Group Without a Name

Arithmetic Expression Evaluator
Software Architecture Document

Version 2

Revision History

Date	Version	Description	Author
11/8/2023	1.0	Adding introduction	Vidur Pandiripally
11/11/2023	1.0.1	Finished base iteration of document	Jake Bernard
11/30/2023	2	Updated document for final release	Jake Bernard

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of the Software Architecture Document is to provide an overview of the high-level architectural designs used in the implementation of the software. The major abstractions and modules that the software is divided into are presented at different levels of granularity and from different views to give a thorough description of the software's components, their interactions, the flow of operation, and interactions with external entities and interfaces. Explanations will be given within to elucidate the reasoning behind the major design decisions.

Important note in version 2 of this document:

Diagrams and sections have been updated to reflect the architecture of the current release. There will be two diagrams and descriptions in each section: the "**Planned**" architecture, which reflects the original plan, and the "**Implementation**" architecture, which reflects the implementation in the current stable version. If nothing has changed between the plan and implementation, it will be noted in the implementation section briefly.

1.2 Scope

The Software Architecture Document applies directly to the implementation of the program, its interfaces, and its methods of externally storing data. The organization and design decisions made within should reflect and guide the implementation process.

1.3 Definitions, Acronyms, and Abbreviations

For anything not covered below, see the *Project Glossary*.

1.3.1 Lexing/lexer and tokenization/tokenizer

Both refer to "lexical tokenization": the process of converting text to some kind of input into meaningful/useful "tokens". In the case of arithmetic expressions, this would be the numerical values, operators, and parentheses. This process is referred to either as "lexing" or "tokenization" and is carried out by a "lexer" or "tokenizer", respectively.

1.3.2 Package, Class, Object, Data Type, Data Store (Section 5)

Section 5 uses the terms **Package**, **Class**, **Object**, **Data Type**, and **Data Store** to classify the various components of the decomposed program. Used in this context, these classifications have very specific meanings related to their expected behavior. These are definitions created specifically for this document and may not reflect outside standards.

Package

A group of functionality and/or data that the program can be decomposed into. May correspond to one, many, or possibly no classes in the implementation.

Class

A grouping of functionality (and possibly data) which most likely corresponds to a class in the actual implementation of the program. Their duties are primarily functional and any data inside should be directly related to their functionality.

Object

A grouping of data (and possibly functionality) which exist mainly to be used by functional parts of the program. Ideally, they should be almost entirely passive and any internal methods exist to present their data or make functional use of the data they embody (such as OpNodes). The potential multiplicity of objects within the program is identified using the UML format for specifying multiplicity in brackets (eg, [*], [n..m], [n]).

Data Type

A totally passive custom form of data used by the program. No internal methods are associated with it. Can be an alias for a pre-defined type.

Data Store

A totally passive store of data which is written to/read from by another module. May be external to the runtime program.

1.3.3 Reverse Polish Notation

A way of writing mathematical expressions where operands precede their operators. Used when algorithmically evaluating expressions because it requires no parentheses and can be easily evaluated using a stack and a queue data structure.

1.4 References

The *Software Architecture Document* references these documents:

- *Project Glossary,* artifacts/project_glossary.pdf, Group Without a Name, 2023
- *Project Requirements,* artifacts/project_requirements.odt, Group Without a Name, 2023
- *Project Description,* artifacts/project_description.pdf, University of Kansas, 2023

1.5 Overview

The *Software Architecture Document* contains the following sections:

- **Architectural Representation**

  Describes the high-level software architecture, presents several views of it, and explains details necessary to understanding the architecture and its views

- **Architectural Goals and Constraints**

  Describes the constraints, requirements, and objectives of the software that inform and motivate the architecture's design.

- **Use-Case View**

  N/A

- **Logical View**

  Decomposes the archtectural representation into sub-systems, packages, and classes, which are presented and described, with important attributes, functions, and relations explained.

- **Interface Description**

  Describes the program's interfaces in detail, including the types of input and resultant outputs.

- **Size and Performance**

  N/A

- **Quality**

  Describes how the software architecture affects the non-functional attributes of the system.

2. Architectural Representation

Planned:

The software architecture for the Arithmetic Expression Evaluator is a component-based architecture which separates input/output and display operations (**User Interface**), runtime data storage and state information (**User Context**), and major program logic/operations (**Program Logic**).

These three elements hold references to each other and pass messages through these internal references, but the ownership of each is placed withing a single control module (**Control Module**), which also exists to modify the **User Interface** at runtime.

The **User Interface** and **Program Logic** both can access and modify the same runtime data via the **User Context**, allowing more sophisticated message passing and state updates to be stored and accessed by each, greatly simplifying the interface needed between the two.

Data stored by the program to be kept outside of runtime exists within the **Config File**. The main data stored here relates to the options the program last was run with, eg the interface style.

The architecture of the Arithmetic Expression Evaluator bears a superficial resemblance to the Model-View-Controller pattern, where separate modules exist to hold state information and program logic (*Model*), to present output and handle input (*View*), and to create, modify, and pass information between the View and Model (*Controller*).

It is easy to classify the **User Context** and **Program Logic** as the *Model*, but the **Control Module** does not mediate the sharing of data between the **User Interface** and the other components. If one were to further compare the architectures, the *View* would correspond to the **User Interface**, and the *Controller* would correspond to the **User Interface** and the **Control Module**.

If there is sufficient time to refactor existing code, an increased mediatory role may be assigned to the **Control Module** so the **User Interface** no longer directly interfaces with the **Program Logic** and **User Context**. This implementation will likely follow the observer pattern and this document will be updated to accomodate the changes. See also: *Section 3, "Existing Code & Deadlines"*.

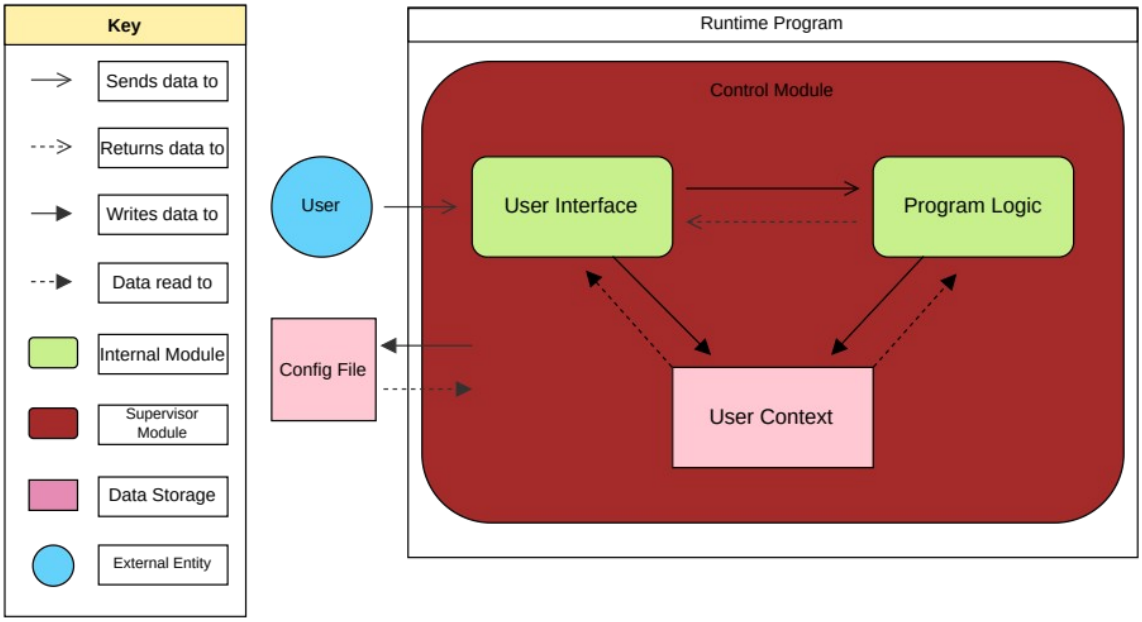
The first two high-level views presented below contain the same components, but different types of relationships between each are highlighted in each view. The third high-level view decomposes the program into packages and contains more information about some modules than the other two, but is largely unconcerned with the relations between packages. These packages and sub-packages will be explored in depth in the *Logical View* (Section 5).

Implementation:

The high-level architecture is largely unchanged from the initial plan, although currently the **Config File** serves no purpose so its functionality has been commented out and the **Program Logic** never writes to or reads from the **User Context**.

2.1 The Runtime-Ownership View

Planned:



This view emphasizes the ownership relations between the different aspects of the program and presents a simplified view of how each component exists at runtime, placing specific emphasis on:

- The **Control Module's** ownership of the **User Interface**, **User Context**, and **Program Logic**
- The modular composition of the program at runtime
- The externality of the config file

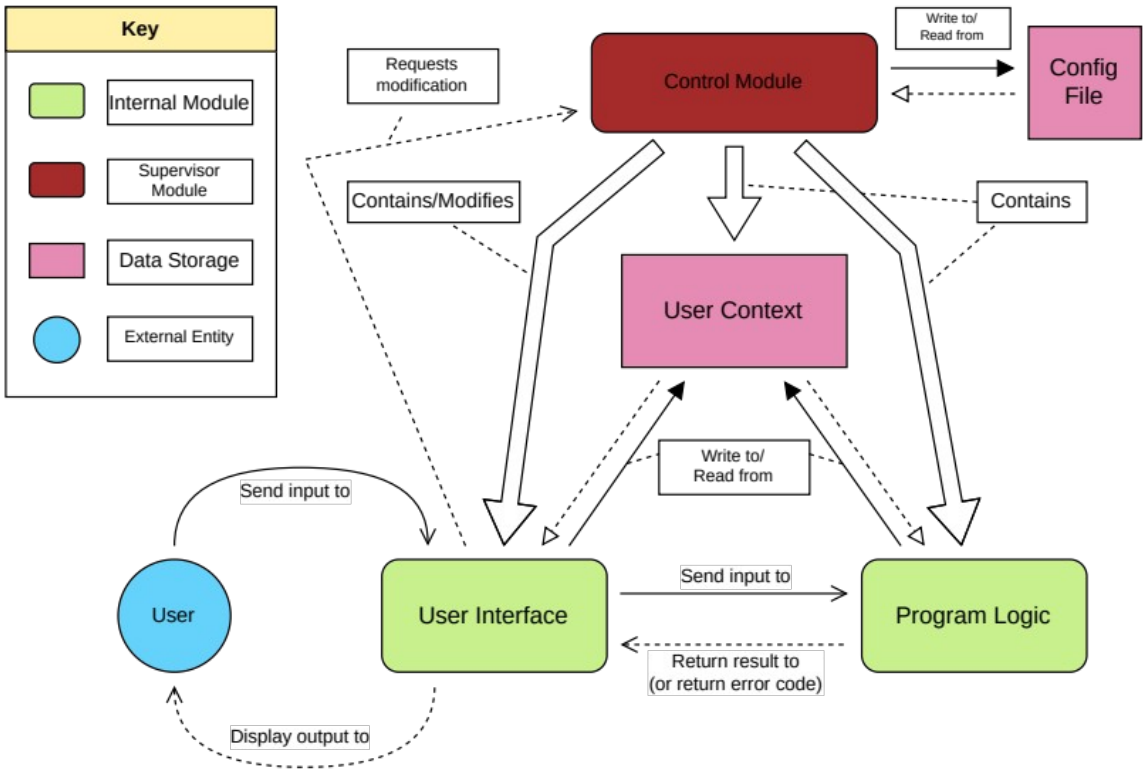
Simplified relations between modules are indicated with different arrow types. "Sending/returning data" and "writing/reading data" are kept distinct to highlight the difference between actions that do not directly modify/access persistent internal storage within a module (eg evaluation of an expression) and those which directly modify/access persistent internal storage (eg binding a value to a user-defined variable). Note that writing to the **User Context** can also include modifying/deleting data.

Internal modules are distinguished from supervisor modules (which own them) by color, and data storage is distinguished from functional components by color and shape (hard vs rounded edges). External entities expected to interact with the system are given a circular shape and distinct color to distinguish them from the actual components of the architecture.

Implementation:

As stated previously, this is largely unchanged (outside of the unused **Config File** and lack of communication between the **Program Logic** and **User Context**.

2.2 The Component-Relation View



This view more verbosely informs how each component interacts with the others, with less emphasis on where each is contained within the program.

Specific notice should be given to:

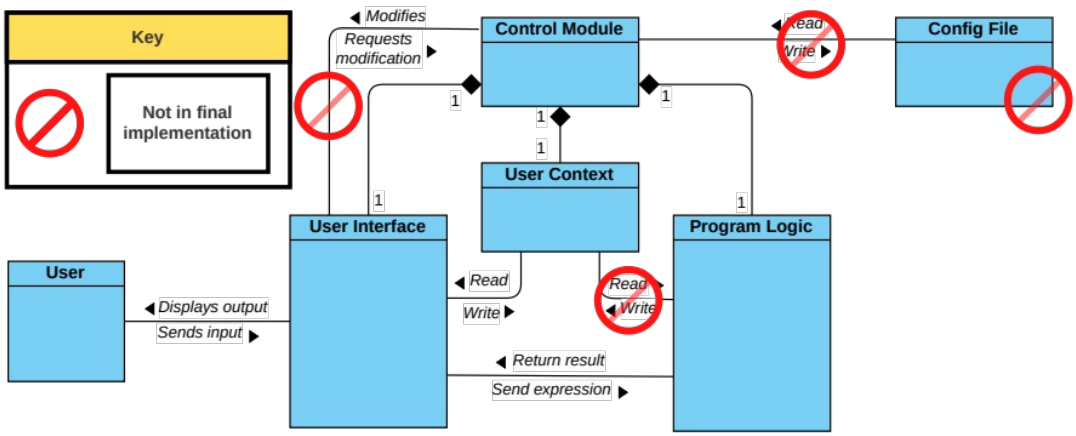
- The **User Interface's** need to request modification from the **Control Module**
- The input/output relationship between the **User Interface** and **Program Logic**, with the **Program Logic** being able to return an error code.
- The **User Interface's** responsibility to both take input and display outputs.

The same notation is used to distinguish between different modules, data storage, and external entities. The choice of arrows for each relation is more arbitrary, but each follows a somewhat similar pattern of a dotted line meaning returning/reading and a solid line meaning sending/writing, and a filled/white arrowhead corresponding with writing and reading and an open arrowhead corresponding to sending/returning data. Large block arrows convey ownership. This pattern is not shown in the key because every interaction is labelled to clarify how each module interacts with the others.

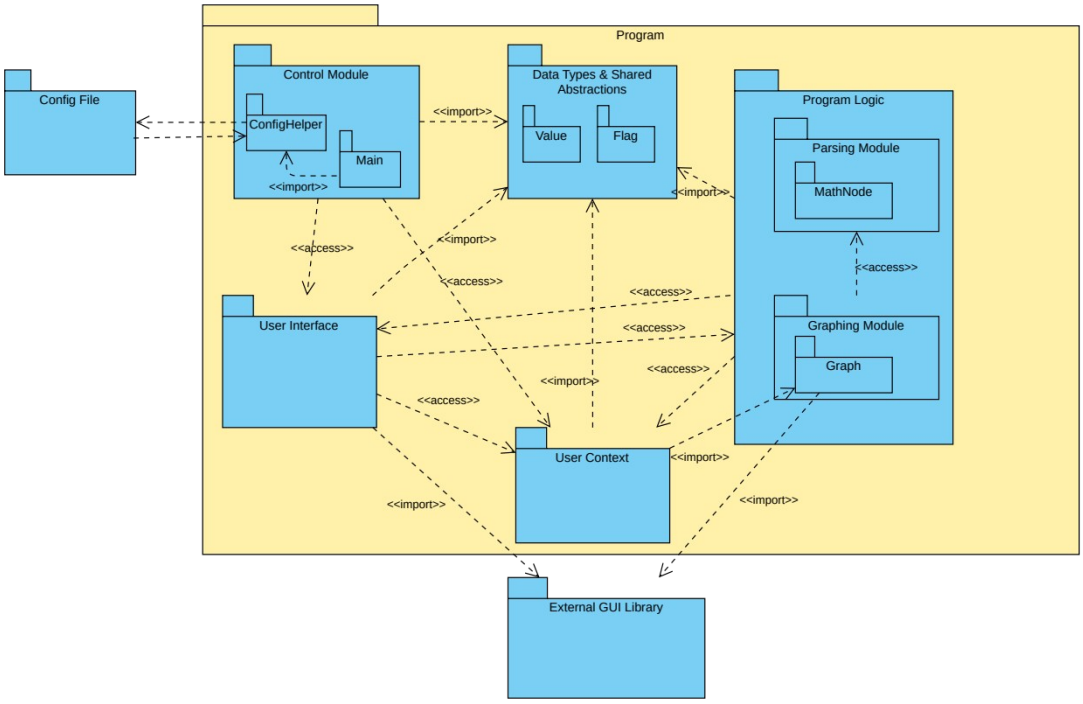
The same distinction is made here as in the other diagram between sending/returning data and writing/reading data based upon persistence. The externality is of the config file is deemphasized here, while its relationship to the **Control Module** is emphasized.

Implementation:

As stated, this is largely unchanged. A more up-to-date and possibly easier to read semantically-equivalent UML diagram is provided, which also highlights which areas are not included in the final implementation.



2.3 The Package View



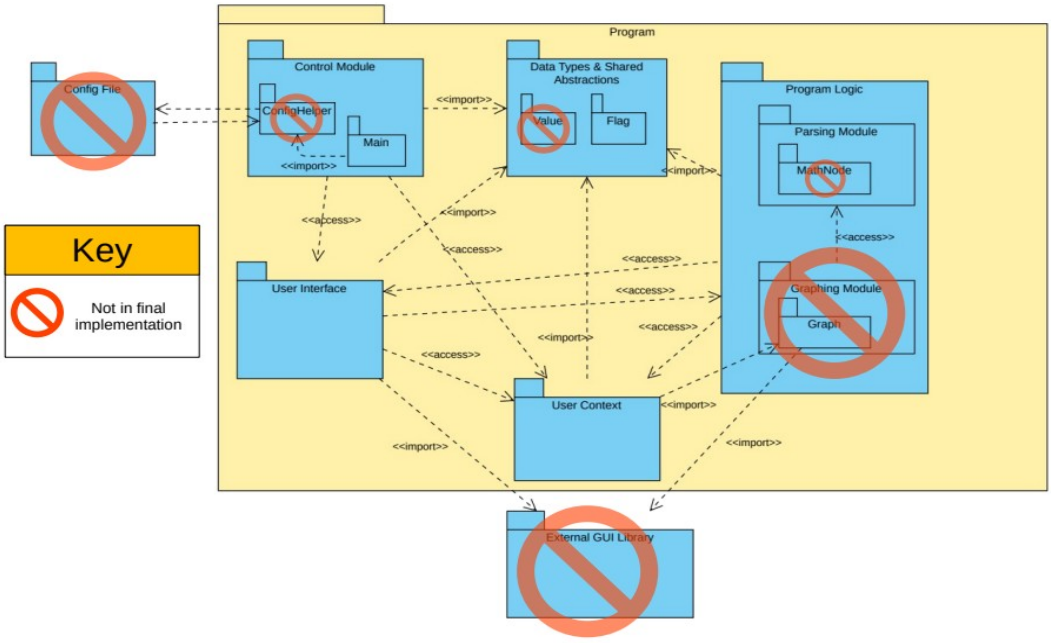
This view presents a higher-granularity look at the packages/modules within the program and includes several packages not seen in the other two views, including some internal sub-packages such as the **Parsing Module** and **ConfigHelper**, while also referring to outside packages used within the program, such as the **External GUI Library** (most likely GTK).

The module that does not appear in the other two architectural views, **Data Types & Shared Abstractions** is purposefully left out of the other two due to its relation to internal representations of data rather than functionality. It also is universally imported throughout the program, so its inclusion in the other two would have only served to obscure the main functionality and flow of the program.

These packages and sub-packages will be explained in greather depth in Section 5.

Implementation:

The modified package diagram below shows which packages are not included or used in the final implementation.



3. Architectural Goals and Constraints

The following qualities, objectives, and constraints impacted the design of the software architecture:

Object-Oriented Programming

The constraint that the project must apply principles of object-oriented programming (see *Project Description*) informed the development of a modular, component-based architecture. Each component is representative of a class (or multiple classes) used during the implementation of the system.

Team Structure & Unit Testing

The team structure of this project necessitated the design to have multiple, (nearly) functionally isolated units which could be worked on in parallel by different members of the team, with interfaces between each being well-defined. This also allows each module to be tested separately and for all of them to be gradually linked together.

User Interface

The requirement of a user interface (see *Project Description*) lead to the separation of the input/output (presentation) of data from the internal logic of the program and the runtime data.

Multiple User Interfaces

The desired functionality of having multiple interfaces (see *Project Requirements*) influenced the architecture to use polymorphism to dynamically change the type of interface used at runtime. This is also necessitated the development of the **Control Module** to decouple the lifetime of the user interface from the lifetime of the program.

Shared Runtime Storage

Desired requirements such as custom user variables and keeping a history of previously evaluated expressions (see *Project Requirements*) created a need for the user interface and program logic to be able to access a shared program memory (such as for assigning a value to a variable and later retrieving the value assigned to that variable). This is the need the U**ser Context** exists to satisfy.

Changeability & Opaque Interfaces

By separating functionality, presentation, and program data, each module may be expanded with only minimal effort from the other modules to accomodate. Providing interfaces which do not reveal much about the internals of each module also allows for a better separation of concerns.

Extendability

Many features were marked as desirable or optional in the *Project Requirements*, and for those to be added gracefully as the project develops, care needed to be taken to make sure the elements of the architecture could safely be separated from each other and have a high degree of functional independence. Although not currently ideal, any changes done to the architecture in the future will emphasize this value.

Existing Code & Deadlines

As of writing, work has already been done on the codebase and while some architectural choices are sub-optimal, such as the multiple roles the **User Interface** has to perform in passing user data to the **Program Logic**, they are currently implemented and must therefore have an impact on the overall architecture in order to ensure all work is completed on time. If sufficient time exists to refactor some of these deficiencies, they will be implemented and this document will be changed to reflect those changes.

Implementation: Deadlines & Programmer Knowledge

Certain details of the implementation had to be cut due to a lack of time (GUI, Value types, and graphing capabilities) while others were cut due to team members' inexperience with the C++ language and conventions making proper implementation too difficult or time-consuming.

4. Use-Case View

N/A

4.1 Use-Case Realizations

N/A

5. Logical View

5.1 Overview

See the Glossary for explanations on the classifications of the pieces of the logical view.

Modules which didn't make it into the final implementation are marked as **[not implemented]**

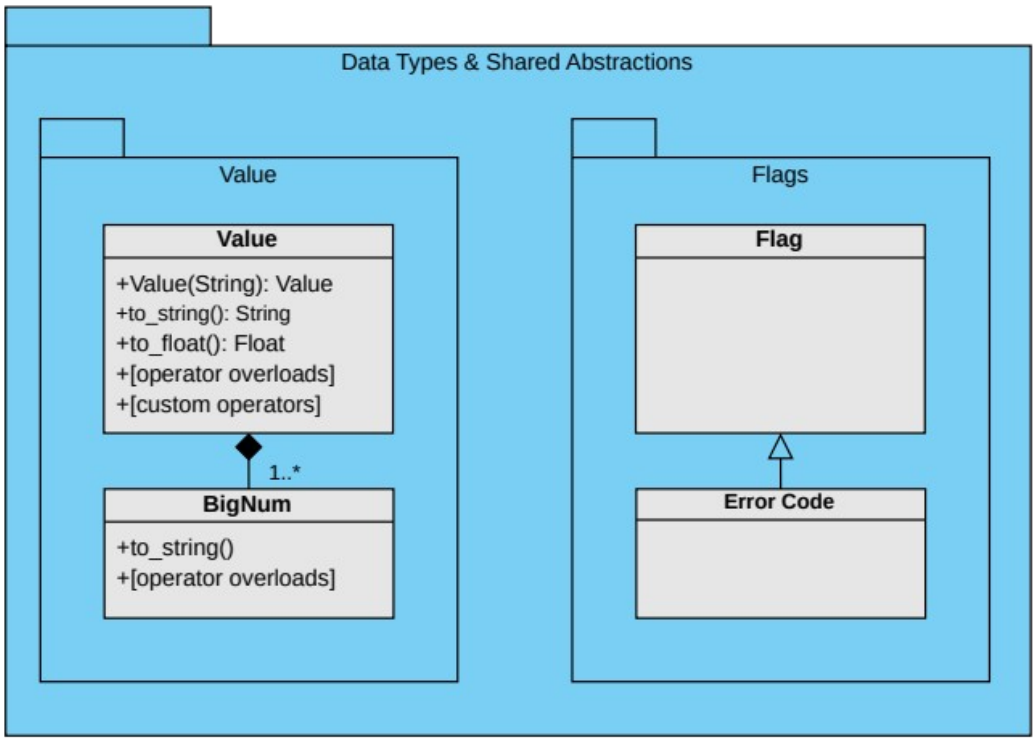
Any difference between described functionality and implementation functionality will be expanded upon within *Implementation* sections.

The design model is decomposed into the following packages, sub-packages, classes, data types, and data stores in the following section (Note that indentation implies group membership).

- **Data Types and Shared Abstractions [Package]**
  - **Value [Object [*]] [not implemented]**
  - **Flags and Error Codes [Data Type]**
- **Control Module [Package]**
  - **Main [Class]**
  - **ConfigHelper [Class] [not implemented]**
- **User Context [Package/Object [1]]**
- **User Interface [Package]**
  - **AppInterface [Abstract Class]**
    - **TUI [Class]**
    - **GUI [Class] [not implemented]**
- **Program Logic [Package]**
  - **Parsing Module [Sub-package]**
    - **Lexer [Class]**
    - **Parser [Class]**
    - **Evaluator [Class]**
    - **MathNode [Abstract Object [*]] [not implemented]**
      - **NumNode [Object [*]] [not implemented]**
      - **OpNode [Object [*]] [not implemented]**
    - **NodeFactory [Class] [not implemented]**
  - **Graphing Module [Sub-package] [not implemented]**
    - **GraphMaker [Class],[not implemented]**
    - **Graph [Object [*]] [not implemented]**
- **Config File [Package/Data Store] [not implemented]**

5.2 Architecturally Significant Design Modules or Package

5.2.1 Data Types and Shared Abstractions



The **Data Types & Shared Abstractions** comprise the common data types/classes that will be used in multiple modules of the program.

Value [not implemented]

The value object is a type made to represent a numerical value used within the program, with its internals being intentionally opaque so that work can be done with it no matter the underlying representation. It contains a method to be created from a string and can return its internal value as a string (for the user interface) or a float (for the graphing module). All expected operators should be overloaded, and any custom operators (eg, logarithms) implemented will exist within.

BigNum [not implemented]

If possible to implement within deadlines, several **BigNums** will be the internal storage type for the **Value** class. See the *Project Glossary* for a definition. It can also be created from a string and converted to a string or float.

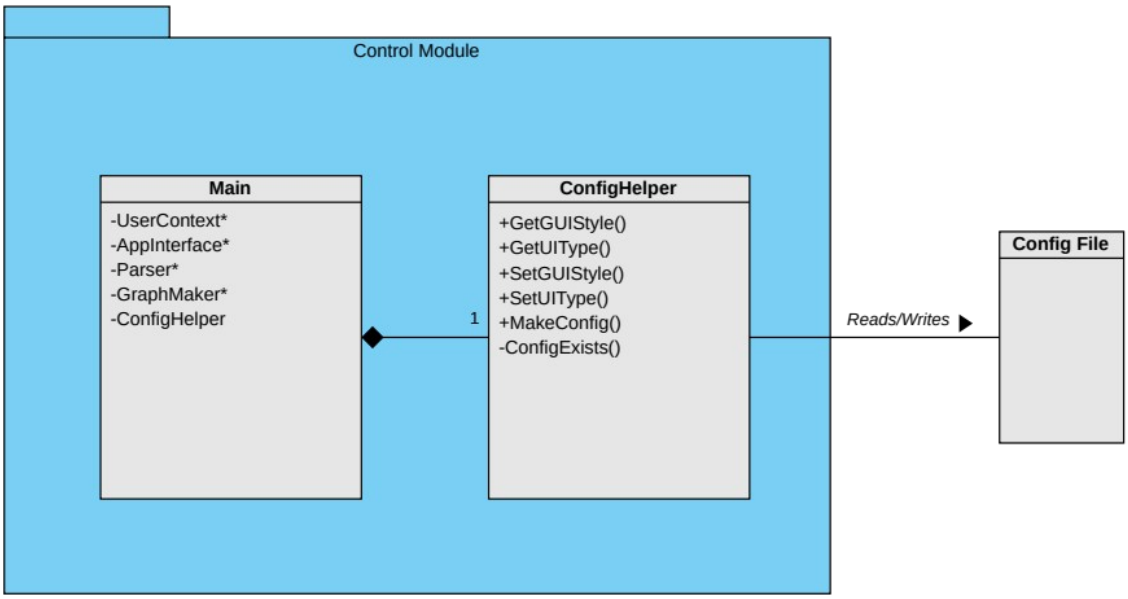
Flags [implemented] and Error Codes [not implemented]

Flags and **Error Codes** are just enums used throughout the program to pass messages between modules where booleans are not informative enough, eg when evaluation of an arithmetic expression does not work because of a division by zero. Error handling will not done through exceptions, so it is necessary to define internal error codes. In many cases where the architecture document specifies return values, that is mostly done by reference and the actual return value of many operations is a flag signalling success or an error.

Implementation

Error codes are not used by the parser and instead strings are returned directly.

5.2.2 Control Module



This module performs administrative duties on the rest of the program and owns/instantiates all the sub-modules. It also performs all reads/writes to the external config file. This module is the first one called at runtime and is responsible for starting and ending the program.

Main

This is the main file of the program. Pointers to the main components are stored here and referenced within each other. The main loop runs the **User Interface** until it returns a code to either exit the program or swap to a different interface type.

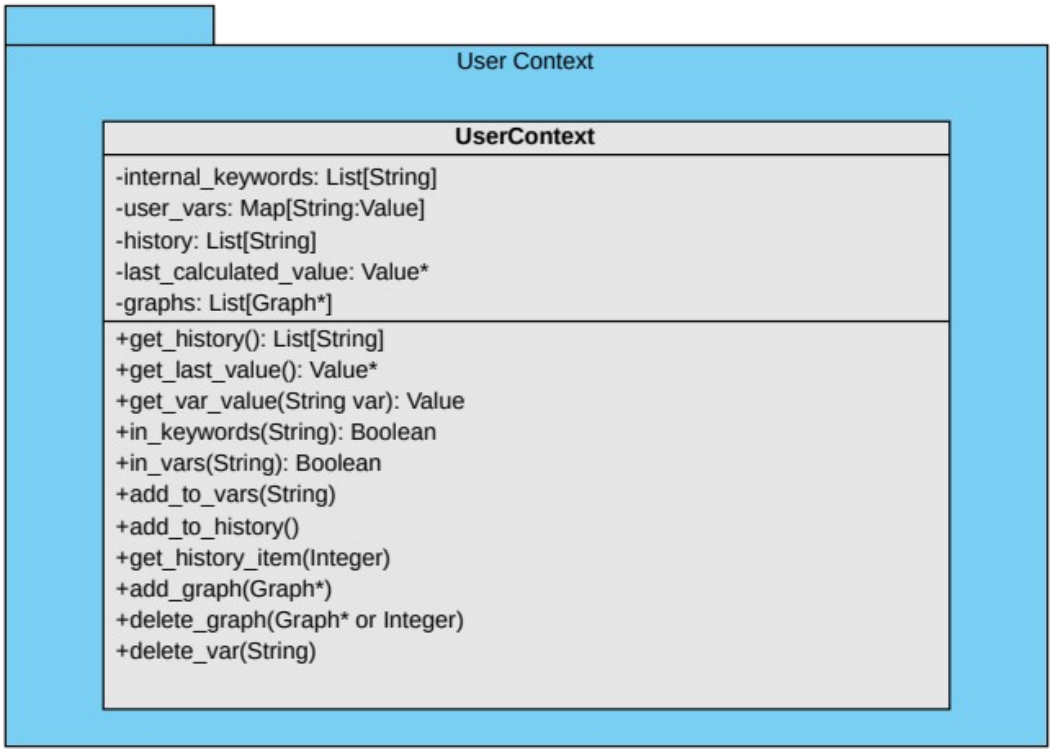
Implementation

The code to exit the program is returned but no other functionality for swapping to different interface types is present.

ConfigHelper [not implemented]

This module is only a collection of functions and is called at the beginning of the program to read what the last used settings for the interface of the program were and apply them, then on exit it is used again to write the currently used settings to the config file so they can be used again on startup.

5.2.3 User Context

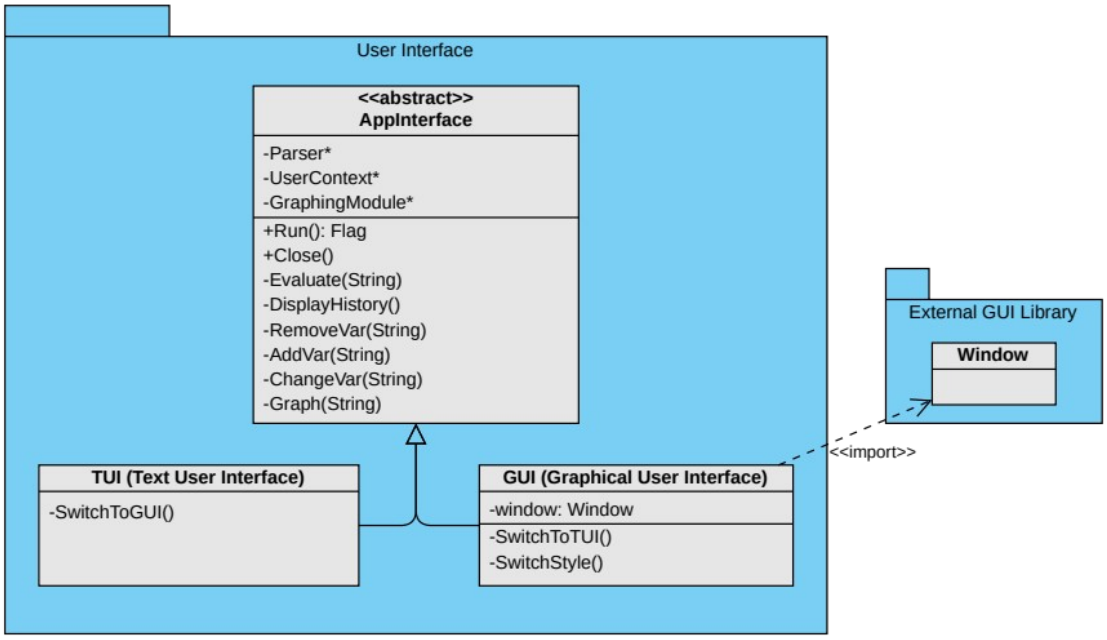


The **User Context** exists to store data which is used between modules and may require updating/reading from in each.

The main attributes it holds are:

- **internal_keywords [not implemented]:** A list of all internal keywords (to compare against suggested names for user variables in order to make sure the user does not store a required keyword as a variable name)
- **user_vars [not implemented]:** A map of all stored user variables, with each name corresponding to a **Value**
- **history:** A list of the last several successfully calculated expressions as strings (up to some set history length limit). For use in the **User Interface** when checking an old expression.
- **last_calculated_value [not implemented]:** the last successfully calculated value from an expression (or a null pointer if the last expression did not successfully evaluate). Used during binding user-variables to values.
- **graphs [not implemented]:** a pointer to every currently opened graph (graphs are separate windows)

5.2.4 User Interface



This module contains the interface presented to the user which handles all input and display and passes most work to other modules. The level of coupling with other modules is unideal at the time of writing, but this may change during implementation.

AppInterface

The abstract base class which represents a generic type of user interface. Used by the **Control Module** to achieve runtime polymorphism when the Run method is called on either a **TUI** or **GUI** instance. Contains prototypes for several functions used within the derived classes.

Run returns a flag depending upon if the program should exit or if the C**ontrol Module** should switch to a different UI.

Close frees all memory the UI may have been using.

The listed private methods rely on passing input to different modules.

The member variables are pointers to different modules so that messages can be passed directly to each.

All non-fatal errors should be displayed by the AppInterface.

Implementation

The private/protected methods RemoveVar, AddVar, ChangeVar, and Graph have been removed and no pointer to a graphing module is held.

History may still be accessed and re-evaluated in the final implementation.

TUI (Text User Interface)

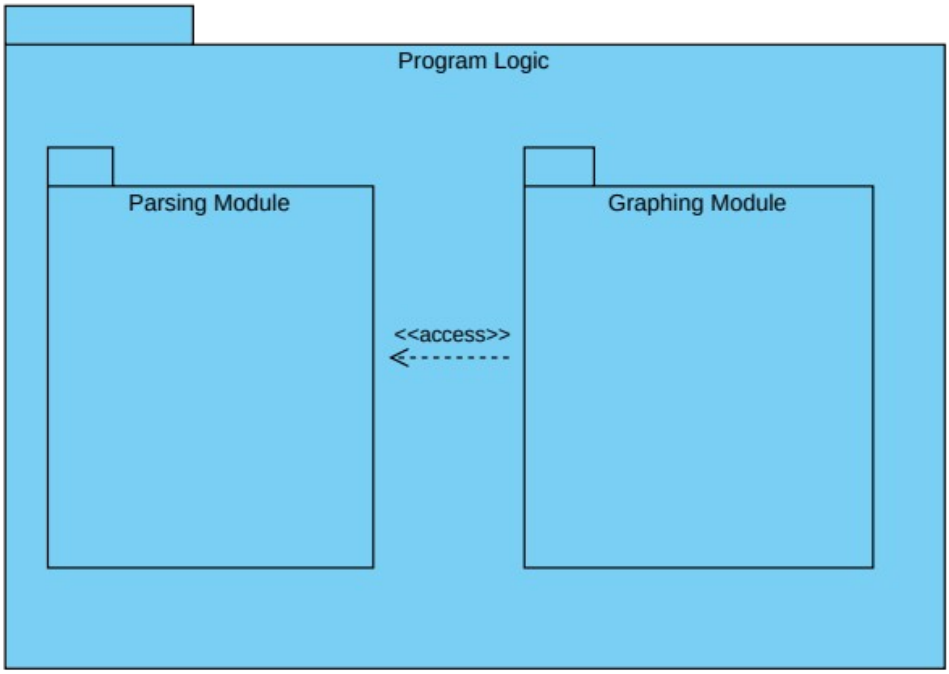
This is the basic command line interface, with several built-in keywords which provide access to the other aspects of functionality the program has.

GUI (Graphical User Interface) [not implemented]

A graphical user interface which responds to both keyboard input and clicking on onscreen buttons. Can be displayed in multiple styles.

An external GUI library's functionality will be imported for the creation of the window (and likely many widgets within the window such as buttons).

5.2.5 Program Logic

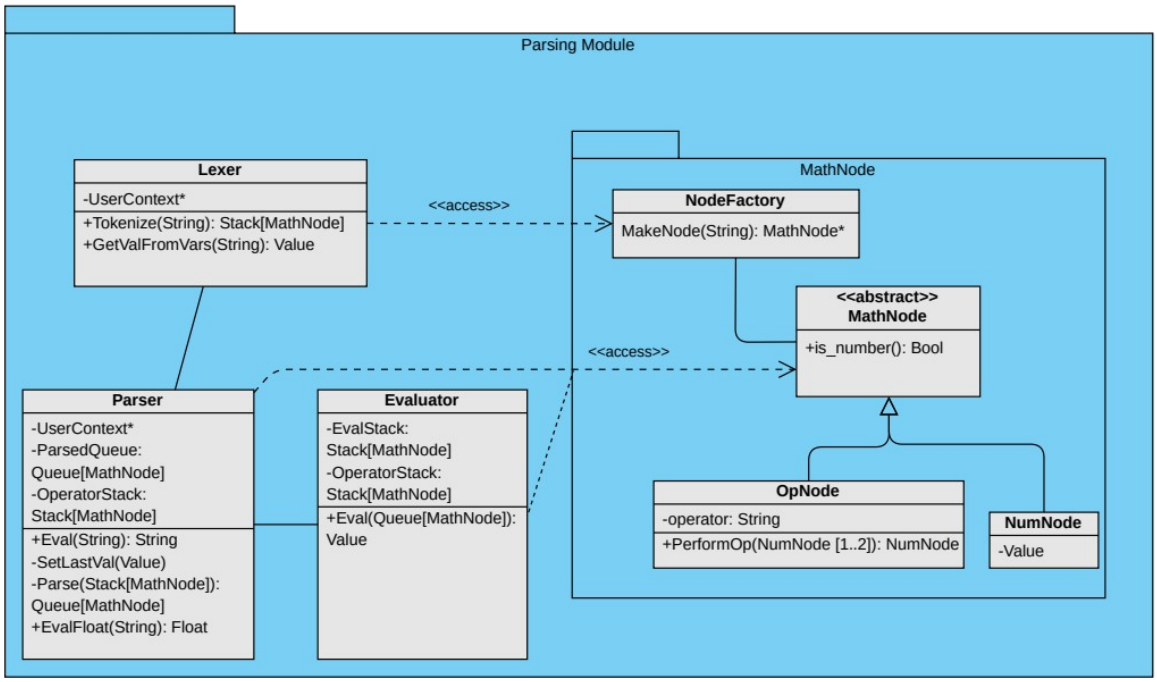


The **Program Logic** contains the modules which perform most of the important logical operations upon input data to transform it into desired outputs. It holds two sub-modules: the **Parsing Module** and the **Graphing Module**.

Implementation

The graphing module is not present in the final implementation.

5.2.5.1 Parsing Module



The **Parsing Module** is expected to take user input from the **User Interface** (or sometimes **Graphing Module**) as a string and return a string (or a float). The **Parsing Module** transforms strings of text which represent arithmetic expressions into a string (or float) representing the output of that expression.

Parser

The **Parser** is the main class involved in parsing input, and is the class designated to return the final output.

It first has the input turned into a series of **MathNodes** by the **Lexer**.

It then sorts those **MathNodes** into a queue in *Reverse Polish Notation*.

That queue is then evaluated by the **Evaluator** and the final value is returned.

The **Parser** then returns this value, either as a string or a float.

Any errors returned during the processed will be passed upwards by the **Parser** back to the **User Interface**, where they will be displayed.

Implementation

The **Parser** does not use **MathNodes** and instead works directly with C++ strings. The **Lexer** and **Evaluator** are part of the **Parser** itself and are not distinct modules, only distinct stages of the process/functions in the module. Values are returned only as strings, and errors are also returned as strings. The **User Context** is never accessed.

Lexer

The L**exer** scans the input text phrase-by-phrase (eg, numbers and operators) and sends meaningful segments to a **NodeFactory** instance to create an appropriate **MathNode** for each, which is then added to a sequence that is returned to the **Parser** to be put into *Reverse Polish Notation*.

Unfamiliar phrases are checked against the user variables stored in the **User Context**. Matching user variables will have their corresponding mapped values inserted into a **NumNode** in their place. Errors encountered here will be returned to the **Parser**.

Implementation

The input is divided into a sequence of separate strings rather than **MathNodes.**

Evaluator

The **Evaluator** takes a queue of **MathNodes** and evaluates them according to *Reverse Polish Notation* rules until a single **NumNode** is remaining or until an error is encountered. The internal **Value** of the **NumNode** is returned to the **Parser**.

The **Evaluator** makes use of the **OpNode**'s PerformOperation function when evaluating the queue it was given.

Implementation

MathNodes are not used and instead work is done on strings that either hold values or operators. Operators are evaluated through an if-else block.

MathNode [not implemented]

An abstract base class representing any meaningful unit of an arithmetic expression.

NumNode [not implemented]

A derived class of **MathNode** representing a number in an arithmetic expression. Holds an internal **Value**. Should be able to be created from a string representing the number within it.

OpNode [not implemented]

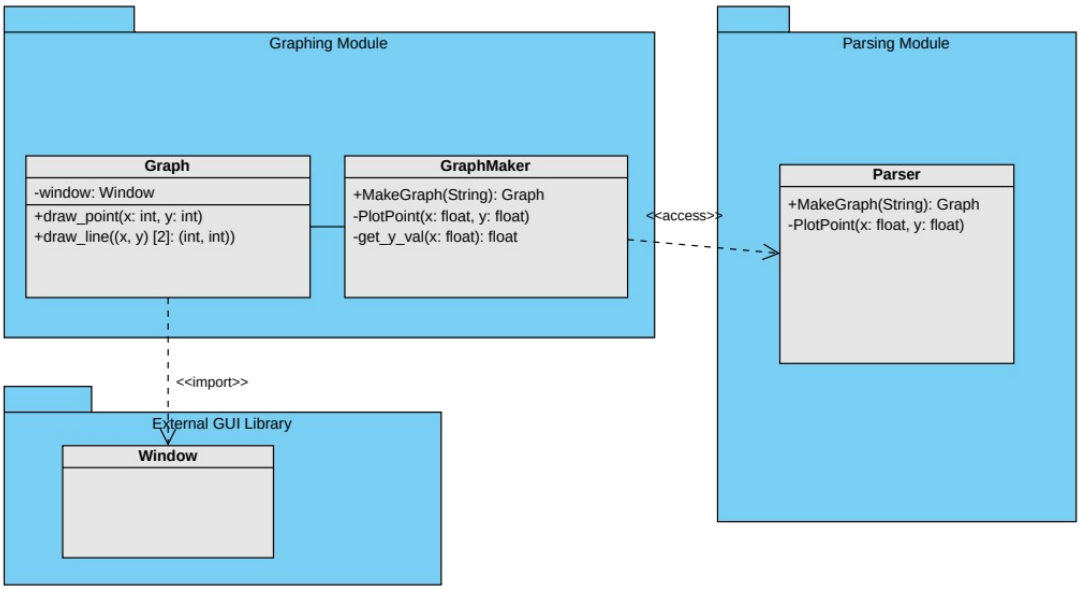
A derived class of **MathNode** representing an operator in an arithmetic expression. For ease of implementation, parentheses are considered operators, although they will throw an error if their corresponding operation is ever called.

Holds an internal method called PerformOperation which takes two **NumNodes** as input (or one if unary operations are implemented) and returns the result of performing that operation on those two **NumNode**s' internal **Values** as a new **NumNode**. Can also return an error (eg, when a division by zero is encountered).

NodeFactory [not implemented]

A class following the factory pattern made to create **MathNodes** of arbitrary type given some input string (and possibly a flag for the type).

5.2.5.2 Graphing Module [not implemented]



The **Graphing Module** is responsible for creating graphs based on user input when a specific function is called by the **User Interface**. An expression containing a special graphing variable is evaluated at multiple points for successive values substituted in place of the graphing variable.

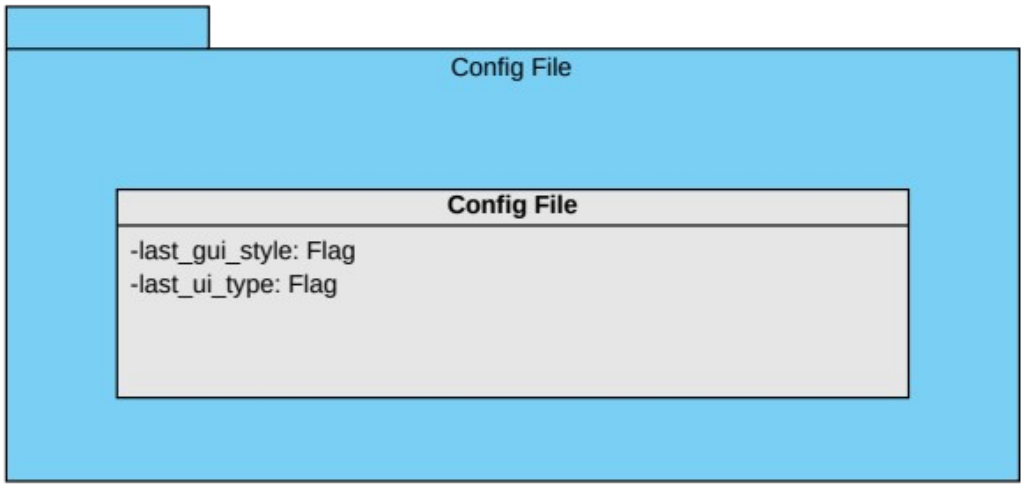
GraphMaker [not implemented]

This module calls the **Parser** to evaluate an expression for successive values of the graphing variable (representing the x-axis on the graph). These values are then plotted and connected on an internal graph that is created and placed in the **User Context's** graph list.

Graph [not implemented]

A separate window which displays a graph representing all the points evaluated for the graphing variable in a user-defined expression. Imports windowing functionality from an external library.

5.2.6 Config File [not implemented]



A plaintext file storing flags for which **User Interface** type was used last (text or graphical) and what style of **Graphical User Interface** was used last. Read and written to by the **ConfigHelper**. Created by the **ConfigHelper** on startup if it does not already exist. Is rewritten to the defaults if it contains errors when read by the **ConfigHelper**.

6. Interface Description

Command Line Interface

The command line interface will run in a terminal and accept user input as strings entered directly into the terminal. The possibility of more expressive text-based user interfaces utilizing outside libraries (such as conio.h and ncurses) may be explored if sufficient time is available, otherwise the command line implementation will be very standard and will accept text input by keyword for expression evaluation and other major operations (graphing, viewing history, saving user variables, accessing help, etc).

Outputs will be displayed as text to the terminal.

Graphs will be created as additional windows and persist across interface changes.

Implementation

The command line interface is very standard and does not use conio.h nor ncurses.

Graphical User Interface [not implemented]

A display will be provided in the graphical user interface to show both current user input and output. Input will be stored internally as strings, but will be rendered to the user differently depending upon the interface style chosen. The user should be able to input both by keyboard and by clicking buttons on the GUI.

Graphs will be created as addtional windows and persist across interface changes.

Valid Screen Formats

The program is expected to be able to run on most systems when using a command line interface.

Any system that has at least 360p resolution (640×360) is expected to be able to run the graphical user interface.

Both Linux and Windows systems will be supported by both interfaces.

7. Size and Performance

N/A

8. Quality

The software architecture contributes to the following capabilities in the following ways:

Extensibility

The separation of program state, program logic, and user interface allows the program to be more easily extended without seriously impacting the base functionality.

Reliability

Having separate modules also allows for better unit testing to be developed so that bugs may be stopped earlier, leading to a more reliable system.

Portability

Relying on certain specific outside libraries for handling windowing and/or more complicated command line interfaces may negatively impact the portability if care is not taken.

Cooperation

Having separte modules for different functionality allows for parallel cooperation upon the development of the program.

Implementation Time

Some of the architectural decisions may negatively impact the time to implement the program if they are confusing or redundant. Some of the other decisions, such the interface choices, may significantly reduce the time to implement the program because of the lack of indirection involved. So while the user interface directly communicating to certain modules may not be optimal, it may save some time during the development process.