结构化设计

需求设计

需求的分类

功能性需求：

最常见、最主要和最重要的需求

能够为用户带来业务价值的系统行为

最需要按照三个抽象层次进行展开

软件产品产生价值的基础

性能需求：

速度、容量、吞吐量、负载、实时性

5.25.11 数据需求

1. **功能需求**的补充：如果在功能需求部分明确定义了**相关的数据结构**，那么就不需要再行定义数据需求
2. 数据需求是需要在数据库、文件或者其他介质中存储的数据描述，通常包括下列内容：
   - 各个功能使用的**数据信息**；
   - 使用频率；
   - 可访问性要求；
   - **数据实体及其关系**；
   - 完整性约束；
   - **数据保持**要求。
3. 例如，连锁超市销售系统可以使用数据需求 DR1 和 DR2。
   - DR1：系统需要存储的数据实体及其关系为图 6-14 的内容。（数据实体及其关系）
   - DR2：系统需要存储 1 年内的销售记录和退货记录。（数据保持）

质量属性：

可靠性、可用性、安全性、可维护性(可修改可扩展)、可移植、易用性

对外接口：

接口的用途、接口的输入输出数据格式、命令格式、异常处理要求

约束：

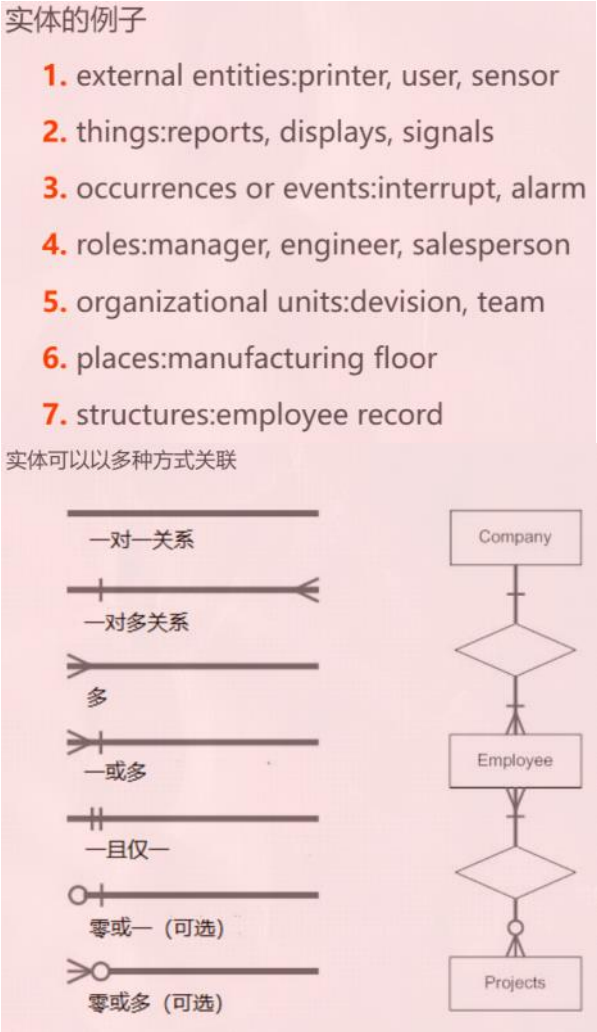
环境、问题域、商业规则

需求分析

DataFlow数据流图

注意分解时的平衡性

实体关系图 ERD - 和 数据库很像



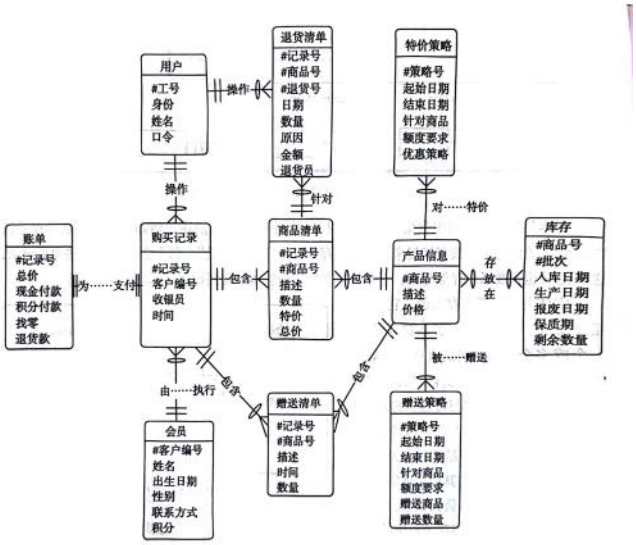


图 6-14　连锁商店管理系统的整体 ERD

概要设计

概要设计

5、概要设计的启发式准则

- 改进软件结构，提高模块独立性
- 模块规模应该适中（最好能写在一页纸上）
- 大模块分解不充分；小模块使用开销大，接口复杂。
- 尽量减少高扇出结构的数目，随着深度的增加争取更多的扇入
- 扇出过大意味着模块过分复杂，需要控制和协调过多的下级模块。一般来说，顶层扇出高，中间扇出少，低层高扇入。
- 模块的作用范围保持在该模块的控制范围内

  模块的作用范围是指该模块中一个判断所影响的所有其它模块；模块的控制范围指该模块本身以及所有直接或间接从属于它的模块。
- 力争降低模块接口的复杂程度

  模块接口的复杂性是引起软件错误的一个主要原因。接口设计应该使得信息传递简单并且与模块的功能一致。
- 设计单入口单出口的模块

  避免内容耦合，易于理解和维护。
- 模块的功能应该可以预测

  相同的输入应该有相同的输出，否则难以理解、测试和维护。

面向数据流的设计（变换流 & 事务流）

变换：

- 第1步　复查基本系统模型。
- 第2步　复查并精化数据流图。
- 第3步　确定数据流图具有变换特性还是事务特性。
- 第4步　确定输入流和输出流的边界，从而孤立出变换中心。
- 第5步　完成“第一级分解”。

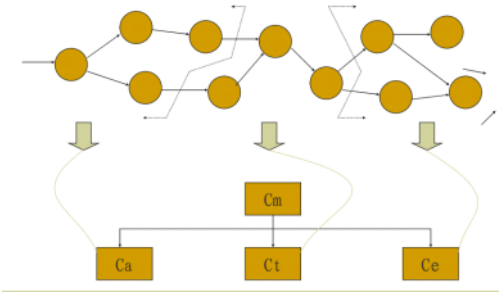
软件结构代表对控制的自顶向下的分配，所谓分解就是分配控制的过程。

对于变换流，数据图将被映射成一个特殊的软件结构，这个结构控制输入、变换和输出信息等处理过程：位于软件结构最顶层的控制模块Cm协调下述从属的控制功能：

（1）输入信息处理控制模块Ca，协调对所有输入数据的接收；

（2）变换中心控制模块Ct，管理对内部形式的数据的所有操作；

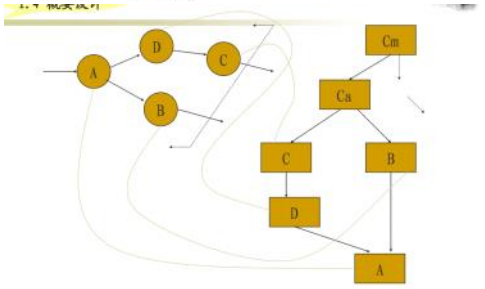
（3）输出信息控制模块Ce，协调输出信息的产生过程。



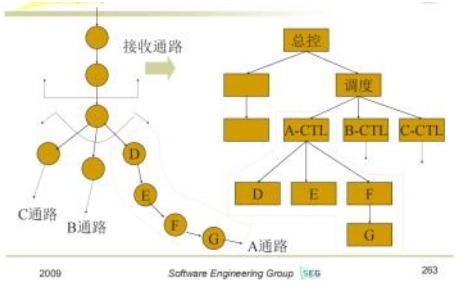
- 第6步 完成“第二级分解”。

  把数据流图中的每一个处理映射成软件结构中一个适当的模块：从变换中心的边界开始沿着输入通路向外移动，把输入通路中每个处理映射成软件结构中Ca控制下的一个低层模块；然后沿输出通路向外移动，把输出通路中每个处理映射成直接或间接受Ce控制的一个低层模块；最后把变换中心内的每个处理映射成受Ct控制的一个模块。

- 第7步 使用设计度量和启发式规则对得到的软件结构进一步精化。

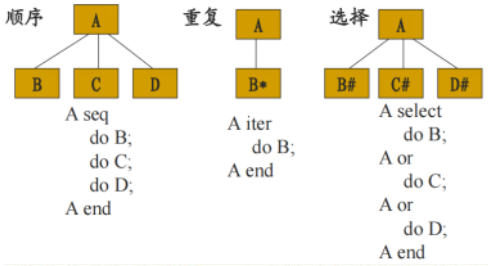


事务：



面向数据结构的设计 Jackson方法

Jackson图（数据结构符号）：



便于表示层次结构，而且是对结构进行自顶向下分解的有力工具；

形象直观，可读性好；

既能表示数据结构，又能表示程序结构。

步骤：

（1）分析并确定输入数据和输出数据的逻辑结构，并用Jackson图描述这些数据结构；

（2）找出输入数据和输出数据结构中有对应关系的数据单元。所谓对应关系是指有直接的因果关系，在程序中可以同时处理的数据单元（对于重复出现的数据单元必须重复的次数相同才可能有对应关系）；

（3）用下述三条规则从描述数据结构的Jackson图导出描述程序结构的Jackson图：

第一，为每对有对应关系的数据单元，按照它们在数据结构图中的层次在程序结构图的相应层次画一个处理框（注意，若这对数据单元在输入数据结构和输出数据结构中所处的层次不同，则和它们对应的处理框在程序结构图中所处的层次与它们之中在数据结构图中层次低的那个对应）；

第二，根据输入数据结构中剩余的每个数据单元所处的层次，在程序结构图中的相应层次分别为它们画上对应的处理框；

第三，根据输出数据结构中剩余的每个数据单元所处的层次，在程序结构图中的相应层次分别为它们画上对应的处理框。

（4）列出所有操作和条件（包括分支条件和循环结束条件），并且把它们分配到程序结构图的适当位置。

（5）用伪码表示程序。

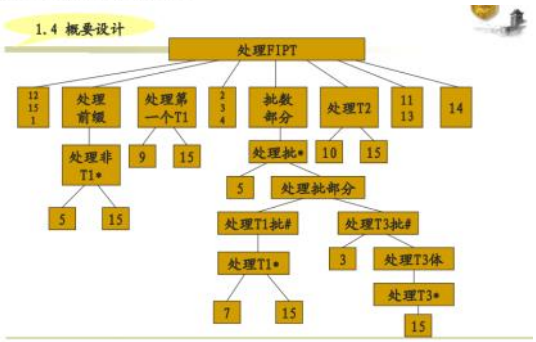
例子：

例： 输入一个文件FIPT，此文件只包含三种记录类型T1、T2和T3，现在要对该文件作如下处理：

（1）统计出现的第一个T1类型的记录前的记录总数（计数A）；
（2）显示第一个T1类型的记录；
（3）显示最后一个记录，最后一个记录是在第一个T1类型的记录后的第一个T2类型的记录；
（4）计算第一个T1类型的记录后的记录批数（一批记录指一串连续的T1类型的记录或一串连续的T3类型的记录（（计数B）；
（5）统计在第一个T1类型的记录后出现的T1类型记录的总数(计数C)；
（6）计算在第一个T1类型的记录后的T3类型记录的批数（计数D）。

列出所有的操作：

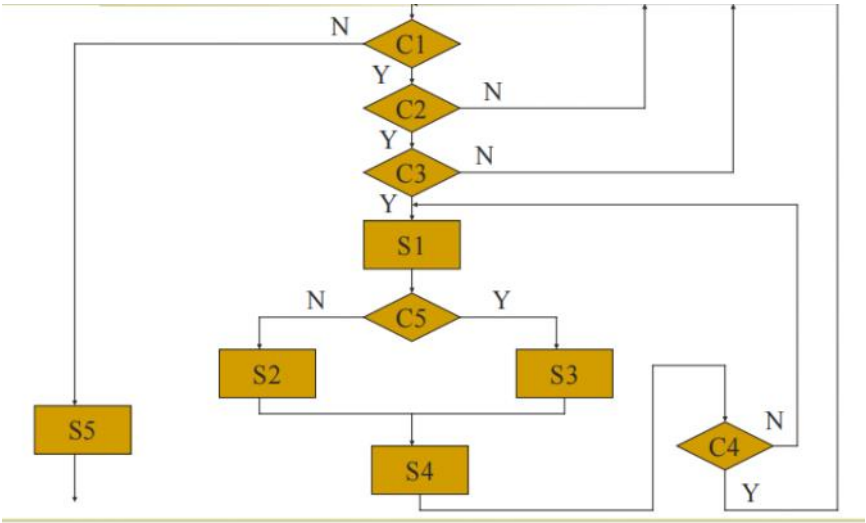
（1）CA：=0 （2）CB：=0 （3）CC：=0（4）CD：=0
（5）CA：=CA+1 （6）CB：=CB+1 （7）CC：=CC+1
（8）CD：=CD+1 （9）显示第一个T1记录
（10）显示最后一个T1记录
（11）显示所有计数器的内容
（12）打开FIPT文件
（13）关闭FIPT文件
（14）终止运行
（15）读FIPT文件记录



详细设计

详细设计

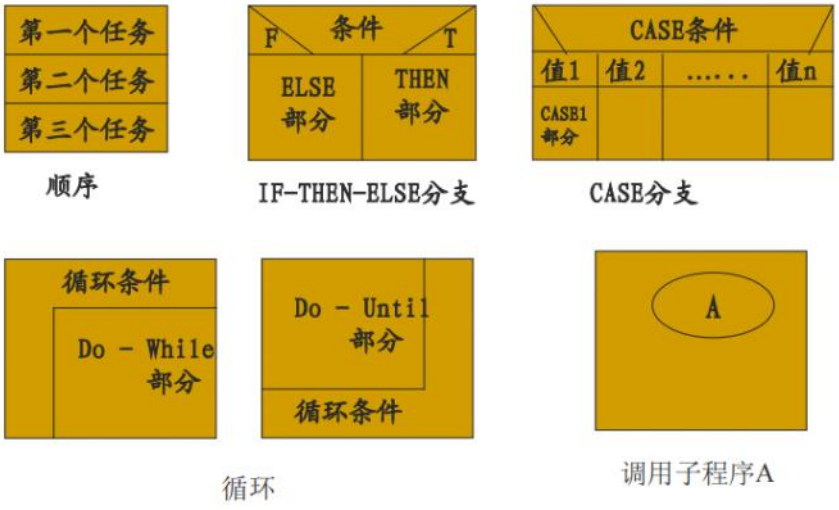
流程图：



流程图的主要缺点：

- 流程图本质上不是逐步求精的好工具，它诱使程序员过早地考虑程序的控制流程，而不去考虑程序的全局结构。
- 流程图中用箭头代表控制流，因此程序员不受任何约束，可以完全不顾结构程序设计的精神，随意转移控制。
- 流程图不易表示数据结构。

方块图：



研制方块图的目的是：既要制定一种图形工具，又不允许它违反结构化原则。

方块图具有以下特点：

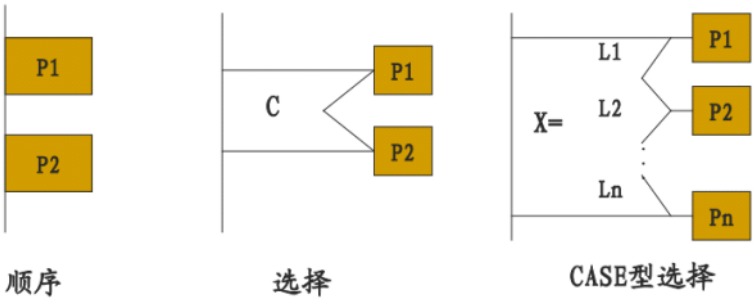
（1）功能域（即某一具体构造的功能范围）有明确的规定，并且很只观地从图形表示中看出来；

（2）想随意分支或转移是不可能的；

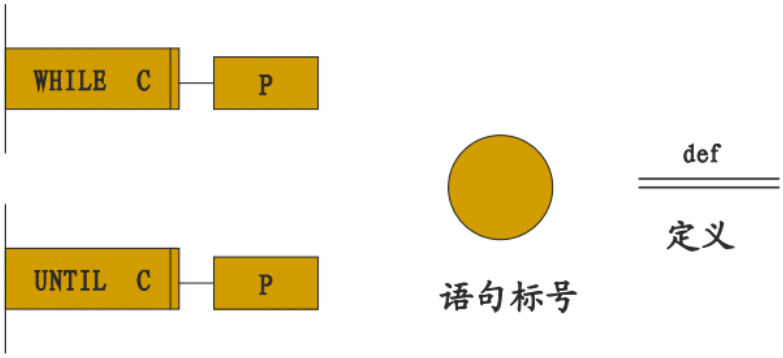
（3）局部数据和全程数据的作用域可以很容易确定；

（4）容易表示出递归结构。

P-A图



顺序　　选择　　CASE型选择



循环

- PAD图的特点：
  - 使用表示结构化控制结构的PAD符号所设计出的程序必然是结构化程序。
  - PAD图所描述的程序结构十分清晰，图中最左面的竖线是程序的主线，即第一层结构，随着程序层次的增加，PAD图逐渐向右延伸，每增加一个层次，图形向右扩展一条竖线，PAD图中的竖线的总条数就是程序的层次数。
  - 用PAD图表现程序逻辑，易读、易懂、易记，PAD图是二维树形结构的图形，程序从图中最左竖线上端的结点开始执行，自上而下，从左向右顺序执行，遍历所有结点。
  - 容易将PAD图转换成高级语言源程序，这种转换可用软件工具自动完成。
  - 既可以用于表示程序逻辑，也可用于描述数据结构。
  - PAD图的符号具有支持自顶向下、逐步求精方法的作用。开始时设计者可以定义一个抽象的程序，随着设计工作的深入而用def符号逐步增加细节，直至完成详细设计。

模块化

模块化

分解：横向 各子系统

抽象：纵向 接口+实现

实现模块化的手段

- **抽象**：抽出事物的本质特性而暂时不考虑它们的 细节，聚焦本质，降低认知复杂度。
- **信息隐蔽**：应该这样设计和确定模块，使得一个 模块内包含的信息（过程和数据）对于不需要这些信息的模块来说，是不可访问的。
- 抽象就是接口，隐藏就是实现上的设计决策
- 每个模块都承担一定的责任，对外是契约，契约之下隐藏设计决策和决策细节。

- 模块独立性：

  模块独立是指开发具有独立功能而且和其它模块之间没有过多的相互作用的模块。

- 模块独立的意义：

  功能分割，简化接口，易于多人合作开发同一软件；

独立的模块易于测试和维护

类　型	耦 合 性	解　释	例　子
内容耦合	最高	一个模块直接修改或者依赖于另一个模块的内容	程序跳转 GOTO；某些语言机制支持直接更改另一个模块的代码；改变另一个模块的内部数据
公共耦合	↑	模块之间共享全局的数据	全局变量
重复耦合	↕	模块之间有同样逻辑的重复代码	逻辑代码被复制到两个地方
控制耦合	↕	一个模块给另一个模块传递控制信息	传递“显示星期天”。传递模块和接收模块必须共享一个共同的内部结构和逻辑
印记耦合	↓	共享一个数据结构，但是却只用了其中一部分	传递了整个记录给另一个模块，另一个模块却只需要一个字段
数据耦合	最低	两个模块的所有参数是同类型的数据项	传递一个整数给一个计算平方根的函数

前三个不可接受，最后一个最理想

类　型	内聚性	解　释	例　子
偶然内聚	最低	模块执行多个完全不相关的操作	把下列方法放在一个模块中：修车、烤面包、遛狗、看电影
逻辑内聚	↑	模块执行一系列相关操作，每个操作的调用由其他模块来决定	把下列方法放在一个模块中：开车去、坐火车去、坐飞机去
时间内聚	↑	模块执行一系列与时间有关的操作	把下列方法放在一个模块中：起床、刷牙、洗脸、吃早餐
过程内聚	↑	模块执行一些与步骤顺序有关的操作	把下列方法放在一个模块中：守门员传球给后卫、后卫传球给中场球员、中场球员传球给前锋、前锋射门
通信内聚	↑	模块执行一系列与步骤有关的操作，并且这些操作在相同的数据上进行	把下列方法放在一个模块中：查书的名字、查书的作者、查书的出版商
功能内聚	↑	模块只执行一个操作或达到一个单一目的	下列内容都作为独立模块：计算平方根、决定最短路径、压缩数据
信息内聚	最高	模块进行许多操作，各个都有各自的入口点，每个操作的代码相对独立，而且所有操作都在相同的数据结构上完成	比如数据结构中的栈，它包含相应的数据和操作。所有的操作都是针对相同的数据结构

信息隐藏

模块的主要秘密：

- 主要秘密描述的是这个模块所要实现的用户需求。是设计者对用户需求的实现的一次职责分配。有了这个描述以后，我们可以利用它检查我们是否完成所有的用户需求，还可以利用它和需求优先级来决定开发的次序。

模块的次要秘密：

- 次要秘密描述的是这个模块在实现职责时候所涉及的具体的实现细节.包括数据结构，算法，硬件平台等信息。
- **模块的角色：**
  - **描述了独立的模块在整个系统中所承担的角色，所起的作用。以及与哪些模块有相关联的关系。**
- **模块的对外接口：**
  - **模块提供给别的模块的接口。**

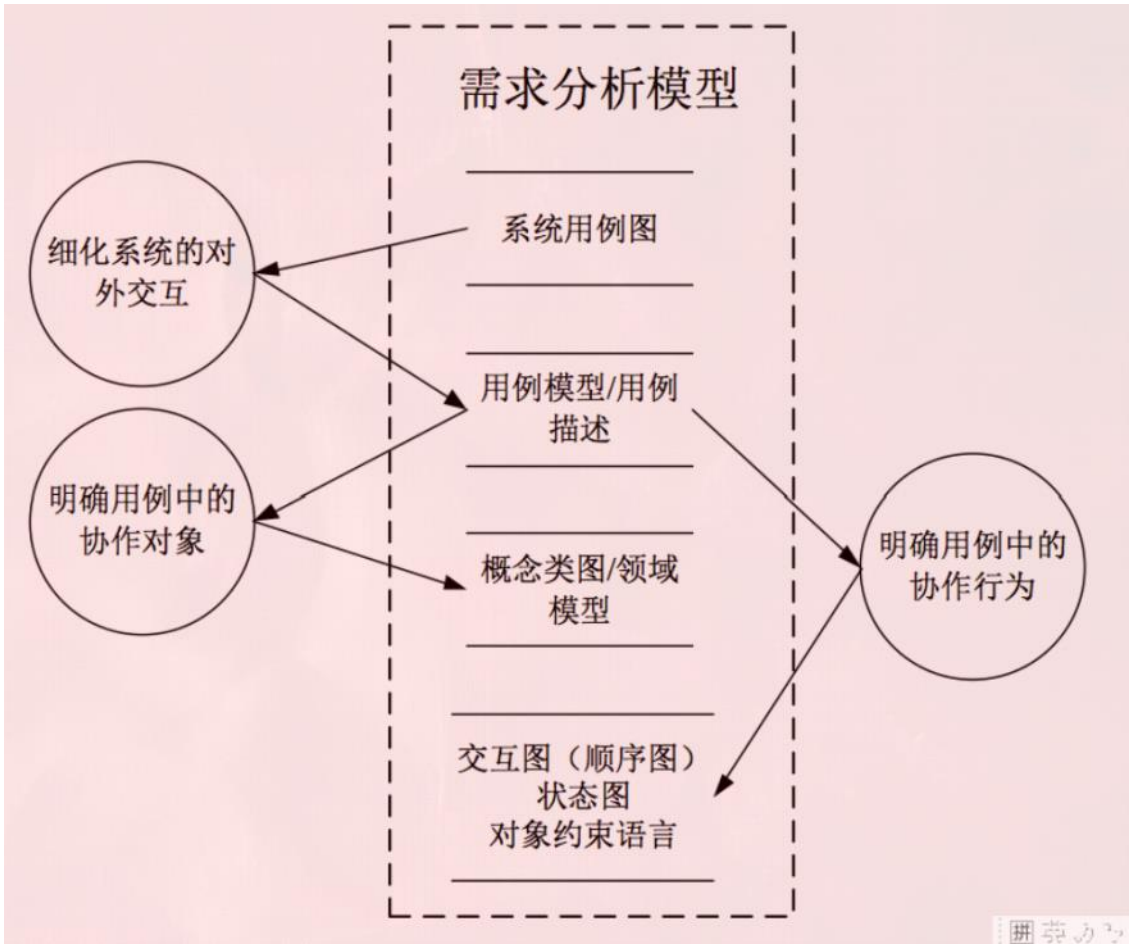
主　题	说　明
主要秘密	实现对字符串的循环位移功能
次要秘密	1）循环位移算法 2）循环位移后字符的存储格式
角色	1）自身由主控对象创建 2）调用 LineStorage 对象的方法来访问字符串 3）完成循环位移之后，提供位移后字符的访问接口给 Alphabertizer 对象以帮助其完成字母排序

对外接口	`public class CircularShifter{`
	`  public void setup(LineStorage lines);`
	`  public char getChar(int position, int word, int line);`
	`  public int getCharCount(int word, int line);`
	`  public String getWord(int word, int line);`
	`  public int getWordCount(int line);`
	`  public String[] getLine(int line);`
	`  public String getLineAsString(int line);`
	`  public int getLineCount();`
	`}`

面向对象设计

作业中已经完成的部分：需求分析模型

需求分析



用例图

调整和细化

合适的粒度在于，用例对应的是一个业务事件，由一个用户发起，并在一个持续的时间内完成，可以增加业务价值的任务。

例如：

特价策略和赠送策略可以合并为一个销售策略

会员管理可以细化为发展会员和礼品赠送

但是注意

不要细化过小

不要将同一个业务目标细化

不要将没有业务价值的内容作为用例，如登录、验证、连接数据库

概念类图

概念类图和设计类图的不同点:

关注系统与外界的交互，而不是软件系统的内部构造机制

4.13.31 概念类图生成的步骤（总结）

1. 识别候选类（名词分析法）
2. **确定概念类**（看是否满足既有状态又有行为）
    1. 既需要维持一定的状态，又需要依据状态表现一定的行为：确定为一个概念类
    2. 如只需要维护状态，不需要表现行为：其他概念类的属性
    3. 不需要维护状态，却需要表现行为：首先重新审视需求是否有遗漏，因为没有状态支持的对象无法表现行为；如果确定没有需求的遗漏，就需要剔除该候选类，并将行为转交给具备状态支持能力的其他概念类
    4. 既不需要维护状态，又不需要表现行为：应该被完全剔除
3. 识别关联（文本中提取出"名词+动词+名词"的结构）：第一标准是满足需求的要求，第二标准是现实状况
4. 识别重要属性：协作的必要信息，通过分析用例的描述，补充问题域信息发现。

5.19 概念类图有助于发现

1. 部分信息的使用不准确
    - 例如步骤 2 中输入的是商品标识,而不是商品,第 5 步显示的已输入商品列表信息和总价。
2. 部分信息不明确
    - 例如会员信息、商品信息、商品列表信息、赠品信息、更新的数据、收据等等各自的详细内容并没有描述。
3. 遗漏了重要内容
    - 例如总价的计算需要使用商品特价策略和总额特价策略,赠品的计算需要使用商品赠送策略和总额赠送策略。

1. 收银员输入会员编号；
2. 系统显示会员信息；
3. 收银员输入商品；
4. 系统显示输入商品的信息；
5. 系统显示所有已输入商品的信息；
收银员重复 3-5 步，直至完成所有输入
6. 收银员结束商品输入；
7. 系统显示总价和赠品信息；
8. 收银员请求顾客付款；
9. 顾客支付，收银员输入支付数额；
10. 系统显示应找零数额，收银员找零；
11. 收银员结束销售；
12. 系统更新数据，并打印收据。

顺序图/系统顺序图

系统顺序图有助于发现交互性的缺失

状态机

状态图有助于发现页面的跳转

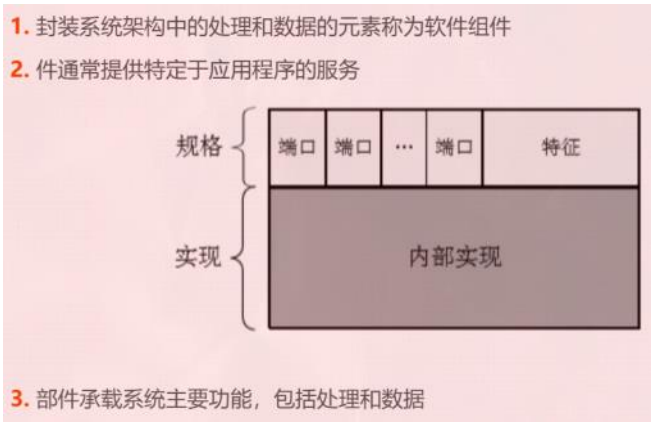
体系结构&体系风格

（一）结构

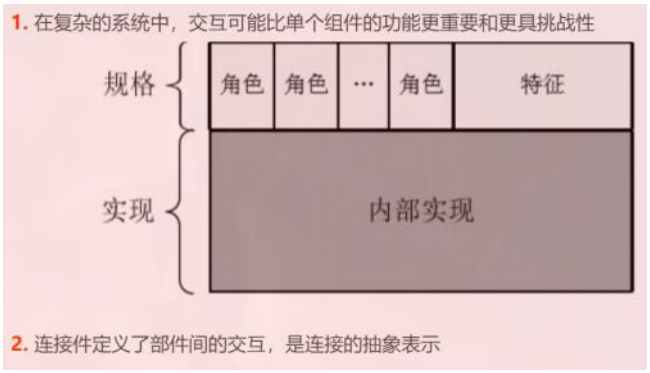
连接件是一个与部件平等的单位。

部件与连接件是比类、模块等软件单位更高层次的抽象。

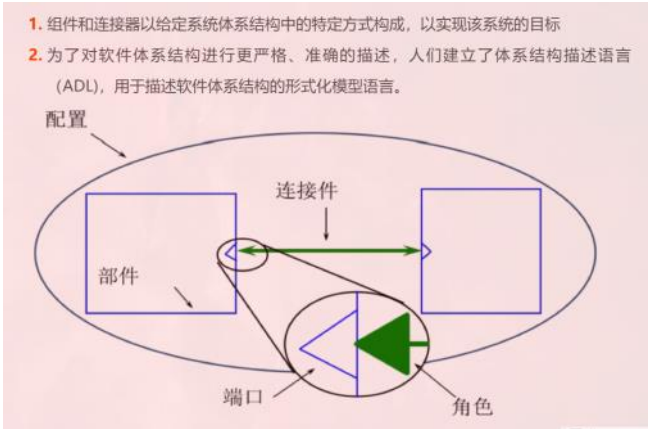
1. 部件

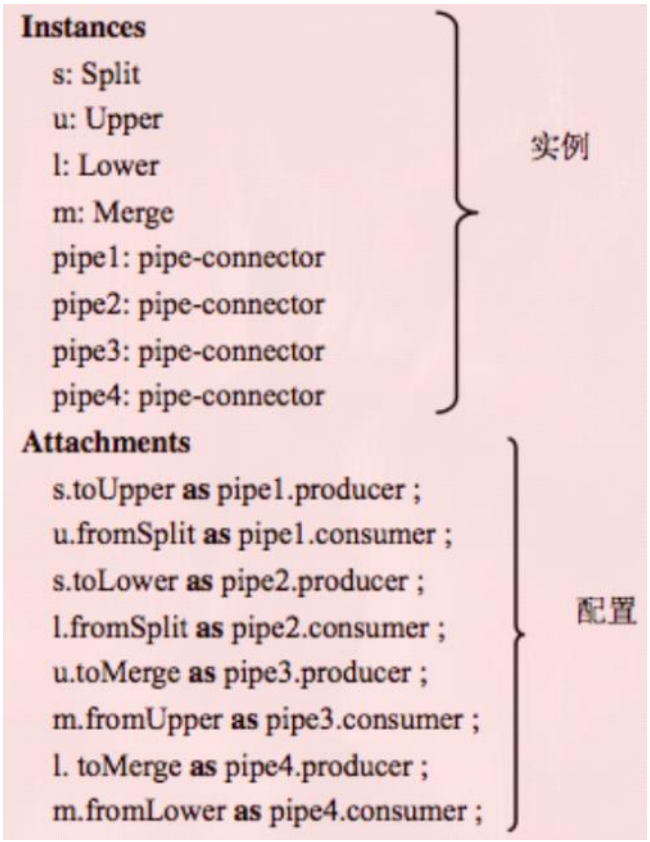


2. 连接件



3. 配置





（二）风格

1. 主程序/子程序

设计决策&约束：

基于“声明-使用”，程序调用关系建立连接件

上层使用下层，不可逆

单线程执行

"深搜"式转移控制权

子程序“部件”仍然是粗粒的块，部件内部实现结构化/OO都可。

优缺点：

流程清晰、分而治之、易于理解，控制性强；

强耦合、难修改复用，限制部件交流、隐含数据共享造成公共耦合、进而可能破坏“正确性”。

场景：

有限制的编程语言

2. OO式

设计决策&约束：

基于"信息内聚"建立对象部件

基于方法调用机制，建立连接件

对象内部维护数据一致性和完整性，外部平级

对象部件模块内部实现结构化/OO都可

优缺点：

内部可修改，结构组织易开发、易理解、易复用；

接口耦合、标识耦合、有副作用

主要场景：

基于数据信息分解和组织

3. 分层

经典：展示层、业务逻辑层、数据层

决策&约束：禁止跨层调用

优缺点：

机制清晰、易于理解，支持并行开发，可复用性内部可修改性好；

层间交互协议难修改，性能损失，层次数目和粒度难确定。

场景：

没有强实时性要求

主要功能能分解

4. MVC 模型视图控制 Web风格

决策&约束：

业务逻辑、表现、控制的抽象

优缺点：

易开发，视图与控制可修改性，Web风格；

复杂、模型难改。

体系设计

一般都是选择分层，画包图

主要理解客户端+服务端结构

客户端：展示层、业务逻辑层、网络模块

服务端：网络模块、数据层

包的设计原则：

1. 重用发布等价原则（REP）：重用的粒度就是发布的粒度
2. 共同封闭原则（CCP）：包中所有类对于同一类性质的变化应该是共同封闭的，一个变化若对一个包产生影响，则对该包中的所有类产生影响，而对于其他包不造成任何影响。
3. 共同重用原理（CRP）：一个包中的所有类应该是能够共同重用的。
4. 无环依赖原则（ADP）：在包的依赖关系图中不能存在环。
5. 稳定依赖原则（SDP）：朝着稳定的方向进行依赖
6. 稳定抽象原则（SAP）：包的抽象程度应该和其稳定程度一致
7. 前三条描述的是依赖性，后三条描述的是耦合性

5.6 Common Closure Principle (CCP) 共同封闭原则

1. **一起修改的类应该放在一起**
2. 最小化修改对程序员的影响
3. 当需要更改时，对程序员有利
4. 如果更改由于编译和链接时间以及重新验证而影响了尽可能少的软件包

5.7 Common Reuse Principle (CRP) 共同重用原则

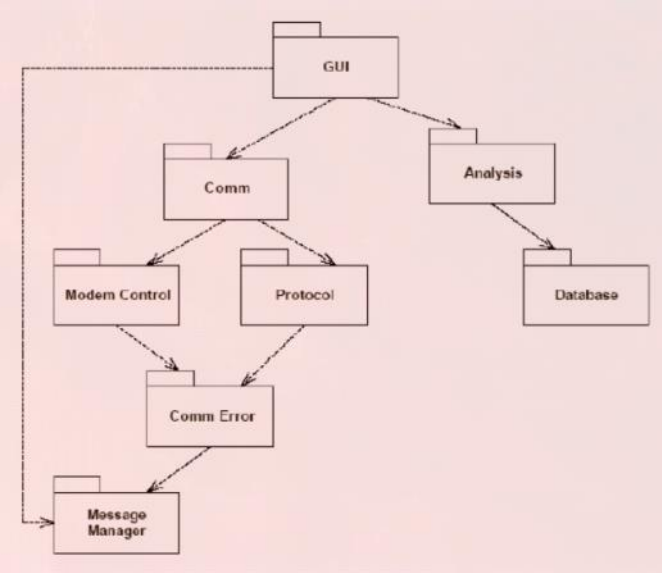
1. **一起被重用的应该在一起**
2. 包应重点突出，用户应使用包中的所有类
3. 总结
    1. 根据常见重用对类进行分组：避免给用户不必要的依赖
    2. 遵循 CRP 通常会导致软件包拆分：获得更多，更小，更专注的包
    3. 减少重新使用者的工作

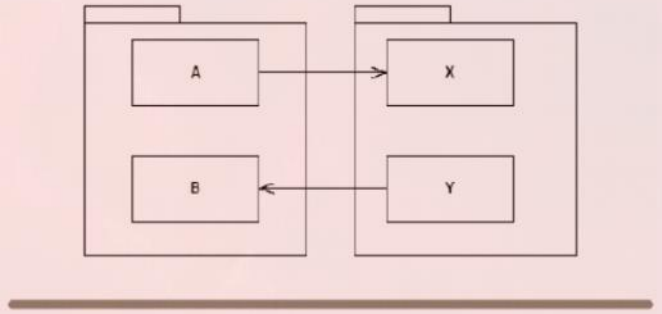
5.8 共同封闭原则和共同重用的原则的折衷

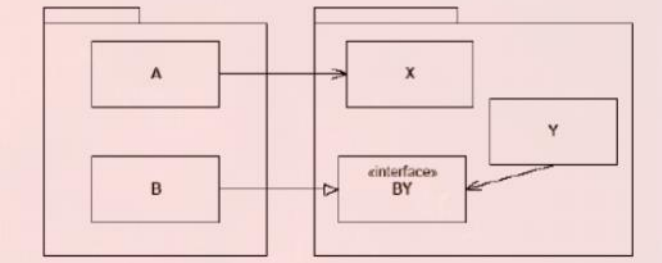
1. CCP 和 CRP 原则是**互斥**的，即不能同时满足它们。
2. CRP 使重用者的生活更加轻松，而 CCP 使维护者的生活更加轻松。
3. CCP 致力于使包装尽可能大，而 CRP 则尝试使包装很小。
4. 在项目的早期，架构师可以设置包结构，使 CCP 占主导地位，并协助开发和维护。后来，随着体系结构的稳定，架构师可以重构程序包结构，以使**外部重用程序的 CRP 最大化。**
5. 也就是说在软件开发的不同阶段 CCP 和 CRP 的重视重用程度时不同的，要根据项目进展来进行不同程度的学习。

5.11 Stable Dependencies Principle (SDP) 稳定依赖原则

1. **依赖应当指向一个稳定的方向（更改他所需要付出的努力）**
2. 稳定性：对应于更换包装所需的时间
3. 稳定的软件包：项目内难以更改
4. 稳定性可以量化







5.12 包的稳定性度量

1. 一种方法是**计算进、出该包的依赖关系的数目。**可以使用这些数值来计算该包的位置稳定性（positional stability)。
2. （Ca）入耦合度（Afferent Coupling）：指处于该包的外部并依赖于该包内的类的类的数目。
3. （Ce）出耦合度（Efferent Coupling）：指处于该包的内部并依赖于该包外的类的类的数目。
4. （不稳定性 I）I = Ce / (Ca + Ce)

5.13 Stable Abstractions Principle (SAP) 稳定抽象原则

1. **稳定的包应该是抽象的包（接口）**
2. **不稳定的包应该是具体的包**
3. 稳定的包装包含高层的设计。
4. 使它们成为抽象可以打开它们进行扩展，但可以关闭它们进行修改（OCP)。
5. 稳定的难以更改的包装中保留了一些灵活性。

5.13.7 抽象性度量

1. 包的抽象性用抽象类的数目和包中所有类的数目进行计算。
2. 假如说包中类的总数是 Nc，抽象类的数目是 Na,那么抽象度 A = Na / Nc

基于此，要会画包的逻辑设计 & 开发包图

5.14 包设计的过程

1. 迭代的过程：先用 CCP 原则对把可能一同变化的类组织成包进行发布
2. 随着系统的不断增长,我们开始关注创建可重用的元素,于是开始使用 CRP 和 REP 来指导包的组合。
3. 后使用 ADP、SDP、SAP 对包图进行度量，去掉不好的依赖。（修改设计）

详细设计

职责->静态的设计模型类图

协作->动态的顺序图&对象状态图

（一）静态的设计模型类图

GRASP Patterns

Low Coupling

High Cohesion

Information Expert

Creator

Controller

4.18.11.1 拇指原则

1. 当存在替代设计选择时，请仔细研究替代方案的**凝聚力和耦合含义**，并可能对替代方案的未来发展压力。
2. 选择具有良好内聚性，耦合性和稳定性的替代方案。

4.18.11.2 信息专家

1. 问题：在面向对象设计中分配职责的最基本原则是什么？
2. 解决方案：将具有完成任务所必需的信息的班级分配给班级。
3. 维护信息封装
4. 促进低耦合
5. 促进高内聚类

（二）顺序图&对象状态图

顺序图：“消息流”

状态图：对象的“状态”

（三）明确创建者

4.19.15.3 创建者模式

1. 问题：谁负责创建某个类的新实例？
2. 解决方案：根据潜在的创建者类与要实例化的类之间的关系，确定哪个类应创建类的实例。
3. 问题：谁负责创建对象？
4. 回答：如果有以下情况，则由创建者分配 B 类创建 A 类实例的职责：
    1. B 聚集了 A 对象
    2. B 包含了 A 对象
    3. B 记录了 A 的实例
    4. B 要经常使用 A 对象
    5. 当 A 的实例被创建，B 具有传递给 A 的初始化数据（也就是 B 是创建 A 的实例这项任务的信息专家）
    6. 在有选择的地方，更喜欢 B 聚合或包含 A 对象

表 12-4　对象创建者

优　先	场　景	创建地点	创建时机	备　注
高 ↕ 低	唯一属于某个整体的密不可分的一部分（组合关系）	整体对象的属性定义和构造方法	整体对象的创建	例如，销售的业务逻辑对象由销售页面对象创建
	被某一对象记录和管理（单向被关联）	关联对象的方法	业务方法的执行中对象的生命周期起始点	例如，连接池管理对象需要负责创建连接池对象
	创建所需的数据被某个对象所持有	持有数据的对象的业务方法	业务方法的执行中	也可以考虑在此持有数据对象不了解创建时机时，由别的对象创建，由它来初始化
	被某个整体包含（聚合关系）	整体对象的业务方法（非构造方法）	业务方法的执行中	如果某个对象有多个关联，优先选择聚合关联的整体对象。如果某个对象有多个聚合关联的整体对象，则考查整体对象的高内聚和低耦合来决定由谁创建
	其他			通过高内聚和低耦合来决定由谁创建

第一个（组合关系）

第二个（单向被关联）：比如访问数据库，你要访问的时候，我就给一个访问对象来使用，不用的时候归还就行。

第三个（持有必要数据）：根据业务的情况决定什么时候被创建，有时候 B 可以创建但是不知道什么时机来创建，如果 C 知道，那么我们可能让 C 创建对象，然后 B 进行初始化

第四个（聚合关系）：关系比较多，要看时机等什么时候合适

（四）控制器

4.19.17 选择合适的控制风格（重要）

1. 集中式控制风格
2. 委托式控制风格
3. 分散式控制风格

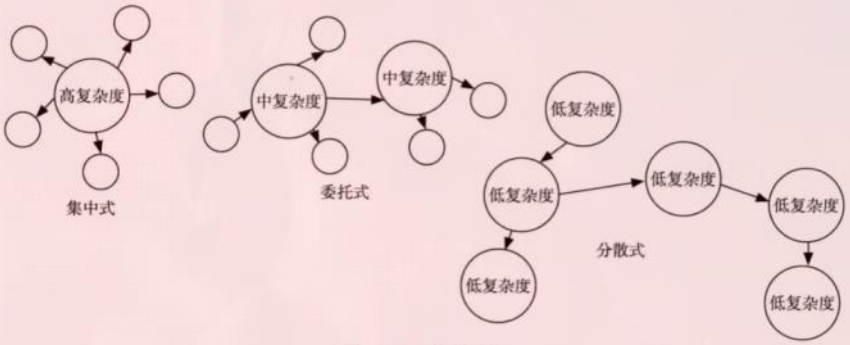


图 12-13　控制风格

1. 系统行为的逻辑在对象（组件）网络之间分布的方式。
2. 分散的：系统行为的逻辑通过对象网络“**广泛传播**”
3. 集中式：一个额外的控制器记录系统行为的所有逻辑。

模块化

信息隐藏

设计模式

代码设计

可靠的代码

1. 契约式设计(断言式设计)
   a. 异常方式，if(not satisfy preCon) throw error…
   b. 断言方式，assert(preCon)
   c. 复杂系统推荐异常方式
      i. 为什么？assert 并不知道到底是什么原因寄了！
2. 防御式编程

- 防御式编程的基本思想是：在一个方法与其他方法、操作系统、硬件等外界环境交互时，不能确保外界都是正确的，所以要在外界发生错误时，保护方法内部不受损害。
- 常见场景
  - 输入参数是否合法？
  - 用户输入是否有效？
  - 外部文件是否存在？
  - 对其他对象的引用是否为NULL？
  - 其他对象是否已初始化？
  - 其他对象的某个方法是否已执行？
  - 其他对象的返回值是否正确？
  - 数据库系统连接是否正常？
  - 网络连接是否正常？
  - 网络接收的信息是否有效？
- 异常和断言都可以用来实现防御式编程，两种实现方式的差异与契约式设计的实现一样。

模型辅助设计

1. **决策表**：可以用一个表格来表示，其中行表示条件，列表示可能的行动。每个单元格显示在给定条件下应采取的行动。
2. **伪代码**：伪代码可以用缩进来表示代码块，使用自然语言和编程关键字混合来描述算法的逻辑流程。
3. **程序流程图**：流程图通常包括开始/结束符号、处理步骤（矩形框）、决策点（菱形框）、流程线（箭头连接各个框）等，展示程序执行的顺序和条件分支。

软件测试

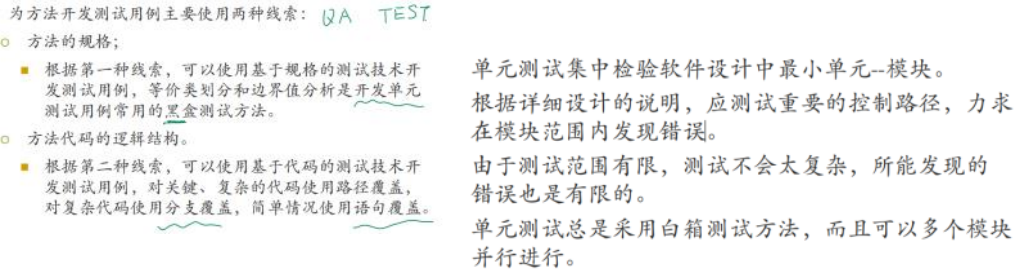
1. **测试的完备性：**
   - 软件测试的完备性是指测试能够尽可能多地发现程序中的错误。然而，由于资源和时间的限制，测试不可能是完全完备的。测试的目的是使用有限的测试用例，在合理的成本和时间内发现尽可能多的缺陷。
   - 测试可以检测到缺陷的存在，但不能保证缺陷的不存在。测试的完备性受到多种因素的限制，包括测试用例的选择、测试执行的时间和测试资源的分配。
2. **动态测试方法：**
   - **白盒测试**：基于程序**内部逻辑和结构**的测试方法。测试者需要了解程序的内部实现，以设计能够揭示潜在错误的测试用例。白盒测试包括**语句覆盖、条件覆盖、路径覆盖**等技术。
   - **黑盒测试**：基于软件**功能**需求的测试方法，测试者不需要了解程序的内部结构。黑盒测试**侧重于检查软件的输入和输出**，以验证功能的正确性。常用的黑盒测试技术包括**等价类划分、边界值分析、错误推测**等。
3. **测试用例的设计：**
   - 3.1 **白箱逻辑覆盖：**
     - 语句覆盖：确保程序中的**每一行代码**至少执行一次**(不管条件语句是往左往右)**。
     - 判定覆盖：确保程序中的**每个判定表达式的结果**都至少执行一次。
     - 条件覆盖：确保每个判定表达式中**每个条件的所有可能**结果都被测试到。
     - 条件组合覆盖：确保判定表达式中**条件的所有可能组合**都被测试到。
     - 路径覆盖：确保程序中每一条可能的**执行路径**都被测试到。 **(若程序图中有环，则每个环至少经过一次)**
   - 3.2 **白箱各类覆盖的区别：**
     - 语句覆盖是最基本的覆盖标准，只关注代码是否被执行。
     - 判定覆盖进一步确保了每个决策点的所有可能结果都被测试。
     - 条件覆盖关注更细粒度的条件结果，确保每个条件都能取到真值和假值。
     - 条件组合覆盖要求测试所有条件组合，以发现更复杂的交互错误。
     - 路径覆盖是最全面的覆盖标准，确保所有可能的执行路径都被测试，这通常需要大量的测试用例。
   - **等价类划分**：将输入数据分为有效等价类和无效等价类，以减少测试用例的数量，同时保持较高的测试覆盖率。
   - **边界值分析**：特别关注输入域的边界值，因为错误往往发生在边界上。
   - **错误推测**：基于错误的一般模式来设计测试用例，以发现可能的错误。
   - **黑箱测试类型**：
     1. **等价类划分示例**：
        - 描述了一个Sale类的getChange方法，该方法接受支付金额（payment）和销售总额（total）作为输入，返回找零金额。
        - 测试用例包括有效数据（payment大于total）和两种无效数据情况（payment小于或等于0，以及payment小于total）。
     2. **边界值分析示例**：
        - 继续使用Sale类的getChange方法，强调了边界值分析的重要性，特别是在等价类边界上设计测试用例。
        - 测试用例包括边界值（如payment等于-1, 0, 1，以及payment等于total，total+1，total-1）。
     3. **基于决策表的方法示例**：
        - 描述了一个根据积分和消费情况决定赠品级别的getGift方法。
        - 使用决策表来设计测试用例，决策表列出了不同积分范围和赠品级别的对应关

系。

4. **基于状态转换的方法示例**：
    - 描述了一个Sales类，该类有多个方法，如添加会员、添加销售项目、计算总额、返回找零和结束销售。
    - 通过建立状态图和转换表来设计测试用例，测试用例包括状态转换和预期的行为。

测试种类

1. **单元测试：**
   - 单元测试是软件开发过程中最基础的测试形式，它关注于程序中最小的可测试部分，通常是单个函数或方法。
   - 单元测试的目的是验证每个单元（函数或方法）按照预期工作，并且能够独立于系统的其他部分进行测试。
   - 单元测试通常由开发人员编写和执行，以确保代码的每个部分都符合设计和功能要求。



2. **集成测试：**
   - 集成测试发生在单元测试之后，目的是检查多个单元或模块联合工作时的交互和通信。
   - 这种测试确保当各个部分组合在一起时，它们能够作为一个整体正常运作，并且满足设计要求。
   - 集成测试可以揭示模块间的接口问题，确保数据在模块之间正确传递。
3. **回归测试：**
   - 回归测试是针对软件进行修改或升级后进行的测试，目的是确保新代码没有破坏现有功能。
   - 当软件的任何部分发生变化时，都需要执行回归测试来验证更改没有引入新的错误或缺陷。
   - 回归测试可能包括之前执行过的测试用例，以确保软件的修改不会影响其他部分的功能。

MC/DC准则

1. 程序中的每个入口点和出口点至少被调用一次；
2. 判定中每个条件的所有取值至少出现一次；
3. 每个判定的所有可能结果至少出现一次；
4. 每个条件都能独立地影响判定的结果，即在其它所有条件不变的情况下改变该条件的值，使得判定结果改变。

软件过程模型

1. **构建-修复模型（Build-and-Fix Model）**
   - **原理**：这是软件开发中最基本的模型，没有明确的阶段划分。开发者直接开始编码，遇到问题时再进行修复。这种方式缺乏规划，通常是基于开发者的直觉和即时决策。
   - **应用**：适用于非常简单和小型的项目，或者当需求非常明确且不会变化时。
2. **瀑布模型（Waterfall Model）**
   - **原理**：瀑布模型将软件开发过程分为一系列阶段性的活动，如需求分析、设计、实现、测试、部署和维护。每个阶段完成后，项目才能进入下一个阶段。这种模型强调预先定义的需求和严格的阶段顺序。
   - **应用**：适用于需求明确、变更不频繁的项目，如某些系统软件或嵌入式系统。
3. **增量迭代模型（Incremental Iterative Model）**
   - **原理**：增量迭代模型通过一系列迭代周期逐步构建软件。每个迭代周期都会增加新的功能或组件，同时对现有功能进行改进。这种方式允许在开发过程中逐步完善软件。
   - **应用**：适用于大型软件项目，允许在每个迭代结束时获得用户反馈，并根据反馈进行调整。
4. **演化模型（Evolutionary Model）**
   - **原理**：演化模型强调软件是在不断变化的环境中逐步演化的。它允许在项目早期就开始开发和交付软件，然后根据用户反馈和市场变化不断演进。
   - **应用**：适用于需求不断变化或高度不确定的项目，常见于新兴市场或创新型产品。
5. **原型模型（Prototyping Model）**
   - **原理**：原型模型使用原型作为探索和沟通工具。通过快速构建一个原型，开发者和用户可以更好地理解需求，并在开发过程中对其进行迭代改进。
   - **应用**：适用于需求不明确或需要用户交互设计的产品，如用户界面设计或用户体验验证。
6. **螺旋模型（Spiral Model）**
   - **原理**：螺旋模型结合了迭代开发和风险分析。每个迭代都包括风险识别、风险缓解、工程和管理系统的评估。这种模型允许逐步细化需求和解决方案。
   - **应用**：适用于高风险或复杂项目，需要在开发过程中不断评估和降低风险。
7. **Rational统一过程（Rational Unified Process, RUP）**
   - **原理**：RUP是一种迭代的、增量的软件开发过程，强调使用统一的软件开发生命周期。它包括需求管理、分析和设计、实现、测试和部署等活动，并支持过程的定制。
   - **应用**：适用于需要严格管理和控制的大型复杂项目，特别是在金融、电信和医疗等领域。
8. **敏捷过程（Agile Process）**
   - **原理**：敏捷过程强调适应性、灵活性和快速响应变化。它倡导跨功能团队的协作、持续交付、技术卓越、简洁性和自组织。敏捷过程包括多种实践方法，如Scrum、极限编程（XP）等。
   - **应用**：适用于需求快速变化或市场竞争激烈的项目，特别强调团队协作和用户反馈。

构建-修复模型（Build-and-Fix Model）

- **特点**：最自然产生的软件开发模型，没有明确的开发活动规划和组织。
- **优点**：适用于非常简单的项目，开发过程灵活。
- **缺点**：缺乏规范和组织，不适合复杂项目，可能导致无法有效控制开发过程。
- **应用场景**：软件规模很小，开发复杂度低，质量要求不高。

瀑布模型（Waterfall Model）

- **特点**：将软件开发活动划分为一系列阶段性活动，每个阶段完成后才能进入下一个阶段。
- **优点**：阶段划分清晰，有助于系统性地处理复杂项目。
- **缺点**：对需求的变更不够灵活，文档驱动可能导致工作量和成本增加。
- **应用场景**：需求明确且稳定，技术成熟，项目复杂度适中。

增量迭代模型（Incremental Iterative Model）

- **特点**：通过迭代开发逐步交付软件，每次迭代增加新的功能或组件。
- **优点**：迭代式开发提高适用性，渐进交付加强用户反馈，降低风险。

- **缺点**：需要开放式体系结构，项目前景和范围难以早期确定。
- **应用场景**：大规模软件系统开发，需求相对明确但可能随时间演变。

演化模型（Evolutionary Model）

- **特点**：适用于需求变更频繁或不确定性较多的领域，强调迭代和并行开发。
- **优点**：更好地应对需求变更，适用于不稳定或新颖领域。
- **缺点**：项目范围难以早期确定，可能忽略分析与设计工作。
- **应用场景**：不稳定领域的大规模软件系统开发，需求复杂或频繁变更。

原型模型（Prototyping Model）

- **特点**：使用原型来解决需求不确定性，包括抛弃式原型和演化式原型。
- **优点**：加强与客户的交流，适用于新颖领域的开发。
- **缺点**：原型开发成本高，可能带来新的风险。
- **应用场景**：需求不稳定或需要澄清，新颖领域开发。

螺旋模型（Spiral Model）

- **特点**：结合了迭代开发和风险分析，强调尽早解决高风险问题。
- **优点**：降低风险，减少项目损失。
- **缺点**：模型复杂，不利于管理者组织开发活动。
- **应用场景**：高风险的大规模软件系统开发。

Rational统一过程（Rational Unified Process, RUP）

- **特点**：总结了传统的最佳实践方法，提供过程定制手段。
- **优点**：适用于从小型到大型的不同规模项目，有软件工程工具支持。
- **缺点**：未考虑软件维护问题，裁剪和配置过程复杂。
- **应用场景**：大型软件团队开发大型项目，需求相对稳定。

敏捷过程（Agile Process）

- **特点**：强调适应需求变更、重视用户价值，轻量级过程方法。
- **优点**：灵活应对需求变更，快速反馈，持续交付价值。
- **缺点**：可能缺乏严格的项目管理和文档记录。
- **应用场景**：需求变化快，需要快速响应市场变化的项目。

原型模型

抛弃式原型（Throwaway Prototype）

- **目的**：主要用于需求验证和风险降低。通过快速构建一个功能简化的模型来展示软件的预期行为和外观。
- **开发过程**：开发团队在短时间内创建一个基本的软件版本，集中于核心功能和用户界面，忽略细节和非关键特性。
- **用户交互**：用户可以与原型交互，提供反馈，帮助明确需求和偏好。
- **后续使用**：一旦需求被验证或澄清，抛弃式原型通常会被丢弃，不会成为最终产品的一部分。
- **优点**：
  - 快速迭代，低成本。
  - 允许用户早期参与，提高需求明确性。
  - 减少开发错误和后期变更的风险。
- **缺点**：
  - 可能存在对原型的过度依赖，导致不愿意丢弃原型。
  - 原型开发可能占用资源和时间，影响项目进度。

演化式原型（Evolutionary Prototype）

- **目的**：作为开发过程中的一个持续组成部分，逐步发展成为最终产品。
- **开发过程**：开发团队构建原型作为开发过程的起点，然后通过迭代添加功能和改进设计，最终形成完整的软件产品。
- **用户交互**：用户在每个迭代阶段提供反馈，指导原型的演化方向。
- **后续使用**：与抛弃式原型不同，演化式原型的代码和设计会被保留并集成到最终产品中。
- **优点**：
  - 允许逐步集成和测试新功能，降低整体风险。
  - 通过持续改进，提高产品的适应性和质量。
  - 有助于团队更好地理解问题空间和解决方案。
- **缺点**：
  - 可能需要更多的前期规划和设计，以确保原型可以演化为最终产品。
  - 如果管理不当，可能会导致产品过于复杂或偏离目标。
  - 演化过程中可能会引入技术债务，需要在未来版本中解决。

适用场景：

- **抛弃式原型**适用于需求不明确或需要快速展示概念的项目。它常用于探索性的项目或在项目初期快速验证想法。
- **演化式原型**适用于需求逐渐明确且需要逐步开发和完善产品的项目。这种方法适合于长期项目，其中软件的功能和性能会随着时间逐步提升。

思考题

思考题1：

如果正在开发一个系统，其中客户不确定他们想要什么，需求通常定义得很差。以下哪种过程模型适合这种类型的发展？

- a. 原型制作
- b. 瀑布
- c. 增量交付
- d. 螺旋

在这种情况下，最合适的过程模型是 a. 原型模型 (Prototyping)。因为当客户不确定他们想要什么，或者需求定义不明确时，原型模型可以通过创建一个或多个原型来帮助澄清需求，允许客户与开发团队互动，从而更好地理解和定义他们的需要。

思考题2：

开发新系统的项目团队在该领域有丰富经验。尽管新项目相当大，但预计与该团队过去开发的应用程序不会有太大变化。哪种过程模型适合这种类型的发展？

- a. 原型制作
- b. 瀑布
- c. 增量交付
- d. 螺旋

对于这种情况，b. 瀑布模型 (Waterfall) 可能是合适的，尤其是如果项目需求相对明确且变化不大。由于团队在该领域经验丰富，他们可能对开发过程和最终产品有清晰的认识，这使得瀑布模型的线性顺序和阶段化方法成为可能。

思考题3：

假设A公司在与你签约构建一个系统时，要求你使用一个给定的过程模型。你遵守了约定，在构建软件时使用了规定的活动、资源和约束。在软件交付和安装后，你的系统经历了灾难性失败，当A公司调查失败原因时，你被指责没有进行代码评审，而代码评审原本可以在软件交付前发现问题。你回答说在公司要求的过程中没有代码评审。

- 你如何看待过程的失败？
- 请问这场辩论的法律和道德问题是什么？

- **过程失败的看法**：过程失败可能是由于过度严格遵循一个过程模型而没有考虑到项目特定的质量保证需求。即使客户指定了过程模型，开发团队也应该强调代码质量和评审的重要性，并在必要时提出调整过程模型的建议。
- **法律和道德问题**：
  - **法律问题**：可能涉及合同义务和责任，如果合同中明确规定了必须遵循的过程模型，但没有提及代码评审，那么可能存在对合同条款的解释和履行的争议。
  - **道德问题**：开发团队有道德责任确保软件质量，即使在客户指定的过程模型中没有包括某些最佳实践，如代码评审，也应该提出这些问题，并寻求解决方案以保证最终产品的质量。

软件维护

软件维护类型：

1. **完善性维护（Perfective maintenance）**：为了满足用户新的需求、增加软件功能而进行的软件修改活动。
2. **适应性维护（Adaptive maintenance）**：为了使软件能适应新的环境而进行的软件修改活动。
3. **修正性维护（Corrective maintenance）**：为了排除软件产品中遗留缺陷而进行的软件修改活动。
4. **预防性维护（Preventive maintenance）**：为了让软件产品在将来可维护，提升可维护性的软件修改活动。

软件维护技术：

1. **遗留软件处理**：针对老旧且可能基于过时技术的软件，决定是否继续维护、替换或重新开发。
2. **逆向工程（Reverse Engineering）**：分析目标系统，识别系统的部件及其交互关系，并使用其他形式或更高层的抽象重新创建系统表现的过程。它用于建立对系统更加准确和清晰的理解，帮助识别可复用资产、发现软件体系结构、推导概念数据结构等。
3. **再工程（Reengineering）**：对遗留软件系统进行分析和重新开发，以利用新技术改善系统或促进现存系统的再利用。再工程包括重新文档化、重组系统结构、转换为更新的编程语言、修改数据结构组织等活动。

逆向工程（Reverse Engineering）

逆向工程是对现有软件系统进行分析，目的是理解其内部结构、设计和实现。这个过程有助于：

- **识别系统组件**：理解系统中各个组件的功能和它们之间的交互。
- **恢复需求和设计**：从现有代码中抽取软件的需求和设计信息。
- **改善理解**：为开发人员提供对系统更深入的理解，特别是在文档不足或缺失的情况下。
- **促进重用**：识别可重用的软件资产，为未来的项目或系统改进提供基础。
- **支持迁移**：帮助将系统从旧技术迁移到新技术。

逆向工程的常用技术包括：

- **代码分析**：使用工具检查源代码，理解其逻辑和结构。
- **文档生成**：从代码中生成技术文档，如类图、序列图等。
- **数据流分析**：识别数据在系统中的流动路径。

再工程（Reengineering）

再工程是在逆向工程的基础上，对软件系统进行改造和升级，以提高其性能、可维护性或适应性。再工程的活动包括：

- **重新文档化**：更新和改进系统文档，包括设计文档、用户手册和技术规范。
- **架构改进**：改进软件架构，提高其模块化和可扩展性。
- **代码重构**：优化代码结构，提高代码质量和可读性，同时不改变其外部行为。
- **技术更新**：将系统迁移到更新的技术平台或编程语言。
- **数据迁移**：转换和迁移数据存储格式，以适应新的系统需求。

再工程的目标是使软件系统更易于维护和扩展，同时减少技术债务和运营成本。

总结

2024年6月14日 14:00

系统分析设计方法 { 结构化
                  { OO

什么方式 软件出生到死亡.

测试

参考

https://eaglebear2002.github.io/
分类 - SpriCoder的博客

https://kimi.moonshot.cn/