**CHAPTER 3**

**RESEARCH CORE**

In this chapter, we will mainly discuss about the analysis and design phases of our project. For the analysis phase, we wanted to find out if our idea (creating an educational role-playing game / RPG to teach people how to program) will be interesting and marketable, and elaborate the technical aspects needed to create the game. So we conducted an online questionnaire and made comparisons between our idea and other related papers. Whereas for the design phase, we will illustrate our game project in the form of UML diagrams, i.e. use case diagram and class diagram. We use UML diagrams since we mainly use object-orientation in the development phase (Unity / C#).

**3.1 Analysis / Thinking Framework**

**3.1.1 Questionnaire Results and Explanation**

In order to find out how people will be able to get engaged to our game, we have conducted an online questionnaire for it in this practical work. In the questionnaire, we would like to examine people’s programming experiences, especially in their learning process. We also asked about their opinion about educational games, and how we could create an educational game to guide them to learn programming. We tailored the questions in our questionnaire for both computer science students and people from other professions, so that we could obtain both the needs and perspectives of a typical programmer and a non-programmer.

Based on our last questionare result in June 12, 2016, 6:00 PM, there were 38 respondents. Below are each question in the questionnaire at the time, along with their corresponding results and explanations:

1. **“What is your gender?”**

*Figure 3.1: Gender proportions of our respondents.*

The result of the first question in this questionare is surprisingly good where we obtain there is only a small difference between the male respondent (55%) and female respondent (45%) , based on this questionare we can get respond from both gender almost equal which is good for our future research.

1. **“What is your age?”**

*Figure 3.2: Ages of the respondents answering our questionnaire.*

The respondent of the our second question is mostly from 18-22 years old, which we can presume most of them were the university student,in the other hand there is some of respondent which is from 22 years old or more category . We purposely distribute this questionnaire towards university students, especially in computer science major – since we see that their ages and major are ideal for learning how to program.

1. **“Have you programmed or coded anything before?”**

*Figure 3.3: The respondents’ prior programming experience (Yes/No)*

Since our survey, we have expected a lot of the respondents come from the computer science background. And as seen in the result above, 79% / 30 respondents have ever programmed before.

1. **“Which programming languages have you used or heard of?”**

*Figure 3.4: The number of programming languages the respondents have used or heard of. The sum of the data above are not the total number of respondents in the questionnaire. The number beside each blue bar only indicates the number of respondents using the corresponding programming language.*

As we can see, most of the respondents have used C/C++ and/or Java languages to program. If we analyze further, 4 to 25 respondents used general-purpose programming languages (C/C++, Java, C#, etc.), 2 to 11 respondents used web programming languages (PHP, JavaScript, JSP), and 9 respondents used other languages (most of them are non-programming languages, commonly HTML). While the other four respondents haven’t used or heard of any programming language.

Note that in this question, we do not necessarily ask whether the respondents actually used those languages. We also ask if they have heard of them. That is why we received different results from the previous question and this question.

1. **What makes you want and/or don’t want to learn programming/coding?**

The basic reason of why the respondents wanted to learn programming was their internal motivation and curiosity. Most of the respondents felt challenged and sought the importance and benefits of programming in their prospective careers.

However, not a few of them and other respondents who aren’t willing to learn programming stated about the difficulty or learning curve they previously faced. Those who don’t want to learn programming also wrote that programming is complicated.

*(The next question is on the next page)*

1. **Have you ever played any educational games?**

*Figure 3.5: Proportions of respondents who have and have not ever played educational games*

Most of our respondents (31 people) have prior experience with educational games, while the remaining seven haven’t played them. Educational games (e.g. Edu-Games) have been quite popular since several years ago, especially in Indonesia. But usually those who played educational games are children under age 18, and the target age for the consumers of our games are from 18 to 22 years old. From here we aimed to create an educational game that is universal to all ages, but still themed around fantasy RPG that is suitable to their ages.

1. **Which aspects do you prefer from an educational game? (May choose more than one)**

*Figure 3.6: Aspects preferred in educational games*

As seen above, most of the respondents preferred a good learning system in the educational game they would play. A fair amount of them would like to see the game being addictive and capable to provide great action for them. Less respondents have other preferences, i.e. a good story and playability.

1. **If you weren’t able to program, and there was a game to teach you how to program, what do you hope/expect from that game?**

From the results of this questionnaire, we can see five recurring points that the respondents are looking for in a game that teaches programming:

* An interesting gameplay that doesn’t bore the player,
* Educational value in the game,
* The player finds the game simple and understandable,
* The game is supported with good and clear tutorials that can make the player understand and own great skills in programming, and
* The game should encompass most problems found in programming, and teaches the player problem-solving.

**3.1.2 Comparison with Related Papers**

As described by Entwisle, there are three types of motivation:

* Extrinsic: doing something because of external rewards;
* Intrinsic: doing something due to the interest one has; and
* Achievement/Competitive: based on doing something well, and sometimes better than one’s peers.

Based on Feldgen and Clúa’s survey (2004), most students who took programming courses in their university relied on extrinsic motivation to do well. So theythoughtof introducing game-related concepts and approach to how students study the courses. The lecturers encapsulated the raw concepts taught in programming with real-world and game-related examples while teaching. They also asked their students to do role-playing: some as ‘designers’, and others as programmers. Eventually, they really did well in the courses, because of how this “game approach” helped their learning process. They also saw the increase of students doing well in game-context courses (despite the decrease of students doing well in engineering-context courses). We now see that by using the “game approach” in learning programming and other abstract concepts, students will eventually be able to grasp what they have learned, and in return harness better skills and deliver well-done results.

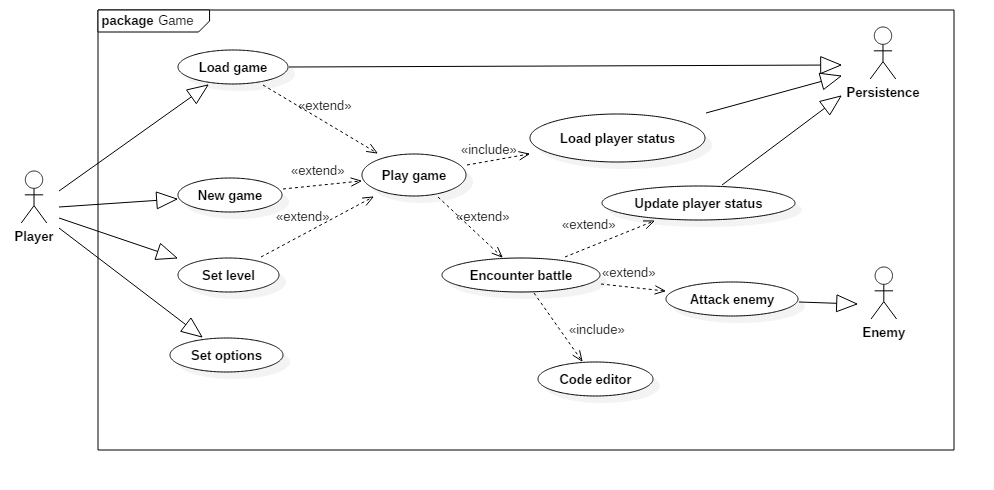
In a paper written by Mathrani, et al. (2016), they introduced a more concrete game-based learning approach to students who took programming courses. According to Connolly & Stansfield, (2006), as cited in her article, giving programming assignments without simple or well-known solution will leave most students unable to translate what they've learned into related situations. There is a need to make the learning experience more interactive and scaffold the concepts in games to engage the students. So they decided to introduce PlayIT, a game-based learning approach to teach programming concepts to the students. From there, they mapped the programming modules from their university with an educational game. By their observations, students find that they understand concepts better by using games in their courses. They had helpful experience in learning how to program, perceived programming as a positive activity, and finally improved their skills and confidence through PlayIT.

From the research articles above, we can see the importance of creating a fun, educational, and practical atmosphere for people to learn programming, and how effective it is in helping them learn. They are able to relate to the concepts better, given real-world examples, and also will be motivated to learn programming and perceive it as an enjoyable and beneficial activity to do. For us, programming is a universal and essential skill for every computer science student, so that is why it is important to teach them programming in the right and encouraging way.

**3.2 Design Phase**

**Note:** Some of the features or components in the diagrams may not represent the features in the current version of our game. They are for the improved version of the game.

**3.2.1 Use Case Diagram**

****

*Figure 3.7: Use case diagram of The Crystals Quest game*

If the player is playing the game for the first time, he/she will start a new game. The new game loads all the initial level, map, and other components the game needs for the player, so that the player can then play the game.

We are planning to divide our gameplay in terms of levels, and each level represents a single programming concept the game will teach. As the player walks through the map, he/she will encounter the next level (If the game is new: starting from level 1), and in each level there is an enemy. A tutorial will be provided per level, so that the player can follow through and implement it easily. After the tutorial, the player has to attack the enemy. To attack the enemy, the player can use the programming concept taught before, and then write down some code in the code editor. If the code is correct, then the player will be able to attack the enemy. Otherwise, the player can’t attack. In either case, the enemy will attack the player after the player’s turn, as long as the enemy is still alive.

If the player wins the battle, the player’s status (the player’s position in the map, health points, etc.) will be updated and saved in the game’s persistence. After storing it into the persistence, the new status is loaded for the player to play again in the next levels.

The player can set whichever level the player has entered into, perhaps to refresh their skills on a certain concept. The player can also set their options for the game.

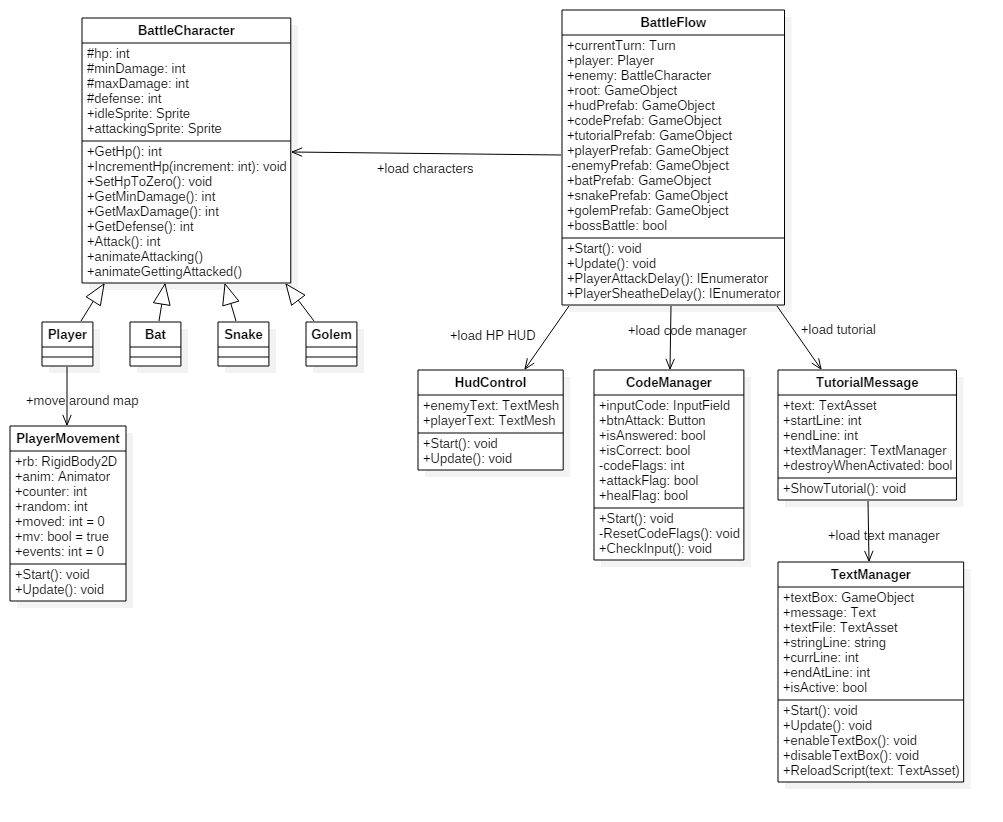
**3.2.2 Class Diagram**

***Note:*** *the class diagram is shown on the next page, in landscape orientation.*

As described before, the game is divided in terms of levels. BattleFlow is the core class of the battle system in the level. It handles the necessary components required to run the battle, i.e. the player object, the enemy object, the HUDs, the code editor, etc.

Inside of the BattleFlow class, there are the following attributes:

* currentTurn: which handles whose turn it is to attack (the player or the enemy);
* player: contains the battle data of the player;
* enemy: contains the battle data of the enemy. Since it is a BattleCharacter object, it will choose one of the following child classes: Bat, Snake, or Golem;

**

*Figure 3.8: Class diagram of each level in the game*

* root: Controls the Root object of the battle scene (scene as in Unity’s vocabulary), which consists of all the objects inside that scene;
* hudPrefab: controls the HUD prefab (as in Unity’s vocabulary) of the health points of both the player and enemy;
* codePrefab: controls the code editor in the battle system. It shows whether the code is correct or not, which will in turn affect whether the player can attack the enemy or not;
* tutorialPrefab: controls the tutorial message for each level in the battle. It relies on a TutorialMessage object, which also depends on TextManager. (Both TutorialMessage and TextManager classes will be explained later.);
* playerPrefab: controls the player prefab, which contains the player’s sprites and script to control his/her sprites and animations in the battle;
* enemyPrefab: controls the player prefab, which contains the player’s sprites and script to control its sprites and animations in the battle. A random enemy will be spawned per level, so enemyPrefab will choose either batPrefab (bat as the enemy), snakePrefab (snake as the enemy), or golemPrefab (golem as the enemy);
* batPrefab, snakePrefab, and golemPrefab: the enemies that can be generated in the battle. golemPrefab is the boss enemy, which will be spawned at end of the game.
* bossBattle: controls whether the player will meet the boss (golem) or not in the game.

The operations of the BattleTurn class include:

* Start(): initializes the data needed for the battle system;
* Update(): called for every frame, it controls the flow of the battle, which turn it is to attack, and the underlying operations to take for the current turn, i.e. calculate damage, animate attack, etc.
* PlayerAttackDelay(): changes the sprite when player/enemy is attacking
* PlayerSheatheDelay(): changes the sprite when the player/enemy is about to return to its initial position.

As seen in the class diagram, the BattleTurn class relies on four other classes: BattleCharacter, HudControl, CodeManager, and TutorialMessage. These classes control the individual units of the battle system.

The BattleCharacter class represents the character that will be spawned in the battle system, along with their battle status (particularly HP / Health Points). It has inheritance relationship with four classes: Player, Bat, Snake and Golem. Bat, Snake, and Golem are the three enemies that need to be attacked by the player throughout the game, while the Player is the object the user controls throughout the game.

A BattleCharacter object has the following attributes, unique to each of its child classes:

* hp: indicates the health points the character (player/enemy) has;
* minDamage: the minimum amount of damage the character can deal;
* maxDamage: the maximum amount of damage the character can deal;
* defense: the amount of damage that can be absorbed or subtracted from by the character;
* idleSprite: the sprite/image when the character stays in place without moving; and
* attackingSprite: the sprite/image when the character attacks its opponent.

And it has the following operations:

* GetHp(): return current HP of the character;
* IncrementHp(increment : int): increase (if increment is a positive integer) or decrease the character’s HP (if increment is a negative integer), usually used for dealing damage or healing;
* SetHpToZero(): set the character’s HP to zero. This is used to prevent displaying negative number on the character’s HP, if the damage they receive reduces their HP to a negative integer;
* GetMinDamage(): return the minimum damage from the character, used as the data for calculating the damage it can deal;
* GetMaxDamage(): return the maximum damage from the character, also used for damage calculations;
* GetDefense(): return the defense of the character, also used in damage calculations;
* Attack(): generates a random number as the amount of damage it can deal, inclusively between its minimum and maximum damage;
* animateAttacking(): this method will animate the character attacking its opponent.
* animateGettingAttacked(): this method will animate the character getting attacked by its opponent.

As mentioned before, the user of the game will control the Player object throughout the game, and that Player object will move around the map. To support this object’s movements, we designed a PlayerMovement class. It helps the Player object to walk, makes decision on whether it will encounter an enemy after several steps, and saves the Player’s coordinates/position in the map when it encounters the enemy.

PlayerMovement has the following attributes:

* rb: the RigidBody2D object used to render the Player in the map;
* anim: supports in animating the Player’s movement, if he/she walks;
* counter: counts the number of steps the Player walks. This variable is usually used to spawn an enemy after a particular number of steps has been reached;
* random: the number of steps it takes for the Player to encounter an enemy. This is randomly generated, as the name suggests.
* moved and mv: acts as flags for the Player’s movement; and
* events: controls whether the Player is walking or not.

PlayerMovement also has the following operations:

* Start(): initializes the Player’s position (or loads the previously reached position after a battle)
* Update(): controls the Player’s movements per frame, checks for certain conditions and then carries out the corresponding actions.

As shown in the diagram, BattleFlow also relies on the other main components for the battle system: the HUD control (which displays the player and enemy’s health points), the code editor/manager (which receives and checks correctness of the code), and the tutorial manager (which loads the tutorial message box in that level).

The HudControl class only has two attributes:

* enemyText: displays the enemy’s health points; and
* playerText: displays the Player’s health points.

It also has only two methods:

* Start(): initializes enemyText and playerText; and
* Update(): gets the enemy and Player’s health points (using the .GetHp() method) and displays it by enemyText and playerText.

CodeManager has the following attributes:

* inputCode: a text field in where the user inputs their code;
* btnAttack: a button to submit and attempt to attack the enemy, based on the user’s code. If the code is correct, then the Player can attack;
* isAnswered: set to true when the code is about to be checked, set to false after the checking;
* isCorrect: indicates whether the submitted code is correct or not;
* codeFlags: used to check whether every part of the code needed is already written in the user’s code.
* attackFlag: set to true when the user wants the Player to attack the enemy; and
* healFlag: set to true when the user wants the Player to heal him/herself.

CodeManager has the following methods:

* Start(): initializes the code manager;
* ResetCodeFlags(): reset the code flags after the code has been checked before; and
* CheckInput(): parses the code submitted by the user.

TutorialMessage class is responsible for displaying the tutorial. It has the following attributes:

* text: the text file / asset from which we get the tutorial sentences/strings;
* startLine: the starting index of the strings to display;
* endLine: the final index of the strings to display;
* textManager: a TextManager object which will support in displaying the tutorial; and
* destroyWhenActivated: which will destroy the TutorialMessage object as the whole tutorial has been displayed.

It has only one operation: ShowTutorial(), which relies on and enable TextManager, so that the tutorial can be displayed.

The TextManager class has the following attributes:

* textBox: refers to the box from where the text is displayed;
* message: the text object which will display each sentence from the tutorial text file;
* textFile: refers to the text file which stores the tutorial for the current level;
* stringLine: an array of strings that store each sentence from the tutorial, to be displayed in message;
* currLine: an index which refers to the current line to display in the tutorial;
* endAtLine: the index referring to the final sentence which the tutorial will display; and
* isActive: a boolean flag indicates whether the tutorial message box is active (will display the tutorial) or not.

TextManager also has the following operations:

* Start(): initializes the text manager by loading the tutorial sentences;
* Update(): displays the sentences from the tutorial. This method is also called per frame;
* enableTextBox(): enable the textbox to display the tutorial;
* disableTextBox(): disable the textbox after the entire tutorial has been displayed; and
* ReloadScript(text: TextAsset): reloads the text into TextManager.