Literatur

- Frank Westphal: *Testgetriebene Entwicklung mit JUnit & FIT*; dpunkt-Verlag.
- Martin Fowler: *Refactoring – oder wie Sie das Design vorhandener Software verbessern –* ; Addison-Wesley.
- Stefan Roock & Martin Lippert: *Refactoring in großen Softwareprojekten*; dpunkt-Verlag.
- Joshua Kerievsky: *Refactoring to Patterns*; Addison-Wesley Signature Series, 2004.

Tests im Softwareentwicklungsprozess

traditionell

test-getrieben

- **Geschäftsprozessmodellierung**
- **Requirements Engineering**
- **Analyse**
- **Design**
- **Implementierung**
- **Tests**

test-getrieben: **Tests vor Implementierung und Design inkrementell**

Test-getriebenes Vorgehen TDD

- **entstammt dem eXtreme Programming (XP):**
  - sehr leichtgewichtiger Prozess
  - iterativ, inkrementell, aber mit extrem kurzen Iterationen
  - Kosten von Änderungen im Projektverlauf ungefähr konstant
  - für kleine und mittlere Projektteamgrößen geeignet
  - dauerhafte Einbindung der Kunden
  - kurze Releasezyklen
  - einfaches Design
  - von Beginn an automatisierte Tests
  - Refactoring: Designänderung ohne Funktionalitätsänderung
  - Pair Programming
  - fortlaufende Integration

Test-getriebenes Vorgehen (Forts.)

- **häufig Mischform:**
  - außen V-Modell: Anforderungen und Grobentwurf
  - innen XP: Rest

- **Test-Driven Development (TDD) – Zyklus**
  - Analyse
  - Testfälle erzeugen
  - Kodierung
  - Test
  - Refactoring

- **Testentwurf -> analytische Fragen**

- **Testimplementierung zuerst -> weniger Fehler im Code**

TDD nach Kent Beck

- **Überlege, was du tun willst**
- **Überlege, wie du es testen kannst**
- **Schreibe kleinen Test -> API des zu testenden Objekts**
- **Schreibe nur soviel Code, dass Test fehlschlägt**
- **Führe Test aus und überprüfe, dass er fehlschlägt**
- **Schreibe nur soviel Code, dass alle Tests erfolgreich**
- **Refactoring bei Code-Duplikaten oder unverständlichem Code**
- **Führe Tests erneut aus**
- **Iteriere bis kein neuer Test mehr gefunden werden kann**

Vorteile von TDD

- **durch programmierte Tests: vorher detaillierte, ausführbare Spezifikation**
- **klar, wann Klasse fertig ist**
- **Fehler früh gefunden**
- **Vermeidung langwieriger Debug-Sessions**
- **Abhängigkeiten zu anderen Objekten früh erkannt und minimierbar**
- **Re-Designs sicher bzw. überhaupt erst möglich**
- **direkter Nutzen für Entwickler -> Motivationssteigerung**
- **besonders deutlich bei Wartung und Erweiterung**
- **fremder Code anhand der Testfälle leichter verständlich**
- **Vor.: Testautomatisierung, hier durch jUnit**

TDD-Bsp. (aus F. Westphal)

- **1. Direktive:**
  Motivieren Sie jede Änderung des Programmverhaltens durch einen automatisierten Test
  - gehe immer von Test aus
  - immer nur kleiner Schritt
  - danach direkt Code erweitern, so dass alle Tests erfüllt

- **Testgetriebener Entwicklungszyklus**

  - **grün -> rot**

  neue Anforderungen durch neuen Test

  - **rot -> grün**

  Test auf möglichst einfache Art durch Codeänderung erfüllen

  - **grün -> grün**

  Code in einfache Form bringen -> Refactoring

Programmierzüge

1. Test schreiben, der fehlschlagen soll

- nicht dran denken, dass manches noch gar nicht da
  -> häufig Compilefehler

2. Gerade soviel Code schreiben, dass Test fehlschlägt

- alles schreiben, dass sich Code übersetzen lässt
- mehr nicht

3. Gerade soviel Code schreiben, dass alle Tests laufen

- mehr nicht

4. Refactoring so, dass alle Tests immer noch laufen

- lernen aus den Tests
- bringen Code in Form, damit erweiterbar

Testplan

- **Bsp.:**
  - Filme kosten regulär für die ersten drei Tage insgesamt 1,50€.
  - Für jeden weiteren Ausleihtag kommen 1,50€ dazu.
- **Testfälle für interessante Entwurfsentscheidungen**
  - 1 Tag
  - 3 Tage
  - 4 Tage
  - 5 Tage
- **0 oder negative Eingaben interessieren nicht**
  - hierfür aufrufende Klassen verantwortlich
- **gehen von Euro-Klasse aus**
  - Objekte nicht veränderbar, wie String

Einstieg

- **neue Testklasse anlegen**

```
public class RegularPriceTest extends TestCase {
}
```

- **und in Test-Suite eintragen**

```
public class AllTests {
   public static Test suite() {
      TestSuite suite = new TestSuite("Test for tdd");
      suite.addTestSuite(EuroTest.class);
      suite.addTestSuite(RegularPriceTest.class);
      return suite;
   }
}
```

1. Testfall

- **kleines Inkrement der Codebasis mit großem Lernerfolg**
- **hier: 1. Tag**

1. Test schreiben, der fehlschlagen soll

```
public void testChargingOneDayRental() {

            RegularPrice price = new RegularPrice();

            assertEquals(new Euro(1.50),price.getCharge(1));

}
```

- **Design-Entscheidungen:**
  - Klasse heißt **`RegularPrice`**
  - Defaultkonstruktor wichtig
  - Ausleihgebühr durch **`getCharge`**
  - Rückgabetyp: **`Euro`**

Schritte 2 und 3

2.Test fehlschlagen lassen

- von Entwicklungsumgebung unterstützen lassen

```
public class RegularPrice {
        public Euro getCharge(int daysRented) {
                return null;
        }
}
```

3.Test erfüllen

- nur das Nötigste programmieren, damit Test erfüllt

```
public Euro getCharge(int daysRented) {
        return new Euro(1.50);
}
```

Zusammenspiel: Tests & Programm

- **fehlgeschlagener Test -> nächster Programmierschritt**
- **bisher geschriebener Code -> nächster Test**
- **Programmierung <-> Design**
  - starkes Zusammenspiel
  - nicht testbar -> Änderung der Klassenschnittstelle
  - immer nur ein Test und sofort programmieren

  -> erkennt, was uns voran bringt
  - spätere Tests auf Karteikarte merken
  - zuviel Code

  -> zuwenig Tests

  -> Fehleranfälligkeit bei weiterem Refactoring

Ausnahme grün -> grün

- **Test sollte fehlschlagen, warum läuft er?**

```
public void testChargingThreeDaysRental() {
    RegularPrice price = new RegularPrice();
    assertEquals(new Euro(1.50),price.getCharge(3));
}
```

  - bringt keine neue Anforderung rein -> Test wertlos
  - 2. Möglichkeit: Test fehlerhaft oder schon vorhanden
  - hier besser: Test für 4 oder 5 Tage
  - bei schwierigen Tests überlegen, ob Design zu kompliziert

Ausnahme rot -> rot

- Test sollte laufen, warum schlägt er fehl?
- Inkrement vielleicht zu groß

```
public void testChargingFourDaysRental() {
        RegularPrice price = new RegularPrice();
        assertEquals(new Euro(3.00),price.getCharge(4));
}               -> expected: <EUR 3.0> but was <EUR 1.5>
```

- rot -> Ändere:

```
public Euro getCharge(int daysRented) {
        Euro result = new Euro(1.50);
        if (daysRented == 4) result.add(new Euro(1.50));
        return result;
}       -> expected: <EUR 3.0> but was <EUR 1.5>
```

Backtracken

- **ändere Methode wie folgt**

```
public Euro getCharge(int daysRented) {
      if (daysRented <= 3) return(new Euro(1.50));
      return new Euro(3.00);
}           ->    grün
```

- **aber konstante Rückgabe -> 1. Versuch weiter**

```
public Euro getCharge(int daysRented) {
      if (daysRented <= 3) return new Euro(1.50);
      return new Euro(1.50).add(new Euro(1.50));
}           ->    grün
```

- **unschön:** **3 x** `new Euro(1.50)`

- **4.Refactoring**

1. Refact.: sinnvolle Bezeichnungen

- **führe drei sinnvolle Bezeichnungen ein**

```
public class RegularPrice {
  private static final Euro BASEPRICE = new Euro(1.50);
  private static final Euro PRICE_PER_DAY = new Euro(1.50);
  private static final int DAYS_DISCOUNTED = 3;
  public Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;
       return BASEPRICE.add(PRICE_PER_DAY);
    }
}
```

- **Rückschritt für den Fortschritt**
  - Schritte klein, darum einfacher wieder zurück zu gehen
  - nutze History der Tools

Ausnahme rot -> ???

- **Test sollte sich relativ schnell erfüllen lassen, wie machen?**
- **manchmal fehlen weitere Methoden**
  -> Test nicht mit einem Schritt erfüllbar
  -> Test vielleicht zu groß

```
public void testChargingFiveDaysRental() {
     RegularPrice price = new RegularPrice();
     assertEquals(new Euro(4.50),price.getCharge(5));
}     -> expected: <EUR 4.50> but was <EUR 3.0>
```

- **Formel reinbringen**

```
public Euro getCharge(int daysRented) {
     if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;
     int additionalDays = daysRented - DAYS_DISCOUNTED;
     return BASEPRICE.add(newEuro(PRICE_PER_DAY.getAmount()
                                        *additionalDays));
}    -> grün
```

2. Refact.: weitere Methoden einführen

- **Refaktorisieren Sie nur mit grünem Balken**
  - zuerst einfache, dirty Lösung
  - sonst oft zu weit
  - in der Nähe des grünen Lichts bleiben
- **Probleme nacheinander lösen, nicht gleichzeitig**
  - divide and conquer
- **führe Mulitiplikation in Euro-Klasse ein, hierzu Test:**

```
public void testMultiplying() {
        Euro two = new Euro(2.00);
        assertEquals(new Euro(14.00),two.times(7));
        assertEquals(new Euro(2.00),two);
}
```

Schritte 2 und 3

- **Test fehlschlagen lassen**
  - von Entwicklungsumgebung unterstützen lassen

```
public Euro times(int factor) {

      return null;

} -> expected: <EUR 14.0> but was <null>
```

- **Test erfüllen**
  - nur das Nötigste programmieren, damit Test erfüllt

```
public Euro times(int factor) {

      return new Euro(this.cents * factor);

}  -> grün
```

Refactoring in getCharge

- **times in getCharge übernehmen**

```
public Euro getCharge(int daysRented) {
      if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      return
      BASEPRICE.add(PRICE_PER_DAY.times(additionalDays));
}
```

-> grün

- **immer möglich:**
  - zuerst Abkürzung
  - dann Refactoring

Zusammenfassung

- **immer nur ein offener Test**
  - Wert des Tests durch rot -> grün
  - sonst Änderungen am Code zu groß
- **optimale Schrittweite finden**
  - Ziel: trockenen Fußes ans andere Ufer zu kommen
- **Refactoring**
- **häufige Integration**

Refactoring

- **2. Direktive**
  Refactoring: Code immer in einfache Form bringen

- **schlecht riechenden Code aufspüren**
  - Duplizierte Logik
  - Große Klassen und lange Methoden
  - schlechte Namen und unpassende Codestrukturen
  - Kommentare: möglichst nur Javadoc, Rest selbstdokum.

- **Refactoring planen**
- **kleine Schritte und häufig testen**
- **Problem: Veränderung der öffentlichen Schnittstelle**

Häufige Integration

- **3. Direktive**
  Code so häufig wie nötig integrieren
- **mindestens einmal am Tag**
- **erhält Feedback der anderen Entwickler**
- **Synchronisation durch**
  - lokaler Stand mit Repository synchronisieren
  - neuen getesteten Build erzeugen
  - aktuellen Stand versionieren

wichtige Refactorings

- **in Martin Fowler: *„Refactoring*"; Addison-Wesley über 70 Refactorings**
- **siehe auch: Katalog auf http://www.refactoring.com/catalog/**
- **hier Auswahl aus Kapiteln:**
  - Methoden zusammenstellen
  - Eigenschaften zwischen Objekten verschieben
  - Daten organisieren
  - Bedingte Ausdrücke vereinfachen
  - Methodenaufrufe vereinfachen
  - Umgang mit Vererbung
  - Große Refactorings

Methoden zusammenstellen

- **Ziel: kurze, durch Bezeichnung erklärte Methoden**

1. Methode extrahieren

- Code-Fragmente in Methode mit aussagekräftigem Namen.

2. Methode integrieren

- Methodenaufrufe durch Methodenrumpf ersetzen.

3. erklärende Konstante einführen

- für komplizierten Ausdruck eine erklärende Konstante einführen.

4. temporäre Variable zerlegen

- anstelle einer temporären Variable `temp`, die mehrmals gesetzt wird, mehrere mit sinnvollen Namen einführen.

5. Methode durch Methodenobjekt ersetzen

- bei langer Methode mit vielen lokalen Variablen

Bsp.: Methode extrahieren

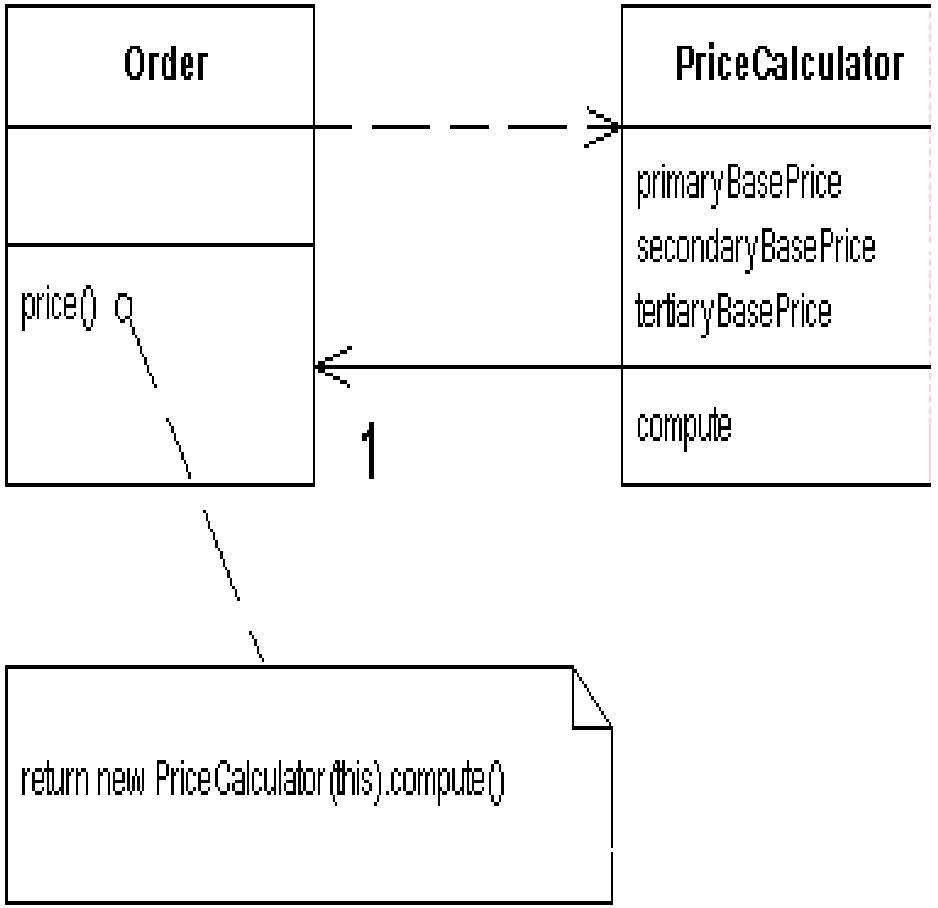
```
void printOwing() {
  printBanner(); //print details
  System.out.println ("name: " + this.name);
  System.out.println ("amount " + getOutstanding()); }

->
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}
void printDetails (double outstanding) {
  System.out.println ("name: " + this.name);
  System.out.println ("amount " + outstanding);
}
```

- **Problem: lokale Variablen und Parameter**
- **Lösung: siehe Übung**

Bsp.: Methode durch MethObj ersetzen

mache aus Methode eigenes Objekt und lokale Variablen und Parameter zu Attributen

```
class Order...
  double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // long computation; ...
  }
```



Eigenschaften zwischen Obj. verschieben

- **Ziel: richtige Verteilung der Verantwortlichkeiten auf Objekte**

6. Methode oder Attribut verschieben

- falls in falscher Klasse

7. Klasse extrahieren oder Klasse integrieren

- falls Klasse zuviel oder zuwenig macht

8. Delegation verbergen oder Vermittler entfernen

- falls Client Klasse aufruft, an die Server delegiert, siehe ORB

9. Fremde Methode einführen

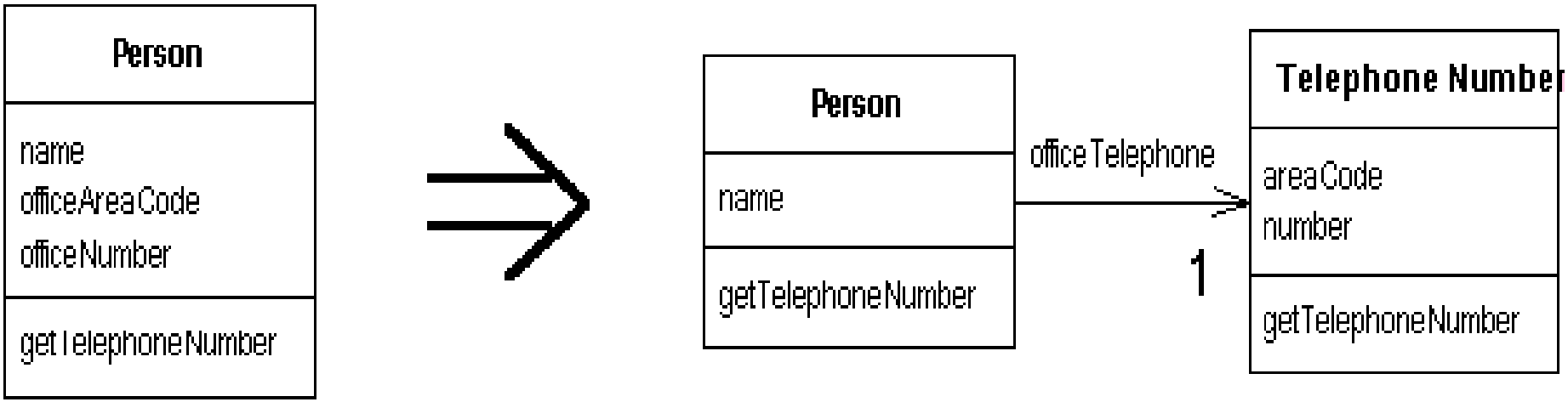
- falls nicht änderbare Serverklasse 1 weitere Methode braucht

10. Lokale Erweiterungen einführen

- falls nicht änderbare Serverklasse >1 weitere Methode braucht

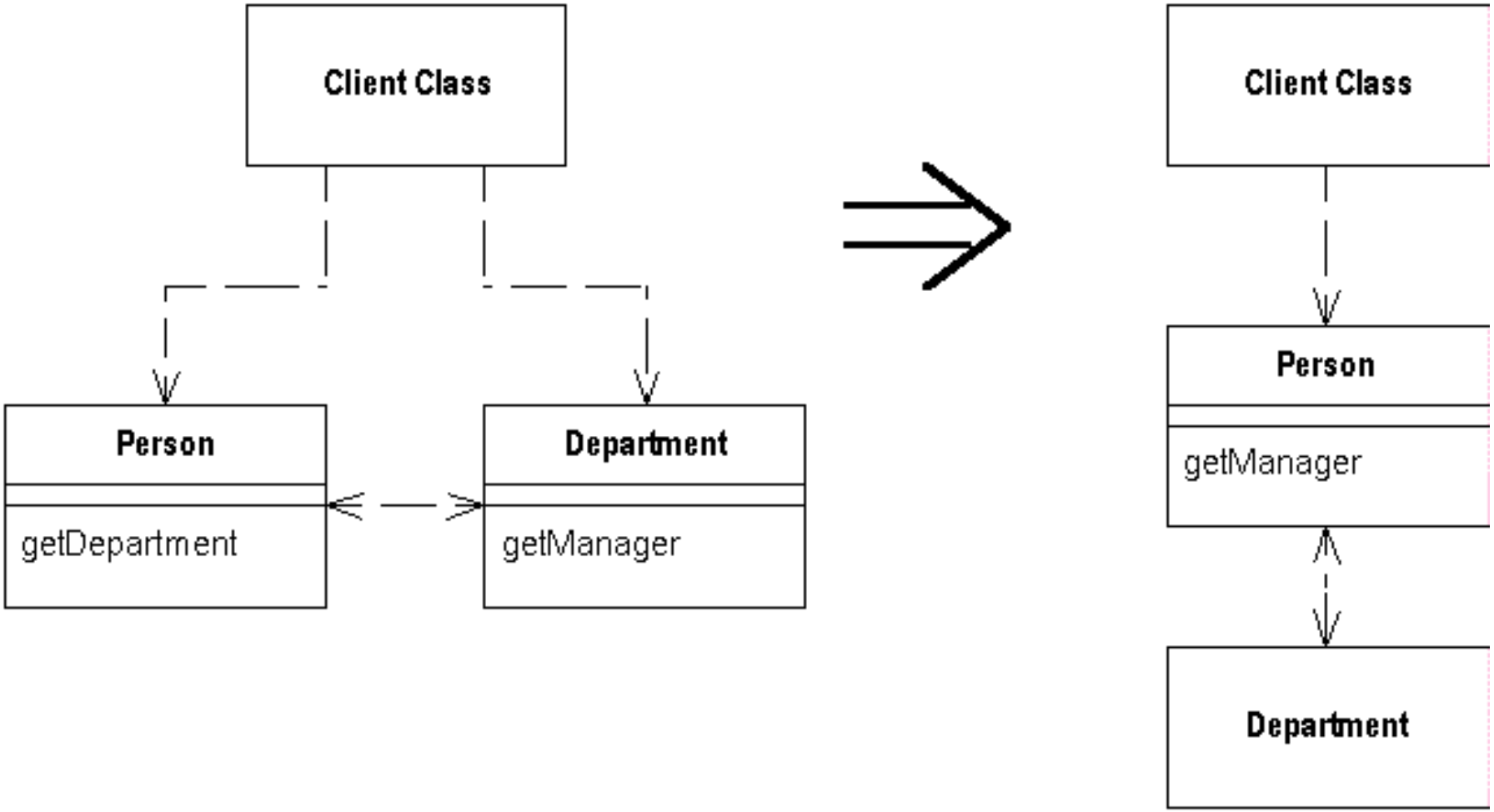
Bsp.: Klasse extrahieren

- **neue Klasse erstellen und relevanten Attribute und Methoden dorthin verschieben**



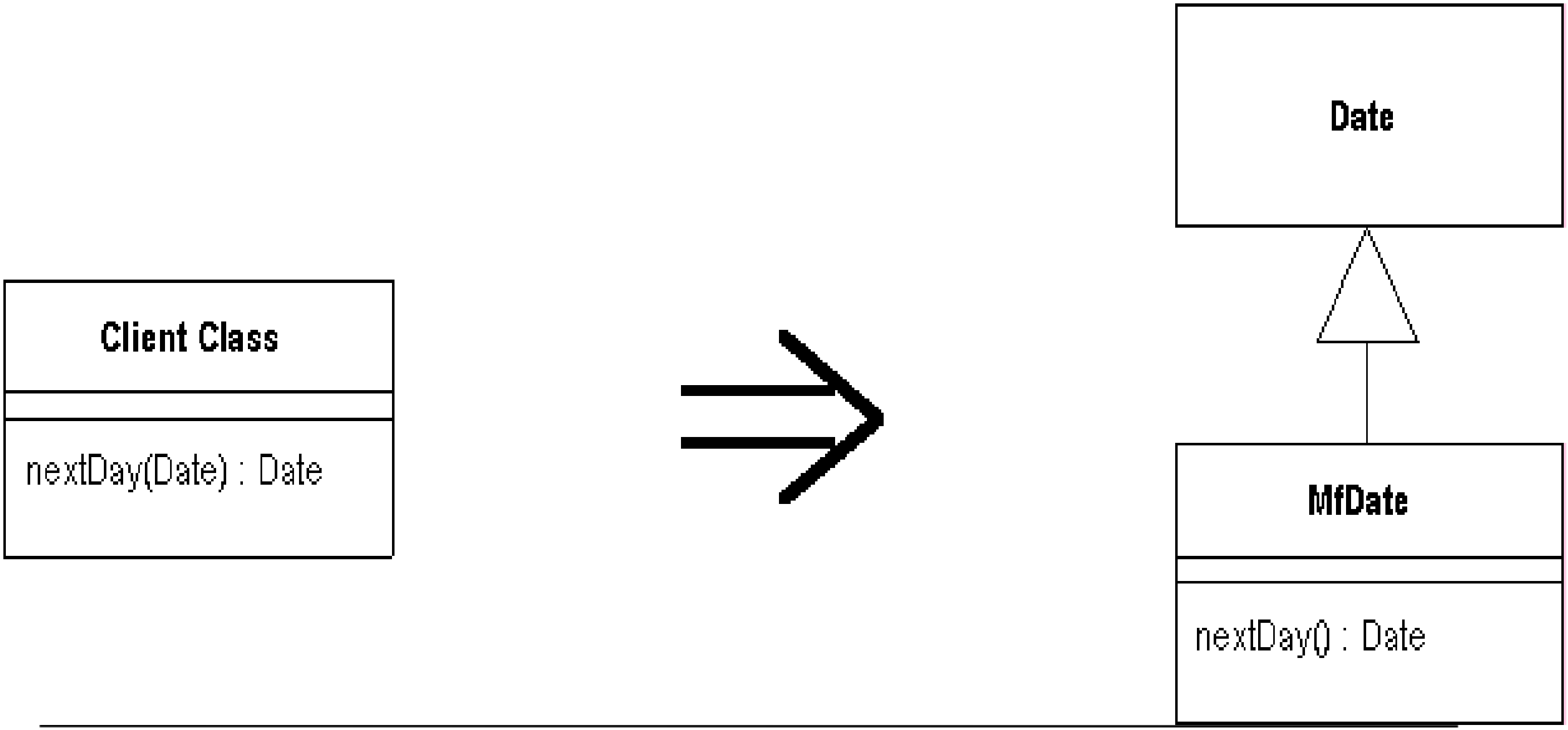
Bsp.: Delegation verbergen

- **Methode des Servers erstellen, um Delegation zu verbergen**



Bsp.: Lokale Erweiterungen einführen

- **neue Klasse erstellen, die zusätzliche Methoden enthält und von Serverklasse erbt**



Daten organisieren

- **Ziel: Vereinfachung des Umgangs mit Daten**

11. Eigenes Attribut kapseln

- Zugriff auf Attribut nur über getter und setter

12. Wert durch Objekt ersetzen

- Datenelement benötigt zusätzliche Daten und Verhalten

13. Array durch Objekt ersetzen

- Einträge des Arrays haben unterschiedliche Bedeutung

14. Gerichtete Assoziation durch bidirektionale ersetzen

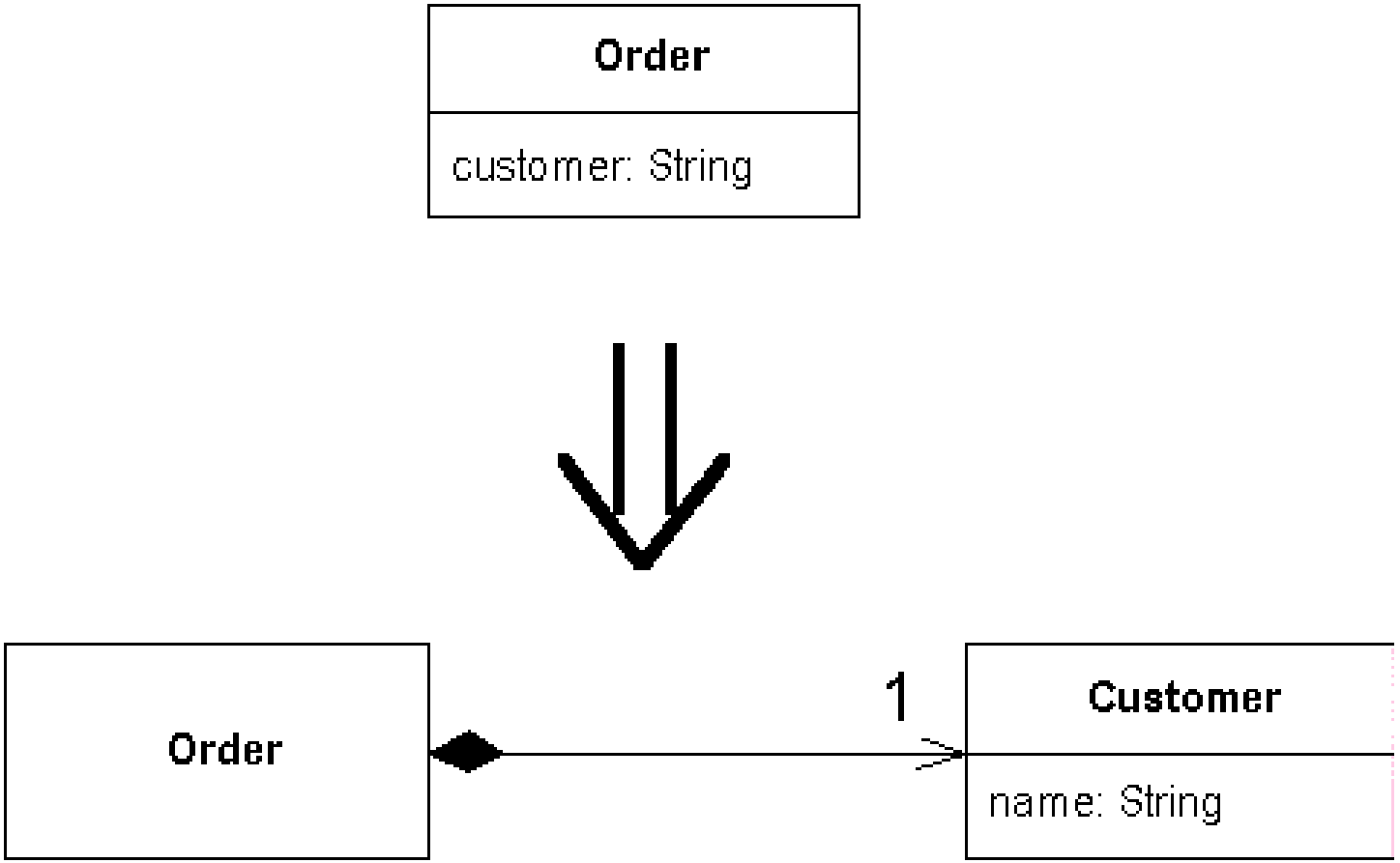
- braucht call-back, aber Assoziation nur in eine Richtung da

15. Bidirektionale Assoziation durch gerichtete ersetzen

- braucht call-back nicht mehr, dann einfachere Ass. nehmen

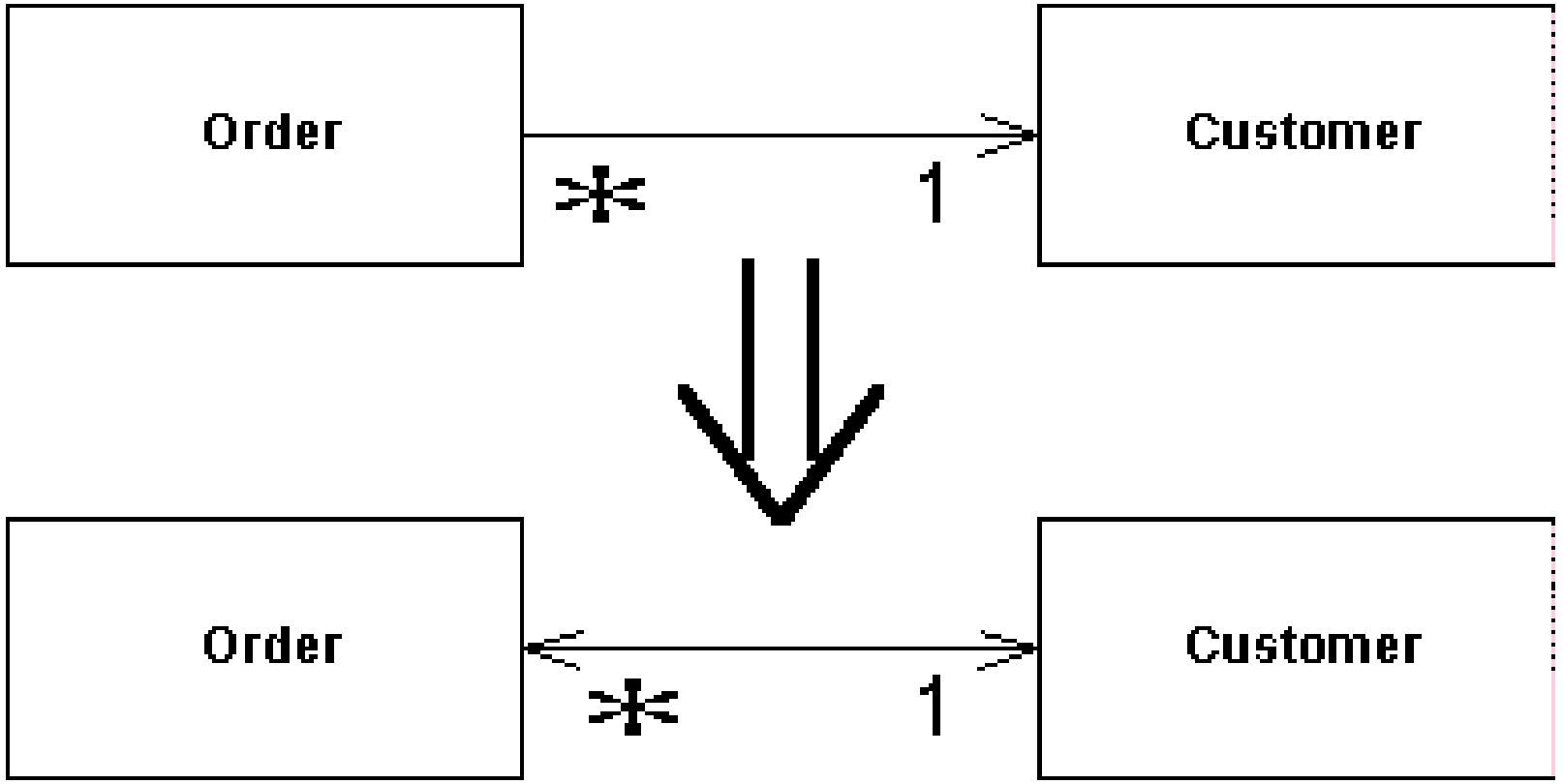
Bsp.: Wert durch Objekt ersetzen

- **Datenelement in Klasse verwandeln und aggregieren**



Bsp.: Gerichtete Ass. -> bidirektionale

- **Rückverweise einfügen und beide Verweise über setter aktualisieren**



Bedingte Ausdrücke vereinfachen

- **Ziel: oft komplexe Bedingungslogik vereinfachen**

16. Bedingung zerlegen

- if_then_else-Teile jeweils Methode extrahieren anwenden

17. Bedingte Ausdrücke konsolidieren

- Folge von Bedingungen mit gleichem Ergebnis kombinieren

18. Redundante Bedingungsteile konsolidieren

- Codeteile, die in jedem Zweig vorhanden, hinter Verzweigung

19. Steuerungsvariable entfernen

- `return/break` statt wilder Steuerungsausdrücke

20. Polymorphismus statt bedingtem Ausdruck

- siehe Einleitung „Flächenberechnung“

Methodenaufrufe vereinfachen

- **Ziel: Schnittstellen verständlich und benutzbar machen**

21. Methode umbenennen

- aus Namen der Methode soll Aufgabe ersichtlich sein

22. Parameter ergänzen oder Parameter entfernen

- wenn Parameter fehlt oder Parameter gar nicht gebraucht

23. Abfrage von Veränderung trennen

- wenn Methode Wert liefert und Zustand ändert, zerlegen in 2

24. Methode parametrisieren

- Methoden machen ähnliches, aber mit verschiedenen Werten

25. Ganzes Objekt übergeben

- bekommt verschiedene Werte eines Objekts als Parameter

Umgang mit der Generalisierung

- **Ziel: richtige Position in Vererbungshierarchie**

26. Attribut oder Methode nach oben verschieben

- zwei Unterklassen haben gleiches Attribut oder Methode

27. Konstruktorrumpf nach oben verschieben

- Konstruktorrümpfe in Unterklassen fast identisch

28. Attribut oder Methode nach unten verschieben

- Methode oder Attribut wird nicht in allen Unterkl. gebraucht

29. Unterklasse extrahieren

- gewisse Elemente werden nur von einigen Instanzen genutzt

30. Schnittstelle extrahieren

- gleicher Teil der Schnittstelle wird verwendet oder 2 Klassen haben Teil ihrer Schnittstelle gemeinsam

Große Refactorings

- **bisher: immer kleine lokale Schritte**
- **jetzt: Blick auf das ganze „Spiel“**

31. Vererbungsstruktur entzerren

- Vererbungshierarchie erledigt 2 Aufgaben
- mache 2 daraus und delegieren von einer in andere

32. Prozedurale Entwürfe in Objekte überführen

- aus Datensätze Objekte machen und Verhalten darauf verteilen

33. Anwendung von Präsentation trennen

- Logik aus GUI-Klassen abspalten in separate Klassen (MVC)

34. Hierarchie extrahieren

- Klasse macht zuviel -> Unterklassen erstellen, die Teil machen

Zitate zum TDD

- **von Sabine Embacher, Brockmann Consult GmbH:**

  „Die für mich prägendste Erfahrung und der Grund, warum ich TDD nicht mehr missen möchte, sind die Veränderungen, die TDD in meinem Leben bewirkt hat. Ich hatte noch nie zuvor so wenig Angst, bestehenden Code erneut anzufassen, zu erweitern, zu verändern, zu refaktorisieren. Ich ging nie zuvor nach getaner Arbeit so entspannt und zufrieden nach Hause. Mir fiel es noch nie so leicht, nach der Arbeit abzuschalten."

- **von Michael Feathers, Object Mentor:**

  „Unser Ziel ist es, einen Fluss mit trockenen Schuhen zu überqueren. Sie können von Stein zu Stein gehen oder zu den weiter entfernten springen. Wenn wir das andere Ufer erreicht haben, können wir alle Schuhe vergleichen."