Luleå Tekniska Universitet

Home Exam

Course: D7032E, Software engineering, Lp1, H23

Author:

André, Roaas androa-0@student.ltu.se

October 20, 2023

1 Unit Testing & Identifying Shortcomings in the Original Design

The original code raises several concerns related to best practices and SOLID principles. To start, the code complies with Requirements 1 to 10(b), 10(e), 11, and 12. Yet, the deeply nested structure makes effective unit testing much harder. This complexity stems from a lack of segregation into distinct classes or modules as the code merges networking logic and game mechanics.

Regarding requirements, for 10(c), the Collection amount may contain negative numbers, compromising the score. In 10(d), results are provided for only one animal pair, not all. Requirement 12(a) lacks a solution for tie breaker scenarios.

When delving into best practices, the Player class exemplifies some primary issues. It undertakes many roles, from round score calculations to client-server communications. Ideally, these tasks would be separated to ensure adherence to the Single Responsibility Principle. For instance, score calculations could be better managed by a dedicated scoring system, implying that methods in the original design aren't very primitive. On top of this, the necessity to frequently simulate input makes testing burdensome.

Further problems arise from the system's high coupling. Accessing key classes requires initiating Boomerang, which in turn begins the game automatically. This is indicative of a tightly coupled system where classes like BoomerangAustralia oversee multiple tasks such as game logic and networking. Such high coupling might be detrimental, and as with the ServerSocket and client roles of Boomerang, decoupling is advised.

In conclusion, the code would benefit from reevaluation and restructuring, particularly given the violation of SOLID principles. This would make unit testing easier.

2 Software Architecture Design and Refactoring

2.1 Updated Software Architecture Design

Best Practices

In my refactored code (refer to the UML diagram at the end of the report), there's a clear emphasis on modularity and separation of concerns. Instead of cluttering multiple functionalities into a single class or file, my code has been organized into distinct classes and packages, each focusing on a specific aspect of the game. This not only makes the code easier to understand and maintain but also ensures that any future changes or additions can be made without disrupting the existing functionality.

SOLID principles

The Single Responsibility Principle is more evident in my design considering how each of my classes have a clear and more distinct responsibility then the original design. For instance, the CommunicationHandler class is solely responsible for handling communication aspects, while the GameMode class focuses on the game's core logic. The Open-Closed Principle is reflected in my design of classes like Card, where new types of cards can be added without modifying the existing code. The Liskov Substitution Principle is adhered to, especially in the player package, where subclasses like Human and Bot can be used interchangeably with the base Player class. The Interface Segregation Principle and Dependency Inversion Principle, is alot more evident in my codebase, ensuring that classes don't depend on unnecessary methods and that high-level modules aren't directly dependent on low-level ones.

Booch's metrics

From a metrics perspective, the refactored code exhibits reduced coupling, with each class and package being relatively independent of others. This ensures that changes in one part of the code don't inadvertently affect other parts. My code also showcases high cohesion, with related functionalities being grouped together, leading to a more organized and intuitive code structure. In terms of sufficiency, the classes and methods in my refactored code provide all the necessary functionalities required for the game's operation. The completeness of the code is evident in how it covers all requirements specified. Lastly, the primitiveness is addressed by ensuring that most classes and methods provides fundamental operations without unnecessary complexities or redundancies.

2.2 Design Choices Addressing Quality Attributes

Modifiability

My refactoring has transformed the original codebase into a more modular structure, evident from clear separation of concerns across different packages (again, refer to diagram at the end of the pdf). For instance, the DisplayManager class from the displaymanager package is solely responsible for managing the game display, ensuring that any modifications related to display logic can be made without affecting other unrelated components. Similarly, the Drafting class in the drafting package encapsulates the game's drafting mechanic, allowing for changes to drafting mechanics without impacting other game components. This modular approach ensures that future changes can be localized, reducing the risk of unintended side effects.

Extensibility

My redesigned codebase is good for extensibility. The use of interfaces, such as IInputHandler, IDisplayManager, and IScoring, provides a more clear and comprehensive way for future implementations. For example, if a new game mode needs a different scoring mechanism, such as Boomerang USA, a new class, ScoringUSA can be created that implements the IScoring interface without altering the existing ScoringAustralia class. Similarly, the GameMode class provides a generic representation of a game mode, but specific modes like BoomerangAustralia can extend it to introduce mode specific logic. This adherence to the Open-Closed Principle ensures that the system can be extended without modifying existing code.

Testability

The separation into distinct classes and interfaces significantly enhances the testability of the system. For instance, the IExecutableTask, ITaskExecutor, and ITaskManager interfaces in the Threadpool package can be mocked during unit testing, allowing for isolated testing of components that depend on them. The InputHandler class can be tested independently of the game's core logic, ensuring that input mechanisms function as expected. Furthermore, the modular design means that components like the Player class or the Drafting system can be tested in isolation, ensuring that each module's functionality is robust and reliable.

2.3 Choice of Design Patterns

These are the design patterns that I chose to focus more on when refactoring the original code considering my needs.

The Singleton Pattern (Creational Design Pattern)

I chose the Singleton pattern for the Server class to guarantee a single server instance throughout the application. This prevents potential conflicts and ensures centralized communication, streamlining resource management.

The Abstract Factory Method (Creational Design Pattern)

I introduced abstract classes like Card, Drafting, GameMode, and Player to foster flexibility in the system. This design allows for easy extensions, such as adding new game modes, without major code overhauls.

Observer Pattern (Behavioural Design Pattern)

The server's capability to broadcast to all clients led me to the similar Observer pattern. This ensures real time updates, enhancing responsiveness and allowing the server acts as the subject, and the clients are observers.

Thread Pool Pattern (Concurrency Design Pattern)

Given the system's need for concurrent task execution, I implemented the Thread Pool pattern. This optimizes resource use, boosts responsiveness, and handles high volume client-server interactions efficiently.

UML Diagram

It's very hard to analyze this UML diagram in greater detail. I advise to look at the UMLHighQuality.jpg image inside 'src/boomerang' for better quality when zooming (this image is however cluttered with 'copyright' tags from where i generated the UML and I cant do anything about it, but still better then the image shown below).