Software Architecture - Patterns

Michael Brusegard

February 2025

1 The implemented program (HelicopterPong)

The implemented program I chose to refactor in this exercise is the HelicopterPong game from the exercise-introduction technology exercise, written in Kotlin. The game was my sole implementation from the previous exercise, as I had misunderstood that we were supposed to create separate applications for each of the tasks.

The HelicopterPong game consists of two main components:

- A background system with four floating helicopters that move at random speeds and bounce when colliding with each other or the window edges
- A classic pong game featuring dynamic difficulty where:
  - Paddles shrink as the round progresses
  - Ball speed increases over time
  - The first player to reach 21 points wins

When creating the HelicopterPong game in the first assignment, I focused solely on functionality without considering architectural design patterns. The code, while functional, lacked proper structure and organization. Through this patterns exercise, I aim to improve the code's maintainability and structure, while keeping the game's functionality unchanged.

2 Implementation of the Singleton pattern

For the implementation of the singleton pattern, I identified several areas where the pattern could improve the code structure and maintainability. The application already had some implicit use of singleton-like behavior, particularly in the `BackgroundManager` class. During the refactoring process, I made the following key changes:

1. **BackgroundManagerSingleton**: The original `BackgroundManager` class was converted from a regular Kotlin `class` to an `object`, utilizing Kotlin's built-in singleton implementation. This change removed the need for manual

instance management while maintaining the same functionality. The object declaration ensures lazy initialization and thread-safe instance creation.

2. **GameManagerSingleton**: A significant portion of game state management was previously handled directly in the `HelicopterPongGame` class. I extracted this logic into a new `GameManagerSingleton` object, which now manages:

- Game state (started, over, timer)
- Score tracking
- Game object initialization and updates
- Round and game reset functionality

3. **InputManagerSingleton**: Input handling was separated into its own singleton object, removing this responsibility from the main game class. This singleton manages:

- Touch input processing
- Keyboard input handling
- Input state queries

Comparing the original and refactored code shows significant improvements:
Original:

- The `HelicopterPongGame` class was handling multiple responsibilities
- Input handling was mixed with game logic
- Background management required explicit instantiation

Refactored:

- Clear separation of concerns with dedicated singletons
- Simplified main game class
- Centralized state management
- Thread-safe initialization through Kotlin's `object` declaration

While the singleton pattern provided these benefits, I limited its use to only these three core components to avoid excessive global state and maintain loose coupling where possible. This balanced approach helps maintain code modularity while providing the benefits of centralized state management where appropriate.

3 The patterns

I chose to implement both the Model-View-Controller (MVC) and Observer patterns to improve the structure and maintainability of the HelicopterPong game.

3.1 Model-View-Controller (MVC)

The MVC pattern was implemented to separate the game's concerns into distinct components:

Model:

- Created a `GameModel` interface and `GameModelImpl` class to manage game state
- Moved game objects (Ball, Paddle, Score) into the model layer
- Encapsulated game logic and state management within the model

View:

- Refactored the `Renderer` class to focus solely on presentation
- Created a `GameView` interface and `GameViewState` data class
- Separated rendering logic from game state management

Controller:

- Utilized `GameManagerSingleton` as the main controller
- Created `InputController` interface for input handling
- Kept the `AIController` for computer-controlled paddle

The MVC implementation improved the codebase by:

- Providing clear separation of concerns
- Making the code more testable and maintainable
- Allowing independent modification of presentation and game logic
- Reducing coupling between components

3.2 Observer Pattern

The Observer pattern was implemented to handle score updates and notifications:

Implementation:

- Created `ScoreObserver` interface with `onScoreChanged` method

- Modified `Score` class to maintain a list of observers
- Implemented observer functionality in the `Renderer` class
- Added visual feedback for score changes through the observer mechanism

Benefits:

- Real-time score updates without polling
- Decoupled score tracking from visual representation
- Extensible system for adding new score-related features
- Improved responsiveness through immediate notifications

The combination of MVC and Observer patterns significantly improved the code structure while maintaining all original functionality. The patterns work together seamlessly, with the Observer pattern handling specific state changes within the broader MVC architecture. This refactoring demonstrates how architectural patterns can enhance code organization and maintainability without compromising the user experience.

3.3 Additional Improvements

While implementing the architectural patterns, I also made two small improvements to the game:

- Changed the ball from a circle to a square using `rectLine` rendering for better visual consistency
- Added proper window resize handling to maintain game functionality across different window sizes

4 Theoretical questions

4.1 Architectural vs Design Patterns

From the list of patterns provided, the following classification can be made:

Architectural Patterns:

- Model View Controller (MVC)
- Entity Component System (ECS)
- Pipe and Filter

Design Patterns:

- Observer

- State
- Template Method
- Abstract Factory
- Singleton

The main differences are:

- Architectural patterns define the overall structure and organization of a system
- Design patterns solve specific, recurring problems at the class and object level
- Architectural patterns have a broader scope and impact on the entire application
- Design patterns are more focused and localized in their implementation

4.2 Pattern Implementation

In my implementation, I utilized both the Singleton pattern and MVC pattern with Observer pattern support:

Singleton Pattern:

- `GameManagerSingleton`: Central game state management
- `InputManagerSingleton`: Input handling
- `BackgroundManagerSingleton`: Background animation management

MVC Pattern:

- Model: `GameModel` interface and `GameModelImpl` implementation
- View: `GameView` interface and `Renderer` implementation
- Controller: `GameManagerSingleton` and `InputManagerSingleton`

Observer Pattern:

- Subject: `Score` class
- Observer: `ScoreObserver` interface implemented by `Renderer`

4.3 Advantages and Disadvantages

Advantages:

- Singleton pattern ensures consistent state management across the application
- MVC provides clear separation of concerns and improved maintainability
- Observer pattern enables loose coupling for score updates
- Combined patterns create a more organized and extensible codebase

Disadvantages:

- Increased complexity compared to the original implementation
- Singletons can make testing more difficult
- More boilerplate code required for pattern implementation
- Need to carefully manage relationships between patterns

5 Comment

I completed this task individually.