PROO/SEW III - Assignment: Recruiting Company

Objective

Simulate a recruiting company contacting possible applicants using the *Observer pattern*.

Things To Learn

- Creating and using a *Maven* project.
- Implementing the *Observer* and *Factory* patterns.
- Importing, configuring and logging with *Log4j 2*.

Submission Guidelines

- Your implemented solution as ***zipped*** *IntelliJ*-project.

Task

In the ever expanding *IT* job market, both businesses and job seekers use recruiting companies and online platforms to find suitable workplaces.

For this assignment you'll simulate a `RecruitingCompany` sending `JobOffer`s to possible `Applicant`s. Both the provided and required `Skill`s as well as the salary expectations need to match for a a possible employment!

The program should consist of the following parts, listed in the suggested order of implementation:

1. `JobOffer`
   - Besides an *ID* - that is ***used for establishing equality*** - a description of the position and the name of the company, the required `Skill`s are listed in a *set*.
     - The required skills should be *read-only* - make sure they can't be overwritten using the *getter*!
   - Additionally, two properties define the possible *pay range*.
     - If no *maximum pay* is specified, it is 1.5 times the *minimum pay* by default.
     - The maximum pay must be higher than or equal to the minimum pay, and the minimum pay must be above 1614€ (according to the Austrian *Kollektivvertrag*). Otherwise, a built-in `IllegalArgumentException` should be thrown.
   - A *boolean flag* describes if a company paid a *premium* fee for the offer to be ranked higher.
   - Each job offer is `Comparable`: The sort order is defined by the max pay - **but all *premium* offers come first!**

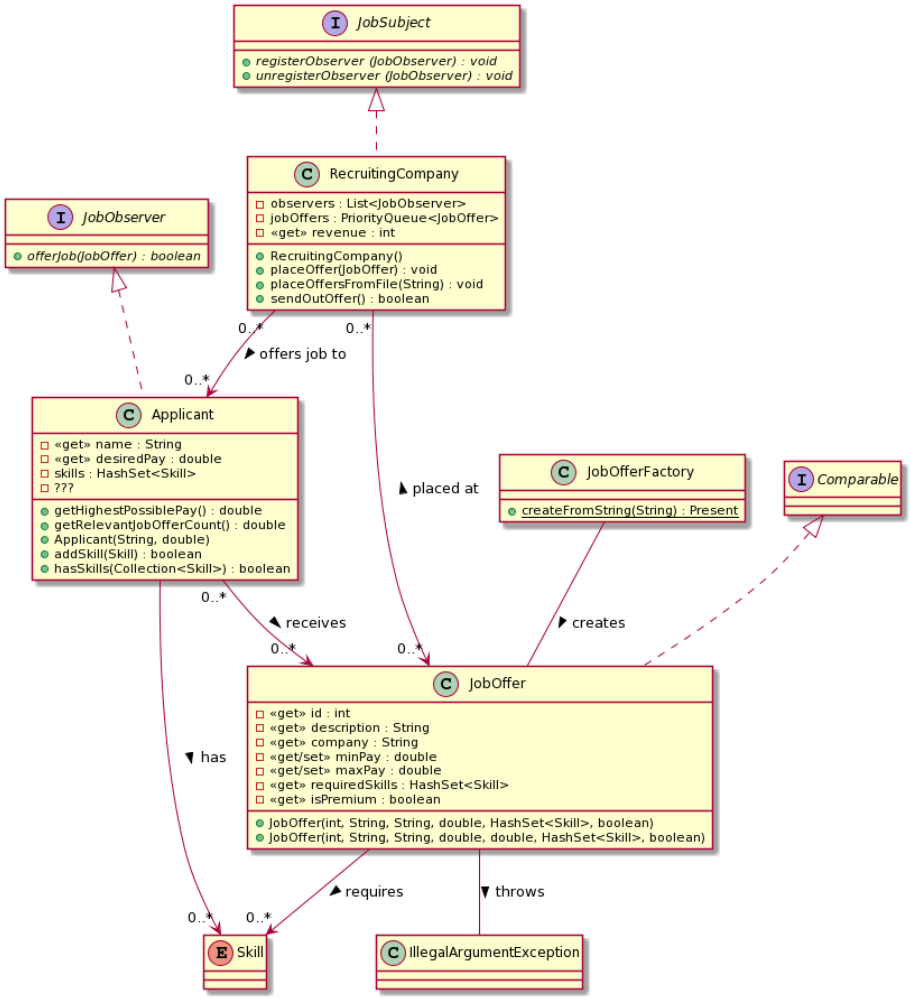

Figure 1: Class Diagram.

2. `JobOfferFactory`
   - A *static factory method* allows the easy creation of job offers given a *semicolon-separated string*.
   - Make sure to *strip* both leading and trailing whitespaces.
     - Luckily, the data doesn't seem to be corrupted otherwise.
   - The *ID* needs to be extracted from the *URL* of the offer on a popular Austrian job portal.
   - The required skills are separated by *commas*.
     - Hint: Similar to the *wrapper classes* `Integer` or `Double`, each `enum` also provides a `valueOf`-method allowing the parsing of a string value.
3. `Applicant`
   - An *applicant* has a name, a desired pay and a *set* of `Skill`s.
     - Skills can be added using the `addSkill`-method, that returns `false`, if the skill is already present.
     - Using the `hasSkills`-method one can check, if an applicant has ***all*** skills contained in a `Collection<Skill>` parameter.
   - Each applicant implements the `JobObserver`-*interface*, allowing us to `offerJob`s that could be interesting for them.
     - A job offer is relevant for an applicant, if...
       - ...the desired pay is not higher than the maximum salary offered.
       - ...the applicant provides ***all*** skills that are needed for the job.
     - A *boolean* return value describes if the job offer was relevant.
   - The applicant counts the number of relevant job offers and the highest possible pay they can reach. The details of the implementation are up to you!
4. `RecruitingCompany`
   - The *recruiting company* is the *subject* in our *observer pattern*. Thus it allows us to *register* and *unregister* `JobObserver`s, that are notified when a job offer is *sent out*.
   - Companies can `placeOffer`s (i.e. *add*) that are stored in a `PriorityQueue`. Of course the *recruiting company* gets payed for its work:
     - Placing a *premium* offer costs 1000€.
     - A *normal* offer costs 500€.
   - Multiple offers can be placed at once using the `placeOffersFromFile`-method, that accepts a path to a `.csv`-file like the one provided.
   - The `sendOutOffer`-method removes the *top* offer from the queue and *sends* it to all job seekers.
     - The recruiting company receives a bonus of 200€ **for each applicant, that is interested in the offer.**
     - The method returns `false`, if there were no more offers to process.
5. *Logging*
   - Use *log4j2* to implement logging wherever it makes sense.

- Play around with different *logging levels* to enable the distinction between the importance (or *severity*) of various events - from `FATAL` for the most critical errors up to `TRACE` for e.g. *parsing* details.