DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

REFACTORING

Lars Briem

(briem.lars@googlemail.com)

Duale Hochschule Baden Württemberg - Standort Karlsruhe

Was bedeutet Refactoring

- Bereits geschriebener Code wird erneut durchgegangen
  - Ein Code Review kann auch von mehr als einer Person gemacht werden
- Die Intention und Funktion des Codes wird nachvollzogen
- Die Testabdeckung wird überprüft und bei Bedarf erhöht
  - Refactoring erfordert eine gute Testabdeckung

Was bedeutet Refactoring

- Der Code wird umgestaltet
  - Das Gesamtverhalten bleibt gleich
  - Die Schnittstellen nach außen bleiben gleich
  - Neues Wissen zum Problem wird integriert
- Ziel: Codequalität verbessern
  - Code wird einfacher lesbar
  - Code kann flexibler genutzt werden
  - Struktur des Codes passt besser zur Problemdomäne

Definition Refactoring

- Refactoring (Substantiv)
  - Eine Änderung der internen Struktur des Codes, um die Software einfacher verständlich und veränderbar zu machen, ohne das nach außen sichtbare Verhalten zu ändern.
- Refactoring (Verb)
  - Das Ausführen einer Reihe von Refactorings, um die Qualtität des Codes zu verbessern.

Wie sieht Code in der Praxis (oft) aus

- Historisch gewachsene Codebasis
- Komplizierte Strukturen
  - Viel „Spaghetti-Code“
- Sonderfall Behandlungen
- Unverständliche Namen
- Keine ausreichende Testabdeckung
  - Oft manuelles Testen
  - Tests sind immer noch nicht Standard

Warum sollten wir refactorisieren

- Das Design der Software wird verbessert
  - Um das Problem zu verstehen wird zunächst die Funktionalität implementiert
  - Beim Refactoring oder Review wird anschließend die Lesbarkeit und Wiederverwendung verbessert
- Die Software wird wartbarer
  - Höhere Strukturen und Konzepte der Problemdomäne bilden sich heraus
  - Verwendung höherer Konzepte verkleinert die Codebasis

Warum sollten wir refactorisieren

- Die Software wird einfacher verständlich
  - Software muss für den Menschen verständlich sein, nicht nur für den Computer
  - Softwareentwicklung ist Teamarbeit
  - Je länger die Implementierung zurückliegt, desto größer ist der Einarbeitungsaufwand um den Code erneut zu verstehen
- Fehler werden einfacher gefunden
  - In verständlichem Code sind Fehler einfacher zu finden
  - Bei einem erneuten Betrachten des Codes werden Spezialfälle oft häufiger gefunden

Warum sollten wir refactorisieren

- Neue Funktionalität kann schneller entwickelt werden
  - Am Anfang eines Projekts entscheidet die Qualität des Codes, nicht die Entwicklungsgeschwindigkeit
  - Je länger das Projekt läuft, desto entscheidender wird die Qualität für die Geschwindigkeit
  - Neuer Code reduziert die Qualität des gesamten Codes
  - Refactoring hebt die Qualität wieder auf das vorherige Niveau oder sogar darüber

Warum ändern wir Software

- Hinzufügen oder Erweitern der Funktionalität
  - Neue Funktionalität verstehen und implementieren
  - Einfaches Design
- Beheben eines Fehlers bzw. Bugs
  - Testen der entwickelten Funktion
  - Fehler finden und beheben
- Verbesserung des Designs bzw. Refactoring
  - Code verständlicher machen
- Optimierung des Ressourcenverbrauchs

Warum ändern wir Software

- make it!
  - Neue Funktionalität verstehen und implementieren
  - Einfaches Design
- make it run!
  - Testen der entwickelten Funktion
  - Fehler finden und beheben
- make it better!
  - Code verständlicher machen

In der Entwicklung - make it!

Aufgabe: Lösche in einer Verzeichnisstruktur alle Log-Dateien, die älter als 5 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;
      }
      long currentTime = System.currentTimeMillis();
      if (file.lastModified()<(currentTime-432000000L)) {
        file.delete();
      }
    }
  }
}
```

In der Entwicklung - make it run!

Fehler: Bei leeren Verzeichnissen stürzt das Programm mit einer NullPointerException ab

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

In der Entwicklung - make it better!

Refactoring: Forme den Code solange um, bis er einfach verständlich wird

```
public class CleanupProcess {

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

Wann sollten wir Refactorieren

- Refactoring sollte Bestandteil der normalen Entwicklung sein und dadurch kontinuierlich genutzt werden
- Einfache Regel
  - *Three Strikes and you refactor*

1. Beim ersten Implementieren neue Funktionalität hinzufügen
2. Wird eine ähnliche Funktionalität ein zweites Mal benötigt $\Rightarrow$ Code kopieren
3. Wird eine ähnliche Funktionalität erneut benötigt $\Rightarrow$ Code refactorn und wiederverwenden

Wann sollten wir Refactorieren

- Während einem Code Review
  - Der Reviewer hilft den Code für andere Entwickler verständlicher zu gestalten
- Vor dem Hinzufügen von neuer Funktionalität
  - Alter Code wird auf die Änderung vorbereitet
  - Refactoring vereinfacht das Einbauen neuer Funktionalität
- Beim Beheben eines Fehlers
  - Refactoring hilft beim Verstehen von bestehendem Code
  - Fehler betreffen oft mehrere nahe Stellen

Exkurs: Lokalitätsprinzip

- *Ein Bug kommt selten allein*
- Bugs im Code *klumpen* zusammen
  - Fehler werden durch Gegenfehler kompensiert (siehe Testing Vorlesung)
  - Komplexer Code ist anfälliger für Bugs
  - Zusammenhängender Code wird oft vom gleichen Entwickler geschrieben
  - Zusammenhängender Code wird oft zur gleichen Zeit geschrieben

Warum Refactoring funktioniert

- Entscheidungen der Vergangenheit werden nicht zur Last der Zukunft
  - Entscheidungen können geändert werden
- Software ist schwieriger zu lesen als zu schreiben
  - Schlecht lesbare Software ist auch schlecht änderbar
  - Komplexe Logik ist schlecht änderbar
- Refactoring hält die Geschwindigkeit langfristig konstant
  - Langfristiger Nutzen

Wie erklärt man das dem Management

- Im Idealfall gar nicht
  - Die Aufgabe eines Entwicklers ist es professionell Software zu entwickeln
  - Das Management sollte sich nicht dafür interessieren, wie er entwickelt
  - Refactoring erhöht langfristig die Geschwindigkeit und hilft den Zeitplan einzuhalten
- Ein Chef der Wert auf Qualität legt, lässt sich leicht von Refactoring überzeugen
  - Das Hauptziel von Refactoring ist die Codequalität

Wann wird Refactoring schwierig

- Datenbankschemata können nicht so einfach wie Code geändert werden
  - Änderung des Datenbankschemas bedeuten auch immer eine Umwandlung von Daten
- Bei der Änderung von Schnittstellen (Interfaces) hat man unter Umständen nicht allen aufrufenden Code selbst in der Hand
  - Interfaces gibt es in mehreren Veröffentlichungsstufen
    - öffentlich
    - teilweise öffentlich
    - intern

Wann wird Refactoring schwierig

- Zentrale Designentscheidungen, die später nur schwer geändert werden können
  - Bei zentralen Komponenten sollte überlegt werden, ob diese später einfach geändert werden können
  - Können sie nicht einfach geändert werden, sollte mehr Aufwand in das Design investiert werden
- Wird der Aufwand für Refactoring zu groß, können Teile oder alles komplett neu entwickelt werden
  - z.B. bei der Änderung einer zentralen Technologie

Auswirkungen auf das Design

- Klassische Vorgehensweise eines Ingenieurs
  1. Design bzw. Entwurf des Produkts
  2. Konstruktion bzw. Produktion des Produkts
- Software ist kein klassisches Produkt
  - Software lässt sich deutlich einfacher ändern als Hardware
  - Während der Entwicklung das Problem besser verstehen
  - Durch Refactoring eine bessere Lösung für das Problem finden

Nachteile von Refactoring

- Refactoring kostet Zeit
  - Aber neue Funktionalität kann später einfacher eingebaut werden
- Refactoring verschlechtert die Performance von Software
  - Die Annahme ist, dass Indirektionen und Abstraktionen mehr Overhead erzeugen
  - Annahmen bei der Performance Optimierung immer schlecht

Exkurs: Performance Optimierung

- Donald Knuth: *premature optimization is the root of all eval*
- 90/10 Regel oder Pareto Prinzip
  - 90% der Zeit wird in 10% des Codes verbracht
  - Optimierung des gesamten Codes unnötig
- Optimierungen nur nach einer passenden Messung

Was sind Code Smells

- *If it stinks, change it.*
  - „Code Smells“ deuten auf verbesserungswürdige Stellen im Code hin
- Code Smells geben einen Eindruck, welche Konstrukte die Entwicklung behindern
  - Kochrezepte zum Finden von Problemstellen
  - Lösungen werden in Form von Refactorings gegeben
- Stärke des „Gestanks“ schlecht messbar
  - Teilweise Unterstützung durch Tools oder Algorithmen

Code Smell: Duplicated Code

- Doppelt vorhandener Code
- Wichtigster Code Smell
  - Kommt am häufigsten vor
  - Meistens durch Unwissenheit
- Die gleiche Code-Struktur ist an mehr als einer Stelle im Code vorhanden
  - Die gleiche Anweisung mehrfach in einer Methode / Klasse
  - Ähnlicher Code in mehreren Methoden oder Klassen

Code Smell: Duplicated Code

- Auseinanderdriften mehrerer Code-Stellen
  - Der Wartungsaufwand wird unnötigerweise erhöht
  - Programm wird unvollständig
- Lösung: Gemeinsamen Code auslagern und aufrufen
  - Neue Strukturen schaffen
  - Wiederverwendung im Code erhöhen

Code Smell: Long Method

- Lange Methoden
- Je länger ein zusammenhängendes Stück Code ist, desto schwerer ist es zu verstehen
  - Früher wurden Entwickler vom Overhead eines Funktionsaufrufs abgeschreckt
  - Heutige Sprachen haben diesen nahezu eliminiert
- Objektorientierter Code sieht für Anfänger oft aus, als ob nirgendwo etwas passiert
  - In vielen Methoden sind nur Delegationen

Code Smell: Long Method

- Eine große semantische Distanz zwischen Code und Name der Methode ist hinderlich
  - Je größer Methoden sind, desto schwerer ist es einen sprechenden Namen dafür zu finden
- Lösung: Auftrennen der Methode
  - Für kleinere Methoden lässt sich leichter ein Name finden
  - Gute Namen steigern die Lesbarkeit
  - Kommentare, Schleifen und If-Bedingungen bieten gute Nahtstellen

Exkurs: Code Seams

- Nahtstelle oder Saum
- Kleidungsbranche: Verbindung zweier Stoffteile
  - Ohne sichtbare Effekte auftrennbar
  - Beispiele: Patchwork oder Aufnäher
- Softwareentwicklung: Verbindung zweier Codeteile
  - Verbundener Code kann getrennt werden
  - Neuer Code bzw. Funktionalität kann eingefügt werden
  - Beispiele: Methodenaufrufe, Polymorphie

Code Smell: Large Class

- Große Klasse
- Häufige Indizien
  - Zu viele Instanzvariablen
  - Zu viele Methoden
  - Gleiche Prefixe bzw. Suffixe bei Variablen
- Ist ein Hinweis, dass die Klasse zu viel Verantwortung hat
- Lösung: Unterteilen der Klasse
  - Zusammengehörende Teilfunktionen extrahieren

Code Smell: Shotgun Surgery

- Flickenteppich Änderung
  - *Schrotflinten Chirurgie*
- Für eine kleine funktionale Änderung müssen viele Stellen im Code angepasst werden
  - Die Wahrscheinlichkeit, dass eine Stelle vergessen wird ist sehr hoch
  - Große Gefahr einen neuen Bug einzubauen
- Lösung: Umstrukturierung des Codes
  - Idealerweise wird eine Klasse nur aus einem Grund geändert

Code Smell: Switch Statements

- Switch Statements bringen viele Probleme in den Code
  - Gleicher Switch wird oft an mehreren Stellen genutzt $\Rightarrow$ Duplicated Code
  - Switch Statesments bieten wenige Nahtstellen, kann nur *im Platz* wachsen
  - Syntax fördert Fehler *break vs. fall-through*
- In objektorientierter Entwicklung ist das deutlich einfacher
  - Funktionalität muss nur an einer Stelle geändert oder erweitert werden

Code Smell: Code Comments

- Kommentare im Code sind wie ein Deodorant
  - Sie versuchen schlechten Code mit anderem Geruch zu überdecken
- Kommentare beheben nicht das eigentliche Problem
  - Sie versuchen Symptome zu beheben
- Kommentare sind gute Indikatoren zur Trennung von Methoden
  - Kommentierte Stelle in Methode auslagern
  - Intention des Kommentars als Namen für die Methode nehmen

Refactorings

- Kleine Auswahl aus über 60 Refactorings
  - Extract Method
  - Rename Method
  - Replace Temp with Query
  - Replace Conditional with Polymorphism
  - Replace ErrorCode with Exception
  - Replace Inheritance with Delegation

Extract Method

Problem

- Zusammenhängendes Stück Code kann extrahiert werden

Lösung

- Passenden Namen wählen und Code in eigene Methode auslagern
- Code feingranularer aufteilen
  - Fördert die Wiederverwendbarkeit von Methoden
  - Behebt die Vermischung verschiedener Abstraktionsebenen

Extract Method

Verbesserung

- Bessere Strukturierung des Codes
  - Größere Methoden lesen sich wie eine Abfolge von Kommentaren
  - Trennung zwischen Problemdomäne und ausführbaren Anweisungen
  - Unter Umständen lässt sich Code in natürlicher Sprache lesen

Hilft gegen

- Code Comments, Duplicated Code, Long Method

Extract Method - Ein kleines Beispiel

Vorher

```
void printAddresses() {
  printSchool();
  // Print student locations
  for (Student student : students) {
    print(student.getNumber() + "`: "` + student.getLocation());
  }
  // Print teacher locations
  for (Teacher teacher : teachers) {
    print(teacher.getNumber() + "`: "` + teacher.getLocation());
  }
}
```

Extract Method - Ein kleines Beispiel

Nachher

```
void printAllStudents() {
  printSchool();
  printLocationsOf(students);
  printLocationsOf(teachers);
}

void printLocationsOf(List<Person> persons) {
  for (Person person : persons) {
    print(person.getNumber() + "`: "` + person.getLocation());
  }
}
```

Extract Method - Beispiel aus der Praxis

Vorher

```
public void finishTrip(Scheduler scheduler, SimulationDate atCurrentDate) {
  if (scheduler.getActivitySchedule().hasCurrentTrip) {
    Logger.log(WARNING, "No trip to end available");
    return;
  }

  Trip trip = scheduler.getActivitySchedule().currentTrip();
  Activity previousActivity = trip.previousActivity();
  Activity nextActivity = trip.nextActivity();
  Location previousLocation = previousActivity.location();
  Location nextLocation = nextActivity.location();

  TripfileWriter.writeTripToFile(person(), trip, previousActivity, nextActivity,
      previousLocation, nextLocation);

  asDriverReturnCarAfter(trip);
  start(nextActivity, atCurrentDate);
}
```

Extract Method - Beispiel aus der Praxis

Nachher

```
public void finishTrip(Scheduler scheduler, SimulationDate atCurrentDate) {
  Trip trip = findTripIn(scheduler);
  log(trip);
  startNextActivityAfter(trip, atCurrentDate);
}
public Trip findTripIn(Scheduler scheduler) {
  if (scheduler.getActivitySchedule().hasCurrentTrip) {
    return scheduler.getActivitySchedule().currentTrip();
  }
  throw new TripNotAvailable();
}
public void log(Trip trip) {
  Activity previousActivity = trip.previousActivity();
  Activity nextActivity = trip.nextActivity();
  Location previousLocation = previousActivity.location();
  Location nextLocation = nextActivity.location();

  TripfileWriter.writeTripToFile(person(), trip, previousActivity, nextActivity,
    previousLocation, nextLocation);
}
public void startNextActivityAfter(Trip trip, SimulationDate atCurrentDate) {
  asDriverReturnCarAfter(trip);
  Activity nextActivity = trip.nextActivity();
  start(nextActivity, atCurrentDate);
}
```

Extract Method - Tipps für die Praxis

- Code Seams dienen als Hilfe zum Auftrennen langer Methoden
  - Kommentare, If-Bedingungen, Schleifen
- Typischerweise iteratives Vorgehen
  - Kleine Teile extrahieren
  - Anschließend kann der Aufruf mehrere kleiner Methoden wieder extrahiert werden
- Lokale Variablen sind problematisch
  - Eine Methode kann nur einen Wert zurückgeben
  - Vermeidung von lokalen Variablen über andere Refactorings

Rename Method

Problem

- Methodenname passt nicht zum Inhalt der Methode
  - Methodenname ist kryptisch, oftmals eine unverständliche Abkürzung

Lösung

- Methodenname ändern

Rename Method

Verbesserung

- Code wird für Menschen lesbarer und damit verständlicher
  - Kurze Namen sparen keine Zeit beim Programmieren
  - IDEs besitzen Autovervollständigung

Hilft gegen

- Code Comments

Rename Method - Einfaches Beispiel

Vorher

```
int getAccCDLmt() {
  return accound.getCreditCard().getLimit();
}
```

Nachher

```
int getCreditCardLimitForAccount() {
  return accound.getCreditCard().getLimit();
}
```

Vorher

```
Dialog createDvcDlg() {
  return system.getDevice().createDialog();
}
```

Nachher

```
Dialog createDeviceDialog() {
  return system.getDevice().createDialog();
}
```

Rename Method - Beispiel aus der Praxis

Vorher

```
public class Person {

  private final String officeAreaCode;
  private final String officeNumber;

  public String getTelephoneNumber() {
    return this.officeAreaCode + "/" + this.officeNumber;
  }
}
```

Rename Method - Beispiel aus der Praxis

Nachher

```
public class Person {

  private final String officeAreaCode;
  private final String officeNumber;

  /**
  * @deprecated use getOfficeTelephoneNumber() instead
  */
  @Deprecated
  public String getTelephoneNumber() {
    return getOfficeTelephonNumber();
  }

  public String getOfficeTelephonNumber() {
    return this.officeAreaCode + "/" + this.officeNumber;
  }
}
```

Rename Method - Tipps für die Praxis

- Methoden umbenennen hilft beim Lesen und Verstehen des Codes
  - Code Review
- Methodennamen sind „lebende“ Kommentare
  - Methodenname sollte beschreiben was die Methode macht, nicht wie sie es macht
- Schon kleine Änderungen verbessern die Lesbarkeit
  - `void printLocation(Student student);`
  - `void printLocationOf(Student student);`

Replace Temp with Query

Problem

- Das Ergebnis einer Berechnung wird temporär in einer Variable gespeichert
  - Wert der Variablen wird nur einmal gesetzt und nicht mehr verändert
  - Die Berechnung hat keine Seiteneffekte

Lösung

- Berechnung des Werts in eine Methode auslagern

Replace Temp with Query

Verbesserung

- Es wird einfacher Methoden zu extrahieren und wieder zu verwenden
- Schreibzugriffe auf Variablen werden sichtbar
- Wert der Berechnung wird nicht zwischengespeichert und ist immer aktuell

Hilft gegen

- Long Method

Replace Temp with Query - Beispiel

Vorher

```
double basePrice = quantity * itemPrice;
if (basePrice > 1000) {
  return basePrice * 0.95;
}
return basePrice * 0.98;
```

Nachher

```
if (basePrice() > 1000) {
  return basePrice() * 0.95;
}
return basePrice() * 0.98;

double basePrice() {
  return quantity * itemPrice;
}
```

Replace Conditional with Polymorphism

Problem

- Das Verhalten wird mit Konditionalstrukturen und einer Typ-Kodierung gesteuert
  - If-Else oder Switch
  - Kommt das gleiche Konstrukt mehrfach vor, muss es mehrfach gepflegt werden

Lösung

- Verhalten der einzelnen Pfade in abgeleiteten Klassen überschreiben
- Basismethode als abstrakt definieren

Replace Conditional with Polymorphism

Verbesserung

- Bei der Erweiterung mit Hilfe von abgeleiteten Klassen, muss das ganze Verhalten implementiert werden
- Anwender kennen die Unterklasse nicht, die Software ist besser gekapselt
- Dynamisch erweiterbar

Hilft gegen

- Switch-Statements

Replace Conditional with Polymorphism

Vorher

```
class Employee {
  private final EmployeeType type;
  private final Money monthlySalary;
  Money payAmount(Balance lastMonth) {
    switch (type()) {
      case EmployeeType.ENGINEER:
        return monthlySalary;
      case EmployeeType.SALESMAN:
        return monthlySalary.plus(lastMonth.bonus());
      case EmployeeType.MANAGER:
        return monthlySalary.plus(lastMonth.bonus()
                  .times(lastMonth.salesFactor()));
      default:
        throw new UnknownEmployee("No salary for you!");
    }
  }
}
enum EmployeeType {
  ENGINEER, SALESMAN, MANAGER;
}
```

Replace Conditional with Polymorphism

Nachher

```
class Employee {
  private final Money monthlySalary;

  abstract Money payAmount(Balance lastMonth);
  protected Money monthlySalary() {
    return monthlySalary;
  }
}
class Engineer extends Employee {
  @Override
  Money payAmount(Balance lastMonth) {
    return monthlySalary();
  }
}
class Salesman extends Employee {
  @Override
  Money payAmount(Balance lastMonth) {
    return monthlySalary().plus(lastMonth.bonus());
  }
}
```

Replace Conditional with Polymorphism - Tipps für die Praxis

- Dynamische Switch Konstrukte sind mit Lookup-Mechanismen realisierbar
  - HashMaps als Switch Ersatz verwenden
  - Switch-Parameter als Key für die HashMap verwenden
  - Mehr Datentypen verwendbar verglichen mit Switch-Parameter

Replace Conditional with Polymorphism - Tipps für die Praxis

```
public class Translator {

  private final Map<Language, Text> translation;

  Translator() {
    translation = new HashMap<>();
  }

  Text getTextFor(Language language) {
    return translation.get(language);
  }

  void addLanguage(Language language, Text text) {
    translation.put(language, text);
  }
}
```

Replace ErrorCode with Exception

Problem

- Bei der Verarbeitung von Daten können Fehler auftreten
  - Im Fehlerfall wird oft ein „Fehlerwert“ anstatt eines normalen Wertes zurückgegeben

Lösung

- Anstelle des Fehlerwertes eine Exception werfen
  - Exceptions müssen getrennt von normalen Werten verarbeitet werden

Replace ErrorCode with Exception

Verbesserung

- Klare Definition von Fehlern
  - Automatische Trennung bei der Verarbeitung von Fehlern und normalen Werten
  - Fehler können nicht in Berechnung einfließen
  - Zurückgegebene Werte müssen *nicht* auf Fehler geprüft werden
- Genauere Steuerung des Kontrollflusses
- Code wird verständlicher und lesbarer

Replace ErrorCode with Exception

Vorher

```
int withdraw(int amount) {
  if (amount > balance) {
    return -1
  }
  balance -= amount;
  return 0;
}
```

Nachher

```
void withdraw(int amount) throws BalanceException {
  if (amount > balance) {
    throw new BalanceException();
  }
  balance -= amount;
}
```

Replace ErrorCode with Exception

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) {
    AngleSensor sensor = new AngleSensor(withDefaultHardware());
    Display display = new DisplayFor().currentMonitor();
    while (systemIsRunning()) {
      int currentAngle = sensor.getAngle();
      if (AngleSensor.SENSOR_ERROR == currentAngle) {
        display.showError();
        continue;
      }
      display.show(currentAngle);
    }
  }
}
```

Replace ErrorCode with Exception

Nachher

```
public class AngleSensor {

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

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

Replace Inheritance with Delegation

Problem

- Funktionalität der Oberklasse in abgeleiteter Klasse nicht brauchbar
  - Schnittstelle nach außen entspricht nicht dem Verhalten der Klasse

Lösung

- Instanzvariable mit dem Typ der Oberklasse anlegen
  - Alle notwendigen Methoden an Instanzvariable delegieren
  - Ableitung zu Oberklasse entfernen

Replace Inheritance with Delegation

Verbesserung

- Klarer definierte Schnittstellen
  - Mehr Kontrolle welche Funktionalität der delegierten Klasse verwendbar ist
- Trennung zwischen eigener Funktionalität und bereits vorhandener Funktionalität

Tipp für die Praxis

- Beim Entwurf von APIs kann man die Schnittstelle exakter definieren
  - Weniger absichtliche oder unabsichtliche Missbrauchsgefahr

Replace Inheritance with Delegation

Vorher

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

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

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

}
```

- Wirklich benötigte Funktionalität
  - push, pop, size, isEmpty

Replace Inheritance with Delegation

Nachher

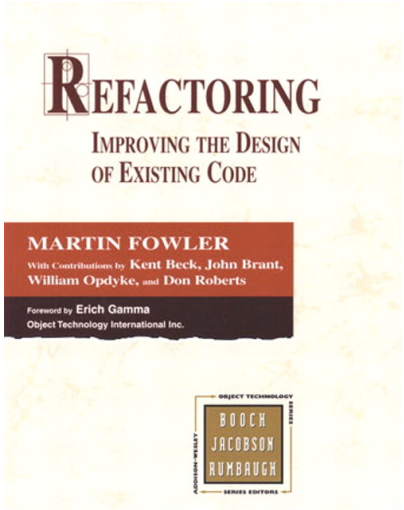
```
class MyStack<T> {
  List<T> elements;

  public MyStack() {
    elements = new ArrayList<>();
  }
  public void push (T element) {
    elements.add(0, element);
  }
  public T pop() {
    return elements.remove(0);
  }
  public boolean isEmpty() {
    return elements.isEmpty();
  }
  public int size() {
    return elements.size();
  }
}
```

- Schnittstelle ist auf den Anwendungszweck angepasst
- Kein Missbrauch mehr möglich

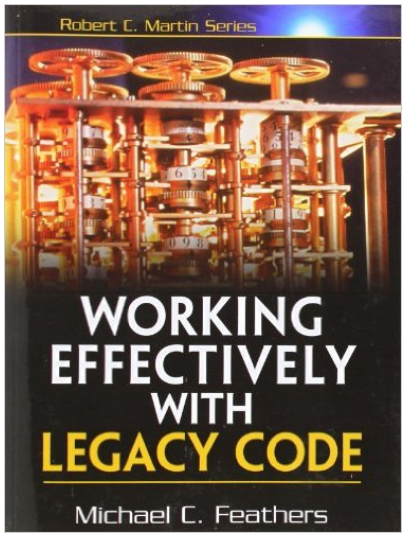
Zusammenfassung

- Code Smells
  - Code kann „stinken“
  - Schlechte Konstrukte im Code verlangsamen die Entwicklung langfristig
  - Ansatzpunkte an denen Code verständlicher werden kann
- Refactorings
  - Helfen den Code lesbarer zu gestalten
  - Verändern das beobachtbare Verhalten des Codes nicht
  - Halten die Entwicklungsgeschwindigkeit langfristig konstant

Literatur

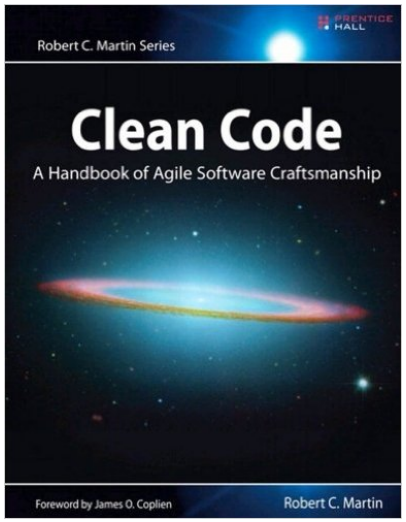


- Refactoring
  - Martin Fowler
  - Addison-Weslay
  - ISBN: 978-020148567

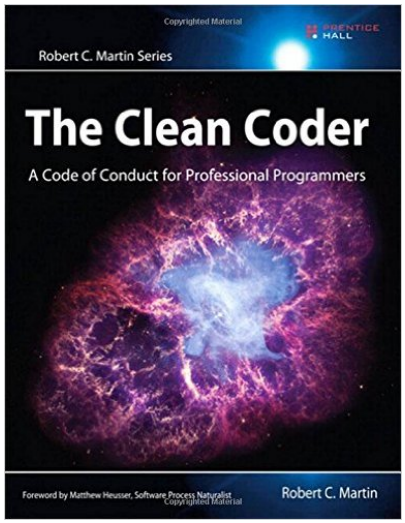


- Working Effectively with Legacy Code
  - Michael C. Feathers
  - Pearson Education
  - ISBN: 978-0131177055

Literatur



- Clean Code
  - Robert C. Martin
  - Pearson Education
  - ISBN: 978-0132350884



- The Clean Coder
  - Robert C. Martin
  - Pearson Education
  - ISBN: 978-0137081073