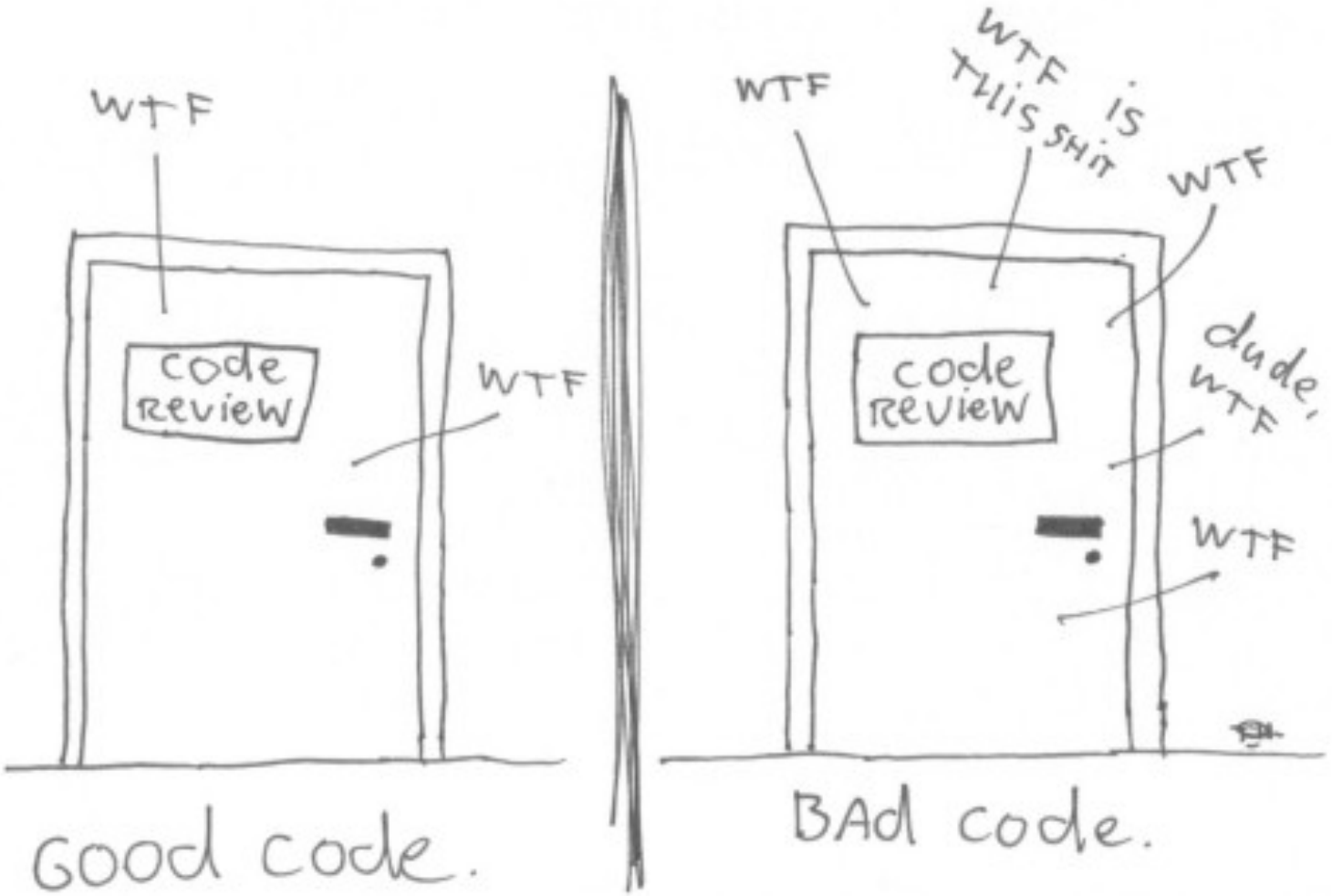
EECE 310

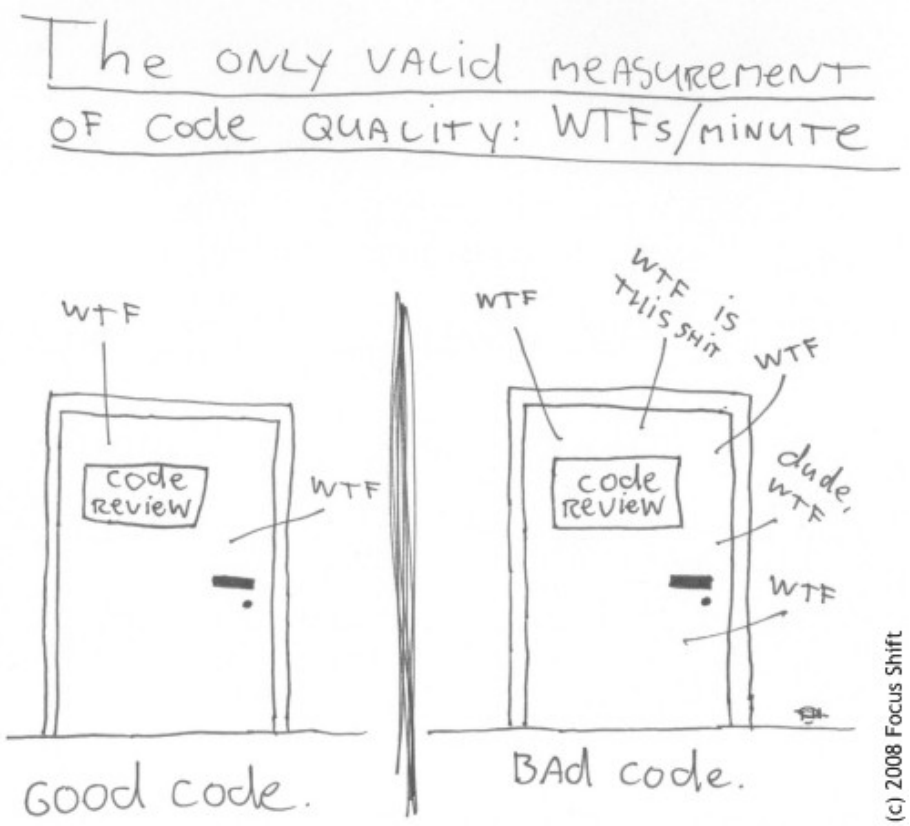
Code Quality

What defines code quality? How do we measure it?

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



Which door is yours? How can we end up at the right door?



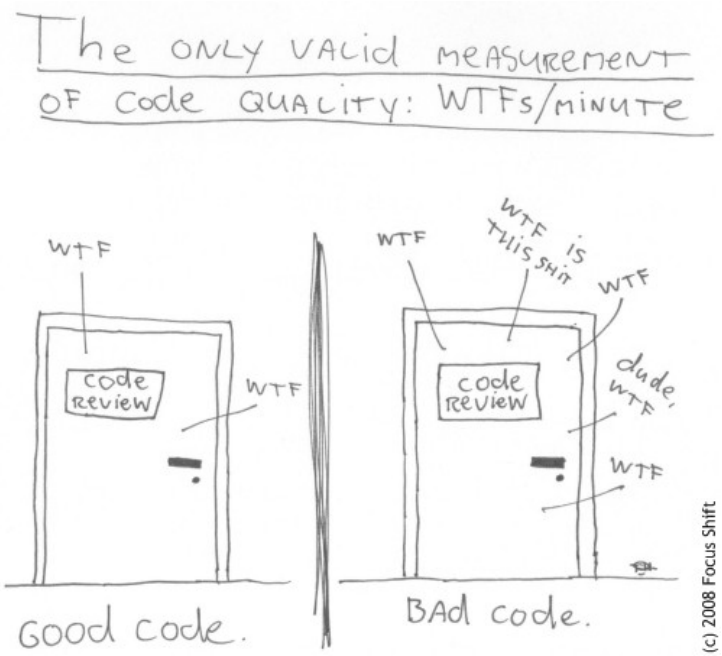
Ratio of Writing versus Reading Code

*"The ratio of time spent reading code versus writing is well over **10** to **1**.*

Therefore making it easy to read makes it easier to write."

Which door is yours? How can we end up at the right door?

- (Automated) Analysis
  - Code Metrics
  - Formatting
  - Static and Dynamic Analysis
  - Measuring and Monitoring
- Craftsmanship
  - Code Reviews
  - Clean code
  - Smells and Refactoring



Metrics

- What metrics can we use to measure the quality of our code?

Metrics

- Lines of code (LOC)
- Number of classes, functions/methods
- Lines of comments

Metrics

- Complexity
  - Cyclomatic Complexity: measures number of linearly independent paths through a program's source code.
  - Cohesion and Coupling
  - Fan in and Fan out
- Maintainability Index
  - calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. Uses a combination of LOC, and complexity metrics

Code Formatting

- Adopt a code standard. Why?

Code Standards (1)

- Why?
  - Gives **less** defects.
  - **Easier/cheaper** maintenance.
  - Several people may **work on and understand** the same code.
  - Makes seeing the DIFFs easier when working with version control systems.

- JPacman coding standard:
  - code-formatter.xml
  - Checkstyle
    - mvn checksytle

    https://maven.apache.org/plugins/maven-checkstyle-plugin/

Code Standards (2)

- Allows automated formatting of your code.

- General rules:
  - **Code geometry**, e.g., line width,
  - **Code look and feel**

```
public String foo(a ,b, c)
{
    return “example”;
}
```

Code Standards (3)

- Always use a coding standard in your projects. Set it up in the beginning and do it **right** the first time!
- Automate the formatting.
- Your **professionalism** is expressed by applying code standards!

Formatting

- Is important for **readability**, not for the compiler.
- Use a **common** standard for code formatting.
- Do **not alter** the style of old code to fit new standards.

- Formatting also makes it easier to detect bugs!

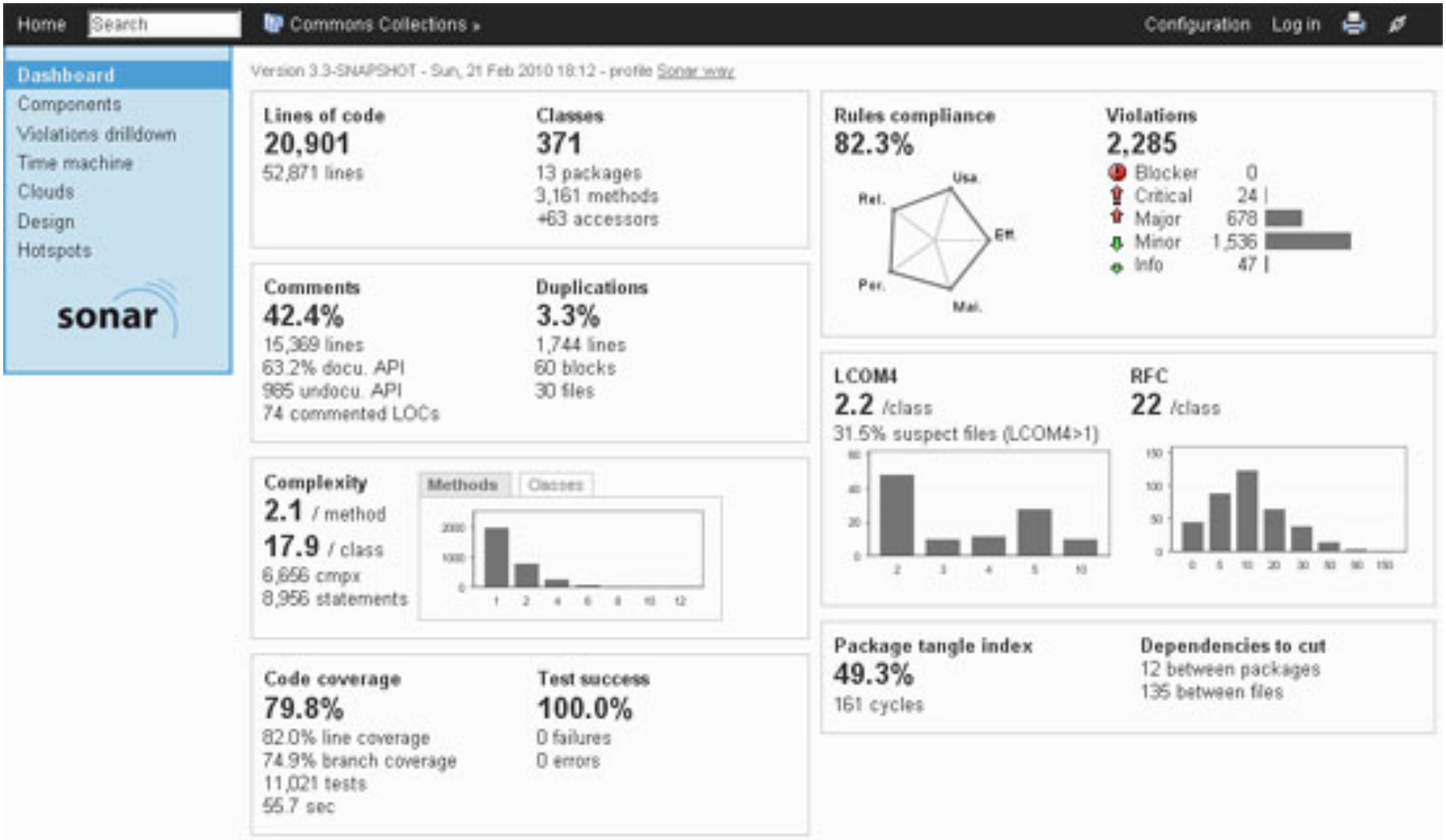
Static and Dynamic Analysis

- Measure test **coverage** (EclEmma)
  - Spot uncovered potions of your code
  - Treat uncovered code as a suspect (could contain bugs)
- Deploy static analysis tools on your code
  - FindBugs (binary)
  - PMD (source code)

Measuring and Monitoring

- Set up Continuous Integration
  - Jenkins, Travis
- Integrate Monitoring Systems
  - Sonar: many plugins to measure code quality

Sonar


Home
Search
Commons Collections »
Configuration
Log in
Dashboard
Components
Violations drilldown
Time machine
Clouds
Design
Hotspots
sonar
Version 3.3-SNAPSHOT - Sun, 21 Feb 2010 18:12 - profile Sonar way
Lines of code
20,901
52,871 lines
Classes
371
13 packages
3,161 methods
+63 accessors
Rules compliance
82.3%
Usa
Rel.
Eff.
Per.
Mai.
Violations
2,285
Blocker 0
Critical 24
Major 678
Minor 1,536
Info 47
Comments
42.4%
15,369 lines
63.2% docu. API
985 undocu. API
74 commented LOCs
Duplications
3.3%
1,744 lines
60 blocks
30 files
LCOM4
2.2 /class
31.5% suspect files (LCOM4>1)
RFC
22 /class
Complexity
2.1 / method
17.9 / class
6,656 cmpx
8,956 statements
Methods
Classes
Package tangle index
49.3%
161 cycles
Dependencies to cut
12 between packages
135 between files
Code coverage
79.8%
82.0% line coverage
74.9% branch coverage
11,021 tests
55.7 sec
Test success
100.0%
0 failures
0 errors

Dashboard
SQALE
Issues
Hotspots
Time Machine
By Developers

TOOLS
Components
Issues Drilldown
Design
Libraries
Clouds
Compare

Version 3.1-SNAPSHOT - Feb 08 2014 23:23 Time changes...

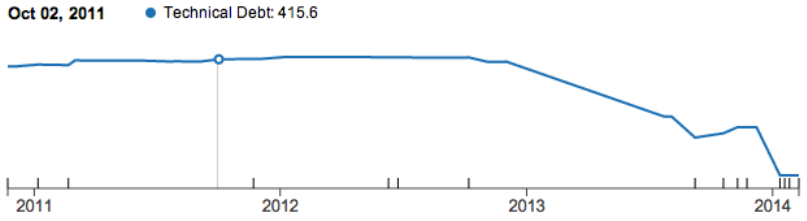
Lines of code
55,746
91,134 lines
18,646 statements

Files
623
700 classes
4,475 functions
683 accessors

Issues
1,620

Blocker	2
Critical	109
Major	1,100
Minor	409
Info	0

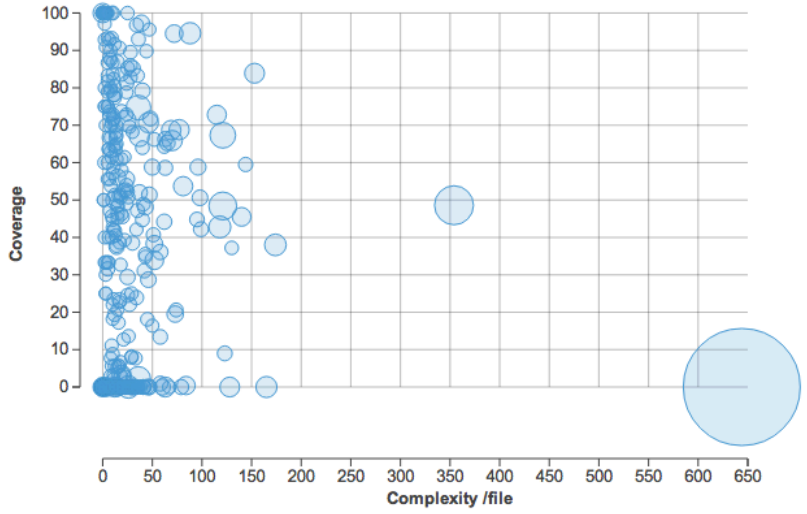
Technical Debt
71.9 days



No alerts.

Unit Tests Coverage
35.9%
38.5% line coverage
30.4% branch coverage

Unit test success
100.0%
0 failures
0 errors
685 tests
51.6 sec

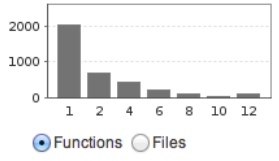


Documentation
29.7% docu. API
3,878 public API
2,726 undocu. API

Comments
20.7%
14,509 lines

Duplications
2.4%
2,174 lines
77 blocks
46 files

Complexity
2.2 /function
14.2 /class
16.0 /file
Total: 9,971

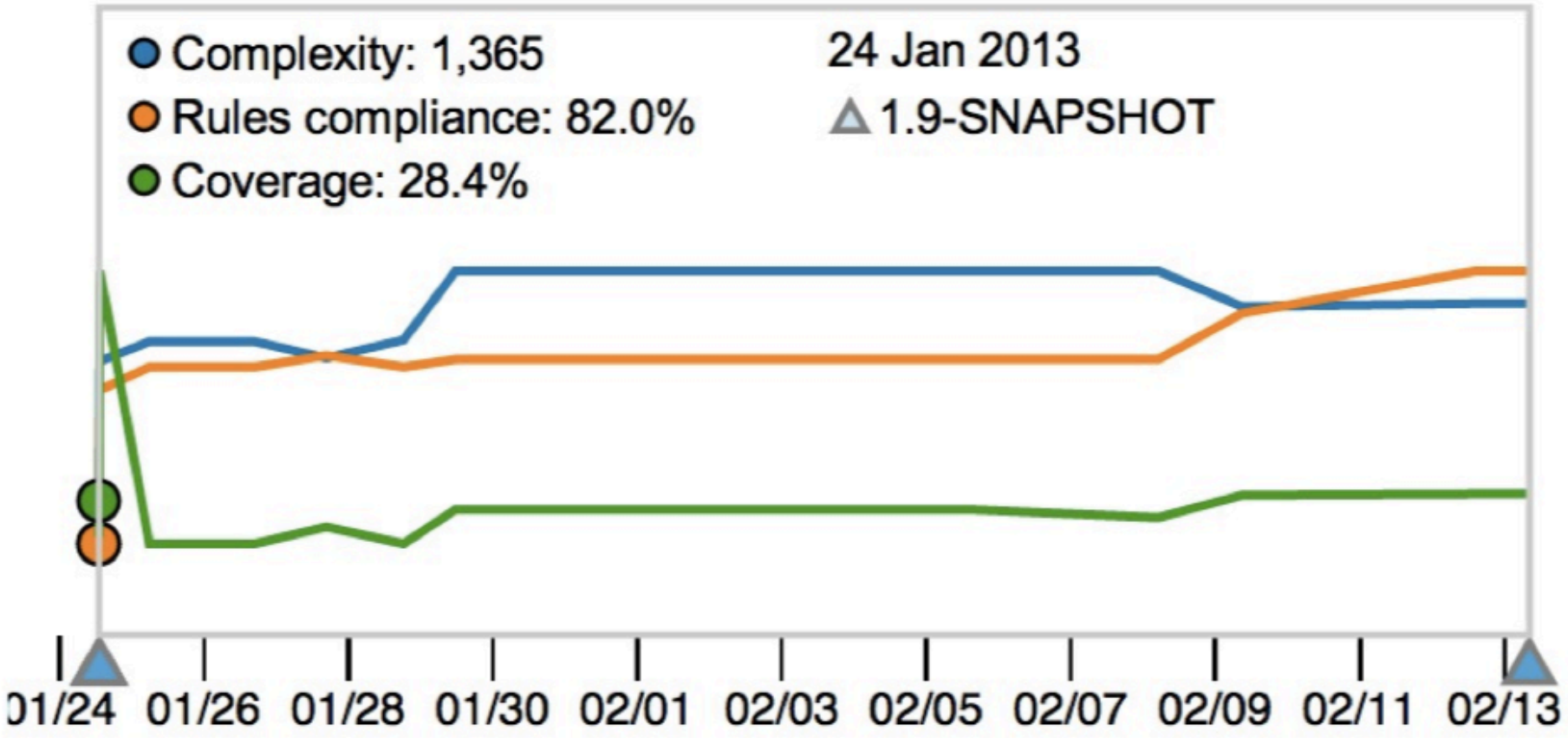


Size: **Lines of code** Color: **Coverage** 0.0% 100.0%

Package tangle index
24.9%
> 64 cycles

Dependencies to cut
37 between packages
99 between files

Sonar


Complexity: 1,365
Rules compliance: 82.0%
Coverage: 28.4%
24 Jan 2013
1.9-SNAPSHOT
01/24
01/26
01/28
01/30
02/01
02/03
02/05
02/07
02/09
02/11
02/13

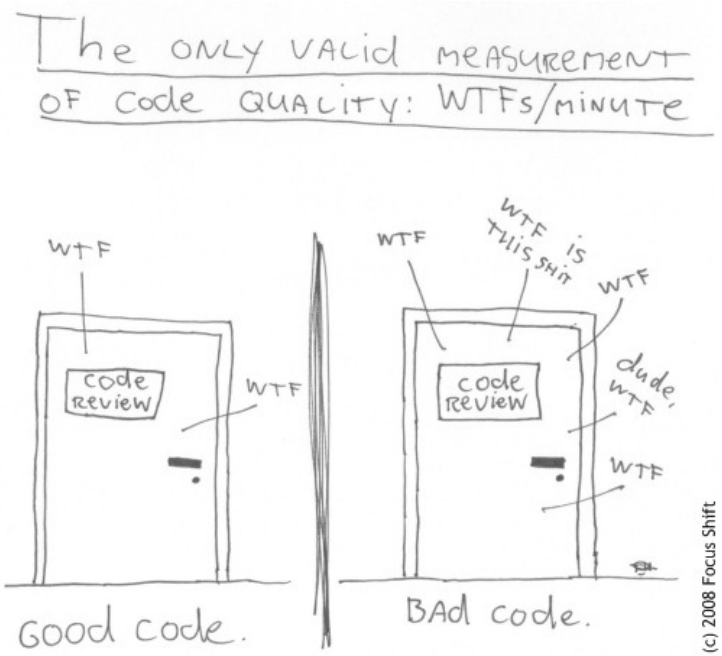
Violations

	24 Jan 2013 1.9-SNAPSHOT	24 Jan 2013 2.2	13 Feb 2013 2.3-SNAPSHOT
Violations	384	403	370
Blocker violations	0	0	0
Critical violations	22	6	3
Major violations	341	362	323
Minor violations	21	35	36
Weighted violations	1,154	1,151	1,020

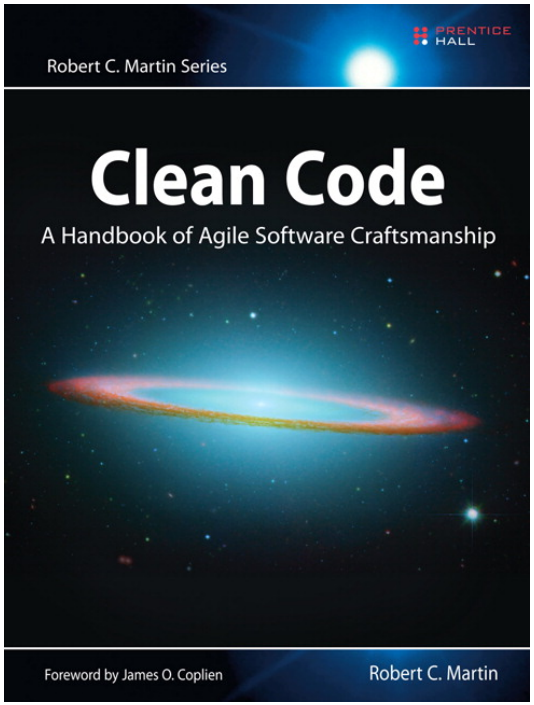
Which door is yours? How can we end up at the right door?

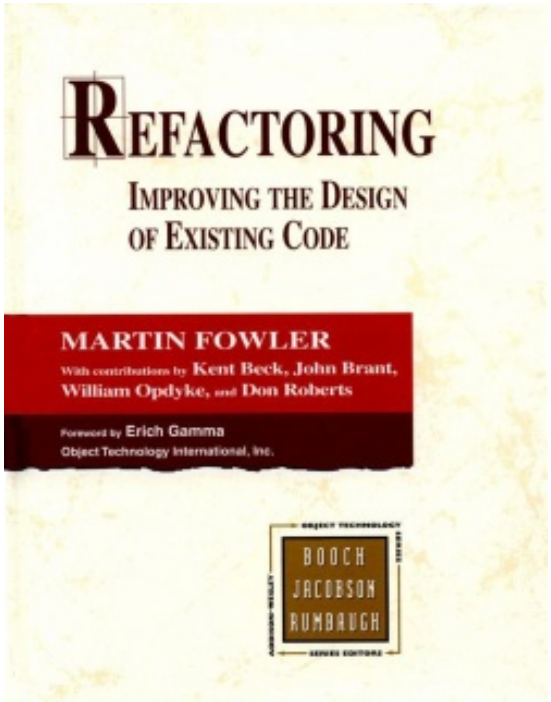
- **(Automated) Analysis**
  - **Code Metrics**
  - **Formatting**
  - **Static and Dynamic Analysis**
  - **Measuring and Monitoring**

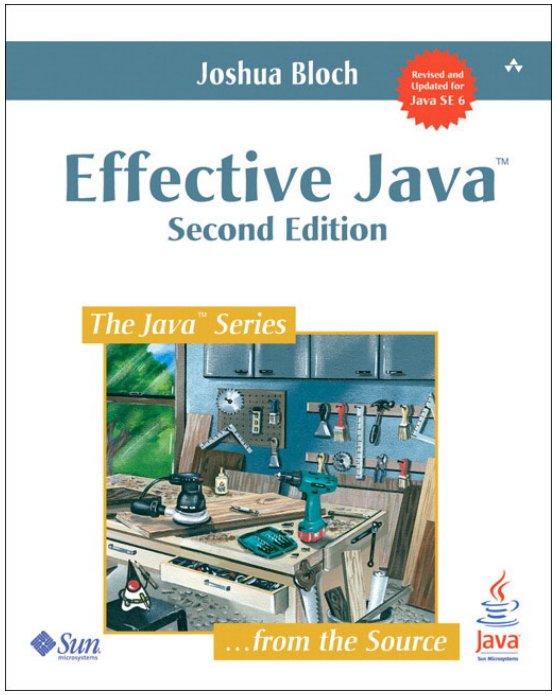
- Craftsmanship
  - Code Reviews
  - Clean code
  - Smells and Refactoring



Software Craftsmanship







Code Reviews

- Detect bugs early on

But, recent studies show that is not necessarily the case:

- Helpful to detect inconsistencies
- Getting familiar with the code
- Knowing what is being changed where

Code Reviews

- The GitHub workflow facilitates code reviews

- Send your changes as Pull Request
- Someone else reviews your changes
- Gives feedback
- Only when the feedback is applied
- Your changes get merged into the master branch

Clean Code

- Has tests which all succeed
- Is easy to read
- Has a clear intent
- Makes it hard to hide bugs

Naming

- Clear, unambiguous, readable, **meaningful.** Describe the purpose of the item:
  - Bad: X1, X2, mth, get, tmp, temp, result.
  - Give a descriptive name to temporary variables.

Naming

- Establish and use a **common** naming convention.
- Problems creating a good name ⇒ purpose of the operation is not clear.
  - Bad: void get(…)., better: retrieveDataSamples.
  - Bad: Time day(Time p_day), better: getDate or getTruncDate.
  - Bad: void result(…), better: createResults.
  - Bad: void gas/oil/water, better: calculate…VolumeRate.

Java Naming Convention

- Package: scope.mypackage
- Classes: MyClass
- Methods: myMethod
- Constants: MY_CONSTANT
- Attribute: myAttribute
- Variable: myVariable

Clean methods

- Good methods and classes:
  - do as they promise
  - do one thing, and one thing only
  - are small

Code Smells

Can you find the smells?

```
public ArrayList<int[]> getThem() {
   ArrayList<int[]> ArrayList1 = new
      ArrayList<int[]>();

   for (int[] x : globalArrayList)
     if (x[0] == 4)
         ArrayList1.add(x);
   return ArrayList1;
}
```

Can you find the smells?

```
public ArrayList<int[]> getThem() {
   ArrayList<int[]> ArrayList1 = new
      ArrayList<int[]>();


   for (int[] x : globalArrayList)
      if (x[0] == 4)
         ArrayList1.add(x);
   return ArrayList1;
}
```

Why is it smelly?

- Because we cannot comprehend what the author wants to express in his code. The code is very implicit:

1. What kinds of things are in globalArrayList?
2. What is the significance of the 0th subscript of an item in globalArrayList?
3. What is the significance of the value 4?
4. How would I use the list being returned?

Code Smells and Refactoring

- Definition: Refactoring modifies software to improve its readability, maintainability, and extensibility without changing what it actually does.
- External behavior does NOT change
- Internal structure is improved

Refactoring

- The goal of refactoring is NOT to add new functionality
- The goal is refactoring is to make code easier to maintain in the future

Can you refactor the example?

```
public ArrayList<int[]> getThem()
{
   ArrayList<int[]> ArrayList1 =
   new ArrayList<int[]>();
   for (int[] x : globalArrayList)
      if (x[0] == 4)
          ArrayList1.add(x);
   return ArrayList1;
}
```

Can you refactor the example?

- **No because we don't know what the code is supposed to do.**
- **But if we knew the goal is to get the list of *running members?***

Refactored

```
public ArrayList<Member> getRunningMembers() {
  ArrayList<Member> runningMembers = new
       ArrayList<Member>();
  for (Member member : allMemberList) {
       if (member.isRunner()) {
            runningMembers.add(member);
       }
   }
  return runningMembers;
}
```

```
Public void updateLastName(int id, String name)
{
  Connection con = Database.getConnection();
  try { con.open(); }
  catch (Exception e) {
   throw new RuntimeException(e);
  }
 String q = "select * from person where id = "
+ id;
 Person p = con.executeQuery(q);
 p.setLastName(name);

 try {
  con.persist(p);
 }  ….
```

Refactored

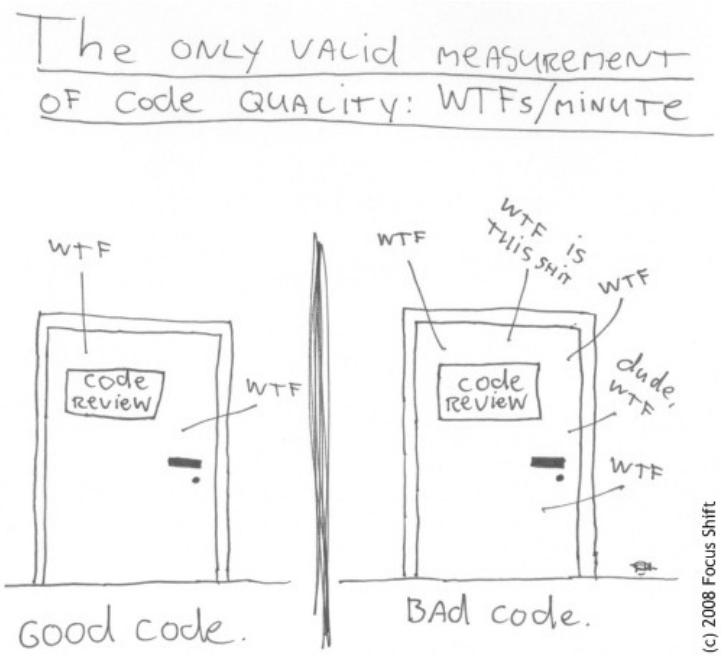
```
public void updateFirstName(int id,
String name) {

 Person person = getPersonById(id);
 person.setFirstName(name);
 store(person);
}
```

What defines code quality? How do we measure it?

Which door is yours? How can we end up at the right door?

- (Automated) Analysis
  - Code Metrics
  - Formatting
  - Static and Dynamic Analysis
  - Measuring and Monitoring
- Craftsmanship
  - Code Reviews
  - Clean code
  - Smells and Refactoring



Do we want comments in our code?

Good comment?

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
  …
}

/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;
```

Noise comment

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
  …
}

/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;
```

Good comment?

```
 /**
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of in minutes
 */
 public void addCD(String title, String author,
 int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    …
}
```

Mandated comments

```
 /**
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of in minutes
 */
 public void addCD(String title, String author,
 int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    …
}
```

Commented-Out Code

```
Public void() readFile() {
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(),
formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount())

…
}
```

Good or bad?

Commented-Out Code

- Avoid this bad practice!
- Others won't know why the code is kept in there
  - Is it still valuable?
  - Was the developer unsure of what they were doing?
  - Can I delete this? What if they need it?
- Maybe was needed in the 60s when there were no version control systems

?

```
private void startSending(){
  try { doSending(); }
  catch(SocketException e) {
    // normal. someone stopped the request.
  }
  catch(Exception e) {
    try {
     response.add(ErrorResponder.makeExceptionString(e));
     response.closeAll();
    }
    catch(Exception e1) {
      //Give me a break!
    }
} }
```

Worthless and noisy comment

```
private void startSending(){
  try { doSending(); }
  catch(SocketException e) {
    // normal. someone stopped the request.
  }
  catch(Exception e) {
    try {
     response.add(ErrorResponder.makeExceptionString(e));
     response.closeAll();
    }
    catch(Exception e1) {
      //Give me a break!
    }
} }
```

Do we want comments in our code?

- Comments do not make up for bad code.
- Explain yourself in code where ever possible
- Replace the temptation to create **noise** with the determination to clean your code. You'll find it makes you a better and happier programmer.

Comments

- **What**; Document complex algorithms, ***avoid obvious comments***!
- **Why**: To be able to find out what a method does after a half, one or two years. Automatic API documentation.
- **When**; Document your code before or when you write it; Design before you implement. Put the design in the method.
- **Where**; Ideally before the method, if needed at specific decision points in the method.

JavaDoc (1)

- Generates HTML-formatted class reference or API documentation.
- Only recognizes documentation comments that appear immediately before
  - class
  - Interface
  - constructor,
  - method
  - field declaration

JavaDoc (2)

- Purpose: To define a **programming contract** between a *client* and a supplier of a *service*.
- Keep the documentation synchronised with the code. ⇒ **MAKE DOCUMENTATION FIRST**, then code!
- JavaDoc tags:
  - @author, @version, @see, @param, @return, @exception.
  - {@link}.

JavaDoc Example: Class

```
/**
 * <code> MyConstants </code> is the base class for all the constants and
 *  implements any general constant responsibilities.
 *
 * @author John Doe
 * @version $Revision: 6.2$
 * @invariants p > 0
 * …
 * @see MySpecificConstants
 */
```

JavaDoc (3)

- Document pre-, post- and invariant conditions
  - @pre-condition, @post-condition, @invariants.
- Document known defects and deficiencies.
  - @defect.
- Use @todo only if really really needed.
  - Todo's tend to be ignored most of the time.

JavaDoc Example: Method

```
/**
 * Method description
 *
 * @param paramName  Name of the mechanism to search for,
 *        one of the constants in the <code>MyClass</code> class.
 * @return The concrete instance of an <code>MyClass</code>
 *        that is currently in effect if one found, null if not.
 * @exception Throwable Default finalizer exception.
 * @pre-condition valid paramName.
* @post-condition (…)
 */
```

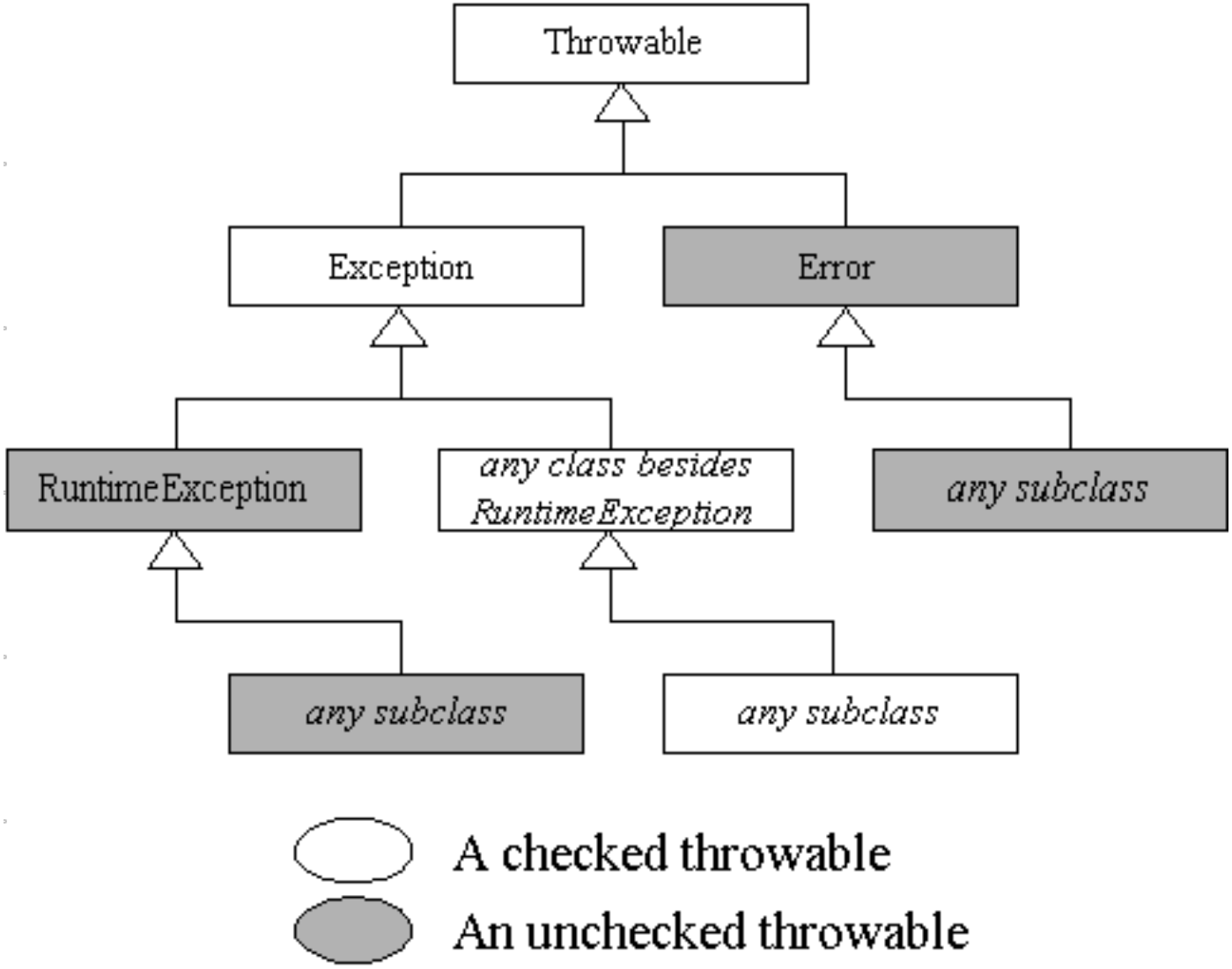
Defensive Coding

- Check input data for validity **(Pre-conditions**).
  - **Range**, comment assumptions about acceptable input ranges in the code.
  - Use a general approach for error handling when erroneous data is given as input.
- Use exception handling only to draw attention to unexpected cases (Do **NOT** perform any processing in exceptional code)
- Hide as much as possible to minimise impact of change.

Checked vs unchecked exceptions

- Any opinions?

Checked vs unchecked exceptions



Checked vs unchecked exceptions

- It's better to use unchekced exceptions
- Use checked exceptions only for critical cases that require special handling
- Checked violates Open/Closed principle:
  - If you throw a checked exception and the catch is three levels up, you must declare that exception in the signature of each method between you and the catch.
  - A change in low lever method can force signature changes on many higher levels

Logging

- **Avoid** using System.out.println("…") in your code
  - Why?
- Introduce debugging aids early (logging).
  - Log4j library
- Return friendly error messages; Write to a log file any system specific error messages (IO/SQL Exceptions, error codes etc.).