# 迭代三设计文档

## 功能需求

迭代三需要我们对迭代二的编程题评测进一步扩展，要求实现一种代码性能评测限制指标——时间复杂度和一种代码风格评测指标——圈复杂度。

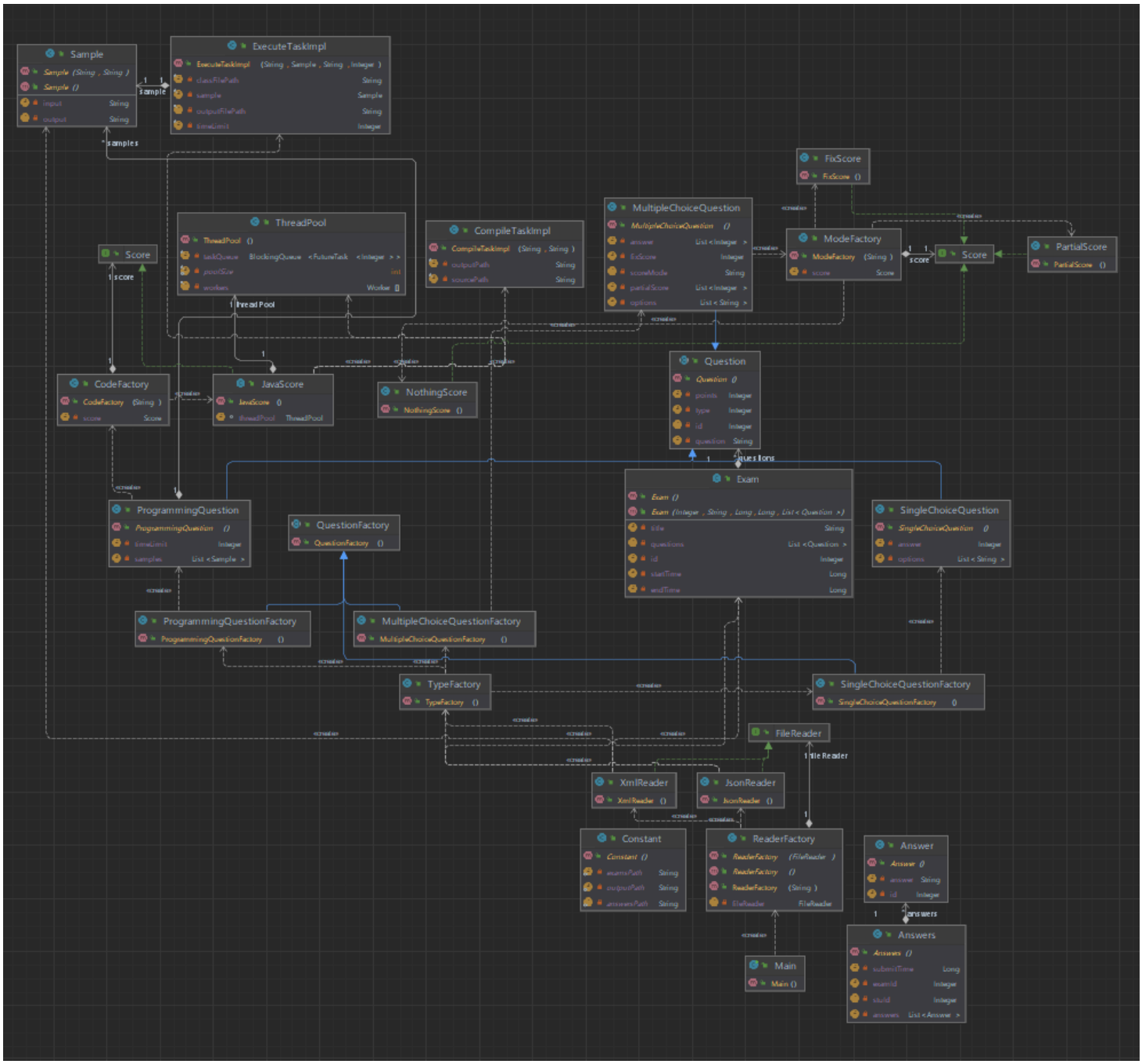
## 时间限制

在代码执行阶段对运行时间进行计算并对于超出时间限制的作答结果判0分。

## 圈复杂度

采用节点判定法，按照基本判定规则以相对统一的方式分别实现单个函数和整个类的圈复杂度实现。

# 设计类图




三次迭代后，整体的设计类图如上。

从这张图不难看出以下几点：

- 几乎所有的具体实现类的创建，我们都交由工厂去进行，符合工厂类的职责。
- 客户端调用尽量持有抽象类而非具体实现类
- 最小接口以及单一职责，每一个类尽量只有一个功能对外提供。

因此，该OJ系统是高内聚。低耦合的，我们可以说拓展新功能以及修改功能也会变得十分方便，且影响较小。

# 迭代三功能设计

## 时间限制

```
// 等待执行线程
boolean completedWithinTime = process.waitFor(timeLimit, java.util.concurrent.TimeUnit.MILLISEC(

if (!completedWithinTime) {
    // 超出时间限制，返回0分
    process.destroy(); // 终止进程
    // 输出超时信息
    System.out.println("TimeOut!");
    return 0;
}
```

超时则终止运行进程并使得线程返回。

## 圈复杂度

考虑到不同编程语言圈复杂度的计算是会有所区别的，我们复用先前的JavaScore，为Score接口加上calculateCyclomaticComplexity方法，在具体实现类中针对不同编程语言进行具体实现。将相关的变化内聚起来。

```java
public Integer calculateCyclomaticComplexity(String code) {
    CompilationUnit cu = StaticJavaParser.parse(code);
    int classComplexity = 0;

    for (MethodDeclaration method : cu.findAll(MethodDeclaration.class)) {
        int methodComplexity = calculateMethodComplexity(method);
        classComplexity += methodComplexity;
    }

    return classComplexity;
}

private int calculateMethodComplexity(MethodDeclaration method) {
    int complexity = 1;

    for (IfStmt ignored : method.findAll(IfStmt.class)) {
        complexity++; // 每个 if 语句增加 1
    }

    for (WhileStmt ignored : method.findAll(WhileStmt.class)) {
        complexity++; // 每个 while 循环增加 1
    }

    for (DoStmt ignored : method.findAll(DoStmt.class)) {
        complexity++; // 每个 do-while 循环增加 1
    }

    for (ForStmt ignored : method.findAll(ForStmt.class)) {
        complexity++; // 每个 for 循环增加 1
    }

    for (ConditionalExpr ignored : method.findAll(ConditionalExpr.class)) {
        complexity++;
    }

    for (BinaryExpr binaryExpr : method.findAll(BinaryExpr.class)) {
        BinaryExpr.Operator operator = binaryExpr.asBinaryExpr().getOperator();
        if (operator == BinaryExpr.Operator.AND || operator == BinaryExpr.Operator.OR) {
            complexity++; // 布尔运算符作为判定节点，增加 1
        }
    }
```

```
    return complexity;
}
```

由于直接采用节点判定法，我们直接采用上述一段简单的代码，查找所有符合的节点即可。

当然，简化一下如下写就行：

```
complexity += method.findAll(IfStmt.class).size() + method.findAll(WhileStmt.class).size() + met

for (BinaryExpr binaryExpr : method.findAll(BinaryExpr.class)) {
    BinaryExpr.Operator operator = binaryExpr.asBinaryExpr().getOperator();
    if (operator == BinaryExpr.Operator.AND || operator == BinaryExpr.Operator.OR) {
        complexity++; // 布尔运算符作为判定节点，增加 1
    }
}
```

~~为了以相对统一的方式实现单个函数和整个类的圈复杂度实现，我们规定他们的输入都为String，return都为计算得到的圈复杂度。~~

## 设计调整

在笔者依据设计文档设计完成后重新审视该设计时，发觉这里其实与课堂所讲树形结构是极其相似的，类中有函数，类中也可以有其他的类，当然这里测试极其简单，只有单一的类。

如此，应该可以使用组合模型对于上述的实现进行优化？加之需求要求相对一致地对待单个函数和整个类，不就更加符合组合模式的适用情形？

进行设计的修改

抽象元素：

```
public abstract class AbstractElement {
    public abstract Integer calculateCyclomaticComplexity();
    public abstract void add(AbstractElement element);
    public abstract void remove(AbstractElement element);
    public abstract AbstractElement getChild(int i);
}
```

类元素：

```
public class ClassElement extends AbstractElement{

    private ArrayList<AbstractElement> children = new ArrayList<>();

    @Override
    public Integer calculateCyclomaticComplexity() {
        int complexity = 0;
        for (AbstractElement obj : children) {
            complexity += obj.calculateCyclomaticComplexity();
        }
        return complexity;
    }

    @Override
    public void add(AbstractElement element) {
        children.add(element);
    }

    @Override
    public void remove(AbstractElement element) {
        children.remove(element);
    }

    @Override
    public AbstractElement getChild(int i) {
        return children.get(i);
    }
}
```

方法元素：

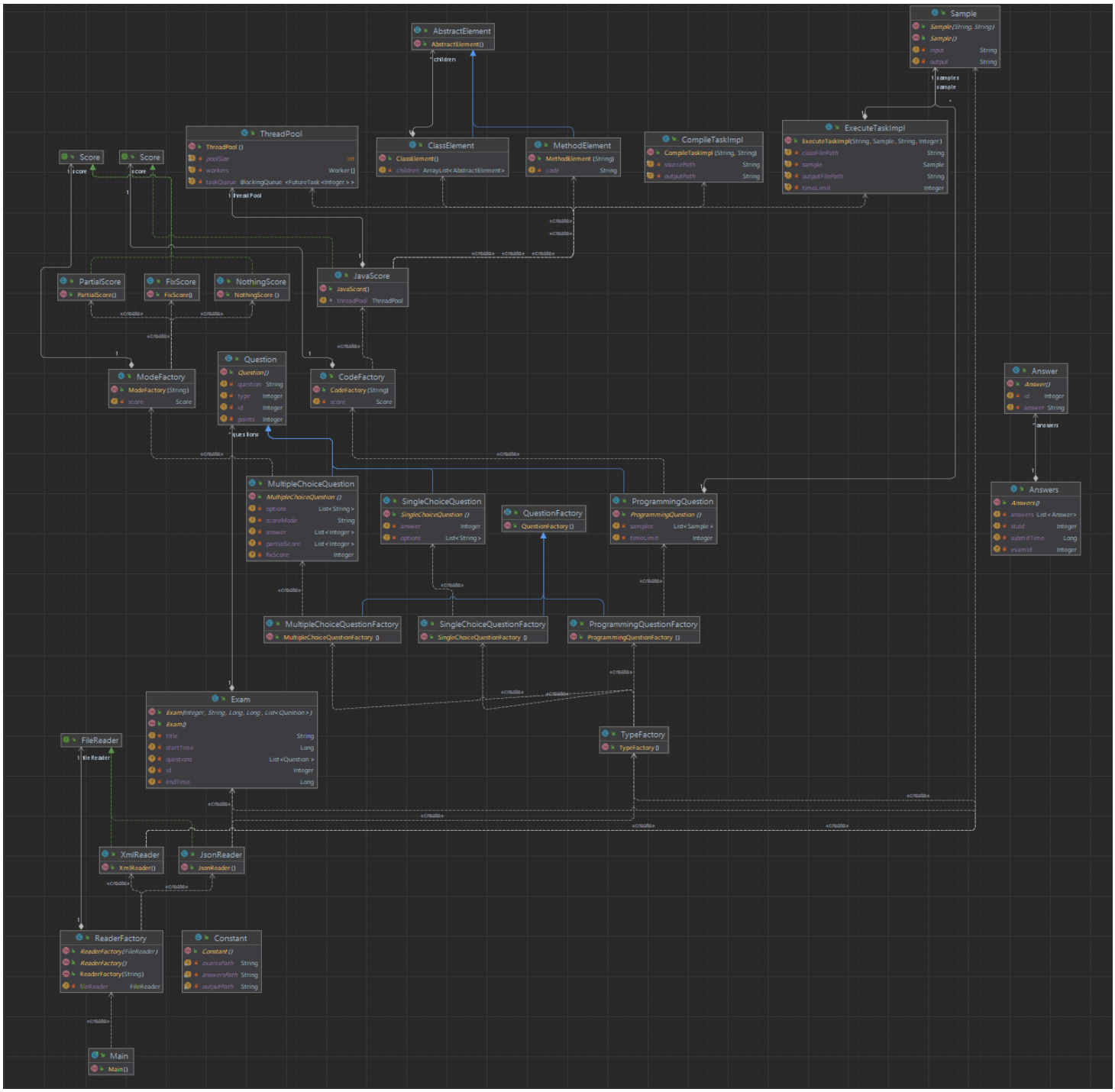
```
public class MethodElement extends AbstractElement{

    private String code;

    public MethodElement(String code) {
        this.code = code;
    }

    @Override
    public Integer calculateCyclomaticComplexity() {
        MethodDeclaration method = StaticJavaParser.parseMethodDeclaration(code);
        int complexity = 1;
        complexity += method.findAll(IfStmt.class).size() + method.findAll(WhileStmt.class).size
        for (BinaryExpr binaryExpr : method.findAll(BinaryExpr.class)) {
            BinaryExpr.Operator operator = binaryExpr.getOperator();
            if (operator == BinaryExpr.Operator.AND || operator == BinaryExpr.Operator.OR) {
                complexity++; // Boolean operators as decision points, add 1
            }
        }
        return complexity;
    }

    @Override
    public void add(AbstractElement element) {

    }

    @Override
    public void remove(AbstractElement element) {

    }

    @Override
    public AbstractElement getChild(int i) {
        return null;
    }
}
```

客户端调用：

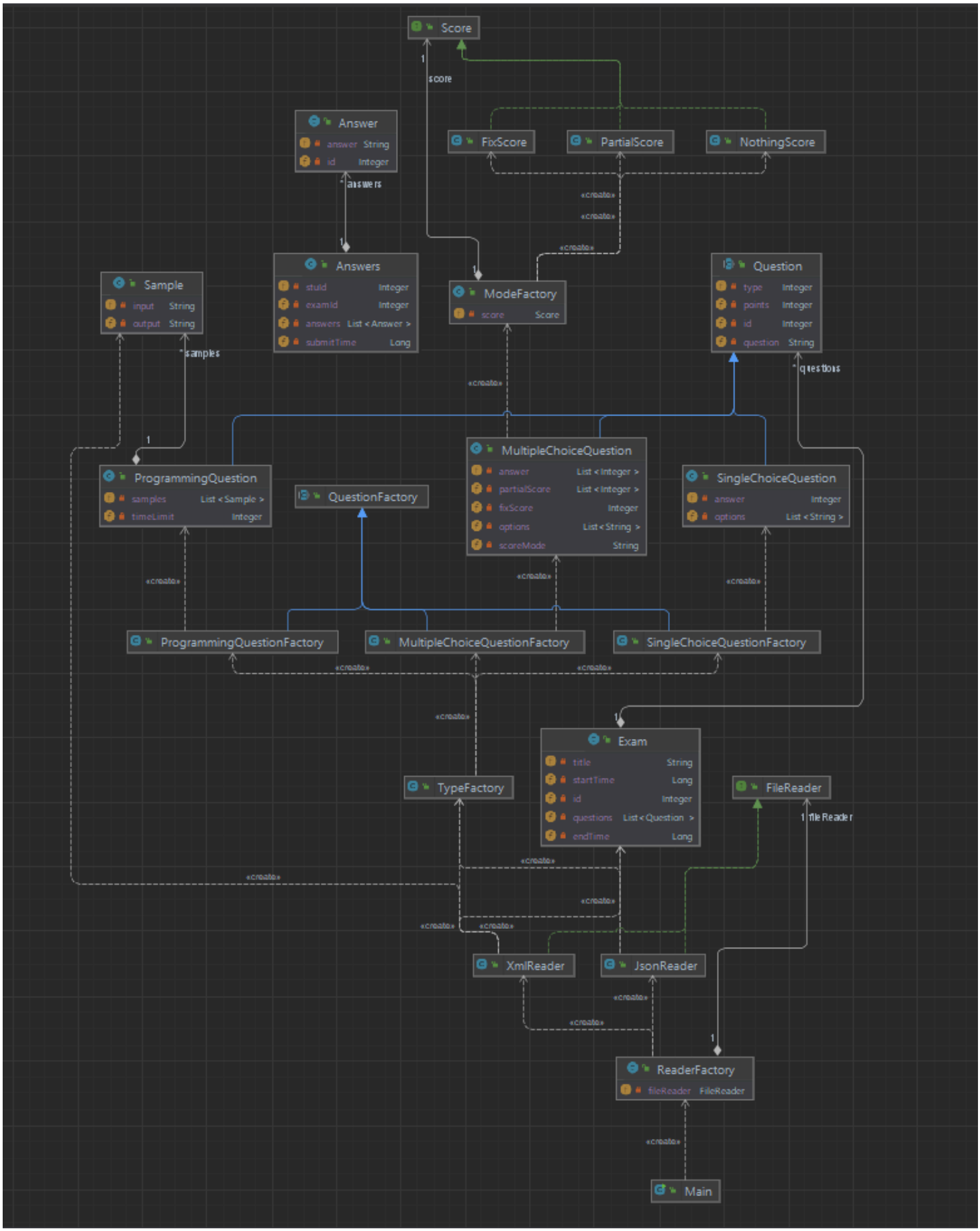
```
public Integer calculateCyclomaticComplexity(String code) {
    AbstractElement classElement = new ClassElement();
    CompilationUnit cu = StaticJavaParser.parse(code);
    for (MethodDeclaration method : cu.findAll(MethodDeclaration.class)) {
        MethodElement methodElement = new MethodElement(method.toString());
        classElement.add(methodElement);
    }
    return classElement.calculateCyclomaticComplexity();
}
```

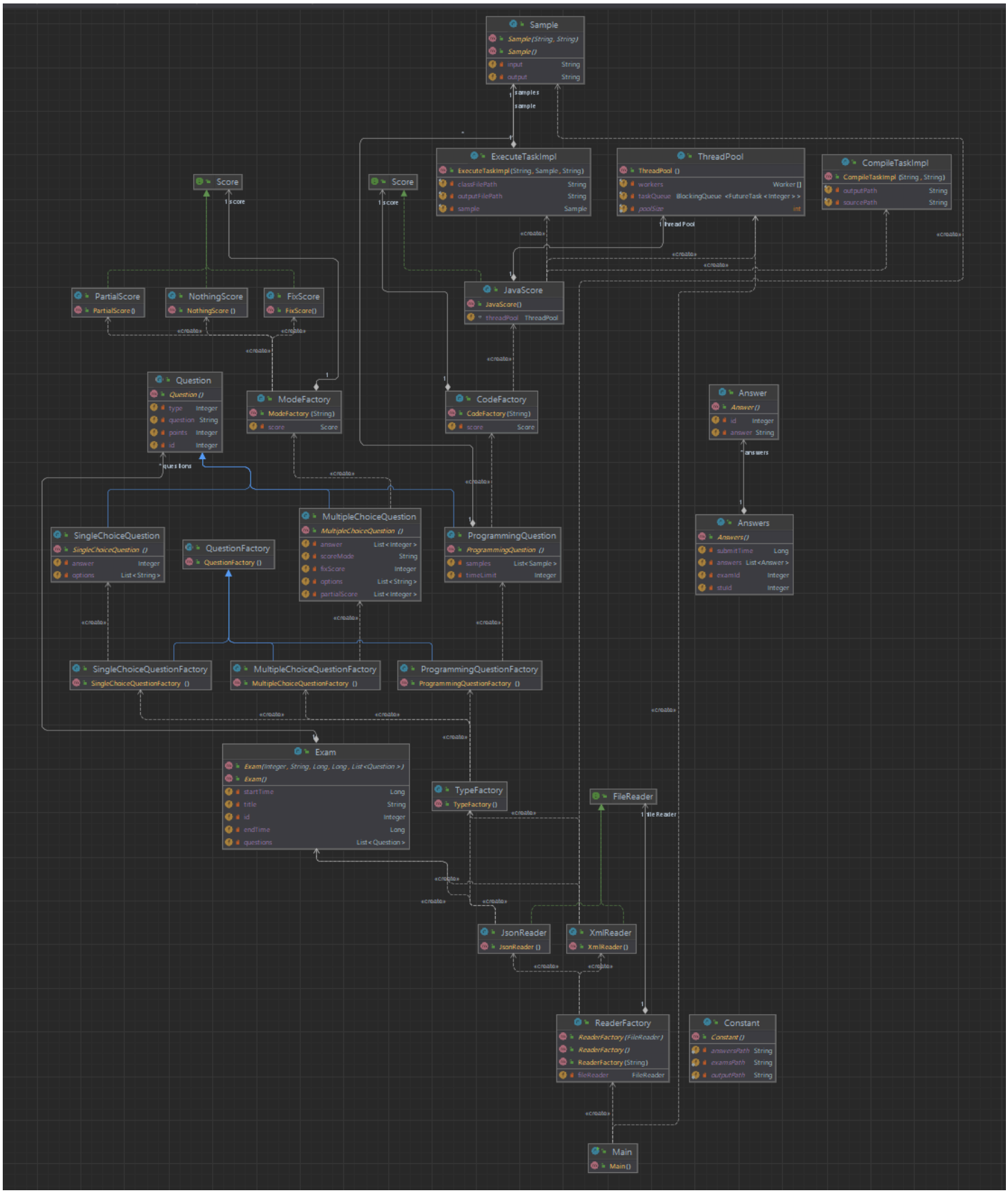
采用上面的设计，进行递归调用即可。当然这里只有一个类，不必如此。

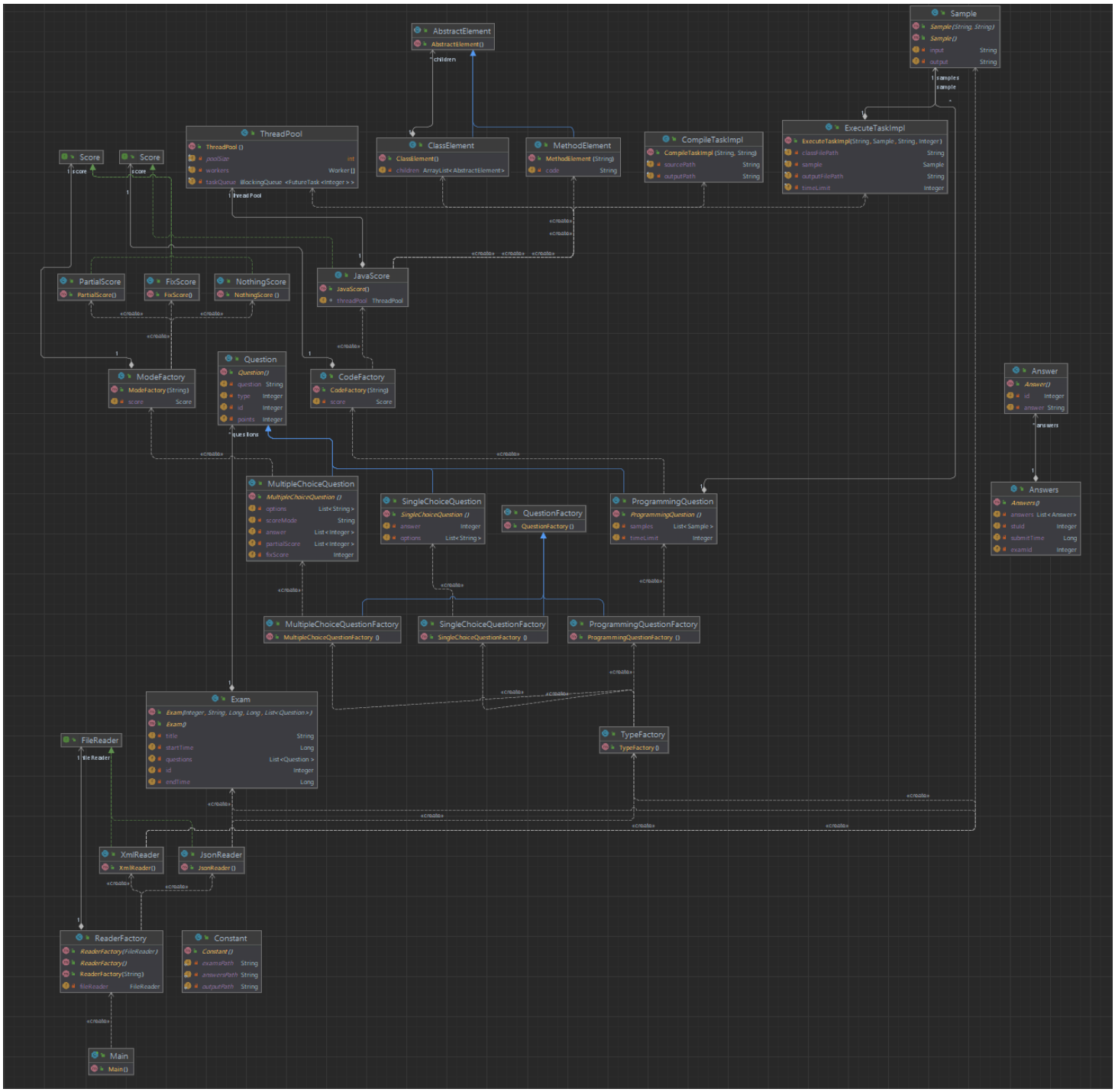
更改后的设计类图：


AbstractElement
AbstractElement()
children
Sample
Sample(String, String)
Sample()
input String
output String
samples
sample
ThreadPool
ThreadPool()
poolSize int
workers Worker[]
taskQueue BlockingQueue<FutureTask<Integer>>
ClassElement
ClassElement()
children ArrayList<AbstractElement>
MethodElement
MethodElement(String)
code String
CompileTaskImpl
CompileTaskImpl(String, String)
sourcePath String
outputPath String
ExecuteTaskImpl
ExecuteTaskImpl(String, Sample, String, Integer)
classFilePath String
sample Sample
outputFilePath String
timeLimit Integer
Score
Score
score
threadPool
«create»
PartialScore
PartialScore()
FixScore
FixScore()
NothingScore
NothingScore()
JavaScore
JavaScore()
threadPool ThreadPool
ModeFactory
ModeFactory(String)
score Score
Question
Question()
question String
type Integer
id Integer
points Integer
questions
CodeFactory
CodeFactory(String)
score Score
Answer
Answer()
id Integer
answer String
answers
Answers
Answers()
answers List<Answer>
stuId Integer
submitTime Long
examId Integer
MultipleChoiceQuestion
MultipleChoiceQuestion()
options List<String>
scoreMode String
answer List<Integer>
partialScore List<Integer>
fixScore Integer
SingleChoiceQuestion
SingleChoiceQuestion()
answer Integer
options List<String>
QuestionFactory
QuestionFactory()
ProgrammingQuestion
ProgrammingQuestion()
samples List<Sample>
timeLimit Integer
MultipleChoiceQuestionFactory
MultipleChoiceQuestionFactory()
SingleChoiceQuestionFactory
SingleChoiceQuestionFactory()
ProgrammingQuestionFactory
ProgrammingQuestionFactory()
Exam
Exam(Integer, String, Long, Long, List<Question>)
Exam()
title String
startTime Long
questions List<Question>
id Integer
endTime Long
FileReader
fileReader
TypeFactory
TypeFactory()
XmlReader
XmlReader()
JsonReader
JsonReader()
ReaderFactory
ReaderFactory(FileReader)
ReaderFactory()
ReaderFactory(String)
fileReader FileReader
Constant
Constant()
examsPath String
answersPath String
outputPath String
Main
Main()

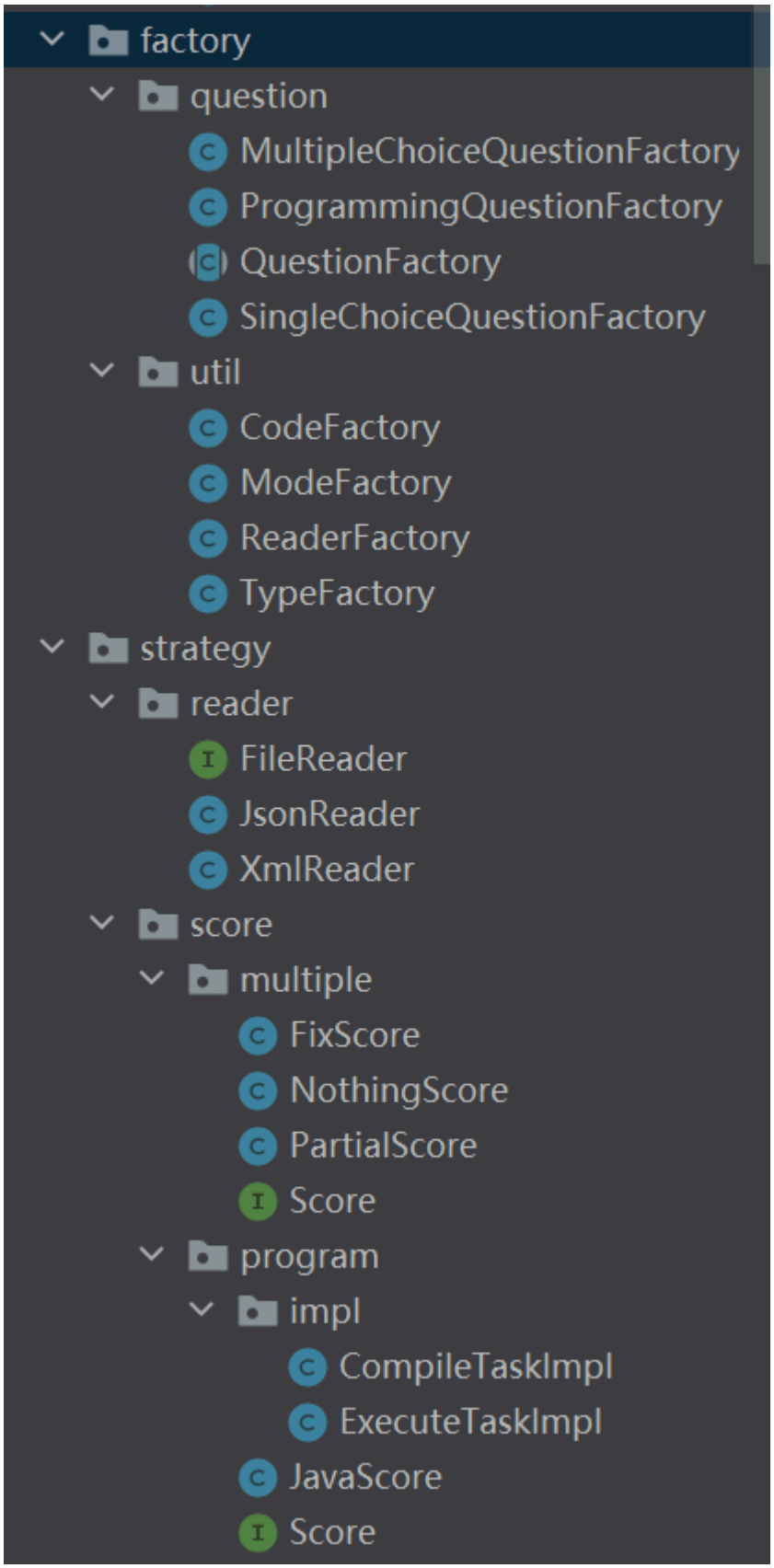
# 三次迭代的变化


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
Sample
input String
output String
samples
Question
type Integer
points Integer
id Integer
question String
questions
MultipleChoiceQuestion
answer List < Integer >
partialScore List < Integer >
fixScore Integer
options List < String >
scoreMode String
ProgrammingQuestion
samples List < Sample >
timeLimit Integer
QuestionFactory
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


Sample
Sample(String, String)
Sample()
input String
output String
samples
sample
ExecuteTaskImpl
ExecuteTaskImpl(String, Sample, String)
classFilePath String
outputFilePath String
sample Sample
ThreadPool
ThreadPool()
workers Worker[]
taskQueue BlockingQueue<FutureTask<Integer>>
poolSize int
threadPool
CompileTaskImpl
CompileTaskImpl(String, String)
outputPath String
sourcePath String
Score
score
Score
score
«create»
PartialScore
PartialScore()
NothingScore
NothingScore()
FixScore
FixScore()
JavaScore
JavaScore()
threadPool ThreadPool
Question
Question()
type Integer
question String
points Integer
id Integer
questions
ModeFactory
ModeFactory(String)
score Score
CodeFactory
CodeFactory(String)
score Score
Answer
Answer()
id Integer
answer String
answers
Answers
Answers()
submitTime Long
answers List<Answer>
examId Integer
stuId Integer
SingleChoiceQuestion
SingleChoiceQuestion()
answer Integer
options List<String>
QuestionFactory
QuestionFactory()
MultipleChoiceQuestion
MultipleChoiceQuestion()
answer List<Integer>
scoreMode String
fixScore Integer
options List<String>
partialScore List<Integer>
ProgrammingQuestion
ProgrammingQuestion()
samples List<Sample>
timeLimit Integer
SingleChoiceQuestionFactory
SingleChoiceQuestionFactory()
MultipleChoiceQuestionFactory
MultipleChoiceQuestionFactory()
ProgrammingQuestionFactory
ProgrammingQuestionFactory()
Exam
Exam(Integer, String, Long, Long, List<Question>)
Exam()
startTime Long
title String
id Integer
endTime Long
questions List<Question>
TypeFactory
TypeFactory()
FileReader
fileReader
JsonReader
JsonReader()
XmlReader
XmlReader()
ReaderFactory
ReaderFactory(FileReader)
ReaderFactory()
ReaderFactory(String)
fileReader FileReader
Constant
Constant()
answersPath String
examsPath String
outputPath String
Main
Main()




可以看出，随着不断进行迭代，系统的复杂度以及组件不断增加，采用好的设计原则可以使得我们不改变原有函数以及类的情况下，进行系统的扩展和维护。




工厂类和策略类的分别存放，良好的项目组织结构。

# 关于前两次迭代的反思

前一次需要实现线程池用以加速，当时的想法就是将一道题目的所有输出全部重定向到一个文件之中去。当时就被一个问题困扰许久，也就是多线程重定向先后的问题。

由于上次的测试用例较为宽容，这个问题虽然纠结许久，但是最后采用了一个十分不合理的方式通过了测试，即遍历重定向后的文件内容，只要有与答案文件相同的输出且数量等于测试用例数量即判定为通过测试。

但这次测试用例中出现一道题目测试样例中有相同的答案，此方法便行不通了。

最后思考出来的解决方案：将每个测试用例输出重定向到不同文件，如此也不需要考虑写入先后的问题，读取相对应文件内容即可。

反思：很多在某些阶段看起来能解决问题（就比如这里通过测试）的设计，在很多时候，往往是灾难性的，会给未来的维护以及出现的未曾想过的情况造成许许多多的困扰。