Code Smells and Refactoring

Evolutie – Kevin Bruhwiler and Jason Stock

1 Detecting Code Smells

Code smells within jEdit and PDFsam are automatically detected using a variety of static analysis tools. Initial experiments are done using JDeodorant[1] to detect smells with suggestions for resolving them. This tool is used to identify five different smells, namely Feature Envy, Type Checking, Long Methods, God Class, and Duplication. However, it is not guaranteed that every component will fall in one or many of these categories. Furthermore, duplication resolution recommendations are only computed if a clone report exists. Therefore, to detect the level of duplication in each system we use the NiCad5 Clone Detector[2] developed by James R. Cordy and Chanchal K. Roy. We use the default configuration over all methods with a minimum of 10 lines having a difference of greater than 30%.

JDeodorant provides a targeted analysis on classes and components that are chosen and does not provide quantitative statistics on the project (e.g., number of bad smells per class or method in a project). Therefore, we first analyze each project with PMD[3] and Designite for Java[4] to obtain a comprehensive overview of code violations and the number of bad smells per class. Both of these tools report on hundreds of rule based metrics but not every issue is defined as a code smell and there are no direct instructions to resolve such issues. However, we investigate how these metrics help to identify the smelliest classes and the code smells that they report.

1.1 JEdit Analysis

Results from PMD show significant counts in rule violations among all classes as shown in Figure 1. For example, the `org.gjt.sp.jedit.bsh.Parser` class has 1,616 violations with the majority being control statement braces, method naming conventions, and unnecessary modifiers. It is of no surprise this is high as the class is composed of 5,932 Lines of Code (LOC) and 351 Number of Methods (NOM) reported by Designite. However, JDeodorant reports no code smells for this class in any category except duplication. Therefore, we exclude this class from our analysis and continue with the preceding violators.

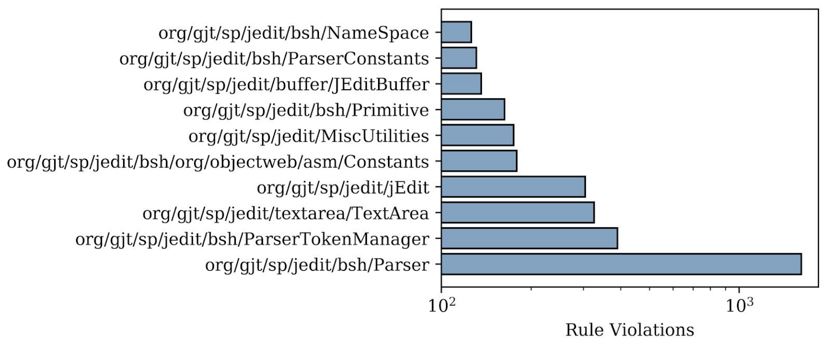

Fig. 1: Ten classes with the highest rule violation reported from PMD on jEdit.

[1] https://github.com/tsantalis/JDeodorant
[2] https://www.txl.ca/txl-nicaddownload.html
[3] https://pmd.github.io
[4] https://www.designite-tools.com/designitejava/

	Duplication	God Class	Long Method	Type Checking	PMD	Designite
TextArea	***X***	***X***	***X***	***X***	MethodNamingConventions GuardLogStatement ProperCloneImplementation	Complex Method Magic Number Long Statement
JEditBuffer	***X***	***X***	***X***	-	FormalParameterNamingConventions GuardLogStatement ControlStatementBraces	Long Statement Complex Conditional Missing Default
Primitive	***X***	-	***X***	***X***	FormalParameterNamingConventions BooleanInstantiation PreserveStackTrace	Complex Conditional Magic Number

Table 1: Overview of the different code smells in jEdit as reported by different tools. An ***X*** signifies that the violation is detected in JDeodorant, and PMD and Designite show a subset of the top high priority items.

We capture three classes from the top ten with the highest rule violations in PMD that detail numerous code smells from each analysis tool. Table 1 summarizes these results at a high level. Each class resides in a different package and are disjoint in functionality but have similar code smells.

`org.gjt.sp.jedit.textarea.TextArea`

The `TextArea` class suffers from many code smells. JDeodorant identifies multiple instances of every code smell, including six instances of feature envy. The offending methods are `addExplicitFold(int caretStart, int caretEnd, int lineStart, int lineEnd)`, `joinLineAt(int line)`, `setText(String text)`, `getSelectedText(Selection s)`, `turnOnElasticTabstops()`, and `deletePrevCodePoint(int offset)`. In five of these cases, JDeodorant recommends moving the methods to the `JEditBuffer` class. In the sixth case, `getSelectedText(Selection s)`, it recommends moving it to the `Selection` class instead.

For most of these methods, the cause is obvious. The `addExplicitFold` method, for example, operates entirely on data retrieved from a `JEditBuffer` with the intention of modifying that data and sticking it back into the `JEditBuffer`. Not only that, but it does not use a single instance variable from the `TextArea` class. The other four methods all follow the same pattern: none of them use any instance variables, all use data retrieved from a `JEditBuffer` class, and all ultimately either set or remove data in the buffer, with the data that is set or removed computed exclusively by data that was already in the buffer. The sole exception, `getSelectedText`, also largely operates on a buffer object and does not make use of instance variables, however it makes one more call to the `Selection` object than the `JEditBuffer` object, which is likely the reason that JDeodorant thinks it would fit better in the `Selection` class.

It's difficult to disagree with JDeodorant's assessment of these methods. They were most likely placed in the `TextArea` class because they are called by events, and the caller has a reference to the `TextArea` but not the `JEditBuffer` associated with the `TextArea`. Under this assumption, the ideal solution would be to move the methods in the `JEditBuffer` class and give those callers a reference to the buffer, which could be as simple as adding a getter to the `TextArea`. Even the method that

JDeodorant thinks should be moved to `Selection`, `getSelectedText`, would be better placed in `JEditBuffer`. Because the `Selection` object it's envious of is passed as an argument it may be complicated to ensure that the `Selection` in question can reference the `JEditBuffer` without going through `TextArea`, leaving the `Selection` object as an argument is likely to be the simplest solution.

<u>**`org.gjt.sp.jedit.buffer.JEditBuffer`**</u>

`JEditBuffer` contains several code smells, including god class, long methods, and code duplication. First, the entire `JEditBuffer` is identified as a god class. This is not unreasonable, as the `JEditBuffer` class is responsible for maintaining a buffer, reading from it, editing it, listening to events that occur on it, and locking it for concurrent access. In some of these cases changes are clearly necessary (and may simplify code). There is a good deal of locking-related code in this class and extracting it to another class would not only simplify a large chunk of the `JEditBuffer`, it would also create a class useful for managing concurrency in other issues. Creating a new class that handles all the logic around the buffer listeners would also be helpful as there is currently a great deal of complex code surrounding responding to the events on the buffer. On the other-hand, creating a new class that effectively acts as a middle-man between the buffer and the buffer listener could introduce new code smells. JDeodorant also recommends extracting the editing and the reading functionalities to new classes, however I'm uncertain if I agree with that assessment. In both cases the code extracted would be a single method and a single instance variable, which seems rather trivial to be given its own class.

`JEditBuffer` also suffers from a plethora of long methods, too many to list, but they generally fall into a few categories. "Fire" methods, which a solely responsible for calling methods on all of the buffer listeners, token markers, which use a very complicated process to identify certain tokens in the buffer, manipulator methods (undo, redo, insert, etc…) which manipulate both the content of the buffer and the caret position, and getters methods which are often bafflingly complicated, running through numerous nested if-statements and try-catch blocks. I'm inclined to agree with JDeodorants assessment of every method identified. Not only are many of the methods exceptionally long, running over 20 or 30 lines of code, they're frequently very difficult to understand. Extracting out smaller methods would not only increase the maintainability of the code, but giving them good method names would make it much easier to understand for newcomers to the code. The exception to this might be the fire methods, which are not particularly long or hard to understand and would be better solved by the method described below.

One of the major culprits of the duplication code smell is the fire methods. Eight different methods have 83% code similarity and are all responsible for iterating through the list of buffer listeners and calling a single method on them. In effect, these are all exactly the same method with only a single line of difference, clearly validating the code smell. That said, reducing them all to a single method could be challenging in Java, as dynamically passing methods as arguments is not easy. One solution might be a large conditional block or switch statement, however that is likely to introduce more code smells. Using the `getMethod()` of the class object to dynamically call methods based on a `String` argument would be a much cleaner solution, and would allow for all of the fire methods to be compressed into a single method.

Exactly two instances of duplicate code in `JEditBuffer` are seen having 100% similar code. Each instance contains two different methods within the same class, and share 11 and 23 lines of code for the `getLineText`/`getLineSegment` and `undo`/`redo` methods, respectively. Both of these methods act on data local to their method and class members. The only difference between their counterparts are the one or two method invocations. Therefore, if these calls can be generalized, then the code smell will be removed.

`org.gjt.sp.jedit.bsh.Primitive`

Numerous code smells are detected by all of the analysis tools. NiCad5 displays three clone instances with upwards of 82% method similarity in the worst case. The offending methods in this scenario, namely `intUnaryOperation(Integer I, int kind)` and `longUnaryOperation(Long L, int kind)`, are both static and perform identical functions with differences in parameter types. Both methods utilize switch statements and rely on constants from the `org.gjt.sp.jedit.bsh.ParserConstants` interface. If a new unary operation is added, then both methods will have to be modified similarly. This increases the risk of introducing bugs; hence the item being identified as a code smell.

JDeodorant only marks the `Primitive` class with having one long method, namely `castPrimitive(Class toType, Class fromType, Primitive fromValue, boolean checkOnly, int operation)`. This method has 21 if-else statements and over 80 LOC. All interactions within this method are with static global variables and the parameters passed by the caller. The logic is mostly checking preconditions and the validity for casting, and could easily be broken into individual methods. It is clear this method is smelly and should be refactored via extraction to resolve the issue. By implementing new methods that consolidate logic, the `castPrimitive` method will be easier to comprehend and maintain.

The only other code smells reported by JDeodorant for the `Primitive` class are the ten methods that exhibit type checking. These are all a form of unary and binary operation that perform a transformation on data passed to the method based on the `int kind` parameter (e.g., `intUnaryOperation` previously mentioned). A switch statement is used to check the value of this variable based on the constants within the `ParserConstants` interface. Type checking is problematic because it signifies that additional conditions must be checked to cope with poorly abstracted objects that have been lost in translation. That is, an object is being acted on in a way where the type of said object is forgotten. However, this is not necessarily the case in the `Primitive` class. Specifically, the flagged methods are statically invoked to perform a known operation on a known object (e.g., binary negation of an int).

Refactoring steps are suggested by replacing the conditional structure with polymorphic invocations and adding a method to switch over the kind of operation specified. This will replace every unary and binary operation with an object that performs said operation. Doing so will be redundant as value checking of the parameter kind is still performed and the single statement operation is replaced by over 200 new classes for all primitives. Therefore, we do not agree that the detected code smell is actually a smell.

Three high priority violations from PMD are evaluated, including: FormalParameterNamingConventions, BooleanInstantiation, and PreserveStackTrace. The first is a violation of code style and could lead to confusion amongst teams. For example, the `intUnaryOperation` method passes an integer `I` with capitalized naming, which often denotes a class name. The usage of `I` is local to its method and referenced only once. We agree that this is a code smell, but the impact is very minor for this class. BooleanInstantiation is caught in various methods that try to incorrectly instantiate a new boolean object (e.g., the `floatBinaryOperation(...)` method). This impacts space and time performance, and the recommended solution from the JDK is to make use of the static factory `valueOf(boolean)` method.

Furthermore, the use of deprecated code should be avoided as future upgrades may not contain the soon-to-be-retired code, hence being flagged as a code smell. Finally, the PreserveStackTrace violation, seen in the `getDefaultValue(Class type)` method on line 890, is problematic as the stack trace of the original exception could be lost, resulting in the inability to follow an error to the source. This is considered a code smell as the smelly code will work, but increase the proneness to finding bugs.

Two categories of implementation smells exist in the `Primitive` class that influence maintainability of the application. First, is the use of complex conditionals in various methods, such as: `Primitive(Object value)` and `binaryOperation(...)`. The `Primitive` constructor, for example, has an expression with three different conditionals that act on the object parameter and static final fields of `Primitive.Special`. Having a high conditional count decreases code readability and subsequently impacts maintainability of the application. A compliant solution to fix this code smell is to create a new method for the two similar checks by means of extraction to reduce the conditional count.

Magic numbers only appear once throughout the class, however, it is not a significant code smell nor one that we agree with. The `hashCode()` method uses an arbitrary number times the hashcode of an instance variable of the class. A hash code is meant to provide a numeric representation of an object, and oftentimes an arbitrary prime number is used to equally distribute the hash.

1.2 PDFsam Analysis

We conduct a similar analysis with PDFsam by first evaluating the ten classes with the highest number of rule violations reported by PMD (Figure 2). The results are significantly lower than jEdit with only four violations marking the highest class. This observation is evident across all analysis tools. NiCad5 reports 51 clone pairs over 12 classes as compared to the 944 clone pairs over 44 classes in jEdit. Feature envy from JDeodorant is obsolete in PDFsam and type checking is only found in two classes within the `pdfsam-fx` module.

Overall, code smells in PDFsam are sparse in existence, and the distribution of unique types and their counts are low. This is likely due to having smaller classes w.r.t. NOM and LOC that appear structurally modular. We study three individual classes with unique violations from PMD and Designite as well as those reported by JDeodorant (Table 2).

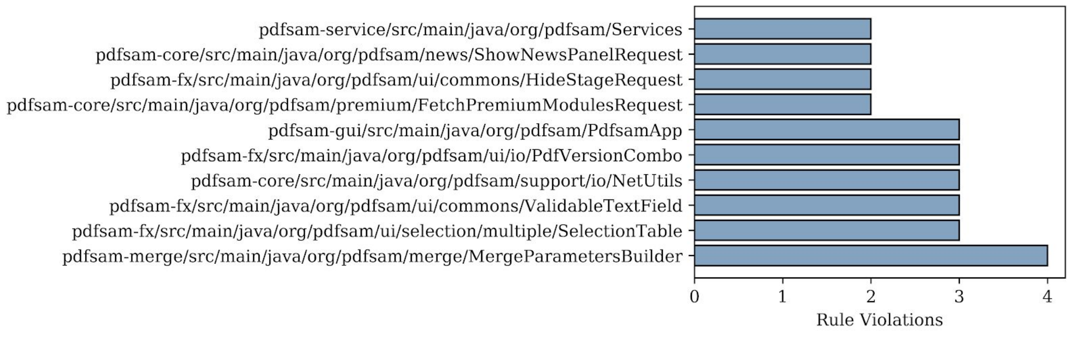

Fig. 2: Ten classes with the highest rule violation reported from PMD on PDFsam.

	Duplication	God Class	Long Method	Feature Envy	PMD	Designite
MergeParametersBuilder	***X***	***X***	-	-	UnusedImports LooseCoupling UnusedLocalVariable	-
SelectionTable	-	***X***	***X***	**X**	OptimizableToArrayCall UnnecessaryFullyQualifiedName	Long Statement Complex Method Magic Number
PdfsamApp	-	***X***	***X***	-	SingularField UnnecessaryFullyQualifiedName UseLocaleWithCaseConversions	Complex Conditional Long Statement

Table 2: Overview of the different code smells in PDFsam as reported by different tools. An ***X*** signifies that the violation is detected in JDeodorant, and PMD and Designite show a subset of the top high priority items.

`pdfsam-merge::org.pdfsam.merge.MergeParametersBuilder`

PMD identifies a number of code smells related to the MergeParametersBuilder class, amusingly mostly introduced by the changes we made in A2. Some of them are obvious: there is an unused local variable in the addInput method that was never removed, the class is loosely coupled due to the fact that all the methods take different arguments (not because of our changes), and one of the imports, the `NullSafeSet`, was not removed after we replaced it with an `ArrayList`. These are all valid smells, although it would be difficult to fix the loose coupling without dramatically changing the structure of all of the builder classes and adding many more small, data-holders; something which would likely add code smells, and seems generally unnecessary.

JDeodorant identifies two major code smells: duplication and god class. The duplication is caused by two similar if-statements in the addInputs method which both and `PdfMergeInput` objects to the Inputs array list. While it would not take much effort to extract them into a separate method, it also seems unnecessary. The additional method calls would not reduce the length of the code, and the additional method would actually make the class larger. While it does contain a couple blocks of similar code, I'd be far more inclined to consider the method to be too long rather than containing duplication.

The second code smell that JDeodorant identifies, god class, is also the result of the changes in A2. In this case, the amount of code added related to the inputs object has JDeodorant arguing that inputs should be its own class. I'm on the fence about this one: the input class would be very small, with only a single instance variable and a single method, and would only be accessed by the `MergeParametersBuilder` class. On the other-hand, code relating to the inputs object makes up almost half of the functional code in `MergeParametersBuilder`, and does seem to relate a different concept than the rest of the class.

`pdfsam-fx::org.pdfsam.ui.selection.multiple.SelectionTable`

The `SelectionTable` class is relatively large for PDFsam with a total of 325 LOC and 22 NOM. It comes as a surprise that there are no reports of duplication from NiCad5, which suggests that there may be no direct correlation between class size and duplicate code. However, other existing code smells still remain as reported by the different analysis tools.

JDeodorant identifies a single reference of a god class for operations that relate to the `private Label placeHolder` object. This object is used to provide a contextual label when there are no items to be shown in the table. The state of the `placeHolder` is updated via `onDragEnteredConsumer()`, `onDragExited(DragEvent e)`, and the constructor. These methods, except the constructor, provide no functionally specific to the `SelectionTable` class outside of updating the labels state. This can lead to difficulties in understanding what the class does and how the methods interact. The code smell can be fixed by extracting the instance variable and these methods to a specific class following object oriented design. We agree that by doing so, we can narrow the functionality of the `SelectionTable` class and increase code comprehension.

Three long methods are identified within the class, including:

- `showPasswordFieldPopup(ShowPasswordFieldPopupRequest request)`
- `initTopSectionContextMenu(ContextMenu contextMenu, boolean hasRanges)`
- `initItemsSectionContextMenu(ContextMenu contextMenu, boolean canDuplicate, boolean canMove)`

These are each flagged as a code smell because their length increases complexity which decreases maintainability. Each method is multipart with numerous blocks of code and conditional statements. For example, `initItemsSectionContextMenu(...)` first computes some logic using the first parameter then has two seperate if statements based on the subsequent booleans. All computations act on local variables and static final constants from different enumerators. Multiple conditionals increase the cyclomatic complexity which leads to difficulties understanding what the method does and how to test it. Furthermore, multiple blocks of code suggests the method is not modular with a single function. This is like writing an entire program in the main method. Correct object oriented design should break functionality down into unified components. Therefore, we agree each of these methods are valid code smells and should be extracted into multiple supporting methods.

Only the OptimizableToArrayCall and UnnecessaryFullyQualifiedName rule violations are caught by PMD. Both of which are classified with medium-low priority, and do not significantly impact the code. The `restoreStateFrom(Map<String, String> data)` method uses a fully qualified name for the static method `Optional.`*`ofNullable`* where the `initDragAndDrop(boolean canMove)` method does not. In this case, there is an import for both `Optional` and `Optional.`*`ofNullable`*. While both invocations only utilize local data, their method of calling is inconsistent and potentially misleading. The occurrence for the UnnecessaryFullyQualifiedName exists within the `onMoveSelected(final MoveSelectedEvent event)` method and is considered a code smell. The poor implementation of allocating the size of an array within the `Collection.toArray()` method leads to poor performance in the application. While both violations only introduce minor side effects, we agree that their existence influences usability and maintainability.

Designite reports three main code smells (i.e., long statements, complex methods, and magic numbers) across a number of different methods. However, the `initDragAndDrop(boolean canMove)` method is a great candidate showing all three instances. The structure of this method is quite complex with 12 conditional statements all within a single extended lambda expression. Such expressions were introduced in JDK 8, and introduce powerful semantics. The `initDragAndDrop` method is filled with nested lambda expressions all interacting with local variables and class methods and instance variables. Magic numbers come from using decimal values to set opacity levels for the drag and drop table. This is not much of a code smell as there are no implications and the details are straight forward. The violations for long statements and the complex method do impact the maintainability and understandability of the class. Thereby increasing the possibility of opportunity for unexpected bugs and are considered code smells.

pdfsam-gui::org.pdfsam.PdfsamApp

Similar to the `SelectionTable` class, the `PdfsamApp` class has no duplication but an instance of god class and long methods with violations from PMD and Designite. A total of 22 methods of which five are public and eight private fields construct the class. We will begin by discussing the code smells reported by JDeodorant and follow with the individual violations reported by PMD and Designite.

A private `UserContext` instance variable is created to hold user related settings. Within the `PdfsamApp` class the user context is initiated and saved via method invocations from said class. While this functionality is important it acts as a supporting feature when the application is started and exited. As such, delegated as a god class with potentially misplaced functionality. JDeodorant provides a fix to this by extracting the instance variable and the supporting methods (`saveWorkspaceIfRequired` and `initActiveModule`). However, it appears that one of the methods, namely `loadWorkspaceIfRequired`, is not picked up in this resolution but should be included as an extraction.

Three instances of long methods are identified and two of which are the same method. Perhaps the `start(Stage primaryStage)` method is a misidentification from JDeodorant as the two instances offer the same resolutions. Furthermore, the third method `startLogAppender()` is only four short statements long. These statements create a new local object, sets a variable within the object, and then instanties a new object for the class from the local variable. Extracting three of these lines into a new method is viable, but it could interfere with understanding the class more than it would help.

It is hard to argue that the god class violation is not code smell, but the instances of long methods are questionable. Fixing the detected god class will isolate appropriate method invocations, thereby improving maintainability. Similarly, fixing the `start` method by means of extracting code and shortening the method will aid in understanding of the class.

Within the entire class there are three violations reported by PMD. The first is the SingularField design smell by having the private instance variable `broadcaster` only being referenced once in the `startLogAppender` method of the offending class and nowhere else in the application. This could instead be replaced by a local variable in the scope of the method. Secondly, a violation of an UnnecessaryFullyQualifiedName is referenced for the `Optional.`*`ofNullable`* method. This is redundant as multiple imports exist for the `Optional` class and this method. Finally, there is a violation of UseLocaleWithCaseConversions on line 323 in the public method `getOpenCmd(String url)`.

Including the locale does not enforce a reliance on the JVM setup to get expected conversion results. Moreover, it will reduce the risk of unintended behavior and improve communication to developers. We agree that both the SingularField and UseLocaleWithCaseConversions violations are indeed code smells that if fixed will improve maintainability and reduce the risk of introducing bugs in the application.

Designite identifies two violations (i.e., complex conditionals and long statements) within the `PdfsamApp` class. The former can be seen within the `verboseIfRequired()` and `getOpenCmd(String url)` methods. Having expression with three or more conditions reduces readability and increases testing challenges since there are more execution branches through that given expression. This can be fixed by extracting related conditions to a new method that can be tested on its own. The latter violation, long statements, can be seen within the `loadWorkspaceIfRequired` method. Specifically, the first statement follows a message chain with a series of method invocations. This is problematic as a change to one of these methods will propagate to the surrounding methods and possibly lead to bugs. A resolution will be to separate method invocations to multiple lines which will improve readability and maintainability of the software.

2 Evaluation Overview

A number of different code smells have been identified in the two applications during our analysis phase. In this section we will select two components for each application containing distinct smells and refactor the code and add supporting tests. Thereafter, we validate resolution of such smells by reevaluating the output of JDeodorant, PMD, and Designite.

2.1 JEdit Evaluation

`org.gjt.sp.jedit.buffer.JEditBuffer`

NiCad5 reports 100% similarity for two different clone instances within `JEditBuffer`. We remove the **duplication** by means of composing new methods within the combined functionality that can be called. The first instance contains 23 lines of identical code in the `undo()` and `redo()` methods in a variety of different steps, including:

1. Create a new custom functional interface named `Procedure` with one abstract method, `execute()`, that accepts no arguments and returns void. This interface will be used in conjunction with method references to execute the expected operations generically.
2. Copy and paste the `undo`/`redo` method content into a new method named `unredo(...)` that will replace the duplicate functionality and later be called from both methods.
3. Locate the exact statements that require updating — there are exactly three method invocations, including: `fireBeginUndo/Redo`, `undoMgr.undo/redo`, and. `fireEndUndo/Redo`.
4. Add the following parameters alongside `TextArea textArea` to generalize the three method invocations for their respective calls: `Procedure fireBegin, Supplier<Selection[]> operation, Procedure fireEnd`. The `java.util.function.Supplier` interface is used for the call to `undoMgr.undo/redo` as there are no parameters supplied, but value of type `Selection[]` is returned.
5. Replace the three statements that need to be generalized as found in step 3 by calling `execute` and `get` on the `Procedure` and `Supplier` methods, respectively.

6. Remove the duplicated code from the both the `undo` and `redo` methods, and replace with a call to `unredo` with method references to the associated procedural and supplier methods. For example, the `redo` method will now only contain one line:

   ```
   unredo(textArea, this::fireBeginRedo, undoMgr::redo, this::fireEndRedo);
   ```

7. No other references are updated as the changes are only within `undo/redo` which call the newly created method `unredo`.

The other clone instance refers to the two methods `getLineText` and `getLineSegment` with 11 identical lines. Similarly, we compose a new method and replace the duplicate code with a single line invocation as detailed with the following steps:

1. Copy and paste the `getLineText/getLineSegment` method content into a new method named `getLineDefault(...)` that will replace the duplicate functionality and later be called from both methods.
2. Locate the exact statements that require updating — there are exactly one method invocations to either `getText(...)` or `getSegment(...)`from either the `getLineText` or `getLineSegment` method, respectively.
3. Add the parameter `BiFunction<Integer, Integer, R> getter` alongside `int line` to generalize the method invocation. The `java.util.function.BiFunction<T, U, R>` interface is used for the call to `getText/getSegment` as there are two parameters passed, and return a unique value of type `R` of either `String` or `CharSequence`.
4. Replace the one statement that needs to be generalized as found in step 2 by calling `getter.apply(T, U)` with the appropriate parameters, i.e., the same as `getText/getSegment`.
5. Remove the duplicated code from the both the `getLineText` and `getLineSegment` methods, and replace with a call to `getLineDefault` with method references to the associated method. For example, the `getLineSegment` method will now only contain one line:

   ```
   return getLineDefault(line, this::getSegment);
   ```

6. No other references are updated as the changes are only within `getLineText/getLineSegment` which call the newly created method `getLineDefault`.

Even though JDeodorant claims to provide automatic refactoring of duplicate code it was shown not to be fully compatible with NiCad5. Therefore, we had to perform all refactoring manually by following the extract method to place calls to the newly created method from offending locations.

`org.gjt.sp.jedit.textarea.TextArea`

JDeodorant has identified an instance of the **type checking** code smell in the `TextArea` class in the `delete(boolean forward)` method. We remove it using the “Replace Conditional with Polymorphism” refactoring type. The following steps describe the operations that were performed to remove the code smell:

1. Create a method in `Selection` class named `delete(boolean: forward, TextArea: textArea)`.
2. Extract the code that was previously inside the type-checked block to the new method.
3. Add an import in selection for `javax.swing.UIManager`, which is required for the newly created delete method.
4. Change the `tallCaretDelete(s: Selection.Rect, forward: boolean)` method in `TextArea` from private to public so that it can be called by `Selection` in the newly created method.
5. In place of the type checking block in `TextArea`'s `delete(forward: boolean)` method, call the new method in Selection, and pass the current TextArea as an argument.

The final result is that `TextArea` no longer checks the type of `Selection` it is working with, it simply tells `Selection` to delete and `Selection` decides how to handle that based on its internal state. This refactoring was done entirely with the guidance of JDeodorant's automatic refactoring tool.

2.2 PDFsam Evaluation

`pdfsam-fx::org.pdfsam.ui.selection.multiple.SelectionTable`

Several instances of **long method** are reported by JDeodorant for the `SelectionTable` class, in the `initTopSectionContextMenu` method, `initItemsSectionContextMenu` method, the `showPasswordFieldPopup` method, and the `initItemsSectionContextMenu` method. JDeodorant suggests using the Extract Method method to resolve the code smells. The following steps were taken to resolve the `initTopSectionContextMenu` code smell.

1. A new private method is created named `setDestinationItem` which takes no arguments and consists of the first eight lines of code previously in the `initTopSectionContextMenu` method.
2. The first line of the `initTopSectionContextMenu` method is changed to call the new `setDestinationItem` method.
3. Another private method is created named `setPageRangesItem` which also takes no arguments and contains all but the last line of code from the conditional block inside the `initTopSectionContextMenu` method.
4. All but the last line from the conditional block inside of the `initTopSectionContextMenu` method are replaced with a function call to the newly created `setPageRangesItem` method.
5. Added a new test for the `setPageRangesItem` method to verify that it has been correctly created and bound to the right `KeyCombination`.
6. Added a new test for the `setDestinationItem` method to verify that it has been correctly created and bound to the right `KeyCombination`.
7. Verified that all tests pass and build successfully completes.

The majority of the refactoring was done using JDeodorant's automatic refactoring tool, although the unit tests had to be manually written and verified. Extracting methods is not a complex task and does not require assistance from JDeodorant, although the informative naming and relative safety of an automated approach is a welcome convenience.

`pdfsam-gui::org.pdfsam.PdfsamApp`

God class is reported by means of the `UserContext` instance variable of the `PdfsamApp` class as discussed in section 1.2. A new class is created to hold a reference to the instance variable and a variety of extracted methods. The following steps describe in detail the refactorings used to remove the smell:

1. Create a new class, `PdfsamAppContext`, to hold reference to the `UserContext` member variable. This is a public class created in the `org.pdfsam` package alongside the `PdfsamApp` class.
2. Create a new getter method for the private `UserContext` variable. This is to be used by calling classes to access said variable.
3. Add a private log4j logger instance to the `PdfsamAppContext` class for logging in the method.
4. Move the three methods `saveWorkspaceIfRequired()`, `initActiveModule()`, and `loadWorkspaceIfRequired()` to the new class that references this variable. Their method declaration, return type, and visibility is not changed upon extraction. However, a reference to the `Application` class that `PdfsamApp` extends is passed to the `loadWorkspaceIfRequired` method. This does increase coupling but is essential for the method to function.
5. Create a new field to hold a reference of a new `PdfsamAppContext` variable within the `PdfsamApp` class. This object is initialized during declaration.
6. Update references to the `UserContext` by invoking the created getter method within the `PdfsamAppContext` class. This only occurs twice within the `init()` method.
7. Update references to the extracted supporting methods invocations in various calling methods in `PdfsamApp`. This is done by replacing each call to the associated methods within the `PdfsamAppContext` class. Only three references need to be updated and are identified in the `start` and `stop` methods.
8. No existing tests exist for these methods as found using the Eclipse IDE search functionality. We create a new test class named `PdfsamAppContextTest` to test the newly added methods.

A significant portion of refactoring in `PdfsamApp` occurs with support of JDeodorant's automatic refactoring. The `loadWorkspaceIfRequired` method was not previously identified and as such, we had to perform manual refactoring by directly copying the method to the new class. Resolving the god class does not require intensive interactions and could all be done manually. However, JDeodorant moves the required methods, composes a new getter method, and organizes state into an independent class.

Glossary

MethodNamingConventions

"Configurable naming conventions for method declarations. This rule reports method declarations which do not match the regex that applies to their specific kind (e.g. JUnit test or native method). Each regex can be configured through properties."

GuardLogStatement

"Whenever using a log level, one should check if the loglevel is actually enabled, or otherwise skip the associate String creation and manipulation."

ProperCloneImplementation
"Object `clone()` should be implemented with `super.clone()`."

FormalParameterNamingConventions
"Configurable naming conventions for formal parameters of methods and lambdas. This rule reports formal parameters which do not match the regex that applies to their specific kind (e.g. lambda parameter, or final formal parameter). Each regex can be configured through properties."

ControlStatementBraces
"Enforce a policy for braces on control statements. It is recommended to use braces on '`if … else`' statements and loop statements, even if they are optional. This usually makes the code clearer, and helps prepare the future when you need to add another statement."

PreserveStackTrace
"Throwing a new exception from a catch block without passing the original exception into the new exception will cause the original stack trace to be lost making it difficult to debug effectively."

BooleanInstantiation
"Avoid instantiating `Boolean` objects; you can reference `Boolean.TRUE`, `Boolean.FALSE`, or call `Boolean.valueOf()` instead. Note that new `Boolean()` is deprecated since JDK 9 for that reason."

UnusedImports
"Avoid unused import statements to prevent unwanted dependencies. This rule will also find unused on demand imports, i.e. `import com.foo.*`."

LooseCoupling
"The use of implementation types (i.e., `HashSet`) as object references limits your ability to use alternate implementations in the future as requirements change. Whenever available, referencing objects by their interface types (i.e, `Set`) provides much more flexibility."

UnusedLocalVariable
"Detects when a local variable is declared and/or assigned, but not used."

OptimizableToArrayCall
"Calls to a collection's `toArray(E[])` method should specify a target array of zero size. This allows the JVM to optimize the memory allocation and copying as much as possible."

UnnecessaryFullyQualifiedName
"Import statements allow the use of non-fully qualified names. The use of a fully qualified name which is covered by an import statement is redundant. Consider using the non-fully qualified name."

SingularField
"Fields whose scopes are limited to just single methods do not rely on the containing object to provide them to other methods. They may be better implemented as local variables within those methods."

UnnecessaryFullyQualifiedName
"Import statements allow the use of non-fully qualified names. The use of a fully qualified name which is covered by an import statement is redundant. Consider using the non-fully qualified name."

UseLocaleWithCaseConversions
"When doing `String::toLowerCase()/toUpperCase()` conversions, use an explicit locale argument to specify the case transformation rules."

Complex Method
A method has a cyclomatic complexity (i.e., the total number of traversable execution paths) greater than or equal to eight.

Magic Number
A non-zero/one numeric literal that has no contextual meaning in a statement. For example, the erroneous statement: `int prediction = 2000 * result;` gives no meaning to what the value of `2000` is.

Long Statement
Length of a syntactic expression for a single command has a length of greater than 120 characters.

Complex Conditional
The number of boolean sub expressions in a single `if` statement is greater than or equal to three.

Missing Default
A `switch` statement is missing the default case clause. Unexpected behavior may exist if no conditions are met within the expression.