IT kompakt

Karl Eilebrecht
Gernot Starke

Patterns kompakt

Entwurfsmuster für effektive Softwareentwicklung

6. Auflage

Springer Vieweg

IT kompakt

Die Bücher der Reihe „IT kompakt“ zu wichtigen Konzepten und Technologien der IT:

- ermöglichen einen raschen Einstieg,
- bieten einen fundierten Überblick,
- eignen sich für Selbststudium und Lehre,
- sind praxisorientiert, aktuell und immer ihren Preis wert.

Karl Eilebrecht • Gernot Starke

Patterns kompakt

Entwurfsmuster für effektive Softwareentwicklung

6., erweiterte und aktualisierte Auflage

Springer Vieweg

Karl Eilebrecht
Karlsruhe, Deutschland

Gernot Starke
Köln, Deutschland

ISSN 2195-3651 ISSN 2195-366X (electronic)
IT kompakt
ISBN 978-3-658-43233-1 ISBN 978-3-658-43234-8 (eBook)
https://doi.org/10.1007/978-3-658-43234-8

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über https://portal.dnb.de abrufbar.

Planung/Lektorat: Petra Steinmueller
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Das Papier dieses Produkts ist recyclebar.

Danksagung

Wir bedanken uns bei allen, die das Thema Patterns in der Softwareentwicklung begründet haben, in erster Linie der Gang-of-Four, Martin Fowler und Robert „Uncle Bob“ Martin sowie den zahlreichen Autorinnen und Autoren der{Euro|Viking|Chili.*}PloP-Konferenzen. Ihre Kreativität und Offenheit hat die Software-Welt besser gemacht! Herzlichen Dank auch unseren zahlreichen Kolleginnen und Kollegen sowie all jenen, die anlässlich von Gernots Seminaren und Schulungen regelmäßig für fruchtbare Diskussionen über Softwarearchitekturen, Software-Entwurf und Patterns sorgen. Stefan Wießner und Jürgen Bloß aus dem KOMPR-Team sei gedankt für Espresso und wertvolle Einsichten. Michael „Agent-M“ Krusemark sowie Wolfgang Korn leisteten Erste Hilfe in C++. Schließlich geht unser Dank an Karsten Himmer nach Berlin für den ersten Hamster auf Pattern-Basis. Dr. Martin Haag und Markus Woll riskierten freundlicherweise vorab einen prüfenden Blick auf die Neuerungen der zweiten Auflage. Dmitry Dodin gab großartige „Formelumformungsnachhilfe“ für Auflage Nummer 5. Allen Leserinnen und Lesern, die so fleißig Verbesserungsvorschläge per E-Mail geschickt haben, an dieser Stelle ein kollektives „Danke und weiter so“!

K. E.: Ich bedanke mich bei meinen Kolleginnen und Kollegen für anregende Diskussionen und natürlich bei meinen Eltern für ihre Geduld.

G. S.: Ich danke meiner Traumfrau Cheffe Uli sowie meinen Kindern Lynn und Per, ihr seid die bestmögliche Familie. Danke auch an Karl – es macht riesigen Spaß, mit Dir Bücher zu schreiben!

Einleitung

„This book is meant to be played,
rather than to be read in an armchair".
Jerry Coker et al.:
Patterns for Jazz, Studio P/R, 1970

Wozu benötigen wir Entwurfsmuster?

Entwurfsmuster lösen bekannte, wiederkehrende Entwurfsprobleme. Sie fassen Design- und Architekturwissen in kompakter und wiederverwertbarer Form zusammen. Sowohl in der Softwareentwicklung als auch bei der Softwarearchitektur bieten Entwurfsmuster wertvolle Unterstützung bei der Wiederverwendung erprobter Designentscheidungen. Sie geben Hinweise, wie Sie vorhandene Entwürfe flexibler, verständlicher oder auch performanter machen können.

In komplexen Software-Projekten kann der angemessene Einsatz von Mustern das Risiko von Entwurfsfehlern deutlich senken.

Warum ein weiteres Buch über Entwurfsmuster?

Seit dem Kultbuch der berühmten „Gang-of-Four" [GoF] hat es viele Konferenzen und noch mehr Literatur zu diesem Thema gegeben – der Fundus an verfügbaren Entwurfsmustern scheint nahezu grenzenlos: mehrere tausend Druckseiten, viele hundert

Seiten im Internet. Für Praktikerinnen und Praktiker inmitten von Projektstress mit konkreten Entwurfsproblemen stellt sich das Problem, aus der Fülle der verfügbaren Muster die jeweils geeigneten auszuwählen. Wenn Sie sich mit Softwarearchitektur, -design oder -entwicklung befassen, benötigen Sie Unterstützung bei konkreten Entwurfsproblemen, und das auf möglichst engem Raum konzentriert.

Für solche Situationen haben wir dieses Buch geschrieben: Es erleichtert den Entwurf flexibler, wartbarer und performanter Anwendungen, indem es das Wissen der umfangreichen Pattern-Literatur auf praxisrelevante Muster für kommerzielle Software-Systeme konzentriert. Die kompakte Darstellung erleichtert den Überblick und damit die Anwendbarkeit der ausgewählten Muster.

Ganz bewusst verzichten wir bei den vorgestellten Mustern auf ausführliche Implementierungsanleitungen und Beispielcode in gedruckter Form. Anstelle dessen erhalten Sie Hinweise auf weitere Informationsquellen und finden ausführbare Beispiele auf unserer Webseite (siehe [PK]). Die Praxis-Erfahrenen unter Ihnen können anhand der kompakten Darstellung die Entwurfsentscheidung für oder gegen den Einsatz bestimmter Muster treffen. Grundlegende Kenntnisse einer objektorientierten Programmiersprache und der UML setzen wir in diesem Buch voraus.

Ein Wort zur Vorsicht

„Used in the wrong place,
the best patterns will fail“.
Jerry Coker et al.:
Patterns for Jazz, Studio P/R, 1970

Patterns eignen sich hervorragend zur Kommunikation über Entwurfsentscheidungen. Sie können helfen, Ihre Entwürfe flexibler zu gestalten. Häufig entstehen durch die Anwendung von Patterns jedoch zusätzliche Klassen oder Interfaces, die das System aufblähen. Eine der wichtigsten Regeln beim Software-Entwurf lautet: Halten Sie Ihre Entwürfe so einfach wie möglich. In diesem

Sinne möchten wir Sie, trotz aller Begeisterung für Entwurfsmuster, zu vorsichtigem Umgang mit diesen Instrumenten auffordern. Ein einfach gehaltener Entwurf ist leichter verständlich und übersichtlicher. Hinterfragen Sie bei der Anwendung von Mustern, ob Ihnen die Flexibilität, Performance oder Wiederverwendbarkeit nach der Anwendung eines Musters einen angemessenen Mehrwert gegenüber dem ursprünglichen Entwurf bieten. In Zweifelsfällen wählen Sie das einfachere Konzept.

Die meisten Entwurfsmuster in diesem Buch konzentrieren sich auf objektorientierte Systeme. Fans strikt funktionaler Sprachen (etwa Haskell, Clojure), deklarativer Programmierung (etwa Prolog, SQL) oder solcher Sprachen, die mehreren Paradigmen folgen (etwa Scala, Erlang, Go) mögen uns verzeihen, dass wir auf ihre „Lieblinge“ nicht gesondert eingehen – sonst hätten wir das Buch nicht so kompakt halten können.

Die Pattern-Schablone

Wir haben für dieses Buch bewusst eine flexible Schablone für Muster gewählt und ergänzende Informationen je nach Pattern aufgeführt.

- **Zweck**: Wozu dient das Pattern?
- **Szenario** (noch weitere Teile sind optional): Ein Beispielszenario für das Pattern oder das Problem.
- **Problem/Kontext**: Der strukturelle oder technische Kontext, in dem ein Problem auftritt und auf den die Lösung angewendet werden kann.
- **Lösung**: Die Lösung erklärt, wie das Problem im Kontext gelöst werden kann. Sie beschreibt die Struktur, hier meist durch UML-Diagramme. Christopher Alexander, Begründer der Pattern-Bewegung, selbst schreibt dazu: „Wenn Du davon kein Diagramm zeichnen kannst, dann ist es kein Muster.“ [Alexander], S. 267.
- **Vorteile**: Welche Vorteile entstehen aus der Anwendung dieses Patterns?

- **Nachteile**: In manchen Fällen können durch die Anwendung eines Musters Nachteile entstehen. Dies ist häufig der Fall, wenn gegensätzliche Aspekte (wie etwa Performance und Flexibilität) von einem Muster betroffen sind.
- **Verwendung**: Hier zeigen wir Ihnen Anwendungsgebiete, in denen das Muster seine spezifischen Stärken ausspielen kann.
- **Varianten**: Manche Patterns können in Variationen oder Abwandlungen vorkommen, die wir Ihnen in diesem (optionalen) Abschnitt erläutern.
- **Verweise**: Dieser Abschnitt enthält Verweise auf verwandte Muster sowie auf weiterführende Quellen.

Sorry

Ungewöhnlich – eine Entschuldigung am Anfang eines Buches: Wir möchten Sie schon jetzt um Nachsicht bitten, falls wir Ihr Lieblings-Pattern in unserer Darstellung ignorieren. Leider müssen wir aus dem riesigen Fundus interessanter Patterns eine relativ kleine Auswahl treffen – und daher viele Muster außen vorlassen. Architektur- oder Cloud-Patterns etwa bilden eigene Universen, die wir nicht abdecken können. Es tut uns wirklich leid um Blackboard (eine wichtige Grundlage regel- oder wissensbasierter Systeme), um die vielen Patterns der Systemintegration, um die technikspezifischen Muster und Idiome und viele andere. Aber in diesem Buch finden Sie eine solide Basis und Verweise auf weitere interessante Literatur – und wir bleiben natürlich dran, versprochen!

Inhaltsverzeichnis

1 Grundlagen des Software-Entwurfs

Für den Entwurf von Softwaresystemen gelten einige fundamentale Prinzipien, die auch die Basis der meisten Entwurfsmuster bilden. Im Folgenden stellen wir Ihnen einige dieser Prinzipien kurz vor. Detaillierte Beschreibungen finden Sie in [Riel], [Eckel], [Fowler] sowie [Martin2].

Als weitere Hilfe stellen wir Ihnen einige Heuristiken vor: allgemeine Leitlinien, die Sie in vielen Entwurfssituationen anwenden können. [Rechtin] und [Riel] bieten umfangreiche Sammlungen solcher Heuristiken zum Nachschlagen. [Rechtin] bezieht sich dabei grundsätzlich auf (System-)Architekturen, [Riel] gezielt auf objektorientierte Systeme.

Sowohl Prinzipien als auch Heuristiken können Ihnen helfen, sich in konkreten Entwurfssituationen für oder gegen die Anwendung eines Entwurfsmusters zu entscheiden.

1.1 Entwurfsprinzipien

Einfachheit vor Allgemeinverwendbarkeit

Bevorzugen Sie einfache Lösungen gegenüber allgemeinverwendbaren. Letztere sind in der Regel komplizierter. Machen Sie normale Dinge einfach und besondere Dinge möglich.

K. Eilebrecht, G. Starke, *Patterns kompakt*, IT kompakt,
https://doi.org/10.1007/978-3-658-43234-8_1

Prinzip der minimalen Verwunderung

(Principle of least astonishment) Erstaunliche Lösungen sind meist schwer verständlich.

Vermeiden Sie Wiederholung

(DRY: Don't Repeat Yourself, OAOO: Once And Once Only) Vermeiden Sie Wiederholungen von Struktur und Logik, wo sie nicht unbedingt notwendig bzw. sinnvoll sind. Betrachten Sie diese Empfehlung bitte nicht als Dogma. Wiederholung kann angebracht sein, wenn ihre strikte Vermeidung zu unerwünschten technischen oder organisatorischen Abhängigkeiten führt. Unter [INNOQ] finden Sie einen interessanten Vortrag, der diese Problematik anhand eines Beispiels erläutert.

Prinzip der einzelnen Verantwortlichkeit

(Single-Responsibility Principle, Separation-of-Concerns) Jeder Baustein eines Systems sollte eine klar abgegrenzte Verantwortlichkeit besitzen. Auf objektorientierte Systeme gemünzt heißt das: Vermeiden Sie es, Klassen mehr als eine Aufgabe zu geben. Robert Martin formuliert es so: „Jede Klasse sollte nur genau einen Grund zur Änderung haben." [Martin], S. 95. Beispiele für solche Verantwortlichkeiten (nach [Larman]):

- Etwas wissen; Daten oder Informationen über ein Konzept kennen.
- Etwas können; Steuerungs- oder Kontrollverantwortung.
- Etwas erzeugen.

Das Prinzip ist auch auf Methodenebene anwendbar. Eine Methode sollte für eine bestimmte Aufgabe zuständig sein und nicht (durch Parameter gesteuert) für mehrere. [Martin2] legt das sehr streng aus und fordert sogar den Verzicht auf boolesche

Steuer-Flags. Eine Methode wie `writeOutput(Data data, boolean append)` müsste dann in zwei Methoden gesplittet werden. Wir sehen das etwas weniger streng und empfehlen, dass eine Methode eine durch Methodennamen und Kommentar ersichtliche Aufgabe erfüllen und keine undokumentierten Seiteneffekte haben sollte. Unter der Überschrift *Command/Query-Separation Principle* fordert [Meyer], dass eine Methode, die eine Information über ein Objekt liefert, nicht gleichzeitig dessen Zustand ändern soll.

Offen-Geschlossen-Prinzip

(Open-Closed Principle) Software-Komponenten sollten offen für Erweiterungen, aber geschlossen für Änderungen sein. Es ist eleganter und robuster, einen Klassenverbund durch Hinzufügen einer Klasse zu erweitern, als den bestehenden Quellcode zu modifizieren. Dieses Prinzip ist eng verwandt mit dem Prinzip der einzelnen Verantwortlichkeit und ebenso auf Methodenebene anwendbar. Eine entsprechende Klasse oder Methode wird *nur* im Rahmen der Verbesserung oder Korrektur der Implementierung geändert (geschlossen für Änderungen). Neue Funktionalität wird dagegen durch eine neue Klasse bzw. Methode hinzugefügt (offen für Erweiterung). Einige Patterns (z. B. → Strategy (Abschn. 3.5) oder → PlugIn (Abschn. 6.3)) unterstützen dieses Prinzip.

Prinzip der gemeinsamen Wiederverwendung

(Common Reuse Principle) Die Klassen innerhalb eines Pakets sollten gemeinsam wiederverwendet werden. Falls Sie eine Klasse eines solchen Pakets wiederverwenden, können Sie alle Klassen dieses Pakets wiederverwenden. Anders formuliert: Klassen, die gemeinsam verwendet werden, sollten in ein gemeinsames Paket verpackt werden. Dies hilft, zirkuläre Abhängigkeiten zwischen Paketen zu vermeiden.

Keine zirkulären Abhängigkeiten

(Acyclic Dependency Principle) Klassen und Pakete sollten keine zirkulären (zyklischen) Abhängigkeiten enthalten (siehe Abb. 1.1). Solche Zyklen sollten in der Softwarearchitektur Teufelskreise heißen: Sie erschweren die Wartbarkeit und verringern die Flexibilität, unter anderem, weil Zyklen nur als Ganzes testbar sind. In objektorientierten Systemen können Sie zirkuläre Abhängigkeiten entweder durch Verschieben einzelner Klassen oder Methoden auflösen, oder Sie kehren eine der Abhängigkeiten durch eine Vererbungsbeziehung um.

Prinzip der stabilen Abhängigkeiten

(Stable Dependencies Principle) Führen Sie Abhängigkeiten möglichst in Richtung stabiler Bestandteile ein. Vermeiden Sie Abhängigkeiten von volatilen (d. h. häufig geänderten) Bestandteilen.

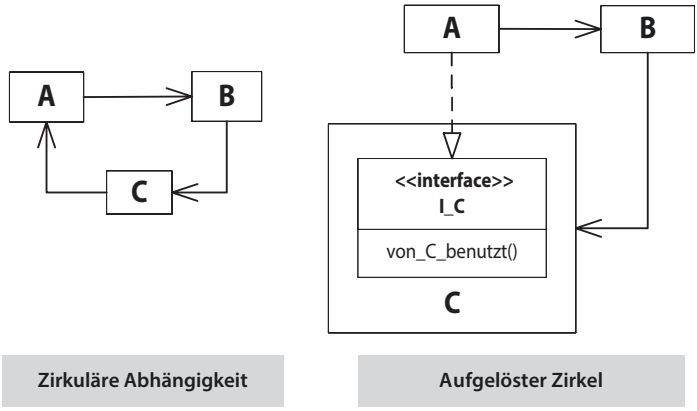

Abb. 1.1 Auflösung von Abhängigkeitszyklen

Liskov'sches Substitutionsprinzip

(Liskov Substitution Principle) Unterklassen sollen anstelle ihrer Oberklassen einsetzbar sein. Sie sollten beispielsweise in Unterklassen keine Methoden der Oberklassen durch leere Implementierungen überschreiben. Stellen Sie beim Überschreiben von Methoden aus einer Oberklasse sicher, dass die Unterklasse in jedem Fall für die Oberklasse einsetzbar bleibt. Denken Sie besonders beim Design von Basisklassen und Interfaces an dieses Prinzip. Unbedacht eingeführte Methoden, die später doch nicht für alle Mitglieder der Klassenhierarchie passen, werden Sie nur schwer wieder los.

Prinzip der Umkehrung von Abhängigkeiten

(Dependency Inversion Principle) Implementierungen, die eine Methode oder einen Service nutzen, sollten möglichst von Abstraktionen (d. h. abstrakten Klassen, Interfaces, API-Definitionen), nicht aber von konkreten Implementierungen abhängig sein (siehe Abb. 1.2). Abstraktionen dürfen nicht von konkreten Implementierungen abhängen.

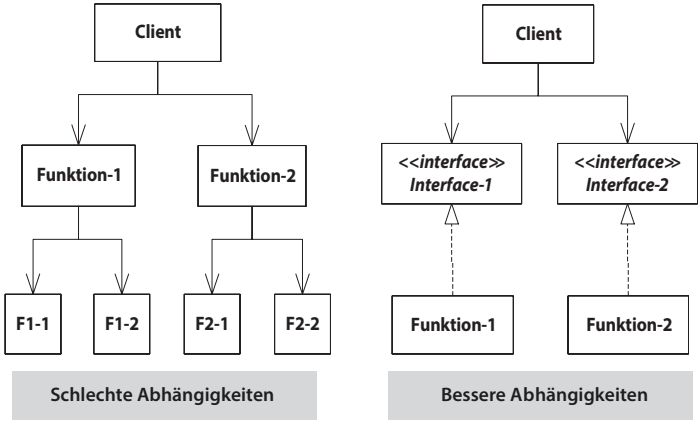

Abb. 1.2 Umkehrung von Abhängigkeiten

Prinzip der Abtrennung von Schnittstellen

(Interface Segregation Principle) Clients sollten nicht von Diensten abhängen, die sie nicht benötigen (siehe Abb. 1.3). Interfaces gehören ihren Clients, nicht den Klassenhierarchien, die diese Interfaces implementieren. Entwerfen Sie Schnittstellen nach den Clients, die diese Schnittstellen brauchen.

In [Fowler] finden Sie eine Variante dieses Prinzips unter der Bezeichnung *Separated Interface* als Pattern beschrieben (siehe Abb. 1.4). Die Schnittstellen liegen dabei von ihren Implementierungen getrennt in eigenen Paketen. Zur Laufzeit müssen Sie eine konkrete Implementierung auswählen. Dafür bieten sich → Erzeugungsmuster (Kap. 2) oder auch → PlugIn (Abschn. 6.3) an.

Prinzip solider Annahmen

Bauen Sie Ihr Haus nicht auf Sand! Die Gefahr versteckter Annahmen zieht sich durch den gesamten Prozess der Software-Entwicklung. Das beginnt bereits damit, dass Sie nicht einfach davon ausgehen können, dass Sie vorhandene Strukturen wie

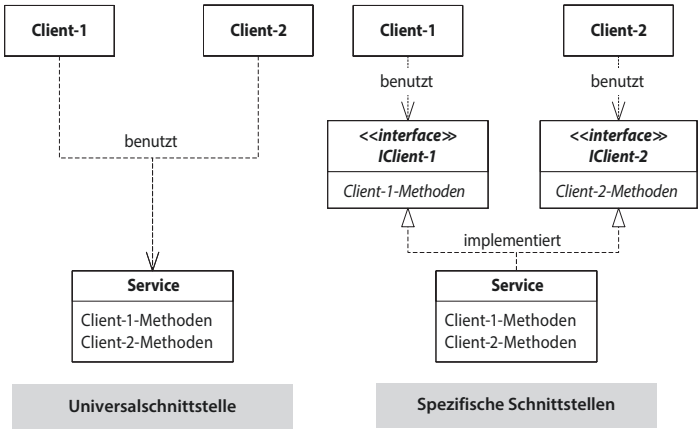

Abb. 1.3 Abtrennung von Schnittstellen

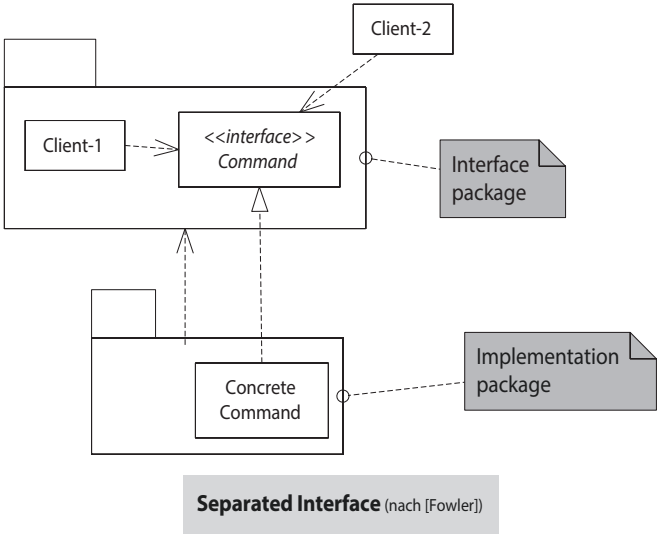

Abb. 1.4 Paket-Trennung: Separate Implementierung

einen verteilten Cache für Ihre Applikation mitbenutzen dürfen. Besonders unangenehm können Annahmen auch auf der Implementierungsebene sein. Vor Jahren wurde in einem Forum eine Strategie beschrieben, mit der angeblich ein bekanntes Problem mit der Java Garbage Collection umgangen werden könnte. Ein wenig skeptisch, aber doch neugierig, bat ich (K. E.) um eine Erklärung anhand von Fakten wie der Spezifikation oder weiterführender Dokumentation. Das Resultat war eine ziemliche Abfuhr: „[…] Great. Feel free to program to what the API says. And I will continue to program to what the API actually does […]". Nun, auf den ersten Blick klingt diese Argumentation geradezu verlockend plausibel. Andererseits machen Sie sich dabei jedoch abhängig von einer ganz bestimmten Implementierung – ohne jede Garantie. Interfaces verlieren ihren Sinn. Es bildet sich ein Nährboden für Legenden und versteckte Fehlerquellen, bevorzugt in Verbindung mit verschiedenen Releases oder Plattformen.

Nicht fundierte oder gar implizite Annahmen über Systeme oder Schnittstellen sollten Sie also unbedingt vermeiden!

Falls dies nicht möglich ist, z. B. bei der Verwendung unfertiger oder mangelhaft dokumentierter Module bzw. bei der Erstellung von Prototypen, dokumentieren Sie Ihre Annahmen sorgfältig.

Konvention vor Konfiguration

(Convention over Configuration, Configuration by Exception) Bei diesem Ansatz wird die Konfiguration eines Frameworks oder einer Software-Komponente durch Konventionen und sinnvolle Voreinstellungen erleichtert.

In der Software-Entwicklung sind, bedingt durch den Einsatz komplexer Frameworks, sehr viele Einstellungen konfigurierbar, beispielsweise mittels XML-, JSON oder YAML-Dateien. Der Höhepunkt war wohl mit EJB 2.1 erreicht. Entwicklerinnen und Entwickler waren schon fast genötigt, weitere Frameworks (z. B. xdoclet) einzusetzen, um die Konfigurationsseuche halbwegs in den Griff zu bekommen. Bedingt durch die Einführung von Annotationen im Quellcode und mit EJB 3 ist vieles besser geworden. Die Ursprungsfrage bleibt aber: Warum muss ich etwas aktiv konfigurieren, wenn es doch in 99 % aller Fälle immer gleich ist, einem einfachen Schema folgt oder schlimmstenfalls für meine Anwendung völlig irrelevant ist?

Moderne Frameworks wie Spring (https://www.spring.io/) und insbesondere Spring Boot drehen den Spieß um. Sie führen für Framework-Einstellungen oder auch das Mapping von Entitäten (siehe auch → O/R-Mapping (Abschn. 7.1)) strikte Konventionen und Standardwerte ein. Hält man sich bei der Framework-Verwendung an die Konventionen, muss man nur noch vergleichsweise wenig individuell einstellen. Nur in den Fällen, in denen das geplante Szenario ein Abweichen von den Konventionen erfordert, ist Handarbeit nötig – dann allerdings meist viel. Dieses Paradigma ist nicht nur auf Frameworks sondern auch auf kleinere Komponenten anwendbar, sofern diese Konfigurationseinstellungen vorsehen. Machen Sie sich Gedanken über Kon-

ventionen und sinnvolle Defaultwerte. Das erleichtert anderen, Ihre Software zu evaluieren und zu integrieren. Nachteilig ist, dass viel Arbeit in der Ausarbeitung langlebiger Konventionen steckt. Zudem werden Weiterentwicklungen und Änderungen aufwendiger.

Einen schönen Artikel von Nicholas Chen dazu finden Sie in [NCHEN].

Prinzip der Verhältnismäßigkeit

Prüfen Sie Ihre Entscheidungen immer mehrdimensional. Während der Begriff der Angemessenheit im Rahmen der Softwarequalität (ISO/IEC 25010) auf die erstellte Lösung, also das Produkt und seine Eigenschaften abzielt, geht es uns bei der Verhältnismäßigkeit auch um den Prozess der Entwicklung, den Betrieb und die spätere Wartung, also den kompletten Lebenszyklus. Oft müssen mehrere, nicht immer nur technische Aspekte sorgfältig gegeneinander abgewogen werden. Verhältnismäßigkeit bedeutet, Prioritäten zu erkennen und ggf. zu setzen. Beispielsweise ist Wiederverwendung eine gute Praxis. Eine bekannte Bibliothek sollten Sie einer Eigenimplementierung normalerweise vorziehen. Andererseits ist es nicht verhältnismäßig, wegen einer trivialen Funktion umfangreiche Abhängigkeiten in ein Projekt einzuschleppen. Damit wird nicht nur Ihr Deployment unnötig groß, Sie erhöhen auch die Risiken (z. B. Versionskonflikte, Sicherheit, Wartung). Auch auf der Architekturebene müssen ambitionierte Entwurfsentscheidungen immer mit der Realität abgeglichen werden. Die Einführung einer neuen Technologie oder eines neuen Paradigmas kann trotz offenkundiger technischer Vorteile unverhältnismäßig sein, wenn sich niemand außer Ihnen im Projekt damit auskennt, wenn dadurch die Architektur einen „exotischen Balkon“ bekommt, wenn zu viele umgebende Abläufe verkompliziert werden, wenn damit unfundiert eine Richtungsentscheidung getroffen wird. Gespräche mit unterschiedlichen Beteiligten (aus Entwicklung, Test, Betrieb oder von der Fachseite) können Fehlentwicklungen vermeiden. 😉

1.2 Heuristiken des objektorientierten Entwurfs

Entwurf von Klassen und Objekten

- Eine Klasse sollte genau eine Abstraktion („Verantwortlichkeit“) realisieren.
- Kapseln Sie zusammengehörige Daten und deren Verhalten in einer gemeinsamen Klasse (*Maximum Cohesion*).
- Wenn Sie etwas Schlechtes, Schmutziges oder Unschönes tun müssen, dann kapseln Sie es zumindest in einer einzigen Klasse.
- Kapseln Sie Aspekte, die variieren können (vgl. „information hiding“ [Panas72], [Panas71]). Das → Bridge-Pattern (Abschn. 4.2) wendet diese Heuristik an.
- Vermeiden Sie übermäßig mächtige Klassen („Poltergeister“ oder „Gott-Klassen“).
- Eine Implementierung, die Methoden einer Klasse benutzt, soll ausschließlich von deren öffentlichen Schnittstellen abhängen. Klassen sollten nicht von aufrufenden Implementierungen abhängig sein. Anders formuliert: Wenn eine Methode mit sehr generischem Namen abhängig vom übergebenen Datentyp oder Parametern ein vollkommen anderes Verhalten zeigt, dann ist normalerweise etwas faul.
- Halten Sie die öffentlichen Schnittstellen von Klassen möglichst schlank. Entwerfen Sie so privat wie möglich.
- Benutzen Sie Attribute, um Veränderungen von Werten auszudrücken. Um Veränderung im Verhalten auszudrücken, können Sie Überlagerung von Methoden verwenden.
- Vermeiden Sie es, den Typ eines Objekts zur Laufzeit zu ändern. Einige Sprachen erlauben dies zwar mittels mehr oder weniger gut dokumentierter Hacks. Dies geschieht dann aber in der Absicht, eine existierende Instanz zu einem anderen Typ kompatibel zu machen. Es ist deutlich besseres Design, ein solches Problem mit dem → Decorator-Pattern (Abschn. 4.3) oder dem State-Pattern (s. [GoF]) zu lösen. Das .NET-