迭代一设计文档

软件设计的核心目的：软件的可维护和可复用。

设计原则：

- 单一职责原则
- 开闭原则
- 里氏替换原则
- 依赖倒转原则
- 接口隔离原则
- 合成复用原则
- 迪米特法则

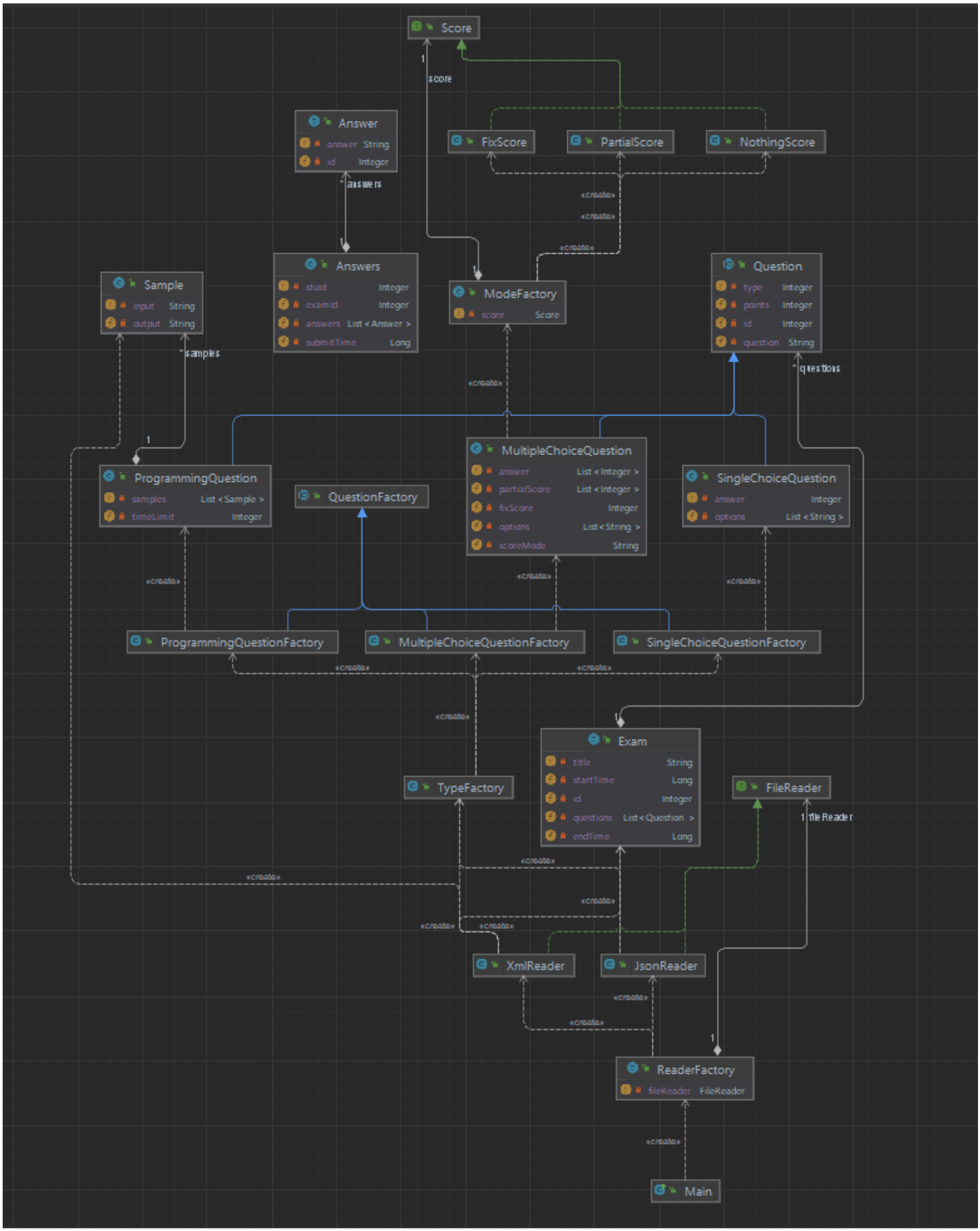
现有已学的设计模式：

- 策略模式
- 简单工厂
- 工厂方法
- 抽象工厂
- 建造者模式
- 原型模式

我目前迭代一的设计就基于此进行。

首先明晰一下迭代一的功能需求：题目读取和评分，我就依据流程来进行设计了。

整体设计类图如下：


Score
score
Answer
answer String
id Integer
answers
FixScore
PartialScore
NothingScore
«create»
Answers
stuId Integer
examId Integer
answers List < Answer >
submitTime Lang
ModeFactory
score Score
Question
type Integer
points Integer
id Integer
question String
questions
Sample
input String
output String
samples
ProgrammingQuestion
samples List < Sample >
timeLimit Integer
QuestionFactory
MultipleChoiceQuestion
answer List < Integer >
partialScore List < Integer >
fixScore Integer
options List < String >
scoreMode String
SingleChoiceQuestion
answer Integer
options List < String >
ProgrammingQuestionFactory
MultipleChoiceQuestionFactory
SingleChoiceQuestionFactory
Exam
title String
startTime Lang
id Integer
questions List < Question >
endTime Lang
TypeFactory
FileReader
fileReader
XmlReader
JsonReader
ReaderFactory
fileReader FileReader
Main

Part1

文件读取

首先是试题文件的读取。

第一个变化的量为需要支持XML与JSON文件读取一次测试的所有题目信息。我在这里也没有做过多考虑，不同的读取方式，那就简单工厂+策略模式，其中简单工厂的参数为文件的类型（后缀名），由此得到了下面的一段代码：

```
public class ReaderFactory {
    // 读取方法策略类
    private FileReader fileReader;

    public ReaderFactory(String type) {
        // 简单工厂 + 策略模式选择构建Reader策略
        if (type.equals("json")) {
            this.fileReader = new JsonReader();
        } else if (type.equals("xml")) {
            this.fileReader = new XmlReader();
        }
    }

    public Exam getExam(String examPath) {
        return this.fileReader.getExam(examPath);
    }
}
```

由于目前工厂类负责创建的对象较少，简单工厂即可满足我的需求和日后可能的拓展需求。同时结合策略模式，依赖于抽象接口而非具体实现，很大程度上减少了代码的耦合。日后若要新增不同的文件，直接implements接口并在上述简单工厂增加 if 即可。

题目读取

紧接着来到了第二个会变化的点，三种类型的题目，该如何抽离出变化的量呢？我是想到了利用工厂方法去做到这一点。

```
public abstract class QuestionFactory {
    public abstract Question getQuestion();
}
```

之所以不直接采用简单工厂，一来是我进行题目分析读取的地方为不同的具体策略类中，如果使用简单工厂，增加一种题型，如果我存在5种不同的文件类型，则需要修改至少5处代码实现，耦合度过高。

这里我采用了一种简单工厂+工厂方法的设计方法。

```
int type = Integer.parseInt(questionElement.getChildText("type"));
Question question;
QuestionFactory questionFactory;
TypeFactory typeFactory = new TypeFactory();
// 简单工厂获取具体工厂类
questionFactory = typeFactory.getQuestionFactory(type);
```

根据题目的type属性，用一个简单工厂去实例化具体的工厂，代码如下：

```
public class TypeFactory {
    public QuestionFactory getQuestionFactory(int type) {
        switch (type) {
            case 1:
                return new SingleChoiceQuestionFactory();
            case 2:
                return new MultipleChoiceQuestionFactory();
            case 3:
                return new ProgrammingQuestionFactory();
            default:
                return null;
        }
    }
}
```

新增题目类型时，由于类型全部交由简单工厂，因此只需修改简单工厂新增具体工厂即可，各个原先策略类中的代码完全不需要修改，高度的可扩展。

一个问题在这里就困扰我了，根据工厂方法是由具体的QuestionFactory创建出具体的Question类型了，如何赋值呢？我们仅仅是new了一个对象出来啊，尽管这个对象目前已经是具体子类了。传参数给工厂去创建？可是这样我们不是还得判断现在具体是什么类型，由此我们需要读取什么参数？如果不这样，那么就需要将所有参数一股脑全部交给工厂，这行吗？

其实json这里是没有问题的，我可以直接通过 `questions.add(objectMapper.treeToValue(questionNode, question.getClass()));` ，让他帮我自动赋值转化为具体的对象。可是xml我似乎并没有发现如此简单的方法，又或者今后新增不同的文件类型，我们都这样做吗？

我是觉得不妥的，因此我采取了Map的方式，在 `Question` 抽象类中增加了 `init(Map<String, Object> map, Question question)` 方法。

在每一个具体实现类中，都像下面这样：

```
public class xxx extends Question {
    private List<String> options;
    private Integer answer;

    @Override
    public Question init(Map<String, Object> map, Question question) {
        SingleChoiceQuestion singleChoiceQuestion = (SingleChoiceQuestion) question;
        singleChoiceQuestion.setOptions((List<String>) map.get("options"));
        singleChoiceQuestion.setAnswer((Integer) map.get("answer"));
        return singleChoiceQuestion;
    }
}
```

调用了子类的 `init()` 方法则必然存在这些特有的Key，今后新增文件类型时只需要遵循这一点，传递map，则这些部分代码基本不需要修改，同时确保只拿到了需要的数据。

由此，我们就结束了读取的相关工作。

Part2

评分

这里我首先关注了题目类型的变化，在每一个 `Question` 类中新增方法 `testAnswer` 进行不同题型的对应评分。

以多选题为例：

```
public class MultipleChoiceQuestion extends Question {
    @Override
    public int testAnswer(Question question, Answer answer) {
        MultipleChoiceQuestion multipleChoiceQuestion = (MultipleChoiceQuestion) question;
        String scoreMode = multipleChoiceQuestion.getScoreMode();
        // 利用工厂调用不同策略
        ModeFactory modeFactory = new ModeFactory(scoreMode);
        return modeFactory.getScore(multipleChoiceQuestion, answer);
    }
}
```

此时注意到多选题的需要中对于给分策略也是变化的，将变化抽离出来。跟先前类似，采取简单工厂+策略模式，借助一个简单的工厂 `ModeFactory`

```
public class ModeFactory {
    private Score score;
    public ModeFactory(String mode) {
        if (Objects.equals(mode, "fix")) {
            this.score = new FixScore();
        }else if (Objects.equals(mode, "nothing")) {
            this.score = new NothingScore();
        } else if (Objects.equals(mode, "partial")) {
            this.score = new PartialScore();
        }
    }

    public Integer getScore(MultipleChoiceQuestion question, Answer answer) {
        return score.getScore(question, answer);
    }
}
```

由此工厂决定具体 `score` 策略，进而计算分数。