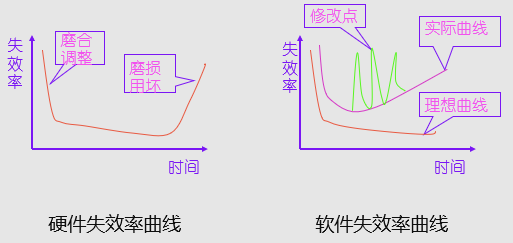
一、软件工程概述

1. 软件工程学科概览
   1. 软件工程学的存在价值：**促进软件项目成功**
   2. 软件的概念

**软件**是计算机系统中与硬件相互依存的另一部分。它包括**程序**、**数据**及其**相关文档**的完整集合

* + 1. 能够完成预定功能和性能的可执行**指令**（program）
    2. 使得程序能够适当地操作信息的**数据结构**（data）
    3. 描述程序的操作和使用的**文档**（document）
  1. 软件特点
     1. 软件是一种**逻辑实体**，而不是具体的物理实体。
     2. 软件的生产与硬件不同。
     3. 在软件的运行和使用期间，没有硬件那样的机械磨损，老化问题，但它存在退化问题，开发人员必须维护软件。
     4. 大多数软件是自定的，而不是通过已有构件组装而成的。
     5. 软件成本相当昂贵。
     6. 软件本身是复杂的。



* 1. 软件危机
     1. 定义：软件在开发和维护过程中遇到的一系列严重问题

两层含义：如何开发软件、如何维护数量不断膨胀的已有软件。

* + 1. 表现
       1. 软件开发的**进度难以控制**，经常出现经费超预算、完成期限拖延的现象。
       2. 软件**需求在开发初期不明确**，导致矛盾在后期集中暴露，从而对整个开发过程带来灾难性的后果。
       3. 软件文档**资料不完整、不合格**。由于缺乏完整规范的资料，加之软件测试不充分，从而造成软件质量低下。
       4. 软件的**可维护性差**，程序错误难以改正，程序不能适应硬件环境的改变。
       5. 软件**价格昂贵**，软件成本在计算机系统总成本中所占的比例逐年上升。
    2. 克服危机的途径

软件工程学的存在价值：促进软件项目成功

* 1. 软件工程概念Software Engineering

是研究和应用如何以系统化的、规范的、可度量的方法去开发、运行和维护软件，即**把工程化应用到软件上**

* 1. 软件工程学研究的目标

采用先进的软件工程方法，使质量、成本和生产率三者之间的关系达到最优的平衡状态。

* + 1. 软件开发成本较低；
    2. 软件功能能够满足用户的需求；
    3. 软件性能较好；
    4. 软件可靠性高；
    5. 软件易于使用、维护和移植；
    6. 能按时完成开发任务，并及时交付使用
  1. 软件生存周期

是指软件产品从考虑其概念开始到该软件产品交付使用，直至最终退役为止的整个过程。一般包括**计划、分析、设计、实现、测试、集成、交付、维护**等阶段。

**注意：**在实践中，软件开发并不总是按照计划、分析、设计、实现、测试、集成、交付、维护等顺序来执行的，即各个阶段是可以重叠交叉的。整个开发周期经常不是明显地划分为这些阶段，而是分析、设计、实现、再分析、再设计、再实现等迭代执行。

* 1. **软件生存周期的各个阶段主要任务**
     1. **计划阶段**

确定待开发系统的**总体目标**和**范围**。

研究系统的**可行性**和可能的解决方案，对资源、成本及进度进行合理的估算。

* + 1. **分析阶段**

分析、整理和提炼所收集到的**用户需求**，建立完整的分析模型，将其编写成**软件需求规格说明**和初步的用户手册。

* + 1. **设计阶段（总体设计和详细设计）**

设计阶段的目标是**决定软件怎么做**。

软件设计主要集中于**软件体系结构、数据结构、用户界面和算法**等方面。

* + 1. **实现阶段（编码）**

实现阶段是将所设计的各个模块编写成计算机可接受的程序代码。

* + 1. **测试阶段**

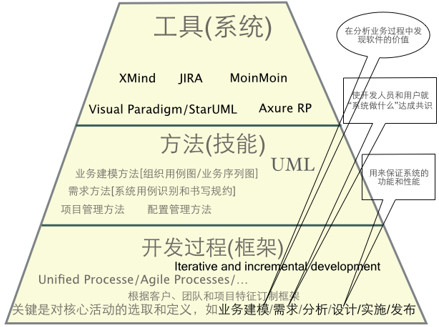
设计测试用例，对软件进行测试，发现错误，进行改正。

* + 1. **运行和维护阶段**

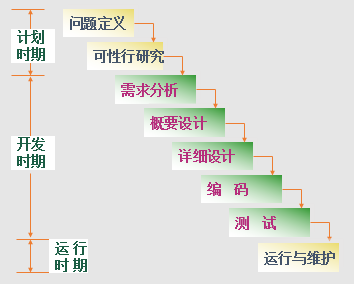
应当在软件的设计和实现阶段充分考虑软件的**可维护性**。

维护阶段需要测试是否正确地实现了所要求的修改，并保证在产品的修改过程中，没有做其他无关的改动。

1. 软件过程—软件工程的核心组成部分
   1. 软件工程三要素



* 1. 瀑布模型



**特点：**

自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。

上一阶段的变换结果是下一阶段变换的输入，相邻两个阶段具有因果关系。

**问题：**

各个阶段的划分完全固定，阶段之间产生大量的文档，极大地增加了工作量。

开发模型是线性的，用户只有等到整个过程的末期才能见到开发成果，增加了风险。

早期的错误可能要等到开发后期的测试阶段才能发现，进而带来严重的后果。

* 1. RUP统一软件过程（Rational Unified Process)

RUP的中心思想是：**用例驱动、架构为中心、迭代和增量**

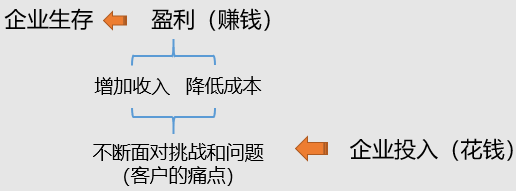
* 1. 迭代与增量

增量：逐块建造

迭代：反复求精

二、踏上iconix软件过程之路

1. 站在客户的角度思考



如果能帮助客户“**开源、节流**”，客户就愿意购买我们的软件

注意：

企业的运作是个复杂系统，无法通过一种方法或一个软件解决所有问题。只能不断优化局部，而实现整体的进步。

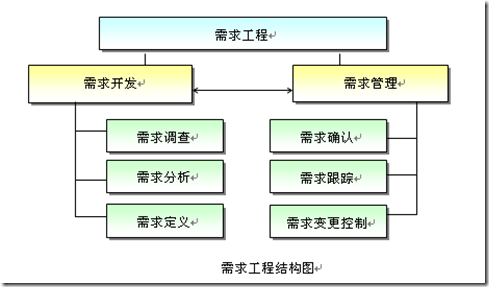
每一个软件系统都解决企业某方面的痛点，这个痛点直接或间接地与企业的“开源、节流”相关。

1. 需求工程概述
   * 1. 需求是软件成功的基础

需求十分重要，并且贯穿整个软件开发的整个过程，需要引起足够的重视

2、需求工程

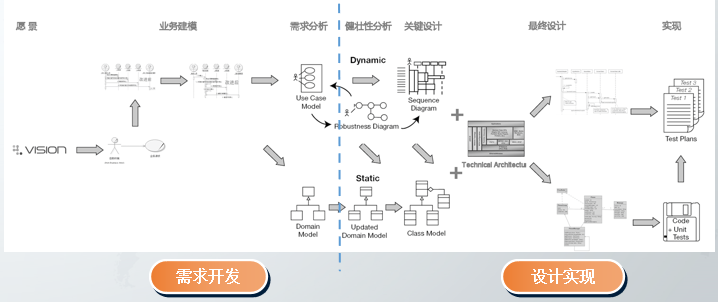
通过合适的工具和记号系统地描述待开发系统及其行为特征和相关约束，形成需求文档，并对用户不断变化的需求演进给予支持



**需求开发**(Requirement Development, RD)目的是通过调查与分析，获取用户需求并定义产品需求。

**需求管理**(Requirement Management, RM)目的是在客户与开发方之间建立对需求的共同理解，维护需求与其他工作成果的一致性，并控制需求的变更。

1. iconix软件过程的需求阶段
   * 1. iconix过程总览



从图中可以看出扩展ICONIX过程可分为：**愿景、业务建模、需求分析、健壮性分析、关键设计、最终设计和实现**这几步。它基于极限编程和敏捷软件开发的思想，提倡在项目开始阶段构建域模型和用例模型，其中用例模型驱动整个动态模型，而域模型驱动整个静态模型。ICONIX过程是一种以最小步骤实现用例到代码的方法学，覆盖了软件过程中所有关键的环节。

* + 1. iconix过程特点

尽早进入编码阶段，缩短分析设计周期的软件开发方法

合理的简化统一过程(RUP)， 基于敏捷软件开发的思想

与 RUP相比，是轻量级的过程。与敏捷相比，ICONIX提供充足的需求和设计文档，但不过度分析设计

ICONIX过程从把需求文档变成可运作的代码过程只需四步，使用四张UML图：**用例图、序列图、类图、健壮性图**（非UML标准）

1. 愿景（愿景是核心和起点）
   * 1. 获取愿景的三步曲

**第一步：找到软件项目的“老大”；**

**第二步：得到“老大”对项目的期望（愿景）；**

**第三步：描述出愿景的度量指标；**

* + 1. 第一步：**找“老大”**

要点：软件项目要改善哪个组织的流程？

老大就是要**改善的组织中最有权力的干系人**

* + 1. 第二步：**得到“老大”对项目的期望（愿景）**

软件项目的**愿景是“老大”愿意掏钱开发这个系统的目的**

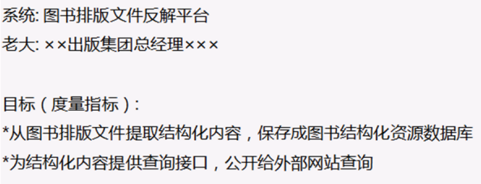
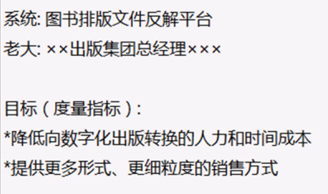
注意：**愿景不是功能**

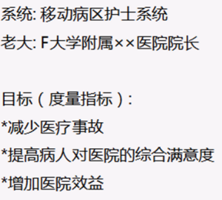
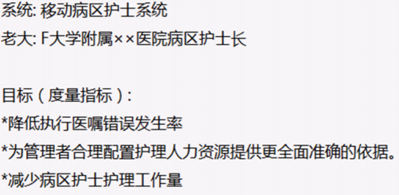
* + 1. 第三步：描述出愿景的度量指标

描述愿景必须指出**度量指标**

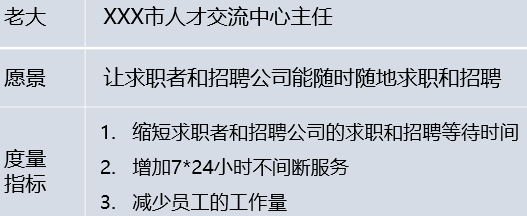
愿景描述不是记录具体做了什么事，而是描述改善组织的哪些指标

示例：减少采集数据所花费的时间、提高动画的速度、缩短订单的处理周期

 错误—>正确 

 错误—>正确 

正确示例：



1. 业务建模
2. 业务建模的意义和步骤
   1. 意义和注意事项

业务建模是从**组织**的角度来定位系统应该提供的价值

业务建模要求我们**把视角从软件系统转向客户组织，站在客户角度看问题**，以达到清晰准确地“诊断”，对症“开方”。

* + - 1. 明确为谁服务--找准**客户**及其**愿景**，切记不是在为自己做系统；
      2. 要改进的组织是什么现状--有什么**痛处**和**不足**；
      3. 如何**改进**--新系统的价值就是解决客户痛处、改良客户不足，这才是客户愿意掏腰包的动力；
      4. 在业务建模和需求分析阶段，忘掉自己技术专家的身份；
  1. 业务建模的步骤
     1. **明确我们为谁服务**（**选定**愿景要改进的**组织**）

最佳的研究范围就是愿景涉及的、需要改进的组织

选定的业务组织跟老大的职权范围有关

选定业务组织要结合系统愿景（可能是整个行政组织，也可能是行政组织中的部分）

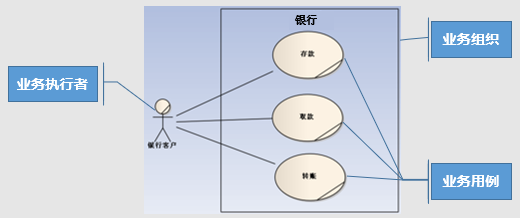
* + 1. **要改进的组织是什么现状**（业务用例图、现状业务序列图）
    2. **我们如何改进**（改进业务序列图）

1. 业务用例：从外部看企业

从外部看：**组织**是**价值**的集合，用业务用例图来建模。

从内部看：**组织**是**系统**的集合(人是一种智能系统)，用业务序列图来建模。

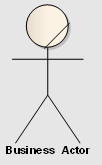
业务用例图帮助我们从**高层次**了解组织的**业务构成**



1. 业务执行者（**Business Actor**）

在组织之外和组织交互的人群或组织

例如储户是银行的业务执行者，食客是餐馆的业务执行者



1. 业务用例（**Business Use Case**）

组织为业务执行者提供的价值

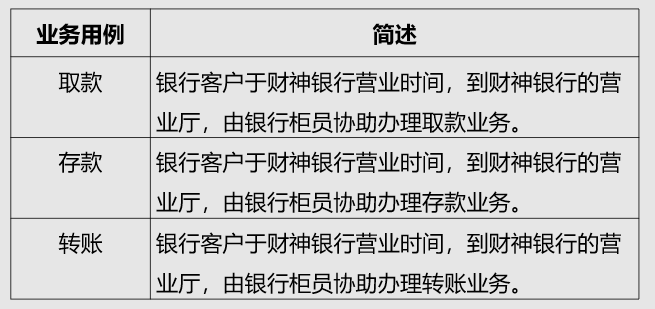
例如银行的业务用例有存款、取款、转账等；餐馆的业务用例有吃饭



撰写**业务用例简述**

以简明扼要的文字对每个业务用例进行描述，通常的格式是：业务执行者通过业务组织的某些工作，达到固定目的。

例如取款用例的简述为：**银行客户于财神银行营业时间，到财神银行的营业厅，由银行柜员协助办理取款业务**。



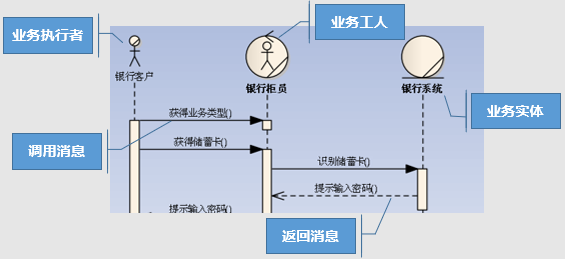
1. 业务组织
2. 业务序列图：从内部剖析企业

业务序列图帮助我们从细节上了解组织的业务流程

序列图以**面向对象**的思想来看业务流程

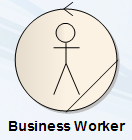
把业务流程看作是一系列业务对象之间为了完成业务用例而进行的协作

业务序列图详细描述**业务执行者、业务工人、业务实体**之间如何交互，以完成某个业务用例的实现流程



1. 业务工人（**Business worker**）

位于**组织内部**，负责业务流程中某些工作的人员。例如银行工作人员，医院医生



1. 业务实体（**Business Entity**）

在业务用例的实现流程中，业务工人所使用的“系统”。例如银行的取款机、点钞机，学校的校园卡系统。

可以和业务工人相互取代各自的职责

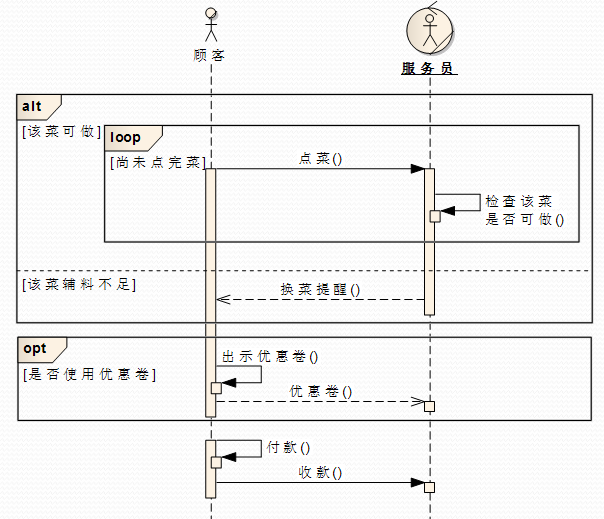


1. 业务序列图的作用
   * + 1. 识别业务对象：业务执行者、业务工人、业务实体；
       2. 确定业务对象间的职责、协作及交互顺序；
       3. 通过序列图来了解现状，为业务的优化提供依据；
2. 分支的画法

Loop循环分支

Alt= alternative条件分支

Opt= optional 可选分支



1. 注意

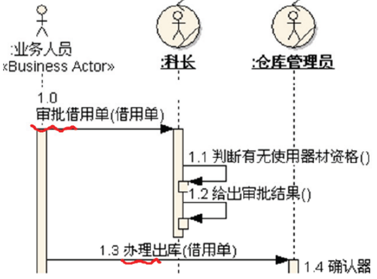
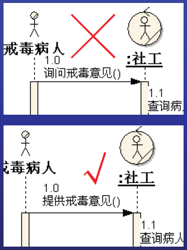
本阶段不要考虑要实现什么系统

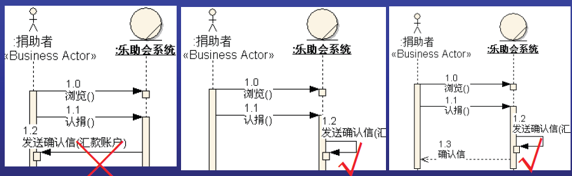
调用消息的名字，代表责任和目的，**左边请求右边做某事**

调用消息的方向，**责任委托**，不是数据流动

只画领域相关的系统，最小的单位是人或独立智能系统

时间是特殊的业务实体





1. 改进序列图
   1. 改进点
      1. 信息自动流转
      2. 封装复杂业务逻辑
      3. 职责的转移
      4. 访问和操作业务对象
   2. 采用改进业务序列图来改进现有业务流程
      1. 将“新系统”作为一个业务实体进行整体设计；
      2. 将“新系统”引入组织现有业务流程；
      3. 查看其可以改进的流程；
      4. 评估改进结果；
   3. 业务序列图和改进业务序列图的价值

业务序列图展现企业内现有**业务流程**，暴露问题，**为优化提供直观依据**。

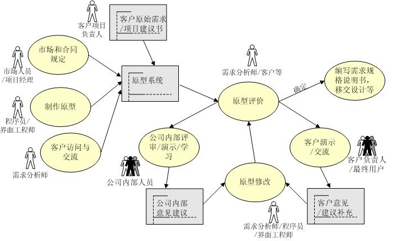
通过改进业务序列，可以提前**模拟出新系统**的出现，将对组织现行的业务流程造成哪些影响，可以提前**评估新系统的可行性**或提前进行相应的准备工作，实现安全平稳的组织改进。

1. 需求分析
2. 需求分析的方法

需求分析就是确定新系统的目的、范围、定义和功能

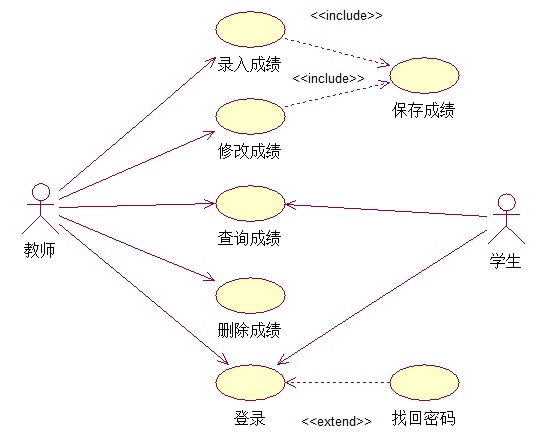
1. 原型法

原型法是指在获取一组基本的需求定义后，利用高级软件工具可视化的开发环境，快速地建立一个目标系统的最初版本，并把它交给用户试用、补充和修改，再进行新的版本开发。反复进行这个过程，直到用户满意为止的一种方法



1. 用例法

用例图(Use Case Diagram)是由软件需求分析到最终实现的第一步，它描述人们如何使用一个系统。用例视图显示谁是相关的用户、用户希望系统提供什么样的服务，以及用户需要为系统提供的服务，以便使系统的用户更容易理解这些元素的用途，也便于软件开发人员实现这些元素。用例图最常用来描述系统及子系统



1. 域建模：以面向对象思想构建术语表

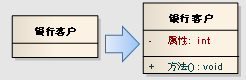
为项目创建一个**术语表**。确保项目中的每个人都能以清晰一致的术语来理解和交流问题领域

1. 域建模

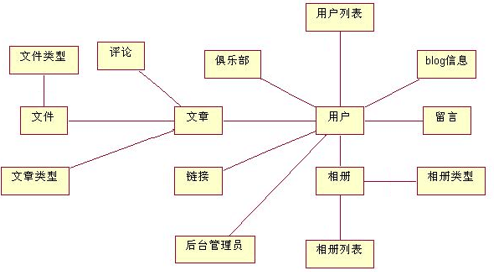
**域建模**比普通的项目**术语表**优良的地方体现在：以图的方式清晰地显示出不同术语间的关系（减少理解偏差）。

描述业务中涉及到的**实体及其相互之间的关系**，是帮助系统分析人员、用户认识现实业务的工具。

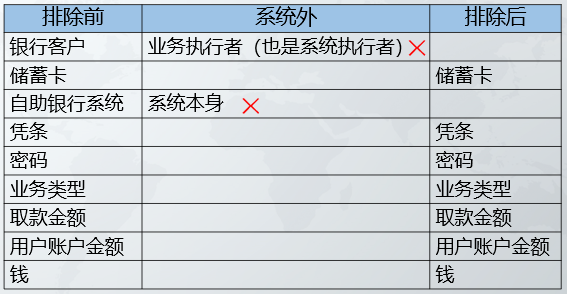
域模型图将通过不断修正完善逐步演化为最终的**静态类图**



示例：



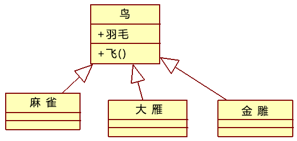
1. 域建模的步骤
   1. **仔细阅读需求文档，提取出名词和名词短语**
   2. **排除列表中重复、相似的术语**
   3. **排除超出系统范围的术语**



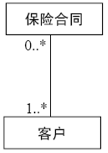
* 1. **画出第一版域模型图**
  2. **整理第一版域模型**

1. 域模型之间的关系

泛化[Generalization]，一般元素和特殊元素的关系。



关联[Association]，两个类之间存在着某种语义上的联系



1. 高级话题
   1. **域模型≠数据模型★**

**域模型设计期间不用考虑数据的存放问题**，只考虑业务描述中涉及的实体以及实体之间的关系

* 1. 注意

不要花费太多的时间纠缠在最初的域建模工作上

不要把域模型错误地认为是数据模型

不要指望最终的类视图和域模型图完全匹配，但它们在很大程度上应该是类似的

不要把与界面相关的类加入域模型中

在用例分析前做域建模，以避免命名混淆

1. 用例分析：系统**功能性需求**分析的好工具
   1. 系统用例建模的意义

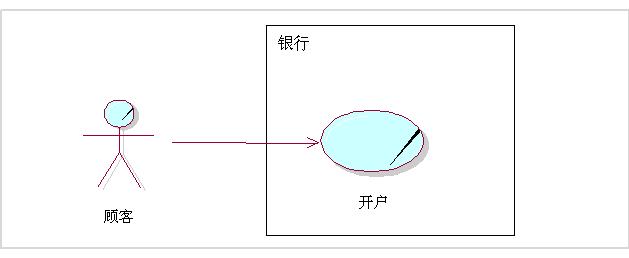
系统需求分析的目的是把视角从**业务组织**转向**新系统**，站在**最终用户**及其它干系人的角度看问题

系统用例是对**（新）系统**为系统执行者提供的**价值**的建模

* 1. 业务用例&系统用例
     1. 业务用例

对银行进行**业务建模**，研究对象是**银行**

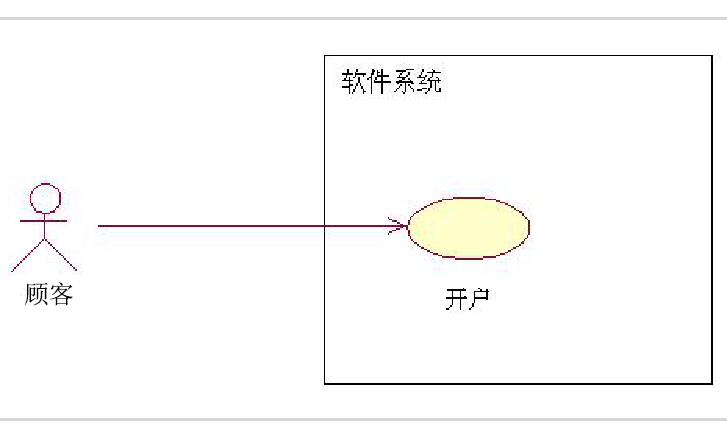
用于分析**现有业务**的利与弊



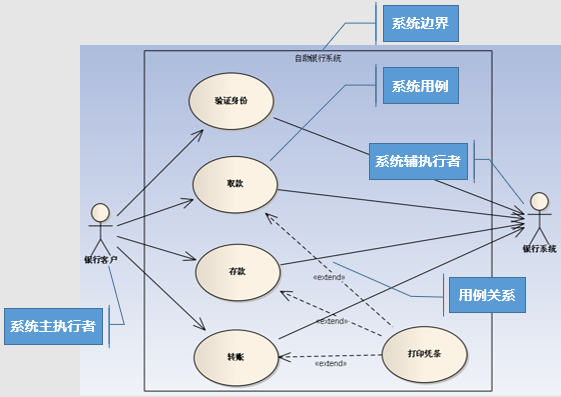
* + 1. 系统用例

对银行的软件系统进行**系统建模**，研究对象是银行的**软件系统**

用于分析**新系统**所带来的价值



* 1. 系统用例图



* 1. **系统用例建模步骤**
     1. **绘制系统用例图**
        + 1. 确定系统边界

系统边界，即系统包含的功能与不包含的功能之间的界限

**即分割出系统内和系统外**

系统内，需要我们投入大量的精力进行**建设**

系统外，需要我们考虑与它们的**接口**

* + - * 1. 识别系统执行者

执行者[actor]是在**系统之外**，透过**系统边界**，与系统进行**有意义交互**的**任何事物**

**主执行者**：用例发起者、用例为其实现有价值的目标

**辅执行者**：用例支持者、用例的完成需要与其交互，得到其支持

* + - * 1. 识别系统用例

系统用例是系统执行的一系列动作，这些动作可以生成“执行者”可观测的**有价值**的结果

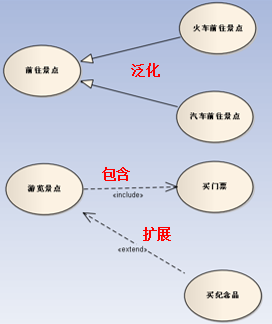
系统用例是执行者通过系统可以**达到的某个目标**

* + - * 1. 确定用例间的关系

泛化[Generalize]：子用例和父用例相似，但表现出更特别的行为；子用例将继承父用例的所有结构、行为和关系

包含[Include]：使用包含用例来封装一组跨越多个用例的相似动作（行为片断），以便多个基用例复用

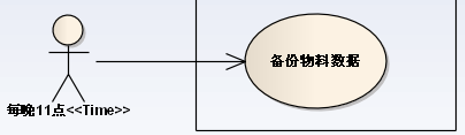
扩展[Extend]：将基用例中一段相对独立并且**可选**的动作，用扩展用例加以封装，再让它从基用例中声明的扩展点上进行扩展，从而使基用例行为更简练和目标更集中



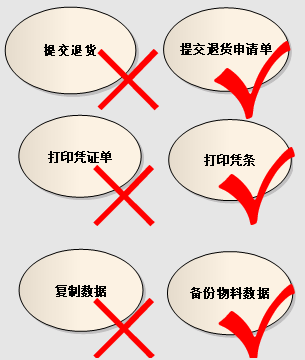
* + - * 1. 高级话题

·先找到执行者，从执行者的角度去寻找用例找到一个执行者，就可以找到一堆用例

·‘时间’执行者



·用例命名必须是动宾短语，采用域建模中的名词术语



**·用例≠功能**

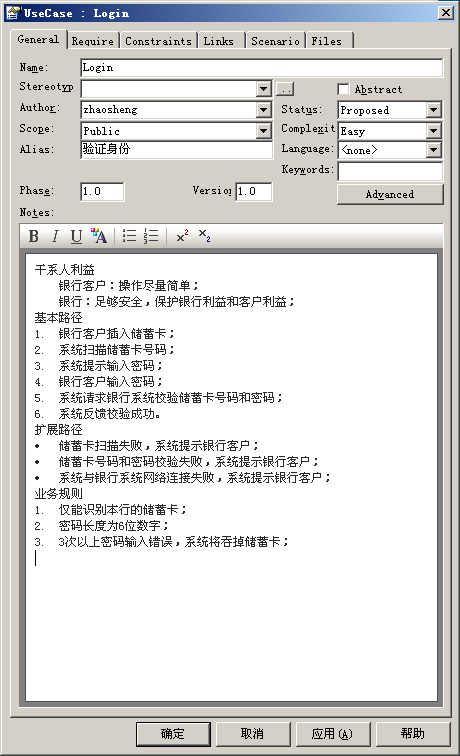
**·用例≠步骤**（完成这个愿望可能需要经过很多步骤，但任何步骤不能够完整的反映执行者的目标）

·所有用例应该都能追溯到愿景目标，所有的愿景目标都应有对应的用例实现

* + 1. **编写系统用例描述**
       - 1. 用例描述的作用

**系统用例图**描述**总体**，**系统用例文档**描述**细节**

* + - * 1. 用例描述基本组成

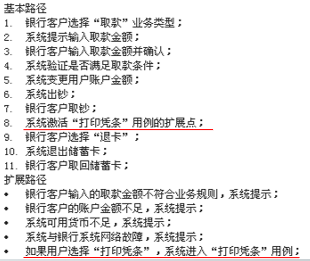


**基本路径**：以主动语态、“**名词-动词-名词**”格式来书写，**主语只能是执行者或系统**

* + - * 1. 高级话题

·不要涉及页面细节（示例：会员从下拉框中选择类别（错误））

·基本与扩展分开



* + 1. **更新域模型**
  1. 非功能性需求分析及需求定义与评审
     1. 功能性需求&非功能性需求

功能性需求，通俗讲就是系统可以做什么。

非功能性需求，通俗讲就是系统可以把某项功能做到什么程度。

* + 1. 软件产品的典型非功能性需求
       - 1. **可靠性**[Reliability]

系统在一定时间内、在一定条件下无故障地执行指定功能的能力

示例：自助银行系统7\*24小时不间断服务

* + - * 1. **可用性**[Usability]

一个产品可以被特定的用户在特定的上下文中，有效、高效并且满意得达成特定目标的程度

示例：用户完成任何一项业务，操作步骤不超过5步

* + - * 1. **性能**[Performance]

系统实现预期功能的能力的特性

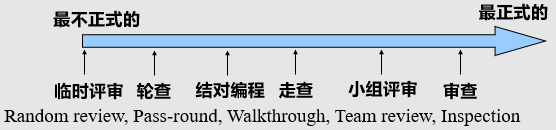
示例：身份验证时间<1秒

* + - * 1. **可支持性**[Supportability]

系统在故障解决和系统升级方面的能力

示例：95%的系统故障可在两小时内解决

* + 1. 需求评审





结对编程（/需求分析/设计/测试）。

轮查：交叉交换文档产物，互相提出意见。

临时评审：在日常沟通中做回顾确认。

* + 1. 审查过程概述
       - 1. 规划（谁参加？评审什么？）
         2. 总体会议（会前会）

召集参加评审会的所有成员开一个简短的会议，讨论、明确要评审的内容、评审的要点、评审时所需的资料、缺陷检查表

* + - * 1. 准备

评审人员提前阅读和准备，才能提出有价值的建议

* + - * 1. 审查会议：暴露问题、讨论问题
        2. 返工

评审中发现的错误必须得到重视和回应

* + - * 1. 跟踪

解决问题：对评审提出的问题进行解决

避免问题再次出现：对问题进行分类、因果分析，找到问题的深层次原因

* + 1. 需求规格说明书的检查要点
       - 1. 正确性；
         2. 必要性；
         3. 优先级；
         4. 明确性；
         5. 可测性；
         6. 完整性；
         7. 一致性；
         8. 可修改性

五、健壮性分析

1. 健壮性分析的价值和基本概念
   1. 健壮性分析帮助**完善和确认需求分析的成果**
   2. 健壮性分析的优点

用例的对象化图示，将用例和对象链接起来。

指出了参与用例场景的对象相互之间如何交互。

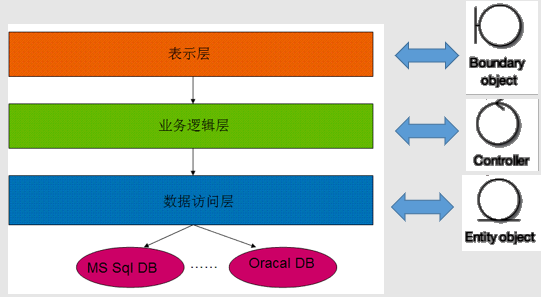
确保用例文本的正确性，从而提供了健康性检查。

确保用例考虑了所有必需的扩展路径，从而提供完整性和正确性检查。

让你能够(持续)发现对象。

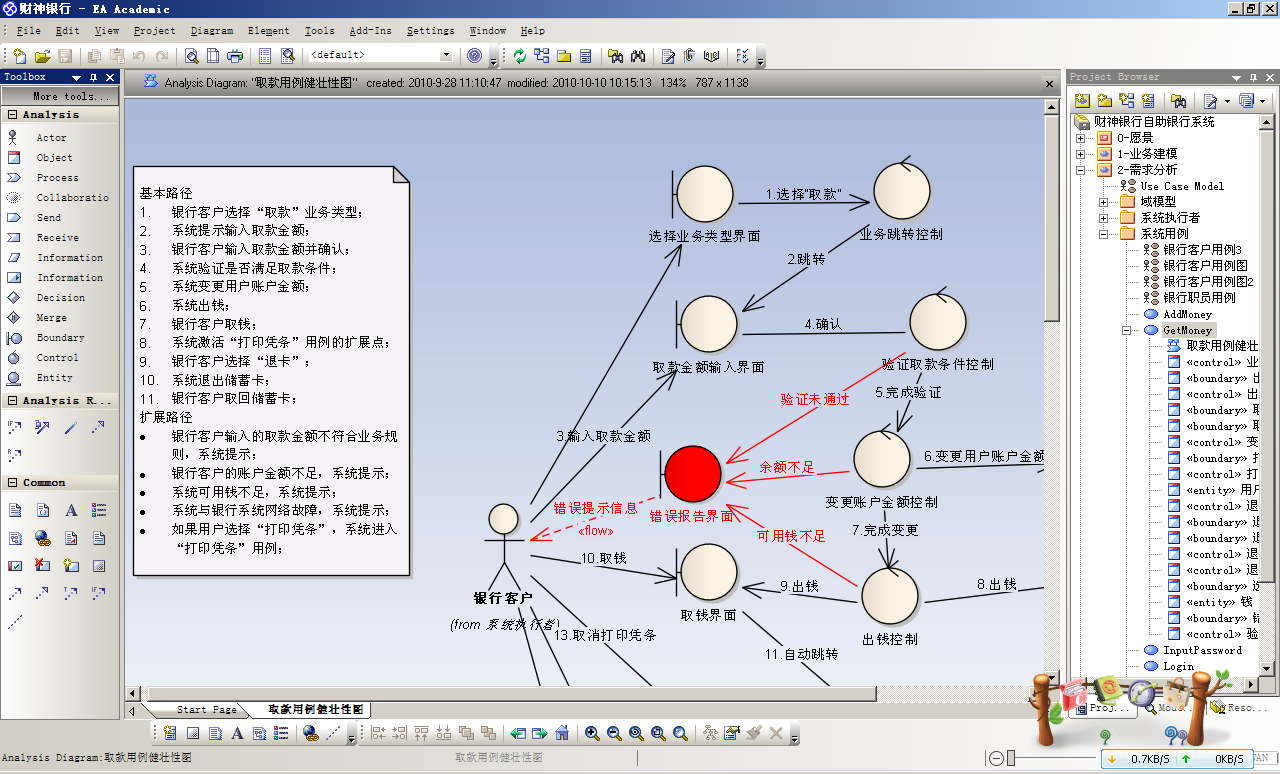
缩小分析和设计的鸿沟，从而最终完成初步设计。

* 1. 健壮性分析中的三种元素
     1. **边界类[Boundary objects]**与用户交互的对象，系统和外部世界的界面，如窗口，对话框等等。
     2. **实体类[Entity objects]**是现实世界存在的实体对象，域模型中的类，它常对应于数据库表和文件。有些实体对象是“临时”对象（如搜索结果），当用例结束后将消失。
     3. **控制器类[Controller objects]**边界和实体间的“粘合剂”,将边界对象和实体对象关联起来，它包含了大部分应用逻辑，在用户和对象之间架起一座桥梁。控制对象中包含经常修改的业务规则和策略。
  2. 经典的三层架构



* 1. 健壮性分析中三种元素的交互规则
     1. **执行者**只可以和**边界**对象通话；
     2. **边界对象**和**控制器**可以互相通话（**名词<->动词**）；
     3. **控制器**可以和另一个**控制器**通话（**动词<->动词**）；
     4. **控制器**和**实体**对象可以互相通话（**动词<->名词**）；

1. 健壮性分析的步骤
2. 创建一个新的健壮性图
3. 将**用例描述**关联到健壮性图上（方便同步更新）
4. 从**基本路径**的第一句话开始画健壮性图
5. 贯串整个用例基本路径，一次一个句子，画执行者、适当的边界对象和实体对象以及控制器，和各元素之间的连线
6. 将每一个**扩展路径**画在健壮性图上，并以红色标示出



1. 健壮性分析的9项指导建议
2. 将用例描述关联（链接）到健壮性图上
3. 从域模型中提取实体对象，如果发现之前有缺漏，则补充上
4. 在画健壮性图时修正之前用例中模糊的地方
5. 将每一个屏幕对象定义为边界对象，并进行清晰的命名
6. 切记控制器对象大部分时候对应的是逻辑操作方法，偶尔也会对应真实的控制器对象
7. 在画健壮性图时，如果调用另一个用例，就直接在图上画出调用此用例即可
8. 切记**健壮性分析描绘的是概要设计**而不是详细设计
9. 健壮性图上的**边界对象和实体对象会转化为时序图中的对象实例**，而**控制器对象会转化为消息或控制器实例**
10. 切记健壮性图是用例的“对象化图示”，它的**目的是优化和完善用例文本和域模型**
11. 更新域模型

健壮性分析技术两个主要的价值：其一帮助完善用例分析结果；其二完善域模型，作为需求分析走向系统设计的过度技术

六、关键设计

1. 关键设计的意义和方法

基于用例图、用例描述和健壮性图，采用**序列图**来描述参与者、边界、实体之间的交互

主要意义：就是要通过寻找对象之间的交互关系，进而进行方法（操作或行为）分配

1. 关键设计的步骤
   * 1. 将现有的域模型直接作为第一版静态类模型；
     2. 基于用例描述和健壮性分析结果，画出每个用例的序列图；

健壮性图中的**控制类**会转化为**方法**；

如果也转化为**控制类**，那么就添加到类图中（**一般边界类不添加到类图中**）；

1. 逐步贯穿健壮性图上的每一个控制器，每次一个，画出序列图上相应的方法，然后核对，移至下一个控制器。
2. 控制器和方法之间并不一定是完全一对一匹配的，也可能会转化为两个或多个方法。
3. **有时，控制器也可能会转换为一个真正的控制类。**（例如：检查余额可以在账户类中，但跳转放在账户类中不合适，可以单独放到一个控制器类里）
4. 序列图会对类图做进一步的更新，完善其方法

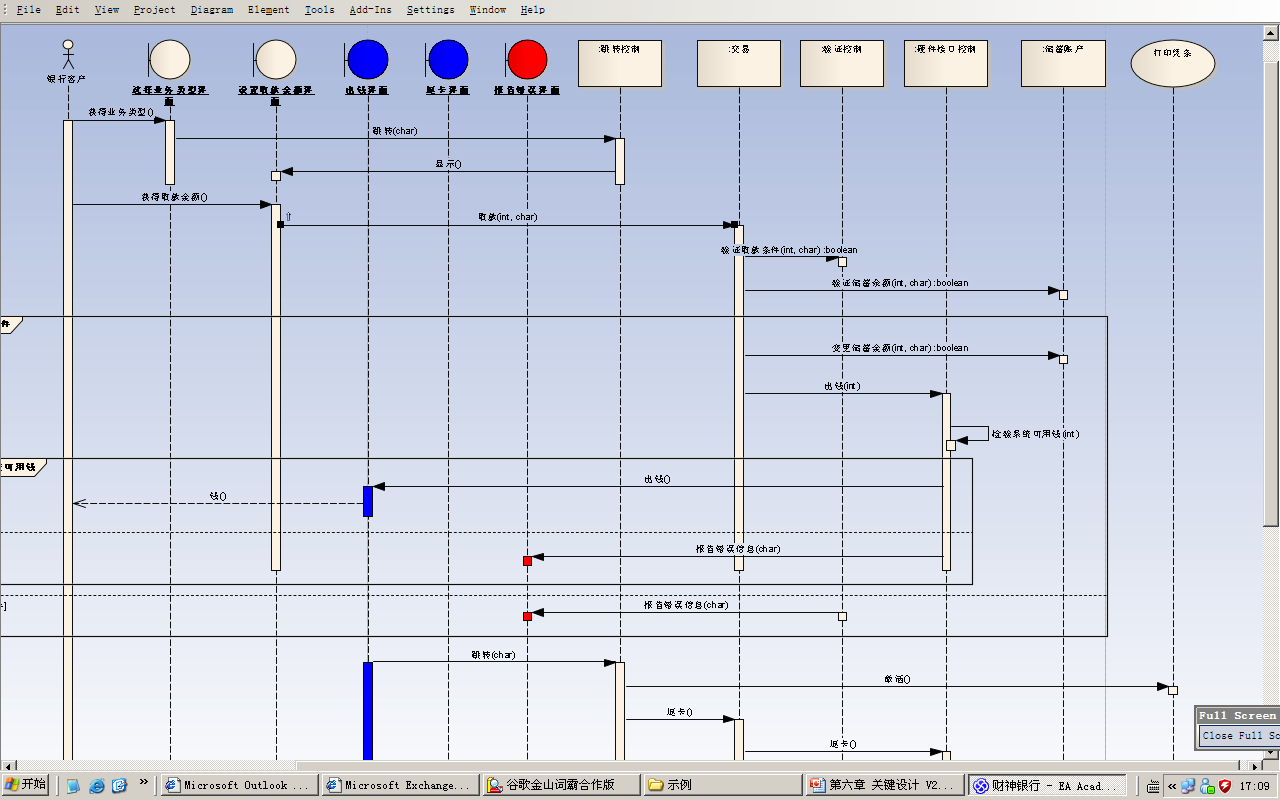
如何决定控制器分配给哪个类？**高内聚、低耦合**

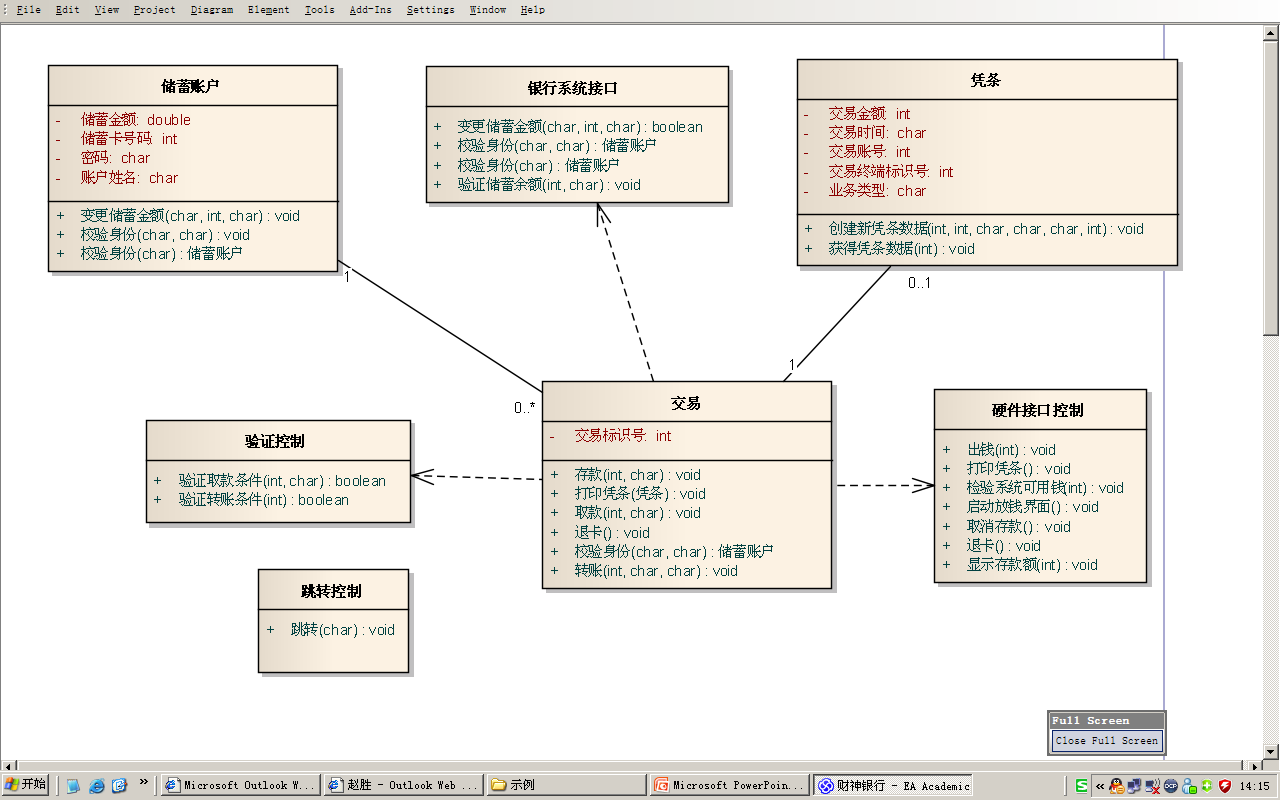
高内聚是指一个软件模块(类)是由相关性很强的代码组成，只负责一项任务，也就是常说的单一责任原则。

低耦合是指一个软件模块与模块之间的接口，尽量的少而简单。如果某两个模块间的关系比较复杂的话，最好首先考虑进一步的模块划分，降低相互的依赖。这样有利于修改和组合。

目的：使得模块的“可重用性”、“移植性”大大增强

* + 1. 整理静态类图和序列图；
    2. 关键设计复核，迭代更新用例图、类图和序列图；





1. 关键设计的复核

形式：面对面会议。**（可能多次、每次会很久** ☺**）**

参会人：分析设计师、专家（分析、设计、开发）。

被审材料：用例图、用例描述、类图、序列图

过程：需求分析师主持，介绍需求分析成果，所有参与者交流讨论，达成统一理解和确认。

结论：所有参与者签字确认。

**注意：已不需要甲方人员参与。这是一个技术性的复核对话，因此，需要的是拥有技术思想的人员**

七、详细设计

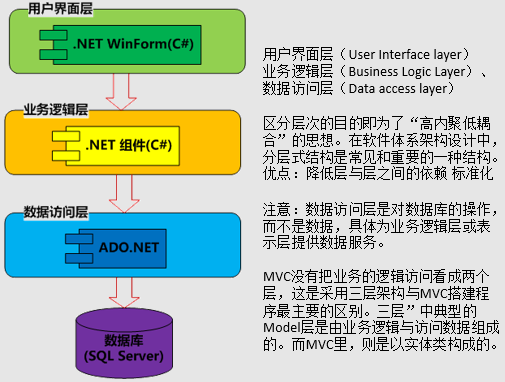
1. 详细设计内容

技术架构及相关考虑

选择开发语言

网络拓扑及安全

体系结构



硬件支持环境

软件支持环境

1. 认识敏捷开发
2. 什么是敏捷开发
   * + 1. 传统软件开发过程的常见症结

交付周期长；害怕需求变更；中间过程不可控；测试周期被一缩再缩；最终结果差强人意

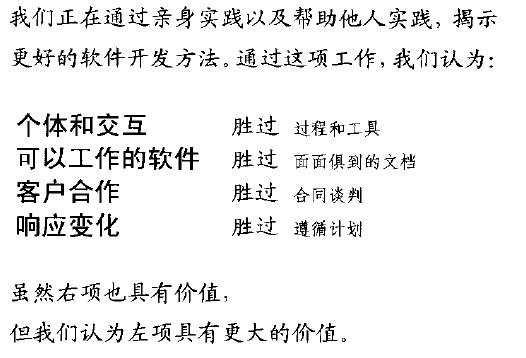
* + - 1. 敏捷软件开发模式

敏捷开发的核心思想是以**用户的需求进化**为核心，采用**迭代、循序渐进**的方法进行的软件开发。

由传统迭代式软件开发模式发展而来，强调产品价值、团队协作、客户参与、先期验证、简化流程、拥抱变化。

总结吸收成功软件项目研发的最佳实践；与现代管理思想相辅相成

* + - 1. 敏捷宣言揭示更好的软件开发方法



敏捷宣言（ 2001年）是敏捷起源的基础，由上述4个简单的价值观组成

* + - 1. 敏捷更符合软件开发规律

软件更像一个活着的植物，软件开发是自底向上逐步有序的生长过程，类似于植物自然生长敏捷开发遵循软件客观规律，不断的进行迭代增量开发，最终交付符合客户价值的产品

二、敏捷开发已逐渐成为主流趋势

敏捷对生产率、质量、满意度、成本有明显改进

1. 正确认识敏捷开发
   * 1. 对敏捷的常见误解

误解一：敏捷开发意味着可以不需要文档、设计和计划

误解二： 敏捷只是一些优秀实践，或者是优秀实践的结合

误解三： 敏捷只适用于小项目开发

误解四： 敏捷只会对研发产生改变

**误解五： 管理者不需要亲自了解敏捷，只需要管理上支持就可以了**

**误解六： 引入敏捷只需要按照既定的步骤去做就可以了**

误解七： 敏捷是CMM的替代品，是另一种流程

误解八： 敏捷只注重特性的快速交付，在敏捷下架构不重要了

* + 1. 敏捷 = 理念+优秀实践+具体应用

敏捷包括三个层次：理念（敏捷核心思想）、优秀实践（敏捷的经验积累）、具体应用（能够结合自身灵活应用才是真正敏捷）

* + - 1. **理念**（value、team、adapting）

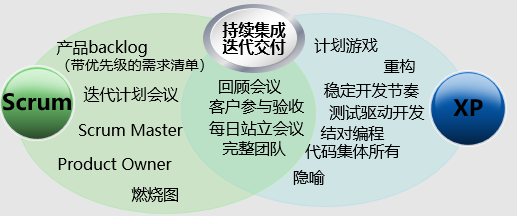
**聚焦客户价值，消除浪费**

**激发团队潜能，加强协作**

**不断调整以适应(Adapting)变化**

* + - 1. 优秀实践

业界敏捷优秀实践概览



* + - 1. 具体应用

因地制宜选择适合的敏捷实践

团队在透彻理解敏捷理念的基础上，可以灵活选择最适合自己的实践，避免教条化

1. Scrum敏捷过程一
2. Scrum敏捷开发过程概述
3. 什么是scrum

Scrum是一个**增量的、迭代的**敏捷开发过程

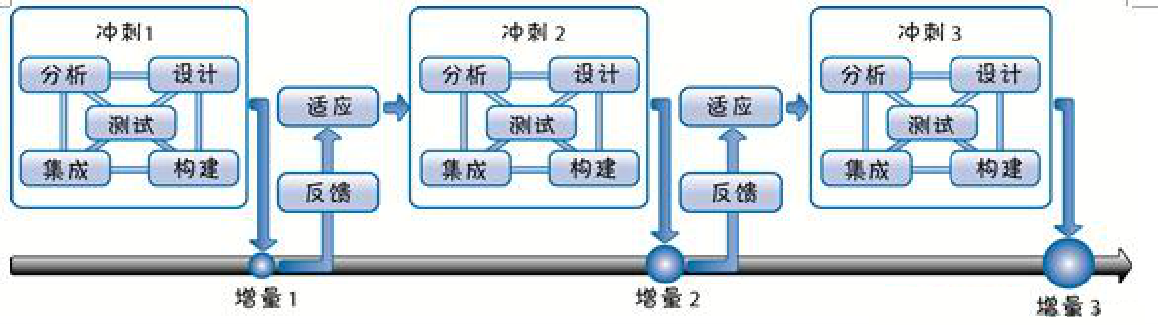
1. 迭代式开发

迭代开发将整个软件生命周期分成多个小的**迭代（一般2-6周）**。

**每一次迭代都由需求分析、设计、实现、测试和集成在内的多个活动组成。**

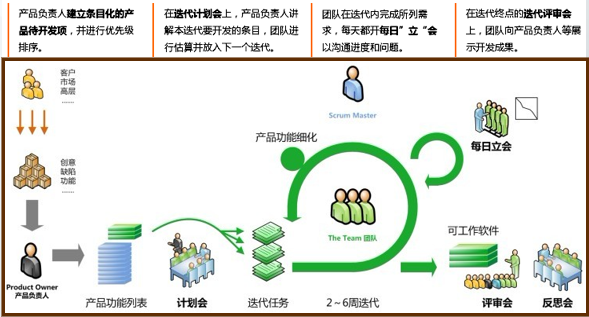
每一次迭代都可以生成一个稳定和被验证过的软件版本。

1. 产品增量



1. 迭代式开发的好处
   1. 通过将**高技术风险的需求在早期迭代里实现**，有助于尽早暴露问题和及时消除风险
   2. 通过提供功能渐增的产品，持续从客户获得反馈，根据反馈及时调整，使最终产品更加符合客户的需要
   3. 通过小批量减少排队，提供更灵活、快速的交付能力
   4. 平滑人力资源的使用，避免出现瓶颈
2. **Scrum敏捷开发过程**
   1. 项目整个开发周期包括若干个小的**迭代周期**，每个迭代周期称为一个**Sprint**，每个Sprint的建议长度**2到6周**。
   2. 使用**产品Backlog**来管理项目的需求，产品Backlog是一个按照商业价值排序的**需求列表**，体现形式通常为**用户故事（UserStory）**。
   3. 团队从产品Backlog中挑选最有商业价值的需求，经过Sprint计划会议上的分析、讨论和估算得到**任务列表**，称为**Sprint Backlog**。
   4. 在每个迭代结束时，**Scrum团队**将交付潜在可交付的**产品增量**。

二、scrum敏捷开发涵盖内容



1. 敏捷团队组成

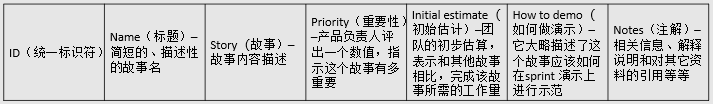
敏捷团队包括3个核心角色: **PO**(Product Owner 产品负责人)、**Scrum Master**(Scrum教练)、**Team**(开发团队)



1. scrum敏捷过程二
2. scrum敏捷工作件
3. 产品Backlog

经过优先级排序的动态刷新的**产品需求清单**，用来制定发布计划和迭代计划

产品Backlog是需求动态管理的载体



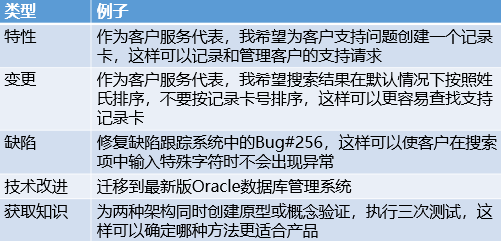
1. 产品Backlog的好处

通过需求的动态管理应对变化，避免浪费；

易于优先交付对用户价值高的需求

1. 产品Backlog关键要点
   1. **清楚表述列表中每个需求任务对用户带来的价值**，做为优先级排序的重要参考；
   2. **动态的需求管理而非“冻结”方式**，PO持续地管理和及时刷新需求清单，在每轮迭代前，都要重新筛选出高优先级需求进入本轮迭代；
   3. **迭代的需求分析过程，而非一次性分析清楚所有需求**（只对近期迭代要做的需求进行详细分析，其它需求停留在粗粒度）。
2. 产品功能列表

按优先级排序的预期产品功能集，通常组成包括



1. 完成标准

基于“随时可向用户发布”的目标制定衡量团队工作是否已完成的标准，由团队和PO形成共识

完成标准确保团队每一步前进都奠定在坚实的质量基础之上

·关键要点

团队自协商：团队根据项目实际情况来定义完成标准，并严格遵守；

有层次：一般分为三个层次：Story级别，迭代级和发布级，每个级别都有各自的完成标准。

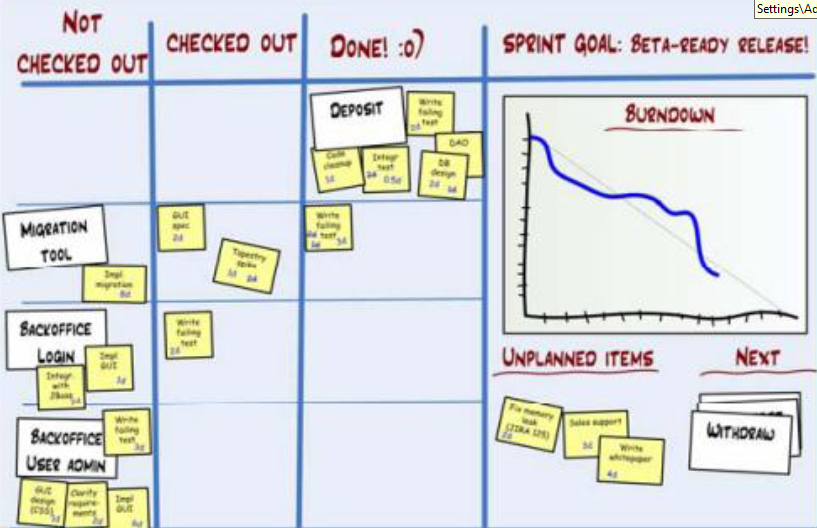
·完成标准的好处

共同协商的完成标准是团队的自我承诺，团队会更认真；

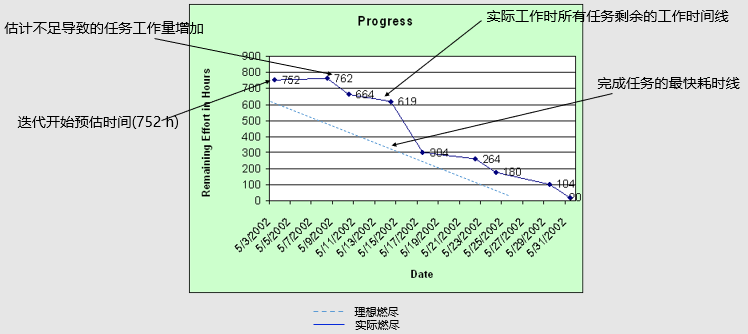
用于准确评估团队工作进展；

清晰和明确的完成标准保证了每次迭代是高质量的。

1. 任务看板



1. 燃尽图



二、scrum会议及过程实践

1、敏捷软件开发过程

1. PO和开发团队对产品业务目标形成共识
2. PO建立和维护**产品需求列表**（需求会不断新增和改变）,并进行优先级排序
3. PO每轮迭代前，Review需求列表，并筛选高优先级需求进入本轮迭代开发
4. 开发团队细化本轮迭代需求，并按照需求的优先级，依次在本轮迭代完成
5. 开发团队每日站立会议、特性开发、持续集成，使开发进度真正透明
6. PO对每轮迭代（2－4周）交付的可工作软件进行现场验收和反馈
7. 团队内部进行本轮冲刺的过程回顾，发现可改进的方面，指导下一轮迭代
   * 1. 迭代计划会议

迭代计划会议由团队共同确定迭代交付内容和完成标准

· **每轮迭代启动前**，团队共同讨论**本轮迭代详细开发计划的过程**，输入是**产品Backlog**，输出是**团队迭代Backlog**

* 1. **什么是迭代计划会议**
* 多团队迭代计划会议要分层召开

版本迭代计划会议：将产品Backlog（需求）分配给团队

团队迭代计划会议：将选取的产品Backlog需求转换成迭代Backlog（任务） ，分配给团队成员；

* 迭代计划会议内容：

**澄清需求、对“完成标准”达成一致**

**工作量估计、根据团队能力确定本轮迭代交付内容**

**细化、分配迭代任务和初始工作计划。**

* 1. 迭代计划会议的好处

通过充分讨论，使团队成员对任务和完成标准理解一致

团队共同参与，促进团队成员更认真对待自己的承偌。

* 1. 迭代计划会议的关键要点

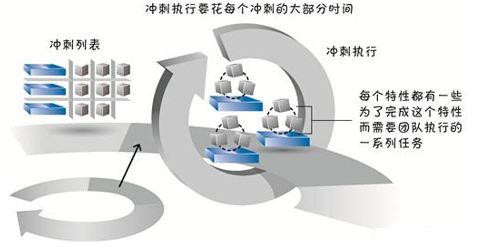
充分参与：**Scrum Master确保PO和Team充分参与讨论，达成理解一致**；

相互承诺：**Team承诺完成迭代Backlog中的需求并达到”完成标准“，PO承诺在短迭代周期不增加需求**；

确定内部任务：Team和PO协商把一些内部任务放入迭代中（例如重构、持续集成环境搭建等），由PO考虑并与其他外部需求一起排序 。

* + 1. 迭代执行

迭代执行包含规划、管理、执行和沟通工作，以确保创建可工作的、经过测试的特性



* + 1. 每日站立会议

每日站立会议促进团队沟通协调，及时暴露问题

* + - 1. 什么是每日站立会议

每日工作前，团队成员的例行沟通机制，由Scrum Master组织，Team成员全体站立参加

聚焦在下面的三个主题：**我昨天为本项目做了什么？我计划今天为本项目做什么？我需要什么帮助以更高效的工作？**

* + - 1. 每日站立会议的关键要点

准时开始：按计划会议制定的时间地点开会，形成团队成员的自然习惯；

高效会议：会议限时15分钟，每个人都保持站立，依次发言，不讨论与会议三个主题无关的事情（如技术解决方案等）；

问题跟踪：Scrum Master应该记录下所有的问题并跟踪解决；

* + - 1. **每日站立会议的好处**

增加团队凝聚力，产生积极的工作氛围

及时暴露风险和问题；

促进团队内成员的沟通和协调。

* + 1. 迭代评审会

迭代验收尽早演示可工作的软件，收集反馈意见

* + - 1. **什么是迭代验收**

**每次迭代开发结束时举行**，通过演示可工作的软件检查需求是否满足客户要求；

由Scrum Master组织， PO和用户代表（外部或内部利益相关人）负责验收、Team负责演示可工作软件

* + - 1. **迭代验收的关键要点**

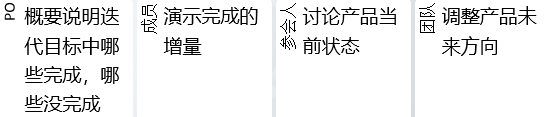
展示“真实”的产品：Team 应在真实环境中展示可运行的软件，判断是否达到“完成”标准；

收集反馈：PO 根据验收情况及客户反馈意见，及时调整产品Backlog

* + - 1. 迭代验收的好处

通过演示可工作的软件来确认项目的进度，具有真实性；

能尽早的获得用户对产品的反馈，使产品更加贴近客户需求。



* + 1. 迭代回顾会议

**迭代回顾会议是促进团队持续改进的最有效手段**

* + - 1. **什么是迭代回顾会议**

**在每轮迭代结束后举行的会议**，目的是分享好的经验和发现改进点，促进团队不断进步；

围绕如下三个问题：本次迭代有哪些做得好？本次迭代我们在哪些方面还能做得更好？我们在下次迭代准备在哪些方面改进？

* + - 1. 关键要点

会议气氛：Team全员参加，气氛宽松自由，畅所欲言，头脑风暴发现问题，共同分析根因；

关注重点：Team共同讨论优先级，将精力放在最需要的地方（关注几个改进就够了）；

会议结论要跟踪闭环：可以放入迭代backlog中。

* + - 1. **迭代回顾会议的好处**

激励团队成员；

帮助团队挖掘优秀经验并继承；

避免团队犯重复的错误；

营造团队自主改进的氛围。

三、总结

1、scrum组成

三个角色：产品负责人、敏捷教练、团队

四个会议：迭代计划会议、每日站会、迭代评审会议、迭代回顾会议

三个产物：Product Backlog、Sprint Backlog 、燃尽图

1. scrum特点

**适于在不确定性高的环境中开发复杂产品。**

**简洁但有效（**易于学习和掌握；能够在开发进程中不断检查，并作出相应调整）

**项目信息对所有干系人高度透明。**

**便于快速发现问题，促使团队和组织持续改进。**

十一、敏捷工程实践

1. 用户故事（user story）
2. 用户故事

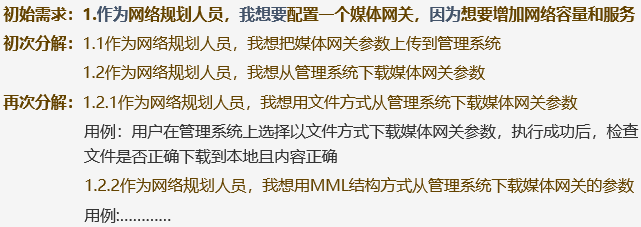
用户故事是一个用来确认用户和用户需求的简短描述

典型的描述句式为：作为一个XXX客户，我需要XXX功能，能够带来XXX好处

用户故事是站在用户角度描述需求的一种方式；

每个用户故事须有对应的验收测试用例；

用户故事是分层分级的，在使用过程中逐步分解细化



1. 关键要点

I – Independent，可独立交付给客户

N – Negotiable，便于与客户交流

V - Valuable ，对客户有价值

E - Estimable ，能估计出工作量

S - Small ，分解到最底层的用户故事粒度尽量小，至少在一个迭代中能完成

T - Testable，可测试

1. 用户故事的好处

站在用户视角便于和客户交流，准确描述客户需求；

用户故事可独立交付单元、规模小，适于迭代开发，以获得用户快速反馈；

用户故事强调编写验收测试用例作为验收标准，能促使需求分析人员准确把握需求，牵引开发人员避免过度设计。

1. 特征

·体现客户（用户）价值，轻量级的点位符

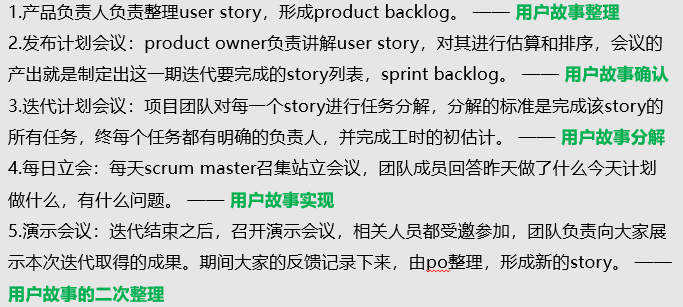
·遵循3C原则

卡片（Card）：作为XX，我希望XXX，这样可以XXX

对话（Conversation）：不描述到细节，由团队通过持续对话细化，激发大家的深度理解

确认（Confirmation）：有明确的验收标准，由测试人员完成，测试报告是对用户故事实现程度的最直接体现

1. 用户故事的流转过程（以Scrum为例）



1. 用户故事大小级别

史诗故事（1-2个月）

特性故事（1-2周）

冲刺故事（1-2天）

任务（几小时）：可分工执行

1. 用户故事invest标准

独立性（Independent）：故事之间松耦合，具有独立性，不应该依赖于其他的用户故事。

可协商（Negotiable）：开始仅用于做占位符，逐步细化。

有价值（Valuable）：用户故事对于最终的用户是有价值的，因此应该站在用户的角度去编写。

可估算（Estimatable）：设计、开发、测试团队可估算工作量和成本。（不可估算的原因：太大需要分解，或信息不全需要进一步探索）

短小（Small）：故事应该尽量的短小（如两周冲刺，故事一般是2天以内的）

可测试（Test）：有相应测试验收标准。

1. 结对编程

**结对编程提高代码质量和工作效率**

1. 什么是结对编程

两位程序员在一台电脑前工作，一个负责敲入代码，而另外一个实时检视每一行敲入的代码；

操作键盘和鼠标的程序员被称为“驾驶员”，负责实时评审和协助的程序员被称为“领航员”；

领航员检视的同时还必须负责考虑下一步的工作方向 ，比如可能出现的问题以及改进等。

1. 关键要点

程序员应经常性地在“驾驶员”和“领航员”间切换，保持成员间平等协商和相互理解，避免出现一个角色支配另一个角色的现象；

开始新Story开发的时候即可变换搭档，以增进知识传播；

培养团队成员积极、协作的心态能够增进结对编程效果；

实施初期需要精心辅导，帮助团队成员克服个性冲突和习惯差异。

1. TDD（测试驱动开发）

TDD**以测试作为编程的中心**，它要求在编写任何代码之前，首先编写定义代码功能的测试用例，编写的代码要通过用例，并不断进行重构优化

1. 持续集成

持续集成（CI）是一项软件开发实践，其中团队的成员经常集成他们的工作，通常每人每天至少集成一次，每次集成通过自动化构建完成

1. CodeReview
2. 发布规则

十二、极限编程

1. 极限编程
   1. 极限编程（eXtreme Programmin，简称xp）

极限编程是一种适用于**中小型团队**在**需求不明**或**快速多变**的情况下进行软件开发的轻量级软件开发方法论

极限编程中的“Extreme”（极限）是指将有效的软件开发原理和实践应用到极限

* 1. 特点

简单、快速、低缺陷率、适应需求变化

适用于2-12人的小型团队

* 1. 极限编程与敏捷开发的关系爱

极限编程不研究管理，也没有严格的项目进度管理（就是白板+贴纸），scrum涉及一点项目管理。与敏捷开发相对应的有“精益管理”



* 1. 目标

极限编程的主要目标在于降低因需求变更而带来的成本。

极限编程透过引入基本价值、原则、方法等概念来达到降低变更成本的目的

* 1. 极限编程的价值观
     1. 沟通

把沟通看作项目中间协调与合作的主要推动因素

* + 1. 反馈

软件开发不是执行预定计划的过程，而是根据反馈不断改进的过程

* + 1. 简单

在保证满足用户需求的前提下，尽量选择最简单的设计

* + 1. 尊重

团队成员间的互相尊重、团队成员对工作的尊重

* + 1. 勇气
  1. 优点

拥抱需求变化。

强调团队合作。

XP可以让开发者专注于编写代码，避免了不必要的文案工作及会议。

增强代码和产品质量，并有效的减少BUG。

程序员互相帮助，互相教对方，实现能力互补。

从公司管理的角度来看，这种方法可以减少对牛人的依赖。同时它也提升了员工满意度

* 1. 缺点

缺乏设计文档，局限于小规模项目

缺乏质量规划

没有提供数据的收集和使用的指导

开发过程不详细

全新的管理手法带来的认同度问题

1. 极限编程实践
   1. 现场客户（On-site Customer）

要求至少有一名实际的客户代表在整个项目开发周期在现场负责确定需求、回答团队问题以及编写功能验收测试

* 1. 计划游戏（Planning Game）

要求结合项目进展和技术情况，确定下一阶段要开发与发布的系统范围

* 1. 系统隐喻（System Metaphor）

通过隐喻来描述系统如何运作、新的功能以何种方式加入到系统。通常包含了一些可以参照和比较的类和设计模式

* 1. 简单设计（Simple Design）

认为代码的设计应该尽可能的简单，只要满足当前功能的要求，不多也不少

* 1. 代码集体所有（Collective Code Ownership）

开发小组的每个成员都有更改代码的权利，所有人对于全部代码负责

* 1. 结对编程（Pair Programming）
  2. 测试驱动（Test-driven）
  3. 小型发布（Small Releases）

强调在非常短的周期内以递增的方式发布新版本，从而可以很容易地估计每个迭代周期的进度，便于控制工作量和风险，及时处理用户的反馈

* 1. 重构（Refactoring）

重构是在不改变软件外部行为的前提下改善其内部结构

模块重构的级别：函数级整理、对象级整理、业务逻辑的整理

·一次重构的步骤

* 1. 发现“代码坏味”（即代码中不符合设计原则的部分）
  2. 使用重构消除代码坏味
  3. 自动测试，保证重构没有改变代码外部行为

不断重复以上三步，改进代码的设计

·代码坏味儿

* + 1. 不可读

词不达意——函数改名

超长函数——抽取函数

过长参数列——用函数取代参数

抽象层级不一致——抽取函数

* + 1. 不简洁

重复代码——合并代码

重复逻辑——合并逻辑

过度设计——移除间隔层

* + 1. 不面向对象

过程代码——生成对象

* + 1. 不单一职责

发散式变化——切分代码

代码耦合——添加间隔层 or 依赖注入

散弹式修改——重组代码

层间耦合——领域层与界面层分离 or 软件与硬件分离

·为什么要“增加函数减少参数”？

* 1. 提高代码可读性
  2. 减少代码间的耦合
  3. 降低代码复杂度，易于维护

·什么时候重构？

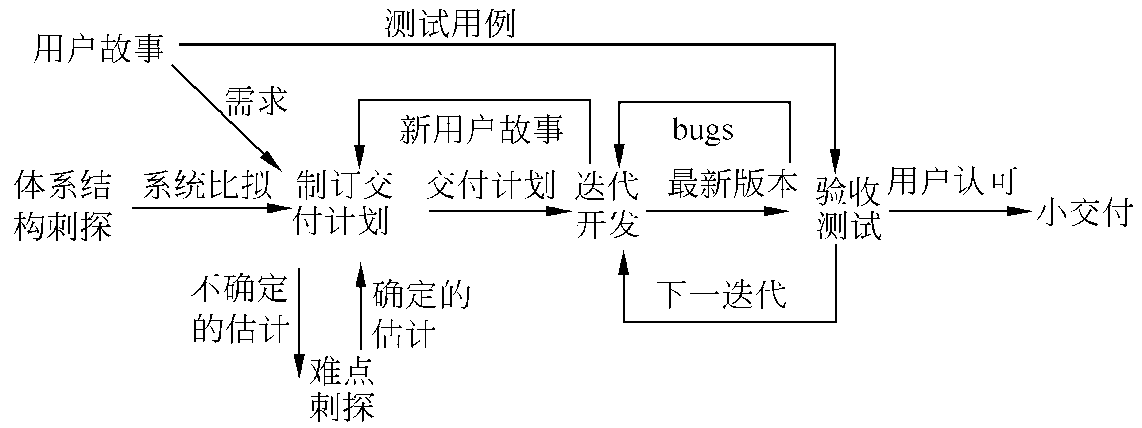
任何修改代码的时候都需要重构。如添加新功能之前、修改bug之后都需要重构。也可以抽专门的时间来重构

* 1. 持续集成（Continuous integration）
  2. 每周40小时工作制（40-hour Weeks）

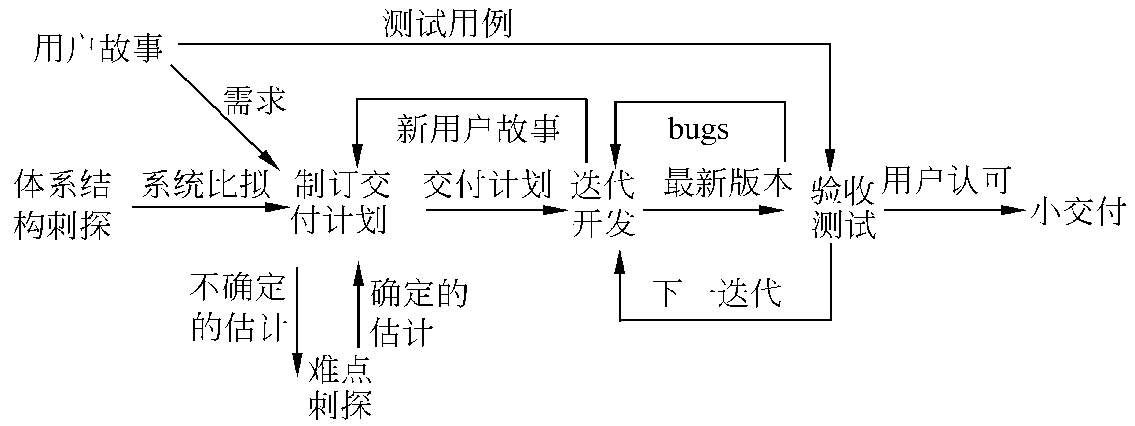
要求项目团队人员每周工作时间不能超过40小时，加班不得连续超过两周，否则反而会影响生产率

* 1. 代码规范（Coding Standards）

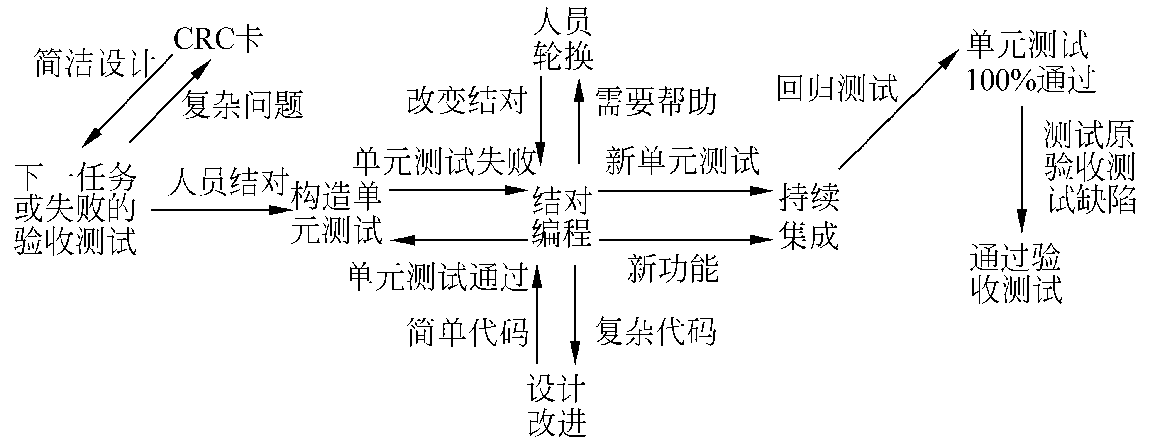
1. 极限编程开发过程
   1. xp项目开发过程



* 1. xp迭代开发过程



* 1. xp的代码共享编程



* 1. xp与scrum的相同点

都属于敏捷开发

都追求短周期交付，持续交付

都注重交付产品的质量

都注重反馈

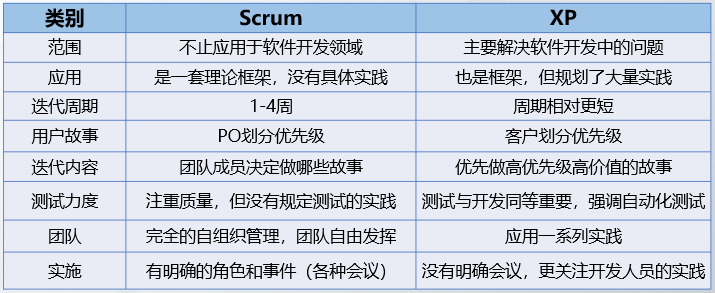
都使用迭代的方式进行开发

都有固定的迭代周期

都使用用户故事描述

都积极响应客户变化

* 1. xp与scrum的不同点



十三、重构

* + 1. 重构介绍
       1. 基本概念

重构（名词）：对软件内部结构的一种调整， 目的是在不改变软件可观察行为的前提下， 提高其可理解性， 降低其修改成本。

重构（动词）：使用一系列重构手法， 在不改变软件可观察行为的前提下，调整其结构

重构的关键在于运用大量微小且保持软件行为的步骤， 一步步达成大规模的修改。 每个单独的重构要么很小， 要么由若干小步骤组合而成

* + - 1. 重构与优化

重构与性能优化有很多相似之处： 两者都需要修改代码， 并且两者都不会改变程序的整体功能

两者的差别在于其目的：

**重构是为了让代码“更容易理解，更易于修改”。** 这可能使程序运行得更快， 也可能使程序运行得更慢。

在性能优化时， 我只关心让程序运行得更快， 最终得到的代码有可能更难理解和维护。

* + - 1. 整体流程
  1. 初识程序

重构的第一个步骤永远相同： 确保即将修改的代码拥有一组可靠的测试

* 1. 分解函数
  2. 拆分计算
  3. 分离文件

分离文件的目标是根据逻辑功能将不同逻辑的代码分到对应的文件中

虽然代码的行数增加了， 但重构也带来了代码可读性的提高。 额外的包装文件将混杂的逻辑分解成可辨别的部分， 分离了详单的计算逻辑与样式

* 1. 逻辑重构

逻辑重构的目标是根据实际功能需求，引入类型的多态性

* + - 1. 为什么重构

重构是一个工具， 它可以用于以下几个目的：

* + 1. 重构改进软件的设计
    2. 重构使软件更容易理解
    3. 重构帮助找到bug
    4. 重构提高编程速度
       1. 何时重构

应该把重构融入到编码工作中，其中几个重构的时间点：

* 1. 预备性重构：让添加新功能更容易
  2. 帮助理解的重构：使代码更易懂
  3. 捡垃圾式重构
  4. 有计划的重构和见机行事的重构
  5. 长期重构
  6. 复审代码时重构
     + 1. 重构的挑战

1. 延缓新功能开发
2. 代码所有权
3. 分支
4. 测试
5. 遗留代码
6. 数据库
   * 1. 代码重构
        1. 命名

函数声明、变量改名、字段改名等

* + - 1. 重复代码
      2. 过长函数
      3. 过长参数列表
      4. 全局数据
      5. 过大的类
      6. 纯数据类
    1. 函数重构
       1. 提炼函数
       2. 内联函数
       3. 提炼变量
       4. 内联变量
       5. 改变函数声明
       6. 变量改名
    2. 类级别重构
       1. 封装记录
       2. 封装集合
       3. 以对象取代级别类型
       4. 以查询取代临时变量