Design Document

HR System

(Tian Shu, Canyang, Shihan, Yuelin, Yiteng, Mihir)

**Specification:**

User → Employer, Employee, Intern

* a username, a password, salary, attendance, id, email, major
* Employee: access and edit their information
* Employer: access information of all users, has control over who’s suppose / not supposed to be in the system
* e.g., recruit and fire individuals

Use Cases

* User Manager: Array List of Users
* Salary Manager: HashMap <user id, salary>
* Job Manager:  HashMap <user id, job>
* Schedule Manager:  HashMap <user id, schedule>
* Intern Manager: HashMap <Intern, rating>

Controllers

* User Manager Controller
* Salary Manager Controller
* Job Manager Controller
* Schedule Manager Controller
* Inter Manager Controller
* provide additional handling of information for UI to use

Gateway

* Read Writer
* User Read Writer
* Salary Read Writer
* Schedule Read Writer
* Job Read Writer
* Intern Read Writer
* save data to file using serialization when data stored in use case class is being modified
* read data from use case when needed
* save data permanently using relative path

UI

* User surface
  + create the personal profile for new employees or employer
  + login by using the exist id
  + allow employees to check their information. e.g. salary and id
  + exit the system
* Salary surface
  + employer puts monthly money to the employee with his/her id
* Intern surface
  + employer has the power to create information for interns
* Schedule surface
  + Each user can use his/her id to check and update his/her current schedule
* Jobp surface
  + Each user can get a reference job by his/her major.

**SOLID**

* Single Responsibility Principle
  + In Entity folder, we use only one class to handle one single role (intern, employee, employer, user), such that every class should have only a single responsibility. This happens to our controller folder as well, each controller file contains one class that handle one specific scenario, so each class only have one reason to change.
* Open/Close Principle:
  + In each of the user interface (UI), we always have buttons, inputInfo and outputInfo. We repeat those codes in each of the files in UI (although they are not totally the same). But the basic use of them is the same, so we think we can create another folder “panel” which includes InputInforPanel, OutputInfoPanel, ButtonPanel and even more functionalities. This way in UI files we just need to import the file(panel) we need and then call it there. With this design, we can add any number of buttons or text fields and we don't need to re-write the code.
* Liskov Substitution Principle:
* Employer, Employee, and Intern extended from User and didn’t remove or alter any features
* Use case specific gateway implements ReadWrite Interface, and only extended.
* Interface Segregation Principle
* ReadWrite is a small interface, so that all class that implemented ReadWrite used all of its attributes
* Dependency Inversion Principle:
* Followed clean architecture so that individual pieces are separated
* For example, we modify Controllers, and it will never affect Use Case and Entities

**Clean Architecture**

CRC Model：Entity

树状图

中度可信度描述已自动生成 树状图

描述已自动生成 树状图

描述已自动生成 树状图

中度可信度描述已自动生成 表格

描述已自动生成

CRC Model：Use Case

表格

中度可信度描述已自动生成 表格

描述已自动生成 图形用户界面

低可信度描述已自动生成 表格

描述已自动生成 表格

中度可信度描述已自动生成

CRC Model：Controller

树状图

描述已自动生成 表格

中度可信度描述已自动生成 图形用户界面

中度可信度描述已自动生成 图形用户界面, 表格

中度可信度描述已自动生成 图形用户界面

描述已自动生成

CRC Model：Gateway Package

图表, 树状图

描述已自动生成 表格

描述已自动生成 表格

中度可信度描述已自动生成 树状图

低可信度描述已自动生成 图形用户界面

中度可信度描述已自动生成 图片包含 表格

描述已自动生成

CRC Model：User Interface Package图形用户界面

中度可信度描述已自动生成 图形用户界面

低可信度描述已自动生成 表格

描述已自动生成 表格

描述已自动生成 表格

中度可信度描述已自动生成 图形用户界面, 应用程序

描述已自动生成 图表, 树状图

描述已自动生成

Scenario Walk Through

Employer Scenario:

* Firstly, employer uses this HR system, and the system shows him the log-in page using the UserSystemController.
* Register himself through the registration button UserManager use case, we will need to enter his ID, username, password, salary, and attendance.
* After he successfully registered himself, the system will send him back to the login page.
* When the employer login, he can change put employee’s salary, add new Interns in the company and he can also check for the schedule about himself.
* Noticed that only the employer has the authorization to create an Intern account. *(So far, everyone can create a User since our code did not add the restriction on different users’ rights.)*

Employee Scenario:

* An employee who is recruited by an employer should register his username, temporary password, major and email by themself. The system will show the employee the log-in page, he will use his username and password to login in.
* We believe the first time the employee logs in, the system will show him/her 5 options, salary, jobposition, intern, schedule and exit, he/she can choose one of them, but they don’t have rights to do some of the function in this option.
* The employee logs in to the system and he can see his profile page, he now can see all his personal information including the detailed components of his salary and the reference job position.

Intern Scenario:

* Firstly, only employer can edit the detailed information of interns. Employees and interns cannot change the information. The system will check their IDs to judge whether he (or she) is an employer.
* Employer needs to add the detailed information of one intern and his (or her) internship grade to our HR System.
* Next, the employer can enter the standard grade to select the interns that their grades exceed the standard grade. There will be accepted as our employees.
* After the grading, interns can check their accepting status by entering their IDs.

**Design Pattern**

The template design pattern is significantly important for our group project. Our HR system includes the detailed information of tons of people (User). In order to avoid code duplication, we only want to create one class to represent User inside of Entity, this class is the template that decides the main structure of User, then after this, we are free to create subclasses to override specific steps of the algorithm without changing its structure. So, we create different kinds of roles for the User in the company, such as manager, boss, employee and applicants, User with those roles is still part of User so they share the same properties as User. These are the subclasses that override the information of people (User superclass). The User superclass are the foundation of the whole HR system.

**Use of GitHub Features**

* Created Branches of our own to work locally
* pull requests and merge when necessary
* able to work together efficiently and simultaneously

**Code Style and Documentation**

In Intellij, there are only a few warnings related to more features that we plan to extend. Other than that, all there are no warnings related to styles and etc. Javadoc is used where needed. And if a Java programmer were to open your repo and navigate to a random file, they would be able to understand your code.

**Testing**

(Under:  src / test)

Most components of our system are tested. The user interface components would be difficult to test due to your design, but we will find a way to test it later.

**Refactoring**

There is evidence that our team has refactored code in a meaningful way during the project. After phase0, we reorganized all the codes and separated them into five folders. In each of the folders, we created more files where each file handles a specific scenario. But at that time, we did not know how to use pull request, so you may not be able to find it thought requests. For phase 1, some of us used branches with pull requests to work on project simultaneously and refactor different parts of the program. For code smells, some of our entity have relative long parameters, we may fix this in the future by giving default values.

**Code Organization**

* Organized code into packages according to clean architecture
* Inner most layer: entities
* Second layer: use case
* Third layers: gateway and controllers
* Outer most layer: user interface
* Able to change outer layers without interfering inner layers

**Functionality**

* Our program does what the specification says it should do
* Demo program's functionality to our TA through presentation
* Functionality sufficiently ambitious, given the size of your group
* Data are being stored permanently in relative paths in txt files using serialization

**Decision Making Process**

* Our users are employees, employer, and interns. So, we have corresponding classes in Entity, Usecase, and Controller
* In order to make our data can be used flexibly, we write our own database.
* Only the employee has the authority to change the information, and other users can create and access the information

**Progress Report**

Open Question

* How to write unit tests for classes in package User Interface?
* How to further make UI more appealing to Users?
* How to utilize data stored in the database effectively with the UI?

What Work Well

* We expanded our specification.
* We solved the questions of phase0.
* Our project has been able to operate normally and can provide users with a basic operating platform

Group Members’ Contribution

***Tian Shu***

* Expanded CRC model and some Unit Tests
* Database for all of 5 use cases using serialization

Plan for Phase 2:

* Expand CRC model
* Write Unit Test before the implementation of new classes and interfaces
* connect UI to the database

***Canyang***

* Expand use cases, controllers and UI(GUI)

Plan for Phase 2:

* Try to add some new features in controllers and UI(GUI). Improve the stability.

***Shihan***

* Annotations
* Tests

Plan for Phase 2:

* Checking mistakes/Debug through writing annotations and Unit tests.

***Yuelin***

* Design pattern and solid design principle
* some code, comments, annotation, improve ui functionality

Plan for Phase 2:

* Implement more controller class to enhance user experience.
* add more controller for ui to make it better
* try to create one interface for each kind of user

***Yiteng***

* Expand use cases, controllers and some UI, write some tests for use cases and controllers

Plan for Phase 2:

* Add some new features for controllers and UI(GUI).  Improve the stability.

***Mihir***

no contribution for phase 1