[词汇 2](#_Toc277616018)

[使用UML建模，可以建立什么模型 2](#_Toc277616019)

[草图与蓝图 2](#_Toc277616020)

[建模参与人员任务分配 2](#_Toc277616021)

[UML 4](#_Toc277616022)

[UML模型关系的种类 8](#_Toc277616023)

[规格描述 10](#_Toc277616024)

[UML修饰与通用划分 11](#_Toc277616025)

[UML扩展机制 11](#_Toc277616026)

[UML定义的图 12](#_Toc277616027)

[UML视图和图 13](#_Toc277616028)

[UML图形分类 14](#_Toc277616029)

[开发过程 15](#_Toc277616030)

[开发过程解析 15](#_Toc277616031)

[Use Case之间的依赖关系 16](#_Toc277616032)

[几种用例关系的比较 18](#_Toc277616033)

[检查点：用例 18](#_Toc277616034)

[用例规格说明 20](#_Toc277616035)

[事件流描述到什么程度才是足够的？ 20](#_Toc277616036)

[将Use Case分层或分包 20](#_Toc277616037)

[要做软件先得了解软件的生命周期 21](#_Toc277616038)

[常用的软件生命周期模型及选择 21](#_Toc277616039)

[UML——Unified modeling language 22](#_Toc277616040)

[什么是模型 23](#_Toc277616041)

[UML图 23](#_Toc277616042)

[类图和对象图 24](#_Toc277616043)

[用例图 24](#_Toc277616044)

[Rose简介: 24](#_Toc277616045)

[用例描述文档 25](#_Toc277616046)

[类图 26](#_Toc277616047)

[接口 29](#_Toc277616048)

[类图—依赖关系（Dependence） 30](#_Toc277616049)

[类图—关联关系（Association） 31](#_Toc277616050)

[类图—聚合关系（Aggregation）(关联关系的特例) 32](#_Toc277616051)

[类图—组合关系（Composition）(关联关系的特例) 32](#_Toc277616052)

[活动图(Activity Diagram) 33](#_Toc277616053)

[时序图(Sequence Diagram) 34](#_Toc277616054)

[协作图(Collaboration Diagram，也叫合作图)是一种交互图（interaction diagram） 36](#_Toc277616055)

[状态图 37](#_Toc277616056)

[组件图 39](#_Toc277616057)

[部署图 40](#_Toc277616058)

## 词汇

|  |
| --- |
| Actor：主角  Use Case：用例  UML是Unified（统一）Modeling（建模） Language（语言） |

## 使用UML建模，可以建立什么模型

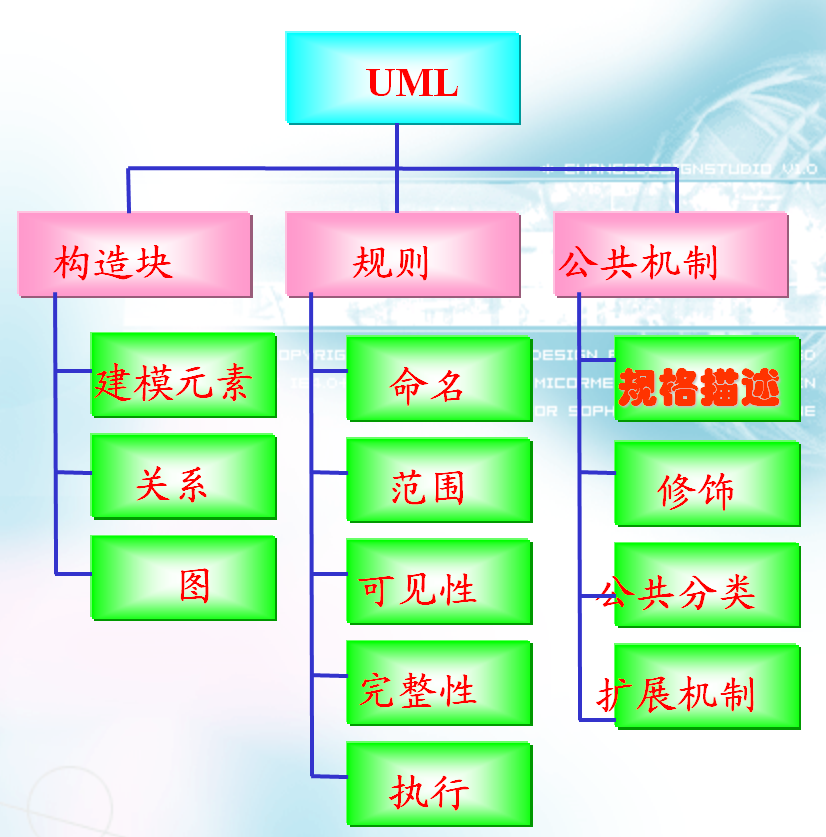
|  |  |
| --- | --- |
| 模型的种类 | 模型的用途 |
| 业务模型 | 对业务过程、工作流、组织的建模 |
| 需求模型 | 对捕获的需求进行整理和分析的工具，辅助开发人员与用户进行沟通 |
| 设计模型 | 包含高层设计（架构模型）和详细设计模型，用于统一开发人员、沟通设计信息 |
| 数据库模型 | 设计数据库的结构、表结构以及与应用系统的交互 |
| 实现模型 | 用来理清软件的组成、部署方案，为安装与维护人员的工作提供指导 |

## 草图与蓝图

|  |  |
| --- | --- |
| 蓝图 | 一般是指采用CASE工具绘制的、正式的、规范的UML模型 |
| 草图 | 则通常是指手工绘制的、规范度较低的在纸张的UML模型 |
| 大胆地绘制草图，尽可能基于草图进行讨论。对于局部的、重要性不高的、共享范围较小的UML模型，直接将草图扫描到电脑存档即可；  对于全局的、重要性高的、高度共享的，在草图的基础上用CASE工具绘制成为正式的蓝图，并将其纳入统一的模型管理中 | |

## 建模参与人员任务分配

|  |
| --- |
| 业务建模：以领域专家为主，需求分析人员是主力，系统分析员、架构师可参与  需求模型：以需求分析人员为主，系统分析员是主力，领域专家提供指导，架构师和资深开发人员参与  设计模型：高层设计模型以架构师为主，系统分析员从需求方面提供支持，资深开发人员从技术实现方面提供支持。详细设计模型则以资深开发人员为主，架构师提供指导。  实现模型：以资深开发人员（设计人员）为主，架构师提供总体指导。  数据库模型：以数据库开发人员为主，架构师提供指导，资深开发人员（设计人员）予以配合。 |



# UML

|  |  |  |  |
| --- | --- | --- | --- |
| 构造块 | 建模元素 | 1.结构元素(类,接口,协作,用例,对象,构件,节点等)  2.行为元素(交互,状态机)  3.分组元素(包)  4.注解元素 | 构造块是对模型中最具有代表性的成分的抽象  建模元素：UML中的名词，它是模型基本物理元素。  行为元素：UML中的动词，它是模型中的动态部分，是一种跨越时间、空间的行为。  分组元素：UML中的容器，用来组织模型，使模型更加的结构化。  注释元素：UML中的解释部分，和代码中的注释语句一样，是用来描述模型的。 |
| 关系 | 1.关联  2.依赖  3.泛化  4.实现 |  |
| 图 | 1.静态模型(类图,构件图,部署图)  2.动态模型(对象图,用例图,顺序图,协作图,状态图,活动图) |  |
| 规则 | 命名 |  |  |
| 范围 |  |  |
| 可见性 |  |  |
| 完整性 |  |  |
| 执行 |  |  |
| 公共机制 | 规格描述 |  |  |
| 修饰 |  |  |
| 公共分类 |  |  |
| 扩展机制 |  |  |

|  |  |
| --- | --- |
| 建模元素 | 构造元素、行为元素、分组和注释 |
| 关系 | 关联、泛化、依赖、实现 |

|  |  |
| --- | --- |
| 构造块 | 构造块是对模型中最具有代表性的成分的抽象  建模元素：UML中的名词，它是模型基本物理元素。  行为元素：UML中的动词，它是模型中的动态部分，是一种跨越时间、空间的行为。  分组元素：UML中的容器，用来组织模型，使模型更加的结构化。  注释元素：UML中的解释部分，和代码中的注释语句一样，是用来描述模型的。 |
| 建模元素 | 类（class）和对象（object）  接口（interface）  主动类（active class）  用例（use case）  协作（collaboration）  构件（component）  节点（node） |
| 类和对象 | 类是对一组具有相同属性、相同操作、相同关系和相同语义的对象的抽象  UML中类是用一个矩形表示的，它包含三个区域，最上面是类名、中间是类的属性、最下面是类的方法  对象则是类的一个实例 |
| 接口 | 接口是描述某个类或构件的一个服务操作集 |
| 主动类 | 主动类实际上是一种特殊的类。引用它的原因，实际上是在开发中需要有一些类能够起到  启动控制活动的作用  主动类是指其对象至少拥有一个进  程或线程，能够启动控制活动的类 |
| 用例与协作 | 用例是著名的大师Ivar Jacobson首先提出的，现已经成为了面向对象软件开发中一个需求分析的最常用工具  用例实例是在系统中执行的一系列动作，这些动作将生成特定执行者可见的价值结果。一个  用例定义一组用例实例。  协作定义了一个交互，它是由一组共同工作以提供某协作行为的角色和其他元素构  成的一个群体。  对于某个用例的实现就可  以表示为一个协作 |
| 构件 | 在实际的软件系统中，有许多要比“类”更大的实体，例如一个COM组件、一个DLL文件、一个JavaBeans、一个执行文件等等。为了更好地对在UML模型中对它们进行表示，就引入了构件（也译为组件）  构件是系统设计的一个模块化部分，它隐藏了内部的实现，对外提供了一组外部接口。在系统中满足相同接口的组件可以自由地替换 |
| 节点 | 为了能够有效地对部署的结构进行建模，UML引入了节点这一概念，它可以用来描述实际的PC机、打印机、服务器等软件运行的基础硬件  节点是运行时存在的物理元素，它表示了一种可计算的资源，通常至少有存储空间和处理能力 |
| 行为元素 | 交互（interaction）： 是在特定语境中，共同完成某个任务的一组对象之间交换的信息集合  交互的表示法很简单，就是一条有向直线，并在上面标有操作名  状态机（state machine）：是一个对象或交互在生命周期内响应事件所经历的状态序列  在UML模型中将状态画为一个圆  角矩形，并在矩形内写出状态名  称及其子状态 |
| 分组元素 | 对于一个中大型的软件系统而言，通常会包含大量的类，因此也就会存在大量的结构事物、行为事物，为了能够更加有效地对其进行整合，生成或简或繁、或宏观或微观的模型，就需要对其进行分组。在UML中，提供了“包（Package）”来完成这一目标 |
| 注释元素 | 结构事物是模型的主要构造块，行为事物则是补充了模型中的动态部分，分组事物而是用来更好地组织模型，似乎已经很完整了。而注释事物则是用来锦上添花的，它是用来在UML模型上添加适当的解释部分 |

## UML模型关系的种类

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 种类 | 变种 | 表示法 | 关键字或符号 | 种类 | 变种 | 表示法 | 关键字或符号 |
| 抽象 | 派生 | 依赖关系 | 《derive》 | 导入 | 私有 | 依赖关系 | 《access》 |
| 显现 | 《manifest》 | 公有 | 《import》 |
| 实现 | **实现关系** | 虚线加空心三角 | 信息流 |  | 《flow》 |
| 精化 | 依赖关系 | 《refine》 | 包含并 |  | 《merge》 |
| 跟踪 | 《trace》 | 许可 |  | 《permit》 |
| 关联 |  | **关联关系** | 实线 | 协议符合 |  |  | 未指定 |
| 绑定 |  | 依赖关系 | 《bind》(参数表) | 替换 |  | 依赖关系 | 《substitute》 |
| 部署 |  | 《deploy》 | 使用 | 调用 | 《call》 |
| 扩展 | Extend | 《extend》(扩展点) | 创建 | 《create》 |
| 扩展 | extension | **扩展关系** | 实线加实心三角 | 实例化 | 《instantiate》 |
| 泛化 |  | **泛化关系** | 实线加空间三角 | 职责 | 《responsibility》 |
| 包含 |  | 依赖关系 | 《include》 | 发送 | 《send》 |

各种关系

|  |  |
| --- | --- |
| 关联关系 | 关联（Association）表示两个类之间存在某种语义上的联系。关联关系提供了通信的路径，它是所有关系中最通用、语义最弱的。  在UML中，使用一条实线来表示关联关系 |
| 在关联关系中，有两种比较特殊的关系：聚合和组合  聚合关系：聚合（Aggregation）是一种特殊形式的关联。聚合表示类之间的关系是整体与部分的关系  如果发现“部分”类的存在，是完全依赖于“整体”类的，那么就应该使用“组合”关系来描述 |
| 组合是聚合的变种，加入了一些重要的语义。也就是说，在一个组合关系中一个对象一次就只是一个组合的一部分，“整体”负责“部分”的创建和破坏，当“整体”被破坏时，“部分”也随之消失  聚合就像汽车和车胎，汽车坏了胎还可以用。组合就像公司和下属部门，公司倒闭了部门也就不存在了！ |
| 泛化、实现与依赖 | 泛化关系描述了一般事物与该事物中的特殊种类之间的关系，也就是父类与子类之间的关系。    image008 |
|  | 有两个元素X、Y，如果修改元素X的定义可能会引起对另一个元素Y的定义的修改，则称元素Y依赖（Dependency）于元素X。  image004 |

UML规则

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 命名：也就是为事物、关系和图起名字。和任何语言一样，名字都是一个标识符  范围：与类的作用域相似.  可见性：   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 可见性 | 规则 | 标准表示法 | Rose属性 | Rose方法 | | public | 任一元素，若能访问包容器，就可以访问它 | **+** |  |  | | protected | 只有包容器中的元素或包容器的后代才能够看到它 | # |  |  | | private | 只有包容器中的元素才能够看得到它 | - |  |  | | package | 只有声明在同一个包中的元素才能够看到该元素 | ~ |  |  | |

## 规格描述

|  |
| --- |
| 在图形表示法的每个部分后面都有一个规格描述（也称为详述），它用来对构造块的语法和语义进行文字叙述。这种构思，也就使可视化视图和文字视图的分离 ： |

## UML修饰与通用划分

|  |
| --- |
| 在为了更好的表示这些细节，UML中还提供了一些修饰符号，例如不同可视性的符号、用斜体字表示抽象类  UML通用划分：  1）类与对象的划分：类是一种抽象，对象是一个具体  的实例  2）接口与实现的分离：接口是一种声明、是一个契  约，也是服务的入口；实现则是负责实施接口提供  的契约 |

## UML扩展机制

|  |
| --- |
| **构造型**：在实际的建模过程中，可能会需要定义一些特定于某个领域或某个系统的构造块    **标记值**: 则是用来为事物添加新特性的。标记值的表示方法是用形如“{标记信息}”的字符串  **约束**: 是用来增加新的语义或改变已存在规则的一种机制（自由文本和OCL两种表示法）。约束的表示法和标记值法类似，都是使用花括号括起来的串来表示，不过它是不能够放在元素中的，而是放在相关的元素附近 |

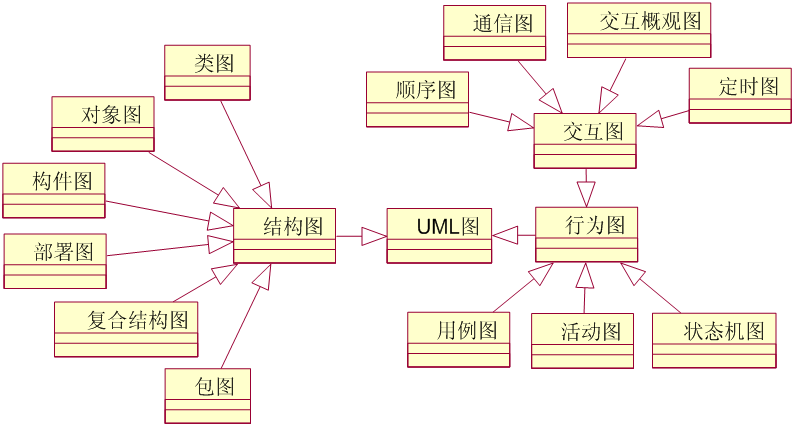
## UML定义的图

|  |  |  |
| --- | --- | --- |
| 图名 | 功能 | 备注 |
| 类图 | 描述类、类的特性以及类之间的关系 | UML 1原有 |
| 对象图 | 描述一个时间点上系统中各个对象的一个快照 | UML 1非正式图 |
| 复合结构图 | 描述类的运行时刻的分解 | UML 2.0新增 |
| 构件(组件)图 | 描述构件的结构与连接, 表示封装了其内容的系统模块, 构件是相对独立的模块. | UML 1原有 |
| 部署图 | 描述在各个节点上的部署, 表示软件元素在物理架构上的部署, 以及物理元素之间的通信. | UML 1原有 |
| 包图 | 描述编译时的层次结构 | UML中非正式图 |
| 用例图 | 描述用户与系统如何交互 | UML 1原有 |
| 活动图 | 描述过程行为与并行行为 | UML 1原有 |
| 状态机图 | 描述事件如何改变对象生命周期 | UML 1原有 |
| 顺序图 | 描述对象之间的交互，重点在强调顺序 | UML 1原有 |
| 通信图 | 描述对象之间的交互，重点在于连接 | UML 1中的协作图 |
| 定时图 | 描述对象之间的交互，重点在于定时 | UML 2.0 新增 |
| 交互概观图 | 是一种顺序图与活动图的混合 | UML 2.0新增 |

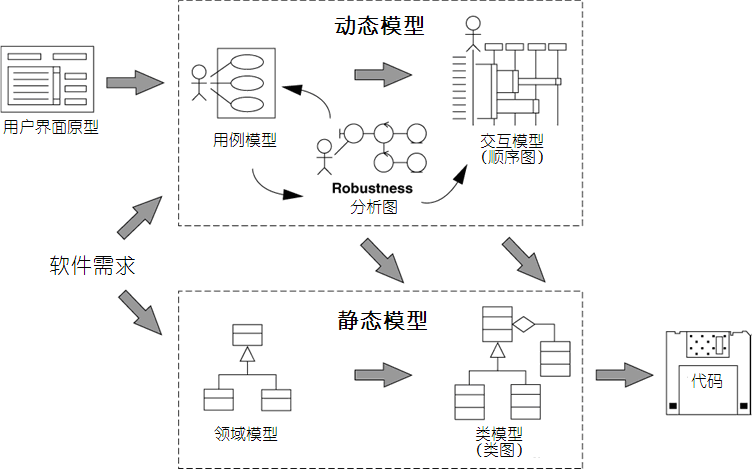
## UML视图和图

|  |  |  |
| --- | --- | --- |
| 主要领域 | 视图 | 图 |
| 结构 | 静态视图 | 类图 |
| 设计视图 | 复合结构图、协作图、构件图 |
| 用例视图 | 用例图 |
| 动态 | 状态视图 | 状态机图 |
| 活动视图 | 活动图 |
| 交互视图 | 顺序图、通信图 |
| 物理 | 部署视图 | 部署图 |
| 模型管理 | 模型管理视图 | 包图 |
| 特性描述 | 包图 |

## UML图形分类



## 开发过程



## 开发过程解析

|  |  |  |
| --- | --- | --- |
| 业务 | 目前的现实是什么？  前景-希望系统达到什么目的？ | 起源：为什么要开发 |
| 用例 | 谁“用”软件？用来做什么？做的细节？ | 开发出来什么样？ |
| 类图 | 软件内部需要哪些成分？  顺序图-这些成分怎样合作来完成用例？ | 分析（内部和新结构行为） |
| 设计 | 如何用选择的平台来实现分析？ | 设计（具体环境下的实现） |

## Use Case之间的依赖关系

|  |  |  |
| --- | --- | --- |
| <<extend>>关系  B的行为可以被组成到A中  扩展用例可以但不需要定义为泛化（基于）关系  在扩展Use Case(A)中的扩展点需要指定从基本Use case(B)扩展的位置 | 如果基本用例的一部分是可选的，或对于理解该用例的主要目的来说不是必需的，  那么您可以将这部分分离出来，形成一个附加用例，以简化基本用例的结构。  利用扩展关系，可以将附加用例隐式插入基本用例中。 |  |
| <<include>>关系  A的行为包括B  B必须也是一个用例（不要同备选事件流混淆） | 如果基本用例中有一部分功能，该用例的执行与否由它的结果唯一决定，  而不是由产生该结果的方法来决定，则可以将这一部分功能分离出来，  放到一个附加用例中。采用包含关系，可以将附加用例显式插入基本用例中。 |  |
| generalization关系  与类之间的泛化关系类似  B继承了A的行为并且允许重载和扩展它  B继承了A的通信关系（与actor或和其它use case）  对抽象use case建模也是可能的 ({abstract}) | 如果用例在行为和结构上具有共同点而且在目的上又很相似，  则可以将它们的共同部分分离出来，形成一个基本用例（父用例）。  而附加用例（子用例）可以继承该父用例。 子用例可以在从父用例继承的结构中插入新的行为或修改现有的行为。 |  |

## 几种用例关系的比较

|  |  |  |  |
| --- | --- | --- | --- |
| **问题** | **扩展关系** | **包含关系** | **泛化关系** |
| 该关系有什么方向？ | 附加用例引用基本用例。 | 基本用例引用附加用例。 | 附加用例（子用例）引用基本用例（父用例）。 |
| 关系是否存在多重性？ | 是，在附加用例一方。 | 否。 如果您希望不只一次包含相同的行为段，则必须在基本用例中声明。 | 否。 |
| 该关系是否存在某个条件？ | 是。 | 否。 如果您希望在该包含关系中表达某个条件，则必须在基本用例中明确说明。 | 否。 |
| 附加用例是否为抽象用例？ | 通常是，但不一定。 | 是。 | 通常不是，但也有可能是。 |
| 基本用例是否通过附加用例修改？ | 扩展关系隐式修改基本用例的行为。 | 包含关系显式修改基本用例的效果。 | 如果将基本用例（父用例）实例化，则其不受子用例的影响。 为了获得附加用例的效果，附加用例（子用例）必须实例化。 |
| 基本用例是否必须完整而有意义？ | 是。 | 包括附加用例在内，都必须是。 | 如果它是抽象用例，则为否。 |
| 附加用例是否必须完整而有意义？ | 否。 | 否。 | 包括基本用例（父用例）在内，必须是。 |
| 附加用例是否可以访问基本用例的属性？ | 是。 | 否。 包含关系是被封装的，而且只能“看见”它自己。 | 是，通过标准的继承机制。 |
| 基本用例是否可以访问附加用例的属性？ | 否。 在没有附加用例的情况下，必须很好地组织基本用例。 | 否。 基本用例只了解附加用例的效果。附加用例已被封装。 | 否。 在没有附加用例（子用例）的情况下，必须要很好地组织基本用例（父用例）。 |

## 检查点：用例

|  |
| --- |
| 每个具体用例是否至少涉及一个主角？ 如果不是，则其中存在问题；与主角不进行交互的用例是多余的，您应该将它删除。  用例之间是否相互独立？如果两个用例总是以同样的顺序被激活，您可能需要将它们合并为一个用例。  所有用例是否都有非常相似的行为或事件流？如果有 - 而且您还希望它们的行为在将来也相似 - 您应该将它们合并为一个用例。这便于在将来引入变更。注：如果决定合并用例就必须包含它们的用户，因为与新合并的用例进行交互的用户可能会受到影响。  该事件流的一部分是否已被构建为另一个用例的模型？如果是，您可以让新用例使用旧的事件流。  此事件流的某一部分是否已经成为另一个用例的组成部分？如果是，您应该提取该分支流并让上述的用例使用它。注：如果决定“重新使用”分支流就必须包含这些用户，因为现有用例的这些用户可能会受到影响。  是否应该将一个用例的事件流插入另一个用例的事件流中？如果是，利用与另一个用例的扩展关系来建立此模型。  用例的名称是否具有唯一性、直观性和解释性，这样以后就不至于将它们混淆？如果没有，则更改它们的名称。  客户和用户是否都了解用例的名称和说明？每个用例名称必须说明其支持的行为。  用例是否满足明显支配其性能的所有需求？在用例特殊需求中必须包含要在对象模型中处理的任何（非功能性）需求。  主角和用例之间的通信顺序是否符合用户的期望？  是否明确说明用例的事件流开始及结束的方式和时间？  可能存在仅在未符合某一特定条件时才被激活的行为。是否说明未满足给定条件时将发生的事情？  是否有过于复杂的用例？如果要使用例模型易于理解，您最好将复杂用例进行分解。  用例是否包含完全不同的事件流？如果包含，最好将其分成两个或更多个单独的用例。包含完全不同的事件流的用例将难于理解和维护。  是否精确建立业务用例中的分支流的模型？  是否清楚地说明执行用例的对象？是否还清楚地说明用例的目的？ |

## 用例规格说明

|  |
| --- |
| 描述Use Case  命名  每个用例都应有一个名称，以表明它与主角进行交互的结果。名称应该是几个便于理解的单词。用例名称不应重复。  简述  用例的简要说明应反映用例的角色和目的。  事件流  采用文本描述Use Case中系统做什么的流程。可能有多个事件流，如基本流和备选流  基本流：用例正常执行的主要场景（流程）  备选流：分支流,异常流  特殊需求  说明事件流没有包括的所有用例需求。有些非功能性需求将会影响设计模型。  前置条件  成功执行的先决条件  后置条件  用例成功完成后的系统状态  扩展点  扩展点使用例具有了扩展的可能性。一个扩展点有一个名称和一系列对用例事件流中一个或多个位置的引用。 用在具有扩展关系的用例中 |

## 事件流描述到什么程度才是足够的？

|  |
| --- |
| 不能太多也不能太少  用例需要清晰地描述它的行为，不多也不少，太多则可能影响对系统的正确理解，太少则可能信息不够，以至于无法进行后期的分析和设计  不能遗漏  如果系统要以某种方式响应某个事件，那么它一定要说明  不能指定或限制设计  不要忽视细节  事件流中交换的信息是重要的，必须要清楚的描述。  重复的定义应当放置到术语表中 |

## 将Use Case分层或分包

|  |
| --- |
| 为达到更易于理解的目的，可以对用例模型的进行分包和分层。  分层。用例模型可以组织为一个有层次的用例包结构， 而主角或用例是该结构中的“树叶”。  分包。对用例进行分组，形成多个包。 |

## 要做软件先得了解软件的生命周期

|  |
| --- |
| 1、问题的定义及**规划**： （可行性分析报告和软件开发计划）  此阶段是软件开发方与需求方共同讨论，主要确定软件的开发目标及其可行性。  2、**需求分析**： (需求分析说明书和初步的用户手册)  在确定软件开发可行的情况下，对软件需要实现的各个功能进行详细分析。需求分析阶段是一个很重要的阶段，这一阶段做得好，将为整个软件开发项目的成功打下良好的基础。"唯一不变的是变化本身。"，同样需求也是在整个软件开发过程中不断变化和深入的，因此我们必须制定需求变更计划来应付这种变化，以保护整个项目的顺利进行。  3、**软件设计**： (概要设计、详细设计)  此阶段主要根据需求分析的结果，对整个软件系统进行设计，如系统框架设计，数据库设计等等。软件设计一般分为总体设计和详细设计。好的软件设计将为软件程序编写打下良好的基础。  4、**程序编码**： （提交源程序及清单)  此阶段是将软件设计的结果转换成计算机可运行的程序代码。在程序编码中必须要制定统一，符合标准的编写规范。以保证程序的可读性，易维护性，提高程序的运行效率。  5、软件**测试**： (提交软件维护测试报告)  在软件设计完成后要经过严密的测试，以发现软件在整个设计过程中存在的问题并加以纠正。整个测试过程分单元测试（白盒）、集成测试（黑盒，功能测试、强度性能测试）以及系统测试三个阶段进行。测试的方法主要有白盒测试和黑盒测试两种。在测试过程中需要建立详细的测试计划并严格按照测试计划进行测试，以减少测试的随意性。  6、**运行维护**：软件维护是软件生命周期中持续时间最长的阶段。在软件开发完成并投入使用后，由于多方面的原因，软件不能继续适应用户的要求。要延续软件的使用寿命，就必须对软件进行维护。软件的维护包括纠错性维护和改进性维护两个方面。（提交软件维护报告） |

## 常用的软件生命周期模型及选择

|  |
| --- |
| **1. 严格按软件生命周期执行**的**瀑布模型** |
| **2. 增量**和**迭代**模型  增量迭代是RUP （Rational Unified Process）统一过程常采用的软件开发生命周期模型.增量和迭代有区别但两者又经常一起使用.所以这里要先解释下增量和迭代的概念.假设现在要开发A,B,C,D四个大的业务功能,每个功能都需要开发两周的时间.  则对于**增量**方法而言可以将四个功能分为两次增量来完成,第一个增量完成A,B功能,第二次增量完成C,D功能;  而对于**迭代**开发来将则是分两次迭代来开发,第一次迭代完成A,B,C,D四个基本业务功能但不含复杂的业务逻辑,而第二个功能再逐渐细化补充完整相关的业务逻辑.  在第一个月过去后采用增量开始时候A,B全部开发完成而C,D还一点都没有动;而采用迭代开发的时候A,B,C,D四个的基础功能都已经完成.  RUP强调的每次迭代都包含了需求,设计和开发,测试等各个过程,而且每次迭代完成后都是一个可以交付的原型.迭代不是并行,在每次迭代过程中仍然要遵循需求->设计->开发的瀑布过程.迭代周期的长度跟项目的周期和规模有很大的关系.小型项目可以一周一次迭代,而对于大型项目则可以2-4周一次迭代.如果项目没有一个很好的架构师,很难规划出每次迭代的内容和要到达的目标,验证相关的交付和产出.因此迭代模型虽然能够很好的满足与用户的交付,需求的变化,但确是一个很难真正用好的模型.  未标题-1.jpg |

## UML——Unified modeling language

|  |
| --- |
| 一件复杂事情的做法怎样才能想清楚呢？是不是可以借助一些道具、技巧和方法来帮助想呢？想清楚以后是不是应该在纸面上留下一些结果，以方便自己使用和与人交流？这个结果用什么格式来记录，别人才容易看懂呢？UML是一种用于软件系统分析和设计的语言工具，它用于帮助软件开发人员进行思考和记录思路的结果。  uml本身是一套符号的规定，就像数学符号和化学符号一样，之所以出现这些符号定义，是因为这些符号背后对应着一套思想和方法，这些符号用于帮助描述这套思想和方法的，这些符号是由这套思想和方法催生的。要学uml，就是要借助这些符号来掌握背后的思想和方法，这些符号虽然必须掌握，但它远不如它背后对应的思想和方法重要。  必须熟练掌握了某种面向对象的编程语言和跟着实施了若干个软件项目，才适合学习uml和理解uml中的一些内容，才会有好的学习效果。很难想象一个没有在铁路工地上工作过的人，怎么去设计铁路！！！ UML解决编码前的**设计**问题，而不解决编码过程的实施问题，我们前面学习的各种编程技术是解决编码过程的**实施**，必须有了这些基础才能理解和运用UML。 |

## 什么是模型

|  |
| --- |
| 一幢大楼在建造前必须先有图纸将其外观、内部结构、施工方案描述清楚，这些图纸就是其模型，现实生活中你还接触过哪些模型呢？地图、地球仪、乐谱等都是模型。  模型的三个特点：  简化：模型可以简化表达出事物的某些方面，让人们可以方便快速地审视、交流和评估事物和系统。  多视角：模型具有不同的视角，有的是对总体进行考量、有的是对局部细节进行考量，例如整个城市的地图和某个街道的局部放大图，整个地图只展示主要的街道和干道，而街道的局部放大图则详细展示出街道的信息。  通用符号：模型的符号要通用和标准，人们讨论模型时的精力就可以集中在对事物本身的理解和研究上，而不是把时间和精力花费在对符号模型的理解上。模型符号需要学习，但不是重点，例如，学音符只是学音乐中的一小部分工作，只是为了能够通过音符将乐曲表达出来和能够看到乐曲，学音乐的重点工作在于学习作曲技巧上。  软件系统在开发前，也要通过模型来将其功能，内部结构，开发方案描述清楚。 |

## UML图

|  |
| --- |
| **用例图**  静态结构图：**类图**、对象图、包图、组件图、部署图  动态行为图：交互图（**时序图**与协作图）、**状态图**、**活动图**  画UML图与写文章差不多，都是把自己的思想描述给别人看，关键在于思路和条理，图好看与否就是看你的字是否规范，至于工具，就像你用什么笔，不算非常重要，关键在于意，而不在于形。 |

## 类图和对象图

|  |
| --- |
| 用于描述系统中的对象类本身的组成和对象类之间的各种静态关系。  类之间的关系：泛化（继承）、依赖、关联、聚合与组合  对象图描述一组对象和它们之间的联系，它是系统状态的某一时刻的快照，它的使用相当有限，它主要用于了解系统在某个特定时刻的具体状况和数据结构。  对象图表示方法与类图大体相同，对象图中的对象属性可以有具体值，类图中的一个类可以对应成对象图中多个对象，例如，部门类的自关联就可以对应成多个部门对象之间的关联。 |

# 用例图

|  |
| --- |
| 用例建模是UML建模的一部分，用例建模的最主要功能就是用来表达系统的功能性需求或行为。  参与者（Actor）：参与者不是特指人，是指系统以外的，在使用系统或与系统交互中所扮演的角色。因此参与者可以是人，可以是事物，也可以是其他系统等等。需要注意的是，参与者不是指人或事物本身，而是表示人或事物当时所扮演的角色。参与者在画图中用简笔人物画来表示：    用例是是系统为参与者提供的功能。  对于对用例的命名，我们可以给用例取一个简单、描述性的名称，一般为带有动作性的词。  用例在画图中用椭圆来表示，椭圆下面附上用例的名称： |

## Rose简介:

|  |  |  |
| --- | --- | --- |
|  | 放哪些图? |  |
| 用例图 | 用例图 |  |
| 逻辑视图 | 包图  类图  状态图 |  |
| 组件图 | 部署部分 |  |

## 用例描述文档

|  |
| --- |
| 用例图只是简单地用图描述了一下系统，但对于每个用例，我们还需要有详细的说明，这样就可以让别人对这个系统有一个更加详细的了解，这时我们就需要写用例描述。  　　对于用例描述的内容，一般没有硬性规定的格式，但一些必须或者重要的内容还是必须要写进用例描述里面的。用例描述一般包括：简要描述（说明）、前置（前提）条件、基本事件流、其他事件流、异常事件流、后置（事后）条件等等。下面说说各个部分的意思：  **简要描述**：对用例的角色、目的的简要描述；  **前置条件**：执行用例之前系统必须要处于的状态，或者要满足的条件；  **基本事件流**：描述该用例的基本流程，指每个流程都“正常”运作时所发生的事情，没有任何备选流和异常流，而只有最有可能发生的事件流；  **其他事件流**：表示这个行为或流程是可选的或备选的，并不是总要执行它们；  **异常事件流**：表示发生了某些非正常的事情所要执行的流程；  **后置条件**：用例一旦执行后系统所处的状态；  用例图和用例描述设计实例请看附件文档  例如:  主动登录 |

# 类图

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 新建类图    类图的图标     |  |  | | --- | --- | |  | 类 | |  | 接口 | |  | 单向关联 (谁关联谁, 就谁指向谁) | |  | 双向关联 | |  | 关联类 | |  | 依赖: "尾" 依赖 "头"  谁使用谁, 就是谁依赖谁.  eg:  Action使用request, 就是Action依赖request, 箭头就是Action指向request. | |  | 泛化(就是: 继承) 是依赖关系的特例(具体一点的东西) | |  | 实现 |   选择需要的类, 生成代码(只用于学习, 由于功能还不够强, 所以工作中不用): |
| 包, 版型表示法(双尖括号内描述了所属分层) |

## 接口

|  |
| --- |
| 改变现实图标 |

## 类图—依赖关系（Dependence）

|  |
| --- |
| 只要是在类中用到了对方，那么他们之间就存在依赖关系。如果没有对方，连编绎都通过不了。  public class PersonServiceBean {  private PersonDao personDao;  public void save(Person person){}  public IDCard getIDCard(Integer personid){}  public void modify(){  Department department = new Department();  }  }  public class PersonDao{}  public class IDCard{}  public class Person{}  public class Department{} |

## 类图—关联关系（Association）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 关联关系实际上就是类与类之间的联系，他是依赖关系的特例  关联具有导航性：即双向关系或单向关系  关系具有多重性：   |  |  | | --- | --- | | 0 |  | | 0..1 |  | | 0..n |  | |  |  | | 1 | 有且仅有一个 | | 1..n |  | | n |  | |  |  | | 0... | 0个或者多个 | | 0，1 | 0个或者一个 | | n...m | n到 m个都可以 | | m...\* | 至少m个 | |
| **单向一对一关系**  public class Person {  private IDCard card;  }  public class IDCard{  } |
| **双向一对一关系**  public class Person {  private IDCard card;  }  public class IDCard{  private Person person  } |

## 类图—聚合关系（Aggregation）(关联关系的特例)

|  |
| --- |
| 聚合关系（Aggregation）表示的是整体和部分的关系，**整体与部分可以分开**。聚合关系是关联关系的特例，所以他具有关联的导航性与多重性。  如：一台电脑由键盘(keyboard)、显示器(monitor)，鼠标等组成；组成电脑的各个配件是可以从电脑上分离出来的  使用带空心菱形的实线来表示： |

## 类图—组合关系（Composition）(关联关系的特例)

|  |
| --- |
| 也是整体与部分的关系，但是**整体与部分不可以分开**。整体被删除, 部分也同时被删除.  如：在Hibernate中我们定义了两个实体：Person与IDCard，在Person实体中定义了对IDCard进行**级联**删除，即删除Person时连同IDCard一起删除  public class Person{  private IDCard card;  }  public class IDCard{  } |

# 活动图(Activity Diagram)

|  |
| --- |
| 在UML里,活动图本质上就是流程图,它描述系统的活动,判断点和分支等。下面是用户登录的活动图    新建一个活动图 |

# 时序图(Sequence Diagram)

|  |
| --- |
| 时序图（Sequence Diagram）是强调消息时间顺序的交互图。时序图描述类系统中类和类之间的交互，它将这些交互建模成消息交换。时序图是一个模型，用于描述对象组如何随着时间在某些行为方面进行协作。  时序图是一种强调消息时序的交互图，他由活动者（Actor）、对象（Object）、消息（Message）、生命线（Lifeline）和控制焦点（Focus of control）组成。在UML中，对象表示为一个矩形，其中对象名称标有下划线；消息在时序图中由有标记的箭头表示；生命线由虚线表示，控制焦点由薄薄的矩形表示(也称可为Activation Bar “活动条”)。  新建时序图: |

## 协作图(Collaboration Diagram，也叫合作图)是一种交互图（interaction diagram）

|  |
| --- |
| 强调的是发送和接收消息的对象之间的组织结构。一个协作图显示了一系列的对象和在这些对象之间的联系以及对象间发送和接收的消息。  时序图主要侧重于对象间消息传递在时间上的先后关系，而协作图则侧重于对象间及对象和角色间交互的静态关系。  时序图转换为协作图 |

# 状态图

|  |
| --- |
| 状态图: 描述状态的变化.  巴巴运动网的订单状态:    新建状态图 |

## 组件图

|  |
| --- |
| 组件图用来建立系统的各个组件之间的关系，它们是通过功能或者文件组织在一起，使用组件图可以帮助读者了解某个功能位于软件包的哪一位置，以及各个版本的软件包含那些功能。如javabean、ejb 、jsp都是组件。在UML中，组件使用在左侧有两个小矩形的大矩形表示。  组件图可以用来帮助设计系统的整体构架。  新建组件图: |

## 部署图

|  |
| --- |
| 用UML画太丑了, 一般不用UML来画. |

OOAD 最关心流程与组件

|  |
| --- |
| 1. 描述流程(剧情) ---- 分a析  2. 安排主/配角(组件)演出 ---- 设计 |