Matrix: Design Document

Updated specification + Functionality

The primary goal of our worksheet generator is to create highly customizable math worksheets. Specifically, the user can choose the operator (addition, subtraction, multiplication, division, exponentiation, LCM, and GCD), number type (whole number or fraction), question format (horizontal, vertical, division bracket), title, number of questions, and number of rows/columns per page. Not all of these choices are interchangeable (**Compatible with one another**?); for instance, the user cannot select a subtraction problem with the division bracket question format. If this occurs, the UI will prompt the user that the input is invalid. Once they select these details, the user will be directed to another screen with parameters for equation generation (ex. operand ranges). The available parameters depend on the operator selected. Finally, after generating the worksheet, the user will be able to preview their PDF and download it to a path of their choice.

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 can be identical to the first if desired.

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 coalesce into Matrix.

Code Organization + Packaging

From the start, we noticed there were three distinct features in our project: (1) user functionality, (2) worksheet generation, and (3) the graphical user interface. We decided to package our code by components within these overarching categories. A component refers to a combination of business and data retrieval logic that can be used together. This essentially means that each component should have one public interface and package specific implementation details.

In the case of worksheet generation, this code was split into 3 distinct packages: equation_builder, equation _entities, and worksheet_maker. Similarly, user functionality was packaged as user_package and the GUI was packaged into a user_interface. Each of these packages followed the definition of a component in that they had a single interface class and encapsulated implementation classes. For example, in the equation_builders package our interface was the StandardEquationDirector while everything else was package private.

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

While this worked well for Phase 1, as our project evolved we began adding packages that were not components. This includes the exceptions, utilities, equation_parameters, and constants packages. At first, these classes were embedded in a single package. However, this was problematic since these classes were used outside their assigned package. For instance, the Randomizer utility is used across the worksheet_maker and equation_builders packages. As a result, we decided to group these shared classes into separate packages. Given we are a smaller development team, creating these shared packages seems to be the logical choice. However, if this project were to proceed with more developers, these packages would likely need to change with new boundaries to avoid compromising effective communication.

SOLID

SRP

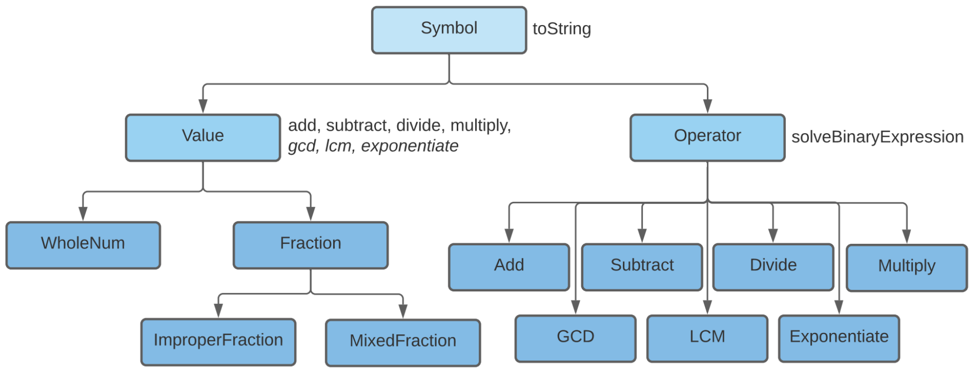
Within the user_package, the entities User and History show ownership of distinct and single responsibilities: storing user data and storing worksheet generation history respectively.

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 Equations as an expression tree of class Symbol that can be solved using a .solve() method. Symbol is an interface implemented by two abstract classes, Value and Operator. Value is an abstract class that has abstract methods for several operations (add, subtract, multiply, divide, exponentiate, LCM, GCD). Value itself is subclassed into WholeNum and Fraction which provide implementations for each of these operations (when mathematically possible). Value and its subclasses demonstrate OCP since we can add new features, like Fractions (or in the future, Decimals), by extending Value and implementing its abstract operation methods without modifying the Value class.

Following the OCP allowed for a great degree of polymorphism, enabling us to use the same method to solve Equations or represent them as a String regardless of the combination of Values and Operators used.



LSP

LSP states that for any inheritance relationship (ex. Child extends Parent), an instance of Child can replace the Parent, or in other words subclasses can only add more behaviour (not remove or modify it). This is easy to follow for most relationships. For example, in the FractionOperandConstuctor lineage (extended by FractionAddSubOperandConstuctor, which is extended by FractionAddOperandConstuctor), all of the methods were directly implemented without removing or modifying them. However, one case where we violate the LSP can be seen in the Fraction (child) and Value (parent) relationship. In the Fraction class, we throw an IllegalOperatorForOperandTypeException for the exponentiation, LCM, and GCD abstract methods in Value, modifying the original behaviour of the function. We had to do this since LCM and GCD for fractions are not mathematically possible and fractional exponents, while possible, would be overly challenging. Thus, violating LSP seemed to be reasonable in this case.

ISP

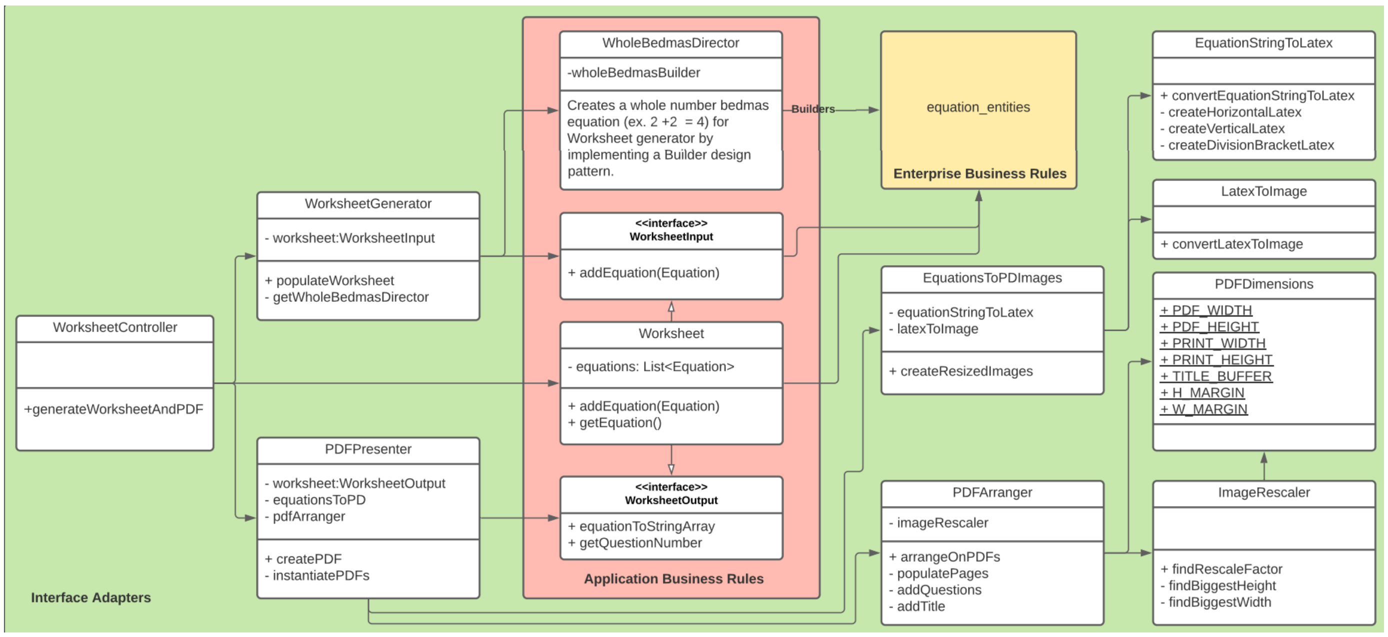
One notable example of ISP can be observed in our Worksheet class and its dependents. The Worksheet class (which is essentially a list of Equations) is accessed by both WorksheetGenerator (which adds Equations to the worksheet) and PDFPresenter (which uses the Equations and formatting details to make a PDF). To accomplish this, the Worksheet class needs getters and setters. However the former is only used by PDFPresenter while the latter is used by WorksheetGenerator. To ensure these classes were not forced to depend on Worksheet methods they don't use, we created two interfaces, WorksheetInput and WorksheetOutput, that were implemented by Worksheet (UML diagram below).

DIP:

One requirement of the DIP is to ensure that low level components depend on high level components, not the other way around. Our code consistently adheres to this by establishing and enforcing the layers of clean architecture (UML diagram below). The second condition of DIP however is more difficult to satisfy, as it states that every class should depend on abstractions. This is certainly true for many of our classes, particularly those in the equation_entities package. However, our group agreed that enforcing this rule everywhere would sacrifice readability and sustainability. Therefore, we limited this aspect of the DIP to only the sections of code that needed it the most.

Clean architecture

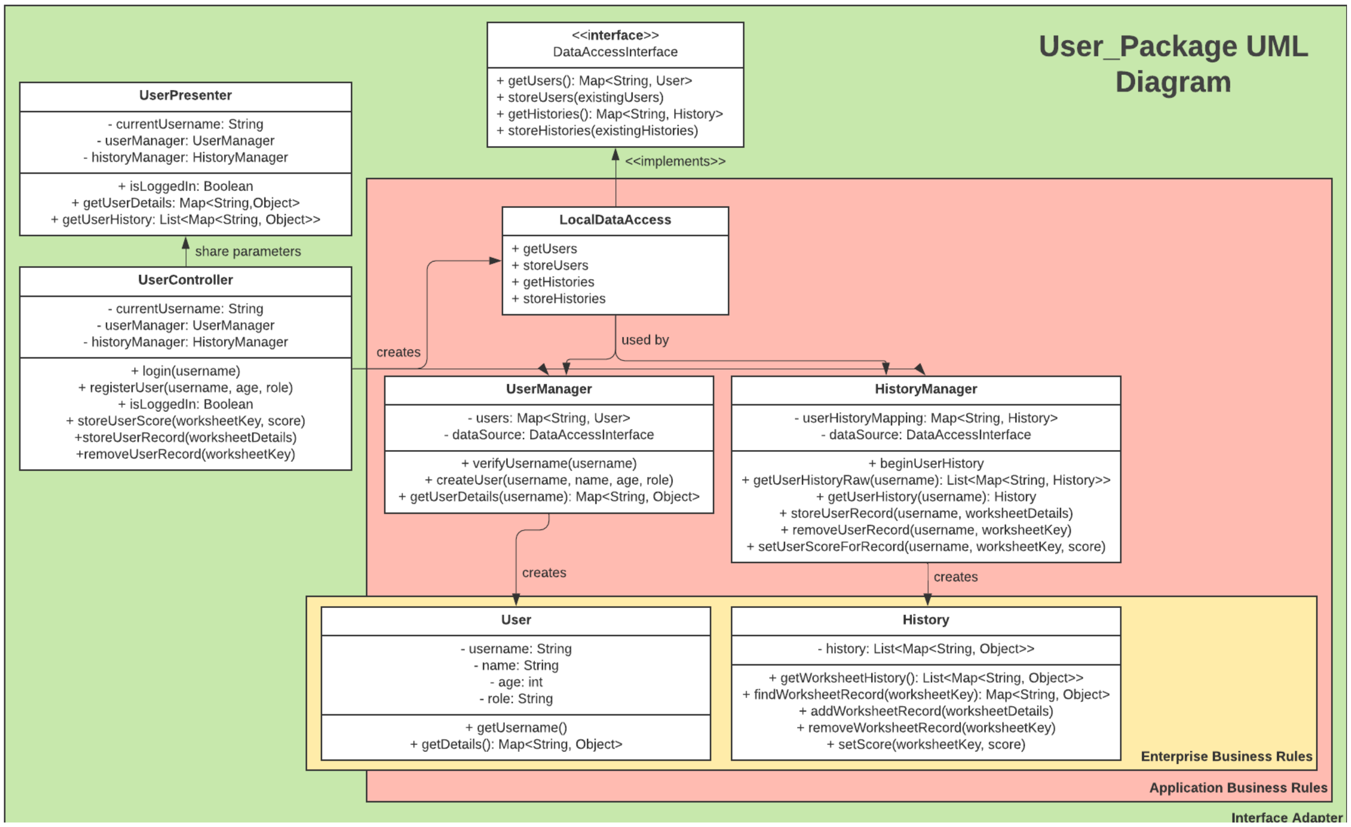
Worksheet_Maker UML Diagram



Worksheet_Maker Scenario Walkthrough

Various equation details and formatting details are passed into the WorksheetController. The WorksheetController first creates an empty Worksheet class, and instantiates a WorksheetGenerator and a PDFPresenter with the same instance of Worksheet. The WorksheetController then tells the WorksheetGenerator to create the Equations using the equation details (ex. operators, operand ranges, number of equations etc.) provided. In turn, WorksheetGenerator calls on the StandardEquationDirector, which uses a Strategy design pattern, to create a StandardEquation that is saved in Worksheet. To convert the Worksheet into a PDF, WorksheetController sends PDFPresenter various formatting details (ex. number of rows/columns, type of questions). PDFPresenter receives the Worksheet as an array of Strings (to prevent undesirable dependencies) and converts each equation into a PDImage using EquationsToPDImages (which calls on 2 helper classes). PDFPresenter then arranges these PDImages onto a newly created PDF using PDFArranger. Finally, PDFArranger uses various PDFDimension constants and the imageRescaler method to optimally rescale the images to fit onto the PDF.

User_package UML Diagram


User_Package UML Diagram
<<interface>> DataAccessInterface
+ getUsers(): Map<String, User>
+ storeUsers(existingUsers)
+ getHistories(): Map<String, History>
+ storeHistories(existingHistories)
<<implements>>
UserPresenter
- currentUsername: String
- userManager: UserManager
- historyManager: HistoryManager
+ isLoggedIn: Boolean
+ getUserDetails: Map<String,Object>
+ getUserHistory: List<Map<String, Object>>
share parameters
UserController
- currentUsername: String
- userManager: UserManager
- historyManager: HistoryManager
+ login(username)
+ registerUser(username, age, role)
+ isLoggedIn: Boolean
+ storeUserScore(worksheetKey, score)
+storeUserRecord(worksheetDetails)
+removeUserRecord(worksheetKey)
creates
LocalDataAccess
+ getUsers
+ storeUsers
+ getHistories
+ storeHistories
used by
UserManager
- users: Map<String, User>
- dataSource: DataAccessInterface
+ verifyUsername(username)
+ createUser(username, name, age, role)
+ getUserDetails(username): Map<String, Object>
HistoryManager
- userHistoryMapping: Map<String, History>
- dataSource: DataAccessInterface
+ beginUserHistory
+ getUserHistoryRaw(username): List<Map<String, History>>
+ getUserHistory(username): History
+ storeUserRecord(username, worksheetDetails)
+ removeUserRecord(username, worksheetKey)
+ setUserScoreForRecord(username, worksheetKey, score)
creates
creates
User
- username: String
- name: String
- age: int
- role: String
+ getUsername()
+ getDetails(): Map<String, Object>
History
- history: List<Map<String, Object>>
+ getWorksheetHistory(): List<Map<String, Object>>
+ findWorksheetRecord(worksheetKey): Map<String, Object>
+ addWorksheetRecord(worksheetDetails)
+ removeWorksheetRecord(worksheetKey)
+ setScore(worksheetKey, score)
Enterprise Business Rules
Application Business Rules
Interface Adapter

User_package Scenario Walkthrough

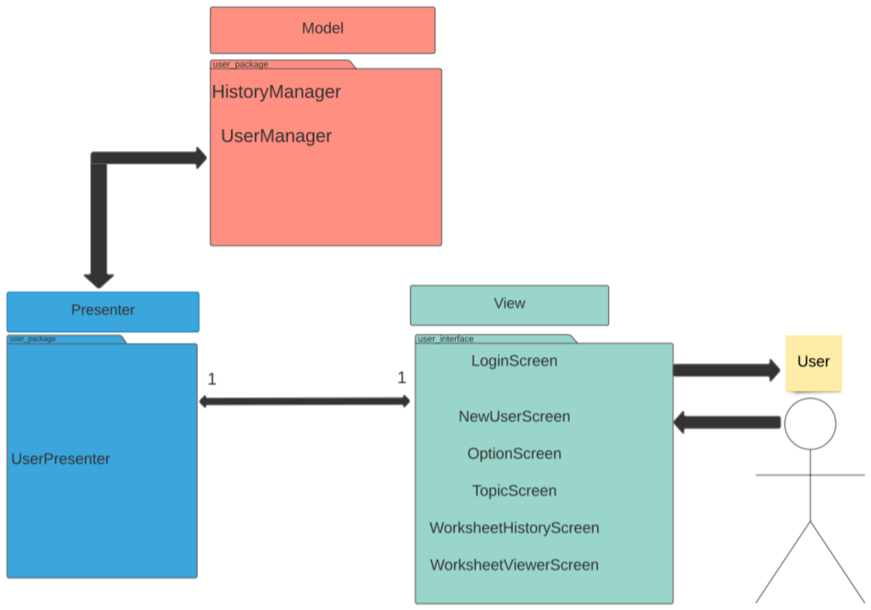
The Dependency Rule is followed in (1) the user_package and (2) its interaction with the UI. Inside the package, we invert the dependency between data retrieval/storage and its usage in UserManager and HistoryManager. The DataAccessInterface interface specifies methods to retrieve/store user or history 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.

Model View Presenter UML diagram



The View layer consists of the classes named: LoginScreen, NewUserScreen, OptionScreen, TopicScreen WorksheetHistoryScreen, WorksheetViewerScreen. These classes provide the view that the user can interact with. As soon as the user makes any request, these screen classes immediately call the presenter. The presenter is the UserPresenter. UserPresenter delivers the requests to the model. The model consists of UserManager and HistoryManager. These classes handle the requests that the presenter delivers to them and then the output is parsed by the UserPresenter. UserPresenter sends the output information back to the view classes so that the user can see the result.

Design Patterns

Strategy design pattern for equation generation

Initially, we used the builder design pattern to generate BedmasEquations, mainly because it adhered to the SRP by separating the generation of various equations from the equations themselves. However, we noticed that redundant code began to surface between Builder subclasses (for addition, subtraction, multiplication, ... for whole numbers or fractions). Furthermore, it occurred to us that operand generation was the only main component that differed between Builders. To address this, we implemented the Strategy design pattern, such that Builders became OperandConstructors and overlapping code (not needed for operand creation) got absorbed into one main StandardEquationMaker. Moreover, specific operand creation (e.g. for integer multiplication) was delegated to OperandConstructors. Now, the StandardEquationMaker is responsible for calling the right OperandConstructor, and the OperandConstructor will return the first and second operand. All the while, StandardEquationMaker is unaware of how each OperandConstructor creates the two operands.

Facade design for worksheet_maker

We implemented the Facade design pattern to simplify how our UI interacts with the worksheet_maker package. The goal of the UI is to specify the parameters to create a worksheet and retrieve its PDF representation. However, implementing this at the UI level would require a series of complex method calls that would add to the UI's numerous responsibilities. Instead, we structured our code to have UI interact with a WorksheetController facade. WorksheetController then delegates roles to a WorksheetGenerator and PDFPresenter, to ultimately produce a PDF that the UI can interact with.

Facade design for EquationsToPDImages

Converting text equations to a fully formatted PDF is a long process with numerous responsibilities. While the PDFPresenter itself is part of the WorksheetController Facade, implementing these responsibilities in PDFPresenter directly would still be problematic. To fix this, we implemented another two 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 about the implementations of these classes and is simply responsible for delegating roles. The second Facade design pattern is within the EquationsToPDImages class. Ultimately, this class must convert a string of equations into a latex intermediate and finally PDImages. Given these are separate responsibilities, EquationsToPDImages instead calls EquationStringsToLatex to create the latex intermediate and then LatexToImage to convert this latex intermediate into a PDImage.

Facade design for user_package

The UserController (and UserPresenter) implement a Facade design pattern. It instantiates a DataAccessInterface that implements the 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

Similar to Phase 1, we have branches from main for worksheet-related, user-related or UI-related features. From these branches, we create sub-branches to develop features. After verifying functionality, we send pull requests to merge sub-branches and their respective non-main branches. Not long after, we send pull requests from these non-main branches to the main branch. We ensure that pull requests were reviewed and merged by at least one other person. Finally, we automated testing using Apache Ant such that all JUnit tests were compiled and run whenever a commit/merge was done on the main branch. Ultimately, these checks ensured the main branch was always functional.

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 nearly all classes. However, there are cases where no 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, 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).

For the equation_builders package, we created a single test file for the StandardEquationDirector. This file is used to test other classes in the package since these classes can only be accessed through StandardEquationDirector and are otherwise package-private or protected.

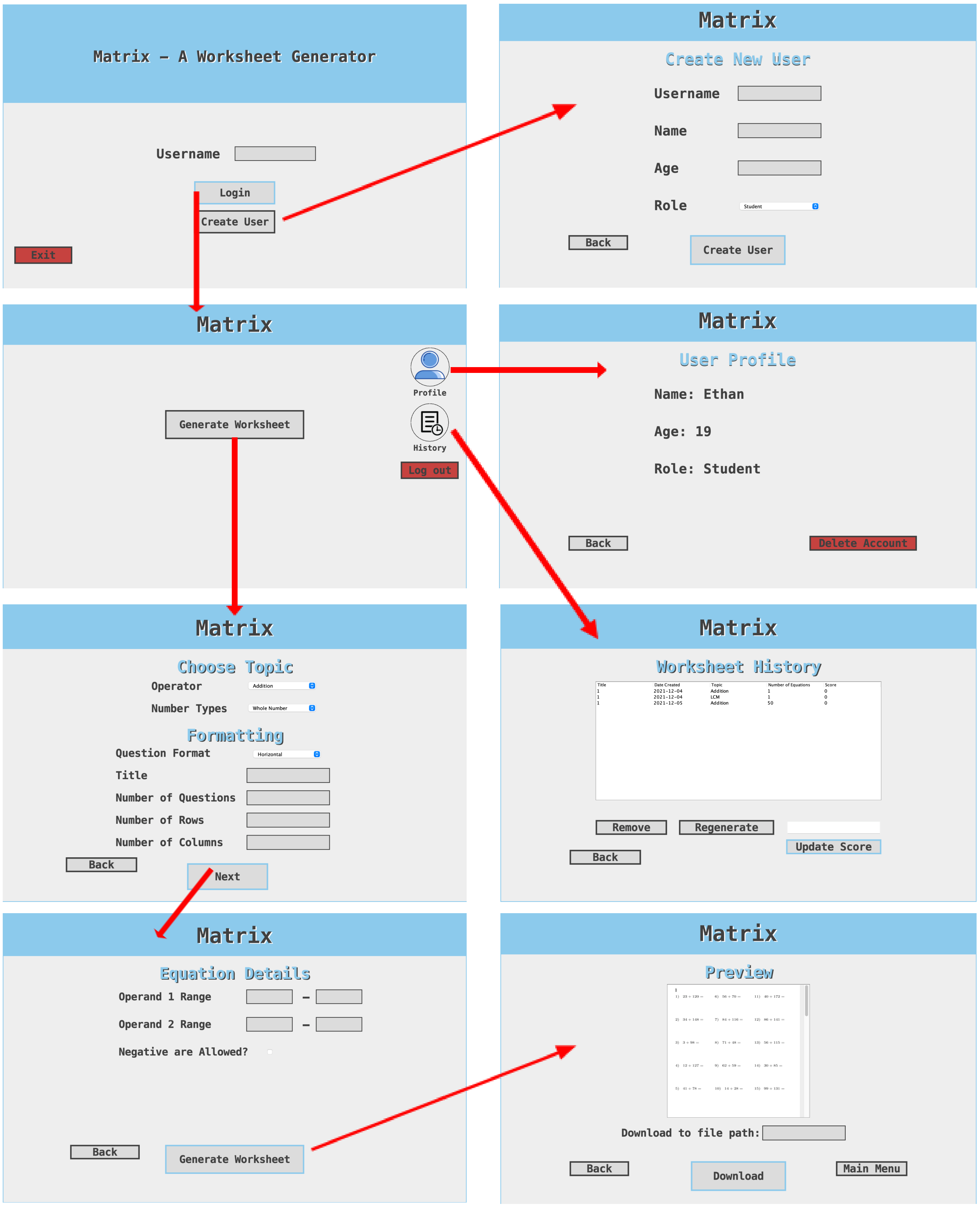
Lastly, notice that there are no tests for the user_interface package and only a few tests for the PDF generation. This is because these classes are following the Humble Object Pattern. Specifically, the logic of these classes is bound to their dependents rather than themselves. These dependents are extensively tested (ex. WorksheetGenerator has test cases to determine if the equations are formatted correctly). From there, we assume the presentation aspect works correctly and this can be easily confirmed by a visual inspection.

Refactoring

Refer to progress reports for refactors and references to specific pull requests.

One code smell that we still have is the data class code smell in EquationDetails and others in the equation_parameter package. Originally these classes were HashMaps storing all the input from the user on the 'generate worksheet' screens to ultimately be passed into WorksheetController, the facade class that generates the PDF from those parameters. These parameters include, operators, number of questions, whether negatives are allowed, how many columns and rows are on the screen and many more. As we added fractions we realized that the worksheet could have slightly different parameters (for example, we want the user to be able to choose the denominator range of the answer). While using the same hashmap could have worked, it would be very confusing trying to keep track which parameters were assigned and it's a hassle to type in the exact key needed to access that value. A class to store the parameter was much more convenient to use and we could use it polymorphically (ex. EquationDetails is subclassed by WholeNumEquationDetails, FractionAddSubEquationDetails, FractionMultiDivEquationDetails, but we passed it around as EquationDetails). However, all these classes only have getters and setters, thus they are data classes. Originally, we thought that we might be able to counter this later on by moving some utility functions such as randomization or factor finding into these classes, however, due to their ubiquity it felt more appropriate to create the utilities into their own classes (in the utilities package). While this code smell exists, in our opinion, it's at least better than the hashmap option and one could argue that being able to serialize them (which is how we store worksheet history) is an additional function.

User Interface Diagram



Title	Date Created	Topic	Number of Equations	Score
1	2021-12-04	Addition	1	0
1	2021-12-04	LCM	1	0
1	2021-12-05	Addition	50	0

1

1) $23 + 120 =$ 6) $56 + 70 =$ 11) $40 + 172 =$

2) $34 + 148 =$ 7) $84 + 116 =$ 12) $86 + 141 =$

3) $3 + 98 =$ 8) $71 + 48 =$ 13) $56 + 115 =$

4) $12 + 127 =$ 9) $62 + 59 =$ 14) $30 + 85 =$

5) $41 + 78 =$ 10) $14 + 28 =$ 15) $99 + 131 =$