2 Worte stecken in PatternTesting – *Pattern* und *Testing*. Auf beide Bestandteile werde ich in diesem Vortrag eingehen, ehe ich zu PatternTesting selbst kommen werde (das im übrigen unter http://patterntesting.sourceforge.net zu finden ist).

Ursprünglich wollte ich zwar dieses Jahr das JFS aus der Zuhörer-Perspektive geniesen, aber unser Marketing hat mich zu einem Vortrag „ermuntert“.

Häßlicher Code begegnet uns fast täglich in den unterschiedlichsten Projekten. Aber auch auf den ersten Blick „schöner“ Code oder ehemals als „schön empfundener“ Code kann so manche Überraschungen verbergen:

- Haben Sie schon mal Ihren eigenen Code nach x Jahren wiedergesehen?
- Würden Sie Ihn nochmals so schreiben?
- Glauben Sie, dass Ihr jetziger Code auch noch nach 5 Jahren Ihre Zustimmung findet?

Ziel dieses Vortrags (und auch von PatternTesting) ist es, dass Sie Ihren eigenen oder auch fremden Code mit anderen Augen sehen. Welches sind denn die „häßlichen“ Stellen? Und sind denn die schönen Stellen wirklich so „schön“ - oder bergen sie Risiken und Nebenwirkungen?

about me:
bis 1999 C++, Fan v. Design-Pattern
ab 1999 Java, Wechsel zu Neuen Markt
ab 2000 Wechsel in Finanzbereich (kurz bevor der Neue Markt zusammengebrochen ist)
ab 2005 AspectJ
2007: Finanzkrise, aber immer noch im Finanzbereich tätig

about agentes:
Finanzbereich
langlebige Software
(im Gegensatz zum Neuen Markt)

- Pattern begegnen uns heute überall
- PatternTesting noch nicht (wer kennt es?)
- dazu brauche ich Ihre Hilfe

1997 GoF-Buch „Design Pattern“ (Gamma et al)
1998 POSA (Pattern Oriented Sotware Architecture)
(Buschmann)

Wer kennt GoF-Buch?
Wer kennt POSA-Buch?

Beispiel:
Singleton (GoF-Buch)
Schichtenarchitektur (POSA-Buch)

Bildquelle: http://www.flickr.com/photos/kh-67/3339157498/

typische Signatur für Singleton oder das Highlander-Prinzip („Es kann nur einen geben“)
Das Singleton (aus dem GoF-Buch) ist das einfachste und wohl auch am meisten verwendet Pattern. Aber: Pattern haben neben Vorteile auch Nachteile (meist: zusätzliche Komplexität). Statt Singleton können auch statische Methoden verwendet werden (static import).
Vorteil Singleton: kann abgeleitet werden

Factory (Erzeuger) wird gern eingesetzt bei Objekten, die sich schwer erzeugen lassen.

Factories begegnen uns überall im JDK:
java.lang.management.ManagementFactory
java.security.cert.CertificateFactory
java.security.KeyFactory
java.util.concurrent.ThreadFactory
javax.xml.parsers.DocumentBuilderFactory
...

Buildquelle: http://www.flickr.com/photos/extranoise/278465198/

Schichtenarchitektur wird eingesetzt, um Komplexität besser in den Griff zu bekommen.

bekannteste Beispiel: IP-Stack

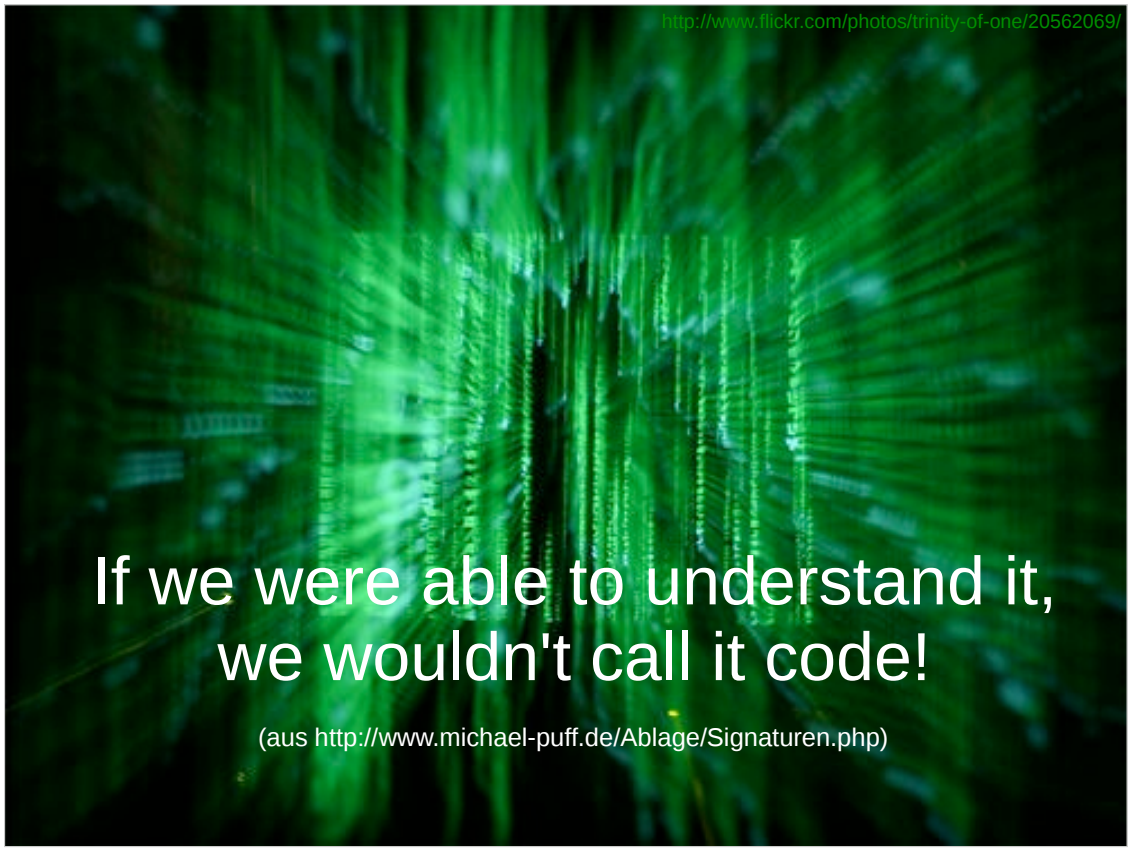
Bildquelle: http://www.flickr.com/photos/ann/1329361/

1998 gab es dann auf der OOPSLA den ersten Beitrag zu Anti-Pattern von Thomas J. Mowbray.
Eines der bekanntesten Anti-Pattern ist sicherlich der Spaghetti-Code.
Symptome: jeder ruft jeden auf, verschlungene Aufruf-Pfade
Gegenmaßnahme: Schichtenarchitektur

Buildquelle: http://www.flickr.com/photos/telstar/61497944/

Lavaflow / Dead Code:
„aktiver“ Code fliest um „toten“ Code herum
Symptome: lange Methoden, viele if-Abfragen

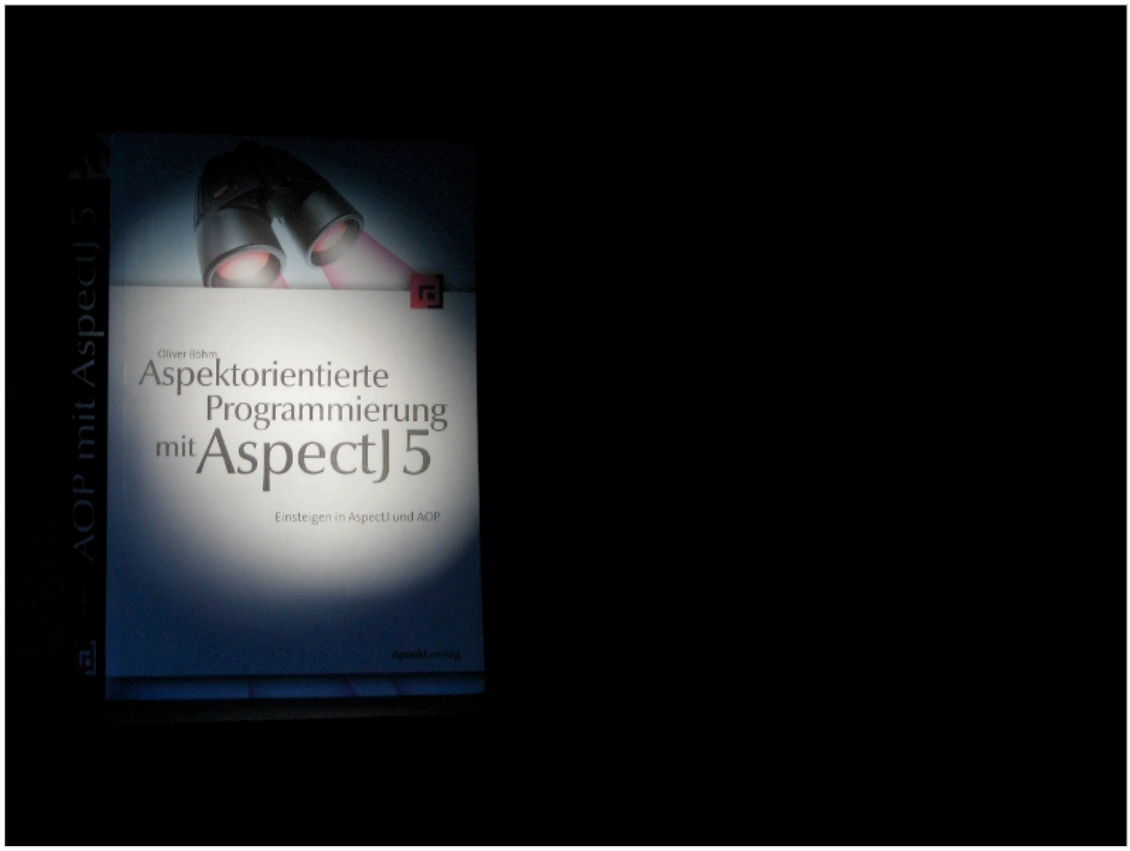
Bildquelle: http://www.flickr.com/photos/wildphotons/2443327380/



Lavaflow führt meistens dazu, dass der Code unverständlich wird:

- It smells!

Bildquelle: http://www.flickr.com/photos/trinity-of-one/20562069/



Während der Recherchen über AOP/AspectJ bin ich mehrfach über Pattern gestolpert:

- Diplomarbeit von Jan Hannemann:
  hier wurden alle 23 GoF-Pattern in Form einer AspectJ-Bibliothek bereitgestellt
  Bsp: Singleton: Konstruktor liefert immer das gleiche Objekt zurück
  http://www.cs.ubc.ca/~jan/AODPs/
- PatternTesting 0.3
  Framework zum Aufspüren von Anti-Patterns von Vincent Massol und Matt Smith

Beispiel für generierten Code (Eclipse):

```
try {
    ...
} catch (MalformedURLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Problem: statt e.printStackTrace() sollte die Exception mit Hilfe von Log4J oder anderes Logging-Framework protokolliert werden.

=> PatternTesting gibt hier eine Warnung aus:

No logging should be done using the default output and error streams.

In diesem Teil gehe ich auf folgende Fragen entstanden:

- Wie ist PatternTesting entstanden?
- Was muss man über die PatternTesting-Architektur wissen?
- Für welche Anti-Patterns gibt es Unterstützung?

2002: Projekt-Start
2003: 0.1
2004: 0.3

Einige der Ideen aus PatternTesting sind in verschiedene Projekte eingeflossen:

- eBanking-Lösung für Giropay
- SSO-Lösung
- Altersverifikation

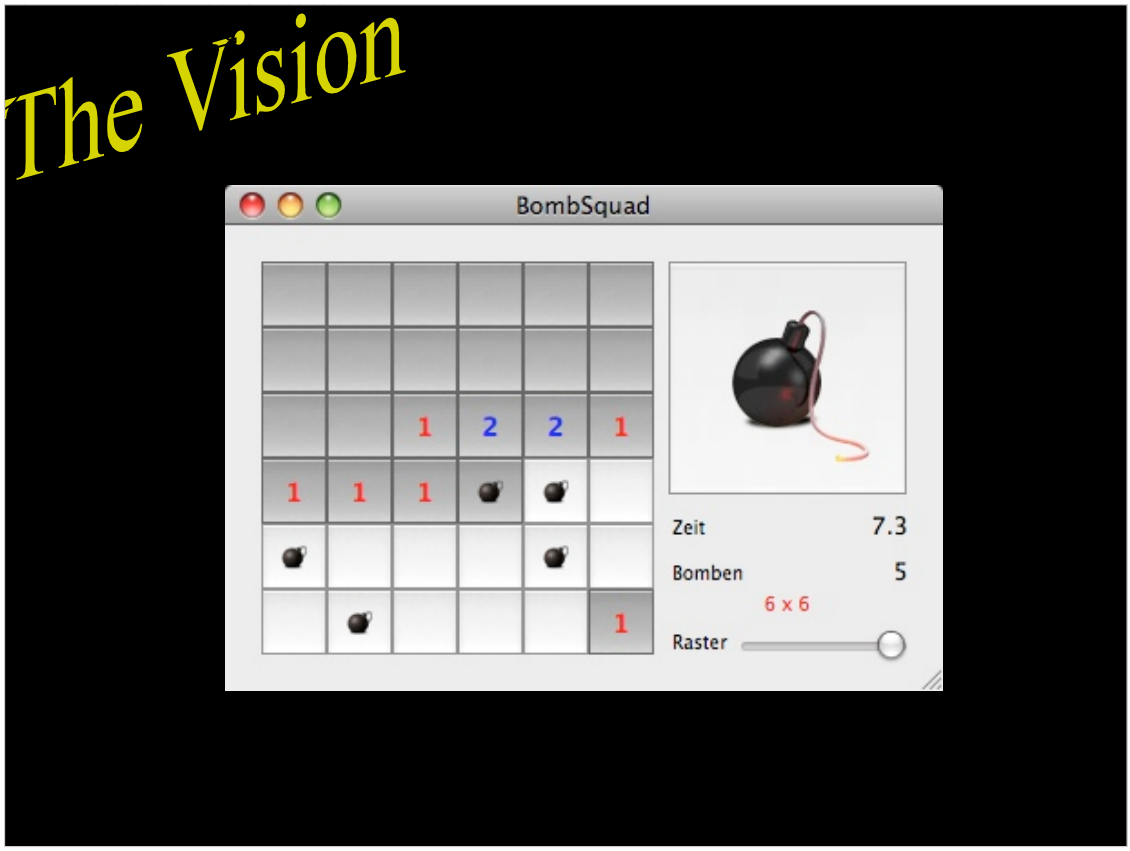
Wann geht’s weiter?

ab 2004 Stillstand
2007: Kontakt zu Vincent Massol

- Einladung zu AOP-Day 2007
- Hilfe f. PatternTesting angeboten

Einige Ideen aus den verschiedenen Projekten sind inzwischen wieder zurückgeflossen (z.B. ProfileAspect)

2008: 0.5, 0.6, 0.8
2009: 0.9
2010: 1.0



Vision von PatternTesting:
„To be the Minesweeper for your Java Code!“

Kennen Sie Minesweeper? Bei diesem Spiel geht es darum, versteckt Mine auf dem Spielfeld zu finden, ohne dass man dabei drauf geht. Ähnlich verhält es sich mit PatternTesting: finden Sie die versteckten Zeitbomben in Ihrem Code, bevor jemand (Sie?) zu Schaden kommt.

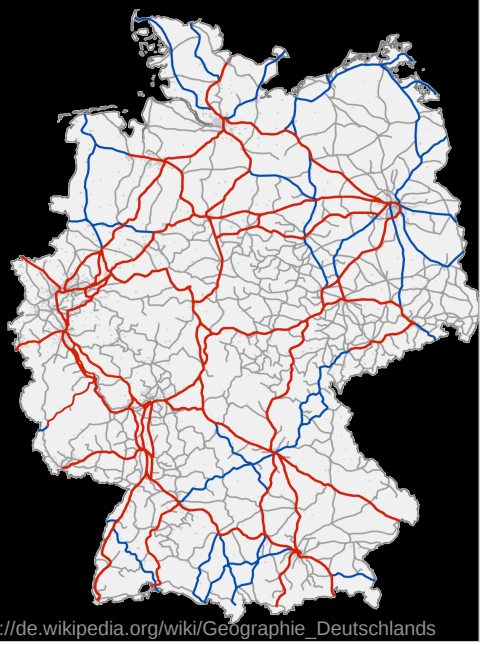
Räumen Sie die Minen aus Ihrem Code und bleiben Sie in ruhigem Fahrwasser!

Bildquelle: http://www.flickr.com/photos/neodelphi/1707209395/
(alter Minenräumer aus dem 2. Weltkrieg, Australien)

Ursprung: Check.CT + Check.RT

- patterntesting-rt (Runtime)
  enthält sämtliche Annotationen der anderen Komponenten (=> andere Komponenten können ohne Compile-Fehler entfernt werden)
- patterntesting-check-ct:
  Compile-Time-Checks (z.B. e.printStackTrace)
- patterntesting-check-rt:
  Runtime-Checks
- patterntesting-exception:
  Exception-Wrapper, Test-Support f. Exceptions
- patterntesting-concurrent:
  Multithreading-Support, Test-Unterstützung
- patterntesting-tools:
  Ant-Support, Maven-Plugin (veraltet)
- patterntesting-samples:
  Beispiele

Wie findet man
tote Code-Strecken?



Quelle: http://de.wikipedia.org/wiki/Geographie_Deutschlands

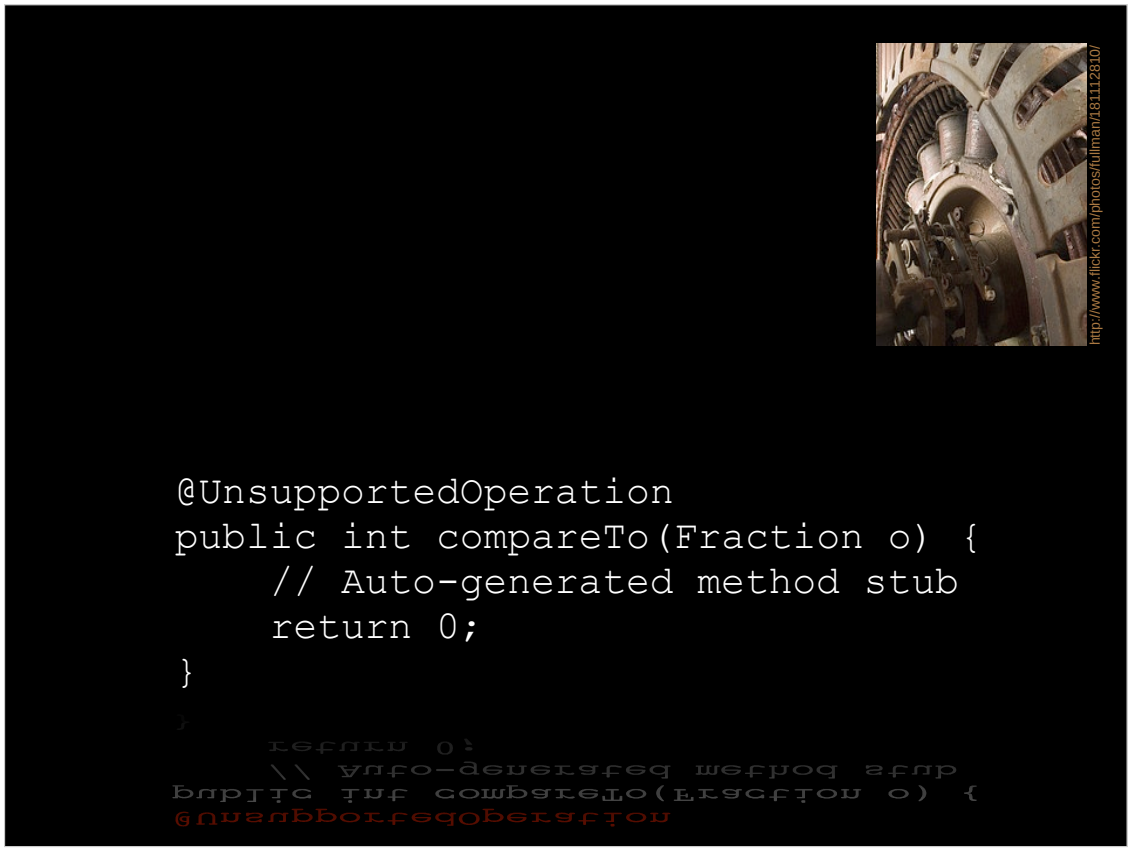
Wie findet man tote Code-Strecken?
Welche Punkte (Bahnhöfe) werden angefahren?

http://www.flickr.com/photos/mormon_railfan/2615245227/

Idee: Zug (Programm) starten

- ClasspathMonitor (Java):
  - kennt „Streckennetz“
  - kann tote Strecken (unbenutzte Klassen) finden
- ProfileStatistic (Java + AspectJ)
  - merkt sich Haltestellen (Methoden)
  - kann tote Haltestellen (unbenutzte Methoden) finden

=> kann zur Bekämpfung des JavaFlow-Antipattern eingesetzt werden.



„Faulty Default Implementation“

```
@UnsupportedOperation
public int compareTo(Fraction o) {
   // Auto-generated method stub
   return 0;
}
```

Problem: Verlass auf (Default-)Implementierung

- Code-Generierung (MDA)
- Eclipse

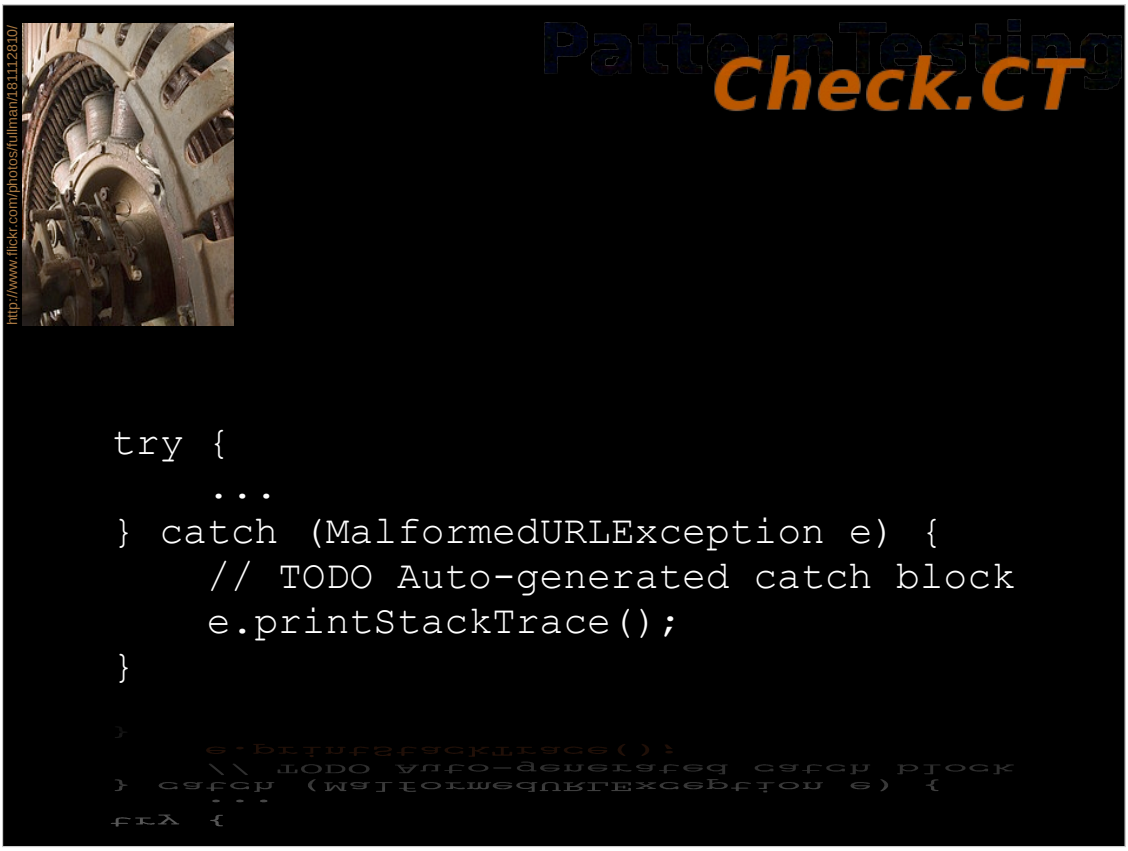
(Tipp f. Eclipse: Template abändern und RuntimeException werfen)

Vorteil gegenüber selbstgestrickte RuntimeException: kann zur Javadoc-Generierung vewendet werden (Beispiel: @Deprecated)

Buildquelle: http://www.flickr.com/photos/fullman/181112810/

Runtime enthält sämtliche Annotationen für anderen PatternTesting-Bibliotheken
Vorteil:

- andere Bibliotheken können ohne Compile-Fehler wieder entfernt werden
- kann auch als reine Java-Lib verwendet werden
- kann auch als Doku verwendet werden




```
try {
   ...
} catch (MalformedURLException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
}
```

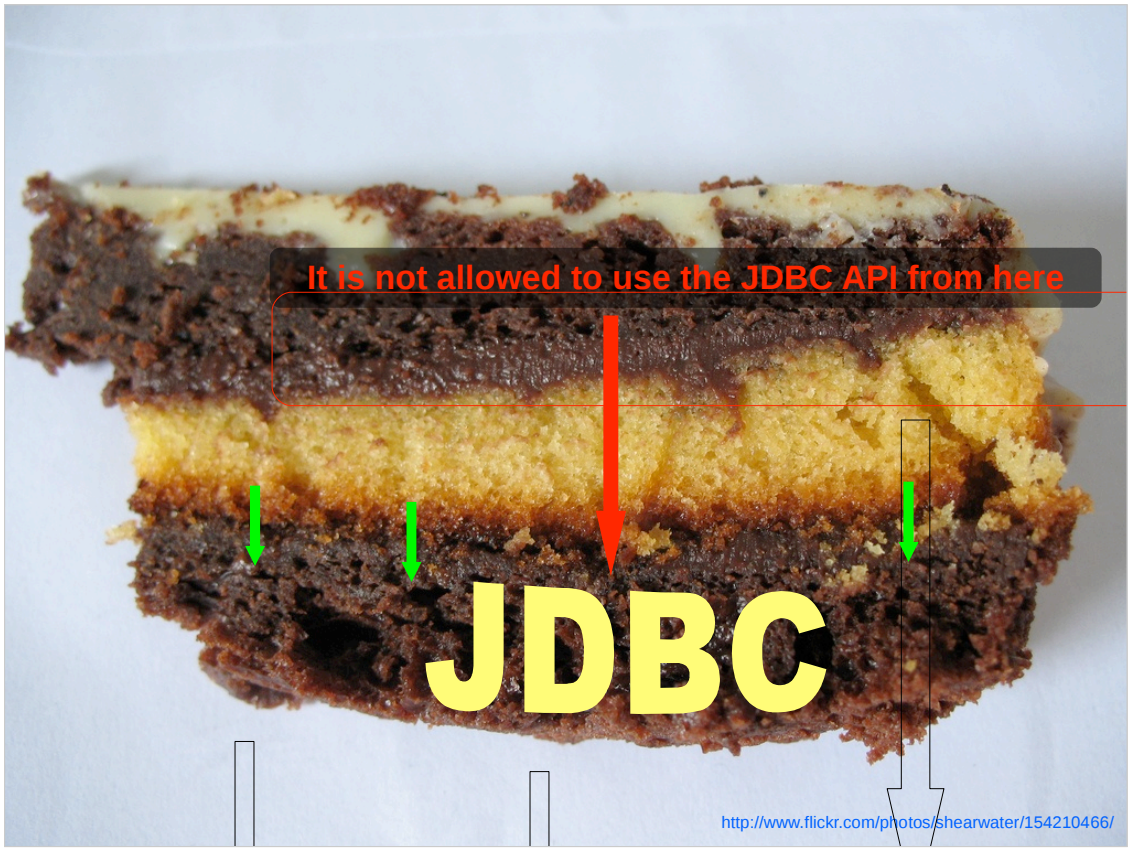
Vor/Nachteil gegenüber FindBugs / PMD / ...:

- Teil des Compile-Vorgangs:

  No logging should be done using the default output and error streams.

Empfehlung: Eclipse 3.4 + AJDT >= 1.6.2 (bis zu 30x schneller als Eclipse 3.3)

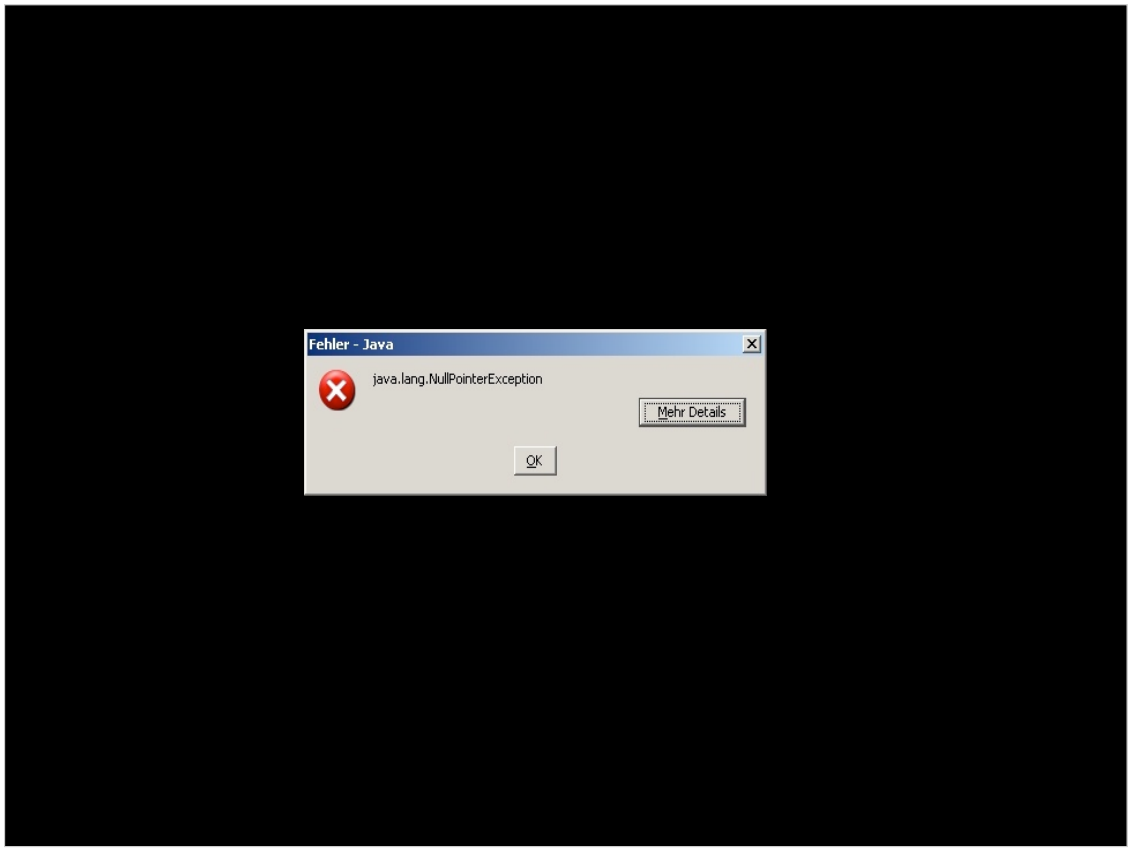
Buildquelle: http://www.flickr.com/photos/fullman/181112810/



Beispiel für Schichtenarchitektur

- unterste Schicht: Persistenzschicht
- Anwendungsschicht: darf nicht direkt auf JDBC zurgreifen
- wird durch PatternTesting unterstützt (@DamnJDBC)

Bildquelle: http://www.flickr.com/photos/shearwater/154210466/



Obwohl Java keine Pointer kennt, ist die NullPointerException (NPE) einer der (wenn nicht sogar *der*) häufigste Fehler, der in der freien Wildbahn anzutreffen ist. Das Fatale an einer NPE ist, dass die Ursache dafür oftmals an einer ganz anderen Stelle zu suchen ist, als es der Stacktrace vermuten lässt.
Die eigentliche Ursache für NPEs sind oftmals ein allzu sorgloser Umgang mit null-Werten.


```
List<Bug> findBugs() {
    ...
    // nothing found
    return null;
}
```

Ist es eine gute Idee, dass findBugs() *null* zurückliefert, wenn keine Bugs gefunden wurde?

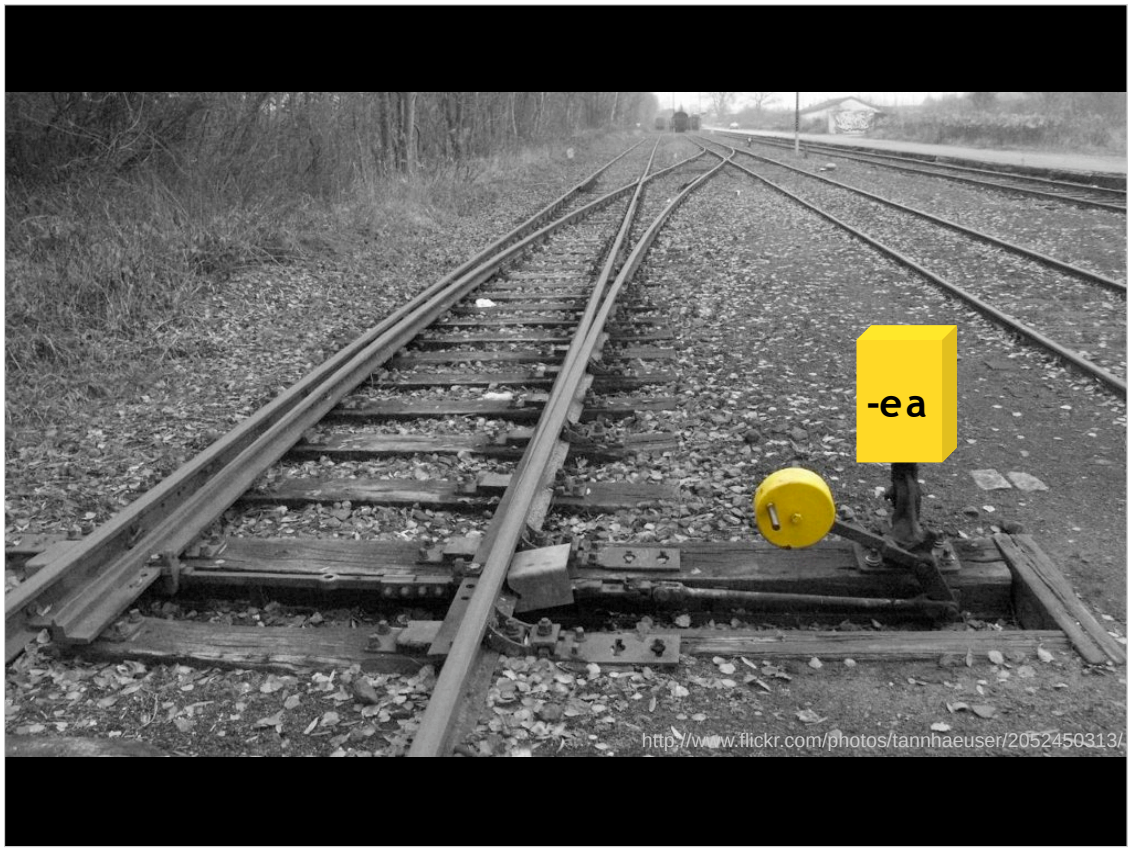
Nein! Daher meldet PatternTesting Check.RT auch ein

AssertionError: findBugs() returns null!

zur Laufzeit. wenn *null* zurückgegeben wird.

Was heißt „nix“? Fast immer lässt sich das durch eine eigenes „Null“-Objekt (anstatt „null“-Value) ausdrücken (**Nullsemantik**).
Im Falle einer Liste ist dies die leere Liste:

```
List<Bug> findBugs() {
    ...
    // nothing found
    return Collections.EMPTY_LIST;
}
```



Philosophie hinter PatternTesting Check.RT ist, dass alle Laufzeit-Überprüfungen explizit eingeschaltet werden müssen. Sie sind an die Assertions gekoppelt und lassen sich über

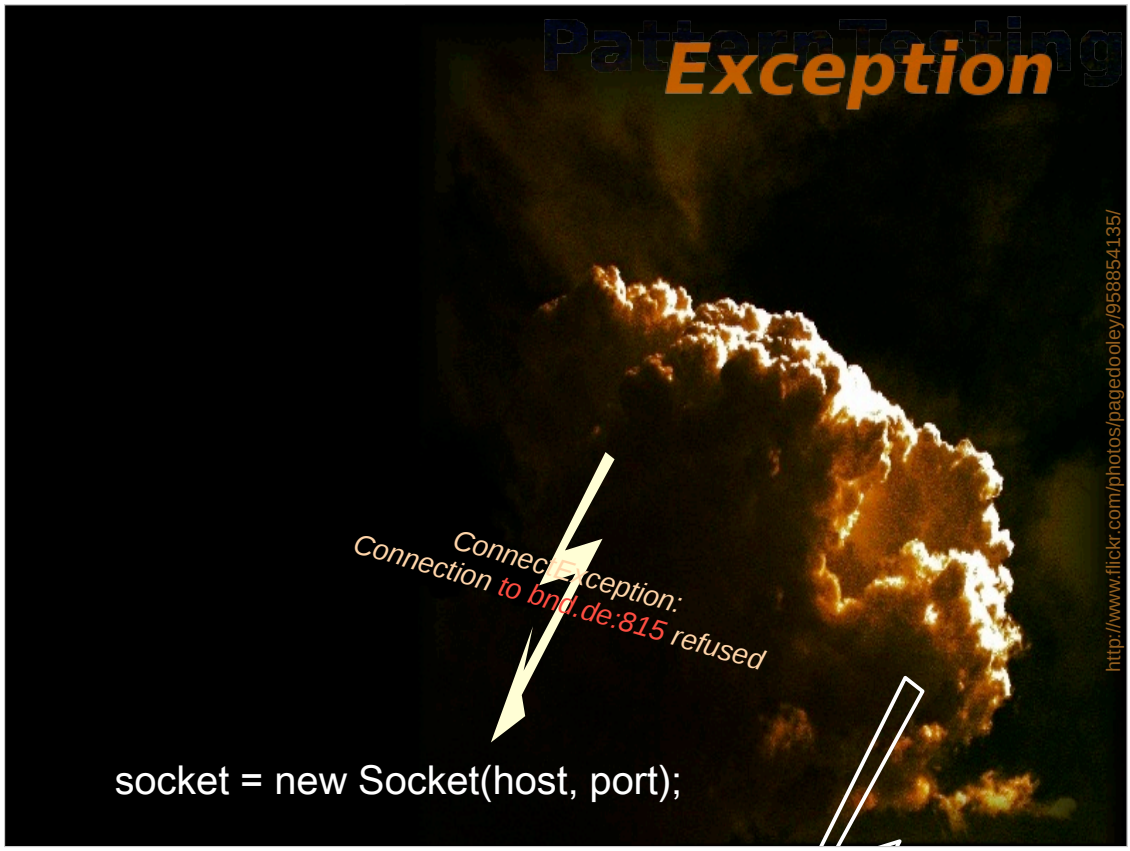
```
java -ea ...
```

aktivieren.

Vorteil:

- können für Produktion abgeschaltet werden (minimaler Overhead, keine „Fehl“-Alarme, falls Nullsemantik nicht sauber durchgezogen wird)
- zum Testen wird mit Überprüfung gestartet (z.B. Default bei JUnit-Tests über Maven)

Bildquelle: http://www.flickr.com/photos/tannhaeuser/2052450313/



Viele Exceptions liefern neben dem Stacktrace wertvolle Hinweise, was die Ursache für den Fehler sein kann. Dies gilt aber leider nicht für alle Exceptions:

- ConnectionException liefert leider nicht den Host mit
- FileNotFoundException f. Input/OutputStream liefert den Filenamen nur relativ
- File.createNewFile() liefert nichtssagende IOException, falls z.B. Verzeichnis fehlt
- ...

Zum Testen gibt es eine ExceptionFactory(MBean), mit der sich zur Laufzeit Exceptions provozieren lassen.

Bildquelle: http://www.flickr.com/photos/pagedooley/958854135/

Der Umgang mit Parallelität ist recht mühsam, auch wenn Java Untertützung für Multi-Threading mitbringt. PatternTesting bringt Unterstützung per Annotationen mit:

- @RunParallel
- @RunBackground
- @ForceThreadSafeCollection

Auch das Testen von parallelen Abläufen ist schwierig. Daher bietet PatternTesting über

- @TestThread

die Möglichkeit, den Thread-Wechsel bei mehreren gleichzeitig ablaufenden Threads zu „erzwingen“.

Status: experimentell (noch keine Projekt-Erfahrung)

Bildquelle: http://www.flickr.com/photos/lembagg/2541715558/

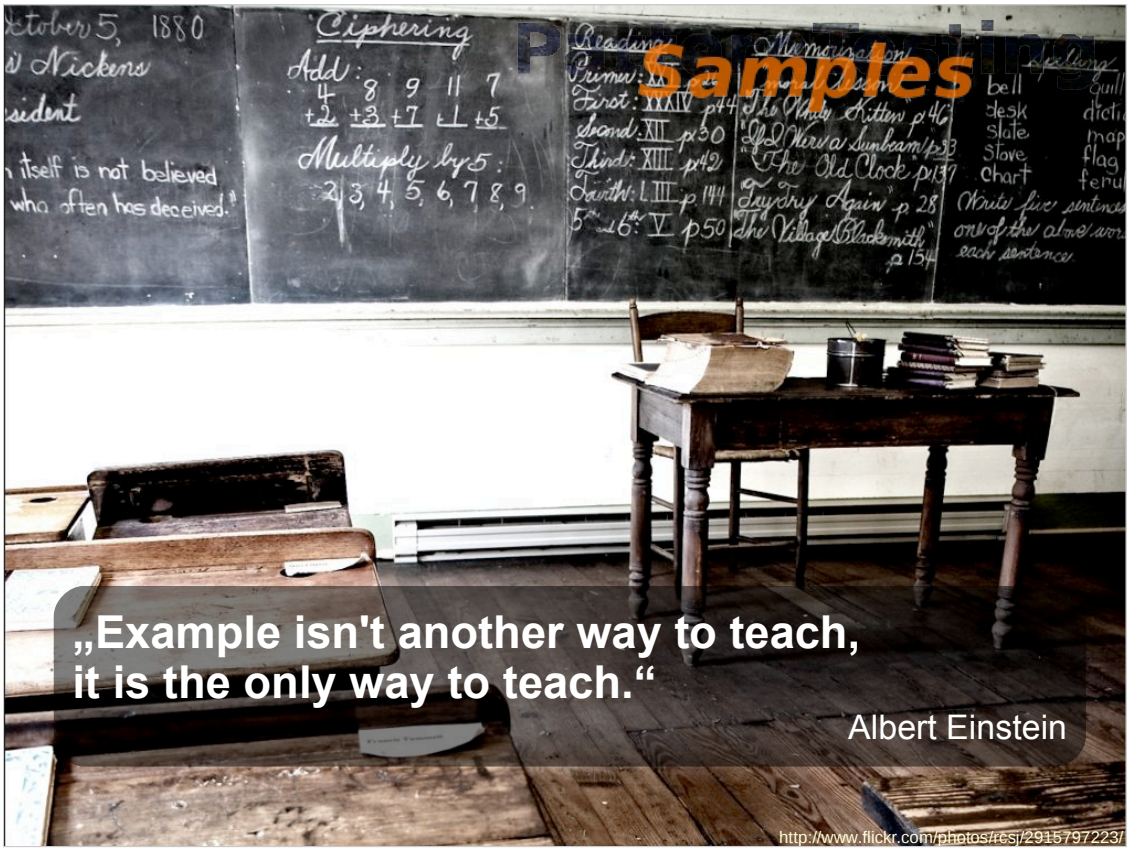
Wohin geht die Reise? Aktuell ist die Version 0.9.6 draußen. Für nächstes Jahr (2010) ist Version 1.0 geplant:

- Stabilisierung der API
- deprecated Code wird rausfliegen
- Java 6
- Unterstützung WLS-Classloader

größte Herausforderung:

- Dokumentation
- WLS-Classloader

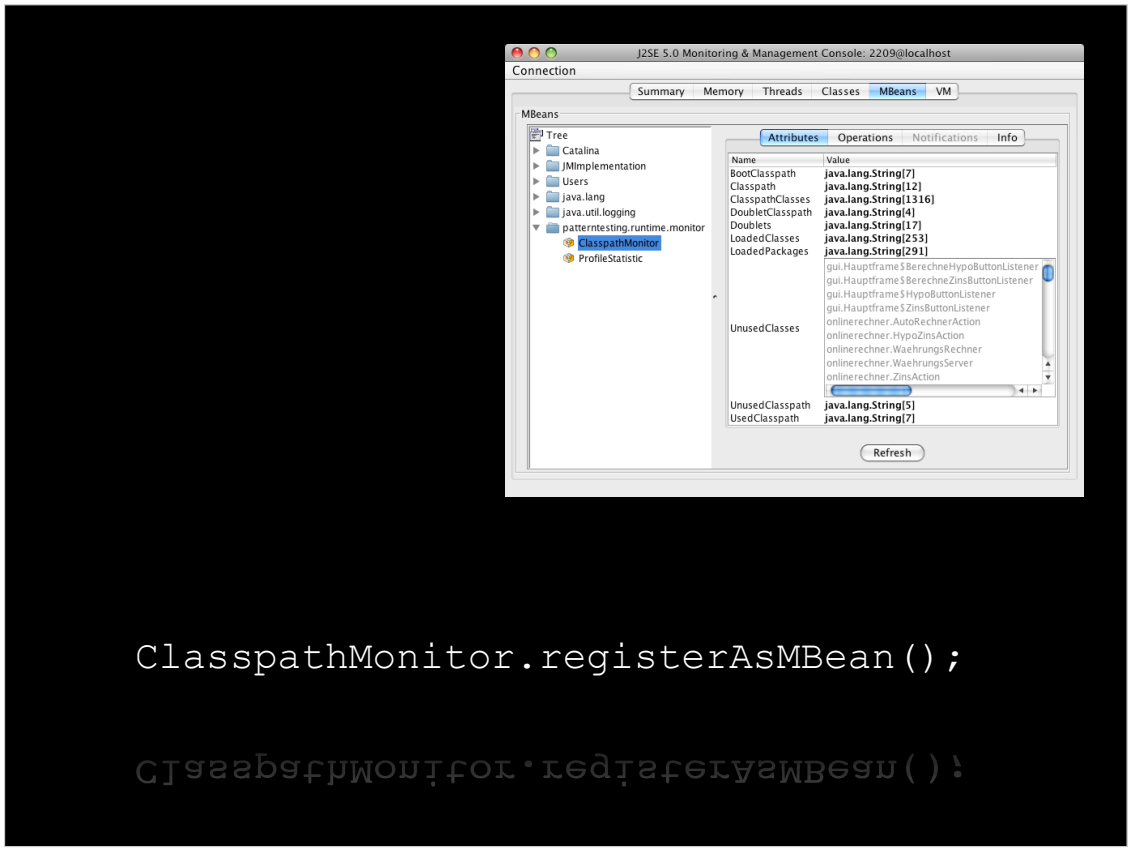
Bildquelle: http://www.flickr.com/photos/jsome1/2226394415/



PatternTesting Samples ist aus zwei Gründen heraus entstanden:

- zur Dokumentation bzw. zum leichteren Erlernen (s. Einstein-Zitat)
- zum Testen der verschiedenen PatternTesting-Bibliotheken

Bildquelle: http://www.flickr.com/photos/rcsj/2915797223/



Um den ClasspathMonitor auszuprobieren, reichen 4 Bibliotheken aus:

- patterntesting-rt-0.9.6.jar
- aspectjrt.jar
- commons-lang-2.3.jar
- commons-io-1.3.1.jar

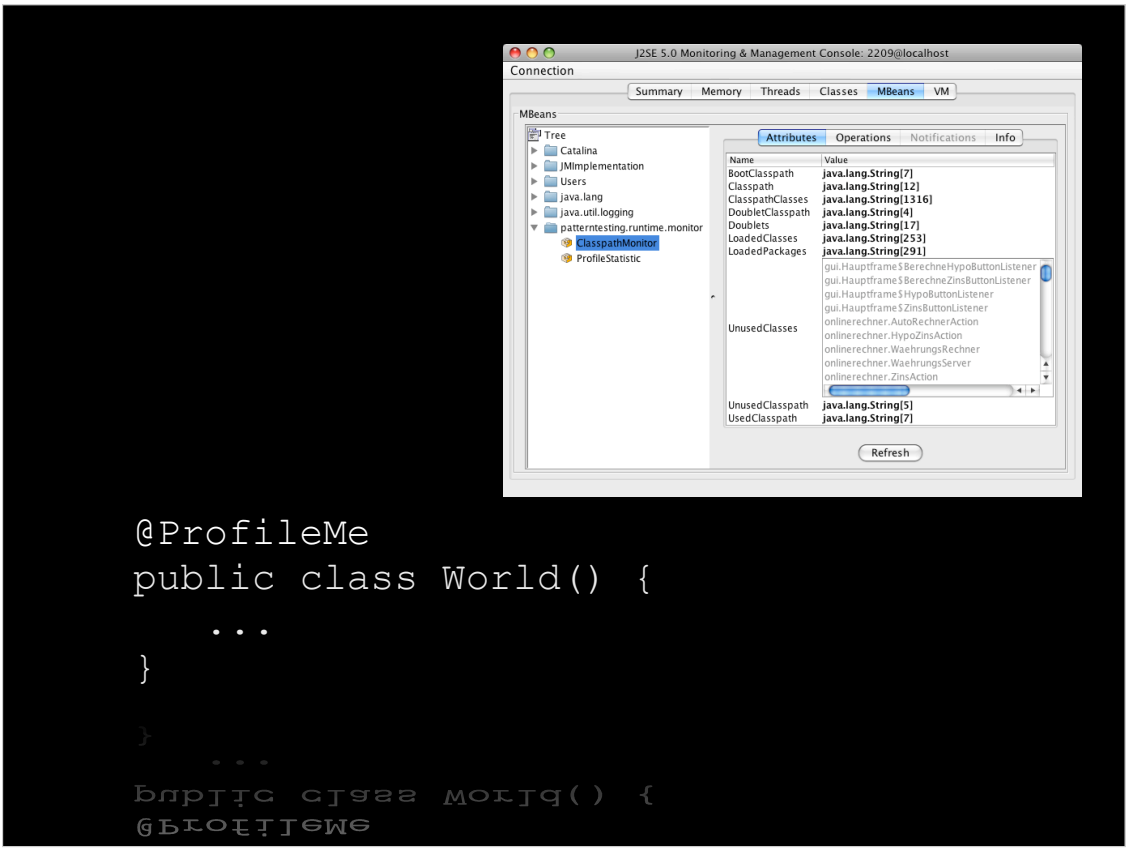
Eine Demo dazu gab's auf den Stuttgarter Test-Tagen 2009. Die Anleitung dazu findet sich in meinem Blog (http://oli.blogger.de) unter

- http://oli.blogger.de/stories/1394945/ bzw.
- http://oli.blogger.de/static/antville/oli/files/patterntestingeinstieg.pdf

Wichtig beim Start des Programms bzw. AppServer sind die Optionen

-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.local.only=false -ea

um über jconsole auf die ClasspathMonitorMBean zugreifen zu können.

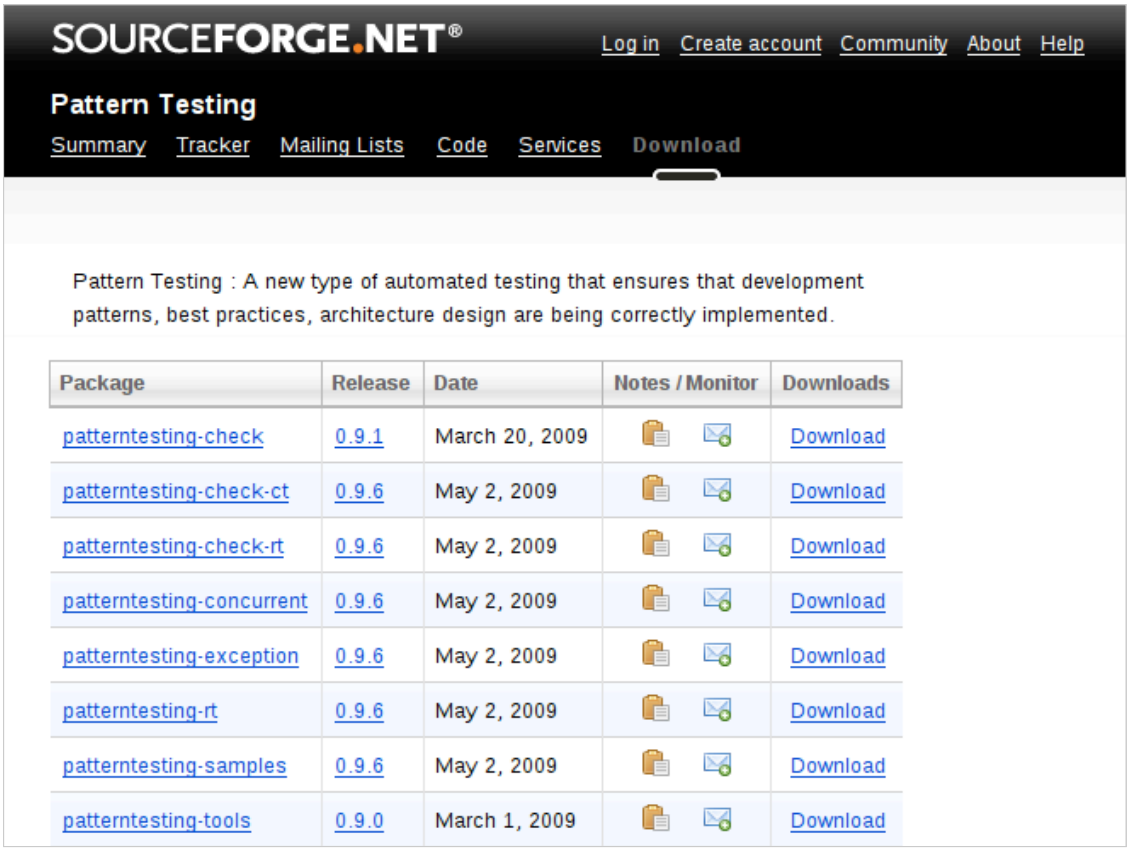


evtl. vorführen

„Ich brauche Hilfe!“ aus Sicht

- des Anwenders
- des Projekts

Bildquelle: http://www.flickr.com/photos/zanthia/3260169669/



Man kann die verschiedenen PatternTesting-Bibliotheken ganz normal über http://patterntesting.sourceforge.net herunterladen (Download-Link ist auf der Homepage). Bei patterntesting-rt.jar sollte man immer die Neueste nehmen (die sollte abwärtskompatibel sein – falls nicht => Bugreport), bei den anderen Bibliotheken ist es egal, die haben keine Abhängigkeiten untereinander.
Jede Bibliothek gibt es als .jar, sources.jar und test-sources.jar.

Wer sich nicht mit den Abhängigkeiten rumschlagen will, kann selbstverständlich auch Maven verwenden: die *groupId* ist net.sf.patterntesting, die *artifactId* z.B. patterntesting-rt oder patterntesting-exception:

```
<dependency>
  <groupId>net.sf.patterntesting</groupId>
  <artifactId>patterntesting-rt</artifactId>
  <version>0.9.6</version>
</dependency>
```

In der Regel werden die Bibliotheken parallel zum Donwload und für Maven bereitgestellt. In Maven-Repository (ibilio.org) sind sie dann i.d.R. einen Tag später verfügbar (nach Sync-Lauf).

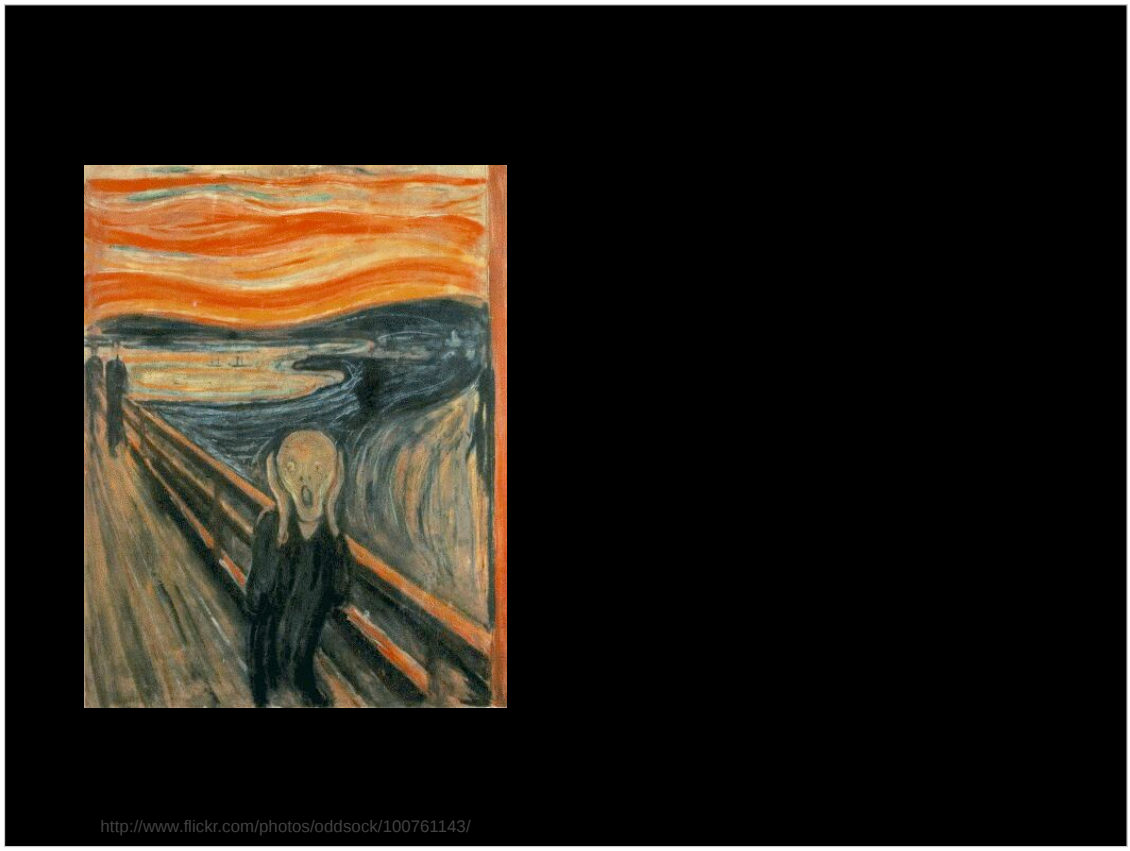
Bildquelle: http://www.flickr.com/photos/designpackaging/3349979152/in/photostream

How to contact?

http://www.flickr.com/photos/paulgi/283789943/

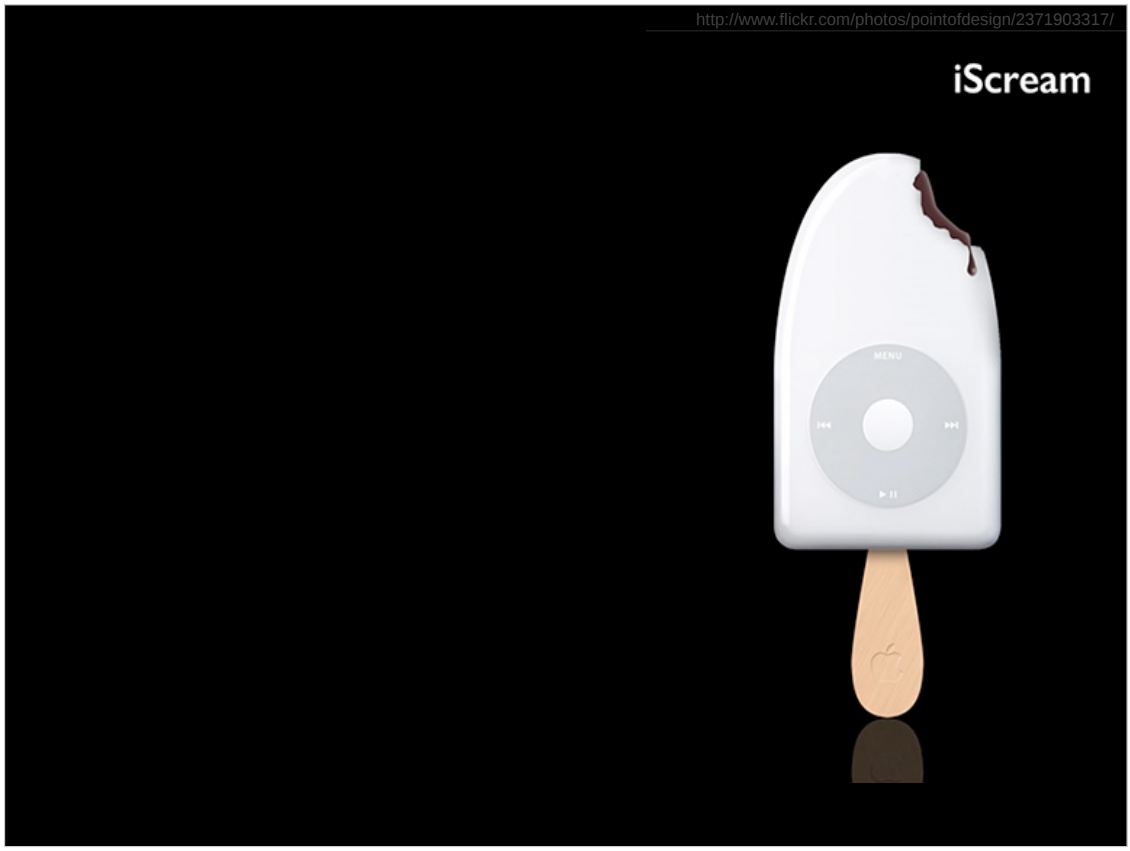
viele Möglichkeiten:

- persönlich
- Mailing-List auf patterntesting.sf.net
- (JUGS-)Forum
- Feature-Request über patterntesting.sf.net



Möchten Sie manchmal auch nur schreien, wenn Sie Ihren alten Code nach 5 Jahren wieder zu Gesicht bekommen? Warum hat mir damals niemand gesagt, dass ich Reader und Writer statt Input/OutputStream nehmen soll. Oder dass man nicht über System.out loggt.
Gibt es irgendetwas, das geächtet werden soll (z.B. „Log- and Throw“-Antipattern, s. http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html)?

Bildquelle:
http://www.flickr.com/photos/oddsock/100761143/
Edward Munch: der Schrei (scream)



Feedback: Fehlt irgendwas? Sollte z.B. bei den JUnit-Antipatterns („Misuse of Assertions“, s. http://www.exubero.com/junit/antipatterns.html) auch JUnit 4 unterstützt werden)?
Gibt es einige wichtige Antipatterns, die fehlen (mit Sicherheit)?
Fehlt es an Beispielen?
Reicht die Dokumentation (wäre schön)?

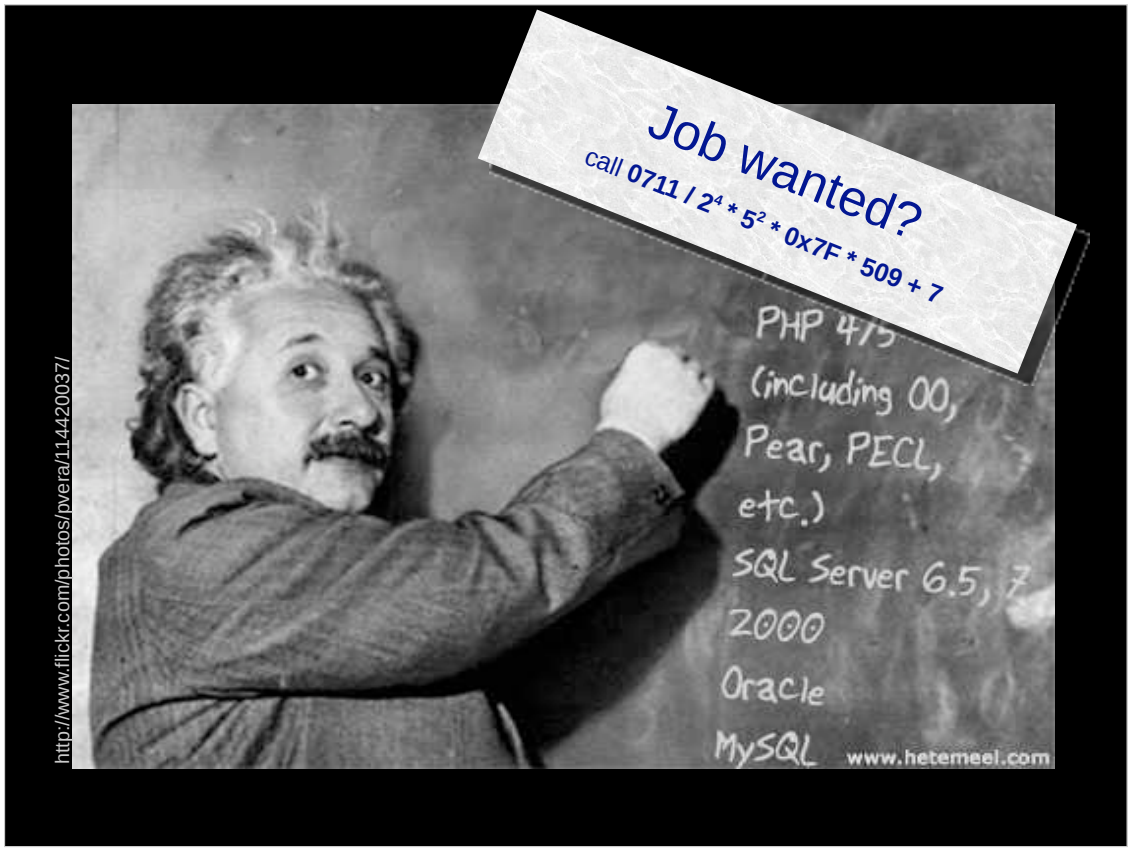
Bildquelle: http://www.flickr.com/photos/pointofdesign/2371903317/

Es existieren noch viele Baustellen wie Dokumentation, Testen, Tool-Unterstützung (Ant, Maven-Plugin), für die Hilfe in Form von

- Code-Vorschläge
- Test-Fälle
- Tipps u. Tricks (z.B. f. Maven-Build)
- Mitarbeit
- Öffentlicer CI-Server (continuum, Hudson, ...)
- ...

gesucht wird.

Bildquelle: http://www.flickr.com/photos/untitlism/2609684221/



Falls Plan A (freiwillige Helfer) nicht greift, gibt es auch noch einen Plan B: Für den Einsatz und Weiterentwicklung von PatternTesting suchen wir noch Mitarbeiter:

- AspectJ-Kenntnisse wünschenswert
- Maven-Kenntnisse wünschenswert

Bildquelle: http://www.flickr.com/photos/pvera/114420037/

PatternTesting lebt von Erfahrungen aus diversen Projekten. So hatten wir schon in vielen Projekten mit dem Classpath zu kämpfen (=> ClasspathMonitor). Thema Exceptions: *Aussagekräftige* Exception sind leider Mangelware in vielen Projekten, aber auch im JDK. Immer die gleichen „üblichen Verdächtigen“ mit zusätzlichen Information anzureichern ist stupide. Man muss es ja nicht jedesmal selbst machen – hier kann man z.B. auch von PatternTesting profitieren.

Wenn Sie also Projekte haben (oder planen)

- in denen Sie Bedarf für PatternTesting sehen oder
- wo sie denken, dass das eine oder andere in PatternTesting besser aufgeben wäre

kontaktieren Sie uns.

Bildquelle: http://www.flickr.com/photos/pvera/114420037/

Oliver.Boehm@agentes.de

0711/25857-207
oli.blogger.de

Sollte jemand auf den Folien einen Fehler bzw. Bug entdecken, darf er gern auf mich zukommen. Unter http://oli.blogger.de gibt es einige Artikel auch zu PatternTesting. PatternTesting selbst ist unter http://patterntesting.sourceforge.net beheimatet.

Bildquellen: http://www.flickr.com/photos/automania/83183437/
Muttis Album