Quality Metrics: GTE, CAP and CKJM

Kostis Kapelonis

Athens Greece, March 2010

Menu

- More Quality metrics:
- Google Testability Explorer (Starter)
- Code Analysis Plugin (Main Course)
- CKJM metrics (Dessert)

Starter

Google Testability Explorer

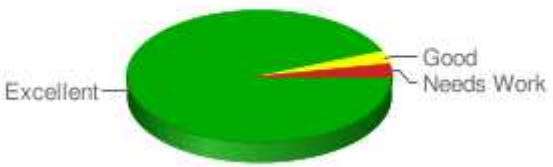
- Google Testability Explorer shows how “Testable” code you write
- Homepage is at http://code.google.com/p/testability-explorer/
- Includes reports for known projects (hibernate, ant, spring, maven)
- It is a Pre-Alpha release ! (Some things do not work at all)
- It is essentially “Propaganda” for Google Guice

Testability Report: hibernate/hibernate/3.0

Artifact Information:

- Project Website:
- Jar Location: http://mirrors.ibiblio.org/pub/mirrors/maven2/hibernate/hibernate/3.0/hibernate-3.0.jar
- Source Location:
- Analysis tool used: http://code.google.com/p/testability-explorer/
- Command: `java -jar testability-explorer.jar -cp hibernate-3.0.jar -print html .`

Overall score: 66(what is this?)



Analyzed classes 911		
- Excellent	845	92.8%
- Good	34	3.7%
- Needs work	32	3.5%

TRASYS
WE GET IT DONE

GTE: How does it work

- GTE scans all your classes
- It assigns a score to each class.
- The lower the number the “better”
- A High score means that this class needs refactoring
- Score is “testability cost” or “testability difficulty”
- How do you gain score
  - Non-Mockable-Total Cyclomatic Complexity
  - Global state, Static methods, Singletons
  - Constructors that do too much (and include the **new** keyword)
  - Calling methods on collaborators

GTE Metrics (1/4)

- Cyclomatic complexity is already calculated by other tools
- GTE finds it recursively (This is the “total”)
- GTE *excludes* complexity that can be injected(This is “non-mockable”)

```
public long calculateTax(Salary sal)
{
        long total = sal.findSalary();
        total = total - GlobalTaxObject.findTax(total);
        if(total > 2000)
                total = 2000;
        else
                return total;
}
```

This is excluded

This is included!

Total is 2 + CC of findTax()

GTE Metrics (2/4)

- All global state is bad!
- Singleton is an anti-pattern!

```
public long calculateTax()
{
        long total = Salary.findSalary();
        total = total - GlobalTaxObject.findTax(total);
        return total;
}
```

Bad

```
public long calculateTax(Salary sal,Tax tax)
{
        long total = sal.findSalary();
        total = tax.applyTax(total);
        return total;
}
```

Good

GTE Metrics (3/4)

- Minimal constructors (fast and simple)!
- No object initialization!

```
public TaxCalculator()
{
        Account account = new BankAcount()
        account.connect();
}
```

Bad

```
public TaxCalculator(Account account)
{
        this.account = account;
}
```

Good

GTE Metrics (4/4)

- Do not use an object in order to get something else
- Inject “else” directly

```
public void calculate(Account account)
{
        List<Salary> salaries = account.getSalaries(2007);
        calculateTax(salaries);
}
```

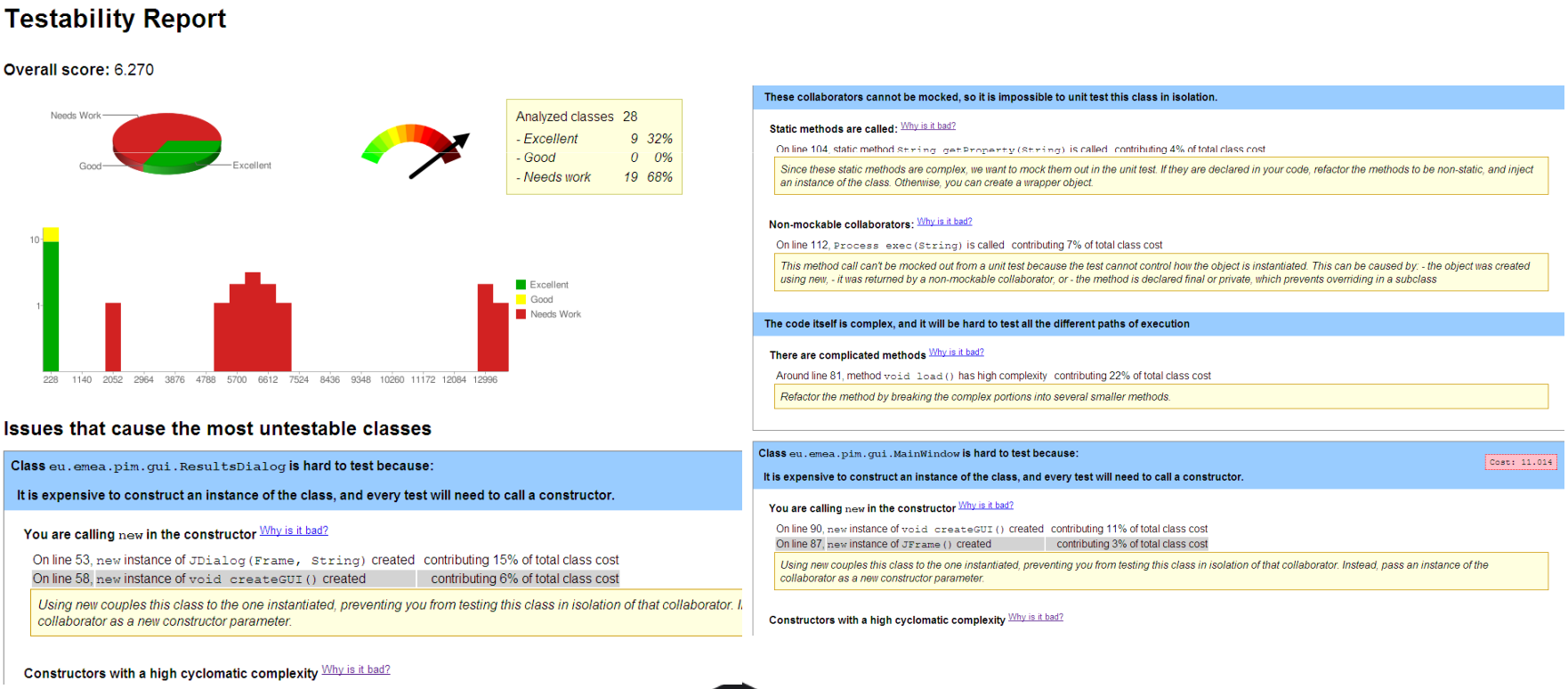
Bad

```
public void calculate(List<Salary>) salaries)
{
        calculateTax(salaries);
}
```

Good

GTE Usage

- Can run from command line
- Can run via Ant task
- Can run via Maven
- In the last case it can be integrated into mvn:site



Testability Report

Overall score: 6.270

Needs Work — Good — Excellent

Analyzed classes	28	
- Excellent	9	32%
- Good	0	0%
- Needs work	19	68%

10 / 1

Excellent
Good
Needs Work

228 1140 2052 2964 3876 4788 5700 6612 7524 8436 9348 10260 11172 12084 12996

Issues that cause the most untestable classes

Class `eu.emea.pim.gui.ResultsDialog` is hard to test because:

It is expensive to construct an instance of the class, and every test will need to call a constructor.

You are calling `new` in the constructor Why is it bad?

On line 53, `new` instance of `JDialog(Frame, String)` created contributing 15% of total class cost

On line 58, `new` instance of `void createGUI()` created contributing 6% of total class cost

Using new couples this class to the one instantiated, preventing you from testing this class in isolation of that collaborator. I collaborator as a new constructor parameter.

Constructors with a high cyclomatic complexity Why is it bad?

These collaborators cannot be mocked, so it is impossible to unit test this class in isolation.

Static methods are called: Why is it bad?

On line 104, static method `String getProperty(String)` is called contributing 4% of total class cost

Since these static methods are complex, we want to mock them out in the unit test. If they are declared in your code, refactor the methods to be non-static, and inject an instance of the class. Otherwise, you can create a wrapper object.

Non-mockable collaborators: Why is it bad?

On line 112, `Process exec(String)` is called contributing 7% of total class cost

This method call can't be mocked out from a unit test because the test cannot control how the object is instantiated. This can be caused by: - the object was created using new, - it was returned by a non-mockable collaborator, or - the method is declared final or private, which prevents overriding in a subclass

The code itself is complex, and it will be hard to test all the different paths of execution

There are complicated methods Why is it bad?

Around line 81, method `void load()` has high complexity contributing 22% of total class cost

Refactor the method by breaking the complex portions into several smaller methods.

Class `eu.emea.pim.gui.MainWindow` is hard to test because: Cost: 11.014

It is expensive to construct an instance of the class, and every test will need to call a constructor.

You are calling `new` in the constructor Why is it bad?

On line 90, `new` instance of `void createGUI()` created contributing 11% of total class cost

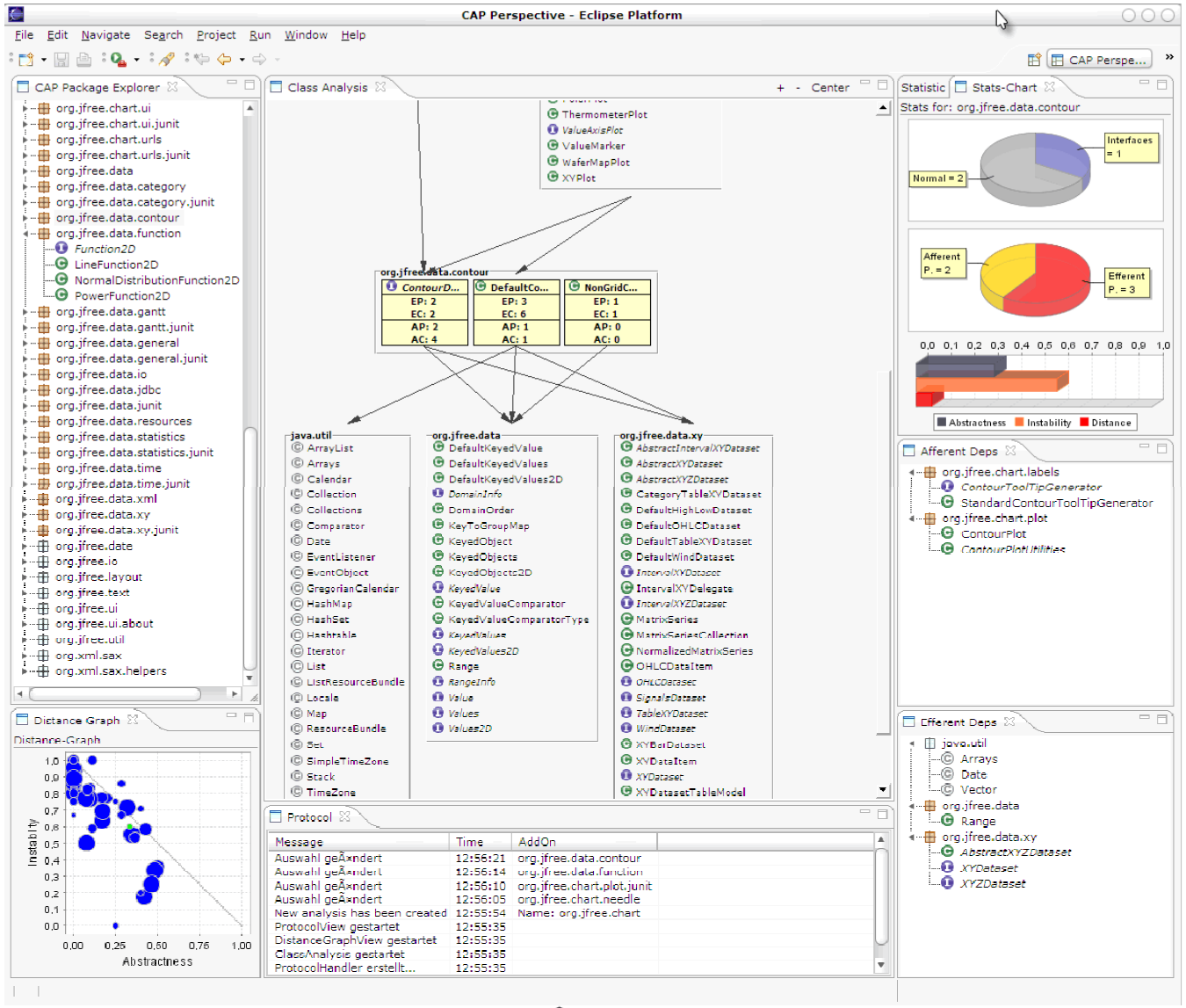
On line 87, `new` instance of `JFrame()` created contributing 3% of total class cost

Using new couples this class to the one instantiated, preventing you from testing this class in isolation of that collaborator. Instead, pass an instance of the collaborator as a new constructor parameter.

Constructors with a high cyclomatic complexity Why is it bad?

Main Course

Code analysis plugin



CAP Description

- Eclipse Plugin (http://cap.xore.de/)
- Essentially a GUI on JDepend
- If you understand JDepend, CAP will be trivial to use
- Metrics used are covered in “OO Design Quality Metrics” by Robert Martin in 1994
- http://www.objectmentor.com/resources/articles/oodmetrc.pdf

Metric Results

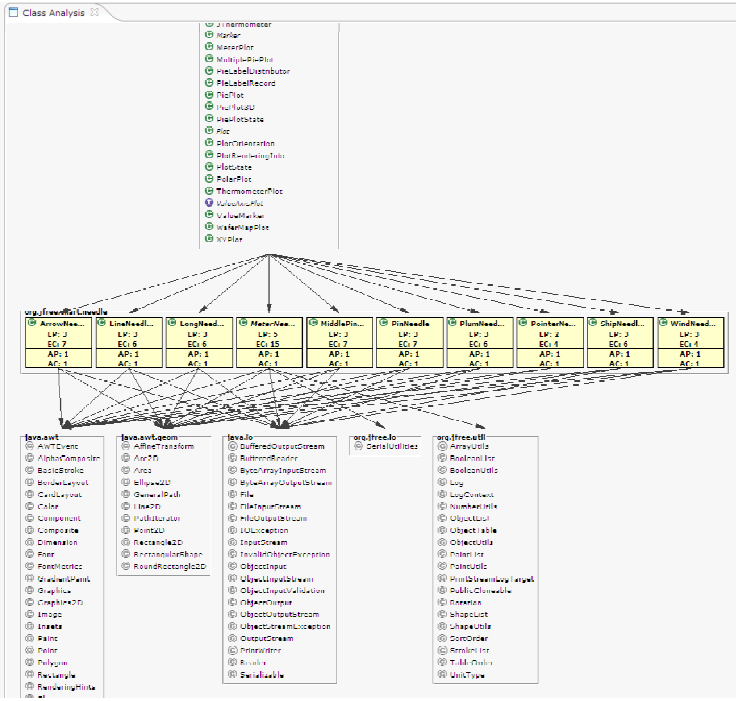
[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

Summary

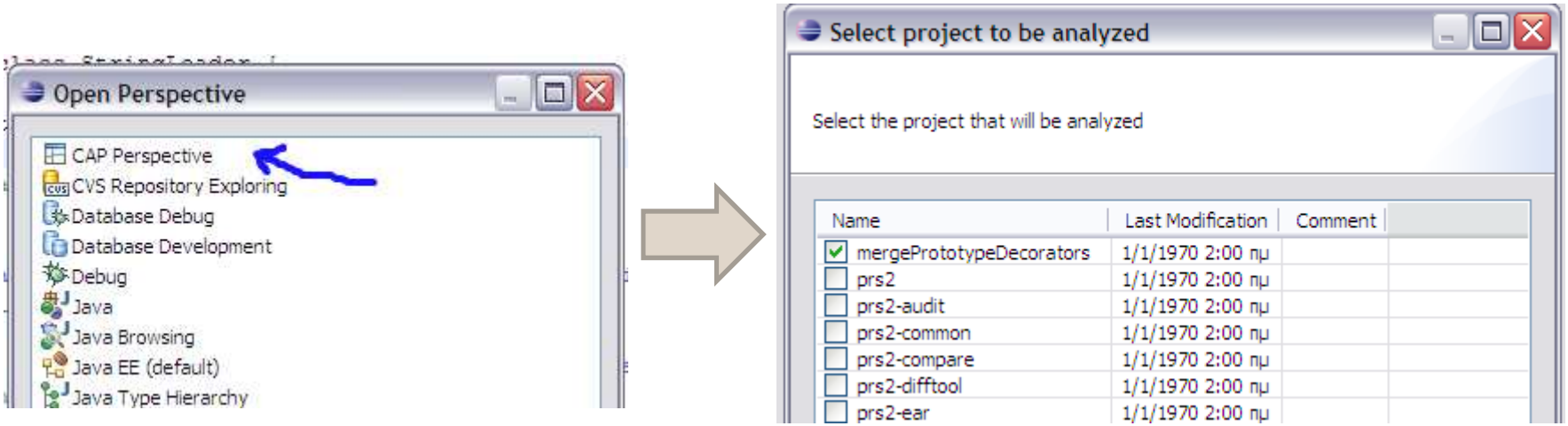
[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.displaytag	1	1	0	9	3	0.0%	25.0%	75.0%	1
org.displaytag.decorator	13	8	5	5	13	38.0%	72.0%	11.0%	1
org.displaytag.exception	14	12	2	7	6	14.0%	46.0%	40.0%	1
org.displaytag.export	16	11	5	1	17	31.0%	94.0%	26.0%	1
org.displaytag.filter	6	5	1	0	10	17.0%	100.0%	17.0%	1
org.displaytag.localization	6	4	2	2	18	33.0%	90.0%	23.0%	1
org.displaytag.model	9	9	0	5	13	0.0%	72.0%	28.0%	1
org.displaytag.pagination	5	4	1	2	8	20.0%	80.0%	0.0%	1
org.displaytag.properties	5	5	0	7	17	0.0%	71.0%	29.0%	1
org.displaytag.render	6	3	3	2	16	50.0%	89.0%	39.0%	1
org.displaytag.tags	12	10	2	2	24	17.0%	92.0%	9.0%	1
org.displaytag.tags.el	8	8	0	0	8	0.0%	100.0%	0.0%	1
org.displaytag.util	18	14	4	7	15	22.0%	68.0%	10.0%	1



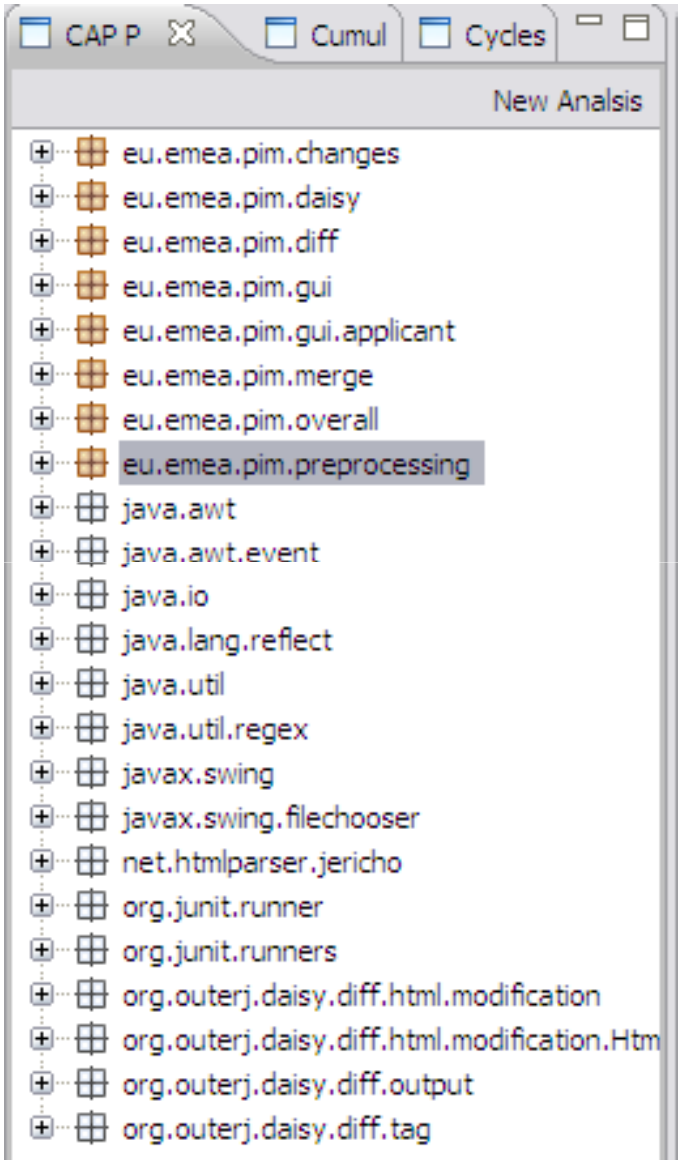
Cap Usage

- Install CAP from the Eclipse Update site
- http://cap.xore.de/update (JFreechart will also be installed)
- CAP has its own Eclipse perspective!
- While CAP has a lot of screens, they all show the same thing
- There are actually two things that you can do
  - Find cycles in your packages
  - Inspect the JDepend distance (how good is your architecture)



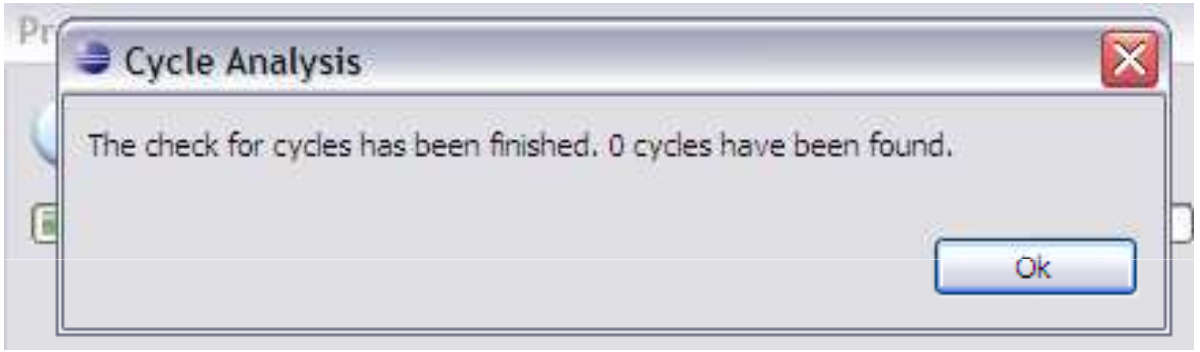
Cap Screens (1/5)

- On the left
- All packages of the application.
- White packages are libraries
- Acts as selector for the rest of the screens
- Cycle Detection

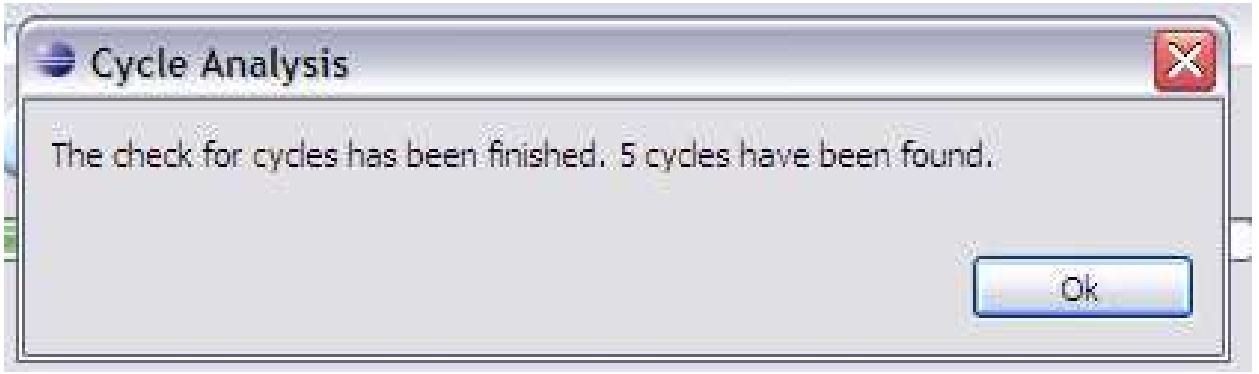


Detecting cycles

- Detect Cycles as JDepend does
- Also detected by Sonar or CAST
- Click the “Check Cycles” button on the “Cycle” Tab



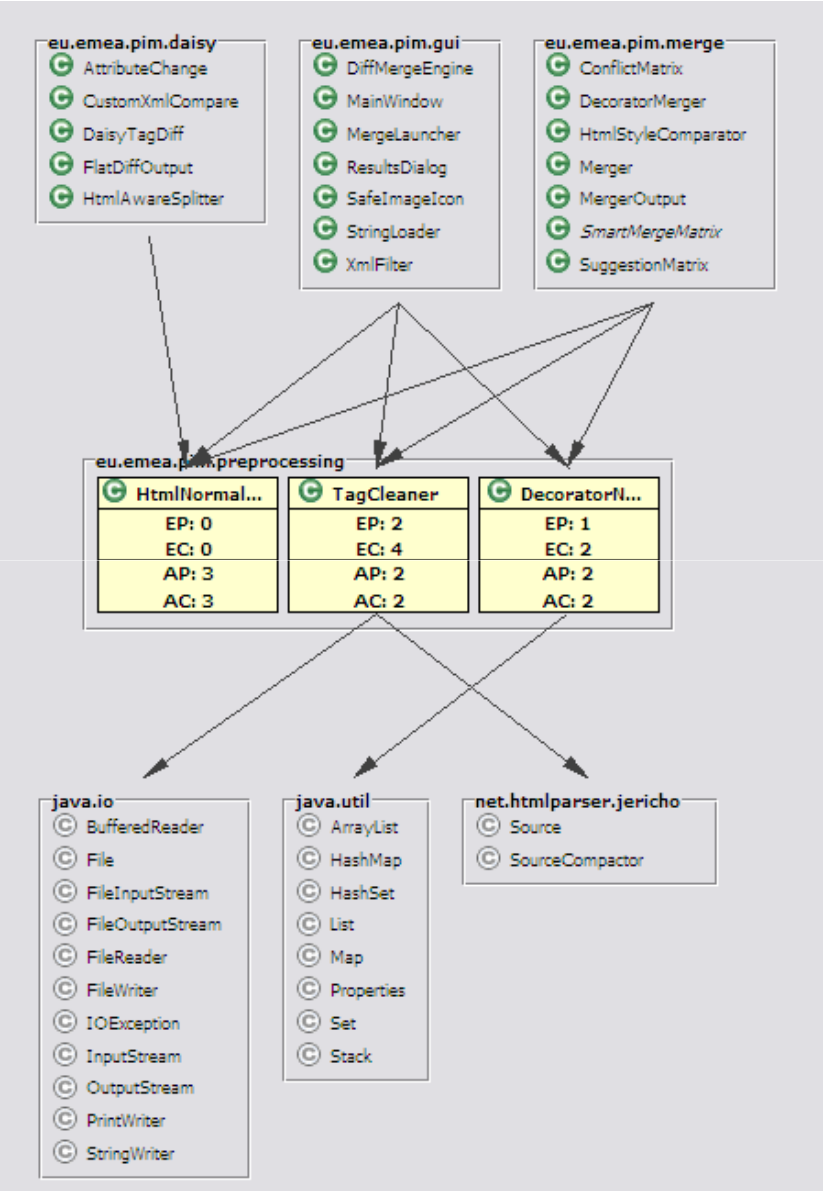
Good



Bad

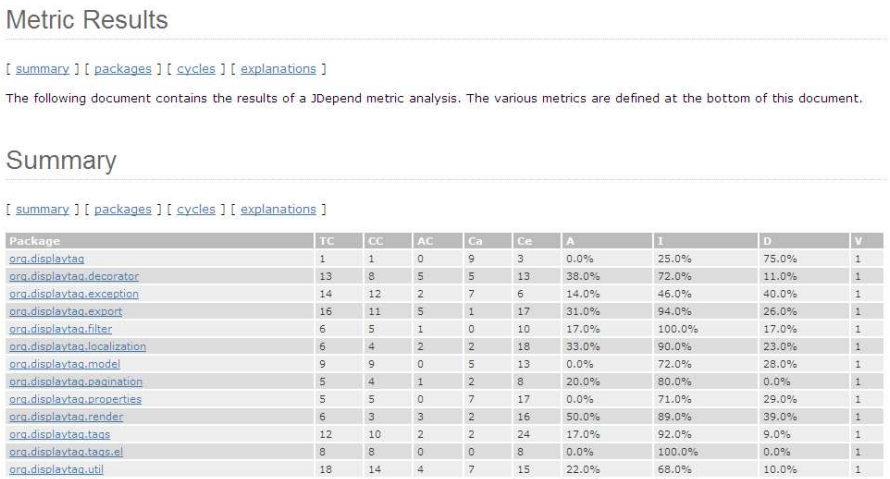
Cap Screens (2/5)

- Main view shows package dependencies (imports)
- Same information as JDepend
- Shows package selected from the tree (Screen 1)



Cap Screens (3/5)

- Right view shows numerical info
- Same information as JDepend
- Abstractness, instability and Distance



Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

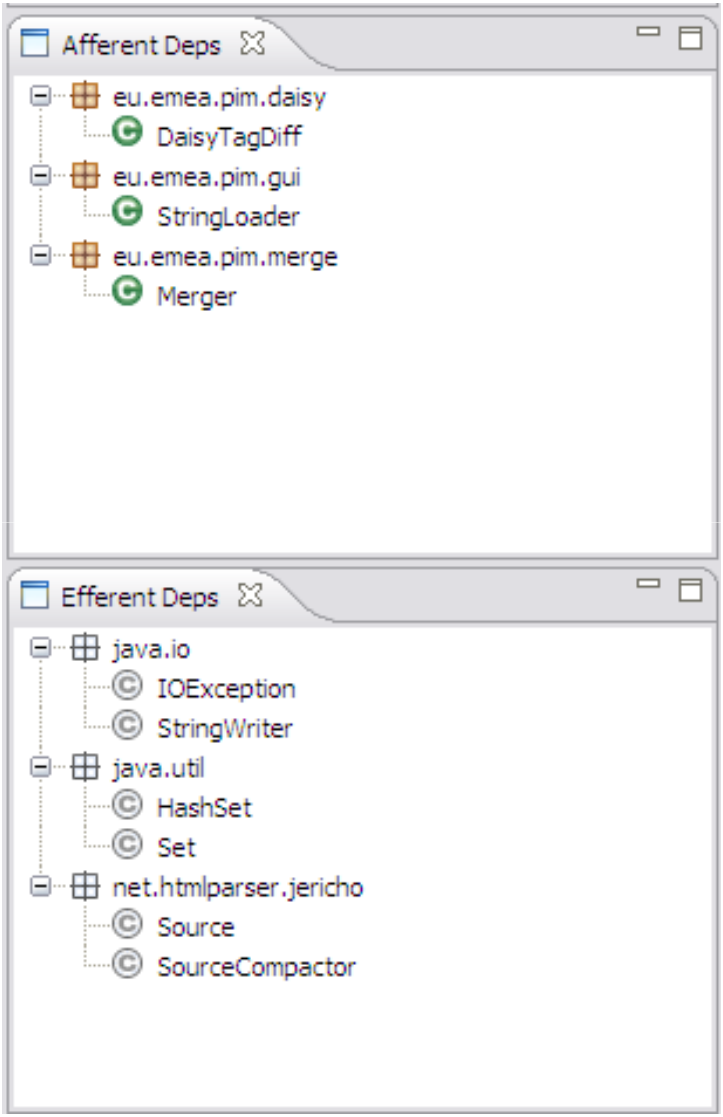
Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.displaytag	1	1	0	9	3	0.0%	25.0%	75.0%	1
org.displaytag.decorator	13	8	5	5	13	38.0%	72.0%	11.0%	1
org.displaytag.exception	14	12	2	7	6	14.0%	46.0%	40.0%	1
org.displaytag.export	16	11	5	1	17	31.0%	94.0%	26.0%	1
org.displaytag.filter	6	5	1	0	10	17.0%	100.0%	17.0%	1
org.displaytag.localization	6	4	2	2	18	33.0%	90.0%	23.0%	1
org.displaytag.model	9	9	0	5	13	0.0%	72.0%	28.0%	1
org.displaytag.pagination	5	4	1	2	8	20.0%	80.0%	0.0%	1
org.displaytag.properties	5	5	0	7	17	0.0%	71.0%	29.0%	1
org.displaytag.render	6	3	3	2	16	50.0%	89.0%	39.0%	1
org.displaytag.tags	12	10	2	2	24	17.0%	92.0%	9.0%	1
org.displaytag.tags.el	8	8	0	0	8	0.0%	100.0%	0.0%	1
org.displaytag.util	18	14	4	7	15	22.0%	68.0%	10.0%	1

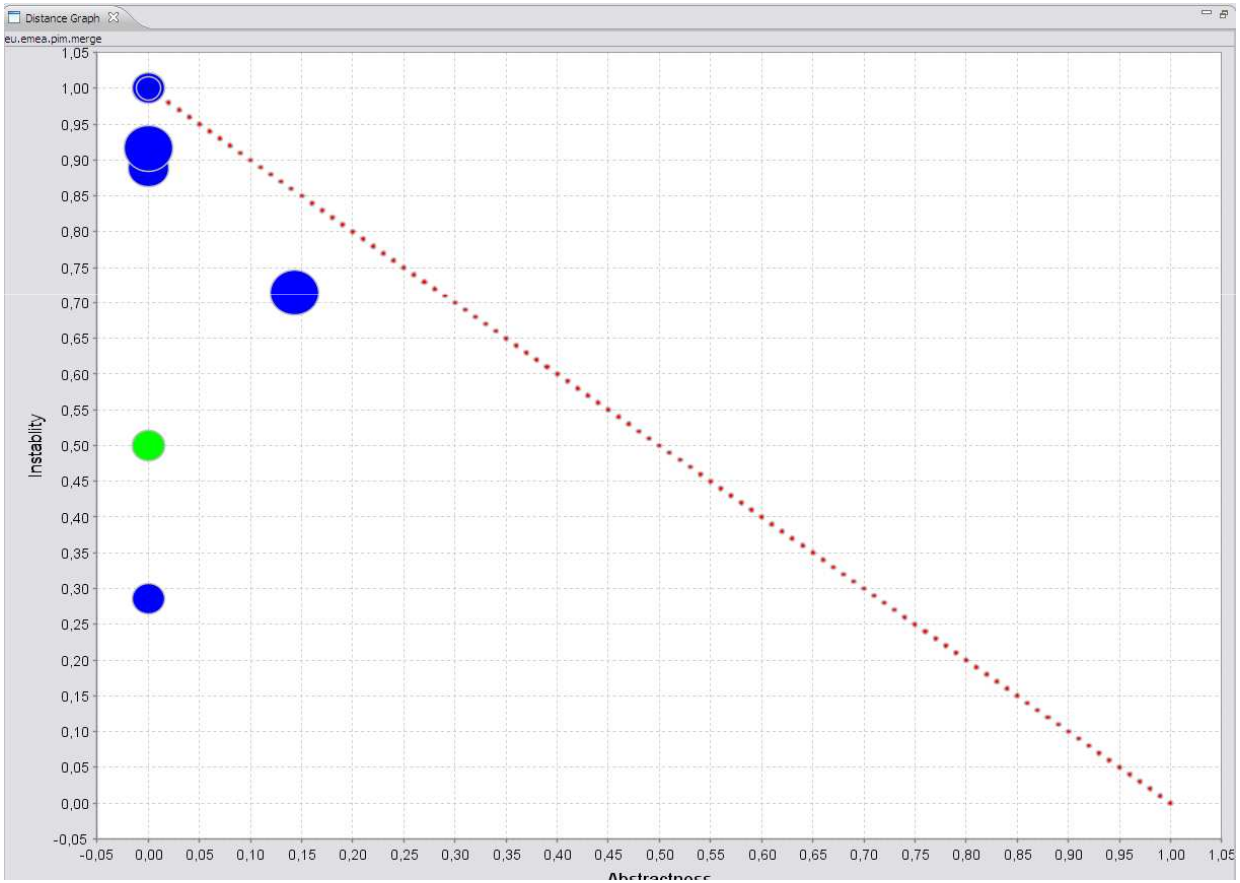
Cap Screens (4/5)

- Right view shows same thing as main view
- Same information as JDepend



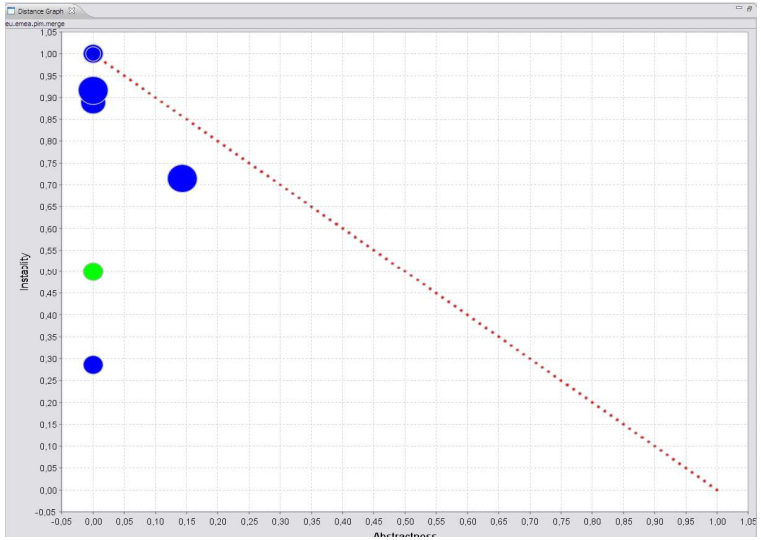
Cap Screens (5/5)

- Most **important** screen
- Shows architecture distance
- I would be happy if you use **only** this from the presentation



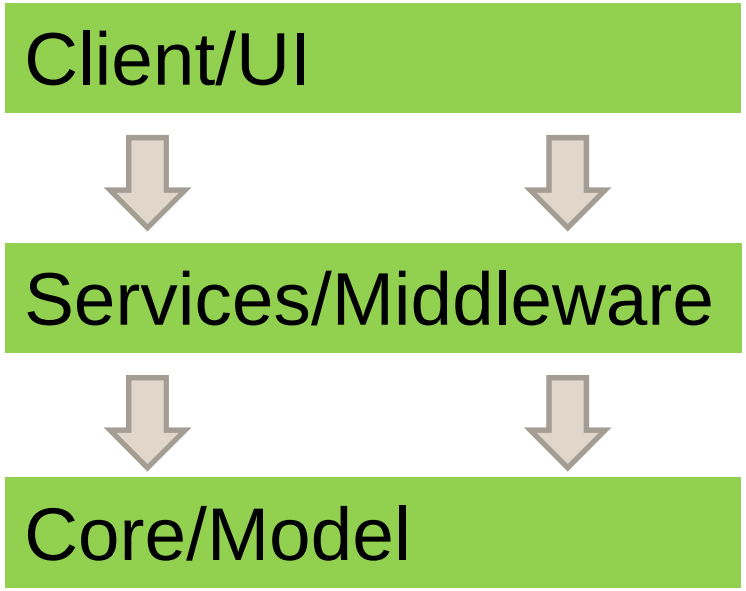
Architecture Distance

- The graph shows architecture distance
- Each circle is a package from your code
- Distance is a number from 0% to 100%
- Distance is also reported by JDepend
- Distance is 0% means perfect system, 100% means ugly system
- We need to define what is the *perfect system according to JDepend*
- We also define instability and abstractness (also by JDepend)



Typical Enterprise system

- There are classes *used* by everybody
- There are classes that *use* everybody else
- Each layer depends on the one below (ideally)
- Core classes do not depend on anything
- Clients are not used by anything

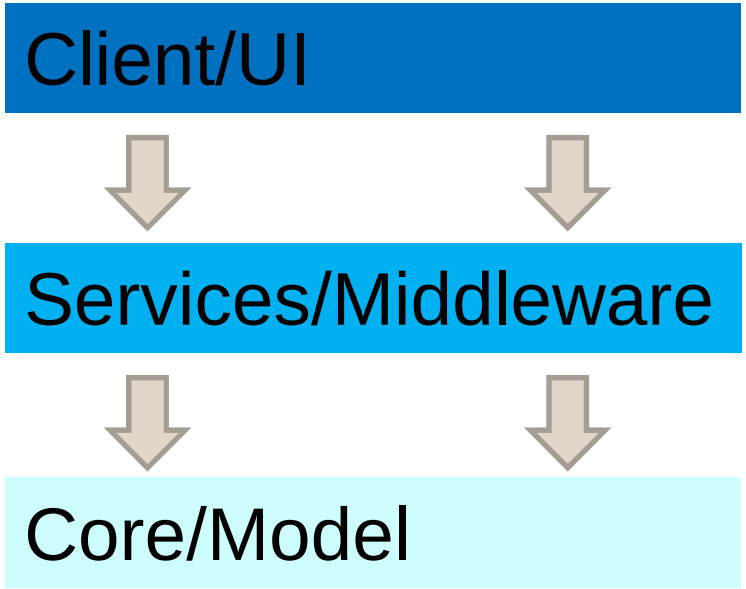


“Perfect” Enterprise system

- JDepend suggests that:
- Classes that *use* everybody else should be concrete
- Classes *used* by everybody should be abstract
- Distance is how far you are from this perfect system

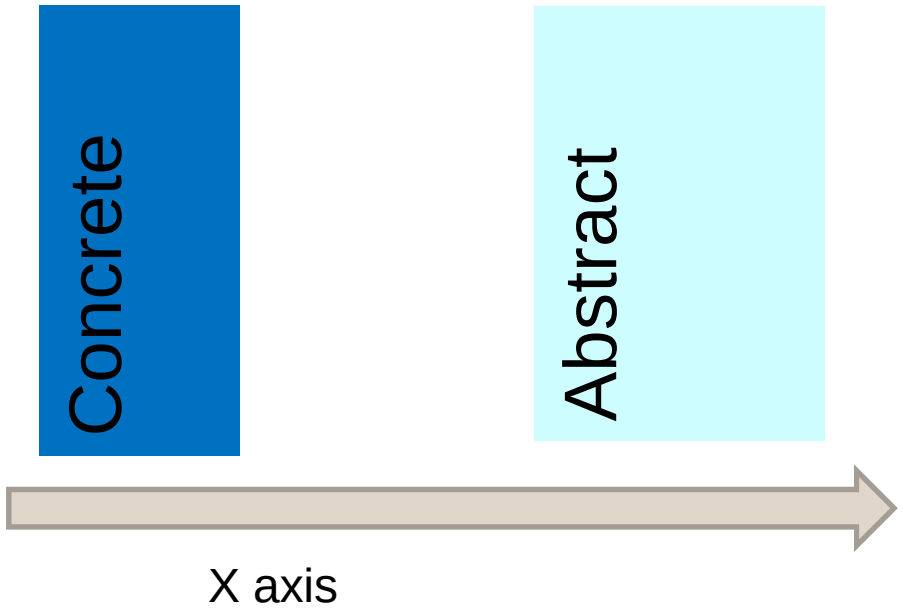
Concrete Classes

Abstract/Interfaces



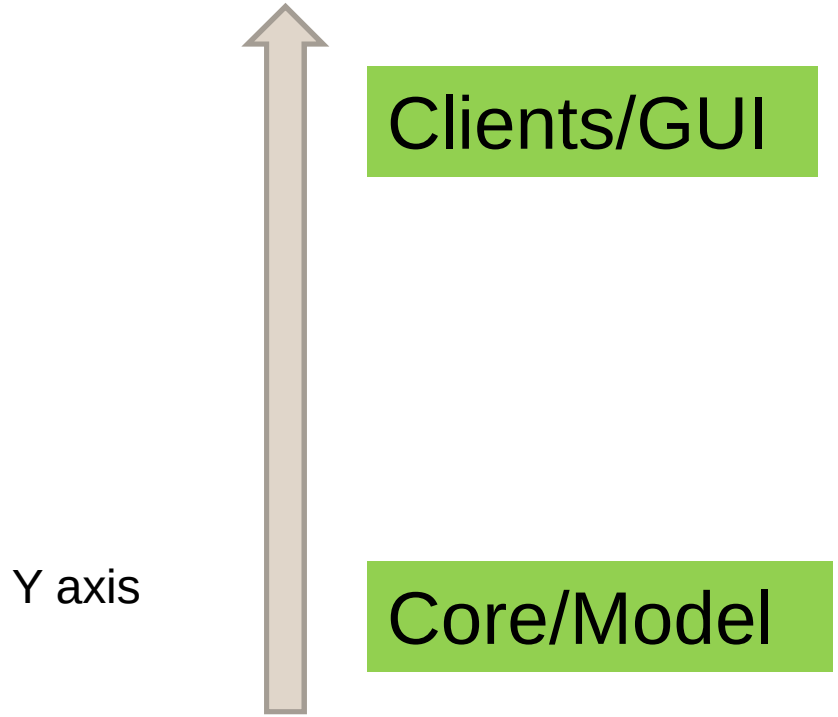
Abstractness

- Percent of classes in a package that are abstract/interfaces
- 0 = a package with concrete classes only
- 1 = a package with only abstract classes
- The x – direction shows “abstractness” of a package



(In)stability

- (Inverse) Ratio of packages that are depended upon this package
- 0 = a package that everybody uses
- 1 = a package nobody uses
- The y – direction shows “instability” of a package



Distance Example (1/5)

- Perfect package for gui/clients
- Used by nobody (instability = 1)
- All classes are concrete (abstractness = 0)

Good!

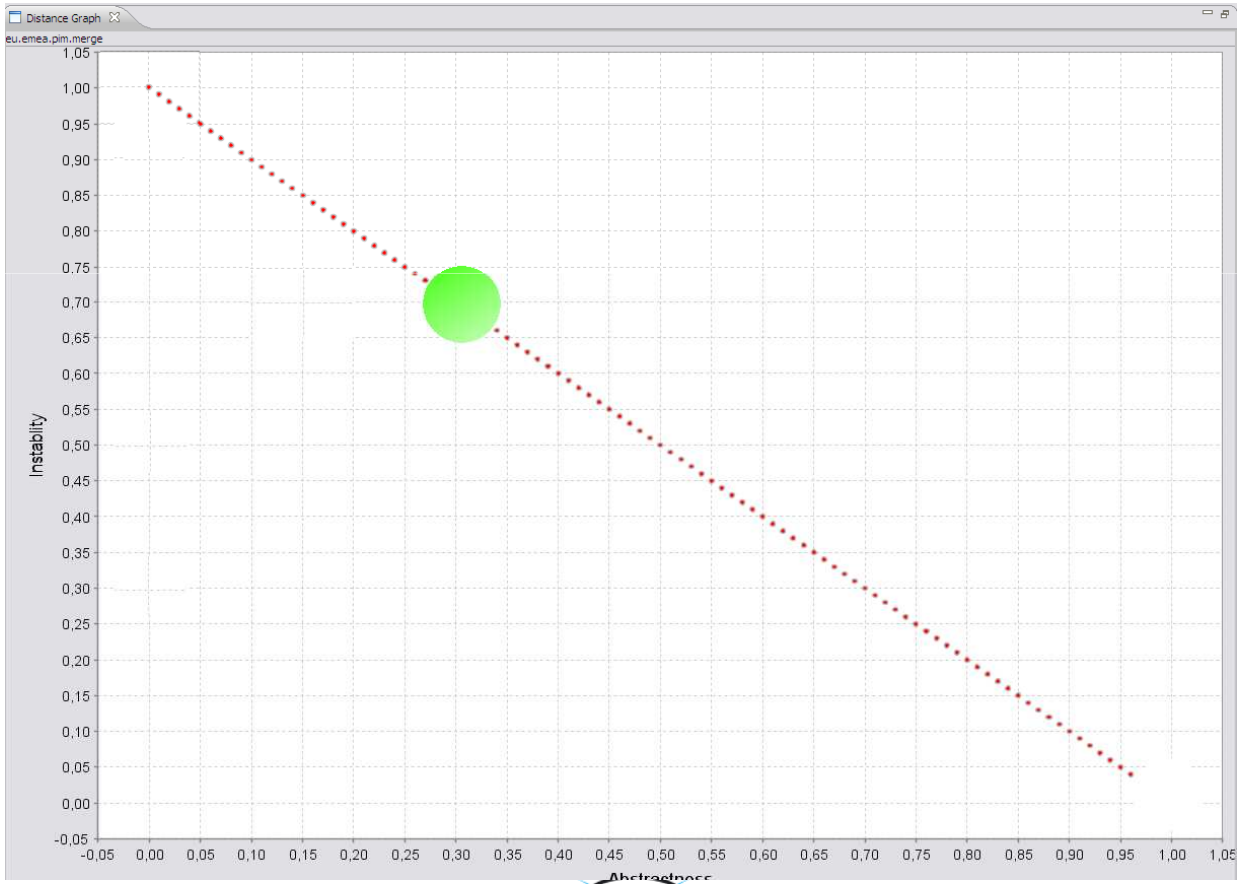
Distance Example (2/5)

- Perfect package for core/model
- Used by everybody(instability = 0)
- No concrete class (abstractness = 1)

Good!

Distance Example (3/5)

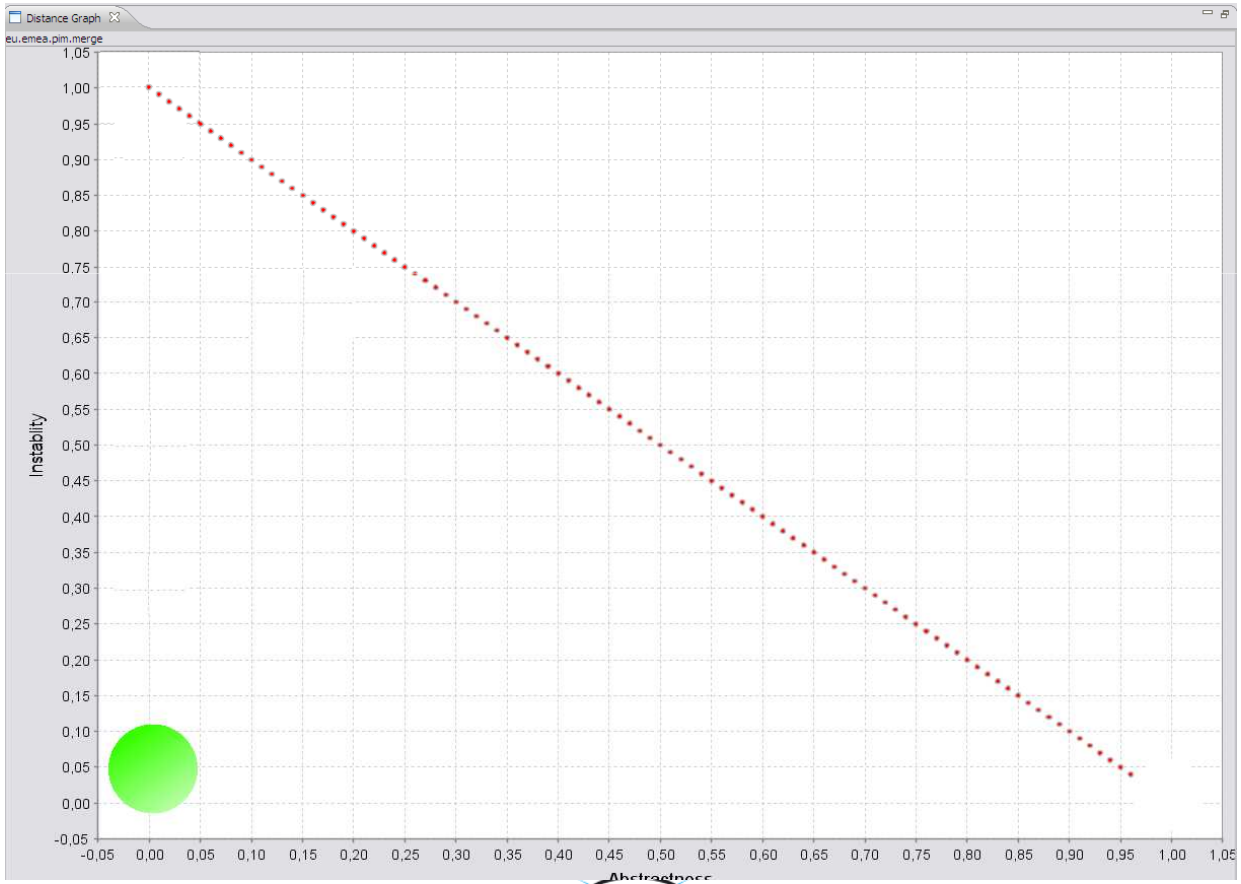
- Middleware
- Used by some and uses others (instability = 0 -1)
- Both concrete class and abstract classes(abstractness = 0 -1)



Good!

Distance Example (4/5)

- Badly designed core (most common case!)
- Used by everybody (instability = 0)
- Concrete implementation – hard to change (abstractness = 0)



Bad

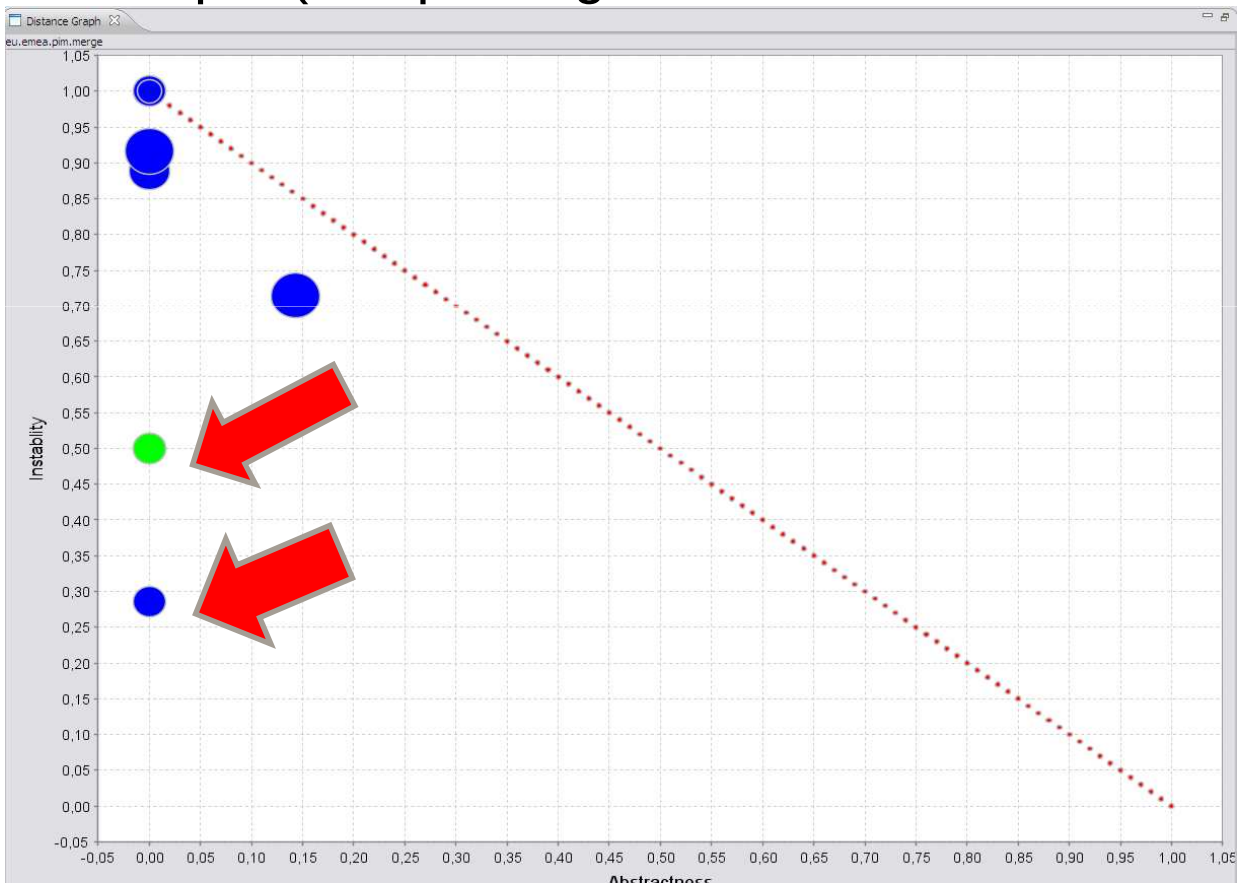
Distance Example (5/5)

- Un-needed abstract classes (uncommon case)
- Used by nobody(instability = 1)
- Abstract implementation (abstractness = 1)

Bad

JDepend distance on small PIM prototype

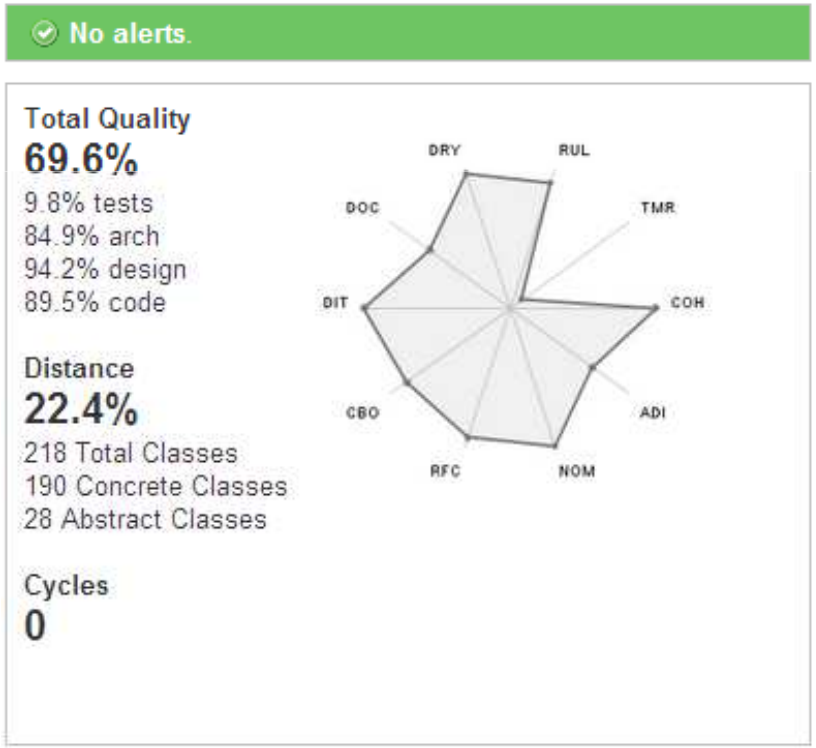
- With one look you can see the general architecture
- You can select a package on the graph to see details
- Real life example (two packages should be more abstract)



Dessert

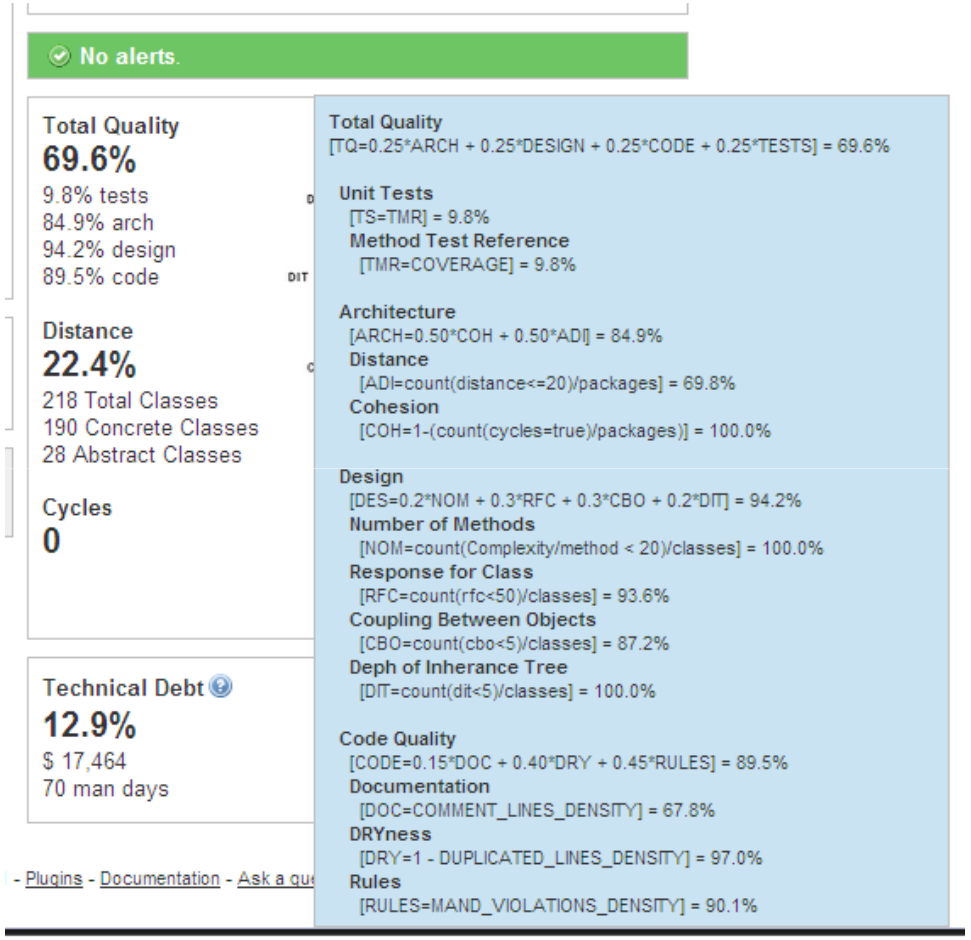
Chidamber and Kemerer Java Metrics

- CKJM Metrics (1996 paper)
- http://www.spinellis.gr/sw/ckjm/
- Sonar plugin: http://docs.codehaus.org/display/SONAR/Isotrol+MetricsAnalytics
- Defines (not all are used)
  - WMC
  - DIT
  - NOC
  - CBO
  - RFC
  - LCOM
  - CA
  - NPM



Usage of the Sonar plugin

- Total Quality of a project
- Defined by 10 metrics
- Grouped in 4 categories
- Formula of total quality:
  - 25% Tests
  - 25% Architecture
  - 25% Design
  - 25% Code



Metric: Test coverage (1/10)

- Test Coverage
- Is a whole category on its own (25% of Total quality)
- Formula: Test coverage = Tests Category

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	55	75%	1625/2179	64%	472/738	2.319
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	1.848
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	2.429
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	2.277
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	1.854
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	5.5
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	2.882
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	4.444
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	4.5
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	1.524
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	2.892
someotherpackage	1	83%	5/6	N/A	N/A	1.2

Metric: ADI (2/10)

- Distance from the main sequence (same as JDepend)
- Is the first half of Architecture category (12% of Total quality)
- Sonar suggests that values less that 20% are optimal
- Formula: Percent of optimal packages / total packages

Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.displaytag	1	1	0	9	3	0.0%	25.0%	75.0%	1
org.displaytag.decorator	13	8	5	5	13	38.0%	72.0%	11.0%	1
org.displaytag.exception	14	12	2	7	6	14.0%	46.0%	40.0%	1
org.displaytag.export	16	11	5	1	17	31.0%	94.0%	26.0%	1
org.displaytag.filter	6	5	1	0	10	17.0%	100.0%	17.0%	1
org.displaytag.localization	6	4	2	2	18	33.0%	90.0%	23.0%	1
org.displaytag.model	9	9	0	5	13	0.0%	72.0%	28.0%	1
org.displaytag.pagination	5	4	1	2	8	20.0%	80.0%	0.0%	1
org.displaytag.properties	5	5	0	7	17	0.0%	71.0%	29.0%	1
org.displaytag.render	6	3	3	2	16	50.0%	89.0%	39.0%	1
org.displaytag.tags	12	10	2	2	24	17.0%	92.0%	9.0%	1
org.displaytag.tags.el	8	8	0	0	8	0.0%	100.0%	0.0%	1
org.displaytag.util	18	14	4	7	15	22.0%	68.0%	10.0%	1

Metric: Cohesion (3/10)

- Package cycles (same as JDepend)
- Is the second half of Architecture category (12% of Total quality)
- Sonar assumes that only 0 cycle packages are optimal
- Formula: Percent of optimal packages / total packages

Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

Summary

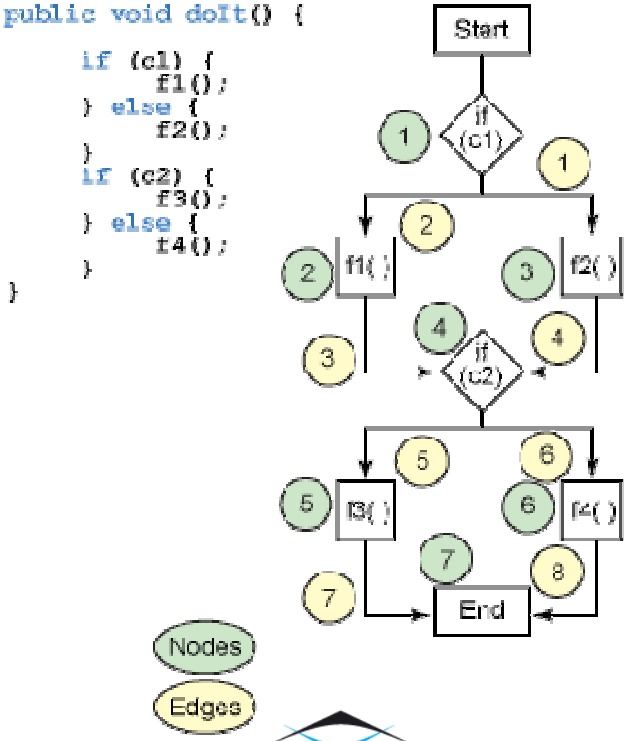
[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.displaytag	1	1	0	9	3	0.0%	25.0%	75.0%	1
org.displaytag.decorator	13	8	5	5	13	38.0%	72.0%	11.0%	1
org.displaytag.exception	14	12	2	7	6	14.0%	46.0%	40.0%	1
org.displaytag.export	16	11	5	1	17	31.0%	94.0%	26.0%	1
org.displaytag.filter	6	5	1	0	10	17.0%	100.0%	17.0%	1
org.displaytag.localization	6	4	2	2	18	33.0%	90.0%	23.0%	1
org.displaytag.model	9	9	0	5	13	0.0%	72.0%	28.0%	1
org.displaytag.pagination	5	4	1	2	8	20.0%	80.0%	0.0%	1
org.displaytag.properties	5	5	0	7	17	0.0%	71.0%	29.0%	1
org.displaytag.render	6	3	3	2	16	50.0%	89.0%	39.0%	1
org.displaytag.tags	12	10	2	2	24	17.0%	92.0%	9.0%	1
org.displaytag.tags.el	8	8	0	0	8	0.0%	100.0%	0.0%	1
org.displaytag.util	18	14	4	7	15	22.0%	68.0%	10.0%	1

Metric: Nom(4/10)

- Sonar Metric Number of Methods
- Is the first part of Design category (5% of Total quality)
- Average Cyclomatic complexity for methods of a class
- Sonar assumes that less than 20 for a method is optimal
- Formula: Percent of optimal classes/ total classes

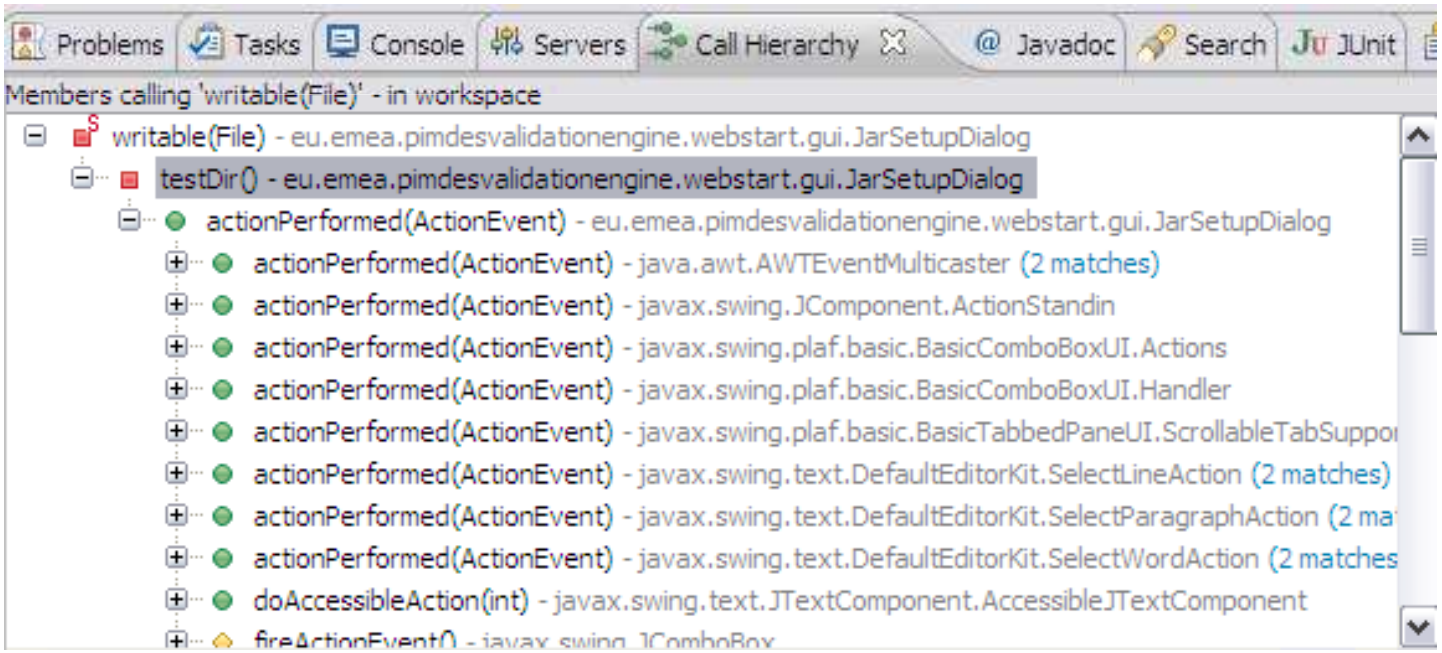
```
public void doIt() {
    if (c1) {
        f1();
    } else {
        f2();
    }
    if (c2) {
        f3();
    } else {
        f4();
    }
}
```



TRASYS
WE GET IT DONE

Metric: RFC(5/10)

- CKJM Metric Response per class
- Is the second part of Design category (7,5% of Total quality)
- Number of methods a method calls (recursively)
- Sonar assumes that less than 50 for a class is optimal
- Formula: Percent of optimal classes/ total classes



Metric: CBO(6/10)

- CJM Metric Coupling between objects
- Is the third part of Design category (7,5% of Total quality)
- Number of classes used by a class (similar to JDepend CE)
- Sonar assumes that less than 5 for a class is optimal
- Formula: Percent of optimal classes/ total classes

Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
org.displaytag	1	1	0	9	3	0.0%	25.0%	75.0%	1
org.displaytag.decorator	13	8	5	5	13	38.0%	72.0%	11.0%	1
org.displaytag.exception	14	12	2	7	6	14.0%	46.0%	40.0%	1
org.displaytag.export	16	11	5	1	17	31.0%	94.0%	26.0%	1
org.displaytag.filter	6	5	1	0	10	17.0%	100.0%	17.0%	1
org.displaytag.localization	6	4	2	2	18	33.0%	90.0%	23.0%	1
org.displaytag.model	9	9	0	5	13	0.0%	72.0%	28.0%	1
org.displaytag.pagination	5	4	1	2	8	20.0%	80.0%	0.0%	1
org.displaytag.properties	5	5	0	7	17	0.0%	71.0%	29.0%	1
org.displaytag.render	6	3	3	2	16	50.0%	89.0%	39.0%	1
org.displaytag.tags	12	10	2	2	24	17.0%	92.0%	9.0%	1
org.displaytag.tags.el	8	8	0	0	8	0.0%	100.0%	0.0%	1
org.displaytag.util	18	14	4	7	15	22.0%	68.0%	10.0%	1

Metric: DIT(7/10)

- CJM Metric Depth of Inheritance Tree
- Is the fourth part of Design category (5% of Total quality)
- How deep the hierarchy goes
- Sonar assumes that less than 5 for a class is optimal
- Formula: Percent of optimal classes/ total classes

Metric: DOC(8/10)

- Sonar Metric Documention
- Is the first part of Code category (3,75% of Total quality)
- How many comments exist in the code
- Sonar assumes 40% of lines should be comments
- Formula: Percent of comment * 10 / 4

```
/**
 * Reads an HTML file from the filesystem and cleans it up.
 * e.g. all tags are converted to lower case
 * @param filename full path of the HTML file
 * @param cleanup Cleanup and normalize the String loaded.
 * @return the text contained in the HTML file
 * @throws Exception something went wrong
 */
public static String loadString(String filename, boolean cleanup) throws Exception {
    File file = new File(filename);
    byte[] buf = new byte[(int) file.length()];
    FileInputStream in = new FileInputStream(filename);
    in.read(buf);
    in.close();
```

Metric: Dry(9/10)

- CPD Metric Duplicated lines
- Is the second part of Code category (10% of Total quality)
- How many code lines are the same
- Sonar assumes no code lines should be the same
- Formula: Percent of non-duplicated lines/ total lines

CPD Results

The following document contains the results of PMD's CPD 4.1.

Duplications

File	Line
eu\emea\pim\gui\applicant\ApplicantMergeEngine.java	69
eu\emea\pim\gui\DiffMergeEngine.java	59

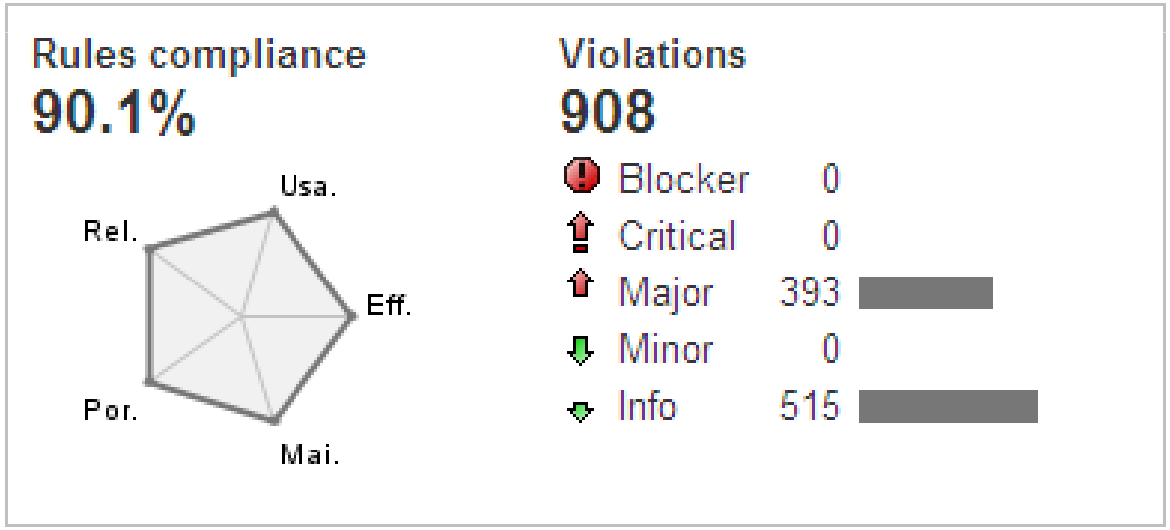
```
htmlDocument.append(original);

// load all comments from files and flush to output html
for (int i = 1; i < filepaths.size(); i++) {
        String comment = StringLoader.loadString(filepaths.get(i), true);
        htmlDocument.append("<h2>[Comment " + i + "]</h2>");
        htmlDocument.append(comment);
        comments.add(comment);
}

// for each comment diff with original and apply changes
for (int i = 1; i <= comments.size(); i++) {
        String current = comments.get(i - 1);
        List<String> temp = new ArrayList<String>();
        temp.add(original);
        temp.add(current);
        long start = System.nanoTime();
        Differ differ = new Differ(temp);
```

Metric: Violations(10/10)

- PDM, Findbugs, Checkstyle Violations
- Is the third part of Code category (11,25% of Total quality)
- Violations not in the “info” category.
- Sonar assumes no violations should be present
- Formula: Rules Compliance Percent



Thank you