CS 581A3 Software Maintenance & Evolution

Spring 2018

Assignment 3: Software Quality & Code Smells

Contributions

We divided the work in such a way that each person gets to know both the software, and also gets an insight into both the tasks (metrics and smells). Here is a breakup of the taks:

Task	jEdit	PDFSam
Metrics Analysis	Gururaj	Abhimanyu
Smells Analysis	Abhimanyu	Gururaj

1.1 General Metric Changes

In this part of the assignment, we have determined how our code changes (in Assignment 2 for this course) affected the code metrics for jEdit and PDFSam [1,2]. We have used the Metrics plugin in Eclipse to calculate all the metrics [3].

Metrics Changes for jEdit

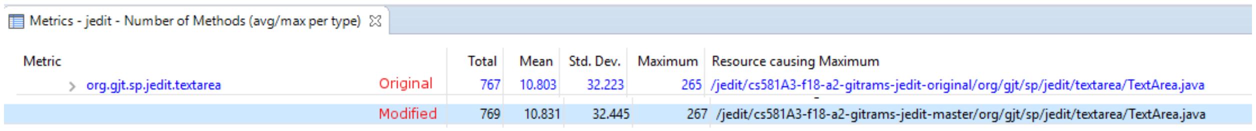
We computed all the metrics for the original version of jEdit and the modified version. We then compared the results and found that the following metrics changed their values:

Metric	jEdit Original	jEdit Modified
Weighted methods per Class	23858	23860
Number of Children	459	465
Number of Methods	5854	5856
Number of Normal Methods	5313	5315
No of inherited methods	67814	67820
Total lines of code	121034	121042
Method lines of code	85940	85944
Class interface size	4875	4877
Number of Polymorphic Methods	257	260

Reasons for Changes:

From the table above, it can be seen that the number of methods has changed. This is because we had added a couple of methods for implementing of the change requests. The two new methods we added were `disableScrollbar()` and `enableScrollbar()`. These were added to the class "`org/gjt/sp/jedit/textarea/TextArea.java`"

Figure 1 confirms that the change in the metrics originate from the addition of these two methods, where we compare the number of methods metric for just the `TextArea.java`



Metrics - jedit - Number of Methods (avg/max per type)

```
Metric                                      Total   Mean    Std. Dev.  Maximum  Resource causing Maximum
  > org.gjt.sp.jedit.textarea    Original   767     10.803  32.223     265      /jedit/cs581A3-f18-a2-gitrams-jedit-original/org/gjt/sp/jedit/textarea/TextArea.java
                                 Modified   769     10.831  32.445     267      /jedit/cs581A3-f18-a2-gitrams-jedit-master/org/gjt/sp/jedit/textarea/TextArea.java
```

Figure 1. Comparison of metrics (Number of Methods) for TextArea.java

Metrics Changes for PDFSam

For PDFSam the Metrics tool computes metrics for the individual modules. We would be focusing on the four modules which we changed in some way or the other while in the process of accomplishing the change requests in A2.

a) PDFSam-fx

Metric	PDFSam-fx Original	PDFSam-fx Modified
Weighted methods per Class	874	876
Number of Children	45	47
Number of Classes	185	187
Measure of Functional Abstraction	6	8
No of inherited methods	18981	19983
Total lines of code	7639	7653
Method lines of code	3300	3306
Reusability	93.846	94.832
Depth of Inheritance Tree	2.659	2.738

b) PDFSam-core

Metric	PDFSam-core Original	PDFSam-core Modified
McCabe Cyclomatic Complexity	1.104	1.116
Weighted methods per Class	597	605
Number of Static methods	30	31
Total lines of code	3719	3772
Method lines of code	1241	1287
Class Interface Size	430	431

c) PDFSam-alternate-mix

Metric	PDFSam-alt-mix Original	PDFSam-alt-mix Modified
McCabe Cyclomatic Complexity	1.231	1.385
Nested Block Depth	1.231	1.269
Weighted methods per Class	32	36
Total lines of code	353	368
Method lines of code	138	153

d) PDFSam-merge

Metric	PDFSam-merge Original	PDFSam-merge Modified
McCabe Cyclomatic Complexity	1.068	1.164
Nested Block Depth	1.068	1.109
Efferent Coupling	2.333	2.667
Instability	0.933	0.944
Normalized Distance	0.067	0.056
Depth of Inheritance Tree	2.778	2.7
Weighted methods per Class	47	64
Number of Overridden Methods	0	9
Number of Attributes	28	30
Number of Methods	41	50
Number of Normal Methods	41	48
Number of Inherited Methods	969	971
Specialization Index	0	0.2

Specialization Index2	0	0.164
Number of Classes	9	10
Total lines of code	634	733
Method lines of code	290	328
Design Size in Classes	9	10
Average Number of Ancestors	2.778	2.7
Class Interface Size	29	38
Data Access Metric	5.571	6.571
Cohesion Among Methods	1.883	1.994
Reusability	5.636	6.475
Flexibility	0.183	0.189
Effectiveness	0.924	0.891
Extendibility	0.389	0.45
Understandability	-5.813	-6.119

Reasons for Changes:

We observe that a lot of metrics have changed in the modified version. Since we added some methods, *lines of code* in the code increased and metrics related to the *number of methods*, *class interface size*, *number of ancestors*, *inheritance* etc., changed.

Apart from these, we also noticed some interesting changes in the metrics. Like the *McCabe Cyclomatic Complexity* increased, though not substantially. This metric counts the number of flows through a piece of code. Each time a branch occurs (owing to conditionals, loops, and logical operators in the methods) this metric is incremented by one. Since we added and changed some methods, that changed the logical flow in some of the classes. And thus, the McCabe Cyclomatic Complexity changed.

Another interesting change was the increase of *Efferent Coupling* in the "merge" module. This is because some of our changes made code in this package access attributes from other packages. The increase in Efferent Coupling led to the increase in *Instability*, since Instability metric is calculated as:

$$I = C_e/(C_a + C_e)$$

Where C_e is the Efferent Coupling and C_a is Afferent Coupling. This is not a good sign, and suggests the edits we made were not very professional. This was confirmed by changes in some other metrics, like *Effectiveness* and *Understandability*, whose values went down in our modified version.

We also noticed some positive changes in the metrics for *Reusability*, *Flexibility* and *Extendibility*. It seems since we added new functionality to the software, hence these metrics improved.

However, there were some metrics, like *Specialization Index* and *Specialization Index2*, for which we could not come up with any reasoning explaining their change.

1.2 Coupling and cohesion

Cohesion

The Metrics plugin calculates the LCOM (Lack of Cohesion of Methods) values to measure the cohesiveness of a class. It uses the ***Henderson-Sellers*** [4] revised method to calculate LCOM values. According to the Henderson-Sellers method, LCOM is calculated as follows.

Let:

M	be the set of methods defined by the class
F	be the set of fields defined by the class

MF number of class methods, which have access to a field
Then:

$$LCOM\ HS = \frac{1}{M-1}\left[M - \frac{1}{F}\sum_{i=0}^{n} MF\right]$$

The LCOM HS metric indicates class associativity. In other words, it indicates whether all the methods of a class of are using all the class fields. In the ideal case, all the class methods are using all its fields, and the class is absolutely associated. In the case of absolute class connectedness, the LCOM HS value is 0. So, we see that lower LCOM values are good, since a low "lack of cohesion" score implies a lot of cohesion. And higher values are considered bad. The values of LCOM as per Henderson-Sellers method lie in the range [0-2]. For Henderson-Sellers LCOM, values higher than 1 are generally considered to be alarming.

a) jEdit

Classes with Lowest Cohesion

In jEdit, we identified the following two classes with the lowest cohesion (i.e., highest values of LCOM HS):

1) org.jedit.options.CombinedOptions

LCOM value: 1.333

Reason: This class has 4 methods and only one of them uses the class attributes. Thus, it demonstrates a very low cohesion.

2) org.gjt.sp.jedit.gui.DockableWindowManagerImpl

LCOM value: 1

Reason: This class has 28 methods and 11 attributes, and not all methods use these attributes. Most of its methods access attributes from other classes, 15 attributes from 5 external classes. Thus, this class also shows a low cohesion.

Classes with Highest Cohesion

In jEdit, we identified the following two classes with the highest cohesion (i.e., lowest values of LCOM HS):

1) org.gjt.sp.jedit.gui.KeyEventWorkaround

LCOM value: 0

Reason: This class has 3 methods and 3 attributes. All the methods use all these attributes, and do not access attributes from external classes.

2) org.gjt.sp.jedit.textarea.ElasticTabstopsTabExpander

LCOM value: 0

Reason: This class has 3 methods and just one attribute. All the methods use this attribute. Thus, the class demonstrated a high degree of cohesion.

b) PDFSam

Classes with Lowest Cohesion

In PDFSam, we identified the following two classes with the lowest cohesion (i.e., highest values of LCOM HS):

1) org.pdfsam.ui.selection.single.SingleSelectionPane

LCOM value = 0.906

Reason: This class has 18 methods and 14 attributes, but only 8 attributes are being used by the 3 of the class methods. This is the reason for the low cohesion. This can be seen in Figure 2, which is the cohesion perspective from the software ***inCode.***

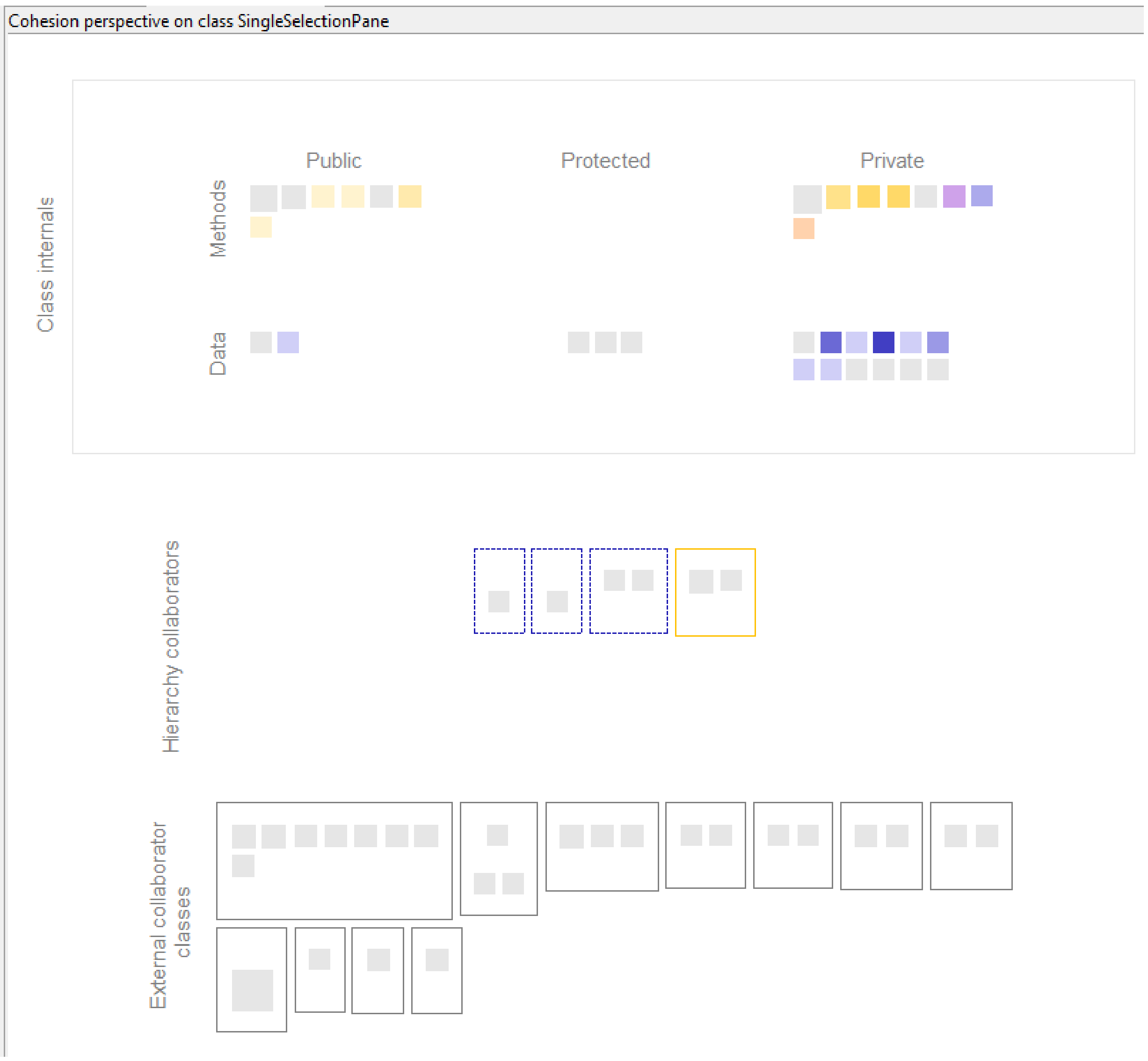

Cohesion perspective on class SingleSelectionPane
Class internals
Public
Protected
Private
Methods
Data
Hierarchy collaborators
External collaborator classes

Figure 2. Cohesion analysis of the class from inCode

2) org.pdfsam.merge.MergeParametersBuilder

LCOM value = 0.875

Reason: This class has 10 methods and 8 attributes, but only 3 attributes are being used by the class methods. This shows a low cohesion.

Classes with Highest Cohesion

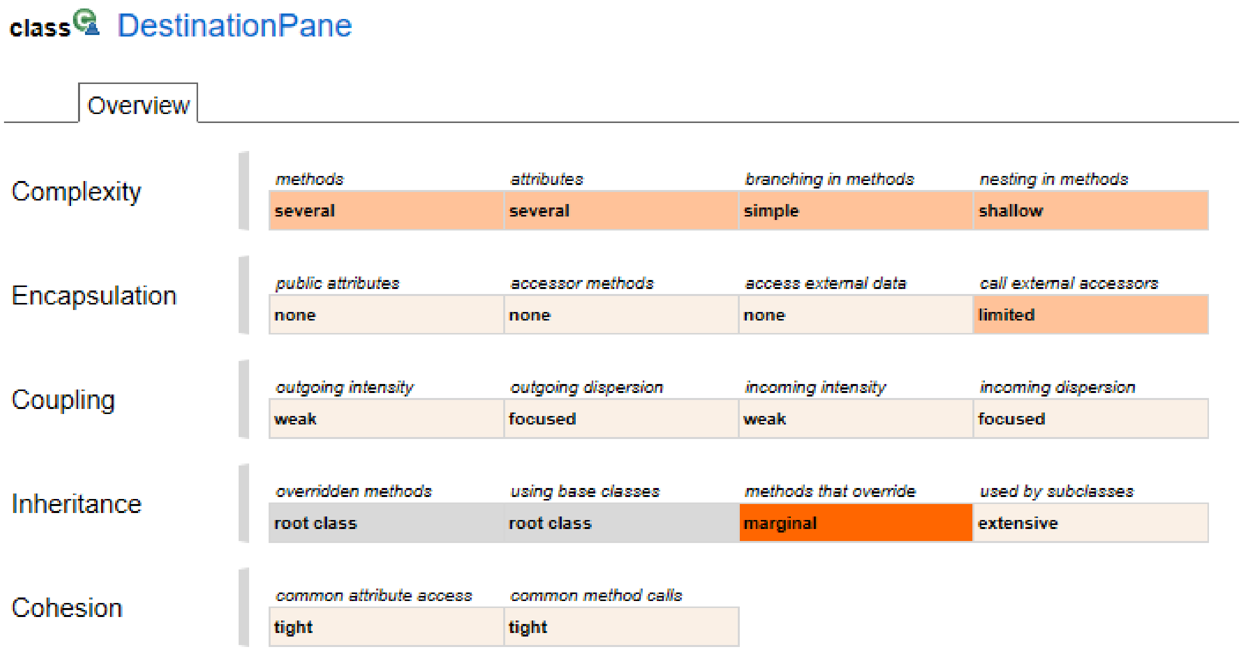
In PSDSam, we identified the following two classes with the highest cohesion (i.e., lowest values of LCOM HS):

1) org.pdfsam.task.PdfRotationInput

LCOM value = 0.167

Reason: 4 of the 5 class methods access all the 3 attributes in this class. This shows a high degree of cohesion.

2) org.pdfsam.ui.io.DestinationPane
LCOM value = 0.33
Reason: All the 4 methods of this class access the 2 attributes present in this class. This class thus has a tight cohesion. inCode software also confirmed this, Figure 3.



class DestinationPane

Overview

Complexity	methods	attributes	branching in methods	nesting in methods
	several	several	simple	shallow
Encapsulation	public attributes	accessor methods	access external data	call external accessors
	none	none	none	limited
Coupling	outgoing intensity	outgoing dispersion	incoming intensity	incoming dispersion
	weak	focused	weak	focused
Inheritance	overridden methods	using base classes	methods that override	used by subclasses
	root class	root class	marginal	extensive
Cohesion	common attribute access	common method calls		
	tight	tight		

Figure 3. inCode overview for the class DestinationPane, shoing a tight cohesion

Difference between Classes with Highest Cohesion and Classes with Lowest Cohesion

We observed that the classes with highest cohesion are small in size, have a few methods and a few attributes, and most of the method pairs (almost all pairs) access a common attribute. But classes with lowest cohesion are large, have many methods and many attributes. And only a few of these method pairs access a common metric.

Coupling

The metrics plugin provides four metrics related to coupling. These are:

1) Afferent Coupling (avg/max per packageFragment)
The number of classes outside of the package, that depend on the classes of the current package. High values of afferent coupling indicate that a given package is of high importance.
2) Efferent Coupling (avg/max per packageFragment)
The number of classes within the package that depend on classes outside the package. High values of efferent coupling show how dependent the given package is on external packages.
3) Direct Class Coupling (avg/max per packageFragment)
The number of methods of a class that depend on methods outside of the class, grouped by package.
4) Direct Class Coupling (avg/max per type)
The number of methods of a class that depend on methods outside of the class.

For an idea of coupling, Efferent Coupling would have been a good choice, but the metrics plugin calculates this value for a package as a whole, rather than for individual classes. So, we cannot identify individual classes with highest and lowest coupling using this metric. We thus use the last metric in the above list, i.e., Direct Class Coupling to identify classes with extreme coupling values.

a) jEdit

Classes with Highest Coupling

In jEdit, we identified the following two classes with the highest coupling

1) org.gjt.sp.jedit.textarea.TextArea

   Direct Class Coupling value = 20

   Reason: Many of its methods access directly (or via getter/setters) 24 attributes from 12 external classes.

2) org.gjt.sp.jedit.View

   Direct Class Coupling value = 19

   Reason: Many of its methods access directly (or via getter/setters) 108 attributes from 43 external classes

Classes with Lowest Coupling

In jEdit, we identified the following two classes with the lowest coupling:

1) org.gjt.sp.jedit.textarea.TextAreaMouseHandler

   Direct Class Coupling value = 1

   Reason: This class has 14 methods and only one of them accesses attributes from an external class.

2) org.gjt.sp.jedit.JEditRegisterSaver

   Direct Class Coupling value = 1

   Reason: This class has 3 methods and just one of them accesses attributes from an external class.

b) PDFSam

Classes with Highest Coupling

In PDFSam, we identified the following two classes with the highest coupling:

1) org.pdfsam.ui.selection.multiple.SelectionTable

   Direct Class Coupling value = 9

   Reason: 8 of the 22 methods of this class access attributes from external classes. This shows a high degree of coupling, for PDFSam.

2) org.pdfsam.ui.notification.NotificationsController

   Direct Class Coupling value = 9

   Reason: 5 of the 10 class methods access attributes from external classes. We can see this from Figure 4, which shows how the class is accessing elements from external classes.

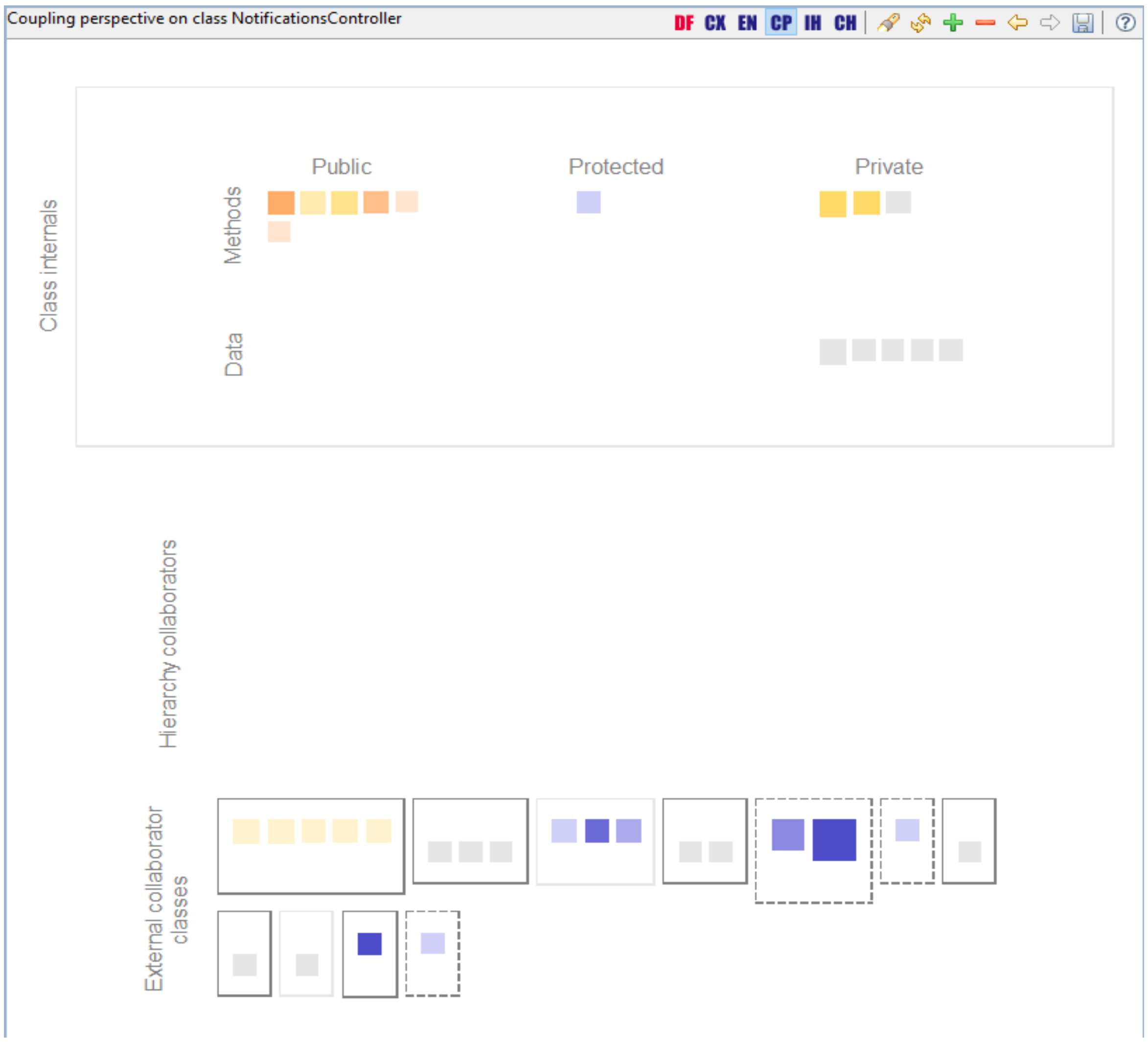


Figure 4. inCode analysis of the class NotificationsController

Classes with Lowest Coupling

In PSDSam, we identified the following two classes with the lowest coupling:

1) org.pdfsam.ui.selection.multiple.ReverseColumn
   Direct Class Coupling value = 0
   Reason: None of the class's methods access external classes.
2) org.pdfsam.alternatemix.AlternateMixParametersBuilder
   Direct Class Coupling value = 0
   Reason: None of the class's methods access external classes.

Difference between Classes with Highest Coupling and Classes with Lowest Coupling

We observed that classes with the highest coupling are generally large, have many methods and many attributes, and implement a complex or diverse functionality. Thus, they need to access elements from other classes or packages. Whereas classes with the lowest coupling are small in size, have a few methods and a few attributes, and implement a very simple or "one-task" functionality. They, thus, do not need to access other classes.

2.1 Detecting and analyzing code smells

In this part we used the modified versions of the projects jEdit and PDFsam from assignment 2. We completed the detection and analysis of code smell using a couple of tool called **JDeodorant** and **iPlasma** [5] [6]. JDeodorant is an Eclipse plugin available for free that detects five types of bad smells. The iPlasma tool is an integrated platform/tool for quality assessment of object-oriented systems that works with multiple languages [7]. As compared to JDeodorant, iPlasma is able to detect several code smells called disharmonies, which are classified in identity disharmonies, collaboration disharmonies, and classification disharmonies. We also tried some other tools such as inCode. InCode works fine with modified PDFSam because the total number of lines are less than 100000 which is the set limit on the free version of this software, but this is not the case with jEdit. Unfortunately, the licensed version is not available anywhere over the web so that we can uses it for jEdit.

Analysis of JEdit

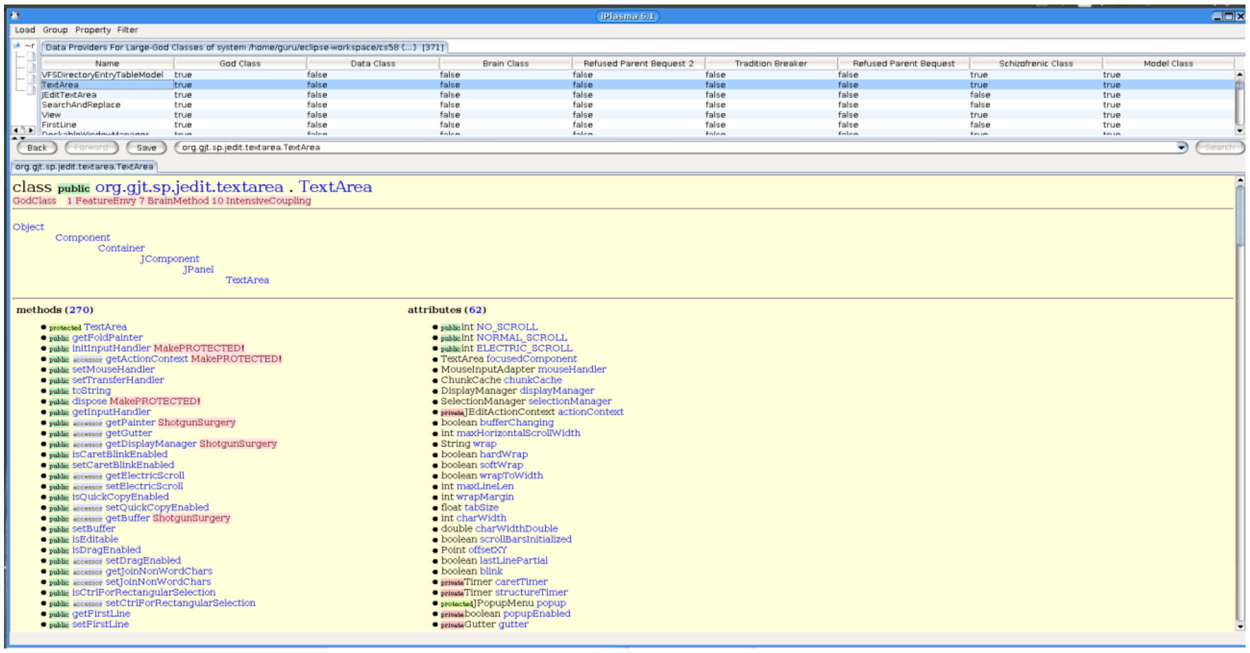
Smelly Class: `class public org.gjt.sp.jedit.textarea.TextArea`

Which Bad Smell? -> God Class

In JEdit, this class represents an Abstract TextArea component. JEdit uses a concrete instance of this class called the JEditTextArea. This class has a very large number of methods and attributes (Figure 5), and only second to another class called `Parser.java` which has the highest number of methods and attributes when the JEdit codebase is considered as a whole.

1. *Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.*

   It is a good object-oriented design practice to distribute the intelligence of a system uniformly among the top-level classes. Conversely, the God Class bad smell refers to classes that tend to centralize code base and therefore the intelligence of the system into a few large classes. From the class-interaction diagram (Figure 6), we observe that this class performs significantly large amount of work and delegates small work to other classes. It also uses data from other classes as represented by red lines. This aligns with the definition of a God class.



Data Providers For Large-God Classes of system /home/guru/eclipse-workspace/cs58 (...) [371]

Name	God Class	Data Class	Brain Class	Refused Parent Bequest 2	Tradition Breaker	Refused Parent Bequest	Schizofrenic Class	Model Class
VFSDirectoryEntryTableModel	true	false	false	false	false	false	true	true
TextArea	true	false	false	false	false	false	true	true
JEditTextArea	true	false	false	false	false	false	false	true
SearchAndReplace	true	false	false	false	false	false	false	true
View	true	false	false	false	false	false	true	true
FirstLine	true	false	false	false	false	false	false	true

org.gjt.sp.jedit.textarea.TextArea

class public org.gjt.sp.jedit.textarea . TextArea
GodClass 1 FeatureEnvy 7 BrainMethod 10 IntensiveCoupling

```
Object
    Component
        Container
            JComponent
                JPanel
                    TextArea
```

methods (270)

- protected TextArea
- public getFoldPainter
- public initInputHandler MakePROTECTED!
- public accessor getActionContext MakePROTECTED!
- public setMouseHandler
- public setTransferHandler
- public toString
- public dispose MakePROTECTED!
- public getInputHandler
- public accessor getPainter ShotgunSurgery
- public accessor getGutter
- public accessor getDisplayManager ShotgunSurgery
- public isCaretBlinkEnabled
- public setCaretBlinkEnabled
- public accessor getElectricScroll
- public accessor setElectricScroll
- public isQuickCopyEnabled
- public accessor setQuickCopyEnabled
- public accessor getBuffer ShotgunSurgery
- public setBuffer
- public isEditable
- public isDragEnabled
- public accessor setDragEnabled
- public accessor getJoinNonWordChars
- public accessor setJoinNonWordChars
- public isCtrlForRectangularSelection
- public accessor setCtrlForRectangularSelection
- public getFirstLine
- public setFirstLine

attributes (62)

- public int NO_SCROLL
- public int NORMAL_SCROLL
- public int ELECTRIC_SCROLL
- TextArea focusedComponent
- MouseInputAdapter mouseHandler
- ChunkCache chunkCache
- DisplayManager displayManager
- SelectionManager selectionManager
- private JEditActionContext actionContext
- boolean bufferChanging
- int maxHorizontalScrollWidth
- String wrap
- boolean hardWrap
- boolean softWrap
- boolean wrapToWidth
- int maxLineLen
- int wrapMargin
- float tabSize
- int charWidth
- double charWidthDouble
- boolean scrollBarsInitialized
- Point offsetXY
- boolean lastLinePartial
- boolean blink
- private Timer caretTimer
- private Timer structureTimer
- protected JPopupMenu popup
- private boolean popupEnabled
- private Gutter gutter

Figure 5. Methods and Attributes of the class TextArea in jEdit

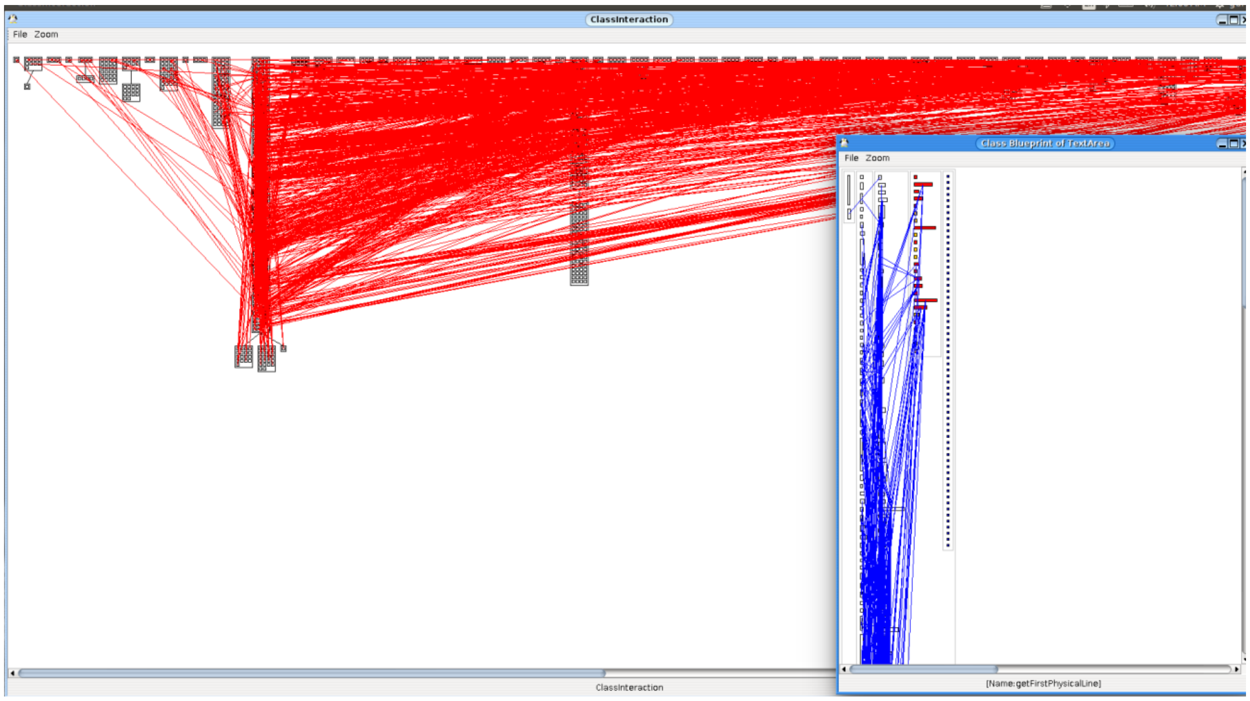
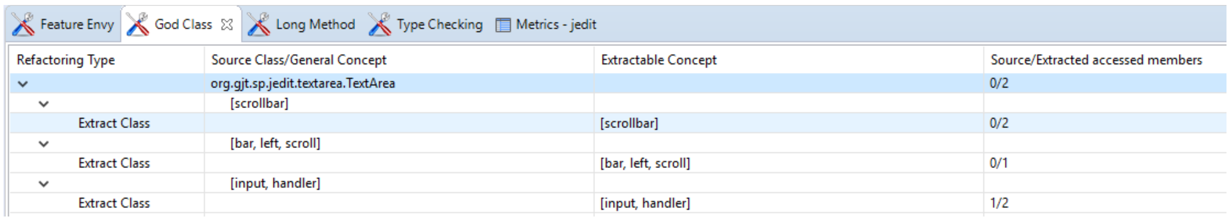


Figure 6. Class interaction and blue print for the TextArea class in jEdit



Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted accessed members
	org.gjt.sp.jedit.textarea.TextArea		0/2
	[scrollbar]		
Extract Class		[scrollbar]	0/2
	[bar, left, scroll]		
Extract Class		[bar, left, scroll]	0/1
	[input, handler]		
Extract Class		[input, handler]	1/2

Figure 7. JDeodorant's analysis of the TextArea class

2. *Explain why the class/method is flagged as smelly (be specific).*

   This class is a huge class in terms of number of methods, attributes, and lines of code. This class has 270 methods and 62 attributes. The total number of lines in this class are 6762. According to the authors, "This class uses a minimal set of jEdit APIs because it is the base class of the `JEditEmbeddedTextArea` and `StandaloneTextArea`, so it needs to be embeddable and separable," however, this class still has a large number of methods and attributes (explained in brief in answer 3). Therefore, this class is flagged as one of the God classes by both iPlasma and JDeodorant while analyzing the JEdit source code. JDeodorant's analysis for this class is shown in Figure 7.

3. *Do you agree that the detected smell is an actual smell? Justify your answer.*

   Yes, we agree with the results from the tools that this is a God Class. As per the definition of the God Class, it performs too much work on its own and delegates only minor work to other classes. It also uses the data from other classes, for example, attributes of class instance `JEditBuffer`

`buffer`. As we went through this class using Eclispe, we found that TextArea functionalities such as `goToPrevLine`, `scrollUpPage`, `deleteParagraph`, `showPopupMenu`, and many more, are implemented in this class. All these methods have very little in common to be placed in one class because they represent different parts of the TextArea of JEdit. This class has 270 methods and 62 attributes which is beyond the threshold for being a God Class. Therefore, we can confirm that this is actually a God Class. Jdeodorant suggests some refactorings to get rid of this smell. These are shown in Figure 8.

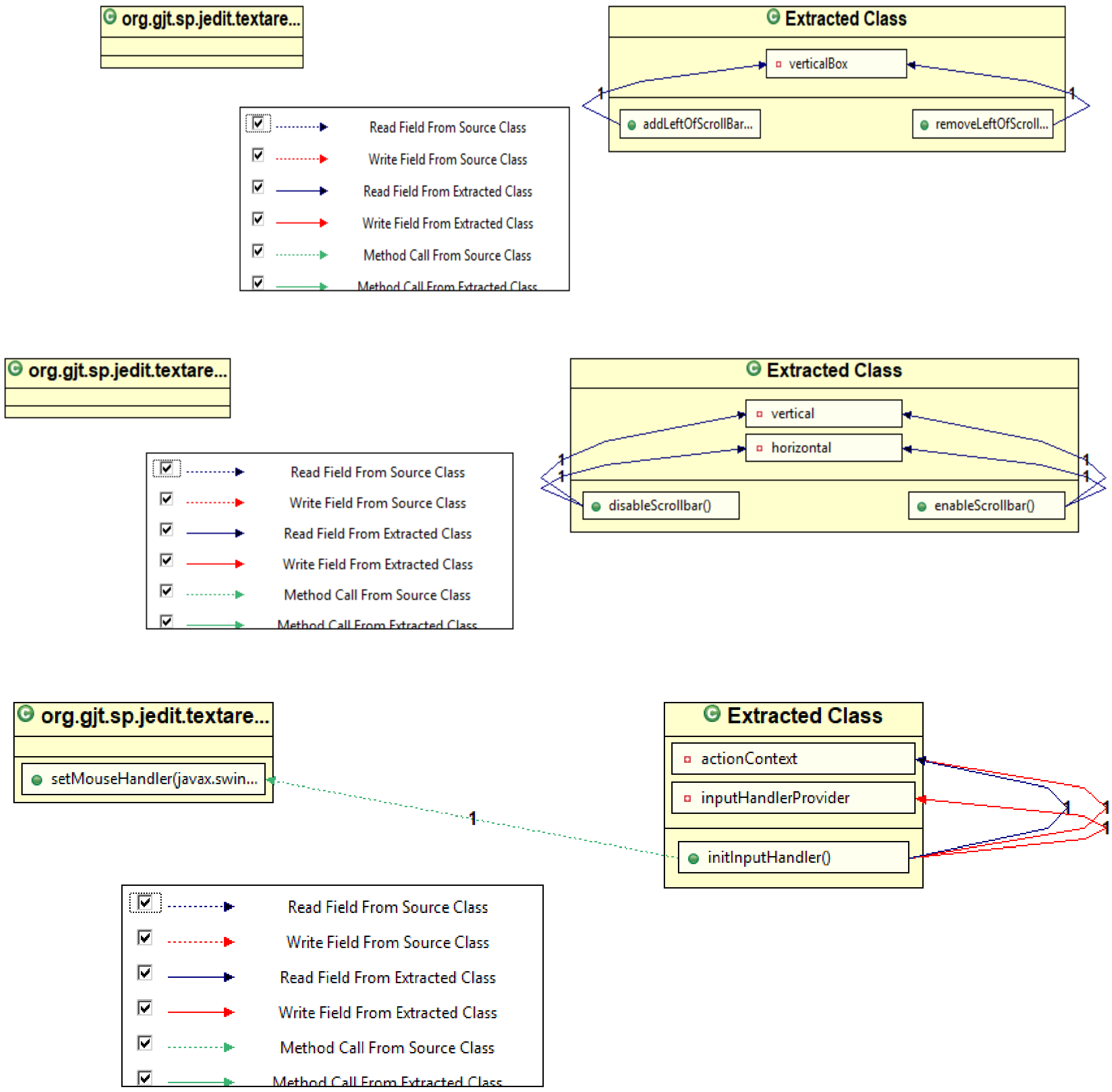

org.gjt.sp.jedit.textare...
Extracted Class
verticalBox
addLeftOfScrollBar...
removeLeftOfScroll...
Read Field From Source Class
Write Field From Source Class
Read Field From Extracted Class
Write Field From Extracted Class
Method Call From Source Class
Method Call From Extracted Class
org.gjt.sp.jedit.textare...
Extracted Class
vertical
horizontal
disableScrollbar()
enableScrollbar()
org.gjt.sp.jedit.textare...
setMouseHandler(javax.swin...
Extracted Class
actionContext
inputHandlerProvider
initInputHandler()
1

Figure 8. Suggested Refactoring Operations from JDeodorant for the TextArea Class

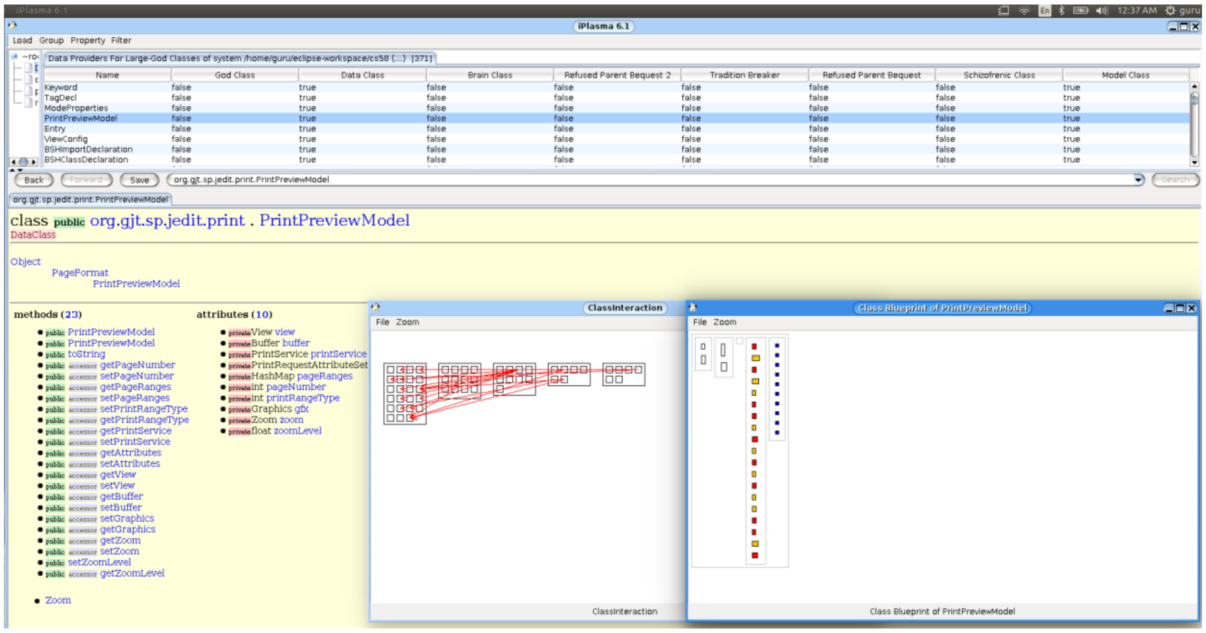
Smelly Class: `class public org.gjt.sp.jedit.print.PrintPreviewModel`

Which Bad Smell? -> Data Class

In JEdit, this class is a data model for the print preview pane and it contains only setter and getter methods for the print preview display of JEdit.

1. *Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.*

   The bad smell - Data class - refers to a “dumb” data holder class that does not have any complex functionality, however, other classes strongly rely on it. These classes exhibit a poor object-oriented design because they lack the principle of encapsulation of data and lack the data-functionality proximity. The `PrintPreviewModel` class mentioned above has the symptoms of a Data Class. In Figure 9, we see that the methods in this class are mostly getter and setter type of methods that retrieve or change the values of attributes listed on the right side of the figure. These methods do not have significant functionalities, which aligns with the definition of a Data Class.


iPlasma 6.1
12:37 AM
guru
iPlasma 6.1
Load
Group
Property
Filter
Data Providers For Large-God Classes of system /home/guru/eclipse-workspace/cs58 (...) [371]
Back
Forward
Save
org.gjt.sp.jedit.print.PrintPreviewModel
Search
org.gjt.sp.jedit.print.PrintPreviewModel
ClassInteraction
File
Zoom
ClassInteraction
Class Blueprint of PrintPreviewModel
File
Zoom
Class Blueprint of PrintPreviewModel

Name	God Class	Data Class	Brain Class	Refused Parent Bequest 2	Tradition Breaker	Refused Parent Bequest	Schizofrenic Class	Model Class
Keyword	false	true	false	false	false	false	false	true
TagDecl	false	true	false	false	false	false	false	true
ModeProperties	false	true	false	false	false	false	false	true
PrintPreviewModel	false	true	false	false	false	false	false	true
Entry	false	true	false	false	false	false	false	true
ViewConfig	false	true	false	false	false	false	false	true
BSHImportDeclaration	false	true	false	false	false	false	false	true
BSHClassDeclaration	false	true	false	false	false	false	false	true

class public org.gjt.sp.jedit.print . PrintPreviewModel
DataClass

Object
PageFormat
PrintPreviewModel

methods (23)

- public PrintPreviewModel
- public PrintPreviewModel
- public toString
- public accessor getPageNumber
- public accessor setPageNumber
- public accessor getPageRanges
- public accessor setPageRanges
- public accessor setPrintRangeType
- public accessor getPrintRangeType
- public accessor getPrintService
- public accessor setPrintService
- public accessor getAttributes
- public accessor setAttributes
- public accessor getView
- public accessor setView
- public accessor getBuffer
- public accessor setBuffer
- public accessor setGraphics
- public accessor getGraphics
- public accessor getZoom
- public accessor setZoom
- public setZoomLevel
- public accessor getZoomLevel

- Zoom

attributes (10)

- private View view
- private Buffer buffer
- private PrintService printService
- private PrintRequestAttributeSet
- private HashMap pageRanges
- private int pageNumber
- private int printRangeType
- private Graphics gfx
- private Zoom zoom
- private float zoomLevel

Figure 9. Details for the class PrintPreviewModel in jEdit

2. *Explain why the class/method is flagged as smelly (be specific).*

   This class is a lightweight class that provides almost no functionality through its methods. The methods in this class are either getters or setters for the attributes of the class, for example, `getPageNumber(), setPageNumber(), getPageRanges(), setPageRanges()`, etc. that modify or access attributes like `pageNumber` or `pageRange`. Therefore, according to the definition mentioned above, this class was flagged as a Data Class by both iPlasma and JDeodorant tool.

3. *Do you agree that the detected smell is an actual smell? Justify your answer.*

   Yes. As we analyzed this class in Eclipse IDE, we observed that the methods in this class had no real functionality because most of them were either getX() or setX() type methods that operated on the attributes such as `View`, `Buffer`, `pageNumber`, `zoomLevel`, etc. This class violates the object-oriented design principle of encapsulation and therefore it is a Data Class.

Smelly Method:

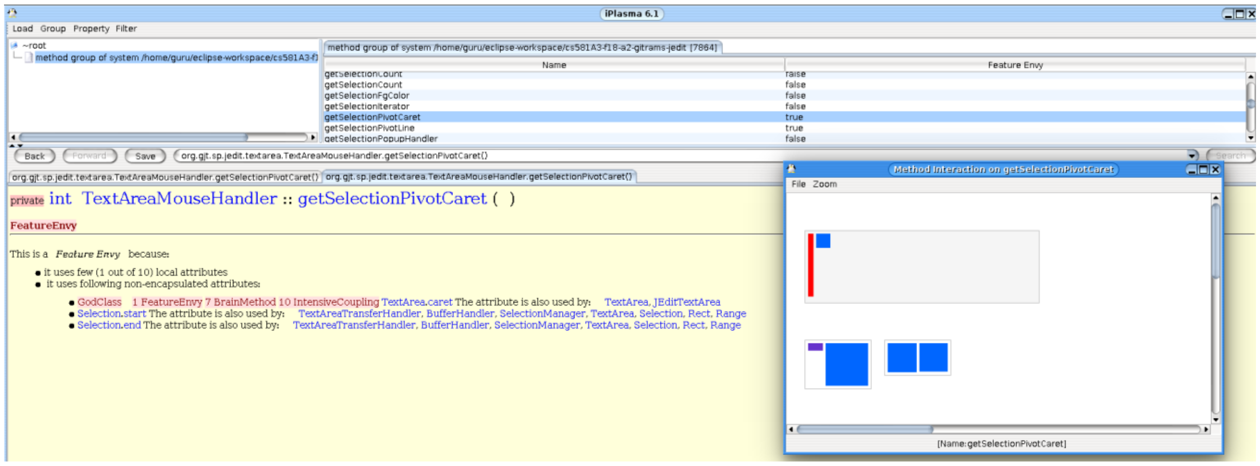
`private int TextAreaMouseHandler::getSelectionPivotCaret()`

Which Bad Smell? -> Feature Envy

We detected this method, which is an example of Feature Envy bad smell, in both the tools - JDeodorant and iPlasma.

1. *Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.*

   The Feature Envy bad smell refers to methods that are more interested in the data and attributes of other classes than that of their own class. These methods are symptoms of a bad design because they access directly or indirectly (via accessors) the data of other classes. For a good object-oriented design, the data and the methods modifying that data should stay as close together as possible. The `getSelectionPivotCaret`() method accesses data from classes other than its own class. This might be a sign that this method might have be misplaced and it should be moved to the class, attributes of which this method accesses.


iPlasma 6.1
Load Group Property Filter
~root
method group of system /home/guru/eclipse-workspace/cs581A3-f1
Back
Forward
Save
Method Interaction on getSelectionPivotCaret
File Zoom
[Name:getSelectionPivotCaret]

method group of system /home/guru/eclipse-workspace/cs581A3-f18-a2-gitrams-jedit [7864]

Name	Feature Envy
getSelectionCount	false
getSelectionCount	false
getSelectionFgColor	false
getSelectionIterator	false
getSelectionPivotCaret	true
getSelectionPivotLine	true
getSelectionPopupHandler	false

org.gjt.sp.jedit.textarea.TextAreaMouseHandler.getSelectionPivotCaret()

private int TextAreaMouseHandler :: getSelectionPivotCaret ( )

FeatureEnvy

This is a *Feature Envy* because:

- it uses few (1 out of 10) local attributes
- it uses following non-encapsulated attributes:
  - GodClass 1 FeatureEnvy 7 BrainMethod 10 IntensiveCoupling TextArea.caret The attribute is also used by: TextArea, JEditTextArea
  - Selection.start The attribute is also used by: TextAreaTransferHandler, BufferHandler, SelectionManager, TextArea, Selection, Rect, Range
  - Selection.end The attribute is also used by: TextAreaTransferHandler, BufferHandler, SelectionManager, TextArea, Selection, Rect, Range

Figure 10. Details of the method exhibiting the FeatureEnvy smell

2. *Explain why the class/method is flagged as smelly (be specific).*

   The `getSelectionPivotCaret()` method accesses data from an instance (`textArea`) of another class called `TextArea` in the line "`int caret = textArea.caret;`". Here, the caret attribute of the class instance `TextArea textArea` is accessed by this method. Similarly, the other variable "`Selection s`" defined in this method is also an attribute of the class instance `TextArea textArea`. Instead of using the attributes and data from the own class

(`TextAreaMouseHandler`), the method is accessing data from other classes. Therefore, by definition this is an example of Feature Envy bad smell.

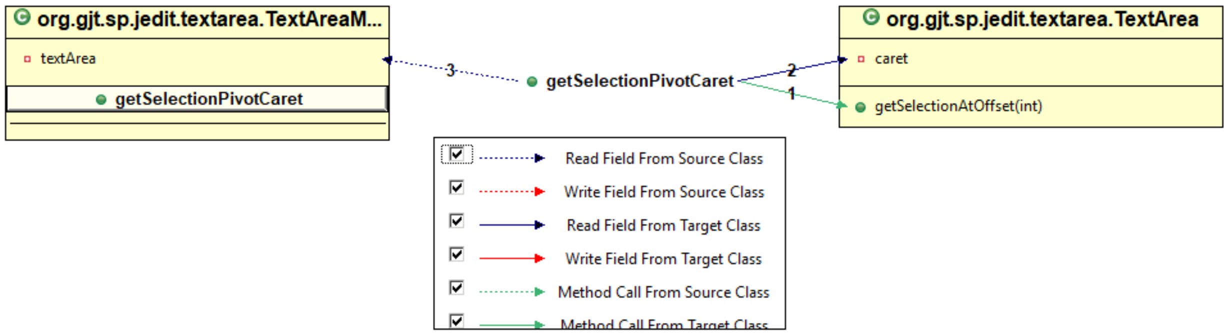

org.gjt.sp.jedit.textarea.TextAreaM...
textArea
getSelectionPivotCaret
3
getSelectionPivotCaret
2
1
org.gjt.sp.jedit.textarea.TextArea
caret
getSelectionAtOffset(int)
Read Field From Source Class
Write Field From Source Class
Read Field From Target Class
Write Field From Target Class
Method Call From Source Class
Method Call From Target Class

Figure 11. JDeodorant's analysis for the TextAreaMouse : :`getSelectionPivotCaret method`

3. *Do you agree that the detected smell is an actual smell? Justify your answer.*

   Yes, we agree with the detected Feature Envy smell for this method. Reason being, this method has only two defined variables (`int caret and Selection s`) that were initialized with the data from classes other than the own class of this method. This method (a really short method with only 6 lines of code) does not have any operations that directly work on the attributes of the own class (`TextAreaMouseHandle`). Therefore, this method is evidently an example of Feature Envy bad smell.

Analysis of PDFSam

Analysis of PDFSam using both the smell detecting tools (JDeodorant and iPlasma) revealed that the master branch (v.3.3.5) was written really well in terms of good object-oriented design. PDFSam code base has very few structural design issues that were uncovered by the smells described below. JDeodorant was only able to detect the classes exhibiting Feature Envy bad smell out of the five smells it offered to detect. Therefore, we used iPlasma to detect some more smells. The iPlasma software detected a few Data Class bad smells and just one example of Shotgun Surgery bad smell. Again, because of modular and object-oriented design of the original master branch and the modified branch (from assignment 2), there were only three types of bad smells detected in the entire code base using the two tools mentioned above.

Smelly Class:

`class org.pdfsam.ui.selection.multiple.move.SingleSelectionAndFocus`

Which Bad Smell? -> Data Class

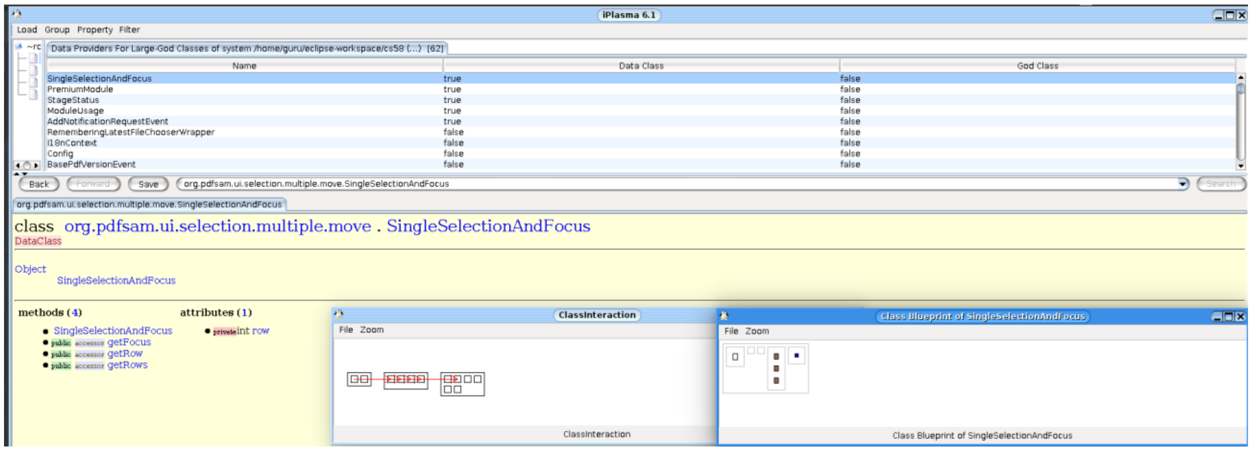
This class is a part of the PDF Split And Merge source code of PDFSam and provides a data model for the Split And Merge module.

1. *Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.*

   Data class is a bad smell that refers to a "dumb" data holder class that does not have any complex functionality, however, other classes strongly rely on them. They exhibit a poor object-oriented design because they lack the principle of encapsulation of data and lack the data-functionality proximity. The `SingleSelectionAndFocus` class has the symptoms of a Data Class. From Figure 12, we see that the methods in this class are all getter type of methods that retrieve the data from the attribute (only one: `int row`) listed on the right side of the figure. These getter methods do not have significant functionalities, which aligns with the definition of a Data Class.

2. *Explain why the class/method is flagged as smelly (be specific).*

   This class is very small class containing getter methods that provide almost no functionality. Three getter methods in this class are public accessor `getFocus(), public accessor getRow(),` and `public accessor getRows()`. They modify or access the only attribute in the class which is 'int row'. Therefore, by the definition of a Data Class, this class was flagged as a Data Class by iPlasma and JDeodorant tools.


iPlasma 6.1

Load Group Property Filter

Data Providers For Large-God Classes of system /home/guru/eclipse-workspace/cs58 (...) [62]

Name	Data Class	God Class
SingleSelectionAndFocus	true	false
PremiumModule	true	false
StageStatus	true	false
ModuleUsage	true	false
AddNotificationRequestEvent	true	false
RememberingLatestFileChooserWrapper	false	false
I18nContext	false	false
Config	false	false
BasePdfVersionEvent	false	false

Back Forward Save org.pdfsam.ui.selection.multiple.move.SingleSelectionAndFocus Search

org.pdfsam.ui.selection.multiple.move.SingleSelectionAndFocus

class org.pdfsam.ui.selection.multiple.move . SingleSelectionAndFocus
DataClass

Object
SingleSelectionAndFocus

methods (4)
- SingleSelectionAndFocus
- public accessor getFocus
- public accessor getRow
- public accessor getRows

attributes (1)
- private int row

ClassInteraction
File Zoom
ClassInteraction
Class Blueprint of SingleSelectionAndFocus
File Zoom
Class Blueprint of SingleSelectionAndFocus

Figure 12. Details for the class SingleSelectionAndFocus in PDFSam

3. *Do you agree that the detected smell is an actual smell? Justify your answer.*

   Yes, we agree that this is a Data Class. Our analysis of this class in Eclipse IDE confirmed that these getter methods in have no real functionality because they operated on the attribute (`int row`) and returned the row immediately. This class violates the object-oriented principle of encapsulation and lacks the data-functionality proximity; therefore, it is a Data Class.

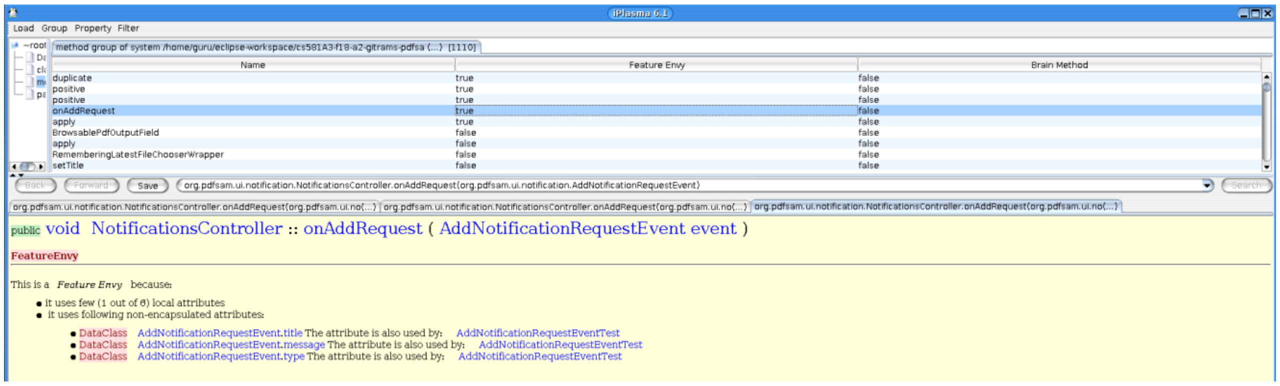
Smelly Method: `public void NotificationsController::`
`onAddRequest(AddNotificationRequestEvent event)`

Which Bad Smell? -> Feature Envy

This method which is an example of Feature Envy bad smell was detected in both the tools, JDeodorant and iPlasma.

1. *Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.*

   Methods that are more interested in the data and attributes of other classes than that of their own class exhibit the Feature Envy bad smell. These methods are symptoms of a bad design because they access directly or indirectly (via accessors) the data of other classes. The `onAddRequest(AddNotificationRequestEvent event)` method accesses more data from classes other than its own class (`public class NotificationsController{}`). This might be a sign that this method might have been misplaced and it should be moved to the class, attributes of which this method accesses. For a good object-oriented design, the data and the methods modifying that data should stay as close together as possible.



method group of system /home/guru/eclipse-workspace/cs581A3-f18-a2-gitrams-pdfsa (...) [1110]

Name	Feature Envy	Brain Method
duplicate	true	false
positive	true	false
positive	true	false
onAddRequest	true	false
apply	true	false
BrowsablePdfOutputField	false	false
apply	false	false
RememberingLatestFileChooserWrapper	false	false
setTitle	false	false

org.pdfsam.ui.notification.NotificationsController.onAddRequest(org.pdfsam.ui.notification.AddNotificationRequestEvent)

org.pdfsam.ui.notification.NotificationsController.onAddRequest(org.pdfsam.ui.no(...) | org.pdfsam.ui.notification.NotificationsController.onAddRequest(org.pdfsam.ui.no(...) | org.pdfsam.ui.notification.NotificationsController.onAddRequest(org.pdfsam.ui.no(...)

public void NotificationsController :: onAddRequest (AddNotificationRequestEvent event)

FeatureEnvy

This is a *Feature Envy* because:

- it uses few (1 out of 6) local attributes
- it uses following non-encapsulated attributes:
    - DataClass AddNotificationRequestEvent.title The attribute is also used by: AddNotificationRequestEventTest
    - DataClass AddNotificationRequestEvent.message The attribute is also used by: AddNotificationRequestEventTest
    - DataClass AddNotificationRequestEvent.type The attribute is also used by: AddNotificationRequestEventTest

Figure 13. Analysis from iPlasma for the NotificationsController::onAddRequest method

2. *Explain why the class/method is flagged as smelly (be specific).*

   The `onAddRequest(AddNotificationRequestEvent event)` method accesses only 1 out of 6 local attributes as we see in Figure 13. The only local attribute accessed by this method is "`private NotificationsContainer container`". This method accesses data from an instance (`event`) of another class called `AddNotificationRequestEvent` (incidentally it's a Data Class). The attributes such as `event.title`, `event.message`, and `event.type` of the class instance `AddNotificationRequestEvent event` are accessed by this method. Instead of using the attributes and data from the own class (`NotificationsController`), this method is accessing data from other classes. Therefore, by definition this is an example of Feature Envy bad smell.

3. *Do you agree that the detected smell is an actual smell? Justify your answer.*

   Yes, we agree with the detected Feature Envy smell for this method, because this method uses only one local attribute (`private NotificationsContainer container`) out of 6 local attributes. The other attributes on which this short method (of only one line) operates are event.title, event.message, and event.type. These attributes represent the data from other class than the own-class of this method. Therefore, this method is evidently an example of Feature Envy bad smell.

Smelly Method:

`public String Pdfsam::property(ConfigurableProperty prop)`

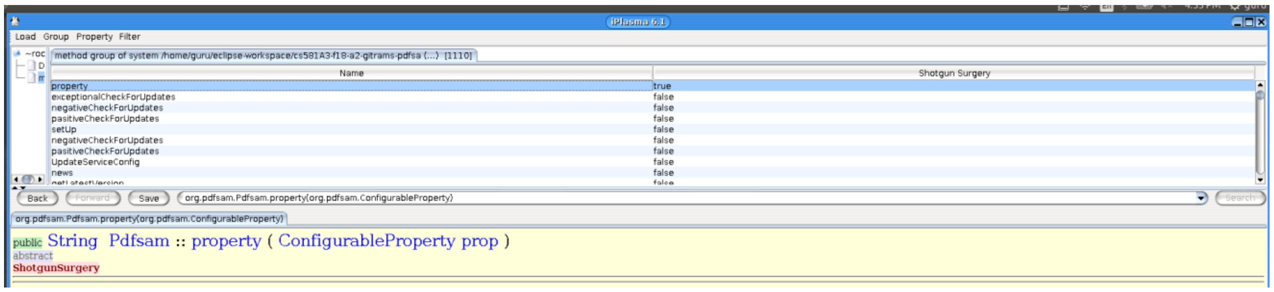
Which Bad Smell? -> Shotgun Surgery

This method was detected using iPlasma software and it is the only example of Shotgun surgery bad smell from the entire code base.

1. *Briefly describe the smell by considering the class, methods, attributes, etc. involved in the smell.*

A method demonstrates Shotgun Surgery bad smell if a change in that method requires cascading changes (small but many) in several related methods or classes. The changes are required to be made all over the place in the codebase, they are hard to find and maintain, therefore the method should be refactored so that the changes are limited to a single class.

The method `property(ConfigurableProperty prop)` from the interface `public interface Pdfsam {}` is detected as a Shotgun Surgery bad smell. This class interface keeps all the information about the current running version of PDFsam. The `ConfigurableProperty prop` could be any property such as project version, build date, license URL, homepage URL, download URL, etc.



method group of system /home/guru/eclipse-workspace/cs581A3-f18-a2-gitrams-pdfsa (...) [1110]

Name	Shotgun Surgery
property	true
exceptionalCheckForUpdates	false
negativeCheckForUpdates	false
pasitiveCheckForUpdates	false
setUp	false
negativeCheckForUpdates	false
pasitiveCheckForUpdates	false
UpdateServiceConfig	false
news	false
getLatestVersion	false

org.pdfsam.Pdfsam.property(org.pdfsam.ConfigurableProperty)

org.pdfsam.Pdfsam.property(org.pdfsam.ConfigurableProperty)

public String Pdfsam :: property (ConfigurableProperty prop)
abstract
ShotgunSurgery

Figure 14. iPlasma tool showing the method marked as having Shotgun Surgery smell

2. *Explain why the class/method is flagged as smelly (be specific).*

The `property(ConfigurableProperty prop)` method is written inside an interface called `public interface Pdfsam {}.` This interface allows implementing a class that keeps information about the version of PDFSam software. This method is flagged as a Shotgun Surgery because whenever someone changes the method inside this interface, they need make changes at all the places in the codebase where the interface was implemented into a class. In this way, a small change in the method from the interface leads to a large number of small changes in the classes where this interface was implemented.

3. *Do you agree that the detected smell is an actual smell? Justify your answer.*

We partially agree with this method being flagged as a Shotgun Surgery. It is correct that if we change the method to say `public Char[] Pdfsam::property(ConfigurableProperty prop)`, which returns char array instead of a string, then we also need to make changes in places where the interface was implemented and the method was used. So, by definition of a Shotgun Surgery code smell, this is technically a bad smelling method. However, we do not completely agree because, there is close to zero probability that someone or authors themselves will change this top-level interface in the PDFSam codebase. Also, it does not make sense to change the method to return anything else other than a String or input anything other than a ConfigurableProperty prop type input. Therefore, even though this is flagged as a bad smell, we think that this is not an issue in terms of a good object-oriented design practice.

References

[1] https://github.com/abchawla/cs581A3-f18-a2-gitrams-jedit

[2] https://github.com/abchawla/cs581A3-f18-a2-gitrams-pdfsam

[3] Eclipse Metrics plugin 1.3.8: http://metrics2.sourceforge.net/

[4] Henderson-Sellers, Brian et al. "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)." *Object Oriented Systems 3* (1996): 143-158.

[5] https://users.encs.concordia.ca/~nikolaos/jdeodorant/

[6] http://loose.upt.ro/reengineering/research/iplasma

[7] Fontana Francesca et al. "An experience report on using code smells detection tools." *Fourth International Conference on Software Testing, Verification and Validation Workshops* (2011).

[8] Lanza, Michele, and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media (2007).