CS246 – Final Document

James Treap	jtreap
Robin Li	x977li
Stefan Min	j28min

Overview

Our project is broken down into the following components:

- **Cell:** This represents a single tile on our board. Each tile has an x and y value, a character value (e.g ., \, |, +, -), and a pointer to a creature (see below for more information).
- **Creature:** This is an abstract superclass, which every creature and item in the game inherits from. The main two abstract subclasses for it were player and enemy.
    - **Player:** has **Human**, **Dwarf**, **Elves**, & **Orc.**
    - **Enemy:** has **Troll, Werewolf**, **Vampire**, **Goblin**, **Merchant**, **Phoenix**, and **Dragon.**
    - Misc items (directly inherit): **Barriersuit**, **Compass**, **Gold**, **Potion.**
- **Ladder:** Represents a ladder.
- **Board:** This class contains a 2D vector of Cells. It is responsible for reading in the board's layout from a .txt file, initializing enemies at specific locations, moving both the player and enemies around, and picking up items that the player interacts with as they move.
- **Combat:** this class is a friend class of Board which handles the player getting attacked by enemies. Here we attempted to use the MVC (Model-View-Controller) method by making it a controller which automatically handles these interactions for the board class.
- **Main:** handles getting player input and calls upon the respective commands in the board and combat classes. It also handles the core logic of the game (what occurs and when), and prints the board after every iteration, calling upon the overloaded << operator in the board class.

Updated UML

See uml.pdf for our UML. Notable differences between our original UML and our final version:

- We no longer have a **controller** class. Instead, most of our logic is handled by main, which calls upon public methods that we moved from our hypothetical controller class into the board class.
    - The combat class now has an 'owns-a' relationship with the board class.
- Potions, the compass, the barrier suit, and gold are now all subclasses of the creature.
- We no longer have subclasses for the potion class.

Design Questions

For this project, we focused heavily on encapsulation, inheritance, and dynamic generation of our program. Our game has a 2D vector of 'Cells' which have a pointer to a specific creature. To 'move' a specific creature, we chose a direction they want to move in, determined if that movement option was valid (i.e does the cell point to a nullptr and is not an edge tile or out-of-bounds), and if so, swap the pointers between those two cells.

As long as a new object inherits from the creature class, we can use the std::swap command to swap the pointers between two cells, effectively mimicking movement across the board. Thus, our design focuses on an abstract creature class, from which we have numerous concrete subclasses that focus on initializing different types of creatures. Changes to one subclass of creatures would not impact the other subclasses.

We also used the MVC (model-view-controller) design principle for handling our combat interactions. As our program stood, our board class had multiple reasons to change, whether it be relaying information to the user or changing a game state. We wished to decouple the interface of the board by using the single responsibility principle, where the board would manage both its state/data and present it to the user. We would then add external classes which dealt with the manipulation of the data/state.

We did this with our combat class, which manipulates the state of the board to determine if a player is being attacked, allowing the player to attack back, and kill enemies within the board (or terminate the game).

Had time not been a constraint, we would have also created controller classes for moving both the player and enemies around, rather than keeping them in one large board file (minimizing the reasons the board had to change within its own implementation file).

Resilience to Change

Our program supports the quick creation of new enemies and new player characters – creating them is incredibly simple. First, a subclass of the respective enemy or player needs to be made. Then, a spawning method needs to be added in the board class (calling upon the constructor of the newly created class to add them into the game). The board will automatically handle the printing of new players and enemies. From here, one of the following will occur:

- If they are a player, their interactions will automatically be handled by the board and combat classes.

- If they are an enemy, we can add them to our list of enemies in the board class to have them move around similarly to our other enemy types, or we can implement custom properties for them in the board class (such as movement options).
- Enemies can also have custom combat properties set up in the combat class (e.g having a greater or smaller chance of hitting attacks).

Our aim here was low coupling between the various subclasses of the creature and high cohesion with each other, having them each manage their fields and values. The board class communicates through these subclasses exclusively with public function calls – the private fields for the classes are never directly invoked. or public fields to modify values. All of these classes cooperate with the board to solve one task of spawning items within the board, yielding high cohesion.

Unfortunately, a limitation of our approach is due to everything inheriting from the abstract creature class, any changes to it will require recompilations for all of its inherited subclasses. In this regard, we will inevitably have high coupling.

Adding new commands to the game is also incredibly easy. Main handles all player input – so to add a new command, we merely add a new 'if' statement in main. From there, it would call upon the public methods in the board class (which would require new methods to support the respective feature sought to be implemented).

Main calls exclusively upon the public fields of the board class – never directly accessing the private fields. The purpose of main is well-defined, controlling the logic flow of the game. Thus, we have low coupling and high cohesion.

On the other hand, adding custom items (e.g gold, compasses, e.t.c) is difficult. Due to each cell only storing a pointer to a creature and these items having custom properties that vastly differ with their interaction with the player, we ended up having the board handling most of the logic relying on the consumption of these items.

As an unfortunate result, this means the consumption of items directly impacts the workflow of other items. We have high coupling between the item classes (potions, gold, barrier suit) and the board class, and mixed cohesion between them.

Between the board and combat class, we attempted to decouple our program interfaces by implementing the MVC pattern. Here the combat class uses the Single Responsibility Principle by exclusively handling interactions between the player and the various enemies on the board, minimizing coupling between the board and its controller and promoting code re-use.

Unfortunately, due to time restraints, we were unable to implement public methods for accessing various fields of the board class in time. We ended up implementing the combat class as a friend class to the board class, having direct access to its implementation rather than accessing public fields within the board class. Unfortunately, this produces high coupling but still has high cohesion with the board class.

Had time not been a concern, we would have separated the moving of enemies and moving of the player into additional controller classes as well, rather than having the majority of our game functions stored in the board class.

Answers to Questions

Q1: 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?

Similarly to our response for DD1, we created an abstract Creature superclass that acted as the parent class for all creatures and items in the game, containing attribute fields that are shared by all of them. Examples of such attribute fields include a constant integer MAX_HEALTH, as well as integers for health points, defense, attack, and gold values. We then had a concrete Player and a concrete Enemy subclass that was inherited from this creature class, which defined the respective abstract methods for the players and Enemies class. To construct different races for the player, the abstract player class was inherited by its various concrete subclasses, each representing a race respectively. Under such an inheritance structure, adding new classes (new types of Player) is simple – we just need to create more subclasses of the original player class and add modifications to our board class to allow the user to select the new subclass. This also has the added benefit of not breaking encapsulation.

We had planned to overload some of the operators for the various player subclasses, allowing for unique traits and methods that would not have been shared by other players (such as human players receiving twice as much gold when their addGold() method was called). These would have allowed us to implement the respective special abilities of the characters. Unfortunately, we did not have time to fully implement abilities as we focused our efforts on fixing issues with segmentation faults and memory leaks, however, the framework remains in our code if we ever had the opportunity to revisit it in the future.

Q2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

Our system calls upon uses the aforementioned inheritance relationship we discussed, where the abstract Enemy class is a subclass of the abstract Creature class. Similarly to the generation of players, all enemies call constructors that produce a specific type of creature since they share the same fields and methods (except for the Dragon class, due to their unique spawning requirements). The player class is similar in that all subclasses of player call upon the player constructor to create a specific type of player.

Provided we add more methods for each enemy, then they will all be similar to the Merchant and Dragon classes, which inherit from the Enemy abstract class, but have unique properties. We initially aimed to use the factory method pattern to control the generation of enemies to make them progressively more difficult as the floors progressed, however due to time constraints, we opted with random number generators as listed in the project specifications. Our hypothetical factory method pattern would have contained an enemyFactory class with spawning methods such as createGoblin(). With this pattern, subclasses could generate different types of enemies freely without exposing the underlying logic and complexity of creation directly to the user. This would have achieved encapsulation and lowered coupling. Unlike the other enemies, the Merchant and Dragon enemies had special inheritance, overloading movement() and combat() methods accordingly to display their unique properties.

Q3: How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

We initially planned to include a pure virtual method in the Enemy class called performAbility(player &p), which would be overloaded by its various children (e.g Goblins, Trolls, Vampires) to perform their various abilities. In each of these concrete subclasses, we would have taken a reference to the player and used public methods within the player class to manipulate various player fields and fields of the enemy class. This would allow us to add elements such as the loss of player gold when encountering a goblin for gold-stealing, or vampires healing themselves when attacking a player (dealing damage to the player and increasing the health field in the vampire class). The plan was for these ability methods to be public and use our random number generator to randomly trigger these skills whenever the player entered combat with a particular enemy. Unfortunately, due to time constraints, we did not get around to implementing them.

Q4: 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?

We still believe the decorator pattern was the best method of modeling the effects of temporary potions. The player creature would be the basic, concrete component of a virtual decorator class. Subclasses inherited from the decorator would be the types of potions the user has consumed (both positive and negative). After the player consumes a potion, its respective decorator would be added to the basic player creature. Using the decorator pattern, the attributes of the basic component - namely HP, Def, and Atk - would be modified according to each decorator added on top, similar to the image transformations in A4Q2, or the toppings of the pizza example provided in class. This makes explicit tracking of the consumption of potions unnecessary since the decorator automatically modifies the attributes at runtime. This design pattern also allows us to store the original value of the component, allowing us to remove them when the potion wears off, or reset potion effects when the player changes floors. One thing to pay attention to is that the HP attribute cannot exceed a certain maximum value, nor fall below the threshold of 1 health. We would handle this by setting a constant integer MAXIMUM_HP and MINIMUM_HP.

Unfortunately, we had written our combat class before we had an opportunity to implement potions. Consequently, using a decorator pattern here would completely break our working combat class and necessitate a complete re-write of it, which we did not have time for within the few days we had remaining in the project.

Q5: How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?

We initially aimed to use template classes to handle both the generation and picking up of different types of objects, promoting code reusability as the template allows us to handle the various objects in a similar manner. This also would have reduced the amount of unique code we needed to write for interactions with each individual object. We had hoped to make one template for consumable items (e.g Gold, Potions, Compasses major items), and a second template for items protected by dragons (which would attach one of the aforementioned template objects for them to ‘protect’). Once the dragon was slain, the template object it was protecting (i.e dragon hoard or the Barrier suit) could have been picked up, which would use the logic mentioned in the first template. Unfortunately, what ended up occurring was we wrote the classes for our potions, compasses, and gold piles independently and simultaneously (with each member taking up a specific item). This made it too difficult to coordinate on a template as we had conflicting implementations with different requirements, so we ended up opting to make all of these items a concrete subclass of the creature class.

Extra Credit Features

We used vectors rather than arrays for our board to handle its memory management. This way, we can easily clear the board by using the std::vector.clear() command and generate a new one to simulate changing levels, where the vector calls upon the destructor for the cell class.

Furthermore, our program dynamically spawns all items. Using an input file (board.txt or any other command line argument), we generate a map using that and spawn the items in random locations.

This supports the possibility of multiple floor layouts, provided a valid level map is fed as an input file (one where rooms are filled with '.' characters and have outlines defined by '|' and '-'), which we can populate with dynamically generated items and enemies.

Another additional feature we added was the ability for enemies to stand on the stair tile and prevent the player from climbing the stairs until they have been defeated, or until the enemy gets bored and moves away from that tile. Just as in high-octane action movies, you need to subdue the enemy on the stairs or sneak past them – going through them is not an option.

Final Questions

Q1: What lessons did this project teach you about developing software in teams?

Firstly, developing software in teams made it incredibly difficult to merge code together between different branches of the project. One person may have changed core logic in one part or fixed a bug locally on their branch of the program, only to completely break another member's contribution when merging. We also had issues of accidentally overwriting functions and methods when merging. Using GitHub helped remedy some of these issues, however, we found the best way was to modify as few files as necessary for our contributions, communicate which files we edited, and clearly mark out where we made changes before merging them. This significantly sped up the code merging process but was a difficult learning curve to overcome and required communication within the team.

Secondly was the importance of bug testing and quality assurance. As we added more functions, methods, and features to the game, the complexity skyrocketed, and so did the number of issues we encountered (from graphical errors to segmentation faults). Seeing as each of us thinks differently, this allowed us to identify errors with each others' code and provide different ways of fixing them that we hadn't previously considered. This not only allowed us to create more resilient and robust code but also learn as developers from each other.

Perhaps the biggest lesson was the importance of coordination and properly evaluating what we could achieve within a given time span. We set out a plan and a list of what we should have accomplished between each due date. However, our plan did not account for any issues that would cause delays with the project – such as the aforementioned merging issues, or external factors. For instance, Stefan got sick halfway through the project, James had a final test on Thursday, and Robin had an assignment due Friday, all of which impeded the progress we could each individually do on those dates. Thus, we fell behind our schedule and ended up squeezing right against the deadline to fix all the last-minute issues we encountered. We should have coordinated with each other's schedules appropriately and shifted the workload of each team member on those critical days to avoid this circumstance.

Q2: What would you have done differently if you had the chance to start over?

If we had a chance to start over, we would have reordered our steps. For instance, we initially planned to use the decorator class to simulate the picking up of potions. The player would have been a concrete component of a decorator, and then the potion classes would have been added on top of it.

Unfortunately, we wrote our combat class first which handles the interaction of players with enemies, and added potions afterward. By doing steps in this order, our combat class was still using the base player's stats to calculate damage and not applying the potion effects correctly. This would have necessitated revisiting the combat class and completely writing it from scratch, which unfortunately was not an option due to time constraints (we had 1 week to complete this project, while simultaneously completing tests and final projects for other classes). Thus, we resorted to using the public methods we had in the abstract creature class to adjust the player's fields, which would allow the combat class to apply the appropriate damage calculations without any changes.

We also would have grouped tasks together better. Initially, we had hoped to have an abstract Item class that would handle the picking up and consumption of all items. Unfortunately, due to time constraints, we ended up having two different people working on the various items, one person dealing with potions and ladders, and the other dealing with gold and the barrier suit. This completely threw our idea of using template classes out the window as we each cobbled our own implementation in the time we had remaining.

Conclusion

This was the first-time developing software as a team for all of us and provided key insight into how the design and complexity of a project can quickly change from our initial plans. While we aim to employ good object-oriented design, this is not always possible when working with others as the project scope we initially planned can change with time, as we encounter unexpected errors and make major revisions to our framework. It served as a valuable experience for us – demonstrating the pivotal role that planning plays within the development process, the challenges of programming in a team environment, and how to overcome adversity. As developers, sometimes we are forced to make do with what we have and adapt to circumstances to meet deadlines, hence why effective planning has such a critical role to avoid these incidents.