# SOLID

We think our project is mostly consistent with the SOLID design principles. We will explain why it is the case from five different perspectives.

## Single responsibility principle (SRP)

we think our project follows SRP because every class created by us is responsible to one, and only one actor. Initially for each of the classes we have created such as User, Resource and Achievement, we have created managers for each of them. However, the downside of this solution is that we have too many managers that we need to keep track of which share similar functions. In this case, we then create a Facade pattern. It contains very little code but is responsible for instantiating and delegating to the classes with the functions add, delete and modify.

## Open-Closed Principle (OCP)

we think our project does not exactly follow OCP because we have the class that opens for extension but it is not dependent on another class that is closed for modification. However, we have some ideas of how to fix it if we had more time. The class PublishedContent is the class that is open for extension. In our project, the classes Post and Comment extend PublishedContent because they share the functionalities that are related to like and visible. However, they are different in terms of how they are initialised. In this case, we want to make a class Publish that is closed for modification. It could publish all the subclasses of PublishedContent and we could add more PublishedContent without changing anything to the class Publish. Therefore our project will follow OCP in future.

## Liskov Substitution Principle (LSP)

we think our project follows LSP because we have created many interfaces and superclasses that are implemented and extended by subtypes. One of the examples would still be PublishedContent. In our project, the classes Post and Comment extends PublishedContent. Whatever the method that needs a PublishedContent variable, it could be replaced by its subclass Post and Comment without any error.

## Interface Segregation Principle (ISP)

we think our project follows ISP because we try to keep our interface as small and specific as possible so that no class implements the method that is unnecessary. For example, instead of making one big interface with all functionality of PublishedContent, we have broken it down into Likable, Visible and Serializable and each interface is in charge of different methods.

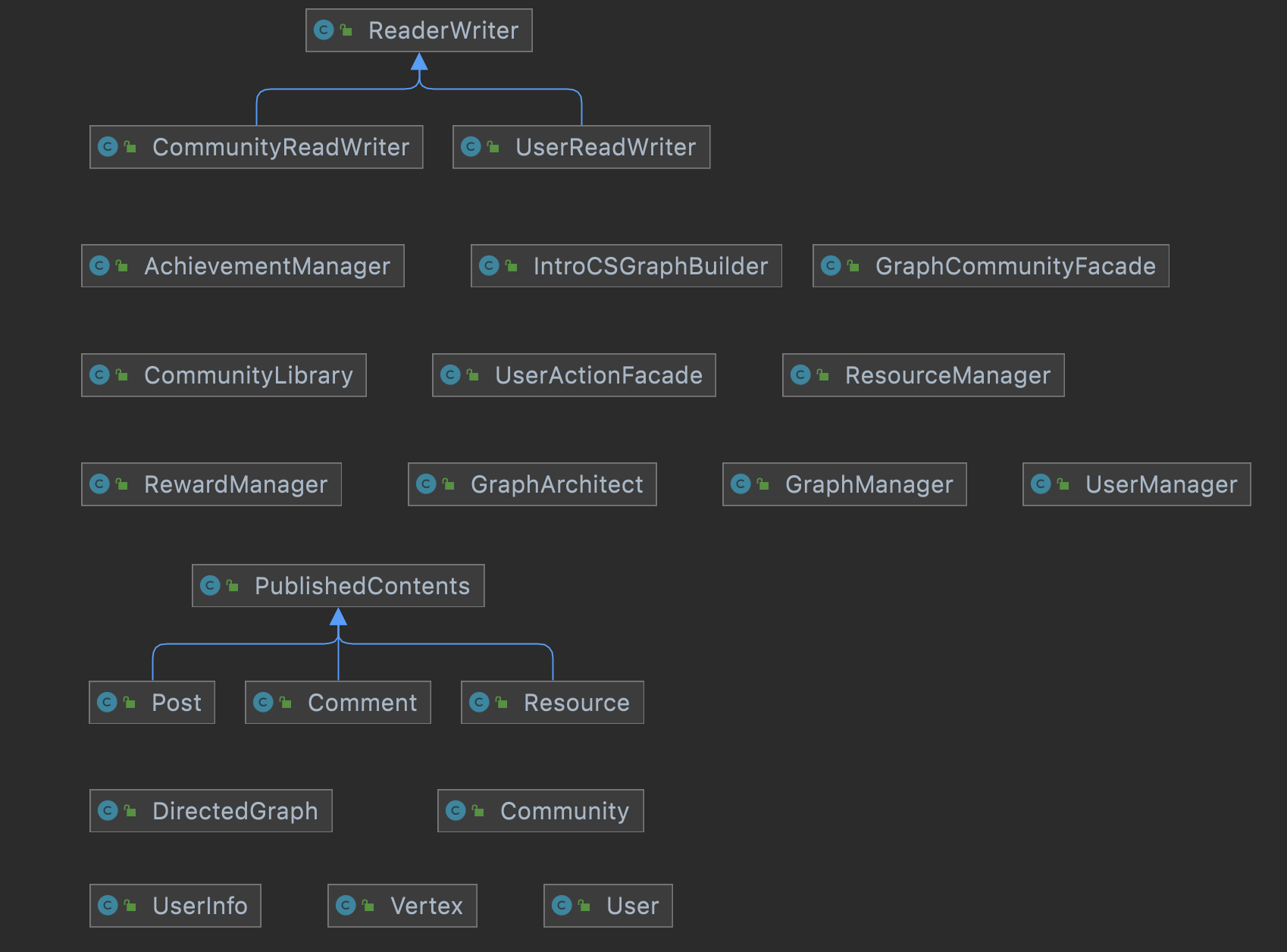
## Dependency Inversion Principle (DIP)

we think our project follows DIP because we have a class that has a variable type interface. In our project, our class GraphArchitect has only one variable graphBuilder of type interface GraphBuilder

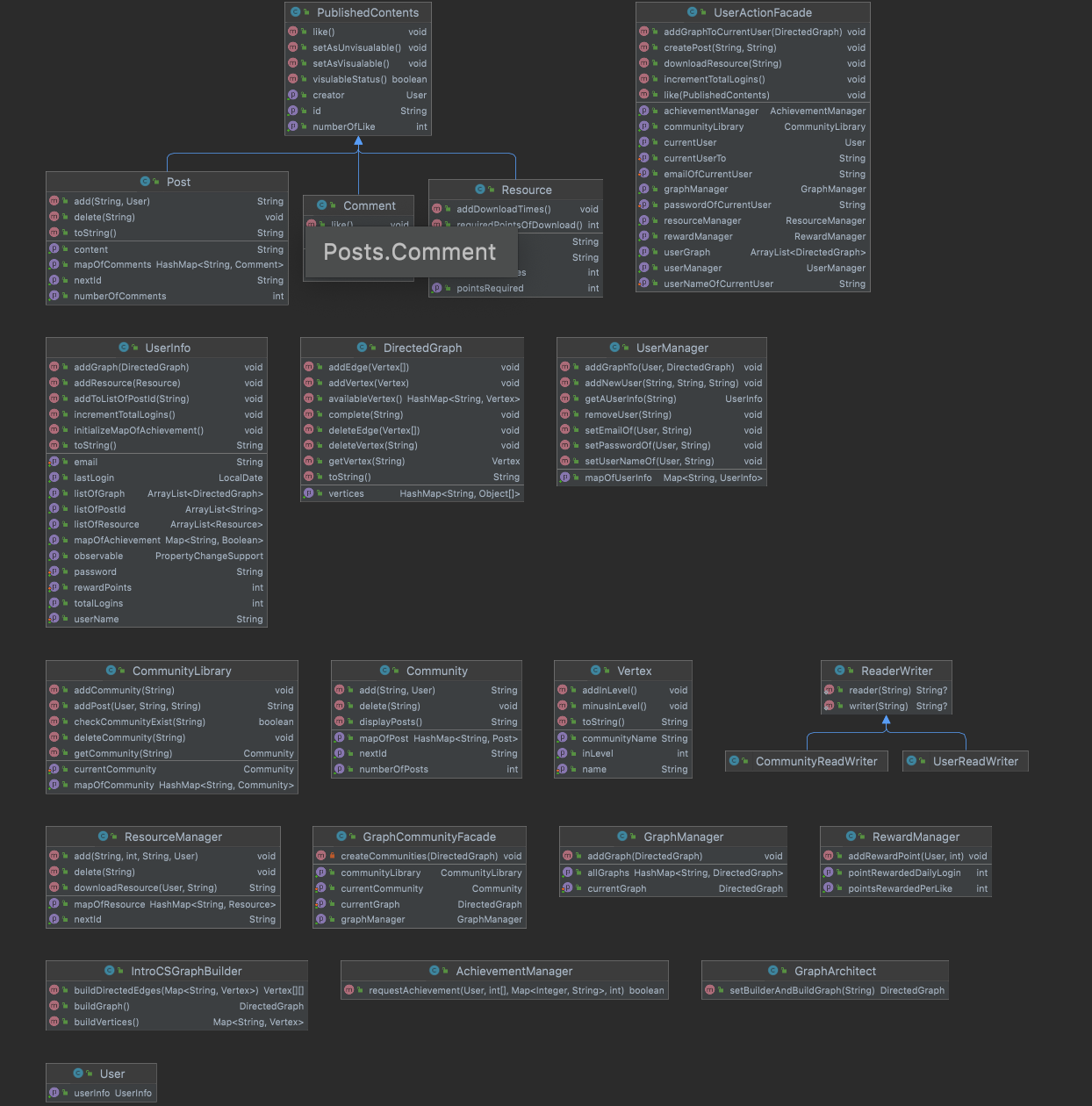
# Clean Architecture/Class Diagram

## UML diagram

(For simplicity I omitted the variables and the methods at here, but I attached the complete UML graph in the last)



## Complete Version of Class Diagram:



## 

## Scenario Walkthrough:

I will use a short scenario walkthrough to illustrate that our program here adheres to clean architecture.

Suppose that a user is using the tech tree for learning purposes and he encounters some problems and now wants to put up a question post in the community to discuss with other people in the community. He would simply input the title of the post and the content of the post then press post. A post will then be created in the community. In this process, the program would execute in this way: First, our program will use Facade, which is one of the use cases, to access the CommunityLibrary to find the intended community, this is the interaction between use cases. The CommunityLibrary would then find the specific community and access it. Now we have accessed an entity, a community, which follows the dependency rule. Then the add() method in the class community will take in the content and the User to create a post in this community. The interaction between community and User and community between posts are all interactions between entities. We haven’t violated any principle of clean architecture in this whole process.

Volations: Not any so far.

Is the Dependency Rule consistently followed when interacting with details in the outer layer?

* Give us a concrete example from something like your UI or an interaction with a database.

Given a scenario that a user now wants to login to his own account and this process involves the interaction between the database and our use case. I will demonstrate below that it follows the Dependency Rule:

When the account number and the password is read in from the user. Similar to the process of posting a post in the community. The Facade class, which is the use case, will handle the current user and interact with the userManager, which is also a use case. As the userManager takes in the current user, it will then interact with the database and verify whether the user has input the correct username and password.This is an interaction between use case and database. At last, the userManager will find this specific user and access it. This is the interaction between use case and entity. In this whole process. The Dependency Rule is not violated.

# Design patterns

| Builder Design Pattern | | |
| --- | --- | --- |
| Classes/Interfaces involved:  GraphArchitect (Class), GraphBuilder (Interface), IntroCSGraphBuilder (Class) | | |
| Using a builder design pattern to build built-in Technical Trees  IntroCSGraphBuilder is the interface that declares the methods every Technical Tree builder should have. IntroCSGraphBuilder implements the GraphBuilder interface and specializes in building the Technical Tree, “CS Introductory Series”. GraphArchitect is a class that controls the Technical Tree builders. It receives a Technical Tree order, uses the corresponding Technical Tree builder to build the tree, then outputs it. | | |
| Facade Design Pattern | | |
| Classes/Interfaces involved:  Facade (Class), UserManager (Class), RewardManager (Class), GraphManager (Class), CommunityLibrary (Class), ResourceManager (Class) | | |
| Using a facade design pattern to unify all of the manager classes.  Facade class connects the client to all of the manager classes (UserManager, RewardManager, GraphManager, CommunityLibrary, ResourceManager). By doing so, we don’t need to instantiate all of these managers, keep track of dependencies, etc. whenever we need to use their functions. | | |

# 

# 

# Use of Github Features

We used the Issues function to communicate some of the big issues we think need to be fixed in our code. Whenever we complete writing our code, we use the Pull Request function to create a new pull request with descriptions of the changes done to the code and notify the other members for review. Then, other members of the groups will go over the changes that are made in the new pull request and comment on anything they think is good/need improvement. Once some members have reviewed and approved the pull request, we will merge the changes to the main branch and everyone will pull the new version of code.

# Code Style and Documentation

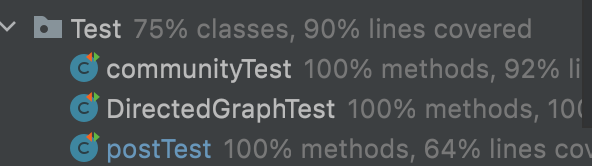
We included detailed descriptions of the changes we made to the code in our pull requests. We also reviewed and commented on the pull requests before merging them to the main branch.

We do have some warnings in IntelliJ about unused variables and methods. These variables and methods are created for functions that we will extend in the future. These warnings will be solved once we start extending the functions in Phase 2.

We also have javadocs for our methods, especially for the more complicated ones, that explain what they are, what they take in, and what we should expect from them.

# Testing

We have some tests that cover overall 75% of the method and 90% lines. In our test package, we have created four tests: communityTest, DirectedGraphTest, postTest and SerialisationTest. For communityTest, it tests add and delete post, number of posts, display post, and id of the post. For DirectedGraphTest, it tests the function of get, add and delete vertex, add and delete edge, and whether the tree functions well when we create a tree. For postTest, it tests add and delete posts.



# Refactoring

We have used refactoring functions to rename class, variable or method names that are not clear enough. We have also used refactoring to create different packages for the classes and interfaces that share the same functionality.

# Packaging strategies

We considered packaging by component and by layer. We eventually settled with packaging by component because it allows us to quickly locate a class that we are looking for. For example, if we are looking for the class AchievementManager, we know that it is most likely to be in the package, AchievementSystem.

# Updated Specification

## Resource System

In Phase 1, we have added the resource system for Users to download resources.At first, we only have limited resources for users, the users could create their own resource and share them. The resource is created with its description, the points needed to download and the contents. When a resource is created, the other users could use their reward points to download the resource.

There is a ResourceManager that gathers all the resources.

## Reward System

In Phase 1, we have added the reward system that will give Users reward points for completing certain requirements. Currently, whenever a User likes the post/resource/comment of another User, the creator of the post/resource/comment will receive reward points from this like. When a User completes an achievement, the reward system will also give the User reward points which the amount will vary depending on the achievement completed.

There is a RewardManager that is responsible for rewarding the points.

## Achievement System

In Phase 1, we have added the achievement system that has different types of achievements that the Users can complete. Currently, there are two types of achievements: number of posts (1st post, 5th post, 10th post,ect.) and number of total logins (1st login, ect.). Each achievement has its own threshold that needs to be satisfied in order to trigger the completion of the achievement. Each achievement also has a corresponding number of reward points that should be rewarded to the User when the User completes the achievement.

There is an AchievementManager that checks and changes the status of achievements

depending on whether their threshold has been met or not.

# Design decisions/Code Organization

We have two ways to implement user.achievementList. First one is to have a hashmap, and this hashmap has all achievements in our program, and the value for each one is True or False, which denotes that the user achieved or not achieved an achievement. For example, user.achievementList = {“achievement1”:True, “achievement2”:False}The other one is to have an arraylist for storing the achievement that the user achieved. For example, user.achievementList = {“achievement1”}. Both of them are describing the same thing. After discussing, we finally decided to use version1, because it would be easier for visualization of the achievement library of a user. The user will be able to look through all the achievements, no matter if they are achieved or not, so it will be easier to have True or False for each achievement to implement the UI.

We need to have a resource class that initialises resources. However, how to download that resource becomes a problem. At first we thought that it was enough to download the resource inside the Resource class. However, we have encountered a problem that the resource could not be stored inside the user library. In this case, we have created a class called resourceManager that keeps track of all the resources. The resourceManager could not noly add or delete any resources but also download the resource and add one download times whenever the resource is downloaded. Since the resourceManager keeps track of all the resources, in future we want to add a search method that searches the resource that contains a specific keyword. We would also like to add a sort method that arranges the resource according to the download times.

For the userinfo constructor, we decide to only take username, password and email as the parameter, since they are the most basic info for a user, and when creating a new user, these three parameters are useful. For an existing user, we decide to set their info when they log in by the setter and getter method in UserInfo class. This will make the program clear and easy to follow, and also reduce the workload and code length for creating a new user.

# Progress report

| Code | | |
| --- | --- | --- |
| 10/27  Alfred | Implemented the Builder Design Pattern to build built-in Technical Trees  Refactoring the program to resolve obvious violations of clean architecture  New classes/interfaces added:  GraphArchitect  GraphBuilder  IntroCSGraphBuilder  Edited:  UserManager  GraphManager  Added currentUser&currentGraph | |
| 11/3  Alfred | Created new Exceptions class under package, “constant”, to store the error messages  Refactored the “add” method in Post and Comment to reduce dependency  New classes/interfaces added:  Exceptions  Post and Comment used to be instantiated by classes in layers two levels outside of the entity. Relocating the instantiation of Post to the use case Community, and that of Comment to Post, solves this issue. | |
| 11/8  Alfred  Ashley  Arthur  Coco  David | Have a discussion for the implementation for the achievement system and serializable. Also have a discussion for how we can make the user class better by extracting userinfo to a new class. | |
| 11/9  Alfred  Ashley  Coco | Created new UserInfo class, to store user’s instance variables  Added achievements constants that represent some of the achievements we may have  Discussed about the implementation for reward system | |
| 11/10  Alfred  Tong Su | Had a discussion on how users can use reward points, and whether it is necessary to implement the “resource” class, to allow users to share study resources, such as links to youtube tutorials or private online inventories.  Implemented the achievement system.  Created Resource class, resourceManager class and HasResource interface. | |
| 11/11  Coco, Alfred | Discussed the need of a currentUser variable in a use case so that we can control the actions of the user currently using our program. | |
| 11/12  Coco  Alfred | Coco: Created RewardManager and some of the implementations of the reward system.  Alfred: Refined UserManager so that it controls the actions of currentUser, as discussed on 11/11. | |
| 11/13  Alfred  Ashley  Arthur  Coco  Tong Su | Implemented Facade and moved the variable currentUser from UserManager to Facade. | |
|  | | |
| Documentation | | |
| 11/13  Alfred  Ashley  Arthur  Coco  Tong Su | Working on the Design Document and polish up the code  We have completed SOLID and Clean Architecture part of the Design Document  Discussed the need to have a Facade that unifies all use cases that User concerns. | |
| 11/14  Alfred  Ashley  Coco  Tong Su | Polishing code.  Finishing the Design Document. | |