Employee Workshop (Simplified Version)

Dr. Matthias Hölzl

January 18, 2022

Goal

Make a small program conform to the SOLID and GRASP guidelines, in particular to the SRP.

Notes

- The code is in `Workshops/EmployeeSimplified`, a possible solution is in `Completed/EmployeeSimplifiedCompleted`.
- To simplify the code, most classes are defined as structs with publicly accessible members. While you should obviously not do this in production code, you can follow the same style during the refactorings to reduce the amount of boilerplate code you need to write.
- Add constructors to your structs, otherwise `std::make_unique()` might not work.
- You don't have to keep interfaces stable when refactoring, except for the interface to the `AugurDB` database.
- **There is no single 'correct' solution.** You don't have to follow the steps proposed in this outline; they are just there to help you if you get stuck. Try to follow your own ideas about refactoring the code and see how it turns out.
- **Refactor in small steps. Try to keep your code buildable and running as long as possible.**

Step 1: Identify Reasons to Change

- What reasons to change does the `Employee` class have?
- What strategies can you use to resolve them?
- Try to refactor the code using the strategy you have identified. If you are uncertain whether you're on the right track you can look at the following steps to see a possible way forward.

Step 2: Extract a `ReportPrinter` interface

The probably easiest class to extract is one that takes care of printing Employees. Extract a `ResourcePrinter` interface and a concrete implementation that prints the data to an output stream.

Step 3: Extract a `PaymentScheme` interface

Since payment and work hours are closely related, it might be useful to extract them together. Define a `PaymentScheme` interface that provides the necessary member functions to deal with these aspects.
Implement subclasses for the various kinds of employees that store the required data; this allows you to remove a lot of the data from the `Employee` class and clean up the usage of variables for multiple unrelated uses.
To save to the database you will probably need to provide functionaliyt in `PaymentScheme` that supports this. This is not strictly conforming to the clean code rules, but it is a good compromise to implement a 'quick and dirty' solution for this right now, since you need to address saving to the database anyway, and any proper solution you implement right now would have to be changed anyway.

Step 4: Extract an `EmployeeDao` interface

- Extract an interface that can be used to save employees to a database.
- Implement the interface for the Augur database.
- *Note:* In a realistic scenario you would probably create a factory that creates the correct DAO based on the database you provide. To simplify the example, you can simply hard-code the creation of an `AugurDbEmpleyeeDao` whenever you need a DAO.
- There is no completely clean way to implement saving the `PaymentScheme` subclasses to the database in C++. You either have to add a member function to the `PaymentScheme` subclasses that provides the data in a standardized format, or you have to use a case distinction with `dynamic_cast<>()` to identify the subclass of `PaymentScheme` you are currently trying to save.

Step 5: Admire your clean `Employee` class

Congratulations! You now have an `Employee` class where all reasons to change are delegated to an interface that serves exactly that purpose!