Shawn S Hillyer

CS 162-400

November 28, 2015

Final Project: Reflections Documentation

Concept Overview / Game Design

I've decided to do an underwater adventure game where the player is a deep sea treasure diver. They dive down into the water and a current sweeps them into an underwater cavern.

The game must contain rooms or compartments or spaces of some sort. My game will utilize several area types the player will be able to explore. The first type will be an underwater cave (**`Cave`** subclass). The cave will be dark, so the player can try to light a diving flare (**`Special`**()) to illuminate the cave and search for lost treasure or objects. Doing so also reveals all of the exits from the cave without having to check north, south, east, and west. If the player gets the treasure, they will later be able to use it to bargain a way out of the ocean with the MerKing.

The second type will be a sunken ship (**`Ship`** subclass). In the sunken ship, the player can decide to enter the ship (**`Special`**()). There's a chance upon exploring the ship to find extra treasure or artifacts, but sometimes a shark will attack them – perhaps a hidden 25% chance. A shark attack will cause the player to expend precious oxygen (2x a normal turn?) in attempting to escape.

The third type will be a Coral Reef (**`Reef`** subclass). The special here will be to examine a corpse (**`Special`**()) on the reef. Doing so will give the player extra oxygen tank and gives them a distress beacon in their bag. If the player uses the distress beacon while in Open Ocean game space and waits a certain number of turns (perhaps 10?) then they will be rescued. I will make it so that a player with exactly 1 air left when they explore the body will get enough oxygen so that they can swim to open water and use the beacon without dying, with no margin for error. Players that get here earlier will be safer.

There will be a fourth area called Open Ocean (**`Ocean`** subclass). The only thing a player can do is wait for rescue (**`Special`**()) or explore what's north, south, east, and west of them. If they wait for rescue, normally nothing will happen (except lose air). However, if they've used the distress beacon, then after the determined number of turns, a rescue ship will fish them out of the water.

The fifth and final area will be called Abyssal Trench (**`Trench`** subclass). A player can use all their remaining oxygen to explore into the depths of the trench (**`Special`**()). In doing so, a great MerKing will capture him. The MerKing will ask the player if he will pay tribute to escape. The player will then get the choice of presenting the MerKing with treasure or not. If the player has found an Ancient Artifact item or is carrying 10 minor treasures, then the MerKing will bring them to the surface and leave them on an occupied island. If not, the MerKing will feed them to his daughter, Ariel, for supper.

The player will have a Diving Belt to carry/attach items. The container will have a maximum size of, say, 25-35 items (I will adjust depending on how big I end up making the map). Other items (in addition to

flares) they can find in the caves, ships, and reefs are special Diving Flippers, additional oxygen tanks, the aforementioned distress beacon, and an Ancient Artifact to give to the King.

The enforced time limit will be the oxygen mechanic. I'll make the oxygen cap at 100. Reaching 0 causes death. Each time a player tries to move in a direction or take an action it uses some oxygen. Escaping a shark takes extra. Certain items like oxygen tank might increase the oxygen again. Each area will have a different type of cost involved. I'll make this more interesting by providing diving flippers that reduce the movement cost in certain situations. Diving deep into the Trench uses all of the oxygen except 1, although at this point the player either wins or dies after the prompts with the king.

Class Design

This project must be text-based game or puzzle. As such, we won't worry about drawing the various objects, etc. We will focus on providing a simple text-based feedback system.

The class structure will require a main **`GameClient`** that handles the game logic. I will use the `Client` class I've used throughout the last few projects as a base class and modify it to create a `GameClient` class. Menu options will include, at a minimum, Play New Game, Game Hint (reveals goal), and Quit. The `GameClient` will be extended with the main logic flow I will need. Methods should involve those that advance the game forward, update the screen with information for the player, and variables to track the game status (ie, if player has won or "died", etc.).

All menu screens will be handled by my **`Menu`** class I developed. Each **`Space`** subclass will have its own menu file so that when that space loads, it loads the menu relevant to the area they are in, and overrides the Special function.

The **`Space`** superclass will have the pure virtual `Special`() method and some other basic functions. It will have a north, east, south, and west pointer that points to the other Space objects, so this will be a pointer in each case. We will think of the **`Space`** class as a data structure. The five areas described in the concept section will be the subclasses: **`Cave`**, **`Ship`**, **`Reef`**, **`Ocean`**, **`Trench`**.

To connect the various spaces, we will first initialize any instance of a space to point to null. There will be a public method, `check_connected`() that returns true if the node for a certain direction is connected to another space. Connecting the spaces will actually be interesting challenge. When a node is added to the north, then the new node's south node must point to the linker. And so forth for each of the different directions.

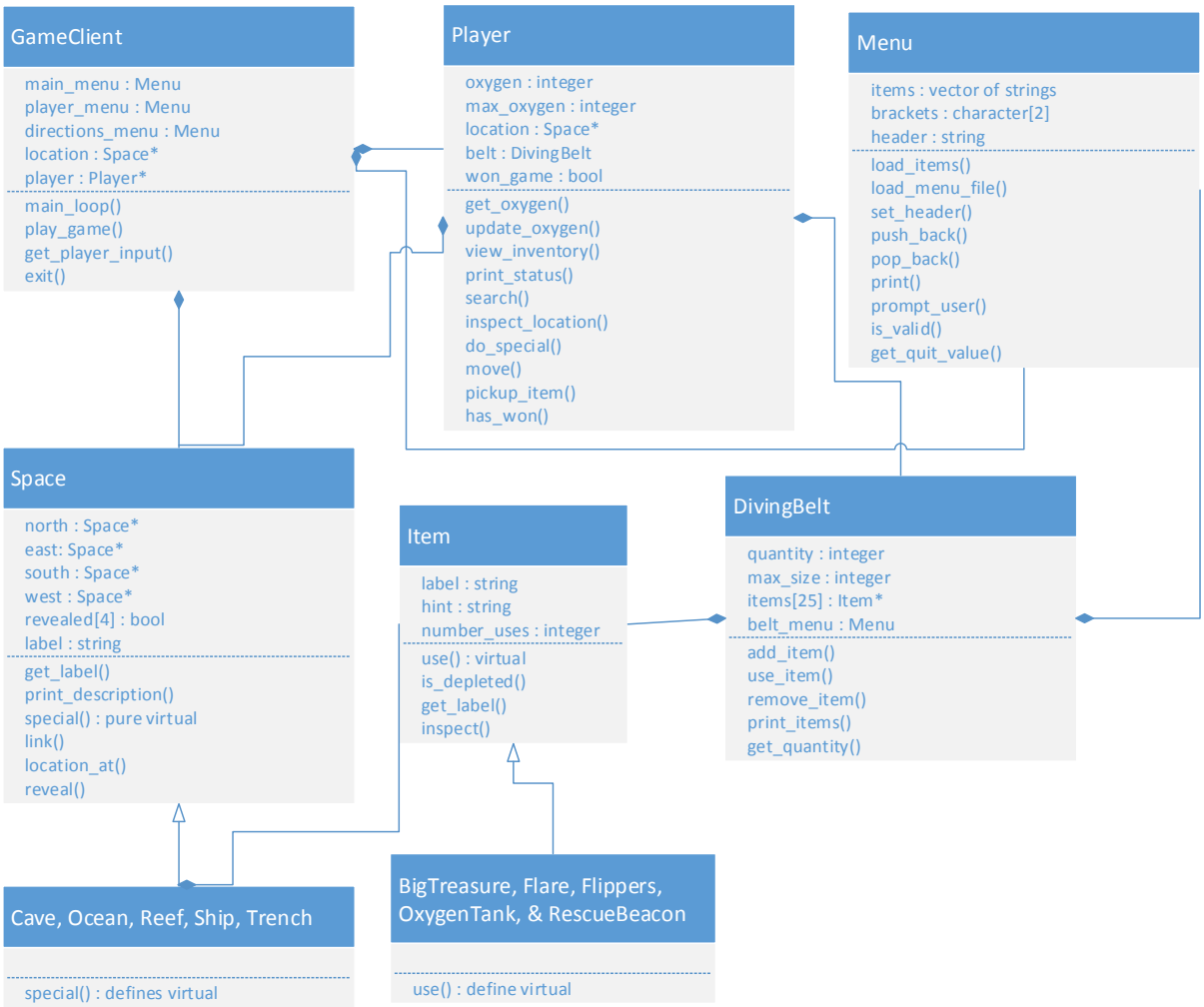
Although I'd love to build a super robust random level generator, I think for sanity, I will have function that just builds the gameworld based on a map I preconfigure. I'll draw out what the map will look like and what the connections would be needed to properly link it, then I'll manually connect them all.

For the player's "bag", I'm going to make my own simple ADT called **`DivingBelt`**. The diving belt will be implemented as an array of a fixed size. It will have a member to check if the bag is_full(), add_item(), remove_item(), and print_items(), etc.

There will be an Item class as well. Items are going to be simple in this game; they will have a label of what they are called, and probably a use() function. Each type of item I make will be inherited from item, so I can allow the player to use various items polymorphic behavior without needed to know what type of item is in the slot through lots of if-else statements.

There will of course be a **`Player`** class which will have a `DivingBelt` data member (composition), oxygen level (simple int should suffice), and a variety of functions based on what options they pick. They will need to be able to pick_up an item, drop items, etc. See the class diagram sketch on next page.

Class Diagram



Test Plan & results

I used incremental testing as I made each individual method during the process of creating this program. There were some interdependencies that meant some of the time I was modifying multiple classes so that I had a design pattern that would work. My test plan was incremental because of that. However, I knew that I wanted to conduct a final playability test that would verify all of the functions and classes were working properly in the context of the entire game. I wrote a testing plan which has been expanded to account for my final design changes.

The tests are in alphabetical order by the class that is involved in the test.

Class	**Test**	**Expected Result**	Result

BigTreasure	Inspect item in inventory	"Ancient Artifact" label "You might be able to barter with this, or take it home and sell it." hint	Pass
	Use item in inventory while not in Trench zone	"You admire the Ancient Artifact. It must be priceless!"	Pass
	Use item in inventory while in trench zone but haven't swam down	"You admire the Ancient Artifact. It must be priceless!"	Pass
	Use item in inventory after meeting the MerKing	Prompted if will give to king	Pass
	(part 2) : say yes	Player wins game	Pass
	(Part 3) : say no	Player is killed by king and loses all oxygen/game over	Pass

Cave	Use special menu option in caves	Should get either a Flare or Flippers	Pass
	(part 2) Verify menu updated	Menu should display that area already searched	Pass
	(part 3) Try to dive again	Should display message area already searched	Pass
	(part 4) Try again at a different cave	Should be a unique instance and start again	Pass

Direction	Inspect screen output throughout program	All labels should match the user input and results	Pass

DivingBelt	Inspect belt before having any items	Should print message the belt empty and return to player input menu	Pass
	Inspect belt after having any item	Should print all item(s) in bag and provide a prompt to Inspect, Use, or Drop an item	Pass
	Inspect an item	Should print the information about item name, number of uses left, and the hint	Pass
	Use an item then look in bag again and inspect item if it's still there	Bag should remove an item if it had only 1 use left, or the item should have 1 less use upon inspecting item	Pass
	Find an item in any area	Item should go into bag	Pass
	Find an item when bag already full	Should get prompt to use or drop an item in bag and try to add it again	Pass
	Drop an item and view inventory again	Item should be removed from bag	Pass

Flare	Use a flare in Cave	All exits are "revealed"	Pass
	Use a flare in Ocean	All exits are "revealed"	Pass
	Use a flare in Ship	All exits are "revealed"	Pass
	Use a flare in Reef	All exits are "revealed"	Pass
	Use a flare in Trench	All exits are "revealed"	Pass

Flipper	Use the flippers after finding	Screen output that flippers are on, item no longer in belt	Pass
	Search after using flippers	Should cost 1 instead of 2 oxygen	Pass
	Move after using flippers	Should cost 5 instead of 10	Pass

GameClient	Play game until out of oxygen	Should get a game over message about running out of air	Pass
	Play game until use rescue beacon in open water	Should get a message about winning game	Pass
	Play game until give king big treasure	Should get a message about winning game	Pass

Item	Inspect every item type once	Should get correct description of items	Pass

Map	Navigate to every node of the map and reveal all positions	There should be no linkages except where noted on map, and every node should allow traversal in both directions.	Pass

Menu	Verify all menus only allow the input range that they display	Reprompts if invalid menu option entered	Pass
	Verify last menu item updates when in a different type of Space subclass	Last menu item should display contextually depending on area the player is in	Pass

Ocean	Use the special option	Player should lose 1 oxygen	Pass

OyxgenTank	Use the oxygentank	Player should get 100 extra oxygen	Pass

Player	Verify starting oxygen	Oyxgen should be 200	Pass
	Verify starting location	First location should be open water #1	Pass
	Verify oxygen goes down/up when appropriate	Verify based on the dialogue before and after the cost is displayed	Pass
	View inventory	Should call the diving bag's print function	Pass
	Verify bubbles on print_status	Should be one bubble ('O' character) for each oxygen also should see correct/current oxygen level and location	Pass
	Search in a direction not yet searched	Should use 2 oxygen (1 with flippers) and reveal what (if anything) is in a direction	Pass
	Search in a direction already searched	No oxygen cost, message that already searched	Pass
	Inspect area	Should print the current location's description	Pass
	Move in a direction that isn't revealed	Should get message that havne't searched that way yet	Pass
	Move in a direction that IS revealed	Should cost 10 oxygen (5 with flippers) and player location will be updated	Pass

Reef	Use special menu option in Reef	Should get either a Oxygen Tank or Rescue Beacon	Pass
	(part 2) Verify menu updated	Menu should display that area already searched	Pass
	(part 3) Try to search again	Should display message area already searched	Pass
	(part 4) Try again at a different reef	Should be a unique instance and start again	Pass

RescueBeacon	Use when not in Ocean	Message that it doesn't work	Pass
	Use when in Ocean	Prompt user if they want to wait for rescue	Pass
	(part 2) respond yes	Should get a message that you win if > 10 oxygen	Pass
	(part 3) respond yes when oxygen < 10	Should get a message that you ran out of oxygen and lost	Pass
	(part 4) respond no	Should get message that you decide not to wait	Pass
			Pass

Ship	Use special menu option in Ship	Should get either a BigTreasure or OxygenTank; cost 15 oxygen	Pass
	(part 2) Verify menu updated	Menu should display that area already searched	Pass
	(part 3) Try to search again	Should display message area already searched	Pass
	(part 4) Try again at a different Ship	Should be a unique instance and start again	Pass
	Verify shark attacks occasionally	Should get message that lost an additional 10 oxygen	Pass

Space	tested through concrete classes		Pass

Trench	use special	prompt if you want to dive down or not	Pass
	(part 2) Say yes	Should dive down and lose all oxygen except 1 and be presented with option from inventory	Pass
	(part 3) Use Artifact and say yes to give	Win game	Pass
	(part 3) Use Artifact and say No to give	Lose game, king takes your oxygen	Pass
	(Part 3) Use oxygen tank	oxygen goes up, presented with menu; you have enough oxygen to flee now	Pass

UserInterface	Verify the files created all print	Should see the files from the text folder print at various points	Pass

Overcoming Challenges During Implementation

Game Design Changes

There were some game design changes I made that were more about making a balanced game, rather than program design flaws. First, I decided to give some areas loot tables and changed the special for the cave to allow the player to find loot. Second, I reduced the size of the bag down 7 because 25 was excessive. I figured with a map size of about 28 nodes, I could have a pretty fun game, so that's what I ended up doing. The oxygen level also was increased from 100 to 200 to balance things a bit.

Design Extensions

I extended my design in a couple of relatively simple ways on a high level. I added a UserInterface class to handle the printing of large blocks of text rather than using cout statements. The text is read from a plaintext file in a subdirectory. This just means I need to distribute those text files with the executable /

source code if somebody wants to play the game, which is fine. The method also has a way to validate yes/no input from the user and print bars of variable size pretty easily.

I also realized I wanted a simple Direction class to handle the conversion from integers to strings. I could have somehow converted user input from digits to somehow be an enumeration, but I decided to just make it translate the integers to strings for the user.

I also decided to encapsulate the various Space* concrete objects into a Map class that would have the function for making the map and a pointer to the start. I did this so I could also make the game just have a Map, and the player would get a pointer to a Space that was in the Map. The map would basically hold on to all of the nodes in memory so they never fell out of scope and deconstructed.

Overcoming Challenges

The first major challenge I ran into when implementing this program was the realization that my Special() function – which took no arguments and returned no value at the time – would have a hard time doing anything to any other function. For some reason I decided I needed to hack around this and so I took a break for the night. I realized after mulling it over the next morning that I could just pass the Player as a pointer into the Special() method so that I could access the player's data, like oxygen, items container, etc.

I also realized that a vector – although the first simple ADT I used during this class – was not really ideal for grabbing individual items from the list and removing them. So I set out to find a better structure to use in my DivingBelt class to store items. I ended up implementing a variation of the Bag structure so that I could add exactly the methods that I needed. I made the structure an array and allowed for the deletion of items at specific locations when passed an index. When an item is deleted in the "middle" of the occupied bag slots, I made the operation shift all of the values after the deleted value up one slot. Honestly, this was one of my favorite things to implement into the program.

I also realized that my use of the enum Direction was cumbersome because it couldn't automatically convert integers to and from the type of Direction, so I had to upgrade it to a class that accepted integers and could return integers that represented the four cardinal directions.

My design didn't make it very easy to quickly edit large paragraphs of text output that was used to describe scenes or information. I implmented a UserInterface class to help deal with this. The class basically was a place to provide static methods to do various functions that involved the user interface. For example, print_file allowed me to just call the function and pass in a file path as an argument to print any number of lines of text. This meant I could go into any of my functions, like all of the subclasses of the Space class, and write up flavor text and information that was static and just drop it into the game without having to edit the source code and use lots of cout statements, etc. The idea was heavily inspired by my use of the Menu class I built.

I had to extend several smaller methods on a lot of the classes that I didn't quite realize. Most of them were related to accessing member variables. For example, I had to add a method to my player that would make him use special, which would have access to the Player::location pointer. I also had to add a variable to Space to track whether the special had been done or not (otherwise a player might do the same Special repeatedly).

I wish that I had some knowledge of a design pattern that would make my code more elegant. As it stands, I have to use forward declarations in a few places and call the methods on the players Items by accessing the Player, the players Belt, and then the Item… then, I pass the .use() method a pointer to the Player(). This works and more closely resembles functional programming than I was expecting to have to get.

As I got done implementing the Space (and subclasses), Map, the Player, the DivingBelt, and the Item classes, I felt I had a fully playable game. I started trying to think of a way to allow for random loot that is specific to the type of zone you're in. The simplest way was to give the Space class a pointer to loot_table which is just a vector of pointers to Item objects. During the constructor method of each subclass of Space, I designed it such that I could populate the vector with as many items as I'd want to. In order to make this random, I added a method to Space that would return a random element from the array (this was trivial).

So now I had random loot and all of my methods working, and a playable game. Overall I felt that my design was strong and was able to allow me to address most of the macrodetails of implementation. When it came to the finer points, I ended up having to work out a lot of the details as I went along – specifically, the arguments I'd need to accept, setting up "getter" functions that I didn't anticipate needing, and coming up with elegant handling of various game mechanics.