第一章 软件工程导论

- **软件系统的特点：**

1. 复杂的创造 complex creations

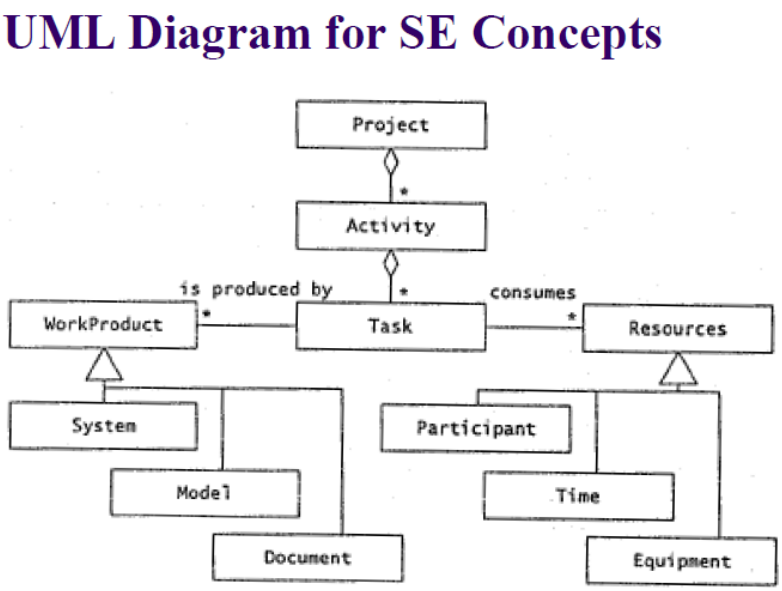
很多功能

实现许多不同的（往往又是矛盾的）目标

包含许多组成部分

不同的参与者

开发流程和软件生命周期经常持续很多年



2. 容易发生变化 subject to constant changes

客户或终端用户需求变化

发现错误

开发者有了很好的理解

新技术出现，员工变迁

- **软件工程的定义：**

软件工程是一项建模活动，是一项解决问题的活动，是一项知识获取的活动，是一项受软件工程原理指导的活动。

1. 建模

软件工程师通过建模解决复杂性问题

模型：系统的抽象体现，使我们可以回答系统的问题并直观理解系统

2. 问题求解

在有限的预算和时间下，模型寻求合理的解决方案

OOSE：object-oriented software engineering

需求获取和分析阶段：问题形式化并构建问题域模型（明确问题，分析问题）

系统设计：分解问题，选择通用策略设计系统

对象设计：从可选择方案中为每一小问题选定一种最合适的方案（系统设计和对象设计产生解答域模型，对应于“寻找解决方案”“选定合适的解决方案”）

实现：把解答域模型转化为可执行表达（详细说明解决方案）

测试：软件工程在问题解决过程中，应用域和解答域还在发生变化。

3. 知识获取

软件工程师收集数据，组织成信息，形成知识

非线性过程

4. 原理驱动

软件工程师通常的任务是修改一个当前可操作的软件系统以同这种新技术相协调。需要了解作出决定的环境条件和做出这些决定的基本原理（额外的知识）用来应对变化

- **SE 概念：**

技术、方法和工具的集合，可以在有限的时间、预算以及变化出现的情况下实现高质量的软件系统

参与者和角色 Participants and Roles

所有参与到这个项目中的人员都称为参与者（具体的一个或一类人）

项目或系统的一组职责称为角色（抽象的与任务相关联的）

一个角色与一组任务联系在一起且被指派给一个参与者。一个参与者能充当多个角色。

系统和模型 Systems and Models

系统指内部相关联部分的集合

建模指系统的任何抽象

一个开发项目本身就是一个可建模的系统，其项目规划、预算，以及预计的交付期都是开发项目的模型

工作产品 Work Products

内部产品 Internal work product：供给项目内部使用的工作产品（eg：state report 状态报告，test manual 测试手册，class model，source code）

交付的 Deliverable 工作产品：必须交付给客户使用的工作产品（eg:规格说明 specification，操作手册 operation manual）

活动、任务和资源 Activities, Tasks, and Resources

一种活动是为了某一具体目的所需完成的任务的集合。

一种任务代表一种可管理的原子工作单位。任务消耗资源，并依赖于其他任务产生的工作产品来产生出新的工作产品。

资源是用来完成工作的资产，包括时间、设备和劳动力

功能需求和非功能需求 Functional and Nonfunctional Requirements

需求说明了一个系统必须具有的一组特征

功能性需求是系统必须支持功能的规格说明

非功能性需求是对系统操作的一种约束，与系统的功能没有直接关系

记号，方法，方法论 Notations, Methods and Methodologies

记号是表示一个模型规则的图示或者文本集合

方法是一种可以重复的技巧，它说明解决某个具体问题所用的步骤

方法学是解决一类问题的方法集合，它规定了每种方法何时、一何种方式使用

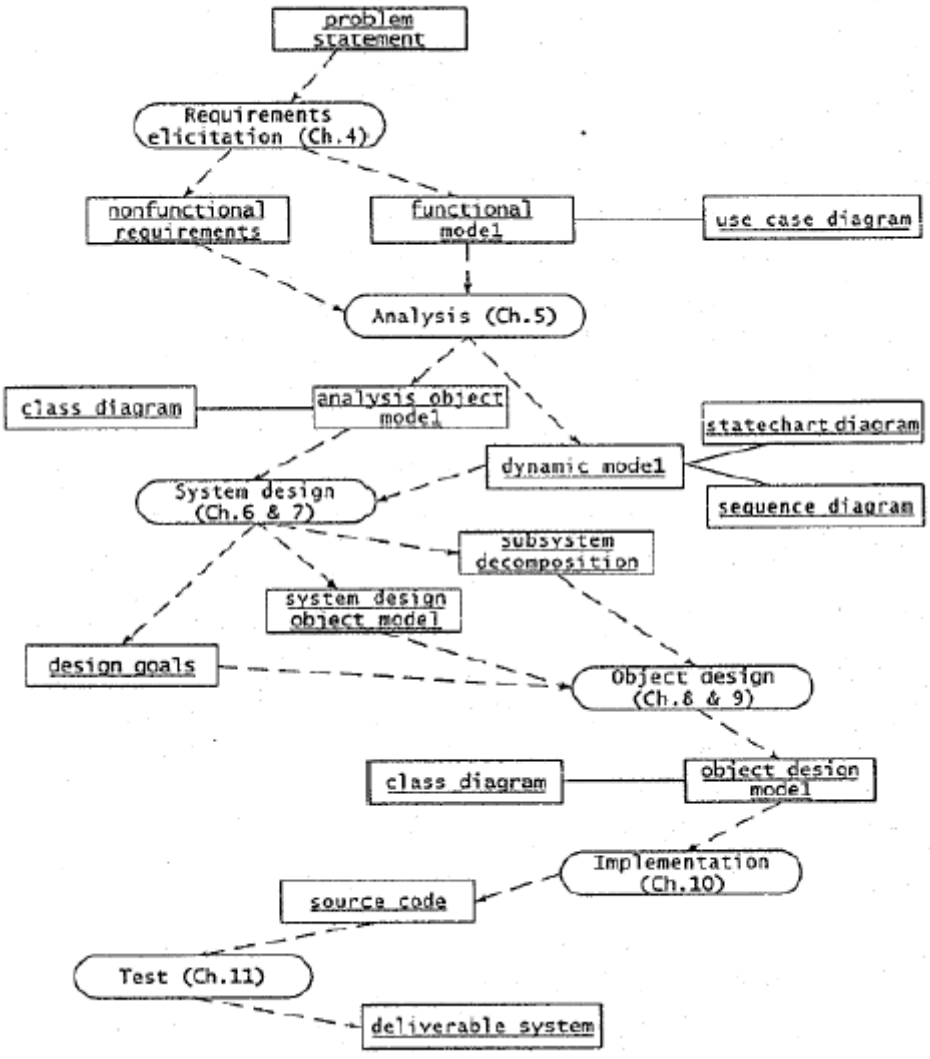
OMT：三种活动方法——分析、系统设计、对象设计（假设需求已经被定义）

本书用到的方法论：

1.需求获取和分析
2.系统设计和对象设计
3.变化相关的活动
4.配置管理

SE 不仅是有关开发，也关于管理

开发活动：需求获取，需求分析，系统设计，对象设计，实现测试
管理活动：交流，原理管理，软件配置管理，项目管理，软件生命周期



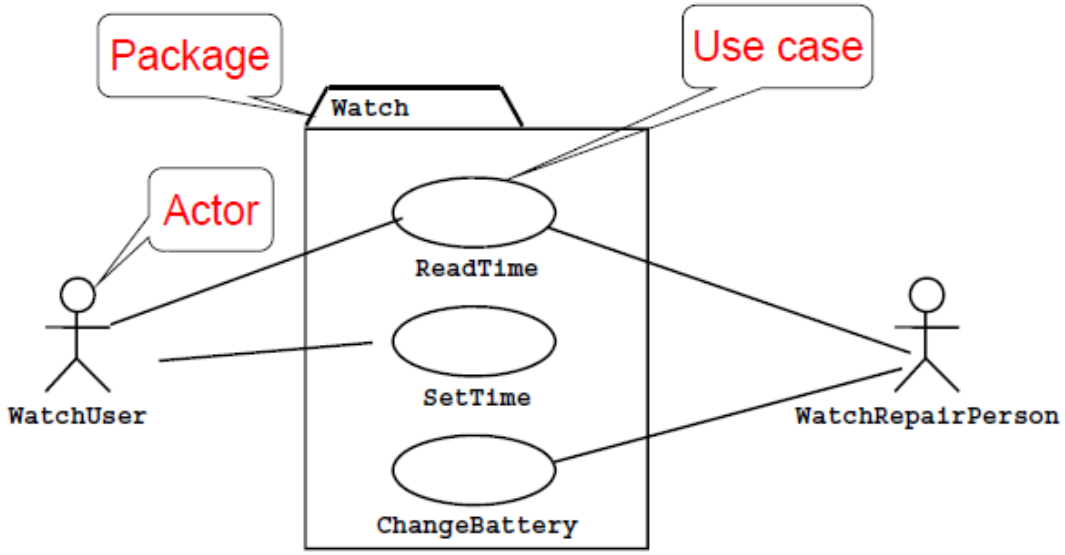
第二章 使用 UML 进行建模

UML：Unified Modeling Language
面向对象软件建模中出现的标准
创始人：
OMT (James Rumbaugh)
OOSE (Ivar Jacobson)
Booch (Grady Booch)

用途广泛：Powerful, but complex
功能模型：用例图（用户角度）
对象模型：类图（对象、属性、联系、操作角度）
　　需求分析—>分析对象模型—>应用概念
　　系统设计—>系统设计模型—>系统接口描述
　　对象设计—>对象设计模型—>解决方案对象的详细描述
动态模型：交互图—>在一系列对象之间进行一系列消息交换来描述行为
　　状态机图—>针对某一个对象的状态转换
　　活动图—>针对控制和数据流描述行为

应用域：表示用户问题的所有方面，刻画真实问题的要求
解答域：是所有可能系统的建模空间，利用应用域的结果，对功能进行实现，针对计算机世界
面向对象分析关心应用域的建模，面向对象设计关心解答域的建模

用例图：（&& 第四章）
描述系统功能，在需求获取和分析时使用，从外部角度来关注系统行为
用例：描述系统提供的功能，产生用户可见的结果
参与者：任何与系统交互的人（用户，另一个系统，系统的物理环境）
参与者在系统边界外，用例在系统边界内



（乃们一定要记住用例名是写在这个椭圆下面的啊 T T 学长考试的时候全写在里面了！）
　　用例从一个参与者的观点出发，描述了所看到的系统行为。用例模型描述的行为称为外部行为
用例描述：
1. **用例名称：**可以无二义性的使用该用例

2. **参与者：**与系统进行交互的外部实体
3. **事件流：**核心！描述了用例交互序列。公共情况和意外情况使用不同的用例分别描述
4. **入口条件：**描述了在该用例被启动之前需要满足的条件
5. **出口条件：**描述了在该用例完成之后需要满足的条件
6. **质量需求：**与系统功能无关的需求（包括系统性能、其实现、其运行的硬件平台方面等约束）

用例图可以包含四种不同的关系

通信：有交换信息，用实线表示。《initiate》通过一个参与者表示是谁启动了该用例；《participate》表示了一个参与者（其未启动该用例）与该用例的通信。《include》代表被分离出来的用例，使用包括关系可以将用例中的冗余部分分离出来。用开箭头的虚线表示，箭头指向使用的用例（一个用例中包含其他用例）在用例的质量需求中说明。《extend》代表**异常或很少调用的用例**，用开箭头的虚线表示，箭头指向被扩展的用例，扩展关系表示成扩展用例的入口条件。

对包括关系而言，触发目标（即被包括）事件用例使用源用例的事件流描述（在包括用例要使用的地方，必须说明每一个包括用例）；对扩展关系而言，触发源事件用例用源用例作为前置条件（只是在扩展用例时说明哪一个用例被扩展，而在使用时不必再进行说明）。对异常行为、选择行为、或很少发生的行为而言，使用扩展关系；当行为被两个或多个用例共享时，使用包括关系。

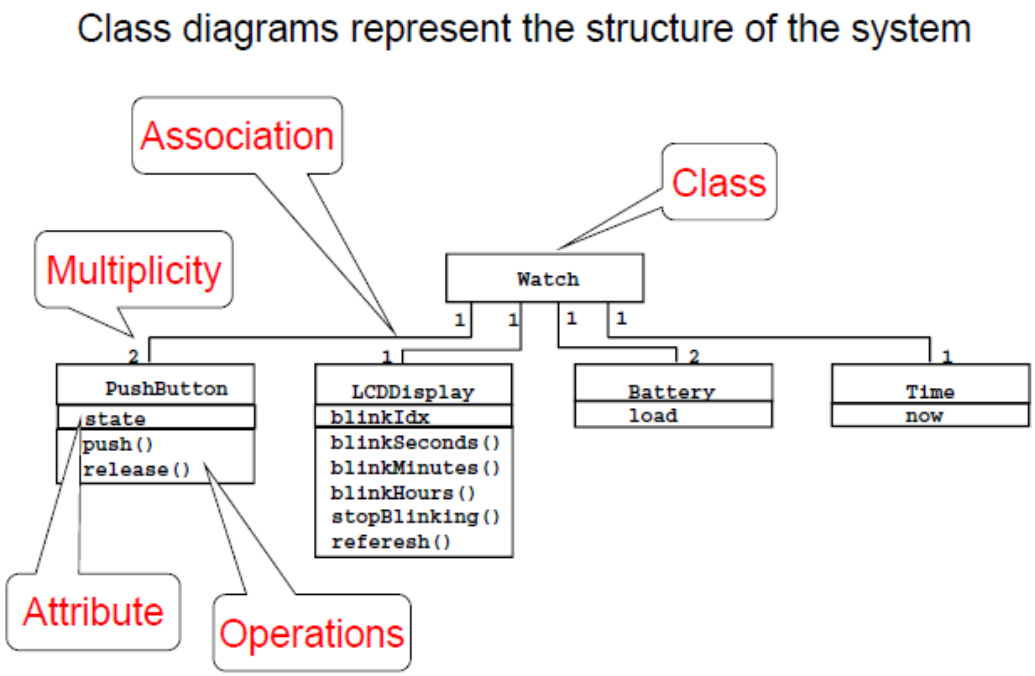
《inheritance》代表一个用例可以通过添加细节特化另一个更一般的用例，用空箭头的实线表示，箭头指向那个一般的用例

场景：是用例的实例，用于描述一组活动的具体集合，用做实例，举例说明公共情况；关心的是可理解性（名字：无二义性，名字下带有下划线以说明是一个实例；参与者实例域说明参与者实例包含了这一场景，参与者实例也有下划线；事件流一步步描述事件顺序）

用例是场景的抽象提取。用例可以被多个对象使用。用例描述了一系列从初始情况出发的相关交互

类图：（client 和 end user 不关注类图）

描述系统的结构（静态性质，运用于不同阶段）



类：描述具有相同结构和行为的对象集的抽象

对象：在系统执行过程中被创建、修改和销毁的类的实体

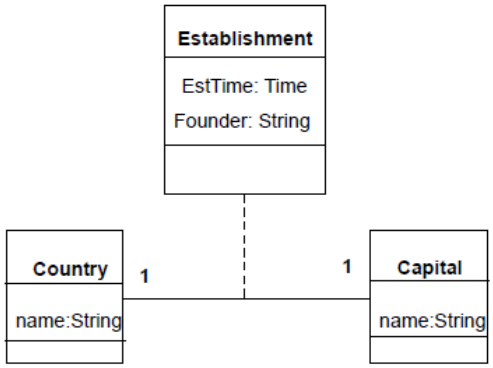
有状态（包括属性值和与其他对象的联系）

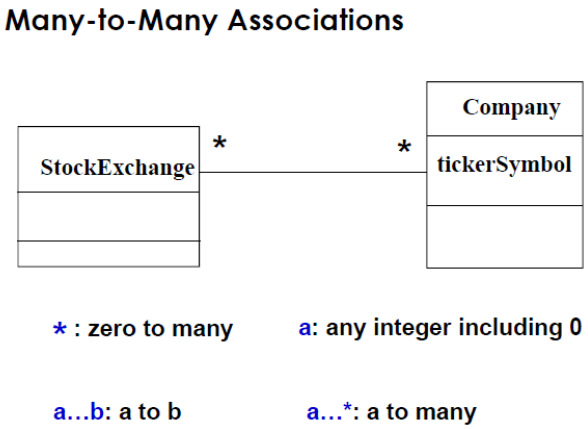
类图的成分：类，对象，属性，操作，联系

需求获取分析中对应用域概念建模，类表示了在用例图和交互图中发现的参与对象，可描述类的属性和操作

在系统设计中对子系统建模，在对象设计中说明类的详细行为和属性。这两个阶段对类图进行求精，将代表解答域的类包括进模型中

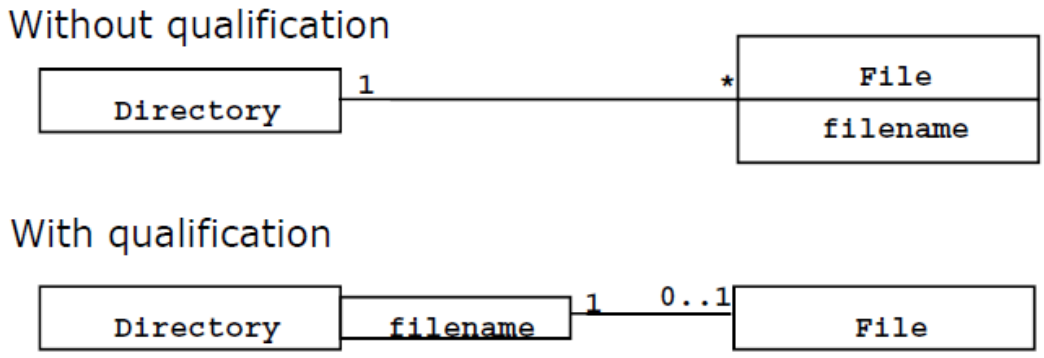
一条链表示两个对象之间的连接，关联是类之间的关系并表示为一组链
关联可以具有属于自己的属性和操作，称之为关联类
关联的每一端可以用一个串标识，该串称为角色
关联的每一端可以用一整数标识，以说明链的数目，该整数集合称为关联端的重数
关系也可以通过属性来刻画
关系类可以用自己的操作和属性，用虚线连在关系线上





聚集：整体——部分
　　父类：被聚集的对象、整体
　　子类：聚集的对象、部分
共享聚集：表示一种“属于”继承，空心菱形
组合聚集：更强形式的聚集，生命周期必须一致，实心菱形

限制（qualification）：减少关系的复杂性
在此的关键字成为受限符



继承：一般到特殊
是通用类和一个或者多个特殊类之间的一种关系。
“是一种”的继承 a “kind-of” hierarchy，通过分类简化分析模型，子类继承父类的操作和属性
用斜体名字描述抽象类

对象模型建立步骤：
1. 找到新的类
2. 定义名字、属性、方法
3. 找到类之间的关系
4. 标注一般的关系（has\owns,etc.）

5. 决定关系之间的多重性
6. 重审关系
7. 找到分类（使用继承）
8. 简化、重组

用于描述系统动态性质——交互图、状态机、活动图

交互图：（顺序图或协作图）

用于将系统动态行为形式化，将对象之间的通信可视化。在用例中涉及的对象（参与对象），表现的就是这些对象之间发生的交互

揭示类图中类的可响应性，甚至可以通过这一构造发现新类。每一个关注事件流的用例有一个交互图

顺序图：

用表示交互的横向轴和表示时间的纵向轴描述了参与交互的对象

在分析中，优化用例描述，找到更多的对象（参与对象）（应用域）

在系统设计中，优化系统接口

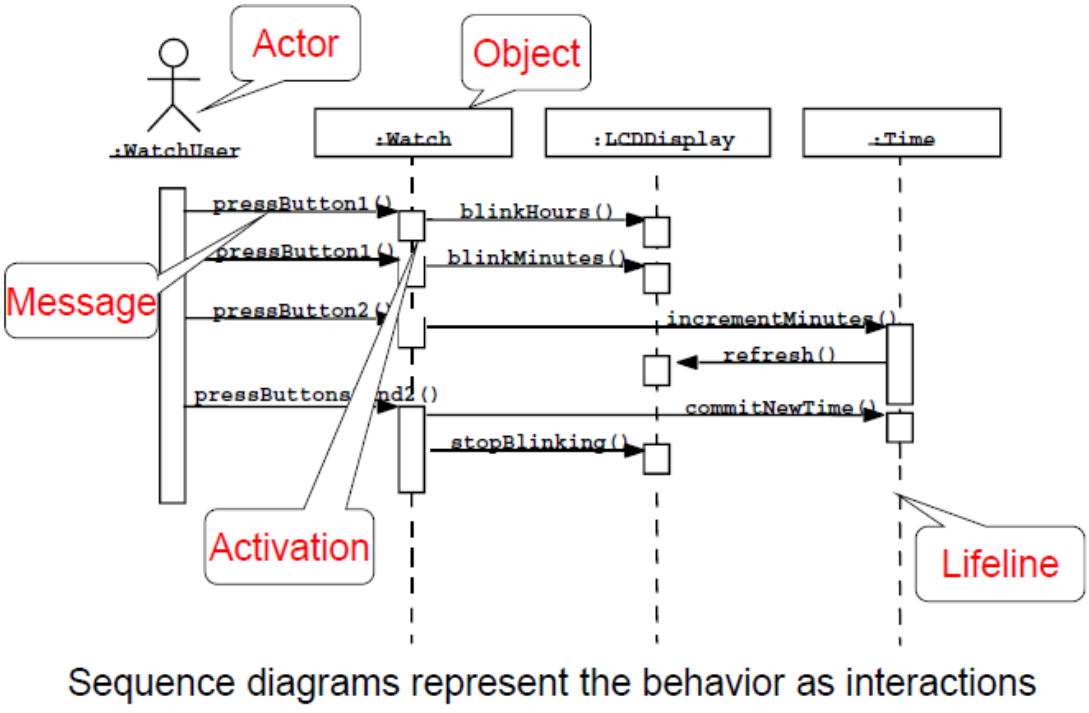
消息—>参与对象，消息是参与对象中的方法

穿越各栏的水平箭头表示从一个对象向另一个对象发送的消息或激励。一个消息的接收触发了一个操作活动。活动用从其他来源的消息出发的垂直矩形来表示，矩形的长度表示操作活跃的时间或者说激活时间。一个操作可以被认为是一个对象向其他对象提供的服务。

若在消息前有个*代表迭代循环发送消息，在消息前有个[布尔表达式]代表一种发送消息的条件。消息指向一个对象的激活（就是对象下面的长方形）则代表创建（creation），在最后的激活上有个✖代表销毁，销毁可以代表一个对象有用的生命的结束。对象不可以接收销毁符号下面的消息。

顺序图可以用于描述一个抽象的序列（即所有可能的交互）或者具体的序列

协作图通过对交互进行编号，表示了消息出现的序列



Sequence diagrams represent the behavior as interactions

布局：第一列——初始化用例的参与者

第二列——边界对象

第三列：管理剩余用例的控制对象

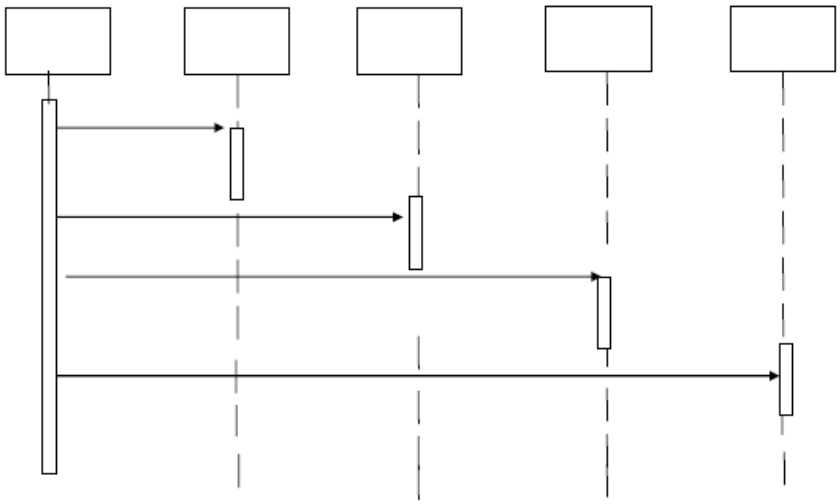
创建：边界对象在用力初始化的时候被创建，控制对象由边界对象创建《create》，之后由控制对象创建其他边界对象并也可与其他控制对象交互。

访问：实体对象被控制对象和边界对象访问，实体对象决不能调用边界对象和控制对象

顺序图也说明了对象的生命周期，在顺序图中的第一个激励之前的已经存在的对象在该图的顶端表示。
构建顺序图中，可能会有新出现的事件，这些事件可以被认为是接受这个事件的类的一个 public 权限的操作

Fork Diagram

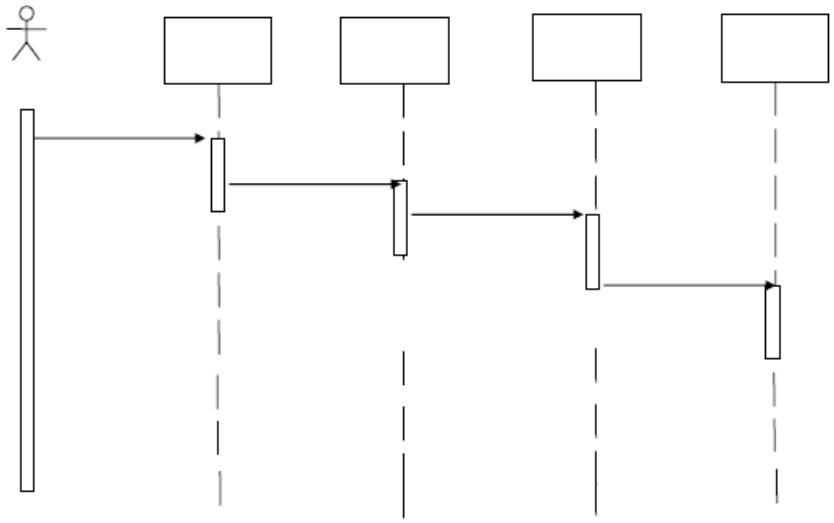
- Much of the dynamic behavior is placed in a single object, usually the control object. It knows all the other objects and often uses them for direct questions and commands.



集中控制式结构，操作可以交换顺序，有新需求时可以插入新操作

Stair Diagram

- The dynamic behavior is distributed. Each object delegates some responsibility to other objects. Each object knows only a few of the other objects and knows which objects can help with a specific behavior.



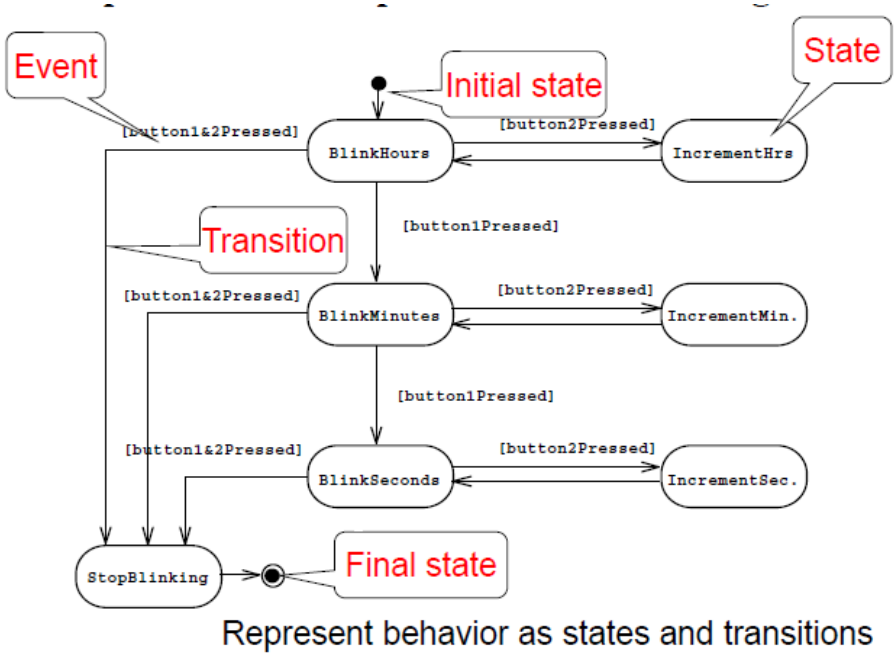
非集中式控制结构，操作之间有很强的联系和顺序性

状态机图：用一组状态以及这些状态之间的迁移描述单个对象的动态行为。转换包括对象未来可以转向的状态和转变条件。

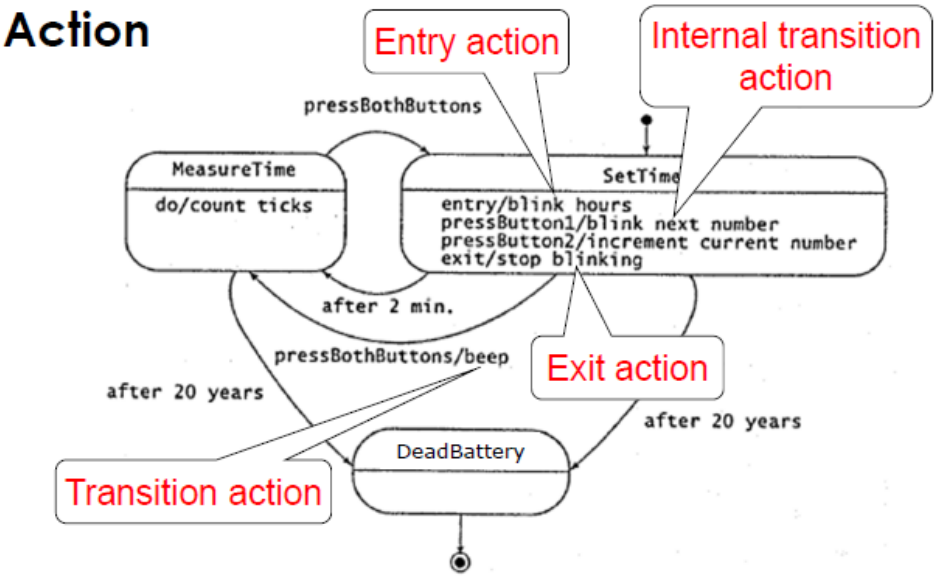
状态：满足一个对象（或系统）某种属性的一种情形/条件（一般一个状态可以通过一些属性值来计算）一个对象的状态量是根据建模需要来确定的

转换/迁移：被事件、条件或时间触发的状态转变

圆角矩形表示状态，小实心黑圆表示起始状态，套有小实心黑圆的圆表示终止状态

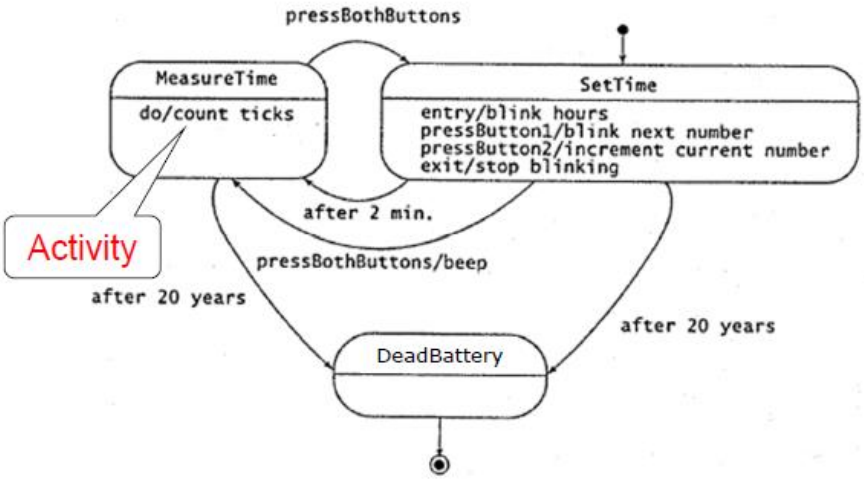


Represent behavior as states and transitions



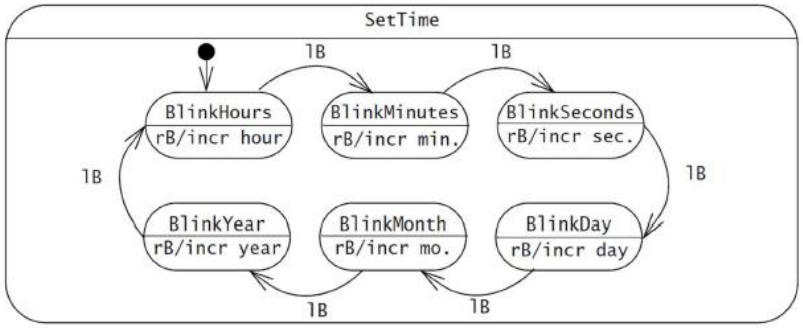
Activity 活动:一个地位等同的对象集合，一个状态可关联到一个活动。活动在状态内部的且可能是长时间，当退出转换被调用的时候 activity 会被打断。活动用 do 做标签

Activity



嵌套状态机：内部转换来减少复杂度

Nested State Machines



活动图：

活动：代表一系列操作执行的建模元素；代表底层行为的排序与协调，说明了一个行为是怎样使用一个或者多个活动的序列以及需要协调这些活动的对象流来实现的

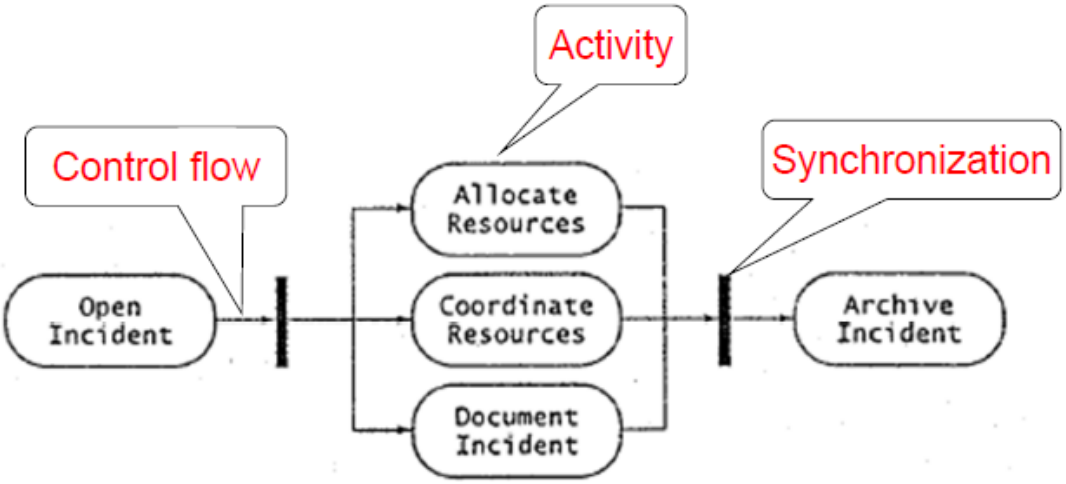
分层次，针对活动来描述系统行为

提供了一个以一组对象行为的任务为中心的四点，用于描述用例之间的顺序约束

其他活动的结束、对象的可用和外部的活动都可以出发活动的执行

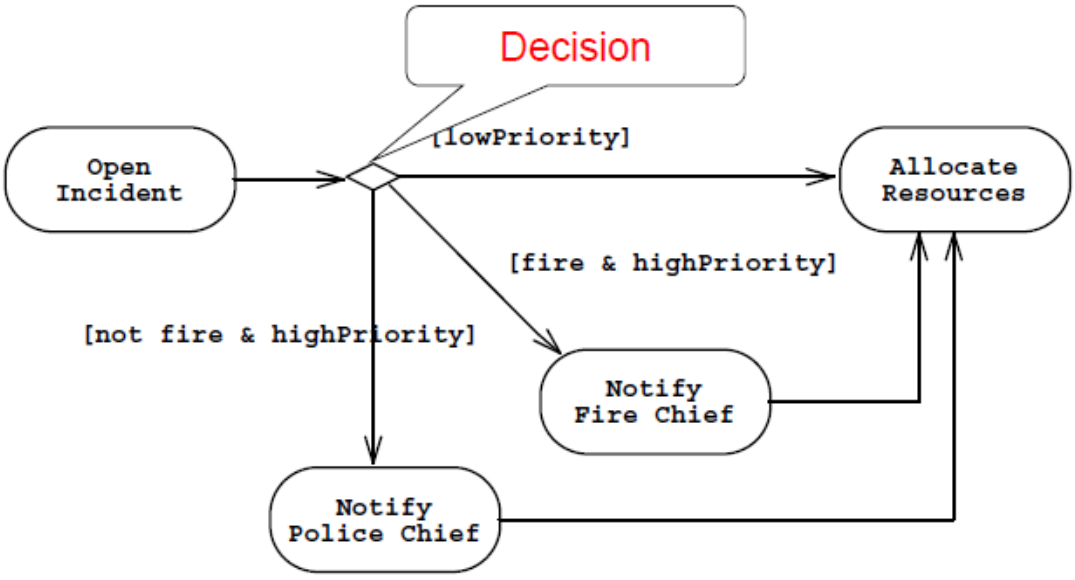
与流程图相似：控制流（操作发生的顺序），数据流（操作中对象的交互）

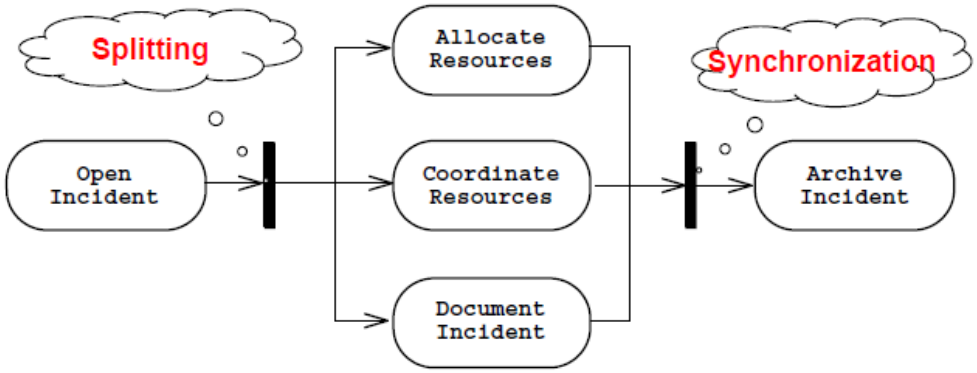
圆角矩形表示动作和活动，活动之间的边表示控制流



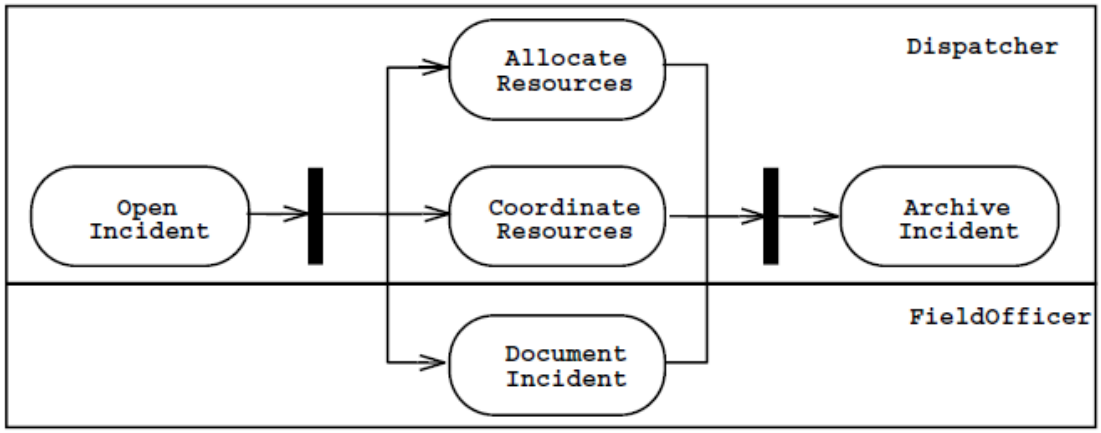
Activity diagrams represent the behavior of system with respect to activities

是状态图的特殊形式
活动图中的状态是活动（“功能”）
活动结束自动跳转，不需要驱动
活动图对于描述系统的工作流程非常有用
活动允许对“决策”（一个菱形）和“并发性”（分叉结点和会和结点）（粗竖线，包括把控制流分成几个线程和同步多个活动之后将控制流合并成一个单一线程）进行建模



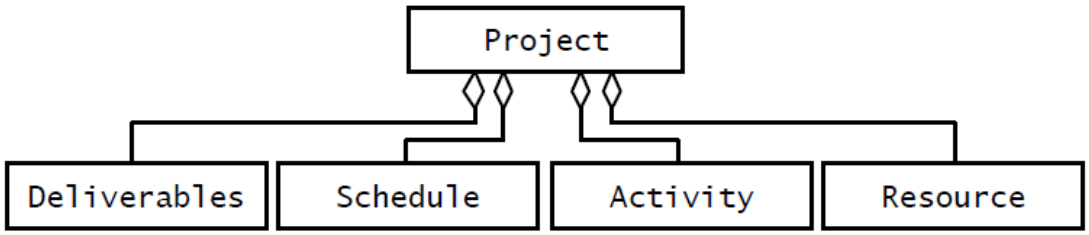


活动还可以被分成若干个泳道（活动划分），代表这些活动是被哪个对象或子系统实现的，迁移可以跨越泳道。

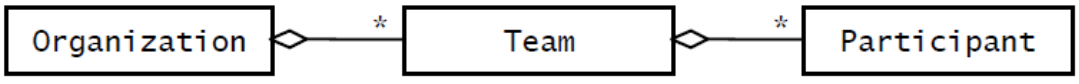


第三章 项目的组织与沟通

一个项目包含四个部分：工作产品、时间表、任务/活动、项目成员



任何项目的组织的一个重要部分是定义参与者之间的关系，以及定义参与者的任务、进度表和工作产品之间的关系



角色：
定义了责任（要去做什么），定义了一个参与者或项目团队要完成的技术和广利任务的集合
区分出四种角色：开发角色、跨功能角色、管理角色、咨询角色
责任被指定到角色身上，角色被指定到人身上

任务：
描述了被管理人员追踪的最小数量级的工作，是部署给一个角色的且已经经过充分定义的工作，包括：

1.角色 2.工作产品 3.开始时间 4.计划工期 5.需要资源

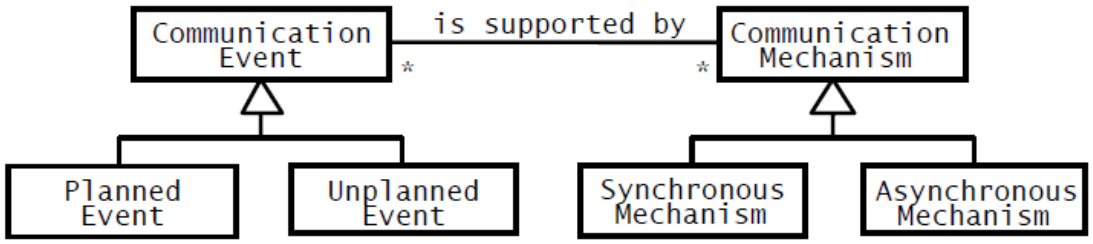
工作产品：
可见的任务产出，交付到用户手中的叫作交付产品

活动：
一系列相关的任务

工作包
描述了完成一项任务和活动所需要的工作规格说明。一个工作包有任务名、任务描述、完成任务所需的资源、对输入（由其他任务产生的工作产品）和输出（正在讨论任务所产生的工作产品）的依存性，以及对其他任务的依存性

进度表：任务到时间的映射，最常用的图——PERT 图和 Gantt 图

Communication Event vs. Mechanism



计划内沟通：表现为时间上的计划点
问题陈述，客户评审，项目评审，同行评审，状况评审，集思广益，发布，事后评审
计划外沟通：
需求的澄清，需求的变化，问题求解
同步沟通机制：
面对面交谈，调查表和正式会面，会议，同步群件
异步沟通机制：
电子邮件，新闻组，万维网，Lotus 公司的 Notes

第四章 需求获取

软件生命周期：
需求获取，需求分析，系统设计，系统细节设计，实现，测试
（用例模型、应用域对象、子系统、解答域对象、源代码、测试实例）
需求说明用自然语言，分析模型用 UML

功能需求：描述系统和环境之间的交互，与实现无关。刻画了系统能干什么事，对应于动作。
非功能需求：描述了不与功能行为直接相关的方面，刻画了系统功能的质量和性质（非功能性需求之间可能是矛盾的）
可用性：是一种用户可以学会的操作、输入准备、解释一个系统或者构件输出的情况。必须是可测量的
有效性：当提出使用要求时，系统或构件的可操作性和可访问性程度；系统正常运行时间的比例

鲁棒性（健壮性）：在不正确输入提供或者压力环境条件的情况下，系统或构件能正确地完成功能的程度
可支持性：关注在进行部署后去改变系统的情况
性能：
（**依赖性**：
可靠性：系统或构件在给定时间内以及指定条件下完成其要求功能的能力
安全性：对环境而言，对缺乏考虑灾难后果的量度
响应时间：对用户输入而言，系统响应的快慢程度
吞吐量：在一个指定时间量内系统可完成的工作量
准确性、可扩展性
可维护性：改变系统以适应外部应用域概念的能力
可移植性：系统或构件从一类软件或硬件环境移植到其他环境的情况
国际化）
限制（伪需求 Pseudo requirements）：由客户或环境提出。关心的是使用许可证。规则和认证方面的问题

需求确认 Requirements Validation：
正确性：是否代表了客户的本意，表示客户需要的系统以及开发者倾向构建的系统
完整性：系统能应用的场景是否全面
相容性/一致性：没有相互矛盾的需求
明确性/无二义性：需求只能有一种解释方式
可实现性：需求能够被实现和交付
可追踪性：每个系统行为都可以追踪到一系列的功能需求

需求获取的不同方式：
1.绿地工程：需求从用户和客户处抽取，没有现成系统存在，开发过程将从草稿开始。
2.再工程：新需求从现存系统中抽取，对一个现存系统的再设计和再实现
3.界面工程：需求从技术使能者或新的市场需求抽取，对一个现存系统的用户界面的再设计

标识参与者：区分参与者和对象 eg.一个数据库系统有时可以是一个参与者，在其他情况下，该数据库又可以是系统的一部分

场景：一种对人们使用计算机系统和应用时的所作所为的陈述性描述。
一个场景是来自单一参与者的、具体的、关注点集中的系统单一特征的非形式化描述。场景将重点放在特定实例和具体事件上。一个场景是一个用例的实例，但场景不能代替用例
As-is 场景：描述了当前情况，常用于再工程项目，通过观察用户并将他们的活动描述成场景，一次来理解当前系统，可以被用来进行正确性和准确性的确认。
Visionary 空场景（还未出现的场景）：描述了未来的系统，常用于绿地工程和再工程项目。开发者和用户两个方面。

分析的关键：从用例出发，找到参与对象

第五章 分析

分析将关注点放在系统模型的产生上，该模型应该是正确的、完全的、一致的、可确认的。在分析中，开发者将关注点放在对从用户处抽取的需求进行结构化和形式化上面。

分析模型（3 个）：**用用例和场景表示的功能模型；用类和对象图表示的分析对象模型；用状态图和顺序图表示的动态模型**

对象建模的步骤：
1.类定义
2.寻找属性
3.寻找方法
4.寻找类之间的关系

实体对象：代表被系统追踪的持久性信息（应用域对象，也被称作“业务对象”）
边界对象：代表用户和系统之间的交互，表示了系统与参与者之间的接口。在每个用例中，每一个参与者至少要和一个边界对象进行交互
控制对象：代表系统所执行的控制任务，协调边界对象和实体对象。在现实世界中，控制对象通常在应用域中没有具体的对应物。控制对象通常从一个用例开始时创建并在该对象退出时终止，从边界对象处收集信息并将这些信息分配给实体

不同的对象类型让我们更好的应对变化

对象类型起源于 Smalltalk(Alan Kay):
Model, View, Controller (MVC)
Model <-> Entity Object 最稳定
View <-> Boundary Object
Controller <-> Control Object 最不稳定

版型机制，模板机制（stereotype mechanism）：
在 UML 中可以引入新的建模元素类型 《String》Name
还可以用图标(icons)和图形符号来定义模板
《entity》 《control》 《boundary》
除了版型外可以在语法层次上利用命名来区分三种对象：控制对象可在名字上加上表示控制意义的 eg.Control；边界对象可以在名字上标识接口特征；实体对象命名则不使用任何前缀。

泛化是建模中的活动，标明了源于多个底层概念的抽象概念；
特化是建模中的活动，标明了一个高层概念的多个特化概念。

在用例中寻找参与对象：
1.选中一个用例，观察事件流
2.做语法分析（Abbott’s Technique），名词就是对象或类的候选项，动词就是操作的候选项（课本表 5-1）
3.定义不同对象的类型

Mapping parts of speech to model components (Abbot's Technique)

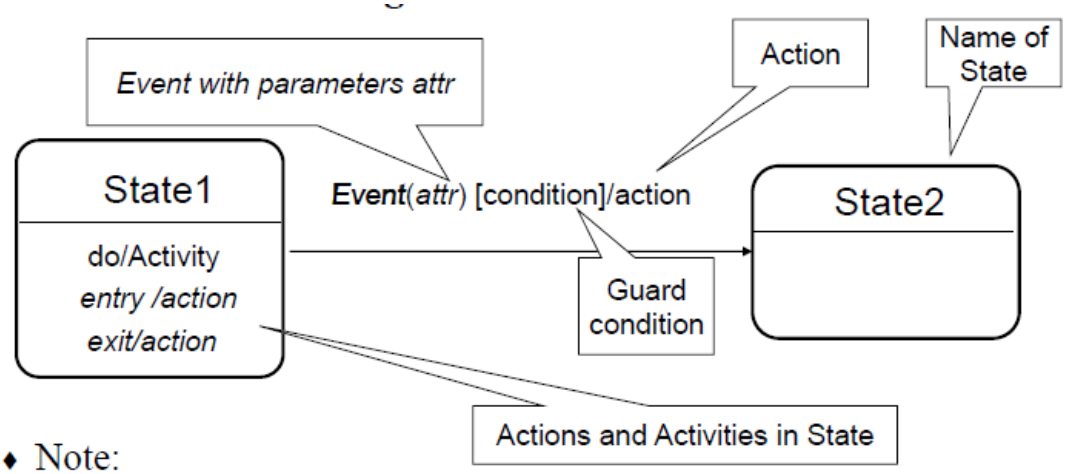
Example	*Part of speech*	*UML model component*
"Monopoly"	Proper noun	object
Toy	Improper noun	class
Buy, recommend	Doing verb	operation
is-a	being verb	inheritance
has an	having verb	aggregation
must be	modal verb	constraint
dangerous	adjective	attribute
enter	transitive verb	operation
depends on	intransitive verb	Constraint, class, association

注意：这只是在用例中寻找对象，如果是整个对象定义过程，则是如下：
1. 从用户和应用域专家处获得一系列场景
2. 从场景中抽取出用例
3. 再进行上面“用例中寻找参与对象”的过程，同时还要进行的是组建 UML 类图

组建类图的过程：
1. 类定义（语法分析，领域专家）
2. 属性和操作的定义（有时在类之前就被发现）
3. 类关系的定义
4. 多重性的定义
5. 任务的定义
6. 继承的定义

动态图为对象模型发现和补充操作

UML 状态图符号（Notation）



- Note:

事件是斜体，条件用[]括起，动作和活动之前都要加/
Notation 是根据 Harel 的工作基础上建立起来的

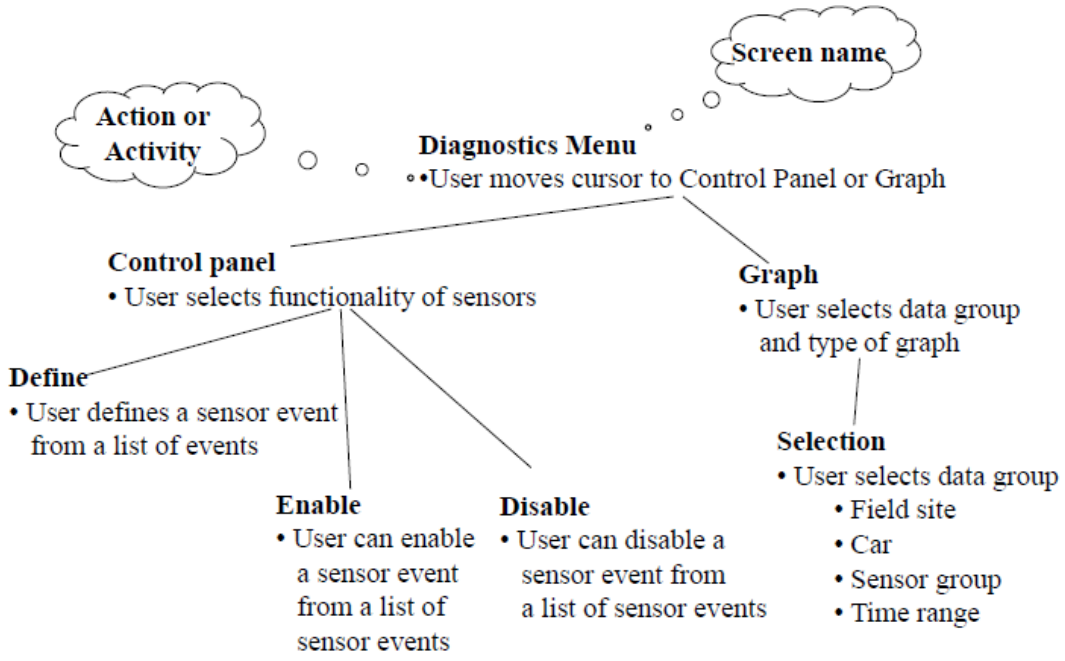
超状态：
其中有很多嵌套子状态
从其他状态进入超状态是进入的第一个子状态
从超状态的任意一个子状态都可以离开

两种类型的并发性：
系统并发性：
对象并发性：

导航路线：
状态图可以被用作用户接口的设计
状态：界面名称
活动/动作：在界面名下面，前面有个“ • ”，一般只有退出动作会写出来
状态转换：退出动作的结果（点击按钮、选择菜单、光标移动）

Requirements Analysis Document Template

1. Introduction
2. Current system
3. Proposed system
   - 3.1 Overview
   - 3.2 Functional requirements
   - 3.3 Nonfunctional requirements
   - 3.4 Constraints (“Pseudo requirements”)
   - 3.5 System models
     - 3.5.1 Scenarios
     - 3.5.2 Use case model
     - 3.5.3 Object model
       - 3.5.3.1 Data dictionary



Model verification检验模型之间转换
model validation把模型结合实际进行比较（更高一层），正确性、完整性、一致性、无歧义性、可实现性
Model Dominant
对象模型占主要：系统有很多类，这些类的状态无关紧要，但是类之间的关系很复杂
动态模型占主要：模型有很多事件：输入、输出、异常、错误等
功能模型占主要：模型执行复杂的转换（计算包含许多步）

RAD (Requirement Analysis Document)

系统设计 System Design：
1. Design Goals 设计目标
2. Subsystem Decomposition 子系统分解
3. Concurrency

4. Hardware/ Software Mapping 硬件/软件映射
5. Data Management 数据管理
6. Global Resource Handling
7. Software Control 控制流
8. Boundary Conditions 边界条件

第六章 系统设计：分解系统

设计目标是对不同利益关注的权衡

子系统：

在UML中以包的形式出现

对象模型中的对象和类是子系统的“种子”

服务：

一组服务是一组有着公共目的相关操作

功能模型中的用例是服务的“种子”

服务在系统设计中被定义

子系统接口：

是服务的优化，可被其他子系统调用的某个子系统的操作集合

包括操作名、操作参数、类型及其返回值

子系统接口在对象设计中被定义

对象设计关注的则是应用程序员接口Application programmer’s interface (API) ：它求精并扩展了子系统的接口

APIs 在实现中定义

层次：

一个子系统向另一个子系统提供服务

某个层次只依赖于它下层的服务，并且对它的上层一无所知

划分：

一个层次可以水平的被分为独立的子系统，即划分

同一层的划分之间可以互相提供服务

划分是低耦合的子系统

P2P 关系（Peer to Peer）

虚拟机：

上下层之间是一种“提供服务”的关系（关于子系统）

封闭架构（模糊）Closed Architecture (Opaque Layering)：上层只能向其下一层调用操作

可维护性，灵活性

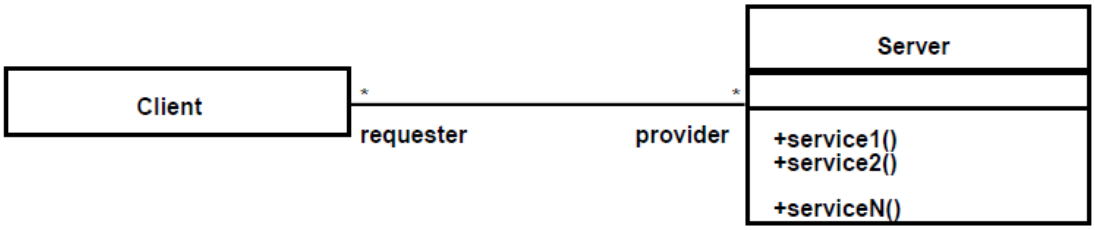
开放结构（透明）Open Architecture (Transparent Layering)：每个虚拟机可以向任何一个下层的虚拟机调用操作

运行更有效率

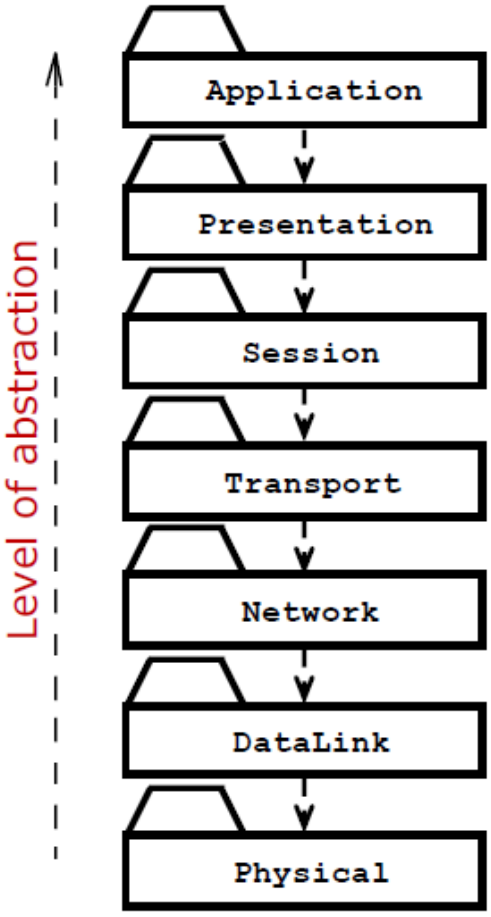
一致性/内聚 Coherence：测量一个子系统中各个类之间的依赖程度

耦合性 Coupling：测量各个子系统之间的依赖程度（若为松散耦合则各子系统相对独立，其中一个系统改变对另一系统影响不大）
一个好的设计需要有高内聚和低耦合度

体系结构风格：子系统分解的模式
（软件体系结构：一个体系结构风格的实例，包含系统分解、全局控制流、边界条件处理和子系统间的通信协议）

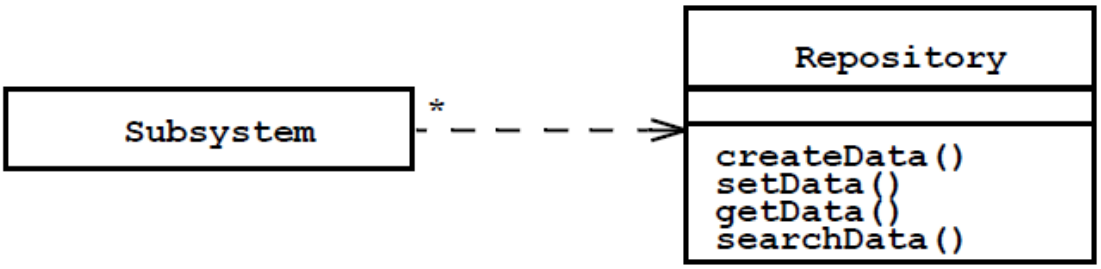


客户端/服务器：
一个或多个服务器向称谓客户端的子系统实例提供服务
客户端知道服务器端的接口
服务器端不知道客户端的接口
常用于数据库
设计目标：服务的可移植性；位置透明度；高性能；可扩展性；灵活性；可靠性
缺点：不能满足 P2P 的要求



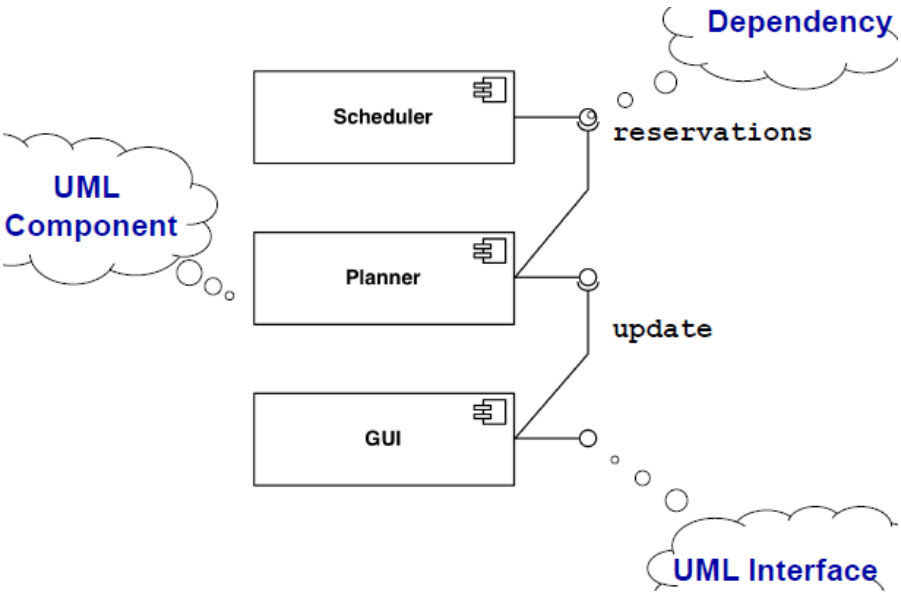
P2P（对等体系风格）：是客户机/服务器的一种泛化。
子系统既可以做客户机又可以做服务器
ISO = International Standard Organization
OSI = Open System Interconnection：
应用层→表示层→会话层→传输层→网络层→数据链路层→物理层

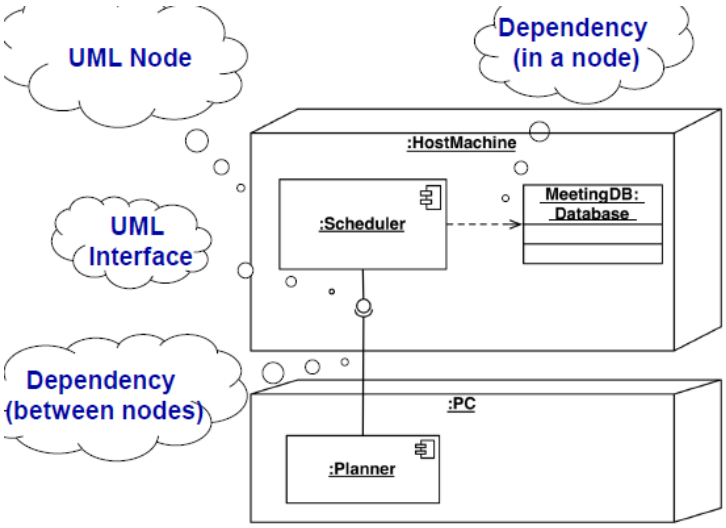
仓库体系结构(Repository Architectural Style)：
子系统访问和修改单一的数据结构，仅仅依赖于称为仓库的集中式数据结构
子系统之间相互独立，子系统之间交互需要通过仓库完成
典型的使用在数据库管理系统中



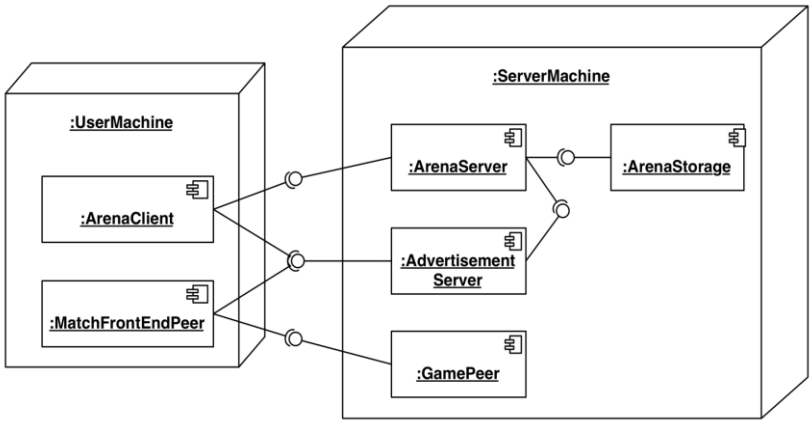
第七章 系统设计：选择设计目标

并发性 Concurrency：线程的同步。
如果两个对象可以在同一时间没有交互的接收事件则为并发。本质上并发对象可以分配给不同的线程的控制。具有相互的专属活动对象可以由单个线程的控制
硬件/软件映射 Hardware/Software Mapping：
控制对象→处理器；实体对象→存储；边界对象→输入/输出设备
UML 构件图：编译时间下子系统之间的设计依赖关系；在模型的组件和组件之间的依赖关系方面被用于系统设计的顶级视图的建模。（各构件的依赖关系用虚线表示，如果用到圆球插座图标则为实线）





UML 部署图：描述系统的执行时间配置；使用节点和连接系统中描述的物理资源
（部署图用实线，但部署图中各构件的依赖关系仍用虚线）

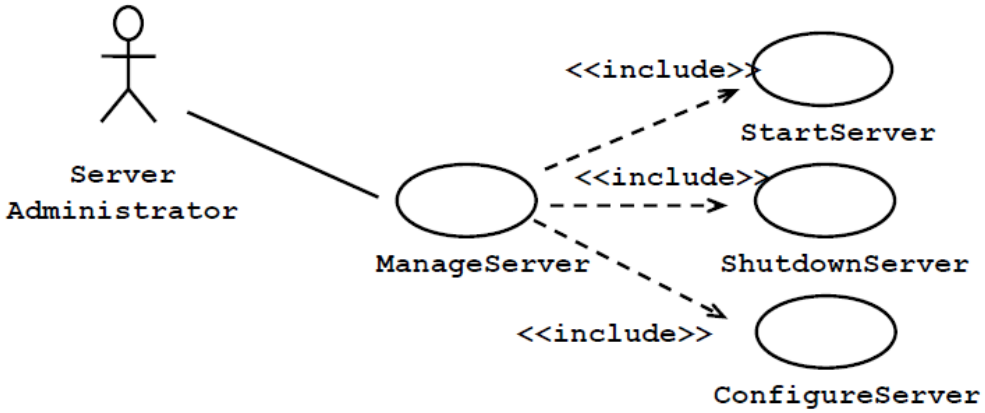


数据管理 Data Management
文件系统：数据只被一个写被很多读。
当有大量的数据、临时数据、只需要保存一段时间、低信息密度的时候应该使用文件系统
数据库：数据被并发的读和写
当需要被多个用户访问、数据跨越多重平台或应用、需要大量基础设施的时候要使用数据库

访问控制：在多用户系统定义访问控制，需要为每个参与者定义一个共享对象，说明该对象哪些操作可以供这些参与者访问。
访问矩阵：（对类的访问控制建模）矩阵行表示系统中的参与者，矩阵列代表要进行控制访问的类，访问矩阵的一个条目元组（class，actor）称为访问权
全局访问表 Global access table：以（参与者，类，操作）的方式显式地展示了矩阵中的每一个元素
访问控制表 Access control list (focusing classes)：以（参与者，操作（针对于某个类））形式的列表关联到每个将被访问的类
能力 Capability (focusing actors)：以（类，操作（针对于某个参与者））的元组关联到某个参与者

Software Control
控制流是指某个系统中的动作序列
两个主要的设计选择:
1.选择隐式控制 (例如逻辑编程)
2.选择显式控制 (集中或分散)
集中的控制：（一个控制对象或一个子系统控制全体）
过程驱动：控制驻留在程序代码中，多半应用于遗留系统和用过程式语言编写的系统中。
事件驱动：控制驻留在一个通过回调的调度程序调用函数。
分散的控制：（有多个控制对象）
控件驻留在多个独立对象
通过映射对象在不同处理器上和增加通信开销来提速

Boundary Conditions
边界条件：系统如何启动、如何初始化及如何关闭
初始化：该系统是从非初始化状态到稳定状态
终止：清理资源，并在终止之后通知其他系统
Failure：可能出现的故障: Bug、 错误、 外部问题
良好的制度设计能预见致命故障并提供机制来处理
边界用例 boundary use case 与管理员相对应



第八章 对象设计：复用模式解决方法

对象设计：是将详细信息添加到需求分析和实施决策的过程，对需求分析进行补充并作出实现决定
对象设计包括四组活动：复用、接口规格说明、重构、优化
应用对象：与系统相关的对象（应用域专家和终端用户定义），描述了与系统有关的域概念
求解对象：与应用域没有对应（开发人员定义），表示在应用域中没有对应物的构件

继承：用新的操作或重新操作来扩展基类
1. 描述分类：需求分析中使用
2. 接口定义：对象设计中使用，定义所有对象的签名

实现继承（类继承）：以复用代码为唯一目的来使用继承，父类的操作已经被实现
定义继承（子类型化）：只是继承被声明的操作，并不实现
授权：一种可供选择的实现继承的方法，抓取一个操作发送到另一个对象上
灵活性：任何对象可以在运行时被另一个取代(只要它具有相同的类型)
对象需要封装

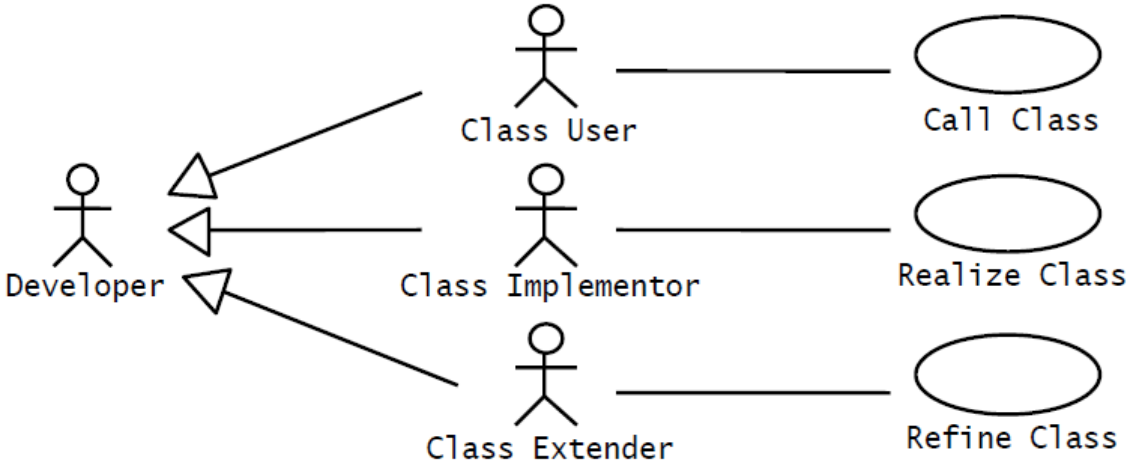
框架：一个可部分复用的应用程序，主动的，影响控制流
（白盒框架依靠继承和动态绑定来获得扩展性；黑盒框架通过定义可以插入到框架之中的构件接口支持扩展性。黑盒框架比白盒框架使用起来容易些，因为黑盒框架采用的是授权而不是继承，但是黑盒框架开发难度更大，需要接口的定义和预见很大范围内的潜在用例的钩子）
可重用性利用了的应用域知识和经验
可扩展性是通过提供钩子方法，由应用程序的重写来扩展框架。
类库：被动的，在控制流上没有约束，较少的域特定，提供重用范围较小，通用性功能。
框架&类库：通常在同一个系统中同时使用类库和框架
框架&构件：构件是可组合起来构成完整应用的类的自含实例，就复用而言，构件是定义了一组聚集在一起的操作集合的黑盒。和框架相比，构件之间的耦合联系不是那么紧密，可以在二进制代码级别上进行复用。一般使用框架来简化基础设施和中间件的开发，使用构件来简化终端用户应用软件的开发工作
设计模式&框架：框架注重于具体设计、算法以及在某个特定编程语言实现下的复用，模式注重于抽象设计的复用和小型协同操作类集合的复用。框架注重一个特定的应用域，设计模式是一组成框架构造块出现的。

第九章 对象设计：说明接口

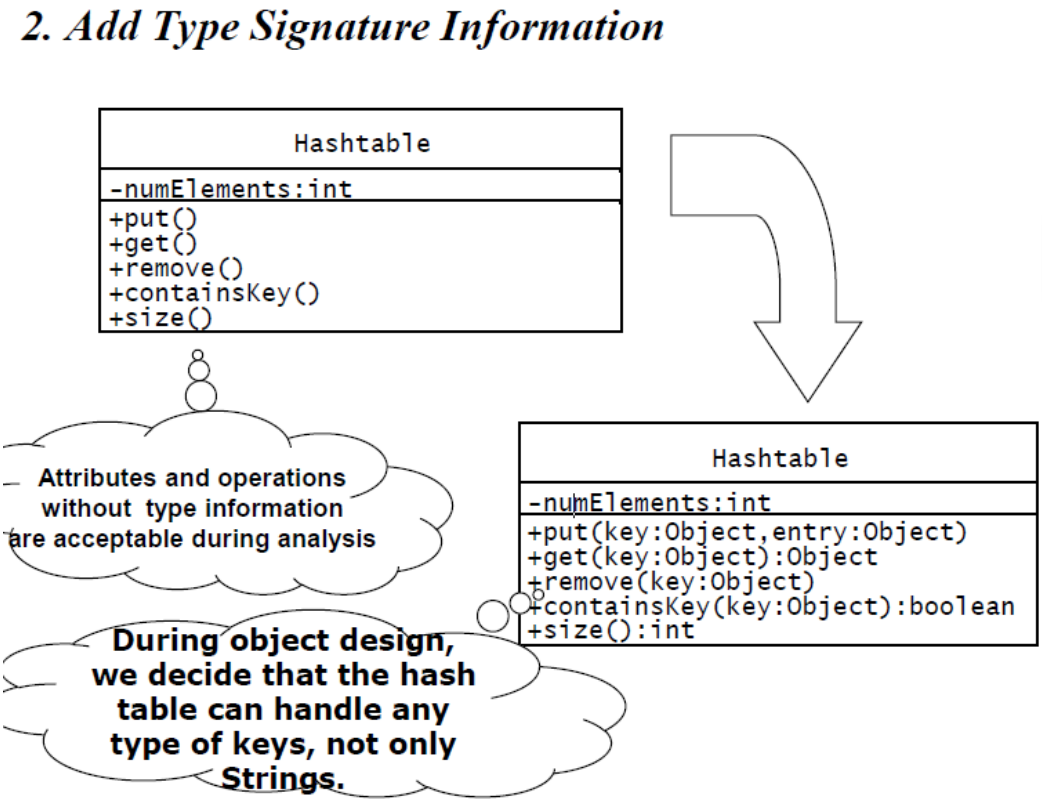
类用户在其他类的实现过程中，调用由正在考虑类所提供的操作（这个类称为客户类）。对其而言，接口规格说明根据类提供的服务和对客户类所做的假设，揭示了类的边界

类实现者负责实现正在考虑中的类，设计内部的数据结构并为每个公共操作实现代码。对其而言，接口规格说明是一个工作分配

类扩展者开发正在考虑中的类的指定扩展，也可以调用其感兴趣类所提供的操作，关注同一个服务的指定版本。对其而言，接口规格说明既说明了当前的类行为有说明了指定类提供服务的所有约束



添加类型签名信息：（输入/输出操作）在对象设计期间利用 Hash 表来处理任何类型的关键词



添加契约：

不变式：一个对该类的所有实例都为真的谓词，和类或接口有关的约束，通常用来说明类属性之间的一致性约束

前置条件：一个在调用一个操作之前必须为真的谓词。与某个指定的操作相关联，是类用户用来说明在调用一个操作之前必须满足的约束

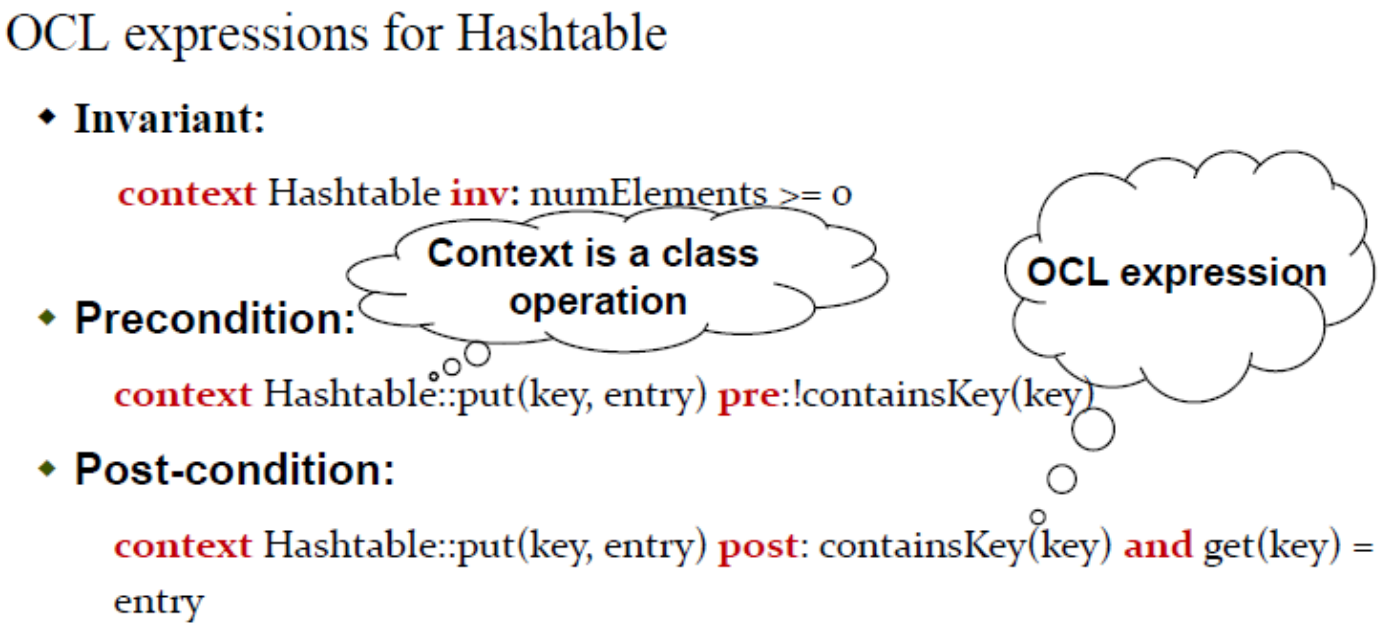
后置条件：一个在调用一个操作之后的必须为真的谓词，与某个指定的操作相关联，是类实现者用来说明在调用一个操作之后必须满足的约束

OCL（对象约束语言）——用自然语言来描述一个约束

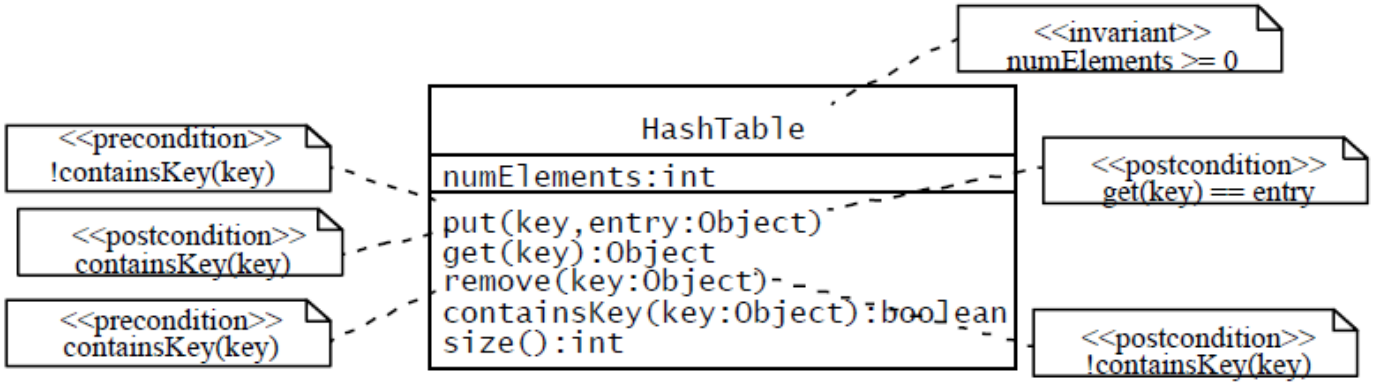
OCL 是一种允许在单个模型元素和模型元组上对约束进行形式化说明的语言

一个约束被表述为一个返回值为真或为假的布尔表达式，可以被描述成通过依赖关系关联到一个受约束的 UML 元素的说明，用文本形式进行描述。

关键字 context 标明该表达式所适用的实体，关键字后面会是如下关键字 inv,pre,post 中的一个，分别对应 UML 版型《invariant》《precondition》《postcondition》，然后就是实际的 OCL 表达式



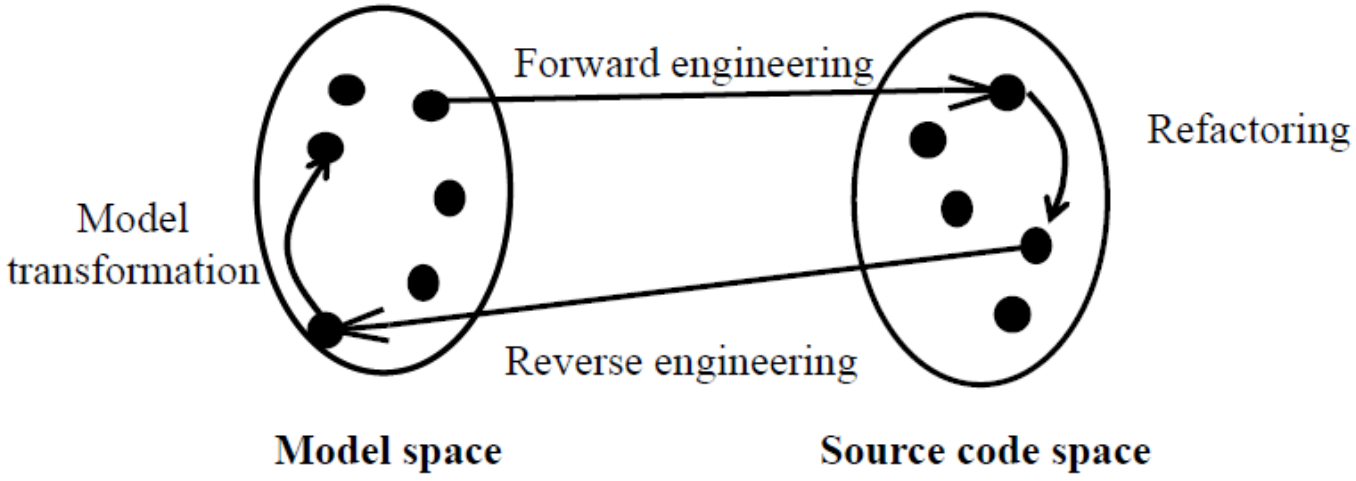
Local 本地属性——该约束包括对于研究的类来说是本地的属性



第十章 将模型映射到代码

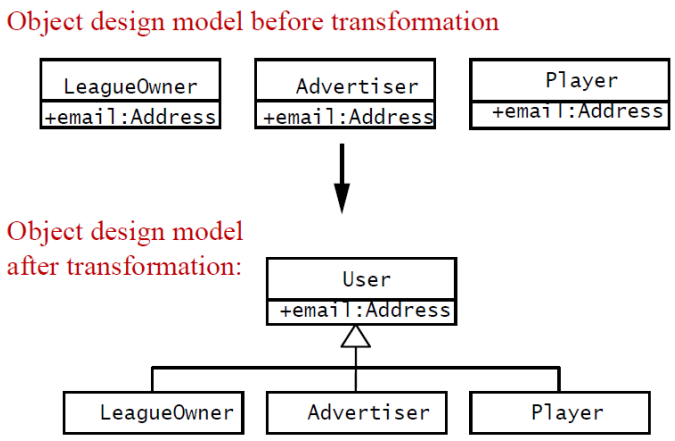
转换：优化、实现关联、将契约映射到异常上、将类模型映射到某一存储模式上

四种类型的转换：



模型转换 Model transformation：作用于某一模型上，以产生另一模型。对象模型转换的目的是简化或优化原始模型，使该模型与规格说明书中的需求一致。转换可能增加删除或者重命名类、操作、关联或属性，也可能往模型中增加信息或删除信息（eg.通过创建父类消除冗余属性）

重构 Refactoring：对源代码的转换，在不影响系统行为的前提下，提高源代码的可读性和可修改性。通过关注类的特定字段或方法来达到改进工作系统设计的目的




```
public class Player {
  private String email;
  //...
}
public class LeagueOwner
  {
  private String eMail;
  //...
}
public class Advertiser {
  private String
  email_address;
  //...
}
```

```
public class User {
  private String email;
}
public class Player
  extends User {
  //...
}
public class LeagueOwner
  extends User {
  //...
}
public class Advertiser
  extends User {
  //...
}
```

正向工程 Forward engineering：应用于对象元素集合上，并生成一组对应的源代码语句集合，其目的是保持对象设计模型和代码间的高度一致，并减少在实现期间引入的错误数，减少了实现上需做的努力。（模型→代码）

逆向工程 Reverse engineering：应用于源代码元素集合并产生模型元素集合，用于为现存系统重新构建模型，本质上是正向工程的反向转换（代码→模型）

映射活动：优化对象设计模型，将关联映射到集合，将契约映射到异常，将对象模型映射到持久存储模式

英文全称：

UML：unified modeling language

OMT:object modeling technology

OCL (Object Constraint Language)

OOAD (Object Oriented Analysis And Design)

FRIEND: First Responder Interactive Emergency Navigational Database

RAD ：Requirement Analysis Document

ODD：The Object Design Document

CMM：Capability Maturity Model for Software

LOC：length of code (代码行)

COCOMO：constructive cost model

API: Application programmer’s interface

OSI ： Open System Interconnection

ISO ： International Standard Organization

OOSE：object-oriented software engineering

OMT (James Rumbaugh)

OOSE (Ivar Jacobson)

Booch (Grady Booch)