  
  
  
  
CS 319 - Object-Oriented Software Engineering  
Analysis Report  
RUNNING JON

Group  
ERKAN ÖNAL

NİHAT ATAY  
BARIŞ ARDIÇ

MERT KARA

**Pattern Applications**

**Adapter Pattern**

The adapter pattern is a design pattern that allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code. Thus, we can use this pattern to make our source code reusable.

In this pattern, an adapter helps the software engineer use two incompatible interfaces to work together. The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

Since Java does not allow us to make multiple inheritances, we can use the Object Adapter pattern. That is, we will have a middle interface that would use inheritance to be qualified of the traits of our actual interface classes (the middle interface will extend the real interface class) and it would use delegation to have access to the system class’ (an engine class) attributes and data.

**Specifying Contracts**

For both our system and the user, the system class methods will warn the user about its acceptable and unacceptable input values or types, and their meanings and also return values or types, and their meanings. Running Jon will have error and exception conditions that can occur, and these will inform the user about their meanings.

Since we use Model View Controller attributes, all class relationships are between user interface classes and engine classes (GameEngine). An interface class is obliged to make calls to its corresponding engine class features where the resulting state of the engine class is not violated by the interface call. Subsequently, the engine classes are obliged to provide a return states and data that does not violate the state requirements of the interfaces, while the game is played. For instance, GameEngine class data buffer may require that data is present in the buffer when a delete feature is called. Subsequently, the GameEngine guarantees to the interface that when a delete feature finishes its work, the data item will, indeed, be deleted from the buffer. Java also requires this attribute by its Garbage Collector.

As for the class invariants it shall be said that the state of the class (any class that is used) will be maintained at the end of each feature execution.

Regarding the correctness of the Running Jon, if the class invariant and preconditions are true before an engine class is called by and interface class, then the invariant and the post-condition will be true after the service. Moreover, when making calls to an engine class, the system should not violate the engine class' preconditions.

**Conclusions and Lessons Learned**

In conclusion, in this report, we aimed to create an action game called Running Jon. Our report has parts such as, requirements specification, System Model, Subsystem Decomposition, Object Design and we also include a part that we clarify our contracts.

In requirements specification, we examined mostly all of the possible requirements which any player could have performed. Moreover, we have indicated our functional and nonfunctional requirements regarding these performances.

In system model, our report consists of the following sections:

1. Use case model

2. Dynamic models

3. Class model

4. User interface

Firstly, we have decided our use case scenarios. Then, in the second part, System Design, we have made our sequence diagrams and activity diagrams. In our sequence diagrams we tried to demonstrate the possible actions that the player could take. Activity diagram indicates mainly our game play. It represents the actions of our game components, cursor, baits, bonuses and penalty objects. Moreover, our class diagram represents our implementation. Moreover, ın the Subsystem Decomposition part, the detailed class diagram is depicted in order to provide a better understanding about the interactions and basic fundamentals of our software. We have divided our class diagram into subsystems or in other words packages. With the help of this class diagram, the detailed subsystem descriptions were made easier to comprehend.

In the Object Design part, we have commented and clarified the attributes of our classes and interfaces that we have decomposed in the Subsystem Decomposition part. Furthermore, we have showed our classes in Class Diagrams and as particular and ındividual class diagrams.

In addition, we have included a contracts part to clarify our contracts of Running Jon. In brief, programming by contract prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, post-conditions and invariants. These specifications are referred to as "contracts”. Moreover, the programming by contract approach assumes all interface components that invoke an operation on a server component will meet the preconditions specified as required for that operation.

Furthermore, this project has been a successful and meaningful assignment for us, on our path to be Computer Engineers. We have seen that in order to make a time wised efficient and user friendly software, we have to make preparatory work and preliminary examinations of the design and clients’ needs. In addition, we have seen that if we use design patterns it is easier for us to change our software attributes without changing all of our source code.

To sum up, Running Jon is a game that we have decided to develop from the idea of Space Intruders. However, its scenario is different than Space Intruders. Our game’s scenario is based on the famous book and also TV series; Game of Thrones. We applied various design patterns and intentional choices that would make our game easily upgradable and reliable. We believe that this small scale project achieved the desired requirements while providing values such as playability and upgradability.