CSC207 Final Presentation

Group 208

Jillian Escobar, Yael Lyshkow, Terry Fu, Tiana Chan, Timothy Li

Table of contents

01 **Specification**

What our application does!

02 **API Usage**

What API we used and how

03 **Functionality**

Demonstration of functionality

04 **Design**

SOLID, CA and Design Patterns

05 **Testing**

Testing of our application

06 **Organisation**

How we organised our code

01

Specification

Specification

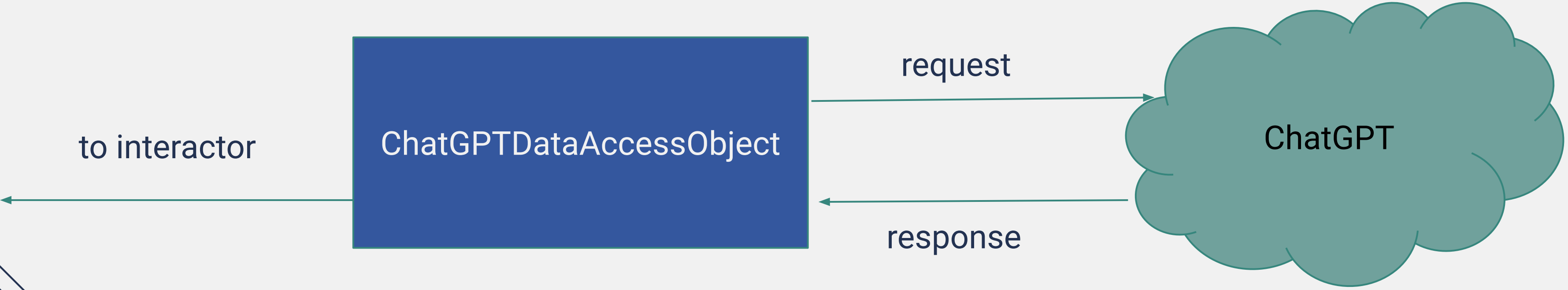
- The main goal of our program is to allows users to create their own **customized tierlists**
- Users can **make an account** on which they can **create and save their tierlists**
- The application allows users to generate new tierlists through multiple different methods
  - **Automatically generated** tierlists from a prompt
  - Tierlists generated from **manually inputted** data
- A user's previously made tierlists can then be **loaded and edited** if desired
- Users are able to **search, view, and follow other users,** and are able to look at their customized tierlists

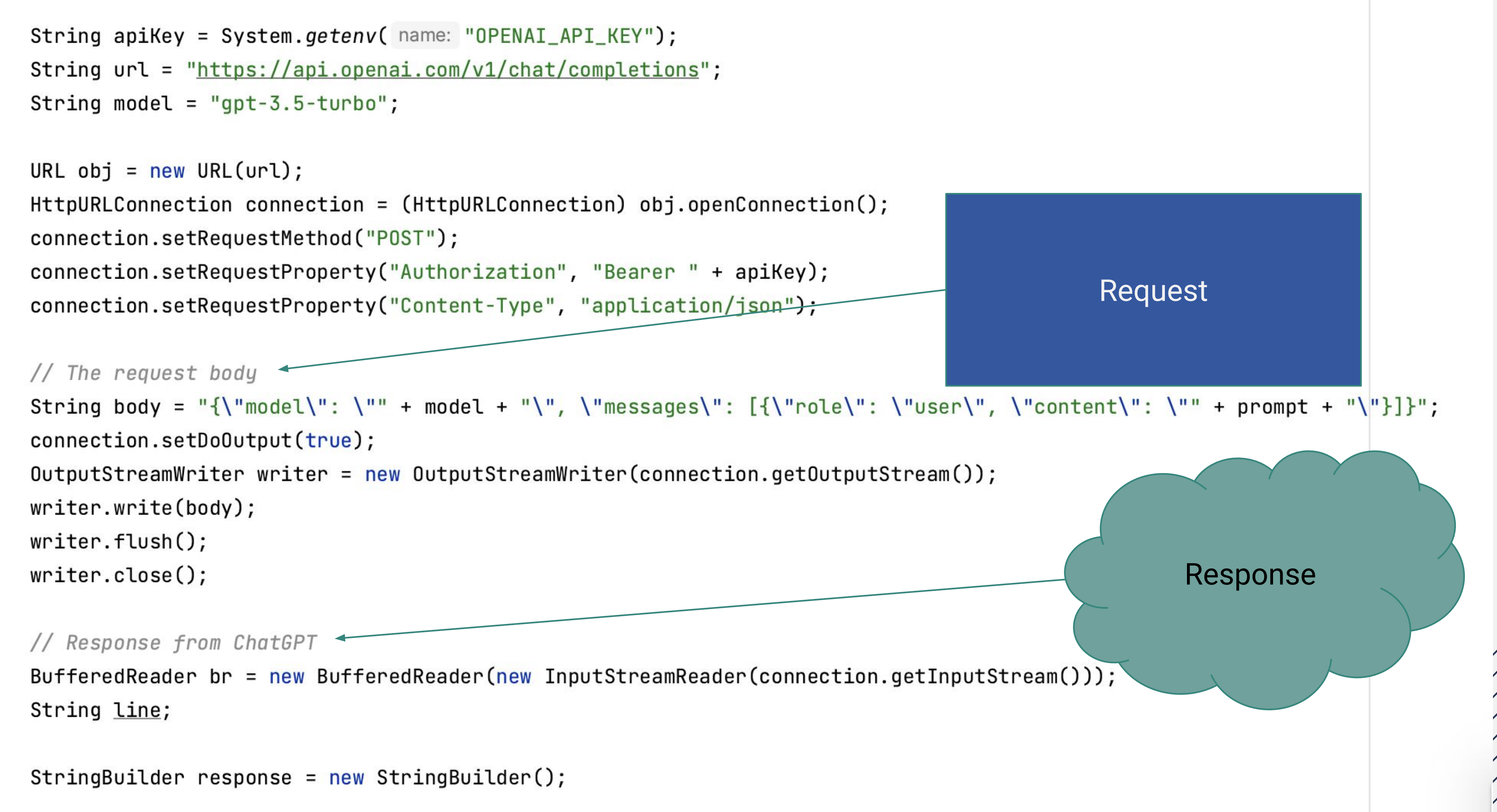
02

API Usage

API Used

- Our team chose to use the **OpenAI API** for our project in order to include an element of customizability on the user's end into our application
- API usage within our code:





```java
String apiKey = System.getenv( name: "OPENAI_API_KEY");
String url = "https://api.openai.com/v1/chat/completions";
String model = "gpt-3.5-turbo";

URL obj = new URL(url);
HttpURLConnection connection = (HttpURLConnection) obj.openConnection();
connection.setRequestMethod("POST");
connection.setRequestProperty("Authorization", "Bearer " + apiKey);
connection.setRequestProperty("Content-Type", "application/json");

// The request body
String body = "{\"model\": \"" + model + "\", \"messages\": [{\"role\": \"user\", \"content\": \"" + prompt + "\"}]}";
connection.setDoOutput(true);
OutputStreamWriter writer = new OutputStreamWriter(connection.getOutputStream());
writer.write(body);
writer.flush();
writer.close();

// Response from ChatGPT
BufferedReader br = new BufferedReader(new InputStreamReader(connection.getInputStream()));
String line;

StringBuilder response = new StringBuilder();
```

Request

Response

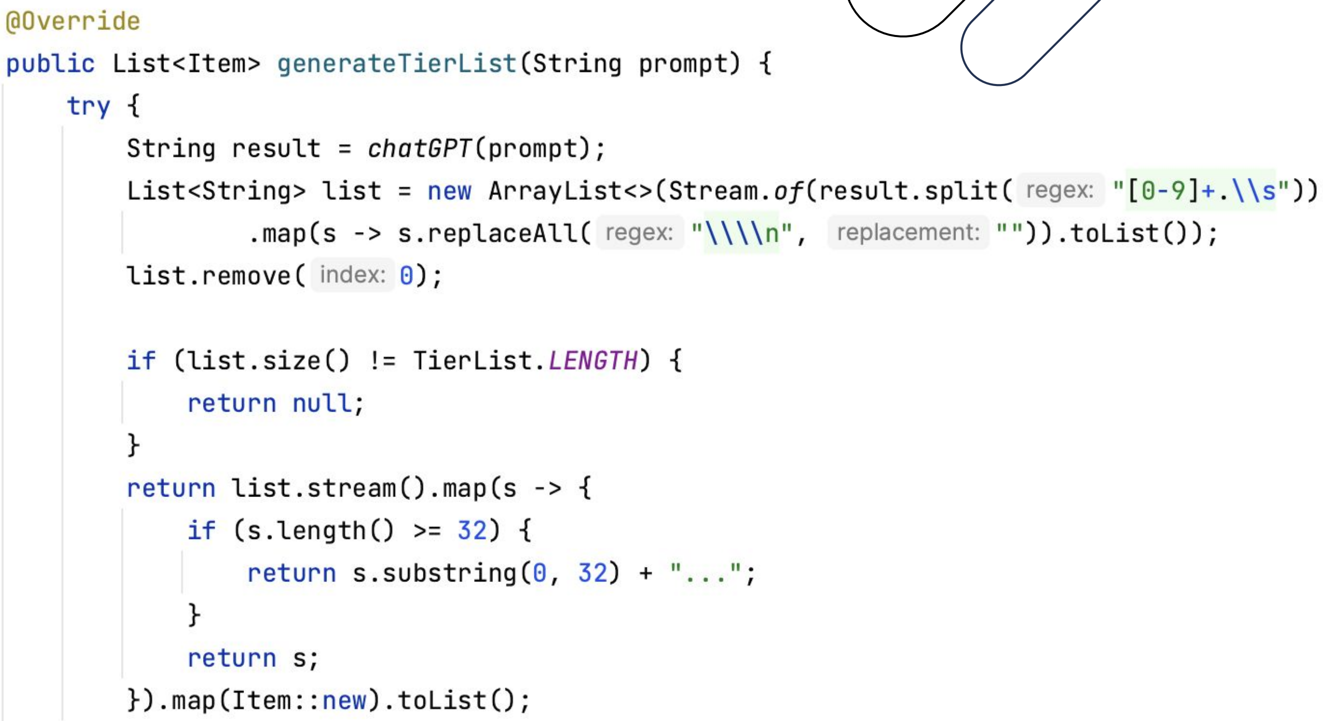
List 8 famous actresses

Scarlett Johansson, Meryl Streep, Emma Watson, ...

ChatGPT

Data Access

Taking the API response and parsing it to a usable format



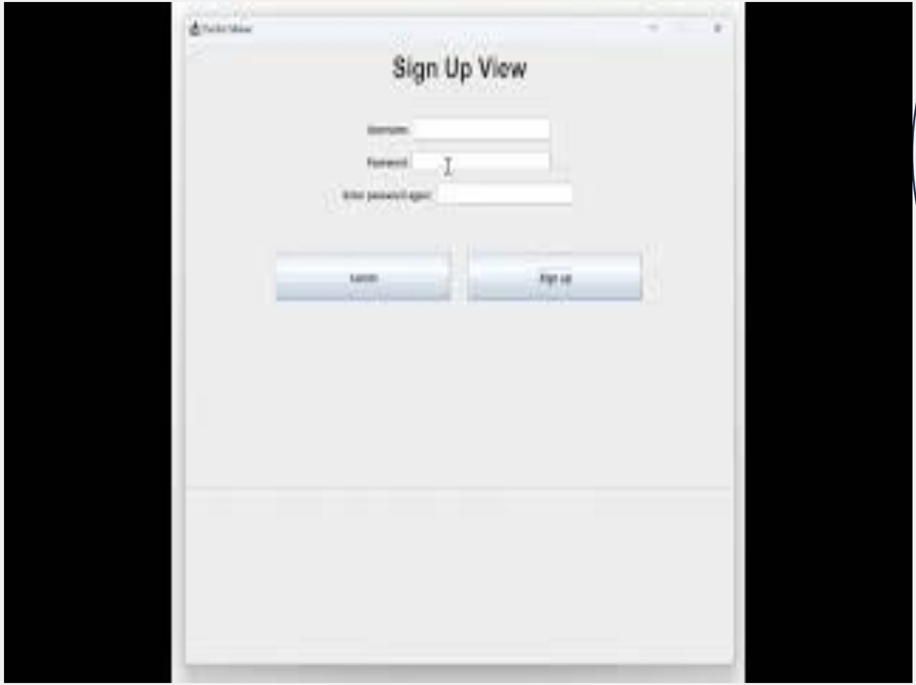
```
@Override
public List<Item> generateTierList(String prompt) {
    try {
        String result = chatGPT(prompt);
        List<String> list = new ArrayList<>(Stream.of(result.split( regex: "[0-9]+.\\s"))
                .map(s -> s.replaceAll( regex: "\\\\n",  replacement: "")).toList());
        list.remove( index: 0);

        if (list.size() != TierList.LENGTH) {
            return null;
        }
        return list.stream().map(s -> {
            if (s.length() >= 32) {
                return s.substring(0, 32) + "...";
            }
            return s;
        }).map(Item::new).toList();
```

03

Functionality Demonstration

Signup & Login

menu view, cancel buttons,
signup fail and success,
login fail and success,
log out



Sign Up View

Username:

Password:

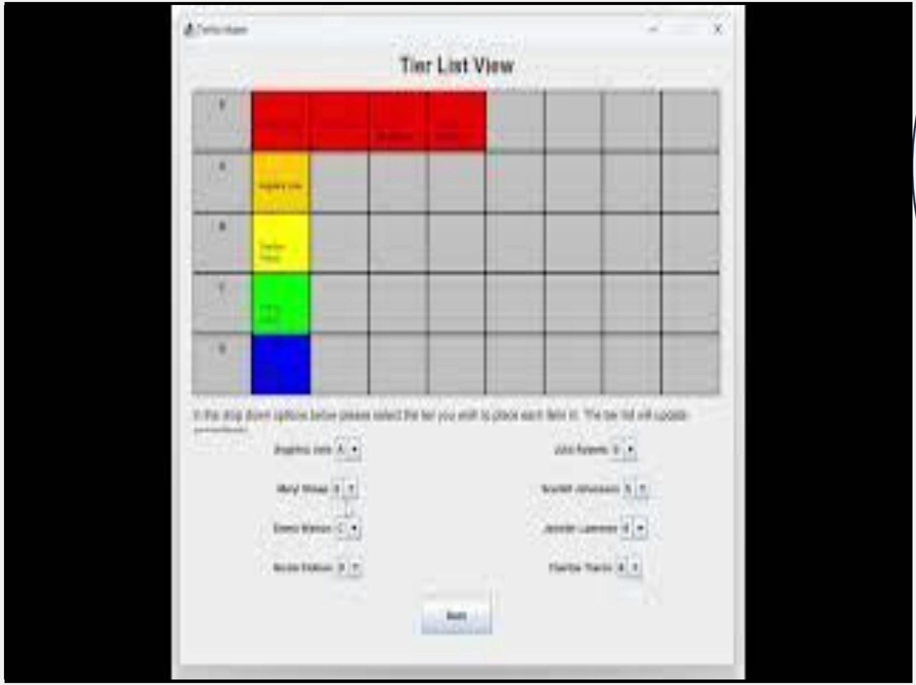
Enter password again:

Search & Follow

search fail, search success,
follow, unfollow,
view tierlists

View Existing Tierlist

back button, choose tierlist, edit tierlist



Tier List View

In the drop down options below please select the tier you wish to place each item in. The tier list will update [illegible]

Angelina Jolie [illegible] ▾

Meryl Streep [illegible] ▾

Emma Watson [illegible] ▾

Nicole Kidman [illegible] ▾

Julia Roberts [illegible] ▾

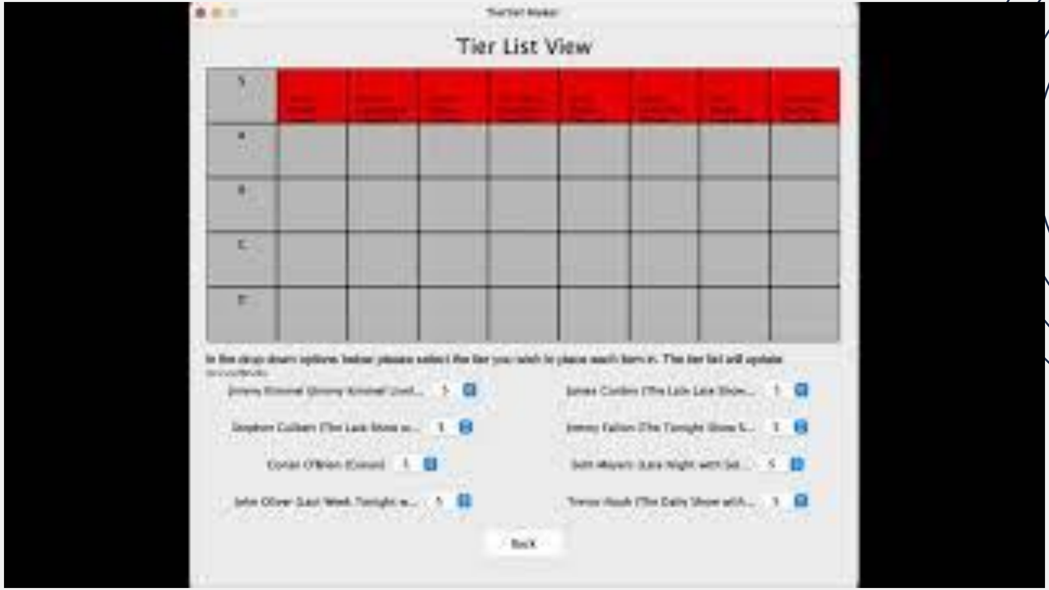
Scarlett Johansson [illegible] ▾

Jennifer Lawrence [illegible] ▾

Charlize Theron [illegible] ▾

Custom and Random Tierlists

edit existing tierlist, create random tierlist, change tiers, create custom tierlist



Tier List View

S

A

B

C

D

In the drop down options below please select the tier you wish to place each item in. The tier list will update accordingly.

Jimmy Kimmel (Jimmy Kimmel Live!... S

James Corden (The Late Late Show... S

Stephen Colbert (The Late Show w... S

Jimmy Fallon (The Tonight Show S... S

Conan O'Brien (Conan) S

Seth Meyers (Late Night with Set... S

John Oliver (Last Week Tonight w... S

Trevor Noah (The Daily Show with... S

04 Design

SOLID

How we adhered to the SOLID Design principles within our programming

CA

How we implemented the logic of Clean Architecture Engine in our code

Design Patterns

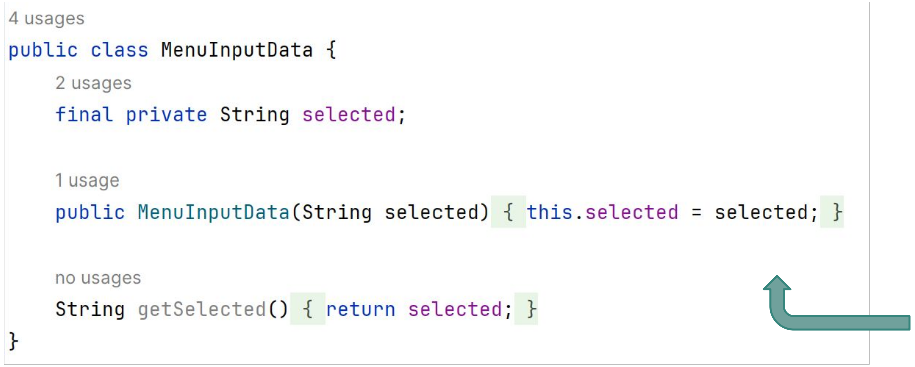
How we used design patterns to make our code more efficient

SOLID Principles

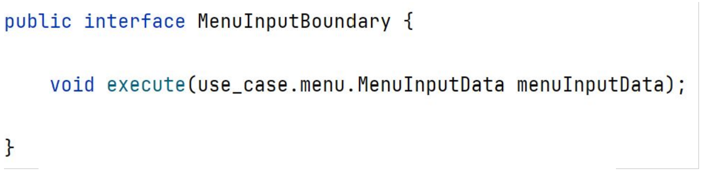
Single Responsibility Principle

The Single Responsibility Principle states that **a class should have one and only one reason to change**.

- This is demonstrated in the use case layer of our Menu, where we have implemented a MenuInputBoundary interface and a MenuInputData class:



```
4 usages
public class MenuInputData {
    2 usages
    final private String selected;

    1 usage
    public MenuInputData(String selected) { this.selected = selected; }

    no usages
    String getSelected() { return selected; }
}
```



```
public interface MenuInputBoundary {

    void execute(use_case.menu.MenuInputData menuInputData);

}
```

Should we ever want to change the specifics of the input data (such as not storing selected as a String), **only** MenuInputData has to change

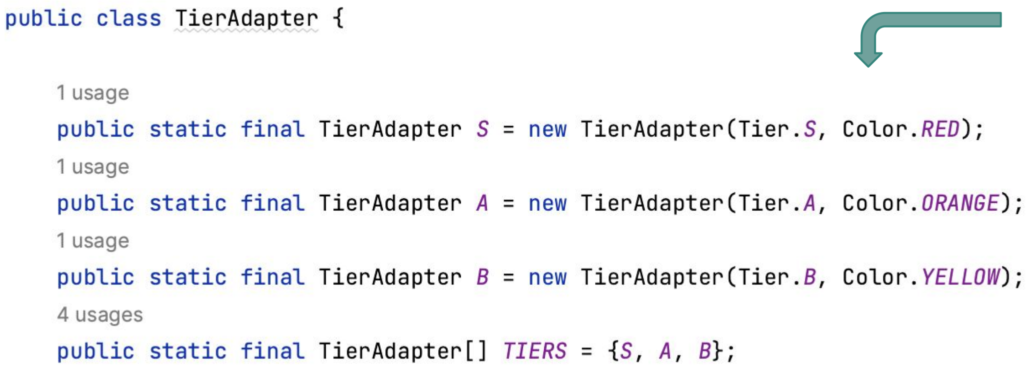
SOLID Principles

Open-Closed Principle

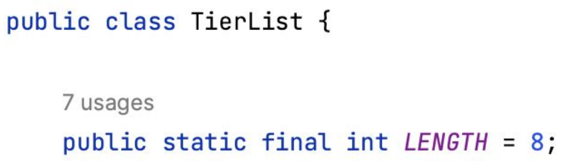
The Open-Closed Principle states that **classes should be open for extension but closed for modification**.

- Within our code we can see an example of this SOLID design principle often within the TierList and TierAdapter entities:

Here the **number** of tiers, their **names** and their **colour** are all customisable within the TierList and TierAdapter entities and all references to the tier names or colours are to here



```
public class TierAdapter {

    1 usage
    public static final TierAdapter S = new TierAdapter(Tier.S, Color.RED);
    1 usage
    public static final TierAdapter A = new TierAdapter(Tier.A, Color.ORANGE);
    1 usage
    public static final TierAdapter B = new TierAdapter(Tier.B, Color.YELLOW);
    4 usages
    public static final TierAdapter[] TIERS = {S, A, B};
```



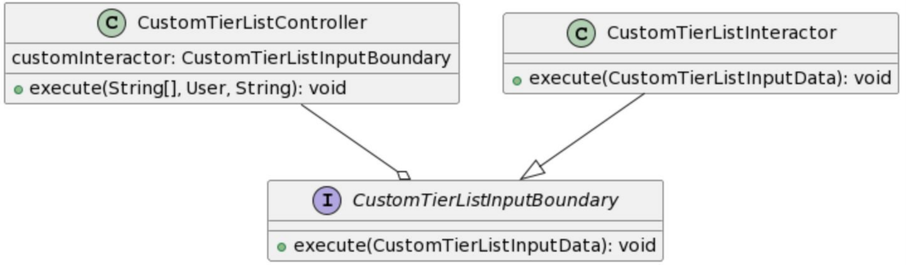
```
public class TierList {

    7 usages
    public static final int LENGTH = 8;
```

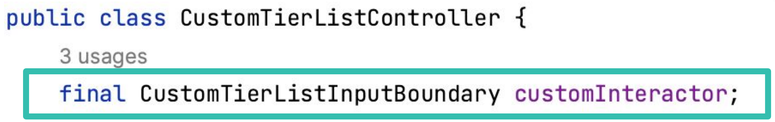
SOLID Principles

Dependency Inversion Principle

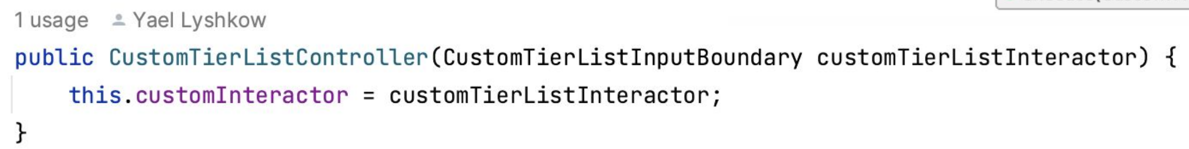
The Dependency Inversion Principle states **that details should depend on abstractions not concretions**.

- Within our code we implemented this primarily by adhering to Clean Architecture, which places a lot of emphasis on relying on interfaces instead of implementations of them.
- **For example:**


C CustomTierListController
customInteractor: CustomTierListInputBoundary
execute(String[], User, String): void
C CustomTierListInteractor
execute(CustomTierListInputData): void
I CustomTierListInputBoundary
execute(CustomTierListInputData): void



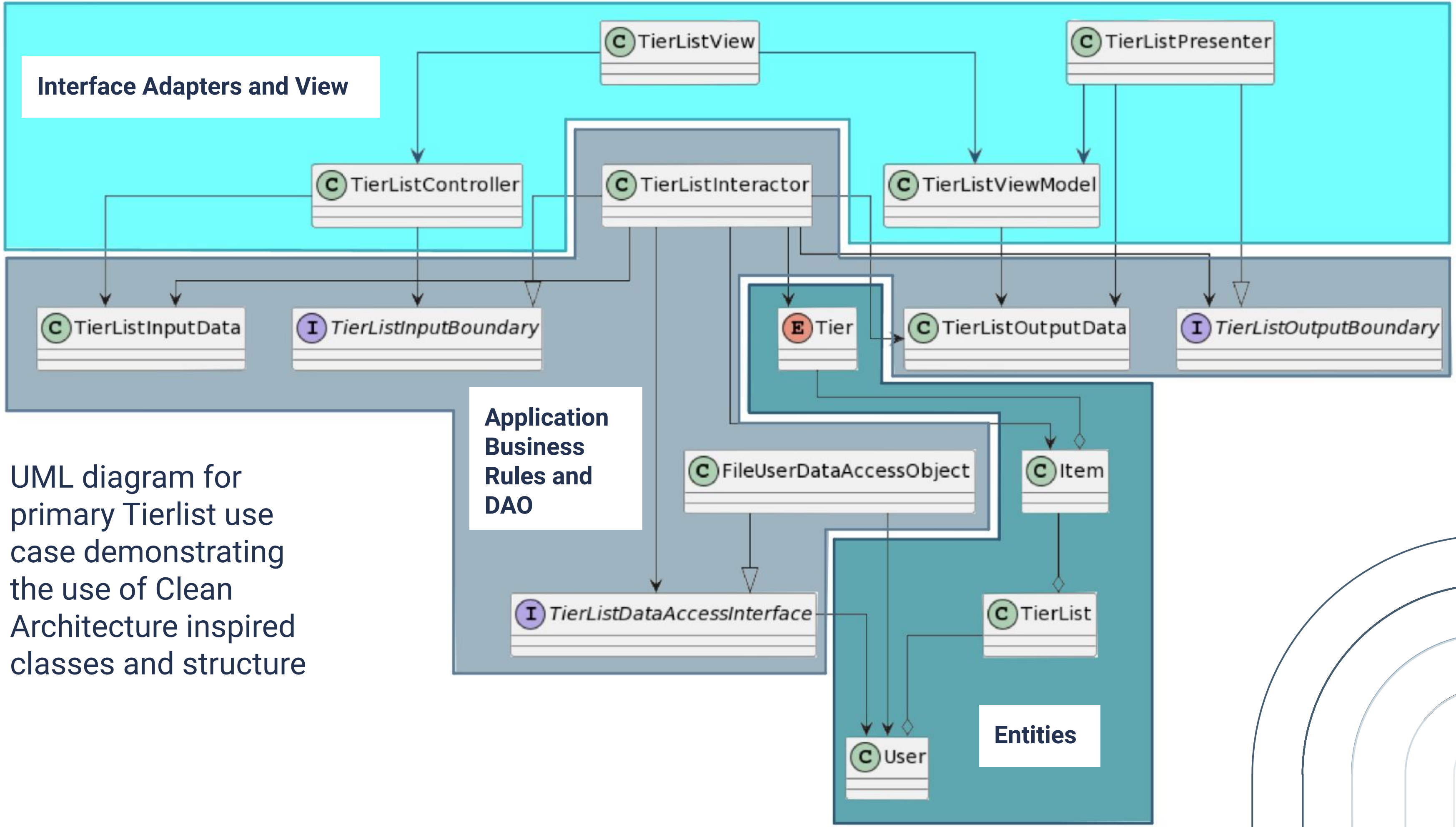
```
public class CustomTierListController {
    3 usages
    final CustomTierListInputBoundary customInteractor;
```



```
    1 usage   Yael Lyshkow
    public CustomTierListController(CustomTierListInputBoundary customTierListInteractor) {
        this.customInteractor = customTierListInteractor;
    }
```

Clean Architecture

- Throughout our program we have adhered to the principles of Clean Architecture
- The following slide contains a UML diagram of our TierList use case where we have implemented the Clean Architecture Engine
- We used the same setup for all our use cases



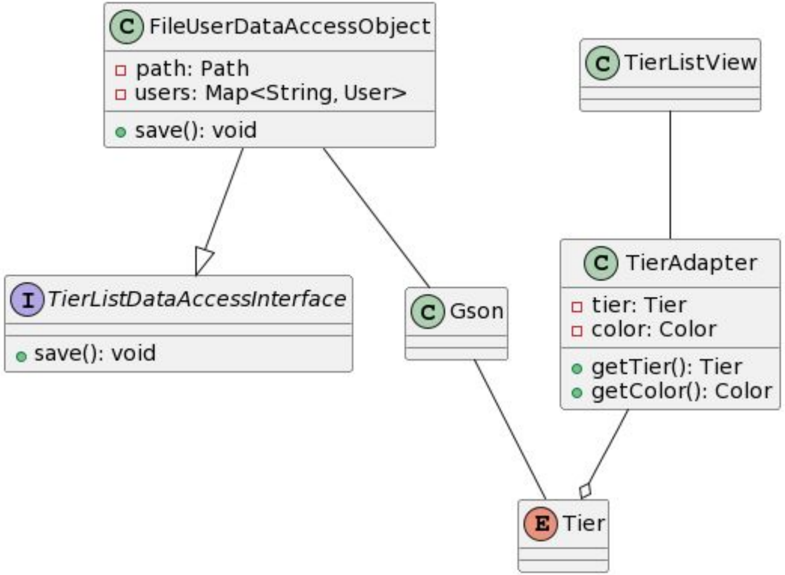
UML diagram for primary Tierlist use case demonstrating the use of Clean Architecture inspired classes and structure

Design Patterns

Adapter

The Problem:
Gson's .json read/writing capabilities store **superfluous information** about tiers, including the colours…

The Solution:
Using the **adapter** design pattern, we can separate our **Tier** and **TierAdapter** classes and store other information in a separate class!



Design Patterns

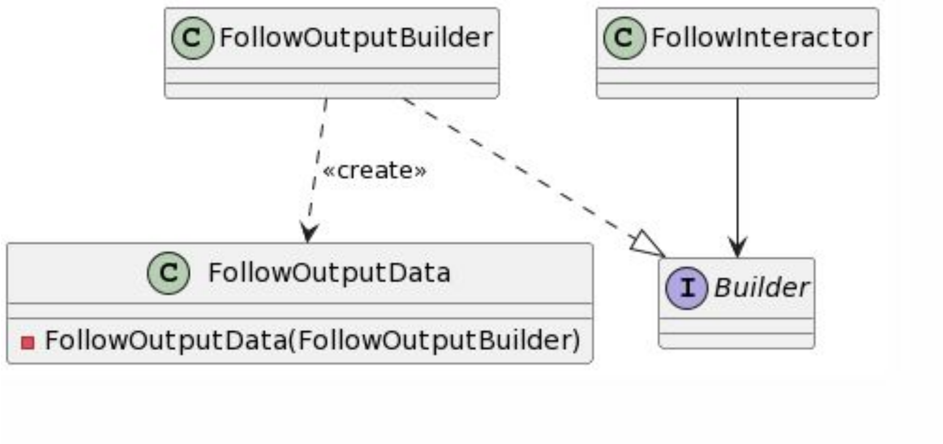
Builder

The Problem:

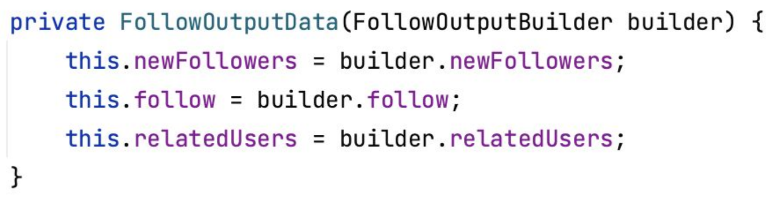
FollowOutputData has both **optional and required variables.** We do not want the rest of the program to be dependant on the constructor signature when constructing the object

The Solution:

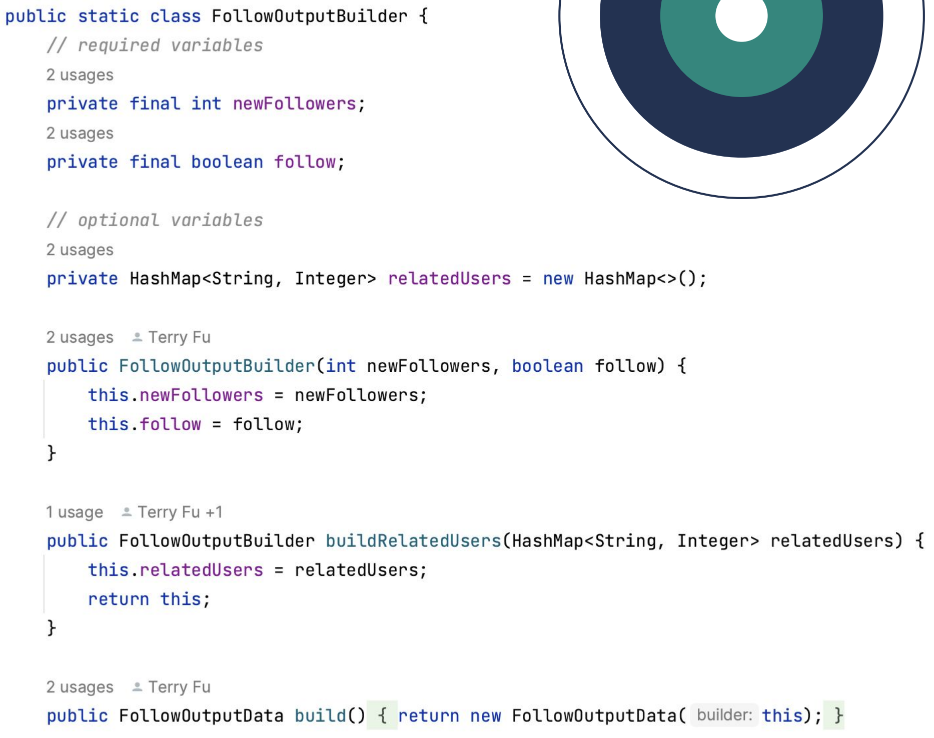
The **Builder** design pattern **encapsulates** all constructor inside its class. This **reduces coupling** between the constructor and the rest of the code, and facilitates customization of the object



Inside FollowOutputData

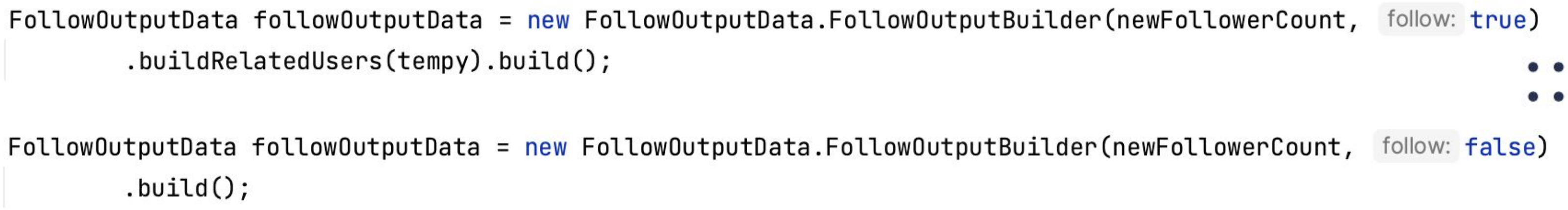


```
private FollowOutputData(FollowOutputBuilder builder) {
    this.newFollowers = builder.newFollowers;
    this.follow = builder.follow;
    this.relatedUsers = builder.relatedUsers;
}
```



```
public static class FollowOutputBuilder {
    // required variables
    2 usages
    private final int newFollowers;
    2 usages
    private final boolean follow;

    // optional variables
    2 usages
    private HashMap<String, Integer> relatedUsers = new HashMap<>();

    2 usages  Terry Fu
    public FollowOutputBuilder(int newFollowers, boolean follow) {
        this.newFollowers = newFollowers;
        this.follow = follow;
    }

    1 usage  Terry Fu +1
    public FollowOutputBuilder buildRelatedUsers(HashMap<String, Integer> relatedUsers) {
        this.relatedUsers = relatedUsers;
        return this;
    }

    2 usages  Terry Fu
    public FollowOutputData build() { return new FollowOutputData( builder: this); }
```

Elsewhere in FollowInteractor



```
FollowOutputData followOutputData = new FollowOutputData.FollowOutputBuilder(newFollowerCount,  follow: true)
        .buildRelatedUsers(tempy).build();

FollowOutputData followOutputData = new FollowOutputData.FollowOutputBuilder(newFollowerCount,  follow: false)
        .build();
```

05

Testing

Test Line Coverage

84%

Data Access

89%

Entities

94%

Factories

84%

Interface Adapter

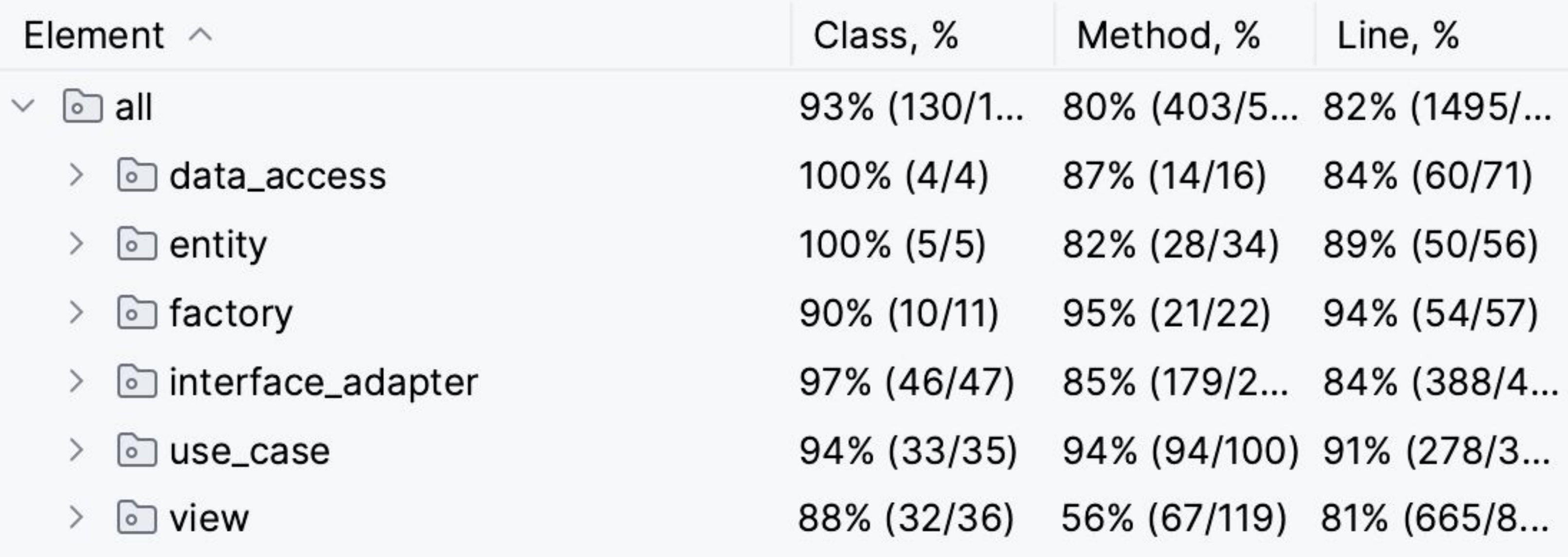
91%

Use Case

81%

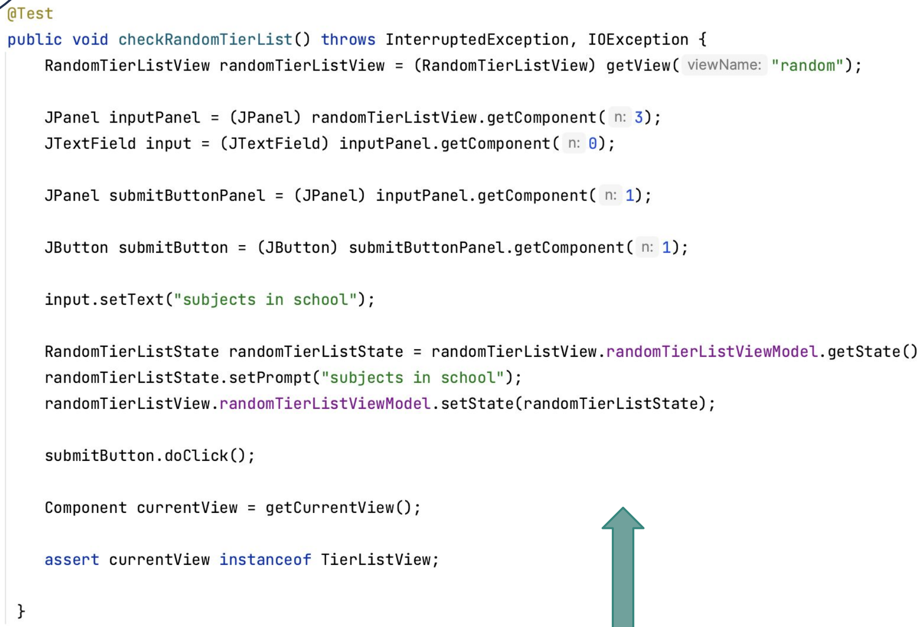
View

Test Coverage



Element	Class, %	Method, %	Line, %
all	93% (130/1...	80% (403/5...	82% (1495/...
data_access	100% (4/4)	87% (14/16)	84% (60/71)
entity	100% (5/5)	82% (28/34)	89% (50/56)
factory	90% (10/11)	95% (21/22)	94% (54/57)
interface_adapter	97% (46/47)	85% (179/2...	84% (388/4...
use_case	94% (33/35)	94% (94/100)	91% (278/3...
view	88% (32/36)	56% (67/119)	81% (665/8...

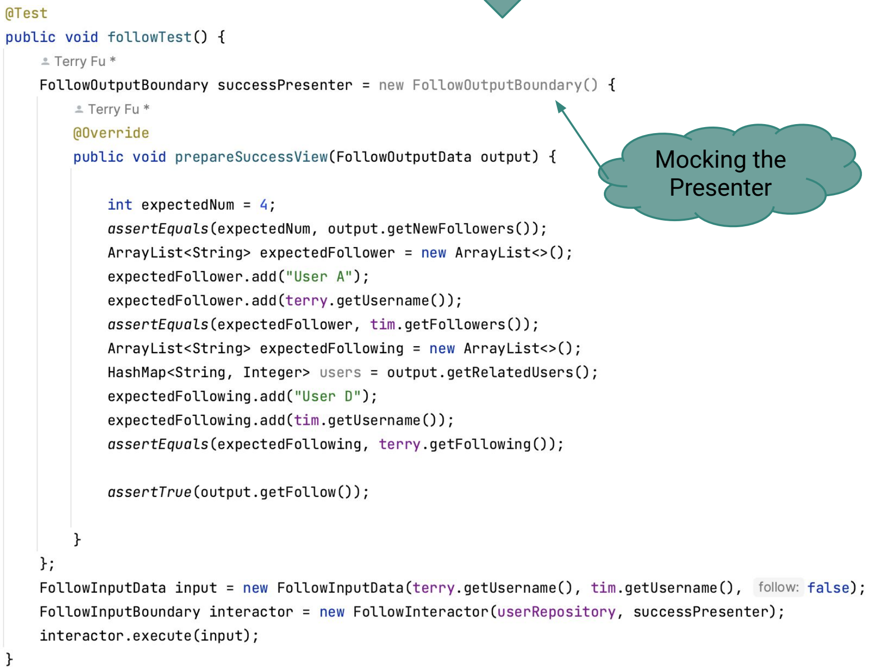
Use Case Interactor Tests



```
@Test
public void checkRandomTierList() throws InterruptedException, IOException {
    RandomTierListView randomTierListView = (RandomTierListView) getView( viewName: "random");

    JPanel inputPanel = (JPanel) randomTierListView.getComponent( n: 3);
    JTextField input = (JTextField) inputPanel.getComponent( n: 0);

    JPanel submitButtonPanel = (JPanel) inputPanel.getComponent( n: 1);

    JButton submitButton = (JButton) submitButtonPanel.getComponent( n: 1);

    input.setText("subjects in school");

    RandomTierListState randomTierListState = randomTierListView.randomTierListViewModel.getState()
    randomTierListState.setPrompt("subjects in school");
    randomTierListView.randomTierListViewModel.setState(randomTierListState);

    submitButton.doClick();

    Component currentView = getCurrentView();

    assert currentView instanceof TierListView;

}
```

Example user interface test

Example use case test



```
@Test
public void followTest() {
    Terry Fu *
    FollowOutputBoundary successPresenter = new FollowOutputBoundary() {
        Terry Fu *
        @Override
        public void prepareSuccessView(FollowOutputData output) {

            int expectedNum = 4;
            assertEquals(expectedNum, output.getNewFollowers());
            ArrayList<String> expectedFollower = new ArrayList<>();
            expectedFollower.add("User A");
            expectedFollower.add(terry.getUsername());
            assertEquals(expectedFollower, tim.getFollowers());
            ArrayList<String> expectedFollowing = new ArrayList<>();
            HashMap<String, Integer> users = output.getRelatedUsers();
            expectedFollowing.add("User D");
            expectedFollowing.add(tim.getUsername());
            assertEquals(expectedFollowing, terry.getFollowing());

            assertTrue(output.getFollow());

        }
    };
    FollowInputData input = new FollowInputData(terry.getUsername(), tim.getUsername(),  follow: false);
    FollowInputBoundary interactor = new FollowInteractor(userRepository, successPresenter);
    interactor.execute(input);
}
```

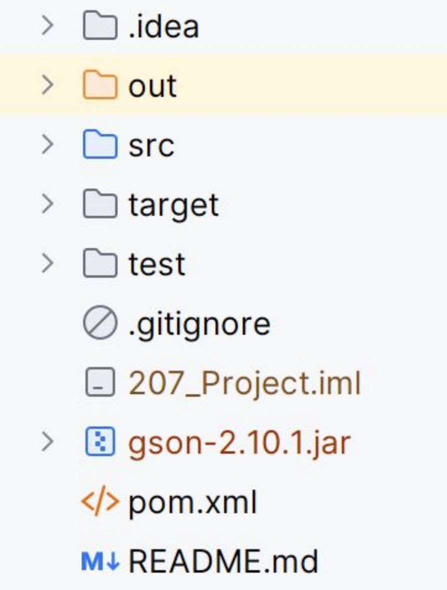
Mocking the Presenter

06

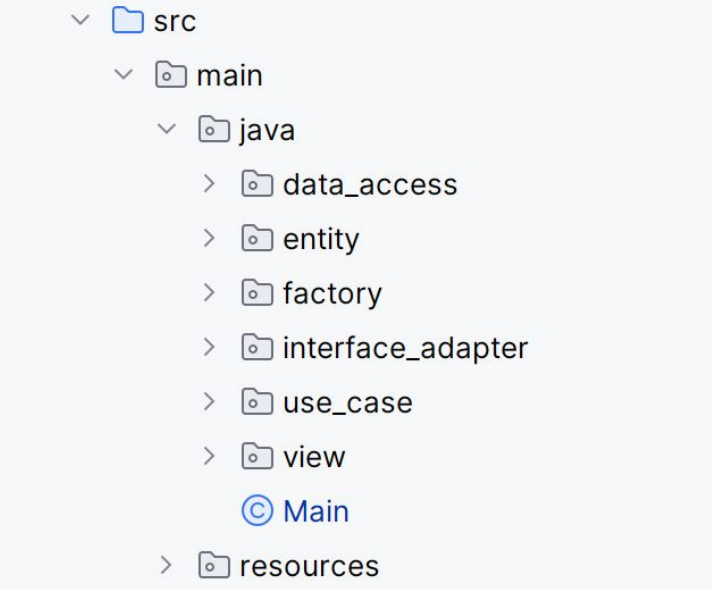
Code Organization

Organized by Layer of CA

Our packages are clearly organized by the levels of clean architecture



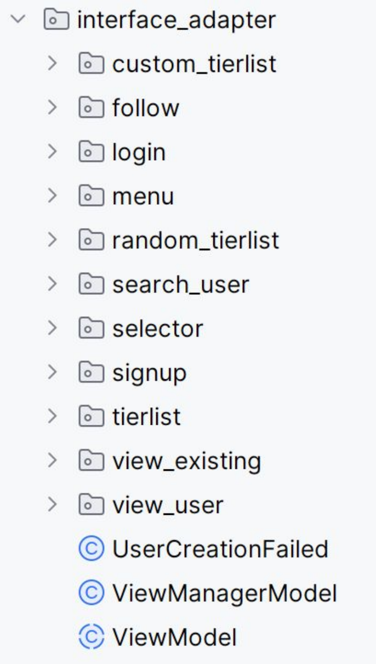
```
> .idea
> out
> src
> target
> test
  .gitignore
  207_Project.iml
> gson-2.10.1.jar
  pom.xml
  README.md
```



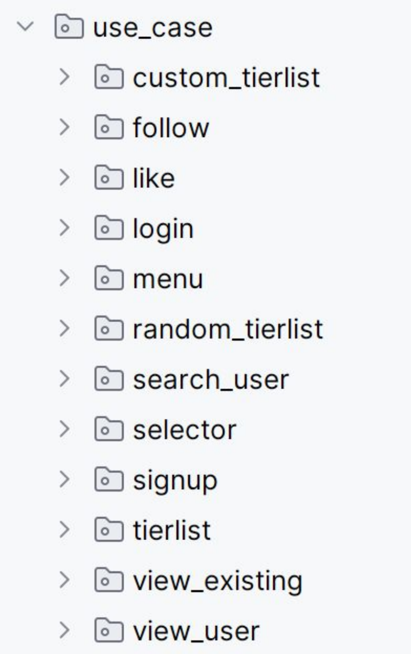
```
v src
  v main
    v java
      > data_access
      > entity
      > factory
      > interface_adapter
      > use_case
      > view
        Main
    > resources
```

Packages

Within each layer of clean architecture, each use case is neatly organized



```
interface_adapter
    custom_tierlist
    follow
    login
    menu
    random_tierlist
    search_user
    selector
    signup
    tierlist
    view_existing
    view_user
    UserCreationFailed
    ViewManagerModel
    ViewModel
```



```
use_case
    custom_tierlist
    follow
    like
    login
    menu
    random_tierlist
    search_user
    selector
    signup
    tierlist
    view_existing
    view_user
```

Implementation Responsibilities

Yael

Tierlist creation and arrangement use case front-end

Jillian

Tierlist creation and arrangement use case back-end

Tiana

Signup, Login, and Menu use case

Terry

Follow, Search, and View User use case back-end

Tim

Follow, Search and View User use case front-end

Thanks!