设计模式导入

参考资料

"Design Patterns, elements of reusable object- oriented software" By E. Gamma, R. Helm, R. Johnson, and J. Vlissides

我们将选取几个设计模式，目标是掌握OO中抽象、继承、组合和多态，并选取恰当的模式在lab和pj中运用。

- Visitor
- Strategy
- Composite
- Command
- Observer

关于示例代码

示例代码使用的是`typescript`。

这个语言的OO部分与 C++ / Java具有比较相似的语法。

示例：访问复杂的结构

对于下面的一段源程序：

```
class SimpleClass{
    testMethod(param: number) : number {
        if(param < 0)return -1
        else return 1
    }
}
```

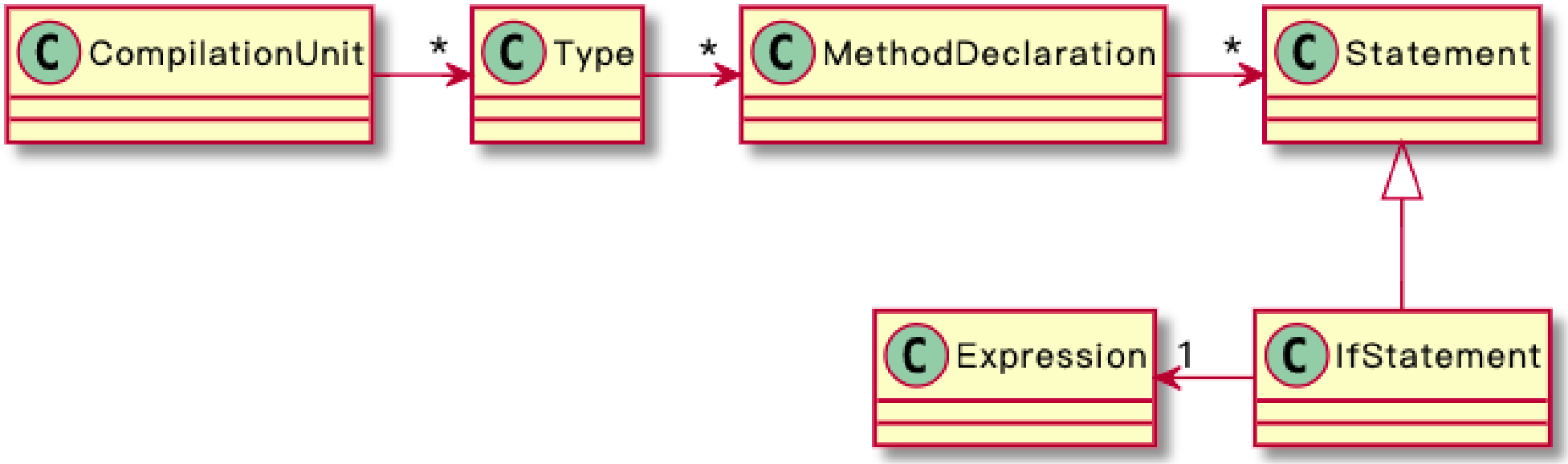
我们将上述**源程序的文本**作为**输入**，如何写一段程序找出以下的信息：

1. 找出类中所有方法的定义
2. 找出类中所有方法中的if语句及其条件表达式

访问复杂的结构

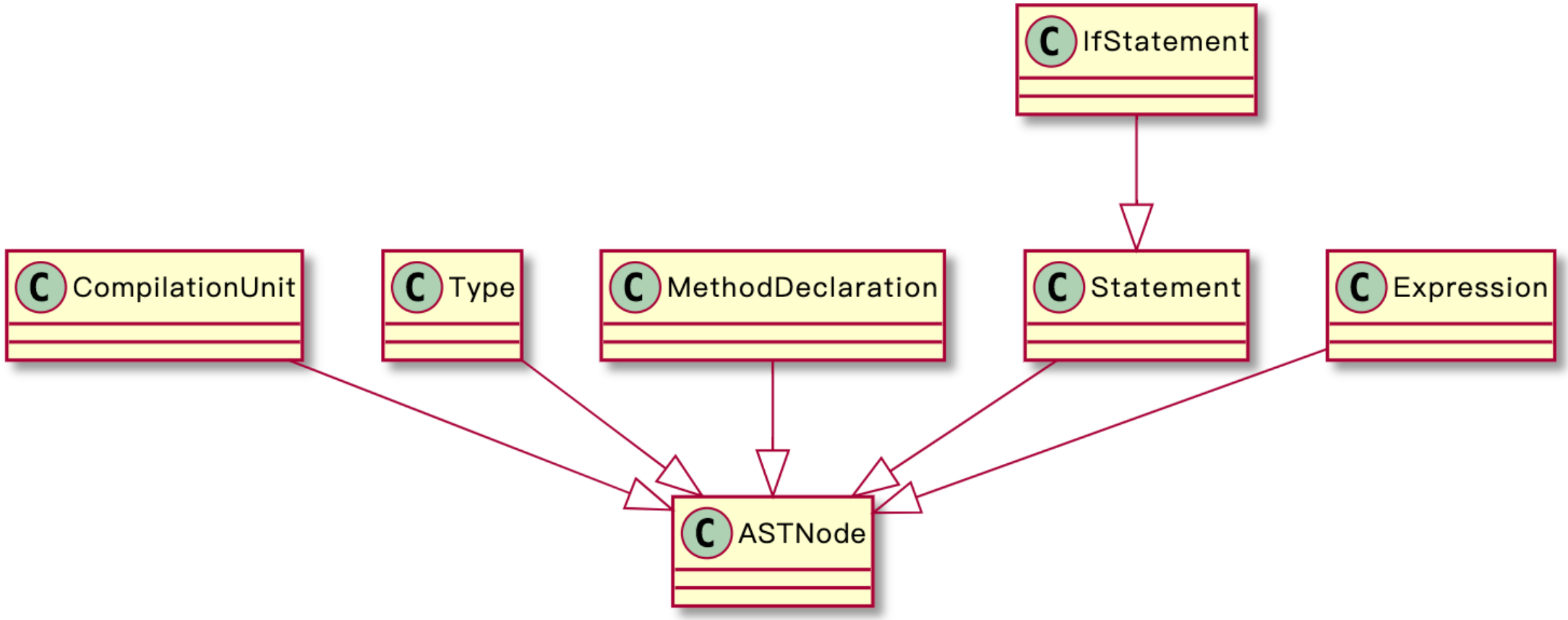
```
//simpleclass.ts
class SimpleClass{
    testMethod(param: number) : number {
        if(param < 0) return -1
        else return 1
    }
}
```

上面的源文件中对象的组合关系可以用下面的对象模型来表示：

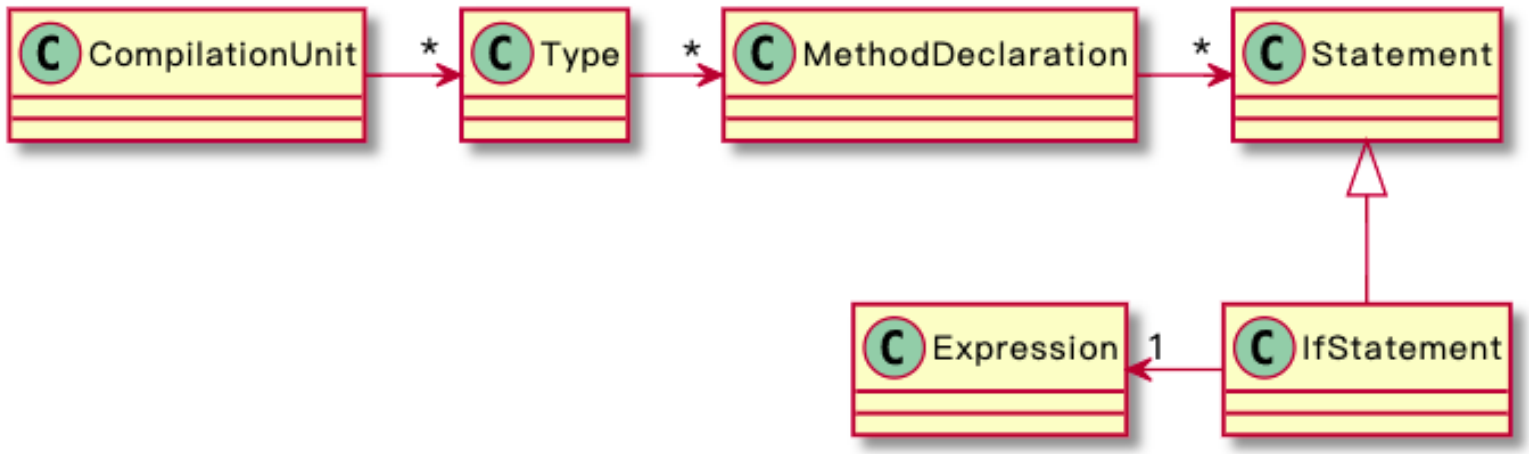


访问复杂的结构

上面的源文件中对象的继承关系可以用下面的对象模型来表示：



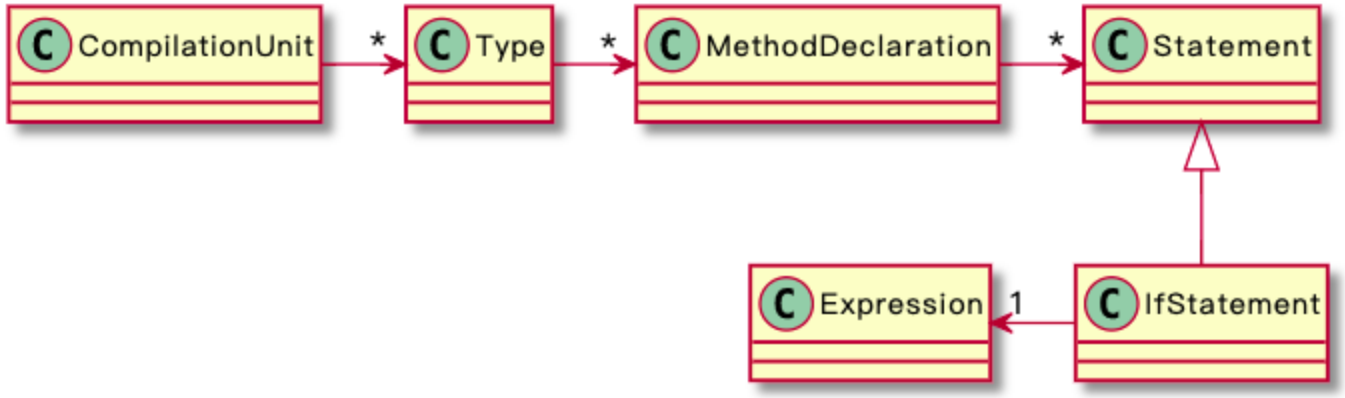
访问复杂的结构



找出类中所有方法的定义：

```
const CompilationUnit cu = ...//从文本文件构建对象结构
for(const t of cu.types)
    for(const md of t.methodDeclarations)
        console.log(......)
```

访问复杂的结构



找出类中所有方法中的if语句及其条件表达式:

```
const CompilationUnit cn = ...//从文本文件构建对象结构
for(const t of cu.types)
    for(const md of t.methodDeclarations)
        for(const stmt of md.statements)
            if(statement instanceof IfStatement)
                console.log(......)
```

这种具有比较深的嵌套结构的代码往往意味着程序结构可以改进。

访问复杂的结构

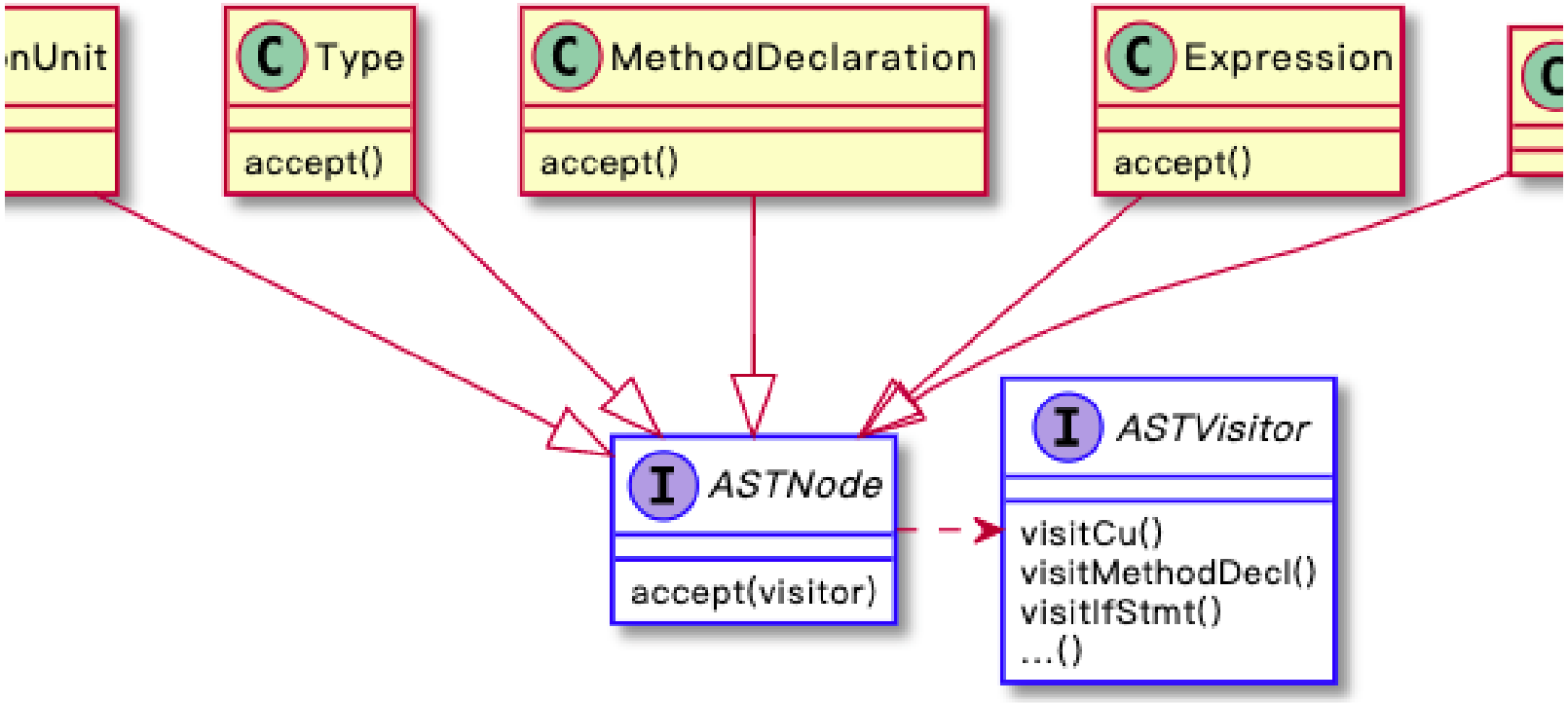
上面方法存在的问题：

- 每一个访问者都需要知道被访问的对象的复杂结构。
- 如果对象结构发生了改变，很多代码都需要修改。

如何解决：

- 考虑现实世界中类似问题的解决方法：引入导游的概念

引入Visitor接口



定义Visitor接口

C++

```cpp
class ASTVisitor{
public:
    virtual void visit(const CompilationUnit& node) const = 0;
    virtual void visit(const IfStatement& node) const = 0;
    virtual void visit(const MethodDeclaration& node) const = 0;
    // ...
};
```

Typescript

```typescript
interface ASTVisitor{
    visitCu(node: CompilationUnit): void
    visitIfStmt(node: IfStatement): void
    visitMethodDecl(node: MethodDeclaration): void
    //...
}
```

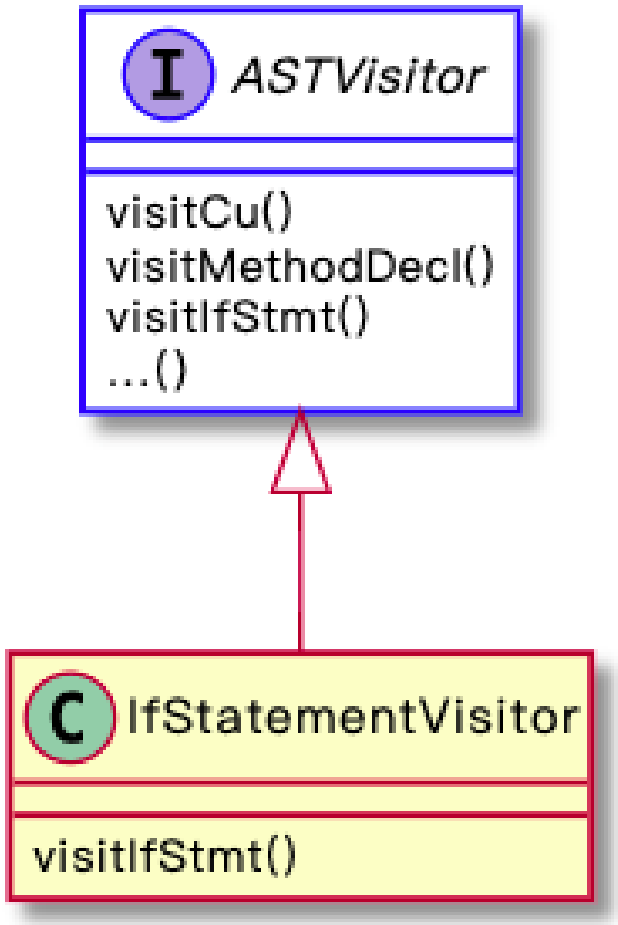
使用Visitor示例（一）

使用Visitor模式打印出所有的if语句及其其条件语句表达式

```
class IfStatementVisitor extends ASTVisitor{
    visitIfStmt(node: IfStatement){
        console.log( .....)
    }
}

const CompilationUnit ast = //....

//下面代码会输出if语句及其条件语句表达式
ast.accept(new IfStatementVisitor())
```



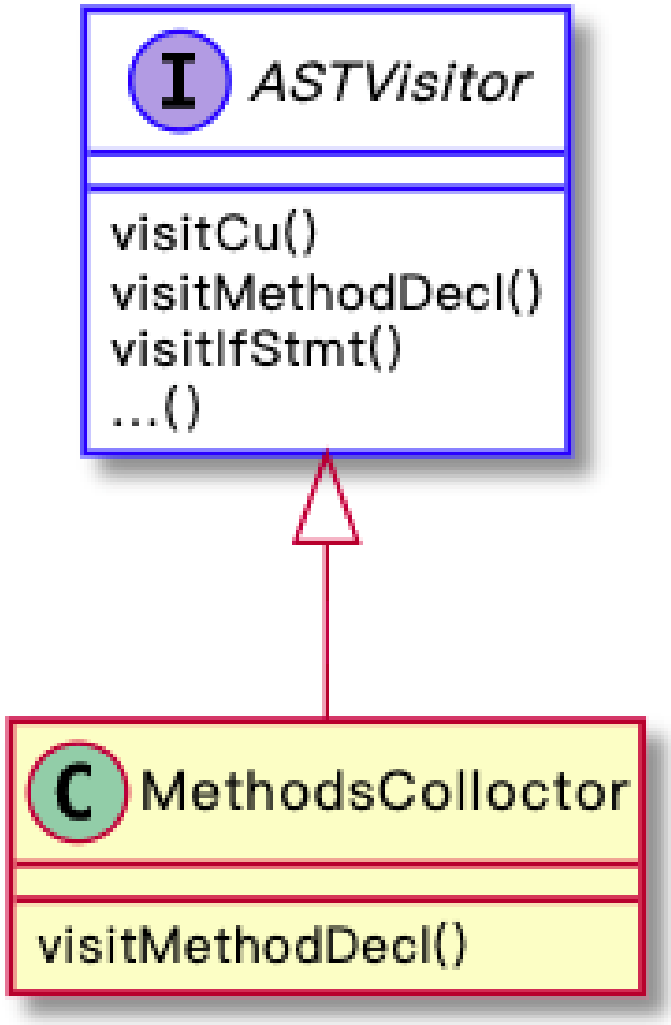
使用Visitor示例（二）

使用Visitor模式打印出类中所有的方法的定义

```
class MethodsCollector extends ASTVisitor{
    visitMethodDecl(node: MethodDeclaration){
        console.log( .....)
    }
}

const CompilationUnit ast = //....

//下面代码会打印出类中所有的方法的定义
ast.accept(new MethodsCollector())
```



比较

- 普通方法

```
const CompilationUnit cn = ...//从文本文件构建对象结构
for(const t of cu.types)
    for(const md of t.methodDeclarations)
        for(const stmt of md.statements)
            if(statement instanceof IfStatement){
                console.log(......)
            }
```

- Visitor模式

```
const CompilationUnit cn = ...//从文本文件构建对象结构
class IfStatementVisitor extends ASTVisitor{
    visitIfStatement(node: IfStatement){
        console.log( .....)
    }
}
ast.accept(new IfStatementVisitor())
```

设计模式导入：Visitor

只有源程序的语法树才需要这种方案吗?

只要遇到下面的问题

需要遍历复杂的对象结构，并对该结构中不同类型的对象作不同的操作。

就可以采用下面的解决方案

将遍历的过程和操作对象的过程分离，设计为不同的类。

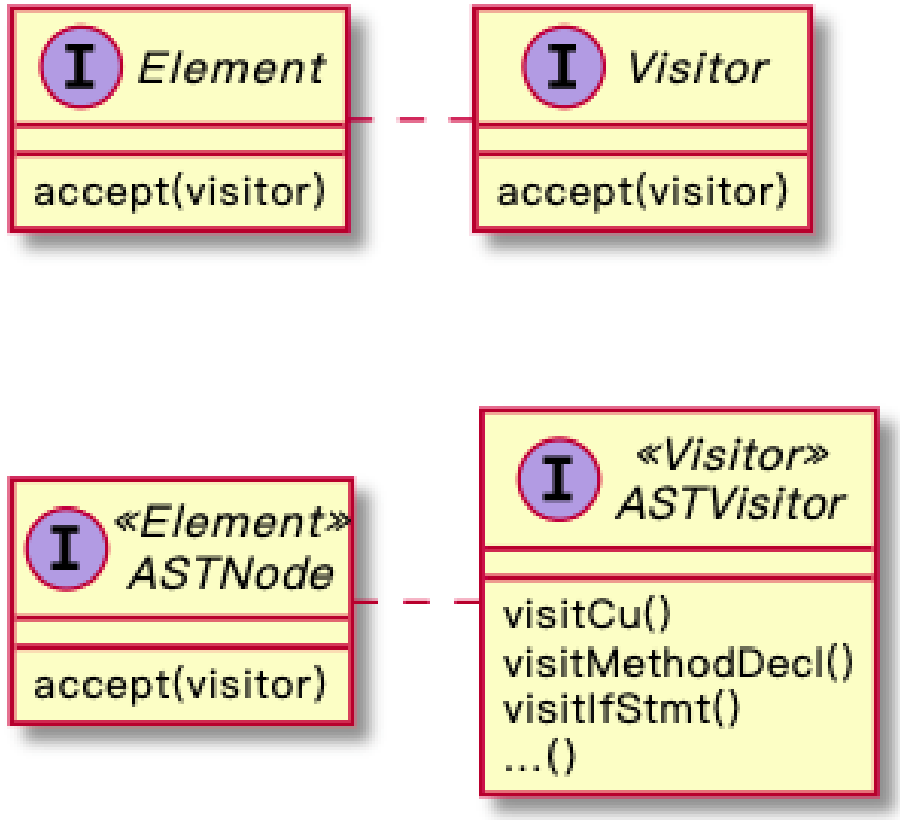
这就是Visitor模式

模式中的参与者

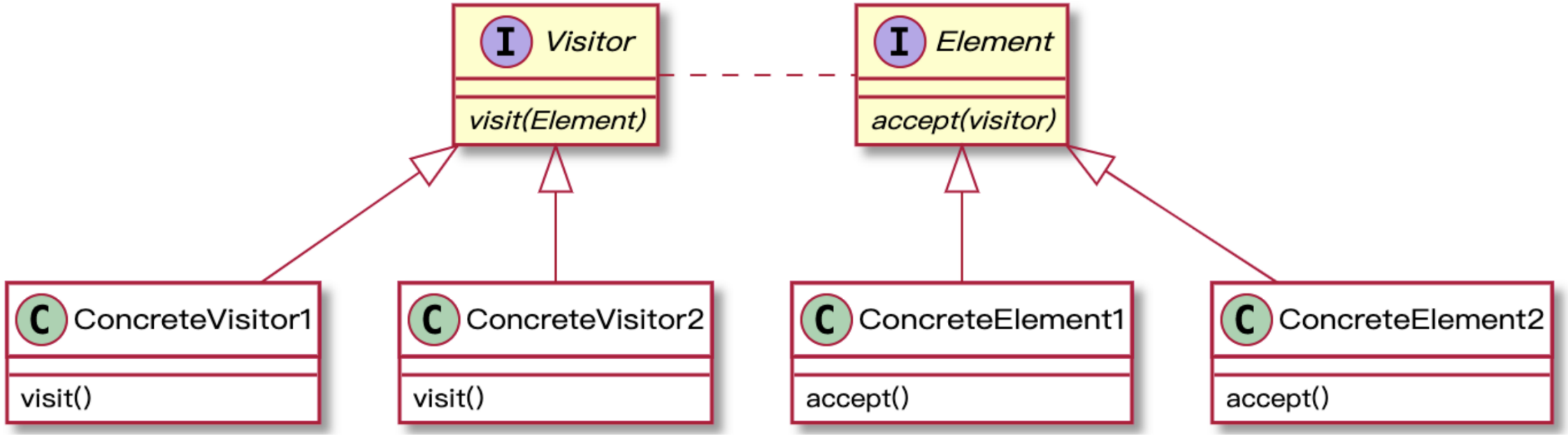
一个好的面向对象的设计通常是一组相互协作的类，这些类称为参与者。这些参与者在协作中担当不同的角色。

在上面的例子中，我们可以抽象出两个参与者：

- Element
  这表示结构中的元素，这是相对比较稳定的部分，负责遍历结构部分。
- Visitor
  这表示对结构中元素的访问者，负责对不同类型的对象做不同的处理。



Visitor模式的完整结构



- **Visitor**: 访问者，访问一个复杂结构中的元素。定义为抽象类或接口。
- **Element**: 元素，作为为访问的对象，需要定义一个accept操作。
- **ConcreteVisitor**: 具体访问者
- **ConcreteElement**: 对Element的实现，需要实现Element

参考代码

- Visitor模式是个中等复杂的模式，如果你的模型具有相对稳定的复杂的结构，经常需要在这个结构中遍历处理，这时候可以考虑使用visitor模式。
- 一个简单的visitor模式的实现可以参考deom-doc

什么是设计模式

模式是对反复发生的问题的解决方案

- 在遇到类似问题时可以迅速给出方案
- 有一个标准的名称，便于交流
- 初学者掌握面向对象设计技巧的捷径

GoF设计模式

Creational	Structural	Behavioral
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor