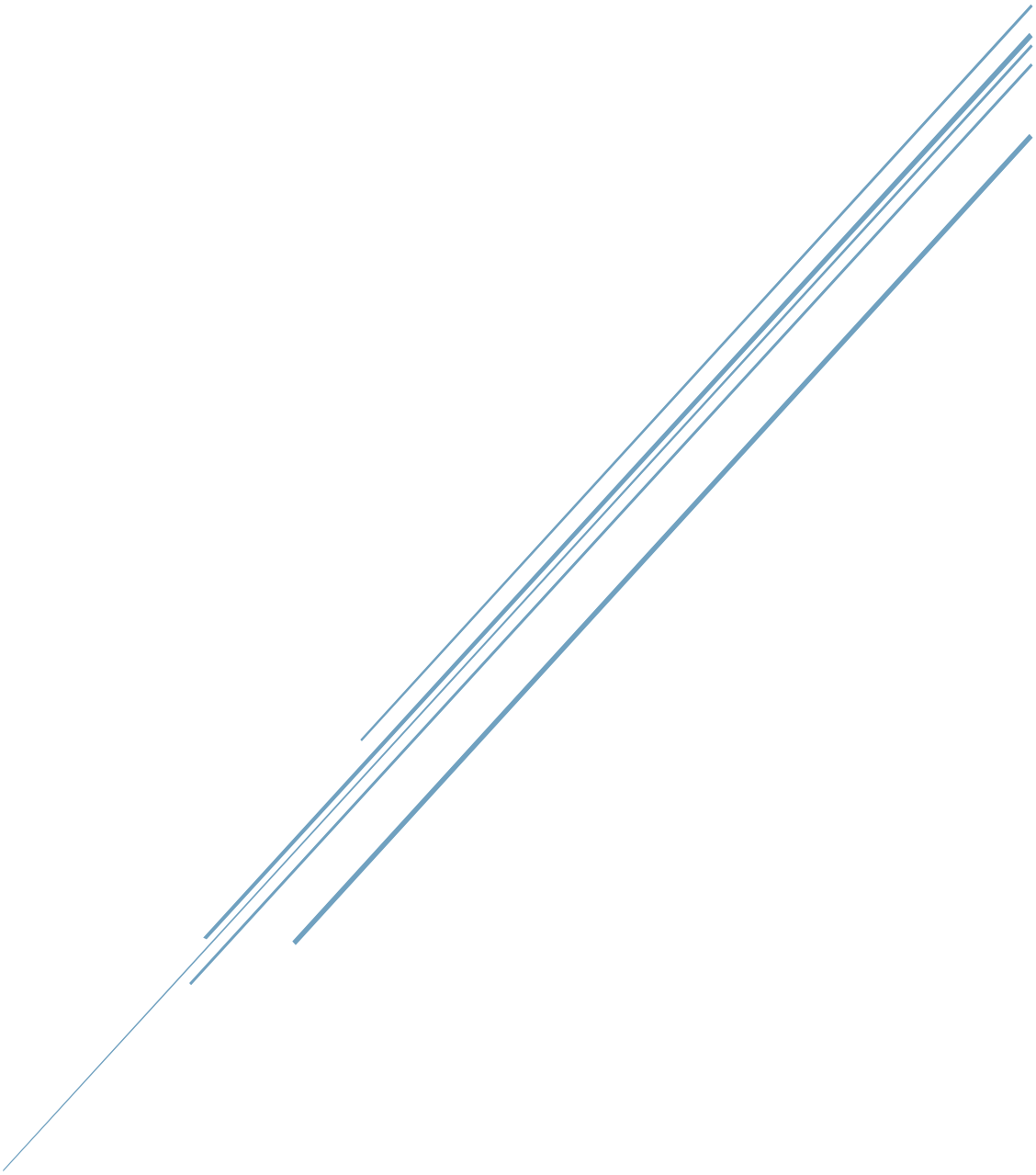
ACCEPTANCE RATE CALCULATOR

Modul 226b



Alisha Khalid, Nuwera Mohammad

Table of Content

Introduction

This documentation is about an Acceptance-Rate-Calculator for the Modul 226b. We decided to make a Rate-System to see the possibilities for the future. We thought about classes which we should use and first came up with a person, their resume, educational locations, and jobs.

With this project we were able to use and enhance our Java knowledge. It also helped us to deepen and understand the theory of this module in practice. With this project we want to prove our knowledge for association, delegation, composition, aggregation, and the ***newly learned inheritance***.

Development Environment

IDE

For the practical implementation we used IntelliJ. Therefore, we listed the most used shortcuts.

Shortcut	Description
Double shift	Search everywhere (files, settings)
Ctrl-shift-A	Find actions (example help)
F2	Navigate between code issues
Ctrl-shift-enter	Complete statement
Ctrl-alt-L	Format Code
Ctrl-shift-/	Add / remove block line comment
Alt-Insert	Generate Constructors, Getters / Setters
Ctrl-W	Jump over words / select words

GitHub

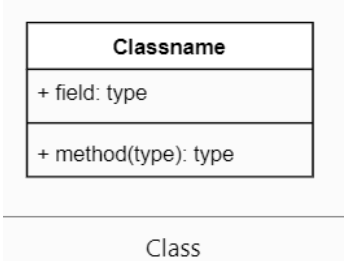
To work together efficiently we used GitHub. For this we created a repository on which our whole project is pushed. Therefore, we used following Commands:

Command	Description
Git add <file>	This command is used, after changing something in the code to add it to the repository
Git commit -m “message”	With command we write a short text about what is new in the code
Git push	With this command the changes are pushed to the repository
Git stash	You can do this, when both team partners worked in the same file one has to stash his to not overwrite each other’s code. It can be retrieved later
Git pull	With this command you pull / get the latest version of the project. → status

Draw.io

Draw.io is a leading diagramming tool that allows you to collaborate visually in your browser with your work partners. We used to make the class diagram. It is very simple to use, it gives plenty of options to choose what kind of diagram you are making and accordingly to that there are different options such as UML or Entity Relation. To make the Class Diagram we used the Options from UML.

This is an example; the UML provide over many options such as this class layout.



Planning

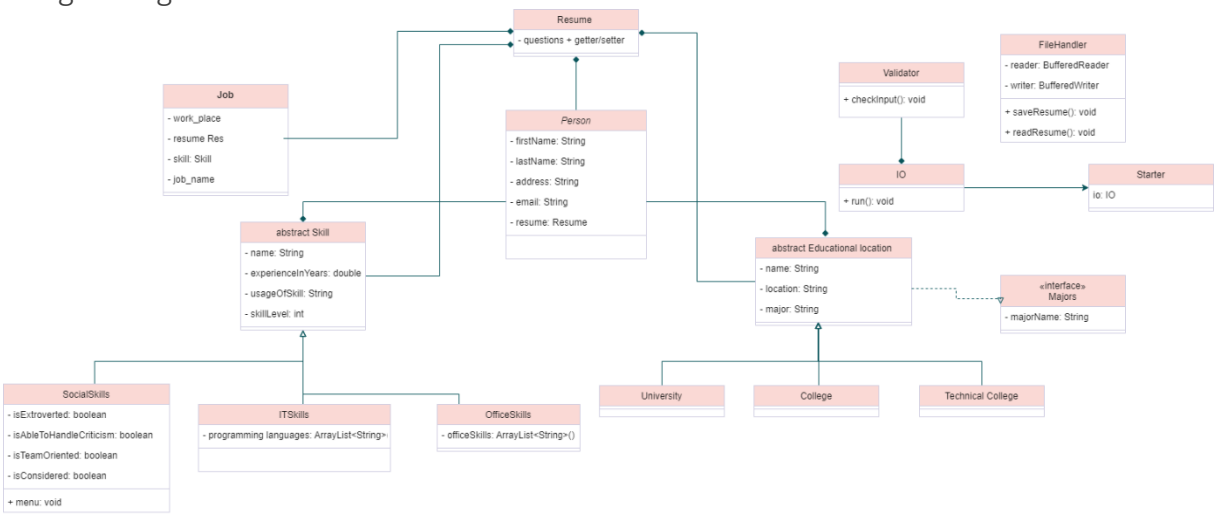
To start with a project, planning is the most important part. That's why as soon as we got an idea, we started to think about the different use cases and what you're able to do with this program. To get an overview of all classes we made a rough design of the class diagram and got feedback from our teacher.

Use Cases

Pre-Condition You just finished your apprenticeship now you are applying for a job	***Pre-Condition*** You just finished your apprenticeship with BMS now you want to continue to study
Description You choose the job-path and get redirected to the job-menu	***Description*** You choose to apply for a college and get redirected to the college menu
Post-Condition You choose your job type and fill out the resume	***Post-Condition*** You fill out your college resume and according to your answers you'll get feedback of your acceptance rate
Pre-Condition You just finished your apprenticeship without BMS now you are applying for a technical college	***Pre-Condition*** You just finished your resume for university application
Description The menu of the technical college pops up and asks you for your previous employment direction	***Description*** According to your answers the resume gets compared
Post-Condition You get a bunch question to fill out the resume and get your acceptance rate as a result	***Post-Condition*** You get a percentage of your chances to get accepted in the university
Pre-Condition You want to see all colleges	***Pre-Condition*** You want to see all jobs from the different job directions
Description You choose from the main menu to see all colleges	***Description*** From the main menu you choose to see all jobs
Post-Condition All colleges available are shown	***Post-Condition*** From the main menu you choose your preferred profession and you're able to see their direction

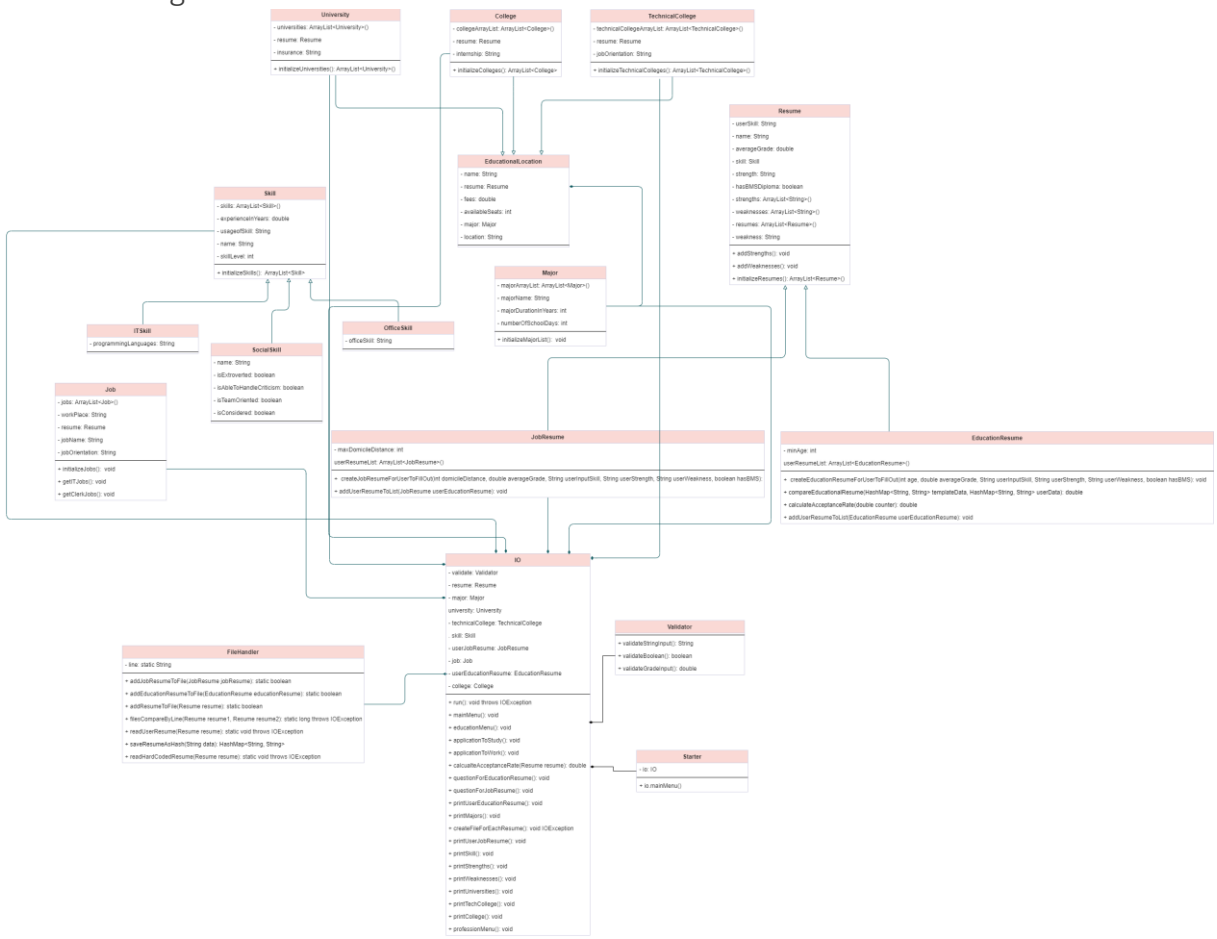
Class Diagram

Rough design



This is our rough design of the class diagram for our program. As you can see, we didn't add much attributes or methods, with the purpose to get a quick overview of all the classes, we need and their relation. Down below you will see a more detailed version of our class diagram and the supplemented diagram to get a better understanding and more clear vision of our idea. The purpose of this project is to implement inheritance as we learned it in this module.

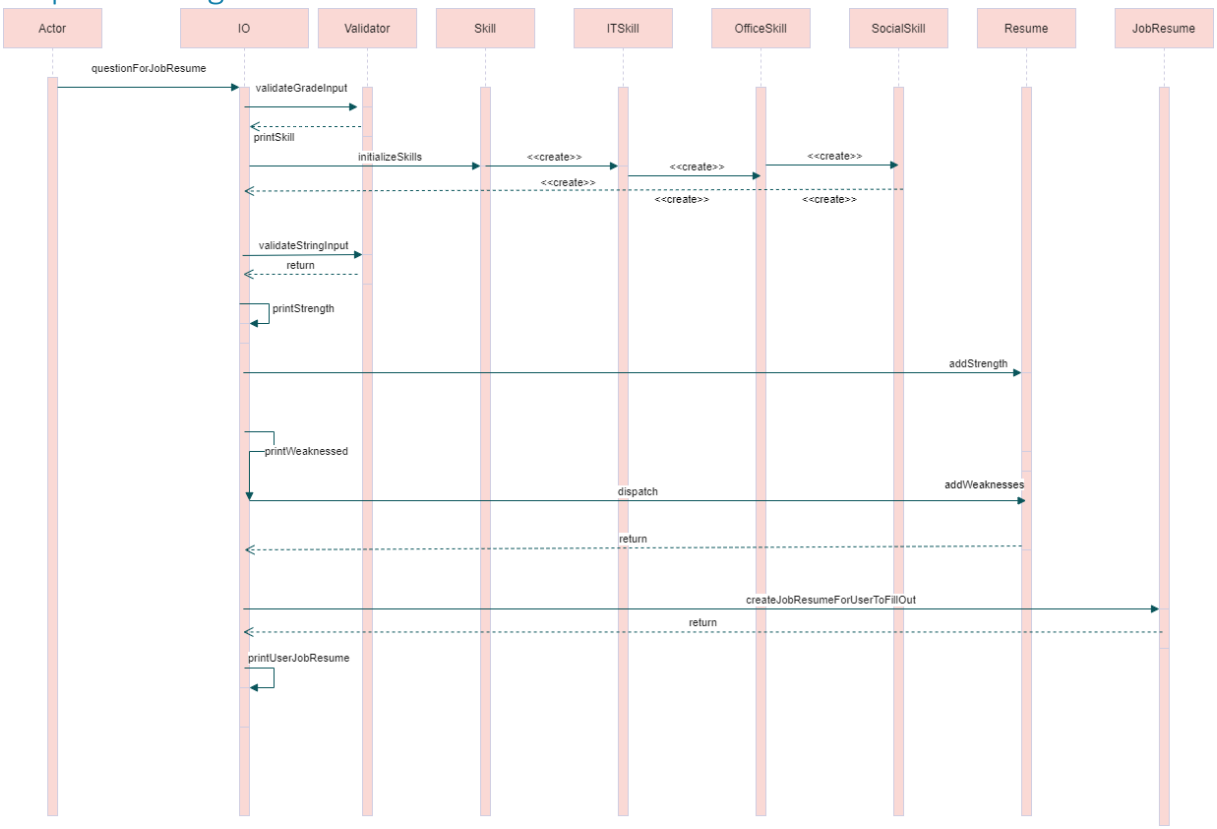
Detailed diagram



This is the detailed version of our class diagram. Compared to the old one we made a few changes. This demonstrates the structure of our program. We used inheritance multiple times as you can see e.g

OfficeSkill extends from Skill. Through the arrows you can clearly see where inheritance is used. Mosty the relationship is a “has-a” and we split the logic with the IO class.

Sequence Diagram



This is a Sequence Diagram for one use case specifically when the user gets asked for the job resume. It demonstrates step by step each sequence showing when which specific method gets called.

Diary

We decided to keep a journal during the project to keep track of our goals but also to note our progress. This provides an overview of the whole project for a better and more structured work.

04.01.2022

GOALS

- Search for project ideas
- Start with documentation, define goals for each week

Today we started looking for a project idea. The teacher has also provided some ideas however, we wanted to be creative and look for our own idea that has a personal connection with us.

11.01.2022

GOALS

- Finalize project idea
- Write use cases
- Start class diagram

Today we have finalized our idea and started to write the use cases. We have been thinking about what our program can do. Then we started to consider what classes we need and where inheritance is needed. We got feedback from the teacher and made some minor changes.

18.01.2022

GOALS

- Start with basic code structure
- Make necessary changes in class diagram

Today we started with the basic structure and created all classes. The class diagram was very roughly designed, so we had to think about which attributes would make sense.

20.01.2022

GOALS

- Work on Code
- Have a working orientation

We met outside of school and worked on the code. Our program requires predetermined data, so we created the necessary hardcoded item. For the orientation we created an initial menu and chose different options.

25.01.2022

GOALS

- Create methods to fill out resumes

Today we worked well. We have started to create resumes as well as the user can fill them out depending on the selection for his path.

27.01.2022

GOALS

- Validation of Code
- Start writing Javadoc
- Create sequence diagram

For the Code on Tuesday, we started to validate the methods. For this we created a Validation Class where we wrote a method which is used quite often for user input. Mostly try and catch blocks were used to catch possible errors e. g InputMissmatchException. It's important to document our Code for which we started describing every method, class in our Project with javadoc.

29.01.2022

GOALS

- Improve Code
- Work on documentation
- Finish javadoc
- Created final and detailed version of Class diagram

The weekend before the deadline, we made a few last changes in the code and supplemented if necessary.

30.01.2022

Last day before handing our project we just made sure that everything works and did some final touch ups.

Practical Implementation

Code Example

Encapsulation (Hiding information)

```
private String name;
private String location;
private Major major;
private double fees;
private int availableSeats;
private Resume resume;
```

```
public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getLocation() { return location; }

public void setLocation(String location) { this.location = location; }

public Major getMajor() { return major; }
```

Here is how we worked with Encapsulation. This is an Example of the EducationalLocation Class (parent Class of other classes). Every field is private, to have access on the fields we created getters and setters for it. The child Classes used super () in the constructor to have access.

Inheritance

Used in Code

Inheritance in Java means that you can create new classes which build on existing classes. By inheriting from an existing class, you can reuse its methods and fields as well as add your own. That's exactly what we did! For inheritance we have two different parent Classes (Skill and Educational Location), you can see them down below.

```
public class EducationalLocation{
    private String name;
    private String location;
    private Major major;
    private double fees;
    private int availableSeats;
    private Resume resume;

    /**
     * Common attributes in the constructor which
     * the child classes will be using
     * @param name of the Educational Place
     * @param location of the Educational Place
     * @param major of the Educational Place
     * @param fees of the Educational Place
     * @param availableSeats at the Educational Place
     * @param resume requirements of the Educational Place
     */
    public EducationalLocation(String name, String location, Major m
        this.name = name;
        this.location = location;
        this.major = major;
        this.fees = fees;
        this.availableSeats = availableSeats;
        this.resume = resume;
    }
```

```
/**
 * This is a super class defining
 * attributes for child classes
 * to use
 */
public class Skill {
    private String name;
    private double experienceInYears;
    private String usageOfSkill;
    private int skillLevel;
    private  ArrayList<Skill> skills = new ArrayList<>();

    /**
     * Common attributes in the constructor which
     * the child classes will be using
     * @param name of the skill
     * @param experienceInYears of the skill
     * @param usageOfSkill where the skill is used
     * @param skillLevel from a scale of 1 to 10, how good are you
     */
    public Skill(String name, double experienceInYears, String usageOfSkill, int skillLevel
        this.name = name;
        this.experienceInYears = experienceInYears;
        this.usageOfSkill = usageOfSkill;
        this.skillLevel = skillLevel;
    }
```

This are some general attributes we defined, that we thought make sense to summarize in one class. Such as for Educational Location we thought of three different school types. → College, Technical College and University. In Skill we thought of three different types of skills one needs →Social Skills, IT Skills and Office skills such as usage of Word or Excel (used later for job)

```java
public class University extends EducationalLocation{

    /**
     * additional attributed for the university
     */
    private String insurance;
    private ArrayList<University> universities = new ArrayList<>();
    private Resume resume = new Resume();

    /**
     * Constructor to create a university, inheriting attributes
     * from EducationalLocation through super()
     * @param name of the university
     * @param location of the university
     * @param major the university offers
     * @param fees how much the student would pay for his studies
     * @param availableSeats how many seats are available
     * @param resume requirements of the university to get accepted
     * @param insurance to study in Switzerland the Student must have a insurance
     */
    public University(String name, String location, Major major, double fees, int
        super(name, location, major, fees, availableSeats, resume);
        this.insurance = insurance;
    }
```

This is an example of a child class from Educational Location. As you can see University is extending from it and we also added additional attributes. The goal is to have pre-defined universities from which we will be comparing the resumes from the user.

```java
public class OfficeSkill extends Skill{

    private String officeSkill;

    public OfficeSkill(String name, double experienceInYears, String usageOfSkill, int
        super(name, experienceInYears, usageOfSkill, skillLevel);
        this.officeSkill = officeSkill;
    }
}
```

This is an example of a child class from Skill. As you can see here as well, we added additional attributes and the rest is inherited through super ().

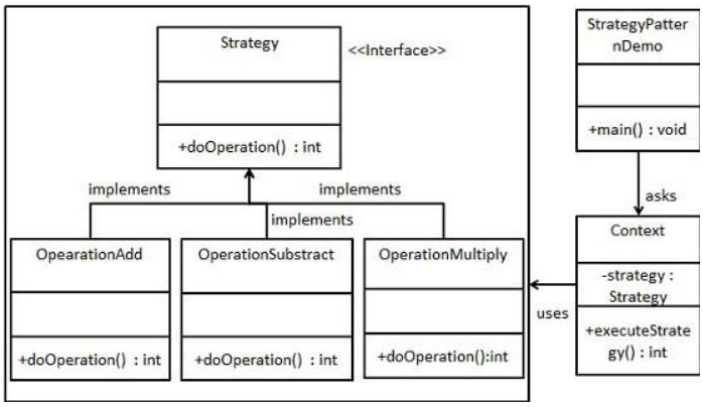
Why didn't we use Abstract Classes or any Interface?

An Interface didn't make much sense for our program because we mostly used common attributes than methods. As we know attributes in an interface are always per default public and **final**, which must be initialized.

At first, we thought that abstract classes would make sense in our program but soon we changed our minds. Abstract classes can be extended but we can't create an instance of it. This was our main problem, why we had to remove it. Such as Skill was planned at first to be abstract but later on, we decided to save them in an Array List for which we added them in a separate method.

Design Patterns

Strategy Pattern



This is how the Strategy Pattern is built. At first, we thought we could use this Pattern in our project according to our rough Class diagram design thinking Resume would make sense as an interface. This changed soon because we couldn't think of many methods that we could use in the child classes. Even though the resume varies depending on school type and work.

Problems

We had a few problems while working on this project. The validation was sometimes hard or the merge conflicts, which we surprisingly solved very fast. We also had to deal with some Scanner problems by creating multiple scanner instances. The reason for that was to prevent the scanner to skip questions because there was already an input saved in.

We thought saving different resumes in a file would be kind of a piece of cake. It took us some time to get back into this topic, since we learned it in module 411. After that it was not that hard, we just needed a little time to understand how to create a separate file for each element of the array list.

Testing

Exception-Handling

After having our own validation class, we had to make sure that every case of a DAU is handled. We mostly worked with a try and the caught the certain exception in the catch-block. Here is an example of that:

```
try {
    answer = Integer.parseInt(input.nextLine());
} catch (InputMismatchException | NumberFormatException e) {
    System.out.println("\u001B[31mInvalid answer\u001B[0m");
    System.out.print("> ");
    answer = Integer.parseInt(input.nextLine());
}
```

Mostly the *InputMissmatchException* and *NumberFormatException* were used in our program for the different menus and their switch cases. If the user happens to enter a number or a random character, we will catch and point it out for the user.

Own Validation

In our program we ask the user for quite bit of input. Though there are pre-written exceptions we still thought about having our own for the resume. For this we created a new Class called Validator.java in which we wrote three different methods to check for user Input.

Here is an example:

```
public String validateStringInput() {
    Scanner scan = new Scanner(System.in);
    Pattern p = Pattern.compile("[^a-z]", Pattern.CASE_INSENSITIVE);

    String input = scan.nextLine();
    Matcher m = p.matcher(input);
    boolean hasCharacter = m.find();

    while (hasCharacter) {
        System.out.println("\u001B[31mInvalid answer, your answer can
only contain letters\n");
        System.out.print("Try again: \u001B[0m");
        input = scan.nextLine();
        m = p.matcher(input);
        hasCharacter = m.find();
    }
    return input;
}
```

This method checks for only String inputs. It's used in our IO.java Class (Input / Output). We defined a pattern that only allows characters not checking if upper or lower case. The method will keep asking for user input until it's satisfied with the answer he gives, to be precise if the input matches the pattern.

```
public String validateStrengths() {
    Scanner scan = new Scanner(System.in);
    ArrayList<String> listOfStrengths = new
ArrayList<String>(Arrays.asList("caring","self-controlled", "motivated",
"fair", "dedicated" ));
    String input = scan.nextLine();
    while (!listOfStrengths.contains(input.trim())) {
        System.out.println("\u001B[31mInvalid answer, not on the strength
list\n");
        System.out.print("Try again: \u001B[0m");
        input = scan.nextLine();
    }
    return input;
}
```

This method checks for only inputs which contain a strength from the options we give through the list. It's used in our IO.java Class (Input / Output). The method will keep asking for user input until it's satisfied with the answer he gives, to be precise if the list contains the input.

```
System.out.print("What is your biggest strength (choose 1) \n");
printStrengths();
System.out.print("> ");
String userStrength = validate.validateStrengths();
```

Such as here, the user is being asked for his strength, for what we only accept an answer from the list we give as an option. If the user chooses to write something else, he will be asked again.

JUnit Tests

For the actual testing of the project, we worked with JUnit. This framework lets us create automated tests for regression testing.

Here are some of the examples we did:

```
@BeforeEach
void setUp() {
    resume.initializeResumes();
    College zhaw = new College("ZHAW", "Zuerich", new Major("Computer
Science", 4, 6), 5000, 20,  resume.getResumes().get(5), "Software
Developer" );
    College hwz = new College("HWZ", "Zuerich", new Major("Business", 3,
3), 6500.50, 12,  resume.getResumes().get(6), "Retail Manager");
    collegeArrayList.add(zhaw);
    collegeArrayList.add(hwz);
}
```

So first we are adding default data to the arraylist. This step is called the setup. Here as you can see, we added two colleges to the list and filled their attributes.

Now:

```
@Test
void getCollegeArrayList() {
    assertThat(collegeArrayList.get(0).getName(),
collegeArrayList.get(0).getName().equals("ZHAW"));
    assertThat(collegeArrayList.get(1).getInternship(),
collegeArrayList.get(1).getInternship().equals("Retail Manager"));
    Assert.assertNotSame(collegeArrayList.get(0),
collegeArrayList.get(1));
    Assert.assertNotEquals(collegeArrayList.get(0),
collegeArrayList.get(1));
}
```

Now that we have our list which is not supposed to be empty, we should verificate that too. At the first line we are checking whether the name of the first item is ZHAW or not. The second, I get the internship of the zhaw college to make sure that the internship in zhaw would not be as a Retail Manager → assertNotSame().