Design Document

Geoffry Song and David Choi

Command Line Arguments

"-gnarly" enables gnarly mode, which uses all the DLC we made and the Curses UI. "-f [filename]" takes input from a file and uses that to decide floor layouts.

Project Design

Important Classes

The fundamental abstract classes are: Object (any concrete object), LevelObject (any Object that belongs inside a Level including Player, Monster, Gold, etc), Displayable (anything that can be displayed including Monster, PopUps, Dungeons, etc), Surface (a drawing surface), UI (a method of receiving input and output), Character (extends LevelObject and represents any living object like Monsters or Players), Item (anything that can be placed in an inventory), LevelGen (a method of generating a Dungeon layout and placing monsters therein), Spawn (a method of choosing which monsters to use), Class (a collection of skills added to a player), Skill (a skill associated with a class), and AttributeProvider (provides attributes inherent to a Character such as starting HP/ATK/DEF).

Overview (Including DLC)

In this document, "basic" will refer to the base version of the game without DLC and "gnarly" will refer to the version of the game with DLC. Also, () will denote something as a function, not necessarily that the function has no arguments.

When the game begins, the main function calls Game::instance which creates an instance of the Singleton Game and tells it to run. The Game class is the central class that takes receives input from a UI object and uses this input to control the game. When the Game is initialized, first the UI singleton is decided (either the basic command line UI or a Curses UI), the race/class the player wants is selected through the PlayerSelect class, and the Display created. The Display is simply all the objects that need to be drawn, sorted by z-index so we can draw certain Displayables on top of others. Then, the LevelPlan object is created, either the basic plan consisting of eight of the same dungeon layout with monsters placed randomly, a sequence of dungeon and monster layouts specified in a command line argument, or the gnarly plan which will be discussed later. The Level is then created from the LevelPlan using the generateLevel() function.

After that, the main function calls the run() function on game which begins the game loop. The game loop takes a command for the Player, tells Level to step all its objects, tells the Display to draw onto the UI, and tells the UI to refresh (draw its contents). run() waits for the game to end, and after it does returns and allows the main function to decide whether or not to restart based on a flag changed in Game. Specifics of how everything is implemented are described below.

Default Drawing and Input

Input and output are controlled through the UI. The Game extends CommandHandler, which is an interface passed to UI for the UI to call certain methods on. For example, when a direction command alone is sent, the UI calls move() on the CommandHandler instance. The UI thus interprets low level input and sends the high level command to a CommandHandler. Low level input can also be received through the readChar()/readLine() methods.

For output, the UI extends Surface. Surface has pure virtual methods for drawing individual characters, strings, setting color, etc. BasicUI implements these, drawing whatever is necessary in certain positions on a grid, and drawing that grid when redraw() is called. UI includes a variation on the singleton pattern, because its instance can be set to the desired type of UI. To display game messages, the say() method (part of UI) is called with the message, and draws it at the bottom of the screen. Display manages the things that need to be drawn, Displayables, and sorts the by z-index for drawing.

Characters

Characters are any living thing inside the game like Players and Monsters. Characters have HP, are affiliated with a Team, can attack and be attacked, and can die. Attacking other Characters calls the other Character's computeDamage() which uses the specified damage calculation; this method returns a Damage object, which can be queried before its effect is finalized by calling its apply() method. Whether a Character will attack another by default is determined by their Team. Each Team is a collection of alliance statuses with every other Team, and is used to see if two monsters are enemies by checking if their Teams are. Characters also have attributes, which determine their attack, defence, starting hp, name, tile, etc. These attributes are provided by an AttributeProvider object, the concrete instance of which is Attributes.

When Characters die, they usually inform the Level they are associated with (using the observer pattern) that they are dead.

Player Control

Player belongs to and is controlled by the Game, as opposed to all other LevelObjects which are managed by Level. Each game loop, Game calls queryCommand() to get input from UI, and controls the Player using that input. move() moves the player in a certain direction by calling the LevelObject method moveRelative() with a direction, which then calls moveTo() by interpreting the direction and its current position, and then informs Level that it is moving. How Staircases, Gold, and Potions (through use()) work will be described below.

When the attack() method is called, the player attacks the target by calling attack(). Notably, when the Player dies, it does not notify the Level and instead notifies the Game.

The Main Game Loop

Each loop, Game first calls print(), which tells the Display to draw every Displayable associated with it to the UI and then draw the UI. Then, it waits for a command from the UI using queryCommand(), acts on the command as described above, and tells the Level to step everything associated with it if appropriate. Level then calls step() on each of its objects, which gives its LevelObjects a chance to

perform turn-based actions, such as attack()ing an adjacent enemy or wandering randomly. Finally Level removes all dead LevelObjects from itself.

Player and Monster Races

Player and Monster races are determined primarily in the differences in their starting Attributes, which are passed to the base Character constructor at creation. Individual special abilities, though, require subclassing off the main class and overriding certain virtual methods related to the special ability. For example, DwarfPlayer overrides addGold() to add twice the amount of gold. Notably, each Dragon is tied to a DragonGold instance and notifies the DragonGold when the Dragon is dead through the observer pattern, allowing the DragonGold to be picked up. Also, when a Merchant is killed, it tells its Team to unally with the Player Team (so that future Merchants will attack Players by default).

Potions, Gold, and Staircases

Potions are Items that are applied by passing a Player through their use() method when Game tries to use them. They then call a method — either applyBuff() or heal() — depending on the Potion's type, on the Player which operates on itself. However, to implement inventories (for DLC), Potions are not LevelObjects (since a LevelObject must be part of a Level). Instead, during their existence on the level, they reside in an ItemAdapter, which extends LevelObject and provides an interface using the Adapter pattern that allows a Potion to both act on its own and be placed into a level.

Gold and Staircases are not Items and are placed directly onto the level. When a Player tries to move onto a space where there is a LevelObject, a MoveIntoVisitor is used (using the Visitor pattern), which either descends the staircase (creating a new level) or calls Gold::use() with the player passed in, adding gold to the player.

Basic Level Generation

LevelPlan contains a vector of LevelGen objects, each of which specifies how each level should be created. Each time a new floor is necessary, LevelPlan's array is indexed into to get a LevelGen, whose generateLevel() method is called. LevelGen first decides how it wants to create the Dungeon layout and creates a Level with that Dungeon, then uses a Spawn instance to create LevelObjects and place them.

By default, we have two LevelGen subclasses and a BasicSpawn class. The first LevelGen subclass is ConstantGen which takes in a constant dungeon layout and uses that. It then uses BasicSpawn to create monsters, and places them by calling Dungeon's randomPlacement() to get a position. The second is FileGen which reads a constant layout from a file and uses that exact layout to both generate the Dungeon and to place LevelObjects.

DLC Content Design

Classes

Classes are aspects of Players, and consist simply of arrays of Skills with some buffs. When a Player is created (or levels up), it asks its Class which buffs it should apply on itself and uses those. When Game receives a command to use a

skill(), it tells Player to use the Skill which tells Class to use a Skill which tells Skill to use itself on the Player, receiving a target if necessary.

Skill use is managed by mana (MP), a new field on Characters and attributes. Each skill uses a certain amount of mana and cannot be activated if the player has insufficient mana.

Field of view

Field of view (FOV) is implemented using the Restrictive Precise Angle Shadowcasting algorithm. Every time Game tries to print(), it asks the Level to compute the player's FOV (a 2D array of boolean values of what the player can see) using the player's current position. Then, only tiles that are visible to the player are drawn. For tiles that were once visible, a Memory class that is both a Surface and Displayable is used. It records what the tiles last looked like when the Player could see them, and draws them (but greyed out).

Targeting

Targeting is done through the Target class which extends CommandHandler. When a direction key is pressed, it moves the current target. The static method getTarget() uses this class to pause Game until the user chooses and submits a target, then returns the targeted location.

Leveling Up

Players have an additional field called _currentXP which represents the amount of experience they have, and attributes have a field _xp which represents how much XP the relevant monster gives upon death. When a Player kills a monster, in addition to getting gold, they also receive experience specified by that monster's attributes. When a Player reaches their targetXP, it levels up, asking its Class which buffs to apply and fully restoring HP/MP.

Monster AI

Each time a monster steps, instead of wandering it first checks if it is currently following someone and if so uses moveToward() to approach them. Otherwise, it asks its Level for all LevelObjects that it can see (and Level uses FOV to determine that), chooses one valid enemy at random, and sets it as its follow target, then moves towards it.

PopUp

PopUps are Displayables that show large amounts of text, drawing their own border. PopUps are created with a make() method similar to Target, which waits for and exit signal to close the PopUp. Instead of creating PopUps directly, PopUpCreator uses the Proxy pattern to create them with specific texts.

Additional Monsters/Races

Additional Monsters/Races were added by adding different possible attribute sets, and used for later dungeon levels to add difficulty. The only exception where we added something with a special ability is the Halfling race, which instead of taking damage in computeDamage(), avoids the attack 20% of the time.

Inventory

An Inventory belongs to a Player, and is a set of items indexed by chars. Instead of automatically using Potions, they are unwrapped from their adapter and added to the Inventory. Then, when Game receives a command to show the inventory, it tells Player to show its inventory, and Inventory creates an InventoryPopUp which functions as a normal popup, except any key other than close window will tell the associated Inventory to try to use that item on the player, calling the use() method on the item as above.

Other Level Generation Methods

We added four more subclasses of LevelGen: RoomsGen (which generates and connects rooms, with everything not set either floor or empty depending on whether it was inside or outside), ForestGen (which places random trees everywhere), AggregationGen (which uses the “diffusion-limited aggregation” algorithm to create a cave-like area), and FinalGen (which extends ConstantGen, with a fixed layout, and places many random monsters). The gen() method in each of these uses the Factory Method Pattern. We also added a GnarlySpawn, using the Abstract Factory pattern, which returns our custom monsters instead of the basic monsters that BasicSpawn returns, and decided which monsters to return based on dungeon level.

Curses UI and Color

The ncurses library UI functions almost exactly the same as the BasicUI with solely syntax differences. Color is an exception to this though, because CursesUI supports color while BasicUI does not. Color was implemented simply with the curses library function.

Player Selection

PlayerSelect runs at the beggining of the game and creates Player. It works by having an active copy of the currently selected class and player race, and asking the UI for the next char. It switches the class/race based on that char, and when space is pressed it returns the current Player with Class attached. PlayerSelect thus does not truly own the player, it simply returns it. PlayerSelect doubles as the titlescreen by drawing a title image

Design Patterns Used

Pattern Name	How it was used
Abstract Factory	The Spawn class returns Monsters, Gold, or Potions, but exactly which ones it returns depends on the subclass. GnarlySpawn and BasicSpawn return two different sets of LevelObjects.
Factory Method	The gen() method in LevelGen returns a Dungeon, but its layout and whether or not it was random is decided by its subclasses.
Template Method	Used in many places. For example, Surface implements fillLine() using draw(), but draw() is deferred to the subclass.
Adapter	ItemAdapter provides an interface for an Item to allow it to be placed onto a Level.

Proxy	PopUpCreator is a proxy that serves as an interface to PopUps. When its methods are called, it creates a PopUp with the appropriate text.
Decorator	Buffs, from Potions or Classes, decorate Attributes to allow semi-permanent stat modification.
Observer	Used in many places. For example, when Monsters die, they notify Level of their death which queues them for deletion. Similarly, Player notifies its skills whenever an action happens that they should know about.
Singleton	Game and UI are singletons, because only one instance of them should exist at a time.
Multiton	Teams are multitons with indices corresponding to their teams.
Visitor	When a Player moves onto a space occupied by a LevelObject, MoveIntoVisitor decides what should happen.

How it differed from original design

The core of the game remained the same as we originally planned. Changes were only made to accommodate to additional content. Because we added different level generation methods and more monsters, we created a new LevelGen class which determines the layout of a dungeon floor, which objects to spawn, and where to spawn them instead of the Level. Thus, we separated the creation of the level and the management of how the objects are handled. This change also means that Dungeon no longer decides its own layout, but is instead given a layout to conform to.

The other change is that staircases no longer tell the Game to descend a level, and instead Game checks for staircases, and if they are present uses them in order to decrease coupling and rigidity of the Staircase class.

Questions:

None of our answers differ from those given in week one. We simply elaborated on a few of them.

Question 1:

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

We could create a base player class from which races inherit. This system would allow us to simply add a new subclass of player to create a new race, so we could reuse the code for player. In our implementation of classes, an array of skills that can be used by the player along with stat modifications applied when levelling up or choosing the class, it is easy to add classes because we only have to modify the base player class. Then, when a player wants to use a skill, it is called through

the player, and when a player levels up it tells its class to apply a buff onto its attributes.

Question 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 has a base “Monster” class from which specific monster inherits if necessary. Because of this, we can reuse most of the code of monsters and only implement methods when necessary. Since the only difference between many of our monsters is their HP/Atk/Def, this means that for most monsters, there is no subclass necessary. This is different from how we add player races only because not all monsters have special abilities so we do not always need to have a subclass. If subclassing is necessary, the solution is no different from how we generate races because they are both similar problems.

Question 3:

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 could create subclasses of the “Monster” class that override certain methods (like those that relate to attacking for goblins) to add special ability functionality. For example, in the step function for trolls, they could call addHP() inside step() after their action is complete. Similarly, for gold stealing goblins, the attack method could decrease the amount of player gold.

Question 4:

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 could use the Decorator pattern. We made an AttributeProvider class which AttributeDecorator and Attributes derive. AttributeDecorator decorates an AttributeProvider but by default just returns the base class values. Attributes are the collection of attributes associated with a race (for example, a werewolf’s attributes would be 120HP, 30ATK, 5 DEF, the tile ‘W’, and the name “werewolf”). A Potion’s buff extends the AttributeDecorator and returns whatever the base class returns modified by a value specified the potion. A slight extension of the pattern was a strip() method which removes all decorators created by potions and returns the base provider. This method could be called when changing floors, to not explicitly track which potions were consumed.

Question 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 do not duplicate code?

We could have a base LevelObject class, which represents any object on a level and includes placement methods like setPos(), and a Dungeon class, which can return a valid placement following the given rules. There is also an abstract LevelGen class that generates the dungeon layout and decides what to place and another abstract Spawn class that returns a LevelObject to be spawned when asked for it (purely for our DLC purposes for different generation methods). The derived LevelGen would simply ask Dungeon where it should put that LevelObject, ask the derivative of Spawn which LevelObject it should put there, then place it there and put it into the level. Since Gold and PotionAdapter (which we use as an adapter for potion so that we can add a potion to the inventory but still have it placed onto the level) both extend LevelObject, they can be placed in the same way.

Final Question 1:

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

We learned the importance of planning both the project and who does what, using source control, and compromise. Since two of us were working together, we needed to talk to each other about what we were doing and how we would structure the project instead of immediately starting to code. Coding without discussion and planning first sometimes led to having to rewrite large sections of code when the other partner thought something should be done in a completely different way. We also disagreed about how exactly some of the bonus content should function when we didn't have a requirement for it. This led to sometimes having to compromise on some details after we had finished implementing them, wasting time. Planning out the details of how it would work would avoid that.

We found source control very helpful in the project, and noticed that other teams that did not use it sometimes ran into problems. We could look over each other's code, easily revert things when necessary, and easily work asynchronously, because we used git. Other teams sometimes lost data or were inconvenienced because they did not.

Final Question 2:

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

Overall, we are very happy with what we achieved in this project. We added a lot of extra features, and stuck to our planned schedule of deadlines. From what we learned, if we started over we would better plan out exactly how things were going to work and what the DLC features would involve before coding.