigt und eventuell sogar unbemerkt auf andere Akteure auswirken ([Mar18], S.85-88).

### Das Open-Closed Prinzip (OCP)

Das Open-Closed-Prinzip wurde 1988 von Bertrand Meyer formuliert ([Mey88], S.23) und besagt: „Eine Softwareentität sollte offen für Erweiterungen, aber zugleich auch geschlossen gegenüber Modifikationen sein. “([Mar18], S.91, Z.4-5).
Die Möglichkeit, Module erweitern zu können, ohne bestehenden Code verändern zu müssen, stellt den Idealfall dar. Das OCP sollte darum stets ein leitendes Motiv beim Entwurf von Modulen sein.

### Das Liskov'sche Substitutionsprinzip (LSP)

Das Liskov'sche Substitutionsprinzip wurde 1987 von Barbara Liskov formuliert [Lis87]. Es besagt übersetzt: „Was hier erreicht werden sollte, ist etwas wie die folgende Substitutionseigenschaft: Wenn für jedes Objekt o1 vom Typ S ein Objekt o2 vom Typ T existiert, sodass für alle Programme P, die in T definiert sind, das Verhalten von P unverändert bleibt, wenn o1 für o2 substituiert wird, dann ist S ein Subtyp von T “([Mar18], S.97, Z.5-8)
Einfacher ausgedrückt lautet die Aussage: S ist ein Subtyp von T, wenn T durch S ersetzt werden kann und das Programmverhalten weiterhin gleich bleibt. Es ist also gefordert, dass Module durch ihre Subtypen wechselseitig ersetzbar sind. Das LSP spielt vor allem bei Vererbung eine wesentliche Rolle. Wenn das LSP eingehalten wird, kann ein Modul der Klasse T von allen Klassen, die von T erben, substituiert werden, was ein hohes Maß an Flexibilität erlaubt. Wenn T ein Interface ist, dann kann T von allen Klassen substituiert werden, welche das Interface implementierten. Umgekehrt wird das LSP verletzt, wenn die Unterklassen keine echten Unterklassen sind und sich das Systemverhalten bei einem Austausch ändert([Mar18], S.98-99).

### Das Interface-Segregation-Prinzip (ISP)

Das Interface-Segregation-Prinzip „hält Softwaredesigner dazu an, Abhängigkeiten von nicht genutzten Modulen zu vermeiden“([Mar18], S.84, Z.15-16).
Beispielsweise stellen transitive Abhängigkeiten eine solche Abhängigkeit von ungenutzten Modulen dar, die es gilt aufzulösen ([Mar18], S.105).
Solche ungenutzten Abhängigkeiten kann es auch auf kleinerer Ebene in Form von ungenutzten Funktionen geben. Wenn eine Klasse viele Funktionen enthält, aber nur wenige Funktionen tatsächlich benötigt werden, macht es Sinn, ein Interface dazwischenzuschalten, welches nur die benötigten Funktionen enthält. Auf diese Weise haben Änderungen an den nicht genutzten Funktionen keine Auswirkungen

mehr. Werden für verschiedene Klassen, die unterschiedliche Funktionen nutzen, jeweils ein solches Interface dazwischengeschaltet, erhält man eine Trennung durch Interfaces, was den Namen des Prinzips begründet ([Mar18], S.103-104).

### Das Dependency-Inversion Prinzip (DIP)

Das letze der SOLID-Prinzipen besagt: „Der Code, der die übergeordneten Richtlinien (engl. Policy) implementiert, sollte nicht von dem Code abhängig sein, der untergeordnete Details implementiert. Vielmehr sollten Details von den Richtlinien abhängig sein.“([Mar18], S.84, Z.17-21).
Abhängigkeiten von Modulen, die sich oft ändern, d.h. von flüchtigen, sogenannten *volatile*-Elementen, sind zu vermeiden. In diesem Fall ist es vorzuziehen, eine Abstraktion zwischen den Modulen einzubauen, sodass sowohl die übergeordneten als auch die untergeordneten Module beide von der Abstraktion abhängig sind ([Mar18], S.107-108). Bei der Abstraktion kann es sich beispielsweise um ein Interface oder eine abstrakte Klasse handeln.
Eine alternative Defintion des DIP lautet darum: „Unser Entwurf soll sich auf Abstraktionen stützen. Er soll sich nicht auf Spezialisierungen stützen.“([BL09b], Z.12-13).
Durch Quellcode-Abhängigkeiten, die sich aussschließlich auf Abstraktionen beziehen, ist das System sehr flexibel. Die flüchtigen Module, die untergeordnete Details enthalten, können sich beliebig oft ändern, ohne dass die übergeordneten Module davon beeinflusst werden, und umgekehrt, denn beide hängen von einer Abstraktion ab, die in den meisten Fällen unverändert bleibt. Beispielsweise bleibt ein Interface unbeeinflusst, wenn sich eine Klasse ändert, die das Interface implementiert. Diese Einschränkung der Änderungsauswirkungen ist ausgesprochen wichtig, da Änderungen an flüchtigen Elementen häufig zu erwarten sind. Wird das Dependency-Inversion Prinzip eingehalten, ist das System flexibel und auf Modifikationen vorbereitet.
Damit das DIP funktioniert, muss allerdings bewusst darauf geachtet werden, dass die abstrakten Schnittstellen so stabil wie möglich gestaltet werden, denn mit flüchtigen Schnittstellen wäre nichts gewonnen ([Mar18], S.108).
Überall Abstraktionen einzubauen ist in der Praxis nicht realistisch. Nicht alle Abhängigkeiten zu konkreten Modulen lassen sich vermeiden. Es ist jedoch hier wichtig, zwischen flüchtigen und nicht flüchtigen Modulen zu unterscheiden. Konkrete Module, die stabil sind, werden eher unwahrscheinlich geändert. Hier sind Abhängigkeiten tolerierbar ([Mar18], S.107-108). Zudem können die Auswirkungen von Abhängigkeiten abgeschwächt werden, indem die konrekten Module gemeinsam in konkreten Komponenten gruppiert werden, sodass die Abhängigkeiten lokal begrenzt und vom restlichen System getrennt sind ([Mar18], S.110).
Ein Beispiel für das DIP erfolgt an späterer Stelle in diesem Kapitel (siehe 1.2.3).

## 1.2.2 Weitere Designprinzipien

Im folgendenn werden einige weitere Designprinzipien forgestellt, die nicht zu den SOLID-Prinzipien gehören, aber ebenso dazu beitragen, die Auswirkungen

von Änderungen lokal einzuschränken ([NR12], S.553). Anders als die SOLID-Prinzipien gelten sie teilweise nur für objektorientierte Programmierung, dann wird bewusst von Klassen anstatt Modulen gesprochen.

#### Encapsulation

Encapsulation oder Kapselung bechreibt das Prinzip, Daten mit dem kleinstmöglichen Zugriffsrecht zu versehen und lediglich über eine öffentliche Schnittstelle zur Verfügung zu stellen. Beispielsweise können Variablen auf private gesetzt und lediglich über Getter- und Settermethoden zugänglich gemacht werden. Analog sollten klasseninterne Methoden vor Zugriffen von außerhalb geschützt werden ([Mar09], S.136).
Dies sorgt für eine geringe Kopplung zwischen den Klassen, da auf die Daten nicht uneingeschränkt zugegriffen werden kann. Je weniger andere Klassen mit den Daten oder Funktionen in Berührung kommen, desto weniger wirken sich Änderungen an den Daten auf jene anderen Klassen aus [BL09a].

#### Separation of Concerns

Das Separation of Concerns Prinzip besagt, dass jedes Systemelement eine klare Verantwortlichkeit haben sollte. Ein Element kann beispielsweise ein Modul, aber auch eine Funktion sein. Separation of Concerns sollte auf allen Ebenen eingehalten werden. Wird das Prinzip befolgt, wirken sich Änderungen an einem Element nur lokal eingeschränkt aus. Wird das Prinzip missachtet, können hingegen bei kleinsten Änderungen weite Teile des Systems mitbetroffen sein ([NR12], S.553).

#### Funktionale Kohäsion

Funktionale Kohäsion ist eine Software-Metrik, die angibt, wie stark die Funktionen eines Moduls miteinander in Beziehung stehen und damit logisch zusammengehören. Demnach spricht eine hohe Kohäsion dafür, dass das System sinnvoll in Module unterteilt ist. Bei einer hohen funktionalen Kohäsion wirken Änderungen sich meist nur lokal auf jenen zusammenhängenden Bereich aus ([NR12], S.553). Die konsequente Befolgung dieses Prinzips führt zu vielen kleinen Modulen([Mar09], S.140).

#### Single Point of Definition

Das Single Point of Definition Prinzip besagt, dass Datentypen, Werte, Algorithmen, Konfigurationen Schemata etc. nur einmal definiert und implementiert werden sollen ([NR12], S.553). Es gibt demnach stets genau einen Definitionspunkt. Dies bietet den Vorteil, dass jene Elemente bei einer Anpassung nur einmal geändert werden müssen, nämlich an der Stelle, an der sie definiert sind.

### 1.2.3 Komponentenprinzipien

Im folgenden werden weitere Designprinzipien vorgestellt, die sich jedoch nicht mehr auf die Modul-, sondern auf die Komponentenebene beziehen.

### Das Acyclic-Dependencies-Prinzip (ADP)

Das Acyclic-Dependencies-Prinzip bezieht sich auf die Komponentenkopplung, d.h. die Beziehungen zwischen Komponenten. Das Prinzip besagt, dass im Schema der Komponentenabhängigkeiten keine Zyklen auftreten dürfen ([Mar18], S.129, Z.8). Die Auswirkungen von Änderungen lassen sich nicht mehr klar abschätzen, sobald ein Abhängigkeitszyklus vorliegt. Abbildung 1.1 zeigt einen solchen Abhängigkeitszyklus in einem für Anwendungen typischen Komponentendiagramm ([Mar18], S.131).
Durch die zyklische Abhängigkeit verschmelzen die Komponenten Entities, Authorizer und Interactors praktisch zu einer einzigen Komponente, obwohl sie unabhängig sein sollen. Wenn Authorizer sich ändert, muss nicht nur Entities angepasst werden, um kompatibel zu sein. Interacters ist von Entities und damit transitiv von Authorizer abhängig und muss somit ebenfalls angepasst werden. Dies gilt analog für Interactors und Entities. Die durch den Zyklus entstandenen transitiven Abhängikeiten erschweren sowohl das Entwickeln als auch das Testen. Es gibt keine richtige Reihenfolge mehr, in der die Komponenten erstellt oder geändert werden sollten ([Mar18], S.133-134). Da ein Zyklus keinen Endpunkt hat, kann es in der Theorie eine endlose Folge von notwendigen Anpassungen geben. In der Praxis werden die Anpassungen zwar nicht endlos sein, aber unangenehm und vor allem vermeidbar aufwendig.

*Auflösung mittels DIP*

Der Zyklus kann mittels Anwendung des Dependency-Inversion-Prinzips unterbrochen werden. Es wird ein Interface erzeugt, in der alle Methoden enthalten sind, welche User benötigt. Dieses Interface gehört zur Komponente Entities. User hängt von diesem Interface ab, aber die Abhängigkeit ist innerhalb derselben Komponente und geht nicht mehr zu Authorizer. Authorizer implementiert nun dieses Interface, d.h. Authorizer ist nun vom Interface und damit von Entities abhängig. Damit zeigt der Abhängigkeitspfeil in die umgekehrte Richtung und der Zyklus wurde unterbrochen. Abbildung 1.2 verdeutlicht dieses Vorgehen.

### Das Stable-Dependencies-Prinzip (SDP)

Das Stable-Dependencies-Prinzip bezieht sich auf die Komponentenkopplung und lautet: „Abhängigkeiten sollten in dieselbe Richtung verlaufen wie sie Stabilität.“([Mar18], S.137, Z.16).
Demnach sollen stabile Komponenten nicht von instabilen Komponenten abhängen, die sich mit hoher Wahrscheinlichkeit ändern werden. Stabile Komponenten lassen sich nur schwer modifizieren und behindern damit die Änderungen an der instabilen Komponente, da sie nur schwer daran angepasst werden können. Stattdessen sollen die instabilen von den stabilen Komponenten abhängen[Mar18], S.137).
Stabilität im Zusammenhang mit Software gibt an, wie viel Aufwand erforderlich ist, eine Komponente zu ändern. Je größer der Aufwand, desto stabiler ist sie. Eine Komponente ist sehr stabil, wenn sie viele eingehende , aber kaum ausgehende Abhängigkeiten aufweist ([Mar18], S. 138). Die Instabilität einer Komponente

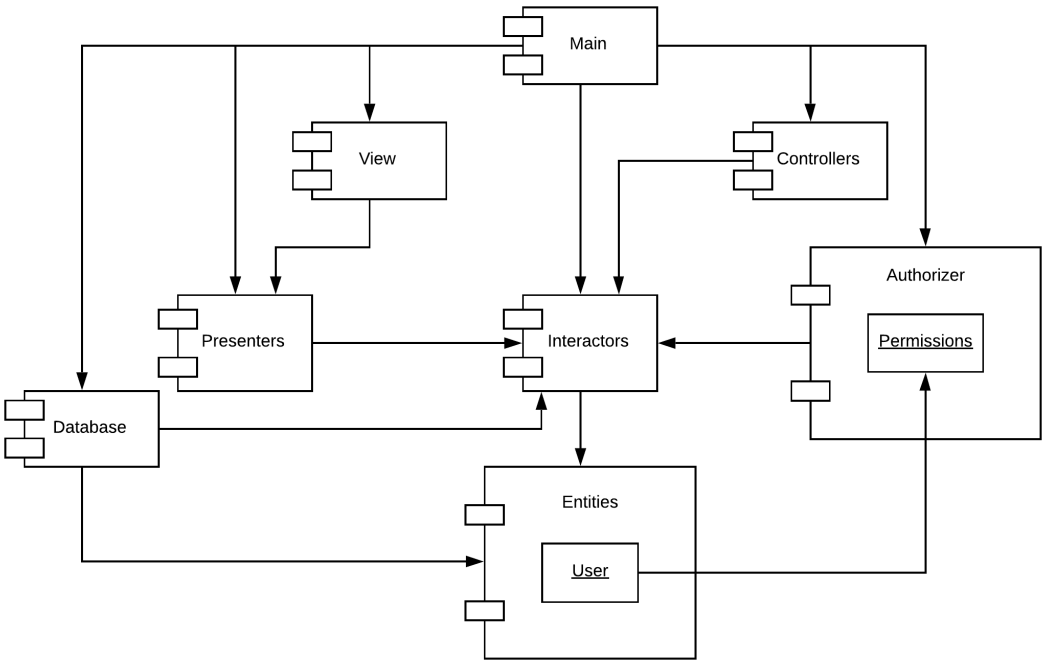


**Abb. 1.1.** Ein Abhängigkeitszyklus(basierend auf [Mar18], S.134 Abb. 14.2. sowie S.135, Abb.14.3)

$I \in [0, 1]$ lässt sich wie in Formel 1.1 angegeben bestimmen ([Mar18], S.139), wobei 0 maximal stabil entspricht und # ist die Anzahl bezeichnet. Verstöße gegen das SDP lassen sich mit dem DIP (siehe 1.2.1) lösen ([Mar18], S.139-142).

$$I = \frac{\#AusgehendeAbhängigkeiten}{\#EingehendeAbhängigkeiten + \#AusgehendeAbhängigkeiten} \tag{1.1}$$

### Das Stable-Abstractions-Prinzip (SAP)

Das Stable-Abstractions-Prinzip bezieht sich ebenfalls auf die Komponentenkopplung und lautet: „Eine Komponente sollte ebenso abstrakt sein, wie sie stabil ist.“([Mar18], S.143, Z.14).
Damit sollen stabile Komponenten aus Schnittstellen sowie abstrakten Klassen und instabile Komponenten aus konkreten Klassen bestehen. Diese Abstraktion erlaubt es, stabile Komponenten trotz ihrer Stabilität erweitern zu können.
SAP und SDP zusammengenommen ergeben das DIP auf Komponentenebene. Der wesentliche Unterschied ist, dass Komponenten im Gegensatz zu Modulen nicht entweder abstrakt oder konkret sind, sondern sich auch dazwischen befinden können. Der Grad der Abstraktion einer Komponente $A \in [0.1]$ lässt sich genau wie Stabilität als Software-Metrik messen mittels der in 1.2 angegebenen Formel ([Mar18], S.144), wobei 1 für maximal abstrakt steht.

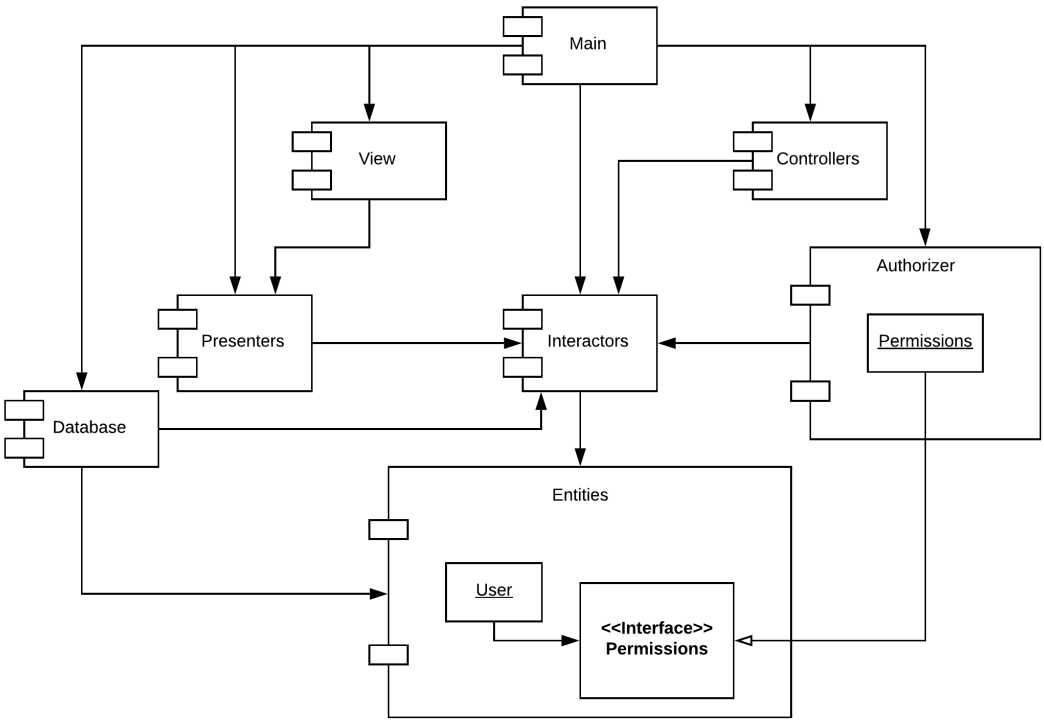


**Abb. 1.2.** Auflösung des Abhängigkeitszyklus mittels DIP (basierend auf [Mar18], S.134 Abb. 14.2. sowie S.135, Abb.14.3)

$$A = \frac{\#abstrakteKlassen/Schnittstellen}{\#GesamtzahlKlassen} \tag{1.2}$$

## 1.3 Design-Patterns

Es exisitiert eine breite Auswahl von Abstrahierungs- und Genreralisierungs-Design-Patterns, die Änderungen vereinfachen. Im Folgenden werden Beispiele aufgeführt.

### 1.3.1 Abstract Factory

Das DIP (siehe Abschnitt 1.2.1 bedingt, dass konkrete, flüchtige Objekte nicht ohne Weiteres erzeugt werden können, denn jede Objekterzeugung stellt eine Abhängigkeit zu einer konkreten Klasse des Objektes dar ([Mar18], S.109). Abhilfe schafft das Design-Pattern Abstract Factory. Dieses Pattern erlaubt es, Familien verwandter Objekte zu erzeugen, ohne deren konkrete Klassen zu spezifizieren, d.h. ohne Abhängigkeiten zu jenen Objekten herzustellen ([Shv20], S.90).
In Abbildung 1.3 ist das Pattern am Beispiel einer plattformübergreifenden grafischen Anwendung gezeigt. In der Mitte befindet sich das Interface GUIFactory, dies ist die Abstract Factory. Hier werden Methoden für die konkreten Factories

vorgegeben. Windows Factory und MacFactory sind die konkreten Factories, sie implementieren die Methoden der Abstract Factory. Das Interface Checkbox stellt ein abstraktes Produkt dar, das von den konkreten Produkten WindowsCheckbox und MacCheckbox implementiert wird, analog verhält es sich mit dem Interface Button ([Shv20], S.96).
Der erste Vorteil besteht in der Konsistenz der Produkte: Die WindowsFactory stellt nur zueinander passende Windows UI-Elemente her, analog bei der MacFactory. Das Pattern lässt sich um eine Linux-Factory erweitern, die konsistente Linux UI-Elemente enthält.
Der zweite Vorteil besteht darin, dass die Anwendung lediglich von GUIFactory abhängig ist, d.h. von einer Abstraktion, was dem DIP entspricht.
Es stellt sich die Frage, an welcher Stelle eine Factory-Instanz erzeugt wird, z.B. eine WindowsFactory. Dies geschieht in einer konkreten Klasse innerhalb einer konkreten Komponente, wie etwa in der main-Methode, wobei die erzeugte Instantz in einer globalen Variablen gespeichert wird, auf welche die Anwendung dann global zugreifen kann ([Mar18], S.110).

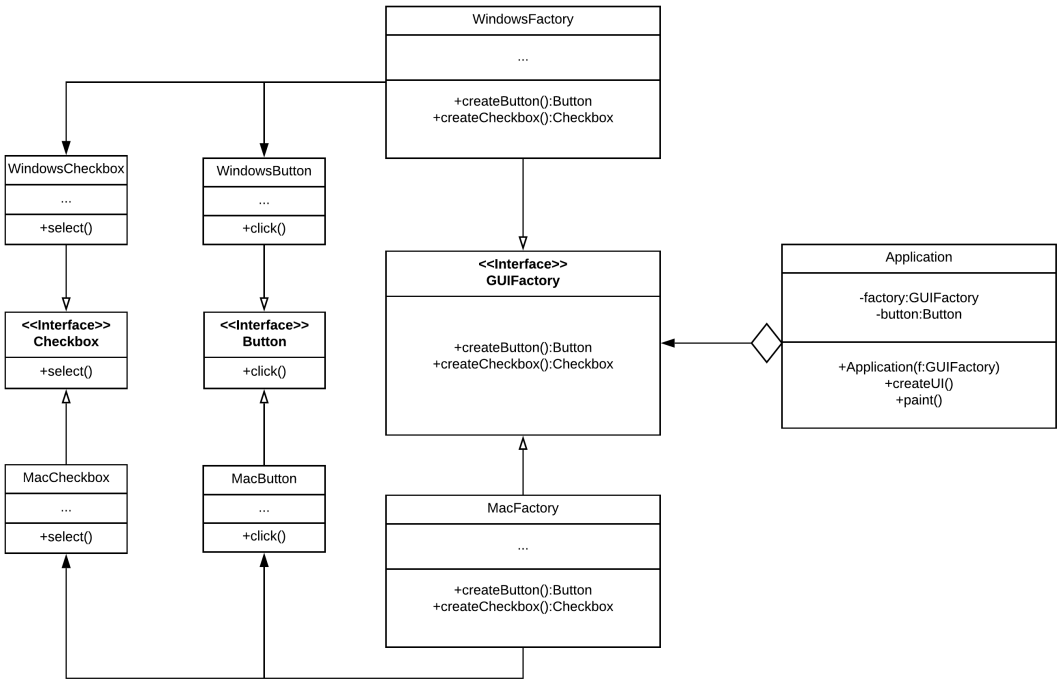


**Abb. 1.3.** Design Pattern Abstract Factory (Abbildung basierend auf [Shv20], S.97)

### 1.3.2 Dependency Injection

Dieses Design-Pattern hilft, übergeordnete Module vor den Implementierungsdetails von untergeordneten Modulen zu schützen ([NR12], S.555).

Abbildung 1.4 zeigt Constructor Dependency Injection an einem konkreten Beispiel. Die Klasse *PaymentTerms* beinhaltet alle Informationen, die zur Berechnung der monatlichen Kosten eines Kredits notwendig sind. Die Klasse *PaymentCalculator* enthält die Methode *getMonthlyPayment()*, d.h. hier wird die konkrete Berechnung durchgeführt. Das Problem besteht darin, dass *PaymentTerms* diese Methode aufrufen muss und ohne Dependency Injection direkt von *PaymentCalculator* abhängig wäre. Dies wäre bedenklich: Es könnnten neue Arten von *Calculator*-Klassen hinzukommen, welche die Berechnung anders durchführen, oder vielleicht gibt es auch bereits mehrere solcher *Calculator*-Klassen. Dann sollte es nicht in der Klasse *PaymentTerms* entschieden werden, welche Berechnung verwendet wird, sondern in einem übergeordneten Modul.
Constructor Dependency Injection löst dies folgendermaßen: Es wird ein Interface erzeugt, hier *IPaymentCalculator* genannt, das die benötigte Methode enthält und das von *PaymentCalculator* und allen möglichen weiteren *Calculator*-Klassen implementiert wird. Der Klasse *PaymentTerms* wird im Konstruktor ein *IPaymentCalculator*-Objekt übergeben. Damit besteht nur noch eine Abhängigkeit zu einer Abstraktion, was dem DIP entspricht (siehe 1.2.1). Der konrkete Calculator-Typ wird erst zur Laufzeit entschieden, da die Abhängigkeit über den Konstruktor injected wird, was den Namen des Patterns erklärt. Das bietet eine hohe Flexibilität, denn die Implementierungsdetails in *PaymentTerms* stellen nun kein Problem mehr da. Wird ein anderer Berechnungstyp benötigt, kann von einem übergeordneten Modul aus einfach ein anderes *IPaymentCalculator*-Objekt übergeben werden. Es ist anzumerken, dass Constructor Injection nur ein möglicher Typ von Dependency Injection ist. Es existieren zudem noch Setter Injection und Interface Injection [Car10]. Dependency Injection ist auch unter dem Namen Inversion of Control bekannt ([NR12], S.555).

### 1.3.3 Weitere Patterns

Weiter Patterns, die Änderungen vereinfachen, sind das Prototype Pattern ([Shv20] S.124), das direkte Abhängigkeiten beim Kopieren von Objekten vermeidet, sowie das Adapter Pattern, dass es Objekten mit inkompatiblen Schnittstellen erlaubt, dennoch zusammenzuarbeiten ([Shv20], S.150).

## 1.4 Erweiterbare Interfaces

Diese Architekturtaktik bezieht sich auf die Gestaltung von Interfaces. Ihnen sollte besondere Aufmerksamkeit zukommen, denn Änderungen an Interfaces haben den größten Einfluss auf das Systen und verursachen damit die größten Kosten. Wird beispielsweise ein Parameter einer Funktion geändert, müssen alle Klassen angepasst werden, welche das Interface implementieren und damit diese Funktion verwenden.
Eine mögliche Lösung besteht darin, anstatt vieler Paramter Objekte oder andere strukturierte Datentypen, z.B. structs in C++, in der Funktion zu übergeben. Alle

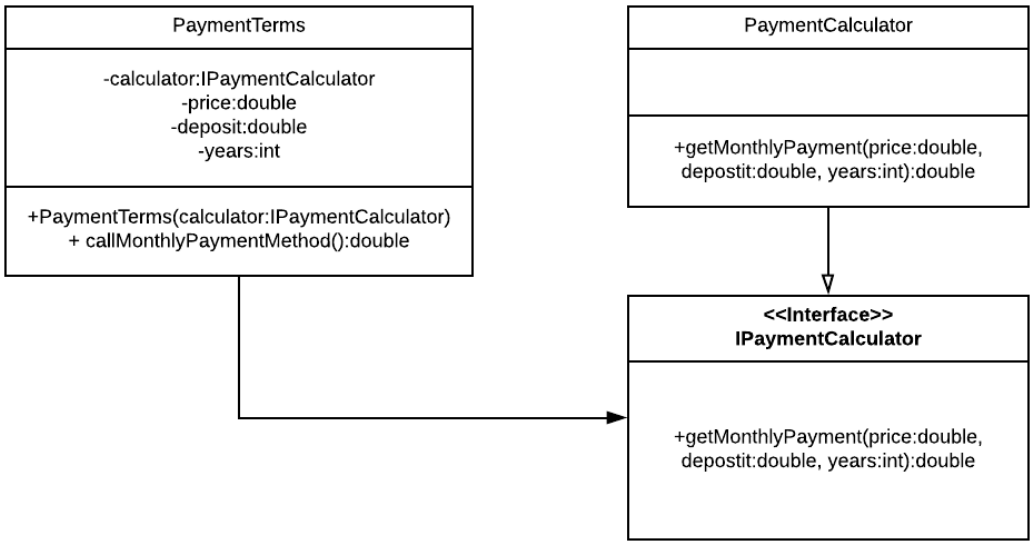


**Abb. 1.4.** Constructor Dependency Injection (Abbildung basierend auf [Car10])

Parameter sind dann als Attribute in der Klasse des Objekts vorhanden. Der Vorteil ist, dass Attribute sich optional gestalten lassen, indem sie mit Default-Werten initialsiert werden.
Analog verhält es sich mit Information Interfaces. Anstelle von Objekten können selbstbeschreibende Nachrichtenformate übergeben werden, wie z.B. XML. Wenn das Nachrichtenformat es zulässt, wie XML, können Elemente optional gesetzt werden.
Eine solch flexible Gestaltung von Interfaces geht zulasten von Verständlichkeit und Testbarkeit, da es beispielsweise nur schwer auffällt, wenn Elemente fehlen. Es kann zudem auch zu Performanzeinbußen kommen. Darum gilt es abzuwägen, welche Interfaces in Zukunft wahrscheinlich erweitert werden und eine solche Flexibilität benötigen ([NR12], S.553-554).

## 1.5 Metamodell-basierte Architekturstile

Metamodell-basierte Architekturstile sind, wie der Name bereits sagt, eigene Architekturstile, d.h. es handelt sich hierbei um eine sehr tiefgreifende Architekturtaktik. Metamodell-basierte Systeme sind von Grund auf auf Änderungen eingestellt und ausgelegt ([NR12], S.555). Man bezeichnet sie auch als Metasysteme, da es sich um Systeme von Systemen handelt ([Dja14], S.1).
Metamodell-basierten Architekturstilen gemeinsam ist ein übergeordnetes Metamodell, welches das Zusammenspiel der Komponenten koordiniert. Die Konfigurationen des Metamodells entscheiden, wie die Komponenten zur Laufzeit zusammengesetzt werden. Oft reicht es, bei Änderungen nur die Konfigurationen des Metamodells anzupassen. Da die Konfigurationen jedoch zur Laufzeit eingelesen und

umgesetzt werden, sind Metasysteme im Hinblick auf Performanz eingeschränkt. Der hohe Grad an Flexibilität schlägt sich zudem auch in einer erhöhten Komplexität nieder, die sowohl das Entwickeln als auch das Testen erschwert ([NR12], S.556).
Es gibt verschiedene Arten von Metasystemen. Diese näher zu erläutern, würde den Rahmen der Ausarbeitung sprengen. Es sei an dieser Stelle jedoch ein Beispiel genannt.
Nach [Dja14] ist ein Metasystem ein groß angelegtes, verteiltes System, dessen Komponenten Enterprise-Systeme sind, die durch den Governance Mechanismus des Metamodells miteinander verknüpft sind, um ein gemeinsames strategisches Ziel zu erreichen ([Dja14], S.1, Z.66-73). Das übergeordnete Governance System leitet, überwacht und koordiniert alle Operationen des Metasystems. Die untergeordneten Enterprise-Systeme, d.h. die Komponenten des Systems, können bei Bedarf rekonfiguriert werden, ohne dass die Implementierung des Systems angepasst werden muss. Zudem lassen sich die Enterprise-Systeme austauschen, es lassen sich zudem Systeme entfernen und neue hinzufügen. Damit ist diese Architektur hochflexibel und passt sich einer sich ändernden Umgebung in co-evolutionärer Weise an ([Dja14], S.2). Anwendung findet ein solcher Architekturstil beispielsweise bei großen Energieverteilungs-, Informations- oder Kommunikationsnetzwerken oder auch bei Netzwerken, welche die Luftfahrt kontrollieren. Dort müssen die Systeme sich an ständig ändernde Informationen und Bedingungen anpassen. Nachteil ist hierbei, wie bei allen Metamodell-basierten Systemen, eine hohe Komplexität ([Dja14], S.1).
Nicht alle Metasysteme sind gleich so komplex, dass sie sich aus mehreren Enterprise-Systemen zusammensetzen. Das Beispiel soll jedoch veranschaulichen, dass Kosten und Nutzen beim Einsatz eines solchen Stils abgewägt werden müssen. Nur, wenn wirklich ein solch hoher Grad an Flexibilät benötigt wird, sollten Metamodelle zum Einsatz kommen.

## 1.6 Variation Points

Diese Architekturtaktik besteht darin, Variation Points zu verwenden.
Dies sind lokale Design-Lösungen, um bestimmte Änderungen an bestimmten Stellen im System zu unterstützen. Hierbei müssen die Stellen, an denen Variation Points erforderlich sind, identifiziert werden. ([NR12], S.556).
Im ersten Abschnitt werden allgemeine Vorgehensweisen präsentiert, worauf im zweiten konkrete Design-Patterns folgen, mit denen sich Variability realsieren lässt.

### 1.6.1 Vorgehensweisen

Ein mögliches Vorgehen besteht darin, Elemente austauchbar zu gestalten. Werden die SOLID-Prinzipien konsequent befolgt, ist dies kein Problem. Im Idealfall sind Implementierung und Interface getrennt und die Implementierung hängt vom Interface ab. Dann kann die Implementierung des Interfaces durch eine andere

Implementierung des Interfaces ausgetauscht werden. Dies entspricht sowohl dem DIP (siehe 1.2.1) als auch dem LSP (siehe 1.2.1).
Eine weitere Vorgehensweise besteht darin, Konfigurationen zu verwenden. Bestimmte Teile des Systemverhaltens lassen sich durch Parametrisierung steuern. Dann lassen sich Änderungen oft alleine durch das Anpassen der Parameter realisieren. .
Variation Points können außerdem erreicht werden, indem selbstbeschreibende Daten sowie eine generische Verarbeitungsweise gewählt werden. Bei solchen selbstbeschreibenden Datenformaten wie z.B. XML lassen sich die Informationen nutzen, um die Daten generisch zu verarbeiten.
Außerdem ist es von Vorteil, die Verarbeitung des Datenformats von der logischen Verarbeitung getrennt zu halten. Dann lässt sich das Datenformat wesentlich einfacher ändern, wie z.B. beim Umstieg von CSV zu XML.
Zudem sollten größere Prozesse stets in Teilschritte unterteilt werden. Dies bietet den Vorteil, dass einzelne Schritte austauschbar sind ([NR12], S.556-557).

### 1.6.2 Beispiele

In diesem Abschnitt werden Design-Patterns vorgestellt, die Variability ermöglichen.

#### Facade

Dieses Pattern bietet Variability bei der Verwendung von Biblithoken, Frameworks oder einer anderen komplexen Menge an Klassen, indem es eine vereinfachte Schnittstelle zu jenen Elementen zur Verfügung stellt ([Shv20], S.210). Diese Schnittstelle enthält nur die Methoden, die wirklich benötigt werden. Damit ist das System nicht mehr so stark an die externe Bibliothek, das Framework etc. gekoppelt und Änderungen daran, die zu erwarten sind, wirken sich weniger stark aus ([Shv20], S.211). Upgrades zu neueren Versionen oder das Austauschen der Software hinter der Schnittstelle stellen mit diesem Design-Pattern kein Problem mehr dar ([Shv20], S.214).

#### Template Method

Dieses Design Pattern bietet Variation Points innerhalb eines Algorithmus. Das Template Method Pattern definiert das Grundgerüst eines Algorithmus in der Superklasse und lässt Subklassen bestimmte Schritte des Algorithmus überschreiben, ohne dabei die Struktur des Algorithmus zu verändern. Die Verwendung bietet sich dann an, wenn der Algorithmus zwar grundlegend gleich bleibt, aber Details stellenweise angepasst werden müssen. Beispielsweise ist in einer Data-Mining-Anwendung Variabilität bezüglich des Datenformats erforderlich. Der Algorithmus sollte nicht jedes Mal neu geschrieben werden müssen, wenn auf ein anderes Datenformat, z.B. von CSF auf PDF, umgestiegen wird. Der Algortihmus wird hierzu in einzelne Methoden unterteilt, wobei jede Methode einen Schritt darstellt. Die Abfolge dieser Methoden wird in eine einzige übergreifende Template Method geschrieben, die entweder abstrakt ist oder eine Default-Implementierung aufweist.

Die Subklasse implementiert dann alle abstrakten Schritte und überschreibt bestimmte Methoden, wenn dies benötigt wird ([Shv20], S.381-383).

#### Weitere Patterns

Weitere Patterns, die Variability erlauben, sind Bridge ([Shv20], S.163-177), Chain of Responsibility ([Shv20], S.250-267) sowie das Visitor Pattern ([Shv20], S.393-408).

## 1.7 Extension Points

Diese Architekturtaktik besteht darin, Extension Points zu nutzen. Extension Points sind Schnittstellen für Erweiterungen ([KK08], S.1). Sie geben vor, an welchen Stellen des Systems Erweiterungen anknüpfen sollen und welche Voraussetzung diese Erweiterungen erfüllen müssen.
Hinter Extension Points steht ein Extension Mechanismus, d.h. die Art und Weise wie die Erweiterung intern vom System unterstützt und umgesetzt wird, inklusive Berücksichtigung der Software-Umgebung ([KK08], S.1).
Bei vielen Standardsoftwares werde Extension Points mitgeliefert, sodass an diese angeknüpft werden kann. So unterstützt die J2EE Plattform beispielsweise die Anbindung neuer Datenbanktypen und externer Systeme([NR12], S.558). Ein anderes Beispiel für Standardsoftware, die Extension Points anbietet, stellt Eclipse dar [KK08]. Plug-Ins können an diese Extension Points anknüpfen ([ecl]).
Die Vorteile bestehen darin, dass den Entwicklern bei der Erweiterung eine Menge Arbeit abgenommen wird. Zudem ist klar, wo erweitert werden muss und wie. Dennoch sollten Extension Points nicht blind genutzt werden, denn schlecht umgesetzte Extension Mechanisms können Performanzeinbußen und erhöhte Komplexität mit sich bringen ([KK08], S.6).
Es sei zudem angemerkt, dass es ggf. sinnvoll sein kann, selbst Extension Points in das System enzubauen, wenn klar ist, dass andere Entwickler in Zukunft das System erweitern werden.

## 1.8 Reliable Change

Bei dieser Architekturtaktik *reliable change* geht es darum, Änderung so zuverlässig und sicher wie möglich umzusetzen ([NR12], S.558). Es gibt einige Maßnahmen, um dies zu erreichen.

### 1.8.1 Konfigurationsmanagement

An erster Stelle ist hier das Konfigurationsmanagement zu nennen. Konfigurationsmangament fasst alle Aktivitäten zusammen, die zur Verwaltung der Konfigurationen dienen ([AS12], S.253, Z.29-30). Eine Konfiguration ist „die Anordnung

eines Computersystems bzw. einer Komponente oder eines Systems, wie sie durch Anzahl, Beschaffenheit und Verbindung seiner Bestandteile definiert ist "([AS12], S.253, Z.24-26). Für Konfigurationsmanagement exisiteren zahlreiche Tools, die eingesetzt werden können. Konfigurationsmanagement schließt Versionskontrolle mit ein ([AS12], S.205).

### 1.8.2 Automatisieren

Vorgänge wie der Build-Prozess, der Release-Prozess und das Testen sollten automatisiert werden. So werden die Prozesse zuverlässig, konsistent und lassen sich wiederholen mit demselben Ergebnis.
Automatisiertes Testen sollte auf keinen Fall vernachlässigt werden, denn es ist wichtig, sicherzustellen, dass sich mit den Änderungen keine Fehler eingeschlichen haben. Dafür müssen hunderte bis tausende Tests durchgeführt werden, was sich manuell nicht mit vertretbarem Aufwand erreichen lässt ([NR12], S.559). Auch für das Automatisierte Testen existieren zahlreiche Tools.

### 1.8.3 Dependency Analysis

Eine Depency Analysis lässt sich mit Tools durchführen und gibt hilfreichen Aufschluss über unentdeckte Abhängigkeiten, die sonst eventuell übersehen worden wären. Anhand von Dependency Analysis lassen sich die Auswirkungen von Änderungen abschätzen ([NR12], S.558).

### 1.8.4 Continuos Integration

Continuous Integration beschreibt die Vorgehensweise, geänderte oder neue Systemelemente so bald wie möglich in das System zu integrieren und zu testen ([NR12], S.559).
Es ist hingegen zu vermeiden, zu warten und alle Elemente auf einmal zu integrieren. Eine solche nicht inkrementelle *big bang* Integration sorgt dafür, dass alle Probleme gleichzeitig auftreten. Die Lokalisierung und Behebung von Fehlern wird unnötig erschwert ([AS12], S.60).

### 1.8.5 Änderungen zurücksetzen

Nichts bietet so viel Sicherheit wie die Option, eine Änderung jederzeit wieder rückgängig machen zu können. Wenn Konfigurationsmanagement beachtet wird, gibt es zwangsläufig eine Versionsverwaltung. Wird hierfür z.B. das Versionsverwaltungssystem Git verwendet, können Änderungen einfach rückgängig gemacht werden, z.B. mittels der Befehle *git revert* oder *git reset*.

# Literaturverzeichnis


AS12. Andreas Spillner, Tilo Linz: *Basiswissen Softwaretest.* dpunkt.verlag, 2012.

BL09a. Bernhard Lahres, Gregor Rayman: *Objektorientierte Programmierung - Das umfassende Handbuch, Kapitel 2.2 Die Kapselung von Daten.* Galileo Computing, 2009. `http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_02_002.htm#mjd081ab77f1ee1d3bf3f15782d17777a4` Eingesehen am 02.07.2020.

BL09b. Bernhard Lahres, Gregor Rayman: *Objektorientierte Programmierung - Das umfassende Handbuch, Kapitel 3.6 Prinzip 6: Umkehr der Abhängigkeiten.* Galileo Computing, 2009. `http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_03_006.htm` Eingesehen am 02.07.2020.

Car10. Carr, Richard: *Dependency Injection - Implementing Constructor Injection*, 2010. `http://www.blackwasp.co.uk/DependencyInjection_3.aspx` Eingesehen am 11.07.2020.

Dja14. Djavanshir, R.: *Exploring Metasystems.* IT Professional, 16(06):4–7, nov 2014.

ecl. *Eclipse Documentation, Extensions and Extension Points.* `https://www.ibm.com/support/knowledgecenter/SSQ2R2_14.2.0/org.eclipse.pde.doc.user/concepts/extension.htm` Eingesehen am 10.07.2020.

Kan03. Kandt, Kirk: *Software design principles and practices.* 2003.

KK08. Klatt, Benjamin und Klaus Krogmann: *Software Extension Mechanisms.* Seiten 11–18, 01 2008.

Lis87. Liskov, Barbara: *Keynote Address - Data Abstraction and Hierarchy.* SIGPLAN Not., 23(5):17?34, Januar 1987.

Mar09. Martin, Robert C.: *Clean Code: a handbook of agile software craftsmanship.* Prentice Hall Pearson Education, 2009.

Mar18. Martin, Robert C.: *Clean Architecture: Das Praxis-Handbuch für professionelles Softwaredesign, 1. Auflage.* mitp Verlag, 2018.

Mey88. Meyer, Bertrand: *Object Oriented Software Construction.* Prentice Hall, 1988.

NR12. Nick Rozanski, Eoin Woods: *Software Systems Architecture, Second Edition*. Addison-Wesley, 2012.

Shv20. Shvets, Alexander: *Dive Into Design Patterns*. Selbstverlag, 2020.

# A

# Glossar

| | |
|---|---|
| DisASTer | DisASTer (Distributed Algorithms Simulation Terrain), A platform for the Implementation of Distributed Algorithms |
| DSM | Distributed Shared Memory |
| AC | Linearisierbarkeit (atomic consistency) |
| SC | Sequentielle Konsistenz (sequential consistency) |
| WC | Schwache Konsistenz (weak consistency) |
| RC | Freigabekonsistenz (release consistency) |