Design Document
for Chamber Crawler 3000

Alya Berciu, Aidan Day, Bharatt Kukreja

Schedule

by July 17th: Implement interfaces, abstract classes, and one concrete implementation of each set subclasses. Implement a basic UI for testing purposes. Minimal functionality in classes like Cell and GameMap will allow testing.

by July 20th: Fully implement all classes. At this stage, our program should run.
Deviation: This stage took us longer than expected, and we finished it July 21st.

by July 24th: Refine implementation and test. If possible, implement DLC.
Deviation: We did finish minimal deviation of special abilities for enemies.

Group Member	Responsibilities
Alya	GameMap, Cell
Aidan	Sprite
Bharatt	GameHandler, UI, TextUI, AI

Sprite

Sprite exists because Cells need a pointer to things, exactly one of which can be in that Cell.

Sprite has a pure virtual dtor and a pure virtual getType() method. getType() returns a SpriteType, which is an enum class. getType() is overridden in all concrete subclasses.

Stationary, Stairs, Item & Gold

Stationary, a subclass of Sprite, is an abstract superclass to Stairs and Item. Unlike Character, the other direct subclass of Sprite, Stationary is not moved by the GameMap.

Stairs are what the player must interact with to reach the next floor. Stairs do not have any fields or large methods; instead, the GameMap just needs to know its type to change floors when the player tries to move to its tile.

Item is an abstract superclass with method use(PC &). Its two subclasses, Gold and Potion, have different uses, so use() is pure virtual. After an Item calls use(), GameMap calls its dtor.

Gold, a subclass of Item, overrides use() to add its value to a PC's collection. It's called by GameMap when the PC enters the Cell containing Gold.

Potions

Potion has base attributes value and positive. Value is the effect on a stat, positive determines whether that change is positive or negative. Potion doesn't implement use(). Its three subclasses AtkPot, DefPot, and HPPot implement it to affect the right stat.

Character

Character is the class from which Player Characters (PCs) and enemies (NPCs) inherit.

Characters contain four fields: atk (attack), def (defence), hp (current health), and maxHP (starting health). In the base game, only player characters strictly need a maxHP value, but we could have included DLC to give Trolls health regeneration; we'd need to cap that.

Since those fields are protected (but not private, since some overrides from Character subclasses need full access), the class has getters. It doesn't have setters, though. Instead, it has methods to change its attributes. Each attribute has a minimum value (0 for hp, 1 for atk and def). All changes made to a stat are relative to their original values, so while setters are sort of like operator=(), changers are sort of like operator+=().

Character also implements of getHit(int), which handles damage formulas.

character.h also provides changeAttr(unsigned int &, const int, const int), which it uses in its change functions. This is primarily used for PC, which has a changeGold(int) method.

NPC

NPC is a subclass of Character that introduces one fields: goldDropped. It has a getter but its setter is protected since only subclasses should use it, only in operator=().

NPC hostility is determined by that race's variable. Dragons are technically only hostile when the character is in a one-block radius, but if they are always immobile, their behavior is the same.

NPC also has a hit(PC &) method that randomly generates a boolean to determine whether to call its target's getHit() method. Note that hit() returns a pair of a bool and an int so the UI knows whether it missed and, if not, how hard it hit.

PC

PC is the superclass to the subclass that is the race of the player's character in the game. As such, it has a lot of methods, many of them virtual so races can override them as needed.

resetPotions() resets stat changes to atk, def by resetting them to PC's added fields: baseAtk and baseDef, which are never changed from their default. If there were another way to modify stats, we would instead store the changes in an int per stat changed, then reverse the change at the end of the floor and reset the int to 0.

changeGold(const int) works like the other changeAttr()-derived functions. It is not overridden by Dwarf or Orc, who instead override getGold(). We did it this way because Orcs cannot have .5 of a gold, and if we took the quotient when we added gold of value 1, Orc would end up adding 0 to its gold.

score() is a method that returns a PC's score, called by GameMap on the player when they finish the game. Humans override this.

We did not add a method drink() to handle the various potions because we'd be creating a visitor pattern that would just call use() on the Potion that would just call PC method. GameMap instead calls use() on the item, who modify the PC. Elves are a special case in use() for Potions.

GameHandler

GameHandler is the top class of CC3k. It has a GameMap that is the world the user interacts with, a reference to a UI that is the way the players sees the world, and a pointer to the PC. GameHandler's only method is play(), in which the whole game runs.

GameMap

GameMap contains a grid of Cells. It handles interaction between Sprites. Its main fields are: 1) grid, a vector of vectors of Cells, 2) player_location, a pair of ints as coordinates, and 3) enemy_locations, a vector of pairs of ints as coordinates for the enemies. It also encapsulates the AI, that moves the enemies.

setUpMap() makes the Cells in the map randomly, but setUpMap(vector<vector<CellType>>) reads the map from a grid of CellTypes, which is the enum class for Cell.

GameMap also has methods to return the floor number, and whether the game is won.

Since GameMap is the only thing in the program that knows of the space around the Sprites, it is the class that controls the turns of the PC and NPCs.

GameMap contains the method clear(), which resets the map before the next floor. It also has initialize() which sets up a map for the first floor.

UI & TextUI

UI is an abstract class to the possible UIs CC3k can have. Its concrete subclasses interpret the GameMap and display commands and feedback for actions. All methods in UI are pure virtual so it does not restrict the type of UI used.

TextUI is our concrete UI in this implementation. It displays the GameMap as lines of chars, input as commands, and displays actions as English sentences.

Cell

Cells are the individual parts of the grid. Cells have four types, represented by a CellType, the enum class for Cells. A Cell can be a floor, door, wall, passage, or space. Note that only floors, doors, and passages can contain a Sprites.

Cells also contain their coordinates in the GameMap so that they can return their coordinates to the GameMap when it needs information on the Sprite it contains.

Cell also runs an observer pattern where it notifies and watches the Cells adjacent to it in the GameMap. This allows easy checking for interactions the player will try to make; it's easy for NPCs to see whether the PC is in a one-block radius, and it's easy for the PC to pick up Gold or use Potions.

AI

AI is a class that generates moves for NPCs. It contains a vector of pairs of their locations in the GameMap and the grid of Cells so it doesn't try to move NPCs to where they shouldn't be (walls, etc.).

If we had more time, we would have made AI an abstract superclass of an AI implementation that could easily be changed (for example, giving enemies actual intelligence).

Questions, Answered & Revised

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

A: All races inherit from the PC (Player Character) superclass, containing methods and fields that can be overridden to give each race special properties. This model is easy to extend -- all we needed to add is a subclass that overrides PC's methods as needed. Races starting stats can easily be set by a default constructor.

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

A: GameMap's populate() method spawns enemies. populate() is in turn called by init(). We use a map whose keys are integers (with value SpriteType for the enemies) to store the probabilities of spawning enemies. The map contains 18 keys (numbers from 1 to 18), with the number of values of an enemy depending on its probability. The map has type map<int,SpriteType>.

For example, 4 of the 18 keys map to Werewolf, since its spawn distribution is 2/9.

In the map, we randomly generate an integer between 1 and 18, then spawn the enemy corresponding to that key.

It's different from generating a player character because the same PC is generated each map -- only potions need to be reset. Placing the PC is done the same way as the enemies, since they both follow the same restrictions, but the PC is generated first.

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

A: We have a Character class which is a base class for a NPC (Non Playable Character) class which is a superclass to enemies. The Character class contains all the methods that the enemies would need in a combat. By overriding those methods in the particular enemy class, we could implement special abilities for enemies.

For example, Goblin could steal gold by overriding NPC's toHit() method. Vampire could steal health by overriding NPC's hit method.

To regenerate health at the end of each turn, Troll would need to override all functions that count as a turn to call changeHP(n) where n is the amount of health re-added per turn. Note that Character, the superclass to NPC, contains the field maxHP. Since move(const Character &) is a GameMap method, we can just write a move(const Troll &) that does the same thing (storing shared code in a helper), but regenerating at the end.

Q: 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?

A: We added baseAtk and baseDef fields to the PC class. At the start of a new floor, Atk and Def are reset to baseAtk and baseDef.

We would have used a decorator pattern, but this is a much simpler solution that more intuitively represents the effects of potions. It's also easier to model their effect on Elf, a subclass of PC.

Q: 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?

A: The Gold and Potion subclasses both extend Item, since both override the use method, called by the PC class. All Items interact the same way with GameMap -- stationary, destroyed on use.

Potion also has subclasses that represent the different kinds of potions that are present on the map, such that most of the code for each different kind of potion can be reused. The GameMap class generates the necessary Potion and Treasure instances when the populate() method is called, spawning them randomly in empty cells on the game grid.

Q: What lessons did this project teach us about developing software in teams?

A: Never procrastinate. Because our code is coupled, we depended on each other to write or debug something. We all procrastinated at some points in this project, which held up the rest of the group. If we'd done more on the first weekend specifically, we would have easily had time to do significant DLC.

It also taught us that it's super important for a file to compile on its own because when we met up, we'd inevitably find linker errors that would take hours to fix. To generalize, it's important to finish all you can on your own so you don't waste everyone's time.

When we debugged, we found that we did not have sufficient documentation in some .cc files, so it was inefficient for anyone but the author to fix bugs. Once the design document exists, it's important to document .h and .cc files such that it's clear from either the document or the code.

Q: What would we have done differently if we had the chance to start over?

A: We should have kept better track of the changes we would have to make in our UML and design doc, since we did change some of our plans.

Weekends are very useful because our group was available, unlike weekdays when we had varying schedules. Thus, we should have used the first weekend much more.