Software Maintenance Autumn 2023

Frederik Busch

Januar 4, 2024

Student Mail: frbus21@student.sdu.dk
Course ID: T510001101
Student Exam Number: 182160123

Contents

1 Change Request

For this Software Maintenance report document, i have chosen to work with the feature called *Tool Palette.* The refactoring of the code will be done in a group consisting of 5 students total, including myself. We have each choosen af feacture to reactore doing the course of this project.

The infomation we have gotten on the different features are only a short descriptive text, with the name of the feature. For my chosen feature the text is the following: *Tool Palette - Display, Drag and Drop.* With this feature name i can with some analysis and implementation of the given code I can figure out what i have to reactore within my feature. As I am working with the feacture called *Tool Palette,* I will assume the whole of the tool palette is within my feacture. The *Tool Palette* after inspection looks to contain tool sections, where it has different tools that one can select within these sections.

As part of the project we have made individual User Stories for our chosen feature. My User Story are the following:

Drag and Drop

The *Drag/Drop* user story outlines a feature, that is designed to enable users of the program to customize their workspace within the program itself. It allows the user to drag and drop different sections of the toolbar, to a location of their choosing. By allowing the user to customize their workspace, it can impove their work efficiency, but have their most used tools and options within easy reach.

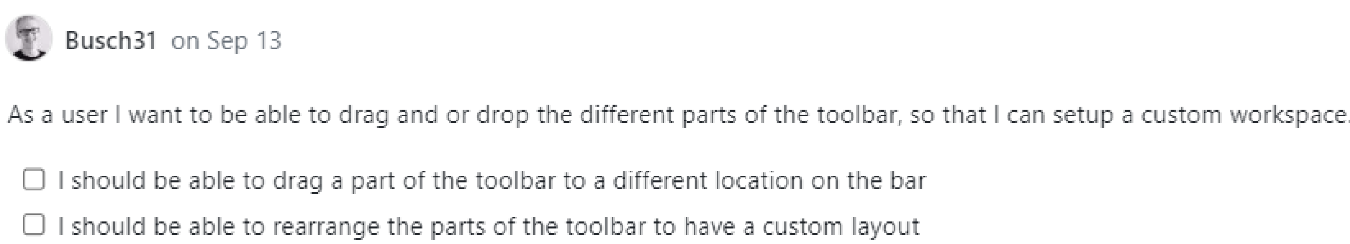

Figure 1: User Story for Drag and Drop

Display

The *Display* user story outlines a feature, that is designed to enable the users to show or hide different sections of the toolbar to their liking. Thereby allowing the users to hide or show only the tool section, that are relevant to their current task. It will also give the user less clutter on their screen doing their work.

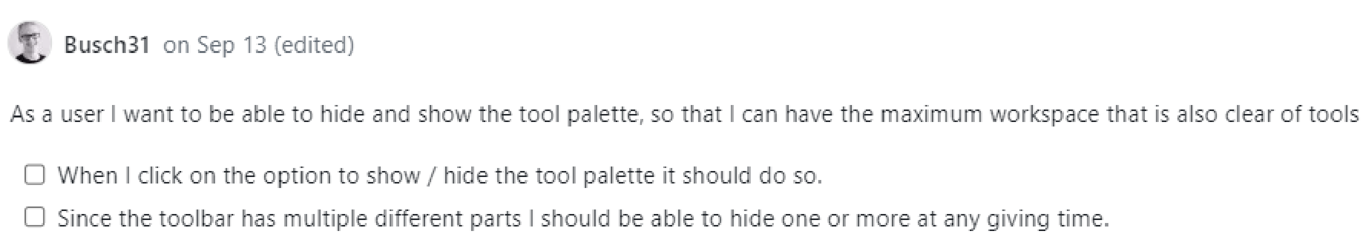

Figure 2: User Story for Display

To successfully complete the refactoring, the following steps should be undertaken by us as a group:

- Learn the feature scope of our different features within the codebase by doing a concept location to identify the relevant classes and tools.
- Evaluate the estimated impact of the refactoring on each developers features to anticipate any potential overlaps or conflicts our different features might have or could have.
- Understand the sections of code that require refactoring by identifying it with code smells.
- Carry out the refactoring while trying to minimize any unintended cascading changes that could happen with refactoring.
- Verify the changes after refactoring to ensure they achieve the desired outcome and that the primary function of the code is still maintained.

Besides having to do this refactoring, we as a group also have to setup continuous integration, thereby ensuring that any code is tested and verified before it is merged into the main branch.

2 Concept Location

2.1 Methodology

The location of the classes that I identified was based on the following tools, which I used to locate the different classes and the different methods that I found was relevant for the feature I had chosen *Tools Palette*.

- Different search methods such as:
  - Find all references.
  - Go to definition.
  - Quick search.
  - Global search.
  - Keyword search
- Tree scaling both up and down with Extension reference.
- Removing code to see what functionality it would affect, thereby better understand what the different pieces of code did what and affects.

2.2 Table Content Overview

The table below provides an easy overview of the different tools and processes I used to locate the different classes that I found relevant for my chosen feature.

#	Domain classes	Tools used	Comments
1	*AbstractToolbar*	Quick Search Find all references Code removal	I started by looking at the different abstract classes for the whole project. Here I found the *AbstractToolbar* class which looked like the right abstract class I was looking for when my features name is *Tool Palette.* I then tested with *code removal* to see what it would impact in the toolbar, but I just not see any changes to the behavior of the program itself when running, so I started looking at what the *AbstractToolbar* was extended from.
2	*JDisclosureToolbar*	Code removal Extension reference Go to definition	When I look at what *AbstractToolbar* was extended from, I found the abstract class named *JDisclosureToolbar*, I again tried code removal, this time giving my first result. The abstract class *JDisclosureToolbar* is responsible for the *show/hide* feature of the *tool palette,* which I needed for my user story *Display.*
3	*ToolsToolbar*	Code removal Extension reference	I then went back down the reference tree to see where in what class it would end. I ended up in the class *ToolsToolbar*. Here again with *code removal* I tried to see what the class was responsible for. I found that it was not the full toolbar as my first thought had been, but it was only a part of the whole *tool palette.*
4	*PaletteToolbarUI*	Code removal Keyword Search	After I hit a dead end with the *Extension reference* tool method, I tried to do a *global search* on different keywords such as *Tool, UI, Palette, Bar, ToolBar* and other keywords that could be assimilated with my feature. With this tool method I found the *PaletteToolbarUI* class which contained handlers. I tried to remove some of these handlers to see what it would affect.

Table 1: Overview of Domain Classes and Tools Used

3 Impact Analysis

3.1 Brief Introduction

The impact analysis is used to understand the implications of changing a specific feature within the *JHotDraw* project. The project itself will receive the different feature changes at different times, as the development team, consisting of 5 members, they are all working on different features of the application at their own pace, and some members might be further ahead than others at any given time. After inserting the feature entry points into the *JHotDraw* project, I was able to get an output of different relevant figures: *Feature-code Characterization, Feature-code Correlation Grid,* and *Feature-Package Correlation Graph.*

The feature entry points I used in this project are the following:

- *Tools-display*
- *Drag-drop*
  - *Pressed*
  - *Dragged*
  - *Released*

The Tools-display, as stated in the Concept Location chapter, is a class that references the *JDisclosureToolbar* class. The *Tools-display* is a handler callback that has been added to a button; when pressed, it will change the visibility of the chosen toolbar section from visible to hidden or vice versa.

The *Drag-drop*, which consists of *Pressed*, *Dragged*, and *Released*, are handlers that are connected with the *PaletteToolbarUI*. They are activated in the following order:

- *Pressed*: when a tool is pressed on with the click of a mouse.
- *Dragged*: when a tool has been pressed and the mouse moves while the button is still being held down by the user.
- *Released*: when the user releases the pressed mouse button again.

3.2 Featureous Feature-code Characterization

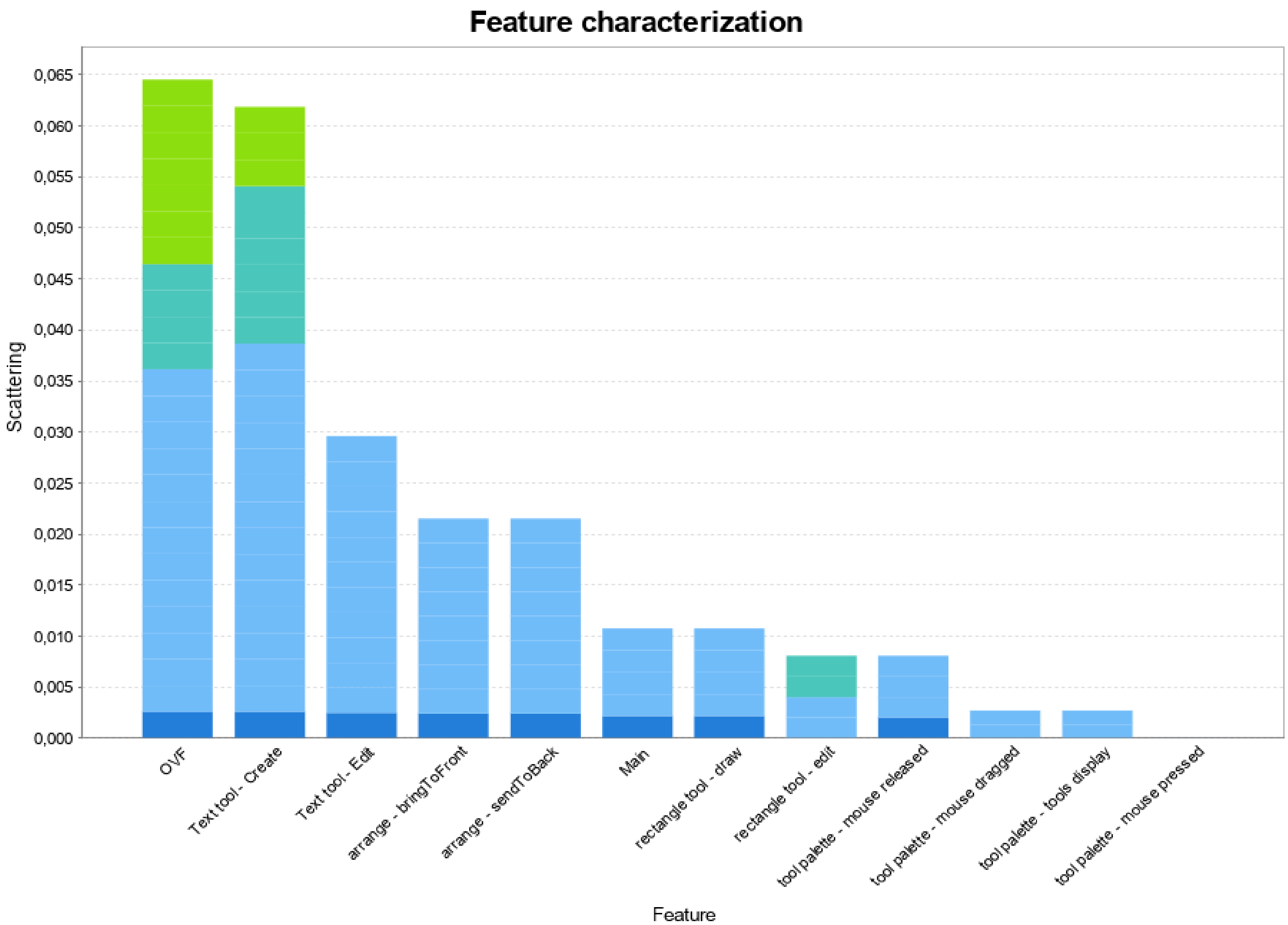

Figure 3: Featureous Feature-Code Characterization

As we can see, the units do contain alot of inter-group units, this can if one is not careful, end in entanglement with the other units that other developers are using in their part of the project, but it does offer us some insights for future refactorings.

3.3 Featureous Feature Relations Characterization

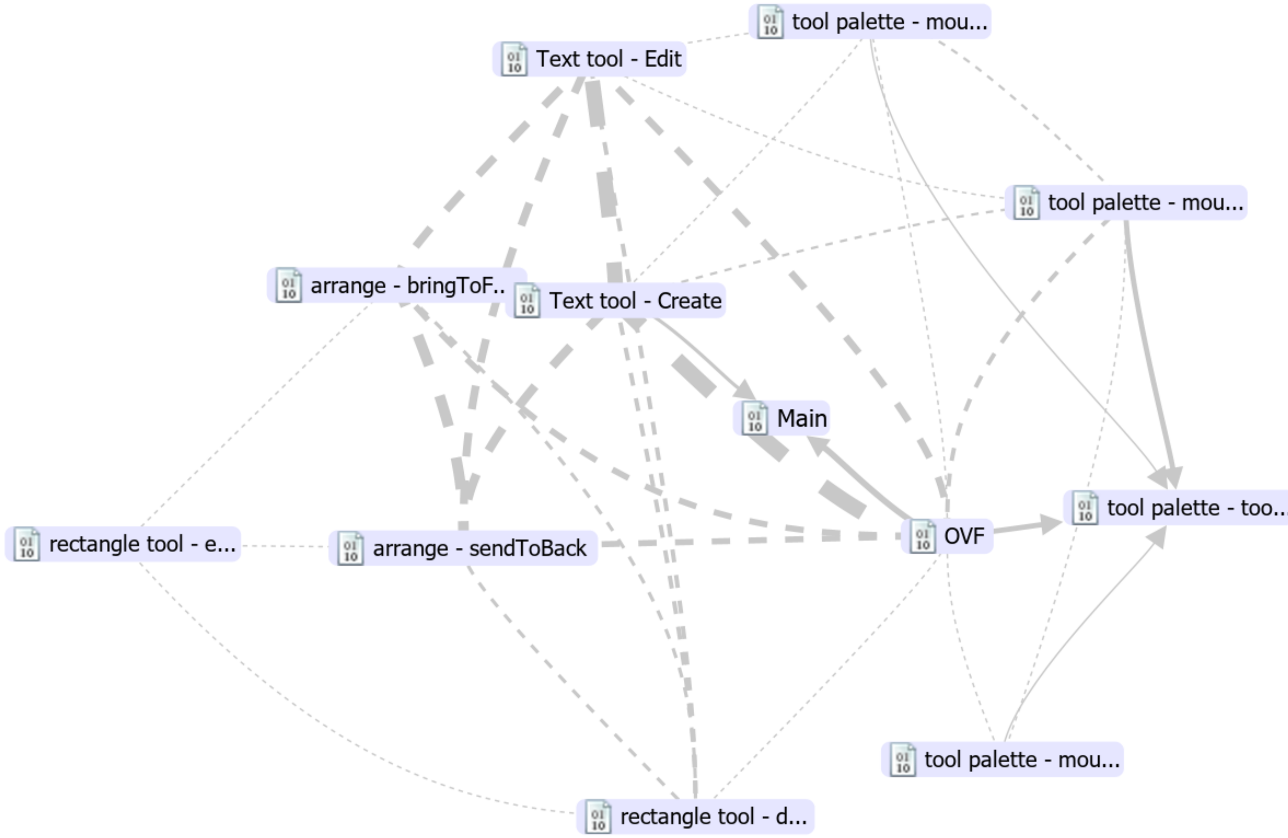

Figure 4: Featureous Feature Relations Characterization

As one can see, the connections which the entry points have made, does not contain strong connections with the other features that are also in this project. it does not look like they even engage in any consumer/producer connections.

3.4 Feature-code correlation graph and feature-code correlation grid

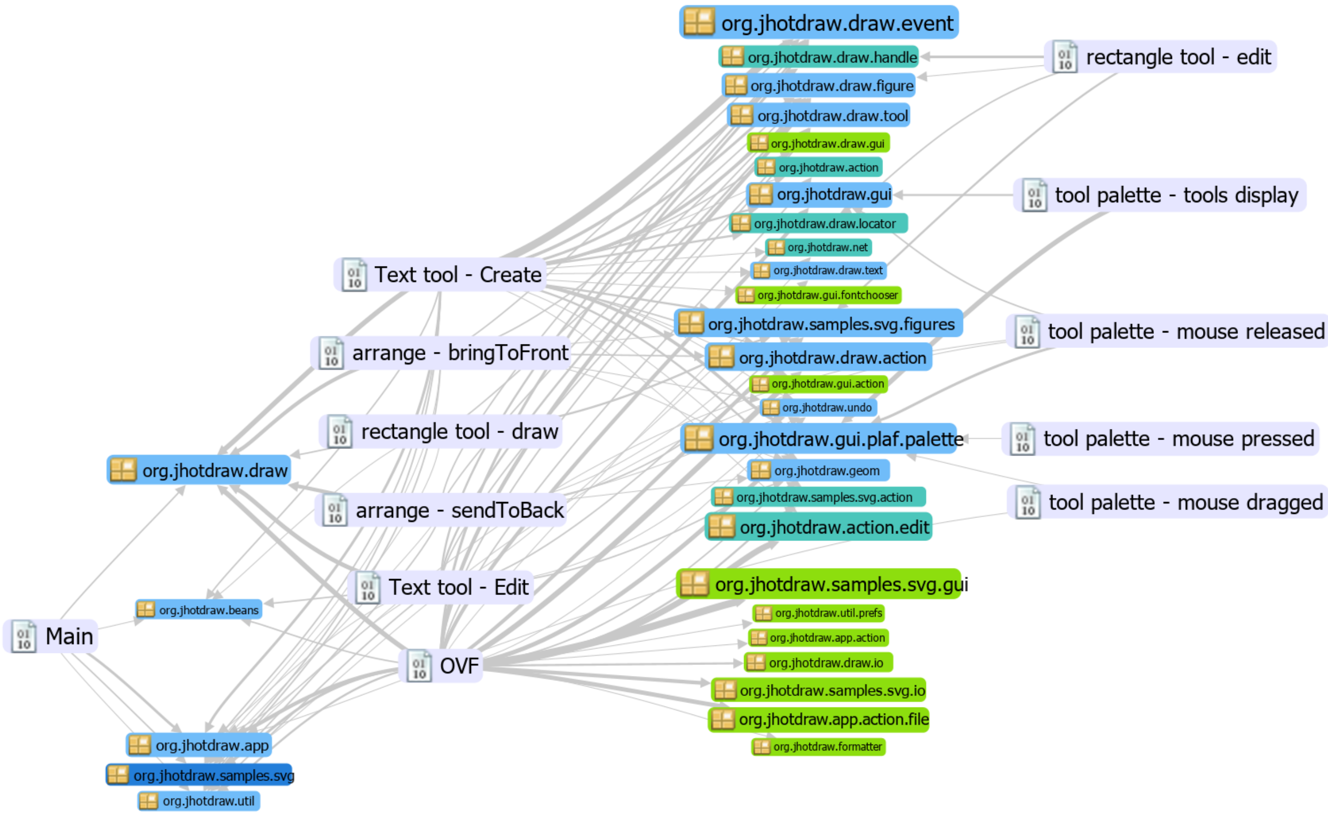

Figure 5: Feature-Package Correlation Graph

These tools can provide a deeper look into the connections between source code units and features, highlighting relationships and dependencies important for understanding the overall impact of the change request.

But however the *Feature-Package Correlation Graph* does not give the best overview of the connections between the features and the packages. What it can provide however, is a visual confirmation that there is no strong connections between the *Tool Palette* and the other entry points in the project.

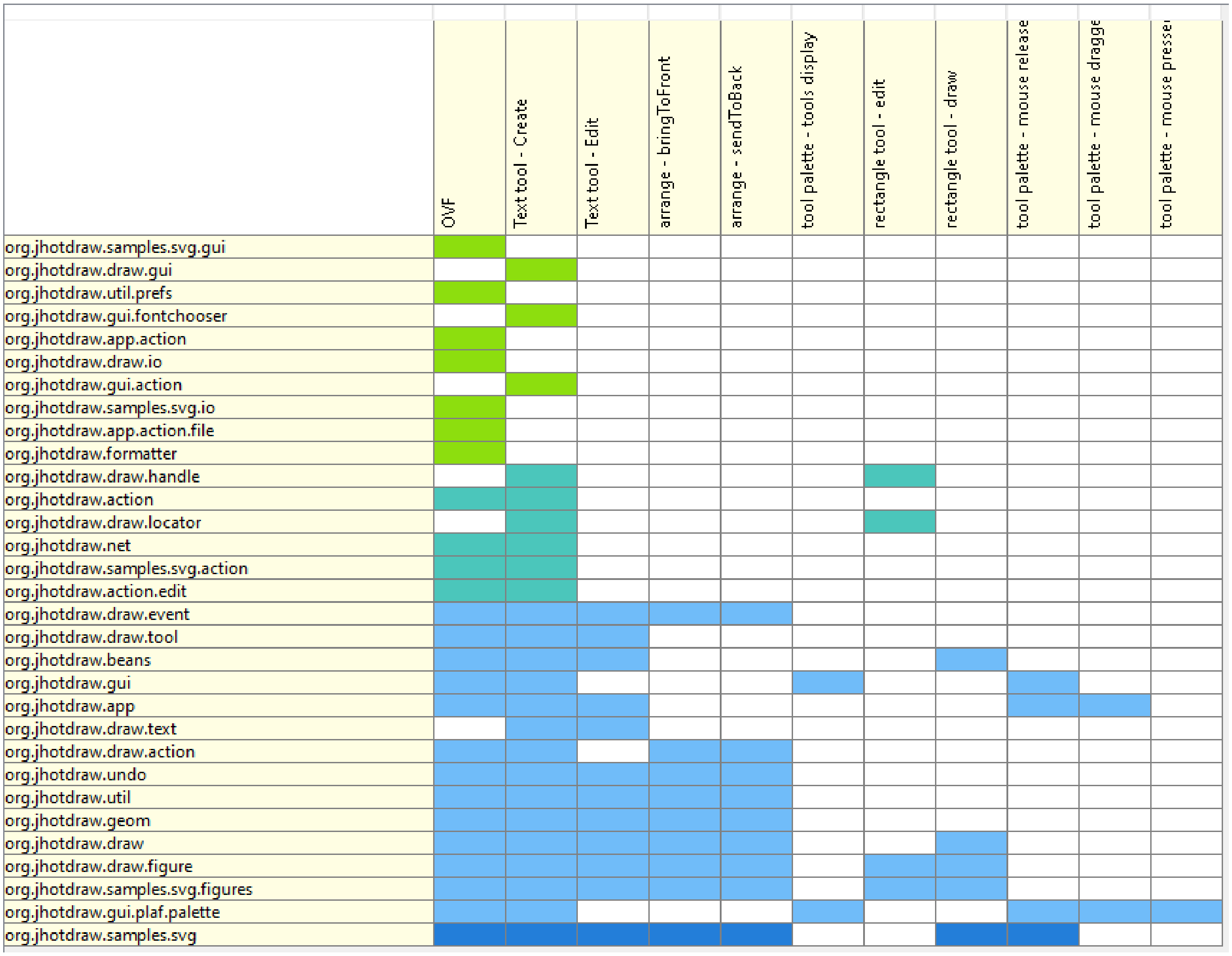

Figure 6: Feature-Code Correlation Grid

The analysis reveals only a small entanglement between my own features and those being used by other developers in the project.

The *drag-drop-released* feature *(mouse released, mouse dragged, mouse pressed)* only connected to the *OVF* package. The *tools-display* feature also only cross with intergroup packages and the *OVF* package. For my own features, they are not connected to any other features, meaning that they are not dependent on any other features in the project, and therefore can be refactored without any worries of breaking other features. Normally interlinked nature of features requires very careful mindsets of other developers' work before proceeding with development and refactoring of their parts of the project, but as stated before my own feature does not seem to requires the same deep of a careful mindset when it comes to its refactoring.

3.5 Table - Impact Analysis

Package name	**# of classes**	**Tool used**	**Comments**
.gui	76	Correlation grid	changed
.gui.plaf.pallete	38	Correlation grid	changed
.samples.svg	65	Correlation grid	unchanged
.draw.event	20	Correlation grid	unchanged
.app	39	Correlation grid	unchanged
.draw.gui	6	Correlation grid	unchanged

Table 2: Feature-code correlation data

The *PaletteToolbarUI* class displays a high level of independence, with no connections into *JHotDraw* outside of the *gui.plaf.palette* package. Its use is restricted to *PaletteToolbarBorder* within the same package and *JDisclosureToolbar*, related to another primary feature. Also the *JDisclosureToolbar* is connected only with the *gui.plaf.palette* package. given that no changes are made to its public methods, any refactoring is not likely to disturb other parts of the project.

4 Refactoring Patterns and Code smells

By using the plugin *SonarLint* in *IntelliJ* I was able to find several code smells within the scope of my chosen feature.

4.1 JDisclosureToolBar class

The *JDisclosureToolBar* class, part of the JHotDraw project, exhibits several code smells. These code smells potentially impact the maintainability, readability, and scalability of the code. below is the code that is referred to in the code smell analysis table that is also below.

```java
    private void initComponents() {
        GridBagConstraints gbc;
        AbstractButton btn;
        setLayout(new GridBagLayout());
        gbc = new GridBagConstraints();
        if (disclosureButton == null) {
            btn = new JButton();
            btn.setUI((PaletteButtonUI) PaletteButtonUI.createUI(btn));
            btn.setBorderPainted(false);
            btn.setIcon(new DisclosureIcon());
            btn.setOpaque(false);
            disclosureButton = (JButton) btn;
            disclosureButton.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, 1);
            disclosureButton.putClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY, 2);
            disclosureButton.addActionListener(new ActionListener() {
                @Override
                public void actionPerformed(ActionEvent e) {
                    int newState = ((Integer) disclosureButton.getClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY) + 1)
                            % (Integer) disclosureButton.getClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY);
                    setDisclosureState(newState);
                }
            });
        } else {
            btn = disclosureButton;
        }
        gbc.gridx = 0;
        gbc.insets = new Insets(0, 1, 0, 1);
        gbc.anchor = GridBagConstraints.SOUTHWEST;
        gbc.fill = GridBagConstraints.NONE;
        gbc.weighty = 1d;
        gbc.weightx = 1d;
        add(btn, gbc);
        putClientProperty(PaletteToolBarUI.TOOLBAR_INSETS_OVERRIDE_PROPERTY, new Insets(0, 0, 0, 0));
        putClientProperty(PaletteToolBarUI.TOOLBAR_ICON_PROPERTY, new EmptyIcon(10, 8));
    }
```

Figure 7: Orginal initComponents function

```java
public void setDisclosureState(int newValue) {
        int oldValue = getDisclosureState();
        disclosureButton.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, newValue);
        removeAll();
        JComponent c = getDisclosedComponent(newValue);
        GridBagConstraints gbc = new GridBagConstraints();
        if (c != null) {
            gbc = new GridBagConstraints();
            gbc.gridx = 1;
            gbc.weightx = 1d;
            gbc.weighty = 1d;
            gbc.fill = GridBagConstraints.BOTH;
            gbc.anchor = GridBagConstraints.WEST;
            add(c, gbc);
            gbc = new GridBagConstraints();
            gbc.gridx = 0;
            gbc.weightx = 0d;
            gbc.insets = new Insets(0, 1, 0, 1);
            gbc.weighty = 1d;
            gbc.fill = GridBagConstraints.NONE;
            gbc.anchor = GridBagConstraints.SOUTHWEST;
            add(disclosureButton, gbc);
        } else {
            gbc = new GridBagConstraints();
            gbc.gridx = 1;
            gbc.weightx = 1d;
            gbc.weighty = 1d;
            gbc.fill = GridBagConstraints.NONE;
            gbc.anchor = GridBagConstraints.SOUTHWEST;
            gbc.insets = new Insets(0, 1, 0, 1);
            add(disclosureButton, gbc);
        }
        invalidate();
        Container parent = getParent();
        while (parent.getParent() != null && !parent.getParent().isValid()) {
            parent = parent.getParent();
        }
        parent.validate();
        repaint();
        firePropertyChange(DISCLOSURE_STATE_PROPERTY, oldValue, newValue);
    }
```

Figure 8: Orginal setDisclosureState function

Method Name	**Description**	**Recommendation**
setDisclosureState	*setDisclosureState* is lengthy and handles multiple tasks, affecting readability and maintainability.	Break down into smaller, focused methods.
setDisclosureState	Repeated setup of *GridBagConstraints* in multiple methods.	Abstract common setup into a separate method or utility class.
initComponents	*initComponents* mixes UI element creation with layout management.	Separate UI creation from layout management.
initComponents	Unnecessary conditional logic in *initComponents*.	Review the necessity and simplify if possible.

Table 3: Code Smell Analysis for JDisclosureToolBar Class

4.2 PaletteToolbarUI class

The *PaletteToolbarUI* class in JHotDraw, designed for managing toolbars in a specific UI context, presents various code smells. These issues potentially affect the code's maintainability, readability, and scalability. The subsequent analysis and illustrations focus on specific methods within this class where these code smells are evident. below is the code that is referred to in the code smell analysis table that is also below.

```
protected void floatAt(Point position, Point origin) {
        if (toolBar.isFloatable() == true) {
            try {
                Point offset = dragWindow.getOffset();
                if (offset == null) {
                    offset = position;
                    dragWindow.setOffset(offset);
                }
                Point global = new Point(origin.x + position.x,
                        origin.y + position.y);
                setFloatingLocation(global.x - offset.x,
                        global.y - offset.y);
                if (dockingSource != null) {
                    Point dockingPosition = dockingSource.getLocationOnScreen();
                    Point comparisonPoint = new Point(global.x - dockingPosition.x,
                            global.y - dockingPosition.y);
                    if (canDock(dockingSource, comparisonPoint)) {
                        setFloating(false, comparisonPoint);
                    } else {
                        setFloating(true, null);
                    }
                } else {
                    setFloating(true, null);
                }
                dragWindow.setOffset(null);
            } catch (IllegalComponentStateException e) {
                // allowed empty
            }
        }
    }
```

Figure 9: Orginal floatAt function

```java
public void setFloating(boolean b, Point p) {
        if (toolBar.isFloatable() == true) {
            if (dragWindow != null) {
                dragWindow.setVisible(false);
            }
            this.floating = b;
            if (b && IS_FLOATING_ALLOWED) {
                if (dockingSource == null) {
                    dockingSource = toolBar.getParent();
                    dockingSource.remove(toolBar);
                }
                constraintBeforeFloating = calculateConstraint();
                if (propertyListener != null) {
                    UIManager.addPropertyChangeListener(propertyListener);
                }
                if (floatingToolBar == null) {
                    floatingToolBar = createFloatingWindow(toolBar);
                }
                floatingToolBar.getContentPane().add(toolBar, BorderLayout.CENTER);
                if (floatingToolBar instanceof Window) {
                    ((Window) floatingToolBar).pack();
                }
                if (floatingToolBar instanceof Window) {
                    ((Window) floatingToolBar).setLocation(floatingX, floatingY);
                }
                if (floatingToolBar instanceof Window) {
                    ((Window) floatingToolBar).setVisible(true);
                }
            } else {
                if (floatingToolBar == null) {
                    floatingToolBar = createFloatingWindow(toolBar);
                }
                if (floatingToolBar instanceof Window) {
                    ((Window) floatingToolBar).setVisible(false);
                }
                floatingToolBar.getContentPane().remove(toolBar);
                Integer constraint = getDockingConstraint(dockingSource,
                        p);
                if (constraint == null) {
                    constraint = 0;
                }
                int orientation = mapConstraintToOrientation(constraint);
                setOrientation(orientation);
                if (dockingSource == null) {
                    dockingSource = toolBar.getParent();
                }
                if (propertyListener != null) {
                    UIManager.removePropertyChangeListener(propertyListener);
                }
                dockingSource.add(toolBar, constraint.intValue());
            }
            dockingSource.invalidate();
            Container dockingSourceParent = dockingSource.getParent();
            if (dockingSourceParent != null) {
                dockingSourceParent.validate();
            }
            dockingSource.repaint();
        }
    }
```

Figure 10: Orginal setFloating function

```java
protected void dragTo(Point position, Point origin) {
        if (toolBar.isFloatable() == true) {
            try {
                if (dragWindow == null) {
                    dragWindow = createDragWindow(toolBar);
                }
                Point offset = dragWindow.getOffset();
                if (offset == null) {
                    //Dimension size = toolBar.getPreferredSize();
                    Dimension size = toolBar.getSize();
                    offset = new Point(size.width / 2, size.height / 2);
                    dragWindow.setOffset(offset);
                }
                Point global = new Point(origin.x + position.x,
                        origin.y + position.y);
                Point dragPoint = new Point(global.x - offset.x,
                        global.y - offset.y);
                if (dockingSource == null) {
                    dockingSource = toolBar.getParent();
                }
                constraintBeforeFloating = calculateConstraint();
                Point dockingPosition = dockingSource.getLocationOnScreen();
                Point comparisonPoint = new Point(global.x - dockingPosition.x,
                        global.y - dockingPosition.y);
                if (canDock(dockingSource, comparisonPoint)) {
                    dragWindow.setBackground(getDockingColor());
                    Object constraint = getDockingConstraint(dockingSource,
                            comparisonPoint);
                    int orientation = mapConstraintToOrientation(constraint);
                    dragWindow.setOrientation(orientation);
                    dragWindow.setBorderColor(dockingBorderColor);
                } else {
                    dragWindow.setBackground(getFloatingColor());
                    dragWindow.setBorderColor(floatingBorderColor);
                }
                dragWindow.setLocation(dragPoint.x, dragPoint.y);
                if (dragWindow.isVisible() == false) {
                    //Dimension size = toolBar.getPreferredSize();
                    Dimension size = toolBar.getSize();
                    dragWindow.setSize(size.width, size.height);
                    dragWindow.setVisible(true);
                }
            } catch (IllegalComponentStateException e) {
                // allowed empty
            }
        }
    }
```

Figure 11: Orginal dragTo function

Method Name	**Description**	**Recommendation**
setFloating	The method is doing too much, affecting readability and maintainability.	Break down into smaller, more focused methods.
setFloating	Multiple nested if-else statements increase complexity.	Refactor to reduce nested conditionals and simplify logic.
dragTo	The method is quite lengthy and performs many tasks.	Break down into smaller, more focused methods.
dragTo	Use of similar calculations and procedures as seen in other methods.	Abstract common functionality into a separate method or utility class.
floatAt	The method is overly long.	Break down into smaller, more focused methods.
floatAt	Contains logic that appears to be duplicated from ‘dragTo‘.	Abstract common functionality into a separate method or utility class.

Table 4: Code Smell Analysis for PaletteToolbarUI Class

5 Refactoring Implementation

5.1 JDisclosureToolBar class

5.1.1 initComponents

This method now gives a clearer and more modular approach than before the Refactoring. It initializes the layout of the toolbar and configures its components, including the disclosure button.

Improvements: The refactoring makes it more readable and maintainable by breaking down the process into more clear steps steps, thereby improving the overall structure of the code.

```java
    private void initComponents() {
        setLayout(new GridBagLayout());
        if (disclosureButton == null) {
            initializeDisclosureButton();
        }
        addDisclosureButtonToToolBar();
        configureToolBarProperties();
    }
```

Figure 12: Refactored initComponents Method

5.1.2 setDisclosureState

This method manages the disclosure state of the toolbar, updating its state and rearranging components based on the new state.

Improvements: The refactoring of this method has streamlined it to be more clear and efficient than before. By handling the state change and component rearrangement, it new enhance the toolbars adaptability state changes in the program. with these Improvements its more readable and maintainable than before, making it easier to understand and modify the toolbars behavior.

```java
    public void setDisclosureState(int newValue) {
        int oldValue = getDisclosureState();
        disclosureButton.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, newValue);
        removeAll();
        JComponent c = getDisclosedComponent(newValue);

        if (c != null) {
            addComponentWithConstraints(c, createGridBagConstraints(1, 1d, 1d, GridBagConstraints.BOTH,
GridBagConstraints.WEST));
            addComponentWithConstraints(disclosureButton, createButtonGridBagConstraints());
        } else {
            addComponentWithConstraints(disclosureButton, createButtonGridBagConstraints());
        }

        validateAncestor();
        repaint();
        firePropertyChange(DISCLOSURE_STATE_PROPERTY, oldValue, newValue);
    }
```

Figure 13: Refactored setDisclosureState Method

5.1.3 initializeDisclosureButton

The method is used to initializing the disclosure button, setting up its UI, and attaching an action listener for state changes, to monitor when they happen.

Improvements: Separating the initialization of the disclosure button into its own method, to enhance the single responsibility principle, making the code easier to understand and maintain by other developers.

```
private void initializeDisclosureButton() {
    JButton btn = createDisclosureButton();
    btn.addActionListener(createDisclosureButtonActionListener());
    disclosureButton = btn;
}
```

Figure 14: Refactored initializeDisclosureButton Method

5.1.4 createDisclosureButton

This method creates a JButton specifically configured as a disclosure button, setting its UI and properties.

Improvements: By encapsulating the creation of the disclosure button, the code becomes more modular and reusable, promoting better coding practices.

```
private JButton createDisclosureButton() {
    JButton btn = new JButton();
    btn.setUI((PaletteButtonUI) PaletteButtonUI.createUI(btn));
    btn.setBorderPainted(false);
    btn.setIcon(new DisclosureIcon());
    btn.setOpaque(false);
    btn.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, 1);
    btn.putClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY, 2);
    return btn;
}
```

Figure 15: Refactored createDisclosureButton Method

5.1.5 createDisclosureButtonActionListener

This method creates an ActionListener for the disclosure button, handling the action when the changing the disclosure state of the toolbar happens.

Improvements: By isolating the action listener creation, it will improve the clarity of the event handling and also simplify any modifications or extensions that might be needed in the future by other developers learning the system code.

```
private ActionListener createDisclosureButtonActionListener() {
      return new ActionListener() {
          @Override
          public void actionPerformed(ActionEvent e) {
              int currentState = getDisclosureState();
              int stateCount = getDisclosureStateCount();
              int newState = (currentState + 1) % stateCount;
              setDisclosureState(newState);
          }
      };
  }
```

Figure 16: Refactored createDisclosureButtonActionListener Method

5.1.6 addDisclosureButtonToToolBar

This method adds the disclosure button to the toolbar with appthe right GridBagConstraints, it will handle the positioning within the toolbar it self.

Improvements: By isolating the process of adding the button to the toolbar, the method will be more readable and make it easier to make any adjustments to the buttons positioning.

```
  private void addDisclosureButtonToToolBar() {
        GridBagConstraints gbc = createGridBagConstraints(0, 1d, 1d, GridBagConstraints.NONE,
GridBagConstraints.SOUTHWEST);
        gbc.insets = new Insets(0, 1, 0, 1);
        add(disclosureButton, gbc);
    }
```

Figure 17: Refactored addDisclosureButtonToToolBar Method

5.1.7 configureToolBarProperties

this methods now sets up additional toolbar properties such as insets, icons and applying custom settings.

Improvements: This isolateds the method for additional set up aids in maintaining clean code and it will make it easier to modify toolbar properties without affecting other functionalities of the toolbar palette.

```
    private void configureToolBarProperties() {
        putClientProperty(PaletteToolBarUI.TOOLBAR_INSETS_OVERRIDE_PROPERTY, new Insets(0, 0, 0, 0));
        putClientProperty(PaletteToolBarUI.TOOLBAR_ICON_PROPERTY, new EmptyIcon(10, 8));
    }
```

Figure 18: Refactored configureToolBarProperties Method

5.1.8 validateAncestor

Validates the ancestor container of the toolbar, making its the right layout and rendering shown the user.

Improvements: This method provides an method to validate the container hierarchy, which will improving the dependability of the toolbars rendering process.

```
    private void validateAncestor() {
        Container parent = getParent();
        while (parent.getParent() != null && !parent.getParent().isValid()) {
            parent = parent.getParent();
        }
        parent.validate();
    }
```

Figure 19: Refactored validateAncestor Method

5.2 PaletteToolBarUI class - setFloating, dragTo and floatAt

5.2.1 setFloating

This method is responsible for the floating state of the toolbar, it will choose whether it should be docked or floating based on the given parameters.

Improvements: The code is more readable and maintainable by breaking down the complex conditional logic into smaller methods with more clear responsibility, than from before the refactoring.

```
public void setFloating(boolean b, Point p) {
    if (!toolBar.isFloatable()) {
        return;
    }

    hideDragWindow();
    this.floating = b;

    if (b && IS_FLOATING_ALLOWED) {
        prepareFloating();
    } else {
        prepareDocking(p);
    }
}
```

Figure 20: Refactored setFloating Method

5.2.2 floatAt

The method adjust the toolbars position, which is based on the current drag operation that is happening. It will be considering its floatability and docking potential doing so.

Improvements: Optimize the toolbars positioning during dragging of a toolbar, this is done by centralizing position calculations, leading to an more maintainable code thats easier to read also.

```
public void floatAt(Point position, Point origin) {
    if (!toolBar.isFloatable()) {
        return;
    }

    try {
        Point global = calculateGlobalPosition(position, origin);
        Point comparisonPoint = calculateComparisonPoint(global);
        boolean shouldDock = canDock(dockingSource, comparisonPoint);

        setFloatingLocation(global.x, global.y); // This sets the location for floating
        setFloating(!shouldDock, shouldDock ? comparisonPoint : null); // This sets the floating state
    } catch (IllegalComponentStateException e) {
        // Handle exception if necessary
    }
}
```

Figure 21: Refactored floatAt Method

5.2.3 prepareFloating

This method together wit the *prepareDocking* method will separately handle the set up steps for floating and docking the toolbar.

Improvements: It now encapsulates the floating and docking actions, it also now have increased modularity and reducing complexity from than before the refactoring.

```
private void prepareFloating() {
    initializeDockingSource();
    constraintBeforeFloating = calculateConstraint();
    showFloatingWindow();
}
```

Figure 22: Refactored prepareFloating Method

5.2.4 initializeDockingSource

This method makes sure that the toolbars docking source is initialized before changing its floating state.

Improvements: By isolating the initialization of the docking source, it will be promoting code reuse and also simplifying the primary floating method at the same time.

```
private void initializeDockingSource() {
    if (dockingSource == null) {
        dockingSource = toolBar.getParent();
        dockingSource.remove(toolBar);
    }
}
```

Figure 23: Refactored initializeDockingSource Method

5.2.5 showFloatingWindow

this method together with the metod *updateFloatingWindowAppearance* will manage the visual presentation of the toolbar when it is in a floating state.

Improvements: By seperating the visual aspects of the old methods into new dedicated methods, will enhance the readability of UI changes and at the same time simplifies the overall floating logic.

```
private void showFloatingWindow() {
    createFloatingToolBar();
    updateFloatingWindowAppearance();
}
```

Figure 24: Refactored showFloatingWindow Method

5.2.6 dragTo

This methods is responsible for the toolbars dragging functionality, it will calculate and be setting the new position during a drag operation.

Improvements: By breaking down the dragging process into smaller steps, it will make the code and metod more readable and thereby the code easier to follow and maintain by other developers in the future.

```java
protected void dragTo(Point position, Point origin) {
    if (toolBar.isFloatable()) {
        initializeDragWindow();
        Point dragPoint = calculateDragPoint(position, origin);
        updateDockingSource();
        updateDragWindowAppearance(dragPoint, origin);
        setDragWindowLocationAndVisibility(dragPoint);
    }
}
```

Figure 25: Refactored dragTo Method

5.2.7 initializeDragWindow

Together with the methods *calculateDragPoint, getDragWindowOffset, updateDragWindowAppearance* and *setDragWindowLocationAndVisibility* they will together handle the initialization and appearance of the drag window, including its positioning and visibility.

Improvements: By make the code modular for the drag window functionality, it will enhance the structure and make it more maintainable, by allowing each aspect to be modified independently of eachother.

```java
private void initializeDragWindow() {
    if (dragWindow == null) {
        dragWindow = createDragWindow(toolBar);
    }
}
```

Figure 26: Refactored initializeDragWindow Method

5.3 Impact of refactoring

After having refactored the code for my feature *Tool Palette*, there have been no breaking of the code or unexpected issues with other parts of the project. Which is what was expected from the analysis of the code before the refactoring was performed. but there is still parts of the of code that could be refactored further, but these parts of the code were not within the scope of the refactoring for my feature, but it could be something that could be done in the future by other developers.

6 Verification

Testing is important in software development to ensure that written code new and refactored are both fucntional and stable. In this project there have been a focus on unit tests for important logic and Behavior-Driven Development (BDD) for user testing.

by doing unit testing and BDD together, it can ensure that both the functionality of the individual components and the overall user experience are done as intended by the developers with the end user in mind. by doing unit testing and BDD, it will help immensely with the software not only works as intended from a technical view point, but also meets end users expectations in a given scenario.

6.1 Unit Testing

6.1.1 JDisclosureToolBar Tests

Unit tests for the *JDisclosureToolBar* class are made to ensure its will be fucntional readable as intended. For the JDisclosureToolBar class, unit tests were developed to check its key functionalities to make sure its working as inteded. The focus for this unit tests was on the toolbars ability to update and retrieve state information correctly.

6.1.2 testSetDisclosureStateCount

This test is used to verify that the setDisclosureStateCount method updates the state count of the disclosure button correctly. The test involves setting a new state count and confirming that the change is done correctly.

```
@Test
void testSetDisclosureStateCount() {
    int newStateCount = 4;
    disclosureToolBar.setDisclosureStateCount(newStateCount);
    verify(disclosureButton).putClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY, newStateCount);
}
```

Figure 27: testtSetDisclosureStateCount Test

6.1.3 testGetDisclosureState

This test is used to ensure that the getDisclosureState method correctly gets the current state of the disclosure button. The test mocks the return value of the state and checks if the method returns this mocked value.

```
@Test
void testGetDisclosureStateCount() {
    int expectedStateCount = 3;
    when(disclosureButton.getClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY)).thenReturn(expectedStateCount);
    assertEquals(expectedStateCount, disclosureToolBar.getDisclosureStateCount());
}
```

Figure 28: testGetDisclosureStateCount Test

6.1.4 testGetDisclosureStateCount

This tests the getDisclosureStateCount method to check if it does return the correct count of disclosure states. The test does this by setting an expected state count, mocking the retrieval, and checking if the returned count matches the expected value.

```
@Test
void testGetDisclosureState() {
    int expectedState = 2;
    when(disclosureButton.getClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY)).thenReturn(expectedState);
    assertEquals(expectedState, disclosureToolBar.getDisclosureState());
}
```

Figure 29: testGetDisclosureState Test

6.1.5 PaletteToolBarUI Tests

The unit test for the *PaletteToolBarUI* class, are unit tests were implemented to assure that when the tool palette is floating and trying to dock, functionalities are working as inteded. These tests are important when it comes to verifying the behavior of the toolbar.

6.1.6 testSetFloatingTrue

This test checks the setFloating method when the toolbar is set to float. It cheks that when the floating state is set to true, the toolbars isFloating method shows this change of the state.

```
@Test
void testSetFloatingTrue() {
    Point p = new Point(100, 100);
    paletteToolBarUI.setFloating(true, p);
    assertTrue(paletteToolBarUI.isFloating());

}
```

Figure 30: testSetFloatingTrue Test

6.1.7 testSetFloatingFalse

This test checks the scenario where the toolbar is set to not be floating. It well ensure that setting the floating state to false is correctly shown in the toolbars state.

```
@Test
void testSetFloatingFalse() {
    Point p = new Point(100, 100);
    paletteToolBarUI.setFloating(false, p);
    assertFalse(paletteToolBarUI.isFloating());

}
```

Figure 31: testSetFloatingFalse Test

6.1.8 testSetFloatingLocation

This test checks the setFloatingLocation method. It ckechs that the method correctly updates the floating coordinates of the toolbar.

```
@Test
void testSetFloatingLocation() {
    int x = 200, y = 300;
    paletteToolBarUI.setFloatingLocation(x, y);
    assertEquals(x, paletteToolBarUI.floatingX);
    assertEquals(y, paletteToolBarUI.floatingY);
}
```

Figure 32: testSetFloatingLocation Test

6.1.9 testFloatAt

This test checks the floatAt method, which is responsible for updating the toolbars position when dragged. The test simulates the dragging action and checks the outcome is correct.

```
@Test
void testFloatAt() {
    Point position = new Point(50, 50);
    Point origin = new Point(20, 30);

    paletteToolBarUI.floatAt(position, origin);
}
```

Figure 33: testFloatAt Test

6.2 Behavior-Driven Testing (BDD)

The BDD focuses on application behavior from the user perspective. I used JGiven to make the BDDs automatically generated.

6.2.1 ToolPaletteTestDisplay Scenario

This scenario, *scenario_The_user_wants_to_hide_a_palette_on_the_tool_bar*, tries to mimic when a user wants to hide a toolbar palette from the toolbar itself. The scenario code is build by 4 files *GivenToolPaletteDisplay, ThenOutcomeDisplay, WhenUserInteractsDisplay* and *ToolPaletteTestDisplay*.

The *ToolPaletteTestDisplay* file is the main file that runs the scenario, and the other files are extended by it. below is the output when the scenario is run.

```java
public class ToolPaletteTestDisplay extends ScenarioTest<GivenToolPaletteDisplay, WhenUserInteractsDisplay,
ThenOutcomeDisplay> {

    @Test
    void scenario_The_user_wants_to_hide_a_palette_on_the_tool_bar() {
        given().a_tool_palette_is_shown();
        when().the_user_wants_to_hide_a_given_palette();
        then().the_user_clicks_the_hide_button_and_the_palette_is_hidden();
    }
}
```

Figure 34: BDD ToolPaletteTestDisplay

```
Test Class: org.jhotdraw.gui.BDD.Display.ToolPaletteTestDisplay

 Scenario The user wants to hide a palette on the tool bar

   Given a tool palette is shown
    When the user wants to hide a given palette
    Then the user clicks the hide button and the palette is hidden
```

Figure 35: BDD ToolPaletteTestDisplay output

6.2.2 ToolPaletteTestDragAndDrop Scenario

The *ToolPaletteTestDragAndDrop* scenario minic an user interaction with drag-and-drop functionality in the toolbar. The scenario code is build by 4 files *GivenToolPaletteDragAndDrop, ThenOutcomeDragAndDrop, WhenUserInteractsDragAndDrop* and *ToolPaletteTestDragAndDrop*.

The *ToolPaletteTestDragAndDrop* file is the main file that runs the scenario, and the other files are extended by it. below is the output when the scenario is run.

```java
class ToolPaletteTestDragAndDrop extends ScenarioTest<GivenToolPaletteDragAndDrop, WhenUserInteractsDragAndDrop,
ThenOutcomeDragAndDrop> {

    @Test
    void Scenario_Toolbar_can_be_dragged_from_theTool_palette_bar_by_the_user() {
        given().the_tool_palette_is_visible_and_can_be_interacted_with();
        when().the_user_wants_to_change_the_position_of_a_tool();
        then().the_toolbar_is_placed_according_to_user_preference();
    }
}
```

Figure 36: BDD ToolPaletteTestDragAndDrop

```
Test Class: org.jhotdraw.gui.BDD.DragAndDrop.ToolPaletteTestDragAndDrop

 Scenario Toolbar can be dragged from theTool palette bar by the user

   Given the tool palette is visible and can be interacted with
    When the user wants to change the position of a tool
    Then the toolbar is placed according to user preference
```

Figure 37: BDD ToolPaletteTestDragAndDrop output

7 Continuous Integration

7.1 Understanding Continuous Integration

Continuous Integration (CI) is a development practice where developers frequently integrate their code into a shared repository. Each integration is then automatically checked by an automated build, this will allow teams to detect problems early and thereby use less time trying to bug fix and error locate doing development.

In our context, CI gives os the opportunity to do rapid development of our software by ensuring that new code changes do not break existing functionalities of the program when we push refactored code to the development branch. CI works by doing a series of automated tests and builds, this is triggered when developers push code to the repository ensuring code quality and stability when multiple developers are working simultaneously on the program.

7.2 Continuous Integration Implementation

For our project, CI is implemented using GitHub Actions, which is a tool on github that allows developers to do CI/CD which will automates our build and test processes. Our pipeline, defined in the .github/workflows directory, specifies the actions to be taken when a pull request is made to the develop branch.

The pipeline is composed of several steps, each step is a job that is run on a virtual machine hosted by GitHub.

Checkout: Retrieves the code from the repository.

Set up JDK 11: Configures the Java Development Kit, version 11, which is necessary for building and testing Java applications.

Build with Maven: Compiles the project, skipping the tests for speed in this initial phase, using Maven - a software project management tool.

```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

```

Figure 38: Continuous Integration yml file

By using Integrating CI in our development process it gives us several benefits such as.

Early Bug Detection: Bugs and integration issues are detected and fixed early in the development cycle, since we will be alerted sooner.

Rapid Feedback: Developers will receive instant feedback on their code, speeding up the development and review process immensely.

Consistent Code Quality: Automated tests ensure code quality is maintained by all developers, reducing the chance of introducing new bugs or erros unknowingly.

Enhanced Collaboration: Promotes collaboration among team members, as CI will ensure that the code in the repository is always in a stable state and running correctly.

7.3 Version Control with Git

Our project uses Git, which is a distributed version control system, to manage any changes to the codebase that developers on the project might make. Git allows multiple developers to work on the same codebase simultaneously, without overwriting each other's changes.

In our project we started by baching out from main to a develop branch, this was done to ensure that the main branch always contains a stable version of the codebase. From the develop branch we could then branch out into our own feature branches, where we could work on our own features without affecting the develop branch. When we were done with our feature, we could then merge our feature branch into the develop branch, where it would be reviewed before being merged into the main branch by the ind of the project, when the delevop branch had a stable running build.

8 Conclusion

The path to creating a baseline of JHotDraw involved integrating the developed features, bug fixes, and enhancements into the main codebase, this was done by using Git and working from a develop branch of the main repository. The process needed careful planning and execution to ensure compatibility and maintain code quality, and we had to redo some parts of the process due to unforeseen issues like the feacture traces, so this needed to be down correctly before proceeding with the rest of the project. In the beginning of the project we had one understanding of how the feacture trace should be done, but we later come to understand that our approach was wrong and we had to redo the feacture trace again, this we found out was important, since the *Impact Analysis* would lay the basis for each persons feature refactoring of their given feature.

8.1 Merging baseline

The merging of the baseline was done by using Git, and the process was done by first creating a develop branch from the main branch, and then creating a feature branch for each person in the group, this was done to ensure that each person could work on their own feature without affecting the main codebase. When a feature was done, it was merged into the develop branch, where it was reviewed by a group member. This process was repeated for each feature, bug fix, and enhancement made.

8.2 System Testing

Testing played an important role to ensure the stability and functionality of the system when doing the refactoring of ones feature.

Unit Testing: We implemented unit tests to check the methods which was refactored to ensuring each worked as intended.

Integration Testing: After merging features, integration tests were conducted to check the seamless interaction between different parts of the application which had been refactored.

User Acceptance Testing: User acceptance tests were carried to check that the system would met the end user expectations of a refactored feature.

8.3 Reflections

Reflecting on the final baseline creation, there were both things that went well and things that did not went as planned.

What Went Well:

The team worked together as intended, with clear communication, regular updates and helping eachother when needed, contributing to a smooth process.

Effective Use of CI assisted in maintaining code quality and accelerating the development cycle.

What went not so well:

The feature trace was not done correctly in the beginning, which led to a lot of extra work and time spent redoing it and thereby being a bit behind, which put pressure on our timeline and planning.

Time constraints did become a problem somtimes and it affected the the scope of the feature refactoring at some points, leading to a lesser scope for my own feature.

8.4 Scope Adjustments

During the development process, I encountered situations where I had to adjust the scope of my feature. I had to priorities what methods I would refactor due to time constraints, some less critical methods were choosen not to be refactored to ensure that the more critical methods were refactored correctly first. In some cases, features were scaled down to their essential functionalities to meet deadlines while maintaining quality.

8.5 Lessons Learned

Working on this project has been a valuable learning experience for me, and I have gained a lot of knowledge and skills that I can apply in future projects when it comes to refactoring others code, features and programs. I have learned alot about version control using Git, and how to use it to manage a codebase with multiple developers, and how to use Git to merge features into a delevopment codebase to make a stable baseline, before merging into the main itself. I have also learned how to use CI to ensure code quality and stability, and how to use it to automate the build and test process, which is a very useful tool. I have also learned to better manage my time and how to prioritize tasks to ensure that the most critical tasks are done first, and how to adjust the scope of a feature, when actively working on it, to ensure that the most critical parts are done first.

All in all I am very satisfied with what i have learned doing the project, and I am looking forward to applying the knowledge and skills I have gained in future projects where I will need to do code refactoring again.

9 Source Code

Delevop Branch: https://github.com/Autowinto/JHotDraw

Feature Branch https://github.com/Autowinto/JHotDraw/tree/refactoring-busch

10 References

Kerievsky, J. (2005). Refactoring to Patterns. Addision-Wesley. Rajlich, V. (2013). Software Engineering: The current Practice, volume 38. New York: ACM.

Table of figures

List of Figures

Table of tables

List of Tables