Software Architecture Assignment Semester One

Henry Senior
@00454779

December 2017

Introduction

When designing and creating software, it is common to think about the end user experience. However, it is also important to consider what software development consultant Kevlin Henney describes as the *"Programmer Experience"*. [2]. This refers to the readability, maintainability, and reusability of the source code. He argues that if programmers cannot understand the source code, then they cannot use it effectively. However, as Gamma *et al* (commonly known as The Gang of Four or GoF) state *'Designing object-oriented software is hard, and designing reusable object-oriented software is even harder.'* [1]. This report explains and evaluations the design decisions taken during the creation of my educational game, Bug Busters, and how they impacted on the; usability, efficiency, reliability, maintainability, and reusability of the source code.

Design

Use Case Diagrams

Use case diagrams provide a way of representing the scenarios that are connected by a common goal [4]. Creating use case diagrams is a key part of Responsibility Driven Design as it allows the system architect to go from an abstract brief such as *'a tower defence game where the user plays as the immune system'* to a more concrete understanding of what the final product should offer.

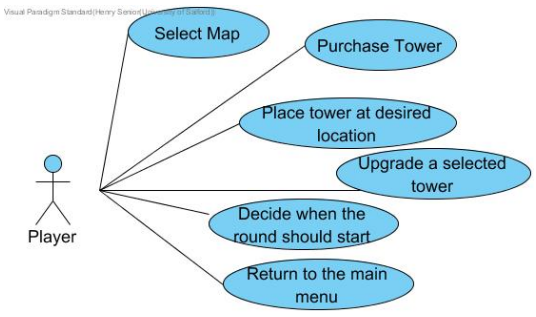

Figure 1: A use case diagram drawn in Visual Paradigm

By creating a use case diagram, I was able to focus on key features that I wanted my game to have. As it was a tower defence game, a lot of the key use cases revolve around the user's interaction with the towers. The fun of tower defence comes from selecting where you think certain towers would be best placed to defend from the oncoming waves. This made it clear

that the user should be able to purchase towers and select where they are placed throughout the map. It also became clear that users should be able to upgrade towers to a certain level. As the game is intended for Primary School children, it was clear that having different difficulty levels would be beneficial as it would increase replay value.

Software Components and Sequence Diagrams

After creating a use case diagram it was possible to decide on the main modules that would make up the game. It became clear that there would be different screens, and these would me grouped into one module. In order to decouple the user interface of the game screen from the screen itself and increase the cohesion the different parts of the user interface have with the other modules, the individual components that make up the user interface were grouped into a second component. The third component would house all the classes that make up the towers and the fourth component would be made up of the pathogen classes.
Sequence Diagrams are especially good at showing the flow between different components when a use case is being executed. A sequence diagram has been used to show the dynamic interaction between software components for the 'select map' use case.

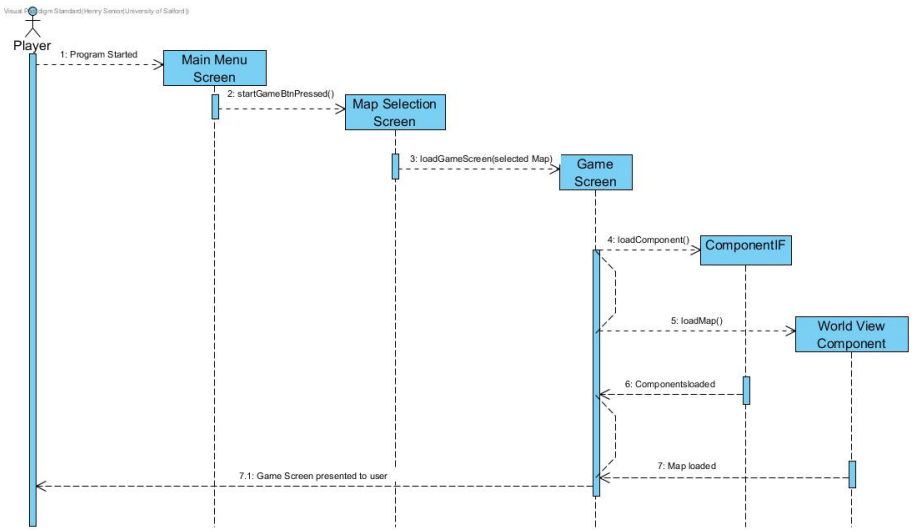

Figure 2: A sequence diagram drawn in Visual Paradigm

Class Diagram

The class diagram shows a detailed overview of the game and shows the relationships between different classes within the game.

Figure 3: A class diagram drawn in Visual Paradigm

Design Choices

When designing the game, focus was placed on ensuring low coupling between the different classes that make up the game as well as ensuring high cohesion between an object and it's behaviours.
One of the design choices I made was to break the sections of the user interface that make up the GameScreen object into separate components. By providing the components with a common interface, *ComponentIF*, I was able to ensure the components could be interacted with in universal ways. Dividing the GameScreen into smaller components, I reduced the coupling within the user interface. This made it very easy to make changes to the user interface without those changes having effects on other portions of the interface. To make the separation of the user interface components and screens explicitly clear for future developers, I grouped the classes that inherrit from ComponentIF together in a package, alongside the interface.

Design Patterns

Design Patterns originate from structural architecture and refer to generalised solutions to common problems. It was found by Gamma *et al* that a similar approach could be applied to object oriented software design. [1]. The software design patterns used in the project are outlined below, with detail and critique on the implementation.

Singleton

The goal of the singleton design pattern is to ensure that only one instance of the object can exist at any given time, also ensuring that the object has a single point of global access [1]. The singleton design pattern is widely used within the project as there many objects of which only one needs to exist. By ensuring there is only once instance of any user interface components that make up the GameScreen object, the components become well encapsulated as well as more memory efficient. This feeds back into the *'programmer experience'* as at any given time in development the programmer can be confident that only one instance of the object exists, meaning any change made to it will be processed by any other objects that interact with the singleton.
The singleton design pattern is also implemented in the player object. Throughout the game, only one player will be needed as it is only single player. Using a singleton design on the player stops an instance of the player having to be passed between different parts of the game, and allows the player to keep a strong encapsulation. Without using the singleton design, the player object would have to be static, which would limit further development as it would greatly increase the coupling between the player object and the objects that interact with it. There are instances where the player needs to be reset, such as the player dies and starts a new game. To accommodate for this event, the player class also includes a setToNull() method, which sets it's instance to null. This results in a new instance being created the next time an objects tries to access the player - resetting the field variables held by the player.

Model View Controller (MVC)

Originally created for use in SmallTalk, the MVC design pattern achieves the decoupling of the user interface and the logic of the program. In modern systems, MVC is common in web applications as it decouples the application in such a way that developers can apply their skills to the most applicable area (front-end developers work on the view whilst back-end developers work on the model).
The MVC design pattern was implemented into the game through using a controller object

which handles events from UI Components. The implementation of the design pattern, only event operations that use multiple interface components, such as placing a tower, are decoupled. This ensures cohesion between buttons and their events. The controller class provides a central point through which the user interface components can communicate.

Factory

In order to decouple the creation of pathogens and towers from the user interface, a factory was incorporated into the project. An abstract factory interface was produced, allowing method prototypes to be defined. Concrete definitions of the factory interface were developed to provide factory objects for creating pathogens and towers.
The *'programmer experience'* was greatly improved through the use of factories as meant the towers and pathogens could change, but the way they were created and displayed on the screen for the player to interact with.

Command and Marker

The Pathogen class utilises the command pattern as part of it's path finding algorithm. Before a pathogen moves through the map, it searches for tiles which it can move to and builds a queue of movement commands that it will need to execute to move through the map. As the commands required by the pathogen are the directions it must take to navigate the map, the marker design pattern can be used to recast the direction as a command that requires execution.

Null Object with Delegation

One design that was considered and then rejected was the use of the null object design pattern alongside the delegation pattern. The Null Object pattern allows an empty object to be created and used like a standard object, but the behaviours are overridden so they have no effect. It would be feasible to delegate a pathogen to a null object pathogen when a tower kills the pathogen. The use of the null object pattern was rejected as it was not the most memory efficient solution. By adding a pathogen to a removal list once it was dead, and then removing it from the main pathogen list once that list had been iterated, the amount of memory the game used was reduced. This will provide the game with better performance and is a design choice that makes it more scalable. Through freeing up memory, it possible that more difficult rounds could have hundreds of enemies rather than tens of enemies.

Future Improvements

Interpreter Design Pattern

In the current implementation, the map and the enemies that the player face are both hard coded into the game. Whilst this is removes the need for map files, it tightly couples the map and rounds into the game's source code. A better design decision would have been to implement an interpreter design pattern. The interpreter pattern allows for a language to be defined alongside it's grammar using Backus-Naur Notation[1]. These rules can then be used to interpret a string that follows these rules. Using the interpreter pattern it would be possible to decouple the map files and the enemies that are played each round to separate files. This would also allow for the users to create their own maps if they so wish.

Prototype Design Pattern

Through the implementation of a prototype design pattern, the game could clone instances of the bacteria pathogens as they move through the screen. This would help teach the end user that bacteria can multiply, but viruses cannot. The prototype design pattern allows for objects to be cloned and then edited[1] - which would allow for a bacteria object to be cloned, but be given a different graphic, signifying that it had been cloned.

Conclusion

Whilst the project was challenging, it provided a good learning experience. Through the production of an educational game, I was able to learn about design patterns within software architecture and experience first hand how different design choices impact the project later on. For the first time since I learnt to program, I was able think of a feature that needed to be added, and then add it without the worry of something else breaking. This has shown me how effective design patterns are as well as demonstrating the power of following good software practice, such as; lose coupling, high cohesion, and Parna's Principles.

Media Sources

Whilst I created all the graphics used in the games, I used sound files from royalty free sources.

1. Background.mp3, Ben Sound, `https://www.bensound.com`
2. default-laser.mp3, Sound Bible, `http://soundbible.com/1087-Laser.html`

References

[1] Gamma E, Helm R, Johnson R, Vlissides J. 2005. Design Patterns: elements of reusable object-oriented software. Addison Wesley.

[2] Henney K. 2017. FizzBuzz Trek HD. `https://www.youtube.com/watch?v=LueeMTTDePg`. Accessed December 2nd 2017.

[3] Drumm, I. 2017. Software Architecture Lecture Notes. University of Salford

[4] Fowler M, Kendall S. 1999. UML Distilled: a brief guide to the standard object modelling language. 2nd Ed. Addison Wesley.