W17 - CS 162: Final Project Design/Reflection

Howard Chen

March 21, 2017

Requirements

Your program must implement a computer game using object-oriented programming, inheritance, and polymorphism with pointers. In the game, the player must move through a series of rooms in pursuit of some goal, which may be random, to win the game. To make choices, the player will be presented with a menu of choices in each room, and must enter an integer corresponding to their choice.

From the beginning, the player must know the goal they are searching for. They will start in a beginning room, and each room has four adjacent rooms (up, down, left, and right) that the player can go into. There must be at least 6 rooms in your program, and there must be at least 3 types of rooms. Each type of room must give the player a special action they can perform (in the context of the game). As a result, the player *will be able to interact with each room and change the state of the room!* This makes it something like a puzzle game.

In addition, the player must have some sort of container for carrying items that they either found on the train or originally had. The container should have some sort of upper limit on size or weight, and the player should be able to at least view and examine the items in the container. One or more of these items will be required to reach the goal of the game.

Finally. the game itself must do two things. First, it must keep track of the player's current location, so the player can interact with the correct room. Second, the game must keep track of how much "time" the player has left to reach the goal. This time will probably be measured in how many rooms/actions the player has visited or used, and the "time" can change abruptly based on player actions (for example, if the player starts the timer on a bomb).

Other than that, you are free to design the game as you wish.

Analysis of Required Program Elements

Program Theme: You are the first in the series of experimental medical microbots that have been developed to enter the human body and physically destroy pathological structures that are causing illness. While it has been shown that you can enter and exit healthy human bodies with no side effects, you have not yet been tested in the very ill. However, the beloved leader Kim Jong Un has secretly fallen incredibly ill. Medical imaging indicates several brain clots, a malignant tumor in the adipose (fat) tissue, and several cavities. You have been deployed to bring him back to full health before the public finds out, and the North Korean generals depose him! Good luck!

Specific Design: So now I have the theme. But unfortunately, as I found out with Project 4, that does not mean the requirements will be easy to translate into a design. I have to make decisions about the large parts of my program and the small parts of my program. But large-scale decisions may force me to make small-scale decisions that I don't want to make. Conversely, small-scale decisions may determine certain factors in my large-scale design. So it's hard to decide where to start.

The safest place to start is with the specific requirements of the program. First, I must have a Room hierarchy, I must have items to interact with, and I must have a Player with a goal to interact with those items in those rooms. In addition, the Rooms must somehow differ. This suggests that the Player should have a Backpack that contains the items, and perhaps has a menu for interacting with these items. It also suggests that the items should have subtypes - items that

can be picked up and items that CAN'T be picked up, for example.

These considerations lead me to make my first few design choices:

1. The program will center, not on the Room types, but on the item types. There will be a single Object class and then four derived classes from Object:

   (a) Permanent - these objects cannot be picked up, but they can be interacted with, presumably by an object that the Player has equipped, like in a typical RPG.

   (b) Goal - these objects also cannot be picked up (perhaps they will be derived from Permanent), and the point of these objects is to allow the player to interact with them to confirm that the Goal has been achieved. This mechanic is sort of like a "save" or "confirm" mechanic that changes the state of the game in a way that will somehow be monitored to determine when the Player has won the game.

   (c) Holdable - these objects can be picked up by the Player and put into the Player's backpack. Perhaps they contain key information that can be read when the Object is examined, or confer some sort of attribute to the Player.

   (d) Usable - these objects are derived from Holdable, but their main point is that they can be equipped. In other words, the Player can use Usable objects to interact with ALL OTHER OBJECT TYPES.

   By making and enforcing the above design choices, we will tentively declare that we will try to do the following things with the Object class hierarchy.

   (a) All Objects will have a name and description that the Player will be able to read through the game, via get/set functions. *In addition*, all Objects will have **map of interactions** and a **key**. The map of interactions will emulate JavaScript's event handlers, in that each pair in the map will map a string to a function. An object's key will be passed to another object, and if that key is found in the other object's map, the matching function will be executed. We know that this is possible with lambdas and the STL in C++11. The question is whether this will work as intended.

   (b) Permanent - this Object will essentially be the same as the parent class. I only make it a base class in case I need to make any modifications to it that I DON'T want to be modified in other derived classes of Object.

   (c) Goal - this inherits from Permanent, and presumably it will also contain strings that will tell the Player who interacts with them whether they have yet succeeded/failed in meeting the goal.

   (d) Holdable - this inherits directly from Object. Since it fits into the Backpack, and the Backpack has a weight limit, the Holdable must have a WEIGHT VARIABLE and the associated get/set functions.

   (e) Usable - this inherits from Holdable. Presumably the only difference is that it contains strings that explain what happens when the Usable object is used.

   Now that we understand the design of the Object class hierarchy in a deeper fashion, we can turn to look at the Player class.

2. (a) The Player class has a Backpack, obviously. It also has a pointer to a Usable object that is the Player's currently "equipped" item. The player should have a function for **viewing** the Backpack, which will give the user a MENU for interacting with the Backpack. Both Player and Backpack should have functions for picking up Holdable objects through a Holdable pointer.

   (b) the Backpack itself will be a separate class that has a container for pointers to Holdable objects, and variables for storing the maximum weight and the current weight. It should also have functions for creating the appropriate menu, displaying it to the user, and carrying out the appropriate actions.

   (c) But that these to the new design question: what should the Player be able to do with the Objects? I choose the following: drop Holdable items, examine Holdable items, equip Usable items, and unequip the currently equipped item. And that's it. Nothing else. The *problem* here is that I have no idea how I'm going to generate such a menu. The menu's size will obviously have to change based on the number *and type* of the objects. That's crazy! I am not sure I can do this using the usual procedural program logic.

(d) To solve this, we will again look to lambdas and STL maps for help. It seems reasonable to me that we would be able to set up several loops that each look at the types of pointers in the Backpack (using dynamic_cast and use them to generate menu choices AND pairs that map menu choices to the correct lambdas that call the right functions.

(e) If the player has an equipped item, it should expose this equipped item with a get function.

So now we have some fairly specific design choices made for the Player, the Backpack, and the Objects. This is going to force some of our decisions for the design of the Room classes and the Game class, which I will presumably need. But I will again start with the more-specific Room class, because I can't design the Game without understanding the Room interface.

Obviously the Room abstract base class will have four Room pointers that point to other Rooms. The Room will also have a container of Object pointers. Somehow the Player will be able to interact with this container to pick up Objects. Since I need 3 derived classes, the simplest way to differentiate them through a virtual function is to separate them by what Objects they are able to hold. So I will have a GoalRoom that can only contain Goal pointers, a ResourceRoom that can only contain Holdable pointers, and a InteractRoom that can only contain Permanent pointers. These divisions are arbitrary, but they satisfy the requirements, so whatever. I need to get coding. The Room also seems to be like the Backpack, in that it should probably be responsible for presenting a menu to the Player that allows the Player to move from Room to Room and interact with each of the Room's objects. So this suggests we need to make the following important choices:

1. The Room will have a menu similar to that of the Backpack. The Menu will be generated using the items currently in the Room, and dynamic_cast will allow us to tell between the Object types and generate a pair mapping each menu choice to a lambda function. We can do this with the STL map. Since the Player interacts with Objects using its equipped Usable, the Room somehow needs to have access to it. Perhaps as a function argument.

2. At the same time, the menu must also allow the Player to move from Room to Room. It makes sense that we would be tracking the current Room using a Room pointer, so perhaps the menu function needs to have a Room pointer parameter as well.

3. The Room should also have a name and description variable for printing to the Player so they know some details about the Room they are in.

Now, finally, we have enough details that it's almost obvious how the Game class will look. Clearly the Game class must contain a Player object, this is a single-player game. The Game must also have a Room* that points to the current Room, two int variables that track the current and maximum number of steps the Player can take, where a step is the Player moving to a different room. Finally, we have two last important design decisions:

1. The Objects and Rooms that are used in the Game will by dynamically allocated in the Game constructor, and the Rooms will also be linked together there.

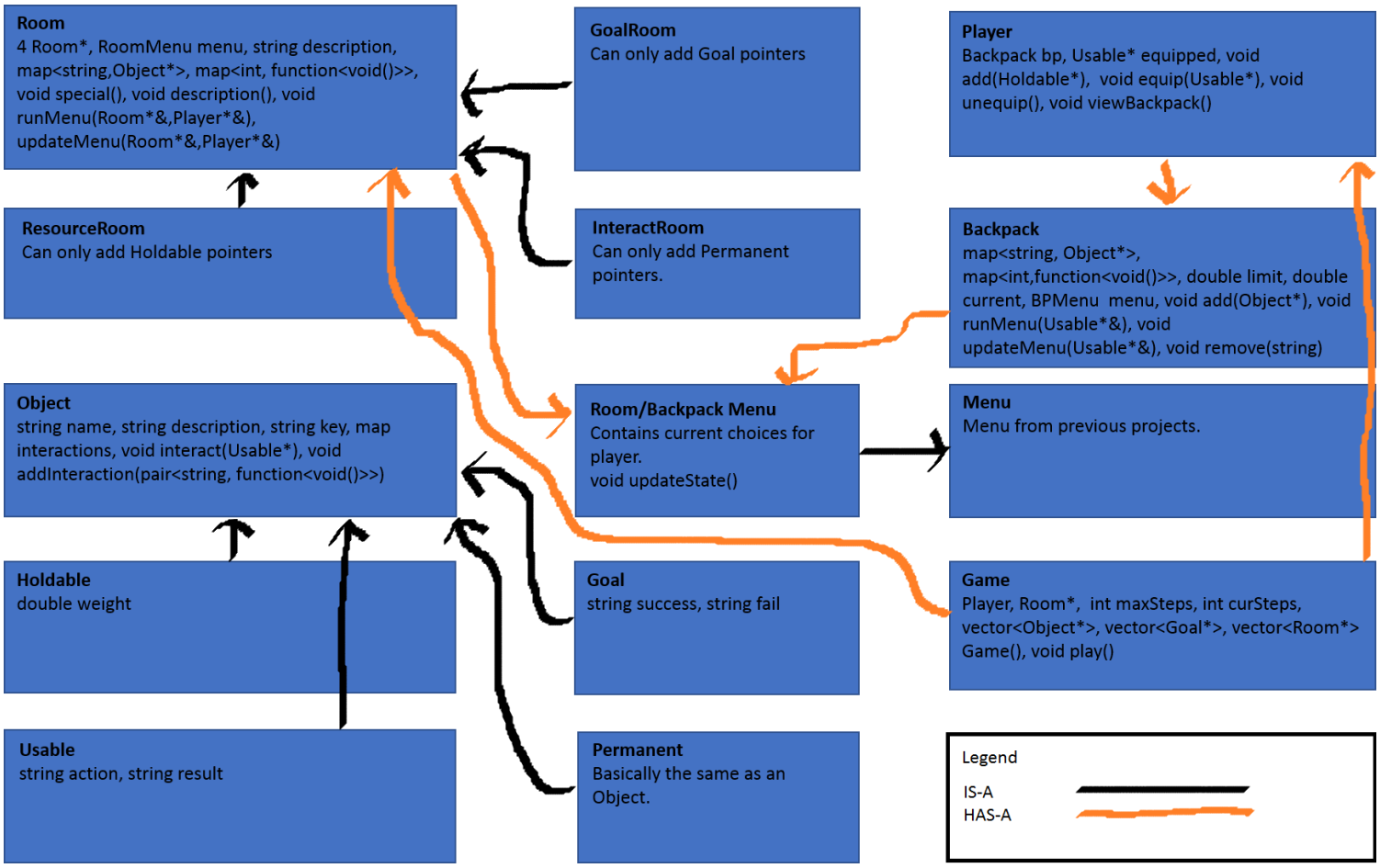
2. The Player will have 3 VECTORS: one for Object pointers, one for Goal pointers, and one for Room pointers, for storing the addresses of the allocated Objects, Goals, and Rooms, so that they can be deallocated before the program ends to avoid memory leaks. This is ideal, because it means the memory is allocated and deallocated within the same 'class scope', which makes me happy.

3. To actually run the game, a Game object will have a play() function that calls the current Room's menu function, which, recall, takes a pointer to the Player and a pointer to the current Room. Ideally, this should basically be all that's needed. From then on, the Room and Backpack's menu should be responsible for getting user input and using it to execute the right LAMBDA FUNCTIONS.

4. The game will continue playing until the current step is greater than the maximum number of steps, or until all the Goal objects in the Goal vector have had their name changed to 'COMPLETED' by the lambda functions in the game.

So there. That's the complete game design. Honestly I'm not sure I can be too confident in it just yet, since I'm relying on lambda functions, which I've read up on but haven't had much chance to use. Speaking honestly, it's weird to generate functions on the fly. I mean, I've done it quite a bit in CS 290, but JavaScript support for functional programming feels a lot more natural.

Class Design Based on Element Dependencies



Class Design of Important Classes

Other classes depend heavily on the following important classes, whose prototypes I give here (Table 1 through Table 5).

Table 1: Room Functions, Room Member Variables

Prototype	Description
void special()	pure virtual function. When overriden, should change what Object* subtypes can be added to the derived class.
void description()	prints appropriate description to console for the player to see.
void runMenu (Room*& ,Player*&)	uses Objects* in the Room to build and show a menu.
void updateMenu (Room*& ,Player*&)	Uses Objects* in the room to build a map from int to function.

Declaration	Description
Room* north	pointer to north room
Room* south	pointer to south room
Room* east	pointer to east room
Room* west	pointer to west room
RoomMenu menu	responsible for handling menu for room. Source of interaction with user.
string description	contains description of room
map <string,Object*> stuff	contains the objects in the room.
map <int,function > actions	map of menu choices to appropriate functions.

Table 2: Player Functions, Member Variables

Prototype	Description
void add (Holdable*)	Allows player to add a Holdable object to his Backpack.
void equip (Usable*)	Allows user to equip one of the Usable objects in his Backpack.
void unequip()	Unequips the user's current equipment, if he has any.
void viewBackpack()	calls function of the Backpack to run the menu.

Declaration	Description
Backpack backpack	Contains Player's Holdable objects, and allow's player to view them and interact with them.
Usable* equipped	Pointer to the Usable object that the Player currently has equipped. Allows user to interact with other objects.

Table 3: Backpack Functions, Member Variables

Prototype	Description
void add (Holdable*)	Adds object to map of Holdable* if weight limit is not exceeded.
void runMenu (Usable*&)	Shows menu and gets input
void updatemenu (Usable*&)	generates menu from the Object pointers in the Backpack.
void remove (string n)	Removes the specified object from the Backpack. Object cannot be recovered.

Declaration	Description
map <string, Holdable*> stuff	contains Player's objects.
map <string, function>	maps menu choice to the correct lambda function.
double limit	upper limit of weight of Holdable objects in the Backpack
double current	current weight of objects in Backpack.
BPMenu menu	Prints player's available options for interacting with the Backpack.

Table 4: Object Functions, Member Variables

Prototype	Description
void interact (Object*)	causes change to Object based on the key in the Object.
void addInteraction (pair<string,function>)	adds an interaction to the map.

Declaration	Description
string name	unique identifier of this object.
string description	description of object. Should suggest its use or clues as to its purpose.
string key	key that identifies how this Object interacts with other objects in the game.
map <string, function> interactions	map of keys to the correct functions.

Table 5: Game Functions, Member Variables

Prototype	Description
void Game()	set up the Game with its network of Rooms, menus, Player, etc.
void play()	Start the game for the user.

Declaration	Description
Player player	Player of the game
Room* room	Current Room of the game.
int maxSteps	maximum steps Player can take before GAME OVER
int curSteps	Current step count
vector <Object*>	Non-Goal Objects in the Game.
vector <Room*>	Rooms in the Game.
vector <Goal*>	Goal objects in the Game.

Testing

For testing, I tested each class hierarchy separately to ensure they worked. The results are as follows:

Class Hierarchy	Driver	Expected	Result
Objects	objectTest()	All Object types should be able to add key-function interactions and successfully interact with those interactions	PASS
Menus	menuTest()	All Menu types should allow choices to be added, and the menus should display correctly	PASS
Backpack	backpackTest()	Holdable* should be successfully added to backpack, menu should display based on current state of Holdable* in backpack, backpack current weight should change and limit what items can be added, maps from menu choices to lambdas should work	PASS
Player	playerTest()	Player should be able to pick up, equip, and unequip items, and access the Backpack menu, where the Backpack menu still works properly.	PASS
Rooms	roomTest()	Each Room subtype should only add objects of the correct subtype. The menus should print correctly based on the objects in a Room, and the menu options should lead to the correct lambda functions. When all the Rooms are linked together, it should be possible to navigate from Room to Room, and run the Player's backpack.	PASS
Game	main()	Game constructs correctly with all the desired rooms. Game plays correctly, where correct text is displayed in every room. Only way to win should be to complete the 3 Goals. Otherwise player loses when step limit runs out.	PASS

Final Reflections

The design in this document is actually my second one. The first one was awful. I hadn't realized that in a program of any complexity, the minute details are just as important as the large-scale design. So, thinking that I was being flexible, I created a design where many of my classes had pointers to Player and Room, creating dependencies that I thought would *help* me by giving me access to any functions I needed. But, unfortunately, all these dependencies meant that my files wouldn't compile! Object needed Room which needed Player which needed Object.... Since the dependencies were blocking compilation, I had to tear up my papers and start anew.

The result of that is this program and this document here. And while I liked my design conceptually, and thought I could implement it, the sticking point was my decision to use a map and lambdas to implement my user-menu interactions. But once I tried programming the Backpack to have these interactions, I found that it was actually pretty easy because the Backpack class had direct access to the container of Holdable pointers. The same thing proved to be true of my Room class menu – much easier than I expected. And then the final issue – could I set up the Game itself?

Setting up the Game turned out to be a matter to trial-and-error: giving Objects and Rooms names, descriptions, and keys until the output looked correct, and linking the Rooms up in an order that made just a little sense with respect to

the human body.

So overall, implementing this design was easier than I expected. Any problems I ran into could be resolved by inspection. And in large part, I think that is because I designed this project from the inside out, instead of the outside in. Working out the inner machinery first made the whole thing easy to do, whereas in previous projects I my design focused mostly on the outside aspects and not on the inner machinery. I hope doing this in the future will improve the quality of my projects.

One issue I still have is this. Most of the functions in my program are void functions, which makes sense since so much of this program is text-based. But at the same time, it makes me wonder how good my design is. Typically, I would think a good program is one that relies on passing information between classes around through function returns and function arguments. But my program doesn't really do that – changes in classes aren't communicated clearly through their own functions; instead they're tied up with console output statements. And the lambdas complicate things even more, since they don't belong to any class at all, but are their own standalone callable objects that get fed into all of my classes. So my design is probably brittle. But at least it was fairly comfortable to code.

Also, despite my planning, there is still an error in my code, which you will notice (I didn't have time to fix this). After you use the calcium to heal Jong Un's cavities, the name of the cavities doesn't change to "beautiful teeth". This is because the way I designed the code doesn't allow the lambdas to access the keys of each Room's object map (nor should it). I was wrong to use maps as each Room's container for Object*. So yeah. Check out the hint map in the program if you get stuck (it's in the intestines).