CMPT 370 Software Design Document for BattleBots

Mackenzie Power

Haotian Ma

Ryan Tetland

Will Revell

Mitchel Kovacs

Architecture Design

The chosen architecture for designing and implementing the BattleBots program outlined in the requirements document will be a Model-View-Controller.

Model-View-Controller consists of three different parts: Model, View and Controller. The Model does not rely on either View or Controller (it does not care how it is being controlled or viewed) it just controls the behavior of the application following by the rules and manage the data of the application domain. Model can update and display the state to View so the user can see what changes has he/she made by using the Controller. View needs to register into the which Model it wants to view, once it has been registered it will display whatever the Model allows it to the user. If View is the output tool in this relationship, Controller will be the input tool in the relationship, it interacts with the user by taking user’s input from either keyboard or mouse and informing it to the Model.

There are a number of reasons why this architecture has been chosen over other types. The primary reason for this design choice is that all members of the A5 software team already have a familiarity with how this architecture should work. All members of the group have taken computer science courses which involved creating programs based on a Model-View-Controller architecture, so experience with this method is present. We acknowledge that generally a Model-View-Controller architecture is a more efficient architecture for projects that are much larger than this. However, we feel as though our experience with this architecture relative to other architectures outweighs any difficulties we may have in implementing such a small program with a Model-View-Controller. Another reason why this architecture was chosen is because of the heavy use of visual displays driven by the user (relative to other types of programs). This program is driven by the users' input. Everything from viewing the rules; selecting team attributes; choosing game properties; and executing individual moves require input from the user in order to be performed. Every action done by the user will require a change in display either as a new interface window or as a visual change in board state. There will be very few actions done without the user's input.

A benefit of this architecture is the separation of the interface and the model. This will make the code easier to work on because each class will have a specific purpose. This will result in less coupling because changes made to a class will not affect many other classes. Cohesive classes also result in much easier testing because each class can be tested individually.

Despite all these advantages, there are some disadvantages that we must acknowledge. One disadvantage with Model-View-Controller is it may be difficult to divide the work other than between the model, the view and the controller. This is because these parts of the architecture have many interactions within themselves. Another disadvantage is that it may take more time planning the interactions properly to interact without having too much unnecessary coupling. We feel as though the benefits of a high level of cohesion achieved through this architecture will outweigh the time costs we may endure during designing and implementing the program. The aspects of Model-View-Controller architecture are……

MVC vs Pipe-and-Filter

Pipe-and-Filter contains individual programs transforming input data to output data, it is different than MVC because it only simply consist with the three main parts Model, View and Controller. Base on what we have learned, we see Pipelines as an unidirectional tool, it may going to cause us some unnecessary troubles for the later designing stages. Additionally, we know that Pipe-and-Filter has very high reuse potential but for us to look back in the individual parts will be a huge amount of time consuming, MVC seems pretty friendly for editing because the Model is basically the only portion we need to edit if it is needed.

MVC vs Layered System

Layered System is effective for separating concerns by creating multiple layers, also reduce the impact of change when changes don’t affect layer interfaces. But because they are too many layers, program’s performance will be degraded, especially it is a multi players game that is going to challenge players patient which we do not want to see that.

MVC vs Blackboard Style

Since we are not designing this game for networking reason, blackboard is not appropriate for this program.

MVC vs Independent Component

To have all Components individually allows decoupling and autonomy of components it also makes enhances reuse and evolution much easier due introducing new components without affecting existing ones. However, the connections between components are not guaranteed for example: components announcing events cannot be sure getting a response nor have control over the order of responses. MVC on the other hand, View will always watching Model once it gets registered and Controller has the control about the output through Model.