系统设计

|  |
| --- |
| 目标：   * 理解软件系统的设计阶段的重要性； * 了解分析和设计的关联关系和区别； * 掌握主要的设计目标，以及如何确定设计目标； * 掌握包图和组件图两种表述系统架构的方式； * 掌握系统分解的重要概念和方法； * 掌握系统设计原则； * 了解系统设计的基本管理活动； |

系统设计是将分析模型转化为系统设计模型。在系统设计期间，设计人员定义项目的设计目标，并将系统分解为更小的子系统，这些子系统可以由各个团队实现。系统设计的结果是一个模型，其中包括子系统分解和对每个策略的清晰描述。设计人员必须在许多经常相互冲突的设计目标之间做出权衡。他们也无法预测他们将面临的所有设计问题，因为他们还没有清楚地了解解决方案领域。

本章我们通过一个例子来让读者理解房屋的设计和软件的设计两者之间的相似性。然后介绍设计目标，设计目标的主要来源是分析活动中确定的非功能性需求。包图和组件图是用来表达系统设计结果的两个非常有用的UML图，可以用来描述系统的逻辑结构和物理结构。接着讨论系统分解涉及到的相关概念。最后我们描述与系统设计相关的管理问题。

# 设计简介

系统设计、对象设计、实现是构建系统的最主要活动。在这三个活动中，开发人员弥合了在需求获取和分析过程中产生的需求规格与交付给用户的系统之间的差距。系统设计是此过程的第一步，重点是将系统进行分解以便实现。在需求获取和分析期间，我们专注于系统的目的和功能。在系统设计过程中，我们关注实现它所必需的流程、结构以及软件和硬件组件。

系统设计的挑战是在分解系统时必须满足许多相互冲突的标准和约束。例如，考虑一个设计住宅的任务。在与客户就房间和楼层的数量、起居区的大小以及房屋的位置达成一致后，建筑师必须设计平面图，即墙壁、门和窗户的位置，满足一些要求比如：厨房应该靠近餐厅和车库，浴室应该靠近卧室等等。建筑师在确定每个房间的尺寸和门的位置时也可以依赖一些标准：厨柜按固定增量排列，床按标准尺寸排列。但是请注意，建筑师不需要知道每个房间的内部装饰细节和家具的布局；相反，这些决定应该延迟并留给客户。

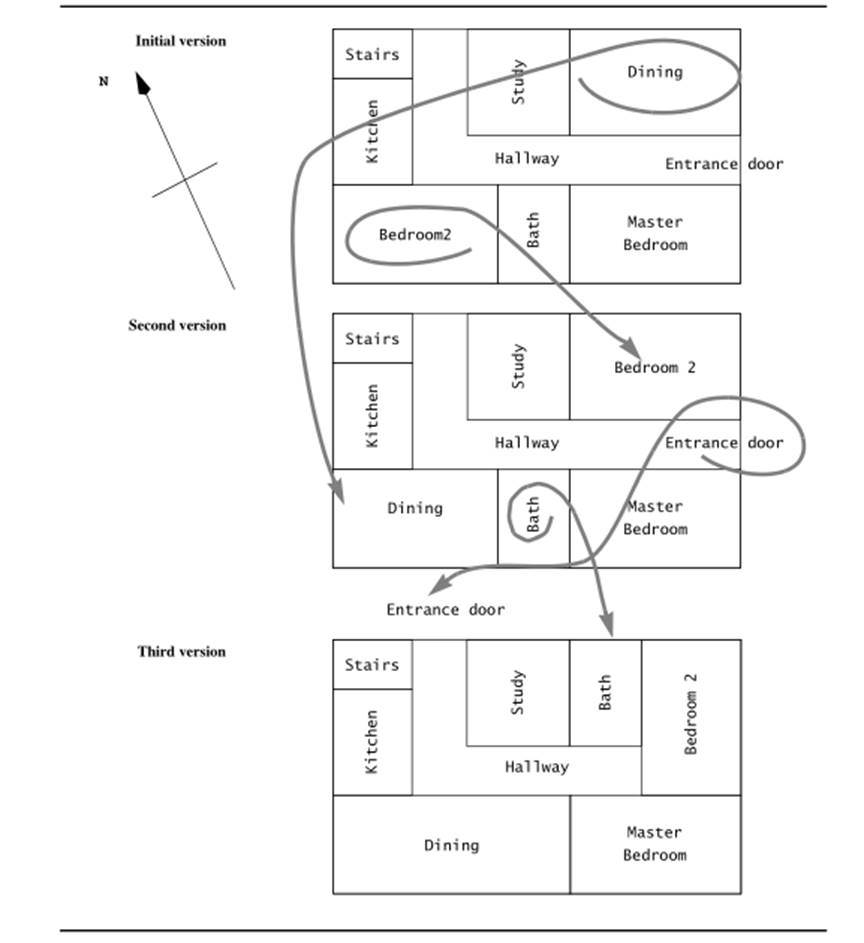


**图**11-0 住宅示例。

假设房屋的设计师需要满足用户以下几个需求设计房屋布局平面图：

* 这套房子应该有两个卧室，一个书房，一个厨房，一个客厅。
* 尽量减少住户每天步行的总距离。
* 采光要好
* 最大化利用空间
* 注重隐私

假设房子的居住者大部分时间都在起居/用餐区和主卧室度过。另外，平面图方向是上北下南左西右东。为了满足上述限制，平面图的第一个版本中，我们发现餐厅离厨房太远。为了解决这个问题，我们将它与卧室2交换。这样也有把客厅搬到房子南墙的好处。第二次改版，我们发现厨房和楼梯离入户门太远了。为了解决这个问题，我们将入口门移到了北墙。这使我们能够重新调整卧室 2 的方向，并将浴室移近两间卧室。居住面积增加了，这样我们满足了所有原来的限制。



**图**11-1 平面图设计示例。三个连续的版本展示了我们如何减少步行距离并利用阳光。

此时，我们可以定位每个房间的门窗，以满足本地化的要求。一旦完成，我们就完成了设计，而无需详细了解每个房间的布局。管道、电线和加热管道的计划可以继续进行。

房屋平面图的设计类似于软件工程中的系统设计。整体分为更简单的组件和接口，同时兼顾非功能和功能需求。系统设计会影响开发活动，如果以后更改会导致代价高昂的返工。

**表**11-1 体系结构和软件工程概念的映射。

|  |  |  |
| --- | --- | --- |
|  | 建筑概念 | 软件工程概念 |
| 组件 | 房间 | 子系统 |
| 接口 | 门窗 | 服务 |
| 非功能性需求 | 光、空间、距离、隐私性 | 响应时间、性能、可靠性 |
| 功能要求 | 睡觉、吃饭、学习 | 用例 |
| 代价高昂的返工 | 门窗改造 | 改变子系统接口 |

## 分析结果

需求模型的分析结果由以下产品描述：

* 一组***非功能性需求和约束***，例如最大响应时间、最小吞吐量、可靠性、操作系统平台等
* ***功能模型***，从参与者的视角描述系统功能；
* ***结构模型***，描述系统中的对象；
* ***行为模型***，显示参与对象之间的交互活动；

分析模型完全从参与者的角度描述系统，并作为客户与开发人员之间沟通的基础。但是，分析模型不包含有关系统内部结构、硬件配置或更一般地应该如何实现系统的信息。系统设计是朝着这个方向迈出的第一步。系统设计产生以下产品：

* ***设计目标***，描述开发人员应该优化的系统质量；
* ***软件架构***，从子系统职责、子系统之间的依赖关系、子系统到硬件的映射等方面描述子系统分解；

设计目标源自非功能性需求。设计目标指导开发人员在需要权衡时做出的决定。子系统分解构成了系统设计的主体。开发人员将系统划分为可管理的部分以应对复杂性：每个子系统都分配给一个团队并独立实现。为此，开发人员需要在分解系统时解决系统范围的问题。

## 设计任务

软件设计的主要任务包括：

  系统设计：将软件划分为子系统和组件，以及决定它们如何连接和交互的过程，包括确定它们的接口。架构设计通常被称为“软件架构”，尽管后者也可以指代生成的文档和整个研究领域。

  对象设计：对于子系统中每个类的细化，包括类中以及类之间各种特性，如关联、属性、交互和状态的设计，甚至包括类的行为的详细描述。

  用户界面设计：侧重于基于网络的界面和使用窗口、菜单、图标和鼠标的图形用户界面（GUI）的设计。

  数据库设计：设计如何持久存储数据，以便许多程序和用户可以无限期地访问它。

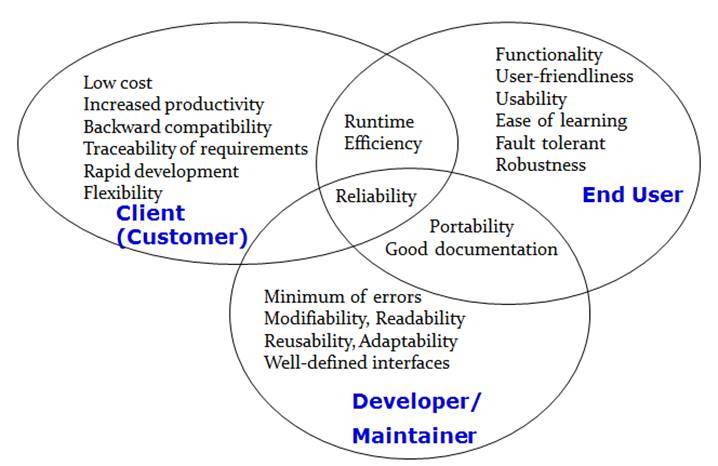
  物理架构设计：设计系统的硬件、软件和网络环境等。

# 确定设计目标

一般来说，我们可以从一长串非常理想的品质中选择设计目标。一些设计目标，例如可靠性、可维护性、易用性、可复用性、性能和信息安全性等，是多数信息系统都需要满足的需求。

## 设计目标

有时，不同的利益相关者可能有不同的设计目标。对于客户（customer）来说，低成本、提高生产力、向后兼容、需求的可追溯性、快速开发、灵活性、运行时效率、可靠性是重要的设计目标。对于最终用户，功能性、用户友好性、可用性、易学性、容错性、健壮性、运行时效率、可移植性、良好的文档、可靠性是重要的设计目标。对于开发人员或维护人员来说，最小错误、可修改性、可读性、可重用性、适应性、定义良好的接口、可移植性、良好的文档、可靠性是重要的设计目标，如图所示。



**图**11-3 利益相关者有不同的设计目标。

表 6-2 到 6-6 列出了一些可能的设计标准。这些标准分为五组：性能、可靠性、成本、维护和最终用户标准。性能、可靠性和最终用户标准通常在需求中指定或从应用程序域推断。成本和维护标准由客户和供应商决定。

**2.1.1    性能标准**

性能标准 （表 6-2）包括对系统的速度和空间要求。系统应该响应，还是应该完成最大数量的任务？是否有可用于速度优化的内存空间，或者应该谨慎使用内存？

**表**11-1 性能标准。

|  |  |
| --- | --- |
| 设计规范 | 定义 |
| 响应时间 | 发出请求后多久会确认用户请求？ |
| 吞吐量 | 系统在固定时间内可以完成多少个任务？ |
| 内存空间 | 系统运行需要多少空间？ |

**2.1.2    可靠性标准**

可靠性标准决定了在最小化系统崩溃及其后果方面应该付出多少努力。系统崩溃的频率是多少？系统对用户的可用性应该如何？系统是否应该容忍错误和故障？系统环境是否存在安全风险？安全问题是否与系统崩溃有关？

**表** 可靠性标准。

|  |  |
| --- | --- |
| 设计规范 | 定义 |
| 鲁棒性 | 能够承受无效的用户输入 |
| 可靠性 | 指定行为和观察到的行为之间的差异 |
| 可用性 | 系统可用于完成正常任务的时间百分比 |
| 容错性 | 在错误条件下操作的能力 |
| 信息安全 | 抵御恶意攻击的能力 |
| 物理安全 | 能够避免危及人的生命，即使存在错误和故障 |

**2.1.3    成本标准**

**成本标准**（表 6-4）包括开发系统、部署系统和管理系统的成本。请注意，成本标准不仅包括设计方面的考虑，还包括管理方面的考虑。当系统更换旧系统时，必须考虑确保向后兼容或过渡到新系统的成本。在开发成本、最终用户培训成本、过渡成本和维护成本等不同类型的成本之间也存在权衡。保持与先前系统的向后兼容性会增加开发成本，同时降低转换成本。

**表**11-1 成本标准。

|  |  |
| --- | --- |
| 设计规范 | 定义 |
| 开发成本 | 开发初始系统的成本 |
| 部署成本 | 安装系统和培训用户的成本 |
| 升级费用 | 从以前的系统转换数据的成本。此标准导致向后兼容性要求 |
| 维修费用 | 错误修复和系统增强所需的成本 |
| 行政成本 | 管理系统所需的成本 |

**2.1.4    维护性标准**

维护性标准决定了部署后更改系统的难度。添加新功能有多容易？修改现有功能有多容易？系统能否适应不同的应用领域？将系统移植到不同的平台需要付出多少努力？这些标准更难优化和规划，因为很少清楚项目将有多成功以及系统将运行多长时间。

**表** 维护标准。

|  |  |
| --- | --- |
| 设计规范 | 定义 |
| 可扩展性 | 向系统添加功能或新类有多容易？ |
| 可修改性 | 改变系统的功能有多容易？ |
| 适应性 | 将系统移植到不同的应用领域有多容易？ |
| 可移植性 | 将系统移植到不同平台有多容易？ |
| 可读性 | 通过阅读代码了解系统有多容易？ |
| 需求的可追溯性 | 将代码映射到特定需求有多容易？ |

**2.1.5    用户标准**

用户标准包括从用户的角度来看需要的质量，但尚未包含在性能和可靠性标准中。该软件是否难以使用和学习？用户能否在系统上完成所需的任务？这些标准通常不会受到太多关注，尤其是当系统的客户与其用户不同时。

**表**最终用户标准。

|  |  |
| --- | --- |
| 设计规范 | 定义 |
| 实用性 | 系统对用户工作的支持程度如何？ |
| 可用性 | 用户使用该系统的难易程度如何？ |

**2.1.6    设计目标权衡**

在定义设计目标时，只能同时考虑这些标准的一小部分。例如，开发安全、可靠和廉价的软件是不现实的。通常，开发人员需要确定设计目标的优先级，并在项目落后于进度或超出预算时将它们相互权衡，并与管理目标进行权衡。下表列出了几种可能的权衡。

**表**11-1 设计目标权衡示例。

|  |  |
| --- | --- |
| 权衡 | 基本原理 |
| 响应时间与内存空间 | 如果软件不满足响应时间或吞吐量要求，则可以扩展更多内存来加速软件。如果软件不满足内存空间限制，考虑以牺牲速度为代价压缩数据。 |
| 可移植性与可读性 | 可以通过引入特殊开关来保证不同平台的特殊处理。也可以使用设计模式保持系统优秀可移植性并仍然保持可读性。 |
| 功能性与易用性 | 当功能太多时，系统就会变得复杂，从而使得用户掌握系统的难度增加。 |
| 性能与可移植性 | 通常为了提高性能，需要使用和系统底层或者硬件结合较紧密的技术，从而损害了可移植性。 |
| 可维护性与性能 | 为提高可维护性，体系结构中不可避免的需要增加一些中间类或者中间层，从而降低了通信成本，降低了性能。 |

## 示例

我们将通过代码来解释重要设计目标的含义，下面是一个简易版成绩查询系统的代码示例：

public class Client {

public static void main(String[] args) {

School school = new School();

System.out.println("Please input student name：");

Scanner scanner = new Scanner(System.in);

String name = scanner.next();

Student student = school.findStudent(name);

System.out.print("Average score of the student: ");

System.out.println(student.getAvgScore());

}

}

class School {

private Student[] students;

public School() {

this.students = new Student[5];

initializeStudents();

}

public void initializeStudents() {

students[0] = new Student("Tom", "20220002", "SE", 92.5, 19);

students[1] = new Student("Alice", "20220034", "CS", 77.2, 21);

students[2] = new Student("Bob", "202200032", "CS", 86.7, 21);

students[3] = new Student("Alice", "20220011", "NE", 93.3, 20);

students[4] = new Student("Spark", "20220078", "SE", 67.8, 18);

}

public Student findStudent(String name) {

for (int i = 0; i < students.length; i++) {

Student student = students[i];

if (student.getName().equals(name)) {

return student;

}

}

return null;

}

}

class Student {

private String name;

private String id;

private String major;

private double avgScore;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getMajor() {

return major;

}

public void setMajor(String major) {

this.major = major;

}

public double getAvgScore() {

return avgScore;

}

public void setAvgScore(double avgScore) {

this.avgScore = avgScore;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public Student(String name, String id, String major, double avgScore, int age) {

this.name = name;

this.id = id;

this.major = major;

this.avgScore = avgScore;

this.age = age;

}

@Override

public String toString() {

return "Student{" +

"name='" + name + '\'' +

", id='" + id + '\'' +

", major='" + major + '\'' +

", avgScore=" + avgScore +

", age=" + age +

'}';

}

}

实际上，上面的简易系统存在着一些可靠性问题。比如假设系统存在同名的同学，那么查询到的可能是其他学生的成绩。所以需要改成使用学号进行查询。主方法修改为：

public static void main(String[] args) {

School school = new School();

System.out.println("Please input student ID：");

Scanner scanner = new Scanner(System.in);

String studentID = scanner.next();

Student student = school.findStudent(studentID);

System.out.print("Average score of the student: ");

System.out.println(student.getAvgScore());

}

现在系统仍然存在可靠性问题，比如用户输入了错误的学号，系统就会报错，而无法正常执行下去，容错性较低。为提高可靠性，确保学生输入错误信息，系统依然能够正常执行，修改代码中的main()方法如下：

public static void main(String[] args) {

School school = new School();

System.out.println("Please input student ID：");

Scanner scanner = new Scanner(System.in);

String studentID = scanner.next();

Student student = school.findStudent(studentID);

while (student == null){

System.out.println("Student ID does not exist, please reinput:");

studentID = scanner.next();

student = school.findStudent(studentID);

}

System.out.print("Average score of the student:");

System.out.println(student.getAvgScore());

}

下面继续观察成绩查询系统，这里面存在安全性问题，用户可以随意查看任何学生的成绩数据，无论是否有授权，因此需要添加一个简单的认证登录过程。我们将修改代码如下：

public class Client {

public static void main(String[] args) {

School school = new School();

Scanner scanner = new Scanner(System.in);

System.out.println("Please input user name：");

String userName = scanner.next();

System.out.println("Please input password：");

String password = scanner.next();

try {

Account account = school.login(userName, password);

if (account.getRole().equals("0")) {

school.sort();

school.printStudents();

}else{

System.out.println("Your average score is:");

Student student = school.findStudent(account.getUserName());

if (student != null){

System.out.println(student.getAvgScore());

}

}

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

class School {

private Student[] students;

private Account[] accounts;

public School() {

this.students = new Student[5];

this.accounts = new Account[6];

initializeStudents();

initializeAccounts();

}

public void initializeStudents() {

students[0] = new Student("Tom", "20220002", "SE", 92.5, 19);

students[1] = new Student("Alice", "20220034", "CS", 77.2, 21);

students[2] = new Student("Bob", "202200032", "CS", 86.7, 21);

students[3] = new Student("Alice", "20220011", "NE", 93.3, 20);

students[4] = new Student("Spark", "20220078", "SE", 67.8, 18);

}

public void initializeAccounts() {

accounts[0] = new Account("Tom", "111111", "1");

accounts[1] = new Account("Alice", "111111", "1");

accounts[2] = new Account("Bob", "111111", "1");

accounts[3] = new Account("Jerry", "111111", "1");

accounts[4] = new Account("Spark", "111111", "1");

accounts[5] = new Account("Admin", "111111", "0");

}

public Account login(String userName, String password) throws Exception {

for (int i = 0; i < accounts.length; i++) {

Account account = accounts[i];

if (account.getUserName().equals(userName)) {

if (account.getPassword().equals(password)) {

return account;

} else {

throw new Exception("Password is error.");

}

}

}

throw new Exception("User name does not exist. ");

}

public Student findStudent(String name) {

for (int i = 0; i < students.length; i++) {

Student student = students[i];

if (student.getName().trim().toLowerCase().equals(name.toLowerCase())) {

return student;

}

}

return null;

}

public void printStudents() {

for (int i = 0; i < students.length; i++) {

System.out.println(students[i]);

}

}

public void sort() {

for (int i = 0; i < students.length; i++) {

for (int j = 0; j < students.length - 1; j++) {

if (students[j + 1].getAvgScore() > students[j].getAvgScore()) {

Student stuTemp = students[j + 1];

students[j + 1] = students[j];

students[j] = stuTemp;

}

}

}

}

}

class Account {

private String userName;

private String password;

private String role;

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public Account(String userName, String password, String role) {

this.userName = userName;

this.password = password;

this.role = role;

}

public String getRole() {

return role;

}

public void setRole(String role) {

this.role = role;

}

}

上面的代码中，我们添加了账号信息，以便用户可以登录。同时通过账号类中role属性区分是普通学生还是管理员（普通学生只允许查看自己的成绩，管理员可以查看所有学生的成绩）。如果某用户不掌握任何账号信息，是无法使用该系统获取信息的，这就提高了系统的安全性，避免了数据的泄露。

下面我们就成绩查询系统中的排序算法做优化，代码如下：

class School {

…

public void sort() {

boolean isSorted;

for (int i = 0; i < students.length; i++) {

isSorted = true;

for (int j = 0; j < students.length - i - 1; j++) {

if (students[j + 1].getAvgScore() > students[j].getAvgScore()) {

Student stuTemp = students[j + 1];

students[j + 1] = students[j];

students[j] = stuTemp;

isSorted = false;

}

}

if (isSorted){

break;

}

}

}

}

上面的排序算法，做了两个地方的修改，一个是内循环次数最大值减少i，另一个是使用了一个判断是否已经排完序的判断布尔变量，如果已经排完序就不再进入下一轮外层循环。这样就对性能做了改进。

下面我们对成绩查询系统进一步做改进，从两个方面提高其可维护性。第一是登录的账号目前有两种角色，一类是普通用户，一类是管理员。如果将来还需要扩展新的角色，那么势必需要修改其中的代码。第二是排序的方式目前是通过比较平均成绩，但也有可能按其他信息（比如按年龄或者学号）排序，那么就需要修改排序代码。利用多态，修改代码如下：

public class Client {

public static void main(String[] args) {

StudentSystem studentSystem = new StudentSystem(new AgeComparator());

Account account = studentSystem.login();

if (account != null){

account.query(studentSystem.getSchool());

}else{

System.out.println("Login failed, system exited. ");

}

}

}

class StudentSystem {

private School school;

public StudentSystem(Comparator<Student> comparator) {

this.school = new School(comparator);

}

public School getSchool() {

return school;

}

public void setSchool(School school) {

this.school = school;

}

public String getUserName() {

Scanner scanner = new Scanner(System.in);

System.out.println("Please input user name:");

return scanner.next();

}

public String getPassword() {

Scanner scanner = new Scanner(System.in);

System.out.println("Please input password:");

return scanner.next();

}

public Account login() {

return school.login(getUserName(), getPassword());

}

}

class StudentAccount extends Account {

public StudentAccount(String userName, String password) {

super(userName, password);

}

@Override

public void query(School school){

System.out.println("Your average score is:");

Student student = school.findStudent(this.getUserName());

if (student != null){

System.out.println(student.getAvgScore());

}

}

}

class AdminAccount extends Account {

public AdminAccount(String userName, String password) {

super(userName, password);

}

@Override

public void query(School school){

school.sort();

System.out.println("Scores of all the students are:");

school.printStudents();

}

}

class School {

private Student[] students;

private Account[] accounts;

private Comparator<Student> comparator;

public School(Comparator<Student> comparator) {

this.comparator = comparator;

this.students = new Student[5];

this.accounts = new Account[6];

initializeStudents();

initializeAccounts();

}

public void initializeStudents() {

…

}

public void initializeAccounts() {

accounts[0] = new StudentAccount("Tom", "111111");

accounts[1] = new StudentAccount("Alice", "111111");

accounts[2] = new StudentAccount("Bob", "111111");

accounts[3] = new StudentAccount("Jerry", "111111");

accounts[4] = new StudentAccount("Spark", "111111");

accounts[5] = new AdminAccount("Admin", "111111");

}

public Account login(String userName, String password) {

for (int i = 0; i < accounts.length; i++) {

Account account = accounts[i];

if (account.getUserName().equals(userName) &&

account.getPassword().equals(password)) {

return account;

}

}

return null;

}

public Student findStudent(String studentId) {

…

}

public void printStudents() {

…

}

public void sort() {

boolean isSorted;

for (int i = 0; i < students.length; i++) {

isSorted = true;

for (int j = 0; j < students.length - i - 1; j++) {

if (this.comparator.compare(students[j + 1], students[j]) > 0) {

isSorted = false;

Student stuTemp = students[j + 1];

students[j + 1] = students[j];

students[j] = stuTemp;

}

}

if (isSorted) {

break;

}

}

}

}

class ScoreComparator implements Comparator<Student> {

@Override

public int compare(Student stuSrc, Student stuObj) {

return stuSrc.getAvgScore() > stuObj.getAvgScore() ? 0 : 1;

}

}

class AgeComparator implements Comparator<Student> {

@Override

public int compare(Student stuSrc, Student stuObj) {

return stuSrc.getAge() > stuObj.getAge() ? 0 : 1;

}

}

abstract class Account {

private String userName;

private String password;

public String getUserName() {

return userName;

}

public void setUserName(String userName) {

this.userName = userName;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public Account(String userName, String password) {

this.userName = userName;

this.password = password;

}

public abstract void query(School school);

}

成绩查询系统中，当用户不小心输入了错误的用户名或者密码，程序就退出了，用户需要重新运行程序才能再次尝试。这种设定明显对用户不友好，所以需要对其改进，提高易用性。我们修改StudentSystem类，改进其login()方法，代码如下：

class StudentSystem {

…

public Account login() {

Account account = school.login(getUserName(), getPassword());

while (account == null){

System.out.print("Login failed, system exited!");

account = school.login(getUserName(), getPassword());

}

return account;

}

…

}

当用户输入错误时，系统给出错误信息，并让用户重新尝试输入信息，无疑提高了易用性。

## 设计的坏味道

软件开发的前期设计，对于软件的健壮性、可维护性非常重要。如果软件开发前期设计不当，在后期的开发和维护中，会逐渐“腐化变臭”。通常，无论是开发者还是客户，都希望设计出来的软件产品易扩展、高可用、易用可靠、简单而又安全。与之相反，一个设计糟糕的软件系统，会发出如下“臭味”：

1. 僵化

僵化意味着很难对系统进行改动，因为每个改动都会迫使许多对系统其他部分的其他改动。如果单一的改动会导致依赖关系的模块中的连锁改动，那么设计就是僵化的，必须要改动的模块越多，设计就越僵化。僵化对可扩展性带来损害。

1. 脆弱

脆弱意味着对系统的改动会导致系统中和改动的地方在概念上无关的许多地方出现问题。出现新问题的地方与改动的地方没有概念上的关联。要修正这些问题又会引出更多的问题，从而使开发团队就像一只不停追逐自己尾巴的狗一样。脆弱不利于提高灵活性。

1. 固化

很难解开系统的纠结，使之成为一些可在其他系统中重用的组件。设计中包含了对其他系统有用的部分，而把这些部分从系统中分离出来所需的努力和风险是巨大的。固化损害了可复用性。

1. 粘滞

做正确的事情比做错误的事情要困难。面临一个改动的时候，开发人员常常会发现多种改动的方法。有的方法会保持系统原来的设计，而另外一些则会破坏设计，当那些可以保持系统设计的方法比那些破坏设计的方法更难应用时，就表明设计具有很高的粘滞性，做错误的事情就很容易。粘滞对于可插入性是一个大问题。

1. 不必要的复杂性

过度设计导致了复杂性，设计中包含有不具任何直接好处的基础结构对系统是有害的。如果设计中包含有当前没有用的组成部分，就含有不必要的复杂性。当开发人员预测需求的变化，并在软件中放置了处理那些潜在变化的代码时，常常出现这种情况。不必要的复杂性不利于程序的易读性。

1. 不必要的重复

设计中包含有重复的结构，而该重复的结构本可以使用单一的抽象进行统一。当拷贝、剪切和粘贴大量发生在编程的时候，这种情况就会发生。不必要的重复对于程序的可扩展性有很大影响。

1. 晦涩性

很难搞懂理解设计者的意图，设计的架构没有直观地表现出设计意图。设计结果可以用清晰、富有表现力的方式建模。一般说来，随着时间的推移，设计会变得越来越晦涩。晦涩影响了程序的可读性，从而对可维护性造成损害。

由于需求没有按照初始设计预见的方式进行变化，从而导致了设计的腐化。通常，改动都很急迫，并且进行改动的开发人员对于原始的设计思路并不熟悉。因而，虽然对设计的改动可以工作，但是它却以某种方式违反了原始的设计。随着改动的不断进行，这些违反渐渐地积累，设计开始出现臭味。

然而，我们不能因为设计的退化而责怪需求的变化。作为设计人员，我们对于需求变化有非常好的了解。事实上，我们中的大多数人都认识到需求是项目中最不稳定的要素。如果我们的设计由于持续、大量的需求变化而失败，那就表明我们的设计和实践本身是有缺陷的。我们必须要设法找到一种方法，使得设计对于这种变化具有弹性，并且应用一些实践来防止设计腐化。

# 包图

在表达系统设计中的架构模型时，需要使用UML两种图来表示，一种是包图，另一种是组件图。包图从逻辑角度表达系统中的主要逻辑元素以及它们之间的关系，而组件图表达的是逻辑元素和它们之间的关系。本节先来看看包图的基本语法。

在 UML中，协作、分区和层可以由更高级别的构造型表示——包。包是一种通用结构，可以应用于 UML 模型中的任何元素。包图是一种结构图，显示了大中型项目中模型元素的排列和组织。包图可以显示子系统或模块之间的结构和依赖关系，显示系统的不同视图，对于复杂系统的结构展示尤其重要。

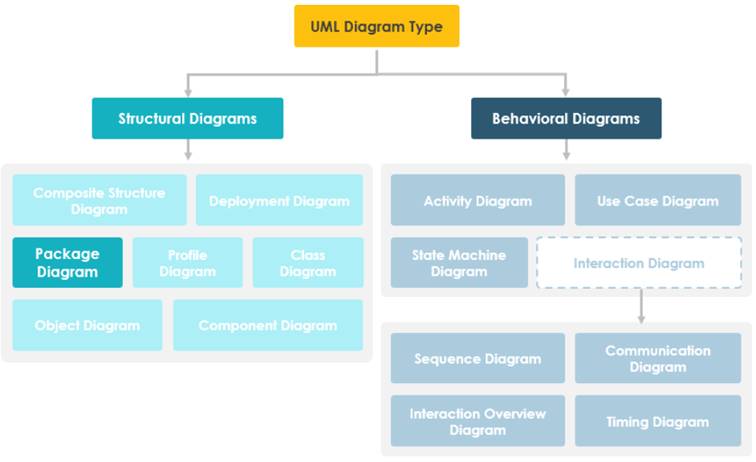


图 UML包图

包图用于构建高级系统元素，它用来组织包含图表、文档和其他关键可交付成果的大型系统。包图可以用来简化复杂的类图，它可以将类分组到包中。包是逻辑上相关的 UML 元素的集合。包可以看作磁盘上的文件夹，可以在任何 UML 图上使用，用来对于UML元素进行归类整理。

## 包图的作用

包图用于构建高级系统元素。包用于组织包含图表、文档和其他关键可交付成果的大型系统。

* 在逻辑上把一个复杂的系统模块化：反映系统的高层架构，在逻辑上将系统进行模块化分解；。
* 用来描述需求的高阶概况。可以通过包来简要描述系统的业务需求。
* 用来描述设计的高阶概况。可以通过包来组织系统的业务设计模型和框架模型。
* 用来组织源代码，在实际应用中，包是组织源代码的方式。

## 包图语法

包图用于简化复杂的类图，可以将类分组到包中。包是逻辑上相关的 UML 元素的集合。

下图是一个业务模型，其中类被分组到包中：

* 包显示为顶部带有小选项卡的矩形。
* 包名在选项卡上或矩形内。
* 虚线箭头是依赖关系。
* 一个包中可以放置其它子包。
* 一个包依赖于另一个包，如果另一个包的变化可能会强制第一个包发生变化。

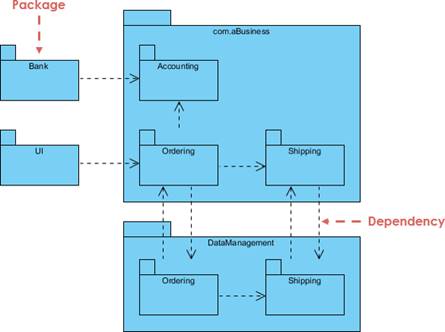


图 封装图的 FIGURE 语法

包图中，依赖关系最常用，由虚线箭头表示。依赖关系表示一个包中的元素需要依赖另一个包中的某些元素。包图中的主要元素是包，显示为顶部带有小选项卡的矩形。它的名称在选项卡上或矩形内。虚线箭头是依赖关系。如果另一个包的更改可能会迫使第一个包发生更改，则一个包依赖于另一个包。

### 包

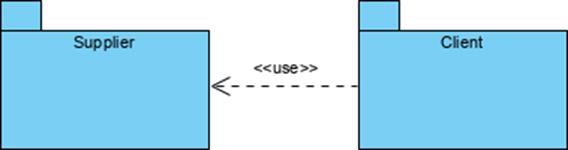
包是用于对元素（包括其他包）进行分组的 UML 机制。包图遵循嵌套包的层次结构。嵌套包的原子模块通常是类图。使用包图时遵循以下约束：

* 一个系统的包名不应该相同，但是不同包内的类可以有相同的名字。
* 包可以包括整个图表、单独的组件名称或根本没有组件。
* 每个包都有自己的命名空间，在这个命名空间内所有的名字都必须是唯一的。
* 每个模型元素都属于一个包。
* 包形成层次结构。一个包可以放其它子包，原子元素是类。

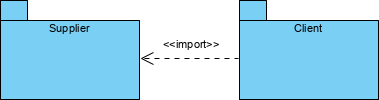
### 依赖

两个包之间最常见的关系就是依赖关系，是指两个包所包含的模型元素之间存在着一个或多个依赖。包之间的依赖有use、import、access、trace四种构造型。使用依赖和引入依赖比较常见。

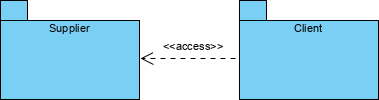
使用依赖的构造型是《use》，是一种默认的依赖关系，因此它可以在建图时，不指出构造型。说明客户包中的元素以某种方式使用提供者包的公共元素，也就是说客户包依赖于提供者包。



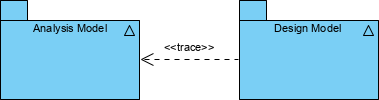
引入依赖的构造型是《import》，是包与包之间的一种存取依赖关系。引入是指允许一个包中的元素存取另一个包中的元素。引入依赖使用虚线箭头从客户包方向指向提供者包。引入依赖没有传递性，一个包的输出不能通过中间的包被其他的包引入。



访问依赖使用构造型《access》，表达了提供者包命名空间的公共元素被添加为客户包命名空间上的私有元素的一种关系，如下图：



跟踪依赖使用构造型《trace》，表达了一个包发展成为另一个新版本的包。下图中基于分析模型包进一步演化，升级成了设计模型包，两者的关系可以表示为：



## 创建包图

在本节中，我们描述了一个简单的四步创建包图的过程：

1. 确定包图的上下文，确定这个包图的基本作用。

对于大型复杂系统来说，往往一个包图是不足以来描述完整所有元素的。一个包图可用于对某个区或层进行建模。在绘制包图之前需要对该图用来描述哪个层或者分区有明确的认知。

1. 根据类共享的关系将类归类到各个包中。

基于系统分析模型中的对象模型，将其中的类基于关系进行归纳。这些关系包括泛化、聚合、组合、关联、依赖等。分析系统的模型元素，运用分层设计原则把概念、语义和逻辑上相近的元素包含在同一个包中。

1. 识别包之间的依赖关系。

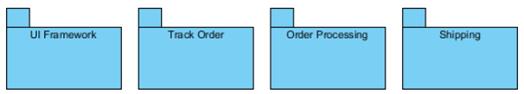
在这种情况下，我们会审查跨越包边界的关系，以发现潜在的依赖关系。如果包内某个类需要依赖另一个包中的某个或者某些类，那么两个包之间就要表示依赖关系。

1. 优化包图。

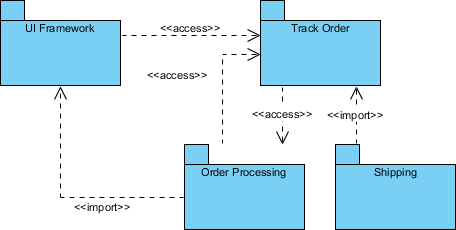
分析包结构，基于系统设计原则进行包的优化、合并以及进一步分解等操作。系统设计原则会在后面再来讨论。

下面通过一个例子来说明绘制包图的过程。将为某个在线购物商店的订单处理子系统设计包图，该负责为客户订购的产品提供跟踪信息。客户输入跟踪序列号，跟踪订单模块引用系统并向客户更新当前的运输状态。

根据项目描述，我们首先要识别系统中的包，然后根据关系将它们关联在一起。这里有一个跟踪订单模块，它必须与其他模块交谈才能了解订单详情，我们称之为“Order Processing”。接下来在获取订单详细信息后，它必须知道运输详细信息，让我们称之为“Shipping”。

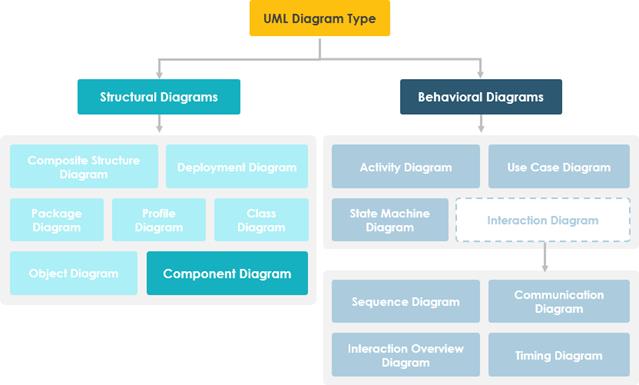


然后识别系统中的依赖关系，跟踪订单应该从“订单详细信息”中获取订单详细信息，“订单详细信息”必须知道客户提供的跟踪信息。两个模块相互访问，这就满足了<<access>>双重依赖。想要了解运费信息，“Shipping”可以导入“Track Order”，方便检索。最后，“Track Order”对“UI Framework”的依赖性也被映射，这完成了我们的订单处理子系统的包图。如下图所示：



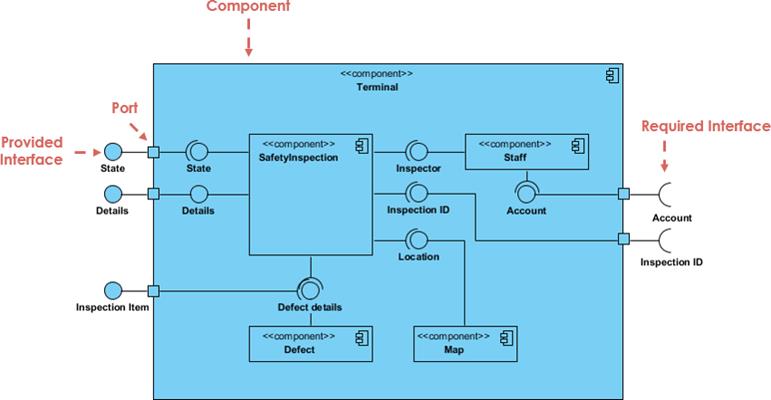
# 组件图

UML 组件图用于对面向对象系统的物理方面进行建模，这些系统用于可视化、指定和记录基于组件的系统，还用于通过正向和反向工程构建可执行系统。组件图本质上是类图，它侧重于通常用于对系统的静态实现视图建模的系统组件。



## 概述

组件图将正在开发的实际系统分解为各种高级功能。每个组件负责整个系统中的一个明确目标，并且仅在需要知道的基础上与其他基本元素交互。



上面的示例显示了一个较大组件的内部组件：

数据（帐户和检验 ID）通过右侧的端口流入组件，并转换为内部组件可以使用的格式。右侧的接口称为必需接口，表示组件为履行其职责所需的服务。

然后，数据通过各种连接传递到并通过几个其他组件，然后在左侧的端口输出。左边的那些接口被称为提供接口，它代表展示组件要交付的服务。

重要的是要注意内部组件被一个大“盒子”包围，它可以是整个系统本身（在这种情况下右上角不会有组件符号）或整个系统的子系统或组件（在这种情况下，“盒子”本身就是一个组件）。

## 要素

### 组件

组件代表系统的模块化部分，它封装了它的内容并且其表现形式在其环境中是可替换的。在UML2.0中，组件是一个矩形，右上角有个组件的标记。UML2.0中组件的高级抽象视图可以建模为：

* 带有组件构造型的矩形
* 带有组件图标的矩形
* 带有构造型和图标的矩形

组件的外观

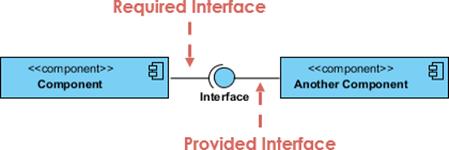
图 组件的图形外观

组件是定义了良好接口的、可重用的、可替代的物理实现单元，它一般表示实际存在的、物理的物件。程序源代码、可执行文件、子系统、一个脚本、配置文件、中间件、动态链接库（DLL）、ActiveX控件等都可以成为系统中的组件。组件隐藏了内部实现的细节，仅通过接口提供服务。我们可以把组件理解为一个黑盒子，这个黑盒子使用接口来公开其公共可见的属性及操作，这一点与类非常相似。

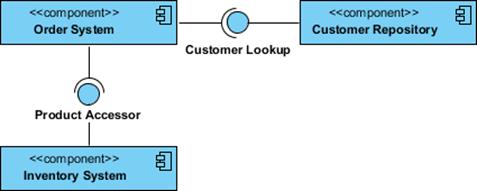
### 接口

在下面的示例中显示了两种类型的组件接口：

* 提供接口，也称棒棒糖接口，末尾有一个完整的圆圈，表示该组件通过此接口提供一组服务。
* 需求接口，也称插座接口，末尾有个半圆，表示组件需要通过该接口接收服务。

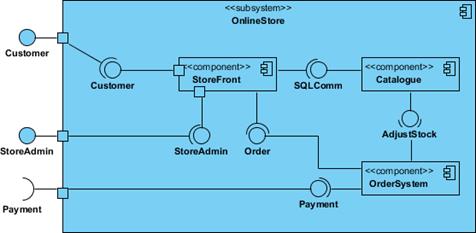


例如，订单系统的使用界面如下：



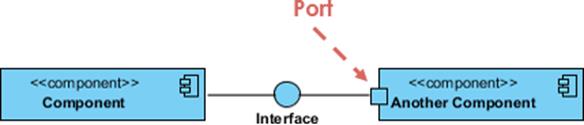
### 子系统

子系统分类器是组件分类器的特殊版本。因此，子系统符号元素继承了与组件符号元素相同的所有规则。唯一的区别是子系统符号元素具有子系统而不是组件的关键字。



### 端口

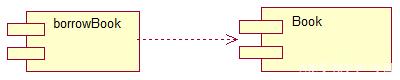
端口使用沿系统或组件边缘的正方形表示。端口通常用于帮助公开组件所需和提供的接口。



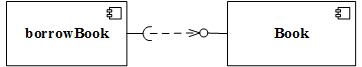
### 关系

组件图中组件之间有依赖关系、聚合、约束、泛化、关联和实现等关系，但最常见的还是依赖关系。

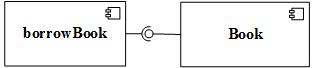
组件图中的依赖关系与类图中相同，都是“供应者-客户”关系（supplier-client），使用虚线箭头由客户（client）组件指向供应者（supplier）组件。下图表示组件“borrowBook”依赖于组件“Book”，其中，“borrowBook”是客户组件，Book是供应者组件。



下图也表示了两个组件之间的依赖关系，它表示了borrowBook组件的一个需求接口依赖于Book组件的一个提供接口：



一般这种依赖关系都是从需求接口引出指向提供接口。上图也可以简化成下面的形式：



上面这个图表示了borrowBook使用某个需求接口与Book组件的提供接口产生依赖关系。

## 创建组件图

### 创建过程

对源代码结构建模的流程如下：

* 识别出同一功能点的源代码文件集合，并建模为组件
* 如果系统规模较大，使用包对组件进行分组
* 使用接口和依赖关系来表示这些源文件之间的关系
* 检查组件图的合理性，并识别源代码文件的优先级以便进行开发工作

对可执行程序结构建模的流程如下：

* 识别出相关的运行组件集合
* 考虑集合中每个组件的类型
* 如果系统规模较大，可以使用包对组件进行分组。这里包的使用可以对应于相应文件的文件存储结构。
* 分析组件之间的关系，使用接口和依赖关系建模这些关系。
* 考量建模结果是否实现了组件的各个特性，对建模的结果进行细化。

### 创建示例

对于源代码来说，可以通过正向或反向工程，可以识别感兴趣的源代码文件集，并将它们建模为原型为文件的组件。对于较大的系统来说，可以使用包来显示源代码文件组。使用依赖关系对这些文件之间的编译依赖关系进行建模。如下图所示：



图组件示例 - Java 源代码

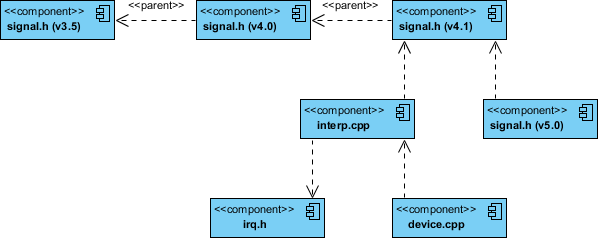
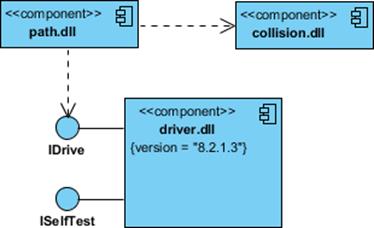


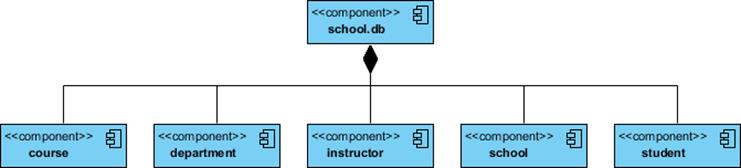
图 组件图示例 - 带版本控制的 C++ 代码

对于可执行文件来说，首先确定要建模的组件集。通常，这将涉及存放在某个计算机节点上的部分或全部组件，或者这些组件集在系统中所有节点上的分布。

对于该集合中的每个组件，请考虑其与其关联的组件之间的关系。大多数情况下，这将涉及由某些组件导出（实现）然后由其他组件导入（使用）的接口。如果您想暴露系统中的接口，请明确地为这些接口建模。如下图所示：



还可以使用组件图对数据库进行建模，识别模型中代表逻辑数据库模式的类。选择将这些类映射到表的策略。需要考虑数据库的物理分布。您的映射策略将受到您希望数据在已部署系统上的存放位置的影响。要可视化、指定、构造和记录这些映射，可以创建一个组件图，其中包含构造为表的组件。这样有助于在数据库设计阶段将数据库中的逻辑设计转变为最后物理设计。如下图所示：

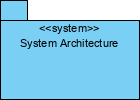


# 系统分解

在本节中，我们将更详细地描述子系统分解及其属性。首先，我们定义子系统的概念及其与类的关系。接下来，我们看看子系统的接口：子系统为其他子系统提供服务。服务是一组具有共同目的的相关操作。在系统设计期间，我们根据子系统提供的服务来定义子系统。稍后，在对象设计期间，我们根据它提供的操作来定义子系统接口。接下来，我们看看子系统的两个属性，耦合和内聚。

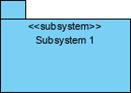
## 系统

系统表示为具有 <<system>> 构造型的包，如下图所示。该系统代表属于特定项目的所有模型元素。在构建更详细的模型时，您还可以将系统分解为<<业务系统>>和<<应用系统>>，使它们更小、更实用。在 UML 中，包表示为文件夹。

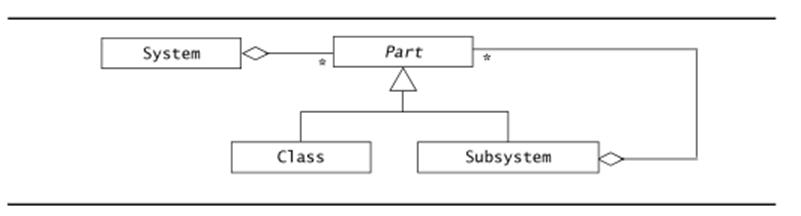


## 子系统和类

子系统是一组模型元素，它们是整个系统的一部分。子系统与系统一样，是具有 <<subsystem>> 构造型的构造型包，如下图所示。



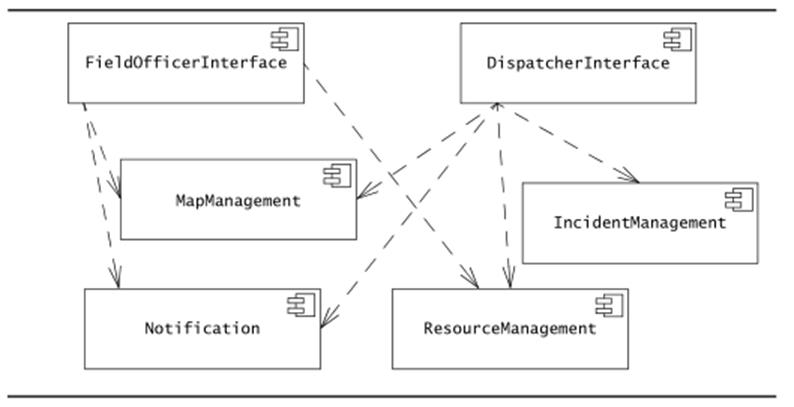
为了降低应用程序域的复杂性，我们确定了称为“类”的较小部分并将它们组织成包。同样，为了降低解决方案域的复杂性，我们将系统分解为更简单的部分，称为“子系统”，它们由许多解决方案域类组成。子系统是系统的可替换部分，具有定义良好的接口，封装了其所含类的状态和行为。一个子系统通常对应于单个开发人员或单个开发团队可以处理的工作量。通过将系统分解为相对独立的子系统，并发团队可以以最小的通信开销在各个子系统上工作。在复杂子系统的情况下，我们递归地应用这一原则并将一个子系统分解为更简单的子系统。



**图**11-3 子系统分解（UML 类图）。

例如，我们之前描述的事故管理系统可以分解为  DispatcherInterface子系统，实现 Dispatcher 的用户界面；一个FieldOfficerInterface子系统，实现 FieldOfficer 的用户界面；一个事件管理子系统，负责事件的创建、修改和存储；一个ResourceManagement 子系统，负责跟踪可用的资源（例如，消防车和救护车）；用于描述地图和位置的MapManagement ；Notification子系统， 实现FieldOfficer终端与调度台之间的通信。

下图使用 UML 组件描述了这个子系统分解。组件被描述为右上角带有组件图标的矩形。组件之间的依赖关系可以用虚线箭头表示。在 UML 中，组件可以表示逻辑组件和物理组件。逻辑组件对应于没有显式运行时等效项的子系统，例如，组合在一起成为单个运行时应用程序逻辑层的各个业务组件。物理组件对应于作为显式运行时等价物的子系统，例如数据库服务器。



**图** 事故管理系统的子系统分解（UML 组件图）。

子系统显示为 UML 组件。虚线箭头表示子系统之间的依赖关系。几种编程语言（例如，Java 和 Modula-2）为建模子系统（Java 中的包，Modula-2 中的模块）提供构造。在其他语言中，例如C或C++，子系统没有明确建模，因此开发人员使用约定来对类进行分组（例如，子系统可以表示为包含实现子系统的所有文件的目录）。无论子系统是否在编程语言中明确表示，开发人员都需要仔细记录子系统分解，因为子系统通常由不同的团队实现。

## 服务和子系统接口

子系统的特征在于它向其他子系统提供的服务。服务是一组具有共同目的的相关操作。例如，提供通知服务的子系统定义了发送通知、查找通知通道以及订阅和取消订阅通道的操作。一个子系统可供其他子系统使用的操作集构成了子系统接口。子系统接口包括操作的名称、它们的参数、它们的类型和它们的返回值。系统设计的重点是定义每个子系统提供的服务，即枚举操作、它们的参数和它们的高层行为。对象设计将集中在应用程序员接口上 (API)，它改进和扩展了子系统接口。API还包括参数的类型和每个操作的返回值。

提供的和需要的接口可以在 UML 中用装配连接器（也称为球窝连接器）描述。提供的界面显示为球形图标（也称为棒棒糖），旁边有它的名称。所需接口显示为套接字图标。两个子系统之间的依赖关系通过连接组件图中相应的球和插座来表示。

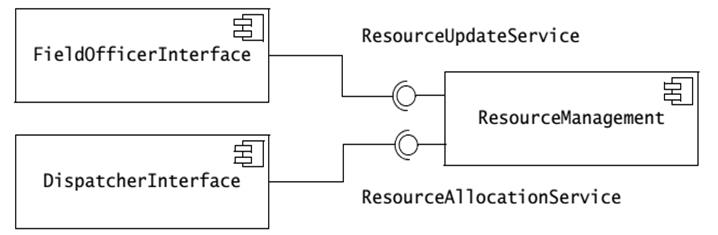


图 6-5 描述了 FieldOfficerInterface、DispatchterInterface和  ResourceManagement 子系统之间的依赖关系。FieldOfficerInterface需要  ResourceUpdateService 来更新 FieldOfficer 的状态和位置。DispatcherInterface 需要  ResourceAllocationService来识别可用资源并将它们分配给新的事件。资源管理子系统提供这两种服务。请注意，当子系统分解已经相当稳定并且我们的重点已从子系统的识别转移到服务的定义时，我们使用球窝表示法。在系统设计的早期阶段，我们可能对子系统的功能分配没有如此清晰的认识，在这种情况下，我们使用图 6-4 中的依赖表示法（虚线箭头）。

根据它提供的服务来定义子系统有助于我们专注于它的接口而不是它的实现。在编写子系统接口时，应尽量减少提供的有关实现的信息量。例如，子系统接口不应引用内部数据结构，例如链表、数组或哈希表。这使我们能够在修改子系统的实现时将变更的影响降至最低。更一般地说，我们希望通过最小化子系统之间的依赖关系来最小化变更的影响。

## 组件 vs 子系统 vs 包

组件是遵循一组接口并在系统中提供实现的物理组件。它通常是指开发类和运行时类的物理实现。组件通常表示实际和物理对象。组件可以是：程序源代码、子系统、动态链接库等，组件一般包含很多类，实现很多接口。组件分类，配置组件是形成可执行文件的基础，如动态链接库、二进制可执行文件；工作产品组件是配置组件的类元素，例如数据文件和程序源代码；执行组件是最终可运行程序产生的运行结果。

子系统是一个组件，通常包括许多较小的组件，这是一个大组件。

包是一组元素，它们在逻辑上形成了更高级别的单元。用于分组类。与命名空间类似，名称在同一个包内是唯一的。

总的来说，组件和包的相似之处在于它们是一个大的“集合体”。组件比较齐全。它是一个完整的个体，具有独立的功能。对于外部来说，内部的组件并不重要，重要的是它能提供的接口。包只是多个个体的集合，就像一个类。没有独立的功能，内部个体有功能。对于外面来说，包本身并不重要，重要的是包里面的类。

## 包设计原则

根据包图的知识，我们知道，包可以用作包容一组类的容器。通过把类组织成包，读者可以在很高层次的抽象上来理解设计者的设计理念和思想。但是，如何合理而科学地将类划分到合适的包里不是一件很容易的事，需要设计者有着较深的理论基础和丰富经验。这里有一些关于系统分解的基本原则，它们将为设计者提供一些好的建议。

### 重用发布等价原则

重用发布等价原则（简称REP）告诉我们，包的重用粒度应该就是包的发布粒度，即一个包中的软件要么都是可重用的，要么都是不可重用的。

REP指出，一个包的重用粒度应该和发布粒度一样大。我们说重用的任何东西都必须同时被发布和跟踪。简单的编写一个类，然后声称它是可重用的做法是不现实的。只有在建立一个跟踪系统，为潜在的使用者提供所需的变更通知、安全性以及支持后，重用才有可能。

为了有效地重用，代码必须是以包为单位，完整且透明。以包为单位的代码并不表示它是可被重用的，我见太多这样做而失败的案例。只有真正发布的，且透明的完整代码才存在被重用的可能性。同时，重用也驱动我们在写代码时构建更小的包，业务逻辑分离得更彻底并保持完整。

可重用性不是唯一的标准，我们也要考虑重用这些软件的人。当然，一个容器类库是可重用的，一个金融方面的框架也是可重用的。但是，我们不希望把它们放进同一个包中。很多希望重用容器类库的人可能对于金融框架根本不感兴趣。因此，我们希望一个包中的所有类对于同一类用户来说都是可重用的。我们不希望一个用户发现包中所包含的类中，一些是他所需要的，另一些对他却完全没用。

### 共同重用原则

共同重用原则（简称CRP）意思是一个包中的所有类应该是共同重用的。如果重用了包中的一个类，那么应该重用包中的所有类。

包经常以JAR、DLL的形式出现。如果被使用的包以JAR包的形式发布，那么使用这个包的代码即使只依赖这个包的一个类，也将依赖这个包。所以如果JAR包的内容进行了修改，即使是非依赖的类进行了修改，也会影响使用者重新验证和发布新的版本。因此这就告诉我们，一个包中的类都是不可分开的，仅仅只依赖其中的一部分情况尽可能不要出现。反过来讲，相互之间没有紧密联系的类不应该在同一个包中。

### 共同封闭原则

共同封闭原则（简称CCP）告诉我们，包中所有类应该对于同一类性质的变化应该是共同封闭的。一个变化若对包产生影响，则将对包中所有的类产生影响。而对其它的包不造成影响。

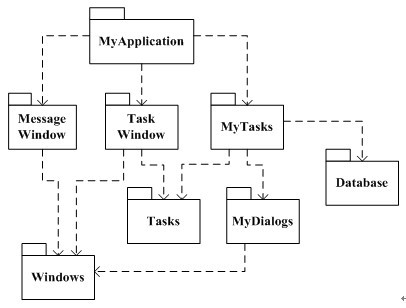
在大多数的应用中，可维护性的重要性是超过可重用性的。如果一个应用中的代码必须更改，那么我们宁愿更改都集中在一个包中，而不是分布在多个包中。如果更改集中在一个单一的包中，那么我们仅仅需要发布那一个更改了的包。不依赖于那个更改了的包的其他包则不需要重新验证或重新发布。

CCP鼓励我们把可能由于同样的原因而改变的所有类共同聚集在同一个地方。如果两个类之间有非常紧密的绑定关系，不管是物理上的还是概念上的，那么它们总会一同进行变化，因而它们应该属于同一个包中，这样做会减少软件的发布、重新验证、重新发行的工作量。

### 无环依赖原则

无环依赖原则（简称ADP），告诉我们，在包图中依赖路径不应该存在环路。

下面的包图中展示了组成一个应用程序的包结构。该图可以看作一个有向图，其中，包是结点，依赖关系是有向边。



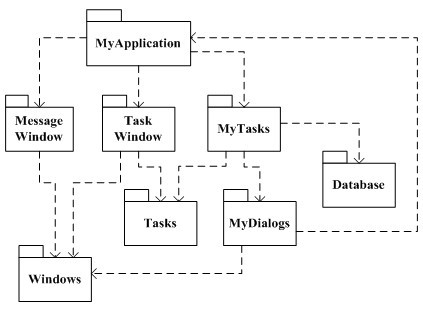
现在，请注意另外一件事情。无论从哪个包开始，都无法沿着依赖关系而回绕到这个包。该结构中没有环，表示该图是一个有向无环图。

当负责MyDialogs的团队发布了该包的一个新版本时，会很容易找出受影响的包：只需逆着依赖关系指向寻找即可。因此，MyTasks和MyApplication都会受到影响。当前工作于这两个包的开发人员就要决定何时应该和MyDialogs的新版本集成。

当MyDialogs发布时，完全不会影响到系统中许多其他包。它们不知道MyDialogs，并且也不关心何时对MyDialogs进行了更改。这很好。这意味着发布MyDialogs的影响相对较小。当工作于MyDialogs包的开发人员想要运行该包的测试时，只需把他们的MyDialogs版本和当前正使用的Windows包的版本一起编译、链接即可。不会涉及到系统中任何其他包。这意味着工作于MyDialogs的开发人员只需较少的工作即可建立一个测试，而且他们要考虑的变化因素也不多。

在发布整个系统时，是自底向上进行的。首先编译、测试已经发布Windows包。接着是 MessageWindow和MyDiaolgs。在它们之后是Tasks，然后是TaskWindow和Database。接着是MyTasks，最后是MyApplication。这个过程非常清楚并且易于处理。我们知道如何去构建系统，因为我们理解系统各个部分的依赖关系。

如果一个新需求迫使我们更改MyDialogs中的一个类去使用MyApplication中的一个类。这就产生了一个依赖关系环，如下图所示：



图具有依赖环的包图

这个依赖环会存在如下后果：

首先难以发布版本。发布新的MyTasks包，必须要兼容Task、MyDialogs、Database以及Windows。然而，由于依赖关系的存在，它们现在必须也兼容MyApplication、 TaskWindow、以及MessageWindow。也就是说，现在MyTasks依赖于系统中所有其他的包。这就致使MyTasks非常难以发布。其它包也有着同样的问题。它们实际 上已经变成了同一个大包。于是，在这些包上工作的所有开发人员，他们彼此之间的发布行动要完全一致，因为它们必须都要使用彼此间完全相同的版本。

第二个问题是每次发布必须要链接进系统中所有其他的包，包括Database包。这意味着仅仅为了测试MyDialogs就必须做一次完整的构建。这是不可忍受的。

第三，依赖关系图中存在环，使得非常难以对模块进行隔离。单元测试和发布变得非常困难且易于出错。有些语言比如C++中，编译时间会随模块的数目成几何级数增长。

最后，如果依赖关系图中存在环，就很难确定包构建的顺序。事实上，也许就不存在恰当的顺序。对于像Java一样要从编译过的二进制文件中读取它们的声明的语言来说，这将非常麻烦。

有两种主要方法来解除包之间的依赖环。第一种是创建一个具有MyDialogs需要的接口的抽象基类。然后，把该抽象基类放进MyDialogs中，并使MyApplication中的类从其继承。这就倒置了MyDialogs和MyApplication间的依赖关系，从而解除了依赖关系环。

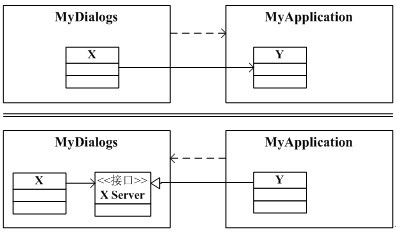


 图2.6.2.1.2-1 使用依赖倒置解除依赖环

第二个办法是新创建一个MyDialogs和MyApplication都依赖的包。把MyDialogs和MyApplication都依赖的类移到这个新包中。如下图：

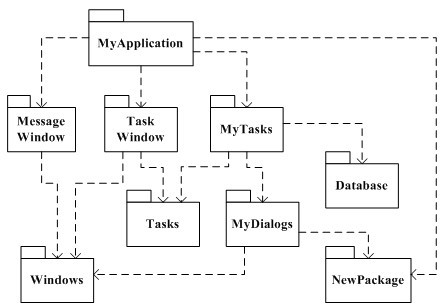


图 使用新包解除依赖环

第二个解决方案意味着，在需求改变面前，包的结构是不稳定的。事实上，随着应用程序的增长，包的依赖关系结构会抖动和增长。因此，必须始终要对依赖关系结构中环的情况进行监控。如果出现了环，就必须要使用某种方法把其解除。有时这意味着要创建新的包，致使依赖关系结构增长。

讨论到现在，我们可以得出一个必然的结论：不能自顶向下设计包的结构。事实上，包结构应该时随着系统的增长、变化而逐步演化的。

也许你会认为这是违反直觉的。我们已经认为像包这样的大粒度分解同样也是高层的功能分解。当我们看到一个像包依赖关系结构这样的大粒度分组时，就会觉得包应该以某种方式描绘了系统的功能。然而，这可能不是包依赖关系图的一个属性。

事实上包的依赖关系图和描绘应用程序的功能之间没有任何关系。它们是应用程序可构建性的映射图。这就是不在项目开始设计它们的原因。包的依赖关系结构是和系统逻辑设计一起增长演化的。

### 稳定依赖原则

稳定依赖原则（也称SDP）告诉我们，包应该朝着稳定方向依赖。

通常，要使设计有好的可维护性，某种程度的灵活性是必要的。我们通过遵循共同封闭原则（CCP）来达到这个目标。使用这个原则，可以创建对某些变化类型敏感的包。这些包被设计成可变的。我们期望它们的变化。

对于任何包而言，如果期望它是可变的，就不应该让一个稳定的包依赖于它。否则，可变的包同样也会难以更改。比如设计者设计了一个易于更改的包，其他人只要创建一个对它的依赖就可以使它变得难以更改，这就是软件的反常特性。没有改变你的模块中任何一行代码，可是它突然之间就变得难以更改了。通过遵循SDP，我们可以确保那些打算易于更改的模块不会被那些比它们难以更改的模块所依赖。

并非所有的包都应该是稳定的。如果一个系统在所有的包都是最大程度稳定的，那么该系统就是不能改变的。这不是所希望的情形。事实上，我们希望所设计出来的包结构中，一些包是不稳定的而另外一些是稳定的。下图展示了一个具有3个包子系统结构。

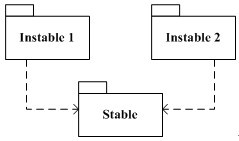


图 理想的包依赖

可改变的包位于顶部并依赖于底部稳定的包。把不稳定的包放在图的顶部是一个有用的约定，因为任何一个向上的箭头都意味着违反了SDP。

下图展示了违反SDP的情况。打算让Flexible包易于更改。我们希望Flexible是可改的。然而，一些工作于包 Stable的开发人员，创建了一个对Flexible的依赖。这违反了SDP。结果，Flexible就不再易于更改了。对Flexible的更改会迫使我们去处理该更改对Stable及其所有依赖者的影响。

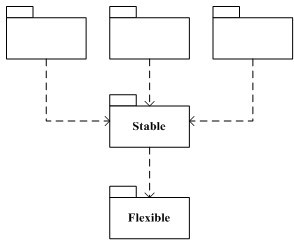


图 违反了SDP的包依赖

要修正这个问题，我们就必须要以某种方式解除Stable对Flexible的依赖。为什么会存在这个依赖关系呢？我们假设Flexible中有一个类C被另一个Stable中的类U使用。

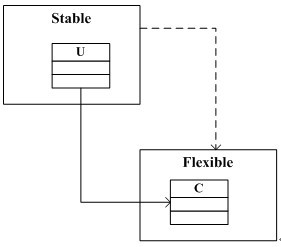


图 糟糕依赖关系的原因

我们可以创建一个接口类IU并把它放到包UInterface中。我们确保IU中声明了U要使用的所有方法。接着，我们让C 从这个接口继承。这就解除了Stable对Flexible的依赖并促使这两个包都依赖于UInterface。UInterface非常稳定，而Flexible仍保持它必须的不稳定性。

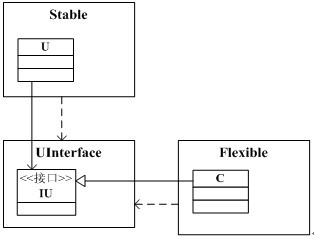


 图 修正稳定性违规

系统中的某些软件不应该经常改变。该软件代表着系统的高层构架和设计决策。我们希望这些构架决策是稳定的。因此，应该把封装系统高层设计的软件放进稳定的包中。不稳定的包中应该只包含那些很可能会改变的软件。

### 稳定抽象原则

稳定抽象原则（简称SAP）告诉我们，包的抽象应该和包的稳定程度一致。

该原则把包的稳定性和抽象性联系起来。它规定，一个稳定的包应该也是抽象的，这样它的稳定性就不会使其无法扩展。另一方面，它规定，一个不稳定的包应该是具体的，因为它的不稳定性使得其内部具体代码易于更改。

因此，如果一个包是稳定的，那么它应该也要包含一些抽象类，这样就可以对它进行扩展。可扩展的稳定包是灵活的，并且不会过分限制设计。

# 通用设计原则

在进行良好设计时，我们希望实现的一些总体目标是：

* 通过降低成本和增加收入来增加利润。对于大多数组织来说，这是中心目标。然而，有许多降低成本的方法，也有许多不同的方法可以增加软件产生的收入。
* 确保我们的设计真正符合需求，从而解决客户的问题。
* 加快开发进度。这有助于降低短期成本，有助于确保软件足够快地进入市场以有效竞争，并且对于满足客户设定的期限可能是必不可少的。
* 提高可用性、效率、可靠性、可维护性和可重用性等软件质量。这些可以帮助降低成本并增加收入。

在本节中，我们将向您介绍在设计软件时应该遵循的一些通用原则。无论哪个设计阶段，努力遵循这些原则将使得设计增加许多优势。

## 分而治之

 分而治之原则可以追溯到有组织的人类活动的最早时期。试图同时处理一件大事通常比处理一系列小事要困难得多。军事战役是这样进行的：指挥官尽量避免同时在所有战线上作战。汽车也是采用分而治之的策略制造的：一些人设计发动机，而另一些人设计车身等。此外，组装汽车的任务也被分成更小、更易于管理的块——每个装配线工人都将专注于一项小任务。

在软件工程中，分而治之原则有多种应用方式。我们已经了解了开发过程如何划分为需求收集、设计和测试等活动。在本节中，我们将研究如何划分软件系统本身。

将软件系统分成多个部分有很多好处：

1. 不同的人可以处理每个部分。因此，原来的开发工作可以并行进行。
2. 单个软件工程师可以专注于他或她的组件，成为该组件的专家。一个人可能对系统的一小部分了如指掌，但不可能对整个系统了如指掌。
3. 每个单独的组件都更小，因此更容易理解。
4. 当需要更换或更改一个部件时，无需更换或大量更改其他部件即可完成。
5. 使组件可重用的机会出现了。

一个软件系统可以有多种划分方式：

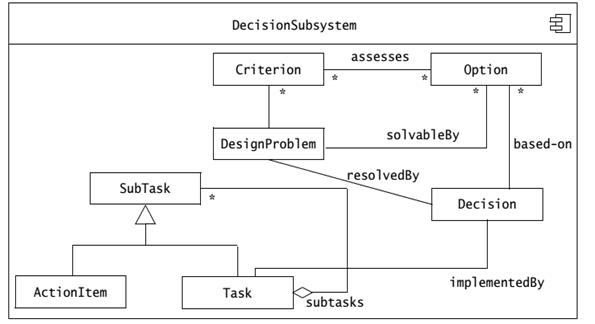
1. 分布式系统分为多个客户端和若干服务器。
2. 系统分为子系统。
3. 一个子系统可以分成一个或多个包。
4. 包由若干类组成。
5. 类由多个属性和方法组成。

## 高内聚低耦合

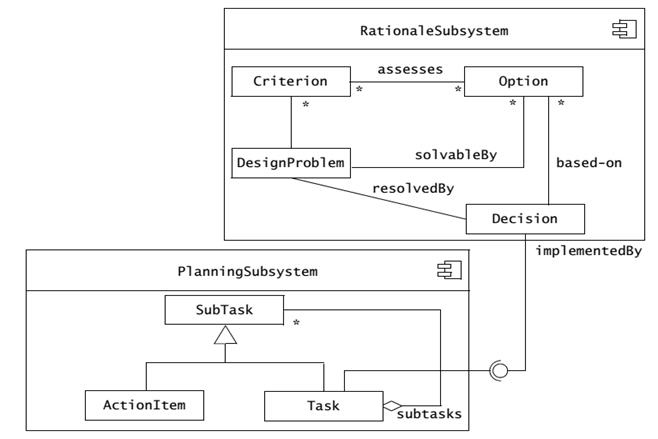
高内聚、低耦合原则是分而治之原则的延伸——分而治之简单地说就是把事情分成更小的块。高内聚、低耦合原则是说要聪明地去做：该分开的分开，该合并的合并。

如果一个子系统或模块将彼此相关的事物放在一起，而将其他事物拒之门外，则它具有高内聚性。这使得整个系统更容易理解和更改。

如果大部分交互发生在子系统内，而不是跨子系统边界，则可以实现高内聚。这里有一些问题要问：一个子系统是否总是为特定服务调用另一个子系统？如果答案是肯定的，请考虑将它们一起移到同一个子系统中。另一个问题是哪些子系统相互调用服务？是否可以通过重构子系统或更改子系统接口来避免这种情况？子系统甚至可以按层次排序（分层）吗？例如，下图向我们展示了一个名为 DecisionSubsystem 的系统，其中包含许多类。



但是，有SubTask和子类AtionItem等类，Task与其他类的关联度较低。所以我们考虑把这个子系统分成两个更小的子系统，如下图所示。



内聚有如下的种类，它们之间的内聚度由弱到强排列如下：

1. 偶然内聚

一个模块内的各处理元素之间没有任何联系，只是偶然地被凑到一起。这种模块也称为巧合内聚，内聚程度最低。

1. 逻辑内聚：

这种模块把几种相关的功能组合在一起， 每次被调用时，由传送给模块参数来确定该模块应完成哪一种功能 。

1. 时间内聚：

把需要同时执行的动作组合在一起形成的模块称为时间内聚模块。

1. 过程内聚：

构件或者操作的组合方式是，允许在调用前面的构件或操作之后，马上调用后面的构件或操作，即使两者之间没有数据进行传递。简单的说就是如果一个模块内的处理元素是相关的，而且必须以特定次序执行则称为过程内聚。

1. 通信内聚：

指模块内所有处理元素都在同一个数据结构上操作或所有处理功能都通过公用数据而发生关联（有时称之为信息内聚）。即指模块内各个组成部分都使用相同的数据数据或产生相同的数据结构。

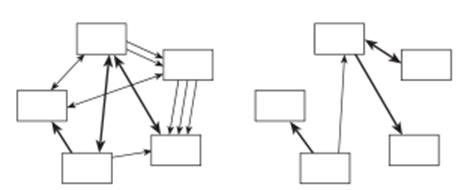
1. 顺序内聚：

一个模块中各个处理元素和同一个功能密切相关，而且这些处理必须顺序执行，通常前一个处理元素的输出时后一个处理元素的输入。例如某模块完成工业产值求值的功能，前一个功能元素求总产值，后一个功能元素求平均产值，显然该模块内两部分紧密关联。顺序内聚的内聚度比较高，但缺点是不如功能内聚易于维护。

1. 功能内聚：

模块内所有元素的各个组成部分全部都为完成同一个功能而存在，共同完成一个单一的功能，模块已不可再分。即模块仅包括为完成某个功能所必须的所有成分，这些成分紧密联系、缺一不可。功能内聚是最强的内聚，其优点是它的功能明确。判断一个模块是否功能内聚，一般从模块名称就能看出。如果模块名称只有一个动词和一个特定的目标（单数名词），一般来说就是功能内聚，如：“计算水费”、“计算产值”等模块。功能内聚一般出现在软件结构图的较低层次上。功能内聚模块的一个重要特点是：它是一个“暗盒”，对于该模块的调用者来说，只需要知道这个模块能做什么，而不需要知道这个模块是如何做的。

内聚指的是子系统内部各个元素之间的联系紧密程度，而耦合指的是两个子系统之间的关系强弱程度。当一个模块与另一个模块之间存在相互依赖关系时，就会发生耦合。下图说明了紧耦合和松耦合系统的概念。



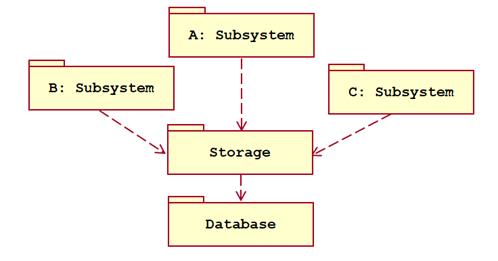
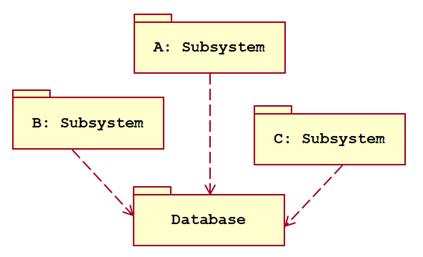
**图**6-1 紧耦合系统（左）和松耦合系统（右）的抽象示例。箭头的粗体表示耦合的强度

一般来说，一组模块耦合得越紧密，就越难理解，因此也就越难改变系统。这样做的两个原因是：

* 当存在相互依存关系时，一个地方的变化将需要其他地方的变化。要求在多个地方进行更改是有问题的，因为找到需要更改的不同地方非常耗时，而且很可能会出错。
* 相互依赖的网络让人很难一眼看出某些组件是如何工作的。

此外，耦合意味着如果你想重用一个模块，你还必须导入与它耦合的模块。这是因为耦合组件需要彼此才能正常工作。

如何实现低耦合？例如，在一个软件系统中，A、B、C 三个子系统都需要访问一个关系数据库。在下面的设计方案中，三个子系统直接访问数据库子系统。在子系统和数据库之间增加一个存储子系统，以屏蔽底层数据库的变化对上层子系统的影响。



耦合可以分为以下几种，它们之间的耦合度由高到低排列如下：

1. 内容耦合

一个模块直接访问另一模块的内容，则称这两个模块为内容耦合。若在程序中出现下列情况之一，则说明两个模块之间发生了内容耦合：

* 一个模块直接访问另一个模块的内部数据。
* 一个模块不通过正常入口而直接转入到另一个模块的内部。
* 两个模块有一部分代码重叠（该部分代码具有一定的独立功能）。
* 一个模块有多个入口。

内容耦合可能在汇编语言中出现。大多数高级语言都已设计成不允许出现内容耦合。这种耦合的耦合性最强，模块独立性最弱。

1. 公共耦合

一组模块都访问同一个全局数据结构，则称之为公共耦合。公共数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。如果模块只是向公共数据环境输入数据，或是只从公共数据环境取出数据，这属于比较松散的公共耦合；如果模块既向公共数据环境输入数据又从公共数据环境取出数据，这属于较紧密的公共耦合。

公共耦合会引起以下问题：

1. 无法控制各个模块对公共数据的存取，严重影响了软件模块的可靠性和适应性。
2. 使软件的可维护性变差。若一个模块修改了公共数据，则会影响相关模块。
3. 降低了软件的可理解性。不容易清楚知道哪些数据被哪些模块所共享，排错困难。

一般地，仅当模块间共享的数据很多且通过参数传递很不方便时，才使用公共耦合。

1. 外部耦合

一组模块都访问同一全局简单变量，而且不通过参数表传递该全局变量的信息，则称之为外部耦合。

1. 控制耦合

模块之间传递的不是数据信息，而是控制信息例如标志、开关量等，一个模块控制了另一个模块的功能。

1. 标记耦合

调用模块和被调用模块之间传递数据结构而不是简单数据，同时也称作特征耦合。表就和的模块间传递的不是简单变量，而是像高级语言中的数据名、记录名和文件名等数据结果，这些名字即为标记，其实传递的是地址。

1. 数据耦合

调用模块和被调用模块之间只传递简单的数据项参数。相当于高级语言中的值传递。

1. 非直接耦合

两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。耦合度最弱，模块独立性最强。

所以，耦合是影响软件复杂程度和设计质量的一个重要因素，为提高模块的独立性，应建立模块间尽可能松散的系统，在设计上我们应采用以下原则：若模块间必须存在耦合，应尽量使用数据耦合，少用控制耦合，慎用或有控制地使用公共耦合，并限制公共耦合的范围，尽量避免内容耦合。

## 面向抽象设计

抽象使得程序稳定而不易变动。在设计时，需要确保允许隐藏或推迟对细节的考虑，从而降低复杂性。赋予设计的这种属性的通用术语是抽象。抽象可以让我们在任何时候只需要处理有限数量的信息。

抽象的工作原理是让您了解事物的本质并在不知道不必要的细节的情况下做出重要的决定。可以通过多种方式提供详细信息：

1. 在设计后期。例如，在创建类图时，最初通常会省略属性的数据类型，并且不会显示关联的实现细节。
2. 由编译器或运行时系统。例如，动态绑定会负责运行哪些方法。
3. 通过使用默认值。例如，绘制操作总是使背景变白，除非采取某些明确的操作来更改默认值。

面向对象的语言直接支持一些抽象，如类和方法。其他的，比如关联，纯粹出现在设计师使用的模型中。

面向抽象设计的主要做法如下：

1. 面向接口设计和编程。当设计一个类时，不让该类面向具体的类，而是面向抽象类或接口，即所设计类中的重要数据是抽象类或接口声明的变量，而不是具体类声明的变量。
2. 封装变化。寻找软件中可能存在的“变化”，然后利用抽象的方式对这些变化进行封装。由于抽象没有具体的实现，就代表了一种无限的可能性，使得其扩展成为了可能。

## 保持适度设计

所谓适度设计，就是指不能设计不足，也不能过度设计。

什么是设计不足？设计出来的系统复用性差，扩展性不强，不能灵活的应对变化，简言之，设计没到位。设计不足，多半是因为经验有限，设计能力有限。

什么是过度设计呢？比如说应用现在还是处在一个单体应用的阶段，那么这时候就去考虑分布式事务该如何进行。或者说你的数据库还是一个单库的时候，就思考如何去做数据异构；甚至业务逻辑在并不复杂的阶段的时候，就去考虑如何使用工作流引擎或规则引擎。当然，并不是说不能去考虑，而是你完成当前的任务，并让系统保证将来可以扩展就行了。所以千万不要为了技术而技术，一定要让技术服务于业务，过度设计是灾难。

我们在软件设计的时候要尽量保持简单，使用一些成熟的、适合业务的技术方案。另外从一个使用者的角度来思考，你设计时要思考如何让自己的架构设计变得简单，足够易用，比如你开发的框架是不是对于接入成本低甚至没有成本? 你设计的框架是否不侵入业务代码？

不仅软件架构设计层面，在代码层面也处处要体现简单原则。代码的可读性和可维护性是衡量代码质量非常重要的两个标准。而该原则就是保持代码可读和可维护的重要手段。代码足够简单，也就意味着很容易读懂，问题比较难隐藏。即便出现问题，修复起来也比较简单。越是能用简单的方法解决复杂的问题，越能体现一个人的能力。

满足该原则需要我们做到如下几点：

1. 不要轻易使用新技术来进行设计。一些技术因为过于新，导致团队中其它人理解设计非常困难。
2. 不要重复造轮子，要善于使用已经有的工具类库。经验证明，自己去实现这些类库，出问题的概率会更高，维护的成本也比较高。
3. 不要过度优化设计。不要为了追求某些设计目标，而过度地使用令人困惑的一些做法。或者大量使用设计模式，相比过程式代码的简单与易于修改，设计模式常常导致代码复杂，增加了理解与修改的成本。
4. 不要太超前设计。不能忽视用户的真实需求，追求一个项目大而全，反而增加了用户的认知和使用成本，影响了真正重点元素的设计。

## 避免重复设计

不要重复你自己，是软件开发的一个基本原则，目的是减少信息的重复。每一个知识或逻辑必须在一个系统中有一个单一的、明确的表示。

为了避免违反该原则，需要把你的系统尽可能地细化。将代码和逻辑划分为更小的可重用单元，并通过在需要的地方调用代码来使用这些单元代码。不要编写过于冗长的方法，要进行逻辑拆分，并尽量使用现有方法中已经写好的逻辑。

在设计中，尽可能利用抽象，把共同的事物抽象出来，放到一个地方去。如果在一个类构造的时候，需要进行成员的初始化，在进行了某些操作以后，同样要进行初始化，那么就可以把“初始化”抽象出来，做成一个方法Initial()，在构造和需要用到的地方调用它。

该原则会给代码维护带来很大的好处。以类的初始化为例，假设类修改了，增加、减少或是修改了成员，如果不写Initial()，那么你可能至少要修改两处，而且，修改之处也可能出现不一致，维护成本大大增加。而写了Initial()方法，那么只要集中修改Initial()就行了。

设计对应者需求，多个重复的需求可能导致多个重复或者相近的类，最后导致重复代码。所以该原则是一个广泛适用的原则。

## 不用过时技术

过时的预期是灵活性设计的一个特例。软件系统使用的技术及其运行环境不可避免地会发生变化。预测过时意味着规划技术或环境的发展，以便软件将继续运行或可以轻松更改。

以下是设计师可以用来更好地预测过时的一些规则：

1. 避免使用早期发布的技术。

直接的问题是早期版本可能比后期版本有更多的缺陷。然而，即使有可能解决一个缺陷，也可能会出现次要问题：如果技术提供者在该技术的后续版本中修复了该缺陷，则原来的解决方法可能不再有效。即使在不存在缺陷的情况下，对技术的改进（在前几个版本中尤其可能出现）也会使使用需要更改的技术的设计出现。

例如，Java的早期采用者后来需要对其代码进行许多更改，因为他们使用的某些类和方法已被弃用。Java设计者在开发改进设计时宣布组件将被弃用——他们不打算无限期地支持旧组件，因此用户被迫更新他们的软件。

1. 避免使用特定于特定环境的软件库。

例如，使用仅在一个操作系统或一种特定类型的硬件中发现的特定功能的软件在遥远的将来不太可能得到支持。

1. 避免使用软件库中未记录的功能或很少使用的功能。

很少使用的功能不仅更容易出现缺陷，而且更重要的是，制造商可能会觉得删除或更改它们不会造成什么损害。另一方面，如果技术提供商对大量使用的功能进行更改，则会遭到许多用户的强烈抗议；因此不太可能进行此类更改。

1. 避免使用来自较小公司或不太可能提供长期支持的公司的可重复使用软件或特殊硬件。

较小的公司更有可能倒闭或没有资源来支持旧版本。建议我们只应该信任大公司似乎有些苛刻——他们也可能倒闭。但是，小公司不得不放弃产品的可能性更高。

1. 使用多个供应商支持的标准语言和技术。

这样做可以让您确信重要技术不会被孤立。今天仍然存在的许多软件系统都是基于晦涩的专有语言。然而，标准不是万灵药：它们可以改变，并且标准的实现可能存在细微差异，这使得很难切换到竞争供应商，即使新供应商表面上支持相同的标准。

## 使用防御思维

不要假定其他人会如何使用你设计的系统。就像汽车司机被教导不要相信其他司机，因此要防御性驾驶一样，软件设计师不应该信任其他设计师或程序员，因此应该进行防御性设计。换句话说，为了增加系统的可靠性，您不仅需要确保自己不添加任何缺陷，而且还必须妥善处理其他代码试图不恰当地使用您的组件的所有情况。具体措施包括：

1. 检查组件的所有输入是否有效。

或者，更准确地说，检查每个组件的先决条件。例如，假设您有一个方法可以确定某个日期是否为工作日。此方法要做的第一件事是检查日期是否有效。不幸的是，过于热心的防御设计可能会导致不必要地一遍又一遍地执行相同的有效性检查。例如，设想以下方法：

public boolean isWorkingDay(String aDate)throws InvalidDateException{

if(!isValidDate(aDate))

throw new InvalidDateException();

return !(isWeekEnd(aDate) || isHoliday(aDate));

}

方法主体的第一行验证日期。但是，由于防御性设计，*isWeekEnd* 和*isHoliday*也可能验证日期。像这样检查日期最多三次是一种计算能力的浪费。

1. 按契约设计

按契约设计是一种允许您以高效和系统的方式进行防御性设计的技术。按契约设计背后的关键思想是每个方法都与其调用者有明确的契约。合同有一组断言，声明：

1. 被调用方法在开始执行时需要哪些先决条件为真。调用者有责任在调用之前使这些先决条件为真。
2. 被调用方法同意确保的后置条件在它完成执行时为真。被调用的方法有责任在返回之前使这些后置条件为真。
3. 被调用方法同意的不变量在它执行时不会改变。前置条件、后置条件和不变量都是布尔表达式。如果它们评估为 false，则表明存在失败。它们类似于我们在后面章节中讨论的 OCL 表达式；事实上，OCL可以用来编写断言。

在程序内部执行断言检查是检测和纠正错误的最有效方法之一。许多语言包含不同的机制来编写断言。在 Java 中，关键字assert在 1.4 版本中已经被引入。例如，在下面的代码中，添加了一个明确的前提条件断言，这意味着必须始终使用有效日期调用该方法。不满足此条件将是一个错误。

public boolean isWeekEnd(String aDate){

assert isValidDate(aDate); // precondition.

return (dayOfTheWeek(aDate)==SUNDAY || dayOfTheWeek(aDate)==SATURDAY);

}

请注意，按契约设计意味着一定程度的信任。就像有一个驾车教练陪着你，边学边讲明道路规则，不让你出事故。但是，如果您在获得驾驶证后犯了错误，您仍然可能发生事故。因此，在主要组件（例如层）之间的边界处，您应该始终严格检查输入。

# 本章小结

系统设计、对象设计、实现是构建系统的最主要活动。分析阶段所作的工作集中在问题域，主要工作是获取用户的需求，搞清楚用户到底希望系统能够做什么，表现出什么样的特性。设计阶段的工作则是使用设计者的知识和经验、科学合理地选择合适的解决域方案来实现用户的需求。分析和设计之间存在紧密的关联关系同时也有显著的区别。

如果软件开发前期设计不当，在后期的开发和维护中，会逐渐“腐化变臭”。通常，无论是开发者还是客户，都希望设计出来的软件产品易扩展、高可用、易用可靠、简单而又安全。与之相反，一个设计糟糕的软件系统，会发出“臭味”，设计者在设计种要尽可能避免这些坏味道。

包图是在UML中用类似于文件夹的符号表示的模型元素的组合。系统中的每个元素都只能为一个包所有，一个包可嵌套在另一个包中。使用包图可以将相关元素归入一个系统。一个包中可包含附属包、图表或单个元素。

组件图表示组件是如何互相组织以构建更大的组件或是软件系统的。组件图显示组件以及它们之间的依赖关系，它可以用来显示程序代码如何分解成模块或组件。一般来说，组件就是一个实际文件。

当我们面对一个复杂的业务系统时，往往需要将大的系统进行分解，直到粒度合适为止。系统分解时需要注意分解的原则。