System Design Document for Gastuen

David Andreasson, Erik Berg, Karl Gunnarsson,
Jakob Henriksson, Jakob Ristner

2020-10-23

1 Introduction

The general purpose of the the application ”Gastuen” is to act as a digital representation of a board game inspired from the physical board game ”Betrayal at House on the Hill”. The premise is that a group of adventurers are exploring a haunted house filled with traps, items and different events. After a while one of the players betrays the other players, thus taking the side of the monsters.

The SDD for the application ”Gasuten” will contain information about the design, development and testing of the application.

1.1 Definitions, acronyms, and abbreviations

- rollDiceEvent - An event where the outcome affects the players stat. Based on a dice roll.
- moveEvent - An event where the outcome affects the players coordinates. Based on a dice roll.
- itemEvent - An event giving the player an item. Not based on a dice roll.
- gameWonEvent - An event that results in the player winning the game when entered. Not based on a dice roll. Resembles the ”escape hatches” that are distributed when a player gets haunted.
- Tile map - a grid of squares representing the different rooms in the house.
- Sprite - Circles on the Tile Map representing the different players.

2 System architecture

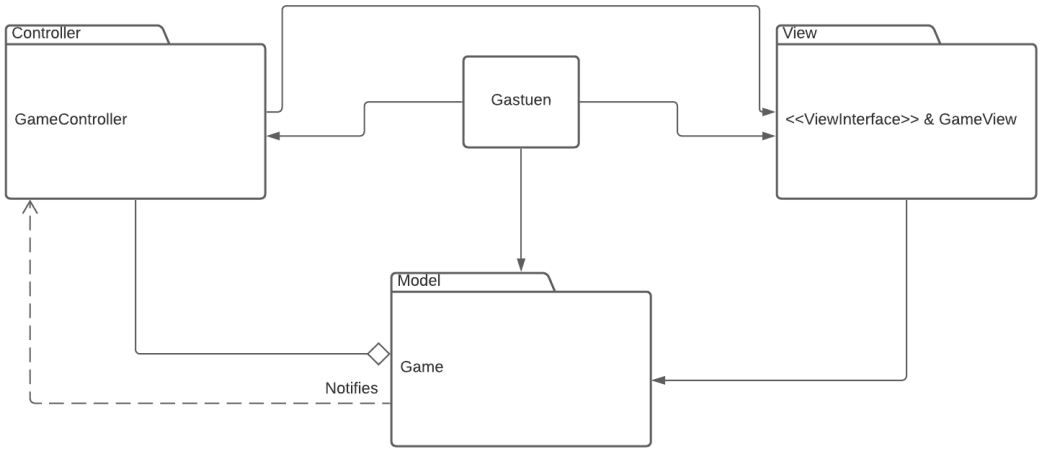

Figure 1: Top level architecture of the application

The Application is built upon the foundation and structure of the Model, View, Controller design pattern. This means that the model or ”logic” of the application is completely independent and can for instance be tested independently from the view and controller modules. The controller module handles all of the user input and forwards this to the model. Last but not least the View module is what shows all of the graphics of the game. The application is also structured and programmed as an incremental turn based game. This means that things only happen behind the scenes when an action is performed by the user such as pressing a button.

3 System design

As stated above, the game uses an overall structure of the MVC pattern. The package diagram shows how the components of the applications communicates.

3.1 Package Diagram

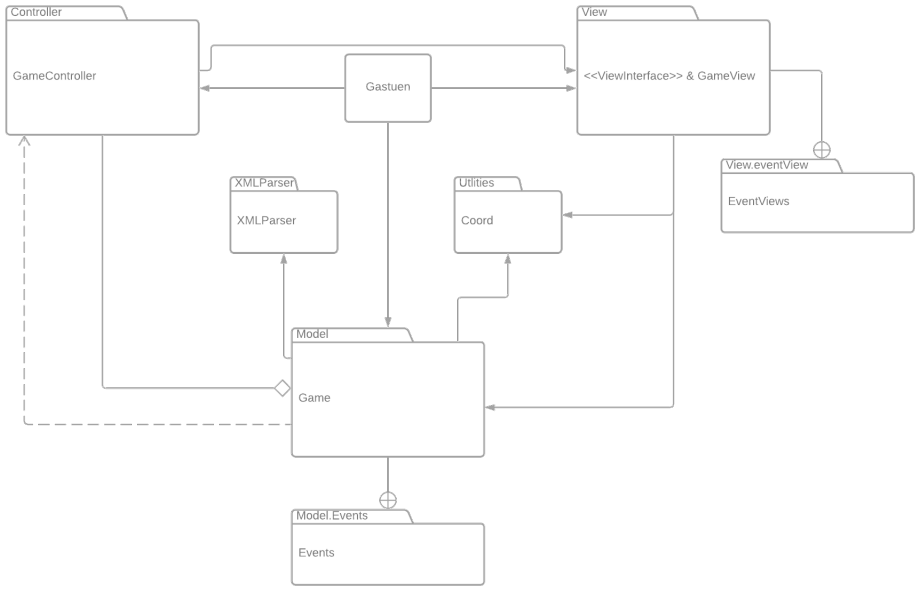

Figure 2: UML package diagram

As can be seen in figure 2 we have tried to eliminate all circular dependencies between the packages. The model communicates back to the controller via an Observer pattern which means that there is no direct dependency going the wrong way. Gastuen is the main class which only works as an initiator of the different packages and as the initiator of the game.

3.2 Design principles

Our game has been designed with a number of design principles in mind. The most important of which are:

- **Open Closed principle**
  The principle states that ”Objects or entities should be open for extension, but closed for modification.” [1]. The game is designed with the principle in mind since we make use of both factories for creation of objects and the extensive use of interfaces that we have. For example is it very easy to create more eventtypes and it should be easy to create more haunts.

- **High cohesion, Low coupling**
  High cohesion can be described as to what degree a module of a code base forms a logically single autonomous unit, while low coupling can be described as to what degree the single autonomous unit is independent from other units in the code base [2]. Thus High cohesion implies keeping components related to each other in the same place and Low coupling implies to separate unrelated modules from each other in the code base as much as possible. In our project the designer pattern Model view controller is used to separate the the model view and controller from each other thus forcing them to work as independent autonomous units. The XML parser is also in its own package, which improves high cohesion and is only connected to the EventFactory, which improves low coupling. There are other examples where classes have been placed in packages. For instance, the Event package and the eventView package.

- **Dependency Inversion principle**
  Dependency Inversion principle states that ”Entities must depend on abstractions not on concretions” [1]. The way this is implemented in the game is through our many interfaces. Events are abstracted to an interface called Event and the views are abstracted to an interface called ViewInterface. These help reduce coupling since the other classes in the game can depend on these interfaces instead of having to depend directly on the many views for example.

3.3 UML diagrams

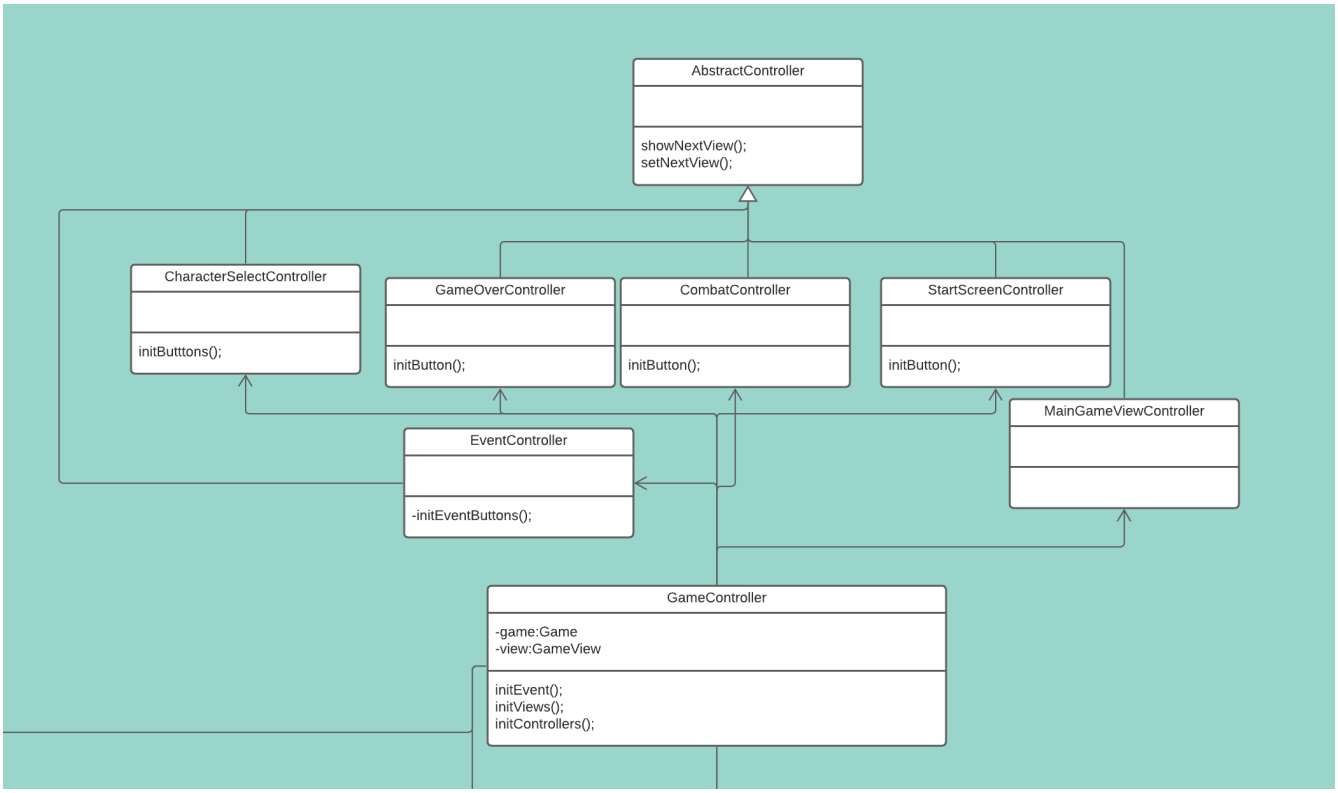

Figure 3: UML of the controller package

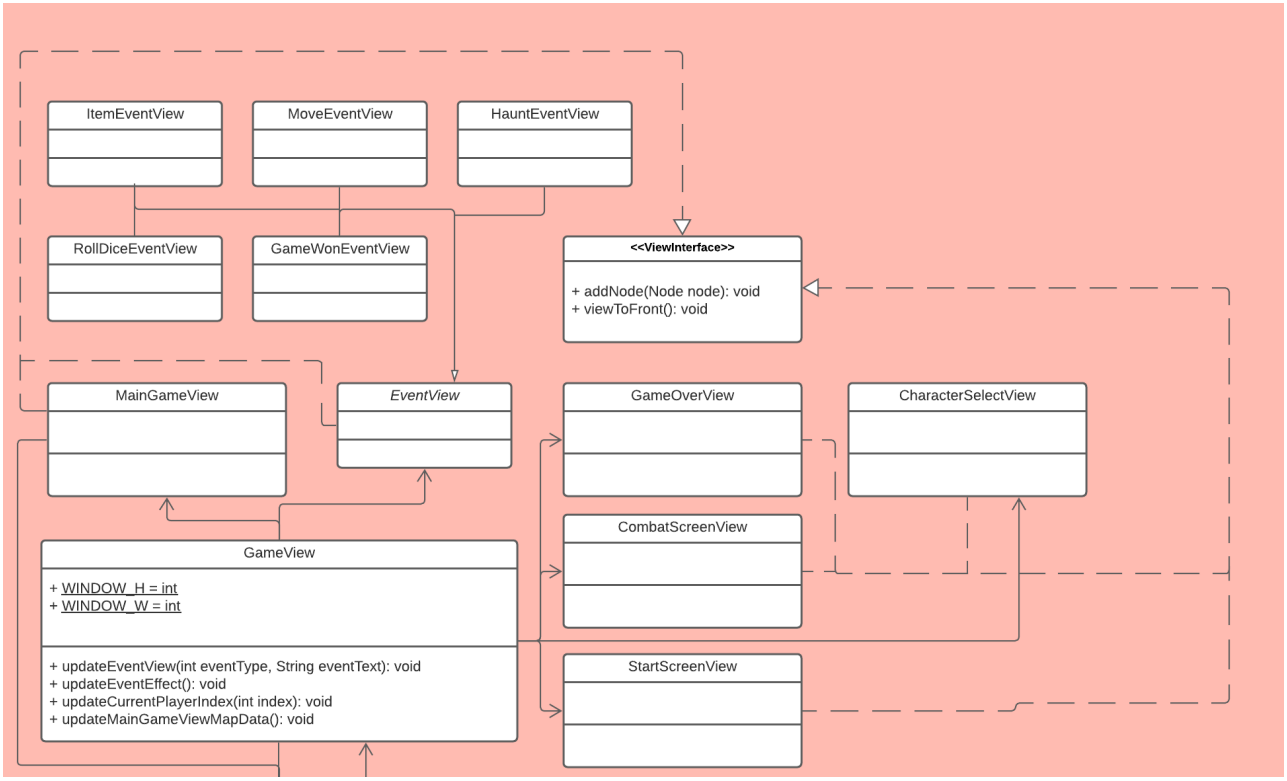

Figure 4: UML of the view package

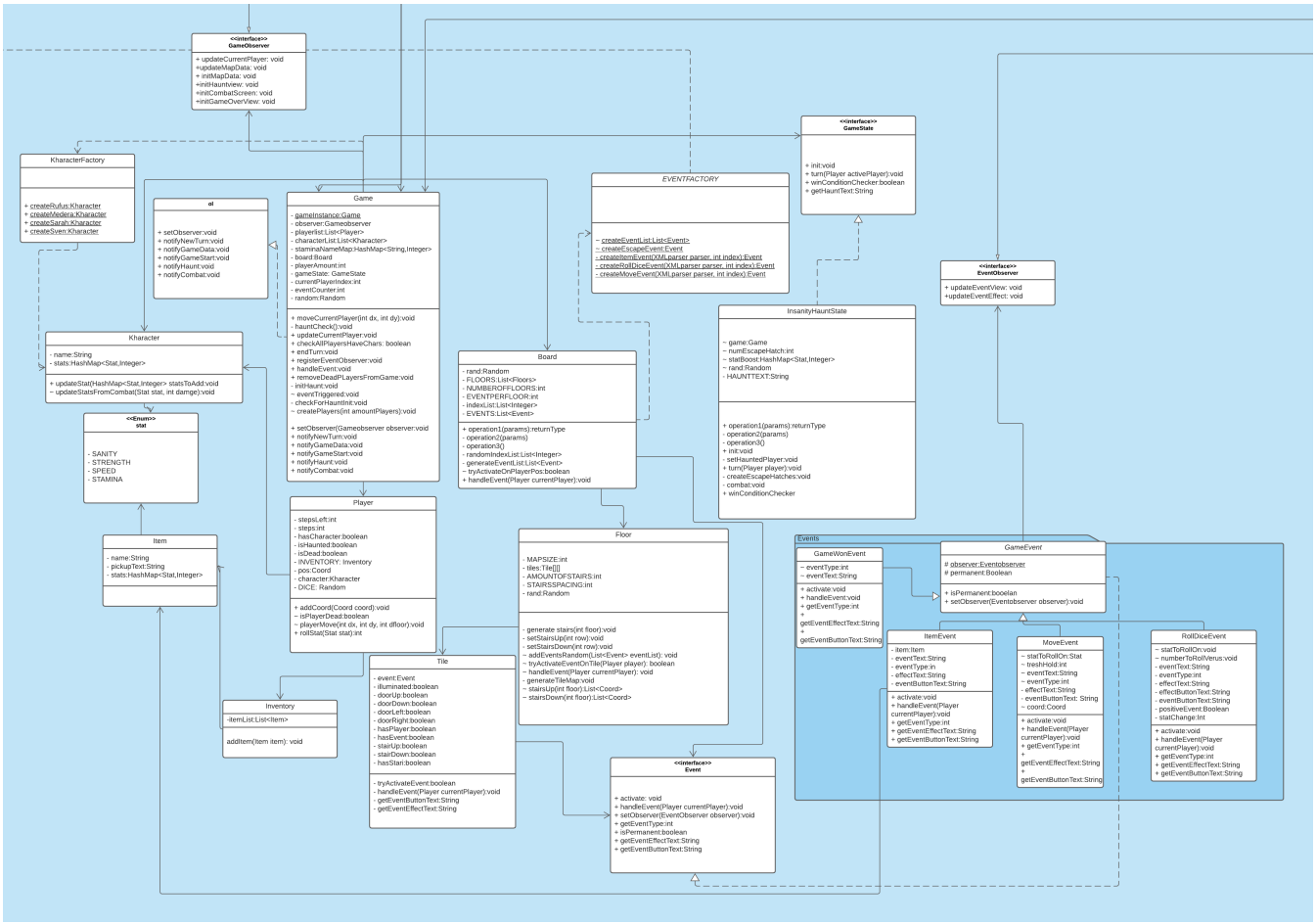

Figure 5: UML of the model package, higher resolution can be found on Github[3]

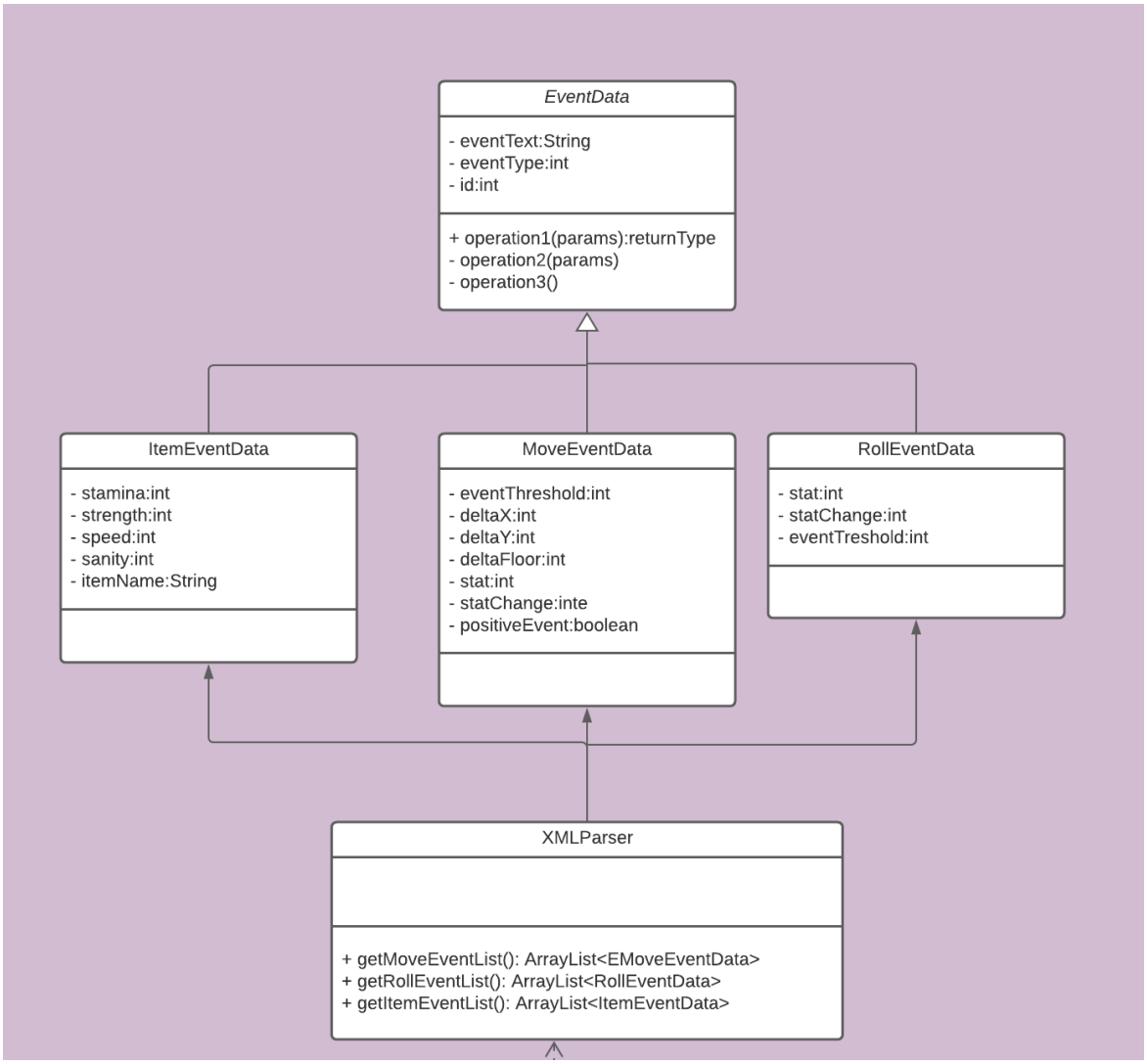

Figure 6: UML of the xmlParser package

Figures 3, 4 and 5 represent the complete UML diagram of the game. Figure 6 represent the Parser package. Together they represent the overall MVC structure of the program. To avoid strong dependencies, a number of interfaces has been created. For example do every View implement the interface ViewInterface and then separate controllers have instances of the ViewInterface instead of having direct dependencies on the smaller views.

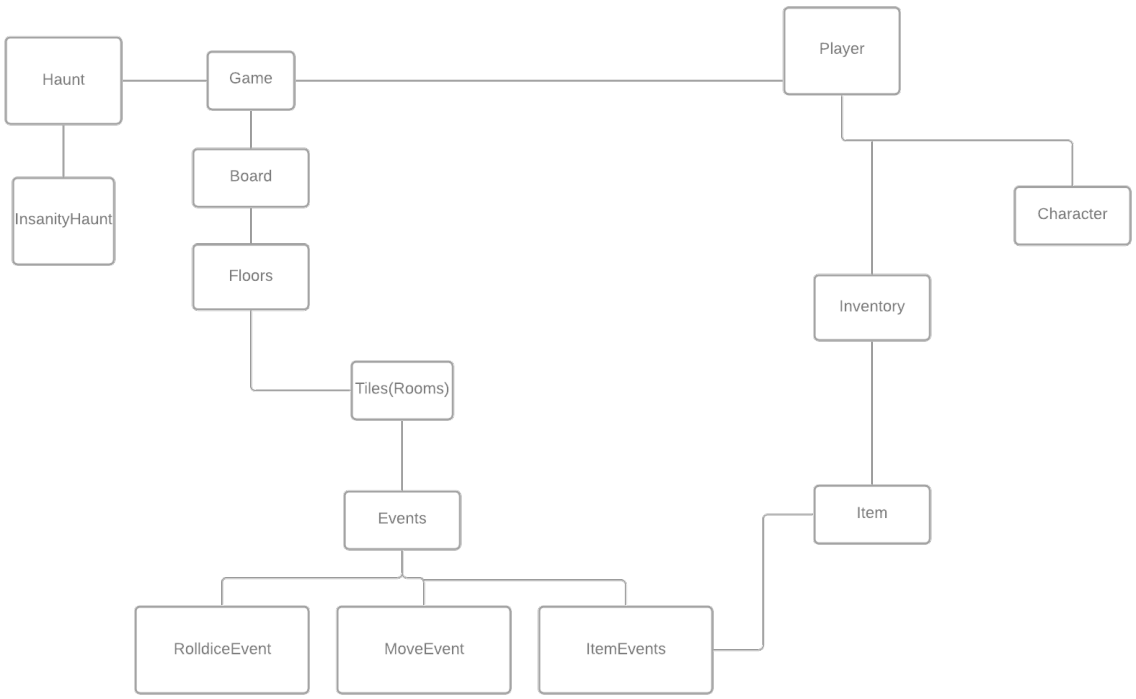

Figure 7: Domain model of the application

The domain model shows how the application on a top level and how all the structural parts of the model connect and communicate. The UML diagram in figure 5 explains on a more detailed way how the model works and elaborates on the technical design of the model. The domain model is a simpler way of representing the model than the UML model diagram.

In the model diagram, as seen on figure 5, one can also see how the inner structure of the model is designed. There one can find features such as a singleton Game, interfaces for abstractions and how all these connect and depend on each other, whilst in the domain model, fig 7, it's only a rudimentary way of showing how the parts connect. It doesn't show any interfaces or abstractions.

The UML diagram also shows on a deeper level how the model communicates with the other packages. Since the model can't depend on the other packages, the view and controller need to depend on the model. The facade of the model is the Game class and it is this class that the View and Model depend on. The controller uses the model facade to update it when for instance a button is pressed. This could happen when a player should move to another tile or in combat for example. The controller uses many of the void methods in the model facade for this reason.

The view on the other hand makes use of all the public return methods in the model

facade to update the GUI. The basic process for the application is as follows; a button is pressed which updates the model and then the view is notified that something has happened and gets the new data from the model.

The XMl parser package is used as an abstraction of an xml parser used to generate our events so that the model doesnt have to depend on an external libraby. It then sends ”Data object” versions of these events to the model and are converted into concrete event objects used in the application.

3.4 Design patterns

- **Model-view-controller**
  Model-view-controller is used to bring structure and good dependencies [4]. The pattern's focus is to separate the code into three packages, a model which is holding data, a view which is responsible for visualising everything and a controller which is responsible for handling all user requests(inputs etc). The perks of using the MVC pattern is that the code gets high cohesion and low coupling.

- **State Pattern**
  State pattern is a design pattern that allows an object to change between its internal state and hence altering its behaviour [5]. In our case the state pattern is used to handle the different ”haunts”. The logic that is shared between all the different ”haunts” will be implemented in the game class. If a state is activated, the state methods will delegate to the state class for the specific haunt, thus making it easy to add extra functionality for the different haunts and also makes the ”haunts” extendable.

- **Observer Pattern**
  The Observer Pattern is used to reduce the amount of dependencies necessary in the MVC. An Observer Pattern is a pattern where if there are a lot of dependencies that is necessary then one can use an Observer interface to notify all of the dependent classes[6]. That way the amount of dependencies is reduced to only have a dependency on the Observer Interface.

- **Facade Pattern**
  The Facade pattern is used to encapsulate most of the game logic [7]. This is done by the game class thus providing simpler methods for the client to use and hides the complexity of the model. The Facade pattern also reduces the amount of dependencies between the model, the view and the controller[7] as it is the only class that acts as a entry point to the model.

- **Factory Pattern** The Factory pattern is used to hide information of how the object is created from the client. This pattern is used by having a factory class with

static create methods and create all objects of the type through the instance of the factory[8]. The Factory Pattern also makes the code easier to maintain since changes to how a type is instantiated can be done in one place.

- **Singleton Pattern** A singleton pattern is used when you only need and only should have one instance of an object in the whole application [9]. It is created by having a private constructor and a static factory method to get the instance. In our program the class Game is a singleton, because that class is only supposed to be instantiated once. Game is the facade of the model which is why it makes sense that only one should exist.

4 Quality

- Testing of the application is done with JUnit and can be found on *github* [3].
- List of known issues:
  - The arrow that show where the stairs are can sometimes end up over the movement buttons which renders movement impossible.
  - Players can't enter combat if they are standing on a stair tile.
  - After all players have been killed by the haunted player in the "insanity haunt", the haunted player must take another step to trigger the view for winning the game.
- We have run the tool STAN [10] on our code and the results are as follows:
  - **Composition of package**

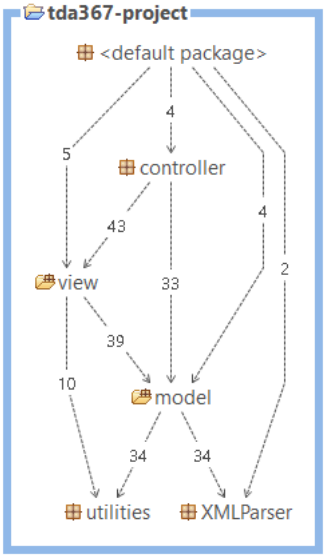

Figure 8: STAN model of package composition

– **Model dependencies**

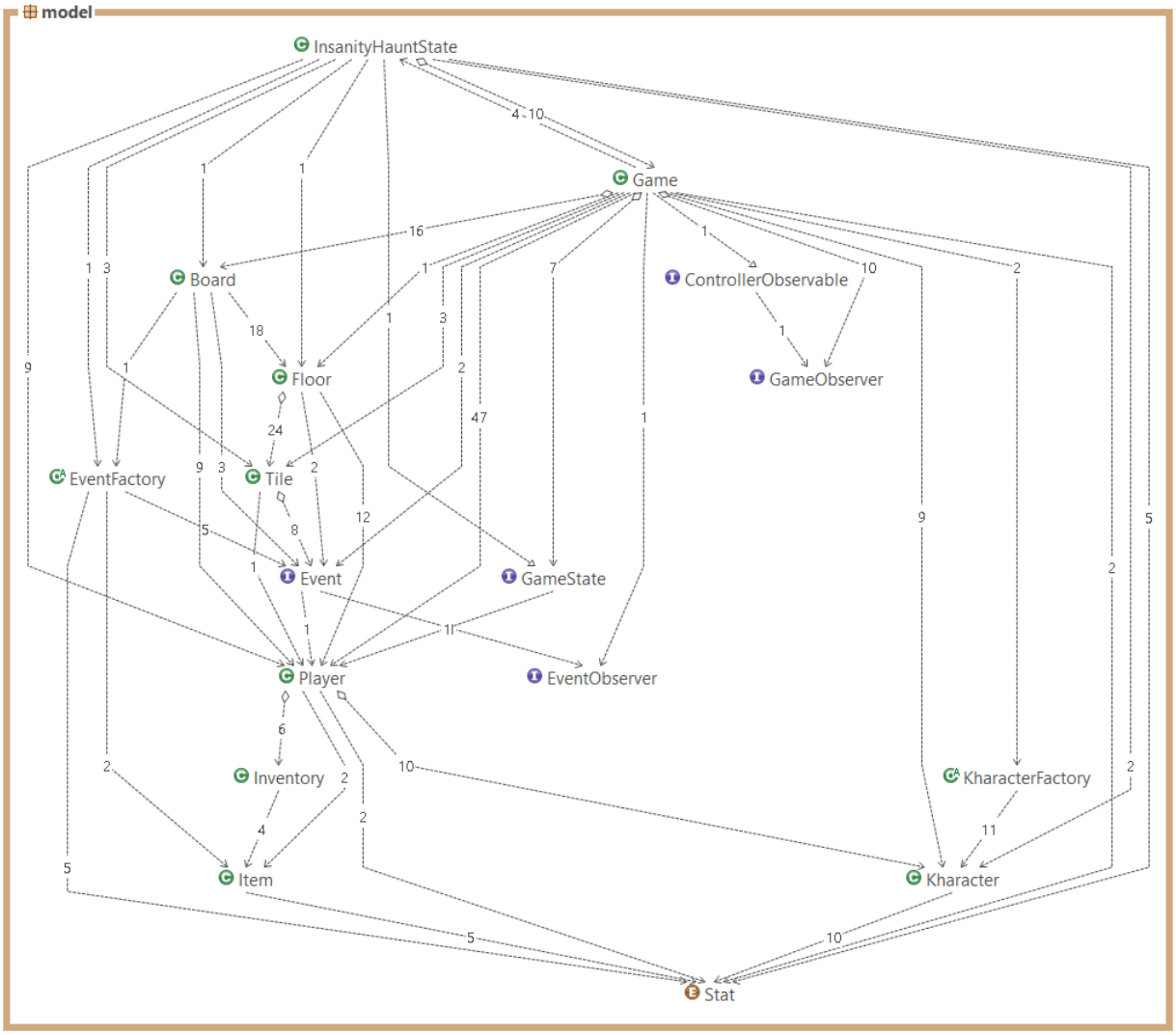

Figure 9: STAN model of model package dependencies

5 References

References

[1] S. Oloruntoba. (Sep. 2020). S.o.l.i.d: The first 5 principles of object oriented design, DigitalOcean, [Online]. Available: `https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#open-closed-principle` (visited on 10/23/2020).

[2] V. Khorikov. (Sep. 2015). Cohesion and coupling: The difference, Enterprise Craftsmanship, [Online]. Available: https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference (visited on 10/23/2020).

[3] J. Ristner, J. Henriksson, K. Gunnarsson, D. Andreasson, and E. Berg. (2020). Gastuen github, [Online]. Available: https://github.com/rille2/tda367-project.

[4] C. —. C. MVC: Model View. (2019). Mvc: Model, view, controller — codecademy, Codecademy, [Online]. Available: https://www.codecademy.com/articles/mvc (visited on 09/28/2020).

[5] I. Darwin. (2019). The state pattern, Oracle.com, [Online]. Available: https://blogs.oracle.com/javamagazine/the-state-pattern#anchor_4 (visited on 10/23/2020).

[6] (2020). Design patterns - observer pattern - tutorialspoint, www.tutorialspoint.com, [Online]. Available: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm (visited on 10/23/2020).

[7] (2020). Design patterns - facade pattern - tutorialspoint, www.tutorialspoint.com, [Online]. Available: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm (visited on 10/20/2020).

[8] (2020). Design pattern - factory pattern - tutorialspoint, www.tutorialspoint.com, [Online]. Available: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm (visited on 10/23/2020).

[9] baeldung. (Sep. 2017). Singletons in java, Baeldung, [Online]. Available: https://www.baeldung.com/java-singleton (visited on 10/23/2020).

[10] B. I. Consulting. (2017). Stan - structure analysis for java, [Online]. Available: http://stan4j.com/.