FIT3077: Software Engineering: Architecture & Design

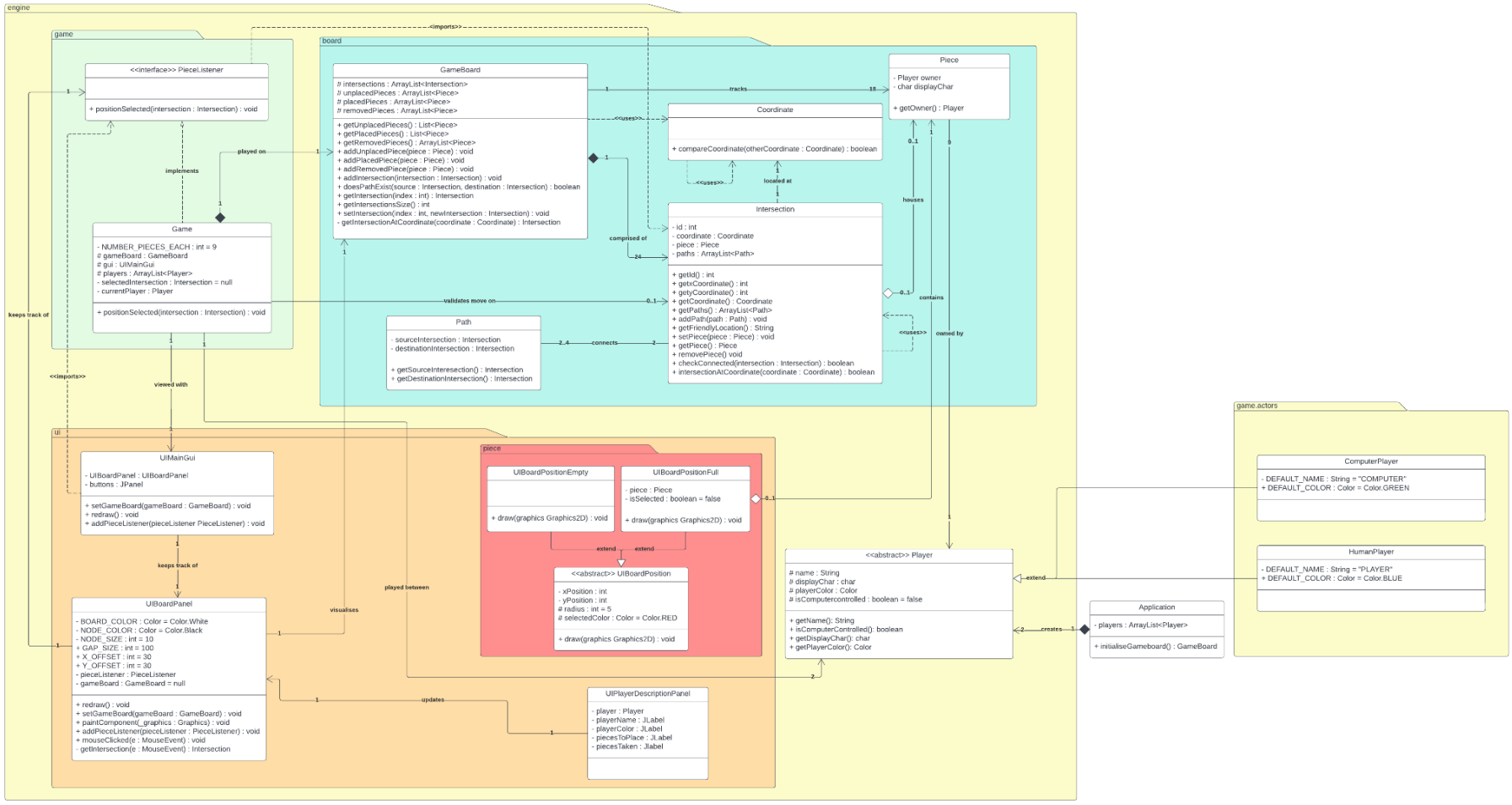
Nine Men's Morris - Sprint 2

Tech-based WIP, Architecture, and Design Rationales

Group: *CL_Monday6pm_Team38*
Team Members: Rebekah Fullard, Nikola Mutic, Joshua Van Der Veen

Table of Contents

Architecture - Class Diagram[1]



[1] High quality images can be found in the repository/zip in both PNG and PDF format in docs folder.

Design Rationales

Classes

Explain two key classes that you have debated as a team, e.g. why was it decided to be made into a class? Why was it not appropriate to be a method?

Game

- Since we decided to follow the MVC architecture, we decided that it would be appropriate to implement a Game class as the 'controller'. In the Game class we only implemented one method which is the positionSelected method. This method is implemented from the PieceListener interface. The Game passes itself through to the GUI so that the BoardPanel can call this positionSelected method when a piece is selected.
- In the positionSelected method, we implement turn based logic. Furthermore, we implement the appropriate game logic to move the piece. Using the MVC architecture, we can delegate the responsibility of moving the piece to the GameBoard, rather than in the Game class. We also call the redraw function to tell the GUI to update.
- In order to follow MVC design pattern, we decided that this logic should not be put in the Application class, rather a separate class is needed to handle this appropriately. Having this separate class also offers us better scalability in case we want to extend the features of the game. It also offers us better testability since there exists a separation of concerns.

UIBoardPosition, UIBoardPositionEmpty and UIBoardPositionFull

- Since UIBoardPanel is creating and drawing Board Pieces at Intersections on the board, a lot of this repetitive code/methods can be placed inside an abstract class. By doing so, we are allowing for code reuse which removes lots of duplication that we previously had. UIBoardPosition provides us with this abstract class that defines a draw method that draws it to the game board.
- UIBoardPositionEmpty and UIBoardPositionFull classes extend UIBoardPosition class and represent empty and occupied positions on the board. On our board, we decided to draw empty/unoccupied spaces with a smaller black circle, hence the override method in UIBoardPositionEmpty that sets the colour to black. Similarly with UIBoardPositionFull, we want to be able to draw a circle with the colour that represents the Piece owner.
- The inheritance relationship between UIBoardPositionEmpty and UIBoardPositionFull and UIBoardPosition allow for common functionality to be inherited, and specific functionality to be overwritten depending on the scenario. UIBoardPositionEmpty and UIBoardPositionFull use this

relationship to extend common functionality, but they use overrides to specify how things are drawn.
- The cardinality of these classes is such that there can be any number of UIBoardPositionEmpty and UIBoardPositionFull classes, which depends on the number of empty and occupied positions remaining on the board. However since UIBoardPosition is an abstract class, we cannot instantiate any instances.
- As discussed previously, these classes make use of inheritance. Therefore, this is an example of the template method design pattern.

Relationships

Explain two key relationships in your class diagram, e.g. why is something an aggregation not a composition?

Path - Intersection

- We recognised the strong Association between Path and Intersection is not a Navigable Association because Path knows about Intersection and Intersection knows about Path. This is recognisable by the fact that Path stores variables designating its source and destination Intersections, while Intersection stores the Paths that are attached to it in an ArrayList. This design decision was made to ensure ease of navigation between Intersections. If only each Path knew what Intersections they connected, then when we needed to move a Piece from one Intersection to another we would need to check through the list of all existing Paths to find the one connecting the relevant Intersections. Our implementation means that the Intersection can check what Intersections it is connected to by checking the Paths that it is connected to.

Intersection - Piece

- The Association between Intersection and Piece is an Aggregation rather than an Association because the Piece exists independently of the Intersection. This can clearly be demonstrated in the fact that a Piece can be moved from Intersection to Intersection, and in fact may not be on any Intersection at all before it is placed on the GameBoard or if it is removed from play by the opponent. Because an Aggregation describes a relationship in which an object isn't owned by a class it is associated with - and so can be shared - this is perfect to describe this relationship.

Inheritance

Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?

Inheritance was fairly heavily relied upon in our design. As well as extending several of the built in Java classes, we also created an engine which we extended in the game. This meant that we could create some base classes, but then add some extra functionality. For example, our chosen advanced requirement is the computer controlled player. We wanted a way to store all the information needed for both of them, but also have the ability to add extra logic to both since they have different roles. We decided instead of initialising all that information in the constructor, to just go with two separate classes that would inherit an abstract player class. This gave us those abilities we were looking for.

Cardinalities

Explain how you arrived at two sets of cardinalities, e.g. why 0..1 and why not 1...2?

Cardinalities between GameBoard and Piece:

- A Piece can exist on 1 GameBoard. In this case this is because the logic behind our GameBoard means that it tracks Pieces that aren't yet placed on the GameBoard. In real world terms this is reminiscent of keeping the Pieces next to the GameBoard. They are part of it even though they aren't yet placed in a specific position. Additionally, the GameBoard tracks 18 Pieces because it keeps track of each Piece before it is placed, while it is in play, and once it has been removed. Therefore the GameBoard is always aware of all 18 of its Pieces (9 belonging to each Player) and the state of play that they are in.

Cardinalities between GameBoard and Intersection:

- A GameBoard consists of 24 Intersections. This cardinality was arrived at because a Nine Men's Morris GameBoard will always have 24 Intersections. No Intersections can ever be added or taken away, and from the beginning of the game all intersections exist. An Intersection exists on only one GameBoard because for each Game there is only one GameBoard, and when any given GameBoard ceases to exist so too will those Intersections.

Design patterns

Explain why you have applied a particular design pattern for your software architecture? Explain why you ruled out two other feasible alternatives?

Applied design pattern

For the application we chose to use an MVC architecture. This decision was made because it is a good way to structure an application so that different areas of the application have different responsibilities, but also it is an industry standard so we aren't trying to reinvent the wheel. It also complies with the SRP where different areas of the application work together but have their own roles. This also means that the application could be extendable in the future for example a different UI could be implemented, and we could still keep the same backend.

Rejected alternative design patterns

One idea we had was to use a primarily view controlled structure where the UI would handle the logic as well as the view elements. This would have worked because most of the game actions are built around mouse press events, and having the mouse listeners call the functions with the logic required for the event would have made it quite easy to program. The problem here is that it breaks the SRP principle and would have created god classes that did too much.

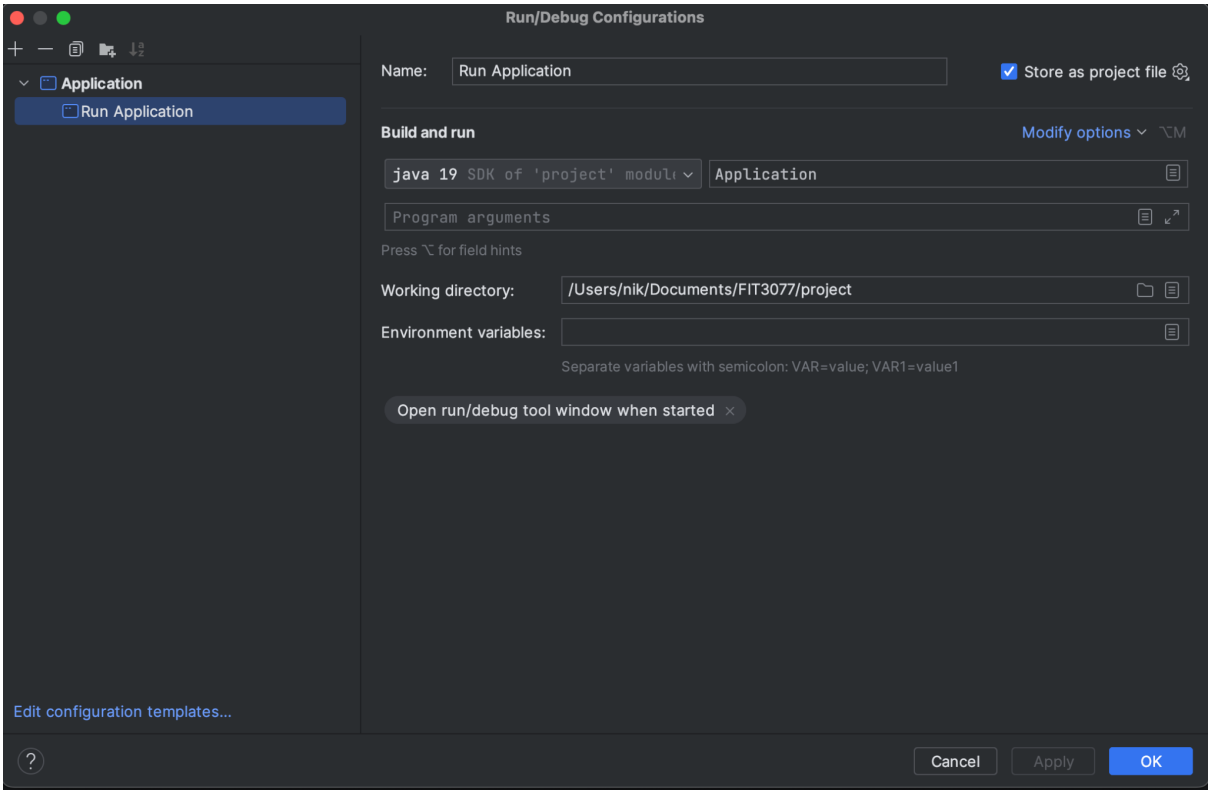
Another idea was to create observers that would subscribe to every event required, and register actions for the different events that took place. We found that this would be much more complicated to implement, and with some actions such as menu items, the bidirectional interaction of the backend logic with the frontend UI to show menus and information would be very complicated, so we decided against it.

Changes to UI design

The decision was made to change the UI to the framework Java Swing. This decision was made because the end product will be more user friendly, and easier to use. This will also increase the enjoyment factor of the end user (not assessed, but still important). An overwhelming majority or games have a graphical user interface, so it only made sense to implement one for our game. Although we did stray from the original design of creating a console application, the initial designs can be mimicked in the GUI implementation. There are some panels which have not yet been implemented, however that is due to the lack of requirements for those features (in this sprint).

How to run the application

First open the project in intellij. There is a run application configuration included in the project, but it can also be run in Intellij by running the Application main function. In configurations create a new configuration for application, and under build and run section, use Java 19, and select 'Application' file as the entry point. It should look something like this:



If you have done this successfully, you will be able to build and run the application using the Run 'Application' button on the top bar.

How to create and run an executable

1. Open the repository in Intellij
2. To build the executable, press Build > Build Artifacts > project.jar > Build
3. Then run the file at ***out\artifacts\project_jar\project.jar***

Notes:

- SomeGithubNoob is Josh Van Der Veen's personal Github account, and was accidentally not configured at the start.
- We did many 'Code with Me' sessions which is Intellij's built in collaboration tool, and because of that some of the team members didn't have as many commits. The commits were created by the person who was hosting the 'code with me' session, however we had a great amount of collaboration during the implementation of Sprint 2.