Opening various image formats

Software Maintance Autumn 2023

Magnus Christian Larsen

February 28, 2024

GitHub Username: Autowinto
Student Mail: magla21@student.sdu.dk
Project: https://github.com/Autowinto/JHotDraw
Date: 28/02/2024

Contents

1 Change Request

For this project in the Software Maintanence course, I decided to refactor the feature that allows the user to open and load various files into the software. The feature itself had several subfeatures, namely subfeatures for jpg, png, svg, text etc., but for this exact report, I decided to focus specifically on PNG files.

1.1 User Story

The change request, defined as a user story, is as follows:

"As a user, I want to be able to open image files on my computer, so that I can edit them."

1.2 Acceptance Criteria

The following subtasks as identified serve as acceptance criteria:

- When I click on the 'Open' option, the software should allow me to browse and select files in png, jpg, gif, pct, and text formats.
- Upon selecting a valid file of the mentioned formats, the software should display the image or content correctly within the workspace.
- If I attempt to open a corrupted or unsupported file type, the software should provide a clear error message.
- The software should maintain the original quality of the image when opening it.
- For text files, the software should provide an option to convert the text into an editable shape or object in the drawing workspace.

2 Concept Location

2.1 General Methodology

In locating classed for the refactoring process of the report, there are several options available to us within Featureous and NetBeans, which we of course will make use of in the future impact analysis section of the report. For actually finding the location of classes relevant to the feature, I mainly made use of the powerful tools provided by the IntelliJ IDE, such as the following:

- Quick find - Looking through the code manually, searching for relevant keywords and such.
- Find usages - Allows you to find usages of any given method or variable
- Go to implementation - Allows you to go directly to the implementation of a method
- Search within file - Allows you to search for any symbol or token within a single file.
- Inspect code - Allows you to see a bunch of metrics about code such as maturity, implementation issues and more, helping to pinpoint problematic parts of the code.

In addition to tooling provided by NetBeans and IntelliJ, there are also methods that are much less technical, such as simply removing parts of the code and seeing what happens and if said code is relevant to the feature that is being refactored.

2.2 Classes Overview

Table 1:

#	**Domain classes**	**Tool used**	**Comments**
1	LoadFileAction	Quick find	I took a look at the codebase as a whole, scanning through the different files and classes and quickly idscovered LoadFileAction, which has a name easily identifiable to be relevant to the feature that is to be refactored.
2	LoadRecentFileAction	Quick find	I took a look at the codebase as a whole, scanning through the different files and classes and quickly idscovered LoadFileAction, which has a name easily identifiable to be relevant to the feature that is to be refactored.

Continued on next page

Table 1: (Continued)

3	OpenFileAction	Quick find	Again, while taking a look through the codebase I found the OpenFileAction class which made sense to include as a relevant class.
4	OpenRecentFileAction	Quick find	Again, while taking a look through the codebase I found the OpenFileAction class which made sense to include as a relevant class.
5	AbstractSaveUnsavedChangesAction	Go to implementation	The AbstractSaveUnsavedChangesAction class was found by finding the implementation of the class that LoadFileAction extends.
6	AbstractApplicationAction	Go to implementation	The AbstractApplicationAction is extended by the OpenFileAction
7	AbstractViewAction	Go to implementation	The AbstractViewAction is extended by the AbstractSaveUnsavedChangesAction class.
8	AbstractAction	Go to implementation	Both the AbstractApplicationAction and the AbstractViewAction extends the AbstractAction class, which is worth looking into, as it doesn't make a lot of sense why there are so many different abstract classes that are extended from.
9	URIChooser	Go to implementation	The URIChooser is a class used by both the LoadFileAction and the OpenFileAction and has their own implementation in each class. It doesn't necessarily make much sense for two almost identical implementations to exist.

3 Impact Analysis

An impact analysis serves the purpose of helping a developer understand the implications and impact of performing a refactor or introducing a feature into a software project. This is especially relevant, as the group consists of 5 total members, each refactoring their own feature, which could very quickly result in overlapping code changes, code constantly breaking and the end product becoming unstable.

To perform this impact analysis, we're making use of the program featureous and FeatureEntryPoint annotations to annotate parts of the program that is planned to be refactored. By doing this, we can gain a further understanding of the interconnectivity of the program and see how the features overlap code-wise.

The FeatureEntryPoints created for my feature are for the following methods and constructors.

- OpenFileAction constructor
- OpenRecentFileAction constructor
- actionPerformed in OpenFileAction.java
- openViewFromURI in OpenFileAction.java
- showDialog in OpenFileAction.java
- createDialog in OpenFileAction.java

Based on these FeatureEntryPoints, the generated report can be used to analyze the impact of the feature.

3.1 Featureous Feature-Code Characterization

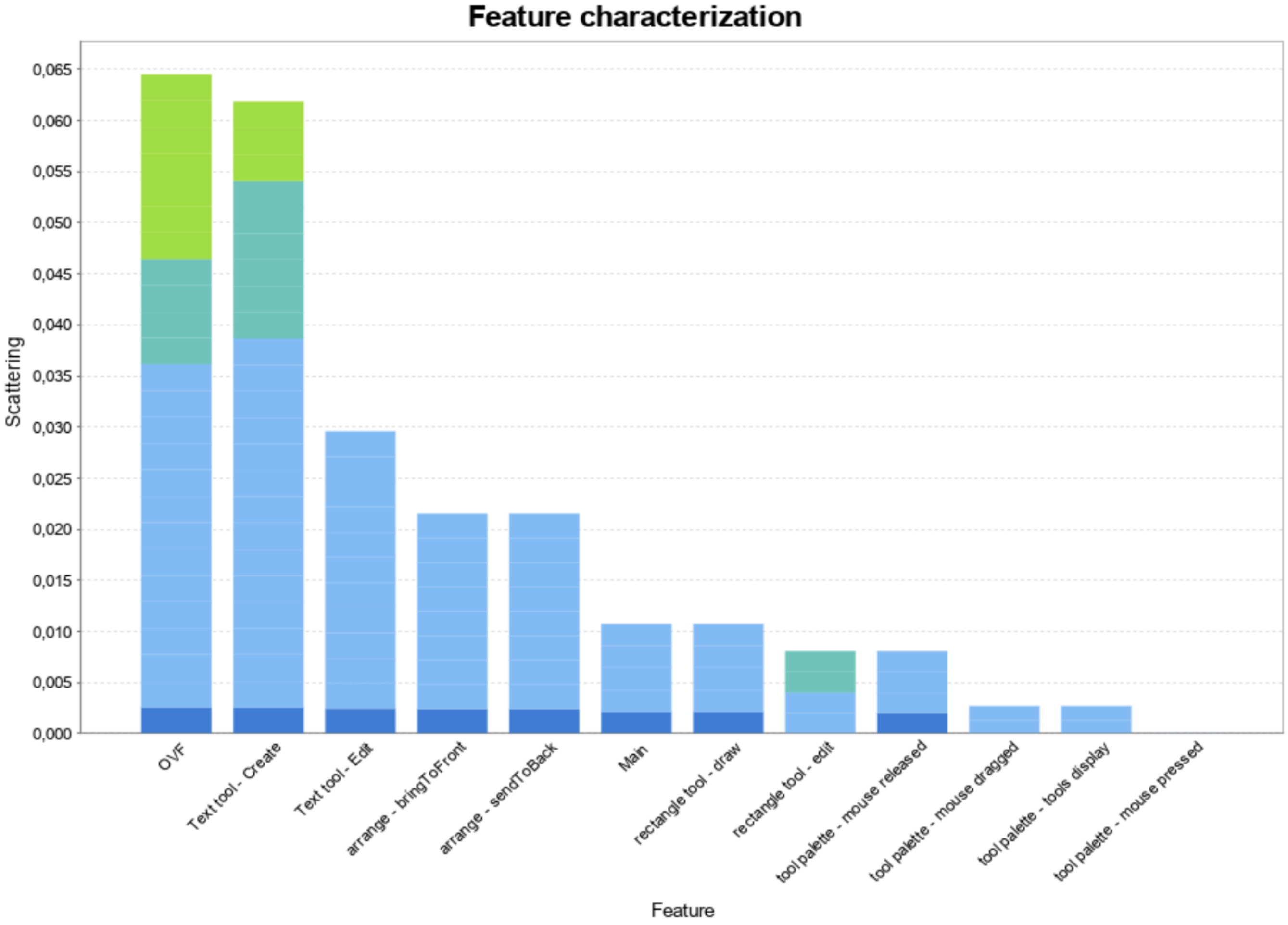
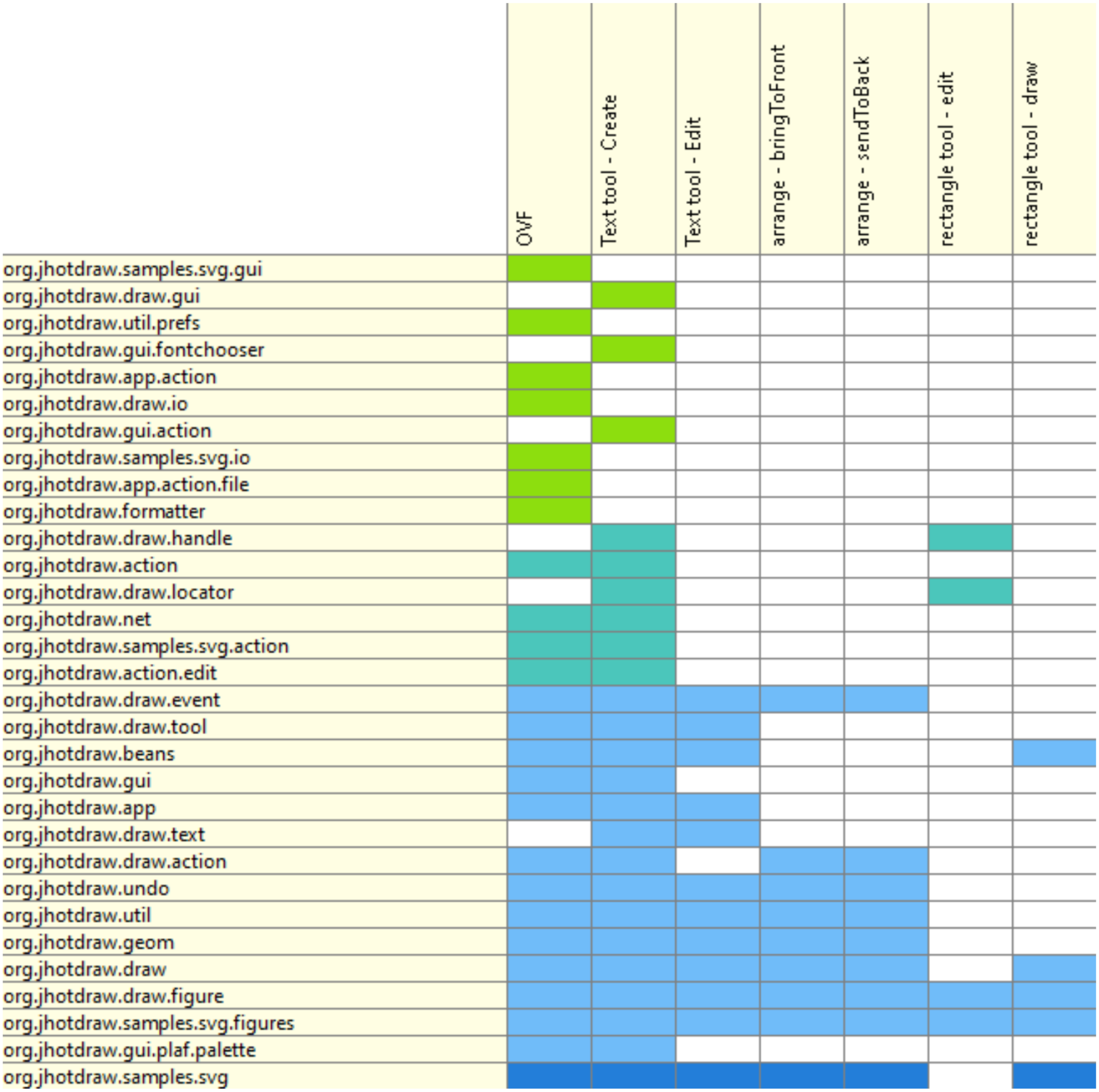


Figure 1: Feature-Code Characterization

The above graph from featureous shows that my feature, here annotated as OVF, has a great deal of overlap with other features and is very scattered in the codebase. This means, that when refactoring, I need to be extra careful not to accidentally break other features, causing additional refactoring between me and another team member to be necessary.

3.2 Feature Correlation Grid



	OVF	Text tool - Create	Text tool - Edit	arrange - bringToFront	arrange - sendToBack	rectangle tool - edit	rectangle tool - draw
org.jhotdraw.samples.svg.gui	🟩						
org.jhotdraw.draw.gui		🟩					
org.jhotdraw.util.prefs	🟩						
org.jhotdraw.gui.fontchooser		🟩					
org.jhotdraw.app.action	🟩						
org.jhotdraw.draw.io	🟩						
org.jhotdraw.gui.action		🟩					
org.jhotdraw.samples.svg.io	🟩						
org.jhotdraw.app.action.file	🟩						
org.jhotdraw.formatter	🟩						
org.jhotdraw.draw.handle		▩				▩	
org.jhotdraw.action	▩	▩					
org.jhotdraw.draw.locator		▩				▩	
org.jhotdraw.net	▩	▩					
org.jhotdraw.samples.svg.action	▩	▩					
org.jhotdraw.action.edit	▩	▩					
org.jhotdraw.draw.event	🟦	🟦	🟦	🟦	🟦		
org.jhotdraw.draw.tool	🟦	🟦	🟦				
org.jhotdraw.beans	🟦	🟦	🟦				🟦
org.jhotdraw.gui	🟦	🟦					
org.jhotdraw.app	🟦	🟦	🟦				
org.jhotdraw.draw.text		🟦	🟦				
org.jhotdraw.draw.action	🟦	🟦		🟦	🟦		
org.jhotdraw.undo	🟦	🟦	🟦	🟦	🟦		
org.jhotdraw.util	🟦	🟦	🟦	🟦	🟦		
org.jhotdraw.geom	🟦	🟦	🟦	🟦	🟦		
org.jhotdraw.draw	🟦	🟦	🟦	🟦	🟦		🟦
org.jhotdraw.draw.figure	🟦	🟦	🟦	🟦	🟦	🟦	🟦
org.jhotdraw.samples.svg.figures	🟦	🟦	🟦	🟦	🟦	🟦	🟦
org.jhotdraw.gui.plaf.palette	🟦	🟦					
org.jhotdraw.samples.svg	■	■	■	■	■		■

Figure 2: Feature Correlation Grid

Reading the grid reveals, that my feature has great entanglement with other features. There are several reasons why this could be the case. Firstly, my feature handles many things. Performing IO to load files, placing these image files in the GUI, drawing said feature on the screen and more. Therefore, the entanglement on display doesn't necessarily mean, that refactoring will be difficult; it does however show, that we should be wary when refactoring and extra careful that core features aren't broken when refactoring.

3.3 Feature Correlation Graph

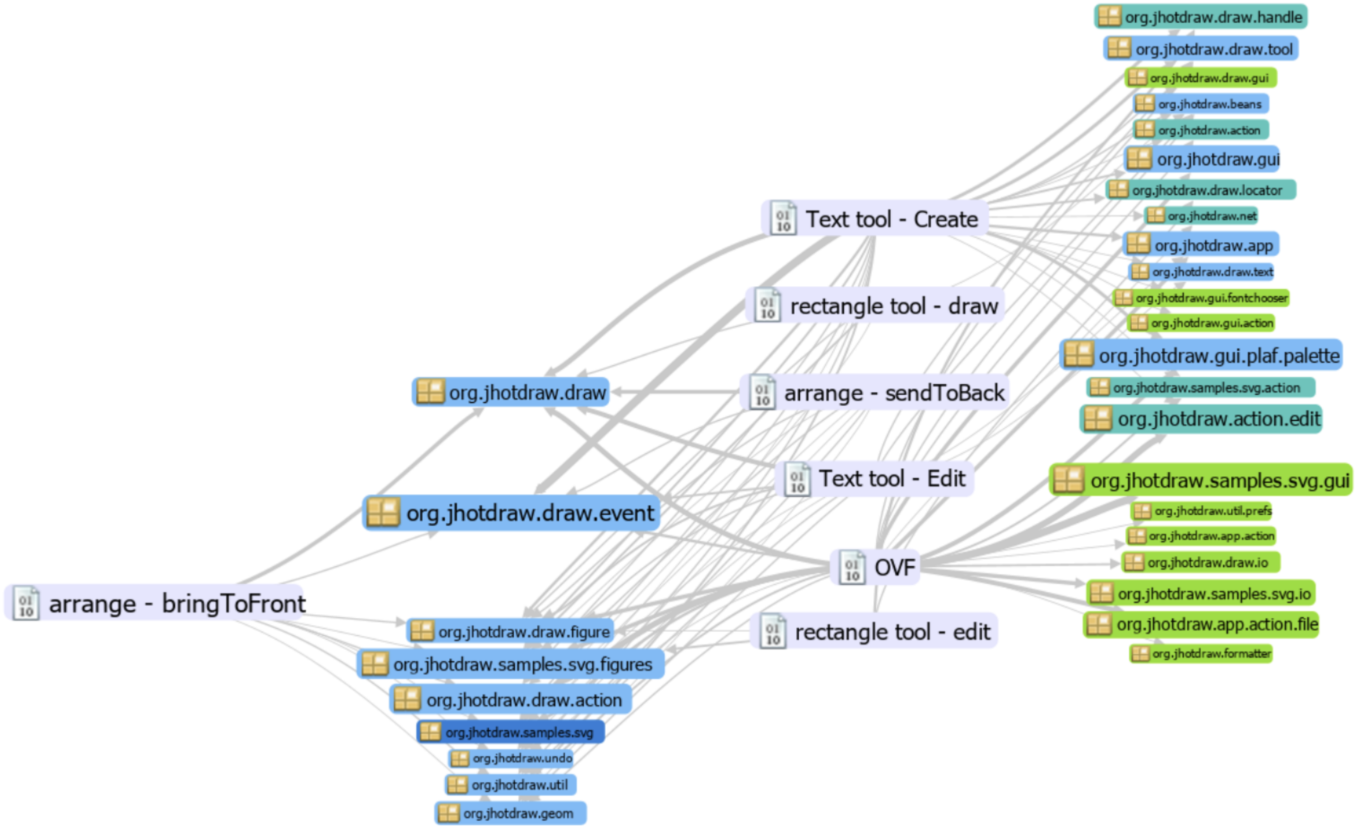


Figure 3: Feature Correlation Graph

The graph generated by the Featureous doesn't give us a lot of new information, it does however present it in another way, which gives further evidence to the fact, that while the code overlap previously seemed to be deeply ingrained, the overlaps happen in key packages such as the draw event package, figures, action etc. many of which, we won't be touching for the coming refactor or will be extra careful not to break.

3.4 Feature Relations Characterization

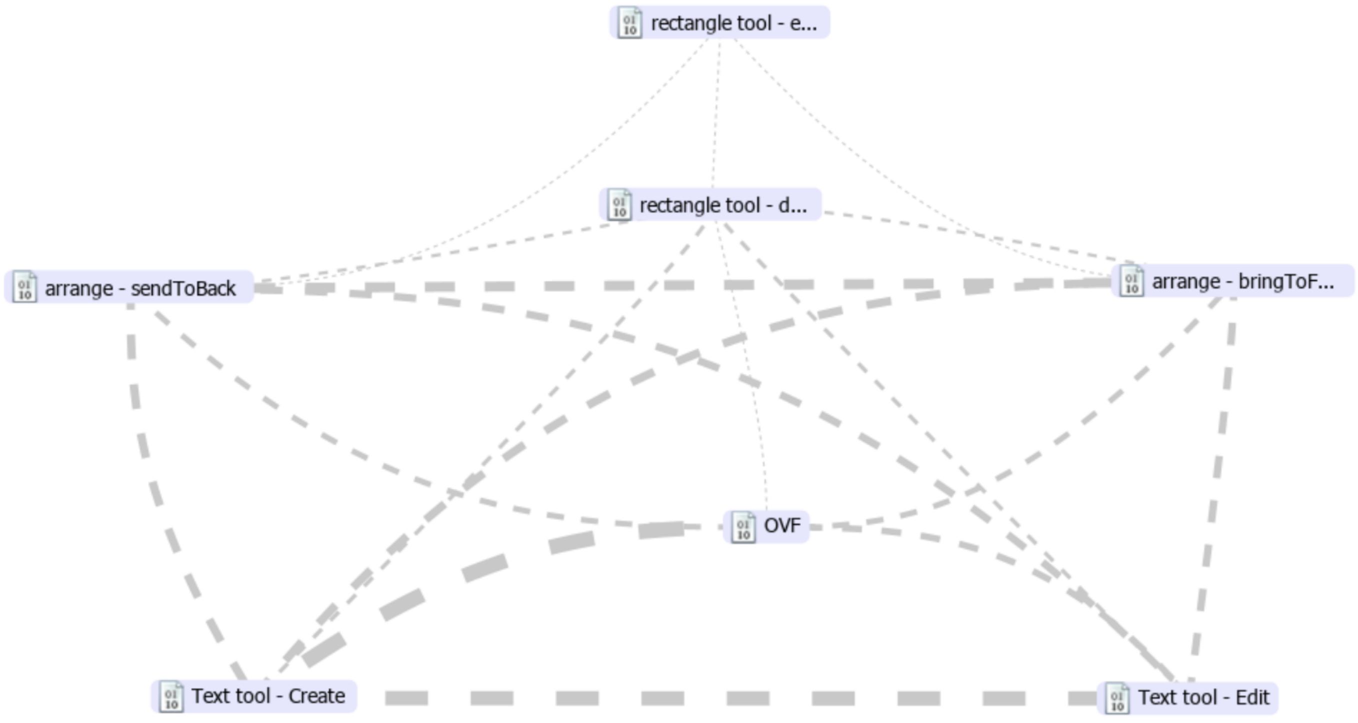


Figure 4: Feature Relations Characterization

Yet another visualization tool of the Featureous software, is the feature relations characterization, which is a simpler visualization, showing the amount of shared relations between the different features with a dashed line, increasing in thickness directly proportional to the amount of shared code. This is helpful if, like our feature correlation graph, the amount of packages on display is hard to read.

While a useful tool in many contexts, in my case, I feel it doesn't help me a lot, as my feature seemingly connects to most other features in one way or another and doesn't help me gain an entry point into analyzing my feature's impact.

3.5 Impact Analysis

The impact of the feature that I will be refactoring can be broken down in a table of classes and packages, helping with a general overview of the things that I need to be wary of when refactoring. These will be marked by one of the three following tags.

- **Unchanged**: These are classes that I visited in my search for relevant classes, but will not be touching.
- **Propagating**: These are classes that I looked at, and helped me find the correct classes that I was looking for.
- **Changed**: These are classes that I have planned to refactor based on my analysis.

Table 2: The list of all the packages visited during impact analysis.

Package name	**# of classes**	**Tool used**	**Comments**
action			
action.file			
action.app			

4 Refactoring Patterns and Code Smells

The class that are gonna be the focus of the upcoming refactor is the OpenFileAction class. In order to find relevant code smells, I made use of the SonarLint plugin for IntelliJ, which is specially created to identify non-critical code smells in code, helping to improve clarity and maintainability of the code written.

4.1 Duplicated Code

The two files are similar in functionality. Unfortunately, they are also very similar in code, violating the DRY principles[1], as they each have plenty of code that is duplicated across the clas that could very well be refactored into some shared class or helper methods. An example of this can be seen below with two code snippets from OpenFileAction and OpenRecentFileAction respectively that has almost identical code for finding an empty view.

```
            // Search for an empty view
            View emptyView = app.getActiveView();
            if (emptyView == null
                    || !emptyView.isEmpty()
                    || !emptyView.isEnabled()) {
                emptyView = null;
            }
            final View view;
            boolean disposeView;
            if (emptyView == null) {
                view = app.createView();
                app.add(view);
                disposeView = true;
            } else {
                view = emptyView;
                disposeView = false;
            }
```

Figure 5: OpenFileAction.java

[1] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

```
// Search for an empty view
View emptyView = app.getActiveView();
if (emptyView == null
        || !emptyView.isEmpty()
        || !emptyView.isEnabled()) {
    emptyView = null;
}
final View p;
if (emptyView == null) {
    p = app.createView();
    app.add(p);
    app.show(p);
} else {
    p = emptyView;
}
```

Figure 6: OpenRecentFileAction.java

This also isn't the only example of duplicate code being employed. In fact, the two classes are more or less identical and could greatly benefit from this refactor.

4.1.1 Refactoring Strategy

The strategy that we'll be employing to refactor the class, is to move the shared code into some kind of shared location or utility class, so that we reduce the size of the classes themselves and have an easier time splitting up the code. Additionally, as there are many action classes, future refactors could benefit from these utility classes as well.

It's also a possibility that an entire abstract AbstractOpenFileAction class could be employed.

4.2 Long Methods

Some of the methods of the classes are really long. In the case of the openViewFromURI method in OpenFileLocation, as many as 70 lines long. This is hard to read and created cognitive dissonance. We should refactor this into smaller methods and pieces of code.

4.2.1 Refactoring Strategy

To refactor this issue, we'll be splitting our code into smaller methods wherever it makes sense; splitting the responsibility of the method into smaller, more readable pieces.

4.3 Magic Strings

The two classes both contain multiple instances of magic strings, which even set the exact same value. This is unmaintainable, as if we were to ever want to change

4.3.1 Refactoring Strategy

Performing this refactor should be rather simple. The way I've decided to go about it, is to simply set the variable as a constant, as it doesn't change in the program and if there was a time when changing the variable when the program is running becomes relevant, it's much easier to simply change one place.

4.4 Commented Out Code and redundant comments

There are several places in the class where code is simply commented out instead of being removed. This creates cognitive dissonance and makes the code harder to read when developing, as you have to consider whether the code was left there for a reason. Code should be stored in version control. It shouldn't be stored in a comment.

Additionally, some comments like the below snippet are just unnecessary and create clutter.

```
    /**
     * Creates a new instance.
     */
```

Figure 7: OpenFileAction.java

4.4.1 Refactoring Strategy

Refactoring this is as simple as identifying unused commented-out code and deleting it.

4.4.2 Inner Classes

The software makes use of inner classes, such as shown below, which creates unnecessary nesting and makes the code harder to read.

```
        chooser.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if ("CancelSelection".equals(e.getActionCommand())) {
                    returnValue[0] = JFileChooser.CANCEL_OPTION;
                    dialog.setVisible(false);
                } else if ("ApproveSelection".equals(e.getActionCommand())) {
                    returnValue[0] = JFileChooser.APPROVE_OPTION;
                    dialog.setVisible(false);
                }
            }
        });
```

Figure 8: OpenFileAction.java

4.4.3 Refactoring Strategy

We'll be replacing these inner classes with lambda expressions, as this is much more readable and the moder n equivalent to inner classes anyway.

5 Refactoring Implementation

Explain where and why you made changes in the source code. That is, explain the difference between the actual change set compared to the estimated impacted set from the impact analysis. Which classes or methods did you create or change? Furthermore, describe used design patterns and reflect about improved design to improve future software maintenance.

Firstly, the part of the code that handles searching for an empty view and creating a new one to put the image if it does exist, was extracted into its own method. Below, the code as it was before can be seen.

```
            // Search for an empty view
            View emptyView = app.getActiveView();
            if (emptyView == null
                    || !emptyView.isEmpty()
                    || !emptyView.isEnabled()) {
                emptyView = null;
            }
            final View view;
            boolean disposeView;
            if (emptyView == null) {
                view = app.createView();
                app.add(view);
                disposeView = true;
            } else {
                view = emptyView;
                disposeView = false;
            }
```

Figure 9: The old snippet finding the view and setting the disposeView boolean

After the refactor, the code was cleaned up and extracted into its own method, which is much cleaner and more maintainable. Additionally, the large check for whether the view should be disposed, was refactored into a simple one-liner, as nothing else is needed in this case.

```
        View view = findEmptyView();
        boolean disposeView = view == null;
```

Figure 10: The function calls in the actionPerformed method

```java
    private View findEmptyView() {
        Application app = getApplication();
        View emptyView = app.getActiveView();
        if (emptyView == null
                || !emptyView.isEmpty()
                || !emptyView.isEnabled()) {
            emptyView = null;
        }
        return emptyView;
    }
```

Figure 11: The new findEmptyView method of the OpenFileAction class

The code prior to being refactored had multiple levels of nesting, which made it very hard to read and follow along, especially when if and else statements were tens of lines apart. This can be seen below.

```
            URIChooser chooser = getChooser(view);
            chooser.setDialogType(JFileChooser.OPEN_DIALOG);
            if (showDialog(chooser, app.getComponent()) == JFileChooser.APPROVE_OPTION) {
                app.show(view);
                URI uri = chooser.getSelectedURI();
                // Prevent same URI from being opened more than once
                if (!getApplication().getModel().isAllowMultipleViewsPerURI()) {
                    for (View v : getApplication().getViews()) {
                        if (v.getURI() != null && v.getURI().equals(uri)) {
                            v.getComponent().requestFocus();
                            if (disposeView) {
                                app.dispose(view);
                            }
                            app.setEnabled(true);
                            return;
                        }
                    }
                }
                openViewFromURI(view, uri, chooser);
            } else {
                if (disposeView) {
                    app.dispose(view);
                }
                app.setEnabled(true);
            }
        }
```

Figure 12: The old actionPerformed code of the OpenFileAction class

Refactoring for this part involved reversing the if statement, making sure to return inside of this reversed if statement so that the rest of the code isn't run, as well as extracting the large if statement which handled URIs being opened multiple times, into its own method.

```
    private void processViewsForUniqueURI(URI uri, boolean disposeView) {
        Application app = getApplication();

        if (!app.getModel().isAllowMultipleViewsPerURI()) {
            for (View view : app.getViews()) {
                if (view.getURI() != null && view.getURI().equals(uri)) {
                    view.getComponent().requestFocus();
                    if (disposeView) {
                        app.dispose(view);
                    }
                    enableApplication();
                    return;
                }
            }
        }
    }
```

Figure 13: The new processViewsForUniqueURI method

```
        URIChooser chooser = getChooser(view);
        chooser.setDialogType(JFileChooser.OPEN_DIALOG);

        if (showDialog(chooser, app.getComponent()) != JFileChooser.APPROVE_OPTION) {
            if (disposeView) {
                app.dispose(view);
            }
            enableApplication();
            return;
        }

        app.show(view);
        URI uri = chooser.getSelectedURI();

        processViewsForUniqueURI(uri, disposeView);

        openViewFromURI(view, uri, chooser);
```

Figure 14: The new non-nested snippet calling the processViewsForUniqueURI method

5.1 Result

Finally, after performing the refactor, we've succeeded in implementing our planned refactors. Useless comments have been removed from the class, inner methods have been replaced by lambda expressions making the code overall much cleaner.

Overall, the refactoring has been executed in a satisfactory manner where no interfaces or abstract classes have needed to be changed causing a ripple effect of refactors. Every refactor has been performed within the scope of its interface, allowing for minimal impact across the software, keeping it stable and preserving functionality.

This makes the code much easier to maintain and will lead to higher code quality in the future.

6 Verification

At class level document unit tests of important business functionality. Document how you have verified your implemented change. Document the results of your acceptance test that test your feature from your change request.

6.1 Manual Verification

One way of testing whether our code works as intended, is by simply running the program and trying out the feature. Humans are, however, quite prone to errors and usually won't do the exact same thing twice, and therefore, this is a bad idea to have as your only test. It is, however, while developing a very good way of ensuring that you haven't catastrophically broken the program to the point of it not compiling.

6.2 Unit Testing

A much more consistent type of test is the unit test, testing a small section of your program with a tiny mockup of a real situation.

6.3 BDD Testing

Finally, we have BDD testing or Behavior-Driven Development, which is a method focusing on communication and collaboration between developers and non-technical people such as project managers, project owners etc.

By using BDD, we define the user stories that we previously defined using code, which will then perform the tests for us. By mapping these user stories to tests, we ensure that no misunderstanding between project manager, developer and tester can occur, or at least that it will be very hard for misunderstandings to occur.

These BDD tests are defined in the "Given-When-Then" format, meaning that we define the initial scenario, then we describe a specific event, usually triggering an action in the program and then we define what we expect as a result.

7 Continuous Integration

7.1 What is Continuous Integration?

Continuous Integration is the concept of continuously testing code and performing builds, artifact pushes and more, when code is pushed to relevant git branches or when pull requests are made. These things should be completely automatic and run without human input, notifying a developer of issues with their code if any are encountered.

7.2 Implementation

In our project, we make use of a pretty simple pipeline, simply called maven.yml, which is a pipeline performing a build test. Firstly, in the **on** key, the fact that the pipeline should run whenever pull requests to the develop branch are made. Next, the **jobs** key defines what should happen whenever a pull request to the develop branch is created. In this case, it sets up Java version 11 and Maven, and performs a Maven build. Should this build fail, GitHub is setup to disallow merging before the issues are fixed.

Additionally, to ensure that nothing is pushed to the main branch, circumventing the build check that is put in place, the repository has settings disallowing pushes to the main branch as well as pull requests to the main branch from anything other than the develop branch. This means that at least theoretically, no bad code should get through to the expectedly stable main branch.

```
name: Java CI with Maven

on:
  pull_request:
    branches: ["develop"]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 11
        uses: actions/setup-java@v3
        with:
          java-version: "11"
          distribution: "temurin"
          cache: maven
      - name: Build with Maven
        env:
          SECRET_USER: ${{secrets.SECRET_USER}}
          SECRET_TOKEN: ${{secrets.SECRET_TOKEN}}
        run: mvn -B package --batch-mode -DskipTests -s settings.xml --file pom.xml
```

.github/workflows/maven.yml

7.3 Effectively using git in our project

Collaboration during the project, of course, made use of git. Specifically, each group member managed their own refactoring branches to ensure minimal conflicts and code deletion. The use of Continuous Integration allowed for smooth development, in that, if issues arose, the build test would notify group members immediately, so that they could resolve their issues together in a pull request, separately from the rest of the code.

8 Conclusion

Overall, I think the project went well and taught me a lot about how much work actually goes into a refactor. I was surprised to realize that I didn't have enough time to do everything that I wanted with the software, but that experience was valuable in and of itself, as I will be more mindful of that in the future.

Working together as a group, overall was also very succesful. The initial stages of the project had everyone work together to produce the necessary reports and CI/CD pipelines so we could work together as effectively as possible. I did however, run into an issue with the Featureous software. I realized much too late, that the Featureous report that I had generated showed my feature as correlating with my feature, which ended up forcing me to cut my impact analysis short and made the final analysis table somewhat lackluster.

8.1 Merging with the baseline

Merging of the baseline version was performed using Git version control, which was done by developing on my own separate feature branch. Then, when I felt the feature was stable, I would rebase my branch on the develop branch before ultimately merging my feature branch into develop. Afterwards, I verified using CI/CD as well as manual verification, that everything was still working as intented.

This workflow allowed me to merge my feature into the baseline branch without any issues.

8.2 Testing

Testing was a very important aspect of this project to ensure stability and functionality post-refactor. Therefore, multiple ways of testing was employed.

Unit Testing: I made use of unit tests to test single parts of the software, ensuring that nothing broke while refactoring my feature.

User Acceptance Testing: I made use of User Acceptance tests in accordance with BDD to test that the software could perform the feature in the exact way that the user had intended prior to the refactor.

Manual Testing: I made use of period manual tests in the form of starting the software and going through the motions of loading in an image, to verify that nothing ctritical in the feature had broken while refactoring.

The system was tested thoroughly using unit tests and behaviour-driven development ensuring that refactoring of my feature was fully functional, as well as ensuring that the refactor didn't break any other team members' features.

8.3 Scope Changes

I ended up spending way too much time on files that in the end weren't really relevant to my feature, such as the LoadFileAction class, which resulted in me having to cut parts of my planned refactor short, focusing on other aspects of the project after refactoring only a single class.

8.4 Avoiding Problems in the Future

Also, in the future, I need to be more careful that the report that I am basing my impact analysis off of has been produced correclty, so that I can avoid spending unnecessary time on an analysis that has way too much information compared to what is actually needed.

9 Source Code

Source Repository: https://github.com/Autowinto/JHotDraw **Feature Branch:** https://github.com/Auto
magnus

References

Kerievsky, Joshua. *Refactoring to Patterns*. Addison-Wesley, 2005.