System Design Document for Pawntastic

Martin Jonsson, Jonathan Lindqvist,
Jacob Bredin, Mathias Prétot

October 10, 2021

Contents

1. Introduction **1**
1.1. Definitions, acronyms, and abbreviations 1

2. System architecture **2**

3. System design **3**
3.1. Domain model and design model 5
3.2. Design patterns 7
3.2.1. Observers in UI and World 7
3.2.2. Singleton 8
3.2.3. Factories for Roles and Actions 9
3.2.4. Adapter for viewport 9

4. Persistent data management **10**
4.1. Save Games 10

5. Quality **10**
5.1. Known Issues 10
5.2. Analytics 11
5.2.1. Dependencies 11
5.2.2. Quality Assurance 12

References **14**

A. Domain model diagram **i**

B. Class diagrams **i**
B.1. Models package i
B.2. Views package vii
B.3. Controllers package x
B.4. Listeners package xii
B.5. Utils package xiii

1. Introduction

Pawntastic is a game where a player is in control of a medieval-themed colony, and the goal is to survive for as long as possible. Threats such as natural disasters, raids from other groups of beings and starvation are always present. The game takes full advantage of object-oriented design. The current scope of the game allows the player to play locally with and against a simulation, the player is in control of building construction and role designation. The target audience is gamers that seek a niche genre that satisfies the need for simplicity and relaxation whilst also having a lurking threat to be vary of.

1.1. Definitions, acronyms, and abbreviations

Expansion: a set of User Stories that when completed marks a milestone.

Being: a singular independent entity in the world.

Pawn: a human-like Being who is a member of the Colony.

Being Group: a group of Beings that belong and act together.

Player: the user who is playing the game.

Colony: a Being Group of Pawns managed by the Player.

Action: a set of steps executed by a Being to perform a simple task.

Role: a giver of Actions in a specific order to fulfill a certain responsibility.

Terrain: a part of the underlying World that has specific characteristics, like how easily it can be traversed.

Structure: a building belonging to a Colony, existing in the World.

World: the environment the game takes place in.

Resource: a natural resource located in the in the World.

Item: collected from a Resource to be used for constructing Structures.

Inventory: a collection of Items.

HP: health points for either structure (structural integrity) or beings.

UI: User Interface, the part of the program that is responsible for communicating with the player.

NPC: Non Player Character, a Being that is not owned by nor managed by the player.

2. System architecture

The game is implemented as a standalone Java desktop application using the open source library libGDX [1] for user input and rendering. When the application is started a two-dimensional world is randomly generated. It consist of different terrain, collectable resources and a small group of pawns. The pawns belong to a colony. The player manages the colony by assigning roles to pawns, and by placing blueprints that the pawns can construct. This way they can build houses and other structures.

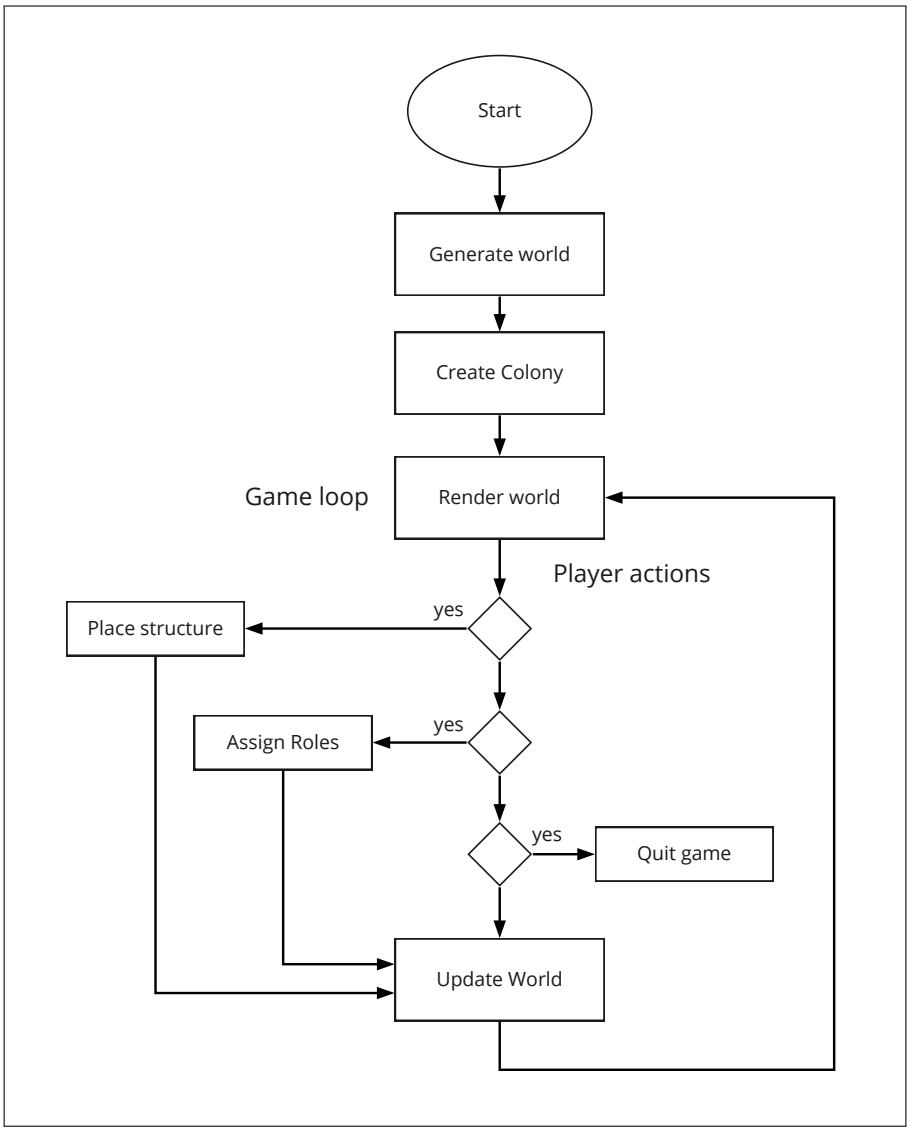


Figure 1: Top-level flow chart of how the game runs.

3. System design

The program uses the MVC design pattern [2], in our case with three packages; `models`, `views` and `controllers`. The application class `Pawntastic` uses these three packages, as seen in figure 2. The application class is responsible for instantiating and connecting the various parts.

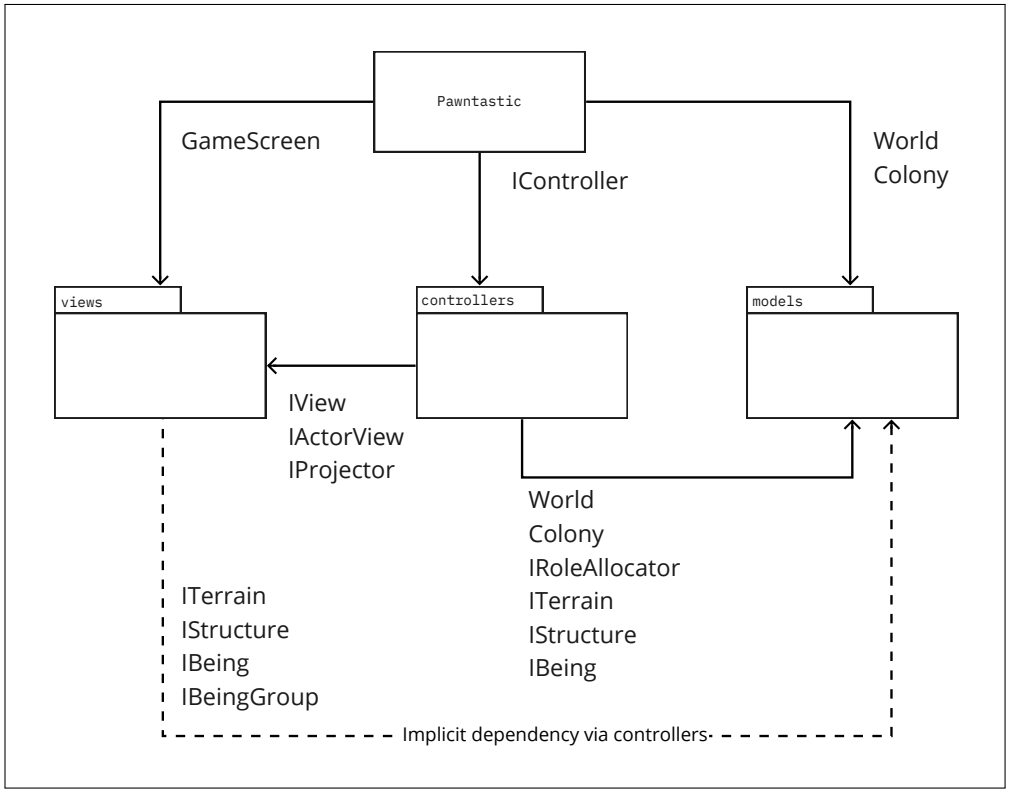


Figure 2: How the application class and the MVC packages relate to each other, including interfaces and classes used.

The `views` package consists of several views, all of which render their own part of the game using the listed interfaces. Each view has a corresponding controller in the `controllers` package, which is responsible for instantiating and updating the view with data from the model. Each controller handles the input of its view and processes this input before relaying it to the model.

The `models` package has been designed to have no outward dependencies on the `views` and `controllers` packages, nor on the library LibGDX. The application class instantiates and uses the classes `World` and `Colony`. There is no need for interfaces due to the fact that there is only one world and one colony in the application.

Between `controllers` and `models`, three interfaces and the classes, `World` and `Colony`, are used. `StructureController` needs access to the `World` object in order to place structures in the world. `RoleController` needs access to `Colony` as an `IRoleAllocator` in order to assign roles to `Pawn`s. The rest of the interfaces are used to bridge the communication and have low coupling between `models` and `views`.

In addition to these packages, there are two more: `com.thebois.listeners` and `com.thebois.utils`. `listeners` contains classes for event management that are used both in the model package and between different views and controllers. `utils` contains classes with useful methods for handling `Strings` and `matrices`.

Detailed UML class diagrams can be found in appendix B:

- `views`, figure B.10.
- `controllers`, figure B.14.
- `models`, figure B.2.
- `listeners`, figure B.17.
- `utils`, figure B.18.

3.1. Domain model and design model

The domain model, seen in figure 3, corresponds to the package `models` which is the design model. Because of its complexity it is composed of multiple sub-packages in the design model. This is illustrated with the colored boundaries in figure 4 and 5.

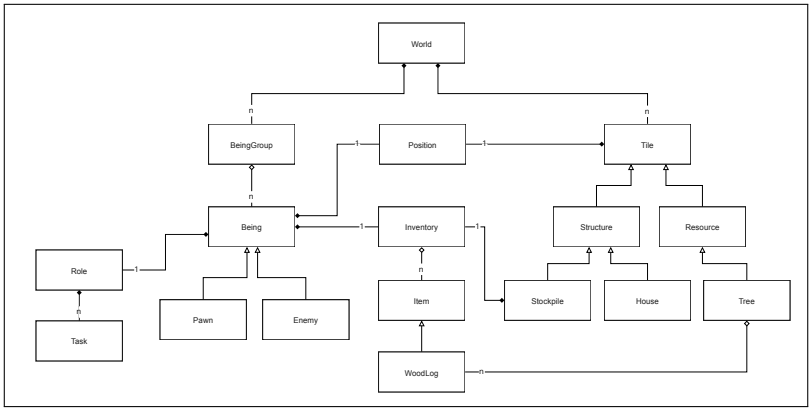


Figure 3: The domain model of Pawntastic.

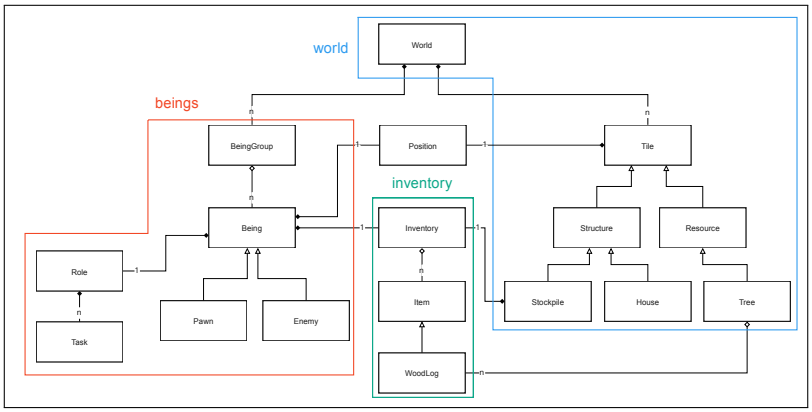


Figure 4: The domain model with package boundaries of the design model highlighted.

The different sub-packages in the design model communicate using a number of interfaces to make them loosely coupled. This can be seen in figure 5. The package also contains the class `Position` which is equivalent to Position in the domain models. `Position` is used by the game entities `ITile`s and `IBeing`s to give them a location in the game world. The `beings` package is dependent on the `world` package through the interfaces `ITile`, `ITerrain`, `IStructure` and `IWorld`. This is a one-to-one relation between the domain

model and the design model. The not yet implemented interface `IFinder`, will give beings the ability to locate resources and structures in the `World`. This will not be part of the domain model since its only a technical solution. The UML class diagram for `world` can be found in appendix B, figure B.3.

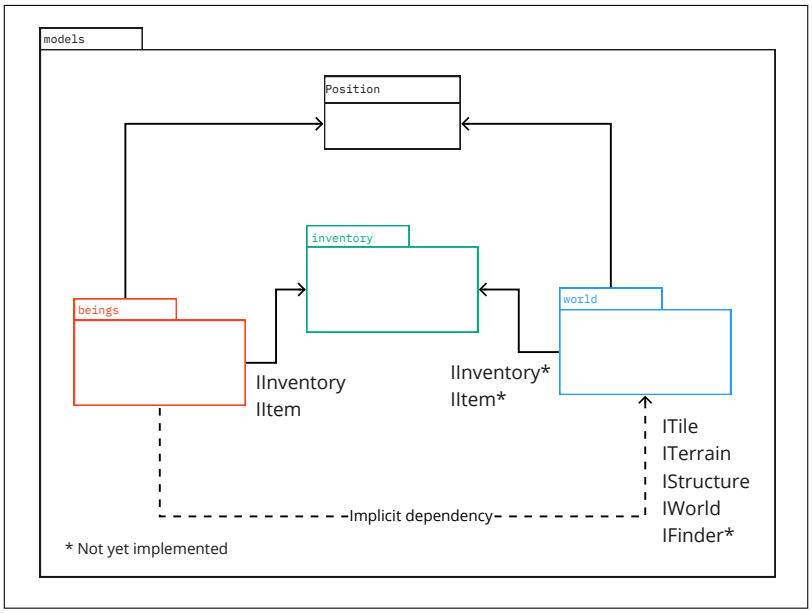


Figure 5: An overview of the packages in the design model.

The `beings` package contains two class hierarchies that implement `IBeing` and `IBeingGroup`, that directly correspond to the domain models Being and BeingGroup. The sub-package `roles` contains the different roles that can be assigned to an `IBeing`. The UML class diagram for `beings` and the sub-package `roles` can be found in appendix B, figures B.5 and B.6.

The `inventory` package is a small package directly mapped to the domain model and contain the interfaces `IInventory` and `IItem`. In the game it functions as a shared item inventory that can be used to build structures. The UML diagram for the package can be found in appendix B, figure B.8.

Dependencies that exist in the design model but are missing in the domain model are those that have to do with technical solutions, such as the use of an `IPathFinder`, which the Pawns use to navigate. Other missing dependencies are to the helper classes, in the packages `listeners` and `utils`, that exist outside of the model. For a complete view of the design model, see appendix B, figures B.7, B.17 and B.18.

3.2. Design patterns

A multitude of different design patterns have been implemented in the game, in order to improve code readability and functionality.

3.2.1. Observers in UI and World

Observers, in the form of events and event listeners, are used widely in the project. Both for UI elements, such as buttons, and for things occurring in the game world.

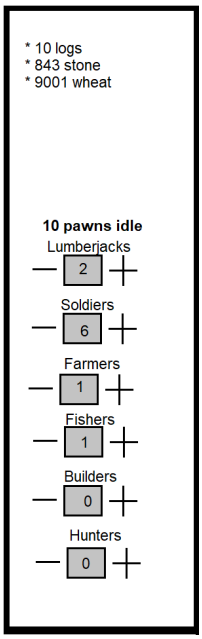


Figure 6: A sketch of the panel used to allocate pawn roles.

In the UI, the role allocation buttons use events to communicate state changes. The buttons are in the form of spinners, allowing for a simple increase and decrease of the number of pawns allocated to different roles, see figure 6. Whenever the value is changed, the button emits a value changed event to all its listeners. The controller is registered as a listener, which then updates the model with the new role allocations.

In the game, beings need to find paths to move to places in the world. They can not walk directly towards their destination as there might be structures, resources or terrain in the way. That is why they find a path free of obstacles and follow it, see figure 7. Therefore, they also need to be informed whenever a new obstacle is placed in the way of their current path. This is done through the event bus of the game. An obstacle placed event is generated by the world

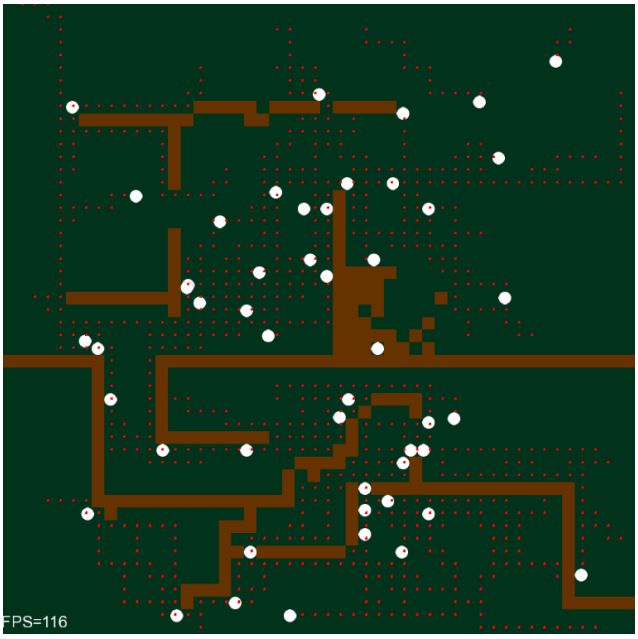


Figure 7: A debugging view of how each being follows its own path, avoiding the brown structures.

whenever a structure is placed, and each movement action listens to this event. When they receive the event, they check if the obstacle is in the way of the current path, and if it is, the path is regenerated to avoid the obstacle. If not, the path remains unchanged.

Events are used in both cases to ensure low coupling between the different components, while still allowing different components to respond to things happening in others.

3.2.2. Singleton

Singleton is a pattern that should be used very carefully, therefore it is only used for a single part of the game: the event bus, implemented using the Guava EventBus [3]. The reason for this is that events can be created and consumed anywhere in the model. Passing around an instance of the event bus is, while possible, not an ideal way of going about it.

Using a single instance of an event bus for the entire game is useful, because there is a single endpoint both for posting and listening to events. If using multiple buses, great confusion can be had when a component is listening or posting to the wrong bus.

3.2.3. Factories for Roles and Actions

Factory classes with static factory methods have been employed to reduce coupling between the consumers of `Role`s and `Action`s, see figure 8. The consumers are `Being`s and `BeingGroup`s.

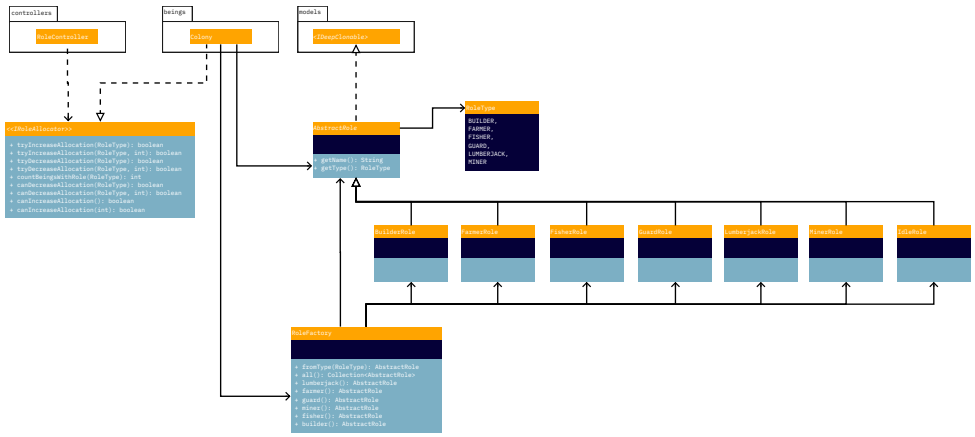


Figure 8: The package class diagram for roles. Notice how no outside dependencies are pointed at the implementations of AbstractRole.

Using a factory that only returns created objects through an abstraction means that the specific implementation behind each created object is hidden, meaning it can be switched out and modified without impacting consumers. This also allows each concrete implementation of Role and Action to be package private, entirely hidden from outside packages.

3.2.4. Adapter for viewport

The adapter pattern is used to make a part of the libGDX library [1] compatible with our project and to limit the amount of coupling between the library and the rest of our code.

One example of where the Adapter Pattern is used is in `StructureController`. The controller needs to know where the mouse click happened in the game world. However, when the player left-clicks, the controller only gets the screen coordinates of the click. In order to convert these screen coordinates to world coordinates, the `unproject` method of the `Viewport` is needed. Instead of giving the controller access to all `Viewport` methods, it is wrapped and this wrapper implements the `IProjector` interface that exposes the needed method.

4. Persistent data management

Sprites, fonts and style configuration data is located in the `./src/main/resources/` directory of the project. They are loaded into memory at the start of the game using libGDX [1] and do not change during runtime. Only one copy of each resource is stored in memory at any time, and they are disposed of before shutting down.

4.1. Save Games

Not implemented yet.

5. Quality

To minimize the risk of bugs occurring and to assure new features do not break code, the project utilizes JUnit [4]. The project also uses AssertJ [5] and Mockito [6] to make writing tests easier. The tests are located at `./src/test/java/com/thebois/`. The project also requires all developers to follow the same code style with the help of Checkstyle [7], this ensures style consistency across the whole project.

To automate tests and Codestyle-checks during development, the project first took advantage of Travis [8], but is currently using GitHub Actions [9]. The reason behind the transition was to conveniently and automatically create releases [10] of the game. Github also keep logs [11] of all previous action executions.

The final step for developers is creating pull requests, each pull request requires a screenshot of the latest test coverage with at least 90 percent coverage. It also requires at least two approved reviews, which ensures readable code and a better understanding of it throughout the entire team.

5.1. Known Issues

Since no code is bug-free nor flawless, Pawntastic does have some issues to this day, the *known* issues are as follows;

- Moving the game window will freeze the game temporarily.
- Pawns can get stuck in the terrain when creating a new world.
- Idle beings will still consider walking to spots they can not reach.

- Structures can be placed on beings.

5.2. Analytics

To ensure Pawntastic follows good code practices, a number of tools have been and is being used in the development of the project.

5.2.1. Dependencies

By continuously using Miro [12] for creating and updating UML class diagrams as the project evolved, problems and unwanted dependencies were easily identified and handled. All class diagrams seen in section 3 were created with Miro and as an example look at figure B.3, where all dependencies from and to the package can be easily traced.

Another usage for Checkstyle, the tool mentioned earlier, is by warning if a class has a high dependency usage. At one point the class `Pawntastic` was flagged for this exact problem, and because of that, two refactors were implemented to fix the issue [13] [14].

And the final tool for keeping dependencies in check is JDepend [15], which generates a report containing metrics such as extensibility and reusability, shown in figure 9.

Metric Results

[summary] [packages] [cycles] [explanations]

The following document contains the results of a JDepend metric analysis. The various metrics are defined at the bottom of this document.

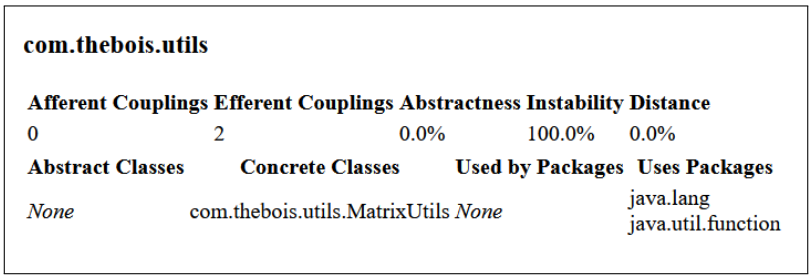
Summary

[summary] [packages] [cycles] [explanations]

Package	TC	CC	AC	Ca	Ce	A	I	D	V
com.thebois	2	2	0	2	16	0.0%	89.0%	11.0%	1
com.thebois.controllers	2	0	2	2	2	100.0%	50.0%	50.0%	1
com.thebois.controllers.game	3	3	0	1	8	0.0%	89.0%	11.0%	1
com.thebois.controllers.info	1	1	0	1	7	0.0%	88.0%	12.0%	1
com.thebois.listeners	2	0	2	0	2	100.0%	100.0%	100.0%	1
com.thebois.listeners.events	3	2	1	2	2	33.0%	50.0%	17.0%	1
com.thebois.models	2	0	2	9	1	100.0%	10.0%	10.0%	1
com.thebois.models.beings	5	1	4	4	8	80.0%	67.0%	47.0%	1
com.thebois.models.beings.pathfinding	4	3	1	2	4	25.0%	67.0%	8.0%	1
com.thebois.models.beings.roles	10	8	2	2	3	20.0%	60.000004%	20.0%	1
com.thebois.models.inventory	2	1	1	1	3	50.0%	75.0%	25.0%	1
com.thebois.models.inventory.items	4	3	1	2	1	25.0%	33.0%	42.0%	1
com.thebois.models.world	5	2	3	5	2	60.000004%	29.0%	11.0%	1
com.thebois.models.world.structures	4	2	2	1	4	50.0%	80.0%	30.000002%	1
com.thebois.utils	1	1	0	0	2	0.0%	100.0%	0.0%	1
com.thebois.views	5	4	1	4	12	20.0%	75.0%	5.0%	1
com.thebois.views.debug	1	1	0	0	8	0.0%	100.0%	0.0%	1
com.thebois.views.game	5	4	1	3	13	20.0%	81.0%	1.0%	1
com.thebois.views.info	4	3	1	2	7	25.0%	78.0%	3.0%	1

Figure 9: JDepend report of Pawntastic, generated on 2021-10-07.

One of the more interesting metrics is “Instability” (marked with “I” in figure 9). A higher instability percentage indicates that the package relies a lot on other packages, therefore more prone to change when the external packages are changed, and a lower percentage indicates the opposite. Though the report is not as bad as it seems, the metric is inflated by standard Java packages. An example of this is the `com.thebois.utils` package that has an instability of 100 percent, but as figure 10 shows, the only packages it uses are `java.lang` and `java.util.function`.



com.thebois.utils

Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
0	2	0.0%	100.0%	0.0%

Abstract Classes	Concrete Classes	Used by Packages	Uses Packages
None	com.thebois.utils.MatrixUtils	*None*	java.lang java.util.function

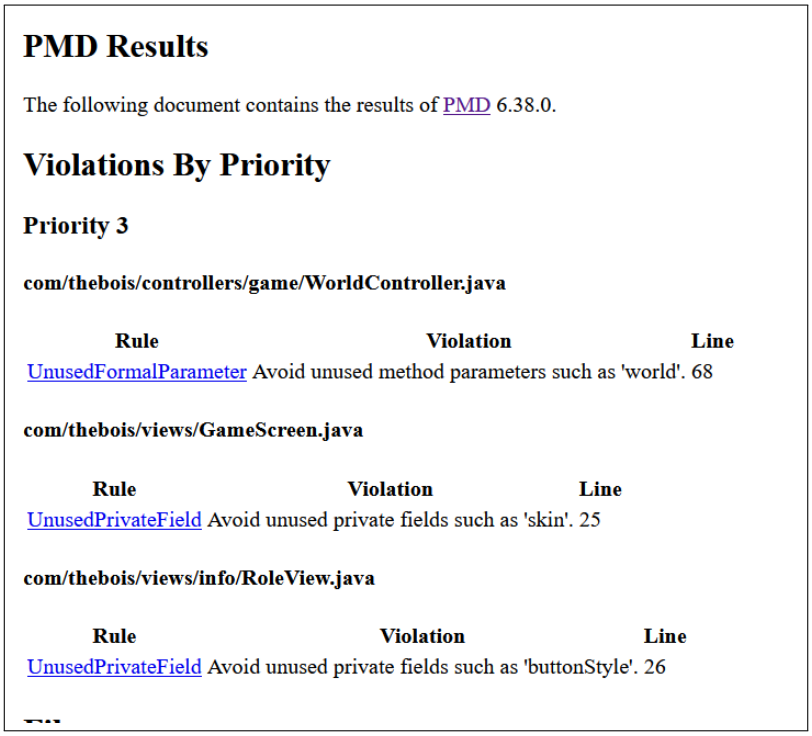
Figure 10: JDepend report of Utils package in Pawntastic, generated on 2021-10-07.

5.2.2. Quality Assurance

The last tool to be utilized is PMD [16], and it is being used in two ways. The first way is that it performs a code analysis, reporting any bad code practices and code smells. The latest report can be seen in figure 11, and as can be seen the violations at the moment are not serious and can be easily fixed.

The second way to use PMD is by using the ”Copy/Paste Detector” (CPD), as the name suggest it scans the project for any code that has been copied and pasted. For the latest report see figure 12.

As expected the CPD found nothing.



PMD Results

The following document contains the results of PMD 6.38.0.

Violations By Priority

Priority 3

com/thebois/controllers/game/WorldController.java

Rule	Violation	Line
UnusedFormalParameter	Avoid unused method parameters such as 'world'.	68

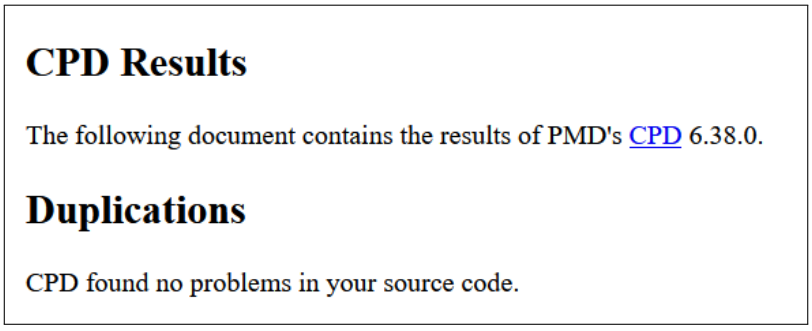
com/thebois/views/GameScreen.java

Rule	Violation	Line
UnusedPrivateField	Avoid unused private fields such as 'skin'.	25

com/thebois/views/info/RoleView.java

Rule	Violation	Line
UnusedPrivateField	Avoid unused private fields such as 'buttonStyle'.	26

Figure 11: PMD report of Pawntastic, generated on 2021-10-07.



CPD Results

The following document contains the results of PMD's CPD 6.38.0.

Duplications

CPD found no problems in your source code.

Figure 12: CPD report of Pawntastic, generated on 2021-10-07.

References

[1] *libGDX*, ver. 1.10.0, Apr. 2021. [Online]. Available: https://libgdx.com/, visited on 09/29/2021.

[2] GeeksforGeeks, *Geeksforgeeks - mvc design pattern*, Feb. 2018. [Online]. Available: https://www.geeksforgeeks.org/mvc-design-pattern/, visited on 10/10/2021.

[3] Google, *Guava*, ver. 31.0-jre, Sep. 2021. [Online]. Available: https://guava.dev/, visited on 09/29/2021.

[4] J. Team, *JUnit*, ver. 5.7.0, Sep. 2020. [Online]. Available: https://junit.org/junit5/, visited on 09/29/2021.

[5] *AssertJ - fluent assertions java library*, ver. 3.20.2, Jun. 2021. [Online]. Available: https://assertj.github.io/doc/, visited on 09/29/2021.

[6] *Mockito framework*, ver. 3.12.4, Aug. 2021. [Online]. Available: https://site.mockito.org/, visited on 09/29/2021.

[7] The Apache Software Foundation, *Apache Maven Checkstyle Plugin*, ver. 3.1.2, Jan. 2021. [Online]. Available: http://maven.apache.org/plugins/maven-checkstyle-plugin/, visited on 09/29/2021.

[8] *Travis CI*. [Online]. Available: https://www.travis-ci.com/, visited on 09/29/2021.

[9] Microsoft, *GitHub Actions*. [Online]. Available: https://docs.github.com/en/actions, visited on 10/07/2021.

[10] *Releases*. [Online]. Available: https://github.com/martinjonsson01/OOPP-WITH-THE-BOIS/releases, visited on 10/04/2021.

[11] *Github actions history*. [Online]. Available: https://github.com/martinjonsson01/OOPP-WITH-THE-BOIS/actions, visited on 10/07/2021.

[12] *Miro*, 2021. [Online]. Available: https://miro.com/, visited on 09/29/2021.

[13] *Pull request 54*. [Online]. Available: https://github.com/martinjonsson01/OOPP-WITH-THE-BOIS/pull/54, visited on 10/07/2021.

[14] *Pull request 55*. [Online]. Available: https://github.com/martinjonsson01/OOPP-WITH-THE-BOIS/pull/55, visited on 10/07/2021.

[15] M. Clark, *Jdepend*, ver. 2.10, Mar. 2020. [Online]. Available: https://github.com/clarkware/jdepend, visited on 09/29/2021.

[16] PMD, *Pmd*, ver. 6.39.0, Sep. 2021. [Online]. Available: https://pmd.github.io/, visited on 09/29/2021.

A. Domain model diagram

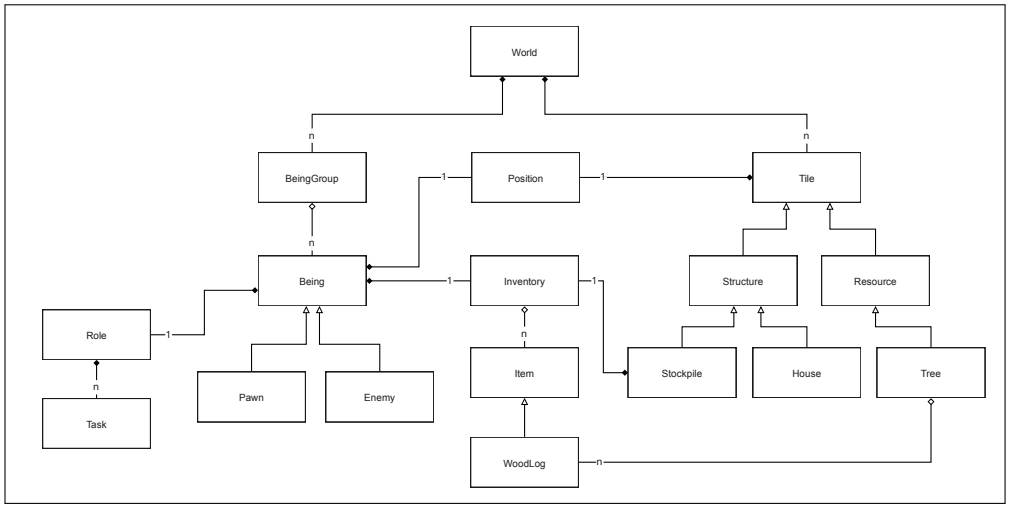


Figure A.1: Domain model

B. Class diagrams

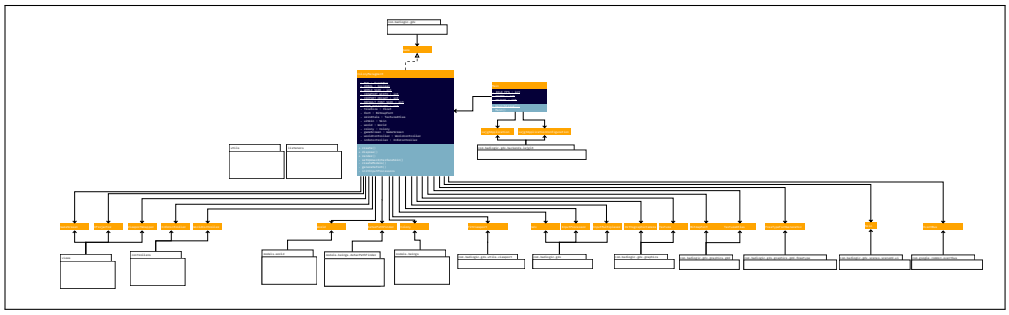


Figure B.1: Top-level UML class diagram of the application.

B.1. Models package

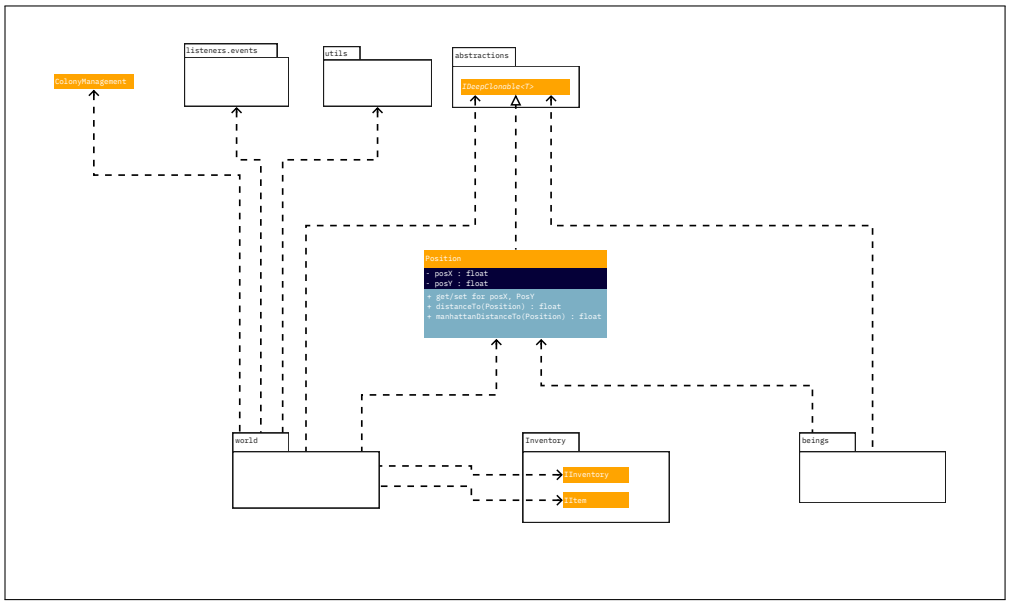


Figure B.2: UML class diagram of the `models` package.

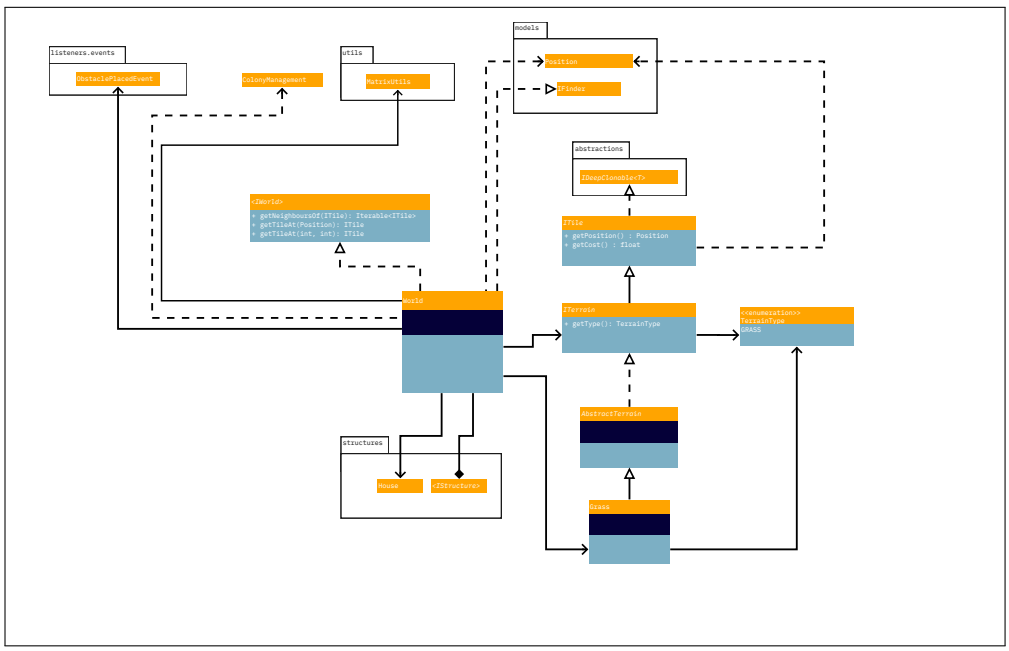


Figure B.3: UML class diagram of the `world` package.

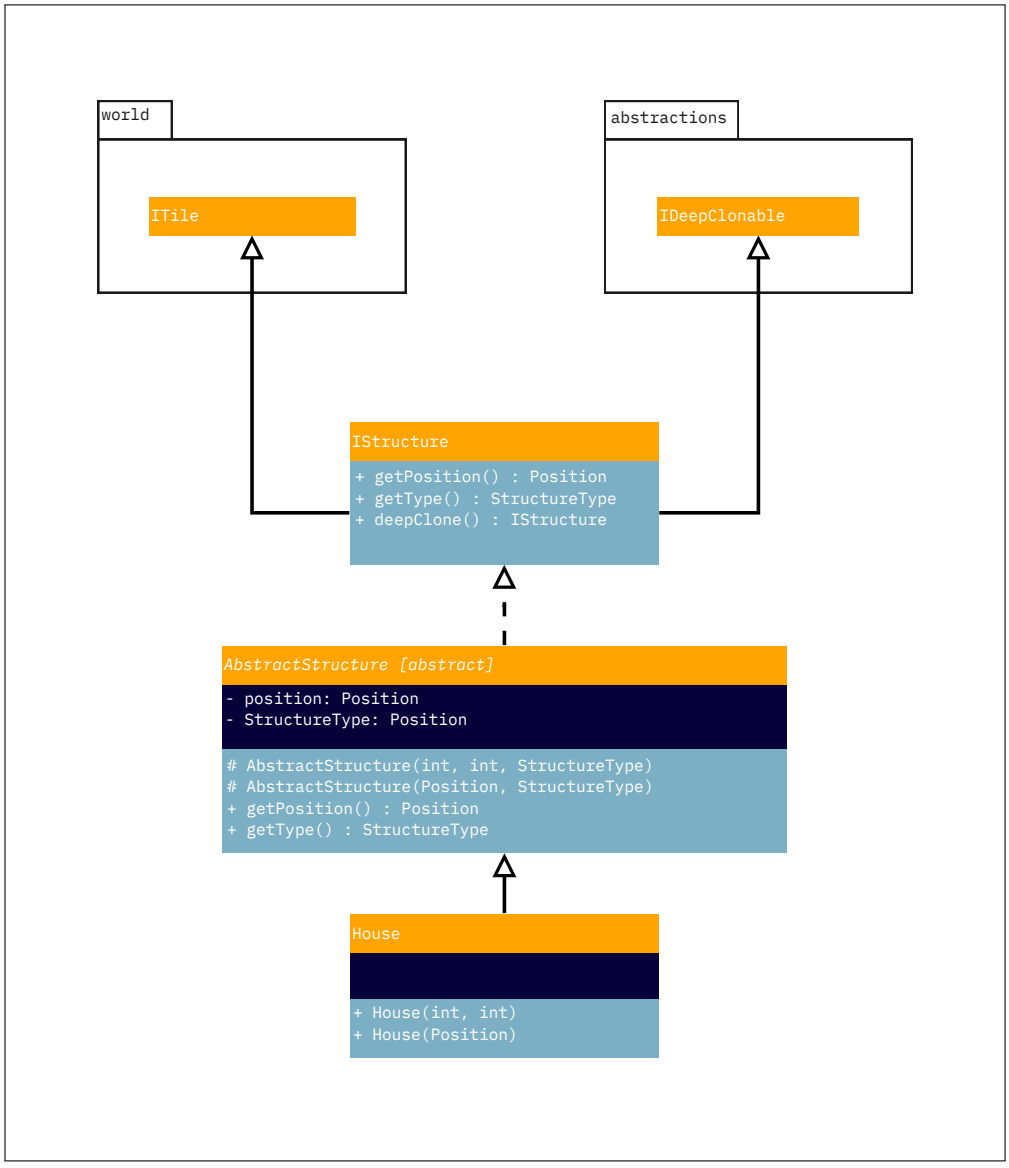


Figure B.4: UML class diagram of the `structures` package.

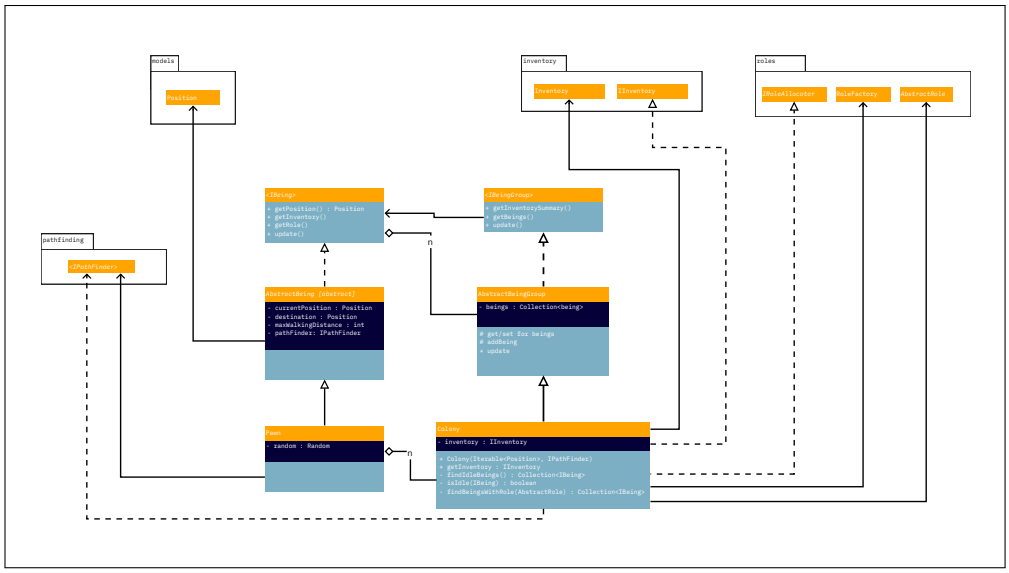


Figure B.5: UML class diagram of the `beings` package.

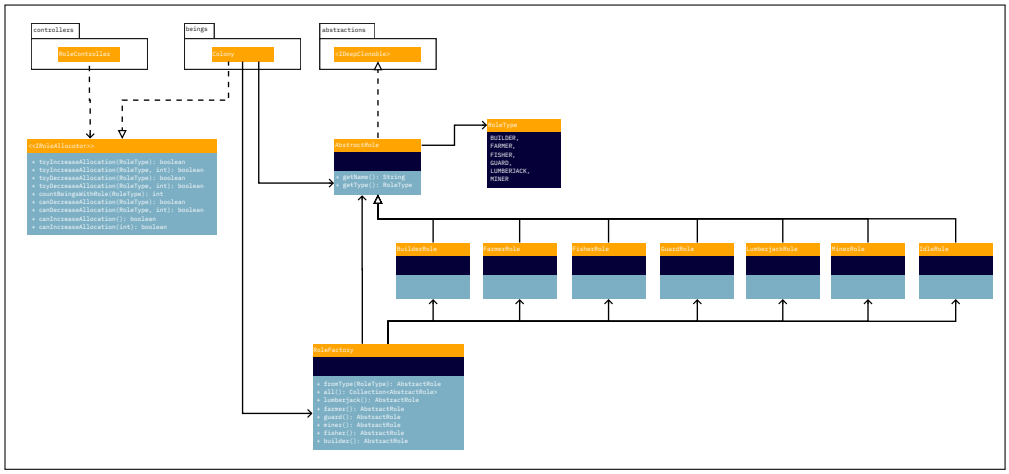


Figure B.6: UML class diagram of the `roles` package.

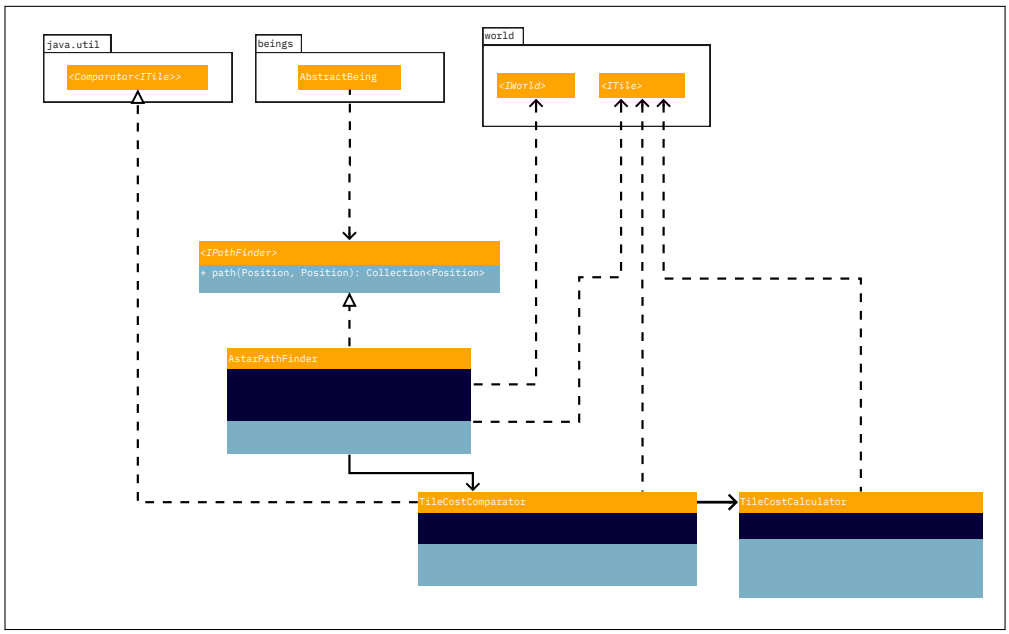


Figure B.7: UML class diagram of the `pathfinding` package.

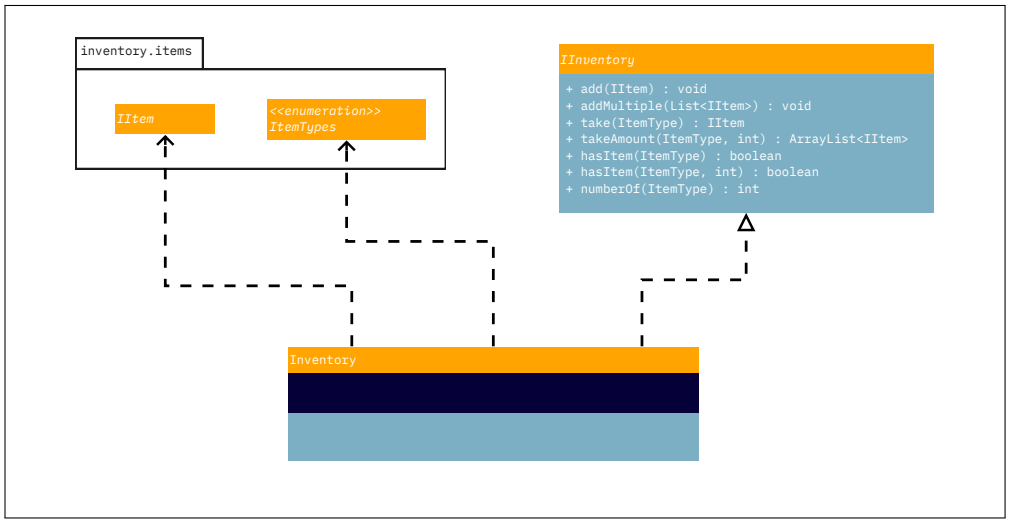


Figure B.8: UML class diagram of the `inventory` package.

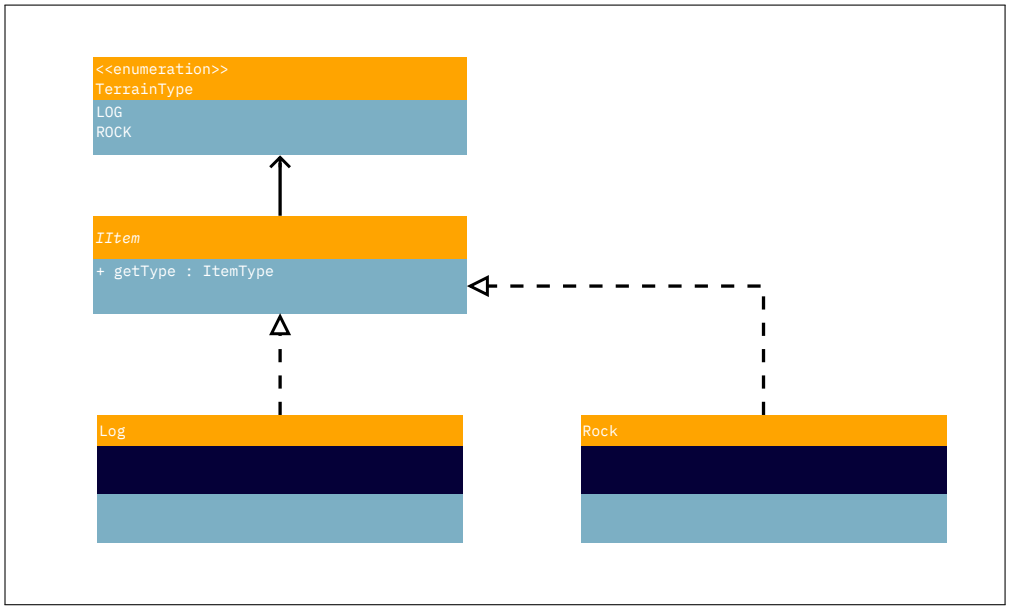


Figure B.9: UML class diagram of the `items` package.

B.2. Views package

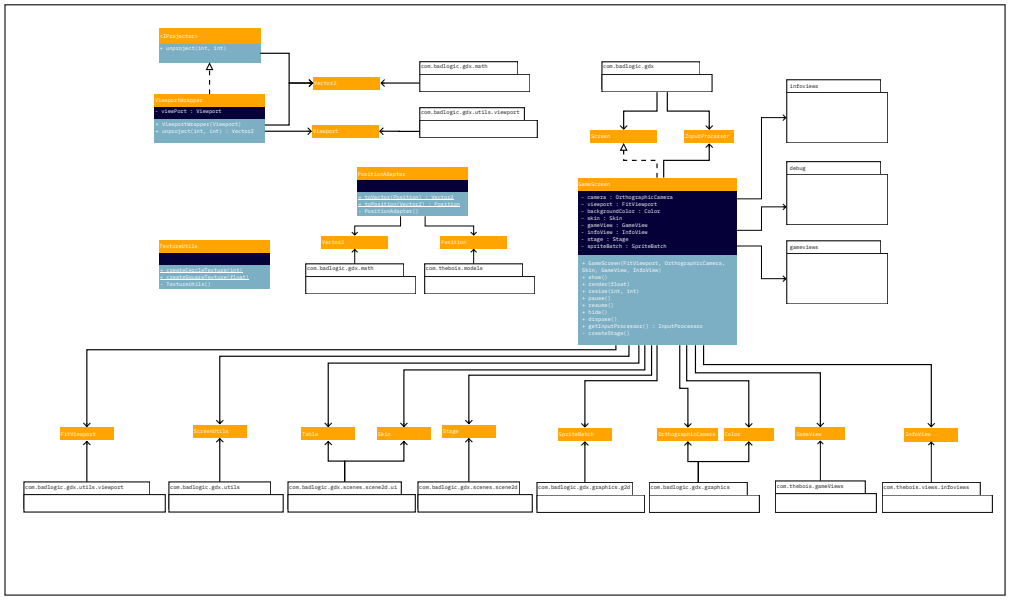


Figure B.10: UML class diagram of the `views` package.

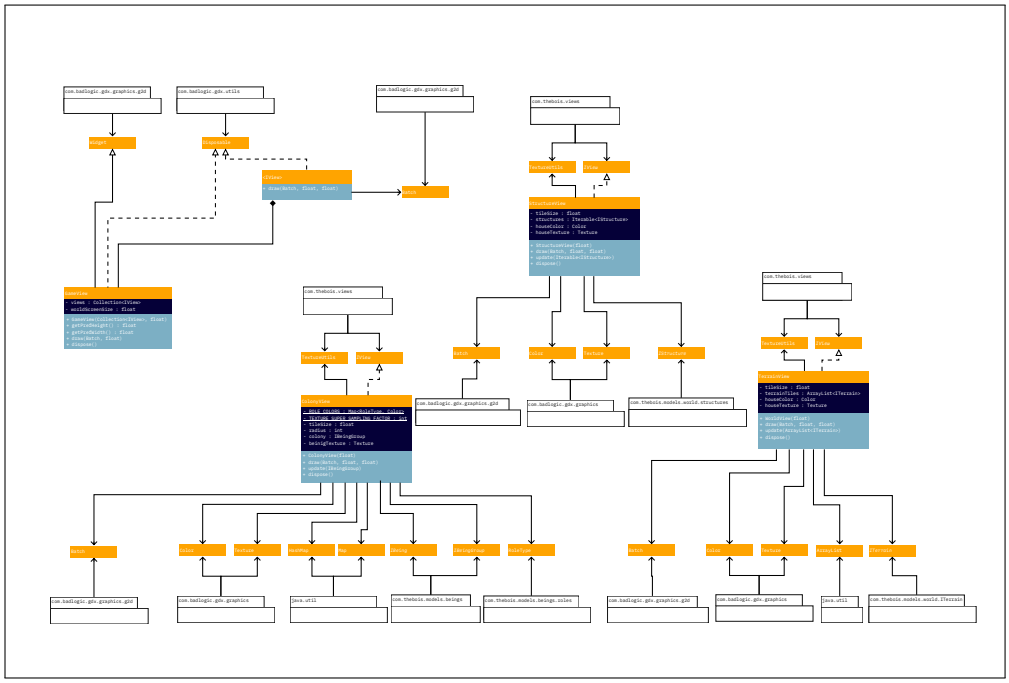


Figure B.11: UML class diagram of the `gameviews` package.

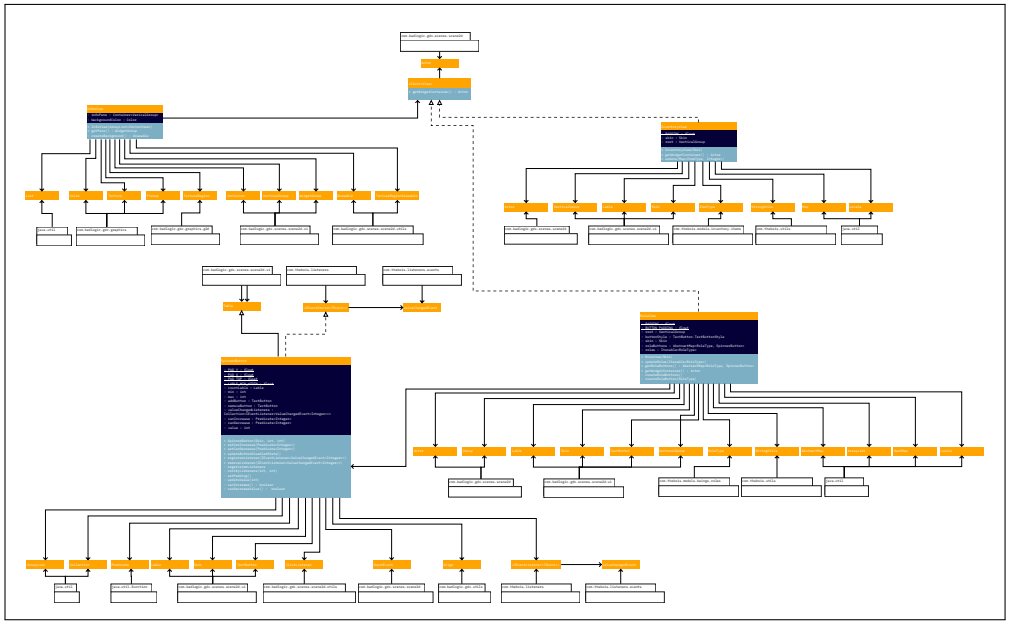


Figure B.12: UML class diagram of the `infoviews` package.

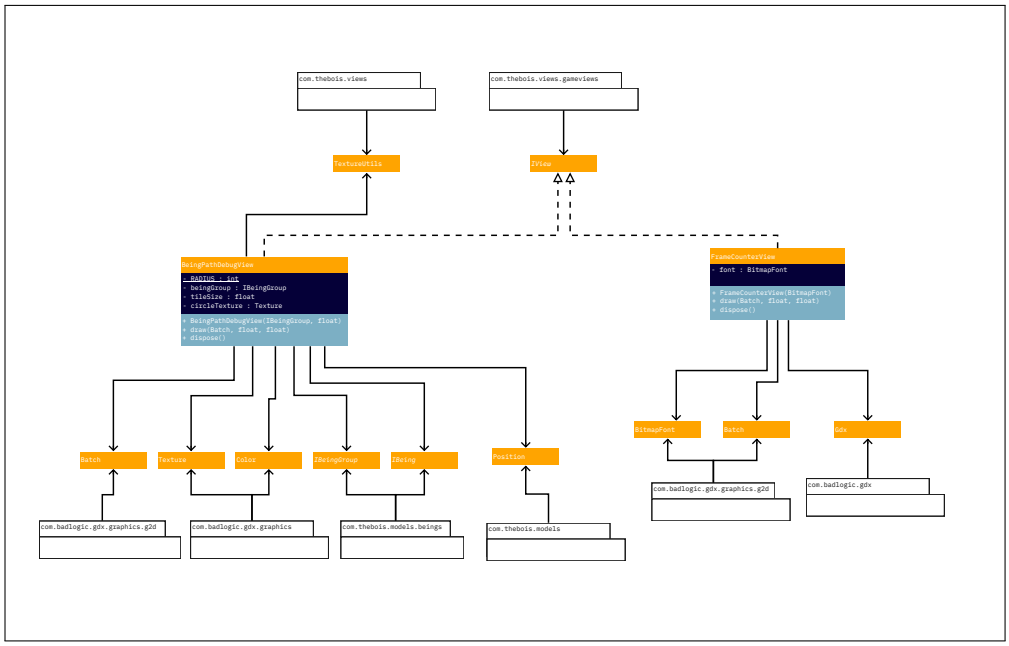


Figure B.13: UML class diagram of the `debugviews` package.

B.3. Controllers package

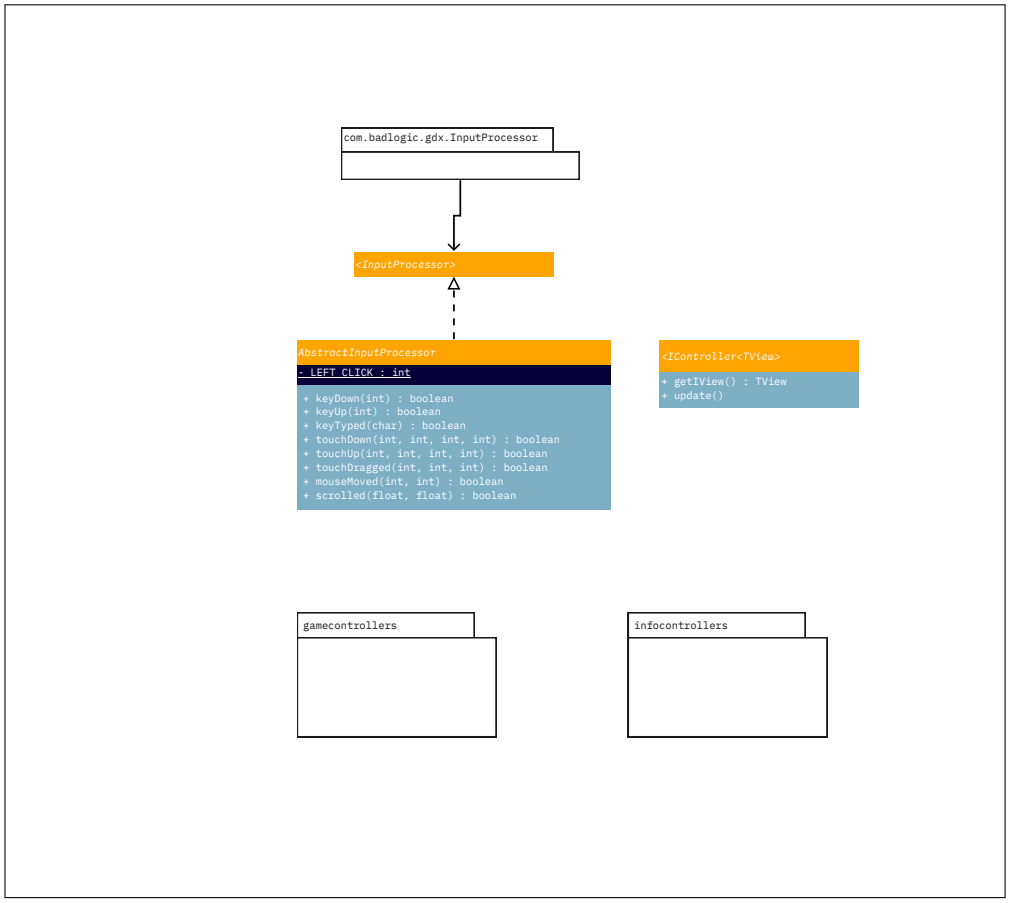


Figure B.14: UML class diagram of the `controllers` package.

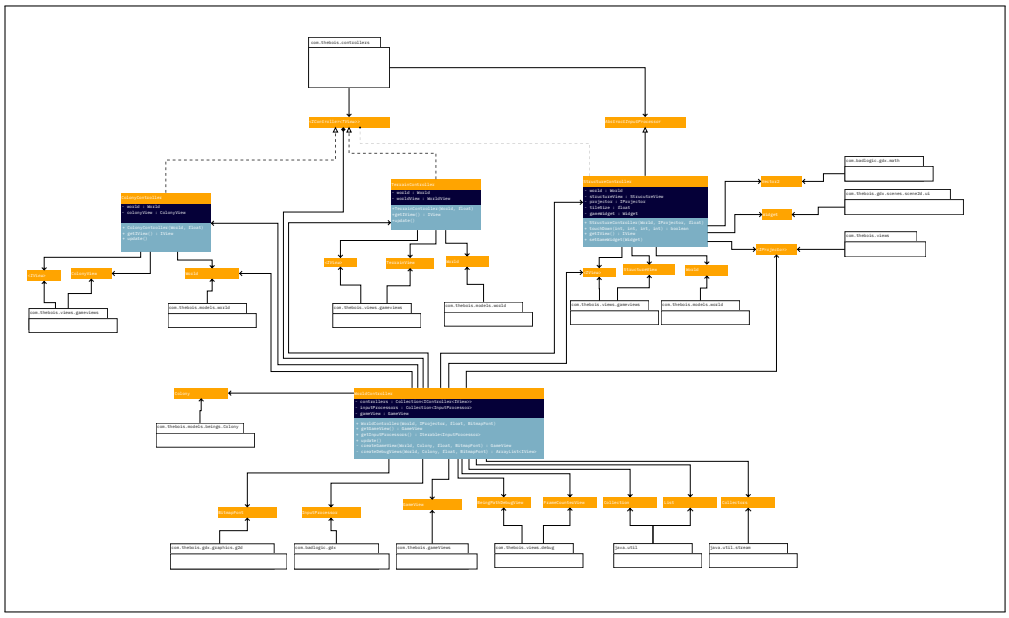


Figure B.15: UML class diagram of the `gamecontrollers` package.

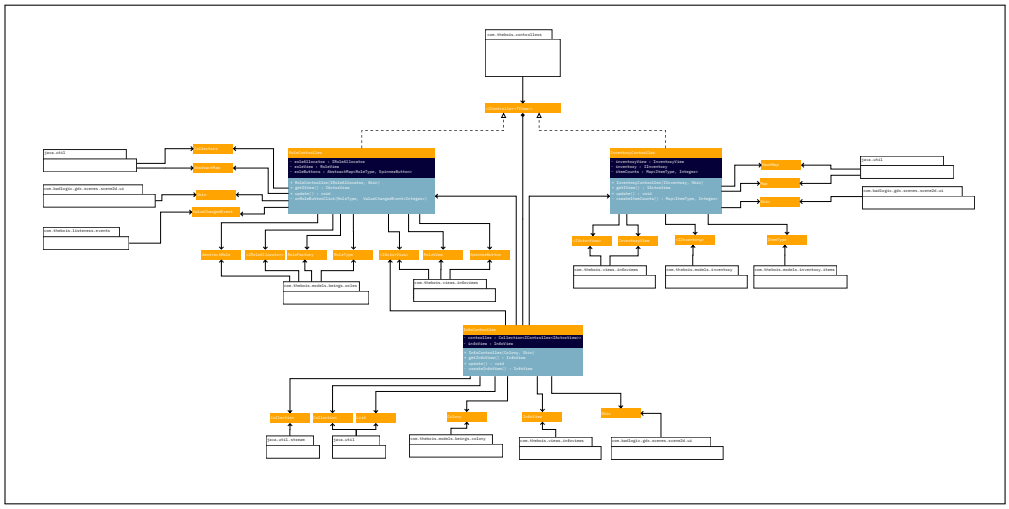


Figure B.16: UML class diagram of the `infocontrollers` package.

B.4. Listeners package

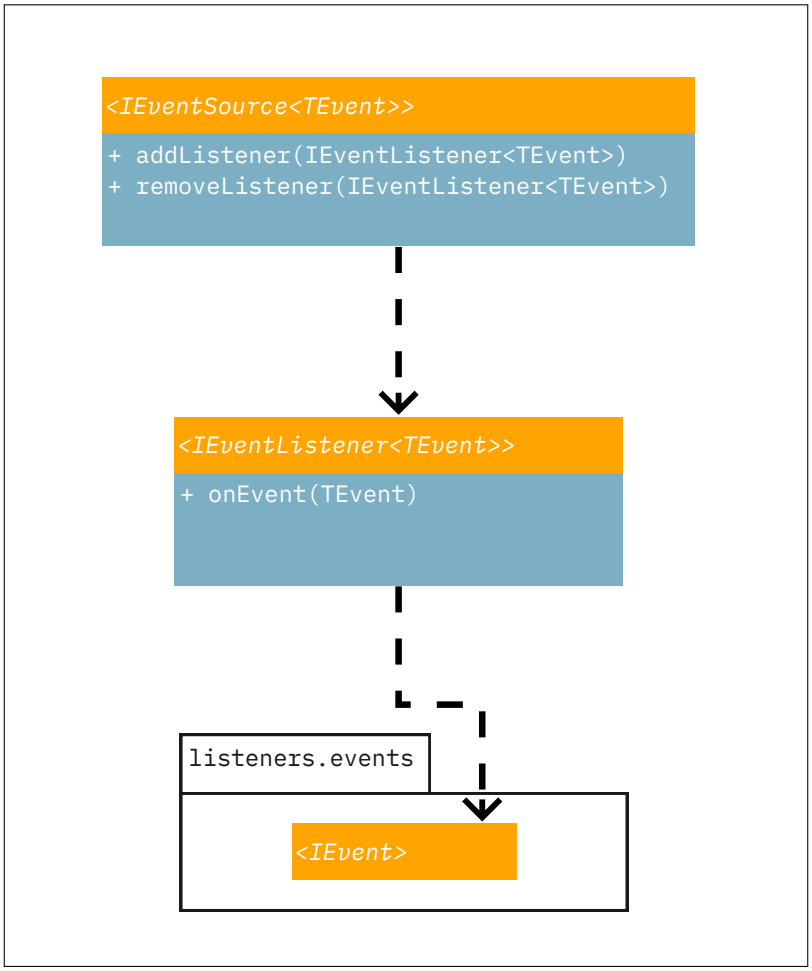


Figure B.17: UML class diagram of the `listeners` package.

B.5. Utils package

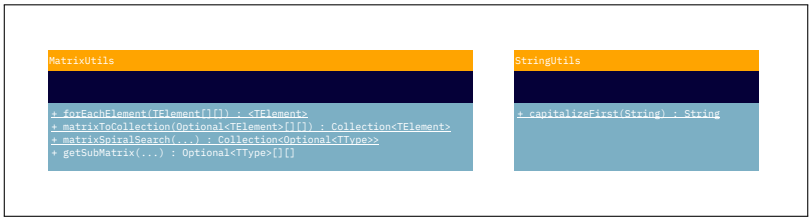


Figure B.18: UML class diagram of the `utils` package.