13. Programme werden durch Testen erst zu Software

Prof. Dr. rer. nat. Uwe Aßmann
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 13-1.0, 29.04.13

1) Grundlagen
2) Vertragsüberprüfung
3) Testfalltabellen
4) Regressionstests mit dem JUnit-Rahmenwerk
5) Entwurfsmuster in JUnit

Literatur

- Obligatorische Literatur
  - Zuser Kap. 12 (ohne White-box tests)
  - ST für Einsteiger Kap. 12
  - Essential Java tutorials on Exceptions and Pattern Matching http://docs.oracle.com/javase/tutorial/essential/index.html
  - www.junit.org
  - Junit 3.x arbeitet noch ohne Metadaten (@Annotationen)
    - junit3.8.1/doc/cookstour/cookstour.htm. Schöne Einführung in Junit
    - junit3.8.1/doc/faq/faq.htm Die FAQ (frequently asked questions)
  - Achtung: JUnit 4 versteckt mehr Funktionalität in Metadaten
    - http://junit.sourceforge.net/doc/cookbook/cookbook.htm
    - http://junit.sourceforge.net/doc/faq/faq.htm
- Weiterführend
  - Andrew Hunt, David Thomas. The pragmatic programmer. Addison-Wesley. Deutsch: Der Pragmatische Programmierer. Hanser-Verlag.
  - Uwe Vigenschow. Objektorientiertes Testen und Testautomatisierung in der Praxis. Konzepte, Techniken und Verfahren. dpunkt-Verlag, 2005.

Even Worms are Tested: StuxNet Tests in Israel

- http://catless.ncl.ac.uk/Risks/26.31.html#subj3.1
- Over the past two years, according to intelligence and military experts familiar with its operations, Dimona/Negev has taken on a new, equally secret role - as a critical testing ground in a joint American and Israeli effort to undermine Iran's efforts to make a bomb of its own.
- Behind Dimona's barbed wire, the experts say, Israel has spun nuclear centrifuges virtually identical to Iran's at Natanz, where Iranian scientists are struggling to enrich uranium. They say Dimona tested the effectiveness of the Stuxnet computer worm, a destructive program that appears to have wiped out roughly a fifth of Iran's nuclear centrifuges and helped delay, though not destroy, Tehran's ability to make its first nuclear arms.
- "To check out the worm, you have to know the machines," said an American expert on nuclear intelligence. "The reason the worm has been effective is that the Israelis tried it out."

Deploy First, Test Later

- "Steven M. Bellovin" <smb@cs.columbia.edu> Tue, 25 Sep 2007 21:16:45 +0000
- RISKS readers are familiar with the difficulty of deploying new software systems. Even with the best will in the world, some things with just break.
- In an effort to forestall this, Arizona State University decided to act like a 90s-style .com: deploy first, even if the software is buggy, try to cope with the problems, and fix the code later. As I read the Wall Street Journal story (for subscribers, http://online.wsj.com/article/SB119067729479838055.html), it didn't work very well. 3,000 employees were unpaid or underpaid, and the backup procedures couldn't scale by nearly enough.
- Some of the trouble was that many employees in, say, janitorial positions didn't have their own computers, and not enough departmental machines were available. More of the trouble was the usual: the new system didn't behave the same way as the old one did, especially when handling minor errors.
- They had a backup plan: the HR department would write checks, no questions asked, for any employee who received an inaccurate paycheck. But there were too many errors, and HR couldn't keep up.

- Mr. Reinke says instead of writing him a check to replace his blank paycheck, he was told that a change would be made in the system. He received his check a week later. In the meantime, he had to extend his overdraft protection in order to pay his $800-a-month mortgage. Hundreds of other employees had to wait as many as 12 days to have their paychecks fixed. A spokesman for the Arizona State Credit Union says that 55 people took out short-term loans.
- The new strategy's pain is undeniable. "Morale is the lowest it's been in the 14 years I've worked here," says Allan Crouch, who works in the university's human-resources department.
- The university seems to be blaming HR, not IT. Two HR employees have been placed on leave. And the IT folks? They think the conversion was a success: While unpaid employees may have been less than thrilled, school administrators, and consultants and software companies involved in the project rave about Arizona State's strategy. Oracle hailed it as a model for both universities and corporations to follow in a report it published in April 2007. In a statement, Jim McGlothlin, an Oracle vice president called the project "highly successful." Gary Somers, who worked on the project for CedarCrestone, Inc., the consulting company that helped implement the system, calls Arizona State's method "the wave of the future."
- Ship first, debug later, use employees who haven't volunteered for financial hardship as your test subjects. Imagine the reaction of the school's Institutional Review Board if a professor has proposed a human subjects study with similar characteristics.
- Steve Bellovin, http://www.cs.columbia.edu/~smb

13.1. Grundlagen des Testens und testgetriebender Entwicklung

Softwaretechnologie, © Prof. Uwe Aßmann
Technische Universität Dresden, Fakultät Informatik

Grundlagen

Wikipedia:
Verification and Validation: In engineering or a quality management system, "verification" is the act of reviewing, inspecting, testing, etc. to establish and document that a product, service, or system meets the regulatory, standard, or specification requirements.
By contrast, validation refers to meeting the needs of the intended end-user or customer.

- **Validation**: Überprüfung der Arbeitsprodukte bzgl. der Erfüllung der Spezifikationen und der Wünsche des Kunden
- **Verifikation**: Nachweis, daß ein Programm seine Spezifikation erfüllt
- **Formale Verifikation**: Mathematischer Beweis desselben
- **Testen**: Stichprobenhafte Verifikation: Überprüfen von ausgewählten Abläufen eines Programms unter bekannten Bedingungen, mit dem Ziel, Fehler zu finden
- Wichtig: Da Vollständigkeit nicht erreicht werden kann, wie hoch ist die Test-Abdeckung?

Testing shows the presence of bugs, but never their absence (Dijkstra)

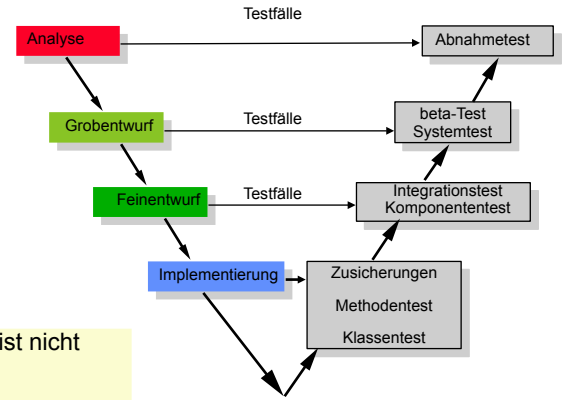
Verifikation zeigt, dass das Programm seine Spezifikation richtig erfüllt.
Validation zeigt, dass das Programm das richtige Problem löst.

Test-First Development

- **Gesetz 62** des Pragmatischen Programmierers: *Testen Sie frühzeitig, häufig und automatisch*
- **Gesetz 49 (PP):** Testen Sie Ihre Software, sonst tun es die Anwender!
- **Gesetz 32 (PP):** Ein totes Programm richtet weniger Schaden an als ein schrottreifes.

V-Modell mit Standard-Testprozess (Rechter Ast des V)

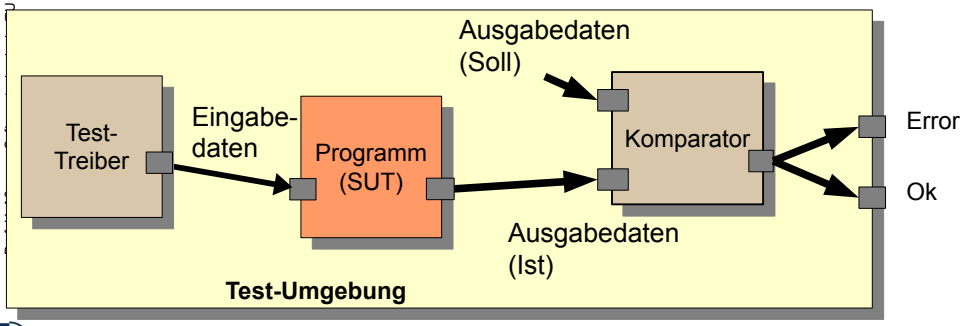
- Tests werden *bottom-up* erledigt:
  - Zuerst Verträge und Testfälle für die Klasse bilden
  - Dann die einzelne Klasse testen
  - Dann die Komponente
  - Dann das System
  - Dann der beta-Test
  - Zum Schluss der Akzeptanztest (Abnahmetest)
- Bitte, auch so im Praktikum vorgehen.



- Gesetz 63 (PP): Das Programmieren ist nicht getan, bis alle Tests erfolgreich waren

Software hat eine test-getriebene Architektur

- Solange ein Programm keine Tests hat – ist es keine Software
- **Software** hat, i.G. zu Programmen**,** eine testgetriebene Architektur, die das Testen von vorne herein unterstützt
  - Testen ist eine **Ist-Soll-Analyse**
- Ein **test-getriebenes System (test-driven system)** besteht aus einem Programm (*System under Test, SUT*) mit seiner *Testumgebung (Test-Treiber, test runner)*
  - Entwicklungsmethode: *TDD (Test-Driven Development)*



Beispiel: Wie schreibt man einen Test für eine Methode?

- Wie testet man *parseDay(String d)?*

```
// A class for standard representation of dates.
public class Date {
  public int day; public int month; public int year;
  public Date(String date) {
    day = parseDay(date);
    month = parseMonth(date);
    year = parseYear(date);
  }
  public int parseDay(String d) {
    if (d.matches(„\\d\\d.\\d\\d\\.\\d\\d\\d\\d“)) {
      // German numeric format day.month.year
      return Integer.parseInt(d.substring(0,2));
    } else {
      .. other formats...
    }
  }
}
```


Antwort: Innere Checks und äussere Tests

- **Innere Checks (Vertragsprüfungen, contract checks):** innerhalb der Methode werden Überprüfungen eingebaut, die bestimmte Zusicherungen erreichen
  - Überprüfen von Verträgen *innerhalb von Methoden* mit Hilfe von Probebeschicken von Methoden mit ausgewählten Parametern *(Testdaten)*
  - Auslösen von *Ausnahmen (exceptions)*
- **Äussere Tests:** Nach Aufruf einer Methode mit Testdaten werden Relationen zwischen Ein-, und Ausgabeparametern sowie dem Zustand überprüft
  - **Black-box-Tests** (schwarze Tests): bestehen aus
    - dem Aufstellen von Testfällen (Testdaten für Eingabe-, Ausgabeparametern und Zustandsbelegungen)
    - und deren Überprüfung nach Abarbeitung der Methode
    - ohne Kenntnis der Implementierung der Methode
  - **White-box-Tests** (weiße Tests): dto,
    - aber mit Kenntnis über die Implementierung der Methode (z.B. Kenntnis der Steuerfluss-Pfade)

13.2. Vertragsprüfung

Innere Checks: Vertragsprüfung für eine Methode

- **Invarianten (invariants):** Bedingungen über den Zustand (Werte von Objekten und Variablen), die *immer* gültig sind
- **Vorbedingung (precondition, Annahmen, assumptions):**
  - Zustand (Werte von Variablen), der *vor* Aufruf (bzw. Vor Ausführung der ersten Instruktion) gilt bzw gelten muss
  - Typen von Parametern
  - Werteeinschränkungen von Parametern
- **Nachbedingung (postcondition, Garantien, guarantees, promises):**
  - Zustand (Werte von Variablen), die *nach* Aufruf (bzw. nach Ausführung der letzten Instruktion) gültig sind
  - Zuordnung von Werten von Eingabe- zu Ausgabeparametern (Ein-Ausgaberelation)

Gesetz 31 (PP): Verwenden Sie Design by Contract (Vertragsprüfung), damit der Quelltext nicht mehr und nicht weniger tut, als er vorgibt.

Gesetz 33 (PP): Verhindern Sie das Unmögliche mit Zusicherungen.

13.2.1 Vertragsprüfung für eine Methode mit Rückgabe eines Fehlercodes

DateSimple.java

- Preconditions werden im Prolog, Postconditions im Epilog einer Methode geprüft; Invarianten überall
- Abbruch bei Fehlschlag der Prüfung

Vorbedingung (precondition):
d ist ein String
d ist nicht leer

Invarianten (invariants):
d ist mindestens 10 Zeichen lang (Datum plus Trenner)

Nachbedingung (postcondition):
Ein int wird zurückgegeben
Zwischen 1 und 31

```
public int parseDay(String d) {
  if (d.equals(„")) { System.err.println(„empty"); return 0;}
  if (d.size() <10) { System.err.println(„size too small"); return 0; }
  if (d.matches(„\\d\\d.\\d\\d.\\d\\d\\d\\d")) {
    if (d.size() < 10) System.err.println(„size too small"); return 0; }
    // German numeric format day. month. Year
    int day = Integer.parseInt(d.substring(0,2));
    if (day < 1 || day > 31) throw new DayTooLarge();
  } else {
    .. other formats...
  }
  if (d.size() < 10) { System.err.println(„size too small");  return 0; }
  if (day < 1 || day > 31) { System.err.println(„illegal day");  return 0; }
  return day;
}
```

13.2.2 Vertrag einer Methode – Prüfen durch assert

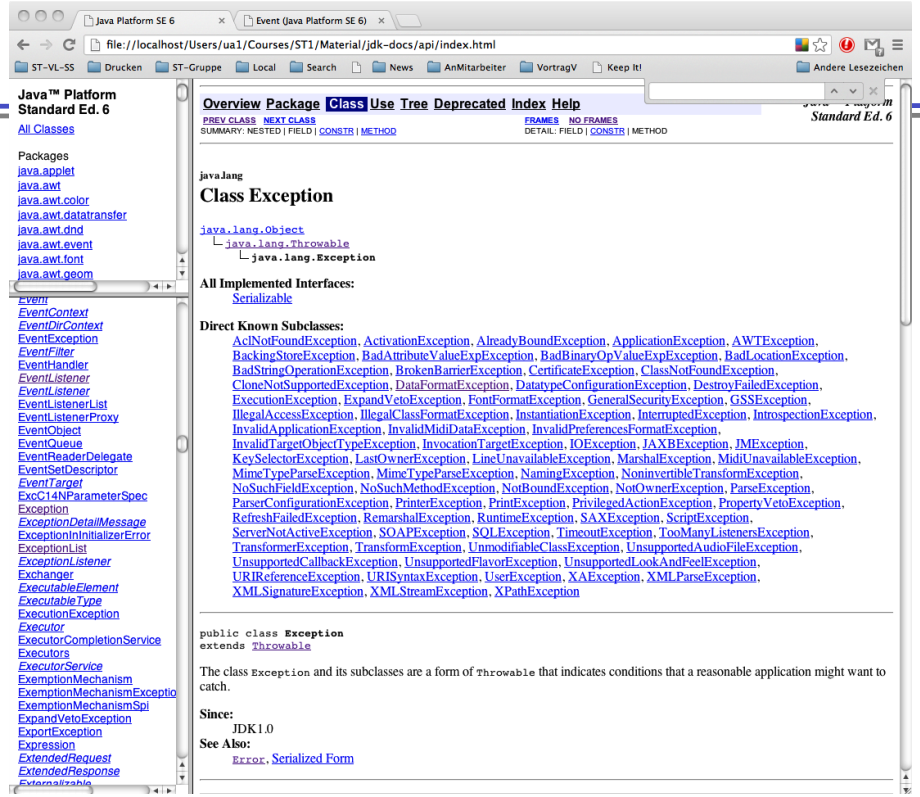
- `assert()`, eine Standardmethode, bricht das Programm bei Verletzung einer Vertragsbedingung ab
- Achtung: Bedingungen müssen dual zu den Bedingungen der vorgenannten Ausnahmen formuliert sein!

```
public int parseDay(String d) {
  assert(!d.equals(„"));
  assert(d.size() >= 10));
  if (d.matches(„\\d\\d.\\d\\d.\\d\\d\\d\\d")) {
    assert(d.size() >= 10);
    // German numeric format day. month. Year
    int day = Integer.parseInt(d.substring(0,2));
    assert(day >= 1 and day <=  31);
  } else {
    .. other formats...
  }
  assert(d.size() >= 10);
  assert(day >= 1 and day <=  31);
  return day;
}
```

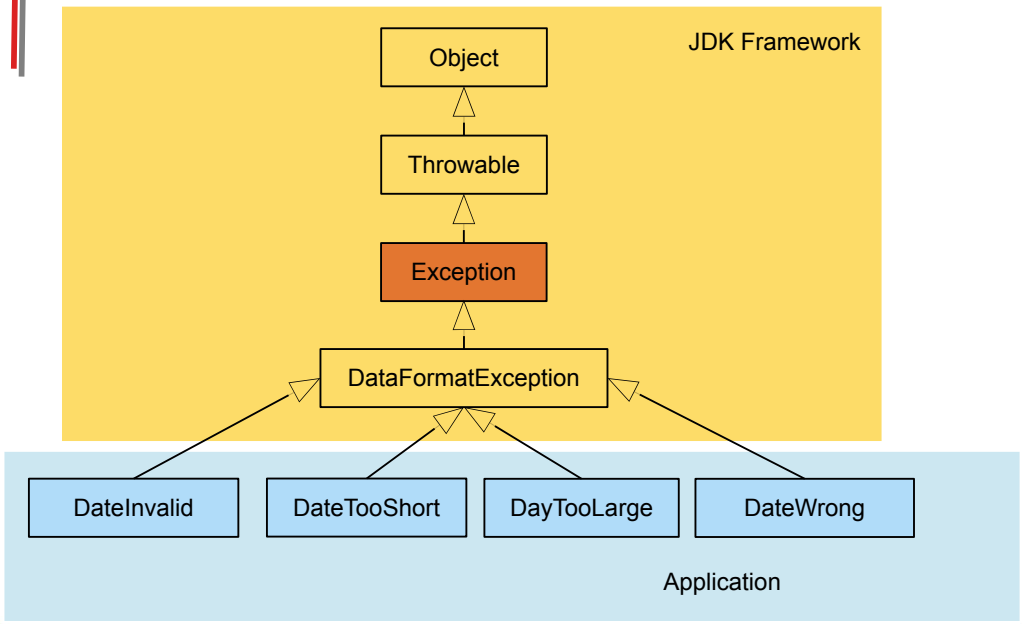
13.2.3 Vertragsprüfung mit Exceptions (Ausnahmen)

- **Ausnahme *(Exception):***
  - Objekt einer Unterklasse von java.lang.Exception
  - Vordefiniert oder und selbstdefiniert
- Ausnahme
  - **auslösen** *(to **throw** an exception)*
    - Erzeugen eines Exception-Objekts
    - Löst Suche nach Behandlung aus
  - **abfangen** und **behandeln** *(to **catch** and **handle** an exception)*
    - Aktionen zur weiteren Fortsetzung des Programms bestimmen
  - **deklarieren**
    - Angabe, daß eine Methode außer dem normalen Ergebnis auch eine Ausnahme auslösen kann (Java: **throws**)
    - Beispiel aus java.io.InputStream:

```
public int read() throws IOException;
```

Anwendungsspezifische Ausnahmen



Vertragsprüfung für eine Methode mit Exceptions

- Eine fehlgeschlagene Vertragsprüfung kann eine Ausnahme (exception) auslösen, mittels `throw`-Anweisung
  - Dazu muss ein Exception-Objekt angelegt werden
- Vorteil: Ursache des Fehlers kann in einem großen System weit transportiert werden, gespeichert werden, oder in eine Testumgebung zurückgegeben werden

DateWithExceptions.java

```
public int parseDay(String d) {
  if (d.equals(„"))    throw new DateInvalid();
  if (d.size() < 10)   throw new DateTooShort();
  if (d.matches(„\\d\\d.\\d\\d.\\d\\d\\d\\d")) {
    if (d.size() < 10)  throw new DateTooShort();
    // German numeric format day.month.year
    int day = Integer.parseInt(d.substring(0,2));
    if (day < 1 || day > 31) throw new DayTooLarge();
  } else {
    .. other formats...
  }
  if (d.size() < 10)   throw new DateTooShort();
  if (day < 1 || day > 31) throw new DateWrong();
  return day;
}
```

Java-Syntax für Ausnahmebehandlung


```
class TotalDiv {
  public static int tDiv (int x, int y)
    throws ArithmeticException {
    try {
      return (x / y);
    }
    catch (ArithmeticException e) {
      System.err.println(„Division by zero“);
      throw new ArithmeticException();
    }
  }
}
```

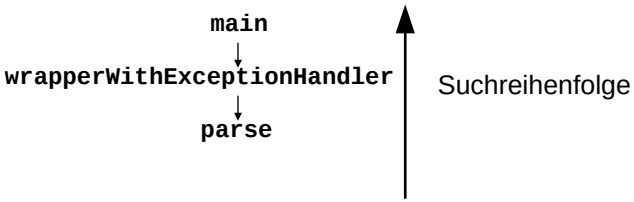
Schnittstellen-Deklaration

Gekapselter Block

Fänger

Dynamische Suche nach Ausnahmebehandlung

- Suche nach Abfangklausel (catch) entlang der (dynamischen) Aufrufhierarchie:



- Bei mehreren Abfangklauseln an der gleichen Stelle der Hierarchie gilt die zuerst def nierte Klausel:

```
try { }
catch (DateInvalid e)
catch (DayTooLarge e)
catch (DateWrong e)
```

Suchreihenfolge

Regeln zum Umgang mit Ausnahmen

- Gesetz des pragmatischen Programmierers 58: **Bauen Sie die Dokumentation ein**
  - Ausnahmebehandlung niemals zur Behandlung normaler (d.h. häufig auftretender) Programmsituationen einsetzen
  - Ausnahmen sind Ausnahmen, regulärer Code behandelt die regulären Fälle!
- Gesetz 34: **Verwenden Sie Ausnahmen nur ausnahmsweise**
  - Nur die richtige Dosierung des Einsatzes von Ausnahmen ist gut lesbar
- Gesetz 35: **Führen Sie zu Ende, was Sie begonnen haben**
  - Auf keinen Fall Ausnahmen “abwürgen”, z.B. durch triviale Ausnahmebehandlung
  - Ausnahmen zu propagieren ist keine Schande, sondern erhöht die Flexibilität des entwickelten Codes.
- Gesetz 33 **Verhindern Sie das Unmögliche mit Zusicherungen**
  - Vertragsüberprüfungen, generieren Ausnahmen

13.3. Testfallspezifikation mit Testfalltabellen

13.1.4 Aufschreiben von Testfällen in Testfalltabellen

- Ein **Testfalltabelle** setzt für eine aufzurufende Methode Ein-, Ausgabeparameter, lokale Variablen und Objektattribute in Beziehung.
  - **Gut-Fall (Positivtest):** Testfall, der bestanden werden muss
  - **Fehlerfall (Negativtest):** Testfall, der scheitern muss
  - **Ausnahmefall (Exception Test):** Testfall, der in einer Exception (Ausnahme) des Programms enden muss, also einer kontrollierten Fehlersituation
- Die Testfalltabelle spezifiziert also einen Vertrag exemplarisch
- Aus jeder Zeile der Testfalltabelle wird ein assert()-Test erzeugt, der nach dem Aufruf der Prozedur ausgeführt wird
  - Testet die Relation der Ein- und Ausgabeparameter, sowie der Objektattribute

Beispiel: Test einer Datumsklasse


```
// A class for standard representation of dates.
public class Date {
  public int day; public int month; public int year;
  public Date(String date) {
    day = parseDay(date);
    month = parseMonth(date);
    year = parseYear(date);
  }
  public int equals(Date d) {
    return day == d.day &&
           year == d.year &&
           month == d.month;
  }
}
```

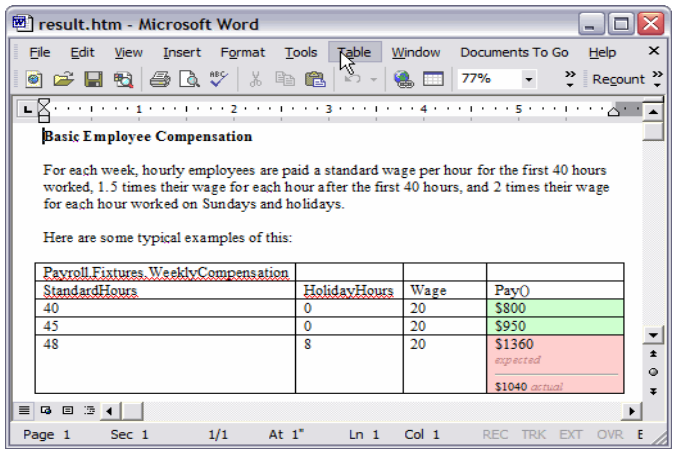

Beispiel einer zugehörigen Testfalltabelle

- Testfalltabellen enthalten Testfälle (**Gut-, Fehler-, Ausnahmefälle**)

Nr	Klasse	Eingabedaten	Ausgabedaten			Erwarteter Status
			day	month	year	
1	Gutfall	1. Januar 2006	1	1	2006	Ok
2	Gutfall	05/12/2008	5	12	2008	Ok
3	Gutfall	January 23, 2017	23	1	2017	Ok
4	Fehlerfall	44, 2007				Failure
5	Fehlerfall	Aup 23, 2005				Failure
6	Ausnahme	March 44, 2007				Exception

FIT Testfalltabellen Framework http://fit.c2.com

- FIT bietet eine Spezifikation der Testfälle in Word oder Excel
  - Automatische Generierung von Junit-Testfällen
  - Automatischer Feedback
- siehe Softwaretechnologie-II, WS



http://fit.c2.com/files/WelcomeVisitors/example.gif

Neuer Testfall

29

- Testfälle werden in eine Testfallklasse geschrieben
  - Die Testdaten befinden sich in einer *Halterung (fixture)*
  - Eine Testfallklasse kann mehrere Testfälle aus der Testfalltabelle enthalten

```
public class DateTestCase {
   Date d1;
   Date d2;
   Date d3;
                          Halterung (fixture)
  public int testDate() {
    d1 = new Date(„1. Januar 2006“);
    d2 = new Date(„05/12/2008“);
    d3 = new Date(„January 23rd, 2009“);

    assert(d1.day == 1); assert(d1.month == 1); assert(d1.year == 2006);
    assert(d2.day == 5); assert(d2.month == 12); assert(d2.year == 2008);
    assert(d3.day == 23); assert(d3.month == 1); assert(d3.year == 2009);
  }
}
```

Wie wähle ich Testdaten aus?

30

- Bestimme die **Extremwerte** der Parameter der zu testenden Methode
  - Nullwerte immer testen, z.B. 0 oder null
  - Randwerte, z.B. 1.1., 31.12
- Bestimme **Bereichseinschränkungen**
  - Werte ausserhalb eines Zahlenbereichs
  - negative Werte, wenn natürliche Zahlen im Spiel sind
- Bestimme **Zustände**, in denen sich ein Objekt nach einer Anweisung befinden muss
- Bestimme **Äquivalenzklassen** von Testdaten und teste nur die Repräsentanten
- Bestimme alle Werte aller **boolschen Bedingungen** in der Methode
  - Raum aller Steuerflußbedingungen

13.4. Regressionstests mit dem JUnit-Rahmenwerk

31

Testfälle

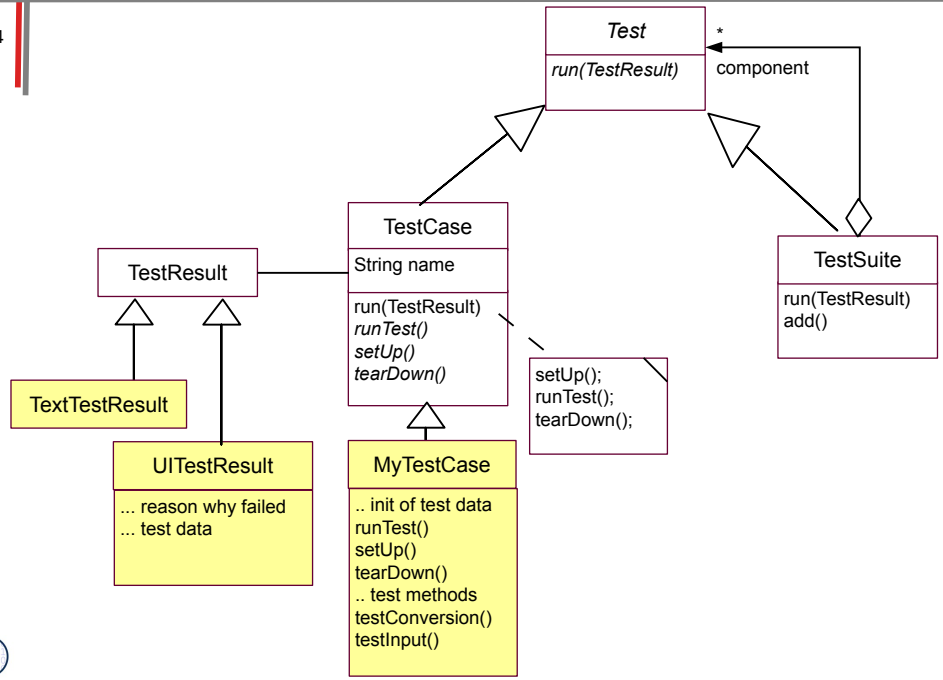
32

- **Testfall**: Herbeiführen einer bestimmten Programm-Situation, mit bestimmten Testdaten
- **Testdaten**: Eingabe, die einen konkreten Testfall herbeiführt
- **Testdatensatz**: Ein- und Ausgabedaten, die zu einem Testfall gehören
- **Regressionstest**: Automatisierter Vergleich von Ausgabedaten (gleicher Testfälle) unterschiedlicher Versionen des Programms.
  - Da zu großen Systemen mehrere 10000 Testdatensätze gehören, ist ein automatischer Vergleich unerläßlich.
  - Beispiel: Validierungssuiten von Übersetzern werden zusammen mit Regressionstest-Werkzeugen verkauft. Diese Werkzeuge wenden den Übersetzer systematisch auf alle Testdaten in der Validierungssuite an

Das JUnit Regressionstest-Framework

- **JUnit** www.junit.org ist ein technisches Java-Framework für Regressionstests, sowohl für einzelne Klassen (*unit test)*, als auch für Systeme
  - Durchführung von Testläufen mit Testsuiten automatisiert
  - Eclipse-Plugin erhältlich
  - Mittlerweile für viele Sprachen nachgebaut
- Junit 3.8.1:
  - 88 Klassen mit 7227 Zeilen
  - im Kern des Rahmenwerks: 10 Klassen (1101 Zeilen)
- Testresultate:
  - Failure (Zusicherung wird zur Laufzeit verletzt)
  - Error (Unvorhergesehenes Ereignis, z.B. Absturz)
  - Ok
- JUnit-4 versteckt mehr Funktionalität mit Metadaten (@Annotationen) und ist wesentlich komplexer. Empfehlung: Lernen Sie zuerst 3.8.1!

Kern von JUnit 3.8.1



Exkurs: Erkunde JUnit 3.8.x mit Javadoc

- Aufgabe:
  - laden Sie die API-Dokumentation von JUnit mit einem Brauser Ihrer Wahl
  - finden Sie die Aufgabe der Klassen TestResult, TestCase und TestSuite heraus
  - Welche Aufgabe hat die Klasse Assert?

/home/ua1/Courses/ST1/Material/junit3.8.1/javadoc/index.html

Gesetz 68 (PP): Bauen Sie die Dokumentation ein, anstatt sie dranzuschrauben

Beispiel: Test der Datumsklasse in JUnit


```
// A class for standard representation of dates.
public class Date {
  public int day; public int month; public int year;
  public Date(String date) {
    day = parseDay(date);
    month = parseMonth(date);
    year = parseYear(date);
  }
  public int equals(Date d) {
    return day == d.day &&
           year == d.year &&
           month == d.month;
  }
}
```


Neuer Testfall in Junit 3.8.x

- TestCases sind Methoden, beginnend mit der Markierung `test`
- Initialisierungen der Halterung mit `setUp`, Abbau mit `tearDown`

```
public class DateTestCase extends TestCase {
   Date d1;
   Date d2;
   Date d3;                          Halterung (fixture)
  protected void setUp() {
    d1 = new Date(„1. Januar 2006“);
    d2 = new Date(„01/01/2006“);
    d3 = new Date(„January 1st, 2006“);
  }
  public void testDate1() {
    assert(d1.equals(d2));
    assert(d2.equals(d3));
    assert(d3.equals(d1));
    .... more to say here ....
  }
  public void testDate2() { .. more to say here .... }
  protected void tearDown() { .. .. }
}
```


Testfall als Kunde einer zu testenden Klasse

- Testfallklassen sind also “*Kundenklassen*” von zu testenden Klassen,
  - die mittels äusserem Test deren Vorbedingungen, Invarianten und Nachbedingungen überprüfen
  - Angewandt auf verschiedene Testdaten
- Testfallklassen imitieren “Kunden”
  - unterliegen also dem Lebenszyklus eines objektorientierten Programms
  - Aufbauphase (Testobjektallokation, Vernetzung)
  - Arbeitsphase
  - Rekonfigurationsphase
  - Abbauphase

Benutzung von TestCase

- Von Eclipse aus: In einer IDE wie Eclipse werden die Testfall-Prozeduren automatisch inspiziert und gestartet
- Von einem Java-Programm aus:
  - Ein Testfall wird nun erzeugt durch einen Konstruktor der Testfallklasse
  - Der Konstruktor sucht die Methode des gegebenen Namens (“testDate1”) und bereitet sie zum Start vor
    - mit *Reflektion,* d.h. Suche nach dem Methode in dem Klassenprototyp
  - Die *run()* Methode startet den Testfall gegen die Halterung und gibt ein TestResult zurück

```
public class TestApplication {
   ...
   TestCase tc = new DateTestCase(„testDate1“);
   TestResult tr = tc.run();
}
```


Testsuiten

- Eine Testsuite ist eine Kollektion von Testfällen
- TestSuites sind komposit

```
public class TestApplication {
   ...
   TestCase tc = new DateTestCase(„testDate1“);
   TestCase tc2 = new DateTestCase(„testDate2“);
   TestSuite suite = new TestSuite();
   suite.addTest(tc);
   suite.addTest(tc2);
   TestResult tr = suite.run();
   // Nested test suites
   TestSuite subsuite = new TestSuite();
   ... fill subsuite ...
   suite.addTest(subsuite);
   TestResult tr = suite.run();
}
```


TestRunner GUI

- Die Klassen junit.awtui.TestRunner, junit.swingui.TestRunner bilden einfach GUIs, die Testresultate anzeigen
- Gibt man einem Konstruktor eines Testfalls eine Klasse mit, findet er die “test*”-Methoden (die Testfallmethoden) selbständig
- Dies geschieht mittels *Reflektion,* d.h. Absuchen der Methodentabellen im Klassenprototypen und Methodenspeicher

```
public class TestApplication {
  public static Test doSuite() {
    // Abbreviation to create all TestCase objects
    // in a suite
    TestSuite suite = new TestSuite(DateTestCase.getClass());
  }
  // Starte the GUI with the doSuite suite
  public static main () {
    junit.awtui.TestRunner.run(doSuite());
  }
}
```


13.4.2) Testläufe in Junit 4.X

Neuer Testfall in Junit 4.X mit Metadaten-Annotationen

- TestCases sind Methoden, annotiert mit `@test`, Initialisierung und Abräumen mit `@before, @after`
- Metadaten-Annotationen in Java entsprechen Stereotypen bzw Tagged Values in UML. Sie sind durch **Annotationstypen** typisiert

```
public class DateTestCase /* no superclass */ {
    Date d1;
    Date d2;
    Date d3;                          Halterung (fixture)
  @before protected int setUp() {
    d1 = new Date(„1. Januar 2006“);
    d2 = new Date(„01/01/2006“);
    d3 = new Date(„January 1st, 2006“);
  }
  @test public int compareDate1() {
    assert(d1.equals(d2));
    assert(d2.equals(d3));
    assert(d3.equals(d1));
    .... more to say here ....
  }
  @test public int compareDate2() {
    .... more to say here ....
  }
}
```


Benutzung von Testfall-Klasse in 4.x

- Von der Kommandozeile:
  - java org.junit.runner.JUnitCore DateTestCase
- Von Eclipse aus: In einer IDE wie Eclipse werden die Testfall-Prozeduren automatisch inspiziert und gestartet
- Von einem Java-Programm aus:
  - Ein Testfall wird erzeugt durch einen Konstruktor der Testfallklasse
  - Suche den Klassenprototyp der Testfallklasse
  - Die *run()* Methode von JUnitCore startet alle enthaltenen Testfälle über den Klassenprotoypen
    - Starten aller annotierten Initialisierungen, Testfallmethoden, Abräumer
  - und gibt ein “Result”-Objekt zurück

```
public class TestApplication {
   ...
   DateTestCase tc = new DateTestCase();
   Result tr = JUnitCore.run(tc.getClass());
}
```


Junit 4.X mit vielen weiteren Metadaten-Annotationen

- Viele weitere Test-Annotationstypen sind definiert

```
public class DateTestCase {
    Date d1;
  @beforeClass protected int setUpAll() {
    // done before ALL tests in a class
  }
  @afterClass protected int tearDownAll() {
    // done before ALL tests in a class
  }
  @test(timeout=100,expected=IndexOutOfBoundException.class)
  public int compareDate2() {
    // test fails if takes longer than 50 msec
    // test fails if IndexOutOfBoundException is NOT thrown
    .... more to say here ....
  }
}
```

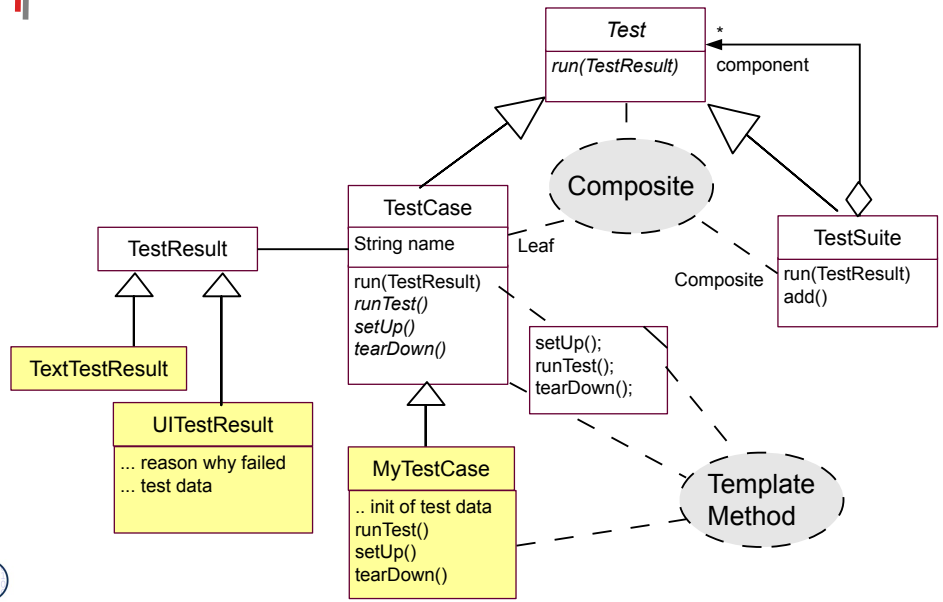

13.5. Entwurfsmuster in JUnit

Was ist ein Entwurfsmuster?

- Ein **Entwurfsmuster** ist...
  - Eine Beschreibung einer Standardlösung für
  - Ein Standardentwurfsproblem
  - in einem gewissen Kontext
- Ein Entwurfsmuster wiederverwendet bewährte Entwurfsinformation
  - Ein Entwurfsmuster darf nicht *neu*, sondern muss wohlbewährt sein
- Ein Entwurfsmuster enthält mindestens:
  - Klassendiagramm der beteiligten Klassen
  - Objektdiagramm der beteiligten Objekte
  - Interaktionsdiagramm (Sequenzdiagramm, Kommunikationsdiagramm)
- Entwurfsmuster sind ein wesentiches Entwurfshilfsmittel aller Ingenieure
  - Maschinenbau
  - Elektrotechnik
  - Architektur
- Entwurfsmuster treten auch in Frameworks wie JUnit auf

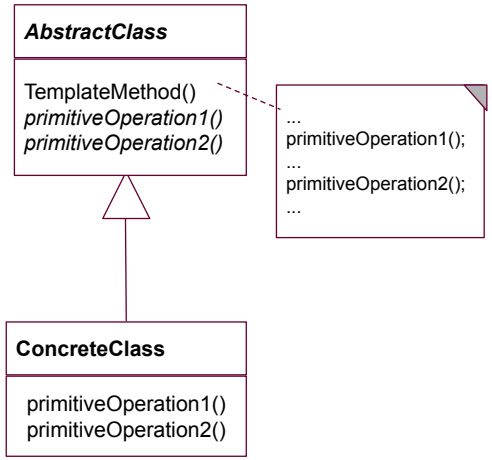
Beispiel: Entwurfsmuster in Junit 3.x

- Entwurfsmuster sind spezifische Szenarien von Klassen



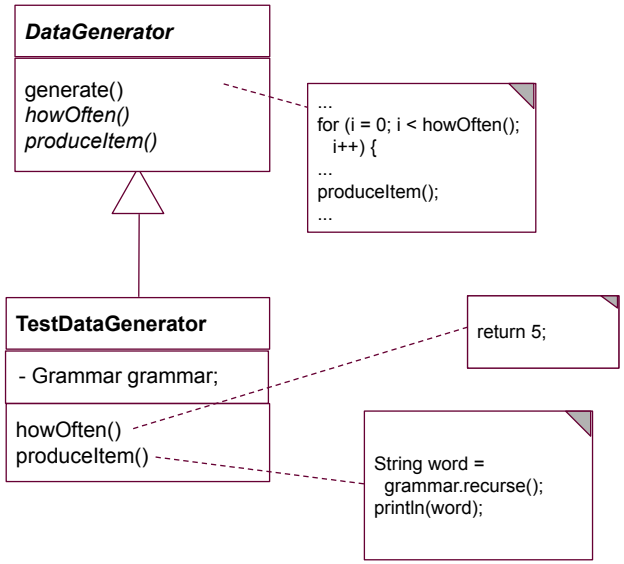
Entwurfsmuster TemplateMethod

- Definiert das Skelett eines Algorithmusses in einer *Schablonenmethode (template method)*
  - Die Schablonenmethode ist konkret
- Delegiere Teile zu abstrakten *Hakenmethoden (hook methods)*
  - die von Unterklassen konkretisiert werden müssen
- Variiere Verhalten der abstrakten Klasse durch verschiedene Unterklassen
  - Separation des “fixen” vom “variablen” Teil eines Algorithmus



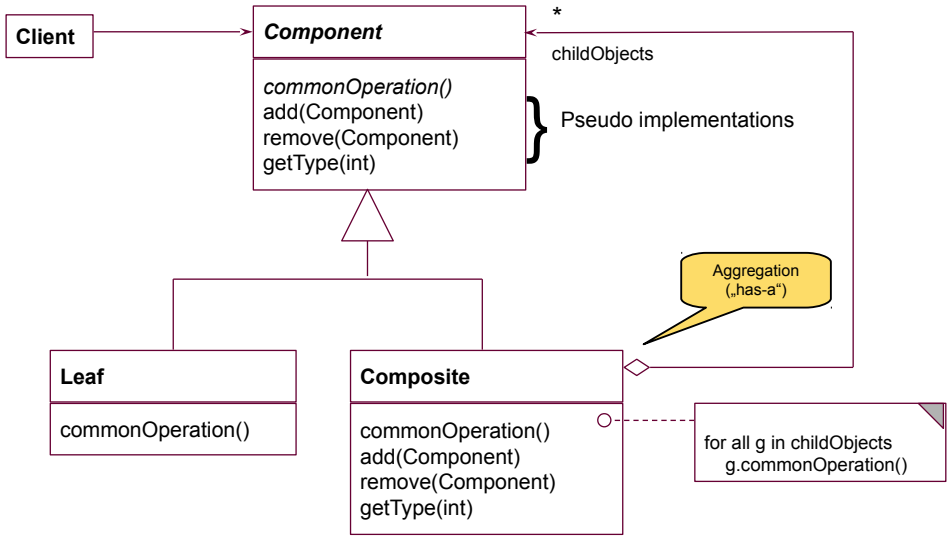
Beispiel TemplateMethod: Ein Datengenerator

- Parameterisierung eines Generators mit Anzahl und Produktion
  - (Vergleiche mit `TestCase` aus JUnit)



Entwurfsmuster Composite

- Composite besitzt eine rekursive n-Aggregation zur Oberklasse

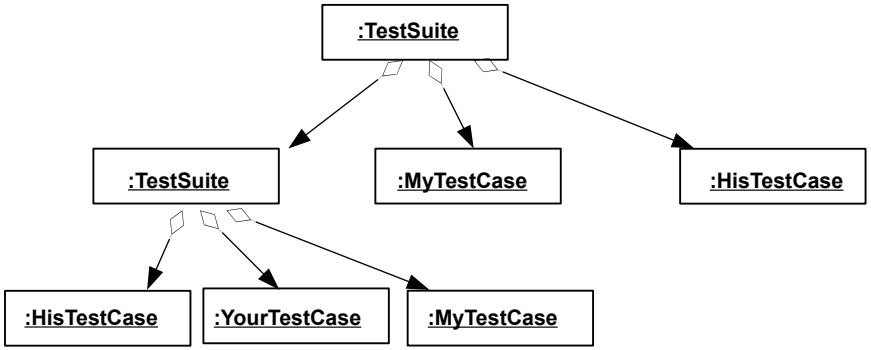


Composite in Junit 3.x

- Mehrere Methoden von *Test* sind komposit strukturiert
  - run()
  - countTestCases()
  - tests()
  - toString()

Composite

- Beschreibt Teil/Ganzes-Hierarchien von Laufzeit-Objekten, z.B. geschachtelte Testsuiten und -fälle



Bsp.: Zählen von Testfällen in JUnit


```
abstract class Test {
   abstract int countTestCases();
}
class TestSuite extends Test {;
   Test [20] children;// here is the n-recursion
   int countTestCases() { // common operation
     for (i = 0; i <= children.length; i++) {
       curNr += children[i].countTestCases();
     }
     return curNr;
   }
   void add(Test c) {
       children[children.length++] = c;
   }
```

```
class TestCase extends Test {
    private int myTestCaseCount = 10;
    int countTestCases() { // common operation
        return  myTestCaseCount;
    }
    void add(Test c) {
        /// impossible, dont do anything
    }
}

// application
main () { int nr = test.countTestCases(); }
```

Funktionales Programmieren:
•Iteratoralgorithmen (map)
•Faltungsalgorithmen (folding)

Praktikum Wintersemester

- Erstellung eines Akzeptanztestbeschreibung im Vertrag (Pflichtenheft)
  - Ohne Erfüllung kein Bestehen des Praktikums!
  - Eine Iteration: Kunde stellt einen Zusatzwunsch: Wie reagiert man auf die Veränderung?
- **Tip**: Erstellen Sie sich von Anfang an einen Regressionstest!
  - Und lassen sie diesen bei jeder Veränderung laufen, um zu überprüfen, ob Sie wesentliche Eigenschaften des Systems verändert haben

Was haben wir gelernt?

- Software ohne Tests ist keine Software
- Achten Sie auf das Management Ihres Projekts im Praktikum
  - Planen Sie hinreichend
- Testen Sie sorgfältig und von Anfang an (*test-driven development, TDD)*
  - Erstellen Sie eine Akzeptanztestsuite
  - Erstellen Sie einen Regressionstest
- Erste Entwurfsmuster TemplateMethod, Composite
- Lernen Sie, Java zu programmieren:
  - Ohne ausreichende Java-Kenntnisse weder Bestehen der Klausur noch des Praktikums
  - Nutzen Sie fleissig den Java-Praktomaten!

End

Edison and Bulb Tests

"If I find 10,000 ways something won't work, I haven't failed. I am not discouraged, because every wrong attempt discarded is another step forward."

T. A. Edison

"Müsste Edison eine Nadel im Heuhaufen finden, würde er einer fleißigen Biene gleich Strohhalm um Strohhalm untersuchen, bis er das Gesuchte gefunden hat."

- Nikola Tesla, New York Times, 19. Oktober 1931

Aber: Ein Wort der Warnung

[Edison] had no hobby, cared for no sort of amusement of any kind and lived in utter disregard of the most elementary rules of hygiene. [...] His method was inefficient in the extreme, for an immense ground had to be covered to get anything at all unless blind chance intervened and, at first, I was almost a sorry witness of his doings, **knowing that just a little theory and calculation would have saved him 90% of the labour.**

But he had a **veritable contempt for book learning and mathematical knowledge,** trusting himself entirely to his inventor's instinct and practical American sense.

Nikola Tesla

LAUSD payroll fiasco

- http://catless.ncl.ac.uk/Risks/24.84.html
- <"David E. Ross" <david@rossde.com>> Thu, 27 Sep 2007 16:56:28 -0700
- Relating to Steve Bellovin's ``Deploy first, test later'' (RISKS-24.83), a similar fiasco has been afflicting employees in the Los Angeles Unified School District (LAUSD) since early this year. LAUSD is the second largest K-12 public school system in the nation.
- Some eight months after "going live" with their new payroll system, employees are still receiving incorrect paychecks or no paychecks at all. The administration does not yet know whether correct W2 forms will be issued in January. Employees retiring cannot get correct pension benefits.
- Of course, when the new system was deployed, there were no contingency plans to roll back to the prior system. By now (after a delay of months), a roll-back is likely to be impossible.
- [On 1 Oct 2007, an NPR report mentioned that Deloitte Touche had received $95M for the original system, which did not work, and that another $10M had been spent on contracts aimed at fixing the system -- which to date still does not work. PGN]

Another case of Deploy First, Test Later (Re: Ross, RISKS-24.84)

- http://catless.ncl.ac.uk/Risks/24.85.html#subj6.1
- <Huge <huge@huge.org.uk>> Wed, 10 Oct 2007 15:00:03 +0100
- Many years ago, I was involved in 'porting' the payroll system of a large British TV company from an ICL 1902S to an ICL 2903 (told you it was a long time ago). We actually rewrote the whole thing in RPG2, from its original Autocoder. We moved the data between the two machines on punched cards.
- So, come the day of the first parallel run, after months of testing, and the results were different. Not much, a few pennies, but different nonetheless. Huge panic, much headless chicken behaviour until we discovered that ... the old system was the one that was wrong. And had been for years.
- So, what have we learned in the intervening 30 years? Not a whole lot, it appears.

Definition neuer Ausnahmen

Benutzung von benutzerdefinierten Ausnahmen möglich und empfehlenswert !

```
class TestException extends Exception {
  public TestException () {
    super();
  }
}
class SpecialAdd {
  public static int sAdd (int x, int y)
                          throws TestException {
    if (y == 0)
      throw new TestException();
    else
      return x + y;
  }
}
```


Deklaration und Propagation von Ausnahmen

- Wer eine Methode aufruft, die eine Ausnahme auslösen kann, muß
  - entweder die Ausnahme abfangen
  - oder die Ausnahme weitergeben (***propagieren***)
- Propagation in Java: Deklarationspflicht mittels **throws** (außer bei Error und RunTimeException)

```
public static void main (String[] argv){
  System.out.println(SpecialAdd.sAdd(3,0));
}
```

Java-Compiler: Exception TestException must be caught, or it must be declared in the throws clause of this method.

Bruch von Verträgen und Ausnahmen

- Man kann Verträge auch mit Ausnahmetests prüfen:

```
class ContractViolation extends Exception {..};
class ParameterContractViolation extends ContractViolation {..};
class FigureEditor{
  draw (Figure figure) throws ContractViolation {
    if (figure == null) throw new ParameterContractViolation();
  }}
```

- im Aufrufer:

```
try {
   editor.draw(fig);
} catch (ParameterContractViolation) {
   fig = new Figure();
   editor.draw(fig);
}
```

- Vorteil: kontrollierte Reaktion auf Vertragsbrüche.