THE LOST EMPIRE

Version 5.0

05/03/2013

GROUP 10 - DANIEL KEASLER, YUE BEN, NIANZU MA

Table of Contents

I.Project Drivers

1. The Purpose of the Project

1a. the User Business or Background of the Project Effort

We will build a 3D thinking game related to domain of scuba diving named The Lost Empire. It will be a treasure hunting game in the deep unknown dangers of the sea. The Player will start out with nothing but a scuba suit and the hopes of finding the ultimate treasure.
Ideally in the final release he game will include Adventure Mode, Online Mode and Creation Mode. In the Adventure Mode, user chooses existing maps in the system to take adventure. In Online Mode, Players can either team up or compete against one another. In creation Mode, users can create maps themselves and share it with others in order to build their own story.
Currently in this document and the first release we only focus on the Adventure Mode.

1b. Goals of the Project

We want to develop a 3D-thinking featured scuba diving game on the PC, eliminating the real environment of scuba, as well as additional gaming elements such as treasure collecting and selling, and equipment upgrading.

2. The Client, the Customer, and Other Stakeholders

2a. the Client

EA is expected to publish the game.

2b. the Customer

Target customers include PC game players and scuba diving fans.

2c. Other Stakeholders

- System designers, developers, testers

Every team member in our group.

- Domain experts

Domain experts will introduce technical details of scuba diving, including the underwater environment, proper equipment, threats and solutions, and so on. Based on his suggestions, system designers design game settings such as available treasure in level

and equipment in store, health constraints within level, game-over conditions, and so on and so forth. Professor John Bell[1] and Mr. Michael Angelo Gagliardi have provided detailed information about scuba diving.

3. Users of the Product

The Lost Empire is aimed at several different types of Players, according to their gaming and diving experience.

Table 1 Users of the Product

User Name	User Role	Subject Matter Experience	Technological Experience	Characteristics
Experienced Scuba Diver	Playing the game	Master	Master	Well-educated, at least mid-twenties
Recreational Scuba Diver	Playing the game	Master / Journeyman	Master / Journeyman	Well-educated, at least early twenties
New Scuba Diver	Playing the game	Novice	Journeyman / Novice	At least early twenties
Experienced Gamer	Playing the game	Novice	Master	Teenager or twenties, probably Male
Recreational Gamer	Playing the game	Novice	Master / Journeyman	Teenager to thirties,
New Gamer	Playing the game	Novice	Journeyman / Novice	Teenager to thirties

II.Project Constraints

4. Mandated Constraints

The Lost Empire is a 3D adventuring game, and it's mandated to be 3D thinking. To this point, we allow the character to see and move 3-dimensionally, under the constraints of how real scuba divers see and move.

[1] Dr. John Bell is a professor in the department of Computer Science in UIC, as well as the president of the Underwater Archaeological Society of Chicago.

5. Naming Conventions and Definitions

Player: the one who is playing the game.

Character: the scuba diver in the game, controlled by the Player.

Level or **Game** or **Environment**: one level implies scuba diving once. Different Levels match different difficulties (with respect to depth, map area, and so on) and featured environment (such as sunk ship, deep sea. shallow water, and so on). For every play, the Player choose one Level and adjust equipment for the character to meet the minimum requirement for the Level, and then starts the game. After successfully diving and collecting Items within Level, the Character exits Level with Items collected.

Breathing Rate: Breathing Rate is calculated within Level. The Character has initial Gas volume and Breathing Rate values while starting the Level. As time goes by, also if the Character meets with Threats, the Breathing Rate values will change. The Breathing Rate affects the speed of consuming Gas. If no Gas left, the Player fails this Level.

Item: treasure or anything that the Character collects within Levels, such as Fish or Pearls in the sea, Gems found in a sunken ship, and so on.

Threat: any threat for Character within Levels, such as big Fish that will attack Character, Traps such as Silt and so on. Character can use Tools to escape Threats, but their Health or Equipment might be affected or damaged.

Store: a Store is provided outside Levels, where player can sell Items for money, and also spend money in buying or upgrading Equipment.

Equipment or **Tool**: equipment for scuba, which includes Scuba Suit, Gas, Knife, Flashlight, Rope, and so on. Player carries his tools in his bag during diving. Player can buy new equipment or upgrade existing equipment in the Store outside Level.

Oxygen Toxicity: the condition in which the partial pressure of Oxygen becomes toxic to breath, causing seizures. This occurs at 1.6 ATA Oxygen (O2). A diver may have a chance to survive if diving with a group of people, but will more often than not die if diving by themselves because the seizure causes them to lose their breathing regulator and get water into their lungs.

Nitrogen Narcosis: the condition where Nitrogen has a similar effect to consuming alcohol.

Decompression Sickness: the condition where the body is breathing gas at a different pressure than the outside environment, and the gas forces itself out of the body similar to opening a plastic bottle of soda.

6. Relevant Facts and Assumptions

6a. Relevant Facts

The game is designed to be virtual reality, and we will eliminate the real scuba diving sports as much as we can. Therefore, there are some basic facts we shall follow:

- The scuba diver can only dive for a limited time in the water, under constraints of their equipment.
- According to the equipment, the diver can dive in different depth in the sea.
- The diver can only carry certain equipment and certain amount of gas with him.
- The diver can only carry certain amount of items with him.

6b. Assumptions

- For each Level, there is a fixed time value (i.e. the initial Breathing Rate of the character over the initial Gas volume) for the Character.
- The Gas volume can only decrease within Level, negative correlated to Breathing Rate, which can increase while coping with Threats. If the Character exit game with Gas greater than 0, he succeeds the Level and keep all Items collected within Level; else he fails, and cannot take Items out of the Level.
- The Character can buy certain amount of certain Gas before Level starts in Store, according to the Level he will choose.

III.Functional Requirements

7. The Scope of the Work

- 3D graphics display
- Geography analysis

To present the scuba diving environment in the sea, including silt, rock, and so on.

- Scuba diver movement

The diver can move forward, or push / pull themselves up, down, left, right, forward-left-up, and so on. He can also turn his head to see different directions.

- Equipment and Tool design

Table 2 Equipment design

Equipment	Property (for upgrading)
Suit	Weight, Strength

Gas	Content, Volume
Knife	Sharpness, Hardness, Weight
Rope	Length, Toughness, Weight
Flashlight	Weight, Brightness, duration

- Threat and escaping design

Table 3 Threat and escaping Tool design

Threat	Effects	Escaping Tool
Stuck in rocks	Breathing Rate, Suit (may be damaged)	Rope, Knife
Silt	Breathing Rate	Rope
Fish attack	Breathing Rate, Suit, Items (may be dropped)	Knife

Any threat can be deadly if the Character has no proper Tool to escape. As long as the Character chooses right Tool accordingly, the system will automatically use the Tool and save the Character from Threat.

A small development team should be able to implement the simulation of character movement in first person view. They should be able to implement real time calculations based on the statistics needed for diving (as outlined in the preliminary first person UI). They should be able to implement inventory maintenance inside and out of levels, tool selection and use inside of levels, and the store and character progression. The adventure level selection could be implemented, but unless someone amongst the team is a graphical programmer, only 1 very simple level is possible to create. Limits could be placed on the complexity of tools and items to simplify development.

8. The Scope of the Product

Our Functional Requirements, as seen below, came mostly from a number of use cases that were created for the initial project proposal. Not all of them translated to Functional Requirements, and these use cases were not updated to reflect the Functional Requirements, but below are the use cases that most of our Functional Requirements came from. We kept them based on the previous, outdated step to talk about how and why we changed them. The other Requirements came from an increased knowledge of scuba diving or from collaboration on the vision of the game.
Basically the game is imitating scuba diving in the real world. Its key functions are:

- Scuba diving supporting, including dealing with dangers, treasure collection;

- User data storage, so as to maintain information for each user.
- Trading, for selling items collected while diving, and buying or upgrading equipment.

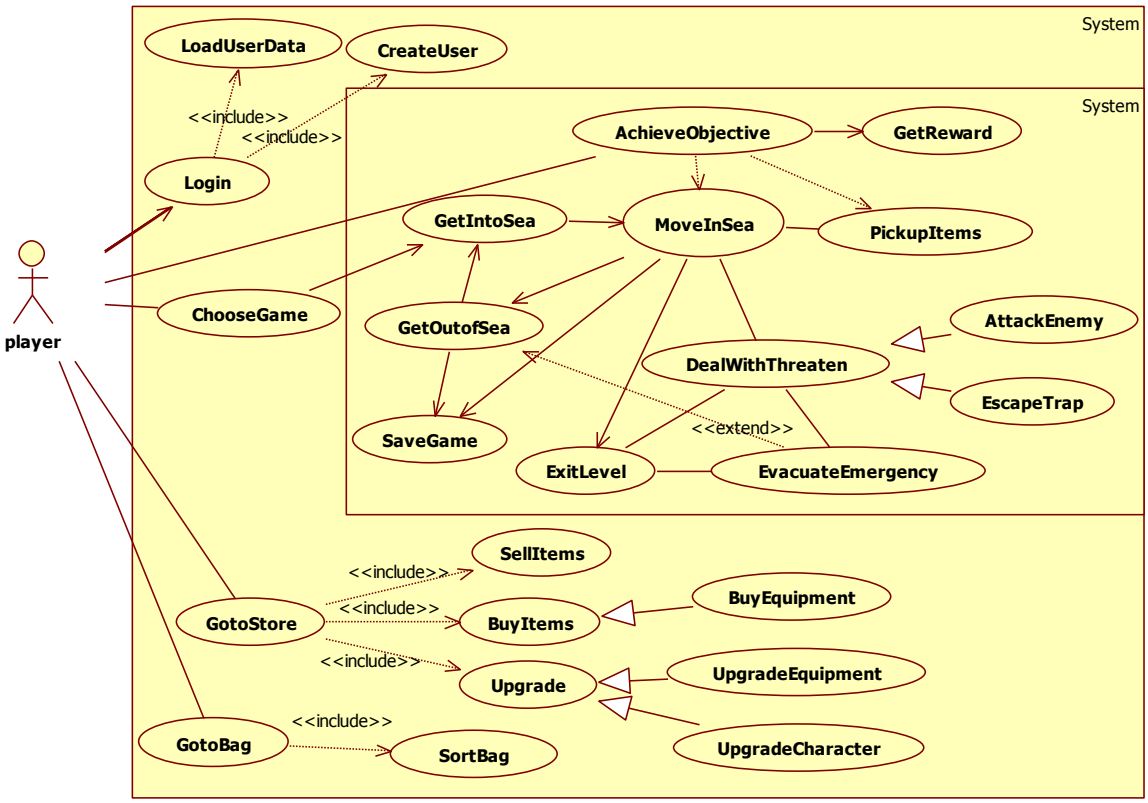


Figure 1 Use Case diagram (Adventure Mode only)

We allow different users to use the program and save their game progress each in a single file. Thus after the game starts, the Player can create a new user, or select his username and load his data.

UC[2] name	CreateUser
Actors	Initiated by Player
Flow of events	1. The Player type userName and password in the system 2. The system check the userName in the existing list and create new account in the game
Entry condition	The Player started the program
Exit condition	The system successfully created the user account, OR

[2] "UC" is for short of "Use Case" while introducing the use cases.

	The userName is identical to some name in the userName list in the system, the player is prompted with suggested new userName according to the name user typed, OR the player is prompted with illegal character in the userName
Note	Included in the Login use case.

UC name	LoadUserData
Actors	Initiated by Player
Flow of events	1. The Player selected a userName to login 2. The system check the userName in the existing list and load the corresponding user data
Entry condition	The Player started the program
Exit condition	The system successfully loads the user data, OR The userName does not match with anyone in the userName list in the system, the player is prompt with error
Note	Included in the Login use case.

UC name	ChooseMap
Actors	Initiated by Player
Flow of events	1. The Player chooses from the available maps listed by the System. 2. The System loads the data of the map.
Entry condition	The Player has chosen the Adventure Mode
Exit condition	The chosen map is successfully loaded for Player, OR The System prompts the reason for fail loading map.

This was intended for our Adventure mode map selection, and has been expanded upon for functional requirement #4 (i.e. functional requirement #4 in Section 9).

UC name	AchieveObjective
Actors	Initiated by Player
Flow of events	1. The player collect some objective during diving underwater

	2. The system check the volume restriction of bag and whether the objective is the kind which should give reward to player when it is collected. 3. The system adds the objective in the users bag
Entry condition	The number of objective in user's bag does not reach the volume limitation. The play is in underwater environment
Exit condition	The objective is successfully added in the bag of player, OR The player is prompt that the bag has reached the volume limitation, OR the objective is not eligible to be achieved because of size or weight.

Bound with the game levels are some Objectives. As long as the player achieves an object, he will get corresponding reward.

UC name	GetReward
Actors	Initiated by Player
Flow of events	1. The Player exits a level in which they completed at least one objective. 2. The system updates the objective storage to reflect the completed objectives and prompts the user of the objective completion and rewards. 3. The Player acknowledges accepting the rewards, as money, new items, or new levels to play. 4. The system gives the rewards to the player, as money, items, or new levels.
Entry condition	The Player exited a level with one or more objectives completed.
Exit condition	The rewards have been added to the Player's game data.

UC name	Save Game
Actors	Initiated by Player
Flow of events	1. Player chooses to save the state of their game 2. System prompts user with confirmation message. 3. Player confirms save operation.

	4. System makes a copy of all game data and saves it on top of the Player's save file. The system then prompts the Player that the save operation was successful.
Entry condition	The Player is in the World Map view.
Exit condition	The system prompts the Player that the save was successful, OR The system was unable to overwrite the save file with the copied game data, and the system prompts the Player that the save operation could not be completed.

The Player can save his game progress at any time.

UC name	PickUpItems
Actors	Initiated by the Player
Flow of events	1. The Player chooses to pick up the item found in the game 2. The System adds the item to the Character
Entry condition	The Character has got into sea.
Exit condition	The Character successfully added the items, OR The System prompts the reason why the Character cannot collect the item

PickUpItems became #20, and was expanded upon to include specific inventory constraints.

UC name	DealWithThreats
Actors	Initiated by the Player
Flow of events	1. Faced with threats in the sea, the Character deals with it through escaping or attack. 2. The System calculates and shows the health status of both the Character and the Threat if it's some entity in the sea.
Entry condition	The Character's health status is not zero
Exit condition	The Character's health status drops to zero which results in his death and this level of the game is over, OR The Character successfully gets rid of the threats while possibly losing some non-fatal health value

DealWithThreats was an abstract use case containing similar elements amongst two use cases. AttackEnemy and EscapeTrap are also referenced below. DealWithThreats is relevant because the concepts of being damaged and/or escaping are still in the game, but increasing the Character's breathing rate due to stress from injury was a more realistic option for our domain instead of health bars and underwater sea battles. DealWithThreats outlined the need for requirements #15 to #18, #21, and #23.

UC name	AttackEnemy
Actors	Inherited from DealWithThreats
Flow of events	1. The Character uses some weapon to attack the enemy 2. The System calculates and shows the health status of both the Character and enemy
Entry condition	Inherited from DealWithThreats
Exit condition	The Character's health status drops to zero and the Character is dead, OR The Character's scuba diving equipment is damaged, the Character has to evacuate the emergency The enemy is killed by the Character

AttackEnemy was deemphasized after gaining more information about the domain of scuba diving. The focus of the game will be about exploration. The user will not have a health bar and instead experience increases to breathing rate. But the user's character will have a number of knives and some blunt objects, and they do have the option of attacking the computer entities if they choose. #7, #17 #18, #21, #22, and #24 are related to AttackEnemy.

UC name	EscapeTrap
Actors	Inherited from DealWithThreats
Flow of events	1. The Character is stuck in the trap 2. The System adds some move restriction on the Character 3. The Character tries to escape the trap 4. The System records the activity made by the Character and compares with the requirement to escape the trap.
Entry condition	Inherited from DealWithThreats
Exit condition	The Character successfully escaped from the trap, OR The health status of Character drops to zero and he is dead

EscapeTrap maintained a similar functionality at this stage, except health bars were removed and the breathing rate concept was used again. Since the diving experience cannot entirely be simulated due to the lack of real physical contact, we decided to include a 3rd person view in addition to the 1st person view. This is a choice; however; 3rd person view does not allow any actions. Rather, it is intended to be used when the user is stuck, so they can see how they are stuck and how to remove themselves. EscapeTrap translated to #12, #15, #16, and #18.

UC name	MoveInSea
Actors	Initiated by the Player
Flow of events	1. The Player chooses the direction 2. The System gets the command and changes the

	position of the Character
Entry condition	The Character is alive
Exit condition	The position of the Character is updated

Our original idea for MoveInSea does not match a realistic scuba diving scenario and has been changed. The basic concept is the same, but we greatly expanded on it. #13 and #14 were created to satisfy the new intent of underwater navigation.

UC name	GoToStore
Actors	Initiated by the Player
Flow of events	1. The Player enters the store sells the items he or she owns in this level 2. The System adds money in the account of the Player 3. The Player chooses the items he needs in the next level 4. The System reduces the money in the account of the Player
Entry condition	The Character is alive and has collected some items in the level
Exit condition	The Player bought some items for the next level, OR The Player sold some items and increased their money, OR The money in the account is not enough to afford the items needed in the next level; the Character goes out of the store without items.
Quality Requirements	At any point during the flow of events, this use case should include SellItem and BuyItem. The SellItem is initiated when the Player chooses the items he wants to sell. When invoked in this use case, the System adds money to the account of the Player according to the value of the items. The BuyItem is initiated when the Player choose the items he wants to buy. When invoked in this use case, the System reduces the money in the account and equips the Character with the item he bought.

GoToStore was a use case that was related to several other use cases, but we collapsed it to one requirement. We realized that we needed a few extra things to make the store more ideal at this stage. The user doesn't necessarily have to sell his collected items, they will have a storage outside of levels. And the user must have some way to gain money without spending money, otherwise they may be too scared to buy anything from the store if all of their money is needed for entering levels. GoToStore is related to #6, #7, #8, #19, and #22.

UC name	Upgrade
Actors	Initiated by Player
Flow of events	1. The Player chooses an upgrade to purchase.

	2. The system prompts the user for a confirmation of the purchase. 3. The Player confirms the purchase. 4. The system validates that the upgrade can be applied to the Player or to an item the Player owns. The system subtracts the cost of the upgrade from the Player's currency, updates the value corresponding to the upgrade, and prompts the Player that the upgrade was purchased.
Entry condition	The Player is in the Store.
Exit condition	The system has prompted the Player that the upgrade was purchased, OR The system prompts the Player that the upgrade could not be purchased, and informs the Player why.
Note	Include upgrade equipment and upgrade character

Inside the Store, the player can upgrade his equipment and so on.

UC name	GotoBag
Actors	Initiated by Player
Flow of events	1. Player chooses to view their bag. 2. System displays the bag
Entry condition	The Player is in the World Map or Game view.
Exit condition	The system displays the Player's bag.

The GotoBag use case allows the player to open his bag and look at his owning, further actions can be taken such as SortBag.

UC name	SortBag
Actors	Initiated by Player
Flow of events	1. Player chooses to sort their bag based on a criteria. 2. System sorts the bag based on the criteria selected.
Entry condition	The Player is viewing their bag.
Exit condition	The system has sorted the items in the bag based on that criteria.

The SortBag use case will allow the player to arrange and sort his bag based on some selected criteria.

UC name	ExitLevel
Actors	Initiated by the Player
Flow of events	1. The Player chooses to exit this level 2. The System prompts for confirmation 3.The Player confirms to exit
Entry condition	The Character is dead, OR The Character is alive
Exit condition	The System save the progression and save the data in the database, OR The System goes back to current level

ExitLevel has been changed slightly for clarity and depth. If the user dies in a level, it is not a proper exit and it is instead a game over. Also, decompression stops change where the exit is. If the user decides to perform decompression stops, then the exit is at the surface instead of a cave entrance. Functional requirements #3, #11, #22, and #23 are related to ExitLevel.

9. Functional and Data Requirements

#1. From the starting menu, the system will provide access to load user profile as well as previously saved adventure modes.
#2. The system will continuously save the states of five adventure modes, and the creation of a sixth adventure mode must overwrite one previously saved state.
#3. The system will save the state of a Player's adventure mode before beginning and upon the Player’s successful exit of each adventure level.
#4. The system will allow the Player to select a level to explore from any available adventure level, but there will be a base cost to explore that level from equipment and gas supply.
#5. The system will unlock additional adventure levels for the Player to explore upon completing the required objective(s) to unlock a particular adventure level.
#6. The system will contain some method to receive money at no original monetary cost.
#7. The system will provide a store outside levels for the Player to receive money from selling collected items and spending money on equipment or character training.
#8. The system will allow the Player to search through their bag, and to select or un-select equipment to carry with them into the next level while they are outside of levels.
#9. The system allows the Player's selection of time to explore each level based on the size of that level, but this selection has a direct relationship to a monetary cost.

#10. system will allow the Player’s selection of maximum depth to dive in each level; however, this selection will alter the cost, an entrance must still be accessible at the selected depth, and the selection cannot exceed the shallowest or deepest points in that level.
#11. The system will allow the Player to choose to avoid decompression stops before they begin each level, but the system will reward the completion of any level with decompression stops.
#12. The system will display either a first person view of each level the Character is currently diving, and will also display statistics about the Character’s current dive (breathing rate, oxygen partial pressure, remaining gas supply or time remaining) and the objective(s) as a projection on their mask, or a third person view of the Character, with no statistics displayed and no allowed actions except to return to first person view.
#13. The system will allow the Character to swim forward, push or pull themselves, reposition their body, control their buoyancy, follow a reel line, or rotate their head subject to normal human being movement capabilities, which will also change the user's location inside and the first person view of the level.
#14. The system will not prevent the Character from navigating too shallowly or too deeply and the system will make the Player’s character feel any negative effects of diving outside the ranges of the at depth gas.
#15. The system will have collision detection between the Character and traps, which themselves are either intentional traps or unstable objects in the levels and can damage or restrict the movement of the user.
#16. The system will allow the Character to free themselves when the Character becomes stuck or buried, possibly displaying the actions freeing the Character in third person view, but this extra use of force will temporarily increase the Character's breathing rate.
#17. The system will control non playable entities that may or may not be able to move around, damage the Character, or be damaged by the Character.
#18. The system will translate Character damage from entities or traps as minor cuts or bruises, and automatically increase the Character's breathing rate proportional to the damage as an increase in stress from the injury.
#19. The system will have collectible items in each level; however, the system will only save those items outside of that level if the Character successfully exits that level. The system will limit the volume of items the Character can carry. The volume of items impacts buoyancy and breathing rate. The Character can drop items, and the system will remove items from level if the Character exits this level with the items.
#20. The system will allow the use of tools that the Character brought into the level, including but not limited to secondary light sources, reels, and knives; all which may

temporarily increase the Character’s breathing rate depending on the tool and how it is used.

#21. The system will force the death of non-playable entities if their health reaches 0.

#22. The system will force the death of the Character if the Character runs out of gas or if they suffer from Oxygen Toxicity or Decompression Sickness.

#23. If the system has forced the death of the Character, the system will stop the Player's current playing session, display the game over screen, and will eventually display the starting menu unless the Player acknowledges the game over themselves.

IV. Nonfunctional Requirements

10. Look and Feel Requirements

10a. Appearance Requirement

1) The appearance of the operation interface should be sports-oriented, dynamic and shows movement. The user interface could integrate the dynamic lines, pictures and shape for every bar, button and background. The dominant hue should be blue, which is the main color the divers would see when they are diving.
2) Mystery and exciting experience should be provided in the game, to attract the target uses – teenagers, young adults, and scuba diving fans.
3) The product should comply with corporate branding standards.

10b. Style Requirement

The style of the software should be authentic, professional and accurate. It should not have too much cartoon or animation elements in the design.

11. Usability and Humanity Requirements

1) Operation conventions of this game for users are compatible with the popular PC games in the market.
2) Users that are not familiar with diving will understand diving concepts through the tutorial to be able to play the game, but it will not substitute real diving training.

11a. Ease of Use Requirement

1) The game should be easy for teenagers older than 11 years old or adults to start after a few minutes’ video tutorial.
2) The system shall provide a scuba diving video tutorial and an operation instructions document, so as to help any user start the game in a few minutes.

3) The system shall provide a naive tutorial/entry level. 70% of test panel of 11-year-old children should be able to successfully complete the first entry level within 20 minutes.
4) The game should attract users from different background (teenager, young adult, with diving knowledge, without diving knowledge) and help them start.
5) Any users that have some experience playing PC games before should have the instinct and be able to operate the game without referring to any instruction or tutorial (basic operations take 70% of the total operation).
6) The game should stimulate users' interest in diving and encourage them to play/practice regularly.
7) An anonymous survey should show that 75% of the intended users usually play the game after 2 hours familiarization.

11b. Learning requirements

The learning process to play the game should be divided into two part and discussed in several situation. First is the process to learn how to operation the character in the game, this should be easy for most game players. Second, because this game is designed in the scuba diving domain, it will require some basic domain knowledge for user to learn, so it takes a little time to learn this part. For different users, the requirement should be specified as below:
1) The game should be quite easily for users with basic domain knowledge to use.
2) For users who do not know the domain, there are entry level and video tutorials for them. Learning process through the help facility provided by the game should be interesting, clear and heuristic.
3) For users who know the basic domain knowledge of scuba diving, they do not need the second learning process and can easily operate the game within 10 minutes. If they usually play PC games, they should have the instinct to know the basic operations of the game, and by a glance at the simple instruction they know all the operation disciplines of the game.
4) 85% of the users in the test panel who do not have any basic knowledge of scuba diving, they shall have the interest to finish the interactive help with patience because the learning process is quite interesting and attractive for them.

11c. Personalization and Internalization Requirement

1)The user interface of the software should at least support if not be optimized for multiple languages, spelling preferences and language idioms to facilitate the users from different countries.
2)The game should supply several kinds of interface with different styles of icons and buttons or other elements to fit the preference of the people of different backgrounds.
3)The game should support customization of short keys for game operation.

4)For people of different gender, character, they can always find the style they like in the kinds of interface provided. However, all kinds of interfaces should comply with the criteria mention in 10. Look and Feel Requirement.

11d. Understandability and Politeness Requirement

1) The game should use both symbols and words if necessary to express items and concepts so that they could be easily understood by scuba divers and also users who do not have basic domain knowledge of scuba diving.
2) For items and concepts illustrated by symbols or pictures (e.g. various equipment), users should know the meaning of difference and know how to choose without seeing the detailed instruction.
There should not be much content in the game frustrating users who do not have much domain knowledge of scuba diving. Most concepts and items should be friendly to all kinds of users, which means they should be either illustrated by symbol and picture, or given detained explanation or both.

12. Performance Requirements

1) The system could automatically adjust the graphic quality according to the hardware of the computer to optimize both image quality and fluency.
2) The response time should be consistent during the game.

12a. Speed and Latency Requirements

1) Any interface between a user and the system of the game should have maximum response time of 0.5 second.
2) The response shall be fast enough to avoid making users feel latency of the game and user should have the feeling of the fluency of the game.
3) When making diving plan for users, the time taken should be within 5 seconds, and should display progress bar for the user.
The response time during playing should give user the feeling that the system of the game is an accurate, fast-responding, and reliable system.

12b. Safe-Critical Requirements

1) For this game is planned to be sold internationally, the product should follow the safety standards of the intended market different country.
2) For China market, the game shall embed the avoid-addiction-system into the system.

12c. Precision or Accuracy Requirement

All values will be calculated to two decimals places. This includes time (likely to hundredths of seconds), depth (likely to hundredths of feet or meters depending on

country), partial pressure of Oxygen (likely to hundredths of ATA), and breathing rate (likely to hundredths of meters or feet cubed per second). The narcotic effect of Nitrogen will also be recorded to two decimal places, but its value is equivalent to feeling under the effects of alcoholic beverages, so it will likely just be represented as a number with a fractional component and no units. All values should also be accurate to within ±.05 of the real time value.

12d. Reliability and Availability Requirement

1) The game should be available for use 24 hours per day, 365 days per year.
2) The game's possibility of failure should be very low and almost can be ignored.

12e. Robustness or Fault-Tolerance Requirement

1)When comes with a crashes caused by software bugs, the user could recover the software by restarting and all the information in the last progress save point could be completely reloaded in the system.
2) If the software is forced to be closed or improperly close, the data and progress information of user should well kept by the system.
3) The cause of crash should be collected by the software and send to maintenance department of this game for future improvement.
The system should be stable even when crash happened it could also give user the feeling that system is reliable for all the data and record is kept properly anytime.

12f. Capacity Requirements

So far our game is stand-alone game, so support a single player is fine in this game.

12g. Scalability or Extensibility Requirements

1) The development company of the game should be able to add new series of adventure mode of different theme/story to the game by adding new package of maps into the system. Such additions do not require modification of existing system.
2) The development company of the game should be able to add new mode to the game such as online mode and creation mode. Such additions do not require modification of existing system.

13. Operational and Environmental Requirements

13a. Production Requirement

1) The software shall be distributed as compressed file (as for the format, use the most popular compression format in windows, Unix-like systems).

2) The software shall be able to be installed by an untrained user without the help of other instructions.
3) The product should be of the size that can fit into DVD.
4) The product should also be downloaded at the website to facilitate the users who do not have CD-driver. And the software can be activated by using the activation key bought be user.
5) The install process may be various in different computer, but the most time taken should within 20 minutes under the minimum requirement of computer hardware.

13b. Release Requirement

1) Because the development department collect data from user each time the software have abnormal behavior, the development department will give maintenance releases every two months if necessary.
2) The users could easily install the fix package of the game with the help of instruction and it would not affect or change the customization or progress records of user in the system.

14. Maintainability and Support Requirements

14a. Maintenance Requirement

The maintenance process of the software should be done by both user and developers. When the software does not work well (crash happened or other bugs), developer remotely collect the error report and develop a fix package and give users announcement. Then user choose to install the released package to strengthen the robustness of the game.

14b. Implementation Environment and Adaptability Requirements

1) The Lost Empire should be written in Java and compatible with any desktop, laptop, or Notebook with Java runtime environment.
2) The operation systems environment should be any popular version of windows operating system (windows XP, windows vista, windows 7, windows 8) and any UNIX operating systems (Mac OS X, Linux, Solaris)
3) Java OpenGL is required for the graphic display.
Below are Minimum requirement and recommended specifications, equivalent hardware are acceptable.

Table 4 Supported OS - Windows

Windows		
	MINIMUM REQUIREMENTS	RECOMMENDED SPECIFICATIONS

Operating System	Windows® XP/Windows Vista®/Windows® 7/Windows® 8 (Updated with the latest Service Packs)	Windows 7/ Windows 8 64-bit with latest service pack
Processor	Intel® Pentium® D or AMD Athlon™ 64 X2	Intel Core 2 Duo 2.2 GHz, AMD Athlon 64 X2 2.6GHz or better
Video	NVIDIA® GeForce® 6800 or ATI™ Radeon™ X1600 Pro (256 MB)	NVIDIA GeForce 8800 GT, ATI Radeon HD 4830 (512 MB) or better
Memory	2 GB RAM (1 GB Windows XP)	4 GB RAM
Storage	25 GB available hard drive space	
Internet	Broadband internet connection	
Media	None for the recommended digital installation	
Input	Keyboard and mouse required. Other input devices are not supported.	Multi-button mouse with scroll wheel
Resolution	1024 x 768 minimum display resolution	

Table 5 Supported OS - Mac

Mac		
	MINIMUM REQUIREMENTS	RECOMMENDED SPECIFICATIONS
Operating System	Mac OS® X 10.7.x (latest version)	Mac OS X 10.8.x (latest version)
Processor	Intel Core™ 2 Duo	Intel Core i3 or better
Video	NVIDIA GeForce 8600M GT or ATI Radeon HD 2600	ATI Radeon HD 5670 or better
Memory	2 GB RAM (1 GB Windows XP)	4 GB RAM
Storage	25 GB available hard drive space	
Internet	Broadband internet connection	
Media	None for the recommended digital installation	
Input	Keyboard and mouse required. Other input devices are not supported.	Multi-button mouse with scroll wheel
Resolution	1024 x 768 minimum display resolution	

15. Legal Requirement

1) User should agree with the liability contract when register the new player in the game. Agree that this game should not be used as the sole source of reference nor should replace the real life training provided by accredited instructors and diving professionals.
2) Users should agree the Released Parties should not be responsible or liable for any death, injury or other damages caused as a result of using the software.

V.System and Object Design

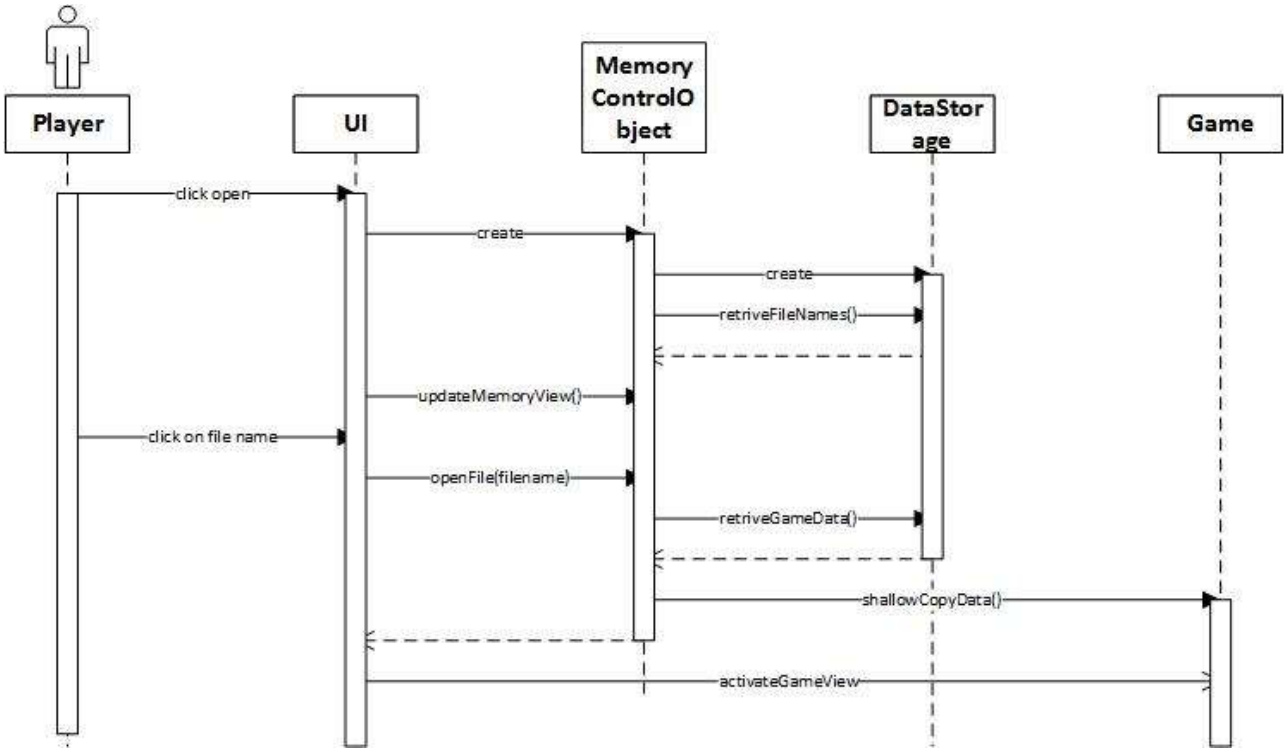
In this section we introduce the system and object design of the project. Its system architecture and subsystems will be explained in detail.

16. Purpose of the System

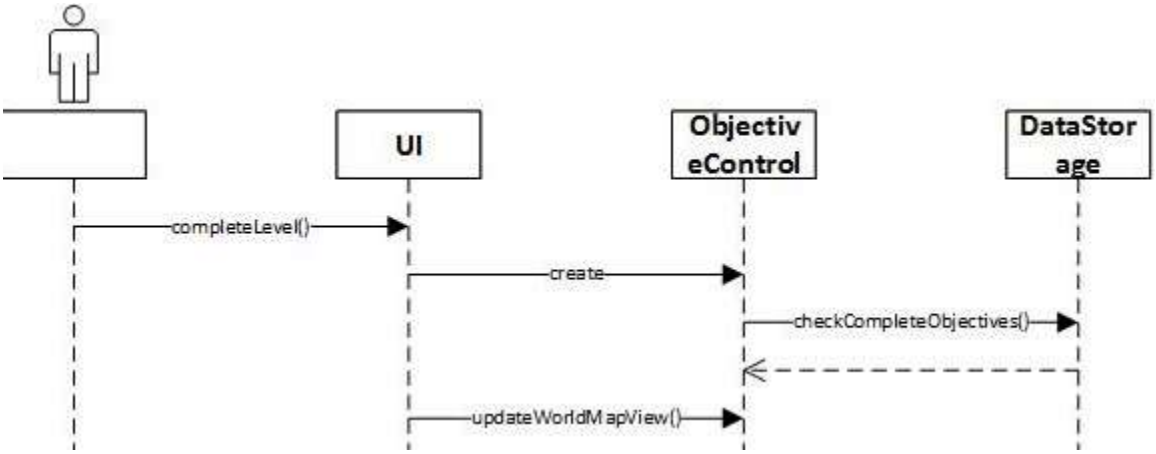
The designed system for The Lost Empire is intended for optimal game play for the user in terms of performance and reliability.

17. Sequence Diagrams

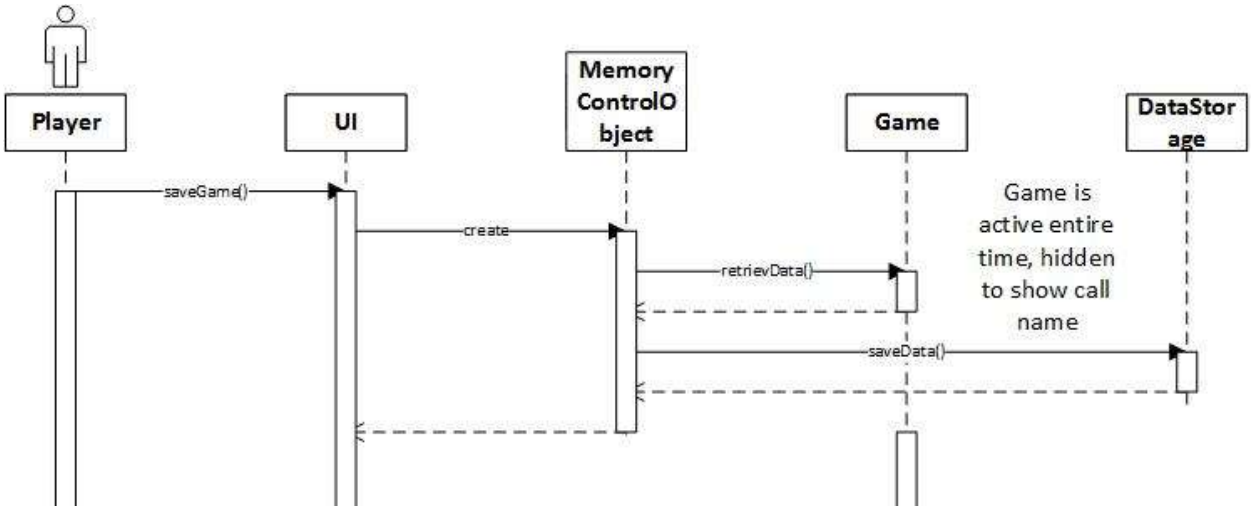
Following is a series of Sequence Diagrams. They were created as a rough illustration of the behavior the system is expected to respond with when the Player interacts with the system.
Many of the class names proposed in the Sequence Diagrams are not included in the object or the system design but the series of actions are included in the system and object design.



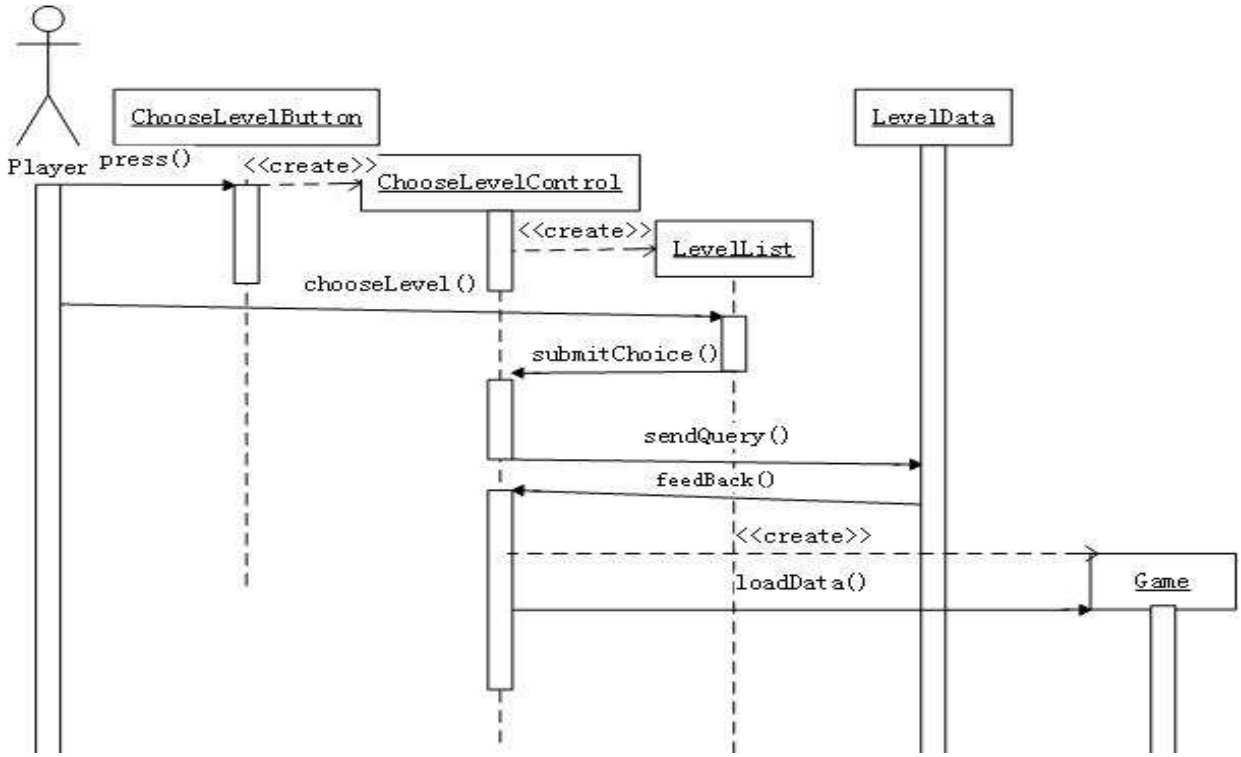
Open Sequence ↑



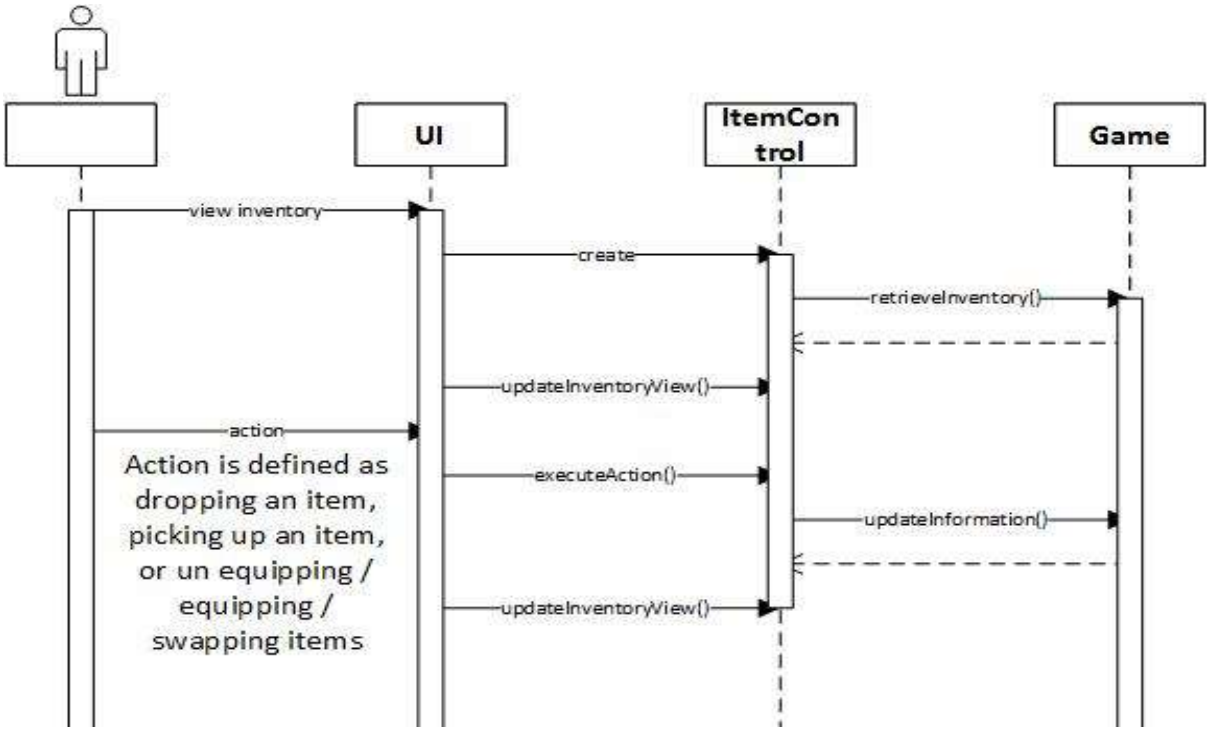
Objective Completion after returning from levels ↑



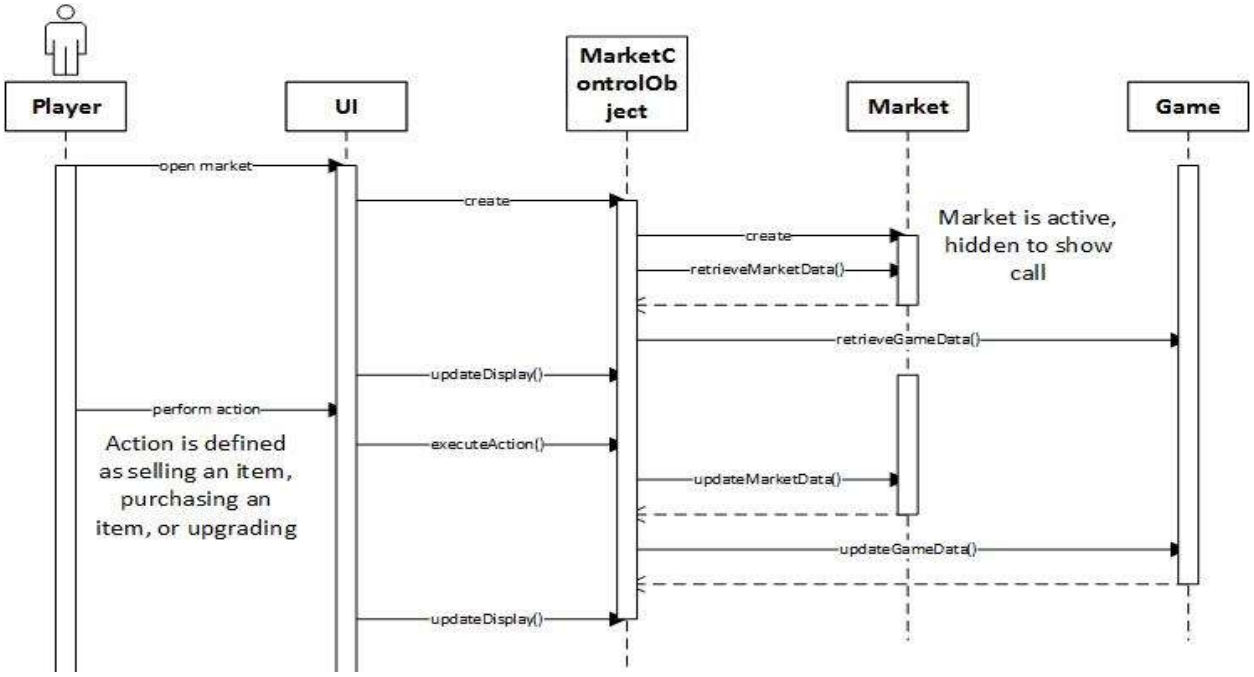
Save Sequence ↑



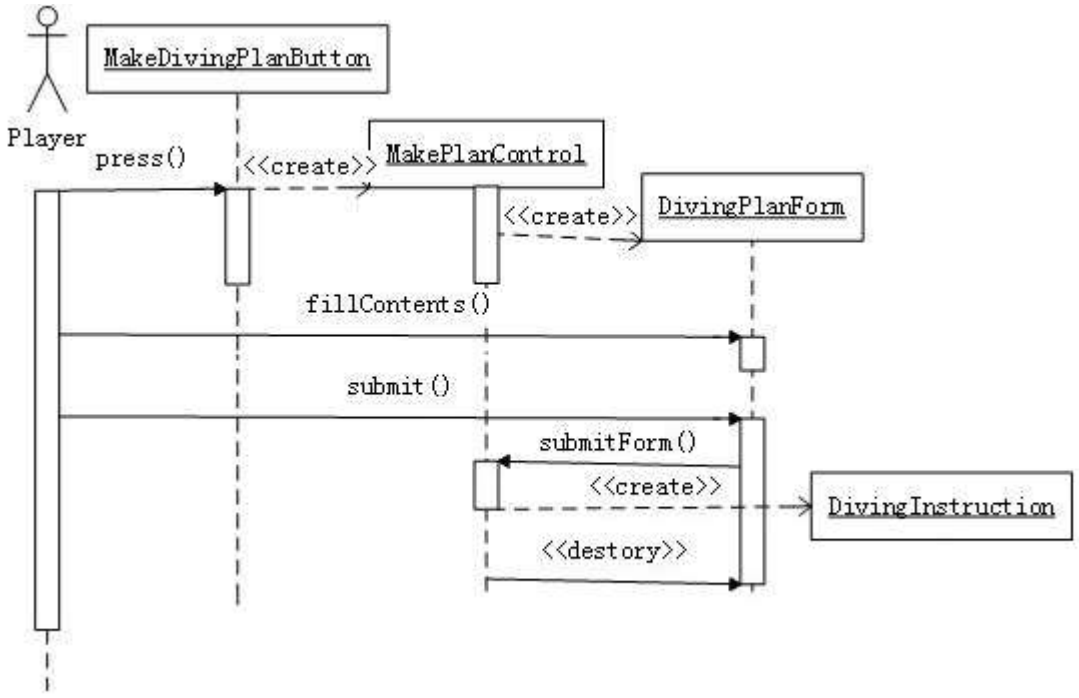
Choose Level Sequence ↑



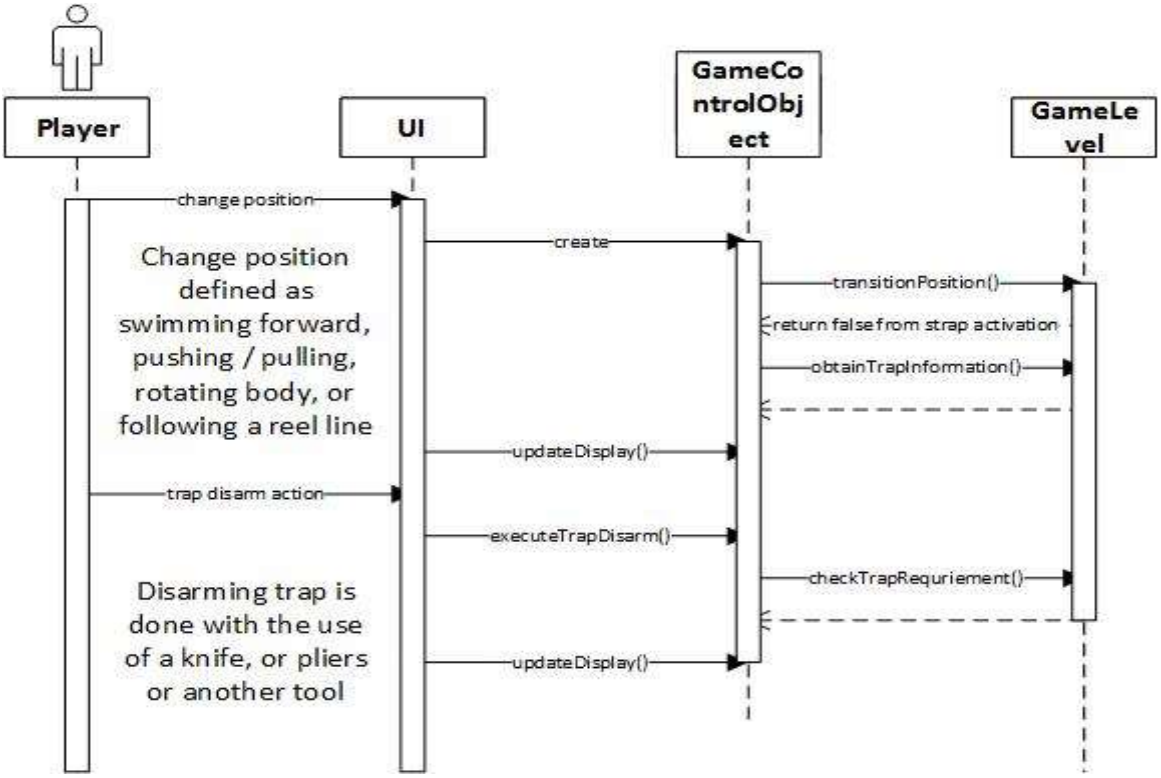
Item Operations Sequence ↑



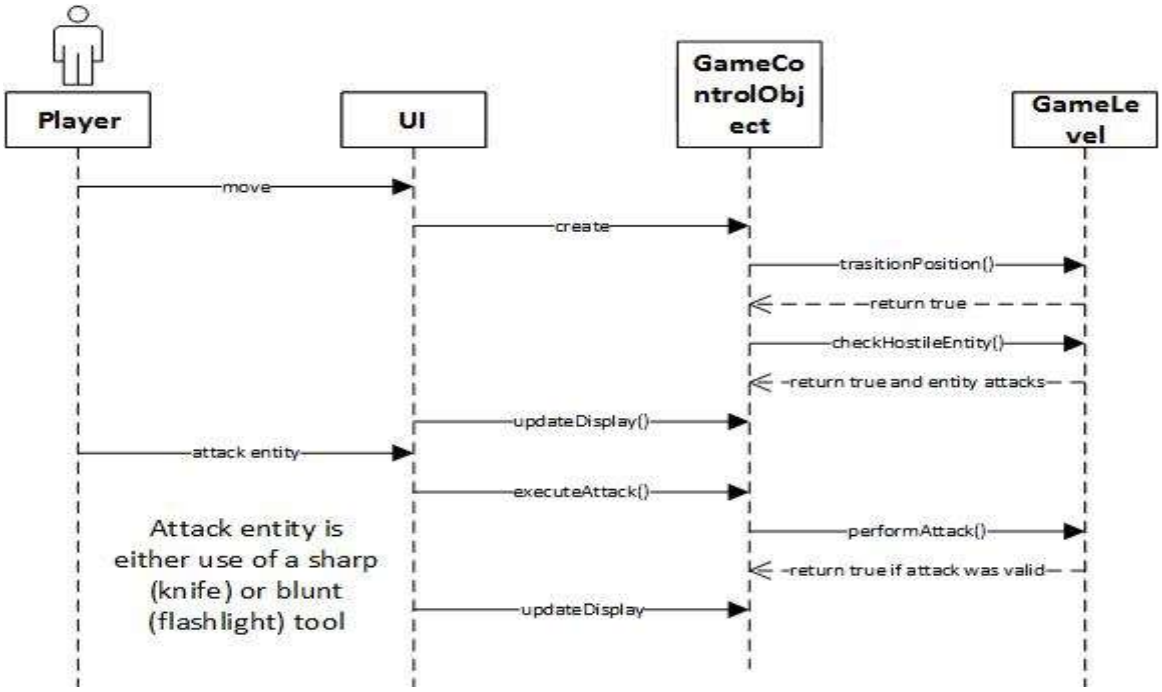
Store Operations Sequence ↑



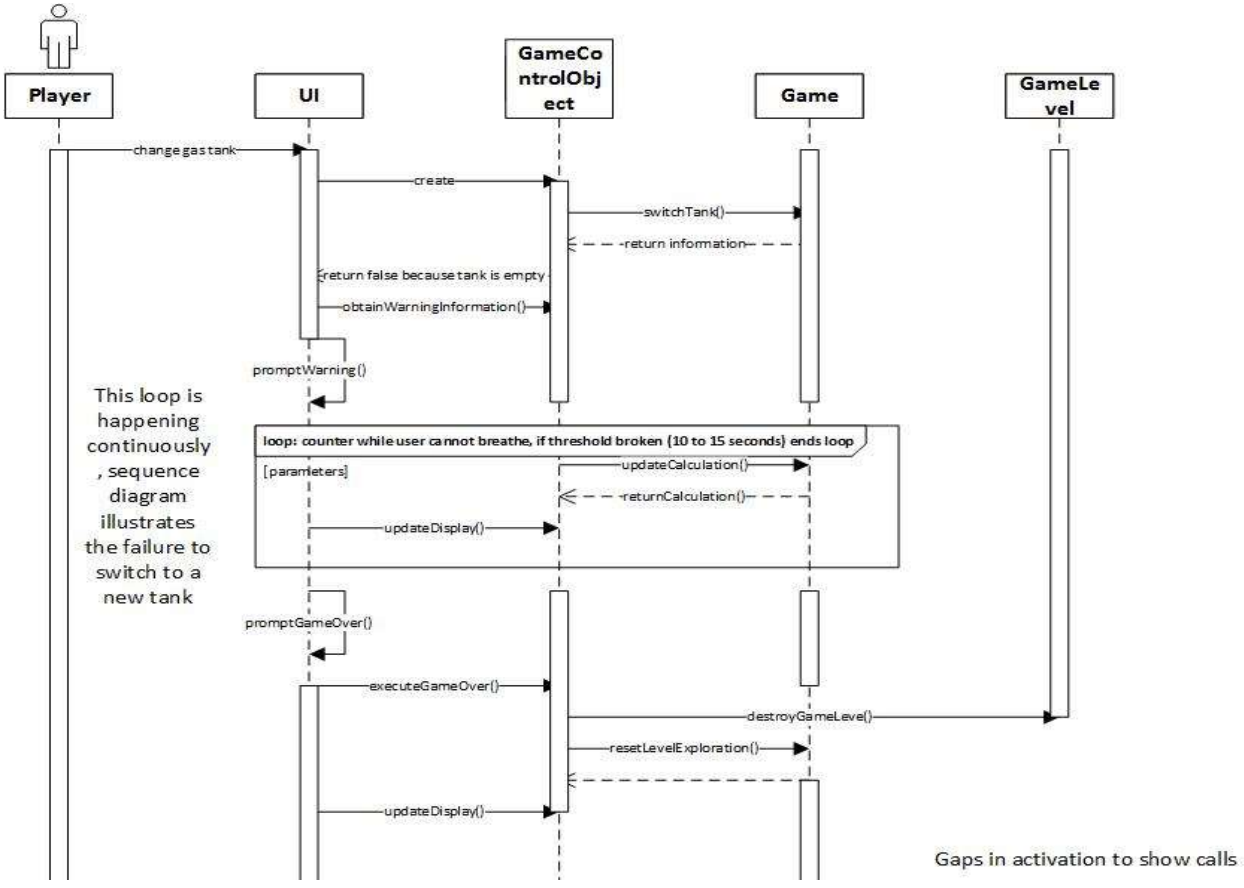
Making Diving Plan Sequence ↑



Trap Operation within Game Sequence ↑



Entity Operations within Game Sequence ↑



Gas Tank Sequence ↑

18. Design Goals

In this section we describe our design goals in performance criteria, dependability criteria, maintenance criteria, cost criteria and the end user criteria. Together with the functional requirement, they describe the overall image of our desired game.

a. Performance criteria

Response time

- Any interface between a user and the system of the game should have maximum response time of 0.5 second.
- The response time should be consistent during the game.
- The response shall be fast enough to avoid making users feel latency of the game and user should have the feeling of the fluency of the game.
- When making diving plan for users, the time taken should within 5 seconds, and should display progress bar for the user.

Graphics performance

- The system could automatically adjust the graphic quality according to the hardware of the computer to optimize both image quality and fluency.
- User can also manually choose the level of Graphics quality.

Memory

- In default state of the system, 1G memory is available for the system to run.
- User can change the configuration for more space for system to use according to their hardware status to have better experience of graphics performance.

b. Dependability criteria

Reliability and Availability Requirement

- The game should be available for use 24 hours per day, 365 days per year.
- The game’s possibility of crashes or interruptions should be very low and almost can be ignored.

Robustness or Fault-Tolerance Requirement

- When comes with a crashes caused by software bugs, the user could recover the software by restarting and all the information in the last progress save point could be completely reloaded in the system.

- When user making diving plan, the system should detect and prompt the user for any invalid input and also give suggestions about what kind of input is valid.
- The user's data created when playing game should be stored in the disk every ten minutes for recover as much data as possible from abnormal state of the software.
- If the software is forced to be closed or improperly close, the data and progress information of user should well kept by the system.
- The cause of crash should be collected by the software and send to maintenance department of this game for future improvement.

Security

- The data of specific user should be safe i.e., It's not allowed one user uses this system to access or change the data of another user.

Safety

- For this game is plan to be sold internationally, the product should follow the safety standards of the intended market different country.
- For China market, the game shall be embedded the avoid-addiction-system.

c. Maintenance criteria

Scalability or Extensibility Requirements

- The system is stand-alone game, so support a single player is fine in this game.
- The development company of the game should be able to add new series of adventure mode of different theme/story to the game by adding new package of maps into the system. Such additions do not require modification of existing system.
- The development company of the game should be able to add new mode to the game such as online mode and creation mode. Such additions do not require modification of existing system.

Modifiability

- The organization and documentation of the game framework should be clear and make it easier for new developers to add features to the code. The source code documentation should support low-level changes and improvements. Also the architecture-level document should support the addition of new features.
- The code should be well structured according to the design pattern during development process. The functionality should be easily modified by modifying only the code closely related to this functionality without affect the functionality of other component.

Readability and Traceability of requirements

- To code should be well commented, structured and documented. Functionally related code should be encapsulated and put in the same package.
- It should take no more than 1 minute to map the code to specific requirement.

Adaptability Requirements

- The Lost Empire should be written in Java and compatible with any desktop, laptop, or Notebook with Java runtime environment.
- The operation systems environment should be any popular version of windows operating system (windows XP, windows vista, windows 7, windows 8) and any UNIX operating systems (Mac OS X, Linux, Solaris)
- Java OpenGL is required for the graphic display.

d. Cost criteria

There'll be development cost, deployment cost, maintenance cost, administration cost, as well as upgrading cost for the whole project.

Development cost

- The initial system should be developed by a small team like four people team to finish in three month.

Deployment cost

- To minimize the cost (including hardware resources, network resources, administration costs) to run this game in most user's computer, we should use free or open-source component during when selecting the off-the-shelf component during develop process.
- The software shall be able to be installed by an untrained user without the help of other instructions.
- The install process may be various in different computers, but the most time taken should within 20 minutes under the minimum requirement of computer hardware.
- The game should be easy for users who have basic domain knowledge to use.
- For users who do not know basic domain knowledge, there are entry level and video tutorials for the user to learn. Training process should be done by users themselves with half an hour.

Maintenance cost

- The maintenance process of the software should be done by both user and developers. When the software does not work well (crash happened or other bugs), developer remotely collect the error report and develop a fix package and give users announcement. Then users choose to install the released package to strengthen the robustness of the game.
- Because the development department collect data from user each time the software have abnormal behavior, the development department will give maintenance releases every two month if necessary.

Administration cost

- This system should be self-administrated, does not need to have other administration activity.

Upgrade cost

- The users could easily install the fix package of the game with the help of instruction and it would not affect or change the customization or progress records of user in the system.

e. End user criteria

Utility

- This game should give user fancy entertainment experience.
- This game should implicitly make user learn basic diving knowledge during playing the game.

Usability

- The Operation convention of this game for user is compatible with the popular PC games in the market.
- The game should be easy for teenagers older than 11 years old or adults to start after a few minutes’ video tutorial.
- The system shall provide a scuba diving video tutorial and an operation instructions document, naive tutorial/entry level so as to help any user start the game in a few minutes.
- The game should be easy for users who have basic domain knowledge to use.
- For users who do not know basic domain knowledge, there are entry level and video tutorials for the user to learn. Learning process through the help facility provided by the game should be interesting, clear and heuristic.

19. Definitions, Acronyms and Abbreviations

Refer to section 3 in Requirement Specification for certain definitions in this project.

No acronyms and abbreviations will be frequently used in our project. They'll be explained if used locally in certain parts.

20. References

Professor Bell and Michael Angelo Gagliardi have provided information and materials regarding scuba diving to make the product more accurate. Some domain knowledge that we have followed is contained in:

- Jablonski, J. Doing It Right: The Fundamentals of Better Diving. Global Underwater Explorers, 2000.
- Odom, J., and Technical Diving International. A Diver's Guide to Decompression Procedures: Theory, Equipment and Procedures. Technical Diving International, 2000.
- International Training, Incorporated, and Scuba Diving International. Sdi Solo Diving Manual. International Training, Incorporated, 2007.
- Ange, M.R. Advanced Wreck Diving. Technical Diving International, 2004.
- Ange, M.R. Diver Down: Real-World Scuba Accidents and How to Avoid Them. International Marine/McGraw-Hill, 2005.

21. Overview

The system is briefly decomposed into User Interface subsystem, Game subsystem, User Data subsystem, and Item subsystem, using an open layered architecture.

The decomposition is basically based on different concepts covered; within each subsystem, there might be structure similar to the three-tier architecture, such as the Game subsystem and the Item subsystem, where operations and data files are separated.

Figure 2 System Architecture below presents the brief system architecture, while detailed introduction for each subsystem will be introduced later in the following sections. In this diagram we contain minor internal information for each subsystem to make it easier to understand the system function.

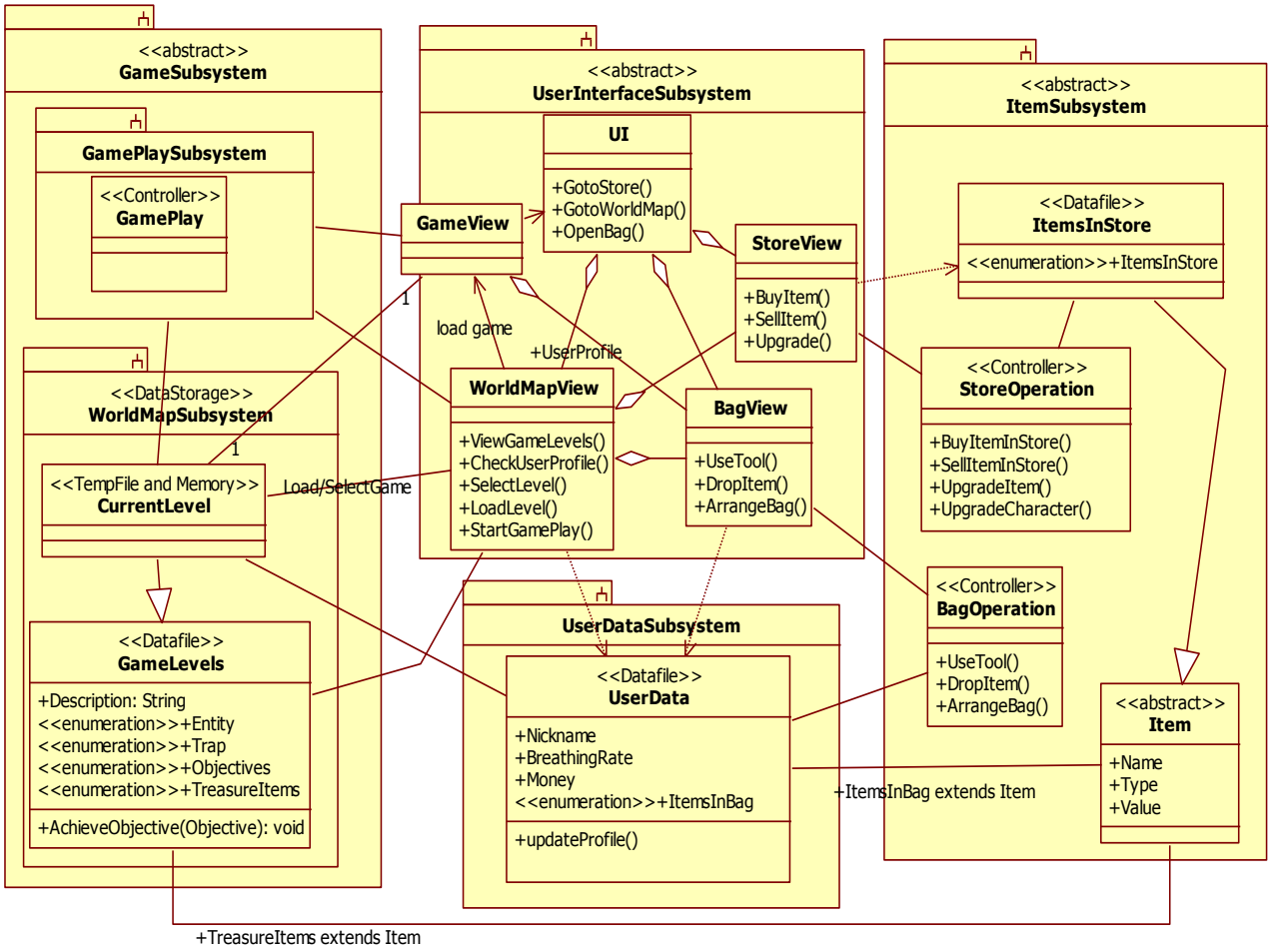


Figure 2 System Architecture and partial Object Design included (outline only)

22. Subsystem Decomposition

In this section, each subsystem will be introduced in more detail, covering their core functions, main components and cooperating subsystems.

a. User Interface Subsystem

The User Interface Subsystem is responsible for the initial game start up, allowing the user to select and load user data file. The Lost Empire will be launched from and be run by the User Interface System.

Mainly user interfaces are included in this subsystem. Each user interface will be linked to its cooperating functional subsystems (including operations and data storage) to make real changes.

When the user comes into the World Map view, he can view and select game level. Based on the user's selection, a filename will be sent to the World Map Subsystem (in

the Game subsystem) for loading the selected game, which will also transfer control to the World Map Subsystem.

The User Interface Subsystem also contains other menu related interaction, such as the Store view and the Bag view. Although the Game view is also included in this subsystem, it's relatively independent, and only used after game loaded.

The World Map view

The World Map view is responsible for game play outside of level exploration. It checks user's ability (interacting with the User Data subsystem) to decide the game levels (interacting with the Game subsystem) to display, allow the user to load saved game or start a new level (interacting with the Game subsystem, also connecting to the Game view). The World Map view also enables the user to interact with the Store and the Bag.

As with displaying game levels for user to select, there are a pool of Levels in the system, with a recommended routine for the surfing order among the Levels, basically according to the difficulties and the player's characteristics. The Player can click and view the description for each Level. After a level is selected, there will be a "start game" button showing up, clicking on which starts the game and transfers the player to the Game view.

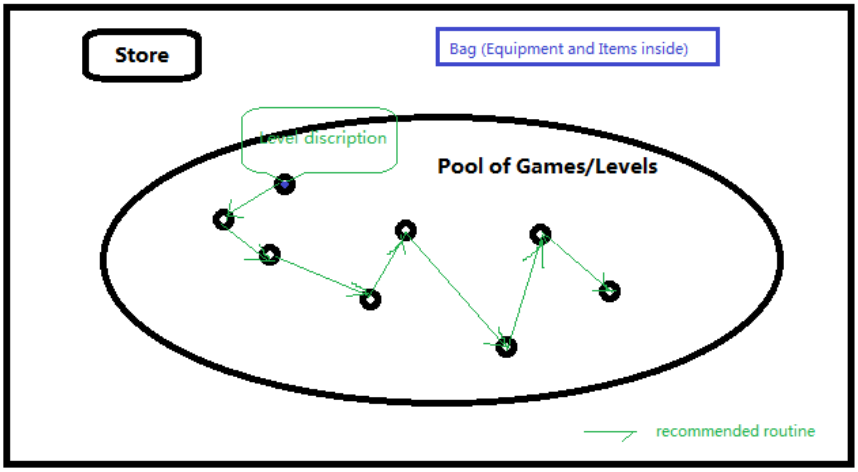


Figure 3 UI – World Map

The Game View (within Game/Level)

The Game view is what the player will see inside a game level.

Figure 4 UI – Game view

A few things to mention not containing letters include the Objective(s) at the top of the screen, any potential Warning Messages related to the information at the bottom, and the blacked out portion on the corners representing a scuba mask.

A: Effect of Nitrogen Narcosis. More knowledge is needed for custom gases, but if Player is breathing air, every 33 feet = 1 Gin, if player is at 66 feet, A would show 2.

B: Oxygen ATA: Player must keep their Oxygen ATA below 1.6 ATA or they will die. Example: If player is at 200 feet depth, 200 / 33 feet = 6.66, + 1 for sea level = 7.66.

1.2 / 7.66 = .157. Player could be breathing 15.7% Oxygen gas mixture at this depth to be safe, and this may be showing 1.2 ATA. However, (1.6 / .157) - 1 = 9.19, * 33 = 303.3 feet. Player would die if deeper than 303.3 feet and breathing 15.7% Oxygen because Oxygen ATA would be at 1.6 ATA and this would cause Oxygen Toxicity.

C: Breathing Rate: More information is needed for specific calculations, but this would be in some volume of gas consumption per second.

D: Time remaining: This would be represented as a bar and with a time in minutes/seconds on the bar. The time remaining would also be based on the time it would take for the gas volume to be consumed at the current breathing rate, and the bar may actually increase if the Player can decrease their breathing rate.

E: Reel line: If the reel line is in sight, the Player will be able to see it in their first person view. At intersections or maybe randomly by the Player's placement, there will be arrows pointing to the exit that they may also be able to see if in their view.

F: Tool in Right Hand: This will be a selectable item based on the user, implementation to select pending, but could be anything including knife, container of gas, reel line, extra flashlight, or their bare hand. Bare hand would be required to follow a reel line on the right side of their body.

G: Tool in Left Hand: This is the same concept as the right hand; however, they may not have access to the same tools with their left as their right depending on the placement of tools on their body.

H: Current depth.

The Bag view

At any time inside or outside the Level, the Player can click Bag and see Equipment/Tools and Items inside.

The basic contents shown in the Bag view are 3 parts: the Carry-on containing items in use (including treasure items and equipment/tools), i.e. that the player is carrying; the Storage containing items the player owns but not carrying; the Money he has.

Both Carry-on and Storage have volume restrictions. For instance, the player can carry at most 5 equipment and 5 treasure items; but he can store 10 each.

The structure of the Bag may look like this:

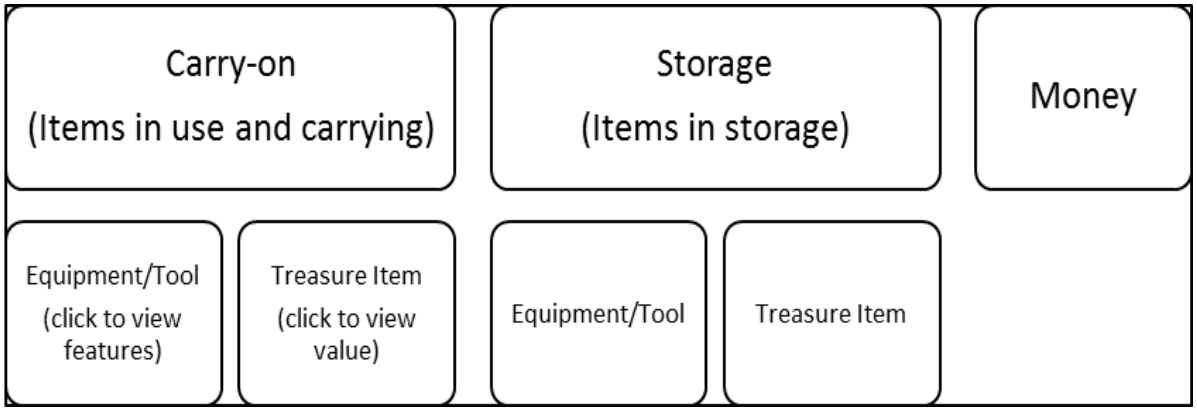


Figure 5 UI - sample Bag view structure

The Store view

At any time outside the Level, the player can go to Store to sell Items, and buy or upgrade Equipment, as well as upgrade his own physique such as breathing rate. The Bag view can be accessed from the Store view.

A sample Store view might be like this:

Equipment for sale	Suite 4 $35	Knife 3 $20	Gas Tank $40			The Bag view
Upgrade	Suit weight -1 $15		Knife sharp +1 $10			
	Rope length +2 $5		GasTank volume+50 $25			
	Breathing -5% $50					

Figure 6 UI - sample Store view

After the Player select some equipment for sale, he can Buy it. The Player can also select one equipment from bag and one corresponding upgrade in store to upgrade his own equipment; the player can also upgrade his own physique by paying for certain “training”. All operations under Bag volume and money constraints.

b. Game Subsystem

The Game subsystem contains two sub-subsystems - the World Map Subsystem and the Game Play Subsystem, and itself has no real responsibilities.

The World Map Subsystem

The World Map Subsystem is mainly in the data storage level, maintaining all game level maps and saved games. Besides, it also contains temporary runtime data between the data level and the function level.

To be more specific, this subsystem is responsible for managing all game data. The Game Play subsystem will either create a new file with a particular filename, load from a file with that filename, or append temporary game to a file, or overwrite a file with the same filename – all the data involved is provided and stored in the World Map subsystem. Note all storage is also on that user’s particular computer.

For each game level, its data file will contain certain information such as game description, and the treasure items, traps and entities within the level, as well as some objectives, achieving which will gain the player corresponding reward.

The Game Play Subsystem

The Game Play Subsystem is responsible for core game play. The parameters of the selected level exploration will be sent from the World Map, and the Game Play Subsystem will be initialized based on the parameters.

The Game Play Subsystem is responsible for translating user commands into game actions; it’s also responsible for real time calculations of depth, ATA, Nacosis, breathing rate, gas consumption, and time left (based on breathing rate and gas consumption) as well as collision detection between entities, traps, and treasure.

The Game Play Subsystem will also save current game if requested, append the user’s saved file if the user exits the level successfully and returns to the World Map view, or it will exit the user’s saved state and return to the User Interface Subsystem

Its direct cooperator is the Game view in the User Interface subsystem, which is responsible for displaying a viewport, or in some cases a 3D view, in real time.

To fully realize the GamePlay class, some assistant classes will be needed:

- Viewport deals with the player's viewport in game, interacting with the Game view.
- CommandInterpreter read player's input and call relevant functions to execute them.

c. User Data Subsystem

The User Data subsystem is responsible for maintaining user profile and game-related user data, such as the player's breathing rate and so on.

Note Items in the player's bag are also stored in this user data file.

The User Data subsystem is in the data storage layer of the overall system architecture.

d. Item Subsystem

The Item subsystem is responsible for Item operations in Store and in Bag – realizing function called in the Store view and the Bag view, also defining the basic attributes in the abstract class Item (not realized), as well as maintaining data files for Items in Store.

Note that we decide to store player-owned Items (that are in his bag) in the player's user data (thus in User Data subsystem), and store Treasure Items in Game/Level data files, thus separated from the Item subsystem, although related.

For example, the Gas Tank below is an equipment item, and it will be stored both in the User Data for player-owned gas tanks, and in the Items in Store data file for store items.

23. Hardware / Software Mapping

The Lost Empire will be written in Java, and would ideally be compatible with any Desktop, Laptop, or Notebook with Java installed. Priority compatibility will be with the more recent versions of Windows: XP, Vista, 7, and 8. Java OpenGL will be necessary to create and display the underwater ruins. (Refer to Section 0)

The *UI component* of The Lost Empire will be implemented by Java OpenGL. Java OpenGL is an independent open source project under a BSD license to be manipulated in Java programming language. Using open source project would cater to our design goal of reducing the operating cost.

The application logic part will be developed in Java to conveniently meet the Adaptability Requirements and reduce the development cost and deployment cost.

Lost Empire will be run directly and exclusively from the user's local machine, which needs to have a compatible version of Java runtime environment being installed.

For the data storage, we'll use data files everywhere, including Items in store, User Data file, Game Level data file, and saved Game Level. Since our data are mostly low coherent, and multiple instances in some places, we prefer file storage to database management such as relational database.

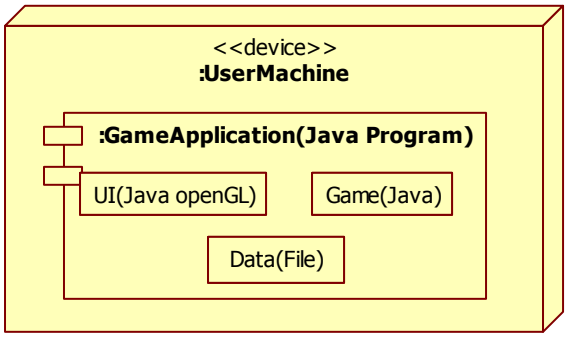


Figure 7 Hardware/Software Mapping Diagram (UML deployment diagram)

24. Persistent Data Management

The user's data and game states are persistent data. The user's information including owned items must be stored in a timely manner. The user must be able to create new game files, load previously saved files, or append / overwrite previously saved files. The state of the game is based on Adventure Mode progress, inventory, and money. Adventure Mode progress itself is based on what objectives the user has completed, and what the user has collected in those levels. Since The Lost Empire will be run on the users local machine, and the save data will not require access from multiple sources, flat files will be used to store the user's data.

To meet the priority design goal of our game which is to minimize the operating costs, we decided to use flat files for storage for many reasons.

- The software could be installed easily, users do not have to install database management system to configure and manage. This also reduces the deployment cost.
- The system is stand-alone game. It manipulates the data of a single player during runtime and it may store several users' data in the same computer. So it is just small sized data set.
- The file abstraction is low level. So it enables our developer to perform a variety of size and speed optimization. It can guarantee we meet the performance criteria of our design goal such as response time and memory restriction.

- Also developers have to deal with many issues, such as concurrent access and loss of data in case of system crash. It is worthy to minimize the operating cost at the cost of improve a limited amount of development cost.
- Since the system meets the design goal that “the game’s possibility of crashes or interruptions should be very low and can almost be ignored”. The chance we meet the loss of data is small. So using flat files for storage is proper.

25. Access Control and Security

The user will be the only actor in the system, and they will have limited access to the system itself. They will be able to see the game states they have previously saved, to save to or load from those files, to view and interact with the Store and their Bag, to select one of the predetermined levels, and to interact with the different objects that are loaded in a particular level exploration during game play.

The user will not be allowed to access the system directly and will not have any more access than outlined above.

26. Global Software Control

The Game Play Subsystem will use an event-driven control mechanism because this is the best way to handle the interaction between all of the various objects involved. The Game Play class will be used to realize the event-driven control, which will include collision events, user command events, as well as real-time calculation based events.

On the other hand, the User Interface Subsystem, Item Subsystem and User Data Subsystem will all have procedure-driven control mechanisms because only one thing will be happening at a time in real time and it is all based on user input. The UI class will include waiting for user input and executing that user’s input.

27. Boundary Conditions

The overall system is started and initialized with the User Interface Subsystem. The Game Play Subsystem is started and initialized based on user input, from a particular save file and parameters of a particular level exploration. The Item Subsystem is always accessed through the UserInterfaceSubsystem.

The UserInterfaceSubsystem will be launched when the system is launched. Initially it will not contain any data, and will be loaded with information when the user loads a save file. The data in the UserInterfaceSubsystem will be saved to file at specific times, and crashes will most likely result in data lost after the previous save. The previous two save files of everything will be updated in case one of the files becomes corrupted. All

specific save operations will be made from shallow copies, so the failed save attempts do not corrupt the original temporary data. The UserInterfaceSubsystem will exit upon user request, which if triggered in the system, will save all of the data in the UserInterfaceSubsystem. The UserInterfaceSubsystem will initiate saving in the Item and GamePlaySubsystems as well.

If the Game Play Subsystem detects that an exception has occurred, it should try to recover first. Loading the map file, calculations and user inputs should be checked again before determining that the game is in an erroneous state. If a recovery cannot happen, then the Game Play Subsystem will suspend the current exploration of a level and return to the World Map, with the previously saved state of the user's game play loaded for them. The byte sequence of a completed level should be copied before being saved so that If an error occurs while trying to append the save file with the completion of a level, the information can be recovered later. The GamePlaySubsystem is always loaded based on user requests, and always exits either from the user exiting from a level or being killed in it.

The Item Subsystem will have similar exception handling as the UI Subsystem. Save attempts are to be made from shallow copies so any errors do not corrupt the temporary data. The two previous versions of the market will be saved, as well as the storage and equipment of the user. The Item Subsystem data will be contained within the ItemSubsystem; however, the UserInterfaceSubsystem will call methods to access the Item Subsystem data as well as to save the Item Subsystem Data. The Item Subsystem itself is always created, accessed, saved, and shut down by the UserInterface Subsystem upon user requests.

28. Object Design

Following is the object design in class diagram form, for each subsystem other than the User Data subsystem where only general operations are needed. This information is subject to change if the developers find it easier to implement the system in a slightly different manner.

a. UserInterfaceSubsystem

UI
-WorldMapView worldMap -BagView inventory -BagView tanks -BagView tools -StoreView market

-GameView currentGame
+goToStore(): boolean//return value for feedback //pre-condition: int param must be nonnegative +viewBag(int): boolean// int param mapped to BagView objects, return value for feedback +saveGame(): boolean +playGame(): boolean //precondition: double params must be nonnegative +buyItem(double, double): boolean//params are coordinates, return value for feedback //precondition: double params must be nonnegative +sellItem(double, double): boolean//params are coordinates, return value for feedback //precondition: double params must be nonnegative +buyUpgrade(double, double): boolean//params are coordinates, return value for feedback //precondition: int params must be nonnegative +arrangeBag(int, int): boolean//params map to bag object and sorting criteria, return value //for feedback +quitGame(): boolean//return value for feedback +goToWorldMap(): boolean//return value for feedback //pre-condition: GameView must not be null +changePosition(): boolean //pre-condition: int param must be nonnegative //pre-condition: GameView must not be null +useTool(int): boolean//int param mapped to index, return for feedback //pre-condition: GameView must not be null //pre-condition: int param must be nonnegative +changeTank(int): boolean// int param mapped to index, return for feedback //pre-condition: GameView must not be null +collectTreasure(): boolean//return value for feedback

BagView
-BagOperation bagReference
BagView(File, int)//params File for parseable Item information, int byte count to start of //Item parsing //pre-condition: int param is nonnegative +UseTool(int): boolean//int param as index mapped to key controls, return value for feedback //pre-condition: double params are nonnegative +DropItem(double, double): boolean//coordinate params of event, conversion to index for //bagReference, return value for feedback //pre-condition: Item must not be null +AddItem(Item): boolean//Item to be added //pre-condition: double params are nonnegative +ArrangeBag(double, double): boolean//coordinate params of event, conversion to int for //bagReference, return value for feedback //pre-condition: String param must not be null +saveBag(String): boolean//String filename to save to, boolean for feedback //pre-condition: userBag.size() != 0 +getBag(): Bag//return users bag

StoreView
-StoreOperation storeReference
//pre-condition: File must point to real flat file StoreView(File)//param is reference to parseable File for StoreOperation //pre-condition: both double params must be nonnegative //invariant: storeReference must not be null +BuyItem(double, double): boolean//coordinates as params, return value used for //feedback to user //pre-condition: both double params must be nonnegative

//invariant: storeReference must not be null +SellItem(double, double): boolean//coordinates as params, return value use for //feedback to user //pre-condition: double params must be nonnegative +Upgrade(double, double): Item//coordinates as params, return Item to append //corresponding value //pre-condition: String must not be null +saveStore(String): boolean//param filename to write to, return value for feedback

WorldMapView
-List<String> mapNames -List<Objective> completedObjectives -Map currentMap -double breathingRate -String Nickname -int money
//pre-condition: File must be real reference to file WorldMapView(File)//File reference to parseable file with character information and //completed objectives +ViewGameLevels(): boolean//return value for feedback +CheckUserProfile(): boolean// return value for feedback //pre-condition: int param must be nonnegative +SelectLevel(int): String//int param is index to mapNames, return String of Filename +LoadLevel(String): Map//string filename as param, returns Map parsed from datafile //post-condition: GameView is not null +StartGamePlay(): GameView//return instantiated GameView to UI //post-condition: String param is name of valid file

+saveGame(): String//return value for other classes to save to file with name string

GameView
-GamePlay currentGame
+GameView(Map, double, Point, Point)//double for breathing rate, two points for //currentLocation and focalPoint +changePosition(): boolean//return boolean or feedback //pre-condition: int must be nonnegative +useTool(int): boolean//int index for tool, return param for feedback +collectTreasure(): boolean//return value for feedback //pre-condition: int must be nonnegative +changeTank(int): boolean//int param for index to use took, return value for feedback +getItems(): List<Item>//returns list of collected items in the level

b. ItemSubSystem

StoreOperation
-List<Item> storeItems
//pre-condition: File must be reference to a real flat file StoreOperation(File)//binary file containing parseable Item information //pre-condition: int parameter must be nonnegative //invariant: storeItems.size() must be positive //post-condition: storeItems.size() decremented by 1 +BuyItemInStore(int): Item//param is index to Item //post-condition: int return value must be nonnegative //invariant: storeItems.size() must be positive //post-condition: storeItems.size() incremented by 1 +SellItemToStore(Item): int//param is Item to add to store, returns items worth //pre-condition: storeItems.size() != 0 +Upgrade(int): Item//param is index in storeItems, return Item

//pre-condition: storeItems.size() != 0 +saveStore(String): boolean//String for filename to write to,, boolean for user feedback +checkStoreSize(): int//return size of storeItems

BagOperation
-List<Item> userBag
//pre-condition: int param must be nonnegative BagOperation(File, int)//File reference to parseable file with item information //int for byte count to start parsing Item information //pre-condition: int param must be nonnegative //pre-condition: userBag.size() != 0 +UseTool(int): boolean //pre-condition: int param must be nonnegative //pre-condition: userBag.size() != 0 //post-condition: userBag.size() decremented by 1 +DropItem(int): boolean//param index to userBag, return boolean for feedback //pre-condition: Item must not be null +AddItem(Item): boolean//Item param to be added //pre-condition: int param must be nonnegative //pre-condition: userBag.size() != 0 +ArrangeBag(int): boolean//param used to indicate sorting criteria (name, value, type), //return value used for feedback //pre-condition: userBag.size() != 0 +saveBag(String): boolean//String filename to save to, boolean for user feedback +getBag(): Bag//return shallow copy of Bag +checkSize(): int//returns size of userBag

c. GamePlay SubSystem

GamePlay
- Map gameMap

//invariant: gameMap != NULL - List<Item> userInventory - List<Treasure> collectedTreasure - List<GasTank> carryingTanks - GasTank currentTank // invariant: currentTank != NULL - Point startingDepth //3D point for entrance / exit // invariant: startingDepth >0 - Point currentLocation//used for real-time calculations //invariant: currentLocation != NULL - Point focalPoint //used for direction of viewport //invariant: focalPoint != NULL - double breathingRate // invariant: breathingRate > 0 - double time//calculated based on currentTank and breathing rate // invariant: time > 0 - double oxygenPressure //calculated based on depth and current gas formula // invariant: oxygenPressure >=0 - double nitrogenNarcosis //calculated based on depth and current gas forumla // invariant: nitrogenNarcosis >=0 - double buoyancy//0 if user is balanced, positive if going up, negative if going down - Trap activeTrap - Viewport playerViewport // invariant: playerViewport != null
+ getTreasure(Point): Treasure//calculation between currentLocation and treasure coords, returns null if not close enough + useTool(Item): bool// if activeTrap is not null, compares tool used to activeTrap.removal, otherwise checks nearby traps or entities to attack if tool matches trap removal or if knife / flashlight to entity + moveViewport(Point, Point): bool //calls update viewport to relocate viewport //post-condition: the viewport != self@pre viewport + checkRange(): bool //called with viewport movement, linearly checks if close enough

to get, trap or harm entity
+ springTrap(Trap)//called while checking for collisions
//post-condition: Trap is activated
+ disableTrap(Trap)//called in useTool, disables trap
//pre-condition: activeTrap != null and Trap parameter==activeTrap, activeTrap==null
// post-condition: Trap is disabled
+ tankUpdate(int): bool//returns false if int index doens't point to GasTank, otherwise makes it currentTank
+ consumeGas() //continuously run to consume gas based on breathing rate and updates remaining time based on breathing rate and remaining value of currentTank
//post-condition: self.currentTank.remaining < self@pre.currentTank.remaining value
//post-condition: self.time < self@pre.time
+ attackingEntity(Entity)//checkRange, check if hostile, then add breathing rate penalty, is called each second until checkRange() returns false
//pre-condition:checkRange()
//post-condition: self.breathRate > self@pre.breathRate
//post-condition: !checkRange()
+ destroyEntity(Entity)//called when Entity is hit with knife or flashlight, body is converted to Treasure with same coordinates and worth of Entity
//pre-condition:checkRange()
//post-condition: self.breathRate > self@pre.breathRate
+ executeObtain(Point): bool //reads user input, prompts user based on results from getTreasure
//per-condition: checkRange()
//post-condition: self.userInventory != self@pre.userInventory
+ executeUse(int): bool //mapping of keys to tool slots (indices in userInventory), check if item in that slot and performs tool function
//pre-condition: int > 0
+ executeLook(): bool//call to Game.moveViewport, translates mouse movement to new viewport location
//post-condition: the viewport != self@pre viewport

+ executeSwim(): bool//call to Game.moveViewport, translates force and direction of swimming to new viewport location + executeRotate(): bool//call to Game.moveViewport, translates change of body positoin to new viewport location // post-condition: self.viewport != self@pre.viewport + executeTankSwitch(int): bool//mapping of keyboard to gas tank to switch too, calls tankUpdate in Game

Entity
bool small //if small, can be killed by a knife or flashlight # bool hostile// if hostile, will attack user if user moves in focal point # Point currentPoint # int pathIndex //invirant: pathIndex >0 # double breathingRatePenalty //penalty per second to breathing rate that user takes while entity is close enough to user and attacking //invirant: breathingRatePenalty >=0 # Point focalPoint # Point previousFocalPoint # Viewport entityViewport # int worth//if killed, treasure worth // invirant: worth > 0 # List<Point> travelingPath//collection of points that Entity will travel, will contain full circle of points (Point A is first entry and last entry is very close to A)
moveEntity(): bool //moves from travelingPath[pathIndex] to travelingPath[++pathIndex], also calls updateViewport and checkPlayerLocation # checkRange(Point): bool//check if close enough to attack player # checkPlayerLocation(Point): bool//returns true if player is relatively close distance wise to line between entity and focal point and if able to see # attackPlayer(Point): bool //moves from currentPoint to users point, also changes focal point, must be hostile

//pre-condition: checkRange() //post-condition: self.currentPoint != self@pre.currentPoint # resumePath(): bool //previousFocalPoint becomes focalPoint, returns to travelingPath[pathIndex], returns to cycle of moveEntity // post-codition: self.focalPoint = self.previousFocalPoint

Trap
- Point currentLocation - bool restrictMovement//if true, user cannot move while stuck in this trap - double breathingRatePenalty // invirant: breathingRatePenalty > 0 - Item removal //if removal == item used while stuck in trap, trap is disabled
+ disableTrap(): bool//trap is disabled immediately after sprung if !restrictMovement // post-condition: retrictMovement ==false

Viewport
- Point currentViewportLocation - Point currentFocalPoint
+ updateViewport(Point, Point): bool //direct transition from previous location to new location, also setting the direction towards the new focal point // post-condition: self.currentViewportLocation != self@pre.currentViewLocation + display(): bool //running in constant time to display viewport

Treasure
- double volume - Point currentLocation - int worth
+ compareCoordinates(Point): bool//true if distance between points is under threshold

GasTank
- double volume //invirant: volume > 0 - double oxygenConcentration // invirant: oxygenConcentration > 0 - double nitrogenConcentration // invirant: nitrogenConcentration > 0 - double heliumConcentration // invirant: heliumConcentration > 0 - String name
+ consumeVolume(double): bool //decrementVolume, but volume cannot be less than 0 // post-condition: self.volume < self@pre.volume // post-condition: self.volume > 0

VI.Testing

In this section, we briefly state the basic acceptance test cases of the game. Also it might be good to decompose the whole project into two releases, if it's developed by a small team in limited time.

29. Acceptance Test

After the game has been launched, the user can either create a new game or load a saved game. Both options shall list the previously saved as well as empty states. Creating a new game will place a new save file in the spot of the users' choice. In either case, the information in the save spot (new or old) will be loaded.

Once the game has been loaded, the user will see the world map. This includes a series of levels to explore and various menu options. They can open their own storage, inventory, or go to the market. The previous market state is loaded when the user opens the market, and the user can sell or buy items if they have enough money (or be told they can't if they do not have enough). The user must also be able to move items they can equip between their inventory and storage. The user should be able to save the game at any time upon the user's request.

One level or area the user may choose to explore will have no costs associated with it, and the successful completion of that level will generate money for the player. If the user selects any other level to explore, a menu is opened to select options about their dive plan. A minimum depth range and time will be selected by default based on the level, and the user can select more time or more depth. The default and user selected options will have a cost, and the cost will change accordingly based on the selections. When the user confirms the dive plan, the current inventory is loaded into the same level they are exploring. The state of the user's game is saved, and then the cost of the dive plan is taken out of the user's money. If they do not have enough money, the user cannot explore the level. The state of that level is retrieved from memory and the user is now able to explore the level selected.

Once they are inside the level, the user will have a first person view of the level they are exploring. They can swim forward, pull themselves in any direction provided they can use their hands on something, follow a reel line, or rotate their body. They must also control their buoyancy. As they are exploring, they must also be able to place a reel line (provided they brought one with them). They should also be able to turn on and off any lights they brought with them, and be able to use the handle of a flashlight as a blunt object. The user should also be able to use knives or other sharp objects to either cut various rope or as a weapon against smaller fish. While navigating, if the user collides with a trap, the trap is sprung. If the trap has any restrictions on movement, the user must also now remove themselves from the trap. This action will be displayed in a third person view. Similarly, if there are any hostile fish, the fish will also attack the user. Any traps or fish that attack the user will cause an increase in the user's breathing rate, dependent on the individual fish or trap. The user can pick up various treasure that they find in the level as well as drop treasure, but this will impact the user's breathing rate and buoyancy.

The user will have the ability to change their gas tank to one of the tanks they brought with them. The user will see various statistics, computed in real time, about the time remaining on their current gas tank, partial pressure of oxygen, nitrogen narcosis, breathing rate, and their depth. If any of these values reach invalid thresholds, a warning will be sent to the user. The failure or inability to resolve these warnings may result in the death of the user. The user will also be able to see the objective(s) of a level, and be informed when they fulfill that objective. If the user successfully leaves the level, the state of that level is saved. Any objectives completed has the possibility of unlocking additional levels, if any levels depended on those objectives. If the user dies in the level, no information is saved and the user is brought to the original menu screen.

30. Release Decomposition

A smaller development team may have success following this slightly related set of releases.

- First Release:

When the system is launched, the user is already submerged in an environment. The environment only needs to contain “walls” and open spaces for navigation. They can see a first person view of the environment they are exploring. They can place a reel line as they explore. They are also given the gas tanks necessary to explore the underwater level, and they are able to switch gas tanks. The user can swim forward, push or pull themselves in any direction if they can reach a solid wall or a reel line, or reposition their body. As the user is exploring the level, the system is performing real time calculations for depth, gas consumption, partial pressure of oxygen, nitrogen narcosis, and breathing rate. If any of these values reach invalid thresholds, the system will prompt the user that the game is over and exit. If the user swims through the entrance (which is also the exit), then the system prompts the user that the level is completed, and the game will exit.

- Second Release:

When the system is launched, there is now a menu. A new save file can be created, and any existing save file can be loaded. Loading a save file will launch a world map with the previous data. At any time outside of exploring a level, the user can opt to save the state of their game. The world map will contain one or more environments for the user to explore. There will also be a score value. When the user selects a level to explore, the user is prompted with choices about the dive plan. They can select an initial depth (to avoid decompression diving if they want), as well as the at depth range of the dive, within ranges dependent on the depth of the level. They can also select how much time they will dive, increasing or decreasing their gas supply. The values of the dive plan, as well as a default set of tools (2-3 knives and a reel line) are parameters of the level exploration. Right before the level exploration begins, the system will automatically save the state of the user’s game. The environment now contains collectible treasure, which the user can pick up. It also contains small fish, which may attack the user and can be killed with a knife. There may also be ropes (as traps) that can tangle the user, which requires the use of a knife to become free. Both entity and trap interaction may affect the breathing rate. If the user completes the level (through the entrance / exit), any treasure they collected is converted to points for the users save file, which is automatically saved before re-loading the world map.

VII. Conclusions

In this section we sum up some successful experience as well as lessons learned while doing the projects in the whole semester.

31. Conclusions and Development Process Review

This semester we have been working on a virtual reality scuba-diving sports game – the Lost Empire. As a software project, we mainly finished the **requirement elicitation**, **analysis**, **system design**, **object design**, as well as **acceptance tests requirement** – following the procedure in Figure 8 OO SE development activities (Fig 1-2 in [1]), producing corresponding work products, and covering everything before implementation.

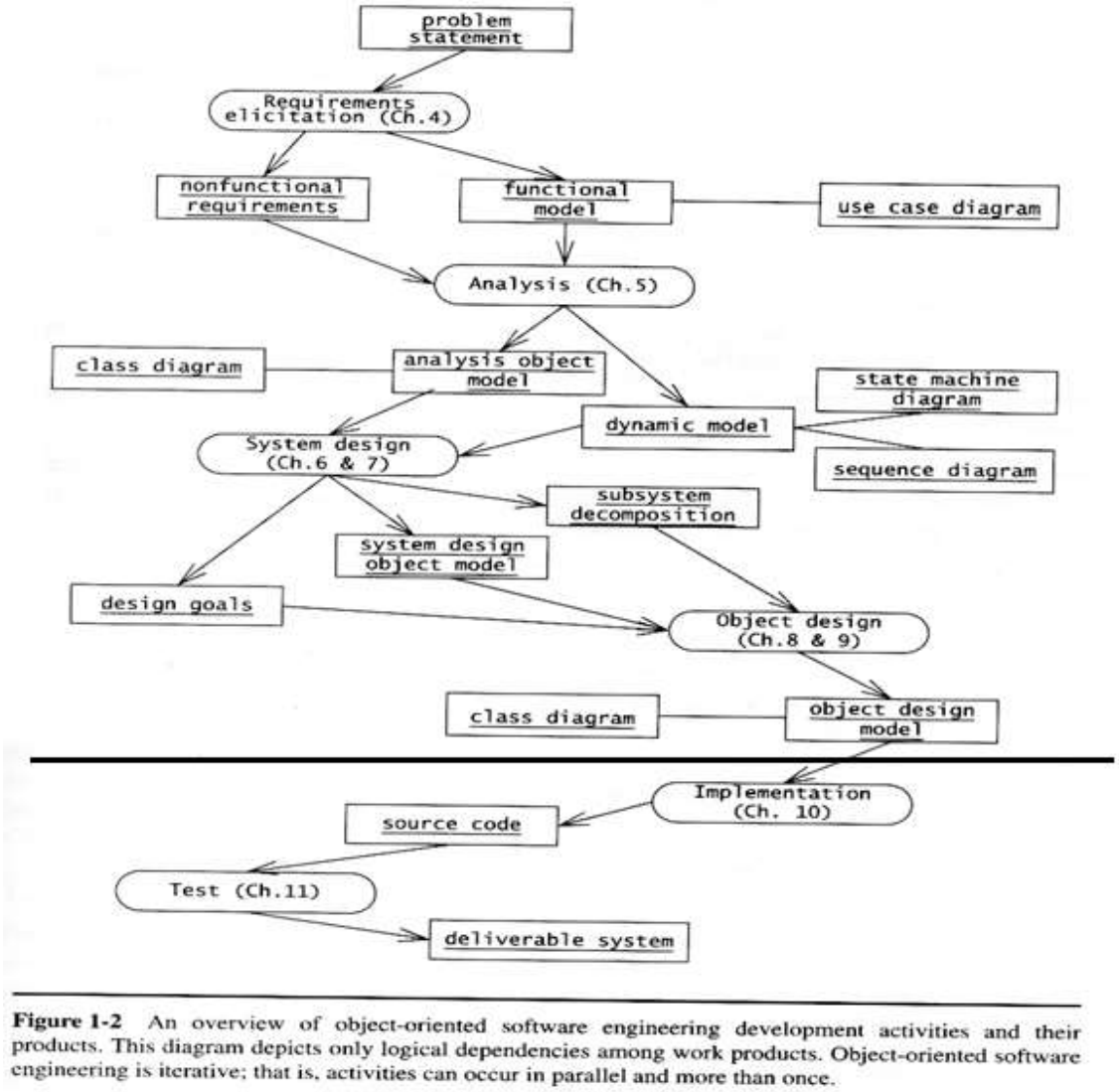


Figure 1-2 An overview of object-oriented software engineering development activities and their products. This diagram depicts only logical dependencies among work products. Object-oriented software engineering is iterative; that is, activities can occur in parallel and more than once.

Figure 8 OO SE development activities

Compared to a complete **Waterfall software development lifecycle**, which contains 3 phases as **Problem definition**, System development, and System operation (Fig 15-2 in [1]), we are only on the first phase actually. The main **work products** we have created include a deliverable Requirement Specification, and an internal Design Document, with a variety of internal work products such as UML diagrams contained in

those documents. In this process, we did learn and carry out a lot of methods of Object-oriented (OO) Software Engineering (SE).

a. Project Organization and Management

Task Assignment and Roles

We've acted out the **Skill Matrix** method although we don't really have one. It's helpful. The first day all our team members met up, we had talked about our skills and interests. Then later after, we tried to respect everyone's interest.

It's a pity we did not really have different roles among our team. Initially until the first code demo, we had different roles, for which we lost score because "not everyone is involved in every task". This is one shortcoming of a small class project, which is avoidable since group members' work load can be judged in different ways.

Communication mechanisms

In our group we prefer to communicate **first synchronously** (meet up and decompose tasks)**, then asynchronously** (each member does his job as well as keeps an eye on others' work)**, and finally synchronously** (meet online and integrate).

Communicating with people outside our group, such as the domain expert, we use both asynchronous emails and synchronous meeting.

Schedule and Time management

We built a **Gantt chart** at the very beginning to schedule our timetable and progress for our project. It works out well.

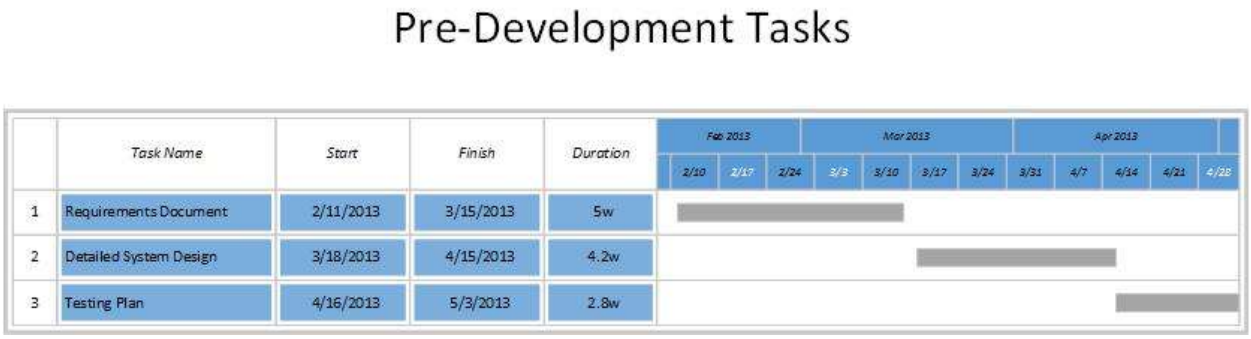


Figure 9 Pre-Development Tasks Timetable (Gantt chart)

For each activity, we decompose it into tasks as soon as possible, in a matter that "everyone who has time devotes to some task immediately". Tasks will be assigned priorities according to its importance and expected finishing time, taking the idea of **Critical Path**. In this way, we finish all activities relatively early (with respect to the deadline – "the planned finishing date"), and also we have time to revamp before submission ("phased release").

Ice Scrum, Git, and Google Drive

Ice Scrum is a powerful site for managing large-scale software projects, but we didn't make full use of it for our tiny project – it was not necessary or helpful.

Git is quite helpful for collaborative development. Work done by different people can be merged by Git automatically. By reading historical comments, you can keep track of what's going on in the whole group. But it is still an issue if more than one people work on one file at the same time, only the last pushed copy will be saved. So make sure this will not happen by confirming with other team members before you start working on it.

In such situation that more than one people work on one file, **Google Drive** can be useful. You can work online at one document at the same time, and you'll see who else is working on this file, where he is modifying. All changes will be saved to server immediately. In addition, Google Drive allows us to comment on the document and mark disputation, as well as to chat with all other editors on this documents.

Also, as the document is growing sharply in size, it's not a good idea to merge every part too early or make everybody work on a single one document – it will be difficult to control concurrency.

b. Knowledge acquisition and Domain expert

To better clarify our product scope and requirements, we had the great honor of having Dr. John Bell and his friend Mr. Michael Angelo Gagliardi as our domain experts, and they contributed a lot. Many details of our game designs came from the knowledge they provided. So it's a good experience to **invite some domain expert** if the project involves domain knowledge.

c. UML Modelling

UML diagrams have been very helpful all through the project, and choosing proper diagrams for each activity is very important. Use Case diagram, and some dynamic UML diagrams, such as Sequence diagrams, and Activity diagrams, can be applied to describe scenarios and clarify functional requirements; some object models, such as Class diagrams, Component diagrams and deployment diagrams, build a bridge from documents to software products.

But pitifully we don't have a chance to take full advantage of UML modelling. For example, we only write documents for one project (and write code for another), thus we cannot use the "export UML models directly to code" function. Therefore, there might be imperfection or even errors in our models, which we will not know since we don't implement them.

d. From Requirement to Design

We learned a lesson when transforming our requirement specification to design: **Do not expect or extend too much when specifying requirements**. No matter how much

effort you devote to make the requirement clear and all-sided, they may not turn into products. Maybe you don't have enough time/money/labor to fulfill the whole huge project as initially scoped; maybe later you don't agree with your prior thoughts. Whatever, just keep focused on key aspects and **build up little by little**; avoid unnecessary branches – they might become a waste of resource.

System Architecture and Design Pattern

When decomposing the system into subsystems, we designed an open layered system architecture, in which we separated User Interface, Control layer and Data Storage, similar to the **Three-Tier Architecture**. We kept "**loose couple, high cohesion**" in mind in this process.

We also purposely used **Design Pattern**s in our Object Design, such as Abstract Factory, Composite, and Bridge. They solve different issues, but all help to increase our abstraction level and make implementation potentially more reasonable.

Interface design

During Object design phase, we used the **OCL language** to describe interface constraints. It's somehow clear, and when implementing classes, the OCL constraints defined in the design document can be referred to, so as to build safe functions.

Reference

[1]. Bernd Bruegge and Allen H. Dutoit, "Object-Oriented Software Engineering", Third Edition

Appendix

Table of Tables

Table of Figures