软件工程(961)

考试题型：概念问答题、实践案例题

总分：50分

一、软件过程

软件过程的概念

软件过程是指软件整个生命周期，从需求获取，需求分析，设计，实现，测试，发布和维护一个过程模型。一个软件过程定义了软件开发中采用的方法，但软件过程还包含该过程中应用的技术——技术方法和自动化工具。过程定义一个框架，为有效交付软件工程技术，这个框架必须创建。软件过程构成了软件项目管理控制的基础，并且创建 了一个环境以便于技术方法的采用、工作产品（模型、文档、报告、表格等）的产生、里程碑的创建、质量的保证、正常变更的正确管理。

经典软件过程模型的特点（瀑布模型、增量模型、演化模型、统一过程模型）

瀑布模型（Waterfall Model）

1970年Winston Royce提出了著名的”瀑布模型”，直到80年代早期，它一直是唯一被广泛采用的软件开发模型。 瀑布模型将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护等六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。

在瀑布模型中，软件开发的各项活动严格按照线性方式进行，当前活动接受上一项活动的工作结果，实施完成所需

的工作内容。当前活动的工作结果需要进行验证，如果验证通过，则该结果作为下一项活动的输入，继续进行下一项活动，否则返回修改。

瀑布模型强调文档的作用，并要求每个阶段都要仔细验证。但是，这种模型的线性过程太理想化，已不再适合现代的软件开发模式，几乎被业界抛弃，其主要问题在于：

(1) 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；

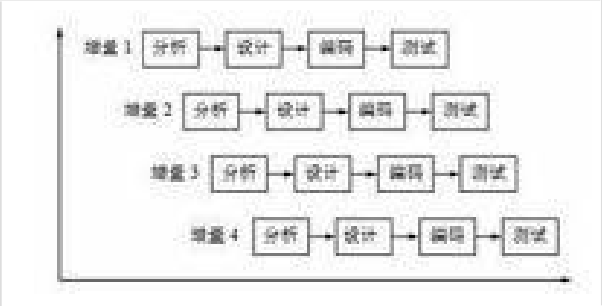
(2) 由于开发模型是线性的，用户只有等到整个过程的末期才能见到开发成果，从而增加了开发的风险；

(3) 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。

我们应该认识到，”线性”是人们最容易掌握并能熟练应用的思想方法。当人们碰到一个复杂的”非线性”问题 时，总是千方百计地将其分解或转化为一系列简单的线性问题，然后逐个解决。一个软件系统的整体可能是复杂 的，而单个子程序总是简单的，可以用线性的方式来实现，否则干活就太累了。线性是一种简洁，简洁就是美。当 我们领会了线性的精神，就不要再呆板地套用线性模型的外表，而应该用活它。例如增量模型实质就是分段的线性 模型，螺旋模型则是接连的弯曲了的线性模型，在其它模型中也能够找到线性模型的影子

增量模型（Incremental Model）

与建造大厦相同，软件也是一步一步建造起来的。在增量模型中，软件被作为一系列的增量构件来设计、实现、集成和测试，每一个构件是由多种相互作用的模块所形成的提供特定功能的代码片段构成. 增量模型在各个阶段并不交付一个可运行的完整产品，而是交付满足客户需求的一个子集的可运行产品。整个产品被分解成若干个构件，开 发人员逐个构件地交付产品，这样做的好处是软件开发可以较好地适应变化，客户可以不断地看到所开发的软件， 从而降低开发风险。但是，增量模型也存在以下缺陷：



(1) 由于各个构件是逐渐并入已有的软件体系结构中的，所以加入构件必须不破坏已构造好的系统部分，这需要软件具备开放式的体系结构。

(2) 在开发过程中，需求的变化是不可避免的。增量模型的灵活性可以使其适应这种变化的能力大大优于瀑布模型和快速原型模型，但也很容易退化为边做边改模型，从而是软件过程的控制失去整体性。

在使用增量模型时，第一个增量往往是实现基本需求的核心产品。核心产品交付用户使用后，经过评价形成下一个增量的开发计划，它包括对核心产品的修改和一些新功能的发布。这个过程在每个增量发布后不断重复，直到产生最终的完善产品。

演化模型

增量模型与演化模型的相同点是：基本思想都是非整体开发，以渐增方式开发系统。他们的目的基本相同：使用户

尽早得到部分软件这样能听取用户反馈。不同点：增量模型再需求设计阶段是整体进行的，在编码测试阶段是渐增进行的。演化模型全部系统是增量开发，增量提交。

统一过程模型

统一过程模型是一种以“用例和风险驱动、以体系结构为核心、迭代及增量”为特征的软件过程框架，一般由UML方法和工具支持。用例是捕获需求的方法，因此，也可以说UP是需求驱动的。

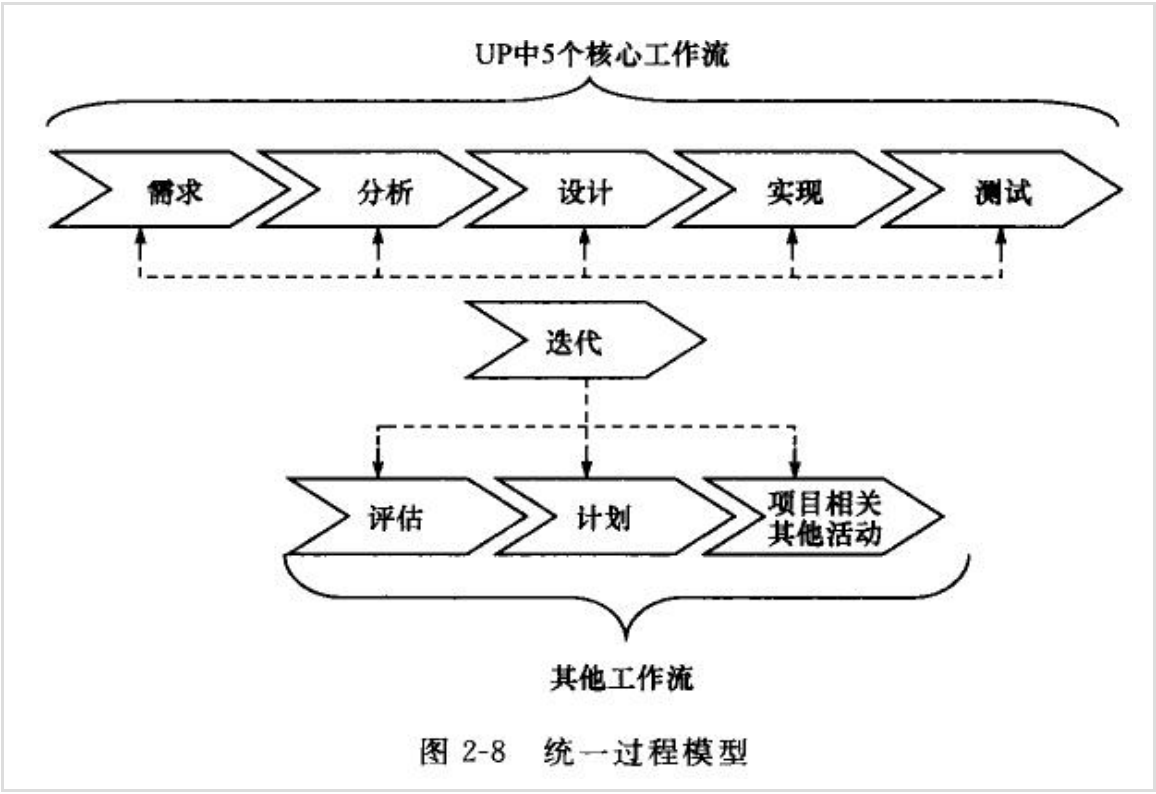


图 2-8　统一过程模型

UP的另一个驱动就是风险，因为如果你不主动预测和防范风险，风险就会主动攻击你。UP需要对软件开发中的风险进行分析、预测并关注软件的构造。

在基于组件的开发总，体系结构描述了系统的整体框架：如何把系统划分成组件以及这些组件如何进行交互和部署在硬件上。UP方法实际上就是开发和演进一个健壮的系统体系结构。

此外，UP也是迭代和增量的。在UP的迭代构建中，每个迭代包括五个核心工作流：

需求R–捕捉系统应该做什么。

分析A–精华和结构化需求。

设计D–基于系统结构来实现需求。

实现I–构造软件系统。

测试T–验证实现是否达到预期效果。

尽管每次迭代都可以包含这5个核心工作流，但是特定工作流的重点依赖于项目生命周期中的迭代发生的位置。迭代的一些可能工作流图解如图2–8所示。

把项目划分成一系列迭代，允许对项目进行灵活计划。最简单的方法是按照时间顺序的迭代序列，一个接一个。然

而，常常可能并行安排迭代。这意味着需要理解每次迭代的制品之间的依赖，需要有方法指导基于框架和模型的并行迭代。并行迭代的好处是缩短面市时间，可以更好地利用团队，但是必须仔细计划。

过程评估与CMM/CMMI的基本概念

过程评估

软件过程评估所关注的是软件组织自身内部软件过程的改进问题，目的在于发现缺陷，提出改进的方向。

CMM的概念

CMM(Capability Maturity Model)是能力成熟度模型的缩写,CMM是国际公认的对软件公司进行成熟度等级认证的重要标准。

CMM共分五级。在每一级中，定义了达到该级过程管理水平所应解决的关键问题和关键过程。每一较低级别是达到较高级别的基础。其中五级是最高级，即优化级，达到该级的软件公司过程可自发地不断改进，防止同类问题二次出现；四级称为已管理级，达到该级的软件公司已实现过程的定量化；三级为已定义级，即过程实现标准化；二级为可重复级，达到该级的软件公司过程已制度化，有纪律，可重复；一级为初始级，过程无序，进度、预算、功能和质量等方面不可预测。

CMMI的概念

（Capability Maturity Model Integration，能力成熟度模型集成）

将各种能力成熟度模型，即：Software CMM、Systems Eng-CMM、People CMM和Acquisition CMM，整合到同一架构中去，由此建立起包括软件工程、系统工程和软件采购等在内的诸模型的集成，

以解决除软件开发以外的软件系统工程和软件采购工作中的迫切需求。

CMMI的基本思想

1、解决软件项目过程改进难度增大问题

2、实现软件工程的并行与多学科组合

3、实现过程改进的最佳效益

敏捷宣言与敏捷过程的特点

敏捷宣言

也叫做敏捷软件开发宣言，正式宣布了对四种核心价值和十二条原则，可以指导迭代的以人为中心的软件开发方法。

敏捷宣言强调的敏捷软件开发的四个核心价值是：

个体和互动高于流程和工具

工作的软件高于详尽的文档

客户合作高于合同谈判

响应变化高于遵循计划

敏捷过程的特点

与传统开发方法相比，在敏捷开发的整个过程中，有以下几个主要的特点：

⑴ 敏捷开发的过程有着更强的适应性而不是预设性，从敏捷宣言的第四条响应变化高于预设计划便可以看出来。因为软件开发过程的本身的不可预见性，很多用户在项目开始时不可能对于这个项目有着一个完整而明确的预期。很多对软件的预期都在后期的修改和完善过程中产生。因此高适应性显然更加符合软件工程开发的实际。而敏捷开发实现其适应性的方式主要在于，第一，缩短把项目提交给用户的周期；第二，增加用户，业务人员，开发人员这三者之间的交流；第三，通过减少重构的成本以增加软件的适应性。

（2）敏捷开发的过程中，更加的注重人的因素。在传统软件工程中，个人的因素很少的被考虑到分工中，每个个体都是只是整个代码开发机器的一个小小的螺丝钉，个人的意志和创造力很大程度上的被抹去为了更好的为集体服务。而在敏捷开发过程中，每个个人的潜力被充分的考虑，应用什么技术很大程度上直接由在第一线开发的技术人员决定；每个人的特点和创造力都可以充分地发挥，这样开发出来的软件更加的具有生命力，因为他融入了开发者的心血和创意，开发者不再是进行机械的乏味的堆砌，而是创造属于自己的艺术品，这样的条件下产生的代码必然在质量上更占优势。

（3）在敏捷开发的过程中，整个项目是测试驱动的而不是文档驱动的。不仅每个模块有着自己的相应的测试单元，开发人员在开发自己的模块的过程中必须保证自己所开发的模块可以通过这一单元的测试，并且集成测试贯穿了整个开发过程的始终。集成测试每天会进行十几次甚至几十次，而不是像传统方法一样只有当各个模块的编码都结束了之后再进行联合调试。这样，在软件开发的进程中每一点改动所引起的问题都容嘉容易暴露出来，使得更加容易在错误刚刚产生的时候发现问题从而解决问题。这样就避免了在最后整个系统完成时错误隐藏的太深给调试造成极大的困难。

二、软件需求

软件需求的概念

软件需求是

1. 用户解决问题或达到目标所需条件或权能(Capability)。
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)所述条件或权能的文档说明。它包括功能性需求及非功能性需求，非功能性需求对设计和实现提出了限制，比如性能要求，质量标准，或者设计限制。

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(business requirement)

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(user requirement)

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(use　case)文档或方案脚本(scenario)说明中予以说明。

功能需求(functional requirement)

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(feature)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。软件需求各组成部分之间的关系如图所示。

非功能需求

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。

需求工程的基本过程

需求工程的活动
划分为以下5个独立的阶段：

- 需求获取：通过与用户的交流，对现有系统的观察及对任务进行分析，从而开发、捕获和修订用户的需求；
- 需求建模：为最终用户所看到的系统建立一个概念模型，作为对需求的抽象描述，并尽可能多的捕获现实世界的语义；
- 形成需求规格：生成需求模型构件的精确的形式化的描述，作为用户和开发者之间的一个协约；
- 需求验证：以需求规格说明为输入，通过符号执行、模拟或快速原型等途径，分析需求规格的正确性和可性，包含有效性检查，一致性检查，可行性检查和确认可验证性；
- 需求管理：支持系统的需求演进，如需求变化和可跟踪性问题。

需求获取阶段

需求获取首先需要的是技术的支持，其次，在需求获取工作中主要涉及了 3 个至关重要的因素：应搜集什么信息；从什么来源中搜集信息；用什么机制或技术搜集信息。再次，需求获取的开始，代表着软件项目正式开始实施，正所谓万事开头难。综合上述 3 个点使得需求获取成为软件开发中最困难、最关键、最易出错也是最需要交流的方面。在工作开展中，主要是就业务流程、组织架构、软硬件环境和现有系统等相关内容进行沟通，挖掘系统

最终用户的真正需求，把握需求的方向。在需求获取调研会中首先对需求获取方法作了验证。现行的需求获

取方法一般有基于调查的需求获取方法、基于用例的需求获取方法、原型法等几种方法。各种需求获取方法各有利弊。[7]

需求分析阶段

需求分析与需求获取是密切相关的，需求获取是需求分析的基础，需求分析是需求获取的直接表现，两者相互促进，相互制约。需求分析与需求获取的不同主要在于需求分析是在已经了解承建方的实际的客观的较全面的业务及相关信息的基础上，结合软、硬件实现方案，并做出初步的系统原型给承建方做演示。承建方则通过原型演示来体验业务流程的合理化、准确性、易用性。同时，用户还要通过原型演示及时地发现并提出其中存在的问题和改进意见和方法。

需求文档编写阶段

需求开发的最终成果是，在对所要开发的产品达成共识后，所编写的具体的文档。需求文档是在需求获取和需求分析两个阶段任务结束时生成的，所以文档要包含所有需求。在此阶段先要从软件工程和文档管理的角度出发依据相关的标准审核需求文档内容，确定需求文档内容是否完整。对需求文档中存留问题进行修改的工作。

需求确认阶段

需求确认主要是针对《需求规格说明书》的评审，保证需求符合优秀需求成熟的特征，并且符合好的需求规格说明的特征。在需求确认阶段需要保证以下几点：

(1) 软件需求规格说明正确描述了预期的满足各方涉众需求的系统能力和特征。

(2) 从系统需求、业务规则或其他来源中正确的推导出软件需求。

(3) 需求是完整的、高质量的。

(4) 需求的表示在所有地方都是一致的。

(5) 需求为继续进行产品设计和构造提供充分的基础。

需求跟踪阶段与需求复用阶段

需求跟踪是指通过比较需求文档与后续工作成果之间的对应关系，确保产品依据需求文档进行开发，建立与维护“需求——设计——编程——测试”之间的一致性，确保所有工作成果符合用户需求。需求跟踪是一项需要进行大量手工劳动的任务，在系统开发和维护的过程中一定要随时对跟踪联系链信息进行更新。需求跟踪能力的好坏会直接影响产品质量，降低维护成本，容易实现复用，同时，需求跟踪还需要建设方的大力支持。

需求复用阶段

在软件项目实施过程中，许多不同项目间存在着许多相似的需求，尤其是类型相同的项目在不同的用户群众的实施中，需求的相似性就更加明显、更加普遍了。有了需求复用，建设方就能快速的形成一个需求的原型，这样，后期

的需求工作只需要在此原型的基础上进行修改、扩充和完善即可，大大提高了需求分析的工作进度。所以，对于需求的复用就需要加以重视。对于需求复用，首要责任就是要提取可复用的需求，对需求复用的理解和扩充。其次就是要保证需求复用不存在冲突。

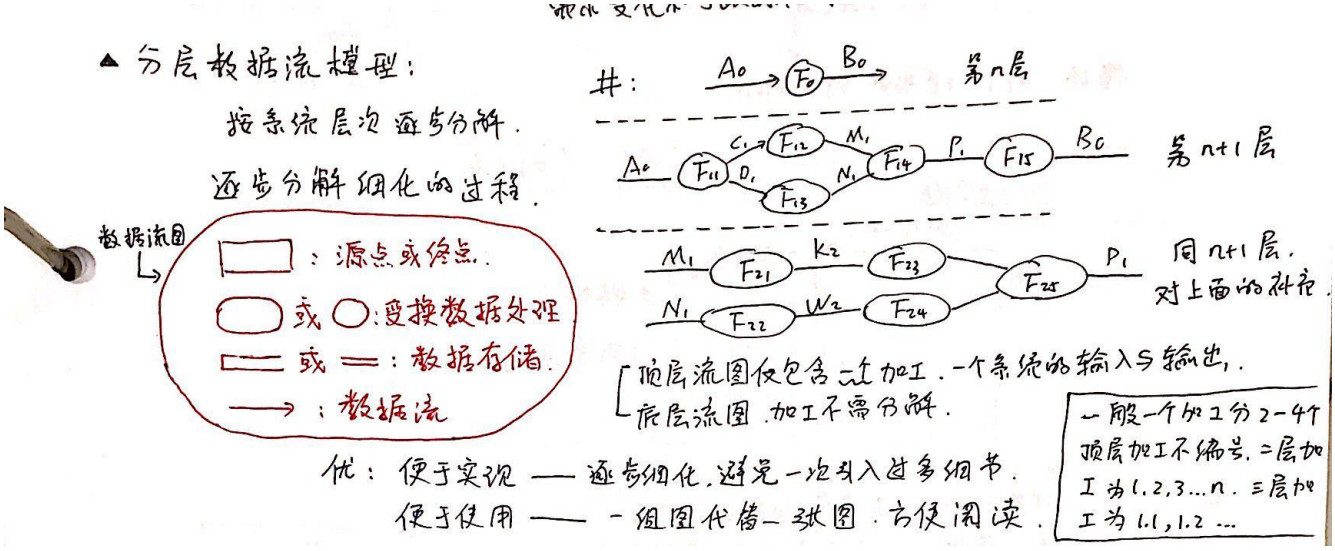
需求变更控制阶段

需求变更在软件项目开发中是不可避免的。无休止的需求变更只会造成各种资源无休止的浪费，但是其中也不乏有许多是必要的、合理的需求变更。对于需求变更，首先是要尽量及早的发现，以避免更大的损失。其次，是要采取相应的、合理的变更管理制度和流程，这样同样可以降低需求变更带来的风险。

版本控制阶段

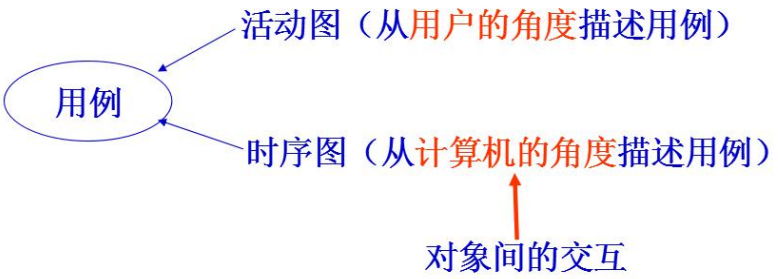
版本控制是管理需求规格说明和其他项目文档必不可少的一个方面，也是需求变更文档化管理的最有效办法。可以详细记录发生需求变更的需求文档版本的版本，发生变更的原因，变更发生的控制记录，并对变更后的需求文档进行唯一版本号的标识。使得每个成员都能及时访问最新版本的需求文档。
实施版本控制的基础是需求基线，所谓需求基线就是项目组成员一经承诺将在某一特定产品版本中实现的功能性和非功能性需求的集合。需求基线的确定可以保证项目的涉众各方可以对发布的产品中希望具有的功能和属性有一个一致的理解。

分层数据流模型



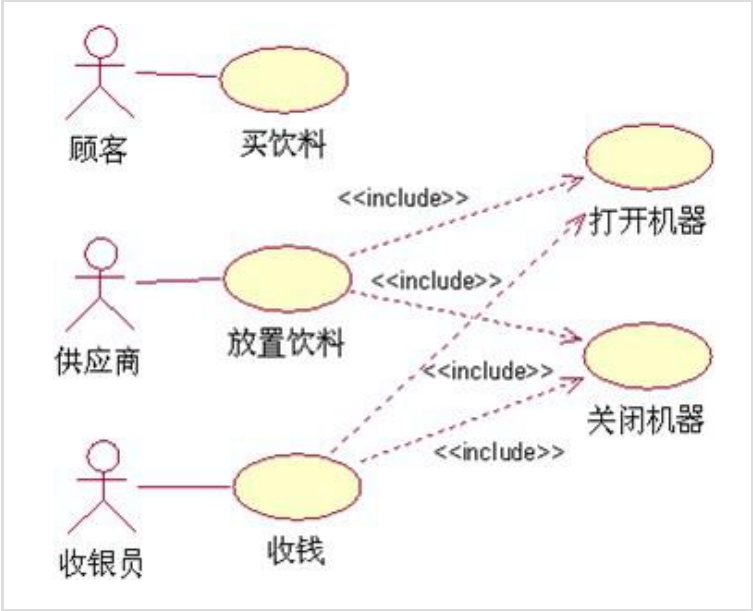
用例和场景建模及其UML表达（用例图、活动图、泳道图、顺序图）

用例图、活动图、时序图之间的关系



用例图

每一个用例表示一个具体的任务，涉及与系统的外部交互，在其最简单的形式中，用例用一个椭圆表示，参与者用一个小人儿表示。



解释：

1. 主角表示执行者(Actor)，其表示的是与当前系统交互交互的人或者其他系统。
2. 用例能够表示系统能够为执行者提供什么功能。
3. 用例是以动词加名词的形式，也就是动宾结构。
4. 外边框表示系统边界，要注明是什么系统，外边框可以不画，个人建议画上比较清晰。
5. 线条有三种：无箭头的，指向用例的箭头，指向执行者的箭头。
6. 箭头可以有两种解释：

   1、数据流向

   箭头指向用例，说明向系统输入数据。箭头指向执行者，说明系统输出数据。

   2、谁启动谁

   箭头指向用例，说明启动系统中某一模块。箭头指向执行者，说明系统启动另一系统。

包含关系

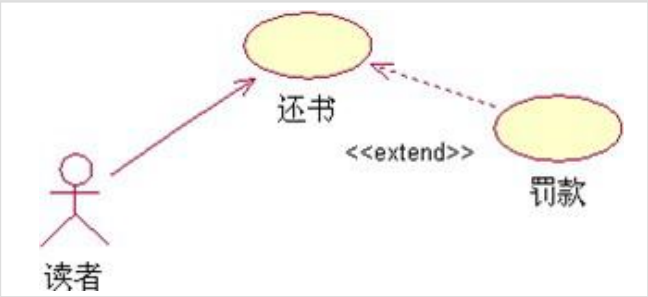
包含关系描述的是一个用例需要某种功能，而该功能被另外一个用例定义，那么在用例的执行过程中，就可以调用已经定义好的用例。

表示符号<>

扩展关系

用一个用例（可选）扩展另一个用例（基本例）的功能。

符号表示<>

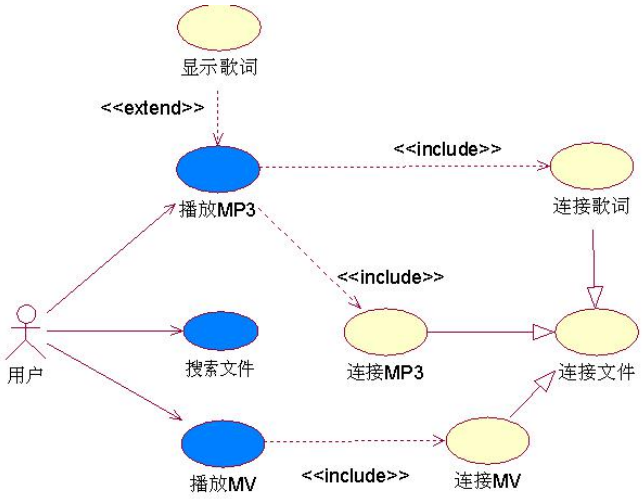


活动图

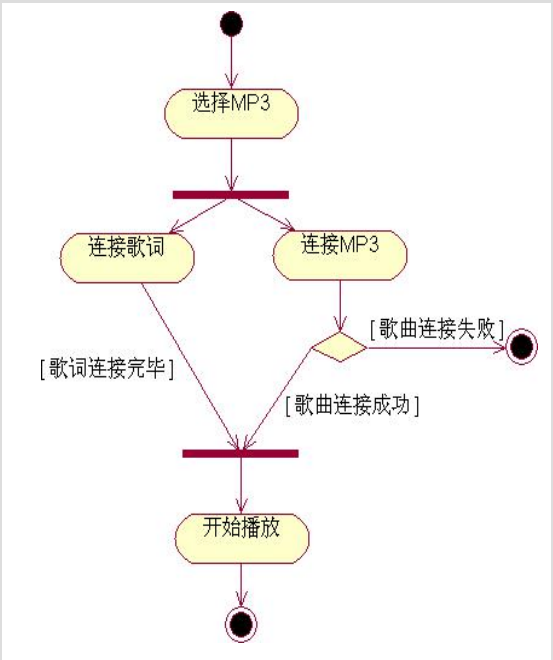
实例描述：

假如现在有一个简单的音乐盒，用户根据歌曲名称搜索自己喜爱的音乐，找到后，用户可以以MP3模式播放（同步显示歌词），也可以以MV模式播放。由于音乐和歌词都来自于互联网，因此在播放之前应连接这些文件。

根据上述描述，该音乐盒的用例图如下：

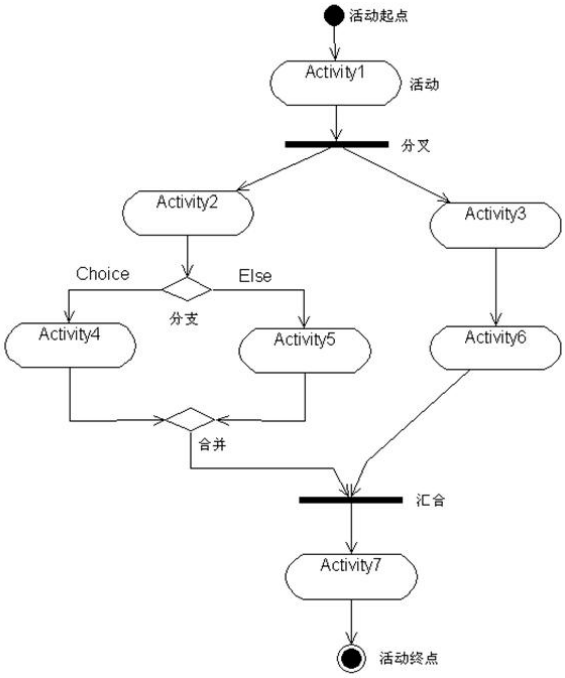


描述MP3播放功能：



活动图的概念及作用

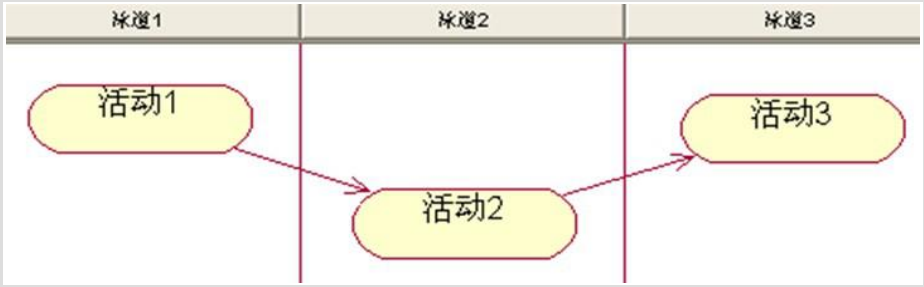
- 概念 活动图本质上是一种流程图，它描述活动的序列，即系统从一个活动到另一个活动的控制流。作
- 用 描述用例，描述类的操作，另外，可以用来描述算法（单独使用）。



注意：一个活动图中只能有一个开始状态，但可以有多个结束状态。（例如上面的MP3活动图，就有2个结 束状态）

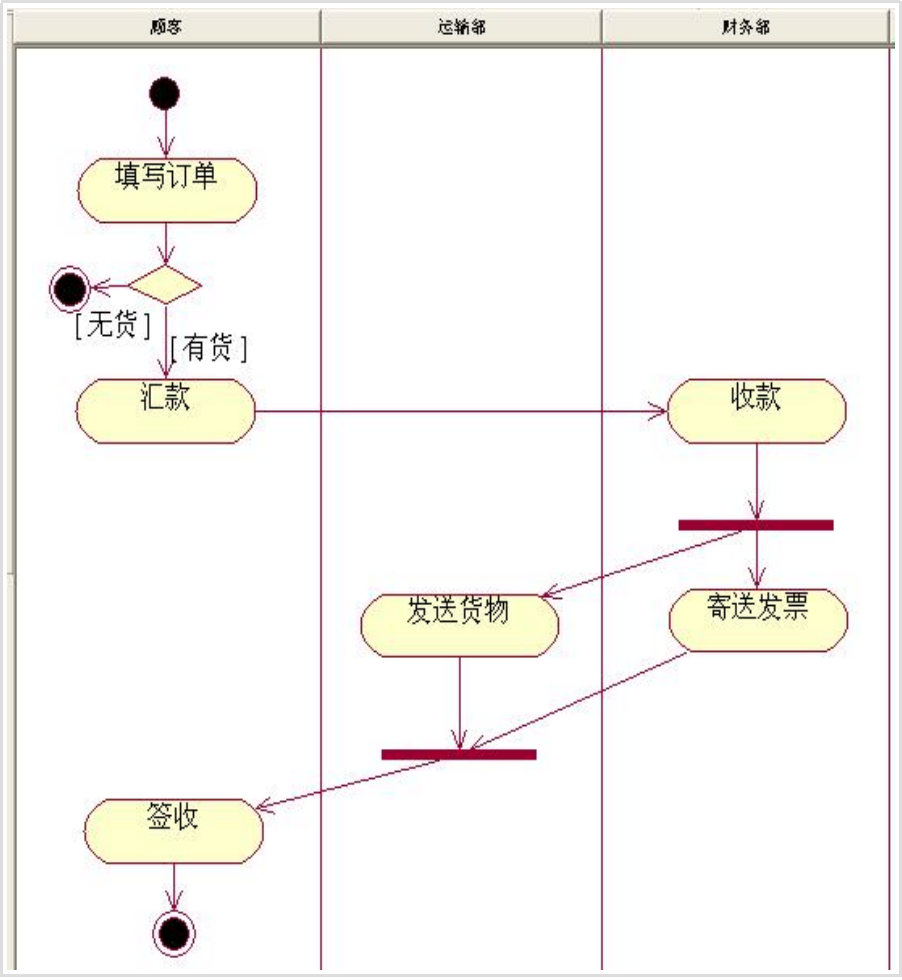
泳道

泳道表明每个活动是由哪些人或哪些部门负责完成。



在活动图中泳道区分了负责活动的对象，它明确地表示了哪些活动是由哪些对象进行的。

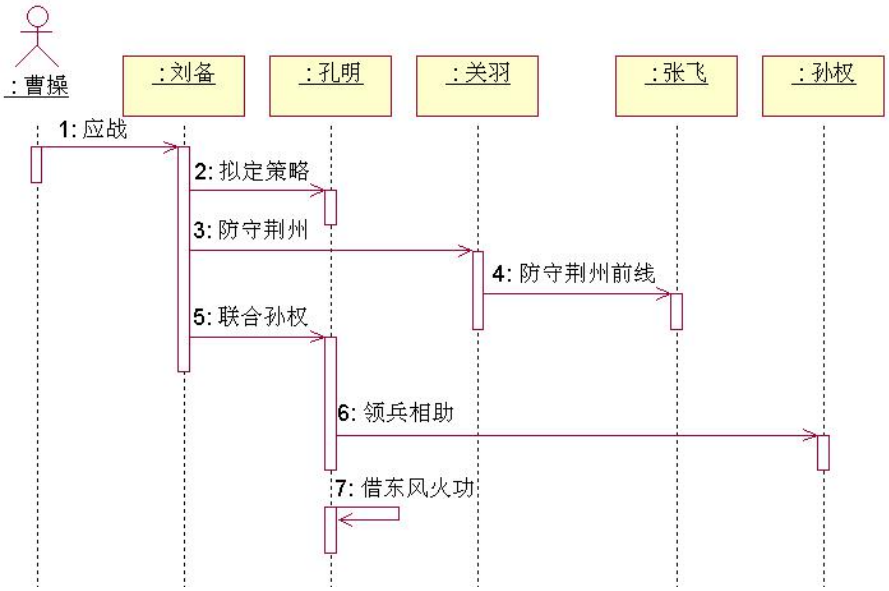
在包含泳道的活动图中每个活动只能明确地属于一个泳道。



时序图

时序图表示在特定用例中的交互发生的顺序

涉及的对象和参与者列在图表顶端，向下垂直画一条虚线。对象之间的交互用带注释的箭头表示。虚线上的矩阵表示相关对象的生命线（比如对象实例运行所需要的时间），从上往下为交互的顺序。箭头上的注释表示对对象的调用，他们的参数，以及返回值。



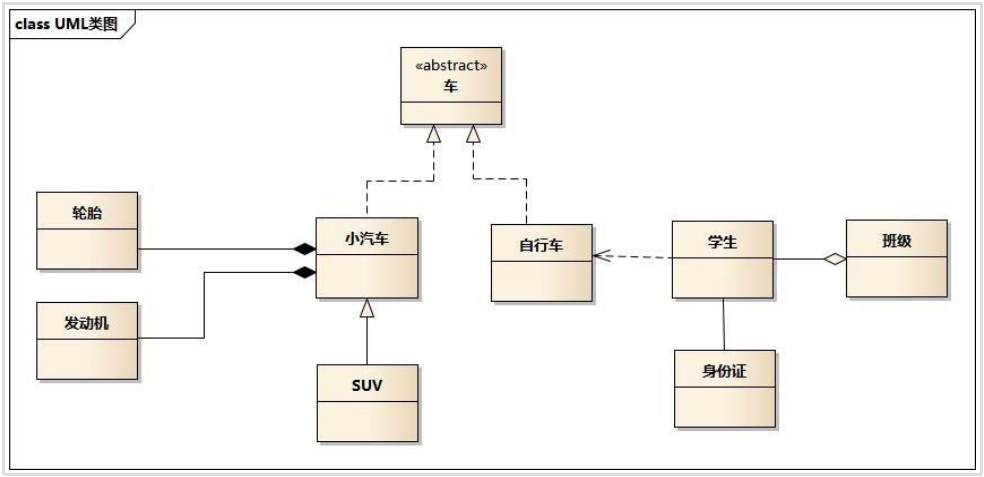
消息的概念：

对象之间的交互是通过相互发消息来实现的。一个对象可以请求（要求）另一个对象做某件事件。

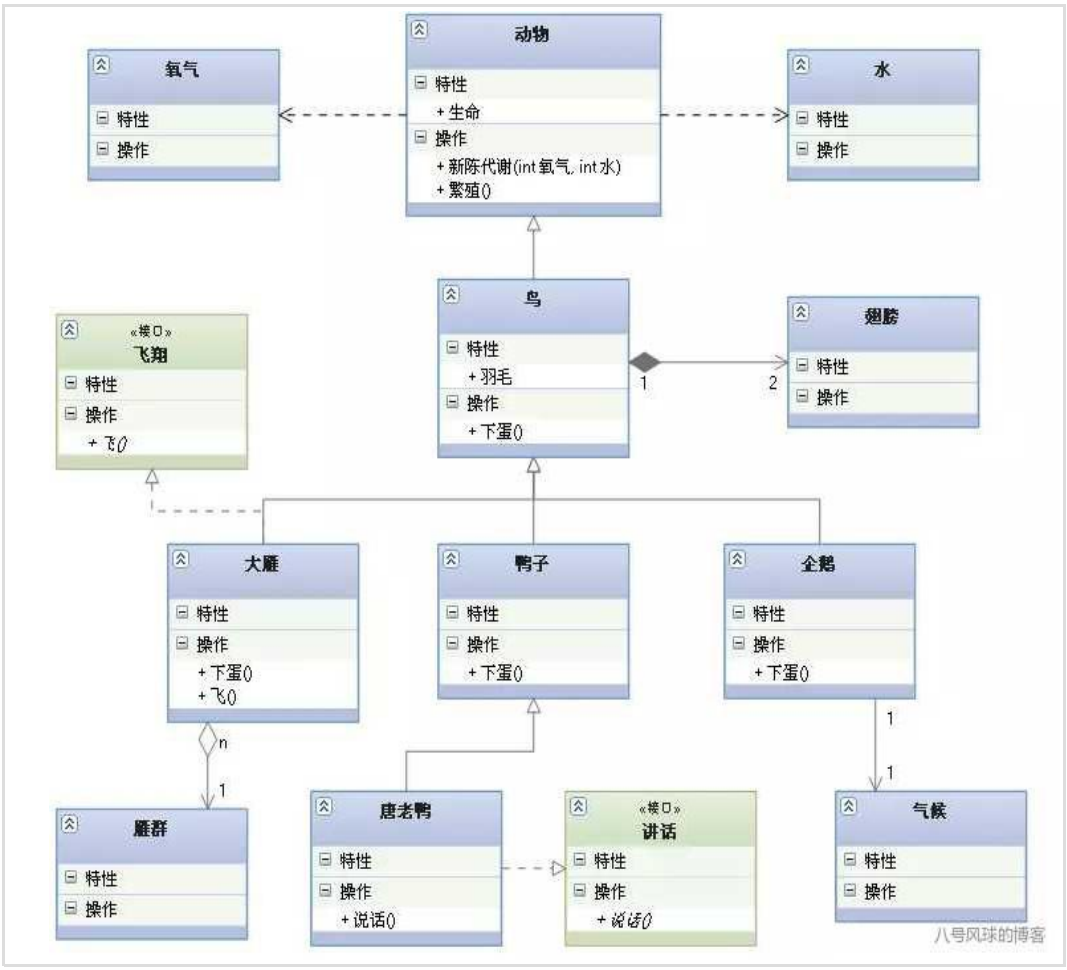
消息从源对象指向目标对象。消息一旦发送便将控制从源对象转移到目标对象。

数据模型建模及其UML表达（类图）

类图



- 车的类图结构为<>，表示车是一个抽象类；
- 它有两个继承类：小汽车和自行车；它们之间的关系为实现关系，使用带空心箭头的虚线表示；
- 小汽车为与SUV之间也是继承关系，它们之间的关系为泛化关系，使用带空心箭头的实线表示；
- 小汽车与发动机之间是组合关系，使用带实心箭头的实线表示；
- 学生与班级之间是聚合关系，使用带空心箭头的实线表示；
- 学生与身份证之间为关联关系，使用一根实线表示；
- 学生上学需要用到自行车，与自行车是一种依赖关系，使用带箭头的虚线表示；

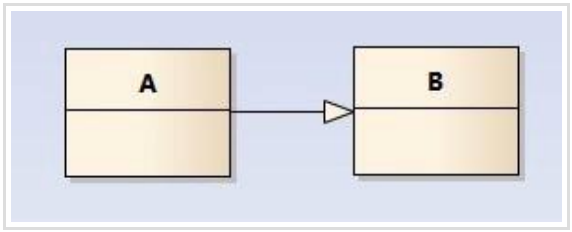


类之间的关系

泛化关系(generalization)

eg：自行车是车、猫是动物

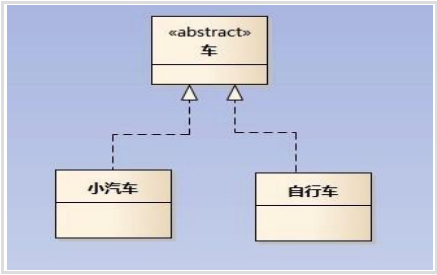
泛化关系用一条带空心箭头的直接表示；如下图表示（A继承自B）；



实现关系(realize)

实现关系用一条带空心箭头的虚线表示；

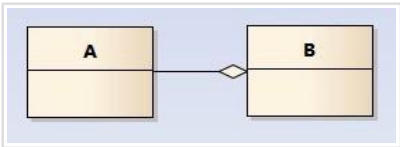
eg：”车”为一个抽象概念，在现实中并无法直接用来定义对象；只有指明具体的子类(汽车还是自行车)，才 可以用来定义对象（”车”这个类在C++中用抽象类表示，在JAVA中有接口这个概念，更容易理解）



聚合关系(aggregation)

聚合关系用一条带空心菱形箭头的直线表示，如下图表示A聚合到B上，或者说B由A组成；聚合关系用于表示实体对象之间的关系，表示整体由部分构成的语义；例如一个部门由多个员工组成；

与组合关系不同的是，整体和部分不是强依赖的，即使整体不存在了，部分仍然存在；例如， 部门撤销了，人员不会消失，他们依然存在；

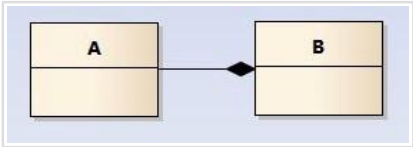


组合关系(composition)

组合关系用一条带实心菱形箭头直线表示，如下图表示A组成B，或者B由A组成；

与聚合关系一样，组合关系同样表示整体由部分构成的语义；比如公司由多个部门组成；

但组合关系是一种强依赖的特殊聚合关系，如果整体不存在了，则部分也不存在了；例如， 公司不存在了，部门也将不存在了；

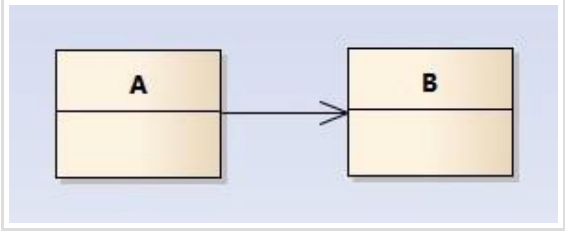


关联关系(association)

关联关系是用一条直线表示的；它描述不同类的对象之间的结构关系；它是一种静态关系， 通常与运行状态无关，一般由常识等因素决定的；它一般用来定义对象之间静态的、天然的结构； 所以，关联关系是一种“强关联”的关系；

比如，乘车人和车票之间就是一种关联关系；学生和学校就是一种关联关系；

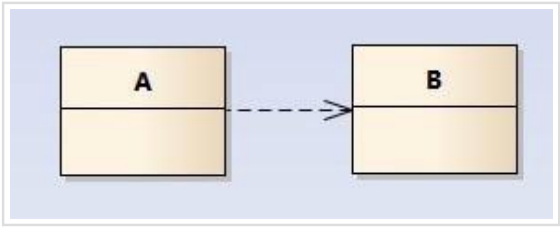
关联关系默认不强调方向，表示对象间相互知道；如果特别强调方向，如下图，表示A知道B，但 B不知道A；



注：在最终代码中，关联对象通常是以成员变量的形式实现的；

依赖关系(dependency)

依赖关系是用一套带箭头的虚线表示的；如下图表示A依赖于B；他描述一个对象在运行期间会用到另一个对象的关系；

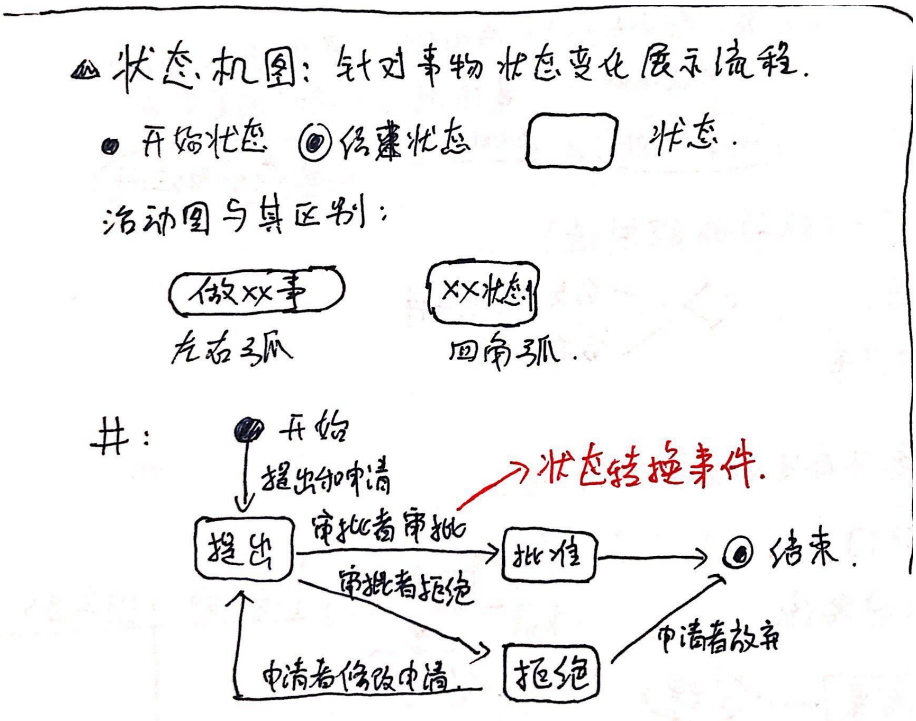


与关联关系不同的是，它是一种临时性的关系，通常在运行期间产生，并且随着运行时的变化； 依赖关系也可能发生变化；

显然，依赖也有方向，双向依赖是一种非常糟糕的结构，我们总是应该保持单向依赖，杜绝双向依赖的产生；

注：在最终代码中，依赖关系体现为类构造方法及类方法的传入参数，箭头的指向为调用关系；依赖关系除了临时知道对方外，还是“使用”对方的方法和属性；

行为模型建模及其UML表达（状态图）。



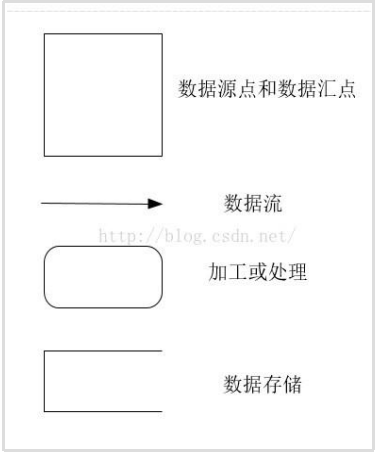
数据流图（DFD）

数据流图(DFD)是结构化分析方法中使用的工具,它以图形的方式描绘数据在系统中流动和处理的过程，

由于它只反映系统必须完成的逻辑功能，所以它是一种功能模型。

在结构化开发方法中，数据流图是需求分析阶段产生的结果。

数据流图表示法



数据源点和数据汇点：指系统以外又与系统有联系的人或事物。

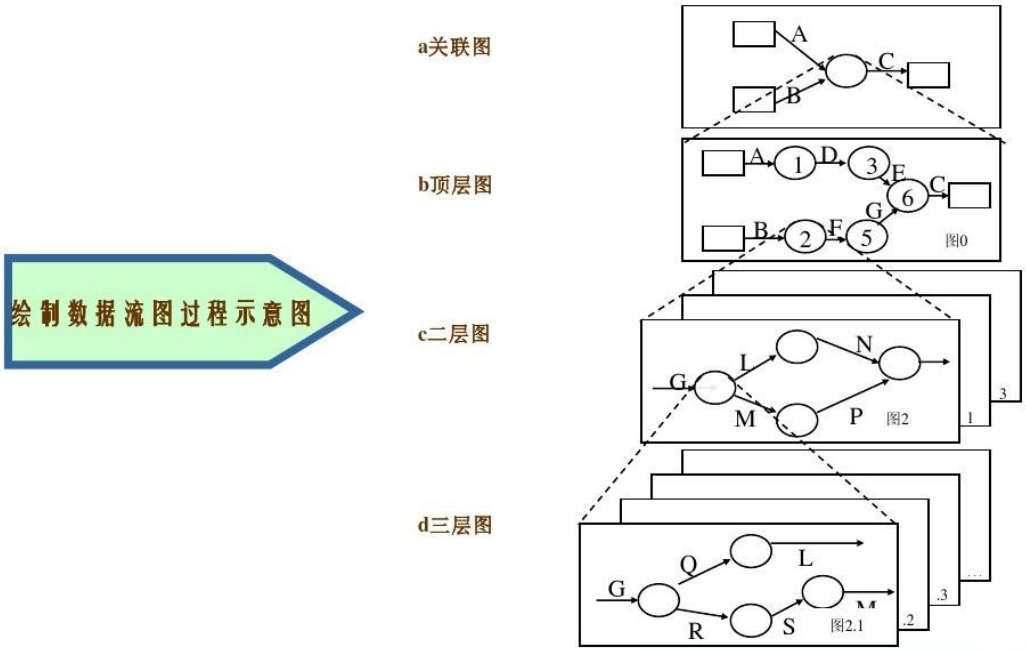
用来表达该系统数据的外部来源和去向。

数据流：指处理功能的输入或输出，箭头表示数据流向。

加工或处理：指对数据进行处理加工，使数据变换。

数据存储：表示某种独居保存后的逻辑统称，一般为表结构。

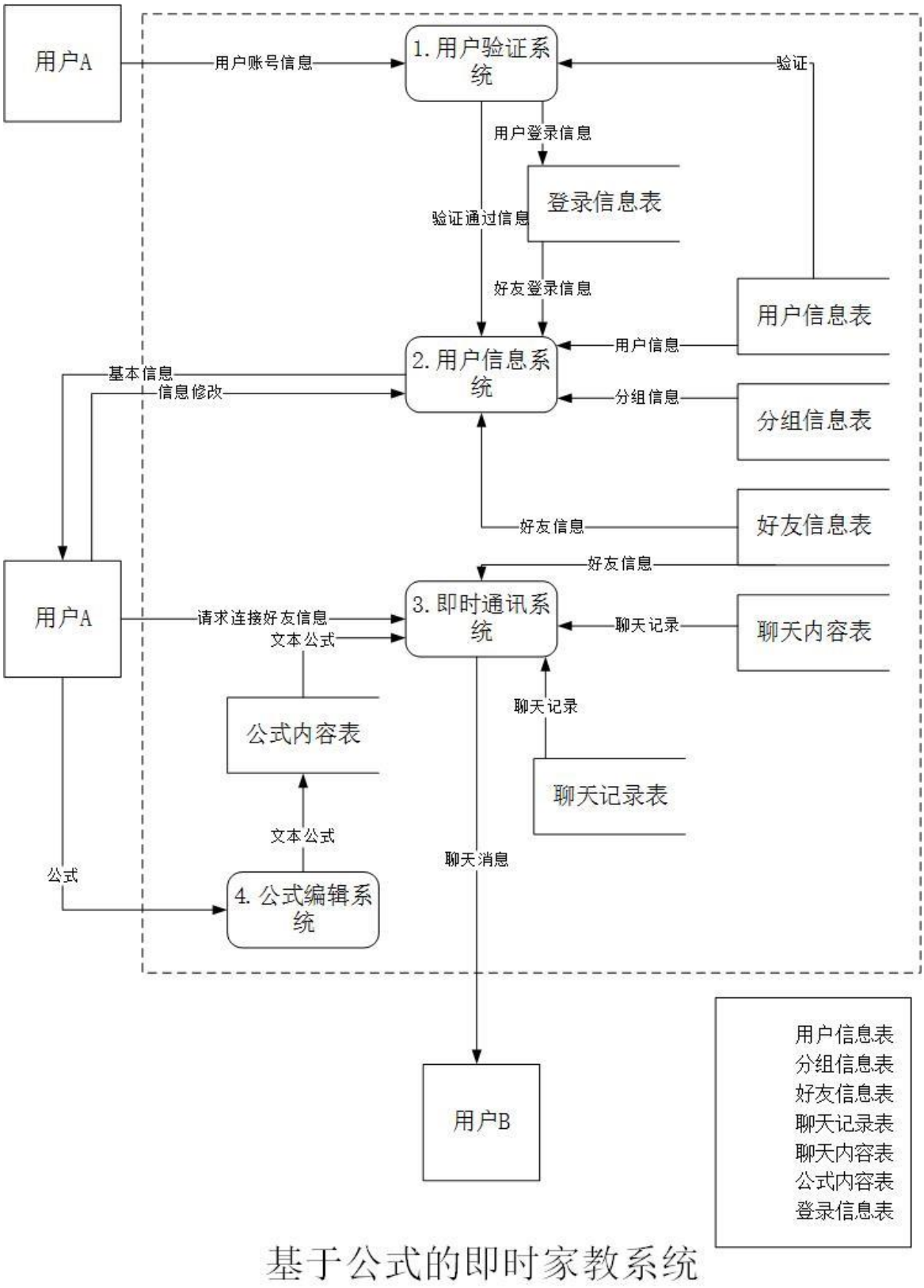
数据流程图实例



顶层数据流图

把整个系统视为一个大的加工，然后根据数据系统从哪些外部实体接收数据流，以及系统发送数据流到那些外部实

体，就可以画出输入输出图。这张图称为顶层图。



基于公式的即时家教系统

把顶层图的加工分解成若干个加工，并用数据流将这些加工连接起来，使得顶层图的输入数据经过若干加工处理后，变成顶层图的输出数据流。这张图称为0层图。从一个加工画出一张数据流图的过程就是对加工的分解。

三、软件设计与构造

软件体系结构及体系结构风格的概念

能够用来具体描述软件系统控制结构和整体组织的一种体系结构，能够表示系统的框架结构，用于从较高的层次上来描述各部分之间的关系和接口。软件体系结构是对系统的一种高层次的抽象描述。主要是反映拓扑属性，有意忽略细节；软件体系结构是由构件和构件之间的联系组成，构件又有它自身的体系结构；

构件的描述有3个方面：计算功能、结构特性及其他特性。

设计模式的概念

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式 可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断 重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

模块化设计的基本思想及概念（抽象、分解、模块化、封装、信息隐藏、功能独立）

模块化设计和建造就是在对产品进行功能分析的基础上，将产品分解成若干个功 能模块，预制好的模块再进行组装，形成最终产品。这里，模块（module）是指提供特定功 能的相对独立的单元。模块一般具有如下特征：

1. 标准化：模块是具有标准尺寸和标准接口的预制功能单元，这是组装、互换等特征 的基础。
2. 可组装：多个模块可以方便、灵活地组合、配置，以构造不同大小、不同形状、不 同功能的系统
3. 可替换：通过用一个模块去更换另一个模块，可以改变系统的局部功能而不影响系 统的其他部分
4. 可维护：可以对模块进行局部修改或设置，以满足用户的需求。另外可以在现有系 统中增加新模块，以扩展系统功能。

- 抽象 以概括性的术语描述解决方案 过程抽象和数据抽象
- 分离 关注点分离是一个设计概念，它表明任何复杂问题如果被分解成可以独立解决或者优化的若干块，该复杂问题可以更容易得到处理 分而治之
- 模块化 是关注点分离最常见的表现 软件被划分为独立命名可处理的构建，有时被称为模块，把这些构建集成到一起可以满足问题的需求。

  模块化设计使开发工作更易于规划；可以定义和交付软件增量；更容易实施变更；更有效的开展测试和调试；可以进行长期维护而没有严重的副作用。
- 封装

- 信息隐藏 每个模块对其他所有模块都隐蔽自己的升级决策，话句话说，模块应该被特别说明并设计，使信息都包含在模块内，其他模块无法对这些信息进行访问
- 功能独立 独立性可以通过两条定性的标准进行评估：内聚性和耦合性。内聚性显示了某个模块相关功能的强度，耦合性相识了模块间的相互依耐性。

软件重构的概念

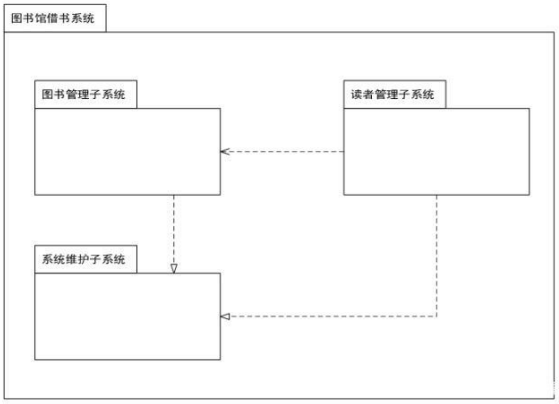
重构是一种重新组织的技术，可以简化构件的设计而无需改变其功能或行为。

软件体系结构的UML建模（包图、类图、构建图、顺序图、部署图）

包图

UML的包图模型类似Package结构，不同的是UML包图模型仅描述Package高层次的模块间关系，对于较低层的模块一般不涉及，这些较低层的模块一般通过设计类图来描述。UML包图属于高层设计模型。

当我们使用包图模型设计一个复杂系统时。首先要将系统进行功能分解，把一个大的系统划分为多个子系统。然后将整个系统作为一个包，划分的子系统作为子包放置在系统包内，包允许多层嵌套，建议最多两层，嵌套多层的包会增加较多的编码工作量，系统架构也会变得复杂。下图是图书馆借书系统的包图模型。



两个符号：

一个是带标题的矩形框，它表示主系统或子系统（也可以说主包或分包），标题填入主系统或子系统的名称，最外层的方框是主系统，子系统被放置在主系统方框之内，表示这些子系统是主系统的一部分。

一个是带箭头的虚线，表示包与包之间存在依赖关系，箭头的尾部表示被依赖的包，而头部是独立的包。例如在上图中，读者管理子系统需要通过图书管理子系统获取可借阅的图书，同时又需要从系统维护子系统中获取登录读者的权限，因此读者管理子系统依赖于图书管理子系统和系统维护子系统。另外，图书管理子系统也依赖于系统维护子系统。

包的依赖性表示一个包中的结构或代码依赖于另一个包。例如读者管理子系统在查询可借阅图书时，会调用图书管理子系统的图书查询方法获取图书列表信息。包之间的依赖关系会导致被依赖包的代码发生变动时，依赖包也要进行相应地修改。

构件图

一.构件图概述

1.概念

用来显示一组构件之间的组织及其依赖关系

2.基本元素

（1）构件：定义了良好接口的物理实现单元。

● 配置构件：形成可执行文件的基础，如：动态链接库（DLL）、ActiveX控件等。

● 工作产品构件：配置构件的来源，如：数据文件和程序源代码。

● 执行构件：最终可运行系统产生的运行结果。

（2）接口：一个类提供给另一个类的一组操作。

● 导出接口：导出接口有提供操作的构件提供。

● 导入接口：访问服务的组件使用导入接口。

（3）关系

依赖、泛化、关联和实现。

3.作用

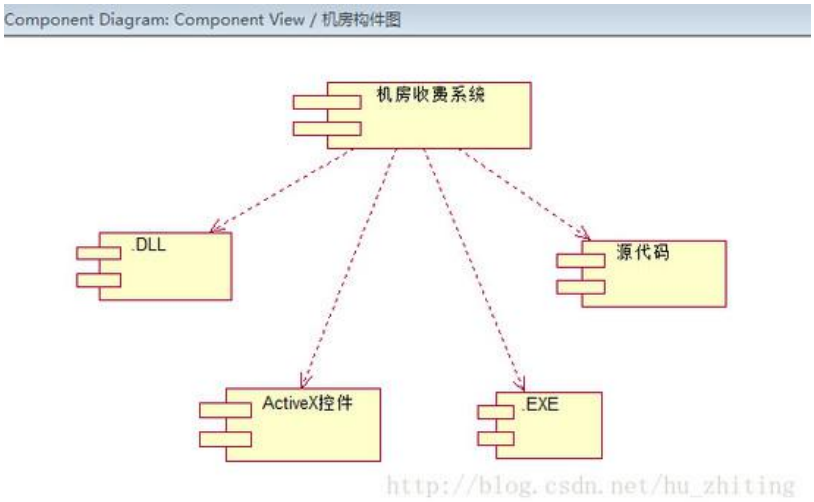
● 帮助客户理解最终的系统结构

● 使开发工作有一个明确的目标

● 有利于帮助工作组其他人员理解系统

● 有利于软件系统的组件重用

4.机房收费系统构件图



部署图

二.部署图概述

1.概念

用来描述系统硬件的物理拓扑结构以及在此结构上执行的软件。

2.基本元素

（1）节点：代表一个物理设备或者是一个运行在其上的软件系统。

（2）构件：可执行的物理代码模块。

（3）对象：类的实例。

（4）接口：对外提供可见操作和属性，其他构件通过接口使用构件。

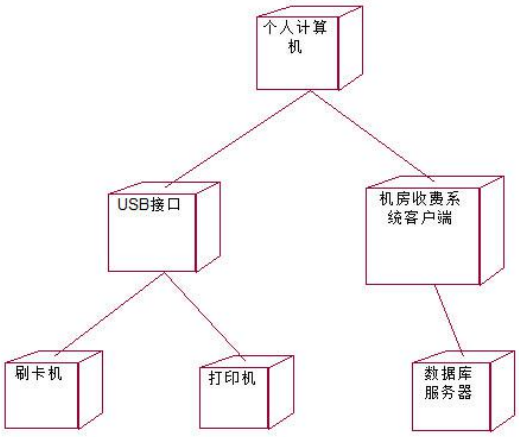
（5）连接：节点之间的连线，表示节点之间的关联。

（6）依赖关系：一个构件的改变随另一构件的改变而改变。

3.目的

- 研究系统投入使用的相关问题
- 描述一个商业应用主要的部署结构
- 设计一个嵌入系统的硬件和软件结构
- 描述一个组织得硬件/网络基础结构

4.机房收费系统部署图



面向对象设计原则（开闭原则、Liskov替换原则、依赖转置原则、接口隔离原则）

- 设 计 类
- 依赖倒置
- 测试设计
- 设计模型
- 数据设计元素
- 体系结构设计元素
- 接口设计元素
- 构件级设计元素
- 部署级设计元素

单一职责原则（Single Responsibility Principle，简称SRP ）

核心思想：应该有且仅有一个原因引起类的变更

问题描述：假如有类Class1完成职责T1，T2，当职责T1或T2有变更需要修改时，有可能影响到该类的另外一个职责正常工作。

好处：类的复杂度降低、可读性提高、可维护性提高、扩展性提高、降低了变更引起的风险。

需注意：单一职责原则提出了一个编写程序的标准，用“职责”或“变化原因”来衡量接口或类设计得是否优良，但 是“职责”和“变化原因”都是不可以度量的，因项目和环境而异。

里氏替换原则（Liskov Substitution Principle,简称LSP）

核心思想：在使用基类的的地方可以任意使用其子类，能保证子类完美替换基类。

通俗来讲：只要父类能出现的地方子类就能出现。反之，父类则未必能胜任。

好处：增强程序的健壮性，即使增加了子类，原有的子类还可以继续运行。

需注意：如果子类不能完整地实现父类的方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系 采用依赖、聚合、组合等关系代替继承。

依赖倒置原则（Dependence Inversion Principle,简称DIP）

核心思想：高层模块不应该依赖底层模块，二者都该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象；

说明：高层模块就是调用端，低层模块就是具体实现类。抽象就是指接口或抽象类。细节就是实现类。

通俗来讲：依赖倒置原则的本质就是通过抽象（接口或抽象类）使个各类或模块的实现彼此独立，互不影响，实现模块间的松耦合。

问题描述：类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案：将类A修改为依赖接口interface，类B和类C各自实现接口interface，类A通过接口interface间接与类B或者类C发生联系，则会大大降低修改类A的几率。

好处：依赖倒置的好处在小型项目中很难体现出来。但在大中型项目中可以减少需求变化引起的工作量。使并行开发更友好。

接口隔离原则（Interface Segregation Principle,简称ISP）

核心思想：类间的依赖关系应该建立在最小的接口上

通俗来讲：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

问题描述：类A通过接口interface依赖类B，类C通过接口interface依赖类D，如果接口interface对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。

需注意：

接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度

提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情

为依赖接口的类定制服务。只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。

迪米特法则（Law of Demeter,简称LoD）

核心思想：类间解耦。

通俗来讲：　一个类对自己依赖的类知道的越少越好。自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

开放封闭原则（Open Close Principle,简称OCP）

核心思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化

通俗来讲：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。

一句话概括:单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则 告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦　合。而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭。

内聚与耦合的概念、常见的内聚和耦合类型

起因：模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准——耦合性和内聚性。

模块

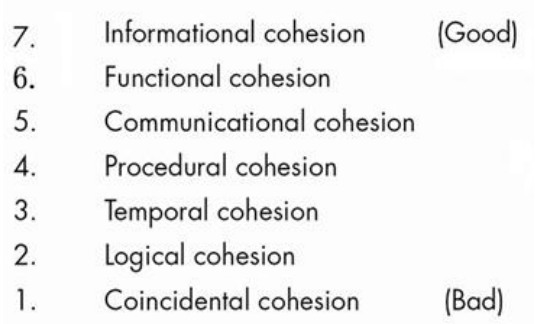
模块是由一组语句组成，并且被标识符组成的边界元素所界定。类，方法都是可以称之为一个模块。

内聚与耦合

内聚是指一个模块内的交互程度，耦合是指模块间的交互程度。我们需要尽力做到高内聚低耦合。

内聚：

内聚分为如下几类：



1）**偶然内聚**：一个模块里各个成分之间没有什么关系，就是很随意的拼凑了在了一起，被封装成了一个模块。

例子：

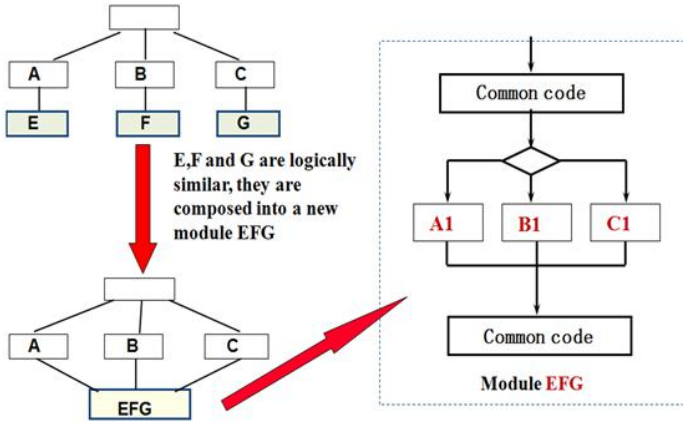
```
Public void funA(){
    If(a>b)
        c = a − b;
    System.out.prinltn(“今天天气真不错”);
}
```

缺点：1.可读性差（一会这个一会那个多乱呀）

2.可维护性差（这种一个功能干好多没关系的事情，维护是不可能维护的2333）

解决：这个解决办法十分的简单，就是将那些不相关的成分，都分别拆解开来形成各自的模块，每个模块都只是执行一个任务。

2）逻辑内聚：几个逻辑上相关的功能被封装到同一个模块里面，然后由调用函数传入控制的参数来确定调用哪个功能。



缺点：1.接口不容易理解，因为传入控制参数至就会添加参数说明否则谁知道参数。

2.修改起来不好办，如果添加或者删除逻辑功能会麻烦。

3.增加耦合度

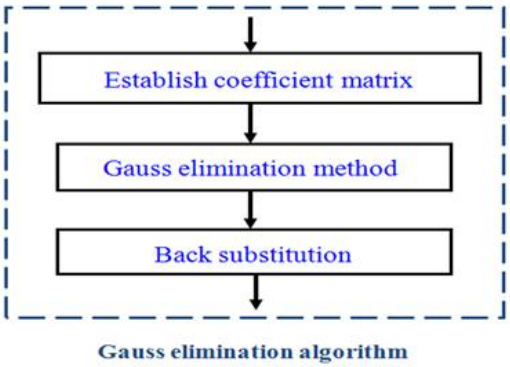
解决：把那几个逻辑相关的拆解成独立的模块。

3）时间内聚：如果一些功能仅仅是因为在同一时间执行，仅仅是由于时间关系就被封装到了一个模块。如一些初始化模块。

缺点：1.成分间的关系不强，但与其它模块相关度很高。

4）过程内聚：把一系列的过程行为放到一起形成一个模块。

例子：

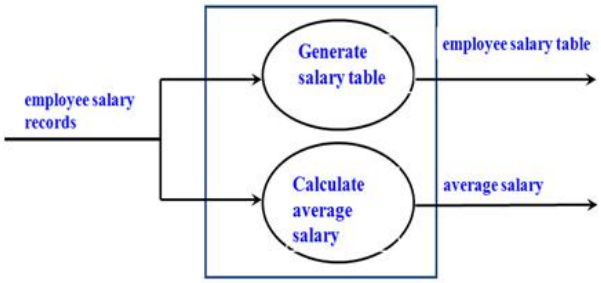


Gauss elimination algorithm

缺点：模块复用性低，多个行为活动封装到一起，别的函数要是用到还要重新写，导致复用性差。

解决：还是拆开。

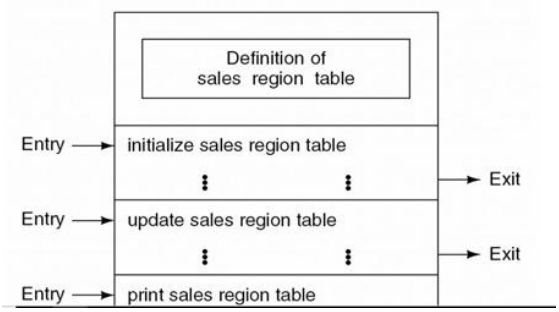
5）**通信内聚**：因为行为活动使用数据参数相同，所以就将他们封装到一个模块。
例子：



缺点：不可复用

6） **功能内聚**：每个模块仅仅是完成一个行为互动，这也体现了单一职责。
例如：判断大小，计算加法
好处： 1.复用性高
2.错误隔离

7）**信息内聚**：一个模块它可以执行很多的行为，但是每个行为都有自己入口，每个行为与其他行为独立，并且都在统一数据结构上执行。



耦合

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

1）**内容耦合**：一个模块不经调用直接引用另一个模块的内容。
例子：

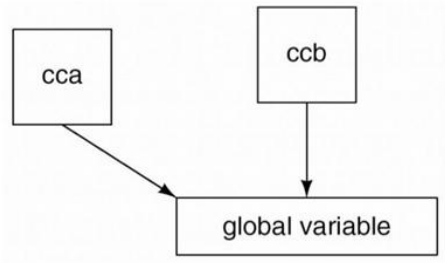
```
public class Product {
    public float unitPrice;
    ……
}

public class Order {
    private Product myProduct=new Product();
    public void setItem() {
        myProduct.unitPrice = 100.0;
    }
}
```

缺点：这种耦合标明，一个模块与另一个模块联系十分紧密，如果另一个模块改变，这个模块必定收到很大影响，以至于无法正常使用。

解决：

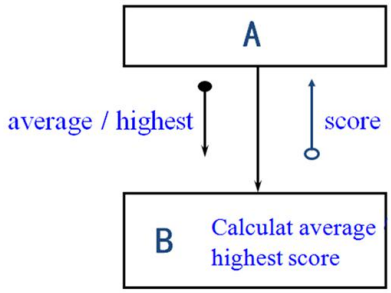
```
public class Product {
    private float unitPrice;
    public void setUnitPrice(float pUnitPrice){ … }
}

public class Order {
        private Product myProduct=new Product();
        public void setItem() {
            myProduct.setUnitPrice (100.0);
        }
}
```

2）**公共耦合**：多个模块共同引用一个全局数据



缺点：全局变量修改会导致引用模块都收影响，不易于维护。

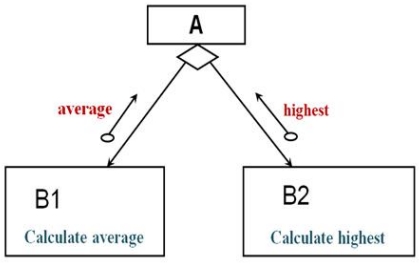
3）**控制耦合**：一个模块通过一个控制参数来控制另外一个模块，它与逻辑内聚有关系。



缺点：1.模块见依赖不独立

2.复用性低

解决：取消控制参数，将B内不同活动拆解开，由A模块进行判断调用哪个模块，而非传递参数到B来判断执行哪个。



4）**标记耦合**：一个数据结构作为参数进行传递，但是被调用的模块只是使用了数据结构中的一部分内容。

例子：

```
function  printFirstStudent(Student[ ]){
   ……
   printf (“The first student is ” + Student[0]);
   ……
}
```

缺点：1.被调用函数必须清楚参数的数据结构，并按照结构的要求进行操作数据，
2.难于理解

5）**数据耦合**：模块间传递基本数据类型或者是所有元素都被使用的数据结构进行调用。

缺点：没啥缺点

优点：可维护性高，没啥缺点233.

四、软件测试

软件测试及测试用例的概念

软件测试：描述一种用来促进鉴定软件的正确性、完整性、安全性和质量的过程。

换句话说，**软件测试是一种实际输出与预期输出之间的审核或者比较过程。**

软件测试的经典定义是：在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。

测试用例

测试用例时为了实施测试而向被测试的系统提供的一组集合，包括：**测试环境、操作步骤、测试数据、预期结果**等

好的测试用例是一个不熟悉业务的人也能依据用例来很快地进行测试；

测试：向被测试的程序输入的一组集合

测试策略

单元测试

单元测试侧重于软件设计的最小单元（软件构建或模块）的验证工作。利用构件级设计描述作为指南，测试重要的控制路径以发现模块内的错误。测试的相对复杂度和这类测试发现的错误受到单元测试约束范围的限制。单元测试侧重于构件的内部处理逻辑和数据结构，这种类型的测试可以对多个构件并行执行。

集成测试

集成测试时构建软件体系结构的系统化技术，同时也是进行一些旨在发现与接口相关的错误的测试。其目标是利用已通过单元测试的构建建立设计中描述的程序结构

增量集成程序以小增量的方式逐步进行构建和测试，这样错误易于分离和纠正，更易于对接口进行彻底测试，而且可以运用系统化的测试方法。

- 自顶向下集成
- 自底向上集成
- 回归测试 每当加入一个新模块作为集成测试的一部分的时候，回归测试重新执行已经测试过的某些子集，以确保变更没有传播不期望的副作用。回归测试有利于保证变更不引入无意识行为或者额外的错误。
- 冒烟测试

确认测试

确认测试始于集成测试的结束，在那时已测完单个构件，软件已经组装成完整的软件包，接口错误已经被发现和改正。测试集中于用户可见的动作和用户可识别的系统输出。

通过一系列表明软件功能与软件需求相符合的测试而获得的。即对于那些最终用户显而易见的错误。

系统测试

- 恢复测试 通过各种方式强制的让系统发生故障，并验证其能适当恢复
- 安全测试 验证建立在系统内的保护机制是否能够实际保护系统不受非法入侵
- 压力测试 要求以一种非正常的数量，频率或容量的方式执行系统
- 性能测试 用来测试软件在集成环境中的运行性能部
- 署测试 在软件将要运行的每一种环境中运行软件

回归测试

回归测试是指修改了旧代码后，重新进行测试以确认修改没有引入新的错误或导致其他代码产生错误。自动回归测试将大幅降低系统测试、维护升级等阶段的成本。

回归测试包括两部分：函数本身的测试、其他代码的测试。

在对被修改的函数重新测试。如果函数的设计功能没有变化，直接运行函数测试就可以了。如果修改了设计功能，则要根据增减的功能点，增加或删除测试用例。另外，还要完成白盒覆盖。

函数代码的修改可能导致调用该函数的代码产生错误，所以需要测试其他代码。如果函数是私有函数并且未涉及到全局变量，应运行类测试，否则应运行工程测试。在函数列表中选择类测试或工程测试，编译运行测试工程，即可执行对其他代码的回归测试。

调试以及调试与测试的关系

调试出现在成功的测试之后，也就是说，当测试用例发现错误的时候，调试就是使错误消除的过程。

调试并不是测试，但是调试总是发生在测试之后，执行测试用例，对测试结果进行评估，期望的表现和实习表现不

一致的时候，调试的过程就开始了。

调试试图找到隐藏在症状背后的原因，从而使错误得到修正。

调试方法：

- 蛮干法 最常用但是低效
- 回溯法
- 原因排除法
- 自动调试

测试覆盖度

测试覆盖度评估是衡量阶段性软件测试执行状态的重要手段之一，来确定测试是否达到事先设定的测试任务完成的标准。测试覆盖率则是测试覆盖度评估中一种量化的表示方法，一般通过被测试的软件产品需求、功能点、测试用例数或程序代码行等来进行计算。

软件测试覆盖率常用的计算公式：

- 功能覆盖率= 至少被执行一次的测试功能点数/ 测试功能点总数 （功能点）
- 需求覆盖率= 被验证到的需求数量 /总的需求数量 （需求）
- 覆盖率= 至少被执行一次的测试用例数/ 应执行的测试用例总数 （测试用例）
- 语句覆盖率= 至少被执行一次的语句数量/ 有效的程序代码行数
- 判定覆盖率= 判定结果被评价的次数 / 判定结果总数
- 条件覆盖率= 条件操作数值至少被评价一次的数量 / 条件操作数值的总数
- 判定条件覆盖率= 条件操作数值或判定结果至少被评价一次的数量/(条件操作数值总数+判定结果总数)
- 上下文判定覆盖率= 上下文内已执行的判定分支数和/(上下文数上下文内的判定分支总数)
- 基于状态的上下文入口覆盖率= 累加每个状态内执行到的方法数/(状态数*类内方法总数)
- 分支条件组合覆盖率= 被评测到的分支条件组合数/分支条件组合数
- 路径覆盖率= 至少被执行一次的路径数/程序总路径数

白盒测试

白盒测试有时候也被称之为玻璃盒测试或者结构化测试。它利用作为构建级设计的一部分所描述的控制结构来生成 测试用例全面了解程序内部逻辑结构、对所有逻辑路径进行测试。"白盒"法是穷举路径测试。在使用这一方案时，测试者必须检查程序的内部结构

利用白盒测试方法导出的测试用例可以：

1. 保证一个模块中的所有路径至少被执行一次
2. 对所有的逻辑判定均需测试取真和取假两个方面
3. 在上下便捷及可操作的范围内执行所有的循环
4. 检验内部数据结构以确保其有效性

基本路径测试

基本路径测试允许测试用例设计者设计出过程设计的逻辑复杂性测量，并以这种测量为直到来定义执行路径的基本集。执行该基本集导出的测试用例保证程序中的每一条语句至少执行一次。

独立程序路径

独立路径是任何贯穿程序、至少引入一组新处理语句或一个新条件的路径。

如何知道要找出多少路径

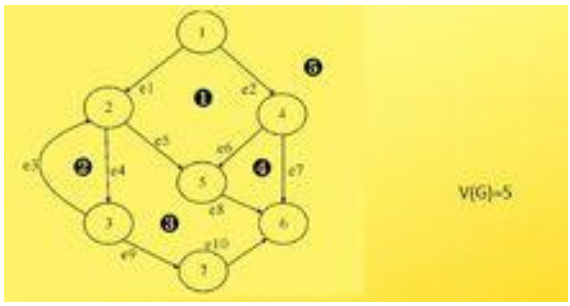
环复杂性的计算（代码质量–圈复杂度计算）

环复杂性是一种软件度量，它为程序的逻辑复杂度提供了一个量化的测度。用在基本路径测试方法的环境下时，环复杂性的值定义了程序基本集合中的独立路径数，并提供了保证所有语句至少执行一次所需测量数量的上限。

可以通过一下三种方法之一来计算环复杂度

1. 流图中域的数量与圈复杂度（环境复杂度）相对应

   $V(G)=R$　　其中R代表平面被控制流图划分成的区域数。

   针对**程序的控制流图**计算圈复杂度V(G)时，最好还是**采用第2个公式**，也即V(G)=e–n+2；

   针对**模块的控制流图**时，可以**直接统计判定节点数（第3个公式）**；

   针对**复杂的控制流图**时，使用区域计算公式**V(G)=R**更为简单

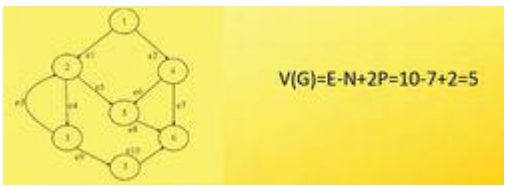
2. 对于流图G，环复杂度V(G)也可以定义如下

   $V(G)=e-n+2p$　　其中，

   e表示控制流图中边的数量，

   n表示控制流图中节点的数量，

   p图的连接组件数目（图的组件数是相连节点的最大集合）。因为控制流图都是连通的，所以p为1.

3. 对于流图G，环复杂度V(G)也可以定义如下

   $V(G)=区域数=判定节点数+1。$

因为圈复杂度所反映的是“判定条件”的数量，所以圈复杂度实际上就是等于判定节点的数量再加上1，也即控制流图的区域数。

对于多分支的CASE结构或IF-ELSEIF-ELSE结构，统计判定节点的个数时需要特别注意一点，要求必须统计全部实际的判定节点数，也即每个ELSEIF语句，

以及每个CASE语句，都应该算为一个判定节点。

圈复杂度(Cyclomatic Complexity)是一种代码复杂度的衡量标准。

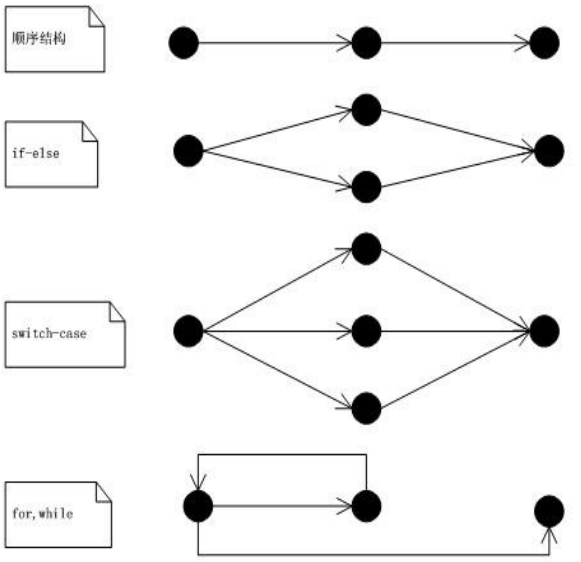
它可以用来衡量一个模块判定结构的复杂程度，数量上表现为独立现行路径条数，

也可理解为覆盖所有的可能情况最少使用的测试用例数。

圈复杂度大说明程序代码的判断逻辑复杂，可能质量低且难于测试和维护。

程序的可能错误和高的圈复杂度有着很大关系。

各判断语句的控制流图



在计算圈复杂度时，可以通过程序控制流图方便的计算出来。

通常使用的计算公式是V(G) = e − n + 2 ，

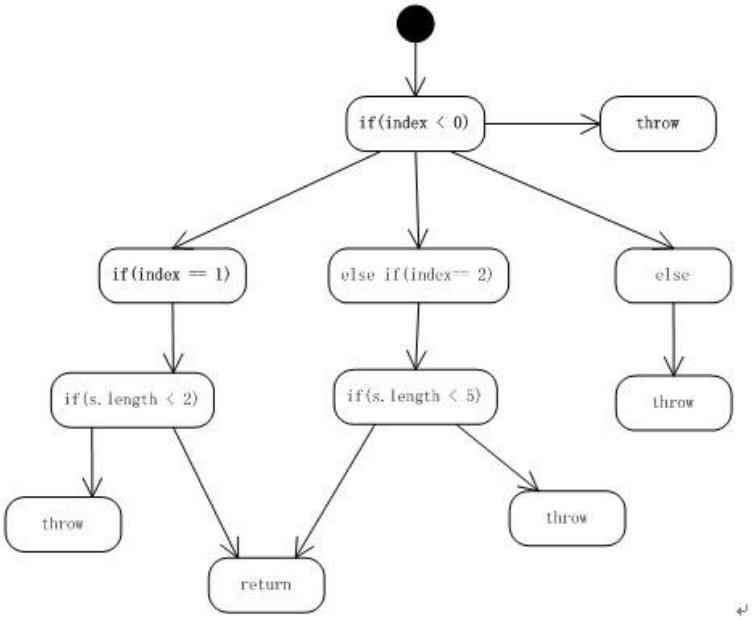
e 代表在控制流图中的边的数量（对应代码中顺序结构的部分），

n 代表在控制流图中的节点数量，**包括起点和终点**（1、所有终点只计算一次，即便有多个return或者throw；2、节点对应代码中的分支语句）。圈复杂度主要与分支语句（if、else、，switch 等）的个数成正相关

例子：

```
public String case2(int index, String string) {
    String returnString = null;
    if (index < 0) {
        throw new IndexOutOfBoundsException("exception <0 ");
    }
    if (index == 1) {
        if (string.length() < 2) {
```

```
            return string;
        }
        returnString = "returnString1";
    } else if (index == 2) {
        if (string.length() < 5) {
            return string;
        }
        returnString = "returnString2";
    } else {
        throw new IndexOutOfBoundsException("exception >2 ");
    }
    return returnString;
}
```



根据公式 V(G) = e − n + 2 = 12 − 8 + 2 = 6 。

case2的圈复杂段为6。

说明一下为什么n = 8，虽然图上的真正节点有12个，

但是其中有5个节点为throw、return，这样的节点为end节点，只能记做一个。

生成测试用路

1. 以设计或源代码为基础，画出相应的流图
2. 确定所得流图的环复杂性
3. 确定线性独立路径的基本集合
4. 准备测试用例，强制执行基本集合中的每条路径

黑盒测试

黑河测试也称为行为测试或功能测试，侧重于软件的功能需求。

黑盒测试试图发现以下类型的错误：

1. 不正确或遗漏的功能
2. 接口错误
3. 数据结构或者外部数据库访问错误
4. 行为或性能错误
5. 初始化和终止错误

   黑盒测试应用在测试的后期阶段，侧重于信息域

等价类划分

等价类划分是一种黑盒测试方法，它将程序的输入划分为若干个数据类，从中生成测试用例。理想的测试用例可以单独发现一类错误，否则在观察到一般的错误之前需要运行许多测试用例。

等价类划分的测试用例设计师基于对输入条件的等价类进行评估。

边界值分析

大量的错误发生在输入域的边界，而不是发生在输入域的中间，边界值分析选择一组测试用例检查边界值