注意

- 疫情防控要求
  - 带好口罩
  - 扫描实验台的二维码签到
- 课程QQ群【请同学们选择班级加入】
  - 教师：房敏、谢佳
  - 助教：王志豪、骆旺达

群名称：软A2020（1、2班）
群　号：920499541

- 课程邮箱
  - 1、2班：sda_hitsz12@163.com

注意

- 疫情防控要求
  - 带好口罩
  - 扫描实验台的二维码签到
- 课程QQ群【请同学们选择班级加入】
  - 教师：谢佳、房敏
  - 助教：徐帅、詹明鑫

群名称:软A2020（4、5班）
群　号:967017147

- 课程邮箱
  - 4、5班：sda_hitsz45@163.com

软件设计与开发实践A

哈尔滨工业大学（深圳）

2020年秋

内容安排

- 课程介绍和任务书下达
- 软件设计和开发的基本流程
- 案例1：文本压缩和检索软件
- 案例2：传感器网络单机模拟系统的设计和实现
- 编码规范介绍
- 编写可维护代码的十大原则

课程基本信息

- 《软件设计与开发实践 A》是计算机专业本科生的第一门以实践为主的必修课程

开课学期：2秋 （2020秋）　　总学时：32学时 = 2学时授课 + 30学时实验
课程学分：2

- 这课程的主要目的：
  1. 综合利用《集合论与图论》、《数据结构》、《高级语言程序设计》等基本原理和方法，开展问题求解和软件开发实践；
  2. 引导学生熟练掌握软件开发的一般过程、相关技术和方法；
  3. 训练学生的分析问题和软件开发能力；
  4. 培养学生能够针对实际问题，选择适当的数据结构、设计有效算法；
  5. 提高程序设计的能力和编码质量；提高工程素质。

课程基本信息

- 参考书

1. 《数据结构（C语言版）》， 严蔚敏， 吴伟民， 清华大学出版社. 2012.

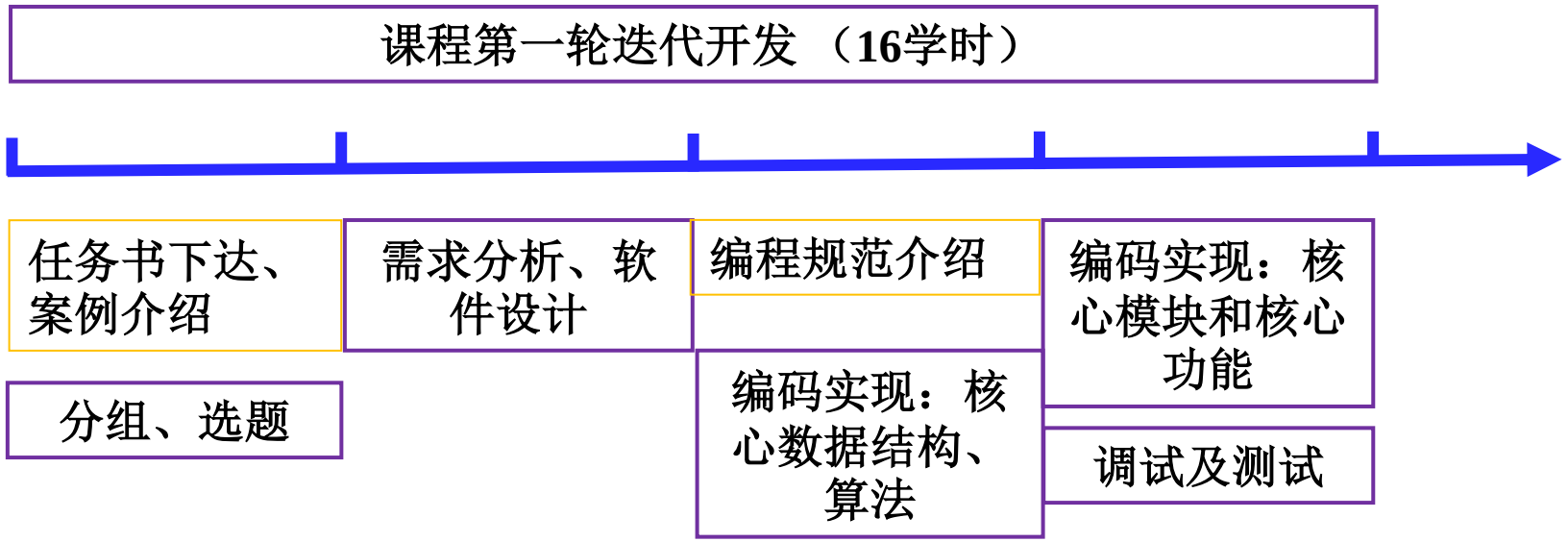
2. “Fundamentals of Data Structures in C”，Ellis Horawitz, Sartaj Sahni, Susan Anderson-Freed著,李建中, 张岩, 李治军译. 机械工业出版社. 2006.

3. 《数据结构与算法(第4版)》，廖明宏，郭福顺，张岩，李秀坤，高等教育出版社，2007.11

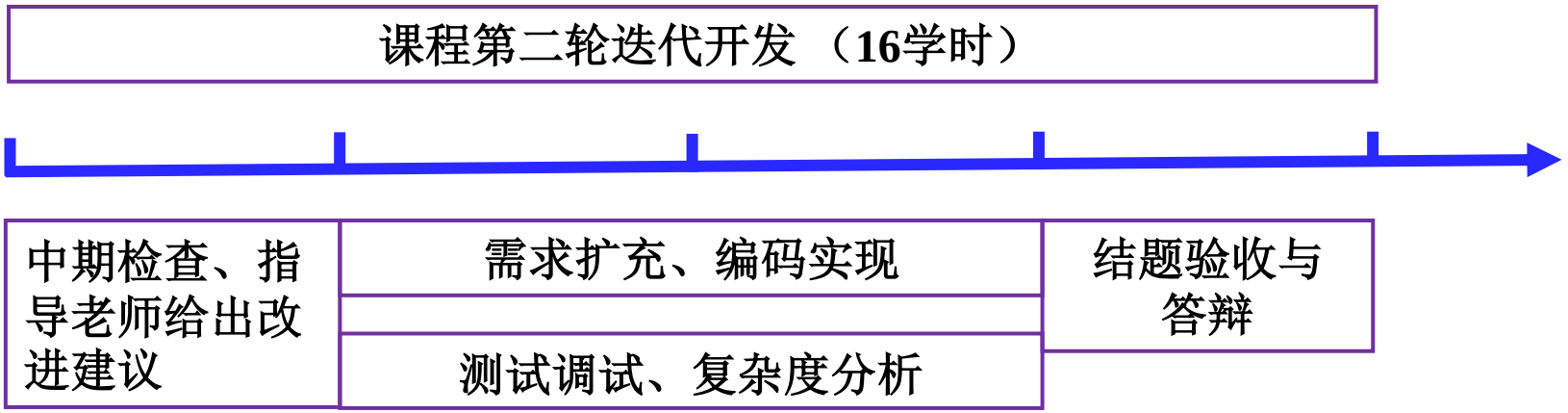
4. 《数据结构与算法实验教程》，李秀坤，张岩，李治军，娄久， 高等教育出版社，2009.06

5. 《软件工程实践教程》，王卫红，江颉，董天阳等,机械工业出版社，2015.09

课程安排


课程第一轮迭代开发 （16学时）
任务书下达、案例介绍
分组、选题
需求分析、软件设计
编程规范介绍
编码实现：核心数据结构、算法
编码实现：核心模块和核心功能
调试及测试
任务书

课程安排


课程第二轮迭代开发 （16学时）
中期检查、指导老师给出改进建议
需求扩充、编码实现
测试调试、复杂度分析
结题验收与答辩
报告
PPT、系统演示

课程成绩

- 主要由4部分构成：
  - 选题（10%）
  - 中期检查+平时（20%）
  - 结题验收软件作品（50%）
  - 软件开发报告（20%）

任务书的撰写

- 认真撰写课程任务书，所有同学独立撰写和提交

1. 每人提交一份，从自己理解角度来写，不可与同组人员大篇幅雷同
2. 认真填写课程任务书中相关内容

哈尔滨工业大学（深圳）软件设计与开发实践 A　任务书

班 号		学 号		姓 名	
院 系			专 业		
组 号		同组人员姓名			
任务书评分（A、B、C、D、E五级）					
任务题目					
问题阐述与分析 **1. 问题背景** **2. 应用意义** **3. 主要问题** **4. 计划实施方案** **5. 拟采用的数据结构及算法**					

工作量
工作计划安排 1. 2.
同组设计者及分工 1. XXX 2. XXX

选题要求

- 要面向现实应用来考虑
  - 自己有兴趣
  - 有一定的实用性，不能凭空瞎想（“用户是上帝”）
  - 能够较好的体现数据结构各知识点的存储操作及算法应用（3种以上数据结构）
  - 要有足够的工作量，尤其是多人同组
  - 编程语言推荐用C/C++，能更好地进行内存操作和呈现；建议自己编写数据结构部分
  - 最好要有GUI操作界面

组织形式

- 分组，每组1~3人
  - 各成员分工必须明确，必须都参与到任务选定、结构设计和编程调试环节中
  - 分组一旦确定，不允许修改，且各组选定组长1人，负责全过程事务
  - 组长职责：负责把握任务整体进展情况
    - 负责给其他组员分配合适足量的工作任务
    - 负责监督其他组员的进展情况
  - 同组成员各环节评分是独立的

评分准则

- 评分依据
  - 选题的意义和价值
  - 主要逻辑结构和存储结构（必须使用至少3种数据结构）
  - 系统界面（UI）、功能及关键算法分析
  - 软件系统开发方法和创新思维
  - 科技文献查阅、文档写作以及讲解演示的能力

评分准则

- 结题验收标准
  - ◆一个中心：要以切实能用的有意义的实际应用题目为出发点，以数据的存储结构为中心，要搞清楚数据在内存中是怎样存储的，是以什么样的结构存储的。
  - ◆两个基本点：
  1. 用到的数据结构相关知识越多越好，如构建和操作线性表、链表、栈、树、图等；
  2. 对应用程序中的数据考虑得越周详就越好，如数据的边界检查及保护等，一定要保证程序中数据的安全性。
  - ◆三个基本要求：
  1. 能熟练解释自己所做程序部分的功能及代码；
  2. 能够利用开发工具调试和运行程序；
  3. 所做的程序要与实际相符，不能想当然。最好是选定题目后从网上搜索一些实用资料做好调研工作。

撰写课程报告

- 1.选题背景与意义
  - 给出选定此课设题目的原因、动机及实际开发价值
- 2.需求分析
  - 分析叙述清楚每个模块的功能要求
- 3.概要设计
  - 说明每个部分的算法设计说明，可以是描述算法的流程图，每个程序中所采用的数据结构及存储结构的说明
- 4.详细设计
  - 开发模块所用的算法的详细设计、数据结构的设计、流程图及系统架构等

撰写课程报告

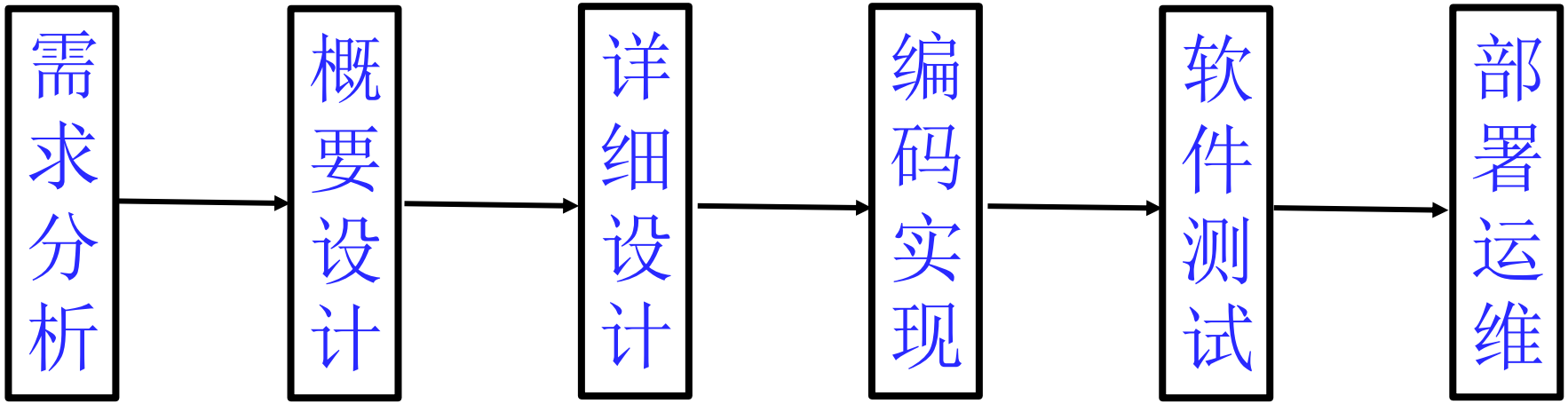
- 5.算法设计与分析
  - 算法的设计思想，所用到的数据结构，并说明如何进行改进或应用
  - 应包括所用数据结构体及关键函数的详细定义及说明
- 6.调试分析
  - 测试数据，测试输出的结果，时间复杂度分析及每个模块设计和调试时存在问题的思考及解决方法
  - 算法的改进设想等
- 7.运行结果与分析
  - 要有多组测试数据及相应结果，并对多组结果进行分析比较；最好是能给出软件操作的部分关键界面截图
- 8.总结
  - 如实撰写课程任务完成过程的收获和体会以及遇到问题的思考、程序调试能力的培养提升等相关内容

内容安排

- 课程介绍和任务书下达
- 软件设计和开发的基本流程
- 案例1：文本压缩和检索软件
- 案例2：传感器网络单机模拟系统的设计和实现
- 编码规范介绍
- 编写可维护代码的十大原则

软件设计开发的基本流程

- 主要包括以下6个基本过程

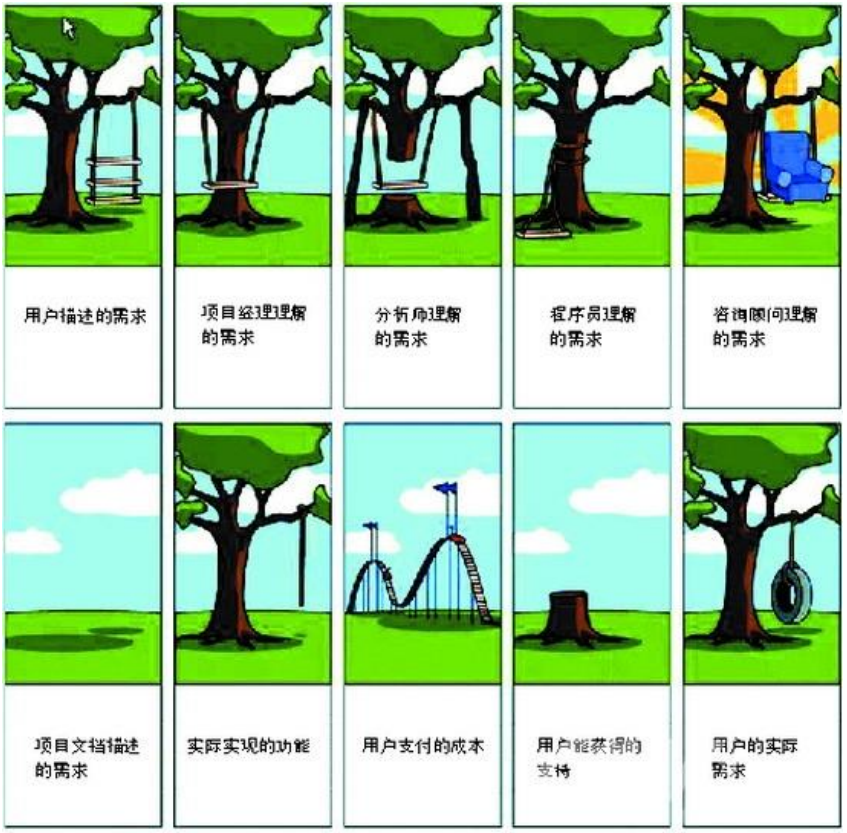


（1）需求分析

- 软件设计和开发的最首要的一个问题就是明确需求，即定义出用户想要的软件到底是什么样的。对用户需求理解的偏差是大部分软件项目失败的一个重要的原因。

我们课程安排是自选题目，大家可能认为自己对自己想做的软件非常清楚，需求理解和分析不重要。然而，从三个层面讲，这仍然是一个不可避免的过程：

(1) 使自己在课程中开发的软件尽可能实用；
(2) 课程的开发项目是以小组为单位共同完成，讨论清楚需求可以消除分歧，让大家向一个方向努力；
(3) 认识需求分析的重要性。

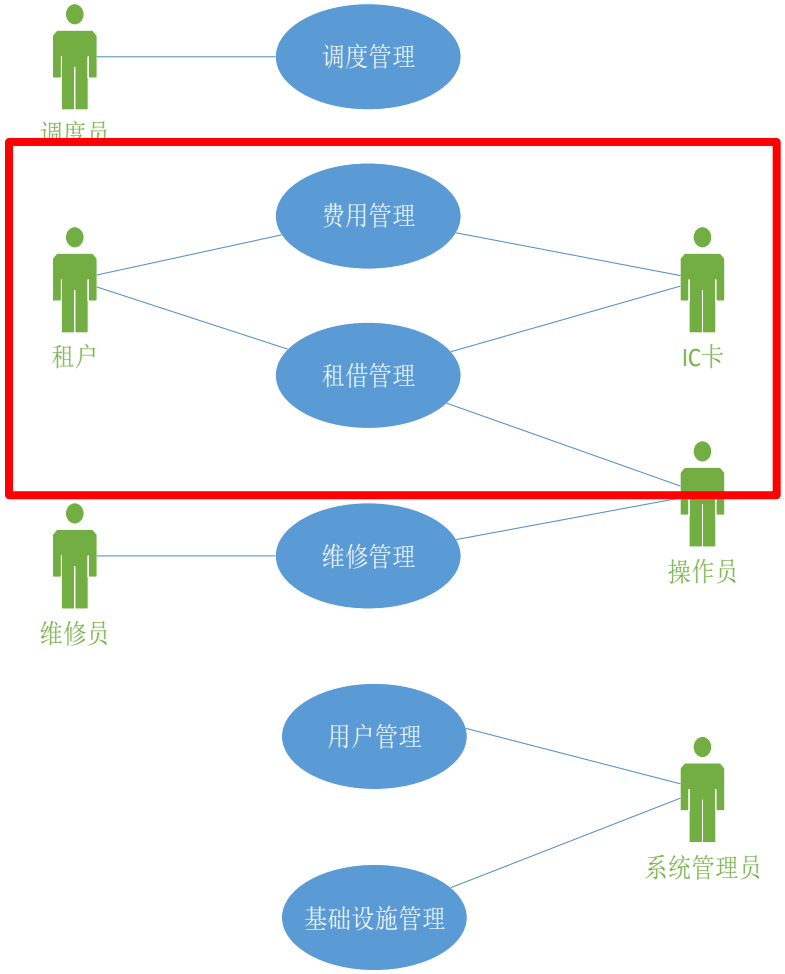


对用户需求的理解偏差造成软件项目失败

例：公共自行车租赁

需求分析：建立业务模型（1）

- 用例图是从用户的观点描述系统的功能



顶层系统用例图

表 1-5　租借管理用例描述

用例名称	租借管理
用例描述	对自行车的租用、归还、查询进行管理
参与者	租户或操作员、IC 卡

表 1-6　用户管理用例描述

用例名称	用户管理
用例描述	对系统中的用户进行管理，如添加租户、添加自行车调度员等操作
参与者	系统管理员

表 1-7　基础设施管理用例描述

用例名称	基础设施管理
用例描述	对系统的基础设施信息进行管理，包括对自行车服务站、车位、自行车信息进行增、删、改、查等操作
参与者	系统管理员

表 1-8　调度管理用例描述

用例名称	调度管理
用例描述	自行车调度员查看当前自行车在各个服务站分布情况，为各服务站按需分配自行车
参与者	调度员

表 1-9　维修管理用例描述

用例名称	维修管理
用例描述	操作员对损坏自行车进行报修，维修员在维修后将自行车状态复位
参与者	维修员，操作员

表 1-10　费用管理用例描述

用例名称	费用管理
用例描述	租户的账户余额管理
参与者	租户，IC 卡接口

需求分析：建立业务模型（2）

- 租借管理用例迭代（逐步深入需求获取）

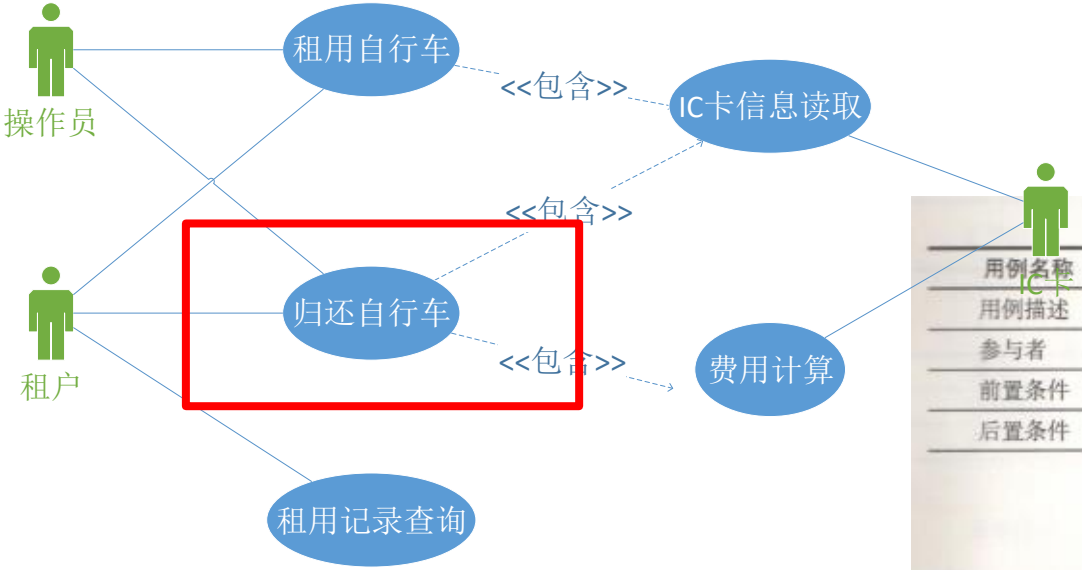


表 1-12　归还自行车子用例描述

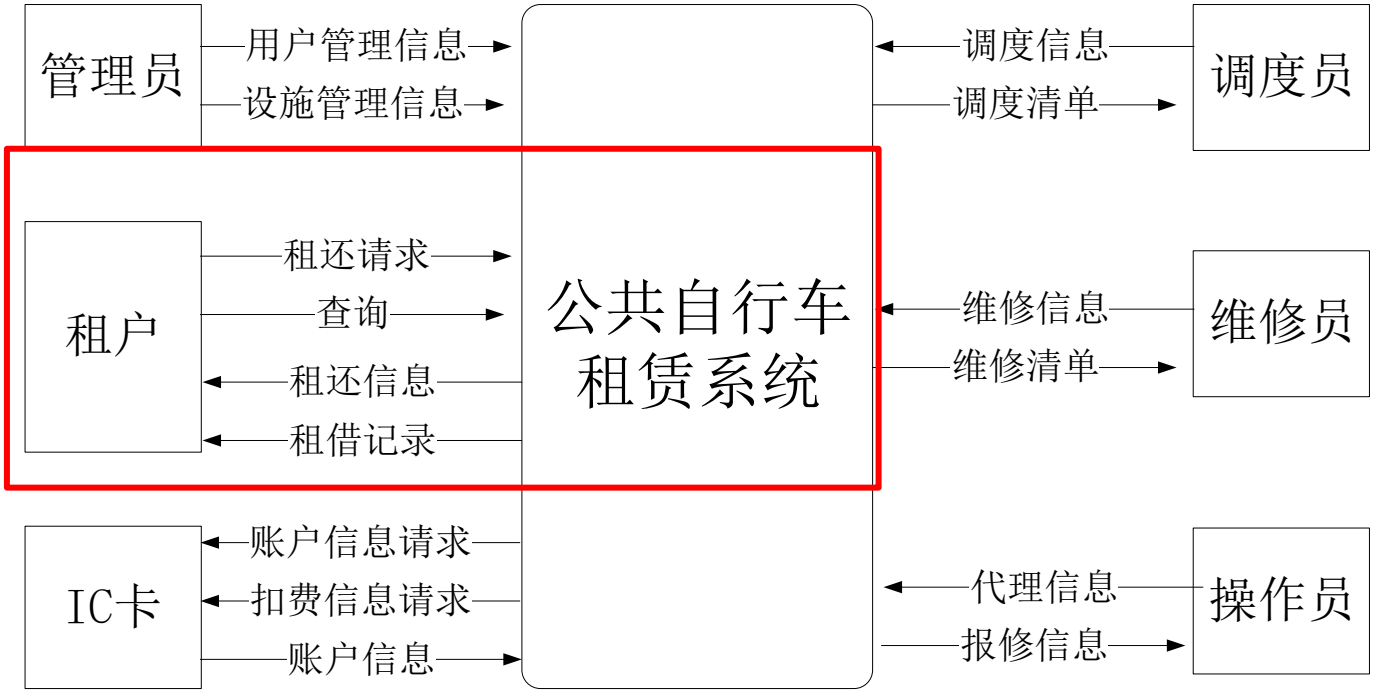
用例名称	归还自行车
用例描述	归还之前所借的自行车
参与者	租户或操作员
前置条件	租户和操作员持有 IC 卡
后置条件	归还成功
基本路径	1. 使用 IC 卡自助还车 （1）租户将自行车推入车位，并将 IC 卡置于车位锁止器感应区 （2）系统通过设备感应获取 IC 卡信息 （3）系统调用“费用计算”子用例计算本次租车所需费用 ①如果 IC 卡中余额大于费用，则从 IC 卡中扣除费用 ②如果 IC 卡中余额不足，则从押金中扣除，更新押金状态 （4）系统更新租户、自行车、车位相关信息 （5）锁止器加锁，自行车归还成功 2. 无空余车位情况下由人工操作员代为还车 （1）人工操作员登录系统选择还车操作 （2）输入用户卡号、归还自行车 ID （3）系统调用“费用计算”子用例计算本次租车所需费用 ①如果 IC 卡中余额大于费用，则从 IC 卡中扣除费用 ②如果 IC 卡中余额不足，则从押金中扣除，更新押金状态 （4）系统更新租户、自行车相关信息 （5）租户自行车归还成功，自行车状态标记为未入车位
扩展路径	车位感应区无法感应 IC 卡

用例描述模板

用例名称	
用例描述	
参与者	
前置条件	
后置条件	
基本路径	
扩展路径	
补充说明	

需求分析：数据流图（1）

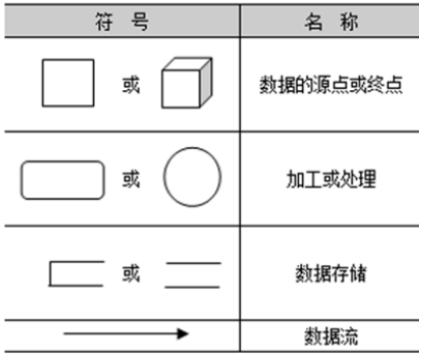
- 以图形的方式描绘数据在系统中流动和处理的过程



顶层数据流图

需求分析：数据流图（2）

- 数据流图迭代



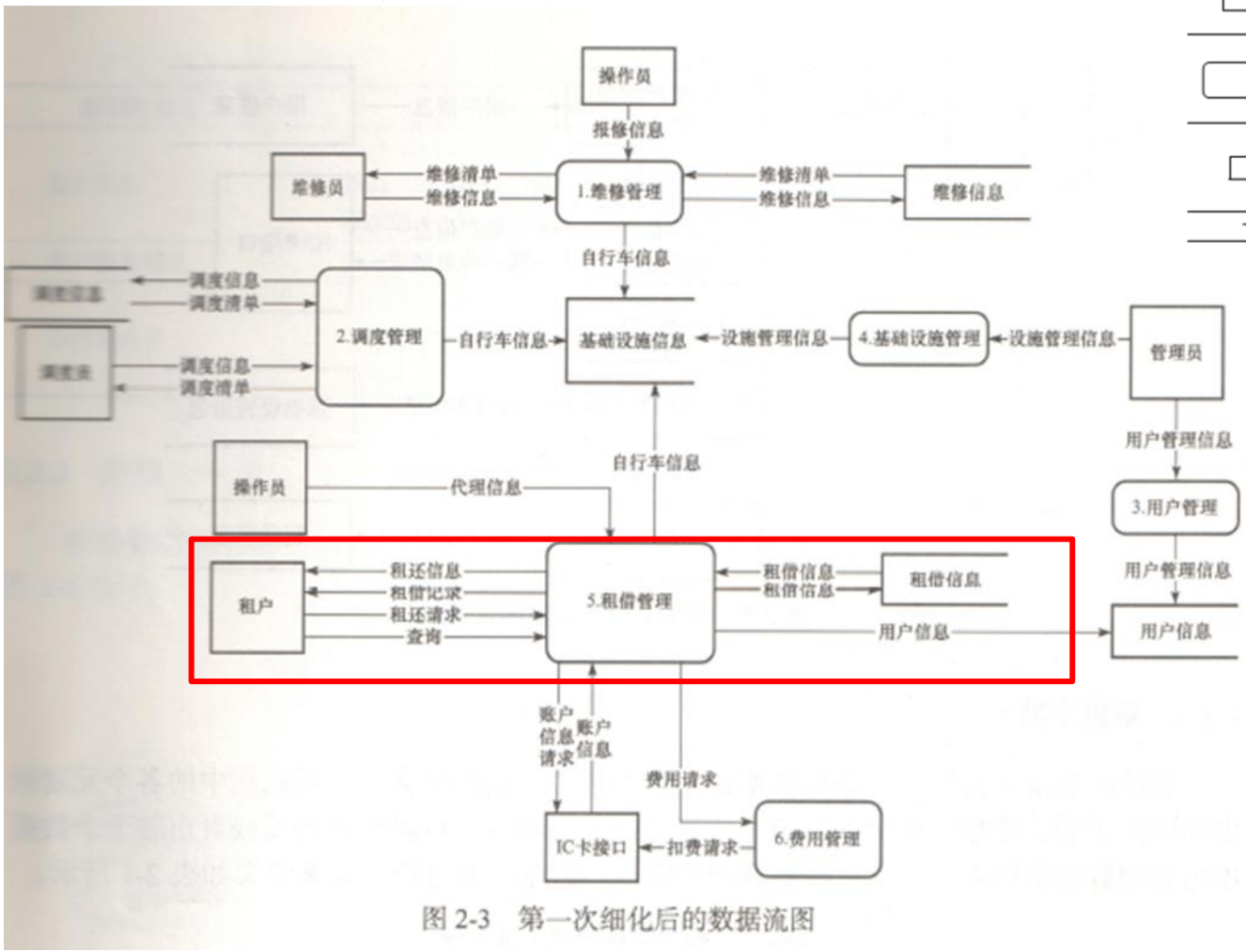


图 2-3　第一次细化后的数据流图

需求分析：数据流图（3）

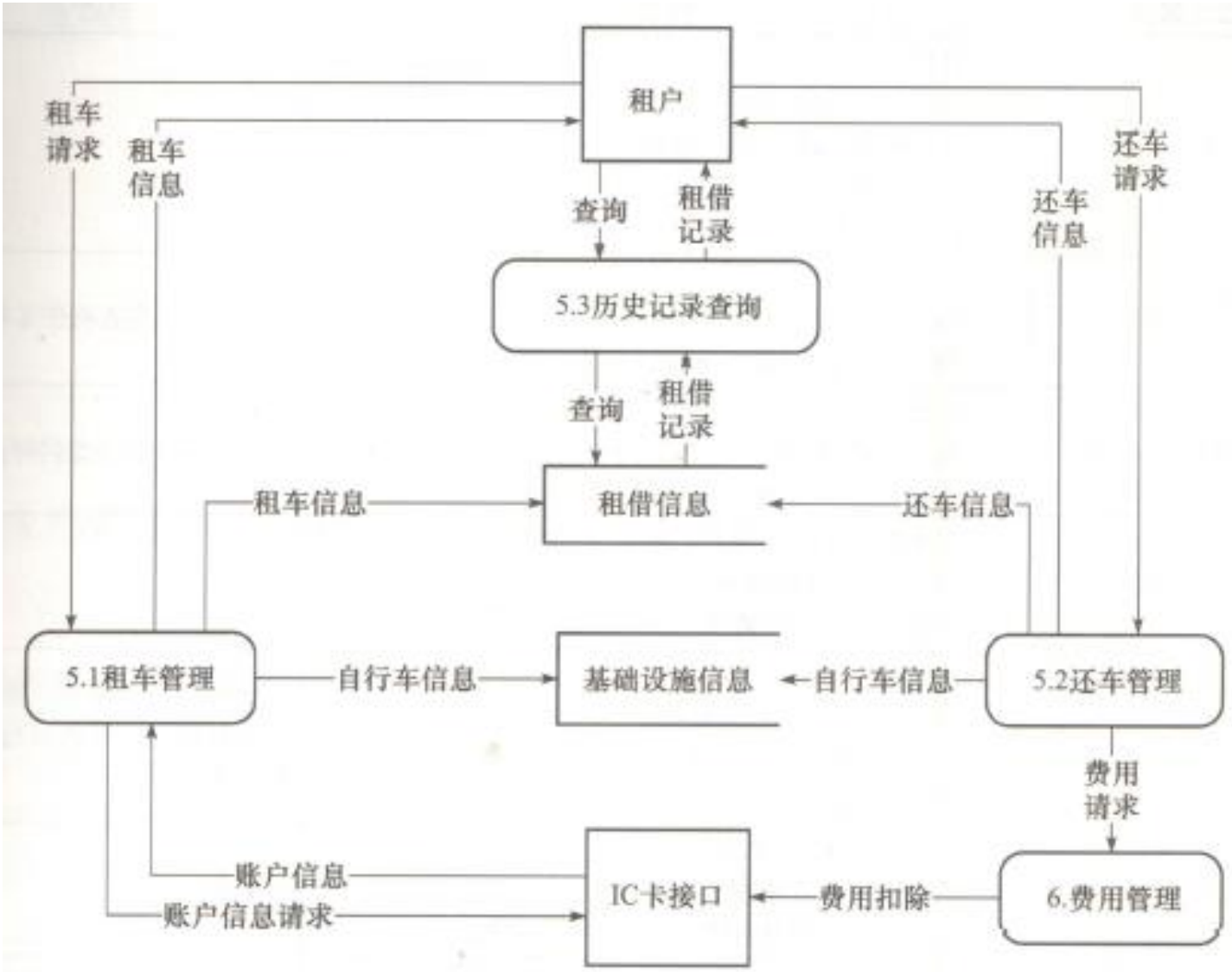
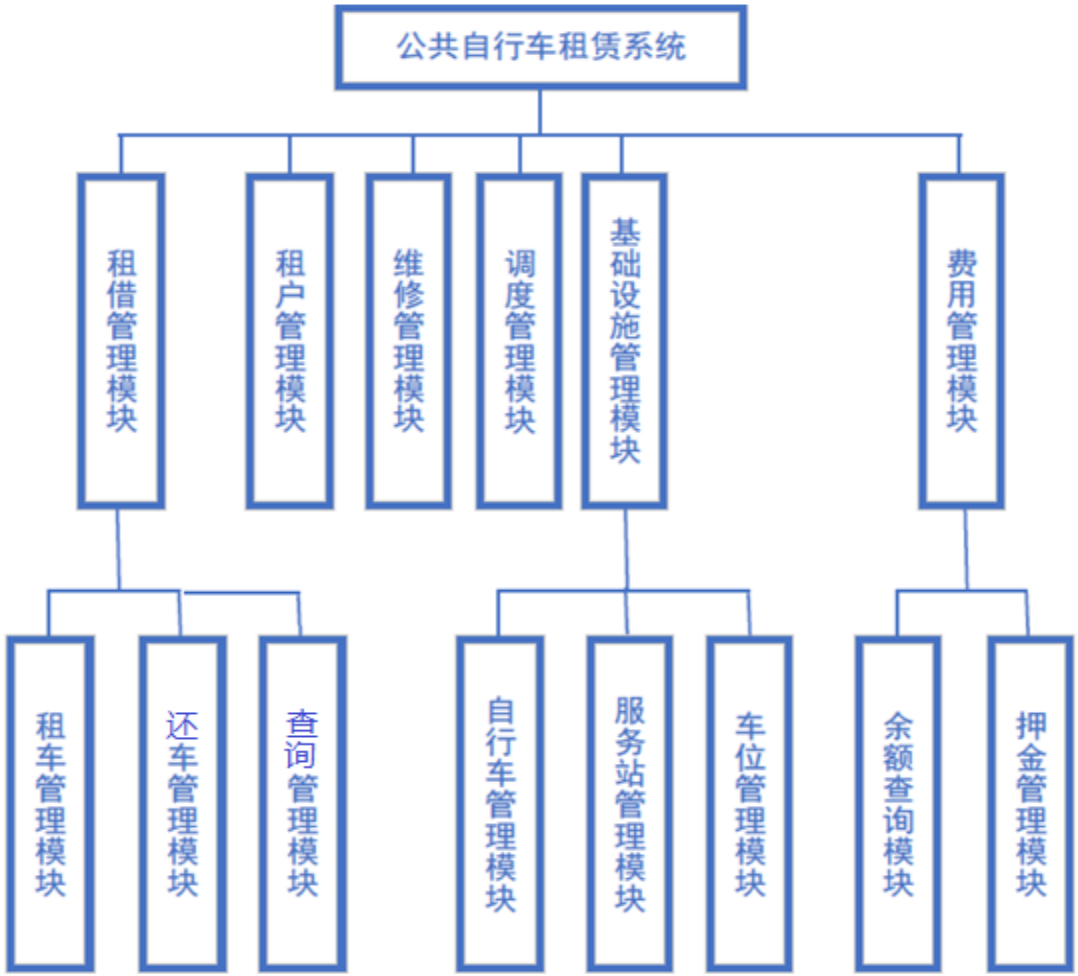


图 2-4　租借管理数据流图

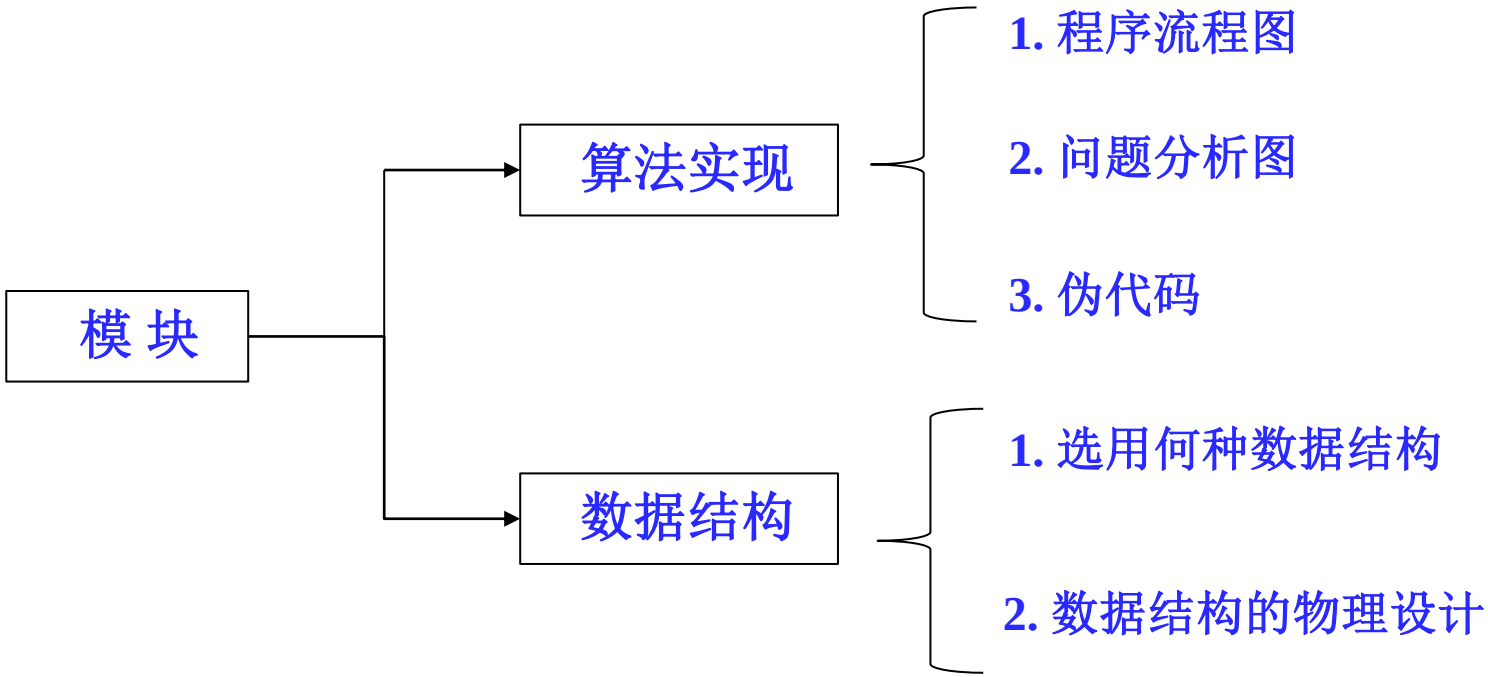
（2）概要设计

- 概要设计是总体实现方案——设计软件的总体结构，包括组成模块、模块的层次结构、每个模块的功能等等。



（3）详细设计

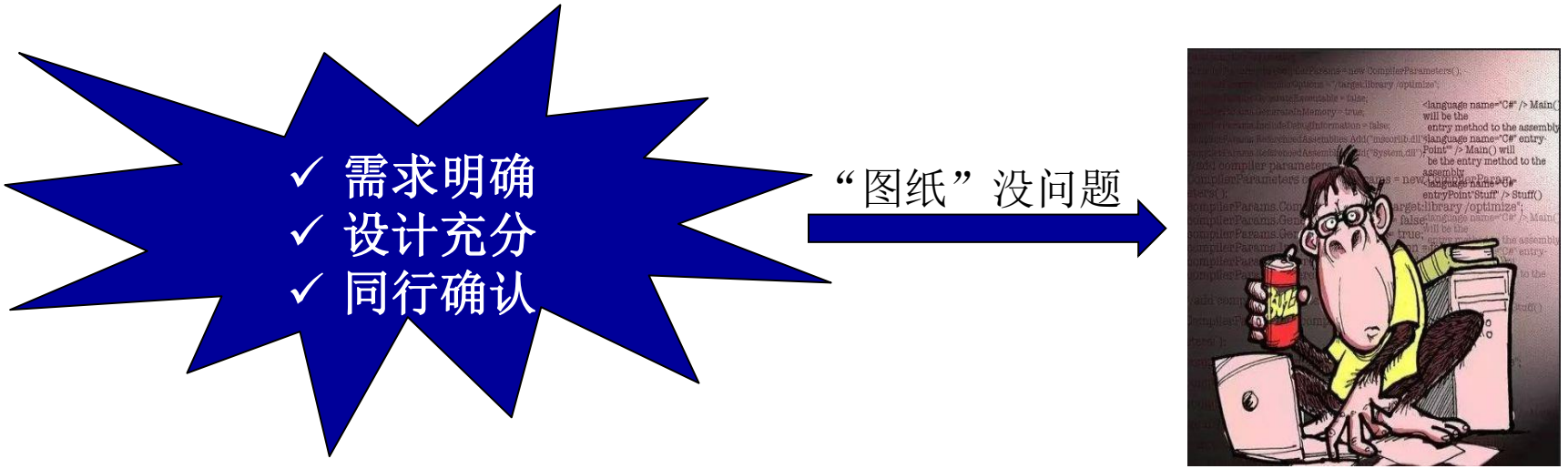
- 详细设计是对概要设计的一个细化，其主要任务是设计每个模块的实现算法、所需的局部数据结构。



（4）编码实现

- 该阶段为每个模块编写程序，也就是将详细设计的结果转化成用某种语言写的程序描述给计算机，让计算机去执行。

- 程序员不要急于编程



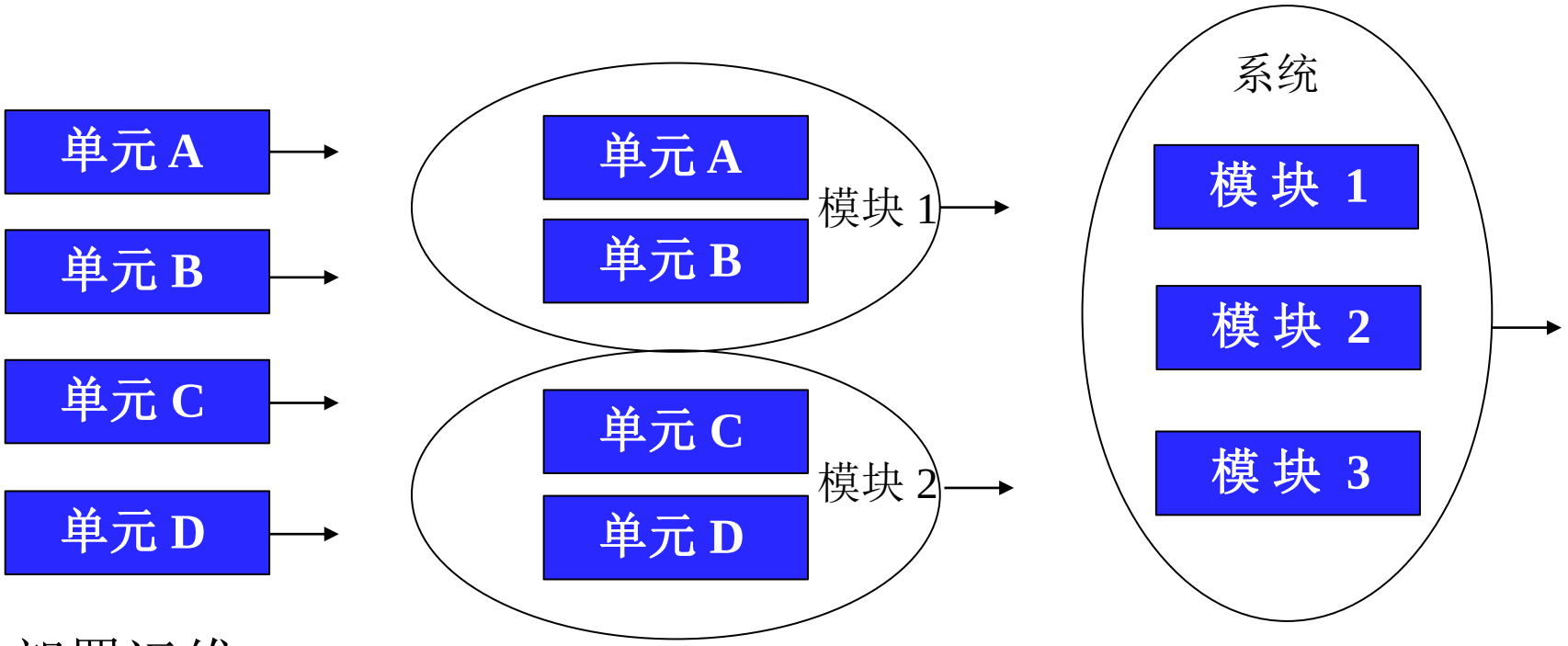
良好的编程规范对于编写易于维护的软件十分重要！！
（后面会跟大家介绍）

（5）软件测试和（6）部署运维

- 软件测试是促进鉴定软件的正确性、完整性、安全性和质量的过程。

软件的实际输出 == 预期输出？

- 单元测试、集成测试和系统测试



- 部署运维

内容安排

- 课程介绍和任务书下达
- 软件设计和开发的基本流程
- 案例1：文本压缩和检索软件
- 案例2：传感器网络单机模拟系统的设计和实现
- 编码规范介绍
- 编写可维护代码的十大原则

案例 1：文本压缩和检索软件

- 设计并开发一个能够对文本压缩，且能实现简单检索的软件。

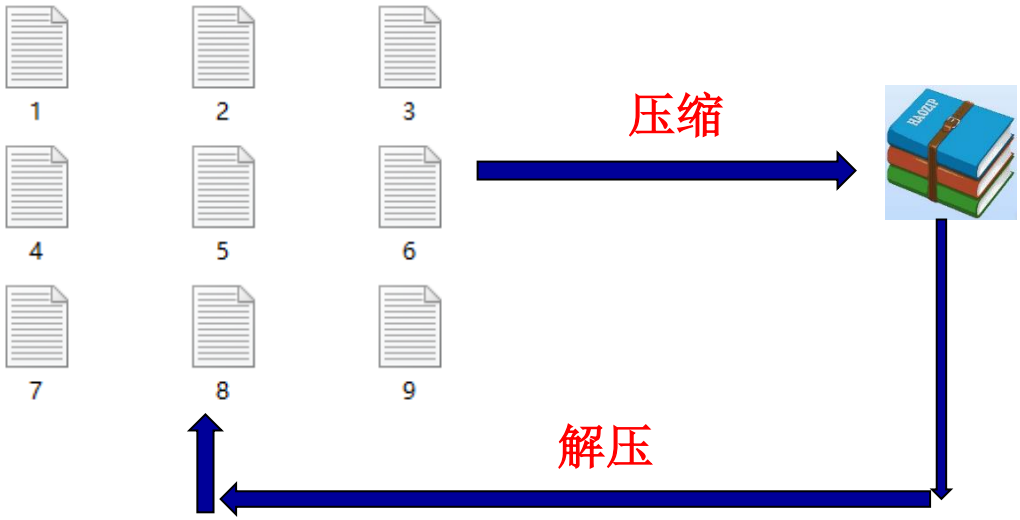
- 要求：

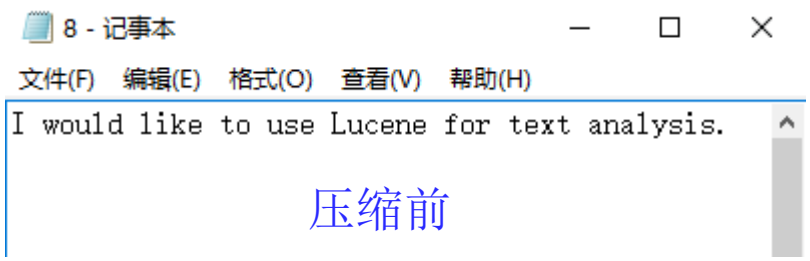
  1. 能对文本进行无损的压缩和解压还原；

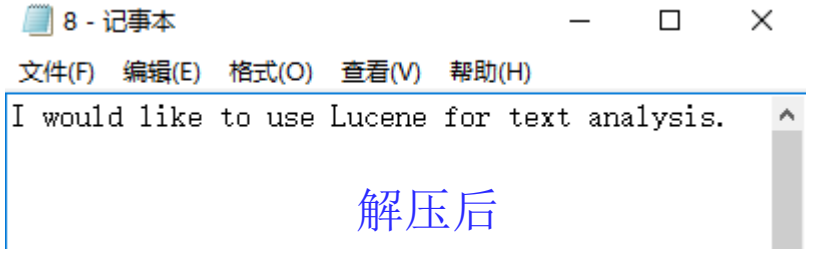
  2. 能快速的实现简单关键词检索，并根据相关性对检索结果进行排序。

需求分析：详细理解需求（1）

- **压缩和解压功能需求：**
  1. 压缩是面向文本的；
  2. 压缩需要是无损的；
  3. 对于压缩后的文件能够解压还原出原文；



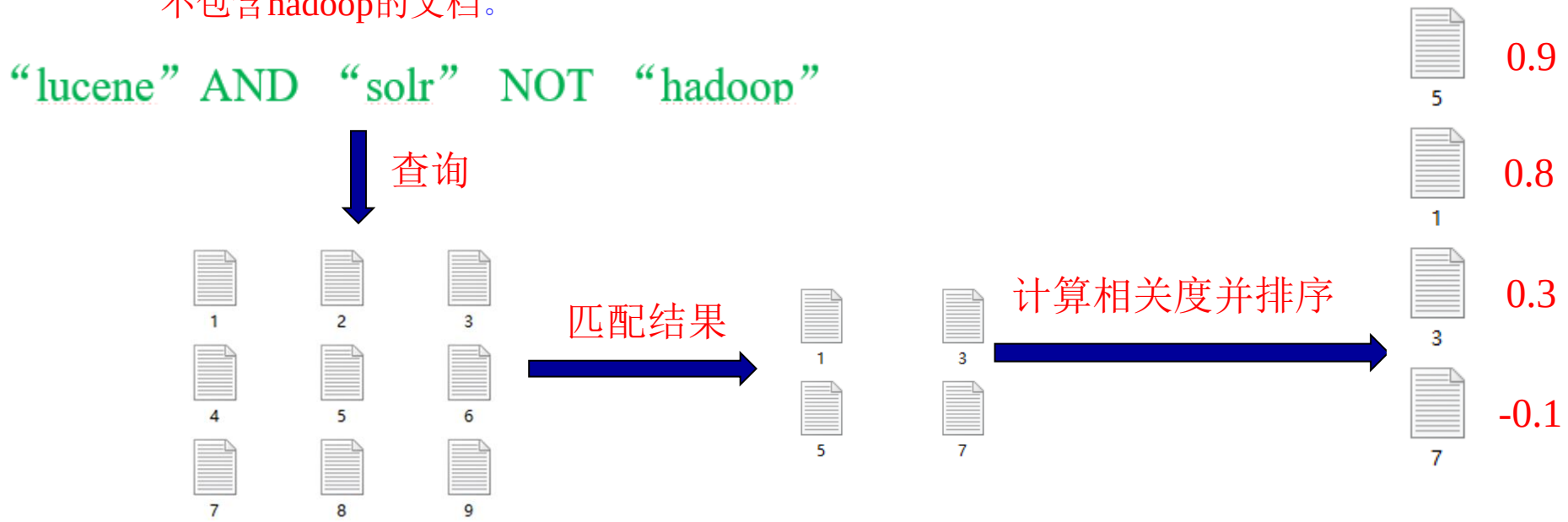




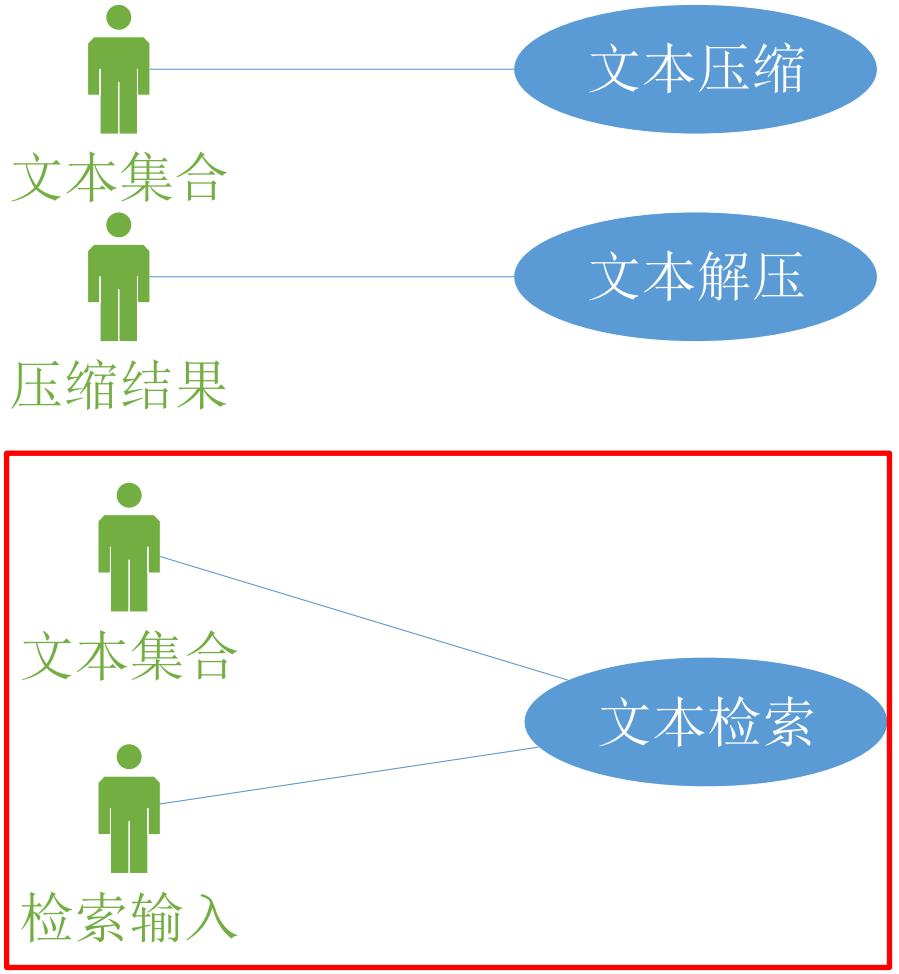
需求分析：详细理解需求（2）

- **检索功能需求：**
  1. 能够实现“AND”，“OR”以及“NOT”操作；
  2. 检索速度要够快；→设计适合快速检索的数据结构
  3. 计算文档与查询关键词的相关度；
  4. 对文档按照相关性进行排序。

例如：输入“lucene”AND “solr” NOT “hadoop”,意思为找出含有lucene和solr但不包含hadoop的文档。



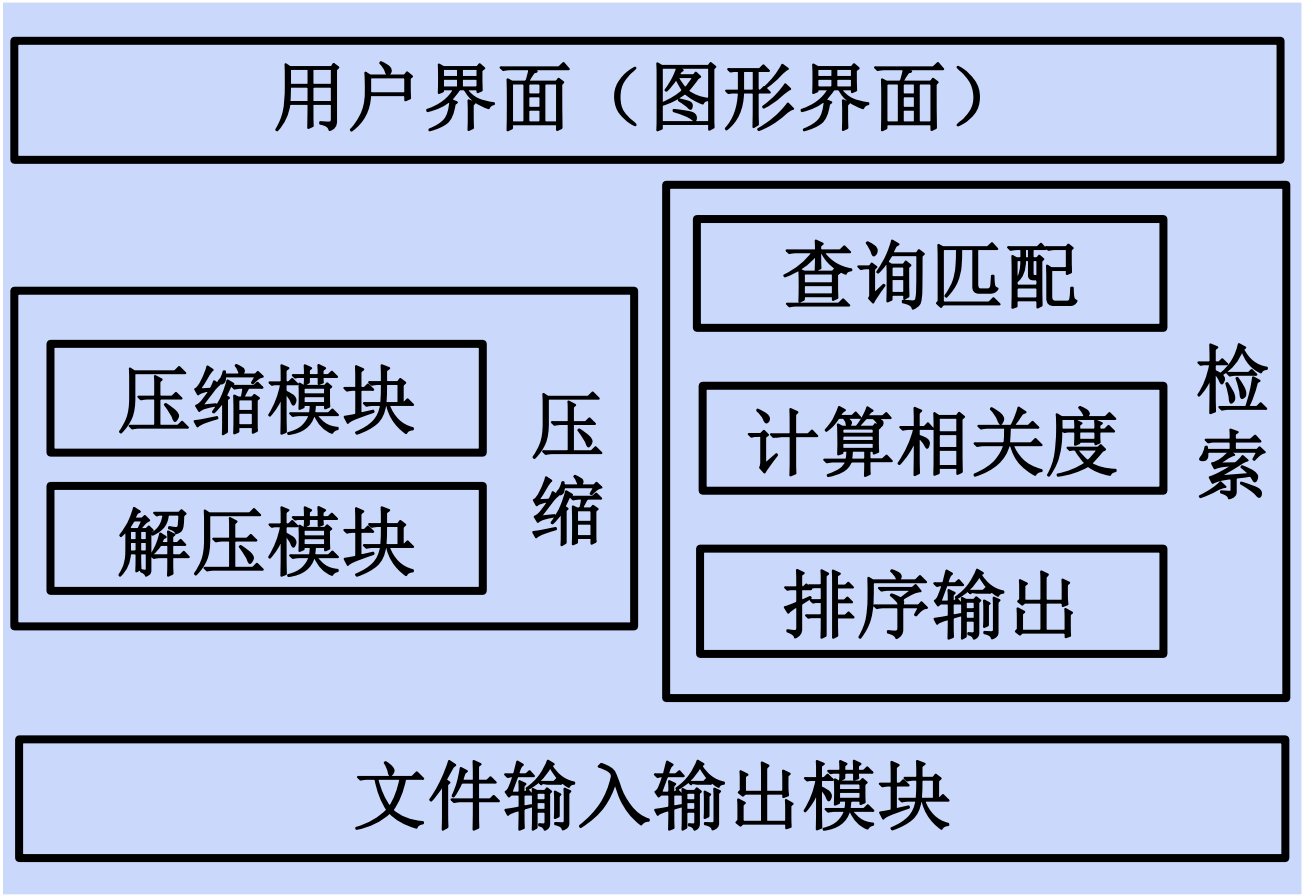
需求分析：用例图和用例描述



用例名称	文本检索
用例描述	根据输入的检索词，考虑AND,OR,NOT关系，找出相关的文档，并根据相关性进行排序
参与者	文档集合和检索输入
前置条件	检索词已输入
后置条件	检索结果
基本路径	1. 解析检索 2. 找出匹配文档 3. 计算相关性 4. 排序
扩展路径	检索输入不符合要求，返回
补充说明	希望检索效率高，在1秒内完成

概要设计

- 基于以上需求分析，设计该文本处理软件的基本模块结构



详细设计：数据结构设计（1）

- **需要管理的数据主要有：**
  - ✓ 文档集合
  - ✓ 词典（词集合）
  - ✓ 完成压缩、解压的辅助结构
  - ✓ 完成检索用的辅助结构

详细设计：数据结构设计（2）

- **词典：**

  词和词之间是对等关系，因此可以使用线性表；

  随着文档集合的变化，该表应具备增、删词和查找词的功能。

其ADT定义为：

```
ADT TermSet
{
    词-词之间的逻辑存储为线性表；  //且须记录每个词的词频
    基本操作：
    void   add（Term t）；          //增加一个词
    void   delete（Term t）；       //删除一个词
    int    find（Term t）；         //查找一个词
    int    frequency(Term t);       //返回词频
};
```

详细设计：数据结构设计（3）

- **文档集合**

  文档-文档之间的关系是对等的，可使用线性表；

  需支持增、删文档及词频查找功能；需能构建词典。

其ADT定义为：

```
ADT DocSet
{
    文档-文档之间的逻辑存储为线性表；
    基本操作：
     void   add（Doc d）；                  //增加一个文档
     void   delete（Doc d）；               //删除一个文档
     int     frequency(int docID, Term t);  //第docID个文档中出现t的频次
     DocSet   find(Term t);                 //词t在哪些文档中出现了
     TermSet   buildTermSet();              //构建词典
};
```

详细设计：数据结构设计（4）

- **压缩和解压辅助结构**

为了实现压缩和解压，需要基于词频建立一棵哈夫曼树，同时需要一个线性表存储每个词的压缩编码。

其ADT定义为：

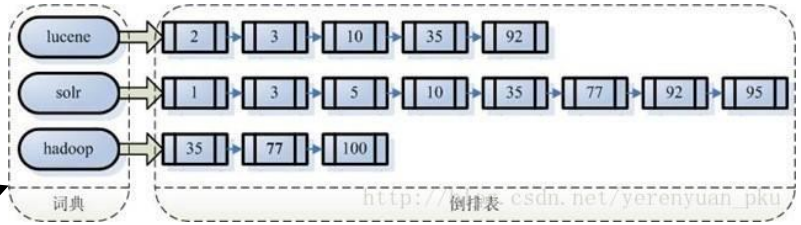
```
ADT Compressor
{
    哈夫曼编码树；
    压缩编码表；
    基本操作：
    void buildCompressor(TermSet ts);          //建立哈夫曼树
    CompDocSet   compressing(DocSet ds);    //对文档集合进行压缩
    DocSet decompressing(CompDocSet  cds ); // 根据压缩结果还原
}
```

其中，CompDocSet为文档压缩后的结果，可用线性表表示。

详细设计：数据结构设计（5）

- **索引结构（检索辅助结构）**

为了快速从文本集合中找出满足查询条件的文本，可建立一个索引数据结构来表示每个词在哪些文档里出现。具体可通过嵌套的线性表来完成。



其ADT可以为：

```
ADT IndexSearcher
{
    嵌套线性表；
    基本操作：
    void createIndexer(TermSet ws, DocSet ds);   //建立索引
    int parseQuery(char * query);                //解析查询
    list<int> searchQuery(char *query);          //匹配查询
    int readerIndexer(char *  filename);         //从文件读入索引
    int writerIndexer(char * filename);          // 将索引写入文件
}
```

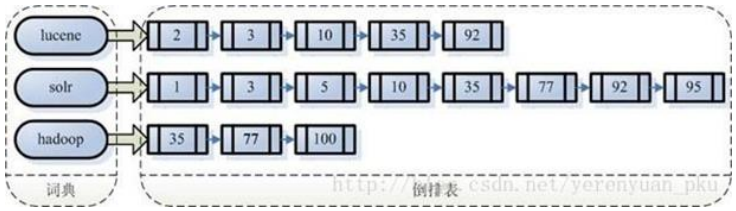
详细设计：算法设计（1）

- **查询匹配算法**：主要是在倒排索引的基础上，找出包含每个词的文档列表，然后再通过这些文档列表的交集、并集、求差等完成；

例如：“lucene” AND “solr” NOT “hadoop”

算法流程

1. 先分别得到包括词“lucene”、“solr”和“hadoop”的三个列表D1, D2, D3
2. 然后求D1和D2的交集 T = D1 ∩ D2
3. 最后求出T和D3的差集 S = T - D3；这时S即为与查询匹配的文档集合



D1 ={2，3，10，35，92}
D2 ={1，2，5，10，35，77，92，95}
D3 ={35,77,100}
T ={2, 10, 35, 92}
S ={2, 10, 92}

详细设计：算法设计（2）

词加权

- 文档与检索词相似度的计算方法

一个文档由多个（或者一个）词（Term）组成，比如：“lucene”，“tutorial”，不同的词可能重要性不一样，比如lucene就比tutorial重要。如果一个文档出现了10次tutorial ，但只出现了一次lucene，而另一文档lucene出现了4次，tutorial出现一次，那么后者很有可能就是我们想要的搜的结果。这就引申出权重（Term weight）的概念。

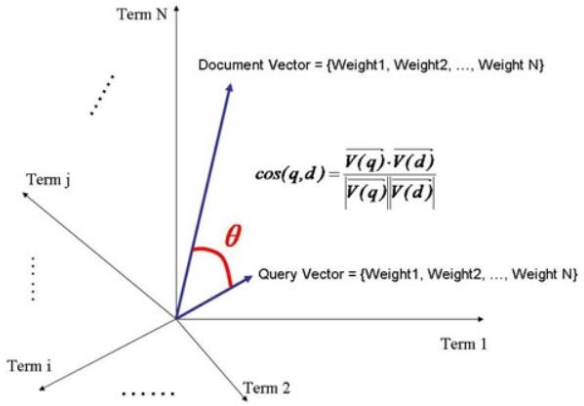
- Term Frequencey（tf）: Term在此文档中出现的频率，tf越大表示越重要
- Document Frequency（df）: 表示有多少文档中出现过这个Term，df越大表示越不重要

$$w_{t,d} = tf_{t,d} \times \log(n / df_t)$$

$w_{t,d}$ = the weight of the term t in document d
$tf_{t,d}$ = frequency of term t in document d
n = total number of documents
df_t = the number of documents that contain term t

详细设计：算法设计（3）

- 文档与检索词的相似度计算方法



空间向量模型计算相关度：

1. 文档中词的权重看作一个向量

Document = {term1, term2, …… ,term N}

Document Vector = {weight1, weight2, …… ,weight N}

2. 把欲要查询的语句看作一个简单的文档，也用向量表示：

Query = {term1, term 2, …… , term N}

Query Vector = {weight1, weight2, …… , weight N}

3. 计算两个向量的cosine相似性

例：Document1 = {lucene, solr, tutorial}

Document1 Vector = {5, 5, 2}

Document2 = {lucene, , tutorial}

Document2 Vector = {5, 0, 2}

Query = {lucene, solr, }

Query Vector = {5, 5, 0}

$$cos(q, d1)=\frac{5\times5+5\times5+2\times0}{\sqrt{5^2+5^2+2^2}\times\sqrt{5^2+5^2+0^2}}=0.962$$

$$cos(q, d2)=\frac{5\times5+0\times5+2\times0}{\sqrt{5^2+0^2+2^2}\times\sqrt{5^2+5^2+0^2}}=0.707$$

详细设计：算法设计（4）

- 根据相似度排序，考虑计算复杂度和稳定性后选择合适的排序算法

排序法	最差时间分析	平均时间复杂度	稳定度	空间复杂度
冒泡排序	$O(n^2)$	$O(n^2)$	稳定	O(1)
快速排序	$O(n^2)$	$O(n*\log_2 n)$	不稳定	$O(\log_2 n)\sim O(n)$
选择排序	$O(n^2)$	$O(n^2)$	稳定	O(1)
二叉树排序	$O(n^2)$	$O(n*\log_2 n)$	不一顶	O(n)
插入排序	$O(n^2)$	$O(n^2)$	稳定	O(1)
堆排序	$O(n*\log_2 n)$	$O(n*\log_2 n)$	不稳定	O(1)
希尔排序	O	O	不稳定	O(1)

编码实现：主要数据结构（1）

- 词典数据结构

```
struct TermEntry
{
        char * term;
        int     frequency;
}
class TermSet
{
        private:
             list<TermEntry>   m_termList;          //词列表（含有频率）
        public:
             …
 };
```

编码实现：主要数据结构（2）

- 文档集数据结构

```
struct DocEntry
{
        char  * filename;
        char * content;
        int     docID;
}
class DocSet
{
        private:
             list< DocEntry >   m_docList;          //文档列表
        public:
             …
 };
```

编码实现：主要数据结构（3）

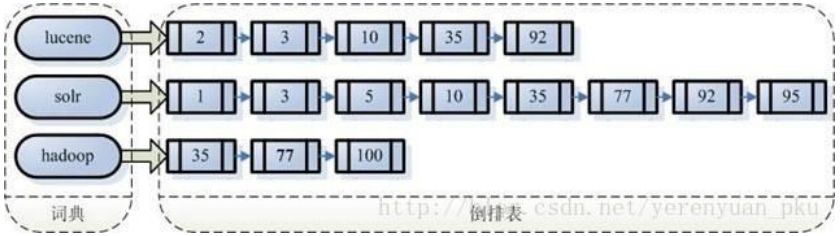
- 压缩和解压数据结构

```
class Compressor
{
        private:
                HuffmanTree    m_hTree;
                list<CodeEntry> m_termCodeList;
        public:
                …
}
struct CodeEntry
{
        char *  term;
        list<bool>   termCode;
}
```

编码实现：主要数据结构（4）

- 倒排索引数据结构

```
class IndexSeacher
{
        private:
            list<IndexEntry>  m_myIndex;
        public:
            …
    }
struct IndexEntry
{
        char*  term;
        list<DocFreqEntry>     docsWithTerm;
 }
```




```
struct DocFreqEntry
{
        int          docID;
        int          Freq;
}
```

编码实现：主要算法

- 压缩和解压过程

```
DocSet        my_corpous(xxx);              //初始化
TermSet       my_vocabulary(xxx);           //初始化
Compressor  my_comp();                      //初始化
my_comp.buildCompressor(my_vocabulary);
CompDocSet  my_compressResult=my_comp. compressing(my_corpous);
DocSet        my_decompResult=my_comp.decompressing(my_compressResult);
```

- 查询检索过程

```
IndexSearcher    myIS();
myIS.createIndexer(my_vocabulary, my_corpous);
list<int> myDocList=myIS.searchQuery(“lucene AND solr NOT hadoop”);
```

对myDocList的每个文档，计算其与查询的相似性，并排序返回

小结

- 该案例用到了哈夫曼树、线性表、嵌套线性表、集合等数据结构
- 该案例用到了排序算法、计算相似度算法
- 可仿照该过程对其选题进行分析、设计和开发

内容安排

- 课程介绍和任务书下达
- 软件设计和开发的基本流程
- 案例1：文本压缩和检索软件
- 案例2：传感器网络单机模拟系统的设计和实现
- 编码规范介绍
- 编写可维护代码的十大原则

任务书提交

请每位同学提交任务书（word或者pdf格式）到课程邮箱

- 1、2班：sda_hitsz12@163.com
- 4、5班：sda_hitsz45@163.com

邮件和任务书的命名规则：　班级_学号_姓名

例：4_1901104_张三**

提交截止时间：下一次课前一天凌晨0点

周二上课的截止时间：下周一凌晨0点

周四上课的截止时间：下周三凌晨0点

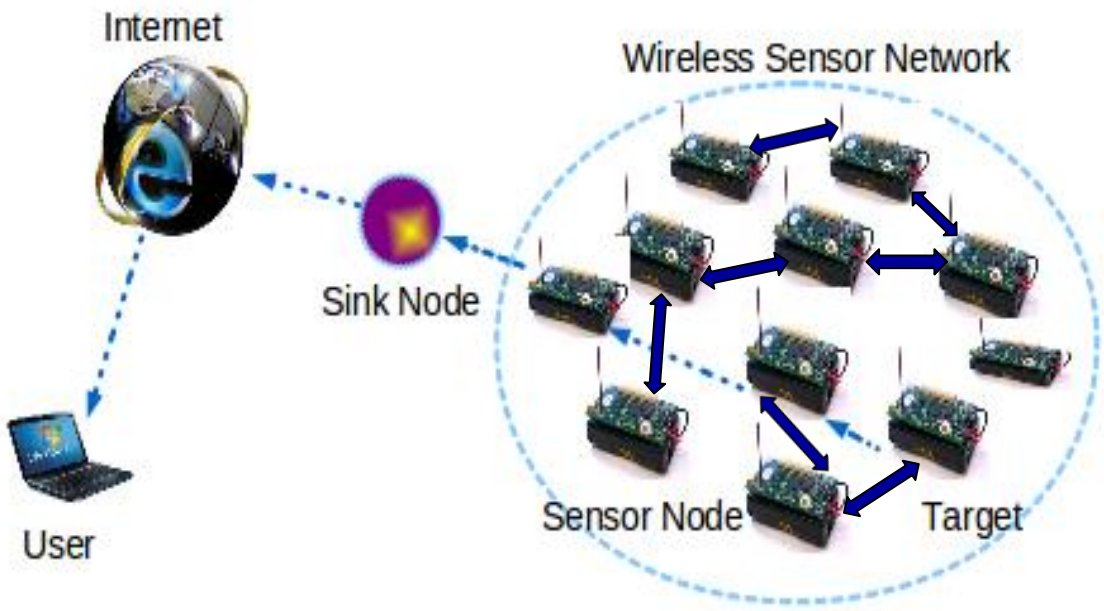
案例2：传感器网络单机模拟系统

- 设计一个软件系统，其能在单机上模拟传感器网络的工作过程

- 功能要求：
  1. 能模拟Sensor传感器的工作；
  2. 能模拟Data Station的工作；
  3. 能模拟传感器网络系统的工作；
  4. 能够可视化整个工作过程。

需求分析（1）

- 什么是传感器网络？基本结构是什么样的？其如何工作？



传感器网络的基本结构

- 1. 什么传感器网络？

集成了传感器，微机电系统和网络三大技术形成新颖的网络，用于军事侦查、环境信息的获取等。

- 2.传感器网络的基本结构

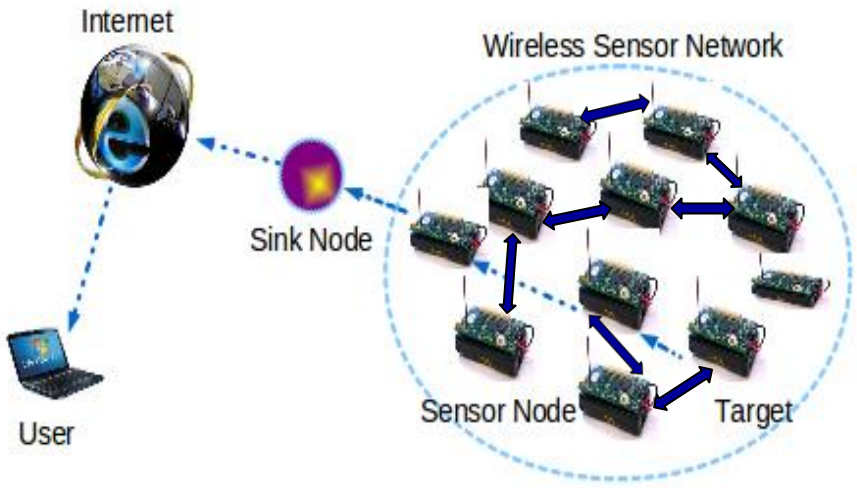
如左图所示，其中的Sensor （传感器）集成有传感器、数据处理和信息通信的模块，这些Sensor通过自组织的方式构成网络，从而探测包括：温度、湿度等信息，并将这些信息发送到接收点（Data Station), 左图中是Sink Node。

- 3. 传感器间的通信方式

有线、无线、红外、光等，这里考虑无线传输

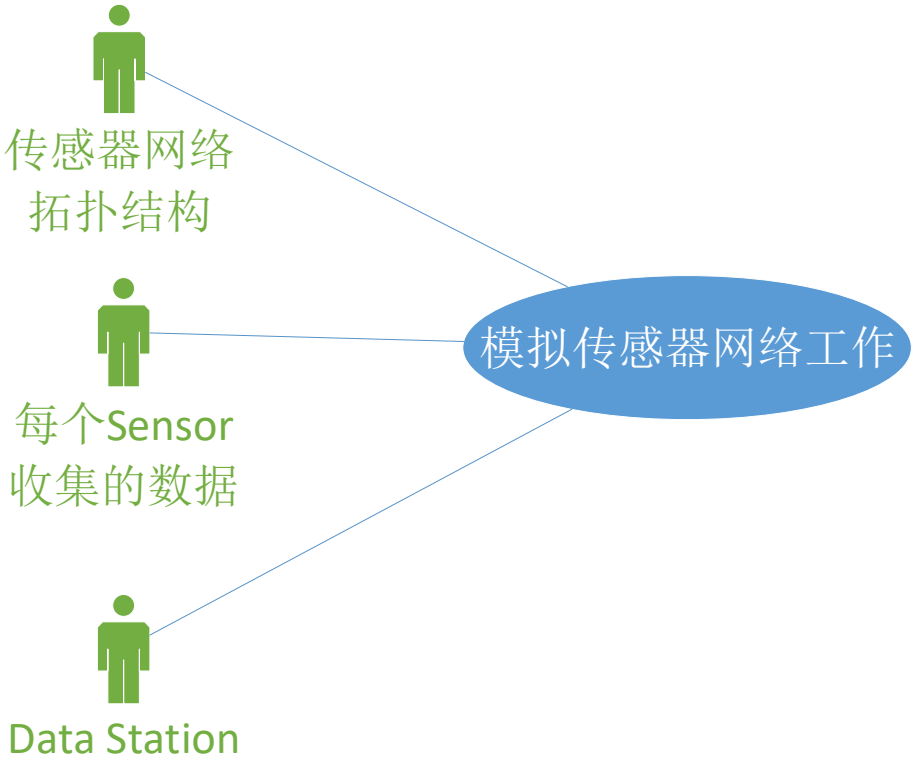
需求分析（2）

- **为了在单机模拟传感器网络，需要完成以下子系统的模拟**
  - 模拟Data Station（即图中的Sink Node）的工作；
  - 模拟Sensor的工作；
  - 传感器网络路由的构建；
  - 模拟传感器网络的工作；
  - 系统的可视化；



说明：其中，Sensor负责收集环境数据，并通过相互之间的连续（无线通信）将消息传送到Data Station。Data Station负责在收集的数据上进行实际的应用推理，本系统只要求模拟网络，与具体应用无关，所以Data Station只假定其存在，并能区别于Sensor即可，其数据处理功能不模拟。

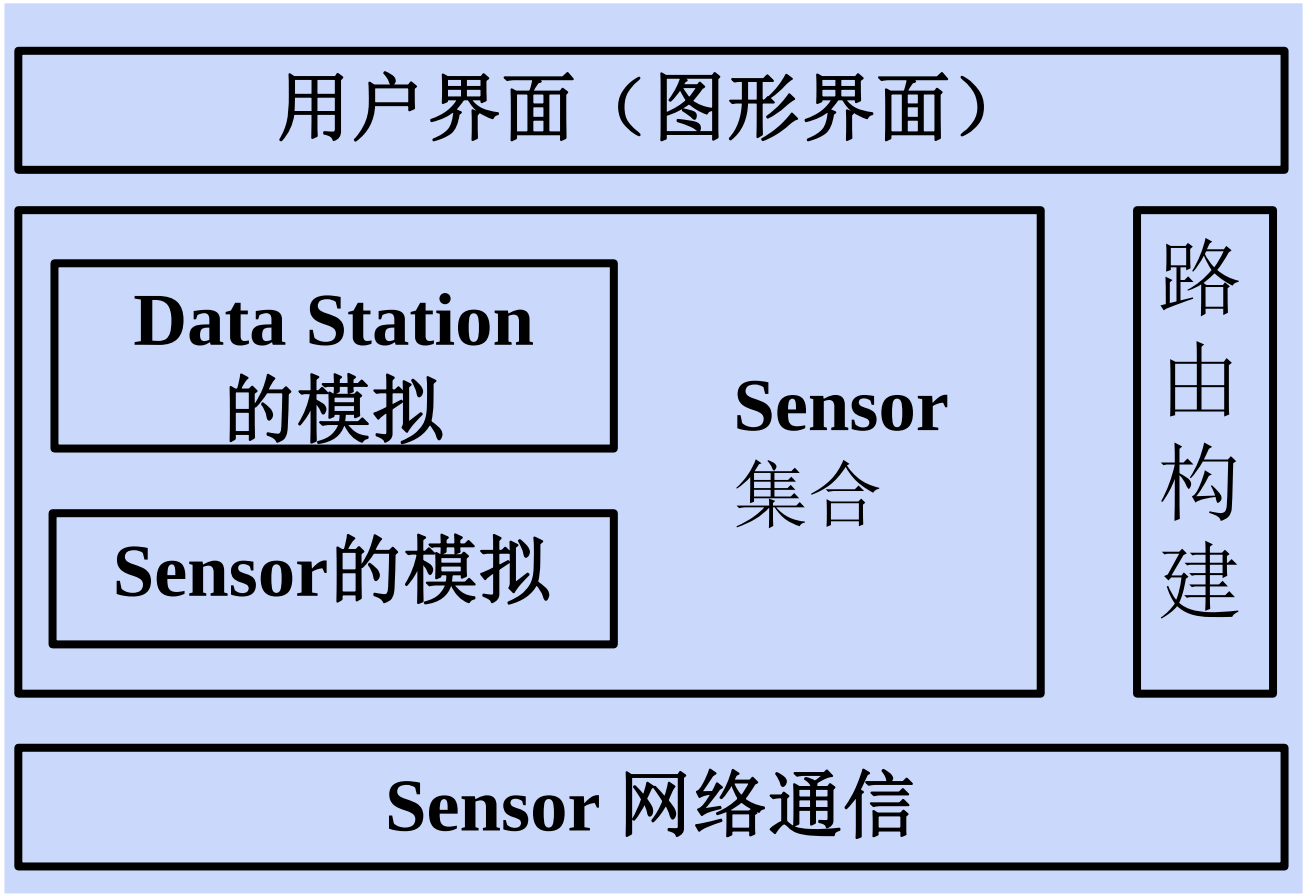
用例图和用例描述



用例名称	模拟传感器网络工作
用例描述	模拟每个Sensor已收集到数据，通过传感器网络拓扑结构，发送到Data Station的过程
参与者	拓扑结构、数据包、Data station
前置条件	拓扑结构已建好
后置条件	Data station 存储数据包
基本路径	1. 每个Sensor将其数据包转发给邻居Sensor 2. 邻居Sensor将数据包存放的消息队列 3. 继续转发，直到数据包到达Data Station

概要设计

- 基于以上分析，可总结出Sensor网络的基本模块结构



详细设计：数据结构设计（1）

- 针对这一系统，需要管理的数据主要有：
  - ✓ Sensor集合
  - ✓ Sensor集合形成的路由结构
  - ✓ 完成Sensor间通信的辅助结构
- **Sensor集合数据结构：**

各Sensor间是对等关系，形成顺序表。因而，需要建立线性表结构，因此其ADT可以定义为：

```
ADT SensorSet
{
   数据之间的逻辑结构为线性表；
   基本操作：
   Sensor Locate(int i)；//定位第i个Sensor
}
```

详细设计：数据结构设计（2）

- **路由数据结构：**

Sensor间构建的路由关系是一种图结构，需在这个图上定义相关操作如，获取邻居节点等。

其ADT定义为：

```
ADT SensorGraph
{
        数据之间的逻辑结构为图结构；
        基本操作：
         Sensor FindFirstAdjacent(); //定位第一个Sensor
         Sensor FindNextAdjacent(Sensor *sen);
                                        //定位下一个Sensor
}
```

详细设计：数据结构设计（3）

- **通信辅助数据结构：**

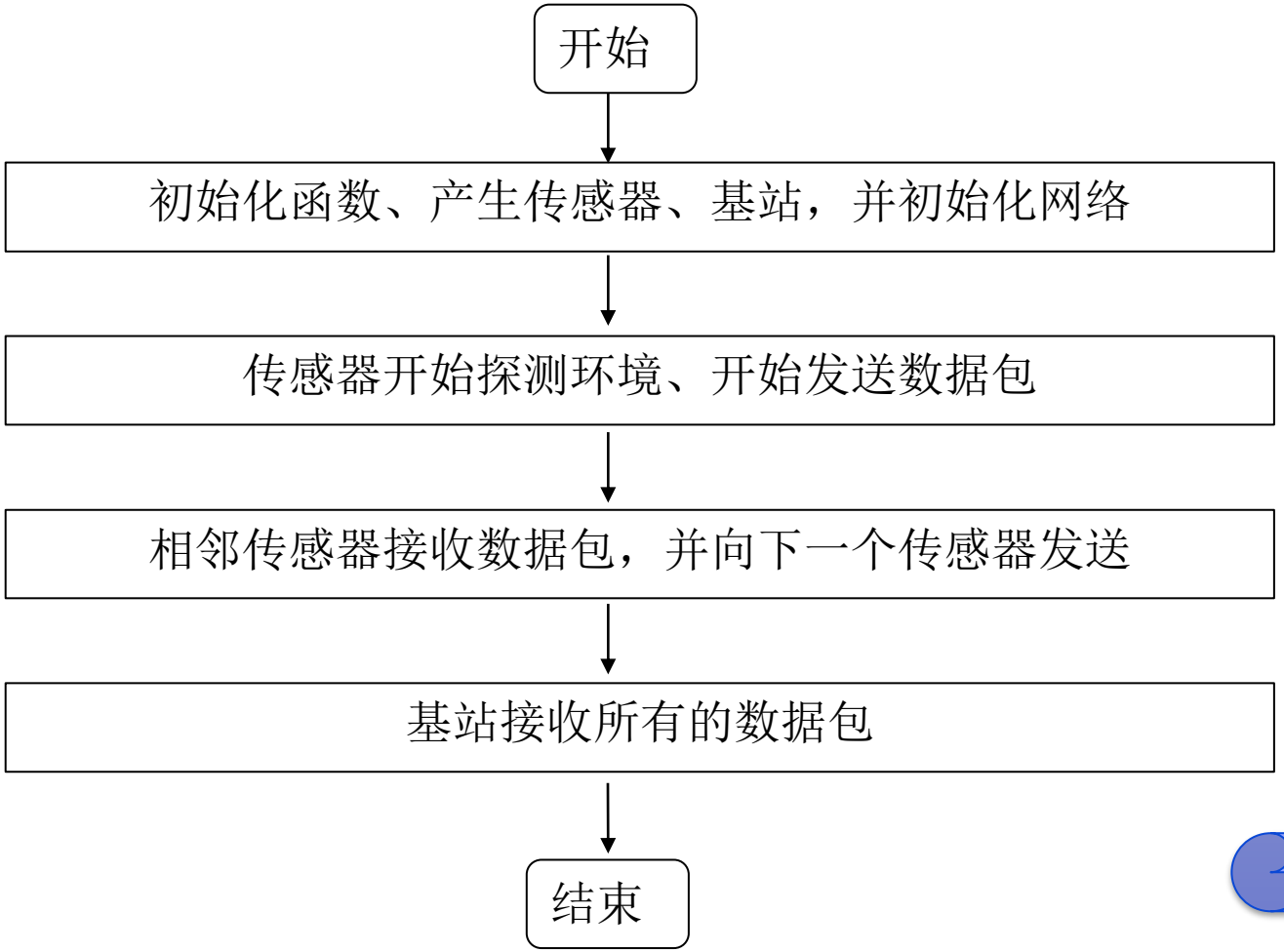
为了完成Sensor之间的通信，各个Sensor间会传递数据包，这些数据包在各Sensor处应该按照队列的方式组织。

其ADT定义为：

```
ADT PackageQueue
{
        数据之间的逻辑结构为线性结构；
        基本操作：
         Package Dequeue();     //取出队列的头
         Void Enqueue(Package *pac);
                                  //将pac放入队列的尾
 } ;
```

详细设计：算法设计（1）

- **Sensor网络的基本工作流程**



主要算法：

1.拓扑网络的初始化

2. 数据包的收发

3. 工作过程的可视化

详细设计：算法设计（2）

- 拓扑路由建立算法的设计

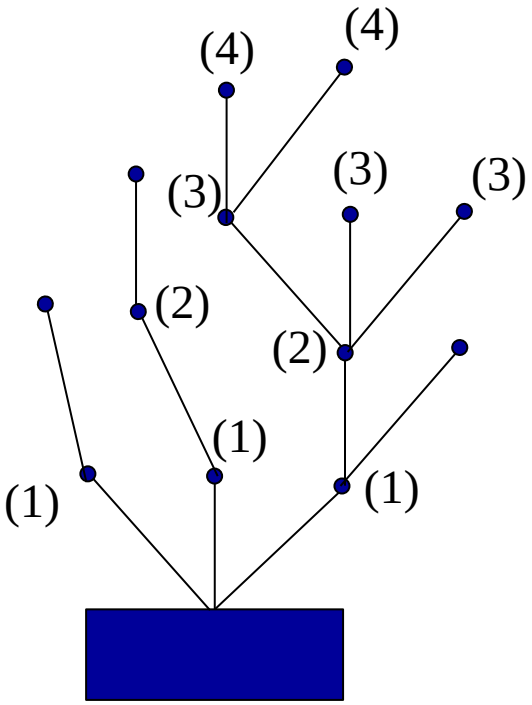
（1）随机路由算法——在无线通信的范围内，随机选择一个或者若干点进行数据转发或（组发）

```
RandomTopo()      //建立随机拓扑路由
Input:  SensorSet S
Output: SensorGraph G

For each s in S
     s broadcast the “s: ping” by radio
EndFor

For each t in S
    If t received the “s: ping” message
       t send to s “t:pong” meassage
    EndIf
EndFor

For each s in S
    If  s receive the “t:pong“ message
        s.BuildEdge(G, t.ID);

    EndIf
EndFor
```

详细设计：算法设计（2）

- 拓扑路由建立算法的设计

（2）基于分层思想的路由算法——通过分层的方法找到一条能量最小的路径。




```
LayerNumberBuild() //层号的建立
Input: SensorSet S, Basestation D
Output: s layer for all s in S

BroadRadius =m;
For(k=1; k<MAX_LAYER; k++)
        D.SetBroadRadius = m*k;
        D.BroadCast("layer(k)");
EndFor
For each s in S
        If t receive the "s. layer(k)" message
            If(s.layer<k) s.layer=k;
        EndIf
EndFor
```

详细设计：算法设计（3）

- **数据包收发算法的设计**

  数据包的收发设计主要是通过每个Sensor上带有的队列来完成的，其算法的核心部分为：

```
pac   =   S.m_packageQueue.Dequeue();  //从S的数据包队列取出
                                       排在队首的数据包pac

T     =   S.SelelectNextHop();         //S根据路由结构选择下一个
                                       发送对象T

T.m_packageQueue.Enqueue(pac)          //T将数据包pac放入其队列
                                       的尾部
```

详细设计：算法设计（4）

- 可视化界面的相关算法设计

根据之前的分析，本模拟系统需要完成三个部分的可视化：

1. Sensor散播后位置的可视化，称其为节点散播状态(BROADING)

通过DisplayAllSensors()来完成，从SensorSet取出所有的sensor，根据其位置绘制

2. Sensor建立了连接，形成了拓扑结构(NETWORKING)

通过DrawTheNet()来完成，根据路由拓扑结构，两个Sensor之间有边画一条线

3. 模拟Sensor间及基站间的数据包收发过程(COMMUNICATING)

通过DrawTheSimulation()来完成，基本思想是将每个节点处的数据包情况显示出来，即针对每一个Sensor，获取其数据包队列PackageQueue, 从队列中依次取出数据包，将其显示在屏幕上，然后再将其插入到该队列中

编码实现：主要数据结构（1）

- **Sensor和Data Station的数据结构**

```
Class Csensor: public Cobject
{
    private:
        int m_nLocationX;
        int m_nLocationY;
        int m_nID;

        int m_nRadius;
        int m_n SenseInterval;
    public:
        成员函数…
}
```

编码实现：主要数据结构（2）

- **数据包的实现**

```
struct package{
        int     SourceID;           //数据包的源节点ID
        int     Destination;        //数据包的目的ID
        char *  Content;            //数据包的内容
    };
```

- **Sensor集合的实现**

由于Sensor集合不动态变化的线性表，采用数组对象来实现。

```
CObjectArray    m_SensorArray;
```

编码实现：主要数据结构（3）

- **由Sensor形成的图（路由结构）的实现**

  图的表示有邻接矩阵（集中式存储）和邻接表两种方式，邻接表更合适。在Sensor类中增加一个存有其邻居Sensor信息的数组。

具体实现为：

```
struct RoutingEntry
{
        int m_iNeighborSensorID;
        other fileds;
 };
   RoutingEntry m_RoutingTable[TABLE_SIZE];
```

表中存放的是邻接节点的ID，根据ID可以再到m_SensorArray上找到该ID值对应的Sensor。

编码实现：主要数据结构（4）

- **每个Sensor上的数据包队列的实现**

每个sensor上的队列定义为CQueue m_qPackageQueue;

其中，CQueue为一个队列类，定义为：

```
class CQueue{
        public:
            CQueue():rear(NULL), front(NULL){};
            ~CQueue();
             void          Enqueue(Package * item);
             Package *     Dequeue();
             int           Isempty() const{return front==NULL;}
        private:
            CQueueNode * rear, *front;
    }
```

编码实现：核心算法实现（1）

- **Sensor间拓扑路由的建立**

✓ 广播范围内的点的确定：

计算sensor s和t的欧式距离： $D(s,t)=((s.x-t.x))^2+((s.y-t.y))^2$

若 D(s,t)<s.radius, 则t在s的广播范围内。

✓ 如何创建一条边：

创建一条边就是在Sensor节点的路由表加入一个新项，将该项包含邻居节点的信息。

✓ 如何在邻居节点随机选取一个节点传送数据包：

```
Select  = rand()% m_nRoutingEntryNum;
SelectedNextHopId = m_RoutingTable[Select].ID;
```

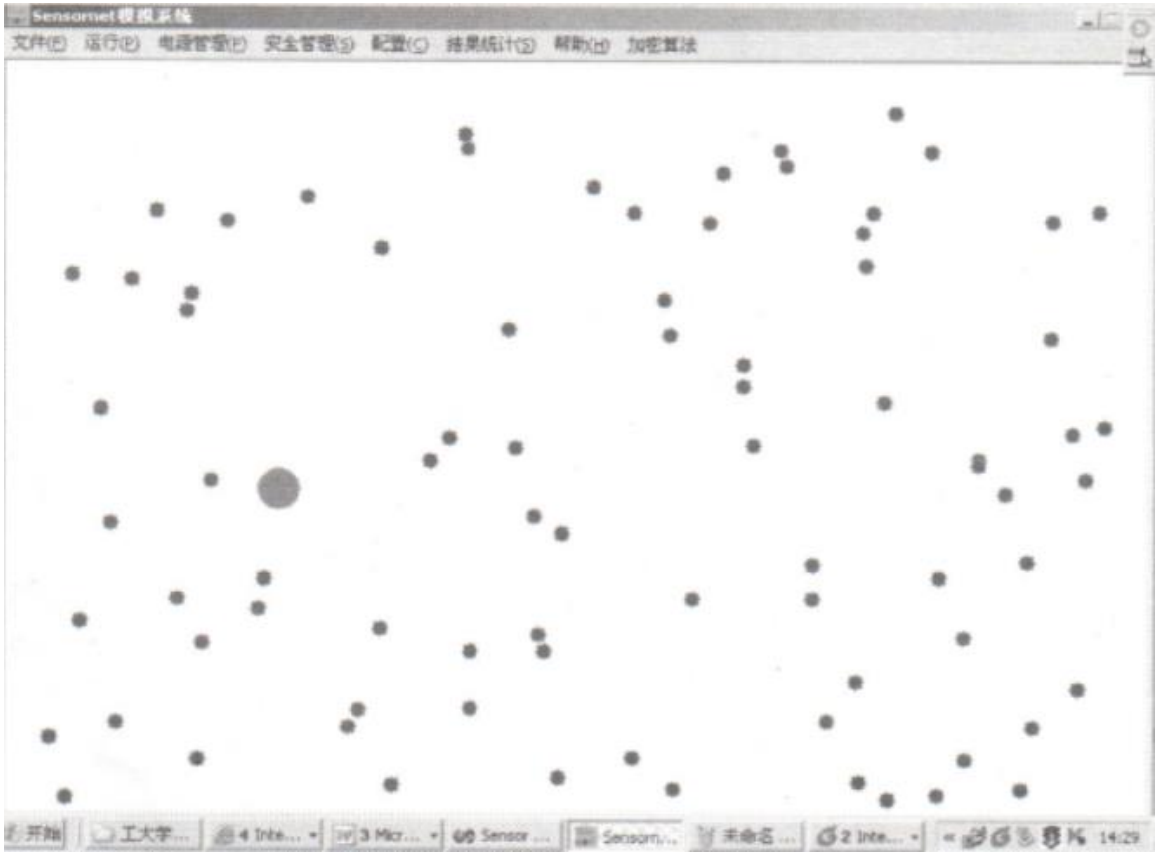
编码实现：核心算法实现（2）

- 可视化的实现

```
void CSensornetworkView:: OnDraw(CDC *pDC)
{
    switch(CConfigure::getCurrentState()) //取出当前的状态
        {
            int SensorNum = Cconfigure::getSensorNum();
            case BROADING:
            {
                DisplayAllSensors(pDC, SensorNum); break;
             }
            case NETWORKING:
            {
                DisplayAllSensors(pDC, SensorNum);
                DrawTheNet(pDC, SensorNum); break;
            }
            case COMMUNICATING:
            {
                DisplayAllSensors(pDC, SensorNum);
                DrawTheSimulation(pDC, SensorNum); break;
            }
        }
}
```

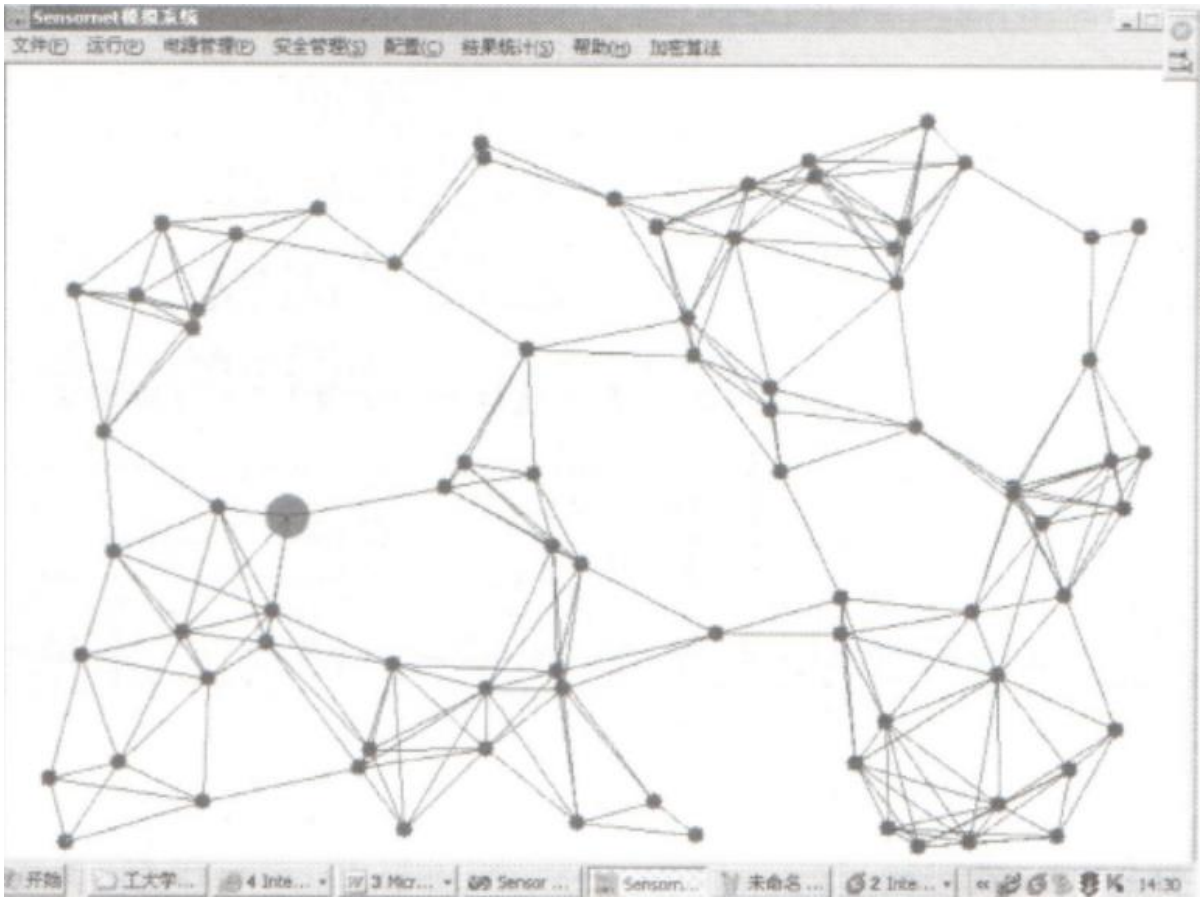
结果测试（1）

- **Sensor散播后的结果：**



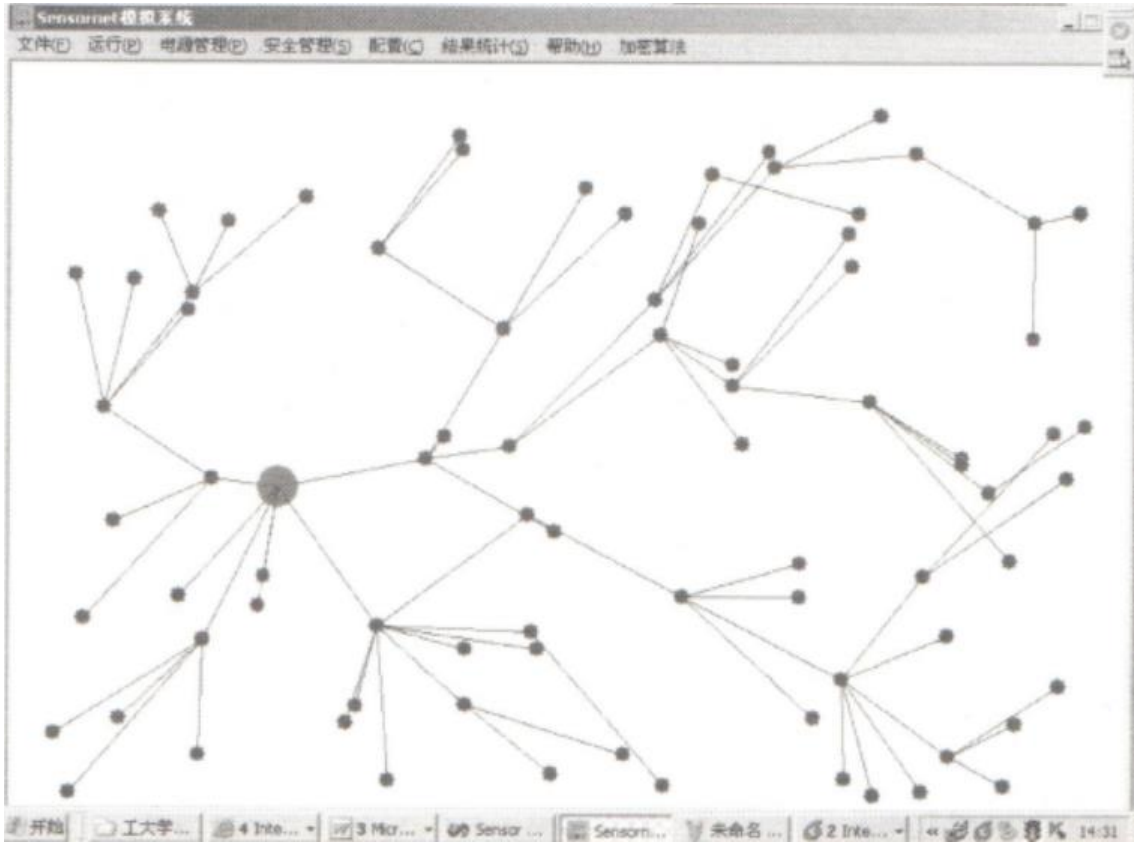
结果测试（2）

- **Sensor随机拓扑后建立的路由结果**



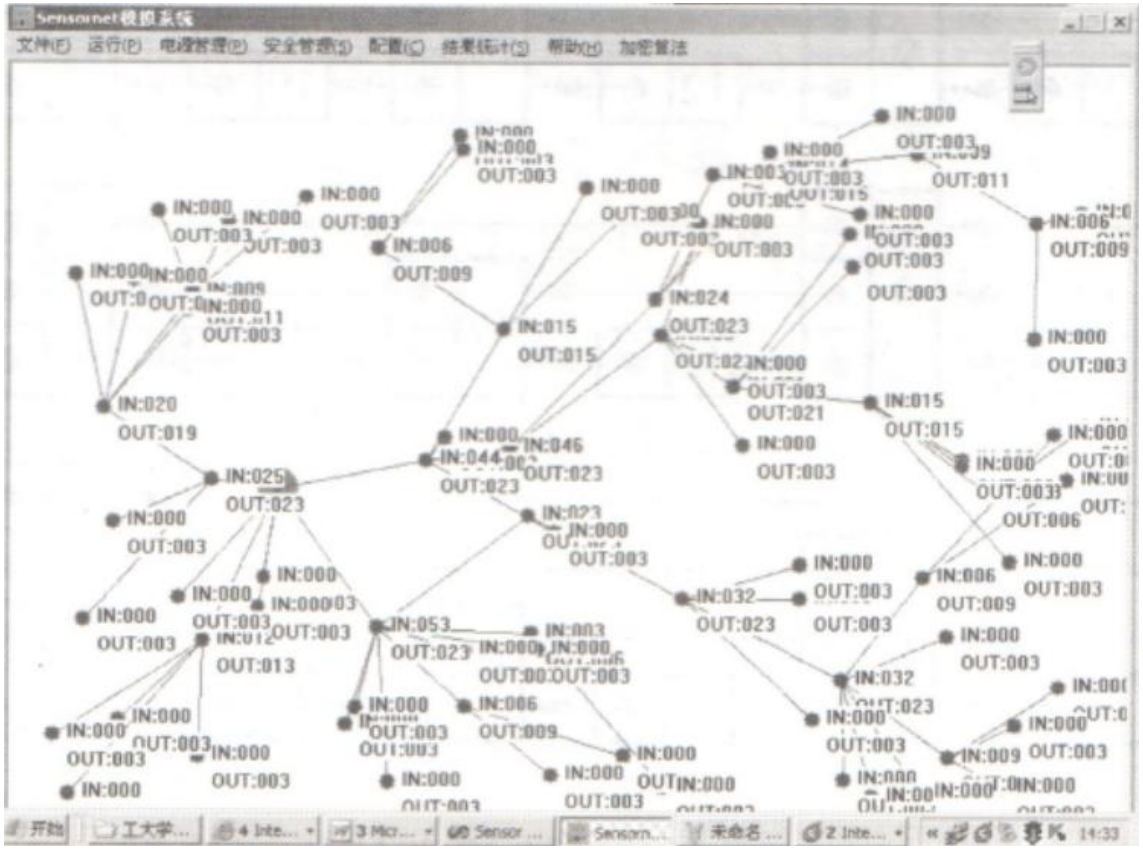
结果测试（3）

- **Sensor分层拓扑建立的路由结果：**



结果测试（4）

- **Sensor网络开始工作后的结果：**



小结

- 主要介绍了用单机来模拟传感器网络工作的开发案例
- 基本从软件设计开发的整个流程介绍了该系统的设计和开发步骤
- 该案例系统用到了集合、图、队列、线性表等数据结构
- 每组可仿照该过程对其选题进行分析、设计和开发

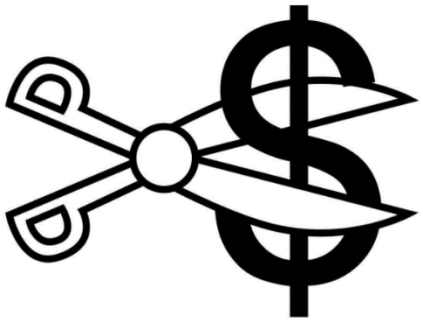
内容安排

- 课程介绍和任务书下达
- 软件设计和开发的基本流程
- 案例1：文本压缩和检索软件
- 案例2：传感器网络单机模拟系统的设计和实现
- 编码规范介绍
- 编写可维护代码的十大原则

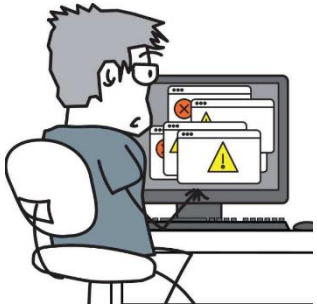
编码规范的重要性

- 规范的代码可以促进团队合作

- 规范的代码可以降低维护成本



- 规范的代码可以减少bug处理



- 规范的代码有助于代码审查

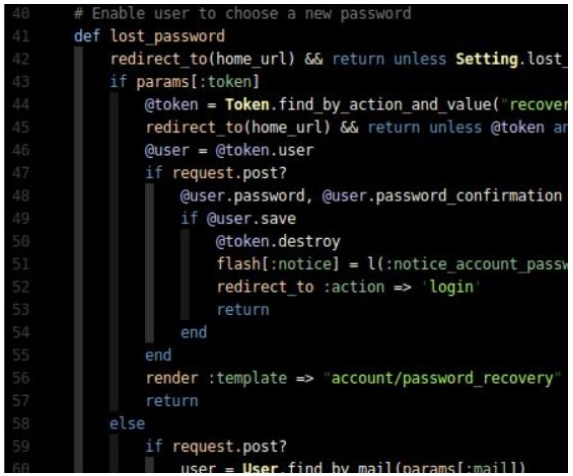
- 养成代码规范的习惯非常重要

如何做到代码规范？

- 排版
- 注释
- 标识符命名
- 可读性
- 变量、结构
- 函数、过程

排版

- 程序块要采用缩进对齐



```
40    # Enable user to choose a new password
41    def lost_password
42        redirect_to(home_url) && return unless Setting.lost_
43        if params[:token]
44            @token = Token.find_by_action_and_value("recover
45            redirect_to(home_url) && return unless @token ar
46            @user = @token.user
47            if request.post?
48                @user.password, @user.password_confirmation
49                if @user.save
50                    @token.destroy
51                    flash[:notice] = l(:notice_account_passw
52                    redirect_to :action => 'login'
53                    return
54                end
55            end
56            render :template => "account/password_recovery"
57            return
58        else
59            if request.post?
60                user = User.find_by_mail(params[:mail])
```

- 不要一行写多条语句

如 rect.length = 0； srect.width = 0;

改为： rect.length = 0;

rect.width = 0;

- 相对独立的代码块之间要加空行

```
if(!valid_ni(ni))
{
    … // program code block A
}

repssn_ni = ssn_data[index].ni;
index = index + 1;
// code block B
```

- 操作符前后加空格

```
a = b + c;
b = c ^ 2;
```

注释

- 注释的原则是有助于对程序的理解，在该加的地方加，注释语言必须准确、易懂、简洁

- TODO注释

- 用代码来阐述

```
//check to see if the employee is eligible
    for full benefits
if(employee.flags & HOURLY_FLAG)
    && (employee.age >65))
```

还是这个？

```
if(employee.isEligibleForFullBenefits())
```

能用函数和变量名说明时就不要用注释！

标识符命名

- 标识符的命名要清晰明了、有明确的含义

- 名副其实

  int d; // 消逝的时间, 以日计

  名称d什么也没有说明，依赖于注释解释。我们应选择如下的名称：

  ```
  int   elapsedTimeInDays;
  int   daysSinceCreation;
  int   daysSinceModification;
  int   fileAgeIndays;
  ```

- 做有意义的区分

```
public static void copyChars(char a1[], char a2[])
{
    for (int i =0；  i < a1.length; i++)
    {
          a2[i] = a1[i];
     }
}
```

这里a1改为source，a2改为destination会更好！

可读性

- 注意运算符的优先级，并用括号说明正确的表达顺序，避免使用默认的优先级产生误读

word = high < < 8 | low ⟶ word = (high < < 8) | low

if (a | b && a & c) ⟶ if ((a | b) && (a & c))

- 避免使用不易解释的数字，用有意义的标识来代替

```
if(Trunk[index]. trunk_state == 0）{
        Trunk[index].trunk_state = 1;
}
```

改为：

```
#define  TRUNK_IDLE   0
#define   TRUNK_BUSY   1
if(Trunk[index]. trunk_state == TRUNK_IDEL）{
        Trunk[index].trunk_state =  TRUNK_BUSY;
}
```

- 避免使用难懂的技巧性高的语句

*stat_poi ++ += 1; ⟶ *stat_poi + = 1;

stat_poi ++ ;

变量、结构

- 去掉没有必要的公共变量，降低耦合度

- 构造仅有一个模块或函数可以修改、创建、而其余有关模块或函数只访问的公共变量，防止多处创建、修改同一公共变量

函数、过程

- 防止函数的参数作为工作变量

```
 void sum_data( unsigned int num, int *data, int* sum)
{
    unsigned int count;
    *sum=0;
    for (count = 0; counter < num; counter ++)
    {
        *sum + = data[count]; //sum 成为工作变量，不太好
    }
}
```

有可能错误改变地址参数sum中的内容！

小结

- 介绍编码的基本规范
  - ✓ 排版
  - ✓ 注释
  - ✓ 标识符命名
  - ✓ 可读性
  - ✓ 变量、结构
  - ✓ 函数、过程

内容安排

- 课程介绍和任务书下达
- 软件设计和开发的基本流程
- 案例1：文本压缩和检索软件
- 案例2：传感器网络单机模拟系统的设计和实现
- 编码规范介绍
- 编写可维护代码的十大原则

“上医治未病，中医治欲病，下医治已病。”

——源自《黄帝内经》

软件可维护的重要性

- 软件维护的4种方式：
  1. 发现并修复Bug（纠正性维护）
  2. 系统需要去适应操作环境的改变（适应性维护）
  3. 用户的新需求（完善性维护）
  4. 质量改进和Bug预防（预防性维护）

- 低可维护性会对业务造成严重的影响
  1. 用户提出新需求，低可维护软件可能导致交付延期甚至无法满足新需求
  2. 竞争对手可能提前几个月抢占市场
  3. 修改1个Bug可能引入更多的Bug

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“任何人都能编写计算机能理解的代码，而好的程序员编写人能理解的代码。”

——Martin Fowler

原则1：编写短小的代码单元

- 代码单元的长度应该在**15**行代码以内。
- 你应该编写不超过**15**行代码的单元、或者将长单元分解为多个更短的单元。
- 该原则能够提高可维护性的原因在于，短小的代码单元易于理解、测试及重用。

例子：吃豆人游戏 JPacman


```
public void Start()
{
    if (inProgress)
    {
        return;
    }
    inProgress = true;
    // Update observers if player died:
    if (!IsAnyPlayerAlive())
    {
        foreach (LevelObserver o in observers)
        {
            o.LevelLost();
        }
    }
    // Update observers if all pellets eaten:
    if (RemainingPellets() == 0)
    {
        foreach (LevelObserver o in observers)
        {
            o.LevelWon();
        }
    }
}
```

共21行

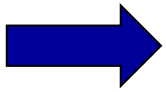
这段代码表示, 如果玩家死亡，我们告诉所有观察者本关结束，如果所有豆子都被吃光，我们告诉所有观察者游戏胜利。

如何使用本原则（1）

- 重构技巧：提取方法

```
public void Start()
{
    if (inProgress)
    {
        return;
    }
    inProgress = true;
    UpdateObservers();
}
// end::start[]
```

```
// tag::updateObservers[]
private void UpdateObservers()
{
    // Update observers if player died:
    if (!IsAnyPlayerAlive())
    {
        foreach (LevelObserver o in observers)
        {
            o.LevelLost();
        }
    }
    // Update observers if all pellets eaten:
    if (RemainingPellets() == 0)
    {
        foreach (LevelObserver o in observers)
        {
            o.LevelWon();
        }
    }
}
```



```
public void UpdateObservers()
{
    UpdateObserversPlayerDied();
    UpdateObserversPelletsEaten();
}
// tag::updateObserversPlayerDied[]
private void UpdateObserversPlayerDied()
{
    if (!IsAnyPlayerAlive())
    {
        foreach (LevelObserver o in observers)
        {
            o.LevelLost();
        }
    }
}

// end::updateObserversPlayerDied[]
```

```
// tag::updateObserversPelletsEaten[]
private void UpdateObserversPelletsEaten()
{
    if (RemainingPellets() == 0)
    {
        foreach (LevelObserver o in observers)
        {
            o.LevelWon();
        }
    }
}
// end::updateObserversPelletsEaten[]
```

如何使用本原则（2）

前面这个例子容易使用“提取方法”的技巧的原因是待提取代码没有使用任何局部变量，也没有任何返回值。下面看JPacman另一个例子：

```
public Board CreateBoard(Square[,] grid)
{
    Debug.Assert(grid != null);

    Board board = new Board(grid);

    int width = board.Width;
    int height = board.Height;
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            Square square = grid[x, y];
            foreach (Direction dir in Direction.Values)
            {
                int dirX = (width + x + dir.DeltaX) % width;
                int dirY = (height + y + dir.DeltaY) % height;
                Square neighbour = grid[dirX, dirY];
                square.Link(neighbour, dir);
            }
        }
    }

    return board;
}
```

共21 行

```
// tag::createBoard[]
public Board CreateBoard(Square[,] grid)
{
    Debug.Assert(grid != null);

    Board board = new Board(grid);

    int width = board.Width;
    int height = board.Height;
    for (int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            Square square = grid[x, y];
            foreach (Direction dir in Direction.Values)
            {
                SetLink(square, dir, x, y, width, height, grid);
            }
        }
    }

    return board;
}
// end::createBoard[]

// tag::setLink[]
private void SetLink(Square square, Direction dir, int x, int y,
    int width, int height, Square[,] grid)
{
    int dirX = (width + x + dir.DeltaX) % width;
    int dirY = (height + y + dir.DeltaY) % height;
    Square neighbour = grid[dirX, dirY];
    square.Link(neighbour, dir);
}
// end::setLink[]
```

传递参数太多，不好！

如何使用本原则（3）

- 重构技巧：将方法替换为方法对象，创建一个新类来替代方法

```
internal class BoardCreator
{
    private Square[,] grid;
    private Board board;
    private int width;
    private int height;

    internal BoardCreator(Square[,] grid)
    {
        Debug.Assert(grid != null);
        this.grid = grid;
        this.board = new Board(grid);
        this.width = board.Width;
        this.height = board.Height;
    }

    internal Board Create()
    {
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                Square square = grid[x, y];
                foreach (Direction dir in Direction.Values)
                {
                    SetLink(square, dir, x, y);
                }
            }
        }
        return this.board;
    }

    private void SetLink(Square square, Direction dir, int x, int y)
    {
        int dirX = (width + x + dir.DeltaX) % width;
        int dirY = (height + y + dir.DeltaY) % height;
        Square neighbour = grid[dirX, dirY];
        square.Link(neighbour, dir);
    }
}
```

在这个新类中，CreateBoard方法的局部变量(board、width和height)和一个参数(grid)都变成了私有变量，不需要传参了。

基于这个重构，我们需将原有的CreateBorad方法修改如下：

```
public Board CreateBoard(Square[,] grid)
{
    return new BoardCreator(grid).Create();
}
```

常见反对意见

- 反对意见1：代码单元过多会影响性能

  不要牺牲可维护性来优化性能，除非有可靠的性能测试证明确实存在性能问题，并且优化措施也真的有效果。

- 反对意见2：代码分散开后难以阅读

  不会！为你的继任者（也为了将来的自己）编写易于阅读和理解的代码。

- 反对意见3：代码单元无法拆分了

  当似乎可以重构但是并没有什么意义时，请慎重考虑系统的架构。

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

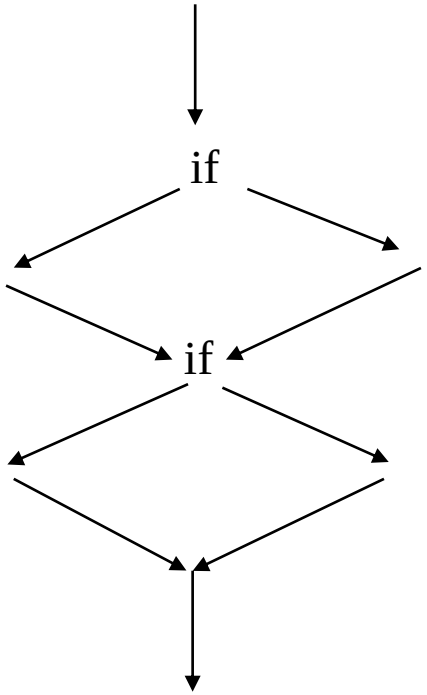
“每个问题的内部都有很多更小的问题。”

——Martin Fowler

原则2：编写简单的代码单元

- 限制每个代码单元分支点的数量不超过4个。
- 应该将复杂的代码单元拆分成多个简单的单元，避免多个复杂的单元在一起。
- 该原则能够提高可维护性的原因在于，分支点越少，代码越容易被修改和测试。

例子：20个控制分支的程序，如何修改？



不易测试，需要很多测试用例。很难修改

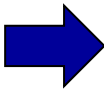
```
private static User getOrCreate(string id, string fullName, bool create)
{
    string idkey = idStrategy().keyFor(id);

    byNameLock.readLock().doLock();
    User u;
    try
    {
        u = byName.get(idkey);
    }
    finally
    {
        byNameLock.readLock().unlock();
    }
    FileInfo configFile = getConfigFileFor(id);
    if (!configFile.Exists && !Directory.Exists(configFile.Directory.FullName))
    {
        // check for legacy users and migrate if safe to do so.
        FileInfo[] legacy = getLegacyConfigFilesFor(id);
        if (legacy != null && legacy.Length > 0)
        {
            foreach (FileInfo legacyUserDir in legacy)
            {
                XmlFile legacyXml = new XmlFile(XmlFile.XSTREAM,
                                        new FileInfo(Path.Combine(
                                        legacyUserDir.FullName, "config.xml")));
                try
                {
                    object o = legacyXml.read();
                    if (o is User)
                    {
                        if (idStrategy().equals(id, legacyUserDir.Name)
                            && !idStrategy()
                            .filenameOf(legacyUserDir.Name)
                            .Equals(legacyUserDir.Name))
                        {
                            try
                            {
                                File.Move(legacyUserDir.FullName,
                                    configFile.Directory.FullName);
                            }
                            catch (IOException)
                            {
                                LOGGER.log(Level.WARNING,
                                    "Failed to migrate user record from {0} " +
                                    "to {1}", new Object[] {legacyUserDir,
                                        configFile.Directory.FullName
                                    });
                            }
                            break;
                        }
                        ...
```

来自开源项目Jenkins(用Java开发的持续集成工具)

如何使用本原则（1）

- 处理链条式条件语句 （分支点的数目不超过4个）

```
public IList<Color> GetFlagColors(Nationality nationality)
{
    List<Color> result;
    switch (nationality)
    {
        case Nationality.DUTCH:
            result = new List<Color> { Color.Red, Color.White, Color.Blue };
            break;
        case Nationality.GERMAN:
            result = new List<Color> { Color.Black, Color.Red, Color.Yellow };
            break;
        case Nationality.BELGIAN:
            result = new List<Color> { Color.Black, Color.Yellow, Color.Red };
            break;
        case Nationality.FRENCH:
            result = new List<Color> { Color.Blue, Color.White, Color.Red };
            break;
        case Nationality.ITALIAN:
            result = new List<Color> { Color.Green, Color.White, Color.Red };
            break;
        case Nationality.UNCLASSIFIED:
        default:
            result = new List<Color> { Color.Gray };
            break;
    }
    return result;
}
```



```
public IList<Color> GetFlagColors(Nationality nationality)
{
    List<Color> result;
    switch (nationality)
    {

        case Nationality.DUTCH:
            result = new List<Color> { Color.Red, Color.White, Color.Blue };
        case Nationality.LUXEMBOURGER:
            result = new List<Color> { Color.Red, Color.White, Color.LightBlue };
            break;
        case Nationality.GERMAN:
            result = new List<Color> { Color.Black, Color.Red, Color.Yellow };
            break;
        case Nationality.BELGIAN:
            result = new List<Color> { Color.Black, Color.Yellow, Color.Red };
            break;
        case Nationality.FRENCH:
            result = new List<Color> { Color.Blue, Color.White, Color.Red };
            break;
        case Nationality.ITALIAN:
            result = new List<Color> { Color.Green, Color.White, Color.Red };
            break;
        case Nationality.UNCLASSIFIED:
        default:
            result = new List<Color> { Color.Gray };
            break;
    }
    return result;
}
```

不容易测试，需要6个用例才能覆盖；容易引入Bug，例如现在要加入卢森堡的国旗

如何使用本原则（2）

- 处理链条式条件语句 （分支点的数目不超过4个）

```
public IList<Color> GetFlagColors(Nationality nationality)
{
    List<Color> result;
    switch (nationality)
    {
        case Nationality.DUTCH:
            result = new List<Color> { Color.Red, Color.White, Color.Blue };
            break;
        case Nationality.GERMAN:
            result = new List<Color> { Color.Black, Color.Red, Color.Yellow };
            break;
        case Nationality.BELGIAN:
            result = new List<Color> { Color.Black, Color.Yellow, Color.Red };
            break;
        case Nationality.FRENCH:
            result = new List<Color> { Color.Blue, Color.White, Color.Red };
            break;
        case Nationality.ITALIAN:
            result = new List<Color> { Color.Green, Color.White, Color.Red };
            break;
        case Nationality.UNCLASSIFIED:
        default:
            result = new List<Color> { Color.Gray };
            break;
    }
    return result;
}
```

引入Map数据结构

```
private static Dictionary<Nationality, IList<Color>> FLAGS =
    new Dictionary<Nationality, IList<Color>>();

static FlagFactoryWithMap()
{
    FLAGS[Nationality.DUTCH] = new List<Color>{ Color.Red, Color.White,
        Color.Blue };
    FLAGS[Nationality.GERMAN] = new List<Color>{ Color.Black, Color.Red,
        Color.Yellow };
    FLAGS[Nationality.BELGIAN] = new List<Color>{ Color.Black, Color.Yellow,
        Color.Red };
    FLAGS[Nationality.FRENCH] = new List<Color>{ Color.Blue, Color.White,
        Color.Red };
    FLAGS[Nationality.ITALIAN] = new List<Color>{ Color.Green, Color.White,
        Color.Red };
}

public IList<Color> GetFlagColors(Nationality nationality)
{
    IList<Color> colors = FLAGS[nationality];
    return colors ?? new List<Color> { Color.Gray };
}
```

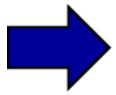
如何使用本原则（3）

- 处理嵌套条件语句

（1）使用卫语句来代替嵌套的条件语句

```
public static int CalculateDepth(BinaryTreeNode<int> t, int n)
{
    int depth = 0;
    if (t.Value == n)
    {
        return depth;
    }
    else
    {
        if (n < t.Value)
        {
            BinaryTreeNode<int> left = t.Left;
            if (left == null)
            {
                throw new TreeException("Value not found in tree!");
            }
            else
            {
                return 1 + CalculateDepth(left, n);
            }
        }
        else
        {
            BinaryTreeNode<int> right = t.Right;
            if (right == null)
            {
                throw new TreeException("Value not found in tree!");
            }
            else
            {
                return 1 + CalculateDepth(right, n);
            }
        }
    }
}
```

在二分查找树中找一个数，返回深度



```
public class BinaryTreeSearch
{

    // tag::calculateDepth[]
    public static int CalculateDepth(BinaryTreeNode<int> t, int n)
    {
        int depth = 0;
        if (t.Value == n)
        {
            return depth;
        }
        if ((n < t.Value) && (t.Left != null))
        {
            return 1 + CalculateDepth(t.Left, n);
        }
        if ((n > t.Value) && (t.Right != null))
        {
            return 1 + CalculateDepth(t.Right, n);
        }
        throw new TreeException("Value not found in tree!");
    }
    // end::calculateDepth[]
}
```

代码更容易理解了，但是复杂度没有降低。

如何使用本原则（4）

- 处理嵌套条件语句

（2）重构技巧：将嵌套条件提取到其他方法中

现在方法的复杂度才降低，我们完成了两件事：让方法更容易理解，且更容易测试。

```
public static int CalculateDepth(BinaryTreeNode<int> t, int n)
{
    int depth = 0;
    if (t.Value == n)
    {
        return depth;
    }
    else
    {
        return TraverseByValue(t, n);
    }
}

private static int TraverseByValue(BinaryTreeNode<int> t, int n)
{
    BinaryTreeNode<int> childNode = GetChildNode(t, n);
    if (childNode == null)
    {
        throw new TreeException("Value not found in tree!");
    }
    else
    {
        return 1 + CalculateDepth(childNode, n);
    }
}

private static BinaryTreeNode<int> GetChildNode(
    BinaryTreeNode<int> t, int n)
{
    if (n < t.Value)
    {
        return t.Left;
    }
    else
    {
        return t.Right;
    }
}
```

常见反对意见

- 反对意见1：高复杂度不可避免

  “我们的业务领域非常复杂，因此代码的复杂度不可避免的会很高。”

  然而，复杂领域并不一定要求技术实现也是复杂的。开发人员的责任就是简化问题，编写简单的代码。

- 反对意见2：拆分方法并不会降低复杂度

  但拆分可以让程序边的更容易理解，也容易测试。

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“代码中排名第一的坏味道就是代码重复。”

——Kent Benk 和Martin Fowler

原则3：不写重复代码

- 不要复制代码
- 你应该编写可重用的、通用的代码，并且/或者调用已有的代码。
- 该原则能够提高可维护性的原因在于，如果复制代码，就需要在多个地方修复Bug，这样做不仅低效，而且容易出错。

例子：管理银行账户

```
public class CheckingAccount
{
    private int transferLimit = 100;

    public Transfer MakeTransfer(String counterAccount, Money amount)
    {
        // 1. Check withdrawal limit:
        if (amount.GreaterThan(this.transferLimit))
        {
            throw new BusinessException("Limit exceeded!");
        }
        // 2. Assuming result is 9-digit bank account number, validate 11-test:
        int sum = 0;
        for (int i = 0; i < counterAccount.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(
                counterAccount[i]);
        }
        if (sum % 11 == 0)
        {
            // 3. Look up counter account and make transfer object:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount);
            return result;
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }
}
```

这个类是支票账户，通过Transfer类的对象，
表示钱在不同账户之间的流转过程

```
public class SavingsAccount
{
    public CheckingAccount RegisteredCounterAccount { get; set; }

    public Transfer makeTransfer(string counterAccount, Money amount)
    {
        // 1. Assuming result is 9-digit bank account number, validate 11-test:
        int sum = 0; // <1>
        for (int i = 0; i < counterAccount.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(
                counterAccount[i]);
        }
        if (sum % 11 == 0)
        {
            // 2. Look up counter account and make transfer object:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount); // <2>
            // 3. Check whether withdrawal is to registered counter account:
            if (result.CounterAccount.Equals(this.RegisteredCounterAccount))
            {
                return result;
            }
            else
            {
                throw new BusinessException("Counter-account not registered!");
            }
        }
        else
        {
            throw new BusinessException("Invalid account number!!");
        }
    }
}
```

新增一个账户类型，储蓄账户，没有转账限额，
但只能转给（固定）支票账户

重复的代码

一般认为超过6行的相同代码段为重复代码。

重复代码的问题：如果这段代码有bug，那么bug也被复制了；如果这段代码要维护、要修改，你不得不检测所有重复的地方。

我们应该抵制以复制代码来获得短期利益的诱惑，等你将来需要对某段重复代码调整时，就不得不重新检查所有重复的地方。

如何使用本原则（1）

- “提取方法”的重构方法，但该方法要在CheckingAccount和SavingAccount 两个类使用，因此可引入一个工具类**Accounts**，在其中加入静态方法IsValid：

```
public static class Accounts
{

    public static CheckingAccount FindAcctByNumber(string number)
    {
        return new CheckingAccount();
    }

    // tag::isValid[]
    public static bool IsValid(string number)
    {
        int sum = 0;
        for (int i = 0; i < number.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(number
        }
        return sum % 11 == 0;
    }
    // end::isValid[]
}
```

```
public class CheckingAccount
{
    private int transferLimit = 100;

    public Transfer MakeTransfer(string counterAccount, Money amount)
    {
        // 1. Check withdrawal limit:
        if (amount.GreaterThan(this.transferLimit))
        {
            throw new BusinessException("Limit exceeded!");
        }
        if (Accounts.IsValid(counterAccount))
        { // <1>
            // 2. Look up counter account and make transfer object:
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            Transfer result = new Transfer(this, acct, amount); // <2>
            return result;
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }
}
```

C#中的方法必须位于类中，所以我们不得不将提取的代码放入另个类中。这个类很快会变成一堆无关方法的大杂烩，从而导致体积过大耦合过紧。

如何使用本原则（2）

- 重构技巧：提取父类

```
public class Account
{
    public Transfer MakeTransfer(string counterAccount, Money amount)
    {
        if (IsValid(counterAccount))
        {
            CheckingAccount acct = Accounts.FindAcctByNumber(counterAccount);
            return new Transfer(this, acct, amount);
        }
        else
        {
            throw new BusinessException("Invalid account number!");
        }
    }

    public static bool IsValid(string number)
    {
        int sum = 0;
        for (int i = 0; i < number.Length; i++)
        {
            sum = sum + (9 - i) * (int)Char.GetNumericValue(number[i]);
        }
        return sum % 11 == 0;
    }
}
```

```
public class CheckingAccount : Account
{
    private int transferLimit = 100;

    public override Transfer MakeTransfer(string counterAccount, Money amount)
    {
        if (amount.GreaterThan(this.transferLimit))
        {
            throw new BusinessException("Limit exceeded!");
        }
        return base.MakeTransfer(counterAccount, amount);
    }
}
```

```
public class SavingsAccount : Account
{
    public CheckingAccount RegisteredCounterAccount { get; set; }

    public override Transfer MakeTransfer(string counterAccount, Money amount)
    {
        Transfer result = base.MakeTransfer(counterAccount, amount);
        if (result.CounterAccount.Equals(this.RegisteredCounterAccount))
        {
            return result;
        }
        else
        {
            throw new BusinessException("Counter-account not registered!");
        }
    }
}
```

常见的反对意见

- 应该允许直接从其他代码库复制代码
  - 如果另一个代码库仍然在维护中，你会遇到麻烦。
  - 如果另一个代码库不再维护，那么你就是在重写这个代码库。
- 由于细微修改而导致的代码重复是不可避免的
  - 系统经常会包含一些略微不同的通用方法，但这并不意味不可避免，你需要找到这些代码的相同部分，然后将它们移入一个父类。
- 这些代码永远不会发生变化
  - 系统的功能性需求可能发生变化
  - 系统所处的环境技术也可能改变，例如操作系统、第三方库等
  - 代码自身也可能因为**Bug**、重构甚至界面优化而发生改变

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“应该将四处分散的数据封装到一个专门的对象中。”

——Martin Fowler

原则4：保持代码单元的接口简单

- 限制每个代码单元的参数不能超过4个
- 你应该将多个参数提取成对象
- 该原则能提高可维护性的原因在于，较少的参数可以让代码单元更容易理解和重用

例子：吃豆人游戏JPacman

- 在JPacman项目中，BoardPanel类的render方法，就是一个拥有许多参数的典型示例。这个方法会在由x,y,w,h四个参数表示的矩形中，绘制一个方块及方块的占有者（例如，表示一个幽灵或者一个豌豆）


```
/// <summary>
/// Renders a single square on the given graphics context on the specified
/// rectangle.
///
/// <param name="square">The square to render.</param>
/// <param name="g">The graphics context to draw on.</param>
/// <param name="x">The x position to start drawing.</param>
/// <param name="y">The y position to start drawing.</param>
/// <param name="w">The width of this square (in pixels.)</param>
/// <param name="h">The height of this square (in pixels.)</param>
private void Render(Square square, Graphics g, int x, int y, int w, int h)
{
    square.Sprite.Draw(g, x, y, w, h);
    foreach (Unit unit in square.Occupants)
    {
        unit.Sprite.Draw(g, x, y, w, h);
    }
}
```

引入参数对象

```
public class Rectangle
{
    public Point Position { get; set; }

    public int Width { get; set; }

    public int Height { get; set; }

    public Rectangle(Point position, int width, int height)
    {
        this.Position = position;
        this.Width = width;
        this.Height = height;
    }

}
```

在render函数利用对象Rectangle

```
private void Render(Square square, Graphics g, Rectangle r)
{
    Point position = r.Position;
    square.Sprite.Draw(g, position.X, position.Y, r.Width, r.Height);
    foreach (Unit unit in square.Occupants)
    {
        unit.Sprite.Draw(g, position.X, position.Y, r.Width, r.Height);
    }
}
```

在render函数利用对象Rectangle

```
private void Render(Square square, Graphics g, Rectangle r)
{
    Point position = r.Position;
    square.Sprite.Draw(g, r);
    foreach (Unit unit in square.Occupants)
    {
        unit.Sprite.Draw(g, r);
    }
}
```

如何使用本原则（1）

- 过长的接口本身不是主要问题，而是实际问题的一个表现——意味着代码可能存在着设计不合理的数据模型

```
public void DoBuildAndSendMail(MailMan m, string firstName, string lastName,
    string division, string subject, MailFont font, string message1,
    string message2, string message3)
{
    // Format the email address
    string mId = $"{firstName[0]}.{lastName.Substring(0, 7)}" +
        $"@{division.Substring(0, 5)}.compa.ny";
    // Format the message given the content type and raw message
    MailMessage mMessage = FormatMessage(font,
        message1 + message2 + message3);
    // Send message
    m.Send(mId, subject, mMessage);
}
```

9个参数，猜猜它要做什么？5各参数组合起来竟然是邮件的内容

如何使用本原则（2）

- “引入参数对象”的重构方法

```
public void DoBuildAndSendMail(MailMan m, MailAddress mAddress,
    MailBody mBody)
{
    // Build the mail
    Mail mail = new Mail(mAddress, mBody);
    // Send the mail
    m.SendMail(mail);
}
```

```
public class MailBody
{
    public string Subject { get; set; }
    public MailMessage Message { get; set; }

    public MailBody(string subject, MailMessage message)
    {
        this.Subject = subject;
        this.Message = message;
    }
}
```

```
public class Mail
{
    public MailAddress Address { get; set; }
    public MailBody Body { get; set; }

    public Mail(MailAddress mAddress, MailBody mBody)
    {
        this.Address = mAddress;
        this.Body = mBody;
    }
}
```

```
public class MailAddress
{
    public string MsgId { get; private set; }

    public MailAddress(string firstName, string lastName,
        string division)
    {
        this.MsgId = $"{firstName[0]}.{lastName.Substring(0, 7)}" +
            $"@{division.Substring(0, 5)}.compa.ny";
    }
}
```

新对象实际上代表了所对应领域的一个对象，封装这些类不仅能减少参数的数量，还可以提升代码的重用性能。

如何使用本原则（3）

- 如果方法的各个参数无法组合在一起怎么办？

例如我们正在创建一个画布上的绘制图表的库，例如柱状图和饼图，为了绘制一个好看的图表，你通常需要很多信息，如区域大小、横纵轴的配置信息，以及实际数据等。一种提供这些信息的方式如下：

```
public static void DrawBarChart(Graphics g,
    CategoryItemRendererState state,
    Rectangle graphArea,
    CategoryPlot plot,
    CategoryAxis domainAxis,
    ValueAxis rangeAxis,
    CategoryDataset dataset)
{
    // ..
}
```

如果 现在希望图表库提供默认值怎么办？**一种方式是使用方法重载。**

```
public static void DrawBarChart(Graphics g, CategoryDataset dataset)
{
    Charts.DrawBarChart(g,
        CategoryItemRendererState.DEFAULT,
        new Rectangle(new Point(0, 0), 100, 100),
        CategoryPlot.DEFAULT,
        CategoryAxis.DEFAULT,
        ValueAxis.DEFAULT,
        dataset);
}
```

有问题！这只是其中一种情况，你需要定义一堆像这样的替代方法，并且拥有7个参数的方法依然存在。

如何使用本原则（4）

- “使用方法对象替换方法”的重构方法

```
public class BarChart
{
    private CategoryItemRendererState state = CategoryItemRendererState.DEFAULT;
    private Rectangle graphArea = new Rectangle(new Point(0, 0), 100, 100);
    private CategoryPlot plot = CategoryPlot.DEFAULT;
    private CategoryAxis domainAxis = CategoryAxis.DEFAULT;
    private ValueAxis rangeAxis = ValueAxis.DEFAULT;
    private CategoryDataset dataset = CategoryDataset.DEFAULT;

    public BarChart Draw(Graphics g)
    {
        // ..
        return this;
    }

    public ValueAxis GetRangeAxis()
    {
        return rangeAxis;
    }

    public BarChart SetRangeAxis(ValueAxis rangeAxis)
    {
        this.rangeAxis = rangeAxis;
        return this;
    }

    // More getters and setters.
}
```

原来版本中的静态方法DrawBarChart被替换成了类BarChart中的Draw方法（非静态的）。DrawBarChart的7个参数，6个被转变成了BarChart类的私有成员。所有这些成员都有默认值。可以通过如下的方法，灵活的绘图，而不需额外定义Draw的重载方法。

```
private void ShowMyBarChart()
{
    Graphics g = this.CreateGraphics();
    BarChart b = new BarChart()
        .SetRangeAxis(myValueAxis)
        .SetDataset(myDataset)
        .Draw(g);
}
```

常见的反对意见

- 反对意见：构造参数对象过于复杂

  “对于我新加入的参数对象而言，其构造函数的参数太多了”。

这种情况通常意味着需要在对象的内部在做更细粒度的划分。例如在Render方法重构时，我们没有采用一个四个参数的构造函数，而是将x和y两个参数封装到了一个Point对象。

```
public class Rectangle
{
    public Point Position { get; set; }

    public int Width { get; set; }

    public int Height { get; set; }

    public Rectangle(Point position, int width, int height)
    {
        this.Position = position;
        this.Width = width;
        this.Height = height;
    }

}
```

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“在一个复杂且紧耦合的系统中，事故是无可避免。”

——Charles Perrow的“正常的意外”理论

原则5：分离模块间的关注点

- 避免形成大型的模块，以便达到模块间的松耦合。
- 你应该将不同职责分给不同的模块，并且隐藏接口内部的实现细节。
- 该原则能够提高可维护性的原因在于，相比起紧耦合的代码库来说，对松耦合代码库的修改更容易监督和执行

在一个复杂且紧耦合的系统中，事故无可避免。
Charles Perrow的“正常的意外”理论

紧耦合案例：名为UserService的类

第一次迭代，只有三个方法

```
public class UserService
{
    public User LoadUser(string userId)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public bool DoesUserExist(string userId)
    {
        // ...
        // end::UserService[]
        return true;
        // tag::UserService[]
    }

    public User ChangeUserInfo(UserInfo userInfo)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }
}
```

第二次迭代没有变化，第三次迭代中实现了一个新的需求，用户注册功能，以便接受特定通知，为此三个方法被加入：

```
public class UserService
{
    public User LoadUser(string userId)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public bool DoesUserExist(string userId)
    {
        // ...
        // end::UserService[]
        return true;
        // tag::UserService[]
    }

    public User ChangeUserInfo(UserInfo userInfo)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public List<NotificationType> GetNotificationTypes(User user)
    {
        // ...
        // end::UserService[]
        return new List<NotificationType>();
        // tag::UserService[]
    }

    public void RegisterForNotifications(User user, NotificationType type)
    {
        // ...
    }

    public void UnregisterForNotifications(User user, NotificationType type)
    {
        // ...
    }
}
```

紧耦合案例：名为UserService的类

第四次迭代中，又增加了新的需求，包括用户搜索、锁定用户，以及列举所有锁定用户。

```
public class UserService
{
    public User LoadUser(string userId)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public bool DoesUserExist(string userId)
    {
        // ...
        // end::UserService[]
        return true;
        // tag::UserService[]
    }

    public User ChangeUserInfo(UserInfo userInfo)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public List<NotificationType> GetNotificationTypes(User user)
    {
        // ...
        // end::UserService[]
        return new List<NotificationType>();
        // tag::UserService[]
    }

    public void RegisterForNotifications(User user, NotificationType type)
    {
        // ...
    }

    public void UnregisterForNotifications(User user, NotificationType type)
    {
        // ...
    }

    public List<User> SearchUsers(UserInfo userInfo)
    {
        // ...
        // end::UserService[]
        return new List<User>();
        // tag::UserService[]
    }

    public void BlockUser(User user)
    {
        // ...
    }

    public List<User> GetAllBlockedUsers()
    {
        // ...
        // end::UserService[]
        return new List<User>();
        // tag::UserService[]
    }
}
```

动机

- 小型、松耦合的模块允许开发人员独立进行工作
- 小型、松耦合的模块降低了浏览代码库的难度
- 小型、松耦合的模块避免了让新人感觉到手足无措

如何使用该原则（1）

- 根据不同的关注点拆分类

为了避免类变的越来越大，开发人员必须在类承担超过一个职责时，对类进行拆分。

举例：我们将UserService拆分成三个独立的类

UserService：负责用户的基本服务

UserNotificationService：负责用户注册的通知服务

UserBlockService：负责用户的锁定服务

如何使用该原则（2）

负责用户通知服务

```
public class UserNotificationService
{
    public IList<NotificationType> GetNotificationTypes(User user)
    {
        // ...
        // end::UserNotificationService[]
        return new List<NotificationType>();
        // tag::UserNotificationService[]
    }

    public void Register(User user, NotificationType type)
    {
        // ...
    }

    public void Unregister(User user, NotificationType type)
    {
        // ...
    }
}
```

负责用户锁定服务

```
public class UserBlockService
{
    public void BlockUser(User user)
    {
        // ...
    }

    public IList<User> GetAllBlockedUsers()
    {
        // ...
        // end::UserBlockService[]
        return new List<User>();
        // tag::UserBlockService[]
    }
}
```

负责用户基本服务

```
public class UserService
{
    public User LoadUser(string userId)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public bool DoesUserExist(string userId)
    {
        // ...
        // end::UserService[]
        return true;
        // tag::UserService[]
    }

    public User ChangeUserInfo(UserInfo userInfo)
    {
        // ...
        // end::UserService[]
        return new User();
        // tag::UserService[]
    }

    public IList<User> SearchUsers(UserInfo userInfo)
    {
        // ...
        // end::UserService[]
        return new List<User>();
        // tag::UserService[]
    }
}
```

如何使用该原则（3）

- 隐藏接口背后的特定实现，来达到松耦合的目的

假设我们有一个如下的类，它实现了数码相机的功能，可通过打开或者关闭闪光灯来拍照。

```
public class DigitalCamera
{
    public Image TakeSnapshot()
    {
        // ...
        // end::DigitalCamera[]
        return Image.FromFile("");
        // tag::DigitalCamera[]
    }

    public void FlashLightOn()
    {
        // ...
    }

    public void FlashLightOff()
    {
        // ...
    }
}
```

```
public class SmartphoneApp
{
    private static DigitalCamera camera = new DigitalCamera();

    public static void Main(string[] args)
    {
        // ...
        Image image = camera.TakeSnapshot();
        // ...
    }
}
```

如何使用该原则（4）

假设现在我们有了一个更高级的数码相机，除了拍照，还可以录制视频、定时器并且可以缩放。为此，我们对DigitalCamera类进行了扩展：

```
public class DigitalCamera
{
    public Image TakeSnapshot()
    {
        // ...
        // end::DigitalCamera[]
        return Image.FromF
        // tag::DigitalCame
    }

    public void FlashLightOn()
    {
        // ...
    }

    public void Flash
    {
    }

    public Image T
    {
        //
        return Image.FromFi
        // tag::DigitalCame
        // ...
    }
```

```
    public Video Record()
    {
        // ...
        // en
        re
    }
```

```
    ...
}
```

智能移动设备上还使用着最初只有三个功能的版本。但由于只有一个Digital Camera类，应用不得不使用经过扩展的体庞大的类。紧耦合！！！

如何使用该原则（5）

为了降低耦合程度，我们**使用一个接口**来定义几个基础相机和高级相机都需要实现的功能列表：

```
public interface ISimpleDigitalCamera
{
    Image TakeSnapshot();

    void FlashLightOn();

    void FlashLightOff();
}
```

```
public class DigitalCamera : ISimpleDigitalCamera
{
    // ...
    // end::DigitalCamera[]
    public Image TakeSnapshot()
    {
        return null;
    }

    public void FlashLightOn()
    {
    }

    public void FlashLightOff()
    {
    }
    // tag::DigitalCamera[]
}
```

```
public class SmartphoneApp
{
    private static ISimpleDigitalCamera camera = SDK.GetCamera();

    public static void Main(string[] args)
    {
        // ...
        Image image = camera.TakeSnapshot();
        // ...
    }
}
```

这里我们通过更高层的封装降低了耦合程度。换言之，只有使用基本相机功能的类，不需要知道任何高级数码相机的功能。这保证了SmartphoneApp不会用到任何高级相机的方法。

常见的反对意见

- 反对意见：松耦合与代码重用相冲突

  代码重用并不一定导致方法尽可能的被调用。好的设计——如使用继承和接口——可以即达到代码重用的同时，又保证代码实现之间的松耦合（因为接口隐层了实现细节）。

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“有两种构建软件设计的方式：一种是简单到明显没有的缺陷，另种是复杂到没有明显缺陷。”

——C.A.R Hoare

原则6：架构组件松耦合

- 顶层组件之间应该做到松耦合
- 你应该尽可能减少模块中需要暴露（例如，被调用）给其他组件中模块的相关代码
- 该原则能够提高可维护性的原因在于，独立的组件可以单独进行维护。

如何使用本原则（1）

- 抽象工厂设计模式

抽象工厂设计模式在一个通用的“产品工厂”接口背后，隐藏了具体“产品”的创建过程。

假设我们的代码包含一个PlatformService的组件，它实现了对某个云主机平台服务的管理功能。该组件支持两种具体地云主机平台：Amazon AWS和Microsoft Azure。为了能够启、停服务器，以及预定存储空间，我们实现如下接口：

```
public interface ICloudServerFactory
{
    ICloudServer LaunchComputeServer();

    ICloudServer LaunchDatabaseServer();

    ICloudStorage CreateCloudStorage(long sizeGb);
}
```

如何使用本原则（2）

为了启、停云平台的服务器，以及预定存储空间，我们为两台主机平台分别实现以下接口：

```
public class AWSCloudServerFactory : ICloudServerFactory
{
    public ICloudServer LaunchComputeServer()
    {
        return new AWSComputeServer();
    }

    public ICloudServer LaunchDatabaseServer()
    {
        return new AWSDatabaseServer();
    }

    public ICloudStorage CreateCloudStorage(long sizeGb)
    {
        return new AWSCloudStorage(sizeGb);
    }
}
```

```
public class AzureCloudServerFactory : ICloudServerFactory {
    public ICloudServer LaunchComputeServer() {
        return new AzureComputeServer();
    }

    public ICloudServer LaunchDatabaseServer() {
        return new AzureDatabaseServer();
    }

    public ICloudStorage CreateCloudStorage(long sizeGb) {
        return new AzureCloudStorage(sizeGb);
    }
}
```

PlatformServices组件之外的代码，现在可以按照如下方式来使用接口模块。

```
public class ApplicationLauncher
{

    public static void Main(string[] args)
    {
        ICloudServerFactory factory;
        if (args[1].Equals("-azure"))
        {
            factory = new AzureCloudServerFactory();
        }
        else
        {
            factory = new AWSCloudServerFactory();
        }
        ICloudServer computeServer = factory.LaunchComputeServer();
        ICloudServer databaseServer = factory.LaunchDatabaseServer();
        // end::ApplicationLauncher[]
    }
}
```

PlatformServices的CloudServerFactory接口为其他组件提供了一个小接口，可以形成组件间的松耦合关系。

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

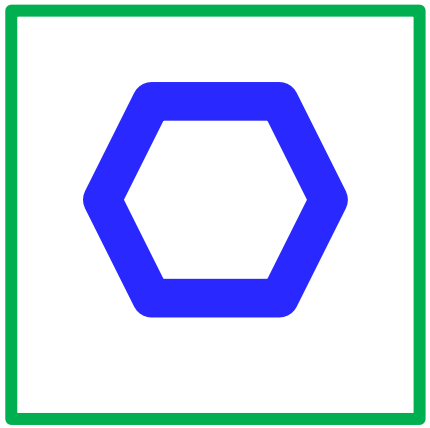
“构建封装边界是设计软件架构的重要技能。”

——George H. Fairbanks

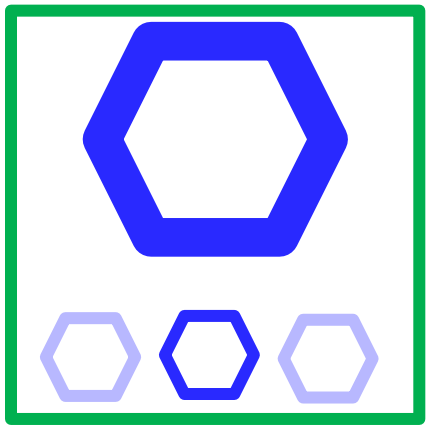
原则7：保持架构组件之间的平衡

- 你需要平衡代码中顶层组件的数量和体积。
- 你应该保持源代码中组件的数量接近与9（例如，在6和12之间），并且这些组件的体积基本一致。
- 该原则能够提高可维护性的原因在于，平衡的组件可以帮助定位代码，并且允许独立组件进行维护。

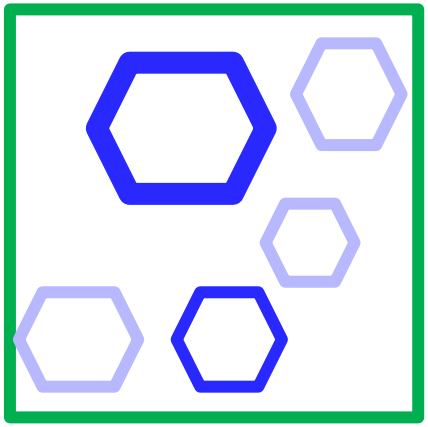
组件平衡



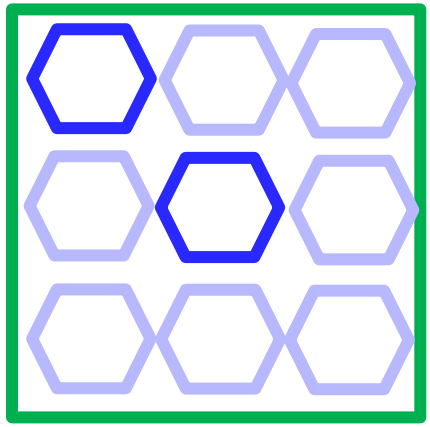
所有的修改发生在一个单独的、巨大的组件当中



大多数修改发生在一个单独的、巨大的组件当中



许多修改散布在多个组件中



对一两个范围有限的组件独立修改

如何使用该原则

- 确定将功能合成组件的合适原则
  1. 按照功能领域去组织：例如资料检索、发票管理、报表、管理员等；
  2. 按照技术专长来划分：例如前端、后端、接口、日志等。

- 明确系统的领域并坚持下去

  一旦确定了系统组件的划分类型，就坚持走下去。

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“程序的复杂会持续增加，直到它超过维护人员的能力。”

——计算机编程的第7条法则

原则8：保持小规模代码库

- 保持代码库规模尽可能小。
- 你应该控制代码增长，并主动减小系统代码的体积。
- 该原则能够提高可维护性的原因在于，拥有小型的代码、项目和团队是成功的一个因素。

动机

- 以大型的代码库为目标的项目更容易失败
- 大型代码库更加难以维护
- 大型系统会出现更密集的缺陷

如何使用本原则

- 功能层面的方法
  1. 控制需求蔓延
  2. 功能标准化

- 技术层面的方法
  1. 不要复制粘贴代码
  2. 重构已有代码
  3. 使用第三方库和框架

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“要想保持代码简洁，请先保持测试进度条是绿色的。”

——JUnit格言

原则9：自动化开发部署和测试

- 对你的代码进行自动化测试
- 你应该通过使用测试框架来编写自动化测试
- 该原则能够提高可维护性的原因在于，自动化测试让开发过程可预测并且能够降低风险

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

“编写简洁的代码是专业开发人员的职责所在。”

——Robert Martin

原则10：编写简洁的代码

- 编写简洁的代码
- 你不应该在完成开发工作后留下代码坏味道
- 该原则能够提高可维护性的原因在于，简洁的代码是可维护的代码

如何使用本原则

1. 不要编写单元级别的坏味道（过长代码单元、复杂代码单元以及长接口代码单元）
2. 不要编写不好的注释（不要用注释来试图美化难懂的代码，应该改变代码本身）
3. 不要注释掉的代码
4. 不要保留废弃代码（无法到达的代码、无用的私有方法、注释的代码）
5. 不要使用过长的标识符名称
6. 不要使用魔术常量（代码中没有清晰定义的数字或字符串）
7. 不要使用未正确处理的异常（捕获一切异常和特定异常）

例子: 魔术常量

```
float CalculateFare(Customer c, long distance)
{
    float travelledDistanceFare = distance * 0.10f;
    if (c.Age < 12)
    {
        travelledDistanceFare *= 0.25f;
    }
    else
        if (c.Age >= 65)
    {
        travelledDistanceFare *= 0.5f;
    }
    return 3.00f + travelledDistanceFare;
}
```

```
private static readonly float BASE_RATE = 3.00f;
private static readonly float FARE_PER_KM = 0.10f;
private static readonly float DISCOUNT_RATE_CHILDREN = 0.25f;
private static readonly float DISCOUNT_RATE_ELDERLY = 0.5f;
private static readonly int MAXIMUM_AGE_CHILDREN = 12;
private static readonly int MINIMUM_AGE_ELDERLY = 65;
```

```
float CalculateFare(Customer c, long distance)
{
    float travelledDistanceFare = distance * FARE_PER_KM;
    if (c.Age < MAXIMUM_AGE_CHILDREN)
    {
        travelledDistanceFare *= DISCOUNT_RATE_CHILDREN;
    }
    else
        if (c.Age >= MINIMUM_AGE_ELDERLY)
    {
        travelledDistanceFare *= DISCOUNT_RATE_ELDERLY;
    }
    return BASE_RATE + travelledDistanceFare;
}
```

编写可维护软件的10大原则

- 原则1：编写短小的代码单元
- 原则2：编写简单的代码单元
- 原则3：不写重复代码
- 原则4：保证单元的接口简单
- 原则5：分离模块之间的关注点
- 原则6：架构组件松耦合
- 原则7：保持架构组件之间的平衡
- 原则8：保证小规模代码库
- 原则9：自动化开发部署和测试
- 原则10：编写简洁的代码

将原则变成实践

- 要想确保你的代码是容易维护的，需要依赖于日常工作的两个行为：**遵守纪律**和**设定优先级**
- 低层级（代码单元）原则要优于高层级（组件）原则
- 对每次提交负责，坚持遵守纪律，看上去更加有效的“权宜之计”会诱导你违反这些原则

谢 谢!