# Case Study

## Overview

In this case study we’ll see how to implement a simple Human Resources (HR) system. In the HR system, employees can have one of the following roles:

* Developer
* Senior Developer
* Manager
* Partner
* CEO

An employee has a base salary, plus a bonus that depends on his or her role, as follows:

* Each role has its own specific *salary multiplier*. For example, a Senior Developer might have an effective salary of 1.2 \* base salary.
* Each role also has its own *flat bonus* that is just added to the base salary. For example, a Senior Developer might have a flat bonus of £2000.

An employee can get promoted from one role to another. The employee’s details (name and base salary) remain the same, but the rules for calculating the effective salary will change.

You will implement the system in an extensible manner, so that new kinds of roles can be added in the future. It's also important that you encapsulate the rules about role transitions in a single place, rather than each specific role knowing what comes next in the promotion chain.

This exercise will reinforce many principles of encapsulation, type safety, and SOLID design that we’ve been discussing during the course.

## Roadmap

1. Defining a Role class
2. Defining a RolePolicy class
3. Defining an Employee class
4. Writing client code
5. Additional suggestions (If time permits)

## Exercise 1: Defining a Role class

## Take a look at the files in the Starter folder for this case study. The folder contains starter Python files for all the classes to be implemented in this case study.

## To get things going, let’s complete the definition of the Role class in role.py. The purpose of the Role class is to represent a single role. If the system needs 5 roles, then the system will simply create 5 Role objects.

## Each Role object needs to hold the following information about a role, which you should set in the \_\_init\_\_() method (make sure all the data in this class, and all the other classes in this case study, is private):

* The name of the role, e.g. "Senior Developer"
* The salary multiplier for that role
* The flat bonus for that role

*Aside:*

*You might have expected to use inheritance here, with a bunch of subclasses named something like DeveloperRole, SeniorDeveloperRole, ManagerRole, PartnerRole, and CEORole.*

*However, inheritance is not the correct approach here. The only difference between the roles is that each role has different data values – there’s no different functionality for each role. In such scenarios, you should just define a single Role class, and then create multiple objects to hold the different data for each role.*

*(You would only need inheritance if there was different functionality for each role).*

The Role class should define a method named get\_effective\_salary(). The method needs a single parameter specifying an employee's base salary, and should return their effective salary after applying the salary multiplier and adding the flat bonus.

The Role class should also define a method named get\_role\_name(), which returns the name of the role. This will be handy for other classes in the system.

*Note*: We've avoided the temptation for each role to know about the next role in the promotion chain. This would cause unnecessary coupling and complication if we needed to add new roles in the future. We'll delegate this knowledge for how to manage role promotions to a dedicated class named RolePolicy, which we’ll implement next. This approach is an example of the Single Responsibility Principle, which is crucial in big systems:

* The Role class just knows about the salary calculation rules for a particular role
* The RolePolicy class will manage the policy for transitioning between roles

## Exercise 2: Defining a RolePolicy class

## You will now complete the definition of the RolePolicy class, in rolePolicy.py. The purpose of the RolePolicy class is to encapsulate all the knowledge about how roles work collectively. It's appropriate to define a separate class for this, rather than putting this knowledge into the Role class (the Role class should just know about one role, not about the interplay between different roles).

Implement the RolePolicy class as follows:

* RolePolicy should be a singleton class (we only need one RolePolicy object in the system, don't we?). You can make use of the @singleton decorator here, which we’ve already provided in singleton.py.
* RolePolicy should create and manage all the Role objects that the system will need. This way, if additional roles need to be added to the system in the future, only the code in the RolePolicy class will be affected. Define an \_\_init\_\_() method as follows:
  + Define an attribute named \_\_roles (for example), to hold a bunch of object references (pointers) to all the Role objects in the system.
  + Populate the list with a collection of Role objects in increasing order of seniority (e.g., a Role for "Developer", followed by a Role for "Senior Developer", and so on). For each role, specify a suitable name (e.g., "Developer"), the salary multiplier, and the flat bonus amount.
* Define a promote() method, to implement a promotion from one role to another. Here are some hints:
  + The method should take a Role object reference as a parameter, indicating an employee's current role.
  + The method should iterate through the collection of roles, to find which Role object in the list corresponds to the employee’s current role (use the Python is operator to compare Role object references here).
  + The method should then return the *next* role in the collection. Think carefully about what to do if the employee is already in the most senior role.
* Define a get\_initial\_role() method that returns a reference to the most junior Role object in the list. This will be handy when you implement the Employee class shortly – it will enable you to create an employee with the "most junior" role initially.

## Exercise 3: Defining an Employee class

## Now it's time to define the Employee class, in employee.py. An employee has a name, base salary, and the associated current role (i.e., a reference to a Role object). Initially, set the current role to the most junior role (as indicated by the RolePolicy class).

## Implement three additional methods in the Employee class:

* get\_effective\_salary()  
  Determines the employee's effective salary, as calculated for the employee's current role.
* promote()  
  Promotes the employee to the next role, according to the promotion policy logic in the RolePolicy class.
* \_\_str\_\_()  
  Returns a string representation of the employee’s details, including the employee's name, effective salary, and current role.

## Exercise 4: Writing client code

## Add code to main.py to create an employee and give him/her several promotions. Output the employee details at each stage, to ensure that the promotion logic (as implemented by the RolePolicy class) is working as expected.

## Exercise 5: Additional suggestions (If time permits)

## Create a collection of employees with various roles. Find the first employee that has the junior role, and promote him/her.