Software Maintenance Note Report

Software Maintenance Report

vipet23@student.sdu.dk

2026-01-05

1. Change Request

For the change request redo and undo functioanllity was choosen, the purpose is to give an user the ability to undo and/or redo actions getting them back to a “good” state. Undo and redo are effectivly two “sub features”, but as they both share simmilair functioanllity they are grouped together under one change feature request.

The user story is formulated as follows:

> As a user, i want the ability to undo and redo my actions so that i can recover from mistakes and return to a previous working state.

2. Concept Location

Concept location helps the developers find which part of the code implements a feature. This is helpfull when adding a new feature as it is important to modify and extend the correct abstractions. Additionally it helps find a set of starting classes, from which impact anylsis can be run, giving a great understand of the scope of a particular change request.

Figure 1 shows an activity diagram, of going from having a change requst. To understanding the concept location and which parts of the code are relevant.

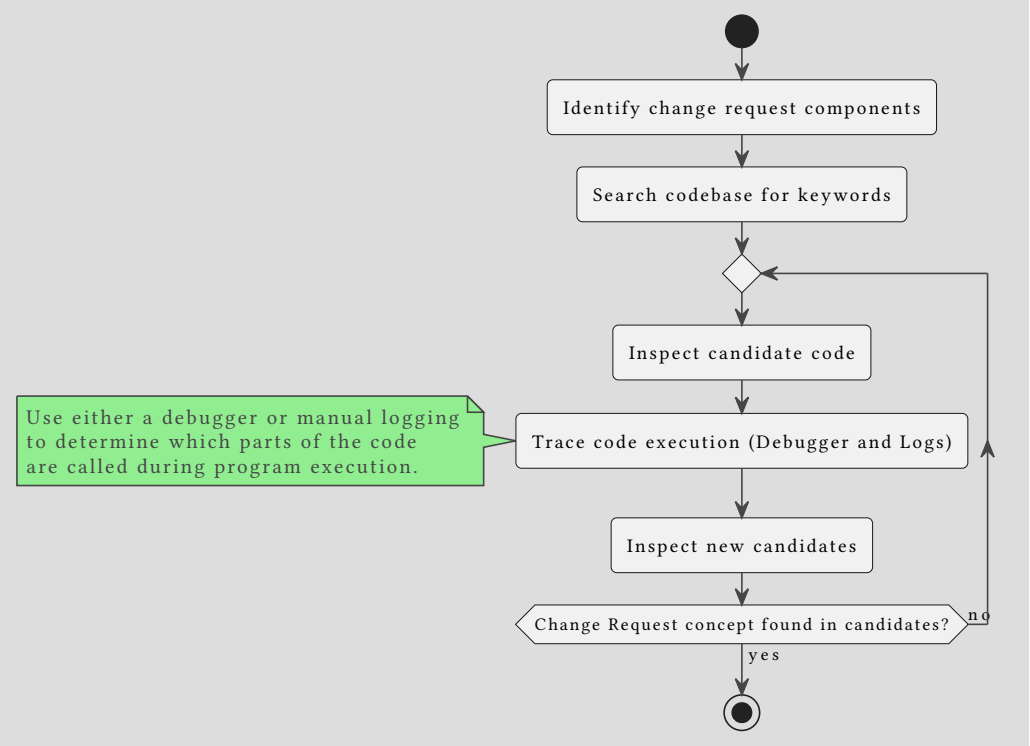

Figure 1: Activity diagram of concept location process.

For finding the concepts involved in the Undo/Redo change request, an iterative approach was applied treating both parts of the feature as sepperate entities. This means the process was repeated for both undo aswell as redo. For each of these processes the general methodology focusses on finding some part of the code where the feature might logically live, and then searching that immediate area for relevant code.

Finding the initial relevant class is achieved by searching the codebase for simmilar and relevant features. Searching a codebase can be done through many means, a common tool found on most

POSIX system would be grep. Finding the relevant classes for undo and redo was relativly easy, as there is a file called **UndoRedoManager** which is a great startpoint for concept location.

With the initial file found, I used the debugger to place stops around the code. I then ran the program and executed a undo/redo action in an attempt to see what functions where called when. This allowed me to branch out into other relevant files which were called as part of the undo/redo action.

This is effectivly an iterative process where I continued to find new break points for my debugger, and continued running the relevant part of the application untill no more new files were being discovered.

Table 1 shows the files visited during concept location, along with what the responsibilty of each class is. Generally the Undo/Redo feature goes through a couple of classes, it has a manager which is responsible for calling the actions. Both Undo and Redo feature has a relevant Action class which extends the abstact Action class. Addtionally some classes related to the ui are present as they define the buttons for the both actions.

Domain Class	Responsibility
UndoAction	• Handle action (safely call **UndoRedoManager.undo()**). • Configure button label (via **ResourceBundleUtil**).
RedoAction	• Handle action (safely call **UndoRedoManager.redo()**). • Configure button label (via **ResourceBundleUtil**).
UndoRedoManager	• Manage undo/redo logic for edits. • Control when undo/redo buttons light up and become pressable. • Define behavior when undo/redo is executed (invoked by Action classes).
ActionsToolBar	• Create buttons for ActionsToolBar. • Determine which **UndoRedoManager** to use for buttons.
SVGDrawingPanel	• Create the drawing panel. • Instantiate the **UndoRedoManager** and pass it down through the chain.
ResourceBundleUtil	• Provide resources associated with features/buttons (eg. undo and redo icons).

Table 1: Resulting classes from change request found during Concept Location.

3. Impact Analysis

Impact analysis takes the initial set of classes found during concept location and expands them to the “real” estimated impact set. This can be done by creating an interaction diagram from the initial impact set, and then iterativly running through said diagram marking classes as either next, inspected, propagating or changed. The goal is to identify specefically which classes will have to be changed for the change request to be implemented.

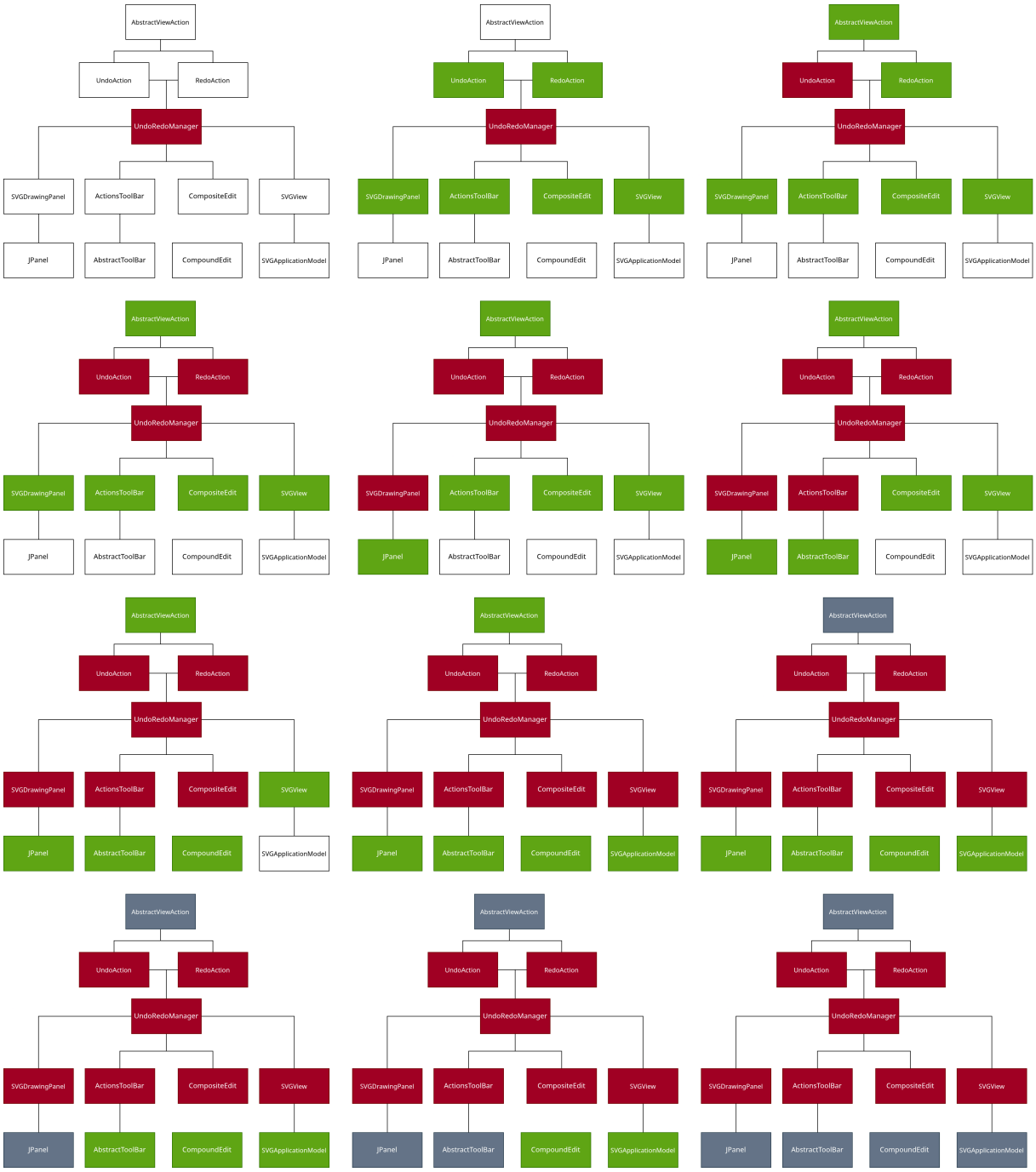

Figure 2: Interaction Diagram showing process of marking classes (green = next, red = change, grey = unchanged.

Figure 2 shows the process of iterativly marking classes which interact with the previously identified **UndoRedoManager**. Starting with only having that one class marked as red/change and ending with multiple classes that have been inspect and marked as either change or no change. Addtionally the complete list of packages and the amount of classes visted can be seen in Table 2, along with some genereal information about each package and what their responsibilities are. This helps keep an overview of the impact along with what is relevant withing each package.

Package Name	# of classes	Comments
org.jhotdraw.undo	4	This package contains the main logic for undo/redo along with the manger that controls them. As such this is the crux of this feature, this layer is effectively the layer between java/javax implementation and ours.
org.jhotdraw.action	1	This package contains the basis of which a action is. Undo/Redo counts as a action, therefore this contains the necesarry implementation and contractual obligations for our new actions to work. This class is unlikely to be changed because of our new feature as it is the basic abstaction multiple other features are build upon.
org.jhotdraw.samples.svg	3	This package contains alot of the logic required specefically for the SVG application. This records the edits that have been made, and sets the specefic instance of Undo/Redo to use.
org.jhotdraw.samples.svg.gui	1	This package contains the logic for the ui bar, where a user can click on buttons to execute an action. This is relevant as there needs to be implemented a button for the user to click.

Table 2: List of packages visited during impact analysis.

Focusing on packages means we only see the top level details, this helps with keeping the focus on what parts of the codebase are impacted and not specefically what file/function is impacted.

4. Refactoring

During anylisis of the **UndoRedoManager** multiple pieces of code which are no longer used within the program was discovered. Unused code is a code smell as it adds no value to the actual program, while still managing to confuse maintainers as they have to filter out the irrelevant code. It is better to remove the code instead. If it is needed in the future then it can still be found in version control, or a new implementation can be written.

```java
public void setLocale(Locale l) {
  labels = ResourceBundleUtil.getBundle("org.jhotdraw.undo.Labels", l);
}
```

Listing 1: Unused code for setting locale in **UndoRedoManager**.

Fixing this issue is done by using the "Remove Dead Code" strategy, which simlpy entails removing the code. Listing 1 shows the offending code before it was removed.

```java
public static final UndoableEdit DISCARD_ALL_EDITS = new
AbstractUndoableEdit() {
    private static final long serialVersionUID = 1L;

    @Override
    public boolean canUndo() {
        return false;
    }

    @Override
    public boolean canRedo() {
        return false;
    }
};
```

Listing 2: Unused code for sending an edit which cannot be undone.

Fixing the code shown in Listing 2, is also as simple as removing it. It serves no functionallity within the program and is likely an artifact of a developer manually testing something at some point.

In **UndoRedoManger** there was also a set of 3 methods which where all structurede exactly the same, and whose purpose is to function in a simmilar fashion. This is a bad code smell where code is duplated which also violates the Don't Repeat Yourself Principles. The duplicate code can be seen in Listing 3. The planned change is to refactor the logic of each method out into a singular abstraction, then each specefic method can call that abstraction. This ensures that if any common changes are to be made in the future, it only has to change in one place. An example could be if logging functionallity is required, currently that code would have to be written in 3 places, making it easy to forget one.

```java
    @Override
     public void undo() throws CannotUndoException {
         undoOrRedoInProgress = true;
         try {
             super.undo();
         } finally {
             undoOrRedoInProgress = false;
             updateActions();
         }
     }

    @Override
     public void redo()
             throws CannotUndoException {
         undoOrRedoInProgress = true;
         try {
             super.redo();
         } finally {
             undoOrRedoInProgress = false;
             updateActions();
         }
     }

    @Override
     public void undoOrRedo()
             throws CannotUndoException, CannotRedoException {
         undoOrRedoInProgress = true;
         try {
             super.undoOrRedo();
         } finally {
             undoOrRedoInProgress = false;
             updateActions();
         }
     }
```

Listing 3: Duplicate Code refactored into singular abstraction.

Listing 4 shows the code after refactoring. This is an example of Extract Method, where the functionallity from multiple methods are extracted into a singular one. This specefic case is a little unique as it requires an addtional abstraction, instead of just statically extracting the code.

```java
@FunctionalInterface
private interface UndoRedoOperation {
    void execute() throws CannotUndoException, CannotRedoException;
}

private void executeUndoRedoOperation(UndoRedoOperation operation)
  throws CannotUndoException, CannotRedoException {
    undoOrRedoInProgress = true;
    try {
        operation.execute();
    } finally {
        undoOrRedoInProgress = false;
        updateActions();
    }
}

@Override
public void undo() throws CannotUndoException {
    executeUndoRedoOperation(super::undo);
}

@Override
public void redo() throws CannotUndoException {
    executeUndoRedoOperation(super::redo);
}

@Override
public void undoOrRedo() throws CannotUndoException, CannotRedoException {
    executeUndoRedoOperation(super::undoOrRedo);
}
```

Listing 4: Duplicate code refactored into common interface.

Listing 5 shows a code snippet of the read method on **SVGDrawingPanel**. The code is fairly long, has many layers of nesting and relies on comments to explain certain gaps in the code (e.g "suppress silently" or "We get here if reading was succesful"). All of these problems leads to code which is hard to read, and prone to bugs. The method **read** is also overloaded to do different things based on given arguments, this implementation contains signifcant amounts of duplicated code. Refactoring code which does not directly correlate to the undo redo feature at hand, might seem like a waste of time, however it is important to consider how this code plays into the feature at large. By refactoring it now, the change becomes easier to implement as we better understand how the code functions and flow. This will in turn give us greater reasurance as to why it works or fails.

The planned refactoring is a "extract methods" where the common parts of the methods are extracted into its own seperate methods. This increases reusability and makes both methods significantly easier to read and understand. Looking at the method signature, the URI object is called f. This is likely shorthand for something, however it is unreadable for a developer without insider knowledge. It can be refactored by changing the name to something appropiate like uri.

```java
    public void read(URI f) throws IOException {
      Drawing newDrawing = createDrawing();
      if (newDrawing.getInputFormats().size() == 0) {
          throw new InternalError("Drawing object has no input formats.");
      }
      IOException firstIOException = null;
      for (InputFormat format : newDrawing.getInputFormats()) {
          try {
              format.read(f, newDrawing);
              final Drawing loadedDrawing = newDrawing;
              Runnable r = new Runnable() {
                  @Override
                  public void run() {
                      setDrawing(loadedDrawing);
                  }
              };
              if (SwingUtilities.isEventDispatchThread()) {
                  r.run();
              } else {
                  try {
                      SwingUtilities.invokeAndWait(r);
                  } catch (InterruptedException ex) {
                      // suppress silently
                  } catch (InvocationTargetException ex) {
                      InternalError ie = new InternalError("Error setting
                      drawing.");
                      ie.initCause(ex);
                      throw ie;
                  }
              }
              // We get here if reading was successful.
              return;
          } catch (IOException e) {
              // We get here if reading failed.
              // We only preserve the exception of the first input format,
              // because that's the one which is best suited for this drawing.
              if (firstIOException == null) {
                  firstIOException = e;
              }
          }
      }
      throw firstIOException;
  }

    /** Overloaded Method Signature **/
  public void read(URI f, InputFormat format) throws IOException {}
```

Listing 5: Example of a long method, which has too many responsibilities.

The final result of both of the **read** methods can be seen in Listing 6. The factored out methods are not included, as one should ideally be capable of understanding the code by simply reading the refactored methods.

```java
public void read(URI uri) throws IOException {
        Drawing newDrawing = createDrawing();
        validateDrawing(newDrawing);

        IOException firstIOException = null;
        for (InputFormat format : newDrawing.getInputFormats()) {
            try {
                format.read(uri, newDrawing);
                updateDrawingOnEDT(newDrawing);
                return;
            } catch (IOException e) {
                if (firstIOException == null) firstIOException = e;
            }
        }
        throw firstIOException;
    }

    public void read(URI uri, InputFormat format) throws IOException {
        if (format == null) {
            read(uri);
            return;
        }

        Drawing newDrawing = createDrawing();
        validateDrawing(newDrawing);

        format.read(uri, newDrawing);
        updateDrawingOnEDT(newDrawing);
    }
```

Listing 6: Read methods after refactoring.

Another simmilar problem is present in the same class, within the write methods. These methods once again share large parts duplicate code, use bad names for arguments and do too much for a singular method. This can be seen in Listing 7, once again there is an overloaded method accepting a format.

Simmilarly to before, the refactoring will consist of extracting duplicate parts of the code, and cleaning up the bad parameter names. At the same time moving the guard clause into tis own method to make easier to understand the validation part of the logic.

```java
    public void write(URI uri) throws IOException {
        // Defensively clone the drawing object, so that we are not
        // affected by changes of the drawing while we write it into the file.
        final Drawing[] helper = new Drawing[1];
        Runnable r = new Runnable() {
            @Override
            public void run() {
                helper[0] = (Drawing) getDrawing().clone();
            }
        };
        if (SwingUtilities.isEventDispatchThread()) {
            r.run();
        } else {
            try {
                SwingUtilities.invokeAndWait(r);
            } catch (InterruptedException ex) {
                // suppress silently
            } catch (InvocationTargetException ex) {
                InternalError ie = new InternalError("Error getting drawing.");
                ie.initCause(ex);
                throw ie;
            }
        }
        Drawing saveDrawing = helper[0];
        if (saveDrawing.getOutputFormats().size() == 0) {
            throw new InternalError("Drawing object has no output formats.");
        }
        // Try out all output formats until we find one which accepts the
        // filename entered by the user.
        File f = new File(uri);
        for (OutputFormat format : saveDrawing.getOutputFormats()) {
            if (format.getFileFilter().accept(f)) {
                format.write(uri, saveDrawing);
                // We get here if writing was successful.
                // We can return since we are done.
                return;
            }
        }
        throw new IOException("No output format for " + f.getName());
    }
```

Listing 7: One of the write methods before refactoring.

An example of one of the refactored write methods can be seen in Listing 8.

```java
    public void write(URI uri) throws IOException {
        Drawing drawingSnapshot = getDrawingSnapshotFromEDT();
        validateOutputFormats(drawingSnapshot);

        File file = new File(uri);
        for (OutputFormat format : drawingSnapshot.getOutputFormats()) {
            if (format.getFileFilter().accept(file)) {
                format.write(uri, drawingSnapshot);
                return;
            }
        }
        throw new IOException("No output format for " + file.getName());
    }
```

Listing 8: One of the write methods after refactoring.

4.1. Clean Code and Solid

Clean code is a set of principles which govern how to write code that is easy for other developers to understand. When writing software we should strive to create code that others can easily read and understand, this increases the likelyhood that they can maintain the software should that be needed. A common set of princples often talked about within Clean code is the SOLID principles which are a set of principles for writing OO code. The goal of SOLID is to ensure code remains readable, testable and extendable.

```java
  public void write(URI uri, OutputFormat format) throws IOException {
        if (format == null) {
            write(uri);
            return;
        }

        Drawing drawingSnapshot = getDrawingSnapshotFromEDT();
        format.write(uri, drawingSnapshot);
    }
```

Listing 9: Example of SOLID principles in refactored code.

Listing 9 shows an example of solid principles in the previously refactored code, in Table 3 the explanations for how each principle is followed in this specefic example can be found.

Principle	Reason
S	The method only has one reason to change, this being the high level calling of the format's write method. The implementation writing has been delegated to another more appropiate place.
O	The method relies upon the **OutputFormat** abstraction, this makes it easy to extend functionallity simply by implementing the **OutputFormat** class.
L	The method does not make a distinction between what implementation the **OutputFormat** has, it only asserts that if it is null then we use the simplier method.
I	Since the code does not define a interface, it can't really adhere to this principle. However it could be made more specefic by accepting in an abstraction for format which only features the **write** method, as that is the only one used.
D	As the method depends upon an abstraction and not a concrete class it follows this principle. This is part of how the method also adheares to the open close principle.

Table 3: SOLID explanations for Listing 9

An overview of the **OutputFormat** abstraction ands the abstractons which extend it can be seen in Figure 3. This clearly shows how the open close principle comes to life by allowing any number of child classes or interfaces to derive from it. Additionally it hightlights the use of Liskov substituion principle where **SVGZOutputFromat** can replace both **SVGOutputFormat** and **OutputFormat** because itself follows the contracts defined within them.

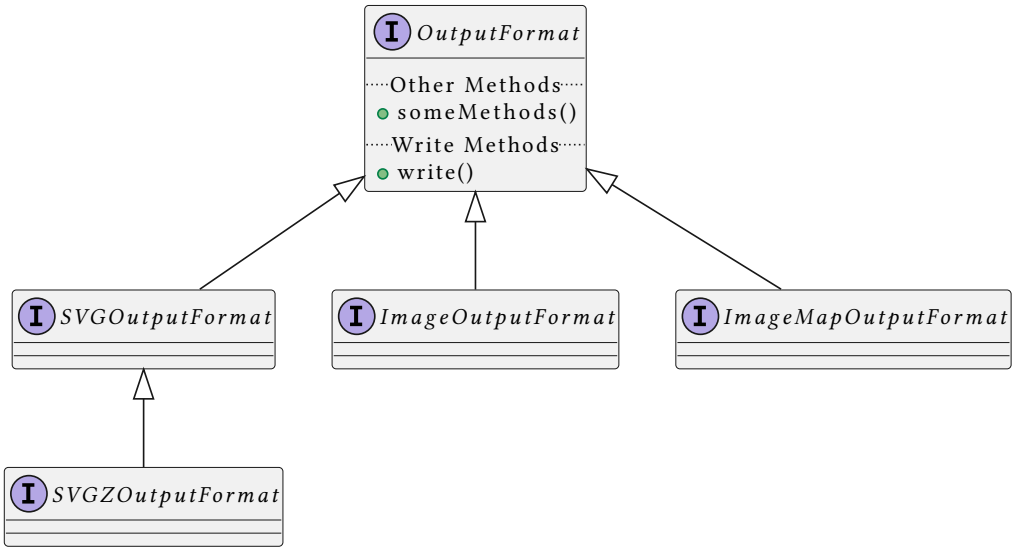

Figure 3: Interfaces in case study adhearing to both open close and Liskov substituion principles

4.2. Clean architecture

Clean architecture is an architectual pattern that attempts to decouple the ui, database, external frameworks, etc. from the core buiness logic. The idea is that the parts of the code which changes the most should be "plugged in to" the code that does not change. This creates a system where developers can easily switch between differnt frameworks, without impacting the actual implementation of its buiness rules. Additionally by not having the buiness rules depend on other external factors, it creates a system where testing can be done independent of the ui, databases or external factors. All in all this helps ensure that the program is maintanable over time, even when frameworks or ui principles becomes abandoned.

JHotdraw follows some parts of clean architecture, with different parts of the application being split into their own seperate modules. This does create a nice seperation, however it does not properly follow clean architecture as multiple parts of the code depend upon the ui framework swing. An example of this can be seen in Listing 10 where the **UndoRedoManger** depends upon the swing framework. This violates the idea that the undo redo logic should be independent from a specefic ui framework. This does also mean that, if the swing framework had to be removed from the application it would require large scale refactorings to large parts of the buiness logic.

```java
public class UndoRedoManager extends UndoManager {
  //javax.swing.undo.UndoManager
  }
```

Listing 10: Swing dependency in non UI utillity class.

5. Vertification

5.1. Assertions

Assertions are used to test that some code enver reaches a specefic state. The idea is that if an assertion evaluates to false then the program crashes. Hopefully the developers or QA personell can catch this before pushing the code.

```java
    private void executeUndoRedoOperation(UndoRedoOperation operation)
      throws CannotUndoException, CannotRedoException {
        assert !undoOrRedoInProgress : "An UndoRedoOperation is already in
        progress.";
        undoOrRedoInProgress = true;
        try {
            operation.execute();
        } finally {
            undoOrRedoInProgress = false;
            updateActions();
        }
  }
```

Listing 11: Asserting that no undo or redo is already in progress.

Listing 11 shows an example of asserting something that must never be true. The **undoOrRedoInProgress** flag must be false when we enter the function, else we might end up in a recursive scenario. By adding the assertion, the likelyhood a bug which causes this to trigger is caught increases.

5.2. Tests

Unit tests are small tests, which test the indiviudal parts of a class. This will most often present itself as test for methods where each test, ensures a specefic outcomes or results happens from said method. A well implemented test suite can help further refactorings by providing a set of critera for how a piece of code should act or perform.

Testing the **UndoRedoManger** should focus primarily on each indiviudal method and what side effects it might introduce into the class. It should test and validate the side effects of the methods, ensuring it is always left in a predicatble state. Addtionally mocks are important for ensuring that only one particular part of the code is being tested. This ensures that tests for **UndoRedoManger** does not fail because an error is introduced somewhere else.

```java
public class UndoRedoManagerTest {
    private UndoRedoManager instance;
    private UndoableEdit significantEdit;
    private UndoableEdit nonSignificantEdit;

      @Before
    public void setUp() {
        ResourceBundleUtil resourceBundleUtil = mock(ResourceBundleUtil.class);
        instance = new UndoRedoManager(resourceBundleUtil);

        significantEdit = mock(UndoableEdit.class);
        when(significantEdit.isSignificant()).thenReturn(true);

        nonSignificantEdit = mock(UndoableEdit.class);
        when(nonSignificantEdit.isSignificant()).thenReturn(false);
    }
}
```

Listing 12: Instances and **setUp** method for **UndoRedoMangerTest**.

To allow for multiple test, a **setUp()** method which declares the necesarry variables and mocks are defined. This makes the code easier to read, and follows the DRY principles. Listing 12 shows the code which mockes the necesarry classes and creates an instance of the system under test. Addtionally the mocks for non relevant parts of the code can be seen, ensuring that only the **UndoRedoManger** is being tested.

```java
    @Test
public void addSignificantEditUpdatesInstance() {
    instance.addEdit(significantEdit);

    assertTrue(instance.hasSignificantEdits());
}
```

Listing 13: Test asserting that the **hasSignificantEdit** flag is set properly.

Listing 13 shows a test which ensures that the instance has the correct flag set to true after adding a significant edit. The test does not make any assumptions about how that flag is set, just that it must be set.

```java
    @Test
public void discardAllEditsResetsState() {
    instance.addEdit(significantEdit);
    instance.discardAllEdits();

    assertFalse(instance.hasSignificantEdits());
    assertFalse(instance.canUndo());
}
```

Listing 14: Test asserting that the internal state of the instance is correct after discarding edits.

Another example of ensuring a specefic internal state after a method call can be seen in Listing 14. This checks for the logic that after having discard all edits, it should not be possible to either have a signifacnt edit, nor undo an edit. Checking the side effects like this, does introduce an issue where how the object works is now defined. If in the future the object was to be refactored to include less side effects, then the test would need to be refactored aswell. This is generally acceptable as the test which use old functionallity should also be refactored or removed when refactoring code. This plays into the idea that test are "first class citizens" i.e. it is as important as production code.

```java
    @Test
public void addSignificantEditReturnsTrue() {
    boolean result = instance.addEdit(significantEdit);
    assertTrue(result);
}
```

Listing 15: Test ensuring that adding an edit runs correctly.

Because code that tests side effect may be refactored in the future, it is important to also test how the code itself behaves. Listing 15 shows how that is tested for the code present in Listing 13. By doing this both the side effects and the code excetion is tested, this will also lead to better test logs for troubleshooting if necesarry.

```java
    @Test(expected = CannotUndoException.class)
public void undoWithEmptyStackThrowsException() {
    instance.undo();
}

    @Test(expected = CannotRedoException.class)
public void redoWithEmptyStackThrowsException() {
    instance.redo();
}
```

Listing 16: Test asserting that the correct exceptions are thrown with bad input.

Simply testing the best case scenario is not enought, a test suite should also contain tests for the worst case / bad input scenario. This often comes down to testing that the right exception or errors are thrown when something goes wrong. In Listing 16 is an example of tests that cover what exceptions should be thrown when undo/redo is called on an non existing edit.

All of the unit tests together ensure that the code which is being tested runs in a predetermined and predicatble way.

5.3. Behavior Driven Development scenarios

Beahvior Driver Development is a development process where the user wan'ts and their behaivor are put first. It is done by translating user stories into Given-When-Then scenariosm which can then be tested in code. These test care more about how the systems functions from the user perspective than the internal affairs of each class.

As an example the initial user story of the undo redo change request is as follows:

> As a user, i want the ability to undo and redo my actions so that i can recover from mistakes and return to a previous working state.

This can be turned into multiple given-when-then scenarios

Scenario 1

Given the user has a new drawing, and the user performs and edit.
When the user selects undo.
Then the edit should be undone, and the redo option should become available.

Scenario 2

Given the user has a drawing with an undone edit.
When the user selects redo.
Then the redo should be done,

Figure 4: Examples of Given-When-Then Scenario translated from a user story.

Figure 4 shows examples of two scenarios that are derived from the change requests user story. It is possible to translate these scenarios into code, which can than automatically be ran along with the other tests. This is another test type, which helps ensure that the parts of the code which users interact with are running optimally. Additionally bdd scenarios are often more readable for non techinal people, thus these scenarios can serve as a form of documentation of the code. Ideally these scenarios should only ever change, if the buiness goals/rules change. Outside of that scenario they should remain static, as they describe the users path through a system.

```java
@Test
public void undoing_a_meaningful_edit_should_enable_redo() {
    given().an_empty_edit_history()
            .and().a_meaningful_edit_was_made();

    when().the_edit_is_made()
            .and().the_undo_action_is_executed();

    then().the_edit_is_stored()
            .and().the_undo_option_is_not_available()
            .and().the_redo_option_is_available();
}
```

Listing 17: Scenario 1 translated into JGiven test.

Shows in Listing 17 is the code translated version of scenario 1. This helps assert that when some edit is made and undone, then the redo action becomes aviable. That is a core part of the undo redo functionallity and is something which should always happen.

```java
public GivenUndoRedoManager an_empty_edit_history() {
    ResourceBundleUtil resourceBundleUtil = mock(ResourceBundleUtil.class);
    instance = new UndoRedoManager(resourceBundleUtil);
    return this;
}
```

Listing 18: Method used to setup a given state.

Each part of the given, when and then methods are defined in a seperate class. This promotes code reuse, as it is defined once and then used in other classes. An example of how **an_empty_edit_history** is setup can be seen in Listing 18.

Bibliography