Game: Square Swapper 5000

Teammates: Jeremy White and Yifan Song

Project Breakdown

Discussion and Planning: November 16, 2014

We share our initial thoughts and ideas about the project. The classes, methods and algorithms to implement are discussed. Specifically, Jeremy starts writing the header files and the UML diagram while Yifan starts working on the plan of attack. We believe this is an efficient separation of tasks as Jeremy is able to update the UML as he writes the header files. Meanwhile, Yifan lays the plans and deadlines and highlights the main design pattern we should implement.

Separation of tasks and test suites: November 17, 2014

The header files are reviewed together and we share any new ideas or improvements we might have. We decide who implements which class who writes the main program. We will strive to have classes with high modularity in order to be able to implement the different classes independently. We start developing test suites for our program and for individual modules.

Implementation and debugging (text program): November 20, 2014

Efforts should first be put into implementing a functioning main and a simple level. Specifically, we want to be able to accept all the required commands and command line arguments. Our tests suite should be completed by now to allow us to test our implementation. Jeremy implements the different type of Squares, a basic level class and some methods in the boardManip class, which holds most of the game's logic. Yifan's main role is to implement the methods of the boardManip class and the Score class.

Addition of levels and further debugging (text program): November 23, 2014

Assuming the basic parts of our program runs correctly, we start implementing level 1 and level 2 and test them accordingly. Our current plan is that Yifan implements level 2 and Jeremy implements level 1. The implementations of the other classes and of the main program are modified if needed.

Implementation of the graphical program: November 25, 2014

We add classes to handle the graphical output to our program and add fields and methods to existing classes. We do not expect this step to require a high amount of effort so we leave it as a low priority task. In fact, this step should be very similar to the graphical part of question 3 in the fourth assignment. The separation of tasks for this step is still unclear as we only have an approximate idea of the work required.

Implementation of additional features: Due date 2

If our program correctly implements all the required features, we will start implement additional features to our program. As this step is a bonus, we will only start to discuss which bonus features we want to add at this date.

Implementation Questions

How could you design your system to make sure that only these kinds of squares can be generated?

Each special square should be a subclass of the Square class. Thus, whenever a square of any type is generated, the constructor of the appropriate subclass is called. If the generated square is a basic square, then only the Square constructor is called.

How could you design your system to allow quick addition of new squares with different abilities without major changes to the existing code?

The squares will be generated with the factory pattern. In addition, every type of square is a subclass of the square class. That way, all the logic for generating square is compartmentalized in the level classes. Thus, we only need to create a new square subclass and modify one method in each level class if we want to add new squares to our game.

How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

Each concrete level is a subclass of an abstract level class. Implementing new levels thus only requires the addition of a new concrete level subclass and some modification to the main program. We note that the level classes are not implemented using the decorator pattern as levels do not add more features to a basic level. Instead, each level has its own unique features and

some unique implementation of some common features. For this reason, we cannot combine different levels, as the Decorator pattern would allow.

How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation?

The main program should handle as little of the game logic as possible. As such, the commands given to the main program should translate into the main program calling methods of the Board, BoardManip and/or Level classes. Thus, the addition or modification of commands only leads to changing and adding case in the main program's switch statement.

How difficult would it be to adapt your system to support a command whereby a user could rename existing command?

We expect this to be a simple task. The command names can be saved as strings in the main program. When the user inputs commands, the main program compares the input with those strings. It is possible to add a "rename" command that would reassign a given command string with the new command name given by the user.