


Architecture Essay

"BetaGo"

History and Context of the Project

"BetaGo"¹ is a small open source Java project based off the ancient Chinese board game – "Go", where the aim is to occupy as much area of the board as possible. It was created by GitHub users "thomas1242" and "crougraff" and is designed to be played via an interactive GUI. BetaGo is able to be played as a 2 player game, or also as a human vs computer experience, AI implementation. Go known to be a particularly difficult game to create an accurate AI for - in fact, AlphaGo, a program developed by Google has only beaten a top-ranking Go player a handful of times², so it is likely that the creators of BetaGo took on the project to test their game building skills, rather than to create a world-class AI.

BetaGo is a fairly recently created and seems to be a collaboration between two individuals who started the project for personal practice outside of work or study. The initial commit was on the 1st March 2018, and since then there have been only 34 other commits, appearing every so often, hinting that this is indeed a personal project, rather than it being work or study related. Looking at all the commits of the GitHub project repository, it is apparent that this project is building off another project by another user, "mikedaaabest". Their README.md has been edited delete mikedaaabest's cloning instructions and has replaced them with their own (see opposite), as well as large dumps of tests and source code just appearing in the space of 1 day. The commit messages, assertions, and deletions make it clear that this new version of BetaGo aims to improve and add functionalities to the mikedaaabest's project, rather than refactoring it to make it architecturally better.



```
3 3  ## Compilation and running
4 4
5 5  ```javascript
6 6  -git clone https://github.com/mikedaaabest/BetaGo/
6 6  +git clone https://github.com/thomas1242/BetaGo/
7 7  find ./BetaGo/src/sample/* | grep .java > argfile
8 8  javac @argfile
9 9  java -cp BetaGo/src/ sample.Main
```

Figure 1

Looking at all the commits of the GitHub project repository, there has been next to no large code redesign and it is clear that the programmers wanted to incorporate the Model View Control (MVC) design pattern from the start, and built their code around this model, rather than writing it, then remodelling it to fit the pattern later.

- 1 March 2018: Initial commit, cloned previous programmer's BetaGo implementation
- 2 March 2018: Improved the visual design of the GUI, refactored the Board code (so much so, that it hints that the previous implementation by mikedaaabest did not work)
- 5 March 2018: Refactoring win screen
- 6-12 March 2018: Visual redesign of CSS
- 23 March 2018: Added code to calculate scoring
- 26 March – 1 April 2018: Added a difficulty slider to set the level of difficulty the player will be playing at

¹ GitHub (7 May 2018). *BetaGo*. Retrieved from: <https://github.com/thomas1242/BetaGo>

² Fortune. (7 May 2018). *Google's Go Computer Beats Top-Ranked Human*. Retrieved from: <http://fortune.com/2016/03/12/googles-go-computer-vs-human/>

- 18 - 24 April 2018: Added a Monte Carlo Tree Search (MCTS) algorithm to calculate the best move to take by the AI, updated the GUI.

Since the goal of the project has always been to create a digital version of Go, the overall purpose of the project has not shifted from its original purpose and is unlikely to do so since the project is fairly un-extensible and isn't meant to fulfil multiple purposes. However, a foreseeable future for the project is extending it into a project that allows the user to play a variety of traditional games such as Go, Chess, etc.

Domain of the Project

BetaGo is designed to be a digital version of a game that is primarily played on a 19x19 sized board. Its domain is purely entertainment-based, because it is a tactical game that teaches no skills apart from the ones directly related to the game itself. It could be argued that BetaGo teaches tactics that can be used in other games such as Chess, and is a way to keep one's brain alert and sharp, but on surface level, the game's domain poses only a recreational purpose. The project targets users of all ages since the rules are so simple, however a key aspect of the project is of course that it is a digital rendition of a very classic game that is not commonplace in today's society, especially amongst youth. The new digitised platform means that it is more attractive to a new generation of players, so the project targets this audience in particular.

The game is intended to be built and run on a Java IDE, or any device with a Java Virtual Machine, and requires Java 8 and Junit 5 to run the tests. It is played on a GUI which makes it easy to play and understand, so its suited well to those who are not experienced with command line interfaces. I ran the program on Eclipse Oxygen from my personal MacBook, and here is proof of execution (name in the console):

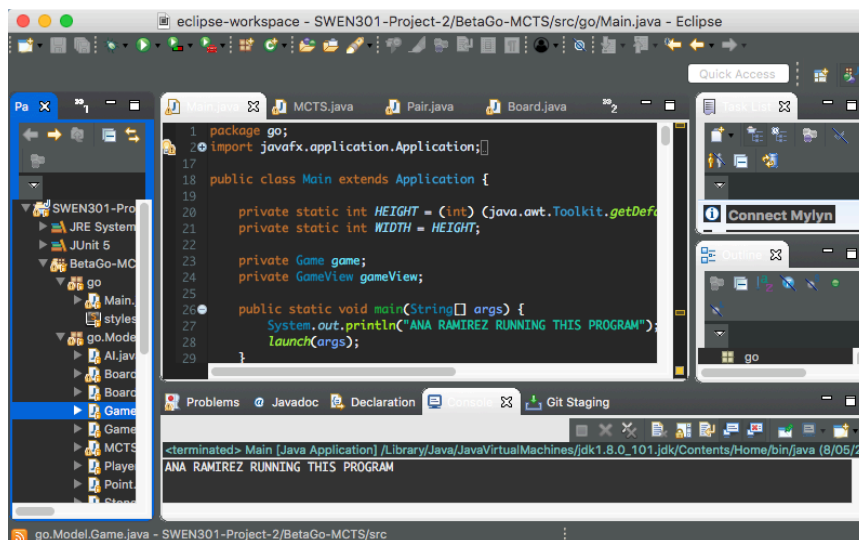


Figure 2

Initially I was running the project on Eclipse Neon, but soon discovered that Neon did not support Junit 5, so I updated to Oxygen and everything ran smoothly from here.

Reflecting on how BetaGo can only be run on a Java IDE, this platform could potentially be a constraint on the usability of the program. Since the project domain is entertainment based, the users of the game will likely value ease-of-use, therefore a mobile device may be a better

platform on which to run this project. Therefore the need for a Java IDE may deter the enthusiasm of an average user.

Component Architecture

The architecture of the BetaGo project seems to be centred around Event Driven architecture since it is run on a GUI, and has listeners to be constantly checking for a change in state initiated by the user, as well as a loosely implemented Model View Controller (MVC) architecture system. MVC architecture focuses on separating the system up into 3 separate parts – the model, the view, and the controller in order to promote ease of understanding when reading the code. The model is responsible for representing the projects object entities, and represents a lot of the data that the user works with, the view provides the interface of the program, and the controller takes user input and communicates with the model to change the view to reflect the given input³. The intention of the MVC is to improve the separation of concerns of the system, which involves insuring that each component is responsible for only one thing, rather than attempting to do a magnitude of tasks. BetaGo has attempted to use MVC by successfully separating the model away from the view and controller, but the lines between the view and the controller have become blurred. The Main class carries all the responsibilities of the view and the controller, making it a fairly large class with numerous responsibilities, leading to low cohesion.

Looking at the dependency diagram (opposite), it is clear that the Game class is a focal dependant point for almost all of the entire program, and is integral to most of the model. All parts of the model are appropriately distributed out amongst many classes, whereas the view and controller are compressed into the Main class, which creates Game, producing the model. Because of this incorrectly implemented MVC, it is difficult to see the flow of communication between the separate components of the system, except for the obvious communication between all parts of the model.

The component diagram below allows us to look at the system's architecture in a more general way, and allow us to look at the large components and how they interact with different parts of the system

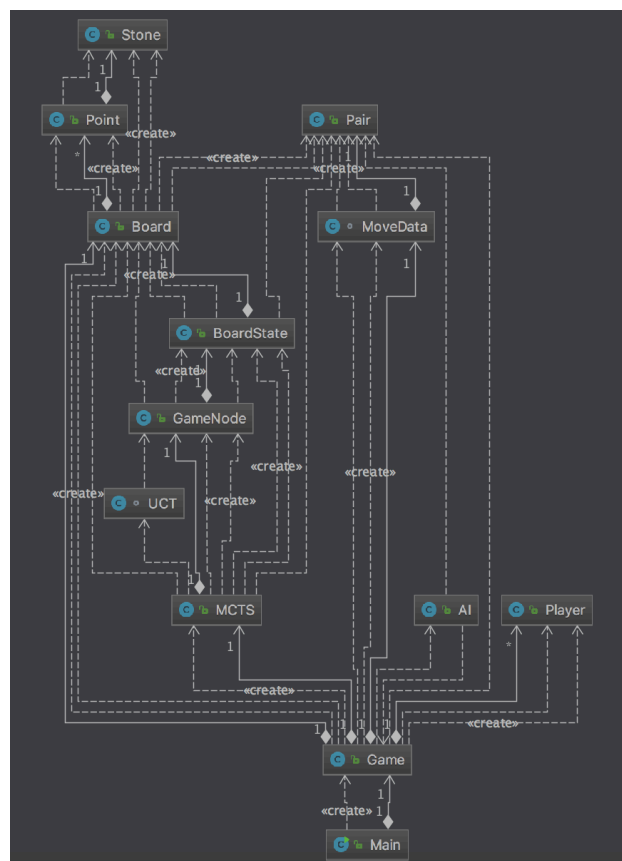


Figure 3

³ Tutorials Point (9 May 2018). *MVC Framework – Introduction*. Retrieved from: https://www.tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm

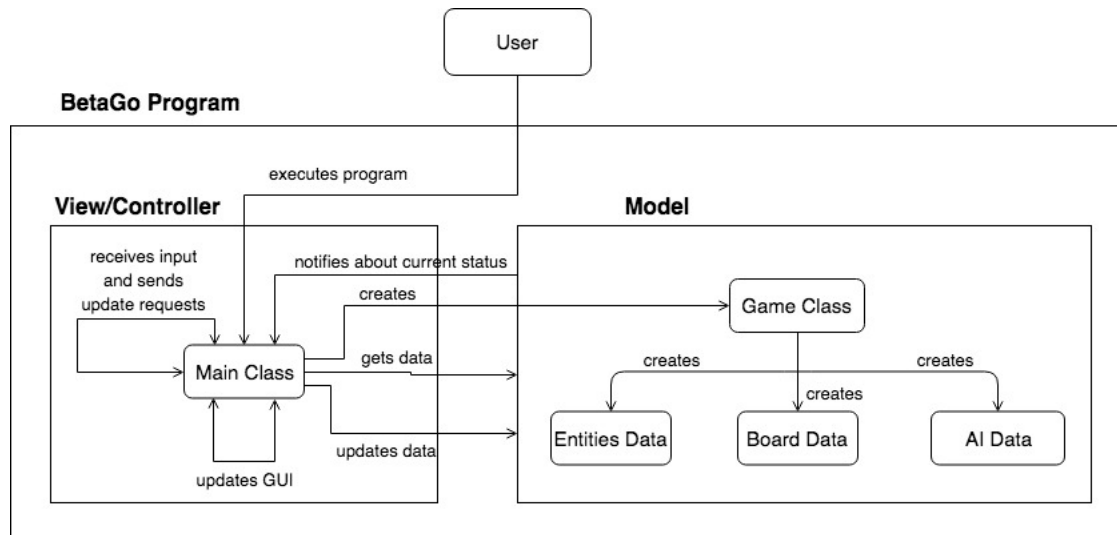


Figure 4

The user is able to run the program through a Java IDE, and play the game through the programs GUI. From here, the Main class creates the Game class which is responsible for many tasks, including creating most of the game related objects and the main entities in the program. The Model is where most of the core data is stored for the entities but is not responsible for any reactive behaviour of these entities, meaning that the model is reliant on the Main class (which acts as the view and the controller) in order for the objects to change and function

The Main class is responsible for creating the GUI for the system and controlling the visuals. The View component is comprised of the GUI, which is created by the JavaFX library, and renders the data from the Model into an interface that is fluid and easy to use by the player. The Controller takes input from the user and makes calls to update the view corresponding to the actions performed by the user.

It is clear to see that there is some separation of concerns within BetaGo, but within the Main Class, there is still some entanglement of responsibilities that could be more easily understood and maintained by splitting them apart.

Data Structures

When analysing the important data structures of the BetaGo program, it is useful to reflect back on Figure 3, the dependency diagram for the program. It is clear when looking at this that there is a lot of dependency by many classes on the Board class, suggesting that there are some important structures here that are used by a lot of the system. The Board class contains a 2D array of Point objects, and the 2D array makes up the board of the game. At the beginning of the game, the user is able to select whether they would like a 19x19, a 13x13, or a 9x9 board, so the 2D array size changes game-to-game depending on user input. The board 2D array is fundamental to the system because it provides a context to the game and is the board that the player is interacting with, and is key to the logic of the game for placing stones, removing defeated stones, and winning the game.

Another key data structure is the list of Stones in the Stone class, representing the adjacent stones next to another Stone. Since the aim of BetaGo is to gain as much board area as possible, a feature of the game is that stones from another player are able to be captured and removed if the stone is surrounded by your own stones. Therefore, the list of adjacent stones is vital in order to defeat the opposing player to progress towards winning the game. The list is important in bringing an element of strategy to the game as it makes the game more complex than just aiming to gain territory.

Reflecting on the class diagram below, it is clear that the Main class is a key data structure because it stores all of the nested classes that create the GUI of the game. Without the Main class storing all of the nested view classes, there would be no user interface and no way to play the game, making it an integral part of the system.

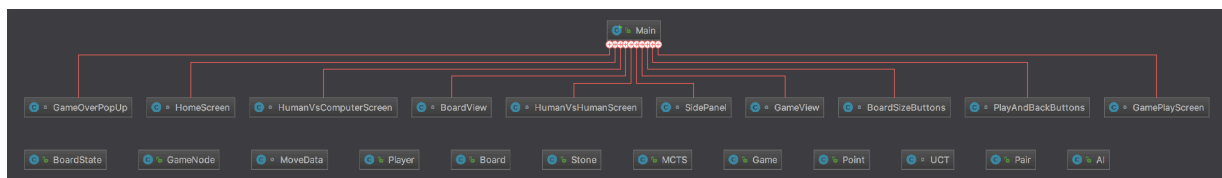


Figure 5

BetaGo uses the MCTS algorithm to create the AI that the user is playing against. The MCTS algorithm builds a tree to represent all possible states of the game. For each node (state of the game), there is another node representing another state if its parent node is executed. By using the MCTS algorithm, the algorithm not only calculates which move to make, but also the number of moves that can be made in the current state. Without this tree data structure, there would be no AI, and no 'intellectual' entity for the player to compete against, so the game would be reduced to a 2 player game only, rather than having a player vs computer option also.