1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Bachelor's program in "IT: Science and Technology"
University of Urbino

Project report relative for the exam of:
Object Oriented Modeling and Programming

Summer Session — 2022/2023

STUDENT:

Nicolas Barzotti

Student ID Number 313687

PROFESSOR:

Eng. Sara Montagna

Contents

1 Problem Analysis

The goal of the project is to create a game based around a maze that you can traverse. Typical mazes in games or movies are characterized by winding paths and dangers lurking about. In this case, the only real danger will be under form of enemies that try to chase you. The other type of danger is only found by people who haven't studied properly, as the player will be asked questions that will determine loss or triumph. The questions will be based on some of the classes offered by the "IT : Science and Technology" bachelor's program of the University of Urbino.

1.1 Requirements

1.1.1 Functional Requirements

The player must:

- be able to move around the maze at will, following the maze's rules;
- be able to increase his score by answering questions or defeating enemies;
- be able to win or lose the game based on his movement.

The game must:

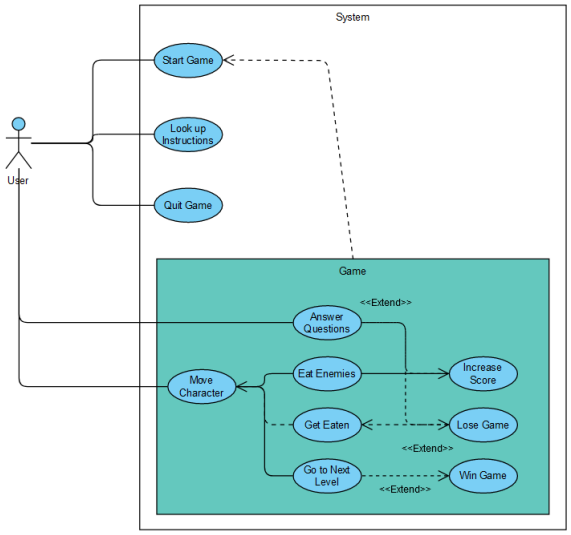
- give instructions to the player;
- allow to reset as many times as necessary without restarting;
- have enemies that chase the player down;
- have a way of asking the player questions about the aforementioned degree course.

1.1.2 Non-Functional Requirements

The game must:

- have some way to add and edit maps on top of the existing ones;
- follow as many good coding practices as possible;
- the code must have the least amount of comments possible, relying on good variable naming practices instead;
- be easily changed and extended by other developers.

1.1.3 Use-Case Diagram



The Use Case Diagram above shows the general interaction that any player will have with the game, excluding the reset which causes no difference in the diagram as it only goes back to the starting point allowing the player to choose again from the same Use Case Diagram.

1.2 Domain Model

As shown in the following UML scheme, the main purpose of the model is to maintain the status of the game, storing the information about each entity and every tile of the map. The game must allow the player to move his character, which will obviously be an ActiveEntity, as it is the interface required to let an Entity move. The reason for this is because not all entities have all capabilities; instead of creating one god-interface that would allow every type of interaction, several smaller interfaces were chosen to differentiate what each entity can do. For example, you may have an entity that can move but is non-interactable while other entities may do the exact opposite. The grid we move in is based on a matrix, the easiest data structure to work with. It is composed of Tiles that each have a position consisting of a simple interface with a pair of integers. And that's that for the state itself, with the only thing missing being the player's score and level, both held by a pair of counters. Needless to say that in future versions of the game the level counter should be removed if the maps stop being sequential, while the score counter would serve better as a bidirectional counter.

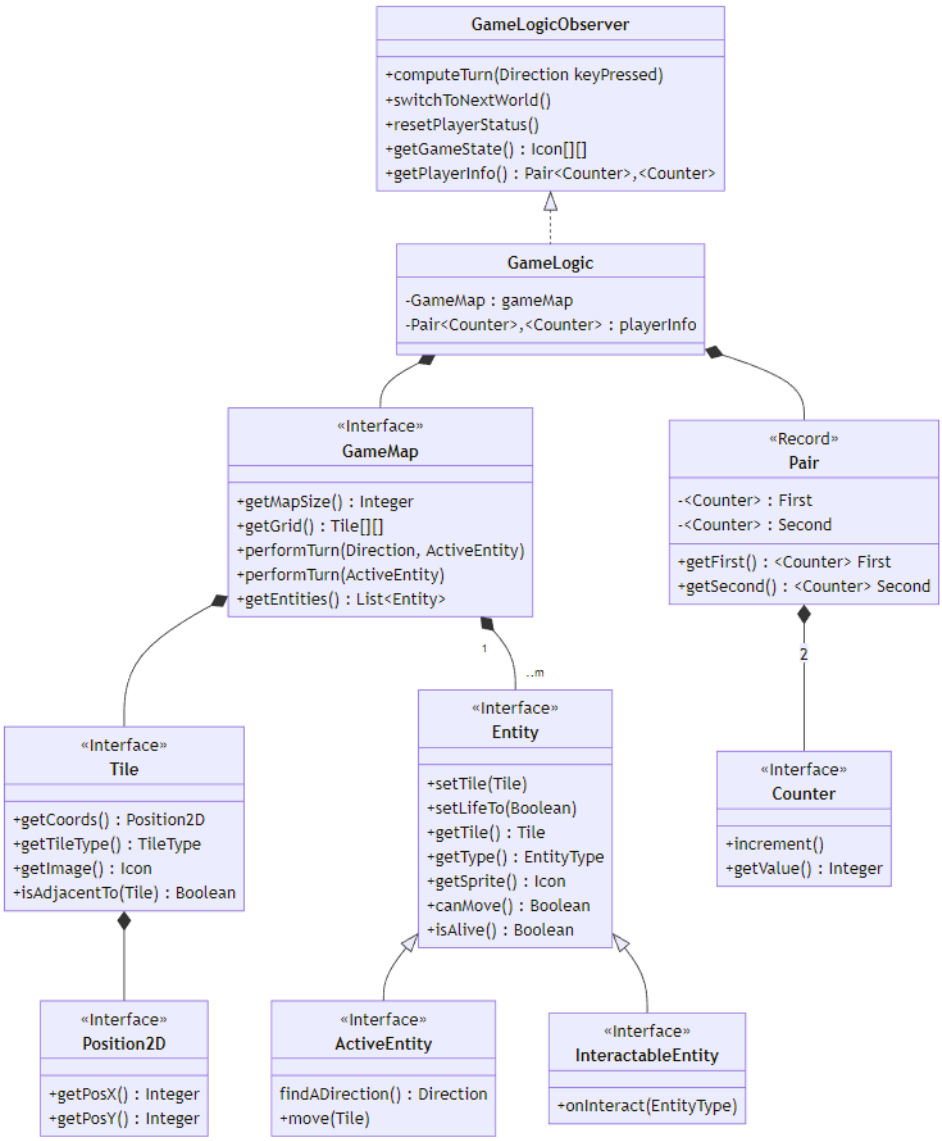


Figure 1: Domain Model UML Scheme

2 Design

2.1 Architecture

Following standard coding practices, the MVVM pattern was decided. This particular architectural pattern lets the ViewModel have only the implementation of the various observers, without any type of state. This pattern lets the ViewModel convey messages (and instructions) from the Model to the View, and vice versa. To keep the pattern strict it has been decided to have the model be able to send itself messages, but through the ViewModel. Saying this, it shouldn't be surprising to find classes without any public methods, as the only ones that should be used are the ones that they implement! This enforces the rules of the pattern, where you can ONLY access a model or view's methods through the ViewModel. A small note is to be added here: only after the creation of the required class the ViewModel will be assigned the observer's methods, thus its getters MUST return an optional value as there's no guarantee that either the views or the model will have been created at runtime. Following are two ViewModels: the first shows the architecture of the menu screen, while the second is specific to the game.

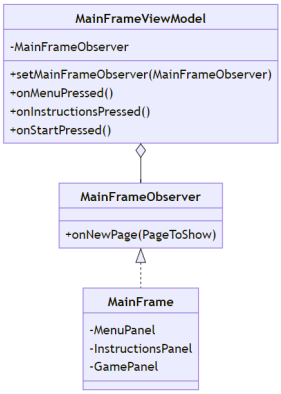


Figure 2: Menu ViewModel

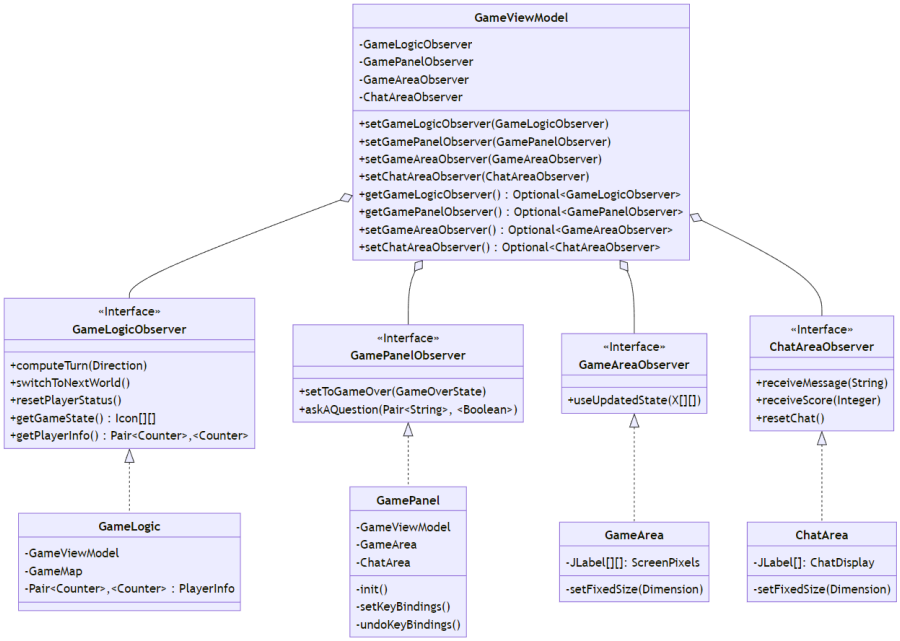


Figure 3: Game ViewModel

As can be deduced from the diagram, the MenuPanel will have some buttons that, when pressed, will let the ViewModel know which page to show to the player. There is no QuitPanel as it straight up quits the application. The MainFrame's panels aren't described properly since the InstructionsPanel has been implemented through HTML text and the GamePanel is present in the next scheme. With the UML now at hand, we can see the ViewModel's entire host of methods, with each observer carefully selected to have its own concern. Looking at ways to improve this architecture different model observers could be added, not unlike what was done with the view. Better yet, the same could be done with the ViewModels.[1]
Following what was explained, we can determine that:

- MainFrameViewModel has no model attached to it;
- Direction can only be up, down, left or right;
- GameOverState can only be win or loss.

2.1.1 GameLogic

As shown before we know exactly how the GameLogic's inner workings elaborate data. This is the model's main class as it maintains the game's status. It also implements the observer that is then used in the Game's ViewModel to perform any type of action in the GameLogic itself. The methods implemented in the observer are crucial as the logic itself has no methods of its own.[2]

2.1.2 View

If the model was simple enough to put it into a single Interface, the same can't be said about the view. The graphics of the game are offered by Java's native Swing framework, and as such there have been different panels created ad-hoc. Following this it is no surprise that multiple observers have been implemented, one for each panel that required an important use of the ViewModel. [3]

[1] In a later version the ChatViewModel was added in order to separate concerns, and it is a static field in the GameViewModel that only the ChatAreaObserver communicates with.

[2] One issue that permeated throughout the whole architecture was the choice of having Icon classes be the way the screen was saved. The reason for this is because there's no easy way to manipulate icons, so for example I had to resort to making an extra method in the version where Integers would be used instead of Icons in order to show the map to the player in the terminal mode. Do note that this is due to the specific nature of the Icon class, as it could be used in the terminal view by changing their "To String" method to make them readable.

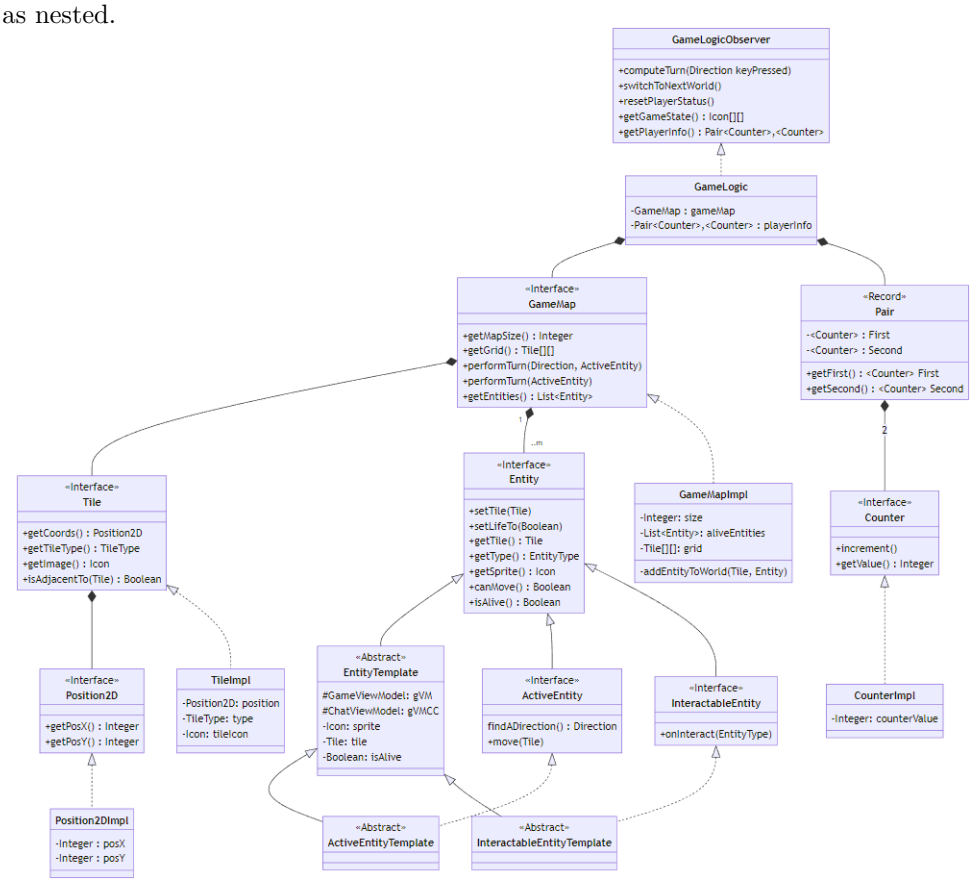
[3] After-implementation note: Look at the Instruction panel for example, it doesn't have it's own observer, instead it reuses the one implemented by the MainFrameObserver

2.2 Detailed Design

In lack of other group members, I will follow up by listing each important component, talking about the relevant details.

2.2.1 GameLogic

The model that was implemented in the system is already well covered in the design section, so the main challenges of the GameMap mentioned in the Domain Model will be the focus of the discussion instead. As the GameMapImpl was created the idea always had been that it would be the one in charge of moving entities, as the entities themselves don't actually know the GameMap. Modeling this proved to be especially difficult. For example, the first implementation was that each Entity needed a Tile, and that each Tile could have an Entity; the reason for this was to make the movement easier as you could access the tile's entity directly. In retrospect, this could be an implementation that could very well be brought back in a future update of the game; by not making the Entity and Tile classes be intertwined you run the risk of making the code very unoptimized, for example: the movement logic requires to know the position of the Entity's starting tile (which is easy, because the entity has a tile), but it also needs the destination tile, for which the code requires to check if any other entity has that particular tile. As an implementation note it must be said that by using streams you could benefit from multithreading, though that is not an approach that is necessary to document to the ends of this report. Another important piece of information about GameMapImpl is that the creation of entities was first designed with a static context in mind. This means that the entities are set when the map is created, resulting in maps that don't have an easy way to add entities after they are defined. The movement system was resolved by enabling each entity to define its own movement logic. Subsequently, the gameMap would request the entity to indicate the direction in which it intended to move. This is not the same for the Character as it clearly requires a direction to move in. Having passive methods (onInteract, onMovement, and so on) allowed for really clean movement logic, also emphasized by returning early in specific conditions. Not only does that imply less instructions per movement, but it also allows the code to not be as nested.



Following the previous diagram:

- EntityType is an enum;
- TileType is an enum;
- In the above scheme, the concrete classes like Character, Enemy, NPC, SmartEnemy and Phantom are all unnecessary, the only thing that they change is how they implement the interactions with the world and with the other entities. What matters most are the abstract classes that act as a template;
- Factory pattern used in the creation of each entity;
- Strategy pattern used to order the list of entities, the chosen strategy was by ordinal, since the entity type is an enum.

In the future, this could all be changed by having an actual visual tool to easily make maps, and to specify which type of entity to have, something that was partially done in the project as can be understood by looking at the "EntityType" enum. A pretty good implementation would let you make the base map first, and then let you add entities as you wished. For example the current version requires you to specify if a tile has the character, an enemy, or an NPC. A specific txt file that held a number for the entity type and a pair of numbers corresponding to their position could be added instead. Another change that could be added would be to have a .txt file in which you specify what dialogue NPCs have, instead of picking them up randomly from a list.

2.2.2 View

The main objective of the view wasn't one of design, as all of the components were created and added manually. However it was enough for the view to let the player know what was going on, and this is where the different observers were created: each one holds a different purpose and has its niche in the project. The problem with this is that by using Swing, and therefore a lot of different panels, a lot of different observers came into play. This becomes problematic when a second version of the view is added, as it needs to fulfill all the requirements of the ViewModel. To this effect, a new version of the view was added in order to check its ability to adapt; the alternative view offers a rudimental, albeit working, version that proves that the MVVM pattern was properly followed. Both views have their own way of displaying the screen, thus it is mandatory to know beforehand what type of data they can expect and what they need to do with it. In this project, each view will avoid changing status in case they are given an incorrect data type, though it'd be easy to change this. To give a concrete example let's look at the terminal view: it requires integers to print out the current state in the terminal; however you may also use Icons, as is expected from the model, in order to work with those instead. Since the terminal view is simply a concept it was decided to not bother with changing the data type and transforming it into something usable, instead having a different method that would give it the status through integers was prefered. [4] With that said, the modeling of the view itself wasn't all too hard, it only required the in-depth study of the Swing framework and to learn the basics of HTML as some components only work properly with that.

2.2.3 ViewModel

Its role is relatively simple: to convey messages. And yet this is crucial for the functioning of the system; it is not surprising that two, technically three, were made in order to separate what each one does, with the most important one being the GameViewModel.
As mentioned earlier, this software lacks a distinct state of its own, relying instead on the methods of the classes that are associated with it. Considering that only one game may be played at once, the Singleton design pattern was used in order to avoid having to pass the ViewModel itself through the whole system, class-by-class, which is tedious and unnecessary. This also makes code cleaner, and whenever needed, classes could save the ViewModel's instance in a variable in order to make code less redundant and have better performance. Following this same idea, the ChatViewModel was created inside of the GameViewModel, also following the singleton design pattern: since it is entirely reliant on the game's existence to transmit messages, it was logical to create a nested class exclusively for this purpose and implement it as a singleton.

[4] Needless to say that if someone were to actually implement it proper, it would be mandatory for them to make the most of Java's capabilities!

As for the GameViewModel itself, the most interesting challenge proved to be actually finishing the game. By having a cushion panel, GamePanel, be the one handling the user inputs and the questions, it also seemed fitting for the screen-blocking functionality to be implemented within this class, particularly to hide the chat when the game concludes. After all, it would be odd for a game to leave its chat open once the game is over. A warning is required here, a very big one at that, since the GameViewModel (and ChatViewModel) has a host of in-built methods: these are the ones that were used in this project to develop the game, and they are a combination of more than one observer. As such, they aren't necessarily the most versatile, for example some entities have a chance at interacting with the player by directly using the chat, which of course requires an actual chat panel (or whatever else was implemented). A simple fix for this is to use the getters for the observers in the ViewModels, letting each component of the system use the methods as they see fit. The problem with this was the redundancy, thus the decision to make standard methods once and for all. Even more important than this though are the interactions in case the ViewModels aren't provided with all the necessary information, such as by not creating a View for the game. A fix is to have the getters return optional values, so that if an Entity can't find a chat to connect itself to, it will simply not talk to the player.

3 Development

3.1 Automatic Testing

Following good coding practices, it has been decided to have some way of having automated tests: this is where JUnit 4 comes in. The choice came naturally as it was previously studied in both the "Object Oriented Modeling and Programming" and "Engineering and Software Architecture" classes. Since the first few actually working classes were implemented, the tests would soon follow using practices from both classes. The only real problem came from trying to understand Java's reflections used to properly implement the tests themselves. Since only the core features were to be tested, the GUI was excluded from testing as it lacks any significant underlying logic. It must be said, however, that any time the game is played the GUI is being automatically tested, and it has not given any unexpected errors, which is not surprising. The tests all follow the same package architecture as the software itself. This was done in order to better represent what the tests actually do: at least one class worth of tests was created and defined for each class among the most important in the model, then leading to combination tests which would make larger use of all the functionalities available.

3.1.1 Basic classes

- Character: it was enough for this class to be created and moved around, making sure that its interactions with the world were the expected ones;
- Enemy: similar to character, the class was tested for movement and interactions;
- NPC: since the NPC has a different function, it wasn't moved. Instead it was interacted with to make sure that it wouldn't cause any type of error when both model and view were missing;
- EntityFactory: simply put, these were all banal tests, as it only had to make sure that the outputs were of the correct classes;
- GameLogic: the GameLogic test was made with the ViewModel in mind: it was also made to throw an exception when the amount of usable maps was too great;
- GameMapImpl: this has some tests about the movement of enemies inside of itself;
- Position2DImpl: simple tests for what is essentially a pojo class;
- Tile: despite being a pojo class at heart, its utility method was intensely tested as it holds the logic movement together;
- TileType: holds the test for each type of Tile;
- CounterImpl: having a couple of constructors, both were tested for their functionality
- Direction: as an Enum with a few methods, all possibilities were tested.

3.1.2 Combination tests

Both of these tests are present under the model package. They cannot be attributed to any specific package, as they encompass multiple aspects and functionalities. Both of the tests are held in the same class, and are as shown under.

- TestMovement: as the name implies, this tests that the movement works properly, as implemented in each entity. Worthy of notice is that the move method (inside the entity) only asks for the destination tile, as seen from the APIs, and no checks are made except for adjacency.
- TestInteractions: this one may need some explaining. When the interactions were implemented, it was made so that only after answering a question, a player would get rid of an NPC. This makes it so that without a view that lets you answer said questions, you won't be able to move through them. As simulated in the test, there is no view, which means that the character must sit still.

3.2 Work Methodology

As taught in the "Engineering and Software Architecture" class, the chosen software development process was that of an incremental process. The reason was quite simple: I would set up basic objectives to work towards and then the classic waterfall model was followed through until completion. To help with this, Git and GitHub were used for their versioning control, letting development branches be opened each day and closed when the task at hand was completed, which of course included several fully commented commits. The objective of these commits is to represent each small step taken towards the goal, which is why there are so many commits with small, yet important changes. Any type of VCS is mandatory in any self-respecting, large-scale project! Aside from personal use, if a group were to use GitHub it would easily let different developers branch off into what they're developing. This means that they could all work in parallel and, if the interfaces are respected, it also means that no conflicts in merging would occur. Interfaces are an extremely important part of any good project and changing or carelessly breaking an interface requires some major refactoring. This is explained through semantic versioning, where any type of software's versions are a triple of natural numbers $X.Y.Z$ where X is the major version, which changes every time an interface is changed. Y is the minor version, where interfaces are respected but updates are still implemented, and Z is the patch of the current version. It is important to note that this particular project did not implement semantic versioning. In regards to the incremental process, the objectives set and successfully accomplished were in line with those stated in the "Requirements" section. However, it should be acknowledged that certain requirements underwent modifications, becoming more specific or introducing additional criteria beyond what was initially outlined in the blended forum.

By going in order of implementation:

1. added a matrix that functions as a map;
2. a rudimentary map editor was added;[5]
3. entities and their interactions with the world were added;
4. the view was implemented as it was necessary to have an event that would force the character to move. Having the character move also meant discerning how the gameplay loop worked. Considering the checkered pattern of the map, it came naturally to have a turn based system. This came at a cost though, as it meant that entities wouldn't be able to act on their own anymore;
5. the instructions panel was added which required to learn some HTML basics;
6. added different entities, most noticeably NPCs;
7. added another type of view to showcase the ease of use of the MVVM pattern.

[5]Needless to say that if this were a full blown project you could add another software that would let you graphically edit maps and add enemies and such, however this is not this project's concern, thus a simple map editor was added.

3.3 Development Notes

Java is a vast language, whose capabilities are heavily extended whenever a new version is put out. I'll list the most important bits of code that made use of newer versions, pointing out what they do.

3.3.1 Java 2

Collections: introduced early in Java's development, collections offer an easy way to add the most common data structures with the necessary methods to work with them. This was used to keep track of entities and more.

3.3.2 Java 5

Static imports: static imports let the developer use the imported class' variables without the need to statically access them. For example, this was used heavily in the testing part of the code: each import statement for the "org.junit.Assert" class is static, meaning that the syntax was shortened a lot.

- before: Assert.assertTrue(...);
- after: assertTrue(...);

Generic types: these were used for the OperateOnMatrix class, more specifically for the operateOnEachElement method. The method itself was only used to have more refined code, to be able to work with a matrix with a single line of code when used in conjunction with Lambda expressions. As the matrix could've been of any type, it only made sense to have the method be generic as it meant that it could be used in any other point of the code. Do note that the methods used in this project could've had better performance; one example is updating only the cells that were used instead of the full screen. The best possible performance isn't completely relevant in the project so it was left mostly as is, with one trick involving modifying the update condition of both "for" loops: using the ++ operator before the variable gives a minor, but noticeable boost whenever working with large numbers of cycles. Noteworthy but not entirely interesting are type inference and bounded generics, both of which are used in the project. We can see an example of bounded generics in the playerInfo field, as it is mandatory to provide a type of class that works AT LEAST as a counter. For example in a future implementation the game could change to deduct points when you get a question wrong; this requires a different type of counter, known as bidirectional, since the one that is currently provided only counts up.

3.3.3 Java 8

Lambda expressions: lambda expressions make use of functional interfaces to shorten the syntax, as Java is known to be quite verbose as programming languages go. This was used extensively throughout the whole project, most notably in the aforementioned operateOnEachElement method.
Method references: used for cleaner use of lambda expressions, the method reference was used in conjunction with streams (see below) to quickly perform the same operation for a lot of variables. In this project it was perfect for the keybindings in the GamePanel class, which let me remove all of the bindings in a single line of code.
Streams: used when referencing data structures as it gives a quick and short way to manipulate a lot of data. For example in the case of method references or, more importantly, when creating the GameMap, streams were used in order to set every smart enemy's target to be a character they found. A stream's characteristic is to require a source, an indefinite amount of intermediate operations and a closing operation. All are noticeable in the given examples.
Do note that the InputStream used for the MapReader class has nothing to do with Java 8's streams! The former is used to read from a certain source, while the latter is strictly used for collections.
Optional types: added to reduce the amount of null pointer exceptions, they are a great addition to safely operate with fields that may go unused, set at a later stage or deleted during runtime. This was used for all the methods in the ViewModel classes to make them as modular as possible.

3.3.4 Java 16

Record types: used for the generic Pair class. They give access to a class whose attributes are set as it's constructor is called. The important part is that it's an immutable class, however if by chance the attributes are mutable classes, their contents may be used and changed. For example: the playerInfo field gives easy access to both the level and score of the player and due to the nature of the counters that make up the pair, they must increment. This was easy to implement with a record class as the counters are created once and no more, and anyone can use the class' implicit getters to use the counters.[6]
As a bonus, Java's compiler recognizes record classes and bestows them the fitting constructor, setters and getters without needing to specify them.

4 Useful Links

- Intellij IDEA |`https://www.jetbrains.com/idea/`
- Git |`https://git-scm.com`
- GitHub |`https://github.com`
- GitHub project repository |`https://github.com/Riahelm/SimpleMaze`
- JUnit |`https://junit.org/junit5/`

[6]Check how immutable data works. The attribute "final" may not let the field's pointer change, yet you can use any of the class' methods which may change it's own state as the Counter class does