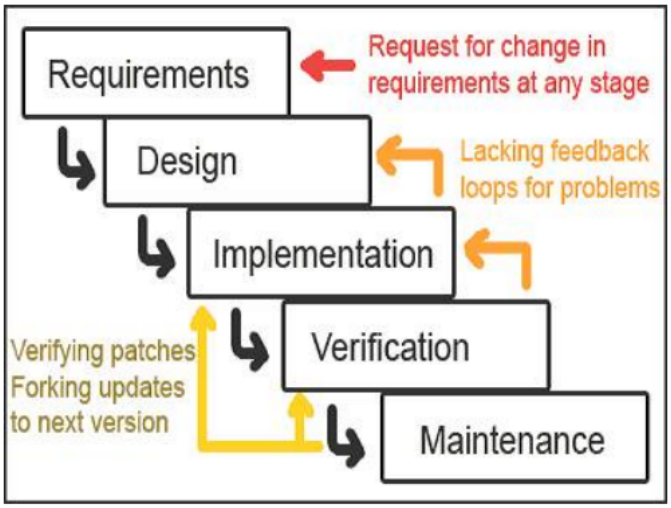
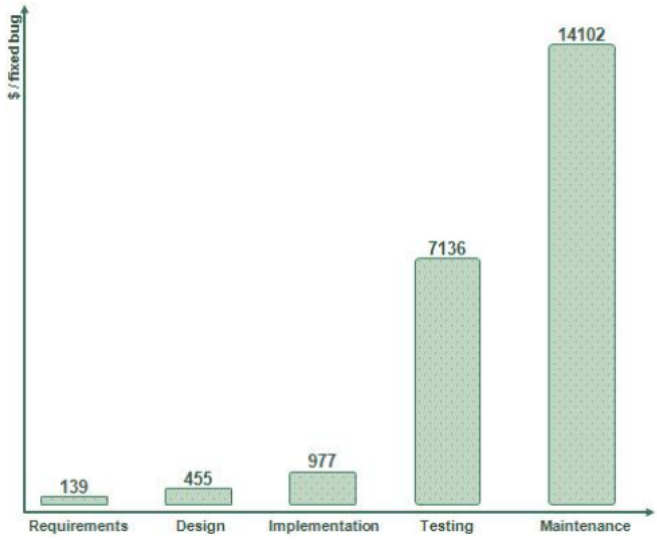
Lecture2 UML

1. 需求工程 Requirements Engineering

Waterfall Model



为什么需求很重要



- 正式的定义：为什么在实现重要需求之前要正式定义需求?
  - 项目的大部分成功或失败在建设（实施）开始之前就已经确定了
  - 要打好基础，做好规划
- 需求工程的总体目标是降低风险
  - 在实现之前尽早发现问题和不一致
  - 虽然存在很多形式主义，但并不是真正的“科学”
    - 模型检验、定理证明、知识表示

2. UML 的介绍

UML 概念

统一建模语言 Unified Modeling Language

- 巨大的语言
- 许多松散相关的风格在一个屋檐下
- 为软件设计和实现提供了**通用的、简单的、图形的表示**
- 允许**开发人员、架构师和用户**讨论软件的工作原理
- http://www.omg.org

建模 Modeling

- 在高抽象级别上描述系统
    - 系统的模型
    - 用于需求和规范
- 随着时间的推移，存在着许多符号
    - 状态机 State machines
    - 实体关系图 Entity-relationship diagrams （E-R diagram）
    - 数据流图 Dataflow diagrams

建模指导思想

- UML 中的几乎所有内容都是可选的
- 模型很少是完整的
- UML 是“开放的解释”
- UML 被设计为可扩展的

UML 中的静态建模 Static Modeling

静态建模捕获系统中**固定的代码级别的**关系

- **类图 Class diagrams**
- 包图 Package diagrams
- 构件图 Component diagrams
- 组合结构图 Composite structure diagrams
- 部署图 Deployment diagrams

UML 中的行为建模 Behavioral Modeling

行为图用于捕获系统的**动态执行**

- **用例图 Use case diagrams**
- 序列图 Sequence diagrams
- 协作图 Collaboration diagrams
- 状态图 State diagrams
- 活动图 Activity diagrams

UML 中的交互建模 Interaction Modeling

注重元素之间的沟通

- **序列图 Sequence diagrams**
- 交流图 Communication diagrams
- 交互概览图 Interaction overview diagrams
- 计时图 Timing diagrams

3. 用例图 Use Case Diagram

https://www.bilibili.com/video/BV1at411Z7Ni?from=search&seid=7082535347371585038&spm_id_from=333.337.0.0

用例图介绍

用例图从用户的角度捕获系统的需求

- 用例图处于比其他 UML 元素更高的抽象级别

每一种用户都有一个或多个用例

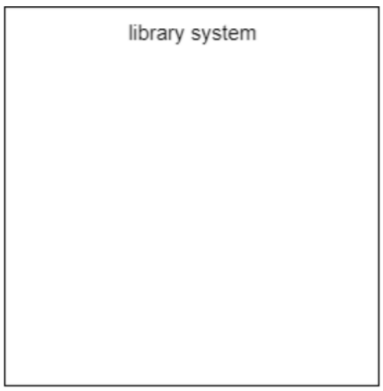
- 对于任何合理的系统来说，有许多许多种用例是很常见的

用例图的术语

- **行动者 actor**：用例中的 actor 是使用系统或与系统交互的外部代理，actor 可以是用户或角色，如人或环境的外部系统特征，如时间或温度变化

- **用例 use case**：指的是系统必须提供（给用户）的特定功能部分，use case 是相关场景的集合，包括正常场景和可选场景。为了完成业务任务，可以将场景视为与行为相关的步骤序列，可以是自动化的，也可以是手动的

- **系统边界 system boundary**：系统边界是指不同系统之间的边界，我们可以用它来区分系统内的元素和系统外的元素，当 actor 引用与系统交互的外部元素时，use case 总是充当系统边界内的功能

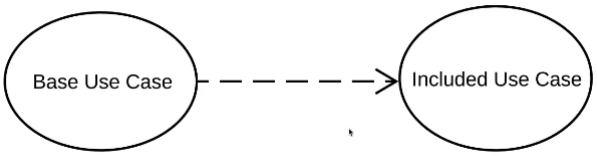


用例的关系

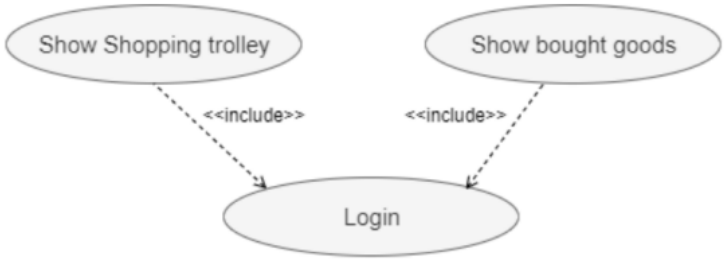
当描述一个复杂的系统时，它的用例模型可能变得非常复杂，并且可能包含冗余

下面使用三种类型的关系来降低模型的复杂性

包含 Include

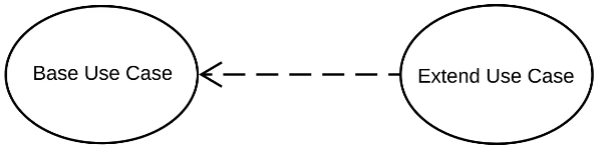


Include 是两个用例之间的一种定向关系，暗示被包含用例的行为被插入被包含用例的行为中

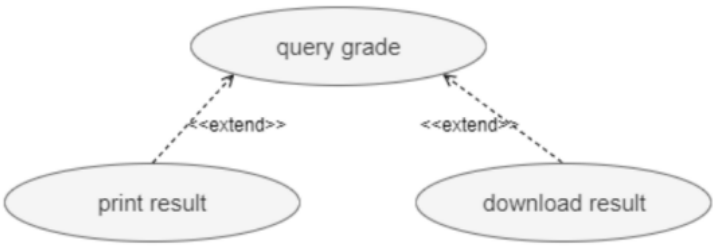


- Show shopping trolley 和 Show bought goods 需要包含 Login
- 包含关系被包含的用例一定会被执行
- Use Case B 不能离开 Use Case A

扩展 Extend



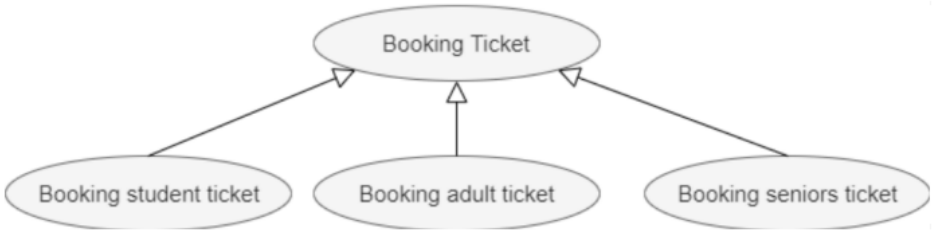
这种关系指定了一个用例的行为可以由另一个用例的行为扩展



- query grade 可以选择 print result 或者 download result，但是并不意味着一定要执行这些用例
- 扩展关系中被扩展的用例不一定被执行
- 表示用例的特殊变化
- Use Case B 可以离开 Use Case A
- B 可以是 A 的某个步骤，但是也可以不执行 B

泛化 Generalization

泛化是更一般的分类器和更具体的分类器之间的分类关系，特定分类器的每个实例也是通用分类器的一个间接实例，因此，特定的分类器继承了更一般的分类器的特征



- Booking Ticket 具体来说有三种实现用例，Booking student ticket, Booking adult ticket 和 Booking seniors ticket

示例 —— 自动火车

考虑一个无人驾驶的交通工具

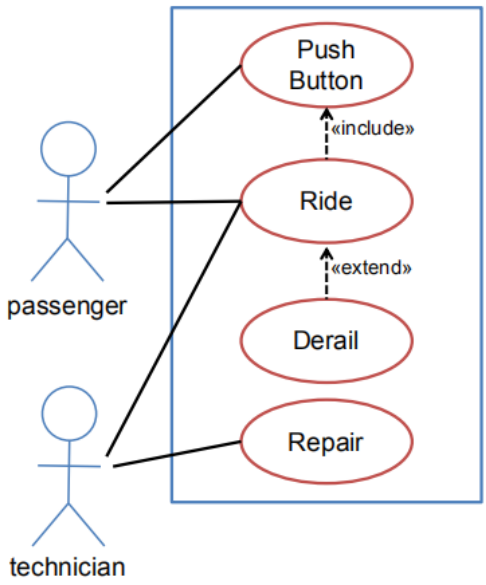
火车

- 沿着圆形轨道移动
- 依次访问两个站点（A和B）
- 每个站都有一个“请求”按钮
    - 一位在等车的乘客请求火车在该站停下来
- 每列火车都有两个“请求”按钮
    - 一位上了车的乘客请求火车在该站停下来

用例图设计

名词	解释
用例图名称	正常的火车
actor	乘客
入口条件（进入系统）	乘客在车站
出口条件（退出系统）	乘客离开车站
事件流	乘客到达并按请求按钮 火车到达并停在月台 车门打开 乘客 P 进入火车 车门关闭 乘客 P 在终点站按下按钮 ... 车门打开 乘客 P 离开火车

用例图关系



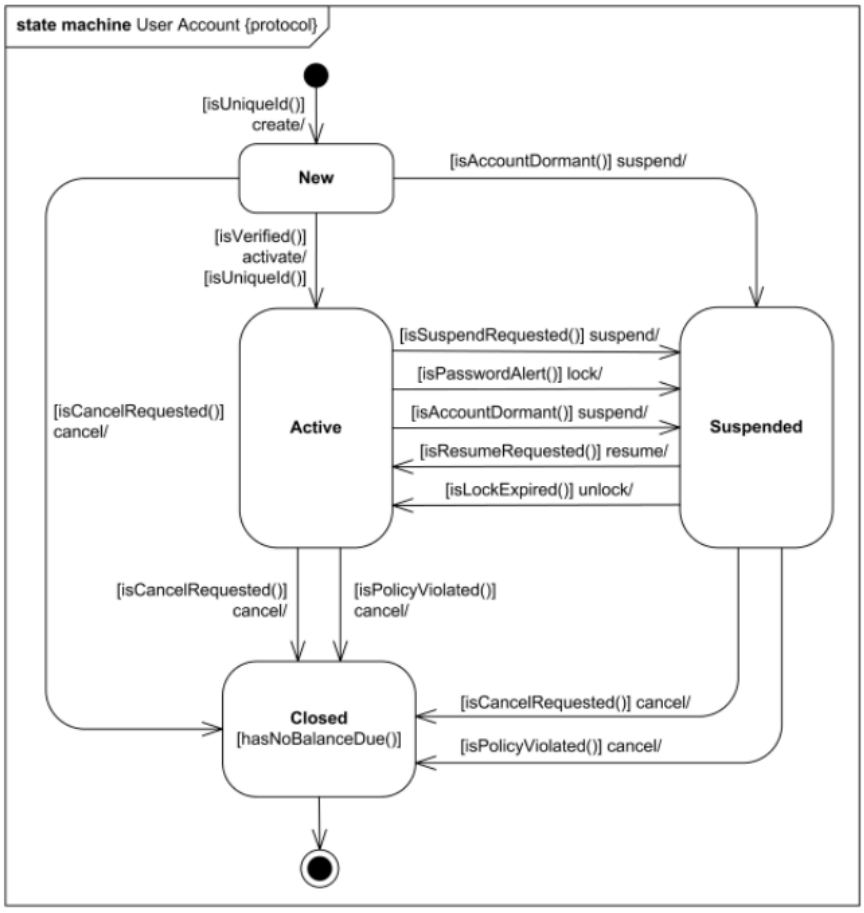
- **包含 Inclusion**
  - 实现某一个 Use Case 是否可以脱离另外一个?
  - 按按钮包含了与行驶的功能
- **扩展 Extension**
  - 脱轨是一段特殊的行驶
- **泛化 generalization**
  - 火车按钮和车站按钮是按钮的专门产品

3. 状态图 State chart Diagram

状态图介绍

- 另一种定义行为需求的方法
  - 建立在**状态机**上
- 显示实体在其生命周期中的各个阶段
- 可以用来显示方法、对象、组件等的状态转换
  - **行为状态机**显示系统中特定元素的行为
  - **协议状态机**显示协议的行为

状态图的组件



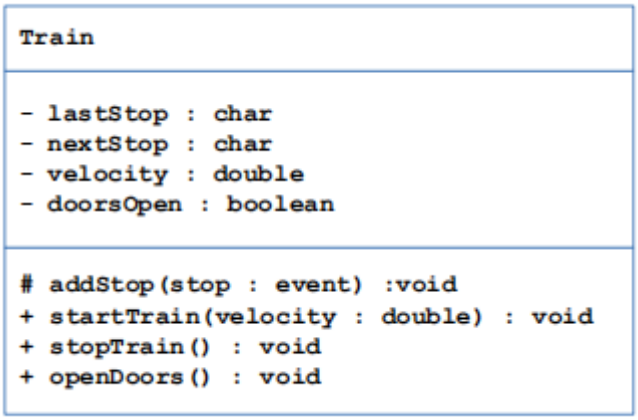
- **状态 state**：表示模型实体的一种状态，在这种状态下，某些动作被执行，某些刺激被接收，或者某些条件在系统的其他地方被满足
  - 初始状态用一个黑色实心圆表示
- **行为 action**：一个原子执行（原子的意思是它不间断地完成）
- **活动 activity**：是可能运行很长时间的更复杂的行为集合
- **转换 transition**：两个状态之间用从一个状态到另一个状态的弧线表示
- 转换可以有触发器、保护条件和行为
- 可以用创建实体的事件或动作进行标记

4. 类图 Class Diagram

类图介绍

- 系统组件之间的静态关系进行建模
  - 描述类（在OO的意义上）
- 一个 UML 模型可以有许多类图
- 类代表系统中的概念
- 一个类表示运行时系统中的一个或多个对象
  - 类的多重性由组件右上角的一个数字指定
    - 通常省略并假设大于 1
    - 指定多重性为 1 表示类应该是**单例**的

类图的组件



- 每个盒子是一个类

可见性

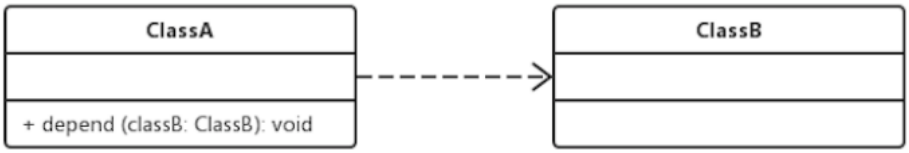
- +：public
- #：protected
- -：private
- ~: package

参数列表

- 变量名
- 数据类型

类图的关系

依赖 Dependency

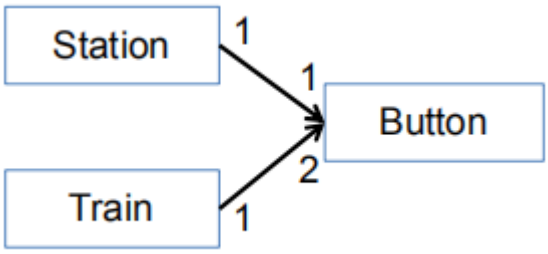


- class A uses class B
- class A 的函数中使用了 class B 作为参数
- 依赖是最弱的关系

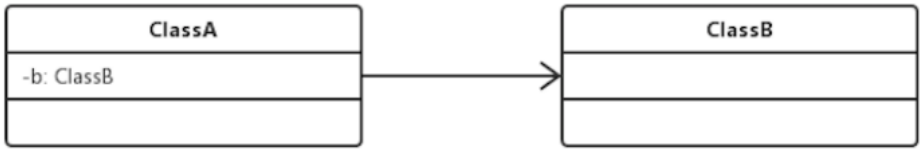
```
public class ClassA {
public void depend(ClassB classB){}
}
public class ClassB {
}
```

```
public class Train {
    ...
    protected void addStop(ButtonPressedEvent stop){
        // update nextStop
        ...
    }
    ...
}
```

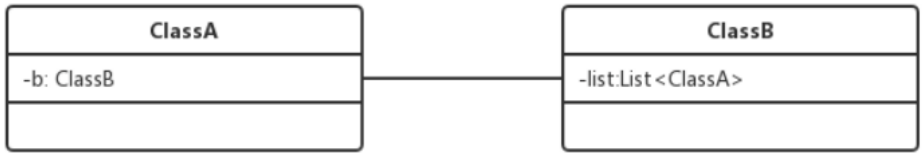
关联 Association



- **单向关联 Unidirectional Associations**
  - classA 在它的属性中至少使用了一个 classB 的变量，但是 classB 中不包含任何 classA 的属性变量

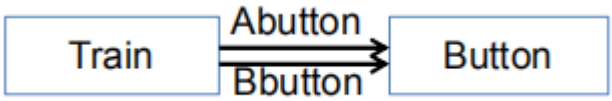


- **双向关联 Bidirectional Associations**
  - classA 和 classB 的属性字段中都至少包括对方的一个变量

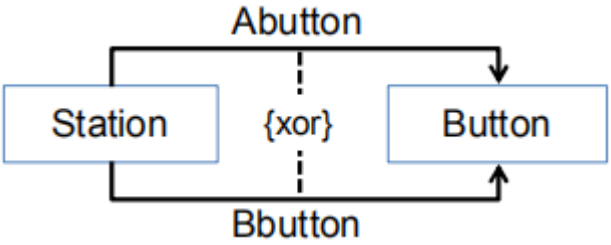


- 表明一种更牢固的关系

    - class A 离开了 class B 仍然可以在别的地方存在
- 使用数字标签来表示多样性
    - 使用 * 表示任意基数



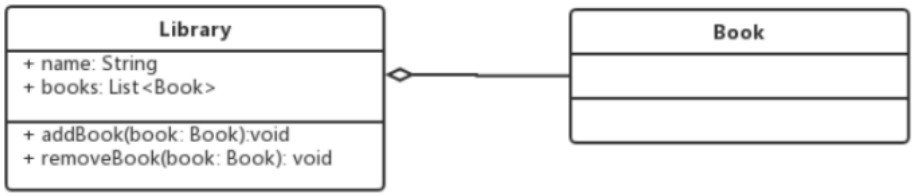
- 也还可以显式地命名关联

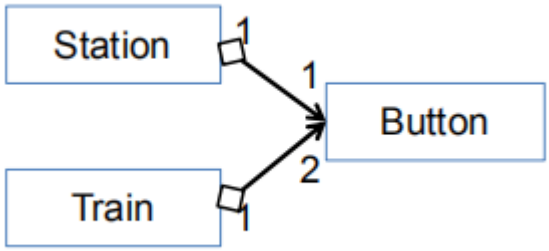


- 让它们有条件

```
class City{
    private List<Tree> trees;
    private List<Car> cars;
}
```

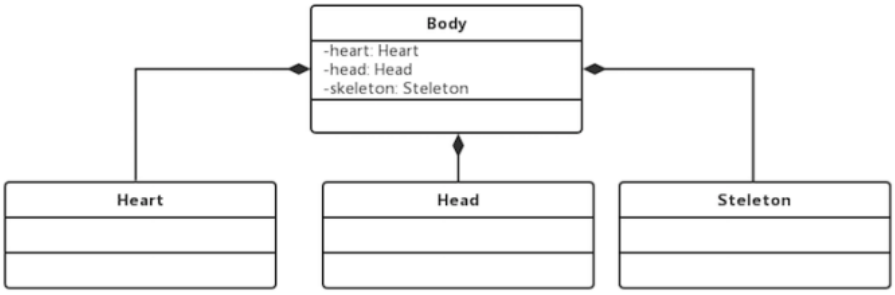
聚合 Aggregation

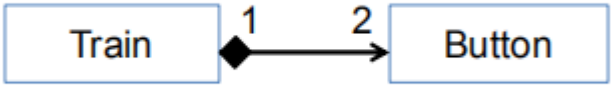




- class A has a class B
- classA 的属性至少包含 classB 的一个变量
- 经常描述 has-A 的关系，classA has a classB, library has-A book
- 当 classA 不存在了，classB 仍然可以存在
- 表示强烈的关联

组成 Composition





- class A is made up of class B
- classA 的属性至少包含 classB 的一个变量
- 经常描述 part of 的关系，classB is part of classA
- 当 classA 不存在了，classB 不再存在，如果身体不存在，心脏也不会存在
- 表示最强烈的关联
- 可以表示内部类的关系

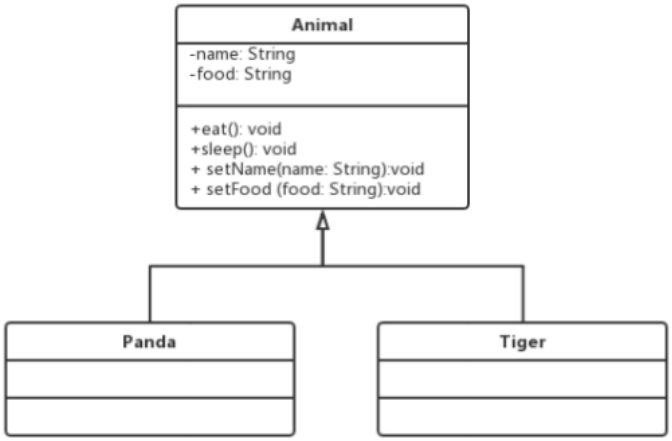
```
class Person {
    private Heart heart;
    private List<Hand> hands;
}
```

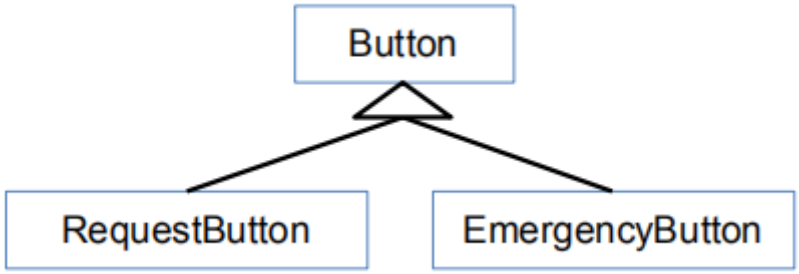
Aggregation vs. Composition

```
public class WebServer{
    private HttpListener listener;
    private RequestProcessor processor;
    // 从外部传入参数绑定 —— Aggregation
    public WebServer(HttpListener l, RequestProcessor p) {
        this.listener = l;
        this.processor = p;
    }
}
```

```java
public class WebServer{
    private HttpListener listener;
    private RequestProcessor processor;
    // 内部直接实例化 —— Composition
    public WebServer() {
        this.listener = new HttpListener(80);
        this.processor = new RequestProcessor("/www/root");
    }
}
```

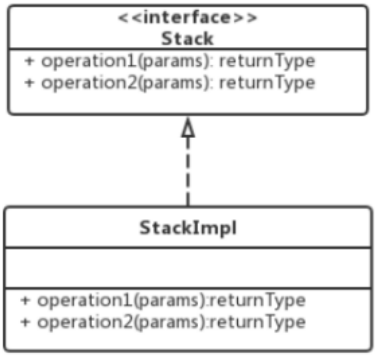
泛化 Generalization

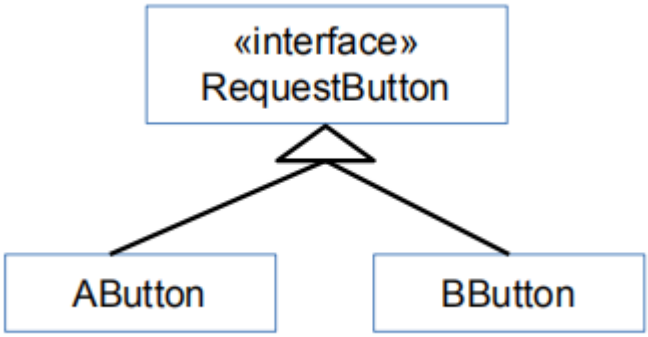




- class B is a class A
- 泛化用于显示继承
- 子类 B 和超类 A 有一个关系
- 这是 Java 中的 extends 关键字

实现 Realization



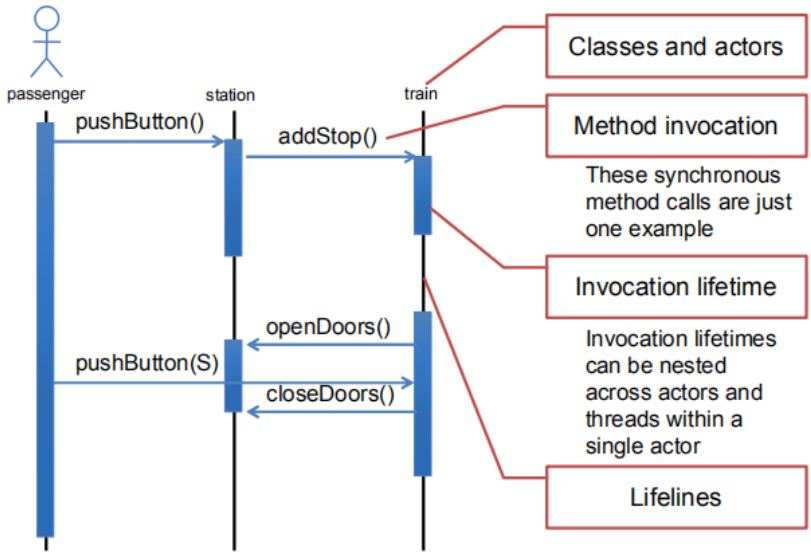


- class A implements interface B
- 实现用于显示子类型

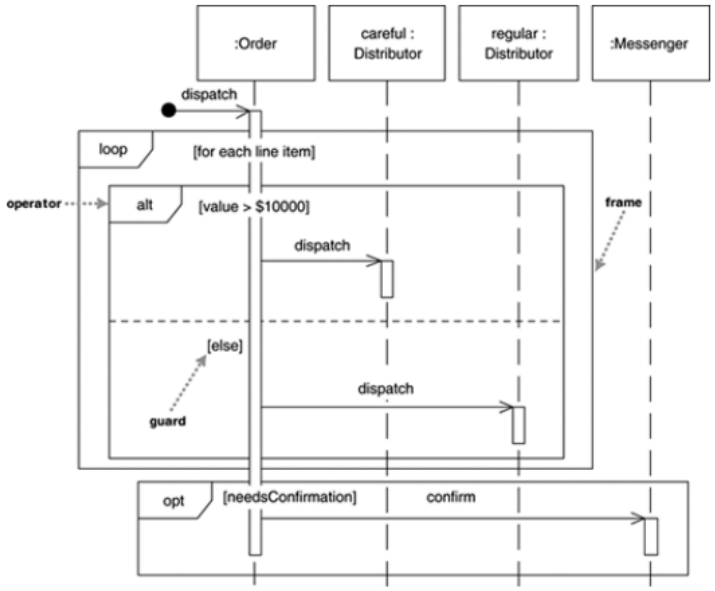
5. 序列图 Sequence Diagram

序列图介绍

- 序列图显示了对象之间基于时间的消息视图
- 把它想象成一张表
    - 列是 class 或 actor
    - 行是时间步长
    - 条目显示控制 / 数据流（例如，方法调用，状态中的重要更改）
- 每个对象都有一个沿图垂直向下运行的虚线**生命线**
    - 在序列所覆盖的时间内销毁的对象通常不会在终止对象的消息之外绘制

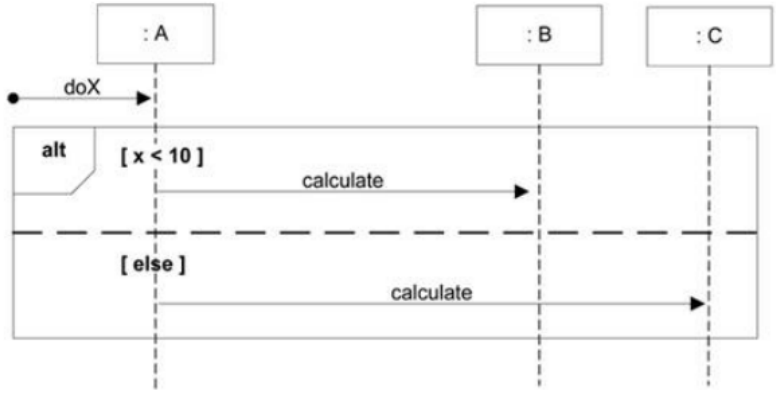


UML 序列图框架



- **alt**：表示的互斥条件逻辑的可选片段
- **loop**：当 true 时，循环片段。也可以写 loop(n) 来表示循环 n 次，还讨论了如何扩展以包含 for 循环
- **opt**：当为 true 时执行的可选片段
- **par**：并行执行的并行片段
- **region**：只有一个线程可以运行的临界区域

序列图示例




```
public class A{
    public void doX(){
        if(x < 10){
            B.calculate();
        }else{
            C.calculate();
        }
    }
}
```

6. 小结

UML 的好处

- 共同语言
  - 使共享需求、规格说明和设计变得更容易
- 可视化语法是有用的
  - 一图胜千言
  - 对于非技术人员来说，掌握简单直观的图甚至比掌握伪代码更容易
- 在某种程度上，UML是精确的，它要求清晰
  - 比自然语言好多了
- 商业工具支持
  - 这是自然语言永远不会有的

UML 的坏处

- 这是一堆想法的大杂烩
  - 最流行的建模语言的联合
  - 其他（有时有用的）子语言在很大程度上仍然是未集成的
- 可视化语法不能很好地伸缩
  - 许多细节很难用视觉描述
  - 大型图表没有可视化优势
- 语义还不完全清楚
  - UML 的部分未指定，不一致