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

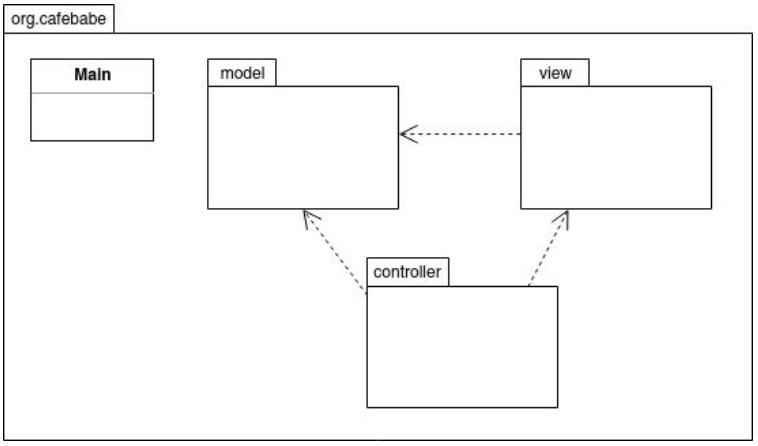
2 System architecture

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.

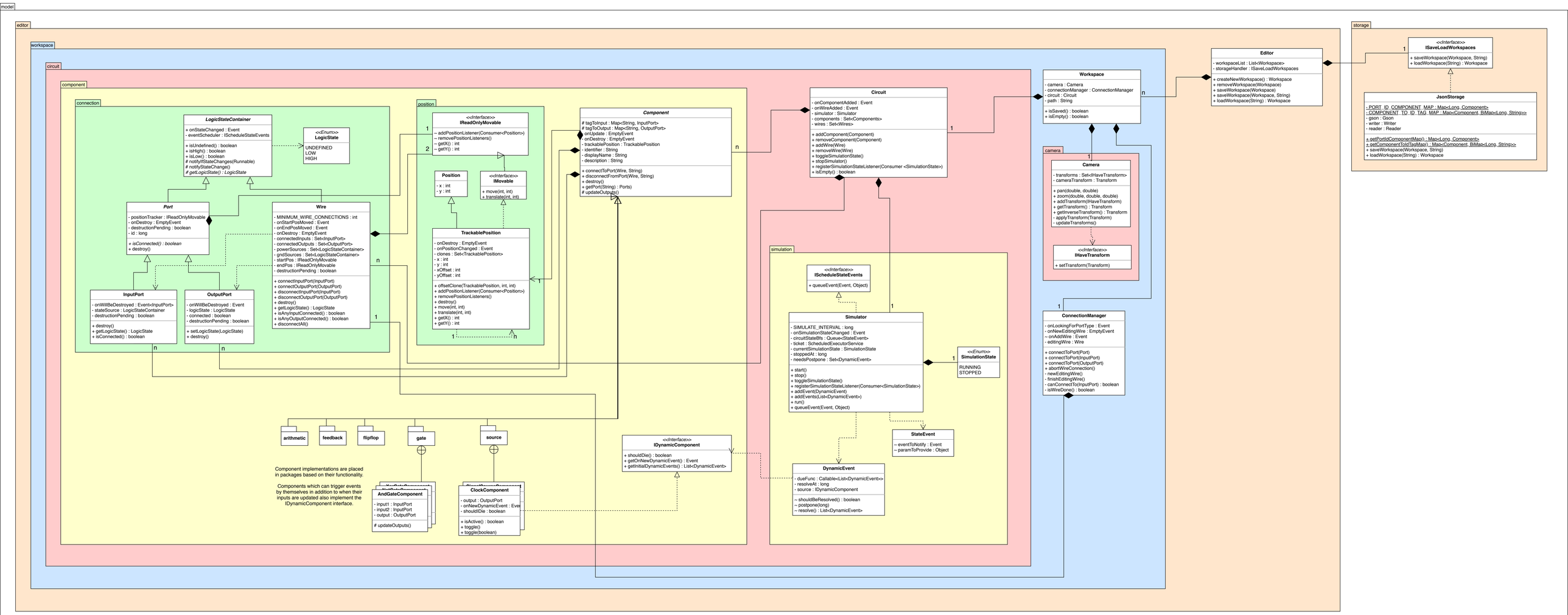
The model subsystem is responsible for storing the users workspaces with the contained circuits, including components and wires. It is also responsible for simulating the signal flow throughout circuits. There are two main subsystems to the model component – the editor subsystem that handles workspaces,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

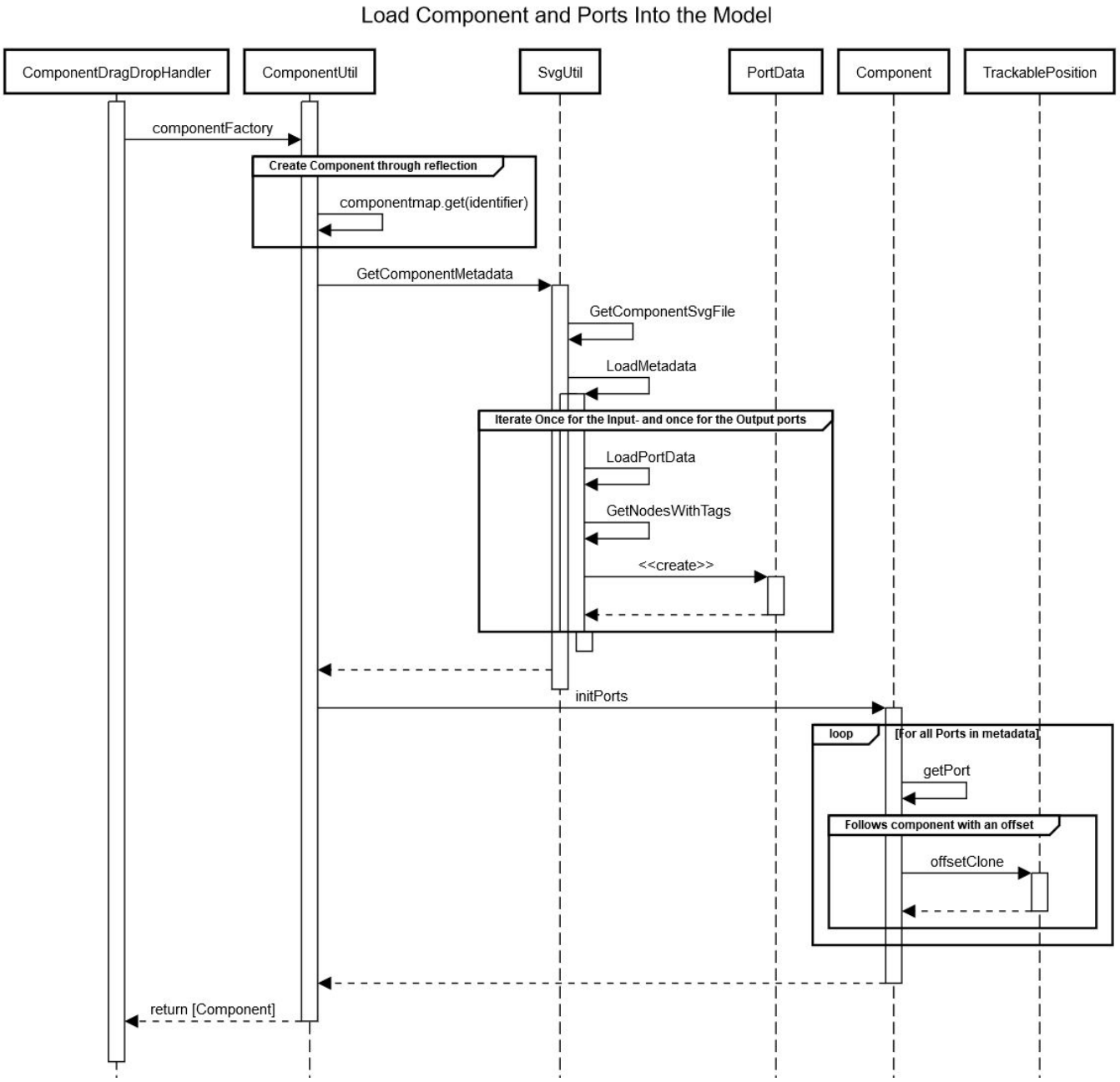
The Delegate Pattern and the Composite Pattern are used for making sure that a user can select different components. The selection itself is handled by a ControllerSelector delegate and a SelectionBox delegate and all of the selected items are treated the same way by conforming to the ISelectable interface.

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.