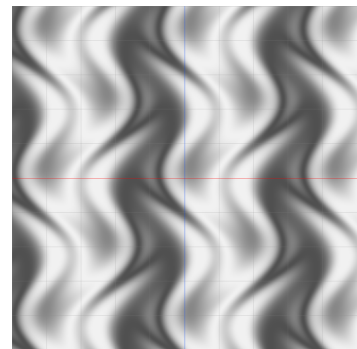
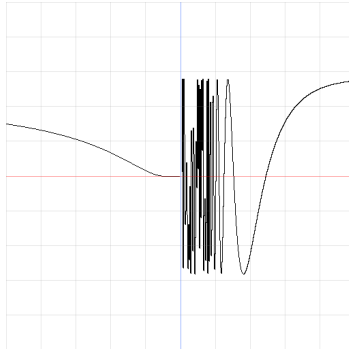
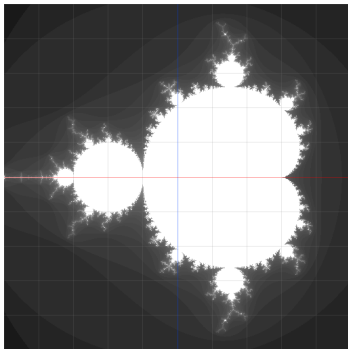


Design Document

CSC207 - 8 Musketeers

Instructions for Use	2
Specifications	2
Clean Architecture	4
Scenario walk-through	6
SOLID	6
Design Patterns	7
Reasoning Behind Major Design Decisions	8
Contributions	10
Additional Info About Our Code:	12
Uses of Github Features	12
Code Style and Documentation	12
Testing	13
Refactoring	13



Instructions for Use

Example calls (everything is case sensitive)

```
-eq "y = sin(x)" -graph BOUNDARY -save "myGraph"
```

```
-eq "mandel(x, y) = 0" -graph GRAYSCALE -save "mandel"
```

We currently have a command line interface (this contains the main method for our project). The valid commands are ["-eq", "-domain", "-save", "-load", "-graph"].

Uses:

- "-eq" <expression> is used to define the expressions that you want to graph
 - Example: `-eq "x^2 + y^2 = 1"`
 - Note that multiplication is *not* implicit, that is "2x" will be regarded as incorrect and will need to be written as "2 * x"
 - Defined functions: ["cos", "sin", "tan", "exp", "mandel", "sqrt"]
 - Explicit functions must be of the form "y = f(x)" (e.g. "y = cos(x)", "y = x^3 + 5*x - 2"). We are working on users being able to define "f(x)=..." so that they refer to f in
 - Implicit functions can be defined normally such as "x^2 + y^2 = 1" (indeed you may note that we currently using implicit functions to graph explicit functions)
- "-graph" <BOUNDARY/REGION/GRAYSCALE> is used to define how the user wants to graph things, the possible arguments are ["BOUNDARY", "REGION", "GRAYSCALE"]
 - "BOUNDARY": graphs the borders or the boundary along which an implicit function is satisfied. This is also used to graph explicit functions
 - "REGION": This graphs the regions on a plane where the left expression of an implicit function is greater than the right side. This can be useful when the detail of an implicit function is very fine. Will also be used for graphing regions later.
 - "GRAYSCALE": Graphs things in grayscale (values beyond [0, 1] are clipped to black/white respectively)
- "-save" <filename> is used to save a set of axes *containing the equations* under "filename"
- "-load" <filename> is used to load a set of axes that have been saved (note although -load will load the saved set of axes, in order to view the image -graph needs to be called in order to let the program know you would like to graph things)

Specifications

Example input: `-eq "f(x) = sin(x^3) + 1" -domain "x > 1 & x < 2" -graph BOUNDARY``

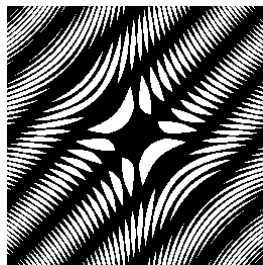
Specifications:

- Predefined functions = ["cos", "sin", "tan", "sqrt", "exp", "arctan", "arccos", "arcsin", "log", "mandel", "max", "min"]
- Use `-eq`` to graph as before
- You can use `-eq "f(x) = ..."` to define your own function named ``f`` which can be referred to later on (e.g. if you write `-eq "f(x) = cos(x)" -eq "g(x) = f(x^2)"` then ``g(x)`` will refer to ``cos(x^2)``. In such cases order matters. Defining g before f above would not work)
- Any -eq command can be followed by a -domain to restrict the points on which the graph is being drawn. Acceptable comparators are ["<", ">", "<=", ">="] and these can be combined using "&" for 'and' and "|" for 'or'
- No equations are graphed unless -graph is called and is followed by one of BOUNDARY, REGION, GRAYSCALE

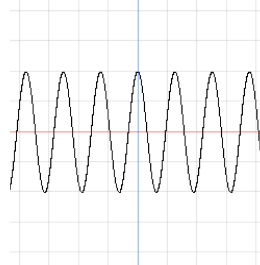
- Use ``-interactive 1`` to bring up a window allowing you to interact with the graph in an interactive way. You can drag the graph around using the mouse and use 's' and 'l' keys to zoom in and out (Note: may be slow if multiple equations are being graphed!)
- Use `-name <imgname>` to save the image of the graph with some name. If `-name` is not used, we save the image as "graph.png"
- Use `-save <filename>` to save all the functions and such from one session in a .ser file (<filename> may be different from <imgname> above). You retrieve these functions using `-load <filename>`

Previous specs implemented in Phase-1:

- Added to the collection of built-in functions (Exponential function, for example).
- The user can use a Command Line Interface to enter expressions that can be graphed and stored on command.
- Can graph functions using lines AND areas AND also grayscale



Before:



After:

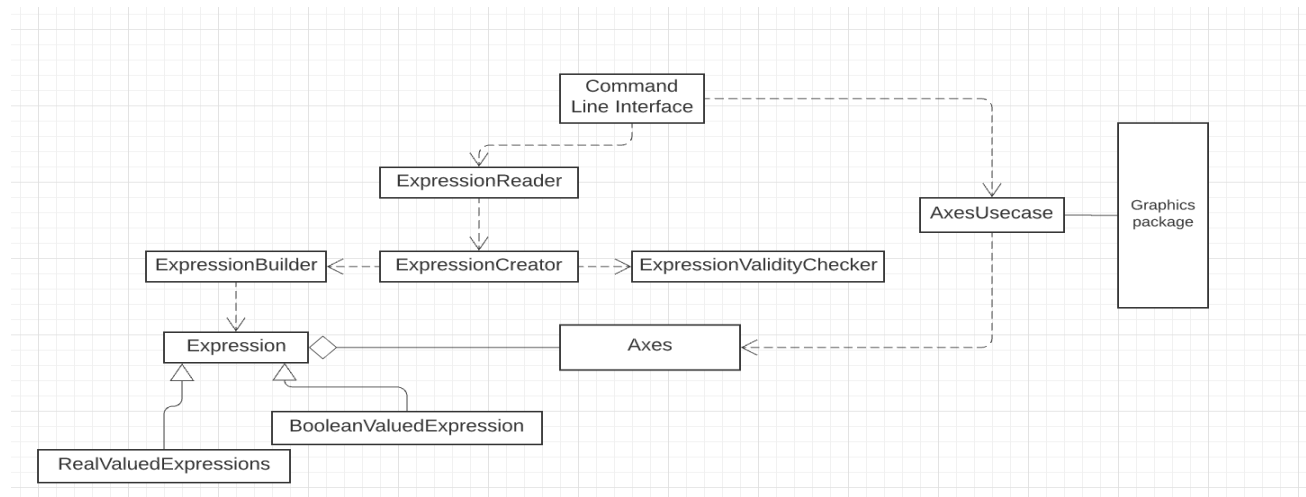
- Our program can now check for many types of invalid inputs from the user and prevent these from being submitted.

Previously completed phase-0 specs:

- Based on the string representations of functions that the user gives (e.g. `'y=x^2'`), our program can create Expression objects that:
 - Our program can graph:
 - functions from \mathbb{R} to \mathbb{R}
 - graph implicit functions, i.e. graphing points that satisfy equations of the form $f(x,y) = 0$ and $f(x, y, z) = 0$.
-

Clean Architecture

Simplified UML



Entities: enforce application rules on our persistent data.

Backend

- Expression (and it's subclasses):
 - they are how we store expressions that the user wants to graph (in a tree structure)
 - evaluate(): returns the output value of the expression that this Expression object represents, when evaluated on the input variable values.
 - These are the very core of our program and allow us to evaluate expressions.
 - They are abstract from any implementation of them and are therefore entities
- Axes:
 - Stores the data for graphing for example location of the origin, scale and of course a list of expressions that we will be graphing

Graphics

- RGBA:
 - represents a 32-bit color, interacts with other RGBA's by mixing / blending color
 - This class is critical for ensuring
- ImplicitGrapher:
 - Responsible for creating an array of pixels that correspond to the graph
 - Only depends on the Evaluatable interface and RGBA class via method calls. The first is used to determine which pixels should coloured (and to what) while the second is actually used to 'fill in' the colours
 - This is once again at the core of our graph by enforcing how graphing should be done
- AxesDrawer:
 - Responsible for drawing gridlines and the x and y axes
 - Only depends on RGBA class for similar reasons to before

Use Cases: enforce business rules on our entity classes.

Backend

- ExpressionBuilder
 - constructExpression(): takes Expression objects and some operator/function, and returns a new Expression built from these components.

- As this class directly manipulates Expressions it is a use case
- ExpressionValidityChecker
 - Enforces the businesses rules for what we consider to be valid expressions
- AxesUseCase
 - Used to work with an Axes object so we never have to depend on Axes directly

Graphics

- Grapher:
 - Mediates between the Graphics package and the Backend package
 - Gets all the necessary data from an Axes object via AxesUseCase
 - Uses ImplicitGrapher to produce an array of pixels corresponding to the final image

Controllers

Backend

- ExpressionReader:
 - Takes a string representing the expression that the user wants to graph, and converts it to a standardised list of strings (from which it's easier to create Expression objects) which can be interpreted by ExpressionCreator
 - Example: $\text{"cos}(x) + 5 \rightarrow [\text{"cos"}, \text{"("}, \text{"x"}, \text{")"}, \text{"+"}, \text{"5"}]$
 - As this class directly works with user input and needs to do some interpretation of it (e.g. whether we have a boolean valued expression or real valued), it is a controller
- ExpressionCreator:
 - Takes the list of strings produced by ExpressionReader to recursively construct corresponding Expressions through calls to the ExpressionBuilder class
 - This class *controls* the flow between ExpressionValidityChecker and ExpressionBuilder

Frontend

- CLIHelper
 - Has all the methods for interpreting user inputs and calls the appropriate methods from the various classes
 - Calls suitable methods from other classes depending on user input
- GUIHelper
 - Controller class for the GUI

Gateways

- DataReadWriter
 - Used for saving and reading from .ser files
- ImageWriter
 - Saves an image (of the graphs)

User Interface

- CommandLineInterface
 - Used to read in user input from command line
- GLGUI
 - Allows user to interact with the graph e.g. by moving around and/or zooming in and out

Scenario walk-through

1. Suppose user input is ``-eq "y = 3 + cos(x^2)" -graph BOUNDARY``
2. The CLI will separate this and try to create an expression for `"y = 3 + cos(x^2)"`. This is done by calling `ExpressionReader.read()` on the expression (note that all CLI is doing is finding the expressions to be graphed)
3. `ExpressionReader` recognises this as an implicit function. It then rearranges the terms to `"y - (3 + cos(x^2))"` and then calls `ExpressionCreator` to actually create these expressions
4. Expression objects are created by calling `ExpressionCreator.create()`.
 - a. This method works recursively by dealing with single terms expression (e.g. numbers and variables) as a base case.
 - b. We then find operators (that don't lie in any pair of brackets) recursively build expressions for the left and right. In this case we build 3 and `cos(x^2)`
 - c. As `cos(x^2)` is a function we first build its input (once again this is done recursively)
 - d. We then build a cos expression and set its input to `x^2` (all functions have an inputs attribute that can be set as needed)
 - e. A similar things is done with `"y"` (this being a single variable is handled in the base case)
5. Finally now that we have a constructed expression this can be passed back to the CLI which passes it onto our `AxesUseCase` class which in turn adds this expression to the list of expressions being stored in `Expression`
6. Grapher stores `Axes` and calls `ImplicitGrapher` on each one (for now, later we will use `ExplicitGrapher` as well)
7. `ImplicitGrapher` calls `evaluate()` on the expression passed in, loops through pixels in image and writes to each pixel depending on the parameters given
8. Grapher then writes the pixel array to an image

SOLID

Single responsibility principle: Each class should have only one responsibility i.e. reason to change. In other words, each class is only responsible to one actor.

- In our program, every class only has one responsibility, this easily seen for classes like `ExpressionCreator` and `ImplicitGrapher`, which only have one public method
- Other classes also follow this principle, for example `'ExpressionReader'` takes a string representation of an expression and creates an equivalent `Expression` object. `'Grapher'` is specifically responsible for mediating between `Axes` and `ImplicitGrapher`, `'AxesUseCase'` is only used to manipulate an `'Axes'` object

Open/Closed principle: Code should be 'open for extension but closed for modification' i.e. we can add new features without changing much existing code.

- One example is that our `Expressions` return a generic type `'T'` upon being evaluated. This means, for example, we could easily add `VectorValuedExpressions` which return an array of floats representing a vector upon being evaluated (and this would require almost no change to the existing code)
- It's also very easy for us to add new 'built-in function' (i.e. functions that can be graphed simply by the user entering their name into their input expression string) `Expression` types to our graphing calculator e.g. `exp`, `log`, `cos`, simply by extending the `FunctionExpression`

subclass of Expression. This is evidenced by the fact that we have done this multiple times throughout development.

Liskov substitution principle: For any class X, objects of subclass Y should always be able to replace class X.

- Wherever RealValuedExpression is used in our code, we've tested that every subclass of RealValuedExpression (e.g. FunctionExpression, ArithmeticOperatorExpression) also works. This is because we've designed the only essential feature of RealValuedExpression to be that its evaluate method returns a float, which all subclasses of RealValuedExpression also do.
- Both BuiltinFunctionExpression (for things like "cos", "sin", etc. which we define) and CustomFunctionExpression (for things like " $f(x) = x^2$ ", " $g(x, y) = \tan(x/2) - y^2$ " that the user may define) are children of FunctionExpression. This allows us to add, subtract, multiply, divide and compose functions regardless of which kind they are (in particular the funcMap attribute in ExpressionCreator is from Strings to FunctionExpression so either type of FunctionExpressions is valid)

Interface segregation principle: Classes shouldn't depend on interfaces or methods they won't use.

- The 'Evaluatable' interface originally had 2 'evaluate' methods which seemed to violate this principle. Since then we changed this interface and the way we graph functions so that we only need one of the these

Dependency inversion principle: Dependencies should be on abstractions rather than on concrete uses. In terms of Clean architecture, this translates to depending on classes in the same layer or one layer below.

- We introduced an interface for 'DataReadWriter' so that 'AxesUseCase' would depend on this interface rather than the class itself.
- Although not strictly inverting a dependency, we have ImplicitGrapher relying on the Evaluatable interface to enforce boundary between it and Expressions
- Expressions for example don't depend on anything as these are the greatest abstractions in our program.
- Introduced "GUI" interface which is an interface adapter/presenter, and GUIHelper which is a controller.gateway. Now, it is justified to access use cases in GLGUI, since GLGUI implements the "GUI" interface.

Design Patterns

Composite Design Pattern:

- Used in storing expressions as a complex object, such that each part of an Expression object is an Expression object in itself which is the core basis of the CDP. For example, $(x+2)*(2*x)$ is composed of the two smaller yet perfectly valid expressions $(x+2)$ and $(2*x)$, which can be further broken down easily.

Builder Design Pattern:

- Description: The pattern handles input (some representation of a complex object) by having one Reader class to parse the input, and call a Builder class to repeatedly build the complex object one step at a time (with each intermediate state of the complex object being stored in Builder).

- Current Implementation: In our case, the input is a list of strings representing the components of an expression e.g. ['x', '<', '5'], which ExpressionCreator (our Reader) parses and repeatedly calls ExpressionBuilder (our Builder) to build.
- Work left before the deadline: The Builder Design pattern may not actually be implemented strictly correctly according to the description we were given. We're looking into either implementing the design pattern more accurately, or replacing it with a more appropriate design pattern like Abstract Factory.

Observer Design Pattern:

- Suppose a user defines $f(x) = x^2$. We want to store this so that that user can refer to $f(x)$ later as well (for example they could then graph something like $y = f(\cos(x))$). This means that we need to store a collection of 'named functions' somewhere. The most natural place for this is in the Axes class as this class stores all the necessary data for graphing. However, ExpressionCreator and ExpressionValidityChecker also need to be able to access this data so that they know what f refers to an input like $y = f(\cos(x))$. Thus we have ExpressionCreator and ExpressionValidityChecker store an attribute of all the named functions and observe Axes so that every time a new function is added to Axes, this attribute gets updated to reflect this. ([PR#129](#))
- The reason that we wanted ExpressionCreator (and ExpressionValidityChecker) to store this attribute rather than access it directly from Axes was so that Expressions could be as encapsulated as possible. In particular we could directly move the classes to another program working with Expressions with minimal change required. Admittedly this leads to a duplication of data. Hence, in hindsight, we believe that it would be a better design choice to construct a common database where all the relevant classes can extract the information they need about functions, instead of storing named functions in multiple classes. Given a bit more time, we would have implemented this change.

Reasoning Behind Major Design Decisions

- Separating ExpressionCreator and ExpressionReader
 - Initially, ExpressionCreator was a part of ExpressionReader. That is, ExpressionReader originally parsed the string representation of the user-input expression, and also constructed the creation of the ExpressionObject.
 - We felt that this violated SRP, because ExpressionReader would need to be changed firstly when we change the way that the user input is parsed, but also secondly if the algorithm for Expression construction changed.
 - Thus, we chose to separate the two classes to better fulfill the SRP by separating the first and second responsibilities into ExpressionReader and ExpressionCreator respectively.
- Having a separate class for checking validity of Expressions
 - Originally, ExpressionReader was supposed to ensure expressions are correct, and if they weren't, we would raise some sort of exception. A lot of work went into Reader simultaneously checking validity of the expression, which violated single-responsibility as well as made the code quite smelly. We ended up deciding that this feature should be separated from ExpressionReader. So ExpressionValidityChecker was created to handle checking validity. This made the code cleaner while also satisfying design principles.

- We would like to emphasise that there is in fact little coupling between ExpressionCreator and ExpressionValidityChecker. We know this because we had a slightly different implementation of ExpressionCreator before and we were able plug in ExpressionValidityChecker with relative ease into it.
- Distinguished between real-valued functions and boolean-valued functions
 - Originally, we had boolean-valued functions handled by having the output be 1 representing 'true' and -1 representing 'false'.
 - We changed it because the expression '1 & 1' previously evaluated to 1, when it shouldn't have. This was a sign that there wasn't a meaningful distinction between real values and boolean values, which could have hindered development and features.
 - Instead, we created two subclasses of Expression: RealValuedExpression and BooleanValuedExpression. The former which accepts only other RealValuedExpressions (e.g. cos, x, 5) as sub-Expressions, and the latter accepts ComparatorExpressions (e.g. representing '<' operators) or LogicalExpressions (e.g. '<boolean expression> & <boolean expression>') as sub-Expressions.
 - In addition, having this distinction helps us with a major design issue -- How to store the domain for an expression. We first found out that it's the most natural thing to do to store the domain of an expression in the expression itself, so that it conveniences the grapher when evaluating (trust us, this is the most natural way we can think of). However, back to when we don't have the distinction between "RealValuedExpression" and "BooleanValuedExpression", we get stuck on an infinite storing domain problem (i.e. we'd have to store the domain for a domain). This is where the two newly defined notions come to rescue -- we only have "RealValuedExpression" to store domain, but never for "BooleanValuedExpression".
 - Also, as mentioned previously, by having a generic return type we can evaluate (and graph) other kinds of Expressions as well
- Using the 'Evaluatable' interface
 - Allowed us to enforce a boundary between Graphics (specifically for ImplicitGrapher) and Expressions. As ImplicitGrapher is an Entity, in principle it depending on Expressions would be fine. However we wanted there to be a boundary between these two, largely unrelated, parts of the program. So we have ImplicitGrapher depend on an interface that (RealValued) Expressions implement.
- Using a 'Constants' class
 - We have a 'Constants' class for things such as the operators, names of functions, special characters, etc. This class acts as a central 'database' that relevant classes can access to retrieve data (for example when parsing through expressions or checking that a given input can be interpreted by our program).
 - All the data is specific to Expressions (it contains specifically the operators, functions and variables that we use to build Expressions) and provides one obvious place to make changes. For example when we added comparators like '>' and '<' it was a simple matter of adding these to our collection of operators. This is why we feel that this provides the cleanest way of storing the relevant information.
- Creating Expressions recursively
 - This is the most natural way of dealing with arithmetic expressions
 - It may not be immediately obvious by looking at the method calls that 'realValuedCreate' is recursive for example, but this is because we broke it into helper methods

- The base case is when we just have a number or a variable, this is handled directly by ``realValuedCreate``
- The next case is we may still have a single term but this time it's a function such as `"cos(x^2)"`
- We find the inputs to a function using ``findFunctionInputs`` and call ``realValuedCreate`` on each of them (note the recursion)
- The final case is we have some kind of operator like `"3 + 5"`, which is handled by ``createOnOperators`` by creating expressions for everything to the left of the operator and everything to the right and creates Expressions for them, once again done recursively

Packaging Strategies

- We primarily packaged by components: 'Backend', 'Frontend', 'Graphics', 'GUI'.
 - Backend consists almost entirely of the construction (by parsing the input string) and storage of various Expression objects.
 - Frontend consists of the user interface. Currently it consists of the CLI and related classes.
 - Graphics consists of the classes involved in evaluating Expression objects at appropriate points (decided by classes).
 - GUI contains the windowing and user interactivity features. Classes in the package are not directly run by the user, instead the CLI can bring up an interactive window.
- We also considered packaging by layer (i.e. which clean architecture layer a given class is in), but we ultimately decided to package by component because that allowed us to more cleanly divide work between members. This is because we had 4 people working on the backend (Expression objects) and 4 people working on the frontend (graphics and the user interface).
- Moreover, we've packaged every subclasses of "Expression" (including itself) in the package "Expressions". The reason is that whenever we work on the code, we don't want to be bothered by many expressions as they are entities and usually we don't do much with them.

Contributions

Andrew (GitHub: Andrew-Huangg): During phase 2, I extended the functionality of the `AxesUseCase` class by adding a `createAxes` method. I added tests to our test suite and worked on improving test coverage, which was previously lacking in phase1. Worked on some minor details in `CLIHelper`, as well as some refactoring in other classes. [PR#17](#), [PR#54](#): Creating Axes and AxesUseCase classes. These classes are significant in manipulating and representing important business data and rules within our software.

Binseong (GitHub: Andric0901): Throughout Phase 2 and the entire semester, I have been working mainly on the Command Line Interface code. After the rough implementation of the CLI has been completed, I have cleaned up the overall code and pulled out the helper methods to a separate `CLIHelper.java` file for better extensibility. Contributed also on minor fixes across all the project files and documentations for other files. [PR#60](#): This pull request mainly fixes the errors present in the CLI code, to account for code smell issues.

Dihan (Github: dnilyo): My contributions during phase 2 have unfortunately been smaller than during phase 1 so far. I'm working on 2D valued expressions (currently we only have 1D valued expressions i.e. real-valued expressions) and finalizing the Builder design pattern (or replacing it with another design pattern if needed). I've also implemented some additional error detection features. I haven't yet pushed my code to the repo.

Louis (Github: lz-uoft) For phase 2 I finished integrating the OpenGL GUI into our project, with the option to use our ImplicitGrapher class to do the graphing instead of using GL shaders ([PR #124](#)). I also refactored the GLGUI to better adhere to the Solid and Clean designs (PR's [#128](#), [#133](#)). I delegated specific smaller tasks, such as GUI input and text drawing, to other members while providing high-level guidance on how to implement these features. I also created a 3D graphing demo using GL shaders ([PR #152](#)) which can eventually be integrated with the rest of our project code. I also fixed several miscellaneous issues and implemented more features.

Kevin (GitHub: kevin-j-wang): I worked on fixing some visual bugs with the graphical element of the project, as well as refactoring a bunch of code to be more in line with clean architecture. I attempted to fix a visual bug relating to asymptotes and NaNs being drawn where they should not have, though I do not know if my implementation is the one in the final product. I also worked on a class to display numbers visually by reading in a font and reproducing the pixels where necessary. The commits are for serialization for saving data, as well as my implementation of ExplicitGrapher which visually graphs functions in 1 variable. [PR#51](#) [PR#63](#)

Omar (Github: Plopstuff): My contributions since phase 1 have not been as substantial as during phase 1. For the most part my job was finished in phase 1 so my main job for this phase is just to be a helping hand as well as providing input on issues we have. I am also now working on figuring out a better design implementation for constants, whether it's to split up the class into smaller ones or using a json file to save them. I have also implemented a toString for expressions in order to be able to extend our program to work with OpenGL. During the last crunch, I will be taking on small issues that the team needs finished that will inevitably come up. I am also in charge of the accessibility report. My most substantial pull request would be the first real implementation of expressionParser. However, it got merged with another branch and so the link is for the pull request of the resulting code. I will include the pull request for the upgraded parser as well (which is when I figured out it's final implementation format).

[PR#26](#)

[PR#39](#)

Ted (Github name: tedtedtedtedted):

Phase 1: My two main contributions to phase 1 were:

1. Created and completed a working-version of ExpressionCreator and ExpressionValidityChecker which construct valid expressions and throw specific, customized exceptions with messages to report various reasons for invalidity of some expressions.
2. Offered a re-design of our Backend (in the old branch UltimateBackend) with the new notions of "RealValuedExpression" and "BooleanValuedExpression" added, and accommodated this change, and refactored out the backend to unite everyone's work together to have everything to be organized and clean. In addition, this is the first branch that is the product of the main branch and the working version of "ExpressionCreator" and "ExpressionValidityChecker" (which were in some other branches early on but not in main). Thanks to Rishibh, this huge "UltimateBackend" branch was successfully merged into the main branch in the link: [P71](#)

Phase 2: I have transitioned to the GUI team, under the guidance of Louis, and also participated in the Backend team (will soon have the strategy design pattern completed before the due!). I've learned a lot about OpenGL through Youtube, online tutorials, as well as Louis' explanations and demo). My code contributions are to implement many features of our GUI, including 1. [P127](#) Flipped the graph when working with OpenGL 2. [P134](#) Enable the "drag-move" (google-map-like) operation 3. [P138](#) Enable the "zoom-in/out" operation. (special thanks to Louis for letting me complete these, as I'm such a beginner). Lastly, I've written the four sections of the presentation slides based on (summarized from) Rishibh's work on the design document. The four sections are "Clean architecture" + "SOLID" + "Design Pattern" + "Major Design Decisions".

Rishibh (Github: rishibhp): After Phase 1, I worked on implementing the Observer Design Pattern for ExpressionCreator/ExpressionValidityChecker and Axes so we could enforce the boundary of Expressions being almost entirely independent ([PR#129](#)). I also took a step back from the code but continued to review pull requests and add to the Design Document.

One of the major features I implemented was domain handling. Originally domains had only been an attribute of CustomFunctions which meant the user wouldn't be able to restrict implicit functions for example. We realised that making domains an attribute simply of RealValuedExpressions would make things a lot easier and wouldn't violate any principles as most 'normal' expressions can be interpreted as having a domain of everything (for example sin and cos are defined everywhere). I worked on implementing this in the code: [PR#116](#)

Additional Info About Our Code:

Uses of Github Features

We have used the following GitHub features throughout the term:

- Pull requests and commenting on pull requests (on specific lines) to discuss about potentials changes that could be made
- Issues: we used this feature to highlight any ongoing issues in the current code, as well as linking the relevant pull request so that merging the pull request would automatically close the linked issue
- Network and rebase: to make sure the commit timeline stays roughly linear, we have used network and rebase, and make sure to review the pull requests first before pushing one.
- Use of project boards to keep track of to-do's in the context of the big picture (although we weren't super frequent about it).

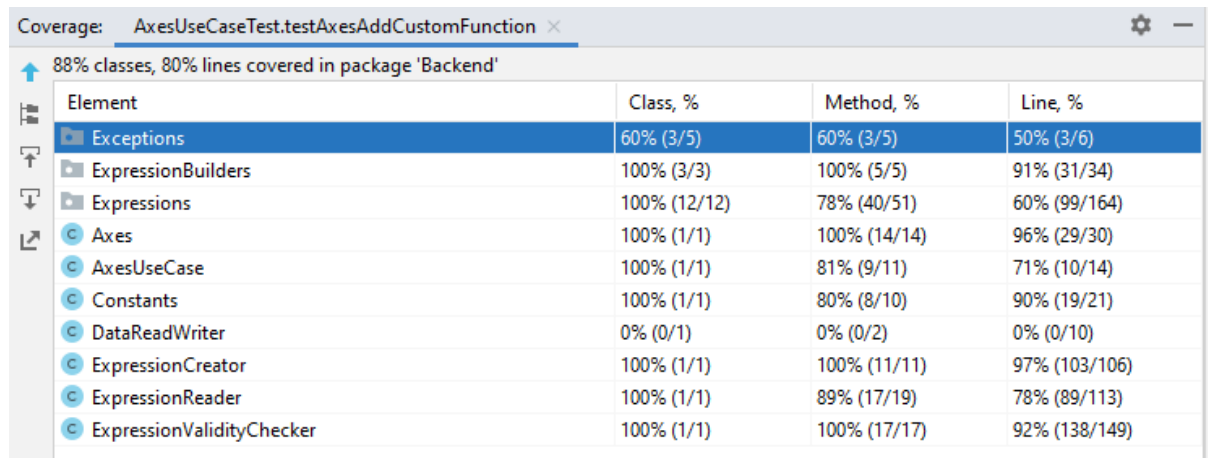
Code Style and Documentation

There are no warnings when IntelliJ is opened.

We added documentation for most of the classes.

A Java programmer would be able to understand any of the files they open. The documented code in the project may be complex, but the documentation and comments clarify the code. They also entertain cases where there may be confusion by giving examples. Other code is simple enough to understand at a glance.

Testing



Element	Class, %	Method, %	Line, %
Exceptions	60% (3/5)	60% (3/5)	50% (3/6)
ExpressionBuilders	100% (3/3)	100% (5/5)	91% (31/34)
Expressions	100% (12/12)	78% (40/51)	60% (99/164)
Axes	100% (1/1)	100% (14/14)	96% (29/30)
AxesUseCase	100% (1/1)	81% (9/11)	71% (10/14)
Constants	100% (1/1)	80% (8/10)	90% (19/21)
DataReadWrite	0% (0/1)	0% (0/2)	0% (0/10)
ExpressionCreator	100% (1/1)	100% (11/11)	97% (103/106)
ExpressionReader	100% (1/1)	89% (17/19)	78% (89/113)
ExpressionValidityChecker	100% (1/1)	100% (17/17)	92% (138/149)

We created test suites for ExpressionCreator (to test whether a given list of strings was converted to the valid Expression object that represents it) and ExpressionReader (to test whether a given expression string results in the correct Expression object being created).

- Tried to identify as many edge cases to test as possible by considering any obvious way that our class algorithms might fail.
- Tested several functions both simple and composite using our custom functions and visually verified they were correct.

As for the graphics package, we would test whether the graphs are correct, the output is what we want, and if there are any issues with the graphs manually by running the code and examining the output. As the code is for graphics and drawing, it is much more efficient to check whether everything is working manually rather than develop test suites. If we had focused on developing actual tests, many bugs that we have found would have taken ages to find given the amount of work that would go into developing test suites for graphics output.

It would take a long time to develop the suites as we can only check whether the output is correct if we evaluate it at every pixel and compare the output of our code with what is correct. That involves knowing in advance what every pixel evaluates to. However, with features like shading, mandelbrot, grids, knowing in advance what a pixel evaluates to would itself require some algorithms, which will need testing too.

Refactoring

1. 'getOuterItems' method is an example where we came across the same problem of trying to find some set of items that were outside any pair of brackets. Originally, we had a 'getOuterOperators' method that was specifically for operators and 'getOuterFunctions'. But then we also needed one for commas and then comparators, etc. Thus we extracted this out into its own function.
2. Originally, we had only ExpressionCreator to create objects, but as we begin to add more functionalities, the class becomes too huge. Then, we find a clever way for the separation of validity checker and actually building the expression. Here, by "separation" we mean if we ever need to change the rule for building or checking validity of some expression, we can do them separately in "ExpressionValidityChecker" and "ExpressionBuilder", because the structure of managing these are fixed in "ExpressionCreator".
3. All exceptions are now packaged in the "Exceptions" package. In addition to that, we have a notion of "BaseCaseCreatorException" and "CompoundCaseCreatorException" which are the

only direct subclasses of “InvalidTermException” which covers all invalid expressions cases and report the specific reason for validity. This way of handling exceptions is helpful for reducing the code (to handle exceptions) and for the understanding of readers.

4. Early on, we only have “Expression” abstract class, and it is real-valued, so the way we treated any boolean-valued expression is to use “1.0” and “-1.0” to represent “true” and “false”. However, we cannot report invalid cases like “1 AND 1” as user input. Due to this, we developed the notion of “RealValuedExpression” and “BooleanValueExpression”, and now we handle these differently in “ExpressionReader”, “ExpressionCreator”.
5. We converted implicit grapher from static to non-static. We refactored the big function implicitGraph by splitting the loop iteration into a separate helper function, writePixel.
6. expressionParser in ExpressionReader initially had too many things going on within the loop, keeping track of multiple fixes, each with their own algorithms. Although it increases run time by a specific factor, we took the decision to extract all fixes from the loop and just pass through the list again after the loop ends through some helper functions, increasing clarity.
7. We originally had different classes for each builtin function like a Cosine class, Sine class, Tangent class, etc where each one only differed by 1 or 2 lines in terms of how they were evaluated. This quickly became unwieldy so we combined these into one class making it a lot easier to work with.