OBJECT ORIENTED DEVELOPMENT OF A MATHEMATICAL EQUATION EDITOR

By

Levi Russell Stahl

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computational Engineering
in the Department of Engineering

Mississippi State, Mississippi

August 2005

OBJECT ORIENTED DEVELOPMENT OF A MATHEMATICAL EQUATION EDITOR

By

Levi Russell Stahl

Approved:

Greg Burgreen Associate Research Professor of Computational Simulation & Design (Director of Thesis)	Hyeona Lim Assistant Professor of Mathematics & Statistics (Committee Member)
Ioana Banicescu Associate Professor of Computer Science & Engineering (Committee Member)	Jonathan Janus Graduate Coordinator of the Department of Computational Engineering

Kirk H. Schulz
Dean of the Bagley College of Engineering

Name: Levi Russell Stahl

Date of Degree: August, 2005

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Greg W. Burgreen

Title of Study: OBJECT ORIENTED DEVELOPMENT OF A MATHEMATICAL EQUATION EDITOR

Pages in Study: 50

Candidate for Master of Science

Computers since their inception have been used to solve engineering problems. Toward support of next-generation, customizable, generalized software, a mathematical equation editor has been designed, developed, and tested using object oriented (OO) programming techniques. The motivating purpose of this equation editor is to allow a user to graphically define mathematical equations to be solved in a computational partial differential equation-based problem solving environment. The OO scripting language Python was used in conjunction with the OO GUI toolkit Qt to create the editor. Analysis of the underlying abstraction of a general equation yielded the key concept of an information-holding bounding box. Such boxes hierarchically contain every character and symbol in an equation. Specific rules were formulated to spatially arrange a set of boxes into a properly formatted equation. Robust insertion logic of alphanumeric characters, mathematical symbols, and common function names was implemented for intuitive point-and-click equation building.

DEDICATION

Dedicated to:

My Wife, for whom I got my Master's Degree

My Father, who inspired me to achieve

Greg, who was the reason I came to Mississippi State

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I.

INTRODUCTION

Using computers to solve engineering problems has been practiced since the advent of the computer. As computers developed, the types of problems they could solve became progressively more complicated, from calculating square roots to approximating flow fields. The computer's interface has also developed, from hard-wired programs and punch cards to an operating system with a Graphical User Interface. The next generation of computers and software needs to be developed. The current paradigm should be scrutinized and questioned to bring simulation into the future.

The future of simulation will involve more flexible software to solve many types of problems. Today's software often has problem solving capability, but tends to be inadaptable, which is a legacy from its roots. It is designed to solve one type of problem, and customization by the user is inconvenient, difficult, or impossible. There is a compelling need therefore to develop new software with new ways of user interaction. This new software would allow modification by the user for solving multiple types of problems. It would have a user interface to set up such modifications.

1.1 Objectives

The objective of this thesis is to develop the Graphical User Interface (GUI) portion of such a future oriented software package. The goal is to program a graphical interface for constructing and editing mathematical expressions, particularly mathematics describing the partial differential equations that govern fluid motion. The significance of this thesis project is that it would be highly customizable, cross-platform, stand-alone, and simple to use, unlike most existing software. When this GUI is tied into a similarly flexible problem-solving environment, a future-oriented software package allowing user customization will be realized.

1.2 Background

What is a "mathematical equation editor?" "Mathematical" means something written in the language of mathematics rather than a spoken language like English or Spanish. The mathematical alphabet includes universally accepted symbols such as the integral sign, vector notation for magnitude, and the summation sign:

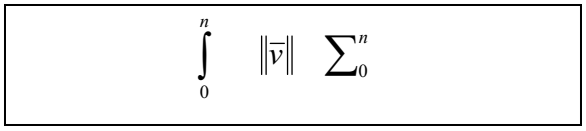

Figure 1.1: Example of Mathematical Symbols

To the scientific mind, these translate easily into ideas. Numbers and standard letters such as *n* are also in the mathematical alphabet. When using mathematical language, ideas are expressed in "equations" or "expressions" rather than sentences. For the purposes of this thesis, both an "equation" and an "expression" will be some equality, such as "$a = b$."

When read in English as “a equals b,” it is a complete sentence, and thus an expression of an idea in a language other than English. Finally, an “editor” is some way of changing the mathematical equation. Specifically, the idea conveyed by the word is to make changes after an equation has been written. Throughout this thesis, “mathematical equation editor” may be abbreviated to “equation editor” or simply “editor.”

How is a mathematical equation editor different than a word processor? Since the written structure of the mathematical language is different than that of a spoken language, different capabilities are required. Instead of writing each character one after the other grouped into words and sentences, a mathematical equation editor writes characters grouped by operations such as adding, subtracting, multiplying, and dividing. And they are not written one after the other. Some characters are on top of others, and some are smaller than others. A mathematical equation editor must also include the standard mathematical alphabet of symbols and lines.

How would a mathematical equation editor be useful in solving numerical flow problems? When approximating flow fields, a standard practice is to use numerical difference equations that approximate the Navier-Stokes equations. There are standard auxiliary equations that go with the Navier-Stokes equations, such as equations of state, body force terms, and viscosity equations. The goal is a flow solver that has an equation editor linked to it so that the user can manually define these auxiliary equations.

CHAPTER II.

SURVEY OF CURRENT RESEARCH

Some mathematical equation editors are available today. Generally, however, this field is new. Compared to other software, equation editors are not as readily available. Research was done on what kinds of similar software are in existence. To suit the purposes of this thesis, an editor would need to be customizable, simple and intuitive, cross-platform, stand-alone, and written in a computer language ideal for prototyping.

2.1 Mathematical Language Formats

The first major standardization of a graphical mathematic language was TEX (pronounced *tech*). It was developed by Donald E. Knuth in 1978, and a second version in 1982 [1]. It is a programming language that allows mathematical expressions to be inserted seamlessly into documents. The program creates the documents, not just the equations. A language-based typesetting program like TEX would not suit this project, since the goal is a graphical representation. It is not an editing tool, and does not have an user interface. The user simply writes code. For example, "x^3" in TEX is [2]:

```
$ {x}_{}^{3} $
```

LaTeX was developed later as a TeX macro package that provides a document processing system. It was originally written by Leslie Lamport [1]. However, it is still far too complicated to use for the purposes of this thesis.

MathML (Mathematical Markup Language) format is a language for expressing mathematical expressions like TeX. Using Presentation MathML, "x^3" would look like [2]:

```
<math xmlns="http://www.w3.org/MathML">
   <mrow>
      <msup>
         <mi> x </mi>
         <mn> 3 </mn>
      </msup>
   </mrow>
</math>
```

Most of the software investigated in this thesis uses the MathML format. Like TeX, it is a pseudo-mathematical language, and not a graphical editor in itself. It is similar in look and feel to HTML because of its use of markup tags [3]. The significant difference from TeX is the use of multiple lines of code. TeX uses a single-line format. Neither are suited for the thesis project, because extensive training is needed to build an equation. A more intuitive interface is needed for this thesis.

2.2 Commercial Software

Graphical equation editing tools were developed in the 1990s, one of the first being MathType (Design Science, Inc., Long Beach, CA) [4]. MathType was one of the first WYSIWYG (What You See Is What You Get) equation editing packages. It is still in wide use and is bundled with word processors such as Microsoft's Word for inserting equations into documents. It has a simple user interface and wide variety of mathematical

symbols available for insertion. It provides a simple, intuitive way to construct and edit equations with pull down-menus, a cursor, and point-and-click selection. Although it provides a good template for an equation editor, MathType is commercial software and does not allow user customization. It therefore cannot be used for the purposes of this thesis.

Mathcad (Mathsoft Engineering & Education, Inc., Cambridge, MA) is a complete mathematical equation editing, solving, and plotting software program. It allows the user to construct multiple equations, define variables, and plot solutions. It combines much of the functionality of a word processor and mathematical solver software, allowing complete report-quality documents to be created. In this way, it is much like a graphical version of T_EX. The equations and plots are dynamically linked, giving instant results when a variable is changed. While this functionality works well as a stand-alone program, it is beyond what this thesis would require. All that is needed is an equation editor. Also, the purpose of Mathcad is to create self-contained documents rather than equations for exporting. Finally, since Mathcad is commercial, it does not allow user customization.

Hermitech Formulator (Hermitech Lab, Zhitomir, Ukraine) [5] is another equation editing application. It provides essentially the same functionality as MathType, namely, the user can write, edit, and save equations in MathML format or as an object that can be inserted into a word processing document. Its user interface is intuitive, but not as well laid out as MathType's. It uses tabs instead of menus, and finding a character or symbol for insertion can be difficult. Also, no customization of the code is permitted. Therefore,

it can not be used for the purposes of this thesis. It does, however, give another perspective on how a mathematical equation editor could be approached.

2.3 Open Source Software

Another option is an open source equation editor. An open source editor will allow the user to access its source code. This customization is a main stipulation for a suitable program. The editors investigated include Mathcast, kFormula and Swift. There are others in existence, but these give an accurate cross section of what functionality is offered in the field today.

Mathcast [6] is an editor similar to the aforementioned programs, released under the GNU General Public License. Its classes are written in C++. It can save equations in MathML format. Its user interface is comparable to Formulator or MathType. It has menus of characters and symbols that can be inserted. One distinctive feature of Mathcast is the "Rapid Mathline." It is a line at the bottom of the interface in which the user enters the equation. The equation is then displayed in the main window. While the Rapid Mathline has its benefits, it is not as graphically oriented as is desirable. Also, the Mathcast source code is too big and cumbersome for use in this thesis.

KFormula [7] is an application in the KOffice suite, which is for use on the K Desktop Environment (KDE), an open source Linux/Unix desktop environment. It is also released under the GNU General Public License. kFormula's classes are written in C++, and can save equations in different formats such as MathML or T_EX. A goal of this thesis is cross-platform capability, which KFormula does not have. In addition, its user interface is not as streamlined or intuitive as some other packages. Also, its inclusion in the KDE

package makes it too dependent for the purposes of this thesis. The entire KDE software package is too big for consideration.

Swift [3] is an equation editor that is written in Java, a good prototyping language. It was developed by two undergraduate students as an open-source equation editor. It is also cross-platform, allowing installation and use on different operating systems. It is independent of any bigger software packages. However, its use is not simple. It has two separate parts, an equation editor and an equation viewer. The equation is built in the editor. While in the editor, each character in the equation has a box drawn around it, making viewing it more difficult. To view the equation as it would appear in a document, the equation viewer must be used. While Swift has desirable qualities, its awkward use rules it out of consideration. Also, the source code proved to be difficult to get, even though it is advertised as open source.

Because no other editors, commercial or open source, met all the thesis goals, it was decided not to invest time into integrating them with in-house solving software. So, the choice made was to develop an in-house graphical equation editor. This allowed for the implementation of all the thesis goals, which were a clean, easy to use, cross-platform, modifiable equation editor. This also allowed the implementation of all the positive aspects of the editors investigated into one editor.

CHAPTER III.

SELECTION OF DEVELOPMENT ENVIRONMENT

Once the decision was made to build an editor from scratch, a set of development tools had to be chosen. Namely, an implementation language and a GUI framework had to be chosen. Programming the editor in-house gave the ability to incorporate all the desirable features of the editors researched into one program. Key features desired in the editor were multi-platform capability, ability to handle text, symbols, basic drawings, intuitive and simple use, and customization. Consequently, the implementation language and GUI framework also needed these features. Another desirable aspect of the development environment was Object Oriented Design (OOD) and Object Oriented Programming (OOP) capability. One of the thesis goals was to utilize these techniques.

3.1 Object Oriented Design

Object Oriented (OO) languages and the concept of OOD have been around since the 1960s, but only in the past 10 years have they seen wide acceptance. OOD is a high level way of solving problems that has evolved hand in hand with high level languages. It involves thinking of a problem in terms of its abstract state and behavior. An OO program is made up of objects that each do a different job and pass messages to each other. Objects can be designed, and many instances of them can then be created in the program. Similar objects can be related by inheritance. Inheritance means one object

"inherits" all the traits of another, but may add traits of its own [8]. Objects can also be dependent on each other. This is called composition. Composition means simply that one object uses another object, one of its functions, or information stored in it [9]. Another tenet of OOD used in this thesis project is encapsulation. Encapsulation is the practice of "encapsulating," or hiding, one object's functionality from another. In other words, one object requests that another do some task, but does not, nor does it need to, know how the other does it. This often allows easier debugging and modification [9].

For an example of OO thinking, most anything can be used. It's natural to think of things in terms of objects. For instance, a drum set can be abstracted as several "drum" objects, classified into the different types: high tom, low tom, snare, and bass. There also would be several "cymbal" objects: crash, splash, ride, and hi-hat. Two "stick" objects would also be needed to produce sound when they hit a particular drum or cymbal object.

This type of thinking fits the thesis problem, also. An equation editor is inherently object oriented. In this thesis, the GUI is considered an object, the drawspace (where the equation is drawn) is considered an object, each character in the equation is considered an object, etc. All these objects must communicate with each other to achieve the final goal.

3.2 Object Oriented Programming

OOP is a natural step in the evolution of computer programming. Today's high level languages are ideal for OOP. It is simply using OOD concepts for computer programming. The benefit and goal of OOP is natural, reliable, reusable, maintainable, and extendable code put together in a timely manner [9]. Since the editor to be developed is a prototype, these benefits coincide with the thesis goals.

OOP code is made up of classes that have attributes and methods. A class is an OOP tool. It is the concrete code that defines each abstract object [10]. Likewise, "methods" and "attributes" are OO terms. A method is a function in a class. For example, "hit drum" (going back to the drum set example) could be a method of the drumstick class. An attribute is a variable held by a class [9]. For example, "diameter" or "thickness" could be attributes of the cymbal class.

3.3 Object Oriented GUI Frameworks

The GUI framework (or GUI library), therefore, should lend itself to OOP. A GUI framework is a set of preprogrammed classes and functions. Common objects included in a GUI library are virtual buttons, menus, graphics functions, and window managing functions. A good GUI library will simultaneously ensure reliable code and ease the coding process. It should be powerful and comprehensive, but without being too big or difficult to use.

Some of the GUI libraries currently in use are Motif, the Microsoft Foundation Classes, Lgi, Tk, and Qt. Motif is for Unix systems, and lacks the necessary cross-platform capability [11]. The Microsoft Foundation Classes are bulky and complicated, with little use outside the Windows operating system [12]. Lgi has multi-platform capability, but is too small [13]. Tk was developed for the language Tcl, and has bindings for OO languages such as Python and Perl [14]. It is also platform independent, but is inferior to the last choice: Qt. Qt is widely recognized as the best choice for efficient, rapid, reliable programming today. It has many powerful, reliable functions for any platform, yet remains easy to use and streamlined [15]. Another significant advantage of

Qt is that is was developed specifically for OOP [16]. It was therefore the choice for this thesis.

3.4 Object Oriented Languages

The computing language choice also needs to be OO friendly. It should be noted, however, that OOP is not restricted to any one language. It is a design process, not one type of language [9]. Both system languages such as C and scripting languages such as Java can be used for OOP. However, some languages were designed specifically with OOP in mind, and are therefore better suited for it. One aspect of scientific computing that is expected to become increasingly appealing, especially for OOP, is scripting. It allows complex pieces of scientific computing to be easily accessible via test based scripts and to quickly prototype and synthesize new capabilities. It would help future work if a scripting language was used, even if scripting itself is not used in this thesis. Also, scripting languages tend to be cleaner and faster to program in than system languages. For the purpose of prototyping an equation editor, these are ideal qualities. Thus a clean, easy to use scripting language with Qt bindings is needed.

Popular scripting languages include Tcl, Python, Ruby, lua, scheme, and Java. Each of these has their strengths and weaknesses. The list can be immediately narrowed to languages with Qt bindings, since that is the choice for the GUI library. These include Python, Ruby, Lua, and Java. Python was chosen for several reasons. It is an OO scripting language. It was designed with OOP specifically in mind. It is free and open source. It is platform independent. It is simple and easy to use like a scripting language, but simultaneously has advanced programming capabilities like a compiled language. It

combines features like automatic memory management, dynamic typing, built-in object types, and exceptional ease of use [10]. Its pseudocode-like, intuitive syntax gives a lot of leverage to a beginning programmer, and even more to an experienced programmer. Java is also a scripting language with many benefits similar to Python, but has more complicated syntax. It therefore takes more lines of code in Java than in Python to accomplish the same thing [17].

The final selections for the program development tools are Python and Qt. These choices are explicitly OO, multi-platform, and powerful yet simple. They make a well-rounded and powerful toolkit for solving a graphically oriented programming problem.

CHAPTER IV.

ABSTRACT USER PROCESS OF BUILDING AN EQUATION

In order to effectively organize the design process and approach the problem from an object oriented standpoint, key objects that interact with each other had to be identified. These abstract objects were then made into concrete classes with attributes and methods. This abstract planning step is the first step of OOD [18].

4.1 Initialization

To begin, the steps the user would go through to build an equation and edit it were enumerated. In other words, what using the editor could be like was abstracted from the user's point of view. This process revealed a list of the objects involved in the problem. First, the user starts the program. This will initialize the editor in a window on the screen. This window will have some drop down menus and a blank area for constructing the equation. In this drawspace there will be a cursor and possibly a blank box shape indicating where the first input from the user will go (see Fig. 4.1).

The initialization of the program has revealed five possible objects, all of which are important to the functionality of the editor:

- The Main Window – The window which is the program
- Drop Down Menus – Standard menus such as "File" and "Help," as well as custom menus containing symbols
- A Drawspace – A blank space to display the equation

- A Cursor – For editing
- A Blank Box – A container for the equation

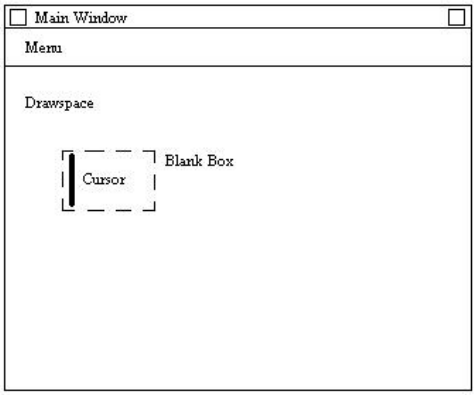

Figure 4.1: Objects Needed for Initialization of Editor

4.2 User Input

Next, the user might type an "x." It will appear on the screen, along with any other characters or symbols the user desires. The cursor will advance after each entry. As an entire equation is entered, each character should be arranged and centered in the equation. These features represent some methods needing support or implementation:

- Add character
- Advance cursor after new character is added
- Resize equation, arrange all characters when new character is added

4.3 Editing

Next, the user might want to erase the “x” they typed and replace it with something else. Essential aspects of equation editing include selecting, deleting, and inserting characters and symbols.

- Selecting a character
- Inserting a character between two existing characters
- Deleting a character

These aspects will also have to be built into the program. The user should be able to select any single character in the equation with the mouse. Some visual cue should signify selection, such as a change in color or highlight. A selected character can then be deleted. New characters should be able to be inserted at any given point in the equation, indicated by the cursor. These editorial objectives could be implemented as class methods.

CHAPTER V.

ABSTRACT COMPONENTS OF A MATHEMATICAL EQUATION

The editor now has three abstract parts. First, the GUI and equation display. Second, the drawn characters of the mathematical alphabet that make up the equation. Third, the editing functions that manipulate the equation. The user manipulates the GUI, which signals the internal editing functions, which manipulate the characters in the equation, which are displayed in the GUI as they are changed (see Fig. 5.1).

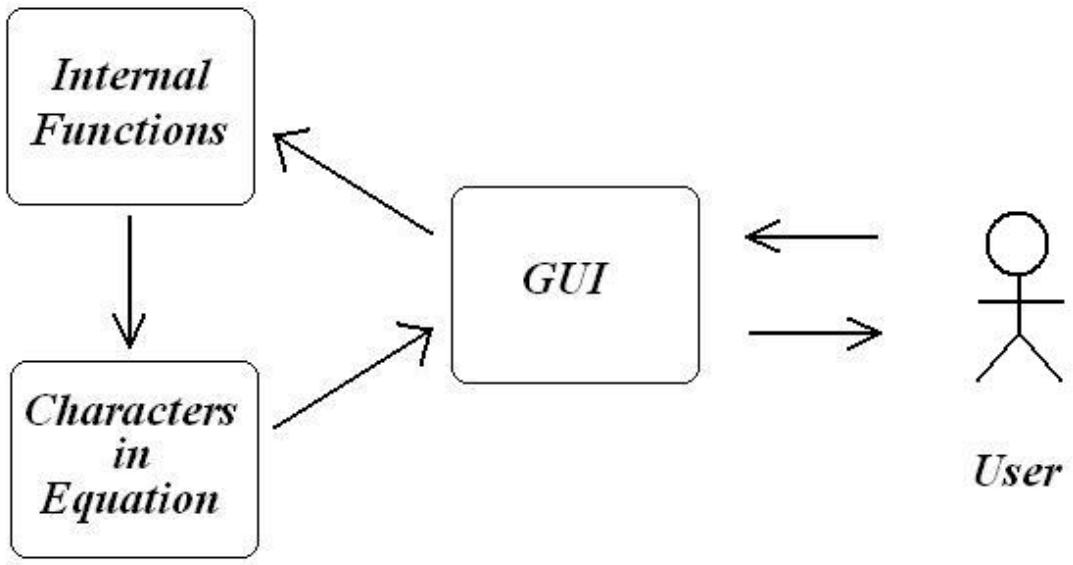

Figure 5.1: Relationship of the User to the Editor and Its Abstract Parts

Laying the problem out this way leads to a main design decision of the thesis. Since the internal functions of the editor act upon all characters in the equation, they must all be the same. Greek letters, English letters, division symbols, and sine functions must all be essentially the same object to the program's internal functions. This is possible if everything in the equation is thought of as a box. By "box," it is meant that each drawn character in the equation is contained in a bounding rectangle. Information about this rectangle could be manipulated to move it, resize it, or delete it. Thus, an abstract equation can be considered a box that contains other boxes.

Instead of the equation being a string of characters and symbols, an object-oriented solution is to have each character be an instance of a character class. Utilizing the OO concept of inheritance, this character class can derive many of its traits and functionality from a generic box class. It can be taken further. In some cases, a single character is inserted into an equation, such as an "x," but other times several need inserted simultaneously with different spatial layouts, such as a summation:

$$\sum_0^n x$$

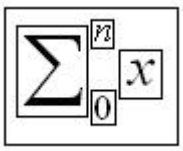

Figure 5.2: Summation Symbol as Boxes

In the summation symbol, four objects or boxes are inserted: the Greek sigma, the small upper bound, the small lower bound, and a "blank box" for the expression to be summed.

In this case the expression is only an “x.” Note that the *n* and the 0 are smaller than the other characters and are placed in the superscript and subscript positions. Similarly, a sine function needs six objects or boxes inserted:

Figure 5.3: Sine Function as Boxes

These box objects used in building an equation were then defined more concretely to allow for the proper visual result. Information needed by each type of object dictated if any were inherited by any others, or if they were inherited from an existing Qt class. The object types that make up the equation itself are Box, Character, StretchLine, and Function. Note that these titles are capitalized to refer to the specific object definitions, rather than the general abstractions being used before.

5.1 Box

A Box is an empty rectangle. It must manage other boxes in it, meaning it automatically sets the spatial layout of any other boxes inside it. It therefore must be able to have children. In programming, a parent-child relationship is simple: the parent knows who its children are and vice versa. This makes it easy for the parent to communicate with its children. Objects outside this relationship are not aware of each other unless a

function explicitly makes them. In the case of this thesis, a child Box is also spatially contained inside its parent Box.

Additionally, the Box object must know its position in the drawspace, and the position of its children. It adjusts its own size to fit around its children. The Box is the fundamental building block of the whole equation. The entire equation is contained in one of these, and consists entirely of these, as well.

5.2 Character

A Character is a specialized Box with a single pixmap (either a character from the English or Greek set of alphabetic characters or a mathematical symbol) drawn in it. This pixmap moves with the Character object as it moves. A Character object has no children, and only one pixmap drawn in it. Thus, a new Character object is created for each symbol, line, or letter inserted into the equation.

5.3 StretchLine

A need was found for a special type of Character called a StretchLine. This is simply a horizontal or vertical line that stretches to remain the same width or height as its parent. An example is the line in a division symbol. If more characters are added to the numerator or denominator, the line should “stretch” with it, remaining as wide as the expression (Fig. 5.4):

$$\frac{1}{2} \xrightarrow{\textit{becomes}} \frac{1+\Delta\theta}{2}$$

Figure 5.4: Example of Stretching Line

5.4 Function

A Function object is merely a blueprint that builds one object from the other types of objects. For example, a division symbol would use two blank Box objects and a StretchLine object, all contained in a parent Box object:

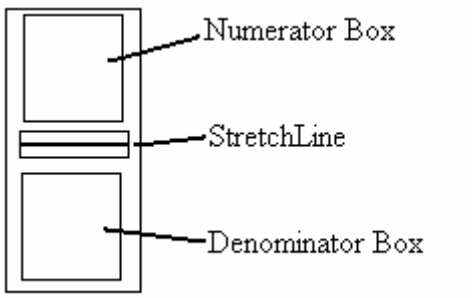

Figure 5.5: Example of a Function Object

In this way, the Function class addresses the problem of multiple characters being inserted at the same time. Blank Box objects are inserted which the user fills with more Characters or Functions. This allows for any mathematical expression to be entered into any Function. A Function object list was made that included most major trigonometric functions, log functions, and derivatives as well as others like absolute value.

CHAPTER VI.

ABSTRACT REPRESENTATION OF A MATHEMATICAL EQUATION

With all the objects used in an equation defined, any equation could be broken down into its parts. At this point in the development, several complex equations were laid out in terms of the abstract objects. The iterative process of writing out these equations in a hierarchal form, then modifying the object definitions ensured thorough logic. For example, the simple equation

$$a = \frac{F}{m}$$

Figure 6.1: Representation of an Equation – Proper Form

can be drawn with boxes as (Fig. 6.2)

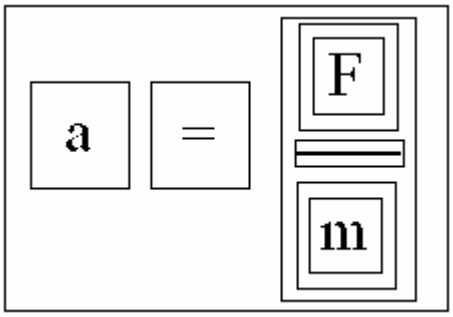

Figure 6.2: Representation of an
Equation – Box Form

which is simply four Character objects, one StretchLine object, one Function object (divide) and one parent Box object (which contains the whole equation). Notice that although there are only four characters drawn, nine Box objects are required to contain them and organize them into the proper mathematical form.

From the Box drawing, the hierarchal form of the equation can be reached. Writing an equation in this hierarchal form showed efficiently which children belong to which parents and how many levels the equation had. As seen below, even a simple equation has four levels (Note: Character is abbreviated Char in Fig. 6.3).

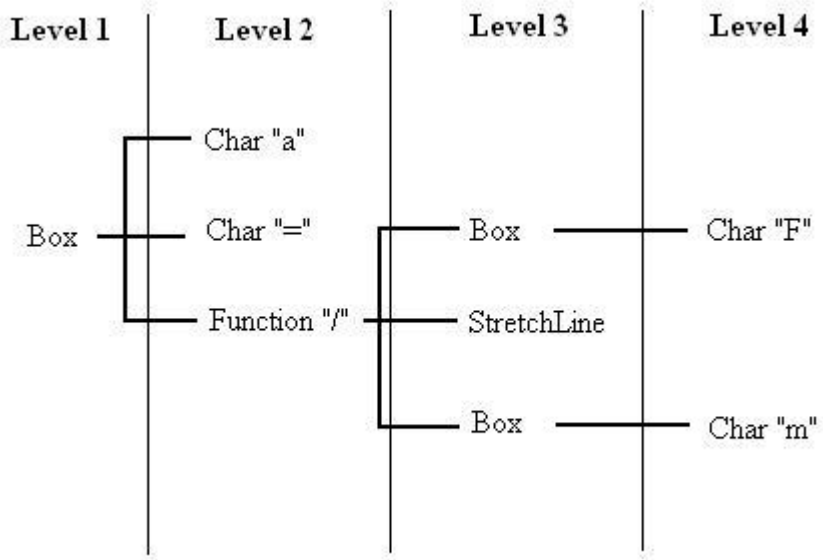

Figure 6.3: Representation of an Equation – Hierarchal Form

It is clear from Figure 6.3 that there are nine Box objects, and how the hierarchy of parents and children is laid out. The more complex equations modeled in this hierarchal form required many more levels. For instance, the equation

$$\phi = \tan^{-1}\left\lceil \frac{-\upsilon \sin(\beta)}{\left\lfloor 1-\upsilon+\cos(\beta)} \right.\right.$$

Figure 6.4: Example of a Complex Equation

required eight levels and thirty-nine Box objects (Fig. 6.5).

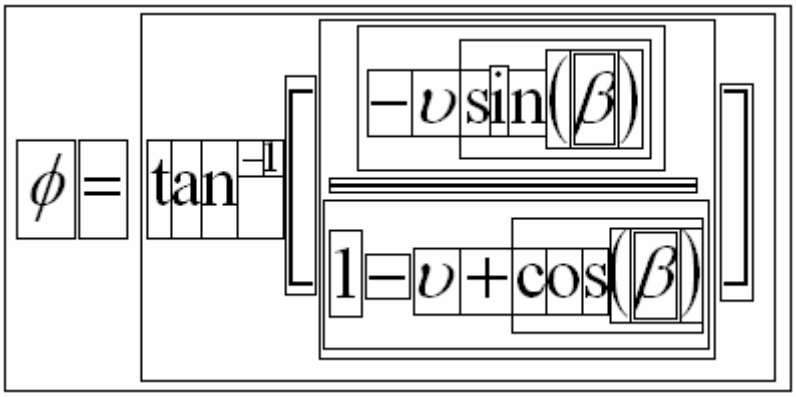

Figure 6.5: Example of a Boxed Complex Equation

Clearly, the proposed object abstraction allows complex equations to be systematically and logically handled using a robust and simple Box based data structure.

CHAPTER VII.

DEVELOPMENT OF THE CODE

The next step in the design was developing the actual classes and their methods and attributes. The abstract Box, Character (Char), StretchLine, and Function objects were proposed as the basic equation object classes. The abstract internal functions (i.e. deletion, selection, insertion) were developed into methods and channeled into the appropriate classes. The abstract "Main Window" object was proposed as two classes, MainWindow and BoxWindow. Other necessary classes include Layout, Cursor, BoxRegistry, and Error. Each class is explained in detail in the following subsections.

The class relationships can be modeled in a Unified Modeling Language (UML) diagram. UML is a tool for conveying OO relationships in a program. A class in UML looks like

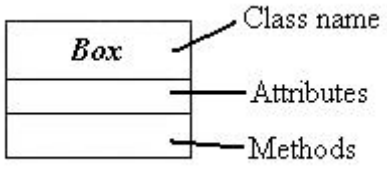

Figure 7.1: A Class Modeled in UML

Note that in Figure 7.2, class attributes and methods are omitted for clarity. It is an overall picture of the editor's design. A solid arrow in UML indicates inheritance and a dashed arrow indicates dependency [9] [18]. What Figure 7.2 does not include is the inheritance from Qt's classes. For example, MainWindow inherits from QDialog, a preprogrammed dialogue window in Qt. Box inherits from QFrame, a Qt class with relevant attributes such as position, size, and children. Whenever possible, Qt's classes were used in this way, which sped up the development process. Refer to this diagram if needed during the explanation of each class in the following subsections.

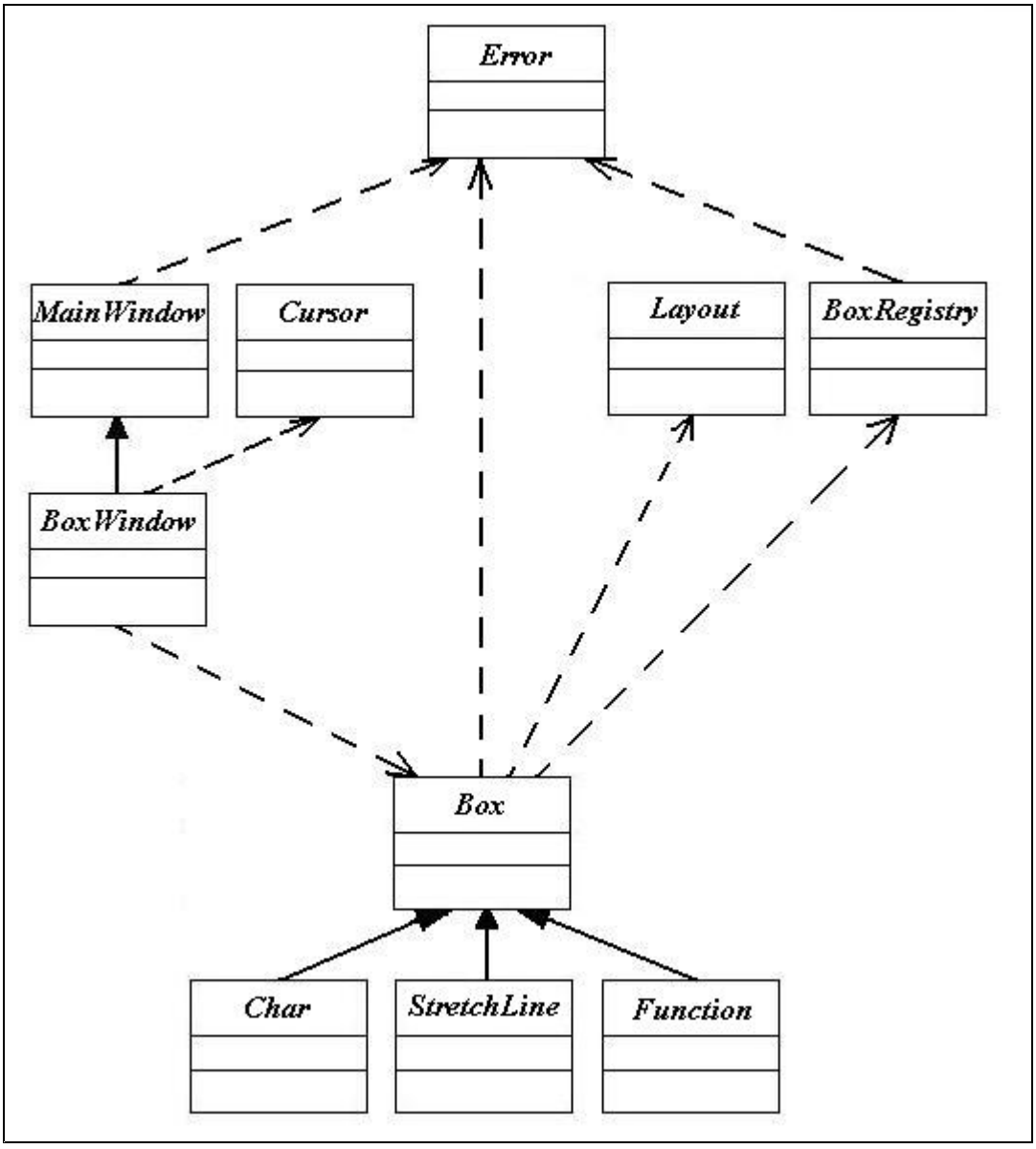

Figure 7.2: UML Diagram of the Editor's Classes

The coding of the editor's classes was done in a series of development stages. The first stages addressed the most basic concepts and classes. Each successive stage built on the last, and the program was built. Each stage did not necessarily produce one class. Some stages of development produced no classes, and some produced several. The process was logically walking through the problem in an OO manner. Also, before the code was written, pseudocode (informal notation describing how the code will work) was written. This is known as the pseudocode programming process [19].

7.1 Selection Functionality

The first stage was simply drawing an empty Box with specific coordinates, and being able to select that Box. Selection is an important feature of the program, but it is not necessarily a separate class. It works best as a method of a Box object. This also assures that every Box object in the equation, and every object that inherits from it, will have that selection functionality. A function was written, and a test confirmed that the user was able to click inside the Box with the mouse and get a message printed to the screen. A mouse click outside the Box would yield no message.

7.2 Allowing a Hierarchy of Parents and Children

Next, the hierarchy of Boxes needed established. First, the parent-child relationship needed to work. A second Box was added as a child of the first Box. The solution was the addition of a list of children as an attribute of the Box class. However, later, the built-in Qt methods of the QWidget class `addChild()`, `children()`, and

`insertChild()` (inherited by both QFrame and QDialogue) were used to replace some of the custom methods developed. This was not an uncommon experience as the editor was being written. Often, after a method was programmed, a Qt function would be discovered that provided the same or similar functionality. This was due in part to the large volume of Qt classes. However, as the project progressed, the Qt functions became more familiar, and were used more frequently. This reveals one of Qt's strengths. The documentation available online or with a copy of Qt is thorough and helpful.

7.3 Hierarchal Box Layout

Once one Box could be added as a child of another Box, a hierarchal system could be implemented. Adding multiple Boxes raised the problem of spatial layout. With more than one child Box, the parent Box needed to spatially arrange them. It was decided that this functionality would be added as a method of the Box class. Therefore, a layout function capable of lining up children horizontally or vertically was programmed. The parent Box was also made to resize itself to fit around all its children, no matter what their dimensions.

As this layout function grew, it was removed from the Box class and made into its own class, called Layout. The Box class has a dependent relationship with the Layout class. It calls it to spatially arrange its children. A tenant of OOP is to keep each object focused on performing one task. With the Layout function in its own class, the Box class was not only cleaned up, but maintenance and debugging were simplified. Now a problem with special object layout can be addressed in the Layout class, and the Box class can be left alone.

Layout doesn't inherit from anything and has one attribute, `self.layoutPolicy`, that can be set to horizontal, vertical, superscript, or subscript. The Box class calls the Layout method `updateLayout()`, letting each Box organize its children. The Layout logic sweeps up through the hierarchy of objects to the first box, adjusting the position and size of each Box when a new Box is added or deleted. This gave the Box class most of its needed functionality.

7.4 Rendering a Character in a Box

Next the abstract Character object was coded into a concrete class, named Char. It is based on Box, and inherits all of Box's attributes. In other words, a Char instance is a Box instance, but with a few key differences. First, a pixmap of a mathematical character is drawn in it. Second, a Char instance has no children. Third, the size of the Char can not be determined by its children, since it has none. Rather, it obtains the dimensions of the pixmap printed in it and sizes itself to it.

Drawing a pixmap in a Box was the first step. Qt provided the needed methods for this with the QFontMetrics class. For any given character, this class provides information like width, height, and rightmost coordinate. The Char class adds only three attributes to those inherited from the Box class: the ASCII character (`self.char`), the font (`self.font`), and the font color (`self.fontcolor`). Its only method is the overloaded Qt function `QPainter.paintEvent()`, which draws the ASCII character as a pixmap. The font attribute allows changes to accommodate the Greek alphabet and mathematical symbols such as "⌠" (the top half of an integral sign). The font color

attribute is not strictly necessary, but is present to give an easy way to change a character's color when it is constructed if needed.

7.5 Rendering a Stretching Line

Following the design of the Char class, the abstract StretchLine object was made into the class StretchLine. It also inherits from Box, and is very similar to the Char class. Instead of a character pixmap, it draws a single line. The only new methods are `setLength()` and `setHeight()`, which set its width or height to the parent's width or height. The StretchLine can be oriented two ways, horizontally or vertically. The horizontal StretchLine is used for division symbols, and the vertical StretchLine is used for the absolute value function.

7.6 Building Functions

With the Box, Char, and StretchLine classes coded, the abstract Function object could be coded as well. Initially, only the division symbol and sine function were coded, each in their own class. They were tested. When they were working, a list of desirable mathematical functions was made. These functions were coded and put into a single module made of many classes, called Function. A module is a file that contains more than one class. The Function classes are short and simple, because they are merely templates made from Box, Char, and StretchLine. Each Function class inherits from Box. They contain no new attributes or methods. Their purpose is only to assemble. They provide a convenient way to do what the user could do manually. These classes were then tested to ensure all previous logic such as selection was intact.

The relationship of the Box class to Char, StretchLine, and Function is represented in the following UML diagram (Fig. 7.3). It is readily apparent how important the Box class is, as the others depend heavily on it for most of their functionality. Note that the Function module is a blank box in UML, because none of the Functions have methods or attributes of their own.

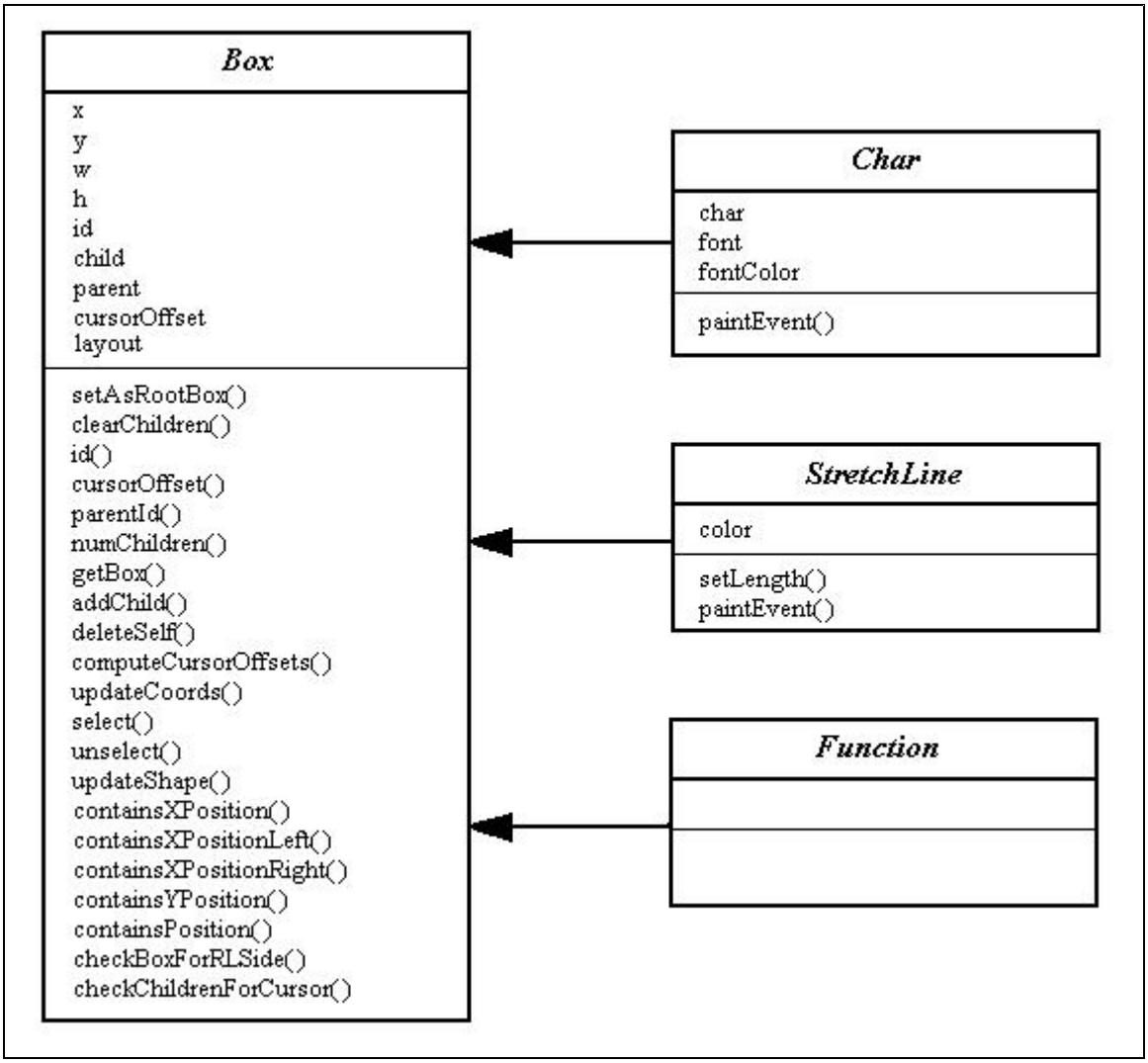

Figure 7.3: UML Diagram of the Equation Object Classes

7.7 Insertion and Deletion

After the equation displaying classes were developed, the next stage of development began. The internal editing functions insertion and deletion were coded. Inserting was defined as clicking somewhere with the mouse and adding a new Char at that point in the equation. Deleting was defined as simply removing a Char from the equation. It was decided that the left mouse button would pick an insertion point, and the right mouse button would delete a character. To implement these methods, the selection logic was refined to return more specific information about each mouse click. Instead of returning only which object had been selected, functions were added that provided additionally whether the object had been clicked to the left or right of its vertical center line. Once the selection logic was modified, the `addChild()` method (in Box) could discern where to insert the new child: to the right of, left of, or in the current active Box.

When tested, an issue arose with the custom made methods and Qt's methods. Before the new `addChild()` would work, both the Box attribute `child()` (a list of children's id numbers) and the Qt attribute `children()` (a list of objects kept by any QWidget object) needed to be informed of the new object.

7.8 Rendering the Cursor

Finally, the visual editing tool of the equation editor, the Cursor, was implemented. It is designed to be a green line drawn to the right or left of a Box. It also includes a line drawn under the current active parent Box. This indicates which Box the user is in as well as what two objects they are between. This is a helpful feature present in other equation editors that were researched. The logic of the Cursor goes hand in hand

with the logic of insertion. The Cursor class obtains the same information that the insertion function does, and the Cursor drawn at the same point in the equation. The Cursor class is not inherited from any other class.

7.9 BoxWindow Class: The Main Construct

The BoxWindow class was developed along side the preceding classes throughout the various stages. It was designed to be the abstract drawspace for the equation. It was therefore the logical construct for the other objects. By construct, it is meant that user input into the BoxWindow class creates instances of the equation object classes. The BoxWindow class both receives the input and calls the appropriate class. It contains the two construct methods for Char. English is input through the keyboard, and Greek is input through a pull-down menu. It also contains the constructs for Function, input through a pull-down menu. It initializes the first empty Box upon startup, and handles other user inputs such as mouse clicks. It also keeps track of the current active Box, which is an important attribute for insertion and deletion.

7.10 MainWindow Class: The GUI

The MainWindow class, like the BoxWindow class, was developed along with other classes. It is essentially the user interface design and pull-down menu design. Any additional Qt GUI features such as buttons would be added to this class. The goals for the GUI were easy access to functionality, intuitive operation, and clean aesthetic features. Many features mimic relevant traits of commercial or open source editors that worked well. Qt's classes such as the pre-assembled button classes, menu classes, and others

were used directly for operation of the editor. A nice feature of the cross-platform nature of Qt and Python is that the editor takes on the aesthetics of whichever operating system it is being run on.

7.11 Error Module

Early on, a simple module was made to handle errors. It was named Error. It merely provided a standardized, easy way to allow a method to report an error. The module then printed the error message to the screen.

7.12 BoxRegistry Class

The BoxRegistry class helps the Box class keep track of all the Box instances currently running. It is called by the Box class when a new Box is created, deleted, or needs to be accessed. It is based on a singleton design pattern, which allows only one instance of a class at a time, with easy access to the one instance [20]. A singleton is one of many OO code design patterns used by OO programmers.

CHAPTER VIII.

RESULTS AND DISCUSSION

The final equation editor was named Equator 1.0, because it equates things, and because "Equator" is a combination of the words "equation" and "editor." Overall, the results were good. The editor has the capability to write almost any mathematical equation. Figure 8.1 is a screenshot of Equator when it has been initialized.

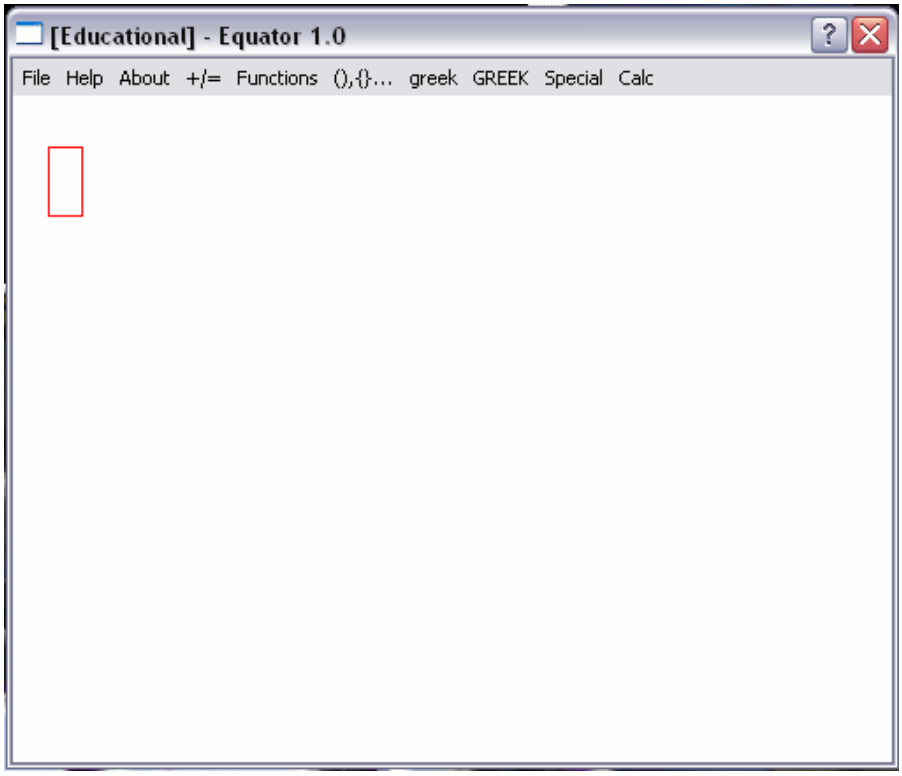

Figure 8.1: Equator 1.0

8.1 The GUI

The graphical interface part of the program consists of pull-down menus for the various functions and characters to be inserted. There is also a File menu, Help menu, and About menu. The File menu provides a way to exit the editor. The Help menu contains information on the mouse buttons. The About menu contains development information.

The other pull-down menus are for building equations. There is an Operators menu, represented by "+/=." It contains the basic mathematical operations: addition, subtraction, multiplication, division, exponent, and equals. There is a Functions menu:

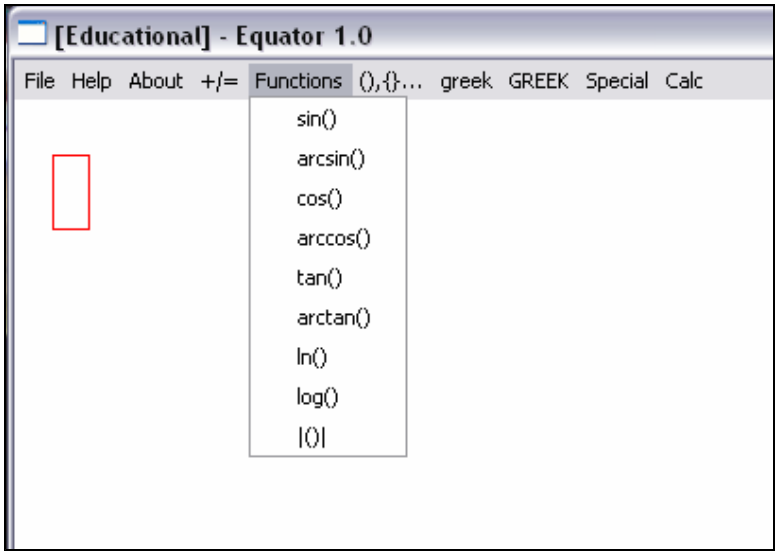

Figure 8.2: The Functions Menu

There is a Brackets menu represented by "( ),{}…" populated with various mathematical brackets. There is an uppercase and lowercase Greek symbol menu:

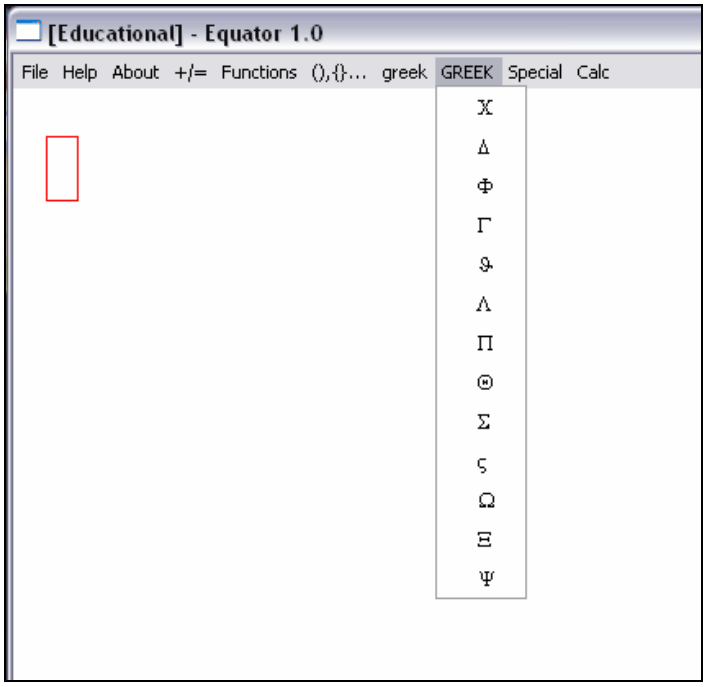

Figure 8.3: The Uppercase Greek Menu

And a Special menu containing the Superscript and Subscript functions:

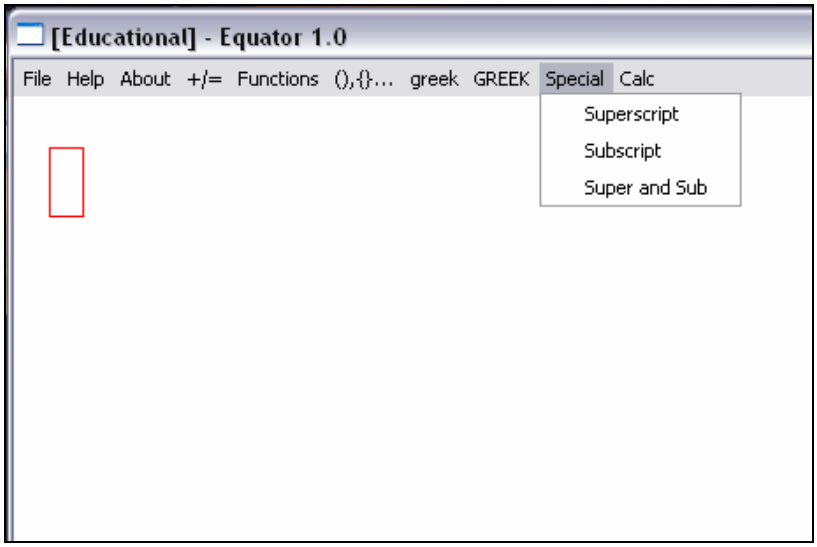

Figure 8.4: The Special Position Menu

Lastly, there is a Calculus menu abbreviated “Calc.” It includes first, second, and third derivatives. The user can fill in both the numerator variable, and the denominator variable. In other words, they can determine what is being differentiated, and with respect to what.

A positive aspect of the GUI design is the user has quick access to all functionality. Also, the menus are not complicated or overcrowded. When an empty Box is selected with the mouse, it is indicated by a red outline. The graphical insertion and deletion work well. When the user clicks at a point in the equation, the next character inserted appears at that point.

8.2 The Representation of a Mathematical Equation

The representation of any mathematical equation is clear using the Equator. The following screenshots show a partial differential equation (Fig. 8.5), an equation with Greek symbols and subscript (Fig. 8.6), and a numerical difference-type equation (Fig. 8.7) built in the Equator. Note that in Figure 8.5, the lowercase greek delta is used instead of a partial derivative ∂. This is because the character ∂ is not available in all computers’ font sets.

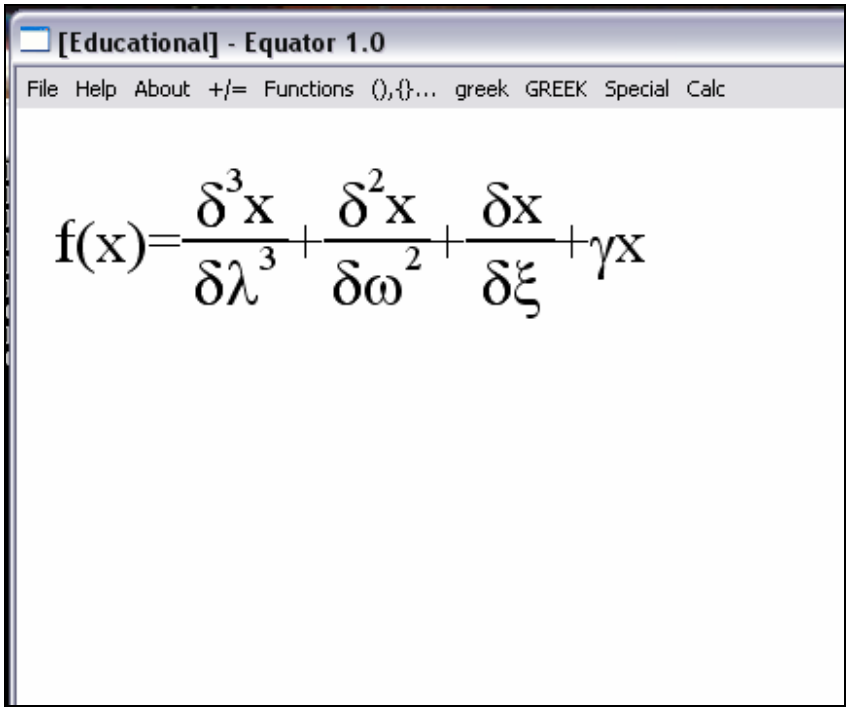

Figure 8.5: A Partial Differential Equation Built in Equator 1.0

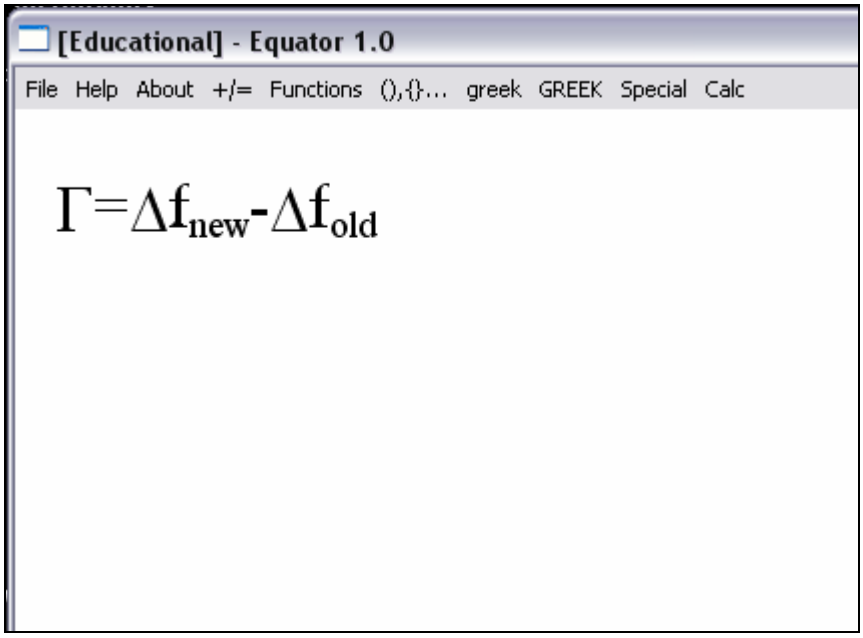

Figure 8.6: An Equation Using Greek and Subscript Built in Equator 1.0

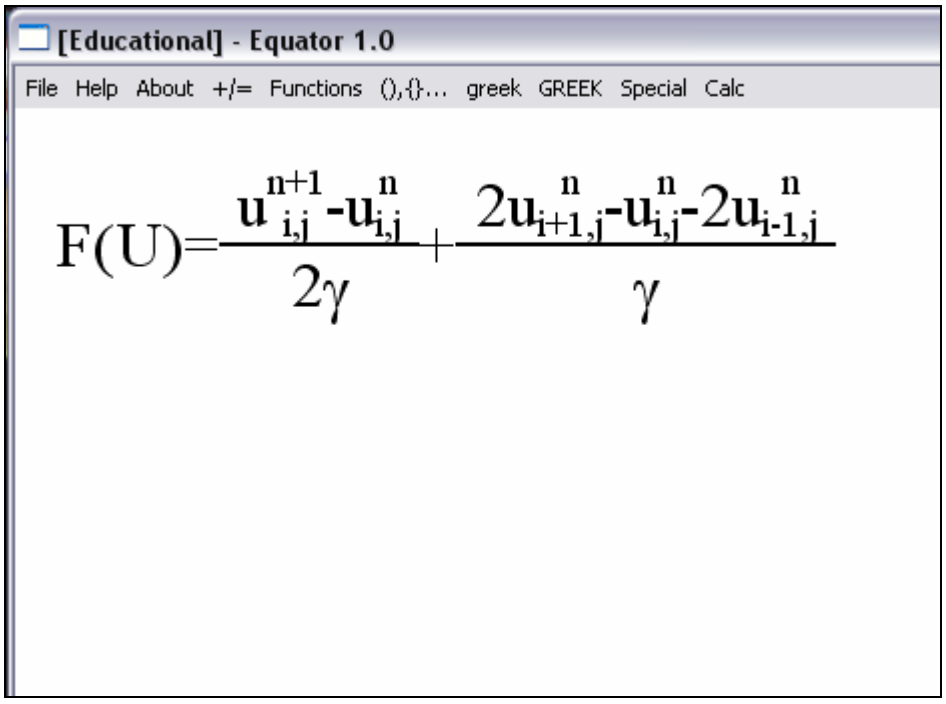

Figure 8.7: A Numerical Difference-Type Equation Built in Equator 1.0

There are many positive aspects of the final editor's ability to display equations. The equations are big and clear. Special characters in the superscript and subscript positions work very well. The "Super and Sub" menu item allows both superscript and subscript to be applied to a character. Another special character, Stretchline, functions well. It adjusts itself to its parent for proper looking equations. All the characters of the keyboard and pull-down menus are displayed fully and correctly.

8.3 Comparison

Comparing Equator 1.0 to current software shows the similarities. Microsoft's Equation 3.0 is similar to MathType, but not as powerful. Equator 1.0 has menus similar

to Equation 3.0. However, while Equator's menu titles are both written out and symbolically displayed, Equation's menu titles are all symbolic.

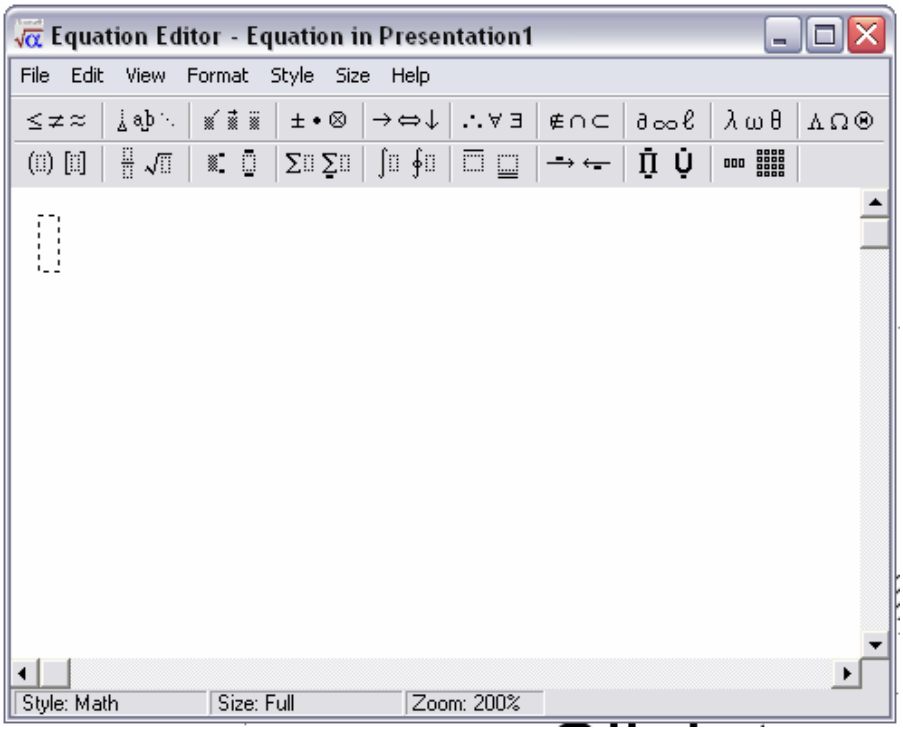

Figure 8.8: Microsoft Equation 3.0

The same equation (from Fig. 6.4) was built in each editor. Each mouseclick and keystroke was counted. Equation 3.0 took forty mouseclicks and keystrokes to build the equation. Equator 1.0 took only twenty-seven to build the same equation, due mainly to its pre-built Functions arctangent, sine, and cosine. The resulting equations are in Figures 8.9 and 8.10.

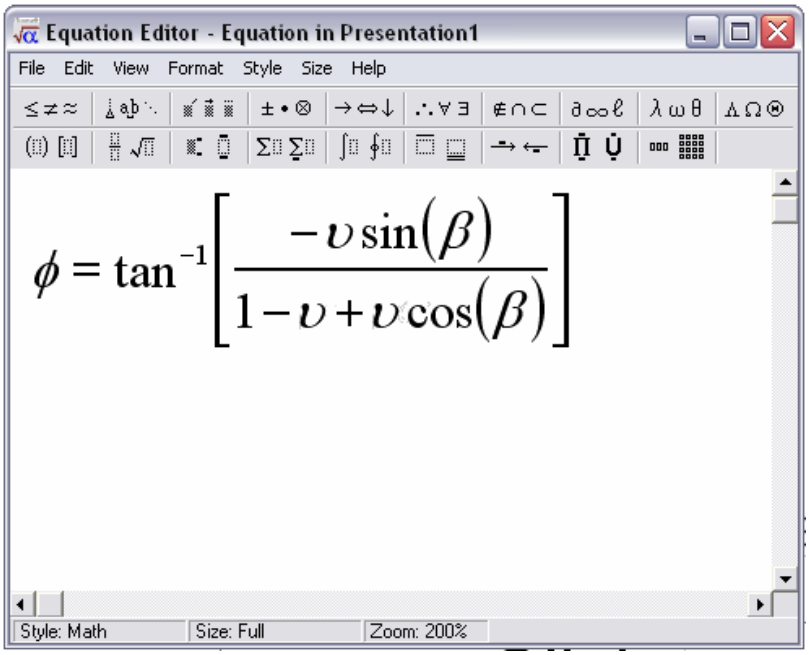

Figure 8.9: Comparison Equation in Equation 3.0

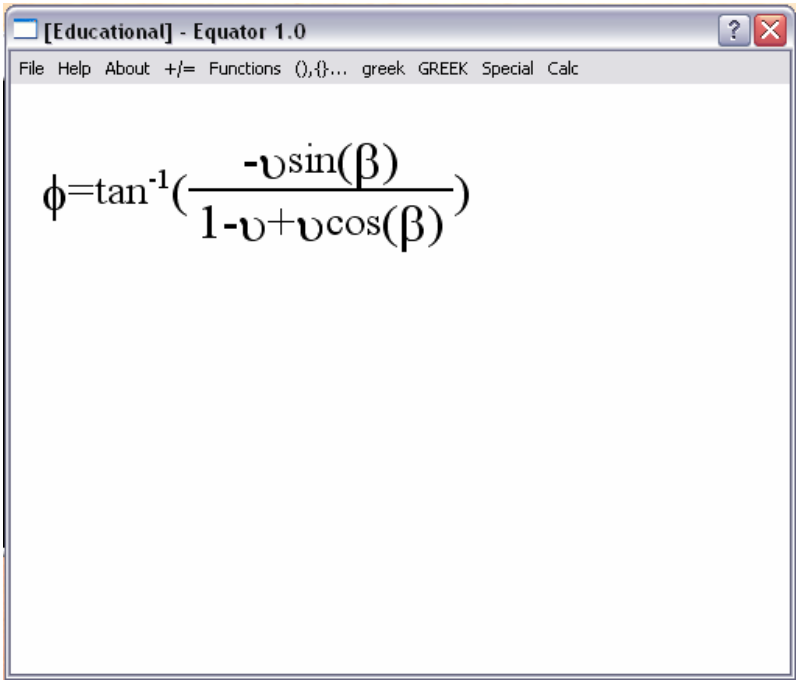

Figure 8.10: Comparison Equation in Equator 1.0

CHAPTER IX.

CONCLUSIONS

The thesis project achieved its main goals. A mathematical equation editor was developed. Multiple types of equations can be easily built and edited. The program code is object oriented. After programming this prototype, the value of OOP and OOD are apparent. Their principles apply to any type of problem. Not only did they aid in solving the problem, but also helped organize it as well.

A custom-built equation editor prototype has now been built, which can function as is, or serve as a solid springboard for a more comprehensive project in a system language. It is lightweight, cross-platform, and highly customizable, yet produces equations more efficiently than other current editors.

Working in conjunction with OO techniques, Python proved a powerful tool. The excellent results are made more so by the rapid development time of the project, which is largely due to Python's ease of use and simple syntax.

Qt also proved an excellent solution for programming a GUI. It's thorough class documentation and solid code ensured a reliable equation editor.

CHAPTER X.

RECOMMENDATIONS FOR FUTURE WORK

Future work could include implementing improvements to the editor. Improvements to the GUI could be made. The cursor is the most significant area needing improvement. It is actually in the editor, and functions, but does not display as it should. Qt draws the other objects on top of the cursor, and covers it. Because of this it is sometimes not apparent where the next character is going to be inserted. Another possible improvement would be the ability to display, build, and edit more than one equation at a time. Also, a "new" button in the File pull-down menu would solve the need to restart the editor to start a new equation. Finally, many current programs carry "Undo" button functionality. An undo button would ease the process of fixing mistakes in an equation.

There are only small improvements to be made how Equator displays equations. There are slight vertical alignment problems between English and Greek characters. Also, the spacing is too tight around operators such as plus, minus, and equals, giving a crowded look. Finally, brackets do not stretch to contain their contents. The stretching functionality is there for a line, but should also be there for parentheses and other brackets.

If this stand-alone editor is to work with flow solver software, the programming of an ASCII translator or equation translator would be necessary. Equations would need

to be exported from the editor. In other words, from a graphical picture of an equation to an equation that has mathematical meaning in computer code. A translator to an existing language such as T_EX or MathML could be implemented, or to an in-house format.

Finally, the most involved work that could be done with Equator is linking it to CFD software to define auxiliary equations to the Navier-Stokes equations to accommodate different flow problems. This doctoral level work would complete its purpose.

REFERENCES CITED

[1] *T_EX Frequently Asked Questions*. 15 Jan. 2005 <http://www.tex.ac.uk/cgi-bin/texfaq2html?introduction=yes>

[2] Clare M. So, "Organizing the Structure of Mathematical Expressions." CS490y Undergraduate Thesis Presentation, April 1, 2003.

[3] *Swift- An Equation Editor in Java*. 15 Jan. 2005 <www.geocities.com/SiliconValley/Heights/5445/swift.html>

[4] Design Science, Inc., *MathType User Manual*. Design Science, Inc., 1992.

[5] *Formulator: Mathematical Equation Editor*. 15 Jan. 2005 <http://www.hermitech.ic.zt.ua/projects/formulator/>

[6] *Mathcast Home*. 15 Jan. 2005 <http://mathcast.sourceforge.net/home.html>

[7] *The KOffice Project – Kformula*. 15 Jan. 2005 <www.koffice.org/kformula>

[8] Icarus, *Object Oriented Programming with Python (part 1)*. Developer Shed, 2000.

[9] Anthony Sintes, *Teach Yourself Object Oriented Programming in 21 Days*. Sams Publishing, 2001.

[10] Mark Lutz and David Ascher, *Learning Python*. O'Reilly, 1999.

[11] *Motif*. 15 Jan. 2005 <http://www.opengroup.org/motif/>

[12] *Welcome to the MSDN Library*. 15 Jan. 2005 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/mfchm.asp>

[13] *Meme Code – Lgi*. 15 Jan. 2005 <www.memecode.com/lgi.php>

[14] *Tcl Sourceforge Project*. 15 Jan. 2005 <http://tcl.sourceforge.net/>

[15] *Qt 3.3: Qt's Classes*. 15 Sept. 2004 <http://doc.trolltech.com/3.3/classes.html>

[16] Matthias Kalle Dalheimer, *Programming with Qt*. O'Reilly, 2002.

[17] *Python & Java: a Side-by-Side Comparison*. 15 Jan. 2005 <www.ferg.org/projects/python_java_side-by-side.html>

[18] Peter Coad and Jill Nicola, *Object-Oriented Programming*. Yourdon Press Computing Series, Prentice Hall, 1993.

[19] Steven C. McConnell, *Code Complete*, 2nd ed. Microsoft Press, 2004.

[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1994.