Project Plan: Quadris

Chuan Liu (c459liu), Hongqian Wu (h223Wu), Hanzhen Yang (h225Yang)

Overview

Our group choose to do the game Quadris. Based on the MVC pattern, we first divide the program into five main modules: Board, Block, Command, Level, and Display. And then we started to think about the design patterns that we are going to use to implement each module. We decide to use Factory Design Patter for Block and Level modules, Observer Design Pattern for Board and Display modules, and PImpl for Command module. This maximizes cohesion and minimize coupling. In our design, we will also try best to maintain encapsulation, and make the program easily extensible to new features.

Answer of Questions

Question 1: How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?

Answer:

In our design, we will do the following modification on the *Board, Block, Level* modules:

Board: we can add 2 more fields in *Board* to keep track of the special blocks and their existing time (turns), one called *specialCells* which is a *Vector<Vector<Cell>>*, and another called *lifetime*, a *Vector<Int>*, to record how many blocks have fallen.

If a special block falls (to determine if a block is special, the board will check the *isSpecial* field in Block), we will put this block into *specialCells*, and push a 0 in to lifetime, indicate that this special block has survived for 1 turn.

When an action (move, rotate, etc.) is complete and special block is not cleared, a loop will be run to find out if a block need to disappear, and pop out the corresponding cells in *specialCells*.

Block: in block we need a new field called *isSpecial*, which is a boolean, this is used to let *Board* know if the current block is special or not in order to determine behavior.

Level: we will let level to decide if a special block or a normal block will be generated. In each level there is a *generateBlock* function, it will call *BlockFactory::generateBlock* and pass some custom parameters to it according to the specification of the level. To make this feature confined in advanced levels, we just need to change the advanced levels code, pass one more parameter to *BlockFactory::generateBlock*.

Question 2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

Answer:

There will be an abstract class called *Level*, and multiple concrete classes *Level_x* inherited from *Level*. When adding a new additional level, a new *Level_x* class will be created and implemented. This reduces the require of recompilation.

To achieve the switching between different levels, a *LevelFactory* class with methods *levelup* and *leveldown* will be implemented. Both of the methods take a *Level* reference as parameter and returns a *Level* pointer. This also contributes to less-changing of the source code, since when

there is change on existing *level_x*, instead of changing the implementation of other *Level_x* (worse case, all other *Level_x*), only the *LevelFactory* need to be modified.

Question 3: 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 counterclockwise cc)? How might you support a “macro" language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

Answer:

There will be two types of command: basic command and alternative command. Basic commands are the commands that could be recognized by the entire system, in our case, these are the commands that provided from the assignment sheet (left, right, down, etc.). They are the only commands that could be directly recognized by the system, in other words, they are the name of the operations. While alternative commands are the user-defined commands, which issued after the program starts. These commands cannot be directly recognized by the system, but each of them have a reference to a unique basic command. When an alternative command is entered, it will be translated to a basic command for later use. Both basic and alternative command appears of the type string.

The program will have a concrete class called *Command*, it contains a field of type map, and methods *translate*, *add*, and *rename*. The map acts like a database of the commands, it stores all the <*alternative, basic*> command pairs the user can use. When user enters an alternative command, it will be automatically *translate* to its corresponding basic command if exist. When adding a new shortcut for a command, the *add* method will be called, and it simply inserts a <*alternative, basic*> pair to the map. When renaming a command, the *rename* method will modify the alternative part of the pair (Note that the map will be initialized with all possible <*basic, basic*> pairs at the beginning of the game. So when "rename counterclockwise cc", the user is actually changing the key of an initial pair). To support the "macro" language, instead of returning a string (the basic command), *translate* will return a vector of strings, and the system will recognize and perform the basic commands sequentially.

To achieve minimal changes to source and minimal recompilation, the fields will be separated from the class by using a pointer to implementation approach (PImpl Idiom). In addition, unless for adding new features (eg. new operations), the methods will stay consistent on the field structure. In such way, implementation of the methods does not need to be changed when changing the data fields.

(Note the Breakdown & Responsibilities is on the next page)

Breakdown & Responsibilities

Tasks:	Team member	Finish date
UML	All	Nov 26th
Board-related classes	Hanzhen Yang	Nov 28th
Block-related classes	Hongqian Wu	Nov 28th
Command-related classes	Chuan Liu	Nov 28th
TextDisplay classes	Hanzhen Yang	Nov 28th
First Round Integration and testing	All	Nov 28th - Nov 29th
Level-related classes	Chuan Liu	Dec 1st
GraphicsDisplay classes	Hongqian Wu	Dec 1st
Main function	All	Dec 1st
Second Round Integration and testing	All	Dec 1st - Dec 3nd
New features	All	If time allows
Revise UML and documentation	All	Dec 3rd