## 软件需求分析思考题

黄永涛 41912210

**01**

1. **现代软件工程如何处理软件开发所具有的本质困难？**

答：软件工程的本质困难是软件固有的复杂性、一致性、可变性和不可见性的产物，这4点也被称作软件开发中的不变事实，是不可控因素。人们就应该着手解决软件工程的偶然因素——由于软件生产实践带来的困难能够由人工干预来解决。1.利益相关者：软件项目的成功由社会因素确定，技术是第二位的。2.选择合适的过程模型：说明执行活动的次序；说明需要交出什么样的制品，以及什么时候交出；将活动和制品分配给开发者；提供监控项目进程、评估产出和计划未来项目的准则。3.合适的建模语言和工具：开发人员需要一种语言来创建可视化的和其他形式的模型，以及与客户和其他开发人员进行的讨论。这个语言应该支持各个抽象层次上模型的构造，以在不同程度上详细表示所提出的解决方案。

**二、需求工程包含哪些主要的活动？如何理解“描述”是需求工程的核心？**

答：需求工程包块需求开发和需求管理。其中，需求开发包括问题获取、建模、分析、编写规格说明、验证等。通常，需求描述系统要做什么，即系统的设计目的，是置身系统外部，对应用领域性质的描述，只有理清系统要做什么，才能在软件开发的下一个阶段中清晰地知道该怎么做。

1. **结合本节讲义，完整的描述什么是需求。**

答：I E E E软件工程标准词汇表（ 1 9 9 7年）中定义需求为：

（1）用户解决问题或达到目标所需的条件或权能（ C a p a b i l i t y）。

（2）系统或系统部件要满足合同、标准、规范或其它正式规定文档所需具有的

条件或权能。

（3）一种反映上面（1）或（2）所描述的条件或权能的文档说明。

软件需求包括三个不同的层次：业务需求、用户需求和功能需求—也包括非功能需求。业务需求反映了组织机构或客户对系统、产品高层次的目标要求，它们在项目视图与范围文档中予以说明。用户需求文档描述了用户使用产品必须要完成的任务，这在使用实例文档或方案脚本说明中予以说明。功能需求定义了开发人员必须实现的软件功能，使得用户能完成他们的任务，从而满足了业务需求。所谓特性是指逻辑上相关的功能需求的集合，给用户提供处理能力并满足业务需求。作为补充，软件需求规格说明还应包括非功能需求，它描述了系统展现给用户的行为和执行的操作等。它包括产品必须遵从的标准、规范和合约；外部界面的具体细节；性能要求；设计或实现的约束条件及质量属性。所谓约束是指对开发人员在软件产品设计和构造上的限制。质量属性是通过多种角度对产品的特点进行描述，从而反映产品功能。多角度描述产品对用户和开发人员都极为重要。值得注意的一点是，需求并未包括设计细节、实现细节、项目计划信息或测试信息。需求与这些没有关系，它关注的是充分说明你究竟想开发什么。一个优秀的需求应该是清楚、完整、一致、具有可测试性的。

**02**

1. **获取原始需求的传统方法和现代方法各有哪些？**

答：传统方法：与客户和领域专家面谈、问卷法、观察、文档和软件系统的研究

现代方法：原型法、联合应用开发、快速应用开发、头脑风暴

1. **需求管理都包含哪些工作？如何执行的？**

答：1.需求识别与分类：统一标识符与编号，要有惟一的标识符、要在文档层次之内连续编号、在需求的种类之内连续编号。

2.建立需求层次化结构：需求可以按父子关系建立层次化结构。父子关系与组合关系相似，父级需求由各种子级需求组成，子级需求是父级需求有效的子需求。需求的层次允许定义在不同抽象层次上的需求，这与在向低抽象层次发展时应系统地细化模型全面的建模原则是一致的。结果是，高级模型可以构造成父级需求，而低级模型可以作为子级需求链接上去。

3.变化管理：需求是变化的。在开发生命周期的任何阶段，需求都可以被改变、可以被删除或者可以增加新的需求。变化不是灾难，但没有管理的变化却是。由于需求的变化是开销很大的，因而对每一个变化请求必须建立正式的业务用例。一个有效的变化（以前没有处理的）必须从技术可行性、对项目其他部分的影响以及开销上进行估计。一旦批准，变化就被结合进相关的模型，并在软件中实现。

4.需求可跟踪性：需求可跟踪性是变化管理的一部分。需求可跟踪性模块对从需求出发贯穿整个开发生命周期的变化维护一个可跟踪的关系。可跟踪性关系可以在连续的生命周期阶段穿越许多模型。只有相邻的可跟踪性链接可以被直接修改。

1. **需求文档包含哪些内容？**

答：1.引言：引言部分主要包括需求背景、需求目的、需求概要、涉及范围、全局规则和名词说明，交互原型地址等。引言部分的写作目的是让阅读者快速理解需求背景和概要。如果是公司内部文档，引言部分可以从简写作。  
 2.业务建模：建模的目的是为了帮助阅读对象更好的理解需要开发的需求，常用的模型种类包括：用例图、实体图、状态图、流程图等。常用的建模语言如UML。UML具体的建模方法请戳这里。  
 3.业务模块：业务模块包含具体页面的元素、用例规则，以及相关的原型，流程图。

**03**

1. **什么是需求规格说明？与需求确定的关系？**

答：在需求分析阶段，将完整、一致的需求与能够满足需求的软件行为以文档的形式明确地固定下来，此文档就是需求规格说明。在一个复杂软件系统的开发中，编写需求规格说明文档是非常必要的。一方面，清晰、明确、结构化的文档可以将软件系统的需求信息和解决方案更好地传递给所有的开发者。另一方面，文档可以拓展人们的知识记忆能力，便于软件需求的客观固定和随时获取。软件需求规格说明不仅是系统测试和用户文档的基础，也是所有子系列项目规划、设计和编码的基础。对于提供什么软件产品，为客户和供方之间的协议建立基础，减少开发工作，为估计成本和进度提供基础，为验证和确定提供基线，便于软件产品转移，作为进一步增强的基础。

1. **如何理解体系结构优先权？**

答：软件体系结构是[软件设计](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E8%AE%BE%E8%AE%A1" \t "https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84/_blank)过程中的一个层次，这一层次超越计算过程中的算法设计和数据结构设计。体系结构问题包括总体组织和全局控制、通讯协议、同步、数据存取，给设计[元素分配](https://baike.baidu.com/item/%E5%85%83%E7%B4%A0%E5%88%86%E9%85%8D" \t "https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84/_blank)特定功能，设计元素的组织，规模和性能，在各设计方案间进行选择等。软件体系结构处理[算法与数据结构](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95%E4%B8%8E%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84" \t "https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84/_blank)之上关于整体系统结构设计和描述方面的一些问题，如全局组织和全局控制结构、关于通讯、同步与数据存取的协议，设计构件功能定义，物理分布与合成，设计方案的选择、评估与实现等。软件体系结构有四个角度，它们从不同方面对系统进行描述：概念角度描述系统的主要构件及它们之间的关系；模块角度包含功能分解与层次结构；运行角度描述了一个系统的动态结构；代码角度描述了各种代码和库函数在开发环境中的组织。如果软件体系结构没有确定，软件需求分析将无从谈起，只有先确定好软件系统的体系结构，以后的软件开发工作才能顺利进行，才能开发出令客户满意的产品。

1. **需求规格说明包括哪些形式？**

答：1.引言：包含目的、背景、参考资料等模块；

2.项目概述：包含产品概述、产品功能、用户场景、一般约束、假设与依据等模块；

3.具体需求：包括功能图、功能需求、外部接口需求（用户接口、硬件接口、软件接口、

通信接口）、属性（可用性、安全性、可维护性）等模块；

1. 验证验收表标准：包括文档验收标准、软件验收标准、界面验收标准、功能验收准。

**04**

1. **如何建立需求规格说明？**

答：需求规格说明的建立应该遵循面向对象分析（OOA）的方法，主要步骤如下：

1.体系结构分析（AA）。识别和分析对体系结构有影响的非功能性需求。对于那些对体系结构有重大影响的需求，分析替代方案并创建解决影响的解决方案。“架构”关注点特别与非功能需求相关，并包括对应用程序的业务或市场上下文的了解。同时，功能要求(如加工销售等)不可忽视;它们提供了必须在其中解决这些问题的环境。此外，识别它们的可变性在架构上是重要的。架构关注涉及系统级、大规模和广泛的问题，这些问题的解决通常涉及大规模或基本的设计决策，例如应用程序服务器的选择甚至使用，相互依赖和权衡。

2.领域驱动设计（DDD）。在大型团队的系统设计和实现过程中，确保业务架构和技术架构能够统一已经成为系统开发成败的关键。而领域驱动设计为我们提供了一种软件开发方法，强调开发人员与领域专家协作交付业务价值，强调把握业务的高层次方向，也强调系统建模工具和方法以满足技术需求。领域驱动设计思想的核心就是认为系统架构应该是业务架构和技术架构相结合的一种过程，并提供了一系列的设计相关工具和模式确保实现这一过程。

3.职责驱动设计（RDD）。随着软件业的不断发展，随着软件需求的不断扩大，软件所管理的范围也在不断拓宽。过去一个软件仅仅管理一台电脑的一个小小的功能，而现在被扩展到了一个企业、一个行业、一个产业链。过去我们开发一套软件，只有少量的二次开发，当它使用到一定时候我们就抛弃掉重新又开发一套。现在，随着用户对软件依赖程度的不断加大，我们很难说抛弃一套软件重新开发了，更多的是在一套软件中持续改进，使这套软件的生命周期持续数年以及数个版本。正是因为软件业面临着如此巨大的压力，我们的代码质量，我们开发的软件拥有的可变更性和持续改进的能力，成为软件制胜的关键因素，令我们不能不反思。对象的行为职责包括：自身执行的一些行为，如创建对象或计算；初始化其他对象中的动作；控制和协调其他对象的活动。对象的认知职责包括：对私有封装数据的认知；对相关对象的认知；对其能够导出或计算的事物的认知。

4.分析类和用例实现。分析类代表了“系统中具有责任和行为的事物”的早期概念模型，用于捕获系统对象模型的“初稿”和粗略剪裁。分析类主要处理功能需求，它们从问题领域建模对象。用例实现即是实现某个用例中的某个场景，因此也叫场景实现。另一方面，用例的实现描述了一组交互对象，这些对象将支持用例所需的功能。

## 读书笔记

### 面向对象分析与面向对象设计

为对象分配职责：在OO开发中，最重要的就是为对象分配职责。因为分配职责是必须进行的一项工作，并且它对软件构件的健壮性、可维护性和可重用性具有重要的影响。  
  相比起在OOA/D中的其他技能，职责的分配是一项难以掌握又至关重要的技能，即使在实际项目中，开发人员没有时间进行其他建模活动，而只能完成仓促编码的开发过程，分配职责这项工作却也是必不可少的。

面向对象分析：分析是指对问题和需求的调查研究。面向对象分析（object-oriented analysis,OOA），指在问题领域内发现和描述对象。

面向对象设计：设计是指满足需求概念上的解决方案。面向对象设计（object-oriented design,OOD），指的是定义软件对象以及它们如何协作以实现需求。

简单示例：书中使用“骰子游戏”进行示例。

定义用例：用例指使用的某些情节或场景，比如说投骰子后，骰子总点数是7，则赢，否则输。

定义领域模型：将现实世界中的概念和想象可视化。例如下图，玩家（player）投掷了两个骰子（die），骰子游戏中有一个玩家和两个骰子，这些信息都能在图中展示出来。

定义交互图：交互图有很多种，主要表现软件对象之间的消息流和由消息引起的方法调用。在骰子游戏中，骰子游戏给骰子发送了投骰子的消息，虽然在现实世界中是玩家掷出骰子，但是在软件设计中其实是骰子游戏发送出消息。

定义设计类图：设计类图用来描述对象的属性和方法，例如在骰子游戏中，骰子游戏发送投骰子的消息，那么就需要play方法，同理骰子就需要roll和getFaceValue方法。  
 UML的三种应用方式：

UML草图：非正式的、不完整的图，借助可视化语言，探讨或解决问题的某一复杂部分。

UML蓝图：相对详细的设计图，主要用于逆向工程和代码生成（前向工程）。  
  在逆向工程中，将代码转换为可视化的图，使代码更易理解。  
  在代码生成中，代码自动生成工具利用UML图生成一些代码，然后开发者再编写和填充其他代码。

UML作为编程语言：可执行代码可以直接被生成，但是开发者并不能看到或者编辑这些代码，而是去将UML作为编程语言来进行使用。

敏捷建模：指UML的草图应用形式，通常对时间的投入具有高回报，即费时较短。

应用UML的三种透视图：

概念透视图：用图来描述现实世界或者概念领域中的事物，在这种图中的类，我们称之为概念类。

规格说明透视图：用图来描述软件的抽象或具有规格说明的构件，在该图中的类称为软件类。

实现透视图：用图来描述特定技术中的软件实现，该图中的类成为实现类。

### 第二章 面向对象分析与面向对象设计

UP：软件开发过程描述了构造、部署以及维护软件的方式。统一过程（Unified Process）是一种流行的构造面向对象系统的软件开发过程。  
 1.UP是迭代过程。迭代开发对OOA/D的最佳实践具有影响。  
 2.UP实践提供了如何实施OOA/D的示范结构。  
 3.UP具有灵活性，可以应用于轻量级和敏捷方法。

迭代开发：迭代开发（iterative development）是UP和大多数其他现代方法中的关键实践。在这种方法中，开发被组织成一系列固定的短期小项目，称为迭代（iteration）；每次迭代都产生经过测试、集成并可执行的局部系统；每次迭代都具有各自的需求分析、设计、实现和测试活动。

迭代生命周期基于对经过多次迭代的系统进行持续扩展和精化，并以循环反馈和调整未核心驱动力，使之最终成为适当的系统，随着一次又一次迭代的递进，系统增量式地发展完善，因此这一方法也被称为迭代和增量式开发。因为反馈和调整使文档规格说明和设计不断进化，所以这种方法也称为迭代和进化式开发

在迭代项目中处理变更：一方面认同和稳定一组需求，另一方面接受需求不断变更的事实。在早期迭代中，对需求和设计的选择对于最终期望来说可能不准确，但是快速地实施一小步的方式可以得到可能来自于用户、开发人员和测试的快速反馈，这种早期反馈具有极高的价值，可以从这些反馈中挖掘出至关重要和实际的观点，并修改和调整对需求或设计的理解。

最好及早解决和验证具有风险的、关键的设计决策，为迭代开发提供完成这项工作的机制。

因为工作是通过一系列有序的构造-反馈-调整循环向前进展的，因此变更是正常情况。

迭代开发的优点：减少项目失败可能性，提高生产率，降低缺陷率。

在早期缓解高风险。

早期可见的进展。

早期反馈、用户参与和调整，会产生更接近真实需求的精化系统。

可控复杂性；团队不会被分析瘫痪或长期且复杂的步骤所淹没。

一次迭代中的经验可以被系统用于改进开发过程本身，并如此反复进行下去。

一次迭代的持续时间和时间定量

2~6周之间，时间过长会破坏迭代开发的核心动机并增加项目风险。太短的迭代时间不足以获得有意义的产出和反馈。

迭代的一个关键思想是时间定量，假如分配好任务后看起来难以满足期限要求，那么应该去除一些任务和需求而不是推迟完成日期。

风险驱动和客户驱动的迭代计划

早期的迭代计划目标要能够识别和降低最高风险，并且能构造客户最关心的可视化特性。

瀑布生命周期

在瀑布（或顺序）生命周期过程中，试图在编程之前详细定义所有或大部分需求，而且通常于变成之前创建出完整的设计，定义“可靠的”计划或时间表，但常常事与愿违。

如果出现开发前确认大多数需求，编程前试图创建完整、详细的规格说明或UML模型和设计，说明瀑布思维已经在无情地折磨着这个项目了

现在的研究表明，瀑布方法对于大多数软件项目是拙劣的实践，平均而言，瀑布方法需求中45%的特性从未被使用，其早期时间表和估计与最终实际情况可相差400%。

避免瀑布

“在开始编程前编写完所有的用例“或者“让我们在开始编程前用UM完成更多详细的OO模型”，这样的想法是不健康的瀑布思维。

反馈和改写的必要性

瀑布模型没有认识到变更对于软件项目来说是永恒的，是必然存在的事物，没有正视和包容它。在大多数软件项目中，反馈和调整是成功的关键要素。

敏捷宣言

个体和迭代，超越过程和工具。  
工作的软件，超越完整的文档。  
客户协作，超载合同谈判。  
响应变更，超载履行计划。

敏捷开发

通常应用时间定量的迭代和进化式开发、使用自适应计划、提倡增量交付并包含其他提倡敏捷性（快速和灵活的相应变更）的价值和时间。

敏捷原则：

优通过早期和持续交付有价值的软件来满足客户。

欢迎变更需求，敏捷过程为客户的竞争优势而控制变更。

以两周到两月为周期，频繁地交付可运行的软件，首推最短的时间定量。

每一天开发人员都要和业务人员合作。

由个体推动项目的建设，为个体提供所需的环境、支持和信任。

团队间最为有效和高效的方法是面对面交谈。

衡量进展的重要尺度是可运行的软件。

敏捷过程提倡可持续的开发。

发起人、开发者和用户应该步调一致。

不断地关注技术上优越的设计会提供敏捷性。

简洁是最重要的，简洁就是尽量减少工作量的艺术。

最佳的架构、需求和设计来自于自组织的团队。

团队要定期反省如何使工作更有效，然后相应地调整行为。

敏捷建模

建模的真正行为能够并且是应该能够对理解问题或解决方案空间提供更好的方式。所以“实行UML”，不是创建大量详细的UML图并递交给编程者，而是为良好的OO设计快速探索可选的方案和途径。

采用敏捷方法并不意味着不进行任何建模。

建模和模型的目的主要用于理解和沟通。

建模时只需对设计空间中不常见、困难和几首的一小部分问题建模和应用UML，对于简单的设计问题可以推延到编程阶段。

尽可能使用简单的工具。

不要单独建模，尽量使每个人参与其中。

敏捷UP

采纳和应用可适应性和轻量级实践的UP成为敏捷UP。

不断地让用户参与评估、反馈和需求。

在早期迭代中建立内聚的核心架构。

不断地难质量；提早、经常和实际地测试。

在适当的地方使用用例。

进行一些可视化建模。

认真管理需求。

实行变更请求和配置管理。

UP的阶段

初始：大体上的构想、业务案例、范围和模糊评估。（立项阶段）

细化：已精化的构想、核心架构的迭代实现、高风险的解决、确定大多数需求和范围以及进行更为实际的评估。（功能需求）

构造：对遗留下来的风险较低和比较简单的元素进行迭代时限，准备部署。（开发实现）

移交：进行beta测试和部署。（交付使用）

UP科目

UP描述了科目中的工作活动。科目也称为流程。UP科目也就是UP流程。  
- 业务建模：领域模型制品，使应用领域中的重要概念的可视化。  
- 需求：用以不糊功能需求和肺功能需求的用例模型及其补充性的规格说明制品。  
- 设计：设计模型制品，用于对软件对象进行设计

初始阶段

大多数项目需要一个简短的起始步骤，在这个步骤中要考虑以下问题：

项目的设想和业务案例是什么；

是否可行；

购买还是开发；

粗略估计一下成本；

项目应该继续下去还是停止；  
初始阶段并不是定义所有需求，或产生可信的预算和项目计划。  
大多数需求分许是在细化阶段进行的，并且伴以具有产品品质的早期编程和测试  
主要任务：预见项目的范围、设想和业务案例。  
解决问题：涉众是否就项目设想基本达成一致，项目是否值得继续进行认真研究。

初始阶段的持续时间

可能只包含第一次需求研讨会，并为第一次迭代制定计划，然后就快速进入细化阶段。

初始阶段会创建的制品

在初始阶段只完成其中部分制品，在后继迭代中对其进行精化。  
制品样例：

设想和业务用例；

用力模型，描述功能需求；

补充性规格说明，描述其他需求，主要是非功能性需求；

词汇表，关键领域术语和数据字典；

风险列表和风险管理计划，描述风险及应对和缓解的方法；

原型和概念验证；

迭代计划，描述第一个细化迭代的任务；

阶段计划和软件开发计划，对细化阶段的持续时间和工作量进行粗略的估计；

**第三章 案例研究**

开发案例；

要有选择性地创建对项目确有价值的制品，如果其价值未被证实，则放弃。重要的不是在初始阶段创建完整的规格说明，而是形成初始、概略的文档，这些文档将在细化阶段中精化，以便响应由早期编程和测试得到的极有价值的反馈。

一般在不同项目中，制品都可能存在大量的相似之处，因此所有UP项目都应该用相同的方式来组织制品，以便从以往的项目中找出能够重用的制品。

**第二部分 初始阶段**

在初始阶段，应该详细编写10%到20%的用例以便获得对问题范围的真实认知。  
在初始阶段不会引入大量的图形。

需求

需求就是系统必须提供的能力和必须遵从的条件。

UP不主张采用瀑布的观点。

UP推崇用一种系统的方法来不断寻找、记录、组织和跟踪不断变更的需求，简而言之，就是通过迭代巧妙地进行需求分析。

需求分析的最大挑战是寻找、沟通和记录什么是真正需要的，并能够清楚地讲解给客户和开发团队的成员。

进化式需求与瀑布式需求

两者的核心差异在于UP能够包容需求中的变更，并将其作为项目的基本驱动力

瀑布思想的错误根源在于，将软件项目与大规模制造业项目视为等同，但后者是可预测的、低变更率的，前者则是具有高变更率的新产品开发领域，具有大量新奇事物，需要大量的发现和探索。

据统计，软件项目的平均需求变更率为25%。试图使用瀑布式需求分析时，早期定义的特性中，有45%的特性从未使用，有19%的特性罕有使用，几乎65%的瀑布式定义的特性少有或根本没有价值。

结合早期时间定量的迭代开发，进行迭代和进化式需求分析，并且引入频繁的涉众参与、评估和对局部结果的反馈。

寻求需求可以采用的方法

例如与客户一起编写用例或者参加需求讨论会，向客户演示每次迭代的成果以求得反馈。

UP欢迎任何能够带来价值并提高用户参与度的需求启发方法。

需求的类型和种类

在统一过程中，需求按照“FURPS+”模型进行分类（FURPS是一种识别软件质量属性的模型）。

功能性（function）：特性、功能、安全性。

可用性（usability）：人性化因素、帮助、文档。

可靠性（reliability）：故障频率、可恢复性、可预测性。

性能（performance）：响应时间、吞吐量、准确性、有效性、资源利用率。

可支持性（supportability）：适应性、可维护性、国际化、可配置性。

“+”指一些辅助性和次要的因素。

实现（implementation）：资源限制、语言和工具、硬件等。

接口（interface）：强加于外部接口系统之上的约束。

操作（operation）：对其操作设置的系统管理。

包装（packaging）：例如物理的包装盒。

授权（legal）：许可证或其他方式。

使用FURPS+作为需求范围的检查列表，可以避免遗漏系统某些重要方面。  
可用性、可靠性、性能和可支持性的这类需求统称为质量属性、质量需求，这类需求对系统架构具有极大影响。  
在一般使用中，需求按照功能性和非功能性来分类。

UP制品如何组织需求

关键的需求制品包括：

用例模型，一组使用系统的典型场景，主要用于功能需求。

补充性规格说明，基本上是用例之外的所有内容，主要用于所有非功能性需求。也用来记录不能表示为用例的功能特性，例如报表生成。

词汇表，定义重要的术语并包含数据字典的概念。（数据字典是指对数据的数据项、数据结构、数据流、数据存储、处理逻辑、外部实体等进行定义和描述，其目的是对数据流程图中的各个元素做出详细的说明，使用数据字典为简单的建模项目）。

设想，概括了高阶需求，这些需求在用例模型和补充性规格说明中进行细化，是简短的执行概要文档，用以快速了解项目的主要思想。

业务规则，也叫作领域规则，通常描述了凌驾于某一软件项目的需求或政策，这些规则是领域或业务所要求的，例如政府的政策。

用例是文本形式的情节描述，广泛应用于需求的发现和记录工作中。

用例和用例模型

用例模型是所有书面用例的集合，同时也是系统功能性和环境的模型。

用例是文本文档，而不是图形，用例模型主要是编写文本的活动，而非制图。

用例有利于用户参与项目，比如领域专家或需求提供方可以自己编写用例，而且用例强调了用户的目标和观点。

用例的优越性在于，能够根据需要对复杂程度和形式化程度进行增减调节。

用例就是需求，主要是说明系统如何工作的功能性或行为性需求。

在统一过程和其他现代方法中，用例被推荐为发现和定义需求的核心机制。

用例的主要思想：为功能性需求编写用例，从而降低详细的老式特性列表的重要性或减少这种列表的使用。

参与者的三种类型

参与者是任何具有行为的事物。

主要参与者，具有用户目标，并通过使用系统的服务完成。例如收银员。

协助参与者，为系统提供服务，一般是计算机系统。比如说自动付费服务授权。

幕后参与者，在用例行为中具有影响或利益，但不是主要或协助参与者。例如政府税收机构。

表示法：用例的三种常用形式

摘要，简洁的一段式概要，通常用于祝成功场景。例如示例中的处理销售。

非正式，非正式的段落格式，用几个不同的段落覆盖不同场景。例如示例中的处理退货。

详述，详细编写所有步骤及各种变化，同时具有补充部分，比如前置条件和保证成功。

示例：详述风格的处理销售

详述用例是结构化的，它展示了更多的细节，并且更为深入。

在迭代和进化式UP需求分析中，第一次需求讨论会应该以这种形式编写10%的关键用例，然后对这10%中最具有重要架构意义的用例或场景进行设计和编程。

详细用例格式的模板采用以下风格，是目前使用最为广泛的格式：

具体：

用例名称：以动词开始。

范围：范围界定了索要设计的系统。通常用例描述的是对一个软件系统（或硬件加软件）的使用，这种情况下称之为系统用例。在更广义的范围上，用例也能够描述顾客和有关人员如何使用业务。这种企业级的过程描述被称为业务用例。

级别：用例主要分为用户目标级别或子功能级别。用户目标级别是通常所使用的级别，描述了实现主要参与者目标的场景，该级别大致相当于业务流程工程中的基本业务流程。子功能级别用例支持用户目标所需要的自步骤，当若干常规用例共享重复的自步骤时，将其分离出来，创建为子功能级别用例（以避免重复公共的文本），比如信用卡支付，该用例可以被许多常规用例所共享。

主要参与者：调用系统服务来完成目标的参与者。

涉众及其关注点列表：该列表建议并界定了系统必须要做的工作。用例应该包含满足所有涉众关注点的事物，在编写用例其余部分之前就确定涉众及其关注点，能够让我们更加清楚地了解详细的系统职责。从涉众及其关注点的角度出发，能够为发现并记录所有行为需求提供彻底、系统的过程。

前置条件：给出在用例中场景开始之前必须永远为真的条件。通常前置条件隐含已经成功完成的其他用例场景，例如“登录”。注意有些条件也必须为真，但是不值得编写出来，例如“系统有电力供应”。前置条件传达的是编写者认为应该引起读者警惕的那些值得注意的假设。

成功保证：给出用例成功结束后必须为真的事物，包括主成功场景及其替代路径。该保证应该满足所有涉众的需求。

主成功场景和步骤（或基本流程）：也被称为“理想路径”场景，或“基本流程”及“典型流程”。它描述了满足涉众关注点的典型成功路径，要注意的是，它通常不包括任何条件或分支，保持一定的连贯性，并且将所有条件处理都推延至扩展部分，这样的做法更易于理解和扩展。场景记录三种步骤：1.参与者之间的交互；2.确认过程（通常由系统来完成）；3.系统完成的状态变更（例如记录或更改某事物）；

扩展：扩展是重要的并通常占据了文本的大部分篇幅。扩展部分描述了其他所有场景或分支，包括成功和失败路径。在整个用例编写中，理想路径与扩展路径相结合应该满足“几乎”所有涉众所关注的问题。但是有些关注问题最好是作为非功能性需求在补充规格说明中描述，比如顾客要求显示商品描述和价格。扩展由两部分组成：条件和处理。尽可能用系统内给或参与者能够检测到的事物作为条件，例如，系统检测到与外部税务计算系统服务的通讯故障和外部税务计算系统工作不正常，前一种比较好，因为那个是系统能够检测的条件，而后一种只是推断。

特殊需求：相关的非功能性需求；将所有非功能性需求集中于补充规范规格说明中，对于内容管理、可理解性和可读性而言更为有效，因为在架构分析时通常将这些需求作为整体来考虑。

技术和数据变元表：不同的I/O方法和数据格式，在需求分析中发现的一些技术变元，这些变元是关于必须如何实现系统的，而不是实现系统的哪些功能。

发生频率：影响对实现的调查、测试和时间安排；

杂项：例如未决问题；

迭代1的需求和重点：OOA/D技术的核心

强调的是基础繁为和在构件对象系统中所使用的常见OOA/D技术，首先处理困难和具有风险的事项。

e.g 销售系统

实现处理销售用例中基本和关键的场景：输入商品项目并收取现金。

实现用于支持迭代初始化需要的启动用例。

不处理任何特殊和复杂的部分，仅仅针对场景的简单理想路径，并对此进行设计和实现。

不与外部服务进行协作，例如，税金计算器或产品数据库。

不应用复杂的定价规则。

在迭代开发中，我们并非一次就实现所有的需求，在多个迭代里对同一用例进行增量式开发。

细化之所在

初始阶段强调面向需求的制品。

对核心、有风险的架构进行编程和测试。

发现并稳定需求的主题部分。

规避主要风险。

迭代时间约2到6周。

细化不是设计阶段，也不是要完成所有模型的开发，不是要创建可以丢弃的模型，与之相反，该阶段产生的代码和设计是具有产品品质的最终系统的一部分，该原型叫做可执行架构或架构基线。

一句话概括：构件核心架构，解决高风险元素，定义大部分需求，以及预计总体进度和资源。

细化阶段一些关键思想和最佳实践

实行短时间定量、风险驱动的迭代。

及早开始编程。

对架构的核心和风险部分进行适应性的设计、实现和测试。

尽早、频繁、实际地测试。

基于来自测试、用户、开发者的反馈进行调整。

通过一系列讨论会，详细编写大部分用例和其他需求，每个细化迭代举行一次。

通过风险（技术复杂性、工作量等）、覆盖范围（涉及系统所有主要部分）和关键程度（客户认为具有高业务价值的功能）组织需求和迭代，因此在迭代1之前完成工作等级划分，但是在迭代2之前要重新划分，因为新需求和新理解会对等级排列产生影响，这就是迭代的可适应性。

细化阶段会构建的制品

领域模型：领域概念的可视化，类似于领域实体的静态信息模型；

设计模型：描述逻辑设计的一组图，包括软件类图、对象交互图、包图等，是对重要设计思想及其在系统中动机的概要；

数据模型：包括数据库方案，以及在对象和非对象表示之间映射的策略；

用例示意板，用户界面原型：描述用户界面、导航路径、可用性模型等；

何时知道自己并没有理解细化阶段

对于大部分项目，细化阶段都比几个月更长。

只有一次迭代。

在细化开始前就定义了大部分的需求。

没有处理具有风险的元素和核心架构。

没有产生可执行架构；没有进行产品代码的编程。

认为细化阶段主要是需求或设计阶段，为构造阶段的实现进行准备。

试图在编程之前进行彻底和细致的设计。

只有少量的反馈和调整；用户没有持续地参与评估和反馈。

没有尽早和实际的测试。

在编程之前推测性地结束架构设计。

Pos系统交互图 ：

