Biquadris

Contents

Section 1: Introduction

Changes to Scheduling (Chronological Program of Action)

Other challenges we faced:

1.
2.
3.

4. [illegible]
5. [illegible]

Section 2: Schedule Task Allocations & Assignments

The initial (longer) schedule is displayed on the due date 1 submission of a 5-page plan. Below, is a shortened, but more realistic schedule of how our workflow proceeded. In the column '(4) Change', we've described and justified scheduling changes (in green).

(1) Task	(2) Person	(3) Description/Notes	(4) Change	Due
Stage 1 - Planning				
[illegible]	[illegible]	[illegible]	**Ahead of Schedule**	Nov 18
[illegible]	[illegible]	[illegible]	**Ahead of Schedule**	Nov 19
[illegible]	[illegible]	[illegible]	**Good, but a bit behind due to design changes (after discussion**	Nov 20
Stage 2 - Project Essentials				
[illegible]	[illegible]	[illegible]	[illegible]	Nov 22
[illegible]	[illegible]	[illegible]	[illegible]	Nov 25
[illegible]	[illegible]	[illegible]	[illegible]	Nov 26
Stage 3 - Further Improvements (Non-Bonus)				
[illegible]	[illegible]	[illegible]	[illegible]	Nov 27

[illegible]	[illegible]		[illegible]	
[illegible]	[illegible]	[illegible]	[illegible]	Nov 27
[illegible]	[illegible]	[illegible]		Nov 27
[illegible]	[illegible]	[illegible]		Nov 28
Stage 4 - Bonus (Time-Permitting & Tentative)				
[illegible]	[illegible] Players	[illegible]	[illegible]	Nov 29

Section 3 & 4: Discussion of our Overall UML & Solid OO Design

In summary, our class hierarchy is as follows:

- [illegible]
- [illegible]
- [illegible]

[illegible] Decoupling Presentation and Control

[illegible]

[illegible]

deletion of a player entails the deletion of their own. Individual grids/boards (ie a set of cell classes) as well as MPN block classes.

General Description of UML and Code Workflow

1. A "GameManager" oversees two "Player" classes, and each player contains a full grid.
2. Grids comprise of a nested vector of "cell" classes. "Cell" class accurate and keeps info about each cell.
3. "Block" classes comprise of 4 cells, an abstract parent class is created, from which T block types inherit common methods and override polymorphic functionalities.
4. Using a similar model, an abstract parent level class is created, from which 5 pieces inherit and implement their own overridden specifications.

Cell Classes - As per our design, virtual functions were executed as private, while non-virtual functions were publicly exposed. The reason we adopted this approach was to separate the cells interface into two: (1) the client interface public non-virtual methods, and (2) the subclass interface which has any combination of virtual and non-virtual methods, enabling an achievement of high encapsulation.

Factory Method Pattern - As our game cannot anticipate the type of block objects being created beforehand, and rather they are randomly generated, the factory method pattern will be employed, enabling a localisation of logic to instantiate a complex object whose creation is delegated away from the actual implementation to a "factory" in higher levels, the "difficult" blocks will be generated with greater probability.

How Our Design Changed from Due Date 1

Removal of Observer Pattern

The biggest change we made from Due Date 1 was a reduction on the reliance on the Observer Pattern. While in the planning stage the Art Rewrite Question seemed very similar to this question, **we did not want to utilise a design pattern simply for the purposes of using a design pattern**. Hence, as we were able to more efficiently implement our code through different methods/toggles (simply having a grid, cell, and block class, and no subclasses/inheritance classes), we chose the more efficient implementation. Another crucial change was that each display class did not need to be as detailed / heavy like in Due Date 1.

Secondly, in Due Date 1 we did not anticipate having enough methods / helper functions in our classes. We needed to add numerous other methods. Specifically, in the block class, we added methods that can check the surrounding blocks. This enabled an analysis if the block could move left/right/down/rotate based on its neighboring cells.

Additionally, we realised that the shift coordinates relations can easily be implemented by adding the inverse of coordinates relation. Finally, we generally changed the algorithms of movements of blocks. Rather than moving the whole block, we decomposed the block each into its cells, made the block disappear from the grid, and then made it reappear in its new location.

Section 5: Resilience to Change:

Section 5.1: Resilience to Change: Overall Design

First, we'll discuss how our overall design is resilient to change (in Section 5.1). Then, in Section 5.2, to elaborate upon our code's resilience to change, we'll provide descriptive answers to the 4 Guiding Questions.

[illegible]

[illegible]

1. [illegible]
2. [illegible]
3. [illegible]
    a. [illegible]
4. [illegible]
    a. [illegible]
5. [illegible]
6. [illegible]

[illegible] ce, [illegible] **by** [illegible]

[illegible]

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible]
5. [illegible]
6. [illegible]
7. [illegible]
8. [illegible]

In order to *further* discuss how *resilient* our program is to change, below is a discussion of the 4 Essential Guiding Questions.

Section 5.2: Resilience to Change: Discussion of the 4 Essential Guiding Questions

Note, that while answers are segmented per each question, their discussions have been done in a fluid manner such that all object-oriented design elements are discussed throughout.

Question 1) (Abbreviated) "How could you design your system to allow generated blocks to disappear if not cleared before 10 blocks [falling afterwards]? Could the generation of such blocks be easily confined to more advanced levels?"

We designed our system in such a manner that:

1. [illegible]
2. [illegible]
3. [illegible]

With such a design, the implementation to this extension question becomes trivially easy. [illegible]

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

[illegible]

The process of creating a new level [illegible]

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible]

Our design accommodated such changes because:

1.
2.
3.
4.

Question 3) "How could you design your program to allow for multiple effects to applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?"

Question 4: (Abbreviated) "How could you [...] accommodate the addition of new command[s], or changes to [them], ... to support a user renam[ing] existing commands, [and to] support a "macro" language to give name[s] to [command] sequences?"

Renaming Commands

Finally, to enable renaming of commands, the main function should contain statements such as helper functions for **reusable code**. Additionally, when if conditions are constructed **rather than them involving strings, they should instead utilize a string stored in a variable**. For example, instead of the if condition being `if (command == "go")`, it should be `if (command == command_for_go)` where `command_for_go` is equivalently initialized as `string command_for_go = "go"`. These two alternatives achieve the exact same result, but the second method enables a renaming option, as demonstrated below.

A rename helper function is created taking two parameters, `void rename(string original, const& string new)`. As the first parameter is a pointer, the rename function changes `original` to instead point to `new`, and hence the `new` string works perfectly in calling the commands. **Edge Cases** Note though, that the helper function which calculates shortcuts for commands (which have to be done) on, to re-calculate all possible command name shortcuts.

Command Sequences and Synonyms

By extension, to accept a list of synonymous commands (e.g. `u`, `up`), rather than just the function to handle another one/twelve relations, simply store a list of all possible commands in an array. Then, a helper function is created that measures the smallest possible command word, that avoid ambiguities.

Now a list of all synonyms, demonstrated by update, as a string field are stored. Then, we created an association method that alternatively compares input with each synonym word by word. **Even when new commands are introduced, our described design above allows a recalculation of all synonyms of commands, thus accommodating any changes.**

Similarly, implementing a macro language to name command sequences can be easily implemented given our aforementioned abstractions which **make our main and command interpreter re-usable**. In essence, implementing command sequences are nothing more than re-iterating through our commands interpreter. Since this has been abstracted, a simple call within an iteration loop is all that is required.

Section 6: Extra Credit Features

etc). ***However, as mentioned earlier, our design easily and readily accommodates them***.

Section 7: Final Questions

(1) What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

[illegible]

(2) What would you have done differently if you had the chance to start over?

[illegible]

Section 8: Concluding Remarks

[illegible]