1 Einführung

1.1 Was ist guter Design

Hier meine Lieblingsdefinition aus einem Tutoriel über Extreme Programming .

The right design for software is one that

1. Runs all the tests
2. Has no duplicated logic.
3. States every intention important to the programmers
4. Has the fewest possible classes and methods

In der obigen Definition ist die Reihenfolge der Anforderungen entscheident. Ein Wesen von Extreme Programming ist, daß es keine Softwareanforderung gibt, die nicht durch Tests abgedeckt wird.

NOT

- Most hooks
- Most abstract
- Designed for ages

1.2 Was ist Refactoring

Durch Refactoring wird die interne Struktur von Software, aber nicht das externe Verhalten geändert. Das Ziel von Refactoring ist es den Design von Software zu verbessern. Dadurch verspricht man sich leichtere Wartung und Weiterentwicklung, sowie ein höherer Grad an Wiederverwendbarkeit

2 Refactoring

2.1 Klassen

2.1.1 Ein oder zwei Klassen

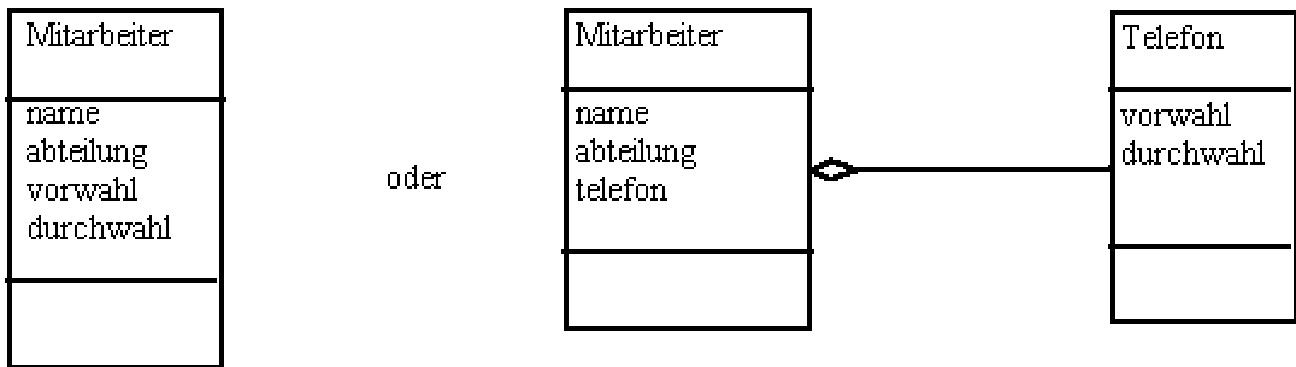


Abb. 2.1: Ein oder 2 Klassen?

- Vermeide faule Klassen (besitzt nur Accessor Methode)
- Gibt es Methoden (ausser Accessor-Methoden), die sich nur sich auf eine Teilmenge der Instanzvariablen beziehen, lohnt es sich eine neue Klasse zu extrahieren.
- Vermeide doppelten Code. Besitzt nicht nur Mitarbeiter, sondern auch andere Klassen ein Telephon, so sollte das Telephon in einer eigener Klasse gespeichert werden (Ausnahme die Klasse ist faul)

Refactoring . Extract Class, Inline Class

2.1.2 Vorteile kleiner Klassen

- leichter zu testen
- höhere Wiederverwendbarkeit
- Versionierung: niedrigere Wahrscheinlichkeit, daß 2 Programmierer an selben Klasse arbeiten

2.1.3 Extract Subclass

Was ist bei diesem Design problematisch?

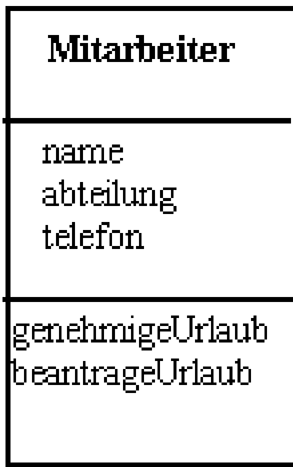


Abb. 2.2: Hier kann Extract Subclass verwendet werden

<u>Lösung</u>

Nicht alle Mitarbeiter müssen Urlaub beantragen, oder können Urlaub genehmigen.Wenn einzelne Methoden nur für eine Teilmenge aller Instanzen sinnvoll sind, so deudet dies auf Extract Subclass hin.

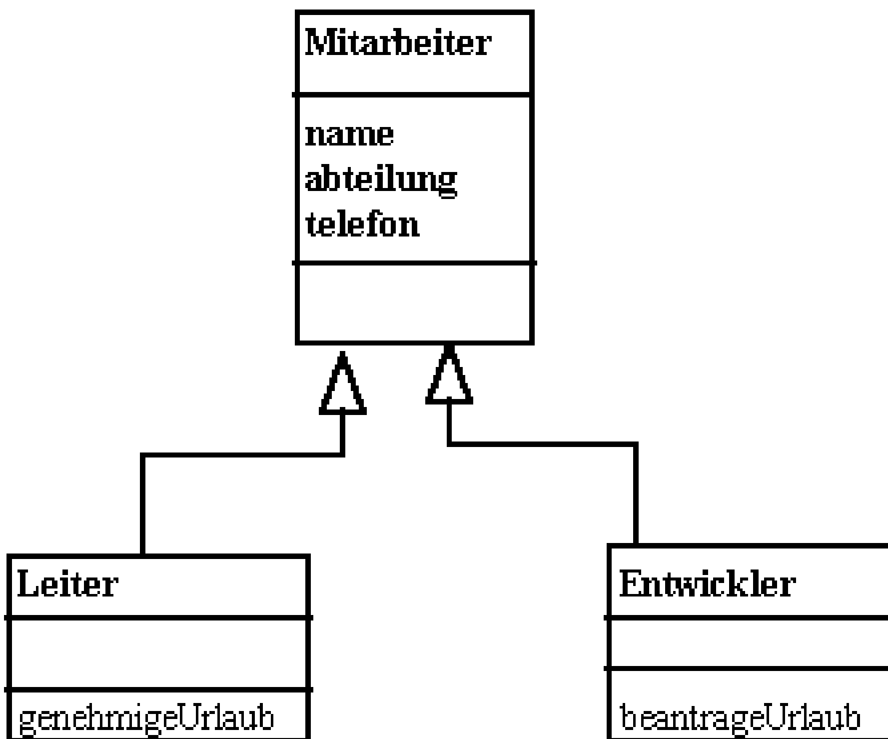


Abb. 2.3: Beispiel von Extract Subclass

2.1.4 Rollen Status Pattern

Das obige Objektmodell ist immer noch nicht flexibel genug. Ein Entwickler kann zum leitenden Mitarbeiter befördert werden, d.h plötzlich verändert sich die Verantwortlichkeiten des Mitarbeiters.

Das Verhalten einer Klasse ist i.a. konstant (ausser man verwendet fiese Tricks).Ein Objekt kann auch nie (Auch hier gibt es unrühmliche Ausnahmen) seine Klasse ändern. In der realen Wert ändern sich aber das Verhalten von Objekten.Eine Lösung zur Modellierung ist das Status-Pattern. Dies wird immer dann verwendet wenn das Verhalten eines Objekts vom Status abhängig ist.

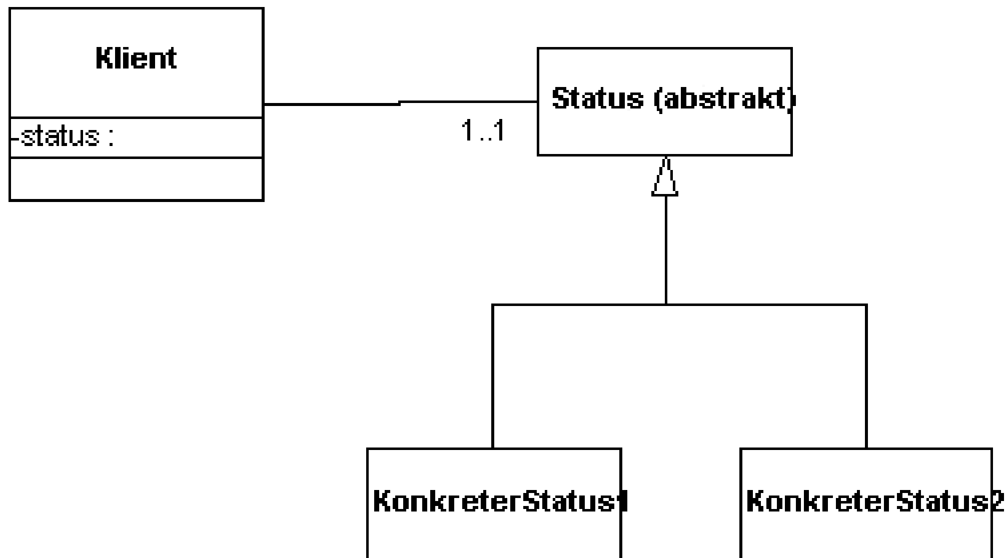


Abb. 2.4: Status Pattern

Das Status-Pattern lohnt sich nur, falls ein Objekt seinen Status ändert, ansonsten verwende einfache Vererbung. Das Rollenpattern ist eine spezielle Form des Status-Patterns:

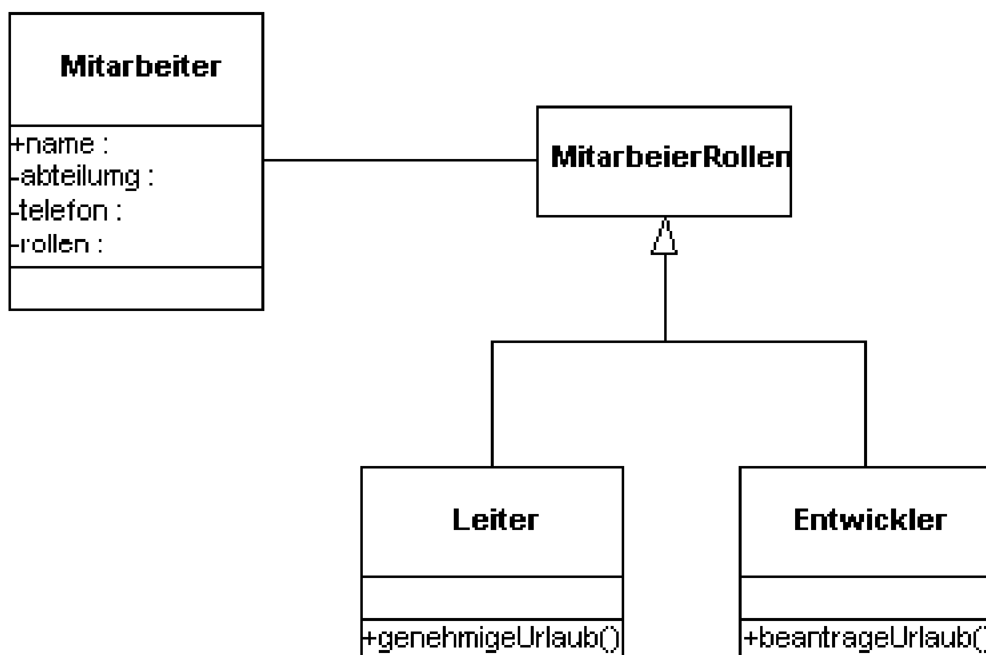
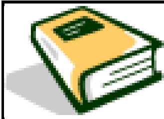


Abb. 2.5: Rollen Pattern



Refactoring Entwurfsmuster Plop Buch Band 3, Kapitel 9. Rollen PDF-File von Martin Fowler.

2.2 Methoden

2.2.1 Extract Method

2.2.1.1 Einführung

```
void myMethod(){
  if (widgetId == #F0F898){   "ist Widget sichtbar"
    tueEtwas();
}
```

daraus wird:

```
boolean isWidgetVisible(){
  return widgetId == #F0F898
}
void myMethod(){
  if (isWidgetVisisble){
     tueEtwas();
  }
}
```

Was ist passiert?

- Die Methode *myMethod* wurde extrahiert. Eine komplexe Methode wurde in einfachere Methoden aufgeteilt.
- Ein Kommentar wurde in einfachere Methoden mit einem aussagekräftigen Namen ersetzt

Do not comment bad code, rewrite it

2.2.1.2 Idealer Design von Methoden

- Eine Methode sollte jeweils nur eine Aufgabe erfüllen
- Die Methoden sollte kurz sein
- Komplexerer Methoden rufen Methoden der naechst niedrigen Abstraktionsstufe auf.
- Methoden der obersten Abstraktionsstufe sind oeffentlich.
- Methoden der untersten Abstraktionsstufe sind Accessor-Methoden
- Wichtig sind aussagekräftige Namen

2.2.1.3 Vorteile kurzer Methoden

- Hoehere Wahrscheinlichkeit der Wiederverwendung
- Leichter zu warten
- effektivere Coverageanalyse

Bemerkung: Ein Coverageanalyse-Tool ist ein unentbehrliches Hilfsmittel für Tests. Es stellt fest, welche Klassen bzw. Methoden, wie oft und ggf. mit welchen Parametern aufgerufen worden sind. So kann man feststellen, welcher Bereich des Source Codes noch nicht ausreichend getestet worden ist. Einfache Coverageanalyse - Tools erkennen nicht welche Verzweigungen (z.B IFThenElse Block) innerhalb einer Methode aufgerufen worden ist. Deshalb verwende kleine Methoden, ohne groessere Verzweigungen

Reusability through self encapsulation Mein Lieblingsartikel aus dem Plop1 Buch von Ken Auer. Es geht um grundsaetzliche Vorgehensweise beim Design einer Klasse

Extrahiere Methode ohne lokale Variable

```
void printOwing(){

Enumeration e = _orders.elements();
    double outstanding = 0.0 ;

// print banner

System.out.println ("**************************");
    System.out.println ("***** Customer Owes *****");
    System.out.println ("**************************");

// calculate outstanding

while (e.hasMoreElements()){
           Order each = (Order) e.nextElement();
           outstanding += each.getAccount();
    }

// printDetails
    System.out.println("name:" + _name);
    System.out.println("amount:" + outstanding);
}
```

Zuerst wird printBanner extrahiert, da in diesem Teil keine lokalen Variablen sind, ist diese Aktion besonders trivial.

```
void printOwing(){

Enumeration e = _orders.elements();
    double outstanding = 0.0 ;

printBanner();

// calculate outstanding

while (e.hasMoreElements()){
           Order each = (Order) e.nextElement();
           outstanding += each.getAccount();
    }
```

```
// printDetails
    System.out.println("name:" + _name);
    System.out.println("amount:" + outstanding);
}
```

```
void printBanner(){
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes *****");
    System.out.println ("**************************");
}
```

Extrahiere Methode mit lokale Variablen

Jetzt wird printDetails extrahiert. Dabei wird eine lokale Variable outstanding benutzt, die als Parameter übergeben wird

```
void printOwing(){

Enumeration e = _orders.elements();
    double outstanding = 0.0 ;

printBanner();

// calculate outstanding

while (e.hasMoreElements()){
           Order each = (Order) e.nextElement();
           outstanding += each.getAccount();
    }

printDetails(outstanding)
}
```

```
void printBanner(){
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes *****");
    System.out.println ("**************************");
}
```

```
void printDetails( double outstanding ){
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Jetzt wird calculate outstanding extrahiert. Die lokale Variable outstanding kann in extrahierten Teil definiert werden. Da sie von printDetails noch gebraucht wird, muss sie als returnWert zurückgegeben werden

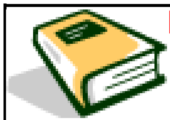
```
void printOwing(){
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding)
}
```

```
void printBanner(){
```

```
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes *****");
    System.out.println ("**************************");
}
```

```
void printDetails( double outstanding ){
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

```
double getOutstanding(){
    Enumeration e = _orders.elements();
    double result = 0.0;
    while (e.hasMoreElements()){
      Order each = (Order) e.nextElement();
      result += each.getAmount();
    }
    return result ;
}
```



Refactoring Extract Method

2.2.1.4 Typisierung von Methoden

Eine einheitliche Benennung der Methoden vereinfacht die Lesbarkeit des Codes. Dirk Riehle hat im JavaReport eine Typsierung von Methoden , je nach Verwendungszweck, sowie eine Einteilung nach Methoden Eigenschaften vorgeschlagen. Jeder Methodentyp wird durch ein bestimmtes Prefix im Namen gekenndzeichnet. Hier ein Link zum Original .

2.2.1.5 Method Object

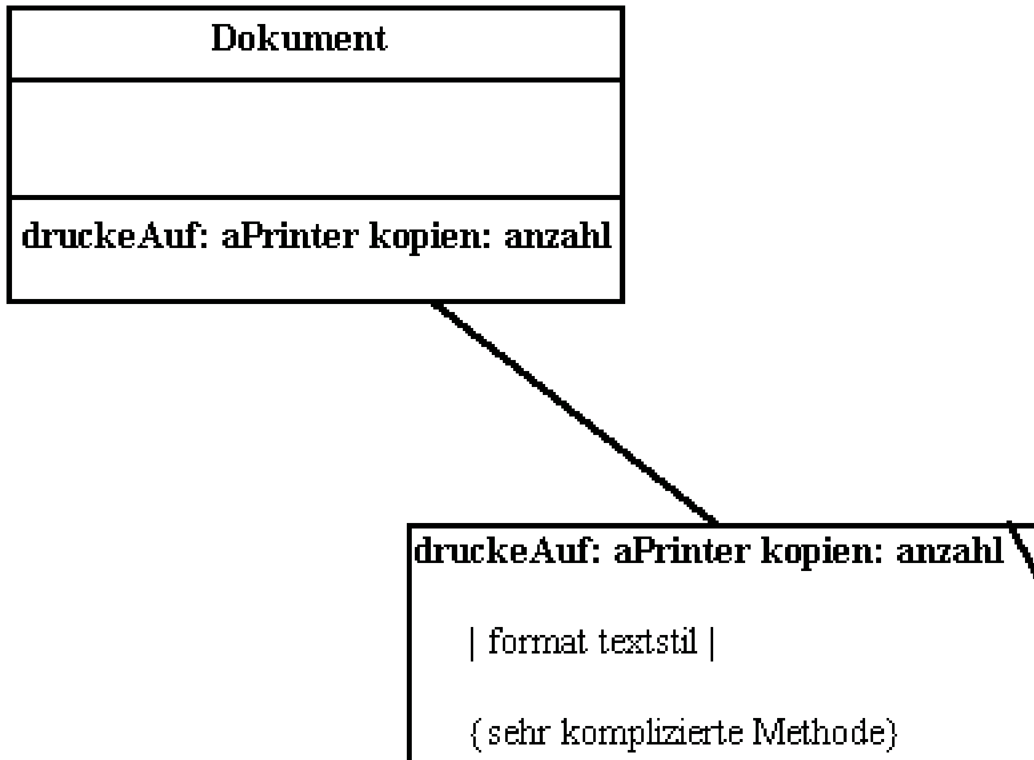


Abb. 2.6: Eine komplizierte Methode

Einige Methoden lassen sich trotz grosser Anstrengungen nicht extrahieren. Dies liegt meist daran, daß sehr viele lokale Variablen verwendet werden, und innerhalb der Methode verschachtelte Kontrollstrukturen und Returns sind. In diesen Fällen hilft es, die Methode durch eine Klasse zu ersetzen. Da in dieser Klasse alle lokalen Variabeln der komplexen Methode als Instanzvariablen umgewandelt sind, lässt sich die Klasse leicht refaktorisieren.

1. Erzeuge eine neue Klassen, mit dem Name der sich aus der Methode ergibt (Drucken).
2. Erzeuge für jeden Parameter (printer,kopien) der Methode und für jede lokale Variable (textstil, format) ein Attribut.
3. Erzeuge ein Attribut für das ursprüngliche Objekt (Dokument).
4. Erzeuge für jedes Attribut des neuen Objekts die Accessor-Methoden.
5. Verschiebe den Text der alten Methode in eine neue Methode compute des neuen Objekts.
6. Ersetze in compute die Parameter und die lokalen Variabeln durch die Accessor-Methoden.
7. Ersetze in compute die Referenzen auf das Ursprungsobjekt (i.a. this) durch die entsprechende Accessor-Methoden (getDokument).
8. Gebe der neue Klasse einen Konstruktor mit folgenden Parametern.
   1. Parameter der alten Methode (printer kopien);
   2. Parameter für das ursprüngliche Objekt (Dokument)
9. Ersetze im ursprünglichen Objekt die komplexe Methode durch den Aufruf der neuen Klasse und rufe die Methode compute auf.

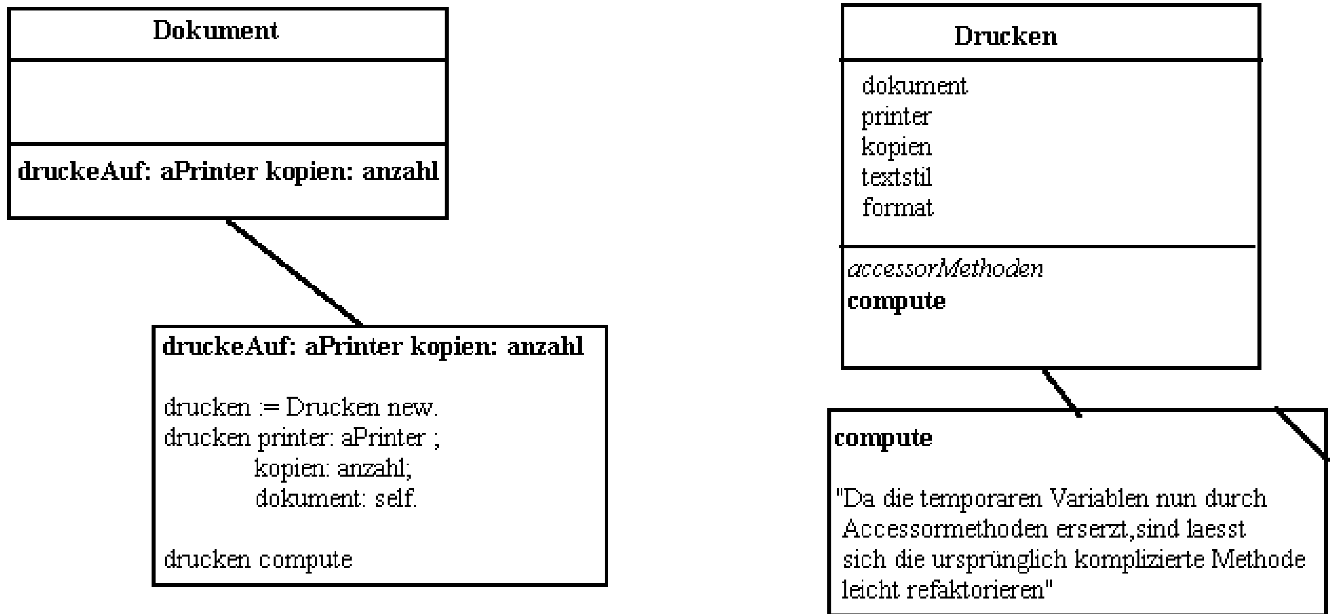
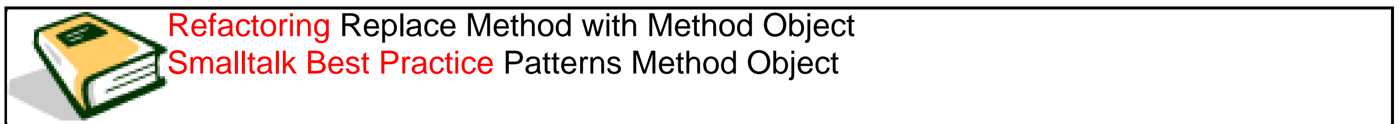


Abb. 2.7: Methode in Klasse umgewandelt



2.3 Vererbung Delegation

2.3.1 Verschiebung Methoden, Attribute

Ein einfaches Beispiel zur Einführung:

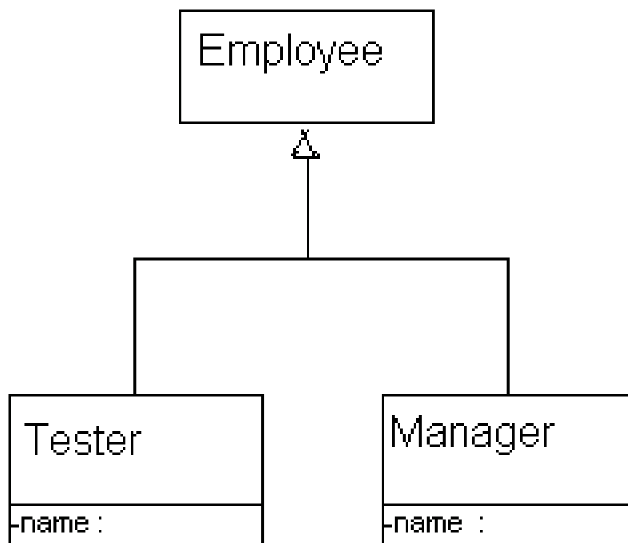


Abb. 2.8: PullUp Field

Diskussion: Was passiert falls es noch eine Unterklasse von Employee gibt, ohne dem Attribut Name ?

2.3.2 Template Methode

Die Methoden in der Unterklassen haben die gleiche Struktur, sind aber nicht identisch.

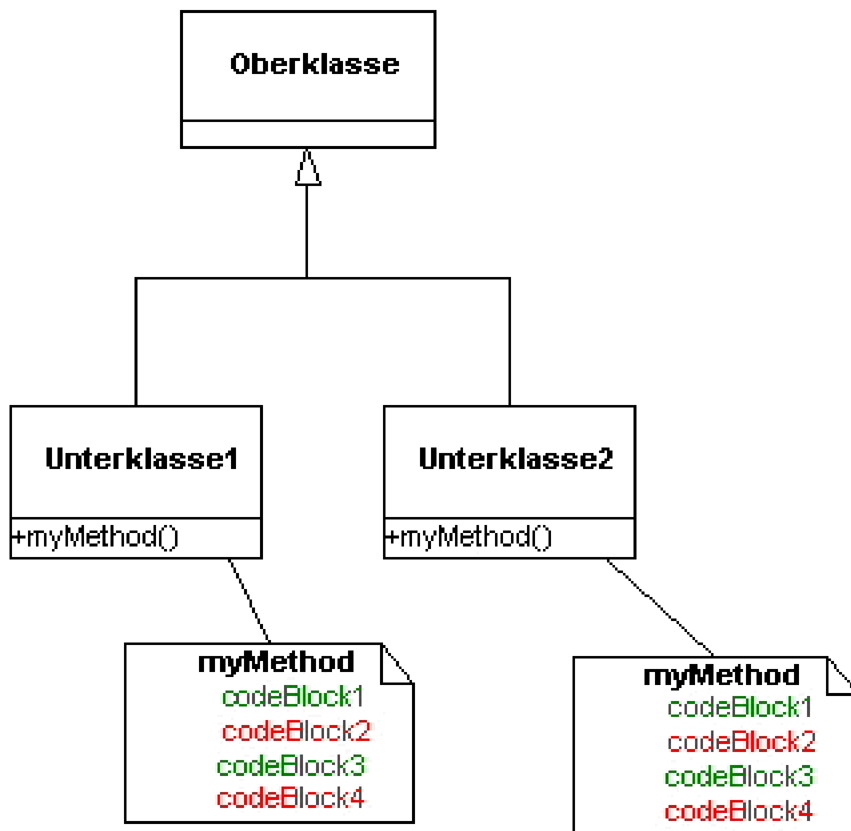


Abb. 2.9: Template Methode

codeBlock1 und codeBlock3 sind jeweils identisch, codeBlock2 und codeBlock4 sind unterschiedlich.

1. Extrahiere die Methoden in den Unterklassen. Gebe den extrahierten Methoden dieselben Namen.

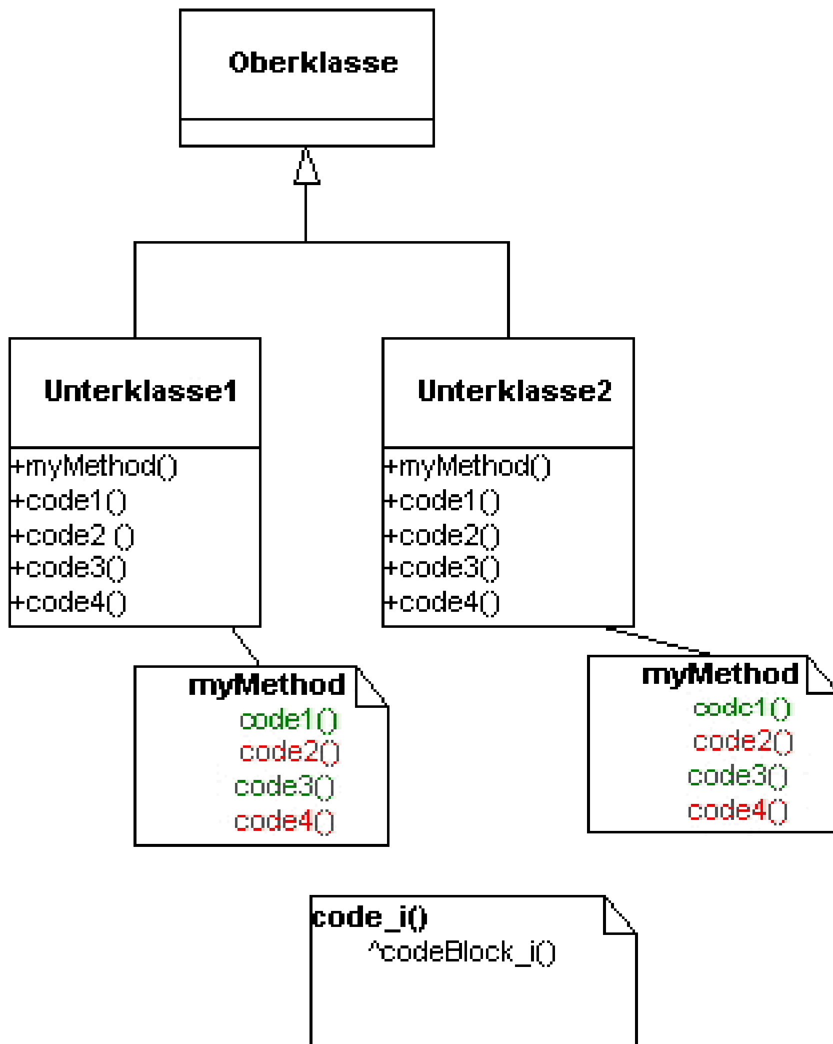


Abb. 2.10: Template Methode

2. Schiebe die identischen Methoden der Unterklasse, in die Oberklasse.

3. Erzeuge in den Oberklasse für die verschiedenen Blöcke abstrakte Klassen

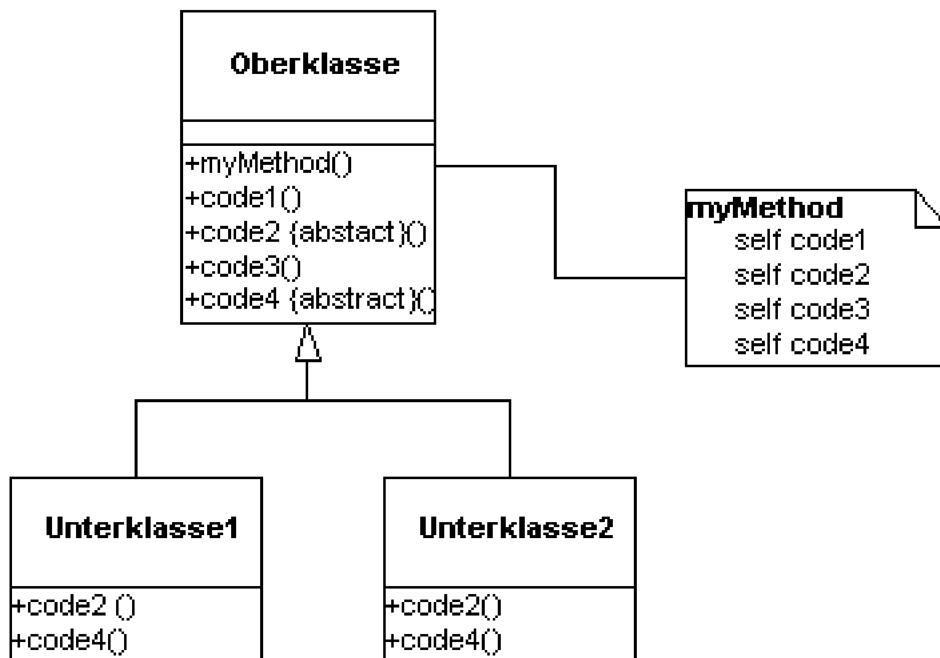
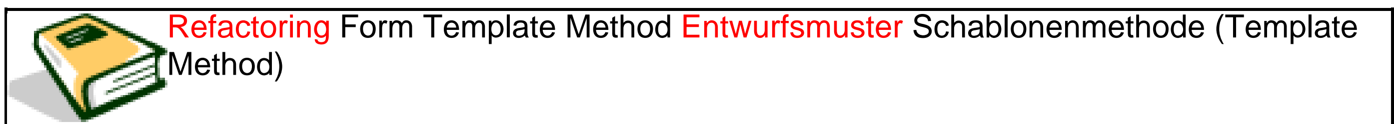


Abb. 2.11: Ergebnis Template Methode

Der eigentliche Algorithmus steht in der Oberklasse. Die Methoden code2() bzw. code4() nennt man hook-up Methoden da sie sich in den Algorithmus einklingen. Die Methode myMethod in der Oberklasse ist eine template-Methode.


Refactoring Form Template Method Entwurfsmuster Schablonenmethode (Template Method)

2.3.3 Vererbung und Delegation

Probleme mit Vererbung:

- Widerspricht der Kapselung: Alle Änderung an der Oberklasse wirkt sich auf die Unterklassen aus.
- Klassenbeziehungen sind statisch, d.h sie können nicht zur Laufzeit geändert werden.

Hier ein völlig verunglücktes Objektmodell:

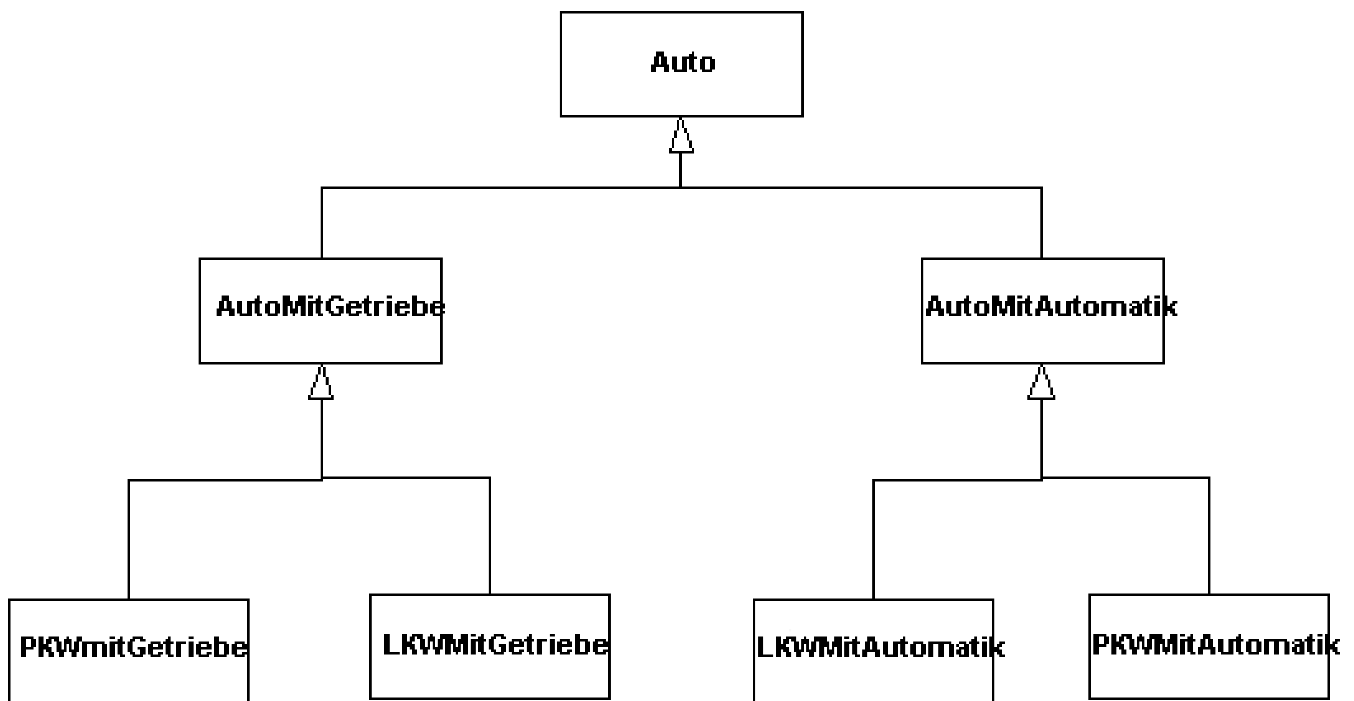


Abb. 2.12: Probleme mit paralleler Vererbungshierarchie

Dies ist ein abschreckendes Beispiel von paralleler Vererbungshierarchie. Eine Änderung an LKW muss in 2 verschiedenen Klassen nachvollzogen werden. Mögliche Lösungen:

- Mehrfachvererbung: In Smalltalk und Java nicht möglich. Diese Lösung ist auch schlecht wartbar
- Ersetze Vererbung durch Delegation

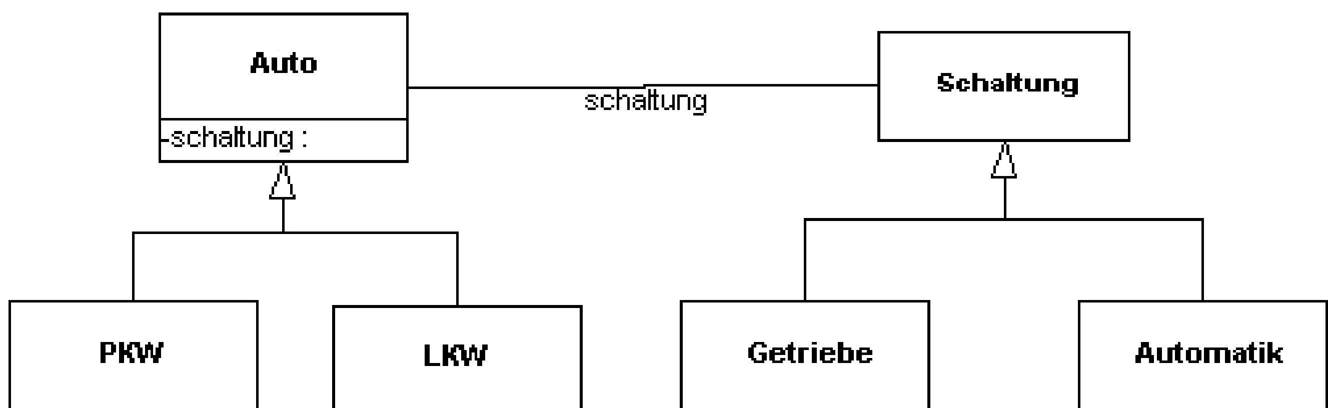


Abb. 2.13: Delegation statt Vererbung

In der obigen Lösung kann im Auto die Gangschaltung ausgetauscht werden.

Die Vererbung ist zwar ein typisches Kennzeichen von OO, man sollte sie aber vorsichtig einsetzen. Bevorzuge kleine Vererbungshierarchien

2.4 Vereinfache Bedingungen

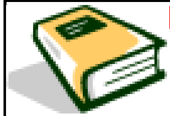
2.4.1 Ersetzte verschachtelte Bedingungen durch Guard Clauses

Wie kann ich folgende Methode vereinfachen:

```
double getPayAmount(){
   double result ;
   if (_isDead) result = deadAmount();
   else {
        if (_isSeparated) result = separatedAmount();
        else {
             if (_isRetired) result = retiredamount();
             else result = normalPayAmount();
        };
   }
   return result;
 };
```

Hier die einfache Lösung:

```
double getPayAmount(){
  if (_isDead) result deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```



Refactoring Replace Nested Conditional with Guard Clauses

2.4.2 Ersetzte Bedingungen durch Polymorhpismus

In vielen Fällen sind verschachtelte Bedingungen ein Zeichen für fehlende Klassen. Siehe bitte folgendes Beispiel:

```
class Employee ...
    private int _type;
    static final int ENGINEER = 0 ;
    static final int SALESMAN = 1 ;
    static final int MANAGER = 2 ;
```

```
...
```

```
int payAmount(){
       switch (_type){
          case ENGINEER:
             return _monthlySalary ;
          case SALESMAN:
             return _monthlySalary + _commission ;
          case MANAGER:
             return _monthlySalary + _bonus
          default:
             throw new RuntimeException("Incorrect Employee");
        }
    }
```

In relationalen Datenbanken ist es in Ordnung eine Tabelle Employee mit einer Spalte Type zu definieren. In rel. Datenbanken wird nicht das Verhalten sondern nur die Struktur eines Objekts gespeichert. Klassen sollten aufgrund des Verhaltens (Methoden), und nicht aufgrund ihrer Struktur (Attribute) entworfen werden. Es lohnt sich die Klassen ENGINEER, SALESMAN bzw. MANAGER anzulegen. Dabei gibt es 2 Möglichkeiten.

- Die neuen Klassen sind Unterklassen der (jetzt abstrakten) Klasse Employee.
- Verwendung des Rollenpatterns

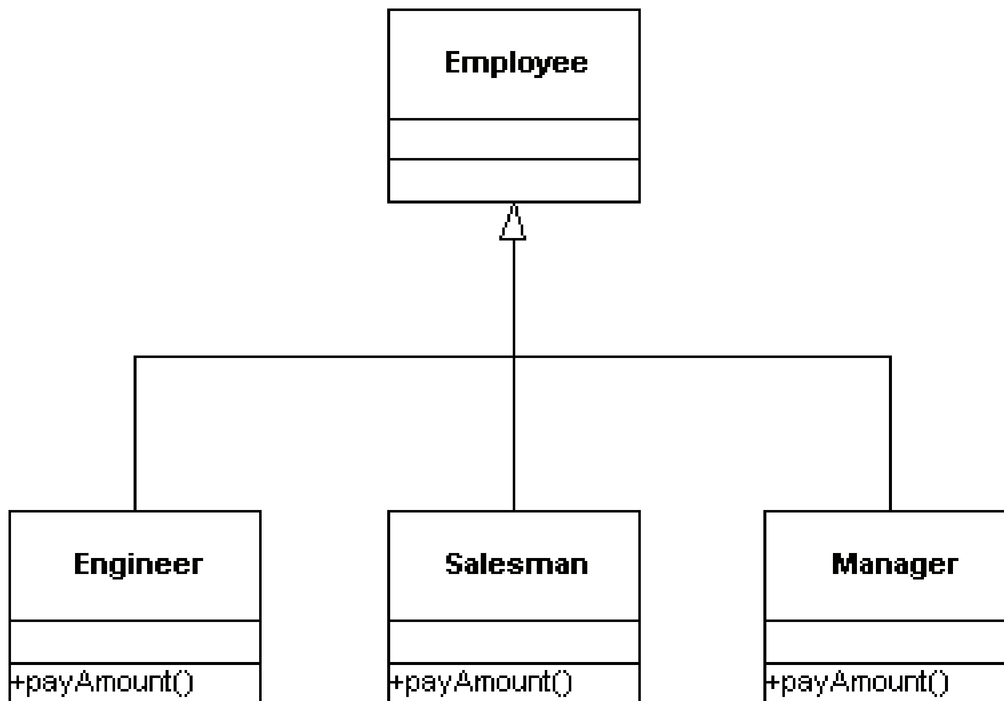


Abb. 2.14: Verwendung von Polymorphismus

Refactoring Replace Type Code with State/Strategy, Replace Type Code with Subclasses.

2.4.3 Führe Null Objekt ein

Viele Bedingungen überprüfen, ob ein Empfänger einer Methode überhaupt vorhanden ist.

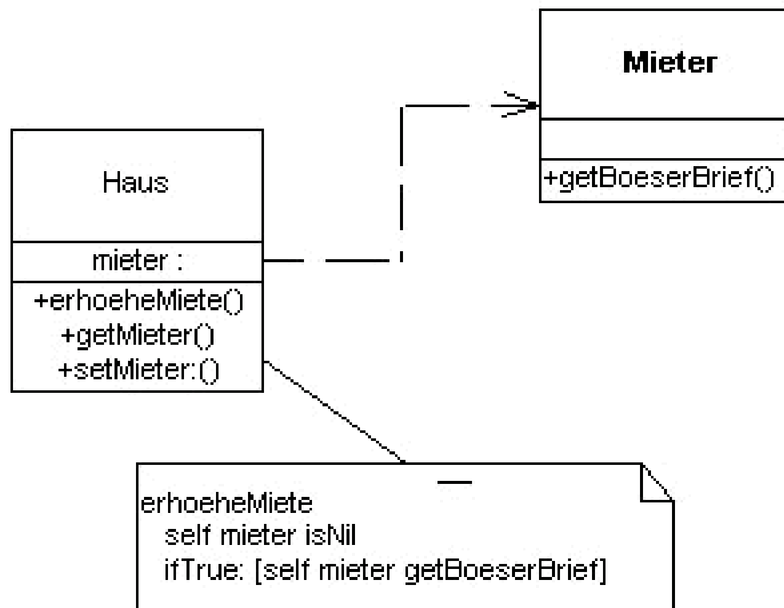


Abb. 2.15: Beispiel ohne Nullobjekt

Die Einführung von NullObjekten ermöglicht es auf andauernde Fragen ob ein Attribut gesetzt sind zu verzichten. Das NullObjekt hat i.a. genau die gleichen Sclhnittstelle wie das entsrechende "reales" Objekt. Allerdings laufen die Methoden i.a. ins Leere oder geben ein NullObjekt einer anderen Klasse zurück. Die untere Lösung zeigt auch das Prinzip der Lazy Initialisation. Der Wert des Attributs mieter in der Klasse Haus, kann nie nil werden. Es ist immer garantiert, daß ein (Null) Mieter gesetzt ist. Alle Methoden der Klasse Haus, die sich auf den Mieter beziehen, können auch ausgeführt werden.

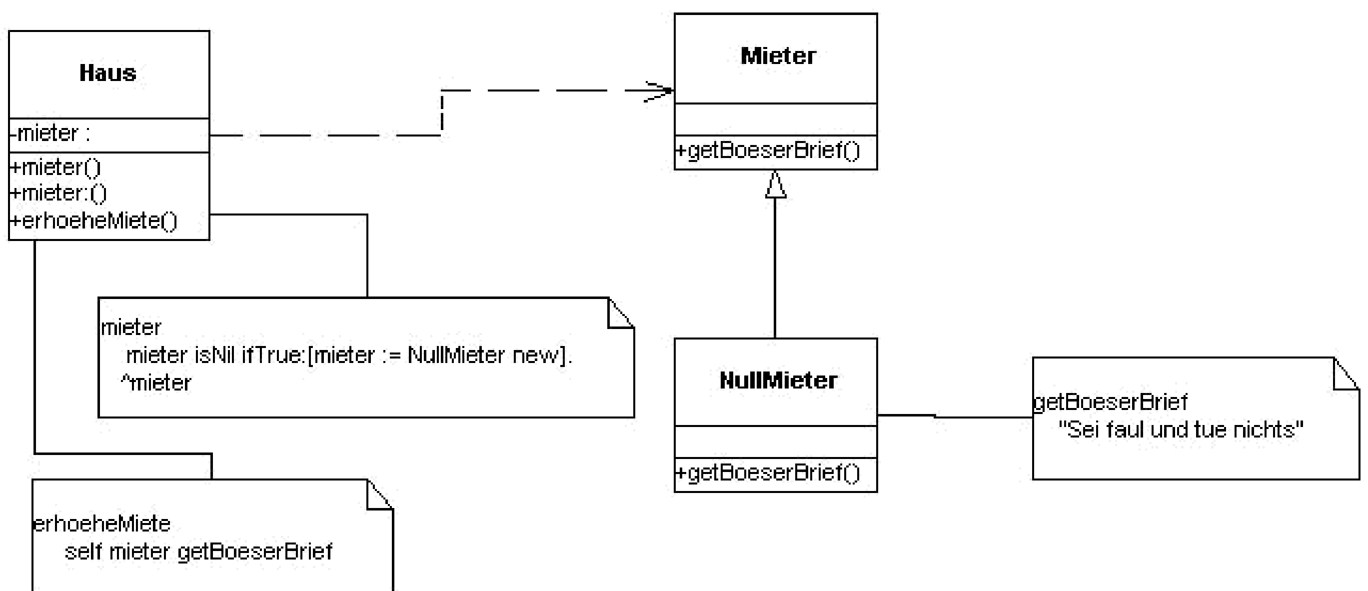
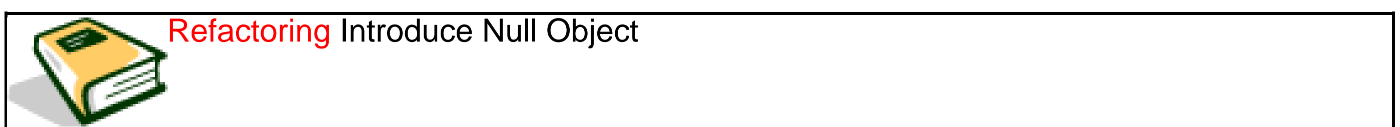


Abb. 2.16: Beispiel mit Nullobjekt



2.5 Schnittstellen

2.5.1 Bedeutung von Schnittstellen

- Schnittstellen spezifizieren Objekte vollständig.
- Änderungen innerhalb des Objekt, wirken sich nicht ausserhalb aus.
- Änderungen an Schnittstellen betrifft alle Clienten
- Schnittstellen legen fest welche Arbeit das Objekt bzw. die Clienten haben

2.5.2 Folgerungen

- Achte auf qualitative Schnittstellen
- halte die Anzahl der möglichen Clienten klein (Schichtenmodell, Fassade)
- veröffenliche Schnittstellen spät

2.5.3 Qualitätskriterien von Schnittstellen

- vollständig
- schlank
- einheitlich
- dokumentiert

2.5.4 Adapter Pattern

Ein Adapter verbindet 2 Objekte, deren Schnittstellen nicht zusammenpassen.

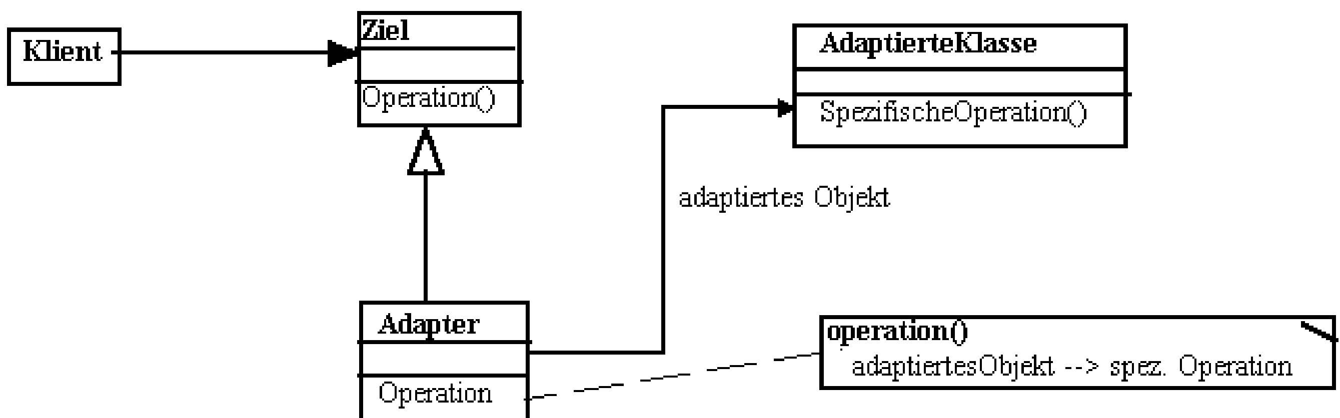
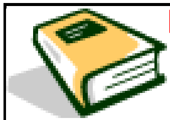


Abb. 2.17: Adapter Pattern

Ein Objektadapter verwendet eine Objektkomposition . Er besteht im Kern aus zwei Objekten: dem Adapterobjekt und dem adaptierten Objekt. Die Klasse Adapter erbt die geforderte Schnittstelle von der Klasse Ziel. Sie implementiert diese Schnittstelle mit Hilfe von AdaptierteKlasse. Klienten verwenden Adapter nur über eine abstrakte Zielklasse. Ruft ein Klient über Polymorphie eine Operationen des Adapterobjekts auf, dann ruft das Adapterobjekt die Operationen des adaptierten Objekts, welche die gewünschte Dienst-leistung liefern.
Entwurfsmuster Adapter

2.5.5 Lange Parameterlisten

- Falls die Methode mehrere Dinge auf einmal macht, ersetze sie durch mehrere Methoden mit jeweils weniger Paramter.
- Ueberpruefe ob ein Parameter durch eine Methode ersetzt werden kann.
- Falls mehrere Parameter vom selben Objekt kommen, so übergebe anstelle der Parameter gleich das ganze Objekt.
- Führe Paramterobjekt ein.


Refactoring Kapitel 10: Making Method Calls Simpler

2.6 Behälter, Law of Demeter

2.6.1 Behälter

Arrays werden i.a. zur Sammlung von Daten verwendet. Die Elemente eines Arrays sollten aber immer gleichartig sein. Konventionen, wie im 1. Element des Arrays ist der Vorname enthalten, lassen sich schwer merken. Wandle ein Array mit verschiedenartigen Elementen in ein Objekt um.

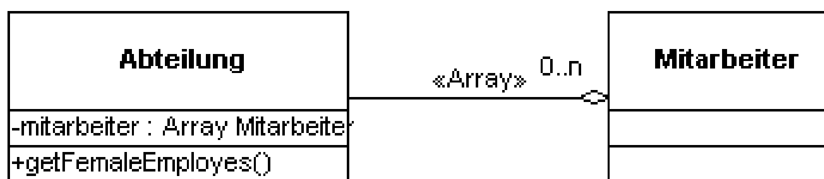


Abb. 2.18: Verwendung von Behaelter

Gibt es Operationen über ein Array (wie z.B: Sammle alle weiblichen Mitarbeiter), so lohnt es sich einen Behaelter für dieses Array einzuführen.

2.6.2 Law of Demeter

Oft sieht man lange Methodenketten wie

```
self myMethod().method1().method2().method3().method4().tueEtwas()
```

Diese Aufrufe sind problematisch, denn wenn einer dieser Methoden etwas anderes als erwartet zurückgibt, wird ein Fehler ausgelöst. Das Law of Demeter verhindert solche Message Chains. Eine kurze Zusammenfassung dieses Gesetzes lautet:

Don't talk to strangers (Rede nie mit Fremden)

Sende Nachrichten nur an:

```
class Demeter {
private:
   A *a;
  int func();
public:
  // ...
  void example(B& b);
}

void Demeter::example(B& b) {
    C    c;
    int    f = func();        <-- itself

   b.invert();                <-- any paramters that were passed in to the method

    a = new A();

   a ->setActive();           <-- any objects it created

   c.print();                 <-- any directly held component objects
 }
```

The Law of Demeter for functions states that any method of an object should call only methods belonging to:

Abb. 2.19: Das Law of Demeter

- zu sich selbst
- zu Paramtern der Methode
- zu einer Instanz (Klassen) variablen
- Element das innerhalb der Methode erzeugt wird.

The Pragmatic Programmer Decoupling with the Demeter Law

3 Systemarchitektur

3.1 Einführung

3.1.1 Womit beschäftigt sich ein Systemarchitekt?

Kommunikation mit externen Systemen

- Welche externen Systeme sind angeschlossen?
- Mit welcher Technologie sind diese Systeme verbunden?
- Wie gross sind die Datenströme zwischen diesen Komponenten?
- In welcher Richtung erfolgt die Kommunikation?
- Wann erfolgt die Kommunikation zwischen den Systemen und wer stosst die Kommunikation an?
- Wie werden die Daten zwischen den Systemen synchronisiert?
- Was passiert bei Ausfall eines Systems bzw. Kommunikationskanal?

Kommunikation der Komponenten innerhalb des Systems

Dieselben Fragen, die die Kommunikation mit den externen Systemen betreffen, müssen auch für die Komponenten innerhalb des Systems beantwortet werden.

Eine Komponente ist nicht auf mehreren Servern verteilt. Eine Komponente besteht aus mehreren Paketen und diese beinhalten Klassen.

3.1.2 Was zeichnet eine gute Systemarchitektur aus?

Am besten gefällt mir eine Regel die ich von Frank Buschmann während der OOP Messe in München gehört habe. Den genauen Wortlaut habe ich nicht mehr in Erinnerung aber sinngemaess war es folgendes

Jeder Programmierer sollte innerhalb weniger Minuten die Systemarchitektur aufzeichnen können.

Ein guter Systemarchitekt achtet auf Abhängigkeiten zwischen den Systemen bzw. Komponenten. Dabei helfen die Begriffe Kopplung und Kohäsion.

Kohäsion Innerer Zusammenhalt zwischen den Merkmalen einer Entwurfs- oder Konstruktionseinheit. Das Prinzip der maximalen Kohäsion fordert eine möglichst hohe Bindungsstärke innerhalb einer Entwurfs- oder Konstruktionseinheit.

Kopplung: Zusammenhang zwischen unterschiedlichen Entwurfs- oder Konstruktionseinheiten. Minimale Kopplung zielt auf die Reduktion der Bindung zwischen den Einheiten, insbesondere die Vermeidung zyklischer Benutzung.

Zwischen den Komponenten (Paketen) sollte eine geringe Kopplung sein, d.h. Komponenten (bzw. Pakete) sind weitgehend unabhängig voneinander. Dies bedeudet einer Änderung an einer Komponente (bzw. Pakete) wirkt sich wahrscheinlich nicht auf eine andere Komponente (Paket) aus.

Zwei Elemente die eine hohe Kohäsion aufweisen, sollen in derselben Komponente (Paket) sein.

3.2 Pakete

Die Entscheidung welche Klassen in welche Pakete zusammengefasst werden ist nicht trivial. Im folgenden werden einige Aspekte bzw. Regeln zusammengefasst. Man wird allerdings nie alle Regeln gleichzeitig einhalten können.

3.2.1 Horizontale und Vertikale Paket Bildung

Oft habe ich eine **horizontale** Paket Bildung gesehen. Die Persistenzschicht, die Klassen mit dem Business Objekten und die Präsentationsschicht sind in eigenen Paketen. Wenn man die verschiedenen Schichten auf verschiedenen Servern laufen lassen möchte, ist dies auch sinnvoll. Ich sehe den Nachteil dass man meisten auch bei kleinen Änderungen in allen Schichten eingreifen muss. Ist z.B: die Anforderung einer Person das Attribut E-Mail Adresse hinzuzufuegen, wird meistens das Domain Objekt Person geändert, auf der Datenbank in der Tabelle eine entsprechende Spalte eingefuegt, und die E-Mail Adresse an der Oberfläche angezeigt.

Eine **vertikale** Paket Bildung ordnet die Klassen die fachlich zusammengehören in ein Paket. So wären die Oberflächen Elemente einer Person, die Datenbank Mapping Klassen, und das Domain Objekt im selben Paket. Dies hat die Vorteil dass sich eine Änderung wahrscheinlich nur in einem Paket auswirkt. Der Nachteil ist das diese Paketbildung es erschwert die Oberfläche bzw. die Datenbank auf einem eigenen Server auszulagern; oder vielleicht die Oberfläche bzw. Datenbank auszutauschen.

Eine Kombination von **vertikale** und **horizontale** Paketbildung kann zu einer Vielzahl von Paketen führen.

3.2.2 Kriterien die Pakete erfüllen sollen

Es gibt Metriken, die die Kohäsion und Kopplung messen, allerdings sind diese Berechnungen entweder kompliziert oder wenig brauchbar.

Schon etwas praktischer sind folgende Kriterien die eine Konstuktionseinheit (Paket) erfüllen soll, aus dem Objektorienterten Konstruktionshandbuch:

- **Zerlegbarkeit:** Ein Entwurfsproblem sollte sich in kleinere, weniger komplexe Teilprobleme zerlegen lassen, die entsprechend als Entwurfs- und Konstruktionseinheiten abgebildet werden. Diese sollten eine einfache Struktur bilden und weitgehend unabhängig konstruiert werden können.
- **Kombinierbarkeit:** Die Entwurfs- und Konstruktionseinheiten sollten sich durch einfache Rekombination zu neuen Softwaresystemen in verschiedenen Anwendungsbereichen zusammen-fügen lassen.
- **Verständlichkeit:** Jede Entwurfs- und Konstruktionseinheit eines Softwaresystems sollte weitgehend unabhängig von den anderen verständlich sein.
- **Kontinuität:** Eine Änderung des Entwurfsproblems, die aus dem Anwendungsbereich oder dem technischen Kontext eines Softwaresystems resultiert, sollte Änderungen nur in einer oder wenigen Entwurfs- und Konstruktionseinheiten erfordern. Um diese Kriterien erfüllen zu können, müssen wir folgende Regeln beachten:
- **Direkte Abbildung:** Die Struktur des Softwaresystems sollte im engen Zusammenhang mit den im Anwendungsbereich identifizierten Strukturen stehen.
- **Wenige Schnittstellen:** Jede Entwurfs- und Konstruktionsein-heit sollte mit möglichst wenig anderen interagieren.
- **Kleine Schnittstellen:** Wenn zwei Entwurfs- und Konstruktionseinheiten interagieren, sollten sie so wenig Information wie möglich und nötig austauschen.

- **Explizite Schnittstellen:** Wenn zwei Entwurfs- und Konstruktionseinheiten interagieren, sollte dieser Austausch explizit sein.
- **Geheimnisprinzip:** Nur relevante Merkmale einer Entwurfs-und Konstruktionseinheit sollten für Klienten sichtbar und zugänglich sein.
- **Offen-Geschlossen-Prinzip:** Entwurfs- und Konstruktionsein-heiten sollten sowohl offen als auch geschlossen sein

Eine Entwurfs- oder Konstruktionseinheit ist **offen**, wenn sie weiterentwickelt werden kann.
Eine Entwurfs- oder Konstruktionseinheit ist **geschlossen**, wenn sie stabil von Klienten verwendet werden kann.

OO Konstruktionshandbuch Kapitel 2.1.9: Modularisierung

3.2.3 Prinzipien des Paket Designs

Robert C. Martin hat folgende Package Design Regeln veröffentlicht. Man wird allerdings wohl nie alle Prinzipien leichzeitig benutzen können.

The Release Reuse Equivalency Principle (REP)

Füge Klassen die gemeinsam benutzt werden, bzw. gemeinsam freigegeben werden in ein Paket

The Common Closure Principle CCP

Füge Klassen die gemeinsam für eine Sache verantwortlich sind in ein Paket

The Common Reuse Principle CRP

Trenne Klassen die von verschiedenen Klienten benutzt werden, in verschiedene Pakete.

The Acyclic Dependencies Principle ADP

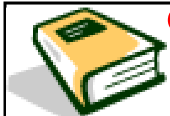
Vermeide unbedingt Zyklen zwischen den Abhängigkeiten von Paketen. Für Java Programmierer hilft hier ein hervorragendes Open-Source Tool JDepend dies zu vermeiden.

The Stable Dependencies Principle SDP

Wenn Paktet A von Paket B abhängig ist, dann sollte Paket B stabiler gegenüber Änderungen sein als Paket A. Das schon erwähnte Tool JDepend benutzt Metriken um die Stabilität eines Pakets zu messen.

The Stable Abstractions Principle SAP

Ein Paket bei denen Änderungen aufwendig sind, sollte nicht von einem Paket abhängig sein, bei dem Änderungen sehr einfach möglich sind.


OO The Principles, Practices and Patterns of Agile Software Development

3.3 Schichtenmodell

Wir haben gelernt, die Anzahl der Klienten einer Schnittstelle möglichst klein zu halten; Pakete sollten moeglichst eigenständige Einheiten, mit so wenig Abhängigkeit zu anderen Paketen, sein. Eine Möglichkeit zum Anordnen von Paketen bilden Schichtenmodelle. Als Beispiel dient das OSI-Schichtenmodell, sicherlich eines der bekanntesten Schichtenmodelle in der Informatik.

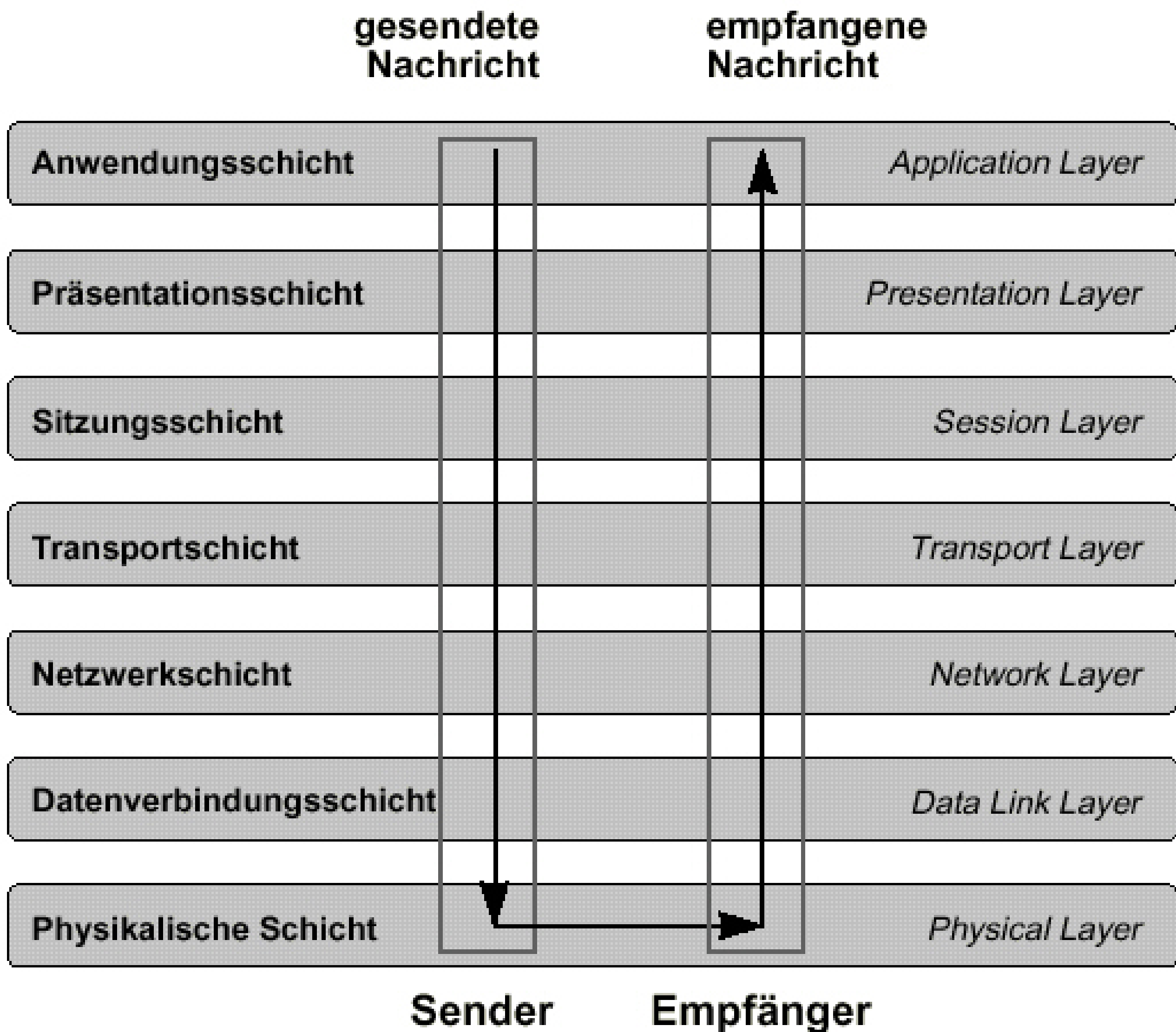


Abb. 3.1: Bekanntes Beispiel für ein Schichtenmodell

Typisch an diesem Schichtenmodell ist

- Der Abstraktionsgrad nimmt von der obersten bis zur untersten Ebene ab.
- Eine Schicht ruft nur Operationen derselben Schicht, bzw. der nächst unteren Schicht auf.
- Es wird nie eine Schicht übersprungen

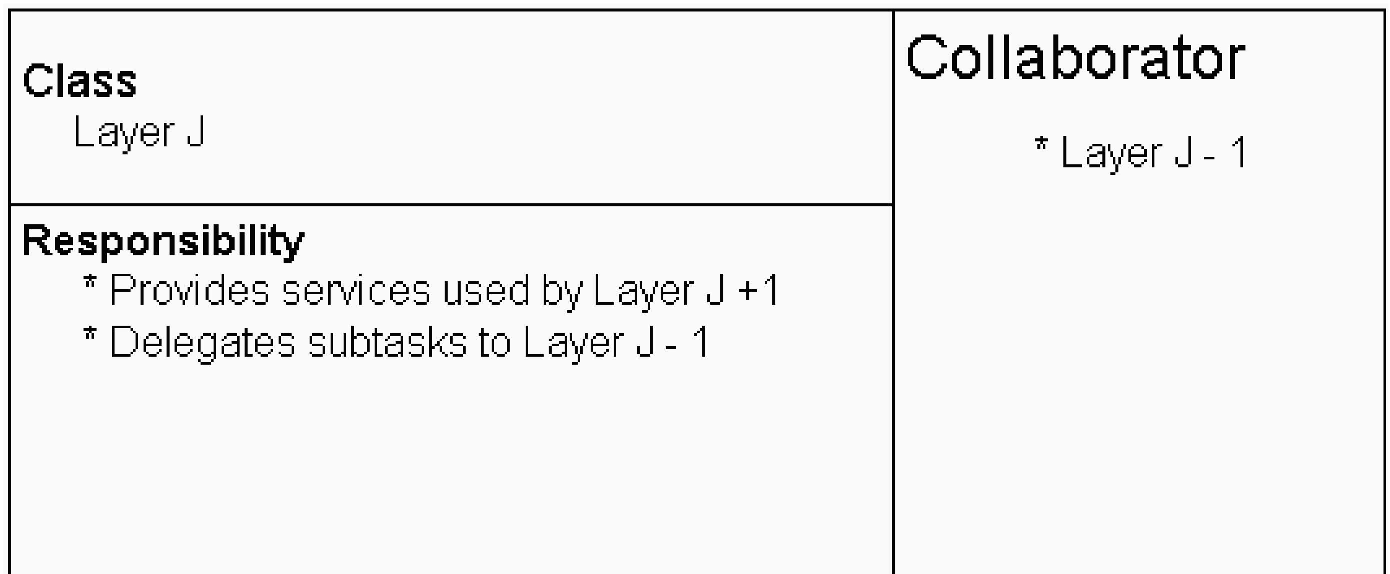


Abb. 3.2: CRC Karte fuer einen Layer

Ein Schichtenmodell unterstützt die Wartbarkeit, da

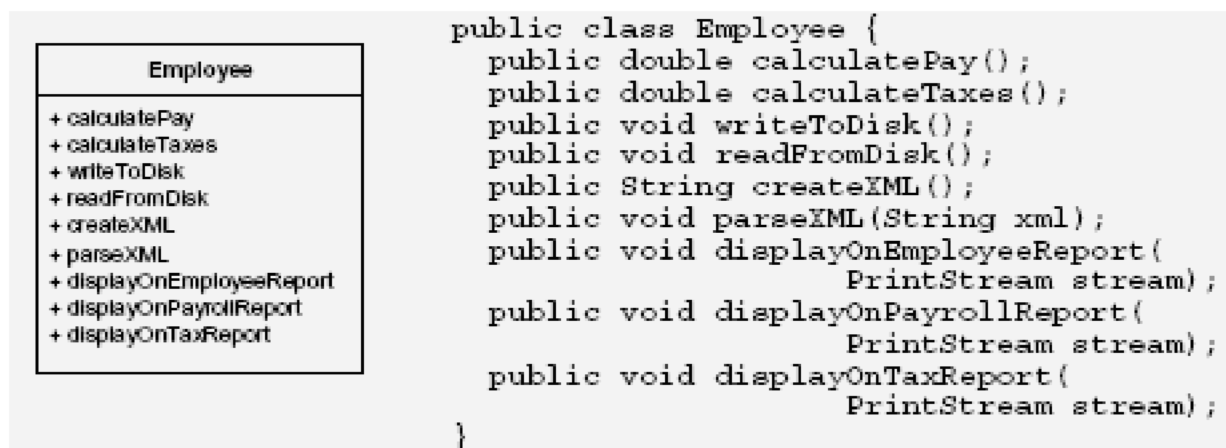
- Ändere ich die Schnittstelle in einer Schicht, so muss ich nur die Objekte der Nachbarschicht anpassen.
- Austausch einer Schicht, hat keine Auswirkung auf die Umgebung (solange die Schnittstellen) eingehalten werden
- Der Methodenfluß ist eindeutig. Er geht i.a. immer von der obersten zur untersten Schicht bzw. umgekehrt. (Variationen sind denkbar, z.B. Schichten die die Ergebnisse cachen)

3.4 Objektorientierte Designprinzipien

Es gibt wahrscheinlich kein komplexeres Programm das nicht gegen mindestens einer der folgenden Designprinzipien vertösst. Man sollte die Designprinzipien nicht als Bibel, aber trotzdem ernst nehmen.

3.4.1 Das Single Responsibility Prinzip SRP

Was ist an dieser Klasse problematisch?.




```
public class Employee {
  public double calculatePay();
  public double calculateTaxes();
  public void writeToDisk();
  public void readFromDisk();
  public String createXML();
  public void parseXML(String xml);
  public void displayOnEmployeeReport(
                   PrintStream stream);
  public void displayOnPayrollReport(
                   PrintStream stream);
  public void displayOnTaxReport(
                   PrintStream stream);
}
```

Abb. 3.3: Klasse die das SRP Prinzip verletzt

Die Klasse in der obigen Abbildung weiss zuviel.

In der Anfangszeit der Objektorientierung hat man fehlerhaft gemeint, ein Domain Objekt weiss selber wie es sich abspeichert, wie es sich selber an der Benutzungsoberfläche darstellt usw.

Wenn man in der obigen Klasse Employee statt in ein XML-Stream in eine Datenbank abspeichert, muss die Klasse Employee geändert werden. Das SRP sagt, dass verschiedene Verantwortlichkeiten in verschiedenen Klassen implementiert werden soll.

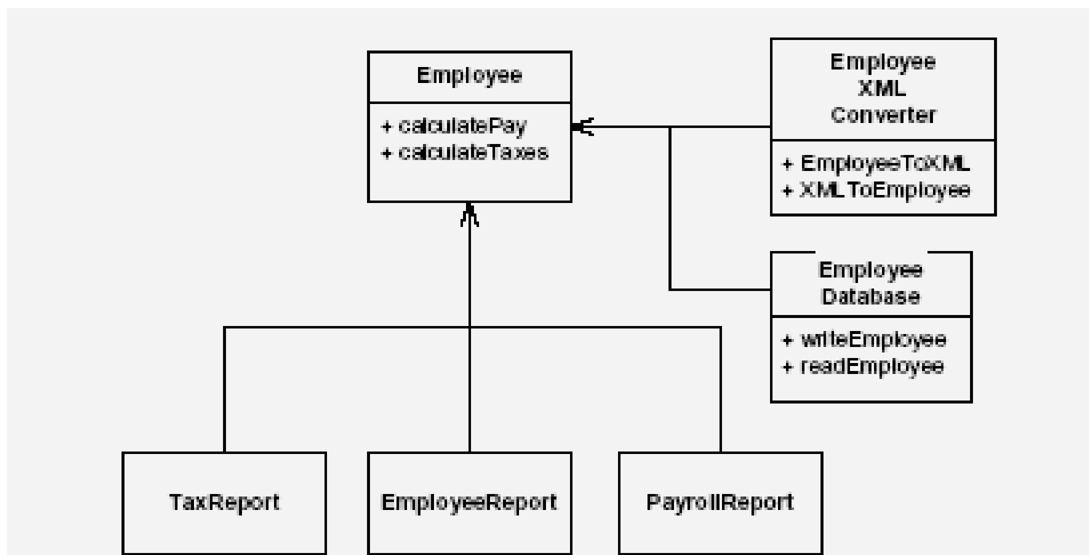


Abb. 3.4: Klassen die das SRP Prinzip berücksichtigen

3.4.2 Das Open Closed Prinzip OCP

Das Open Closed Prinzip besagt, dass es möglich sein sollte ein Modul zu erweitern, ohne es selber verändern zu müssen.

SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.

Betrachte folgendes Klassendiagramm:

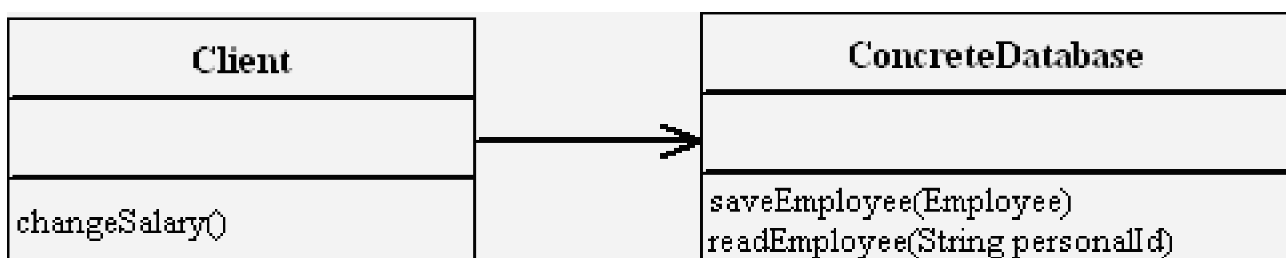


Abb. 3.5: Verletzung Open Closed Prinzip

Im Client verdoppelt die Methode changeMySalary() meinen Gehalt. Schliesslich habe ich es verdient.

```
public void changeMySalary(){
```

```
    ConcreteDatabase database = ConcreteDatabase.soleInstance();
    String personalIdKlausMeucht = "4711";
    Employee ich = database.readEmployee(personalIdKlausMeucht);
    ich.setSalary(ich.getSalary() * 2);
    database.saveEmployee(ich);
}
```

Diese Methode zeigt allerdings, dass ich eine Gehaltserhöhung nicht verdient habe. (Übersehen wir mal die Verwendung von singleton und des MagicValues 4711)

Es ist schwer einen Unittest für diese Methode zu schreiben, da die Klasse Client und ConcreteDatabase fest miteinander verkoppelt sind.

Ein Unittest mit der Verwendung einer konkreten Datenbank ist nicht sinnvoll, da das Schreiben auf die Datenbank Zeit kostet und die Unittests unabhängig von der Datenbank sein soll.

Sinnvoll wäre es die ConcreteDatabase mit einer simulierten Datenbank aus Softwarebasis auszutauschen. Da der Client und ConcreteDatabase eng miteinander gekoppelt sind, kann ich die Datenbank nicht austauschen. Der Client ist nicht offen gegenüber einen Austausch der Datenbank.

Eine Möglichkeit für die Entkopplung ist die Verwendung von Interfaces. Der Client sollte nicht von einer konkreten Implementierung, sondern nur von einer Schnittstelle abhängig sein. Damit kann man die konkrete Implementierung austauschen.

Im folgenden Beispiel ist der Client offen gegenüber einer neuen Datenbankklasse. Ein Unittest kann nun sehr einfach die Datenbank austauschen.

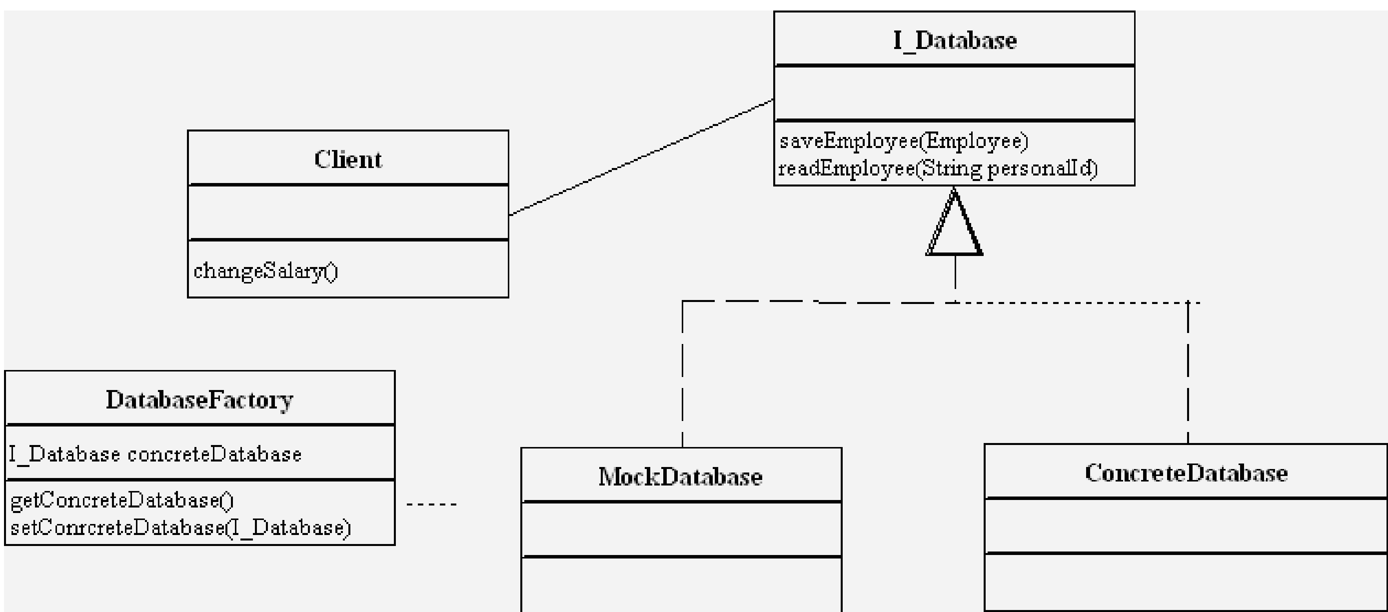


Abb. 3.6: Offen gegenüber Austausch der Datenbank

Die Methode changeMySalary() sieht folgendermassen aus.

```
public void changeMySalary(){
    I_Database database = DataseFactory.getConcreteDatabase();
    String personalIdKlausMeucht = "4711";
    Employee ich = database.readEmployee(personalIdKlausMeucht);
    ich.setSalary(ich.getSalary() * 2);
```

```
    database.saveEmployee(ich);
}
```

3.4.3 The Liskov Substitution Prinzip LSP

Unterklassen sollen anstelle ihrer Oberklassen einsetzbar sein. Unterklassen sollten Methoden von Oberklassen immer erweitern, aber nicht wegnehmen.Vermeide es Methoden der Oberklassen durch leere Implementierungen zu überschreiben. Beim Überschreiben von Methoden aus einer Oberklasse sollte sichergestellt sein, dass die Unterklasse in jedem Fall für die Oberklasse einsetzbar bleibt.

Beipsiel für Verletzung des Liskov Prinzips:

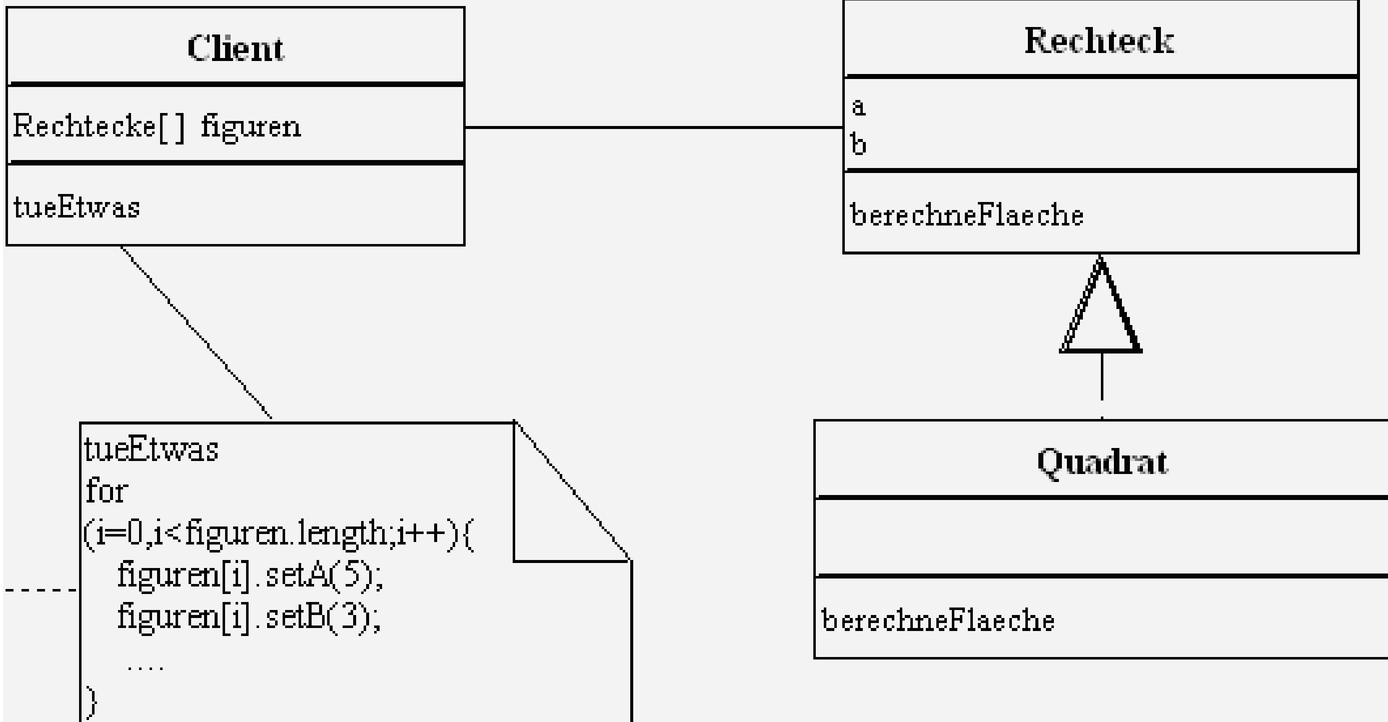


Abb. 3.7: Verletzung Liskov Prinzip

Obwohl wir alle aus dem Mathematik Unterricht wissen, dass ein Quadrat ein Rechteck ist; ist es problematisch ein Quadrat als Unterklasse von Rechteck zu implementieren. Das Problem liegt darin dass ein Quadrat Eigenschaften eines Rechtecks nicht erweitert sonder einschränkt. In der Methode tueEtwas müsste man um korrekt zu sein, abfragen ob ein Rechteck in Wirklichkeit ein Quadrat ist.

D.h der Client muss Wissen über die Unterklassen von Rechteck haben und dies verstösst neben dem Liskov Prinzip auch das Open-Closed Prinzip

3.4.4 The Dependency Inversion Principle DIP

Nutzer einer Dienstleistung sollten möglichst von Abstraktionen (d.h abstrakten Klassen oder Interfacaes) , nicht aber von konkreten Implementierungen abhängig sein. Abstraktionen sollten auf keinen Fall von konkreten Implementierungen abhängig sein.

3.4.5 The Interface Segregation Principle ISP

Clients sollten nicht von Diensten abhängig, 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.

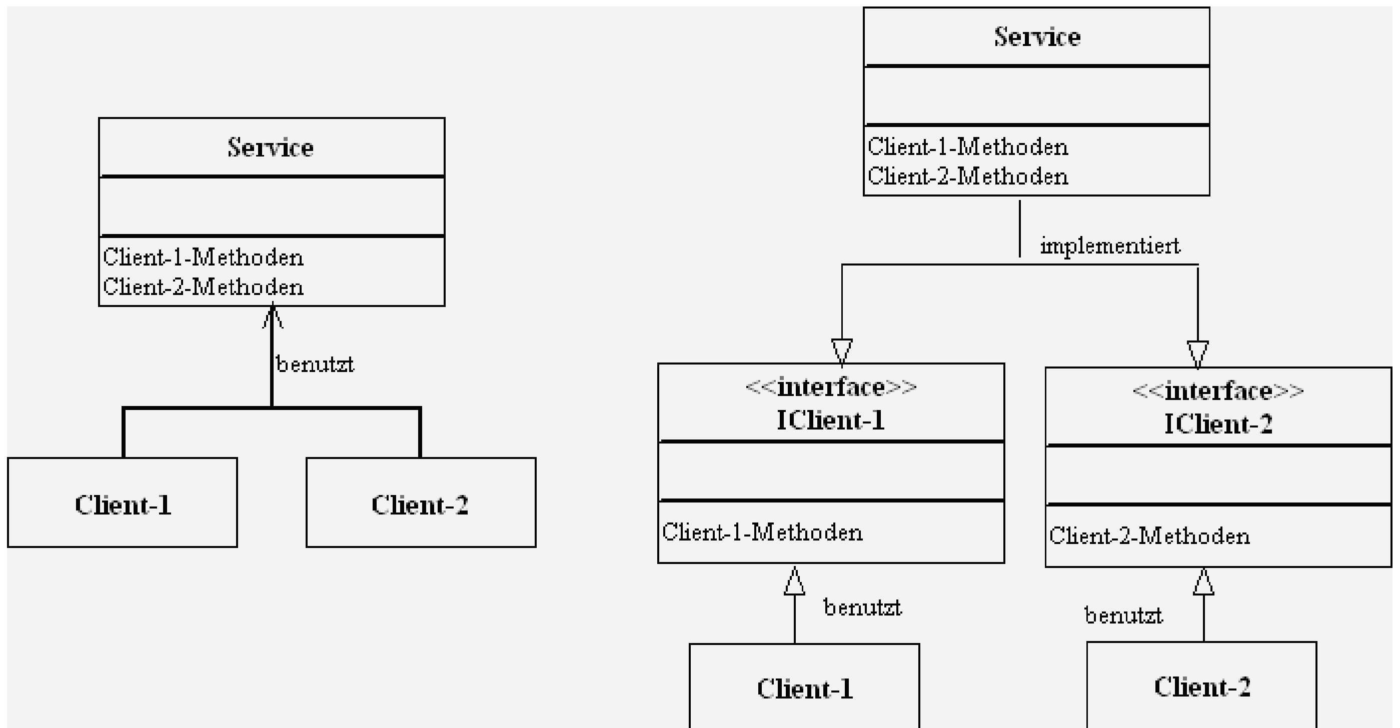


Abb. 3.8: Interface Segregation Principle

3.5 Entkopplung

3.5.1 Fassade Pattern

Klassen kapseln Zustand und Operationen. Subsysteme kapseln Klassen. Subsysteme lassen sich mit Hilfe von Fassadenobjekten definieren.

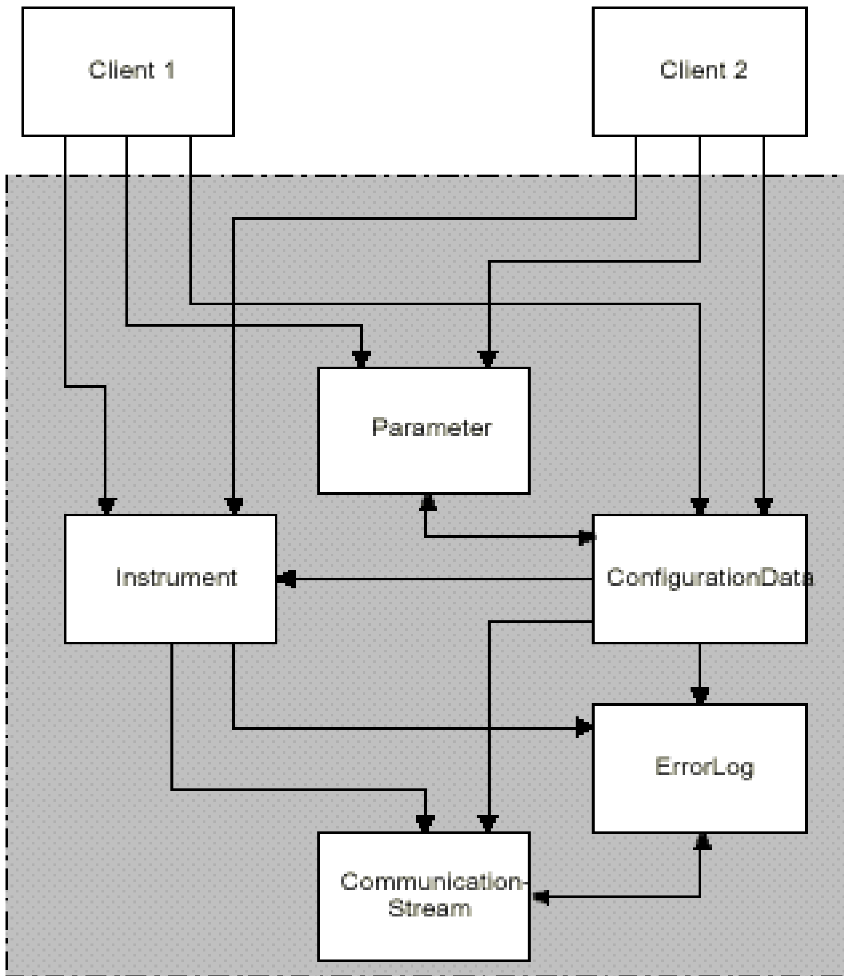


Abb. 3.9: Ohne Fassade Pattern

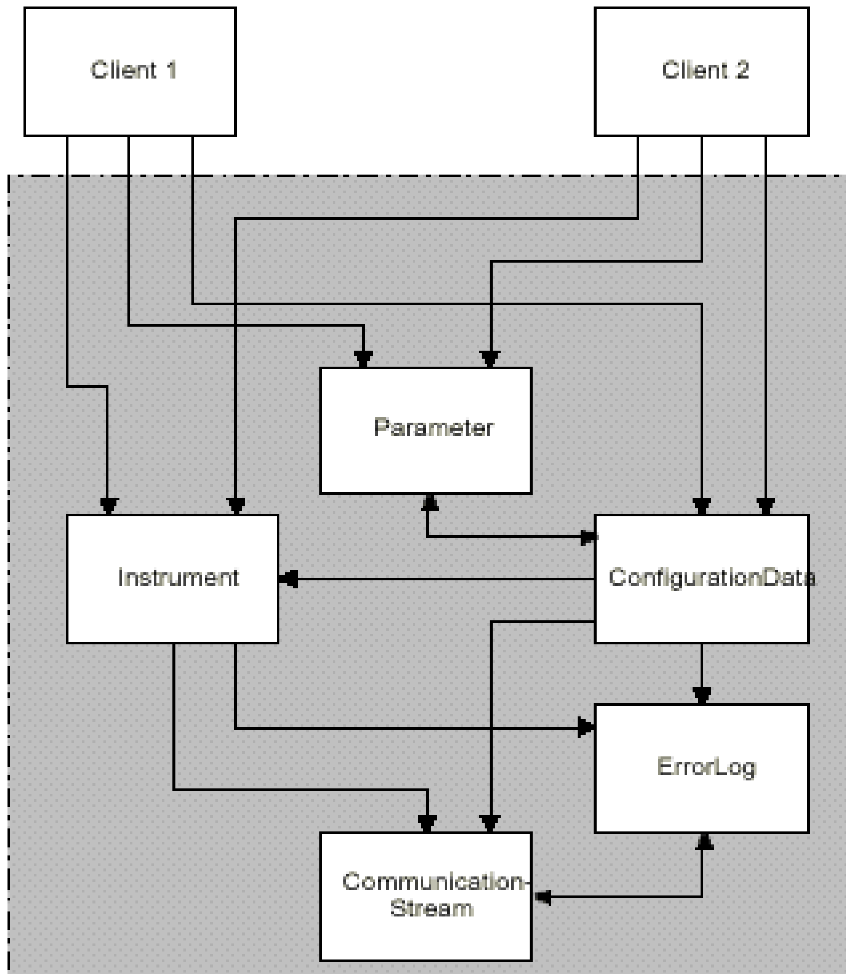


Abb. 3.10: Mit Fassade Pattern

Vorteile des Fassade Patterns:

- Die Clienten brauchen nur die Schnittstelle des Fassadenobjekts zu beachten.
- Ziel der Fassadenklasse ist es, den Klienten eine einfachere Schnittstelle zu liefern.
- Die Objekte und Schnittstellen innerhalb der Fassade können ohne Einfluss auf die Clienten geändert werden; damit habe ich eine Entkopplung von Subsystem und den Clienten.
- Das Fassadenobjekt kann es i.a. nicht verhindern, daß der Client direkt auf eine Klasse innerhalb des Subsystems zugreift.

Allerdings besteht die Gefahr, daß das Fassadenobjekt zu gross wird, das Fassadenpattern eignet sich nur für kleine Subsysteme.

3.5.2 Inverted Association Pattern

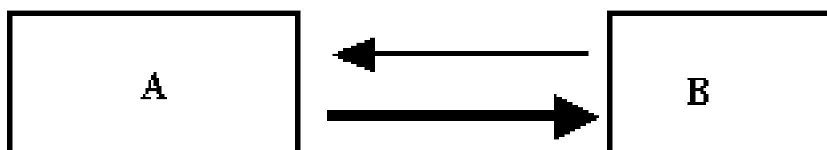


Abb. 3.11: Zirkuläre Abhängikeit zweier Klassen

Das Inverted Association Pattern ist eine Verallgemeinerung des Beobachter Patterns, das weiter unten ausführlich erklärt wird.

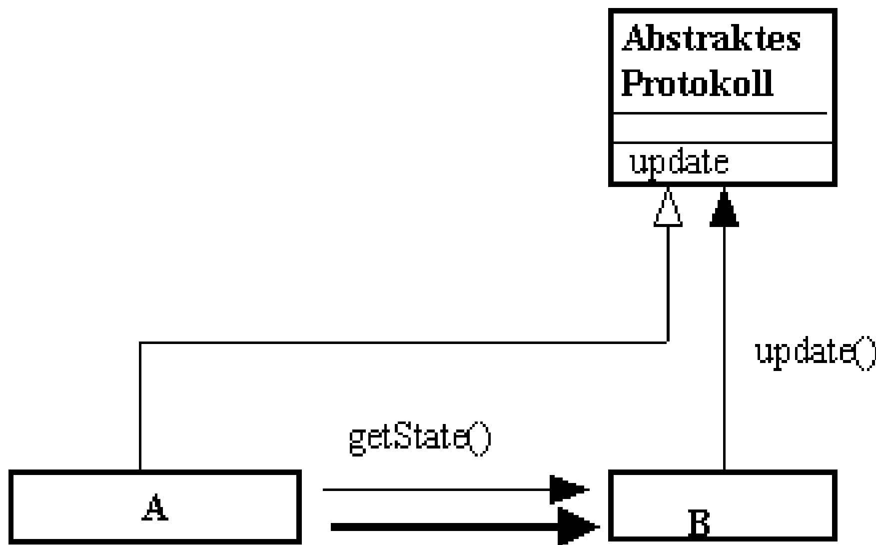


Abb. 3.12: Abhängigkeit aufgelöst

Dieses Pattern wird genommen, wenn 2 Klassen gegenseitig voneinander abhängig sind, aber eine Richtung dominiert. Im unteren Beispiel geht der Methodenfluß hauptsächlich von Klasse A nach Klasse B. Die Idee ist, die nicht dominierende Richtung nur durch ein einfaches standardisiertes Protokoll zu realisieren. Eine abstrakte Klasse definiert das Protokoll. Die Klasse B teilt der Klassse A nur die notwendigsten Dinge über das standardisierte Protokoll mit. Siehe ausführliches Beispiel im Beobachter Pattern.

4 Benutzungsoberläche

4.1 Trennung Interaktion und Funktionalitätskomponente

Zur Einführung ein abschreckendes Beispiel aus einem realen Projekt

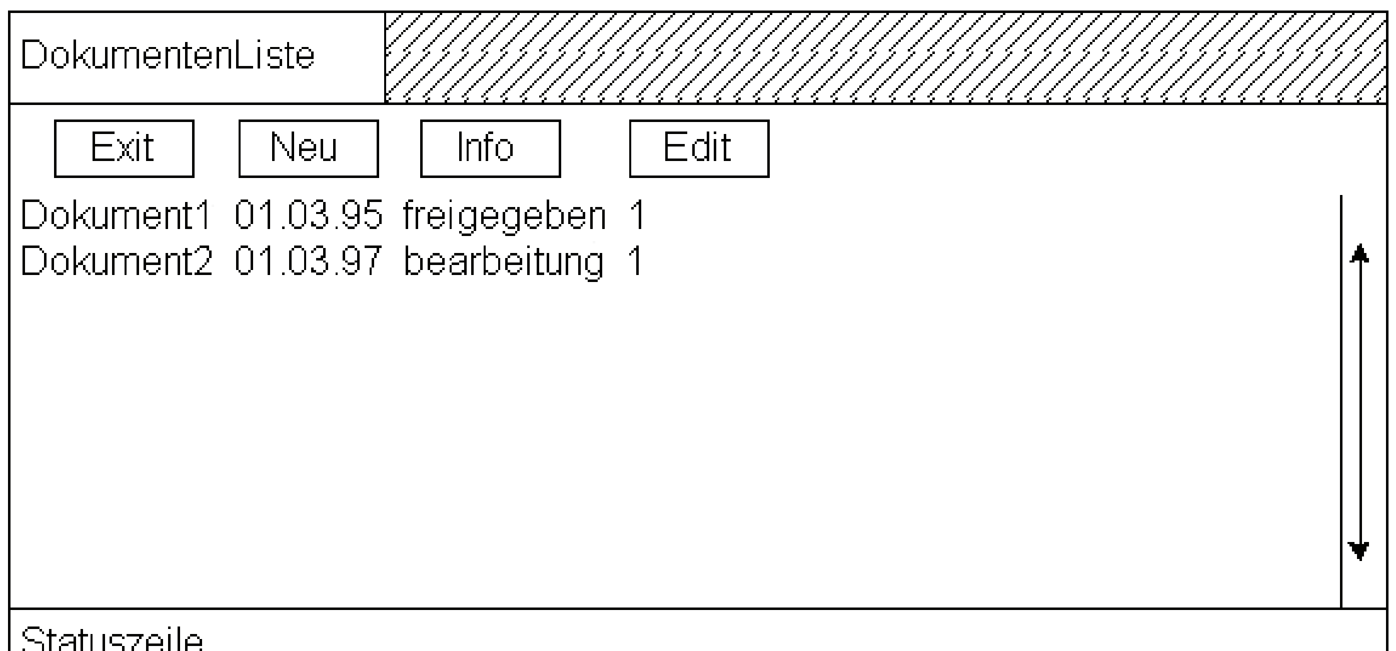


Abb. 4.1: Ein abschreckendes Beispiel

Hier das entsprechende Objektmodell:

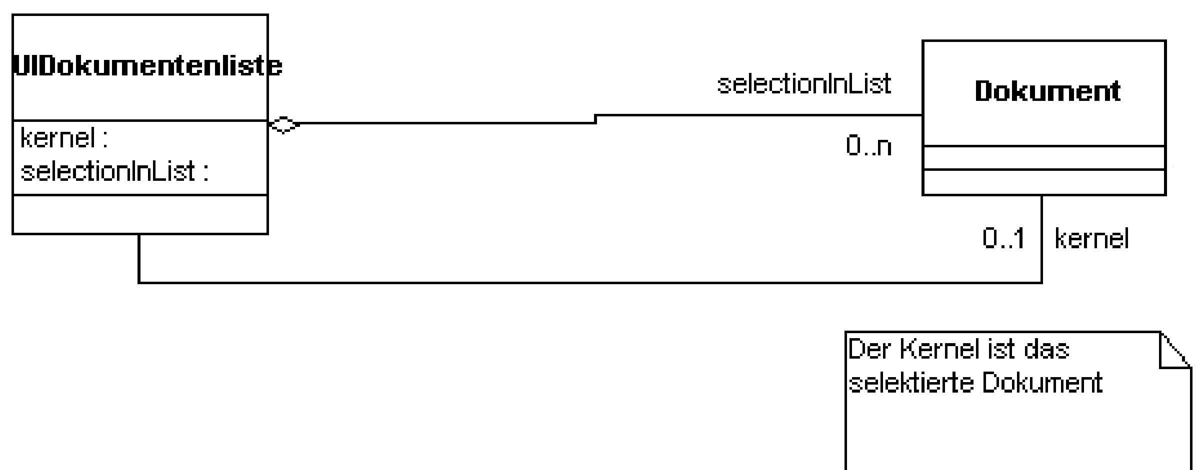


Abb. 4.2: Direkte Verbindung von UI zur Domain Schicht

Diese Lösung erzeugt einige Probleme:

- Der Kernel des UI-Objekts wechselt, jenachdem welches Dokument selektiert ist.
- Falls gerade kein Dokument selektiert ist, ist der kernel nil
- Sind mehrere Dokumente selektiert, was ist der Kernel?
- Bei jedem Aufruf der Operation ist eine Fallunterscheidung notwendig; ob mehrere, genau

ein oder kein Dokument selektiert ist
- Die Funktionen zum Neuanlegen eines Objekts, und alle andere Operationen, die sich nicht auf das selektierte Dokument beziehen, müssen zwangsläufig in der UI-Schicht realisiert werden.

Hier ein besserer Design:

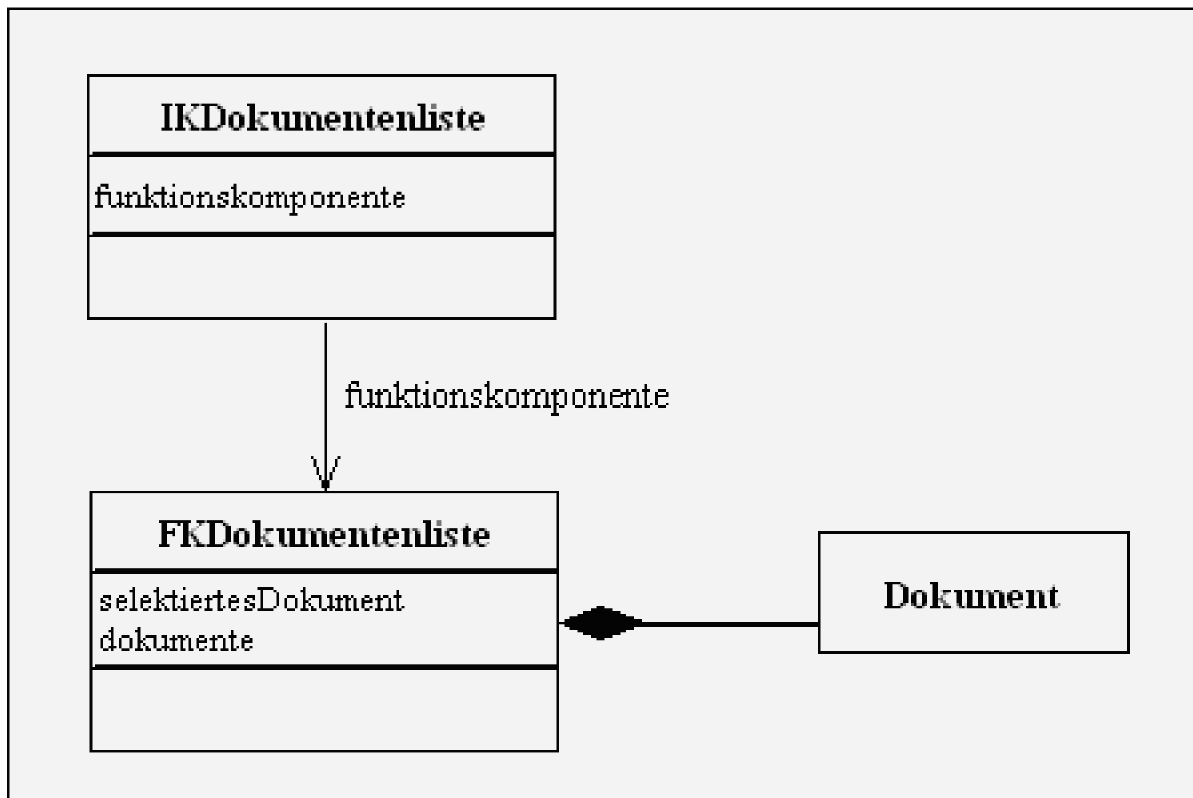


Abb. 4.3: Trennung Interaktion und Funktion

Merke: Teile ein darzustellendes Fenster in eine Interaktion und Funktionskomponente auf.

Ich selbst habe mir beim Design folgende Regeln auferlegt:

- Jedem Interaktionskomponente ist genau eine Funktionskomponente zugeordnet, umgekehrt kann ein Funktionskomponente, mehrere oder keine Interaktionskomponente zugeordnet sein.
- Das Interaktionskomponente kennt seine Funktionskomponente (feste Kopplung) i.a. über eine Instanzvarialble.
- Funktionskomponenten kennen ihre Interaktionskomponenten nicht (direkt). (siehe lose Kopplung mit Hilfe des Beobachter Patterns)
- Interaktionen finden nur in den Interaktionskomponenten statt (d.h Interaktionskomponenente enthält nur Widgets)
- Die Domain Objekte hängen an der Funktionskomponente.

4.2 Bildung komplexer Fenster aus einfachen

Oft werden komplexe Fenster, modular aus einfacheren Fenster zusammengebaut. Ein typischens Beispiel dafür sind Notebooks; jenachdem welchen Reiter man anwaehlt ist ein anderes Subfenster aktiv. Ein Subfenster kann wiederum aus mehreren Subfenstern bestehen. So entsteht ein Baum von Fenstern.

Ich empfehle folgende Richtlinien beim Bau komplexer Fenster:

- Jedes (Unter-)Fenster soll als eigenständiges Fenster, entwickelt und getestet werden.
- Entwickle zuerst die Unterfester, teste die Funktionalität und entwickle und teste zuletzt das oberste Fenster
- Jedes (Unter-) Fenster besteht aus einer Interaktions- und aus einer Funktionskomponente
- Jedes Fenster kennt seine direkten Unterfenster (über eine Instanzvariable).
  - Eine Funktionskomponente eines Fensters kennt seine Funktionskomponenten der direkten Unterfenster
  - Eine Interaktionskomponente eines Fensters muss nicht aber kann die Interaktionskomponente der direkten Unterfenster kennne.
- Die Funktionskomponente eines Unterfensters kennt die Funktionskomponente des übergeordneten Fensters nicht direkt. Kommunikation erfolgt mit Hilfe des Beobachterpatterns
- Eine Fenster koordiniert seine (nur)die direkten Unterfenster. Eine Interaktion im Fenster hat eine der folgenden Auswirkungen:
  - Es wird im Kernel Objekt des Fensters eine Aktion ausgeloest
  - Die Interaktion wird an einen oder mehrere der Unterfenster delegiert.
  - Ggf. wird in Unterfenstern eine andere Interaktion als die im Oberfenster aufgetretene ausgelöst.

Siehe dazu die Abbildung aus dem objektorientierten Konstruktionshandbuch. IAK bedeudet dabei Interaktionskomponente und entspricht den eigentlichen Fensterobjekten. FK bedeudet Funktionskomponenten. Gestrichelte Linien kennzeichnen indirekte Kommunikation über Beobachter Pattern.

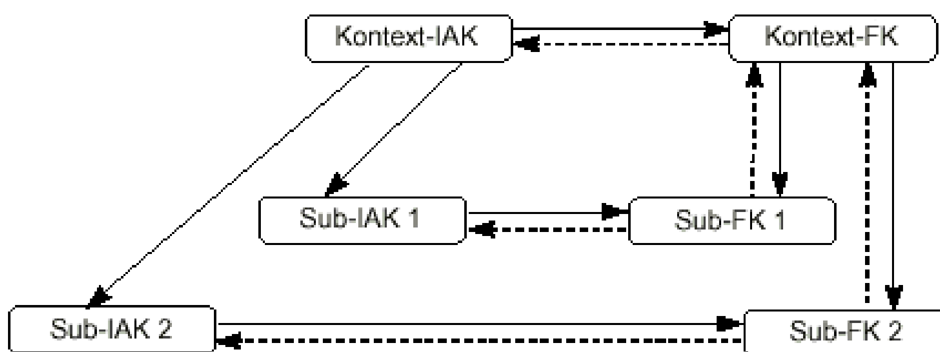


Abb. 3-44
Komposition von FKs und IAKs

Abb. 4.4: Struktur komplexer Fenster

4.3 Kapseln der Benutzeraktionen durch Command Pattern

Im obigen Kapitel wird vom Interaktionsobjekt aus, der Befehl zum Erzeugen eines neuen Dokumentes ausgelöst. Der Erzeugen eines neuen Objekts findet in der Funktionskomponente statt

```
IKDokumentListe >> void activateCreateNewDokument(){
     funktionskomponente.neuesDokument().
}
```

Warum einfach wenn es auch kompliziert geht

```
IKDokumentListe >> void activateCreateNewDokument{
     this.createNewDokumentCommand().execute();
}
```

```
IKDokumentListe >> Command createNewDokumentCommand(){
```

```
// Erzeuge ein Command Objekt mit der Funktionskomponente als Wert der
// Instanzvariable receiver
```

```
Command commandObjekt := new CreateNewDokumentCommand().
        commandObjekt.setReceiver(this.funktionskomponente());
        return commandObjekt ;
}
```

```
CreateNewDokumentCommand >> void execute(){
        receiver.neuesDokument();
}
```

Nachteile der komplizierten Version:

- schwerer zu verstehen
- zusätzlicher Code
- langsamer (vernachlässigbar)

Vorteile der komplizierten Version:

- Benutzeraktionen können gespeichert und protokolliert werden
- Hilfreich bei Fehlersuche
- Benutzer kann kontrollieren was er zuletzt gemacht hat
- Die protokollierten Benutzteraktionen können automatisiert wiederholt werden. Ideal fuer Schulungen
- Vereinfacht Undo und Redo
- Vereinfacht Bilden von Macros

Eine Erweiterung des Command Patterns, ist das Command-Processor Pattern. Es wird u.a. für Undo's verwendet. Bei dem Command-Pattern sind i.a 5 Klassen beteiligt:

Class Abstract Command	**Collaborators**
Responsibility *Defines a uniform interface to execute commands *Extends the interface for services of the command processor, such as undo and logging	

Class Command	**Collaborators**
Responsibility *Encapsulates a function request * Implements interface of abstract command * Uses suppliers to perform a request	* Supplier

Class Controller	**Collaborators**
Responsibility * Accept service requests * Translates requests into commands * Transfers commands to command processor	*Command Processor *Command

Class Command Processor	**Collaborators**
Responsibility *Activates command execution *Maintains command objects *Provides additional services related to command execution	Abstract Command

Class Supplier	**Collabrators**
Responsibility * Provides application specific functionality	

Abb. 4.5: CRC Karten Command Pattern

Hier das zugehörige Objektmodell:

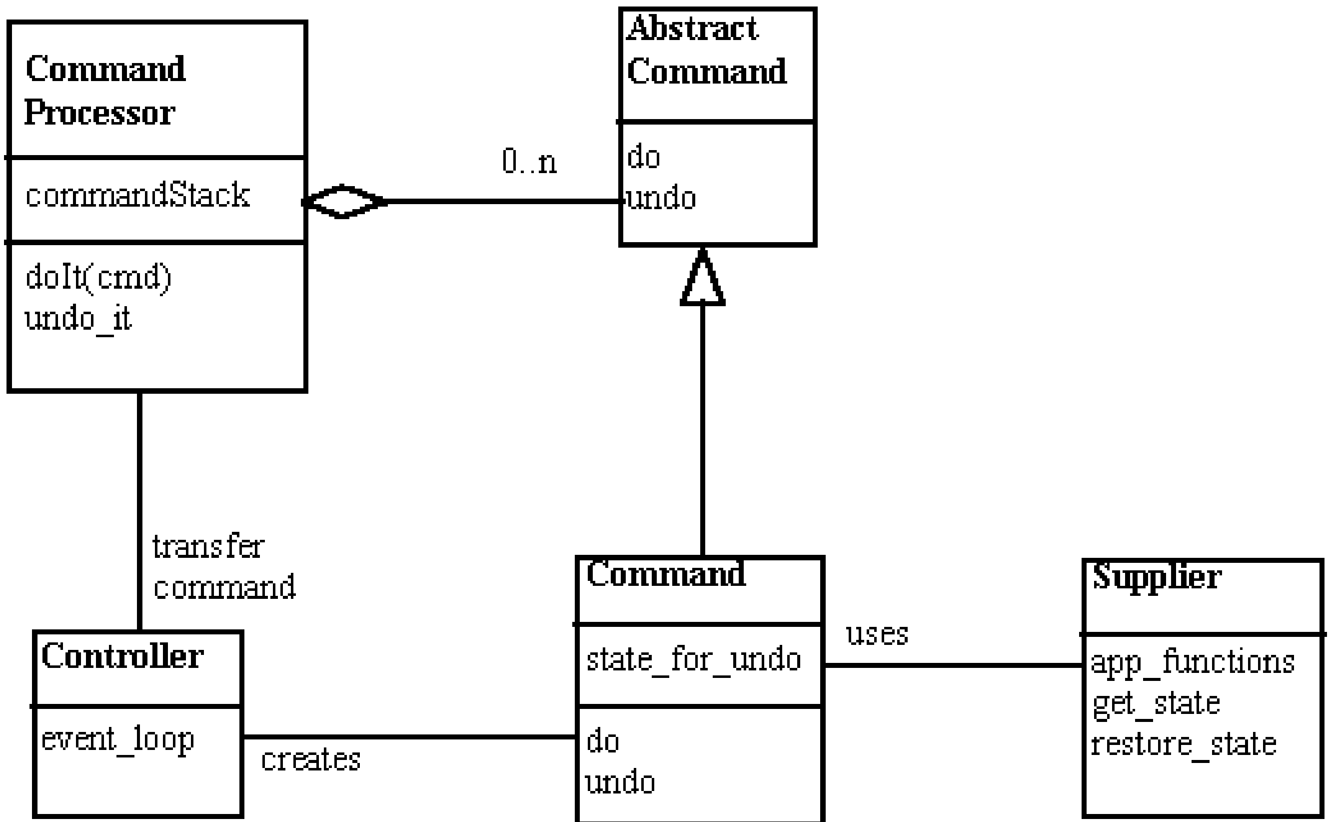


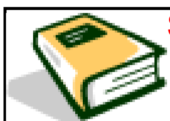
Abb. 4.6: ObjektModell Command Pattern

Beispiel:Der Anwender hat in einem Text ein Wort selektiert, und möchte in dem Wort alle Buchstaben in Grossbuchstaben umwandeln. Im einzelnen passiert folgendes:

1. Der Controller empfängt in seiner EventLoop den Event Capitalize Selection
2. Es wird ein Capitalize Command erzeugt
3. Der Capitalize fragt seinem Supplier (einen Text) nach dem Status und speichert den Status in einer Instanzvariable:
   1. Welcher Bereich im Text selektiert ist
   2. Eine Kopie des selektierten Texts
4. Der erzeugte Capitalize Command wird ausgelöst und im Stack des Command-Processors gespeichert

Bei einem Undo werden folgende Aktionen ausgelöst:

1. Der Controller empfängt den Undo Event
2. Aus dem CommandProcessor wird vom Stack den zuletzt ausgeführten Command (im Beispiel Capitalize Command) geholt
3. Im Capitalize Command wird die Funktion Undo ausgelöst
4. Die Informationen um das Undo erfolgreich zu bewerkstelligen, steht in einer entsprechenden Instanzvariable des Capitalize Command


System of Pattern Command Processor

4.4 Das Beobachter Pattern

Achte bei der Kopplung zwischen UI und Domain Schicht auf folgende Eigenschaften:

- feste Kopplung von Interaktionskomponente zum Funktionskomponente, d.h die Interaktionskomponente kennt sein Kernel Objekt
- Die Funktionskomponente kennt die Interaktionskomponente nicht.Dadurch ist es möglich die Interaktionskomponente auszutauschen.

Problem: Änderung an der Funktionskomponente müssen angezeigt werden.

- Lösung1: Interaktionskomponente pollt jedesmal auf die Funktionskomponente. Dadurch erreicht man totale Unabhaengigkeit der Funktionskomponente zur Interaktionskomponente. Der Methodenfluss geht nur in einer Richtung. Allerdings ist diese Lösung extrem inperformant
- Lösung2: Man erlaubt Methodenfluß von Funktionskomponente zur Interaktionskomponente allerdings nur über eine standardisierte Schnittstelle. Die Funktionskomponenten teilen allen interessierten Interaktionskomponente mit, daß sie sich geändert haben. Die Interaktionskomponenten, lesen alle notwendigen Informationen aus ihrer Funktionskomponente und stellen sich neu dar

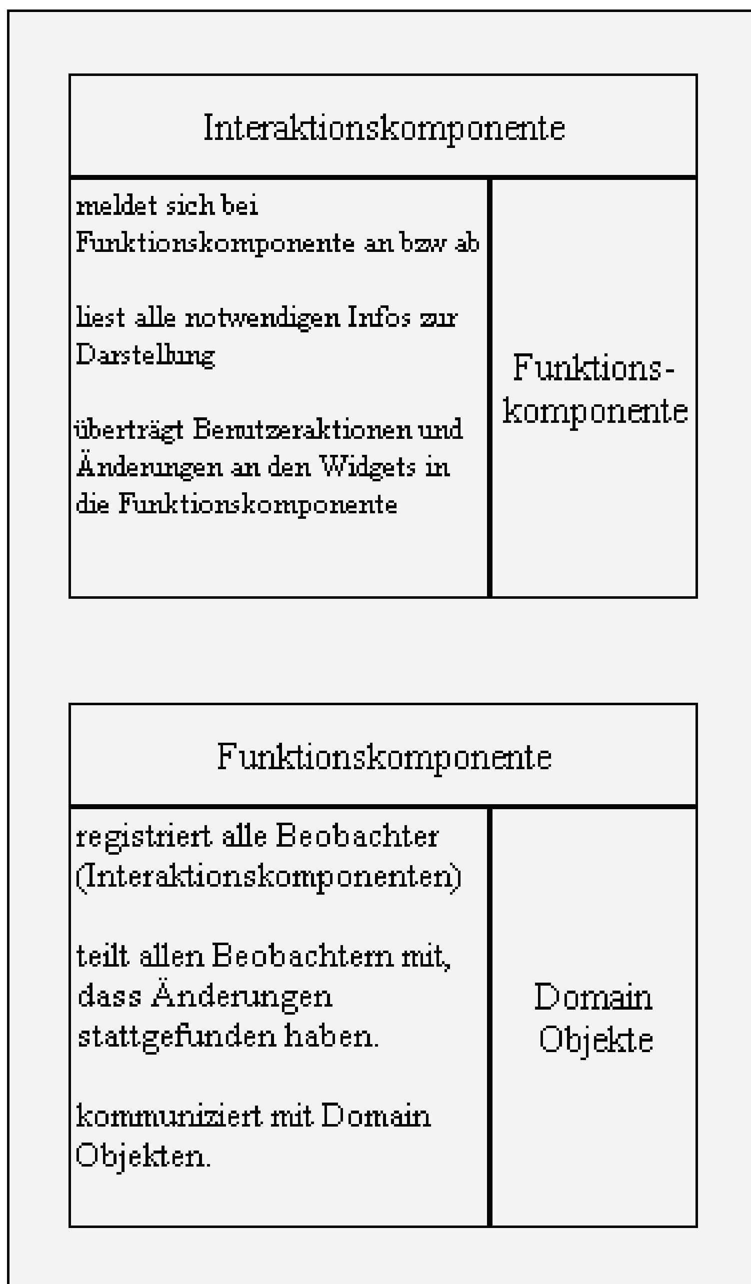


Abb. 4.7: lose Kopplung durch Beobachter Pattern

Eine Erweiterung des Beobachter Patterns:

Logische Ansicht

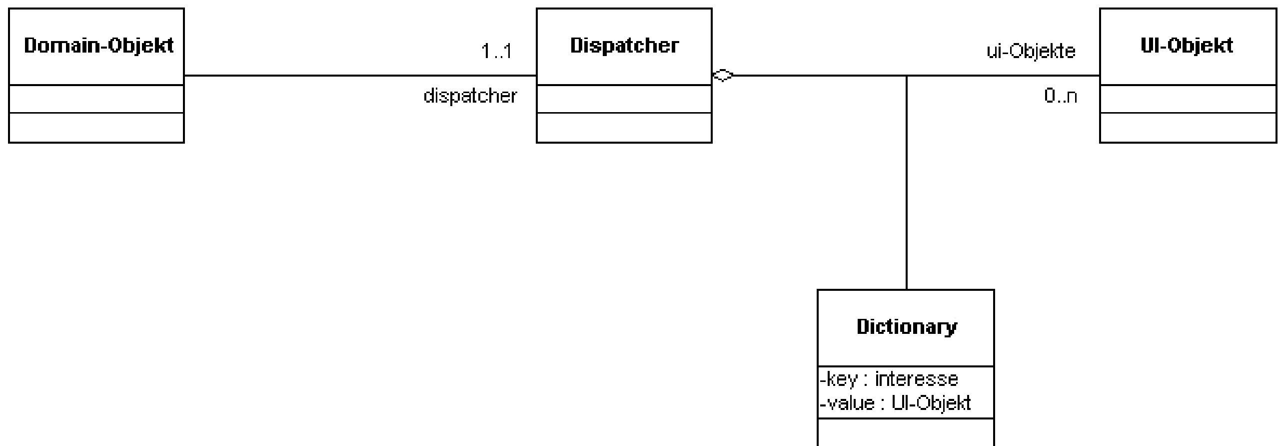
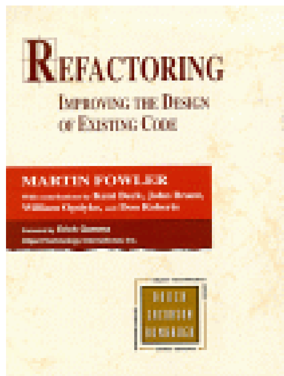


Abb. 4.8: Beobachter Pattern mit Dispatcher

Der Dispatcher sorgt dafür,daß nur dijenigen Objekte nur informiert werden, die an der Änderung im Domain-Objekt auf das UI-Objekt auswirkt. Diese Variante ist besonders dann sinnvoll, falls Domain und UI Schicht auf verschiedenen Rechner verteilt sind.

5 Literatur

Refactoring - Improve the Design of Existing Code

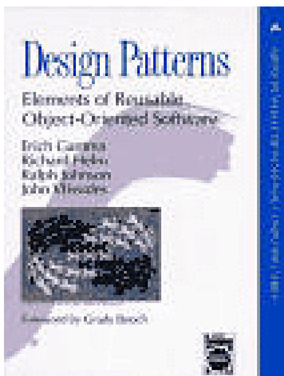


Für mich das wichtigste OO-Buch der letzten Jahre. Das erste Kapitel bietet ein AHA-Erlebnis, über die Bedeutung von Refactoring. Es folgen Kapitel, wie automatische Tests mit Hilfe des JUnits-Frameworks, und eine Katalog von Refactoring-Prozeduren. Alle Beispiele sind in Java. Das Buch ist einfach zu lesen, und orientiert sich fast immer an kleinen und übersichtlichen Beispielen. Die deutsche Ausgabe habe ich nur mal durchgeblättert. Das Layout ist schlechter, zur Übersetzung kann ich nicht sagen. Das Englisch in der Original Ausgabe ist aber so einfach, daß sich die deutsche Ausgabe nur den absolut Englisch-Verweigerer lohnt.

Das objektorientierte Konstruktionshandbuch

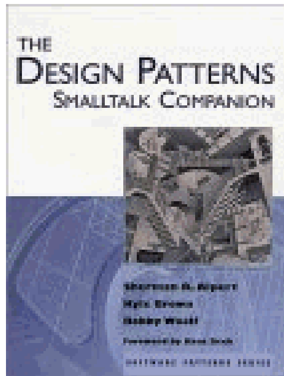
Über dieses Buch freue und ärgere ich mich gleichzeitig. Es ist das einzigste mir bekannte Buch, das eine vollständige gut durchdachte Software-Architektur vorstellt, die ein Rahmen für jegliche OO-Projekte dient. Es wird der Werkzeug- u. Materialansatz vorgestellt, der schon in mehreren groesseren OO-Projekten erfolgreich angewandt worden. Ärgerlich ist, daß die Autoren es nicht verstanden haben, den Inhalt einfacher und verständlicher zu übermitteln. Trotz der Dicke (fast 700 Seiten) ist der Stoff zu knapp beschrieben.

Entwurfsmuster



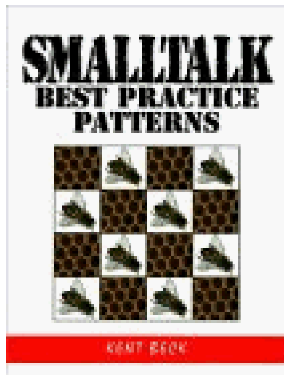
Es ist das Standard-Buch über Patterns, auf das sich jedes Pattern-Buch bezieht. Ich hatte als ich das Buch vor 4 Jahren gekauft hatte, doch grosse Verständnis Schwierigkeiten. Hilfreich waren oft Artikel von Zeitschriften, die einzelne diese Patterns näher betrachteten. Die Beispiele sind in C++.

The Design Patterns Smalltalk Companion



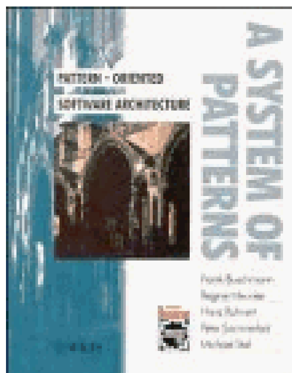
Erklärt die obigen Entwurfsmuster anhand Smalltalk Beispielen. Sehr viel einfacher zu verstehen.

Smalltalk Best Practice Patterns



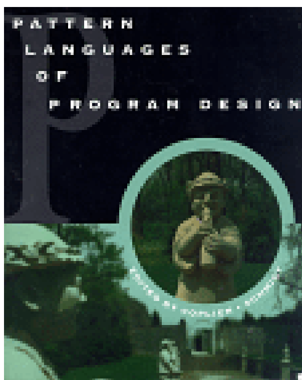
Das Standard-Werk von Kent Beck mit einer Ansammlung von Smalltalk-Idioms. Viele Refaktorierungen haben von diesem Buch ihren Ursprung. Viele dieser Smalltalk-Idioms sind auf Java uebertragbar. Für Smalltalker sollte dieses Buch eine Pflichtlektüre sein.

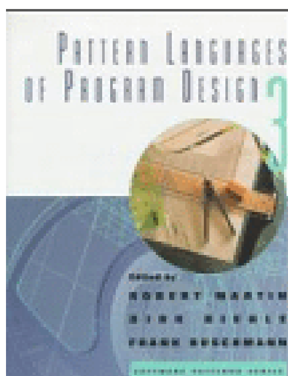
Patternorientierte Systemarchitektur

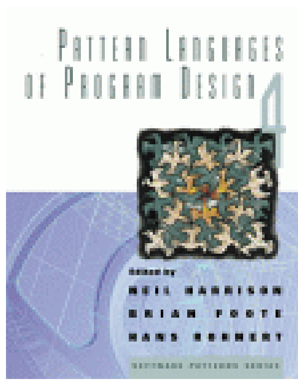


Eine ideale Ergänzung zu den Entwurfsmuster, da sich diese Bücher nur gering überschneiden. Sehr viel verständlicher und einfacher zu lesen als das Entwurfsmuster Buch

Pattern Languages of Program Design Band 1 - 4

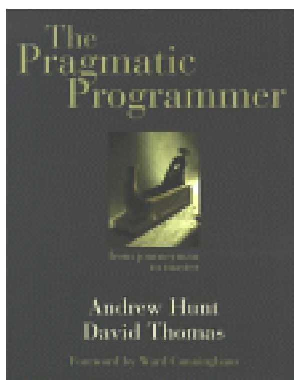






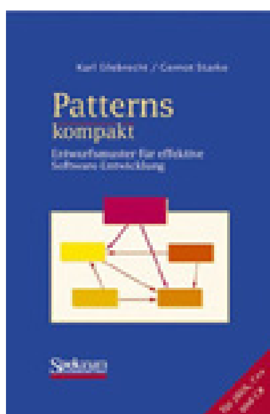
Interessante Sammlung von Patterns aus der regelmaessig stattfindeten PLOP-Konferenzen. Viele Patterns sind sehr speziell aber einige Beiträge sind wahre Juwelen.

The Pragmatic Programmer



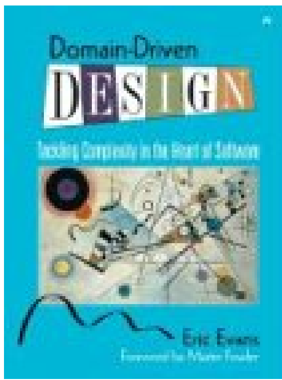
Sehr lesenswertes und leicht geschriebenes Buch. Viele Richtlinien und Tips die ein guter und pragmatischer Programmierer befolgen sollte, um kostengünstig fehlerfreie Software zu entwickeln.

Patterns kompakt



Klein, fein, preiswert. Gibt eine sehr gute und übersichtliche Zusammenfassung der wichtigsten Patterns. Ideal zum Nachschlagen neben dem Schreibtisch.

Domain Driven Design

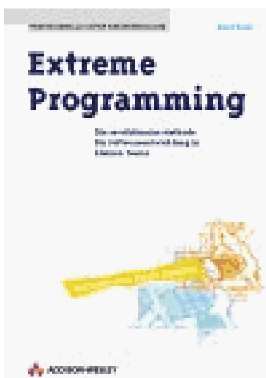


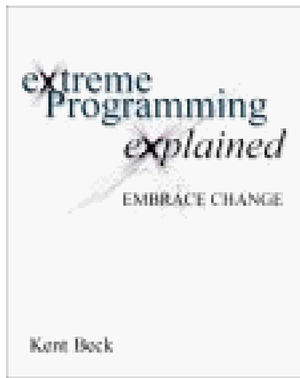
Sicherlich eines der wichtigsten Bücher des Jahres 2003. Das (Geschäfts-) Objekt Modell sollte mit dem fachlichen Modell übereinstimmen, so daß Fachexperten und IT-Experten diesselbe Sprache benutzen. Diese Buch zeigt Techniken und Patterns dieses zu erreichen.

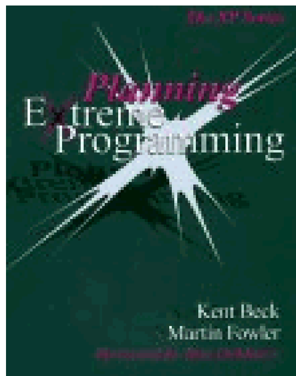
Patterns für Enterprise Architekturen

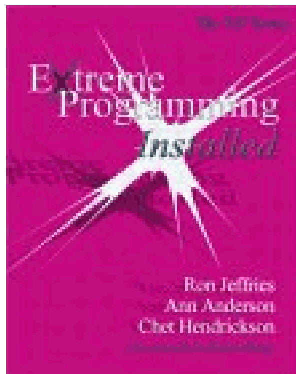
Sammlung wichtiger Patterns die sich u.a. mit Verteilung, Datenbank Mapping, Web Repräsentationen beschäftigen. Sehr empfehlenswert, wie alles was von Martin Fowler kommt.

Extreme Programming Serie



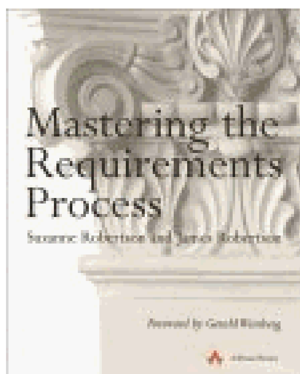






Extreme Programming ist eine leichtgewichtige Methodik zur Softwareerstellung. XP stellt u.a. höchste Anforderungen an die interne Softwarequalität. Dazu gehört andauerndes Refactoring, und automatisierte Tests.

Mastering the requirements process



Suzanne Robertson und James Robertson Das Ehepaar Robertson ist im Gebiet Anforderungsanalyse sicherlich führend. Ich bin von ihrem Buch begeistert. Anforderungen werden auf Karteikärtchen geschrieben und durch einen Qualitätscheck unterworfen. Jede Anforderung muss messbare Akzeptanzkriterien beinhalten. Weiterhin wird ein Template gebeben,in denen Basis sich die Anforderungsdokumente eingliedern. Ich empfehle dieses Buch allen die mit Anforderungsanalyse zu tun haben.