다형성과 위임을 이용한 설계

먼저 상속을 기반으로 하는 다형성을 적용하여 간단한 월급 관리 프로그램을 만든다. 그 다음 클래스의 합성과 인터페이스를 이용하여 그것을 좀 더 유연하고 재사용성이 높은 프로그램으로 변경해본다.

1. 다형성을 적용하지 않은 프로그램

1) 요구사항 정의 및 기본 프로그램 구현

회사 내의 직원들의 월급을 관리하며 현재는 3명의 직원만 있다. 3명의 직원 중에서 두 명은 평사원이며 나머지 한 사람은 관리자다. 이들의 현재 월급은 평사원은 100만원, 관리자는 200만원이다. 이 프로그램의 기능은 그들의 월급을 현재의 월급에서 10%만큼 인상하는 것이며 관리자는 평사원과 달리 20만원을 더 올려주는 것이다. 중요한 것은 앞으로 이 회사에 더 많은 직원들이 채용될 것이며, 그들의 월급 역시 관리되어야 한다는 것이다. 따라서 이런 미래의 변화에 대비하여 유연하게 프로그램을 설계해야 한다.

먼저 평사원과 관리자에 해당하는 MereClerk 클래스와 Manager 클래스를 작성한다.

```
class MereClerk {
	String name;
	double salary;

	public MereClerk(String name, double salary) {
		this.name = name;
		this.salary = salary;
	}

	public String getName() {
		return this.name;
	}

	public double getSalary() {
		return this.salary;
	}

	public void manageSalary(double rate) {
		salary = salary + salary * (rate / 100);
	}
}

class Manager {
	String name;
	double salary;
```

```
        public Manager(String name, double salary) {
                this.name = name;
                this.salary = salary;
        }

        public String getName() {
                return this.name;
        }

        public double getSalary() {
                return this.salary;
        }

        public void manageSalary(double rate) {
                salary = salary + salary * (rate / 100);
                salary += 20; // 20만원을 추가로 받는다.
        }
}
```

이렇게 작성된 평사원과 매니저객체를 생성하고 월급을 계산하는 클라이언트 프로그램을 다음과 같이 구현한다.

```
public class FlexibleCompany {
        public static void main(String[] args) {
                MereClerk mereClerk1 = new MereClerk("철수", 100);
                MereClerk mereClerk2 = new MereClerk("영희", 100);
                Manager manager = new Manager("홍길동", 200);

                System.out.println("현재 월급입니다.");
                System.out.println(mereClerk1.getName() + "의 현재 월급은 " +
mereClerk1.getSalary() + " 만원 입니다.");
                System.out.println(mereClerk2.getName() + "의 현재 월급은 " +
mereClerk2.getSalary() + " 만원 입니다.");
                System.out.println(manager.getName() + "의 현재 월급은 " +
manager.getSalary() + " 만원 입니다.");

                System.out.println("");

                System.out.println("올린 후의 월급입니다.");
                mereClerk1.manageSalary(10);
                System.out.println(mereClerk1.getName() + "의 현재 월급은 " +
mereClerk1.getSalary() + " 만원 입니다.");

                mereClerk2.manageSalary(10);
                System.out.println(mereClerk2.getName() + "의 현재 월급은 " +
mereClerk2.getSalary() + " 만원 입니다.");
```

```
                manager.manageSalary(10);
                System.out.println(manager.getName() + "의 현재 월급은 " +
manager.getSalary() + " 만원 입니다.");

        }
}
```

2) 새로운 종류의 직원 추가

이렇게 구현된 프로그램에서 대학생이 인턴으로 들어왔다고 가정하자. 이 인턴도 직원이기 때문에 월급을 조정할 수 있어야 한다. 인턴은 평사원이나 관리자와 달리 현재의 월급에서 10%를 올린 후, 20만원을 깎은 금액을 지급한다고 가정하자. 프로그램은 다음과 같이 변경되어야 한다.

먼저 인턴에 해당하는 StudentWorker 클래스를 새롭게 추가한다.

```
class StudentWorker {
        String name;
        double salary;

        public StudentWorker(String name, double salary) {
                this.name = name;
                this.salary = salary;
        }

        public String getName() {
                return this.name;
        }

        public double getSalary() {
                return this.salary;
        }

        public void manageSalary(double rate) {
                salary = salary + salary * (rate / 100);
                salary -= 20; // 현재 급여에서 20만원을 빼고 받는다.
        }
}
```

이렇게 작성된 StudentWorker 클래스에는 getName(), getSalary() 메소드가 중복되게 정의되어야 한다. 그리고 이렇게 인턴이 추가된 이후에는 이를 테스트하는 클라이언트 코드도 수정해야 한다.

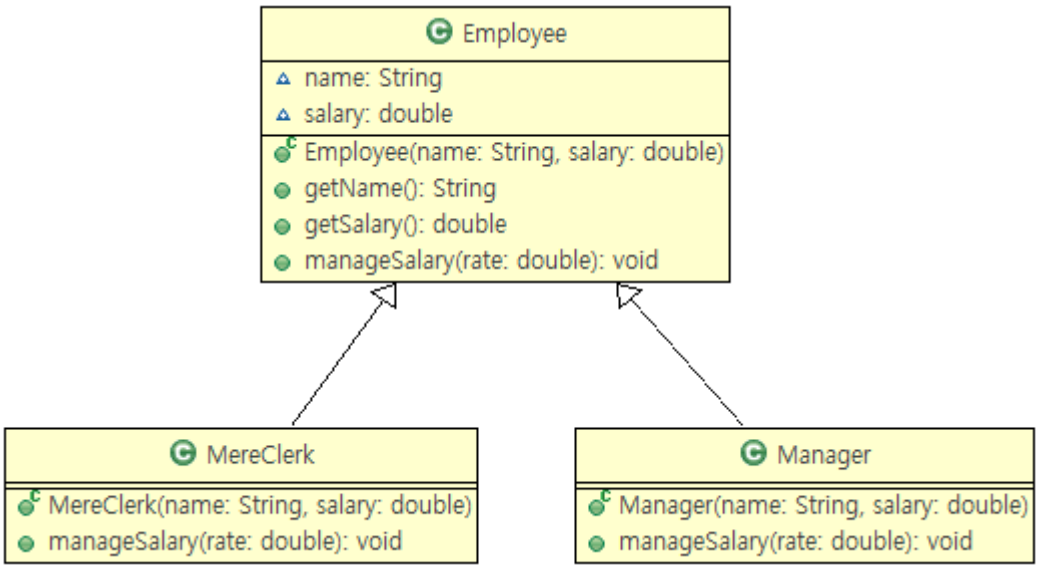
```
public class FlexibleCompany {
        public static void main(String[] args) {
```

```
                MereClerk mereClerk1 = new MereClerk("철수", 100);
                MereClerk mereClerk2 = new MereClerk("영희", 100);
                Manager manager = new Manager("홍길동", 200);
                StudentWorker studentWorker = new StudentWorker("영철", 60);

                System.out.println("현재 월급입니다.");
                System.out.println(mereClerk1.getName() + "의 현재 월급은 " +
mereClerk1.getSalary() + " 만원 입니다.");
                System.out.println(mereClerk2.getName() + "의 현재 월급은 " +
mereClerk2.getSalary() + " 만원 입니다.");
                System.out.println(manager.getName() + "의 현재 월급은 " +
manager.getSalary() + " 만원 입니다.");
                System.out.println(studentWorker.getName() + "의 현재 월급은 " +
studentWorker.getSalary() + " 만원 입니다."); // 인턴

                System.out.println("");

                System.out.println("올린 후의 월급입니다.");
                mereClerk1.manageSalary(10);
                System.out.println(mereClerk1.getName() + "의 현재 월급은 " +
mereClerk1.getSalary() + " 만원 입니다.");

                mereClerk2.manageSalary(10);
                System.out.println(mereClerk2.getName() + "의 현재 월급은 " +
mereClerk2.getSalary() + " 만원 입니다.");

                manager.manageSalary(10);
                System.out.println(manager.getName() + "의 현재 월급은 " +
manager.getSalary() + " 만원 입니다.");

                studentWorker.manageSalary(10); // 인턴
                System.out.println(studentWorker.getName() + "의 현재 월급은 " +
studentWorker.getSalary() + " 만원 입니다.");
        }
}
```

2. 다형성을 적용한 프로그램

1) 프로그램 변경

기존의 프로그램을 상속을 기반으로한 다형성 구조로 변경한다.



먼저 MereClerk과 Manager 클래스의 부모에 해당하는 Employee 클래스를 정의하고 공통의 변수와 메소드를 부모 클래스에 선언한다. 그리고 이를 상속하는 MereClerk과 Manager 클래스를 다음과 같이 구현한다.

```
abstract class Employee {
	String name; // 이름
	double salary; // 월급

	public Employee(String name, double salary) {
		this.name = name;
		this.salary = salary;
	}

	public String getName() {
		return this.name;
	}

	public double getSalary() {
		return this.salary;
	}

	// 추상 메소드로서 rate는 %비율이다.
	public abstract void manageSalary(double rate);

}

class MereClerk extends Employee {
	public MereClerk(String name, double salary) {
		super(name, salary);
	}
```

```
        public void manageSalary(double rate) {
                salary = salary + salary * (rate / 100);
        }
}

class Manager extends Employee {
        public Manager(String name, double salary) {
                super(name, salary);
        }

        public void manageSalary(double rate) {
                salary = salary + salary * (rate / 100);
                salary += 20; // 20만원을 추가로 받는다.
        }
}
```

이제 이를 테스트하는 클라이언트 프로그램은 다음과 같이 다형성을 이용하여 간단하게 수정할 수 있다.

```
public class FlexibleCompany {
        public static void main(String[] args) {

                Employee[] workers = new Employee[3];
                workers[0] = new MereClerk("철수", 100); // 평사원 철수
                workers[1] = new MereClerk("영희", 100); // 평사원 영희
                workers[2] = new Manager("홍길동", 200); // 관리자 홍길동

                // 현재 월급
                System.out.println("현재 월급입니다.");
                for (int i = 0; i < workers.length; i++)
                        // 현재 월급을 프린트한다.
                        System.out.println(workers[i].getName() + "의 현재 월급은 "
+ workers[i].getSalary() + " 만원 입니다.");
                System.out.println("");

                // 현재 월급에서 10%를 올린 월급
                System.out.println("올린 후의 월급입니다.");
                for (int i = 0; i < workers.length; i++) {
                        workers[i].manageSalary(10); // 월급을 10% 올린다.
                        // 현재 월급을 프린트한다.
                        System.out.println(workers[i].getName() + "의 현재 월급은 "
+ workers[i].getSalary() + " 만원 입니다.");

                }
        }
}
```

다형성을 기반으로 구현했기 때문에 새로운 형태의 직원이 추가되는 상황에서 기존의 클라이언트는 거의 수정하지 않는다.

2) 새로운 종류의 직원 추가

StudentWorker 클래스를 기존에 Employee를 상속하여 구현한다.

```
class StudentWorker extends Employee {
	public StudentWorker(String name, double salary) {
		super(name, salary);
	}

	public void manageSalary(double rate) {
		salary = salary + salary * (rate / 100);
		salary -= 5; // 5만원을 빼고 받는다.
	}
}
```

새롭게 추가된 StudentWorker 객체를 사용하도록 클라이언트 프로그램을 수정한다.

```
public class FlexibleCompany {
	public static void main(String[] args) {

		// 아르바이트생이 추가되었기 때문에 종업원 수를 4명으로 한다.
		Employee[] workers = new Employee[4];
		workers[0] = new MereClerk("철수", 100); // 평사원 철수
		workers[1] = new MereClerk("영희", 100); // 평사원 영희
		workers[2] = new Manager("홍길동", 200); // 관리자 홍길동
		workers[3] = new StudentWorker("영철", 60); // 아르바이트생 영철

		// 현재 월급
		System.out.println("현재 월급입니다.");
		for (int i = 0; i < workers.length; i++)
			// 현재 월급을 프린트한다.
			System.out.println(workers[i].getName() + "의 현재 월급은 "
+ workers[i].getSalary() + " 만원 입니다.");

		System.out.println("");

		// 현재 월급에서 10%를 올린 월급
```

```
                System.out.println("올린 후의 월급입니다.");
                for (int i = 0; i < workers.length; i++) {
                        workers[i].manageSalary(10); // 월급을 10% 올린다.
                        // 현재 월급을 프린트한다.
                        System.out.println(workers[i].getName() + "의 현재 월급은 "
+ workers[i].getSalary() + " 만원 입니다.");

                }
        }
}
```

다형성을 이용하면 이렇게 기존의 프로그램 소스를 덜 수정하면서 유지보수할 수 있다.

3. 합성을 이용한 수정

기존의 소스는 Employee 클래스를 하위 클래스들이 상속을 받는다. 이때 getName(), getSalary() 등을 상속받아서 그대로 사용하며, manageSalary() 메소드를 오버라이딩 했다. 상속은 상위 클래스의 속성과 메소드의 구현 코드를 재사용한다는 점에서 장점이 있지만 상위 클래스의 속성과 메소드가 변경되면 상속받은 모든 하위 클래스를 수정해야 하는 일이 발생된다는 단점 또한 존재한다. 또한 하위 클래스의 객체가 생성된 후 그 것을 사용하는 클라이언트도 수정해야 하는 일이 발생한다. 예를 들어 Employee 클래스의 manageSalary() 메소드의 파라미터 타입이 double에서 String으로 변경되면 모든 자식 클래스의 manageSalary() 메소드들도 파라미터 타입을 수정해야 한다. 또한 Employee 클래스의 manageSalary() 메소드를 호출하는 모든 클라이언트 코드도 변경해야 한다. 만약 이런 클라이언트 프로그램이 100개의 파일이 있다면 모든 파일을 찾아 수정해야 하는 것이다.

예를 들어 부모 클래스인 Employee의 manageSalary() 메소드의 매개변수 타입을 double에서 int로 수정하면 모든 자식 클래스의 Overriding된 메소드도 수정해야 한다.

```
abstract class Employee {
        // 추상 메소드로서 rate는 %비율이다.
        public void manageSalary(int rate) {
        }

}

class MereClerk extends Employee {
        public MereClerk(String name, double salary) {
                super(name, salary);
        }
```

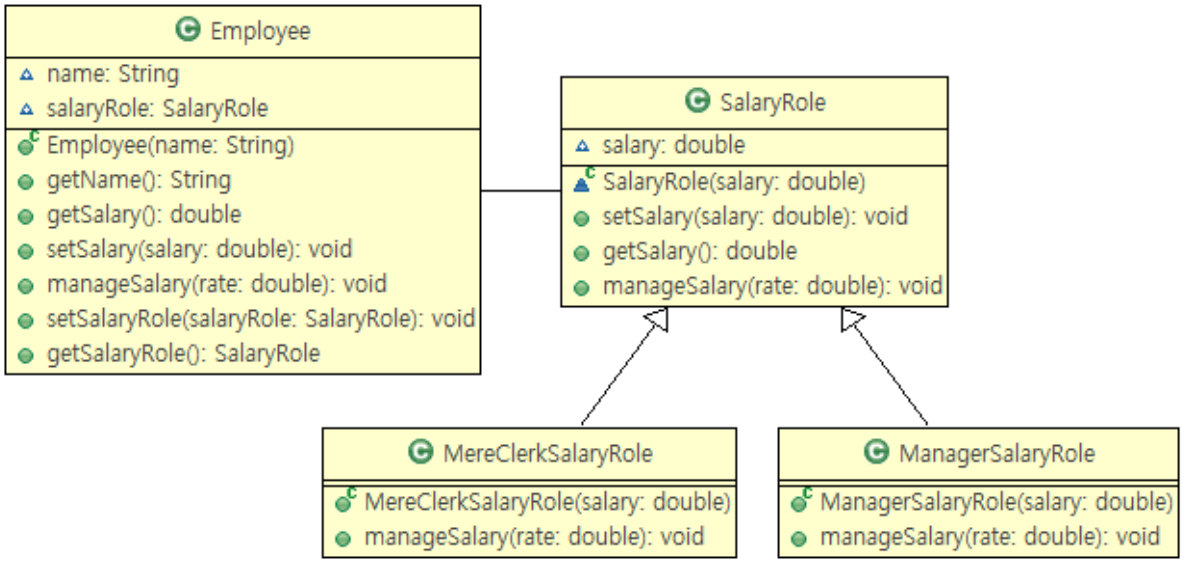
```
        public void manageSalary(int rate) {
                salary = salary + salary * (rate / 100);
        }
}
```

그리고 manageSalary() 메소드의 매개변수 타입을 int가 아니라 String으로 수정한다면 이를 사용하는 클라이언트 프로그램들도 모두 다음과 같이 수정해야 한다.

```
System.out.println("올린 후의 월급입니다.");
for (int i = 0; i < workers.length; i++) {
    workers[i].manageSalary(String.valueOf(10));
    System.out.println(workers[i].getName() + "의 월급 : " +
workers[i].getSalary());

}
```

그런데 현재 예제는 이 외에도 더 큰 문제가 있다. 직원은 시간이 흐름에 따라 인턴이 사원이 되기도 하고 사원이 과장이 될 수도 있다. 즉 직원 객체가 시간이 흐름에 따라 직원의 역할과 관리자의 역할을 교대로 할 수 있도록 하는 것이 나은 것이다. 하지만 상속을 사용하면 모든 직원은 영구적으로 인턴, 사원, 관리자가 되버린다. 왜냐하면 한번 생성된 사원 객체는 나중에 관리자 객체로 변환할 수 없기 때문이다. 형변환은 부모 자식 사이에서만 이루어지면 같은 레벨의 형제 사이에서는 이루어질 수 없기 때문이다.

만약 사원이 관리자가 되고 인턴이 사원이 될 수 있도록 하려면 어떻게 프로그램을 수정해야할까? 이는 상속이 아닌 합성을 이용해야 가능하다.



사원과 관리자는 월급과 관련된 직원의 역할을 가지고 있으며 각 역할을 공통적인 기능을 수행한다. 그것은 manageSalary(), 즉 직원의 역할에 따라서 급여를 조절하는 기능이다. 따라서 SalaryRole이라는 역할과 관련된 별도의 계층 구조를 갖도록 한다. 그리고 Employee는 SalaryRole에게 월급 관리에 대한 권한을 위임한다.

이렇게 합성을 이용하면 시간의 흐름에 따라서 사원이 관리자로 승진한다고 해도 SalaryRole만 교체하면 프로그램이 정상적으로 동작한다. 예를 들어 철수가 사원에서 관리자로 변경되는 상황은 다음과 같이 처리되는 것이다.

workers[0].setSalaryRole(new managerSalaryRole(200));

workers[0].manageSalary(10);

합성을 이용해 변경한 소스는 다음과 같다.

```
class Employee {
	String name; // 이름
	SalaryRole salaryRole = null;

	public Employee(String name) {
		this.name = name;
	}

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

```
        public double getSalary() {
                // SalaryRole 객체에게 위임함.
                return salaryRole.getSalary();
        }

        public void setSalary(double salary) {
                // SalaryRole 객체에게 위임함.
                salaryRole.setSalary(salary);
        }

        public void manageSalary(double rate) {
                // SalaryRole 객체에게 위임함.
                salaryRole.manageSalary(rate);
        }

        public void setSalaryRole(SalaryRole salaryRole) {
                this.salaryRole = salaryRole;
        }

        public SalaryRole getSalaryRole() {
                return salaryRole;
        }

}

/* 직원 역할 클래스 */
abstract class SalaryRole {
        double salary;

        SalaryRole(double salary) {
                this.salary = salary;
        }

        public void setSalary(double salary) {
                this.salary = salary;
        }

        public double getSalary() {
                return salary;
        }

        // 추상 메소드로서 rate는 %비율이다.
        public void manageSalary(double rate) {
        }

}

/* 평사원 역할 클래스 */
class MereClerkSalaryRole extends SalaryRole {
        public MereClerkSalaryRole(double salary) {
                super(salary);
        }

        public void manageSalary(double rate) {
                salary = salary + salary * (rate / 100);
```

```
	}
}

/* 관리자 역할 클래스 */
class ManagerSalaryRole extends SalaryRole {
	public ManagerSalaryRole(double salary) {
		super(salary);
	}

	public void manageSalary(double rate) {
		salary = salary + salary * (rate / 100);
		salary += 20; // 20만원을 추가로 받는다.
	}
}
```

```
public class FlexibleCompany {
	public static void main(String[] args) {

		Employee[] workers = new Employee[3];

		workers[0] = new Employee("철수"); // 철수 직원
		workers[0].setSalaryRole(new MereClerkSalaryRole(100)); // 평사원 철수

		workers[1] = new Employee("영희"); // 영희 직원
		workers[1].setSalaryRole(new MereClerkSalaryRole(100)); // 평사원 영희

		workers[2] = new Employee("홍길동"); // 홍길동 직원
		workers[2].setSalaryRole(new ManagerSalaryRole(200)); // 관리자 홍길동

		// 현재 월급
		System.out.println("현재 월급입니다.");
		for (int i = 0; i < workers.length; i++)
			// 현재 월급을 프린트한다.
			System.out.println(workers[i].getName() + "의 현재 월급은 " + workers[i].getSalary() + " 만원 입니다.");

		System.out.println("");

		// 현재 월급에서 10%를 올린 월급
		System.out.println("올린 후의 월급입니다.");
		for (int i = 0; i < workers.length; i++) {
			workers[i].manageSalary(10); // 월급을 10% 올린다.
			// 현재 월급을 프린트한다.
```

```
                    System.out.println(workers[i].getName() + "의 현재 월급은 "
+ workers[i].getSalary() + " 만원 입니다.");

                }

                System.out.println("");
                System.out.println("철수가 관리자로 승진하였습니다. 승진 후의 월급은
다음과 같습니다.");

                // 철수가 관리자가 됨.
                workers[0].setSalaryRole(new ManagerSalaryRole(200));
                workers[0].manageSalary(10);
                System.out.println(workers[0].getName() + "의 현재 월급은 " +
workers[0].getSalary() + " 만원 입니다.");

        }
}
```

처음에 작성한 프로그램 보다는 클래스의 구조가 복잡해졌지만 훨씬 변화에 유연한 구조로 변경됐음을 확인할 수 있다.