# Design Document

## 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 [“-dim”, “-eq”, “-save”, “-load”, “-graph”].

Uses:

- “-dim” <num> is used to set the dimension of the Axes (currently this does nothing as we do not have 3D graphing implemented, but this will be useful in the future)
- “-eq” <expression> is used to define the expressions that you want to graph. Currently you can only graph one function at a time (calling “-eq” multiple times will lead to only the last equation being graphed)
    - 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 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)

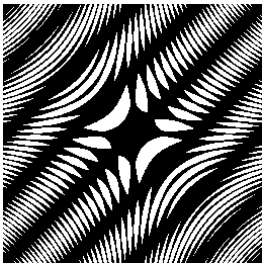
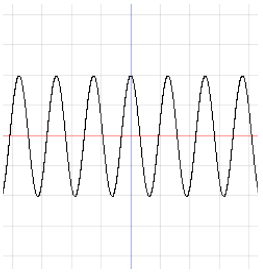
Note: the validity of user inputs is not checked thoroughly hence incorrect input may lead to errors

Note: all graphed images are currently saved as “test.png”

## Updated Specifications

- New specs implemented in Phase-1:
    - Added to the collection of built-in functions (Exponential function, for example).

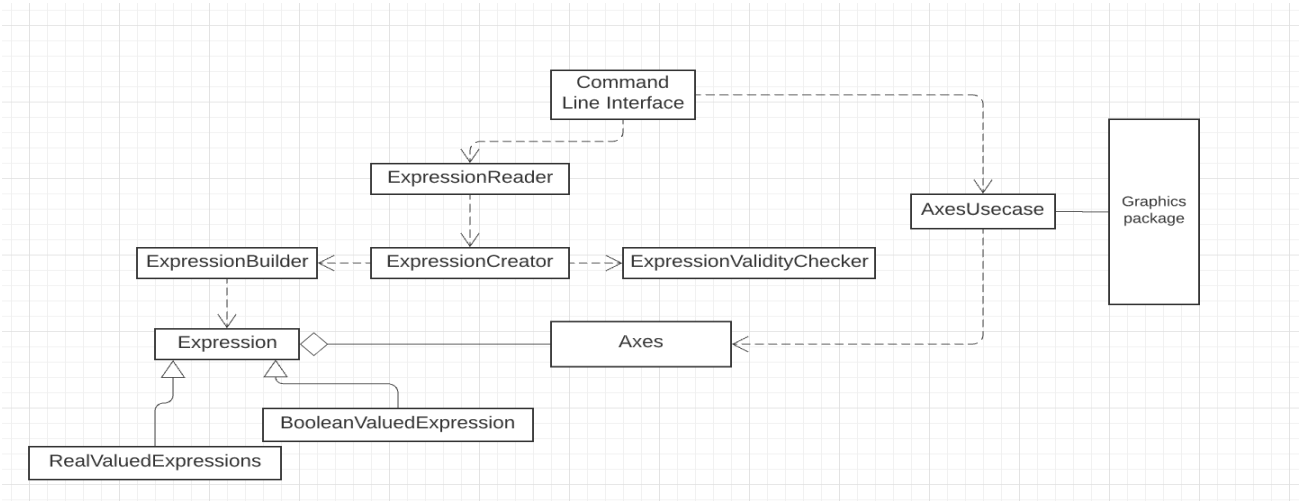
    - The user can use a Command Line Interface to enter expressions that can be graphed and stored on command.
    - Can graph functions using lines AND areas AND also grayscale

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

- 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.
        - Axes: stores a set of expressions that are to be graphed.
        - Constants: stores useful constants in our program (e.g. pre-defined function named, recognized arithmetic/logical operators etc.)
    - Graphics

        - RGBA: represents a 32-bit color, interacts with other RGBA's by mixing / blending color
- 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.
        - AxesUseCase
    - Graphics
        - ImplicitGrapher: method graph() writes to int array representing an image depending on input parameters
        - Grapher: uses ImplicitGrapher to graph equations, uses RGBA to draw gridlines and axes with transparency on top of graph
- Controllers, Gateways and Presenters: classes which mediate user input/output and program algorithms (through use cases).
    - Backend
        - ExpressionReader: takes a string representing the expression that the user wants to graph, and converts it to a standardized list of strings (from which it's easier to create Expression objects)
        - ExpressionCreator: takes the string representation produced by ExpressionReader and constructs the corresponding Expression object through calls to the ExpressionBuilder use case.
        - ExpressionValidityChecker: checks the validity of the user inputted string at various stages of ExpressionReader and ExpressionCreator.
    - Frontend
        - CLI: allows the user to enter commands to display graphs for and store expressions.
- Scenario walk-through
    - Suppose user input is "-eq "y = 3 + cos(x^2)" -graph BOUNDARY"
    - The CLI will separate this and try to create an expression for "y = 3 + cos(x^2)". This is done by called ExpressionReader.read() on the expression. ExpressionReader recognises this as an implicit function. We create left and right expressions and then create a new expression that is the left minus the right expressions
    - Expression objects are created by calling ExpressionCreator.create().
        - This method works recursively by dealing with single terms expression (e.g. numbers and variables) as a base case.
        - 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)
        - As cos(x^2) is a function we first build its input (once again this is done recursively)
        - 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)
        - A similar things is done with "y" (this being a single variable is handled in the base case)
    - 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

- Grapher stores Axes and calls ImplicitGrapher on each one (for now, later we will use ExplicitGrapher as well)
- ImplicitGrapher calls evaluate() on the expression passed in, loops through pixels in image and writes to each pixel depending on the parameters given
- 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 our program, every class only has one responsibility, often evidenced by having only one public method with a very specific purpose that no other class has.
    - Example: `ExpressionReader` is the ExpressionReader.read takes a string representation of an expression and creates an equivalent Expression object.
    - Example: `ExpressionCreator` (ExpressionCreator.create takes a list of strings representing expression components e.g. ). In fact, one of our major design decisions was to separate ExpressionReader and ExpressionCreator (which were originally a single class) in order to ensure that each obeyed the SRP (see page 3 for details).
- 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 Expression structure allows us to easily create new types of Expression objects simply by extending from Expression, without altering any other subclass of expression.
    - For instance, it's 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.
- Liskov substitution principle: For any class X, objects of subclass Y should always be able to replace subclass X.
    - Wherever RealValuedExpression are 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 does.
- Interface segregation principle: Classes shouldn't implement interfaces or methods they won't use.
    - The only interface in our program is the interface "Evalutable" in Graphics. Only RealValuedExpressions implements this interface, and uses all of its methods. All of these methods are necessary in order to be able to graph RealValuedExpressions.
- Dependency inversion principle: Lower level modules (in the clean architecture hierarchy) never import from higher level modules, but may only depend on interfaces that higher-level modules also rely on.
    - We haven't identified any lower level modules that rely on higher level modules anywhere in our program.

# Design Patterns

- Composite Design Pattern is 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.
- The Builder Design Pattern will be used to construct ExpressionObjects.
    - 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. In our Main branch, we currently have a very incomplete implementation of the Design Pattern i.e. Builder doesn’t store the intermediate state but rather returns them (effectively just a UseCase that calls Expression constructors and returns the resulting Expression).
    - Future: We intend to fully implement the design pattern in phase-2 by ensuring that intermediate patterns are stored as an attribute in ExpressionBuilder.

# 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-inputted 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.
- Builder Design Pattern
    - ExpressionCreator was initially a UseCase (with direct calls to Expression subclass constructors). However, keeping the code to construct new Expressions inside the class for () seems to violate the SRP, because we’d have multiple reasons to alter ExpressionCreator:
    - Thus, we thought that the Builder Design Pattern could help follow the SRP and minimize dependency between ExpressionCreator and Expressions.
- 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.

    - 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 (recent) 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".
- Decided to implement a use case class for Axes, which is responsible for manipulating attributes of an instance of Axes, as well as saving and loading Axes instances. Before, higher level code used to call axes methods directly, which violated clean architecture's dependency rule.
- 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.

# Packaging Strategies

- We primarily packaged by components: 'Backend', 'Frontend', 'Graphics'.
    - 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 only consists of the CLI, but will be expanded to include the GUI.
    - Graphics consists of the classes involved in evaluating Expression objects at appropriate points (decided by classes).
- 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.
- Even furthermore, we grouped many built-in functions to "BuiltInFunctions" package since we rarely bother with them compared to other expressions.

# Progress Report

- Louis: Refined implicit grapher, added boundary/region/grayscale options, converted from static to non-static. Added RGBA class for clean color/pixel operations. Created axes/gridline drawer. Started developing OpenGL GUI, completed basic demo of interactive mandelbrot with mouse input. Future: finish GUI, get 3d explicit / implicit graphers working.
- Kevin: Worked on explicit grapher, a class that calculates approximately where the pixels of a function should go and writes it to an array to be rendered. Also made a class to serialize and save data.
    - For phase 2 I plan on working further on the graphical end of the project, hopefully for animated and interactive graphing, and GUI. More specifically, we intend to have a software that can dynamically render any section of a graph at any zoom level, as opposed to currently where it renders a preset area of determined resolution and detail, and everything outside of the domain is simply not visible. This will involve external libraries.
- Binseong: Worked on the Frontend Command Line Interface class, a class that user directly interacts with, which allows the Command Line Interface to split appropriate user inputs to be sent to other classes.
- Dihan:
    - Helped with backend design decisions (Abstract Syntax Tree structure for Expressions, Builder Design Pattern, separating real-valued and boolean-valued expressions etc.)
    - implemented Comparator and Logical Operators
    - created prototype code for the AST structure and validity checking
    - created the test suite for ExpressionCreator
    - Plans for phase-2:
        - completing the Builder Design Pattern
        - possibly altering validity checking
        - test suite for validity checking
        - predicate functions (i.e. user-defined boolean functions)
- Rishibh
    - Worked on implementing domains for functions
    - Worked on implementing multivariable functions and user-defined functions. The latter is nearly complete, all that remains is for ExpressionReader to parse and create these appropriately.
    - Future: We might be able to use Observer Design Pattern in the future to remove a (minor) dependency between `ExpressionCreator` and `Axes`.
    - Future: Manipulation of user-defined functions to create new expressions e.g. if the user defines have functions f and g, they should be able to plot f(x) + g(x), f(x) - g(x), f(x)g(x), \frac{f(x)}{g(x)}, f(g(x))
- Omar: Worked on ExpressionReader class, which is responsible for parsing user input expressions sensibly. Worked on documentation and refactoring for ExpressionCreator, ExpressionReader, and ExpressionValidityChecker. I was also involved in deciding with Rishibh the input we would accept from the user, as well as what we would reject. I plan on working on GUI for phase 2 and some work concerning parsing input at the CLI level and interpreting it.
- Andrew: Worked on Axes and AxesUseCase class in the backend package. Worked on testing these classes. Axes is responsible for storing expressions, scale, etc. AxesUseCase is

responsible for manipulating Axes objects and saving them. For phase 2, plan to work on higher level classes in the graphics package.

- Ted:
    - Created and completed ExpressionCreator and ExpressionValidityChecker which creates any valid expression (as being given a list of strings from ExpressionReader), and most importantly (the hard part), it reports specific reasons for why the expression is invalid by throwing exceptions with customized messages.
    - After long hours of contemplation on our Backend, I felt there's much room for improvement, both in terms of the code styles and the overall structure. Especially, I came to the conclusion that we need the notions of "RealValuedExpression" and "BooleanValuedExpression" instead of just having one "Expression". This is the basis of Backend, and by developing these notions, we've found that it solves many problems that we faced before (e.g. what to do with "1 AND 1"; how to store the domain for expressions). To put my thoughts into actions, I've spent days on restructuring Backend, gathering everyone's pre-existing thoughts and codes, as well as my individual opinions, and eventually, I have offered a complete and minimal design of our Backend, and explain/demonstrate the design details/logic in details to my teammates. (Special thanks to our group leader, Rishibh, for spending long hours discussing and helping me with many design decisions and plenty of technical details, he is a great leader indeed!)
    - In future, I am planning to work on the GUI with Omar&Louis&Kevin (the phase 2 graphic team). More specifically, the goal is to achieve the GUI features in Math3D (take a look at that, please)!!!.
- Open-ended questions
    - Is AxesUseCase worth keeping, given that it mainly just calls setters and getters in Axes?
    - Is it a necessary to mainly keep validity checks in one class?
    - Is it okay for controllers (in terms of clean architecture) like ExpressionCreator and ExpressionReader to the import Expression class?

# Additional Info About Our Code:

## Uses of Github Features

- Started using issues after Evan's suggestion
- Using network to verify that the timeline is roughly linear
    - Using rebase whenever possible to have maximum linearity
- Pull requests and commenting on pull requests (on specific lines) to help
- 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 mostly for ExpressionValidityChecker, ExpressionCreator and ExpressionReader, as they were from the more complicated 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

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).

- 28 tests for ExpressionCreator and 12 tests for ExpressionReader.
- 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 long 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. All expressions are packaged in “Expressions”, especially some built-in functions are grouped together into “BuildInFunctions”. This will bundle many different types of expressions

together and when working with code, we can ignore these and have them hidden under one folder.

5. Before, we had a notion of “chained comparators” which is an entity class. But this is rather a new notion and different from all other (binary) operators (when evaluating, we used a for-loop, so it’s quite complicated), so we have another competing version, “ComparatorExpression”, the way we evaluate this is the same we do with any other binary operators. In this way, we used something that we had to deal with comparators.
6. 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”.
7. 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.
8. 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.