Software Quality Management

Dr. Stefan Wagner
Technische Universität München

Garching
28 May 2010

Some of these slides were adapted from the tutorial "Clone Detection in Practice" from
F. Deissenbock, B. Hummel, and E. Jürgens given at ICSE'10.

Last QOT: On which quality attributes do reviews have the most direct influence?

"Usability"

"Readability"

"Performance"

Reviews might have an influence on usability, depending on what is actually reviewed. Reviews of GUI mock-ups,
for example, might have a huge influence.

Readability is highly influenced in the case of code reviews.

Performance problems are actually very hard to detect with reviews.

New QOT: "Why is cloning a problem?"

Inspection

Review

Walkthrough

Review of last week's lecture

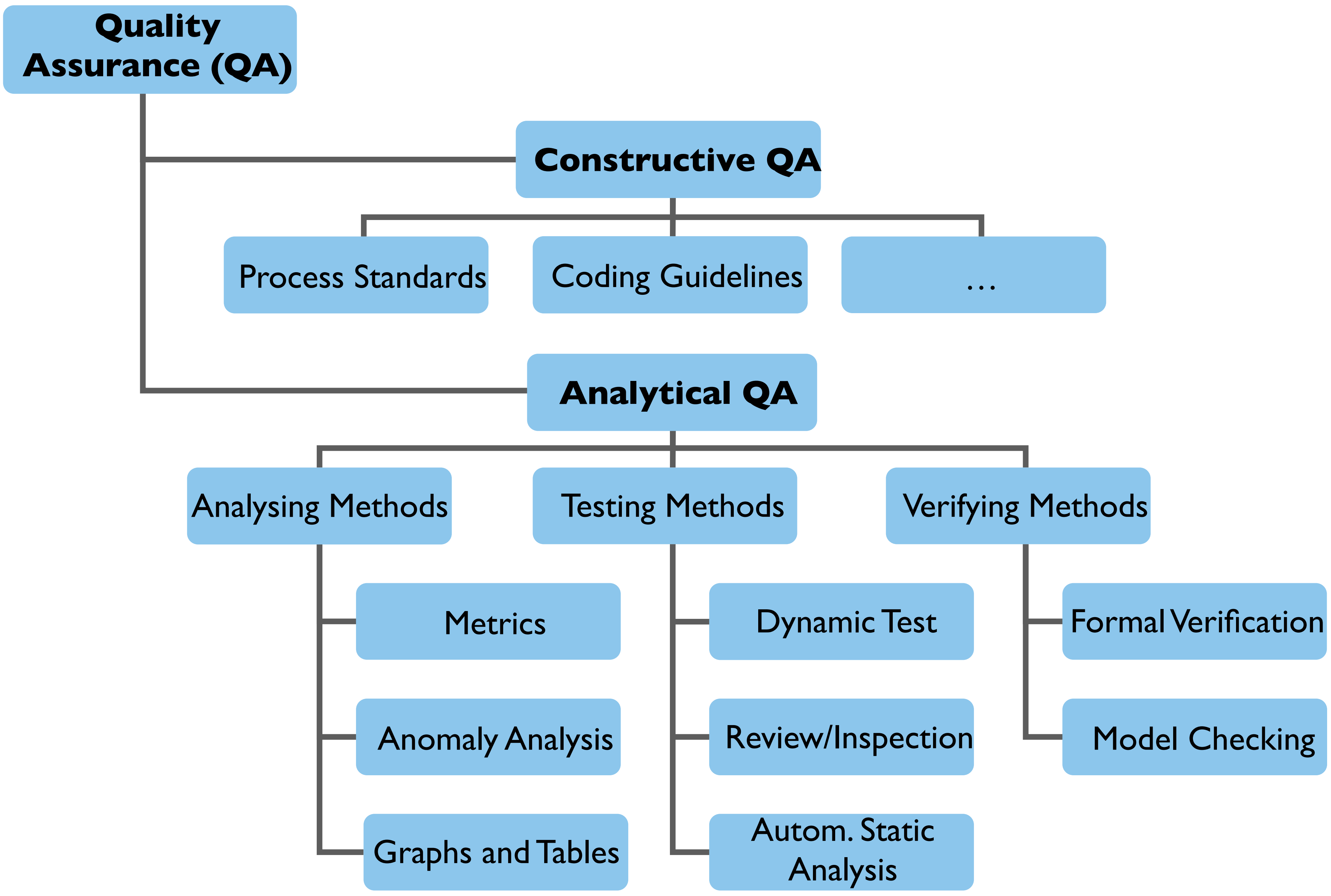
We are still in the part "Product Quality".

Code analysis

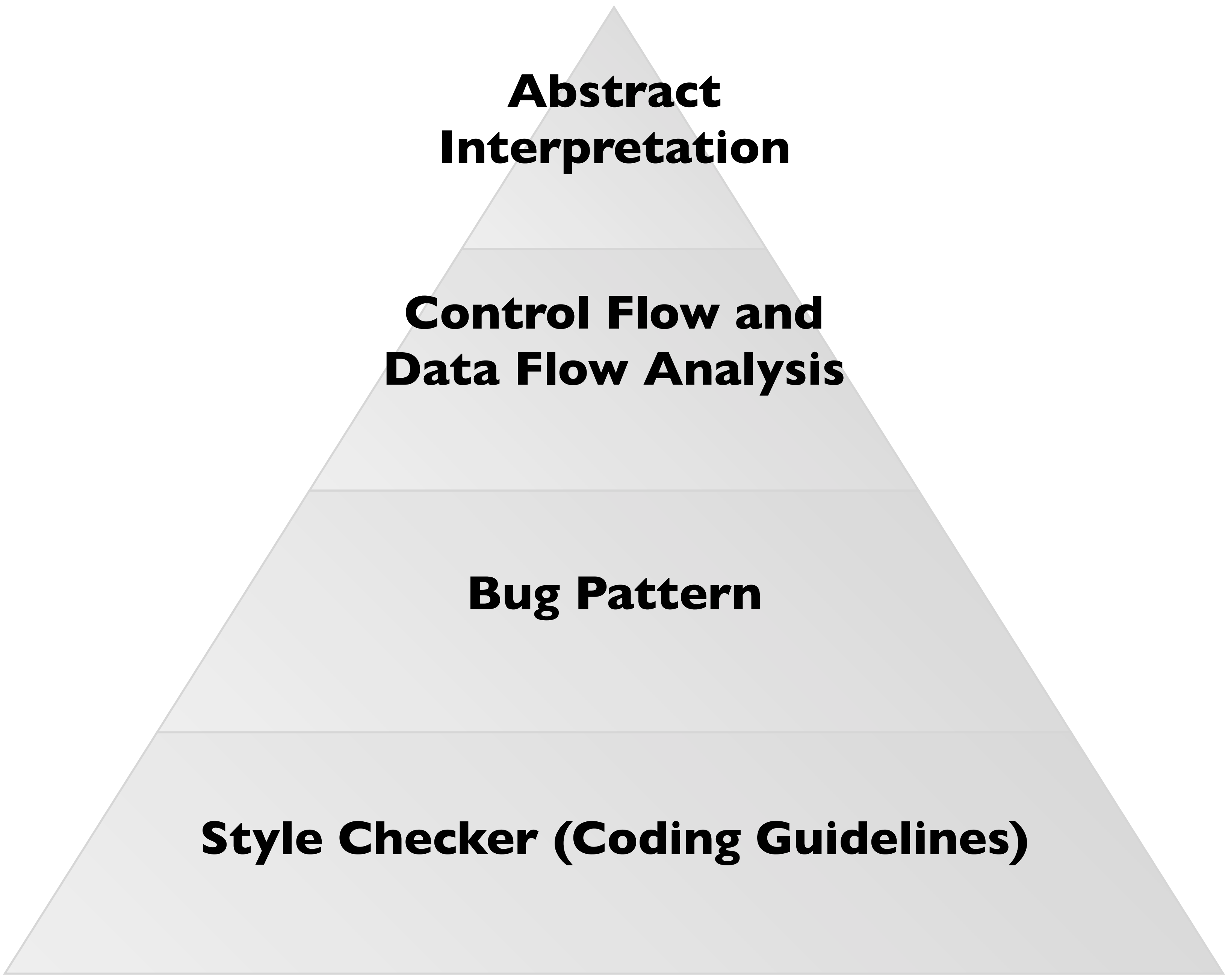
Quality evaluation

This lecture covers code analysis, mostly bug pattern tools and clone detection, as well as an approach for quality evaluation.

Code analysis



With code analysis, I mainly refer to automatic static analysis. Dynamic analyses are not explicitly covered in the following.


Abstract Interpretation
Control Flow and Data Flow Analysis
Bug Pattern
Style Checker (Coding Guidelines)

- Analysis of software by software
- But no execution of the analysed software
- Wide spectrum
- Efficient, but many false positives
- Examples
  - Checkstyle
  - FindBugs
  - PMD
  - Klockwork
  - Coverity

Bug Pattern Tools

On average 1/4 of faults during development

7 person-hours for configuration
<0.5 person-hours per fault

- Empirical studies with Cirquent and o2 Germany
- Subset of defect types of reviews
- Different defects as tests
- Problem: High level of false positives, tends to improve

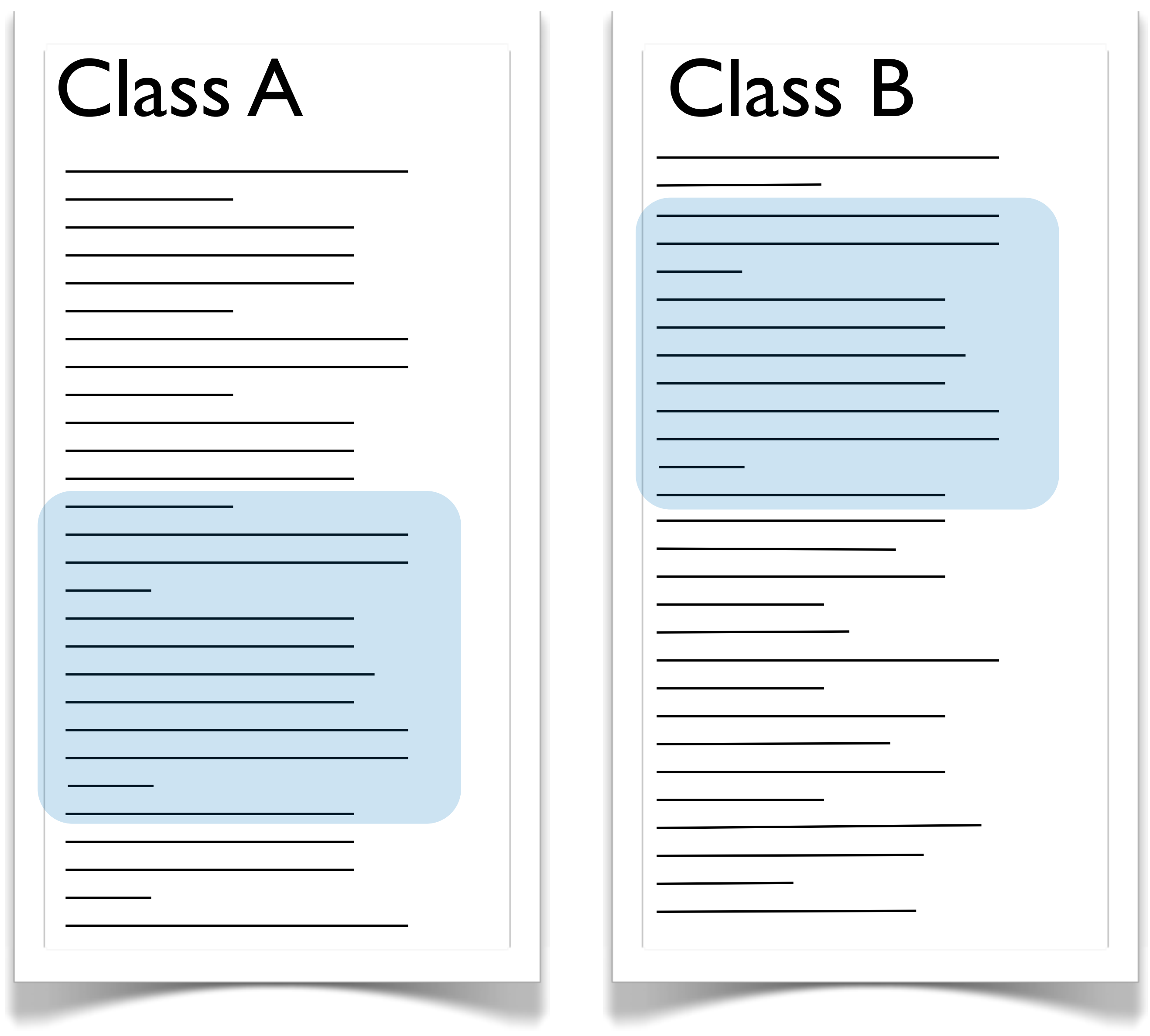
Example: FindBugs

```
String input = textField.content;
if (input == expectedString) {
...
```

```
int someMethod(int y) {...}

if (a) {someMethod(b)} ...
```

A well-known and very usable example of Java analysis tools is FindBugs from the University of Maryland.
The two code snippets are examples of bug patterns that FindBugs can detect:
String comparison with == and missing usage of return value

Code cloning



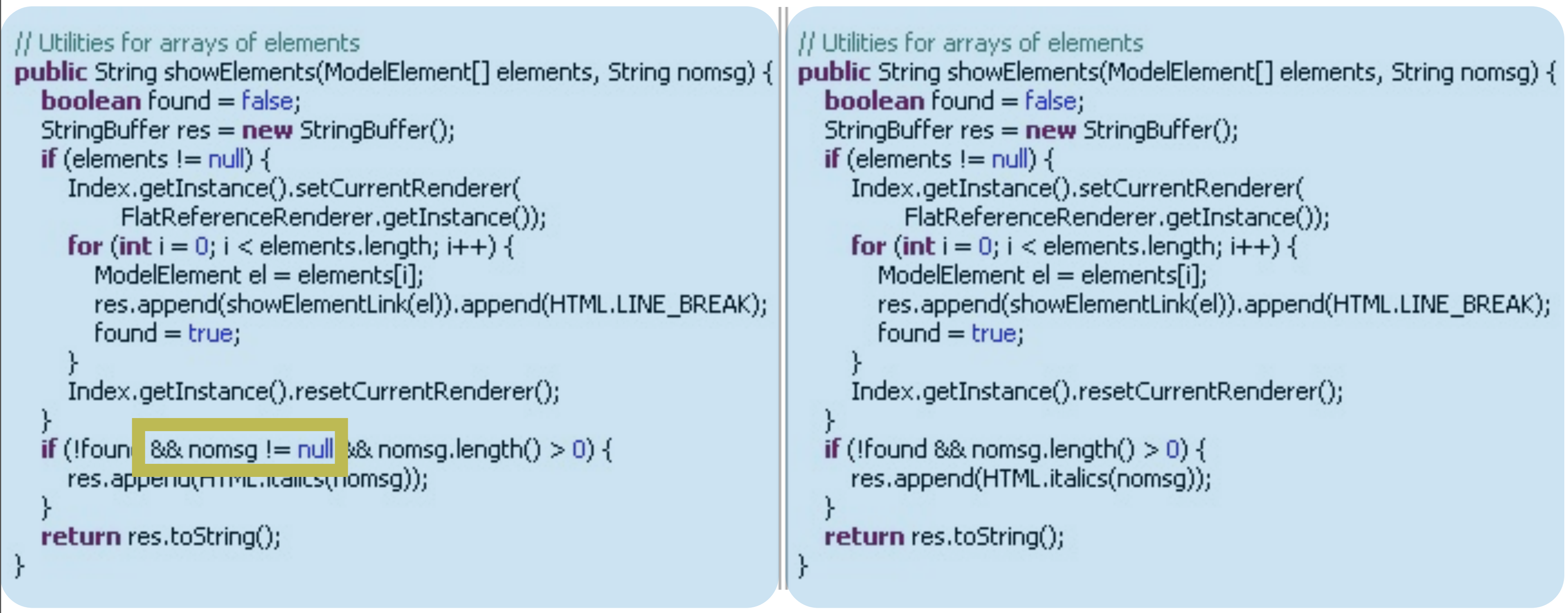
Code clones are parts of a source code that were copied and pasted. In principle, it is a normal,

exploarative approach to look for code that does something similar as what you want to implement.

You should, however, refactor common code to a method or class that is used in both places.

If this refactoring step is omitted, we have clones in the code.

Code clone example



```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
  boolean found = false;
  StringBuffer res = new StringBuffer();
  if (elements != null) {
    Index.getInstance().setCurrentRenderer(
        FlatReferenceRenderer.getInstance());
    for (int i = 0; i < elements.length; i++) {
      ModelElement el = elements[i];
      res.append(showElementLink(el)).append(HTML.LINE_BREAK);
      found = true;
    }
    Index.getInstance().resetCurrentRenderer();
  }
  if (!found && nomsg != null && nomsg.length() > 0) {
    res.append(HTML.italics(nomsg));
  }
  return res.toString();
}
```

```
// Utilities for arrays of elements
public String showElements(ModelElement[] elements, String nomsg) {
  boolean found = false;
  StringBuffer res = new StringBuffer();
  if (elements != null) {
    Index.getInstance().setCurrentRenderer(
        FlatReferenceRenderer.getInstance());
    for (int i = 0; i < elements.length; i++) {
      ModelElement el = elements[i];
      res.append(showElementLink(el)).append(HTML.LINE_BREAK);
      found = true;
    }
    Index.getInstance().resetCurrentRenderer();
  }
  if (!found && nomsg.length() > 0) {
    res.append(HTML.italics(nomsg));
  }
  return res.toString();
}
```

Clone

- Sequence of normalized statements
- At least one other occurrence in the code

Exact clone

- Edit distance between clones = 0

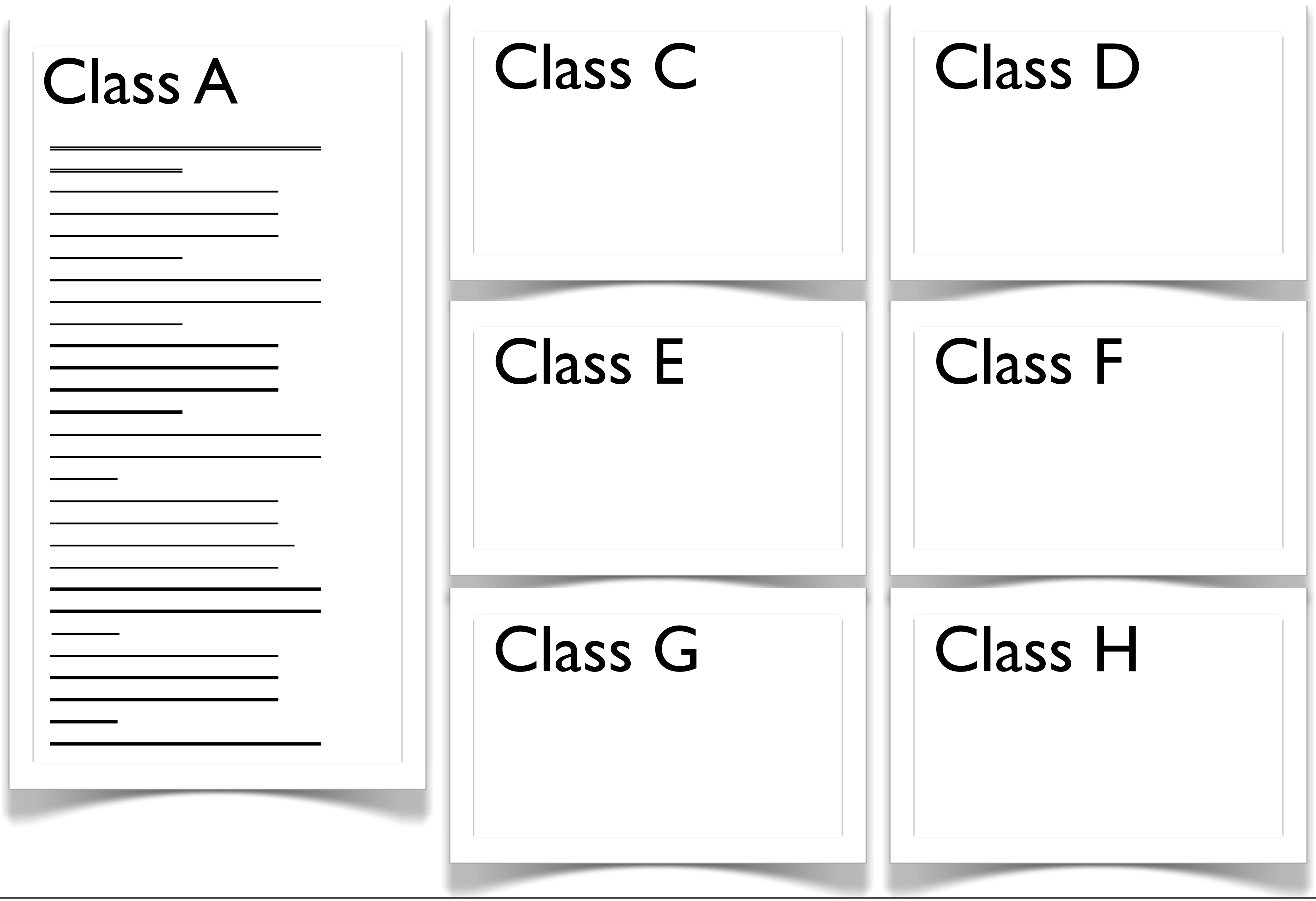
Inconsistent clone

- Edit distance between clones > 0 & below given threshold

(Inconsistent) Clone Group

- Set of clones at different positions (with at least 1 inconsistent clone)
- Semantic relationship between clones

Unnecessary size increase



The main and most direct problem of code clones is the unnecessary size increase.
With many clone groups, which consist of several clones, the code is significantly larger than it
needs to be.
This additional, unnecessary code needs to be read, comprehended, changed, and tested.

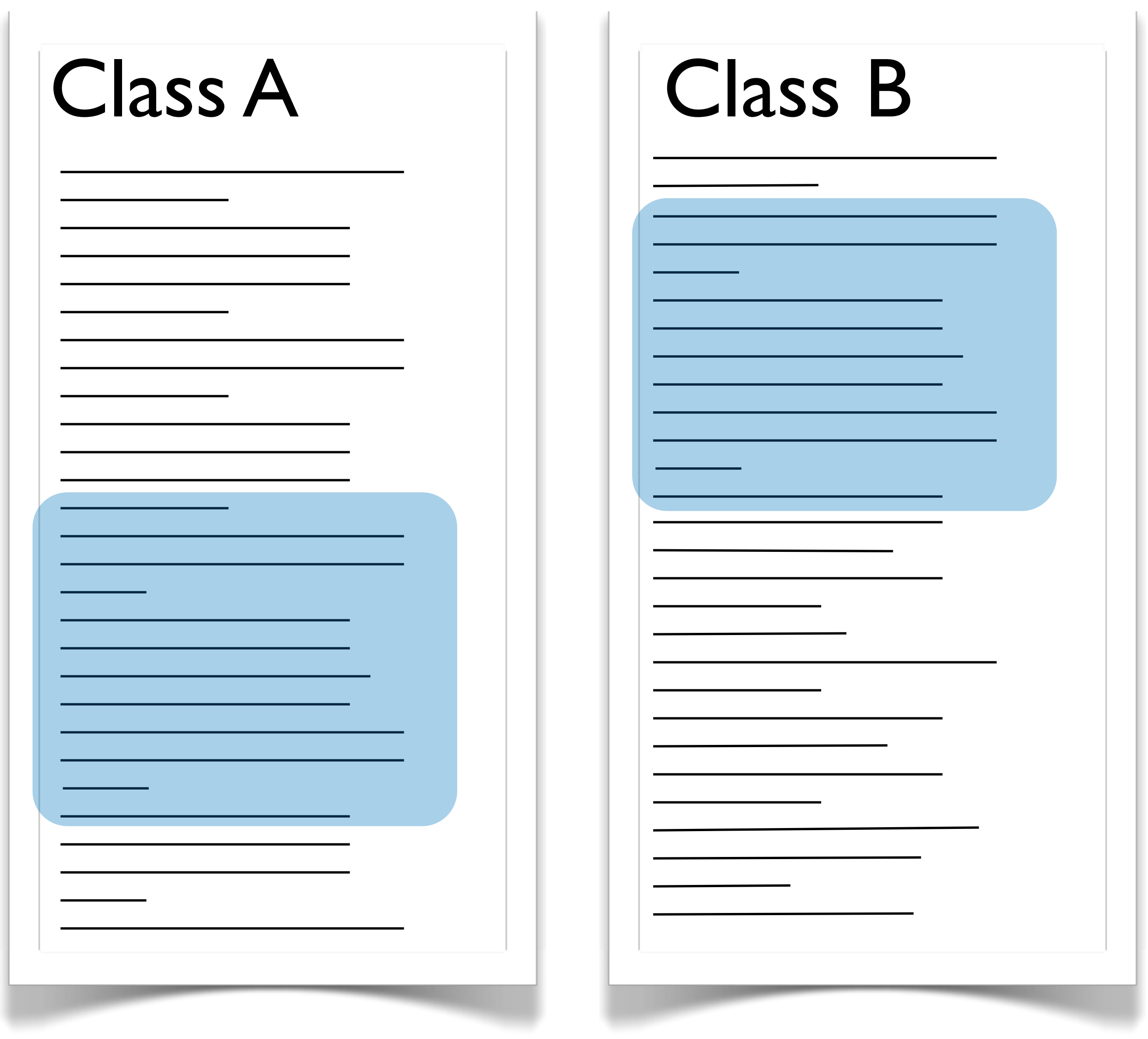
Example: Inconsistent clones

Furthermore, clones can become inconsistent if one of the clones is changed but not the others.

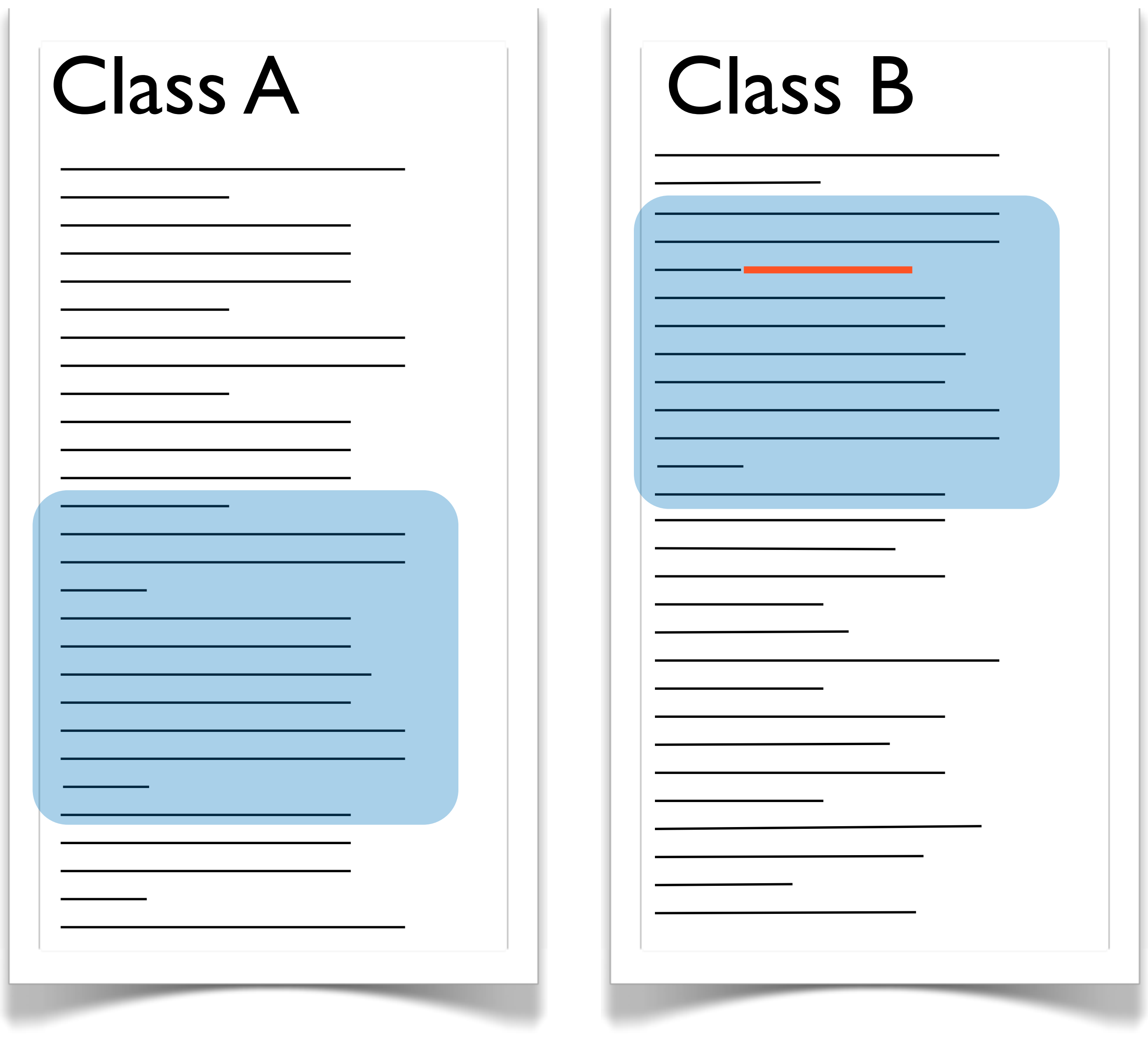
This can be intended if one clone has to conform to different requirements. If this was not intended,

however, it might be a real defect in the system.

We analysed this in more detail with systems from Munich Re, LV 1871, and TUM.



These are two clones of a clone group.

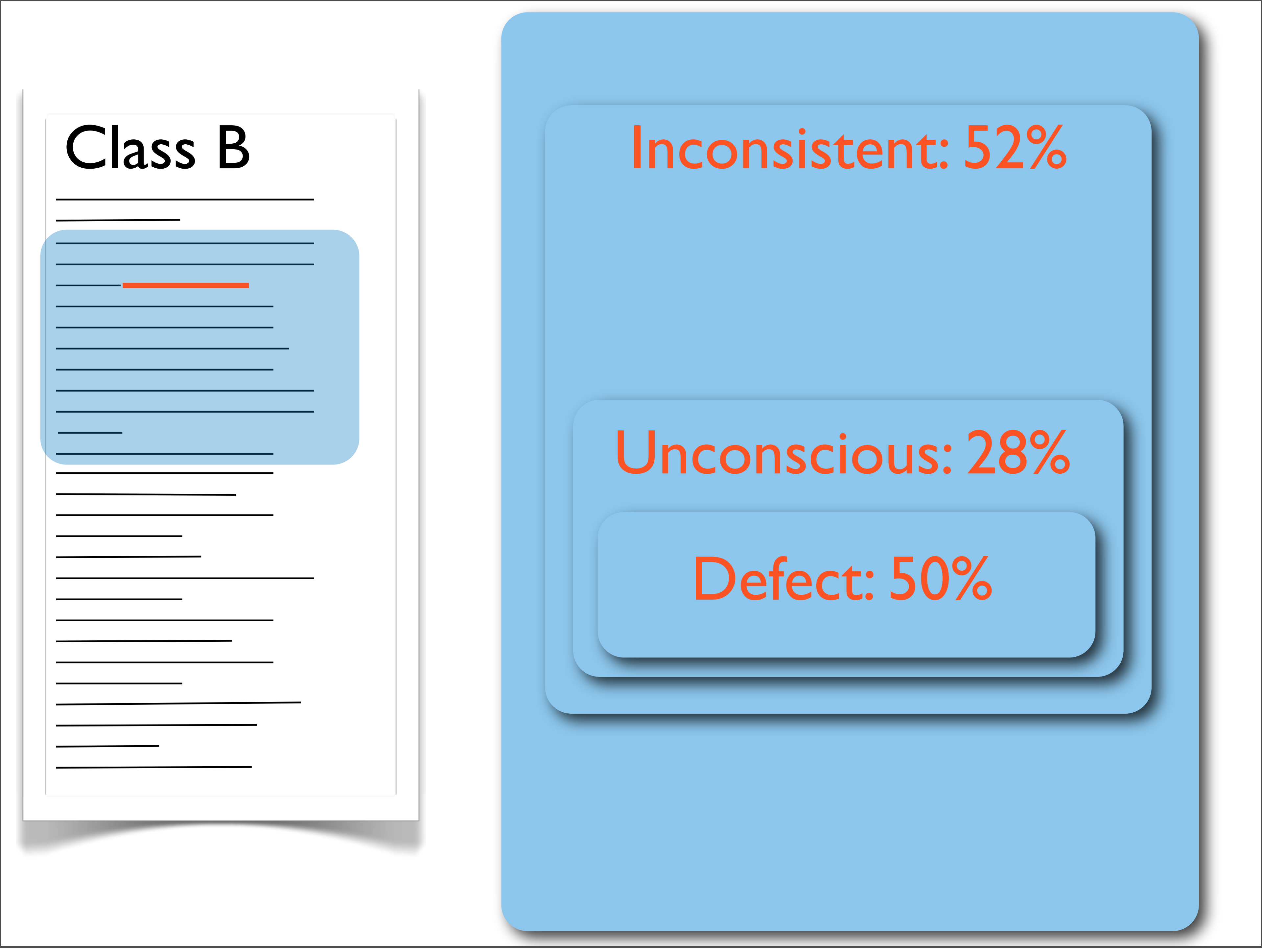


Inconsistent?

Unconscious?

Defect?

One of these clones was changed. In the study we then asked the developers:
Was the change done conscious of the clone?
Does the inconsistency constitute a clone?



Of all clones we found in the analysed systems, 52% contained inconsistencies.
Of these inconsistent clones, 28% were changed unconscious of the copies.
Of the inconsistent clones that were changed unconscious of the copies, 50% were real defects.

Every second unconsciously inconsistent change constitutes a fault.

Jürgens et al., ICSE'09, 2009

Group work

Group I: How do the types of defects found with automatic static analysis compare to defects found by tests or reviews?

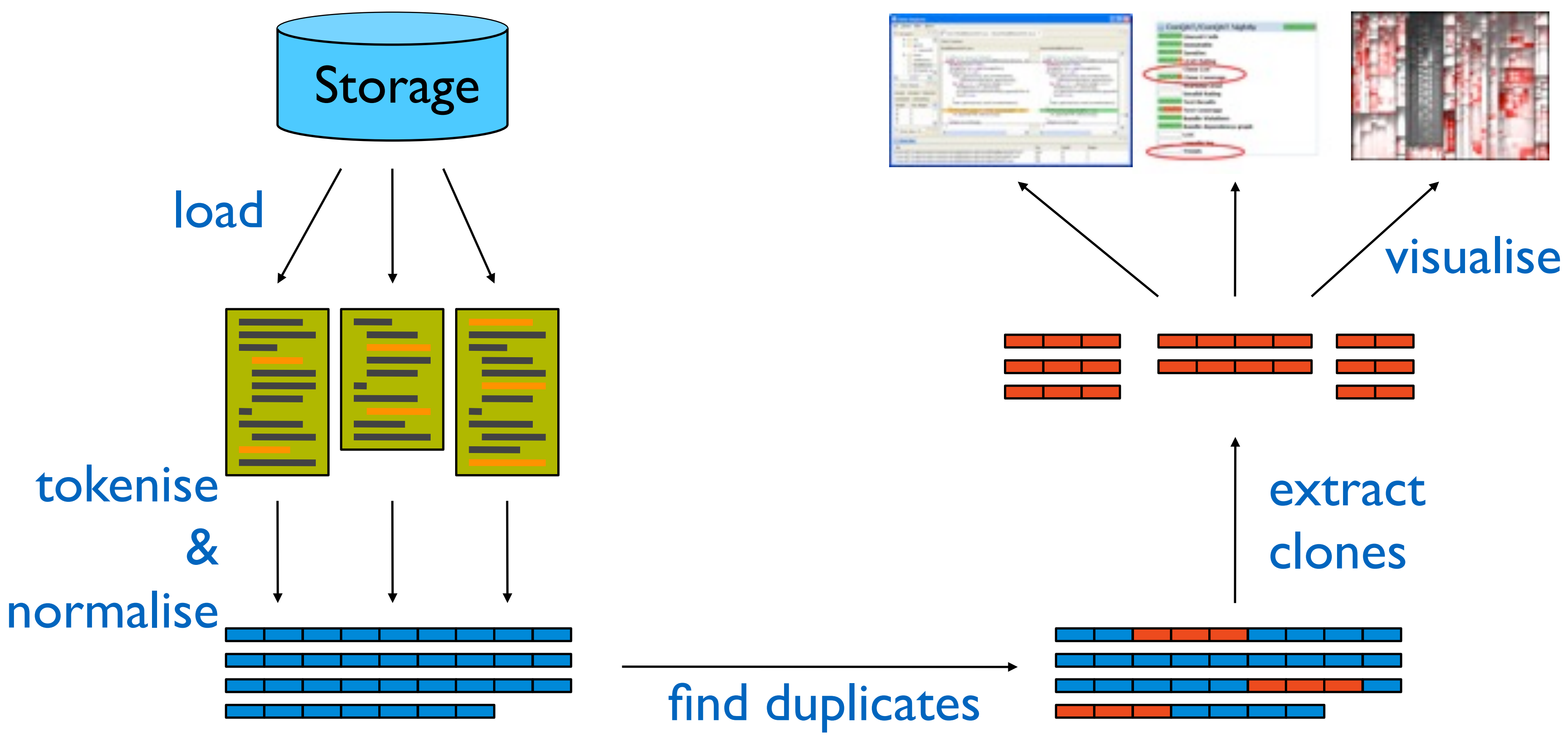
Group II: When in the development process should automatic static analysis be applied (e.g., before or after which other methods, how often)?

10 minutes, cards
Short presentation

Static analysis finds largely different defects than tests, but many defects also found in a review.
If there is a good tool that finds a specific type of defect it often finds it more thoroughly than a review.
Static analysis is mostly suitable for finding readability problems.

Static analysis should be applied as often as possible, e.g., in the nightly build or better directly in
the IDE of the developer. It is a good entry criteria for a code inspection to save the inspectors the
time for noting defects that a tool could detect cheaper.

Clone detection: Processing steps



First, the code is loaded from the storage system.

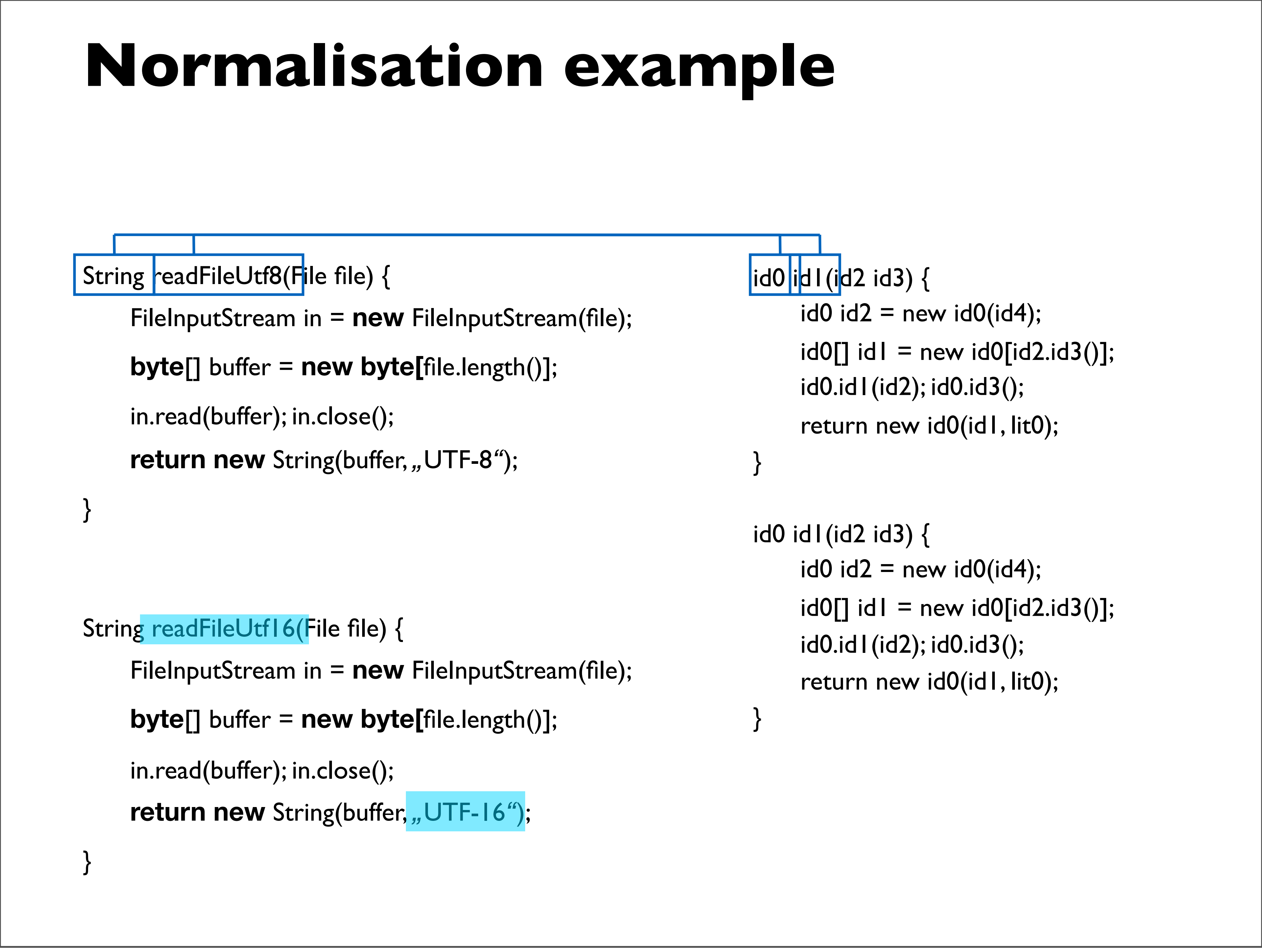
Second, the code separated into tokens and these tokens are normalised, e.g., identifier names are

removed.

Third, in the normalised tokens, duplicates are detected.

Fourth, the duplicates that constitute clones are extracted.

Fifth, the extracted clones are suitably visualised.



Normalisation example

```
String readFileUtf8(File file) {
       FileInputStream in = new FileInputStream(file);
       byte[] buffer = new byte[file.length()];
       in.read(buffer); in.close();
       return new String(buffer, „UTF-8“);
}


String readFileUtf16(File file) {
       FileInputStream in = new FileInputStream(file);
       byte[] buffer = new byte[file.length()];
       in.read(buffer); in.close();
       return new String(buffer, „UTF-16“);
}
```

```
id0 id1(id2 id3) {
       id0 id2 = new id0(id4);
       id0[] id1 = new id0[id2.id3()];
       id0.id1(id2); id0.id3();
       return new id0(id1, lit0);
}

id0 id1(id2 id3) {
       id0 id2 = new id0(id4);
       id0[] id1 = new id0[id2.id3()];
       id0.id1(id2); id0.id3();
       return new id0(id1, lit0);
}
```

This examples shows why normalisation is necessary.

Here the method that reads a UTF-8 file was copied and changed so that it reads UTF-16 files.

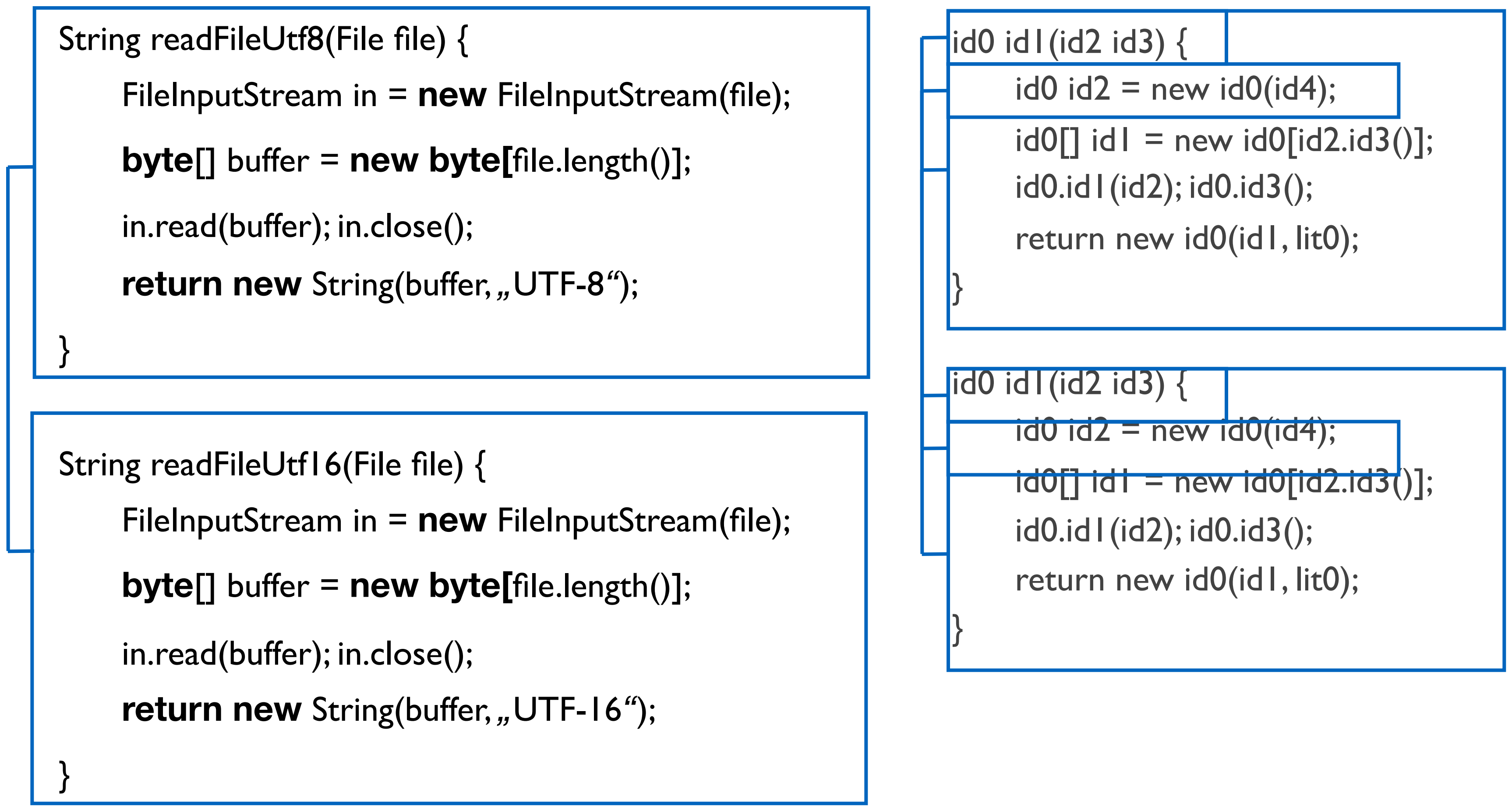
In essence, this still is a copy and if there is a change in readFileUtf8, there is a high probability

that readFileUtf16 also has to be changed.

Hence, identifiers and literals are normalised to "id" and "lit" so that the duplication finder

is still able to find them.

Normalisation example



```
String readFileUtf8(File file) {
    FileInputStream in = new FileInputStream(file);
    byte[] buffer = new byte[file.length()];
    in.read(buffer); in.close();
    return new String(buffer, „UTF-8“);
}
```

```
String readFileUtf16(File file) {
    FileInputStream in = new FileInputStream(file);
    byte[] buffer = new byte[file.length()];
    in.read(buffer); in.close();
    return new String(buffer, „UTF-16“);
}
```

```
id0 id1(id2 id3) {
    id0 id2 = new id0(id4);
    id0[] id1 = new id0[id2.id3()];
    id0.id1(id2); id0.id3();
    return new id0(id1, lit0);
}
```

```
id0 id1(id2 id3) {
    id0 id2 = new id0(id4);
    id0[] id1 = new id0[id2.id3()];
    id0.id1(id2); id0.id3();
    return new id0(id1, lit0);
}
```

Clones contain similar but not necessarily identical code

Measures for cloning

- Number of clone groups/clone instances
- Size of largest clone/cardinality of most frequent clone
- Cloned Statements
  - Number of statements in the system being part of at least one clone
- Clone Coverage
  - #Cloned Statements / #Statements
  - Probability of a randomly chosen statement to be part of a clone
- Redundancy Free Source Statements (RFSS)
  - Size of system after (hypothetical) perfect clone removal

Different measures for cloning are interesting for different goals.

The number of clone groups and instances shows how many refactorings would be necessary.

The size of the largest clone or the cardinality of the most frequent clone show the hot spots.

Clone coverage gives a feeling how big the problem is over the whole system.

RFSS shows the actual size of the system.

Measures example

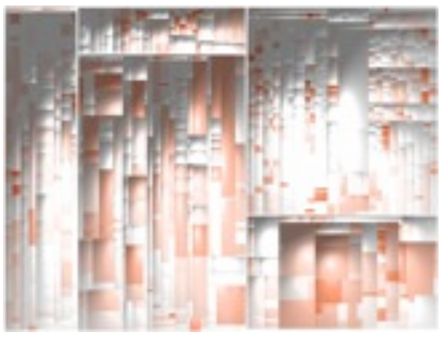
```
class Test {
    int doX (int a, int b) {
        if (a > b) {
            return 2*a; }
        return 2*b;
    }

    int doY (int a, int b) {
        return a+b;
    }

    int doZ (int c, int d) {
        if (c > d) {
            return 2*c; }
        return 2*d;
    }
}
```

- Statements: 11
- Cloned Statements: 8
- Clone Coverage: 8/11 ≈ 70%
- RFSS: 7

Visualisation of clone detection results

Compare View (~20 LOC)

Tree Maps (>1.000.000 LOC)

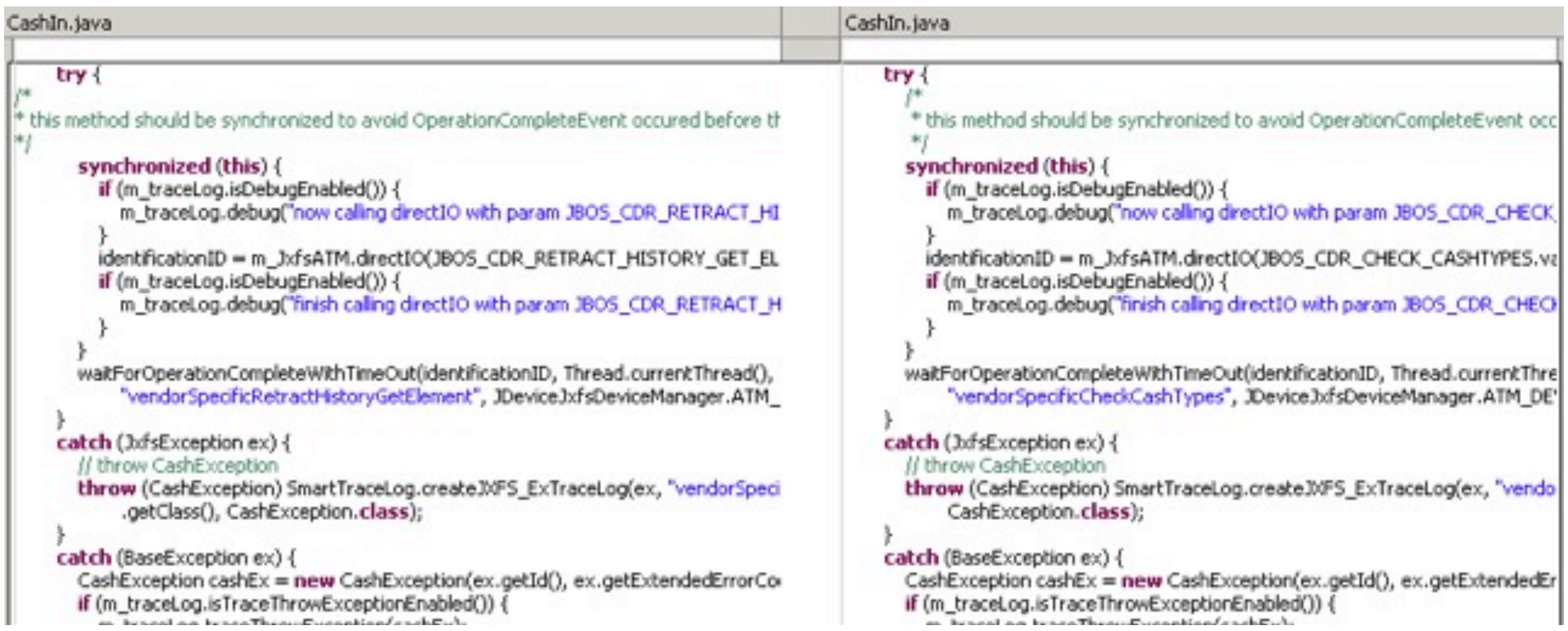


Trends over Time

Seesoft View (~400 LOC)

Depending on the the purpose of the analysis and the size of the part I want to look at,
different visualisations are appropriate.

Clone compare view



CashIn.java

```
    try {
/*
* this method should be synchronized to avoid OperationCompleteEvent occured before th
*/
        synchronized (this) {
            if (m_traceLog.isDebugEnabled()) {
                m_traceLog.debug("now calling directIO with param JBOS_CDR_RETRACT_HI
            }
            identificationID = m_JxfsATM.directIO(JBOS_CDR_RETRACT_HISTORY_GET_EL
            if (m_traceLog.isDebugEnabled()) {
                m_traceLog.debug("finish calling directIO with param JBOS_CDR_RETRACT_H
            }
        }
        waitForOperationCompleteWithTimeOut(identificationID, Thread.currentThread(),
                "vendorSpecificRetractHistoryGetElement", JDeviceJxfsDeviceManager.ATM_
    }
    catch (JxfsException ex) {
        // throw CashException
        throw (CashException) SmartTraceLog.createJXFS_ExTraceLog(ex, "vendorSpeci
                .getClass(), CashException.class);
    }
    catch (BaseException ex) {
        CashException cashEx = new CashException(ex.getId(), ex.getExtendedErrorCo
        if (m_traceLog.isTraceThrowExceptionEnabled()) {
```

CashIn.java

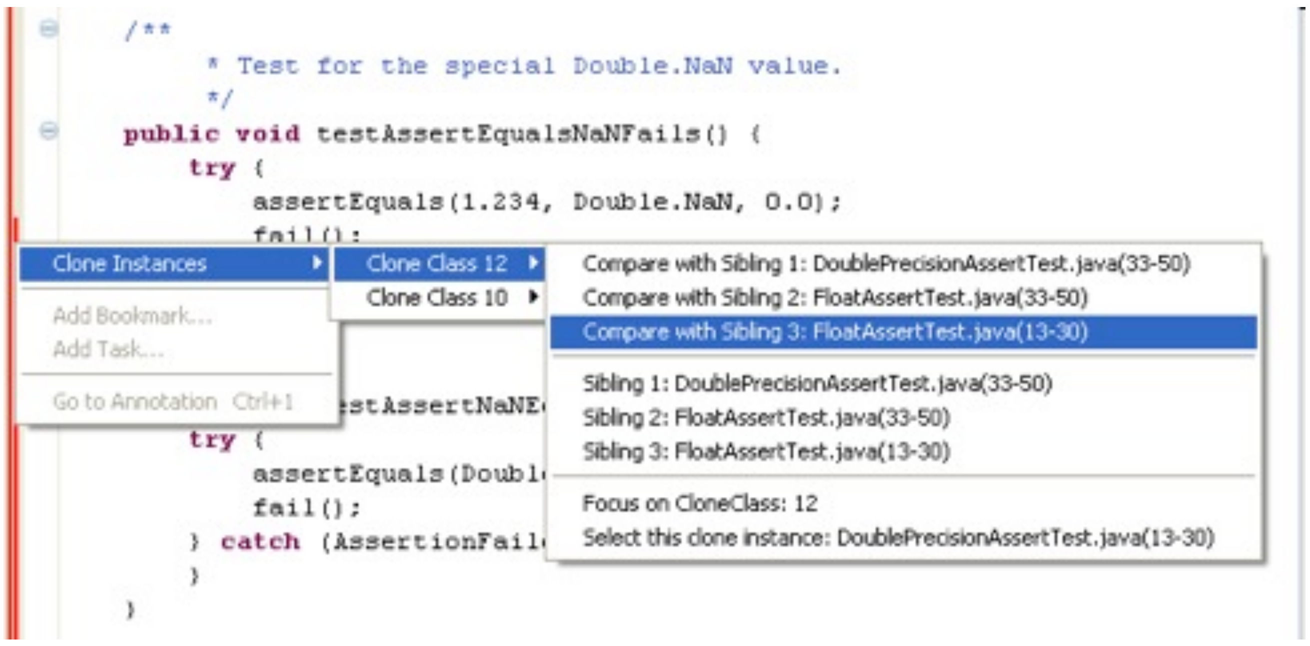
```
    try {
        /*
         * this method should be synchronized to avoid OperationCompleteEvent occ
         */
        synchronized (this) {
            if (m_traceLog.isDebugEnabled()) {
                m_traceLog.debug("now calling directIO with param JBOS_CDR_CHECK_
            }
            identificationID = m_JxfsATM.directIO(JBOS_CDR_CHECK_CASHTYPES.va
            if (m_traceLog.isDebugEnabled()) {
                m_traceLog.debug("finish calling directIO with param JBOS_CDR_CHECK
            }
        }
        waitForOperationCompleteWithTimeOut(identificationID, Thread.currentThre
                "vendorSpecificCheckCashTypes", JDeviceJxfsDeviceManager.ATM_DE
    }
    catch (JxfsException ex) {
        // throw CashException
        throw (CashException) SmartTraceLog.createJXFS_ExTraceLog(ex, "vendo
                CashException.class);
    }
    catch (BaseException ex) {
        CashException cashEx = new CashException(ex.getId(), ex.getExtendedEr
        if (m_traceLog.isTraceThrowExceptionEnabled()) {
```

The clone compare view, here in Eclipse, shows the cloned parts of two classes next to each other.

It allows a detailed inspection of the the clones.

Clone bars

- Displays cloning information in the IDE
- Helps when working with cloned code



```
/**
 * Test for the special Double.NaN value.
 */
public void testAssertEqualsNaNFails() {
    try {
        assertEquals(1.234, Double.NaN, 0.0);
        fail();
...
    stAssertNaNE
    try {
        assertEquals(Doubl
        fail();
    } catch (AssertionFail
    }
}
```

Clone Instances
Add Bookmark...
Add Task...
Go to Annotation Ctrl+1
Clone Class 12
Clone Class 10
Compare with Sibling 1: DoublePrecisionAssertTest.java(33-50)
Compare with Sibling 2: FloatAssertTest.java(33-50)
Compare with Sibling 3: FloatAssertTest.java(13-30)
Sibling 1: DoublePrecisionAssertTest.java(33-50)
Sibling 2: FloatAssertTest.java(33-50)
Sibling 3: FloatAssertTest.java(13-30)
Focus on CloneClass: 12
Select this clone instance: DoublePrecisionAssertTest.java(13-30)

Clone bars are also shown in an IDE, but directly on the side of the code you work with.

If you change existing code, this bar warns you that there are clones that you might have to

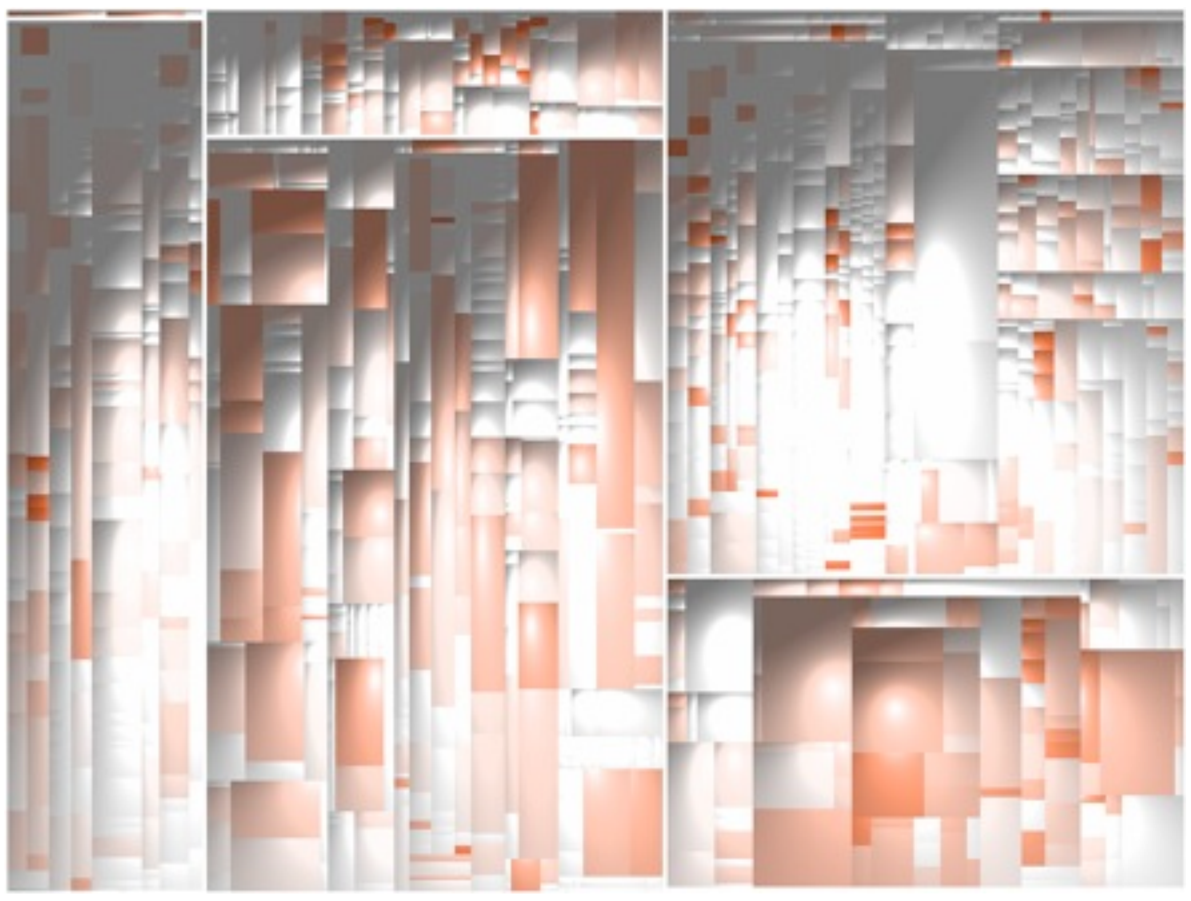
change as well.

Tree map

Visualisation of

- Structure
- Size
- Redundancy

in a single picture

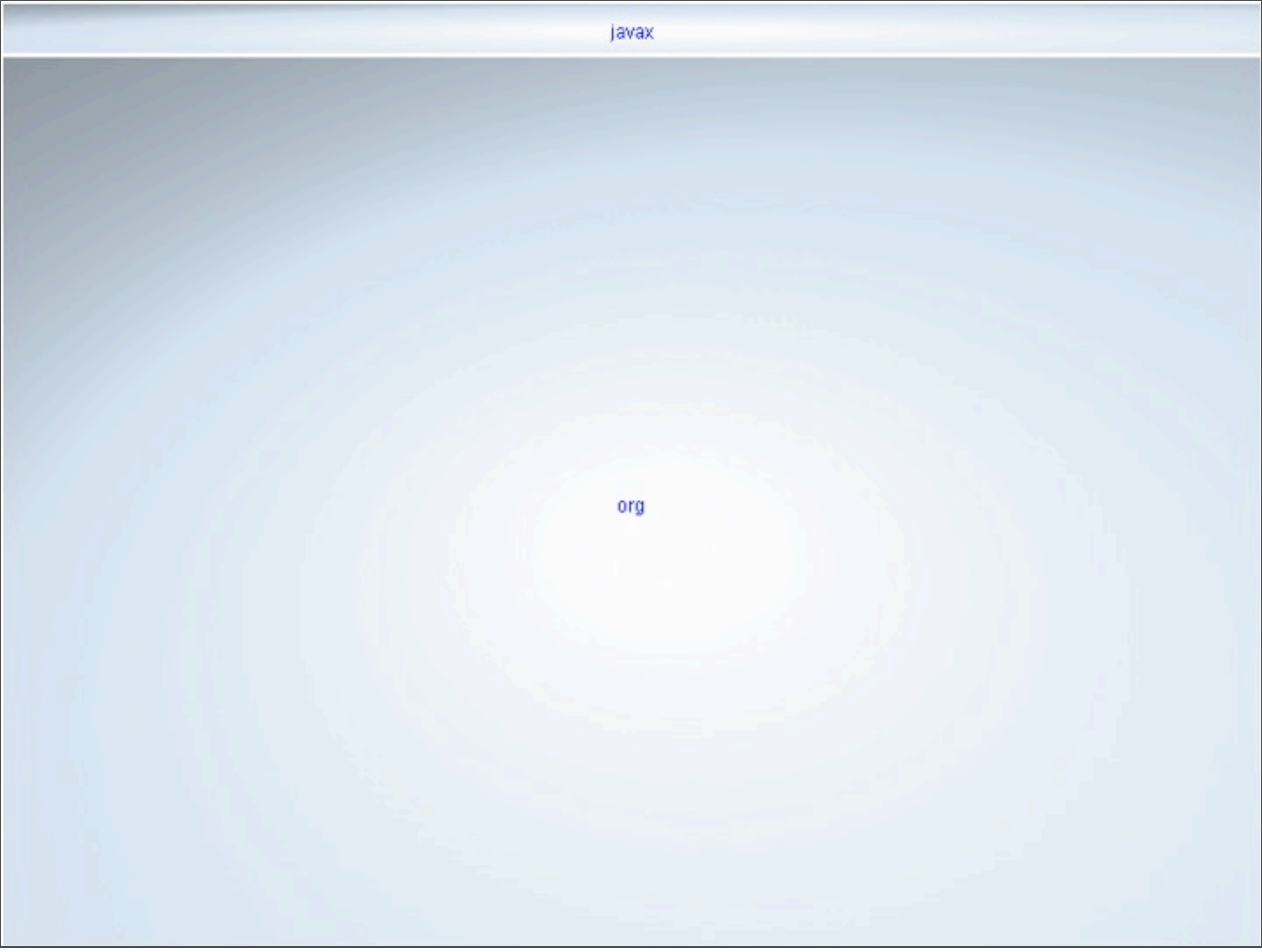


The tree map is useful to get an overview of how cloning distributes over the whole system.

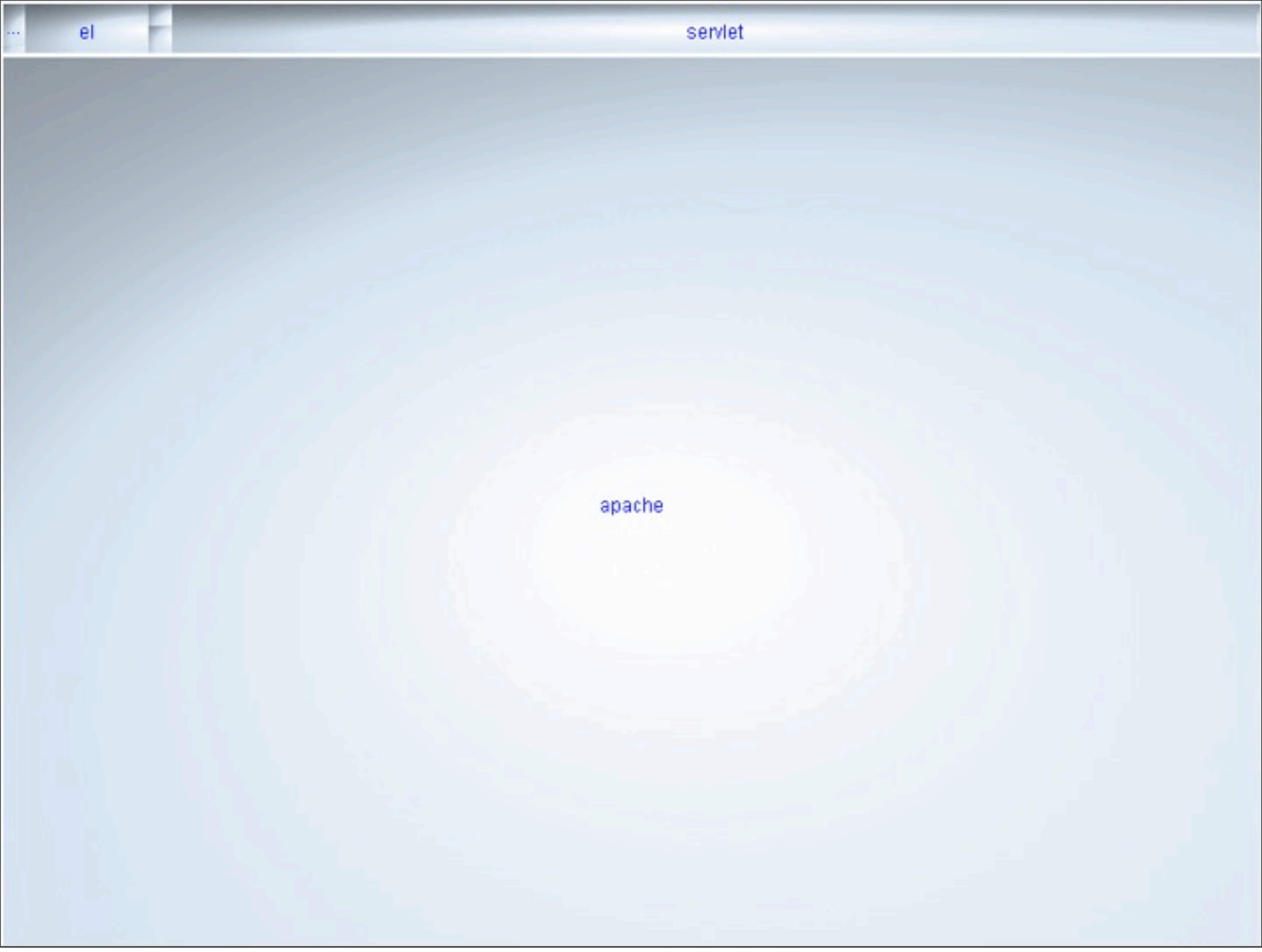
The more red a square is, the higher the clone coverage in this class.

Apache Tomcat 6.0.24

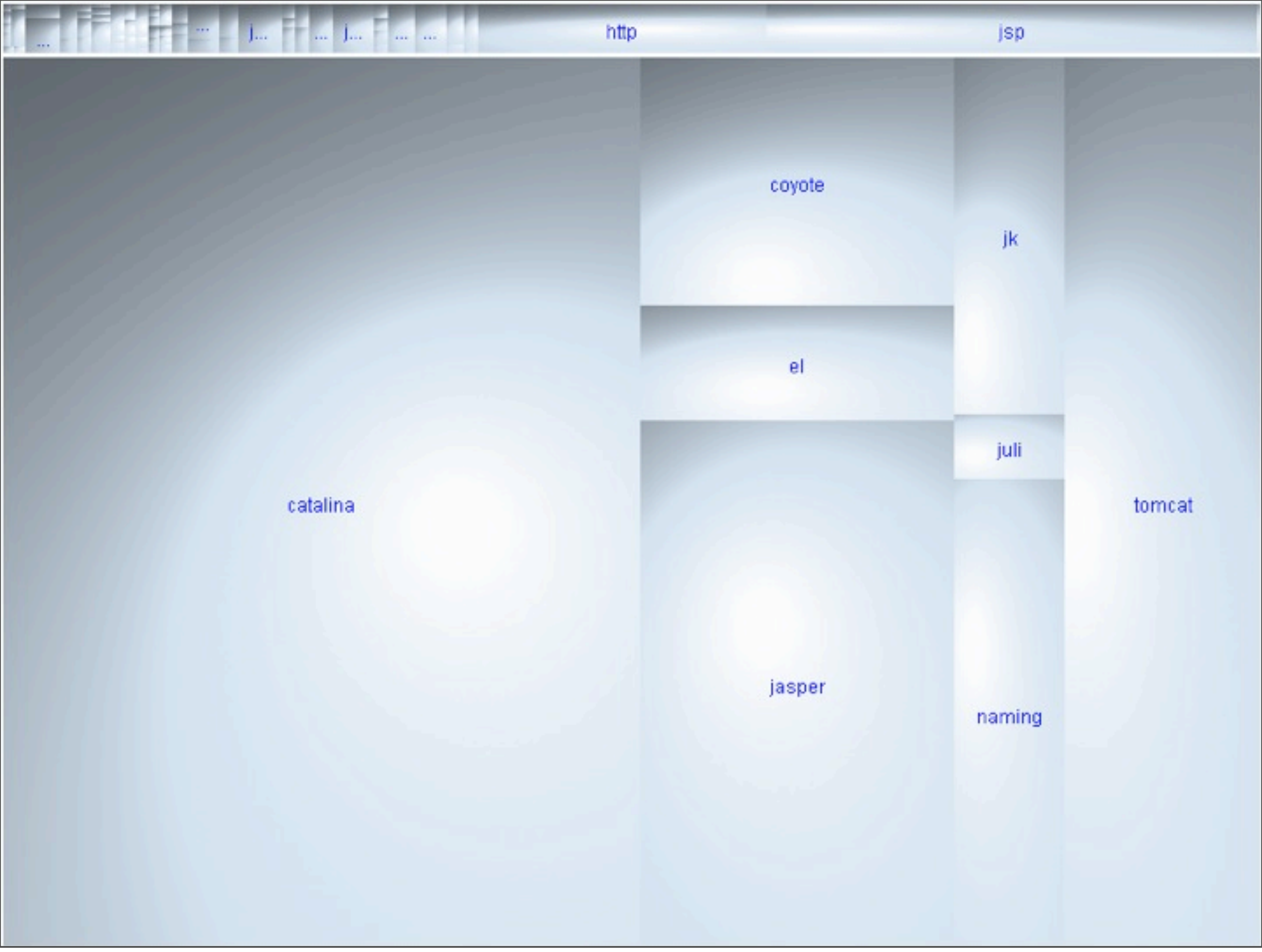
We use the source code of Apache Tomcat to show you tree map visualisation of cloning.



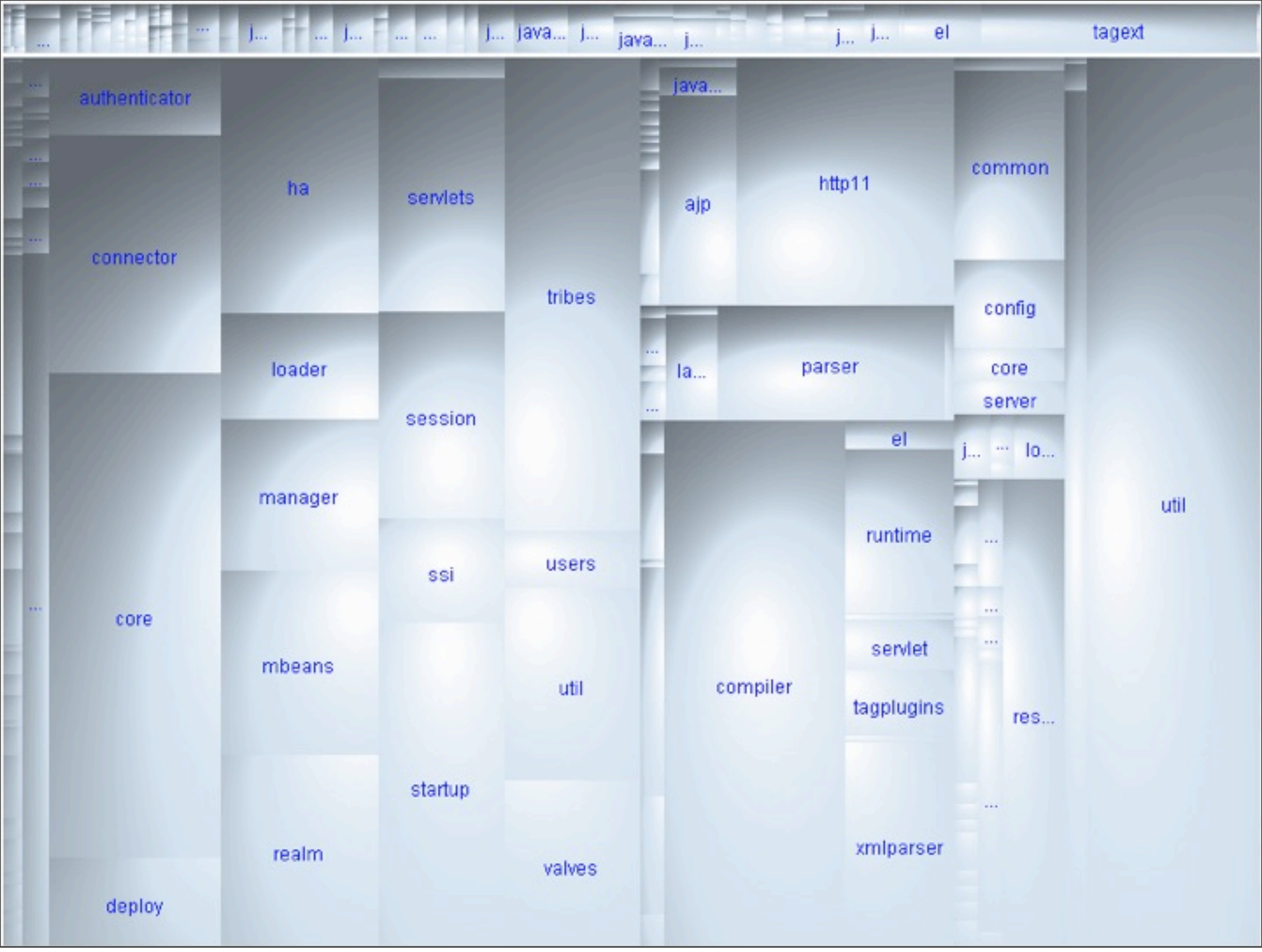
The top-level package in Tomcat is "org".



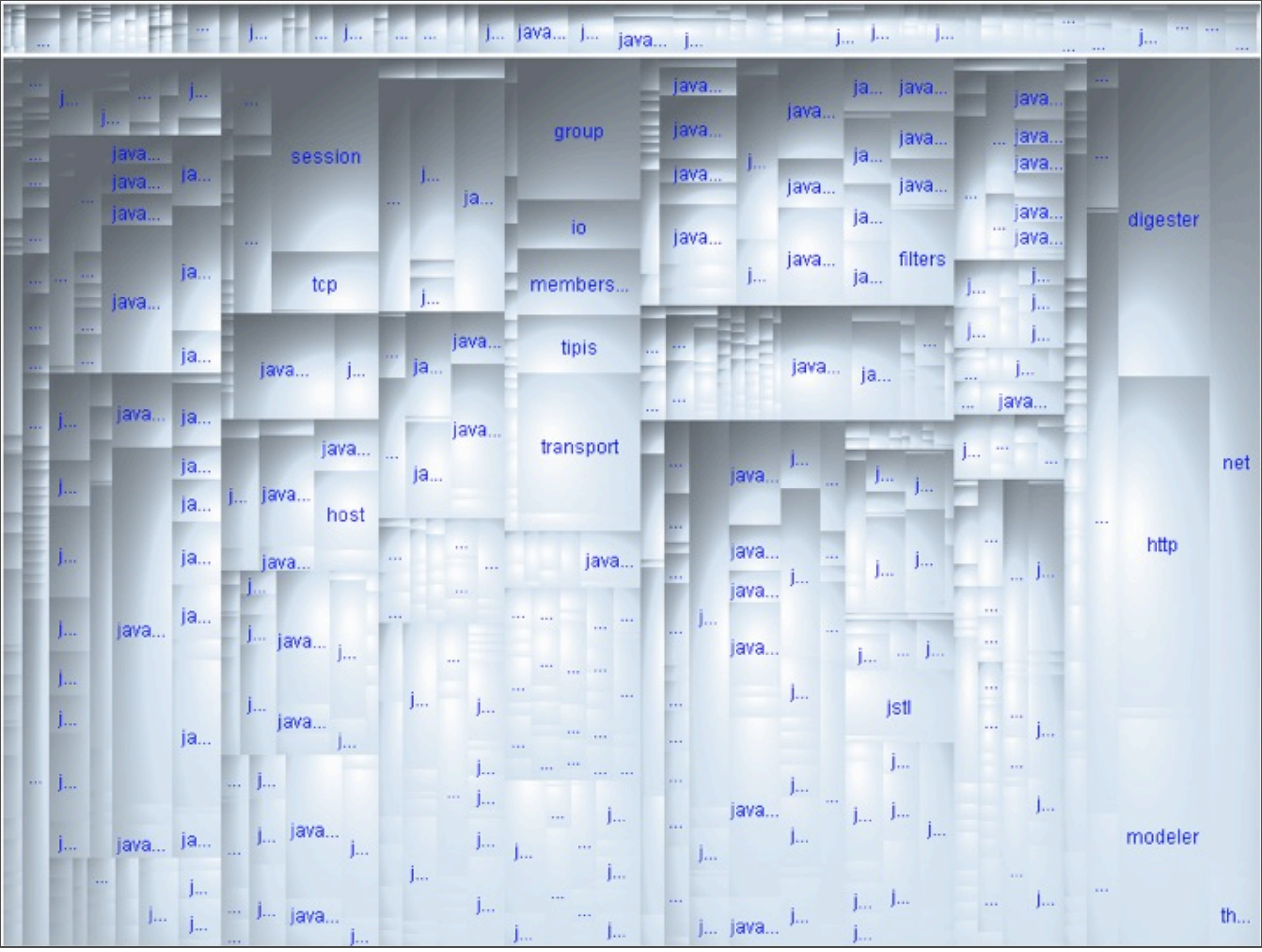
Below "org" there is the package "apache".



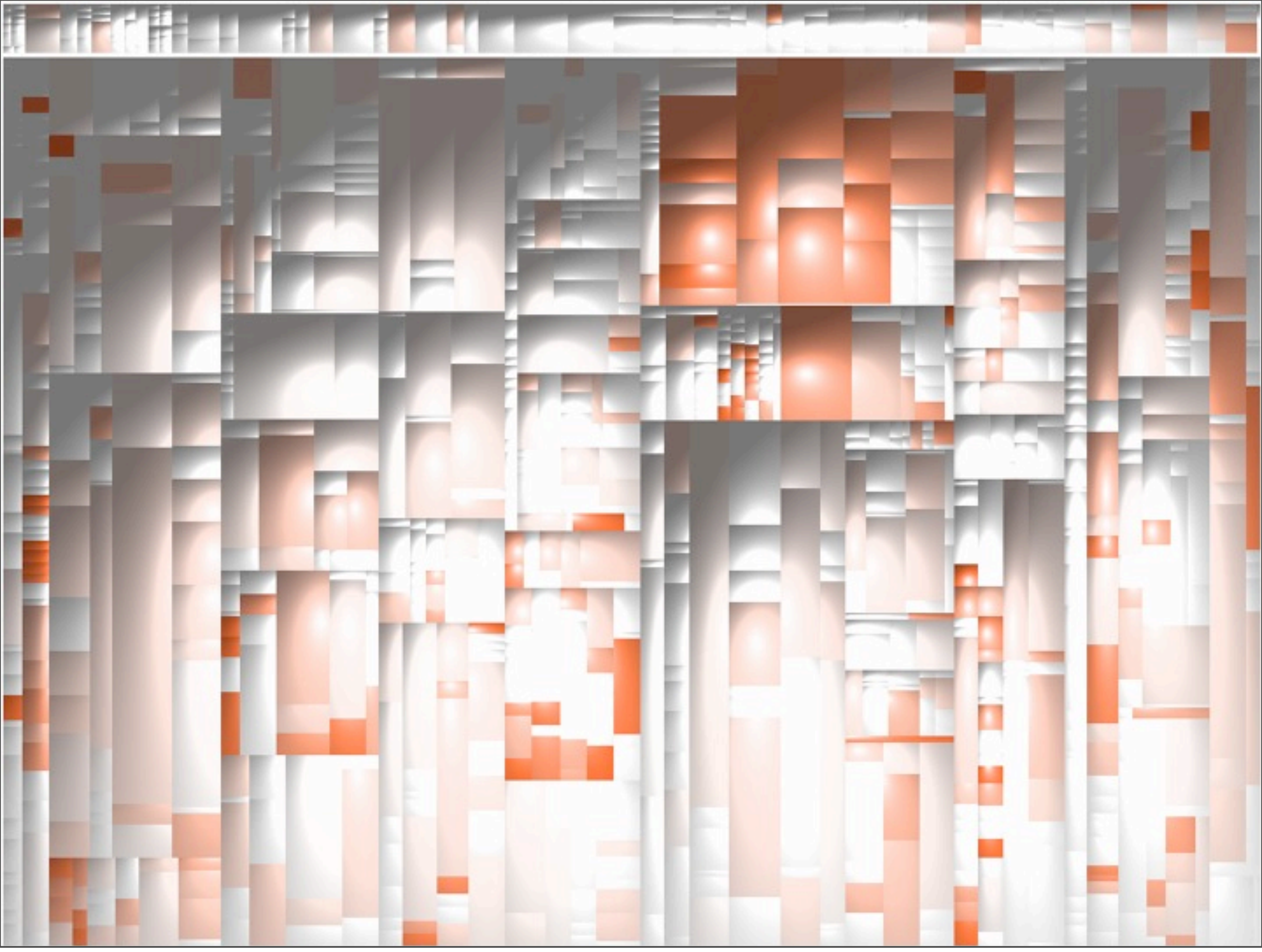
The "apache" package contains several further packages.



Which in turn contain further packages.

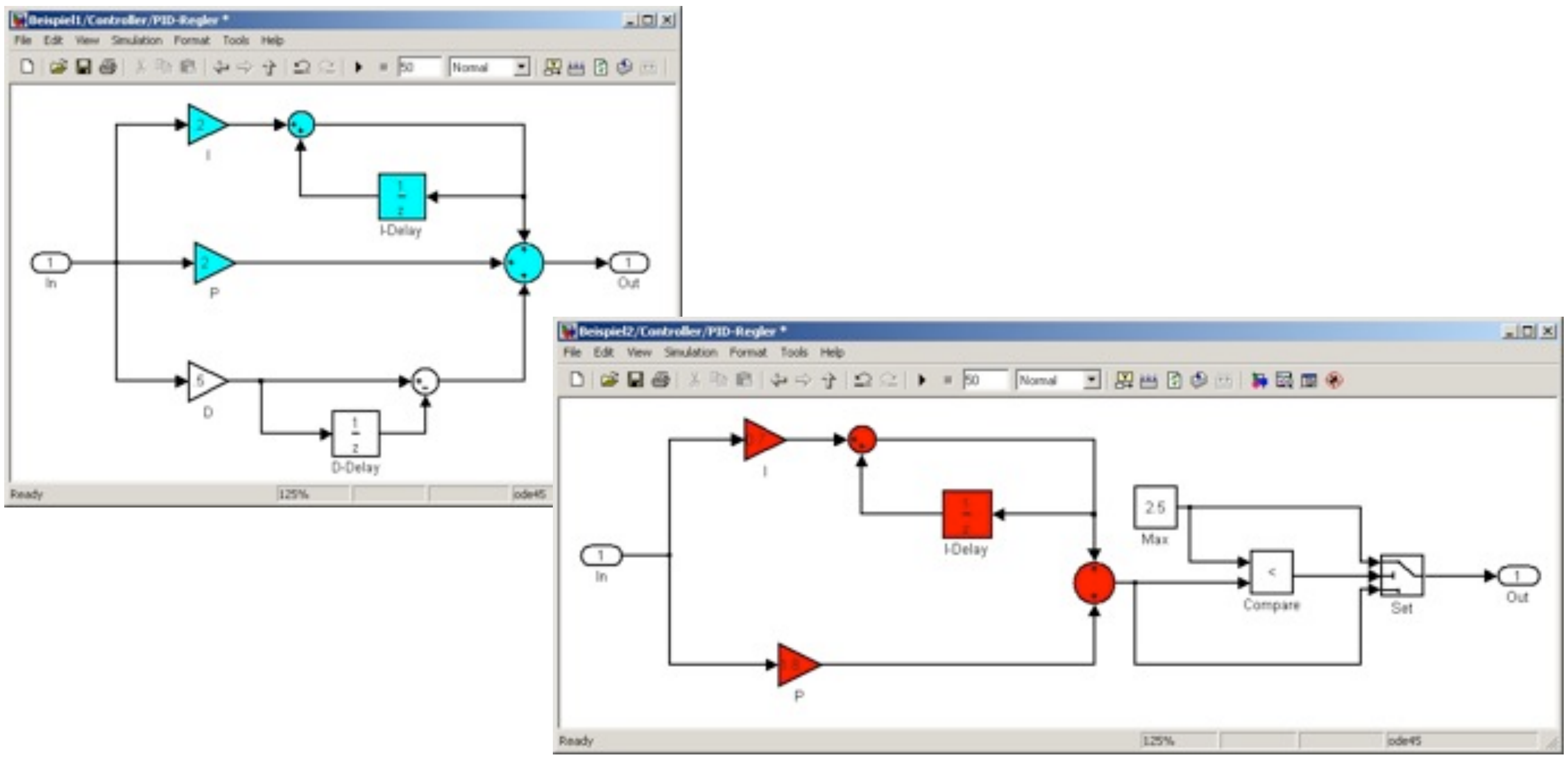


Which contain finally Java classes.



These Java classes are then overlayed with the degree of clone coverage they have.

Clones in models


Beispiel1/Controller/PID-Regler *
I
I-Delay
In
P
Out
D
D-Delay
Beispiel2/Controller/PID-Regler *
I
I-Delay
Max
Compare
Set
In
Out
P

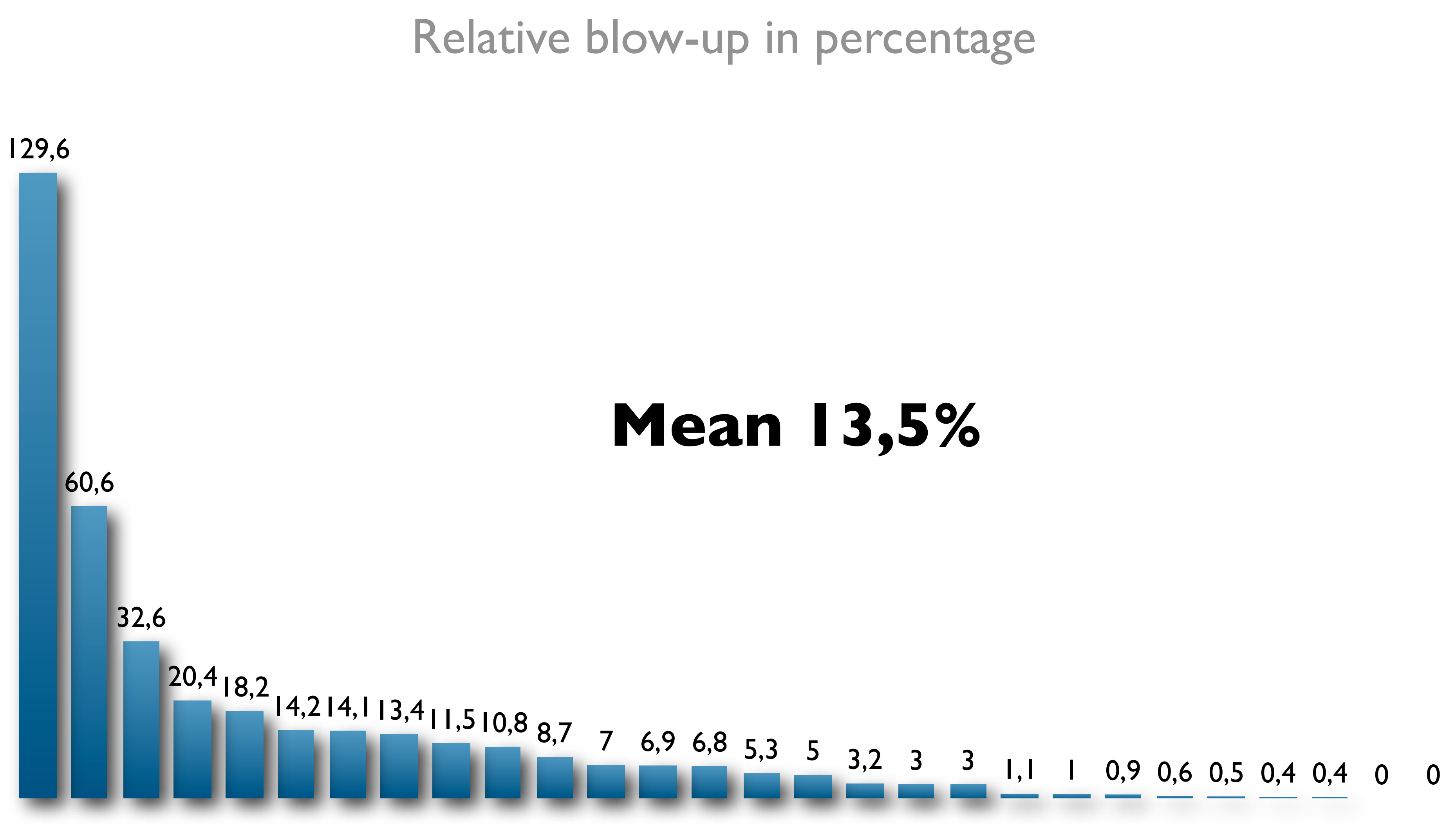
Deissenboeck et al., ICSE'08

Cloning is not only a problem in code. Also models contain copies.

We analysed this at MAN Nutzfahrzeuge and their Simulink/Targetlink models.

- Simulink/TargetLink models with about 20,000 blocks in 71 files
- Identified: 139 clone classes after filtering
- Includes clones of library blocks
- 37% of relevant blocks are part of at least one clone group
- Most clones affect several files/transcend several hierarchies

Clones in requirements specifications



Jürgens et al., ICSE'10

Also requirements specifications are affected by cloning.
The diagram shows results of a study we did with real world specifications. Each bar is the relative
blow-up for a specification. The blow-up denotes how much larger the specification is than it would
need to be without cloning.
The maximum has a blow-up of 11,000 words! But there are specifications with no or almost no
cloning.
On average, an inspection of the requirements would need more than 2 additional person days
because of cloning.

Cloning also in
models
and
requirements specifications

Code analysis

Quality evaluation

Quality evaluation

Group work

What are useful measures for quality or specific quality attributes?

10 minutes
Cards
Short presentation

The results of this group work will be reused in later lectures.

Example:
Average maintenance effort

Quality evaluations will be discussed using the example of maintenance efforts. The average

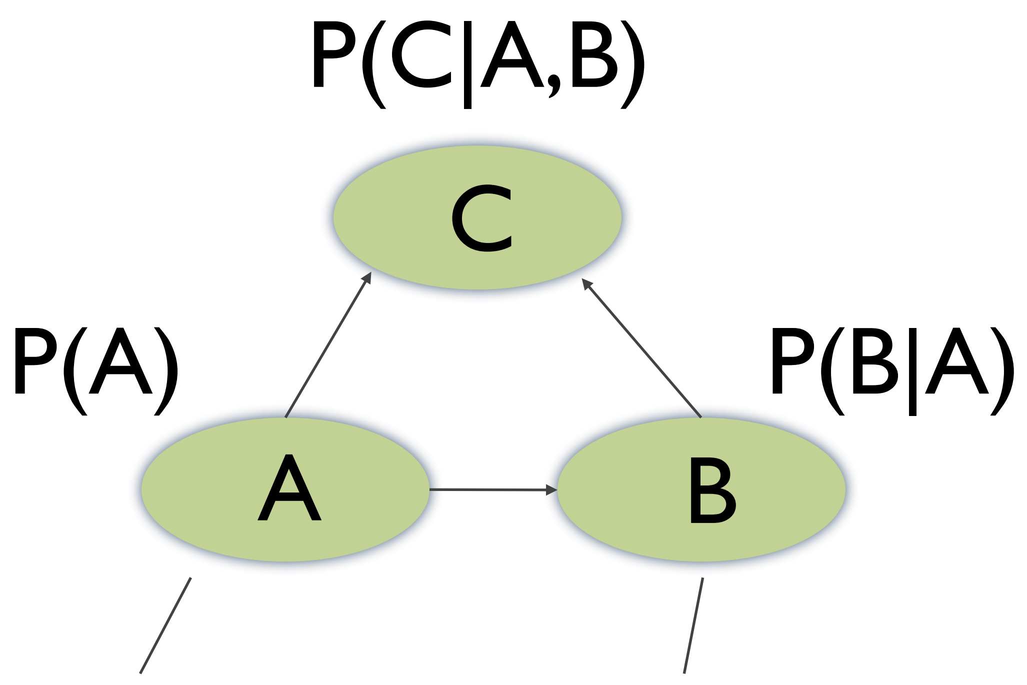
maintenance effort for the next year would be an interesting measure for maintainability.

We use public data from the NASA system "CM1".

- Space craft instrument
- Developed in C

To evaluate its quality, we use our existing maintainability model.

Bayesian nets

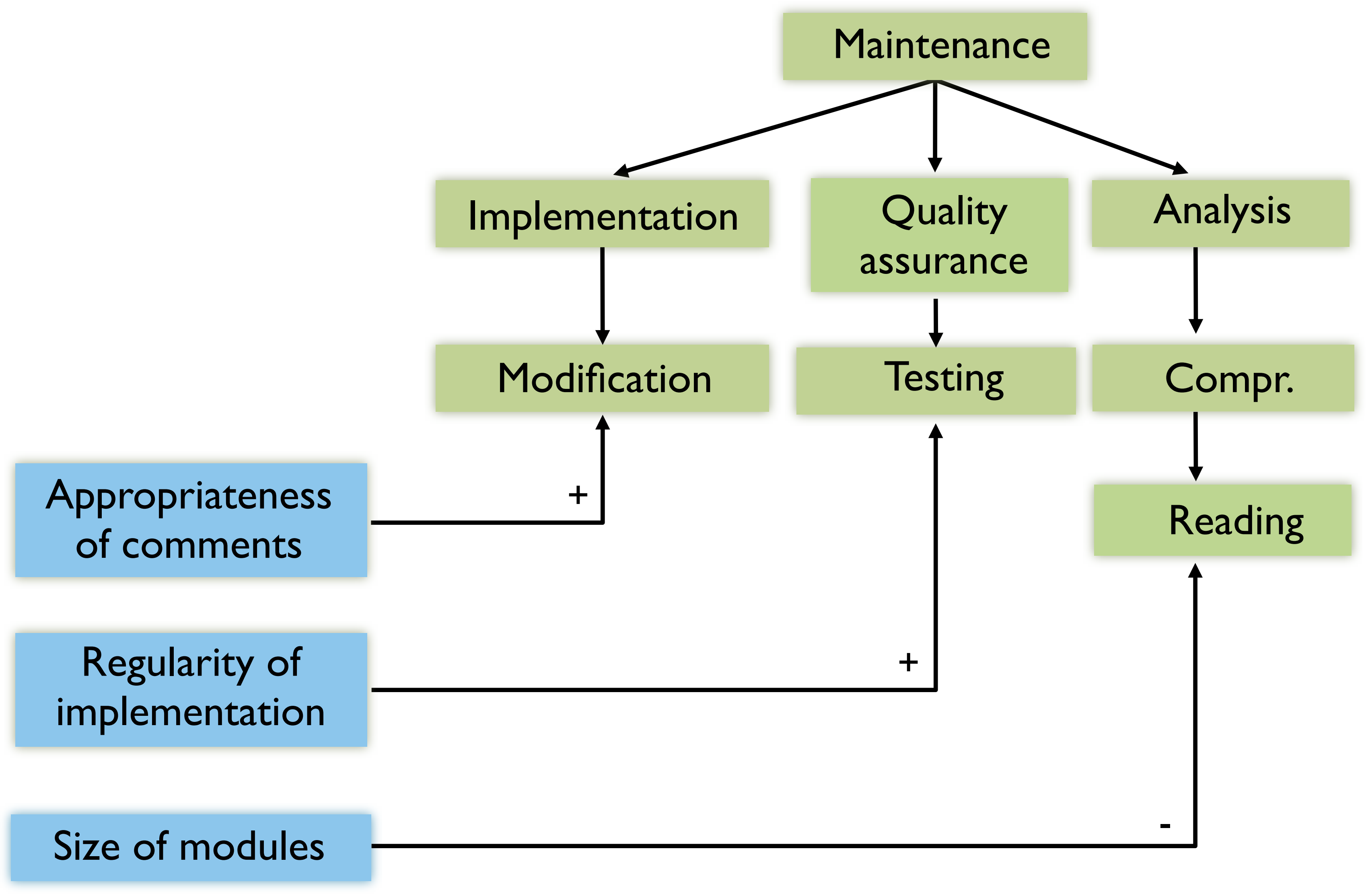


T = 0.6	
F = 0.4	

	T	F
Low	0.3	0.6
Med	0.5	0.25
High	0.2	0.15

- Cause effect graphs
- Based on Bayesian inference
- Are therefore able to model uncertainty
- Node Probability Table (NPT) for each node
- N x M
- N states in the node
- M product of the cause node states

Quality model

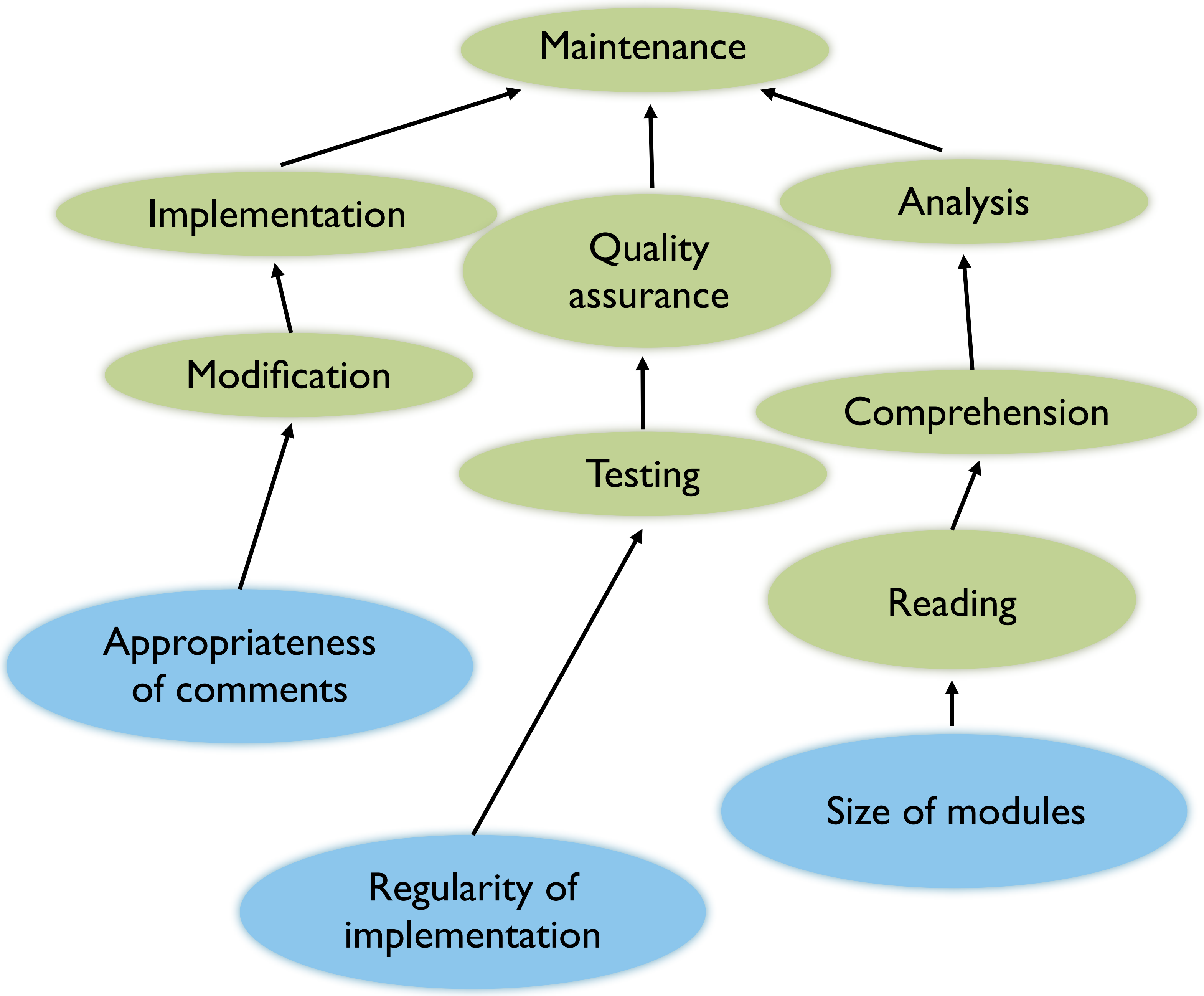


This is the simplified model we use for the quality evaluation.

It has a reduced activity tree that decomposes "maintenance" and the factors are simplified

to just three. A realistic maintainability model contains more than hundred factors.

Transition to Bayesian net

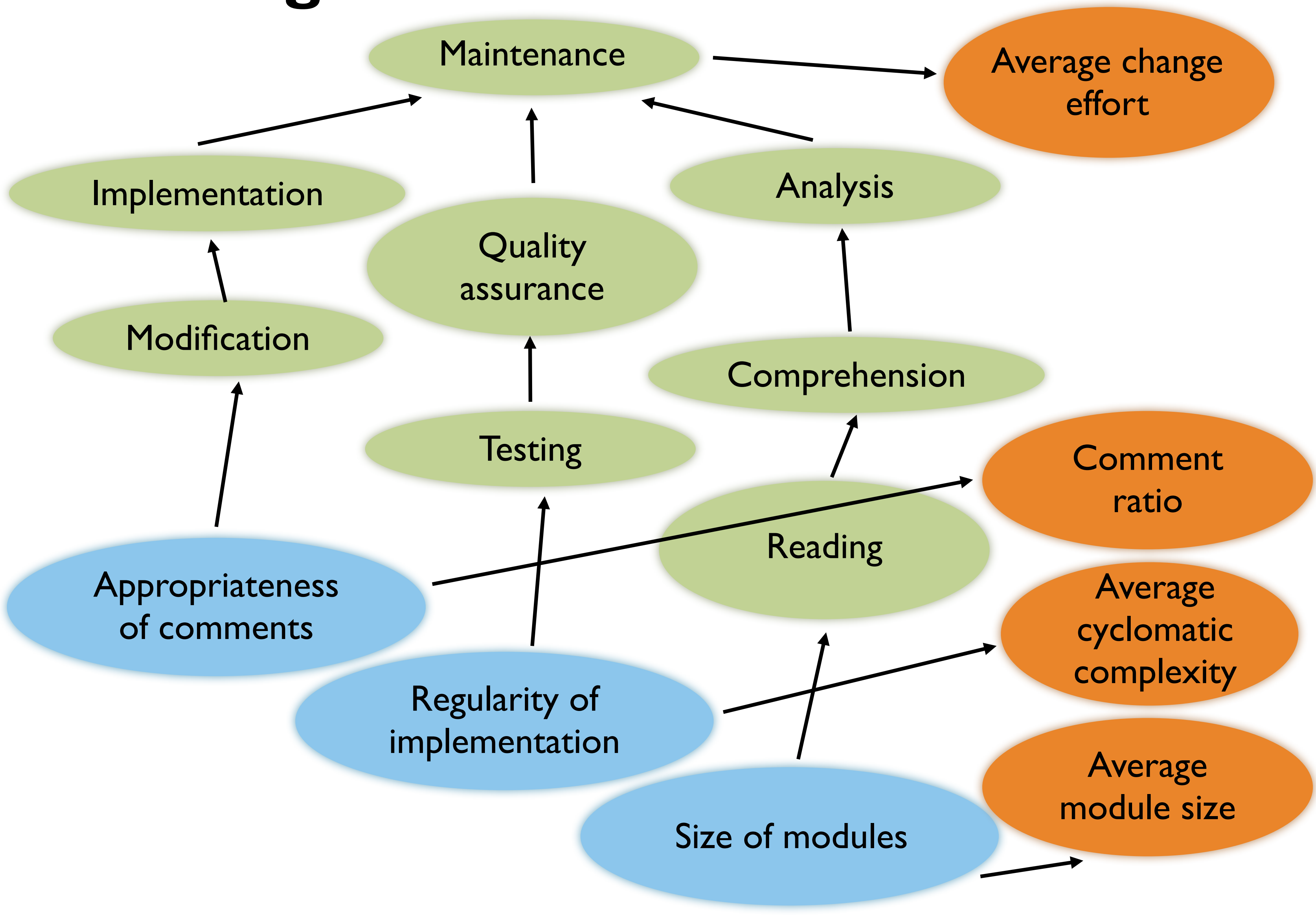


The activity-based quality model can be directly transformed into a Bayesian net.

The factors become nodes that influence the activities on which the have an impact.

The activities than influence their higher-level activities until "maintenance" is reached.

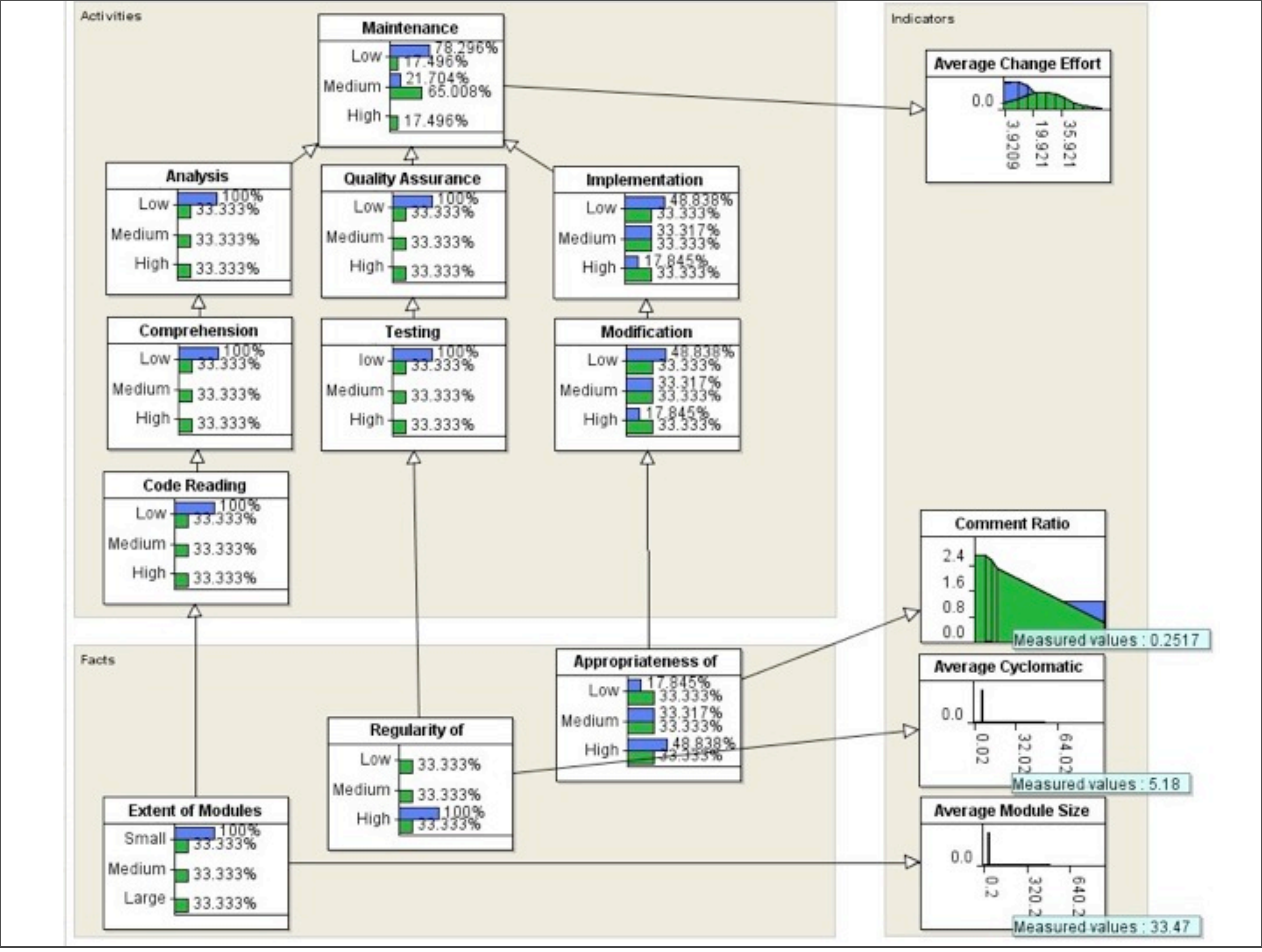
Adding measures



The factors as well as the top-level activity need explicit measures.

In the example, we only use automatically measureable measures, because these are available

in the NASA data set.



This is the Bayesian net implemented in the tool AgenaRisk.
The difficulty lies in defining suitable node probability tables for all nodes.
With the final net, it is possible to set a desired average change effort and calculate the most
probable explanation in comment ratio, average cyclomatic complexity, and average module size.
Alternatively, we can set measured values for these and calculate the distribution of the average
change effort.

Code analysis

Quality evaluation