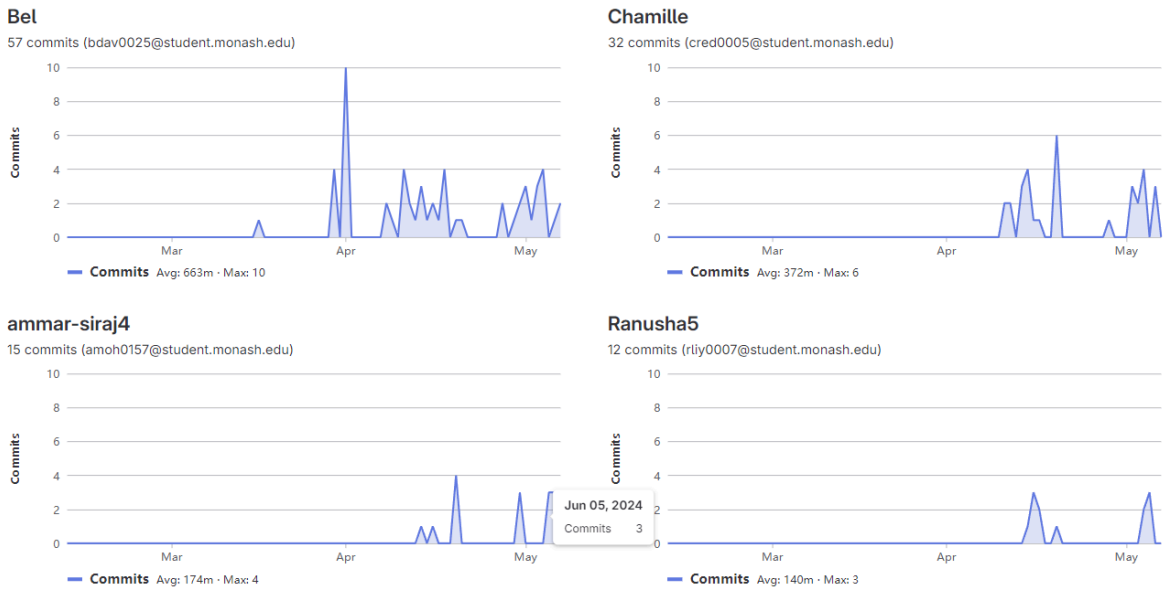
FIT3077 Sprint 4 Deliverables

Group Members

Chamille Reddiar
Belvinjeet Kaur
Mohamed Ammar Ahamed Siraj Mohamed
Ranusha Liyanage

Contributor Analytics



Description of executable

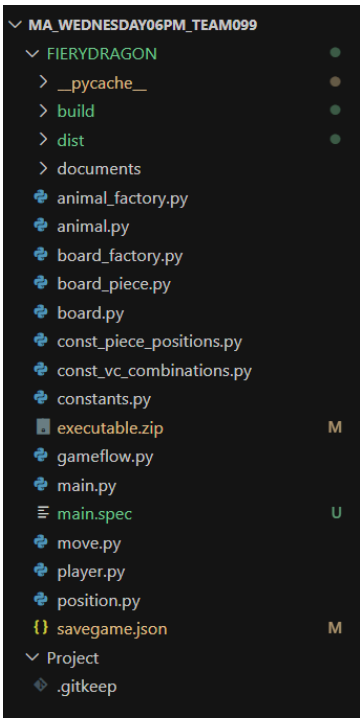
To distribute the Python game application with a Pygame GUI as a standalone executable, we used PyInstaller. Here are the steps and commands involved in creating the executable:

1. Install PyInstaller:
   ```
   pip install pyinstaller
   ```
   We used pip to install the PyInstaller in the Python environment.

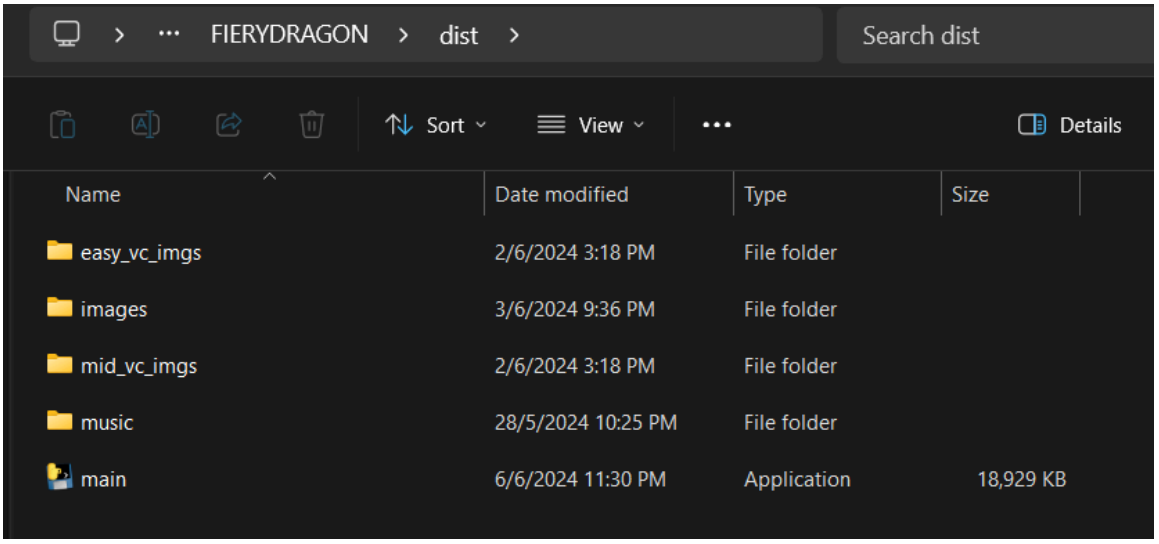
2. Code Implementation:
   Our implementation for the application was using a Python script where our main file was ‘main.py’. The script includes the PyGame GUI logic.

3. Creating the executable:
   We used the PyInstaller to bundle the script into a single executable file by having the ‘--onefile’ option that ensures that all dependencies are packaged into a single executable.
   ```
   pyinstaller main.py --onefile
   ```

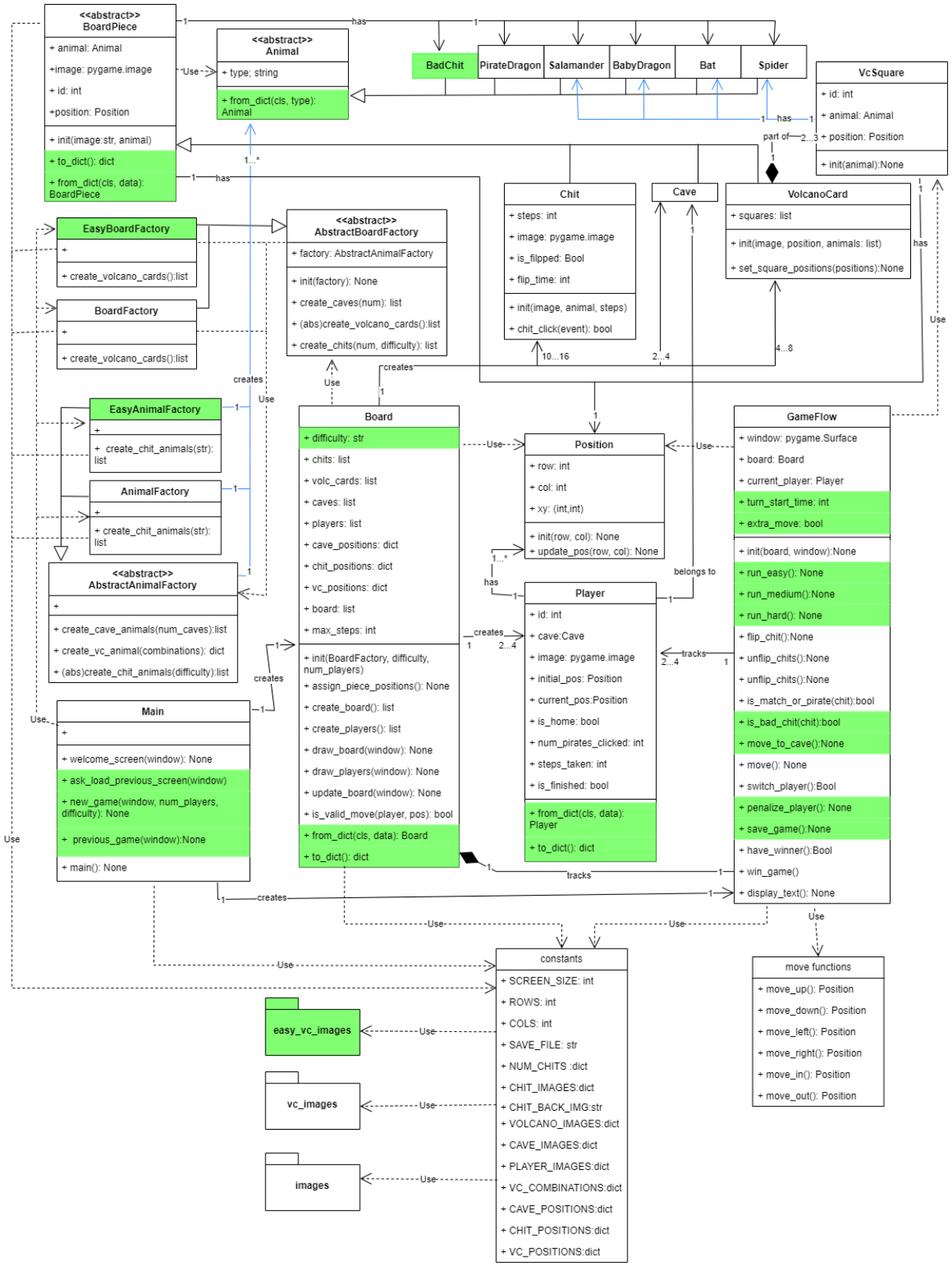
4. Output files:
   The PyInstaller will generate several files and directories which consist of:
   - ‘dist/main.exe’ : The standalone executable file
   - ‘build/’ : A directory containing temporary files used during the building process
   - ‘main.spec’: A spec file that describes how to build the executable which is customizable for advanced configurations



5. Execution of game:
   We made sure the images folder where all the relevant images are within the ‘dist’ folder in order for a proper display of the images used within the application.



Class diagram:



The green highlighted areas are the sprint 4 additions.

Chit Extension:

Bad Chit was extended from Animal Class since it shares the same properties. In GameFlow, the Bad Chit logic was implemented within is_bad_chit and move to cave methods, with some modifications to flip chit to include the new logic.

Self Extensions:

1. **Difficulty:**

Animal and board factory interface was removed since python doesn't need interfaces the way java does because python can support multiple inheritance; any interface defined in python would just be another abstract class. Easy board and Animal factories was added since the easy board has a different number of cards as well as images. There are no hard board factories since it uses the same number of cards as medium the board difficulty.

2. **Timer:**

Since each difficulty has a different implementation of the timer, there are different run methods for each difficulty in the GameFlow class, as well as the penalize_player method to move a player backwards if they exceed the time limit.

Save and Load:

Save occurs when the player exits the app while the game is running.
Since json cannot serialise classes, the BoardPiece and Animal abstract classes have from_dict, to_dict methods to convert relevant information upon saving the game. This doesn't occur if the game has concluded (a player wins). The next time the app is opened, players have the option to reload the previous game or start a new game which will delete the old save file. This is implemented in main.py within the ask_load_previous, new_game and previous_game methods.

Others:

A VcSquare doesn't exist independently without its volcano card, so it has a composition relation with VolcanoCard. In the current application, a volcano card can have 2 or 3 squares per card depending on the difficulty, which is why the cardinality is as shown.
Game flow and board have a composite relation because gameflow is heavily reliant on board and cannot function without it as board holds all information about each piece

Human Values of Extensions:

Based on the human values provided for the Fiery Dragons Game, these are the values implemented in the extensions below:

Bad Chit Extension

Creativity: The design and placement of faulty chits can inspire innovative and clever play.
Pleasure: Adds a sense of surprise and excitement to the game, making it more engaging.
Intelligent: Players must utilise their brains to successfully handle the risks posed by faulty chits.

Different Board Difficulties

Easy Difficulty:

Helpful: Offers a more accessible and beginner-friendly version of the game, which serves as a lesson for new players.
Healthy: Prevents new players from being overwhelmed, encouraging a healthy approach to learning the game.
Pleasure: Maintains enjoyment by providing a less difficult yet equally enjoyable gameplay experience.

Medium difficulty:

Intelligent: Promotes strategic thinking and planning when players tackle moderate hurdles.
Pleasure: Maintains a mix between challenge and fun, keeping players interested.
Choosing Own Goals: Players can select this level based on their existing skill level and learning goals.

Hard Difficulty:

Intelligent: Requires advanced strategic thinking and quick decision-making, which puts players to the test.
Pleasure: Offers a high level of challenge, which may be extremely rewarding for seasoned players.
Social acknowledgment: Success in more difficult levels might lead to larger scores or achievements, resulting in social acknowledgment.

Timer Variations

Easy Timer:
Healthy: Reduces stress by eliminating time limits, allowing participants to play at their own speed.
Helpful: Excellent for beginners who require more time to understand and make judgements.
Pleasure: Provides a relaxing and delightful gaming experience with no time constraints.

Medium Timer:

Intelligent: Encourages players to think swiftly and tactically within a reasonable time frame.
Healthy: Strikes a balance between challenge and time constraint, fostering cognitive development while minimising stress.
Pleasure: Maintains the game's excitement while limiting the amount of time pressure.

Hard Timer:

Intelligent: Requires rapid thinking and effective decision-making under tight deadlines.
Challenge: Increases the game's difficulty, making it more exciting and interesting for experienced players.
Healthy: Promotes resilience and rapid thinking, which is good for cognitive health.

Saving and Loading Data

Helpful: Allows gamers to save their progress and resume later, matching their schedules and preferences.
Healthy: Reduces the demand for long, continuous play sessions, encouraging better gaming behaviours.
Pleasure: Increases overall satisfaction by allowing gamers to pause and continue their gameplay without losing progress.
Setting and achieving goals at one's own speed allows players to have a more personalised gaming experience.

In conclusion, these expansions incorporate human qualities such as creativity, social recognition, intelligence, helpfulness, healthiness, pleasure, and the capacity to set and achieve personal goals, improving the Fiery Dragons game experience.

Reflection:

Chamille's:
The extension I helped implement was the game difficulty extension. The goal was to create boards of different levels of challenge by implementing new board and animal factories that create all the necessary components required for the different game modes.

In terms of extensibility, having already implemented most of the logic in the board and animal factories in sprint 3, it wasn't difficult to add on the Easy mode factories. In sprint 3, there was an interface, which the concrete factories extended; I decided to replace the interfaces with abstract classes due to the nature of interfaces in python being just another abstract class and also because there were a lot of similar codes. After implementing the abstract classes, all that was left to do was implement the abstract methods for the extensions. When implementing the factories for the easy board, which has less board pieces, I realised that in sprint 3, I hardcoded a lot of the code regarding the number of volcano cards and the number of squares each volcano card has. Another problem I realised when implementing the hard board was that the number of normal and pirate chits were also hardcoded, which made it tedious to implement the hard board factories, so after removing the hard coded aspects and making the code more flexible, the result was a reusable code with less repetitions among the different implementations.

Looking back on sprint 3, I had recommended the use of the gameflow class to combine everyone's implementation. Since then, most of the new logic in sprint 4 was added directly to the gameflow class which has made the file large and messy. If I could start over sprint 3, I would like to implement the use of the state and strategy design patterns. Since the different difficulties call its own run function, perhaps the use of an abstract gameflow class with a concrete class for different game modes could be used; with the main class acting as the navigator for which strategy to use, based on the player's input. Also, the use of states could allow us to easily add more extensions; In our initial discussions, we toyed with the idea of two players entering a 'duel' state if a player lands on an occupied volcano card square but decided to forego due to limited time constraints. Other than that, since python is not as strict as java when it comes to access modifiers, most methods and attributes were left public, so I would have made more effort to increase encapsulation and abstraction, ensuring that internal states are protected and only necessary methods are exposed.

In conclusion, the game difficulty extension project was a valuable learning experience in software design and extensibility. By addressing the issues of hardcoded values and implementing abstract classes, we made the codebase more flexible and reusable. However, reflecting on the process has highlighted the need for better design patterns, such as State and Strategy, to manage complexity and improve maintainability. Increasing encapsulation and abstraction would also have contributed to a more robust design. Moving forward, these insights will guide future projects to ensure cleaner, more efficient, and more scalable code. Through continuous reflection and adaptation, we can enhance our practices and produce higher-quality software.

Bel's:
Incorporating the timer functionality for different difficulty levels in Sprint 4 was a little challenging yet a great experience. Since Sprint 3 had only one level without a timer, adding this new feature required significant modifications, additions and enhancements to the existing codebase.

As for the easy level, it does not contain a timer which requires minimal changes. The primary challenge was to ensure that the new timer related logic did not interfere with the existing functionality for this level. Sprint 3's division of the game's logic and difficulty levels contributed to the simple level's preservation. For the medium level, state management had to be included in order to track the timer and reset it after each successful click in order to implement a resettable timer. This is because it requires integrating the timer with the current game logic without introducing bugs or performance difficulties, this was a relatively complex task. This integration was made easier by Sprint 3's modular design, although handling the timer's state changes introduced more complexity. The game's turn-based structure required exact control over the timing mechanism due to the hard level's fixed timer every turn. This presented a difficulty since it needed to be made sure that the timer gave the player consistent input and interacted with the turn logic in a smooth manner. It took significant modifications to the Sprint 3 design—which was not created with timers in mind—in order to make room for this fixed timer.

If I could revisit Sprint 3 and start over I would first of all improve the state management by implementing a more advanced state management system from the beginning to handle different game states such as turn transitions and timer resets more effectively. I would also design the game architecture with future timer functionality in mind that would involve creating flexible integration points for adding timers without deep changes to the core game logic.

Specifically, the Strategy Pattern was a useful design pattern for controlling the different behaviours of the easy, mid, and hard difficulty levels. The game logic was able to stay adaptable and expandable because of this pattern. The behaviour of each difficulty level, along with the timer parameters, could be composed into a plan that the main game engine might carry out. This method not only made it easier to add new features, but it also made sure that every addition followed the same structure, which decreased the possibility of errors or inconsistencies being introduced.

Ammar's:
The extension I incorporated was the additional dragon/chit card. Our team decided to go with the first option: a "bad" chit when clicked, the player needs to go backwards on the board until it finds a free cave.

In Sprint 3, the team incorporated the factory method design pattern, which improves the flexibility software quality of our code. Because of this, creating a new animal for the new chit was relatively easy, it was just a matter of adding a new Animal subclass and then creating an instance of it in the AnimalFactory class. And due to the implementation of the create_chits() method in the BoardFactory class, which uses the AnimalFactory, the bad chit

instance was created, added to the board and displayed on the screen without needing to modify anything other than the new number of total chits (for different difficulty levels) and the adding a new position to draw on the screen in the constants file.

However, the logic for the token to move backwards is more complicated. This is because in Sprint 3, we decided for the move functions to communicate with the Position class for the tokens to know their positions and go around the board, rather than collaborating with other board pieces, particularly Cave and VolcanoCard. The game uses the display dimensions (a grid system) to find the positions of the token and caves and accordingly compares them to find the correct cave to move to.

Another major hurdle was to figure out how to know if a cave is free or not. In Sprint 3, a cave could only be occupied by the one particular player that starts on it, so in our team's implementation, the cave does not know if it is occupied or not and the player only knows if it currently occupies its starting cave. For this requirement, I decided to assign a responsibility to each player to know if their start cave is currently occupied or not, and used this to check if a cave is free for another player to move into.

If I could go back to Sprint 3, I would ditch the grid system that is currently being used and instead have the tokens communicate with other board pieces to know its position and move on the board. Moreover, I would assign responsibilities to the Cave class, particularly if it is being occupied by a token and if that token is the 'start' animal. Unfortunately, implementing both these things in Sprint 4 would have required a large overhaul of the codebase, with too great a cost of change.

Ranusha's:
The extension "saving and loading" the game functionality was very challenging. In sprint 3, there was no option for the player to load a previous game and continue playing which is what this extension accomplishes.

The logic of the save and load extension, the code communicates with each of the board piece classes. Each of the board pieces, the volcano cards, the caves and the chits as well as the players have a method to save their data into a dictionary which can be saved into the json file. The main hurdle to overcome with this extension is finding a way to save all the game data into a format that can be dumped into a json file. To overcome this our team decided to implement the forming of the dictionary into each class rather than trying to compile all the data in the gameflow class. By assigning the compiling of data to each individual class we were able retrieve all the data of the board pieces in the gameflow class and save all in one json file. This reduced the number of json files required to save the game which makes it simpler to load the saved game. Another major hurdle our team faced was loading the data from the json file back into the code. This proved difficult as we had not taken into account the fact that the data must be stored when designing the classes and their data types.

For sprint 4 our team incorporated the factory method design pattern and the memento design pattern. Since each of the classes handles storing and loading their own data into and front the data files it shows the memento design pattern. This implementation improves

the flexibility of the software as it does not rely on a single class or method to store the data into the file. Due to this implementation the save file can be done without having to know the implementation of the data and how it is stored in the individual classes themselves.

If I had a chance to revisit sprint 3 and do it again, i would first clearly declare the game states and have separate run functions depending on which state the game is in. To elaborate, by this I mean for the running method of the game to be more reliant on the current state. I would assign this responsibility to the gameflow class. I would also lay a better framework for the extensions required later. I would design the board object classes with the functionality of storing all that data in mind, so that when it is required the data is in a format that can be easily stored and loaded from. This way it would have reduced the difficulty of sprint 4.