面向对象第四单元及学期总结

本单元架构设计

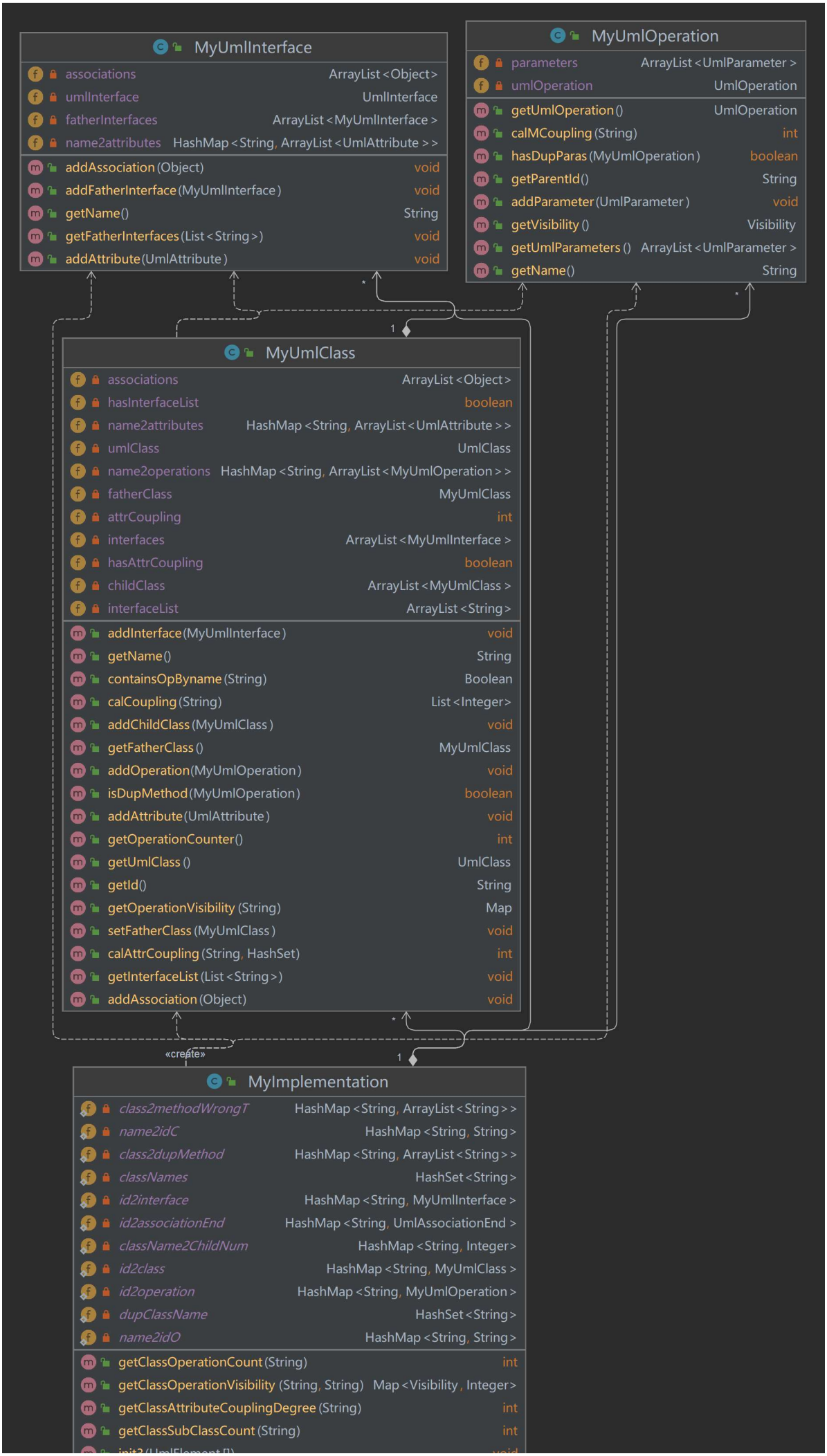
本单元基于对`UML`图的理解，实现对输入`UML`图的查询操作。

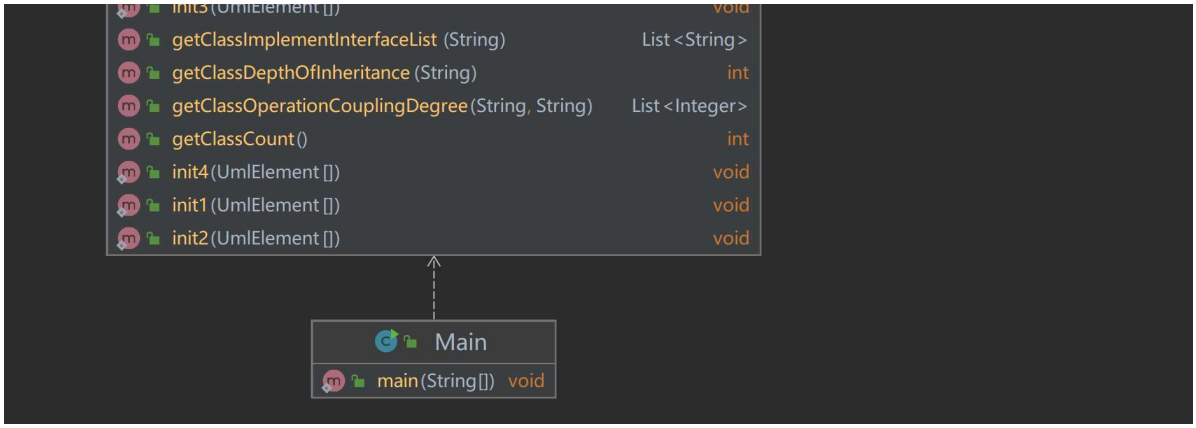
第一次作业

第一次作业只涉及类图的元素，每一个`UmlElement`有独一无二的`id`，且有唯一的一个父亲`parent_id`，即建立树状的层次结构，如一个`UMLClass`包含多个`UMLAttribute`和`UMLOperation`，一个`UMLOperation`又可以包含多个`UMLParameter`。对于这些需要存储与下一级元素关系的类进行了进一步封装，建立了自己的`MyUmlInterface`、`MyUmlOperation`、`MyUmlClass`三个类。

由于`UmlElement`按照id输入按照名字查找，故在顶层工具类`MyImpletation`保存了所有`id`与其`UmlElement`和`name`与其`id`的`HashMap`映射，经过多次循环遍历存储元素信息并建立关系。

`init1`	`UML_CLASS,UML_INTERFACE,UML_ASSOCIATION_END`
`init2`	`UML_ASSOCIATION,UML_ATTRIBUTE`
`init3`	`UML_OPERATION,UML_GENERALIZATION,UML_INTERFACE_REALIZATION`
`init4`	`UML_PARAMETER`


MyUmlInterface
associations ArrayList<Object>
umlInterface UmlInterface
fatherInterfaces ArrayList<MyUmlInterface>
name2attributes HashMap<String, ArrayList<UmlAttribute>>
addAssociation(Object) void
addFatherInterface(MyUmlInterface) void
getName() String
getFatherInterfaces(List<String>) void
addAttribute(UmlAttribute) void
MyUmlOperation
parameters ArrayList<UmlParameter>
umlOperation UmlOperation
getUmlOperation() UmlOperation
calMCoupling(String) int
hasDupParas(MyUmlOperation) boolean
getParentId() String
addParameter(UmlParameter) void
getVisibility() Visibility
getUmlParameters() ArrayList<UmlParameter>
getName() String
MyUmlClass
associations ArrayList<Object>
hasInterfaceList boolean
name2attributes HashMap<String, ArrayList<UmlAttribute>>
umlClass UmlClass
name2operations HashMap<String, ArrayList<MyUmlOperation>>
fatherClass MyUmlClass
attrCoupling int
interfaces ArrayList<MyUmlInterface>
hasAttrCoupling boolean
childClass ArrayList<MyUmlClass>
interfaceList ArrayList<String>
addInterface(MyUmlInterface) void
getName() String
containsOpByname(String) Boolean
calCoupling(String) List<Integer>
addChildClass(MyUmlClass) void
getFatherClass() MyUmlClass
addOperation(MyUmlOperation) void
isDupMethod(MyUmlOperation) boolean
addAttribute(UmlAttribute) void
getOperationCounter() int
getUmlClass() UmlClass
getId() String
getOperationVisibility(String) Map
setFatherClass(MyUmlClass) void
calAttrCoupling(String, HashSet) int
getInterfaceList(List<String>) void
addAssociation(Object) void
«create»
MyImplementation
class2methodWrongT HashMap<String, ArrayList<String>>
name2idC HashMap<String, String>
class2dupMethod HashMap<String, ArrayList<String>>
classNames HashSet<String>
id2interface HashMap<String, MyUmlInterface>
id2associationEnd HashMap<String, UmlAssociationEnd>
className2ChildNum HashMap<String, Integer>
id2class HashMap<String, MyUmlClass>
id2operation HashMap<String, MyUmlOperation>
dupClassName HashSet<String>
name2idO HashMap<String, String>
getClassOperationCount(String) int
getClassOperationVisibility(String, String) Map<Visibility, Integer>
getClassAttributeCouplingDegree(String) int
getClassSubClassCount(String) int



第二次作业

增加了对顺序图和状态图的查询，具体操作基本同第一次作业，一些方法的查询需要本人的难点在于一些图算法的实现……比较困难的指令例如：给定状态机模型和其中的一个状态，判断其是否是关键状态 `STATE_IS_CRITICAL_POINT statemachine_name statename`

我的做法是在初始化完所有`UmlElement`后遍历所有`stateMachine`查找关键状态并存储，调用指令时直接返回。

需要注意的就是搜索时对于空指针等特殊情况的判断...

```
public void findCriticalState() {
        if (finalStates.size() == 0) {
            return;
        }
        List<String> chain = new ArrayList();
        for (int i = 0; i < finalStates.size(); i++) {
            final2critical.add(new ArrayList<>());
            dfs(pseudostate.getId(), finalStates.get(i).getId(), chain, paths);
            if (paths.size() == 0) { continue; }
            List<String> list = paths.get(0);
            int j = 0;
            for (int k = 0; k < list.size(); k++) {
                String id = list.get(k);
                for (j = 1; j < paths.size(); j++) {
                    if (!paths.get(j).contains(id)) {
                        break;
                    }
                }
                if (j == paths.size() && !finalStatesId.contains(id)) {
                    final2critical.get(i).add(id);
                }
            }
            chain.clear();
            paths.clear();
        }
        int k;
        for (int i = 0; i < finalStates.size(); i++) {
            List<String> list = final2critical.get(0);
            for (int j = 0; j < list.size(); j++) {
```

```
                String id = list.get(j);
                if (finalStates.size() > 1) {
                    for (k = 1; k < finalStates.size(); k++) {
                        if (!final2critical.get(k).contains(id)
                                && final2critical.get(k).size() != 0) {
                            break;
                        }
                    }
                    if (k == finalStates.size()) {
                        nameOfCriticalState.add(id2state.get(id).getName());
                    }
                } else {
                    nameOfCriticalState.add(id2state.get(id).getName());
                }
            }
        }
    }

public void dfs(String curId, String finalId, List<String> chain,
List<List<String>> res) {
        MyState curState = id2state.get(curId);
        if (curState.getName() == null && !curId.equals(pseudostate.getId())) {
            if (curId.equals(finalId)) {
                res.add(new ArrayList<>(chain));
            }
            return;
        }

        for (String linkedId : curState.getLinkedStateId()) {
            if (chain.contains(linkedId)) { continue; }
            chain.add(linkedId);
            if (!linkedId.equals(curId)) {
                dfs(linkedId, finalId, chain, res);
            }
            chain.remove(chain.size() - 1);
        }
    }
```

第三次作业

检查`UML`模型中非法的地方并指出其类型。具体实现可分为在`Initialize`循环初始化时进行检查、在初始化完成后基于已经构建好的树结构再进行遍历检查两种方式。对于大部分不需要存储具体出错的`UmlElement`的`check`，选择设置一个全局变量`flag`，在初始化过程中就判断修改标志`flag`。

比较困难的实现是对于循环继承和重复继承的检查。

循环继承中类与接口分开处理，鉴于类的单继承特性只需要不断查找其父类并在这个过程中存储路径；对于可以多继承的接口，采用`tarjan`算法找出所有`size>=2`的连通块，另外在初始化时单独判断自环的情况。

重复继承只需要考虑接口。对于每一个接口按照树形结构一层一层更新自己的父接口集时，应该是对原有的扩充，不应出现重复，如果出现重复就把子接口加入集合。

同时为了减少工具类的代码量尽可能让各个类各司其职，单独提取了`MyCheck`、`MyInit`类分别用于遍历检查和完成部分初始化工作。