Software Quality – 2nd Assigment

Rita Macedo
Faculdade de Ciências e Tecnologia
Departamento de Informática
rp.macedo@campus.fct.unl.pt

Rodrigo Graças
Faculdade de Ciências e Tecnologia
Departamento de Informática
r.gracas@campus.fct.unl.pt

ABSTRACT
Reverse Engineering is an important process to analyze a system code in order to understand its organization and to find problems that can affect its usability and maintenance. In this report, we apply this process with the help of tools to analyze two different systems, JHotDraw and JUnit. By measure metrics and visualize polymetric views, we begin to see the system organization and which software areas can be more vulnerable to bugs and errors. While going deeper into the analysis of the system, we find code smells and disharmonies and propose ways of refactoring.

1. INTRODUCTION
In this report we are going to analyze two different types of software programmed in Java in order to understand their quality in terms of software design. The systems are JHotDraw and JUnit4.

First, we are going to measure their most important metrics and visualize them as an overview Pyramid (the explanation and theory about the pyramid can be found in **[2]**). Next, we are going to visualize the systems using treemap views to better understand their organization. Last, we are going to analyze these systems in terms of disharmonies and code smells. With this analysis, we also propose some refactoring techniques that can be applied in order to improve the overall design of these systems.

According to the twenty-two code smells presented by Fowler **[1]** we focus on analyzing six of them. The code smells with more focus on this work are: *Duplicated Code*, *Long Method*, *Data Class*, *Refuse Bequest*, *Shotgun Surgery* and *Feature Envy*. We will also analyze disharmony presented in **[3]** – three identity disharmony, *God Class*, *Brain class* and *Brain Method*, two collaboration disharmony, *Dispersed Coupling* and *Intensive Coupling* and one Classification Disharmony, *Tradition Breaker*.

We also aim to analyze some Type Checking problems that can be present in the systems, but also for other types of errors, like quality or convention errors.

With this, we aim to identify the systems strong and weak points and propose improvements on the problems we capture on our way. This analysis will also be responsible to find out how these systems could be better in terms of reuse and maintainability.

The report is divided into three chapters and it is structured as follow:

- In Chapter 2 we present the main tools used to help us identify the code smells and important metrics evaluation of the systems;
- In Chapter 3 we present one of the systems used - JHotDraw (version 5.2). In here we analyze the overview pyramid, system's organization, identification of code smells and disharmonies using the tools (presented in Chapter 2), and refactoring.
- Chapter 4 presents the second system – JUnit (version 4). Similar to Chapter 3, we also present the overview pyramid and system's organization, identification of code smells, disharmonies and refactoring.

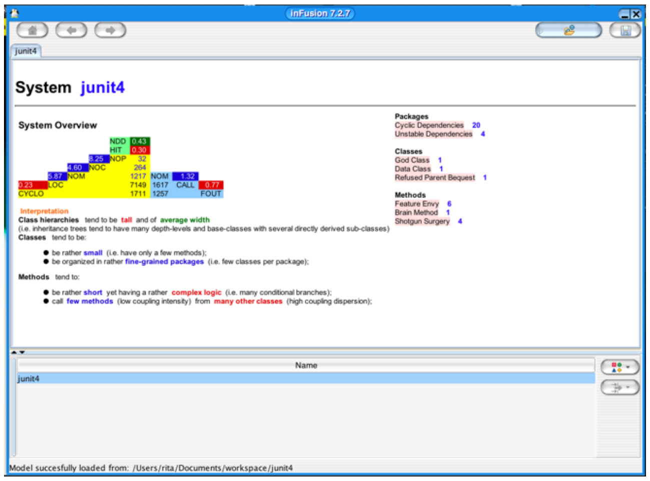
2. TOOLS USED
To help us analyze the two proposed systems, in terms of metrics and organization we are going to use three different tools, they are:

- infusion;
- Metrics;
- SourceMiner.

After that we will use a few tools that helps analyze code and find disharmonies, code smells or simple errors in code. These tools were used together to complement each other and provide the best results possible. These tools are:

- JDeodorant;
- CodeProAnalytiX;
- SonarLint;
- JspIRIT;

2.1 inFusion
inFusion is a stand-alone software that calculates the most important metrics of a given objected-oriented system and returns them as an Overview Pyramid and a resumed interpretation.



System junit4

System Overview

Interpretation
Class hierarchies tend to be tall and of average width
(i.e. inheritance trees tend to have many depth-levels and base-classes with several directly derived sub-classes)
Classes tend to be:

- be rather small (i.e. have only a few methods);
- be organized in rather fine-grained packages (i.e. few classes per package);

Methods tend to:

- be rather short yet having a rather complex logic (i.e. many conditional branches);
- call few methods (low coupling intensity) from many other classes (high coupling dispersion);

Packages	
Cyclic Dependencies	20
Unstable Dependencies	4
Classes	
God Class	1
Data Class	1
Refused Parent Bequest	1
Methods	
Feature Envy	6
Brain Method	1
Shotgun Surgery	4

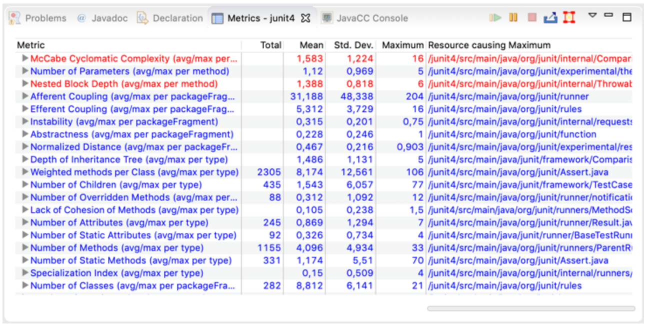
junit4

Model succesfully loaded from: /Users/rita/Documents/workspace/junit4

Figure 1. inFusion interface example

2.2 Metrics

Metrics is a simple plug-in software for eclipse that analyses a system and returns a list of important metrics for the evaluation of the system.



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum
McCabe Cyclomatic Complexity (avg/max per...		1,583	1,224	16	/junit4/src/main/java/org/junit/internal/Compar
Number of Parameters (avg/max per method)		1,12	0,969	5	/junit4/src/main/java/org/junit/experimental/the
Nested Block Depth (avg/max per method)		1,388	0,818	6	/junit4/src/main/java/org/junit/internal/Throwat
Afferent Coupling (avg/max per packageFrag...		31,188	48,338	204	/junit4/src/main/java/org/junit/runner
Efferent Coupling (avg/max per packageFrag...		5,312	3,729	16	/junit4/src/main/java/org/junit/rules
Instability (avg/max per packageFragment)		0,315	0,201	0,75	/junit4/src/main/java/org/junit/internal/request
Abstractness (avg/max per packageFragment)		0,228	0,246	1	/junit4/src/main/java/org/junit/function
Normalized Distance (avg/max per packageFr...		0,467	0,216	0,903	/junit4/src/main/java/org/junit/experimental/res
Depth of Inheritance Tree (avg/max per type)		1,486	1,131	5	/junit4/src/main/java/junit/framework/Comparis
Weighted methods per Class (avg/max per type)	2305	8,174	12,561	106	/junit4/src/main/java/org/junit/Assert.java
Number of Children (avg/max per type)	435	1,543	6,057	77	/junit4/src/main/java/junit/framework/TestCase
Number of Overridden Methods (avg/max per...	88	0,312	1,092	12	/junit4/src/main/java/org/junit/runner/notificati
Lack of Cohesion of Methods (avg/max per type)		0,105	0,238	1,5	/junit4/src/main/java/org/junit/runners/MethodS
Number of Attributes (avg/max per type)	245	0,869	1,294	7	/junit4/src/main/java/org/junit/runner/Result.ja
Number of Static Attributes (avg/max per type)	92	0,326	0,734	4	/junit4/src/main/java/junit/runner/BaseTestRun
Number of Methods (avg/max per type)	1155	4,096	4,934	33	/junit4/src/main/java/org/junit/runners/ParentR
Number of Static Methods (avg/max per type)	331	1,174	5,51	70	/junit4/src/main/java/org/junit/Assert.java
Specialization Index (avg/max per type)		0,15	0,509	4	/junit4/src/main/java/org/junit/internal/runners/
Number of Classes (avg/max per packageFra...	282	8,812	6,141	21	/junit4/src/main/java/org/junit/rules

Figure 2. Interface of Metrics as a plugin in eclipse

2.3 SourceMiner

SourceMiner is a multiple view environment (MVE) design and implemented as an Eclipse plug-in to enhance software comprehension activities. It will be used to create treemap views of the two software in analysis.

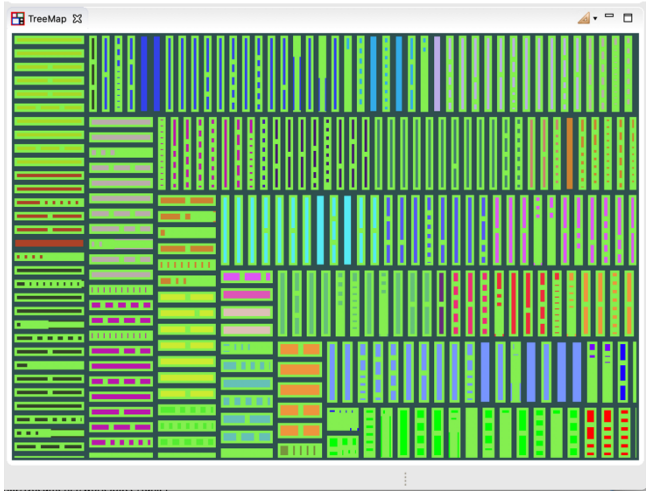
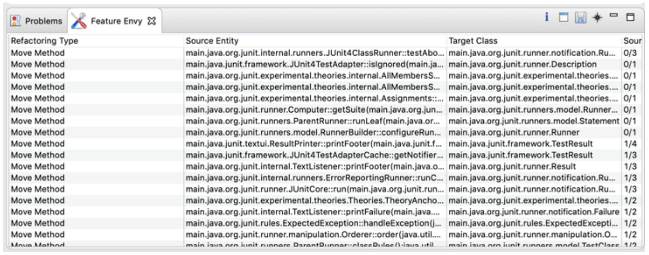


Figure 3. Example of a Tree Map view with SourceMiner

2.4 JDeodorant

JDeodorant is an eclipse plugin that identifies five specific code smells – *Duplicated Code, God Class, Long Method, Type Checking* and *Feature Envy* – and advices on the best course of action, the best refactoring.

In this work this tool was specifically used to find *God Classes, Long Methods, Type Checking* and *Feature Envy*.



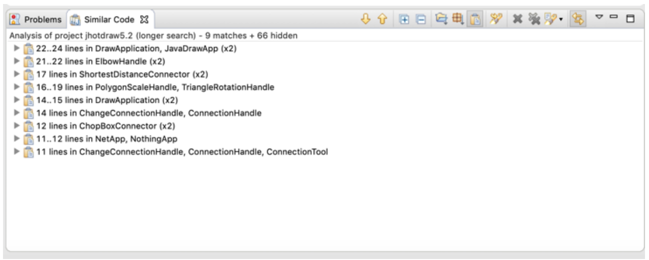
Refactoring Type	Source Entity	Target Class	Sour
Move Method	main.java.org.junit.internal.runners.JUnit4ClassRunner::testAbo...	main.java.org.junit.runner.notification.Ru...	0/3
Move Method	main.java.junit.framework.JUnit4TestAdapter::isIgnored(main.ja...	main.java.org.junit.runner.Description	0/1
Move Method	main.java.org.junit.experimental.theories.internal.AllMembersS...	main.java.org.junit.experimental.theories....	0/1
Move Method	main.java.org.junit.experimental.theories.internal.AllMembersS...	main.java.org.junit.experimental.theories....	0/1
Move Method	main.java.org.junit.experimental.theories.internal.Assignments::...	main.java.org.junit.experimental.theories....	0/1
Move Method	main.java.org.junit.runner.Computer::getSuite(main.java.org.jun...	main.java.org.junit.runners.model.Runner...	0/1
Move Method	main.java.org.junit.runners.ParentRunner::runLeaf(main.java.or...	main.java.org.junit.runners.model.Statement	0/1
Move Method	main.java.org.junit.runners.model.RunnerBuilder::configureRun...	main.java.org.junit.runner.Runner	0/1
Move Method	main.java.junit.textui.ResultPrinter::printFooter(main.java.junit.f...	main.java.junit.framework.TestResult	1/4
Move Method	main.java.junit.framework.JUnit4TestAdapterCache::getNotifier...	main.java.junit.framework.TestResult	1/3
Move Method	main.java.org.junit.internal.TextListener::printFooter(main.java.o...	main.java.org.junit.runner.Result	1/3
Move Method	main.java.org.junit.internal.runners.ErrorReportingRunner::runC...	main.java.org.junit.runner.notification.Ru...	1/3
Move Method	main.java.org.junit.runner.JUnitCore::run(main.java.org.junit.run...	main.java.org.junit.runner.notification.Ru...	1/3
Move Method	main.java.org.junit.experimental.theories.Theories.TheoryAncho...	main.java.org.junit.experimental.theories....	1/2
Move Method	main.java.org.junit.internal.TextListener::printFailure(main.java....	main.java.org.junit.runner.notification.Failure	1/2
Move Method	main.java.org.junit.rules.ExpectedException::handleException(j...	main.java.org.junit.rules.ExpectedExcepti...	1/2
Move Method	main.java.org.junit.runner.manipulation.Orderer::order(java.util....	main.java.org.junit.runner.manipulation.O...	1/2

Figure 4. View of Feature Envy code smell in JDeodorant

2.5 CodePro AnalytiX

CodePro Analytix is an Eclipse plugin that provides a set of analysis tools – code audit, metrics, test generation, JUnit test editing, code coverage and team collaboration features and functionality – to facilitate the analysis and refactoring of the code.

This tool was used in this project to find similar code.

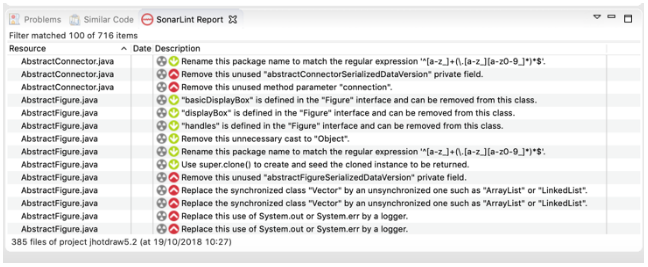


```
Analysis of project jhotdraw5.2 (longer search) - 9 matches + 66 hidden
▶ 22..24 lines in DrawApplication, JavaDrawApp (x2)
▶ 21..22 lines in ElbowHandle (x2)
▶ 17 lines in ShortestDistanceConnector (x2)
▶ 16..19 lines in PolygonScaleHandle, TriangleRotationHandle
▶ 14..15 lines in DrawApplication (x2)
▶ 14 lines in ChangeConnectionHandle, ConnectionHandle
▶ 12 lines in ChopBoxConnector (x2)
▶ 11..12 lines in NetApp, NothingApp
▶ 11 lines in ChangeConnectionHandle, ConnectionHandle, ConnectionTool
```

Figure 5. Results for similar code in CodePro Analytix

2.6 SonarLint

SonarLint is an IDE extension, in this case used in eclipse, that helps detect and fix quality issues in code.



Filter matched 100 of 716 items

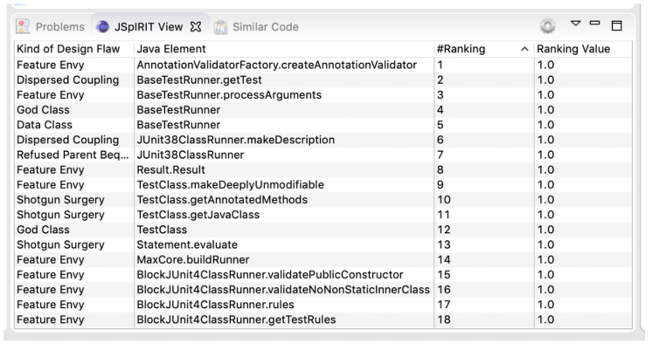
Resource	Date	Description
AbstractConnector.java		Rename this package name to match the regular expression '^[a-z_]+(\.[a-z_][a-z0-9_]*)*$'.
AbstractConnector.java		Remove this unused "abstractConnectorSerializedDataVersion" private field.
AbstractConnector.java		Remove this unused method parameter "connection".
AbstractFigure.java		"basicDisplayBox" is defined in the "Figure" interface and can be removed from this class.
AbstractFigure.java		"displayBox" is defined in the "Figure" interface and can be removed from this class.
AbstractFigure.java		"handles" is defined in the "Figure" interface and can be removed from this class.
AbstractFigure.java		Remove this unnecessary cast to "Object".
AbstractFigure.java		Rename this package name to match the regular expression '^[a-z_]+(\.[a-z_][a-z0-9_]*)*$'.
AbstractFigure.java		Use super.clone() to create and seed the cloned instance to be returned.
AbstractFigure.java		Remove this unused "abstractFigureSerializedDataVersion" private field.
AbstractFigure.java		Replace the synchronized class "Vector" by an unsynchronized one such as "ArrayList" or "LinkedList".
AbstractFigure.java		Replace the synchronized class "Vector" by an unsynchronized one such as "ArrayList" or "LinkedList".
AbstractFigure.java		Replace this use of System.out or System.err by a logger.
AbstractFigure.java		Replace this use of System.out or System.err by a logger.

385 files of project jhotdraw5.2 (at 19/10/2018 10:27)

Figure 6. Example of a sonarLint report

2.7 JSpIRIT

JSpiRIT is an eclipse plugin that helps identifying 10 code smells presented by Lanza et al. [3]. This code smells are *Brain Class, Brain Method, Data Class, Disperse Coupling, Feature Envy, God Class, Intensive Coupling, Refused Bequest, Shotgun Surgery* and *Tradition Breaker*. More about this tool can be read in **[8]**.



Kind of Design Flaw	Java Element	#Ranking	Ranking Value
Feature Envy	AnnotationValidatorFactory.createAnnotationValidator	1	1.0
Dispersed Coupling	BaseTestRunner.getTest	2	1.0
Feature Envy	BaseTestRunner.processArguments	3	1.0
God Class	BaseTestRunner	4	1.0
Data Class	BaseTestRunner	5	1.0
Dispersed Coupling	JUnit38ClassRunner.makeDescription	6	1.0
Refused Parent Beq...	JUnit38ClassRunner	7	1.0
Feature Envy	Result.Result	8	1.0
Feature Envy	TestClass.makeDeeplyUnmodifiable	9	1.0
Shotgun Surgery	TestClass.getAnnotatedMethods	10	1.0
Shotgun Surgery	TestClass.getJavaClass	11	1.0
God Class	TestClass	12	1.0
Shotgun Surgery	Statement.evaluate	13	1.0
Feature Envy	MaxCore.buildRunner	14	1.0
Feature Envy	BlockJUnit4ClassRunner.validatePublicConstructor	15	1.0
Feature Envy	BlockJUnit4ClassRunner.validateNoNonStaticInnerClass	16	1.0
Feature Envy	BlockJUnit4ClassRunner.rules	17	1.0
Feature Envy	BlockJUnit4ClassRunner.getTestRules	18	1.0

Figure 7. Example of JSpIRIT view of code smells

3. JHotDraw5.2

JHotDraw is a Java GUI framework to make technical and structured Graphics.

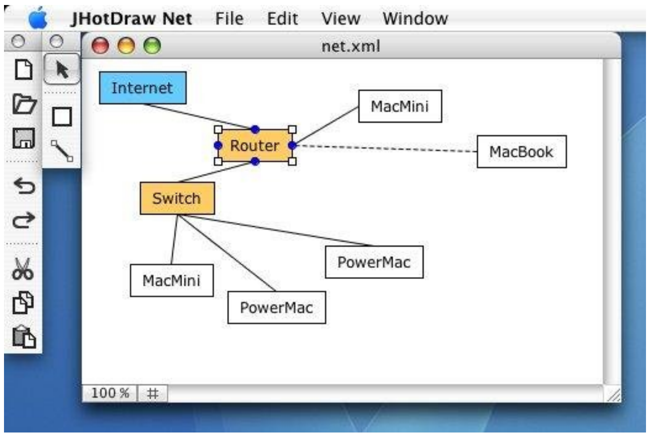


Figure 8. JHotDraw interface example

3.1 Overview Pyramid and metrics

With inFusion we produced an overview Pyramid for JHotDraw as seen next:

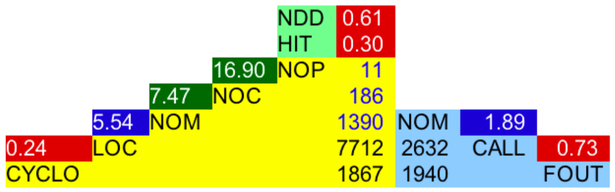


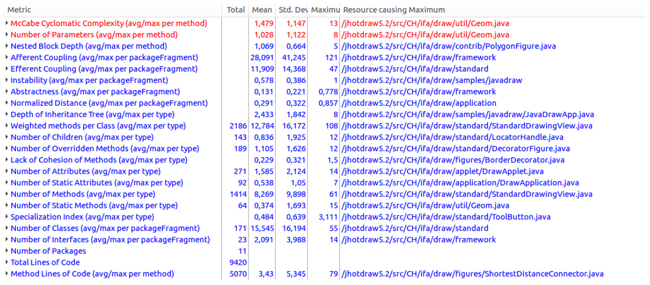
Figure 9. Overview Pyramid of JHotDraw 5.2 obtained with infusion

From this Overview Pyramid we can interpret that:

- Class hierarchies tend to be tall and wide. (i.e. inheritance tree tends to have many depth-levels and base-classes with many derived sub-classes).
- Classes tend to:
    - Contain an average number of methods;
    - Be organized in average-sized packages;
- Methods tend to:
    - Be rather short yet having a rather complex logic (i.e. many conditional branches);
    - Call few methods (low coupling intensity) from many other classes (high coupling dispersion);

If we run Metrics with JHotDraw system, we can also identify some problems and minor improvements that can be worth analyze and address. According to this plugin evaluation, there are two main problems in the method *intersect* from the Geom class:

- A McCabe Cyclomatic Complexity of 13, which is a moderate risk for possible method bugs.
- And a significantly high number of parameters in one method, in this case, 8 parameters.



Metric	Total	Mean	Std. Dev	Maximu	Resource causing Maximum
McCabe Cyclomatic Complexity (avg/max per method)		1,479	1,147	13	/jhotdraw5.2/src/CH/ifa/draw/util/Geom.java
Number of Parameters (avg/max per method)		1,028	1,122	8	/jhotdraw5.2/src/CH/ifa/draw/util/Geom.java
Nested Block Depth (avg/max per method)		1,069	0,664	5	/jhotdraw5.2/src/CH/ifa/draw/contrib/PolygonFigure.java
Afferent Coupling (avg/max per packageFragment)		28,091	41,245	121	/jhotdraw5.2/src/CH/ifa/draw/framework
Efferent Coupling (avg/max per packageFragment)		11,909	14,368	47	/jhotdraw5.2/src/CH/ifa/draw/standard
Instability (avg/max per packageFragment)		0,578	0,386	1	/jhotdraw5.2/src/CH/ifa/draw/samples/javadraw
Abstractness (avg/max per packageFragment)		0,131	0,221	0,778	/jhotdraw5.2/src/CH/ifa/draw/framework
Normalized Distance (avg/max per packageFragment)		0,291	0,322	0,857	/jhotdraw5.2/src/CH/ifa/draw/application
Depth of Inheritance Tree (avg/max per type)		2,433	1,842	8	/jhotdraw5.2/src/CH/ifa/draw/samples/javadraw/JavaDrawApp.java
Weighted methods per Class (avg/max per type)	2186	12,784	16,172	108	/jhotdraw5.2/src/CH/ifa/draw/standard/StandardDrawingView.java
Number of Children (avg/max per type)	143	0,836	1,925	12	/jhotdraw5.2/src/CH/ifa/draw/standard/LocatorHandle.java
Number of Overridden Methods (avg/max per type)	189	1,105	1,626	12	/jhotdraw5.2/src/CH/ifa/draw/standard/DecoratorFigure.java
Lack of Cohesion of Methods (avg/max per type)		0,229	0,321	1,5	/jhotdraw5.2/src/CH/ifa/draw/figures/BorderDecorator.java
Number of Attributes (avg/max per type)	271	1,585	2,124	14	/jhotdraw5.2/src/CH/ifa/draw/applet/DrawApplet.java
Number of Static Attributes (avg/max per type)	92	0,538	1,05	7	/jhotdraw5.2/src/CH/ifa/draw/application/DrawApplication.java
Number of Methods (avg/max per type)	1414	8,269	9,898	61	/jhotdraw5.2/src/CH/ifa/draw/standard/StandardDrawingView.java
Number of Static Methods (avg/max per type)	64	0,374	1,693	15	/jhotdraw5.2/src/CH/ifa/draw/util/Geom.java
Specialization Index (avg/max per type)		0,484	0,639	3,111	/jhotdraw5.2/src/CH/ifa/draw/standard/ToolButton.java
Number of Classes (avg/max per packageFragment)	171	15,545	16,194	55	/jhotdraw5.2/src/CH/ifa/draw/standard
Number of Interfaces (avg/max per packageFragment)	23	2,091	3,988	14	/jhotdraw5.2/src/CH/ifa/draw/framework
Number of Packages	11				
Total Lines of Code	9420				
Method Lines of Code (avg/max per method)	5070	3,43	5,345	79	/jhotdraw5.2/src/CH/ifa/draw/figures/ShortestDistanceConnector.java

Figure 10. Metrics view of JHotDraw

3.2 System organization and distribution

This section aims to understand and visualize JHotDraw according to three dimensions: size, complexity and coupling. We will use Treemap view from SourceMiner, in order to reveal how well the systems are distributed across classes and packages.

Figures 11, 12 and 13 are the results of SourceMiner with the JHotDraw system. By analyzing the different views, we can state that the JHotDraw is a well distributed system in terms of size and complexity. There are more size variations with a relative class complexity. And, in terms of coupling, there is also a low coupling intensity with high coupling dispersion.

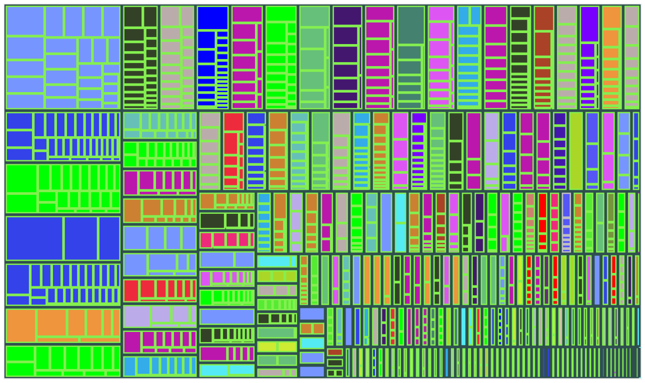


Figure 11. Treemap view of size dimension of JHotDraw

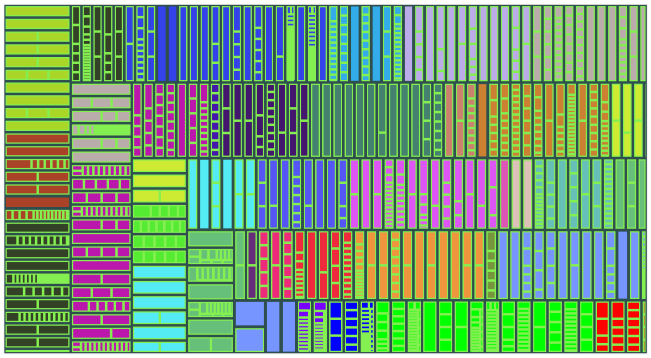


Figure 12. Treemap view of complexity dimension of JHotDraw

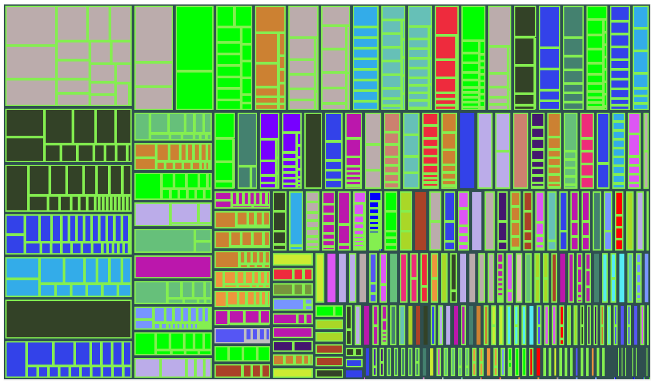


Figure 13. Treemap view of coupling dimension of JhotDraw

3.3 Disharmonies, Code Smells and their Refactorings

With the help of inFusion we were able to find some disharmonies and code smells that can affect the quality of this system:

- On the packages there is 1 unstable dependency;
- In relation to classes:
  - There are 2 *Data classes*;
  - There are 2 *Refuse Parent Bequest*.
- In relation to the method:
  - there are 80 *Feature Envy*;
  - there is 1 *Brain Method*;
  - It was found 1 *intensive coupling*;
  - There is 9 *shotgun surgery*.

Unfortunately, this tool doesn't tell us which classes, methods and packages have problems, so we will have to complement this information with ones from other tools, specifically with JDeodorant, JSpIRIT and Code Pro. In general, with this new tools we obtain results a lot different than the results given by inFusion but since this tools show which problems they refer to we could analyze de code and accept them or not.

There are a few problems in terms of evolution of software, these problems are related to a few code smells that can ease the creation of bugs and errors and few errors of conventions that can affect the understanding of the code.

Running the tools, we can right away say that this software has no *Long Methods*, no *Intensive Coupling*, no *Brain Method* and no *Brain Class*. This means that in this section we are going to analyze and refactor 8 smells or disharmonies:

- *Duplicated Code;*
- *FeatureEnvy;*
- *God Class;*
- *Tradition Breaker;*
- *Data Class;*
- *Refused Bequest;*
- *Shotgun Surgery;*
- *Dispersed Coupling*.

3.3.1 Duplicated Code

Using CodePro we were able to find sevem similar codes that given a closer look can be categorized as the *Duplicated Code* smell and for that reason can be refactored. These codes are:

1. In the class *JavaDrawApp* the methods *createAnimationMenu* and *createWindowMenu*;
2. In the class *ElbowHandler* the methods *constrainX* and *constrainY*;
3. In the class *ShortestDistanceConnector* the method *findPoint* has the same code repeated twice for the x-dimension and the y dimension.
4. The classes *PolygonScalehandle* and *TriangleRotationHandle* have each one a method called *getOrigin* that have few differences between each other.
5. In the classes *ChangeConnectionHandle* and *ConnectionHandle* there are two methods that are exactly the same in both – *invokeStep* and *findConnectableFigure*;
6. In the class *ChopBoxConnector* there are two methods very similar to each other, *findStart* and *findEnd.*

To refactor these codes, it would probably be adequate to use one of two types of refactoring, *Move Method* or *Extract Method*. For 5 this would be easy to do, we could simply extract the method for one class that the other two have in common and those two methods could simply call the new method. For 1, 2 and 6 the right thing to do could be creating a generic method that would receive the variables that are different in both presented methods and that could be called by them. For 3 the solution could be to create different method that would receive the variables and calculate the similar calculations. For 4 we could do the same thing as for 1, 2 and 6 but this generic method would have to be extracted to one class that both the other would have in common.

With SonarLint we also identified a code duplication where two methods in the same class (*AbstractFigure*) do exactly the same thing but have different names, one of them should be change or simple erased and substituted by the other one.

3.3.2 Feature Envy

In terms of *Feature Envy*, with JDeodorant we found two methods in class *PertFigure* that seem to use too much data from other classes (*writeTask* and *readTask*) and one method that seems to do the same in the class *FigureAttribute* (method *write*). These results are a lot different from the information given by inFusion since it says it has 80 Feature Envy, for that

reason to complement this mixed information we also run JSpIRIT that gives a result even bigger than inFusion – 130 other *Feature Envy* results.
Analyzing the methods given by JSpIRIT we can see that most of this method in fact use and alter variables from other classes and for that reason we can consider these 130 smells.
To correct this smell, we should use the refactoring *Move Method*. For the sake of understanding this refactoring we are not going to describe the refactoring of all the 133 smells, but we are going to exemplify with the results given by JDeodorant.
Since the method *writeTasks* and the method write make alterations in the class *StorableOutput* these methods should be moved to that class, with the exception that the method write keeps existing in the class *FigureAttribute* but calls the method with the same name of the class *StorableOutput*. The same happens for *readTask* but this method should be moved to the class *StorableInput*.

3.3.3 *God Class*

10 *God Classes* were found in JHotDraw with JDeodorant which is once again a bit different of what inFusion told us (it didn't identify any god class), complementing this information with the one given by JSpIRIT we found 7 *God Classes*, two of them being the same in both tools. To make sure all of them are God Classes we analyzed the code and see that they can potentially be God Classes. These classes are: *Vector, DrawApplet, FloatingTextField, StorableOutput, StorageFormatManager, CompositeFigure, DrawApplication, TextFigure, NodeFigure Iconkit, PolyLineFigure,StandardDrawingView, LineConnection, PertFigure and ConnectionTool*.
To refactor this classes, we will use *Extract Class*, which means, we will break a *God Class* into one or more classes that have specific objectives, taken a weight from the original class. Three examples of this refactoring are presented in **Figure. 14, Figure 15 and Figure 16**.

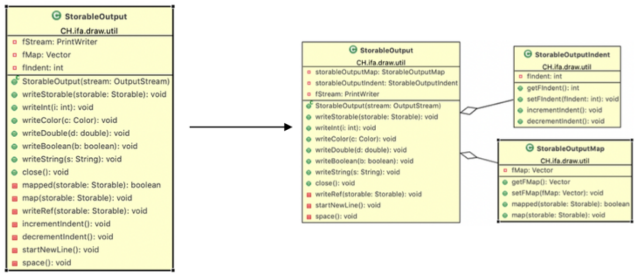

StorableOutput
CH.ifa.draw.util
fStream: PrintWriter
fMap: Vector
fIndent: int
StorableOutput(stream: OutputStream)
writeStorable(storable: Storable): void
writeInt(i: int): void
writeColor(c: Color): void
writeDouble(d: double): void
writeBoolean(b: boolean): void
writeString(s: String): void
close(): void
mapped(storable: Storable): boolean
map(storable: Storable): void
writeRef(storable: Storable): void
incrementIndent(): void
decrementIndent(): void
startNewLine(): void
space(): void
StorableOutput
CH.ifa.draw.util
storableOutputMap: StorableOutputMap
storableOutputIndent: StorableOutputIndent
fStream: PrintWriter
StorableOutput(stream: OutputStream)
writeStorable(storable: Storable): void
writeInt(i: int): void
writeColor(c: Color): void
writeDouble(d: double): void
writeBoolean(b: boolean): void
writeString(s: String): void
close(): void
writeRef(storable: Storable): void
startNewLine(): void
space(): void
StorableOutputIndent
CH.ifa.draw.util
fIndent: int
getFIndent(): int
setFIndent(fIndent: int): void
incrementIndent(): void
decrementIndent(): void
StorableOutputMap
CH.ifa.draw.util
fMap: Vector
getFMap(): Vector
setFMap(fMap: Vector): void
mapped(storable: Storable): boolean
map(storable: Storable): void

Figure 14. Refactoring of the class StorageOutput

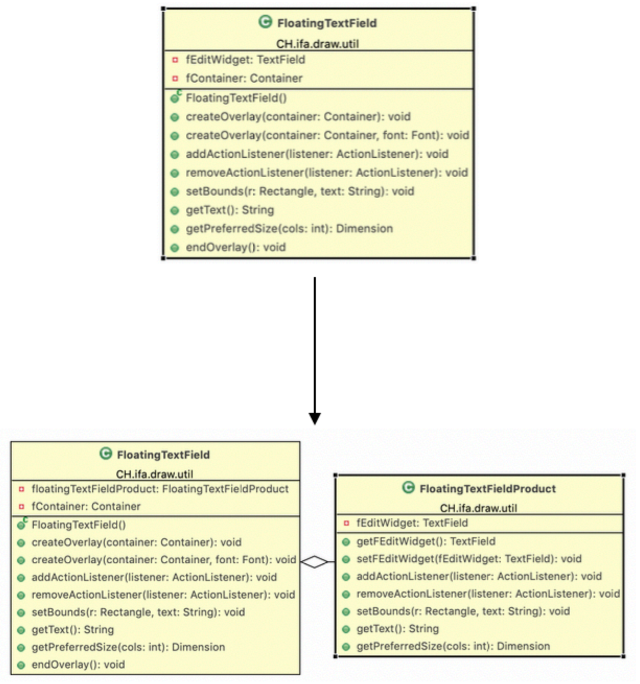

FloatingTextField
CH.ifa.draw.util
fEditWidget: TextField
fContainer: Container
FloatingTextField()
createOverlay(container: Container): void
createOverlay(container: Container, font: Font): void
addActionListener(listener: ActionListener): void
removeActionListener(listener: ActionListener): void
setBounds(r: Rectangle, text: String): void
getText(): String
getPreferredSize(cols: int): Dimension
endOverlay(): void
FloatingTextField
CH.ifa.draw.util
floatingTextFieldProduct: FloatingTextFieldProduct
fContainer: Container
FloatingTextField()
createOverlay(container: Container): void
createOverlay(container: Container, font: Font): void
addActionListener(listener: ActionListener): void
removeActionListener(listener: ActionListener): void
setBounds(r: Rectangle, text: String): void
getText(): String
getPreferredSize(cols: int): Dimension
endOverlay(): void
FloatingTextFieldProduct
CH.ifa.draw.util
fEditWidget: TextField
getFEditWidget(): TextField
setFEditWidget(fEditWidget: TextField): void
addActionListener(listener: ActionListener): void
removeActionListener(listener: ActionListener): void
setBounds(r: Rectangle, text: String): void
getText(): String
getPreferredSize(cols: int): Dimension

Figure 15. Refactoring of the class FloatingTextField

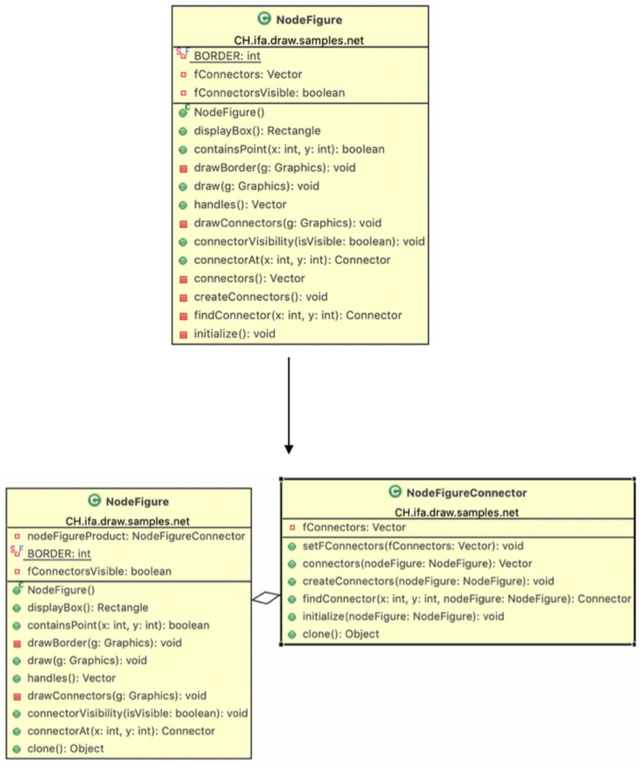

NodeFigure
CH.ifa.draw.samples.net
BORDER: int
fConnectors: Vector
fConnectorsVisible: boolean
NodeFigure()
displayBox(): Rectangle
containsPoint(x: int, y: int): boolean
drawBorder(g: Graphics): void
draw(g: Graphics): void
handles(): Vector
drawConnectors(g: Graphics): void
connectorVisibility(isVisible: boolean): void
connectorAt(x: int, y: int): Connector
connectors(): Vector
createConnectors(): void
findConnector(x: int, y: int): Connector
initialize(): void
NodeFigure
CH.ifa.draw.samples.net
nodeFigureProduct: NodeFigureConnector
BORDER: int
fConnectorsVisible: boolean
NodeFigure()
displayBox(): Rectangle
containsPoint(x: int, y: int): boolean
drawBorder(g: Graphics): void
draw(g: Graphics): void
handles(): Vector
drawConnectors(g: Graphics): void
connectorVisibility(isVisible: boolean): void
connectorAt(x: int, y: int): Connector
clone(): Object
NodeFigureConnector
CH.ifa.draw.samples.net
fConnectors: Vector
setFConnectors(fConnectors: Vector): void
connectors(nodeFigure: NodeFigure): Vector
createConnectors(nodeFigure: NodeFigure): void
findConnector(x: int, y: int, nodeFigure: NodeFigure): Connector
initialize(nodeFigure: NodeFigure): void
clone(): Object

Figure 16. Refactoring of the class NodeFigure

3.3.4 *Tradition Breaker*

We found one *Tradition Breaker* with JSpIRIT (class *CompositFigure*) which if we analyze a little bit further, we can see that it is probably right, since the class doesn't specialize a lot of the inherited methods and mostly only adds brand new services that don't depend on the inherited functionality.

Looking at the inspection and refactoring process of a *Tradition Breake*r in **[3]** it seems right to break this class in two, extracting the independent parts as a new class.

3.3.5 Data Class

7 *Data Classes* where found with JSpIRIT (*Figure,TextFigure, PolyLineFigure, StandardDrawingView, Geom, DrawingView* and *DrawApplication*) but it is somewhat difficult to understand if they are actually *Data Class*. In our point of view this classes do a lot more than just hold data and for that reason they should not be refactored at this point.

3.3.6 Refused Bequest

In relation to the smell *Refused Bequest* JSpIRIT found 26 results (*TextTool, StandardDrawing, PolygonFigure, TextFigure, MDI_DrawApplication, SplitPaneDrawApplication, NodeFigure, PolyLineFigure, RoundRectangleFigure, AnimationDecorator, LineConnection, JavaDrawApp, ChangeConnectionHandle, CustomSelectionTool, AtributeFigure, ConnectionHandle, GraphicalCompositeFigure, ToolButton, RelativeLocator,CompositeFigure, PolygonTool, PetFigure, ConnectionTool, URLTool, ImageFigure, TriangleFigure*).

Analyzing this classes further we can see that they do not refuse interfaces but refuse some implementations, so this is not a very strong smell in most of them since it's alright to use subclassing to reuse a bit of behavior. For example, the class *TextFigure* extends *AttributeFigure* and doesn't refuse the interface it simple rewrites some methods, uses other from the superclass and adds more methods than the original ones. For this reason, this smell is also not going to be refactored.

3.3.7 Shotgun Surgery

23 methods suffering from *Shotgun Surgery* where found:

- methods *displayBox, includes, canConnect,* and *getAttribute* from interface *Figure*;
- methods *drawing* and *view* from class *AbstractTool*;
- method *drawing, selectionCount, clearSelection,* and *checkDamage* from interface *DrawingView*;
- method *owner* from interface *Connector*;
- methods *writeStorable* and *writeInt* from class *StorableOutput*;
- method *view* from class *DrawApplication*;
- methods *readStorable* and *readInt* from class *StorableInput*;
- methods *listener, willChange* and *changed* from *AbstractFigure;*
- method *nextFigure* from *FigureEnumeration*;
- and methods owner and *displayBox* from class *AbstractHandle*.

Analyzing them further we can see that these methods are in fact called by many other classes and if change can affect a lot of other methods.

What we propose to solve this problem is to create a new class to where we would reunite all the methods connected to the original one so that the methods are all together and can be easily identified. If they are easily identified there is less probability that we forget to correct one method when the original one is changed.

3.3.8 Dispersed Coupling

It was found with SpIRIT 7 *Dispersed Couplings* (*UngroupCommand.execute, SelectionTool.mouseDown, ShortestDistanceConnector.findPoint, DrawApplication.print, PastCommand.execute, StandardLayouter.calculateLayout, FigureAtributes.write*). Most of these problems could imply that these methods are also *Brain Methods* but since we didn't identify any *Brain Methods* with the tools, we will consider that this methods are the rare cases in which the *Dispersed Coupling* is not connected a *Brain Method*.

To refactor these methods, we propose a closer look at them and trying to understand if they can't pass part of the method to the method he already invokes. Other option we propose, if the first one is not possible, is to create one or more methods to withdraw some of the weight of the original method.

3.3.9 Type Checking and other problems

Doing a Type Checking with JDeodorant we obtain two errors one in the class *ElbowConnection* and other the class *Execute*. In both of them the problem has to do with constants that affect the behavior of the method. The suggestion is to use a refactoring called *Replace Type Code with a State/Strategy*. The idea is that we can transform those constants in state objects.

Running SonarLint gives more errors and problems that can easily be corrected:

- Some packages are not name according to the convention and should be renamed;
- A few classes have private variables that end up not being used and for that reason should be erased;
- Some methods have parameters that are not used and should be removed;
- Some variables are defined in interfaces, so they shouldn't also be declared in the classes that implement those interfaces;
- There are unnecessary casts;
- There is a few synchronized class "Vector" that should be replaced by an. Unsynchronized one;
- System.out should be replaced by System.err;
- There is one deprecated code that should be erased one day;
- There are some missing deprecated notes;
- There should be nested comments to explain why some methods are empty;
- There are a few unordered modifiers;
- There is one switch that is missing the default case;
- There a few methods that should just inherit the same method of the superclass and not simply call it;

- There are casts missing;
- The is a commented line of code that should be removed;
- There is one case insensitive operation that doesn't need the *toLowerCase()* method;
- There are if statements that could be merged;
- There are "Integer" constructors that should be removed;
- The class *BoxHandleKit* should have a private constructor to hide. The. implicit public one;
- There are a few imports that are unused and should be removed;
- There are variables that are reintroduce instead of just using the ones that were already declared.

Some of these errors are minor bugs and smells that probably don't actually affect the performance of the program but should be altered so that the code complies with the conventions and is easier to understand by other programmer that can have the necessity to reuse or update the code.

4. JUnit4

JUnit is very popular unit testing framework for the Java programming language. It is widely used for the development of test-driven development.

4.1 Overview Pyramid and metrics

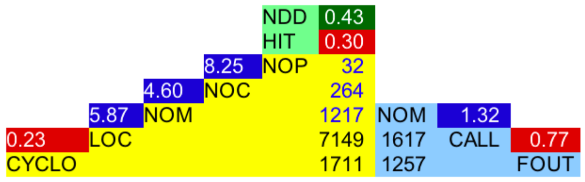


Figure 17. Overview Pyramid of JUnit4 obtained with infusion

From the Overview Pyramid represent before we can interpret that:

- Classes hierarchies tend to be tall and of average width (i.e. inheritance trees tend to have many depths-levels and base classes with several directly derived sub-classes);
- Classes tend to:
    - Be rather. small (i.e. have only. A few methods);
    - Be organized in fine-grained packages (i.e. few classes per package);
- Methods tend to:
    - Be rather short yet having a rather complex logic (i.e. many conditional branches);
    - Call few methods (low coupling intensity) from many other classes (high coupling dispersion);

4.2 System organization and distribution

This section aims to understand and visualize JUnit according to three dimensions: size, complexity and coupling. We will use Treemap view from SourceMiner, in order to reveal how well the systems are distributed across classes and packages.

In **Figures 18 and 19** we illustrate both views using size as dimension. With Treemap, we can see that classes tend to have few methods and methods tend to be short. This agrees with the interpretation of the overview pyramid, from the last section. With the Grid view, we can state that packages tend to have few classPolygones and different sizes.

Figures 20 and 21 reveals that JUnit, in terms of complexity, is well distributed across classes and are just a few that indicate slightly more complex operations. With the Grid view, we can state also that this complexity is well distributed across the project packages.

Figure 22 represents coupling dimension and in here we can state that JUnit is not a system that is intensively coupled by method calls, but methods tend to call many other methods from other classes, with a high coupling dispersion.

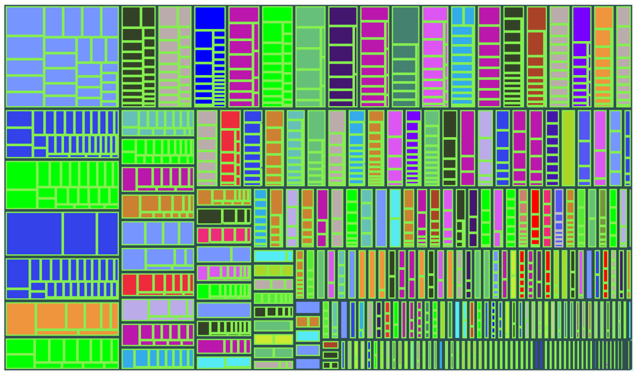


Figure 18. Treemap view of size dimension of JUnit4

Figure 19. Grid view of size dimension of JUnit4

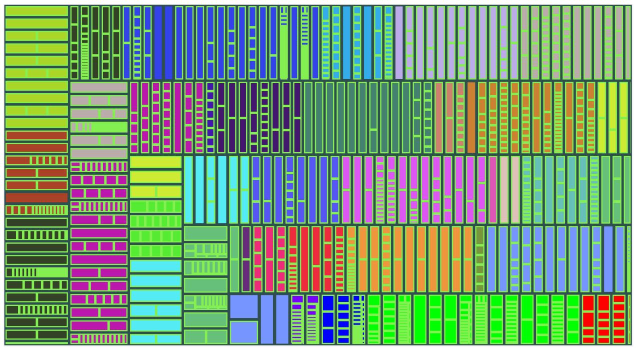


Figure 20. Treemap view of complexity dimension of JUnit4

Figure 21. Grid view of complexity dimension of JUnit4

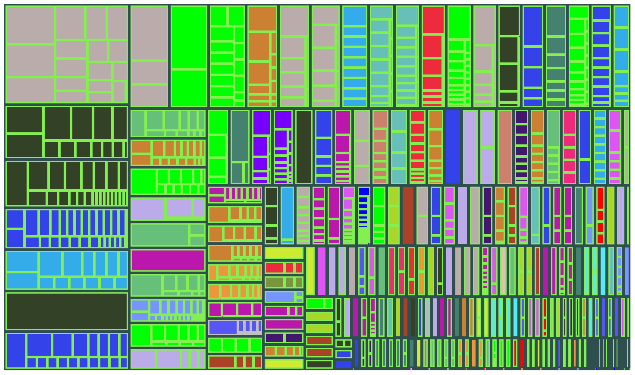


Figure 22. Treemap view of coupling dimension of JUnit4

4.3 Identity disharmonies and Code Smells

There are a few identity disharmonies that can affect the quality of this system. The disharmonies identified are as follow:

- In relation to packages there are:
  - 20 Cycling dependencies;
  - 4 unstable dependencies;
- In relation to classes there are:
  - 1 *God Class*;
  - 1 *Data class*;
  - 1 *Refuse Parent Bequest*;
- In relation to the method:
  - there are 6 *Feature Envy*;
  - there is 1 *Brain Method*;
  - There are 4 *Shotgun Surgery*.

As said for JHotDraw, the inFusion tool give us this information but doesn't tell us where are these concrete problems. To identify this exact problems, we will complement this information with the ones given by JDedodorant, JSpIRIT and Code Pro tools. Once again we will receive information from different sources and with so, expect different results from the ones given by inFusion, but we will analyze the code to validate all this information.

We will also run SonarLint to see if there are problems in terms of evolution of software, these problems are related to a few code smells that can ease the creation of bugs few errors of conventions that can affect the understanding of the code.

Running the tools, we can right away say that this software has no *Brain Method* and no *Brain Class*. This means that in this section we are going to analyze and refactor ten code smells or disharmonies:

- *Duplicated Code;*
- *Feature Envy;*
- *God Class;*
- *Tradition Breaker;*
- *Data Class;*
- *Refused Bequest;*
- *Shotgun Surgery;*
- *Dispersed Coupling;*
- *Intensive Coupling;*
- *Long Method.*

4.3.1 Duplicated Code

Using CodePro we were able to find 11 similar codes patterns. After inspecting each one of them, we filter the most similar ones, they are:

- both *runBefores()* and *runAfters()* methods from *ClassRoadie* and *MethodRoadie* classes;
- *getSingleDataPointFields()* and *getDataPointFields()* from *SpecificDataPointsSupplier* class;
- both *printFooter()* from *ResultPrinter* and *TextListener* classes;
- *evaluate()* from *ExternalResource* and *RunAfters* classes;
- *getAnnotatedFieldValues()* and *getAnnotatedMethodValues()* from *TestClass* class;
- *equals()* in *TestWithParameters* and *TestClass* class*;*
- and *getSingleDataPointFields()* and *getDataPointFields()* from *AllMembersSupplier* class.

These are all methods that have similar code and we should eliminate this duplication by using Extract Method followed by a *Pull Up Field* in 1,2 and 4 bullet lists. We can also consider *Extract Class* but in these cases, the classes are related, and we shouldn't add an extra class. In other bullet cases, we should just *Extract Method* since the code is quite similar but not exactly the same.

4.3.2 Feature Envy

By running JDeodorant in JUnit4, we found a serious of methods that are using other methods from other target classes. With the total of nineteen identified (contrary to inFusion results – six feature envy). But, if we explore the most target accessed features, we can see that the method *testAborted*from the *JunitClassRunner* class uses three methods from *RunNotifier* class. Similar, there are other few methods that use other class methods. For instance, *printFooter* (from *ResultPrinter* class) and *getNotifier* (from *JUnit4AdapterCache*) are methods that call a lot methods from the *TestResult* class. A lof of these methods are not identified by *inFusion*, since there are not critical. Furthermore, we've also compare these results against the JSpIRIT tool and it also identify the same problems and plus others.
Analyzing the different results methods from JspIRIT, we can identify that are still other methods (not identified by JDeodorant) that alter variables and call variables from other target classes. With so, we can consider the total amount of 33 code smells in the system.
For all this code smell problems, we have some refactoring opportunities that can be applied. The opportunity identified by JDeodorant is the Move Method type. To give an example for this type being applied, we can analyze the *runCause* method from *ErrorReportingRunner* class. This method body is just three method calls to get features from *RunNotifier* class. This method logic should be moved to the target class since it is just interested from the target class features.

4.3.3 God Class

A total of 23 possible *God Classes* are identified using JDeodorant. InFusion identified one single God Class. JspIRIT identified only 6. These six possible God classes are: *BlockJUnit4ClassRunner, Assert, TestCase, TestClass, BaseTestRUnner* and *ParentTestRunner.* After analyzing the source code of all this possible classes, the most likely class that performs and is responsible for the most system logic is the *Assert* class. This class is a long class, with a lot of methods and with a total cyclomatic complexity of 108. Since a *God class* is a complex class that can potentially bring disharmonies to the system, we should consider refactoring it. The refactor type to be used in this *God class* cases is the *Extract Class* type, which means that we should divide this class into multiple subclasses.

4.3.4 Tradition Breaker

Using JSpIRIT, we found two Tradition Breaker - *CategoryFilter* and *TemporaryFolder* classes. These classes extend the classes Suite and ExternalResource, respectively. After performing a code analysis, we can validate the two identified Traditional Breaker classes. These classes don't use all protected features from their parent classes and they add new services that don't depend on the inherited functionality.

4.3.5 Data Class

Contrary to one Data Class identified by inFusion, JSpIRIT identified two possible Data Classes – *TestRunner* and *BaseTestRunner*. According to Fowler's definition [1] for the Data Class code smell, these classes only have fields, getting and setting methods for these fields. If we analyze the source code for this two possible classes, in our opinion, we don't think that these classes apply to this definition. Yes, they are simple classes, with fields and get/set methods, but they also have other methods with extra functionality rather than retrieve the class fields.

4.3.6 Refused Bequest

If we run JSpIRIT, 16 *Refused Bequest* were identified. The classes are: *BlockJUnit4CLassRunner, Categories, TheoryAnchor, TestRunner, Theories, JUnit38ClassRunner, JUnit4ClassRunner, CategoryFilter, ErrorReportingRunner, TestCase, BlockJUNit4ClassRunnerWithParameters, AllMembersSupplier, TemporaryFolder, FrameworkMethod, FailOnTimeout* and *ParentRunner.*

For instance, the *TestCase* class only overrides two methods from their parent interface – *Test* - but this interface only has two methods so, we this is not a *Refused Parent Bequest* case. Other examples identified by the tools, are just possible cases for refused bequest cases and, if we inspect the hierarchies for these cases, it does not apply for Fowler's definition that some subclasses don't use the inherited methods from their parents.

Other classes inherit other abstract classes with just one method. This is also another case of a possible refused bequest, but if validate in the code and the methods inherited are implemented. With all this in mind, we don't propose refactoring.

4.3.7 Shotgun Surgery

In terms of the *Shotgun Surgery* code smell, JSpIRIT results found 6 possible cases. They are as follow:

- *getRunner()* from Request class;
- *evaluate()* from Statement class;
- *invokeExplosively()* and *getName()* from *FrameworkMethod* class;
- and *getAnnotatedMethods()* and *getJavaClass()* from *TestClass* class.

These are methods that are called by many other classes and their change can alter many classes. But, in our opinion, *evaluate()* method is not the case, since it is an abstract method, meaning that it implementation can be override and thus, not alter in a method alter chain "reaction". And so, this result is almost accurate as inFusion result, with 4 *Shotgun Surgery* cases.

Fowler refactoring technique is the solution here for the other cases. We propose to to create a new class where we would put all behavior changes. The idea should be to arrange things in order to have a common link between all changes.

4.3.8 Dispersed Coupling

With JSpIRIT we can identify the following 10 Dispersed Couplings:

- validateDataPointFields() and validadeDataPointMethods() from Theories class;
- makeDescription() from JUnit38ClassRunner class;
- validateTestMethods() from MethodValidator class;
- assertThrows() from Assert class.
- createTestUsingFieldInjection() from BlockJUnit4ClassRunnerWithParameters class;
- getTrimmedStackTraceLines() and getCauseStackTraceLines() from Throwables class;
- getTest() from BaseTestRunner class;
- and filter() from ParentRunner class

According to Lanza and Marinescu definition of dispersed coupling, these methods communicate with an excessive number of classes, whereby this communication calls just one or few methods [3]. This type of disharmony could imply the existence of Brain Methods. And *inFusion*, caught one and we can imply it is one of these methods. The detection strategy here would be to identify the methods that have a high dispersion of their respective coupling.

Refactoring these operations need a more precise information about the system but the strategy would be to identify the methods affected by the Shotgun Surgery (in 4.3.7), and move some of the method body to the methods it invokes. Before this, we should have a good understanding about the JUnit domain since we would add new abstractions to the system.

4.3.9 Intensive Coupling

This type of disharmony happens when a method is dependent from many other operations, where these operations are dispersed into two or more classes [3]. There are 5 cases of *Intensive Coupling* in the JUnit system, identified by JSpIRIT.

These disharmonies are:

- *validateTestMethods()* from Theories class;
- *addMultiPointMethods()* from *AllMembersSupplier* class;
- *runTestMethod()* from *MethodRoadie* class;
- *order()* from *ParentRunner* class;
- and *addTestsFromTestCase()* from *TestSuite* class.

These methods tend to call many methods, and they are dispersed in few classes throughout the system. In the JUnit system, the dispersion is not as low as expected and the refactoring to be applied in this case should be to create one class that will act as a provider for the multiple calls in different methods.

4.3.10 Long Method

To identify this code smell, we used JDEodorant. This tool identified 33 possible *Long Methods*. One example is the method *getTest()* from the *BaseTestRunneri* class. This method have almost 50 lines of codes and it should be a good solution to decompose this method. This is not the case for a method with lots of parameters (it only have one), so the *Extract Method* do not apply, but rather a good technique should be to identify for semantic distance between clumps of code. These clumps should be worth extracting. Other cases for refactoring should be consider and always have in mind to decompose into new methods.

4.3.11 Type Checking and other problems

Similar to JHotDraw, when we do a *Type Checking* with JDeodorant we obtain one problem that has to do with constants that affect the behavior of the method. This problem happens in the class *BlockJUnit4ClassRunnerWithParameters* on the method. *createTest*. The suggestion is to use a refactoring called *Replace Type Code with a State/Strategy*. The idea is that we can transform those constants in state objects.

To finalize the analysis of the code we run SonarLint to find some other problems that weren't identified by the other tools. These problems are:

- There is an *ObjectOutputStream* thar should be closed;
- There are two classes that should be renamed;
- There are a few generic wildcard types that should be removed;
- There are three methods that should be refactored to reduce the Cognitive Complexity;
- Some method should have nested comments to explain why they are empty;
- *System.out* should be replaced by *System.err*;
- There is a few synchronized class "Vector" that should be replaced by an. Unsynchronized one;
- There are two classes that should be rename or its inheritance should be corrected;
- There are a few fields that should be renamed;
- There are method parameters that are unused and should be removed;
- There are a few loops that should be refactored;
- There is an if statement that should be merged with the enclosing one;
- There are a few public constructors that should be hidden;
- A few try blocks should be extracted to a separate method;
- There are two blocks of code that should be removed or filled;
- There are deprecated codes that should be erased one day;
- There are a few declarations of thrown exception that should be removed;
- There are unnecessary casts that should be removed;
- There are a few methods that should be moved into other classes;
- Some deprecated methods are being override and shouldn't be or should be marked as Deprecated;
- Some variables should be marked as final.

5. CONCLUSION

With this work we achieved our goal of analyzing a system not only in terms of its metrics and organization but also in terms of code smells, disharmonies and errors. By using different techniques and tools to extract metrics of software we were not only able to identify possible sources of problems and bugs, but we were also able to propose different types of refactoring to better the software quality, design and usability.

In terms of finding code smells there were a few discrepancies between the tools but we were able to analyze the information an

JHotDraw is a system well balanced, very polish in terms of code and structure. But it also has a few problems not only in terms of code smells but also in terms of conventions to which a few refactorings should be consider.

On the other hand, JUnit is also a very well-known system used by other Java programs, which explains the good design structure, with a lot of packages, each with a low number of classes. In comparison with JHotDraw in terms of metric it seems as balanced but in terms of code smells and disharmonies it seems to have a few more problems.

In general, both systems have code with a lot of quality in terms of structure, organization and design that can only benefit with from the refactoring proposed.

6. REFERENCES

[1] Fowler M. Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] JUnit - About[online] Retrieved (2 oct. 2018) from: https://junit.org/junit4/

[3] Lanza M.; Marinescu R. Object-Oriented Metrics in Practice. Springer-Verlag 2006. ISBN-10 3-540-24429-8

[4] JSpIRIT - Santiago Vidal [online]. Retrieved (16 oct.2018) from: https://sites.google.com/site/santiagoavidal/projects/jspirit

[5] JHotDraw download page at sourceforge [Online] Retrieved (2 oct. 2018) from: https://sourceforge.net/projects/jhotdraw/

[6] CodePro AnalytiX [online]. Retrieved (19 oct. 2018) from: https://wiki.eclipse.org/images/7/75/CodeProDatasheet.pdf

[7] JDeodorant | Eclipse Plugins, Bundles and Products [online] Retrieved (12 oct. 2018) from: https://marketplace.eclipse.org/content/jdeodorant

[8] JSpIRIT: a flexible tool for the analizys of code smells [online] Retrieved (20 oct. 2018) from: https://www.researchgate.net/publication/300416283_JSpIRIT_a_flexible_tool_for_the_analysis_of_code_smells

[9]

[10] Tools/Integration | OMD Source Code Analyzer [Online] Retrieved (19 oct 2018) from: https://pmd.github.io/latest/pmd_userdocs_tools.html#eclipse