

第四单元手册

第四单元手册

手册说明

关于本单元作业的理解与应对策略建议

类图、顺序图 and 状态图元素关系图示

类图

顺序图

状态图

暖心帖：当继承关系遇到了关联关系

顺序图和状态图相关 [UML Element](#) 介绍

整体结构

类图

顺序图

状态图

顺序图与状态图主要子类一览

按照逻辑顺序整理主要子类

有关顺序图子类

有关状态图子类

结语

顺序图和状态图参考资料

手册说明

本手册用于大家第四单元的学习，涉及到了 UML 基本概念、作业限制、作业理解等多个部分。手册涉及的内容主要是**理论课的扩展**，以及帮助同学们理解本单元的实验与作业，**基础内容请参考理论课部分**。

本手册的主要内容结构如下：

1. 给出类图、顺序图和状态图元素关系图，对三个图元素结构有一个大致了解；
2. 给出 UML 类图中的继承与关联关系的一些讨论；
3. 给出 UML 顺序图与 UML 状态图元素介绍。

我们建议大家在实验与作业前学习以下内容：

1. 阅读本手册，理解 UML 的基本架构，熟悉类图、顺序图和状态图的基本元素和它们之间的关系；
2. 学习使用 StarUML，了解 StarUML 绘图方法，能够看懂简单的树形图；
3. 学习阅读 mdj 文件，能够快速通过 mdj 获取信息；
4. 完成第四单元训练内容。

关于本单元作业的理解与应对策略建议

注：该部分是吴际老师给同学们的建议，希望大家能看完。

同学们刚刚结束 JML 单元的痛苦折磨，马上又愉快的掉入 UML 单元的学习中（别不承认，哈哈）。这个帖子的目的是帮助同学正确理解这个单元的学习目标，并在作业及实验训练过程中能够很好完成任务。

同学们可以，也应该去学习 UML 语言的表示结构和常见用法，这个很重要，资料也非常多，甚至也有不少 MOOC 课程在讲这个。我们首先要抓住 UML 是谁，它能干什么这两个基础问题，然后才能用好 UML。如课上 PPT 所言，UML 是一种统一建模语言，“统一”这个词内涵丰富，意味着在 UML 之前是百家争鸣。所有的建模语言都有一个共性目标，就是帮助用户抓住问题域或者解决方案域中的核心概念、

结构和逻辑，通过抽象的方式来表达出来（**抽象意味着一定要忽略掉一些细节**）。UML 主要是把三个分支的建模语言特征和表达方式进行了合并，并多次演化发展，成为今天大家所看到的样子。UML 语言形成统一之后（有点类似于秦始皇统一了文字的味道），首先必须解决的一个问题是，如何让不同建模工具所产生的模型（图）能够“互认”。不要小看这个问题，这其实仍然是目前计算机领域的一个让人头疼的问题。

简单来说，这个“互认”问题的解决方案经历三个阶段的设计和演化。**第一个阶段，是人读图的“互认”**，即不同工具必须按照统一的图形化表示规范和语法来绘制和现实 UML 图，这形成了我们所看到的 UML 各种图的语法规则基础。**第二阶段，工具读图的“互认”**，即不同建模工具能够按照统一的方式来解析 UML 图，这带来的结果就是课件中所提到的 UML 元模型，统一官方定义的方式发布了 UML 元模型标准，所有建模工具都支持（MS Visio 是个另类，所以我们一般不把 Visio 称为建模工具，而是一个灵活的画图工具）；**第三阶段，工具交换的“互认”**，即一个工具构造的模型图能够在另一个工具中打开和编辑。由此可以看出，UML 建模的一个基础问题就是如何解析和理解 UML 模型图中所表达的元素及其关系。这个视角不难理解，但是如果这个基础问题不解决，人就难以**在计算层次来理解 UML 模型**。

为什么强调在计算层次理解 UML 模型？这要从软件开发方法发展的角度来谈，篇幅也会很长，我不打算在这里絮叨太多。简单归结，在计算层次理解 UML 模型使得计算机具备了理解和推理 UML 模型语义的能力，从而使得工具可以做更多的事情，比如自动验证，比如自动生成代码，比如自动生成测试用例等。没错，这些功能都依赖于在计算层次理解 UML 模型。这就是本单元训练的切入点，和国内其他高校的做法可能都不太相同。为了对 UML 有所铺垫，我们有意在课程讲授中使用 UML 图来表达一些知识点和逻辑结构，并让同学们在单元总结博客中使用 UML 来整理设计结构。尽管我们强调最好自己动手画 UML 图，但大部分同学都是使用工具自动从代码提取和生成类图，这是比较有遗憾的地方。

上述分析，实际给出了本单元作业理解与应对的两个基本策略：**准确理解 UML 图的表示结构，准确理解 UML 元模型的表示结构**。我们首先必须要能够在 UML 图的层次来理解有哪些元素（注意每种元素都有一个类型定义，如 UMLAttribute），元素的连接规则，元素的表达方式（如图符）；然后在元模型层次理解这些元素的具体定义和关系，比如 UMLClass 与 UMLAttribute 的关系。在建立了这两个准确理解的基础后，再去完成本单元的作业。

注：以 StarUML 角度而言，UML 模型具体存储在 mdj 文件中，它是模型的序列化。作为带数据模型的 JSON 文件，希望同学们能够透彻理解其格式，尤其是背后的数据模型，好处是不小的。

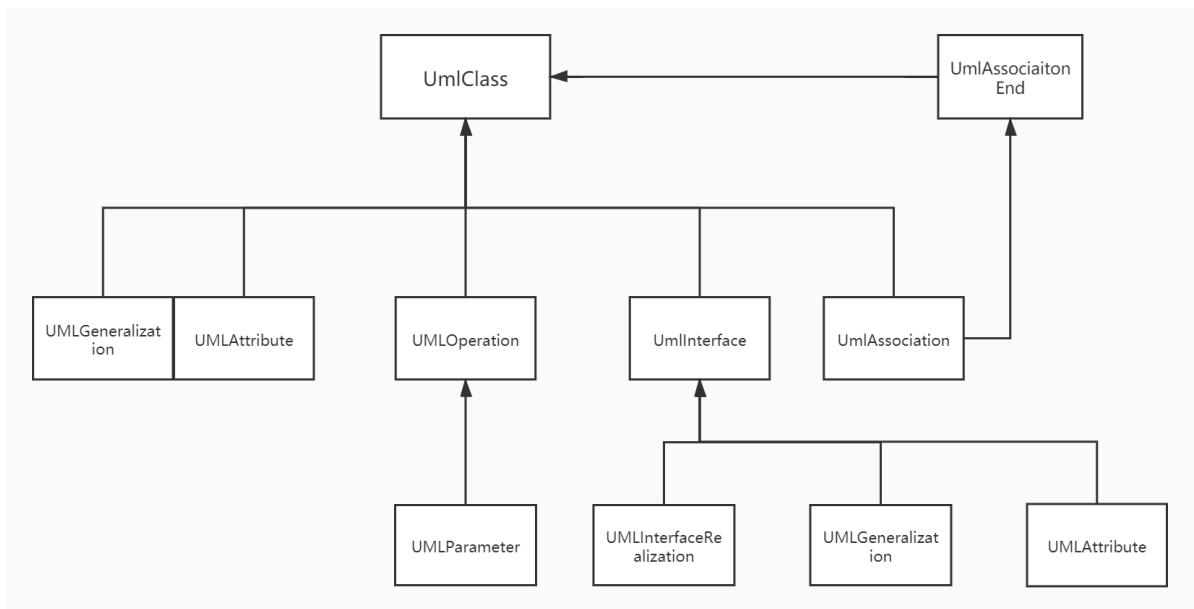
在此送上一句吴老师的话：一定要打开 json 文件(mdj 文件)，浏览其树形结构，对照模型图观察每个元素的内容和其所管理的下层数据对象。

类图、顺序图和状态图元素关系图示

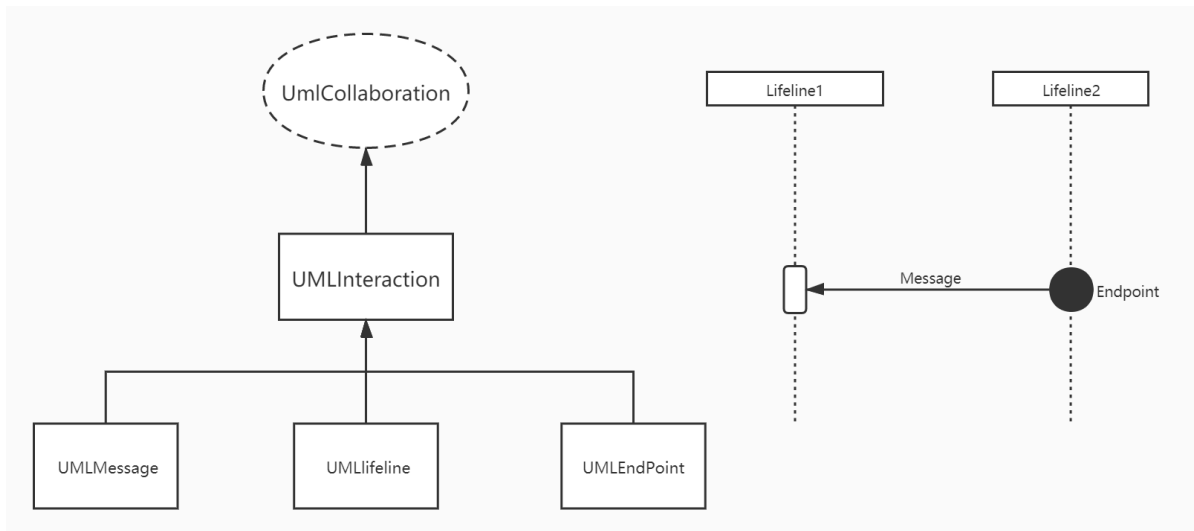
为了方便整体理解，下面给出分别代表了类图、顺序图和状态图元素之间的基本关系图。

如果你目前对 UML 图涉及的元素还不太清楚，可以直接忽略这个章节继续往后看，等全部看完后再回看本章节，你将会对这三张图有一个更完整的理解。

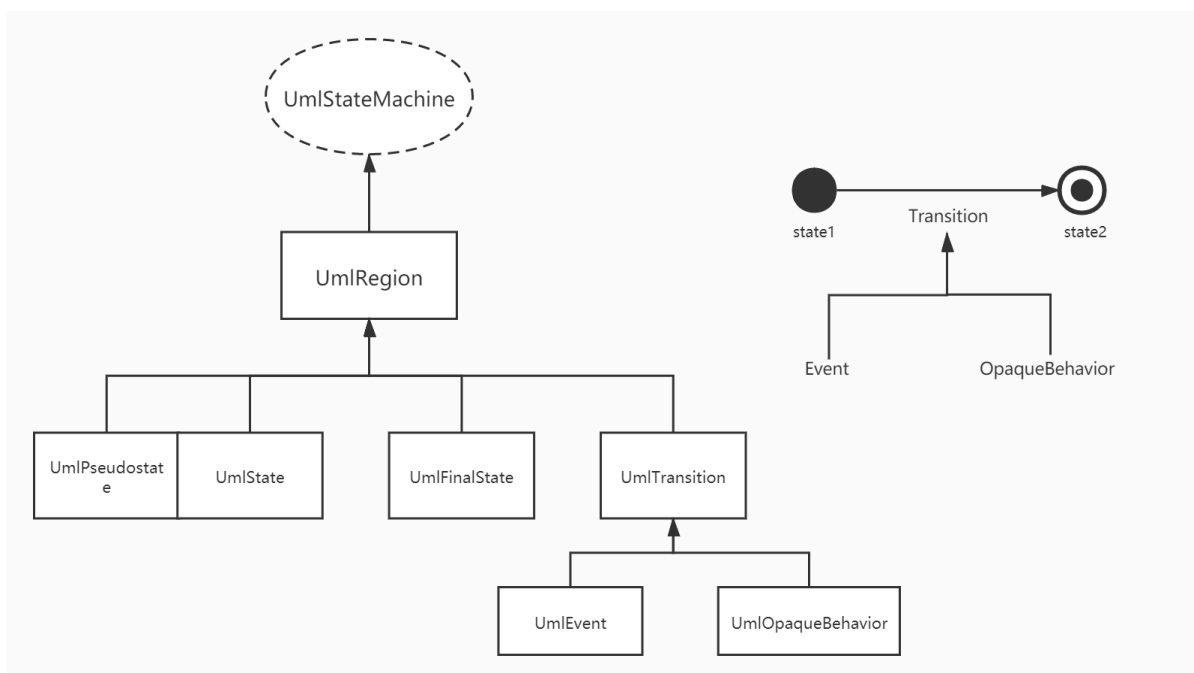
类图



顺序图



状态图



暖心帖：当继承关系遇到了关联关系

注：本帖由吴际老师撰写。旨在帮助同学们理解 UML 类图中和“继承”“关联”相关的一些计算问题。

根据往年经验，本单元难

点更在于对 UML 图的理解，如果同学们有其他关于理解 UML 相关的问题，也欢迎在讨论区进行讨论 or 提问。

注意，本帖的目的是帮助同学们来梳理一些基础概念。如果对这些概念理解不当，可能会带来程序功能实现上的错误。此外，本帖所讲的概念和规则都是针对 UML，而不是 Java。不过为了便于同学们对照，会专门讲一下在相应概念的表达上，Java 与 UML 的可能差异。

我们都知道，当你创建一个类，比如 A，你可以直接为此定义属性（UMLAttribute）和操作（UMLOperation）。当我们问类 A 的属性个数或者操作个数时，直接数数即可。如果这时候另一个类 B 继承了类 A，按照继承的语义，B 自动获得了 A 的所有属性和操作。因此，我们有规则：

```
B.attriCount = A.attriCount + B.self.attriCount (rule 1)
```

其中 `B.attriCount` 表示 B 所拥有的属性数目，`B.self.attriCount` 表示 B 类自身所定义的属性个数。对于操作也有相应的规则：

```
B.operaCount = A.operaCount + B.self.operaCount (rule 2)
```

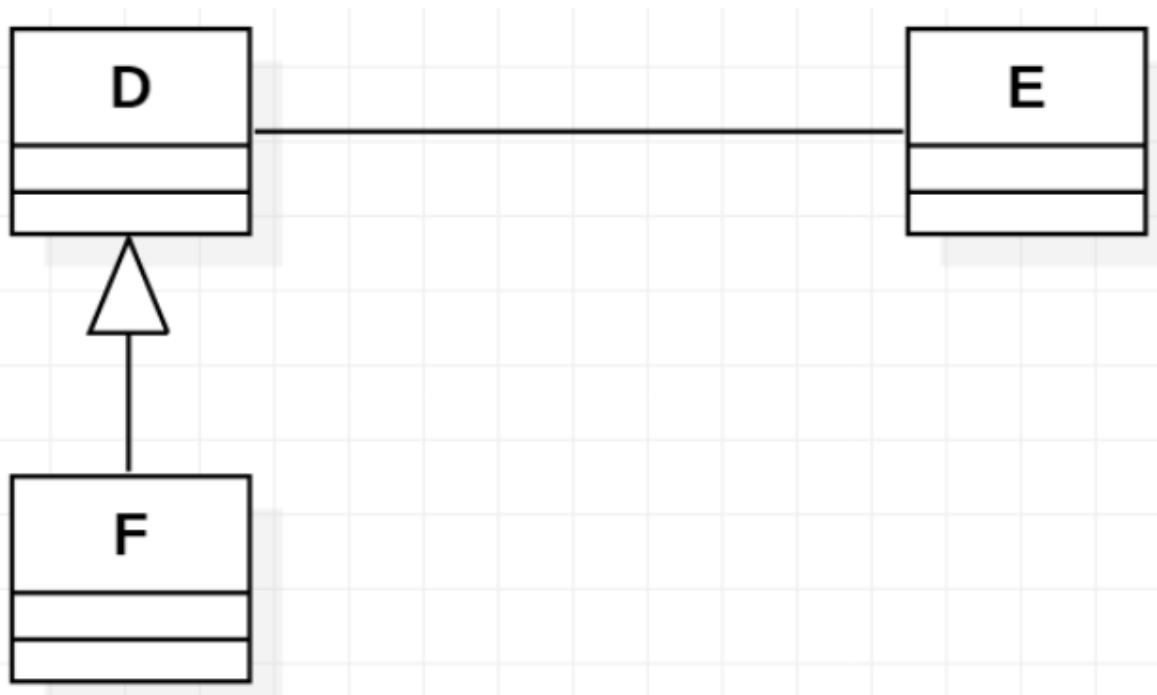
好了，现在关联表示也想参与到这个游戏中来。这带来了至少两个问题：

1. 关联关系是否会被继承下来？
2. 如果能够被继承下来，相应的规则该如何计算？

先来看问题 1。关联的本质是一个类需要另一个类来管理数据或者请求服务。



如图所示，B 需要 A，A 也需要 B。问题来了，如果 A 或者 B 不记录下对方是谁，就无法在需要时来请求对方的服务了。因此，A 和 B 都必须记录下对方，正如上次课上交代过（那个 Course 和 Student 的例子）。这时，意味着 A 中要有一个使用类 B 来定义的数据，假设关联两端的 multiplicity 都是 1，则表明 A 中有一个数据为 `B bobj`，B 中有一个数据为 `A aobj`。这样当 A 需要请求 B 的服务时，直接通过 `bobj.func(...)` 来请 B 的 `func` 操作。不管怎么样，要记住：**关联关系是双向关系**，即从 A 类对象这端能访问到所关联的 B 类对象，同理，从 B 类对象这端也能访问到所关联的 A 类对象。现在我们加入一个继承来看看会怎么样：



此时类 F 继承自 D。我们已知 D 具有找到 E 的能力，F 必然也具备这个能力。所以，我们要明确的一条：**F 类同时继承了 D 类拥有的关联**。仅就这个例子而言，D 类的关联数为 1（`D.assoCount = 1`），E 类的关联数也是 1（`E.assoCount = 1`）。那么 F 类呢？答案是：`F.assoCount = 1`，特别地，`F.self.assoCount = 0`，即 F 类自己没有定义任何关联，唯一的关联是从父类 D 继承而来的。所以我们有规则：

`F.assoCount = D.assoCount + F.self.assoCount` (rule 3)

至此，看起来问题 2 解决了。但是有人提出了新的问题，如果此时接口实现也趁乱加入进来呢？

特此声明：接口实现不会建立任何上层和下层在属性、操作和关联关系数量上的连接。因此，**在回答一个类有多少个（或者哪些）属性/操作/关联时，可以直接无视接口实现关系。**

好了，这个时候我们该来见识一下 UML 的神奇之处：对于类图而言，能在哪些元素之间建立继承关系？仅就我们课上提到过的概念，当发现 A 继承自 B 时，A 可以是 UMLClass 对象，也可以是 UMLInterface 对象；B 同样可以是 UMLClass 对象，也可以是 UMLInterface 对象。

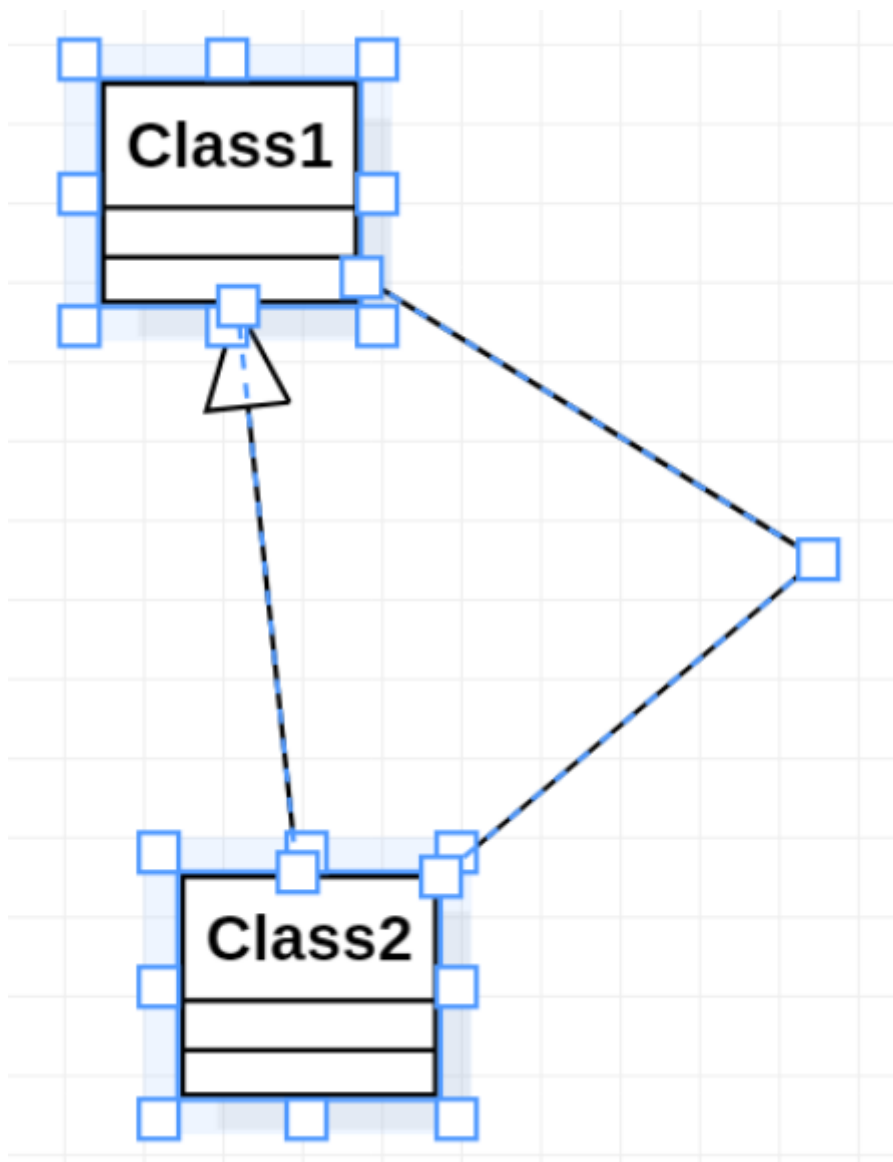
注意：Java 语言做了严格限制，即类只能继承类（*superclass vs subclass*），接口只能继承接口（*superinterface vs subinterface*），二者不可交叉。如果要交叉，只有一种合法情形，即类实现接口。

另外一方面，不论是类，还是接口，都能够和其他类或者接口建立关联关系。即一个类关联到类，也可以关联到接口；一个接口可以关联到类，也可以关联到接口。UML 和 Java 在这点上达成了共识，保持一致。

对于 UML 模型而言，当遇到了 A 继承 B，不论 A 和 B 是 UMLClass 还是 UMLInterface 的实例，统统认为没有区别，此时 rule 1，rule 2 和 rule 3 仍然适用。

好吧，现在来做几个稍微有点绕（但必须承认其实有点小变态）的 case。

A 继承 B，A 同时关联 B



此时如果我们拿掉 Class2 到 Class1 的继承，我们知道 Class1 的关联数量和 Class2 的关联数量都是 1。好吧，把继承关系放回去，此时 Class1 的关联数量仍然是 1（这个显而易见），Class2 的关联数量变成了 2。Class2 本身定义的关联对端是 Class1；Class2 从 Class1 继承得到的关联，其对端是 Class2，这意味着 Class2 既可以关联到 Class1，也可以关联到 Class2（这个是继承来的能力）。所以：

```
Class2.assoCount == Class1.assoCount + Class2.self.assoCount
```

```
==> Class2.assoCount == 1 + 1 = 2
```

最后，来做一个更加复杂一点的例子：

- `C.assoCount = 2` : (C--A, C--IA)
- `A.assoCount = 2` : (A--C, A--B)
- `IA.assoCount = 2` : (IA--C, IA--IB)

此时问题如下：

- 问题 1: `B.assoCount = ?`
- 问题 2: `IB.assoCount = ?`

由上图还可以看到一个 UML 和 Java 的差异之处：

- UML：接口可以继承接口，也可以实现接口
- Java：接口可以继承接口，但不可以实现接口。

顺序图和状态图相关 UMLElement 介绍

注：此部分为历届助教团队和其他同学不断迭代完善而成。

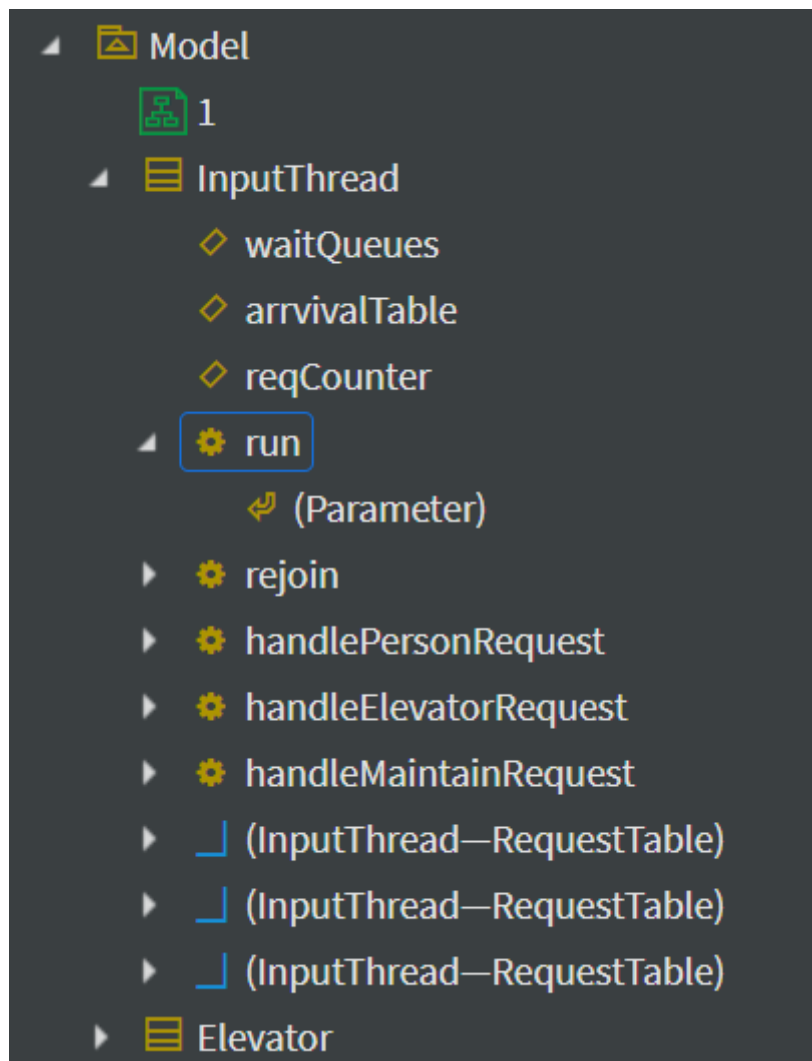
整体结构

类图

类图是面向对象系统建模中最常用和最重要的图，是定义其它图的基础。类图主要是用来显示系统中的类、接口以及它们之间的静态结构和关系的一种静态模型。类图（UMLModel）的大致结构如下：

- UMLModel
 - UMLClassDiagram
 - UMLClass
 - attributes
 - UMLAttribute
 - operations
 - UMLOperation
 - parameters
 - UMLParameter

StarUML 中的结构：



顺序图

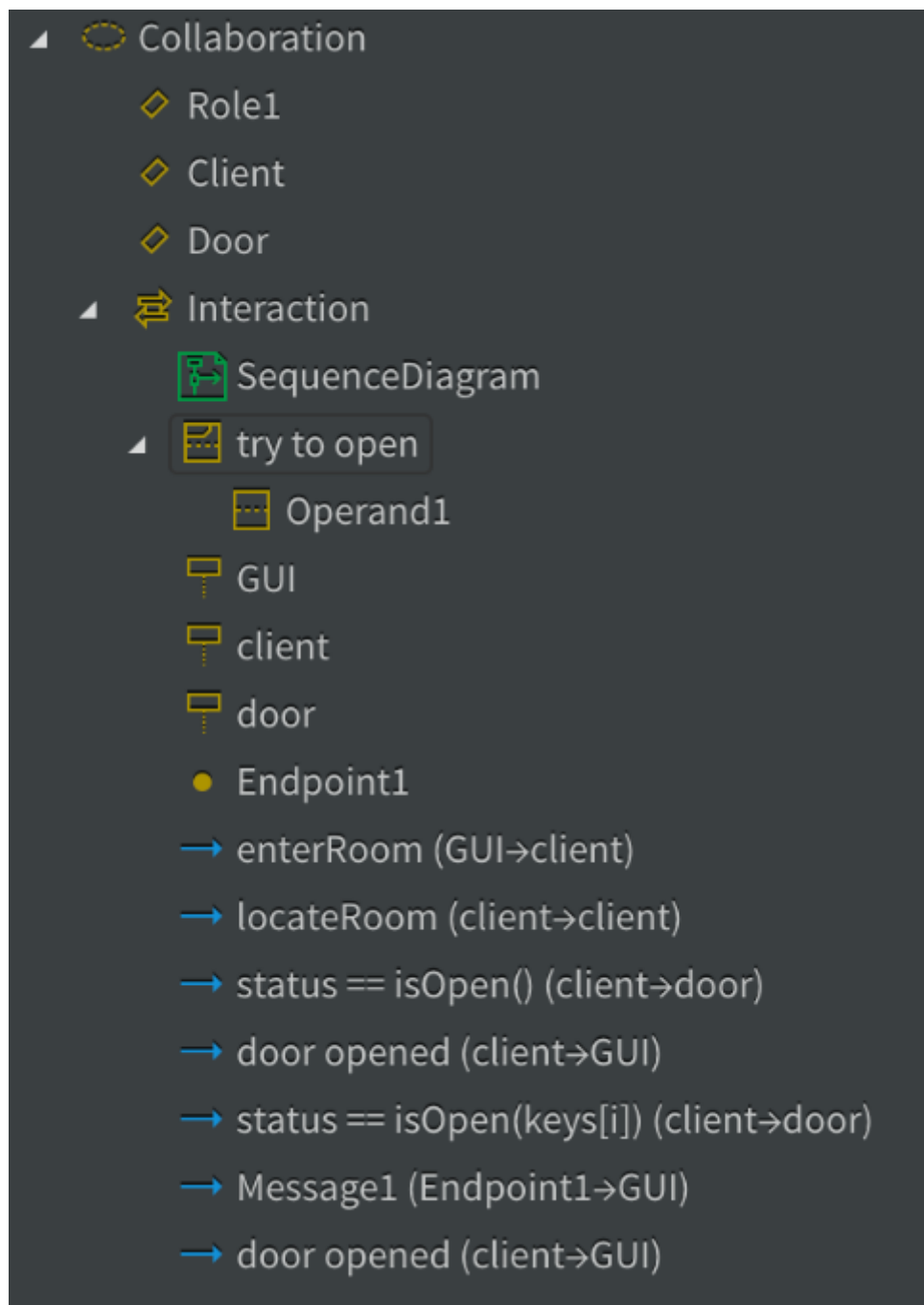
顺序图定义了对象的协同（`UMLCollaboration`），其中包括了 `ownedElements` 和 `attributes` 两个属性。`UMLCollaboration` 的作用有点类似于状态图中的 `UMLRegion`，其中包含了所有为了完成协同相关的元素。

- `ownedElements` 主要由一个或多个类型为 `UMLInteraction` 的对象组成。
- 其中 `UMLInteraction` 类型的对象表示协同行为，用于描述特定主题下的交互行为。包括以下三个属性：
 - `messages` (`UMLMessage`)：各种类型的消息
 - `participants` (`UMLLifeLine / UMLEndpoint`)：对象生命线和 UML 终结点。
 - `fragments` (`UMLCombinedFragment`)：消息控制块（循环、条件判断、同步、可选等等）
- `attributes`：表示来完成协同行为的属性成员（对象），由于一个协同行为可能涉及多个对象，所以主要由多个类型为 `UMLAttribute` 的对象组成。

大致结构：

- `UMLCollaboration`
 - `ownedElements`
 - `UMLInteraction`
 - `ownedElements`
 - `messages` (`UMLMessage`)
 - `participants` (`UMLLifeLine / UMLEndpoint`)
 - `fragments` (`UMLCombinedFragment`)
 - 另外的 `UMLInteraction`
 - `attributes`
 - 若干 `UMLAttribute`

StarUML 中的结构：



状态图

`UMLStateMachine` 是 `UMLClass` 中 `ownedElements` 的一员；`UMLStateMachine` 是一个对象容器，由两部分组成：`ownedElements` 与 `regions`。

- **重点**是 `UMLRegion`，每一个 `UMLStateMachine` 有一个 `UMLRegion`，其作用与画画中的“画布”类似，把状态图中的状态和状态的迁移过程，杂七杂八的画在上面，由 `UMLRegion` 统一管理。另外还可以建立层次化状态机。

大致结构：

- `UmlClass`
 - `ownedElements`
 - `UMLStateMachine`
 - `ownedElements`
 - `regions`：一个或多个 `UmlRegion`
 - `vertices`：一个或多个 `UmlState`

- `transitions`: 一个或多个 `Transition`

StarUML 中的结构:

- StateMachine
 - StatechartDiagram1
 - (Region)
 - (Pseudostate)
 - (Pseudostate)
 - tring2Open
 - locker.unlock(key)
 - trying = trying + 1
 - blocked
 - closed
 - beOpen = false
 - opened
 - beOpen = true
 - (FinalState)
 - open (→tring2Open)
 - (tring2Open→blocked)
 - open (tring2Open→tring2Open)
 - failed (blocked→)
 - reset (blocked→tring2Open)
 - Reset
 - trying = 0
 - close (opened→closed)
 - Close()
 - locker.lock()
 - (tring2Open→opened)
 - open (closed→tring2Open)
 - Open(key)
 - trying = 0

顺序图与状态图主要子类一览

- `UmlEndPoint`
- `UmlEvent`
- `UmlFinalState`
- `UmlInteraction`
- `UmlLifeline`
- `UmlMessage`
- `UmlOpaqueBehavior`
- `UmlPseudostate`
- `UmlRegion`
- `UmlState`
- `UmlStateMachine`
- `UmlTransition`
- `UmlCollaboration`
- `UmlCombinedFragment`

按照逻辑顺序整理主要子类

在本节中，如无特殊说明，代码块内的为对应的子类的样例。样例中 `_parent` 字段经过了简化表示，实际应为

```
"_parent": {"$ref": 该字段原来的值}
```

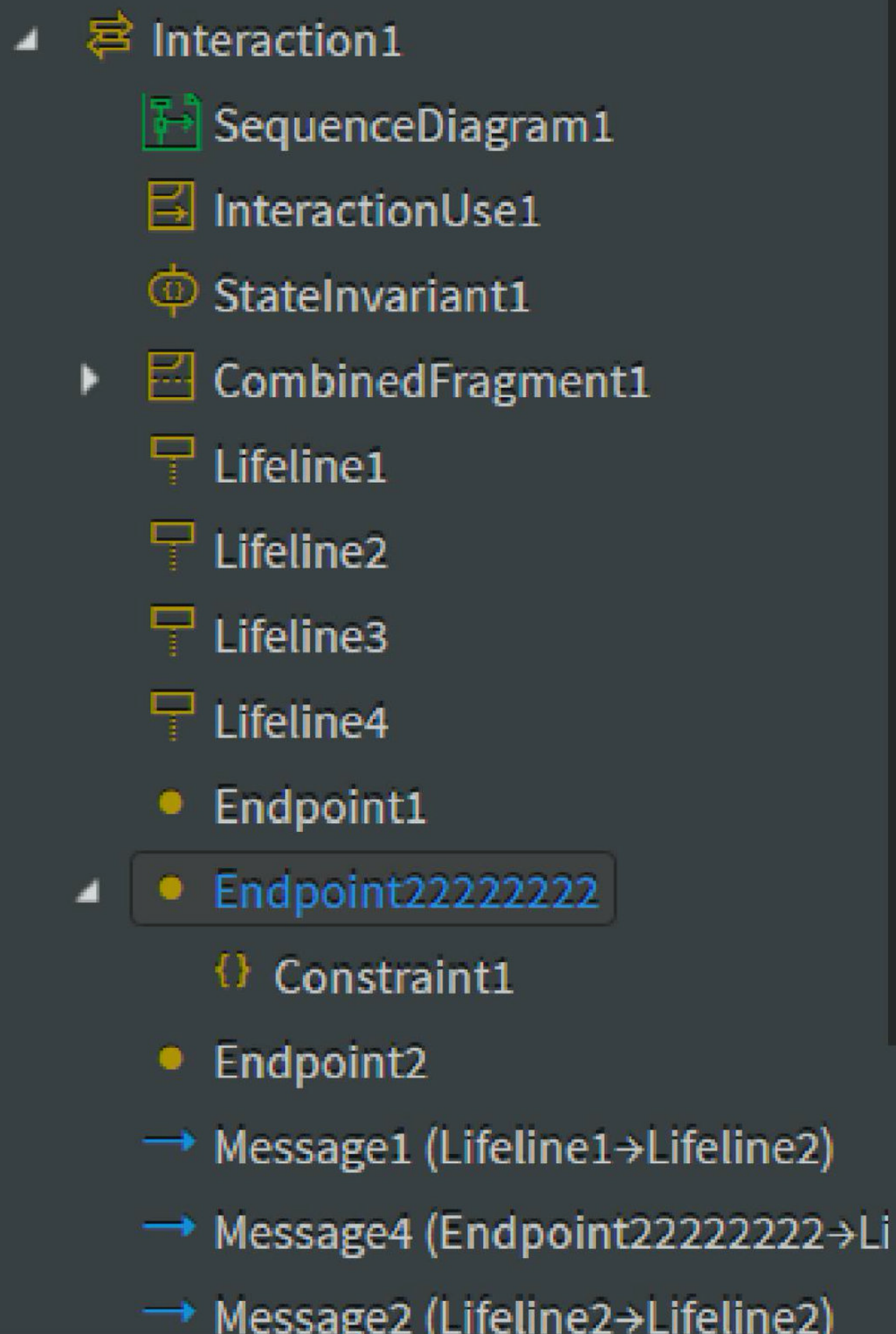
实际上表示对该 mdj 文件中其它对象的引用。

有关顺序图的子类

0. `UmlInteraction`

```
{
  "_parent": "AAAAAAFrZNSJhnMP04E=",
  "visibility": "public",
  "name": "Interaction1",
  "_type": "UMLInteraction",
  "_id": "AAAAAAFrZNSJhnMQrSk="
}
```

顺序状态图中的交互，应该每个顺序图对应一个交互的子类。其 `_parent` 为 `UmlCollaboration`。感觉有点像状态图中的画布 `UmlRegion`，顺序图中出现的所有消息交互行为都在其下：



```
{
  "_parent": "AAAAAAFrZNSJhnMQrSk=",
  "visibility": "public",
  "name": "Lifeline1",
  "_type": "UMLLifeLine",
  "isMultiInstance": false,
  "_id": "AAAAAAFrZNSX43Mf80k=",
  "represent": "AAAAAAFrZNSX43Me0dw="
}
```

顺序图中的生命线，表示对象的生存时间，用矩形下连虚线表示：

Lifeline1

1 : Message1



5 : Rely



`name` 的命名方法有以下几条规则：

- 通常矩形框内为的命名格式为

```
\[对象名][:类名]
```

- 具体有以下几种方式：
 - 对象名
 - 类名:对象名
 - 类名
 - 此时表示匿名对象。
- 当使用下划线时，意味着顺序图中的生命线代表一个类的特定实体。
- 每个生命线都关系到一个实体。

其 `_parent` 为 `UmlInteraction`，因为 `UmlLifeline` 属于 `UmlInteraction` 的属性 `participants`。

每个 `UmlLifeline` 都关联到一个 `UmlAttribute`，即对应到一个具体对象，这个对象对于这个 `UmlInteraction` 而言是一个属性 `represent`，表示实例化数据：

```
// UmlLifeline
{
  "_type": "UMLLifeline",
  "_id": "AAAAAFydGqEOanpQSE=",
  "_parent": "AAAAAFydGp7hana4o8=",
  "name": "Lifeline1",
  "represent": "AAAAAFydGqEOanoogE=",
  "isMultiInstance": false
},
{
  "_type": "UMLLifeline",
  "_id": "AAAAAFydGqap6oIUuc=",
  "_parent": "AAAAAFydGp7hana4o8=",
  "name": "Lifeline2",
  "represent": "AAAAAFydGqap6oHLJE=",
  "isMultiInstance": false
}
```

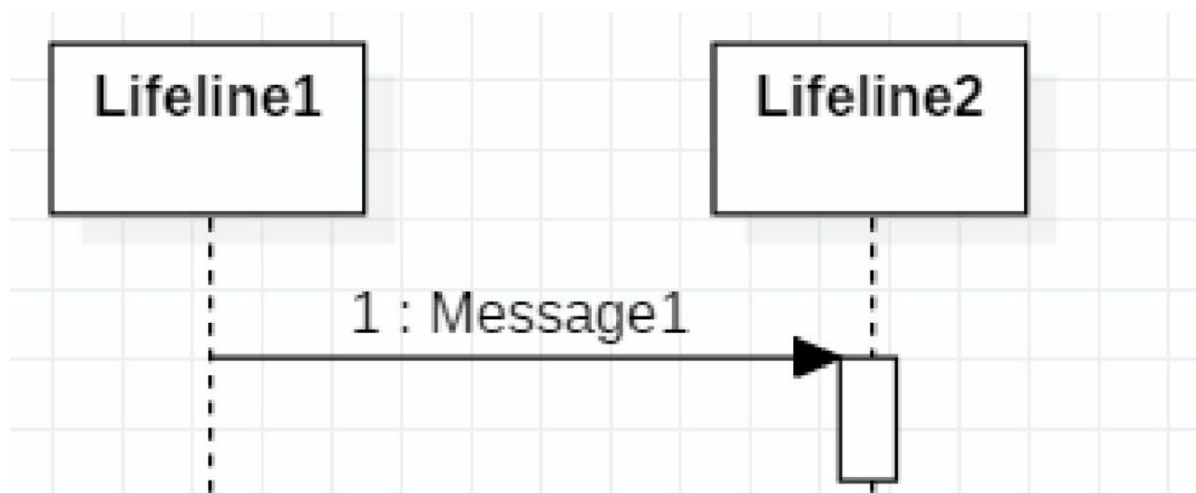


```
// UmlAttributes
"attributes":
{
  "_type": "UMLAttribute",
  "_id": "AAAAAFydGqEOanooGE=",
  "_parent": "AAAAAFydGp7hKnZcpk=",
  "name": "Role1",
  "type": ""
},
{
  "_type": "UMLAttribute",
  "_id": "AAAAAFydGqap6oHLJE=",
  "_parent": "AAAAAFydGp7hKnZcpk=",
  "name": "Role2",
  "type": ""
}
```

1. UmlMessage

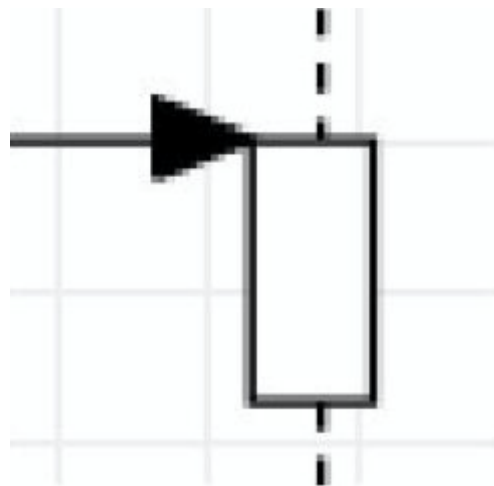
```
{
  "messageSort": "synchCall",
  "_parent": "AAAAAFrZNSJhnMQrSk=",
  "visibility": "public",
  "name": "Message1",
  "_type": "UMLMessage",
  "_id": "AAAAAFrZNTLwXN7x1w=",
  "source": "AAAAAFrZNSX43Mf80k=",
  "target": "AAAAAFrZNSojXM+cCs="
}
```

顺序图中的消息，用黑实线和箭头表示：对象之间的**交互**是通过**相互发消息**来实现的。一个对象可以请求（要求）另一个对象做某件事。消息从源对象指向目标对象。消息一旦发送便从源对象转移到目标对象。其 `_parent` 为 `UmlInteraction`，因为 `UmlMessage` 属于 `UmlInteraction` 的一个属性 `messages`。



消息也分成六类：

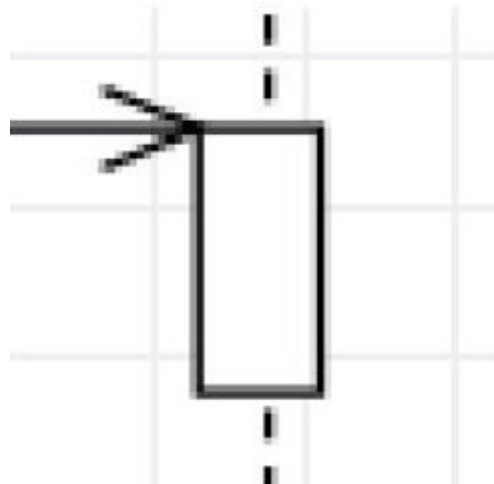
1. 同步消息，用黑三角箭头搭配黑实线表示：



同步

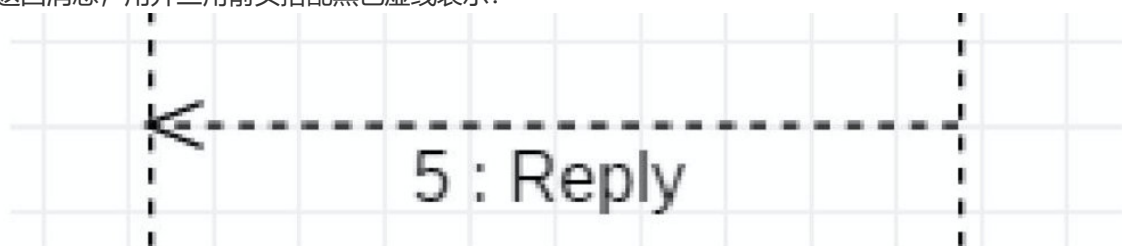
的意义：消息的发送者把进程控制传递给消息的接收者，然后**暂停活动**，**等待**消息接收者的回应消息。

2. 异步消息，用开三角箭头和黑色实线表示：



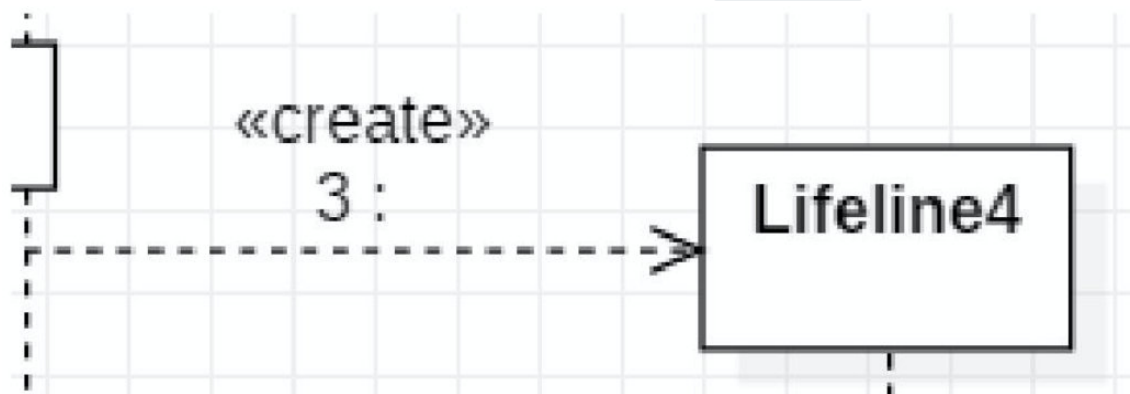
异步的意义：消息的发送者将消息发送给消息的接受者后，**不用等待回应的消息**，即可**开始另一个活动**。

3. 返回消息，用开三角箭头搭配黑色虚线表示：



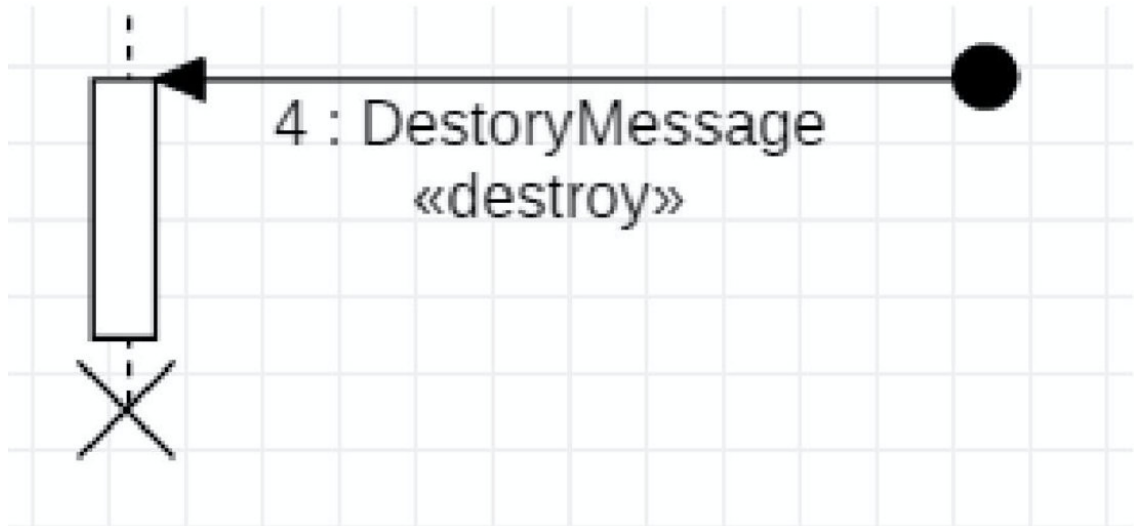
返回消息和同步消息结合使用，因为异步消息不进行等待，所以不需要知道返回值。

4. 创建消息，用开三角箭头搭配黑色虚线表示，其下面特别注明 `<<create>>`



创建消息用来创建一个实例，可以测试出，若指向一个声明线的中部，StarUML 会自动将目标生命线移动到创建消息的地方开始，其上方不存在。

5. 摧毁消息，用黑三角箭头搭配黑实线表示，其下面特别注明 `<<destroy>>`



摧毁消息用来摧毁一个实例，生命线上会出现一个 X 表示结束。

6. Lost and Found Message，这类消息的特点是它可能没有发送者或者接收者，用一个黑色实心的点和黑色实心三角箭头黑实线表示：



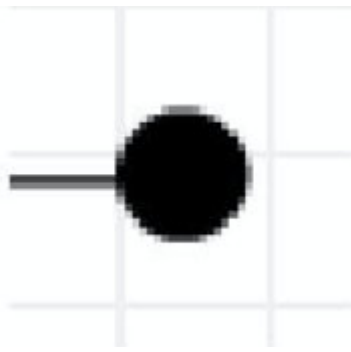
- Lost and Found Message 实际应为两种：Lost Message 和 Found Message。
- 有时候我们不需要关心发送者是哪个对象，此时称该消息为 **Found Message**，其 `source` 值为发送者 `UmlEndpoint`。
- 类似地，有时候我们不需要关心接收者是哪个对象，此时称该消息为 **Lost Message**，其 `target` 值为接收者 `UmlEndpoint`。

注：在某些地方还细分了一个简单消息，不区分同步和异步，StarUML 中省去了。

3. `UmlEndPoint`

```
{
  "_parent": "AAAAAAFrZNSJhnMQrSk=",
  "visibility": "public",
  "name": "Endpoint1",
  "_type": "UMLEndpoint",
  "_id": "AAAAAAFrZNVi13Pksa4="
}
```

顺序图中的终结点，用黑色实心圆点表示：



终结点通常和 Lost and Found 消息搭配使用，表示从非图中生命线地方发出(或接受)的消息。

其 `_parent` 为 `UmlInteraction`，与 `UmlLifeline` 类型同理。

4. `UmlCollaboration`

`UmlCollaboration` 是表示 UML 协作图的虚元素。

```
{
  "_type": "UMLCollaboration",
  "_id": "AAAAAAF5ugLrwsIIEl4=",
  "_parent": "AAAAAAFF+h6SjaM2Hec=",
  "name": "Collaboration1"
}
```

其 `_parent` 直接指向 mdj 文件作为 JSON 对象考虑时的 `_id` 字段，相当于根元素的 ID。（本次作业中用不到，所以不用考虑。）

5. `UmlInteractionOperand` (了解)

代表 UML 顺序图中的某个交互，级别比发出消息更宏观。其 `_parent` 为 `UmlCombinedFragment`。

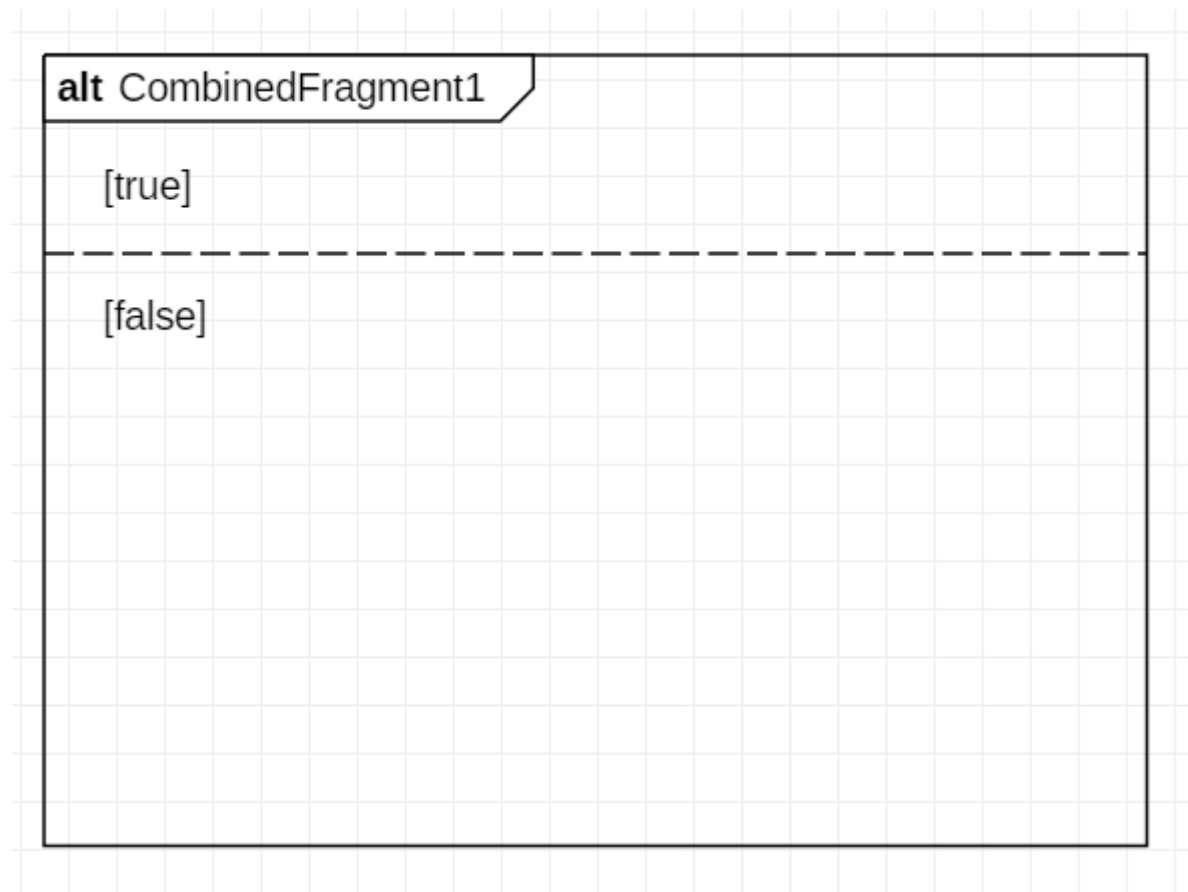
```
{
  "_type": "UMLInteractionOperand",
  "_id": "AAAAAAF5vTEGzUOqCqM=",
  "_parent": "AAAAAAF5vTEGzUOpYdw=",
  "name": "Operand1",
  "guard": "true"
}
```

6. `UmlCombinedFragment` (了解)

属于 UML 顺序图中的一个组合片段，片段本身包含条件结构，自然也能改变顺序图中的消息流以及程序的语义。如何改变需要看 `interactionOperator` 属性。注意 `Operator` 与 `Operand` 的用词。通常用来表示顺序图中的控制与逻辑结构，也可以嵌套。其 `_parent` 为 `UmlInteraction`，因为其属于 `UmlInteraction` 的一个属性 `fragments`。

```
{
  "_type": "UMLCombinedFragment",
  "_id": "AAAAAAF5vTEGzUOpYdw=",
  "_parent": "AAAAAAF5vTDdCkObHaA=",
  "name": "CombinedFragment1",
  "interactionOperator": "alt"
}
```

类型在左上方标签最左边，名称在其右。可能会有以水平虚线分开的分支，不同分支中每个分支都对应自己的唯一——一个 `UmlOperand`。



其 `interactionOperator` 属性有多种取值，分别代表该组合片段的语义。实际上该选项仅仅改变左上角的粗体字，故图略去。

- **alt** (Alternatives)：代表选择有多种可选项的行为。
 - 如果某个 `UmlInteractionOperand` 没有条件 (guard)，就意味着它有一个隐含永远为真的条件。
 - 如果其 guard 为 `else`，其它 `UmlInteractionOperand` 的 guard 均不满足时会选择它。
- **opt** (Option)：代表某一个行为是否发生，要么发生，要么不。语义上与有两个 `UmlOperand`，一个有其内容且 guard 与之相同而另一个为空的 **alt** 组合片段等价。
- **loop**：代表循环，内容会重复执行多次，类似 `seq` 操作子迭代执行若干遍。该选项在左上角的粗体字渲染时，有三种方法，分别代表对循环次数的控制。
 - **loop**
 - **loop(最少循环次数)**
 - **loop(最少循环次数:最多循环次数)**
- **break**：代表紧急内容，若 guard 为真，其内容会被执行，其外直接嵌套顺序图或组合片段之后的内容不再执行。否则忽略其内容。必须有 guard，否则语义不确定。
- **par** (Parallel)：之后的几个 `UmlInteractionOperand` 中的内容并行执行。
- **strict** (Strict sequencing)：要求直接嵌套的一级的 `UmlInteractionOperand` 中的内容严格按照先后顺序执行，再深的嵌套中的内容执行顺序不做要求。
- **seq** (Weak sequencing)：类似 **strict**，但直接嵌套进去的某个 `UmlInteractionOperand` 中，不同时间线发出的不同事件顺序任意。
- **critical** (Critical Region)：确定临界区，使进入临界区的操作原子化，通常与 **par** 一起使用。

- `ignore`：认为某些消息不重要，忽略这些种类。通常在渲染时，用 `ignore{消息种类1,消息种类2,...}` 表示消息的种类。
- `consider`：与 `ignore` 相反，在渲染时用 `consider{消息种类1,消息种类2,...}` 表示种类，不在其中的忽略。
- `assert` (Assertion)：断言某种时序的出现。
- `neg` (Negative)：包括系统失败时应出现的内容。

有关状态图的子类

0. `UmlRegion`

```
{
  "_parent": "AAAAAAFqyeHHXDE0fXE=",
  "visibility": "public",
  "name": "Region1",
  "_type": "UMLRegion",
  "_id": "AAAAAAFq3lVFLb1/ABk="
}
```



吴际 创建于 2019-06-16 22:08:01

已认证的正确解答

UMLRegion这个概念我在课上说过。它基本就是画画中的“画布”含义，一个StateMachine有一个UMLRegion，同来把第一层次的状态和迁移管理起来，就好像一幅画直接看到的内容布局。可以针对一个状态建立下一层次的状态机，即建立UMLRegion，用来管理该子状态机下的第一层状态和迁移。如此往复，可层次化建立相应的状态机。UML的这个设计就是采用了层次化思想。

👍 2

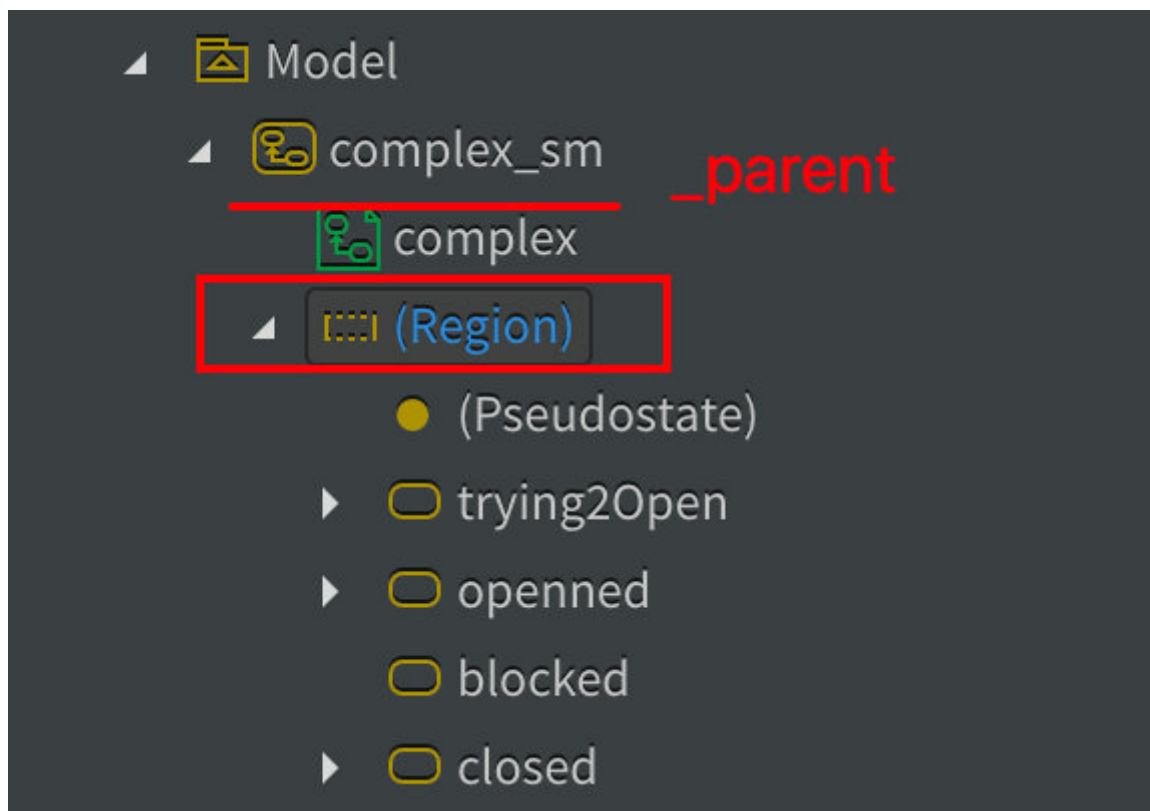
画布的概念，状态机的所有状态和迁移部署在上面，设计的目的是为了达到**层次化**的目的。其

`_parent` 为 `UmlStateMachine`。

1. `UMLStateMachine`

```
{
  "_parent": "AAAAAAFqpiMge7NXBnk=",
  "name": "complex_sm",
  "_type": "UMLStateMachine",
  "_id": "AAAAAAFqyQws9L3/cek="
}
```

声明一个状态机并给其命名。其 `_parent` 为 `UmlClass`。



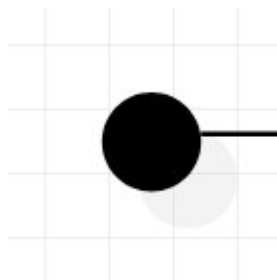
2. UmlPseudostate

```

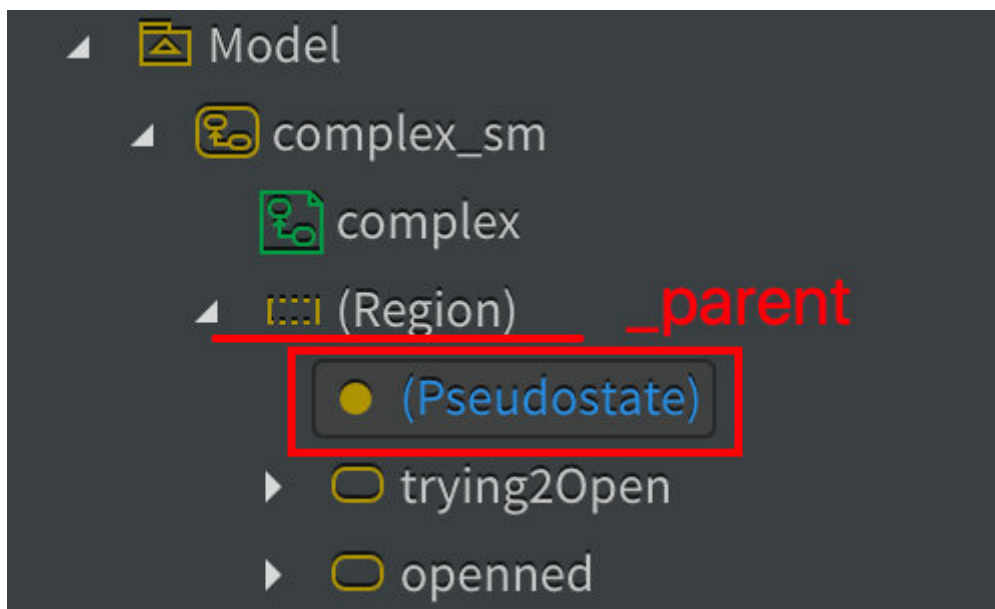
{
  "_parent": "AAAAAAfQyQws9b4A8Bk=",
  "visibility": "public",
  "name": null,
  "_type": "UMLPseudostate",
  "_id": "AAAAAAfQyeEMPTDVjII="
}

```

状态机的起始状态，用黑色实心圆点表示：



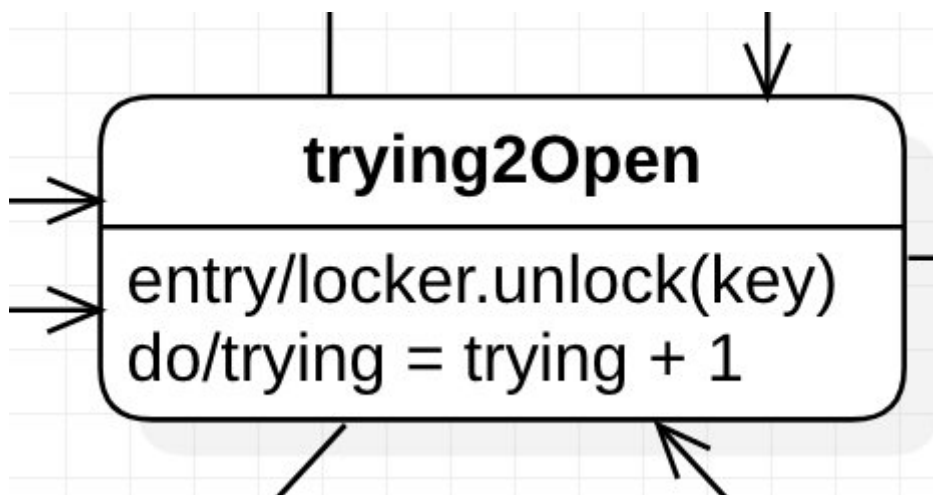
起始状态只有出度没有入度。其 `_parent` 为 `UmlRegion`。



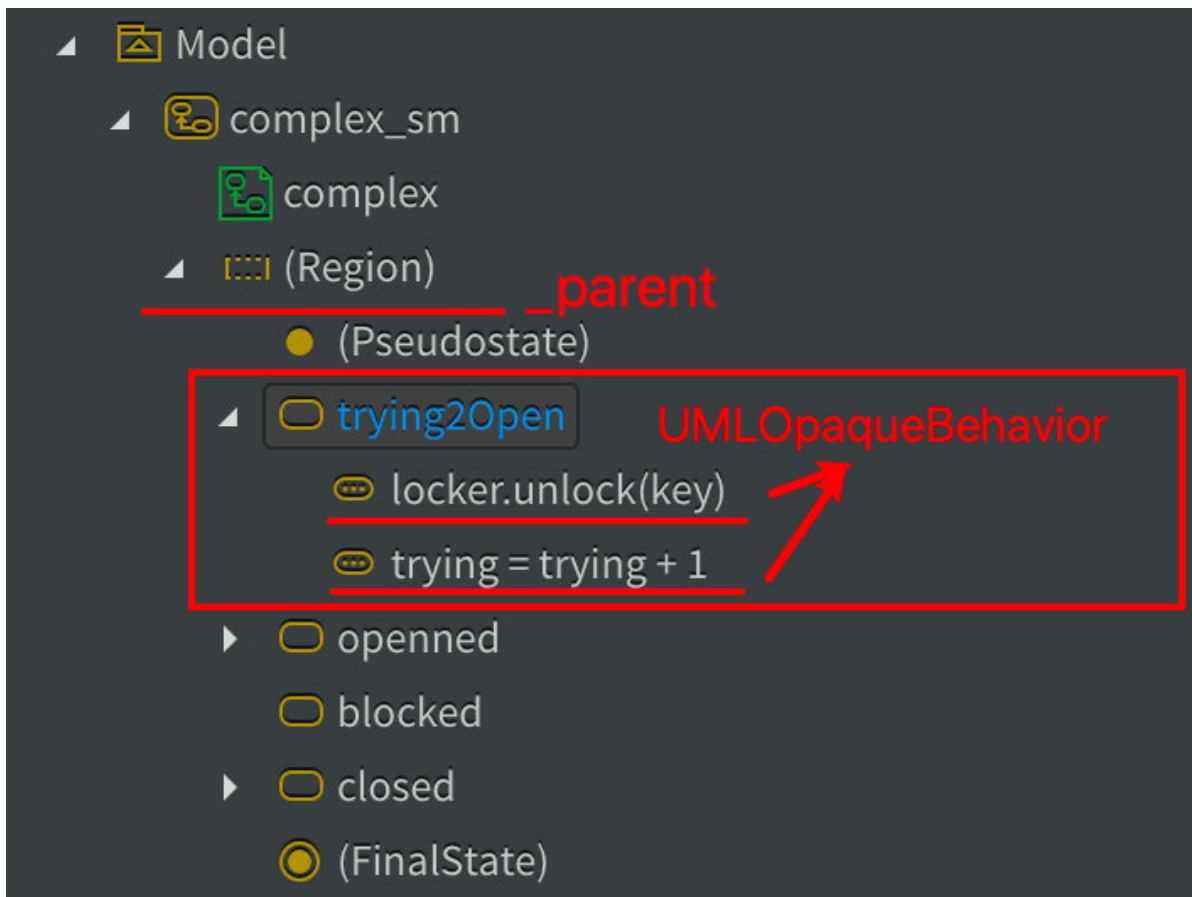
3. UmlState

```
{
  "_parent": "AAAAAAFqyQws9b4A8Bk=",
  "visibility": "public",
  "name": "trying2Open",
  "_type": "UMLState",
  "_id": "AAAAAAFqyeFwgDDmGrM="
}
```

状态机的中间状态，用圆角矩形表示：



其 `_parent` 为 `UmlRegion`。



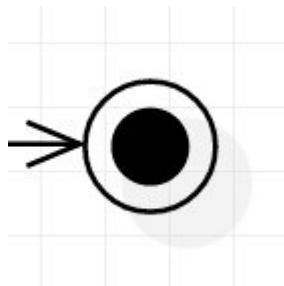
4. UmlFinalState

```

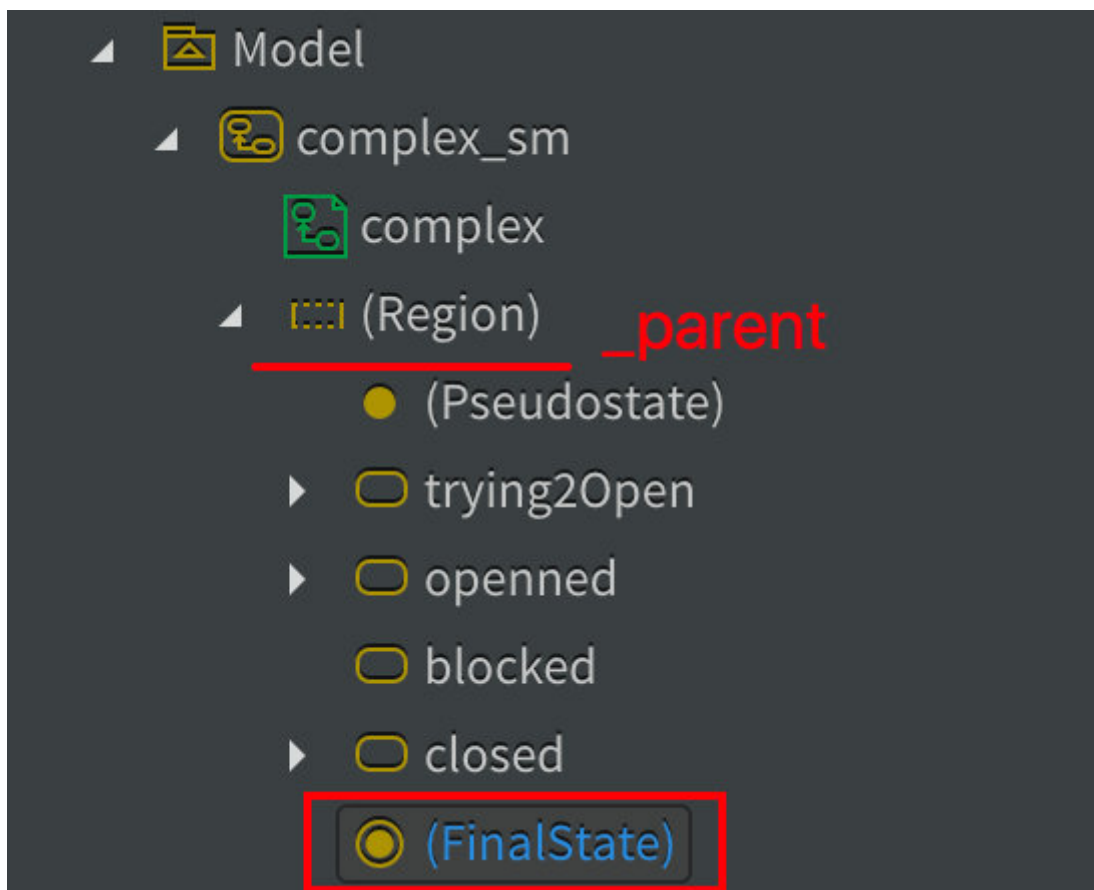
{
  "_parent": "AAAAAAFqyQws9b4A8Bk=",
  "visibility": "public",
  "name": null,
  "_type": "UMLFinalState",
  "_id": "AAAAAAFqyeKjvDGGayc="
}

```

状态机的结束状态，用带外圆的黑色实心圆点表示：



结束状态只有入度没有出度，其 `_parent` 为 `UmlRegion`。



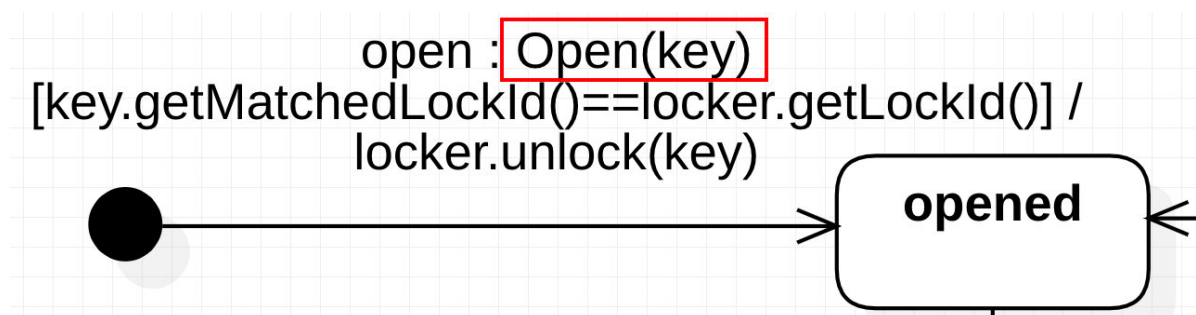
5. UmlEvent

```

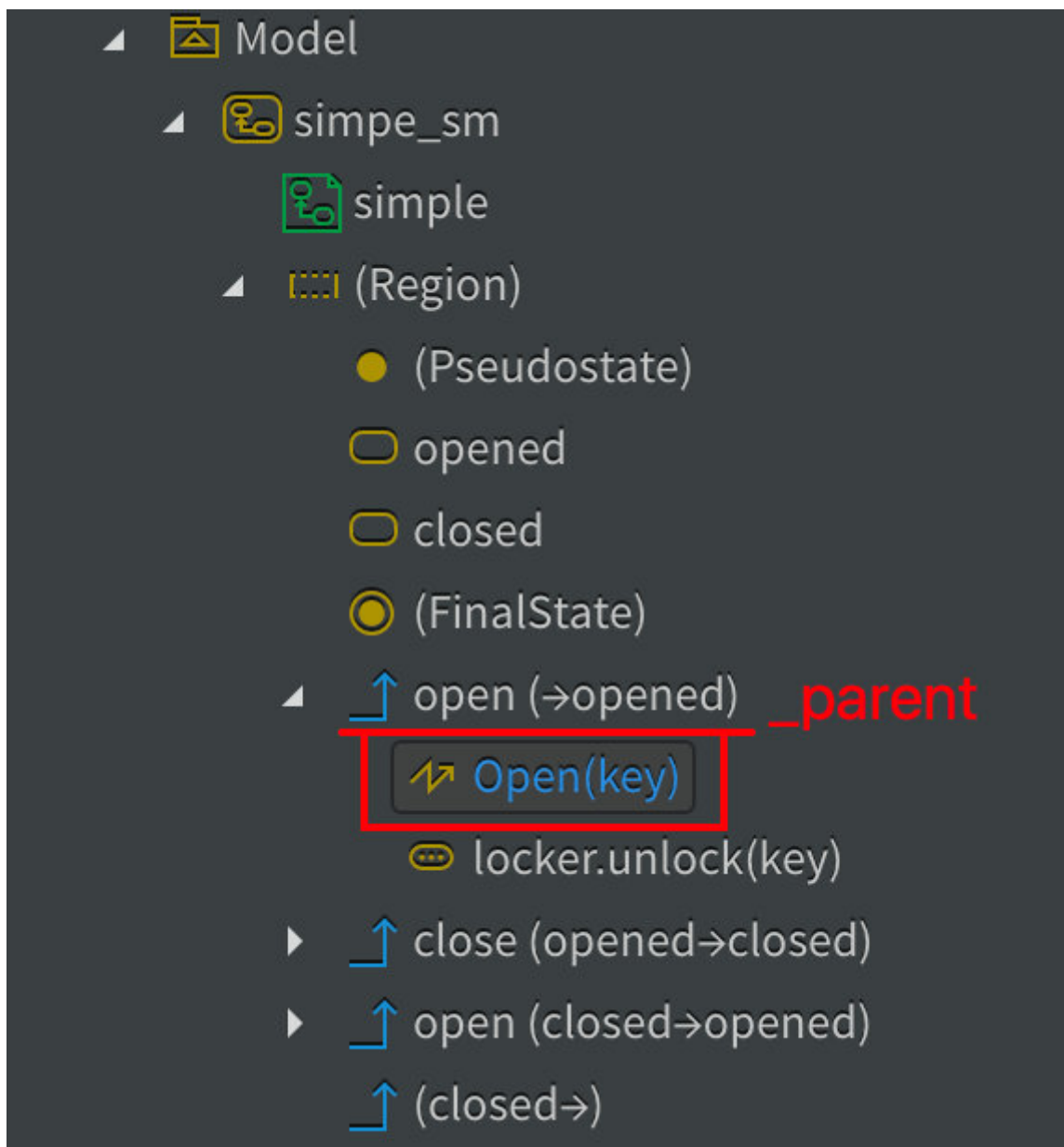
{
  "_parent": "AAAAAAFqyeLuBjGMJ9M=",
  "expression": null,
  "visibility": "public",
  "name": "Open(key)",
  "_type": "UMLEvent",
  "_id": "AAAAAAFqyea1LTirDKQ=",
  "value": null
}

```

状态机的状态转换事件标记（响应事件）：



只有当响应事件发生的时候才**有可能**进行状态转换。事件标记是**转移的诱因**，可以是一个信号，事件、条件变化（a change in some condition）和时间表达式。在我们的课程中**应该**只需要考虑事件（函数调用），所以 `UmlEvent` 通常是方法。其 `_parent` 为所在的 `UmlTransition`。



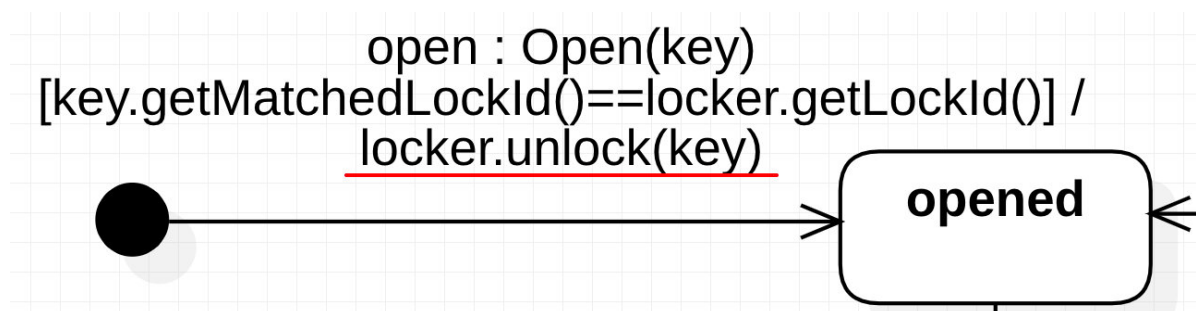
6. UmlOpaqueBehavior

```

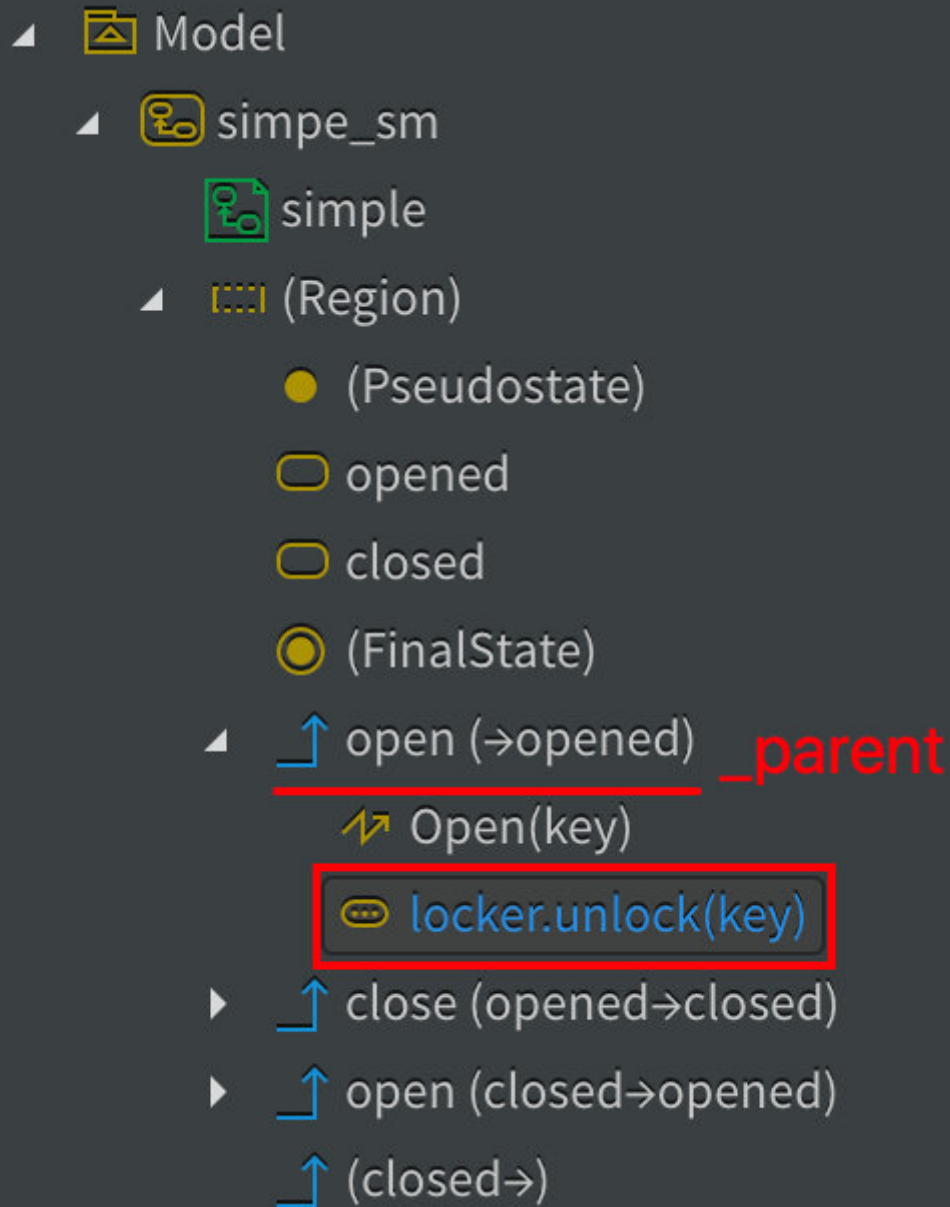
{
  "_parent": "AAAAAAFqyOY/GLngY5I=",
  "visibility": "public",
  "name": "locker.unlock(key)",
  "_type": "UMLOpaqueBehavior",
  "_id": "AAAAAAFqyPbIMrvFRTg="
}

```

状态转移的结果:



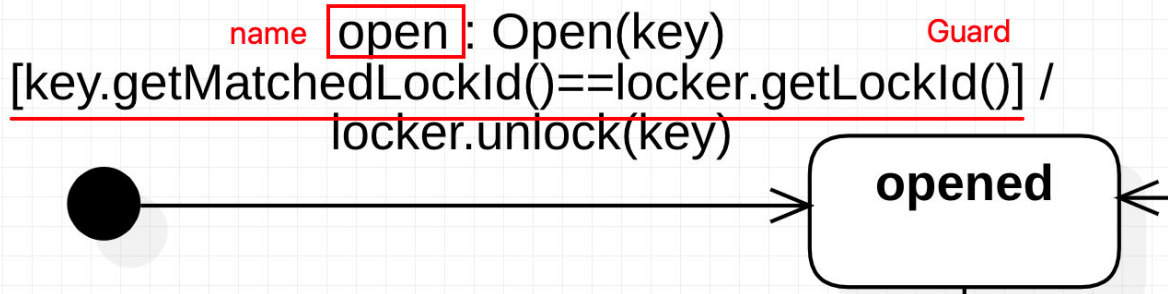
其 `_parent` 为所在的 `UmlTransition`。



7. UmlTransition

```
{
  "_parent": "AAAAAAFqyONLFLlwdXI=",
  "visibility": "public",
  "guard": "key.getMatchedLockId()==locker.getLockId()",
  "name": "open",
  "_type": "UMLTransition",
  "_id": "AAAAAAFqyOY/GLngY5I=",
  "source": "AAAAAAFqyOVx3rmCP2Y=",
  "target": "AAAAAAFqyOW7gLmTuE4="
}
```

状态机的状态转换，在转换前状态指向转换后状态的黑实线上表示：



`name` 表示该状态转换的名称，`guard` 表示该状态转换所能发生的时机（01 阈值）。其 `_parent` 为 `UmlRegion`。

`UmlTransition` 本身声明了一个实际的状态转换事件，其通常由四个部分组成：

1. 名称：如这里的 `open`，方便我们识别这里的动作是什么；
2. 事件标记：如这里的 `Open(key)`，表示状态转移的发生条件，并不是所有的事件都能在任何状态转移，如 `opened` 的状态下事件 `Open(key)` 显然无效；
3. 守护条件：如这里的 `key.getMatchedLockId()==locker.getLockId()` 表示判定是否状态转移发生的条件，显然当钥匙和门不对应的时候不能开锁；
4. 结果：如这里的 `locker.unlock(key)`，只有当事件标记发生且守护条件成立的情况下才能执行后续的结果操作。

在 `mdj` 文件中可以看到它们的结构：

```

{
  "_type": "UMLTransition",
  "_id": "AAAAAFqyOY/GLngY5I=",
  "_parent": {
    "$ref": "AAAAAFqyONLFLlwdXI="
  },
  "name": "open",
  "source": {
    "$ref": "AAAAAFqyOVx3rmCP2Y="
  },
  "target": {
    "$ref": "AAAAAFqyOW7gLmTuE4="
  },
  "guard": "key.getMatchedLockId()==locker.getLockId()",
  "triggers": [
    {
      "_type": "UMLEvent",
      "_id": "AAAAAFqyO3ytLoyj1A=",
      "_parent": {
        "$ref": "AAAAAFqyOY/GLngY5I="
      },
      "name": "Open(key)",
      "kind": "call",
      "targetOperation": {
        "$ref": "AAAAAFqpiRcY7O7pzM="
      }
    }
  ],
  "effects": [
    {
      "_type": "UMLOpaqueBehavior",
      "_id": "AAAAAFqyPbIMrvFRtg=",
      "_parent": {

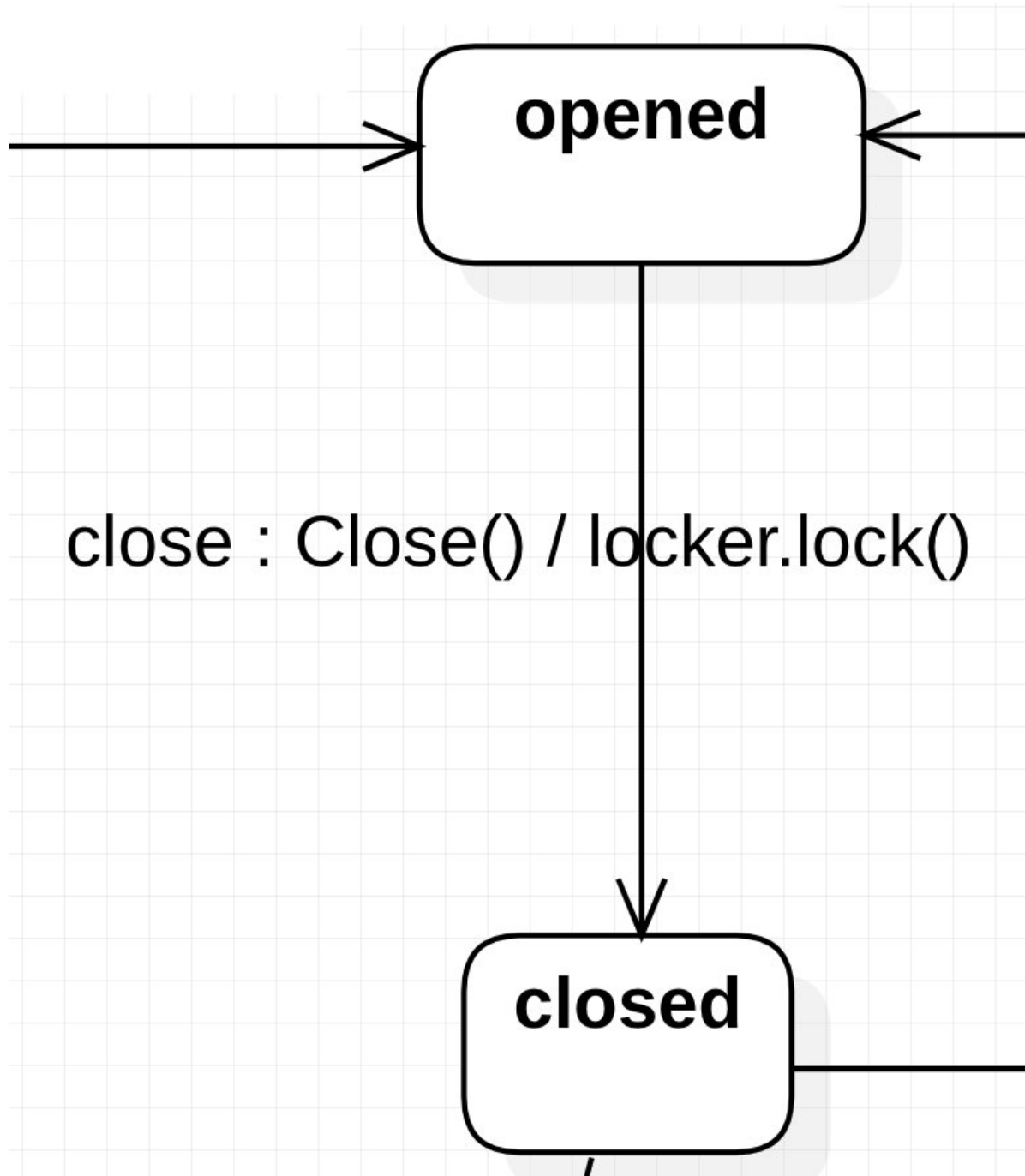
```

```

        "$ref": "AAAAAAFqyOY/GLngY5I="
      },
      "name": "locker.unlock(key)"
    }
  ]
}

```

当然上面四个部分不是必要的，如：



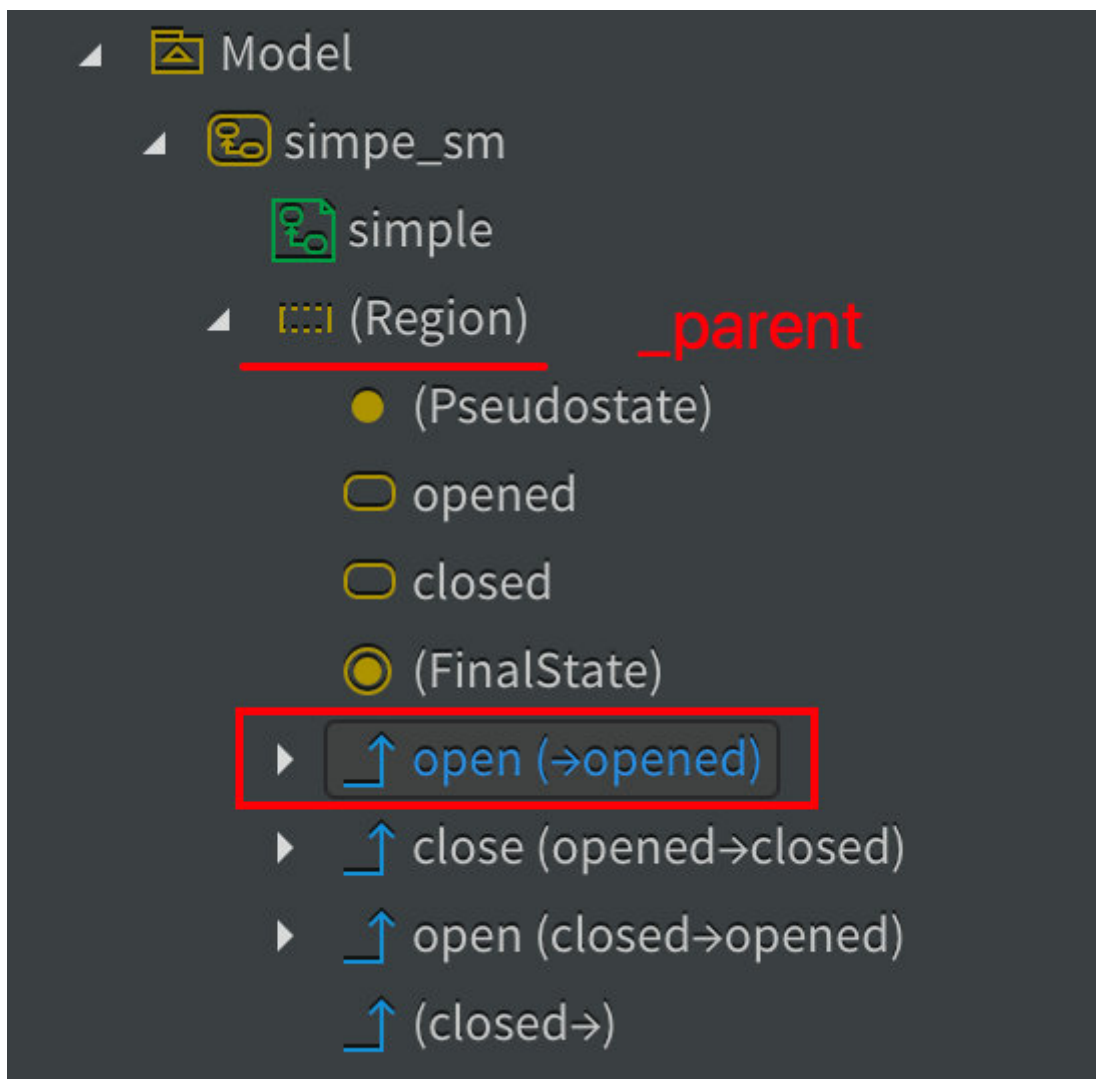
可以没有守护条件 `guard`，即默认为真，只要在 `opened` 状态下有事件 `close()` 发生就会执行 `locker.lock()` 的结果事件，其 mdj 结构为：

```

{
  "_type": "UMLTransition",
  "_id": "AAAAAAFqyObAnrny29A=",
  "_parent": {
    "$ref": "AAAAAAFqyONLFLlwdXI="
  },
}

```

```
"name": "close",
"source": {
  "$ref": "AAAAAAFqyOW7gLmTuE4="
},
"target": {
  "$ref": "AAAAAAFqyOXm0Lm5/v8="
},
"triggers": [
  {
    "_type": "UMLEvent",
    "_id": "AAAAAAFqyP2QWL3j0ls=",
    "_parent": {
      "$ref": "AAAAAAFqyObAnrny29A="
    },
    "name": "close()"
  }
],
"effects": [
  {
    "_type": "UMLOpaqueBehavior",
    "_id": "AAAAAAFqyR4HIb4itVs=",
    "_parent": {
      "$ref": "AAAAAAFqyObAnrny29A="
    },
    "name": "locker.lock()"
  }
]
}
```



结语

以上就是顺序图和状态图主要的 `UmlElement` 的子类，可能乍一看会比较头大，大家可以通过训练题目中的 UML 图熟悉，相信大家很快就可以熟悉~

有些同学可能看顺序图会有点晕，而且很难与类图联系起来，[这里推荐一个博客](#)，它利用三国演义中的赤壁之战形象地说明了顺序图，非常清晰。

如果你能坚持看到这里，再送上一句吴老师的话：

一定要打开 json 文件（mdj 文件），浏览其树形结构，对照模型图观察每个元素的内容和其所管理的下层数据对象。

这一定会让你在本单元的作业中事半功倍。

顺序图和状态图参考资料

1. [UML 建模之状态图 \(Statechart Diagram\)](#)
2. [\[推荐的博客\] UML 系列——时序图 \(顺序图\) sequence diagram](#)
3. [Messages in UML diagrams](#)
4. [UML Sequence Diagrams](#)