Testgetriebene Softwareentwicklung

Fachhochschule Heilbronn

13. November 2006

Johannes Link

E-Mail: **business@johanneslink.net**
Internet: **johanneslink.net**
Weblog: **jlink.blogger.de**

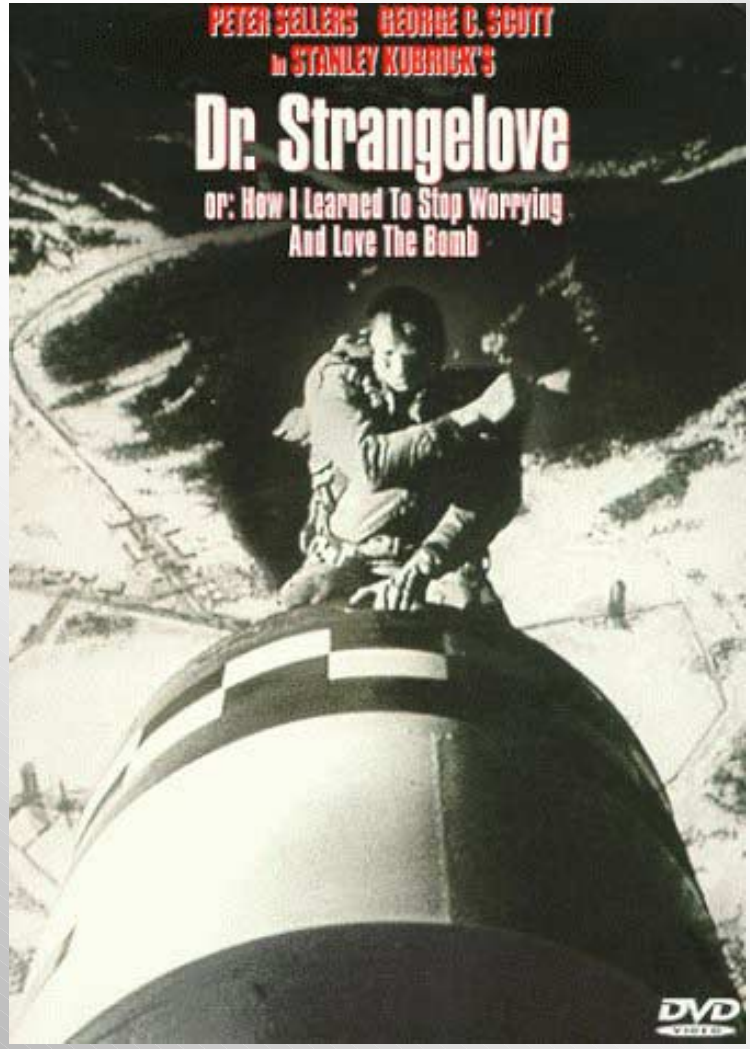
Qualität

Gerald M. Weinberg:

„Quality is value to some person“

„Real quality improvement always starts with knowing what your customers want.”

Dr. Seltsam oder wie ich lernte, die Bombe zu lieben



Dr. Seltsam oder wie ich lernte, die Bombe zu lieben

*„General Turkidson, Als Sie die so genannten **Zuverlässigkeitstests** einführten, versicherten Sie mir, es sei völlig unmöglich, dass so etwas jemals eintreten könne."*

„Ich muss sagen, dass ich es nicht sehr gerecht finde, ein ganzes Programm zu verdammen wegen eines kleinen Versehens, Sir."

Softwarequalität

- Zwei Qualitätsarten:
  - funktionale Qualität, d.h. Funktionalität und Fehlerfreiheit für die einwandfreie Benutzung
  - strukturelle Qualität, d.h. Design und Codestruktur für die nahtlose Weiterentwicklung
- Erfolgreiche Software muss beide Qualitäten besitzen
  - ohne funktionale Qualität keine Benutzerakzeptanz
  - ohne strukturelle Qualität Weiterentwicklung schwierig
- Automatisierte Tests sind eines der Mittel, um Qualität zu erreichen

Lessons Learned in Software Testing...

- Qualität kann nicht nachträglich in die Software „hineingetestet" werden!
- Vollständiges Testen ist unmöglich.
- Testautomatisierung ist Softwareentwicklung.
- Testen beeinflusst Design und Architektur.
- Manuelles Testen wird *nicht* überflüssig.

Was ist testgetriebene Entwicklung?

Testgetriebene Programmierung:
Motiviere jede Verhaltensänderung am Code durch einen automatisierten Test.

Refactoring:
Bringe den Code immer in die "einfache Form".

Häufige Integration:
Integriere den Code so häufig wie nötig.

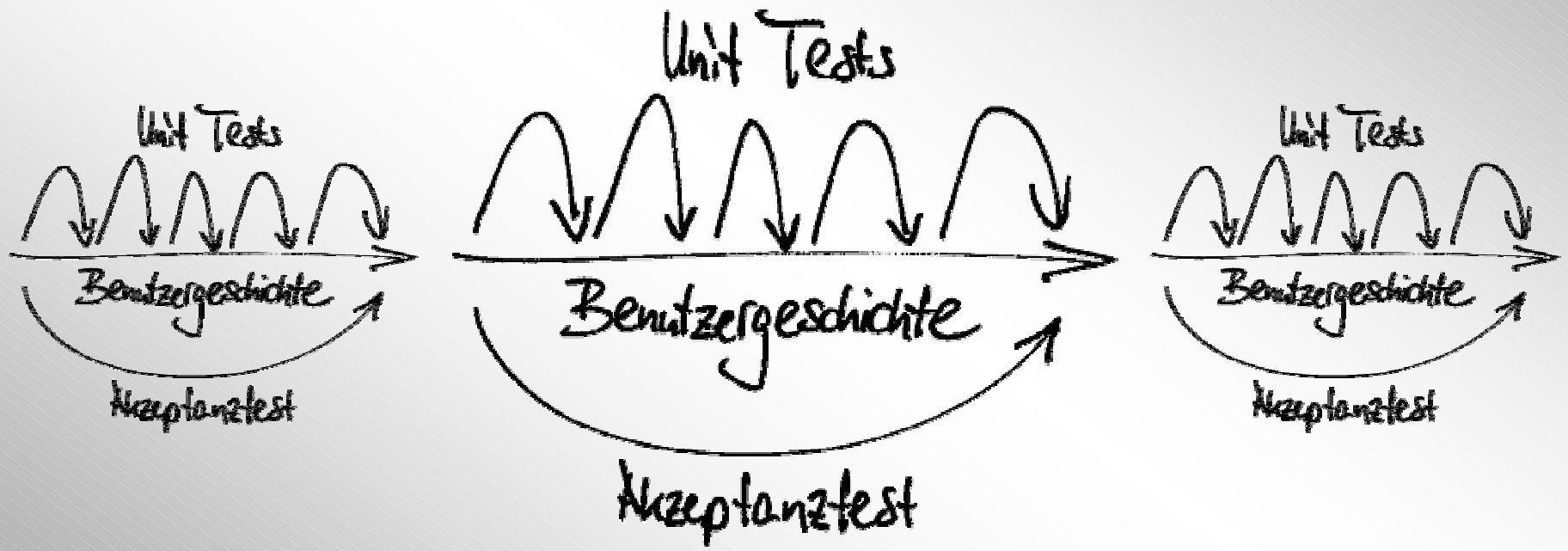
Warum testgetriebene Entwicklung?

- Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken.
- Tests sichern den Erhalt der vorhandenen Funktionen bei Erweiterung und Überarbeitung.
- Refactoring verlängert die produktive Lebensdauer einer Software.
- Code lässt sich im Nachhinein oft nur schlecht testen.

Zwei „Testgranularitäten“

- Im Kleinen: Entwickler schreiben Unit Tests, um ihren eigenen Code zu überprüfen.
- Im Großen: „Der Kunde“ spezifiziert Akzeptanztests, welche die Erfüllung seiner funktionalen Anforderungen verifizieren.

Testgetriebene Entwicklung im Kleinen...



Entwickler schreiben Unit Tests

- geben uns konkretes Feedback
- ermöglichen sichere Änderungen
- sichern Erhalt der vorhandenen Funktionalität
- müssen bei jeder Code-Integration zu 100% laufen

Unit Tests können funktionale Tests auf Systemebene nicht ersetzen!

Effizientes Testen

- möglichst zeitnah zur Programmierung
- automatisiert und damit wiederholbar
- muss Spaß machen
- Testen so oft und so einfach wie Kompilieren
- Fehler finden, nicht Fehlerfreiheit beweisen

JUnit - Testframework für Java

- Java-Framework zum Schreiben und Ausführen automatischer Unit Tests
- Ist auch für zahlreiche andere Programmiersprachen erhältlich
- http://www.junit.org

Testcode = Quellcode

```
import org.junit.*;
import static org.junit.Assert.*;

public class EuroTest {
    private Euro two;

    @Before public void initialize() {
        two = new Euro(2.00);
    }

    @Test public void amount() {
        assertTrue(2.00 == two.getAmount());
    }

    @Test public void adding() {
        Euro sum = two.add(two);
        assertEquals(new Euro(4.00), sum);
        assertEquals(new Euro(2.00), two);
    }
}
```

Testrunner

Java - EuroTest.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

JUnit

Finished after 0,031 seconds

Runs: 2/2 Errors: 0 Failures: 0

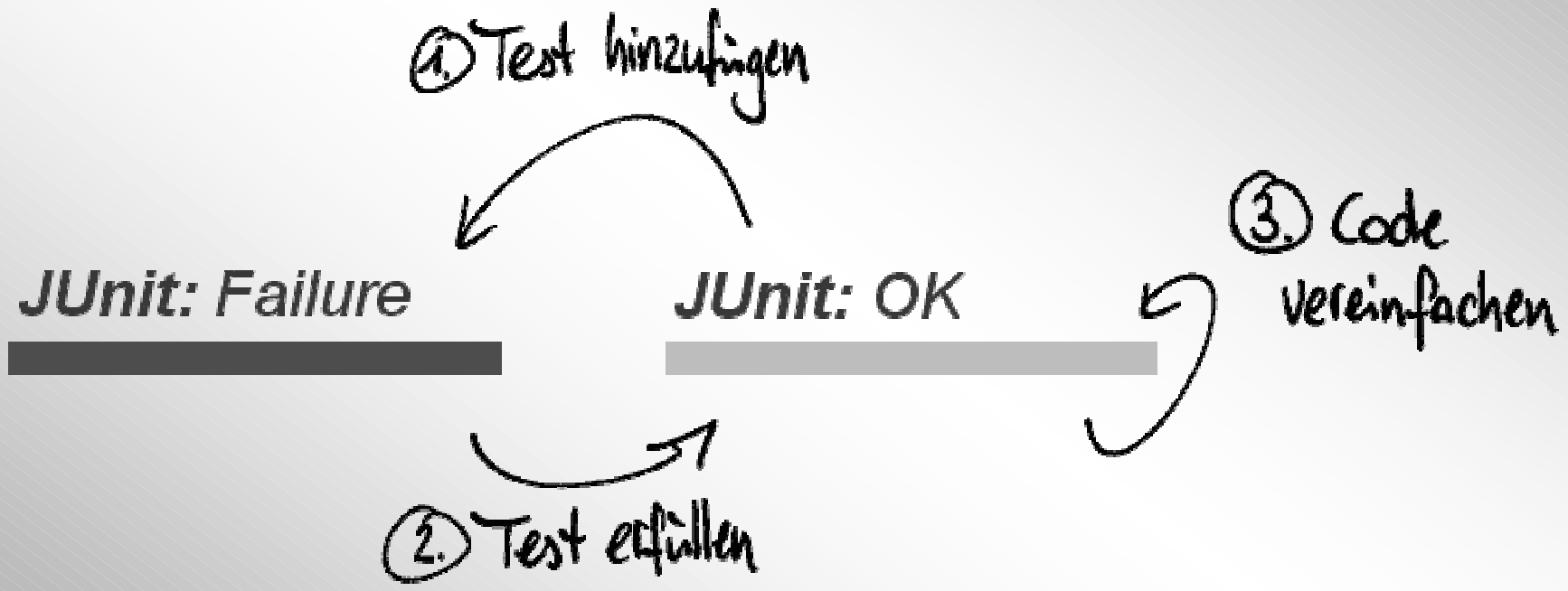
tdd.EuroTest [Runner: JUnit 4]
- amount
- adding

Failure Trace

EuroTest.java

```
package tdd;

import org.junit.*;
import static org.junit.Assert.*;

public class EuroTest {

    private Euro two;

    @Before
    public void initialize() {
        two = new Euro(2.00);
    }

    @Test
    public void amount() {
        assertTrue(2.00 == two.getAmou
    }
```

Test/Code/Refactor – Zyklus (1)



Test/Code/Refactor – Zyklus (2)

grün-rot: Schreibe einen Test, der zunächst fehlschlagen sollte. Schreibe gerade soviel Code, dass der Test kompiliert.

rot-grün: Schreibe gerade soviel Code, dass alle Tests laufen.

grün-grün: Eliminiere Duplikation und andere üble Codegerüche.

Wir überlegen uns erste Testfälle

Auswahl des nächsten Testfalls

- Erzeuge neues Konto (***`Account`***) für Kunden
- Mache eine Einzahlung (***`deposit`***)
- Mache eine Abhebung (***`withdraw`***)
- Überweisung zwischen zwei Konten

Wir entwerfen einen Test, der zunächst fehlschlagen sollte

```
public class AccountTest {
  @Test
  public void newAccount() {
      Account account = new Account("Customer");
      assertEquals("Customer", account.getCustomer());
      assertEquals(0, account.getBalance());
  }
}
```

Wir schreiben gerade soviel Code, dass sich der Test übersetzen lässt

```
public class Account {
   public Account(String customer) {
   }
   public String getCustomer() {
       return null;
   }
   public int getBalance() {
       return 0;
   }
}
```

Wir prüfen, ob der Test fehlschlägt

JUnit (AccountTest)

Runs: 1/1 Errors: 0 Failures: 1

Failures | Hierarchy

testCreateAccount(samplecode.AccountTest)

Failure Trace

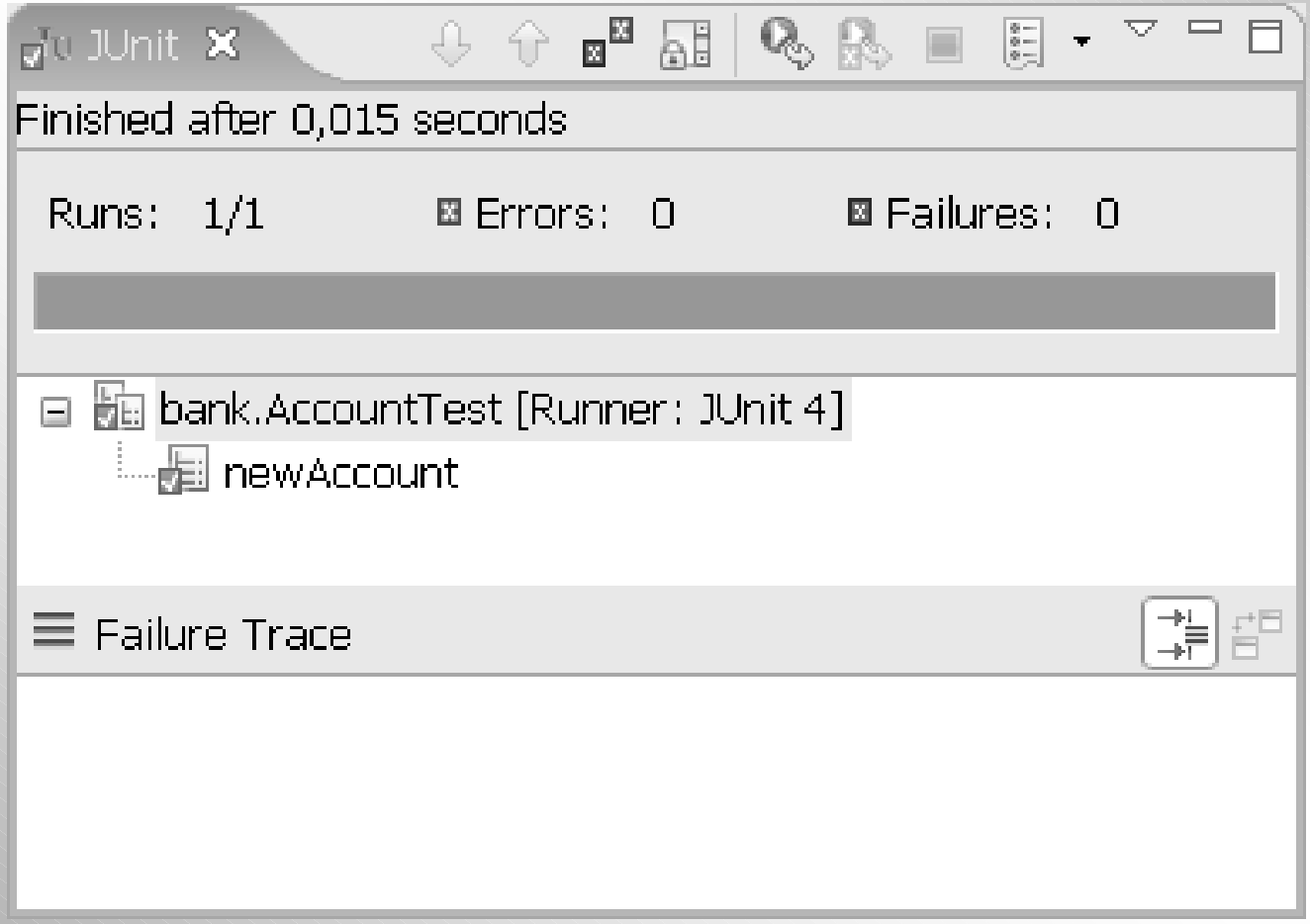
```
junit.framework.ComparisonFailure: expected:<Customer> but was:<null>
  at samplecode.AccountTest.testCreateAccount(AccountTest.java:8)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorIr
  at junit.framework.TestResult$1.protect(TestResult.java:106)
```

Wir schreiben gerade soviel Code, dass der Test erfüllt sein sollte

```
public class Account {
   public Account(String customer) {
   }
   public String getCustomer() {
      return "Customer";
   }
   public int getBalance() {
      return 0;
   }
}
```

Make it work!

Wir prüfen, ob der Test durchläuft



Wir entfernen Duplikation – Aber wo ist sie?

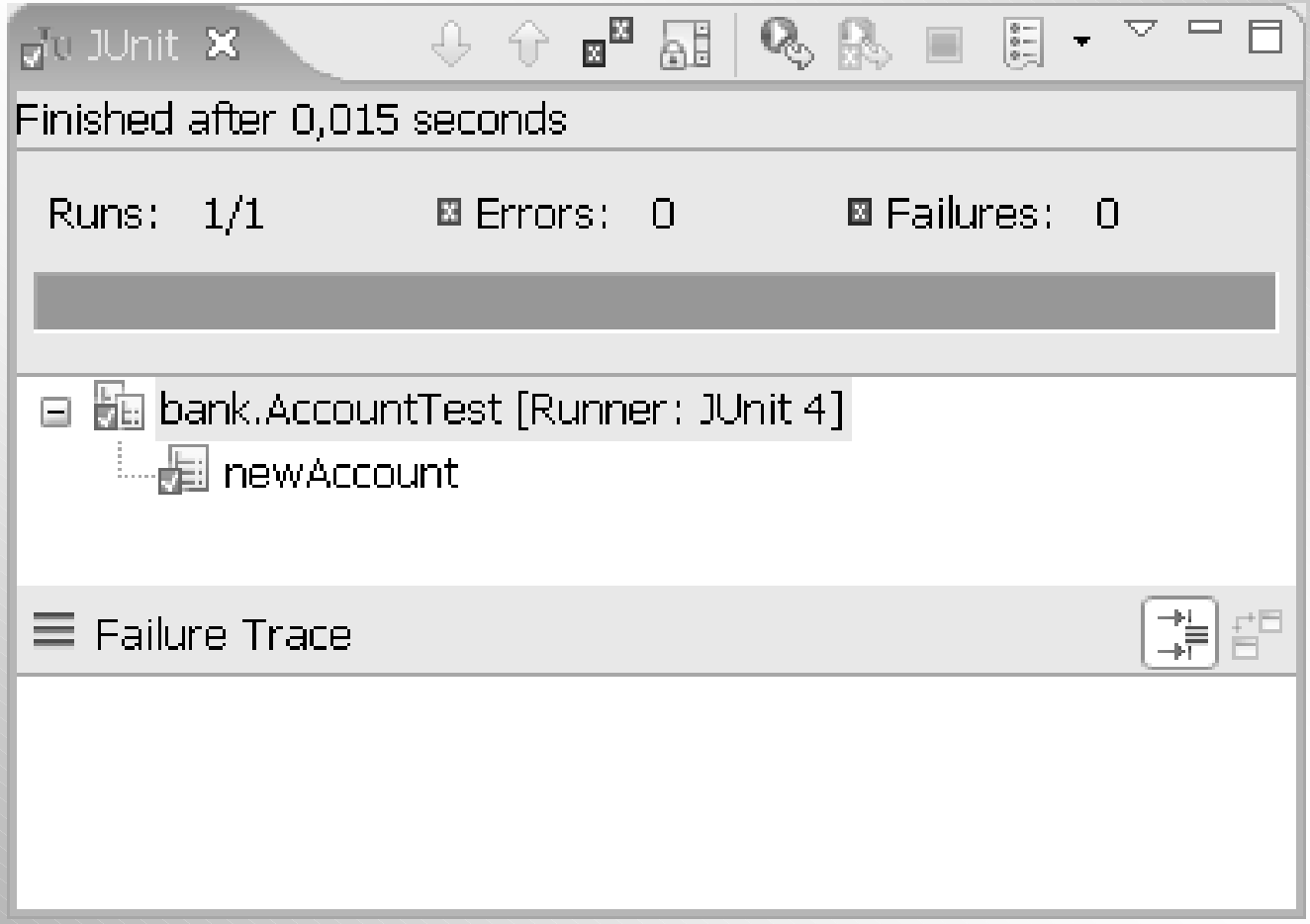
```
public class Account {
   public Account(String customer) {
   }
   public String getCustomer() {
       return "Customer";
   }
   public int getBalance() {
       return 0;
   }
}

public class AccountTest extends TestCase {
  public void testCreateAccount() {
      Account account = new Account("Customer");
      assertEquals("Customer", account.getCustomer());
      assertEquals(0, account.getBalance());
  }
}
```

Wir entfernen Duplikation

```
public class Account {
  private String customer;
  public Account(String customer) {
     this.customer = customer;
  }
  public String getCustomer() {
     return customer;
  }
  public int getBalance() {return 0;}
}
```

Make it right!

Wir prüfen, ob der Test weiterhin läuft



Tests und Code im Wechselspiel

- Der fehlschlagende Test entscheidet, welchen Code wir als nächstes schreiben, um die Entwicklung der Programmlogik voranzutreiben.
- Wir entscheiden anhand des bisher geschriebenen Code, welchen Test wir als nächstes angehen, um die Entwicklung des Designs weiter voranzutreiben.

Auswahl des nächsten Testfalls

- Erzeuge neues Konto für Kunden
- Mache eine Einzahlung
- Mache eine Abhebung
- Überweisung zwischen zwei Konten

Nächster Test: Einzahlen

```
public class AccountTest...
  @Test
  public void deposit() {
      Account account = new Account("Customer");
      account.deposit(100);
      assertEquals(100, account.getBalance());
      account.deposit(50);
      assertEquals(150, account.getBalance());
  }
```

```
public class Account...
  private int balance = 0;
  public int getBalance() {
      return balance;
  }
  public void deposit(int amount) {
      balance += amount;
  }
```

Rolle der Unit Tests

- Tests zur Qualitätssicherung
  Die Tests sichern die vorhandene Funktionalität
- Test-First führt zu »Evolutionärem Design«
  Das Systemdesign entsteht Stück für Stück.
- Tests zur Schnittstellendefinition
  Der Test verwendet schon Klassen und Methoden, bevor sie überhaupt definiert sind.
- Tests zur Modularisierung
  Testbarkeit erfordert Entkopplung der Programmteile.
- Tests als ausführbare Spezifikation und Dokumentation

Evolutionäres Design

- Geplantes Design versucht heutige und zukünftige Anforderungen auf einen Schlag abzudecken.
- Mit evolutionärem Design können wir die heutigen Anforderungen in kleinen Schritten erfüllen.
- Refactoring hält das Design für zukünftige Anforderungen offen.

⇨ Erfolgreiche Software wird weiterentwickelt!

Refactoring erhält strukturelle Qualität

- Definition: „Eine Änderung an der internen Struktur eines Programms, um es leichter verständlich und besser modifizierbar zu machen, ohne dabei sein beobachtbares Verhalten zu ändern.“ [Fowler 99]
- Wir refaktorisieren,
  - um das Design zu verbessern
  - um das Programm leichter verständlich zu machen
  - um zukünftige Änderungen am Code zu erleichtern
  - um der Entropie entgegen zu wirken

Refactoring-Ziel: Einfache Form

Design ist einfach, wenn der Code ...

- ... alle seine Tests erfüllt.
- ... jede Intention der Programmierer ausdrückt.
- ... für die beteiligten Entwickler verständlich ist.
- ... keine duplizierte Logik enthält.
- ... möglichst wenig Klassen und Methoden umfasst.

Reihenfolge entscheidend!

Übelriechender Code

Code Smells: Indizien für notwendiges Refactoring

- duplizierte Logik
- lange Funktionen
- Kommentare
- switch-Ketten, verschachtelte if-then-else
- Code, der seine Intention nicht ausdrückt
- Neid auf die Features einer anderen Klasse
- Datenklassen ohne wirkliches Verhalten
- ...

Refactoring - Vorgehen

- Refactoring findet ständig statt.
- Kleine Schritte
- Entweder Refactoring oder neue Funktionalität
- Unit Tests sind das Fangnetz des Refactoring.
- Oft ist auch ein Refactoring der Tests notwendig.

Eclipse: Eingebaute Refactorings

- Rename / Move Class
- Rename Method / Variable
- Extract Method / Variable / Constant
- Inline Method
- Change Method Signature
- Extract Local Variable / Constant
- Convert Local Variable to Field
- Extract Interface
- Push Up / Pull Down Methods / Variables
- ...

Häufige Integration

- Je häufiger integriert wird,
  - ... desto geringer ist der Integrationsaufwand
  - ... desto früher werden Probleme entdeckt
- Ein komplettes System steht ständig zur Verfügung
- Integration mindestens einmal pro Tag und Entwickler
- Automatisierte Tests sichern die Integrität

⇨ Späte Integration (Big-Bang-Integration) verschiebt das Risiko ans Projektende!

Integrationszüge

1. Integrieren Sie Ihre Änderungen in die vorhandene Codebasis.
   Update, Merge, (lokale) Tests
2. Erzeugen Sie einen neuen getesteten Build.
3. Versionieren Sie den aktuellen Stand.

Voraussetzungen:

- Integrationsmaschine zur Referenz mit dem letzten sauberen Stand
- Build-Skript zur vollautomatisierten Erstellung des Gesamtsystems
- Konfigurationsmanagement zur Versionierung aller Arbeitsdateien

CruiseControl

Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe

cruisecontrol
continuous integration toolkit

Project
– STATUS PAGE –

Latest Build
28/04/2005 15:53:29 (build.30)
28/04/2005 15:42:36 (build.29)
28/04/2005 15:39:21
28/04/2005 15:32:29
28/04/2005 15:28:58
28/04/2005 15:25:28
28/04/2005 15:21:59
28/04/2005 15:18:29
28/04/2005 15:14:57
28/04/2005 15:11:08
28/04/2005 15:07:09
28/04/2005 15:07:09

Build Results | Test Results | XML Log File | Metrics | Control Panel

BUILD COMPLETE - build.30

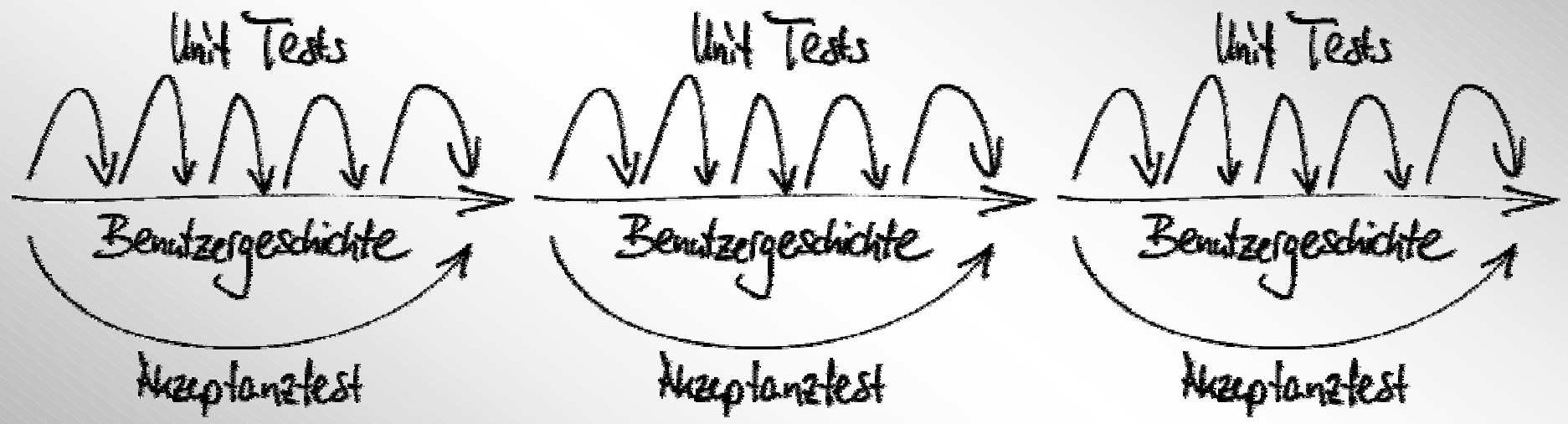
Date of build: 04/28/2005 15:53:29
Time to build: 10 minutes 12 seconds
Last changed: 04/28/2005 15:42:39
Last log entry: Teilnehmer reaktivieren refaktorisiert

Build Artifacts

Errors/Warnings: (126)

cvs server: Updating TopLink 9.03
cvs server: Updating IVS
cvs server: Updating IVS/TestdateienManuellerImport
cvs server: Updating IVS/archive
cvs server: Updating IVS/export
cvs server: Updating IVS/launcher
cvs server: Updating IVS/lib
cvs server: Updating IVS/logging
cvs server: Updating IVS/logging/0309051039
cvs server: Updating IVS/logging/0411301239
cvs server: Updating IVS/logging/0411301242
cvs server: Updating IVS/logging/0411301254
cvs server: Updating IVS/logging/0411301255
cvs server: Updating IVS/logging/0412091534
cvs server: Updating IVS/logging/0412091536
cvs server: Updating IVS/logging/0412091548
cvs server: Updating IVS/logging/0412091550
cvs server: Updating IVS/logging/0412091551
cvs server: Updating IVS/logging/0412102149
cvs server: Updating IVS/logging/0412211835

Testgetriebene Entwicklung im Großen...



Systemtests / Akzeptanztests

- testen das System als Ganzes
- sollten der Implementierung der betreffenden Anforderung vorausgehen
- dienen als Abnahmetests für die Vertragserfüllung
- müssen die Sprache des Kunden sprechen
- müssen vom Kunden geändert und erweitert werden können

=> Aber: Entwickler sind für Automatisierung verantwortlich

FIT - Framework for Integrated Test

- Framework zum Schreiben und Ausführen automatischer Akzeptanztests
- Testdaten werden tabellarisch erstellt (in HTML, mit Excel oder im Wiki)
- Anbindung ans System in Java
- Portierung für aktuelle Sprachen verfügbar
- http://fit.c2.com

FIT...

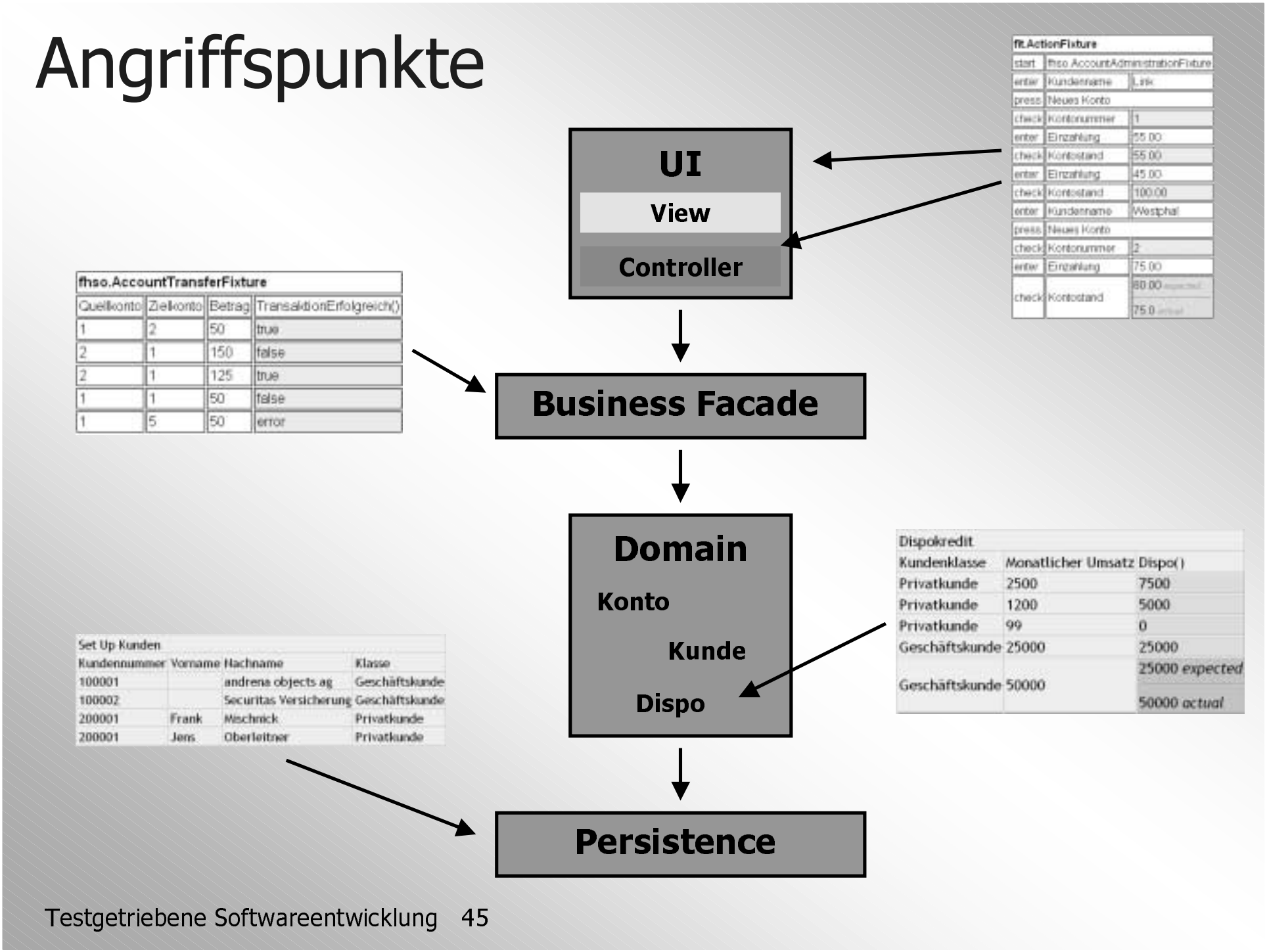
1. liest HTML-Dokumente ein,
2. führt die enthaltenen Testfälle aus
3. reichert HTML um Testergebnis an
4. gibt HTML-Dokumente wieder aus

FIT: Funktionsweise

fit.ActionFixture		
start	fhso.AccountAdministrationFixture	
enter	Kundenname	Link
press	Neues Konto	
check	Kontonummer	1
enter	Einzahlung	55.00
check	Kontostand	55.00
enter	Einzahlung	45.00
check	Kontostand	100.00
enter	Kundenname	Westphal
press	Neues Konto	
check	Kontonummer	2
enter	Einzahlung	75.00
check	Kontostand	80.00

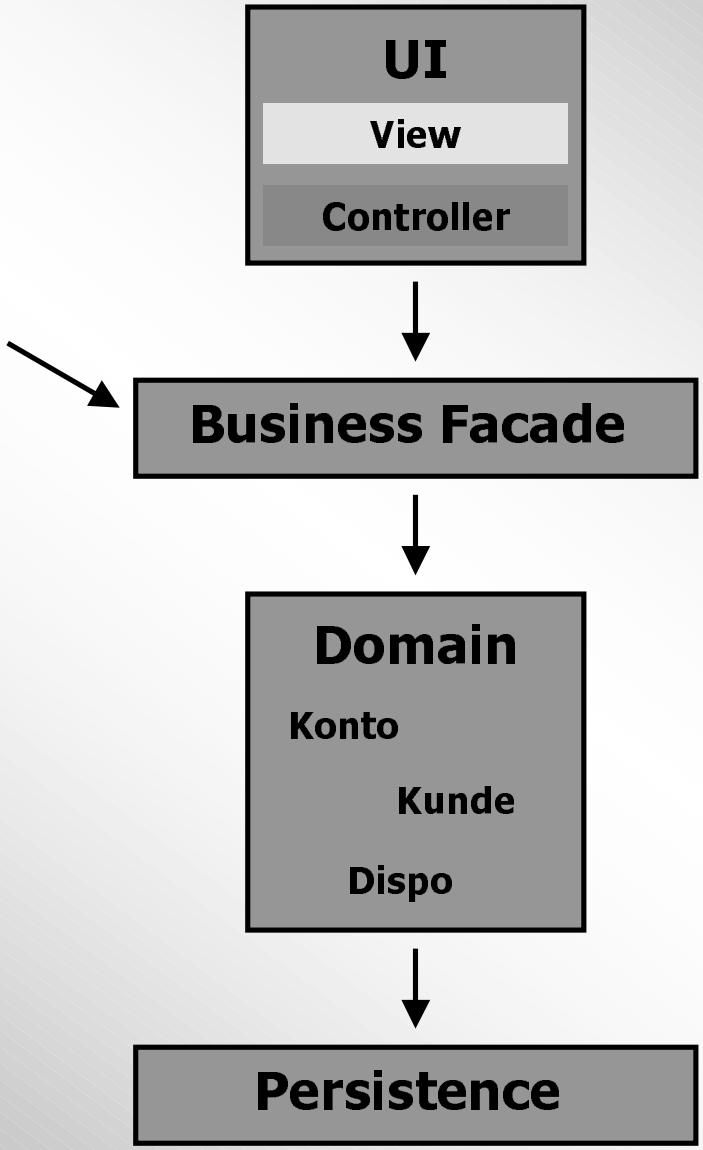
fit.ActionFixture		
start	fhso.AccountAdministrationFixture	
enter	Kundenname	Link
press	Neues Konto	
check	Kontonummer	1
enter	Einzahlung	55.00
check	Kontostand	55.00
enter	Einzahlung	45.00
check	Kontostand	100.00
enter	Kundenname	Westphal
press	Neues Konto	
check	Kontonummer	2
enter	Einzahlung	75.00
check	Kontostand	80.00 *expected* 75.0 *actual*

Angriffspunkte



Angriffspunkte

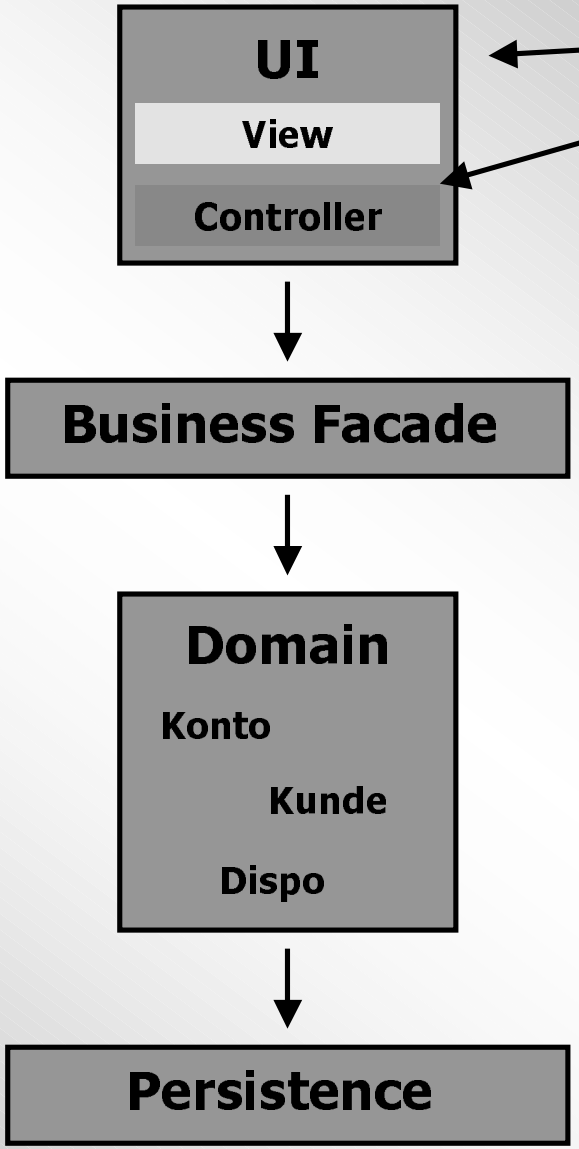
fhso.AccountTransferFixture			
Quellkonto	Zielkonto	Betrag	TransaktionErfolgreich()
1	2	50	true
2	1	150	false
2	1	125	true
1	1	50	false
1	5	50	error



FIT: Business Facade

fhso.AccountTransferFixture			
Quellkonto	Zielkonto	Betrag	TransaktionErfolgreich()
1	2	50	true
2	1	150	false
2	1	125	true
1	1	50	false
1	5	50	error

Angriffspunkte

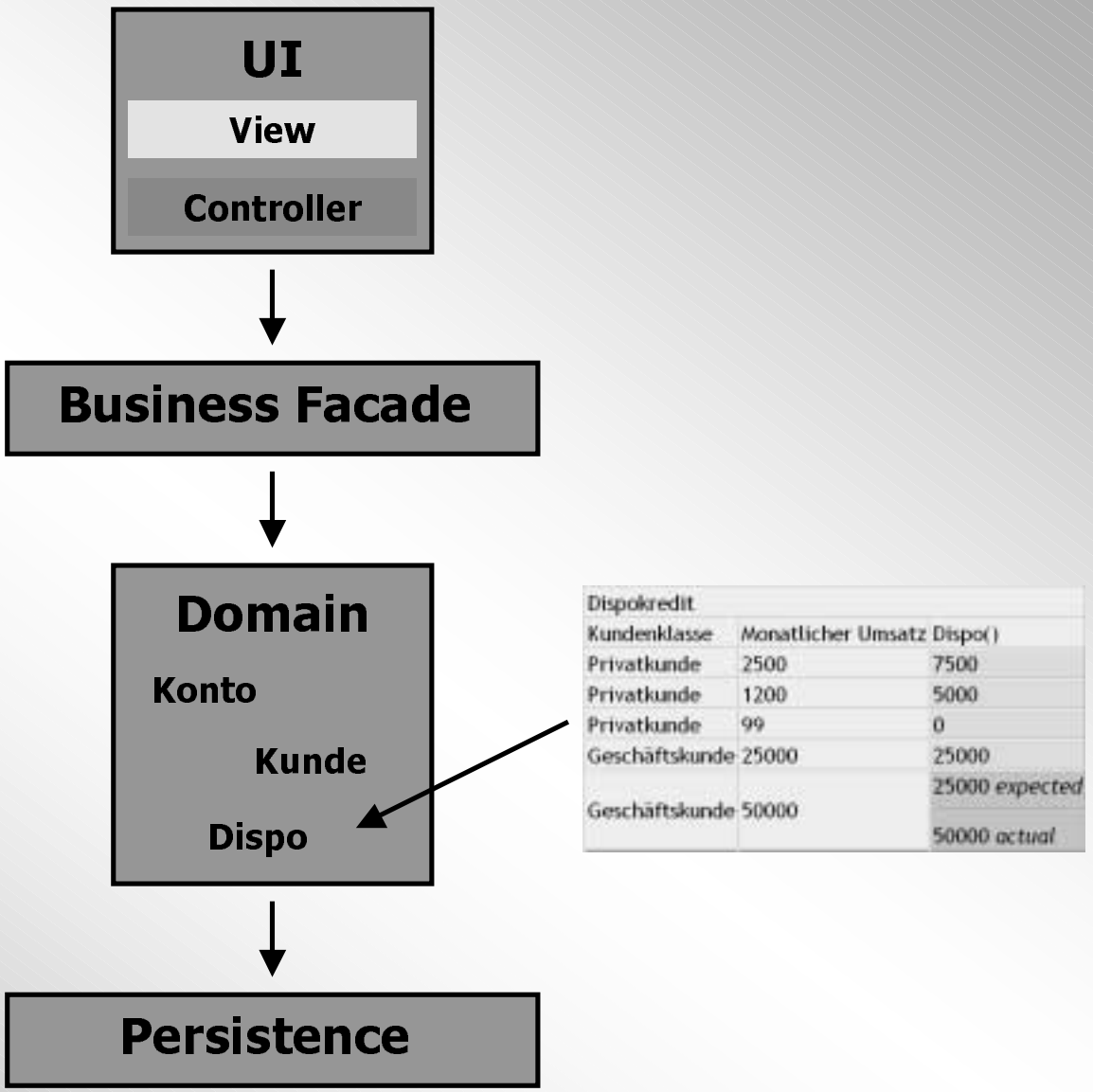


fit.ActionFixture		
start	fhso.AccountAdministrationFixture	
enter	Kundenname	Link
press	Neues Konto	
check	Kontonummer	1
enter	Einzahlung	55.00
check	Kontostand	55.00
enter	Einzahlung	45.00
check	Kontostand	100.00
enter	Kundenname	Westphal
press	Neues Konto	
check	Kontonummer	2
enter	Einzahlung	75.00
check	Kontostand	80.00 expected 75.0 actual

FIT: User Interface

fit.ActionFixture		
start	fhso.AccountAdministrationFixture	
enter	Kundenname	Link
press	Neues Konto	
check	Kontonummer	1
enter	Einzahlung	55.00
check	Kontostand	55.00
enter	Einzahlung	45.00
check	Kontostand	100.00
enter	Kundenname	Westphal
press	Neues Konto	
check	Kontonummer	2
enter	Einzahlung	75.00
check	Kontostand	80.00 *expected* 75.0 *actual*

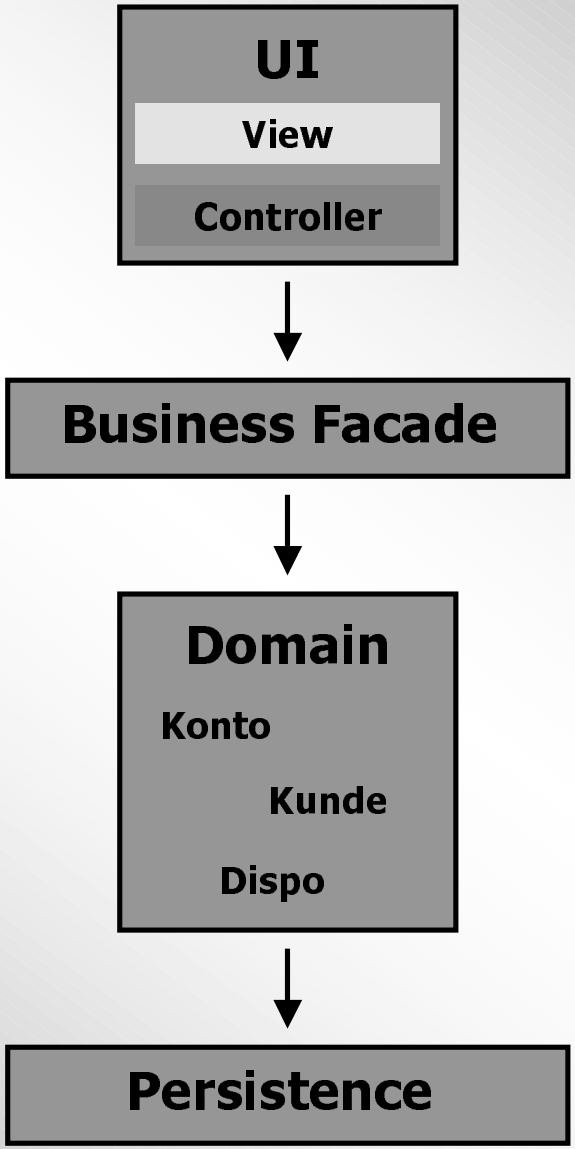
Angriffspunkte



FIT: Geschäftsregeln

Dispokredit		
Kundenklasse	Monatlicher Umsatz	Dispo()
Privatkunde	2500	7500
Privatkunde	1200	5000
Privatkunde	99	0
Geschäftskunde	25000	25000
Geschäftskunde	50000	25000 *expected* 50000 *actual*

Angriffspunkte



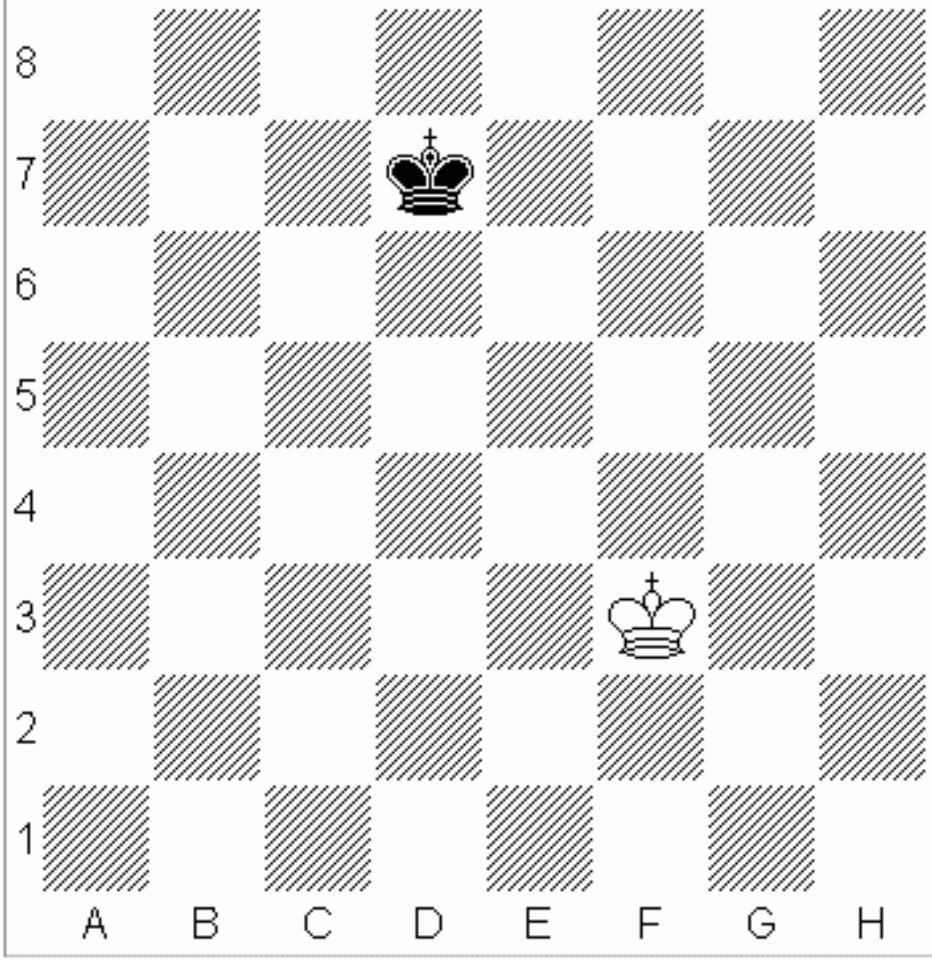
Set Up Kunden			
Kundennummer	Vorname	Nachname	Klasse
100001		andrena objects ag	Geschäftskunde
100002		Securitas Versicherung	Geschäftskunde
200001	Frank	Mischnick	Privatkunde
200001	Jens	Oberleitner	Privatkunde

FIT: Hintertür

Set Up Kunden			
Kundennummer	Vorname	Nachname	Klasse
100001		andrena objects ag	Geschäftskunde
100002		Securitas Versicherung	Geschäftskunde
200001	Frank	Mischnick	Privatkunde
200001	Jens	Oberleitner	Privatkunde

Die Sprache des Kunden

Chess	
command	setboard 8/3k4/8/8/8/5K2/8/8 - - - - -
check moves	f3-f4,f3-e4,f3-e3,f3-e2,f3-f2,f3-g2,f3-g3,f3-g4



FitNesse: Wiki-Server mit FIT-Bridge

Test Results: FitNesse.TwoMinuteExample - Mozilla Firefox

Datei Bearbeiten Ansicht Gehe Lesezeichen Extras Hilfe

Test
Edit
Versions
Properties
Refactor
Where Used
RecentChanges
Files
Search

FitNesse.

TwoMinuteExample

TEST RESULTS

Tests Executed OK

Assertions: 2 right, 1 wrong, 0 ignored, 0 exceptions

▼ Set Up: .FitNesse.SetUp Expand All | Collapse All

A One-Minute Description

AN EXAMPLE FITNESSE TEST

If you were testing the division function of a calculator application, you might like to see some examples working. You might want to see what you get back if you ask it to divide 10 by 2. (You might be hoping for a 5!)

In FitNesse, tests are expressed as tables of **input** data and **expected output** data. Here is one way to specify a few division tests in FitNesse:

eg.Division		
numerator	denominator	quotient?
10	2	5
12.6	3	4.2
100	4	24 *expected* / 25.0 *actual*

In this style of FitNesse test table (ColumnFixture), each row represents a complete scenario of example inputs and outputs. Here, the "numerator" and "denominator" columns are for inputs, and the question mark in the "quotient?" column tells FitNesse that this is our

Warum FIT?

- Tests sind von Entwicklung entkoppelt.
- Testsprache ist frei definierbar.
- Macht explizit, dass die Automatisierung eine Entwicklungstätigkeit ist.
- FIT und FitNesse sind kostenlos

Testgetriebene Entwicklung in der Praxis

- Probleme
- Zahlen
- Erfahrungen

Probleme (1) - Technik

- Was soll ich testen?
- Wie viel soll ich testen?
- Wo fange ich mit dem Testen an?
- Wie teste ich
  - persistente Objekte?
  - grafische Oberflächen?
  - Webapplikationen?
  - Enterprise JavaBeans?
  - ...?

Probleme (2) - Akzeptanz

- Test-First widerspricht der Erfahrung vieler Entwickler.
- Test-First kann nur in der Praxis erfahren werden.
- Coaching ist sehr hilfreich.

Zahlen

	Internet-Brokerage	Intranet-Anwendung	Inhouse-Portal-Framework	Anbindung Kasse an Filialsoftware
Projektgröße (Klassen)	172	623	115	ca. 600
Testklassen / Testfälle	114 / 582	108 / 534	80 / 305	ca. 250 / 830
Testcode : Anwendungscode	ca. 1 : 2	ca. 1 : 3	ca. 1,5 : 1	ca. 1 : 1
Laufzeit aller Tests	ca. 30 min	ca. 10 min	ca. 12 sec	ca. 3 min
Testabdeckung	> 75 %	> 70 %	> 95 %	> 90 %
Bugs in Produktion	10	25	-	1

Fazit

- Testgetriebene Entwicklung erfordert Gewöhnungszeit
- Subjektives Gefühl der Verlangsamung, aber objektiv kurze Projektlaufzeiten
- Auch große Änderungen möglich
- Fast alles ist testbar!
- Fehlende Testbarkeit häufig ein Designproblem
- Akzeptanztests entdecken andere Fehler als Unit Tests

Referenzen (1)

Kent Beck: Test-Driven Development By Example. Addison-Wesley, 2003.

Martin Fowler: Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1999.

Rick Mugridge, Ward Cunningham: Fit for Developing Software. Prentice Hall, 2005.

Joshua Kerievsky: Refactoring to Patterns. Addison-Wesley, 2004.

Referenzen (2)

Johannes Link: Softwaretests mit JUnit, dpunkt.verlag 2005

Frank Westphal: Testgetriebene Entwicklung mit JUnit und FIT, dpunkt.verlag 2006

XP-Days Germany

http://xpdays.de

Hamburg, 23. + 24. November 2006

Konferenz für Extreme Programming
und Agile Softwareentwicklung