Quality Metrics: GTE, CAP and CKJM

Kostis Kapelonis
Athens Greece, March 2010
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/

**Overall score: 66 (what is this?)**

- Excellent: 845 92.8%
- Good: 34 3.7%
- Needs work: 32 3.5%

12/03/2010
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
- 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”)

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

This is included!

Total is 2 + CC of findTax()
- All global state is bad!
- Singleton is an anti-pattern!

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

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

12/03/2010
Minimal constructors (fast and simple)!
No object initialization!

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

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

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

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

```java
public void calculate(List<Salary>) salaries) {
    calculateTax(salaries);
}
```
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
Main Course
Code analysis plugin
CAP Description

- Eclipse Plugin ([http://cap.xore.de/](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
Cap Usage

- Install CAP from the Eclipse Update site
  - [http://cap.xore.de/update](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)
- 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 Example]

![Bad Example]
- Main view shows package dependencies (imports)
- Same information as JDepend
- Shows package selected from the tree (Screen 1)
- Right view shows numerical info
- Same information as JDepend
- Abstractness, instability and Distance
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
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

Client/UI

Services/Middleware

Core/Model
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!
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)
Distance Example (5/5)

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

![Graph showing a distance example]

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

![Graph showing total quality and metrics](image)
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

<table>
<thead>
<tr>
<th>Package</th>
<th># Classes</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Packages</td>
<td>55</td>
<td>73%</td>
<td>64%</td>
<td>2.019</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.ant</td>
<td>11</td>
<td>52%</td>
<td>43%</td>
<td>1.848</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.check</td>
<td>3</td>
<td>0%</td>
<td>0%</td>
<td>2.429</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.coveragedata</td>
<td>13</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.instrument</td>
<td>10</td>
<td>90%</td>
<td>75%</td>
<td>1.054</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.merge</td>
<td>1</td>
<td>88%</td>
<td>88%</td>
<td>5.5</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting</td>
<td>3</td>
<td>87%</td>
<td>80%</td>
<td>2.002</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html</td>
<td>4</td>
<td>91%</td>
<td>77%</td>
<td>4.444</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.html files</td>
<td>1</td>
<td>87%</td>
<td>62%</td>
<td>4.5</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.reporting.xml</td>
<td>1</td>
<td>100%</td>
<td>95%</td>
<td>1.524</td>
</tr>
<tr>
<td>net.sourceforge.cobertura.util</td>
<td>9</td>
<td>60%</td>
<td>65%</td>
<td>2.892</td>
</tr>
<tr>
<td>someotherpackage</td>
<td>1</td>
<td>83%</td>
<td>N/A</td>
<td>1.2</td>
</tr>
</tbody>
</table>
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

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

### Summary

<table>
<thead>
<tr>
<th>Package</th>
<th>TC</th>
<th>CC</th>
<th>AC</th>
<th>Ca</th>
<th>Cn</th>
<th>A</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>cno.displaytag</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>0.0%</td>
<td>25.0%</td>
<td>75.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.decorator</td>
<td>13</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>13</td>
<td>38.0%</td>
<td>72.0%</td>
<td>11.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.exception</td>
<td>14</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>14.0%</td>
<td>46.0%</td>
<td>40.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.exception</td>
<td>16</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>17</td>
<td>31.0%</td>
<td>94.0%</td>
<td>26.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.filter</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>17.0%</td>
<td>100.0%</td>
<td>17.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.localization</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>18</td>
<td>33.0%</td>
<td>90.0%</td>
<td>23.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>cno.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Package</th>
<th>TC</th>
<th>CC</th>
<th>AC</th>
<th>Ca</th>
<th>Cn</th>
<th>A</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>core.displaytag</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>3</td>
<td></td>
<td>0.0%</td>
<td>25.0%</td>
<td>75.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.decorator</td>
<td>13</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>13</td>
<td>38.0%</td>
<td>72.0%</td>
<td>11.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.exception</td>
<td>14</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>14.0%</td>
<td>46.0%</td>
<td>40.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.export</td>
<td>16</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>17</td>
<td>31.0%</td>
<td>94.0%</td>
<td>25.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.lifter</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>17.0%</td>
<td>100.0%</td>
<td>17.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.localization</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>18</td>
<td>33.0%</td>
<td>90.0%</td>
<td>23.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.modal</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>72.0%</td>
<td>28.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.mail</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>20.0%</td>
<td>80.0%</td>
<td>0.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.properties</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>17</td>
<td>0.0%</td>
<td>71.0%</td>
<td>29.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.renderer</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>16</td>
<td>50.0%</td>
<td>69.0%</td>
<td>39.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytagongoose</td>
<td>12</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>24</td>
<td>17.0%</td>
<td>92.0%</td>
<td>9.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.topo</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>1</td>
</tr>
<tr>
<td>core.displaytag.uthl</td>
<td>18</td>
<td>14</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>22.0%</td>
<td>68.0%</td>
<td>10.0%</td>
<td>1</td>
</tr>
</tbody>
</table>
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
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

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

Summary

<table>
<thead>
<tr>
<th>Package</th>
<th>TC</th>
<th>CC</th>
<th>AC</th>
<th>Ce</th>
<th>Ce</th>
<th>A</th>
<th>I</th>
<th>D</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>pro.display</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>0.0%</td>
<td>25.0%</td>
<td>75.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_decorator</td>
<td>13</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>13</td>
<td>38.0%</td>
<td>72.0%</td>
<td>11.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_exception</td>
<td>14</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>14.0%</td>
<td>40.0%</td>
<td>40.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_expert</td>
<td>16</td>
<td>11</td>
<td>5</td>
<td>1</td>
<td>17</td>
<td>31.0%</td>
<td>94.0%</td>
<td>26.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_filter</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>17.0%</td>
<td>100.0%</td>
<td>17.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_intrusion</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>18</td>
<td>23.0%</td>
<td>90.0%</td>
<td>23.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_model</td>
<td>9</td>
<td>5</td>
<td>0</td>
<td>5</td>
<td>13</td>
<td>0.0%</td>
<td>70.0%</td>
<td>20.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_util</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>20.0%</td>
<td>80.0%</td>
<td>0.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_properties</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>17</td>
<td>0.0%</td>
<td>71.0%</td>
<td>29.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_render</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>16</td>
<td>50.0%</td>
<td>89.0%</td>
<td>30.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_test</td>
<td>12</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>17.0%</td>
<td>50.0%</td>
<td>5.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_ui</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>1</td>
</tr>
<tr>
<td>pro.display_udf</td>
<td>18</td>
<td>14</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>22.0%</td>
<td>68.0%</td>
<td>10.0%</td>
<td>1</td>
</tr>
</tbody>
</table>

TRASYS
WE GET IT DONE
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 Documentation
- 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

```java
/**
 * 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**

<table>
<thead>
<tr>
<th>File</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>eu.emea.pim.gul.applicant.ApplicantMergeEngine.java</td>
<td>99</td>
</tr>
<tr>
<td>eu.emea.pim.gul.DiffMergeEngine.java</td>
<td>59</td>
</tr>
</tbody>
</table>

```java
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();
   DiffSet differ = new DiffSet(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