M1IF01 cv-search project report

Mohammed Kais EL HADJ MUSTAPHA, p2113563

Introduction to cv-search

CV-search is a desktop application that allows users to filter out Resume or CV file on a database using multiple criteria of search.

The database consists of .yaml files, where each file represents a CV of a certain individual, which contain info about the person's name, skills and professional experience.

CV-search permits users through data input to select samples from the database that fit the desired profile.

To use the application, the user could modify the settings the search to fit the searched profile through:

- Adding skills
- Selecting a search strategy
- Adding professional experience at a certain company
- Adding the total years of professional experiences

Features

1- Skills:

Simply enough, the user could add skills through the skill bar then click the ‘add skill’ button.

It's been made sure that identical skills added will lead into treating only one and that all values entered into the search bar are treated after being converted to lowercase.

2-Strategies:

A total number of five search strategies were added, which are:

- **None**: indicates the absence of a search strategy.

- **All >= 50**: allows the user to filter out the applicants that have a score higher than 50 for all the searched skills, otherwise the applicant will be ignored.

- **All >= 60**: same as the previous one, but the skill score is 60.

- **Average >= 50**: allows the user to filter out the applicants with an average score that is higher than 50 for the demanded skills.

- **Harmonic mean >= 50**: Harmonic mean is an average value that is calculated in a different way than an arithmetic mean, it helps to detect whether there are outliers in the used dataset, I.e.

The value of the harmonic mean will almost always be lower than the value of the arithmetic mean if applied on the same dataset, by leaning towards the lowest value in the dataset.

Formula: $n / (\sum 1/x_i)$

Where:

- n – the number of the values in a dataset
- x_i – the point in a dataset

3- Professional experience(s):

This feature allows the user to enter a certain company(s), where the searched applicant(s) must've worked before. The user could as well add a duration of the desired working period at the company.

4- Years of professional experience:

The user could also specify a total number of years that the desired applicant must've spent working in a professional environment.

5- **Results**:

After the user had entered the needed profile and clicked the search button, the application will filter out the applicants based on the input data, and will show the applicants that fitted the profile by displaying the applicant's:

- Name
- Skill average score based on the selected search strategy
- Skills
- Professional experiences

The list of the displayed applicants will be ordered by the skills average score.

Design Patterns:

- **MVC**: the MVC pattern was implemented by using multiple packages containing several classes as follows

* View logic:

Had one package ' fr.univ_lyon1.info.m1.cv_search.view', which contained one class 'View.java'.

The View is obviously responsible for the view logic, in this implementation of MVC, the view class communicates with the model through the 'controller.java' class.

* Controller:

Had one package as well 'fr.univ_lyon1.info.m1.cv_search.controller', and contained one class 'Controller.java' (façade).

connects the frontend and the backend through the use of multiple pattern designs.

* Business logic:

Had several packages:

- 'fr.univ_lyon1.info.m1.cv_search.model.applicant', for applicant handling
- 'fr.univ_lyon1.info.m1.cv_search.model.skill', for skill handling
- 'fr.univ_lyon1.info.m1.cv_search.model.strategy', for strategy handling
- 'fr.univ_lyon1.info.m1.cv_search.model.experience', for experience handling
- 'fr.univ_lyon1.info.m1.cv_search.model.observer', for the observer design
- 'fr.univ_lyon1.info.m1.cv_search.model.factory', for the factory design

responsible for the backend logic, responds to the view requests through the controller by manipulating the data.

- **Observer**:

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

the observer pattern is used here to implement having one model-end or one backend object and potential multiple identical or synchronized views.

Since all views are connected to the same model, any action on any interface (view) will lead into changes in the model and consequently, changes on all interfaces.

Considering that we're implementing the MVC pattern mainly, the observer pattern would be a perfect fit for interface synchronization.

Classes concerned by this pattern:

- **fr.univ_lyon1.info.m1.cv_search.model.observer.ViewObserver.java**
- **fr.univ_lyon1.info.m1.cv_search.controller.Controller.java**
- **fr.univ_lyon1.info.m1.cv_search.view.View.java**

- **Factory**:

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

the factory design pattern allows to group objects instantiations in one class, and thus facilitating exploitation of the application's components.

In this case, with the use of multiple model classes, adding the factory pattern helps a lot.

Classes concerned by this pattern:

- **fr.univ_lyon1.info.m1.cv_search.model.factory.ListFactory**
- **fr.univ_lyon1.info.m1.cv_search.model.factory.ModelFactory**

- **Façade**:

this design pattern is one of many other patterns that could've been used to implement the controller part of MVC, it creates one single controller object that receives all requests from view classes and redirects them to the corresponding model classes.

Classes concerned by this pattern:

- **fr.univ_lyon1.info.m1.cv_search.model.controller.Controller.java**

- **Delegation**:

this design pattern was implemented to stick to the SRP principle as much as possible, by creating a model class for each group of related functionalities, and thus assigning that model class to one responsibility.

The requests will be coming from the View class to the controller, which will delegate them to primary model object, and this latter will delegate the request to the suitable model object to handle it.

Classes concerned by this pattern:

- **fr.univ_lyon1.info.m1.cv_search.model.controller.Controller.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.model.IModel.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.model.applicant.ApplicantModel.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.model.experience.ExperienceMode.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.model.skill.SkillModel.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.model.strategy.StrategyModel.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.model.Model.java**

- **Strategy**:

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

This design pattern was considered a good fit, since we have multiple search strategies to implement.

Classes concerned by this pattern:

- **fr.univ_lyon1.info.m1.cv_search.model.controller.strategy.IStrategy.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.strategy.StrategyAll.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.strategy.StrategyAverage.java**
- **fr.univ_lyon1.info.m1.cv_search.model.controller.strategy.StrategyHarmonic.java**

Ethics:

In terms of the ethical aspect for this project, many considerations were taken to make the best decisions concerning people, resources and environment.

As a starter, the data collected about the individuals to fill the database will contain only career information and nothing about their personal lives other than their names.

The search criteria were made to be as professional as possible and fully transparent by excluding the names, genders, ethnicities of the applicants and focusing only on their skill and experience.

Tests

Model Tests

testNone() => testing the 'None' selection strategy by confirming that all applicants are added.

testAvg50() => testing the 'Average>= 50' selection strategy by confirming that all applicants have an average skill score greater or equal to 50.

testAll50() => testing the All>= 50' selection strategy by confirming that each applicant has a set of skills with values higher than 50.

testHrmnc50() => testing the ‘Average>= 50’ selection strategy by confirming that all applicants are have a harmonic average skill score greater or equal to 50.

testAddApplicant () => testing the ‘add applicant’ functionality by measuring the length of the applicant list before and after executing the function.

testdeleteApplicant () => testing the ‘delete applicant’ functionality by measuring the length of the applicant list before and after executing the function.

testaddAndRemoveSkill () => testing the ‘add skill’ and ‘remove skill’ functionalities by using the length of the skill list before and after executing the function.

TestaddAndRemoveExperience () => testing the ‘add experience’ and ‘remove experience’ functionalities by using the length of the experience list before and after executing the function.

testingInvaldSkills () => testing invalid entered skills treatment by treating them simply as missing skills in the applicants' profiles, and giving them a score of 0.

testingIdenticalSkills () => testing handling identical multiple data input, by running some calculations and showing that each skill is processed only once.