Plan of Attack for Square Swapper 5000

Fred Cao and Kevin Xue

We plan to make Square Swapper 5000. We will achieve this by first creating a UML to follow and setting a repository on GitHub to cooperatively work on the project. During the first week, we plan to write out the .h files for all the classes and then start implementing the public methods within each class. On the weekend we will implement the private methods and core functionality of the game, making sure that a working game can be played. The week after we will add the peripheral functionalities like the hint function and start to rigorously test for edge cases and simultaneous matches. If things are going well and we are confident with our work, we will add extra features.

Schedule

Work to be Completed	**Completion Date (November 2014)**
Pick project, sketch rough UML, discuss design, assign responsibilities	Monday 17
Get UML done, answer questions, write rest of Plan of Attack, set up source control (GitHub)	Tues 18
Make all .h files	Wed 19
Implement public methods	Thurs 20
Break	Friday 21
All implementation except main	Saturday 22
All implementation and main	Sunday 23
Dealing with command interpreter and options	Monday 24
Testing, debugging, flex time	Tuesday 25
Testing, debugging, flex time	Wed 26
Testing, debugging, flex time	Thurs 27
Advanced Features	Friday 28
Advanced Features	Saturday 29
Submit project	Sunday 30

Responsibilities

Kevin Xue	Fred Cao
Plan of Attack	UML
Answer questions	Answer questions
Implement Square classes	Implement XWindows
Implement Main	Implement Board class
Implement TextDisplay	Implement Interpreter
Testing	Testing
Documentation	Documentation

Questions

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

By encapsulation. The main.cc file would not have diect access to the Square super and base classes, and thus can't use their constructors. It would have to call methods in the class that manipulates the board (in our case, the Interpreter class) in order to cause a change in the board's state (such as generating new squares). Since the code that actually generates new squares is private and thus encapsulated, we can ensure that only these kinds of squares can be generated.

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

By having a separate class for each type of square,we can quickly add new squares without major changes to the existing code. We would create a new <InsertType>Square class that would inherit from the Square base class. Only minor changes would need to made to logical code where the type of Square must be specified (hence not code which uses Square *).

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

To accommodate the possibility of introducing additional levels into the system, we can implement the visitor pattern on the Board class. For example, if we want to add level 3, we would create a class **BoardL3** - inheriting from **Board**, with its unique constructBoard() implementation. An “accept” method would be able to take in a BoardL3 pointer or reference as a parameter, and call **BoardL3** to construct the board. The separation of levels using the Object-Oriented Programming paradigm minimizes the need of recompilation - only need to recompile main.cc (since it would use the new BoardL3), boardl3.cc (new class), and board.cc (if the accept method was not implemented already).

In Board:

```
void accept(Board &board) {
     board.constructBoard();    // Similar to a generalized .visit()
call
}
```

```
In BoardL3:

class BoardL3 : public Board {

     // Private methods
     Public:
     ~BoardL3();
     void constructBoard() {
          // Implementation of generating a level 3 board
     }

};
```

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? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename swap s)?

We can accommodate the addition of new command names or changes to existing command names by using a map<string, void *> which would store the command name (the String) and the associated function (the Void *) to call for that command. We would have a default map based on the given initial command names given in the instructions. Should we wish to accommodate a new command, we would simply add the command name, function pair to the map. Should we wish to accommodate renaming an existing command name, we simply get the function (the value) associated with the old command (the key), remove the pair, and add a new pair to the map with the new command name and the same function.

For example, we have the command “scramble” associated with a function named scramble(). To change the name:

```
map<string, void *> dict = new map<string, void *>();

// Initialize dict with the default command name, function pairs

// To add a new command

dict.add(dict.end(), std::pair<string, void *>(newCmdName,
&newFunction));

// To change an existing command’s name

// Assume we’re given oldCmdName and newCmdName (as parameters for
example)
// Assume oldCmdName = “scramble”

void (*p)() = dict[oldCmdName];

dict.erase(oldCmdName)
dict.add(dict.end(), std::pair<string, void *>(newCmdName, p));
```