面向对象程序设计入门导引

S.R. WANG

程序设计方法

- 程序员解决问题的思路
  - 在**机器模型**和实际上要**解决的问题的模型**之间建立联系
  - 早期方法特点：对机器模型
- 问题
  - 程序难以编写、维护代价高
- 方案
  - 合理的程序设计方法

- **机器模型-解空间**
  - 建模该问题的空间，例如在计算机，使用算法和程序的空间
- **解决的问题的模型-问题空间**
  - （物理/现实世界，实际的/抽象的）问题存在的空间

面向对象方法

- 提升建模方式
  - 对要解决的问题进行建模
- **面向对象的方法**
  - 为程序员提供在**问题空间**中表示各种事物元素的**工具**
  - 提供更通用的方法

对象与面向对象程序设计（OOP）

- 对象的引入
  - 问题空间中的事物
  - 解空间中的表示
  - 问题空间中没有对应物的抽象概念/工具
- OOP思想
  - 借助对象概念，程序本身可以根据问题的实际情况进行调整
  - 阅读描述解决方案的代码，也是阅读表达该问题的文字
- OOP优点：
  - 允许用问题空间，而不是解空间，的术语来描述问题

面向对象程序设计方法的特点

- 万物皆对象
  - 对象是特别的变量：可以存放数据；对它“提出请求“，要求它执行对它自身的运算。
  - 在需要解决的问题中，提取出任意概念性的成分
- 程序就是一组对象，对象之间通过发送消息互相通信
  - 将消息看做，对于调用某个特定对象所属函数的请求。
- 每一个对象都有它自己的由其他对象构成的存储区
  - 通过包含已存在对象，创造新新对象：构造复杂程序；隐藏复杂性。
- 每个对象都有一个**类型**
  - 类的最重要的突出特征是“能向它发送什么消息”
- 一个特定类型的所有对象，都能接收相同的消息
  - 通过类型来定义接口

类-数据类型

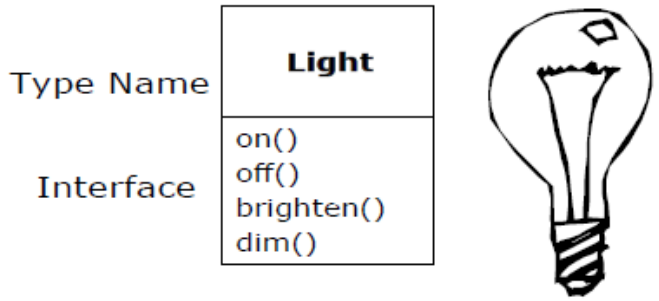
- 创建抽象数据类型是面向对象程序设计的基本思想
  - 创建对象
  - 操纵/使用对象（发送消息或请求），对象根据消息做出响应
- 每个类的成员（对象，实体）
  - 共性：存在属性
  - 自己的状态：具体的属性值
- 类，数据类型，描述了一组对象：
  - 共同特性（数据元素）
  - 相同行为（功能）

类和接口

- 定义类的目的
  - 类的定义为了与具体问题相适应
  - OOP用于我们在设计的系统的一种模拟，类是基本元素
  - 难题之一，在问题空间中的元素和解空间的对象之间建立一对一的映射
- 类的使用
  - 如何向对象作出请求？引入接口

接口

- 可以向对象发出的请求是由它的**接口(interface)** 定义
- 类的实现（implementation）
  - **接口**，规定能向特定的对象发出什么请求
    - 有代码满足这种请求（封装）
  - **隐藏**的数据
- 对象的使用：向对象发送消息（提出请求)
  - 对象根据消息，确定做什么（执行代码）



程序员分类

- **类创建者**：创建类的人，库的设计者
- **客户程序员**：应用程序中，使用类的用户
- 目标不同：
  - **类创建者**：建造类。类只暴露对于客户程序员是必需的东西，其他部分隐藏
  - **客户程序员**：收集一个装满类的工具箱，用于快速应用开发

访问控制

- 在任何关系中， 确定一个所有的参与者都遵从的边界是重要的
- **访问控制**
  - 防止客户程序员插手他们不应当接触的部分。
    - 封装可以突出重要部分，同时隐藏可以被忽略的部分。
  - 允许库设计者去改变这个类的内部工作方式，而不必担心这样做会影响客户程序员
    - 接口和实现的严格分离和被保护，库设计者很容易完成重写任务，用户只需重新连接
    - 可以改变隐藏的部分，而不用担心会影响其他人（内部管理功能，减少误用导致问题）
- 函数的访问控制机制？

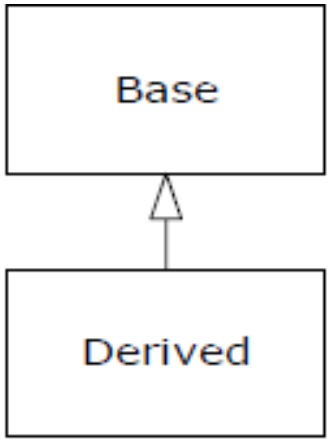
组合-实现的重用

- 代码重用是面向对象程序设计语言的最大优点之一。
- 已经有一个类，简单的重用
  - 直接使用这个类的对象
  - 将这个类的对象放到一个新类的里面
- **组合(composition)，聚合(aggregation)**
  - 由已经存在的类组成新类，得到希望的功能
  - “has-a（有）”关系

- 组合特点（新类中的成员对象通常私有，类的使用者不能访问。因此）
  - 改变新类的成员对象，不会干扰已存在的客户代码
  - 可以在运行时改变这些成员对象，动态地改变程序的行为（注：此时成员为基类指针或引用方式）

继承：重用接口

- 对象的思想本身是一种很方便的工具。使得
  - 可以将数据和功能通过概念**封装**在一起，可以描述合适的问题空间的思想，而不是被迫使用底层机器的用语。
  - 通过使用class机制，这些概念可以被表示为程序设计语言中的基本单元。
- **继承(inheritance)的初始动机**
  - 创建类比较难，创建类似功能的全新类，并不是好的选择，因此
  - 选取已存在的类，先克隆，然后对克隆进行增加和修改



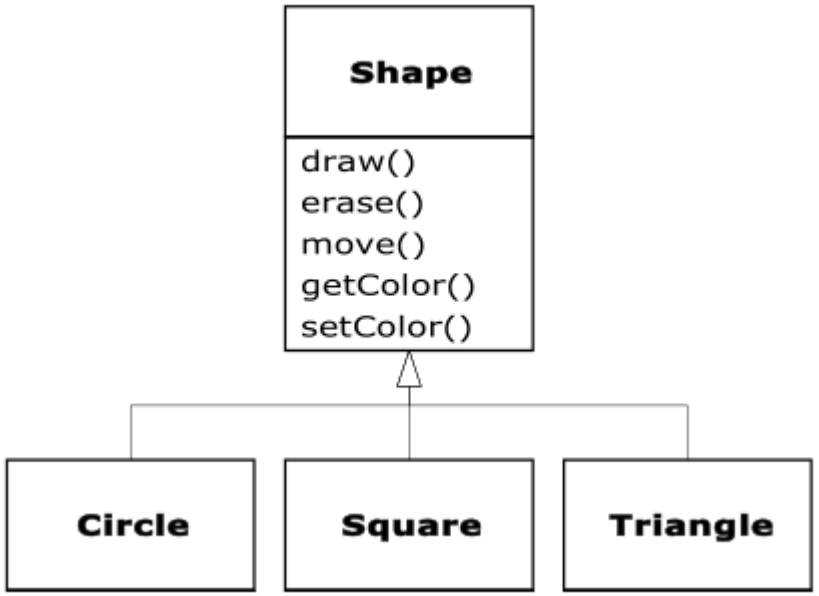
- 基类、超类或父类
- 派生类、继承类或子类

更一般的继承

- 类型除了可以描述一组对象上的约束，也可以描述该类型与其他类型之间的关系
  - 两个类型之间有：共同的特性和行为
  - 一个类型可以包括比另一个类型：有更多特性；处理更多的消息（或对消息进行不同的处理）
- 继承表示了在基类型和派生类型之间的相似性
  - 出发点：一个基类型具有所有由它派生出来的类型所共有的特性和行为。
  - 首先：创建一个基类型以描述关于系统中的一些对象的思想核心。
  - 其次：由这个基类型，可以派生出其他类型来表述实现该核心的不同途径。

类的层次结构

- 一个基类型具有所有派生出来共有的特性和行为。由基类型可以派生出其他类型来表述实现该核心的不同途径。
- 使用继承可以建立类型的**层次结构**，并使用其类型术语来
- 使用与问题相同的术语描述问题的解，优点
  - 无须在问题的描述和解的描述之间使用许多的中间模型。
  - 使用对象，类的层次结构就是最初的模型。能直接从实际世界中的描述。

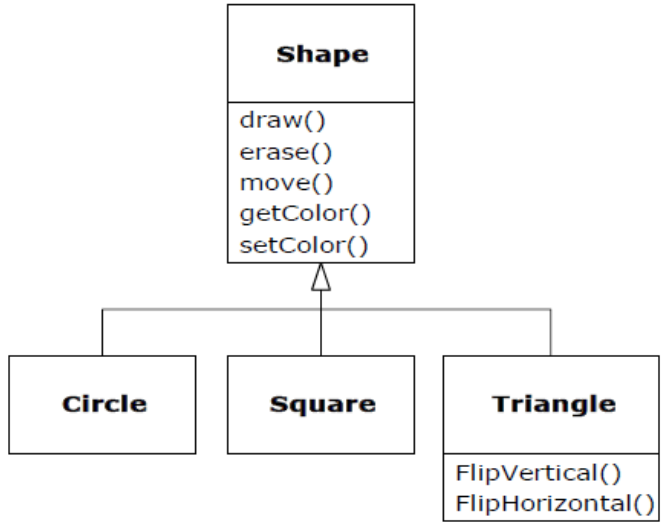


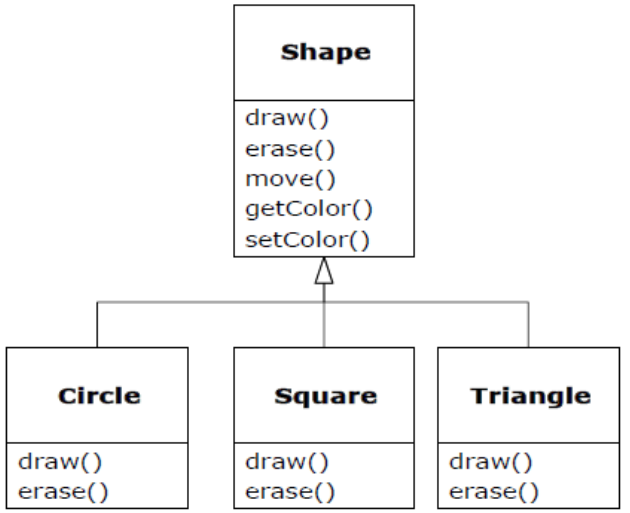
类型等价性

- 继承创造了新类型。新类型
  - 包含已经存在的类型的所有成员
  - 复制了基类的接口。即，所有能够发送给这个基类对象的消息，也能够发送给这个派生类的对象。
- 根据发送给一个类的消息知道这个类的类型
  - 这意味这派生类与基类**相同类型**。
  - 通过继承实现类型等价性

重载：接口的重用

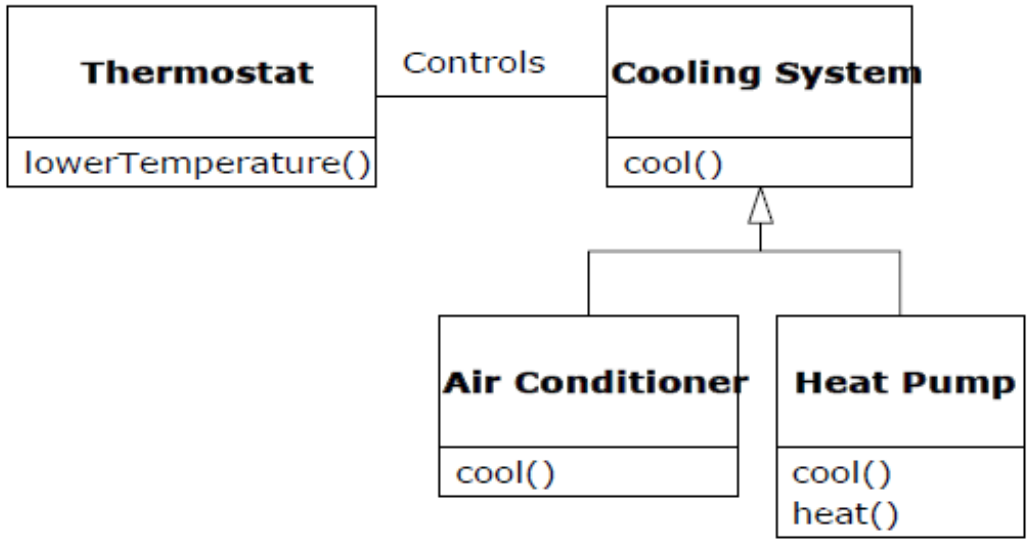
- 简单的继承接口与实现
  - 派生类的对象不仅有相同的类型，而且有相同的行为。
- 向派生类添加全新的函数
  - 新函数不是基类接口的一部分
- 改变已经存在的基类函数的行为，这称为函数重载(overriding)
  - 使用同一个接口函数，但它为新类型做不同的事情。





Is-a vs. is-like-a

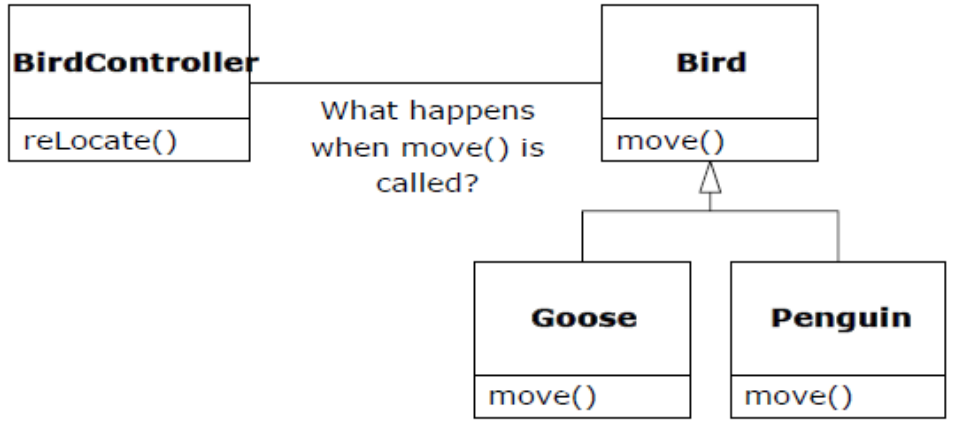
- pure substitution
  - 接口完全相同
- is-like-a
  - 派生类有基类里没有的新接口
  - 基类不知道派生类中扩展的部分



具有多态性的可互换对象

多态机制的动机

- 处理类型层次结构时，常常希望不把对象看做是某一特殊类型(具体派生类)的成员，而是其基本类型(基类)的成员，从而编写不依赖特殊(以及扩展)类型的代码。
- 形体的例子中，函数可以对一般形体进行操作
  - 不关心它们是圆形、正方形还是三角形。所有的形体都能被绘制、擦除和移动
  - 函数能简单地发送消息给一个形体对象(基类类型)，而不考虑这个对象如何处理这个消息
  - 程序代码不受增添新类型的影响。通过派生新的子类型，可以很容易扩展程序。极大地改善软件设计



多态的实现机制：动态绑定

- 原理：把派生类型的对象看做是它们所属的基本类型（基类） 。
- 做法：针对一个一般类型（基类）的对象发出消息
- 表现：对象根据它的实际类型来执行合适的代码。
  - 如果增加一个新的子类型，不用修改函数调用，它就可以执行不同的代码。
- 实现机制：晚捆绑(late binding)

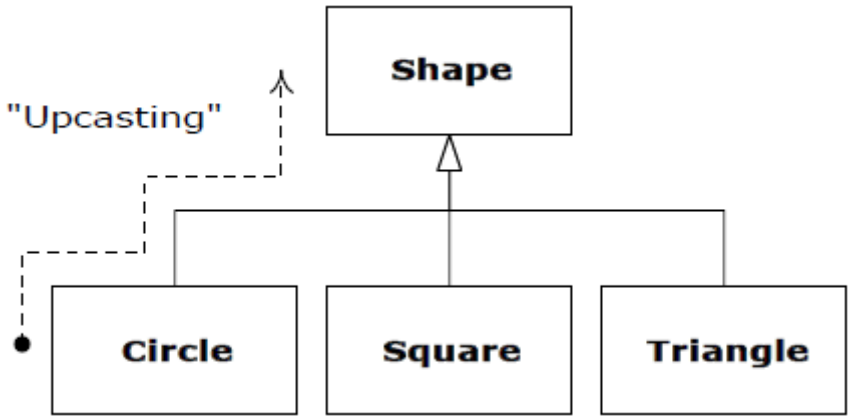
晚捆绑

- 晚捆绑：当给对象发送消息时，在程序运行时才去确定被调用的代码。
  - 编译器保证被调用的函数存在，并执行参数和返回值的类型检查
  - 编译器并不知道将执行的代码是什么样的。
- 用关键字virtual 声明希望某个函数有晚捆绑的灵活性

- 把函数体与函数调用相联系称为**捆绑(binding)**
  - 当捆绑在程序运行之前（由编译器和连接器）完成时，这称为**早捆绑**(early binding)
  - **晚捆绑**(late binding) 。捆绑根据对象的类型，发生在运行时。又称为**动态捆绑**(dynamic binding)或**运行时捆绑** (runtime binding)

多态行为的启用

- doStuff()函数与任何类型的Shape对话，独立于对象的特定类型
  - doStuff()的调用，s会自动做正确的工作，而不管调用对象s的确切类型。
- “s是一种形体，s能erase()和draw()自己，如法操作，并注意细节的正确性。”
- 调用时会发生**向上类型转换(upcasting)**
  - 派生类型处理就如同基类型处理过程一样



多态的优点

- 多态性（C++虚函数实现）是面向对象程序设计基本特征
  - 数据抽象/封装、继承、多态
- (polymorphism) 提供了接口与具体实现之间的另一层隔离，从而将“what"与"how"分离开来。
  - 即，接口不变的情况下，具体实现可以变化
  - 实现方式：虚函数根据类型来处理解隅
- 多态性改善了代码的组织性和可读性，同时也使创建的程序具有可扩展性
  - 程序在项目的最初创建期 “扩展”
  - 在项目需要有新的功能时，也能“扩展”。

C++用法的三个层次

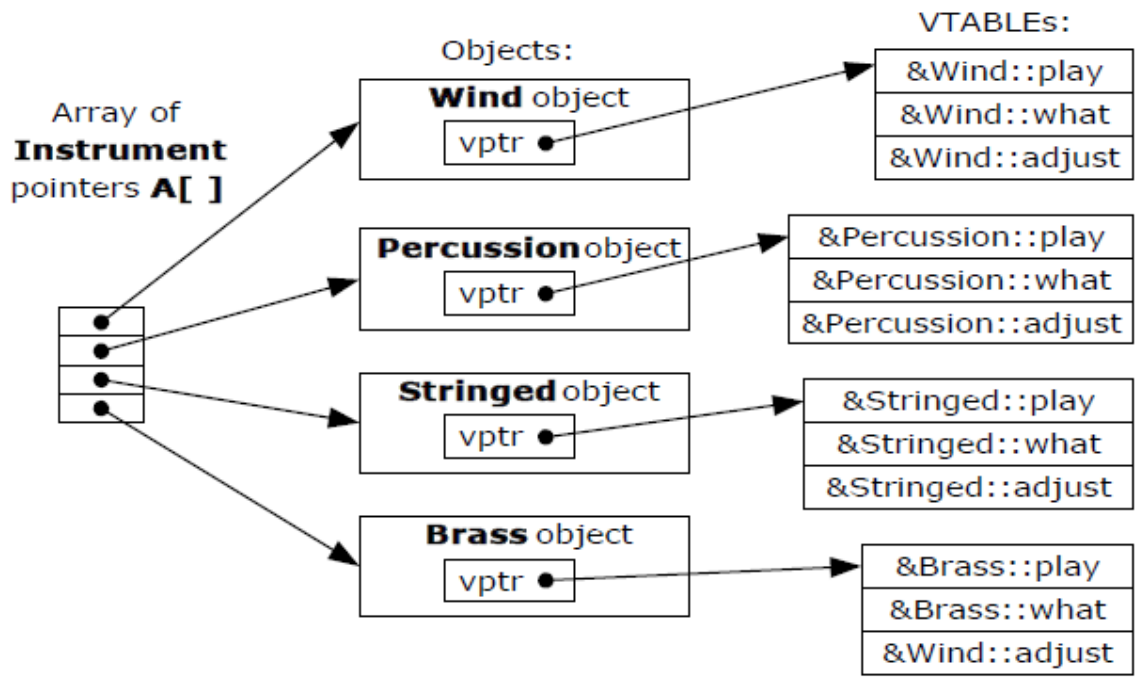
- 简单地把C++作为一个“更好的C”
- “基于对象”的C++ 。这意味着
  - 很容易看到将数据结构和在它上面活动的由数捆绑在 一起的代码组织好处
  - 还可以看到构造函数和析构函数的价值
  - 也许还会看到一些简单的继承。
- 面向对象程序设计(OOP)
  - 最为重要的特性

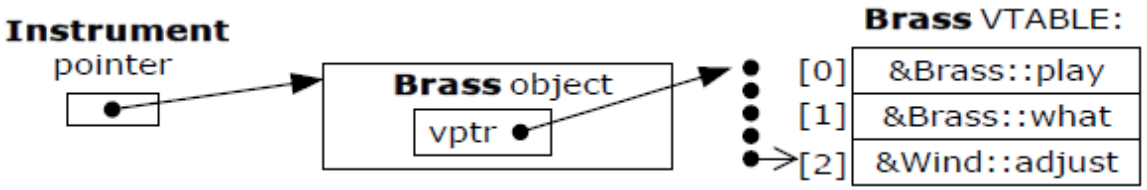
```
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

虚函数机制

- 对每个包含虚函数的类创建一个表（称为VTABLE)

1. 创建VTABLE，编译器放置特定类的虚函数的地址
2. 编译器隐含加入一个数据成员：指针vpointer（VP
   VTABLE。此时，对象就“知道” 它自己是什么类
3. 当通过基类指针做虚函数调用（多态调用）时，通
   VTABLE，取得函数地址（函数在VTABLE中的位





更多细节

- C+＋由数调用的参数与C函数调用一样， 是从右向左进栈的
- 调用每个成员函数时this都必须作为参数压进栈， 所以成员函数知道它工作在哪个特殊对象上。
- 在对象固定位置取VPTR（在对象的开头）
- 构造函数内部自动初始化VPTR

- C+＋中，晚捆绑可选的
  - 因为它不是相当高效的

OOP分析和设计

- 方法(method)［ 通常称为方法论(methodology) ］是一系列的过程和探索，用以降低程序设计问题的复杂性 。
- 如果我们正在考虑的是一个包含丰富细节而且需要许多步骤和文档的方法学，将很难判断什么时候停止。应当牢记我们正在努力寻找的是什么：
- (1) 有哪些对象？（如何将项目分成多个组成部分？）
- (2) 它们的接口是什么？（需要向每个对象发送什么信息？）

分析和设计5个阶段

- 第0阶段：制定计划
  - 任务陈述。高层概念(high concept)设定了项目的基调。用一句或两句话表述。
- 第 1 阶段：我们在做什么
  - 建立需求分析(requirements analysis) 和系统规范说明(system specification)”
- 第 2阶段：我们将如何建立对象
  - 做出设计，描述这些类和它们如何交互。
- 第3阶段：创建核心
  - 从粗线条设计向编译和执行可执行代码体的最初转换阶段
- 第4阶段：迭代用例
  - 在一次迭代(iteration) 期间，我们增加一组特征． 一次迭代是 一个相当短的开发时期。
- 第5阶段；进化
  - 开发周期中，传统上 称为“维护 " 的一 个阶段，