Merciless Refactoring with Eclipse

Martin Lippert, Matthias Lübken
it-agile GmbH
{martin.lippert, matthias.luebken}@it-agile.de
http://www.it-agile.de/

XP2006, Oulu

Part 2: Large Refactorings

- Part 1: Daily Refactoring
  - Quick fixes
  - Local refactorings
  - Small refactorings
  - Hands-on demonstrations
- Part 2: Large Refactorings
  - Large refactorings
  - Dependency management
  - Tools to detect and control refactorings
  - Some Demos

General principles

- Refactoring is done in micro-steps
- These steps can be expressed as mechanics
  - see mechanics in [Fowler 99]
- System is executable after each micro-step!!!
- Continuous integration.

- There are exceptions (like Rename Class without automation)
- So there are
  - **Safe Refactorings** – during the mechanics no compile errors can occur
  - **Unsafe Refactorings** – during the mechanics the system can break

Refactoring in praxis

- Ideal way:
  - Before a new feature is implemented, the structure is checked if it is suitable for the feature. If not: refactoring.
  - Implementing feature. During implementation maybe further refactorings.
  - After the refactorings the structure is checked if it is still good. If not: refactoring.
- Observation: refactorings are done to seldom
- Why?
  - „I don't care…" ?
  - Lack of discipline?
  - Suspended refactorings are getting bigger!
  - No testcases, to verify refactorings?

Consequences

- The structure of the system degenerates and it's getting harder to implement refactorings.
- The necessary refactorings are getting bigger and therefore more risky

➔ **Refactoring is an elementary part of software development!!!**

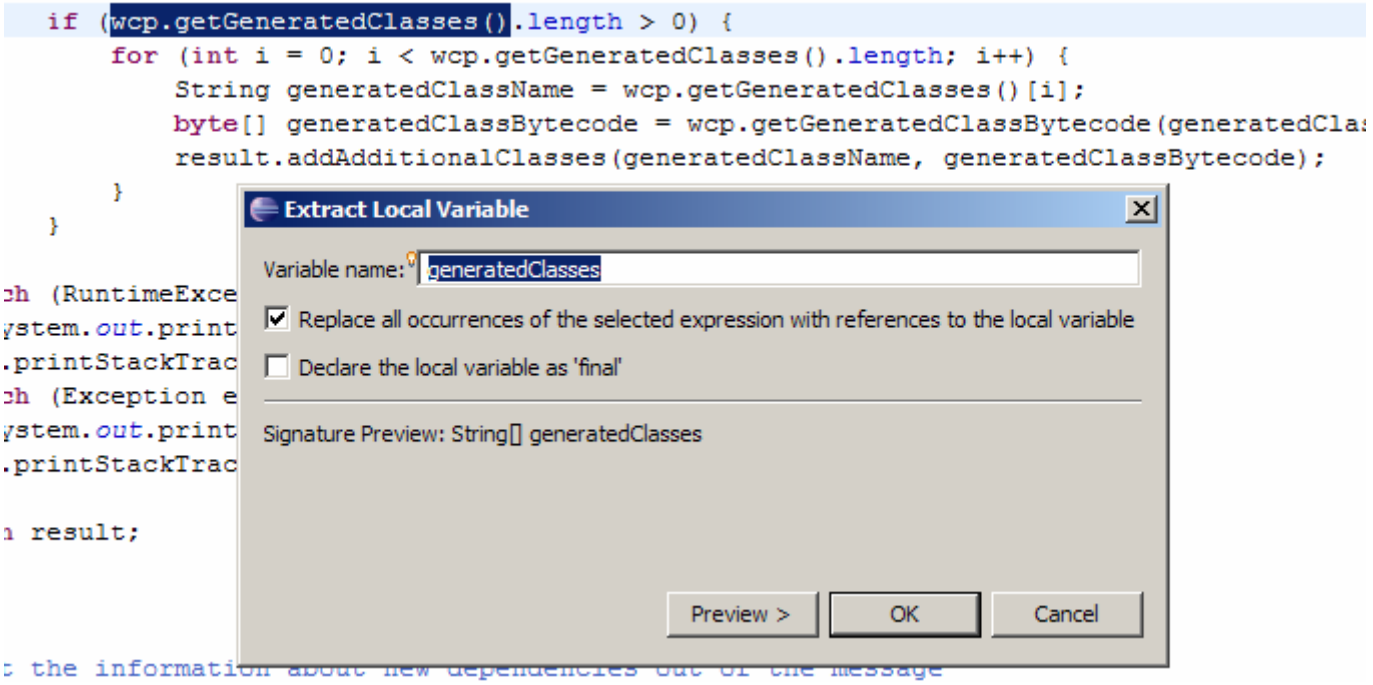
Many small refactorings

- Often small refactorings are not hard:
  - Uses little time
  - Modern IDEs often assist
  - Better than do big refactorings seldom

➔ **Try to let the IDE do the refactorings!!!**

No Refactoring by Copy&Paste

- Refactoring with Copy&Paste is old school!!!



Nevertheless big refactorings?

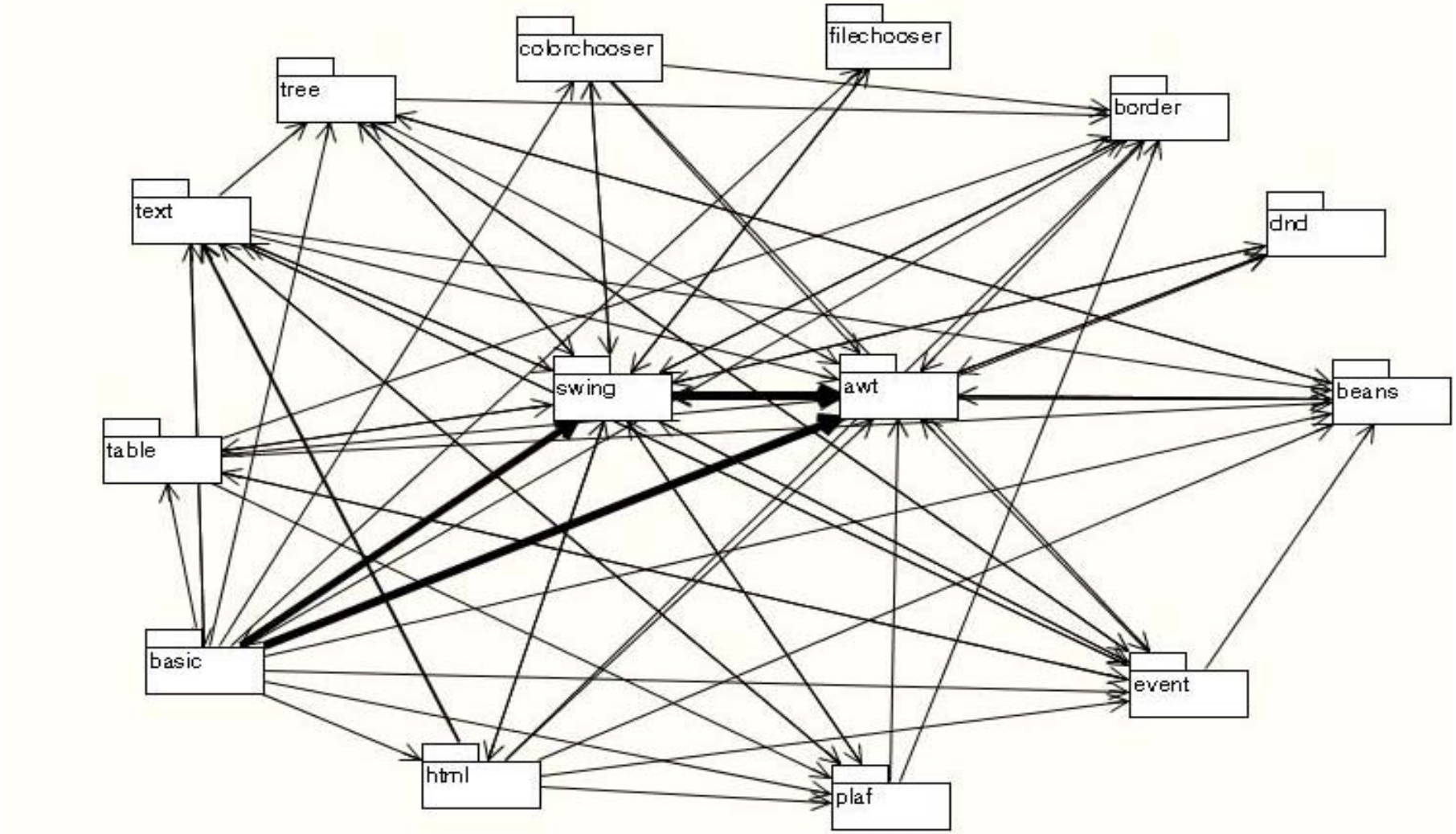
- Many small refactorings are good and irreplaceable.
- The also help us to improve the architecture of the system.

- Can big refactorings also be necessary?
- Yes:
  - Misunderstanding: You don't have to bother about architecture in agile environments.
  - Deadlines.
  - Suspended small refactorings.
  - Prototype goes productive.
  - Incomplete/inconsistent view of requirements.
  - System is very big and unclear.
  - To many developers in a team.
  - Everyone makes mistakes.
  - …

Architecture-smells and big refactorings

- In big projects often structural problems arise, so called architecture-smells.
  - Architecture-smells are potential deficits in relationships among packages, modules, classes.
- Our experience: Every big project has architecture-smells.
  (Big project: more than 6 developers, longer than 6 months)
- *Big refactorings* help, to eliminate architecture-smells.

Example für architecture-smell



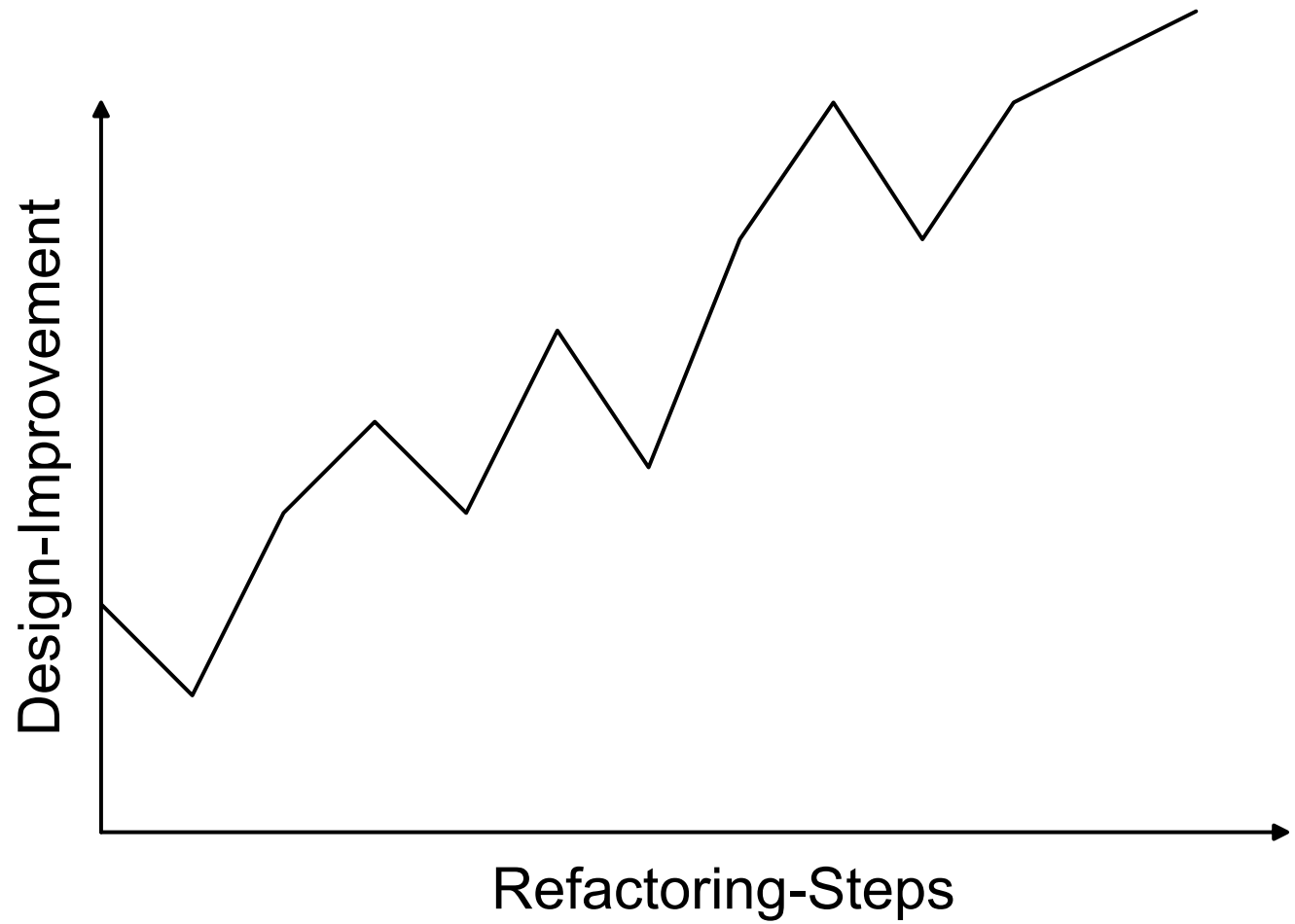
More architecture-smells.

- Parallel inheritance hierarchies
- Wrong usages of inheritance
- Cycles between classes, packages, subsystems, layers
- Upfront technology, Overgeneralization
- Unused code
- To many dependencies to base classes
- No layers, subsystems
- To big packages, subsystems, layers
- Subsystem-API not used
- Subsystem-API to big
- Layer breakthrough
- ...

Big refactorings: characterization

- Lasts longer than one day
- Leads to many changes in many parts of the system
- Affects more than one developer / pair
- Big refactoring needs to be divided
- More than a list of small refactorings
- Contains often unsafe refactorings
- The consequences of steps are hard to predict
- Big refactoring must explicitly planned
- Intermediate steps have to be integrated
- Breaks often unit-tests
- It gets worse before it gets better

Big refactorings:
One step back, two ahead :-)



Big refactorings: problems

- It's easy to run into a blind alley
- To continue with the development is not easy
- The overview can be lost
- Planning is difficult
- Security because of broken Unit-tests is reduced
- Project pressure tends to stop refactorings
  - Half done refactorings make the system structure worse

Solving problems

- Best Practices:
  - Refactor as soon as you smell something
  - Use Refactoring-Tools (many features aren't used properly)
  - Tools to identify weaknesses (during development if possible)
  - Don't be afraid of refactorings, but write test in advance
  - Discuss refactorings in the team

  - Patterns and practices for big refactorings

Best Practices: Planning of big refactorings

- Integrate refactorings explicitly in the planning process
  - Refactoring-budget per iteration
  - Refactoring-iterations if needed
  - Regular refactoring-iterations

Best practices: Refactoring-planning-session

- Refactoring-planning-session
  - Discuss and plan big refactorings with the whole team
  - Area of tension: Upfront-Design vs. Refactoring-planning

Best Practices: Refactoring-plans

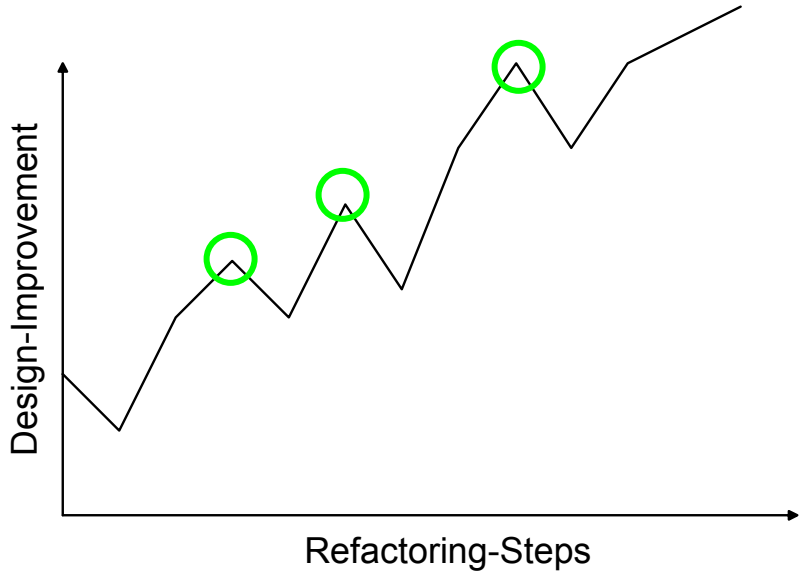
- Creating Refactoring-plans
  - Write down refactoring-route
  - Publish refactoring-plan
  - Use refactoring-plan as tracking instrument
  - Mark unsafe refactoring steps
  - Start with unsafe refactoring-steps if possible

Best Practices: redirection

- Redirection
  - To divide a refactoring into small steps, redirections have to be build into the code.
  - By doing so a refactoring can be implemented step-by-step and the system stays executable.
  - Redirections have to be marked
    - E.g. with *deprecated* tag.

Best practices: safe-points

- Safe-points
  - Divide Refactoring into small steps
  - Not after each steps the structure of the systems gets better (redirections)
  - Definition of safe-points: after each step that made the system better but hasn't reached the final design



Best practices: branches

- Branches and safe-points
  - Branches are not useable for the complete refactoring (Merge effort would be to big)
  - Use branches to a defined safe-point and than merge

Finding architecture-smells

- Develop
  - What is in the way?
- Listening to developers:
  - „This is nerves, but we have no time to change."
  - „This is not useable at all, but if we change it we might break everything."
- Tools for architecture analysis, e.g.
  - **Sotograph** (http://www.sotograph.de).
  - **XRadar** (http://xradar.sourceforge.net)
  - Dr. Freud (http://www.freiheit.com)
  - ...
- Other Tools, e.g.
  - PMD
  - Findbugs
  - Checkstyle
  - ...

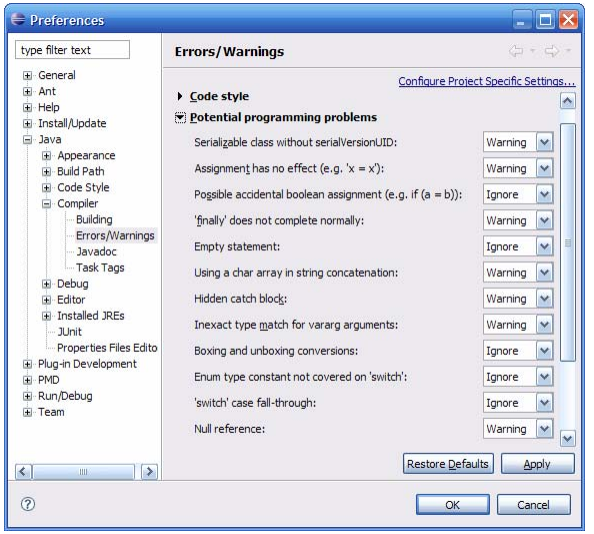
Similarities between the tools

- Open Source
- Java
- Grown-up (at least good beta)

Eclipse !

- Use Eclipse-Warnings/Errors
- Eclipse has many build in code checks
- Preferences > Java > Compiler > Errors/Warnings
- Customize and enable warnings
  - like "Null reference"
- Individual warnings can be ignored with @SuppressWarnings("null")

- Best integration among tools



PMD

- Static code scanner for Java
  - Potential bugs
    - like empty try/catch/finally/switch-statements
  - Dead code
  - Suboptimal code
    - Like, for example, wasteful use of String/StringBuffer
  - Duplicate code
  - More see http://pmd.sourceforge.net/rules/
- Rule based
  - Write own rules or customize existing rules
- Integrated in:
  - Eclipse, IntelliJ's IDEA, Maven, Ant, …
- Book:
  - PMD Applied by Tom Copeland

Example: PMD-Report of Hibernate

➔Live-Demo

PMD report

Problems found

#	File	Line	Problem
1	hibernate/Environment.java	246	Avoid unused private fields such as 'jvmSupportsProxies'
2	hibernate/eg/Edge.java	59	Avoid unused private methods such as 'setKey(long)'
3	hibernate/eg/Edge.java	67	Avoid unused private methods such as 'setCreationDate(Date)'
4	hibernate/eg/Vertex.java	73	Avoid unused private methods such as 'setKey(long)'
5	hibernate/eg/Vertex.java	81	Avoid unused private methods such as 'setVersion(int)'
6	hibernate/eg/Vertex.java	89	Avoid unused private methods such as 'setCreationDate(Date)'
7	hibernate/engine/Cascades.java	206	Avoid unused private methods such as 'cascade(SessionImplementor,Object,Type,CascadingAction,int)'
8	hibernate/engine/Versioning.java	49	Avoid unused formal parameters such as 'versionType'
9	hibernate/engine/Versioning.java	53	Avoid unused formal parameters such as 'versionType'
10	hibernate/helpers/IdentityMap.java	68	Avoid unused formal parameters such as 'k'
11	hibernate/id/UUIDStringGenerator.java	55	Avoid unused private methods such as 'toString(int)'
12	hibernate/impl/CollectionPersister.java	271	Avoid unused private methods such as 'getSQLSelectString()'
13	hibernate/impl/QueryImpl.java	283	Avoid unused private methods such as 'guessType(Object)'
14	hibernate/impl/ScrollableResultsImpl.java	25	Avoid unused private fields such as 'single'
15	hibernate/impl/SessionFactoryImpl.java	90	Avoid unused private fields such as 'properties'
16	hibernate/impl/SessionFactoryImpl.java	93	Avoid unused private fields such as 'supportsLocking'
17	hibernate/impl/SessionFactoryObjectFactory.java	32	Avoid unused private fields such as 'INSTANCE'
18	hibernate/impl/SessionImpl.java	122	Avoid unused private fields such as 'reentrantCallback'
19	hibernate/impl/SessionImpl.java	829	Avoid unused private methods such as 'removeCollectionsFor(ClassPersister,Serializable,Object)'
20	hibernate/impl/SessionImpl.java	2063	Avoid unused formal parameters such as 'owner'

FindBugs

- http://findbugs.sourceforge.net/
- Searches for bugs in the code
- Uses bug-pattern-concept
- Static code analysis of bytecode
- Control with
  - Swing-GUI
  - Ant-Task
  - Eclipse-Plugin
- Results in
  - HTML
  - Swing-GUI
  - Eclipse

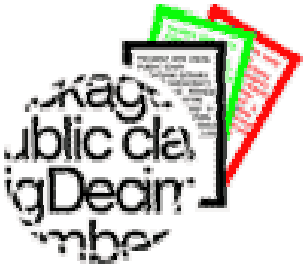
➔Example

➔Live-Demo

CheckStyle

- Static check of code conventions
- Integration for
  - Eclipse, IntelliJ's IDEA, NetBeans, JBuilder …
- Executable with
  - Ant
  - Maven
- Results in
  - HTML
  - Markers in IDEs
- Rule based
  - Write own rules ➔ adjust to your own!

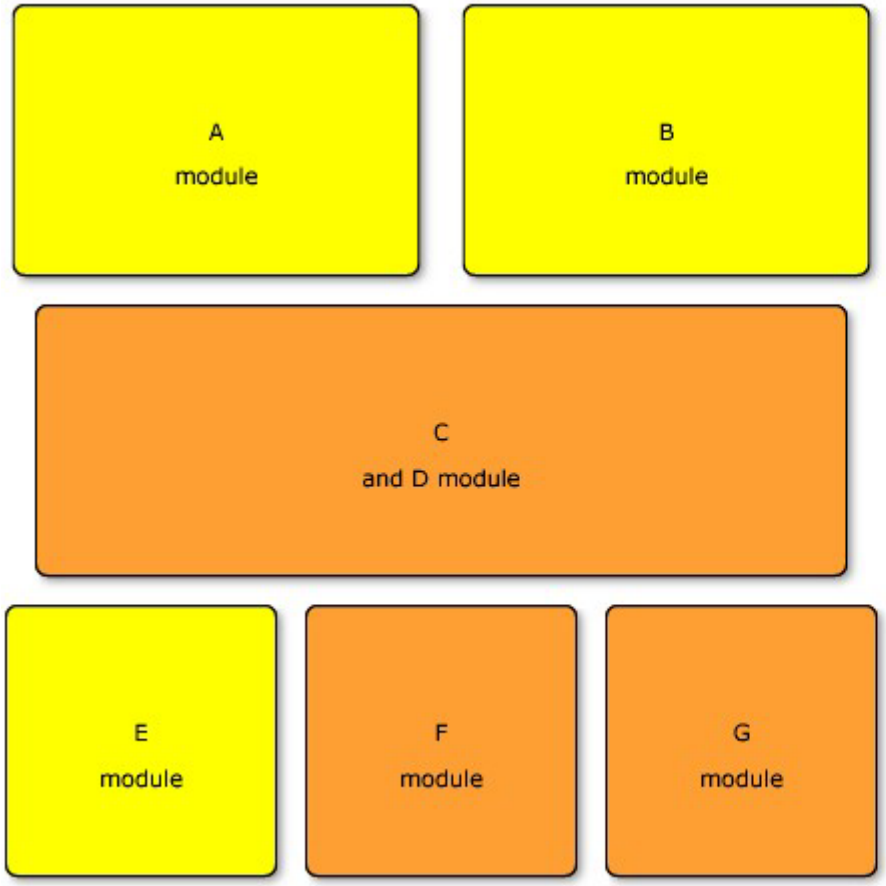
➔ Live-Demo



XRadar

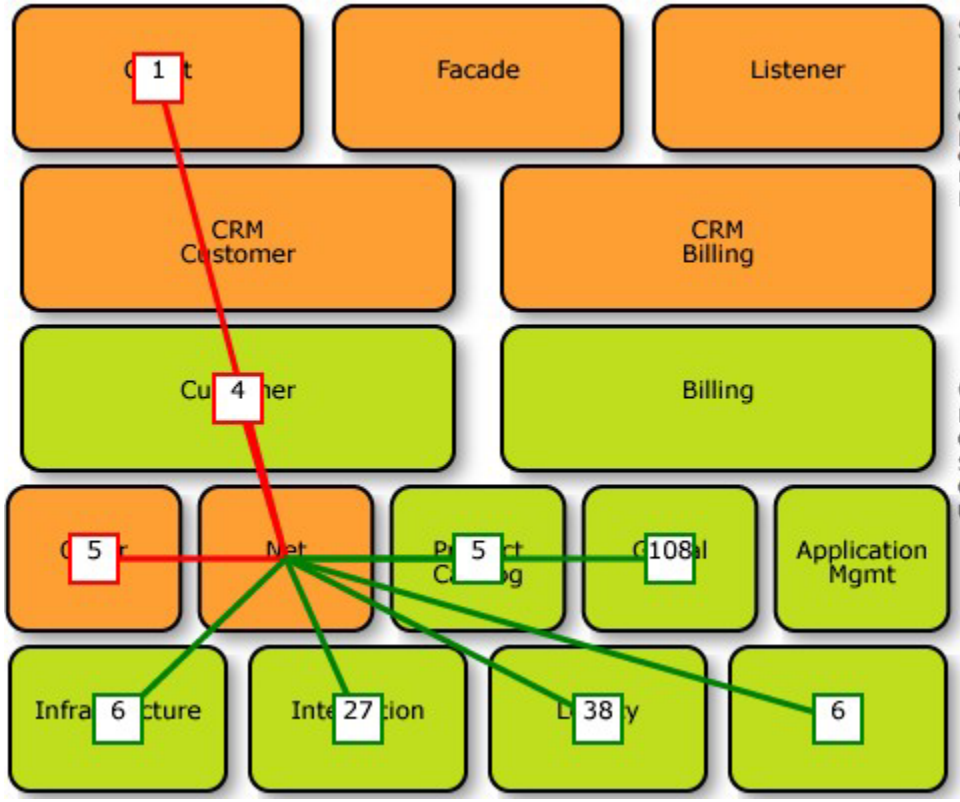
- http://xradar.sourceforge.net/
- Top down view of a SW-Project
  - Static analysis with for current state
  - Dynamic analysis with history
- Integrates many other open source-tools
  - JUnit, Cobertura, JCoverage, JDepend, PMD, CheckStyle, JavaNCSS …
- Results are concentrated in
  - HTML-tables
  - SVG-grafics
- Executable with
  - Ant
  - Maven

XRadar – Example 1/3 – Static analysis



Total Quality [TQ= 0.35*ARCH + 0.30*CODE + 0.35*TS]	0.27
Test Suite [TS= 1*TCU]	0.23
Unit Test Coverage [TCU= source-statements-covered÷ncss]	0.23
Architecture [ARCH= 0.4*MOD + 0.6*COH]	0.34
Modularisation [MOD= 1 - (count_packages(not(illegal-dependencies=0)) ÷total_packages)]	0.43
Cohesion [COH= 1 - (count_packages(cycles=true)÷total_packages)]	0.29
Code Quality [CODE= 0.15*DOC + 0.4*DRY + 0.3*FRE + 0.15*STY]	0.25
Documentation [DOC= javadocs÷(functions + analysed-classes)]	0.24
DRYness [DRY= 1 - (classes-with-duplications÷analysed-classes)]	0.44
Freshness [FRE= 1 - (classes-with-code-violations÷analysed-classes)]	0.11
Stylishness [STY= 1 - (classes-with-style-errors÷analysed-classes)]	0

XRadar – Example 2/3 – Static analysis



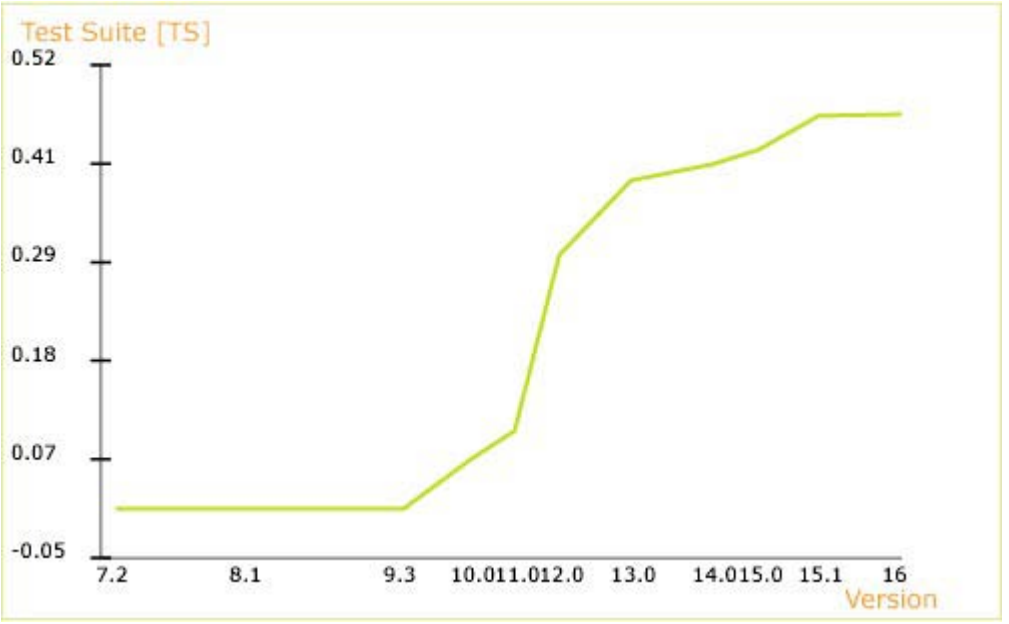
Snapshot - Net

This text describes the subsystem. The text pops up on the screen when you focus on the module. You can use it for educational purposes of both new developers and managers on the system. Below you find some core metrics, and how they relate to the overall picture.

Core Metrics

Packages =	35 (10.6%)
Classes =	503 (10.8%)
Source Statements =	22527 (17.7%)
Cyclomatic Complexity =	7296 (16.5%)
Unit Test Coverage =	38.6%

XRadar – Example 1/3 – Static analysis

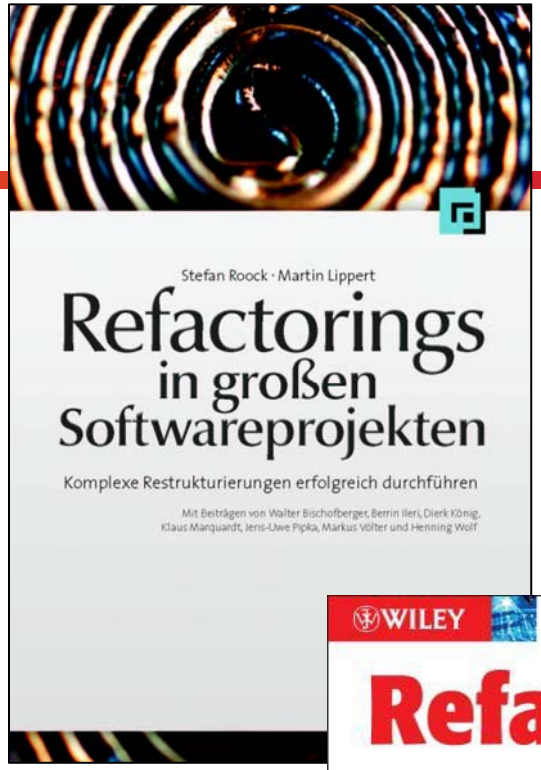


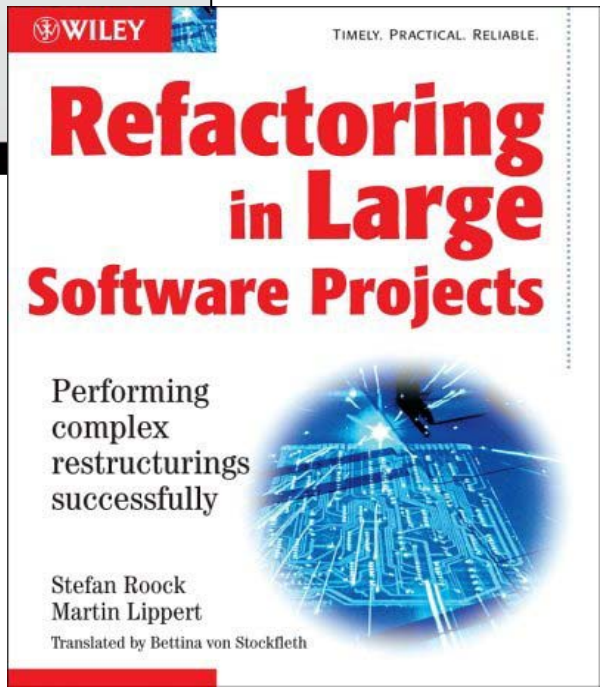
Conclusion

- **Refactoring code is more important than coding.**
  - We use more time improving existing code than implementing new code.
- Use refactoring-tools!!!
- Refactorings are only safe with unit-tests!
- Really important:
  - **Refactorings should not be suspended!**
  - **Refactorings have to be discussed in a team!**
  - **Ask the experts! ;-)**

Some advertisement ... ;-)

- Refactoring-introduction
- Architecture-smells
- Characteristics of big refactorings
- Parts of refactorings
- Process aspects
- Database and refactoring
- APIs und refactoring





More references

- Martin Fowler: *Refactoring – Improving the Design of Existing Code*, Addison-Wesley, 1999
- Joshua Kerievsky: *Refactoring to Patterns*, Addison-Wesley, 2004
- William Wake: *Refactoring Workbook*, Addison-Wesley, 2003.

- On the road:
  - Ramnivas Laddad: *Aspect Oriented Refactoring*, Addison-Wesley, 2006
  - Scott W. Ambler, Pramodkumar J. Sadalage: Refactoring Databases: Evolutionary Database Design, Addison-Wesley, 2006

The end

- Thank you for your attention. Feedback is welcome!
  Martin Lippert: martin.lippert@it-agile.de
  Matthias Lübken: matthias.luebken@it-agile.de

- Some interesting references:
  - http://www.refactoring.com/: Maintained by Martin Fowler, contains a lot of useful other references, articles, tools catalog, …
  - http://www.refactoring.be/: Refactoring Thumbnails as a visualization for refactorings
  - http://groups.yahoo.com/group/refactoring: Refactoring mailing list at Yahoo