Refactoring

Infopoint 05.04.2006

Silver Scherrer

Inhalt

- Softwaretypen
- Gesetze der Software-Evolution
- Einführung anhand eines Beispiels
- Grundidee
- Vor- und Nachteile von Refaktorisieren
- Wann soll refaktorisiert werden und wann nicht?
- Prinzipien und Methoden
- Bad Smells
- Refaktorisieren und Testen

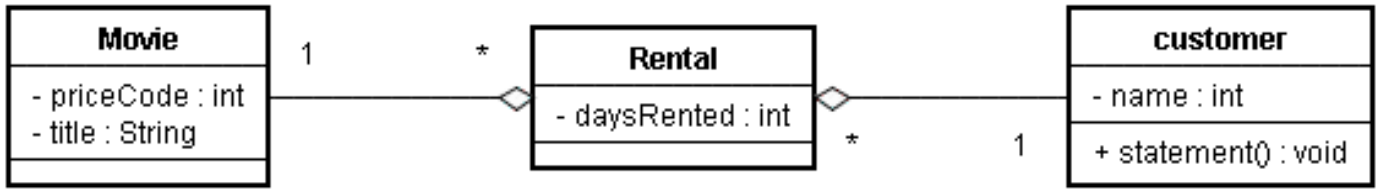
Softwaretypen

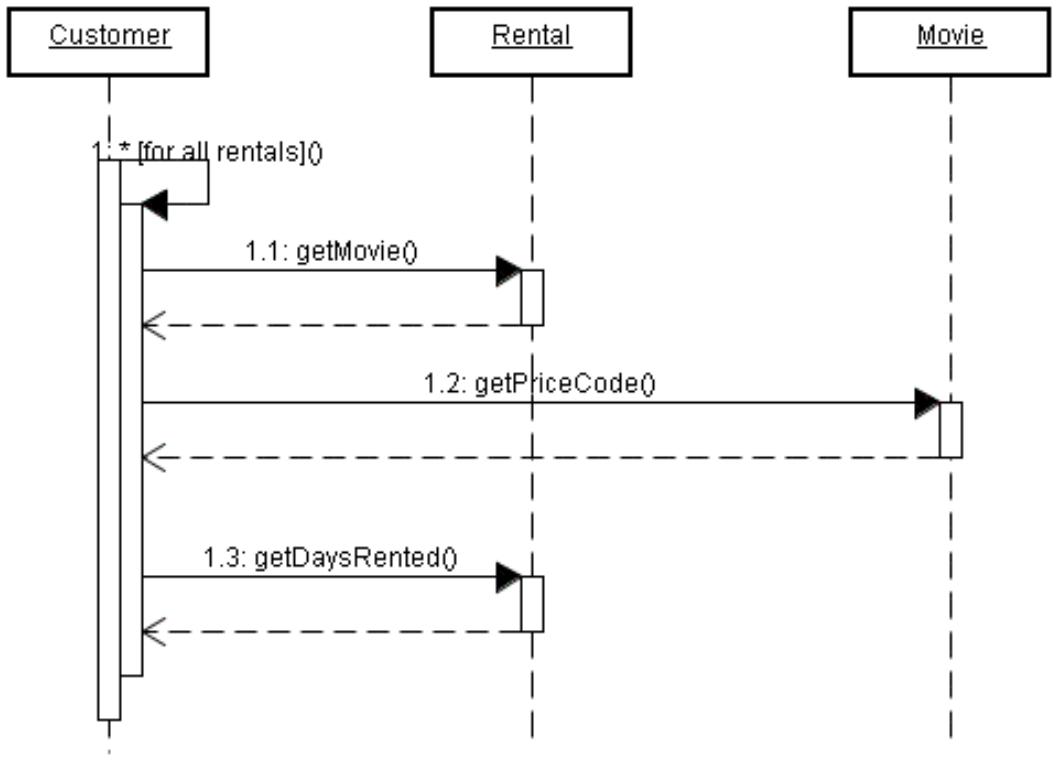
- Reale Software Systeme erliegen einer Evolution
- Lehman und Belady unterscheiden 3 Typen von Software:
  - S-Typ:
    - vollständig durch eine formale Spezifikation beschrieben
    - Entwicklung erfolgreich und abgeschlossen, wenn Spezifikationen erfüllt
    - Beispiel: Programme für numerische Berechnungen
  - E-Typ:
    - eine in der realen Welt eingebettete Anwendung
    - erfolgreich, wenn die Anwender mit der Software zufrieden sind
  - P-Typ:
    - löst ein spezifisches, abgegrenztes Problem
    - Beispiel: Programme zur Berechnung gegebener Modelle, wie etwa Festigkeitsmodelle in der Statik
    - heute an Bedeutung verloren, da meistens einem der beiden anderen Typen zugeordnet

Gesetze der Software-Evolution

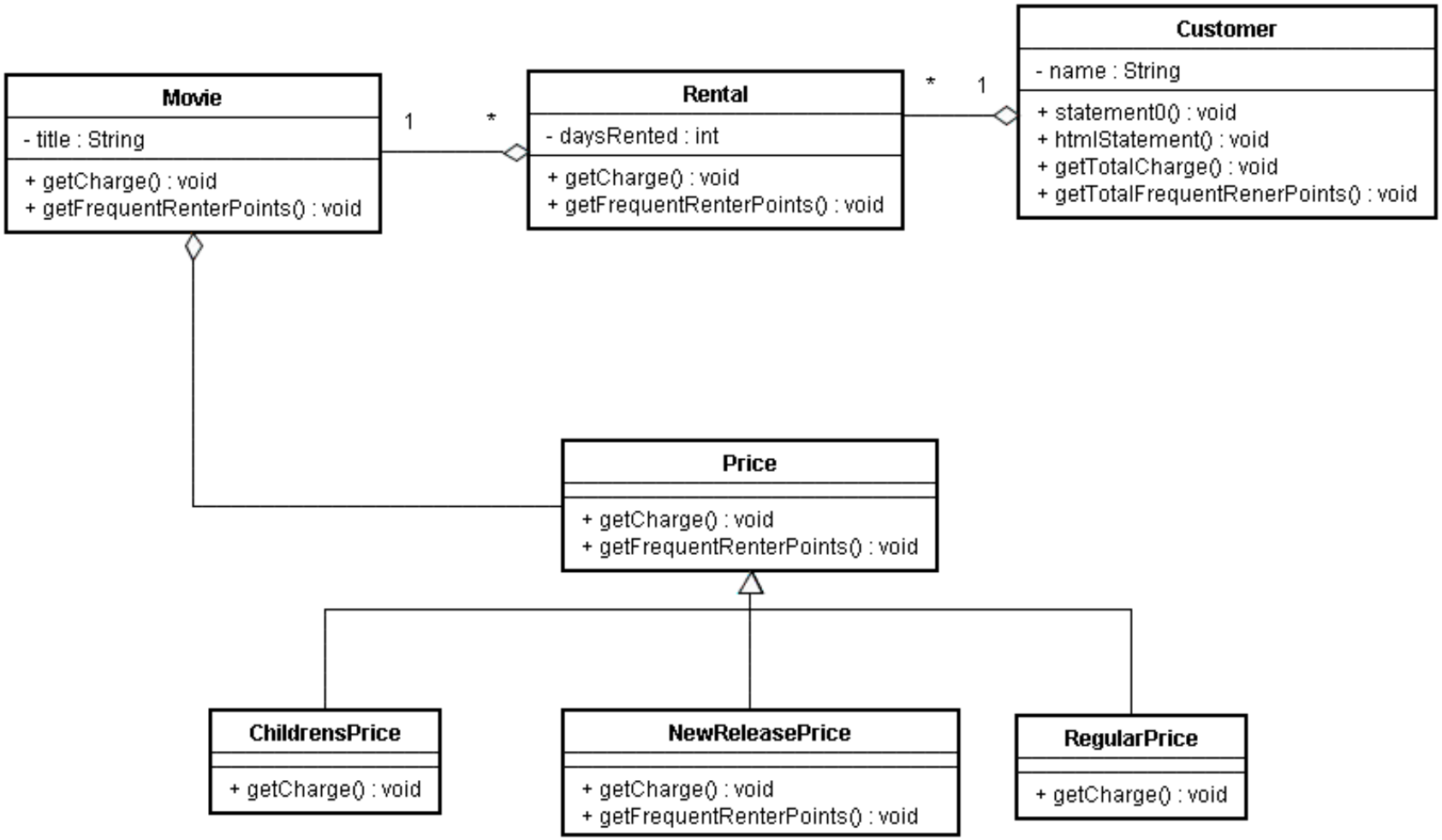
- **Gesetz der kontinuierlichen Veränderung (I).** Entweder verändert sich ein Programm während seiner gesamten Lebenszeit kontinuierlich, oder es verliert nach und nach seinen Nutzen.
- **Gesetz der zunehmenden Komplexität (II).** Die Struktur eines Programms wird immer schlechter, was zu einer kontinuierlichen Zunahme der Komplexität führt.
- **Selbstregulierung (III).** Die groben Trends bezüglich Wartbarkeit werden zu einem frühen Entwicklungszeitpunkt festgelegt und können nur in beschränktem Rahmen beeinflusst werden. (vgl: Massenträgheit)
- **Gesetz der Bewahrung der organisatorischen Stabilität (IV).** Der durchschnittliche Arbeitsaufwand, der in die Pflege eines Programms gesteckt wird, ist während der gesamten Lebenszeit des Programms nahezu konstant und unabhängig von den für die Entwicklung eingesetzten Ressourcen.
- **Gesetz der Erhaltung des bekannten Zustandes (V).** Zusätzliches Wachstum wird durch die Notwendigkeit begrenzt, die Vertrautheit in einem System zu erhalten. Dadurch sind die inkrementellen Änderungen in jedem Release während der gesamten Lebenszeit des Systems annähernd konstant.
- **Gesetz des kontinuierlichen Wachstums (VI).** Kontinuierliches Wachstum während der gesamten Lebensdauer ist unerlässlich, wenn das System gebrauchstauglich bleiben soll.
- **Gesetz der abnehmenden Qualität (VII).** Die Qualität eines E-Typ Systems sinkt mit zunehmender Entwicklungsdauer.
- **Feedback System (VIII).** E-Type Evolutionsprozesse sind multi-level, multi-loop und multi-agent Feedback Systeme

Einführung anhand eines Beispiels





Refaktorisiertes Klassendiagramm



Refactoring vs. Methodologien der Softwareentwicklung

„Im Design kann ich sehr schnell denken, aber meine Gedanken sind voller kleiner Löcher.“ [Fowler]

- Software verhält sich anders als physische Maschinen, denn wie der Name „soft“ schon sagt, ist Software viel weicher.
- Die ideale Lösung gibt es zu Beginn eines Software-Projekts noch gar nicht
- Mit Refaktorisieren führt der Weg über einfaches, leicht umstrukturierbares Design, das erst mit der Zeit zu einer kompletten Lösung heranwächst

Grundidee

„Die Verringerung der Codemenge lässt das System nicht schneller laufen, da die Auswirkungen auf das Verhalten des Programms selten gross sind. Die Verringerung der Codemenge macht aber einen grossen Unterschied, wenn es darum geht, den Code zu ändern.“ [Fowler]

- Ausdruck „Refactoring“ entstand in Smalltalk Kreisen
- Prozess, einem Softwaresystem eine bessere interne Struktur zu geben, ohne dessen externes Verhalten zu verändern (d.h. Performance-Optimierung gilt nicht als Refactoring)
- R. beabsichtigt in den meisten Fällen eine Reduktion der Komplexität eines Codeabschnittes oder eines Designs
- Der Code wird verbessert, nachdem er geschrieben wurde
- Schlechtes Design in gut strukturierten Code umwandeln
- Die einzelnen Schritte sind sehr einfach, aber das Ergebnis kann das Design enorm verbessern

Vorteile von Refaktorisieren

- Verbessert das Design
- Verbessert die Kapselung
- Macht den Code verständlicher und wartbar
- Hilft Software zu verstehen: Es gibt nichts Schlimmeres als Änderungen in Software, die man zu verstehen glaubt.
- Hilft Fehler zu finden: Refaktorisieren kann Fehler aufdecken, die sonst unerkannt geblieben wären.
- Hilft schneller zu programmieren:
  Je schlechter das Design ist, desto mehr Zeit wird für Erweiterungen benötigt.
- Verbessert die Wiederverwendbarkeit

Nachteile von Refaktorisieren

"If you want to refactor, the essential precondition is having solid tests." [Fowler]

- Erfordert Disziplin
- Zeitaufwand
- Sehr viele Tests sind notwendig.
- Performance: Refaktorisieren kann Software langsamer machen. Jeder Methodenaufruf und jede Instanzierung einer Klasse kostet Rechenzeit.
- Änderungen von öffentlichen Interfaces
- Software-Projektmanager muss von Refaktorisieren überzeugt sein
- Einarbeitung der Entwickler in den neuen Code. Grössere Refaktorisierungen müssen unbedingt mit allen betroffenen Entwicklern abgesprochen werden, damit man nicht gegeneinander arbeitet.

Wann soll refaktorisiert werden?

„Sie entscheiden nicht zu refaktorisieren, sondern Sie refaktorisieren, weil Sie etwas anderes machen wollen und das Refaktorisieren Ihnen dabei hilft.“
[Fowler]

- Dreierregel:
  Machen Sie etwas das erste Mal – tun Sie es einfach!
  Machen Sie etwas das zweite Mal – scheuen Sie die Wiederholung, aber machen Sie es trotzdem nochmals!
  Machen Sie etwas das dritte Mal – refaktorisieren Sie!
- Refaktorisieren beim Hinzufügen von Funktionen
- Refaktorisieren zum Beheben von Fehlern
- Refaktorisieren bei Code-Reviews

Wann soll nicht refaktorisiert werden?

„Denken Sie daran, dass Code im Wesentlichen funktionieren muss, bevor Sie ihn refaktorisieren können.“ [Fowler]

- Wenn der Code so viele Fehler hat, dass er erstens nicht funktioniert und zweitens nicht stabilisiert werden kann
- Kurz vor einem Fertigstellungstermin
- Vorsicht beim Ändern einer veröffentlichten Schnittstelle:
  - Abhilfe: die alte Schnittstelle verwendet die neue
  - Alte Schnittstelle auf *deprecated* setzen

Prinzipien und Methoden I

„Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Gute Programmierer schreiben Code, den Menschen verstehen.“ [Fowler]

- Umbenennen von Variablen, Attributen, Methoden, Klassen und Packages
- Methode extrahieren oder integrieren
- Variable einführen oder ersetzen
  - *Einführen von temporären Variablen*
  - *temporäre Variable durch Abfrage ersetzen*
  - *Extraktion einer Methode*
- Eigenschaften zwischen Objekten verschieben
  - *Methode / Feld verschieben*
  - *Klasse extrahieren / integrieren*

Prinzipien und Methoden II

- Umgang mit Generalisierung
  - *Unterklasse / Oberklasse extrahieren*
  - *Hierarchie abflachen*
  - *Superklasse integrieren*
- Daten organisieren
  - *Feld kapseln*
  - *Magische Zahlen durch symbolische Konstante ersetzen*
- Bedingte Anweisungen vereinfachen
  - *Bedingung zerlegen*
  - *Geschachtelte Bedingungen durch Wächterbedingungen ersetzen*
  - *Bedingten Ausdruck durch Polymorphismus ersetzen*
  - *Null-Objekte einführen*

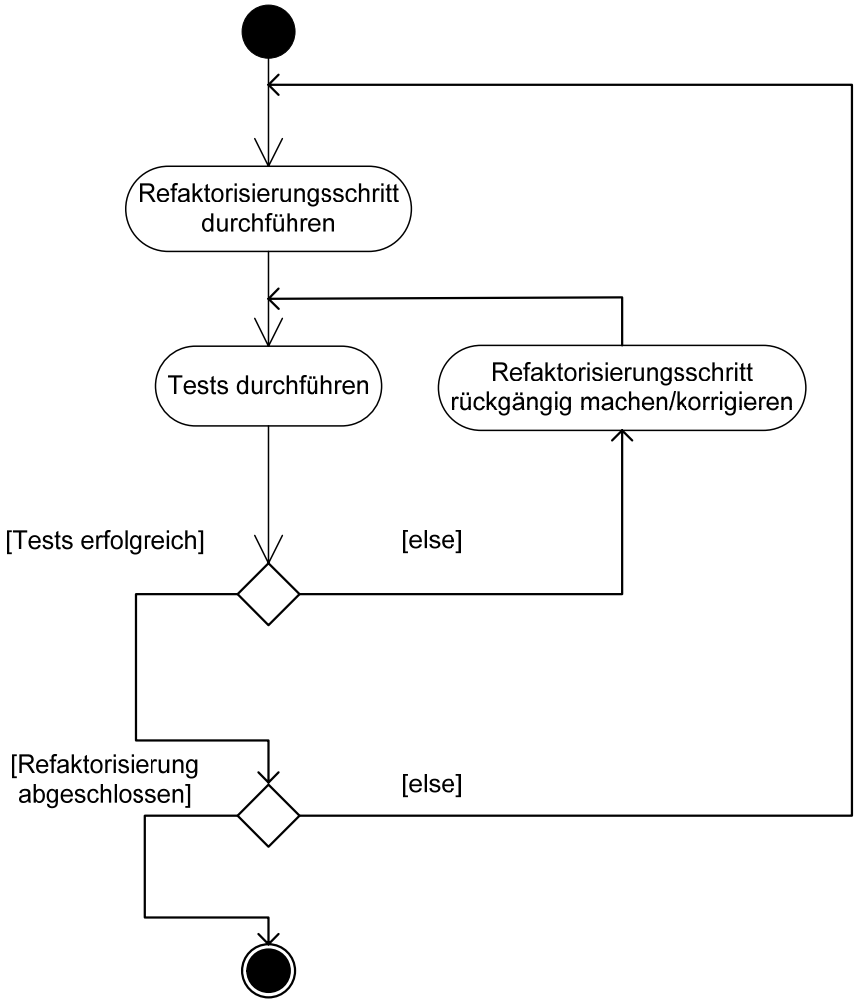
Bad Smells I

- „üble Gerüche“
- Indizien, Anhaltspunkte und Strukturen im Programmcode
- keine absolut präzise definierten Kriterien
- Stechen einem geübten Programmierer ins Auge

Bad Smells II

- Duplizierter Code
- Lange Methode
- Temporäre Variablen
- Grosse Klasse
- Lange Parameterliste
- Divergierende Änderungen
- Schrotkugeln herausoperieren
- Neid
- Datenklumpen
- Switch-Befehle
- Faule Klassen
- Spekulative Allgemeinheit
- Vermittler
- Kommentare

Refactoring und Testen



Referenzen

- Fowler, Martin. *Refactoring: Wie Sie das Design vorhandener Software verbessern* (München: Addison-Wesley, 2000)
- Wake, William C. *Refactoring Workbook* (Boston, Addison-Wesley, 2004)
- Fowler, Martin. *The New Methodology* *http://www.martinfowler.com/articles/newMethodology.html*
- *Lehman, M.M., L.A. Belady. Program Evolution: Processes of Software Change (London, Academic Press, 1985]*
- *Lehman, M.M. Software Evolution: Cause or Effect?* *http://www.cs.vu.nl/icsm2003/slides/lehman.pdf*
- Opdyke, William. *Refactoring Object-Oriented Frameworks.* Dissertation (Illinois, 1992) *ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z*
- *http://www.refactoring.com*