Author: Michael S. Lewis
SID: 9329454888
Date: 06/05/2016
Description: CS162 Final Project

**Reflections Document**
This document contains four sections. The first section is an overview of the project, including the design requirements and my own design decisions that were considered for completing the assignment. The second section is a design section describing the design and approach to the assignment, outlined using charts and pseudocode. The third section is a testing plan with regard to an approach for testing the program. The last section is a retrospective of what worked and what needed to be changed from the initial design, as well as examples of some of the debugging that I had to perform in order to make my program work successfully, according to the specified requirements.

## I. Project Overview

The theme of my final project will be **DOS Boat: An Under-c++ Adventure**.

As the user, you will be the captain of a high-tech Digitally Operational Submarine, the SS Babbage, and will navigate the bit depths to find and retrieve three invaluable relics—a priceless Golden Floppy Disk, an exquisite Silicon Orb, and the legendary Apple of Unobtanium. Years of careful research have allowed you to narrow your search to a particular swath of sea, so the game is partitioned into a 2x2 grid of spaces. Furthermore, though you will start your voyage at sea level, you will dive and surface the vessel, making the game board three-dimensional. The surface is the first, top level, with the lower depths divided into two additional levels. Thus, the game board is a 2x2x3 cube of 12 spaces in total. There are also an additional 3 hidden spaces that will become accessible during the game. One space will disappear during gameplay, and will become no longer accessible. To complete the quest, you must traverse these spaces to find the three treasures and then find the secret passage to safety. Some of the spaces may prove for smooth sailing, but other spaces will have different characteristics. Stay alert and on the lookout for sea monsters, hazards, and underwater alien technology, as well as the ruthless Dr. Bug, who will stop at nothing to sabotage you so that he may locate and recover the ancient relics first.

**Other details:**

The user will begin with 10 Strength Points, which will diminish if damage is taken. If strength reaches 0, the submarine is destroyed and the game is over. If more than 25 moves are made, then you will lose to Dr. Bug and the game is over. There will be opportunities to recover Strength Paints to repair damage the vessel might have taken. Within each space, the user will be offered a choice of commands to issue, such as N for north, S for south, E for east, W for west, D to dive down, and U to move one level toward the surface. These commands will be available depending on where on the map the user is located. For example, at a bottom depth space, it will not be possible to dive further. The user may type Q to quit at any time. At the beginning of game play, a brief list of objectives will be displayed and the user will also be offered to view a hint for each space. There will be a Babbage class, which will act as the user's interface with the world. The world will be represented by an Ocean base class, from which there will be several inherited classes.

## II. Design Requirements

**Classes needed:**

- Babbage class
- Ocean base class
  - AwardSpace derived class (**space1, space4, space6, space7, space9, space13**)
  - RecoverSpace derived class (**space3, spaceBonus**)
  - EnemySpace derived class (**space2, space5**)
  - RequireSpace derived class (**space8, space11, space12**)
  - AlienSpace derived class (**space10**)
  - EndSpace derived class (**spaceFinal**)

**Gameboard Layout:**

**Level 1 (ocean surface:**

| | |
|---|---|
| **1** | **2** |
| **3** | **4** |

**Level 2:**

| | |
|---|---|
| **5** | **6** |
| **7** | **8** |

**Level 3:**

| | | |
|---|---|---|
| **Bonus** | **9** | **10** |
| | **11** | **12** |

**Trench Level:**

| |
|---|
| **Final** |
| **13** |

**Space definitions:**

**space1 (Initial starting space)**
AwardSpace (awards sonar device)
connects to Space 2 (East), Space 3 (South), Space 5 (Down)

**space2**
EnemySpace (minefield damages for 3 Strength Points)
connects to Space 1 (West), Space 4 (South), Space 6 (Down)

**space3**
RecoverSpace (recovers 3 Strength Points)
connects to Space 1 (North), Space 4 (East), Space 7 (Down)

**space4**

AwardSpace (awards cache of depth charges)
connects to Space 2 (North), Space 3 (West), Space 8 (Down)

**space5**
EnemySpace (colossal squid damages for 4 Strength Points)
connects to Space 6 (East), Space 7 (South), Space 1 (Up), Space 9 (Down)

**space6**
AwardSpace (awards Silicon Orb)
connects to Space 5 (West), Space 8 (South), Space 2 (Up), Space 10 (Down)
Down is no longer available from Space 6 once Space 10 has been visited.

**space7**
AwardSpace (awards key)
connects to Space 5 (North), Space 8 (East), Space 3 (Up), Space 11 (Down)

**space8**
RequireSpace (requires Apple of Unobtanium for move down)
connects to Space 6 (North), Space 7 (West), Space 4 (Up), Space 12 (Down)
Inventory is searched for Apple of Unobtanium. If it is in the inventory, the SS Babbage may move down. Otherwise, that direction is not yet accessible.

**space9**
AwardSpace (awards Apple of Unobtanium)
connects to Space 10 (East), Space 11 (South), Space 5 (Up), Bonus Space (West)
East is no longer available from Space 9 once Space 10 has been visited.
West becomes available from Space 9 once Space 10 has been visited.

**space10**
AlienSpace (aliens damage for 5 Strength Points)
connects to Space 9 (West), Space 8 (Up)
once SS Babbage leaves Space 10, all pointers to Space 10 are set to NULL and Space 10 is inaccessible.
once SS Babbage leaves Space 10, Bonus Space to the west of Space 9 becomes accessible.
East is no longer available from Space 9 once Space 10 has been visited.

**space11**
RequireSpace (requires Apple of Unobtanium for move east)
connects to Space 9 (North), Space 12 (East), Space7 (Up)
Inventory is searched for Apple of Unobtanium. If it is in the inventory, the SS Babbage may move east. Otherwise, that direction is not yet accessible.
Dr. Bug is mentioned as present in this cell. No action occurs other than a mention that he has been neutralized if user possesses depth charges.

**space12**
RequireSpace (requires key for move down)
connects to Space 10 (North), Space 11 (West), Space8 (Up), Space13(Down)
Inventory is searched for key. If it is in the inventory, the SS Babbage may move down. Otherwise, that direction is not yet accessible.

North is no longer available from Space 12 once Space 10 has been visited.

**space13**
AwardSpace (awards Golden Floppy Disk)
connects to Space 12 (Up), Space Final (North)

**Space Final**
EndSpace (ends game)
(no connections as gameplay stops here)

**Space Bonus**
RecoverSpace (recovers 10 Strength Points)
Connects to Space 9 (East)
available only once Space 10 has been visited

**Classes and their members/functions:**
**Babbage:** class for members and actions of the player's character (the SS Babbage submarine)

- **Private**
    1. vector<string> inventory – *to hold inventory items*
    2. Ocean* currentSpace – *pointer to ocean, for current space*
    3. int strength – *for current Strength Points*
    4. bool visited – *to flag if space10 (AlienSpace) has been visited, so it can be unlinked*
- **Public**
    1. Babbage() – *default constructor*
    2. void setSpace(Ocean*) – *to set the space to be occupied*
    3. Ocean* getSpace() – *to return the space occupied*
    4. void addItem(string) – *pushes item to inventory vector*
    5. void showInventory() – *displays inventory*
    6. int searchInventory(string) – *parses inventory for items*
    7. void damage(int) – *deducts Strength Points*
    8. void recover(int) – *recovers Strength Points*
    9. void setVisited(bool) – *sets status as visited (for space10)*
    10. bool getVisited() – *returns status of visited*

**Ocean:** base class for members and actions of the gameboard spaces

- **Protected**
    1. Ocean* north – *for movements north*
    2. Ocean* south – *for movements south*
    3. Ocean* east – *for movements east*
    4. Ocean* west – *for movements west*
    5. Ocean* up – *for movements up*
    6. Ocean* down – *for movements down*
    7. string name – *tracks identity of current space*
    8. string headings – *tracks available directions*
    9. string description – *holds description of current space*
    10. string hint – *holds a hint for the current space*
    11. bool visited – *to flag if space10 (AlienSpace) has been visited, so it can be unlinked*
- **Public**

1. Ocean() – *default constructor*
2. Ocean(string, string, string) – *overloaded constructor, accepts name of current space, its headings, and the type of space*
3. void setspaceHeading(Ocean*, Ocean*, Ocean*, Ocean*, Ocean*, Ocean*) – *accepts pointers representing all six directions (North, South, East, West, Up, Down). Each is set to the appropriate pointer for an adjacent room, or the NULL pointer*
4. virtual string getnameSpace() – *virtual function for returning the name of a current space*
5. virtual void playSpace (Babbage*, bool) – *pure virtual function to inform user of their location, status, and provide a hint, if hints are activated*
6. void setHint(string) – *sets the hint for the current space*
7. void displayHint() – *displays the hint for the current space*
8. virtual void setHeading(string) – *sets the available directions that are currently available in a particular space, as will be displayed for the user*
9. virtual string getHeading() – *returns the available directions to be displayed for a particular space*

**AwardSpace:** derived class for gameboard spaces to award an item to the user

- **Protected**
  1. virtual void playSpace(Babbage*, bool) – *virtual function to inform user of their location, status, and provide a hint, if hints are activated*
  2. virtual void award(Babbage*) – *virtual function to inform user of an available object in the current space, so that it can be added to inventory (or not, if already in inventory)*
  3. string awarded – *holds item from current space that will be awarded*
- **Public**
  1. AwardSpace() – *default constructor*
  2. AwardSpace(string, string, string, string) – *overloaded constructor, accepts name of current space, its headings, the type of space, and the item to be awarded for the current space*

**EnemySpace:** derived class for gameboard spaces to deduct strength points from the user

- **Private**
  1. virtual void playSpace(Babbage*, bool) – *virtual function to inform user of their location, status, and provide a hint, if hints are activated*
  2. int damage – *holds number of Strength Points to be deducted by the current space*
- **Public**
  1. EnemySpace() – *default constructor*
  2. EnemySpace(string, string, string, int) – *overloaded constructor, accepts name of current space, its headings, the type of space, and the number of Strength Points to be deducted by the current space*

**AlienSpace:** derived class for gameboard spaces to deduct strength points from a user. Also helps to facilitate removing of a connected space

- **Private**
  1. virtual void playSpace(Babbage*, bool) – *virtual function to inform user of their location, status, and provide a hint, if hints are activated*
  2. int damage – *holds number of Strength Points to be deducted by the current space*
- **Public**
  1. AlienSpace() – *default constructor*

2. AlienSpace(string, string, string, int) – *overloaded constructor, accepts name of current space, its headings, the type of space, and the number of Strength Points to be deducted by the current space*

**RecoverSpace:** derived class for gameboard spaces to recover Strength Points for the user

- **Private**
  1. virtual void playSpace(Babbage*, bool) – *virtual function to inform user of their location, status, and provide a hint, if hints are activated*
  2. int recover – *holds number of Strength Points to be recovered by the current space*
- **Public**
  1. RecoverSpace() – *default constructor*
  2. RecoverSpace(string, string, string, int) – *overloaded constructor, accepts name of current space, its headings, the type of space, and the number of Strength Points to be recovered by the current space*

**RequireSpace:** derived class for gameboard spaces to recover Strength Points for the user

- **Private**
  1. virtual void playSpace(Babbage*, bool) – *virtual function to inform user of their location, status, and provide a hint, if hints are activated*
  2. virtual void nextSpace(Babbage*) – *virtual function to search through inventory for item required by current space. Function checks for next move entered and prevents move to that space if the required item is not in current inventory, otherwise sets SS Babbage to next space*
  3. string required – *holds name of item required for move*
  4. string restricted – *holds name of space that needs to be accessed with item*
- **Public**
  1. RequireSpace() – *default constructor*
  2. RequireSpace (string, string, string, string, string) – *overloaded constructor, accepts name of current space, its headings, the type of space, name of item required for move, and name of space that needs to be accessed with item*

**EndSpace:** derived class for gameboard spaces to allow game to end upon completion

- **Private**
  1. virtual void playSpace(Babbage*, bool) – *virtual function to inform user of their location, status, and provide a hint, if hints are activated*
- **Public**
  1. EndSpace() – *default constructor*
  2. EndSpace (string, string, string, string, string) – *overloaded constructor, accepts name of current space, its headings, the type of space. Program exits, returning 0 at end of call*

**Pseudocode:**

- Display splash screen – note image from splash screen was from a free clipart in the public domain < http://www.clipartlord.com/wp-content/uploads/2015/04/submarine9.png> I then converted this to ASCII using a free method provided at < http://www.text-image.com/convert/ascii.html>
- Press enter to continue
  - cin.get
- Display game objective

- Instantiate all spaces as outlined in space definitions
- Set linked rooms (available headings) for each ocean space, as indicated in map of spaces
  - Links will be changed later for space8, space9, and space12, once space10 has been visited.
- Ask user if they would like to view hints for each class
  - If yes, set displayHints to true.
- Display basic objectives of game for user.
- Press enter to begin
  - cin.get
- Set initial starting space to space1
- Run game routine as long as SS Babbage has Strength greater than 0 and moveCounter does not exceed 25
- While loop gets current space, and returns all information for current space, including space name, description, and status (Strength points lost/gained, item awarded, item needed, available directions, and any direction dependent on having an item).
  - Inventory is displayed when a restricted direction depends on an item
  - Inventory is then searched to look for the appropriate item, if found, restricted direction is allowed.
  - Item is added to inventory in an AwardSpace. Inventory is searched, so that an item cannot be added twice. All six items must be in inventory to win the game, including:
    - Sonar device
    - Depth charges
    - Monochrome Orb
    - Key
    - Apple of Unobtanium
    - Golden Floppy Disk
  - Restricted areas are dependent on having certain items, as outlined in the gameplay map.
  - EnemySpaces and AlienSpaces deduct Strength Points. If this drops to 0 or below, game ends.
  - RecoverySpaces may restore Strength Points. The maximum of 10 cannot be exceeded.
  - moveCounter is incremented
  - If space10 has been visited and user leaves it, unlink space10 from space8, space9, and space12, and set the new appropriate available directions for those spaces, including adding accessibility to the bonusSpace from space9.
- Test for gameLost status
  - If gameLost status returns true and aliveStatus returns as false, return that SS Babbage has been destroyed, and game ends.
  - If gameLost status returns true and aliveStatus returns true, return that user has run out of time, Dr. Bug wins, and game ends.
  - If space10 has been visited and user leaves it, unlink space10 from space8, space9, and space12, and set the new appropriate available directions for those spaces, including adding accessibility to the bonusSpace from space9.
- Otherwise game ends once endSpace is encountered (which returns 0) or by typing 'Q'
- Delete all spaces (pointers to Ocean)

## III. Testing Plan

Bugs might always exist deeper, manifest themselves more subtly, or otherwise be harder to find than first assumed. For this, it is essential to judiciously plan out testing, using test cases that may be more likely to reveal issues. Exhaustive testing of all permutations may be time-consuming or in many cases, impossible. Therefore, by creating sets of test data that yield good code coverage, logic errors within a program can be revealed and debugged before the program is released. It is often useful to begin by checking edge cases and testing inputs at the boundaries of specified ranges for the program.

In a test plan for this program, it will be critical to test the following:
1. Input validation, including rejecting invalid numerical values and characters/strings for numerical input. Input validation testing should also ensure that successive attempts at invalid input, and attempts at invalid input in multiple prompts, are caught, and that valid input is allowed following invalid input. Please see the Input Validation sample test case that follows this test case outline.
2. Testing that names of spaces, descriptions, and items appear as expected..
3. Testing that each space links only to the correct spaces.
4. Testing that any space that requires an item allows access only if that item is in inventory. This must be tested for any space it links from.
4. Testing that inventory items are placed in the inventory stack and displayed in the order in which they were obtained.
5. Testing that user wins only if all objects have been obtained and user reaches the finalSpace.
6. Testing that game ends if user loses all Strength Points or moveCounter exceeds 25.
7. Testing that Strength Points are restored to expected number, and that total never exceeds 10.
8. Testing that user may end game by typing ‘q’ or “Q”.
9. Testing that hints appear if ‘Y’ is selected and do not appear otherwise.
10. Testing that space1 adds sonar device to inventory, and that a second instance cannot be added.
11. Testing that space2 deducts 3 Strength Points upon each visit.
12. Testing that space3 awards 3 Strength Points upon each visit, and total does not exceed 10.
13. Testing that space4 adds cache of depth charges to inventory, and that a second instance cannot be added.
14. Testing that space5 deducts 4 Strength Points upon each visit.
15. Testing that space6 adds Monochrome Orb to inventory, and that a second instance cannot be added.
16. Testing that space7 adds key to inventory, and that a second instance cannot be added.
17. Testing that space8 requires Apple of Unobtanium to move down, and that attempt to move down without is gracefully rejected.
18. Testing that space6 unlinks from space10 if space10 has already been visited.
19. Testing that space9 adds Apple of Unobtanium to inventory, and that a second instance cannot be added.
20. Testing that space9 unlinks from space10 if space10 has already been visited.
21. Testing that space9 links to bonusSpace if space10 has already been visited.
22. Testing that space10 deducts 5 Strength Points upon visit.
23. Testing that space10 requires Apple of Unobtanium to move south, and that attempt to move south without is gracefully rejected.
24. Testing that space10 cannot be accessed after it has been visited once.
25. Testing that bonusSpace awards full Strength Points upon each visit.
26. Testing that space11 requires Apple of Unobtanium to move east, and that attempt to move east without is gracefully rejected.
27. Testing that space12 requires key to move south, and that attempt to move south without is gracefully rejected.

28. Testing that space13 adds Golden Floppy Disk to inventory, and that a second instance cannot be added.
29. Testing that space13 links to finalSpace, that gameplay ends at finalSpace, and program exits.

**Sample test case for input validation (with corner cases)**

| I/O State | Input/Expected Output | Component Under Test |
|---|---|---|
| Expected output: | Splash screen<br>Press the Enter key to continue | Main Menu Display |
| Input: | -1, 0, 6, a | Input validation<br>(invalid values) |
| Expected output: | Invalid characters have no negative effect, these are ignored once the Enter key is pressed.<br><br>Displays game summary<br>Press the Enter key to continue | |
| Input: | -1, 0, 6, a | Input validation<br>(invalid values) |
| Expected output: | Invalid characters have no negative effect, these are ignored once the Enter key is pressed.<br><br>Displays game objectives<br>Press the Enter key to continue | |
| Input: | -1, 0, 6, a | Input validation<br>(invalid values) |
| Expected output: | Invalid characters have no negative effect, these are ignored once the Enter key is pressed.<br><br>Displays initial game screen:<br><br>You are at the ocean surface in the NW, docked at port. Conditions are clear, but hazards can be seen to the east.<br>Hint: Choose a heading or dive to continue upon your journey.<br><br>You have spotted a sonar device in the water.<br>Your submarine's robot arm attempts to retrieve the sonar device.<br>You added the sonar device to your inventory<br>Your inventory contains 1 of 6 items:<br>1: sonar device<br>You may head South, East, or Down.<br>Please select the next move (eg. N for North, D for Dive, U for Up, etc.): | |
| Input: | -1 | Input validation |

| | | |
|---|---|---|
| | | (negative value) |
| Expected output: | That is not a valid entry. Please try again.<br><br>// Displays available moves | |
| Input: | 0 | Input validation<br>(zero value) |
| Expected output: | That is not a valid entry. Please try again.<br><br>// Displays available moves | |
| Input: | 6 | Input validation<br>(positive value) |
| Expected output: | That is not a valid entry. Please try again.<br><br>// Displays available moves | |
| Input: | a | Input validation<br>(incorrect letter) |
| Expected output: | That is not a valid entry. Please try again.<br><br>// Displays available moves | |
| Input: | E | Correct Input<br>(Selects East) |
| Expected output | You are at the ocean surface in the NE. You have a bad feeling about this. The SS Babbage has encountered an active minefield. One has exploded!<br>You lost 3 Strength Points!<br>You have 7 Strength Points!<br>Hint: Avoid this minefield at all cost.<br><br>You can go South, West, or Down.<br>Please select the next move (eg. N for North, D for Dive, U for Up, etc.): | Prompt for naming fighter |
| Verify that direction selection process continues as expected throughout gameplay,<br>with expected directions appearing for each space, and that these change as appropriate after space10 has been visited. Verify that restricted spaces are only accessed from each connected space if the correct item is in inventory. Verify that the bonusSpace becomes available after visiting space10<br>Verify gameplay proceeds and ends as expected at the Final Space. Verify that gameplay ends if gameplay exceeds 25 moves. | | |

## IV. Retrospective:

The planning out of design decisions charts, and pseudocode prior to beginning coding was well worth the effort. The actual programming went somewhat according to what I had envisioned. However, I found it necessary to reduce the scope of some of my intentions as I began coding. I initially wanted to make more spaces, but realized that the complexity of the coding would take me too long with the given time constraints. Apart from a few adjustments that I had not foreseen, there were not many major bug fixes required after coding. I found that the bugs that did appear afterwards have now become much easier for me to track down. From my experience with previous assignments, both this term and last, I have become much quicker at thinking of potential root causes for unexpected behavior.

One adjustment I made as I coded was for tracking whether space10 had been visited. I added a set and get function to the Ocean base class, as well as a bool to track whether or not the space was visited. I then added these functions to a while loop in my main function in order to unlink any spaces connected to space10 if that space was visited.

I also decided to add an extra gamespace as a proceeded, which I titled bonusSpace. This extra room is also dependent on the bool that tracks a visit to space10. Once it returns as true, space9 links to the bonusSpace, so that the user is able to visit this new space and have full Strength Points restored to the SS babbage.

In testing, one unexpected observation was that it was technically possible to visit space12 without having first been awarded the Apple of Unobtanium. This was a bug I discovered late in development, and it was difficult to resolve, since in this case, my implementation allows for each space to either deduct Strength Points or detect of an item is required to move to a restricted adjacent space, but not both. I was able to resolve this simply by making it impossible to visit space12 from space10 by replacing the pointer to space12 for space10's south direction with a NULL pointer. This did not affect gameplay, and prevents the problematic condition.

Finally, the last challenge I faced was making my method for ending the game gracefully. For this, I had originally intended to end the game in space13, but doing so resulted in an extra display of the current conditions and status, which were undesired. For this reason, I added the EndSpace derived class, as well as the finalSpace pointer to Ocean, which is the only pointer that uses that derived class. Once the user has all of the required items, and is in space13, they may proceed north. This is displayed for the user as finding a safe path out of the ocean depths and back to the home base of the SS Babbage, and results in accessing the finalSpace. This finalSpace pointer then calls the overloaded constructor for the EndSpace class, which displays the appropriate results for the user, and exits the program, returning 0.

I found that the most useful aspect in designing the program was mapping out all of the spaces beforehand, and making a chart, as indicated in section II, showing exactly which spaces could be moved to from any given space. While that did not prevent all of the aforementioned challenges, it definitely made coding easier. This assignment also continued to reinforce the importance of having adequate planning and design upon beginning coding. As mentioned, I had initially envisioned a game that had more complexity, including other spaces, items, and features, but once I began designing, I realized just what that would have entailed. Without engaging in the design phase, I would not have properly identified those significant challenges, and would have instead gone into working on the assignment without being aware of the implications. The scope would have been much greater had I not designed first, in which case, the chances for success within the given timeframe would have been dramatically reduced. Fortunately, the lesson imparted over the course of this term, of designing first and only beginning to code once as much of the design phase as possible is complete, proved very valuable for this assignment.