Introduction

Our project, named PopPuyo, is a strategic tile-matching puzzle game. It challenges the player to score as many points as possible by clearing the board populated by colorful Puyos, while avoiding filling the third column, counting from left to right. Once this column is completely filled, the game ends.

Gameplay Overview

Puyos are small balls that fall in pairs from the top of the screen. They come in five distinct colors: red, blue, green, yellow and purple, and can be moved left or right or rotated clockwise or counterclockwise as they fall. The goal is to strategically position these pairs to form groups of four or more adjacent Puyos of the same color, making the groups pop and disappear from the board. When this happens, players are awarded points.

Puyos can connect horizontally or vertically but never diagonally. Once a group of Puyos pops, any Puyos above them fall, potentially creating new groups and causing chain reactions. With each pop, the amount of points added to the score multiplies.

Core Mechanics and Features

- **State Transitions:** The game contains multiple states, including MENU, CREDITS, PLAYING and EXIT. Transitions between states are triggered based on user input and gameplay.
- **Menu Screen:** The game initiates with a menu where players can navigate using the up and down arrow keys, and select options using the Enter key, allowing them to choose which screen or functionality they want to access.
- **Credits Screen:** Selecting the credits option from the menu displays information about the game's creators and contributors. To return to the menu, players can press the 'm' key on the keyboard.
- **Exit Option:** The Exit option in the menu allows players to close the game and exit to the system.
- **Single-Player Gameplay:** The game is designed for single-player only. where that one player is the one interacting with the game arena and controlling the falling puyos.
- **Game Controls:** Players control the falling puyos using the left, right and down arrow keys. They can rotate puyos using the 'x' (clockwise) and 'z' (counterclockwise) keys. This precise control allows players to maneuver puyos and strategize their placements on the game board.

- **Chains and Scoring:** The game includes a chain reaction mechanic. When puyos of the same color connect in chains of four or more, they pop, clearing space on the board and awarding points. The larger the chain, the more points for the player, with each subsequent pop giving exponentially more points.
- **Strategy:** Thinking ahead is key to success. Players must anticipate chain reactions and plan moves beforehand to create bigger chains and maximize points.
- **Game Stages:** As players progress and accumulate points, they reach new stages. Each new stage increases the falling speed of the Puyos, making the game more challenging over time.
- **Game Ending Condition:** The game ends specifically when either the second or third column are filled. Players must maximize their score without neglecting the risk of the board filling up too quickly.
- **Game Reset:** Once the game ends, the board is completely reset, and the game returns to the menu to allow the player to either start a new session or exit.
- **Visuals:** All game images are rendered and displayed on the screen using sprites that represent all game elements.
- **Error Handling:** The game is designed to handle potential errors ensuring that it doesn't crash or freeze if something goes wrong.

Design Choices and Patterns

Problem 1:

As we had to work on a large set of tasks, it became crucial to separate responsibilities to complete the game in a timely manner. One of us was responsible for the visuals, while the other focused on the game mechanics. However, as the project grew in complexity, many classes began to accumulate too many responsibilities, leading to violations of the SOLID principles. These violations would be impossible to deal with without introducing additional classes.

Solution:

To address these challenges, we decided to adopt the Model-View-Controller (MVC) design pattern. This pattern is mostly used to manage complex systems by dividing the application into three distinct components:

- Model: Represents the logic of the application.
- View: Handles the user interface and visual representation.
- Controller: Manages user input and coordinates interactions between the Model and View.

Consequences:

By applying the MVC pattern, we successfully separated the responsibilities not only between the developers but also between classes, allowing both of us to focus on our specific tasks without interfering with each other's work. This division also made it easier to add new features in the future. We could work on individual components without needing to understand the other's choices in their code. By avoiding large, monolithic files, the code became more maintainable and easier to test. The MVC pattern provided the clean and structured architecture necessary for our collaboration while also ensuring that the project could evolve without becoming unmanageable.

Problem 2:

In the beginning of our planning for the project, we recognized that there should only be one instance of the game and a single terminal when running the Application class. Being able to unknowingly create multiple instances of the game or terminal could lead to inconsistent behavior. Both the game and the terminal setup are computationally expensive, so ensuring only one instance would help manage resources better. Also, given that the Game class coordinates various components, it was crucial to manage these resources reliably.

Solution:

To address this issue, we decided to implement the Singleton pattern for both the Game and GameScreen classes. This ensures that only one instance of each class exists during the application's lifetime, providing a global access point to these instances instead of relying on public constructors. The Singleton pattern restricts the instantiation of these classes, allowing only a single instance to be created and reused.

Consequences:

By applying the Singleton pattern, the Game instance became a global access point for all essential game functionality, eliminating the need for static methods, which can make it harder to test and maintain the class. This approach not only ensured that resources were used efficiently but also prevented the creation of redundant instances. The reuse of the existing Game and GameScreen instances during resets allowed us to avoid expensive reinitializations and unnecessary thread creation. This choice also simplified debugging by consolidating logic into a single, centralized instance. All resources were directed to this single instance, ensuring nothing was left unused or wasted. Overall, the Singleton pattern helped achieve better performance and easier maintenance.

Problem 3:

When we began implementing the menu system, we noticed conditional statements were piling up in the model classes as were also the responsibilities of said classes. This created a crowded almost unreadable design. Additionally, we needed a more flexible way to transition between different states of the game. The game needed to be able to seamlessly

transition from one state to another, with each state being able to influence the game's flow. Continuing with the current approach would make it impossible to create an elegant code which is also easy to manage.

Solution:

To address these issues, we decided to implement the State pattern, a behavioral design pattern that allows an object—specifically the Game class—to alter its behavior as its internal state changes. By encapsulating each state in its own class, we enabled the game to transition between these states more smoothly, with each state handling its own behavior and state transitions.

Consequences:

Implementing the State pattern removed the need for large, complex conditional statements that would have made the code harder to understand. With each state encapsulating its own behavior, we get a cleaner, more modular design. This separation of responsibilities improved code readability and maintainability. This pattern also made our MENU and CREDITS states more reusable, allowing us to easily implement them in other projects if needed. Testing became much easier, as each state could be tested independently, and any issues related to specific states could be identified and solved without affecting the rest of the game logic.

Problem 4:

The various states of the game had different behaviors, especially in methods like handling user input and rendering the screen. These methods had similar signatures across states, but their implementations were different. Managing these behaviors in a single class would lead to cluttered and hard-to-maintain code, especially as the number of states grew.

Solution:

To solve this, we implemented the Strategy pattern. This pattern allowed us to define an interface that contained all common methods, like processKey() and draw(), but with slightly different implementations for each state. Rather than hard-coding specific behaviors directly into the Game class, the game selects the appropriate strategy based on its current state, ensuring that the behavior of each state is correctly executed.

Consequences:

By adopting the Strategy pattern, the code in the client class, in this case Game, became cleaner. Instead of managing all behaviors in a single, monolithic conditional statement, we could easily add new strategies or modify existing ones without affecting other parts of the system. This approach also made the system more flexible and adaptable at runtime. The game could switch between different strategies (states) seamlessly, without requiring the developer to manage the specifics of each state while coding. This resulted in clearer, less error-prone code. While the number of classes increased due to the need to define multiple strategy implementations, the additional classes were lightweight and reduced overall

complexity. The behavior was now divided into distinct, reusable pieces, leading to a much more maintainable codebase.

Problem 5:

During the implementation of sprites, we noticed significant duplication of code across multiple Viewer classes. Each time a sprite was required, it had to be created and loaded manually in every individual class. This repetitive process made it easier for inconsistencies to arise as different classes might handle sprite creation differently. Additionally, this approach made the code more prone to mistakes, as each class independently managed the creation of sprites.

Solution:

To address these issues, we implemented a lightweight version of the Factory Method pattern via the SpriteLoader class. The Factory Method is a design pattern that encapsulates the creation logic of objects, ensuring that object creation is handled in one place, rather than being scattered through the code. Instead of having each Viewer class directly create and load sprites, the SpriteLoader class takes on this responsibility, centralizing sprite creation in a single location.

Consequences:

By using the Factory Method, we ensured that all sprite creation logic was consistent across the game. As a result, when changes or new features related to sprite handling were needed, they could be made in one place, simplifying the process of implementing and maintaining sprite visuals. The separation of concerns also benefited the development process. For instance, the one that was working on the mechanics aspects of the game, who was not very familiar with how sprites were loaded and converted into pixels, could focus using the Viewer classes without needing to understand how sprites were created. Moreover, any errors related to sprite loading or creation could now be isolated to the SpriteLoader class, making it easier to fix issues. This made the codebase more maintainable, less error-prone, and easier to extend in the future.

Code Smells

- **Duplication of Code:** In classes where multiple sprites need to be loaded (for example, MenuStateViewer), very similar code is repeated for each sprite. For instance:

  ```
  menuStates[0] = new SpriteLoader("/sprites/menu_states/play_button.png");
  menuStates[1] = new SpriteLoader("/sprites/menu_states/credits_button.png"); menuStates[2]
  = new SpriteLoader("/sprites/menu_states/exit_button.png");
  ```

This leads to duplicated logic for loading sprites. If the logic for sprite loading changes we will need to update multiple locations. The more duplication there is, the higher the chance of introducing inconsistencies or errors.

- **Overuse of Static Methods:** In classes like Arena and Grid, we rely heavily on static methods (for example, isEmpty() and canMoveLeft()). Static methods make unit testing challenging because they affect the global state of the application. They can't easily be isolated during tests, which makes it harder for tests to be independent.

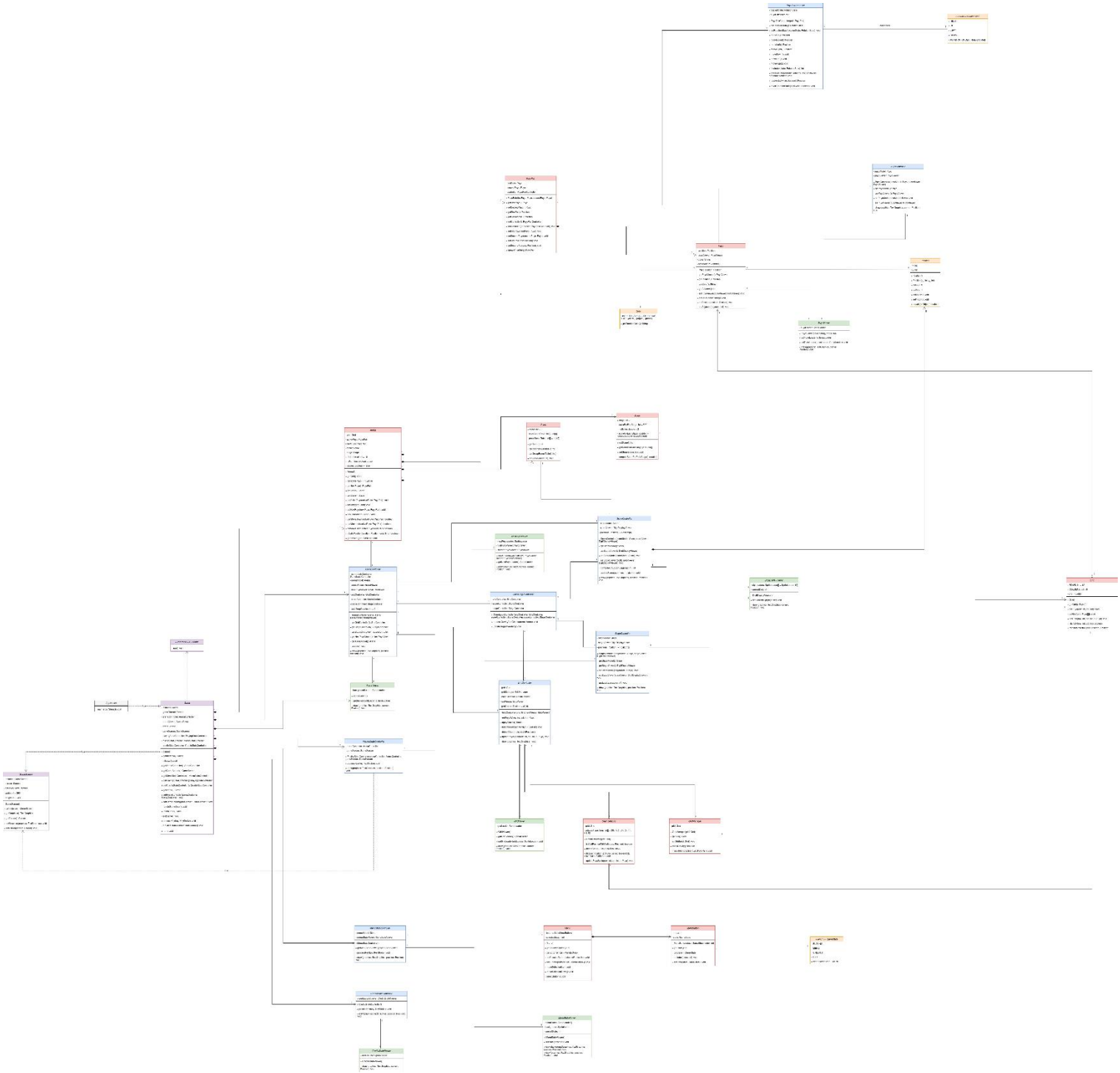
- **High Coupling Between Classes:** In certain classes like Game, there is tight coupling between various components provenient from different classes. This coupling makes the system rigid and difficult to modify. Changes in one class may require changes in other classes, which can lead to a cascade of issues. It also reduces the ability to test classes in isolation since they depend on so many others.

- **Hard-Coded Resource Paths:** Resource paths, such as those of sprites, are hardcoded directly into constructors (for example, /sprites/menu_states/play_button.png). This makes it difficult to update or change resource locations. If the resource structure needs to be altered, every path must be manually updated across the entire project, which is very inefficient.

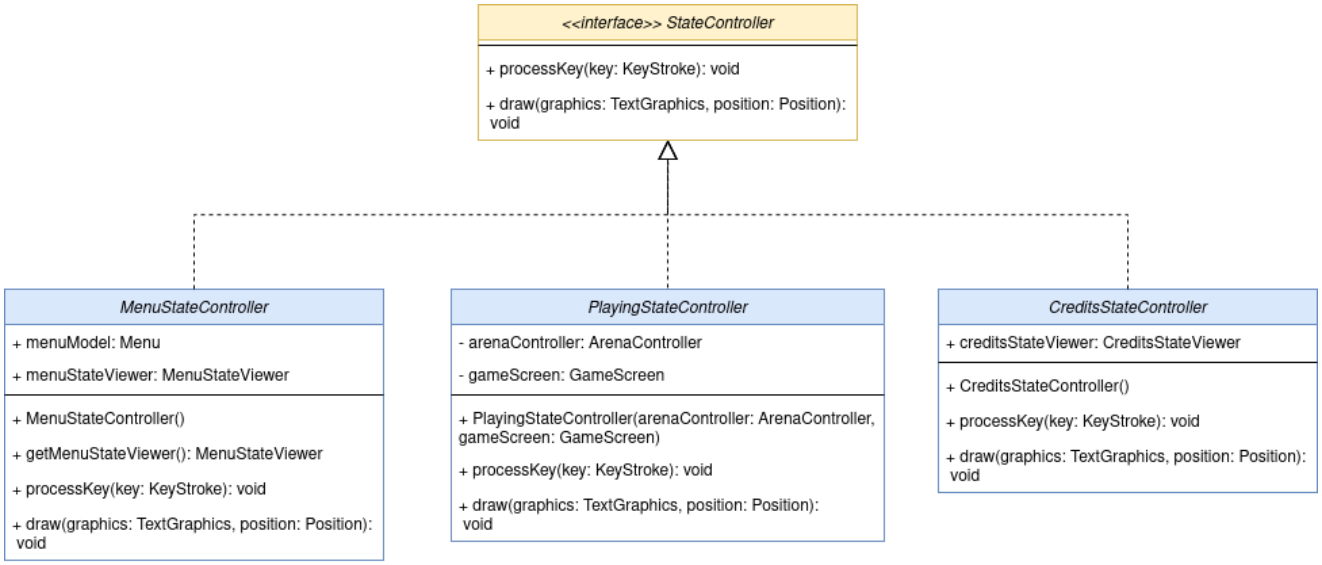
- **Complex Control Flow:** In the Game class, methods like run() and processKey() contain large switch statements that handle state transitions. We couldn't completely eliminate them when using the State Pattern. This makes the code harder to extend or modify. Adding a new state would require updating several methods, leading to an increased risk of bugs.

UML Diagrams

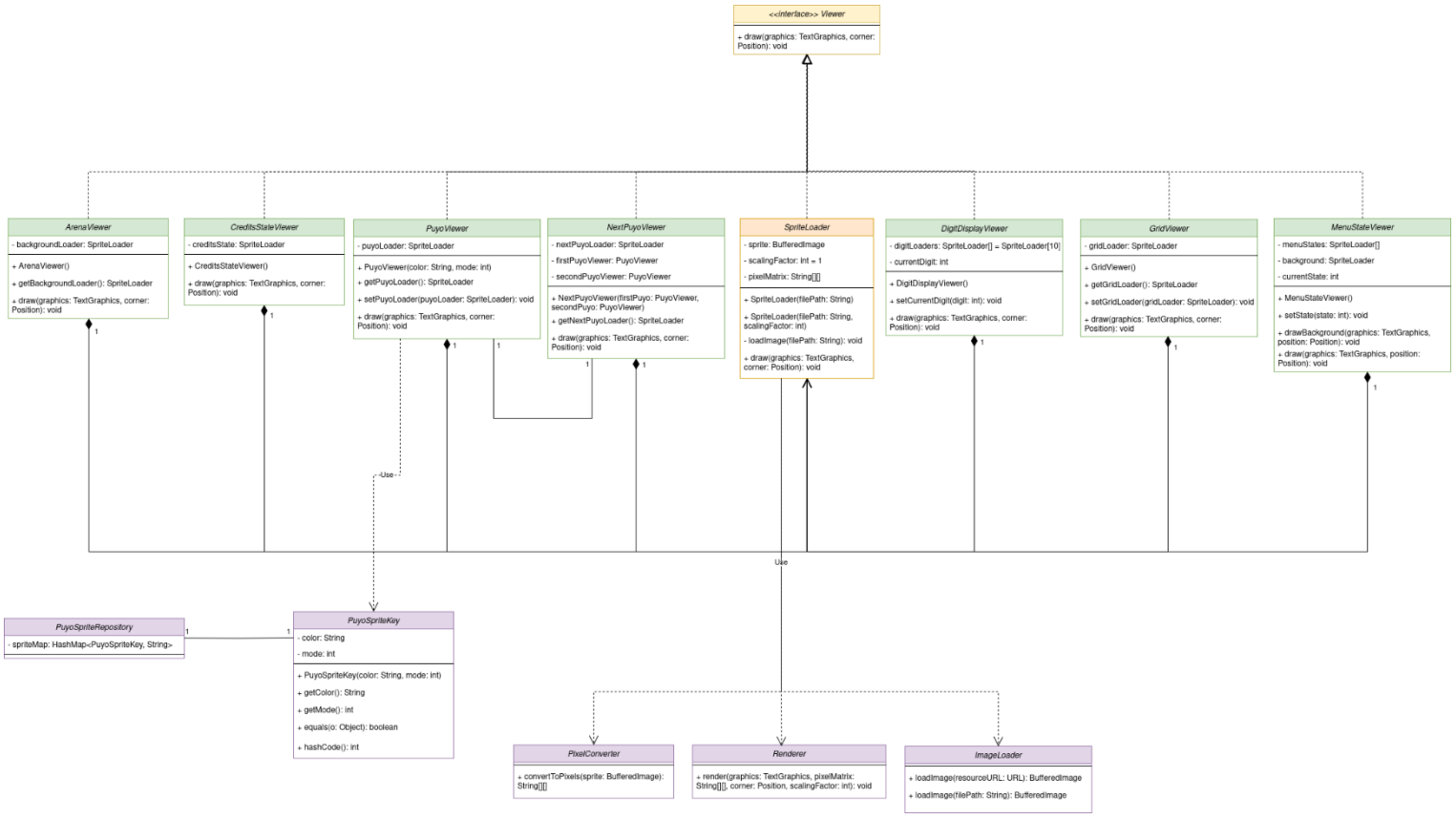
Main Diagram



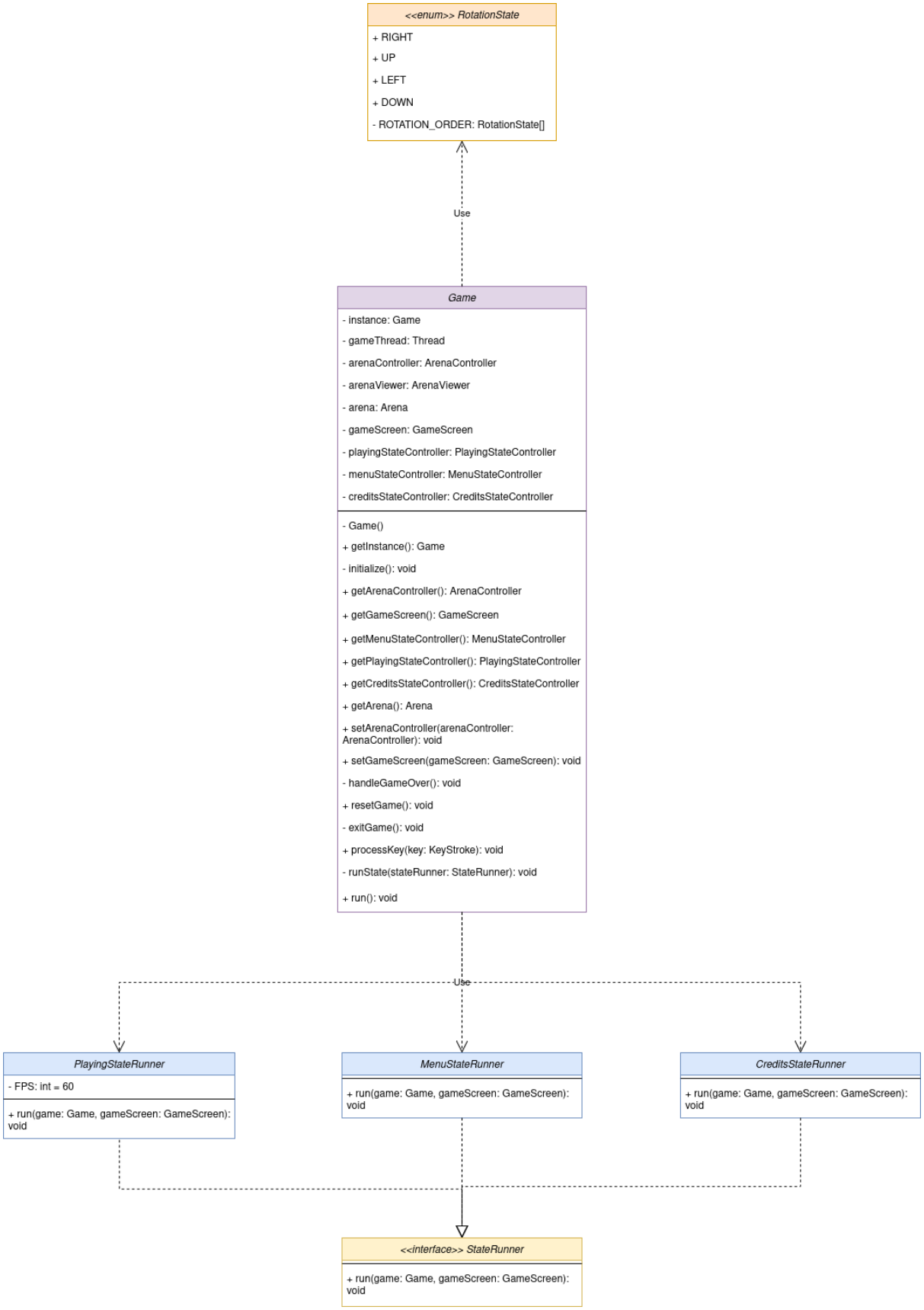
State Controller Diagram



Viewer Diagram



Runners Diagram



Code Coverage

PopPuyo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
poppuyo.gamestates.runners		100%		81%	3	14	0	44	0	6	0	3
poppuyo.gamestates		100%		n/a	0	1	0	3	0	1	0	1
poppuyo.viewer.puyosprites		99%		80%	3	13	1	95	1	8	0	2
poppuyo.model		98%		77%	11	74	1	113	1	52	0	6
poppuyo.utils.puyoutils		95%		90%	3	27	2	42	2	22	0	3
poppuyo.gamestates.controllers		95%		74%	14	38	4	70	1	10	0	3
poppuyo.viewer.loader		95%		100%	1	8	1	19	1	7	0	2
poppuyo.viewer		93%		75%	3	26	5	73	2	24	0	7
poppuyo.viewer.rendering		87%		94%	3	13	3	27	2	4	0	2
poppuyo.controllers		84%		58%	16	87	34	209	2	55	0	8
poppuyo.game		72%		40%	16	39	30	95	6	26	0	2
poppuyo.model.grid		71%		69%	12	46	15	82	0	18	0	3
poppuyo.utils.custom_exceptions		50%		n/a	1	2	2	4	1	2	1	2
default		0%		n/a	2	2	8	8	2	2	1	1
Total	516 of 4,532	88%	82 of 282	70%	88	390	106	884	21	237	2	45

Pitest Report

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
40	82%	707/867	70%	308/443	85%	308/363

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
poppuyo.controllers	7	84%	173/207	66%	67/101	82%	67/82
poppuyo.game	2	1%	1/95	0%	0/31	100%	0/0
poppuyo.gamestates.controllers	3	94%	66/70	90%	43/48	96%	43/45
poppuyo.gamestates.runners	3	100%	44/44	78%	21/27	78%	21/27
poppuyo.model	6	99%	112/113	81%	60/74	81%	60/74
poppuyo.model.grid	3	82%	67/82	63%	50/79	86%	50/58
poppuyo.utils.puyoutils	3	95%	40/42	100%	20/20	100%	20/20
poppuyo.viewer	7	93%	68/73	83%	15/18	94%	15/16
poppuyo.viewer.loader	2	95%	18/19	67%	4/6	67%	4/6
poppuyo.viewer.puyosprites	2	99%	94/95	100%	12/12	100%	12/12
poppuyo.viewer.rendering	2	89%	24/27	59%	16/27	70%	16/23

Report generated by PIT 1.15.0

Enhanced functionality available at arcmutate.com