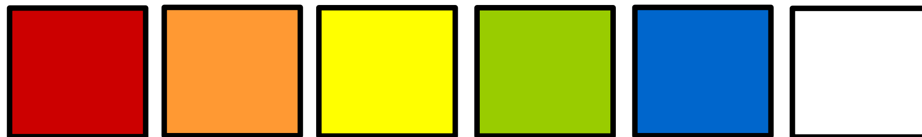


Arbeiten mit Altcode

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Ziele von Testen (aus dem letzten Foliensatz)

Freie Universität Berlin

- Traditionelles Testen:
  - Korrektheit/Zuverlässigkeit
    - +diverse nichtfunktionale Eigenschaften
- Weg dorthin:
  - Alle Register ziehen! z.B.:
  - Viel Zeit für das Testen nehmen
  - Spezialisierte Tester
  - Viel exploratives Testen

Na, super!
Und was mache ich, wenn ich schon Massen von nicht selbsttestendem Code habe???

- Agiles Testen:
  - Evolvierbarkeit
  - Korrektheit (+diverse ...)
- Weg dorthin:
  - **Selbsttestender Code**
  - mit wenig Testredundanz
    - weil die ebenfalls das Ändern erschweren würde
  - und hoher Test-Ablaufgeschwindigkeit
    - weil das Debugging mühsam wird, wenn man die Tests erst nach vielen Änderungen ausführt
  - Videozitat Fowler
    - 1;24:22-24:57

Freie Universität Berlin

Seufzen und gar nichts tun?

And if I want to get up on a high horse
and say you must do something,
I might be inclined to get on
that particular high horse.

- Martin Fowler über die Wichtigkeit selbstestenden Codes: eine der mächtigsten Erfindungen in der Softwaretechnik

Also: Man muss da was tun!
Aber wie?

Quelle [Legacy]

Freie Universität Berlin

- Michael C. Feathers: *"Working Effectively with Legacy Code"*, Prentice Hall, 2004
  - Buch, 430 Seiten
- Michael C. Feathers: *"Working Effectively with Legacy Code"*, objectmentor.com, 2002
  - PDF-Datei, 12 Seiten
- Diverse Präsentationen gleichen Titels
  - slideshare.net

Definition "Altcode" (legacy code)

- Code, für den Änderungen nicht durch Tests abgesichert sind
  - Michael C. Feathers: *"Most of the fear involved in making changes to large code bases is fear of introducing subtle bugs; fear of changing things inadvertently."*
  - *"To me, the difference is so critical, it overwhelms any other distinction."* [Legacy, Artikel S.1]

(Eine inzwischen im agilen Bereich gut akzeptierte Definition)

Altcode hat sehr ungünstige Eigenschaften

Freie Universität Berlin

- Jede Änderung dauert lange
  - denn man muss den Code zuvor sorgfältig verstehen
- Refactoring wird sehr selten gemacht
  - denn man könnte dabei ja Defekte einfügen
- Änderungen erzeugen häufiger subtile Defekte
  - weil man zu wenig Refactoring gemacht hat und also keine "cleane" Ausgangsbasis für die Änderung besitzt
- Man verbringt viel Zeit mit Debugging
  - weil Defekte oft subtil sind,
    weil die Codebasis schwer zu durchschauen ist,
    weil man wenig Refactoring gemacht hat
- Man verbringt viel Zeit mit Defektkorrekturen
  - denn a) man fügt viele Defekte ein und
  - b) jede Änderung dauert lange, also auch Defektkorrekturen

Ein Teufelskreis

Also?
Altcode in Clean Code verwandeln!

Freie Universität Berlin

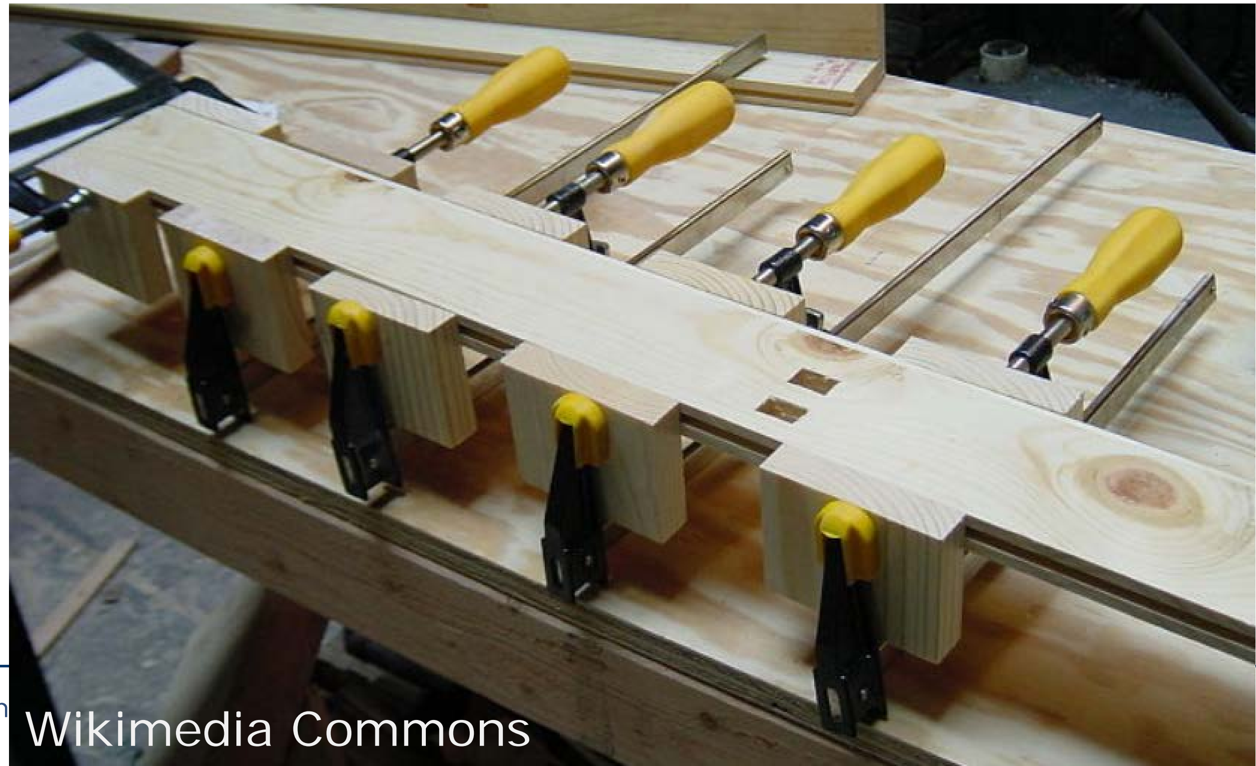
- Das geht nicht auf einmal
  - Viel zu viel Aufwand
- Aber: Jede Änderung ist Anlass, *die zu ändernde Stelle* mit Tests abzusichern
- Änderungen häufen sich meist in wenigen Bereichen
  - → nach einer Weile begegnet man seinen Tests immer wieder
  - → Schreiben unnötiger Tests wird vermieden
- Anlässe für Änderungen:
  - Funktionalität zufügen
  - Defekt korrigieren
  - Struktur verbessern
  - Effizienz verbessern
- Sehr verschieden!
- Gemeinsamkeit:
  - Fast alles Verhalten der SW muss gleich bleiben
  - Das ist leicht(er) zu testen!

Grundidee: Charakterisierungstest

- Ein Charakterisierungstest prüft, ob sich das Verhalten geändert hat
- Nicht wichtig:
  - ob Verhalten korrekt ist
  - ob Test genau 1 Anforderung beschreibt

Riesen-Erleichterung!

- Dient als "Schraubzwinge"
  - Werkstück zusammenhalten

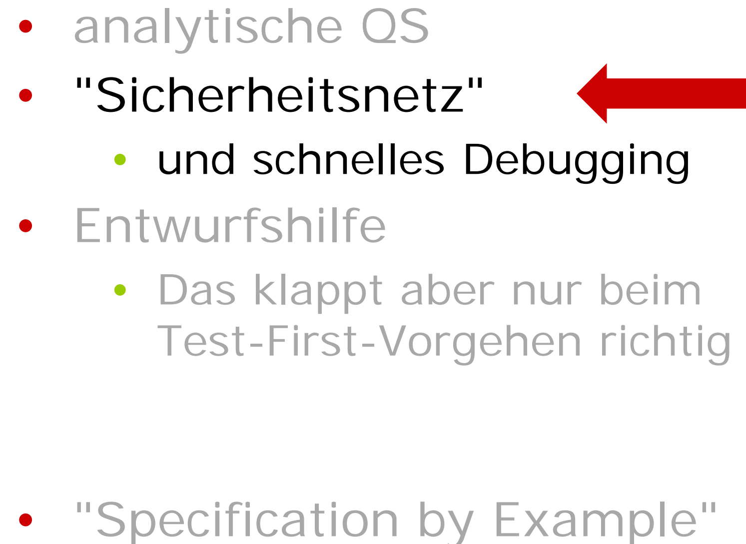


Wikimedia Commons

Wozu dienen automatische Modul- und Integrationstests?

WICHTIG!

- **Risiko reduzieren!**
- Welche Risiken?:
  - Code ist defekt
  - Code wird später defekt
    - insbes. bei Refactoring
  - Entwurf ist ungünstig
    - Das Schreiben der Tests hilft, einen guten Entwurf zu finden
      - insbes. gut entkoppelt
  - Code ist schwer zu verstehen
    - Test dient als Dokumentation oder Spezifikation



- Nutzen also:
  - analytische QS
  - "Sicherheitsnetz"
    - und schnelles Debugging
  - Entwurfshilfe
    - Das klappt aber nur beim Test-First-Vorgehen richtig
  - "Specification by Example"

Vorgehen für jede Änderung

Freie Universität Berlin

1. Änderungsstelle(n) finden
2. Teststelle dazu auswählen
   - wenn mehr als 1 nötig, ist die Änderung zu groß!
3. Teststelle mit Tests abdecken
   - dazu Abhängigkeiten aufbrechen
4. Änderung durchführen
5. Struktur verbessern
   - Refactoring

- Im Rest der Präsentation besprechen wir diese Schritte

Schritt 1: Änderungsstelle(n) finden

- An welchen Stellen muss man für diese Änderung überall in den Code greifen?
  - Feathers: "The amount of work involved varies with the degree of sickness in the code."
    - [Legacy, Artikel S.3]
  - Wähle ggf. den einfachsten Weg, nicht den "richtigsten"
    - denn sonst wird das Abdecken mit Tests schnell unrealistisch
    - Güteniveau steigt aber im Laufe der Zeit an (da mehr Tests vorhanden)
- "Einfachster Weg" heißt nicht Spaghetticode
  - (neuer Code kommt ggf. vielmehr in hübsche neue Methoden und Klassen)
- sondern nur "ohne vorheriges Umstrukturieren"
  - denn <u>das</u> ist zu schwierig mit Tests abzudecken

Schritt 2:
Teststelle auswählen

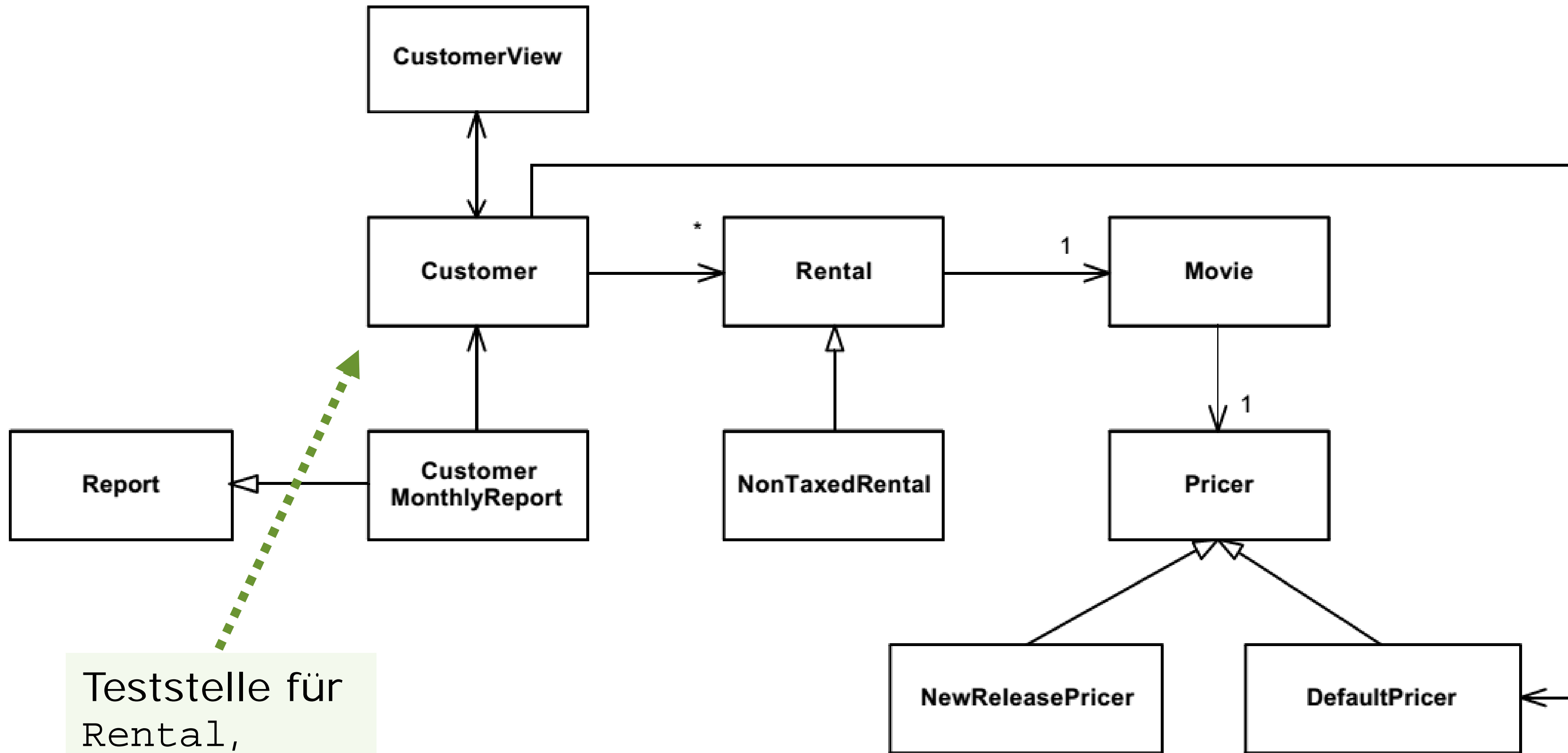
Freie Universität Berlin

- Eine *Teststelle* ist eine schmale Schnittstelle zum Prüfen einer Menge von Klassen, so dass gilt:
  1. Relevante Änderungen im Verhalten dieser Klassen sind an der Teststelle sichtbar
  2. Änderungen, die an der Teststelle nicht sichtbar sind, sind nicht relevant
- Mit etwas Glück besteht die Teststelle nur aus 1 Klasse

- Beispiel: Ein einfaches System für eine Leih-Videothek
  - wir wollen die Tarifregeln ändern ➔

Im [Legacy]-Artikel heißt die Teststelle "inflection point", also Knickstelle (die zwischen Innen und Außen trennt).

Beispiel: Videothek (komplettes Klassendiagramm)

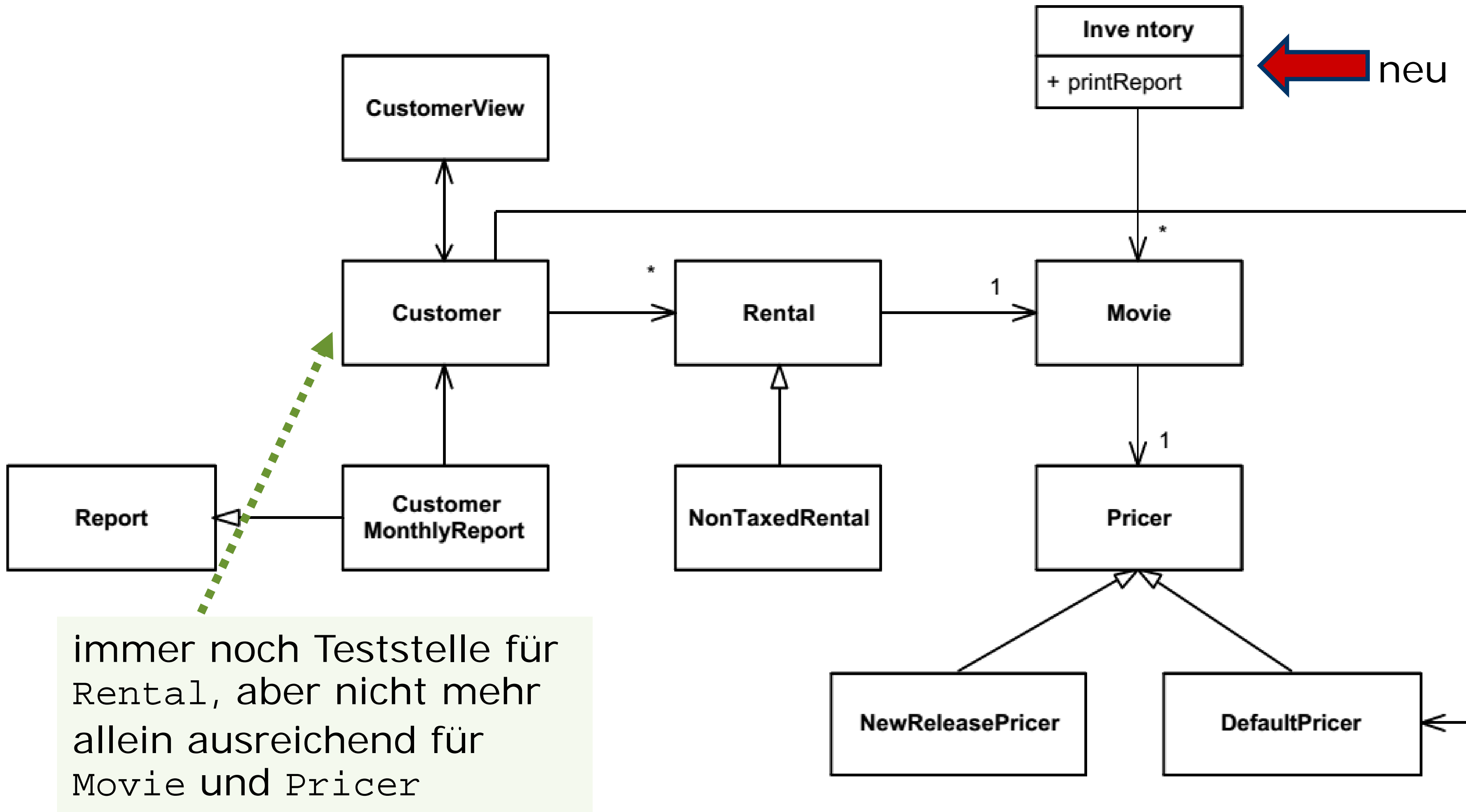


Teststelle für `Rental`, `Movie`, `Pricer`

Achtung:
Nie auf UML-Diagramme verlassen!

[Legacy, Artikel S.4]

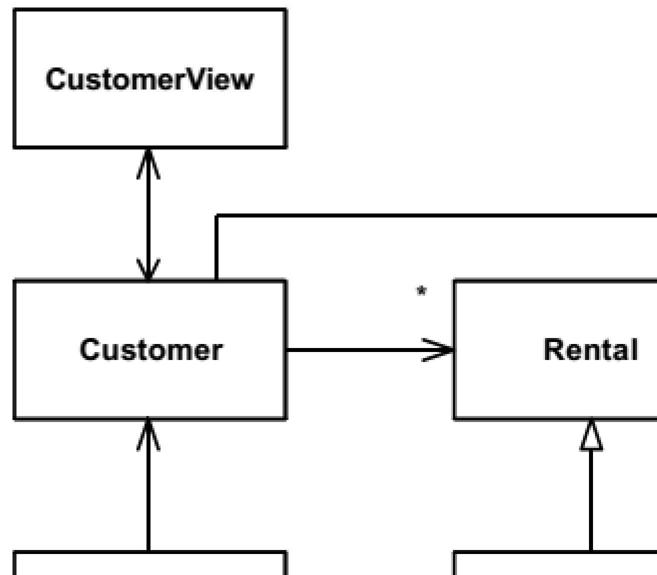
Beispiel: Videothek, erweitert



[Legacy, Artikel S.4]

Schritt 3.1: Teststelle abdecken
Äußere Abhängigkeiten

- Wir wollen also `Customer` in einer Testumgebung ausführen
  - leider hängt da noch `CustomerView` dran (UI):



  - → Dependency Inversion!

Refactoring:

- Verwandle `class CustomerView` in `class StandardCustomerView implements CustomerView`
  - "Extract Interface"
- Bilde für den Test leere Implementierung (Dummy `NullCustomerView`) oder benutze Mocking-Bibliothek

Schritt 3.2: Teststelle abdecken Innere Abhängigkeiten

Freie Universität Berlin

- Angenommen, im `Customer`-Konstruktor steht
  ```
  archiver = new FileArchiver(customerName);
  ```
- Einfachster Weg:
  1. Fabrikmethode in `Customer` einführen:
  ```
  archiver = createArchiver();
  ```
  und
  ```
  protected Archiver createArchiver() {
    return new FileArchiver(customerName); }
  ```
- 2. getestet wird Unterklasse `TestingCustomer`, die createArchiver() überschreibt:
  ```
  protected Archiver createArchiver() {
    return new NullArchiver(); }
  ```
- Injektion statt Fabrikmethode wäre schicker, verlangt aber Änderungen an viel mehr Stellen
  - → erst ggf. später machen

Archivieren sei laaaangsam

Schritt 3.3: Teststelle abdecken
Sonstige Abhängigkeiten

Freie Universität Berlin

- Alle benutzten "globalen Variablen" zählen auch zu den Abhängigkeiten
  - Singletons
  - statische Methoden und Variablen u.ä.
- Wir müssen für Tests für alle davon einen definierten Zustand herstellen

Das kann anstrengend werden.

Schritt 3.4: Teststelle abdecken
Prüfbarkeit herstellen

- Oft sind weitere Änderungen nötig,
  um die *Wirkung* von Eingaben feststellen zu können
  - "sensing"
- Ansätze:
  - Ergebnisprüfung
  - Spying
- Auch hier wieder zuerst "einfache" Ansätze bevorzugen
  - Dies sind meist *Zufügungen* (nicht Änderungen),
    also nicht so heikel

Schritt 3.5: Tests schreiben

- Wir exerzieren nur die Teststelle
- Alle normalen Heuristiken für Testentwurf gelten
  - Äquivalenzklassen, Randwerte etc.
- Wir brauchen die Semantik aber nicht komplett zu verstehen
  - "Korrekt" ist, was *heute* herauskommt (Charakterisierungstest)

- Im Prinzip kann man solche Tests automatisch erzeugen
  - Augen offen halten nach passenden Werkzeugen
  - Redundanz vermeiden!
- Für mehr Verständnis kann man die Wirkungen von Codeänderungen hinter der Teststelle untersuchen

Schritt 3.5:
Tests schreiben: Wie viele?

Freie Universität Berlin

- In einfachen Fällen (gut verständlicher Code, Einzeländerung):
  - Nach Risikogefühl
    - Nur, wenn gründliche Abdeckung zu umfangreich würde
- In mittleren Fällen (weitere Änderungen erwartet):
  - Hohe Anweisungsabdeckung anstreben
- In schweren Fällen (komplexe, wichtige Logik):
  - Per Mutationstests die Gründlichkeit prüfen
    - Werkzeuge: für Java, für Python

Schritt 4: Änderung durchführen

Freie Universität Berlin

- Das ist jetzt normale Programmierung
- Geht leichter mit besserem Semantikverständnis
- Unterwegs Tests häufig laufen lassen!
  - Die Versagen können an unerwarteten Stellen passieren

Freie Universität Berlin

Schritt 5: Struktur verbessern (Refactoring)

- Jetzt haben wir Tests und trauen uns deshalb nicht nur die Änderung,
- sondern auch Refactorings
  - jedenfalls an und hinter der Teststelle
  - Juhu!
- Worauf zielen wir mit denen?
- Als erstes auf weitere Testerleichterung
  - insbesondere der Fähigkeit zur Isolation
  - Dazu helfen "Fugen"
    - siehe gleich
- Dann auf allgemeine Entwurfsverbesserungen
  - Davon gibt es mehrere Modi
    - siehe jetzt

Arten von Refactoring

WICHTIG!

- Martin Fowler: "Workflows of Refactoring", OOP 2014
  - Video
- TDD-Refactoring (2:28)
  - Entwurf nachholen
- Igitt-Refactoring (7:25)
  - Müll aufräumen
- Ich-versteh-das-nicht-Refactoring (10:25)
  - Verständnis in der Struktur festhalten
- *Immer: Richtigen Zeitpkt. f. Änderungen finden (12:38)*
  - *Refactoring geht nur bei Grün!*
- Wir-hätten-das-so-machen-sollen-Refactoring (14:40)
  - Jetzt-kommende Features vorbereiten
- Geplantes Refact. (17:30)
  - nur für Verbesserungen, die man noch nicht gelernt hat, nebenbei zu tun
- Langfristige Änderungen (19:14)
  - zum allmählichen Erreichen großer Entwurfsänderungen
- *Zweck: Design Stamina (21:58-26:55)*
  - *Refactoring dient ökonomischen Zielen*

Schritt 5: Fugen (seams)

Freie Universität Berlin

- Eine *Fuge* (seam) ist ein Ort im Programm, an dem sich das Programmverhalten ändern lässt, ohne dort den Quelltext zu ändern. [Legacy, S.30ff]
  - z.B. jede Stelle mit einem Aufruf an ein Objekt, das sich austauschen lässt.
- Ein *Einfügepunkt* (enabling point) ist ein Ort im Programm, an dem solch ein Objektaustausch vorgenommen werden kann.
    - [Legacy, S.36]
  - z.B. ein Konstruktorparameter für eine Injektion
  - (Ohne Einfügepunkt ist eine Fuge keine Fuge)
    - Bei der Kachel fällt beides zusammen!

Schritt 5: Arten von Fugen

Freie Universität Berlin

- [Legacy, S.29-43]
- Fugen können im Prinzip überall da geschaffen werden, wo Referenzen aufgelöst werden, z.B.:
  1. Quelltext-Präprozessor (C, C++)
  2. Binder (Linker)
  3. Dynamische Code-Lademechanismen
     - z.B. java.lang.ClassLoader, Python importlib
  4. Objekterzeugung: Konstruktor, Fabrikmethode, IoC-Container
  5. Objekteinspeisung: Konstruktorparameter, setter, Methodenparameter, IoC-Container

- Auch zum *Schaffen* von Fugen wollen wir möglichst wenig Code ändern!
  - Mechanismen 1, 2, 3 sind dafür zwar geeignet, aber für spätere Leser kompliziert zu verstehen.
  - Maß halten!

Nochmal:
Altcode in Clean Code verwandeln?

Freie Universität Berlin

- Das geht **nicht** auf einmal
  - Viel zu viel Aufwand
- Aber: Jede Änderung ist Anlass, *die zu ändernde Stelle* mit Tests abzusichern
- Änderungen häufen sich meist in wenigen Bereichen
  - → nach einer Weile begegnet man seinen Tests immer wieder
  - → Schreiben unnötiger Tests wird vermieden
- Anlässe für Änderungen:
  - Funktionalität zufügen
  - Defekt korrigieren
  - Struktur verbessern
  - Effizienz verbessern
- Sehr verschieden!
- Gemeinsamkeit:
  - Fast alles Verhalten der SW muss gleich bleiben
  - Das ist leicht(er) zu testen!

Hausaufgabe

Freie Universität Berlin

- Ein paar Integrationstests schaffen an einer Stelle, wo bislang keine Testbarkeit gegeben ist, aber eine Änderung ansteht:
  - Änderungsstelle(n) lokalisieren
  - Geeignete Teststelle lokalisieren
  - Abhängigkeiten aufbrechen
  - Tests schreiben
  - Änderung durchführen *(oder auch nicht, falls noch nicht akut)*
  - Refactoring hinter der Teststelle

- Nicht *zu* ehrgeizig werden
- Aber eine geeignete Stelle wird es doch geben, oder?

Thank you!