JUMP HIGHER

Final Project Report

CSC3002 Introduction to Computer Science: Programming Paradigms

Authors:

Rulin Liu	115010193
Haoxiang Lin	115010190
Wenbo Luo	115010054

The Chinese University of Hong Kong, Shenzhen

Contents:

1. Abstraction

The idea of our game, Jump Higher, comes from a game called Doodle Jump. These games like Doodle Jump attracts people contributed by their simplicity in manipulations and instructions, while challenging when developed to further stages. However, Doodle Jump can only be played on mobile phones and only offer limited choices of modes. So we decided to make one “advanced Doodle Jump” with the help of Qt Creator and make ours more entertaining by adding more switchable modes and magic tools. This final report starts with showing how to play this game from the view of a client, then detailed explains the implementations from our views as implementers and ultimately concluds with what we have learned via working on this project in details, which both reveals how we solve the problems we have encountered practically and shows our further understanding towards programming and the importance of teamwork.

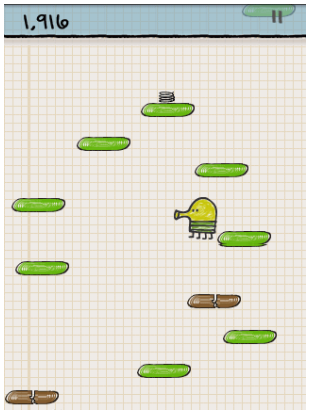

Figure 1-1 Doodle Jump

2. Introduction

The basic idea of this Doodle Jump is to control a character, which looks like a doodle, to keep jumping on the boards, which are generated randomly during the gameplay, without falling out of the screen. The direction controls are realized using the gravity-sensor on the phones.

In Jump Higher, not only the traditional game mode is reserved, we also achieved the goal of making Jump Higher more entertaining by adding more modes (like the Life-Stolen Mode) and more magic tools (like flexible bars) to it. The player doesn’t have to play all day facing the same map, new map would bring brand new playing experiences.

3. Interface Demonstration

In this part, the detailed design of the game modes would be introduced including instructions on how to play the game.

3.1 The Traditional Game Mode

The traditional game mode interface is shown as in Figure 3.1-1.

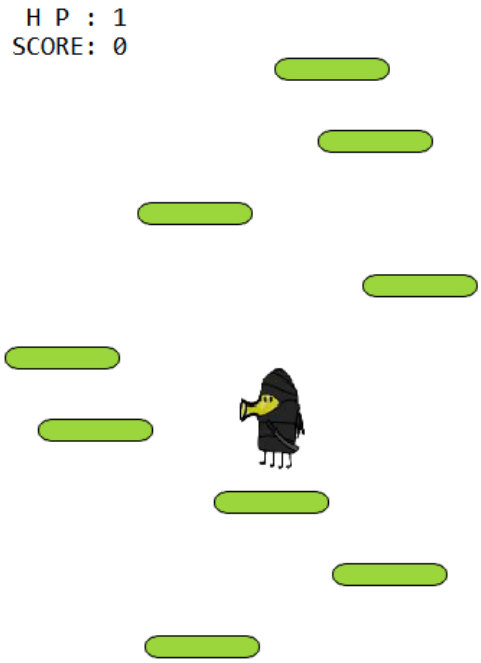

Figure 3.1-1 The Traditional Game Mode

The character can jump automatically, and the gravitational force exerts a constant acceleration on the character as what we have experienced in the real life: the character will reach its maximum speed the moment when it hits the board, gaining a velocity to the amount of velocity but towards the opposite direction at the same moment; the minimum velocity on y coordinates is when the character reaches the highest position where its velocity on y coordinates is zero but users are still able to exert force on the character. The character is controlled by keyboard inputs, to move on the screen. By pushing 'LEFT' or 'RIGHT', the character would gain a velocity on corresponding directions, the this velocity on x coordinates will decrease gradually, through which we also tries to simulate what is happening in the reality; by pushing 'DOWN', the horizontal velocity of the character would be reset to zero, which would help the player stabilize the movement in complicated conditions.

The movement is designed partially based on physical laws. We define a constant downward acceleration on the character, just like the gravity. For moving right and left, considering gameplay, there's also an acceleration, which would gradually reduce the horizontal velocity to zero. All these parameters were calculated and stored in the 'global.h' file. In this way, we don't need to change parameters in all other files when needed. Also, other parameters like the sizes of different items and the widgets are also defined as constants in this header file.

As the game develops in this mode, more magic tools will occur and all of these magic tools will never appear abruptly but along with the fist board appearing on the upper side of the window:

The "HealthPack" (shown in Figure 3.1-2): The HealthPack grants you one 'HP' value, which indicates the health of the character and protect you from one death caused by falling on stings, another magical tool that be later introduced.

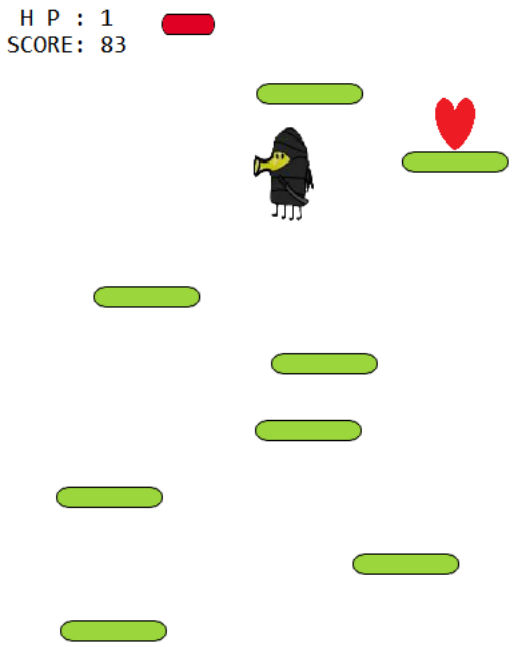

Figure 3.1-2 The "HealthPack"

The "Sting" (shown in Figure 3.1-3): The String magical tool is vividly represents by a group of stings attached to a particular board. Once the character falls on these stings, it will lose one "HP" value. Once its HP value reduced to 0, the user is declared losing the game and they are able to choose between quit the game or retry. Each time falling on the string, a red bloody scene would flash, indicating that the character was hurt.

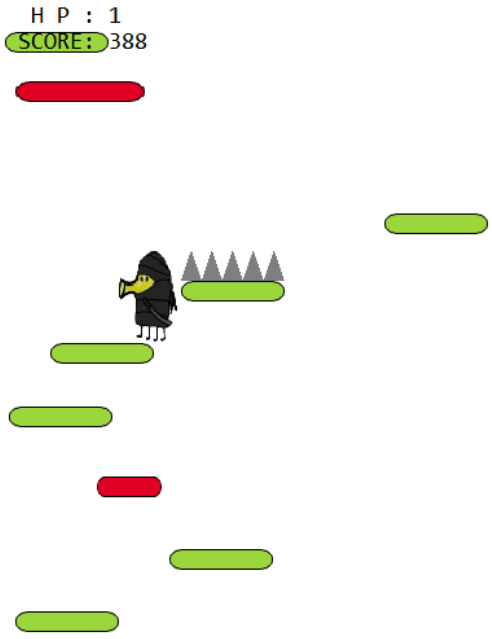

Figure 3.1-3 The "Sting"

The "Spring"(shown in Figure 3.1-4): The spring attached to some boards grants you the ability to jump much higher comparing to a normal jump. Users can still manipulate the character by pushing "LFTT" or "RIGHT" even if the character is rushing upwards.

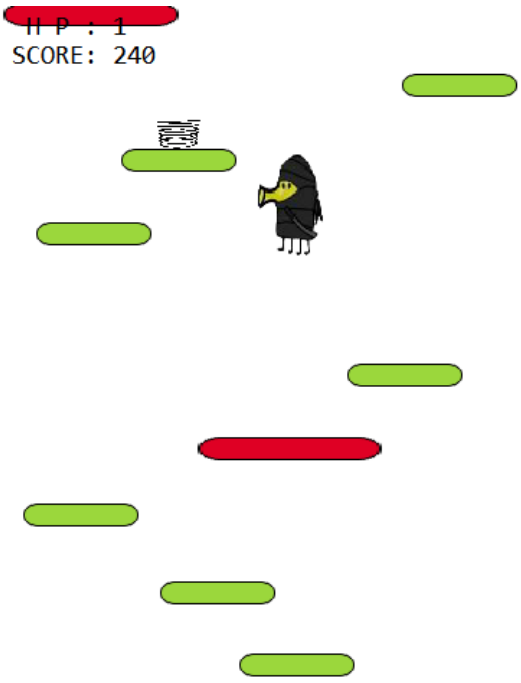

Figure 3.1-4 The "Spring"

The "The Door to Another World" (shown in Figure 3.1-5): Users can regard this tool as a portal to another game mode. Once the character touches "The Door to Another World", the character will be automatically switch to another game mode: The Life-Stolen Mod. This switch between two modes is the one we designed and actually is generated by some algorithms.

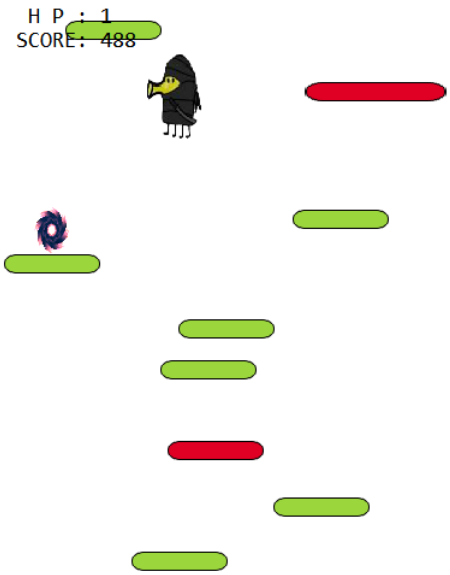

Figure 3.1-5 The "The Door to Another World"

The "Flexible Board" (shown in Figure 3.1-6): This board is a specially designed platform for doodle to stand on. It can automatically change its length, in patterns of elongation and shrink. The character can only stand on the length of the flexible board it has at that particular moment, otherwise, the character will fall off the board. During the gameplay, the score will be calculated based on the distance you've jumped through and be displayed on the top-left corner of the window along with the "HP" value, but the repeated jumps on one single board won't be counted into the score.

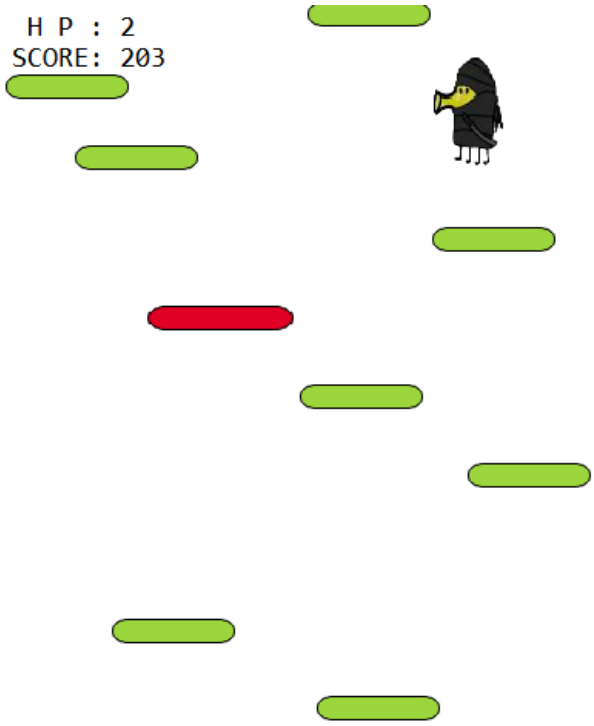

Figure 3.1-6 The "Flexible Board"

3.2 The Life-Stolen Mode

The player can get to the Life-Stolen Mode (shown in Figure 3.2-1) by touching 'the Door to Another World'. In this mode, the map is larger both in width and height and is filled with much more boards. The character will lose half of its "HP" value at the moment when it enters this mode, and a new item named star would only be generated in this new map randomly. The probability of each board generating starts is $\frac{1}{2}$ and the stars will never appear somewhere that the character can never reached (like being with two boards surrounding them). If lucky enough, the character can even earn more "HP" value by collecting stars. However, if they do not try hard enough and falls out of the boards appeared in the Life-Stolen Mode, they will be transfer back to the Traditional Mode even without collecting all the stars and the "HP" values they have lost by entering the Life-Stolen Mode will not be given back.

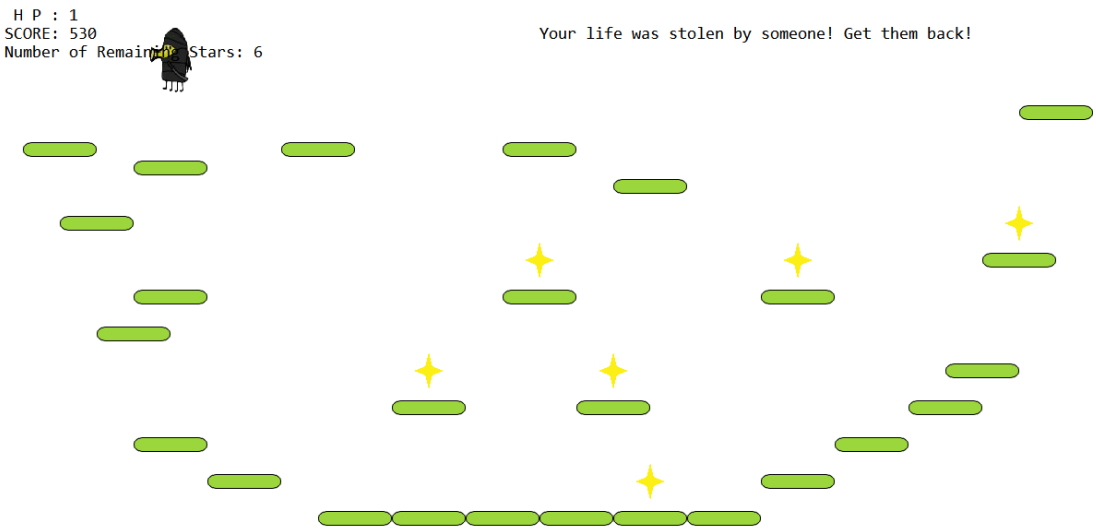

Figure 3.2-1 The Life-Stolen Mode

4. Implementation

To create and operate all these complicated items, classes must be implemented.

4.1 The Object Classes

Since the way of appearing, the position to be placed on board also the needed parameters to construct the magic tools are similar to each other, so we packaged these items into one object classes. They can share the same way to be printed out or calculate their current position.

4.1 The Doodle Classes

The character is generated in “doodle” class. Its data filed contains information about the character’s position, represented in double array [2], velocities both in x coordinates and y coordinates, parameter values like “health”. The position and velocities are stored in arrays of “double” types, which are dynamically stored on the heap using the keyword “new”, so there’s a “delete” in the destructor to free this part of memory. “Double” is used instead of integer because the calculation of integers would automatically round down to become an integer when it’s not. This kind of alteration would make it impossible to calculate using the physical model.

Methods of this class are used to move the character as well as to operate on its parameters values when it comes across some specific functions and takes some items. For example, the take (objects *item) method would first check the type of this item using an “if”, and then operate differently according to the item type. The advantage of using this structure is that, since the item is written to be the parameter of this method, it will be automatically deleted after the method is done running, which is

exactly what we want.

The items are all based on another superclass, which has been shown above, called ‘objects’. Its data field also contains basic parameters of the items, such as the positions, the velocities. But, different from the ‘doodle’, a ‘string’ type parameter called ‘name’ was used to specify the type of the item. Rather than creating different separate classes for each item, putting them all under a superclass and use this ‘name’ to specify them is much better, because the data fields and methods of all of them are actually very similar. Doing this not only saves spaces for unnecessary codes, but also makes it easier for writing methods in the ‘doodle’, just like the take (objects *item).

4.3 The Widget Classes

The main class of this game was created to be a subclass of the QT class ‘QWidget’. It would automatically create a widget and the parameters of it can be altered.

A. Structure

The most important structures of this class are the ‘signal-slot’ system and the ‘update()’ function inherited from the ‘QWidget’.

The ‘signal-slot’ system is very simple, a ‘connect’ function is enough to build it. It would call the ‘slot’ function whenever a ‘signal’ is received. In our code, the ‘signal’ was set to be the passing by of one certain period of time, and the ‘slot’ is a ‘dUpdate()’ function, in which the coordinates of all objects are calculated and the ‘update()’ is called. This way, a loop is formed and it’s actually this ‘dUpdate()’ that would be carried out each time. So all operations on all objects must be called here.

The ‘update()’ function is even simpler. Since it’s inherited from the ‘QWidget’, there’s not much left for us to write. Still, the functions ‘paintEvent(QPaintEvent *)’ and ‘keyPressEvent(QKeyEvent *)’ were overloaded. The former paint our self-designed objects, while the latter check for any keyboard input every time it’s called. These two functions are included in the ‘update()’, so they are also part of the loop and are called every time.

Other functions are the operations on other objects. For example, the ‘onBoard()’ function would go through all the boards and check if any of them is stepped on by the character; and the ‘moveMap()’ would move all things down except the character when the character reach certain height. Details of all the functions were written in the corresponding header files.

All objects are stored in the data field of this widget class in different vectors based on their types, and all of them are in pointer form. This way, the objects of the same type can be treated the same way, and all of them can be treated with certain methods without additional operations like deep copy. Of course, all of these pointers are

deleted in the destructor of the ‘Widget’.

B. Functions
Several important functions will be introduced.

void InitGame();
void backtoOriginal();
void InitNewMap();

These functions make it possible to change between two different maps by adopting connections between a timer and different ‘slot’ functions. In each of ‘backtoOriginal()’ and ‘InitNewMap()’, the former ‘connect()’ is ‘disconnect()’-ed and a new connection is formed. Though the size and style of different maps are not the same, they are actually the same widget. The transformation between them is achieved by altering parameters of the widget.

The boards stepped on by the character and the highest positions between them are recorded in two vectors. By loading data from these two, the monster is able to simulate the path of the player and chase him/her along it. The velocities of monster are redefined every time it reaches a recorded position, and are calculated based on the physical modeling in other time.

void takeItem();
void takeStar();

These two functions would go through the vectors of items and check if any of them should be taken. Once one of them is, the ‘takeItem(objects *item)’ of the class ‘doodle’ will be called. Since the item is taken as a parameter here, it would be deleted automatically. However, it would still take up one space in its original vector, so the corresponding meaningless pointer, whose content has been deleted, must be erased from the vector afterwards.

5. Development Phase

This section introduced our progress during different development phase.

Before handing on our project proposal, we successfully make a very basic version of our Jump Higher on C++ as it is shown in Figure 5-1. Users can control the character, representing by character P. Character W indicates the boundary of the display window and the short bars are boards respectively.

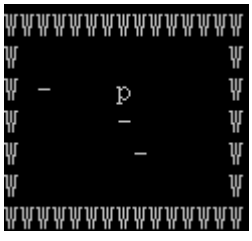

Figure 5-1 Initial Jump Higher

According to our proposal, we planned to use OpenGL to improve our interface. However, weeks later, we realized the display of Jump Higher is simple and we can build Qt Widget Application to display it. The Qt Widgets Module provides a set of UI elements to create classic desktop-style user interfaces. Moreover, the widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.[1] All these properties of widget can facilitate us with better implementing our project. Ultimately, we decided to use Qt Widget Application since user experience and the way Jump Higher weighs a lot in our project.

The detailed problems we have encountered will be discussed in the 7.1 discussion part.

6. Contributions

Our team has maintained a harmonious environment and every teammate works hard to fulfill their work.
Rulin Liu: Doodle Class, the linkage between the Door to Anther World to the Life-Stolen World, the Flexible magic tool
Haoxiang Lin: The Life-Stolen Mode, the very basic version of our Jump Higher on C++
Wenbo Luo: the Sting magic tool, the healthpacks, , UI Interface

7. Discussions

Working on this project is not as easy as it seems and we have encountered a lot of problems. Nevertheless, the experience is what matters and through which we have learned a lot more about programming and the importance of teamwork.

7.1 The Specific Problems We have Encountered

1. Originally we planned to use the physical formula of $\sqrt{2gH}$, but the root formula cannot be calculated in Qt. Thus we have to use logarithm fit to assume some constant variables.

2. The generation of magic tools is random so that it may be at some places that the doodle can never reach as it was shown in Figure 7.1-1.

[1] Qt: http://doc.qt.io/qt-5/qwidget.html accessed time December, 2017.

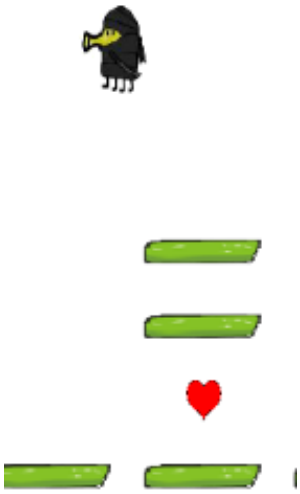

Figure 7.1-1 Items cannot be eaten

For example the health pack is placed one-unit above the board so that it looks like it is on the board. Since the doodle cannot go through a board from upside to downside, the board cannot be continuous since continuous board would form barrier to protect the doodle from going to the lower part. Therefore, we solved this problem by setting the minimum distance between two boards was set to be one-unit length except some special boards. We also found another state. The star was lies between two boards, like this graph.

When debugging, the doodle can successfully get this star as the following graph shows. Although it is extremely hard to get this star, it is not impossible. Just as Figure 7.1-2 what has shown.

Figure 7.1-2 Successfully eat the item

3. When switching from the Traditional Game Mode to the Life-Stolen Mode, program always crashes when we try to open the new window since the old one still not closed.

We solved with problem by building a vector to record the board appearing within the display window. When switching to another mode, the window was resized according to the size of the destination mode. When return from the Life-Stolen Mode, we read

the boards' information previously stored and rearrange the window.

4. The dynamic size arrangement related problem. The program takes a lot more space than expected after executing. We found out several problems that may result in such a condition.
a. For example, when the character takes a health pack, even if the pointer pointing to that particular health pack was deleted, items still exist in vector. So we still need erase that element after every operation.
b. When a lot of pictures loaded at the same time, for example in the mode of the Life-Stolen mode, the speed will be extremely slow but will gradually speed up after the characters get the items one by one. So we conclude that the size of the pictures significantly influenced the loading speed. We solved this problem by drawing some pictures by ourselves and the speed is significantly improved.

5. It's possible for several items to appear on the same board, so we use tags to denote whether an type of item occurred on one board to avoid the condition that multiple item appearing on the same board.

6. After finishing the coding of the nUpdate(), the program would call error every time it was closed. The nUpdate Function is shown in Figure 7.1-3. According to what we had come across, it seemed the error was due to some meaningless pointers. The problem was finally found in nUpdate(). After disconnecting the prior signal and slot, though the update() won't be carried out again, the remaining part, such as the nd->nmove(), would be carried out nonetheless. However, when that line was read, the nd was actually deleted by the backtoOriginal(), which would therefore result in an error. It only called after the program was closed, because the program stays in the backtoOriginal() all the time before it is closed. Only after it was closed, would it jump out of the loop and come across nd->nmove().

```
void Widget::nUpdate() {
    if (newMap == 0) {
        this->backtoOriginal();
    }
    // Without this 'if', the following codes would be carried out without
    // corresponding parameters since they are deleted in backtoOriginal(),
    // which would result in an error, some times a crash.

    else{
        if (nd->dvy == 0) {
            nyzero = nd->dcenter[1];
        }

        this->takeStar();
        this->onBoard();
        nd->nmove();

        nIsGameOver();
    }
    update();
}
```

Figure 7 nUpdate() code

7.2 What We have Learned

7.2.1 About Programming

For a project, which may be consists of a lot of items, the proper paradigm plays a significant role. Our project chose to use Object Oriented paradigm in both Object Class and Doodle Class while used the procedural paradigm was used in widget implementation. During the process of implementation, it was sometimes hard to take care of both sizes. So we can achieve the goal step by step: start from implementing the most basic functions and it is okay even you miss out some parameters or functions. When improve the program we already have, we can add more detailed information into the functions.

7.2.2 About Teamwork

The most difficult part of the teamwork in this project is that all members must work depending on each other's work. Our group chose to work on a very basic program, which has already initialized the jump and hit on board actions separately. Then comparing the code to the original one to add some linkage between our works. We may be both client and implementer to our group mates, so this demands us to think from another person's perspective. For example, as an implementer, we should try to protect the client from as much as detailed information as possible.

How to divide such a work also is also important in this project. We may never be able to divide the work properly, but during the process, we can communicate efficiently. Constant communication between implementers and clients can be extremely efficient. For example, as a client, we can talk to implementer about our expectation to their work, what kind of result should a particular function get. We

should plan all of these at the very beginning of our work.