Patterns kompakt

Werke der „kompakt-Reihe“ zu wichtigen Konzepten und Technologien der IT-Branche:

- ermöglichen einen raschen Einstieg,
- bieten einen fundierten Überblick,
- sind praxisorientiert, aktuell und immer ihren Preis wert.

Bisher erschienen:

- Heide Balzert
  UML kompakt, 2. Auflage
- Andreas Böhm / Elisabeth Felt
  e-commerce kompakt
- Christian Bunse / Antje von Knethen
  Vorgehensmodelle kompakt, 2. Auflage
- Holger Dörnemann / René Meyer
  Anforderungsmanagement kompakt
- Christof Ebert
  Outsourcing kompakt
- Christof Ebert
  Risikomanagement kompakt
- Karl Eilebrecht / Gernot Starke
  Patterns kompakt, 3. Auflage
- Andreas Essigkrug / Thomas Mey
  Rational Unified Process kompakt, 2. Auflage
- Peter Hruschka / Chris Rupp / GernotStarke
  Agility kompakt, 2. Auflage
- Arne Koschel / Stefan Fischer / Gerhard Wagner
  J2EE/Java EE kompakt, 2. Auflage
- Michael Kuschke / Ludger Wölfel
  Web Services kompakt
- Torsten Langner
  C# kompakt
- Pascal Mangold
  IT-Projektmanagement kompakt, 3. Auflage
- Michael Richter / Markus Flückiger
  Usability Engineering kompakt
- Thilo Rottach / Sascha Groß
  XML kompakt: die wichtigsten Standards
- SOPHIST GROUP / Chris Rupp
  Systemanalyse kompakt, 2. Auflage
- Gernot Starke / Peter Hruschka
  Software-Architektur kompakt
- Ernst Tiemeyer
  IT-Controlling kompakt
- Ernst Tiemeyer
  IT-Servicemanagement kompakt
- Ralf Westphal
  .NET kompakt
- Ralf Westphal / Christian Weyer
  .NET 3.0 kompakt

Karl Eilebrecht / Gernot Starke

Patterns kompakt

Entwurfsmuster für effektive Software-Entwicklung

3. Auflage

Spektrum
AKADEMISCHER VERLAG

Autoren:
Karl Eilebrecht
E-Mail: Karl.Eilebrecht@web.de

Dr. Gernot Starke
E-Mail: gs@gernotstarke.de

Für weitere Informationen zum Buch siehe
http://www.patterns-kompakt.de

Wichtiger Hinweis für den Benutzer

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

3. Auflage 2010

Spektrum Akademischer Verlag ist ein Imprint von Springer

10 11 12 13 14 5 4 3 2 1

Planung und Lektorat: Dr. Andreas Rüdinger, Barbara Lühker
Herstellung und Satz: Crest Premedia Solutions (P) Ltd, Pune, Maharashtra, India
Umschlaggestaltung: SpieszDesign, Neu–Ulm

ISBN 978-3-8274-2525-6

Inhalt

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 Software-Entwicklern als auch Software-Architekten 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 Praktiker inmitten von Projektstress und konkreten Entwurfsproblemen stellt sich das Problem, aus der Fülle der verfügbaren Muster die jeweils geeigneten auszuwählen. Software-Architekten, -Designer und -Entwickler benötigen 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. Anstelle dessen erhalten Sie Hinweise auf weitere Informationen. Die erfahrenen Praktiker unter Ihnen können anhand der kompakten Darstellung die Entwurfsentscheidung für oder gegen den Einsatz bestimmter Muster treffen. Grundlegende Kenntnisse einer objektorientierten Programmiersprache und 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 den einfacheren Weg.

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 auftreten. 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 darstellen.
- **Verweise**: Dieser Abschnitt enthält Verweise auf verwandte Muster sowie auf weiterführende Quellen.

Kolophon

Das Titelbild zeigt, Kenner mögen uns diese Erläuterung verzeihen, ein leicht abstrahiertes Fassade-Pattern. Es symbolisiert die Intention dieses Buches – den vereinfachten und kompakten Zugang zu einem äußerst umfangreichen System mit komplexer innerer Struktur.

Die UML-Modelle entstanden mit Microsoft Visio® auf Basis der hervorragenden und effizienten Schablonen von Pavel Hruby, frei verfügbar unter www.phruby.com (Martin Fowler und Uncle Bob: Danke

für den Tipp). Den weiten Weg vom File via EPS in Richtung Papier erleichterte uns das zeitlose GhostScript.

Danksagung

Wir bedanken uns bei Pattern-Erfindern, in erster Linie der Gang-of-Four, Martin Fowler und Robert „Uncle Bob“ Martin sowie den zahlreichen Autoren der {Euro|Viking|Chili|.*}PloP-Konferenzen. Ihre Kreativität und Offenheit hat die Software-Welt besser gemacht! Herzlichen Dank auch unseren zahlreichen Kollegen sowie Seminar- und Schulungsteilnehmern für die fruchtbaren Diskussionen über Software-Architekturen, Software-Entwurf, Patterns und grünen Tee. 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.

K. E.:
Ich bedanke mich bei meinen Kollegen von CM Network e. V. 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, dass Ihr schon wieder so lange ohne Murren auf euren Papa verzichtet habt.

Grundlagen des Software-Entwurfs

Für den Entwurf objektorientierter Systeme 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] und [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] nur auf objektorientierte Systeme.

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

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.

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 sind.

Prinzip der einzelnen Verantwortlichkeit

(Single-Responsibility Principle, Separation-of-Concerns) Jede Klasse sollte eine strikt abgegrenzte Verantwortlichkeit besitzen. 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 (57) oder →PlugIn (106)) unterstützen dieses Prinzip.

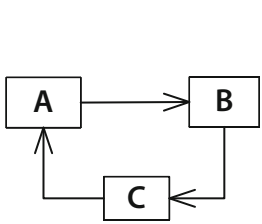
Prinzip der gemeinsamen Wiederverwendung

(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 ver-

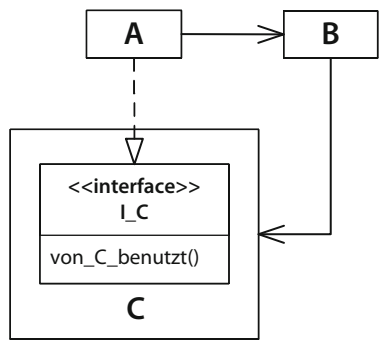
wendet werden, sollten in ein gemeinsames Paket verpackt werden. Dies hilft, zirkuläre Abhängigkeiten zwischen Paketen zu vermeiden.

Keine zirkulären Abhängigkeiten

(Acyclic Dependency Principle) Klassen und Pakete sollten keine zirkulären (zyklischen) Abhängigkeiten enthalten. Solche Zyklen sollten unter Software-Architekten 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.



Zirkuläre Abhängigkeit



Aufgelöster Zirkel

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.

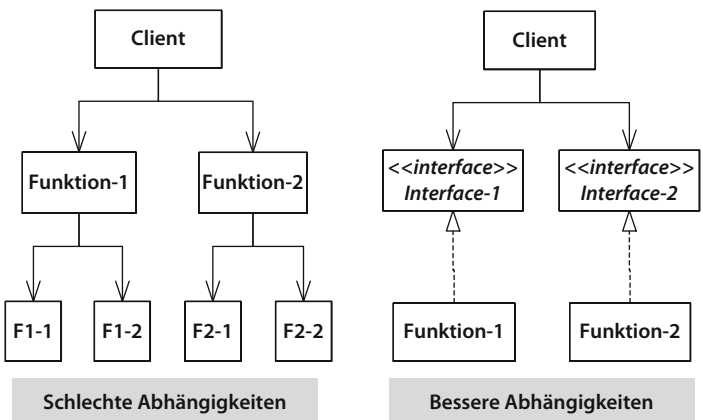
Liskov'sches Substitutionsprinzip

(Liskov Substitution Principle) Unterklassen sollen anstelle ihrer Oberklassen einsetzbar sein. Sie sollten beispielsweise in Unterklassen niemals 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.

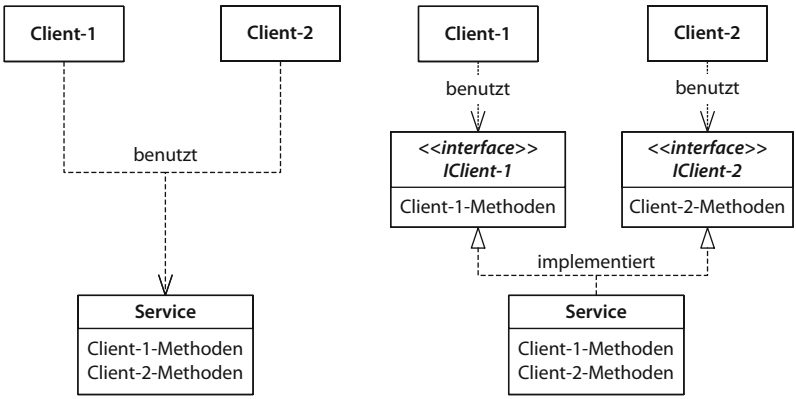
Prinzip der Umkehrung von Abhängigkeiten

(Dependency Inversion Principle) Nutzer einer Dienstleistung sollten möglichst von Abstraktionen (d. h. abstrakten Klassen oder Interfaces), nicht aber von konkreten Implementierungen abhängig sein. Abstraktionen sollten nicht von konkreten Implementierungen abhängen.

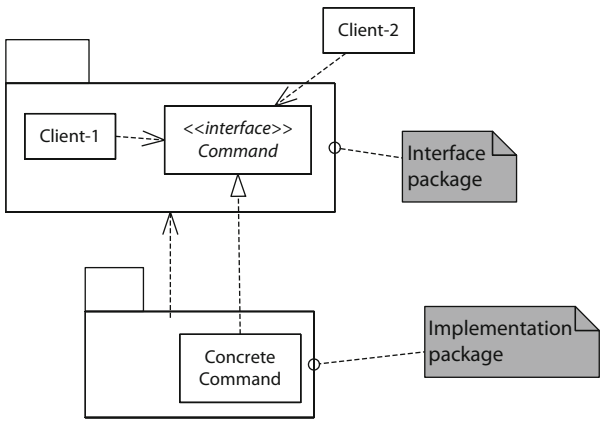


Prinzip der Abtrennung von Schnittstellen

(Interface Segregation Principle) Clients sollten nicht von Diensten abhängen, die sie nicht benötigen. 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. Die Schnittstellen liegen dabei von ihren Implementierungen getrennt in eigenen Paketen. Zur Laufzeit müssen Sie eine konkrete Implementierung auswählen. Dafür bieten sich →Erzeugungsmuster (25) oder auch →PlugIn (106) an.



Separated Interface (nach [Fowler])

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 stillschweigend davon ausgehen können, dass Sie vorhandene Strukturen wie einen LDAP für Ihre Applikation mitbenutzen dürfen. Besonders unangenehm können Annahmen auch auf der Implementierungsebene sein. Vor einiger Zeit 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 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 dem Verwender eines Frameworks oder einer Software-Komponente die Konfiguration durch Konventionen und sinnvolle Voreinstellungen erleichtert.

In der Software-Entwicklung sind, bedingt durch den Einsatz komplexer Frameworks, sehr viele Einstellungen konfigurierbar, heute meist über XML-Dateien. Der Höhepunkt war wohl mit EJB 2.1 erreicht. 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 Prozent aller Fälle immer gleich ist, einem einfachen Schema folgt oder schlimmstenfalls für meine Anwendung völlig irrelevant ist?

Moderne Frameworks wie Spring (http://www.springsource.org/) drehen den Spieß um. Sie führen für Framework-Einstellungen oder auch das Mapping von Entitäten (s. a. →O/R-Mapping (116)) strikte Konventionen und Standardwerte ein. Hält sich der Framework-Verwender an die Konventionen, muss er 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 Konventionen 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 dazu finden Sie unter
http://softwareengineering.vazexqi.com/files/pattern.html.

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. Falls Ihnen dies zu abstrakt ist: Das →Bridge-Pattern (67) wendet diese Heuristik an.
- Vermeiden Sie übermäßig mächtige Klassen („Poltergeister“ oder „Gott-Klassen“).
- Benutzer einer Klasse dürfen ausschließlich von deren öffentlichen Schnittstellen abhängen. Klassen sollten nicht von ihren Benutzern abhängig sein.
- 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 (70) oder dem State-Pattern (s. [GoF]) zu lösen. Das .NET-Framework bietet zudem die Möglichkeit der *Extension-Methods* ([Troelson]).
- Hüten Sie sich davor, Objekte einer Klasse als Unterklassen zu modellieren. In der Regel sollte es von (abgeleiteten) Klassen mehr als eine einzige Instanz geben können.
- Wenn Klassen besonders viele `getX()`-Methoden in der öffentlichen Schnittstelle enthalten, haben Sie möglicherweise die Zusammengehörigkeit von Daten und Verhalten verletzt.
- Halten Sie sich bei der Modellierung von Klassen möglichst nah an die reale Welt. Streben Sie bei Analyse- und Design-Modellen nach strukturellen Ähnlichkeiten.
- Vermeiden Sie überlange Argumentlisten. Darunter leidet die Übersichtlichkeit von Methodenaufrufen. Verschieben Sie die Methode in eine andere Klasse oder übergeben Sie höher aggregierte Objekte als Argumente.

Beziehungen zwischen Klassen und Objekten

- Minimieren Sie die Abhängigkeiten einer Klasse (*Minimal Coupling*).
- Viele Methoden einer Klasse sollten viele Attribute dieser Klasse häufig benutzen. Anders formuliert: Wenn viele Methoden nur mit wenigen Attributen arbeiten, dann haben Sie möglicherweise die Zusammengehörigkeit von Daten und Verhalten verletzt.
- Eine Klasse sollte nicht wissen, worin sie enthalten ist.
- Regel von Demeter (Law of Demeter, LoD): „Reden Sie nicht mit Fremden“, d. h., ein Objekt sollte nur sich selbst, seine Attribute oder die Argumente seiner Methoden referenzieren. Kennen Sie das Spiel Halma? Man springt möglichst weite Pfade über viele Spielsteine hinweg, um schnell ans Ziel zu kommen. Diese *transitive Navigation* sollten Sie bei der Software-Entwicklung tunlichst vermeiden ([Martin2]). Aufrufe wie `outputHandler.getCurrent-`

`Destination().getFile().getSize()` zeigen einen Verstoß gegen LoD und deuten auf ein schlechtes oder unvollständiges Interface hin. Im Beispiel könnten Sie das Design verbessern, indem Sie die Klasse `OutputHandler` um eine Methode `getBytesWritten()` erweitern.

Vererbung und Delegation

- Verwenden Sie Vererbung nur zur Modellierung von Spezialisierungen.
- Keine Oberklasse sollte etwas über ihre Unterklassen wissen.
- Abstrakte Klassen sollten Basisklassen ihrer Hierarchie sein bzw. nur von abstrakten Klassen ableiten.
- Falls mehrere Klassen A und B nur gemeinsame Daten besitzen (aber kein gemeinsames Verhalten), dann sollten diese gemeinsamen Daten in einer eigenen Klasse sein, die in A und B enthalten ist.
- Vermeiden Sie es, den Typ einer Klasse explizit zu untersuchen. Verwenden Sie stattdessen Polymorphismus.
- Überschreiben Sie niemals eine Methode einer Oberklasse mit einer leeren Implementierung (das führt zu einer Verletzung des Liskov'schen Substitutionsprinzips).
- Vermeiden Sie Mehrfachvererbung. Genauer: Vermeiden Sie mehrfache Implementierungsvererbung. Mehrfache Schnittstellenvererbung hingegen ist erlaubt.
- Bevorzugen Sie, wenn möglich, Schnittstellen (*Interfaces*) gegenüber abstrakten Klassen.

Verteilung

- Martin Fowlers Heuristik zur verteilten Datenverarbeitung: Vermeiden Sie Verteilung, wo immer möglich. Die Welt ist auch ohne Verteilung komplex genug.
- Angemessen eingesetzt kann Verteilung die Flexibilität oder Skalierbarkeit von Systemen verbessern.

Nebenläufigkeit

(Concurrency) Moderne Mehrkernprozessoren lassen sich mit einem einzelnen Thread nicht auslasten. Schon deshalb lohnt sich die Beschäftigung mit dem Thema. Leider ist und bleibt Multithreading kompliziert. Es ist ein bisschen wie mit Zeitreisen: Solange Sie