Matrix: Design Document

Updated specification + Functionality

The goal of our worksheet generator is to create highly customizable math worksheets. Users first choose their desired topic: addition, subtraction, multiplication, and division. They can then select the range for the first and second operand (positive integers only) and if they would like to see negative numbers in their questions/answers. From here, the user can choose various formatting details: the title, the number of questions, and the number of question rows and columns per page. Once all parameters have been selected, the user can generate the worksheet. This will automatically save the worksheet to their worksheet history (records the worksheet details and the date created) and redirect the user to an image of the first page of the worksheet PDF. Finally, the user can download this PDF.

Our program also allows for the retrieval and storage of users and their user history. In other words, restarting the application does not remove existing users and their worksheet history. The user can use the saved details in their worksheet history to regenerate past worksheets. Since we save the seed used for randomization as well, the regenerated worksheet will be identical to the first.

Scroll to the bottom of this document to see the program UI which demonstrates all the features mentioned above. (Note: there is a small visual glitch on windows computers which we intend to fix, what is shown below is the correct display on mac computers).

Overall, we believe this program is sufficiently ambitious for our group as there are many different features we have implemented. To name a few, we have a functional UI, worksheet history data persistence, random worksheet generation, and worksheet to PDF conversion. Together, these features interact and coalesce into Matrix.

Code Organization + Packaging

From the start, we noticed there were three distinct parts to our project: (1) user functionality, (2) worksheet generation, and (3) the graphical user interface.

We decided to package our code by components that fall within these overarching categories. A component refers to a combination of business and data retrieval logic that can be used together. This essentially means each package has one public interface and package specific implementation details. In the case of worksheet generation, our code could be split into 3 distinct components: equation_builder, equation _entities, and worksheet_maker. On the other hand, user functionality was made into user_package and the GUI was made into user_interface.

In our equation_builders package, our interface is WholeBedmasDirector. In equation_entities, it is BedmasEquation. Finally, our worksheet maker's interface is WorksheetController. These component packages are packaged entirely separately from the user components as they do not interact.

Similarly, for the user_package, by putting all user-related functionality here we can abstract away the implementations and data storage for all things user-related, such that user_interface only interacts with the UserController.

Finally, the entire user interface was packaged into a user_interface package. This package contains every screen within the user interface, as well as StartUserInterface, which runs the entire program. Since each class within the user_interface interacts with one another (e.g. LoginScreen calls NewUserScreen), it is essential that each user interface class is packaged into a single user_interface package. These classes access worksheet generation and the user information through WorksheetController and UserController.

We chose to package our code this way as it seemed the most logical and would allow us to create many more packages than had we packaged by layer or by feature. Given our number of classes, packaging by layer or feature would have been very difficult for navigation.

Some issues we are working to resolve is preventing certain packages from accessing what really should be implementation details in other packages. For example, our WholeBedmasAddBuilder instantiates an Add class and assigns it to its instance of BedmasEquation, despite the two being part of separate packages and the Add class being an implementation detail rather than an interface. One solution could be to have WholeBedmasAddBuilder pass in a string or some other format that BedmasEquation parses to create an instance of Add, but we feel that this is redundant and introduces more room for error.

SOLID

SRP
Within the user_package, the entities User and History show ownership of distinct and single responsibilities: storing user data and storing worksheet generation history respectively.

The interface adapters in worksheet_maker also adhere to this principle. WorksheetController delegates responsibilities, WorksheetGenerator adds equations to a Worksheet, and PDFPresenter adds formatting details for displaying the worksheet PDF. Because of this division, if we wanted to change the implementation of how equations are added to a worksheet in the future, we can solely change the WorksheetGenerator and nothing else will need to be updated. Additionally, each class is easier to read and therefore debug.

OCP
We stored our questions (in Equation) as an expression tree of class Symbol and can be solved using a .solve() method. Symbol is an interface implemented by Value and Operator. Value is an abstract class that has abstract methods for several operations (add, subtract, multiply, divide). For now, Value is only extended by WholeNum which implements how whole number operations work (which is fairly simple). However, in the future, we plan on adding Decimal and Fraction which will require a different implementation. This demonstrates OCP since we can extend Values, adding new features without changing the expression trees and the methods (ex. solve()) that depend on them.
One problem with this implementation however is that the Symbol interface is currently empty. This was necessary since the expression tree needs to be one type, and it didn't make sense to group up Values and Operators into one large class.

LSP

LSP states that for any inheritance relationship (ex. Child extends Parent), an instance of Parent can be replaced with an instance of Child. This is followed in every example of Java subclassing, but some notable examples include WholeBedmasAddBuilder (and other operators) from WholeBedmasBuilder and WholeNum from Values.

ISP

The Worksheet class (which is essentially a list of Equations) is accessed by both WorksheetGenerator (which adds Equations to the worksheet) and PDFPresenter (which uses the Equations and formatting details to make a PDF). To do this, the Worksheet class needs getters and setters, however the former is only used by PDFPresenter and later by WorksheetGenerator. To make sure these classes aren't forced to depend on Worksheet methods they don't use, we created 2 interfaces, WorksheetInput and WorksheetOutput, that are implemented by Worksheet (check UI diagram below).

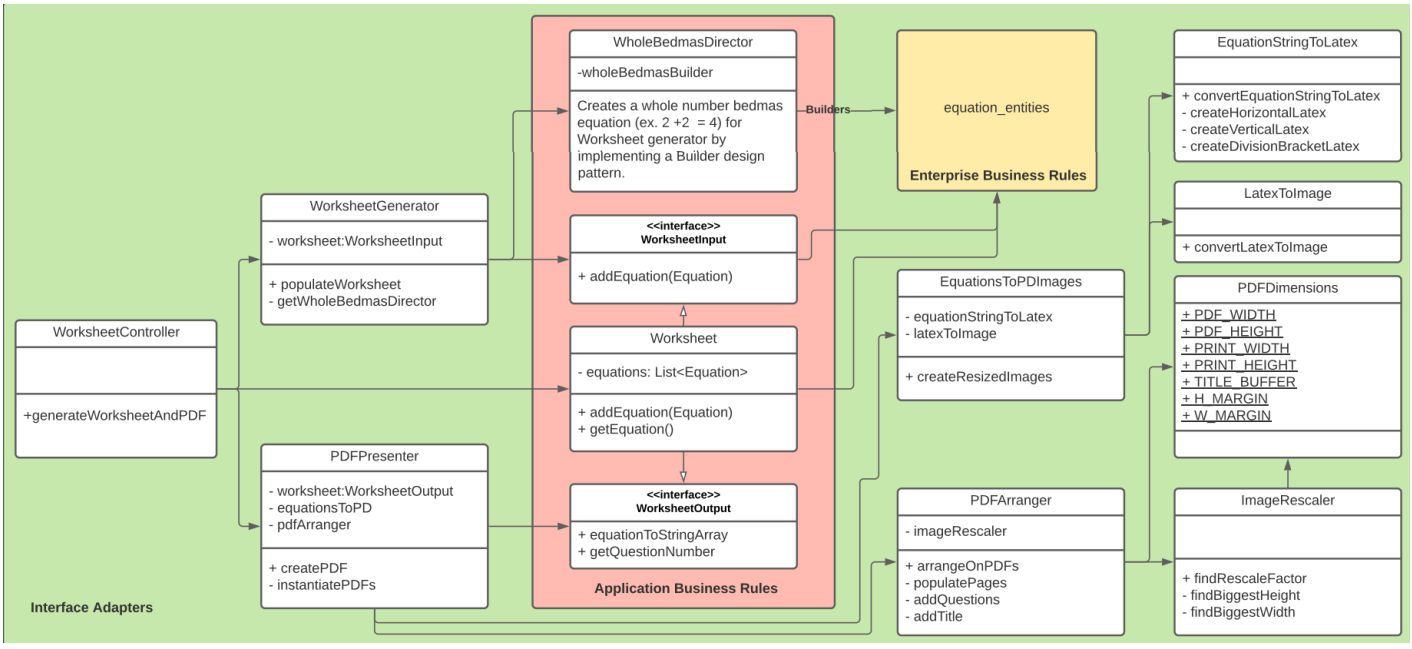
On the user_package side, through the use of DataAccessInterface, we leave it as a possibility to pass the data storage and retrieval to and from a remote database. This also demonstrates OCP.
https://deviq.com/principles/dependency-inversion-principle

DIP:

Our code consistently adheres to this principle in that we follow the layers of clean architecture and ensuring low level components depend on high levels ones, not the other way around (look at UML diagrams below). However, we may be missing some interfaces between major components. For example, our UI directly depends on WorksheetController and UserController. However, the DIP states that all classes should depend on abstractions. To fix this, I believe we should create an interface that both WorksheetController and UserController implement. Then, we would update the UI so that it depends on these interfaces rather than the Controllers. Still, our group is somewhat divided on this design decision so we would like further feedback before proceeding with this choice.

Clean architecture

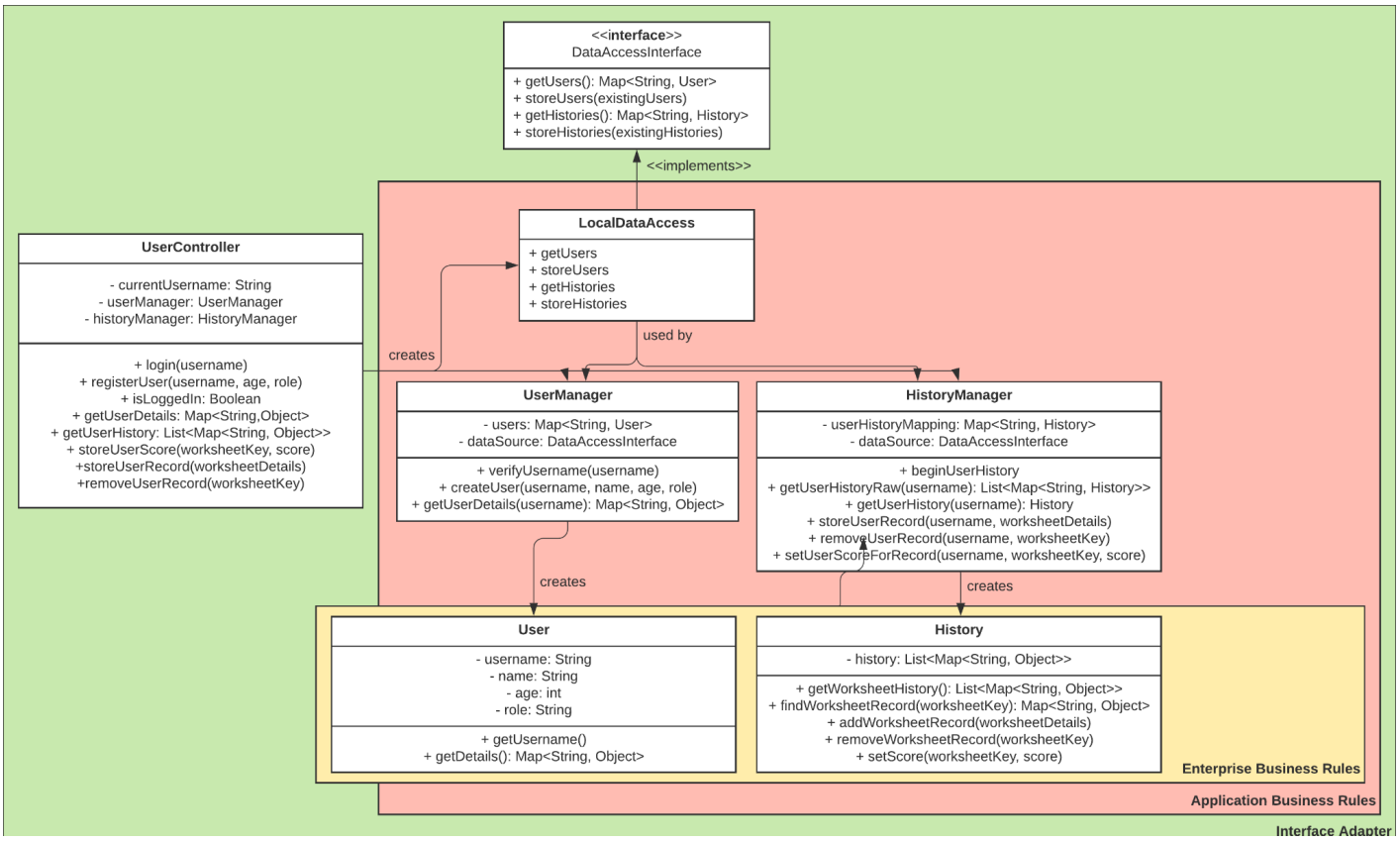
Worksheet_Maker UML Diagram



Worksheet_Maker Scenario Walkthrough

Various equationDetails and formatDetails are passed into WorksheetController. The WorksheetController creates an empty Worksheet class, and instantiates Worksheet Generator and PDFPresenter to the same instance of Worksheet. The WorksheetController tells the WorksheetGenerator to create all the Equations by using the equationDetails (ex. operators, operand ranges, number of equations etc.). In turn, this calls on the WholeBedmasDirector, which uses a Builder design pattern to create an Equation (that is a whole number bedmas type) which is saved in Worksheet (design pattern explained later). To convert the Worksheet into a PDF, WorksheetController now uses formatDetails (ex. number of rows/columns, type of questions) to call on PDFPresenter which generates 2 PDFs (one with and without answers). PDFPresenter receives the Worksheet as a String array, so as to not expose the underlying architecture, and converts it into a PD image (used in PDFs) using EquationsToPDImages (which calls on 2 helper classes). PDFPresenter can now arrange these PDImages onto a newly created PDF using PDFArranger. PDFArranger uses various PDFDimension constants and imageRescaler to optimally rescale the images to fit into the PDF.

User_package UML Diagram



User_package Scenario Walkthrough

The Dependency Rule is followed in (1) the user_package and (2) its interaction with the UI. Inside the package, we invert the dependency between data retrieval/storage and its usage in UserManager and HistoryManager. The DataAccessInterface interface specifies methods to retrieve/store user or history data. However, its implementation is dependent on classes that implement it. The UserController passes an object implementing DataAccessInterface into UserManager and HistoryManager instances, which the UserManager and HistoryManager instances call upon to retrieve/store data. By doing so, these instances do not need to know how data retrieval and storage is implemented, inverting the dependency. Secondly, the UserInterface only needs to interact with the UserController, in order to do user-related functionality. As such, there is no interaction at all between the UserInterface and any other class in the user_package.

The Dependency Rule is consistently followed as the interface adapters (including the UserController and WorksheetController) never mention any code declared in the outermost layer, which is the user interface. SImilarly, the user interface only calls on the UserController and WorksheetController. Both UserController and WorksheetController are instantiated in the startUserInterface class to follow the dependency rule. Instantiating both classes in the superclass Screen would have violated the dependency rule because Screen would need to know what methods are implemented in UserController and WorksheetControler.

The user interface utilizes the UserController to store valid users and their profile (including username, name, age, and role). When the user attempts to login, the user interface calls UserController.login(username) which checks if the username entered was valid. In addition, the user interface uses UserController to access the user's profile information on the "User Profile" screen.

In order to create and preview the worksheet at the user interface level, the user interface utilizes WorksheetController to call the generate worksheet and pdf method within the WorksheetController. The equation details and format details inputted at the user interface level are passed into this method. Clearly, the Dependency Rule is followed as it is solely the user interface that relies on the interface adapters, and not the other way around.

Design Patterns

Builder design pattern for equation generation

We decided to use the builder design pattern to generate our various BedmasEquations (whole number addition, whole number subtraction, whole number division, whole number multiplication) and we plan to apply a similar pattern as we introduce more question (fractions, decimals, LCM/GCF questions). Specifically, we created a Director (WholeBedmasDirector), an abstract Builder (WholeBedmasBuilder), and concrete Builders that extend the Builder (WholeBedmasAddBuilder, WholeBedmasSubtractBuilder, WholeBedmasDivideBuilder, and WholeBedmasMultiplyBuilder) to build various BedmasEquations containing whole numbers. This is useful for various reasons. First, it adheres to the SRP by separating the generation of various equations from the equations themselves. Without this pattern, we might have grouped the building, storage, and retrieval logic all into the equation classes themselves. Second, it helps us adhere to the ISP. Without a builder, BedmasEquation would require multiple generation methods (ex. whole number subtraction vs whole number addition) but not all would be used for a specific instance of BedmasEquation. Finally, it would help us extend our program when we eventually implement other types of equations (ex. fractions). Instead of modifying BedmasEquation directly, using this pattern, we can just create a new Director.

Facade design for worksheet generation

We implemented the Facade design pattern to simplify how our UI interacts with worksheet generation. Essentially, the UI needs to retrieve a PDF representation of the worksheet. However, directly calling on a class to generate the worksheet, retrieving that worksheet, and then converting it to a PDF would make UI's responsibilities very complex. Instead, UI interacts with a WorksheetController class that acts as a facade. WorksheetController then delegates roles to a WorksheetGenerator and PDFPresenter, to ultimately produce a PDF that UI can interact with.

Facade design for EquationsToPDImages

Converting text equations to a fully formatted PDF is a long process with numerous responsibilities. While PDFPresenter itself is part of a Facade design pattern, implementing all these methods would make the class unreadable. To fix this, we implemented another 2 layers of the Facade design pattern. The first is on the level of PDFPresenter. PDFPresenter depends on EquationsToPDImages to generate PDF specific images from the given equations, and then uses PDFArranger to arrange these elements on the PDF. PDFPresenter itself knows nothing of these specific implementations, and is simply responsible for delegating the roles. The second Facade design pattern is within the EquationsToPDImages class. This is because the string of equations must be converted into a latex intermediate and then PDImages. As these are two separate responsibilities, EquationsToPDImages instead calls EquationStringsToLatex to create the latex intermediate and then LatexToImage to convert this latex intermediate to an image.

Facade design for user_package

The UserController implements a Facade design pattern. It instantiates a DataAccessInterface-implementing object and passes that object into both UserManager and HistoryManager as a parameter. After which, most functions in UserController refer to a single function in UserManager or HistoryManager. For example, getUserDetails in UserController calls upon getUserDetails in UserManager to carry out the function on the current user. Using Facade design improved our program's readability and it will be useful if we want to refactor or add features in the future.

Dependency Inversion design for user_package

As mentioned earlier, dependency inversion is shown in how any DataAccessInterface interface-implementing object can be passed into the UserManager and HistoryManager objects, allowing the implementation of data retrieval and storage to be hidden away. This way UserManager and HistoryManager have loose coupling instead of a hard dependency.

Use of GitHub Features

To create the userPackage, Stanley and Kerim created a ***stan-kerim*** branch to work on user-related functionality. They branched off into their own sub-branches ***stan*** and ***kerim*** to work on distinct features, and they performed pull requests to *stan-kerim* to merge their code safely. If they wanted to merge their stable code for others to see, a pull request was sent from ***stan-kerim*** to ***main***. At least two other members reviewed the newly created pull request before merging.

To create the worksheet related packages, Will and Sean created a ***sean_will_2*** branch where they worked on worksheet generation related classes (and some PDF presentation). Upon splitting into the packages above, Will was able to work on equation_builder, equation _entities packages and Sean worked on the worksheet_maker package. These packages were separated enough so that Will and Sean could make frequent pushes directly to this branch without interfering with one another. Once the worksheet generation was stable, a pull request was sent to main in which the group members could do a code review before accepting.

Finally, to create the user_interface package, Ethan and Piotr created an Ethan-Piotr-UI2 branch. We were able to split up the work (each of us working on multiple screens) and we made frequent pushes to the branch whenever a screen was completed (to avoid any conflicts). Once the UI was stable, a pull request was sent to main, where the group members were able to review the code prior to accepting.

Code Style and Documentation

There are no style warnings in any of the classes present. Notably, we specify to suppress warnings in LocalDataAccess, as there is no work around to the “declaration not checked” error when deserializing User and History objects. Javadoc is used extensively in the user_package, worksheet_maker, equation_builders and equation_entities. However, there are cases where documentation is given for methods whose function is clearly given in the name. This is particularly prevalent in the user_interface package.

Testing

For all classes in the user_package (*excluding LocalDataAccess*) equation_builders, equation_entities, a worksheet_maker, notice that almost 100% of lines are covered by test cases. From the user_package, LocalDataAccess has fewer test coverage, as its functionality is tested in UserController (since all of UserController commands requiring that it saves the results immediately and is able to retrieve it upon fresh instantiation of a UserController).

Note that for all the equation builders, we decided to only have one test case for the wholeBedmasDirector. This is because the Builders themselves are all called by the director and that is the only way they will produce testable values.

Lastly, notice that there are no tests for the user_interface package and only a few tests for the PDF generation; this is because they are following the Humble Object Pattern. These classes have been made to have all their logic not within them, but rather in the classes they depend on and these are the classes we test (ex. worksheetGenerator has test cases for if the equations are made correctly). From there, we assume the presentation aspect works correctly and can be easily confirmed by a visual inspection.

Refactoring

Previously, the User object had the responsibility to store a history of worksheets generated. We decided to refactor much of the code in User, UserManagerViewer, and UserManagerUpdater to create a History and HistoryManager class to handle user history functionality. More details are available in this pull request here (LARGE UPDATES to userPackage · CSC207-UofT/course-project-matrix-1@e5c07c9 (github.com)).

Regarding the worksheet generation, the code was originally in the same package, did not use a facade design pattern, and did not use a builder design pattern. In order to complete this large refactor, Will and Sean made these in the sean_will_2 branch and the pull request was accepted by Stan. https://github.com/CSC207-UofT/course-project-matrix-1/pull/15.

One code smell that we all needed to decide on how to fix together was preventing data clumps and long parameters for the equation and format details. These parameters were used heavily by all three aspects of the program (UI collects these parameters, worksheet/PDF generation needs these to work, and the user's worksheet history must store them). In order to solve this, we created 2 hashmaps of <String, object>: equationDetails and formatDetails. We created a small document that planned out the exact key value strings to make sure it was consistent across all group members.

User Interface Diagram