

Appendix 5.1.1

Department of Computing Final Year Project Technical Plan

Name: Daniel Young

Size: double

Mode: ft

Course: BSc (Hons) Software Engineering

Supervisor: Rosemary Brown

Summary

Project Summary:

I intend to make a development environment for a very simple Windows based Scripting language that allows a computer user with very little, if any programming knowledge to create scripts to perform various actions. The development environment must be simple to use, and include all aspects required to fully create a piece of software.

Challenges:

There are many small challenges that each need to be solved in turn in order for a successful project to be created.

Firstly, the scripting language itself needs to be drawn up. This involves:

- **Identifying which Functions to implement.**
- **Identifying which Commands to implement.**
- **Deciding on a suitable syntax for the language.**
- **Deciding how to implement variables.**
- **Deciding how to implement conditions.**
- **Deciding how to implement iteration.**

In order to draw up a comprehensive language, I will need to base my language on the functionality provided by both the programming language chosen to implement the system as well as the Windows API.

Secondly, an interpreter will need to be developed to the strict rules laid out by the definitions of the scripting language. The interpreter must not deviate in any way; as deviation leads to bugs. This is a challenge as it involves a high degree of error checking, complex algorithms and methods, such as Reverse Polish and Parsing.

Thirdly, a form builder must be built; similar to one provided by the programming language chosen to implement the system in order to let people who use my software for their own applications build forms to interact with their scripts. This includes the functionality to add, delete and modify controls and properties. The form builder must also build a base script, compatible with the interpreter that instructs the interpreter to build a form with the relevant components chosen to be built at design time.

Fourthly, a “compiler” must be built in order to let the scripts run self-sufficiently, without the need of a development environment or command-line driven interpreter.

Next, a comprehensive development environment should be built that ties all of these aspects together into a simple, easy to use environment that includes debugging functionality, compilation and editing tools.

In all, this project touches on many aspects of software development, including algorithms, HCI, Windows programming, interpretation, debugging and file handling. Each pose a challenge, each I relish.

Solutions

There is bound to be more than one way to create this project, but I have only thought in depth about two ways. The first method is to build the interpreter and development environment in C++ Builder, then use simpler languages such as Visual Basic to create the Form Designer. The immediate benefit is that the time taken to build the Form Designer would be much less. Further study into this however showed that this would not be a good option – as it would involve tacky, volatile code in order to incorporate the Visual Basic software into the C++ Project. In effect there would be two separate applications running, and the result from one application would be passed to the other through use of the Windows API. Another disadvantage that makes this method unviable is that any time saved creating the Form Builder would be lost finding out how to integrate it into the main system. This method would ultimately lead to extra bugs. It may also be found that it is impossible to get the two pieces of software to communicate – then it would be too late to go back.

The second option, and the method to be taken for this project is to build the entire project in one large C++ Builder project. This is the ideal scenario because all aspects of the project may interact with each other and one enjoys the benefits of speedier code produced by a professional compiler.

The system at first presents a screen similar to that found in the Borland Development Environment. This system features a text editor where a programmer may type in and edit their code. This environment will include standard text editing functionality (e.g. save, close, open, cut, copy, paste). The system will also include other functionality, such as Play, Compile, Step Through and Debugging Utilities.

Algorithms for Interpretation have already been identified and prototypes drawn up.

The source code will probably be saved in typical text format – this allows the editing of source code through other popular editors, should it be required.

Constraints

My project is not for a specific client and therefore is not subject to many external constraints such as

- Is client available out of business hours?
- Can client loan me suitable hardware?

However, there are other constraints:

- **Lack of source code for new language.**

As this is a new scripting language there is no source code readily available to test. I therefore need to develop several scripts that test every aspects of the language. This is going to take time as I will have no development environment to build or debug the scripts before I run them through the interpreter.

Key Problems

1. Design an easy to learn, yet powerful language.

By researching programming languages on the internet, I have found that at the very least, a scripting language should include the following functionality:

a. Functions

Functions normally perform an action then return a value. It is my place to decide which functions this programming language will require and what parameters each function will require.

It should be blatantly obvious to the coder what each function does.

b. Variables

Variables hold a value which can be both read and set by the user. I need to decide whether variables in this scripting language are typeless (e.g. both strings and integers may be put in them) or typed (only integer may be put in an integer variable).

How many variables allowed is also a consideration, and whether they may be given custom identifiers or not.

c. Commands

Commands normally carry out a task, as normally defined by their name.

Which commands are implemented need to be considered, as do how they are ordered. They must be coherent and easy to understand.

d. If Statements

If statements are basically a question:

“If <something = true> Do <something> Else Do <something>”

This is a standard procedure within programming languages, but it will need to be decided how to separate the “true” block of code from the “false”, also how to ask the question itself.

e. Iteration

Iteration is a posh name for a Loop and like the If Statement, is present in all scripting languages. They allow repeating code easy as it will loop through a section of code <X> amount of times until a condition returns “false”.

It is imperative that a lot of research goes into finding out which functions and commands programmers will need. It is no good including functionality to delete a file and then negate simple functionality such as the ability to add two numbers together.

In order for the language to be “easy to learn”, it must be as close to the English Language as possible; While it makes coding in the language easier, it points to harder interpretation – this will be touched on later.

2. Creating an Interpreter for the above language.

Once the language syntax and functionality has been decided on, work will need to

start on building an interpreter for the language.

An interpreter looks at each line in turn, checks it syntactically, parses it, interprets it and then executes it. It is NOT compiled into machine code, nor is it checked for semantic errors (for example, will the code ever exit a loop once it has entered it). This may give the impression that a compiler is simple to build compared to a compiler and while it is easier, it is by no means simple.

Lets take a look at Functions first off. Functions could be nested, which means that the interpreter first has to work its way through the functions, starting from the innermost and work outwards. Each function parameter must then be checked to see if the type is correct, as is the range, and finally execute the function and return the result. If the parameter list to the function is comma delimited, then it must be noted that if the comma is within quotation marks, then the interpreter must scan until it finds a comma outside of them.

Secondly there are If Statements. If statements could be nested, which means the interpreter has to skip over some code, but not other code. While this is relatively simple for If Statements which go no deeper than one level, notes need to be kept in order for the interpreter to know where to skip to, and at what time.

Thirdly, for the same reason as If Statements, Iteration statements can be nested, which means the interpreter needs to note where to return to, and when to exit the iteration statement.

3. Variables must be searched out and replaced with their contents. As with functions however, all calls to identical identifiers within quotes musn't be replaced with their contents.
4. **Getting forms to interact with the code.**

It is not known how many controls (Buttons, Text Boxes, etc) a user will choose to place on their form. This means that research will have to be done to look into how to create controls on a form dynamically.

Typically in a Windows application, if the user clicks a button then code associated with the button would be executed. A way will have to be determined to detect which button was pressed, and go to the appropriate line of code in the script and execute it.
5. **Finding an easy and efficient way to trap errors.**

Due to human error and the very fact that this is a scripting language, error trapping will need to exist for each and every single error possible with the language; just as Borland provides an in-depth error trapping system for each function. It will need to be determined whether custom error trapping techniques will have to be employed or whether Borland's own error trapping techniques can be used to save development time.
6. **Developing a Development Environment.**

The development environment is hard to design and implement as it must include a debugger to help the coder to iron out errors in their scripts. It must also have the ability to step through the individual lines of code in the script to allow the user to see the change of events with each line of code that is executed.

Aswell as debugging and interpreting functionality, it must also include standard text editing features, such as copy, paste, cut, find, replace, save, open, new and close.

HCI techniques will have to be employed to ensure that the system is easy to use.

7. “Compiling” the script into an executable.

This involves appending the script onto the end of a stand alone interpreter that can be distributed to other computers. Firstly the stand alone interpreter will have to be built and programmed to examine itself for an appended script, secondly the appending process will have to be programmed.

Note that this isn’t genuine Compiling, its more of a “Self-Running Interpreted Script”

8. Building a Form Builder.

This is a large part of the project. It involves building an application to allow the user to build their own Forms. It requires dymanically created controls to be placed on the form, subject to the users wishes, and should return a portion of code that is compatible with the interpreter to instruct it to create a form at run time.

Each aspect of the Form must be customisable, height, width, top, bottom and name are to name but a few of the aspects that will need to be covered.

Risk Analysis

| <i>Risk</i> | <i>Severity</i> | <i>Likelihood</i> | <i>Action</i> |
|--|------------------------|--------------------------|---|
| <i>Unexpected Errors In Interpretation</i> | <i>High</i> | <i>10-15%</i> | <i>Create an error log the coder can submit for analysis.</i> |
| <i>Cannot Develop Compiler</i> | <i>Medium</i> | <i>10%</i> | <i>Develop a command-line application to run the script.</i> |
| <i>Project Development Overruns</i> | <i>Low</i> | <i>5%</i> | <i>Devote more time to project development and/or trim original criteria for project.</i> |
| <i>Cannot Integrate Form Support</i> | <i>Highest</i> | <i>5%</i> | <i>Would have to change project criteria completely. Change the project to a scripting language for the Internet, similar to ASP.</i> |

| | | | |
|--|---------------|------------|---|
| <i>Drawing Up the Scripting Language takes longer than anticipated</i> | <i>Low</i> | <i>15%</i> | <i>Start development on aspects decided on; hopefully development of the system act as a catalyst for coming up with extra functionality.</i> |
| <i>No one willing to aid project development by trying to develop applications with the software</i> | <i>Medium</i> | <i>70%</i> | <i>I will need to develop my own test scripts to use with the Interpreter to test functionality.</i> |

Options

Target Environment

There were two target environments for this system. The first, as I have chosen to create an interpreter for the Windows platform. The alternate method is to build an interpreted scripting language for the Internet – similar to ASP or Perl.

The benefits of a web-based interpreter is that nothing has to be related to the Windows platform. This means both scripts and the interpreter can be cross-platform compliant and increase user base. Another advantage is that no form support would have to be built, making the project easier – but this is precisely the reason I have chosen not to go down this route.

The web-based interpreter plan is my contingency plan. If I happen to fail at integrating form support, I will change my original specification accordingly. The result should still be impressive as I would have to incorporate Internet based support, such as cookies, server variables, and standard I/O.

Development Tools

As briefly touched on earlier, I considered building the Form Designer in Visual Basic. Visual Basic is known for easy form creation and manipulation and it would be perfectly suited – but integration problems would simply make this task harder than spending the extra time building the entire project in C++ Builder.

Other programming languages were not considered, either due to lack of knowledge or inadequacy.

All graphics related material with this project is to be simply made in Paint Shop Pro. The only other alternatives were Paint and Photoshop. Paint is not viable for this project as it is clumsy and basic. Photoshop is identical to Paint in that it is clumsy, but it is far too complex for requirements within this project.

Lifecycles

I have chosen the Prototyping Life cycle as it is most relevant to the project.

1. It allows me to test a small batch of implemented functions at a reduced level
2. I can ignore error handling while developing the interpreter.

3. I can explore potential designs and find out if there are any potential serious hazards.
4. I can also use it to check integration and establish a user interface.

Alternate life cycle could be the Waterfall method, however after a small amount of research it became obvious that this would not be suited to a full last year project. The advantages the waterfall method gives is that it is simple, and it is extremely logical. The next stage cannot be undertaken without the previous being completed. However, this life cycle method is not ideal because it covers:

1. Installation Stage
I do not have any clients that need training in it, nor a previous system to upgrade from.
2. Operation and Maintenance
Ideally, the system should have a minimal amount of bugs, and enhancing comes after the system has been built – e.g. completed and handed in.

It also doesn't cover prototyping. On a system that has to be built correct first time, it is far too risky to employ this life cycle. For a reliable system, many prototypes will need to be built.

System & Work Outline

High Level Design

| | | |
|-----------------------|--------------------------------|---|
| Interpreter: | Open: | Determine if external file or internal |
| | Get Script: | Copy script contents into a TList |
| | Execute Script: | Parse each line Use Reverse Polish algorithm on expressions Convert variables on line to their contents Evaluate functions Execute commands until end of script |
| | Close: | Free up any memory used |
| Form Designer: | Create Form: | Use dynamically created form |
| | Add Controls: | Use dynamically created controls |
| | Edit Controls: | Use an options screen to edit control properties |
| | Delete Controls: | Delete dynamically created controls |
| | Convert to Script Code: | Traverse through each control, convert to code |
| Compiler: | Compile: | Take base interpreter Append script onto end Save as new file name Possibility for custom icons |

Personal Development

I will need to seriously look into how Linked Lists work for Variable Support, Reverse Polish algorithm for mathematical expressions, and an extremely thorough insight into the Windows API to make hard low level functionality easy to use in the scripting language.

Limitations

I believe this project will come off successful, I just feel that the ability to create custom icons on the compiled executables will be unavailable, as the interpreter will be a stand alone executable with the script added on. Borland requires to know the icon to use before the application is compiled – and this is not suitable for the project.

Another limitation will obviously be functionality. There is a lot of functionality in the C++ language and Windows API that due to time constraints, will never make it into the final project. This limits the things one can do with the language. It will however still bring around impressive results.

Project Plan

The plan to be utilised for this project is as follows:

- Project Proposal
- Technical Plan
- Research Interpretation and Compilation Techniques
 - Collect as much information as possible – useful for both Mini paper and project
 - Start Mini paper
- Design
 - Design of Interpreter and Form Designer in Warnier Orr
 - Design the Development Environment (User Interface)
- Start Implementation
- Design testing
 - Test the rules application through use of Dry Run and Trace Tables
- Mini Paper review
- Draft project report
- Testing
 - White box testing
 - Black box testing
- Integration of Interpreter and Form Designer into Development Environment
- Testing of the entire project
 - Black box testing and/or White Box with Beta Testing
- Project evaluation
- Project report

Report Issues

The system will be evaluated with users and compared with existing scripting languages.

Evaluation

I will release the software to other programmers in the University and/or Internet and ask them to create novel pieces of software. The software they create will be helpful in determining if the software is useful. Questionnaires will also be created to gather feedback on the software.