Design Overview A6: Graphical User Interface Design

Jack Solon - jps386
Jack Pertschuk - jp954

Summary (50–300 words):

In this assignment we added a view package and a controller package to follow the MVC design pattern in our program. In our MVC implementation we utilized event handlers and the observer design pattern to facilitate communication between the view, controller, and model within the program. We used a top-down implementation strategy -- first implementing the view completely before implementing the controller and completing the program. This design strategy was natural because of the structure of our program and the nature of the MVC design pattern. We tested the program manually using user test scripts that test all the functionality of the GUI and the simulator, and are reproducible. After passing this series of tests we had people not in this class play around with the program to make sure that it worked as expected, and was easy for new users to learn. Jack P designed the UI and implemented most of the view package, while Jack S implemented most of the controller package and wrote the design overview. Both Jacks worked on the written problems and on testing. While Jack P was the lead on the view package, and Jack S was the lead on the controller package, both Jacks worked on the classes inside each package as needed, and they met often to discuss problems with implementation.

Specification (10–500 words):

In this assignment we added two packages: one for the view and one for the controller. These two packages interact with each other, and the controller interacts with the model to complete the MVC design pattern. The model in this case is the rest of the program that was implemented in past assignments (the parser, ast, interpreter, etc.). The view package contains the fxml file that defines what the GUI looks like, and a View class that loads the view from the fxml file and communicates with the controller using action events and the observer pattern. To use the observer pattern we added an Observer interface and implementation in the view package, and had the View file observe the controller. The controller, which implements an Observable interface defined in the controller package, then is able to easily communicate the view when it needs to update. The Controller is the bridge between the model and the view. It holds all the information about the simulator that is being run (the Simulator object and the critters/interpreters/etc. that are necessary to run a simulation) and updates the view when there are changes to the world state using the observer pattern. The view then communicates with the controller using event handlers when the user interacting with the view wants something to happen. We chose to use observers and event handlers in our MVC design because this is a simple way to create a modular program that obeys the MVC design pattern. Using this method cuts down on the amount of new code we needed to add, while allowing for a program with a strong MVC design.

Design and Implementation (150–1000 words):

As was previously stated, in this assignment we implemented a view package and a controller package. These packages communicate with each other through event handlers (defined inline in the View class) and using the observer pattern where the View is the observer, and the controller is observable. To use this design pattern we created an Observer interface in the view package that is implemented by the View class, and an Observable interface in the controller package that is implemented by the Controller class. This design allows for easy communication between the view and controller, while keeping the MVC modular. The view package also contains a fxml file that contains the GUI design. This file was created using java scene builder, which allowed us to easily create a user-friendly GUI. The GUI shows the hex map with the critters on it at their location. The hex map also depicts rocks as gray hexes and food as red hexes with numbers inside that correspond to the amount of food that is contained within that hex. When it's first loaded it opens a random world with no critters and randomly placed rocks. From there, the user can add a critter at a specific hex coordinate ("load critter"), load any number of critters to random hexes ("load critters"), or load a world file ("load world"). Above the hex map, the number of critters in the simulation and the number of steps taken by the simulation is displayed. If the user clicks on a critter on the hex map, the critter species, hex coord, program, last executed rule, and mem information are displayed. To step the user simulation forward, the user can either press the "step once" button, which moves the world forward one step, or press the "start/stop" method to set the simulation running continuously at a rate set by the slider where the integer units are steps/second. Event handlers are used to notify the controller of any changes. The Controller class implements the Observable interface and is in turn observed by the view, so the view can update itself with any changes that occur in the simulation. The Controller will hold the instance of Simulator that is being displayed by the view. This is how the controller communicates with the model and is able to then communicate with the view. We chose to implement the MVC model in this way as it seemed to be the simplest (involving the least amount of added code) possible way of implementing the MVC while keeping the program modular. The most important parts of our code is the creation of Observer and Observable interfaces, and their implementations. This is a crucial part of our program as it is the sole way the view will know of any updates done to the simulation. The controller must be able to communicate with the view about any changes to the simulation done during each time step, so that the view can stay updated and that the user can see what is happening inside the simulation. The most difficult part of our program was handling concurrency in the view and controller. We had to take special care to make sure that threads weren't interfering with each other, and that everything in the program was synchronized to ensure complete correctness in both our simulation and in the view. We used a top-down implementation strategy during this assignment where we started by designing the view, and implementing the classes inside the view package (Observer interface and View class), before moving on to implement the classes within the controller package (starting with the Observable interface, then the Controller class). We decided to use this implementation strategy as it seemed natural to design the view in its entirety before designing the controller whose job is to communicate with that view. We also had to test this project manually (discussed further below), so by implementing the view first we were able to make sure the view worked and looked good before we spent time linking up the view to the rest of the program through the controller, and then being able to test the whole program together.

Testing (150–1000 words):

Because we designed a GUI in this assignment, we decided the only way to adequately test it was to manually test the functionality ourselves. This lends itself well to our top-down implementation strategy as we were able to manually test the view before the controller was fully implemented to make sure the view looked ok and all connections were made. To ensure event handlers were working correctly before the controller was implemented we used many println calls to print messages to the console, so we could make sure any buttons or text fields that the user interacted with were registering with the view. After implementing the controller, we were able to manually test the program as a whole, making sure the entire simulation worked as expected. To do this we wrote and followed user test scripts that detailed steps to take when interacting with the GUI to make sure it worked as expected in all cases. In these testing scripts we will also test boundary cases such as making sure the GUI reacts as expected to unexpected user input. After the program passed our own test, we had other people play around with the program to make sure the user interface was intuitive and easy to learn in addition to being correct and behaving as expected. Once the program passed these tests we knew it was complete.

Known problems (1–500 words):

No known issues.

Comments (1–200 words):

n/a