Requirements and Analysis Document for Winner Is Kungen

Obada Al Khayat Joakim Anderlind Lukas Andersson
Alexander Grönberg Mårten Åsberg

September 2019

Contents

1 Introduction

The project is a Java application for desktops, and this application is a simulator for logic gates where the user can experiment connecting these gates and implementing logical operations.

Other simulators that do the same thing are out there. However, most of these simulators are hard to use or have a poorly designed interface, or both.
So, the primary aim of this project is to have a user-friendly interface and to make the application much simpler to use compared to other simulators like DigiFlisp.

Students who study computer engineering, or anyone interested in learning more about logic gates could benefit from this application.

1.1 Definitions, acronyms, and abbreviations

- **Component:** A logic gate such as AND, OR, NOT.
- **Worspace:** The main layout that contains blueprints.
- **Blueprint:** A "File" that contains components and connections.

2 Requirements

2.1 User Stories

Story Identifier: WIK001

Story Name: The basics of workspace

Description:
As a user I would like to visually see my workspace to be able to work.

Confirmation:

- Functional
    - The application is runnable
    - The user can visually see the layout of the application
    - The program passed the tests

Story Identifier: WIK002

Story Name: View Components.

Description:
As a User I would like to see which components I can use because I want to be able to see what the program can do.

Confirmation:

- Functional
  - I should be able to see all available components in a pallet to the side of the screen

Story Identifier: WIK003

Story Name: Save Blueprints.

Description:
As a user I want to be able to save my circuits because I don't want to start over all the time.

Confirmation:

- Functional
  - I should be able to save a blueprint as a new file.
  - I should be able to overwrite an existing blueprint.
  - I should be able to open existing blueprint files.
  - I should be able to use keyboard commands to save.
  - I should be able to use the menu bar to save.
- Non-functional
  No non-functional criteria.

Story Identifier: WIK004

Story Name: Show the state of in- and outputs.

Description:
As a user, I want to visually see which components are connected together, so I can easily see the circuits.

Confirmation:

- Functional
  - I should be able to determine the state of a input.
  - I should be able to determine the state of a output.
  - The visual representation of the state of in- and outputs should change whenever the result of the simulation changes.
- Non-functional

  - Optimally, users who employ screen readers should be able to determine the state of in- and outputs too.

Story Identifier: WIK005

Story Name: Redo/Undo functionality

Description:
As a user i would like to be able to undo my actions so that i can experiment without repercussions.

Confirmation:

- Functional
  - I should be able to undo any action that should reasonably be able to be undone
  - I should be able to repeat my latest action within reason
  - I should be able to accomplish both of those actions with either a button in the menu bar or a keyboard shortcut
- Non-functional
  - The undo history should be long enough to contain a normal session

Story Identifier: WIK006

Story Name: Resuse blueprints as components

Description:
As a user i would like to use blueprints as containers so that i can A) abstract complicated circuits in bigger projects and B) more easily manage duplicated circuit designs.

Confirmation:

- Functional
  - I should be able to convert any blueprint to a component by designating its inputs and outputs.
  - I should be able to use said component as i would any other component (with exception being using said component in the blueprint it originated from creating an infinitely large circuit, which is currently not supported by this application)
  - Any changes in the original blueprint should propagate into the said component
- Non-functional
  - Blueprint-Components should be able to save(load correctly from save files.

Story Identifier: WIK007

Story Name: Organize/favorites

Description:
As a user i would like to organize components in the palet to improve flow.

Confirmation:

- Functional
  - I should be able to designate components as favorites to allow quick access later
  - All components should be organized into groups (eg inputs, Logic gates, muxers etc)
  - I should be able to quickly find my most used components
- Non-functional
  - All components should be organized into groups

Story Identifier: WIK008

Story Name: Smart connections

Description:
As a user i would like my connections to bend and move automatically so that my circuit looks organized and all of my connections are clearly defined. Confirmation:

- Functional
  - Every connection will take the shortest, non obstructed path to the component. If impossible it will take the path with least amount of bridges over other connectors.
  - Connections will never go over an component.
  - If the connection has to bend to connect two connection points, it will use 90 degree turns to make it there.
- Non-functional

Story Identifier: WIK009

Story Name: Light-up connection points

Description:
As a user i would like to see what each component is outputting for value so that i can more easily debug complicated circuits.

Confirmation:

- Functional

  - When the output of a component is true, its corresponding connection point is highlighted in an appropriate color (strong color, red?).
  - otherwise the connection point returns to its default color.
- Non-functional

Story Identifier: WIK010

Story Name: Replacing components

Description:
As a user i would like to be able to replace components in place, leaving all previous connections in place. Confirmation:

- Functional
  - When the output of a component is true, its corresponding connection point is highlighted in an appropriate color (strong color, red?).
  - otherwise the connection point returns to its default color.
- Non-functional

2.2 Definitions of Done

- All functional requirements should be fulfilled and tested.
- The tests should have approximately 100% code coverage on model.
- The program should still be working.
- JDoc documentation on all classes and functions should be written.

2.3 User Interface

To give the UI a clean look its design is heavily influenced by the default look of JavaFX.

The UI has one view with three parts, all displayed in Figure 1. At the top is a menu bar with the common drop-down buttons. These buttons hold all available actions in the program for easy and comprehensive access.

The rest of the windows is filled by, using the *Canvas Plus Palette* pattern, a palette to the left and a canvas to the right. The palette lists all the different components available to the user, split into separate categories. The canvas is an infinitely large, pannable and draggable blank page. That is, until the user adds something to it. Using the mouse a user can add components from the palette to the canvas, and make connections between them. The components have indicators that show the state, high or low, of their in- and outputs. While editing, the user will see these indicators updated in real-time. Allowing for a quick development cycle.

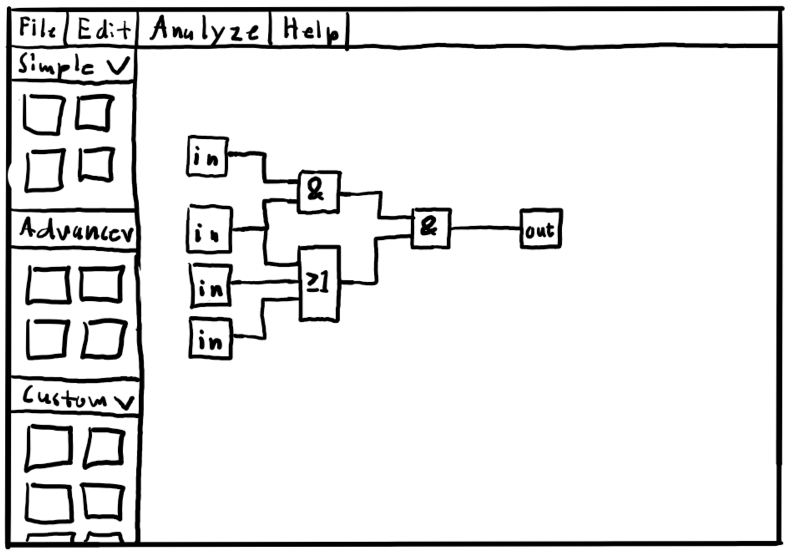

Figure 1: First UI design draft

3 Domain model

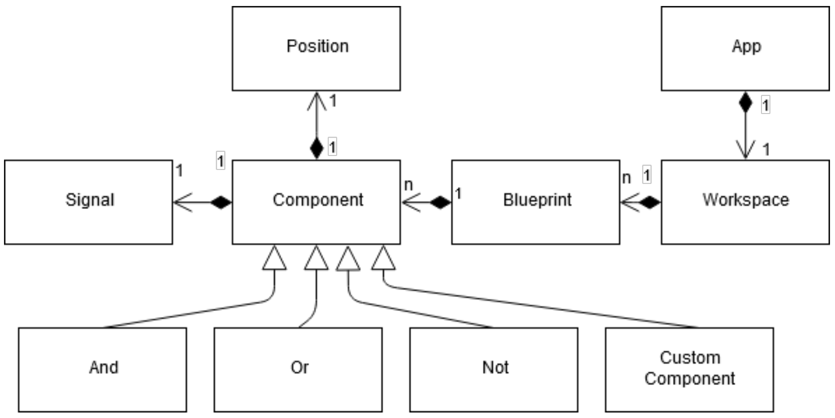

Figure 2: High-level UML diagram of the application

3.1 Class responsibilities

Workspace: The Workspace represents the main layout of the application. Its primary responsibility is to add new Blueprints and manage them.
Blueprint: The blueprint represents the canvas where the user can add Logic components,connect them and maniplulate their inputs.
Component: This class is the core of the application. The Component class represents the components that the user can connect with other components to simulate logical circuits.
Signal: This class takes a value from component and broadcasts the new values to the connected components.
And,Or,Not,... These classes extend from the base Component class and implement the actual logic that differentiates different components from each other. **Position:** The position class is used for keeping track of where components are on the blueprint.
Custom Components: This class is used as a wrapper for using an Blueprint as a Component, thus allowing the user to abstract large and complicated circuits into manageable slices.

System design document for Winner Is Kungen

Obada Al Khayat Joakim Anderlind Lukas Andersson
Alexander Grönberg Mårten Åsberg

September 2019

Contents

1 Introduction

This document will describe the design behind the project, choices that were made along the way and decisions that were crucial on the final product. As well as the different design choices and reasoning behind there will be a full description of the final product with information such as how it works, how its connected, what dependencies it has, what the standard application flow looks like and UML diagram for the top level of the project.

There's also descriptions on how the application will be tested and what criterias the tests have to full fill to be viewed as complete tests, to make sure the tests test all needed aspects of each component properly.

2 System architecture

The application is a standalone Javafx desktop application. It reads in "workspaces" which consists of one or more "Blueprints". Each blueprint is a sandboxed environment where the user can add and connect logic "Components" like And-gates or Switches. If one were to compare it to an IDE like intelj, "workspaces" would be Projects and "Blueprints" would be the files.

When the user activates an input of an "Component", the component calculates an output value and sends it any other component who is connected to it. This creates an domino effect leading to the final result.

To enable the creation of large and complicated projects, a "blueprint" can be used as an "component" as a form of abstraction. This allows for easy duplication of parts(say a bit adder) and a more cleaner project.

When finished the user may save his/her/its "workspace" which can be opened at a later point or shared with other users.

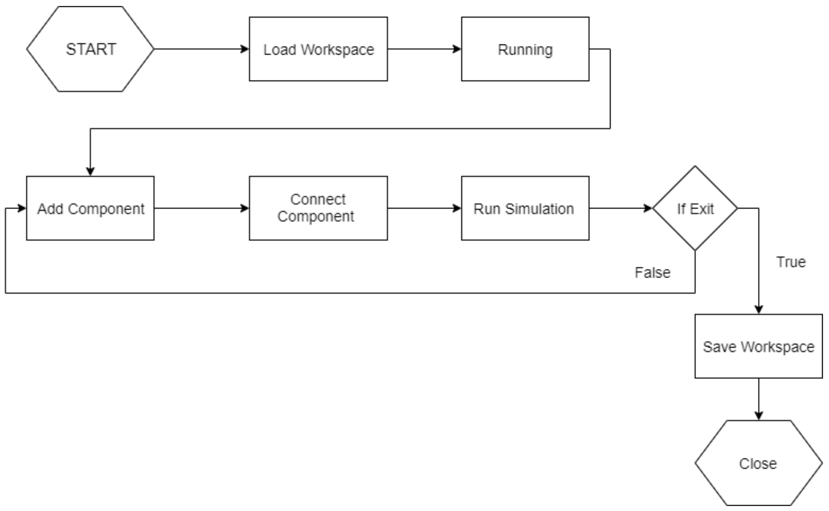

Figure 1: Flow of the application

3 System design

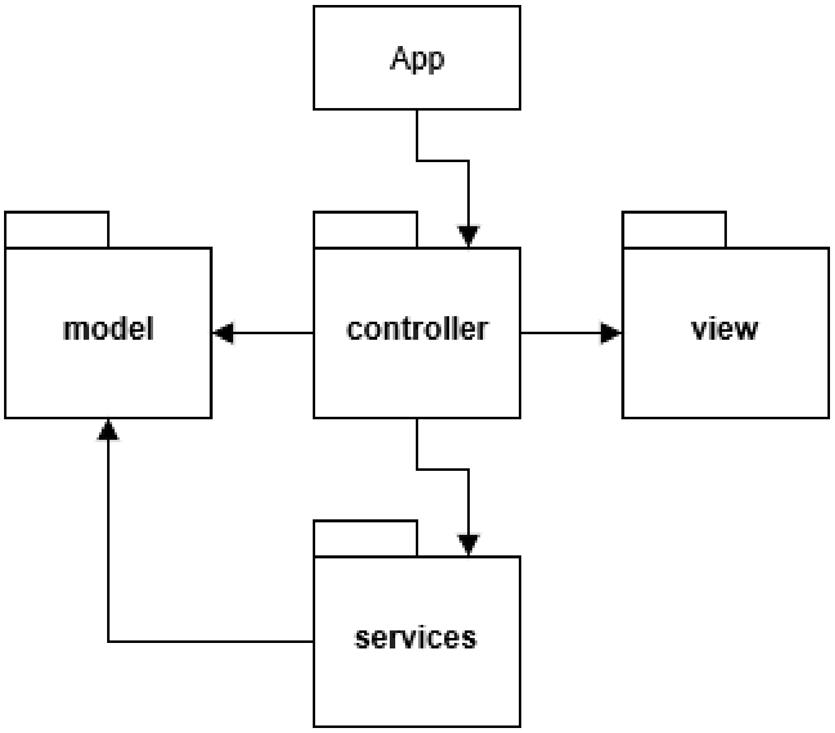

Figure 2: Top level UML of mvc

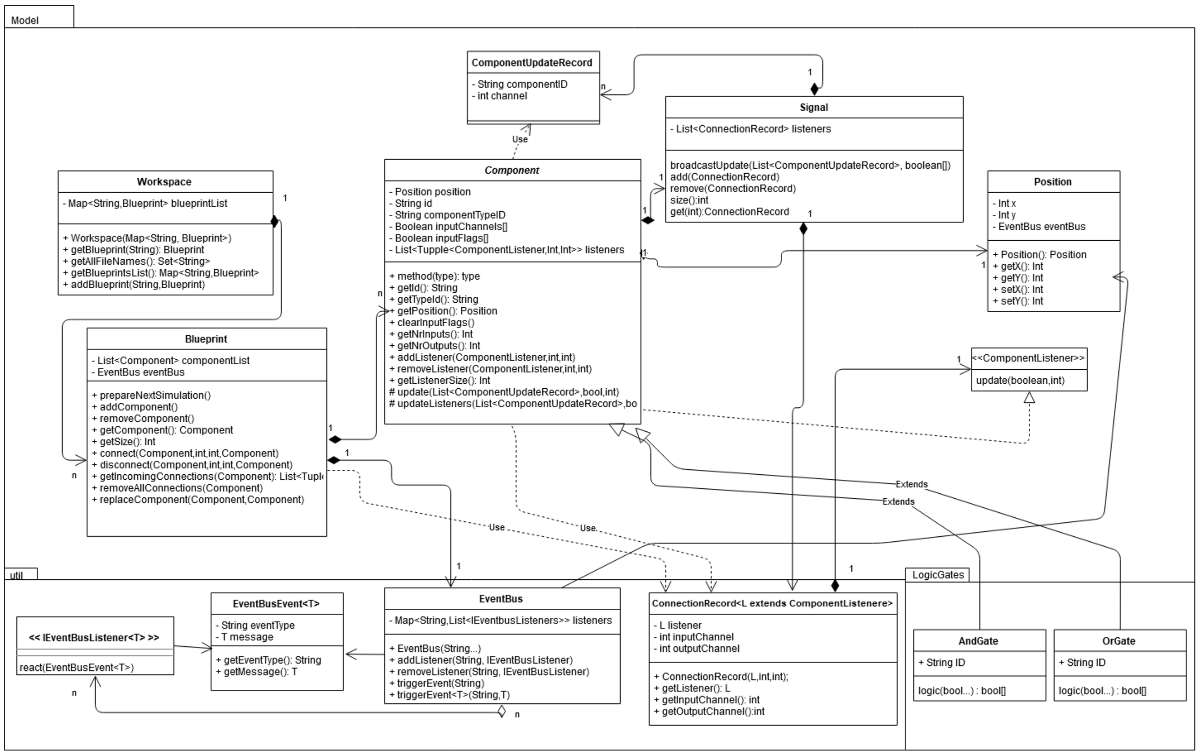
Model
ComponentUpdateRecord
- String componentID
- int channel
Use
Signal
- List<ConnectionRecord> listeners
broadcastUpdate(List<ComponentUpdateRecord>, boolean[])
add(ConnectionRecord)
remove(ConnectionRecord)
size():int
get(int):ConnectionRecord
Workspace
- Map<String,Blueprint> blueprintList
+ Workspace(Map<String, Blueprint>)
+ getBlueprint(String): Blueprint
+ getAllFileNames(): Set<String>
+ getBlueprintsList(): Map<String,Blueprint>
+ addBlueprint(String,Blueprint)
Component
- Position position
- String id
- String componentTypeID
- Boolean inputChannels[]
- Boolean inputFlags[]
- List<Tupple<ComponentListener,Int,Int>> listeners
+ method(type): type
+ getId(): String
+ getTypeId(): String
+ getPosition(): Position
+ clearInputFlags()
+ getNrInputs(): Int
+ getNrOutputs(): Int
+ addListener(ComponentListener,int,int)
+ removeListener(ComponentListener,int,int)
+ getListenerSize(): Int
# update(List<ComponentUpdateRecord>,bool,int)
# updateListeners(List<ComponentUpdateRecord>,bo
Position
- Int x
- Int y
- EventBus eventBus
+ Position(): Position
+ getX(): Int
+ getY(): Int
+ setX(): Int
+ setY(): Int
<<ComponentListener>>
update(boolean,int)
Blueprint
- List<Component> componentList
- EventBus eventBus
+ prepareNextSimulation()
+ addComponent()
+ removeComponent()
+ getComponent(): Component
+ getSize(): Int
+ connect(Component,int,int,Component)
+ disconnect(Component,int,int,Component)
+ getIncomingConnections(Component): List<Tuple
+ removeAllConnections(Component)
+ replaceComponent(Component,Component)
Extends
Extends
Use
Use
1
n
util
<< IEventBusListener<T> >>
react(EventBusEvent<T>)
EventBusEvent<T>
- String eventType
- T message
+ getEventType(): String
+ getMessage(): T
EventBus
- Map<String,List<IEventbusListeners>> listeners
+ EventBus(String...)
+ addListener(String, IEventBusListener)
+ removeListener(String, IEventBusListener)
+ triggerEvent(String)
+ triggerEvent<T>(String,T)
ConnectionRecord<L extends ComponentListenere>
- L listener
- int inputChannel
- int outputChannel
+ ConnectionRecord(L,int,int);
+ getListener(): L
+ getInputChannel(): int
+ getOutputChannel():int
LogicGates
AndGate
+ String ID
logic(bool...) : bool[]
OrGate
+ String ID
logic(bool...) : bool[]

Figure 3: UML of Model

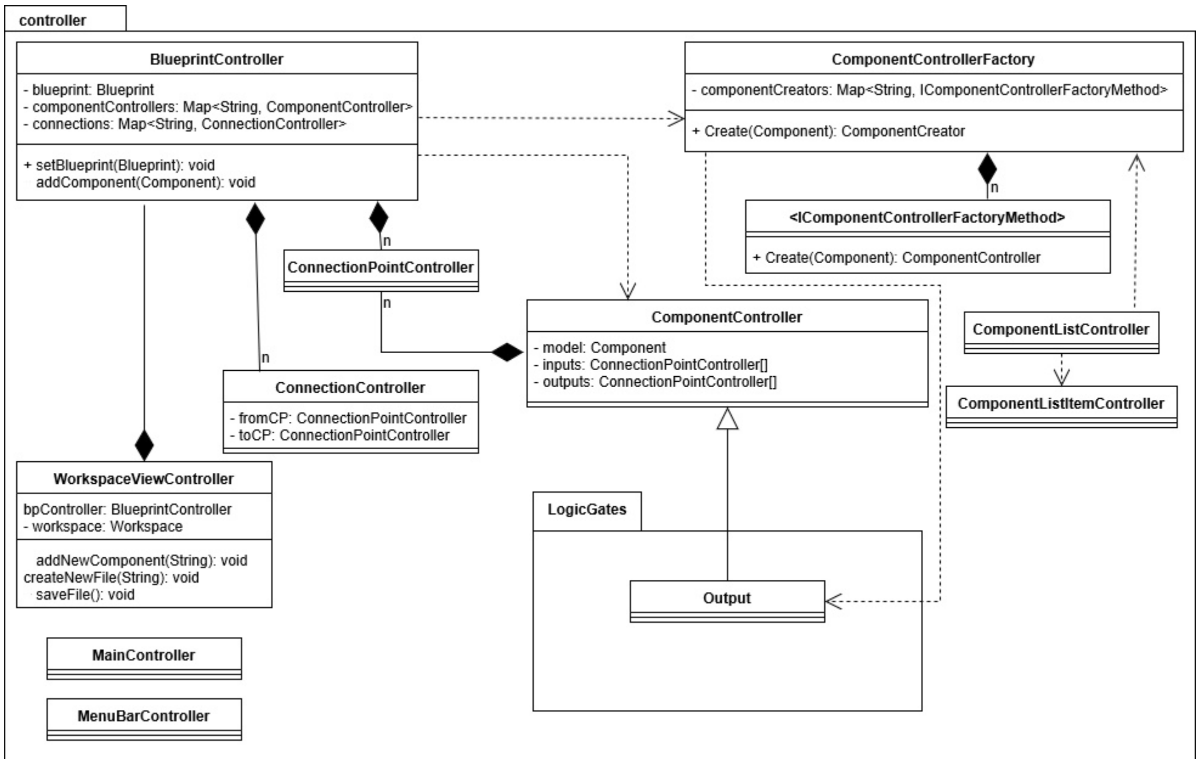
controller
BlueprintController
- blueprint: Blueprint
- componentControllers: Map<String, ComponentController>
- connections: Map<String, ConnectionController>
+ setBlueprint(Blueprint): void
addComponent(Component): void
ComponentControllerFactory
- componentCreators: Map<String, IComponentControllerFactoryMethod>
+ Create(Component): ComponentCreator
<IComponentControllerFactoryMethod>
+ Create(Component): ComponentController
ConnectionPointController
ComponentController
- model: Component
- inputs: ConnectionPointController[]
- outputs: ConnectionPointController[]
ConnectionController
- fromCP: ConnectionPointController
- toCP: ConnectionPointController
ComponentListController
ComponentListItemController
WorkspaceViewController
bpController: BlueprintController
- workspace: Workspace
addNewComponent(String): void
createNewFile(String): void
saveFile(): void
LogicGates
Output
MainController
MenuBarController
n
n
n
n

Figure 4: UML of Controller

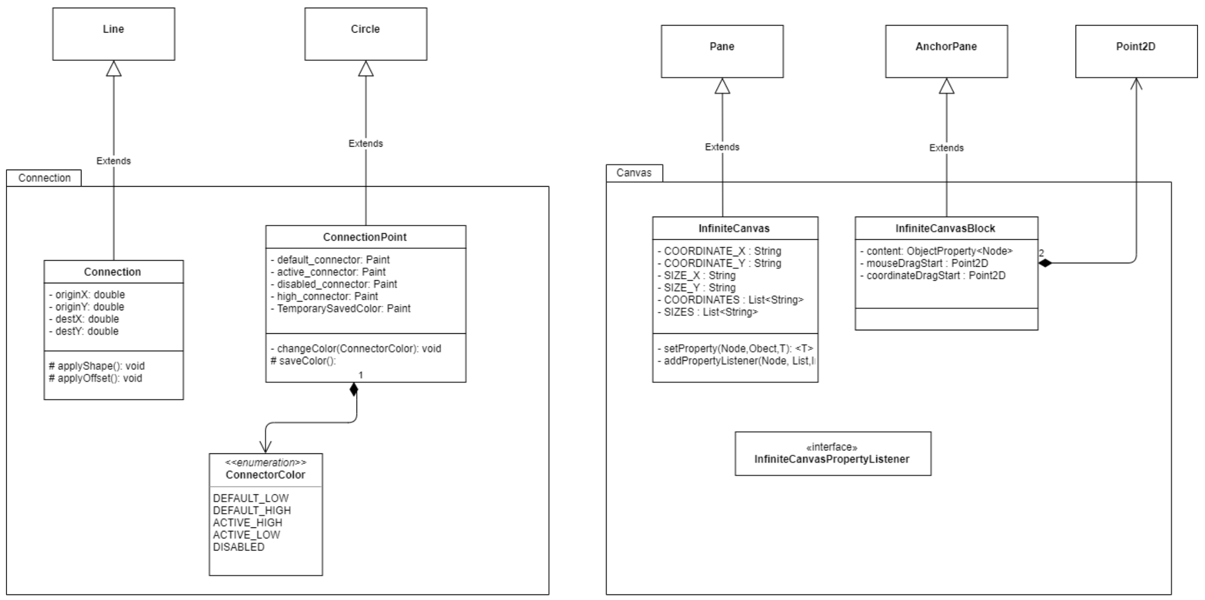

Figure 5: UML of View

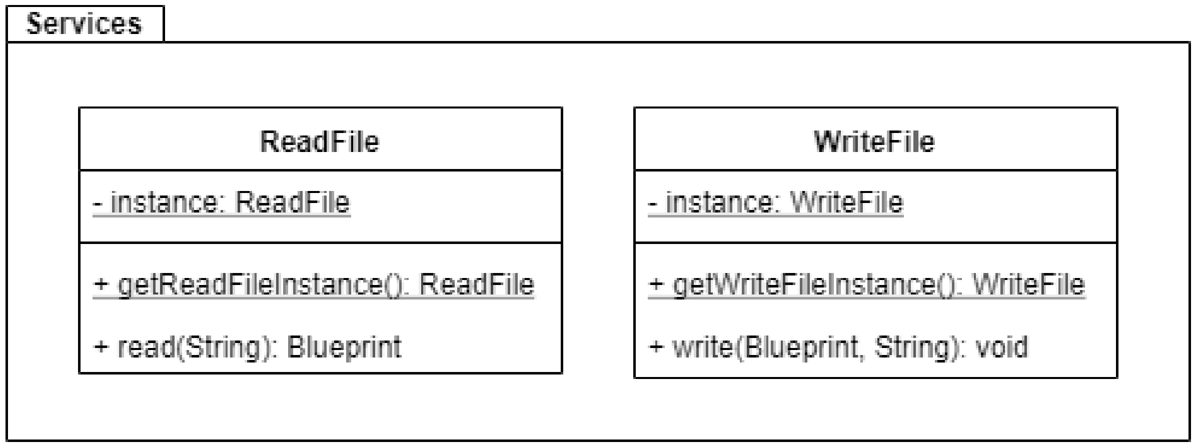
Services
ReadFile
- instance: ReadFile
+ getReadFileInstance(): ReadFile
+ read(String): Blueprint
WriteFile
- instance: WriteFile
+ getWriteFileInstance(): WriteFile
+ write(Blueprint, String): void

Figure 6: UML of Services

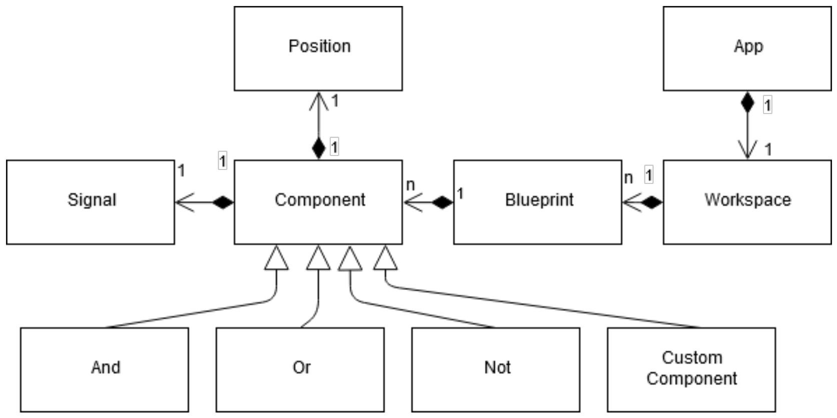

Figure 7: Domain model

The application is divided into five parts, the model, view, controller,services, and app. The model houses all of the logic and does not depend on any other packages. The view package consists of custom nodes that extend from javafx nodes. this package does not depend on the model nor controller. The controller package houses classes that bind the model and the view together. Finally the app starts the program.

There is a great similarity between the domain model and the model.The only classes not featured in the domain model are utility classes needed for sending events to observers, ConnectionRecord for storing information about connections and ComponentUpdateRecord for feedback loops.

The model heavily uses Observer patterns to send messages between classes. Each component acts as a listener to updates from other components and the EventBus is used to inform the controller of events in the model (like the creation of a new component). The model also uses an template pattern by making child classes of components implement logic to improve code reuse.

In the controller package we use a Factory pattern to collect the creation of all Component Controllers into one class.

4 Persistent data management

This application as able to save blueprint files (.dfbp) which are files with info representing one blueprint and all info it contains, such as components and connections. They can later be loaded as blueprints in the application.

5 Quality

5.1 Tests

The aim of the project is to have 100 % test coverage for the model, while having the mindset that tests for other parts are great but not required.

Files for the tests are located in a parallel folder named ”test” with a similar folder structure as the original source code directory.

5.2 Known issues

- Connection lines do not scale when zooming.
- When making a faulty connection the ”connection making state” gets canceled, but the color of the connections aren’t canceled.
- When switching between tabs color the output of some components disappear, just visually however, the simulation still works.

5.3 Analysis

5.3.1 PDM

PDM told us that we should override the hashCode method in all classes where we overrode the equals method. That was easily fixed, and when running PDM on the final state of the project PDM does not complain about anything.

Code Review

Do the design and implementation follow design principles?

Yes, to some extent.

Does the project use a consistent coding style?

Very consistent. All field and variable names are in camelCase, and all class/interface/enum names are in PascalCase. All classes follow the pattern of `static fields`, `instance fields`, `constructor`, `static methods`, and `instance methods`.

Is the code reusable?

A lot of the model is `protected`, `package-private` or `private`, preventing others from extending the model in other projects. Meaning no reusability of the model from the outside.

However, inside the model there is a great deal of possible (and realized) reusability, with many interfaces and abstract classes.

Is it easy to maintain?

Yes. With most files being around or under 150 lines, somewhat documented in code, and few unexpected dependencies the model looks to be relatively easy to maintain.

Can we easily add/remove functionality?

For the most part, yes.

One suggestion: Use maps instead place of `switch`-statements. Less boilerplate when adding/removing options. Some could map to values and some could map to lambda functions.

Are design patterns used?

Nice method chaining with the Vector class.

Yes, some.

Is the code documented?

Somewhat. Many methods lack comments completely, and some only have comments for specific lines. And other are complete with nice JDoc comments.

Are proper names used?

All names consistently use camelCase, which is nice. The content of names is also consistent, and almost all are relevant and understandable.

Is the design modular? Are there any unnecessary dependencies?

Very modular design ▯

One thing that can be improved is the models dependency on the different classes in the `services` package. Some classes, such as the `collisions` package, could probably be moved into the model. And some, such as the `LevelLoader`, could probably benefit from being a interface in the model and then having the implementation be injected into the model. That would allow for different implementations of file format for levels.
The result of all of this would be a completely separated model that could be dropped into other projects.

Does the code use proper abstractions?

Yes. Good use of interfaces and abstract classes to be able to re-use code.

Is the code well tested?

About 50 % of the code base is run tests. Although, many tests tets the implementation of methods, not there result/effect.

Are there any security problems, are there any performance issues?

Didn't find any. Also didn't look.

Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?

The code is as easily understood as can be. It's a lot of code, but not to difficult to wrap your mind around.

It do have an MVC structure that fits nicely with the used framework.

The model is not completely separated form the rest, but it is separated from the other parts of the MVC structure.

Can the design or code be improved? Are there better solutions?

Everything can always be improved. Especially a work in progress.

Document Review

In your system architecture uml, you have included dependencies the packages have with themselves. This removes focus from whats important in the graph, the relationships between the packages.

Include your user stories in the RAD document, not just a link to where to find them. If you feel that the user stories take up too much space you can some, or event most, in the appendix.

All sections should contain text, some of yours only contain subsections and some are empty.

Some spelling and grammatical mistakes, for example: some "it's" where it should have been "its".