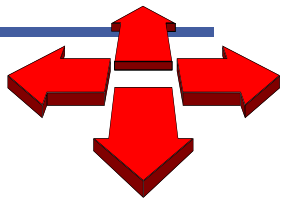
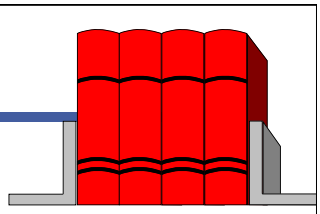
Refactoring



- Motivation: Software Aging
- Definition des Begriffs Refactoring
- „Üble Gerüche“ in Software
- Mechanik von Refactorings
- Werkzeugunterstützung für Refactorings

SE2 – OOPM – Teil 3 2

Literaturhinweise



Martin **Fowler**,*Refactoring – Improving the Design of Existing Code.*
430 Seiten - Addison-Wesley, 2000.
[Das Standard-Buch über Refactoring]

Stefan **Roock**, Martin **Lippert**, *Refactorings in großen Softwareprojekten*. 348 Seiten - dpunkt Verlag 2004.
[Weiterführend: große Refactorings, DB-Refactorings, u.ä.]

Schon mal gehört: Software verändert sich

- Software ist keine Prosa, die einmal geschrieben wird und dann unverändert bleibt.
- Software wird erweitert, korrigiert, gewartet, portiert, adaptiert, …
- Diese Arbeit wird von unterschiedlichen Personen vorgenommen (manchmal über Jahrzehnte).
- Für Software gibt es deshalb nur zwei Optionen:
  - Entweder sie wird gewartet.
  - Oder sie stirbt.

© Barnes, Kölling

„Any fool can write code that a computer can understand. Good programmers write code that humans can understand.“ (Fowler, 2000)

„Üble Gerüche“ in Software

- Software, die sich ändert, muss auch intern restrukturiert werden.
- Ein Entwurf, der heute gut war, ist morgen aufgrund neuer Anforderungen eventuell nicht mehr angemessen.
- Ein Entwurf, der uns **Änderungen erschwert**, sollte überdacht werden.

- Wann werden Änderungen erschwert? Hinweise sind so genannte „**üble Gerüche**“ (engl.: **bad smells**):
  - Eine Methode ist zu lang, um sie auf dem Bildschirm überblicken zu können.
  - Dieselbe Fallunterscheidung wird an mehreren Stellen in einem System vorgenommen.
- Generell sind Verstöße gegen bekannte Entwurfsregeln meist Kandidaten für üble Gerüche.

Begriffsdefinition Refactoring

Refactoring bedeutet, die interne Struktur einer Software zu verbessern, ohne ihr beobachtbares Verhalten zu verändern.

Was ist Refactoring demnach **nicht**?

- Das Einfügen zusätzlicher Funktionalität ist kein Refactoring.
- Das Beheben von Fehlern ist kein Refactoring.
- Auch eine Änderung am Layout der GUI ohne Änderung der Funktionalität ist kein Refactoring.

“Refactoring is a very specific technique, founded on using small behavior-preserving transformations (themselves called refactorings). If you are doing refactoring your system should not be broken for more than a few minutes at a time, and I don't see how you do it on something that doesn't have a well defined behavior.”

(Fowler, 2004)

http://martinfowler.com/bliki/RefactoringMalapropism.html

Refactoring, differenziert

Wir differenzieren den Begriff **Refactoring** etwas genauer:

- Als Bezeichnung für die **eine bestimmte Restrukturierung** eines Systems:
  - Konkreter Zusammenhang, konkrete Tätigkeit
  - Bsp.: „Umbenennen der Methode `fuegeEin` der Klasse `Liste` in `einfuegen`“, „Extrahieren des Interfaces `Melder` aus der Klasse `Hauptfenster`“
- Als Bezeichnung für eine **Klasse von Restrukturierungen**
  - Abstrakte Beschreibung, katalogisierbar
  - Bsp.: **Methode extrahieren** (engl.: Extract Method), **Interface extrahieren** (Extract Interface)
- Als Bezeichnung der **Tätigkeit des Restrukturierens**:
  - Im Sinne von „Beim Refactoring (sprich: beim Restrukturieren der Klassen) haben wir festgestellt, dass wir noch einige Leichen im Keller haben.“

Eine schlechte Metapher für Refactoring(T)

Nicht gut: Refactoring(T) ist wie „Bilder aufhängen“ (Dekorieren)

- „Unser Quelltext ist wie eine Wohnung. Wir wollen uns als Entwickler wohl fühlen und den Quelltext gut lesen und warten können. Wir halten deshalb Namenskonventionen ein, halten unsere Methoden kurz und die Klassen übersichtlich. Wir hübschen alles auf, damit es schöner ist. Bilder machen einen Raum freundlicher.“
- Diese Metapher wird dem Anspruch von Refactoring(T) nicht gerecht; es geht nicht um Schönheitspreise oder Eleganz, sondern um Notwendigkeit.

Eine bessere Metapher für Refactoring(T)

Refactoring(T) ist wie „Müll raustragen“ (Aufräumen)

- „Unser Quelltext ist wie eine Küche. Wie in einer Küche verrichten wir sehr viele alltägliche Dinge an unserem Quelltext. Wir kochen, indem wir ändern, anpassen, erweitern. In der Hitze des Gefechts räumen wir nicht immer alles gleich weg, es bleibt eine Menge liegen. So entsteht mit der Zeit viel Müll. Wenn wir den Müll nicht wegräumen, werden wir mit der Zeit keinen Spaß mehr in der Küche haben und in unserem eigenen Müll ersticken.“

Hintergrund: Broken Window Theory

- Nach der Broken-Window-Theory zieht Schmutz und Zerstörung weitere Verschmutzung und Zerstörungen an. In einem Versuch in den USA wurde dazu ein intaktes Auto mehrere Wochen auf einem Parkplatz abgestellt. Der Wagen wurde nicht beschädigt. Dann hat man gezielt eine Scheibe des Autos eingeschlagen. Diese kaputte Scheibe zog weitere Beschädigungen an: in sehr kurzer Zeit wurden die anderen Scheiben auch eingeschlagen, die Reifen gestohlen etc.
- In New York werden daher Graffitis sofort entfernt, wenn sie entdeckt werden. Man geht davon aus, dass es Sprayern leichter fällt, eine Wand mit einem Graffiti zu „verschönern“, wenn sie nicht die ersten sind.

Übler Geruch: Code-Duplizierung

- Wenn ein Stück Quelltext in identischer Form an mehreren Stellen eines Systems definiert ist, sprechen wir von **Code-Duplizierung**.
- Code-Duplizierung ist problematisch, weil
  - üblicherweise an einer Stelle nicht erkennbar ist, an welchen anderen Stellen derselbe Quelltext erscheint, und
  - Änderungen an einem der Duplikate eventuell auch an allen anderen Duplikaten ausgeführt werden müssen; dies kann bei der Wartung übersehen werden (implizite Kopplung).
- Code-Duplizierung ist ein **Zeichen niedriger Kohäsion**:
  - Wenn zwei Einheiten dieselbe (Teil-)Aufgabe erledigen, ist bei mindestens einer von beiden die Zuständigkeit falsch zugeordnet.

Übler Geruch: Große Einheiten

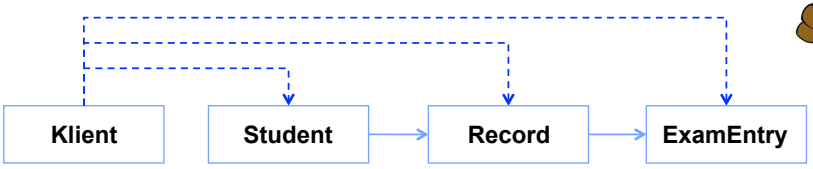
- Ein bekanntes Anti-Pattern - die „**Gott-Klasse**“:
  - Sie ist für alles zuständig und hält 90% des Quelltextes.

- Methoden, die **mehrere Bildschirmseiten** lang sind, sind schlecht wartbar.
- Klassen mit mehreren hundert Methoden oder Exemplarvariablen sind meist zu groß!

Übler Geruch: Verletzung des Law of Demeter

Verletzung des Law of Demeter:

```
student.getRecord().getExamEntry("SE2/2012").addResult(93);
```



Besser:

```
sc = new ScoreCard("SE2/2013");
…
```

<Eintragen der Noten in die Score-Card, bspw. interaktiv>

```
…
student.addScore(sc);
```


Mechanik von Refactorings(K)

- Als die **Mechanik** (engl.: mechanics) eines Refactorings(K) versteht man „**eine knappe und präzise schrittweise Beschreibung, wie das Refactoring ausgeführt werden soll.**“
- Der Anspruch bei den Schritten dieser Beschreibung ist meist, dass das System **nach jedem Schritt übersetzbar** ist.
- Die Schritte sollen deshalb **möglichst klein** gewählt werden.
- Bei Fowler sind bei jedem Refactoring(K) die Mechanics angegeben.

- Schritte einer solchen Mechanik können beispielsweise sein:
  - „Erzeuge eine neue Klasse mit einem bestimmten Namen.“
  - „Füge eine neue Methode hinzu.“
  - „Kopiere den Quelltext einer Methode in eine andere.“
  - „Entferne eine Methode.“

Mal wieder ein Zitat

- *„Some […] of the refactorings […] may seem obvious. But don‘t be fooled. Understanding the mechanics of such refactorings is the key to refactoring in a disciplined way.“*

  Erich Gamma, aus dem Vorwort zu Fowlers Buch

Mechanik von „Methode umbenennen“ (vereinfacht)

Ein häufig benutztes Refactoring(K) ist „**Methode umbenennen**“ (engl.: rename method).

Mechanik-Beschreibung:

- Deklare eine neue Methode mit dem neuen Namen. Kopiere den alten Quelltext in die neue Methode und mache die notwendigen Anpassungen.
- Übersetze.
- Ändere den Rumpf der alten Methode so, dass er die neue Methode ruft.
- Übersetze und teste.
- Finde alle Referenzen auf die alte Methode und ändere sie so, dass sie auf die neue verweisen. Übersetze und teste nach jeder Änderung.
- Entferne die alte Methode.
- Übersetze und teste.

Vereinfachung hier: keine Berücksichtigung von Ober- und Unterklassen

Mechanik von „Methode extrahieren“ (vereinfacht)

Ein Refactoring(K) zum Zergliedern von großen Methoden ist „**Methode extrahieren**“ (engl.: extract method). Dabei wird ein Teil einer Methode in eine neue Methode ausgelagert.

Mechanik-Beschreibung:

- Deklariere eine neue Methode mit einem sprechenden Namen.
- Übersetze.
- Kopiere den zu extrahierenden Abschnitt in die neue Methode.
- Suche in der neuen Methode nach Referenzen auf lokale Variablen der alten Methode. Mache diese entweder zu Parametern oder zu lokalen Variablen der neuen Methode.
- Übersetze und teste.
- Ersetze in der alten Methode den extrahierten Quelltext mit dem Aufruf der neuen Methode.
- Übersetze und teste.

Typen von Refactorings (Auswahl)

Es gibt eine ganze Reihe von etablierten Refactorings(K) (mit unterschiedlicher Komplexität):

- **Rename <Element>**: Ein Element (Paket, Klasse, Methode, Variable, Konstante) im Quelltext wird umbenannt.
- **Move <Element>**: Ein Element im Quelltext wird verschoben.
- **Change Signature**: Eine Methodensignatur wird um einen Parameter ergänzt/reduziert.
- **Bewege** Methode in Super-/Subtyp
- **Extrahiere** ein Interface, eine Methode, …
- **Ersetze** Subtyp durch Supertyp, Vererbung durch Delegation, …

Refactorings(K) zu Werten und Objekten

- Fowler beschreibt auch zwei Refactorings(K) zum Thema **Werte und Objekte**:
  - "Change Value to Reference"
  - "Change Reference to Value"
- Auszug aus "**Change Value to Reference**":
  - *"You can make a useful classification of objects in many systems: reference objects and value objects. **Reference object** are things like customer or account. Each object stands for one object in the real world, and you use the object identity to test whether they are equal. **Value objects** are things like date or money. They are defined entirely through their data values. You don't mind that copies exist; you may have hundreds of "1/1/2000" objects around your system. You do need to tell whether two of the objects are equal, so you need to override the equals methods (and the hashCode method too).*

    The decision between reference and value is not always clear. …"

Werkzeugunterstützung beim Refactoring(T)

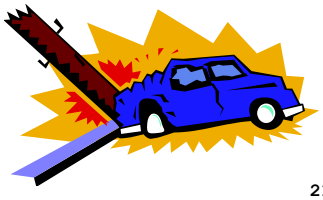
- Aufgrund der starken Mechanisierung von Refactorings(K) ist häufig eine Unterstützung durch Werkzeuge möglich.
- Seit etlichen Jahren bieten Entwicklungsumgebungen wie Eclipse oder IntelliJ IDEA umfangreiche Unterstützung an. Diese wird kontinuierlich weiter entwickelt.
- Der Grad der Unterstützung ist ebenfalls von der Programmiersprache abhängig:
  - Bei Java ist aufgrund des einfacheren Sprachmodells eine bessere Unterstützung möglich als beispielsweise in C++.

Vorgehen beim Refactoring(T)

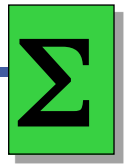
Aus (Fowler 2000), S. 7+8:

- „When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, **first refactor the program** to make it easy to add the feature, **then add the feature**.
- Before you start refactoring, **check that you have a solid suite of tests**. These tests must be self-checking."

- Beim Refactoring(T) besteht immer die Gefahr, dass wir neue Fehler in den Quelltext einbauen.
- Metapher: Refactoring(T) ohne ausreichende Testabdeckung ist wie Autofahren ohne Sicherheitsgurt!

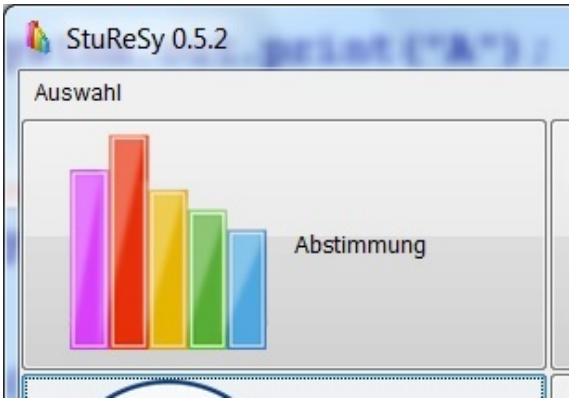


Zusammenfassung Refactoring



- Software wird fortlaufend verändert; dadurch entstehen leicht „üble Gerüche".
- Ein Refactoring ist eine Restrukturierung von Software, die deren Funktionalität erhält.
- Es gibt eine Reihe von etablierten Refactorings.
- Refactorings können in ihrer Mechanik beschrieben werden.
- Refactoring gehört zum Handwerkszeug der Softwareentwicklung.
- Eine gute Entwicklungsumgebung kann für eine gute Programmiersprache bei vielen Refactorings automatisierte Unterstützung anbieten.

Verständnisfragen



tinyurl.com/uhhse1