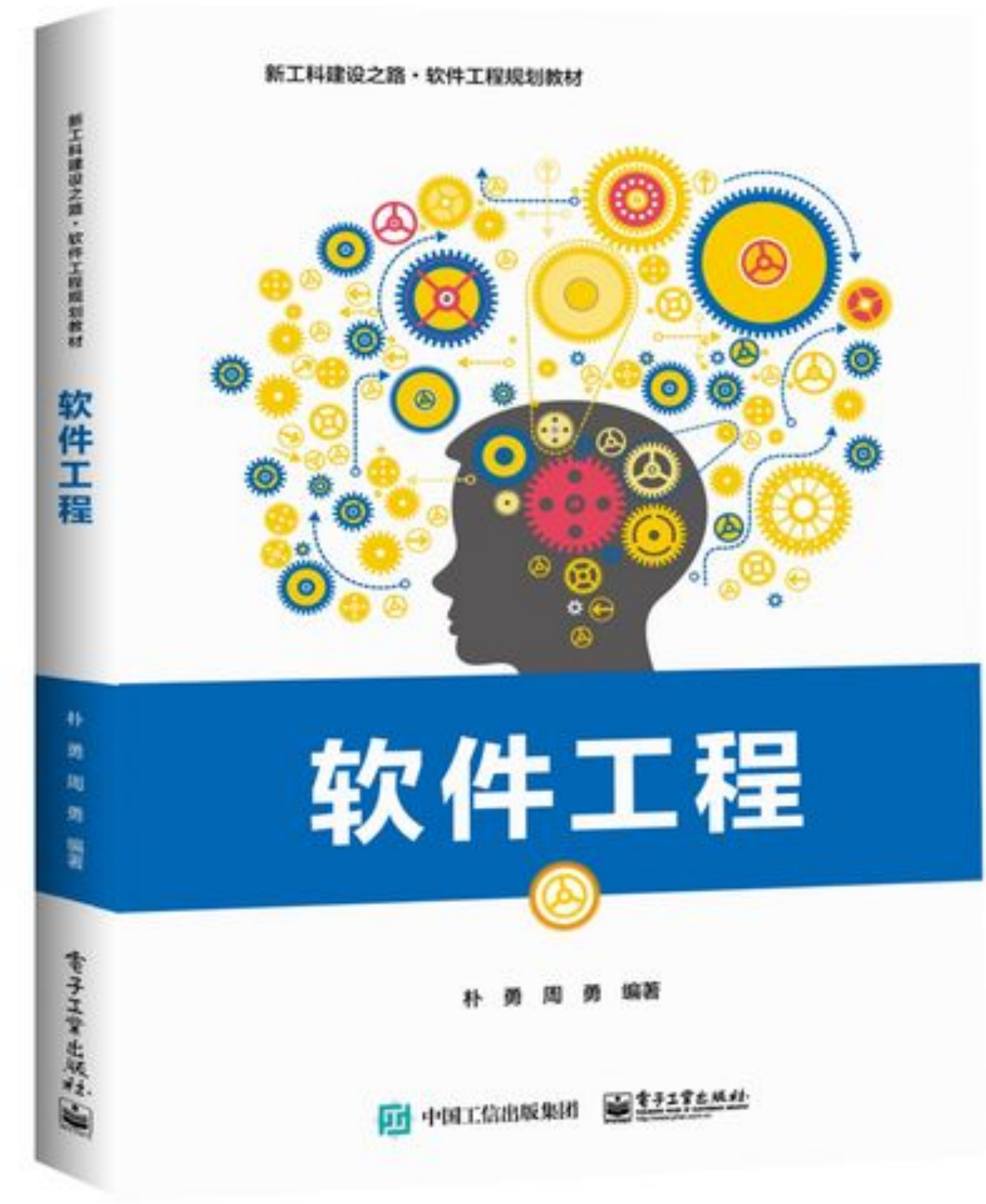
软件工程：知识点整理



第一章：软件工程概述

1.1 软件危机与软件工程

软件的特征

序号	特征
1	软件是逻辑层面上的，不是有形的物理文件，与硬件具有完全不同的特征
2	软件在使用过程中不会磨损，但会退化
3	软件开发早起是一门艺术，但目前越来越趋于标准化
4	软件同时也是一种逻辑实体，具有抽象性
5	软件会变得越来越复杂

软件的组成

组成	描述
程序	能够运行的、能提供所希望的功能和性能的指令集
数据	支持程序运行的数据
文档	描述程序研制的过程、方法及使用的记录

解决软件危机的方法

序号	方法
1	要对软件有正确的认识
2	推广使用软件开发成功的技术和方法，研究探索更有效的技术和方法
3	开发和使用更好的软件工具
4	对于时间、人员、资源等，需要引入更加合理的管理措施

软件工程定义

将系统化、规范化、可量化的工程原则和方法应用于软件的开发、运行、维护及对其中方法的理论研究。其主要目标是高效开发高质量的软件，降低开发成本。

软件工程知识体系

开发过程	支持过程
软件需求	软件配置管理
软件设计	软件工程管理
软件构造	软件工程过程
软件测试	软件工程工具与方法
软件维护	软件量度

软件工程基本原理

序号	方法
1	用分阶段的生命周期计划严格管理
2	坚持进行阶段评审
3	实行严格的产品控制
4	采用现代程序设计技术
5	结果应能清楚地审查
6	开发小组的人员应该少而精
7	承认不断改进软件工程实践的必要性

1.2 系统工程与UML

系统工程定义

为了更好的达到系统目标，对系统的构成要素、组织结构、信息流动和控制机构等进行分析与设计的技术。

系统工程性质

1. 系统工程用定量和定性相结合的系统思想和方法处理大型复杂系统的问题。
2. 系统分析的常用方法是层次分析方法，它将问题分解为不同的组成因素，并按照因素间的相互关联影响及隶属关系将因素按不同层次聚集组合，形成一个多层次的分析结构模型。

产品工程的层次结构模型

层次	表示	描述
最高层	领域目标	层次分析到达的总目标
中间层	中间环节	采取某一方案来实现预订总目标涉及的中间环节
最底层	要选用的解决问题的各种措施、策略、方案	

模型优点：更好的体现出人们对系统的理解和驾驭能力

系统工程建模特点

系统工程本质上是层次化的。对应系统工程的不同层次，相应的模型会被创建和使用，如需求模型中的模型要能体现出对系统宏观上的理解，下层模型要能明确具体子系统的需求。

统一建模语言UML

1. UML (Unified Model Language)是继20世纪80年代末，90年代初面向对象分析与设计方法（OOA&D）而出现的。
2. UML统一了Booch、OMT、OOSE方法，并在此基础上进行了标准化，如今已经成为对象管理组织（OMG）的标准之一。
3. UML是一种语言或工具，不是一种方法。它致力于分析设计的描述，其表述形式以图形方式为主，其描述形式本质上仍归为非正式的方式。

UML的产生——“三朋友”

姓名	贡献
Grady Booch	面向对象方法最早倡导者之一，提出Booch方法
James Rumbaugh	使用对象建模技术（OMT），适用于以数据为中心的信息系统
Ivar Jacobson	提出用例（Use Case），进而提出OOSE方法，以用例为中心，进行系统需求的获取、分析及高层设计

方法的战争

由于采用不同的符号进行类与对象的表示及关联，相似符号在不同模型中表示的意义不尽相同。

UML功能

UML将语言表示与过程进行分离。

功能	说明
可视化	帮助开发者理解和解决问题，方便沟通交流，发现设计草图中
规格说明	通过一种通用的、精确地、没有歧义的通信机制进行
构造	使用软件工具对模型进行解释和说明，将模型映射到某种计算机语言上来实现，加快系统建模和实现速度
文档化	使用UML同时生成系统设计文档

UML特性

1. 提供给用户高层建模方法，针对用户需求的高层抽象，具体表现为描述组建的构成及联系；这与自然语言描述不同的是，自然语言具有歧义和含糊不清的缺点。
2. UML的定义是通过**元模型**描述的。

UML 2.0使用模型

模型	说明
用例图	表示系统与使用者之间的交互，有助于将需求映射到系统
活动图	表示系统中顺序和并行的活动
类图	表示类、接口及其之间的关系
对象图	表示类途中定义的类的对象实例，配置是对系统的模拟
顺序图	表示对象之间的互动顺序
通信图	表示对象交互的方法和需要支持交互的连接
时序图	表示重点对象之间的交互时间安排
交互概况图	表示将顺序图、时序图、通信图收集到一起以捕捉系统中发生的重要交互情况
组成结构图	表示类或组件的内部，描述类间的关系
组件图	表示系统内的重要组件和彼此间交互所用的接口
包图	表示类与组件集合的分级组织
状态图	表示整个生命周期中对象的状态和可以改变状态的事件
部署图	表示系统最终怎样被部署到真是的世界中

1.3 系统开发的解空间

系统开发的解空间的组成

现代的面向对象分析和设计方法是基于模型的，综合使用用例建模、静态建模、动态建模和架构建模来描述软件需求、分析和设计模型，构成了系统开发的解空间。

模型	说明
用例建模	系统的功能型需求按照用例和参与者进行定义
静态建模	提供了系统的结构化视图
动态建模	提供了系统的行为视图，对象交互用于表示对象之间的通信和协作
架构建模	系统的核心，使用包、子系统和构建结构来描述系统的框架结构及框架中各个部分的连接关系

基于UML的软件建模和架构设计方法

1. 用例作为整个分析设计的驱动
2. 基于软件架构的理论和方法构造出软件的总体架构
3. 使用面向对象的静态和动态建模方法，完成以类为单元封装体和以构建为符合封装体的分析设计
4. 生成代码，实现系统

整个过程是一种基于用例的高度迭代的软件开发过程

过程	说明
用例建模	每个用例要开发一个叙述性描述的用例规约
分析设计	迭代式的进行系统的静态建模和动态建模
架构建模	设计系统的软件体系结构
实现	将软件架构模型映射到可部署运行实现的模型解空间中

1.4 软件工程开发方法

技术及规范

内容	说明
方法	完成软件开发各项任务的技术
工具	为方法的运用提供自动或半自动的软件支持环境，回答“用什么做”的问题
过程	获得一些列任务的框架，规定完成各项任务的步骤，回答“如何控制、协调、保证质量”

传统方法

内容	说明
描述	静态的思想，将软件开发过程划分为若干个阶段，规定各个阶段必须完成的任务，各阶段之间具有某种顺序性。
思维	分治思想
缺点	缺少灵活性，不适用于软件规模比较大尤其是开发的早期需求比较模糊或经常变化。

面向对象方法

内容	说明
描述	动态的思想，尽可能模拟人类思维方式；对象中同时封装了实体的静态属性和动态方法
特点	概念层与逻辑层相互协调，强调各种逻辑关系在结构上的稳定性
开发过程	主动多次迭代
思维	由一般到特殊，由特殊到一般，对象独立可重用

面向对象方法层次关系

层次	说明
类与对象	类是对象的抽象
类与类	可以构成“继承”的层级关系
对象与对象	消息机制确保对信息的“封装”

对传统和面向对象方法的理解

1. 传统方法又被称为”面向数据流“的方法；传统方法的分析图被称为”数据流图“
2. 面向对象对实体进行抽象，简化了分析的难度并增加了可理解性，同时保证了分析模型结构的稳定性，提高了应对变化的能力
3. 类的封装可以直接把修改限定在一个特定的范围内
4. “隔离变化、应对变化、以不变应万变”正是面向对象优势的体现

第二章：软件开发过程

软件开发过程，又称软件开发生命周期，是软件产品开发的任务框架和规范。

2.1 软件生命周期模型与软件过程

4种生命周期模型

1. 顺序式
2. 迭代式
3. 增量式
4. 敏捷式

3个主要时期：

1. 软件定义
2. 软件开发
3. 软件维护

6个阶段：

阶段	说明
可行性分析与安全计划	用最小的代价在尽可能短的时间内确定该软件项目是否能够开发，是否值得开发，最后给决策者提供做与不做的依据
需求分析	对目标软件未来需要完成的功能进行的详细分析，输出一份“需求规格说明书”
软件设计	寻求系统求解的框架，如系统的架构设计、数据设计
编码	将软件设计的结果翻译成某种计算机语言可实现的程序代码
软件测试	分为单元测试、集成测试及系统测试三个阶段，软件测试方法又分为黑盒和白盒
软件维护	为软件能持续适应用户的要求延续软件使用寿命的活动

传统生命周期模型和敏捷生命周期模型

传统生命周期模型	敏捷生命周期模型
顺序式瀑布模型	极限编程
快速原型模型	Scrum
增量式增量模型	
迭代式螺旋模型和喷泉模型	

2.2 传统生命周期模型

瀑布模型

一种计划驱动的模型，比较适合规模较大的系统开发或者分布式的开发模式

特点	问题
文档驱动，静态开发	模型的能力天生具有缺陷，尤其是需求模糊或不准确的系统
推迟实现	被动的救火式的应对问题，不希望有变化
质量保证	模型缺少灵活性，变更不容易

快速原型模型

一种适合于全新系统开发，可以借助原型使用户了解到开发的方向是否正确的开发模式，适合于工期比较紧张的项目。

特点	问题
快速构造软件模型	所运用的开发技术和工具不一定是实际项目所需要的
可以尝试运用未来系统中需要的技术	快速建立的模型质量不能保证

增量模型

又称演化模型，软件被视为一系列的增量构建来设计、实现、集成、测试。第一个增量往往只实现基本需求的核心产品。

特点	问题
各阶段交付满足客户需求的可运行产品子集	软件需要具备开放式的体系结构
较好适应变化	容易退化为“边做边改”模型使软件过程控制失去整体性

螺旋模型

一种将瀑布模型和快速原型模型结合起来，强调了其他模型忽视的风险分析，特别适合大型复杂的系统开发的开发模式。

活动	描述
制定计划	确定软件目标，选择实施方案，设定约束条件
风险分析	评估所选方案，识别并排除风险
实施工程	开发、验证下一级产品
客户评估	评估开发工作计划下一阶段

问题	描述
风险分析	要求客户接收和详细风险分析并做出反应是不易的，这种模型往往适用于内部的大规模软件开发
利润受损	风险分析如果大大影响项目的利润，则风险分析毫无意义
寻找风险	开发者应该擅长寻找可能的风险，准确的分析风险，否则将会带来更大的风险

喷泉模型

又称为“面向对象的生命周期模型”生存期的各个阶段可以相互重叠和多次反复，而且在项目的整个生存期中还可以潜入子生存期。

特点	描述
符合习惯思维和表达方式	直接针对问题域中存在的各项事物设立模型中的对象
不存在传统方法与设计之间的鸿沟，能够做到衔接紧密	从分析到设计不存在转换，只有局部的补充和优化，并增加与实现有关的独立部分
面向对象继承和测试也保持了连续性	面向对象开发阶段间的无缝特性和表示方法的一致性，是喷泉模型优于传统软件工程方法的重要因素

2.3 敏捷生命周期模型

敏捷生命周期模型适合集中式的开发模式

原则

1. 个体和互动胜过流程和工具
2. 工作的软件胜过详尽的文档
3. 客户合作胜过合同谈判
4. 相应变化胜过遵循计划

两个基本特点

1. 增量交付
2. 迭代开发

开发建议

先实现必要的用户案例，体现出软件的价值，然后在后续版本中对功能进行细化。

用例定义

用户通过系统完成的有价值的目标。在敏捷生命周期模型中，用例通常作为迭代的单位，这样每次交付的都是可以部署到用户应用环境中被用户使用的、能给用户带来即时效益和价值的产品。

迭代的思想

当我们对用户需求没有信心时，可以先构建后修改，通过多次反复，站到客户真正需要的软件。

敏捷生命周期模型优势

特点	描述
精确	产品由开发团队和用户反馈共同推动
质量	对每一次迭代周期的质量都有严格要求
速度	提倡避免较大的前期规划，新的功能或需求变化尽可能频繁的整合到产品中
丰厚的投资回报率	最具价值的功能优先开发
高效的自我管理团队	拥有一个积极的、自我管理的、具备自由交流风格的开发团队

极限编程

主要目的是降低需求变化的成本，引入一系列优秀的软件开发方法作为开发实践的指导，并将它们发挥到极致。

原则	描述
互动交流	团队成员之间通过日常沟通、简单设计、测试、系统隐喻及代码本身来沟通产品需求和系统设计
反馈	客户的实际使用、功能测试、单元测试等，都能为开发团队提供反馈信息
简单	XP总是从一个简单地系统入手，并且只创建今天需要的功能模块
勇气	鼓励一些应对较高风险的良好做法
团队	激励团队，把相互尊重和实际的开发习惯结合

Scrum

更注重软件开发的系统化过程，它由一个开发过程、集中角色及一套规范的实施方法组成。

冲刺周期	说明
计划会议	最重要或者最具价值的产品需求积压被优先安排到下一个冲刺周期
每日会议	每个团队成员汇报各自的进展情况，同时提出目前遇到的各种障碍
评审会议	开发团队将会向客户或最终用户展示新的系统功能，用户会提出意见和一些需求变化
回顾会议	总结上次冲刺周期中有哪些不足需要改进，有哪些值得肯定的方面

角色	说明	职责
产品拥有者	负责产品的远景规划，平衡所有利益相关者，确定不同的产品需求积压的优先级。	他们是开发团队和客户或最终用户之间的联络点
涉众	该角色与产品之间有直接或间接的利益关系，通常是客户或最终用户代表。	他们负责收集、编写产品需求，审查项目成果。
专家	指导开发团队进行Scrum开发与实践	开发团队与产品拥有者之间交流的联络点
团队成员	项目开发者	

DevOps（*Development and Operations*）

DevOps实际上是一组过程、方法与系统的统称。

目标	说明
1	促进开发、技术运营和质量保证部门之间的沟通、写作与整合。
2	自动化和可持续交付

第三章 需求分析

3.1 需求分析活动

需求分析阶段一般通过一份需求规格说明书的文档描述项目应该实现的内容。

可行性分析

明确系统的功能性需求、边界位置，以及在技术、经济、法律或操作等方面项目能否顺利进行，是否存在潜在的风险，这些风险的影响程度和降低这些风险的措施，并给出一个合理可行的解决方案。根据可行性分析的结果还可以评估出系统初步的开发费用。

用户需求和系统需求

项目	用户需求	系统需求
完善程度	一般	完善和细化
阅读对象	委托方或客户	开发者
相互关系	得到委托方确认	用户需求的开始

系统涉众

类型	说明
最终用户	他们是系统的实际使用者，对目标系统有直接的接触和评价
投资者	主要关心的是目标系统的总成本、建设周期及未来的收益等高层目标
业务提出者	使现有的业务能够更加规范和高效，提升业务质量。
业务管理者	负责业务计划、生产、监督等环节的实际实施和控制
业务执行者	实际的操作人员，是频繁与系统直接交互的人员
第三方	与业务关联的，但并非也无妨的其他人或事
开发方	合同乙方的利益代表，他们关心的是这个项目是否有经济利益，是否能积累核心竞争力，是否能树立品牌，是否能开拓市场。
法律法规	既指国家和地方法律法规，又指行业规范和标准

系统目标

项目	内容
目标	设置该目标可以满足的期望
对涉众的影响	对涉众及其影响
边界条件	附加条件或约束
依赖	是否依赖其他目标，与其他目标关系如何，是相辅相成还是此消彼长
其他	其他需要说明的内容

3.2 用例与系统功能

功能识别

功能识别的方法取决于项目的类型和涉众的计算机经验和技能。

方法	说明
座谈	通过做谈的方式，共同确定未来软件支持的业务工作流程
访谈	访谈关键的涉众人员，因为未来的系统最终会有这些人员组织验收
调查问卷	题目要有一定代表性
问题讨论	挖掘优化潜力
头脑风暴	是开发者逐渐融入开发过程，较为容易的理解业务过程实现原理

用例概念

人们习惯使用一种交互的方式来描述系统的场景，介意“捕获“用户的需求。用力强调的是参与者与系统的交互活动。

用例开发

用例过程	说明
完善	迭代进行
迭代	添加业务细节
规约	对具体用例场景中业务流程的脚本式的说明

用例图

图形	说明	表述
椭圆	用例	代表一项用户与系统的交互
人形符号	使用系统的用户类别	不特指具体的人，按照角色理解

角色

从角色出发来识别用例。

角色	说明
1	现实业务中的人
2	存在与系统之外的其他软件系统

寻找用例切入点

寻找用例要围绕客户的预期，充分体现用户需求，其切入点如下：

切入点	说明
关键业务实体	通常这是与业务相关的一些关键概念或技术术语
确定业务实体是由被创建的软件管理还是由已经存在的其他系统管理	若由其他系统管理，则外部系统可识别一种角色：创建-修改-删除-持久化-访问
相同的专业术语指代同一事物	用例名具体贴切，尊重用户习惯和选择
围绕涉及业务过程数据的动态信息	这些信息是基本业务实体的组合和加工
考虑系统的功能性需求	一般是在识别出的基本业务实体数据和动态过程数据的基础上做进一步的计算和利用
考虑是否存在与正在运行的其他系统的交互	在分布式的环境中，与其他系统的交互，都要设置单独的用例补货其功能性需求

业务用例和系统用例

系统用例	业务用例
开发者视角	客户视角
软件过程	业务过程
对应业务用例在软件系统中的实现	开发者以何种角色参与到具体项目中

用例规约

用例规月是对每个用例的细化。模板如下：

项目	编号	内容
	迭代的编号（在哪一轮完善此信息）	

数据字典

在对用例进行文档化规约的同时，还应该建立一个词汇表即数据字典来描述相关业务术语的定义。

构造型

种类	说明
<>	连接主用例那些为其他主用例提供某种共同的基础性功能的过程
<>	表示次用例是在细化的分析阶段被识别出来的
<>	两个用例之间的拓展关系，隶属于有条件发生的用例

3.3 过程建模与事件流

活动图

使用活动图进行过程建模。

图形	定义	说明
带圆角的矩形	动作	动作的执行顺序由动作间的箭头方向进行指示
黑色实心圆圈	活动图的开始	
带外环的黑色实心圆圈	活动图的结束	
菱形	控制点	形式1：多个箭头从控制点引出，表示分支的选择，最多只能满足一项
		形式2：多个分支的汇入，不同的条件分支的汇聚节点
【本项可选】矩形框	表示数据的对象	对象流一般用来强调上一个动作的输出结果或者下一个动作需要的输入

汇聚类型	说明
{and}	等待所有分支结束后才可以继续进行
{or}	有一个流程分支结束即可

过程模型构建

过程模型一般采用增量的方式进行构建。

1. 描述出典型的流程，如果所有人员对过程的描述达成了一致，那么再逐渐加入其他可能的分支流程。
2. 每个步骤增量完成，逐步加入可选动作
3. 所有内容充实后，考虑对某些部分进一步提炼为单一动作，进而对模型进行简化和优化

泳道

泳道是一种角色的划分方法，每个角色的活动放置在对应的泳道中，实现将模型中的活动按照职责组织起来。

事件流

1. 利用活动图进行的过程建模可以应用于用例中的事件流描述。
2. 对于用例中基本流的每个步骤，可生成一个单独的动作。
3. 如果基本流的每个步骤的动作比较复杂，可以对应使用几个动作来对它进行表示，或者使用另外一个单独的活动图对其进行细化。
4. 最后考虑此用例的备选流，将备选流说明中对应的动作逐渐补充进该活动图。

3.4 功能性需求

困难

困难	说明
隐含的假设	当领域专家向开发者说明他们的想法时，一般会将他们认为是理所当然的信息漏掉，谈话也会变得更加简洁
笼统的注释	如果业务概念确实或者过于笼统，就会给开发者带来迷惑
模糊的概括	概括是指在基本流中经常使用“总是”或"从不"等描述形式。分析人员应该在写需求的同时质疑这些描述形式的表达
迷惑的命名	在对业务活动进行描述时，不可避免的要对动词进行选择。动词作为名词使用时，要特别注意其表达的含义是否准确

文字需求的模板

通过“必须”“应该”“将会”控制右侧需求陈述的组合。

形式	说明
必须	代表用户直接的愿望，是十分重要的
应该	从用户角度出发，可能是有意义的，但是否需要最终进行实现还可继续讨论
将会	在某些已经实现的功能上的拓展功能，需要考虑他们的可行性

需求类型

类型	说明
系统功能	系统自身应完成的功能需求
交互	系统提供给特定用户交互性的功能
外部接口	系统为第三方系统提供的外部访问接口或者需要通过外部系统的接口获取数据的功能

对需求的统一编号并赋予简短的标题

编号	标题	子标题
R1.1	项目创建	项目信息
		项目
		子项目
R1.2	数据存储	
R1.3	项目选择	
R1.4	子项目创建	
R1.5	子项目与项目	
R1.6	项目信息编辑	
R1.7	项目任务添加	项目任务
		完成进度
		工作量
R1.8	项目任务选择	
R1.9	项目任务编辑	
R1.10	对其他项目的依赖	
R1.11	工作量改动的验证	
R1.12	工作量检查	
R1.13	工作量改动失败提醒	
R1.14	与外部系统的协同	

由活动图派生的文本形式的需求，需要将每个动作都在一个或多个需求陈述中体现，而且每个动作的转移和判断分支，都至少要包含在一个功能内，或者作为约束条件在需求中存在，不要遗漏。

3.5 非功能性需求

非功能性需求的种类

1. 质量需求
2. 技术性需求
3. 其他交付物
4. 合同需求
5. 规格说明

质量需求

质量需求是针对目标软件的质量特征进行说明的。某个单一的质量需求可能会对整个项目的成败起决定性的作用。一个简短的质量需求后面可能存在着大量的隐性需求。

1. 正确性
2. 安全性
3. 可靠性
4. 健壮性
5. 可用性
6. 存储和运行效率
7. 可维护性
8. 可移植性
9. 可验证性
10. 易用性

对具体的需求进行描述时注意，对给出的需求陈述应能满足，并且程度是可以度量的，为此要尽可能准确和详细的给出可能采取的质量验证措施。

技术性需求

直接针对软件项目相关的技术及环境等方面的需求，最典型的是硬件需求。功能性需求中主要阐述与协同软件的写作功能，而技术性需求主要包含对协同软件的约束。

技术性需求还可以描述为对开发环境或工具的要求。

功能需求	非功能性需求
与协同软件协作的功能	对协同软件的约束（版本、分布式架构、连接类型、网络传输速度）

其他交付物

功能需求	非功能性需求
最终的安装包	硬件、使用说明书、安装说明书、依赖库、开发文档及培训

合同需求

功能需求	非功能性需求
合同	作为合同的形式规范下来，或者作为补充的合同条款（付款方式、交付期限、计划的项目会议、惩罚和诉讼方式）

规格说明

所有的需求最终都要在需求文档中进行汇集，这份文档通常称为需求规格说明书。

甲方	乙方
委托方	开发方
说明书中明确对未来产品的期望	说明书中明确将要完成的需求任务

需求规格说明书结构	解释
文档说明性内容	文档的版本号、创建者、修改记录、批准者
目标群体和系统目标	利益相关者分析、项目目标、预算成本、开发周期
功能性需求	用户故事到业务场景分析，再到文字的功能性陈述
非功能性需求	注重质量需求及重要的技术需求
其他交付物	需要交付的软件制品清单和时限
验收标准	验收的方式及标准说明
附件	相关文档的清单

3.6 需求跟踪

基本原理

给出每个原始需求项及其对应的目标副产品，并将它们通过某种方式联系起来，由此可以确定出原需求的实现情况。

跟踪关系图

能够马上识别出一个需求有没有原内容与其对应；可以很容易的回答在软件功能被删除或调整前，对其影响范围进行评估。

业务	需求	类模型

能力成熟度模型（CMM）

软件组织必须具备需求跟踪能力，“软件制品之间，维护一致性”。

需求跟踪矩阵（RTM）

验证需求是否得到实现的有效工具。

纵向跟踪矩阵	实例	横向跟踪矩阵	
需求之间的派生关系	客户需求到系统需求	需求之间的接口关系	
实现与验证的关系	需求到设计、需求到测试用例		
需求的责任分配关系	需求的负责人员		

第4章 软件架构的构建

4.1 软件架构及其定义

软件架构设计

建立计算机软件系统所需的数据结构和程序勾践，主要考虑软件系统所采用的体恤结构风格、系统组成构建件的结构和属性及系统中所有体系构建之间的相互关系。

下面来看一个关于软件的比喻

构件	比喻
基础	操作系统之上的基础设施软件
主体	实现计算逻辑的主题应用程序
装饰	方便实用的用户界面程序

软件架构研究

研究方向	描述
内容	软件架构描述、软件架构风格、软件架构评价、软件架构形式化方法
目的	解决软件重用、质量、维护问题

软件架构发展的四个阶段

发展阶段	描述
无体系结构设计阶段	已汇编语言进行小规模应用程序开发
萌芽阶段	程序结构设计出现，以控制流图和数据流图构成的软件结构为特征
初期阶段	从不同侧面描述系统的结构模型，UML为其代表
高级阶段	描述系统的高层抽象结构为中心，不关心具体的建模细节，划分了结构模型和传统软件结构的界限

软件架构的定义

1. 软件架构是具有一定形式的结构化元素，即构建的集合。包括处理构建、数据构建和连接构件。
2. 软件架构是软件设计过程中的一个层次，查阅计算过程中的算法和数据结构设计，并在其上处理整体系统结构设计和描述的一些问题。
3. 软件架构有四个角度：

角度	描述
概念	主要构件及之间关系
模块	功能分解和分层结构
运行	系统的动态结构
代码	各种代码和库函数在开发环境中的组织

4. 程序或计算机系统的软件架构包括一个或一组软件构建、软件构建的外部可见属性机器相互关系。

   通识定义：软件结构为软件系统提供了一个结构、行为和属性的高级抽象，由构成系统的元素的描述，这些元素的相互作用、指导元素集成的模式机器这些模式的约束组成。

软件架构模型

模型	描述
结构	直观、普遍。以体系结构的构建、连接件和其他概念刻画结构，力图通过结构反映系统的重要的语义内容。
框架	不太侧重描述结构的细节，更侧重整体结构。以一些特殊的问题为目标，建立只针对和适应该问题的结构
动态	对结构和框架模型的补充，研究”大颗粒“的行为性质，通常是激励性
功能	体系结构由一组功能构建按层次组成，下层想上层提供服务，可以视为一种特殊的框架模型
过程	研究构造系统的步骤和过程，因而其结构遵循某些过程脚本的形式

”4+1“视图

视图	描述	主要内容
逻辑	支持系统的功能性需求，即系统提供给最终用户的服务	功能描述+类模型
进程	侧重于系统的运行特性，主要关注非功能性需求，定义逻辑视图中的各个类的操作具体是咋哪一个线程中被执行的	处理流程+并行性+同步
物理	主要考虑如何把软件映射到硬件上，通常考虑系统西能、规模、可靠性等、解决系统拓扑结构、系统安装、通信等问题	目标硬件+网络
开发	侧重于软件模块的组织和管理	子系统+接口
场景	重要系统活动的抽象，联系以上四个视图。帮助开发者找到体系结构的构建和他们之间的作用关系。也可以开分析一个特定的视图，描述不同视图构建间的相互作用	

4.2 软件架构模型

元素	定义	分类
构建	具有某种功能的可重用的系统模板单元	复合构件、原子构件
连接件	构件之间的交互	管道、过程调用、事件广播、通信协议、SQL连接
配置	构建和连接件之间的拓扑逻辑和约束	
端口	NULL	
角色	NULL	

4.3 软件架构风格

核心问题

能否使用重复的体系结构模式，即能否达到体系结构级的软件重用。

软件架构风格

描述某一特定应用领域中系统组织方式的惯用模式。

软件架构内容

内容	说明
体系结构定义	领域中众多系统共有的结构和语义特性，如何将各个模块和子系统有效的组成一个完整的系统
词汇表	构件和连接件类型
一组约束	如何将这些构建和连接件组合

软件架构风格分类

1. 数据流风格
2. 调用/返回风格
3. 独立构件风格
4. 虚拟机风格
5. 数据中心风格

管道与过滤器定义

每个构件都有一组输入与输出，构件读取输入的数据流，经过内部处理，然后产生输出数据流。

管道与过滤器实例

1. UNIX shell：提供一种符号以连接各组成部分（Unix进程），有提供某种进程运行时的机制以实现管道。
2. 传统编译器：一个阶段的输出是另一个阶段的输入。

管道与过滤器优势

高内聚、低耦合

1. 适合批处理和非交互处理的系统
2. 良好的信息隐藏性
3. 良好的模块独立性

层次结构定义

层次结构组织成了一个层次结构，每一层为上层提供服务并作为其下层客户

风格特点

1. 支持可增加抽象层的设计，可以将一个复杂问题分解成一个增量步骤序列实现。
2. 便于功能的增强和拓展

风格应用

分层的网络通信协议。

仓库、黑板系统构件

1. 中央数据结构：说明当前状态
2. 独立构件：在中央数据存储上执行

数据库与黑板系统

数据库	黑板系统
输入流中某类时间触发进程执行的选择	中央数据结构的当前状态触发进程执行的选择

黑板系统

组成	内容
知识源	包含独立的、与程序相关的知识；之间不直接通信，只通过黑板通信
黑板数据结构	按照与应用程序相关的层次来组织、解决问题的数据，由知识源改变
控制	完全由黑板驱动的状态，黑板状态的改变决定的知识的改变

黑板系统应用

信号处理

MVC结构

MVC结构的全名是”Model-View-Controller“。将业务逻辑、数据、界面分离的方法组织代码，将业务逻辑聚集到一个部件里。

使用MVC的最初目的

使用MVC是将模型与视图分离，从而使同一个程序可以使用不同的表现形式。

视图

用户看到并与之交互的界面。

模型

企业数据和业务规则。模型与数据格式无关，能为多个视图提供数据。

控制器

接收用户的输入并调用模型和视图完成用户的需求。接收请求并决定调用哪个模型去处理请求，然后再确定调用哪个是土显示返回的数据。

第5章 类的分析与设计

5.1 基本类的确定

面向对象的分析与设计

需求阶段捕获到的结果转换为模型的表示，并基于这些模型进行设计的优化，作为后续代码的实现基础，需求规格说明书是进行面向对象的前提条件。

类

类的类型	应用	包含	来源
实体类	对应系统需求中的每个实体	包括存储和传递数据的类，又包括操作数据的类。	需求分析中的名词
控制类	体现应用程序的逻辑，提供相应的业务操作，将其抽象畜类可以降低交互层和存储层之间的耦合度		动宾结构的短语
边界类	对外部用户与系统之间的交互对象进行抽象	界面类及与外部系统数据的交换类	

初级阶段

1. 识别出实体类
2. 绘制初始类图（也被称为领域模型，主要包含实体类和他们之间的相互关系）

对象设计要求

1. 应具有惟一的表示OID
2. 用具有若干相关的属性
3. 同类对象的共同结构通过他们的类进行说明，所有的实例变量都可包含在类中。

类的识别

这是一个寻找和迭代的过程，开始时建立类的雏形，带有基本的实例变量，然后不断不重新的类和信息并逐渐扩展。

需求规格说明书寻找类

本质是按照语法分析将名词标注为对象的候选，将形容词标志为属性的候选。

数据字典寻找类

渔业武术与相关的类通常是实体类，代表的都是在系统中需要管理的实际的业务对象。

初始类图

把类的识别中发现的所有信息融入一个类图中加以表示，形成初始类图。

/：表示依赖属性，其取值是通过其他相关变量的值计算而来。可能需要进一步确定他们是否真的需要实现。对其处理有两种选择：

1. 由开发者维护数据的一致性
2. 在需要时实时计算

分析类图

在分析类图中，通常先省略变量的类型部分，暂时忽略这些实现的细节。UML中提供了针对实体类的构造型《entity》

类的方法

它们为业务计算或实例变量值的读写提供了服务。

对象中所有实例变量值的组合构成了该类的状态集合，所以通过这些方法可以对类的状态进行修改。

方法作用

提供对变量值的访问和修改。

迭代过程

过程	说明
初始	识别出与业务紧密相关的实体类，实体类一般不存在与其他类的交互，是系统处理业务的基础信息和载体
后续	识别出更多的关于实体类间的交互和作用方式

类的关系

关系	说明	方向	存储方式
关联	它们的对象之间的一种持有联系	可双向	类的实例变量
依赖	体现对象的瞬时使用关系，即一个雷需要另一个类的协助，但不需要进行保持	避免双向	方法的局部变量、方法的参数、静态方法调用

关联关系的类图

元素	说明	方向	角色
任务	无需显式给出	黑色三角形方向	任务类端点处标注角色

多重性

1. 由（*）表示任意多
2. 类似于ER图中的箭头，发出方可以是零或一个或多个，接收方只能是一个或零个。
3. 基数的表达方式如下所示：

基数表达	说明
*	任意多个
1	只有一个
3	正好三个
1..*	最少一个，可能多个
3..*	最少三个，可能多个
0..1	0或1个
3..7	3到7个

自反关联

指向自己本身的关联关系

实例理解：

一个Project对象可以具有多个前驱对象，同样一个Project对象可以作为多个Project对象的前驱。

类图创建实用方法

将每个类描绘在卡片上并挂在以免可擦写的磁性白板上，从而可以很容易的将其拿下或贴上。可以将不同版本的类图拍照，并以照片的形式进行保存。这种在卡片上绘制类的方法又称CRC，卡片上主要记录类的名字、包含的属性、主要职责、有消息交互的其他类等。

类与对象

	说明
类	一种结构
对象	类实例化后的结果，具有类中所有的成员变量，并对它们赋予了具体的值

UML对象指代

（对象名）：类名，（）代表可省略。

5.2 类的细化

管理类和控制类

管理类	控制类
对同类对象的协调和管理，主要负责同类对象的创建，代理访问其它对象的信息等。	对一个或多个用例所特有的控制行为进行建模。控制和协调不同对象的行为，用来封装用例的特有行为。
提供对其所管辖的所有对象统一的处理方式	有效的将边界对象和实体对象分开，将用例所特有的行为与实体对象分开。

控制类的识别方法

对所有的用例进行分析，每个用例对应产生一个控制类，用来对该场景中需要的对象进行管理和协调

控制类设置和细化建议

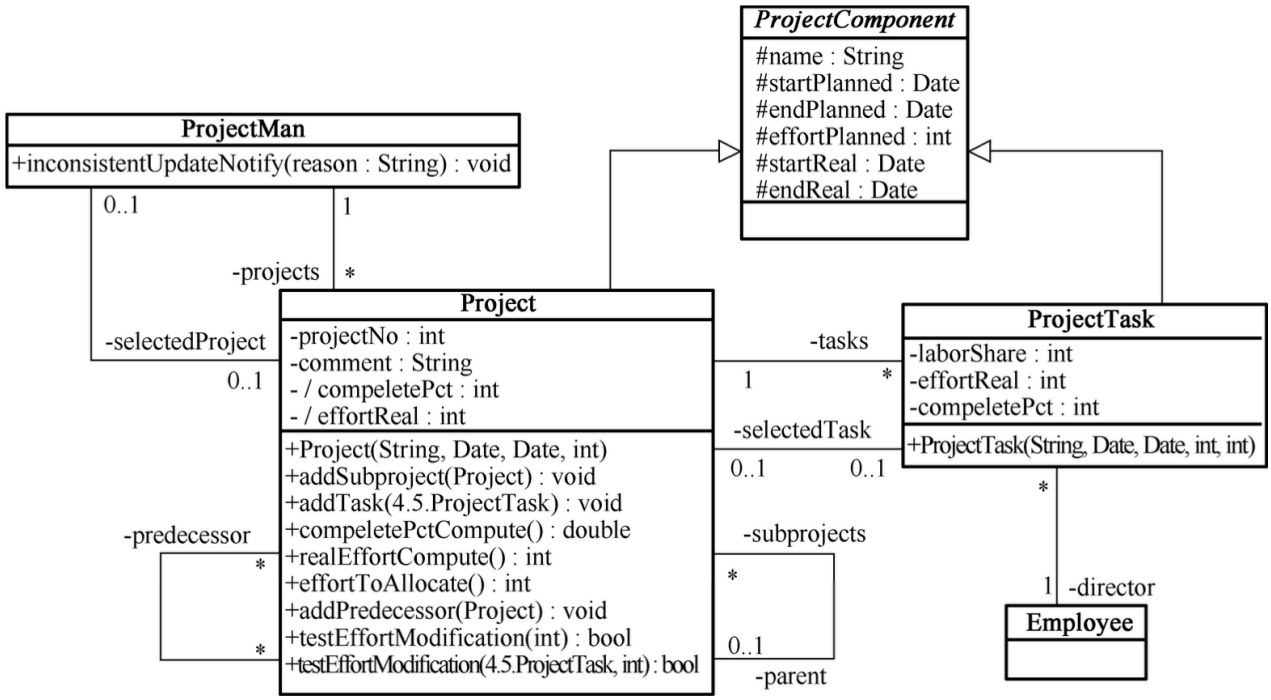
1. 每次只考虑一个任务，只向控制类添加与该类任务相关的方法和方法需要的实例变量。
2. 类与类之间尽可能保持较少的联系，降低接口数量

UML简化设计

基本上，任何类都可以将所有信息隐藏，只保留名字。在不同抽象级别上，类有不同的表示方式；随着开发的推进，类的细化信息也会逐渐的丰富起来。

设计优化规则

1. 删掉那些没有方法和实例变量的类
2. 对于复杂的类，考察其功能独立性，确认是否可以按照功能的不同对其划分，使得拆分后的类在功能上具有不同的侧重点



抽象类

	说明
表示	使用斜体字，不能从该类直接产生对应的实例对象
应用	可以作为变量的一种类型使用，实现松耦合

枚举类型

	说明
优势	可以使设计和代码简单易懂，同时提高取值的可控和安全性
应用	如果一个变量的取指是某个有限集合中的数据

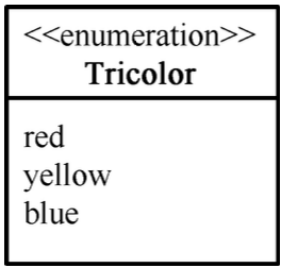


图 5.10　枚举类的表示

5.3 补充和确认

顺序图

构成	说明
上端	将场景中所涉及的对象横向罗列出来，构成对象的横轴
从上到下	时间的延续，构成时间的纵轴
虚线	生命线，其上可嵌入控制交点，用以表示对象当前处于激活状态，如等待某个结果返回
实心三角箭头	同步信息，通常带有一个结果的返回；同步信息表示对象在调用方法后处于等待状态，直到另一个对象结果返回。异步信息表示对象不用出与等待状态，描述上通过一个简单箭头进行区分

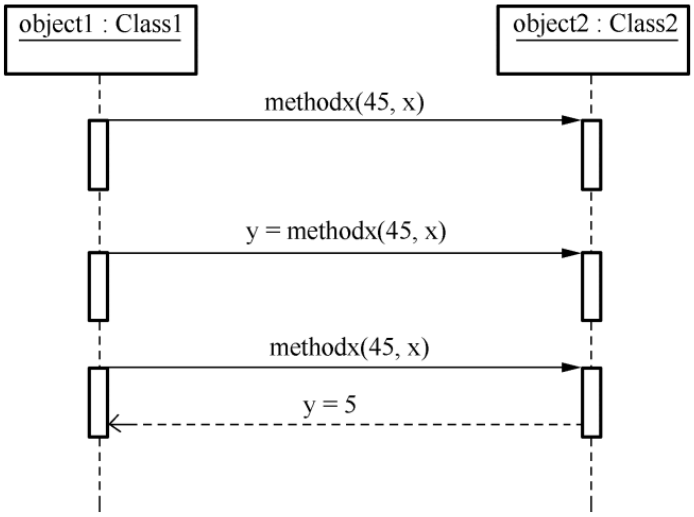


图 5.11　顺序图的基本构成

同步消息的描述方法

方法	说明
具有一个代表控制焦点的矩形框	对象处于激活执行状态

生命线

生命线表示对象实例的生存周期。

1. 通常在被销毁的的对象的销毁处用”X“，并且虚线不再向下延伸

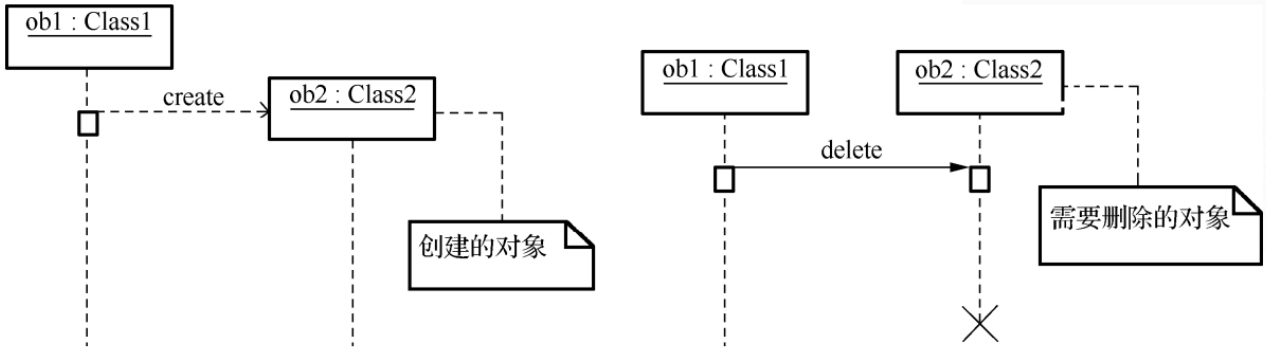


图 5.12　对象在顺序图中的创建和删除

通信图

1. 对象描述格式：通信图中的每个对象与顺序图中的描述格式一样，即矩形内带有可选的对象和类名的形式
2. 连接关系表示：对象之间通过实线连接，表示它们之间具有的连接关系，这其实是对应类的关联关系的实例
3. 交互顺序标识：在连接线的一侧描述了消息的交互信息，交互顺序通过消息前面的数字进行标识（如果在消息内部含有嵌套消息或者

   触发其他事件，那么这种嵌套的层次关系可用上层数字后的扩展数位来表示）

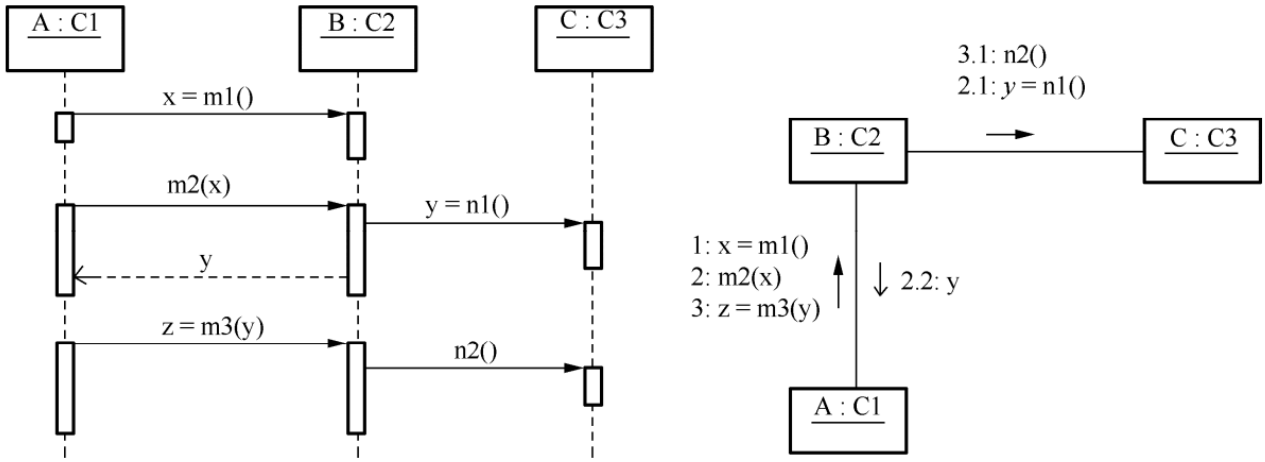


图 5.13　顺序图与等价的通信图

通信图优势

可以很好地用来描述对象的结构和它们之间的依赖关系，更强调交互对象的结构性

顺序图描述选择逻辑

这些逻辑结构通常使用一种矩形框进行表示，并在左上角给出逻辑结构的含义。

逻辑结构	说明
opt	表示在满足方括号内条件的情况下，对应部分会被执行，否则跳过
alt	对多分支的条件进行选择，在矩形框内各分支彼此之间使用虚线进行分割，每个分支都对应一个条件，而且彼此排斥，若条件都没有满足，则 else 部分会被执行
loop(start, end, condition)	对循环结构进行定义，这里必须清楚地给出循环执行的参数，如循环次数和结束条件

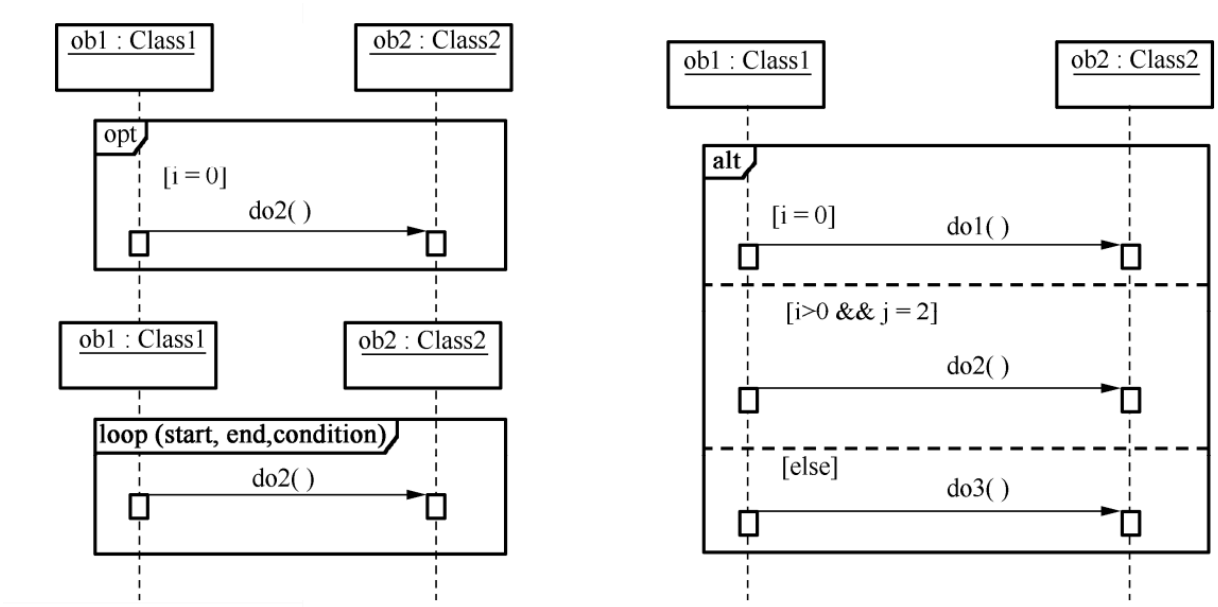


图 5.14　顺序图中的逻辑结构

场景模拟

使用顺序图对活动图(对应着用例规约)进行确认的方法是对活动图中的每个场景使用一个顺序图进行模拟，即对实际的业务流程进行场景模拟。

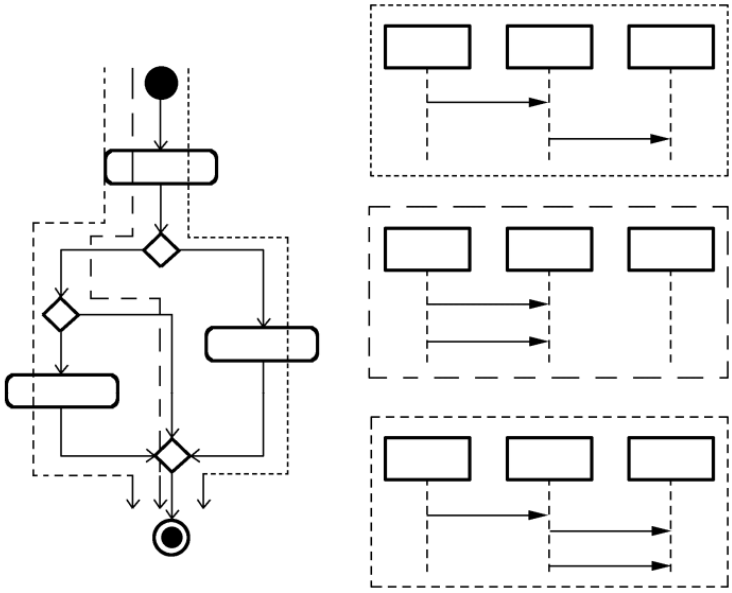
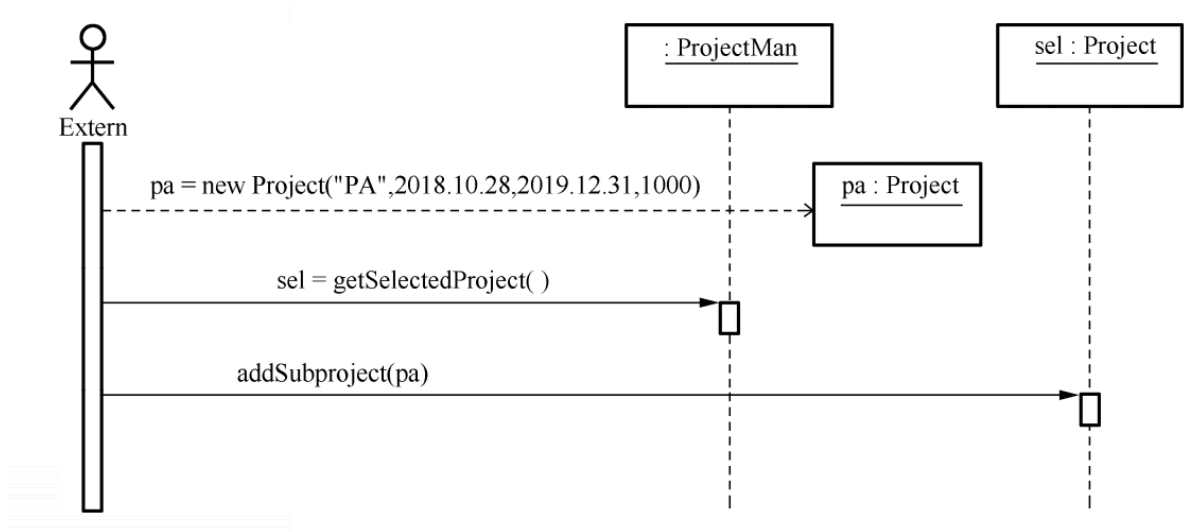


图 5.15　业务流程场景模拟

1. 活动图中描述了 3 个业务流程，分别使用实线、长虚线和短虚线进行了标识
2. 确认的目标是确保每条活动图中的边都被执行
3. 对于每个可能的执行流程，都寻找一个顺序图与之对应，使其具有相同的功能描述

外部角色

场景内会存在一些外部对象，它们与场景内的对象有交互作用，可能暂时无法获知它们的准确信息，但确有存在的必要，因为它们可能是业务流程的驱动者。



继承关系

若项目之间存在继承关系，则通过成员变量 parent 关联。

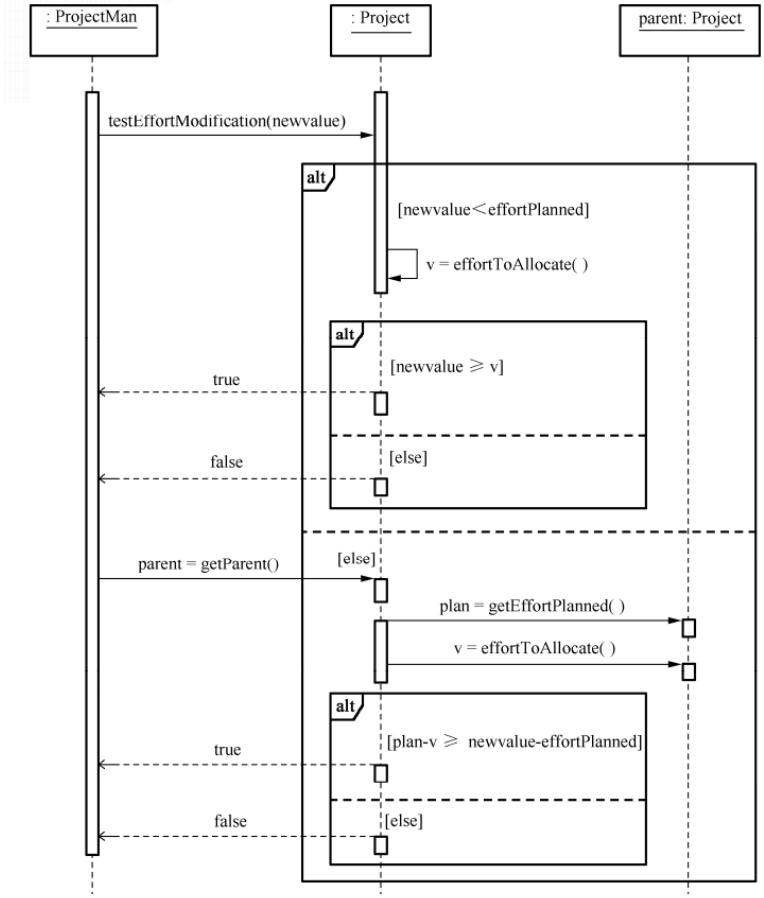
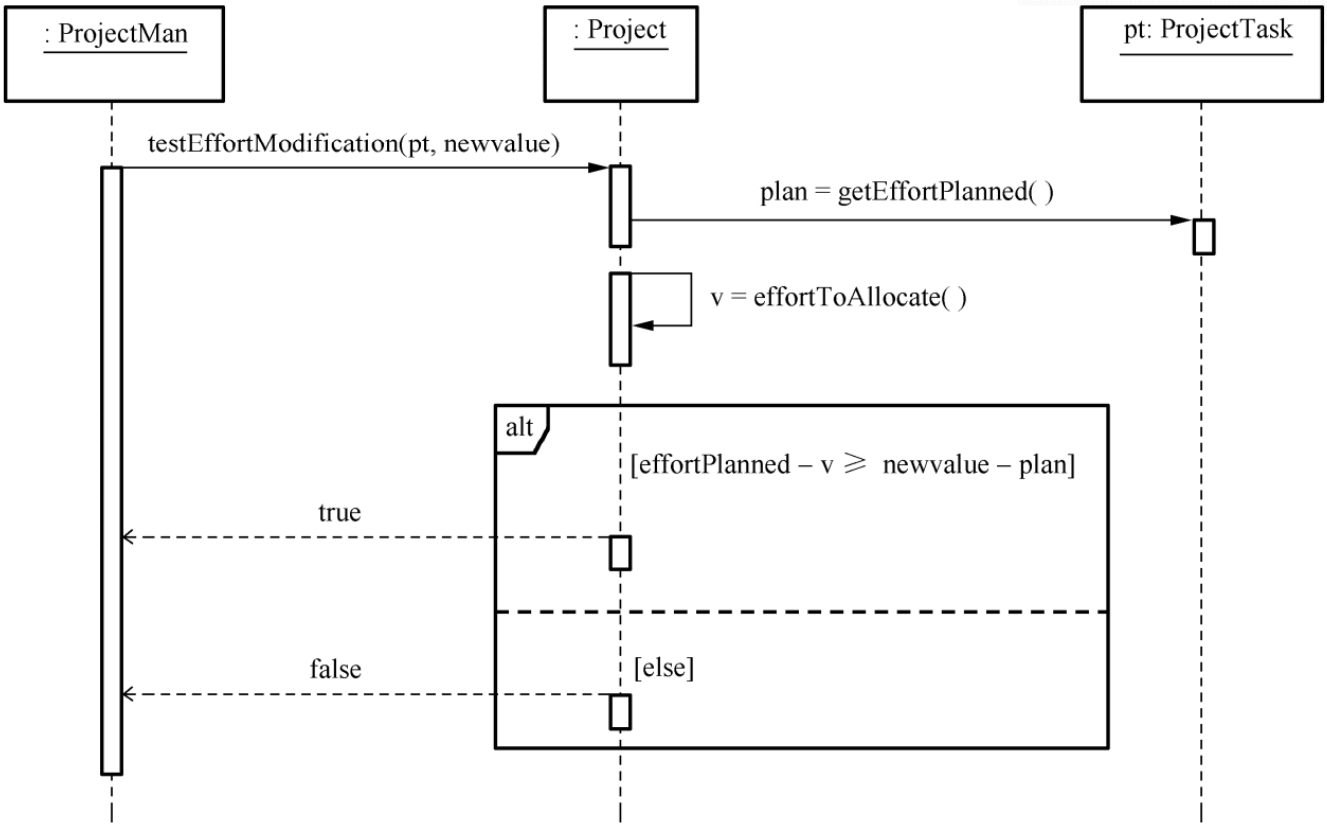
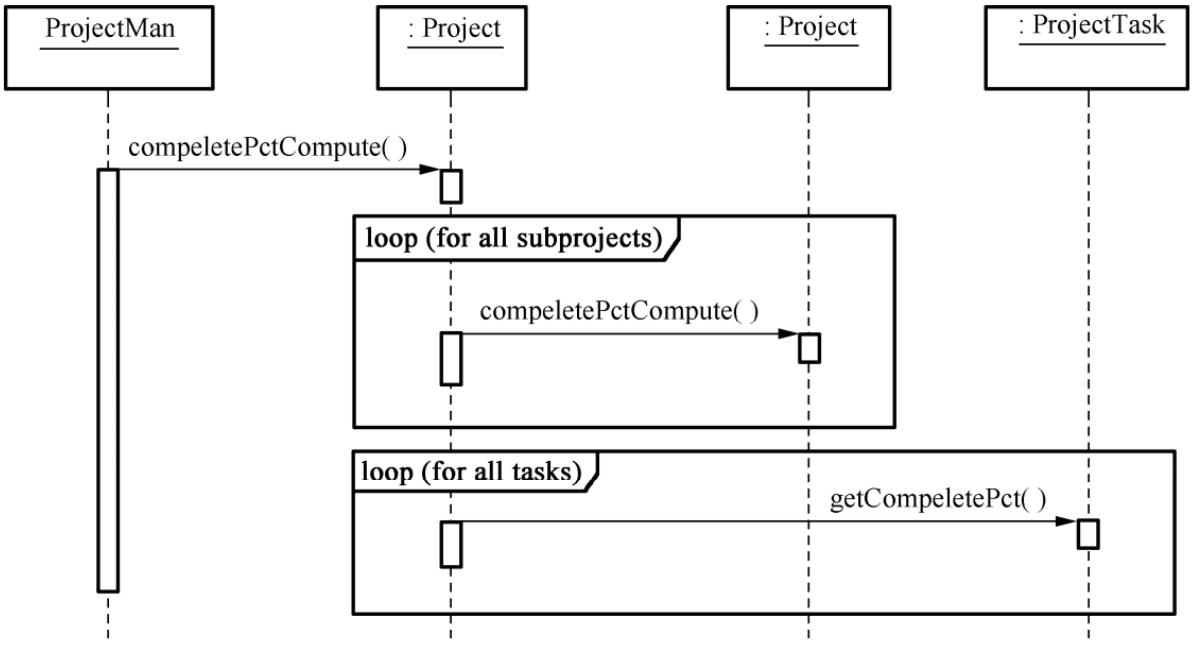


图 5.18　项目工作量的修改

自调用

类中经常会存在一种方法，它们主要用于类内部的计算。在顺序图中的描述形式是从某对象的生命线出发又回到同一生命线的消息，即自调用





5.4 界面类设计

基本要求

通过界面使得模型中含有的业务类的某些部分对外部可见，如用户通过人机交互界面对某些业务内容进行修改或访问。

扩展实体类模型

如果要对现有的实体类模型扩展对应的界面描述，直接的方法就是对每个实体类补充一个对应的接口。

界面类中的方法

1. 界面类中含有的方法一部分来源于对应的实体类，在实现中，这些界面对象会获得并使用输入的数据，然后调用对应的类方法驱动业务。
2. 界面类中还有一部分方法直接来源于需求规格说明书中文本形式的需求。

边界类和控制类

图 5.21　边界类和控制类

边界类和控制类在 UML 中也有对应的构造型<>和<>，表示上也有便捷的图形符号，如图 5.21 所示。

实例

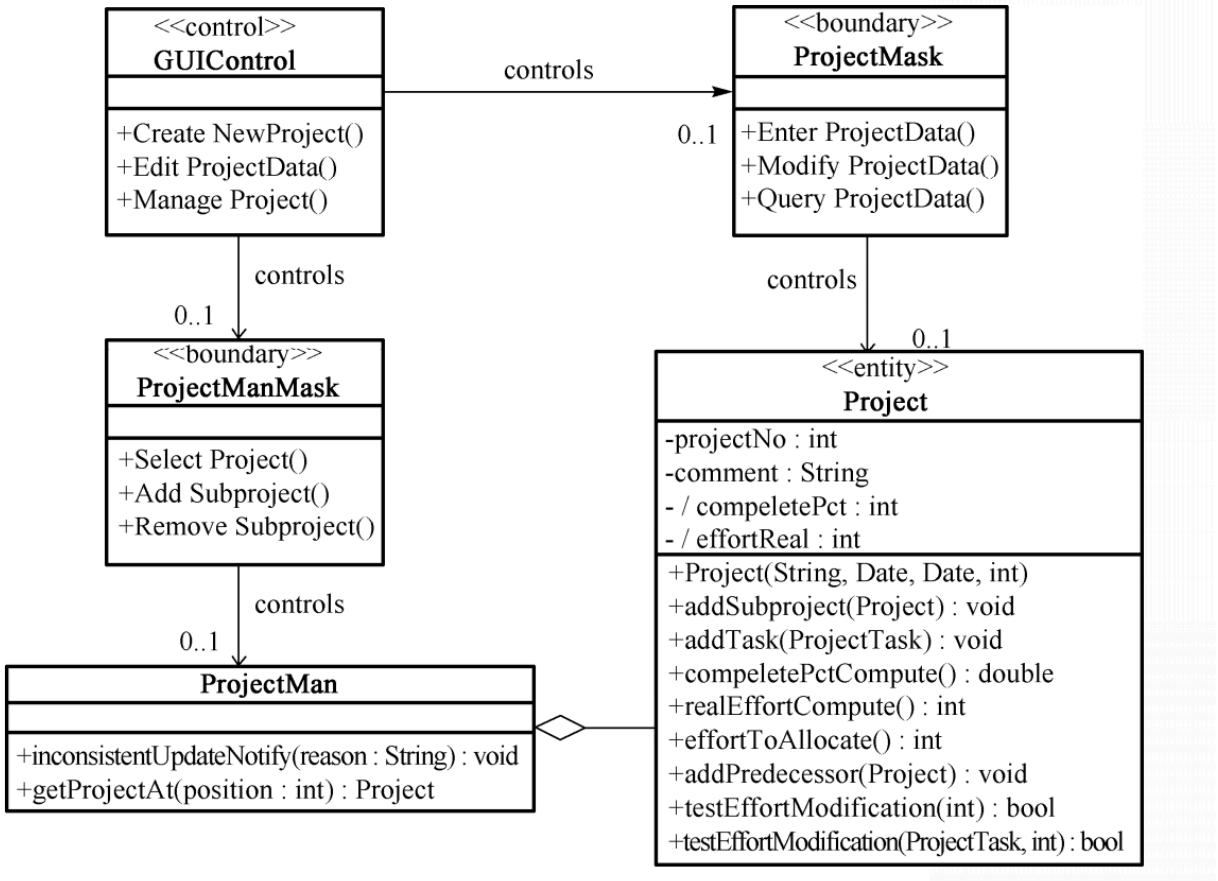


图 5.22　“项目编辑”用例对应的初始设计方案

1. 界面类的方法中没有参数，只是一个对功能性需求的抽象说明，这是因为它们需要的参数一般是在外部通过界

面元素读取并初始化的，如文本输入框。

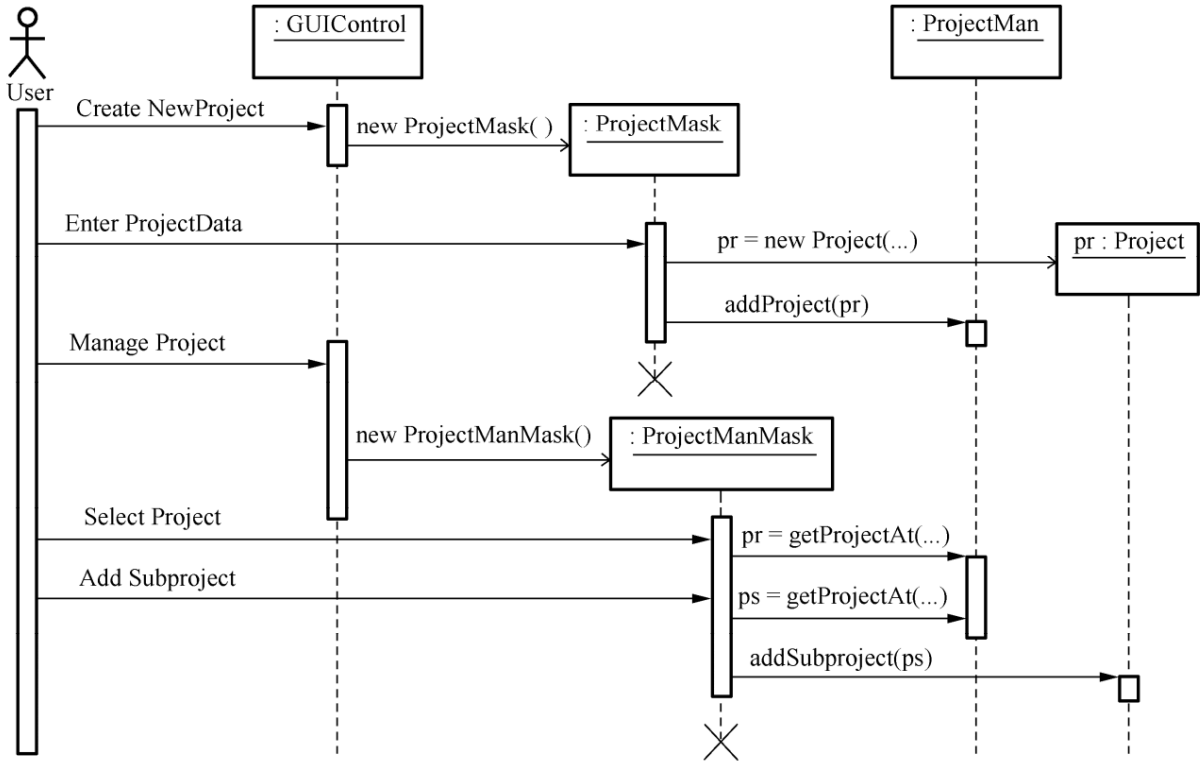


图 5.23　带有子项目的项目对象创建

2. 每个类都配有一个界面类。对于管理类或控制类来说，这是必要的，因为它们都分别需要通过界面来响应对业务实体的创建和访问等请求。
3. 在很多实际的业务中，一个界面类可能会涉及多个业务实体，也就是说，如果所有的业务驱动都围绕控制管理类展开，这样就可以减少场景中参与的类的数量，也就简化了模型的消息，从而有助于对系统整体的理解和维护，这样也就不需要为每个实体类设置界面类了。

第6章：代码生成

6.1 逆向工程与 CASE 工具

软件开发环境

支持软件开发的工具及其集成机制，用以支持软件开发的过程、活动和任务，为软件的开发、维护及管理提供统一的平台，即计算机辅助软件工程(CASE)。

套件

其中包含的工具涵盖了开发的主要阶段，它们各有所长、相互协调，共同完成开发任务，但彼此之间并非紧密依赖，缺一不可。

IDE

在软件开发环境中，一种常用的工具框架是集成开发环境(IDE)，它们有很多开源产品可供使用，如 Eclipse 主要为 Java 项目提供开发支持。

CASE 环境的搭建

CASE 环境的搭建与开发过程的选择具有很强的依赖关系，其中较重要的一点是要考虑开发过程中对各种“变更”的管理方式，这里存在两种较为极端的情况：

1. 需求分析、概要设计和详细设计过程只进行一次或者迭代增量式地进行。当编码开始后，对已经完成产品的修改只在程序代码中体现。需求分析、概要设计及详细设计文档不做更新。
2. 对每个改动的意愿，都要经过完整的分析、概要设计和详细设计流程，所有的改动都需要在所属的文档及代码中对应修改，并保证它们的一致性。

开发方法管理

开发方法的选择与实际的项目相关。

管理方法	使用情景
（1）	项目结束后不需要长期维护
（2）	项目需要长期维护且不断扩展功能

由类图向代码的转换

从类图向代码实现的转化过程中需要引入正向和逆向工程的技术方法。

技术方法	使用情景
正向工程	从类图出发生成程序代码的框架，进一步的开发可直接在代码上进行。
逆向工程	将代码的修改反向映射回类图的设计中，从而在设计与代码实现之间保持一致性。逆向工程的一种特殊用途是完全通过代码生成进行设计，在原始设计缺失时，可以起到对代码快速理解的作用。

逆向工程优劣

优势	说明
优势	逆向工程使得所有的开发可在 CASE 环境中同时开展，并使设计与实现之间相互对应。
劣势	工具之间可能并不能完全地无缝集成，尤其是当引入了某些高效或者独立的开发工具时，它们可能会阻碍逆向工程的进行，使模型和代码之间的紧密性大打折扣。

6.2 单个类的实现

本节主要介绍从类图到程序代码框架的生成过程，这是由设计到实现必然要经过的环节。

类中包含的信息

一个类图如果可以成功翻译为代码，类模型中的内容必须要完整。

1. 每个实例变量，需要指定其类型。——针对成员变量
2. 每个方法中的参数和返回值，需要指定其类型。——针对方法
3. 每个关联关系，其关联类型、导航方向必须说明。——针对类间关系

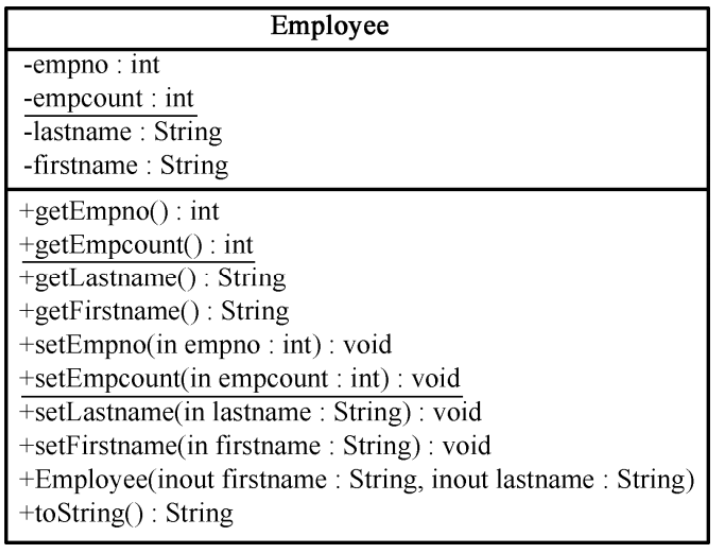


图 6.1　具有类变量和类方法的类

释：

	释
划横线的属性	代表类变量(静态)，类变量表示它用来对类范围的某个属性进行说明，而不是类的每个对象。每个对象可以访问其对应类中的类变量，但对其改动将作用于该类的所有对象，因为类的类变量在内存中只存储一份，在所有的对象实例中共享。
划横线的方法	代表类方法（静态），通过类方法，可以实现对类变量的访问，但类方法不能访问一般的实例变量。类变量在编译时预分配存储空间，不需要使用任何方法在运行时动态分配。

类变量和类方法

1. 对于类方法的调用，在大多数程序设计语言中，可直接使用以下形式：

   .()

2. 类变量的实质是一个静态变量，对于该类的所有实例，都可以共享访问，从而增加了这些对象之间的耦合性。
3. 习惯的做法是将这些常规方法以静态的形式在一个工具类中封装，不含有其他实例变量和方法，所有外部类可方便使用该工具类，因为不需要去创建该类的实例。

   如，Java中的 Math 类，其中包含了绝大多数基本的数值计算功能。

4. 利用类变量在实例之间的共享特性，可以用来记录一些全局信息，如生成唯一的员工编号等。

类中方法的关键字

关键字	说明
in	表示参数在方法内部是只读的，不会被修改。
inout	表示该参数在方法的处理过程中会被访问，也可以被修改，而且在方法结束后对该参数的修改结果可以保持，即可以被外部接收到
out	该参数只能作为方法内部计算结果，也就意味着调用该方法时，out 参数可被赋予任何值(哑值)，对于方法的内部计算不起任何作用，方法结束后该参数记录并保持计算结果。

对图6.1类的实现

```
public class Employee{

 /**
 * @uml.property name="empno"
 */
 private int empno;

 /**
 * Getter of the property <tt>empno</tt>
 * @return Returns the empno.
 * @uml.property name="empno"
 */
 public int getEmpno(){
 return empno;
 }

 /**
 * Setter of the property <tt>empno</tt>
 * @param empno The empno to set.
 * @uml.property name="empno"
 */
 public void setEmpno(int empno){
```

```
this.empno = empno;
}

/**
* @uml.property firstname=""
*/
private String firstname = "";

/**
* Getter of the property <tt>firstname</tt>
* @return Returns the firstname.
* @uml.property firstname=""
*/
public String getFirstname(){
return firstname;
}

/**
* Setter of the property <tt>firstname</tt>
* @param firstname The firstname to set.
* @uml.property firstname=""
*/
public void setFirstname(String firstname){
this.firstname = firstname;
}

/**
* @uml.property lastname=""
*/
private String lastname;

/**
* Getter of the property <tt>lastname</tt>
* @return Returns the lastname.
* @uml.property lastname=""
*/
public String getLastname(){
 return lastname;
}

/**
* Setter of the property <tt>lastname</tt>
* @param lastname The lastname to set.
* @uml.property lastname=""
*/
public void setLastname(String lastname){
 this.lastname = lastname;
}
```

```
 /**
 * @uml.property empcount
 */
 private static int empcount;

 /**
 * Getter of the property <tt>getEmpcount</tt>
 * @return Returns the empcount.
 * @uml.property empcount
 */
 public static int getEmpcount(){
  return empcount;
 }

 /**
 * Setter of the property <tt>empcount</tt>
 * @param empcount The empcount to set.
 * @uml.property empcount
 */
 public static void setEmpcount(int empcount){
  Employee.empcount = empcount;
 }

 /**
 * Construction of the property <tt>firstname</tt> and <tt>lastname</tt>
 * @uml.property firstname
 * @uml.property lastname
 */
 public Employee(String firstname, String lastname){
  this.firstname = firstname;
  this.lastname = lastname;
  this.empno = Employee.empcount++;
}

  /**
 * Override of the method <tt> toString</tt>
 * @return Returns the "empno: firstname lastname"
 */
 @Override
 public String toString(){
  return empno + ": " + firstname + " " + lastname;
 }
}
```

注释分类

1. 在设计类图时给出的，由 CASE 工具自动生成；
2. 是 CASE 工具内部使用的注释，用以在类模型和代码之间起连接的作用，为逆向工程提供支持。

代码风格

首先给出实例或者静态变量的定义，然后紧跟着是对该变量的 get 和 set 方法。

代码自动生成

对于较复杂的计算业务，只有在设计中给出了详细的业务计算逻辑后，才有可能实现自动的、完整的代码生成。业务越复杂，代码能够自动生成的可能性就越低，对于这部分需求，还需要工具的开发者为此付出大量努力。