Design

CS 3704: Intermediate Software Design

Prepared by **Epic Fellows:**

Caroline Joseph
Mason Gelletly
Quinn Anderson
Tatiana Monteiro

11/09/2023

High-level Design

We will be using **a combination of Client-Server** to handle the distributed nature of our application **and a Model-View-Controller (MVC)** to structure the application's internal design. This will allow our system to balance scalability, maintainability, and a clear separation of crises, which are critical for a robust and efficient team-building tool like our Pair-Programming Matcher

Client-Server

Our system requires a robust backend to handle the complex logic of matching engineers, processing feedback, managing user profiles, and more. A client-server architecture will help us achieve this separation, where we have the server handle the data processing and storage, while the client will provide an interactive and user-friendly interface. This architecture will make it easier to achieve scalability because the server can handle multiple requests from many clients at the same time. This is important as we wanted our tool to be used by both companies and universities, which means it will need to potentially support a large number of concurrent users.

Model-View-Controller (MVC)

This architecture will make it easier to manage and update the application by dividing it into three interconnected components. This will allow us to develop well-structured and maintainable code, as well as parallel development of the user interface and the business logic.
Controller: This layer will handle user requests and update the Model.
Model: Here is where our data and logic will be handled. This layer will manage our user profiles, skill sets, workload, project involvement, and things they've collaborated on.
View: Here is where the UI/UX interaction will be handled. It will represent what the users interact with such as how to input skills, and preferences and give feedback. Updated by the model.

Low-level Design

Behavioral Design Family

The Behavioral design pattern family would be helpful for the implementation of this project. This is because the focus on how classes interact with each other as well as the general responsibility of classes serve as large, crucial aspects of the application’s flow. A great deal of the workload will be focused on these same challenges that the Behavioral pattern family addresses. We could organize our classes in such a way that the behavior of the application is straightforward by simply representing users with one class, and the searching/matching algorithms in another- this would greatly simplify the readability and maintenance of the codebase. With this in place, we would need only a driver who would leverage both classes to accomplish our overall goal. Additionally, this would create a direct logic route for our general

flow, which aligns with the chain of responsibility attribute of the Behavioral design pattern family.

Code/Pseudocode Representation of the Design

matcher.py

```
from user import User # Import in our User class

# Class representing the matching algorithm for our application's use. Responsibilities include registering users and returning matches.
class Matcher:
    # Initialize our list of users
    def __init__(self):
        self.users = [ ]

    # Register a user to the end of the matcher
    def register_user(self, user: User):
        self.users.append(user)

   # Find a match for some user
    def find_match(self, user: User):
        for potential_match in self.users: # Parse through users checking for match
            if potential_match.skill_level >= user.skill_level and potential_match != user:
                return potential_match
        return None
```

user.py

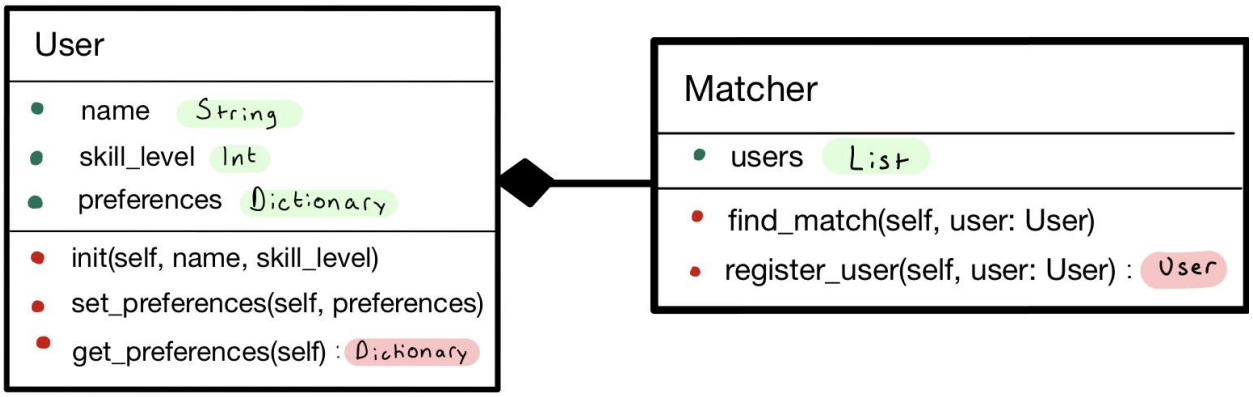
```
# Class representing users of the application
class User:
    # Initialize our basic information
    def __init__(self, name, skill_level):
        self.name = name
        self.skill_level = skill_level
        self.preferences = { }

    # Set preferences
    def set_preferences(self, preferences):
        self.preferences = preferences

   # Retrieve preferences
    def get_preferences(self):
        return self.preferences
```

driver.py

```
from user import User # Import in User
from matcher import Matcher # Import in Matcher

# main() will leverage the User and Matcher classes to satisfy pairing queries
def main():
  matcher = Matcher()

  … # Users are registered, preferences are set via interaction with UI
  … # A request comes in to find a match

  match = matcher.find_match(searching_user) # use Matcher for searching functionality

  if match: # Basic logic presents final output based on if a match was found
      print(f"{searching_user.name} matched with {match.name}.")
   else:
      print("No match found for {searching_user.name}.")
```

Note that this is **not comprehensive of all of our application's described functionalities. This is a ground-level implementation**

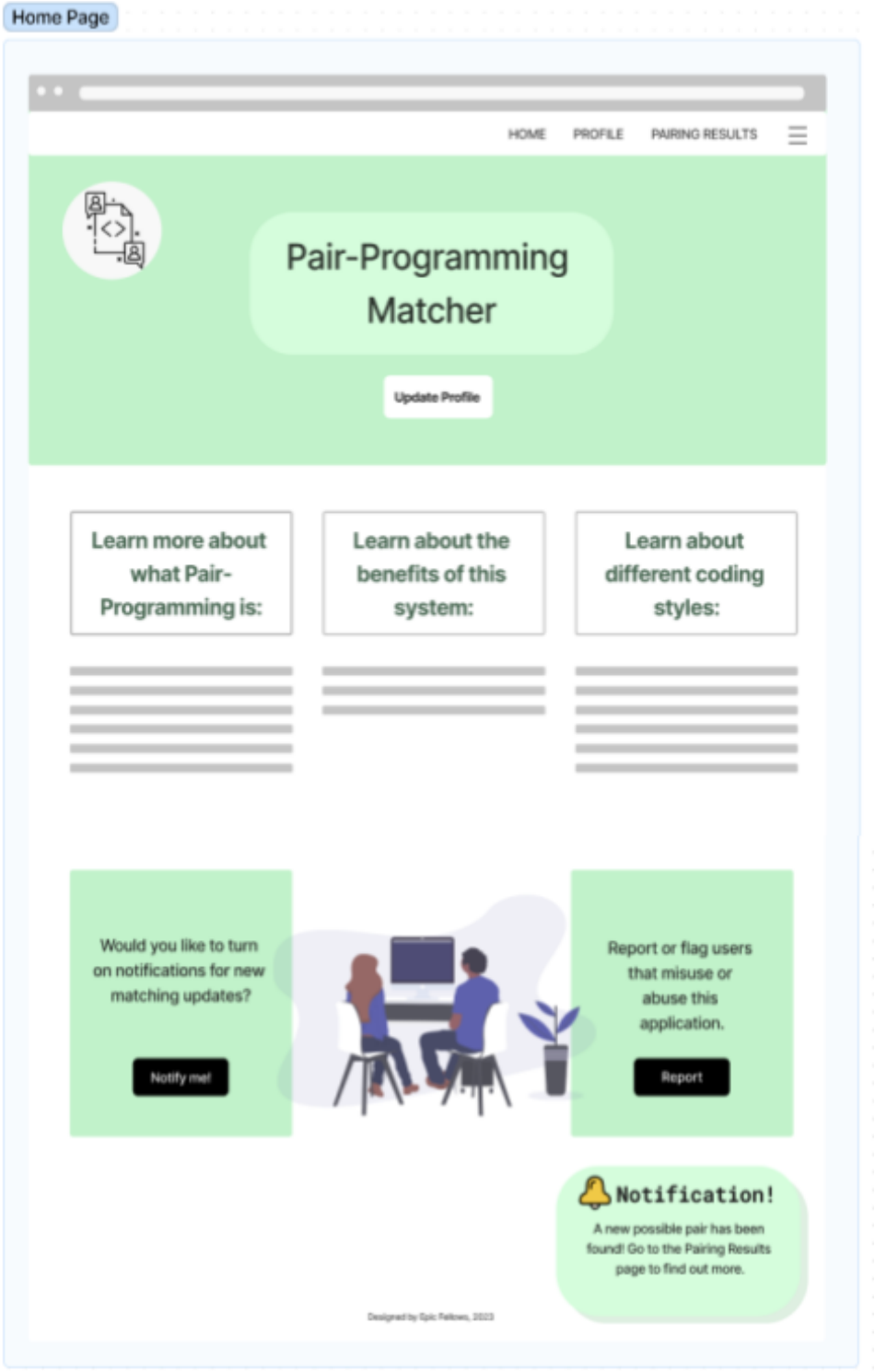
Informal Class Diagram:



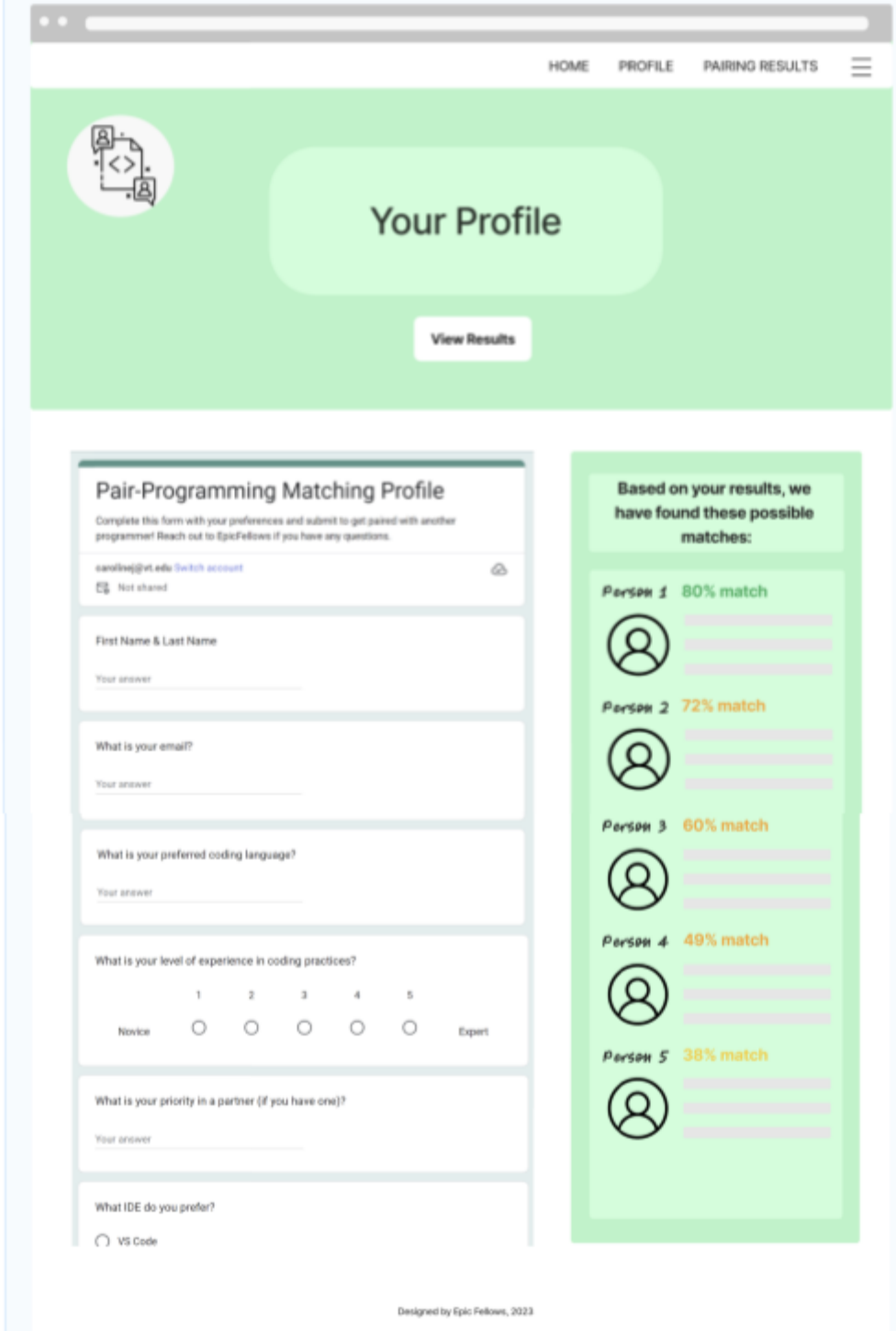
Notice the **aggregation** between the Matcher and User classes. This class setup requires the driver.py script provided earlier to function as intended.

Design Sketch

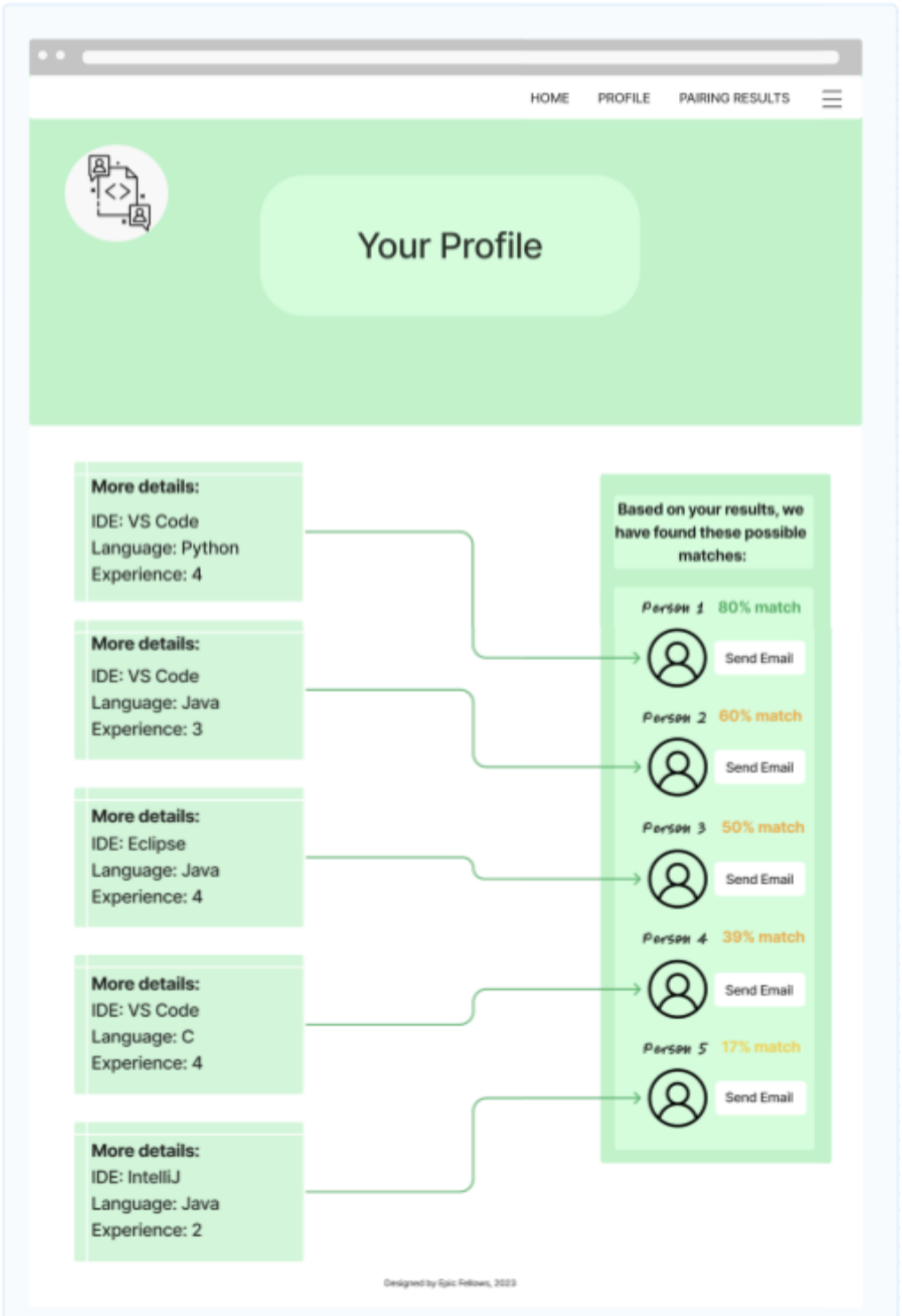
Home Page



Profile Page



Pairing Results Page



The Home page includes details and links about Pair Programming so users can learn more about what it is and how it can benefit them. Furthermore, this page has buttons for turning on notifications of matches and also to report users who are misusing the system. It also shows an example of a notification from the website, which notifies users of possible new matches for them to consider. At the top of the page are links to other pages of the website, which are the Profile page and the Pairing Results page. The Profile Page includes a built-in Google form that the user can fill in and submit to have an opportunity to find a pair. After the form has been submitted, results will be displayed in the section next to the form. Furthermore, the results show the percentage of compatibility for each pair. The Pairing Results page shows more details for each pairing, including their preferred language, IDE, and experience level. Each of the matchings has a “Send Email” button so the user can directly reach out to a programmer of their liking to begin pair programming.

Process Deliverable

Waterfall: submit supplementary planning documentation

Following the work we have done for PM3 we can review our last round of planning documentation and show updates or revisions, along with displaying some of the work and planning that we have completed. During our last set of planning documentation, we focused on the cost, assumptions, risks, and success metrics of our potential product. Overall, when completing the low-level and high-level design we can have more insight into these factors about our product.

When focusing on cost, it is important to still understand our cost is very dynamic as that algorithm for matching and software to implement it can be started to work on just by us in the team as we are currently doing. Also, we can continue to keep planning it at our current cost as well. What changed for us for cost is we feel with more insights on our design we can keep costs lower for now. We feel that by using free programs like Figma to continue to design, we can keep operating at our own cost. The other aspect to consider for current costs is the matching algorithm itself. We can most likely use free open-source machine learning software to match people, but this we as a team are feeling can be more difficult. This could have a cost of more time to figure out or even a cost to hire or invest money into programs that could help with this. The same cost will still hold for the future; As we continue to grow and when we get to the part where we want to have our application available to be used nationally, we will have to invest in advertisement, more hires, and the infrastructure to make sure it works at a bigger scale.

Our assumptions have also been more clear. Right now we are assuming we use a combination of Client-Server and a Model-View-Controller (MVC). We hope that in regard to our stakeholders, the Client-Server high-level architecture will allow multiple clients to use our product while our server supports that. This is important as we want the people using our app to be supported in doing so regardless of other users. Furthermore, we assume that our server will

be able to support multiple requests at the same time, and linking back to cost we need to ensure we invest in a good and efficient server to support the high-level architecture we chose. Focusing on our risk and success metrics, we now believe these are linked. Again our product will be on the path to success if stakeholders want our application. Choosing a low- and high-level design for our product also incurs some risk if either design fails, if stakeholders dislike the design, or if it incurs too much cost. That being said this also affects the success metrics as continuing to narrow down the specification like our design is important in our ability to succeed. In the future, we want to define some more success metrics for our group as well.

The last plan we want to talk about today in our PM3 is the timeline. Our project has been developed throughout our school year and as the semester wraps up we want to discuss the project in the grand scope of things. We are happy so far with our planning and completing the low- and high-level design is good, we are looking forward to the next month. We know that for costs we will not incur any in the next month, and then after that, we can decide on more costs. With costs we then also have risk. Our plan is to then be at a good breakpoint at the end of the semester so that we have the ability to break and potentially come back to it in an effective manner.

Overall this highlights our supplementary planning for PM3.