Christopher Merrill

CS162

12/4/2016

Final Project: Design, Test Plan and Reflection

**DESIGN**

**Overview**

Game Title: Thief’s Dilemma

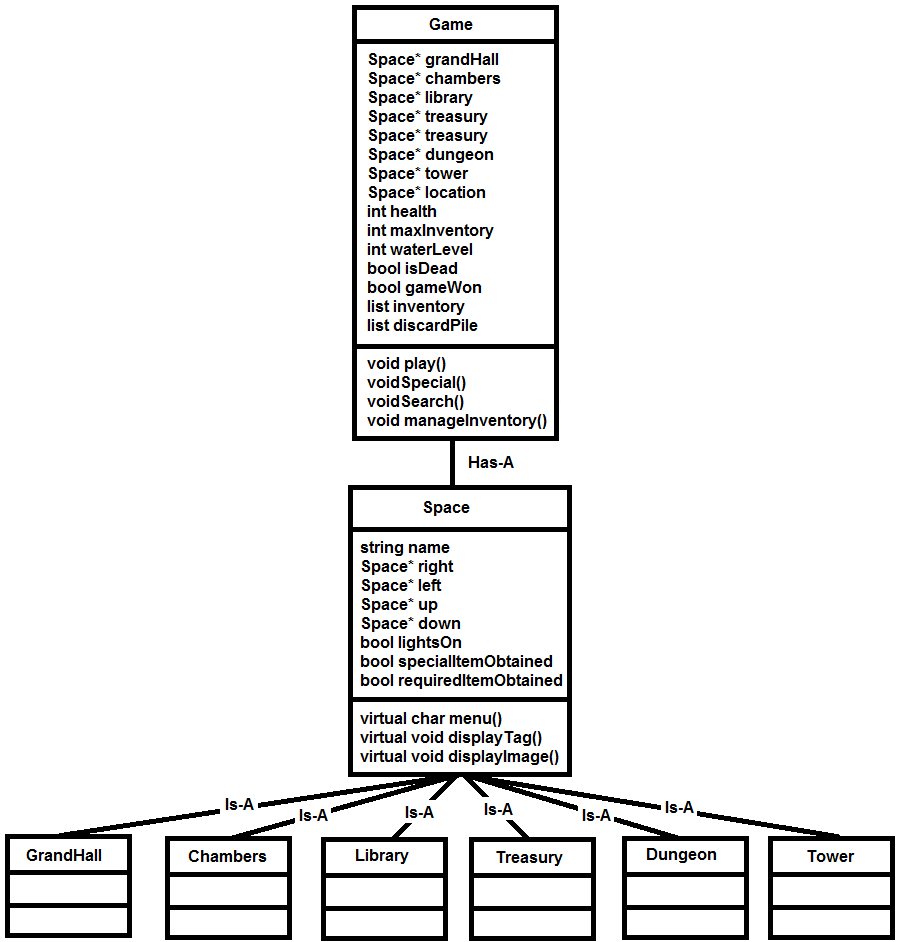
In this game, the user is a thief who takes refuge from a storm in what appears to be an abandoned castle. The castle consists of six rooms: The Grand Hall, the King’s Chambers, the Library, the Hidden Treasury, the Dungeon and the Tower. When the thief enters the castle, he finds out that there is a princess locked away in the dungeon, which is flooding from the storm (the flooding of the dungeon is the time-limit mechanism of the game). The only way she can be saved is if the thief defeats the dragon in the tower to retrieve the dungeon key. The thief must search the castle for a sword and shield so he can face the dragon and save the princess. Along the way false items will be found to throw the player off, as they will not know which items are necessary for the game. The idea being that the thief has to choose between the valuable treasures he finds and the objects necessary to save the princess, he can’t have both (this is the “Thief’s Dilemma”).

Each room will contain an item that is needed to unlock the next room, the solution to the game is as followed:

1. Obtain the mirror from the King’s chambers.
2. Use the mirror to reflect the light from the Grand Hall to light a torch.
3. Use the lit torch to illuminate the dungeon, and obtain the royal jewel from the princess.
4. Use the royal jewel to unlock the Hidden Treasury.
5. Obtain the golden shield from the treasury.
6. Use the golden shield to free the hero’s sword from the encasement in the library.
7. Go to the tower and initiate the fight sequence with the dragon (similar to the die game fight sequences).
8. Obtain the dungeon key by defeating the dragon.
9. Free the princess with the dungeon key.

There will be an abstract base class “Space” from which each of the six rooms of the castle are derived. A Game class will contain all of the game logic and player info.

**Class Hierarchy**

****

\*Note: The Game and Space classes will also need a handful of getter/setter functions that aren’t listed above for the sake of space.

**Game Class**

Data members:

* Six pointers to Spaces for the six rooms of the castle: grandHall, chambers, library, treasury, dungeon tower.
* **location:** a pointer to a Space, representing the player’s current location
* health: The player’s current health, if it reaches zero the game is lost
* **maxInventory:** The maximum number of items the player can hold
* water level: The level of the water in the dungeon, if it gets to high the princess dies and the game is lost
* **isDead:** A bool variable which determines whether or not the princess or the thief has died
* **gameWon:** Bool variable that determines if the game was won
* **inventory:** an STL list container that holds the player’s items
* **discardPile:** an STL list container that holds the items the player discards (for reference)

Member Functions:

* **play:** The main play loop of the game. Utilizes char values returned from the Space class menu functions to decide what actions are carried out in the game. Continues to loop until the game is won, the player dies, the princess dies or the player decides to quit the game.
* **special:** Controls the special actions that occur in each room, like the dragon fight sequence or obtaining special items.
* **search:** Controls the search sequences for each room. Allowing the player to obtain items that may or may not be useful in the game.
* **manageInventory**: Allows the player to remove items when their inventory is full.

**Space Class (abstract base class)**

Data members:

* **name:** The name of the space
* **right:** A pointer to the space to the right of this space
* **left:** A pointer to the space to the left of this space
* **up:** A pointer to the space above this space
* **down:** A pointer to the space below this space
* **lightsOn:** A bool variable which determines if the space has been illuminated
* **specialItemObtained:** A bool variable that determines whether or not the space’s special item has been found (a special item is an item that is required to win the game)
* **requiredItemObtained:** A bool variable that determines whether or not the space’s required item has been found.

Member Functions:

* **menu:** Pure virtual function. Displays the menu when the player enters this space. The menu is unique to each space, as there are different actions the player can take in each space. The menus for one room can also change based on the actions the player has previously taken in the game. Returns a char value that will be used in the Game class **play** function.
* **displayImage:** Pure virtual function. Reads in a .txt file image that is displayed for each space. Some spaces may have different images displayed in different scenarios, for instance if the lights are off or if a special item action has occurred.
* **displayTag:** Pure virtual function. Displays a unique tag line under each space’s image. The taglines sometimes have clues to how to progress in the game.

**TEST PLAN AND RESULTS**

|  |  |  |
| --- | --- | --- |
| **Test** | **Result** |  |
| **Inventory Management:**  Assure that when an item is picked up, it is displayed at the top of the screen in the inventory  Assure that when an item is removed, it is removed from the inventory list at the top of the screen.  Assure that the proper item is removed when the user enters its corresponding number | Item name displayed in inventory after an item is picked up: Yes  Item name removed from display when item is removed:  Yes  Proper item is removed: Yes | I decided later to add a check so that the player can’t remove an item that is essential to the plot.  This makes the game easier, but it was better than letting the player continue on without an essential item, without having the possibility to win the game. |
| **Item Acquisition:**  Successfully add the items to the inventory, remove them, then try to acquire them again.  It should be the case that when an item is acquired from a particular location, if you try to search that location again it should be empty. Regardless of whether or not you removed the item from your inventory. | **Unlit Torch**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: N/A  **Lit Torch**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: N/A  **Silver Tiara**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **King’s Letter**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **Health Potion 1**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **Health Potion 2**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **King’s Robes**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **Royal Jewel**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: N/A  **Hero’s Sword**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: N/A  **Gold Figurine**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **King’s Chalice**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **Giant Ruby**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: Yes  **Golden Shield**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: N/A  **Dungeon Key**  Successful Add: Yes  Successful Remove: Yes  Can’t Acquire Again: N/A | No adjustments necessary |
| **Pointer Test**  Check that the pointers are properly linked so that the player travels to the correct room when they choose right, left, up or down. | **Grand Hall:**  Right to Chambers: Success  Left to Library: Success  Up to Tower: Success  Down to dungeon: Success  **Chambers:**  Left to Grand Hall: Success  **Library:**  Down to Treasury: Success  Right to Grand Hall: Success  **Dungeon:**  Up to Grand Hall: Success  **Tower:**  Down to Grand Hall: Success  **Treasury:**  Up to Library: Success | No adjustments necessary |
| **Room illumination:**  Assure that the library, chambers, dungeon and treasury are dark when the game begins  Assure that illuminate fails when the player doesn’t have a lit torch  Assure that illuminate is successful when the player does have a lit torch. | **Library:**  Starts Dark: Yes  Fails with no torch: Yes  Succeeds with torch: Yes  **Chambers:**  Starts Dark: Yes  Fails with no torch: Yes  Succeeds with torch: Yes  **Dungeon:**  Starts Dark: Yes  Fails with no torch: Yes  Succeeds with torch: Yes  **Treasury:**  Starts Dark: Yes  Fails with no torch: Yes  Succeeds with torch: Yes | No adjustments necessary |
| **Test Health Reduction**  Cat (King’s Chambers): -10 health  Rat (Treasury): -5 health  Bat (Library): -5 health  Assure that at 0 health the player dies and the game ends | **Cat:** Successfully reduces health by 10  **Rat:** Successfully reduces health by 5  **Bat:** Successfully reduces health by 5  When the player’s health dips below 0, the lose screen is displayed and the game ends. | No adjustments necessary |
| **Time Constraint**  Figure out the proper number of moves the player should be able to make before losing:  Determine the minimum number of moves it takes to beat the game (no mistakes)  Have two friends play the game and see how many moves it takes them | **Minimum number of moves required to beat the game:** 21  **Test Subject 1:** 32 moves to win  **Test Subject 2:** 48 moves to win | **Final Decision:** The game should allow for 45 moves before the player loses |
| **Princess Warnings:**  After 15 Moves: Display a notification that the water level is at the princess’ knees  After 30 moves: Display a notification that the water level is at the princess’ chest  After 45 moves: Display a notification that the princess has drowned (the game should end) | **After 15 moves:** Proper notification displayed  **After 30 moves:** Proper notification displayed  **After 45 moves:** Proper notification displayed. The lose screen appears and the game ends | **Note:** The player’s “moves” are referring to actions the player takes. Every action results in another loop of the play function and in each loop the water level increases by 1. |
| **Memory Leak Check:**  Use valgrind to assure that no memory is leaked when exiting the program | **Success.** All heap blocks were freed | No adjustments necessary |

**REFLECTION**

I had to make dozens of minor changes to the design of my program, but overall I was able to stick with the design plan listed above. First off I broke up the “search” and “special” member functions in the game class to be specific to each room, such as searchLibrary or specialDungeon. This didn’t change any of the logic, but it organized my code and allowed for easier debugging. I also included some utility functions in the Game class when I found myself using the same code over and over, such as the resetConsole and pause functions. These significantly reduced the number of lines I had to type out in my Game.cpp file and made formatting the display much easier.

This project more than any other required me to brainstorm every single possibility that may occur while the player navigates the castle, so I truly learned the value of a thorough test plan. I also had to utilize friends as test subjects for the first time. Since I already knew what to do in the game, I would tend to underthink what other players might do who had no reference for what was going on. This helped me not only fix bugs in the game logic, but also find confusing portions of the storyline that needed to be rewritten to help the player move forward.

If I were to start over from scratch, the one thing I would probably do is break up my Game class. It got to be pretty lengthy by the time I was done and it may have been useful organizationally to split it into multiple classes, like a Player class to manage the player’s health and inventory. Overall though I was satisfied with my design and implementation, especially considering it’s the first game I’ve ever created.