Table of Contents

Expected Problems with Waterbear

Risk Matrix

	Low Impact	**High Impact**
High Probability	• Difficulty in getting the UI to behave correctly	• Unable to implement "Object" metaphor • Difficulty deploying the software to areas with bad network connections • Bugs related to "assets": pictures, audio, videos in the application • Difficult to find bugs due to the application design
Low Probability	• Discontinued support from the OS community	• Problems running the software on the netbooks

Analysis of high impact, high probability risks

Unable to implement "Object" metaphor

One of the requirements we need to meet is this idea of an "object". An object is a conceptual framework for grouping scripts, images, and behaviors related to a single item on the screen. For example, in the SB tool there is a skateboarder. This skateboarder is his own object, and manages applying gravity to himself.

Implementing this feature in Waterbear will require a large code rewrite because Waterbear was not designed with the idea of "Objects" in mind. We would need to redo the way scripts are generated, the way they are run, and rewrite almost all of the existing "codelets".

Difficulty deploying the software to areas with bad network

Waterbear pulls in dependencies from all over the place. It uses Iframes on the homepage, and loads libraries such as Jquery. Deploying these assets in such a way that the web browser will be able to load them on a slow internet connection can be tricky.

The most obvious way to deal with this is to have the program served from a local machine. However, we can not simply "copy" the files into the computer and call it done. Because of the way

Javascript handles permissions on local browsers, we will need to setup a small web server on the machines to server up the files.

Bugs related to “assets”: pictures, audio, videos in the application

Waterbear was not originally designed with the intention of moving images around the screen. Scott has hacked in a way to load images, and display them at a specific location, but this is still a far cry from the concept of “costumes” that Scratch has.

To implement this feature in Waterbear, we would first need to implement objects then implement some way of loading assets on top of that. An additional complication, as seen in the current version of pCSDT's, is that if the image is saved somewhere inaccessible to the user that is currently executing the program, it will not appear. The solution in the pCSDT's was to encode the image in Base64 and embed it in the save file. This will not work with Waterbear because:

- We would need an additional Javascript function to convert from Base64 back into an image. Adding this would break the intellectual concept of the “script” in Waterbear
- We would end up with really big text files which are hard to debug
- If we also plan on including video or audio, the size of the files will grow dramaticaly bigger. Accessing them on disk will become annoying, as we need to load the entire file into memory before we can parse it (JSON)

Difficult to find bugs due to the application design

It is more than probable that bugs will be introduced to the program. Because of the way Waterbear was designed, tracking down these bugs will be difficult and time consuimg. There are not logging features in the code, and the files that actually get run are different than the files that were used to write the code.

The tools available to debug Javascript programs are somewhat rudimentary. You can call console.log and log to an output, without any concept of log level or formatter, or you can run through the program step-by-step in most web browsers debugging tools. The issue here is that most of the libraries have been minized, which means they are equivalent to “assembly” in Javascript, and nearly impossible to debug. The nature of Javascript results in lots of asynchronous things going on, making a stack trace almost impossible to follow.

Another complication is that because Javascript is dynamically casted, we may not get an error about passing a function the wrong variable until we are so deep inside that we have no idea where we are. Or, there won't be an error. Just some weird bug that we can't track down.

Benefits of using Waterbear

Working NOW

Waterbear is working. We can show it to people, they can play with, and it works. It is very tangible, and for the most part doesn't have any real issues.

Limited Community Support

Waterbear has a community, and we have contact with the lead developer. There has been some interest in the project from groups other than us, which means we may be able to find help from other people.

Benefits to starting over

We get to design the system

Some of our requirements are NOT provided by Waterbear; for that matter, Waterbear's design is almost the exact opposite of what we need. In these cases, designing our system from scratch will allow us to look at other incarnations of this project and see what worked, what didn't, and from this information we can design a better system.

We get to choose the language

Javascript stinks. It should not be used for making large applications UNLESS you are willing to use hundreds of libraries, and are dealing with a small enough team that you don't have to worry about people misunderstanding what's going on.

Proper technique from the start

One of the biggest problems we've had with the previous version of the pCSDT's is that there is no documentation, tests, or explanation for why certain changes were made. This makes it difficult for new people to come into the project, and makes it more likely that after they do get involved, they will do something that does not "jive" with how the original designed imagined things working.

Disadvantages to starting over

We get to design the system

This will take time. Initially, we will not have anything to show. It will take months, especially if people are busy during the school year. We will NOT have anything people can actually use for a long while. There is debate about how long of a long while, but it won't be next week.

Possibility of the project just flopping

There is always a risk that we simply fail at this project. Reasons could be a bad design, the team falls apart, or we finish everything only to realize we did it all wrong. By using a agile methodology, we can hope to avoid a large majority of these problems.

Made up time frames

These are time frames that, while completely fictional, are what I believe we should expect in terms of how long it will take to do things.

Waterbear

WBS

1. Implement Object metaphor
   1. Change Waterbear to read multiple scripts
   2. Change UI to allow you to switch between scripts
   3. Change "codelets" to work with multiple objects/think OOP
2. Make Waterbear assign "costumes" to objects, and allow changing of costumes

        1. Implement some sort of AssetManager
        2. Implement UI to select costume
        3. Write code to show costume when image is rendered
            1. We **might** have to rewrite a lot the code to render the stage, as we will need a "render" function, which currently doesn't exist
    3. Implement code to save costumes
        1. We will need to redo how all the save files work, which will require rewriting a large section of code
            1. Might break ALL the things

Implement Object metaphor: 75 hours

Make Waterbear assign "costumes" to objects, and allow changing of costumes: 60 hours

Implement code to save costumes: 40 hours

Total: 175 hours

We work **maybe** 10 hours per week (not counting time at school), so we could expect 17.5 weeks. About 4 months. One fo the other big problems is that we can't divide up these tasks; each one will require that one person have access to all the code, and coordinating the project will be a head ache.

After we implement these things, because we are modifying the design to do things it wasn't designed for, we can expect a lot of bugs. The features also won't feel like they were "built in" to the program.

Starting over

WBS

1. LogManager
    1. Most languages have a log manager. We might just need to find it and use it.
2. SaveManager
    1. This is going to be a bigger one. I would strongly suggest using a Zip archive, which we may need to implement outselves.
3. ConfigurationManager
    1. This should pretty straight forward. Except for loading and saving, we would be looking at an abstracted interface for a HashMap
4. AssetManager
    1. Also pretty much an abstracted HashMap, with a few other features. Loading from URL's and putting into the hashmap might be the next difficult part.
5. MessageQueue
    1. A well known pattern. Observer pattern here, and mission complete.
6. CodeBlocks
    1. Our actual program. With all the supporting structures in place, this will consist of just creating the UI, writing a Sprite class, and running through the scripts associated with the Sprite.

LogManager: 20 hours

SaveManager: 60 hours (Zip file stuff will take the longest, if it doesn't exist)

ConfigurationManager: 20 hours

AssetManager: 40 hours

MessageQueue: 20 hours

CodeBlocks: 40 hours

Total: 200 hours

Throughout this, we will be developing documentation and tests. These are things that neither Waterbear nor the current pCSDT's have. A benefit to this is that we can do many things at once; that is, we can have up to about 5 people working on the codebase without interacting very much. Therefore, we might say (200 hours / 5 people) or 40 hours total for a total of a month. More realistically, (200 hours / 3 people) or 67 hours for a total of almost 7 weeks. Add 20% for communication costs, and we would be looking at about 10 weeks.

However, I would suggest working groups of 3. The process would be this:

(a) One person writes stubs and documentation

(b) Second person writes tests to the documentation

(c) Third person writes the implementation

With the following cycles:

All three start off writing documentation and function stubs.

Switch!

Person A writes the tests for Person B's documentation, B for C, and C for A

Switch!

Person A implements person C's code, Person B implementations person A's code, and person C implements person B's code

This way we gaurntee the code gets looked over by a few people, and the documentation is accurate.