Semester 1, 2024

FIT3077 – Software engineering: Architecture and design

Sprint 4: Game Extensions and Reflection of Sprint 3 Design

Team: Chitbusters

Members: Brandon Luu, Jeremy Ockerby, Max Zhuang, Raymond Li

Date of Submission: 6 June 2024

Table of Contents

Contributor Analytics **3**

Self-Defined Game Extensions **4**

Players Fight for Space During Collision 4

Live Leaderboard - Human Value 4

Updated Class Diagram **5**

Sprint 3 Reflection **6**

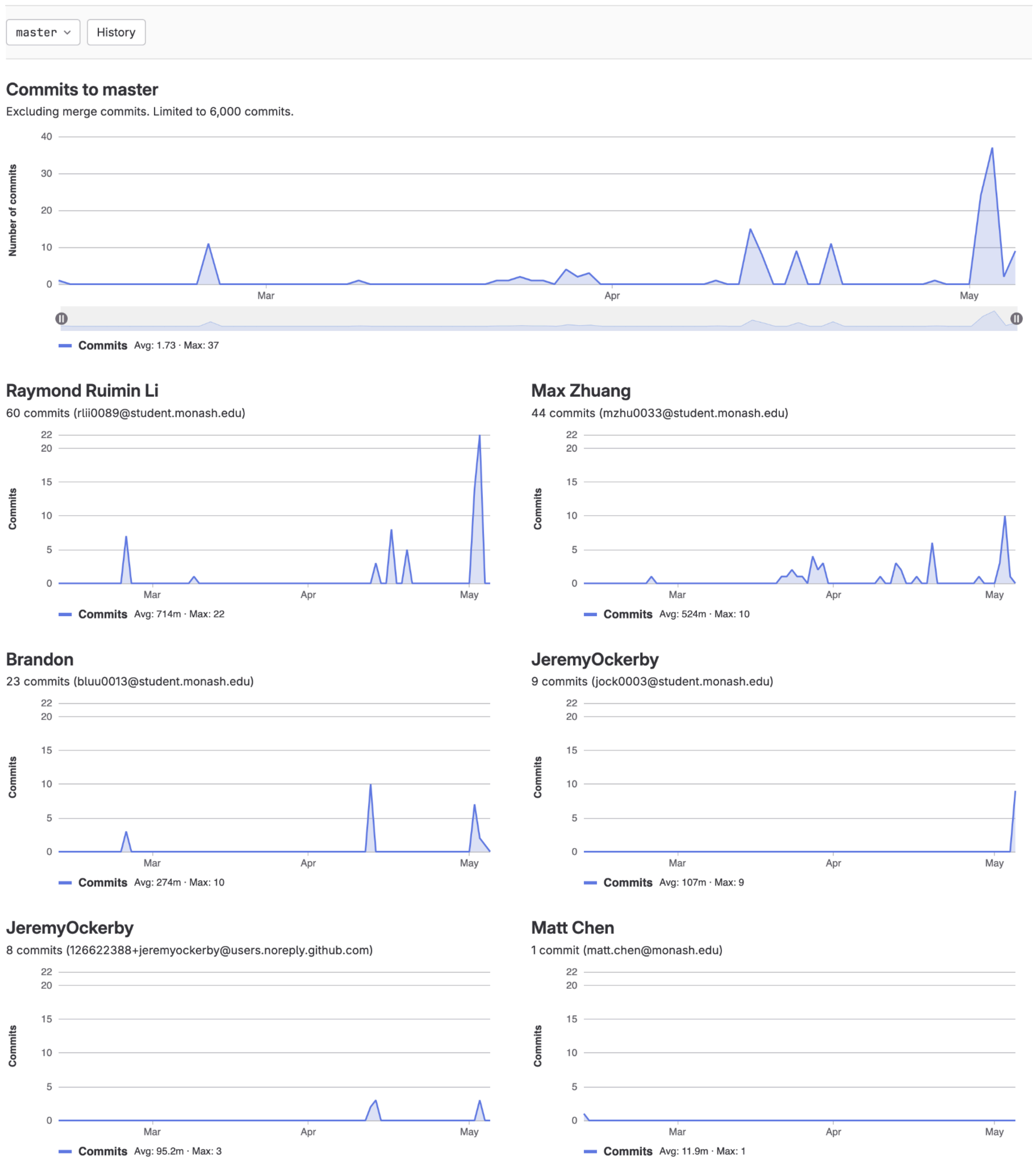
Executable **9**

Description 9

Run Instructions 9

Creation Instructions 9

Contributor Analytics



Commits to master

Excluding merge commits. Limited to 6,000 commits.

Commits Avg: 1.73 · Max: 37

Raymond Ruimin Li

60 commits (rlii0089@student.monash.edu)

Commits Avg: 714m · Max: 22

Max Zhuang

44 commits (mzhu0033@student.monash.edu)

Commits Avg: 524m · Max: 10

Brandon

23 commits (bluu0013@student.monash.edu)

Commits Avg: 274m · Max: 10

JeremyOckerby

9 commits (jock0003@student.monash.edu)

Commits Avg: 107m · Max: 9

JeremyOckerby

8 commits (126622388+jeremyockerby@users.noreply.github.com)

Commits Avg: 95.2m · Max: 3

Matt Chen

1 commit (matt.chen@monash.edu)

Commits Avg: 11.9m · Max: 1

Figure 1. Contributor Analytics Chart (Accurate as of 05/06/2024)

Self-Defined Game Extensions

Players Fight for Space During Collision

The first creative extension implemented in this game is a mechanic where players fight for a space when a collision occurs. Currently, if a player chooses a chit card correctly and moves onto a tile which is occupied by another player, their turn will be ended without moving around the board. This extension instead asks the player whose turn it is to input either the number 0 (heads) or 1 (tails). The game will randomly generate one of these two numbers and if there is a match, the players will switch positions on the board.

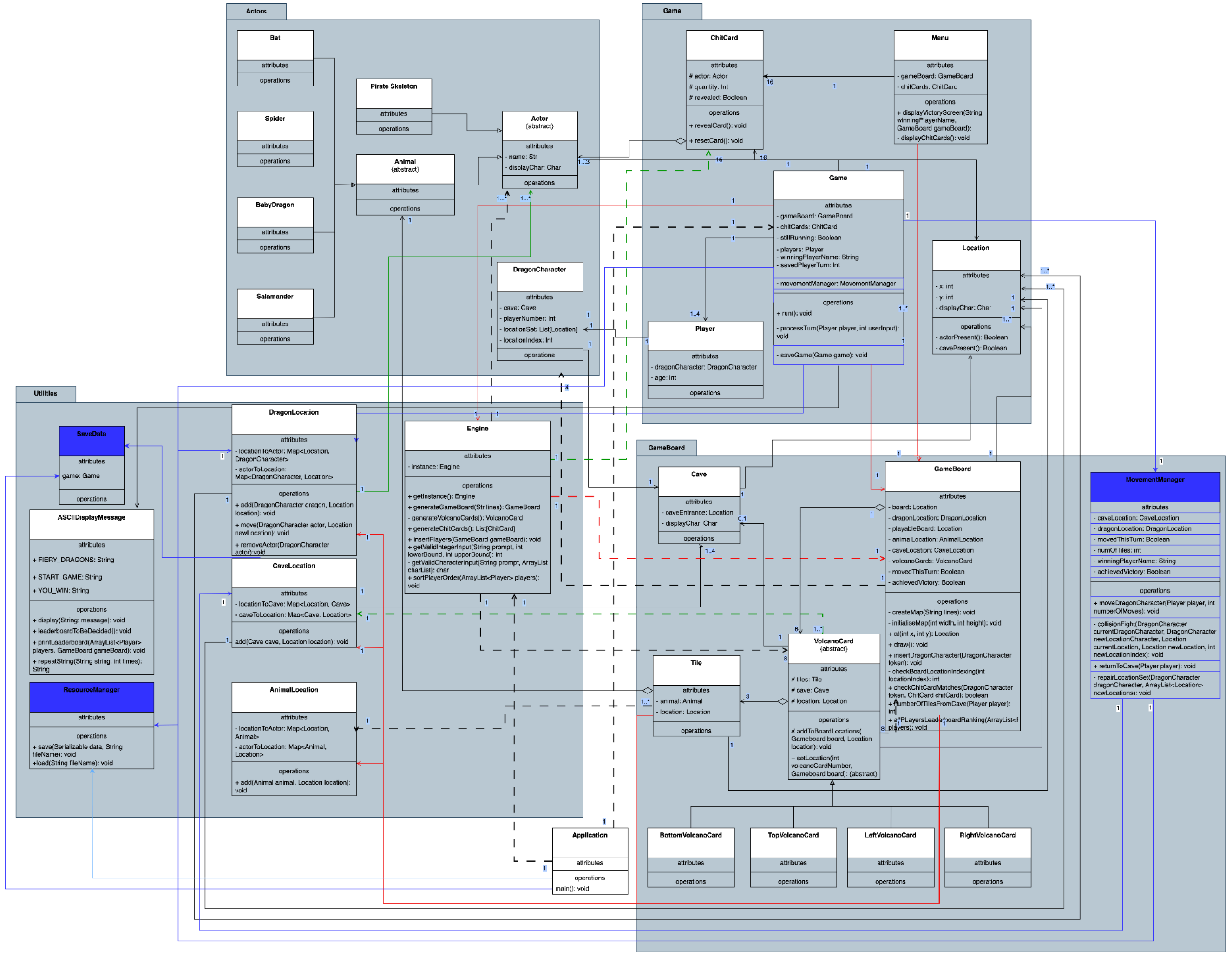
Some constraints needed to be added to ensure balance in the game as swapping players in certain situations could severely ruin the progress of a player and make the game less enjoyable. A collision fight swap will not occur if a player chooses a pirate skeleton card and ends up colliding with another player whilst moving backwards as the other player could be swapped in front of their cave and have to do another lap of the board again, diminishing the enjoyment of the game. Another scenario where a collision fight swap will not occur is when a dragon character collides while starting the game in their cave, as when swapped a different player will end up in the wrong cave.

Live Leaderboard

One of the creative extensions implemented into the game is a dynamic leaderboard. While the main aim of the game is for the player to travel around the game board and return to their cave, the leaderboard adds an additional layer of immersion. At the end of each turn, the leaderboard dynamically updates to show all players, ranked in order of proximity to their respective cave and the number of tiles for them to reach it. This feature not only offers valuable information to the players, it hopefully also increases the competitive spirit between players by showing the progress of everyone.

The inclusion of the leaderboard is chosen to address the human value ‘social recognition’. The game calls into our human nature for comparison by providing the players with a visual representation of their progress relative to others. When a player moves up the leaderboard, it creates a sense of power and achievement when compared to other players.

Updated Class Diagram



Actors
Bat
attributes
operations
Spider
attributes
operations
BabyDragon
attributes
operations
Salamander
attributes
operations
Pirate Skeleton
attributes
operations
Animal {abstract}
attributes
operations
Actor {abstract}
attributes
- name: Str
- displayChar: Char
operations
DragonCharacter
attributes
- cave: Cave
- playerNumber: Int
- locationSet: List[Location]
- locationIndex: Int
operations
Game
ChitCard
attributes
# actor: Actor
# quantity: Int
# revealed: Boolean
operations
+ revealCard(): void
+ resetCard(): void
Menu
attributes
- gameBoard: GameBoard
- chitCards: ChitCard
operations
+ displayVictoryScreen(String winningPlayerName, GameBoard gameBoard):
- displayChitCards(): void
Game
attributes
- gameBoard: GameBoard
- chitCards: ChitCard
- stillRunning: Boolean
- players: Player
- winningPlayerName: String
- savedPlayerTurn: int
- movementManager: MovementManager
operations
+ run(): void
- processTurn(Player player, int userInput): void
- saveGame(Game game): void
Player
attributes
- dragonCharacter: DragonCharacter
- age: int
operations
Location
attributes
- x: int
- y: int
- displayChar: Char
operations
- actorPresent(): Boolean
- cavePresent(): Boolean
Utilities
SaveData
attributes
game: Game
operations
ASCIIDisplayMessage
attributes
+ FIERY_DRAGONS: String
+ START_GAME: String
+ YOU_WIN: String
operations
+ display(String: message): void
+ leaderboardToBeDecided(): void
+ printLeaderboard(ArrayList<Player> players, GameBoard gameBoard): void
+ repeatString(String string, int times): String
ResourceManager
attributes
operations
+ save(Serializable data, String fileName): void
+load(String fileName): void
DragonLocation
attributes
- locationToActor: Map<Location, DragonCharacter>
- actorToLocation: Map<DragonCharacter, Location>
operations
+ add(DragonCharacter dragon, Location location): void
+ move(DragonCharacter actor, Location newLocation): void
+ removeActor(DragonCharacter actor):void
CaveLocation
attributes
- locationToCave: Map<Location, Cave>
- caveToLocation: Map<Cave, Location>
operations
add(Cave cave, Location location): void
AnimalLocation
attributes
- locationToActor: Map<Location, Animal>
- actorToLocation: Map<Animal, Location>
operations
+ add(Animal animal, Location location): void
Engine
attributes
- instance: Engine
operations
+ getInstance(): Engine
+ generateGameBoard(Str lines): GameBoard
- generateVolcanoCards(): VolcanoCard
+ generateChitCards(): List[ChitCard]
+ insertPlayers(GameBoard gameBoard): void
+ getValidIntegerInput(String prompt, int lowerBound, int upperBound): int
- getValidCharacterInput(String prompt, ArrayList charList): char
+ sortPlayerOrder(ArrayList<Player> players): void
Application
attributes
operations
main(): void
GameBoard
Cave
attributes
- caveEntrance: Location
- displayChar: Char
operations
GameBoard
attributes
- board: Location
- dragonLocation: DragonLocation
- playableBoard: Location
- animalLocation: AnimalLocation
- caveLocation: CaveLocation
- volcanoCards: VolcanoCard
- movedThisTurn: Boolean
- achievedVictory: Boolean
operations
- createMap(String lines): void
- initialiseMap(int width, int height): void
+ at(int x, int y): Location
+ draw(): void
+ insertDragonCharacter(DragonCharacter token): void
- checkBoardLocationIndexing(int locationIndex): int
+ checkChitCardMatches(DragonCharacter token, ChitCard chitCard): boolean
+ numberOfTilesFromCave(Player player): int
+ allPLayersLeaderboardRanking(ArrayList<P players): void
Tile
attributes
- animal: Animal
- location: Location
operations
VolcanoCard {abstract}
attributes
# tiles: Tile
# cave: Cave
# location: Location
operations
# addToBoardLocations(Gameboard board, Location location): void
+ setLocation(int volcanoCardNumber, Gameboard board): {abstract}
BottomVolcanoCard
attributes
operations
TopVolcanoCard
attributes
operations
LeftVolcanoCard
attributes
operations
RightVolcanoCard
attributes
operations
MovementManager
attributes
- caveLocation: CaveLocation
- dragonLocation: DragonLocation
- movedThisTurn: Boolean
- numOfTiles: int
- winningPlayerName: String
- achievedVictory: Boolean
operations
+ moveDragonCharacter(Player player, int numberOfMoves): void
- collisionFight(DragonCharacter currentDragonCharacter, DragonCharacter newLocationCharacter, Location currentLocation, Location newLocation, int newLocationIndex): void
+ returnToCave(Player player): void
- repairLocationSet(DragonCharacter dragonCharacter, ArrayList<Location> newLocations): void

Figure 2. Updated Class Diagram

****5/6 CHANGE IF UPDATED*****

Sprint 3 Reflection

Extending upon the implementation of sprint 3, we added 4 new additional functionalities to the game. The first is a new chit card containing a snake, that when flipped over, will move the player backwards until it finds an empty cave and slots that player into that empty cave. The second was the ability to save a game and subsequently load the saved game in future play. The third was the inclusion of a 'collision fight' where two players will fight for the better tile when a collision occurs in the form of a coin flip. Lastly, a live leaderboard was added to show which player is closest to winning as well as how many tiles each player has left to adhere to the human value of social recognition. Overall, these extensions were not too difficult to implement, with each one having to add a few additional methods and slight refactoring to the sprint 3 implementation.

When implementing the snake feature, it was quickly discovered that a method that returns all the caves in the will be required. The class 'CaveLocation' from sprint 3 which held all the caves along with their locations was a quick solution to this issue, as a new method 'getAllCaves' was added to this which returns all caves in the game. Similarly, our sprint 3 implementation had each cave holding the location of the tile directly in front of it as an attribute, thus retrieving these 'cave entrances' was not an issue either. As each dragon character holds a list of the sequence of locations that dragon character will travel through, as implemented in sprint 3, simply iterating backwards through that list and comparing it to all the 'cave entrances' was sufficient to find the nearest empty cave. However, an issue that was not directly extensible from the sprint 3 implementation was the case that a dragon character might fall behind its starting cave, if another dragon character flips over the snake card and falls into its starting cave. This was a big issue as the dragons location set will need to be modified throughout the game, which was not possible without refactoring sprint 3's implementation. Thus, whilst the majority of this extension was not difficult to implement due to a strong distribution of system intelligence from sprint 3, simply relying on the single array list 'locationSet' for movement, proved to be inadequate when extensions involving falling behind a starting cave is incorporated. In the future, having multiple array lists with one holding the initial sequence of locations, and the other holding the current sequence of locations for the dragon character could be used, allowing the location set to be in a constant state of change, thus allowing movement of dragon characters to be more flexible. Additionally, having a movement manager class that is responsible for all movements (implemented in sprint 4 but not 3), could also be implemented rather than having the GameBoard class handle the responsibility of moving the characters along with its other responsibilities. This will not only allow for more flexibility in movement of dragon characters but also ensure the Single Responsibility Principle is kept.

Being able to save a game state and load a previous one seemed to be quite a difficult task at first. The initial plan was to get all instances of all classes, store them in a separate class, then store that into a save file. This was not used as our sprint 3 implementation had the 'Game' class store all relevant information about the current game state, meaning that only the current instance of the 'Game' class would need to be stored. In terms of adjustments made to the sprint 3 implementation to accommodate this new change, almost all classes were made to have 'implements Serializable' which allowed all aspects of the game to be saved using 'ObjectOutputStream'. New classes would hold the saving and loading methods, being the 'ResourceManager' class, but would be accessed by other classes to allow for inputs such as the save file name that would be determined by the users.

Saving the current state of a game would be an option available to the users during the game. Once selected, this process would be facilitated by the new method in the ‘Game’ class called ‘saveGame’, , which would allow the users to type a name for the save file. This save file would be saved as a ‘.save’ file, adhering to the specifications of this sprint.

This saving implementation also allowed for save files to be read by using ‘ObjectInputStream’. However, an issue arose when initially trying to load a save file. The sprint 3 implementation had the some classes, such as the ‘CaveLocation’, ‘DragonLocation’ and ‘AnimalLocation’ classes, were made to be static variables, making them unable to be ‘serializable’. This would result in all aspects of the game still functioning as normal, however the board would not display anything. Remedying this was as simple as making these classes not static, allowing them to be saved properly, and therefore allowing the game board to be loaded properly. In terms of selecting an option to load a previous game, at the beginning of the game, players will be given the option to start a new game or load from save. When load from save was selected, it will then ask users to type the name of the save file, which would load that save file, given it exists.

When implementing the collision conflict feature of the game, there were existing parts of movement and location that caused some difficulty in swapping players. Within the code written during Sprint 3, there were already measures to detect if a player was on a tile and implementing a collision conflict method was easily called within this section. Adding the constraint that a collision conflict could not occur when a player was moving backwards was simply mandated using the existing ‘numberOfMoves’ attribute. Gathering the needed DragonCharacter instances of players involved and the respective locations of each player for the collision fight was easily found due to the current design of the system. As the game board contains tiles with a particular location, the DragonCharacter actor on the conflict tile is easily extracted along with the location. Actually swapping the two DragonCharacter instances when a collision fight finished was more challenging as the existing way that the location of each character is stored within the game was quite confusing. Each DragonCharacter instance contains a set of location instances which reflect the location of the character throughout the game and understanding how this worked required explaining from other team members. Updating the location of the character who is moving along the board before the collision followed the existing means of updating the location but updating the player who needed to go backwards was more complicated when they fell past their starting cave. A repairLocationSet method was used to fix this problem, which was also used to handle similar issues with the snake chit card extension, combined with other statements that determined when a player would be in this scenario. Given that the existing code for moving characters was written within the GameBoard class, there were too many different operations which broke the single responsibility principle. A MovementManager class was created during this sprint with the responsibility of controlling all movement of characters within the game, taking the capability off the GameBoard class.

Adding a leaderboard and its functionality to the game was a moderate challenge, The main task involved calculating and displaying player distances from their respective caves. The complexity of this task mainly involved two aspects: ensuring the accurate calculations and presenting the leaderboard in an aesthetically pleasing manner.

The calculation of player distances required a new method ‘numberOfTilesFromCave’ to be created to determine the number of tiles each player is from their cave. This involved checking the location indexes of both the player and their cave. Initially, the implementation seemed straightforward but there were some complications. The code became quite repetitive as it was comparing the number of tiles from the cave for each player. This redundancy needed to be optimised. Planning the functionality required careful consideration of the game's structure and logic. Despite the initial challenges, the actual implementation of the distance calculation was manageable when coding it.

The main difficulty became ensuring that the method was efficient, easy to maintain and extendable.The main challenge was in the presentation of the leaderboard. It was critical to ensure that the leaderboard was not only functional, but also user-friendly and appealing. The first iteration of the leaderboard functioned correctly but lacked the aesthetics which can detract from the user experience. To address this, I implemented an ASCII leaderboard, however this approach also introduced new problems. The hardcoded template did not fit player names of varying lengths which led to formatting issues. After researching possible solutions, I leveraged the Math library to dynamically adjust the size of the leaderboard based on the longest player name. This approach ensures that the formatting is consistent and improves the overall user experience. There were several aspects of the Sprint 3 implementation that influenced the ease of implementing the leaderboard.

The existing system had a well distributed structure with clear separation of concerns. This modular design allowed for the new methods to be added without much code refactoring. However, some areas lacked abstraction.

There were some code smells particularly with the repetitive code. These were addressed during the extension by refactoring the calculation logic, but the presence of these smells initially slowed down the extension process.

<u>Executable</u>

Description

The game executable has been packaged into a Javar Archive (JAR) file format to ensure compatibility across multiple platforms that support Java runtime environments. It was last tested in maOS 14.5 (23F79) running JDK 22.0.1, but is designed to run seamlessly on, but not limited to, macOs, Windows, Linux.

Run Instructions

To run the game, please follow the instructions below:

1. Install the ‘Game.jar’ file from the repository or Moodle submission.
2. Open terminal or command prompt.
3. Navigate to the file directory. ‘cd [directory path]’.
4. Run the game using the command ‘java -jar Game.jar’ and return.

Creation Instructions

To create the executable file from the source code, follow the instructions below:

1. Download or pull the source code from the repository.
2. Navigate to project structure in the IDE.
3. Select artifacts and create a new artifact.
4. Create it of type ‘JAR’ and add dependencies such as the main class (Application.java).
5. Build the artifacts and locate the executable in ‘~/out/artifacts/Game_jar/Game.jar’