Table of Content

1. Object-oriented Design Considerations

a) Don’t Repeat Yourself (DRY)

Our group knew the importance of not writing duplicate codes, this led us to make our blocks of codes into separate methods.

An example would be our Connection Manager class where it handles the database connection to MySQL.

```
public class ConnectionManager {

    public Object getConnection(String sql, ArrayList<Object> data,
boolean insertUpdateDelete) {
        String host = "localhost";
        int port = 3306;
        String dbName = "magnet";
        String dbURL = "jdbc:mysql://" + host + ":" + port + "/" +
 dbName + "?useSSL=false";

        String username = "root";
        String password = "";

        ResultSet rs = null;

        // step 1: Loads the JDBC driver
        try{
            Class.forName("com.mysql.jdbc.Driver").getConstructor().
newInstance();
        }
        catch (ClassNotFoundException | NoSuchMethodException |
InstantiationException | IllegalAccessException |
InvocationTargetException e){
            System.out.println(e);
            System.out.println("Driver not found.");
        }
```

A snippet of our ConnectionManager class

So for our DAOs, as shown in Image 1 shown below, we will call and initialize the Connection Manager class, in order to connect to the database and execute the query.

```
public ArrayList<String> getFriendRequests(String username){
    ConnectionManager cm = new ConnectionManager();
    ArrayList<Object> data = new ArrayList<>();
    ArrayList<String> result = new ArrayList<String>();
    String sql =
"select username from friends where friend_username = ? and status = 'pending'";

    data.add(username);

    ResultSet rs = (ResultSet)cm.getConnection(sql, data, false);
    try {
        while (rs.next()) {
            result.add((rs.getString("username")));
        }
    } catch (SQLException e) {
        System.out.println("There is an issue with MySQL");
    }

    return result;
}
```

Image 1: getFriendRequests() Method in FriendsDAO.java

Another example would be, in the Farmer's Game, whenever crops wither, we have to clear the crops. Images 2, 3, and 4 shows how we reuse the clearCrop() method in other various functions of the class.

```
// clear crop according to plotnumber if any crops are wilted
public void clearCrop(int plotNumber) {
    PlotDAO plotDAO = new PlotDAO();
    boolean clearing_success = plotDAO.clearPlot(member.getUsername
(), plotNumber);
    if (clearing_success) {
        plots.put(plotNumber, new ArrayList<Object>());
        MemberDAO memberDAO = new MemberDAO();
        memberDAO.updateGold(member, member.getGold() -5);
    } else {
        System.out.println("That plot does not exist.");
    }
}
```

Image 2: clearCrop() Method in FarmLand.java

```
case "C":
                    try{
                        plotNumber = Integer.parseInt(full_choice.
substring(1, full_choice.length()));
                        if (plots.size() ⩾ plotNumber) {
                            clearCrop(plotNumber);
                            break;
                        } else {
                            System.out.println(
"Please enter a plot you own!");
                            break;
                        }
                    }
```

Image 3: An example of reusing clearCrop() method

```
public void harvestCrop(boolean checkIfFriendWall) {
        //declaring variables to be used
        MemberDAO memberDAO = new MemberDAO();
        Set<Integer> keys = plots.keySet();
        HashMap<String, Integer> harvestCropsAndQuantity = new HashMap
<>();
        CropDAO cropDAO = new CropDAO();
        PlotDAO plotDAO = new PlotDAO();

        //Find out which crop is ready to be harvested
        for (Integer k : keys) {
            if(plots.get(k).size() ≠ 0){
                Timestamp timePlanted = (Timestamp) plots.get(k).get(1);
            String cropName = (String) plots.get(k).get(0);
            Crop crop = cropDAO.getCropDetails(cropName);
            Timestamp timestamp = new Timestamp(System
.currentTimeMillis());
            long diffInTime = (timestamp.getTime() - timePlanted.getTime
());
            if (2 * crop.getRipeTime() ≥ (double) diffInTime/60000 && (
double)diffInTime/60000 ≥ crop.getRipeTime()) {
                clearCrop(k);
```

Image 4: An example of reusing clearCrop() method

Our group tries to divide the code and logic into smaller reusable units and use that code to call whenever we want to. The benefit of DRY is in maintenance of code.

b) Encapsulation

Encapsulation is the principle of information hiding. It helps to create code that is loosely coupled. Because the details are hidden, it reduces the ability of other objects to directly modify an object's state and behaviour.

An example would be getter functions. In Image 5, the getter will return an integer. If we want to change the logic of getting yield, we can just change the logic within the getter function. So none of the calls to the function have to change and therefore the code is more maintainable.

```
// get yield according to crop randomly in range
public int getYield(){
    return (int)(Math.random() * ((maxYield - minYield) + 1)) +
 minYield;
}
```

Image 5: getYield() Method in Crop.java

Another feature of encapsulation deals with hiding the access to an object. To do this, we use access level modifiers which give or hide access to objects' states and behaviours.

Access level modifiers define two access levels (that we used):

1. **public** - any object can access the variable or function, at any time
2. **private** - only the object that contains the variable or function can access it

```java
public class Seed{
    private String username;
    private String cropName;
    private int quantity;

    public Seed(String username, String cropName,int
quantity){
        this.username = username;
        this.cropName = cropName;
        this.quantity = quantity;
    }

    public String getCropName(){
        return cropName;
    }

    public int getQuantity(){
        return quantity;
    }
}
```

Image 6: Seed Class

In our Seed class, shown on Image 6, we have 3 private attributes to store information about the Seed. The access modifier private makes these attributes inaccessible for other classes. If our group wants to get more information about the Seed, they are required to call the public methods.

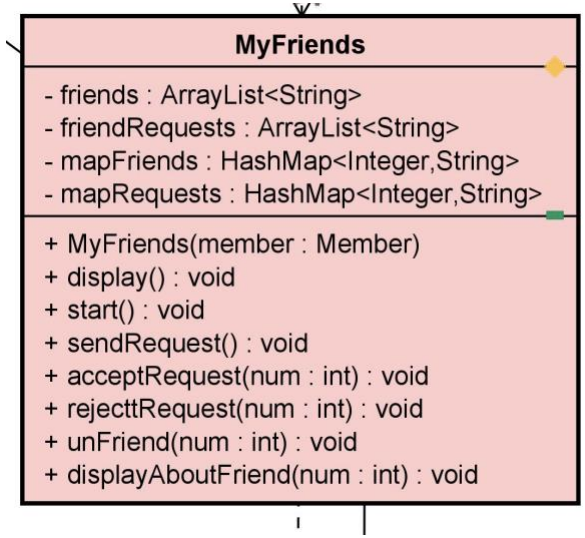
Encapsulation can help to create more maintainable code by helping to prevent the ripple effect of code changes. It also helps to create loosely coupled code by reducing access to an object's state and behaviour.

c) Single-Responsibility Principle (SRP)

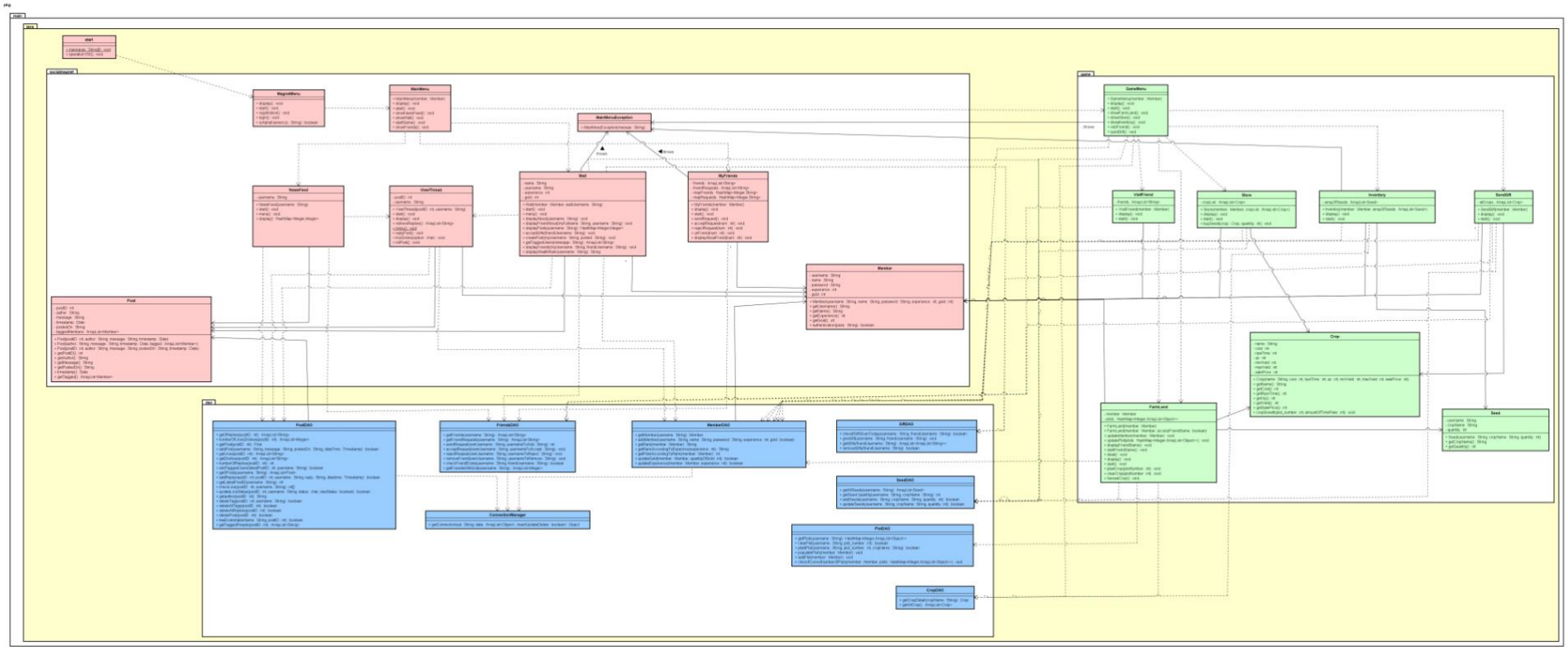
Our group focused on Single-Responsibility Principle (SRP) when developing the Social Magnet application. We focused on the cohesion of what the classes can do and how related or dependent the classes are towards each other.

We tried to maintain a high cohesion where the class is only focused on what it should be doing as it contains only methods to the intention of the class. We have separated classes that house different responsibilities as each responsibility is an axis of change.

An example would be our MyFriends class where it focuses on rejecting, unfriending, requesting, accepting and viewing of friends. In this context, MyFriends's methods will only be invoked when used to either view, accept, reject friends and etc.



2. Class Diagram



Sorry Prof, we could not make the diagram big enough so we attached it as a link. To have a better view of the Class Diagram, please view it here: https://smu.sg/OOPClassDiagram

3. Logical Diagram