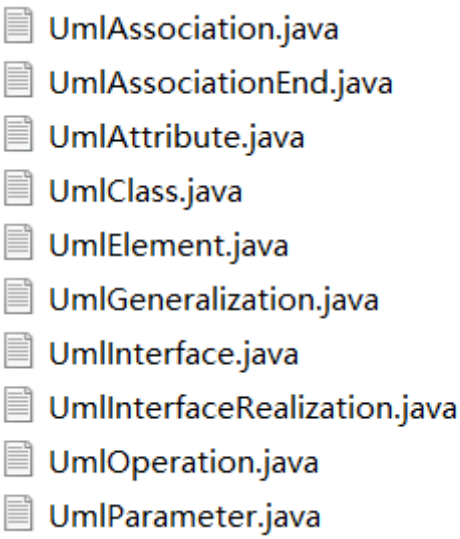
UML第二次作业中的各种UmlElement子类介绍

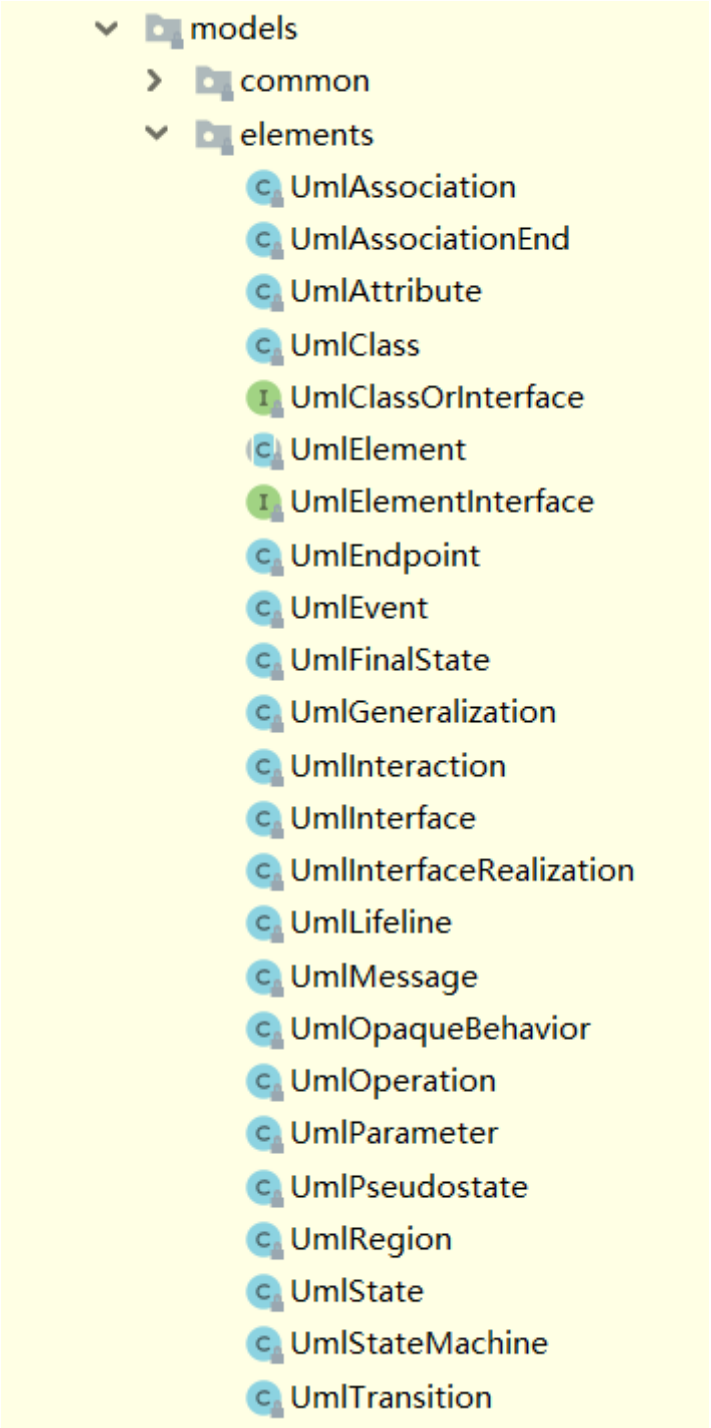
- UML第二次作业中的各种UmlElement子类介绍
  - 多了这么多类?
  - 新增的类
  - 按照逻辑顺序整理新增的类
    - 有关状态图的查询
      - 0. UmlRegion
      - 1. UmlStateMachine
      - 2. UmlPseudostate
      - 3. UmlState
      - 4. UmlFinalState
      - 5. UmlEvent
      - 6. UmlOpaqueBehavior
      - 7. UmlTransition
    - 有关顺序图的查询
      - 0. UmlInteraction
      - 1. UmlLifeline
      - 2. UmlMessage
      - 3. UmlEndPoint

多了这么多类?

上一次的UmlElement子类：



这一次的UmlElement子类：



新增的类

- UmlEndPoint /
- UmlEvent /
- UmlFinalState /
- UmlInteraction
- UmlLifeline /
- UmlMessage /

- UmlOpaqueBehavior /
- UmlPseudostate /
- UmlRegion /
- UmlState /
- UmlStateMachine /
- UmlTransition /

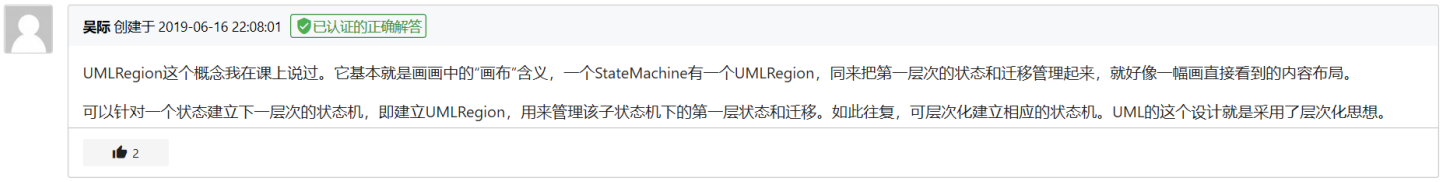
按照逻辑顺序整理新增的类

如无特殊说明，代码块内的为对应的子类的样例

有关状态图的查询

0. UmlRegion

```
{   "_parent":"AAAAAAFqyeHHXDE0fXE=",
    "visibility":"public",
    "name":"Region1",
    "_type":"UMLRegion",
    "_id":"AAAAAAFq3lVFLbl\/ABk="}
```



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

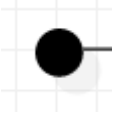
```
{   "_parent":"AAAAAAFqpiMge7NXBnk=",
    "name":"complex_sm",
    "_type":"UMLStateMachine",
    "_id":"AAAAAAFqyQWs9L3\/cek="}
```

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：



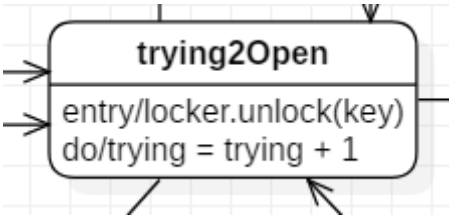
起始状态只有出度没有入度

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：

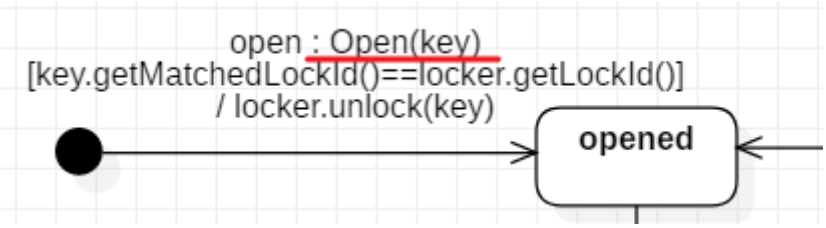
结束状态只有入度没有出度

其_parent为 UmlRegion

5. UmlEvent


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


状态机的状态转换事件标记(响应事件):

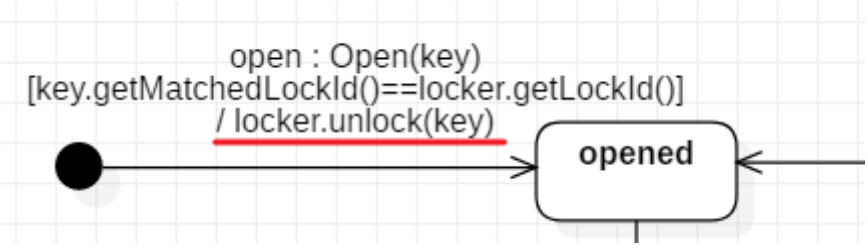


只有当响应事件发生的时候才**有可能**进行状态转换
事件标记是**转移的诱因**，可以是一个信号，事件、条件变化（a change in some condition）和时间表达式。
在我们的课程中**应该**只需要考虑事件(函数调用)所以 `UmlEvent` 通常是方法。
其_parent为 所在的UmlTransition

6. UmlOpaqueBehavior


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


状态转移的结果:

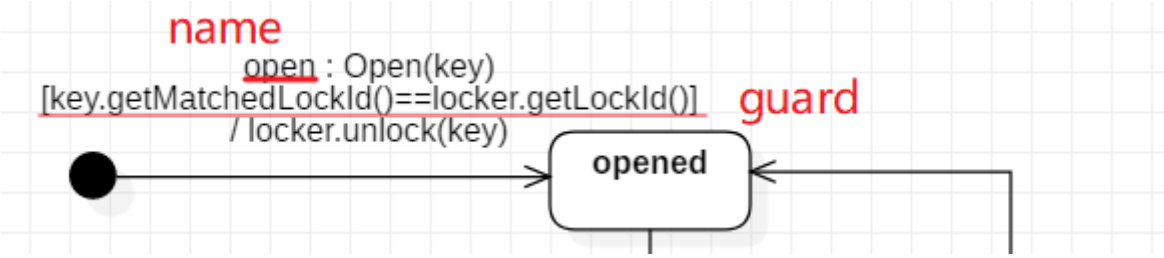


~~`Opaque` 的意思是含混的、不透明的，难道这里想要表示该后续操作不可见？等一个答案~~
其_parent为 所在的UmlTransition

7. UmlTransition

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

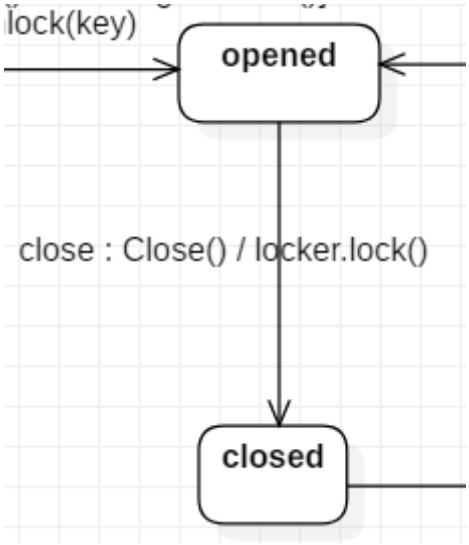

状态机的状态转换，在转换前状态指向转换后状态的黑实线上表示：



`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)
`UmlTransition` 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. 警界条件：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警戒条件成立的情况下才能执行后续的结果操作

当然上面四个部分不是必要的，如：



可以没有警戒条件，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_parent为 UmlRegion

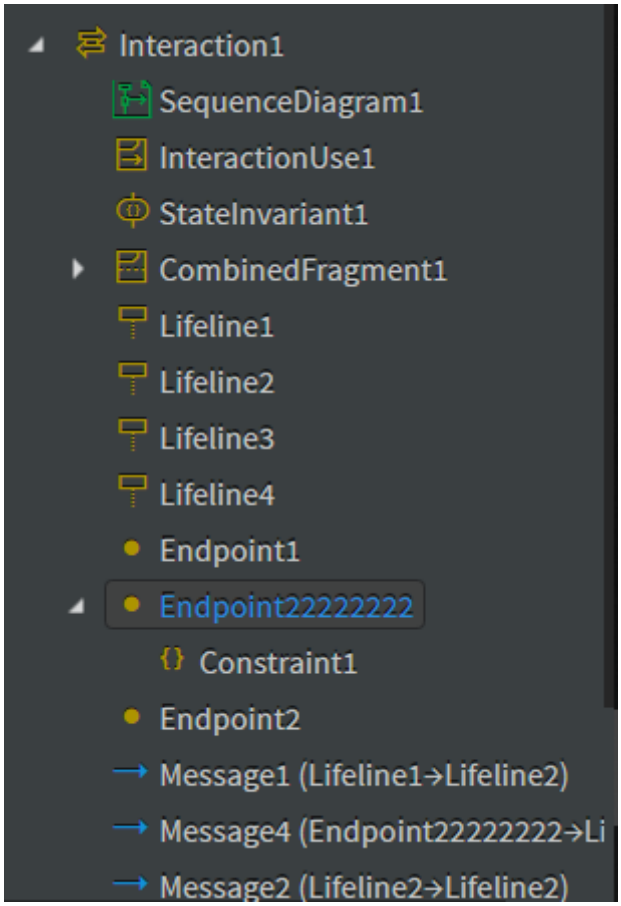
有关顺序图的查询

如无特殊说明，代码块内的为对应的子类的样例

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，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：

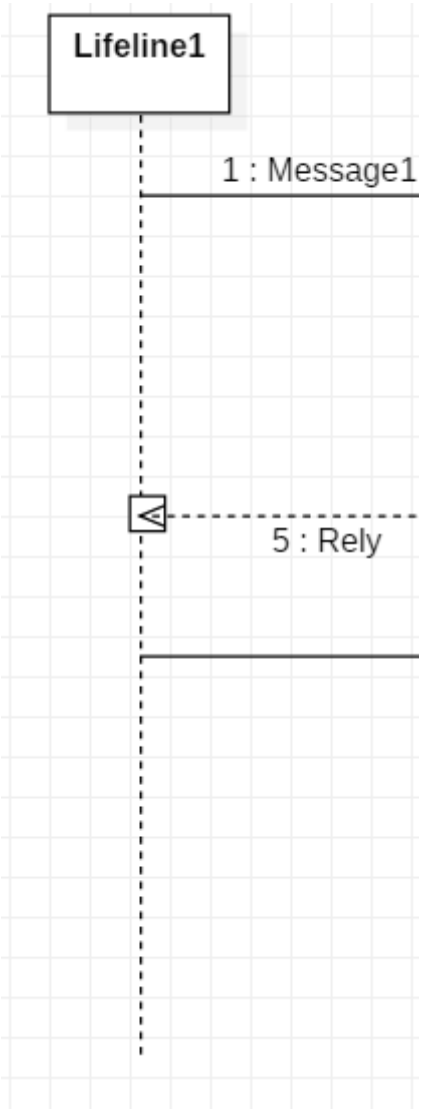


其_parent为 UmlCollaboration(我们无需考虑，所以没有太大意义)

1. UmlLifeline

```
{   "_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：



在类图内我们只讨论类而不讨论由类所创造的具体对象实例，但是在顺序图里面我们描述的是**系统的动态行为**，既然是动态行为就一般具有一个生成和消亡的周期性动作，这里生命线就是来描述对象什么时候被生成(声明)。

`name` 的命名方法一般分为三种：

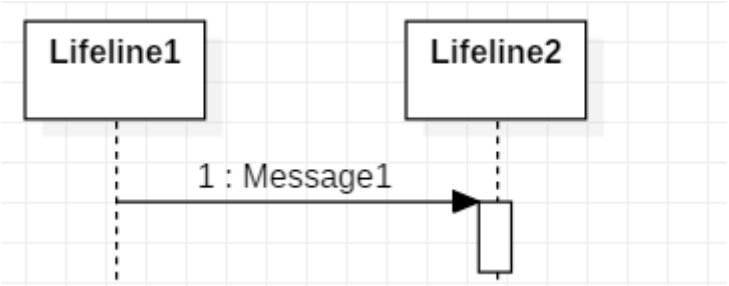
1. 对象名
2. 类名：对象名

3. ：类名(匿名对象)
   其**_parent为 UmlInteraction**

2. UmlMessage

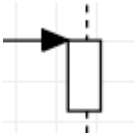
```
{   "messageSort":"synchCall",
    "_parent":"AAAAAAFrZNSJhnMQrSk=",
    "visibility":"public",
    "name":"Message1",
    "_type":"UMLMessage",
    "_id":"AAAAAAFrZNTLwXN7x1w=",
    "source":"AAAAAAFrZNSX43Mf80k=",
    "target":"AAAAAAFrZNSojXM+cCs="}
```

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件。消息从源对象指向目标对象。消息一旦发送便**将控制从源对象转移到目标对象。**

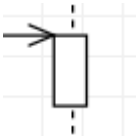


消息也分成两类：

1. 同步消息，用黑三角箭头搭配黑实线表示：

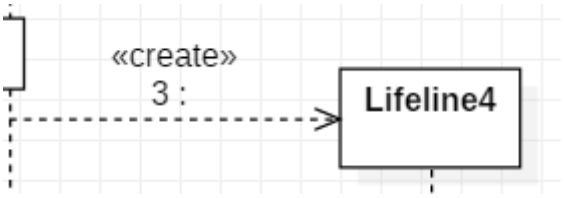
   同步的意义：消息的发送者把进程控制传递给消息的接收者，然后**暂停活动**，**等待**消息接收者的回应消息。
2. 异步消息，用两条小线的开箭头和黑色实线表示：

   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，用黑三角箭头搭配黑色虚线表示：

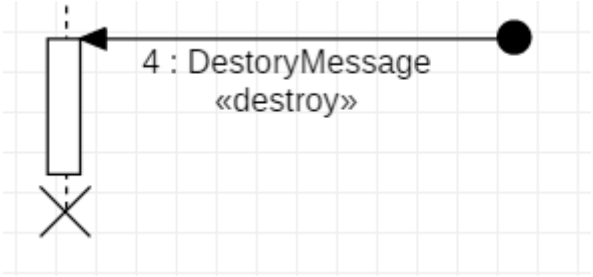
返回消息和同步消息结合使用，因为异步消息不进行等待所以不需要知道返回值

4. 创建消息，用开三角箭头搭配黑实线表示，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

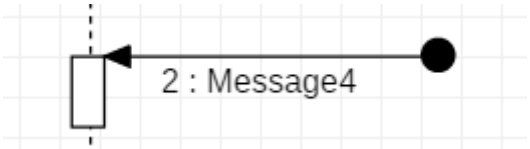
5. 摧毁消息，用黑三角箭头搭配黑实线表示，其下面特别注明 <<destroy>>



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 & Found Message ~~不知道怎么翻译~~

可以理解为野路子的消息，他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：



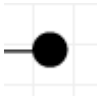
注：在某些地方还细分了一个简单消息，不区分同步和异步，starUml中省去了。

其_parent为 UmlInteraction

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

以上就是所有第二次作业中出现的新的UmlElement的子类，如果能帮助到你非常高兴~

有些同学可能看时序图会有点晕，而且很难与类图联系起来， **这里推荐一个博客**，他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。

Reference:

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**