

改造面向过程式设计

张逸

www.agiledon.com

使用面向对象语言进行过程式设计的例子，可谓俯拾皆是。看这段代码：

```
public class SyncExecutor {
    public void executeSync() {
        syncSchools();
        syncGrades();
        syncFaculties();
    }
}
```

这段代码很清晰，分别执行了对学校、年级与教师信息的同步。一目了然，似乎没有什么问题。然而，如果深入阅读各个同步子方法，就会发现某种坏味道，那就是重复代码。

```
private void syncSchools() {
    List<School> sourceSchools = getSourceSchools();
    List<School> targetSchools = new ArrayList<School>();
    List<String> sourceSchoolCodes = getSchoolCodes(sourceSchools);
    Map<String,School> targetSchoolWithCodeMapping =
        schoolService.getSchoolWithCodeMapping(sourceSchoolCodes);

    for (School sourceSchool:sourceSchools) {
        String schoolCode = sourceSchool.getSchoolCode();
        School targetSchool =
            targetSchoolWithCodeMapping.get(schoolcode);
        if (targetSchool == null) {
            targetSchool = new School(
                sourceSchool.getSchoolCode(),
                sourceSchool.getSchoolName(),
                sourceSchool.getProvinceCode(),
                sourceSchool.getSchoolAddress(),
                sourceSchool.getSchoolZip(),
                sourceSchool.getSchoolTel());
        }
    }
}
```

```
    } else if (isCover) {
        targetSchool.setSchoolCode(
            sourceSchool.getSchoolCode());
        targetSchool.setSchoolName(
            sourceSchool.getSchoolName());
        targetSchool.setProvinceCode(
            sourceSchool.getProvinceCode());
        targetSchool.setSchoolAddress(
            sourceSchool.getSchoolAddress());
        targetSchool.setSchoolZip(
            sourceSchool.getSchoolZip());
        targetSchool.setSchoolTel(
            sourceSchool.getSchoolTel());
    }
    targetSchools.add(targetSchool);
}

syncService.saveOrUpdate(targetSchools);
}

private void syncGrades() {
    List<Grade> sourceGrades = getSourceGrades();
    List<Grade> targetGrades = new ArrayList<Grade>();
    List<String> sourceGradeCodes = getGradeCodes(sourceGrades);
    Map<String,Grade> targetGradeWithCodeMapping =
        gradeService.getGradeWithCodeMapping(sourceGradeCodes);

    for (Grade sourceGrade:sourceGrades) {
        String gradeCode = sourceGrade.getGradeCode();
        Grade targetGrade =
            targetGradeWithCodeMapping.get(gradeCode);

        if (targetGrade == null) {
            targetGrade = new Grade(
                sourceGrade.getGradeCode(),
                sourceGrade.getName(),
```

```
        sourceGrade.getEntranceDay(),
        sourceGrade.getGraduateDay(),
        sourceGrade.getSchoolCode(),
        sourceGrade.getSchoolProperty());
    } else if (isCover) {
        targetGrade.setGradeCode(sourceGrade.getGradeCode());
        targetGrade.setName(sourceGrade.getName());
        targetGrade.setEntranceDay(
            sourceGrade.getEntranceDay());
        targetGrade.setGraduateDay(
            sourceGrade.getGraduateDay());
        targetGrade.setSchoolCode(sourceGrade.getSchoolCode());
        targetGrade.setSchoolProperty
            (sourceGrade.getSchoolProperty());
    }
    targetGrades.add(targetGrade);
}

syncService.saveOrUpdate(targetGrades);
}
```

当然，真实的代码更加复杂与混乱，但如果经过一系列重构，例如 `Rename Method`，`Extract Method` 之后，就会变得逐渐清晰，大体结构如上述展示的代码。阅读这样的代码，是否发现各个同步子方法均有似曾相识的感觉呢？究其原因，在于同步的执行逻辑大体相似，换言之，它们具有相似的模板。我们需要改善其结构，实现代码的重用。然而，在方法层面上，我们已很难实现这一点。事实上，当我们在编写同步方法时，已经落入了过程式设计的窠臼。我们首先想到的是执行的过程，而非对象。现在，我们需要将这些执行过程封装为对象，充分地利用继承等机制实现类级别的重用。显然，这里可以运用 `Form Template Method` 重构。当然，在此之前，我们还需要运用 `Extract Superclass`，对 `School`、`Grade` 等类进行一系列重构，例如为它们建立共同的父类 `Entity`，提供 `getCode()` 方法。并运用 `Rename Method`，将原来各自实体类的相关方法，例如 `getSchoolCode()`、`getGradeCode()` 等，更名为 `getCode()`。

现在，我们需要为同步操作定义一个共同的抽象类 `DataSynchronizer`，然后利用 `Move Method` 重构，将原有 `SyncExecutor` 的相关代码搬移到 `DataSynchronizer` 中：

```
public abstract class DataSynchronizer {
    public void execute() {
        List<Entity> sourceEntities = getSourceEntities();
        List<Entity> targetEntities = new ArrayList<Entity>();
        List<String> sourceEntityCodes =
            getEntityCodes(sourceEntities);
        Map<String,Entity> targetEntityWithCodeMapping =
            getEntityWithCodeMapping(sourceEntityCodes);

        for (Entity sourceEntity:sourceEntities) {
            String entityCode = sourceEntity.getCode();
            Entity targetEntity =
                targetEntityWithCodeMapping.get(entityCode);
            if (targetEntity == null) {
                targetGrade = createEntity(sourceEntity);
            } else if (isCover) {
                updateEntity(targetEntity,sourceEntity);
            }
            targetEntities.add(targetEntity);
        }
        syncService.saveOrUpdate(targetEntities);
    }

    protected abstract List<Entity> getSourceEntities();
    protected abstract List<String> getEntityCodes(
        List<Entity> entities);
    protected abstract Map<String,Entity>
        getEntityWithCodeMapping(List<String> entityCodes);
    protected abstract Entity createEntity(Entity sourceEntity);
    protected abstract void updateEntity(
        Entity target, Entity source);
}
```

注意，在获得 Entity 与 Code 的 Map 对象时，我对原有的代码实现进行了封装，因为不同的实体同步类，所要调用的 Service 对象是不一样的。因此，需要将调用 Service 相

关方法的实现留给子类。现在，只需要定义各个同步类继承 `DataSynchronizer`，重写相关的受保护抽象方法即可：

```
public class SchoolSynchronizer extends DataSynchronizer {}
public class GradeSynchronizer extends DataSynchronizer {}
public class FacultySynchronizer extends DataSynchronizer {}
```

接着，修改 `SyncExecutor` 类的实现。为方便调用同步子类的相关方法，我定义了一个 Factory Method：

```
public class SyncExecutor {
    public void executeSync() {
        for (DataSynchronizer dataSync:createSynchronizers()) {
            dataSync.execute();
        }
    }

    protected List<DataSynchronizer> createSynchronizers() {
        List< DataSynchronizer> synchronizers =
            new ArrayList< DataSynchronizer>();
        synchronizers.add(new SchoolSynchronizer());
        synchronizers.add(new GradeSynchronizer());
        synchronizers.add(new FacultySynchronizer());

        return synchronizers;
    }
}
```

以真正面向对象的方式来完成上述功能，无论在代码结构、重用性还是扩展性方面，比诸之前的实现，都有了长足的改善。这就是面向对象设计的优雅之处。

纵观整个重构过程，实际上，我在运用 `Convert Procedural Design to Objects` 重构时，大量运用了 `Rename Method`、`Extract Method`、`Move Method`、`Extract Superclass`、`Form Template Method` 等重构手法。这是合乎常情的。当我们在对程序进行重构时，往往需要运用各种重构手法，才能达到最终的重构目的。对于大型重构而言，这种特征尤其明显。