A calculator program using Object Oriented Data Structures

Ben Terris (bterris@email.arizona.edu)
Phillip Toussaint (ptoussaint@yahoo.com)
Steve Varga (sdvargs@mail.arizona.edu)
David MacQuigg (macquigg@box67.com)

ECE373–Object-Oriented Software Design

Fall 2007

Instructor: **Prof. Jonathan Sprinkle**

November 9, 2007

Arizona's First University.

College of Engineering

Department of Electrical & Computer Engineering

Contents

List of Figures

List of Tables

A calculator program using Object Oriented Data Structures

Executive Summary

The goal of this project is to create an object-oriented design using both objects that were previously created in our homework assignments, and new objects that implement, extend, or at least relate to the homework objects. The goal of the course is to understand program design at a level above the details of a particular language. In support of that goal, we will implement our design in Python. While missing a few of the advanced features of Java, Python is able to express the essence of program design more concisely and clearly. We will make note of any features left out in the translations from Java to Python.

1 Project Overview

This project illustrates the power of Object-Oriented Design to assimilate pieces of programs from very diverse sources, including the Tk Toolkit, originally written in TCL, some classes from our homework assignments, originally written in Java, and some new classes written in Python. None of this would be practical if not for the encapsulation of complete, fully-functional objects that can be used without any knowledge of their internal details.

The graphical user interface (GUI) for our calculator is based on the Tk Toolkit, originally written 20 years ago by John Ousterhout at UC Berkeley, and documented in a 772 page book by Brent Welch[1] This toolkit was chosen by the developers of Python as the best compromise between the easy-learning, ease-of-use, and low-overhead expected by the Python community, and the total perfection in the appearance of every widget offered by some of the more heavy-weight graphics packages. We can certify the simplicity. The reader may judge the perfection in appearance of our widgets.

The Tk Toolkit was built into Python by making Python objects wrapping the original Tcl functions. (Tcl is not an object-oriented language.) The Python package (Tkinter) was then wrapped in even simpler classes by John Zelle for his Introduction to Computer Science[2]. His package (graphics.py) is what we used to construct our calculator GUI.

The objects (operands) used in our calculator are complete, self-contained objects implementing or extending the Cloneable, Addable, and Polynomial classes developed in our homework assignment 3.

The Calculator itself does nothing but store and display operands. All operations, including, cloning, generating a string display, and whatever mathematical operations we need are provided

as part of the class definition for the particular operand. The string representation of a Polynomial for example, is completely different than the string representation of a matrix. When the Calculator needs a string to display in its stack of registers, it just calls the str method from the particular object.

All actions are initiated by pressing a button on the Calculator. There are two modes - a simple arithmetic mode in which numbers (both integers and floats) and operations (+ - * /) are entered in a string and appear in the main display. Pressing the = button then calls the Python eval() function, which evaluates the expression and displays a numerical result.

A second mode, initiated by pressing the S button, pops up a display of the contents of a 5-register stack. Registers show the string representation of an operand, and an operand can be any of (Integer, Float, Complex, Polynomial, Matrix, or others), though not all of these will be implemented. Keyboard entry (via the Enter widget in graphics.py) is accepted in this mode, which is a good thing, because point-and-click would be rather slow to enter a big Polynomial.

The stack display has a few more buttons to provide simple stack manipulation, like Push a new operand, or Roll Down (moving the bottom operand to the top). Other operations are provided by the stack display for use by the calculator buttons. Hitting + for example, pops two operands off the bottom of the stack, and pushes the result back onto the bottom.

Operations on the more complex operands are very limited. Our purpose is a demo, not a marketable product. Clicking on a button that initiates an unsupported operation just shows the word ERROR in the display.

2 Analysis

2.1 Project Objectives

Our project is to create a functional RPN (Reverse Polish Notation) calculator in python using a data structure. This project involves extending homework 3 to allow for manipulation of data in a useful manner. Our project should have the ability to:

- Interface with a graphic user interface.
- Communication between the GUI and the Calculator functions
- Accurate algorithms of calculator functions
- A storage class for elements provided by the GUI
- Elements to be used by the calculator and its functions.

2.2 Data Dictionary

CalculatorDisplay- Graphic User Interface that user will see and interact with.
CharacterInterpreter- interfaces with the calculator display so that messages passed from the display interface are correctly interpreted by the calculator.
Calculator- an object that will manipulate numbers and polynomials of similar type off the stack and put the result onto the stack.
Stack- an object that stores elements for the calculator's use.
Elements- An object that can be used by the calculator.
AddableRealPolynomial- A polynomial element with real coefficients and exponents that the calculator can manipulate.
Polynomial- a generic polynomial element with integer coefficients and exponents.
Monomial- a generic monomial element with an integer coefficient and exponent.

2.3 Domain and Application Analisis

The analysis part of this project does not seem to add anything of value to our presentation, because a calculator is so well understood that the other sections of this report say it all. Therefore we feel comfortable designing our classes without the redundancy of this section.

2.4 Use Case Diagram

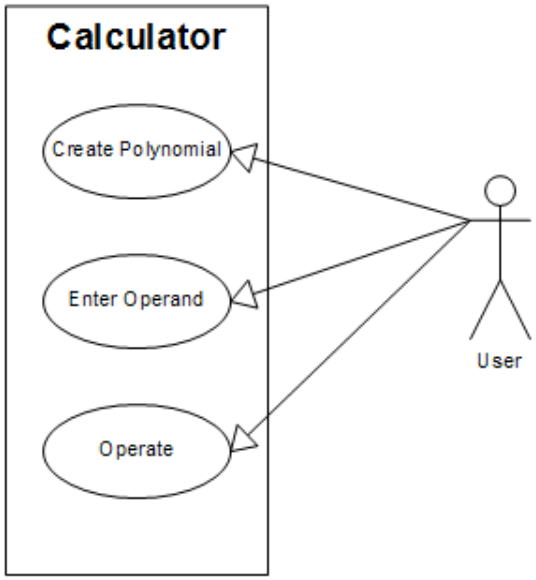

Figure 1. Use Case Diagram

2.5 Use Case Summaries

- **Enter Operand**. The user enters an operand and it is stored on the stack where it will be operated on later.
- **Create a Polynomial**. The user creates a polynomial from a sequence of numbers representing coefficients and exponents.
- **Operate**. The user performs some operation on operands previously entered into the calculator.

2.6 Use Case Descriptions

Use Case: Enter an Operand
Summary: The user enters an operand and it is stored on the stack where it will be operated on later.
Actors: User
Preconditions: The calculator is turned on and waiting for input.
Description: The calculator starts in an idle state waiting for user input. Nothing is initially displayed on the screen. A number is entered by pressing either the keyboard number pad or the on-screen GUI buttons. As buttons are pressed, their corresponding numbers are displayed on the calculator screen. When the user has entered the entire number into the calculator, they will press enter. The number is then moved to the stack and the display is cleared.
Exceptions:
Invalid Operand: If an invalid operand is entered, an error is displayed.
Postconditions: The entered operand is sitting at the top of the stack, and the display is cleared.

Use Case: Create a Polynomial
Summary: The user creates a polynomial from a sequence of numbers representing coefficients and exponents.
Actors: User
Preconditions: The calculator is turned on and waiting for input.
Description: The calculator starts in an idle state waiting for user input. Nothing is initially displayed on the screen. A user enters a sequence of numbers representing coefficients and exponents of a monomial. The sequence entered is coefficient then exponent. Each number is entered individually, followed by the enter key. The numbers are pushed onto the stack. Pressing the Poly key then pops these two numbers, creates a RealPolynomial with one term, and pushes that object back onto the stack. To create a multi-term polynomial, push the Poly key again. If the top two items on the stack are RealPolynomials, they will be added. Repeated Poly operations can thus build a polynomial of any size. **Exceptions:**
Not Enough Operands: If the operator is entered without sufficient operands on the stack, an error is displayed.
Invalid Operand Type: The two operands popped from the stack must be either simple numbers

or polynomials. If both are simple numbers (int and float as described above) the result is a RealPolynomial with one term. If both are polynomials, the result is a RealPolynomial which is the sum of the two operands. Any other combinations of operand types will leave the stack in its original condition, and show the word ERROR in the display.
*Invalid Number:*If an invalid number is entered as a coefficient or exponent, an error is displayed.
Postconditions:A new Polynomial object created from the specified coefficients and exponents is created and sitting on the top of the stack waiting to be operated on.

Use Case: Operate
Summary: The user performs some operation on operands previously entered into the calculator.
Actors: User
Preconditions: Operands have been entered into the calculator and are waiting on the stack.
Description: A user presses an operation key on the calculator GUI. If an operand is currently being edited, it is pushed onto the stack. Two operands are then popped from the stack and are manipulated using the chosen operation. The result of the operation is displayed in the GUI and pushed onto the stack.
Exceptions:
Not Enough Operands: If the operator is entered without sufficient operands on the stack, an error is displayed.
Invalid Operation: If the chosen operation is not valid for a given operand, an error is displayed.
Postconditions: The result of the operation is displayed on the calculator screen and is stored at the top of the stack.

2.7 Sequence Diagram

This shows the sequence diagram for the two types of simple operations that a user can perform. Since any complex operations are combinations of simple ones in this project, This can be extended for most uses.

2.8 State Modeling

This project does not lend itself to state modeling, as all operations should happen instantaneously, and return the system to an idle state after their execution. To avoid complication, all objects have been designed to return to an idle state when they finish. The state diagram would therefore consist of one state, idle, with many lines coming out of and going back to that one state. We could do separate states for one item in stack, 2 items in stack, ..., 1000 items in stack, but that would not help. Such a diagram could only serve to increase the confusion of the reader, and in this particular case that is not necessary.

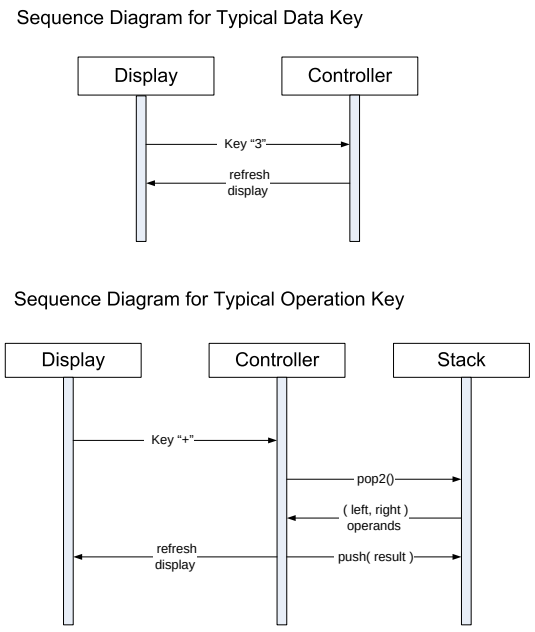

Figure 2. Sequence Diagram

3 Class Design

Our design is divided into 3 primary parts: the underlying number classes that contain the math functions, the classes which implement the calculator, and the classes which implement the GUI. The reason that we do this is that it makes our task theoretically simple if we wish to replace the GUI with a console or dedicated hardware, or if we wish to change the types of numbers that can be manipulated by the system.

3.1 Math Data Structure

The math data structure is a direct extension of homework 3, in which each type has the various important operations, in this case Add, Negate, Multiply, Invert, and Compare. It consists of the Element abstract class, and all classes that are subclasses of element. The requirement that all objects created in the math data structure be of type Element means that all of those operations must be implemented for each subtype. This creates difficulties with the invert function of polynomials, which does not work in most cases. The group decided that it would be simpler to have all operations implemented for all types, but to return an error (actually a null object)

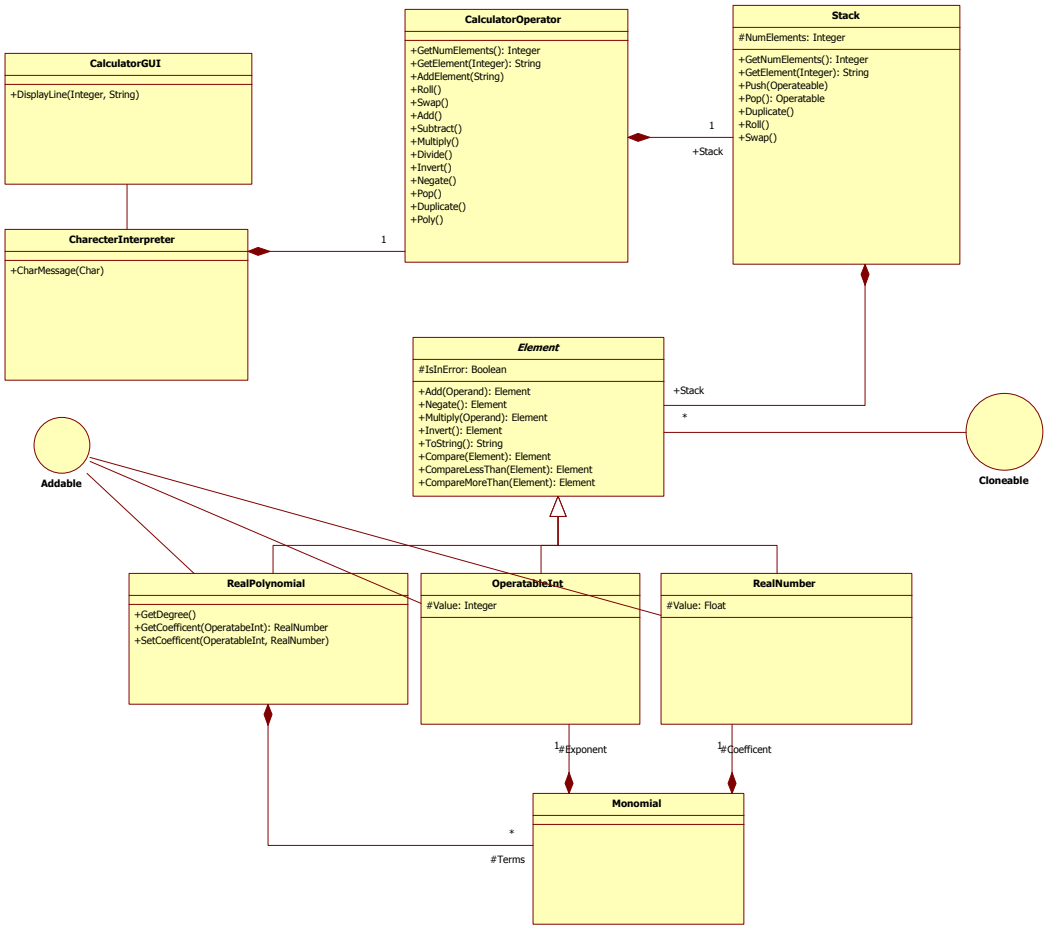

Figure 3. The Class Diagram for this project

if any invalid operation is attempted. This means that availability of operations need only be understood by the relevant type, and not by any higher level classes.

Our decision to have all operation code inherent to the number classes in this part means that other types, such as matrices, booleans, or complex numbers, can be added without changing any of the other code, or with only minimal changes.

3.2 Calculator

The calculator itself consists of two objects, the stack and the calculator object. The stack stores the Elements, and the calculator object has functions which cause it to perform operations to the top numbers on the stack, or to add numbers to the stack, or similar things. This implements a simple RPN calculator, which can be called with simple commands like “Add”.

3.3 GUI

The graphical user interface consists of 2 classes. The first is a generic GUI that we can create from existing code. This uses a second class which converts the characters sent by the GUI into commands for the calculator, and updates the display on the GUI. The reason for separating this into 2 classes is that it minimizes the amount of modification needed to existing code, allows for multiple programmers to more easily work on the separate parts, and allows for simple changes to the graphical user interface.

4 User Interface

The user interface is a GUI built with the Tk Toolkit, emulating a typical four-function calculator, but with an added stack display and the ability to display operands other than simple integers and floats. The Tk Toolkit was chosen over other graphics packages mainly because it is the standard graphics package distributed with Python, the language chosen for this project, and it is well-documented in that context. It is also less difficult to learn and use than the Qt Toolkit, and more compact and efficient than Java Swing.

Tk has been around for a very long time, and it is considered ugly by many who prefer the perfectly sculpted contours of more modern packages, but it is quite adequate for our needs - basic widgets such as buttons and data entry boxes.

We decided to minimize the amount of "circuitry" in the display module, and make it basically a passive display, with the ability to capture mouse clicks and send a code indicating which button was pressed. There are two categories of button - data and operations. Typical user interactions are discussed in the analysis section.

When the user presses a data key, all that happens is that the display (bottom register) is refreshed with the new key appended to any existing data. When an operation key is pressed, the Controller pops the necessary operands off the stack, performs the operation, and refreshes all registers in the display.

We also considered an architecture in which the Display module maintained the stack, and made the Controller much simpler. We decided to go with the current design, however, so it would be easier to change the Display if we later add other data types and operations.

5 Implementation Plan

5.1 Task Breakdown

David MacQuigg will construct the GUI and the Polynomial plus monomial. Ben Terris will implement the stack. Phillip Toussaint will implement the CharacterInterpreter, Element, OperateableInt, and RealNumber. Steve Varga will implement the CalculatorOperator.

Each project member will be responsible for writing a comprehensive set of unit tests prior to writing his section of the project code. After each component is developed to completion and thoroughly tested, the project will be assembled. Ben Terris will be responsible for testing to make sure the final product is in working order.

5.2 Code Management

For this project, we are taking advantage of Google's free project hosting service. Along with a hefty 100 MB of storage, this service provides access to a useful set of collaboration tools including a wiki, a bug tracking system, and a subversion repository. The main tool of interest is the subversion repository, which allows all group members to simultaneously work on any particular part of the project without fear of overwriting or duplicating somebody else's work.

6 Abbreviations

GUI: Graphical User Interface
RPN: Reverse Polish Notation
TCL: Tool Control Language

References

[1] B. B. Welch. *Practical Programming in Tcl and Tk, 3rd ed.* 2000.

[2] J. Zelle. *Python Programming: An Introduction to Computer Science.* 2004.