3.– 6. September 2012
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

„Continuous Bugfixing“. Wie man in einem „10 Mio. Lines of Code Projekt“ statische Codeanalyse einführt …

Holger Thom

main {GRUPPE}

h.thom@main-guppe.de

Agenda

- **Grundlegende Konzepte und Begriffe**
- **Statische Code-Analyse in einem Großprojekt**
- **Statische Code-Analyse mit health4j**

Statische Code-Analyse: grundlegende Konzepte

- **Definition**
  - Statische Codeanalyse oder kurz statische Analyse ist ein statisches Software-Testverfahren. Der Quelltext wird hierbei einer Reihe formaler Prüfungen unterzogen, bei denen bestimmte Sorten von Fehlern entdeckt werden können, **noch bevor die entsprechende Software ausgeführt** wird“

    (Quelle: http://de.wikipedia.org/wiki/Statische_Code-Analyse).

→ **Statische Codeanalyse soll qualitative Schwachstellen im Source- Code finden**

→ **Einhaltung von Architekturvorgaben**

→ **Softwarequalität ist auch definiert durch Fehlerfreiheit, Wartbarkeit, Lesbarkeit…**

Qualitative Schwachstellen?

- **Statische Codeanalyse soll offensichtliche Fehler im Code finden**
  (z.B. offensichtliche Nullpointer)

- **Statische Codeanalyse soll Code finden, der eine hohe Fehlerwahrscheinlichkeit aufweist**
  (z.B. eine Code-Stelle if (var1 & var2))

- **Statische Codeanalyse soll Code finden, der schlecht wartbar und schlecht zu verstehen ist**
  (z.B. undokumentierten, komplexen Code)

- **Statische Code-Analyse dient zur „Schulung des Gespürs“ für problematischen Code**
  (z.B. Funktionen mit 10 Parametern sind schwer zu verstehen)

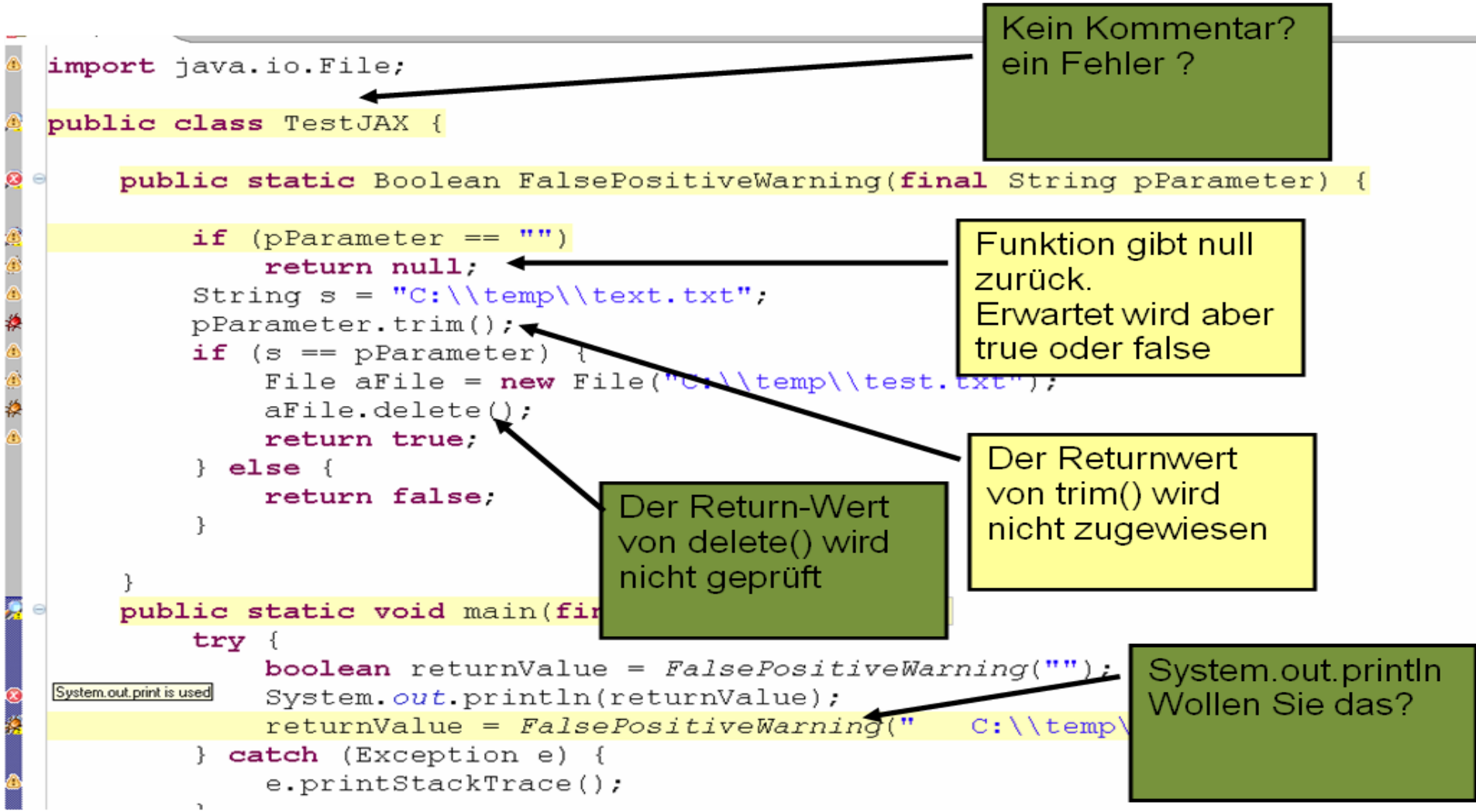
- **Statische Code-Analyse fördert die Einhaltung von Programmier-Konventionen**
  (z.B. jede Funktionen hat genau einen ‚return‘ Zweig)

Kein Compile-Fehler,

aber qualitative Schwachstellen?


```java
import java.io.File;

public class TestJAX {

    public static Boolean FalsePositiveWarning(final String pParameter) {

        if (pParameter == "")
            return null;
        String s = "C:\\temp\\text.txt";
        pParameter.trim();
        if (s == pParameter) {
            File aFile = new File("C:\\temp\\test.txt");
            aFile.delete();
            return true;
        } else {
            return false;
        }

    }
    public static void main(final String[] args) {
        try {
            boolean returnValue = FalsePositiveWarning("");
            System.out.println(returnValue);
            returnValue = FalsePositiveWarning("   C:\\temp\\text.txt   ");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
```

Kein Compile-Fehler, aber qualitative Schwachstellen?



... ist das nur eine „Demo“?

- JBOSS Version 5.0.1
- Klasse org.jboss.mx.timer.JBossTimer

```
222
223     /**
224      * Gets a copy of the flag indicating whether a peridic notification is executed at fi
225      *
226      * @param id The timer notification identifier.
227      * @return A copy of the flag indicating whether a peridic notif
228      */
229     public Boolean getFixedRate(Integer id)
230     {
231        // Make sure there is a registration
232        RegisteredNotification rn = (RegisteredNotification) notifications.get(id);
233        if (rn == null)
234              return null; HEALTH4J >> : NP BOOLEAN RETURN NULL
235
236        // Return a copy of the fixedRate
237           return new Boolean(rn.fixedRate); HEALTH4J >> : BooleanInstantiation
238     }
239
```

Funktion gibt u.U. null zurück. Erwartet wird aber true oder false

- **Zur statischen Codeanalyse existieren viele Open-Source Werkzeuge, die auf verschiedene Schwachstellen optimiert wurden**
  (z.B. Auffinden von undokumentiertem Code)

- **Jedes Werkzeug besitzt eigene Regeln, die es im Code überprüft.**
  (das sog. rule-set)

- **Ein Problem der statischen Code-Analyse ist die Menge an gelieferten Informationen bei großen Projekten (→ Informationsüberflutung)**

- **„Falschmeldungen“ (false-positives).**
  Das Werkzeug meldet einen Fehler, welcher aber aus unterschiedlichen Gründen kein Fehler ist. Der Entwickler muss entscheiden

- **Eine Metrik bildet eine Eigenschaft von Software in einem Zahlenwert ab. Bekannte Metriken sind z.B. Zeilen-Metriken wie „LinesOfCode“**

Best- Practices bei der statischen Codeanalyse mit Open Source Tools

- **Verwende ein einfaches, einheitliches, kleines ‚rule-set'**
  Zu viele (kleinliche) Regeln vermindern die Akzeptanz bei den Entwicklern

- **Binde die Entwickler ein, vermeide „Informationsüberflutung"**
  Zu viele Reports, zu viele bunte Bilder führen zur „Ermüdung" :
  Nach 2 Wochen liest keiner mehr die Reports

- **Informiere aktiv über neue Fehler**
  Kein Entwickler analysiert jeden Tag einen Berg von Auswertungen.
  Wird er direkt über einen schweren Fehler in seinem Code informiert
  (mündlich, per Mail, durch das Team) reagieren die meisten Entwickler schnell

- **Erkenne Fehler früh, führe die Entwickler schnell zu ihren Fehlern**
  Das Auffinden von eigenen Fehlern muss schnell gehen. Idealerweise bereits in der Entwicklungsumgebung. Die Entwickler sind meist unter Druck und haben wenig Zeit….

Best- Practices bei der statischen Codeanalyse mit Open Source Tools

- **Analysiere sowohl in der Entwicklungsumgebung als auch beim Build**
  Unter Zeitdruck in der Entwicklungsumgebung statische Codeanalyse durchzuführen erfordert viel Disziplin. (die nicht jeder hat..;-)

- **Historisiere die Daten, beachte Unterschiede, zeige Trends auf**
  Wichtig bei Alt-Systemen, die man mit hunderten von Fehlern übernimmt. Führt die Weiterentwicklung zu einer Verschlechterung?

- **Automatisiere die statische Code Analyse, gebe Feedback.**
  Zeige die (positiven) Veränderungen auf, wenn Fehler behoben werden

- **Verwende mehrere Open-Source – Werkzeuge für optimale Qualität**
  Jedes Werkzeug hat seine Stärken.

- **Schaffe Verständnis, benenne einen Qualitätsverantwortlichen**
  Die Einführung und die konsequente Durchführung von statischer Code-Analyse muss geplant werden

Bekannte Open-Source Werkzeuge zur statischen Codeanalyse

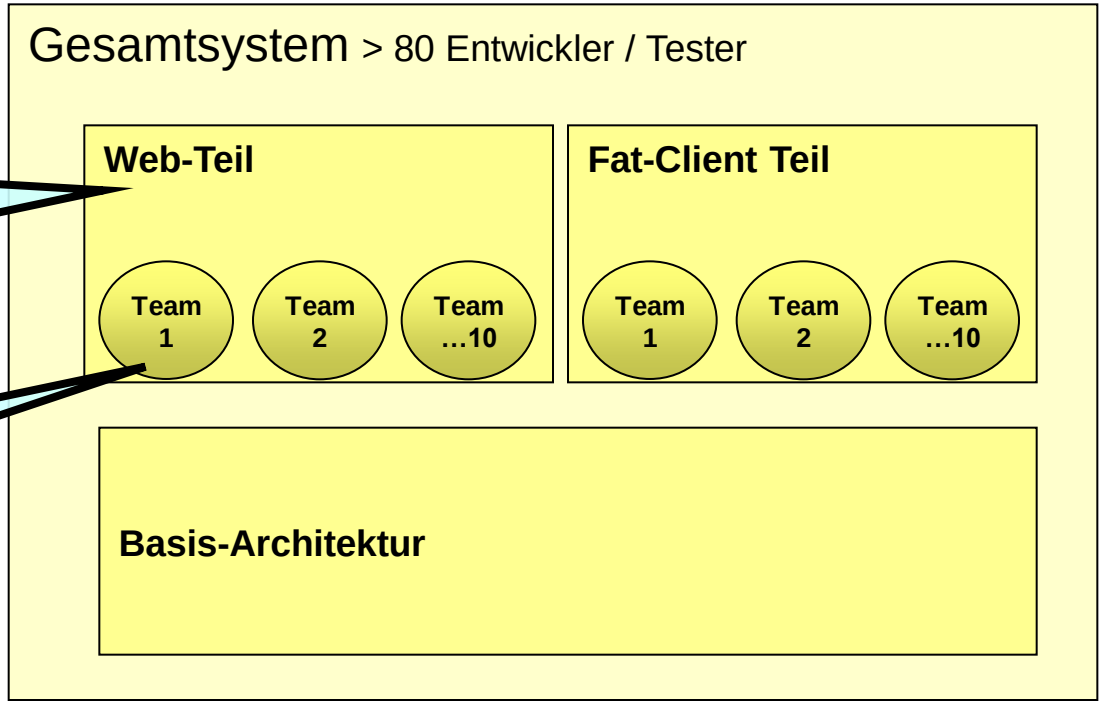
- **FindBugs**: (http://findbugs.sourceforge.net/)
  + Gute Fehlererkennung, Arbeitet mit compilierten .class Dateien.
  - Findet keine Fehler, die nur im Source-Code vorkommen (z.B. undokumentierten Code, Source-Code Annotations)
- **PMD (http://pmd.sourceforge.net)**
  + Regeln für spezielle Umgebungen z.B. Android oder JEE
  - Im Vergleich zu FindBugs findet PMD weniger schwere Fehler
- **Checkstyle (http://checkstyle.sourceforge.net/)**
  + Schnelle Analyse, gut für Code-Conventions
  - Erkennt wenige Fehler
- **CodePro AnalytiX (https://developers.google.com/java-dev-tools/codepro/doc/)**
  + Gute Eclipse Integration
  - Batch Betrieb wird nicht unterstützt
- **Sonar (http://www.sonarsource.org/)**
  + kombiniert PMD, CheckStyle und FindBugs, Historisiert die Daten
  - für Großprojekte ist die OpenSource Variante wenig geeignet (z.B. keine Zusammenfassung von Teilprojekten)

Statische Code-Analyse in einem Großprojekt

Continuous Bugfixing: Das Umfeld

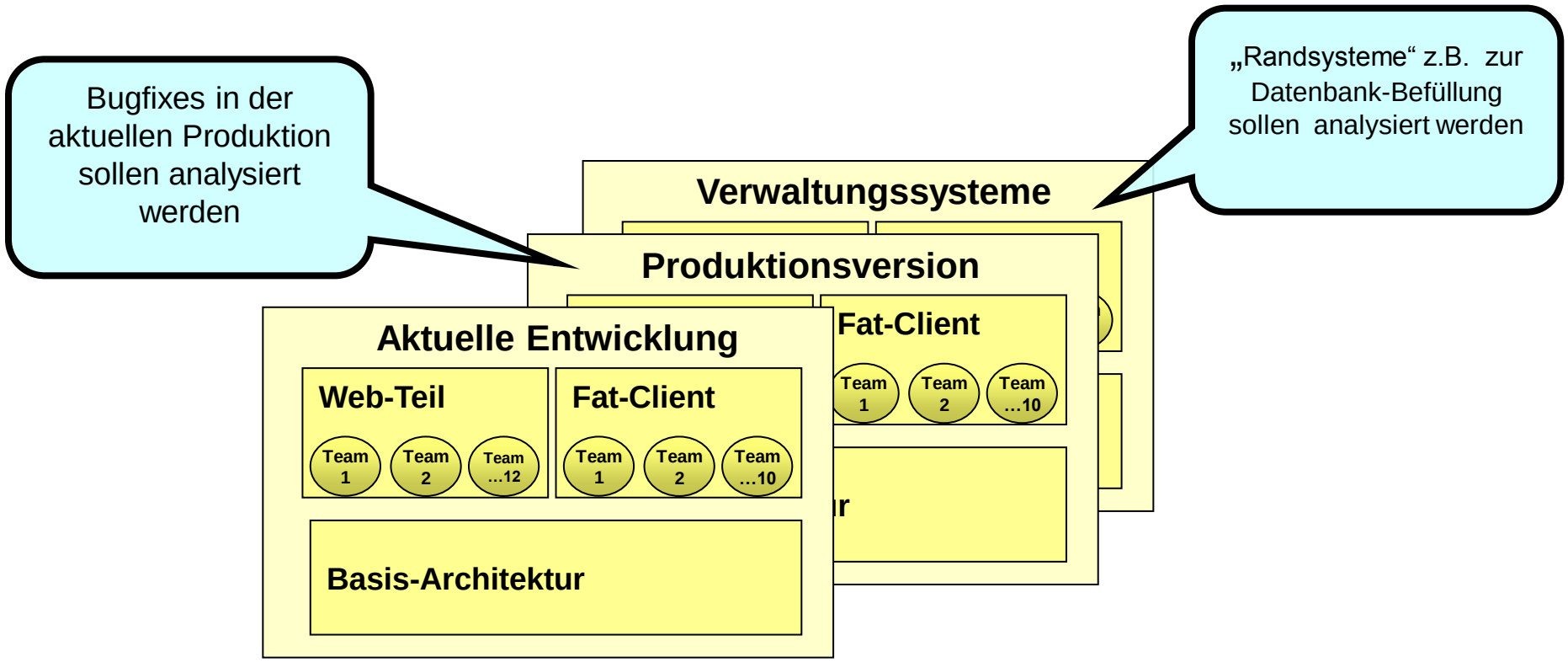
Das Gesamtsystem besteht aus einem ‚Browser- basierten“ und einem ‚Fat Client‘ Teil

Jedes Subsystem wird > 10 Teams entwickelt



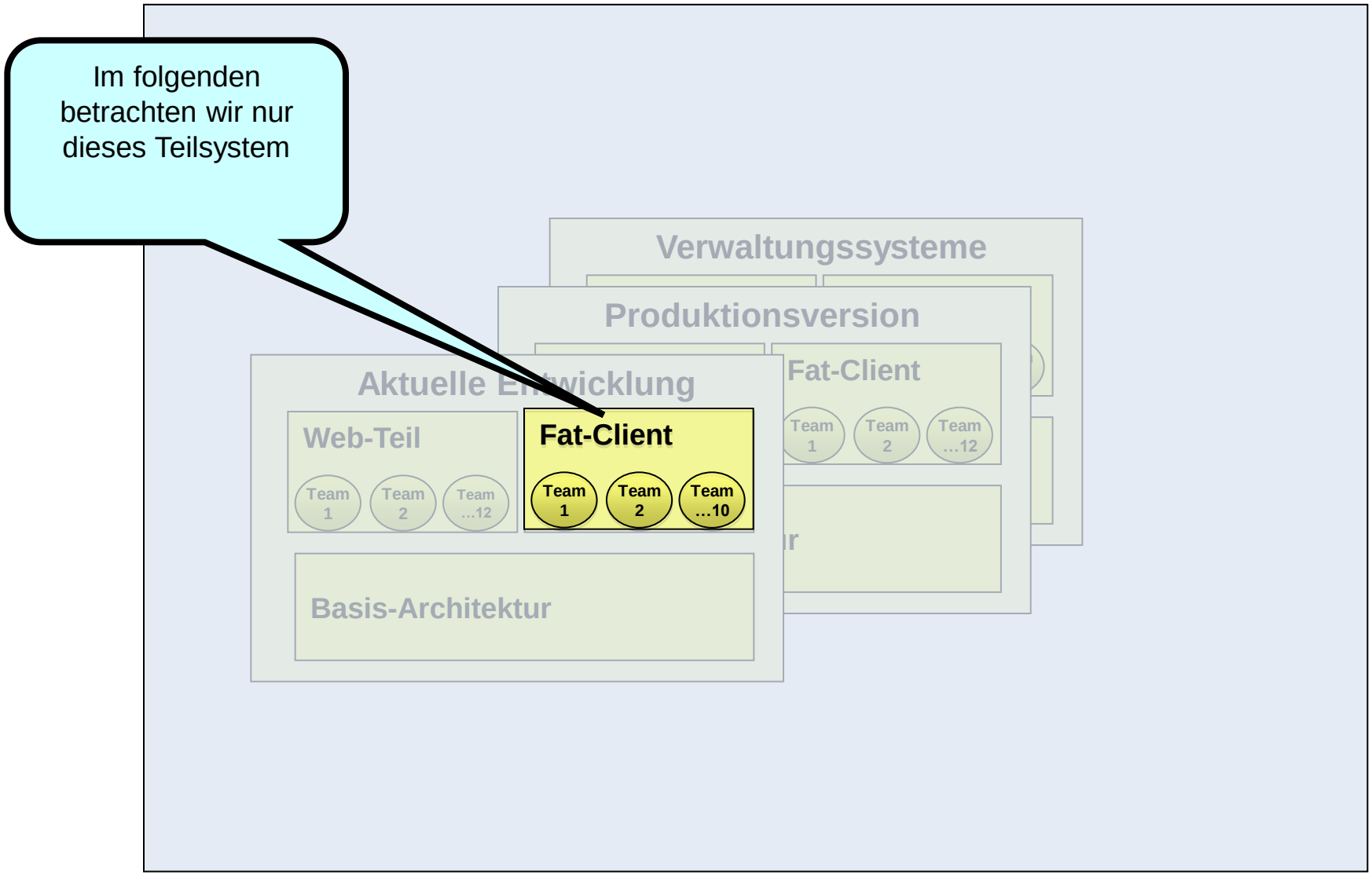
Continuous Bugfixing:

mindestens 2 parallele Zweige

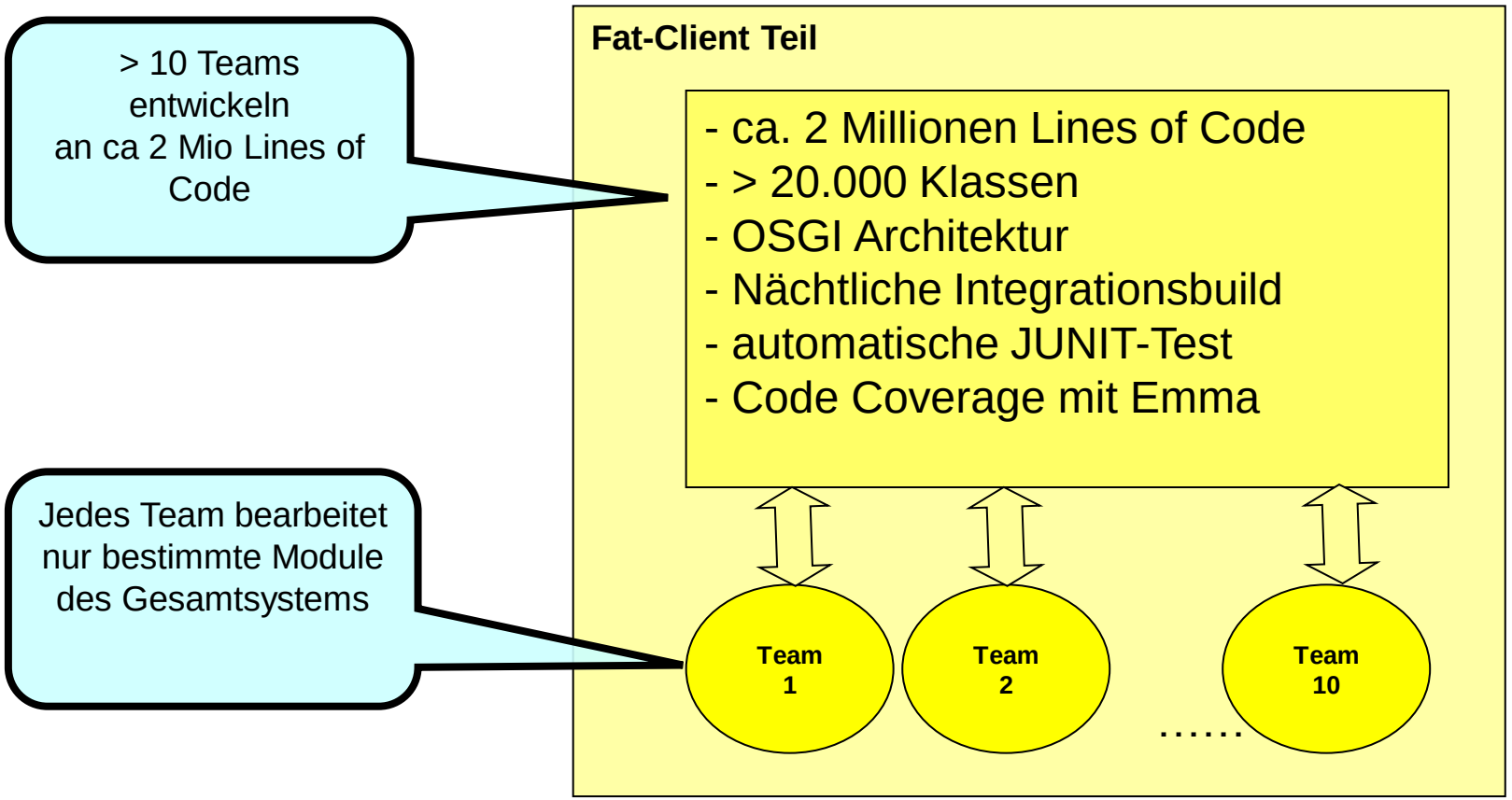


Continuous Bugfixing:

mindestens 2 parallele Zweige



Struktur eines „Teilsystems“



Effiziente statische Code-Analyse in diesem Umfeld?

„Naiver Ansatz“

- **Führende Tools wie PMD, FindBugs, CheckStyle „out of the box“**
- **Einbindung in der Tools die Eclipse Entwicklungsumgebung**
- **Einbindung in das Continous Integration Tool „HUDSON“**
- **Jeder Entwickler ist für „seinen Code“ verantwortlich**

Probleme beim „naiven Ansatz“

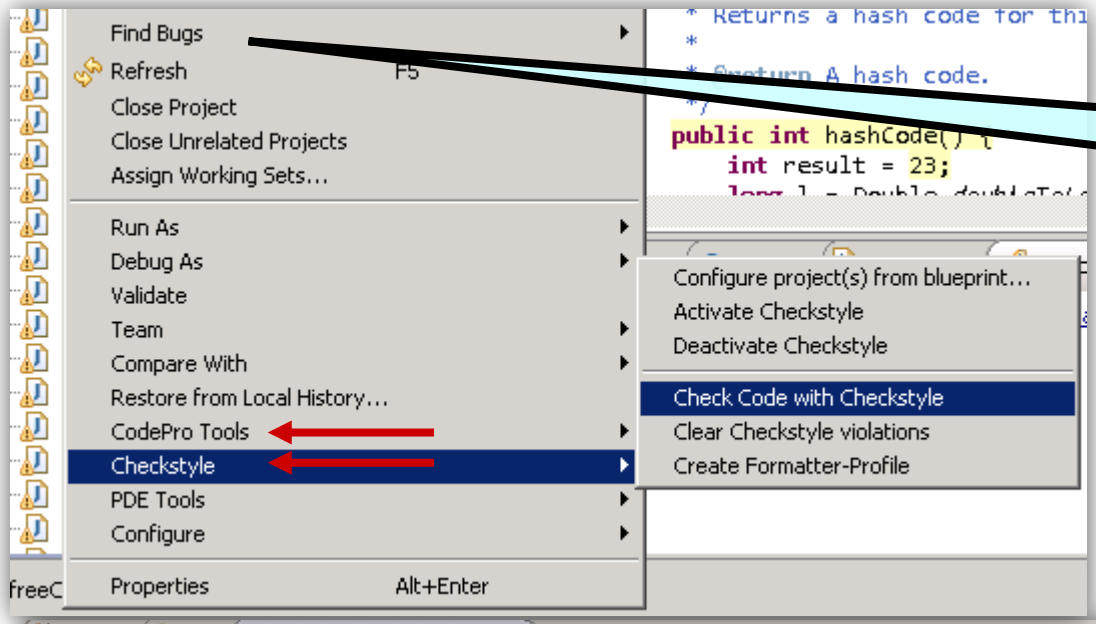
- **Der Entwickler muss die Reports von 3 Tools studieren**
- **Werkzeuge und Reporterzeugung müssen von Hand angestossen werden.**
- **Die Reports werden sehr umfangreich.**
  Die Entwickler müssen „ihren Code“ suchen
- **Keine Gesamtsicht über alle Fehler**
- **Entwickler „vergessen“ diese Reports konsequent zu verfolgen**
- **Keine Historisierung: „Waren wir letzte Woche besser als heute?“**

→ Reality-Check:
Ist dieser Ansatz für diese Projektgröße geeignet?

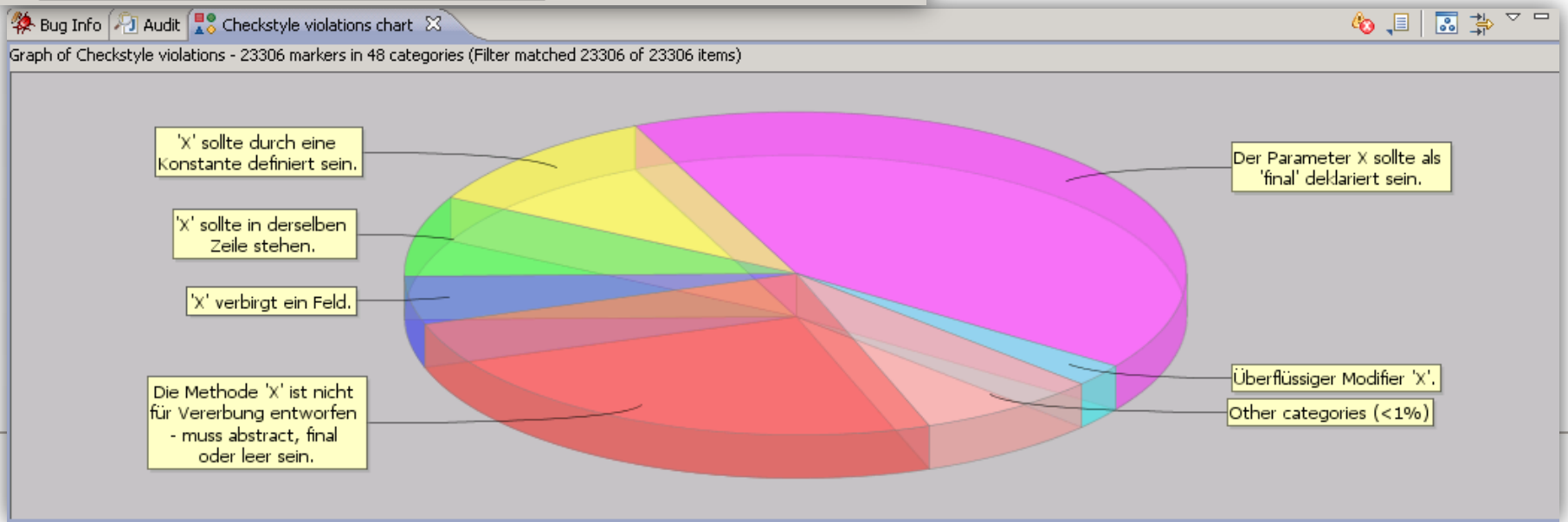
Naiver Ansatz: Probleme

Herbstcampus

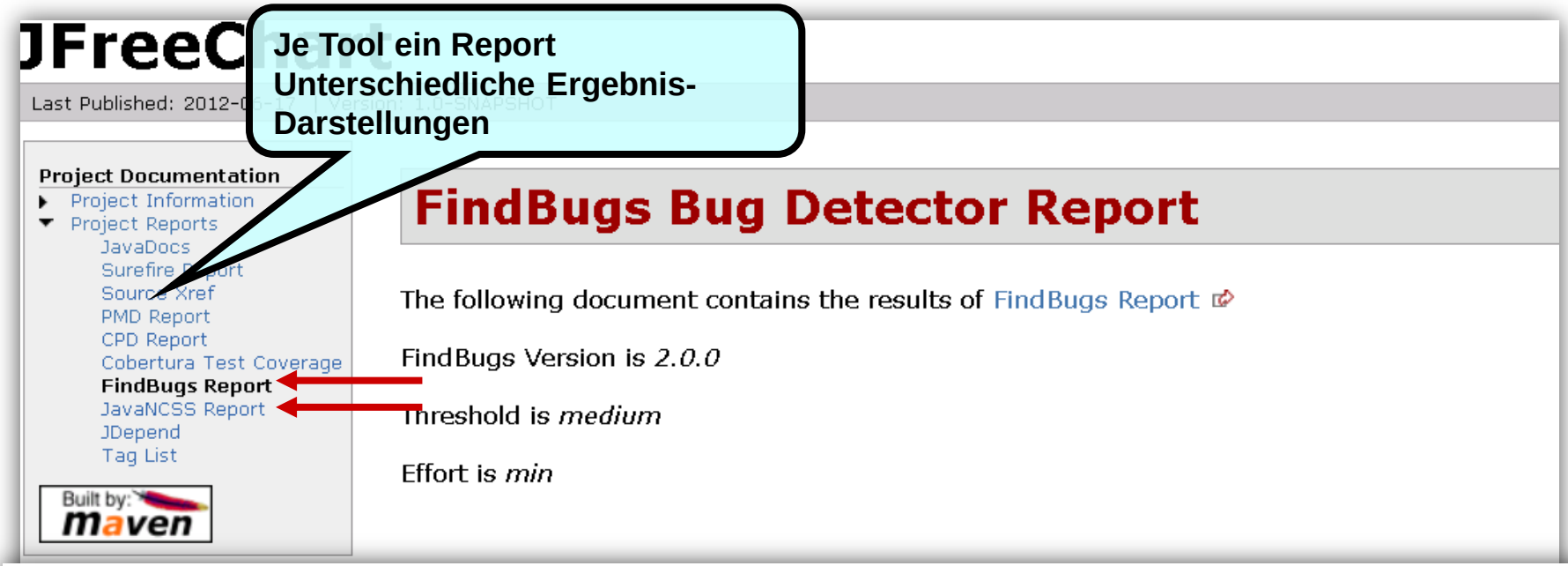
3 Plugins in der Entwicklungsumgebung



Je Tool einen Menüpunkt
Unterschiedliche Ergebnis-Darstellungen



Nach Tools getrennte HTML Reports, unterschiedlicher Aufbau



org.jfree.chart.axis.CyclicNumberAxis

Bug	Category	Details	Line	Priority
Test for floating point equality in org.jfree.chart.axis.CyclicNumberAxis.equals(Object)	STYLE	FE_FLOATING_POINT_EQUALITY	1185	High
Test for floating point equality in org.jfree.chart.axis.CyclicNumberAxis.refreshTicksHorizontal(Graphics2D, Rectangle2D, RectangleEdge)	STYLE	FE_FLOATING_POINT_EQUALITY	485	High
Test for floating point equality in org.jfree.chart.axis.CyclicNumberAxis.refreshVerticalTicks(Graphics2D, Rectangle2D, RectangleEdge)	STYLE	FE_FLOATING_POINT_EQUALITY	634	High
org.jfree.chart.axis.CyclicNumberAxis defines equals but not hashCode	BAD_PRACTICE	HE_EQUALS_NO_HASHCODE	1175-1205	Medium
org.jfree.chart.axis.CyclicNumberAxis.DEFAULT_ADVANCE_LINE_STROKE isn't final but should be	MALICIOUS_CODE	MS_SHOULD_BE_FINAL	131	High

Effiziente statische Code-Analyse in diesem Umfeld?

„Was wäre sinnvoll?“

- **Gesamtsicht auf die Projektqualität soll möglich**
- **Informationsüberflutung der Entwickler soll vermieden werden**
- **Die einzelnen Teams sollen nur „ihren“ Code sehen**
- **Möglichst wenig Falschmeldungen**
- **Historisierung, Integration in Eclipse, Integration in den Build**
- **Alle verfügbaren Open-Source Werkzeuge sollen zur Qualitätssicherung eingesetzt werden**

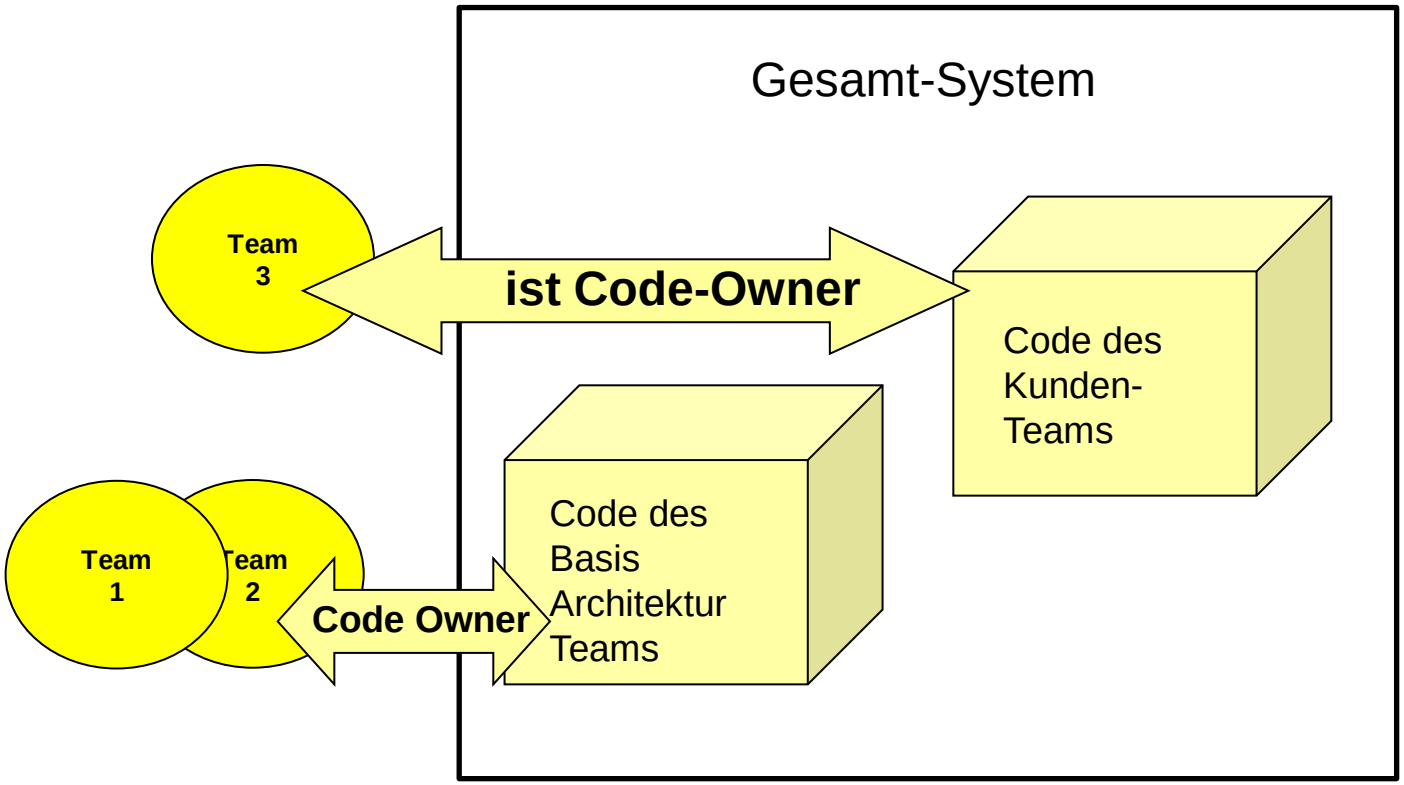
Lösungen mit health4j

Kern-Features von health4j

- **Aggregation der Ergebnisse von Tools wie PMD, FindBugs, CheckStyle…**
- **Einführung eines Tool-übergreifenden ‚rules-sets‘**
- **Umorganisation der vorhanden Code-Basis**
  (z.B. Organisation der Java Sourcen, Zeichensatz-Konvertierung der Sourcen)
- **Intelligente Reduktion von Falschmeldungen**
- **Aufteilung des Codes nach einem Code-Owner -Konzept**
  (Jeder Entwickler erhält den Report mit den Fehlern „seines“ Teams..)
- **Gesamtsicht aller Fehler für den Qualitätsverantwortlichen in anonymisierter Form**
- **Umsetzung eines Qualitätskreislaufs zur laufenden Verbesserung des Codes**

Code-Ownership

- **Der Code wird unterteilt und Teams (nicht einzelnen Entwicklern) zugeordnet**
- **Eine Jar-Modul muss z.B. nicht einem einzigen Code-Owner zugeordnet werden**
- **Für jeden Code-Owner wird die statische Code-Analyse durchgeführt und die Qualität bestimmt**



Code-Ownership: Umsetzung

- **Der Java-Code wird einem „Code-Owner“ zugeordnet**
- **Der Code wird auch fachlich zugeordnet (z.B. Database-Code)**
- **Code-Ownership wird durch eine ‚@Owner-Annotation‘ im Code erreicht**
- **Code-Ownership wird durch Angaben in der manifest.mf erreicht**

Beispiel: Code Owner über eine Annotation

```
@Owner(„ scope=databasecode, team=databaseteam“)
    class DataBaseIfc { }
```

Beispiel: in der Datei ‚manifest.mf‘

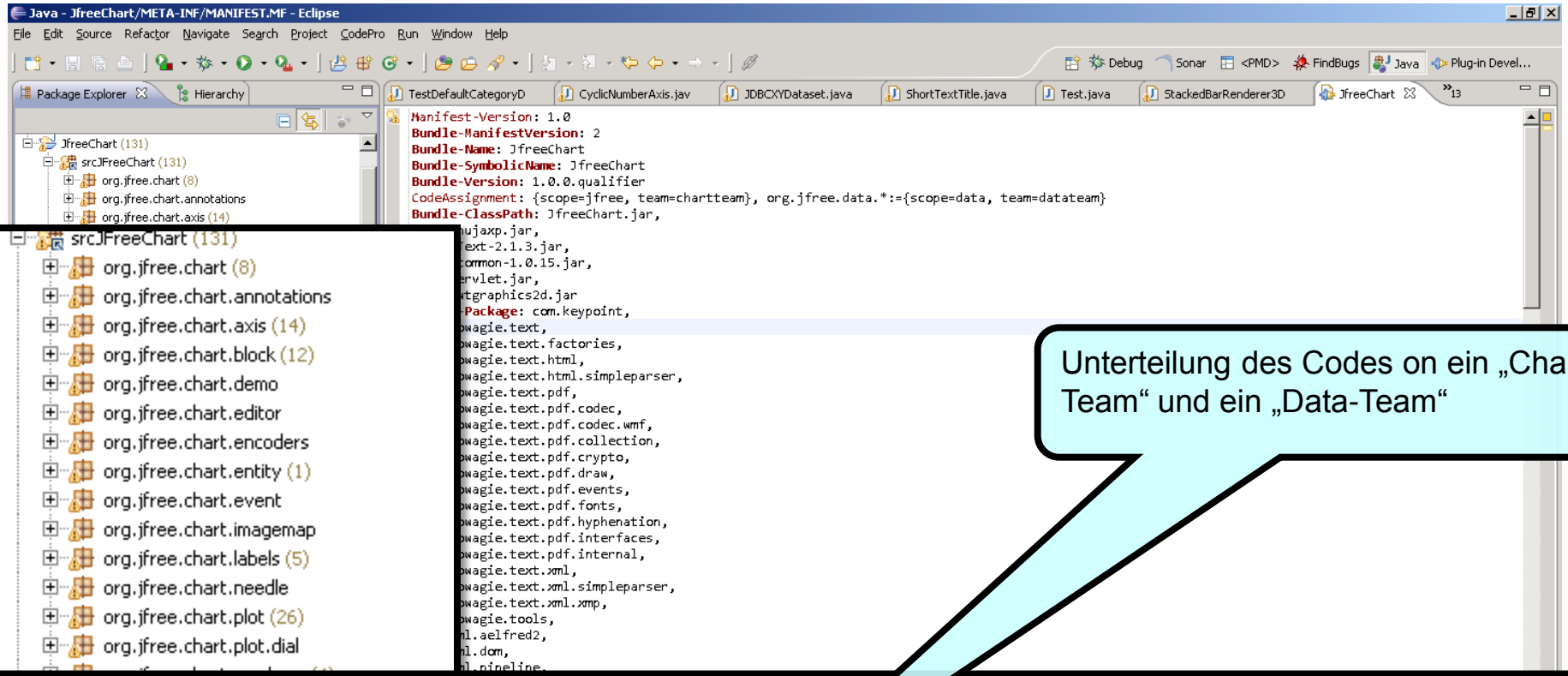
```
CodeAssignment: {scope=databasecode, team=databaseteam}
```

Hinter diesem Team-Namen verbergen sich konkrete Abteilungen und dann erst die Entwickler. Umstrukturierungen in der Organisation vorhersehen!

→ Code Ownership ist die Basis, um aus einem „Riesen-Report“ viele kleine, individualisierte Auswertungen zu erstellen, die genau auf die Teams zugeschnitten sind

Code-Ownership

Herbstcampus

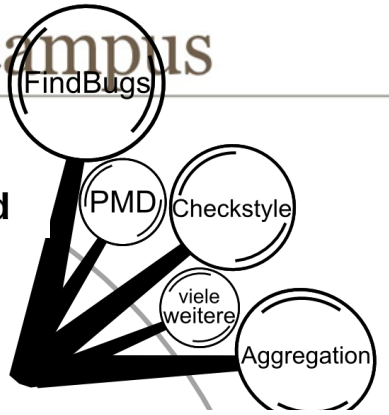



```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: JfreeChart
Bundle-SymbolicName: JfreeChart
Bundle-Version: 1.0.0.qualifier
CodeAssignment: {scope=jfree, team=chartteam}, org.jfree.data.*:={scope=data, team=datateam}
Bundle-ClassPath: JfreeChart.jar,
 lib/gnujaxp.jar,
 lib/iText-2.1.3.jar,
 lib/jcommon-1.0.15.jar,
 lib/servlet.jar,
 lib/swtgraphics2d.jar
```

Der health4j Qualitätskreislauf

Die nach Code-Owner unterteilten Bereiche durchlaufen einen **„Qualitäts-Kreislauf“**

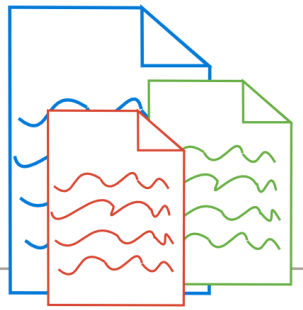
Build



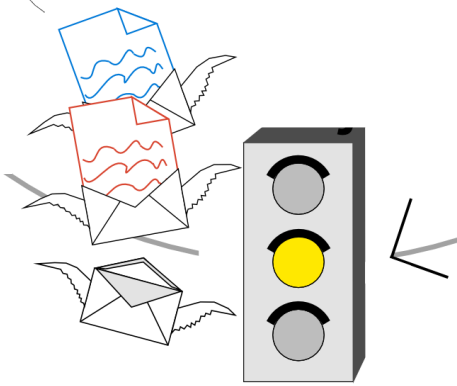
Sortierung nach CodeOwner

.class .jsp .jar .war
.class .java .class
.class .java .java
.class .class .java
.jar .properties .jsp
.war .properties .jsp
.xml .java .jar .war
.xml .java .class
.xml .jar

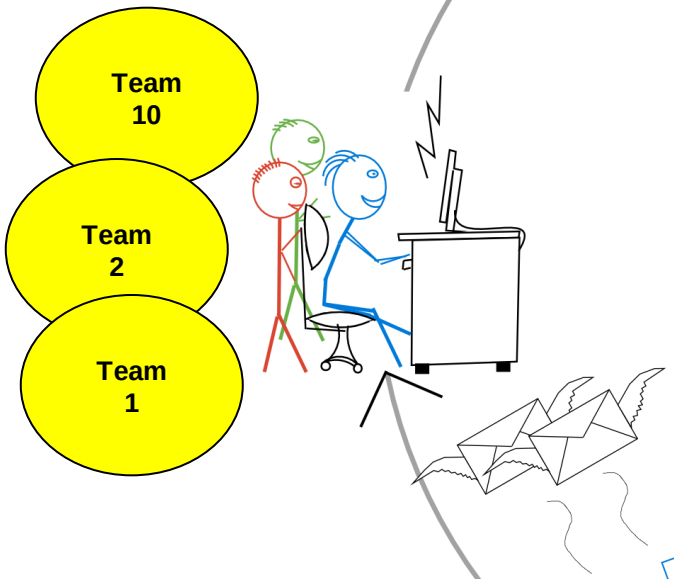
Personalisierte Auswertungen



Qualitätsaussagen je Team



Personalisierte Mails

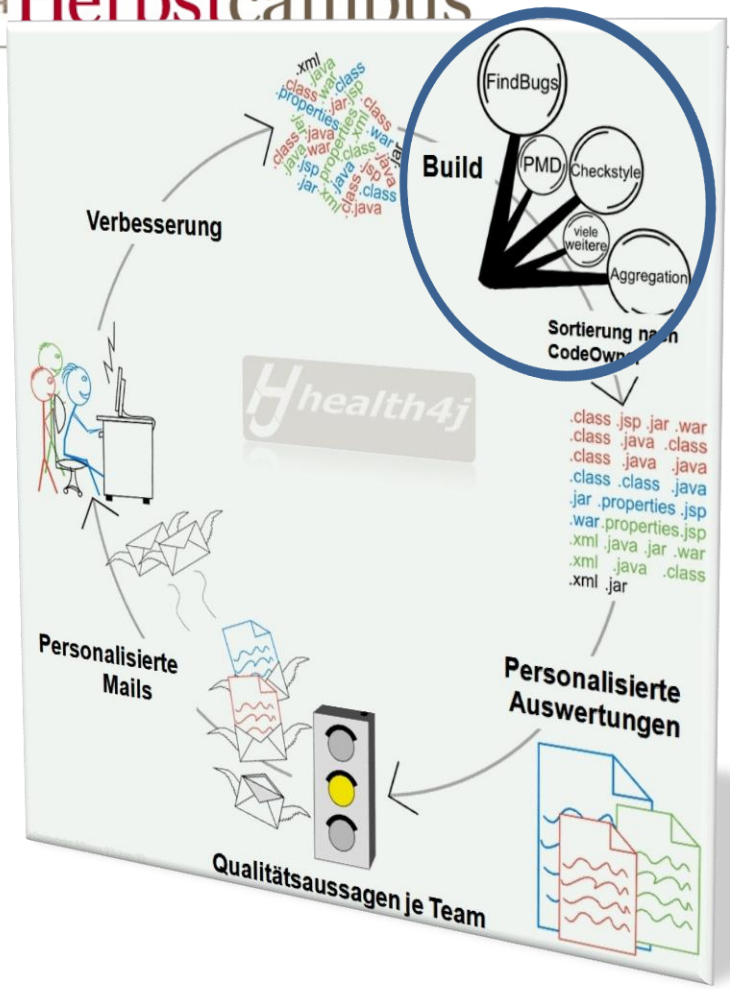


Verbesserung

1. Aufruf der OpenSource Tools

Nach dem Nightly Build erfolgt der Aufruf von

- **FindBugs**
- **PMD**
- **Code-Duplicate Detector (CPD)**
- **CheckStyle**
- **JavaNCSS**
- **Tattletale**
- **JXR (Cross-Reference)**
- **JavaDoc**
- **ApiViz (UML Generierung)**

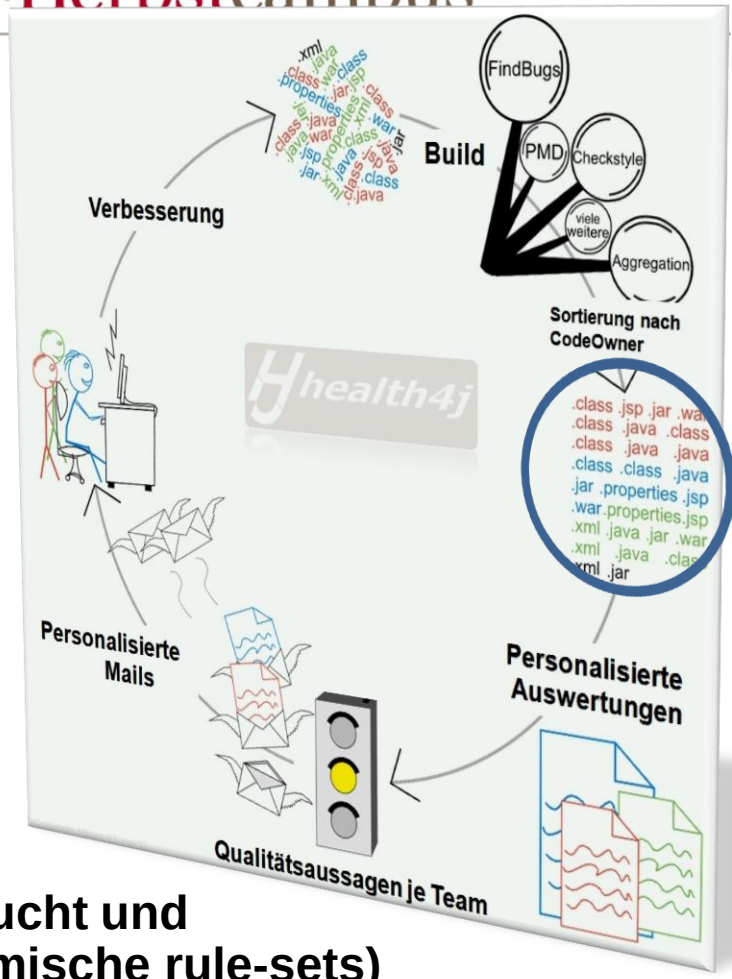


Herausforderungen:

- **Umorganisation des Source-Codes (→ Abh. Libraries finden, Paketstruktur)**
- **Laufzeit der Werkzeuge (→ Ausführung auf mehrere CPU‘s)**
- **Zeichensätze der Java– Sourcen (→manche Tools stoppen bei Umlauten)**
- **Redundante Ergebnisse**
- **Viele Falschmeldungen**

2. Veredelung der Daten

- **Bearbeiten von Falschmeldungen**
- **Redundante Daten entfernen**
- **Kombination der Daten der Einzel-Tools**
- **Anwendung eines Tool-übergreifenden rule-sets**
- **Tool - übergreifende Kennzeichnung von Falschmeldungen**
- **Ausfiltern von generiertem Code**
- **Aufbereitung von JUNIT Reports**



Beispiele

- **Das Code-Umfeld jedes Fehlers wird untersucht und Falschmeldungen werden aussortiert (dynamische rule-sets)**
- **Die Falschmeldungen werden vom Entwickler mit //NOBUG gekennzeichnet (kein //NOPMD oder //CHECKSTYLE:OFF mehr)**
- **Undokumentierter Code muss min. 100 LinesOfCode beinhalten bevor er als Fehler gemeldet wird**

Reduktion von Falschmeldungen

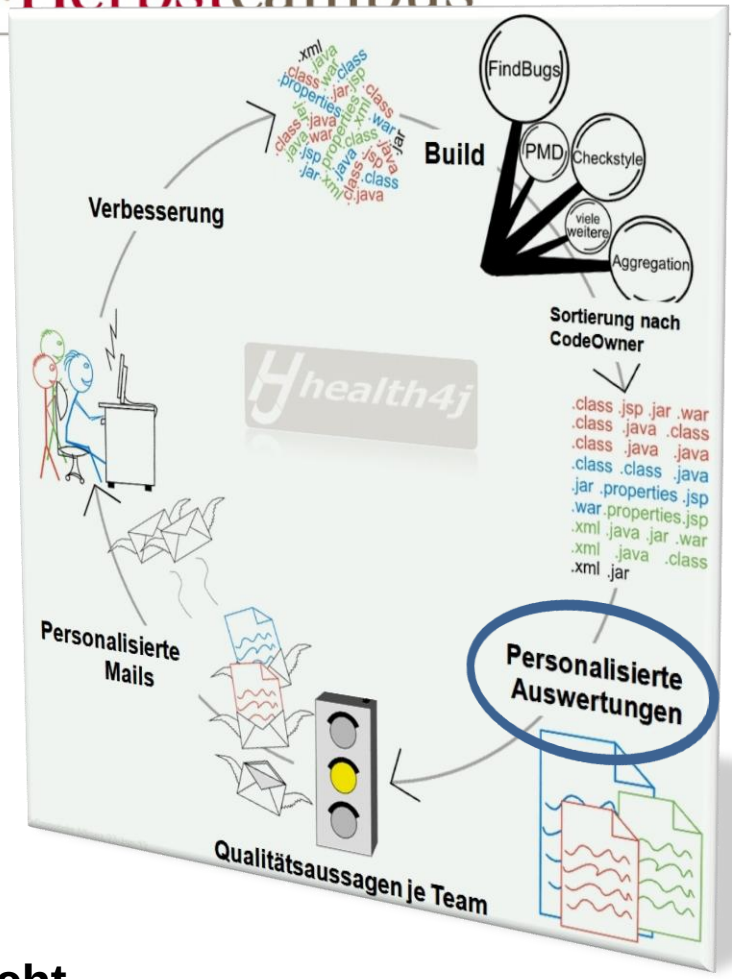
Herbstcampus

main Gesellschaft für Informationsverarbeitung mbH

health4j

3. Personalisierung

- **Für jeden Code-Owner wird ein eigener Report erzeugt**
- **Jeder Code-Owner Report verwendet die gleichen rule-sets**
- **Veränderung und Historisierung nur für diesen Owner**
- **Extraktion der UNIT-Test Daten für diesen Owner**

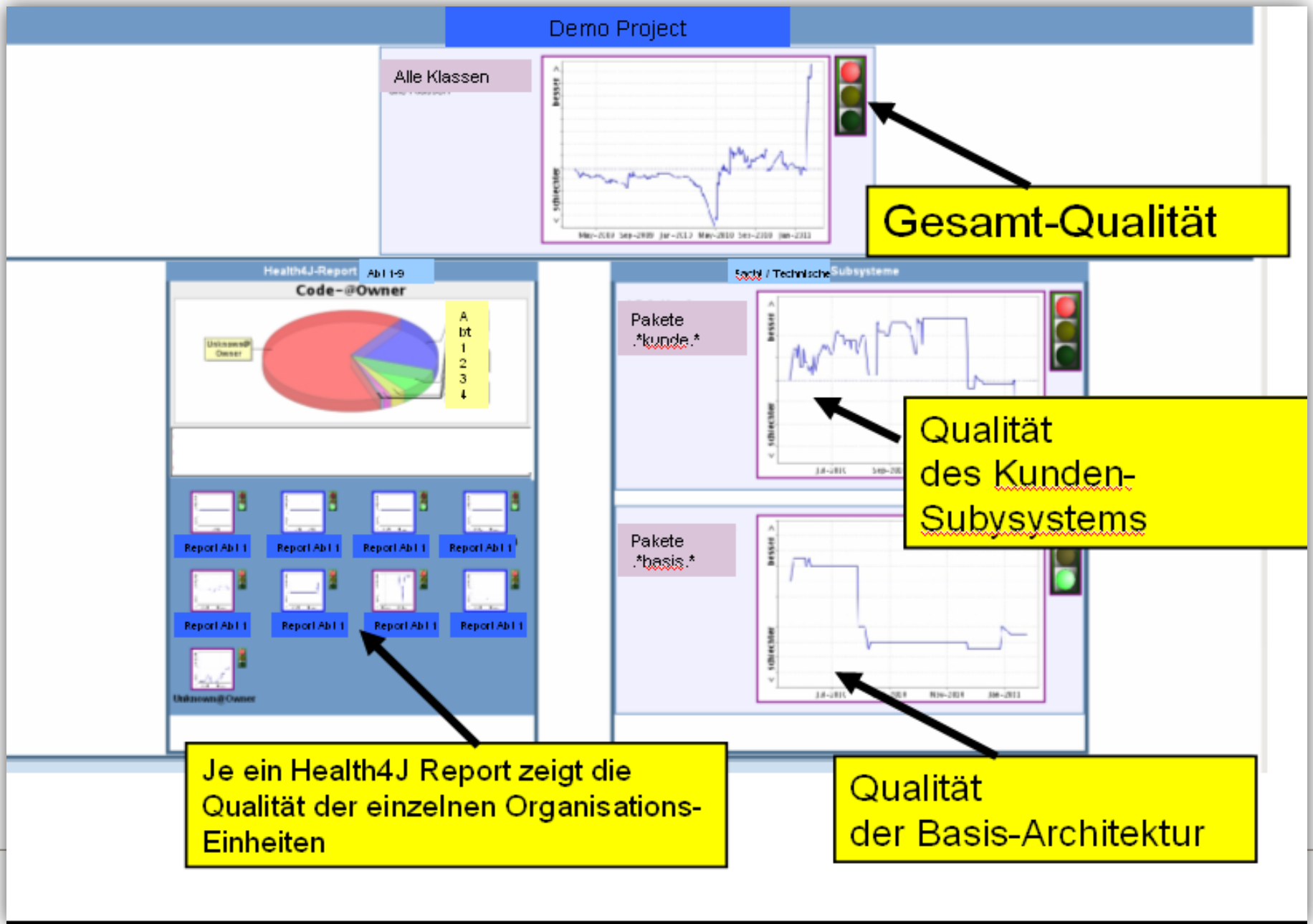


Beispiele

- **Team A bearbeitet alle Klassen der Datenbank-Schicht**
- **Der Report von Team A beinhaltet**
  - alle Fehler der Datenbank-Schicht
  - den Qualitätsverlauf der Datenbankschicht
  - JUNIT – Testergebnisse der Datenbankschicht
  - JavaDoc der Datenbankschicht

→ Alle Auswertungen sind auf das Team A zugeschnitten.
Das Team A „findet sich in dem Report wieder“.

Ergebnis der Personalisierung: Eine Code-Basis, jedem Team „seinen“ Report



4. Qualitäts-Aussage

- **Über eine einfache „Ampel- Metaper“ wird die Qualität des Codes signalisiert**
- **Einfaches Bewertungssystem:**

- Blocker

(offensichtliche Fehler, geringe Wahrscheinlichkeit einer Falschmeldung, keine Auslieferung)

- Stil-Probleme

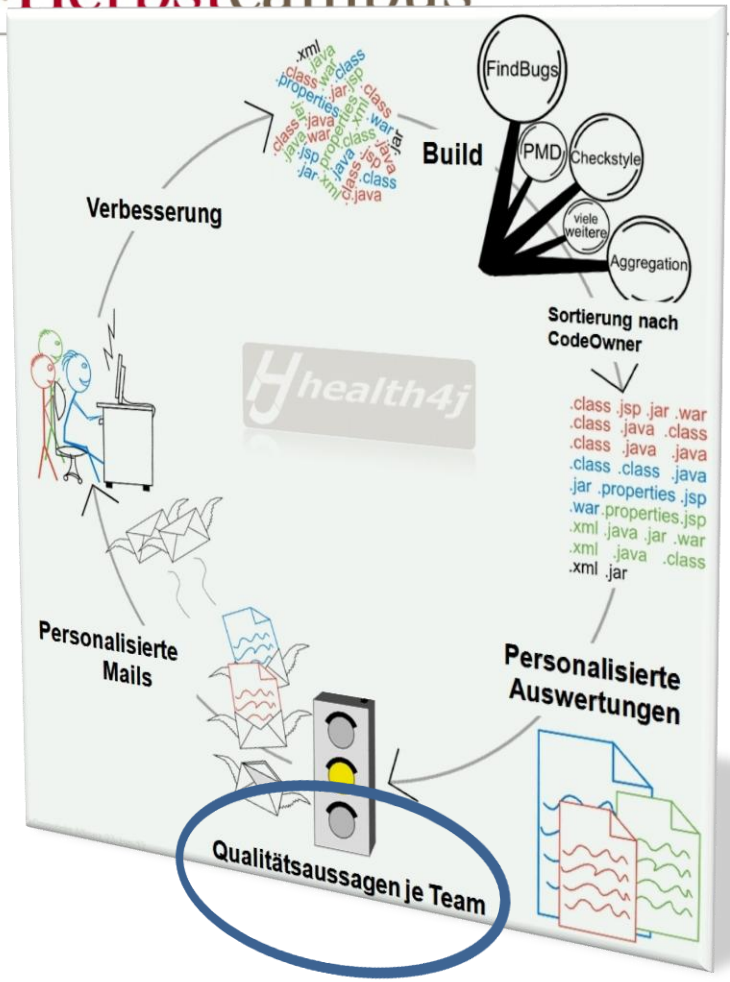
(nicht schön…könnte auch Falschmeldung sein)

- Offene Punkte

(TODO‘s…sollte man nicht vergessen…)

- Undokumentierter Code

(komplexe Codestellen müssen dokumentiert werden)



Beispiele

- **Ein offensichtlicher „Nullpointer“ im Code führt sofort zu einer „roten Ampel“. Das Modul soll nicht ausgeliefert werden. Die Menge an Code in dem Modul ist dabei unerheblich**

Qualitäts-Aussage

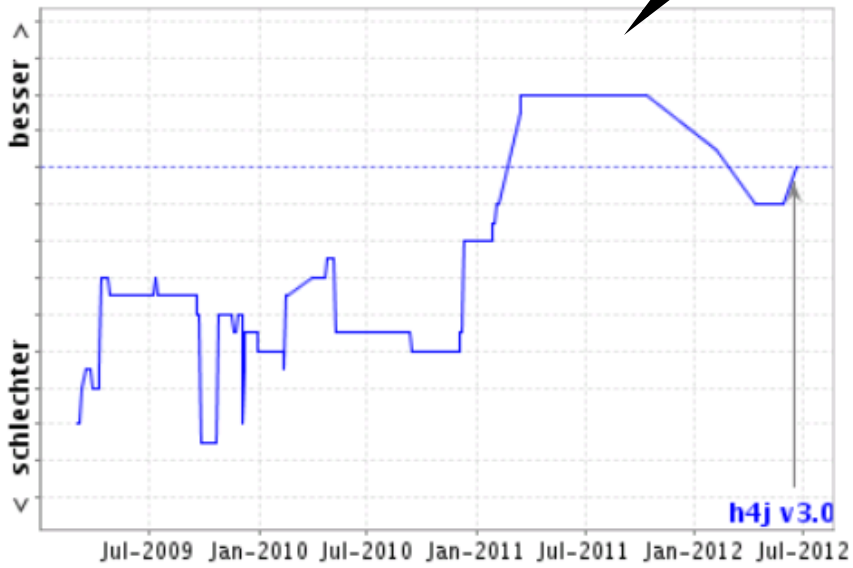
Herbstcampus

Projekt Report 'tomcat'

Aktuell: neue, gefixte, 6 ignorierte Probleme, 128218 (+230) Zeilen

Einfache Erfassung der Qualität

Gesamt-Qualität

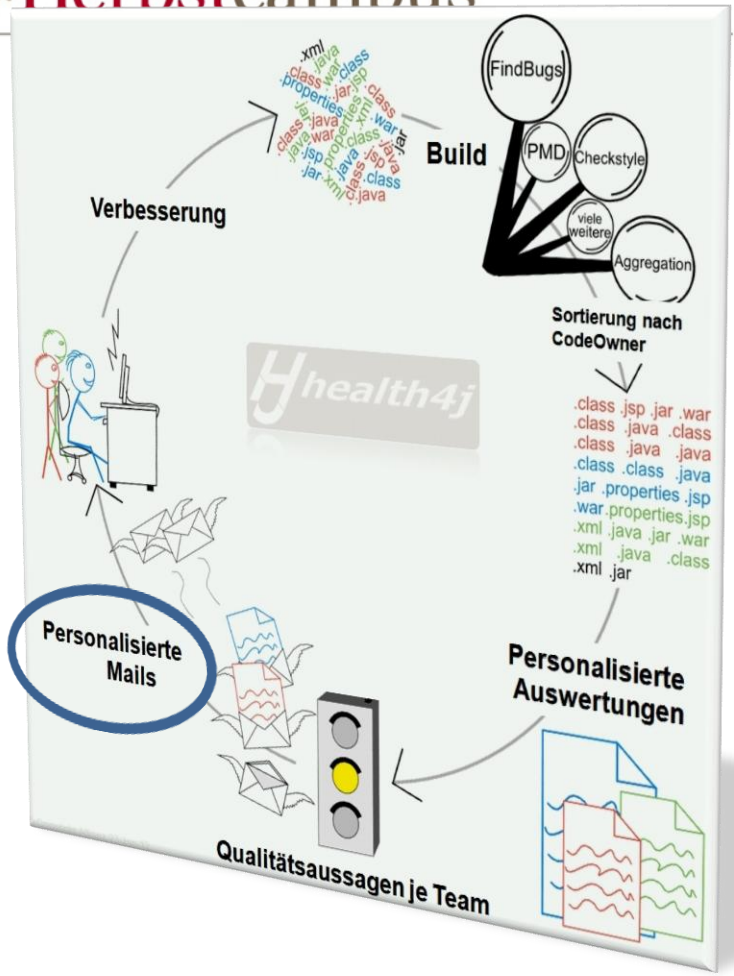


Detail-Qualität

Prio.	Qualitätsmerkmal	Status	Bewertung	Verbesserung voheriger Lauf?
1.	Offensichtliche Fehler		Es gibt Probleme, die behoben werden sollten.	
2.	Stilprobleme		Es gibt Probleme, die behoben werden sollten.	
3.	Offene Punkte		Es gibt Probleme, die behoben werden sollten.	
4.	Undokumentierter Code		Alle Werte sind in Ordnung.	

5. Informieren

- **Keine Informationsüberflutung**
- **Einfache, übersichtliche Reports keinen „Overkill“ mit Metriken**
- **„Drill Down„ bis zum Source Code mit 3 Clicks**
- **Über eine Mail an das Team werden <u>nur neue, schwere</u> Fehler gemeldet**
- **Ein Eclipse Plugin zeigt die gefunden Fehler direkt an**



Informieren per Mail und in Eclipse

Herbstcampus

```
1184        CyclicNumberAxis that = (CyclicNumberAxis) obj;
1185        if (this.period != that.period) {
1186            return false;
1187        }
```

Problems | Javadoc | Declaration | Search | Progress | History | Bookmarks | Console | Health4J Übersicht

Beschreibung	Ressource	Zeile
Stil Probleme (330)		
Offene Punkte (110)		
Ohne Kategorie (44)		
Offensichtliche Fehler (12)		
Berechnung mit Floating-Point-Werten kann durch Rundung zu Ungenauigkeiten führen, so dass Vergleiche fehlschlagen	CyclicNumberAxis.java	1185

Betreff: WG: h4j-report, project:JfreeChart --> 2 neue schwere Fehler

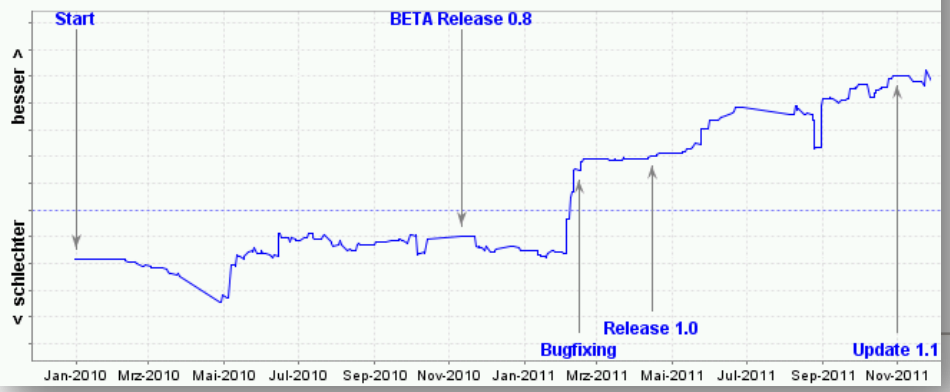
Sehr geehrte Damen und Herren,

im Health4j- Lauf vom 22.06.2012 10:47:43 im Projekt "JFreeChart“ sind

- **2 neue schwere Fehler**

aufgetaucht.

Aktueller Qualitätsverlauf von Projekt 'JFreeChart'

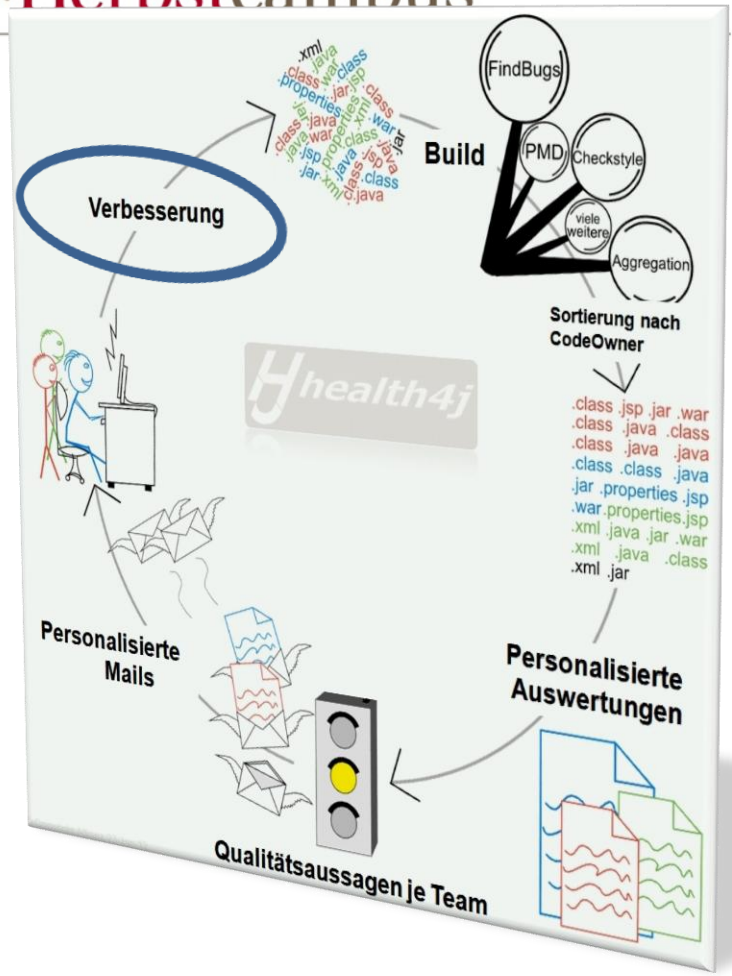


main Gesellschaft für Informationsverarbeitung mbH

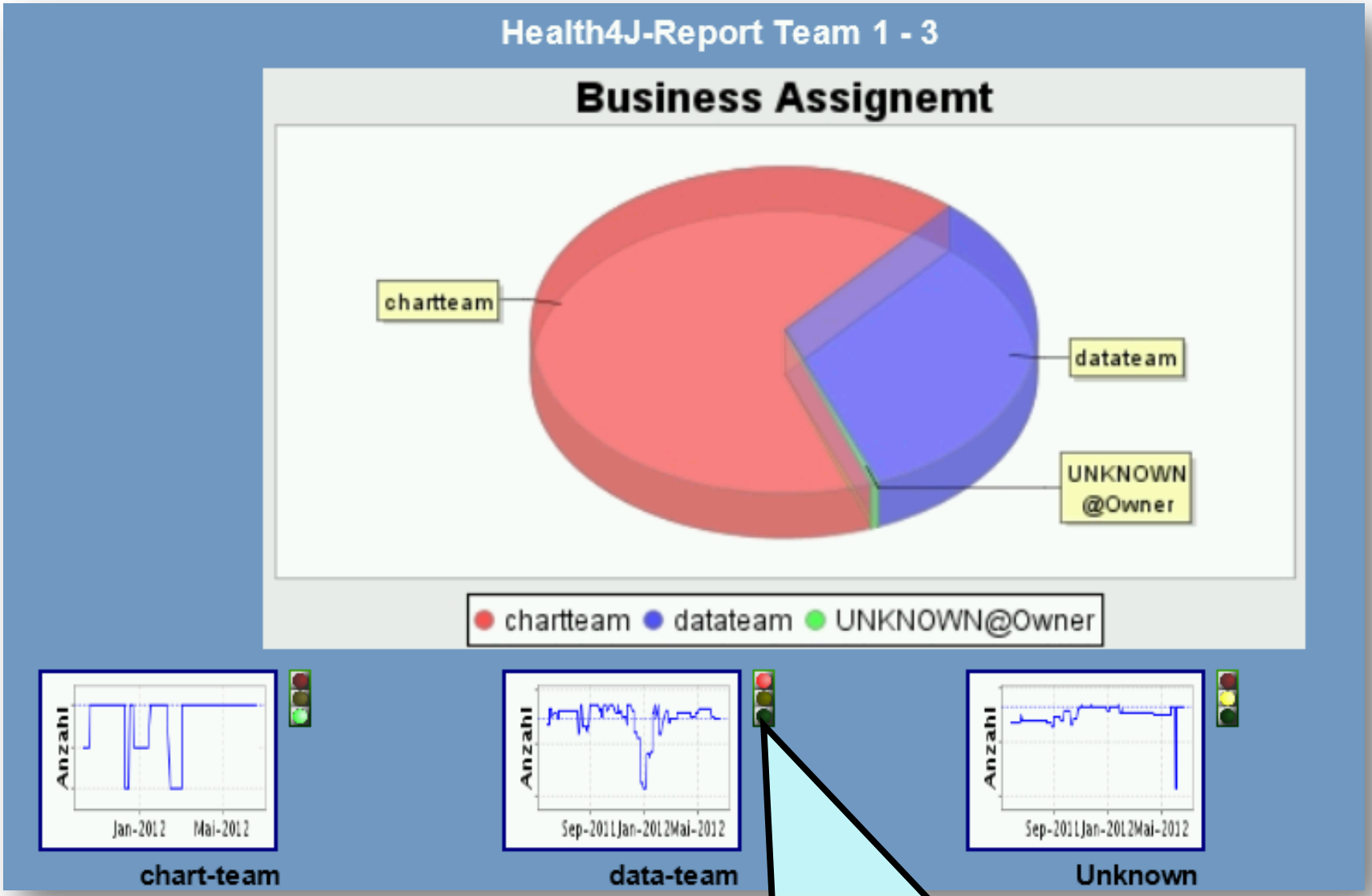
health4j

6. Verbessern

- **Teams werden nur über ihre „individuellen“ Fehler informiert**
- **„Öffentliches Dashboard“ zur Projektqualität spornt zur Qualitätsverbesserung an. (Ohne „blaming“ einzelner Entwickler)**
- **Kontrolle der Gesamtqualität durch eine „Gesamtsicht“ auf die komplette Codebasis**

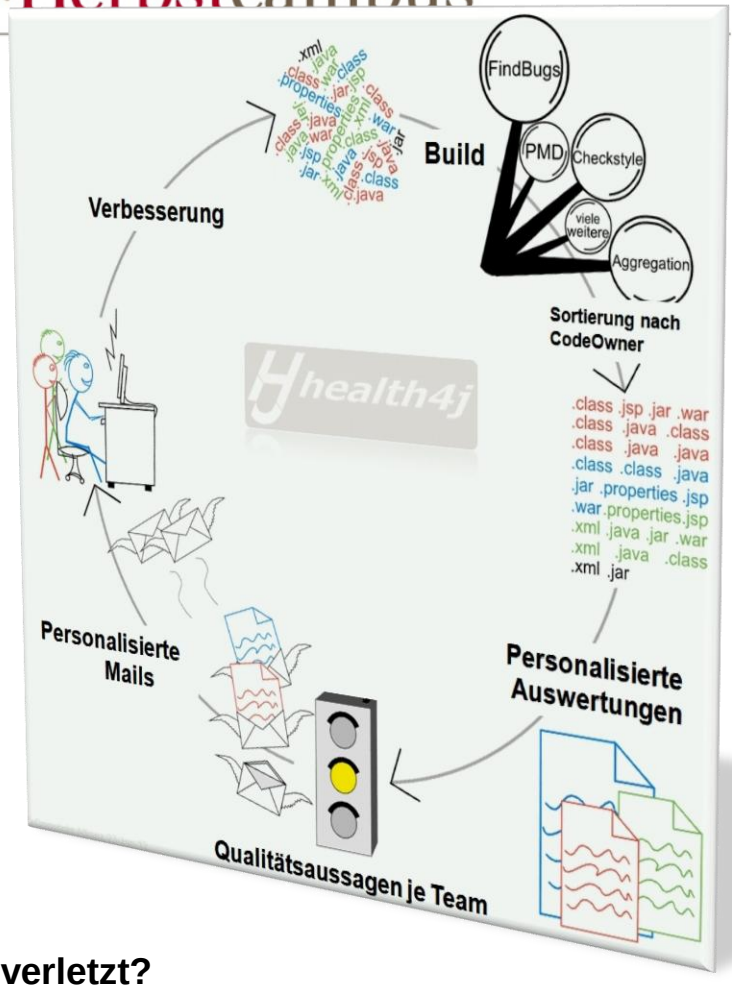


Herbstcampus



Ziele erreicht?

- **Vermeide „Informationsüberflutung“**
- **Informiere aktiv über neue Fehler**
- **Erkenne Fehler früh, Führe die Entwickler schnell zu ihren Fehlern**
- **Verwende ein einfaches, einheitliches, kleines rule-set**
- **Analysiere sowohl in der Entwicklungsumgebung als auch beim Build**
- **Historisiere die Daten, beachte auch Unterschiede**
- **Gebe Feedback**
- **Verwende mehrere OpenSource – Werkzeuge für optimale Qualität**
- **Schaffe Verständnis, Benenne einen Qualitätsverantwortlichen**



Wunschkonzert….

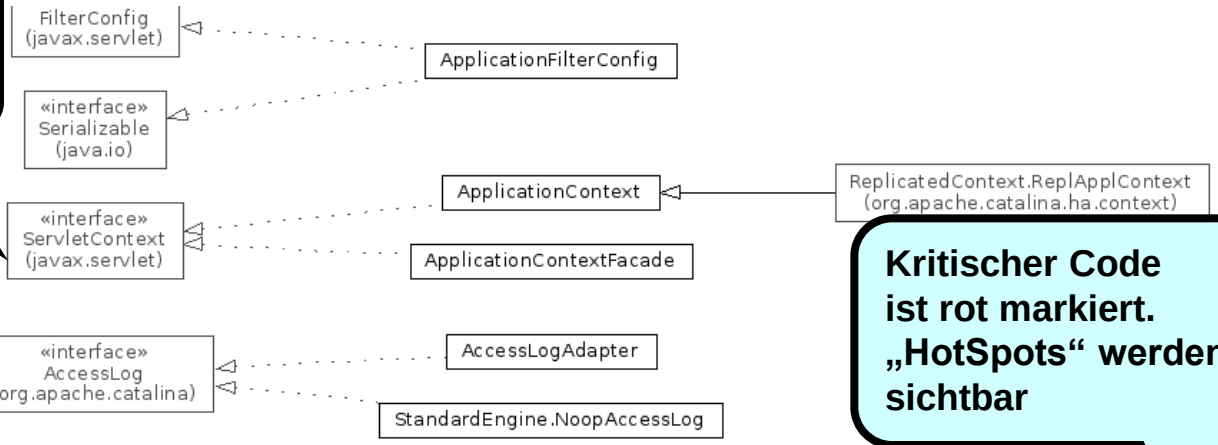
- **gemeinsame Code Reviews**
- **Anbindung an das Fehler Management-System**
- **Architektur-Management: Wo werden Architekturrichtlinien verletzt?**
- **Analyse von C++/C/C#, COBOL, PL/1, Web (z.B. xhtml)**
- **Besser Test-Unterstützung: Welcher Code ist komplex und wird nicht getestet?**
- **„Wenn ich an dieser Klasse eine Änderung durchführe, welche Testfälle müssen neu durchgeführt werden? Welche Testfälle sind betroffen?“**
- **Echt-Zeit-Analysen: Volle Analyse vor dem Checkin in das Versionsmanagment-System**

Weitere health4j - Features

Fehler-Landkarten

Herbstcampus

Aktuelle UML Diagramme



Kritischer Code ist rot markiert. „HotSpots“ werden sichtbar

javax.annotation
javax.annotation.security
javax.ejb
javax.el
javax.mail

All Classes

ASCIIReader
AbsoluteComparator
AbsoluteOrder
AbsoluteOrderingRule
AbstractAjpConnectionHandler
AbstractAjpProcessor
AbstractAjpProtocol
AbstractCatalinaCommandTask
AbstractCatalinaTask
AbstractConnectionHandler
AbstractEndpoint
AbstractGroup
AbstractHttp11JsseProtocol
AbstractHttp11Processor
AbstractHttp11Protocol
AbstractInputBuffer
AbstractObjectCreationFactory
AbstractOutputBuffer
AbstractProcessor
AbstractProtocol
AbstractReplicatedMap
AbstractRole
AbstractRxTask
AbstractSender

Class Summary	ERROR	STYLE	CCN	UNKNO	DOCU	TODO
AccessLogAdapter	0	0	0	0	0	0
AccessLogListener	0	0	0	0	0	0
AnnotationCacheEntry	0	0	0	0	0	0
AnnotationCacheEntryType	0	0	0	0	0	0
ApplicationContext	0	0	0	0	0	0
ApplicationContextFacade	0	0	0	0	0	0
ApplicationDispatcher	0	0	0	0	0	1
ApplicationFilterChain	0	0	0	0	0	1
ApplicationFilterConfig	0	0				
ApplicationFilterFactory	0	0				
ApplicationFilterRegistration	0	0				
ApplicationHttpRequest	8	6	0	0	0	0
ApplicationHttpResponse	0	1	0	1	0	0
ApplicationJspConfigDescriptor	0	0	0	0	0	0

Bugs

Bug Typ	Anzahl
EmptyIfStmt	2
BC_VACUOUS_INSTANCEOF	4

main Gesellschaft für Informationsverarbeitung mbH

health4j

Aufbereitung von JUNIT Ergebnissen

Herbstcampus

- **Besser Aufbereitung als die Original-Reports**
- **Personalisierung**
- **Reports zeigen Veränderungen an**

Von welchem Team kommt der Testfall? JUnit Reports werden personalisiert

9 Error	Error	java.lang.ClassNotFoundException A class could not loaded: org.jfree.axis.TestSuite	0.0	org.jfree.test.AllTests (chartteam)	initializationError

same succes in testcase TOP

current status	last status	runtime, s	package	testcase
1 Success	Success	0.266	org.jfree.test.AxisTest (chartteam)	testSomething
2 Success	Success	0.0	org.jfree.test.DataTest (datateam)	testData

same succes in testcase

current status	last status	runtime, s	
1 Success	Success	0.266	org.jfree.test.AxisTest (chartteam)
2 Success	Success	0.0	org.jfree.test.DataTest (datateam)

Gab es Änderungen? Beim letzten Lauf war dieser Testfall auch in Ordnung…

Herbstcampus

- **Schnelle Anzeige der Code-Änderungen im Vergleich zum letzten Lauf**

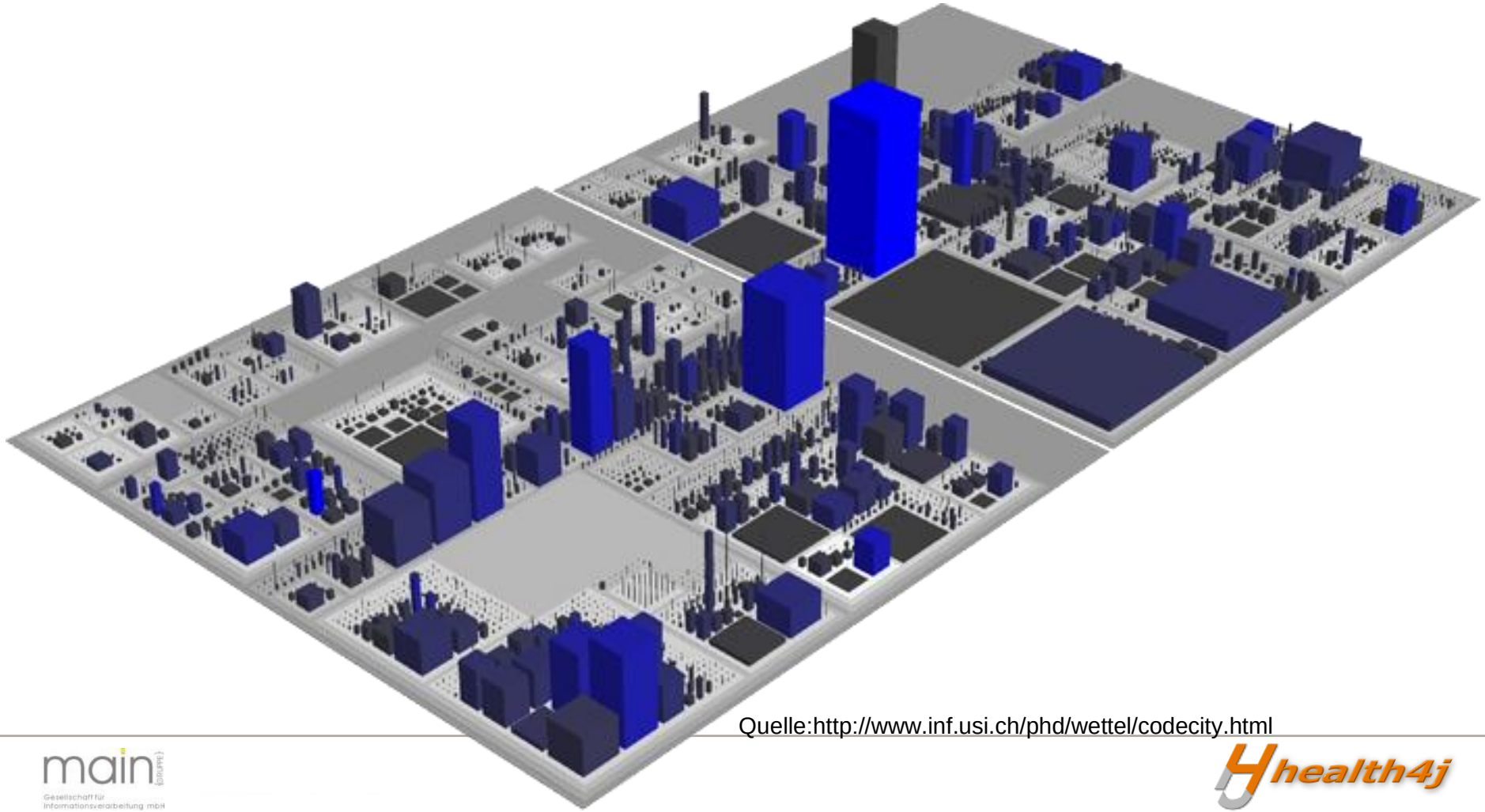
Seit dem letzten Lauf geänderter Code

```
373             wrequest.setContextPath(contextPath);
374             wrequest.setRequestURI(requestURI);
375             wrequest.setServletPath(servletPath);
376             wrequest.setPathInfo(pathInfo);
377             if (queryString != null) {
378                 wrequest.setQueryString(queryString);
379                 wrequest.setQueryParams(queryString);
380             }
381
382             processRequest(request,response,state);
383         }
384
385         // This is not a real close in order to support error
    processing
386         if (wrapper.getLogger().isDebugEnabled() )
387             wrapper.getLogger().debug(" Disabling the
    response for futher output");
388
389         if  (response instanceof ResponseFacade) {
390             ((ResponseFacade) response).finish();
```

```
373             wrequest.setContextPath(contextPath);
374             wrequest.setRequestURI(requestURI);
375             wrequest.setServletPath(servletPath);
376             wrequest.setPathInfo(pathInfo);
377             if (queryString != null) {
378                 wrequest.setQueryString(queryString);
379                 wrequest.setQueryParams(queryString);
380             }
381
382             processRequest(request,response,state);
383         }
384
385 prevtop     if (request.getAsyncContext() != null) {
386>                // An async request was started during the
    forward, don't close the
387>                // response as it may be written to during the
    async handling
388>                return;
389>            }
390>
391         // This is not a real close in order to support error
    processing
392         if (wrapper.getLogger().isDebugEnabled() )
393             wrapper.getLogger().debug(" Disabling the
    response for futher output");
394
395         if  (response instanceof ResponseFacade) {
396             ((ResponseFacade) response).finish();
```

Ausblick

- **3D Grafiken erlauben eine „neue Sicht“ auf ein System**
- **Einfache Erfassung und Vergleich von Metriken**
- **Cockpit fürs Management**



Quelle:http://www.inf.usi.ch/phd/wettel/codecity.html

Haben Sie Fragen?

Kostenlose health4j Version (bis 300.000 LOC) unter
http://www.health4j.de

Demo-Reports von Opensource-Projekten unter
http://www.japhara.de/

Wir unterstützen Sie beim Setup Ihrer Projekte
per Mail unter info@main-gruppe.de

Vielen Dank!

Holger Thom
main {GRUPPE}