Open Logic Gate Simulator

TDA367 Group 1

Jimmy Andersson
Jacob Eriksson
Martin Hilgendorf
Elias Sundqvist

November 22, 2018

Table of Contents

A Note to the Examiners

Some pretty major refactoring work was done during the development of OpenLogicGateSimulator. During the refactoring, moving and editing of files occured within the same commit, which caused files to switch owner (or really, to be deleted from the tree and then re-added as a new file) in the git history. This causes incorrect numbers in gitinspector for the number of added and deleted lines per contributor, as well as the stability percentage.

Requirements and Analysis Document for Open Logic Gate Simulator

Jimmy Andersson
Jacob Eriksson
Martin Hilgendorf
Elias Sundqvist

1 Introduction

A huge benefit of choosing a software based career is that you can work from anywhere as long as you have a laptop and an internet connection. However, when it comes to hardware this is not always the case. In particular, when it comes to building computers it can get expensive real quick. For those students interested in learning how computers work, but do not want to invest in real hardware, there should be an application that helps understand how computers work.

Our approach to the problem is to develop a simulation software called Open Logic Gate Simulator. The purpose of this software is to simulate how low-level logic components work inside of a computer. It will try to emulate the reality close enough for the user to understand it on a logical level, without introducing the complexity of electronics theory to the user. It can be used by both students, teachers and hobbyists to enhance the learning experience, and to get a better intuitive understanding of how logic circuits work.

In practice, this is done by providing an interactive workspace where basic logic gates can be placed, moved and connected. Furthermore, more advanced components (such as clocks) can be placed, and the behaviour of the circuit can be simulated over time, allowing for the construction of more advanced circuits, such as small microprocessors.

The application will be a standalone desktop application with a graphical user interface for the Windows / Mac / Linux environment. It will be scalable and adapt to different screen sizes.

1.1 Definitions, acronyms, and abbreviations

- **Workspace** – The area from which you edit your currently open circuit
- **Circuit** – The set of components and wires shown; essentially the save file
- **Component** – A part of a circuit used to perform operations on a logic state
- **Wire** – A part of a circuit used for propagating a logic state between components
- **Port** – A part of a component that is used to input or output an logic state
- **GUI** – Graphical User Interface
- **MVC** – Model-View-Controller, a design pattern well suited for application with a GUI
- **Java** – The multi-paradigm programming language used to build this application
- **Checkstyle** – An automated tool that help developers stick to a harmonized style of coding
- **PMD** – An automated tool that help developers find suboptimal and unused code
- **SpotBugs** – An automated tool that help developers find potential bugs
- **JaCoCo** – Java Code Coverage, an automated tool that collects information on what code in a code base is covered by the current unit tests

2 Requirements

2.1 User Stories

As a User I want to design and simulate basic logic circuits to get an intuitive sense of what is happening at the lower level of computing.

---

1. As a User I want to see all components, so that I can get an overview of what's available.

- There should be a scrollable list containing all the components.
- Looking at the list, the User should be able to tell what the name of each component is.
- Looking at the list, the User should be able to tell how each component looks.
- The width of the list should be resizable so the User can better adjust it to their liking.
- The components should appear in a sensical order.

1.1. As a User I want to be able to place components on a workspace, so that I can use them.

- There should a big, initially empty, space that will act as the workspace.
- If the user drops the component onto the workspace, the component should stay there.
- If the user drops the component anywhere else, nothing should happen.
- While dragging around the component, the mouse should indicate if the current place is okay to drop the component on.

1.2. As a User I want to be able to get a sense of how big a component actually is and where exactly it will land before placing it, so that I can place components nicely and efficiently.

- If the User drags one of the components in the list towards the workspace, the component visual should follow along the mouse. And the component should be visualized in the same size, and at the same position as it will appear if dropped.

1.3. As a User I want to be able to see details about each available component, so that I can gain insight into what they're doing.

- The User should be able to hover over any component in the list and get a short description of what it does.

2. As a User I want to be able to connect components, so that I can create circuits.

- Each component's ports should be visible so the User can see what can be connected.
- Clicking on 2 ports of opposite types in sequence, should create a wire between them.
- When clicking on the first port in the sequence, all ports of the opposite types that are available to connect to, should highlight. This will make it easier for the User to quickly find ports that are available.
- Pressing ESC or clicking somewhere in the background should cancel the wire creation and thus remove the port highlighting.

2.1. As a User I want to be able to connect one source to several other components so that I can create more compact circuits and minimize unnecessary components.

- Connecting wires from a single output port to several input ports should be possible.

3. As a User I want to get visual feedback from connections that are active (1/on) and ones that are not (0/off), so that I get some live feedback on what is happening in my circuits.

- A connection currently in a HIGH state should be drawn in red.
- A connection currently in a LOW state should be drawn in black.
- Any connection that is in an UNDEFINED state should be marked by drawing it yellow.
- The colors should be updating live as the simulation is happening.

4. As a User I want to be able to remove components, in case I mess up.

- It should be possible to select components and wires by clicking on them, so that the User can choose what to remove.
- It should be possible to select/deselect multiple components/wires by SHIFT-clicking them.
- It should be possible to select an area of stuff to select by dragging the mouse across the background, so that the selection process can be sped up if there are a lot of stuff to select.
- Selecting one or several wires and components and clicking the BACKSPACE or DELETE key should remove them from the workspace.
- Removing a single component should also remove any and all wires that are dependent on the existence of that component.

5. **As a User I want to be able to move around components, so that I can reorganize my circuits as they grow.**

- It should be possible to move around components by dragging them, wires should follow.

6. As a User I want to have a component that gives a signal source, so I can power my circuits.

- A signal source component should be part of the basic components used to provide a circuit with a state to propagate.

6.1. As a User I want to be able to toggle these signal sources, so that I can also use it as a GND source, and interactively toggle it.

- Clicking on a signal source component should toggle its output between HIGH and LOW.

7. As a User I want to have a pulse clock so that I can produce dynamic circuits.

- There should be a pulse clock that produces a 1Hz HIGH/LOW signal as its output.

8. As a User I want to have a access to logic gates such as AND, OR, NOT, XOR, NOR, NAND so that I can perform common logic operations. This is essential for creating any complex and realistic circuit.

- There should be a component for each of these logic gates, ready to use from the component list.

9. **As a User I want to be able to save/load my circuits so that I can continue my work when I have time, and don't have to do everything in one sitting.**

- There should be a menu bar with the options save, save as and load.
- Save and save as should save the currently open workspace.
- Load should open a dialog that asks for a save file to open as a workspace.

10. **As a User I want to have a access to flip-flop latches so I can quickly setup circuits that store various states.**

- There should be SR, T, D and JK flip-flop latches, ready to use from the component list.

11. **As a User I want to have a basic diode component so that I can easily observe binary output states.**

- There should be a diode component that turns bright when it receives a HIGH input value, and should be dark otherwise.

12**. As a User I want to have easy access to multiple workspaces so that I can switch between projects quickly.**

- Each workspace should be contained in a tab at the top of the interface.
- Adding or opening a new workspace should also open a new tab in the editor area, allowing for quick and easy switching between circuits.

13**. As a User I want to be able to start and stop the simulation, so that I can see what is going on in the circuit at a specific point in time.**

- There should be a start/stop button in the graphical user interface, that will start and stop the dynamic components from updating when pressed.

14**. As a User I want to be able to pan across the workspace so that I can utilize more space for my circuits.**

- The workspace area should act as a sort of camera into a bigger workspace, and when the User drags along it while holding down the CTRL modifier, it should pan the position of the camera.

15**. As a User I want to be able to zoom in and out on the workspace to get a more detailed view or an overview of my circuits.**

- Scrolling the mouse wheel while hovering over the workspace area should zoom in/out the focus of the camera.

16**. As a User I want to be able to delay my signals to ensure that some signal reach some destination non-instantaneously.**

- There should be a delay component that delays all state changes passed into it by one second before it outputs that same value.

17**. As a User I want to have segment display components so I that can show numbers.**

- A 7-segment display will give the user the possibility to control the lighting of individual segments on the display by sending a state to each of them.
- A hex display should give the user the possibility to display a hexadecimal number by sending a 4 bit binary number to it.

18. **As a User I want to have access to a basic counter component so that I can conveniently use the segment display.**

- There should be a counter component that increments its count on high flanks and outputs a 4 bit binary mod 16 number.
- The output should overflow and reset to 0b0000 when the counter hits multiples of 16.

19. **As a User I want to have a note component so I that can trigger sound queues.**

- There should be a note component that plays a sound when its single input receives a high flank (I.e. goes from UNDEFINED/LOW -> HIGH).

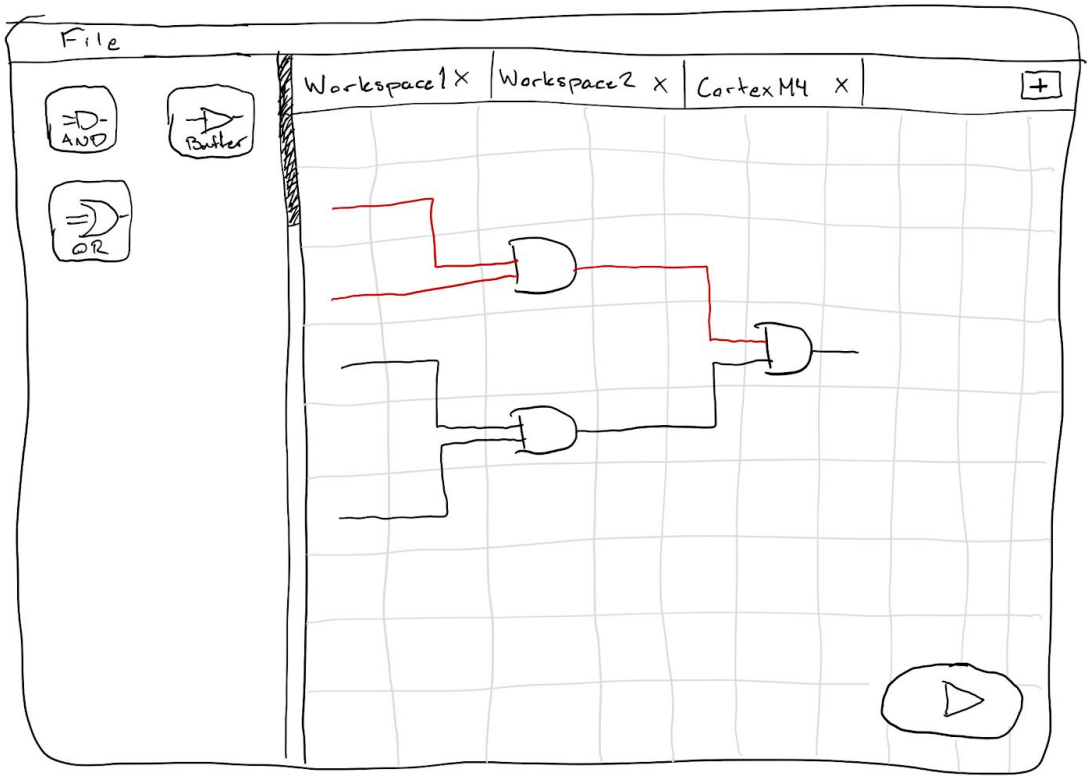
20. **As a User I want the power in my circuits to flow realistically I.e. nearby components and wires should be affected before those far away.**

- The simulation should perform logic state updates in a BFS approach. In layman's term this roughly means that the power needs to expand through circuit, similar to how waves expand from the water surface where a water drop lands.

21. **As a Team we want a proper CI workflow so that all code in the main development branch is ensured to build and follow our desired standards.**

- Setup the main development branch as a protected branch. This way all code added to this branch have to go through PR/CI which means it must follow our requirements.
- Add a Travis setup that builds the project and runs the tests.
- Add Checkstyle to the build.
- Add SpotBugs to the build.
- Add PMD to the build.

2.2 User Interface



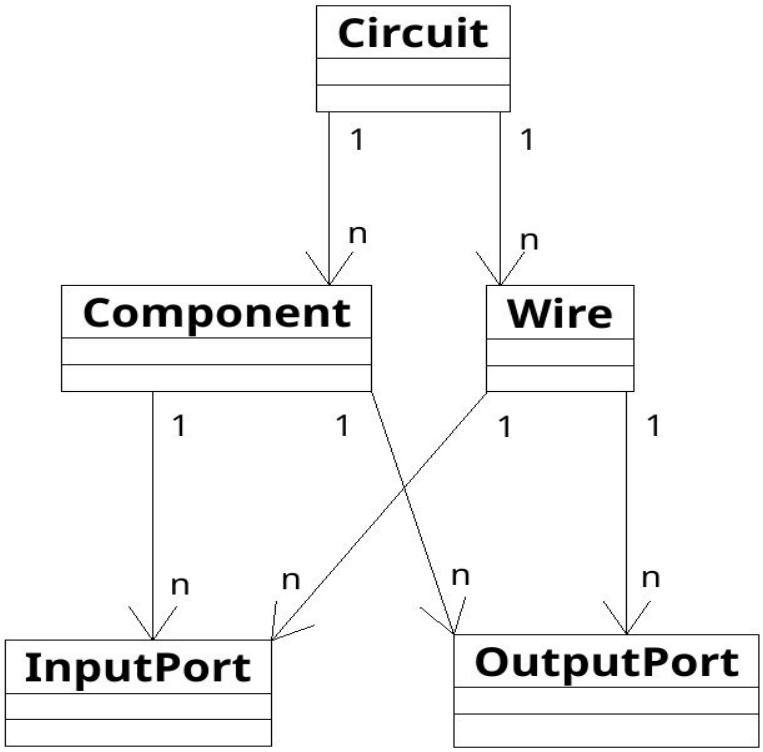
This is the main and only view of the application.

At the top we have a menu bar which allows for saving/loading circuit files and closing the program.

On the left-hand side we have a list of components that can be dragged and dropped onto the circuit.

In the center stage we have the actual circuits (each tab at the top represents a single circuit, and it is possible to switch between). Circuits can be panned/zoomed and you can move components around, as well as connect them together with wires.

3 Domain model



Judging from the domain model it isn't totally obvious how the component and wire actually communicate. We discussed this with our supervisor (after the presentation feedback) and ended up keeping it this way. The options would have been to exclude the ports all together and draw double-arrows between Component and Wire, but this gives false information as the Wire and Component don't actually know about each other. We could have abstracted it to Port but any domain expert would very likely have disagreed as input port and output port serve very different purposes in a real life circuit.

3.1 Class responsibilities

- Circuit: Simulates the power flow between wires and components. Power flow in this context is synonymous with logical state.
- Component: Modifies the logic state of output ports, often based on input ports' logical states.
- Wire: Transmits logic states from all connected output ports to all connected input ports.
- InputPort: Reflects the logic state of the connected wire, without allowing modification to the wire's logic state.
- OutputPort: Affects the logic state of the connected wire.

System Design Document for Open Logic Gate Simulator

Jimmy Andersson
Jacob Eriksson
Martin Hilgendorf
Elias Sundqvist

November 21, 2018
Version 1.1

1 Introduction

This system design document describes the code base and design behind the Open Logic Gate Simulator application. It covers the system architecture and the specific design of every component and how the application handles persistent data storage.

1.1 Definitions, acronyms, and abbreviations

- **Workspace** – The area from which you edit your currently open circuit
- **Circuit** – The set of components and wires shown; essentially the save file
- **Component** – A part of a circuit used to perform operations on a logic state
- **Wire** – A part of a circuit used for propagating a logic state between components
- **Port** – A part of a component that is used to input or output an logic state
- **GUI** – Graphical User Interface
- **MVC** – Model-View-Controller, a design pattern well suited for application with a GUI
- **Java** – The multi-paradigm and multi-platform programming language used to build this application
- **git** – A version control software used to handle revisions of files and keep a detailed history of authors and changes made.
- **GitHub** – An online platform for hosting git repositories and collaboratively developing them.
- **Travis CI** – A free online service with GitHub integration for running continuous integration tasks automatically.
- **Checkstyle** – An automated tool that help developers stick to a harmonized style of coding
- **PMD** – An automated tool that help developers find suboptimal and unused code
- **SpotBugs** – An automated tool that help developers find potential bugs
- **JUnit** – A java library for writing unit tests to assert that certain code functions as it should.
- **JaCoCo** – Java Code Coverage, an automated tool that collects information on what code in a code base is covered by the current unit tests

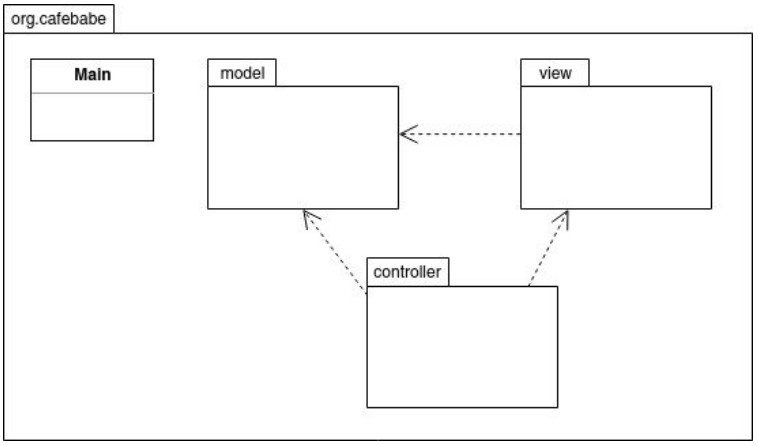
2 System architecture

The software is a desktop application written for a single user per instance, meaning that only a single machine is involved. The system consists of a model, a number of views and a number of controllers. The model is able to exist on its own, while the views keep a reference of the part of the model they need to access to update properly. This way the model can be developed on its own and tested, followed by the views which will update with the model and finally the controller which will make the view interactive.

This is great for separating top level concerns/responsibilities and it makes splitting up workload within the team easier. The views listen to events in the model to ensure it is always notified of state changes. The controller setup listeners for user interaction on the view and updates the model and/or view as necessary. The reason for splitting the view into several views, and similarly with the controller, is for ease of development, small files and separation of concerns.

The events themselves are handled through custom event classes rather than through the typical Observer interface. They are equivalent design wise and our custom events simply bring a few added features, as well as some short and good-looking code.

Here is a top-level overview of the application and how system components interact.
Note that the model does not depend on any other package; it only has incoming arrows from other packages but no outgoing ones. Also note that the view only depends on the model.



Top level overview of the MVC structure.

The system is a single desktop application, so starting and stopping is a trivial task; starting the system is done by executing the jar file, and stopping the system can be done by closing the application window or selecting “Quit” from the file menu in the application.

Adding components to the workspace is done by dragging and dropping any component from the component list into the workspace. When the component is dropped, it is added to the workspace and ready to use. Components that are dropped anywhere else but the workspace are not added, and thus the operation simply cancels. This behavior is indicated by the mouse cursor changing.

Components are loaded in with a plugin approach that was never fully finished, but still brings nice features like being able to add components without editing any existing source code. The directory with components resides under the model, but in reality it could be outside of the project since they are loaded in via reflection. The plan was to allow class files that any user could write to define their own components, and thus extending the program to their liking. The current system should support this ambition, but the feature was never completely implemented due to time constraints.

Some of the components are able to change their state over time. By clicking the start/stop button in the workspace, a user is able to start and stop the simulation of these components. The simulator (if running) is shut down when the workspace is closed, and each workspace has its own simulator. The simulator is implemented as a separate thread, so possible concurrency issues had to be kept in mind when implementing it. The use of events made the communication between the main application and the simulator thread quite simple, because it allows the view to automatically update when the simulator mutates the state of some item.

In order to ensure the quality of the code, the team decided to implement rigorous and thorough checks from early on in the development process. These checks should be run for all code that is pushed to GitHub. To do this, we used the service of Travis CI to automatically run the suite of checks, including Checkstyle, PMD, spotbugs, and JUnit tests. A pull request that does not pass these tests could not be merged. In order to make use some of these tools, they had to be configured. For checkstyle, Google’s checkstyle configuration was used as a base and then adapted to better suit the teams preferences, such as using 4 spaces for indentation rather than 2. The PMD configuration was created from scratch by initially enabling all included checks and then configuring or deactivating those that were deemed irrelevant or overly impractical for the team to adhere to. Next to these automated checks, every pull request required stated approval by at least one other team member before it could be merged. Pull requests were mandatory to propose changes to the development branch, because the branch was marked as protected which disables direct pushes to that branch. Next to the JUnit test suite, the JaCoCo tool was used to see how much of the model subsystems code was actually tested by the test suite.

2.1 Subsystem decomposition

The application follows the MVC pattern which is widely regarded as highly suitable for applications with a graphical user interface. Therefore a natural decomposition into subsystems are the model, the view, and the controller packages.

2.2 Model

The design model is included on the next page. See colored layers there.

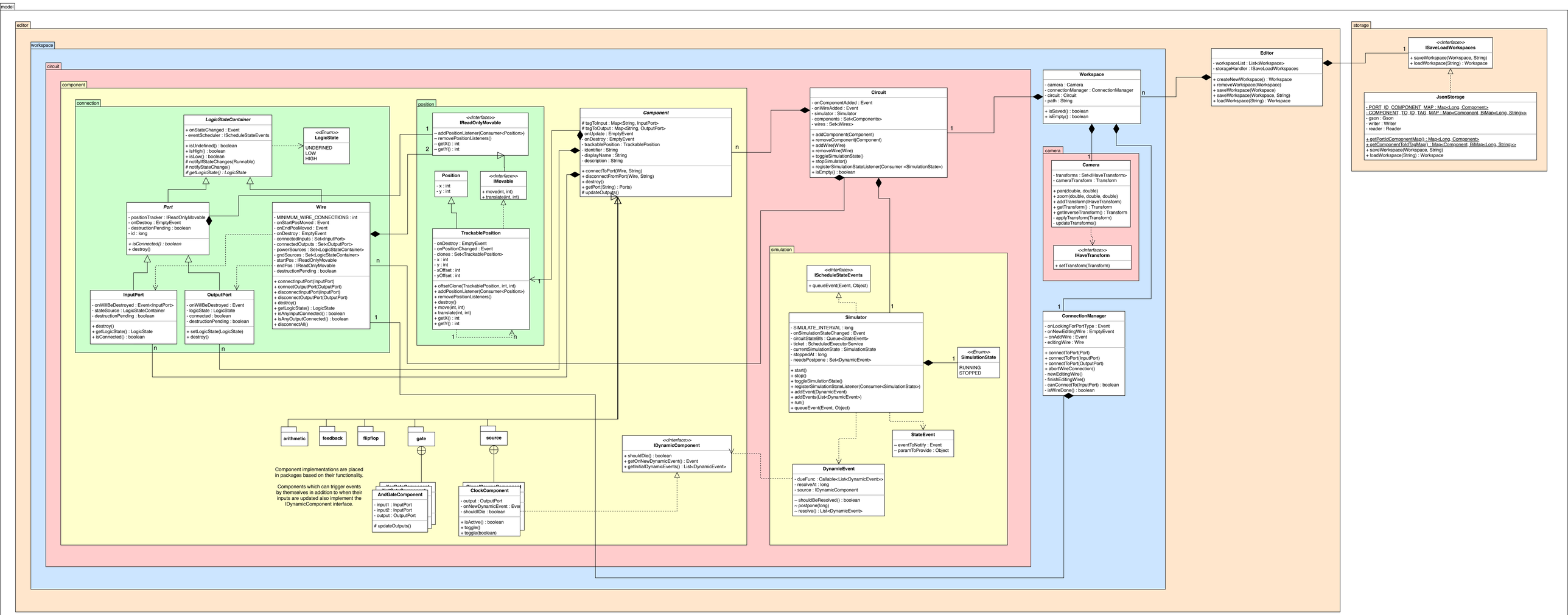
The model subsystem is responsible for storing the users workspaces with the contained circuits, including components and wires. It is also responsible for simulating the signal flow throughout circuits. There are two main subsystems to the model component – the editor subsystem that handles workspaces, and the storage subsystem that handles reading and storing workspaces to and from files (See orange layer).

Further down we have the workspace package (blue layer) which wraps around the circuit package (red layer) and adds features for editing circuits through the connection manager, and for showing subsets of its content through the camera (red layer). The circuit package is split into component and simulation (both yellow layer). The first one handles the content of the circuit and the latter simulates the logic of the circuit i.e. current/signals. Components are connected by the connection package, and their positions are handled through the position package (both green layer).

The model utilizes the Factory Pattern to create its components. It also makes extensive use of the Observer Pattern to update parts of the program through the use of events. The Fail-Fast Pattern/Technique has also been made a part of the model, so that execution will fail quickly and visibly if the program enters a state that it is unable to recover from.
Another pattern that is used to simplify things is the Composite Pattern, for example in the Camera, which will treat views that conform to the IHaveTransform interface as a single unit and make adjustments to them.

Tests for the model package cover functionality and creation of most logic components and connections. They also test the operation of the storage handlers, as well as functionality of the circuits and simulator. All tests are available under OpenLogicGateSimulator/src/test/java.

To let the model code pass CI checks, annotations to suppress warnings are used in a total of 13 places. However, the only one of these that indicates a potential design problem is that Wire.java has too many methods. The other warnings are things like “useless parentheses” according to PMD or “too long lines” according to Checkstyle, which have been consciously suppressed with readability in mind.


model
editor
workspace
circuit
component
connection
LogicStateContainer
+ onStateChanged : Event
- eventScheduler : IScheduleStateEvents
+ isUndefined() : boolean
+ isHigh() : boolean
+ isLow() : boolean
# notifyIfStateChanges(Runnable)
# notifyStateChange()
# getLogicState() : LogicState
<<Enum>>
LogicState
UNDEFINED
LOW
HIGH
Port
- positionTracker : IReadOnlyMovable
- onDestroy : EmptyEvent
- destructionPending : boolean
- id : long
+ isConnected() : boolean
+ destroy()
Wire
- MINIMUM_WIRE_CONNECTIONS : int
- onStartPosMoved : Event
- onEndPosMoved : Event
- onDestroy : EmptyEvent
- connectedInputs : Set<InputPort>
- connectedOutputs : Set<OutputPort>
- powerSources : Set<LogicStateContainer>
- gndSources : Set<LogicStateContainer>
- startPos : IReadOnlyMovable
- endPos : IReadOnlyMovable
- destructionPending : boolean
+ connectInputPort(InputPort)
+ connectOutputPort(OutputPort)
+ disconnectInputPort(InputPort)
+ disconnectOutputPort(OutputPort)
+ destroy()
+ getLogicState() : LogicState
+ isAnyInputConnected() : boolean
+ isAnyOutputConnected() : boolean
+ disconnectAll()
InputPort
- onWillBeDestroyed : Event<InputPort>
- stateSource : LogicStateContainer
- destructionPending : boolean
+ destroy()
+ getLogicState() : LogicState
+ isConnected() : boolean
OutputPort
- onWillBeDestroyed : Event
- logicState : LogicState
- connected : boolean
- destructionPending : boolean
+ setLogicState(LogicState)
+ destroy()
position
<<Interface>>
IReadOnlyMovable
~ addPositionListener(Consumer<Position>)
~ removePositionListeners()
~ getX() : int
~ getY() : int
Position
- x : int
- y : int
<<Interface>>
IMovable
+ move(int, int)
+ translate(int, int)
TrackablePosition
- onDestroy : EmptyEvent
- onPositionChanged : Event
- clones : Set<TrackablePosition>
- x : int
- y : int
- xOffset : int
- yOffset : int
+ offsetClone(TrackablePosition, int, int)
+ addPositionListener(Consumer<Position>)
+ removePositionListeners()
+ destroy()
+ move(int, int)
+ translate(int, int)
+ getX() : int
+ getY() : int
Component
tagToInput : Map<String, InputPort>
tagToOutput : Map<String, OutputPort>
onUpdate : EmptyEvent
onDestroy : EmptyEvent
- trackablePosition : TrackablePosition
- identifier : String
- displayName : String
- description : String
+ connectToPort(Wire, String)
+ disconnectFromPort(Wire, String)
+ destroy()
+ getPort(String) : Ports)
# updateOutputs()
arithmetic
feedback
flipflop
gate
source
Component implementations are placed in packages based on their functionality.
Components which can trigger events by themselves in addition to when their inputs are updated also implement the IDynamicComponent interface.
AndGateComponent
- input1 : InputPort
- input2 : InputPort
- output : OutputPort
updateOutputs()
ClockComponent
- output : OutputPort
- onNewDynamicEvent : Event
- shouldIDie : boolean
+ isActive() : boolean
+ toggle()
+ toggle(boolean)
<<Interface>>
IDynamicComponent
+ shouldDie() : boolean
+ getOnNewDynamicEvent() : Event
+ getInitialDynamicEvents() : List<DynamicEvent>
Circuit
- onComponentAdded : Event
- onWireAdded : Event
- simulator : Simulator
- components : Set<Components>
- wires : Set<Wires>
+ addComponent(Component)
+ removeComponent(Component)
+ addWire(Wire)
+ removeWire(Wire)
+ toggleSimulationState()
+ stopSimulator()
+ registerSimulationStateListener(Consumer <SimulationState>)
+ isEmpty() : boolean
simulation
<<Interface>>
IScheduleStateEvents
+ queueEvent(Event, Object)
Simulator
- SIMULATE_INTERVAL : long
- onSimulationStateChanged : Event
- circuitStateBfs : Queue<StateEvent>
- ticket : ScheduledExecutorService
- currentSimulationState : SimulationState
- stoppedAt : long
- needsPostpone : Set<DynamicEvent>
+ start()
+ stop()
+ toggleSimulationState()
+ registerSimulationStateListener(Consumer<SimulationState>)
+ addEvent(DynamicEvent)
+ addEvents(List<DynamicEvent>)
+ run()
+ queueEvent(Event, Object)
<<Enum>>
SimulationState
RUNNING
STOPPED
StateEvent
~ eventToNotify : Event
~ paramToProvide : Object
DynamicEvent
- dueFunc : Callable<List<DynamicEvent>>
- resolveAt : long
- source : IDynamicComponent
~ shouldBeResolved() : boolean
~ postpone(long)
~ resolve() : List<DynamicEvent>
Workspace
- camera : Camera
- connectionManager : ConnectionManager
- circuit : Circuit
- path : String
+ isSaved() : boolean
+ isEmpty() : boolean
camera
Camera
- transforms : Set<IHaveTransform>
- cameraTransform : Transform
+ pan(double, double)
+ zoom(double, double, double)
+ addTransform(IHaveTransform)
+ getTransform() : Transform
+ getInverseTransform() : Transform
- applyTransform(Transform)
- updateTransforms()
<<Interface>>
IHaveTransform
+ setTransform(Transform)
ConnectionManager
- onLookingForPortType : Event
- onNewEditingWire : EmptyEvent
~ onAddWire : Event
- editingWire : Wire
+ connectToPort(Port)
+ connectToPort(InputPort)
+ connectToPort(OutputPort)
+ abortWireConnection()
- newEditingWire()
- finishEditingWire()
- canConnectTo(InputPort) : boolean
- isWireDone() : boolean
Editor
- workspaceList : List<Workspace>
- storageHandler : ISaveLoadWorkspaces
+ createNewWorkspace() : Workspace
+ removeWorkspace(Workspace)
+ saveWorkspace(Workspace)
+ saveWorkspace(Workspace, String)
+ loadWorkspace(String) : Workspace
storage
<<Interface>>
ISaveLoadWorkspaces
+ saveWorkspace(Workspace, String)
+ loadWorkspace(String) : Workspace
JsonStorage
- PORT_ID_COMPONENT_MAP : Map<Long, Component>
- COMPONENT_TO_ID_TAG_MAP : Map<Component, BiMap<Long, String>>
- gson : Gson
- writer : Writer
- reader : Reader
+ getPortIdComponentMap() : Map<Long, Component>
+ getComponentToIdTagMap() : Map<Component, BiMap<Long, String>>
+ saveWorkspace(Workspace, String)
+ loadWorkspace(String) : Workspace

2.3 View

The view component is responsible for displaying relevant and interesting information to the user. The subcomponents of the view component make use of the Observer Pattern to be informed of changes that are relevant and make the appropriate updates. It also uses the Fail-Fast Pattern/Technique to make sure that invalid states cause the program to fail quickly instead of displaying inaccurate or invalid data.

Most view classes are tailored to represent exactly one object type from the model. Therefore their main dependency become this one model class, and each view class' purpose becomes well defined.

Design issues reported by quality check tools in this package include several false positives where SpotBugs believes that fields annotated with @FXML always will be null, literals in if conditions, and ComponentView.java having too many imports and too many methods. Other less useful warnings include "avoid instantiating objects in loops" in ComponentListView.java, where a loop is necessary to populate the view with children.

2.4 Controller

The controller is responsible for setting up event handlers and making the application interactive. The controller will listen for user interactions with the views and trigger events to update the model or view accordingly.

JavaFX does not support a controller/view separation structure by default so we created our own. Controller.java and View.java outline this separation structure and all controller and view classes need to extend them to be a part of the system. Extending these allows controllers to setup controller classes that will be attached to subviews that the view may add. The attached controllers are, by default, hidden from the controller that attached them to avoid unnecessary relations.

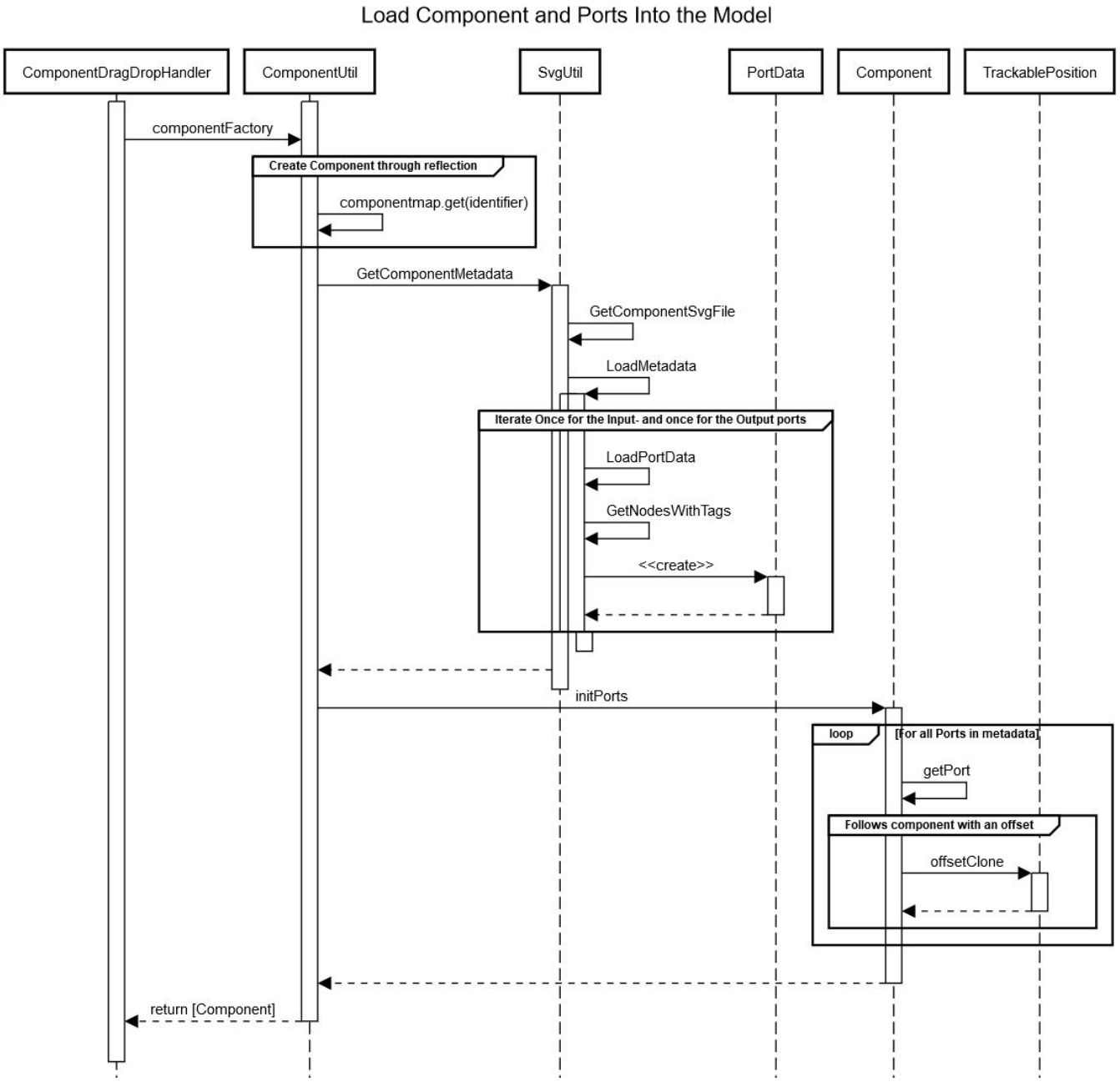
The Delegate Pattern and the Composite Pattern are used for making sure that a user can select different components. The selection itself is handled by a ControllerSelector delegate and a SelectionBox delegate and all of the selected items are treated the same way by conforming to the ISelectable interface.

The controller subsystem has very few design issues reported. The first one is that the abstract class Controller does not have any abstract methods, which disregards the use of the abstract keyword to prevent instantiating classes while still providing a default constructor. Another suppressed warning is that WorkspaceController.java contains too many methods. This class

contains 12 methods, which is reasonable considering its responsibility—it routes any user input on the workspace pane to the delegate class responsible for handling that input.

2.5 Some example flows

All sequence diagrams for the different use cases are available in the repository under OpenLogicGateSimulator/docs/sequence-diagrams.
Some are unfortunately too large to show here. The following diagram does however fit, and it is part of the use case "Spawning a Component onto the Workspace".



2.6 External libraries

net.javainthebox is used to support any type of SVG. JavaFX itself only supports SVG string path, which is very limiting. Supporting SVG is essential to make sure components always look crisp clear regardless of zoom level, and having proper support for the format makes it a lot easier to create your own components.

3 Persistent data management

The application stores two types of data; resources required for running the program, and circuits created by the user. The application resources are bundled with the executable, in the resources directory. The different resources stored here include CSS files for theming, SVG image files for the gates, WAV files for components that emit sound, and FXML files for the visual design of the application.

The user created circuits can be stored anywhere in the filesystem where the user wishes to save them. The location is picked when pressing Ctrl+S on an edit circuit or selecting Save or Save As… from the File menu. The user will then be prompted to specify the location in a FileChooser dialog window. These saved circuits are stored in JSON format with a .olgs file extension.

4 Access control and security

N/A for this application. There is only the role of the “user”, and the application has essentially no interaction with other parts of the host it is running on, nor does it handle any personal data, so security is not a concern for this program.

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.

```
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:

```
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.