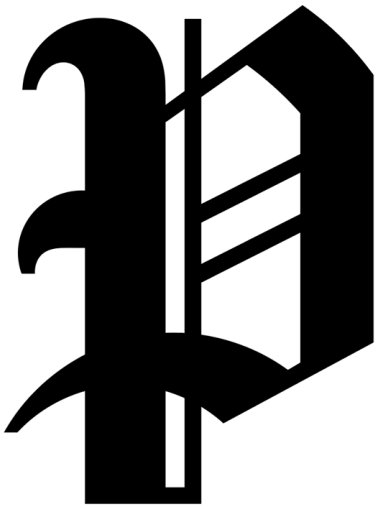


System Design Document for Pawntastic

Jacob Bredin, Martin Jonsson,
Jonathan Lindqvist, Mathias Prétot

October 24, 2021

Contents

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 against a simulation, the player is in control of construction locations 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.

Tile: A section of the world containing a single object.

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, health 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.

Javadoc: A documentation tool for Java.

CPD: Copy/Paste Detector, a tool from PMD that checks if there is any copied and pasted code for the project.

SOLID: The SOLID principles, single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle.

MVC: Model View Controller, a software design pattern for separating the applications business logic (model) from the presentation logic (view) and user-input logic (controller).

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, if a save-file exists it is loaded instead. The game world 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.

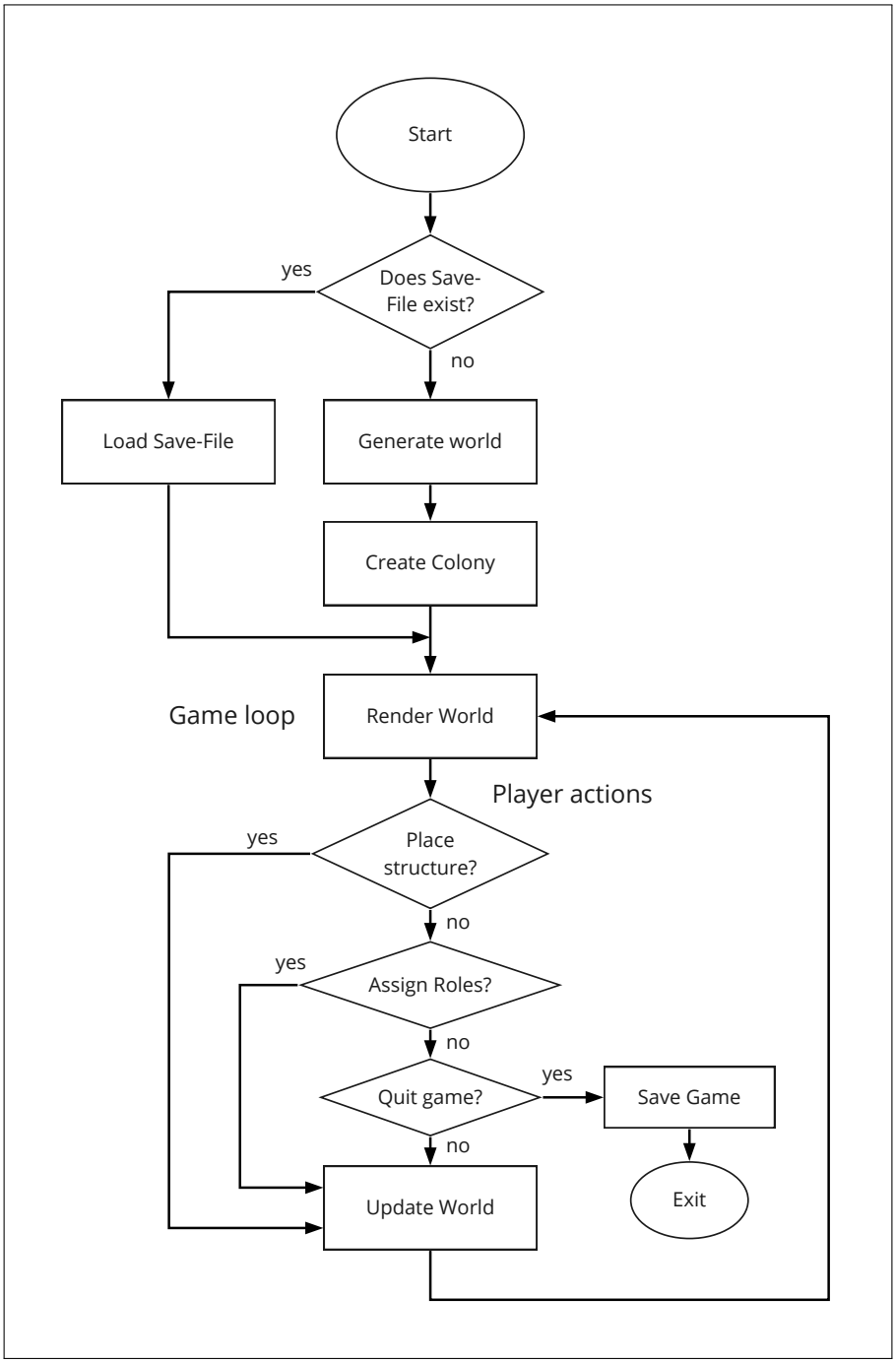


Figure 1: Top-level flow chart of how the game runs.

3. System design

The program uses the MVC design pattern, 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.

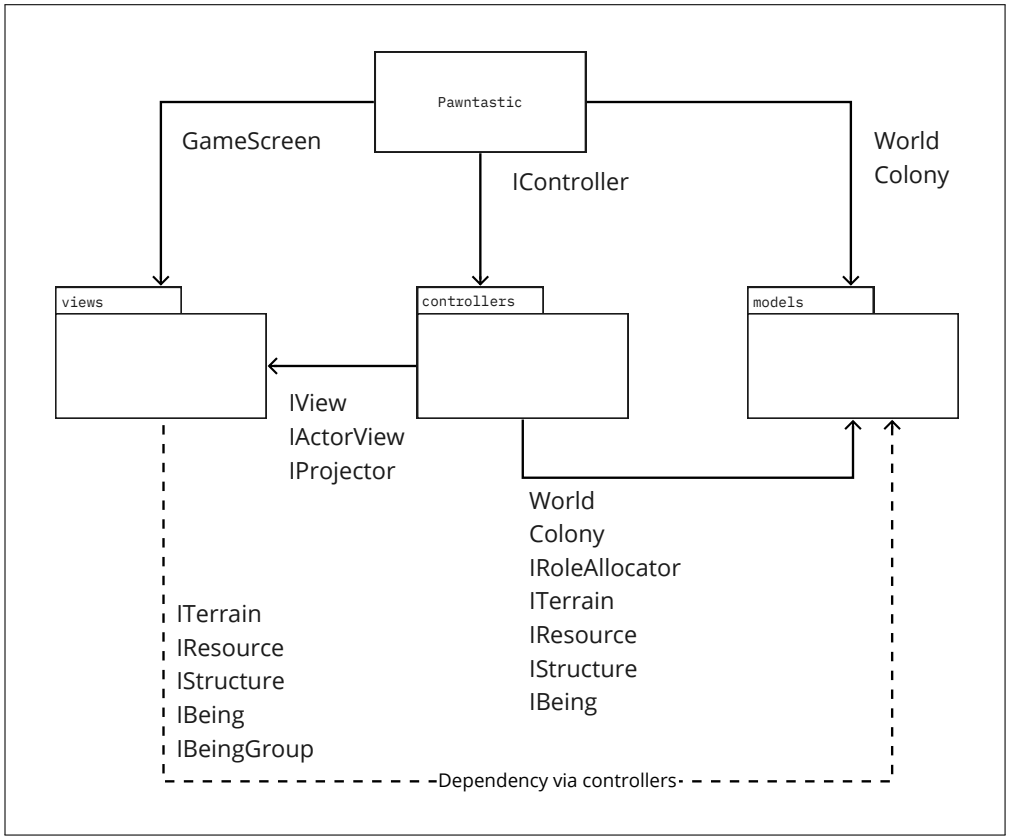


Figure 2: How the application class and the MVC packages relate to each other, including interfaces and classes used.

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`.

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. This is illustrated with the colored boundaries in figure 4 and 5.

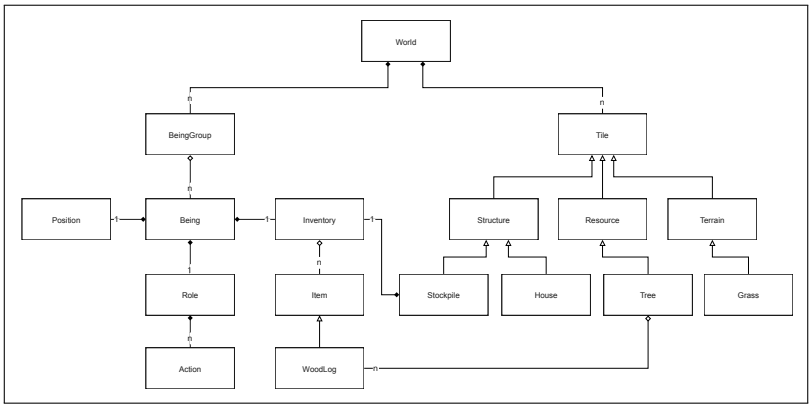


Figure 3: The domain model of Pawntastic.

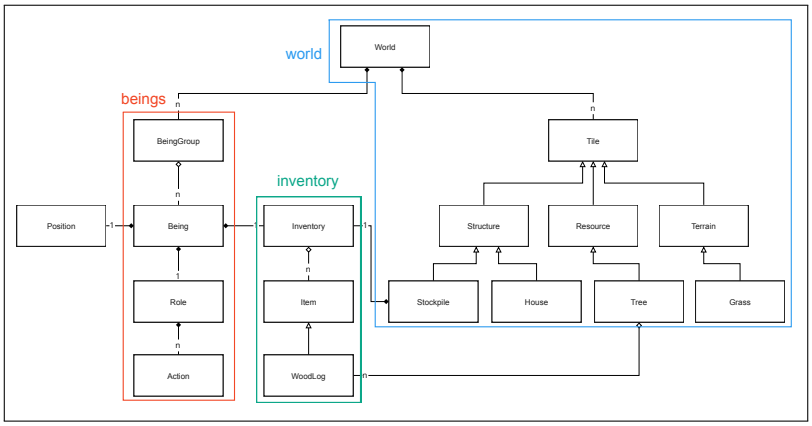


Figure 4: The domain model with package boundaries of the design model highlighted.

The different 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` ,

`IResource`, `IStructure` and `IWorld`. This is a direct conformance between the domain model and the design model.

`World` implements three other interfaces, these are `IResourceFinder`, `IStructureFinder` and `IPositionFinder`, and are used by `Roles` and `Colony` to be able to find different types of tiles in the `World`. This is done to have no direct dependency to the `World` class and allows for moving the implementation to another class in the future. Since this is only a technical solution they do not appear in the domain model.

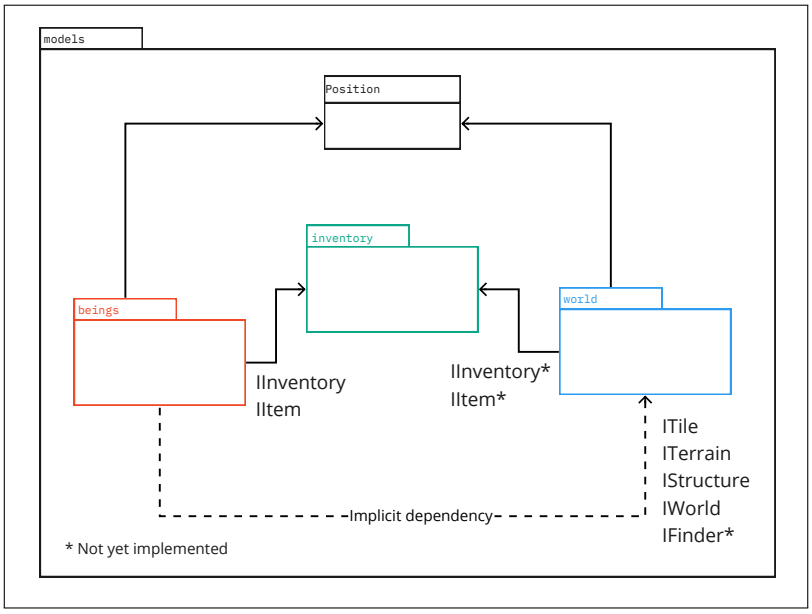


Figure 5: An overview of the packages in the design model.

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 the packages `beings` and `roles` can be found in appendix A.

The `inventory` package is a small package directly mapped to the domain model and contain the interfaces `IInventory` and `IItem`. `IInventory` extends two other interfaces, `IStorable` and `ITakeable`. These interfaces makes it easy to know if for example a structure can store items or if items can be taken. Each `Being` has their own Inventory that is used for storing materials or food. The UML diagram for the package can be found in appendix A.

Types 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 `Being` 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 A.

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 are used to ensure low coupling between different components, while still allowing them to respond to what is happening in others. Here, they are implemented using events in two different ways. Both for UI elements, such as buttons, and for some of what is occurring in the game world.

Figure 6: 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 `ValueChangedEvent` event to all its listeners. A controller, registered as a listener to this event, updates the model in response to the event with the new role allocations.

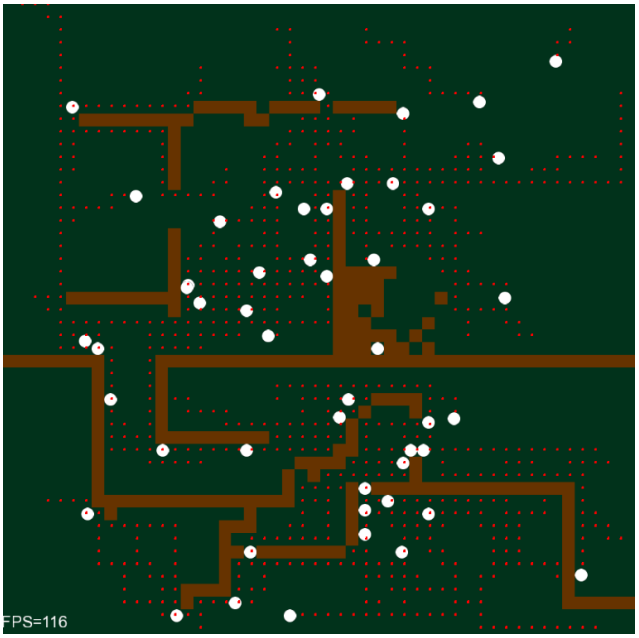


Figure 7: A debugging view of how each being follows its own path, avoiding the brown structures.

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 `ObstaclePlacedEvent` is generated by the `World` whenever an `IStructure` is placed, and each `MoveAction` 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 recalculated to avoid the obstacle.

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 [2]. The reason for this is that events can be created and consumed anywhere in the model.

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.

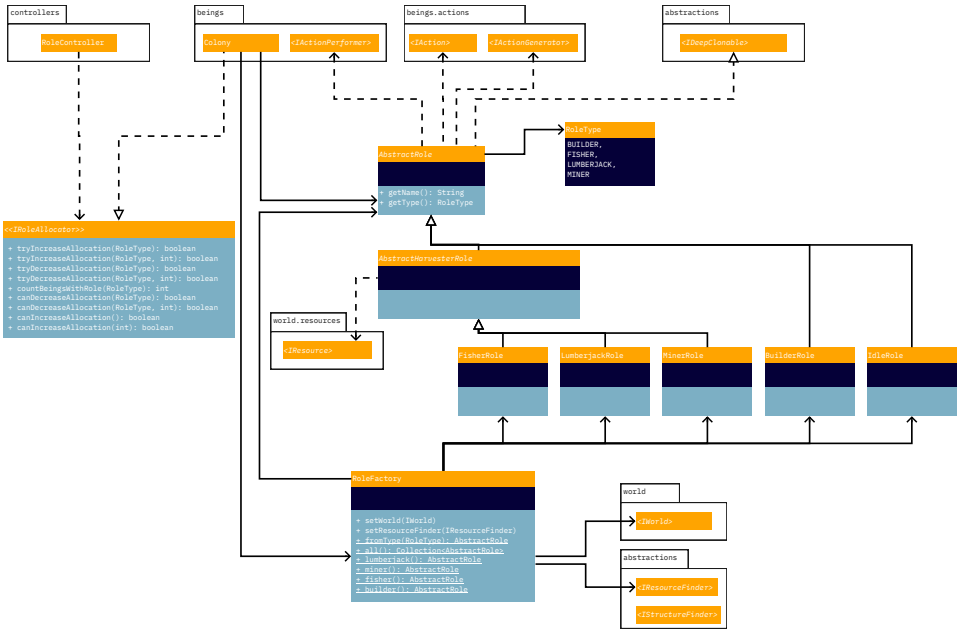


Figure 8: The package class diagram for roles. Notice how no outside dependencies are pointed at the implementations of AbstractRole.

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. Facade for viewport

The facade pattern is used to make complex classes more simple with the use of interfaces. One example of where the facade is used is in

`PlaceStructureController`. 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`, from the libGDX library [1], 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. This also make the `PlaceStructureController` less coupled with the library as the controller only knows about the interface `IProjector` and not the `Viewport`.

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

Pawntastic saves the game state to a file, under `(home directory)/Documents/Pawntastic/saves/`, whenever the game is closed. To achieve this the project utilizes Java's implementation of serialization. All objects that are created separately such as `World`, `Colony` and `Inventory` are explicitly told to be serialized/deserialized. All other objects are contained within them such as `AbstractBeings` and `Items` and they are handled automatically.

5. Quality

To minimize the risk of bugs occurring and to assure new features do not break code, the project utilizes JUnit[3] (with over 600 tests). The project also uses AssertJ [4] and Mockito [5] 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 [6], this ensures style consistency across the whole project.

To automate tests and Codestyle-checks during development, the project first took advantage of Travis [7], but is currently using GitHub Actions [8]. The reason behind the transition was to conveniently and automatically create releases [9] of the game. Github also keep logs [10] of all previous action executions.

The final step for developers are pull requests. For a pull request to be approved test coverage of least 90% coverage is required. It also requires at least two approved reviews, which ensures readable code and a better understanding of it throughout the entire team.

Currently the project is at 100% test coverage but IntelliJ is bugged and reports some classes as less than 80%. But the command `mvn verify`, which runs on all pull requests, is configured to ensure that all methods are tested, it also forces test coverage of at least 90%.

5.1. Known Issues

- 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.
- Beings do not take each other into account when moving or performing roles, so they often overlap.
- Many systems, like pathfinding and resource finding, have not been optimized to work efficiently with worlds larger than 50x50. The systems still function at larger world sizes, but performance deteriorates quickly.

5.2. Analytics

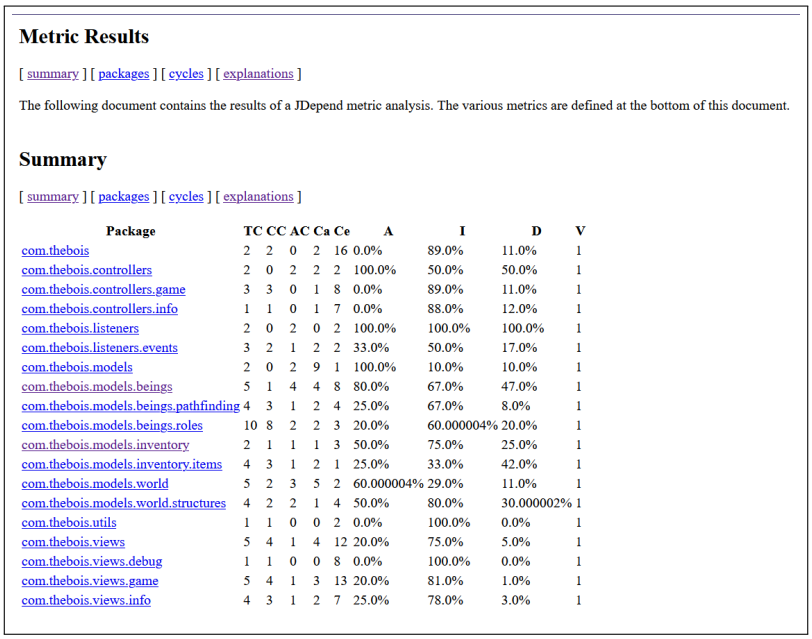
To ensure Pawntastic follows good code practices, a number of tools have been and is being used in the development of the project.

5.2.1. Dependencies

By continuously using Miro [11] 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 3, where all dependencies from and to the package can be easily traced.

Checkstyle got more to offer than just making sure the style is consistent, for example it forces Javadoc on all public classes and methods. But it also warns if a class has high coupling. At one point the class `Pawntastic` was flagged for having high coupling, and because of that, two refactors were performed to fix the issue [12] [13].

The final tool for keeping dependencies in check is JDepend [14], 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.

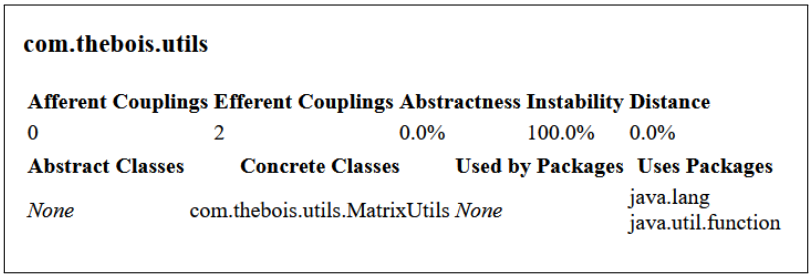
Summary

[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, and it is therefore more prone to change when the external packages are changed. Though what the report says about the project 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%, but as figure 10 shows, the only packages it uses are `java.lang` and `java.util.function`.



com.thebois.utils

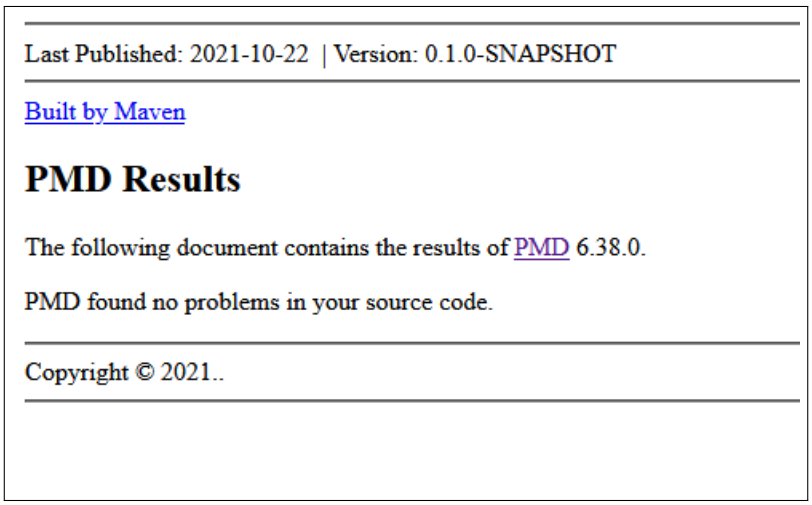
Afferent Couplings	Efferent Couplings	Abstractness	Instability	Distance
0	2	0.0%	100.0%	0.0%

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 [15], 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.



Last Published: 2021-10-22 | Version: 0.1.0-SNAPSHOT

Built by Maven

PMD Results

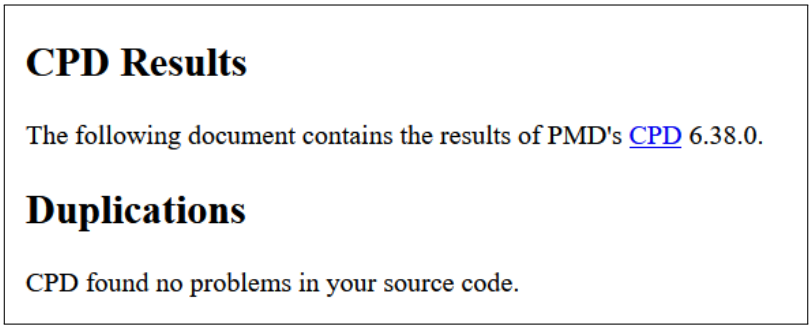
The following document contains the results of PMD 6.38.0.

PMD found no problems in your source code.

Copyright © 2021..

Figure 11: PMD report of Pawntastic, generated on 2021-10-22.

The second way to use PMD is by running the CPD tool, it scans the project for any code that has been copied and pasted. For the latest report see figure 12.



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 2021-10-22.

6. Further work

Whenever a part of the code was identified that could be improved upon, to better follow different design patterns and SOLID, a refactor was conducted to apply that improvement. Because of time constraints, a few improvements that were discovered could not be implemented. This section discusses some improvements that should be implemented and potential plans for how they could be implemented.

6.1. Refactor ITerrain, IResource and IStructure

The interfaces `ITerrain`, `IResource` and `IStructure` all share a similar method, `getType`. The method should be moved to `ITile`, though the move is not that simple. All the interfaces return a different types, for example `ITerrain`s `getType` returns a `TerrainType`, and `IStructure`s retuns a `StructureType`. Though there is a possible solution for this, all the type enums could be one enum, `TileType`. This way `ITerrain` and `IStructure` would disappear.

6.2. IController Segregation

To better follow the inteface segregation principle, `IController` should be split into two interfaces, because at the moment `RoleController` has no use for the `update` method. With `IUpdate` and `IViewSource` as the new interfaces, `RoleController` would no longer need to implement an empty method.

6.3. Double-Abstractions

In the peer review of the code, a common theme of unnecessary double-abstractions was identified. It was pointed out that these were unnecessary, so many of them were removed. One still remains.

A double-abstraction is when both an abstract class and an interface is used for the same purpose. For example, `AbstractBeingGroup` is an abstract class that implements the `IBeingGroup` interface. Either `IBeingGroup` is unnecessary, or `AbstractBeingGroup` should not be abstract. In this case, `IBeingGroup` should most likely be segregated into multiple different interfaces for each different group of behaviours that `AbstractBeingGroup` implements.

6.4. Running UI and model in separate threads

The model is currently updated in the same thread that renders the game. By updating the model in a separate thread it would be possible to render the graphics and update the model at different rates. To avoid concurrency issues between the two threads, the model would not be allowed to modify data that the view is simultaneously reading. This could be done by using resource locking [16], either on each collection or by using resource locking on individual elements in the collections. The element locking approach demands that no elements are removed, only added to the end of the collection. If elements need to be removed, the whole collection needs to be locked during this remove operation.

By further multi-threading of the internals of the model, such as the pathfinder, a smoother update of the UI could be achieved. Currently the pathfinder might take more time than the duration of a single frame, which will freeze the UI. Since there is no need for the pathfinder to calculate new paths in this short of a time frame this could be done in a separate thread and once calculated update the object requesting a new path. Today this is not a major issue but as the complexity of the game increases over time, multi-threading will become important.

References

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

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

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

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

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

[6] 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.

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

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

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

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

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

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

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

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

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

[16] Baeldung, *Guide to java.util.concurrent.locks*, 2020. [Online]. Available: https://www.baeldung.com/java-concurrent-locks, visited on 10/24/2021.

A. UML Class diagrams

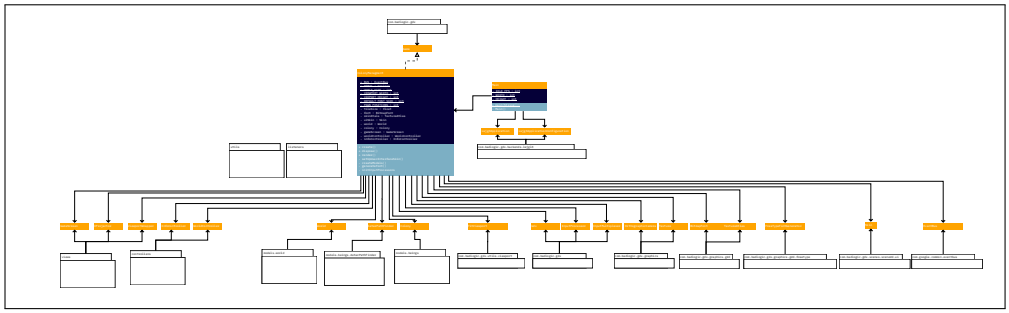


Figure A.1: Top-level UML class diagram of the application.

A.1. Models package

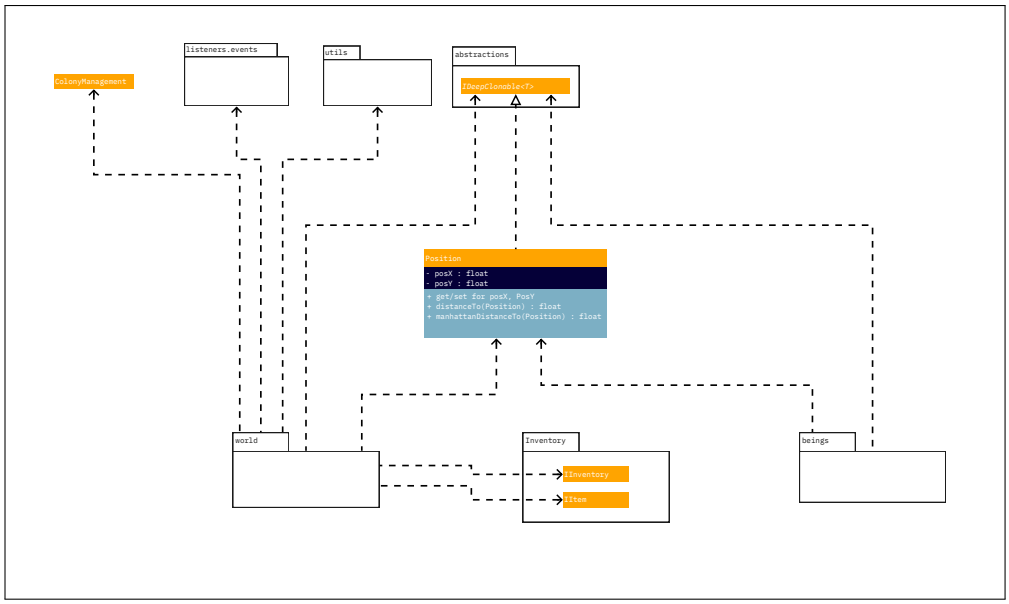


Figure A.2: UML class diagram of the `models` package.

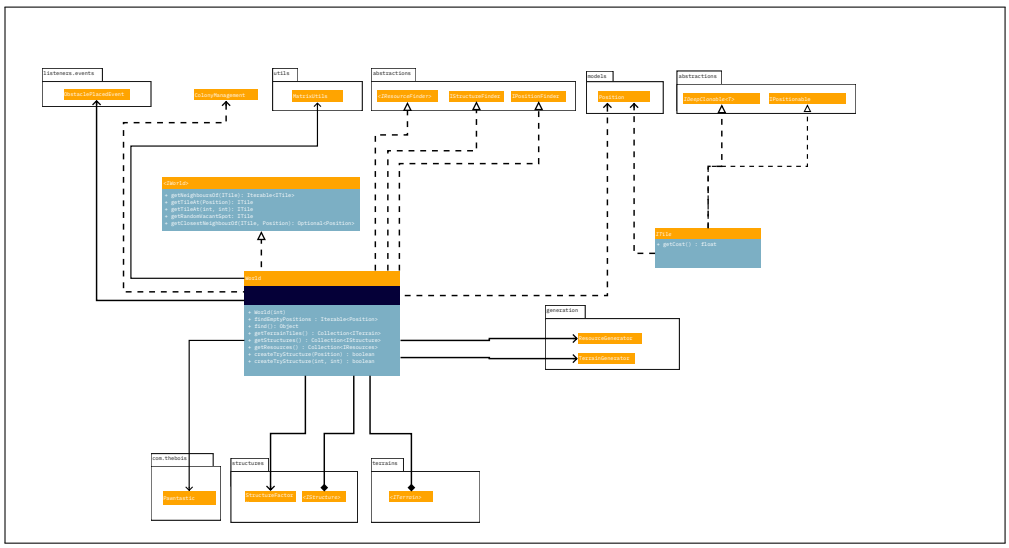


Figure A.3: UML class diagram of the `world` package.

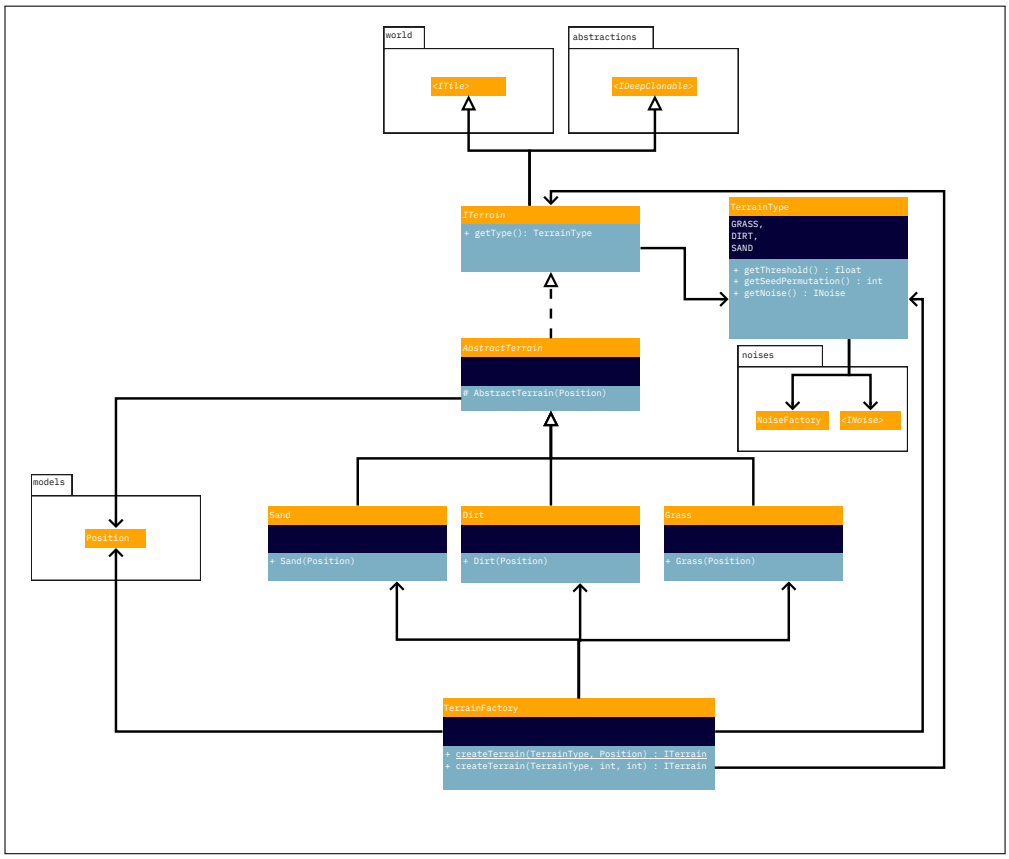


Figure A.4: UML class diagram of the `terrains` package.

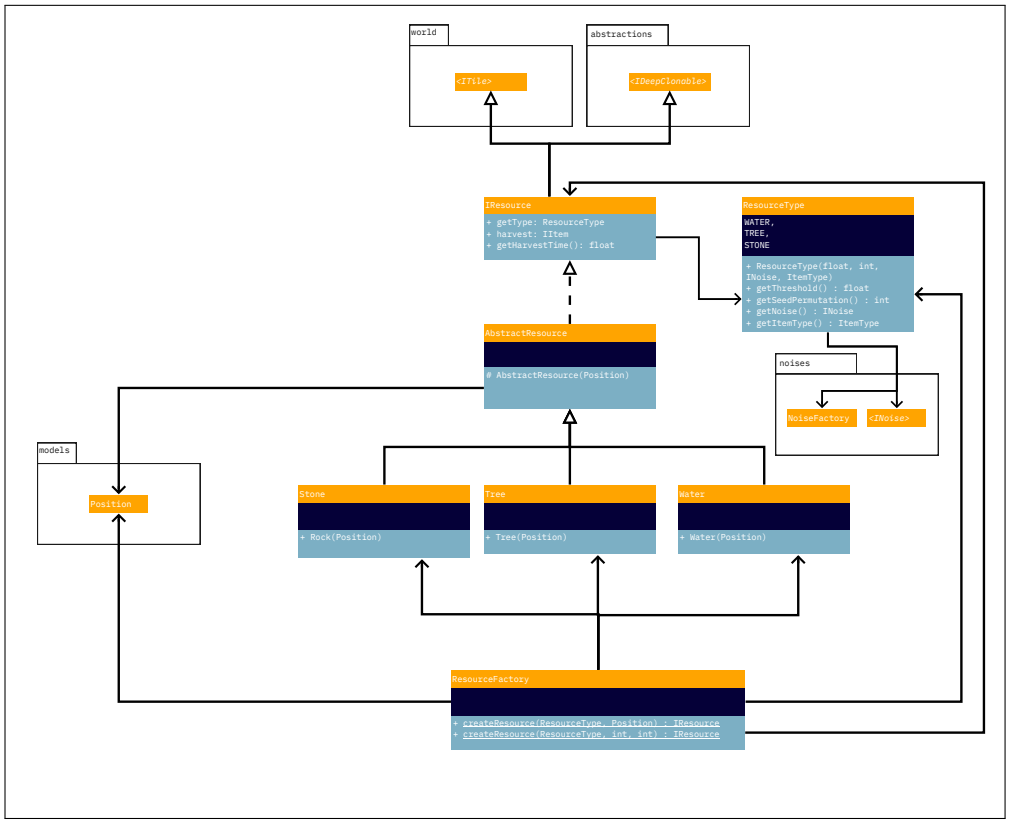


Figure A.5: UML class diagram of the `resources` package.

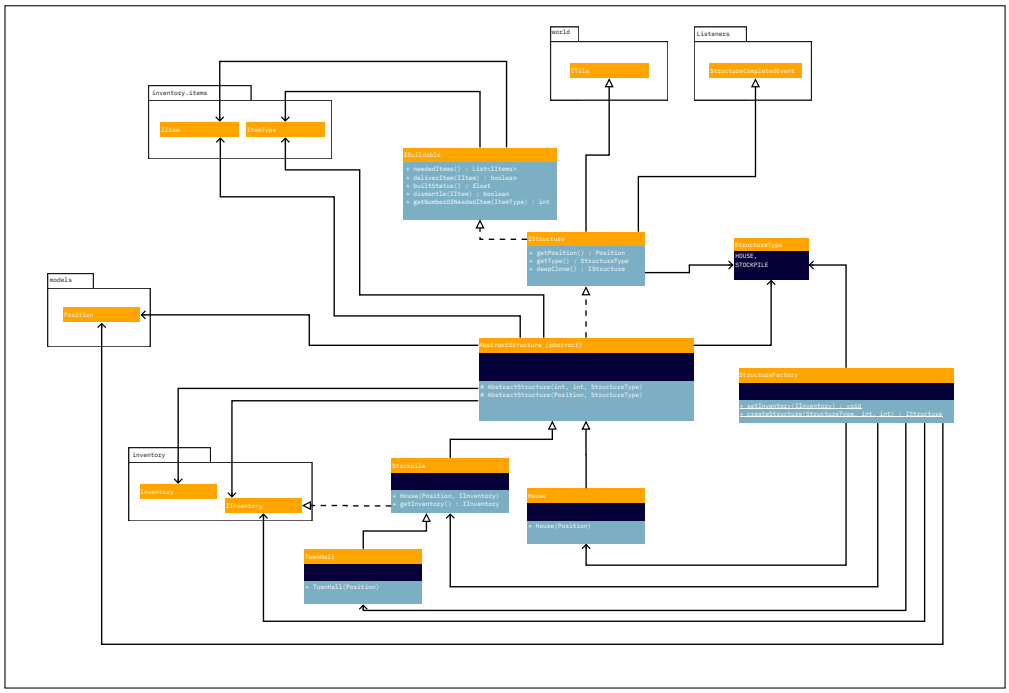


Figure A.6: UML class diagram of the `structures` package.

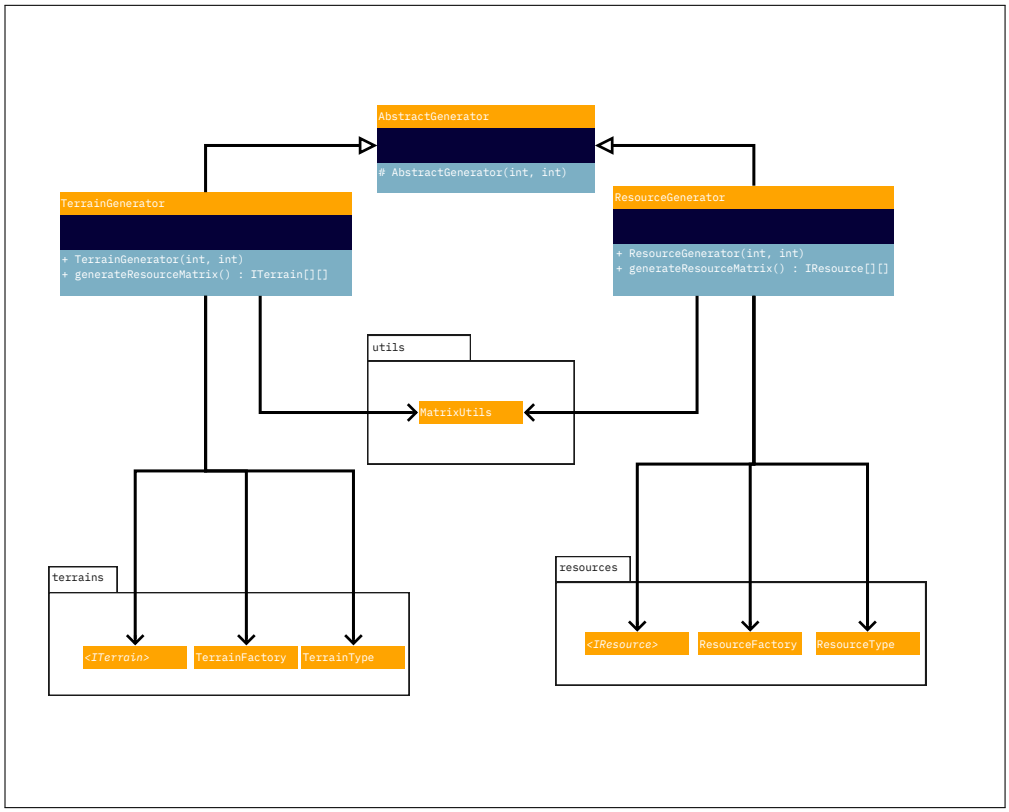


Figure A.7: UML class diagram of the `generation` package.

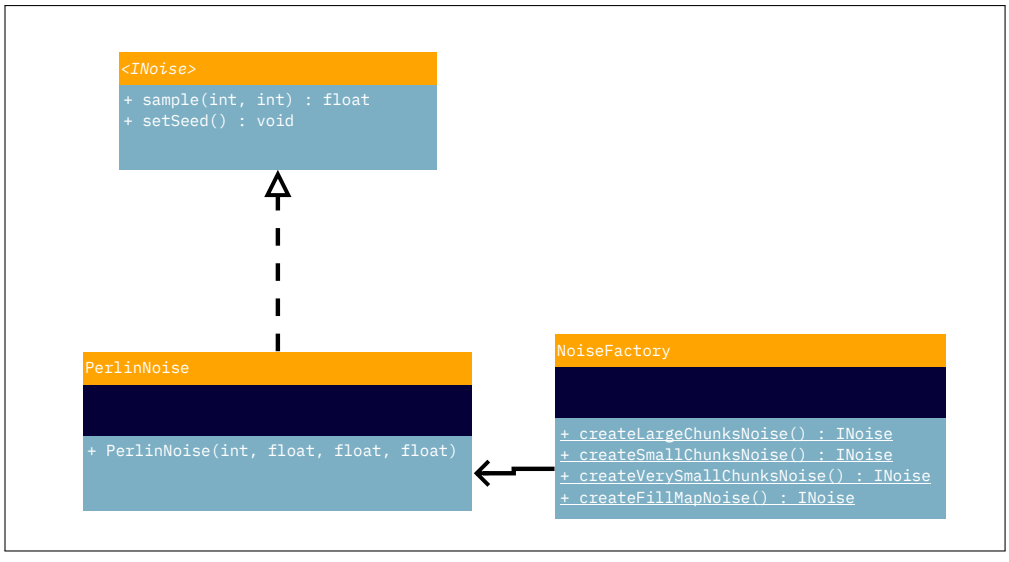


Figure A.8: UML class diagram of the `generation.noises` package.

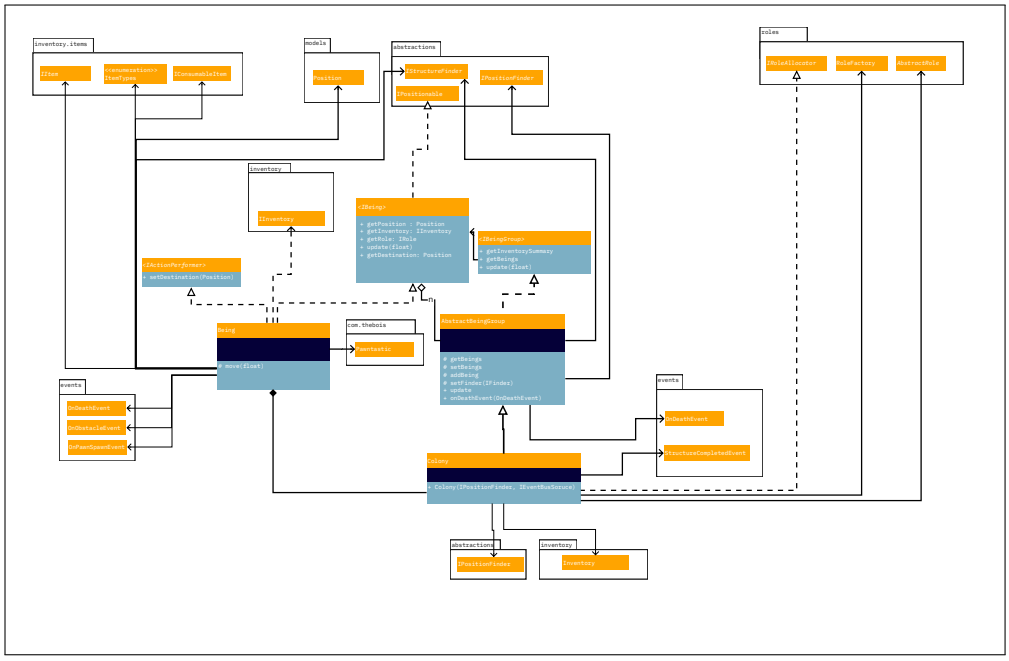


Figure A.9: UML class diagram of the `beings` package.

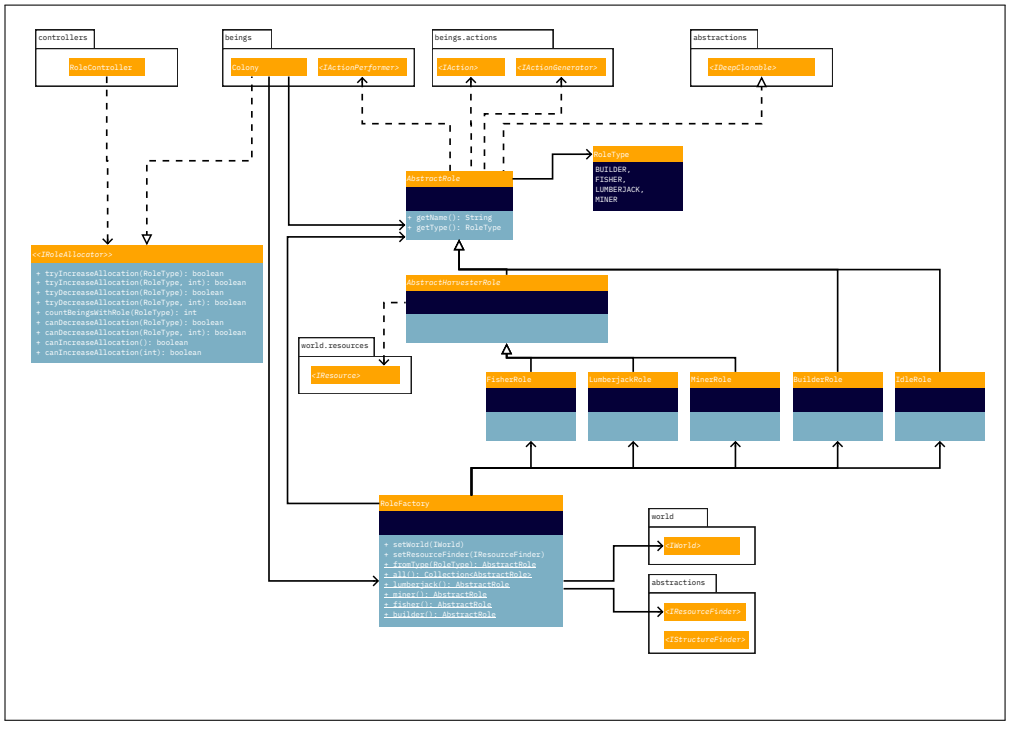


Figure A.10: UML class diagram of the `beings.roles` package.

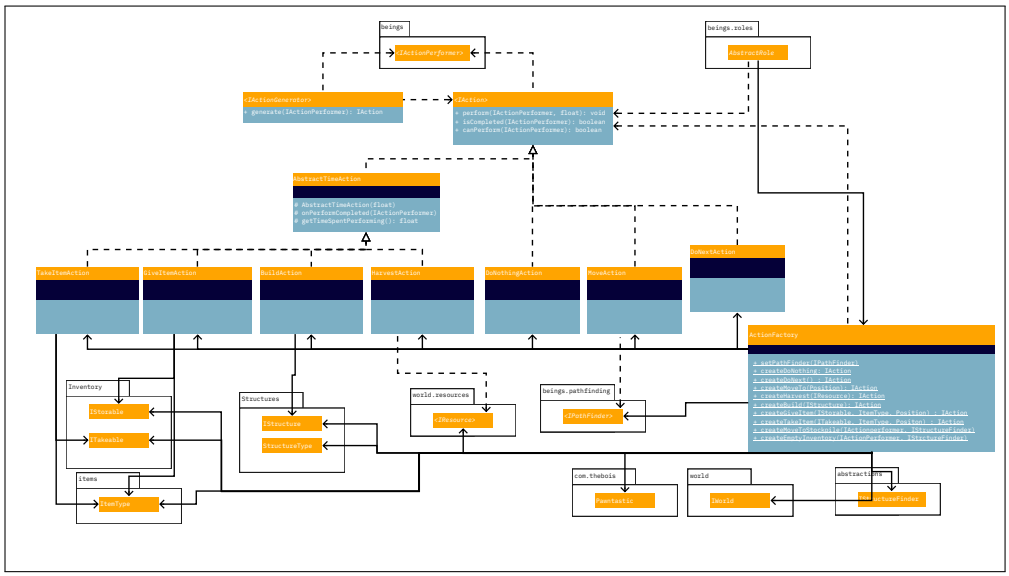


Figure A.11: UML class diagram of the `beings.actions` package.

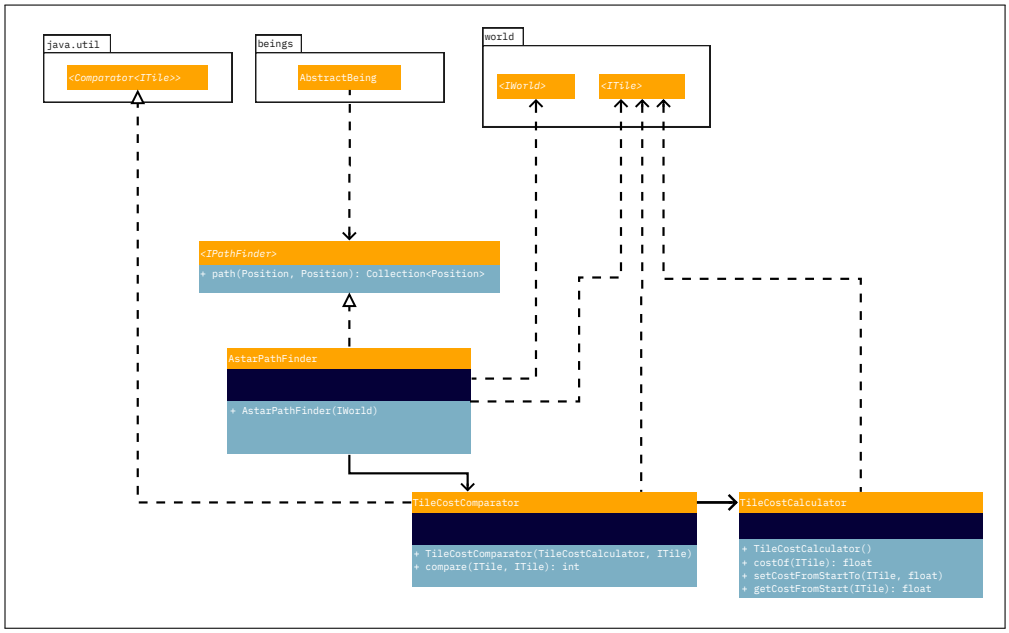


Figure A.12: UML class diagram of the `beings.pathfinding` package.

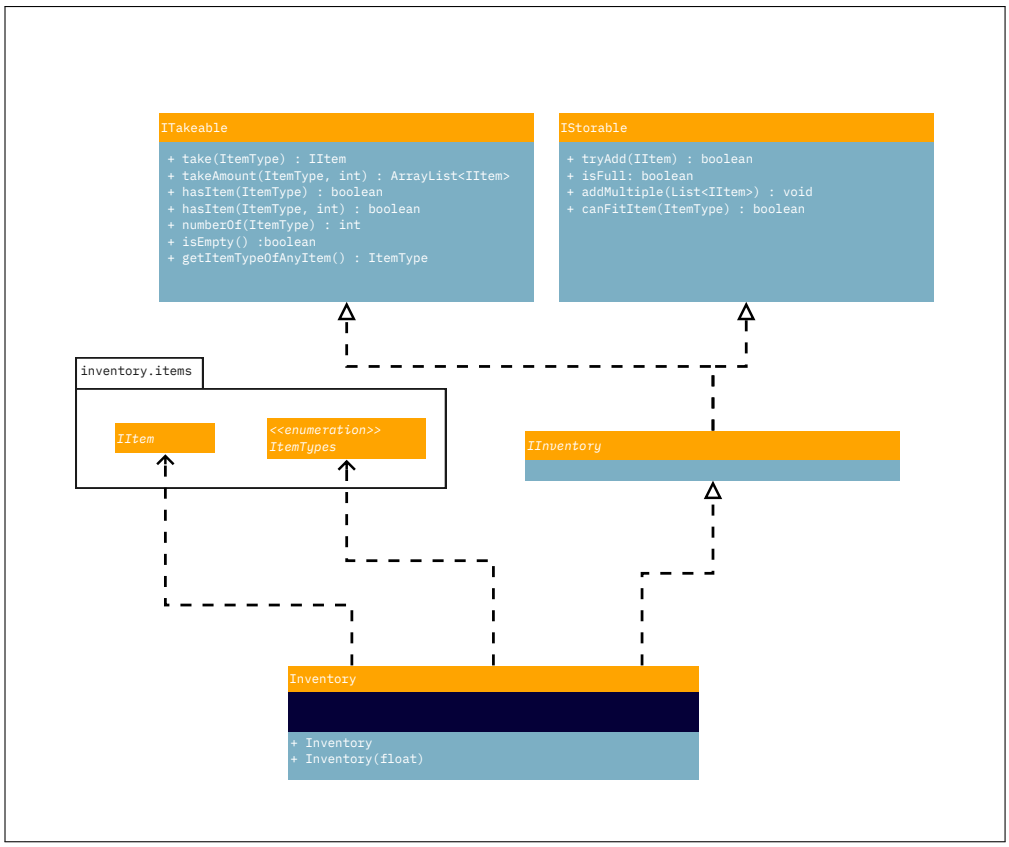


Figure A.13: UML class diagram of the `inventory` package.

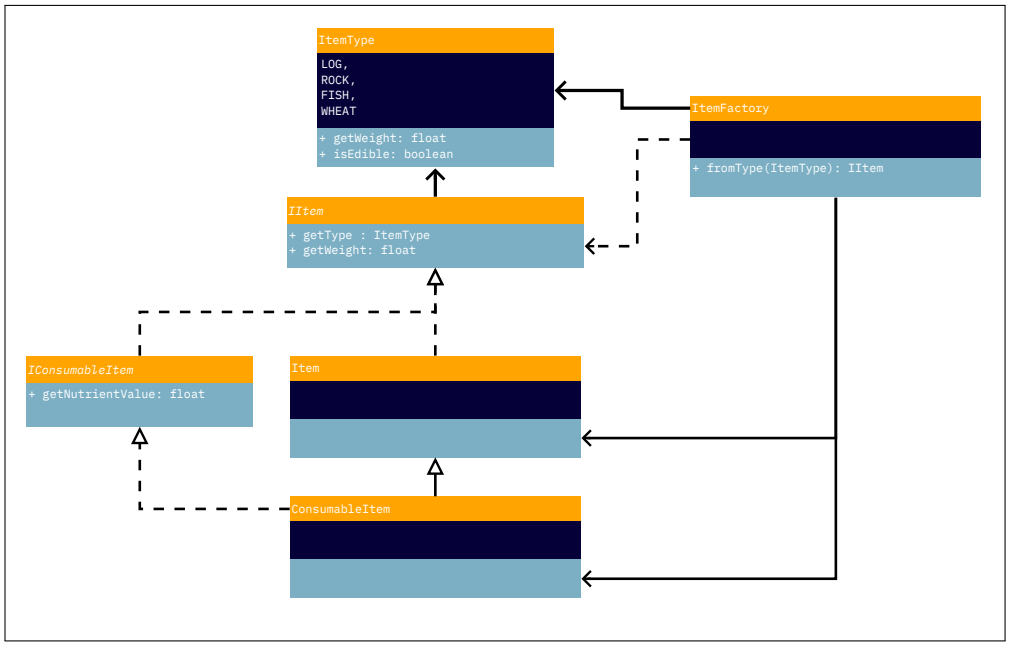


Figure A.14: UML class diagram of the `inventory.items` package.

A.2. Views package

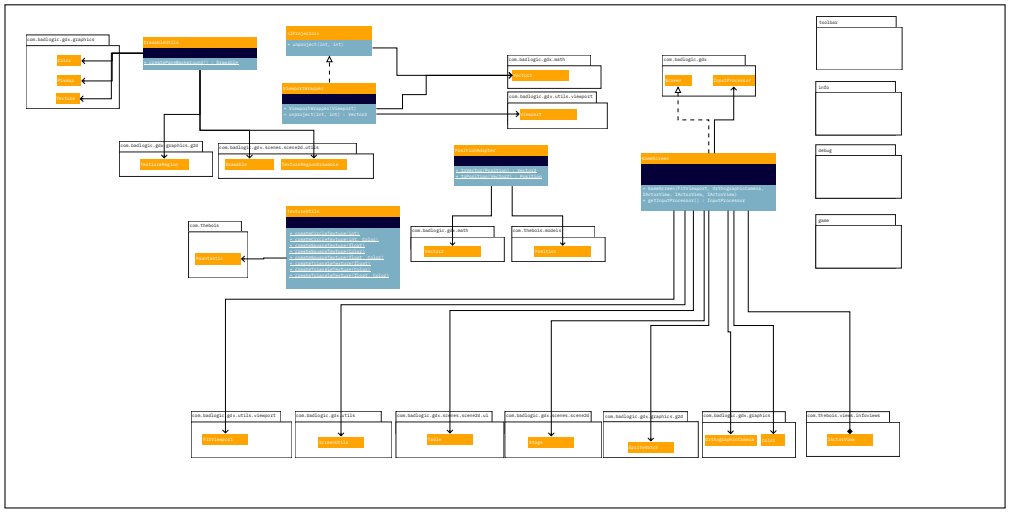


Figure A.15: UML class diagram of the `views` package.

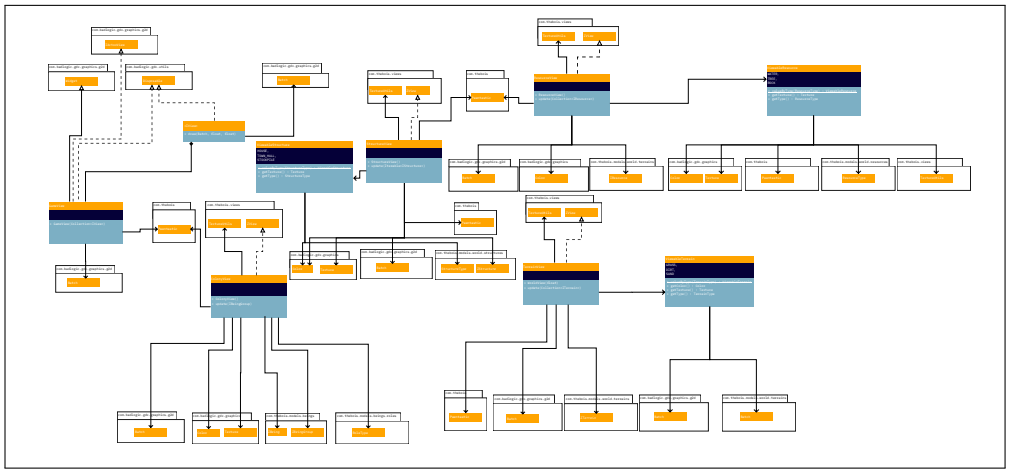


Figure A.16: UML class diagram of the `views.game` package.

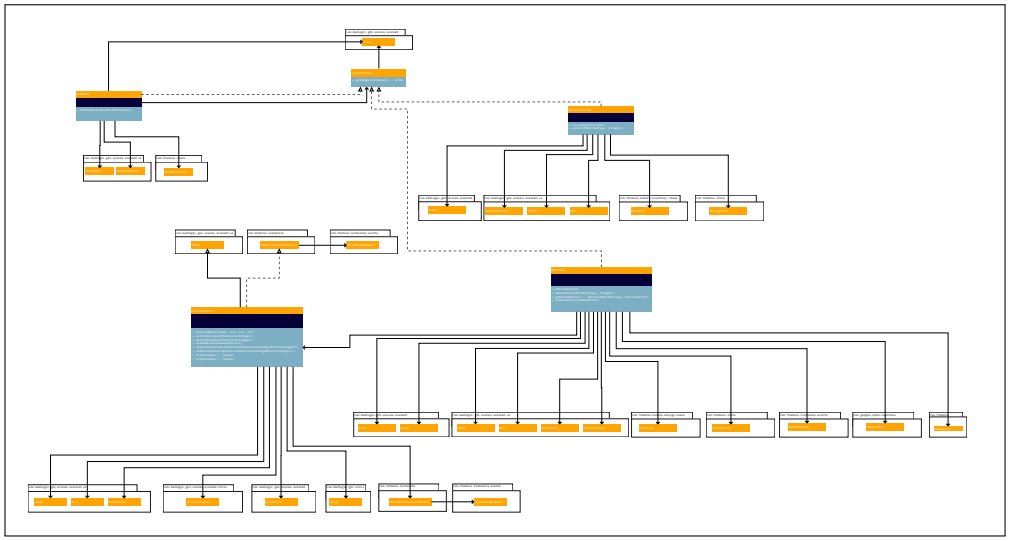


Figure A.17: UML class diagram of the `views.info` package.

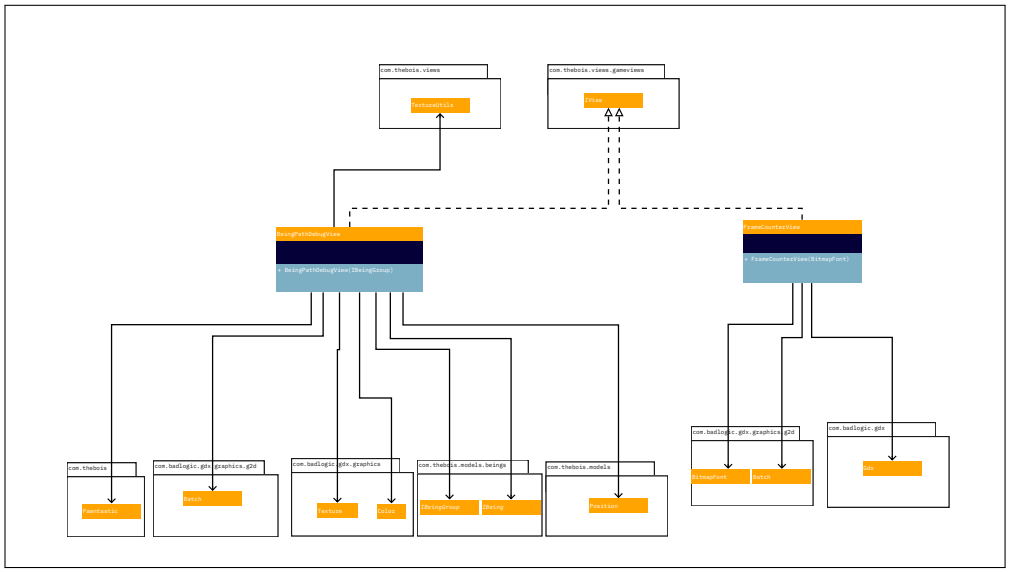


Figure A.18: UML class diagram of the `views.debug` package.

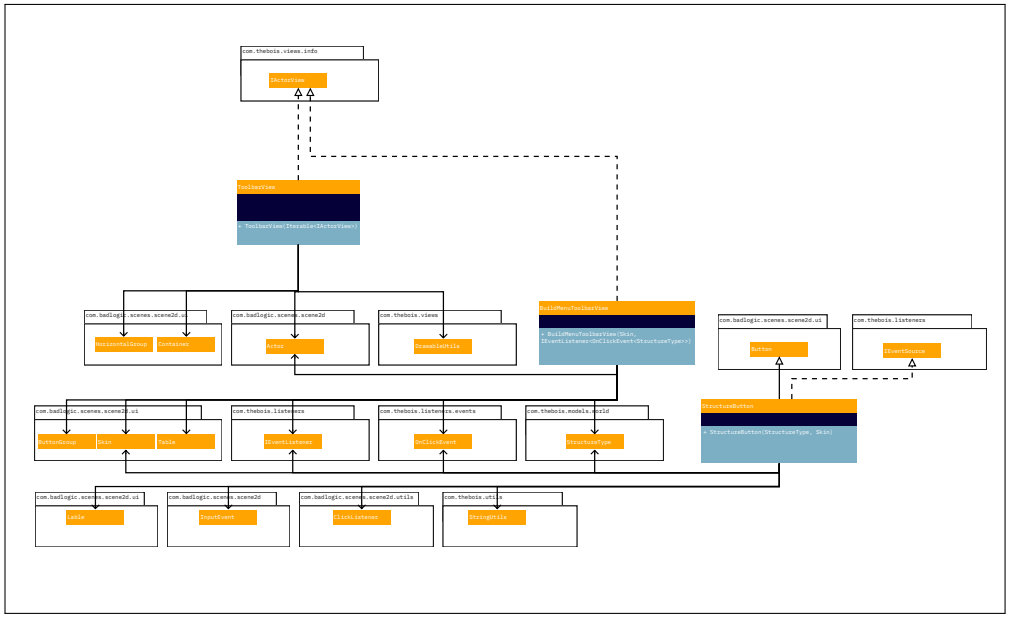


Figure A.19: UML class diagram of the `views.toolbar` package.

A.3. Controllers package

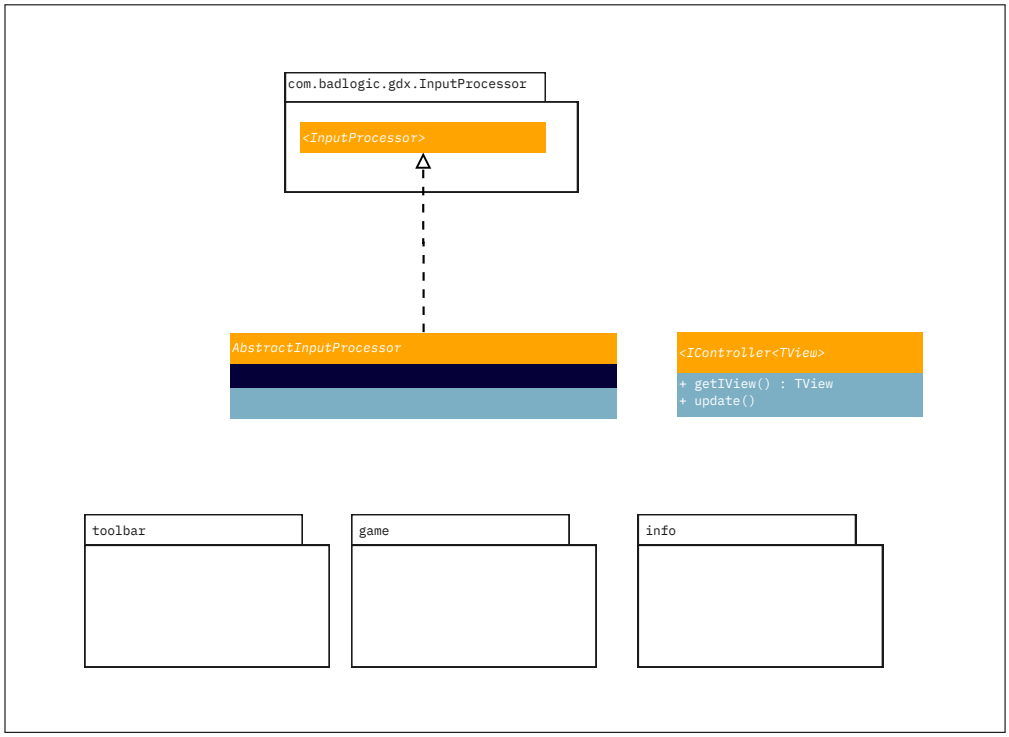


Figure A.20: UML class diagram of the `controllers` package.

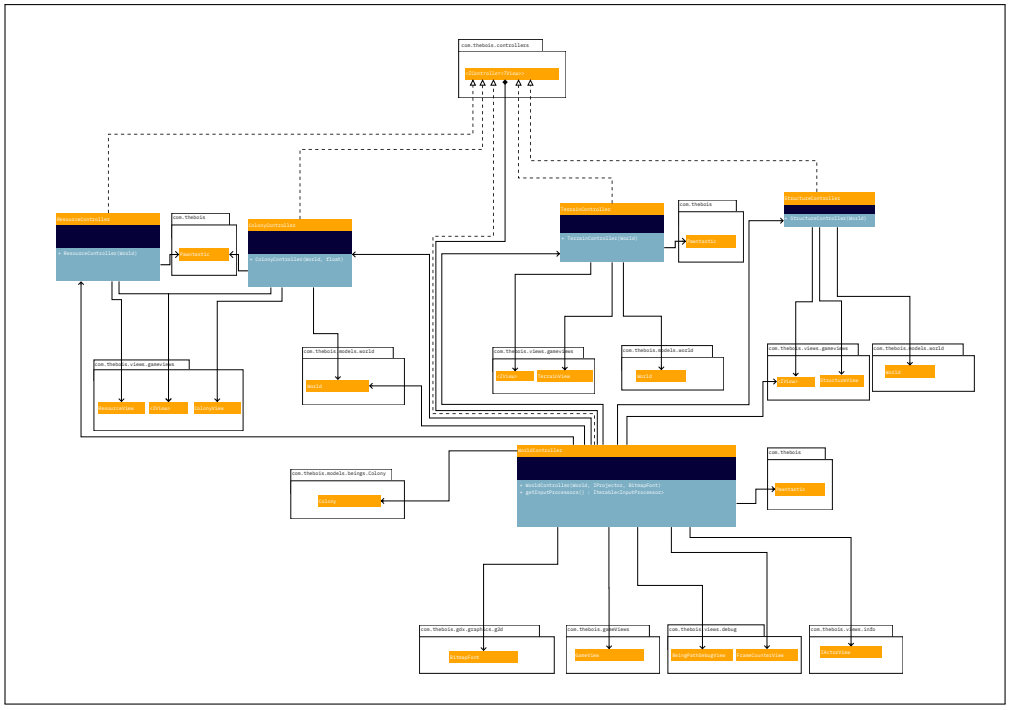


Figure A.21: UML class diagram of the `controllers.game` package.

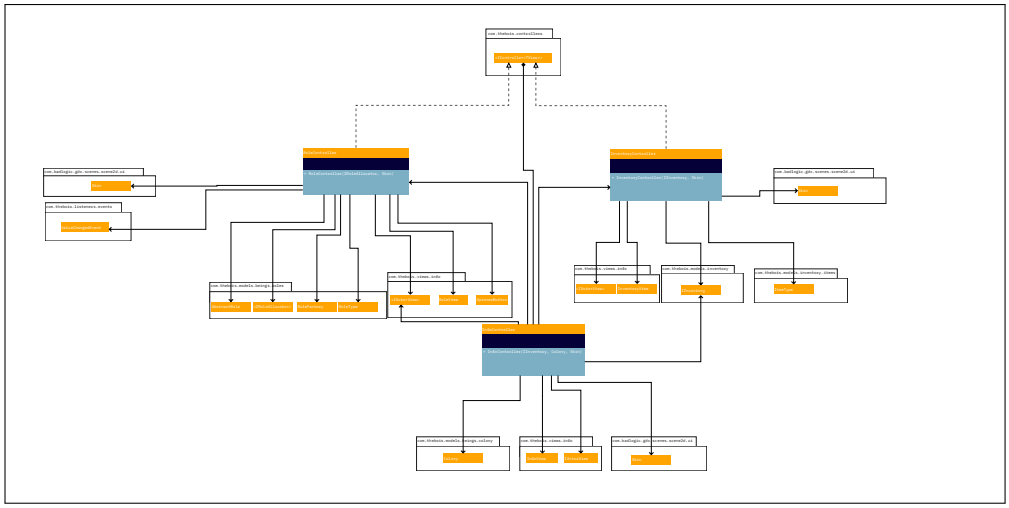


Figure A.22: UML class diagram of the `controllers.info` package.

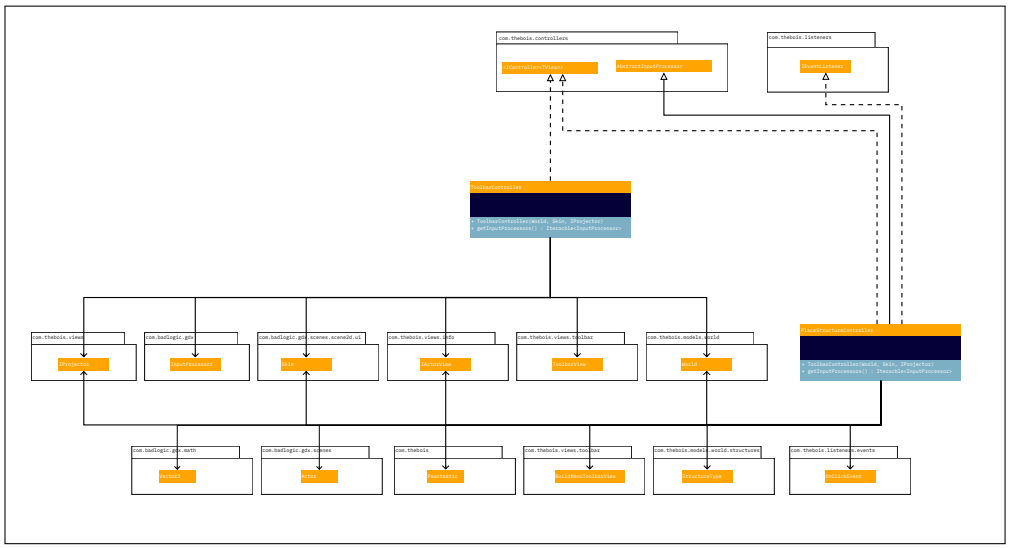


Figure A.23: UML class diagram of the `controllers.toolbar` package.

A.4. Persistence package

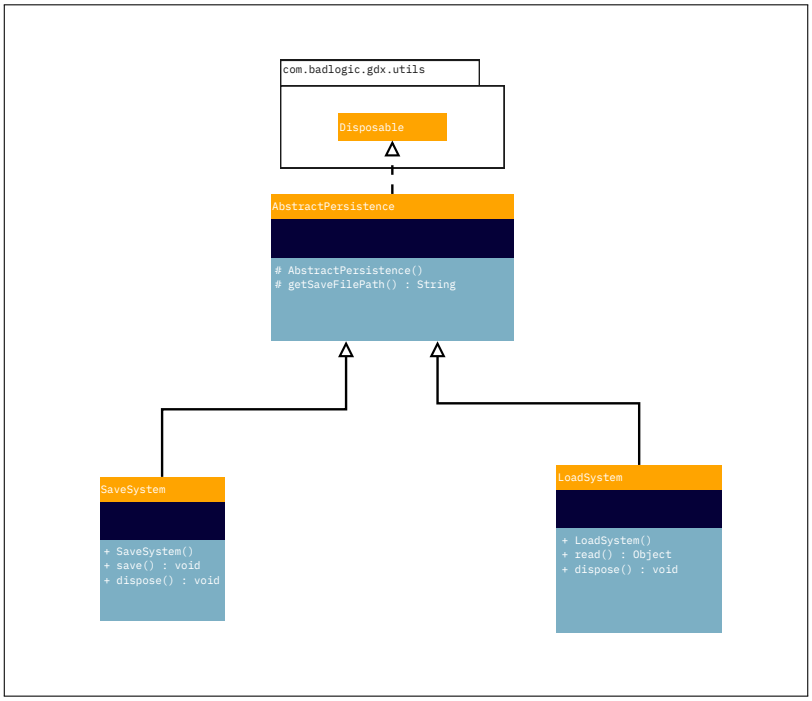


Figure A.24: UML class diagram of the `persistence` package.

A.5. Abstractions package

Figure A.25: UML class diagram of the `abstractions` package.

A.6. Listeners package

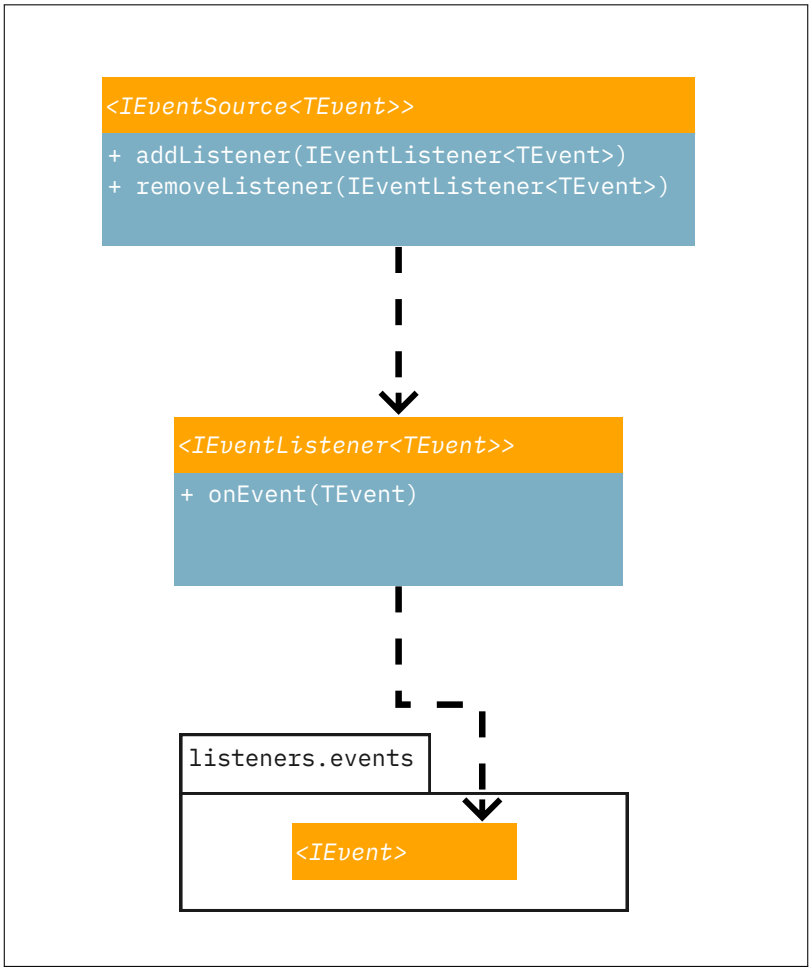


Figure A.26: UML class diagram of the `listeners` package.

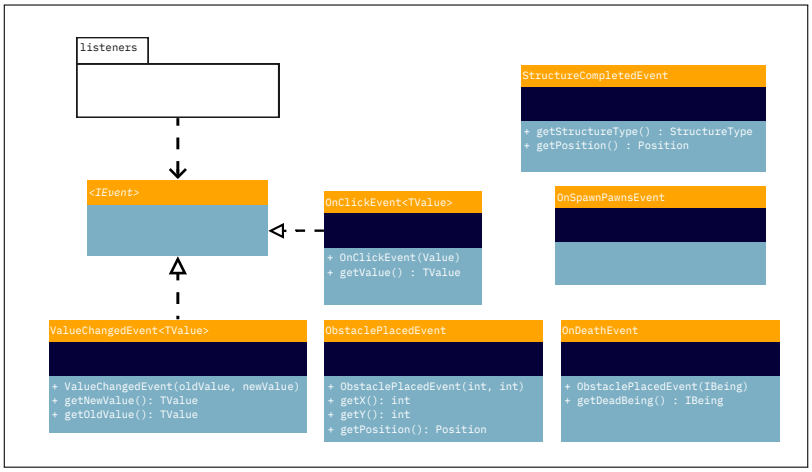


Figure A.27: UML class diagram of the `listeners.events` package.

A.7. Utils package

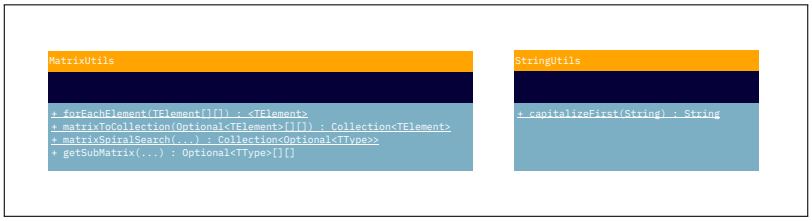


Figure A.28: UML class diagram of the `utils` package.