CS 246 Final Document - CC3k

Lidiya Murakhovs'ka, Daniel Podlovics, Alex Tomala

Overview

From a high-level view, the CC3k project is implemented using a MVC architecture along with a messaging system to facilitate communication between various subjects and observers. The model holds the map and all of the entities in the game. It also handles transitions in the model through various helper methods. The controller is used to handle user input, manage the view/model objects, and handle AI. The view handles outputting the model and primarily receives information through the messaging system.

Design

Model

The model is represented using a Game object, which contains an instance of the Grid class to store the map. The Grid class contains a 2d vector of Cell objects, which contains information about each individual cell. The Cell objects have a cell type associated to it (e.g. Wall, Floor, Stairs), as well as a vector of Entity objects to describe what is on the cell. Entity objects are any object that can be placed on the Map and that includes Characters, Potions, and Treasures. The Entity class itself is abstract and has no meaning unless the methods associated with it are overridden.

Grid generation and Entity generation are handled using an object that inherits from GridInit abstract class, which uses the Factory pattern. During game setup (which occurs in controller), the object is passed to the Game object and then is called as needed. This method makes Grid/Entity generation modular and also prevents the controller from having to keep track of the GridInit object.

By default, there are two GridInit objects implemented. One randomly generates entities (GridInitRandomGen), while the other one generates entities as specified on the provided map file (GridInitPreset).

Characters are handled using the Character abstract class, which inherits from Entity. Each specific race is handled by sub-classing the Character class. Note that there is no Enemy or Player classes because the model does not distinguish between enemies or players races at all. Rather each character is assigned a faction, which the controller then uses to handle interactions. The merits of this is discussed later.

Each Character has a map associated with it that describes what Entities the character knows about. By default, the Character knows nothing [1] and can gain knowledge by using potions. This play no role in any model interaction (in the games base form), but it is used by the view to determine what messages should be printed out.

Character combat is handled using double dispatch. Each character has a overridden attack method which calls the getAttackedBy method for the defending enemy. This method allows character to character overrides to be easily added (e.g. Vampire losing health when only attacking Dwarfs).

The character stats are stored within a StatsContainer object, which seems to be useless as it just returns the Stats struct within itself, but it does serve a valuable purpose. The Modifier abstract class inherits from the StatsContainer class and implements the decorator pattern. This means that modification of a characters stats can be done in a temporary manner by overlaying a subclass of the Modifier object onto the characters StatsContainer object.

When the built-in Potions are used, they either decrease/increase the characters health or they change the characters stats. The first effect is achieved by simply changing the character health, while the second one overlays an AdditiveModifier object (inherits from Modifier) to the characters StatsContainer object. These operations occur in the itemUsedBy function that the Potion classes inherit from the Item class.

Treasure is handled by overriding the walkedOnBy function that is provided from the Entity class. In most cases, the treasure simply removes itself from the board (by sending a message requesting removal to the Grid object) and then increments the characters gold by the amount specified in the object. This differs in the case of the Dragon Hoard, which has a weak pointer to the associated Dragon. The gold is only picked up in the case where the associated Dragon no longer exist.

View

The view is represented by an abstract View class, which is inherited by a specific implementation. Most of the information the view receives comes through messages that are generated by either the controller or model. Since the model is oblivious to who the player is, the view implementation uses a pointer to the player to filter out all of the undesirable messages to pass onto the log such as enemy movement.

Since the model tries to remain decoupled entirely from the view, all of the strings used in the action log (other then Info/Debug messages) must be generated by the view. This gives the view more flexibility on how the action log is formatted.

[1] It may be wise to consider renaming the Character class to JonSnow

Controller

The controller is represented by an abstract Controller class, which is inherited by a specific implementation. It handles initializing the model and view as well as basic game flow. The controller implementation keeps track of who the player is and handles user commands.

The AI is handled in the controller through the strategy pattern. The function that takes in a character and gives it actions to do is in a separate object that inherits from the CharacterAI abstract class.

The design of the controller is fairly open ended since there are not many constraints attached to the Controller class.

Resilience to Change

Model

The Entity system in the model is designed to be easily extended with new items. Each cell in the grid simply holds an array of entities stored on that tile so it is unlikely that the Game/Grid/Cell classes will have to be modified (unless it adds some new special interaction like ranged attack) to accommodate the new entity. All that is usually required is to write a new subclass of Entity or one of its subclasses, update the entity generation to include the new entity, and update the view to display the new entity.

One simple example would be to add a trap underneath some random treasure (evil). This can be done by adding a Trap class inheriting from Entity, which overrides the lookedOnBy function to make the Trap unobservable, overrides the canWalkOn function so the player can walk over it, and overrides the walkedOnBy function to lower the HP of the player when they step on it. Then the generating code can be updated to sometimes add a trap entity before a gold entity to a cell.

The ability to put multiple entities on the same cell allows features like the one above to be added in a very generic matter. An alternative less generic manner may have been necessary if only one entity can occupy a cell at a time. For example, having a CoinTrap class which damages the player and gives coins at the same time. This method doesn't scale well if traps are to be placed underneath another entity like a potion.

Generating new races is something that is very likely to be done, so it is designed to not be difficult to do. It involves making a subclass of Character, doing the generation/view code updates, and adding a new function to the Character class (getAttackedBy(newCharacter)). The need to write new code in the Character class is unfortunate, but it is a requirement for double dispatch to work with the attack function. Adding abilities to the new race tends to be fairly easy as there are a variety of functions that can be overridden. Attacks from one specific race to another specific race can be easily overridden thanks to the double dispatch approach of handling attacks.

Implementing most new potions/treasures shouldn't be fairly difficult to do since most cases just involves changing the values to the super class constructor for Potion or Treasure. The potion may want to use a more complicated effect such as doubling the players attack instead of just increasing it by x amount. This can be handled by making a new subclass of Modifier that is multiplicative instead of additive and by using that in the itemUsedBy function for the new potion.

Grid and entity generation is fairly modular as it is using the factory pattern. Making a new generator just involves making a sub class of GridInit and overriding the two functions which handle generation.

View

The view is fairly flexible in how it can be designed. There are just two functions that are required to be implemented and the rest of the code is messages the controller and model send out. The controller and view are somewhat coupled together as the output may differ depending on what type of game the controller is playing (e.g. Multi-player).

Controller

Due to the way the view and model were made, the controller is insanely flexible in how it can be designed. The controller has control over every other part of the system (including instantiation) and has almost no requirements to satisfy from the Controller abstract class. Things like victory conditions and how the AI interacts with others are not predefined, so lots of interesting things can be done.

One of the best features with the model is that the player and enemies are not treated separately at all. This means that the controller is free to interpret the relation of various characters to each other in anyway it wishes. The model can assist with this as it comes with a built in faction system that the controller can use. Each character is assigned a faction id and the Game object keeps a map of current relations between various factions. Nothing in the model changes these values as faction relations is delegated to the AI and controller.

These ambiguities give the controller great power when it comes to making changes to the game type. One example would be a single floor Battle Royale, where 20 or so characters are all hostile to each other and the player controls one of them. The winner is the last person standing. This can be implemented by changing the victory conditions and changing the AI so everyone attacks everyone else.

But we can still make changes to this. It is possible to make the game have no players at all and have it be an AI only battle. While it may be boring to watch with the current AI implemented in the game, there is always the possibility of adding more complicated AI. The Battle Royale setting provides a nice environment to try out reinforcement learning algorithms which can lead to some rather interesting AI that actually use the abilities they have to their advantage.

Or on the other end, it is possible to make the game only have players in it and make it multiplayer. This can be done in various ways that can be slightly complicated (since can involve network stuff), but can use some cool ideas such as allocating a unique view for each player (that only outputs certain information to each player).

There are so many cool things that can be done with the controller and the author regrets that they don't have much time to actually implement some of these ideas.

As mentioned before, the AI is handled through the strategy pattern. This means that changing the AI used by the controller is a matter of swapping which AI class is instantiated.

Answer to Questions

Most of the answers were updated slightly to make things easier to understand.

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 class constructor.

   Adding new races involves making a new subclass of the Character class and overriding any methods and default variables for that specific new race, as well as adding some additional code for double dispatch in the Character class. It also involves having to modify the view to display the new race (the model has no assumptions on how the Characters will be displayed) and potentially modifying the controller/AI if the player can play as that race or if the race has special AI behaviour.

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

   In our design, the model does not distinguish between enemies and players, which is a role the controller/AI 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 are generated using the same function described 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 method overrides and through a list of basic character stats. This method is fairly flexible and allows new characters with different abilities to be added in the future. The most complicated abilities primarily dealt with attacking, which was handled using the visitor-like pattern.

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 tends to be easier to understand and implement, it simply allows a 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 tends to be 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. As a result, we used the decorator 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 an array of Entity objects (which Treasure/Potions inherit from), an array of rooms (which is an array of cells in the room), and the number of items to generate (which we denote by n). The function can then randomly take one of the arrays of Entity objects and place a copy of the objects in a random cell in one of the rooms (and area surrounding cell if more then one entity in the array). This method is rather extensible and works with the Dragon Hoard/Dragon generation.

   This answer was updated to be more clear and to mention an array of an array of Entity objects instead of an array of Entity objects.

Extra Credit Features

All of the pointers have been implemented using smart pointers. There is no memory allocation/deallocation using new/delete.

Final Questions

6. 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?
7. What would you have done differently if you had the chance to start over?