软件需求分析与系统设计
- Chapter 1 软件过程
  - 1、软件开发的本质
    - 1.1 软件开发的不变事实
    - 1.2 软件开发的意外事件
    - 1.3 开发&集成
  - 2、系统规划
  - 3、三级管理系统
    - 3.1 事务处理系统
    - 3.2 分析处理系统
    - 3.3 知识处理系统
  - 4、软件开发生命周期
    - 4.1 开发方法
    - 4.2 生命周期的阶段
    - 4.3 跨越生命周期的活动
  - 5、开发模型与方法
- Chapter 2 需求确定
  - 2.1、从业务过程到解决方案的构想
    - 2.1.1 业务过程建模
    - 2.1.2 解决方案的构想
  - 2.2、需求引导
    - 2.2.1 需求与需求引导
    - 2.2.2 需求引导的传统方法
    - 2.2.3 需求引导的现代方法
  - 2.3、需求协商和确认
  - 2.4、需求管理
  - 2.5、需求业务模型
    - 2.5.1 系统范围模型
    - 2.5.2 系统用例模型
    - 2.5.3 业务词汇表(Business Glossary)
    - 2.5.4 业务类模型
  - 2.6、需求文档(Requirements document)
- Chapter 3  可视化建模基础
  - 3.1 用例视图(The use case view): The most important behavioral modeling techniques
  - 3.2 活动视图(The activity view)
  - 3.3 结构视图
  - 3.4 交互视图
  - 3.5 状态机视图(The state machine view)
  - 3.6 实现视图(The implementation view)
- Chapter 4 需求规格说明 (Requirements Specification)
  - 4.1 体系结构优先权 (Architectural prerogatives)
  - 4.2 状态规格说明
  - 4.3 行为规格说明
  - 4.4 状态变化规格说明
- Chapter 5 从分析到建模 (Moving from Analysis to Design)
  - 5.1 高级类建模 (Advanced class modeling)
  - 5.2 高级泛化与继承建模 (Advanced generalization and inheritance modeling)
  - 5.3 高级聚合与委托建模 (Advanced aggregation and delegation modeling)
  - 5.4 高级交互建模 (Advanced interaction modeling)
- Chapter 6 系统体系结构与程序设计 (System Architecture and Program Design)
  - 6.1 分布式物理体系结构 (Distributed Physical Architecture)
  - 6.2 多层逻辑体系结构 (Multilayer logical architecture)
  - 6.3 体系结构建模 (Architecture modeling)
  - 6.4 程序设计与复用原则 (Principles of program design and reuse)
  - 6.5 协作建模 (Collaboration modeling)

Chapter 7 图形用户界面设计(Graphical User Interface Design)
7.1 GUI设计原则(Principles for GUI Design)
7.2 桌面GUI设计(Desktop GUI design)
7.3 Web GUI设计(Web GUI design)
7.4 GUI导航建模(GUI navigation Modeling)
Chapter 8 持久性与数据库设计
8.1 业务对象和持久性 (Business objects and persistence)
8.2 关系数据库模型 (Relational database model)
8.3 对象-关系映射 (Object-relational mapping)
8.4 管理持久化对象的模式 (Patterns for managing persistent objects)
8.5 设计数据库访问和事务 (Designing database access transactioins)
Chapter 9 质量与变更管理(Quality and Change Management)
9.1 质量管理(Quality management)
9.2 变更管理(Change management)

# 软件需求分析与系统设计

## Chapter 1 软件过程

### 1、软件开发的本质

从软件固有的问题角度定义：软件是一种创造性行为开发的产品`product of a creative act`，而不是制造业重复性行为的结果`result of a repetitive act of manufacturing`。

#### 1.1 软件开发的不变事实

软件的本质困难定义出了软件开发的**不变事实**`software development invariants`，其主要包括：

- 复杂性`complexity`：软件规模的函数，以及组成软件产品的构件之间相互依存关系的函数；
- 一致性`conformity`：应用软件必须与其依赖的软硬件平台一致，也必须与现有的信息系统相一致，以便于集成；
- 可变性`changeability`：由于业务过程和需求是不断变化的，所以在开发应用软件时也必须能够容纳这些变化；
- 不可见性`invisibility`：尽管应用软件提供了可见的输出，但是负责输出的代码通常隐藏在"不可见"的程序语句、二进制代码库和周边系统软件中。

#### 1.2 软件开发的意外事件

软件开发的不变事实定义了软件生产的本质问题，理解了这些不变事实，就可以处理软件工程中的**意外事件**`accidental difficulties`。由于软件生产实践带来的困难，便可以通过人为的干涉来解决。

意外事件不会增加软件产品的复杂性，也不会降低软件产品的**可支持性**（适应性）`adaptiveness`的潜在匮乏。可支持性由三个系统特征组成的集合来定义，其包括软件的**可理解性**`understandability`、**可维护性**`maintainability`以及**可伸缩性**`scalability/extensibility`。

软件开发意外事件与三个因素相关：

- 利益相关者`stakeholders`：指在软件项目中存在利害关系的人、任何受到系统影响或者影响系统的人。其主要包括：
  - 客户`customers`：用户和系统用户；
  - 开发者`developers`：分析师、设计者、程序员等。

  信息系统是社会系统，它们是由开发者为客户开发的。软件项目的成功由社会因素决定（技术次要）。

在典型的情况下，软件失败的原因可以追溯到利益相关者。

- 过程`process`：软件过程定义了在软件生产和维护中所使用的活动和组织程序。过程的目标是在开发中管理和改进协作并维持团队，交付符合质量要求的产品。

  一个过程模型：

  - 声明了所执行活动的次序；
  - 详细说明要交付哪些人工制品以及交付时间；
  - 将活动和人工制品分配给开发者；
  - 提供用来监控项目进展、评估结果和规划未来项目的标准。

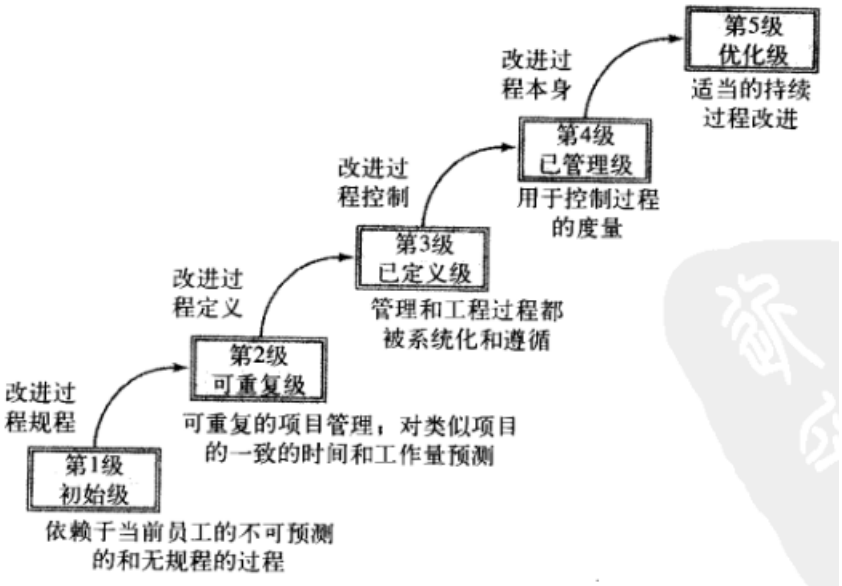
  对于不同的建模和语言，软件过程不易进行标准化，每个组织需要开发属于自己的过程模型。

  - 迭代和增量模型`iterative and incremental model`：现代软件开发的过程总是**迭代和增量的**。系统模型通过分析、设计和实现逐步完善。在连续的**迭代**中增加细节，必要时引入变更和改进（**非功能性的**），而软件模块的**增量**版本则保持了用户的满意度，并未尚在开发中的模块提供反馈。迭代和增量过程有多种变体，例如：

    - 螺旋模型`the spiral model`；
    - `Rational`统一过程`the Rational Unified Process,RUP`；
    - 模型驱动的体系结构`Model Driven Architecture,MDA`；
    - 敏捷开发过程`the agile development process`；
    - 面向方面的软件开发`aspect-oriented software development`。

    对于迭代和增量的开发过程，必须进行合理的规划和控制，且符合预定义的体系结构设计框架（元架构）。

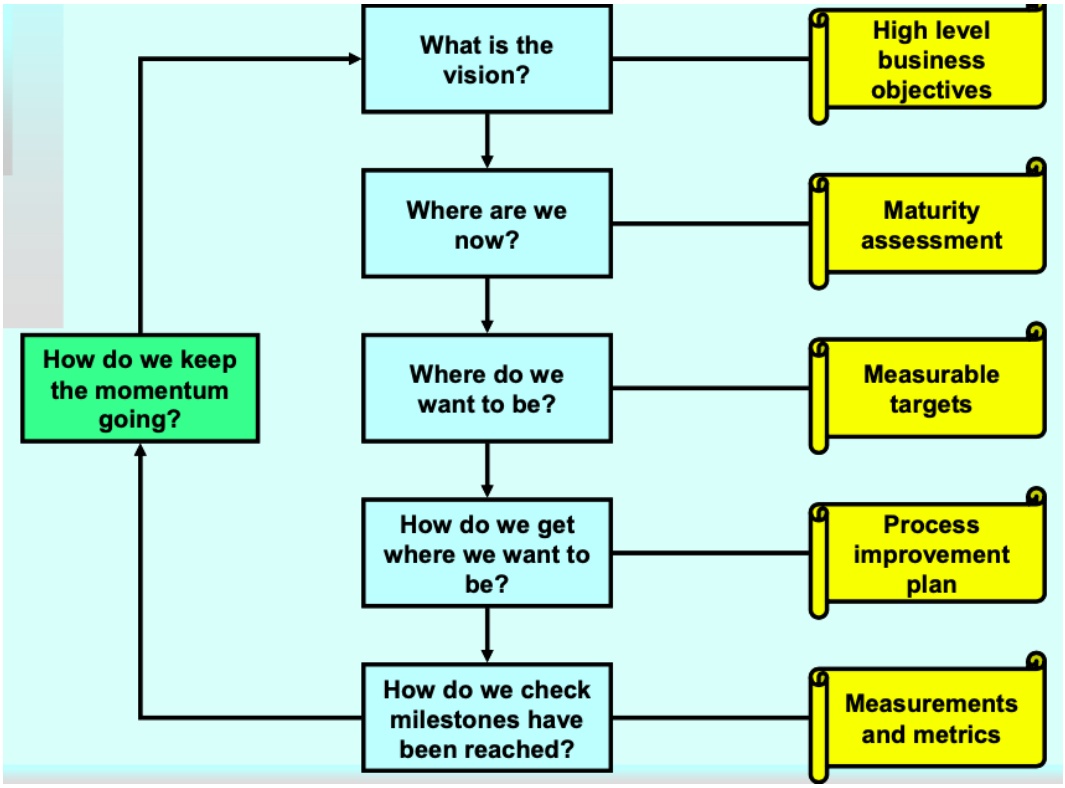
  - 能力成熟度模型`capacity maturity model,CMM`：对于每个组织，软件开发的一个巨大挑战就是改进其开发过程，而能力成熟度模型则是一种用于进行过程评估和改进的流行方法。从本质上讲，`CMM`是一种面向IT组织的问卷调查表：

  - `ISO 9000`系列质量标准：`ISO`标准应用于质量管理和过程，以生产优质产品。这个标准是通用的，适用于任何业务类型。而该标准的主要前提`main premise`是：

    - 如果过程是正确的，那么过程的结果也将是正确的；
    - `ISO`标准不强制执行或具体指定过程，标准提供了必须完成什么的模型，而不是必须怎样执行活动的模型。

  - IT基础设施框架`ITIL`：`ITIL`是最为广泛使用的用于IT服务管理的最佳实践的框架。`ITIL`就是要高效利用`4P`：人`people`、过程`process`、产品`product`和合作伙伴`partner`。下图介绍了作为一项持续性的服务改进方案`continuous service improvement programme,CSIP`、用来实现解决方案管理的`ITIL`方法：




  - 控制目标信息和相关技术 `COBIT`：`COBIT` 是一个服从框架，并致力于处理解决方案管理的控制方面。`COBIT` 框架是具有规定性的，它将相关的IT工作组织到4个领域：
    - 规划与组织 `Plan and Organize`；
    - 获取与实现 `Acquire and Implement`；
    - 交付与支持 `Deliver and Support`；
    - 监控 `Monitor`。

    > `ITIL`、`CMM` 和 `ISO 9000` 是**过程标准**，规定了组织在管理过程方面的要求，以提供优质的产品或服务。
    >
    > `COBIT` 则是一个**产品标准**，侧重于一个组织需要做什么，而不是需要如何去做。

- 建模 `modeling`：软件建模即软件开发活动，这些活动被视为软件人工制品的建模。对人工制品的建模必须进行**沟通** `communicated` 和**文档化** `documented`。沟通依赖于语言，而文档化则依赖于工具。
  - 统一建模语言 `Unified Modeling Language,UML`：一种通用的、可视化的建模语言，用于对软件系统的人工制品进行详细说明、可视化、构造和文档化。

    `UML` 独立于：
    - 任何软件开发过程，采用 `UML` 的过程必须支持通过 `OOP` 技术来开发软件；
    - 实现技术，只要该技术支持 `OOP` 即可，这使 `UML` 在支持开发生命周期的详细设计阶段有些许不足。

    `UML` 模型可以分为三组：
    - 状态模型 `state models`：描述静态数据结构；
    - 行为模型 `behavior models`：描述对象协作 `collaboration`；
    - 状态变更模型 `state change models`：描述随着时间推移，系统所允许的状态。
  - 计辅软件工程 `Computer-assisted Software Engineering,CASE` 与过程改进 `process improve`：`CASE` 工具使在中央存储库中实现模型的存储和检索称为可能，并在计算机屏幕上进行模型的图形和文字操作。

    过程改进远不只是引入新的方法和技术。事实上，在低成熟度的过程中，为组织引入新的方法技术更多的情况是弊大于利。

集成的`integrated CASE`工具可以允许多个开发人员协作和共享设计信息，以便生成新的设计工件。该工具**将过程强加给开发团队**，因为"不成熟"的组织不会遵循该过程。但是`CASE`工具始终会给各个开发人员带来个人生产力和质量上的改进。

#### 1.3 开发&集成

**集成开发**`integration development`相对于从头开始的软件开发而言，使用了相同的迭代声明周期方法和相同的软件生产模型，差别就在于其所强调的集成层次和使能技术。

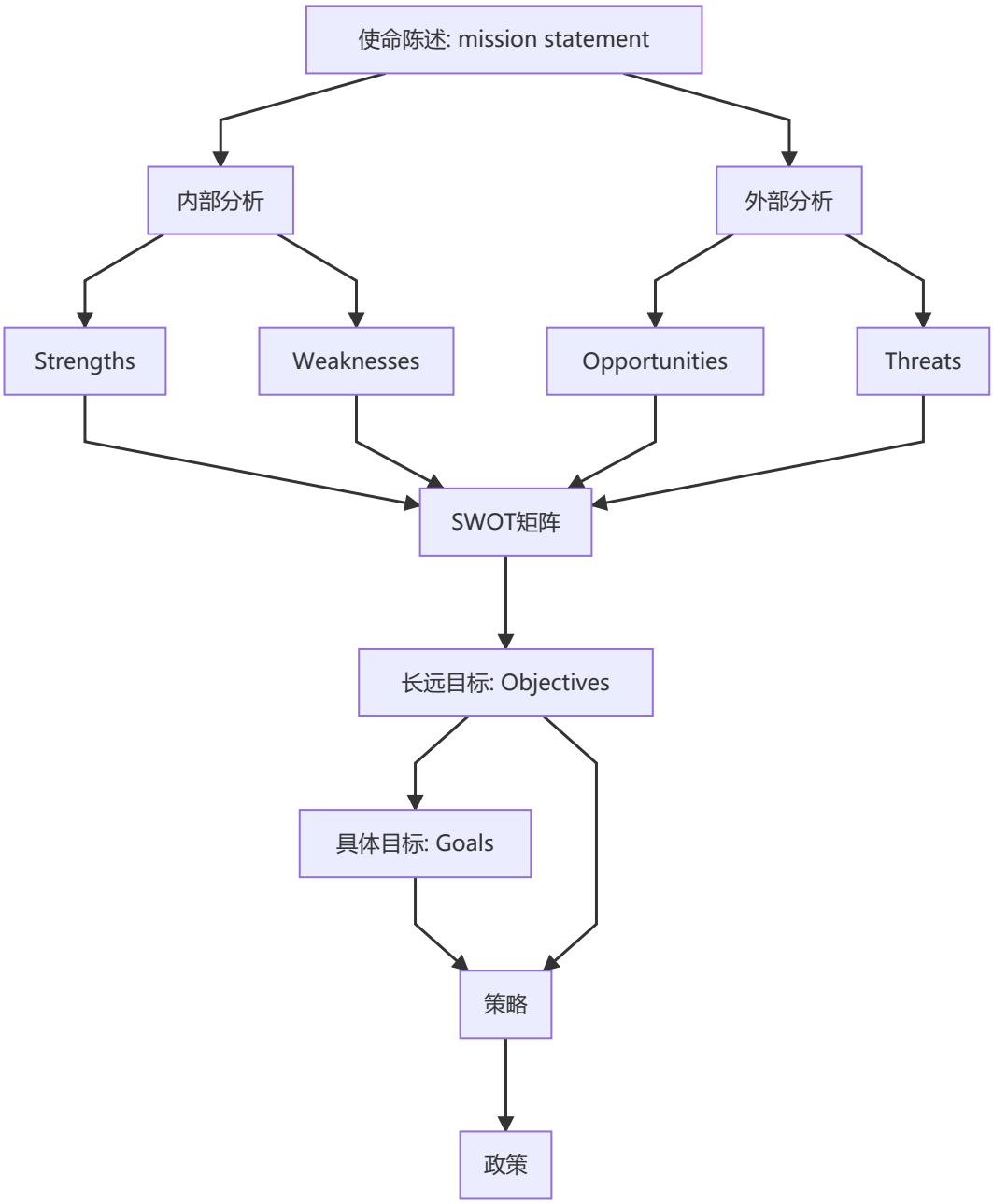
集成方法分为三大类型：

- 面向信息/面向门户的集成`information/portal-oriented`：在数据库或应用程序接口`API`层次上的集成，使信息具体化以供其他程序使用。而面向门户的集成则是特殊的面向信息集成，它将多个软件系统的信息具体化到一个共同的用户界面，例如`web`浏览器；
- 面向接口的集成`interface-oriented`：将应用接口连接在一起；
- 面向过程的集成`process-oriented`：将应用系统连接在一起；

### 2、系统规划

在进行信息系统开发时，必须对信息系统项目进行规划。系统规划`system planning`可以通过多种方式来判定：

- `SWOT`：`SWOT`分析包括优势`strength`、劣势`weakness`、机会`opportunity`、威胁`threat`四个部分。这种方法以调整组织优势、劣势、机会和威胁的方式来进行`IS`开发项目的识别、分类、排序和选择，这是一种从确定组织使命开始的、**自顶向下**的方法：




- 价值链模型`value chain model,VCM`：`VCM`通过分析组织中完整的活动链来评估竞争优势。对于链式模型，一个链接弱将导致整个链的崩溃，而`VCM`的目的就是理解那种价值链配置将产生最大的竞争优势。

  在最初的`VCM`中，组织的职能分为：

  - 基本活动`primary activities`：为最终产品创造或增加价值，其主要包含一下五个过程：
    - 内部物流；
    - 操作；
    - 外部物流；
    - 销售和市场；
    - 服务。
  - 支持活动`support activities`：不直接增加产品价值，但不可或缺。其包括：
    - 行政管理和基础架构；
    - 人力资源管理；
    - 研发；
    - `IS`研发。
- 业务过程重组`business process re-engineering,BPR`：当组织必须被彻底改造，抛弃正在使用的功能分解、分层结构和操作原则时，便可以使用`BPR`方法。

    - 大多数现代组织被编入关注功能、产品或区域的**纵向单元** `vertical units`；
    - **业务过程** `business process` 被定义为“采用一种或多种输入，并创造对客户有价值的输出的活动集合”，没有任何人对这样的业务过程负责；
    - 过程企业和传统组织之间最明显的区别是**过程所有者** `process owner` 的存在。

  `BPR` 的主要目的是在组织中从根本上重新设计业务过程 `process redesign`。在组织中实现 `BPR` 的主要困难为向传统的纵向管理结构中嵌入一个横向过程 `horizontal process`。`BPR` 首先需要改变组织，目的是使作为基础组织单元的**开发团队**成为中心，这些团队负责一个或多个端到端的业务过程。

- 信息系统体系架构 `information system architecture,ISA`：`ISA` 是一个**自底向上**的方法，它为能适应各种业务策略的 `IS` 解决方案提供了一种中立的系统结构框架，且其不包含系统规划方法学 `system planning methodology`；

  `ISA` 框架被描述为一个 `5(1-5)*6(A-F)` 的表格。

  行表示用于复杂工程产品构建的不同视角 `perspectives`：

    - 规划者 `Planners`：确定系统范围；
    - 所有者 `Owner`：定义企业概念模型；
    - 设计者 `Designer`：详细说明系统物理模型；
    - 建造者 `Builders`：提供详细的技术解决方案；
    - 承包者 `Subcontractors`：提供系统构件。

  列表示每个参与者所从事的六种不同的描述 `descriptions` 或体系结构模型 `architectural model`。

以上方法都有一个共同的特性：**关注效果而不关注效率**；

**3、三级管理系统**

一个组织具有三级管理：策略级 `strategic`、战术级 `tactical`、操作级 `operational`。

*Systems for three management levels*

| *Level of decision making* | *Focus of decision making* | *Typical IS applications* | *Typical IT solutions* | *Pivotal concept* |
|---|---|---|---|---|
| **Strategic** *(executive and senior management levels)* | *Strategies in support of organizational long-term objectives* | *Market and sales analysis, Product planning, Performance evaluation* | *Data mining, Knowledge management* | **Knowledge** |
| **Tactical** *(line management level)* | *Policies in support of short-term goals and resource allocation* | *Budget analysis, Salary forecasting, Inventory scheduling, Customer service* | *Data warehouse, Analytical processing, Spreadsheets* | **Information** |
| **Operational** *(operative management level)* | *Day-to-day staff activities and production support* | *Payroll, Invoicing, Purchasing, Accounting* | *Database, Transactional processing, Application generators* | **Data** |

其中，**为组织提供最大回报的 `IS` 应用和 `IT` 解决方案处于策略级**，同时这些方案也是最难实现的方案。

另一方面，支持操作级的系统使用数据库技术，是非常常规的，无法提供竞争优势但又不可或缺。管理得当的组织能够拥有一套策略级 `IS` 应用系统，**为高级别决策及战术决策而存储和检索数据**的技术称为**数据仓库** `data warehouse`。

而知识、信息、数据则是三个重要的 `IS` 概念，其定义如下：

- 数据：代表涉及业务活动的价值、质量、概念和时间的原始事实；
- 信息：增值事实，即已经被处理并概括为产品增值事实的数据，揭示了新的特征和趋势；
- 知识：对信息的理解，由经验或研究获得。

举个例子：电话号码就是数据，按照不同类别的联系人分组就是信息，根据信息知道在什么时候给什么人打电话则是知识。

### 3.1 事务处理系统

在决策的**操作级**上的系统主要是**联机事务处理系统** `OnLine Transaction Processing system,OLTP`。**事务** `transaction` 为工作的一个原子的逻辑单元，能够保证数据库的完整性（数据的一致性）。

事务处理系统与**数据库**技术关联在一起，数据库技术则主要包括：

- 并发控制 `concurrency control`：控制多个用户对数据的并发访问；
- 恢复 `recovery`：通过事务的自动**回滚** `rollback` 来保证数据的一致性；
- 业务逻辑 `business logic`：数据库的功能要符合企业的业务逻辑；
- 安全性 `security`：通过权限管理保证数据的安全性。

### 3.2 分析处理系统

处于决策的**战术级**的主要系统是**联机分析处理系统** `OnLine Analytical Processing,OLAP`。分析处理系统通过对预先存在的历史数据的分析来辅助决策。

分析处理系统与**数据仓库** `data warehouse` 技术关联在一起。数据仓库：

- 通过提取事务数据库中数据的增量拷贝来创建；
- 仓库内的数据只增不减，因此对于存储器的规模有较高要求；
- 数据仓库内的数据是**静态的** `static`，通常情况下不会改变；
- 数据仓库技术主要包含：
  - **汇总** `summarizing`；
  - **封装** `packaging`；
  - **分割** `partitioning`。

由于创建数据仓库具有极高的挑战性，所以出现了其他的解决方案：

- **数据集市** `data marts`：和数据仓库相同，是致力于分析处理的专用数据库；和数据仓库不同，
- `data webhouse`：在 `web` 上实现的**分布式的**数据仓库；

### 3.3 知识处理系统

处于**决策级**的主要系统是**知识处理系统** `knowledge processing systems`。知识是**作为经验的结果而累积的智力资本** `intellectual capital accumulated through experience`。想要维持当前在企业信息系统 `enterprise information system,EIS` 中的智力资本，就需要进行**知识管理** `knowledge management`。

知识管理能够有效地帮助组织**在信息系统中发现、组织、分配和运用被编码的知识** `discover,organize,distribute and apply the knowledge encoded in the information system`。**数据挖掘** `data mining` 技术属于知识管理领域，其主要目的为：

- **关联** association：在数据中发现一个事件导致另一个事件的模式；
- **分类** classification；
- **聚类** clustering。

用于数据挖掘的数据来源于数据仓库，其通过提供预测性的模型来扩充 OLAP 达到策略管理要求的能力。

#### 4、软件开发生命周期

**生命周期** lifecycle 是活动的有序集合，其主要包括：

- **应用建模方法** applied modeling approach；
- 在软件产品被转化的序列中的**精确阶段** exact phases：开始到逐步停止 from initial inception to phasing it out；
- **方法** approach 和相关的**开发过程** development process。

##### 4.1 开发方法

如今的软件已经变得越来越具有**交互性** interactive，具体体现在以下方面：

- 事件驱动 event-driven： GUI；
- 用户控制 user in control；
- 软件对象执行的事件**随机而不可预测** software objects service random and unpredict events。

**结构化方法** structured approach 可以很好地服务于传统软件，而对于现代的 GUI 软件，**对象方法** object approach 则是最好的方式。

- 结构化方法：功能性、过程性和强制性的方法。

  从系统建模的角度看，其主要基于两种技术：

  - 为**过程建模**使用的**数据流图** Data Flow Diagram,DFD；
  - 为**数据建模**使用的**实体关系图** Entity Relationship Diagram,ERD。

  结构化方法**以过程为中心** process-centric，并以 DFD 为开发的驱动力。

  系统在**功能分解** functional decomposition 过程中**被分解为可管理的单元**b be bracked into manageable units，同时将系统有层次地划分为由数据流连接的业务过程。

  结构化方法与现代的软件工程并不统一 not aligned with modern SE：

  - 倾向于按照**有序和可转换的** sequential and transformational 方式，而不是**迭代和增量** iterative and incremental 的方式；
  - 倾向于交付**僵化的** unflexible 解决方案来实现功能需求，导致可拓展性低下；
  - 采用从头开始的开发，**不支持对已存在的组建的复用** don't support the reuse of pre-existing components。
- 面向对象方法 object-oriented approach：将系统分解为不同粒度的构件，其分解的底部为具有对象的**类** class。类通过各种关系联系在一起，并且通过发送消息进行通信。

  面向对象方法**以数据为中心** data-centric。其被应用的最重要的种类为：

  - 工作组计算 workgroup computing；
  - 多媒体系统 multimedia systems。

  面向对象方法流行的主要原因是其能够通过**对象包装** object wrapping 及其他类似的技术来解决**应用程序积压** application backlogs 的问题，并且通过**迭代和增量**来与现代化的软件工程相统一。

  面向对象方法的缺陷：

  - 需要在一个较高的抽象层面进行分析；

- 项目管理较结构化方法更加困难；
- 面向对象解决方案比传统的结构化系统更加复杂 more complex。

### 4.2 生命周期的阶段

软件的生命周期主要包含五个阶段：

- 业务分析 business analysis：又叫做**需求分析** requirements analysis，其主要目的是**确定** determine（业务分析）和**详细说明** specify（系统分析）客户的需求。

  业务分析与**业务过程重组** Business Process Reengineering,BPR 相关联。BPR 的目的是提出开展业务和获得竞争优势的新方式。

  业务分析主要分为以下部分：

  - 需求确定 requirements determination：

    **需求** requirement 就是对系统服务或约束的陈述，其主要分为：

    - 功能性需求 functional requirement：描述了软件必须实现的所有功能以及为了实现这些功能所需执行的动作，这是软件需求的主体；
    - 非功能性需求 non-functional requirement：作为功能性需求的补充描述了对软件性能、运行环境、规范标准等的要求；
    - 设计约束 design constraint：是对软件系统设计过程的限制条件。

    需求确定主要包含以下内容：

    - 服务陈述 service statement：主要描述了：
      - 在任何时候都需要遵守的**业务规则** business rule：工资需要在每月1日支付；
      - 系统必须执行的一些**计算** computation：售货员薪水以其销售额作为自变量代入公式进行计算。
    - 约束陈述 constraint statement：主要描述了：
      - 对**系统行为的限制** restriction on system behavior：安全性约束，只有管理员才可以改变员工的薪水信息；
      - 对**开发上的限制** restriction on development：如必须使用 IDEA 开发工具。
  - 需求说明 requirements specification：需求说明阶段开始的标志是**开发者使用某种特定方法（如 UML）开始对需求建模** developers start modeling the requirements with particular method。

    在需求说明的过程中：

    - CASE 工具用于输入 enter、分析 analyze 和文档化 document 模型；
    - 在前一过程编写的需求文档通过图形模型和 CASE 生成的报表来完善。

    而在需求说明的过程中最重要的技术则是：

    - 类图 class diagrams；
    - 用例图 use case diagrams。

    理想情况下，需求说明模型应当**独立于用于部署系统的软硬件平台。**

- 系统设计 system design：系统设计主要分为两个部分：

  - 体系结构设计 architectural design：即**根据其模块而进行的系统描述** the description of the system in terms of its modules。

    在这一过程中所需要关心的是：

    - 选择合适的**解决方案策略** solution strategy；
    - 系统的**模块化** modularization。

**客户/服务器模型** client/server model 经常被拓展来提供**三层体系结构** three-tier architecture，在此结构中**应用逻辑**构成了独立的层次。

优秀的结构设计需要设计一个**可适应性的** adaptive 系统，即**可理解的** understandable、**可维护的** maintainable 和**可扩展的** scalable 系统。

- 详细设计 detail design：**描述每个软件模块内部的运行** description of the internal workings of each software module，为每个模块的开发提供了详细的算法和数据结构。

  此过程依赖于**底层的** underlying 实现平台：客户端、服务器、中间件。

- 实现 implement，实现解决的是：
  - 已购买软件的安装；
  - 对**客户定制的** custom-written 软件的编码；
  - 其他的重要内容如：测试的加载、生产数据库的加载、测试、用户培训以及硬件事务等等。

  一般情况下，实现团队的程序员分为两组来实现两个工作：
  - 客户端编程 client programming；
  - 服务器编程 server programming。
- 集成与部署 integration and deployment：此过程主要包含以下内容：
  - 模块集成 module integration：在进行模块集成时，如果一些模块尚未准备好，可以使用**桩** stub 来模拟所缺模块的活动。
  - 应用集成 application integration：将**不同的** disparate 应用集成为一个**统一的** unfied 系统。
  - 部署 deployment。
- 运行与维护 operation and maitenance：运维分为运行和维护两个阶段：
  - 运行：标志着**从现有业务解决方案到新方案的转换** change over from existing business solution，无论这种转换是否在软件中。
  - 维护：维护占用了生命周期的大部分时间。其由三个不同的阶段组成：
    - 内容处理 housekeeping：执行日常维护任务来维持系统的可访问性和可操作性；
    - 适应性维护 adaptive maintenance：对系统运行进行监控和审核，调整系统的功能来满足环境变化的需要；
    - 完善性维护 perfective maintenance：重新设计和修改系统来适应新的实质性变化的需求。

    当软件系统的持续维护不可支持时将很难从软件的可用性中获益，此时系统将被**淘汰** phasing out。

#### 4.3 跨越生命周期的活动

有些活动将会跨越软件的整个生命周期，例如：

- 项目规划 project planning：**估计** estimate 项目的可交付性、成本、时间、风险、里程碑和资源需求的活动，它也包括对开发方法、过程、工具、标准和团队组织的选择。

  项目是一项**活动的目标** moving target，在项目规划中，最典型的约束就是**时间与金钱** time and money。

  评估项目可行性的考虑因素有：
  - 操作可行性；
  - 经济可行性；
  - 技术可行性；
  - 进度表可行性；
  - 其他约束/因素。
- 度量 metrics：度量主要应用于一下几个层面：

    - 度量**开发时间和工作量** `developing time and effort`；
    - 度量应用于软件产品的**质量和复杂性** `quality and complexity` 方面；
    - 度量的一个重要的应用时在生命周期的不同阶段度量开发模型，评估**过程**的效果并改进工作质量。
- 测试 `testing`：应当为每个在需求文档中描述的功能模块定义**测试用例** `test case`。

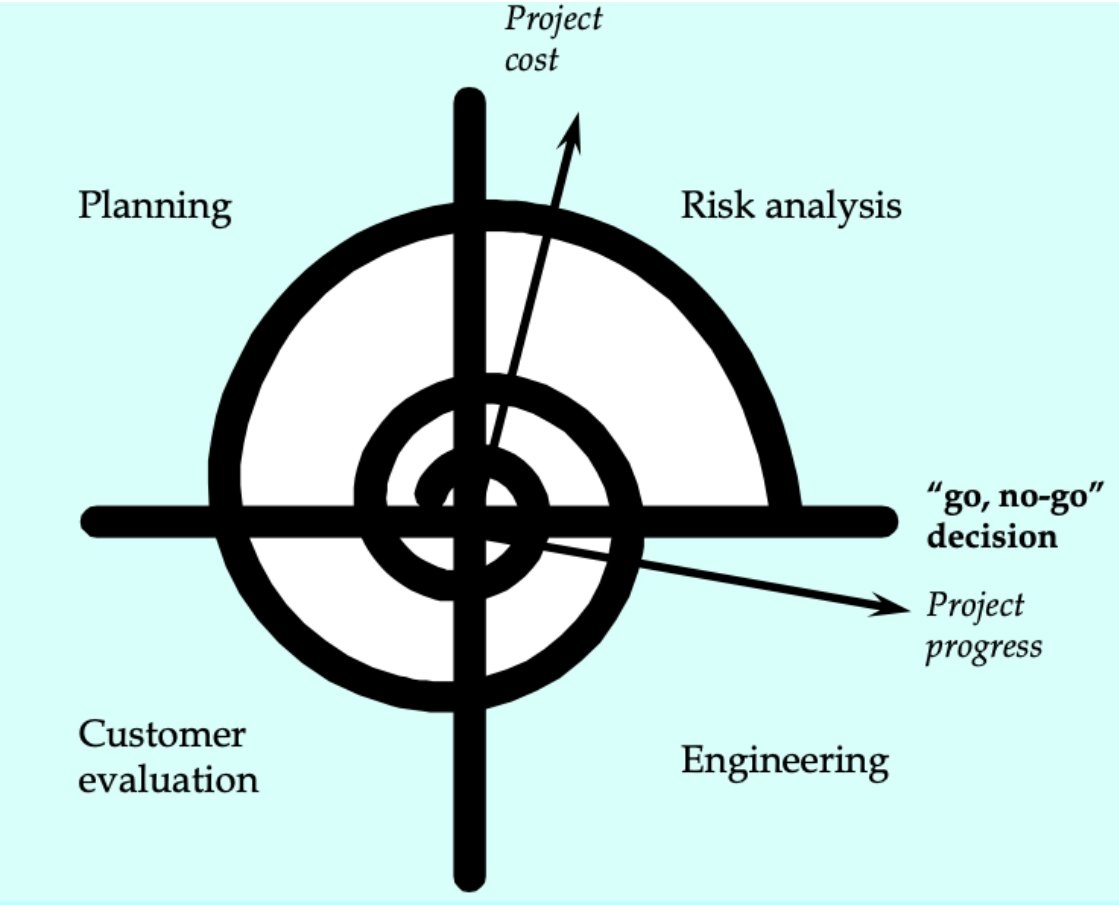
  **基于可执行的** `execution-based` 测试包括：

    - 规格说明测试 `testing to specs`：黑盒测试；
    - 代码测试 `testing to code`：白盒测试。

#### 5、开发模型与方法

常用的开发模型有以下几种：

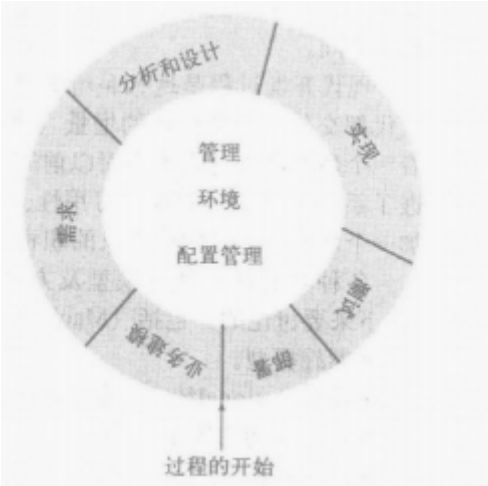
- 螺旋模型 `the spiral model`：该模型将**系统规划** `system planning`、**风险分析** `risk analysis`、**软件工程** `software engineering` 和**客户评估** `customer evaluation` 这四个活动作为四个象限显示在图中，一起创建了笛卡尔图中的螺旋式循环：



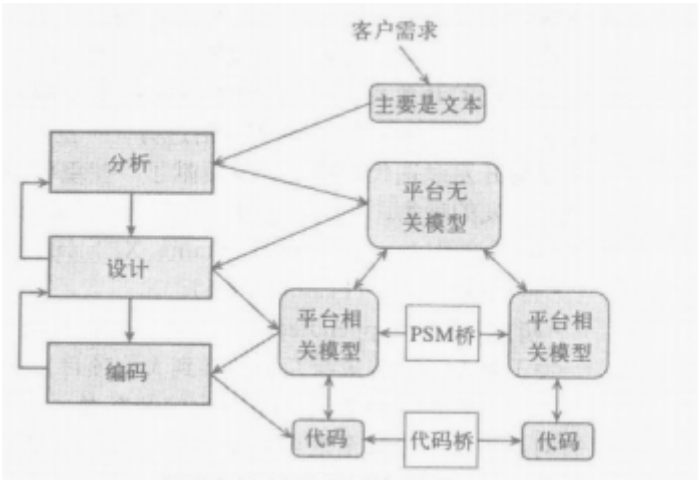

  根据图中的螺旋模型，系统开发始于系统规划，然后进入风险分析。在风险分析阶段的分析结果决定了是否进入下一阶段。在工程象限中，需要评估工程进展。在项目进入下一轮迭代前需要进行用户评估，用户的反馈将在下一次迭代中进行处理；

- `IBM Rational` 统一过程 `IBM Rational Unified Process,RUP`：`RUP` 在二维关系中组织项目：
    - **横向维度** `horizontal dimension` 代表每个项目迭代的连续**阶段** `phase`：
        - 初始 `inception`；
        - 细化 `elabration`；
        - 构造 `construction`；
        - 转换 `transition`；
    - **纵向维度** `vertical dimension` 代表软件开发领域：业务建模、需求、分析和设计、实现、测试和部署等。

在实践中很难区分横向维度和纵向维度之间的区别，为了不混淆，下图显示了在一个活动循环中所安排的纵向`RUP`科目：

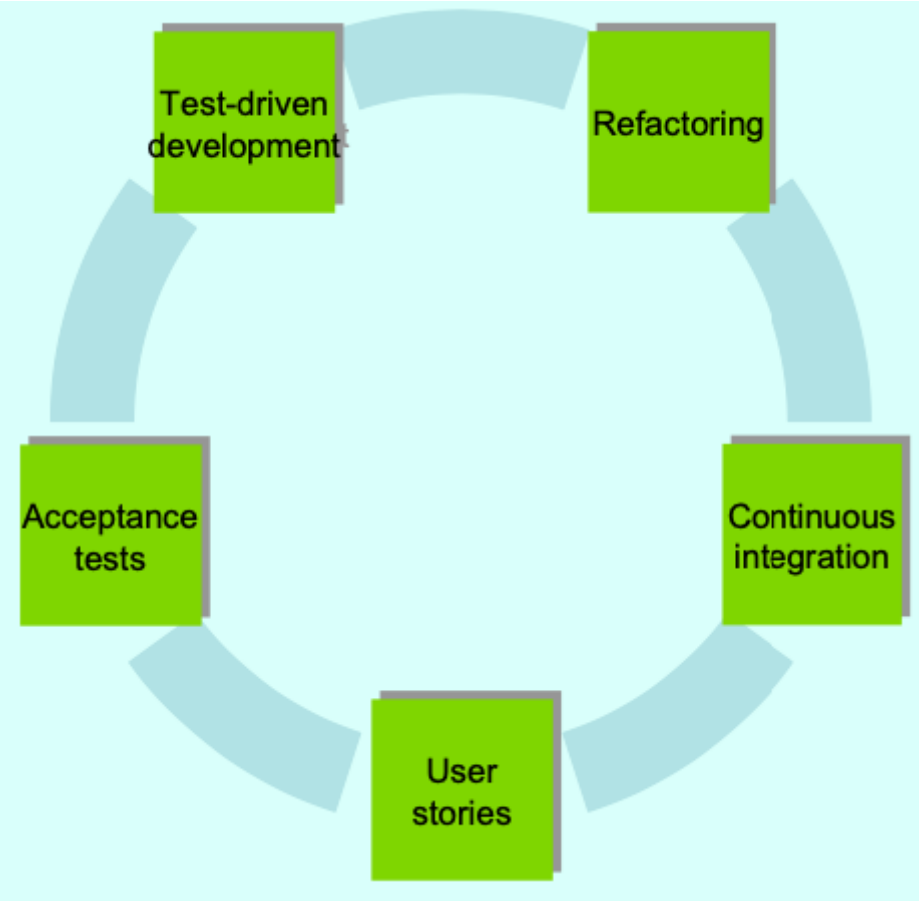



- 模型驱动的体系结构`Model Driven Architecture,DMA`：`DMA`是一个可执行建模和从规格说明生成程序的框架。使这样的规格说明可行的标准如下：
  - **统一建模语言**`UML`，对任务建模；
  - **元对象设施**`Meta-Object Facility,MOF`使用标准元模型库使衍生的规格说明能够共同工作；
  - `XML`**元数据交换**`XML Meta-Data Intercahnge,XMI`将`UML`映射到数据库模式，并允许灵活的数据挖掘。




- 敏捷开发模型`agile software development`：软件生产中的敏捷性的关键点如下：
  - 个体和交互胜过过程和工具`individuals and interactions over processes and tools`；
  - 可工作的软件胜过宽泛的文档`working software over comprehensive documentation`；
  - 客户协作胜过合同谈判`customer collaboration over contract negotiation`；
  - 响应变化胜过遵循计划`responding to change over following a plan`。

  敏捷开发是一种迭代和增量模型，其定义了多个新的术语：




最知名的敏捷开发包括：

- **极限编程** `eXtreme Programming,XP`；
- 特征驱动 `feature-driven` 开发；
- 学习开发 `learn development`；.

- 面向切面编程 `Aspect-Oriented Programming,AOP`： `AOP` 通过识别所谓的**横切关注点** `crosscutting concerns` 来生成更多的模块化系统。这些模块被称为**切面** `aspect`。

  这些切面在被称为**切面编排** `aspect weaving` 的过程中进行集成：

  - 动态编排：在**运行**时进行编排；
  - 静态编排：在**编译**时进行编排。

  方面编排被应用于程序执行的**接入点** `join points`:所谓接入点，就是软件组件预定义的点如：方法的调用、属性的访问等。

## Chapter 2 需求确定

### 2.1、从业务过程到解决方案的构想

#### 2.1.1 业务过程建模

什么是 `IT` 解决方案？ `IT` 解决方案是：

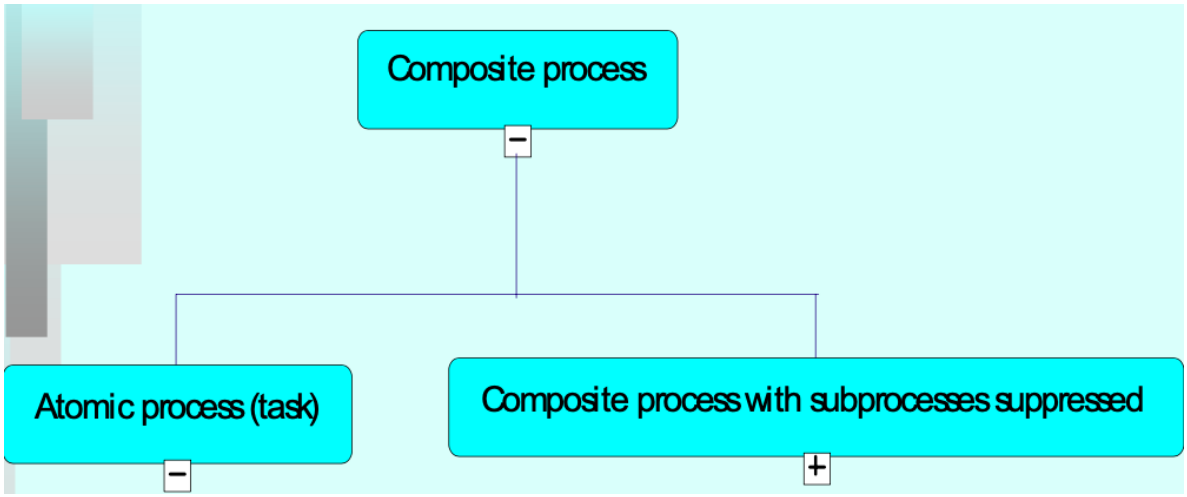
- 一个**业务解决方案** `business solution`；
- 一个**业务过程的实现** `implementation of a business process`；
- 有时能够推动**业务的创新** `business innovation`；
- 一种**基础设施服务** `infrastructure service`；
- 一种**商品** `commodity`。

**业务过程建模表示法** `Business Process Modeling Notation,BPMN` 可以用于填补业务过程设计与过程实现之间的空白，其**专门用于对由活动定义的业务过程建模** `dedicated to modeling business process defined as activities`。

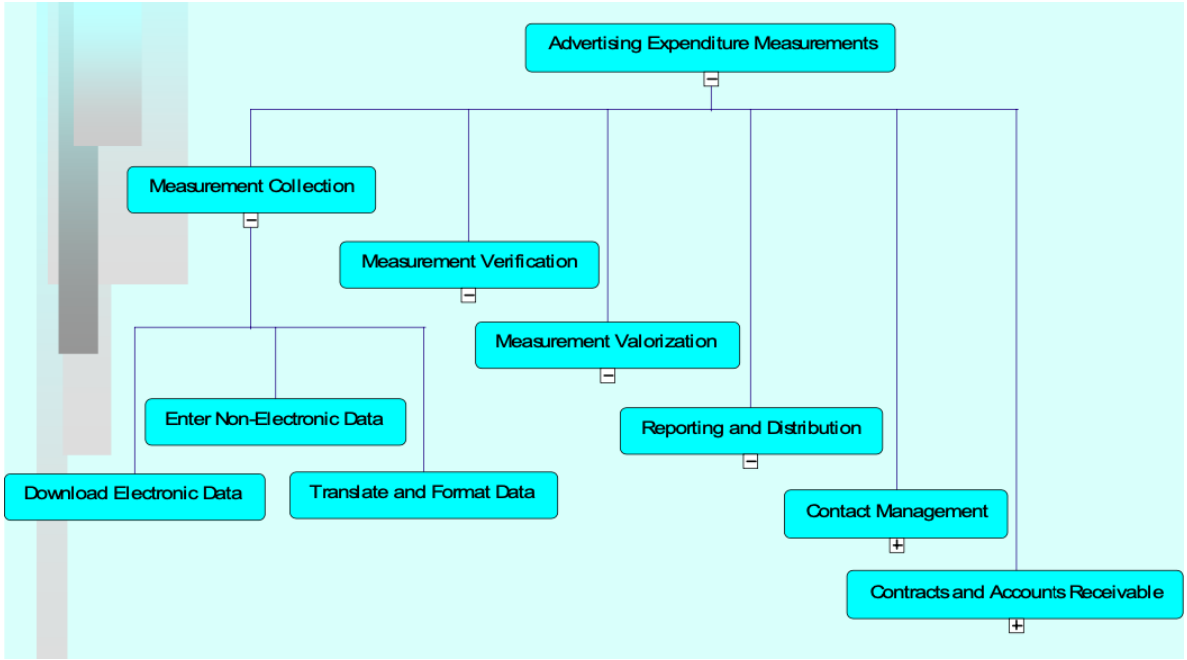
关于过程：

- 业务过程可能是**手工**`manually`操作的活动或**自动化的**`automated`服务；
- 一个过程至少有一个**输入流**`input flow`和一个**输出流**`output flow`，当过程获得控制后，主要通过将输入流转变为输出流来完成相应的活动；
- 过程可能是**原子的**`atomic`和**复合的**`composite`：
  - 原子过程：也被称为**任务**`task`，不包含任何子过程；
  - 复合过程：可以包含多个子过程并通过这些子过程来描述其行为。

过程的层次模型可以通过**过程层次图**`process hierarchy diagram`来表示：



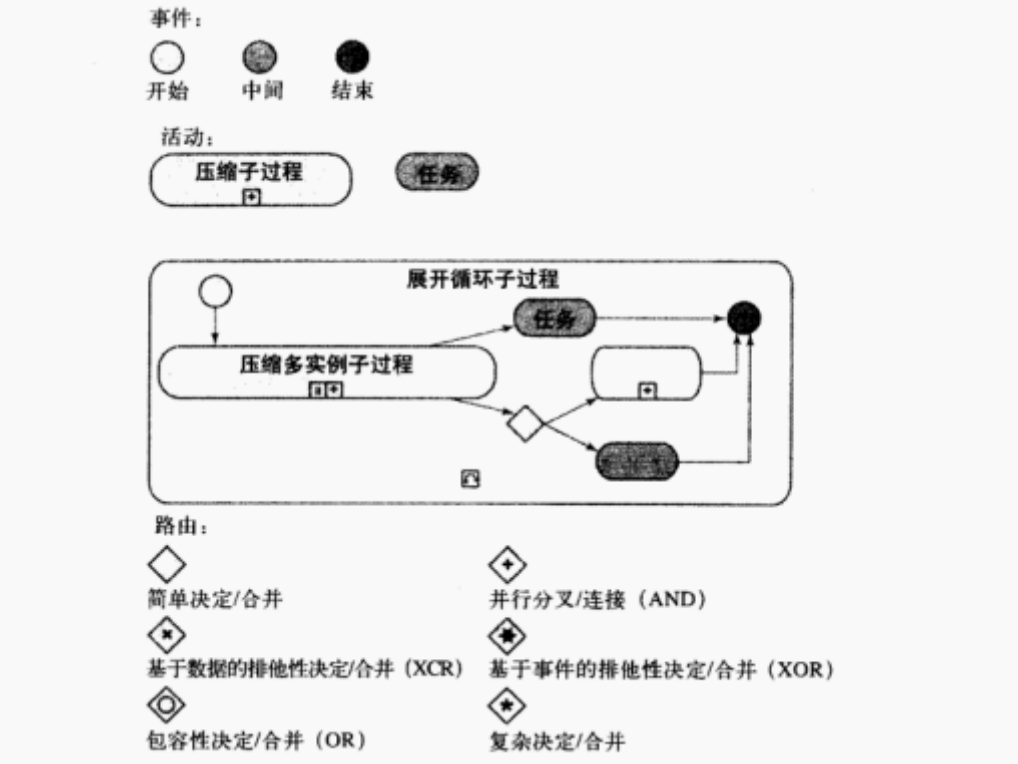

下面是一个实例：



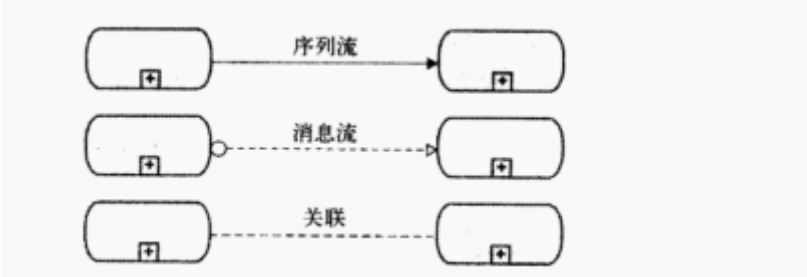

`BPMN`提供了四种基本的建模元素：

- 流对象`flow object`：定义了业务过程的结构，是`BPMN`的核心元素，根据类别其可分为：
  - 事件`event`：
    - 某些“发生的”事物`something that happened`；
    - 由特定的原因触发的一个影响。
  - 活动`activity`：
    - 某些必须进行的工作`some work that must be done`；
    - 可能是一项任务或者是一个子过程。

- 路由`gateway`：控制多个序列流的分支与聚合`control the divergence and convergence of multiple sequence flows`；

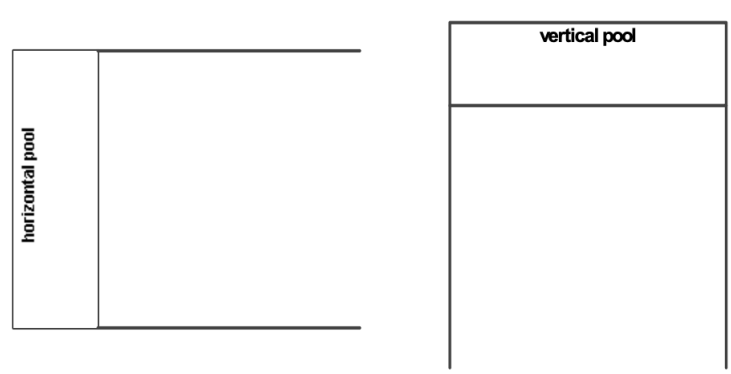



- 连接对象`connecting objects`：连接对象用于连接流对象，根据其类别可以分为：
  - 序列流`sequence flow`：表示一个过程中活动**完成的序列**；
  - 消息流`message flow`：表示准备发送和接收信息的两个业务实体之间的**消息流向**；
  - 关联`association`：关联两个流对象，或者一个人工制品与一个流对象。



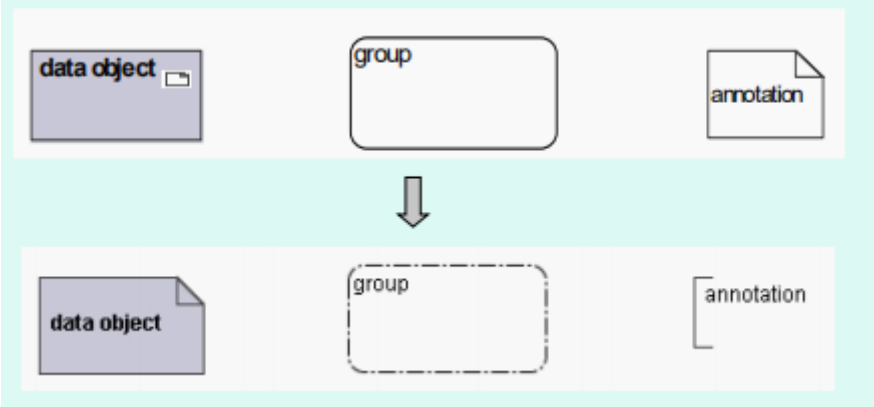

- 泳池/泳道`pools/swimlanes`：泳池描述了一个过程中的**业务实体**`business entity`，泳池群`pools`则是一个**自包含**`self-contained`过程的集合(不同泳池群的成员能够通过**消息流**或**关联**与人工制品进行通信)

  泳池主要由水平和竖直两种：

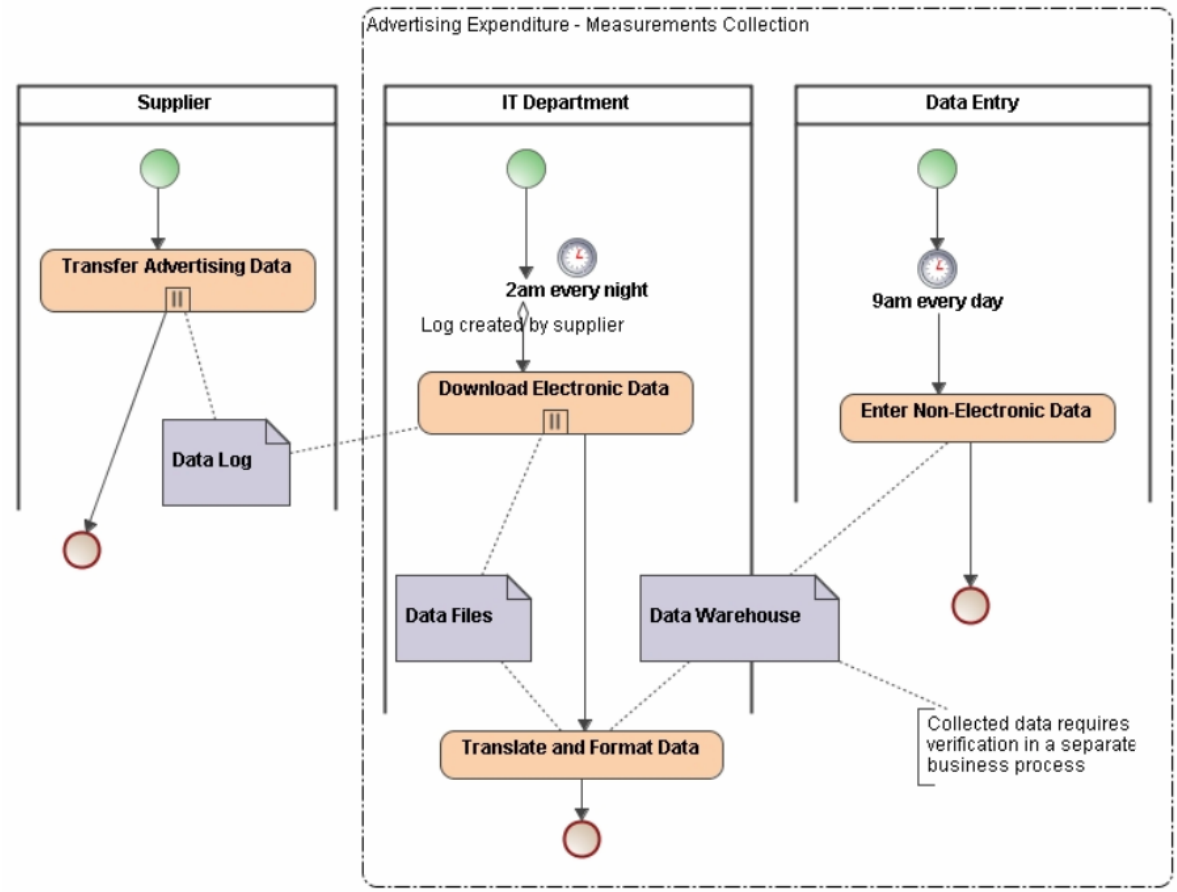



- 人工制品`artifacts`：人工制品通过允许拓展基本表示法来应对特殊的建模环境从而提供了**附加的建模灵活性**`additional modeling flexibility`。预定义的人工制品有三类：
  - 数据对象`data object`：活动需要的数据或者产生的数据；

- 组`group`：不影响过程序列流的一组活动；
- 注释`annotation`：为业务过程图的读者提供附加的文本信息。




下面是一个使用了上述四种元素的业务过程图实例：



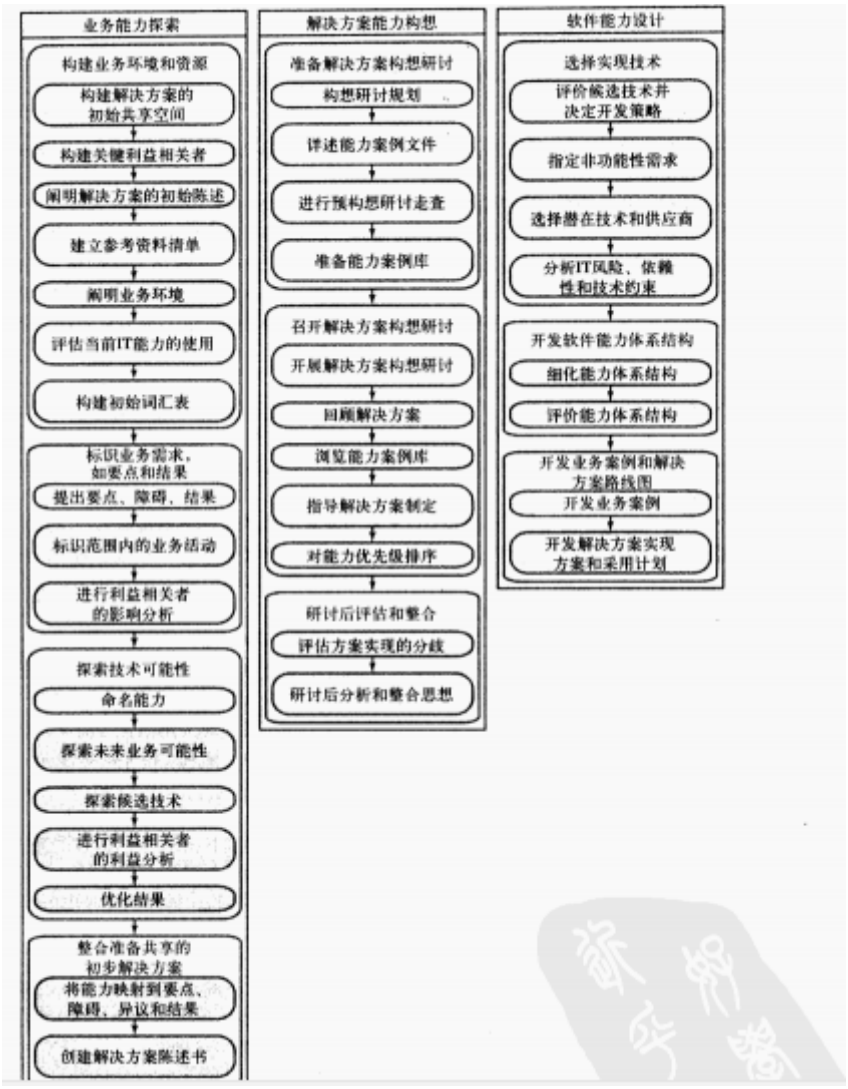

### 2.1.2 解决方案的构想

**解决方案构想**`solution envisioning`是一个业务价值驱动方法，其能够提供解决当前业务问题和促进将来业务创新的`IT`服务。解决方案构想在业务和`IT`利益相关者间建立了紧密的联系，并且整合了业务战略方法和软件开发能力。

解决方案构想实现的支撑：**效果**`effectivativeness`、**效率**`efficiency`、**优势**`edge`。其过程主要分为三个阶段：

- 业务能力探索`business capacity exploration`：确定业务能力（企业的`IT`解决方案提交具体成果的能力），这个阶段描述了**能力用例**`capacity case`，用来提供解决方案的思路；
- 解决方案能力构想`solution capacity evisioning`：将能力用例发展成为结局方案概念，确保利益相关者对这个方案的意见一致性；
- 软件能力设计`software capacity design`：**取决于系统的实现技术。**开发软件能力体系结构，细化具有项目规划和风险分析的业务用例。

使用业务过程图可以图形化描述解决方案构想的过程要素：




以下是三种实现策略：

- 常规开发`custom development`；
- 基于包的开发`package-based development`；
- 基于构件的开发`component-based development`。

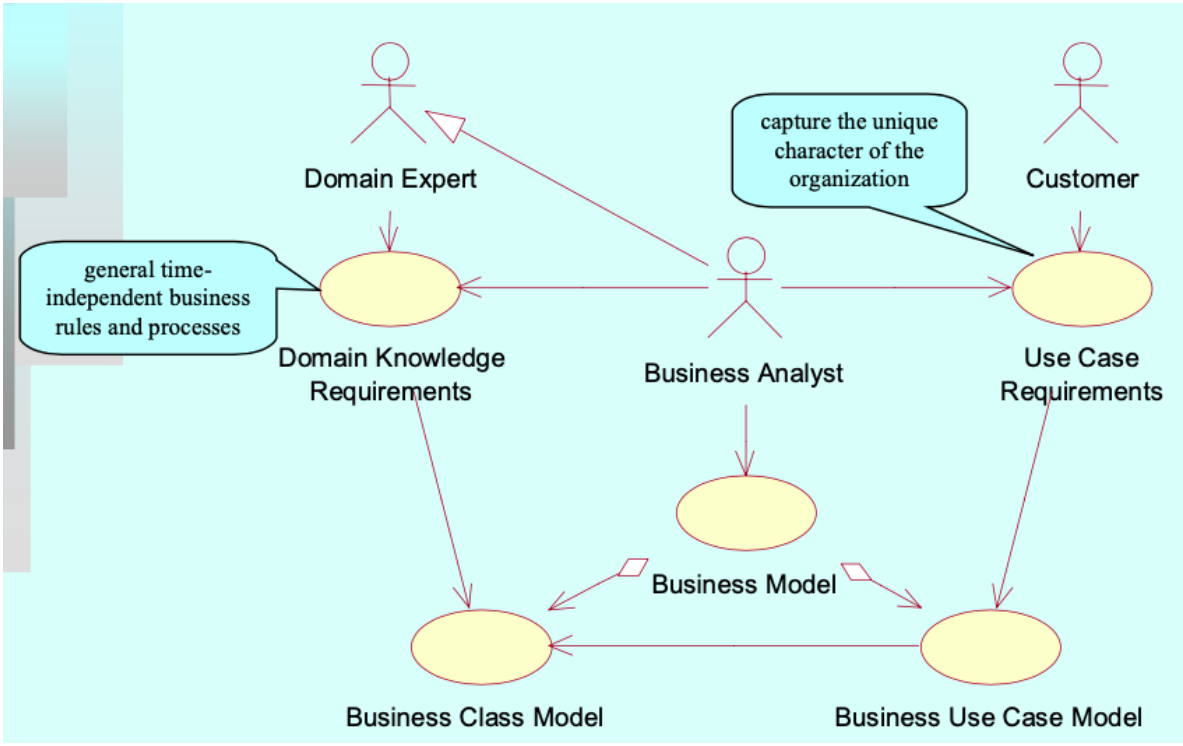
### 2.2、需求引导

#### 2.2.1 需求与需求引导

首先考虑什么是**需求**`requirements`，需求可以分为以下两类：

- 系统服务`system service`：即**功能性需求**`functional requirements`，其内容主要包括：
  - 系统的范围`scope of system`；
  - 必要的业务功能`necessary business functions`；
  - 所需的数据结构`required data structures`。
- 系统约束`system constraint`：**非功能性需求**`nonfunctional requirements`的本质不是行为的，而是系统开发和实现过程中的约束，遵守这些约束的程度决定了软件的质量。其分为以下几种类型：
  - 可用性`usability`：定义使用系统的难易程度；
  - 可复用性`reusability`：定义在新系统的开发中，重复使用之前已实现的软件构件的容易程度；
  - 可靠性`reliability`：与系统失效的频率和严重性以及系统从失效中恢复的程度相关；
  - 性能`performance`：通过系统响应时间、事务处理时间、资源开销等决定；

- 效率`efficiency`：与取得软件成果或达到软件目标的成本和时间相关；
- 适应性`supportability`：定义了系统被理解、修改、完善和拓展的容易程度。包括**可理解性**`understandability`、**可维护性**`maintainability`和**可拓展性**`scalability`。




### 2.2.2 需求引导的传统方法

需求引导的传统方法包括：

- 与客户和领域专家面谈`interviewing customers and domain experts`：面谈是发现事实和聚集信息的基本技术。

  面谈通常有以下两种形式：

  - 结构化的/正式的`structured/formal`：
    - 需要提前准备；
    - 问题预先确定；
    - 需要非结构化面谈补充。
  - 非结构化的/非正式的`unstructured/unformal`：目的是鼓励客户讲出自己的想法，导出业务分析员没能提出的相关问题的需求。

  面谈时应该避免的三种问题：

  - **固执己见的**`opinionated`问题；
  - **带偏见的**`biased`问题；
  - **强加的**`imposing`问题；

  建议提的几类问题：

  - 面谈中提出的任何问题的具体细节`specific details`；
  - 未来的憧憬`vision for the future`；
  - 另类想法`alternative ideas`；
  - 问题的一个解决方案被接受的最低限度`minimal acceptable solution to the problem`；
  - 其它信息来源`other sources of imformation`；
  - 要求图表`soliciting diagrams`；
- 调查表`questionnaire`：一般作为**面谈的补充**`in addition to interview`。

  封闭式`close-ended`问题可以采用如下3种形式：

    - 多项选择 multiple-choice 问题；
    - 评价 rating 问题；
    - 排序 ranking 问题。
- 观察 observation：一般作为面谈甚至调查表的补充。

  观察通常有以下三种形式：

    - **被动** passive 观察：业务分析员观察业务活动而不干扰或不直接干预它；
    - **主动** active 观察：业务分析员参与到活动中且有效成为团队的一部分；
    - **解释** explana 观察：在工作过程中，用户向观察者说明进行的活动；

  要使观察具有代表性，观察应该**持续较长的一段时间** prolonged time，且在不同的时间段和不同的工作负荷 workload 下进行；

  观察的主要困难：

    - 人们在被观察的情况下总想表现出不同的行为 work to rule；
    - 另外，由于有些工作的性质需要处理敏感的个人信息和组织秘密，因此观察法也有道德、隐私，甚至法律问题存在 knowledge work。
- 文档和软件系统的研究 study of documents and software systems：用于发现**用例需求** use case requirements 和**领域知识需求** domain knowledge requirements 的宝贵技术。

#### 2.2.3 需求引导的现代方法

项目风险较高时常使用现代的需求引导方法。导致项目高风险的因素有：

- 不明确的目标 unclear objects；
- 未成文的过程 undocumented process；
- 不稳定的需求 unstable requirements；
- 不完善的用户知识 eroded user expertise；
- 没有经验的开发人员 inexperienced developer；
- 不充分的用户承诺等 unsufficient user commitment；

需求引导的现代方法主要包括：

- 原型法 prototyping：原型是一个演示系统，它是解决方案的一件“快且脏”的工作模型。构造软件原型的目的是为了使整个系统或者系统的一部分对用户可视化，以便获得他们的反馈。

  原型法主要可以分为两类：

    - **丢弃** throw away 原型：以需求确定为目标，在需求引导过程后丢弃；
    - **进化** evolutionary 原型：以产品的发布速度为目标。
- 头脑风暴 brainstorming：头脑风暴是通过**放下** put aside 公正、社会禁忌和规则来产生新思想或者发现专业问题解决方案的一种会议技术；
- 联合应用开发 Joint application deveplopment, JAD：是一种类似于头脑风暴的技术。其参与成员主要包括：
    - 领导 leader：组织和召集这次会议的人；
    - 文书 scribe：在计算机上记录 JAD 会议的人；
    - 客户 customers：包括用户和经理，是交流、讨论需求、做出决策，认可项目目标等工作的主要参与者；
    - 开发人员 developers：开发队伍中的业务分析员和其他人员。
- 快速应用开发 rapid application development, RAD：快速应用开发不仅仅是一种需求引导方法，它还是将软件开发作为一个过程的方法。其目标是快速交付系统解决方案，而技术上的精良对交付速度来说则是次要的。

  RAD 通常组合以下技术：

    - 进化原型 evolutionary prototyping；

- `CASE`工具；
- `SWOT`；
- 交互式`JAD`；
- 时间盒`timing box`。

### 2.3、需求协商和确认

为什么进行**需求协商和确认**`requirements negotiation and validation`：

- 客户的需求**重叠**`overlap`或者**矛盾**`comflict`；
- 需求可能是**模棱两可**`ambiguous`或者**不现实的**`unrealistic`；
- 需求可能还**没有发现**`undiscovered`；
- 需求可能是**超出范围的**`out of scope`。

这里需要注意的一点是，需求协商和确认是与需求引导**同步进行**`parallel`的，在进行需求引导时，就需要接受一定程度的审查。

**需求依赖矩阵**`requirements dependency matrix`是一种发现需求矛盾和重叠的技术。其按一种分类顺序分别在行和列的表头上列出需求标识符，其右上部分没有被使用，剩下的单元格表示任何两个需求是否重叠、矛盾或者独立。

| *Requirement* | *R1* | *R2* | *R3* | *R4* |
|---|---|---|---|---|
| *R1* | | | | |
| *R2* | *Conflict* | | | |
| *R3* | | | | |
| *R4* | | *Overlap* | *Overlap* | |

一旦解决从需求依赖矩阵中发现的矛盾和重叠，就产生了一组修正后的需求。需要对这些需求进行风险分析并排列出优先级：

- 风险分析`risk analysis`：确认那些很可能在开发阶段产生困难的需求；
- 排列优先级`prioritizaiton`：用于在项目面临延迟的时候重新界定其范围。

典型的风险分类如下：

- 技术风险`technical`：需求在技术上难以实现；
- 性能风险`performance`：需求实现后，会延长系统的响应时间；
- 安全风险`security`：需求实现后，会破坏系统的安全性；
- 数据库完整性风险`database integrity`：需求不容易验证，并且可能导致数据不一致性；
- 开发过程风险`development process`：需求要求开发人员使用不熟悉的非常规开发方法，如形式化规格说明方法；
- 政治风险`political`：由于内部的政治原因，证实很难实现需求；

- 法律风险`legal`：需求可能触犯现行法规或者假定了法律的变更；
- 易变性风险`volatility`：需求很可能在开发过程中不断变化或进化。

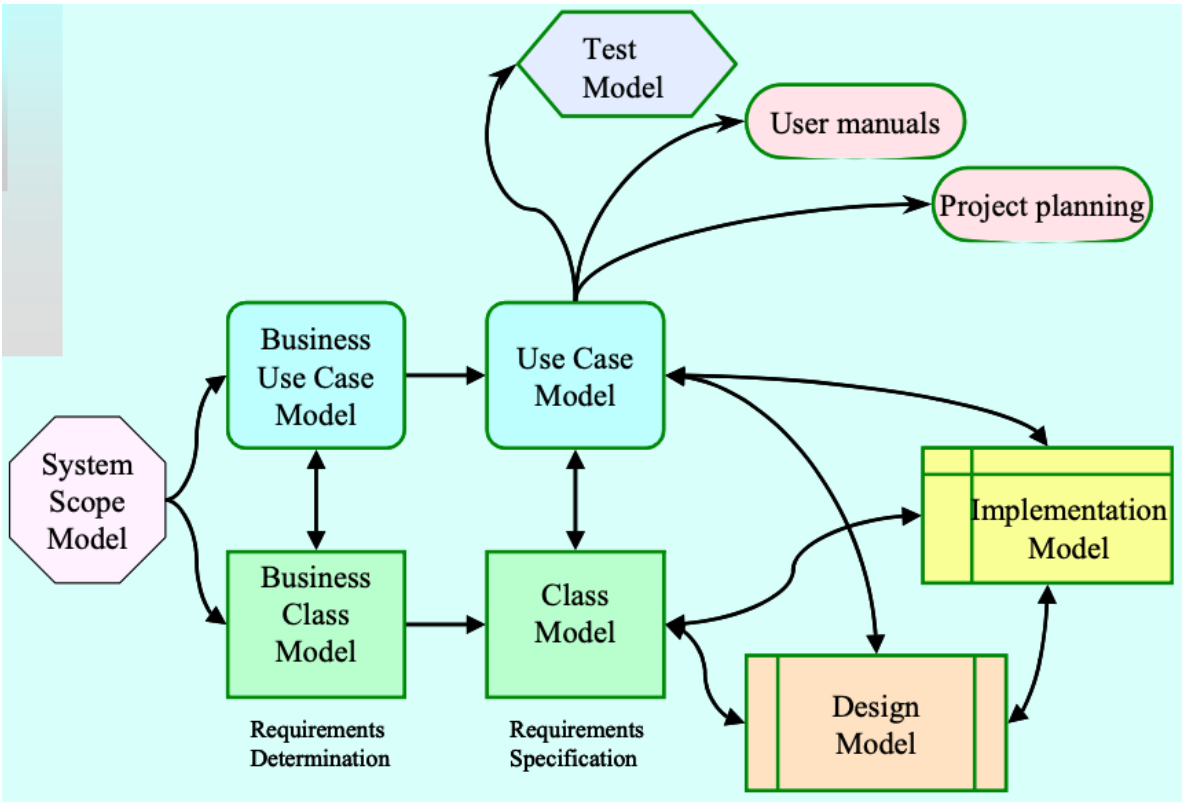
### 2.4、需求管理

需求管理是整个项目的一部分，其主要涉及：

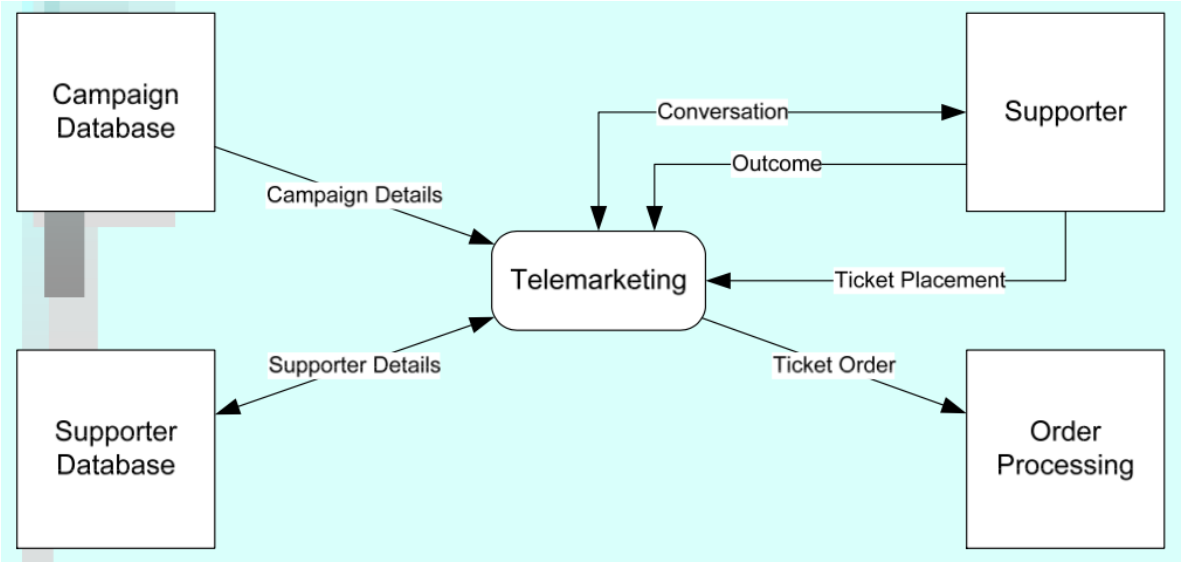
- 需求标识与分类`requirements identification and classification`，其涉及的主要技术有：
  - 唯一标识符`unique identifier`：数据库自动生成；
  - 在文档层次中的顺序编号`sequential number with document hierarchy`；
  - 在需求分类中的顺序编号`sequential number with requirements category`。
- 需求层次`requirements hierarchies`：需求可以按父子关系来建立出层次化结构：
  - 父级需求由子级需求组成；
  - 子级需求是父级需求有效的子需求。
- 变更管理`change management`：需求是变更的，开发阶段越靠后，需求变更的成本越大，所以需要进行合理的变更管理。优秀的变更管理需要有以下步骤：
  - 文档变更请求`document change request`；
  - 访问变更影响`access a change impact`；
  - 对变更作出影响`effect the change`。
- 需求可跟踪性`requirements traceability`：需求可跟踪性是变更管理的重要部分，在连续的生命周期阶段，可跟踪关系能跨越许多模型。

### 2.5、需求业务模型

需求的高层可视化表示需要在需求确定阶段完成。高层可视化模型需要确定系统范围，标识主要用例并建立最重要的业务图。下图显示了需求确定阶段三个模型间以及其余生命周期阶段模型之间的依赖性：




#### 2.5.1 系统范围模型

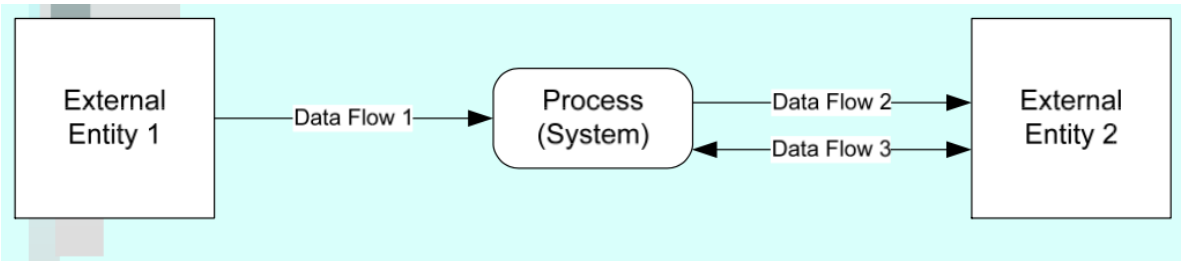



在系统开发中需要考虑的一个主要问题就是由于需求变更引起的范围蠕变，所以就需要对系统范围进行界定，即建立**系统范围模型** `system scope model`。

系统范围可以通过识别外部实体以及在外部实体和我们系统之间的 `I/O` 数据流来确定：

- 外部实体：期望从我们这里得到服务或为我们提供服务的其他系统、组织等；
- 数据流：在业务系统中，上面提到的服务转换成的信息。

`UML` 并未提供定义系统范围的可视化模型，建立系统范围模型时常常采用的是老式的 `DFD` 环境图 `context diagram`：

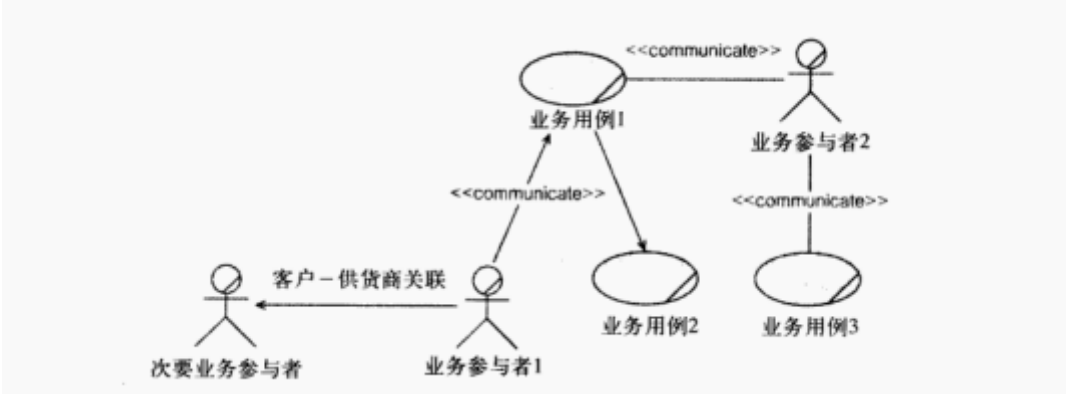



### 2.5.2 系统用例模型

**系统用例模型**是在高抽象层次上的模型，其建立的目的是为了使业务过程被开发中的信息系统支持。

系统用例图提供了对系统用例模型的可视化支持，其提供了对所期望过程和系统行为的鸟瞰图，对每个业务用例的叙述性描述是简洁的、面向业务的、并集中在活动的主要流程上。

业务用例图表示法(Business use case diagram)：




### 2.5.3 业务词汇表(Business Glossary)

软件开发中的一个不显著但是重要的方面是明确业务和系统术语：

- 如果术语不明确，项目利益相关者之间的交流就不会精确，由此会导致解决错误的问题；
- 为了增进交流、避免误解，必须创建术语和定义的词汇表，并且和利益相关者共享该表。

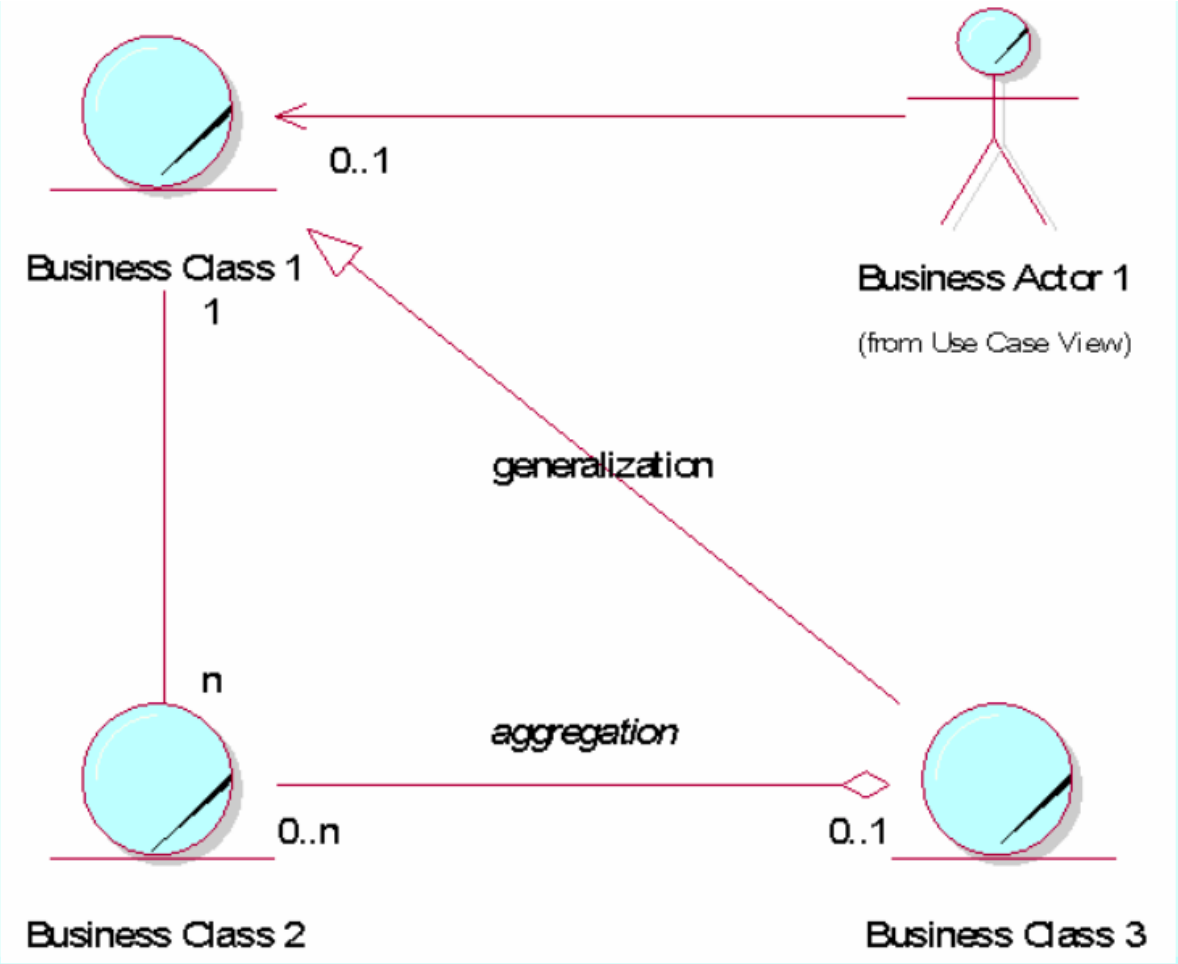
| Term | definition |
| --- | --- |
| bonus campaign | A special serious of activities, conducted within a campaign, to additionally entice supporters to buy the campaign tickets. Typical examples are giving free tickets for bulk or early buying or for attracting new supporters. A particular kind of bonus campaign can be used in many campaigns. |
| campaign | A government approved and carefully planned series of activities which are intended to achieve a lottery objective. |
| draw | An act of randomly choosing a particular lottery ticket as a winning ticket. |
| lottery | A funds raising game of chance, organized by the charity in order to make money, in which people (supporters) buy numbered tickets to have a chance of winning a prize if their number is chosen in a draw. |
| placement | Acquisition of one or more lottery tickets by a supporter during telemarketing. The placement is paid by a supporter with a credit card. |

#### 2.5.4 业务类模型

业务类模型是`UML`类模型(业务类模型与其他固有类模型相比，抽象级别更高，它标识系统中业务对象的主要类型，业务类模型常常没有明确类的属性结构——类名和简要的描述基本足够)，业务类连接到模型通过的是三个`UML`关系：

- 关联`association`：
  - **对象间**的关系，表示一个类对象项关于另一个类对象的知识，或者一个类的不同对象之间的相关知识；
  - 这些知识是有关从一个对象到另一个或者其他更多对象的语义连接，这些连接的存在可以在对象中进行导航(程序可以导航到已连接的对象)；
  - 重要特性：
    - 重数：定义了一个类可能的实例数；
      - 重数在类连接线的两端定义，可以是0、 1或者n (这里“n"表示许多对象可以被连接)；
    - 参与；
- 聚合`aggregation`：**对象间**的关系，是语义更强的关联，如一个类的实例“包含有”另一个类的实例；
- 泛化`generalizaiton`：**类间**的关系。

**业务类图**表示法：




**2.6、需求文档(Requirements document)**

需求确定的最终目的是建立起一份需求文档，其模版(定义了文档的结构，并给出文档的每一节需要书写的内容的详细指南)如下：

- 项目准备`project preliminaries`
  - 项目的目的和范围
  - 业务环境
  - 利益相关者
  - 多种解决方案
  - 文档综述
- 系统服务`system service`(定义系统必须完成什么)
  - 系统范围：用环境图建模(必须清楚地定义所提出项目的边界)；
  - 功能性需求：用业务用例图建模；
    - 只提供了功能性需求详细列表的一个高层范围；
  - 数据需求：用业务类图建模；
    - 每个业务类需要进一步解释，应该描述类的属性内容，确定标识的类属性，否则，不可能恰当地解释关联；
- 系统约束`system constraints`(描述系统在完成服务时怎样被约束)
  - 界面需求：用户接口的定义(定义产品如何与用户交互)；
  - 性能需求：对于系统性能要求的约束；
  - 安全性需求：描述用户在系统控制下对信息的存取权限；
  - 操作性需求：
    - 决定系统运行的软硬件环境;
    - 可能对项目的其他方面有一些影响，如用户培训和系统维护;

    - 政策与法律需求：通常是假定的，而非在需求文档明确表述的；
    - 其他约束：例如维护性需求、可用性需求等；
- 项目的其他问题`project matters`
    - 开放问题：任何可能影响项目成功但在文档其他标题下没有讨论的问题的详细说明；
    - 初步安排：人力资源的初步分配等；
    - 初步预算：可以表示为一个范围而非精确的数字；
- 附录`appendices`
    - 业务词汇表：定义需求文档中的术语、简写和缩写;
    - 业务文档与表格：通过研究工作流过程中使用的文档和表格(完整的业务表格相对于空表格传达的理解层次不同)，可以相当好地理解它所定义的业务领域;
    - 参考文献
        - 提供需求文档中引用的文档，或者在需求文档的准备过程中使用的文档；
        - 会议纪要、备忘录和内部文档也应该包含在引用文献中；

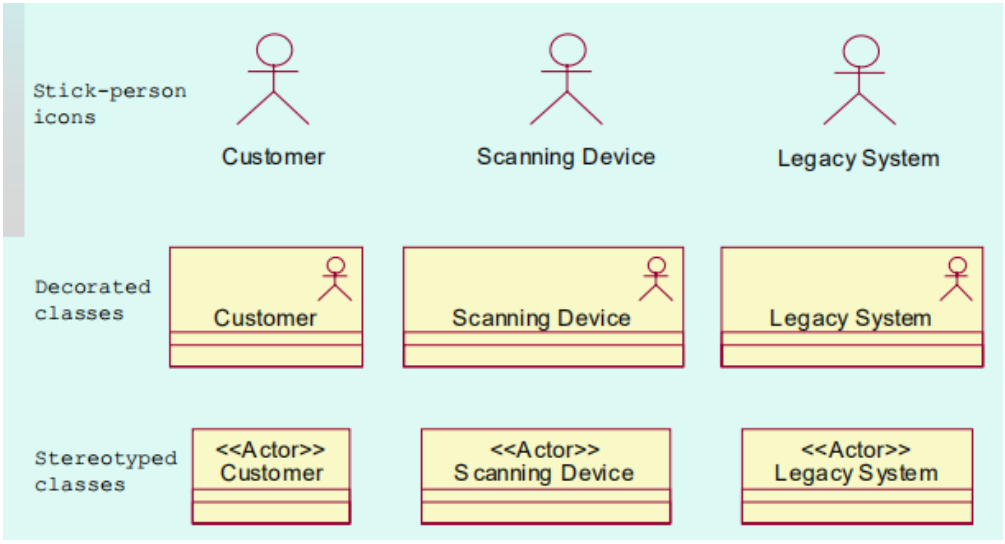
---

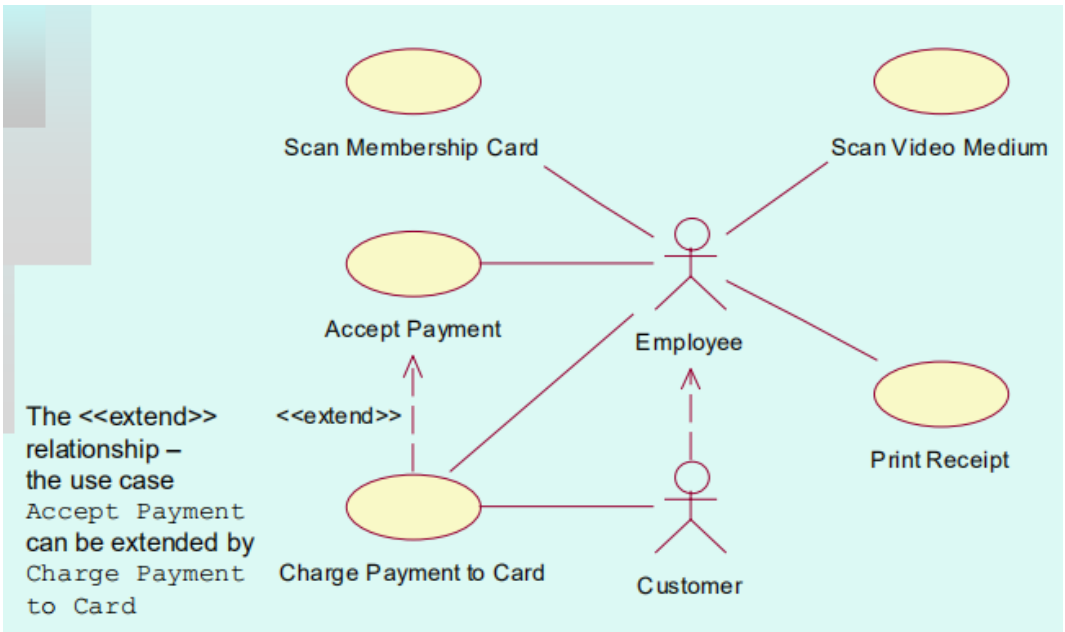
## Chapter 3 可视化建模基础

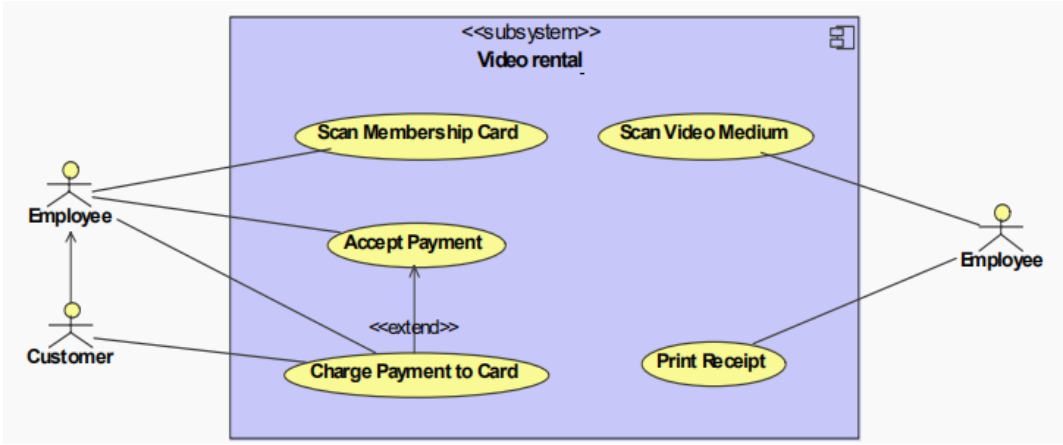
### 3.1 用例视图(The use case view): The most important behavioral modeling techniques

> 1.用例模型：主要的`UML`用例，**行为建模的焦点**;
>
> 2.行为模型：表示商业事务、作用于数据上的操作及算法;
>
> 3.行为建模：**系统的动态视图(功能性需求的建模)**;
>
> 4.行为建模的可视化技术：用例图、顺序图、通信图和活动图;
>
> 5.用例`use case`：
>
> (1).表示参与者从外部可以看到的业务功能，并可在开发过程中单独测试;
>
> (2).驱动整个软件开发的生命周期(从需求分析到测试和维护);
>
> (3).大多数开发活动的焦点和参照;
>
> (4).`UML`用例捕获*外部可见的和可测试的系统行为(指当系统响应外部事件时所做的事情)*;
>
> (5).用例与可以在不同抽象级别上应用的模型保持一致， 就可以捕获整个系统的行为，或者捕获系统任何部件的行为，例如子系统、构件或类;
>
> 6.参与者`Actor`：与用例交互(目的是收到*有用的结果*)的任何事物(人、机器);
>
> 7.用例图`use case diagram`：
>
> (1).参与者和用例的可视化表示，伴随有附加的定义和说明;
>
> (2).完全文档化的系统预期行为的部分模型(被称为"部分"模型的原因：`UML`模型一般由表示模型不同视点的很多图(及相关文档)组成);
>
> (3).用例模型可以被看成是描述所有业务过程的通用技术，而不仅仅是信息系统过程;

- 参与者
    - 参与者和用例由对功能性需求的分析来确定;
    - 功能性需求在用例中出现，用例通过给参与者提供结果来满足功能性需求;
    - 参与者是**主题**外部的人或事物针对用例所扮演的角色(不是某人或某事的特定实例);

        - *主题可以是任何一组用例*，用例模型为这组用例而建(如子系统、构件、类);
        - 参与者通过诸如交换信号和数据的方式与主题交流信息;
    - 参与者的典型图形表示是一个“木头人”(通常，参与者可以被表示为类的矩形符号):
    - 参与者的3种图形表示：木头人、装饰类、构造型类
        - 

- 用例
    - 用例表示对于参与者有价值的功能单元;
    - 主题`Subject`: 可以将用例组织起来表示一个主题
        - 每个用例都明确规定某种行为(也许包括变体)这样主题就可以与一个或多个参与者协作;
        - 用例定义主题所提供的行为，而不需要引用主题的内部结构;
    - 导出\确定:
        - 从参与者任务的标识中导出;
        - 通过对功能性需求的直接分析来确定(一般来说一个功能性需求可以直接映射到一个用例);
    - 命名：从主题或参与者角度(一般不加以从参与者角度);
    - `UML`表示：椭圆，名字置于椭圆内部或下面;
- 用例图
    - 建立**系统行为模型**的主要可视化技术：把用例赋给参与者，并允许用户在用例之间建立关系;
    - 用例图的元素需要进一步描述来提供具有完整的*用例规格说明文档*
    - 

- 

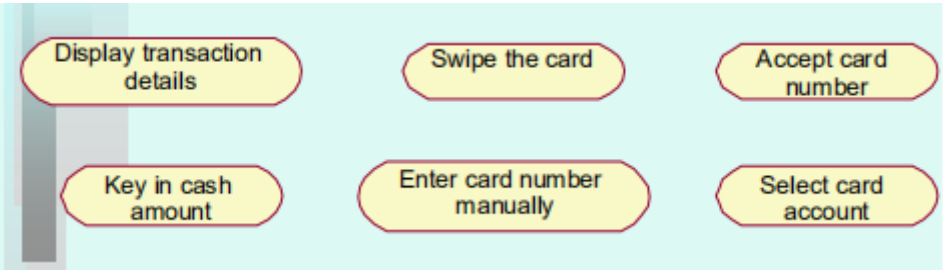

- 用例文档化：
  - 用例文档的典型描述包含：
    - 简要描述`Brief Description`
    - 涉及的参与者`Actors involved`
    - 用例开始所需要的前置条件`Preconditions`
    - 事件流的详细描述：
      - 主事件流`Main Flow`：
        - 可以将主事件流分解为事件子流(子流`Subflows`可以被进一步划分为更小的子流以提高文档的可读性)；
      - 定义异常情况的备选流`Alternative Flows`；
    - 后置条件`Postconditions`：定义用例结束后系统的状态；
  - 

| Use case | Accept Payment |
|---|---|
| Brief description | This use case allows an Employee to accept the payment from `Customer` for a video rental. |
| Actors | Employee, Customer. |
| Preconditions | Customer expresses readiness to rent the video and he/she possesses valid membership card and the video is available for rental. |
| Main flow | The use case begins when the Customer decides to pay for the video rental and offers cash or debit/credit card payment.<br>The Employee requests the system to display the rental charge... |
| Alternative flows | The Customer does not have sufficient cash and does not offer the card payment. The `Employee` asks the system to verify ... |
| Postconditions | If the use case was successful, the payment is recorded in the system's database. Otherwise, ... |

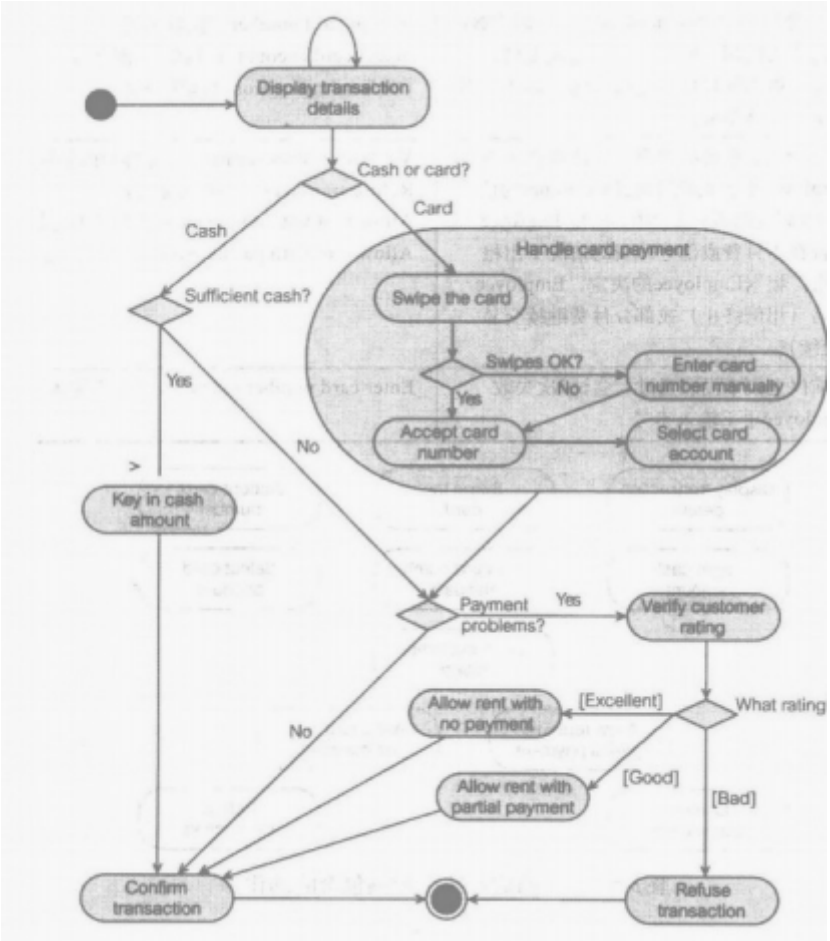
### 3.2 活动视图(The activity view)

> 1.活动模型`Activity model`：
>
> (1).*表示行为(行为可能是用例的规格说明，也可能是可以在很多地方复用的一个功能)，由独特的元素组成;*
>
> (2).填补了用例模型中系统行为的高层表示与交互模型(顺序图和通信图)中行为的低层表示之间的空隙;

> 2.活动图：
>
> (1).显示计算的步骤;
>
> (2).描述哪些步骤可以顺序执行，哪些步骤可以并行执行
>
> 3.动作`actions`：活动的执行步骤(在一个活动内，动作不能被进一步分解);
>
> 4.控制流`Control Flow`：
>
> (1).从一个动作到下一个动作控制的流程(流程的意思是一个节点的运行影响其他节点的运行，同时也受其他节点运行的影响)称为控制流;
>
> (2).在活动图中，用边表示这种依赖;
>
> 5.如果用例文档已经完成，则可以从主事件流和备选流的描述中发现活动和动作
>
> 6.活动模型的用途：活动模型除了为用例提供详细的规格说明外，在系统的开发中还有其他用途
>
> (1).在创建任何用例之前，活动模型可以用于在高抽象层次上理解业务过程;
>
> (2).可用于较低的抽象层次上设计复杂的顺序算法，或设计多线程应用系统中的并发性;

- 动作:
  - 建立：通过用例文档;
  - `UML`表示：圆角矩形;
  - 

- 活动图:
  - 活动图：显示连接动作和其他节点(如决策，分叉、连接，合并和对象节点)的流;
    - 一般情况下，在活动和活动图之间具有一对一的关系——一个活动图表示一项活动;
    - 允许在一个图中嵌套多个活动;
    - 除非一个活动图表示的是一个连续的循环，否则，活动图应该有一个使活动开始的初始动作，还应该有一个或多个终止动作;
    - 实心圆表示活动开始，牛眼符号表示活动结束;
  - 流可以分支`branch`及合并`merge`：
    - **产生可选的计算线程**;
    - 钻石框显示分支条件，分支条件的出口由事件或守卫条件控制;
  - 流可以分叉`fork`及再连接`re-join`：
    - **产生并发(并行)的计算线程**;
    - 流的分叉或连接用短线表示;
    - 没有并发过程的活动图类似传统的流程图;
  - Example:




### 3.3 结构视图

> 1.结构视图：
>
> (1).表示系统的静态视图——表示数据结构、 数据关系及作用在这些数据上的操作;
>
> (2).静态建模的主要可视化技术是**类图——主要的结构图**(其他结构图是构件图和部署图);
>
> 2.类建模：集成和包含了所有其他建模活动
>
> (1).类模型：定义那些捕获系统内部状态的结构
>
> (2).标识类和类的属性，包括关系，还定义必要的操作来完成用例中明确规定的系统的动态行为需求;
>
> (3).当以程序设计语言实现时，类既表示应用系统的静态结构，也表示其动态行为
>
> (4).结果：类图和相关的文档
>
> 3.用例建模和类建模在实际中一般是并行的：
>
> (1).通过提供辅助或补充信息，这两种模型彼此促进;
>
> (2).用例有助于类的发现，反之，类模型又有助于发现被忽略的用例;

- 类
  - 实体类(entity classes)：为应用域定义数据模型的类，表示持久的数据库对象；
  - 表现(边界、视图)类(presentation(boundary\view) classes)：class that define GUI objects.
  - 控制类(control classes)：控制程序逻辑及处理用户事件的类；
  - 资源类(resource classes)：负责与外部数据源通信的类

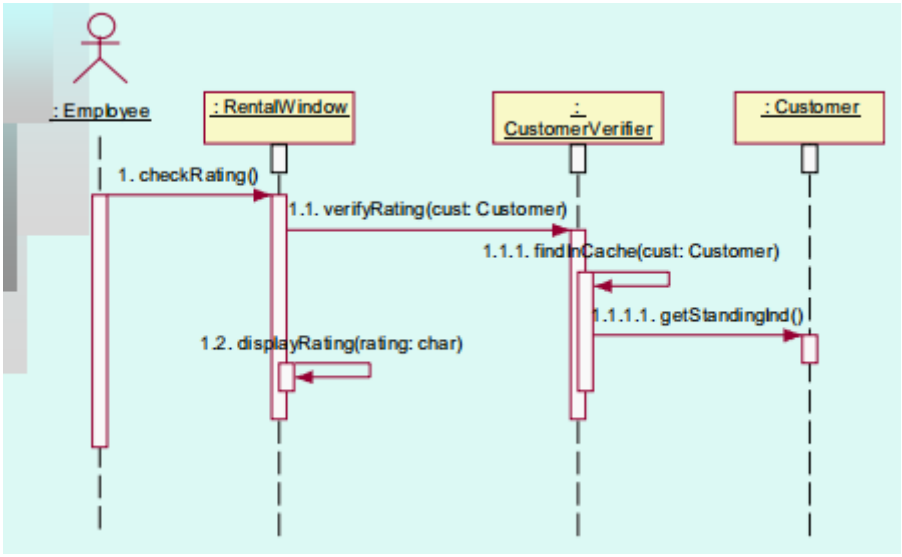
    - 中介者(调解)类(mediator classes)：为满足业务交易，而设立管理内存高速缓存中的实体对象的类；
- 属性：
    - 标识属性(关键字)；
    - 描述属性；
- 关联
- 聚合
- 泛化
- 类图

略

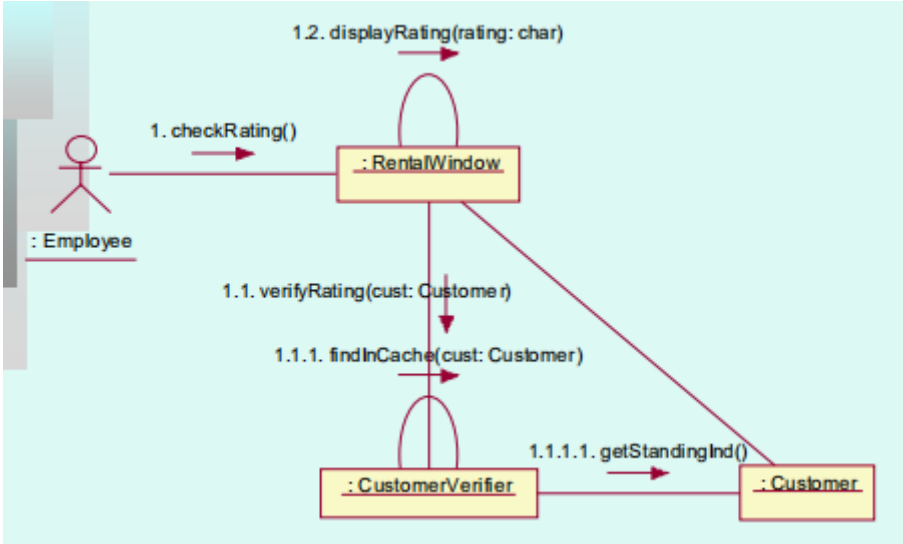
### 3.4 交互视图

> 1.交互建模`intercation modeling`：
>
> 捕获对象之间的交互(为了执行一个用例或用例的一部分，这些对象之间需要通信);
>
> 2.交互模型：
>
> 为用例的一部分、一个或多个活动提供详细的规格说明，用于需求分析的较高级阶段；
>
> 3.活动建模与交互建模的区别与联系：
>
> (1).活动建模经常在较高的抽象级别上完成——它显示事件的顺序， 但没有将事件分配给对象；交互建模显示了协作对象之间的事件(消息)顺序；
>
> (2).活动图更抽象，经常捕获整个用例的行为；交互用例更详细，趋向于对用例的某些部分建模；
>
> (3).活动建模和交互建模都表示用例的实现;
>
> (4).一个交互图有时对话动图中的单个活动建模；
>
> 4.交互图的种类：顺序图`Sequence diagram`和通信图`Communication diagram`
>
> (1).顺序模型强调时间顺序，而通信模型强调对象关系
>
> 5.一次交互是某种行为的消息集合，这些消息在连接(持久或瞬态连接)上的角色之间进行交换；

- 顺序图：
    - 二维图：在水平维上显示角色(对象)，在垂直维上从上到下显示消息的顺序
        - 每一条垂直线称为对象的**生命线**`lifeline`，在生命线上被激活的方法称为**激活**(或执行规格说明)，它作为垂直的高矩形被显示在顺序图上；
        - 对象生命线表示为垂直的虚线；
        - 激活表示为生命线上的窄矩形;
    - Example:




    - 例子使用消息的层次编号来显示消息之间的激活依赖和相应方法
  - 顺序图主要特性：
    - 一个箭头表示一条**消息**：
      - 消息是从调用对象(发送者)发送到被调用对象(目标)中的操作(方法)；
      - 作为最低限度，消息只有名字；可以包括消息的实际参数和其他控制信息；
    - 实际参数与目标对象方法中的形式参数相对应：
      - 实际参数可能是输入参数(从发送者到目标)或输出参数(从目标返回到发送者)
        - 输入参数可以用关键字in标识(如果没有关键字，则认为是输入参数)
        - 输出参数可以用关键字out标识；
      - 参数也可能是`inout`,但在面向对象解决方案中很少见
    - 如所提到的那样，**没有必要显示从目标到发送对象控制的返回结果**
      - 到目标对象的同步`sychronous`消息箭头表示到发送者的控制自动返回;
      - 目标知道发送者的`OID`;
    - 可以将消息发送给对象的**收集**(例如，收集可以是集合、列表或对象数组);
      - 这种情况经常发生在调用对象被连接到多个接收者对象时(因为关联的重数为一对多或多对多);
      - 迭代标记`Iteration marker`——消息标签前的星号——表示在收集上的迭代；
- 通信图：
  - 顺序图的另一种表示方法(虽然它们的区别很明显，在通信图中没有生命线或激活，但两者都隐含地以箭头表示消息);
  - 一般情况下，当表示涉及很多对象的模型时，通信图比顺序图更形象;
  - 另外，与顺序图不同，对象之间的实线可能表明这些对象的类之间需要关联;
  - Example:




- 类方法
  - Examining the interactions can lead to the discovery of methods:
    - 每一条消息触发被调用对象上的一个方法；
    - 操作与消息具有相同的名字；
  - 顺序图中的类所接收到的消息转变为表示这些对象的类中的方法；

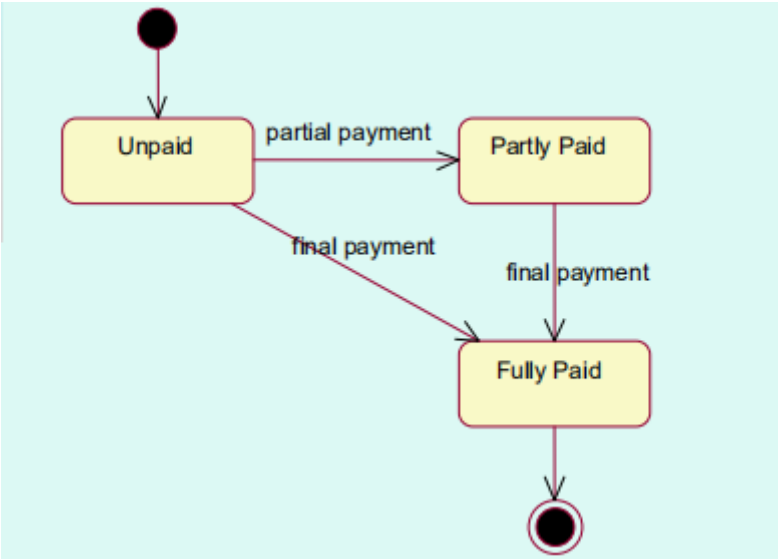
### 3.5 状态机视图(The state machine view)

> 1.状态机模型：
>
> (1).说明类中的动态变化`Specifies dynamic changes in a class`；
>
> (2).描述类的对象可能具有的不同状态`Describe various states in which objects of the class can be`；
>
> (3).对于所有涉及初始化对象的类的用例，这些动态变化描述了这些用例中的对象行为；
>
> 2.**一个对象的状态由对象属性(原始属性和那些指定为其它类的属性)的当前值决定**；在其存在期内，一个对象始终是一个，且其标识保持不变
>
> 3.状态机模型捕获类的生命历史；
>
> 4.状态图：状态和由**事件**引起的**转换**的偶图，是业务规则(在一段时期内保持不变并相对独立于特定的用例)模型；

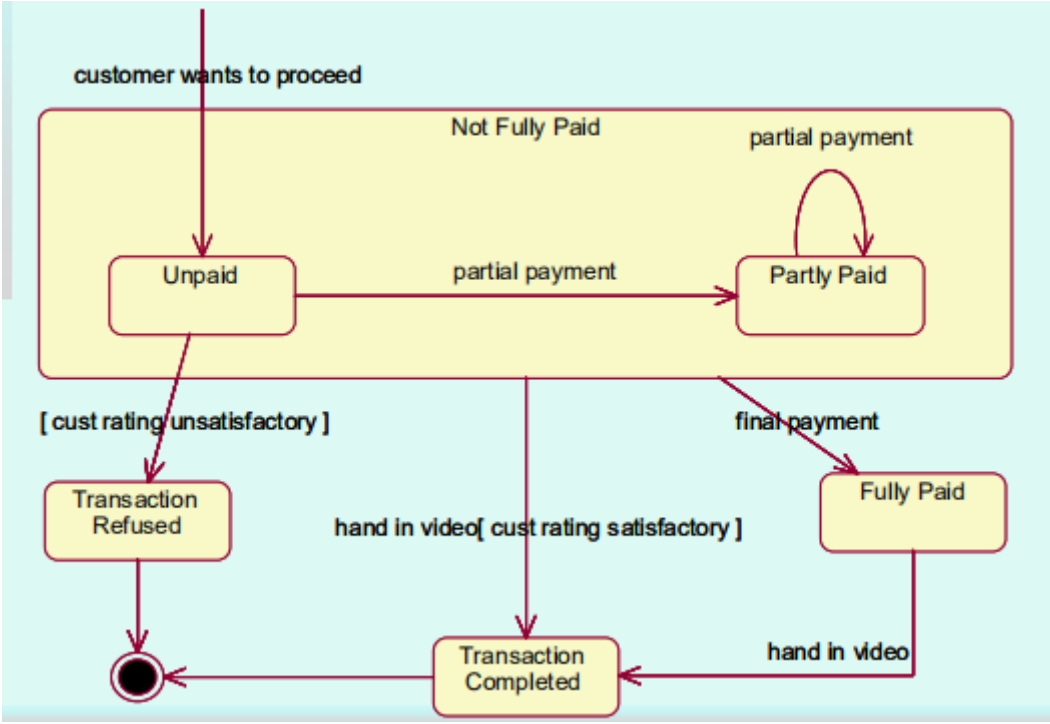
- 状态和转换：
  - 对象会改变其属性的值，但并不是所有属性值的改变都会引起**状态转化**；
  - 状态机建模的本质：
    - 状态机模型是为具有感兴趣状态(由业务建模决定的)的类、而不是为任何状态变化的类构造的；
- 状态机图`State machine diagram`：
  - > *一般将状态机图附到类上，但是，也可以将其附到其他建模概念上，如用例。当被附到一个类上时，状态机图决定那个类的对象如何响应事件*。更确切地说，对于对象的每一种状态，状态机图决定当对象接收到一个事件时，它执行什么*动作*。相同的对象对于同样的事件可能执行不同的动作，这取决于对象的状态。动作的执行通常会导致状态改变。***important***

- **转换(transition)**的完整描述由3部分组成：Event (parameters) [guard] / action
  - 这三部分每一部分都是可选的：如果转换线本身是自解释的，则将所有的部分都省略是可能的。
  - 事件是影响对象的迅速发生的事件，它可能有参数，也可能有一个条件守卫：
    - mouse button clicked (right_button) [inside the window];
- 事件与守卫之间的区别并不总是很明显，区别是事件“发生”， 且在对象准备处理之前被保存起来；在处理事件时，计算守卫条件的值来确定是否应该激发转换。
- 动作：
  - 一个简短的自动计算，在转换激发时运行;
  - 一个动作也可以与一个状态相关联；
  - 通常，一个动作是一个对象对监测到的事件的反应；
  - 动作也可以包括长计算，称为活动；
- 嵌套状态：包含其他状态的状态；
  - 组合状态：抽象的，只是所有嵌套状态的通用标记；
  - 一个转换冲出了组合状态的边界意味着它可以从任何嵌套的状态激发；
  - 出自组合状态边界的转换可以从一个嵌套的状态激发;

- 状态和事件：



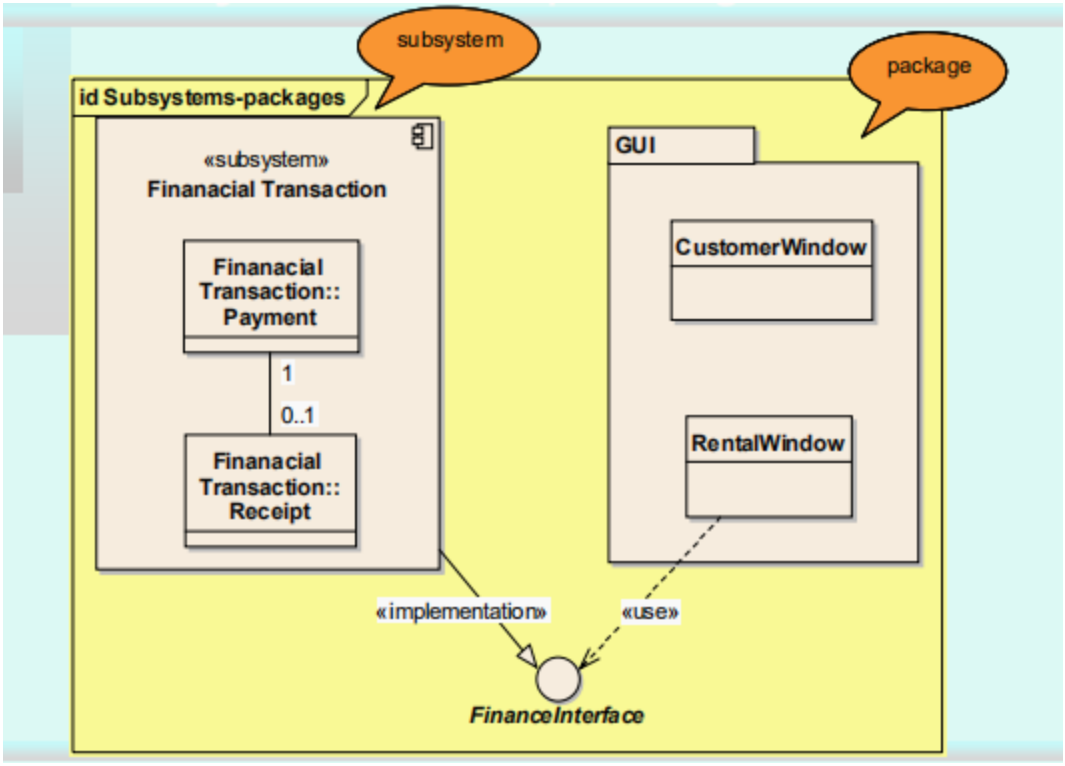

- 状态机图：



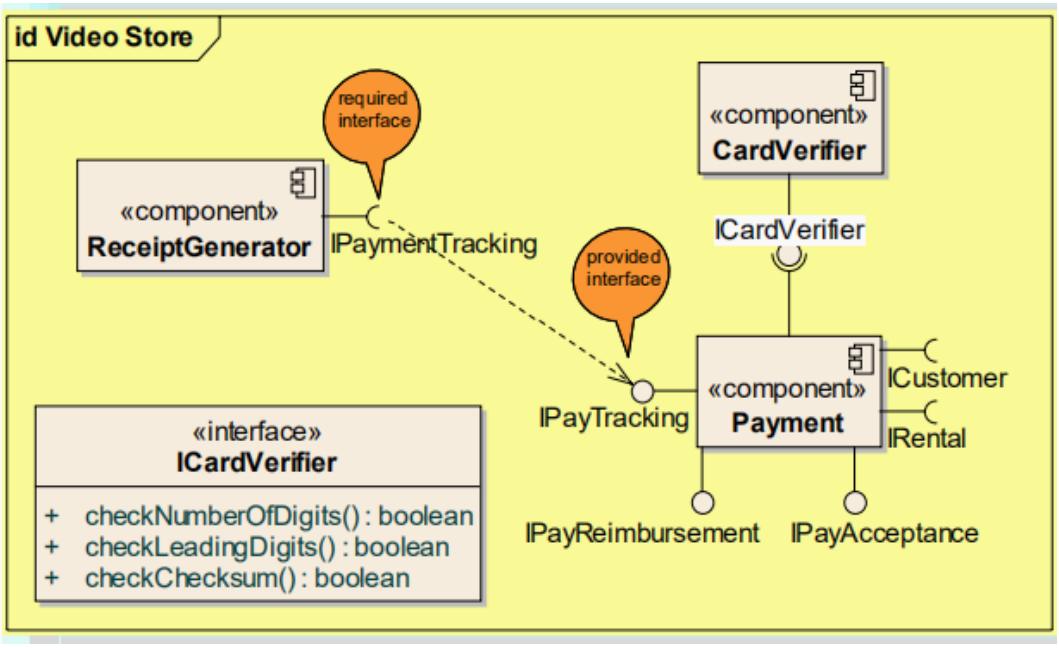

### 3.6 实现视图(The implementation view)

> UML为系统物理实现的体系结构/结构建模提供了工具，两种主要工具是构件图和部署图，这些图属于更广的结构图类型，结构图的主要代表是类图。
>
> 实现图属于物理建模的范畴，但定义它们一定要适当地考虑系统的逻辑结构。主要的逻辑构造块是类，主要的逻辑结构模型是类图。

- 子系统(Subsystem)和包(Packet)：
  - 分而治之：导致问题空间的分层模块化；
  - 子系统：
    - 定义：子系统的概念特殊化(继承)**构件**的概念，并被建模为构件的构造型；
    - 服务：子系统提供的服务是由其内部的组成成分即类所提供的服务的结果；
    - 子系统封装了意欲达到的系统行为的某些部分；子系统不能实例化；
    - 子系统的服务使用接口定义：
      - 封装行为及通过接口提供服务的益处：
        - 隔离变更；
        - 可替换的服务实现；
        - 可扩展性；
        - 可复用性；
    - 子系统可以在体系结构层被结构化，使得层之间的依赖是非循环的、且最小化；
      - 在每一层内，子系统可以嵌套；
  - 包：
    - 定义：具有指定名字的建模元素的分组；
    - 服务：包提供的服务是由其内部的组成成分即类所提供的服务的结果；
    - 与子系统不同，包不通过暴露接口而显露其行为，包可以直接被映射到程序设计语言元素结构；
    - 与子系统一样，包可以包含其他包，包拥有成员；
  - 总的来说，子系统是一个更丰富的概念，既包含包的结构化方面，又包含类的行为方面。行为由一个或多个接口提供，客户通过接口请求子系统的服务。
  -

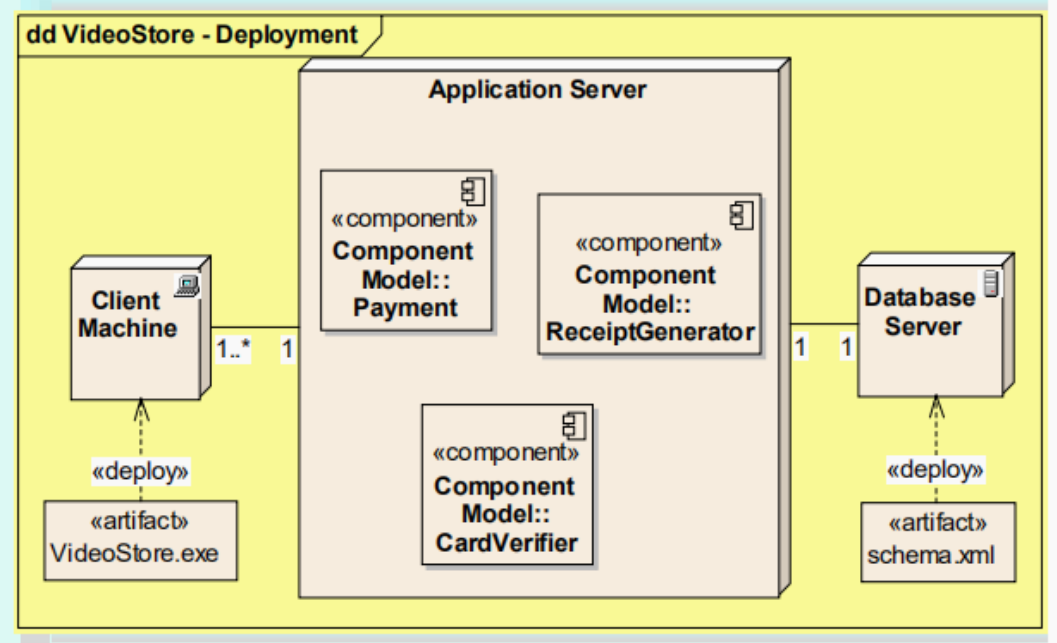



- 构件和构件图：
  - 构件定义：表示封装系统内容的模块化组成成分，并且系统的表示在其环境中是可替换的
    - 构件在所提供的和所依赖的接口方面定义其行为；
  - 接口：
    - 提供接口：由构件(或其他分类符，如类)实现了的接口，它表示构件的的实例为客户所提供的服务；
    - 依赖接口：指可能被构件(或其他分类符，如类)使用的接口，它指明一个构件为了执行它的功能并完成它自己向客户提供的服务所需要的服务；
  - 构件图**关注对所实现的系统的结构及构件的依赖性建模**；
  - 构件图(Component diagram)示例：




    - 构建符号强调构件的组成，可视化地组装提供的接口和需要的接口：
      - 提供接口被显示为附在构件上的小圆，依赖接口被显示为附在构件上的小半圆；
    - <>

- 节点和部署图：

- 节点是人工制品可以在上面部署运行的计算资源，可以将节点通过通信路径连接起来定义网络结构；
  - 人工制品是物理信息块的规格说明，由软件开发过程或系统的部署和运行使用或生成；
  - 通过关联将节点连接起来表示通信路径(关联端的重数值表示在关联中涉及的节点实例的数量)；
  - 可以是任何服务器、任何其它计算元素、构件及其它处理元素可用的人力资源；
- 部署图关注结构和节点依赖建模，节点定义系统的实现环境；
- 部署图定义构件和其它运行时处理元素在计算机节点上的部署；
- 

  - 立方体表示节点；
  - 一个人工制品到一个节点的<<deploy>>依赖表示部署；
    - 可以将人工制品符号放在节点符号内指示它们的部署位置；

---

## Chapter 4 需求规格说明 (Requirements Specification)

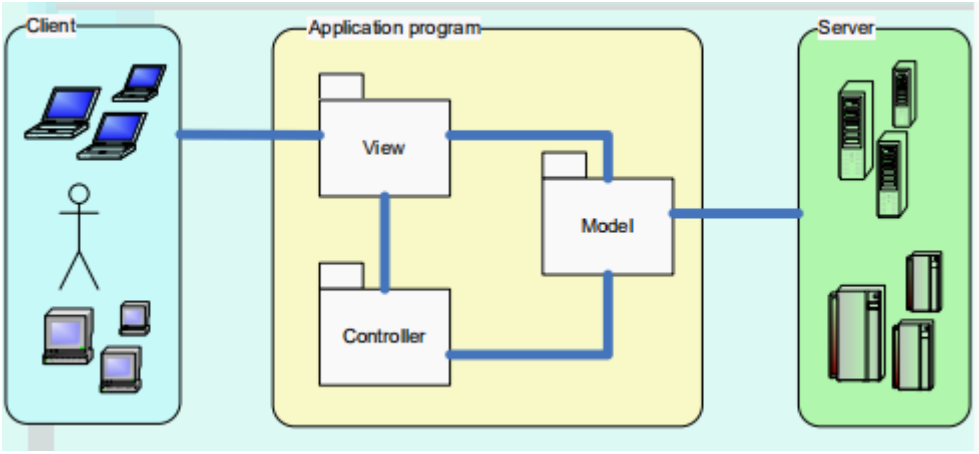
> 1. 需求规格说明以叙述性的用户需求作为输入，构造出规格说明作为输出；
> 2. 规格说明阶段的结果是扩展的(细化的)需求文档，通常将这个文档称为规格说明文档，它没有改变原始文档的结构，但对内容作了非常大的扩展；

### 4.1 体系结构优先权 (Architectural prerogatives)

> 1. 系统体系结构设计：自顶向下，由预先确定的体系结构框架(体系结构元模型)导出特定的设计；
> 2. 需求规格说明涉及对需求确定期间定义的客户需求进行严格的建模，重点放在那些系统将要提供的所期望的服务(功能性需求)上(在规格说明阶段通常不对系统约束(非功能性需求)做进一步的考虑，但系统约束可以指导和验证建模工作，这种指导和验证采用体系结构优先权的形式)；
> 3. 软件体系结构定义了系统中相互作用的软件构件及子系统的结构和组织形式
>    - “软件体系结构捕捉并保存设计人员关于系统结构和行为的设计意图，因此它提供了一种对设计的保护措施，以防止作为系统阶段的设计出现失败。软件体系结构是人们能对一个复杂系统的巨大复杂性进行思维控制的关键”；
> 4. 所有软件建模的最重要目标都是将构件依赖最小化
>    - 为了做到这一点，开发人员不能允许随意的对象通信，那会造成混乱的不可理解的构件通信网络；

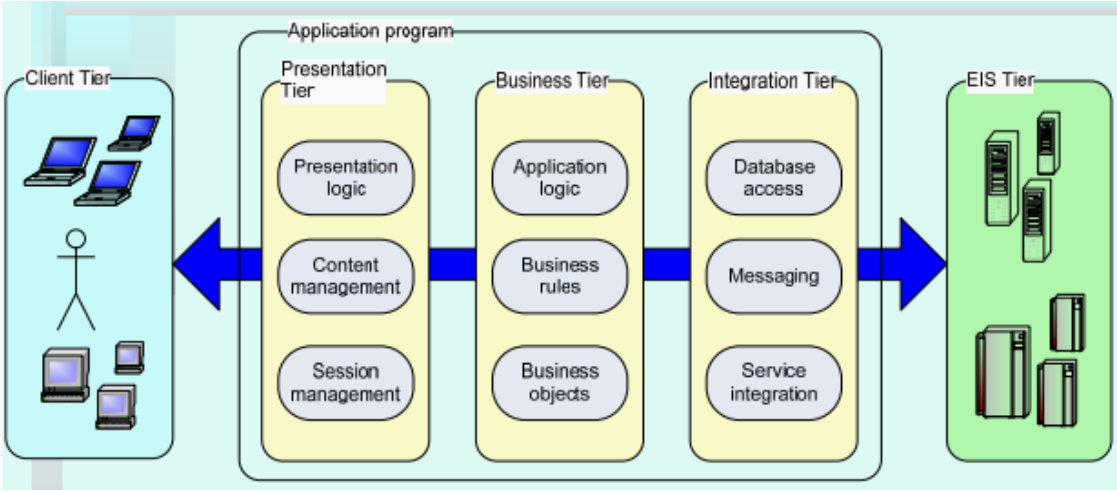
> - 在生命周期早期，就必须采用一种具有分层的构件和子系统、并严格限制对象通信的，清晰的体系结构模型；

- 模型-视图-控制器
  - 模型-视图一控制器(`MVC`)框架支持大多数现代体系结构框架及相关模式；
  - 模型：
    - *模型对象表示数据对象*——应用问题域中的业务实体和业务规则；
  - 模型对象的变化由事件处理通报给视图对象和控制器对象，这使用了发布者/订阅者技术:
    - 模型是发布者，因此它不知道它的视图和控制器;
    - 视图对象和控制器对象订阅模型对象，但它们也能引起模型对象的改变(为了辅助完成这项任务，模型提供了必要的封装了业务数据和行为的接口);
  - 视图：
    - *视图对象表示用户界面*(`UI`)对象(模型的状态以用户需要的格式呈现在用户的图形界面上);
  - 视图对象从模型对象分离出来;
    - 视图订阅模型，以便得到模型变化的通知，然后更新其显示;
    - 视图对象可以包含子视图，来显示模型的不同部分;
    - 每个视图对象一般都有一个控制器对象相匹配;
  - 控制器：
    - *控制器对象表示鼠标和键盘事件*;
  - 控制器对象响应视图发出的请求，该请求是用户与系统交互的结果;
    - 控制器对象给键盘击键、鼠标击键等赋予意义，并将其转换成对模型对象的动作;
    - 控制器是视图对象和模型对象的中间媒介;
    - 控制器对象能从可视化的界面中分离出用户输入，从而允许在不修改`UI`显示的情况下，改变系统对用户动作的响应，反之亦然——改变`UI`，而不改变系统的行为;
  - 参与者(用户)角度描述`MVC`对象之间的通信：




    - 直线表示对象间的通信；
  - 用户GUI事件被视图对象截取，传递给控制器对象进行解释并做进一步的动作；
- `J2EE`的核心体系结构

> `MVC`几乎是所有现代框架的 骨架，例如，`J2EE`的核心体系结构就是这样的框架




  - 用户通过客户层与系统交互；
  - `EIS`层(也称资源层)是任意的持久信息传递系统；
  - 用户通过表示层(也称Web层或服务器端表示层)来访问应用程序：
    - 在基于Web的应用系统中，表示层由用户界面代码和运行于Web服务器与/或应用服务器上的过程组成；
    - 参考`MVC`框架，表示层包括视图构件和控制器构件；
  - 业务层包含表示层中的控制器构件还没有实现的一部分应用逻辑
    - 负责确认和执行企业范围内的业务规则和事务；
    - 管理那些已从`EIS`层被加载到应用高速缓存中的业务对象；
  - 集成层承担着建立和维护与数据源连接的单一职责；

- 表示-控制器-bean-中介者-资源
  - `PCBMER`：表示、控制器、bean、中介者、实体、资源
  - `PCBMER`核心体系结构

    > 1. 层表示为`UML`节点，带箭头的虚线表示依赖关系；
    > 2. `PCBMER`的层次不是严格线性的，上层拥有的相邻下层可以不止一个(而且邻近层可以是叶子内(inter-leaf)的——它可以没有下层)；

    - bean层表示那些预先确定要呈现在用户界面上的数据类和值对象；
    - 表示层表示屏幕以及呈现bean对象的`UI`对象：
      - 当bean改变时， 表示层负责维护表示上的一致性，因此它依赖于bean层；
      - 这种依赖可以用两种方法来实现一使用拉模型直接调用方法(信息传递)来实现，或者使用推模型(或推拉模型)通过消息传递及事件处理来实现；
    - 控制器层表示应用逻辑(控制器对象响应`UI`请求);
    - 实体层响应控制器和中介者：
      - 由描述“ 业务对象”的类构成，存储着(在程序的内存空间中)从数据库取回的对象或者为了存入数据库而创建的对象(很多实体类都是容器类)；
    - 中介者(Mediator)层建立了充当实体类和资源类媒介的通信管道:
      - 该层管理业务处理，强化业务规则，实例化实体层的业务对象，通常还管理应用程序的高速缓冲存储器;
      - 在体系结构上，中介者服务于两个主要目的：
        - 首先，它隔离了实体层和资源层，这样两者的变化能够独立地引入；
        - 其次，当控制器发出数据请求但它并不知道数据是已经加载到内存还是仍存在数据库中时，中介者在控制器层和实体/资源层之间充当媒介；

        - 资源层负责所有与外部持久数据资源(数据库、Web服务等等) 的通信
    - `PCCBMER`核心框架的优点：
        - 层之间相关性的分离；
        - 去除了依赖关系的循环；
        - 确保了相当高的稳定性；
    - `PCBMER`的体系结构原则：
        - `DDP`: 向下依赖原则
        - `UNP`: 向上通知原则(促进自底向上通信的低耦合)
        - `NCP`: 相邻通信原则
        - `EAP`: 显式关联原则
        - `CEP`: 循环去除原则
        - `CNP`: 类命名原则
        - `APP`: 相识包原则(相识包由对象在方法调用参数中传递的接口组成，而不是具体的对象组成); 有效实现非相邻层之间的通信

**4.2 状态规格说明**

> 1. 对象的状态由它的属性值和关联决定(因为对象状态是由数据结构确定的，所以数据结构模型就称为状态规格说明);
> 2. 状态规格说明提供系统的结构(或静态)视图(主要任务是定义应用领域的类、它们的属性以及与其他类的关系，类的操作在一开始一般不予考虑，将来从行为规格说明模型中导出);
> 3. 在通常情况下，我们首先识别实体类(业务对象)， 即定义应用领域的类和将在系统数据库中永久存在的类；至于那些服务于用户事件的类(控制器类)、表示GUI呈现的类(表示类)， 以及管理GUI要表示的数据的类(bean类)，都要等到系统的行为特征已经确定后才建立；

- 类建模：
    - 面向对象系统开发的基础；
    - 高度增量迭代式的；
    - 发现类：
        - 名词短语(Noun phrase)方法(假定需求文档是完整而正确的)：
            - 相关类
            - 模糊类`Fuzzy`： 其分组结果(无关类or相关类)是类模型好与坏的区别所在
            - 无关类
        - 公共类模式方法(Common class pattens，根据通用的对象分类理论来导出候选类)：
            - 概念类；
            - 事件类；
            - 组织类；
            - 人员类；
            - 地点类；

                > 公共类模式方法与特定用户需求的关系太松散而不能提供全面的解决方案

        - 用例驱动方法`use case driven approach`：
            - Similar to the noun phrase approach
            - Relies on the completeness of use case models
            - Function-driven (Problem-driven)
        - 类-职责-协作者(`CRC`)：
            - 涉及头脑风暴式的集体讨论，通过使用一种特殊制作的卡片使其简单易行：

        - 类名写在最上面的分栏中，类的职责列在左边的分栏中，协作者列在右边的分栏中；

          > 职责是当前类为满足其他类所准备执行的服务，很多要履行的职责需要其他类的协作(服务)，这些类被列为协作者

      - 混合方法`Mixed approach`：
        - Middle-out rather than top-down or bottom-up
      - 发现类的指南
        - Statement of purpose(目的陈述)
        - Description for a set of objects
          - 单例类
        - Houses a set of attributes
          - Identifying attributes - keys
          - `OID`
        - Class or attribute
        - Houses a set of operations
    - 类说明：
      - 类命名
      - 发现和说明类的属性
- 关联建模

  > `PCBMER`框架的`EAP`原则，主张程序中显式关联的重要性。该原则要求如果两个对象在运行时进行通信，那么它们就应该存在编译时关联。

    - 发现关联`Discovering associations`
    - 说明关联`Specifying associations`：
      - 给关联命名
      - 给关联的角色命名
      - 确定关联的多重性
- 聚合及复合关系建模(aggregation and composition modeling)

  > - Four semantics of aggregation:
  >   - `ExclusiveOwns`聚合
  >     - 存在依赖性
  >     - 传递性 Transitivity
  >     - 非对称性(非自反性) `Asymmetricity`
  >     - 固定性
  >   - `Owns`聚合
  >     - No fixed property
  >   - `Has`聚合
  >     - No existence-dependency
  >     - No fixed property
  >   - `Member`聚合
  >     - No special properties except membership

    - 发现聚合和复合
      - Discovered in parallel with discovery of associations (聚合是在发现关联的同时被发现的)

        - 检验方法：
            - has
            - is part of
        - Can relate more than two classes
    - 说明聚合和复合(更强形式的聚合)
        - `UML` supports:
            - 聚合：
                - "通过引用"语义：复合对象并不物理地(是逻辑的)包含部分对象
                - 空心钻石
                - `Has`和`Member`聚合
            - 复合\组合：
                - "通过值"语义：复合对象并物理地包含部分对象
                - 实心钻石
                - `ExclusiveOwns`和`Owns`聚合
    - 示例
- 泛化关系建模

> 1. 泛化：一个或多个类的公共特性(属性和操作)可以抽象到一个更一般化的类中；
> 2. 泛化将一般类(超类)与特殊类(子类)连接起来
> 3. 泛化允许超类的特性被子类继承(复用)
> 4. 目的：继承、可替换性、多态性

    - 发现泛化
    - 说明泛化
    - 示例
- 接口建模

> 1. 接口无属性(除了常量)、关联或状态，只有操作，并且所有操作都隐含是公共的和抽象的
> 2. 一个接口可以和另一个接口具有泛化关系，这意味着一个接口可以通过继承另一个接口的操作来扩展另一个接口。

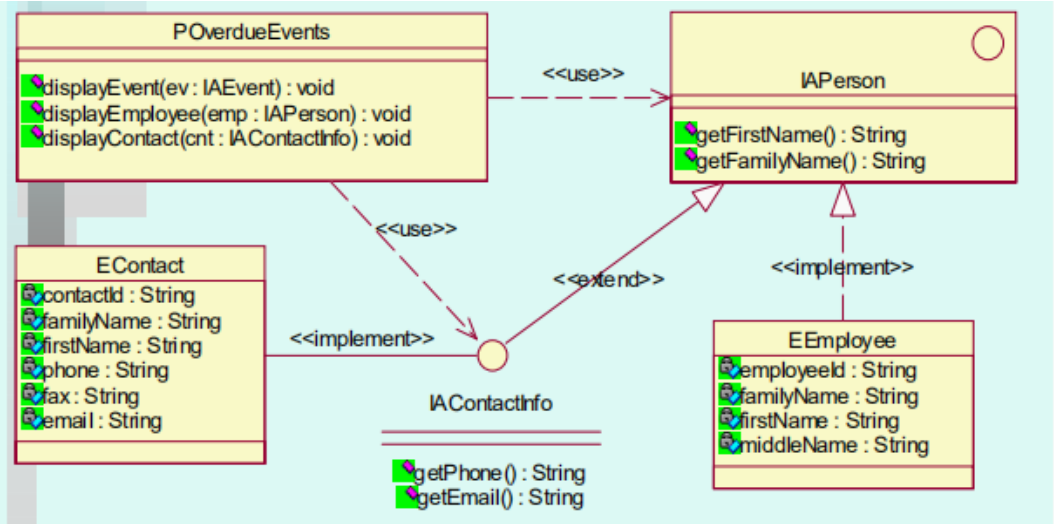
    - 发现接口

> 接口不是在对问题域的分析中发现的，接口与使用合理的建模规则来构建健壮的和可支持的系统有关。通过区分使用接口的类与实现接口的类，接口使系统更容易理解、维护和演进。
>
> 接口是加强体系结构框架的基础。它们可以用于打破系统内的循环依赖，实现发布者/订阅者事件方案，对未注册的客户端隐藏实现等等。一般情况下，接口只显示一个实际类的有限行为。

    - 说明接口

> - 使用象征类的矩形加上关键字<>来构造表示，也可以用矩形右上角的小圆圈代替，也可以显示为一个小圆圈，小圆圈写上接口的名字；
> - 一个类使用(依赖)接口，用指向接口的一条虚线箭头来表示；为了表示得更清楚，箭头可以加上构造型关键字<>；
> - 一个类实现(提供)接口，用一条末端带有三角形的虚线来表示；为了表示得更清楚，虚线可以加上构造型关键字<>，该类必须实现接口支持的所有操作；

    - 实例：




- 对象建模

#### 4.3 行为规格说明

- 用例建模
  - 分析员在发现用例时，必须要遵循用例概念的本质特性；
  - 一个用例表示：
    - 一个完整的功能，包括逻辑的主流、关于它的任何变化(子流)以及任何例外条件(备选流)；
    - 一个外部可见的功能(不是内部功能)；
    - 一个正交功能；
    - 由一个参与者启动的一个功能；
    - 给参与者传递确切值的一个功能；
  - 说明用例：
    - 用例规格说明包括参与者、用例和下面4种关系的图形表示
      - 关联
        - 建立参与者和用例之间的通信渠道；
      - 包含 <<include>>
        - 允许将被包含用例中的公共行为分解出来；
      - 扩展 <<extends>>
        - 通过在特定的扩展点激活另一个用例来扩展一个用例的行为；
      - 泛化
        - 允许一个特殊化的用例改变基础用例的任何方面；
- 活动建模
  - 活动图
- 交互建模
  - 顺序图和交互图
- 操作建模
  - 类将一组操作作为服务提供给系统中的其它类，这组操作就确定了类的公共接口，声明为公共可见性；
  - 发现类操作
    - 从顺序图中发现类操作
      - 顺序模型的每一个消息，都必须有目标对象的一个操作为其服务；
    - CRUD操作(创建、读取、更新、删除)；

- 说明类操作

#### 4.4 状态变化规格说明

- 对象状态建模
  - 状态图

---

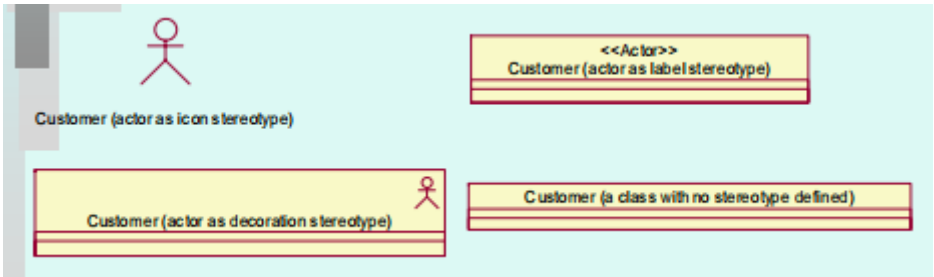
## Chapter 5 从分析到建模 (Moving from Analysis to Design)

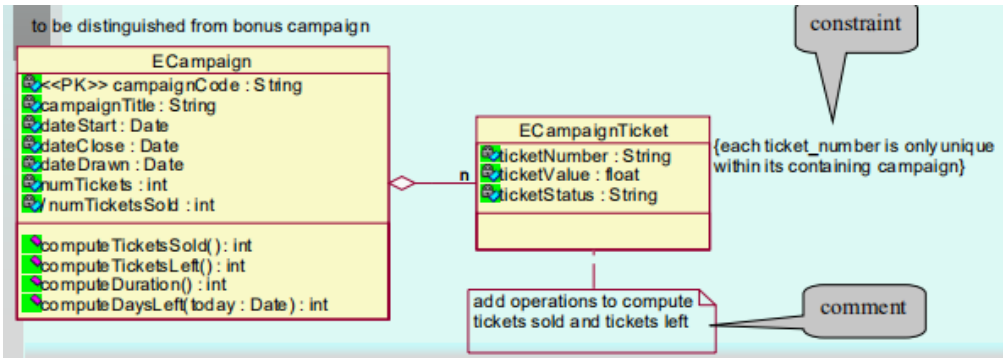
#### 5.1 高级类建模 (Advanced class modeling)

> 构造型(Stereotypes)、约束(Constraints)、导出信息(Derived information)、可见性(Visibility)、限定关联(Qualified associations)、关联类(Association class)、参数化类(Parameterized class)，etc.

- 扩展机制(Extension mechanisms)
  - Specify "How specific UML model elements are customized(定制的) and extended with new semantics by using *Stereotypes*、*Constraints*、*tag definitions(标签定义) and tagged values(标签值)*"
  - UML profile (UML 配置文件):
    - Extends a reference metamodel(扩展参考元模型，e.g UML);
    - Coherent(一致的) set of extensions, define for specific purpose;
  - Stereotypes (构造型)
    - Extends an existing UML modeling element to vary the semantics of an existing element (构造型对现有的UML建模元素进行扩展，使现有元素的语义多样化);
    - It is not a new model element;
    - 它没有改变UML的结构——只是丰富 了现有表示法的含义；它还支持模型的扩展和定制
    - 在模型中，构造型通常用双尖角括号括住的一个名字来表示，如

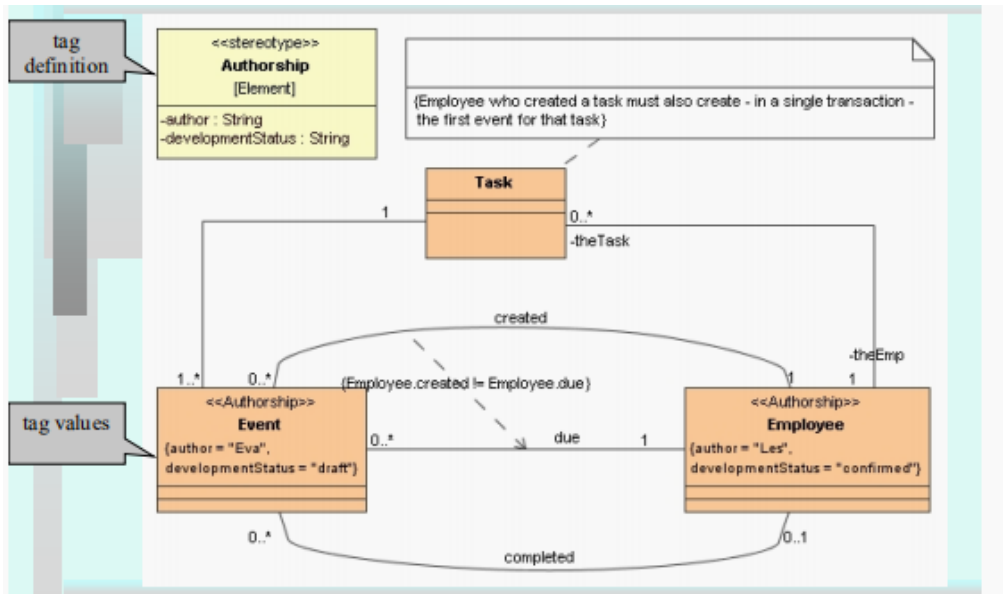
      `<<global>>`、`<<PK>>`、`<<include>>`

      
      
  - Comments and Constraints (注释和约束)
    - Comments: 从属于一组元素的文本解释
      - 提供了给元素添加各种解释的能力；
      - 对语义没有影响，但可能包含有助于建模者的信息；
      - 注释显示为一个右上角卷起的矩形，矩形包含着注释内容，与被注释元素用一条单独的虚线连接起来；
    - Constraint: 指条件或限制，是对一个元素某些语义的声明，表示附加给被约束元素的额外语义信息；
      - 约束是一个断言，表示必须被正确的系统设计所满足的一个限制；
      - 可以用自然语言文本或机器可读语言来表达；

        - 约束显示为一对大括号({}) 中的文本字；
    - 注释和约束的区别不在于表示法，而在于语义结果：
        - 注释对模型语义没有作用，它只是对建模决策的附加说明；
        - 约束对模型具有语义含意，并且(在理论上)应该用形式化的约束语言来描述；
            - 实际上，`UML`为此提供了一个预定义语言， 称为对象约约束语言(OCL)；
    - 在模型图中，只显示简单的注释和约束，更详细的注释和约束(因为描述太长不能显示在图形模型上)作为文本文档存储在CASE资源库中；
    - 

- Tags (标签)
    - 标签定义是构造型的一个特性，显示为含有构造型声明的类矩形中的一个属性；
    - 标签值是一个名-值对，附属于一个模型元素，该模型元素使用了包含标签定义的构造型；
    - 标签与约束相似，表示模型中的任意文本信息，写在大括号中，形式如下：

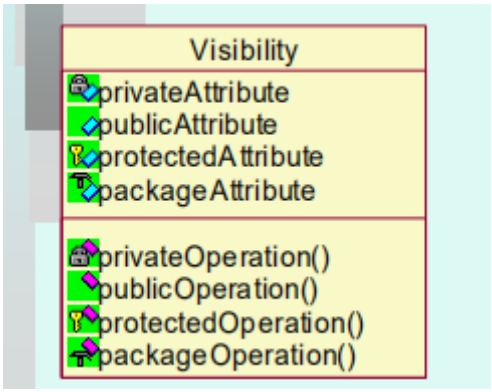
        > tag = value
        >
        > {analyst = Les, developmentStatus = confirmed}

    - 同构造型和约束一样， 几乎没有标签在`UML`中预定义；
    - 典型应用是提供项目管理信息；
    - 

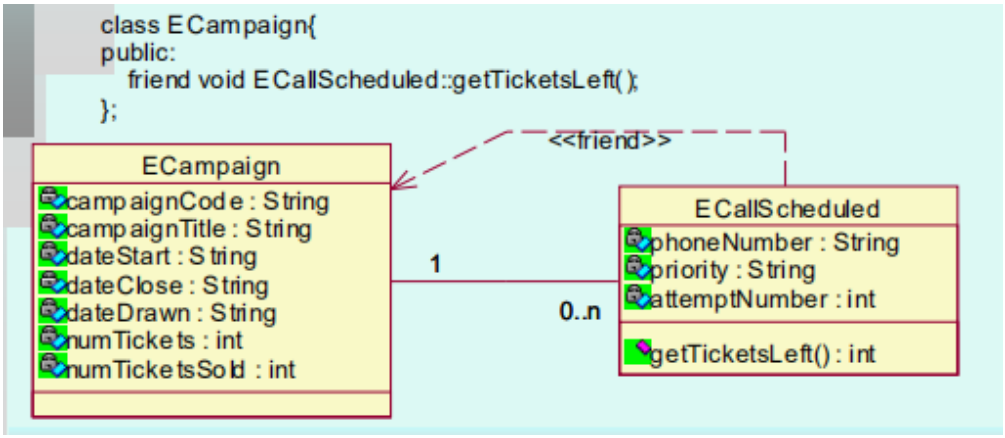

- 可见性与封装 (Visibility and encapsulation)

> 应用于类属性和操作的整套可见性标志是:
>
> + 公共可见性
>
> - 私有可见性
>
> # 保护可见性

> ~ 包可见性



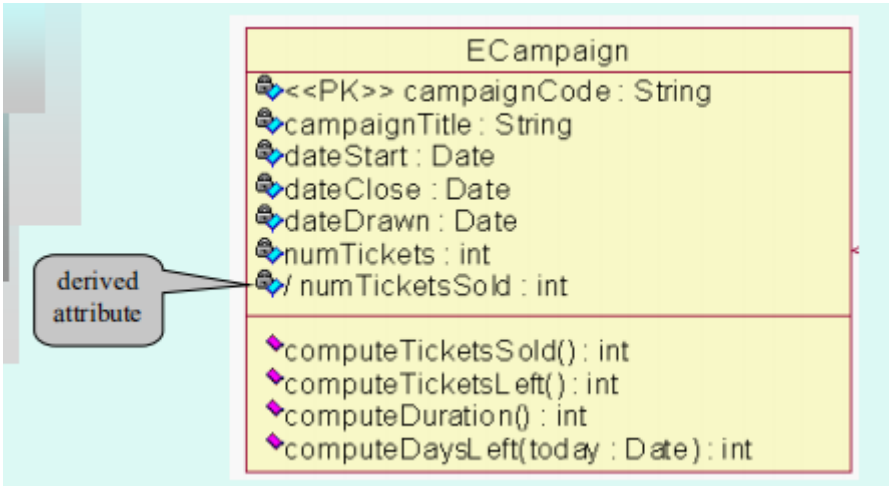

  - 保护可见性：
    - 应用于继承的情形下；
  - 继承来的类特性的可见性：
    - C++；
  - 包可见性和友元可见性：
    - 包可见性意味着包中所含的所有其他类都可以访问这样的一个属性或操作，但是对所有其他包的类来说，这个属性、操作或类看起来是私有的；
    - C++友元可见性；

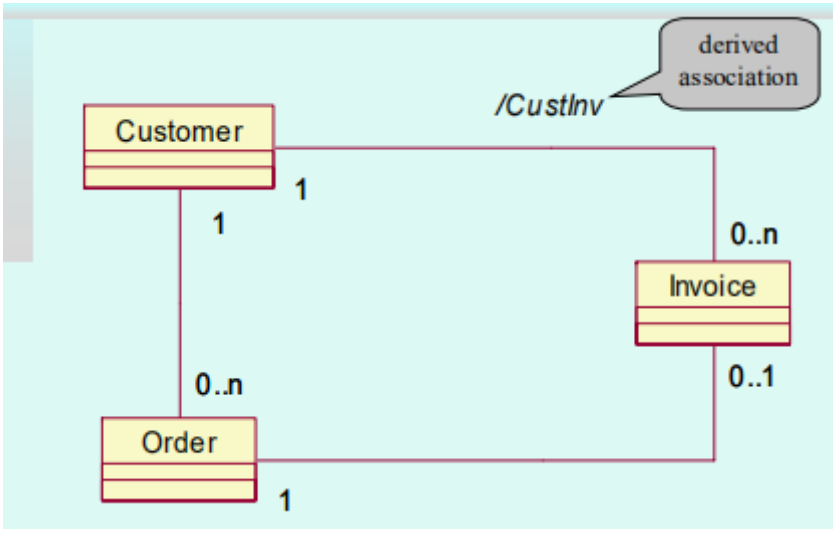



  - 关联的可见性：
    - 应用于角色名(因为角色名(在 UML 2.0中，称之为关联端名)实现为类属性，所以角色名的可见性表示这个结果属性能否用在跨越关联的访问表达式中)；

- 导出信息 (Derived information)
  - 一种(最经常)应用于属性或关联的约束；
  - 导出信息从其他模型元素计算得到(严格地说，导出信息在模型中是冗余的——它可以在需要时计算出来)；
  - 导出信息并没有丰富分析模型的语义，但它可以使模型具有更强的可读性(因为一些通过计算才能得到的信息在模型中直接表达了出来)；
  - The knowledge of what information is derived is more important in a design model, where optimization of access to information needs to be considered;
  - 导出信息的 UML 表示法是在导出属性名或关联名的前面加一条斜线(/)；

> 导出属性与导出关联

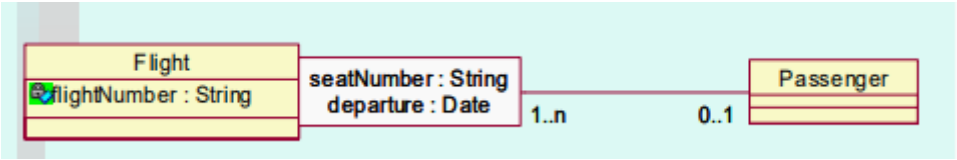



- 限定关联 (Qualified associations)

> 对限定关联这个概念有强烈的争议。有些建模者喜欢它，但另外一些则憎恨它。可以证明，不用限定关联，也可以构造出完整而有充分表现力的类模型。但如果要使用限定关联，就应该全面一致地使用。

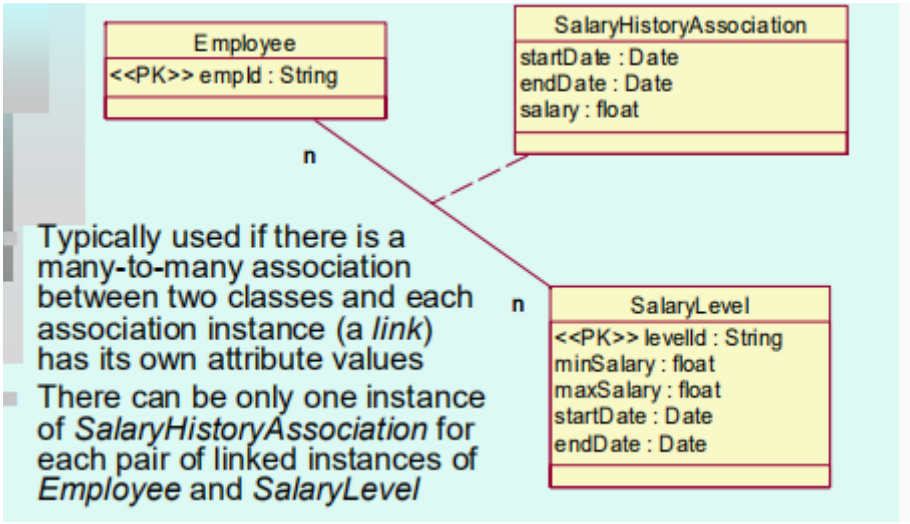
  - 限定关联是在二元关联的一端(一个关联可以在两端都被限定，但这非常少见)，有一个属性框(限定词)，框中包含一个或多个属性(这些属性可以作为一个索引码，用于穿越从被限定的源类到关联另一端的目标类的关联)；




  - 在正向穿越中，关联多重性表示与组合码(限定对象+限定值)相关的多个目标对象；在反向穿越中，多重性描述了被组合码(限定对象+限定值)标识的、与每个目标对象相关的多个对象.

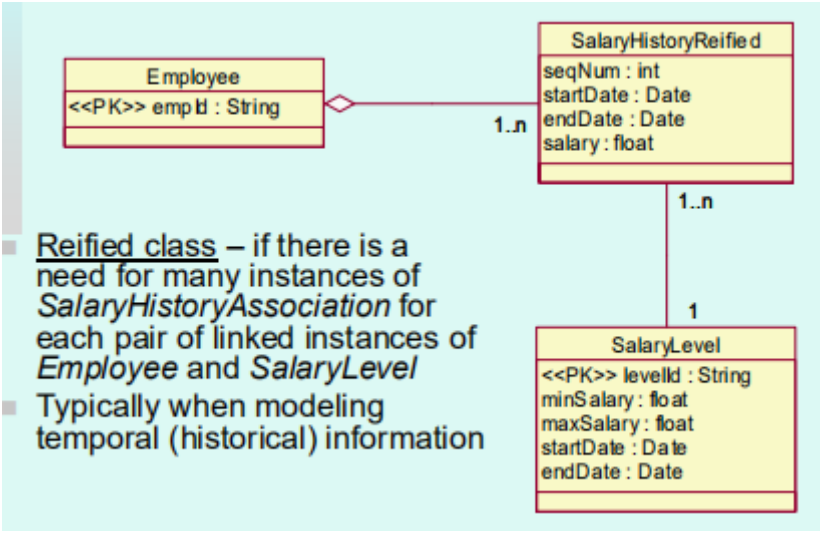
- 关联类与具体化类 (Association class and reified class)
  - 关联类：是一个关联，也是一个类
    - 如果两个类之间存在多对多的关联，并且每个关联实例(一个链接)都有它自己的属性值，这时通常使用关联类(为能够存储这些属性值);
    - 存在的约束：
      - 考虑一个在类A和类B之间的关联类C：约束是指，对于每一对链接起来的A和B的实例，只能存在一一个类C的实例；
  - 带关联类的模型：

> 与任何普通类的对象一样， 关联类的对象在实例化时也要指定`OIDs`；除了系统产生的`OID`外，对象也能用自己的属性值来标识：
>
> 在关联类的情况下，对象从那些指明它所关联的类的属性中取得它的标识(其他属性值对对象的标识不起作用);




- 带有具体化类的模型：

> 关联类对它所关联的类的对象的引用，不能有完全相同的




### 5.2 高级泛化与继承建模 (Advanced generalization and inheritance modeling)

> - Generalization is a useful and powerful concept, but it can also create many problems;
> - 泛化和继承这两个术语是相关的，但又不完全相同；
>   - 泛化是类之间的语义关系，它说明子类的接口必须包含超类的所有(公共的、包的和保护的)特性；
>   - 继承是"一种机制，通过它，较特殊的元素可以合并较一般元素中定义的结构和行为"；

- 泛化和可替换性 (Generalization and substitutability)
  - 从语义建模的观点出发，泛化引入了额外的类，并将它们分为一般类和较特殊的类，在模型中建立超类-子类关系：
    - 虽然泛化引入了新的类，但它可以减少模型中关联关系和聚合关系的总数(因为发生在较一般类上的关联和聚合，还意味着存在与较特殊类的对象的链接)；
    - 根据所期望的语义，可以将来自一个类的关联或聚合链接到泛化层次中最一般的类上，由于子类可以替换它的一般类，所以子类对象具有超类的所有关联和聚合关系；
- 继承和封装 (Inheritance and encapsulation)

    - 封装(是针对类的概念，不是对象)要求，只能通过对象接口中的操作才能访问对象的状态(属性值);
    - 封装与继承和查询能力是正交的，它不得不与这两个特性一起折中考虑;
    - 继承允许子类直接访问保护属性，它削弱了封装；当计算涉及不同类的对象时，可能要求这些不同的类彼此是友元或者让元素具有包可见性，这就进一步破坏了封装;
- 接口继承 (Interface inheritance)
    - 接口继承除了其他优点外，在不支持多重实现继承的语言(如Java) 中，它还提供了一种达到多重实现继承的方法;
    - 子类继承属性类型和操作型构(操作名加上形式参数)，子类被说成是支持超类接口;
    - 继承来的操作的实现可以留待以后进行;
    - 接口和抽象类;
- 实现继承 (Implementation inheritance)

> 泛化可用于隐含可替换性，这一点后来能够由接口继承来实现；然而，泛化还可用于(故意地或不是故意地)隐含代码复用，并且这一点后来由实现继承来实现

    - 实现继承——也称为子类化、 代码继承或类继承——在子类中组合超类的特性， 必要时允许用新的实现重载它们;
        - 载是指在子类的方法中包括(调用)超类方法，并用新的功能来扩展它;
    - 实现继承允许共享特性描述、代码复用以及多态性;

> 当用泛化来建模时，必须清楚其中隐含了哪种继承:
>
> - 接口继承的使用是安全的，因为它只涉及契约部分的继承——操作型构;
> - 实现继承涉及代码的继承——实现部分的继承，如果不注意控制和限制，实现继承带来的坏处会比好处多;

    - 实现继承的恰当使用——扩展继承;
        - 继承的唯一恰当使用就是将继承作为类的增量式定义;
        - 子类拥有比超类更多的特性;
    - 实现继承的有问题的使用——限制继承;

> 在扩展继承中，用新的特性扩展子类的定义。然而，有一些继承来的特性在子类中被禁止(被重载)，因此使用继承作为一种限制机制也是可能的，这样的继承被称为限制继承。

    - 实现继承的不恰当使用——方便继承;
        - 任意选择一个类作为其他类的父类的继承称为方便继承;
    - 实现继承的危害(evils):
        - 脆弱的基类：see detail in book
        - 重载和回调：see detail in book
        - 多重实现继承：see detail in book

### 5.3 高级聚合与委托建模 (Advanced aggregation and delegation modeling)

- 给聚合增加更多的语义
- 作为泛化的可选方案的聚合
- 聚合与整体构件

### 5.4 高级交互建模 (Advanced interaction modeling)

> 1. 顺序图比通信图更常用;
> 2. 顺序图和通信图表达了相似的信息，但顺序图关注消息的顺序，而通信图则强调对象的关系;

> 3. CASE工具倾向于为顺序图的可视化提供更好的工具，更好地支持顺序图中的复杂交互；
> 4. 通信图有一个显著的优点，它们能够在开会和讨论时用作手绘的草图：
>
>    在这种面对面的情形下，通信图比顺序图具有更高的空间效率，更容易修改

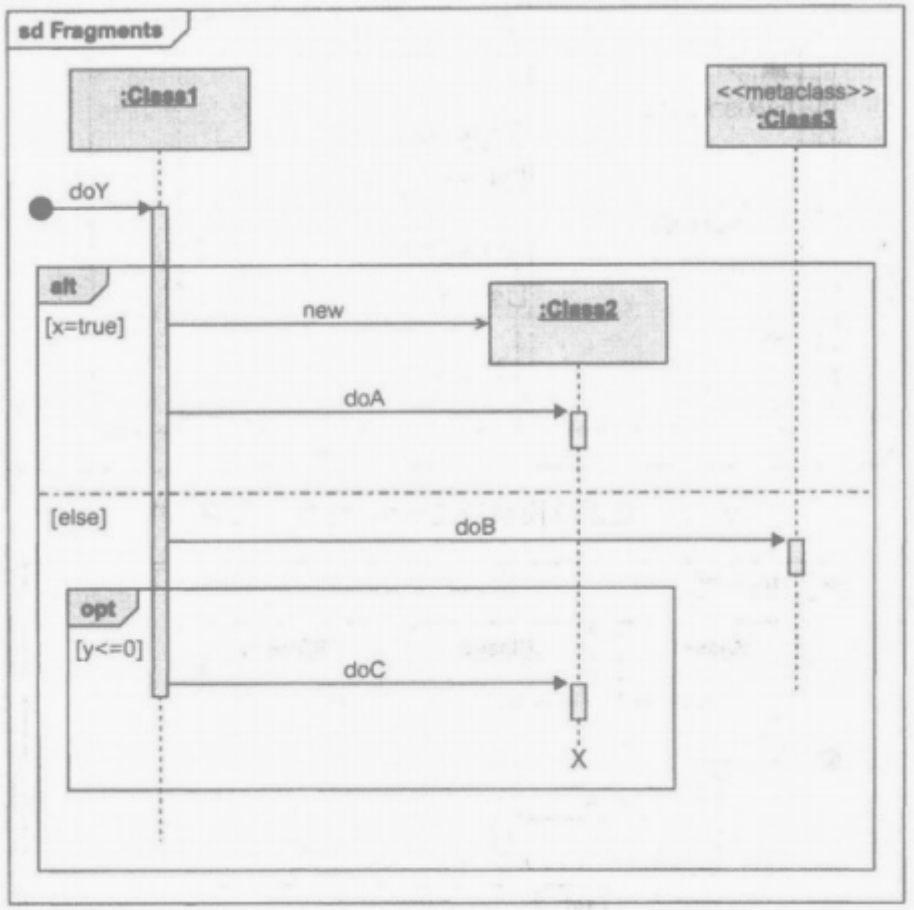
- 生命线(Lifelines)和消息(Messages)
  - 在交互图中，最明显的两个概念是生命线和消息；
  - 一条生命线代表一个交互参与者，它表示在交互过程中一个特定时间段内，对象是存在的；
    - 生命线显示为命名的矩形框加上延伸的垂直直线(通常是虚线)，生命线矩形框能够被命名来表示：
      - 一个类的未命名实例；
      - 一个类的命名实例；
      - 一个类，即元类的一个实例；
      - 一个接口；
  - 消息表示在交互的生命线之间的通信，接收到消息的生命线/对象激活了相关的操作/方法；
    - 交互建模中允许的多种消息类型：
      - 同步消息，其中调用者阻塞，即它等待响应，用实心箭头表示；
      - 异步消息，其中调用者不阻塞，因此允许多线程执行，用开放箭头表示；
      - 对象创建消息常常(但不是必须)用关键字(如new或create) 来命名，表示为带开放箭头的一条直线；
      - 回复消息，将交互的输出值传递给发起动作的调用者，用一条带有开放箭头的虚线表示；
        - 回复消息只是用来显示消息返回结果的两种方法中的一种，另一种是在消息语法中指明返回变量；
        - 依赖于所画图的抽象级别，消息的返回和回复可以显示，也可以不显示

      > 发现消息：没有指定发送者的消息，换句话说，发现消息的来源在模型表示的范围之外；
  - 对象销毁：X (通常假定，在交互模型中创建的对象也将要在同一个交互模型中被销毁)；
  - 在生命线代表的是接口或抽象类的情况下，很显然，被调用的方法是从一个实现了接口的类或一个继承了抽象类的具体类执行；在这两种情况下，推荐的做法是为接口的相关实现或者每个多态的具体类，都提供单独的交互图；
  - 当控制流聚集到一个对象时，在`UML` 2.0中称为执行规格说明(以前称为激活)

> - 说明基本技术(Accounting for basic technology)
> - 有助于交互建模的可视化技术

- 片段(fragments)：
  - 一段交互称为交互片段(interaction fragment)；
  - 交互可以包含更小的交互片段，称为组合片段(combined fragment)，组合片段的语义由交互操作符(interaction operator)确定；
  - UML 2.0预定义的最为重要的操作符：
    - alt: 可选片段，在守卫条件中表达if then else条件逻辑；
    - opt: 选择片段，如果守卫条件为真，执行该片段；
    - loop: 循环片段，服从循环条件而重复多次的片段；
    - break: 中断片段，如果中断条件为真，就执行中断片段，而不执行外围片段剩余的部分；
    - parallel: 并行片段，允许所包含行为交替执行；

- Example：




- 交互使用

---

## Chapter 6 系统体系结构与程序设计 (System Architecture and Program Design)

> 1. 系统设计包括的两个主要问题：系统体系结构设计和系统中程序的详细设计；***
> 2. 体系结构设计是从系统的模块方面对系统进行描述，包括确定系统的客户机构件和服务器构件的解决方案策略；*体系结构定义类与包的分层组织、将进程分配给计算设施、复用和构件管理*;
> 3. 体系结构设计解决多层物理体系结构及多层逻辑体系结构的有关问题；
> 4. 对每个模块(用例)内部工作的描述称为详细设计：详细设计为每个模块开发完整的*算法和数据结构*，这些算法和数据结构是针对底层实现平台的所有(加强的和明显的) 约束专门设计的。
> 5. 详细设计描述协作模型，协作模型是实现*从用例中捕获的程序功能*所需要的

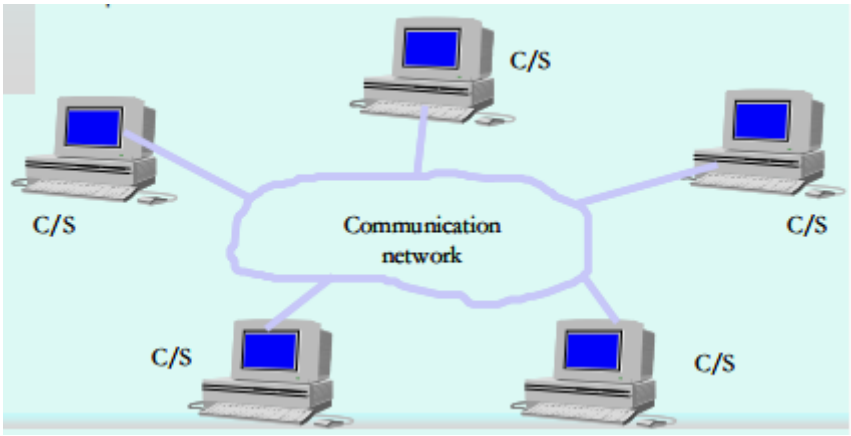
---

### 6.1 分布式物理体系结构 (Distributed Physical Architecture)

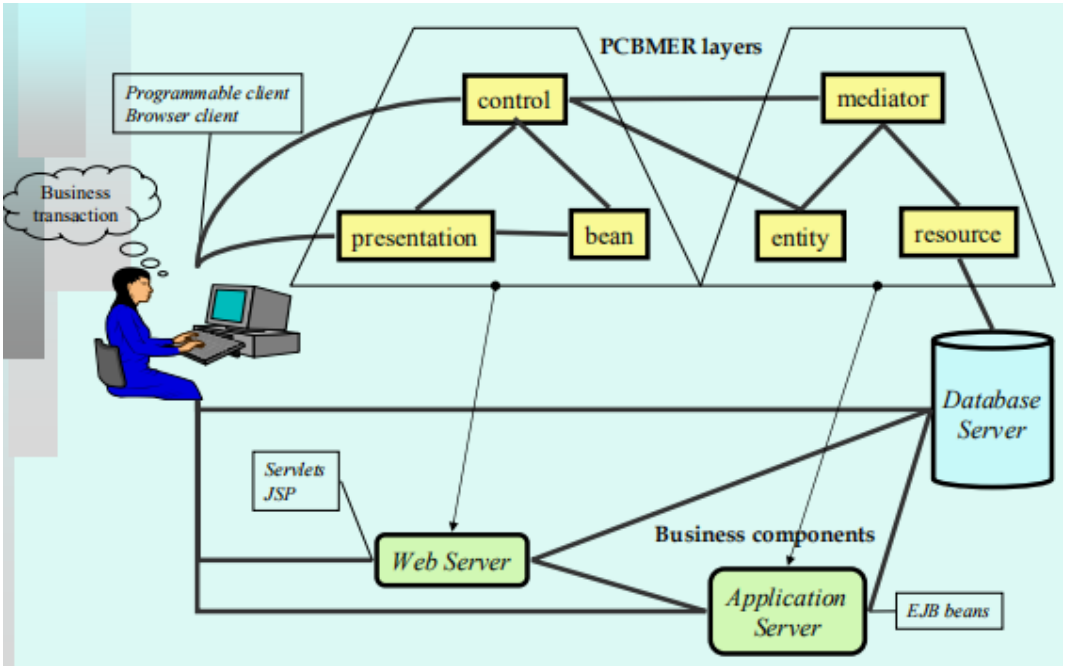
> 1. 体系结构设计具有物理和逻辑两个方面；***
> 2. 物理体系结构设计关注部署方案的选择以及系统的工作负荷在多处理器上的分布；
> 3. 物理体系结构解决客户机和服务器问题，以及“ 粘结”客户机和服务器所需要的任何中间件问题； *它将处理构件分配给计算机结点*;
> 4. 从`UML`建模的角度，物理体系结构设计使用结点和*部署图*;
> 5. 虽然物理体系结构解决客户机和服务器问题，客户机和服务器是逻辑概念：

    - *客户机是请求服务器进程的计算进程，而服务器是服务于客户请求的计算进程;*
    - 客户机和服务器进程通常运行于不同的计算机上;
6. 在典型情况下，客户进程负责控制在用户屏幕上的信息显示，并处理用户事件；服务器进程是任何带有数据库的计算机结点，客户进程可能会请求数据库的数据或处理功能。
7. 可以对客户机/服务器(C/S) 体系结构进行扩展来表示任意的分布式系统 ***，分布需求可能来自很多因素，如:
    - 在指定机器上做专门处理的需求;
    - 从不同的地理位置访向系统的需求;
    - 经济上的考虑——使用多台小机器会比使用一台大型、昂贵的机器更便宜;
    - 适应性需求——确保将来对系统的扩展能够很好地按比例增加;
8. 在分布式处理系统中，客户机能够访问任意多台服务器，然而在同一时间，只允许客户机访问一台服务器(意味着在一个请求中不可能将来自两个或更多数据库服务器的数据组合起来);
9. Distributed database system (分布式数据库系统) - when it is possible in a single request to combine two or more database servers;

- 对等体系结构 (Peer-to-peer architecture, `P2P`):
    - Distributed processing system in which any process or node in the system may be both client and server;
    - 定义了一种单独类型的系统元素——同位体;
    - Special consideration - minimization of the network traffic while maximizing the overall system throughput;




- 分层体系结构 (multi-tier architecture):
    - 定义计算层次:
        - 层次结构中间的每一层即是客户机又是服务器;
        - 每一层只能作为层次结构中下一层的客户机，也只能作为体系结构中更高层调用者的服务器;
    - 三层体系结构：单独的业务逻辑中间层位于`GUI`客户机和数据库服务器之间;
    - 由于分层体系结构在硬件层与软件层之间采用层次依赖方法，故它不同于对等体系结构;

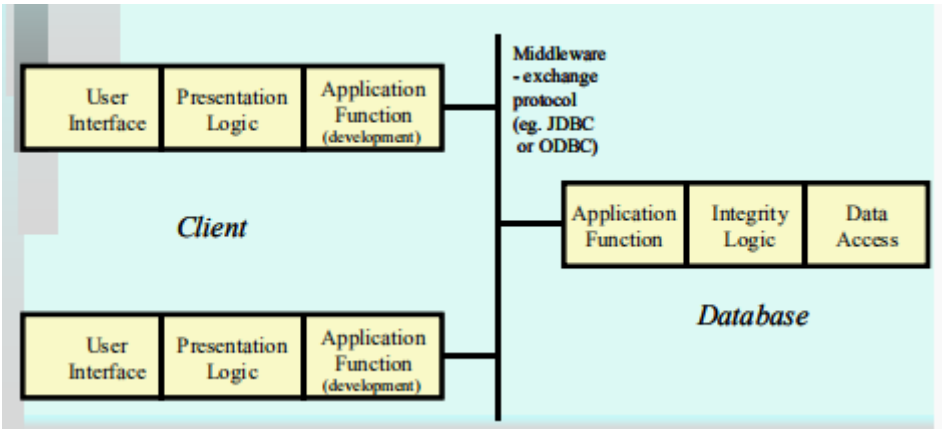



- 应用服务与业务服务

  > For database community:
  >
  > 应用服务是每一个运行在数据库上的单独程序所做的；
  >
  > 业务服务是数据库强迫所有运行程序都要遵循的业务规则；

- 胖客户体系结构(类固醇客户机，“a client on steroids”)：应用逻辑被编译到客户机；
- 瘦客户体系结构(干瘦客户机，“a skinny client”)：应用逻辑被编译到服务器；
- 中介者体系结构：一部分应用逻辑被编译到客户端，另一部分被编译到服务器；

- 数据库为中心的体系结构 (Database-centric architecture)：




- 用户界面：如何一特定的 `GUI` 显示信息；
- 表示逻辑：按照应用程序功能的需求负责处理 `GUI` 对象(表单、菜单、动作按钮等)
- 应用功能：包括程序的主要逻辑(捕获应用程序做什么，客户机和服务器的粘合剂)；
- 完整性逻辑：负责企业范围内的业务规则，这些规则应用于所有的应用程序；
- 数据访问：如何访问磁盘上的持久数据；

---

### 6.2 多层逻辑体系结构 (Multilayer logical architecture)

> Without a clear architectural design and rigorous processer, large software projects are likely to fail.
>
> Unrestricted communication between objects cannot be permitted in system modeling

> - Networks of intercommunicating objects are bad news: in networks, the number of communication paths between objects grows exponentially with the addition of new objects.
>
> Successful systems are organized in hierarchies
>
> - Hierarchies reduce the complexity from exponential to polynomial: in a typical hierarchy, only objects in adjacent layers communicate directly.

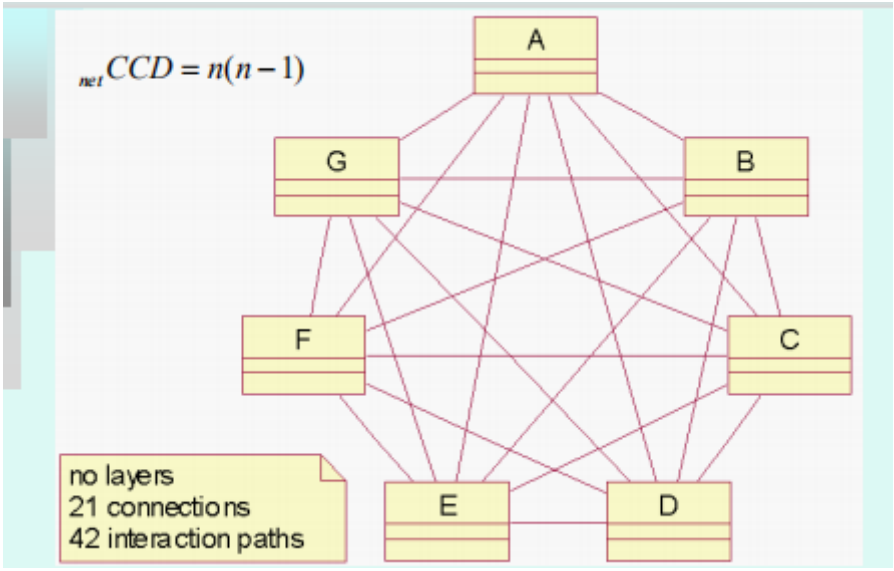
- 体系结构的复杂性：
  - 问题复杂性(Problem complexity)：问题域本身的复杂性，也称为计算复杂性 (软件本质特性的一个分支)；
  - 算法复杂性(Algorithmic complexity)：目标是度量软件算法的效率；
  - 结构复杂性(Structural complexity)：目标是建立软件结构之间的关系及易于维护和易于演化，*度量被应用到软件对象之间的依赖*；
  - 认知复杂性(Cognitive complexity)：度量理解软件所需要的努力，即捕获程序的逻辑流，并度量*逻辑流*的各种特性；

---

> - 空间认知复杂性(Spatial cognitive complexity)：
>   - 认知复杂性的度量：空间复杂性度量，其目标是
>     - 度量软件工程师为了构造软件的智力模型而必须在代码中移动的距离；
>   - 两种空间复杂性度量：函数的(过程的)和面向对象的(不介绍)
>   - 导出复杂性的步骤：
>     - 第一步，计算车程序中每个函数的复杂性值；
>
>       $FC = \sum_{i=1}^{totalcalls} dist_i$
>
>       > `totalcalls`: 调用函数的次数；
>       >
>       > `dist`: 距离，指从函数调用到函数定义的代码行数；
>       >
>       > `FC`: 函数空间复杂性；
>
>     - 第二步，累加得到整个程序的复杂性值
>
>       $PC = \sum_{i=1}^{totalfunctions} FC_i$
>
>       > `totalfunctions`: 程序中函数的数量；
>       >
>       > `PC`: 程序的空间复杂性；
>
>   > 主要弱点：以代码行计算距离
>   >
>   > - 代码行测量缺少相关性；
>   > - 代码行曲解了“空间”的含义
>
> - 结构复杂性：
>
>   1. 两个对象存在依赖：如果更改提供服务的对象，则有必要修改要求此服务的客户对象，那么这两个系统对象之间就存在依赖；
>
>      `UML`：一个依赖在模型元素之间指定了一种供应者/客户关系，对供应者的修改可能会影响客户模型元素；依赖意味着没有供应者，客户的语义是不完整的。
>
>   2. 如果系统中的所有依赖都被标识和理解，则说系统有适应能力——也就是说，它具有可理解性，可维护性和可伸缩性；
>
>      > 由于适应性的一个必要条件是依赖是可追踪的，故软件工程师的任务是减少依赖；

3. 在软件系统中，可以根据不同粒度对象(构件、包，类，方法)来识别依赖

> 位于低层次粒度上的较特殊对象的依赖会向上传播，在高层次粒度上产生依赖;

- 网络的结构复杂性：
  - 

- 层次的结构复杂性：

  (假设层间的1依赖只能是向下的，并且层内的依赖没有循环)

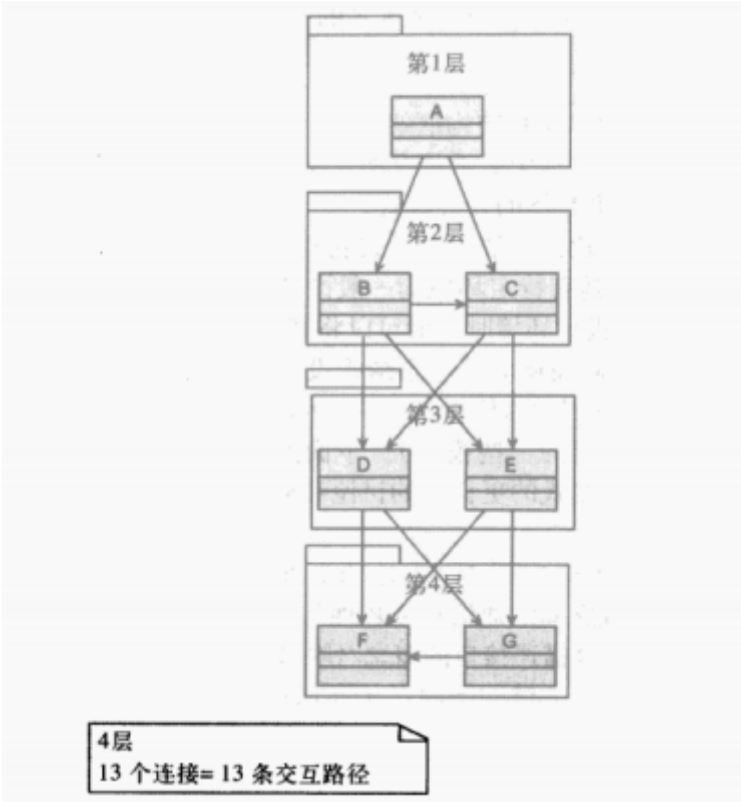
  设层为$l_1, l_2, \ldots, l_n$，对于任一层$l_i$，令：

  - size($l_i$): 第$l_i$层的对象数
  - $l_i$: 第$l_i$层的双亲数;
  - $p_j(l_i)$: 第$l_i$层的第j个双亲;

  then:

  $$_{hierarchies}CCD = \sum_{i+1}^{n} \frac{size(l_i) \times (size(l_i) - 1)}{2} + \sum_{i=1}^{n} \sum_{j=1}^{l_i} (size(l_i) \times size(p_j(l_i)))$$

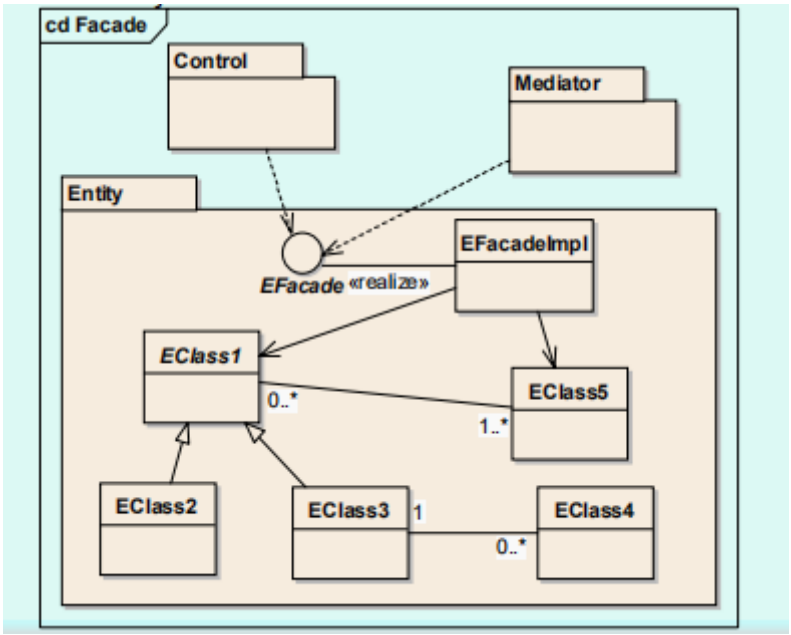
  > $_{hierarchies}CCD$: 层次体系的累记类依赖;
  >
  > n: 每一层的类数量
  >
  > > First part: 计算每一层内所有类之间的潜在(所有可能的)单向路径的数量
  > >
  > > Second part: 计算每一对相邻层的类之间潜在单向路径的数量



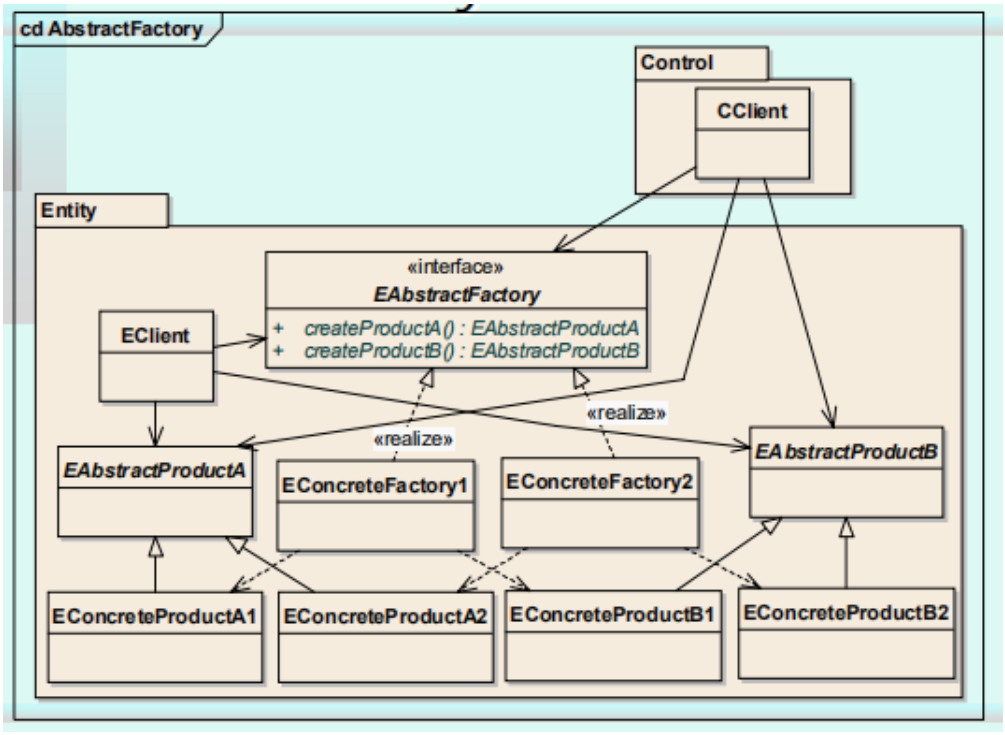

- 体系结构模式(Architectural patterns)

> 1. 体系结构框架必须遵守的原则在任何具体系统设计中的实现需要更具体的设计模式；
> 2. "设计模式提供了精华软件系统的元素或它们之间的关系的方案，它描述通常重复出现的交互设计元素的结构，解决具有特定环境的一般设计问题"；
> 3. 四人帮(`GoF`)模式;
> 4. 设计模式被用于体系结构设计环境是，就可以将其称为体系结构模式；

  - 外观(`Facade`)：
    - 外观模式定义被Gamma等定义为“使子系统更易于使用高层接口”，目标是“减少子系统之间的通信和依赖性”；
    - 高层接口封装了层(子系统、包)的主要功能，并向该层的客户提供主要的甚至是唯一的入口点；
    - 一般情况下，一个层会针对更高层的不同客户定义多个外观；



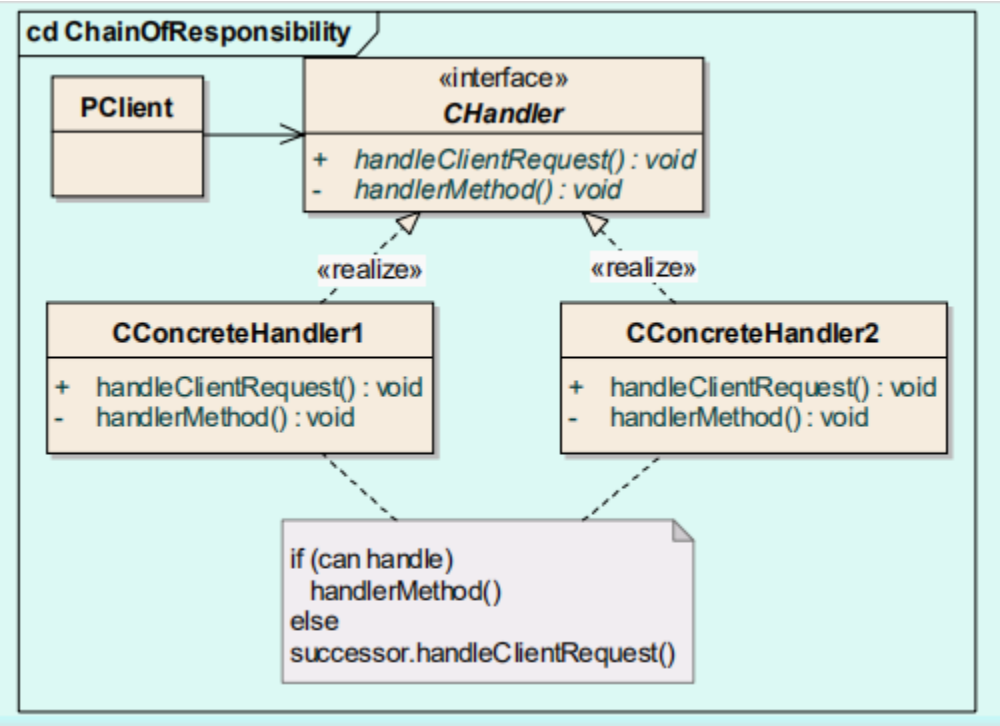

  - 抽象工厂(Abstract Factory)：

- 抽象工厂模式提供“一个接口，创建相关的或依赖的对象，而不需要说明它们的具体类”;
- 通过访问隐藏在抽象工厂接口后面的几个对象家族之一，抽象工厂的典型功能使得应用的表现行为不同(配置参数的值可以控制将访问哪个家族);
- 可以将抽象工厂模式看作是外观模式的一种变体;
- 


- 责任链(Chain of Responsibility):
  - 责任链模式的目的是“通过将处理问题的机会给多个对象来避免问题的发送者和接收者之间的耦合”
  - 可以将责任链看成是委托(delegation)概念的变体，可以理解为消息的引用链;

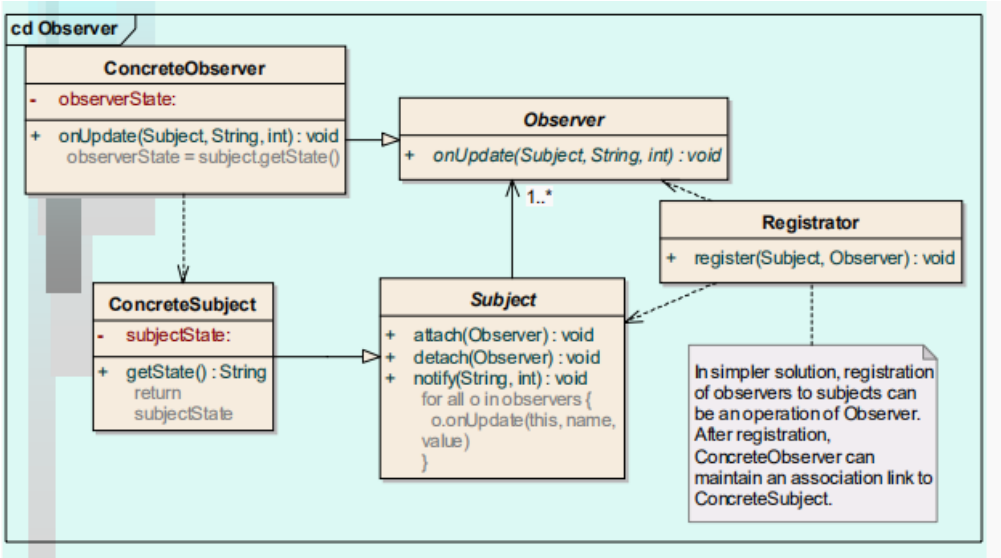
    1. 消息链开始于产生消息的对象, 如果对象本身("this" 或"self" 对象)不能对此消息做出回应它会将此消息委托给某个其他对象，此对象可以再进行委托。
    2. 当某个对象回应了此消息或到达终止条件(在这种情况下，或者返回缺省对象，或者返回某个缺省结果，指明此链是成功还是失败)时，此过程结束。

  - 在其最常见的形式中，“责任链用双亲-子或容器-被包含模型实现
    - 运用这种方法，没有被子对象处理的消息被发送给双亲，也可能是双亲的双亲，直到到达一个合适的处理对象;
  - 责任链非常适合多种面向对象的GUI活动，GUI帮助功能、构件布局、格式编排及定位都可以使用此模式;
  -



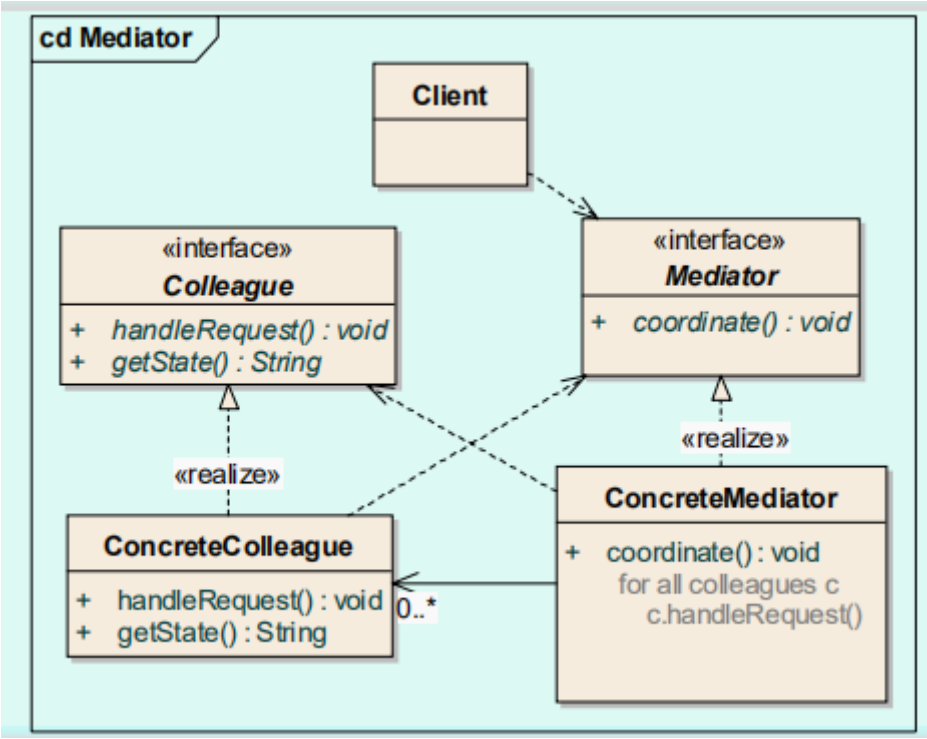

- 观察者(Observer)：
  - 观察者模式(也称为出版-订阅模式)的目的是“定义对象之间的一对多依赖，使得当一个对象变更状态时，其所有依赖者都被通知到并自动更新”；
  - 此模式关系到两类对象：
    - 被观察对象，称为主题(subject)或出版者(publisher)；
    - 观察对象，称为观察者(observer)、订阅者(subscriber)或监听者(listener)；

> 1. 一个主题可以有很多观察者订阅它，主题状态的变更会通知到所有的观察者，然后所有的观察者都会执行必要的处理，使它们的状态与主题的状态同步；
> 2. 观察者之间并不互相关联，并且可以做不同的处理来响应主题状态变更的通知；




- 中介者(Mediator)：
  - 中介者模式定义类来封装其他类之间的相互通信，这些类可能来自不同层；
  - 中介者模式“通过使对象之间不显示引用来降低耦合，并允许独立地改变它们的交互”；

> 中介者模式允许我们将复杂的处理规则，包括复杂的业务规则，组织到专门的中介者类中，从而使其他复杂的对象(称为模式中的成员)不需要交换很多消息就可以做这种处理。结果，成员对象会变得更内聚，也更简单。它们更加独立于业务规则，因此，可复用性也更强。

- 

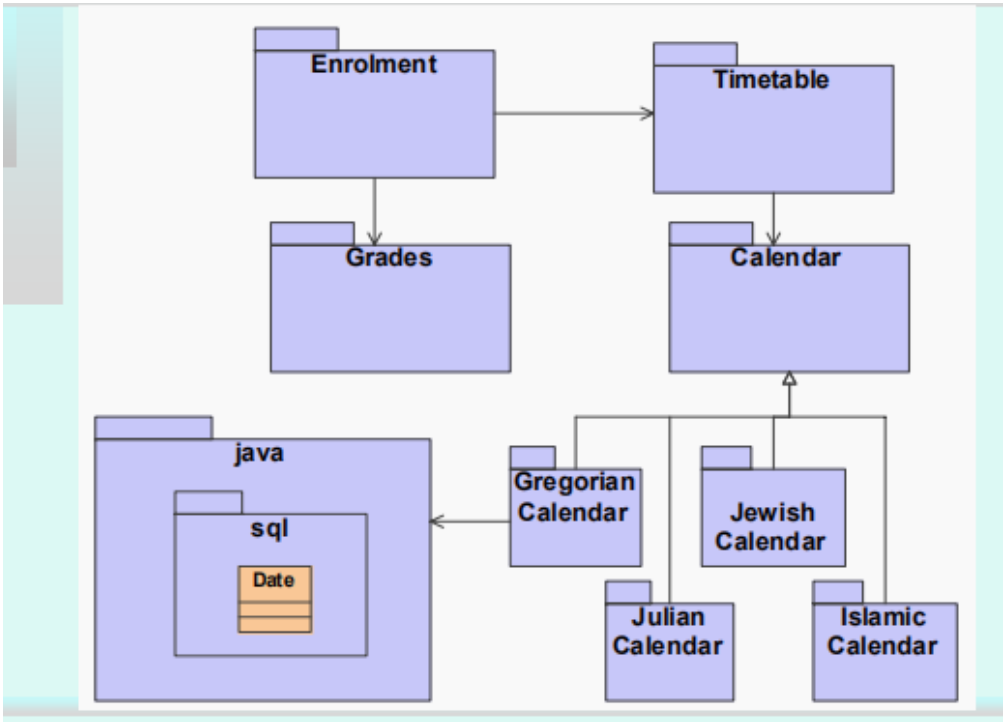

---

### 6.3 体系结构建模 (Architecture modeling)

> 1. 实现建模的设施支持体系结构建模；
> 2. 实现模式是以结点、构件、包和子系统等概念为中心的；
> 3. 除了实现模型，`UML`通过给类图增加设计约束支持体系结构建模；

- 包(Packages)：
  - 包用于表示一组类(或其它建模元素，如用例)；

    > 包是高度相关的类的聚合，这些类本身是内聚的，但相对于其它聚合来说又是松散耦合的；
    >
    > 一个类只能属于一个包；
  - 包用于划分应用程序的逻辑模型；
  - 包可以嵌套(nested)：外层包可以直接访问包括在它的嵌套包的任何类;
  - 包可以以两种关系相关联：泛化和依赖
    - 从包A到包B的依赖说明：对B的变更可能需要A中的变更；
    - 包之间的依赖在很大程度上是由于消息传递，即一个包中的类发送消息给另一个包中的类；
    - 包之间的泛化也意味着依赖，依赖是从子包到超类包；

- 


- 构件(Components):
  - 构件是系统的物理部分、实现的一个片段或一个软件程序;
  - 构件的特性:
    - 构件是独立的部署(deployment)单元——不可以部属构件的部分;
    - 构件是第三方组装单元——它是充分文档化和自