Opening various image formats

Software Maintance Autumn 2023

Magnus Christian Larsen

February 27, 2024

GitHub Username: Autowinto
Student Mail: magla21@student.sdu.dk
Project: https://github.com/Autowinto/JHotDraw
Date: mm/dd/yyyy

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:

Äs a user, I want to be able to open images of the formats png, jpg, gif, pct, and text, so that I can work with many different formats without having to convert myself"

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	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.
3	AbstractSaveUnsavedChangesAction	Go to implementation	The AbstractSaveUnsavedChangesAction class was found by finding the implementation of the class that LoadFileAction extends.

Continued on next page

Table 1: (Continued)

3	AbstractApplicationAction	Go to implementation	The AbstractApplicationAction is extended by the OpenFileAction
3	AbstractViewAction	Go to implementation	The AbstractViewAction is extended by the AbstractSaveUnsavedChangesAction class.
3	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.
4	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.

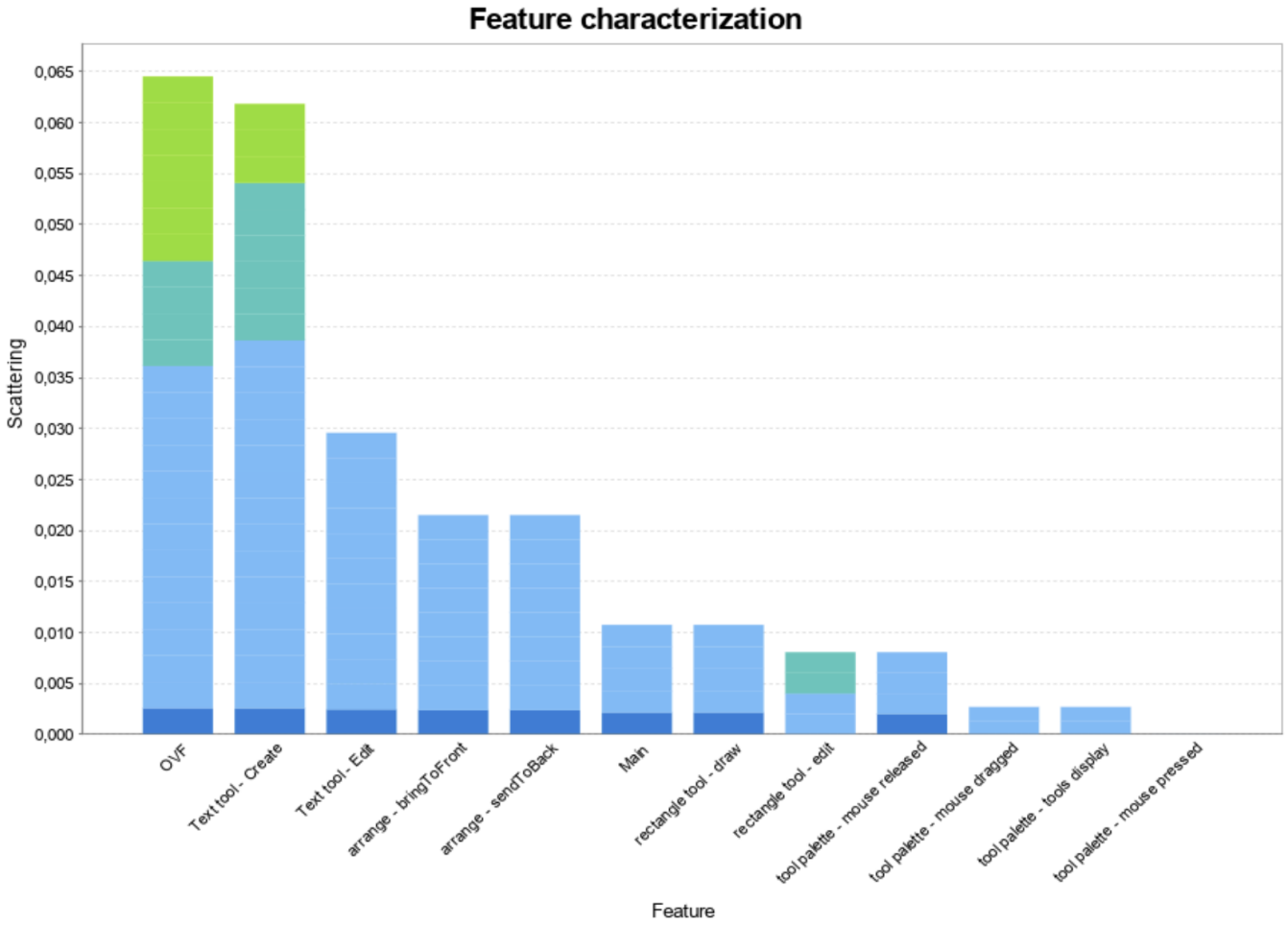
To perform this impact analysis, we're making use of the program featureous and FeatureEntryPoint annotations to annotate parts of the program that is planned to be refactored. By doing this, we can gain a further understanding of the interconnectivity of the program and see how the features overlap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

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

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

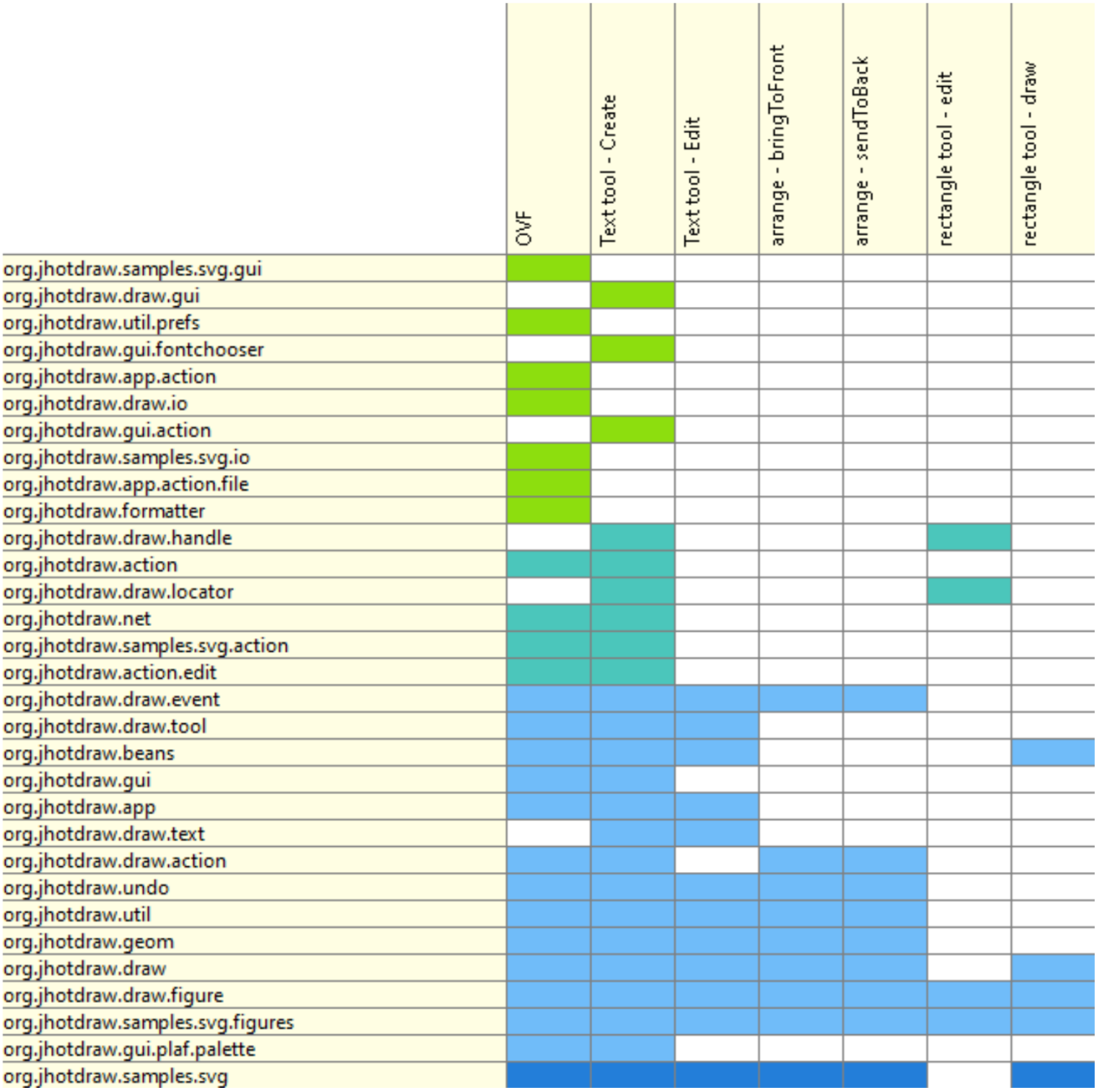
Based on these FeatureEntryPoints, the generated report can be used to analyze the impact of the feature.

3.1 Featureous Feature-Code Characterization



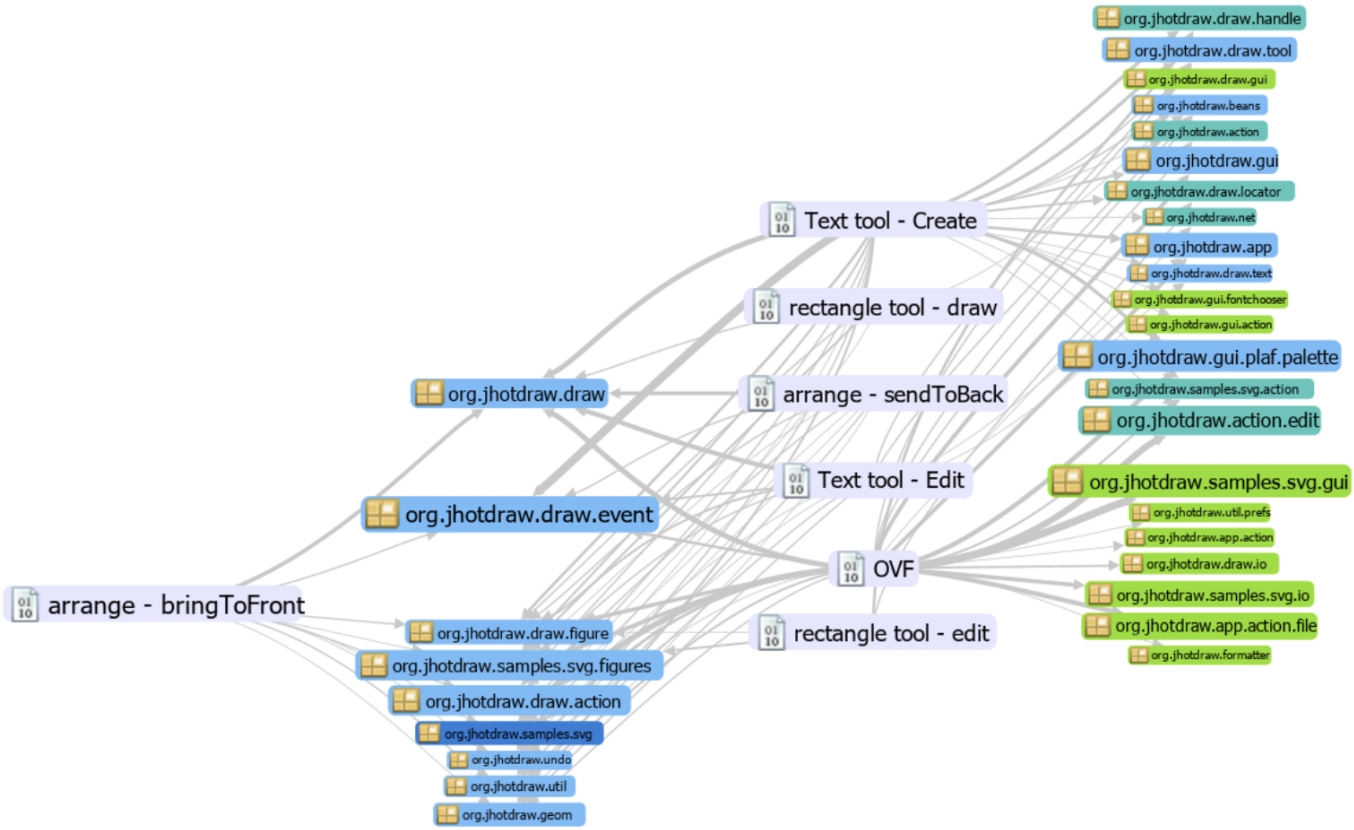
The above graph from featureous shows that my feature, here annotated as OVF, has a great deal of overlap with other features and is very scattered in the codebase. This means, that when refactoring, I need to be extra careful not to accidentally break other features, causing additional refactoring between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

3.2 Feature Correlation Grid



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.

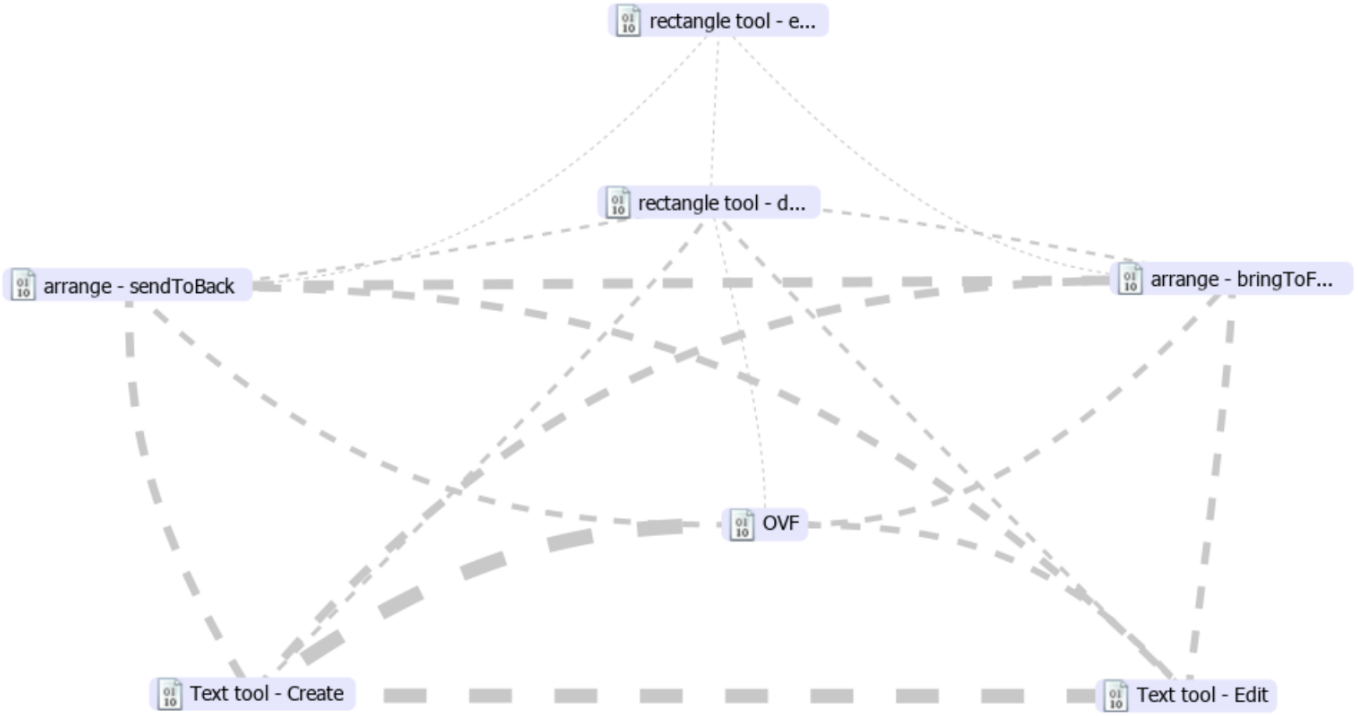
3.3 Feature Correlation Graph



Use Feature-code correlation graph and feature-code correlation grid to illustrate detailed investigations of the correspondences between source code units and related features to your change request.

Using table 2 list the packages and their number of classes that you visited after you located the concept. Write short comments explaining what you have learned about each package and how they contribute to your feature?

3.4 Feature Relations Characterization



Use Featureous Feature Relations Characterization view to relate your change request features to each other with respect to how they depend on the objects created by other features and how they exchange data by sharing these objects with one another.

3.5 Dependency Analysis

Show the features that my feature depends on

3.6 Impact Analysis

In the fourth column, mention if the class is related to the concept. Use one of the following terms:

- Use "Unchanged" if the class has no relation to the concept but you have visited it.
- Use "Propagating" if you read the source code of the class and it guides you to the location of the concept, but you will not change it.
- Use "Changed" if the class will be changed.

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

Package name	# of classes	Tool used	Comments

Continued on next page

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

4 Refactoring Patterns and Code smells

There is so much fucking duplicated code. The methods are incredibly long Method names are fucked and have no proper meaning Long parameter lists Confusing naming of classes and functions.

There is no reason to be abbreviating variables to the point of them being single letters. It makes for confusing code, especially in longer methods.

Describe the code smell that triggered your refactoring, see Chapter 4 in [Ker05]. Describe what you plan to change by refactoring. Why do you think it's good to do the refactoring? Describe the strategy of the refactorings. Remember there is not one way to implement a pattern. Which of the refactorings from [Ker05] did you apply and what was the reasoning behind it?

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.

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.

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.

```yaml
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

Explain your experience with creating the final baseline of JHotDraw. How did you manage to merge in your changes to the baseline ? How was the system tested? Describe your reflection on what went well and what went not so well. Did you have to cut the scope of your change request and did you have to put some issues back into the backlog. What can be done to avoid future problems, i.e. what have you learned from the iteration.

9 Source Code

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