Overview

Careful consideration was put in to how our game world would be modelled, and how our game loop would function. The core functionality of CC3k will reside in a “Game” class, which will handle initializing, starting and ending games, as well as updating and displaying the game state. Player input will be handled by subclassing a separate “Input” class, which will interact with the game to update its state. Similarly, we will also simulate enemy input in a separate subclass, handling move and attack commands appropriately.

In terms of how the actual game object classes will be constructed, we will model Characters, Items, Floors, Chambers, and Cells, which will have appropriate subclasses and design patterns associated with them, which we will talk about at the end of this outline. It’s hard to tell how our final implementation will look, but the enclosed UML diagram is a hopeful approximation.

Questions and Answers

Player Character

Question: How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

Implementing the Factory Design Pattern would be ideal in this case. Rather than using cumbersome if/else statements in multiple parts of our code, a “PlayerFactory” method could be called, where an appropriate indicator is passed in (such as character representing race) to dynamically generate the appropriate race. This will make adding additional classes incredibly easy as only the “PlayerFactory” class will have to be modified, besides adding a new race class.

Enemies

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

Like our “Player” class, we plan to dynamically generate different enemies the same way; using the Factory Design Pattern. Again, this makes is easy to generate enemies, because whenever we need one, we can simply call for a random enemy from our factory. The implementation of the factory itself would be different than the previous example, however since we require random generation, rather than generation based on an indicator (like a character representing enemy type).

Question: How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

In our implementation, we plan for each enemy to interact with the game via a subclassed “Input” class. In order to potentially implement special abilities, we can alter or add specific input subclasses so that enemy actions are handled differently (i.e. instead of just lowering the PC's hp, it will also activate its special ability; so if a goblin attacks, the PC will lose hp and some gold.)

Items

Question: What design pattern could you use to model the effects of temporary potions (Wound/Boost/Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor?

The Visitor Design Pattern could be used in this scenario, with Potions visiting the Player, modifying the Player and his stats appropriately based on the effects of the Potion. When a new Floor is reached the Player will simply be reset to his "base stats" which will be defined upon race creation.

Treasure

Question: 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?

Like player and enemy generation, implementing the Factory Design Pattern would be ideal. An "ItemFactory" method could be called, where an appropriate indicator is passed in (such as a character representing item type). Alternatively, in the case of random item generation our factory implementation would be different, containing code that randomly generates an item.

Estimated Completion Dates and Assigned Tasks

Our goal is to implement the full game as well as at least one extra feature. We have a rough implementation of our game world so far and have started coding in the order of importance (reading in and printing a floor, general purpose character/enemy/items, movement). Below is a breakdown of the timeline of our project.

Friday, March 28th

Basic implementation of the mechanics of CC3k should be complete. Tyler will implement the game display, general classes for the game and gameflow commands like player movement. Steve will model and begin to implement the classes for items, enemies, and the player.

Monday, March 31st

A basic version of CC3k should be working at this point. That is, player movement and attacking will be completed, enemies will move and attack appropriately, and item use will be implemented. Obvious bugs will be ironed out and some testing of the game will be done as well. Tyler will implement player and enemy attack and movement, while Steve will implement item use, fix bugs, and test our implementation so far.

Wednesday, April 2nd

Full game functionality should be completed at this date. That is, floors are randomly generated properly, all enemies behave appropriately based on their type, game state is ended and updated at the correct times, all items, races, and enemies are modelled and in the game, and there are no obvious bugs. Tyler will finish modelling and implementing the remaining races and enemies as well as their behaviours if necessary, and Steve will implement random floor generation, handle how the game state changes, and fixes the remaining outstanding bugs.

Thursday, April 3rd

By now we should be refining features of our program to be more robust. That is, at this point we should be more focused on refactoring our code to improve efficiency, improve readability, and improve coupling and cohesion. Furthermore, we could potentially start adding in extra features provided the required game functionality is entirely complete.
Also, our second project write-up should be done on this date. This will include everything that differs from our original plan, the questions we are required to answer for this project, and our general thoughts on this project as outlined in the project guidelines.

Friday, April 4th

Optimistically our code should be complete, fairly robust, and readable by this point, along with at least one extra feature. We will do final rounds of testing, re-factoring, and if time permits complete more extra features.

Use of Design Patterns

Singleton

Our “Game” class and “Player” character class will both use the Singleton Design Pattern, since you should be able to have at most one instance of them.

Observer

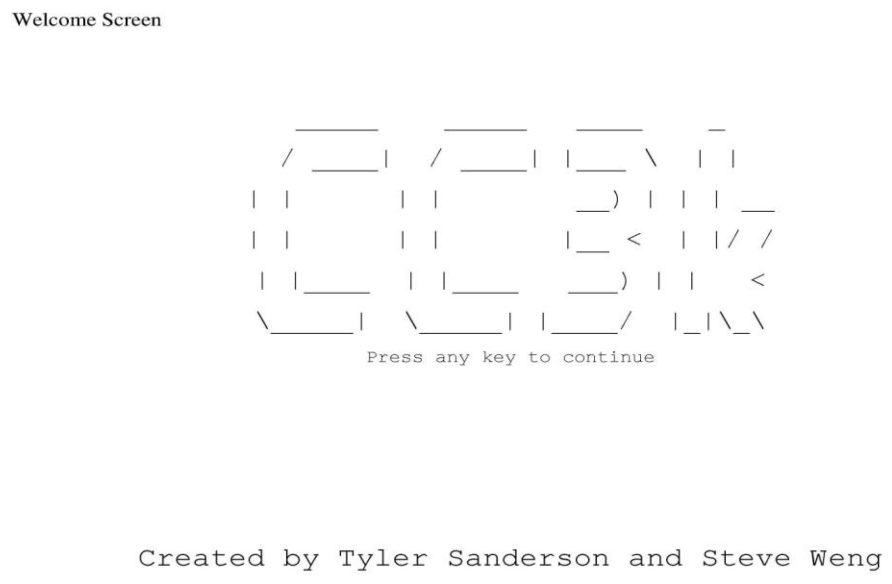
Our “TextDisplay” class will use the observer pattern along with our “Cell” class. Our Text Display will observe every Cell, where each Cell is a subject which will notify our Text Display to update accordingly whenever the Cell’s state changes.

Factory

The “Enemy”, “Player”, and “Item” classes will use the Factory Design Pattern to dynamically generate appropriate objects based the current game state (i.e. random generation of items and enemies for floors or layout parsing).

Extras

Game Screen "Sketches"



Welcome Screen

```
  _____    _____   ____    _
 / ____|  / ____| |___ \  | |
| |      | |        __) | | | __
| |      | |       |__ <  | |/ /
| |____  | |____   ___) | |   <
 \_____|  \_____| |____/  |_|\_\
        Press any key to continue



    Created by Tyler Sanderson and Steve Weng
```

Main Screen

```
   _____     _____     ____     _
  / ____|   / ____|   |___ \   | |
 | |       | |          __) |  | |  | __
 | |       | |         |__ <   | |/ /
 | |____   | |____    ___) |  | |    <
  \_____|   \_____|  |____/   |_|\_\


      (S)tart           (R)ace Selection

      (I)nfo                 (Q)uit
```

Race Selection Screen

	(H)uman	(D)warf	(E)lf	(O)rc
HP	140	100	140	180
ATK	20	20	30	30
DEF	20	30	10	25
Ability	None	Doubled Gold	All Potions Positive	Halved Gold