efactoring

Programme für die Ewigkeit vorbereiten

DHBW

Was ist Refactoring (I)?

- Existierender (und funktionierender) Code wird nochmals durchgegangen
  - Code Review
- Die Intention und Funktion des Codes wird nachvollzogen
  - Whitebox-Analyse
- Die Testabdeckung wird überprüft
  - Der nächste Schritt erfordert ein gutes Sicherheitsnetz

DHBW

Was ist Refactoring (II)?

- Der Code wird umgestaltet
  - Gesamtverhalten bleibt erhalten
  - Schnittstellen ändern sich nicht
  - Neue Erkenntnisse werden eingearbeitet
- Ziel der Umgestaltung: Bessere Codequalität
  - Bessere Lesbarkeit
  - Höhere Flexibilität (falls notwendig)
  - Klarere Strukturen

Begriffsdefinition

- Refactoring, das:
  - Eine Änderung an den Internas einer Software, um diese verständlicher und änderbarer zu gestalten, ohne ihr (sichtbares) Verhalten zu ändern
- refactorisieren (Verb):
  - Eine Abfolge von Refactorings auf eine Software anwenden, um deren innere Qualität zu erhöhen

Was wir vorfinden

- „Zielorientiert“ programmierter Code
  - Rudimentäres Design
  - Umständliche Strukturen
  - Nachträglich eingefügte Erweiterungen
    - Sonderfall-Prüfungen
  - Kryptische Namen
- Keine Absicherung über automatische Tests
  - Tests sind immer noch nicht Marktstandard
  - Brauchbare Testabdeckung nachholen

Warum sollte ich refactorisieren (I)?

- Der Sourcecode wird verständlicher
  - Beim Schreiben des Codes liegt der Fokus auf der Funktionalität
  - Bei der Review bzw. dem Refactoring liegt der Fokus auf Lesbarkeit und Wiederverwendung
- Der Sourcecode wird wartbarer
  - Umarbeiten des Codes arbeitet „höhere“ Strukturen heraus
  - Höhere Strukturen sparen Code, daher Straffung des Codes auf wesentliche Elemente

Warum sollte ich refactorisieren (II)?

- Das Design der Software wird besser
  - Mit den „höheren Strukturen“ kann das initiale Design neu bewertet werden
  - Der zweite Versuch ist praktisch immer besser als der erste
- Fehler im Code werden öfter gefunden
  - Der Code funktioniert bereits (war das Ziel der Erstimplementierung)
  - Beim erneuten Durcharbeiten werden Spezialfälle etc. häufiger bedacht

Warum sollte ich refactorisieren (III)?

- Neuer Code kann schneller entwickelt werden
  - Zu Beginn eines Projekts bestimmt die Codegüte nicht die Entwicklungsgeschwindigkeit
  - Ab einer gewissen Größe ist die Codegüte der entscheidende Faktor für die Geschwindigkeit
    - Entwicklung findet nicht mehr im „luftleeren Raum“ statt
  - Neuer Code reduziert dann meistens die Codegüte
  - Refactoring hebt die Codegüte immer wieder an
    - Teilweise schneller als „luftleerer Raum“, weil die vorhandenen Strukturen unterstützend wirken

Ein einfacher Entwicklungsprozess

- Programmierung der einfachsten Funktionalität
  - Grundlegendes Design („ad-hoc“)
  - Kein Tuning
  - **„make it“**
- Testen des vorhandenen Codes
  - **„make it run“**
- Verbessern des vorhandenen Codes durch Refactorings
  - **„make it better“**

DHBW

Codebeispiel: „make it“

- Aufgabe:
  - Schreibe einen Prozess, der in einem Verzeichnis-baum alle Dateien löscht, die älter als fünf Tage sind

```
public class CleanupProcess {
      public void cleanup(final File directory) {
            File[] files = directory.listFiles();
            for (File file : files) {
                  if (file.isDirectory()) {
                        cleanup(file);
                        continue;
                  }
                  if (file.lastModified() < (System.currentTimeMillis() - 432000000L)) {
                        file.delete();
                  }
            }
      }
}
```

Codebeispiel: „make it run“

- Bug:
  - Bei jedem leeren Verzeichnis fliegen wir mit einer NullPointerException aus der Methode

```
public class CleanupProcess {
      public void cleanup(final File directory) {
            File[] files = directory.listFiles();
            if (null == files) {
                  return;
            }
            for (File file : files) {
                  if (file.isDirectory()) {
                        cleanup(file);
                        continue;
                  }
                  if (file.lastModified() < (System.currentTimeMillis() - 432000000L)) {
                        file.delete();
                  }
            }
      }
}
```

Codebeispiel: „make it better“

- Verbesserung:
  - Forme den Code durch Refactorings so lange um, bis er genau sagt, was du ihn sagen lassen willst

```
public class CleanupProcess {
      public void cleanup(final File directory) throws IOException {
            ForeachFile.in(directory).perform(
                        DeleteIf.olderThan(5).days());
      }
}
```

Wann sollte ich refactorisieren?

- Einfache Merkregel:
  - „Three strikes and you refactor“
- Erklärung
  - Beim ersten Programmieren einer Funktionalität einfach neu erschaffen
  - Beim zweiten Programmieren einer ähnlichen Funktionalität mit schlechtem Gewissen abschreiben und anpassen (Copy&Paste)
  - Beim dritten Programmieren einer ähnlichen Funktionalität setzt man sich den Refactoring-Hut auf

Wann sollte ich auf jeden Fall refactorisieren?

- Bei einer Code Review
  - aktives Review (abgesichert durch Tests)
- Vor dem Hinzufügen einer neuen Funktionalität
  - Vorhandenen Code so umgestalten, dass der Einbau leichter fällt
- Beim Beseitigen eines Fehlers (Bugs)
  - Testabdeckung des Codes muss sowieso erhöht werden
  - Softwarefehler unterliegen dem Lokalitätsprinzip

Exkurs: Das Lokalitätsprinzip für Softwarefehler

- Wo ein Bug ist, da sind noch andere
- Warum Bugs im Code „klumpen“:
  - Komplexe Bereiche enthalten mehr Bugs
  - Fehler werden teilweise durch Gegenfehler kompensiert (Symptombehebung)
  - Zusammenhängender Code wurde meistens vom gleichen Programmierer geschrieben
  - Zusammenhängender Code wurde meistens zeitnah geschrieben
    - Programmierer mit schlechtem Tag

Warum funktioniert dieses Refactorisieren?

- Frühere, jetzt unpassende Entscheidungen werden korrigiert
- Programme sind schwieriger zu lesen als zu schreiben
  - Schwer lesbare Programme sind schlecht änderbar
  - Komplexe Programme sind schlecht änderbar
- Code-Verbesserung hat Langzeit-Effekt
  - Mittlere Entwicklungsgeschwindigkeit, aber dauerhaft

Wie erzähle ich das meinem Chef?

- Gar nicht!
  - Im besten Fall versteht er den Nutzen sofort
  - Im schlechtesten Fall sieht er einen Konflikt zwischen qualitätsbewusster Arbeitsweise und einem Termin/Budget
    - Entwickler müssen verantwortungsbewusst sein
- Als persönlichen Arbeitsstil ausgeben
  - Der Entwickler hat die Aufgabe, fehlerfrei funktionierende Software so schnell wie möglich herzustellen
    - Betonung auf „funktionierend“, nicht „schnell“

Pause

DHBW

Wo funktioniert refactorisieren nicht so prima?

- Zentrale Designänderungen
  - Stillstand des gesamten Projektteams (oder hoher Merge-Aufwand)
  - Gefahr von Fehlern trotz Testabsicherung
- Öffentliche Schnittstellen
  - Abhilfe: alte und neue Schnittstelle parallel anbieten
- Datenbank-Schema
  - Zusätzlich Migration bereits vorhandener Daten
  - Milderung: Explizite Zugriffsschicht auf Datenbank

Welche negativen Auswirkungen haben Refactorings?

- Zeitverbrauch
  - Auch das schnellste Refactoring benötigt Zeit
  - Kompensation: Danach schnellere Entwicklung
- Programmfehler
  - Selbst mit Tests können bei Änderungen Fehler gemacht werden
  - Kompensation: Positive Effekte der Code Review
- Performanceminderung
  - Manche Refactorings machen den Code langsamer

Exkurs: Code Tuning

- „premature optimization is the root of all evil“
  - Donald Knuth
- Verwendung des 90/10-Effekt
  - 90% der Zeit wird in 10% des Codes verbraucht
- Drei Regeln für Tuning:
  - Don't („Mach es nicht“)
  - Not yet („Verschieb es auf später“)
  - Measure (Schwachstellenanalyse durch Messen)

Code Smells

- „Code Smell“ ist die Bezeichnung für eine verbesserungswürdige Codestelle
  - „If it stinks, change it“
- Einstiegsstellen für Refactorings
  - Konkrete Missstände mit konkreten Lösungen
- Keine harten Messwerte
  - Aus Erfahrung wiedererkennbare Problemstellen
  - Teilweise durch Algorithmen oder Heuristiken lokalisierbar

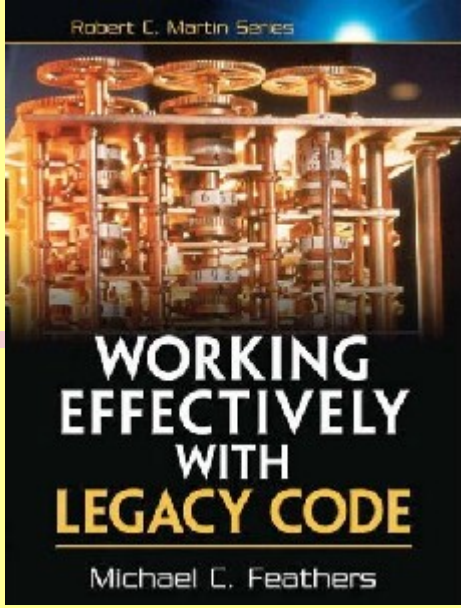
Code Smell: Duplicated Code

- Doppelt vorhandener Code
- Wichtigster, weil gravierendster Problemfall
  - Sehr leicht zu beseitigen
    - Doppelte Stellen zusammenführen
- Änderungen an einer Stelle bewirken keine vollständige Programmänderung
  - Doppelter Pflegeaufwand
  - Mit der Zeit laufen die Stellen auseinander
- Eventuell Anzeichen für fehlende Strukturen

Code Smell: Long Method

- Lange Methoden
- „Objektorientierter Code lebt länger, wenn die Methoden kurz sind“
  - Für Anfänger sieht es so aus, als ob nirgendwo wirklich etwas passiert (überall nur Delegation)
- Abhilfe: Methode aufspalten
  - Gute Benennung der Methoden führt zu besserer Lesbarkeit
  - Beim Aufspalten an Kommentaren orientieren
    - Konditionalstrukturen und Schleifen sind ebenfalls gute „Nahtstellen“

Exkurs: Code Seams



- Deutsch: Nahtstelle, Saum
- Bei Textilien: Verbundstelle zweier Textilstoffteile
  - Kann wieder aufgetrennt werden
  - Deutliche Beispiele: Patchwork, Aufnäher
- Im Programmcode: Trennstelle zweier Codeteile
  - Fest verbundener Code kann geteilt werden
  - Weiterer Code kann „dazwischengefügt“ werden
- Beispiel: Polymorphie, Methodenaufruf

DHBW

Code Smell: Large Class

- Schwergewichtige Klasse
- Häufigste Symptome:
  - Zu viele Instanzvariablen
  - Zu viele Methoden
  - Stark erhöhte Zeilenanzahl
- Meistens Verletzung von OO-Prinzipien
  - Beispielsweise SRP, Low Coupling, High Cohesion
- Einfachste Abhilfe:
  - Teilfunktionalität in neue Klasse(n) auslagern

Code Smell: Shotgun Surgery

- Flickenteppich-Änderung
  - Wörtlich: Schrotschuss-Chirurgie
- Eine (fachlich motivierte) Änderung erfordert die Modifikation von vielen verschiedenen Codestellen
  - Evtl. sogar in gleichartiger Weise → Duplikation
- Falls eine Modifikation vergessen wird, existiert ein neuer Bug im System.
- Abhilfe erfordert Umstrukturierung des Codes

Code Smell: Switch Statement

- Switch-Statements haben inhärente Probleme
  - Duplikation: Oft gleicher Switch an verschiedenen Stellen
  - Komplexität: Der Switch ist nicht aufteilbar, er kann nur „im Platz“ wachsen
  - Fehleranfällige Syntax: Break vs. Fall-through
- Ausrollen auf if/else-Kaskade beseitigt nicht alle Probleme
- Polymorphie ersetzt (u.a.) diese Konditionalstruktur meistens sehr elegant

Code Smell: Code Comments

- Inline-Kommentare
- Kommentare im Code sind meistens kein eigener Smell, sondern „Deodorant“ für einen anderen Smell
  - Beispiel: Erläutert der Kommentar einen eigenständigen Code-Block → Long Method
- Meistens ist der Kommentar nach einem Refactoring überflüssig, weil unnötig
- Inline-Kommentare können auch sinnvoll sein
  - Dokumentation von Annahmen, Unsicherheit, etc.

Pause

DHBW

Refactorings im Beispiel

- Extract Method
- Rename Method
- Replace Temp with Query
- Replace Conditional with Polymorphism
- Replace Error Code with Exception
- Replace Inheritance with Delegation

Extract Method: Theorie

- Symptom und Abhilfe:
  - Ein zusammenhängendes Code-Fragment kann ausgelagert werden
    - In eigene Methode verlagern, mit gut gewähltem Namen
- Positive Effekte:
  - Feingranularerer Code (High Cohesion)
  - Bildet Abstraktionsstufen im Code aus
    - „Höhere“ Methoden sind näher an Problemdomäne und natürlicher Sprache
- Hilft gegen:
  - Duplicated Code, Long Method, Code Comments

Extract Method: Kleines Beispiel

Code vorher: Vermischte Abstraktionsebenen, Code-Kommentar zeigt Blockanfang

```
void printSalesCheck(List itemList) {
    printLogo();
    // print all items
    for (Iterator iter = itemList.iterator(); iter.hasNext();) {
        Item item = (Item) iter.next();
        printLine(item.getName() + ": " + item.getPrice());
    }
    printFooter();
}
```

Code nachher: Methodenhierarchie bildet sich, Methodenname sprechend gewählt

```
void printSalesCheck(List itemList) {
    printLogo();
    printAllItemsOf(itemList);
    printFooter();
}

void printAllItemsOf(List itemList) {
    for (Iterator iter = itemList.iterator(); iter.hasNext();) {
        Item item = (Item) iter.next();
        printLine(item.getName() + ": " + item.getPrice());
    }
}
```

DHBW

Extract Method: Praxis Vorher

```
public void activate(final DelphinValueProvider valueProvider, final RamsesStationManager stationManager) {
    if (ArrayUtil.isEmptyOrNull(stationManager.getAllStations())) {
        DefaultDialog.showWarningDialog(getSession().getMainWindowStack(),
                new I18NKey("warning.no.stations")); //$NON-NLS-1$
        return;
    }
    DelphinChannelSpecificationDialog channelDialog = new DelphinChannelSpecificationDialog(
            getSession(), RamsesStationManagerUtil.getAllStationIdentifiersFor(stationManager));
    channelDialog.showDialog();
    if (channelDialog.wasClosedByCancel()) {
        return;
    }

    final DelphinValueDevelopingDialog dialog = new DelphinValueDevelopingDialog(
            getSession(),
            new GenericDelphinChannelReference(
                    channelDialog.getSelectedStation(),
                    channelDialog.getChannelName())
    );
    valueProvider.addDelphinValueListener(dialog);
    dialog.showDialog();
    dialog.addCloseListener(new DialogCloseListener() {
        @Override
        public void dialogClosed(SchneideDialog schneideDialog) {
            valueProvider.removeDelphinValueListener(dialog);
        }
    });
}
```

DHBW

Extract Method: Analyse

```
public void activate(final DelphinValueProvider valueProvider, final RamsesStationManager stationManager) {
    if (ArrayUtil.isEmptyOrNull(stationManager.getAllStations())) {
        DefaultDialog.showWarningDialog(getSession().getMainWindowStack(),
                new I18NKey("warning.no.stations")); //$NON-NLS-1$
        return;
    }
    DelphinChannelSpecificationDialog channelDialog = new DelphinChannelSpecificationDialog(
            getSession(), RamsesStationManagerUtil.getAllStationIdentifiersFor(stationManager));
    channelDialog.showDialog();
    if (channelDialog.wasClosedByCancel()) {
        return;
    }
    final DelphinValueDevelopingDialog dialog = new DelphinValueDevelopingDialog(
            getSession(),
            new GenericDelphinChannelReference(
                    channelDialog.getSelectedStation(),
                    channelDialog.getChannelName())
    );
    valueProvider.addDelphinValueListener(dialog);
    dialog.showDialog();
    dialog.addCloseListener(new DialogCloseListener() {
        @Override
        public void dialogClosed(SchneideDialog schneideDialog) {
            valueProvider.removeDelphinValueListener(dialog);
        }
    });
}
```

Vorbedingung

Parameterwahl

Ergebnisanzeige

Extract Method: Praxis Nachher

```
    public void activate(final DelphinValueProvider valueProvider, final RamsesStationManager stationManager) {
        ensureThatThereAreStationsAt(stationManager);
        DelphinChannelSpecificationDialog chosenChannel = chooseDelphinChannelFrom(stationManager);
        showValueDevelopmentOf(chosenChannel, valueProvider);
    }

    protected void ensureThatThereAreStationsAt(final RamsesStationManager stationManager) {
        if (ArrayUtil.isEmptyOrNull(stationManager.getAllStations())) {
            DefaultDialog.showWarningDialog(getSession().getMainWindowStack(),
                    new I18NKey("warning.no.stations")); //$NON-NLS-1$
            throw new UserCancelProcessException();
        }
    }

    protected DelphinChannelSpecificationDialog chooseDelphinChannelFrom(
            final RamsesStationManager stationManager) {
        DelphinChannelSpecificationDialog channelDialog = new DelphinChannelSpecificationDialog(
                getSession(), RamsesStationManagerUtil.getAllStationIdentifiersFor(stationManager));
        channelDialog.showDialog();
        if (channelDialog.wasClosedByCancel()) {
            throw new UserCancelProcessException();
        }
        return channelDialog;
    }

    protected void showValueDevelopmentOf(final DelphinChannelSpecificationDialog chosenChannel,
            final DelphinValueProvider valueProvider) {
        final DelphinValueDevelopingDialog dialog = createDevelopmentDialogFor(chosenChannel);
        valueProvider.addDelphinValueListener(dialog);
        dialog.showDialog();
        dialog.addCloseListener(new DialogCloseListener() {
            @Override
            public void dialogClosed(SchneideDialog schneideDialog) {
                valueProvider.removeDelphinValueListener(dialog);
            }
        });
    }

[...]
```

Extract Method: Pro-Tipp

- Erkennen der semantisch zusammenhängenden Code-Blöcke
  - Orientierung an „Landmarken“ im Code
    - For-Schleifen, if-Statements, Code-Kommentare
- Lokale Variablen machen Probleme
  - Rückgabe-Typ der extrahierten Methode ist eventuell ein neuer, komplexer Typ
  - Möglichst auf In-/Out-Parameter verzichten
- Iteratives Vorgehen
  - Nach dem Extrahieren der kleinsten Code-Blöcke bilden sich wieder Extraktionspunkte

Rename Method: Theorie

- Symptom und Abhilfe:
  - Der Name einer Methode ist kryptisch, nicht sprechend oder nicht passend
    - Methodenname ändern
- Positive Effekte:
  - Erhöht die Lesbarkeit und Selbstdokumentations-fähigkeit des Codes
    - Leichtes Lesen ist wichtiger als leichtes Schreiben
      - Moderne IDE haben Automatic Code Completion
- Hilft gegen:
  - Code Comments

Rename Method: Kleines Beispiel

```
String getMsmtSensDescr() {
    return measurement.getSensor().getDescription();
}
```

```
String getMeasurementSensorDescription() {
    return measurement.getSensor().getDescription();
}
```

```
DoseOutput getHiDOSensLmt() {
    return highDoseOutputSensor.getLimit();
}
```

```
DoseOutput getHighDoseOutputSensorLimit() {
    return highDoseOutputSensor.getLimit();
}
```

DHBW

Rename Method: Praxis

```java
public class Person {

    private final String officeAreaCode;
    private final String officeNumber;

    public String getTelephoneNumber() {
        return Embrace.withParentheses(this.officeAreaCode) + Text.SPACE + this.officeNumber;
    }
}
```

```java
public class Person {

    private final String officeAreaCode;
    private final String officeNumber;

    /**
     * @deprecated use getOfficeTelephoneNumber() from now on
     */
    @Deprecated
    public String ~~getTelephoneNumber~~() {
        return getOfficeTelephonNumber();
    }

    public String getOfficeTelephonNumber() {
        return Embrace.withParentheses(this.officeAreaCode) + Text.SPACE + this.officeNumber;
    }
}
```

DHBW

Rename Method: Pro-Tipp

- Methodennamen sind „erlaubte“ Code-Kommentare
  - Name sollte die Intention der Methode möglichst exakt wiedergeben
- Kleine Verbesserungsmöglichkeiten sind auch wertvoll
  - `public String getFaxNumber(Person person)`
  - `public String getFaxNumberOf(Person person)`
    - Führt z.B. zum Aufruf `getFaxNumberOf(joe)`

Replace Temp with Query: Theorie

- Symptom und Abhilfe:
  - Eine temporäre (lokale) Variable wird benutzt, um das Ergebnis einer Berechnung zu speichern
    - Berechnung in Methode auslagern
    - Methode aufrufen statt Variable zu lesen
- Positive Effekte:
  - Seitenfreiheit der Berechnung wird geklärt
    - Schreibzugriffe auf lokale Variable werden sichtbar
  - Extract Method kann leichter verwendet werden
- Hilft gegen:
  - Long Method

DHBW

Replace Temp with Query: Kleines Beispiel

Code vorher: unnötige temporäre Variable

```
Euro basePrice = new Euro(quantity * itemPrice.getValue());
if (basePrice.getValue() > 1000.0d) {
    return basePrice.multiplyWith(0.95);
}
return basePrice.multiplyWith(0.98);
```

Code nachher: Methodenaufruf statt Zwischenspeicher, immer korrekter Wert

```
if (basePrice().getValue() > 1000.0d) {
    return basePrice().multiplyWith(0.95);
}
return basePrice().multiplyWith(0.98);

Euro basePrice() {
    return new Euro(quantity * itemPrice.getValue());
}
```

DHBW

Replace Conditional with Polymorphism: Theorie

- Symptom und Abhilfe:
  - Eine Konditionalstruktur wählt in Abhängigkeit eines Parameters verschiedenes Verhalten aus
    - Jeden Pfad der Konditionalstruktur in die überschreibende Methode einer Unterklasse verlagern
    - Originalmethode abstrakt definieren
- Positive Effekte:
  - Hilft gegen Wiederholung der Konditionalstruktur
  - Leicht änder- und erweiterbar, sogar dynamisch
- Hilft gegen:
  - Switch-Statement

Replace Conditional with Polymorphism: Praxis Vorher

```
public class Employee {

    private final Type type;
    private final Euro baseSalary;

    public Euro getSalaryFor(Balance lastMonth) {
        if (Type.DEVELOPER == this.type) {
            return this.baseSalary;
        }
        if (Type.MANAGER == this.type) {
            return this.baseSalary.plus(lastMonth.getBonus());
        }
        if (Type.SALESMAN == this.type) {
            return (this.baseSalary.plus(
                    lastMonth.getBonus().times(lastMonth.getSalesFactor())));
        }
        throw new WillNotPayYouException("You're fired!");
    }

    public BusinessCarUsage getAllowedCarUsage() {
        if (Type.DEVELOPER == this.type) {
            return BusinessCarUsage.NONE;
        }
        if (Type.MANAGER == this.type) {
            return BusinessCarUsage.cityOnly(CarCategory.EXECUTIVE);
        }
        if (Type.SALESMAN == this.type) {
            return BusinessCarUsage.unlimitedWith(CarCategory.REPRESENTATIVE);
        }
        return BusinessCarUsage.NONE;
    }

    protected enum Type {
        DEVELOPER,
        MANAGER,
        SALESMAN;
    }
}
```

DHBW

Replace Conditional with Polymorphism: Praxis Nachher

```
public abstract class Employee {

    private final Euro baseSalary;

    public abstract Euro getSalaryFor(Balance lastMonth);

    public abstract BusinessCarUsage getAllowedCarUsage();

    protected final Euro baseSalary() {
        return this.baseSalary;
    }
}
```

```
public class Developer extends Employee {
    @Override
    public Euro getSalaryFor(Balance lastMonth) {
        return baseSalary();
    }

    @Override
    public BusinessCarUsage getAllowedCarUsage() {
        return BusinessCarUsage.NONE;
    }
}
```

```
public class Salesman extends Employee {
    @Override
    public Euro getSalaryFor(Balance lastMonth) {
        return (baseSalary().plus(
              lastMonth.getBonus().times(lastMonth.getSalesFactor())));
    }

    @Override
    public BusinessCarUsage getAllowedCarUsage() {
        return BusinessCarUsage.unlimitedWith(CarCategory.REPRESENTATIVE);
    }
}
```

[...]

Replace Conditional with Polymorphism: Pro-Tipp

- Zusätzlich zur Polymorphie Lookup-Datenstruktur verwenden
  - Map bzw. HashMap in Java
  - Ermöglicht einen dezentral und dynamisch befüllten switch-Ersatz
  - Bisheriger switch-Parameter wird Lookup-Schlüssel

```
public class ToppingPriceCalculator {
    private final Map<Topping, Euro> prices;

    public ToppingPriceCalculator() {
        this.prices = new HashMap<Topping, Euro>();
    }

    public Euro getPriceFor(Topping topping) {
        return this.prices.get(topping);
    }

    public void addPriceFor(Topping topping, Euro price) {
        this.prices.put(topping, price);
    }
}
```

Replace Error Code with Exception: Theorie

- Symptome und Abhilfe:
  - Ein oder mehrere spezielle Fehlerwerte werden im Fehlerfall als Rückgabewert geliefert
    - Ausnahme auslösen (Exception schmeißen)
- Positive Effekte:
  - Trennt Fehlerfall vom Normalfall
  - Fehlerwerte müssen nicht mehr explizit abgeprüft werden
  - Exceptions können nicht in Berechnung einfließen
    - Fehlerwerte sind aus dem Wertebereich des Rückgabetyps, Exceptions nicht

Replace Error Code with Exception: Kleines Beispiel

Code vorher: Spezieller Fehlerwert (Magic Number), falls Sensor ausgefallen

```
int getAngle() {
    if (sensor.isWorking()) {
        return sensor.getAngle();
    }
    return -1;
}
```

Code nachher: Exception, falls Sensor ausgefallen

```
int getAngle() throws SensorException {
    if (sensor.isWorking()) {
        return sensor.getAngle();
    }
    throw new SensorException(„Sensor ausgefallen.“);
}
```

DHBW

Replace Error Code with Exception: Praxis Vorher

```
public class AngleSensor {
    public static final int SENSOR_ERROR = -1;
    private final AngleHardware hardware;

    public AngleSensor(AngleHardware hardware) {
        this.hardware = hardware;
    }

    public int getAngle() {
        if (hardware.isWorking()) {
            return hardware.getCurrentValue();
        }
        return SENSOR_ERROR;
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        final AngleSensor sensor = new AngleSensor(with(AngleHardware.DEFAULT));
        final Display display = new DisplayFor().currentMonitor();
        while (Eternity.isInTheFuture()) {
            final int currentAngle = sensor.getAngle();
            if (AngleSensor.SENSOR_ERROR == currentAngle) {
                display.showError();
                continue;
            }
            display.show(currentAngle);
        }
    }

    private static AngleHardware with(AngleHardware hardware) {
        return hardware;
    }
}
```

Replace Error Code with Exception: Praxis Nachher

```
public class AngleSensor {
    private final AngleHardware hardware;

    public AngleSensor(AngleHardware hardware) {
        this.hardware = hardware;
    }

    public int getAngle() throws SensorException {
        if (this.hardware.isWorking()) {
            return this.hardware.getCurrentValue();
        }
        throw new SensorException("Sensor hardware is not working");
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        final AngleSensor sensor = new AngleSensor(with(AngleHardware.DEFAULT));
        final Display display = new DisplayFor().currentMonitor();
        while (Eternity.isInTheFuture()) {
            try {
                final int currentAngle = sensor.getAngle();
                display.show(currentAngle);
            } catch (SensorException e) {
                display.showError(e.getMessage());
            }
        }
    }

    private static AngleHardware with(AngleHardware hardware) {
        return hardware;
    }
}
```

Bemerke: Durch die Exception kann die Fehlermeldung mit angezeigt werden.

Replace Inheritance with Delegation: Theorie

- Symptome und Abhilfe:
  - Eine Unterklasse verwendet nur einen Teil der Funktionalität der Oberklasse
    - Die Oberklasse wird eine Instanzvariable (Delegation)
- Positive Effekte:
  - Die Schnittstelle wird prägnanter
    - enthält keine „unbenutzbaren“ Methoden mehr
  - Eigene Funktionalität wird entkoppelt
  - Beseitigt offensichtliche Verletzungen des LSP (Liskov Substitution Principle)

Replace Inheritance with Delegation: Praxis Vorher

```
public class MyStack<T> extends ArrayList<T> {

    public MyStack() {
        super();
    }

    public void push(T element) {
        add(0, element);
    }

    public T pop() {
        return remove(0);
    }
}
```

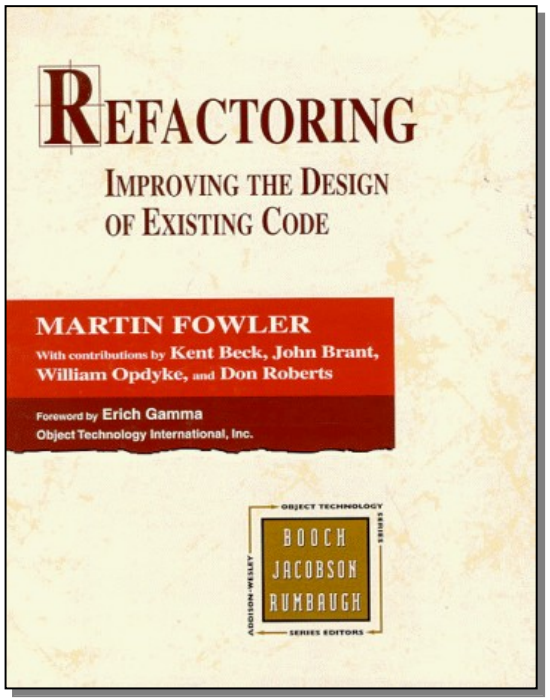
- Probleme der Implementierung:
  - ArrayList bietet sehr viele in diesem Kontext kontraproduktive Methoden, bspw. remove(int)
    - All diese Methoden werden dem Klienten mit angeboten
  - Die Schnittstelle ist zu groß. Für den Stack reicht:
    - push, pop, getSize und isEmpty

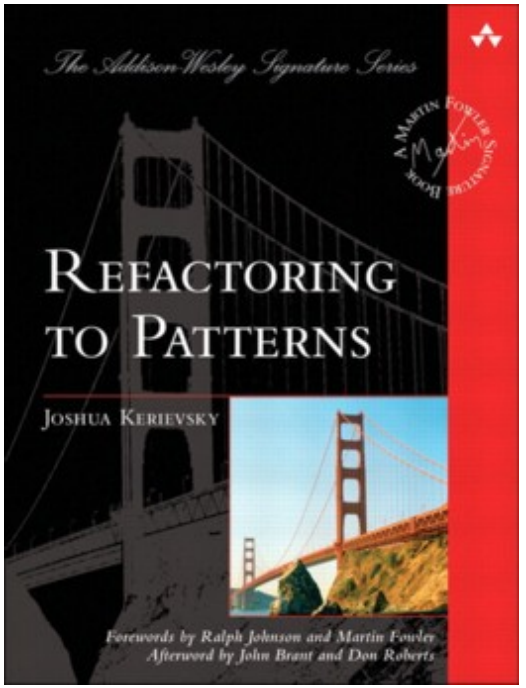
Replace Inheritance with Delegation: Praxis Nachher

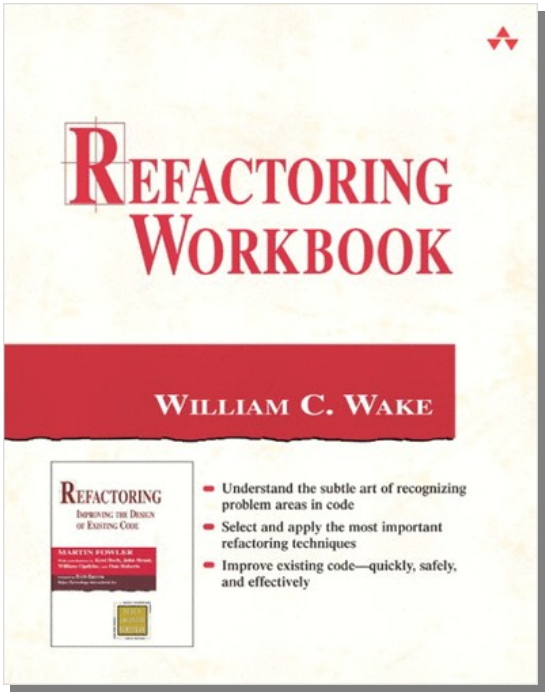
```
public class MyStackWithDelegation<T> {
    private final ArrayList<T> elements;

    public MyStackWithDelegation() {
        super();
        this.elements = new ArrayList<T>();
    }

    public void push(T element) {
        this.elements.add(0, element);
    }

    public T pop() {
        return this.elements.remove(0);
    }

    public int getSize() {
        return this.elements.size();
    }

    public boolean isEmpty() {
        return this.elements.isEmpty()
    }
}
```

- Schnittstelle ist jetzt genau passend
  - Keine verbotenen Methoden
  - Umbenennung möglich

Literatur zum Thema







- http://www.refactoring.com/
  - Mit umfangreichem Online-Katalog der Refactorings

DHBW

Wer hats aufgeschrieben?

- Martin Fowler
  - Autor mehrerer guter Bücher, u.a.:
    - UML Distilled
    - Analysis Patterns
- Vor- und Nachdenker für Software/IT
- Schreibt viel beachtete Aufsätze
  - Veröffentlicht in seinem „Bliki“
- Spricht oft (und gerne) auf Konferenzen
- Veröffentlicht „Signature“-Reihe an Büchern

DHBW

Anhang: Der Teaser in voller Länge

```
public class CleanupProcess {
      public void cleanup(final File directory) throws IOException {
            ForeachFile.in(directory).perform(
                        DeleteIf.olderThan(5).days());
      }
}
```

DHBW

Anhang: Der Teaser in voller Länge

```
public class DeleteIf {
	private static final long MILLISECONDS_OF_SECOND = 1000;
	private static final long MILLISECONDS_OF_MINUTE = 60 * MILLISECONDS_OF_SECOND;
	private static final long MILLISECONDS_OF_HOUR = 60 * MILLISECONDS_OF_MINUTE;
	private static final long MILLISECONDS_OF_DAY = 24 * MILLISECONDS_OF_HOUR;
	private final int multiplier;

	private DeleteIf(int multiplier) {
		this.multiplier = multiplier;
	}

	public static DeleteIf olderThan(int multiplier) {
		return new DeleteIf(multiplier);
	}

	public FileVisitor days() {
		return createOperationForMaximalAge(this.multiplier * MILLISECONDS_OF_DAY);
	}

	protected FileVisitor createOperationForMaximalAge(final long milliseconds) {
		return new FileVisitor() {
			@Override
			protected void performActionFor(File file) throws IOException {
				if (file.lastModified() < (System.currentTimeMillis() - milliseconds)) {
					file.delete();
				}
			}
		};
	}
}
```

Anhang: Der Teaser in voller Länge

```
public abstract class FileVisitor {
    public void visitAllFilesUnder(File directory) throws IOException {
        if (!directory.isDirectory()) {
            return;
        }
        for (File file : listFilesUnder(directory)) {
            visit(file);
        }
    }

    protected File[] listFilesUnder(File directory) {
        File[] files = directory.listFiles();
        if (null == files) {
            return new File[0];
        }
        return files;
    }

    protected void visit(File fileObject) throws IOException {
        if (fileObject.isDirectory()) {
            visitAllFilesUnder(fileObject);
        }
        if (fileObject.isFile()) {
            performActionFor(fileObject);
        }
    }

    /**
     * Intended for subclass usage.
     * Every file found in the subtree will be used as a parameter
     * for a call to this method. Implementations should insert their
     * functionality here.
     */
    protected abstract void performActionFor(File file) throws IOException;
}
```

DHBW

Anhang: Der Teaser in voller Länge

```
public final class ForeachFile {
    private final File parent;

    private ForeachFile(File parent) {
        super();
        this.parent = parent;
    }

    public static ForeachFile in(File directory) {
        return new ForeachFile(directory);
    }

    public void perform(final FileVisitor visitor) throws IOException {
        visitor.visitAllFilesUnder(this.parent);
    }
}
```

DHBW