Architectural design

Large and Heavily Armored Warships for Android

Primary quality attribute: Modifiability

Secondary quality attribute: Usability

Trond Kjetil Bremnes, Marius Glittum, Alexander Perry
Even Stene, Elisabeth Solheim, Trond Klakken

February 27, 2012

Chapter 1

Introduction

This document will provide a documentation of the architecture for Large and Heavily Armored Warships (LaHAW), a game developed for Android. The game is developed as a part of the TDT4240 course at NTNU.

This document will focus on architectural design choices and how we satisfy the quality requirements set. Using the IEEE 1471 standard for architectural descriptions of software-intensive systems[3] we describe architectural tactics and patterns based on the defined architectural drivers and stakeholders. A description of the 4+1 architectural viewpoint[6] and a rationale for our chosen solution is also provided, and how quality and functional requirements influence the choice of architecture.

Patterns used for the realization of LaHAW is Model-View-Controller as an architectural pattern; and Singleton, Observer and State as design patterns. This document will detail the reasoning behind these choises.

1.1 The game

LaHAW is a remake of the classical "Battleship" game[5]. Battleship is a turn based guessing game situated on an grid based ocean space. The game is separated into two parts: an introductory preparation state, where both of the players places their ships on their own ocean grid; and the guessing state, where the players takes turn in guessing where the opposing player has placed his or her ships. A correct guess is effectively noted as a 'hit'. Depending on the size of the ship, several hits might be neccessary to sink the ship. The game ends after a player has hit and sunk all of the opposing player's ships.

Chapter 2

Architectural drivers

We have identified the following architectural drivers for the project: the experience of the developers; need for a short time-to-market; the targeted market and; portability. These architectural drivers are the major quality attributes which influences the system's architecture, and in turn the system's architectural tactics[4].

2.1 Developers' experience

The project will be developed little to no experience with Android game development. For the project to be successful, the game will be realised by early creating a functioning code base consisting of critical functionality, which then will be expanded upon interatively.

2.2 Short time to market

The product needs to be finished and deliverable by the end of the course duration, which means that the game needs to be quick and easy to develop. As mentioned above, the game will be developed iteratively, making a playable game available as early as possible, yet expandable if time allows for it.

2.3 Market

LaHAW will be targeted towards youth and young adults, and is to be used on devices with fairly small screens ranging from 3 to 5 inches. Because of this, the game needs to be easy to understand and use, and usability is a primary focus for this project. Furthermore, for a game to be addictive and fun, it needs to be as intuitive as possible.

2.4 Portability

Future development could include conversion of the game to Apple iOS and Windows Phone 7 platforms for market expansion.

Chapter 3

Stakeholders

For this project, the following stakeholders has been identified:

3.1 End user

The end user of this software will be the players of the game. To make the game appeal to these: Playability: How easy the gmae is to understand and play. This means that instructions should be easy to read and understand. Modifiability: Customizing the players name and the color of ships.

3.2 Developer

Buildability: The product needs to be finished within a short time period. Modifiability: The product should be easy to extend. Testability: The product should be easy to test and verify

3.3 ATAM evaluator

Reviewability: Well written documentation should be used as the basis for an architectural review. Evaluators want the system to be as easy to test as possible, to ensure that all features can be tested. Good testability will therefore result in less work for the testers, and in turn a lower cost.

3.4 Course staff

Reviewability: The course staff will evaluate our final product according to certain criteria like how good our documentation is, how easy it is to read and how easy it is to setup and run our product.

Chapter 4

Architectural Viewpoint

For this project we have chosen to make use of the *4+1 view model*, where we have put most ephasis on the *Logic*, *Development* and *Process* views.

4.1 Logic View

Purpose: In the Logic View we decompose the functional requirements into different classes and show the relationship between these requirements. This in order to showcase the functionality that the system provides to the end users. These classes supports showing the level of abstraction, encapsulation and inheritance.The decomposition also helps identify elements that are shared across the system.
Stakeholders: Course staff, ATAM evaluators and Developers
Form of description: UML Class diagram.

4.2 Development view

Purpose: In the development view we illustrate how the different parts of the system looks a developers perspective. Different parts of the system is divided into layers. This view supports allocation of requirement and work to developers and also monitoring project progress.
Stakeholders: Course staff, ATAM evaluators and Developers
Form of description: Architecture Layer Diagram.

4.3 Process view

Purpose: In the process view we illustrate how different tasks are combined into the final product. In the process view we consider performance and availability requirements, and also system integrity and fault tolerance. The process view can specify which thread of control executes operations in the classes identified in the logic view.
Stakeholders: Users, Course staff, ATAM evaluators and Developers
Form of description: Graph?

Chapter 5

Architectural tactics

5.1 Modifiability

The code is divided up into several different objects and classes in order to separate code changes so that it only affects the object/class being changed. This is done to avoid a ripple effect i.e. the need to change other related components or modules in the architecture when one part of the game is changed. Internal variables should generally be held private, and access to different objects and classes are made through public methods. Objects for boats are made as general as possible to prevent major code changes when initializing a new boat. By just changing some attributes to the common object, one can make a new boat with a given size and color.

5.2 Availability

In order to maintain a degree of availability, LaHAW will implement and handle common exceptions where necessary. This tactic is fast and simple to implement, ensuring a comprehensive architectural design. The project has a limited time frame, and the architecture will reflect that.

5.3 Performance

Performance is not the main focus area of LaHAW, but response time during the game should be as short as possible to give a smooth user interaction. Java does most of the garbage collection, so the goal is to limit the amount of allocated objects at once. In addition, one can increase computational efficiency by making efficient algorithms in critical areas.

5.4 Testability

As per the MVC pattern, the user interface and game logic is held separated, which in turn increases testability. In addition to this, the modifiability tactics require further separation of different sections, with control over input/output.

A built-in monitor of CPU usage and memory load will ensure that the program runs smoothly while avoiding memory leaks.

Our use of singletons are a detriment to testability, but the use of such classes are to be held at a minimum.

5.5 Usability

Usability is achieved by dividing the user interface and the rest of the application, as mentioned in chapter 5.4. MVC is the chosen method here to support modification to the user interface without affecting other parts of the system. During configure-time, the user gives the system a model of their experience level by selecting a difficulty level.

The game will implement a hint system where, during run-time, the game will present hints that shall help the user in understanding the game rules and how to interact with our implementation of the game. This to support usability by giving the user more confidence in their choices. These hints are related to the difficulty level chosen by the user at configure-time.

5.6 Security

LaHAW does not authorize or authenticate users as it require no personal information or other personal details to play. The game is not seen as susceptible to attacks from external sources, and does not send out any information from the device. Therefore, the architecture will not focus much on security in regards of encryption and access limitiation. As mentioned in chapter 5.2, there is a limited time frame from now to the project deadline, and other tactics have a greater priority.

Chapter 6

Architectural Patterns

6.1 Model-View-Controller Pattern

Large and Heavily Armored Warships shall implement the *Model-View-Controller* (MVC) pattern, an *architectural* pattern, where the program code is divided into three separate compartments, the *Model*, the *View* and the *Controller*. The *Model* will contain the code that keeps track of data relevant to the *View*, data which is manipulated by the *controller*.

6.1.1 Implementation

Each of the game world's objects and game data[1] constitutes the *Models* in this architecture. These models keeps track of the present status of each of these elements. Updates to these elements is handled by the controller.

The *View* contains the code that draws the game world as it is dictated by the *Models*.

The *Controller* handles the calculation of the game data. This includes the handling of events (including timers and player input), keeping track of turns and modifying the game's objects. Whenever an event occurs, the controller shall interpret this event and do the neccesary calculations and manipulate the game's models, and then notify the view that the game world has changed and needs to be redrawn. An alternative implementation of *MVC* could use separate controllers for each of the game's models, but seeing as the game is rather small and that the controller will mostly be controlling the states of the classes, this shall not be done in this implementation. .

6.1.2 Rationale

The reasons behind using the *MVC* architectural pattern for the implementation of the game, is to increase modifiability through the separation of the SOMETHING, the visual representation of the game and the code logic. Additionally, use of the *MVC* pattern will also ...

[1]Including the players, their scores, ships and oceans.

6.2 Singleton Pattern

Large and Heavily Armored Warships shall implement the *Singleton* Pattern, a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a class to one object[7]. Use of the Singleton pattern assures that such classes can only have *one* instance which is globally available to those in need of it and its methods.

6.2.1 Implementation

The pattern will be implemented in LaHAW wherever only a single instance of a class is needed, and this class is needed globally at some point in time. Use of singletons in the LaHAW code base will however be limited as much as possible, due to it making unit testing more difficult[1].

- The game itself shall be implemented as a singleton.
- The game scoreboard shall be implemented as a singleton.
- The *states* of the objects, as discussed below, are to be implemented as singletons.

6.2.2 Rationale

By using the Singleton pattern, we can be sure that wherever exactly *one* instance of an object should exist in the game, nothing else is allowed.

6.3 State Pattern

State Pattern is a behavioural *design pattern*

With the use of the State Pattern, the objects that constitutes the game can exist in exactly *one* out of a number of internal states, whose change alter the behaviour of the objects.[2]

6.3.1 Implementation

The way the state pattern will be implemented in LaHAW is described below.

- The players of the game will have a set number of ships, each of which is occupying a set number of the ocean's tiles depending on the ship's size. Each of the tiles can thus be in one of two states: *occupied* or *vacant.*
- The ships are consisting of tiles. The amount of tiles a ship consists of is dependent on the ship's type (which dictated the ship's size). Each of these tiles can be in one of two states: *healthy* or *destroyed.*
- The ships themselves can exist in one of two states: *floating* or *sunk.*
- The players themselves can be in one of the following three states: *won*, *lost* or *play.*
- During *play* state the players will swap between two states: *fire* or *observe*

The states themselves shall be implemented as *Singletons*, as discussed above.

6.3.2 Rationale

The State Pattern is implemented due to most of the game's objects existing in different states as the game proceeds. Rather than implementing these states as separate classes, we can modify the behaviour appropriately with inclusion of state modifiers. An example of this is that we need not implement two types of ocean tile objects (e.g. `OceanTileVacant` and `OceanTileOccupied`), but are able to have a single `OceanTile` object which can exist in different states, depending on how the game plays. This limits the amount of redundancy in the code, and aids in making the code base more readable and modifiable.

6.4 Observer Pattern

The final pattern LaHAW shall adopt is the Observer design pattern wherever one-to-many messages are to be sent between the classes and methods of the game. Whenever a state observed by a number of classes, state changes are to be broadcasted to these listeners so that each of these are kept up to date on this change

6.4.1 Implementation

Below is a description on how the Observer pattern will be implemented in LaHAW.

- When the game controller gets an update that tells it that ship X has been hit, this message is then broadcasted to all of the listeners. This includes the players and the methods that draws the ships.

6.4.2 Rationale

The reasoning of implementing the Observer pattern for the game is to be able to easily send one-to-many messages. The game will consist of several classes that need to broadcast their status updates to several parties. The Observer pattern will aid in this task. The implementation of the Observer pattern will also help in making the code base be more maintainable and modifiable. This is because further evolution of the game's classes and methods might need to access state data, and should this be able to access this data through adding themselves to existing code as listeners. Furthermore, the use of the *State* design pattern and the *MVC* architectural pattern, the use of the *Observer* pattern comes rather natural.

Chapter 7

Views

Bibliography

[1]

[2]

[3] IEEE Standards Association. Ieee 1471, February 2012.

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[5] Boardgame geek. Battleship, February 2012.

[6] Philippe Kruchten. Architectural blueprintsthe ”4+1” view model of software architectur.

[7] Sourcemaking.com. Singleton pattern.