SPRINT 2

Harshath Muruganantham - 33114064

FIT3077– Software Engineering: Architecture and Design
Faculty of Information Technology, Monash University

Table of Contents

INTRODUCTION

The video submission is submitted as a separate file on model. If the file does not work, the link to the video is provided below:

Video Link: https://drive.google.com/file/d/1NzJXKwzUMkhXZQB0_xWd1Y0V-5cyUUy4/view?usp=drive_link

Make sure to select the video quality as 1080p in the above link. The submitted video is an mp4 file.

If the Class Diagrams or Sequence Diagrams shown below are illegible, their original link and an interactive canvas are linked below:

Class/Sequence Diagrams: https://lucid.app/lucidchart/86e41448-72f9-4df1-a5d6-a461f9aeb0d2/edit?viewport_loc=-6430%2C-3500%2C14400%2C7381%2CHWEp-vi-RSFO&invitationId=inv_da0b35fe-dec6-4317-98a2-c82c414cd429

An executable file is submitted along with this Moodle submission.

The executable file exists under the 33 CL_Monday06pm_Team697-Harshath-Sprint2.zip file. This file contains 4 unique folders:

- Build
- Dist
- Project

The Project folder contains all the source code pertaining to this project. The **dist** folder contains the required executable file. The executable file is named **GameInitialisation.app** under the **dist** folder.

This executable file will only run on a **MacOS** machine.

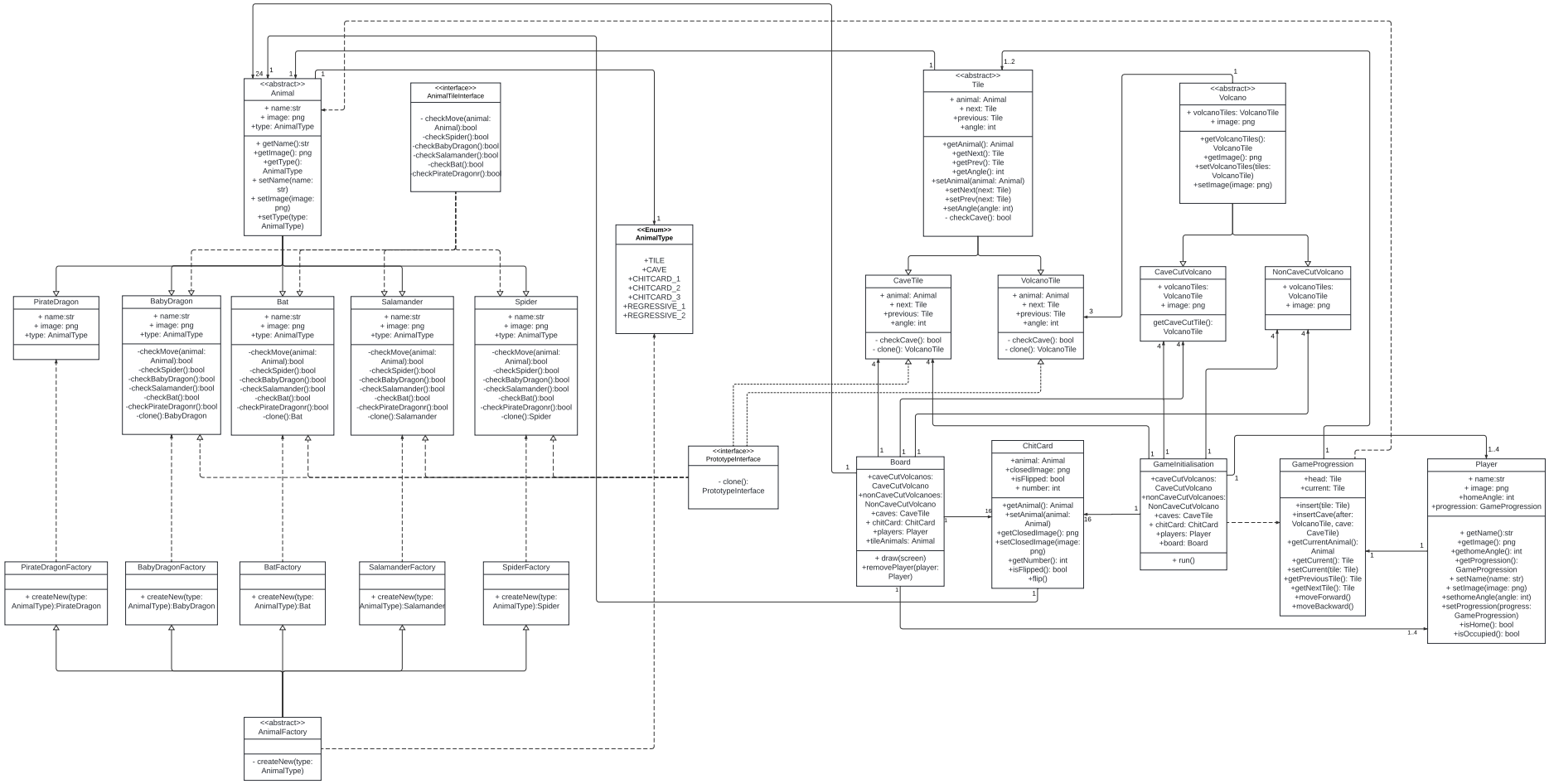
If there exists a privacy warning on the mac, please go into System Settings, Privacy and Security and click Open Anyways on the mac.

The game functionality implemented in this code are:

- set up the initial game board (including randomised positioning for dragon cards);
- movement of dragon tokens based on their current position as well as the last flipped dragon card;

In order to build an executable of this game from the source repo, please run: {python setup.py py2app} with the module py2app installed.

CLASS DIAGRAM



KEY FUNCTIONALITIES

The key classes in this implementation are Animals (Baby-Dragon, Salamander, Bat, Spider and Pirate Dragon), their respective factory classes, Tiles (Cave-Tile and Volcano Tile), Volcanos (Cave-cut and non-cave cut), Animal-Type, Board, Chit-Card, Game-Initialisation, game-Progression and Player as shown in the UML class diagram. Two interfaces were also used, the Animal-Tile Interface and the Prototype interface, both of which are explained in detail in the design patterns section. In the class diagram above, a few noteworthy aspects are that only new methods are included in each class if it's a child class (i.e. methods already defined in the parent class are not included under the child class methods) and the symbol '-' denotes abstract method and '+' denotes normal method/attribute.

REQ. 1

Set up the initial game board (including randomised positioning for dragon cards);

The game starts with the Game Initialisation class being initialised with its required inputs of:

- 4 Cave Cut Volcanos – Each Cave Cut Volcano contains three volcano tiles, with the middle volcano tile being "cave-cut" meaning it has a cave attached to it. Each volcano tile contains 1 animal as specified in game instructions.
- 4 Non-Cave Cut Volcano – Each Non-Cave Cut Volcano contains three volcano tiles. Each volcano tile contains 1 animal as specified in game instructions.
- 4 Caves – Each Cave contains a unique animal as specified in the game directives.
- 16 Chit Cards – Each chit card also contains a unique animal as specified in the game directives.
- 1 – 4 Players – The game can be played with 1 – 4 players.

The random module from python will be used inside the initialisation (constructor) function of Game Initialisation class where each of the above inputs will be randomly shuffled to adhere to the game standards of randomly placing volcanos, chit cards and caves on the map.

Furthermore, each Player will have their own Game Progression attribute (which is a circular doubly linked list containing each tile on the board) as each player will have their own progression from their home cave, around the map, back to their home cave, independent (to a degree – explained more in REQ 3 section) of other players. The current value of that linked list will be set to the player's starting Cave through traversing through the CaveTiles input and assigning one unique CaveTile to each player. This Cave will be the starting position of the player and the player will need to reach back to this cave to "finish" their game. This Game progression attribute (or linked list) will be formed inside the constructor class of the Game Initialisation method with each volcano tile being appended to the Game Progression linked list to form a fully complete "allowable moves" linked list. This helps avoid the Player class being a god class.

Doing the above determines the order of the game, i.e. where each Animal in each volcano is placed (which will be stored in a list called tileAnimals), and this information would also need to be extracted for the board to be drawn.

A Board class is then instantiated within Game Initialisation.

All this information (4 Cave Cut Volcanos, 4 Non-Cave Cut Volcano, 4 Caves, 16 Chit Cards, 1 – 4 Players, and the Animals in each Tile) is passed onto the Board instantiation, and the board class will be used to individually draw each component onto the pygame window/surface (more specifically the draw() function within Board).

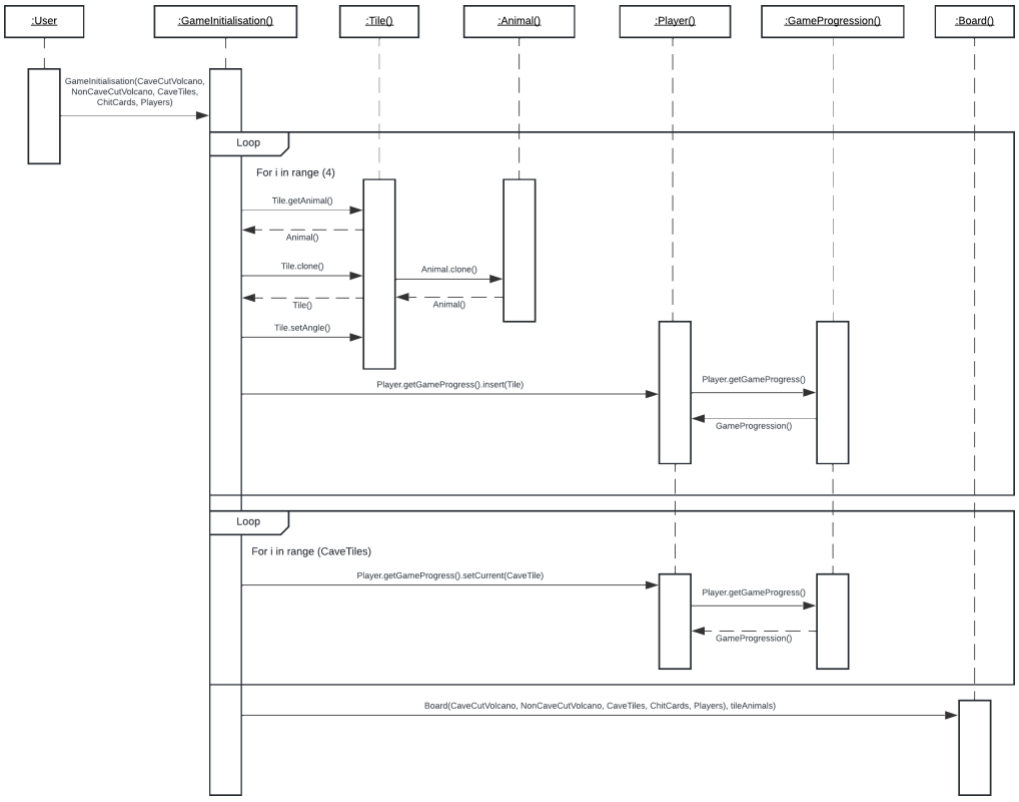
Each of the above-described objects contain an image attribute (for Animals that image attribute will be based on the type of Animal – AnimalType enum they were instantiated with) and that attribute will be accessed from the Board class and a combination of polar coordinates and angles will be used to position each object's image on the surface relative to the centre of the board.

With regards to the players, the players will contain a "current" node in their game progression attribute. The player will be drawn by the board where their current node is. This will, at the start of the game default to the home caves.

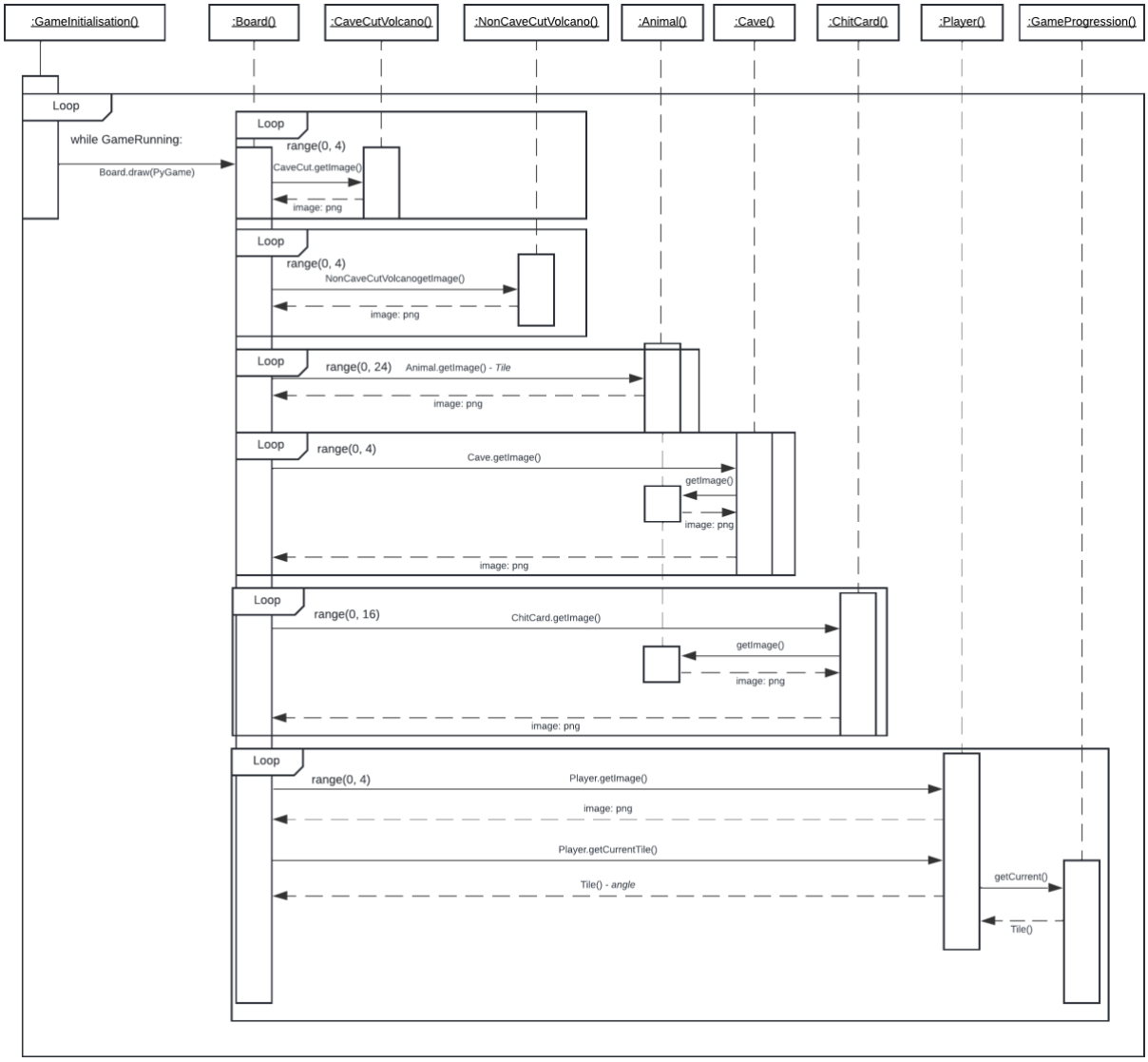
The run method in the game Initialisation class calls on this method in the Board class after initialising the PyGame surface, and the images are drawn on the PyGame window.

Once the images are juxta positioned in an orderly and calculated manner, the pygame surface would depict a board similar to the board described in the Fiery Dragons game.

Sequence Diagrams



The above sequence diagram details the sequences of the Game being initialised by the player. The initialisation of the Game Initialisation, Tile, Animal, Player, GameProgression and Board are all shown above.



The above sequence diagram shows the steps taken by the Board class in order to draw/populate the PyGame window. Th board class will have to retrieve the image of each Cave Cut Volcano, Non-Cave Cut Volcano, each animal in each tile of the Volcanoes, each Cave, Each Chit card and draw them on the correct position on the board based on the predefined polar coordinated. Furthermore, the board will also have to find the current position of each player in order to populate the player on the correct place on the map.

REQ 2

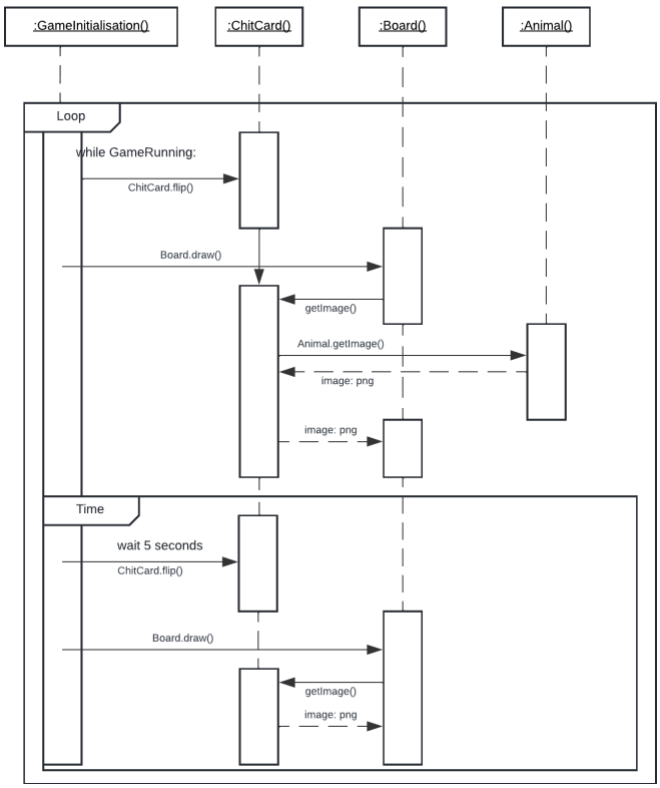
Flipping of dragon (“chit”) cards;

Each Chit Card (where all of them are placed in a ring in the middle of the board) contains an attributed “isFlipped” which is set to either True or False.

The run() method within the Game Instantiation class would create buttons inside the PyGame window surface pertaining to the spots where the ChitCards will be placed by the Board in its draw() method. These buttons will be “clickable” with each click revealing that particular Chit Card for a total of 5 seconds.

To reveal the ChitCard, the “flip()” function inside the ChitCard will be called, which will set the “isFlipped” attribute to dentote that it is flipped. The Board’s draw() function will then be called as usual. The draw() function in the board will have conditional checks before “drawing” the image of each Chit card. If the Chit Card is turned over, it will draw the image of the “animal” within the ChitCard on the place of where the Chit card originally was. If it is not turned over (found through isFlipped() method from ChitCard), then the default “closedImage” attribute of the Chit Card will eb used to draw/visually-represent the chit card on the game board.

A pause of 5 seconds will be used (through the time module) to pause the game to ensure all players get a chance to look at the flipped over chit card. Once the 5 seconds is up, the “flip()” method in the Chitcard will be called again to hid the ChitCard and the board will be redrawn with all the ChitCard using the “closedImage” attribute to represent their position on the board.



the sequence diagram on the left shows the steps taken to “flip” the chit cards on the board. The sequence starts with the chit card being flipped, resulting in Board’s draw() method to draw the image of the animal instead of the default chit card image on the map. The sequence diagram then shows the waiting of 35 seconds, after which the chit card is then flipped back and the image of the animal is replaced with the default chi card image by Board’s draw() function.

REQ 3

Movement of dragon tokens based on their current position as well as the last flipped dragon card;

A few separate mechanisms are used in tandem to move dragon tokens based on their last flipped chit card.

Firstly, the Chitcard's position on the map contains clickable buttons which will be used to reveal the chit card as explained above. Apart from revealing the chit card (turning it over), the button will also serve another purpose. The button will be used to evaluate if the picked chit card allows the player to move forward.

The Animal contained within the chitcard will be accessed from the run() method in the Game Initialisation class (where the above mentioned button conditions will be placed) and the process of "Double Dispatch" will be used to evaluate whether the player will be able to move forward or not (explained more in Design Patterns section as well. To summarise this process, the animal within the chit card will contain an "checkMove" method, with it requiring the animal on the current volcano tile that the player is on as an input. Using this the animal on the Chit Card will call a specific method within the animal on the player's tile (pertaining to the animal on the ChitCard – i.e. if it's a salamander on the chit card, the checkSalamander method on the Animal on the Tile will be called) to check if the player will be allowed to move.

If the Player is allowed to move, then the Game Progression attribute within the player is accessed from the player class in the run() method. Since the Game Progression is a circular doubly linked list, the "current" position of the linked list will be updated to the new position of the player based on the number in the chosen chit card found through getNumber() method (if they move forward 2 positions, then the current attribute will be updated to next node 2 times through the moveForward() method in Game Progression). Since the Game Progression class contains a linked list of all the tiles, the current node updating means that the player's position on the map will also update.

When this is completed, the draw() function in the Board class will be invoked again, where the Bard will attempt to redraw all the attributes om the board. Since the player's current position is updated (which will be used to draw the player on the board), the player will be drawn on a different position (Volcano tile) compared to before, having create the illusion of the player having moved.

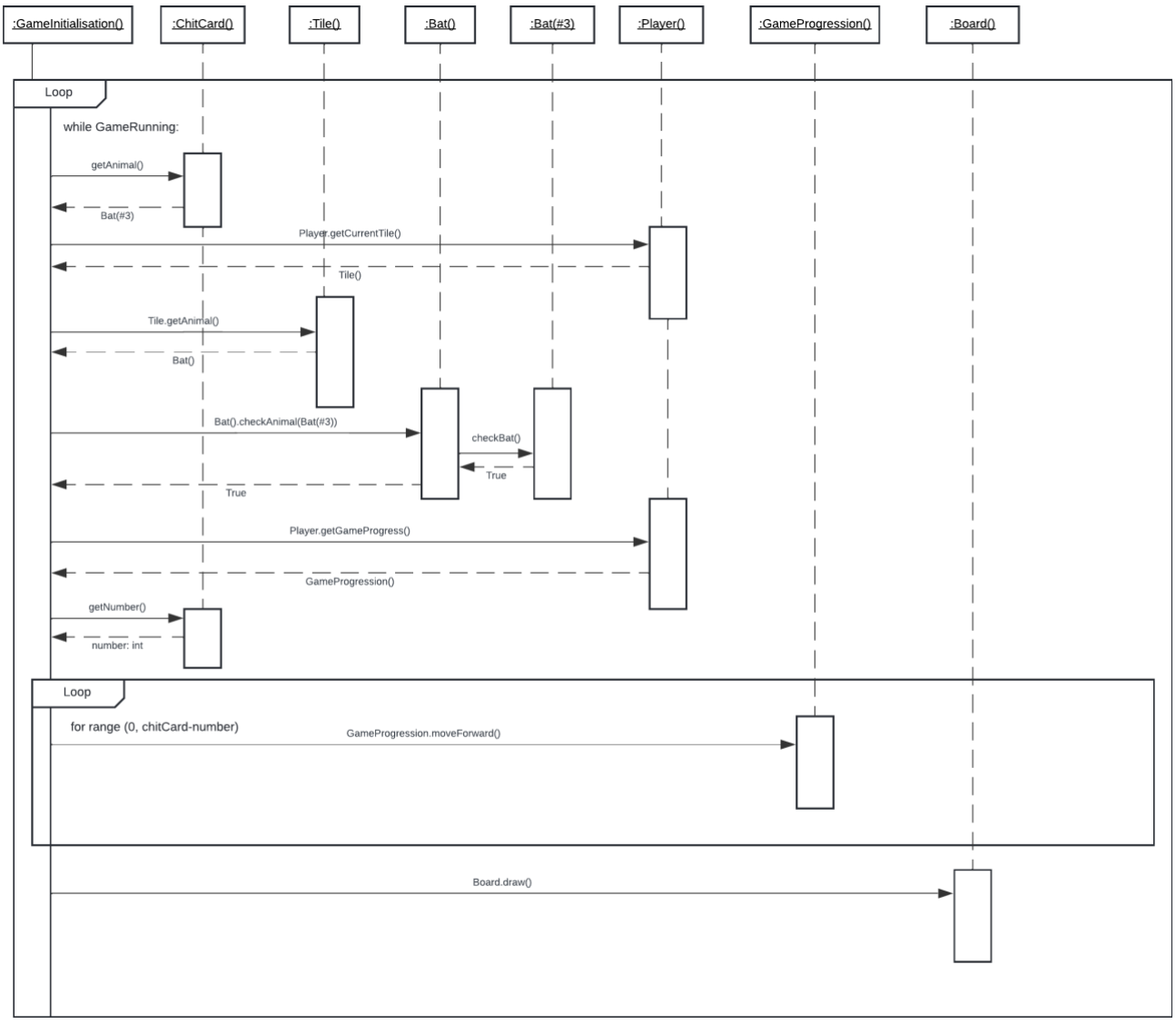
A few checks will be conducted before moving to adhere to the game rules. The Player cannot move into a another player's cave so if a Cave is encountered that is not the player's home cave (found through the player's isHome() method which will check the angle of the player's home cave (stored at initialisation) against the currently encountered cave to determine if the newly entered cave is the player's home), then the current player will "skip" this tile. The run() method will also check if the final tile landed on by the player is occupied by another player. If it is occupied, then the player should not be allowed to move, and the player will move back to their original position before moving forward (by updating

the position of current to the previous tile using .prev() attribute in Game Progression). So, when the final board is drawn, it would be as if the player had never moved.

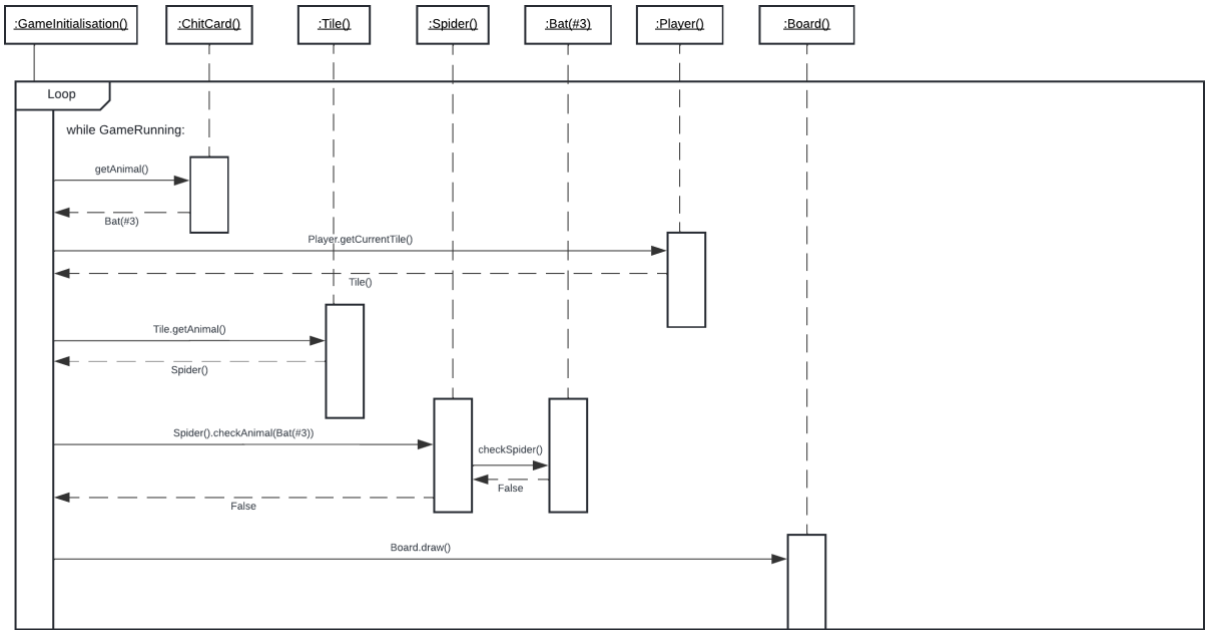
If the Player encounters their home cave (found through the player's isHome() method which will check the angle of the player's home cave (stored at initialisation) against the currently encountered cave to determine if the newly entered cave is the player's home), then the player will move in and their turn will end.

If the player is not allowed to move, the chit card animal – tile animal check fails, then the player will end their turn.

If a player chooses a Pirate dragon, the above double dispatch check will not occur. This is because the player will always have to move backwards if they choose a Pirate Dragon (unless they are still in their Cave, in which case the payer will end their turn again). When moving backwards, the same checks as above will need to take place, and the procedure to move backwards is very similar to the procedure to move forward as described above, but instead of setting the current node to the "next" node/tile, the current node/tile is set to the previous node/tile through the moveBackward() method in the circular doubly linked list Game Progression attribute.



The sequence diagram above shows the sequence of events when a player chooses a “valid” chit card (in this case a Bat #3 whilst standing on a Bat tile)and is allowed to move forward. Events from checking of the validity of the chit card to accessing the Game Progression attribute of the Player to move forward are depicted above.



The sequence diagram above showcases the flow of events when a player is does not choose a “valid” chit cards. The main difference s that “False” is returned when the checkAnimal() method is invoked, which causes the player to remain on thier current tile and lose their turn to the next player.

REQ 4

change of turn to the next player;

The above movement of players, and chit card picking will happen in a loop inside the “run” method of the Game Initialisation class.

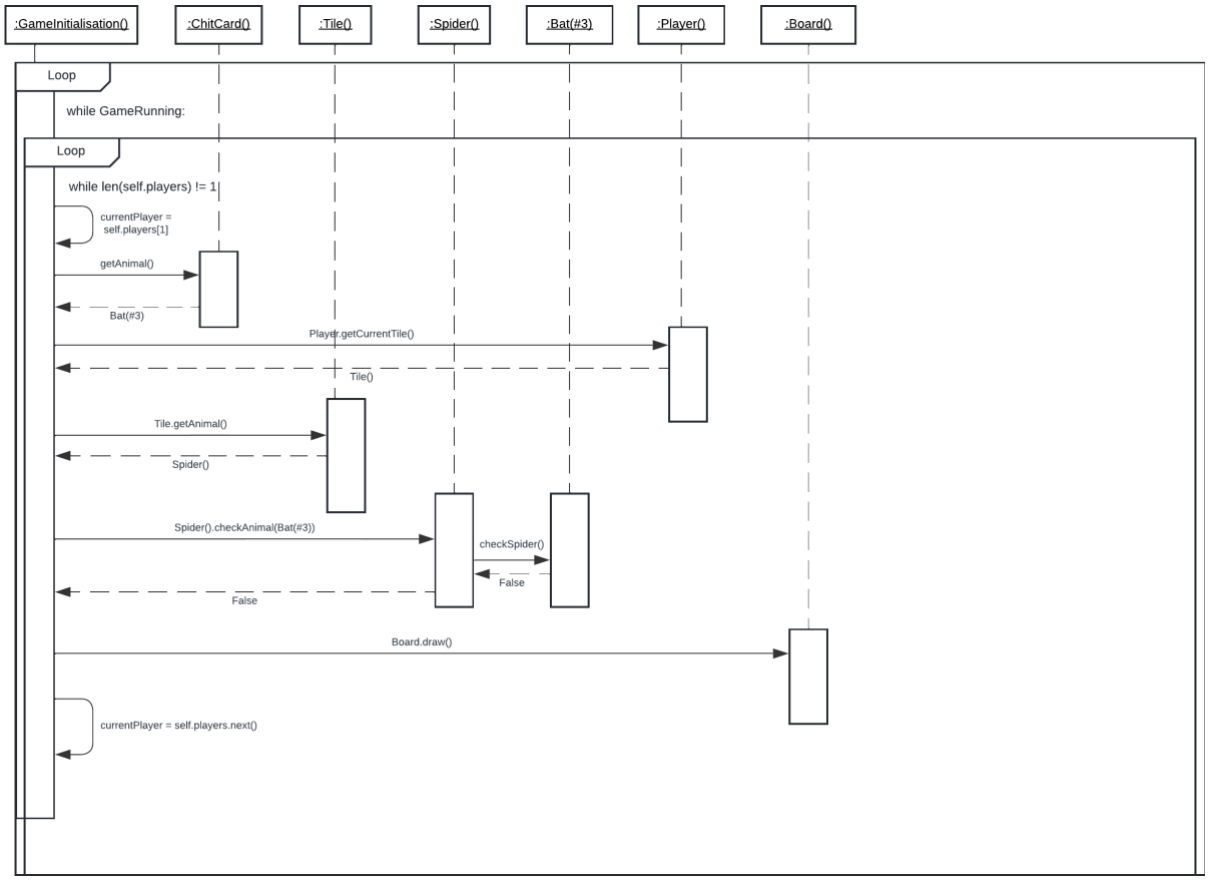
The loop loops through every player indefinitely.

At each player’s turn they will be prompted to choose a chit card (see REQ2). If they choose a “valid” chit card, they will retain their turn and move forward (REQ3). They will then be asked to pick a chit card again and the cycle will continue inside a loop. If they choose an “invalid” chit card or a Pirate Dragon, they will need to exit their turn (after moving backwards in the case of a Pirate Dragon).

The exiting of turn is done though the current iteration of the loop resulting in the “current player” attribute of the loop updating to the next player as found in the self.players list inputted into the Game Initialisation class when it is first initialised. This causes the current player, and their existing attribute to be updated to the attributes of the next player. So the Game Progression attribute will relate to the next plyer instead of the current player so the

moveForward(), moveBackward() methods invoked will cause an update to the “current” node/tile of the next player instead of the previous player in the start of this description.

This process repeats indefinitely, looping through all the players in the list over and over again, until one of the players reaches their home cave tile, when that particular player is removed from the self.players list attribute (REQ 5).



The sequence diagram above showcases the flow of events leading up to a player loosing their turn (through choosing an “invalid chit card in relative to the tile they are currently standing on), leading to the player’s turn being passed up and the value of the currentPlayer updating to a new player.

REQ 5

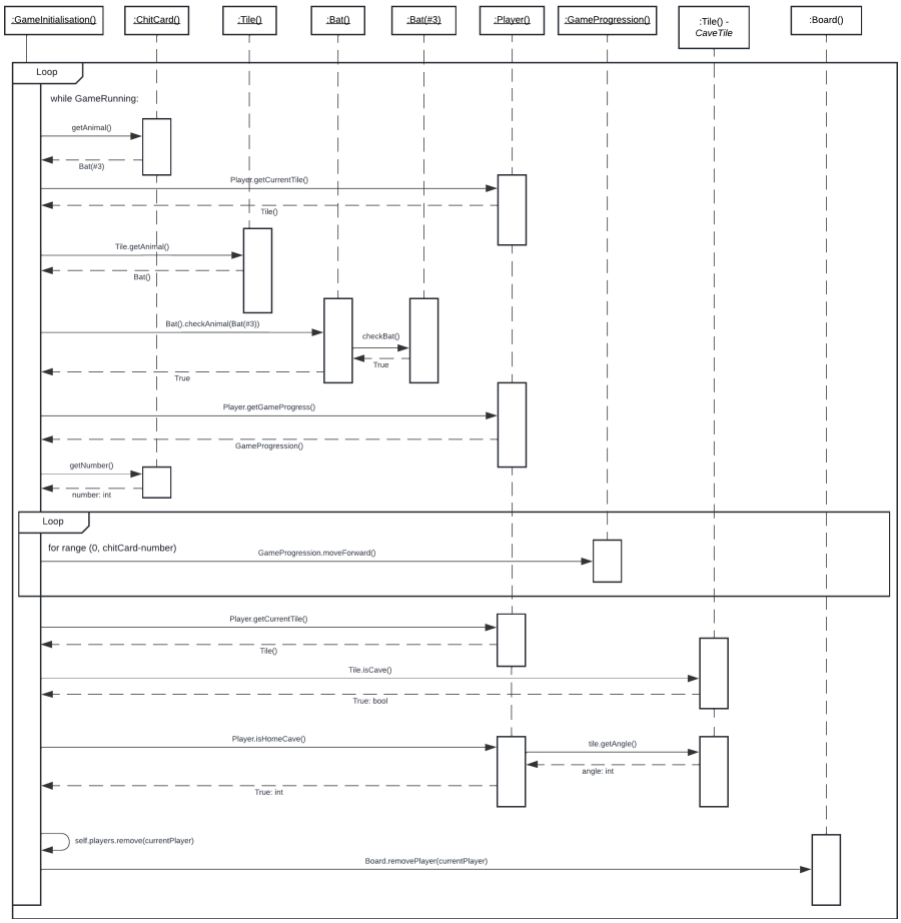
winning the game;

At each player’s turn, if they were to pick a “valid” card to move forward, after each moveForward() invocation of the player’s Game Progression attribute, a check will be initiated to check is the tile the player is on is a Cave. This is done through invoking the tile’s checkCave() method. The current tile the player is on will be retrieved through accesing the getCurrentTile() method in the player’s Game Progression attribute. All Tile objects have a method called checkCave() within them which returns ‘True’ if that particular tile is a “CaveTile” and False if it’s not (i.e. it’s a “VolcanoTile”).

If the tile the player is on is a “CaveTile”, then a check is initiated to check if the tile the player is on is the player’s “Home Cave” (i.e. the tile the player started the game on). This is done through the player’s isHome() method which will check the angle of the player’s home cave (stored at initialisation) against the currently encountered cave to determine if the newly entered cave is the player’s home.

If the encountered Cave is the player’s home cave, and the player does not have any moves left (i.e. their total moves leads them to the cave as their final destination), then that player has won and is removed from the self.players attribute in the Game Initialisation class. So when the run() method in the Game Initialisation class runs, the loop does not count the “winning” player as an allowable player to move their token and the game is allowed to proceed until we have a runner-up/third-place and we reach the point where there is only one active player in the self.players list attribute.

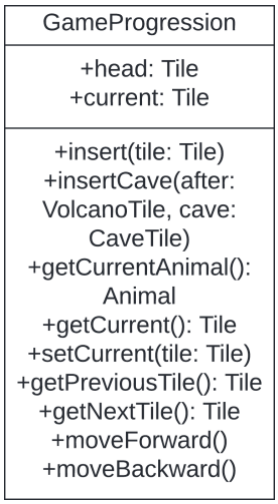
If the Cave is not the player’s “Home Cave”, then the player skips it. If the player “overshoots” the home cave then they would return back to their original spot (through the invocation of the moveBackward() method in the players’ Game Progression attribute) and end their turn (REQ 4).



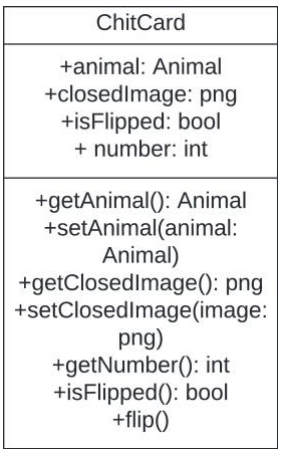
The above sequence diagram showacses the flow of events of a player “winning” the game. The turn of events from checking the validity of a player’s chosen chit card (the player choosing a “valid” chit card), player moving forward, the checking of if the tile the player is standing on is their home cave and the removal of the player from the board after confirming the player’s win is shown above.

DESIGN RATIONALE

CLASSES

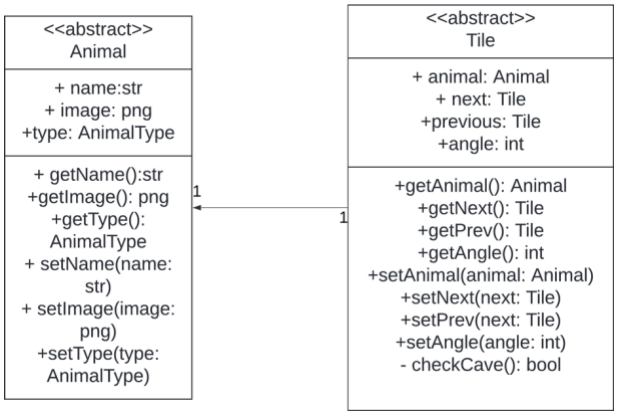


A choice was made to include a “GameProgression” class in the design. The “Game progression class acts as a circular doubly linked list that tracks the progress of the player in the game. This class stores the current, next and previous Tiles to assist the game in moving the player across the gameboard. The reason this sexists as a separate class is to comply with the SRP an OCP design principles in Object-Oriented Programming. The discarded alternative for this design was to put this as methods inside the Player() class. This would give the player more than one ability as the player will also need to manage their own movement across the map in addition to keeping track of their names, images and “Home Caves”. Through creating Game Progression as its own class, dependencies to Tile and Animal can exist in game Progression class but not within Player. This enables for more encapsulation in the design as the Player will remain unaware of the Animal that is on the tile the player is on. The Game Progression class also allows for the design to depict the use of OCP, as through the Game Progression class, it is allowable for other characters to also traverse the board through an extension. Without eth Game Progression class, any new character extensions that needed to traverse the board will need to have all the logic for the circular doubly linked list built into its class (this also goes against the DRY principle) and the logic for the game board will need to be instantiated additionally. However, through the use of the Game Progression class, any new extra characters will just need to have an association with this class, and the new character will be able to traverse the board just as the player currently does without the need for much of the original design needing to change.

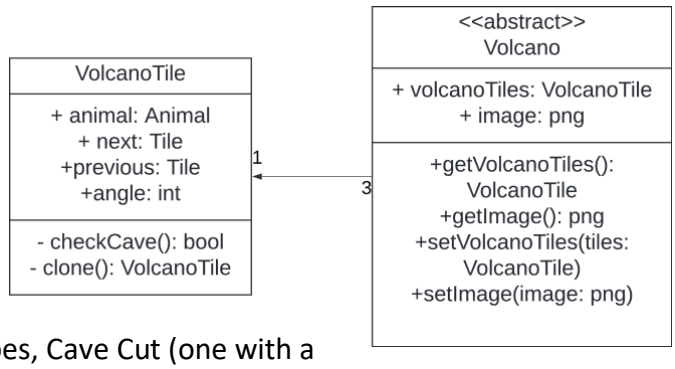


A choice was made to have ChitCard as its own class instead of methods inside the Game initialisation class. This was because of the functions that are required from a “Chit Card”. A “Chit Card” is required to be clickable (flipable). There also exist 16 unique chit cards within the game, each with a unique purpose /difference that sets them apart from another (i.e. the Animal within them). Therefore, having such an object as “methods” would result in 16 different methods within the Game Initialisation class, all with a very similar purpose but with slight differences in the way they operate. Having the above discarded design would result in the design breaking eth DRY principle of OOP. Each chit card also has an image (one flipped over, and one representing the animal inside the card) which is accessed by the Board class whilst populations/drawing the board. Having a chit card as methods inside the Game Initialisation class will create extra associations/dependencies between the Game Initialisation class and the Board class which breaks the SRP principle of Game Initialisation class as it would have more than one responsibility apart from just “running the game”. Therefore, having Chit Card as its own class helps meet OOP principles of encapsulation and SRP as each class will serve a unique purpose.

RELATIONSHIPS

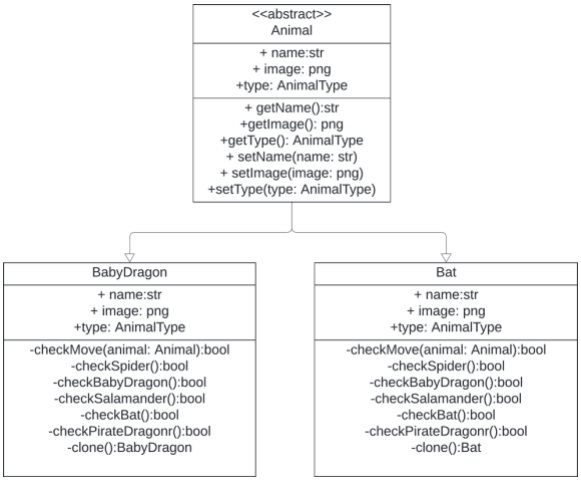


There exists an association (composition) from the Tile abstract class to the Animal abstract class instead of a dependency (aggregation). This is due to the fact that a Tile would not be able to exist without an animal within it as per this design. This approach was chosen after the responsibilities of a Tile was outlined. A tile (as per this design's interpretation) exists to connect the circular board around, be a place where a player has to be at whilst they are trying to make it back to their cave. The second reason is the most important reason between the association here. In order for the player to make it back to their cave, they will have to move forward from the tile. In order for the player to move forward from the tile, they will have to choose a chit card that contains the same *Animal* as the tile. So one of the main responsibilities of Tile class is to guide the player in picking the right chit card by showcasing the animal on the tile. The animal on this tile is therefore one of the main reasons for the tile to exist. There therefore cannot be a tile without an Animal on it as the player will not be able to move forward from the tile, hindering the game's main objective of allowing the player to return back to their cave. This is the reason for an composition over and aggregation as the Tile cannot exist without an animal as per the game's main objective.

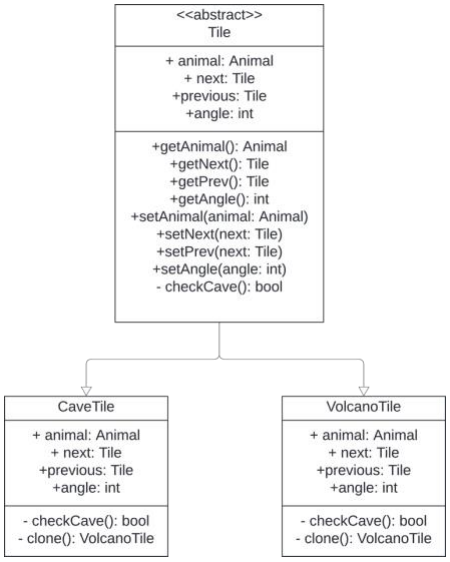


There also exists an association (composition) from the Volcano abstract class to the VolcanoTile (from Tile) class instead of a dependency (aggregation). A Volcano is defined as the collective of 3 volcano tiles, as per the original board game's assembly directives. There exist 2 types of Volcanoes, Cave Cut (one with a cave attached to it) and non-cave cut (one without a cave attached to it). As defined in the game rules, there exists 24 board positions which the player will have to cross in order to return back to their original cave. The Volcano card exists just as a way to connect the board and draw the board. The Volcano card does not contain within itself any logic that will allow the player to progress in the game as the Volcano does not have "Animals" within them which the player can leverage to move forward in the game. Therefore, the Volcano needs to be made up of Tiles (which contain Animals) in order to allow the Player to complete their primary game objective of traversing around the board. The volcano can therefore never exist without a Tile as a Volcano without a tile serves no purpose within the game's primary objective context.

INHERITANCE

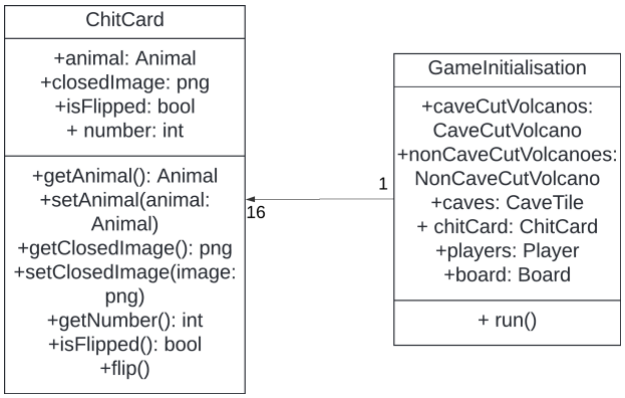


There exist an inheritance between the Baby Dragon, Bat and Animal (among other Animals as well) in the chosen design. This is because Baby Dragon and Bat both serve the same purpose within the context of the game (advance the player), but they possess unique qualities that differentiates them from each other. Therefore, it makes sense to create an abstract class Animal to extract the similar qualities about each Animal and place them in one class. Each unique animal contains the attributes Name, Image and Type, with getter and setter methods for each attribute. By combining these attributes into one abstract class, we follow the DRY principle of Object-oriented design as these methods are inherited from the Abstract Animal class instead of the specific Animal classes. Furthermore, there exists many associations to Animal that are common to all Animal classes. For example, Tile has an association with “Animal” class. A tile could possess any of the Bat, Spider, Salamander or Baby Dragon animals. It will complicate the design heavily if Tile has an association with each of these Animals, especially as one Tile will only contain one Animal. This would result in unnecessary associations existing between a Tile and the other 3 unused Animal classes breaking rules of Encapsulation and Object-Oriented design. However, having an Animal class as a parent class of these specific Animal classes, these animal classes could exist with limited Associations, better following the rules of object-oriented design.



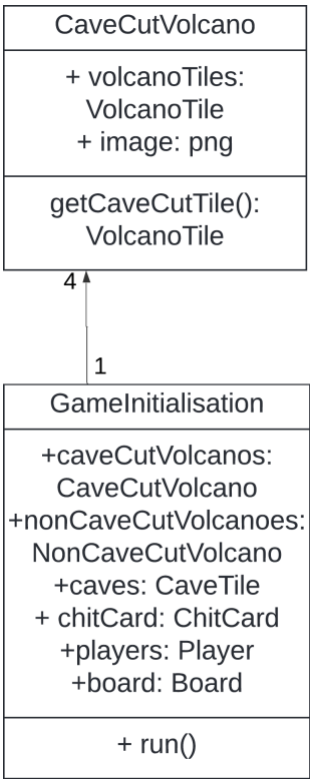
Here exists an inheritance exists between Cave Tile, Volcano tile and Tile as both Cabe Tile, and Volcano Tile both have common attributes such as animal, angle, next tile and previous Tile. Thereby the OOP principle of DRY can be achieved by combining this method into a parent class which will pass on these methods and attributes to the two classes through inheritance. Furthermore, the Game Progression class will need to reference both Cave Tile and Volcano tile when creating a linked list of allowable moves. Without inheritance, there would be extra associations between Cave Tile and Volcano tile, which breaks OOP principles. Furthermore, the Game progression class now with inheritance remains unaware which type of tile is added to the list, helping achieve OOP principle of Encapsulation. Each Tile however has a unique return to the isCave() method which would be called by the Game Initialisation class. Since the Game initialisation class only knows that the object is a Tile, this exists here to provide confirmation that the Game initialisation class needs. Thereby these two classes reman sperate from each other and the abstract class/method in the inheritance can be used by the rest of the game in game logic helping achieve encapsulation.

CARDINALITIES



The exists a 1..16 cardinality between Game Initialisation and Chit Card because for the game to start, there needs to exist 16 unique chit cards. Each chit card represents a unique animal and holds a unique position within the board. Therefore, there needs to exists 16 different chit cards for the game to properly initiate as 1 chit card cannot substitute for another and still maintain the above described characteristics. Furthermore each chit card object will be accessed by the game Initialisation class (to draw it or when the player clicks on it) at one point of the game. Therefore this cardinality needs to exist as per the game's written functionalities.

There exists a 1..4 cardinality between Game Initialisation and Cave Cut Volcano. This is because each Cave Cut Volcano that is input into the Game Initialisation class contains unique Tiles and animals within them. There needs to exist 4 Cave Cut Volcano on the Game board as per the game requirements in order to complete a full board. Since each Cave Cut volcano is unique due to containing unique tiles/animals), the cardinality of 4 CaveCut volcanoes need to exist for the Game to Initialise with a full board. There cannot be a valid game with more that 4 Cave Cut Volcanoes or less than 4 Cave Cut Volcanoes.



DESIGN PATTERNS

FACTORY CLASSES

Each animal class has a Factory class that produces these animals. This is because most Animals can be one of four types: Cave Animal, Tile Animal, Chit Card Animal with Quantity 1, Chit Card Animal with Quantity 2 and Chit Card Animal with Quantity 3. Each of these unique aimaltypes have different instantiation logic as each type has different images representing them and different AnimalType enums. The use of this design patterns allows for encapsulation in the design as the logic for creating each Animal is hidden rom the rest of the classes and the Animal itself. This pattern obfuscates the need from eth animal dealing with the logic of figuring out the right image to use for the particular type of animal needed. This design pattern also allows for code reusability and maintainability as this factor class can be used anywhere in teg ae to create any type of Animal objects, reducing the need for the same specialised code in various different classes and following the DRY principles OOP. Furthermore, this design pattern also allows or increased flexibility as any different type of Animal objects can be created through this Factory method, including new types of animals that may be introduced in the future. If new Animal types are introduced, the process of creating the Animal object remains the same, saving design and development time.

PROTOTYPE

The prototype design patterns is added to game features where there needs to be an exact clone/copy of the item to be created. This method from an interface uses the objects existing attributes to create an exact copy of the object. This design patterns was particularly useful when adding the tiles to Game Progressions of each player in the Game Initialisation class. This is because each player has their own unique Game Progression attribute and therefore tiles need to be added to each Game progression attribute in each player. The same tile cannot be added to two different Game Progressions as the structure of the linked list would get spoiled. Therefore, there needs to be a way to create an exact copy of the tile to be added to each players Game Progression attribute. The Prototype design pattern was used in this instance as the clone method was used to create an exact copy of the tile. This allowed for encapsulation as the Tile object itself is responsible for creating its own clone so private attributes are not revealed to the Game Initialisation class as only the newly created tile object is returned back to the Game Initialisation class. This pattern can also be used in the future to create new objects with increased efficiency.

DOUBLE DISPATCH

The double dispatch design pattern is used to encapsulate the checking of tile Animals versus Chit card Animals in this design. Trough using Double dispatch, OOP principle of encapsulation is followed as the Animal Object checking the secondary animal Object remains unaware of the secondary Animal Object's attribute and details. This allows for increased code security as each Animal object in this scenario uses very little information about the other Animal Object to process game logic and arrive at a conclusion. Theis also relives the use of complicated type checking between animals allowing for much more

readable and easier to understand code. There did exist one drawback in that this code is not very extensible as any new Animal added to the game will need to have many additional checks to be compatible with the rest f the game. However, it was concluded that the benefits of using double dispatch outweighs its cons in this design.

TECH-BASED IMPLEMENTATION

The game functionality implemented in this code are:

- set up the initial game board (including randomised positioning for dragon cards);
- movement of dragon tokens based on their current position as well as the last flipped dragon card;

The player will continually loop through in this game as the game will not conclude (winning the game is a different requirement)