

# Software Maintenance Autumn 2023

Frederik Busch

December 22, 2023

Student Mail: frbus21@student.sdu.dk  
Date: mm/dd/yyyy

# Contents

<b>1</b>	<b>Change Request</b>	<b>3</b>
<b>2</b>	<b>Concept Location</b>	<b>5</b>
2.1	Methodology . . . . .	5
2.2	Table Content Overview . . . . .	5
<b>3</b>	<b>Impact Analysis</b>	<b>7</b>
3.1	Brief Introduction . . . . .	7
3.2	Featureous Feature-code Characterization . . . . .	8
3.3	Featureous Feature Relations Characterization . . . . .	9
3.4	Feature-code correlation graph and feature-code correlation grid . . . . .	10
3.5	Table - Impact Analysis . . . . .	12
<b>4</b>	<b>Refactoring Patterns and Code smells</b>	<b>13</b>
4.1	JDisclosureToolBar class . . . . .	13
4.2	PaletteToolBarUI class . . . . .	15
<b>5</b>	<b>Refactoring Implementation</b>	<b>19</b>
5.1	JDisclosureToolBar class . . . . .	19
5.2	PaletteToolBarUI class - setFloating, dragTo and floatAt . . . . .	22
5.3	Impact of refactoring . . . . .	24
<b>6</b>	<b>Verification</b>	<b>25</b>
<b>7</b>	<b>Continuous Integration</b>	<b>26</b>
<b>8</b>	<b>Conclusion</b>	<b>27</b>
<b>9</b>	<b>Source Code</b>	<b>28</b>

# 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 improve their work efficiency, but have their most used tools and options within easy reach.



Busch31 on Sep 13

As a user I want to be able to drag and or drop the different parts of the toolbar, so that I can setup a custom workspace.

- ☐ I should be able to drag a part of the toolbar to a different location on the bar
- ☐ I should be able to rearrange the parts of the toolbar to have a custom layout

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.



Busch31 on Sep 13 (edited)

As a user I want to be able to hide and show the tool palette, so that I can have the maximum workspace that is also clear of tools

- ☐ When I click on the option to show / hide the tool palette it should do so.
- ☐ Since the toolbar has multiple different parts I should be able to hide one or more at any giving time.

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	<i>AbstractToolbar</i>	Quick Search Find all references Code removal	I started by looking at the different abstract classes for the whole project. Here I found the <i>AbstractToolbar</i> class which looked like the right abstract class I was looking for when my features name is <i>Tool Palette</i> . I then tested with <i>code removal</i> 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 <i>AbstractToolbar</i> was extended from.
2	<i>JDisclosureToolbar</i>	Code removal Extension reference Go to definition	When I look at what <i>AbstractToolbar</i> was extended from, I found the abstract class named <i>JDisclosureToolbar</i> , I again tried code removal, this time giving my first result. The abstract class <i>JDisclosureToolbar</i> is responsible for the <i>show/hide</i> feature of the <i>tool palette</i> , which I needed for my user story <i>Display</i> .
3	<i>ToolsToolbar</i>	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 <i>ToolsToolbar</i> . Here again with <i>code removal</i> 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 <i>tool palette</i> .
4	<i>PaletteToolbarUI</i>	Code removal Keyword Search	After I hit a dead end with the <i>Extension reference</i> tool method, I tried to do a <i>global search</i> on different keywords such as <i>Tool</i> , <i>UI</i> , <i>Palette</i> , <i>Bar</i> , <i>ToolBar</i> and other keywords that could be assimilated with my feature. With this tool method I found the <i>PaletteToolbarUI</i> 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

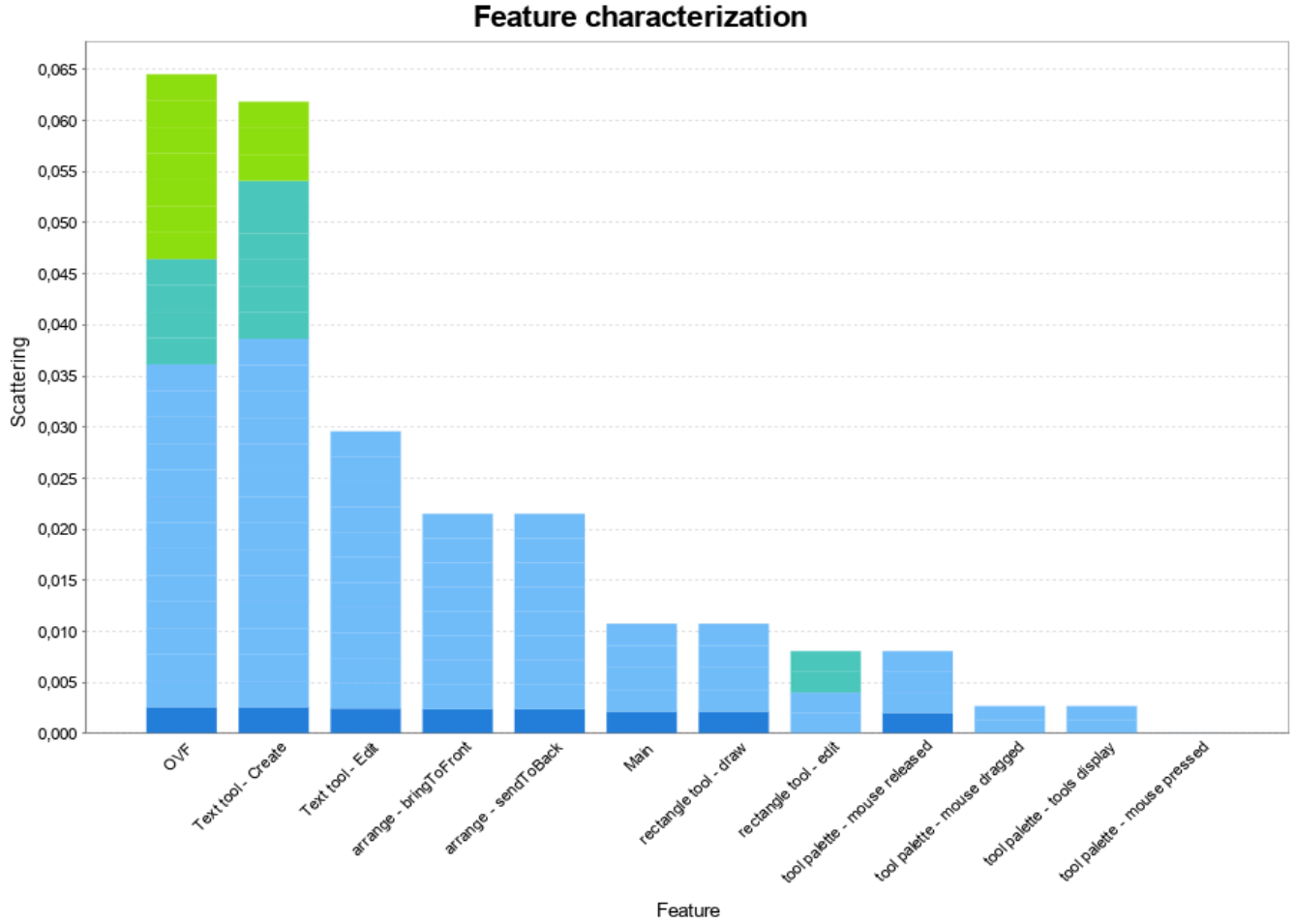


Figure 3: Featureous Feature-Code Characterization

As we can see, the units do contain a lot 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

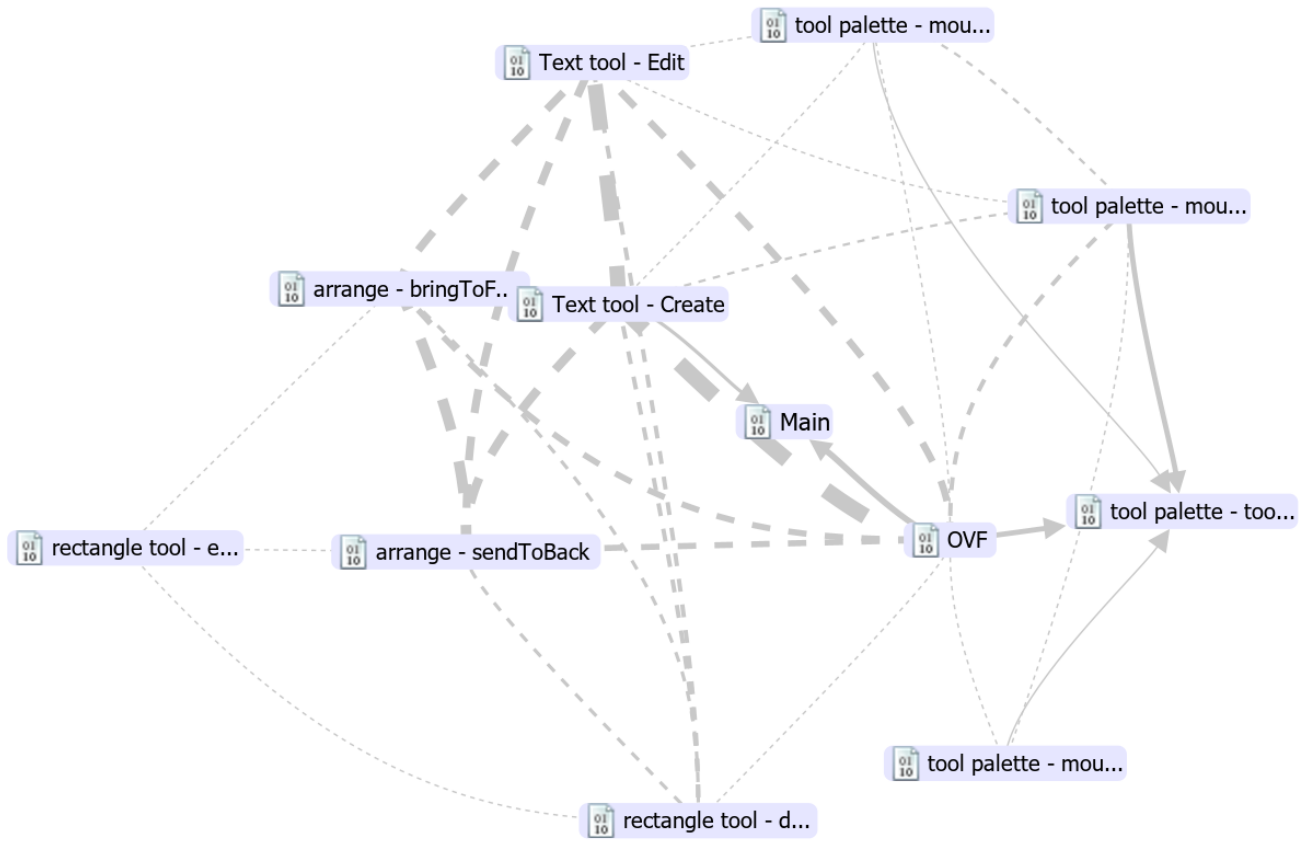


Figure 4: Featureous Feature Relations Characterization

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

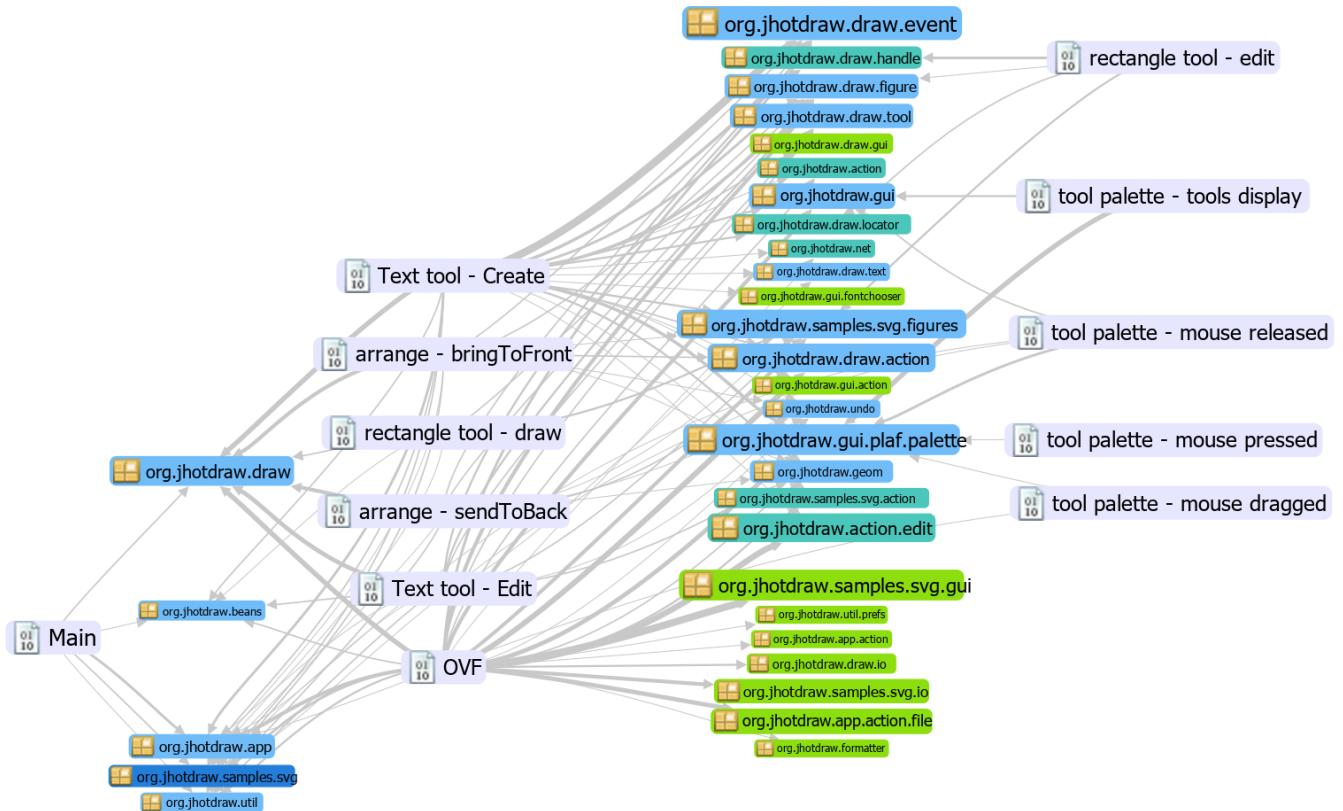


Figure 5: Feature-Package Correlation Graph

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.

	OVF	Text tool - Create	Text tool - Edit	arrange - bringToFront	arrange - sendToBack	tool palette - tools display	rectangle tool - edit	rectangle tool - draw	tool palette - mouse release	tool palette - mouse dragge	tool palette - mouse presse
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 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

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

```
37     private void initComponents() {
38         GridBagConstraints gbc;
39         AbstractButton btn;
40         setLayout(new GridBagLayout());
41         gbc = new GridBagConstraints();
42         if (disclosureButton == null) {
43             btn = new JButton();
44             btn.setUI((PaletteButtonUI) PaletteButtonUI.createUI(btn));
45             btn.setBorderPainted(false);
46             btn.setIcon(new DisclosureIcon());
47             btn.setOpaque(false);
48             disclosureButton = (JButton) btn;
49             disclosureButton.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, 1);
50             disclosureButton.putClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY, 2);
51             disclosureButton.addActionListener(new ActionListener() {
52                 @Override
53                 public void actionPerformed(ActionEvent e) {
54                     int newState = ((Integer) disclosureButton.getClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY) + 1)
55                         % (Integer) disclosureButton.getClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY);
56                     setDisclosureState(newState);
57                 }
58             });
59         } else {
60             btn = disclosureButton;
61         }
62         gbc.gridx = 0;
63         gbc.insets = new Insets(0, 1, 0, 1);
64         gbc.anchor = GridBagConstraints.SOUTHWEST;
65         gbc.fill = GridBagConstraints.NONE;
66         gbc.weighty = 1d;
67         gbc.weightx = 1d;
68         add(btn, gbc);
69         putClientProperty(PaletteToolBarUI.TOOLBAR_INSETS_OVERRIDE_PROPERTY, new Insets(0, 0, 0, 0));
70         putClientProperty(PaletteToolBarUI.TOOLBAR_ICON_PROPERTY, new EmptyIcon(10, 8));
71     }
```

Figure 7: Original initComponents function

```

79 public void setDisclosureState(int newValue) {
80     int oldValue = getDisclosureState();
81     disclosureButton.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, newValue);
82     removeAll();
83     JComponent c = getDisclosedComponent(newValue);
84     GridBagConstraints gbc = new GridBagConstraints();
85     if (c != null) {
86         gbc = new GridBagConstraints();
87         gbc.gridx = 1;
88         gbc.weightx = 1d;
89         gbc.weighty = 1d;
90         gbc.fill = GridBagConstraints.BOTH;
91         gbc.anchor = GridBagConstraints.WEST;
92         add(c, gbc);
93         gbc = new GridBagConstraints();
94         gbc.gridx = 0;
95         gbc.weightx = 0d;
96         gbc.insets = new Insets(0, 1, 0, 1);
97         gbc.weighty = 1d;
98         gbc.fill = GridBagConstraints.NONE;
99         gbc.anchor = GridBagConstraints.SOUTHWEST;
100        add(disclosureButton, gbc);
101    } else {
102        gbc = new GridBagConstraints();
103        gbc.gridx = 1;
104        gbc.weightx = 1d;
105        gbc.weighty = 1d;
106        gbc.fill = GridBagConstraints.NONE;
107        gbc.anchor = GridBagConstraints.SOUTHWEST;
108        gbc.insets = new Insets(0, 1, 0, 1);
109        add(disclosureButton, gbc);
110    }
111    invalidate();
112    Container parent = getParent();
113    while (parent.getParent() != null && !parent.getParent().isValid()) {
114        parent = parent.getParent();
115    }
116    parent.validate();
117    repaint();
118    firePropertyChange(DISCLOSURE_STATE_PROPERTY, oldValue, newValue);
119 }

```

Figure 8: Original setDisclosureState function

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

```
895 protected void floatAt(Point position, Point origin) {
896     if (toolBar.isFloatable() == true) {
897         try {
898             Point offset = dragWindow.getOffset();
899             if (offset == null) {
900                 offset = position;
901                 dragWindow.setOffset(offset);
902             }
903             Point global = new Point(origin.x + position.x,
904                                     origin.y + position.y);
905             setFloatingLocation(global.x - offset.x,
906                               global.y - offset.y);
907             if (dockingSource != null) {
908                 Point dockingPosition = dockingSource.getLocationOnScreen();
909                 Point comparisonPoint = new Point(global.x - dockingPosition.x,
910                                                  global.y - dockingPosition.y);
911                 if (canDock(dockingSource, comparisonPoint)) {
912                     setFloating(false, comparisonPoint);
913                 } else {
914                     setFloating(true, null);
915                 }
916             } else {
917                 setFloating(true, null);
918             }
919             dragWindow.setOffset(null);
920         } catch (IllegalComponentStateException e) {
921             // allowed empty
922         }
923     }
924 }
```

Figure 9: Original floatAt function

```

688 public void setFloating(boolean b, Point p) {
689     if (toolBar.isFloatable() == true) {
690         if (dragWindow != null) {
691             dragWindow.setVisible(false);
692         }
693         this.floating = b;
694         if (b && IS_FLOATING_ALLOWED) {
695             if (dockingSource == null) {
696                 dockingSource = toolBar.getParent();
697                 dockingSource.remove(toolBar);
698             }
699             constraintBeforeFloating = calculateConstraint();
700             if (propertyListener != null) {
701                 UIManager.addPropertyChangeListener(propertyListener);
702             }
703             if (floatingToolBar == null) {
704                 floatingToolBar = createFloatingWindow(toolBar);
705             }
706             floatingToolBar.getContentPane().add(toolBar, BorderLayout.CENTER);
707             if (floatingToolBar instanceof Window) {
708                 ((Window) floatingToolBar).pack();
709             }
710             if (floatingToolBar instanceof Window) {
711                 ((Window) floatingToolBar).setLocation(floatingX, floatingY);
712             }
713             if (floatingToolBar instanceof Window) {
714                 ((Window) floatingToolBar).setVisible(true);
715             }
716         } else {
717             if (floatingToolBar == null) {
718                 floatingToolBar = createFloatingWindow(toolBar);
719             }
720             if (floatingToolBar instanceof Window) {
721                 ((Window) floatingToolBar).setVisible(false);
722             }
723             floatingToolBar.getContentPane().remove(toolBar);
724             Integer constraint = getDockingConstraint(dockingSource,
725                 p);
726             if (constraint == null) {
727                 constraint = 0;
728             }
729             int orientation = mapConstraintToOrientation(constraint);
730             setOrientation(orientation);
731             if (dockingSource == null) {
732                 dockingSource = toolBar.getParent();
733             }
734             if (propertyListener != null) {
735                 UIManager.removePropertyChangeListener(propertyListener);
736             }
737             dockingSource.add(toolBar, constraint.intValue());
738         }
739         dockingSource.invalidate();
740         Container dockingSourceParent = dockingSource.getParent();
741         if (dockingSourceParent != null) {
742             dockingSourceParent.validate();
743         }
744         dockingSource.repaint();
745     }
746 }

```

Figure 10: Original setFloating function



```

847 protected void dragTo(Point position, Point origin) {
848     if (toolBar.isFloatable() == true) {
849         try {
850             if (dragWindow == null) {
851                 dragWindow = createDragWindow(toolBar);
852             }
853             Point offset = dragWindow.getOffset();
854             if (offset == null) {
855                 //Dimension size = toolBar.getPreferredSize();
856                 Dimension size = toolBar.getSize();
857                 offset = new Point(size.width / 2, size.height / 2);
858                 dragWindow.setOffset(offset);
859             }
860             Point global = new Point(origin.x + position.x,
861                                     origin.y + position.y);
862             Point dragPoint = new Point(global.x - offset.x,
863                                       global.y - offset.y);
864             if (dockingSource == null) {
865                 dockingSource = toolBar.getParent();
866             }
867             constraintBeforeFloating = calculateConstraint();
868             Point dockingPosition = dockingSource.getLocationOnScreen();
869             Point comparisonPoint = new Point(global.x - dockingPosition.x,
870                                             global.y - dockingPosition.y);
871             if (canDock(dockingSource, comparisonPoint)) {
872                 dragWindow.setBackground(getDockingColor());
873                 Object constraint = getDockingConstraint(dockingSource,
874                                                         comparisonPoint);
875                 int orientation = mapConstraintToOrientation(constraint);
876                 dragWindow.setOrientation(orientation);
877                 dragWindow.setBorderColor(dockingBorderColor);
878             } else {
879                 dragWindow.setBackground(getFloatingColor());
880                 dragWindow.setBorderColor(floatingBorderColor);
881             }
882             dragWindow.setLocation(dragPoint.x, dragPoint.y);
883             if (dragWindow.isVisible() == false) {
884                 //Dimension size = toolBar.getPreferredSize();
885                 Dimension size = toolBar.getSize();
886                 dragWindow.setSize(size.width, size.height);
887                 dragWindow.setVisible(true);
888             }
889         } catch (IllegalComponentStateException e) {
890             // allowed empty
891         }
892     }
893 }

```

Figure 11: Original dragTo function

Method Name	Description	Recommendation
<i>setFloating</i>	The method is doing too much, affecting readability and maintainability.	Break down into smaller, more focused methods.
<i>setFloating</i>	Multiple nested if-else statements increase complexity.	Refactor to reduce nested conditionals and simplify logic.
<i>dragTo</i>	The method is quite lengthy and performs many tasks.	Break down into smaller, more focused methods.
<i>dragTo</i>	Use of similar calculations and procedures as seen in other methods.	Abstract common functionality into a separate method or utility class.
<i>floatAt</i>	The method is overly long.	Break down into smaller, more focused methods.
<i>floatAt</i>	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, thereby improving the overall structure of the code.

```
49     private void initComponents() {  
50         setLayout(new GridBagLayout());  
51         if (disclosureButton == null) {  
52             initializeDisclosureButton();  
53         }  
54         addDisclosureButtonToToolBar();  
55         configureToolBarProperties();  
56     }
```

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 now enhances the toolbar's adaptability to state changes in the program. With these improvements, it is more readable and maintainable than before, making it easier to understand and modify the toolbar's behavior.

```
146     public void setDisclosureState(int newValue) {  
147         int oldValue = getDisclosureState();  
148         disclosureButton.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, newValue);  
149         removeAll();  
150         JComponent c = getDisclosedComponent(newValue);  
151  
152         if (c != null) {  
153             addComponentWithConstraints(c, createGridBagConstraints(1, 1d, 1d, GridBagConstraints.BOTH,  
GridBagConstraints.WEST));  
154             addComponentWithConstraints(disclosureButton, createButtonGridBagConstraints());  
155         } else {  
156             addComponentWithConstraints(disclosureButton, createButtonGridBagConstraints());  
157         }  
158  
159         validateAncestor();  
160         repaint();  
161         firePropertyChange(DISCLOSURE_STATE_PROPERTY, oldValue, newValue);  
162     }
```

Figure 13: Refactored setDisclosureState Method

#### 5.1.3 initializeDisclosureButton

The method is used to initialize 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.

```
65     private void initializeDisclosureButton() {
66         JButton btn = createDisclosureButton();
67         btn.addActionListener(createDisclosureButtonActionListener());
68         disclosureButton = btn;
69     }
```

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.

```
79     private JButton createDisclosureButton() {
80         JButton btn = new JButton();
81         btn.setUI((PaletteButtonUI) PaletteButtonUI.createUI(btn));
82         btn.setBorderPainted(false);
83         btn.setIcon(new DisclosureIcon());
84         btn.setOpaque(false);
85         btn.putClientProperty(DisclosureIcon.CURRENT_STATE_PROPERTY, 1);
86         btn.putClientProperty(DisclosureIcon.STATE_COUNT_PROPERTY, 2);
87         return btn;
88     }
```

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.

```
97     private ActionListener createDisclosureButtonActionListener() {
98         return new ActionListener() {
99             @Override
100             public void actionPerformed(ActionEvent e) {
101                 int currentState = getDisclosureState();
102                 int stateCount = getDisclosureStateCount();
103                 int newState = (currentState + 1) % stateCount;
104                 setDisclosureState(newState);
105             }
106         };
107     }
```

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.

```
116     private void addDisclosureButtonToToolBar() {
117         GridBagConstraints gbc = createGridBagConstraints(0, 1d, 1d, GridBagConstraints.NONE,
118         GridBagConstraints.SOUTHWEST);
119         gbc.insets = new Insets(0, 1, 0, 1);
120         add(disclosureButton, gbc);
121     }
```

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.

```
128     private void configureToolBarProperties() {
129         putClientProperty(PaletteToolBarUI.TOOLBAR_INSETS_OVERRIDE_PROPERTY, new Insets(0, 0, 0, 0));
130         putClientProperty(PaletteToolBarUI.TOOLBAR_ICON_PROPERTY, new EmptyIcon(10, 8));
131     }
```

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.

```
206     private void validateAncestor() {
207         Container parent = getParent();
208         while (parent.getParent() != null && !parent.getParent().isValid()) {
209             parent = parent.getParent();
210         }
211         parent.validate();
212     }
```

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.

```
704     public void setFloating(boolean b, Point p) {
705         if (!toolBar.isFloatable()) {
706             return;
707         }
708         hideDragWindow();
709         this.floating = b;
710
711         if (b && IS_FLOATING_ALLOWED) {
712             prepareFloating();
713         } else {
714             prepareDocking(p);
715         }
716     }
717 }
```

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.

```
724     public void floatAt(Point position, Point origin) {
725         if (!toolBar.isFloatable()) {
726             return;
727         }
728
729         try {
730             Point global = calculateGlobalPosition(position, origin);
731             Point comparisonPoint = calculateComparisonPoint(global);
732             boolean shouldDock = canDock(dockingSource, comparisonPoint);
733
734             setFloatingLocation(global.x, global.y); // This sets the location for floating
735             setFloating(!shouldDock, shouldDock ? comparisonPoint : null); // This sets the floating state
736         } catch (IllegalComponentStateException e) {
737             // Handle exception if necessary
738         }
739     }
```

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.

```
741 private void prepareFloating() {
742     initializeDockingSource();
743     constraintBeforeFloating = calculateConstraint();
744     showFloatingWindow();
745 }
```

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.

```
752 private void initializeDockingSource() {
753     if (dockingSource == null) {
754         dockingSource = toolBar.getParent();
755         dockingSource.remove(toolBar);
756     }
757 }
```

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.

```
759 private void showFloatingWindow() {
760     createFloatingToolBar();
761     updateFloatingWindowAppearance();
762 }
```

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.

```

932     protected void dragTo(Point position, Point origin) {
933         if (toolBar.isFloatable()) {
934             initializeDragWindow();
935             Point dragPoint = calculateDragPoint(position, origin);
936             updateDockingSource();
937             updateDragWindowAppearance(dragPoint, origin);
938             setDragWindowLocationAndVisibility(dragPoint);
939         }
940     }

```

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.

```

947     private void initializeDragWindow() {
948         if (dragWindow == null) {
949             dragWindow = createDragWindow(toolBar);
950         }
951     }

```

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

hej

## 7 Continuous Integration

hej

## 8 Conclusion

hej

## 9 Source Code

hej