CS 246 Plan of Attack - CC3k

Lidiya Murakhovs'ka, Daniel Podlovics, Alex Tomala

Planning

The schedule is listed below, along with a description of what each section entails.

Section	Dependencies	Assigned To	Expected Completion Date
Base Setup	None	Alex	Nov 25
Add Headers	Base Setup	Alex	Nov 26
Add Messaging	Base Setup	Daniel	Nov 26
Add Game/Grid/Cell classes	Add Headers, Add Messaging	Alex	Nov 27
Add Entity and Character classes	Add Headers, Add Messaging	Lidiya	Nov 26
Implement Grid Loading	Add Headers, Add Messaging	Daniel	Nov 27
Implement Basic Controller	Add Headers, Add Messaging	Lidiya	Nov 27
Implement Basic View	Add Headers, Add Messaging	Daniel	Nov 27
Milestone: Get Basic Program running	Everything Above	Alex	Nov 28
Add Decorator classes	Milestone	Lidiya	Nov 29
Add Potion classes	Add Decorator classes	Lidiya	Nov 29
Add Treasure classes	Milestone	Alex	Nov 29
Add Other Characters	Milestone	Daniel	Nov 29
Implement Character AI	Add Treasure classes, Add Other Characters	Daniel	Nov 30

Implement Preset Entity Loading	Add Potion classes, Add Treasure classes, Add Other Characters	Lidiya	Nov 30
Implement Custom Entity Generation	Add Potion classes, Add Treasure classes, Add Other Characters	Alex	Nov 30
Implement Full Controller	Add Potion classes, Add Treasure classes, Add Other Characters, Implement Character AI	Daniel	Dec 1
Implement Full View	Add Potion classes, Add Treasure classes, Add Other Characters, Implement Full Controller	Lidiya	Dec 1
Code Review	Everything Above	Everyone	Dec 2

Section Description

Base Setup

Set up the git repository and include the basic folder structure. Add .gitignore file along with any other metaconfiguration files. Ensure that a basic Makefile exists and that the project successfully compiles with a simple main.cc file. Set up testing on Gitlab to automatically compile the project.

Add Headers

Add header files for classes such as Character, so some of the code can be compiled successfully even if other portions are not implemented yet. The header files do not need to include all of the private variables as that is not important to the interface.

Add Messaging

Write the struct definitions for messages. Add the code for the Observer/Subject classes. The Observer code should be calling the sendNotificationTo() function in message to call the notify() function in Subject. Messaging is used in many objects and depends on nothing, so it makes sense to start with it.

Add Game/Grid/Cell classes

Write the basic code for the Game/Grid/Cell classes. Most of the functions should be able to be written by referring to the UML diagram when interfacing to other classes. It should be possible to compile the class due to the header files, but running the code is not possible.

Add Entity and Character classes

Write the code for the Entity and Character classes. Also write the code for the default race (Shade).

Implement Map Loading

Write some of the code for the GridInit class and its subclasses. Specifically, make sure that the createGrid() function is implemented. This will be useful for getting a basic program to run.

Implement Basic Controller

Write some of the code for the Controller class and its subclass. We want the Controller to be capable of basic game initialization and to handle basic player interaction (ignore AI). This is also useful for getting a basic program to run.

Implement Basic View

Write some of the code for the View class and its subclass. The View should be able to display the map and handle player display/movement (this information is communicated through messages). This class can be iteratively worked on afterwards to display more information as time progresses.

Milestone: Get basic program running

At this point, the program should be able to run a basic version of CC3k that allows the player to move around the Grid.

Add Decorator classes

Write the code for the Decorator classes.

Add Potion classes

Write the code for the Potion classes.

Add Treasure classes

Write the code for the Treasure classes.

Add Other Characters

Add the other characters other then the shade into the Game.

Implement Character AI

Write the code to control the enemies in the Game. This is abstracted away in the AI classes.

Implement Preset Entity Loading

Write the code for GridInitPreset. This one relies on no custom entity generation and uses a configuration file instead for determining where entities are located. This may be useful to work on earlier as its great for testing purposes.

Implement Random Entity Generation

Write the code for random entity generation. Specifically, make sure that the createEntities() function is implemented in GridInitRandomGeneration.

Implement Full Controller

Write all the code in the controller. This should use the AI class to control the enemies and have all the required commands for the human. Error handling should be done and game progression code (next level, win, lose) should be implemented.

Implement Full View

Write all the code in the View. Game actions should be printed and everything on the Grid should be properly displayed.

Code Review

Review all the code and identify potential bugs/optimizations.

Bonus work

Look into cool ways to expand this project. This is left as an exercise to the reader.

Answered Questions

1. How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional races?

   The system can be designed so that each race is sub-classed underneath a Character class and all of the methods specific to that character are defined within the Character class. Generating a race involves just calling its constructor. Adding new races simply involves making a new subclass of the Character class and overriding any methods and default variables for that specific new race.

2. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

   Our system is implemented using a MVC architecture, where the Characters are defined to be part of the model. The model does not distinguish between enemies and players, which is a role the controller fulfils instead. This distinction simplifies the class inheritance hierarchy (since all races inherit from same Character class) and makes it possible to implement features like Multiplayer and more advanced AI by simply changing the controller implementation. For generation, the player is generated separately from the enemies as the controller needs to keep a copy of who the player is (and stair generation depends on players location). The enemies can probably be generated using the same function in Q5.

3. How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

   The various abilities for the enemy and player characters are implemented as virtual method overrides. This method is fairly flexible and allows new characters with different abilities to be added in the future.

4. The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by weighing the advantages/disadvantages of the two patterns.

   While the Strategy pattern is easier to understand and implement, it simply allows a calling function to be replaced with a different one. This would be fine if there was one possible effect at a time, but a player can take multiple potions (and thus have multiple effects at the same time). This is not easy to work with using a Strategy pattern.
   The Decorator pattern is a bit harder to understand and implementing it takes more time, but it allows functions to effectively stack on top of each other. This means we can handle multiple effects by stacking multiple decorators on top of the character. The idea of stacking effects seems far more intuitive in the Decorator in comparison to trying to represent multiple effects using one function in the Strategy pattern.

5. How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code?

   This can be done using a function that takes in an array of Entity objects (which Treasure/Potions inherit from), an array of rooms, and the number of items to generate. This function can then randomly place copies of the items in the array (randomly selected) on the map. This also has an added benefit of working for enemy generation.