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

We approached this problem as "how to identify shapes". Any shape, with the exception of shapes that utilize diagonal matches, are composed of horizontal and vertical segments. In our solution, we "collect" the matched squares in two vectors, one composed all horizontally matched squares and one composed of all vertically matched squares. From there we can decide if a match took place (either vector has greater than 3 squares), to what kind of shape the match is (3 squares matched horizontally, and 3 squares matched vertically means an L match or a T match).

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

The decorator pattern was our first choice although we did not use the decorator pattern in the final design. Using the decorator pattern squares could inherit from Basic and special squares could have their own "clear" method which differ from type to type. However, in our solution we decided to clear the squares from the Board (i.e. squares do not have a clear method). Since we are working with only 4 kinds of special squares in our game, each special square effect is handled separately. What this means: upon clearing a Lateral square $\rightarrow$ clear the entire row, upon clearing an Unstable square $\rightarrow$ clear a 3×3 or 5×5 hole. If the number of special squares were bigger or their effects were complex, we would have used the decorator pattern, but for a small number of special squares with simple effects this solution made sense.

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

Generally speaking, the difference between levels are: what kind of squares are being produced and the rate at which a type of square is produced (probability distribution). With the exception of Level 0, which is read from sequence.txt, our Generator class takes care of this. When introducing a new level, the distribution of squares and any special rules such as "locked" squares, are defined in Generator, on a level by level basis. After that it is a matter of using Generator to produce all the squares required for a certain level ≥ 0.

How could you design your system toaccommodate 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 designed this game from the ground up to follow the MVC architecture. All commands are handled at the "controller" layer. The addition of a new command would not be difficult at all, as all that is required to do is handle the command at the controller and call the appropriate method in the "model" (Board). If the command requires a new method to be implemented on the models then they would be. However for something like "rename swap s" a method could be easily implemented at the controller level to rename a certain command that the controller handles. As for a command such as "cheat HereIsACheatCode", which perhaps doubles the scoring, a slight modification may have to be implemented at the model (Board) but handling the command itself is made very easy with the MVC architecture.