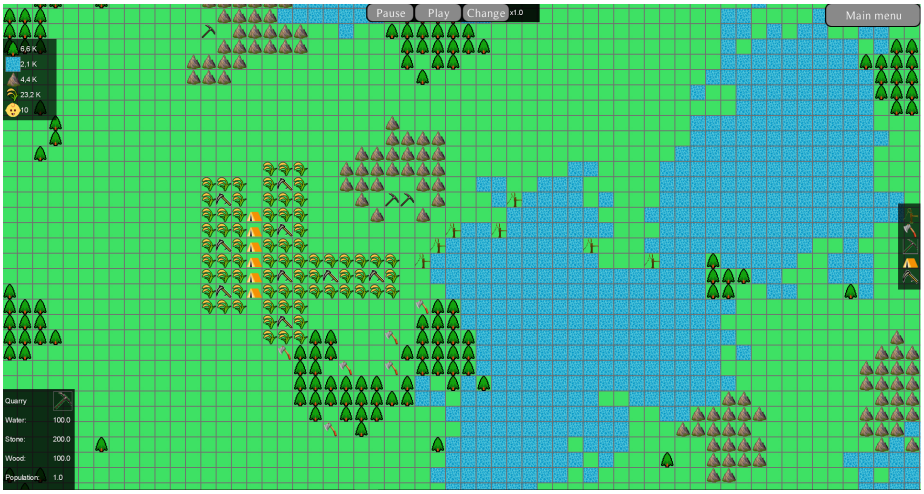
TDA367-Report

Erik Jergéus, Eric Carlsson, Jacob Pedersen,
Theodor Angergård, Vidar Magnusson

October 28, 2018



Grupp 2 - TEVEJ

Contents

1 Introduction

The project is a simulation/strategy game where the user can build buildings, gather resources and then use the resources to progress in the game. The user plays the game from a top down god-perspective with the graphics being two dimensional.

1.1 Definition, acronyms, and abbreviations

- **World** The world refers to a grid of tiles that makes up the game.
- **Inventory** The inventory is the user's collection of resources that can be used to construct new buildings or feed the population.
- **Building** A building is something that the user can place into the world. It may either be a resource gathering building, e.g. a mining building that gathers stone, or it might be a home building, a place for your population to live.
- **Resource** A resource is something the user can use to build buildings or feed the citizens. E.g. Wood, Stone, Food and the available Population.
- **Natural Resource** Natural resources are objects in the world that can be gathered by placing buildings near them. When they are gathered resources are added to the user's inventory.
- **Tile** The world has a tile-based structure, i.e. a grid. Each tile may hold a building, a natural resource or be empty.
- **Ashley** Ashley is a Java Entity, Component System framework that the model depends on.
- **Engine** A central hub for the model's information imported from Ashley.
- **Signal** A small piece of information containing an entity and a signal type. This is used in some parts of the model to broadcast information to other parts of it.
- **GUI** GUI stands for Graphical User Interface.
- **UI** UI stands for User Interface.
- **Camera** The part of the world that is currently visible to the player.

2 Requirements

2.1 User stories

Epic
Description

As a user I want to be able to play a builder/simulation game.

Confirmation

- Start the application.
- See something on the screen.
- See the world.
- Place buildings.
- Gather resources.

2.1.1 Finished user stories

User story

Story ID: 0

Story Name: Start application

Description

As a user I need to be able to start the application.

Confirmation

- Can I run the application.

User story

Story ID: 1

Story Name: See the world

Description

As a user I want to be able to see the world.

Confirmation

- Does a world exist?
- Does tiles exist?
- Does the world have a large amount of tiles in a grid?
- Can I see the tiles?

User story

Story ID: 2

Story Name: See resources

Description

As a user I want to see resources in the world.

Confirmation

- Does resources exist?
- Can tiles have resources on them?
- Can the resource be seen as they are supposed to be?

User story

Story ID: 3

Story Name: See buildings

Description

As a user I want to see buildings.

Confirmation

- Does buildings exist?

- Can tiles have buildings on them?
- Can the buildings be seen as they are supposed to be?

User story
Story ID: 4
Story Name: Navigate the world
Description
As a user I want to be able to navigate the world.
Confirmation
- Can inputs be registered?
- Can I move the camera by dragging my mouse in the world?

User story
Story ID: 5
Story Name: Home building
Description
As a user I want to there to be a Home building.
Confirmation
- Can I have a home building?
- Can I have population in my home?

User story
Story ID: 6
Story Name: Start with home
Description
As a user I want to start with a home.
Confirmation
- Do I start with a home on a tile?

User story
Story ID: 7
Story Name: Inventory
Description
As a user I want to have an Inventory.
Confirmation
- Can I have an inventory?
- Can I see my inventory?
- Can my buildings cost and generate resources in my inventory?

User story
Story ID: 8
Story Name: Deplete Resources
Description
As a user I want some resources to be depletable.
Confirmation
- Does finite resources exist?

- Can I not gather any more resources when it is empty?
- If I deplete a resource, is it deleted?

User story
Story ID: 9
Story Name: Place buildings
Description
As a user I want to be able to place buildings in the world.
Confirmation
- Can I place a building where my mouse is?

User story
Story ID: 10
Story Name: Cost of Buildings
Description
As a user I want buildings to cost resources.
Confirmation
- Can I only build a building if I have enough resources?
- If I build a building, are the proper amount of resources removed from my inventory?
- Can I see that buildings cost resources?

User story
Story ID: 11
Story Name: See buildings to build
Description
As a user I want to see what buildings I can build.
Confirmation
- Can I see the available buildings?
- Can I see the buildings I can not afford?

User story
Story ID: 12
Story Name: What building I am placing
Description
As a user I want to know what building I am placing.
Confirmation
- Can I see buildings next to the mouse when placing?

User story
Story ID: 13
Story Name: Zooming
Description
As a user I want to be able to zoom in and out.
Confirmation
- Can I zoom in?

- Can I zoom out?
- Can I still do the same things when zoomed in/out?

User story
Story ID: 14
Story Name: Growing trees
Description
As a user I want trees to be able to grow by themselves.
Confirmation
- Can trees grow by themselves?

User story
Story ID: 15
Story Name: Population
Description
As a user I want population to exist.
Confirmation
- Can I only build a building if I have population to spare?
- Does the population eat food?
- Does available population die if you run out of food?

User story
Story ID: 16
Story Name: Save game
Description
As a user I want to be able to save my game.
Confirmation
- Can I resume my game after exiting?

User story
Story ID: 17
Story Name: Main menu
Description
As a user I want to be greeted with a main menu.
Confirmation
- When I start the application, am I greeted with a main menu?
- Can I decide to play a new game from the main menu?
- Can I decide to resume my last game from the main menu?
- Can I reach the main menu from the game?

User story
Story ID: 18
Story Name: Time Controller
Description
As a user I want to be able to change the speed of the game.
Confirmation

- Can I increase the speed of my game?
- Can I resume regular speed?
- Can I pause the game?

2.1.2 Incomplete user stories

User story
Story ID: 19
Story Name: Pliancy when placing buildings
Description
As a user I want to know if I can place buildings.
Confirmation
- Can I see if a spot is unavailable for placing?
- Can I easily see if I have enough resources?

User story
Story ID: 20
Story Name: Upgrade buildings
Description
As a user I want to be able to upgrade buildings.
Confirmation
- Can I see a GUI for upgrading buildings?
- Can I see that a building is upgraded?
- If I upgrade a building, does it gain functionality?

2.2 User interface

2.2.1 Main user interface

Figure 1: The initial sketch of the UI.

The initial user interface sketch consisted of four basic elements: One to show the inventory, one to show information about buildings, one to construct buildings and one to control the time.

- The "Inventory" UI exists to show the user what they have in their inventory. That way they will know if they have enough resources to keep their population alive or if they have enough to build a specific building. It was designed to be simple but informative by using both images and an amount that updates in real-time, so that the user always knows what they have in their inventory. The numbers telling the user how much of each resource that remains have suffixes for thousands (K) and millions (M) to avoid filling up the screen with long numbers.
- The "Building Information" UI shows the user what a building is named and how much it costs. It also shows an image of the building to help the user associate it with an actual building on the map. It pops up whenever the user either selects a building from the "Construct Building" UI or clicks on an existing building on the map.

- The "Construct Building" UI shows the user all of the different buildings that are available for construction. If the user clicks on one of the building-icons the icon will then follow the cursor until the user clicks on an empty tile on the map or deselects the building by pressing the icon again. If the user clicks on an empty tile the selected building will be built on the tile that was clicked.
- The "Time control" UI lets the user change the time-speed and also shows the user what the current time-speed is.

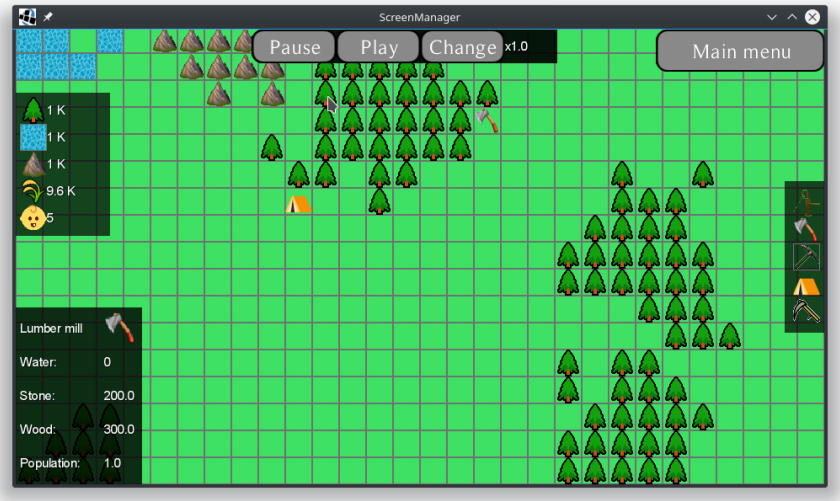


Figure 2: This was the final result of the main UI.

During the development process the UI changed in several ways. The UI element representing the inventory was moved from the top to the top-left corner. This change was mostly made to make the UI more static even when the text gets wider. Another difference between the sketch and the result is the size of the "Construct Building" UI-element on the left side. It is only one column instead of two and it is not as tall as in the sketch. This is a side-effect of the fact that the game does not include as many buildings to choose from as first intended. The "Main menu"-button in the top-right corner of the screen was also added to the UI. It is a simple button that takes the user to the main menu of the game.

2.2.2 Menu user interface

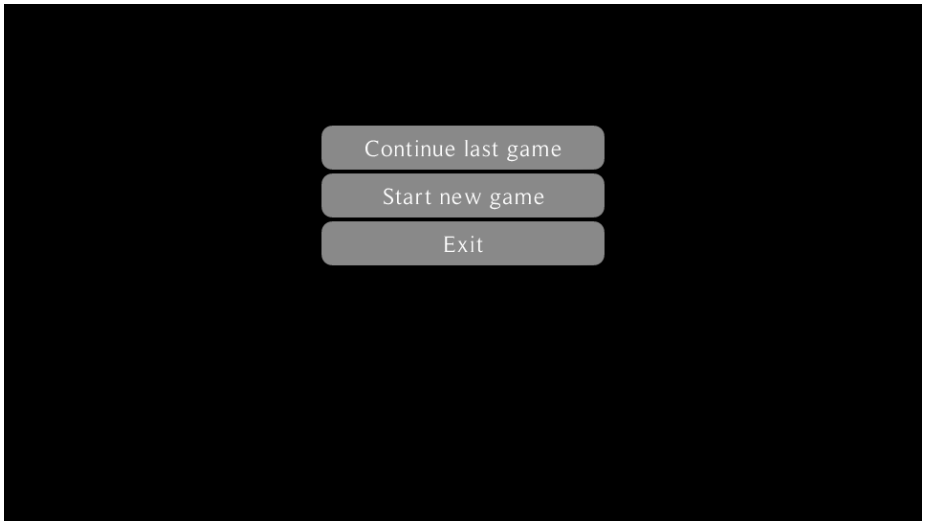


Figure 3: The "Menu" interface

The "Menu" user interface was added to the game to let the user choose whether to:

- **"Continue last game"** - Loads the save-file from the last time the user played the game.
- **"Start new game"** - Begins a new game and delete the previous save-file.
- **"Exit"** - Saves the user's progress and exits the application.

This user interface uses simple buttons to display the user's different alternatives.

3 Domain model

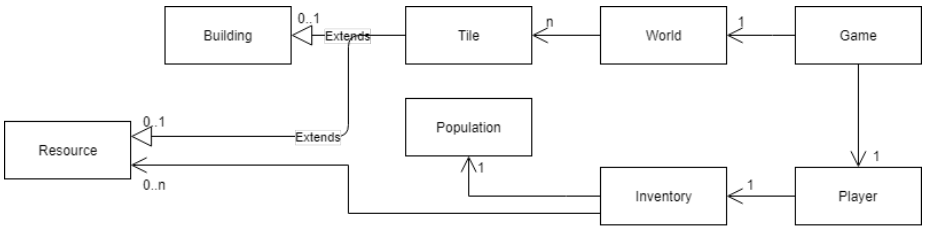


Figure 4: The domain model.

3.1 Class responsibilities

Game: Class that initializes and runs the application.

World: A logical representation of all objects in the world that structures them for convenient use.

Tile: Represents a location on the map.

Building: One of the things the user can place on tiles to build their world.

Resource: Represents the cost for buildings and the amount the user can extract from natural resources such as Trees and Lakes.

Inventory: Represents the current amount of resources the user has at their disposal.

4 System architecture

4.0.1 MVC

In the application is implemented by having the controller initialize the model and the view. After initialization the controller contains the main game loop and updates the model and view continuously. The controller also listens to the view for user input via observer patterns, after receiving a user input the controller updates itself, the model and the view accordingly. The view in turn listens to the model and updates the graphics/user interface to properly reflect the model.

4.0.2 Entity Component System

The model uses Entity Component System (ECS), which favors composition over inheritance.

- **Component** - Components are thin classes that generally only contain data and little to no logic. For example:
  - PositionComponent has two coordinates (x and y).
  - SizeComponent has a width and a height.
- **Entity** - Entities are classes that holds components and represents the objects in the game. An entity is defined by the components it holds, for example the Home building has the components PositionComponent, SizeComponent, BuildingComponent and HomeComponent.
- **System** - The systems manipulates the components data and constitutes most of the game logic. The systems are either updated with the game

loop for example, the NaturalResourceGatheringSystem, that collects resources for the gathering buildings from nearby natural resources. Another way the systems are updated are through signals, that work like global listeners where interested systems listens to specific signals and are then updated accordingly. An example of a system updated through a signal is the BuildBuildingSystem which receives a signal to build a building at a specific location.

4.1 Subsystem decomposition

Not applicable because the application only has one system component.

4.2 The Design Model

The design model of the project has been split up into different parts based on the packages of the application. This is to make them easier to understand and maintain as it might otherwise become relatively large as the application consists of around a hundred java files.

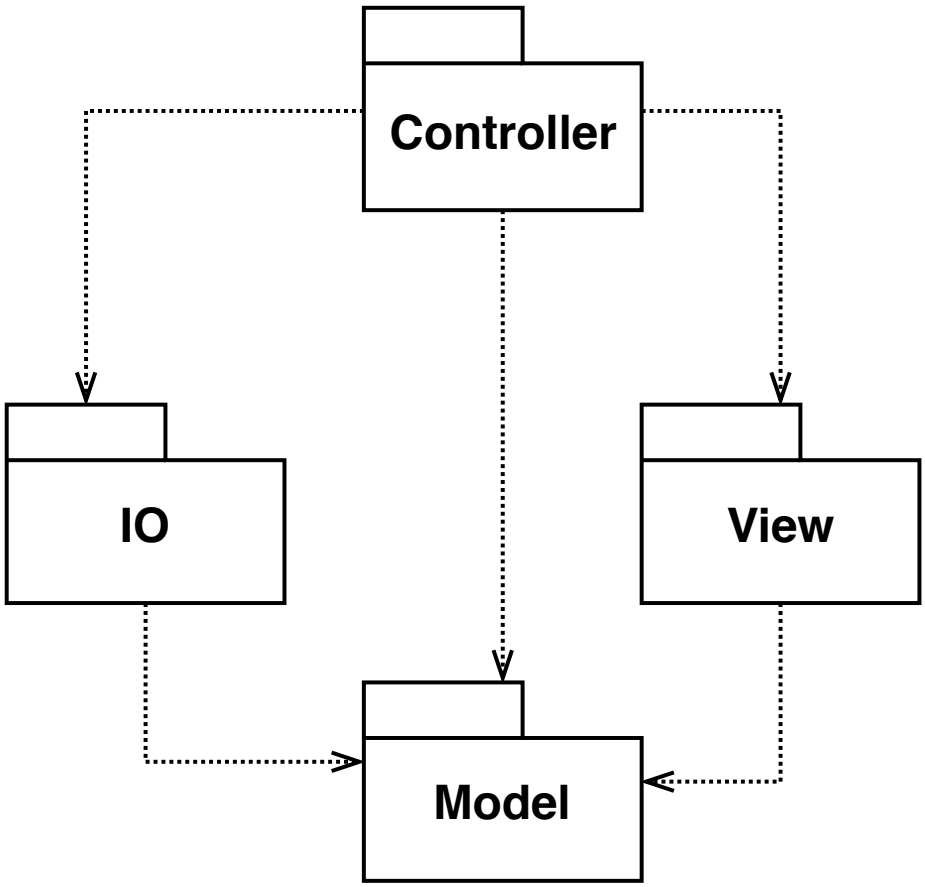


Figure 5: The most top-level package, following a relatively standard MVC format.

4.2.1 Controller

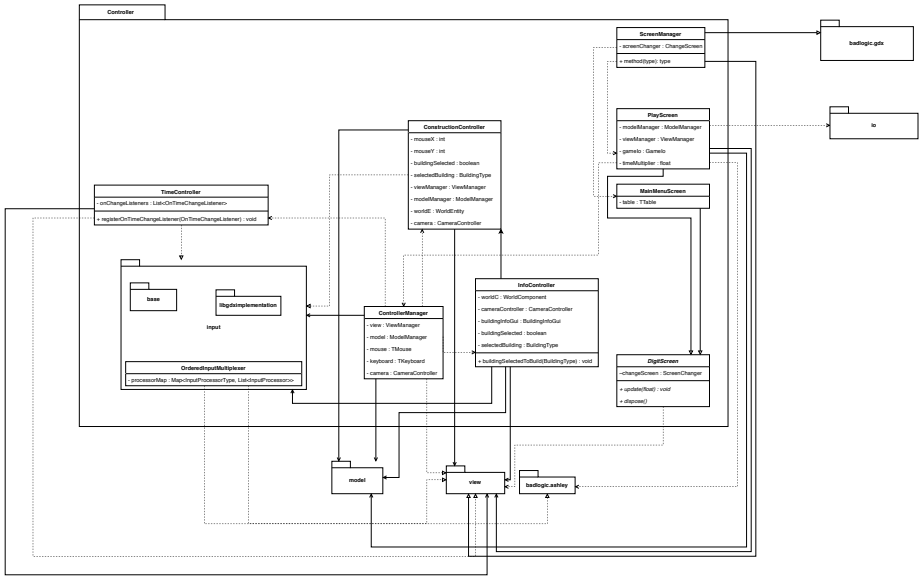


Figure 6: The main controller package.

The controller package contains the general controller classes of the project as well as the "screens" which is the top level classes of each screen in the game. A screen being a setting in the game such as the main menu or the actual game. The ScreenManager class is responsible for keeping track of and updating the different screens and therefore contains the main game loop. The PlayScreen class is the main game screen and intializes the game through the ModelManager, ViewManager and ControllerManager classes. The ControllerManager is then responsible for keeping track of and initializing the rest of the controller package and it's sub-packages. Within the Controller package there is also an input package that only contain two other packages, base and libgdximplementation aswell as the OrderedInputMultiplexer class which is the main input handler in the project.

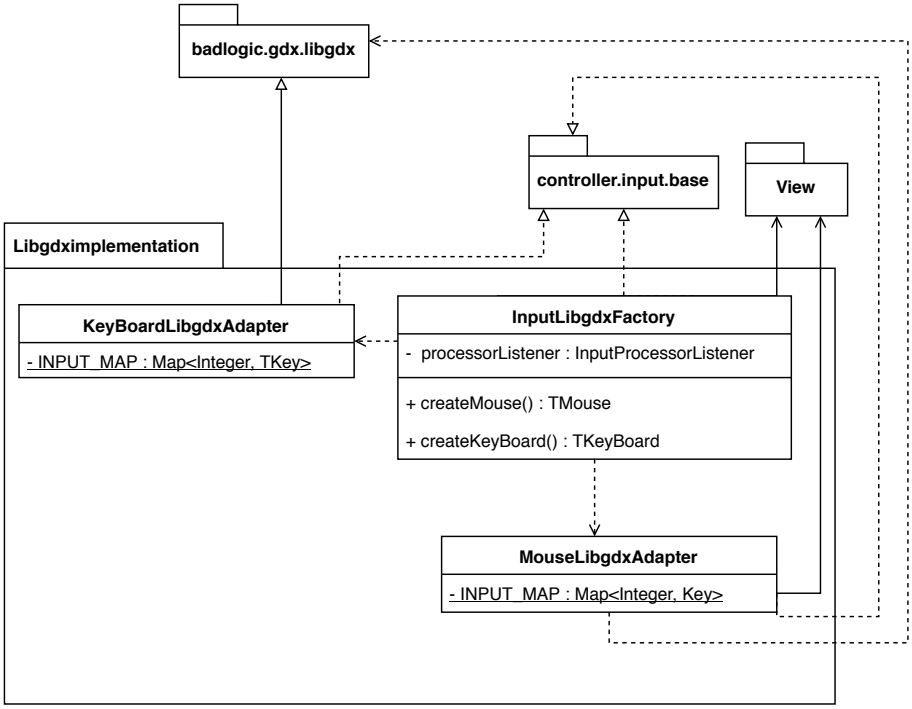


Figure 7: The controller.input.libgdximplementation package.

The controller.input.libgdximplementation package implements libgdx functionality in the game. This package works as a sort of adapter between the controller and libgdx. It also provides a clear interface for other classes that needs to use keyboard or mouse input. 8

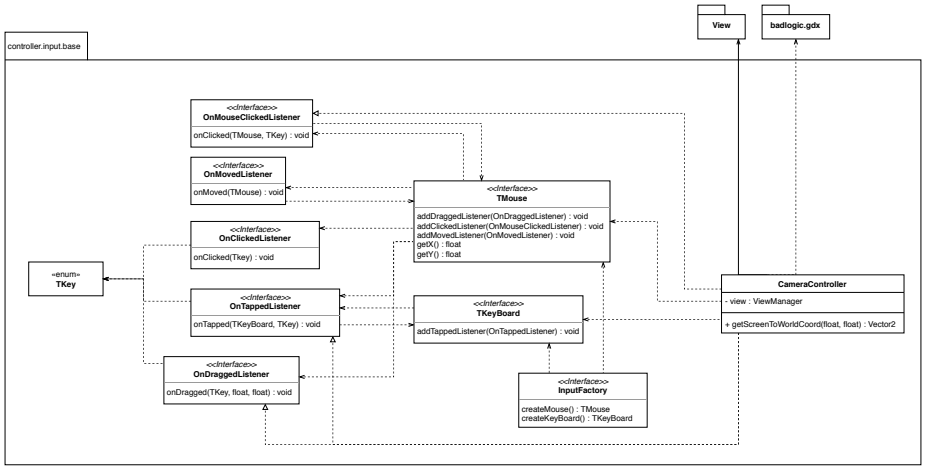


Figure 8: The controller.input.base package

The input.base package mostly contains interfaces that provide the previously mentioned seperation from libgdx, for this purpose it also has the TKey enum which defines all the keys that are used in the project. The package also contains the CameraController which is responsible for the user-camera interaction i.e. zooming and moving the camera.

4.2.2 Model

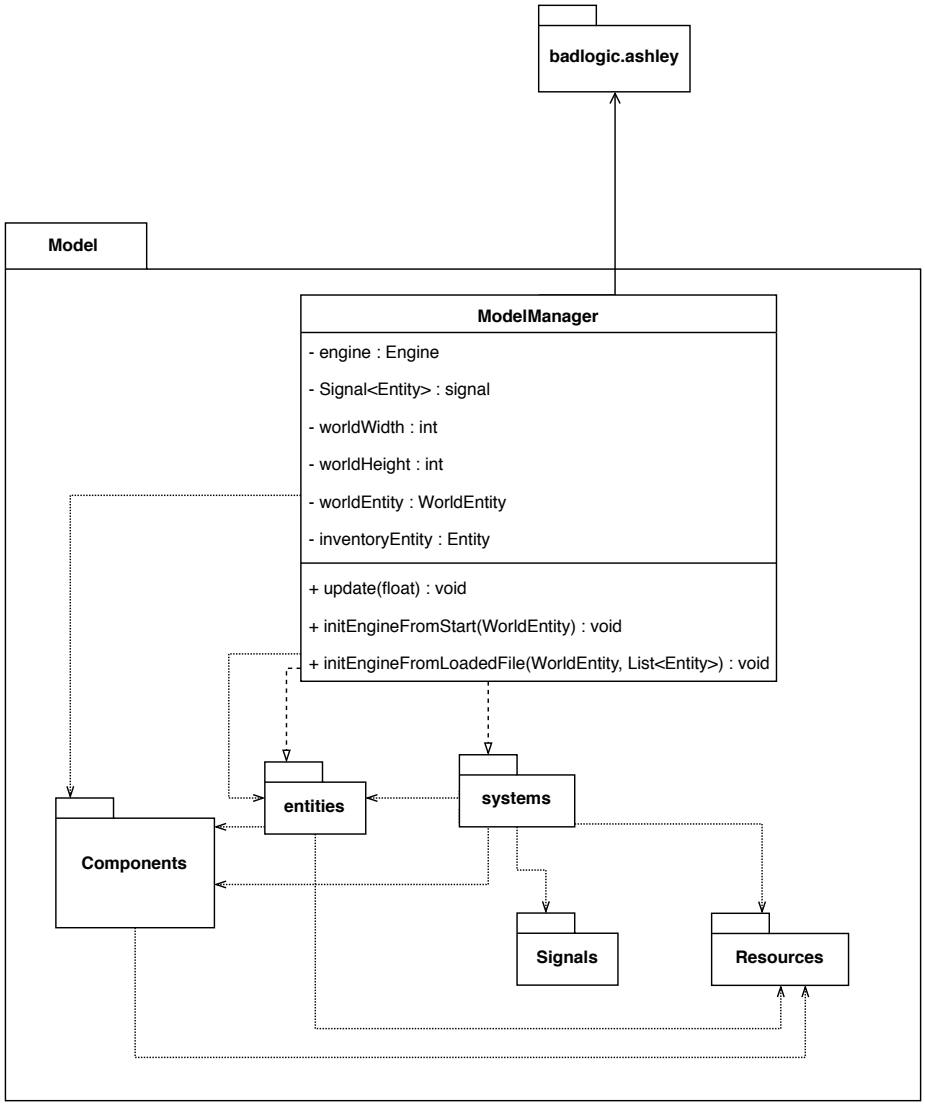


Figure 9: The main model package.

The main class of the model is the ModelManager which is responsible for initializing, keeping track of and updating the model. For this purpose it has an engine, which holds all the entities and systems in the game. The ModelManager is also the only class in the model package, the rest of which is split into several sub-packages by type.

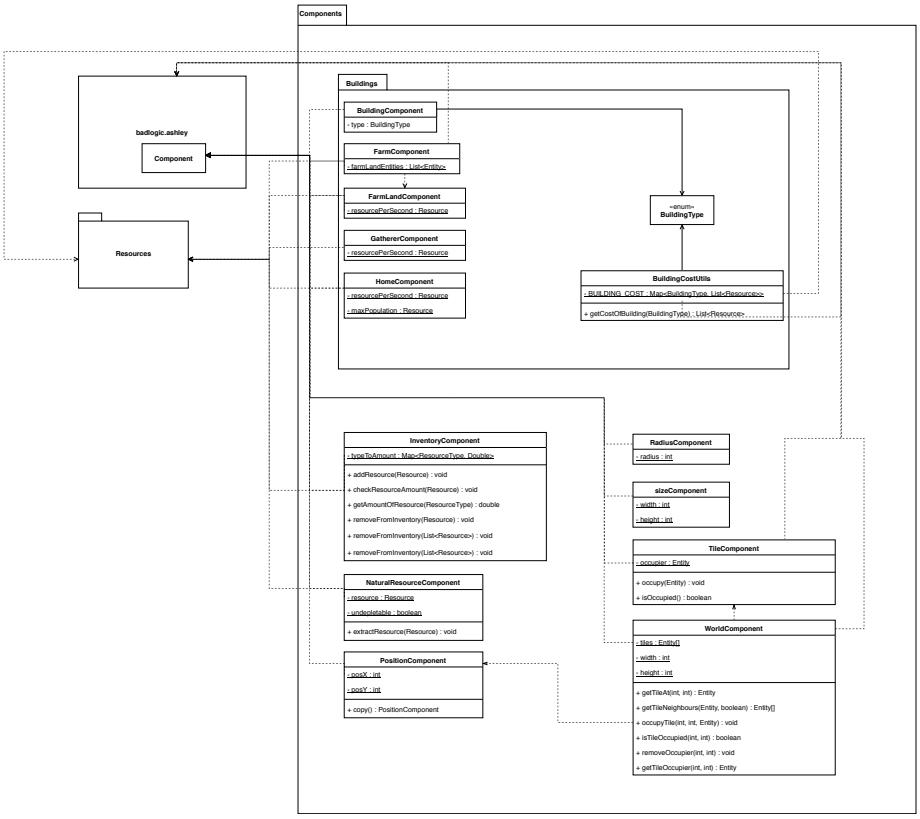


Figure 10: The model.components package.

The component package contains all except one (the SignalComponent, see figure 13) of the components in the game, it also has an inner package, model.components.buildings, to separate the building-related components from the rest. All the components inherits from Ashley's Component class.

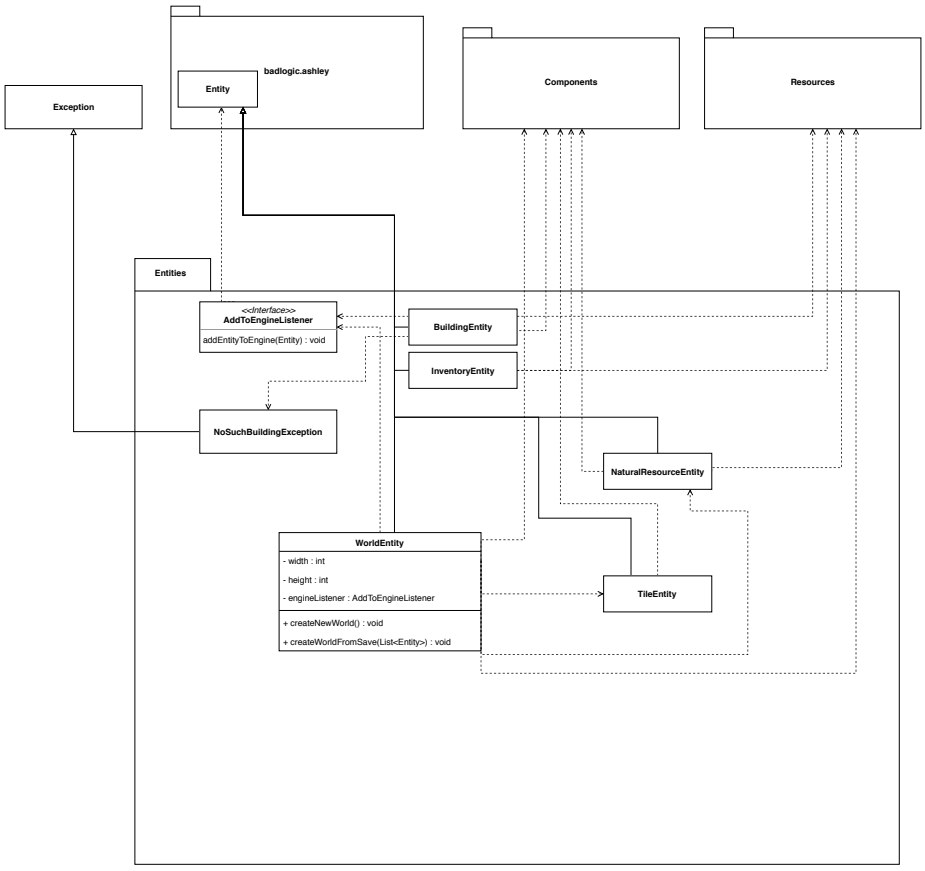


Figure 11: The model.entities package.

The model.entities package contains entity classes that inherits from Ashley's entity class and act as factories. In the entities respective constructor the appropriate components are added to that entity. For example the NaturalResourceEntity simply adds a NaturalResourceComponent, a PositionComponent and a SizeComponent to itself.

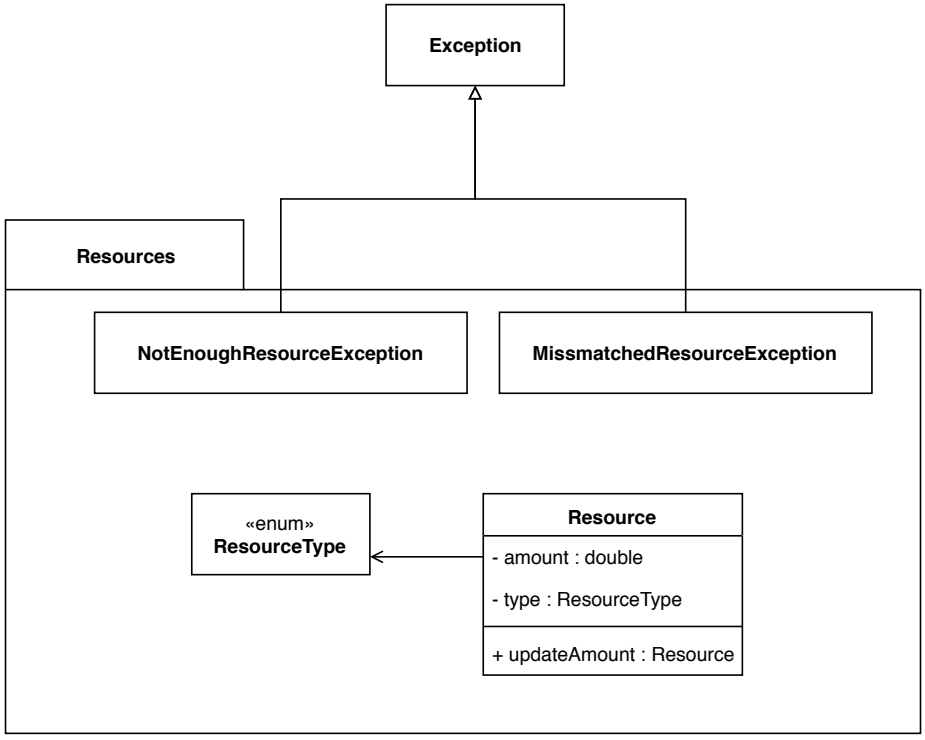


Figure 12: The model.resources package.

The model.resources package simply contain the immutable Resource class as well as a few related classes.

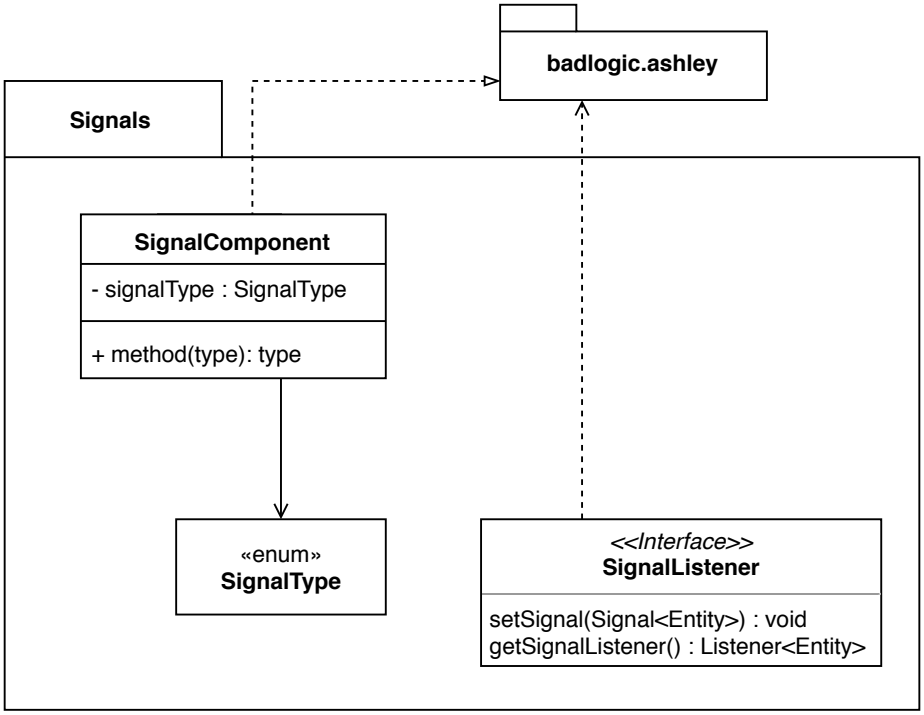


Figure 13: The model.signals package.

The model.signals package contains the SignalComponent which is used on entities that are passed on when a signal is dispatched. The SignalComponent has a SignalType to make sure that only systems interested in the specific signal receives it. Signals are as previously explained used to send messages to systems, with these messages an entity is passed which should have a SignalComponent. The SignalComponent is used to define the SignalType sent so that interested systems can choose which signals to act on.

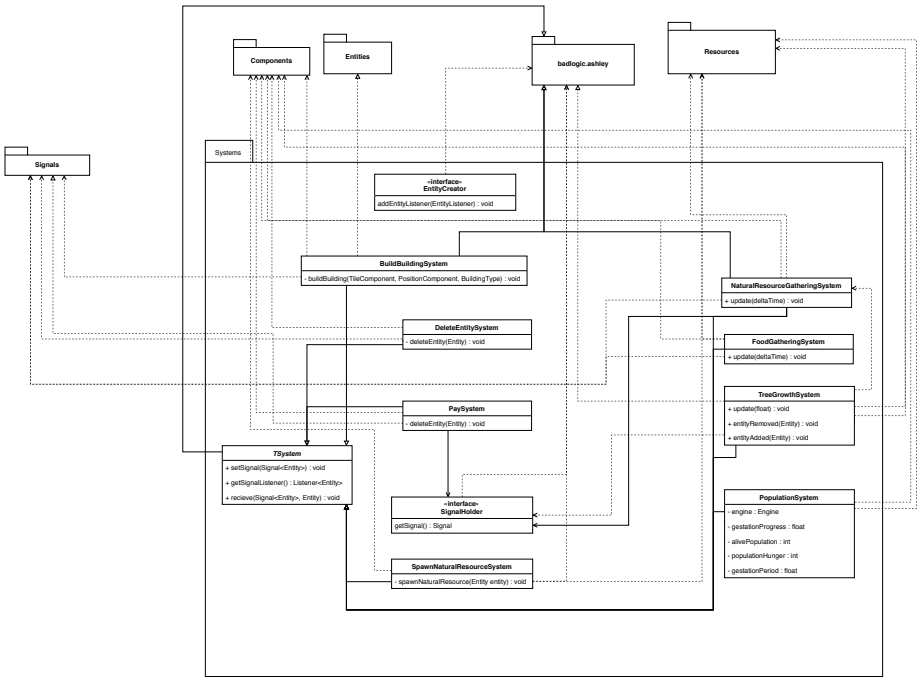


Figure 14: The model.systems package.

The systems package contains all the systems in the application, which in turn manages/manipulates the entities and their components, therefore it needs to have a relatively large amount of dependencies on the rest of the model. Every system extends TSystem which in turn extends EntitySystem from the badlogic.ashley package which gives them a common interface for the rest of the application to interact with.

4.2.3 View

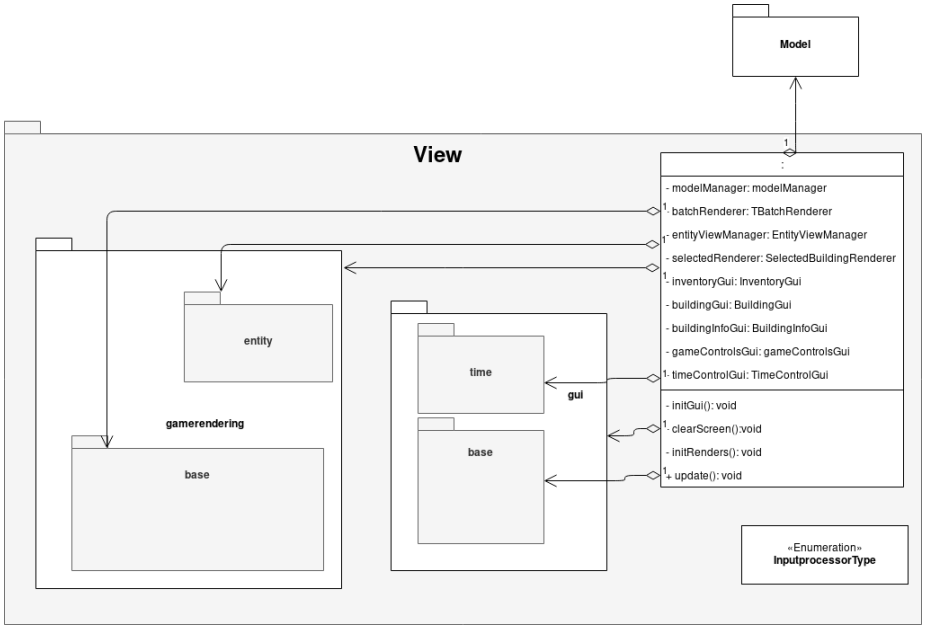


Figure 15: View Overview

The view displays the model and is controlled by the controller and sends events back to it via observer patterns. The View is split into the GUI and the gamerendering package.

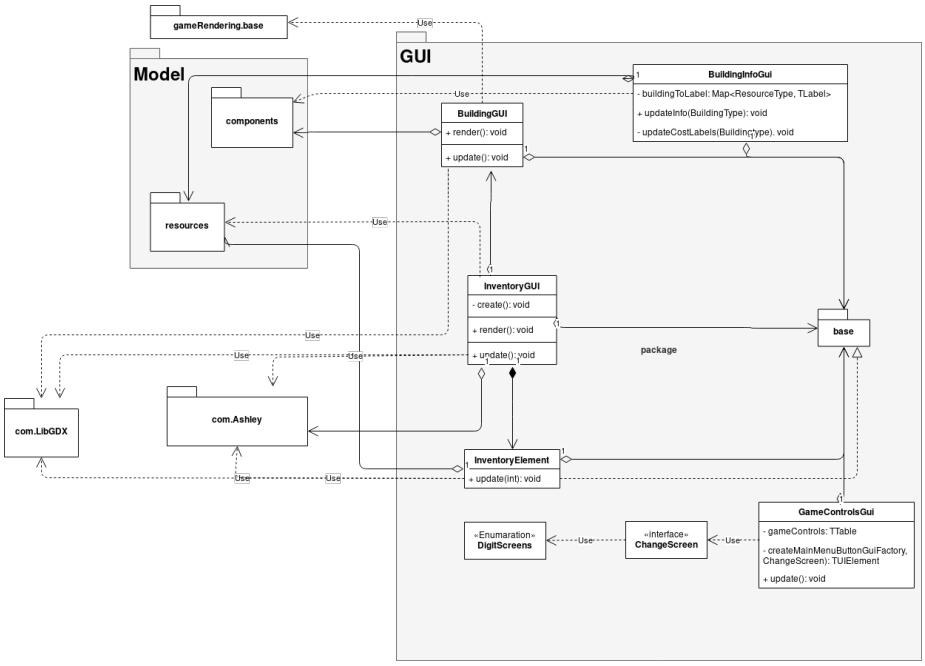


Figure 16: view.gui package

The **GUI** package defines how different kinds of UI element are rendered and how these elements uses the events from interactions with the user to modify the model. Futhermore, it defines how screens are managed and what screens there are. The abstract factory pattern is used to separate libgdx from the view as much as possible.

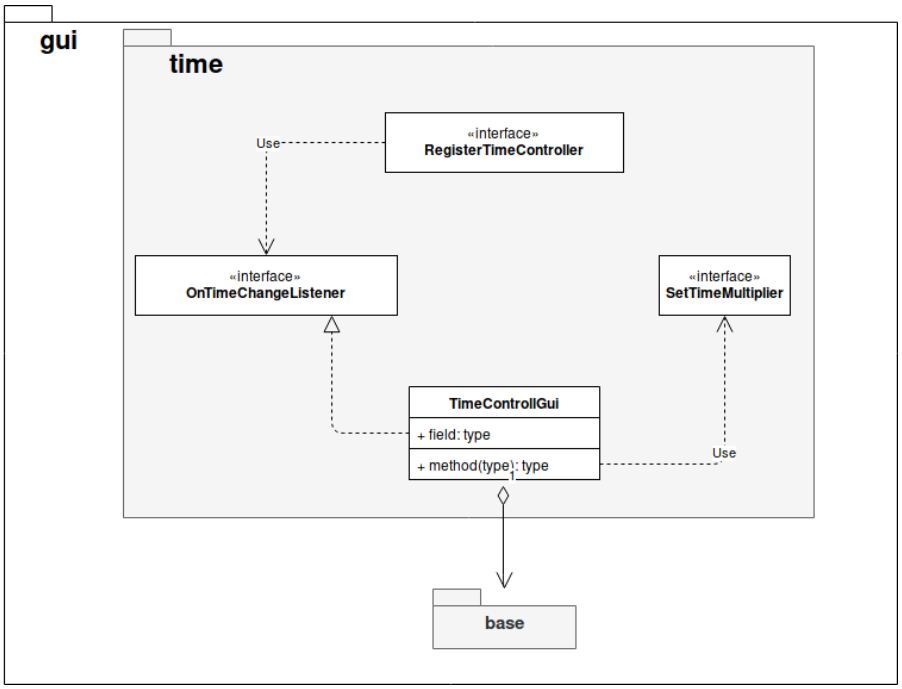


Figure 17: view.gui.time package

The **GUI** for the TimeController and listeners which define how the UI updates based on user input.

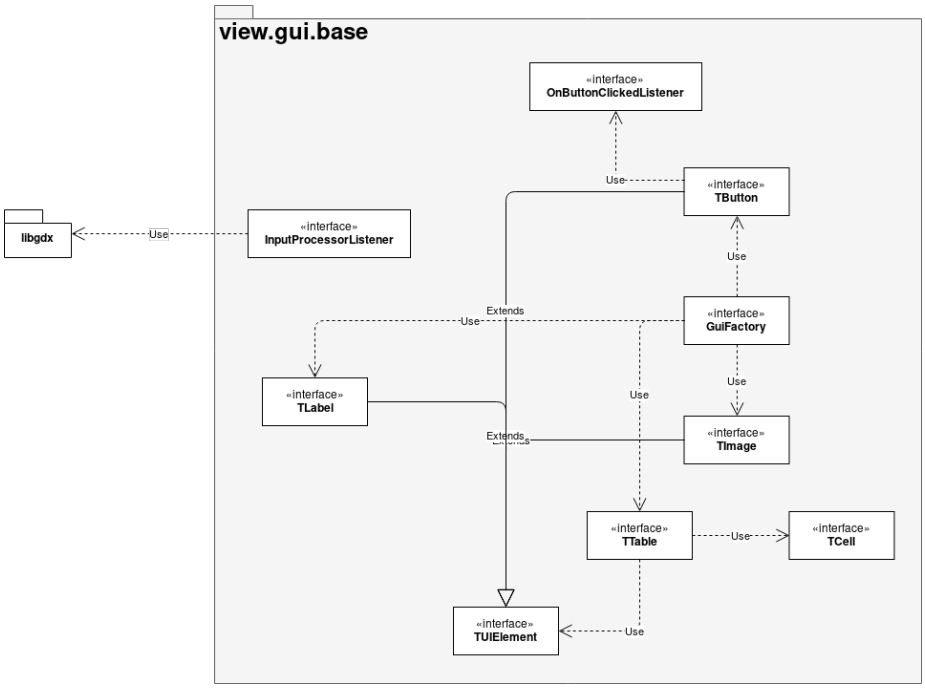


Figure 18: view.gui.base package

The base interfaces of the UI.

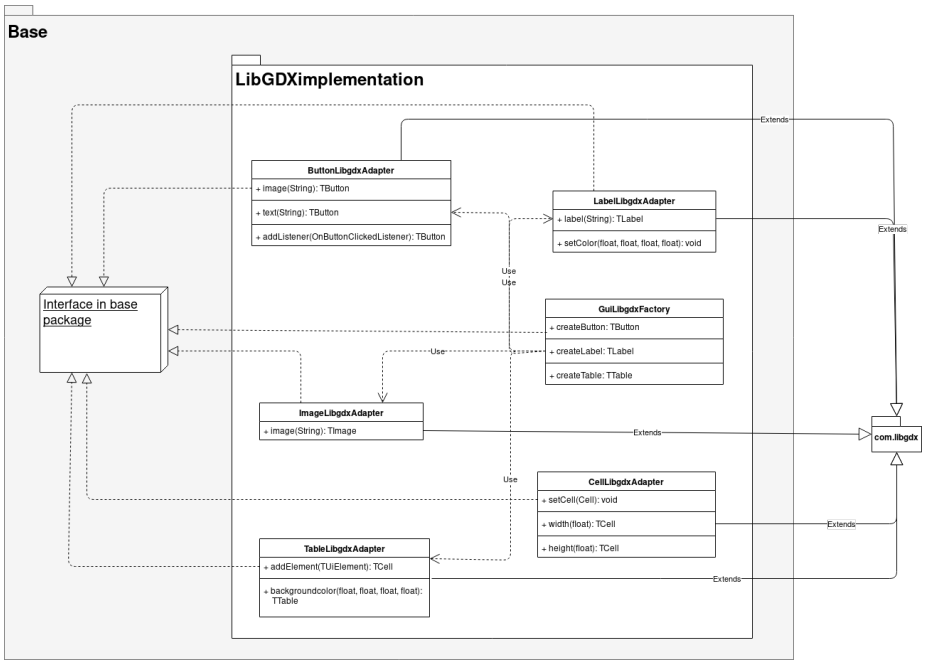


Figure 19: view.gui.base.libgdximplementation package

This is the concrete definition of the UI elements using libgdx.

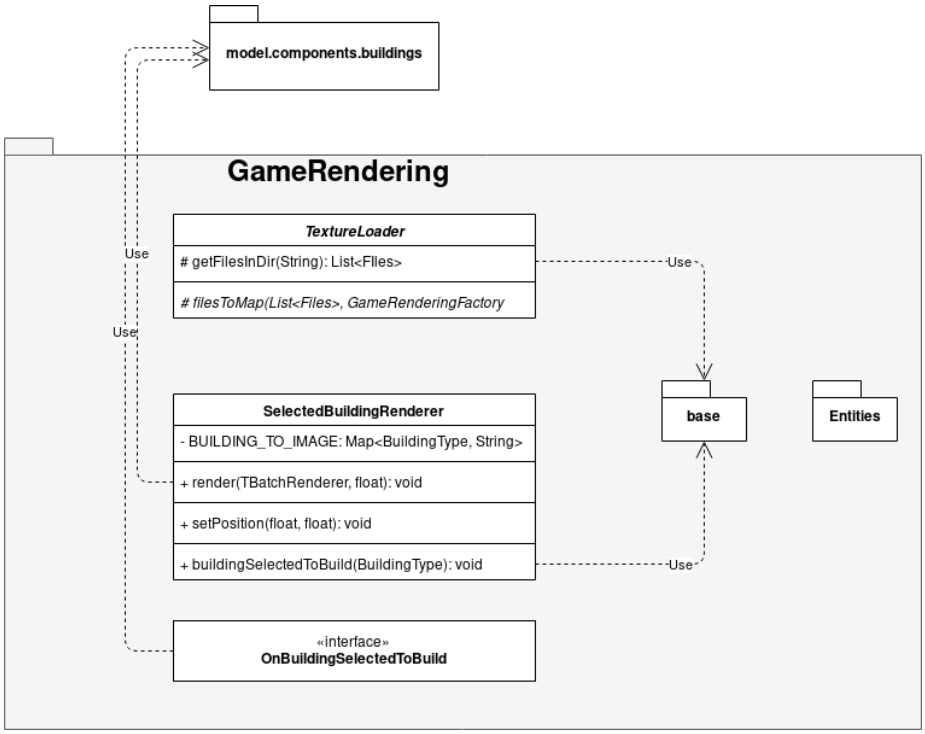


Figure 20: view.gamerendering package

The **gamerendering** package listens to the model's entities, and renders those that need to be rendered. There are a few classes that are used with rendering and the abstract factory pattern is used to define these.

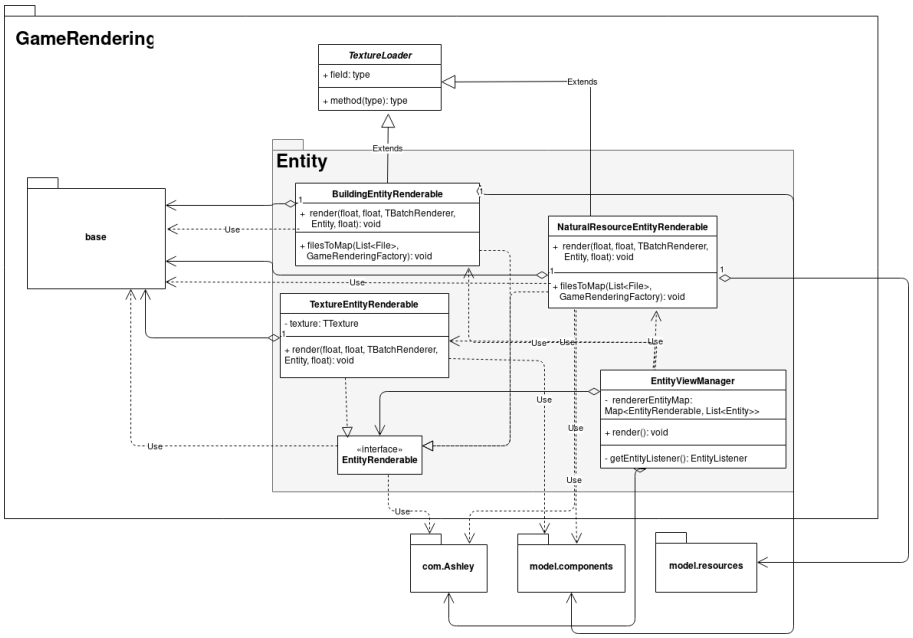


Figure 21: view.gamerendering.entities package

This package contains the logic for entity rendering.

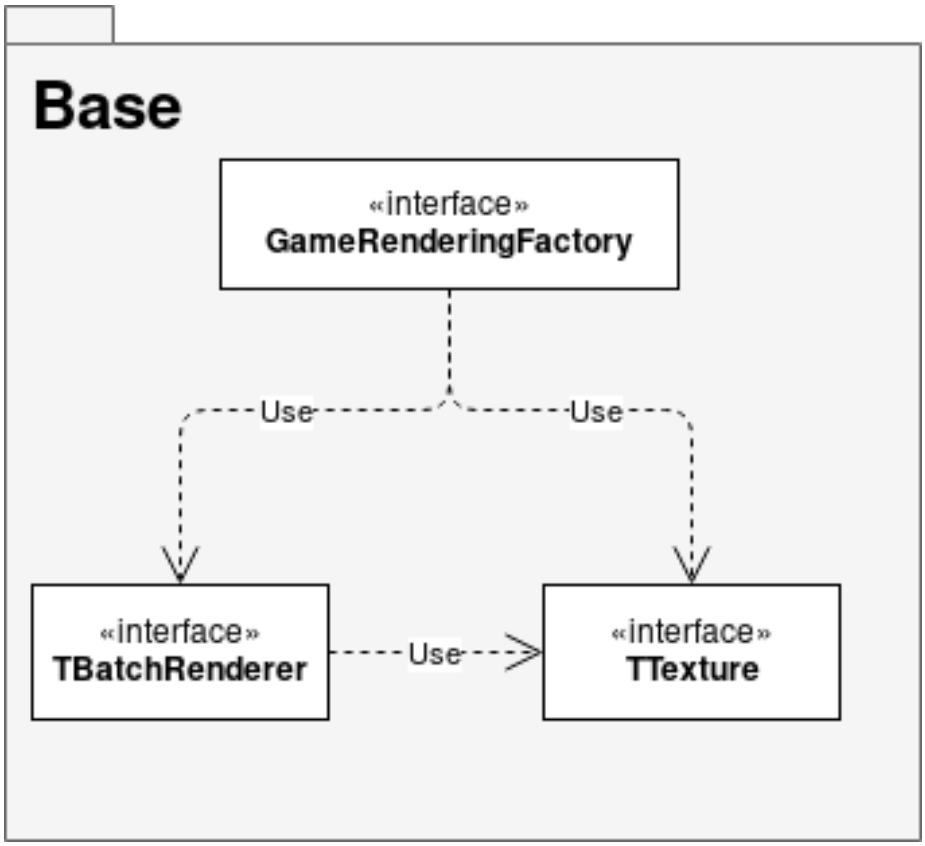


Figure 22: view.gamerendering.base package

The base interfaces of game rendering.

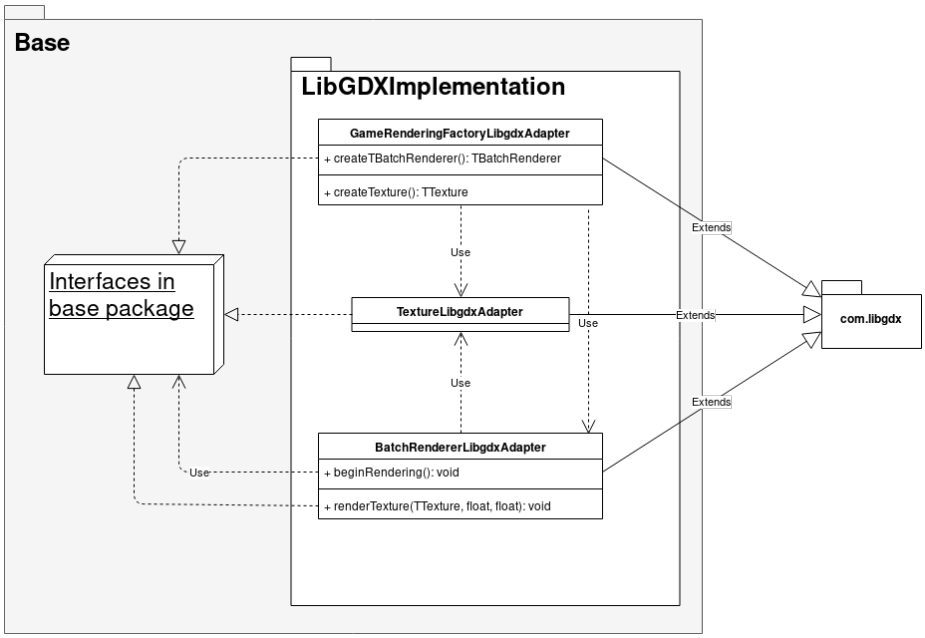


Figure 23: view.gamerendering.base.libgdximplementation package

The concrete implementations in libgdx.

4.2.4 IO

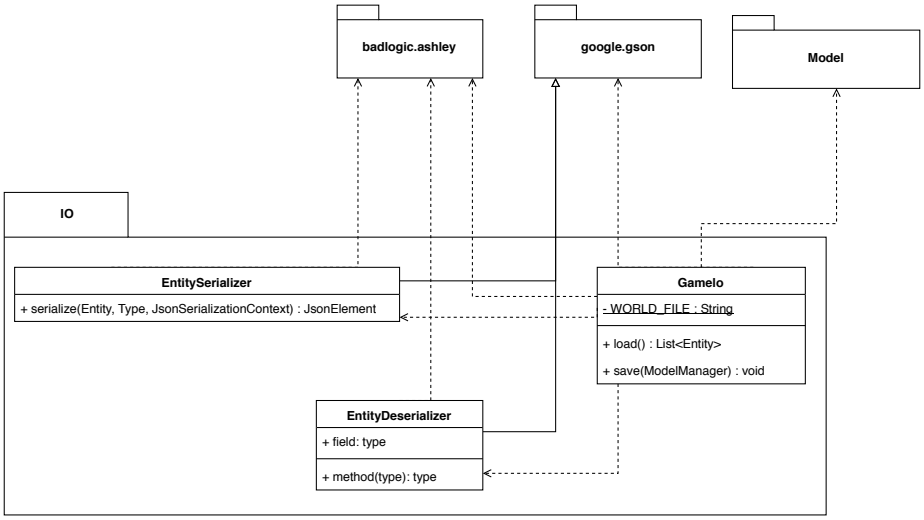


Figure 24: The IO package.

The IO package is responsible for saving and loading the model. The EntitySerializer formats an entity as a json element, the EntityDeserializer converts a json element back to an entity. The GameIO class takes the model and by using the EntitySerializer and EntityDeserializer converts it to and from a json file.

4.3 Sequence diagram

5 Persistent data management

The application saves almost all of the entities in the model. Some of them, like tile entities, cannot change and therefore does not need to be saved. If the size of the world stays the same then there is no reason not to create new tiles when the application is launched. The application uses the GSON library to go through all of the entities and their components and saves them to a json file. When closing the application the world is saved automatically and when starting the application again, the user has the option to either load the previous world or start over.

6 Access control and security

Not applicable since the project is a single-user standalone application. The only time access control or security should be kept in mind is when the save-file is created. If the user has access to the save-file it is easy to simply change it if they want to cheat. However this is not something that is handled by the application because it doesn't make a difference, since the user would only be cheating against the game and not other players.

7 Peer Review

7.1 Design principles

7.1.1 Design patterns

The most significant identifiable design patterns were:

- MVC
  Separates the program into modules with separate responsibilities and is well implemented, with a few exceptions that will be pointed out later.
- Composite pattern
  Is used for the component structure and is a big part of why the model is so easily comprehensible.
- Observer pattern
  Decreases the direct dependencies between classes that need to interact.

7.1.2 Coding style

All the files are structured in a similar way, which makes the entire codebase easier to understand as a whole. Furthermore the code uses good abstractions, such as how the components are dynamically created in an abstract class.

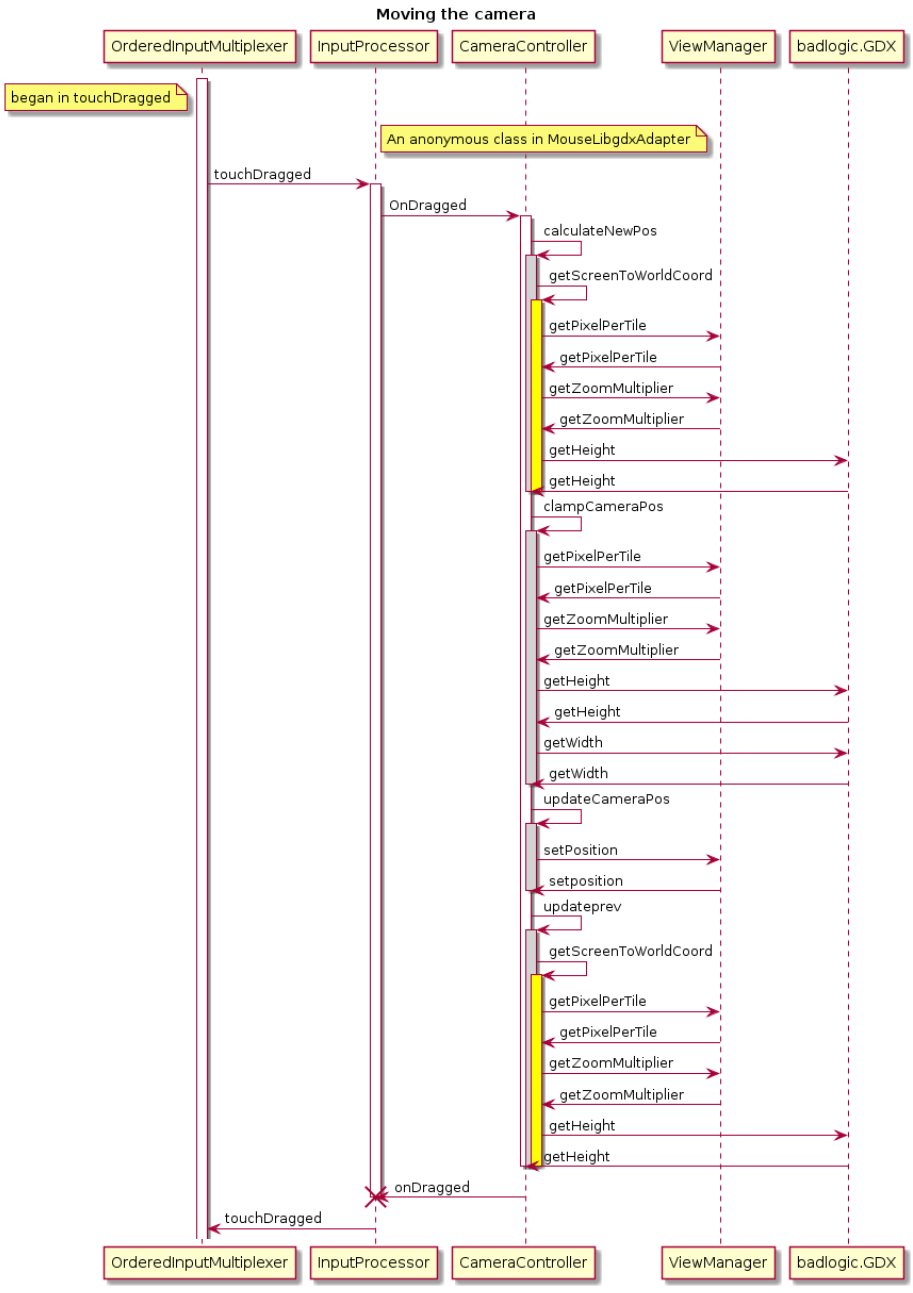


Figure 25: Sequence diagram for moving the camera.

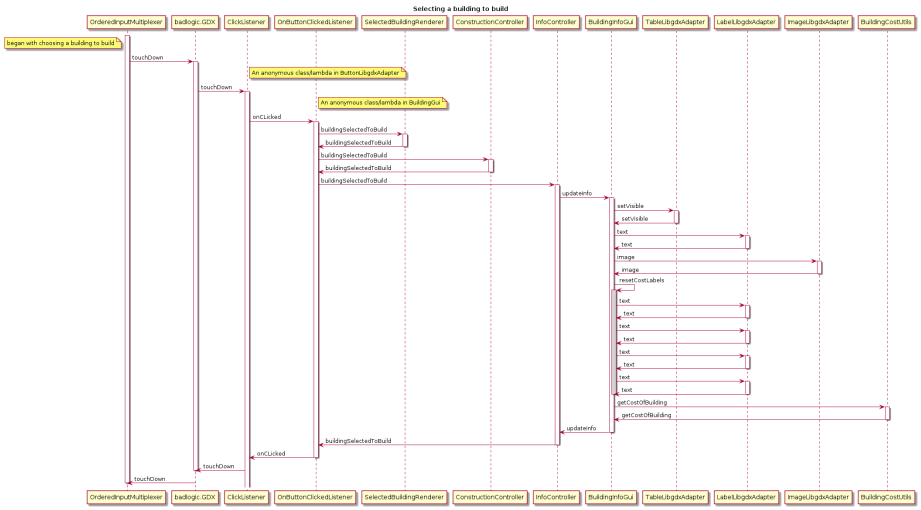


Figure 26: Sequence diagram for selecting a building to build.

7.1.3 Dependencies

The design follows the *high cohesion, low coupling* design principle, which decreases dangerous dependencies and makes the code more modular (seen easily in the UML-diagram). Furthermore the observer pattern helps remove unnecessary dependencies between classes. There are still a few questionable dependencies such as the model having a dependency on JavaFX and the non-"storage" part of the model seem to know that the files are saved in a JSON file format.

The JavaFX dependency only includes the mathematical parts of the library and therefore does not facilitate any direct dependencies on the view, which means that the dependency is not detrimental to the code design. However the dependency on the JSON file format outside of the storage module makes the model less modular.

7.1.4 Reusability and maintainability

Since the project has achieved a modular design with an understandable structure and coding style, it becomes easy to reuse and maintain both parts and the entirety of the project.

7.2 Code documentation and tests

The code-base has relatively few comments but can still be considered well documented as the code is mostly self-explanatory. This has been accomplished with mostly good naming of functions and variables. The naming has obviously been made easier by the fact that functional decomposition has been used

throughout the project to split methods into small, easily described parts. At the top of each class there is also a short comment containing the author of the class as well as a short description of the class.

7.2.1 Tests

They have a relatively large amount of tests comprising of generally small methods testing a specific thing. The tests cover both user interaction as well as the actual program logic. As with the rest of the project the test methods are highly self-explanatory with good naming and single method responsibilities. Examples of test methods found in CircuitTest.java are: "addSingleWireToCircuit", "removeItemShouldRemoveWire" and "removeNoneExistantComponentShouldThrow", while somewhat long and "wordy", these names give a clear explanation of what the test is trying to accomplish.

7.3 Code comprehensibility

7.3.1 MVC

The top-level parts of the program are well separated. The model does not depend on the controller or view and the view does not depend on the controller.

7.3.1.1 Model The model uses a single interface (IModel) to provide abstraction from the rest of the program which makes the rest of the program more comprehensible. The code in the model is mostly well structured but some weird parts of it was discovered during the analysis that will now be summarized in short:

- The model has a camera to control how much of the model that is being represented in the view. Even though the model is not dependent on the view, this is still questionable since the model should not know how it is being represented. It would probably be more logical to place this part of the model outside of the model. The camera also depends on "JavaFX" which indicates that it might not belong in a model package.
- There are Strings referred to as "Component ID's" that are found in several parts of the model, e.g the JsonStorage class and the Component class. This indicates bad design because the code is dependent on certain hard-coded strings instead of e.g. an Enum.

7.3.1.2 View Nothing in particular was noticed about the design of the view-package during the analysis.

7.3.1.3 Controller A problem with the controller that was noticed during the analysis was that JavaFX dialogs such as file chooser and options is instantiated in the controller. This is a problem since for example the text on buttons and labels are set in the controller, which does not follow MVC.