Peer Review of Group 4 - Pixellent

Design principles and Patterns

The application uses the strategy pattern for the different drawing tools. It allows for adding new drawing tools very easily (which is a design goal in the SDD), because all “tool input” strategies are abstracted away in the strategies package. Adding a new tool therefore essentially only requires a new class in the tools package, an edit in ToolFactory, and some changes needed in the view/controller to add the component to the GUI.

The observer pattern is used throughout the application. Listeners are stored as interface types (bonus points for abstraction!) and are generally not implemented as classes, but as single lambdas instead. However java interfaces such as Consumer could be used instead to prevent reimplementing the same things.

The application makes use of the factory pattern in the ActionFactory class. However, this class contains 50 static methods and is generally unwieldy. It goes against the Single-Responsibility principle by doing two things at the same time: keeping a collection of listeners for certain events and creating new Action objects on request. The StrategyFactory and ToolFactory are a lot cleaner, and the latter of which is a very nice implementation of the Factory pattern.

The codebase uses the module pattern successfully to sort source files depending on whether they belong to the model, the view, or the controller. The model package also contains further modules, which is helpful.

The (enormous) ICanvasView interface appears to resemble the facade pattern, in that it provides a facade for interfacing with the CanvasView class. However, the CanvasView class only has implementations of this interface and is only used in one place (WindowController), so the facade is unnecessary. The setText() method is not even implemented in the CanvasView. This pattern should not be used here, because it does not contribute a sensible or useful abstraction layer.

Code structure and abstractions

The project SDD declares that a MVC structure is used for this project. Overall, the MVC structure is respected at the package level. The Model package does not depend on the View or Controller packages, while the View package only depends on the Model package. However, there are places on a lower level where the idea of MVC seems to have slipped out of mind.

The BrushController and ToolBox classes are examples of where the concept of MVC does not hold, as these classes takes on responsibilities concerning both the Controller and the View. Besides setting up event handlers to make the views interactive, they also inherit from JavaFX’s Node subclasses and act as views themselves.

There are also instances where the different packages have dependencies that are not congruent with their responsibilities. The Model, for example, depends on Java awt, which is a visual framework meant to serve mainly in the View part of the program.

A few abstract classes are introduced to increase the reuse of code. This is executed in a good manner for the most part, abstracting actions such as the AbstractTool::use and AbstractPointStrategy::activate that are shared by all subclasses. However, there are places where implementations of abstract methods are left empty in the subclasses.

The parts of the abstract class that are not implemented by every subclass don't really belong in an abstract super class and should perhaps be moved to one or more interfaces instead.

Code Style

The code style is not horrible, but there is room for improvement. The most jarring problem is the inconsistencies of naming, indentation, brackets, etc; things that could easily be avoided by using a linter and proper CI [See more @Quality Assurance].

First of all, some names do not use camel case properly. And when they do it's not very consistent. As an example, we have both the methods setupOKButton and setOkBlur. In general, you should avoid consecutive capital letters, and otherwise use them consequently if you insist on using them for certain acronyms.

Moreover, indentation varies between files, and sometimes even within a single file. A prime example of this is in FileController.java.

```java
public void openImage() {

  final FileChooser fileChooser = new FileChooser();

  final FileChooser.ExtensionFilter extensionFilter = new FileChooser.ExtensionFilter("All
Image","*.png","*.jpg");
   fileChooser.getExtensionFilters().addAll(extensionFilter);

   final File file = fileChooser.showOpenDialog(new Stage());

   if (file !=null){

      final Image image = new Image(file.toURI().toString());

       final GraphicsContext gc = canvas.getGraphicsContext2D();
       gc.drawImage(image,0,0,image.getWidth(),image.getHeight());
   }
}
```

In this single method we see lines of code being indented with both 3, 4, 7 and 8 spaces. We also see varied amount of whitespace before codeblocks and the != operator only has whitespace on the left for some reason? We also see a lack of whitespace between arguments in the gc.drawImage method call, while such whitespace is present in other parts of the code.

However, there are both good and bad things to say about the naming. The tests are in general quite well named and their purpose is made clear. This is especially true for the different tool-tests which have adopted a [cause]Should[Effect] style naming scheme. One critique is again that this is not used consistently everywhere.

One of the strong points of this project is its amount of abstraction. Unfortunately this is more of a "quantity" than a "quality" thing. With great abstraction comes great responsibility; it becomes even more crucial to use good, easily interpretable method names to avoid confusion and keep the code maintainable. Here is one of many possible examples of where the code simply becomes confusing:

```java
private void giveToolStrategyAndUseIt(AbstractTool tool, boolean alternativeFunction) {
   tool.setAlternative(alternativeFunction);
   tool.use(canvas::addOperation, false);
}
```

First of all, it is unclear what the boolean alternativeFunction actually does; the name alone makes it difficult to understand that it is a *boolean* we are dealing with and not a lambda. But even knowing that, the methods called - why they are called and what they do - is confusing.

Quality Assurance

The project's code coverage can be seen in the following table.

Package	% of classes	% of methods	% of lines
Model	60	46	55
View	11	2	2
Controller	18	13	9

The view/controller have some util classes which also have some tests (although InputObserverManagerTest.java is seemingly empty), and that's what being seen here.

The result shows a decent coverage of the model where most of the tests are reasonable. There are exceptions though. Some tests are missing an assert statement while e.g. testDragEvent() in ContinuousPointStrategyTest is a test that is 45 lines long and with 16 asserts. Ideally one would like to have small unit tests with 1 assert per test, and although this might be undesirable in some situations, 16 asserts in one test is too many. On the other hand, all the asserts seem to have debug messages accompanied which justifies it a bit, and makes the tests in general pleasant to work with.

In both the tests and in the actual code base there are a lot of places with empty functions, out-commented code fragments, unused variables/imports and a general lack of code style consistency. Some fields are scattered among the methods rather than at the top of file where you would expect them to be. Some classes have an excessive amount of documentation to the point where it can become tedious and some lack documentation at all.

All these problems indicate that the group hasn't been reviewing each others code and hasn't really been controlling quality/consistency in a good way. The biggest revelation to this conclusion is the fact that 22 of the 103 tests fail. Having lots of tests is a good thing and having tests that fail is still better than having no tests (because now you know about the problems). However, the fact that all of these failing tests have ended up in production is a really bad indication on the QA/CI part. PR:s with failing tests shouldn't be possible to merge into production so they have likely either been pushed to prod directly or merged without any CI nor peer reviews. The project may be manageable in its current state but continuing development like this for another 2 months will cause the code base to slowly become unmaintainable.