Refactoring – Transformationen

Martin Freund
bbl@upb.de
Januar 2003
Seminar
“Refactoring in eXtreme Programming”
AG Kastens
Universität Paderborn

Gliederung

1. Transformationen-Übersicht
2. Beispiel: *Methode verschieben*
3. Beispiel: *Bedingte Anweisung durch Polymorphie ersetzen*
4. Erfahrungen mit Refactoring

Behandelte Transformationen

Kap. 6 - Methodenkomposition

Kap. 7 - Verschieben zwischen Klassen

Kap. 8 - Daten organisieren

Kap. 9 - Bedingte Anweisungen vereinfachen

Kap. 10 - Methodenaufrufe vereinfachen

Kap. 11 - Umgang mit Generalisierung

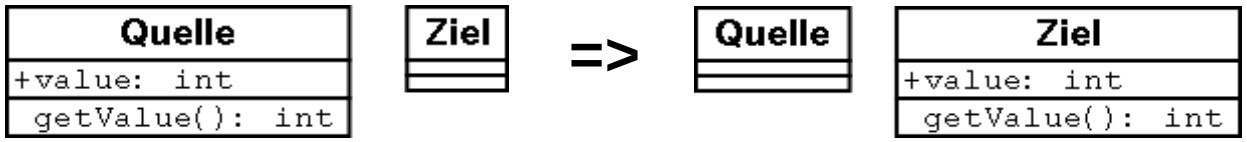
Kap. 12 - Große Refactoring-Transformationen

Quelle: Martin Fowler, et al: Refactoring, Improving the Design of existing Code

Kap. 7: Verschieben zwischen Klassen

- Ziele
  - Variablen und Methoden in gemeinsamen Klassen zusammenführen
  - Aufteilen/Zusammenführen zu großer/kleiner Klassen
  - Aufgaben in andere Klassen verschieben
- 8 Transformationen
  - Elemente (Variablen, Methoden) zwischen Klassen verschieben
  - Klassen extrahieren/einsetzen
  - Delegation herstellen/entfernen
  - Spezialfall: Nicht modifizierbare Bibliotheken in eigenen Unterklassen erweitern

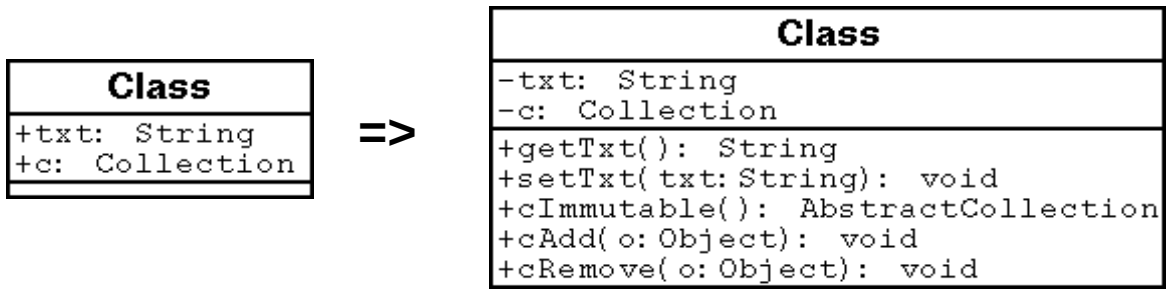
Aufgaben in andere Klassen verschieben: *Methode/Variable verschieben*



Kap. 8: Daten organisieren

- Ziele
  - Datenkapselung
  - Objektverhalten in Klassenhierarchie modellieren
- 16 Transformationen
  - Zugriffskontrollen für einfache Variablen, Collections und Arrays einführen
  - Einführen von Datenobjekten und Referenzen
  - Zwischen injektiver und bijektiver Klassenassoziation wechseln
  - Typvariablen durch Klassenstruktur (Polymorphie) ersetzen
  - Spezialfälle
    - Observer-Pattern einführen für redundante Datensätze (Model+View)
    - Werte mit speziellen Bedeutungen durch Konstanten ersetzen

Datenkapselung: *Variable/Collection kapseln*



Kap. 9: Bedingte Anweisungen vereinfachen

- Ziele
  - Lesbarkeit erhöhen
  - Redundanz vermindern
  - Objektverhalten in Klassenhierarchie modellieren
- 8 Transformationen
  - Bedingungs-/Anweisungsteile in eigene Methoden verlegen
  - Mehrfach vorhandene Anweisungen zusammenführen
  - Identische Rückgabewerte zusammenführen
  - Bedingte Anweisung durch Polymorphie ersetzen
  - Spezialfall: Implizite Annahmen durch `Assertion` ersetzen

Redundanz vermindern: *Doppelte Codefragmente konsolidieren*

```
if (isSpecialDeal) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
```

=>

```
if (isSpecialDeal) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Methode verschieben

- Szenario: Methode ist nicht in Klasse definiert, deren Funktionalität sie nutzt
- Besser: Methode in benutzte Klasse verschieben
- Vorteil: Daten und Operationen werden zusammgeführt
- Elementare Technik für das Verschieben zwischen Klassen
- Oft Teil größerer Refactoring-Transformationen

Szenario für Transformation: *Methode verschieben*

```
class Quelle {
  Ziel lnk = new Ziel();
  public int getValue() {
   return lnk.value;
  }
}
```

```
class Ziel {
  public int value;
}
```

Vorgehen am Beispiel

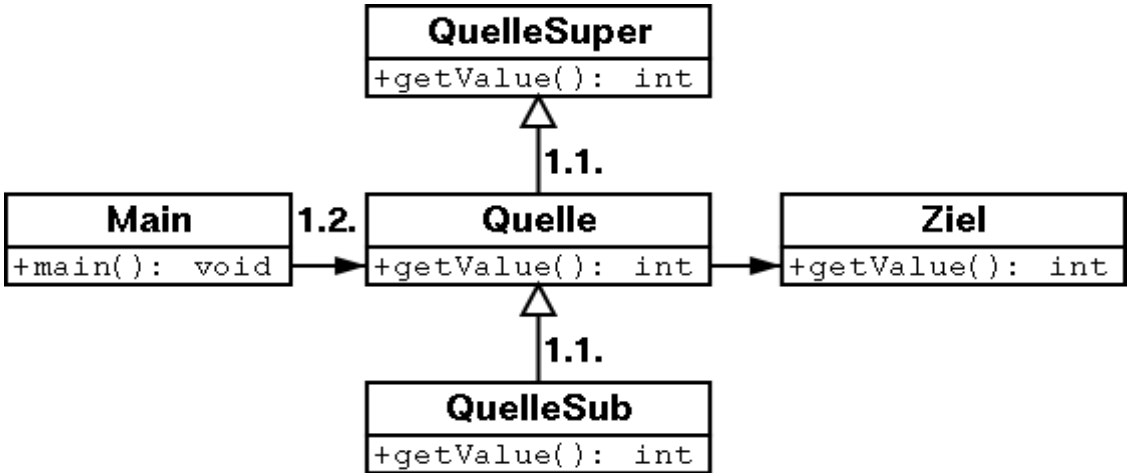
1. Methode in der Zielklasse deklarieren
2. Funktionalität kopieren und anpassen
3. Zielklasse compilieren
4. Zielklasse in der Quellklasse referenzieren
5. Methode in der Quellklasse in Delegationsmethode wandeln
6. Compilieren und Testen

```
class Quelle {
  Ziel lnk = new Ziel();
  public int getValue() {
    return lnk.getValue();
  }
}
```

```
class Ziel {
  public int value;
  public int getValue() {
     return value;
  }
}
```

Randbedingungen

1. Erhalten der Delegationsmethode wenn
   1. Methode in Ober- oder Unterklasse überschrieben wird
   2. Aufrufende Objekte referenzieren kein Objekt der Zielklasse
   3. Zielklasse benötigte Elemente nicht referenzieren kann
- Benötigte Elemente
  - Ebenfalls verschieben wenn möglich
  - sonst als Parameter übergeben

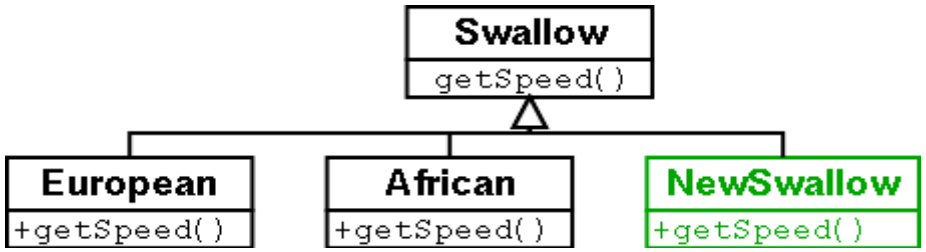


Bedingte Anweisung durch Polymorphie ersetzen

- Szenario: Typcode beeinflusst Verhalten der Objekte einer Klasse zur Laufzeit
  Schlechter Stil in OO-Sprachen
- Besser: Unterschiedliches Verhalten in Unterklassen implementieren
- Vorteile:
  - Neue Typen erfordern nur neue Unterklassen
  - Redundante bedingte Anweisungen werden entfernt

```
class Swallow {
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return BASE_SPEED;
       case AFRICAN:
        return BASE_SPEED +
            A_OFFSET;
    }
  }
}
```

=>



Vorgehen am Beispiel (1/3)

1. Vererbungshierarchie erstellen (Typcode durch Unterklassen ersetzen)
2. Implementieren der zu überschreibenden Methode in Unterklassen entsprechend bedingter Anweisung
3. Compilieren und Testen der Unterklassen
4. Entfernen des Anweisungsteils aus der Originalmethode

```
class Swallow {
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return BASE_SPEED;
      case AFRICAN:
        return BASE_SPEED +
          A_OFFSET;
    }
  }
}
```

```
class European implements
    Swallow {
  double getSpeed() {
    return BASE_SPEED;
  }
}
```

Bsp (2/3): Bedingte Anweisung entfernen

5. Wiederholen, bis bedingte Anweisung ganz entfernt werden kann

```
class Swallow {
  double getSpeed() {
    switch (type) {
      case AFRICAN:
        return BASE_SPEED +
          A_OFFSET;
    }
  }
}
```

```
class African implements
    Swallow {
  double getSpeed() {
    return BASE_SPEED +
        A_OFFSET;
  }
}
```

Bsp (3/3): Polymorphie sicherstellen und benutzen

6. Methode in Oberklasse *abstract* deklarieren
7. Compilieren und Testen
8. An Instanzierungsstellen Unterklassen verwenden

Instanzierung der Oberklasse verhindern:

```
abstract class Swallow {
  abstract double getSpeed();
}
```

Instanzierungsstellen ändern:

```
Swallow s = new Swallow(Swallow.TYPE_EUROPEAN);
```

=>

```
Swallow s = new European();
```

Randbedingungen

- Bedingte Anweisung muß
  - in eigener Methode stehen (sonst *Methode extrahieren)*
  - in oberster Klasse der Hierarchie implementiert sein (sonst *Methode verschieben*)
- Statt Polymorphie State/Strategy Muster benutzen wenn
  - Typ eines Objektes zur Laufzeit geändert wird
  - bereits Unterklassen vorhanden sind

  (*Typcode durch State/Strategy Pattern ersetzen*)

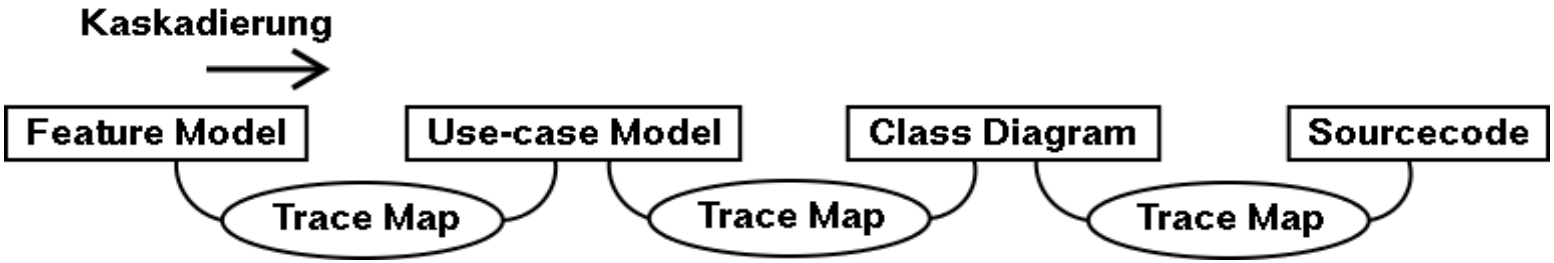
Erfahrungen mit Refactoring

- Praxisberichte: Viele positive Meinungen zu Refactoring
- Keine Daten über Auswirkung von Refactoring
- Ursachen:
  - Vorhandene Daten entstammen der Untersuchung von Softwareprozessen
  - Untersuchung reiner Refactoringanwendung steht noch aus
- Exemplarisch:
  - Refactoring auf mehreren Modellierungsebenen
  - Refactoring in der Lehre

Know-It-All Project
Concordia University

- Ziel: Methoden für Framework-Evolution entwickeln
- Design: Konsequente Nutzung mehrerer Modellierungsebenen
- Verbindung von Modellierungsebenen mittels Trace Maps
- Geplanter Einsatz von Refactoring:
  - Refactoring-Transformationen auf höheren Modellierungsebenen entwickeln
  - Refactoring durch Modellierungsebenen kaskadieren
- Ausstehend:
  - Formalismus für Trace Maps
  - Refactoring-Transformationen für höhere Modellierungsebenen

Pseudonotation: Kaskadierendes Refactoring



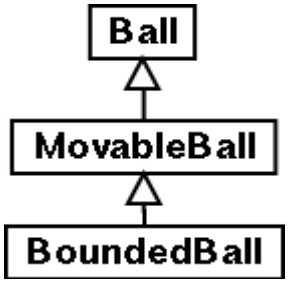
Refactoring in der Lehre

Einsatzerfahrungen mit Extreme Programming in Software-Entwicklungskursen

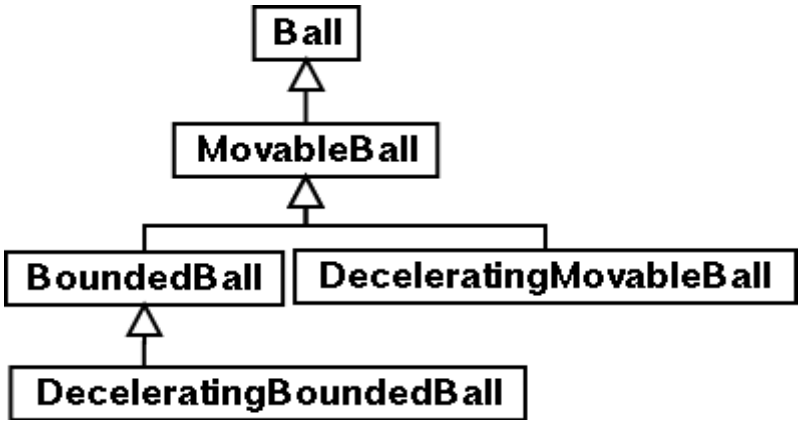
Duke University, Durham NC, USA:

1. Vorgegebene Aufgaben in kleinen Schritten bearbeitet
2. Programmentwurf durch Studierende
3. Zulassen auch fehlerhafter Entwürfe
4. Vorstellen alternativer Entwürfe entsprechend Entwicklungsmustern
5. Refactoring der fehlerhaften Programme

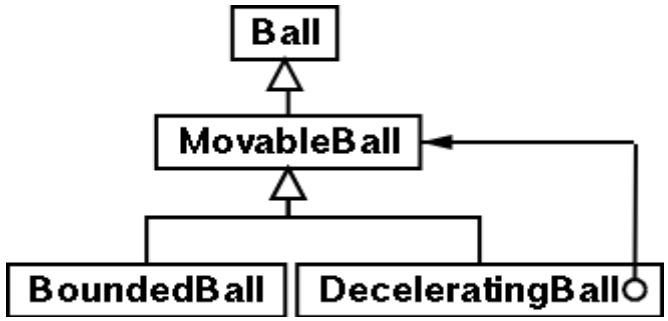
=>Anschließend eingeführte Entwicklungsmuster werden besser aufgenommen



1. Aufgabe:Abbremsenden Ball einfügen



2. Entwurf von Studierenden



4. Refactoringziel: Entwurf mit *Decorator* Muster

Zusammenfassung

- Refactoring-Transformationen für
  - Verschieben zwischen Klassen
  - Daten organisieren
  - Vereinfachen bedingter Ausdrücke
- Durchführung der Transformationen
  - *Methode verschieben*
  - *Bedingten Ausdruck durch Polymorphie ersetzen*
- Refactoring verbreitet sich als Teil neuer Softwareprozesse
- Noch keine Untersuchungen der Wirkung von Refactoring

Danke
für die
Aufmerksamkeit!