码农的自我修养之
软件的结构、特性和描述方法

孟宁

关注孟宁

一、软件是什么?

- 软件的基本构成元素
- 软件的基本结构
- 软件中的一些特殊机制
- 软件的内在特性

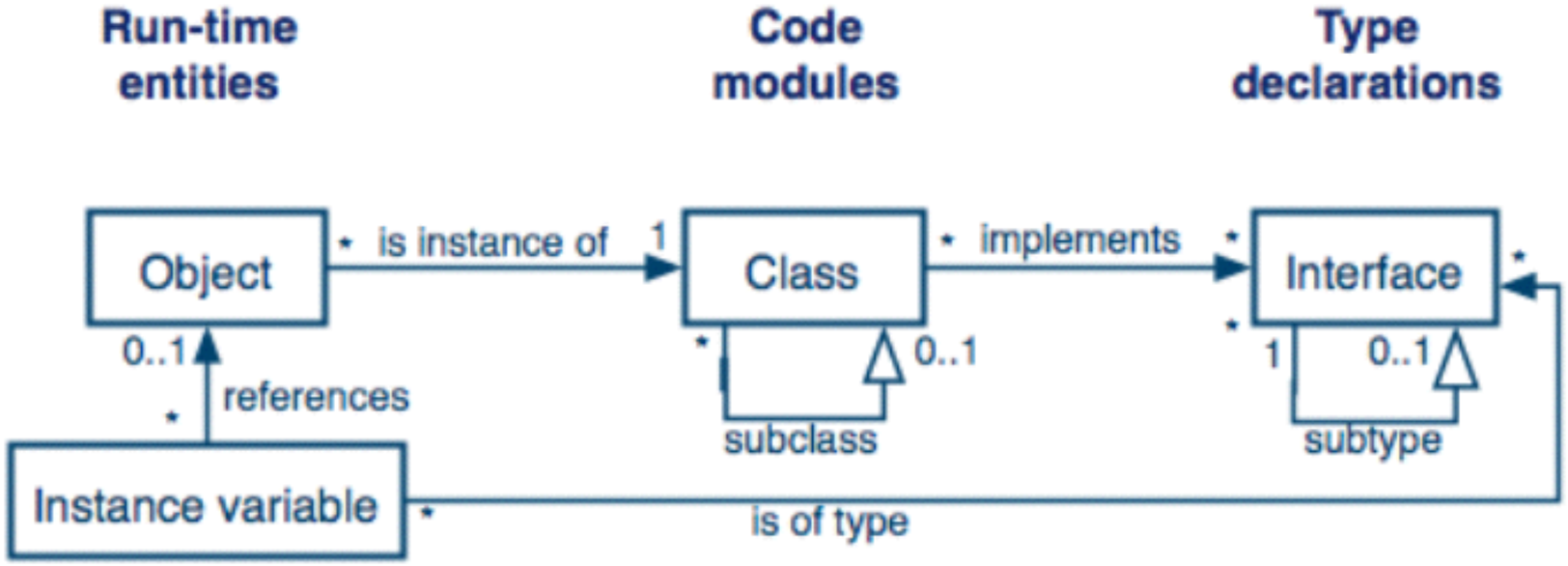
软件的基本构成元素

- 对象（Object）
- 函数和变量/常量
- 指令和操作数
- 0和1是什么?

对象（Object）

- 一个对象作为某个类的实例，是属性和方法的集合。对象和属性之间有依附关系，属性用来描述对象或存储对象的状态信息，属性也可以是一个对象。对象能够独立存在，对象的创建和销毁显式地或隐式地对应着构造方法（constructor）和析构方法(destructor)，。

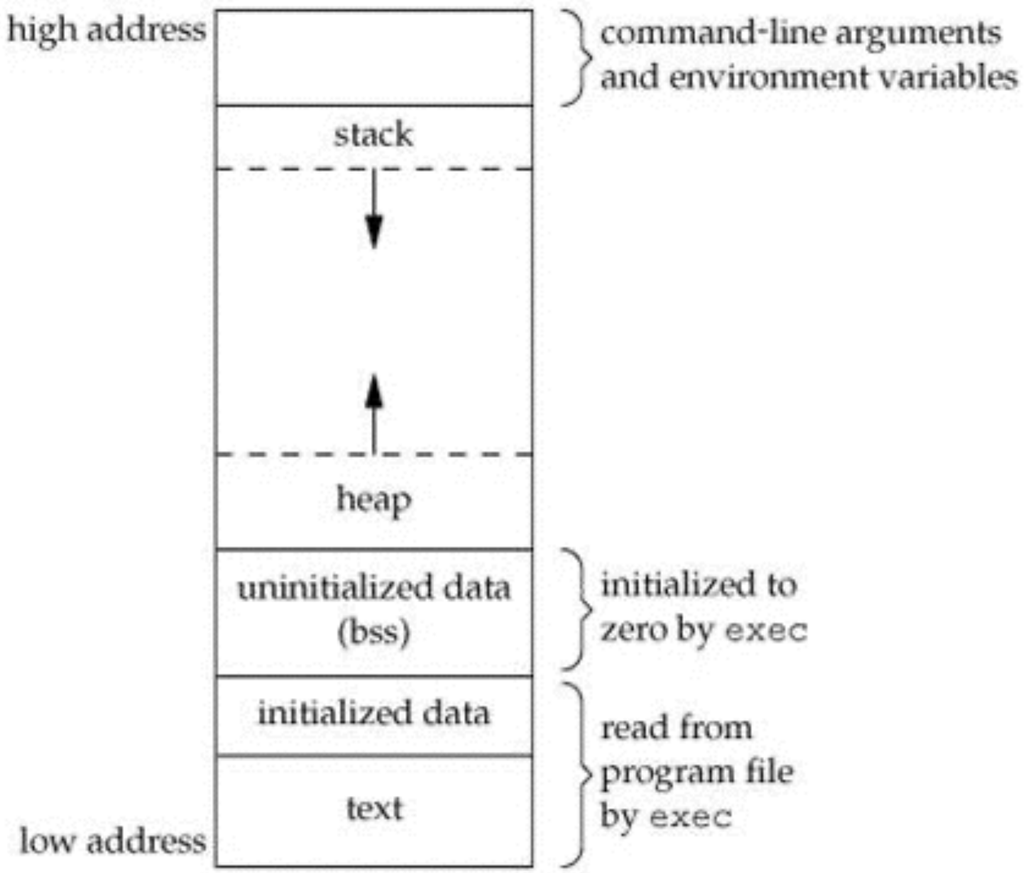
- 面向对象是一种对软件抽象封装的方法



函数和变量/常量

- 我们看看用函数和变量/常量作为软件基本元素的抽象方法。相对来讲面向对象是一种更高层的软件抽象封装的方法，其中方法大致对应函数，属性大致对应变量/常量。由于函数和变量/常量的相关概念相信您非常熟悉，我们这里仅讨论它们的进程地址空间分布。

- 从图中可以看出从低地址到高地址分别内存区分别为代码段、已初始化数据段、未初始化数据段（BSS）、堆、栈和命令行参数和环境变量。

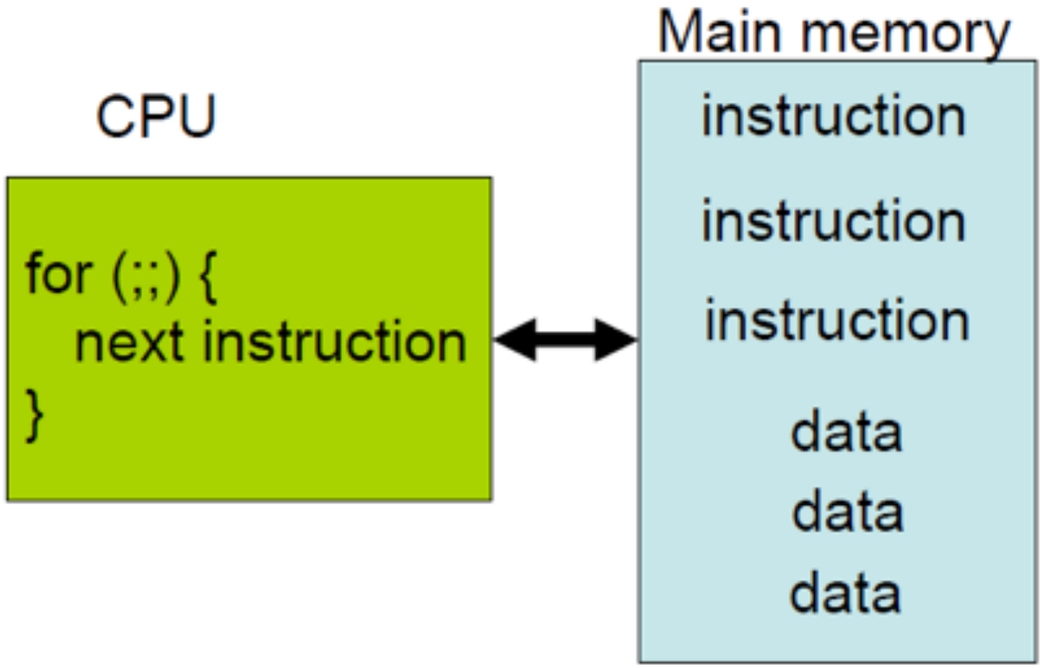


函数和变量/常量

- 面向对象的软件抽象层次更高与进程地址空间对应起来更为复杂，我们以面向过程的编程语言C语言为例简要分析一下函数、常量、全局变量、静态变量和局部变量的存储空间分布。
- - 全局常量（const）、字符串常量、函数以及编译时可决定的某些东西一般存储在代码段（text）；
- - 初始化的全局变量、初始化的静态变量（包括全局的和局部的）存储在已初始化数据段；
- - 未初始化的全局变量、未初始化的静态变量（包括全局的和局部的）存储在未初始化数据段（BSS）；
- - 动态内存分配（malloc、new等）存储在堆（heap）中；
- - 局部变量（初始化以及未初始化的，但不包含静态变量）、局部常量（const）存储在栈（stack）中；
- - 命令行参数和环境变量存储在与栈底紧挨着的位置。
- 如果我们堆和栈扩大，把命令行参数和环境变量作为调用main函数的栈空间，把已初始化数据段和未初始化数据段（BSS）作为扩大的堆空间的一个部分，我们就可以简单化为代码+堆栈的地址空间分布，而且这种简单化与面向过程的软件抽象元素函数和变量/常量保持逻辑一致。

指令和操作数

- 指令（instruction）是由 CPU 加载和执行的软件基本单元。一条指令有四个组成部分：标号、指令助记符、操作数、注释，其中标号和注释是辅助性的，不是指令的核心要素。一般指令可以表述为指令码+操作数。
- 指令码可以是二进制的机器指令编码，也可以是八进制的编码，程序员更喜欢用汇编语言指令助记符，如 mov、add 和 sub，给出了指令执行操作的线索。
- 操作数有 3 种基本类型：立即数。用数字文本表示的数值；寄存器操作数。使用 CPU 内已命名的寄存器；内存操作数。引用内存位置。



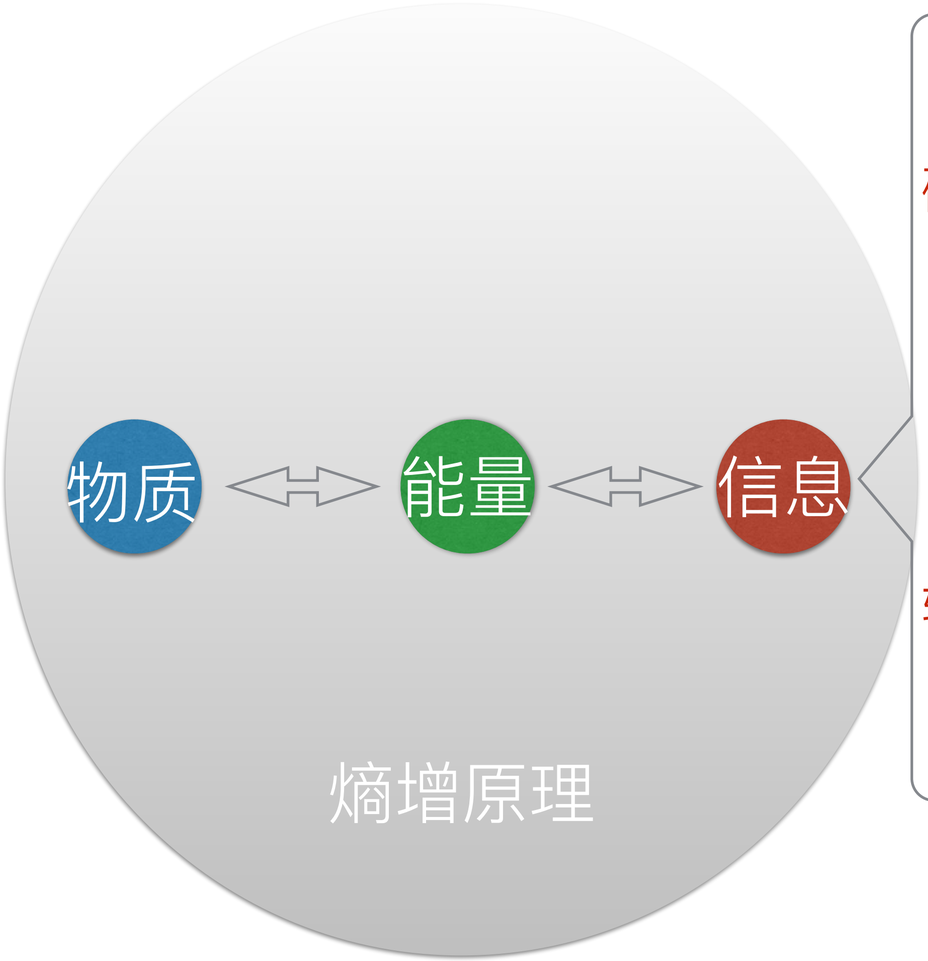
0和1是什么?

- 指令是由0和1构成的特定结构的信息，0和1就是软件的基本信息元素。如果继续追问什么0？什么是1？什么是信息？那就会陷入哲学上的探索。我们这里也简要介绍人类探索世界本源的脉络，如果您感兴趣的话，可以继续探索软件的本质和信息的本源。

人类探索世界本源的脉络

- 生死追问
  - 生死是价值观的内核
  - 信仰是价值观和世界观的圆满状态
- 唯物主义
  - 阴阳五行学说/水是万物之源、原子、夸克/量子、弦
- 认识论、唯心主义和语言学
  - 从认知客观世界深入到对认知本身的探索，乃至对认知的表述方式的探究
- 唯信息主义
  - 对感知、意识和逻辑的本源有了基本认识之后，得出了万物求存万物分化演化的递弱代偿原理
  - 万物演化的基本内核是信息结构的变化，那什么是信息？维护信息结构及万物求存有着怎样的动因?

宇宙观和世界观



简单 → 复杂

硬态

物质世界：形态、分子、基因

生物社会：信号、语言、符号

人类社会：概念、模型、思想

软态

虚拟世界：代码、模式、架构

直观 → 抽象

熵减定律

社会观和方法论

人造太阳计划
基因编辑技术

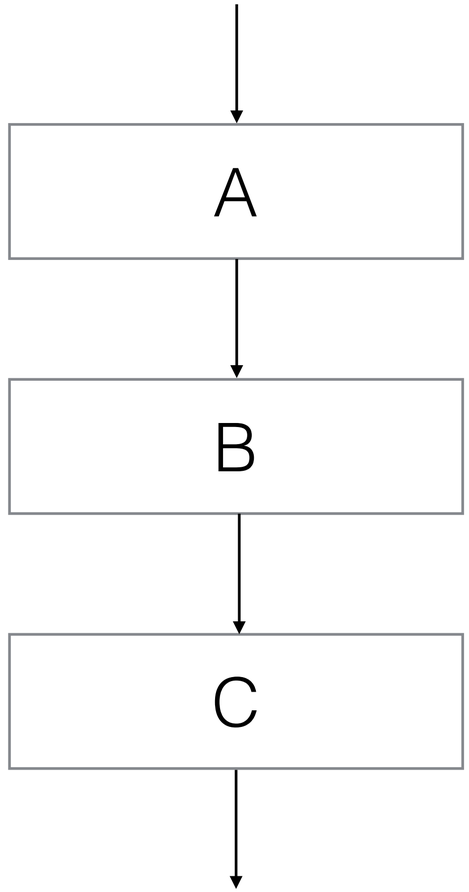
	生物社会	农牧业文明	工商业文明	信息文明
生存方式	光合作用 采集渔猎	种植粮食 养殖动物	分工生产 服务贸易	编写代码 信息消费
与自然的关系	生物是大自然的一部分	天人合一	人定胜天	构建虚拟世界
方法论	基因变异 病毒感染	博物学方法	科学方法	软件方法
	基因序列	经验模型	逻辑模型	抽象模型
	进化	归纳	演绎	重构
思维方式	本能冲动	工匠精神	逻辑思维	计算思维

软件的基本结构

- 顺序结构
- 分支结构
- 循环结构
- 函数调用框架
- 继承和对象组合

顺序结构

- 顺序结构是最简单的程序结构，也是最常用的程序结构，只要按照解决问题的顺序写出相应的语句就行，它的执行顺序是自上而下，依次执行。

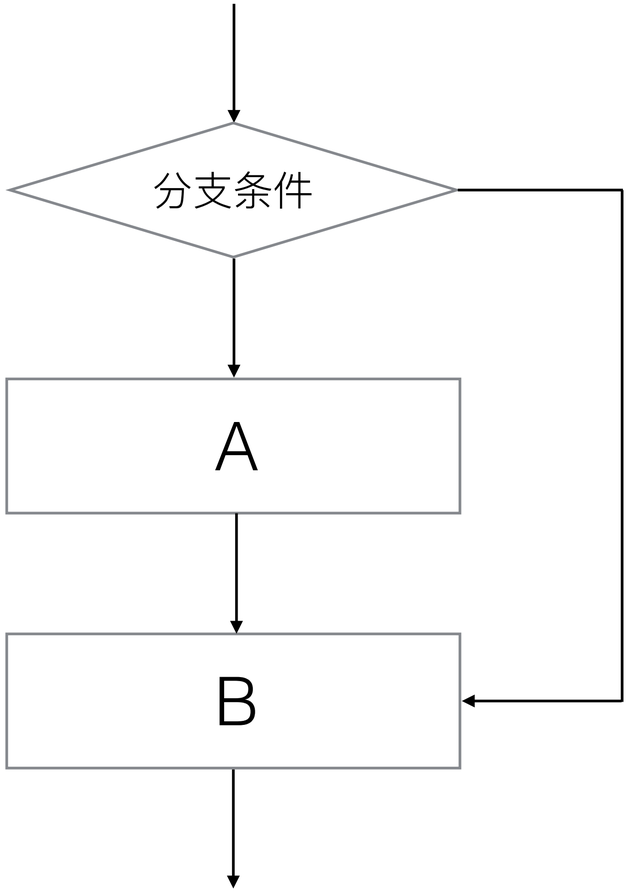


分支结构的实现方法

- 分支结构是在顺序结构的基础上，利用影响标志寄存器上标志位的指令和跳转指令组合起来借助于标志寄存器或特定寄存器暂存条件状态实现分支结构。这么说起来不是很直观，我们具体以X86指令集为例来看看分支结构的实现方法。

- 在X86体系结构下布尔指令、比较指令和能设置标志位的指令都可以影响标志寄存器（FLAGS Register），比如零标志位、进位标志位、符号标志位、溢出标志位和奇偶标志位等。X86指令集包含了 AND、OR、XOR 和 NOT 指令，它们能直接在二进制位上实现布尔操作，还有TEST指令是一种非破坏性的 AND 操作，另外CMP、STC、CLC、INC等指令都能影响标志位。

- 条件跳转指令测试标志位决定是否跳转到指定的地址。X86指令集包含大量的条件跳转指令。有的条件跳转指令还能比较有符号和无符号整数，并根据标志位的值来执行跳转。



条件跳转指令的四种类型

- 基于特定标志位的值跳转。比如JZ为零跳转、JNO无溢出跳转、JNZ非零跳转、JS有符号跳转 、JC进位跳转、JNS无符号跳转、JNC无进位跳转、JP偶校验跳转、JO溢出跳转、JNP奇校验跳转；

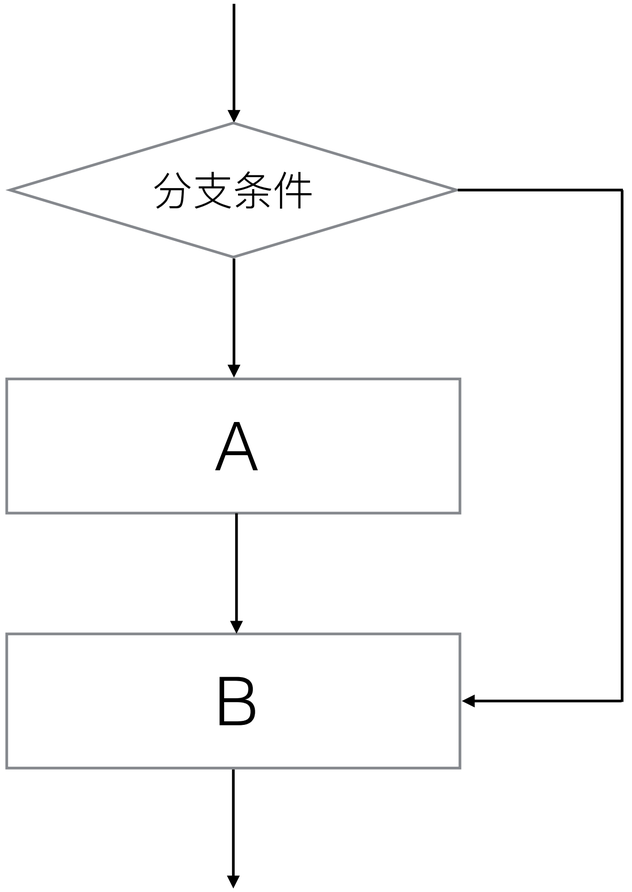
- 基于CMP指令比较两个数是否相等，或一个数与特定寄存器（CX、ECX、RCX）比较是否相等跳转。比如JE相等跳转、JNE不相等跳转、JCXZ当寄存器CX=0跳转、JECXZ当寄存器ECX=0跳转、JRCXZ当寄存器RCX=0跳转（RCX为64位模式下的寄存器）；

- 基于无符号操作数的比较跳转。比如JA大于跳转、JB小于跳转、JNBE不小于或等于跳转（与 JA 相同）、JNAE不大于或等于跳转（与 JB 相同）、JAE大于或等于跳转、JBE　小于或等于跳转、JNB　不小于跳转（与 JAE 相同）、JNA　不大于跳转（与 JBE 相同）；

- 基于有符号操作数的比较跳转。比如JG大于跳转、JL小于跳转、JNLE不小于或等于跳转（与 JG 相同）、JNGE 不大于或等于跳转（与 JL 相同）、JGE大于或等于跳转、JLE小于或等于跳转、JNL不小于跳转（与 JGE 相同）、JNG不大于跳转（与 JLE 相同）。

分支结构

- 将影响标志寄存器上标志位的指令和根据标志寄存器或特定寄存器执行跳转的指令结合起来就可以实现分支结构。看到分支结构在底层指令层面有如此庞大数量的实现方法，作为用高级语言写代码的您是不是幸福感十足^o^。在这些庞大数量的实现上我们以后见之明可以轻松去繁就简清晰地抽象出其中存在的分支结构，如图所示。



- 以C语言为例分支结构在高级语言上至少有三种方式实现：if-else语句、switch语句和goto语句。

if-else语句是典型的条件分支结构

```
char c;
printf("Input a character:");
c = getchar();
if( c < 32)
    printf("This is a control character\n");
else if(c>='0' && c<='9')
    printf("This is a digit\n");
else if(c>='A' && c<='Z')
    printf("This is a capital letter\n");
else if(c>='a' && c<='z')
    printf("This is a small letter\n");
else
    printf("This is an other character\n");
```

switch语句方便处理较大数量的条件匹配

```
switch(day)
{
        case 1: printf("Monday\n"); break;
        case 2: printf("Tuesday\n"); break;
        case 3: printf("Wednesday\n"); break;
        case 4: printf("Thursday\n"); break;
        case 5: printf("Friday\n"); break;
        case 6: printf("Saturday\n"); break;
        case 7: printf("Sunday\n"); break;
        default:printf("error\n"); break;
}
```

goto语句也有经典的用法

- 尽管goto语句的多数用法有害，但是goto无条件跳转作为分支结构的特例也有经典的用法，比如在函数内将异常处理的代码独立出来放到函数结尾的做法就很好。

```
...
if (do_something() == ERR)
    goto error;
if (do_something2() == ERR)
    goto error;
if (do_something3() == ERR)
    goto error;
if (do_something4() == ERR)
    goto error;
...
error:
    //异常处理代码
```

循环结构

- 循环结构是顺序结构和分支结构的组合起来形成的更为复杂的程序结构，是指在程序中需要反复执行某个功能而设置的一种程序结构，可以看成是一个条件判断语句和一个向前无条件跳转语句的组合。

- C语言中提供四种循环，即goto循环、while循环、do…while循环和for循环。四种循环可以用来处理同一问题，一般情况下它们可以互相代替，但一般不提倡用goto循环，因为强制改变程序的顺序结构经常会给程序的运行带来不可预料的错误，一般我们主要使用while、do-while、for三种循环。

while、do-while、for三种循环

while循环的一般形式为：

```
while(表达式)
{
    语句块
}
```

do-while循环的一般形式为：

```
do {
    语句块
} while(表达式);
```

for 循环的一般形式为：

```
for(表达式1; 表达式2; 表达式3)
{
    语句块
}
```

while、do-while、for三种循环的具体语法细节含义我们不在此赘述。

函数调用框架

- 函数调用框架是以函数作为基本元素，并借助于堆叠起来的堆栈数据，所形成的一种更为复杂的程序结构。函数内部可以是顺序结构、分支结构和循环结构的组合。堆叠起来的堆栈数据用来记录函数调用框架结构的信息，是函数调用框架的灵魂。

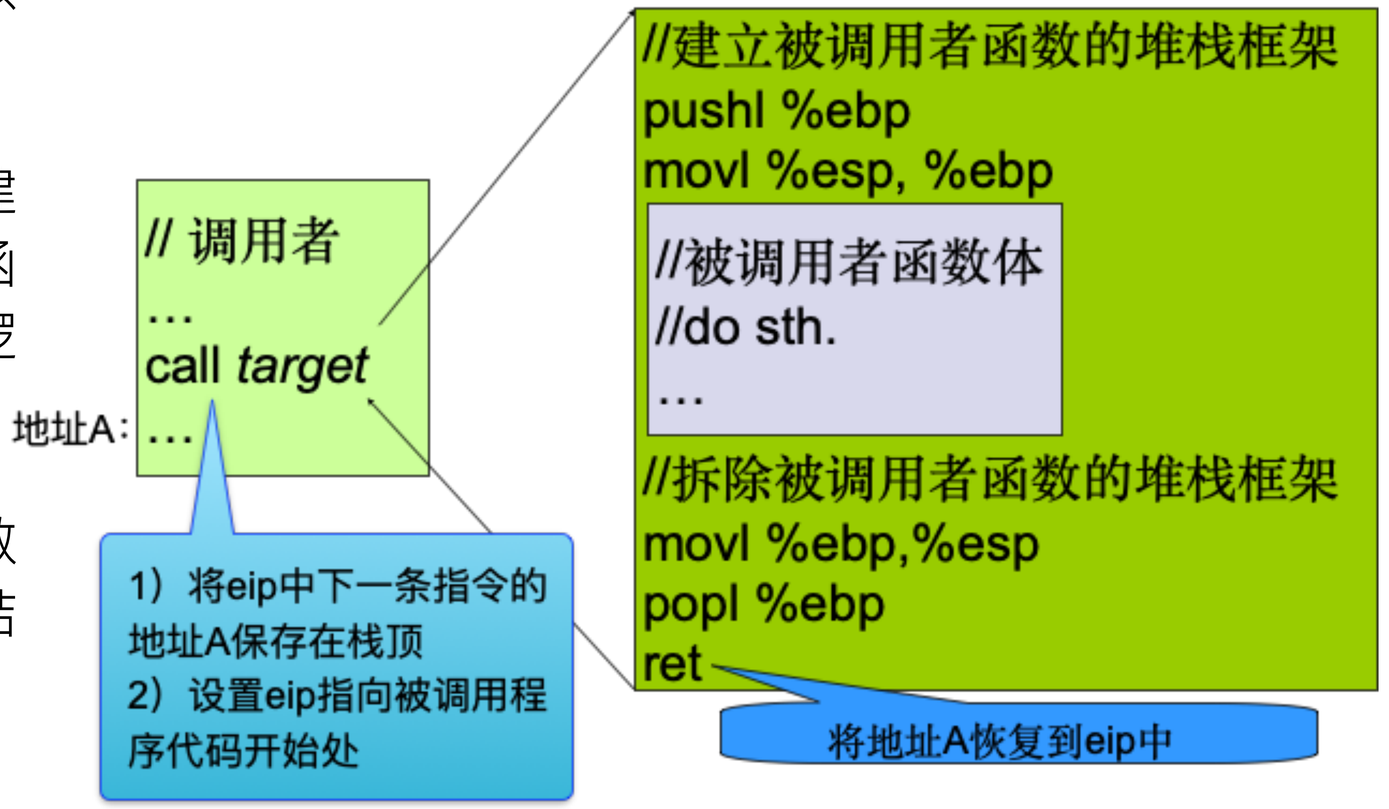
- 为了理解函数调用的堆栈框架，我们需要先来了解一下堆栈操作。以X86指令集为例函数调用相关的寄存器和指令如下：

函数调用框架

- esp，堆栈指针寄存器（stack pointer）。
- ebp，基址指针寄存器（base pointer）。
- eip，指令指针寄存器，为了安全考虑程序无权修改该寄存器，只能通过专用指令修改该寄存器，比如call和ret。
- pushl，压栈指令，由于X86体系结构下栈顶从高地址向低地址增长，所以压栈时栈顶地址减少4个字节。
- popl，出栈指令，栈顶地址增加4个字节。
- call，函数调用指令，该指令负责将当前eip指向的下一条指令地址压栈，然后设置eip指向被调用程序代码开始处，即pushl eip和eip = 函数名（指针）。
- ret，函数返回指令，该指令负责将栈顶数据放入eip寄存器，即popl eip，此时的栈顶数据应该正好是call指令压栈的下一条指令地址。
- 注意以e开头的寄存器位32位寄存器，指令结尾的l是指long，也就是操作4个字节数据的指令。

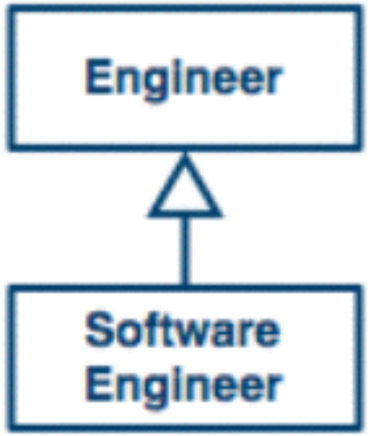
函数调用框架

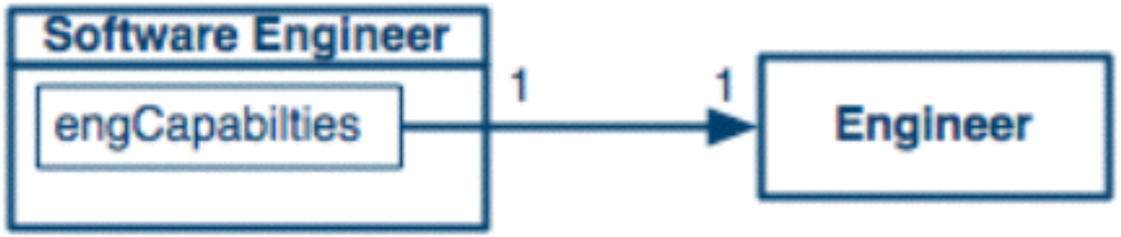
- 函数调用过程中ebp寄存器用作记录当前函数调用基址，当调用一个函数时首先建立该函数的堆栈框架，函数返回时拆除该函数的堆栈框架，大致如下示意图所示。
- 随着函数调用一层层深入，堆栈上依次建立一层层逻辑上的堆栈堆叠起来；随着函数一层层返回上一级函数，堆叠起来的逻辑上的堆栈也一层层拆除。
- 就这样函数调用框架借助堆叠起来的函数堆栈框架形成了一种复杂而缜密的程序结构。



继承和对象组合

- 继承和对象组合都是以对象（类）作为软件基本元素构成的程序结构。对象是基于属性（变量或对象）和方法（函数）构建起来的更复杂的软件基本元素，显然它涵盖了前述包括函数调用框架在内所有程序结构，它是一个更高层、更复杂的抽象实体。基于对象（类）之间的继承关系所形成的两个对象各自独立、两个类紧密耦合的程序结构；对象组合则将一个对象作为另一个对象的属性，从而形成两个对象（类）之间有明确的依赖关系，下图可以清晰地对比继承和对象组合的特点。





继承和对象组合

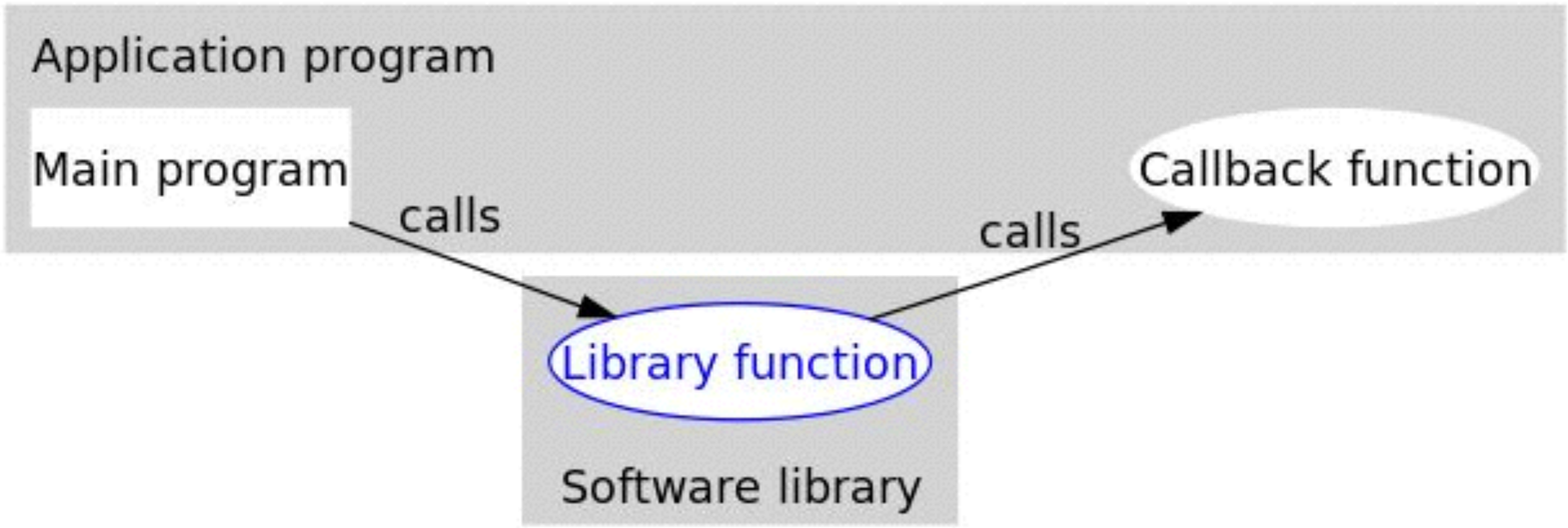
- 继承的概念非常容易理解，于是被广为接受，但是面向对象的真正威力是封装，而非继承! 封装的威力集中体现在对象组合对继承的替代作用上。继承可以重用代码，但是破坏了代码的封装特性，增加了父类与子类之间的代码模块耦合，因此我们需要避免使用继承，在大多数情况下，应该使用对象组合替代继承来实现相同的目标。遗憾的是由于继承的用法被广泛接受，以致反过来造成了对封装的误解。

软件中的一些特殊机制

- 面向对象有三个关键的概念：继承（Inheritance）、对象组合（object composition）和多态（polymorphism），其中多态是一种比较特殊的机制，另外还有回调函数（callback）、闭包（closure）、异步调用和匿名函数也经常用到特殊机制。这几个特殊机制在一些设计模式中比较比较常用，在实际应用中它们又常常交叉综合出现，在理解上带来很多困扰。我们简单做一个介绍。
  - 回调函数
  - 多态
  - 闭包
  - 异步调用
  - 匿名函数

回调函数

- 回调函数是一个面向过程的概念，是代码执行过程的一种特殊流程。回调函数就是一个通过函数指针调用的函数。把函数的指针（地址）作为参数传递给另一个函数，当这个指针调用其所指向的函数时，就称这是回调函数。回调函数不是该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。



典型的回调函数用法举例

```
//回调函数
int SearchCondition(tLinkTableNode * pLinkTableNode, void * args)
{
    char * cmd = (char*) args;
    tDataNode * pNode = (tDataNode *)pLinkTableNode;
    if(strcmp(pNode->cmd, cmd) == 0)
    {
        return  SUCCESS;
    }
    return FAILURE;
}
//传递回调函数
SearchLinkTableNode(head,SearchCondition,(void*)cmd)
//执行回调函数
tLinkTableNode * SearchLinkTableNode(tLinkTable *pLinkTable, int Conditon(tLinkTableNode * pNode, void * args), void * args)
{
    ...
    tLinkTableNode * pNode = pLiAnkTable->pHead;
    while(pNode != NULL)
    {
        if(Conditon(pNode,args) == SUCCESS)
        {
            return pNode;
        }
        pNode = pNode->pNext;
    }
    return NULL;
}A
```

多态

- “面向对象”范式仅仅告诉开发者在需求语句中寻找“名词 ”，并将这些名词构造成程序中的对象。在这种范式中“封装”仅仅被定义为“数据隐藏”，“对象 ”也只是被定义为“包含数据及访问这些数据的东西”。其实面向对象的真正威力不是继承而是“行为封装”，而继承的最典型用法在多态机制的实现上。

- 多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。多态是实例化变量可以指向不同的实例对象，这样同一个实例化变量在不同的实例对象上下文环境中执行不同的代码表现出不同的行为状态，而通过实例化变量调用实例对象的方法的那一块代码却是完全相同的，这就顾名思义，同一段代码执行时却表现出不同的行为状态，因而叫多态。简单的说，可以理解为允许将不同的子类类型的对象动态赋值给父类类型的变量，通过父类的变量调用方法在执行时实际执行的可能是不同的子类对象方法，因而表现出不同的执行效果。

多态举例

```
class A
{
public:
    A(){}
    virtual void foo()
    {
        cout<<"This is A."<<endl;
    }
};

class B: public A
{
public:
    B(){}
    void foo()
    {
        cout<<"This is B."<<endl;
    }
};

class C: public A
{
public:
    C(){}
    void foo()
    {
        cout<<"This is C."<<endl;
    }
};

int main(int argc, char *argv[])
{
    A *a = new B();
    a->foo(); //34

    a = new C();
    a->foo();//37
```

显然34行和37行两句代码都是a->foo()，代码完全相同，执行效果却不同，这就是多态。

多态是设计模式中非常常用的关键机制。

闭包

- 闭包是变量作用域的一种特殊情形，一般用在将函数作为返回值时，该函数执行所需的上下文环境也作为返回的函数对象的一部分，这样该函数对象就是一个闭包。

- 更严谨的定义是，函数和对其周围状态（lexical environment，词法环境）的引用捆绑在一起构成闭包（closure）。也就是说，闭包可以让你从内部函数访问外部函数作用域。在JavaScript中，每当函数被创建，就会在函数生成时生成闭包。

```
function makeFunc() {
    var name = "Mozilla";
    function displayName() {
        alert(name);
    }
    return displayName;
}

var myFunc = makeFunc();
myFunc();
```

异步调用

- Promise对象可以将异步调用以同步调用的流程表达出来，避免了通过嵌套回调函数实现异步调用。

- ES6原生提供了Promise对象。所谓Promise对象，就是代表了未来某个将要发生的事件，通常是一个异步操作。Promise对象提供了一整套完整的接口，使得可以更加容易地控制异步调用。

- ES6的Promise对象是一个构造函数，用来生成Promise实例。下面是Promise对象的基本用法。

Promise对象的基本用法

- Promise对象实际上是对回调函数机制的封装，也就是通过then方法定义的函数与resolve/reject函数绑定，简化了回调函数传入的接口实现，在逻辑上也更加通顺，看起来像是个同步接口。

```
var promise = new Promise(function(resolve, reject) {
  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});

promise.then(function(value) { // resolve(value)
  // success
}, function(value) { // reject(error)
  // failure
});
```

匿名函数

- lamda函数是函数式编程中的高阶函数，在我们常见的命令式编程语言中常常以匿名函数的形式出现，比如无参数的代码块{ code }，或者箭头函数 { x => code }，如下使用Promise对象实现的计时器就用到了箭头函数。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

timeout(100).then(() => {
  console.log('done');
});
```

软件的内在特性

- 开发软件的根本任务是打造构成抽象软件的复杂概念结构，次要任务才是使用代码表达抽象的概念设计并映射成机器指令。因此软件的内在特性更主要体现在复杂的概念结构上，而软件的基本特点是前所未有的复杂度和易变性，为了降低复杂度我们在不同层面大量采用抽象方法建立软件概念模型；为了应对易变性我们努力保持软件设计和实现上的完整性和一致性。
  - 前所未有的复杂度
  - 抽象思维 vs. 逻辑思维
  - 唯一不变的就是变化本身
  - 难以达成的概念完整性和一致性

前所未有的复杂度

- 软件产品可能是人类创造的最复杂的事物，计算机本身已经是人类创造的非常复杂的产品了，而软件有过之而无不及，因为计算机中最复杂的CPU芯片，随着硬件设计软件化，芯片本身已经变成了软件形态。

- 软件系统中没有任何两个部分是相同的，如果有，那么就把它们合并成一个。在这方面软件和飞机、高铁、摩天大楼等其他硬件形态的大规模系统完全不同，硬件形态的产品往往具有大量重复的部分。硬件产品的复杂度是线性增长的，往往是很多相同部件的添加和维护；而软件系统的功能扩展和维护必须是不同软件部件的添加，因而软件系统的复杂度是非常陡峭的非线性增长。

抽象思维 vs. 逻辑思维

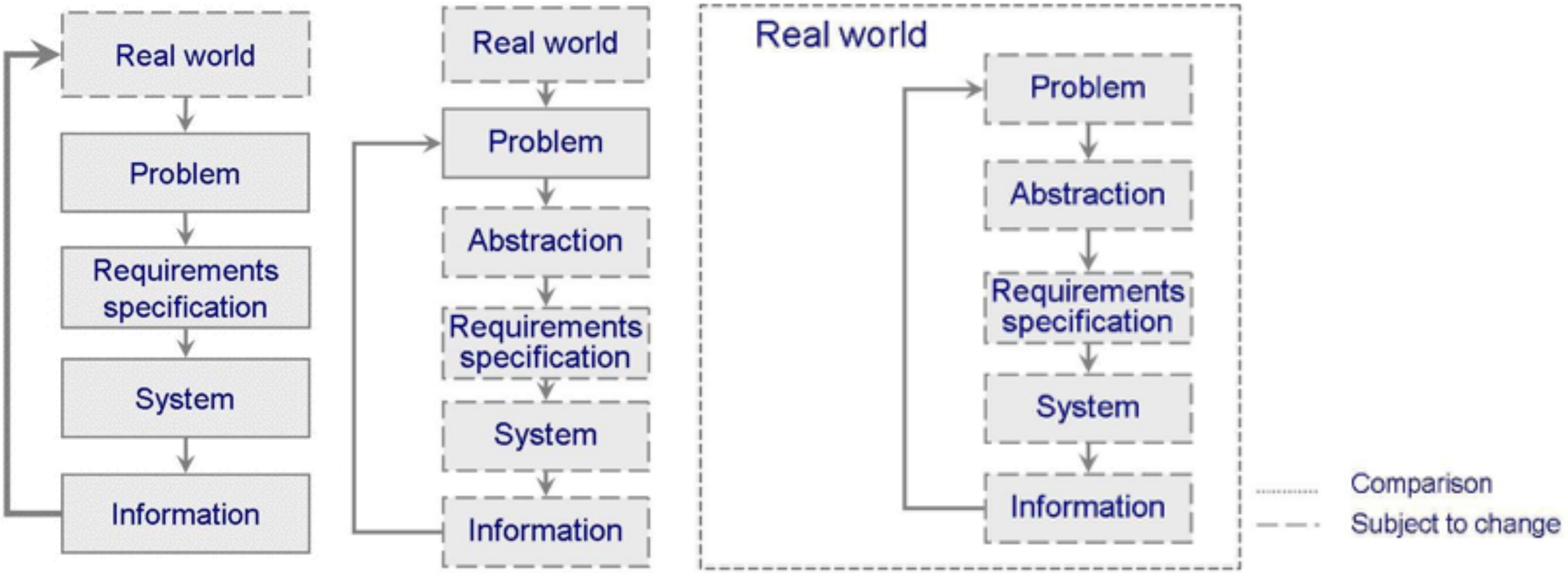
- 科学家为大自然中诸多复杂现象建立了简化的逻辑模型，我们能不能将这样的成功应用到软件领域呢？从软件危机发生依赖人们试图将抽象对象作为软件的基本元素来构建复杂软件的逻辑模型，尽管取得不小的进展和成效，但并没有根本性地解决问题，用Fred Brooks的话说——没有银弹（No Silver Bullet）。究其原因，给自然现象建立简化模型时忽略掉的复杂细节不是自然现象的根本属性，当这一套行之有效的方法应用到软件上，发现复杂度是软件的本质属性，因此也就无法取得根本性的突破。

- 不过在试图引入科研研究的逻辑思维的过程中，我们发现针对不可见的软件进行合理的抽象，在抽象的基础上努力建立逻辑模型的方法，对于应对软件的复杂度还是取得了一些效果。但是抽象思维能力不像逻辑思维能力一样，在古希腊时期就有了数论和几何学，以此为基础可以循序渐进的训练逻辑思维，而抽象思维更多取决于直觉和经验，目前还没有形成一套切实可行的训练方法。

唯一不变的就是变化本身

- Lehman将系统分成三种类型：S系统、P系统和E系统。E系统很好地解释了软件易变性的本质原因。我们依次看看这三种系统类型。
- S-system: formally defined, derivable from a specification
  - Matrix manipulation矩阵运算
- P-system: requirements based on approximate solution to a problem, but real-world remains stable
  - Chess program
- E-system: embedded in the real world and changes as the world does
  - Software to predict how economy functions (but economy is not completely understood)

S系统、P系统和E系统


Real world
Problem
Requirements specification
System
Information
Real world
Problem
Abstraction
Requirements specification
System
Information
Real world
Problem
Abstraction
Requirements specification
System
Information
Comparison
Subject to change

软件易变性

- 人类为了探索大自然的奥秘而进行的科研活动大多属于S系统或P系统，因为人类可以将自然现象抽象成纯粹的数学问题加以研究，即便不能抽象出纯粹的数学问题自然的法则也是稳定的。业务软件是嵌入到现实业务环境中的，现实业务环境不断变化，我们对业务的理解也在不断深入，我们对业务的抽象需要调整的更加合理，软件规格、软件系统及其管理的数据都在不断变化，在软件的世界里唯一不变的就是变化本身。

难以达成的概念完整性和一致性

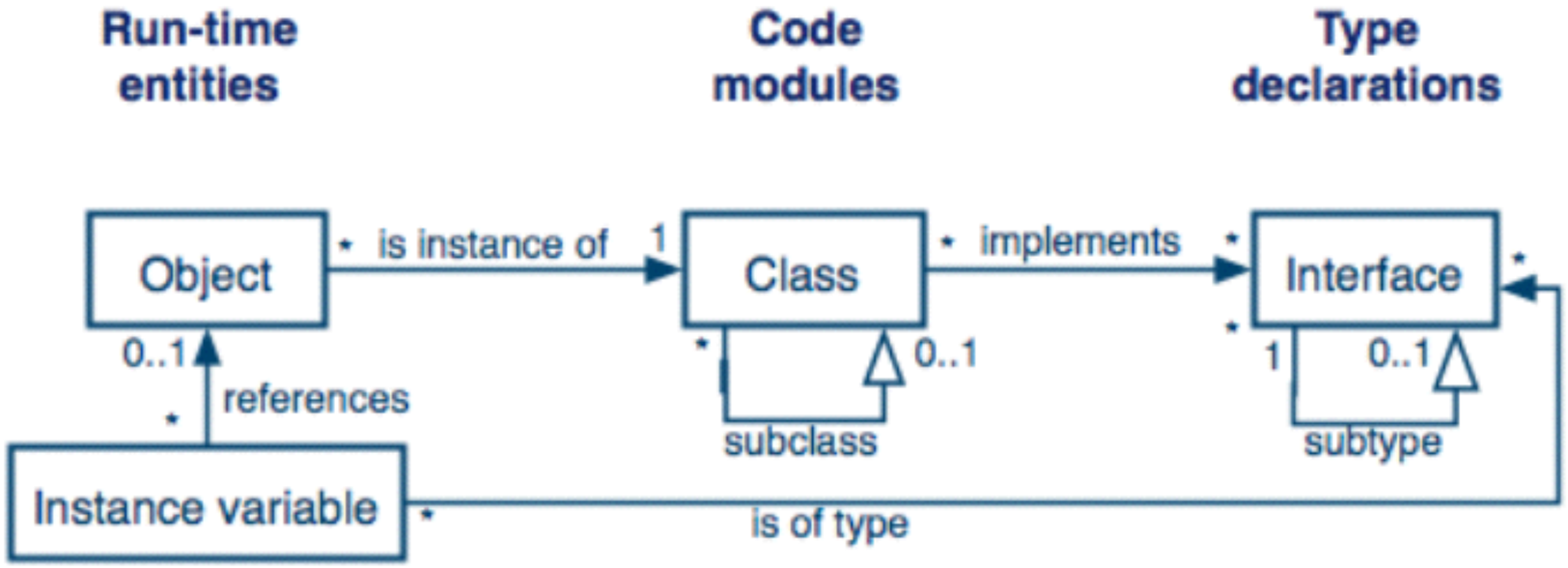
- 软件的复杂度和软件易变性这两个本质属性结合起来，使得我们在抽象的基础上建立逻辑模型的努力，永远达不到终点，难以达成软件概念的完整性和一致性。因为当我们付出极大的努力为复杂的软件建立起概念模型时，与此同时需求或环境已经发生了些许变化，或者发现建立概念模型的抽象有些许偏差，结果永远追不上那个完整的一致的软件概念模型。

二、软件设计模式初步

- 设计模式涉及的基本概念
- 什么是设计模式?
- 设计模式的分类
- 常用的设计模式
- 设计模式背后的设计原则

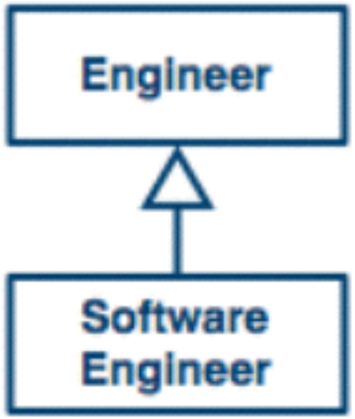
设计模式涉及的基本概念

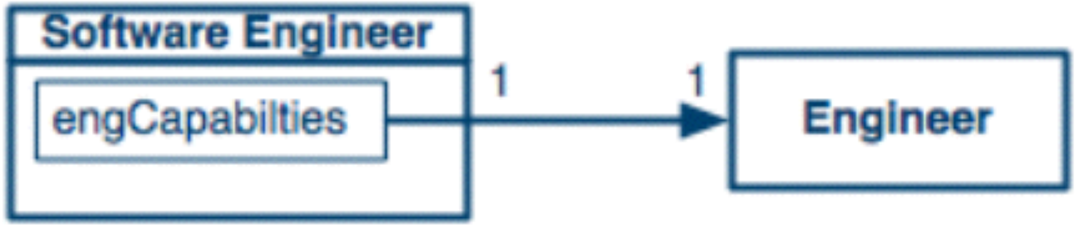
- 在面向对象分析中我们涉及到了类、对象、属性以及类与类之间的关系，在此基础上我们进一步理解面向对象设计和实现所涉及的基本术语之间的关系



继承（Inheritance）和对象组合（object composition）

- 继承的概念非常容易理解，于是被广为接受，但是面向对象的真正威力是封装，而非继承! 封装的威力集中体现在对象组合对继承的替代作用上。





继承（Inheritance）和对象组合（object composition）

- 继承可以重用代码，但是破坏了代码的封装特性，增加了父类与子类之间的代码模块耦合，因此我们需要避免使用继承，在大多数情况下，应该使用对象组合替代继承来实现相同的目标。

- 由于继承的用法被广泛接受，以致反过来造成了对封装的误解。“面向对象”范式仅仅告诉开发者在需求语句中寻找“名词 ”，并将这些名词构造成程序中的对象。在这种范式中“封装”仅仅被定义为“数据隐藏”，“对象 ”也只是被定义为“包含数据及访问这些数据的东西”。其实面向对象的真正威力不是继承而是“行为封装”，而继承的最典型用法在多态机制的实现上。

多态Polymorphism

- 多态（Polymorphism）按字面的意思就是“多种状态”。在面向对象语言中，接口的多种不同的实现方式即为多态。简单的说就是允许将子类类型的指针赋值给父类类型的指针。

```
int main(int argc, char *argv[])
{
    A *a = new B();
    a->foo();


    a = new C();
    a->foo();


    return 0;
}
```

```
class A
{
public:
    A(){}
    virtual void foo()
    {
        cout<<"This is A."<<endl;
    }
};

class B: public A
{
public:
    B(){}
    void foo()
    {
        cout<<"This is B."<<endl;
    }
};

class C: public A
{
public:
    C(){}
    void foo()
    {
        cout<<"This is C."<<endl;
    }
};
```

回调函数（callback）、闭包（closure）和lamda函数

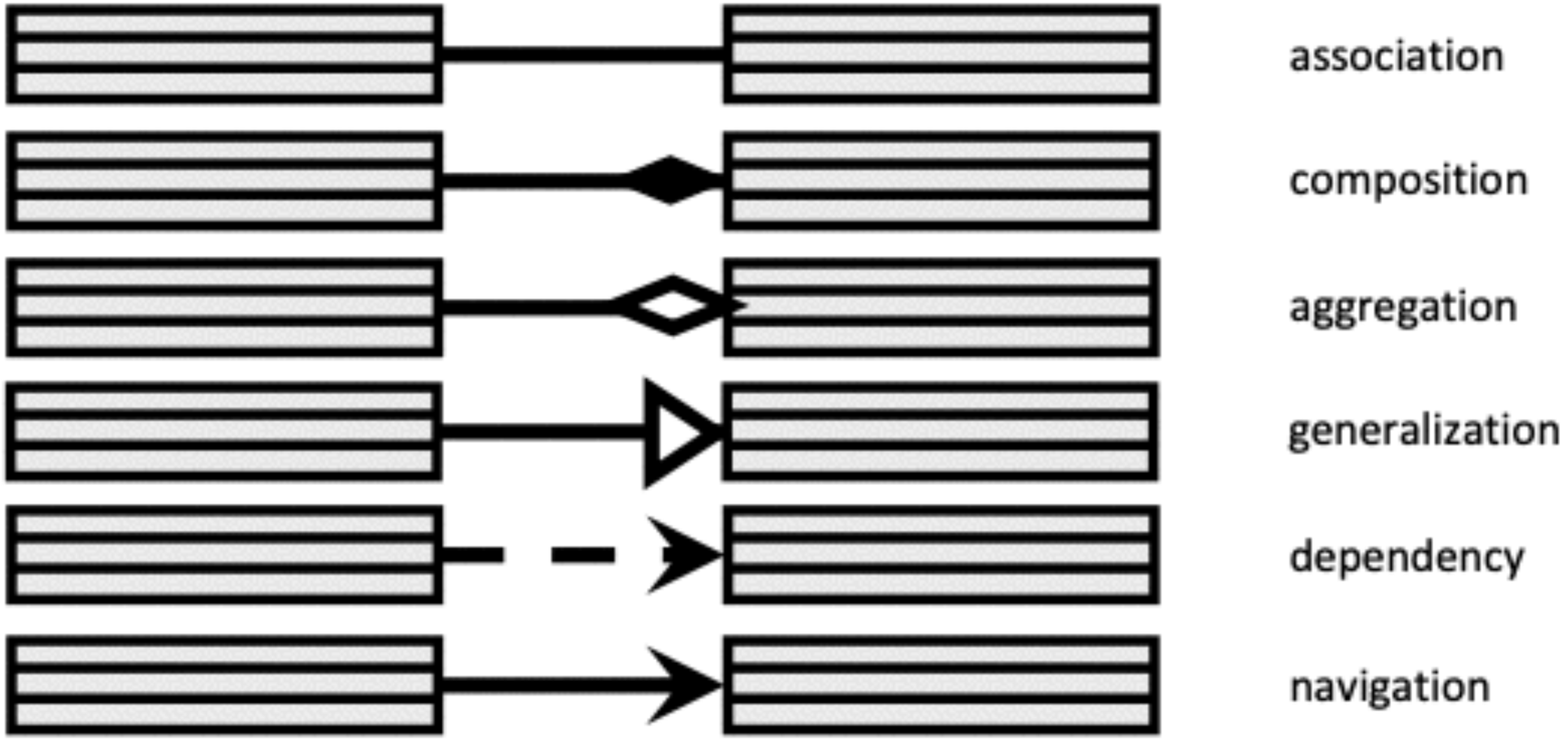
- 多态是设计模式中非常常用的关键机制，另外回调函数（callback）、闭包（closure）和lamda函数在设计模式中也经常用到。这几个概念在实际应用中常常交叉综合出现，在理解上带来很多困扰。

- • 回调函数是一个面向过程的概念，是代码执行过程的一种特殊流程；

- • 闭包是变量作用域的一种特殊情形，一般用在将函数作为返回值时，该函数执行所需的上下文环境也作为返回的函数对象的一部分，这样该函数对象就是一个闭包；

- • lamda函数是函数式编程中的高阶函数，在我们常见的命令式编程语言中常常以匿名函数的形式出现，比如无参数的代码块{ code }，有参数的匿名函数往往会使用箭头函数 { x => code }；

关系


association
composition
aggregation
generalization
dependency
navigation

什么是设计模式?

- “设计模式”这个术语最初并不是出现在软件设计中，而是被用于建筑领域的设计中。
- 1977 年，美国著名建筑大师、加利福尼亚大学伯克利分校环境结构中心主任克里斯托夫·亚历山大（Christopher Alexander）在他的著作《建筑模式语言：城镇、建筑、构造》（A Pattern Language: Towns Building Construction）中描述了一些常见的建筑设计问题，并提出了 253 种关于对城镇、邻里、住宅、花园和房间等进行设计的基本模式。
- 直到 1990 年，软件工程界才开始研讨设计模式的话题，后来召开了多次关于设计模式的研讨会。
- 1995 年，艾瑞克·伽马（ErichGamma）、理查德·海尔姆（Richard Helm）、拉尔夫·约翰森（Ralph Johnson）、约翰·威利斯迪斯（John Vlissides）等 4 位作者合作出版了《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software）一书，书中收录了23个设计模式，这是设计模式领域里程碑的事件，导致了软件设计模式的突破。

什么是设计模式?

- 设计模式的本质是面向对象设计原则的实际运用总结出的经验模型。对类的封装性、继承性和多态性以及类的关联关系和组合关系的充分理解的基础上才能准确理解设计模式。

设计模式的优点

- 正确使用设计模式具有以下优点。
- • 可以提高程序员的思维能力、编程能力和设计能力。
- • 使程序设计更加标准化、代码编制更加工程化，使软件开发效率大大提高，从而缩短软件的开发周期。
- • 使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强。

什么设计模式?

- 设计模式和面向对象的程序设计曾经承诺：让软件设计开发者的工作更加轻松！结果设计模式的抽象和复杂把很多开发者拒之门外。究其原因，人家承诺的是让工作更加轻松而不是让学习更加轻松，相反让学习更为困难。

设计模式要解决的问题

- 先来看看设计模式要解决的问题。功能分解是处理复杂问题的一种自然的方法，但是需求总是在发生变化，功能分解不能帮助我们为未来可能的变化做准备，也不能帮助我们的代码优雅地演化。结果你想在代码中做一些改变，但又不敢这么做，因为你知道对一个地方代码的修改可能在另一个地方造成破坏。

包容变化

- 与其抱怨总是变化的需求，我们不如改进我们的设计和开发方法，这样我们可以更有效地应付需求的变化。

- 用模块化来包容变化，使用模块化封装的方法，按照模块化追求的高内聚低耦合目标，借助于抽象思维对模块内部信息的隐藏并使用封装接口对外只暴露必要的可见信息，利用多态、闭包、lamda函数、回调函数等特殊的机制方法，将变化的部分和不变的部分进行适当隔离。这些都是设计模式的拿手好戏。此外在设计模式的基础上，可以更加方便地用增量、迭代和不断重构等开发过程来进一步应对总是变化的需求和软件本质上的模型不稳定特质。

设计模式是在某一情景下的问题解决方案

- A design pattern codifies design decisions and best practices for solving a particular design problem according to design principles
- Design patterns are not the same as software libraries; they are not packaged solutions that can be used as is. Rather, they are templates for a solution that must be modified and adapted for each particular use

设计模式一般由四个部分组成

- 该设计模式的名称；
- 该设计模式的目的，即该设计模式要解决什么样的问题；
- 该设计模式的解决方案；
- 该设计模式的解决方案有哪些约束和限制条件。

设计模式的分类

- 根据模式是主要用于类上还是主要用于对象上来划分的话，可分为类模式和对象模式两种类型的设计模式：

- • 类模式：用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。比如模板方法模式等属于类模式。

- • 对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。由于组合关系或聚合关系比继承关系耦合度低，因此多数设计模式都是对象模式。

设计模式的分类

- 根据设计模式可以完成的任务类型来划分的话，可以分为创建型模式、结构型模式和行为型模式 3 种类型的设计模式：

- • 创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。比如单例模式、原型模式、建造者模式等属于创建型模式。

- • 结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，比如代理模式、适配器模式、桥接模式、装饰模式、外观模式、享元模式、组合模式等属于结构型模式。结构型模式分为类结构型模式和对象结构型模式，前者采用继承机制来组织接口和类，后者采用组合或聚合来组合对象。由于组合关系或聚合关系比继承关系耦合度低，所以对象结构型模式比类结构型模式具有更大的灵活性。

- • 行为型模式：用于描述程序在运行时复杂的流程控制，即描述多个类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，它涉及算法与对象间职责的分配。比如模板方法模式、策略模式、命令模式、职责链模式、观察者模式等属于行为型模式。行为型模式分为类行为模式和对象行为模式，前者采用继承在类间分配行为，后者采用组合或聚合在对象间分配行为。由于组合关系或聚合关系比继承关系耦合度低，所以对象行为模式比类行为模式具有更大的灵活性。

常用的设计模式

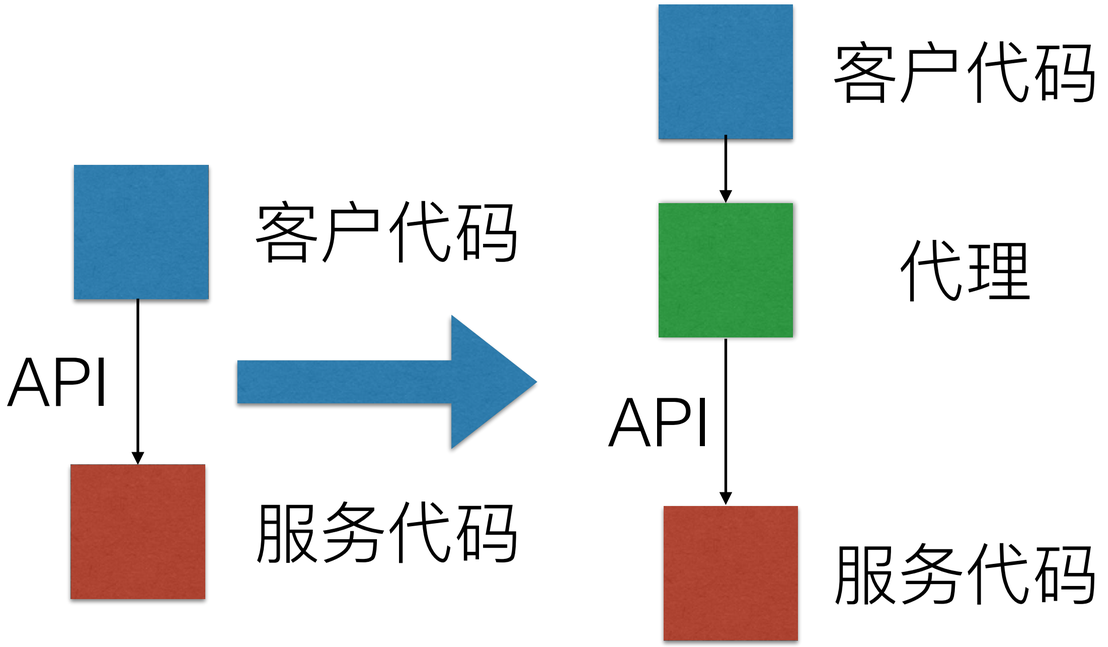
- • 单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，典型的应用如数据库实例。

- • 原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例，原型模式的应用场景非常多，几乎所有通过复制的方式创建新实例的场景都有原型模式。

- • 建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。主要应用于复杂对象中的各部分的建造顺序相对固定或者创建复杂对象的算法独立于各组成部分。

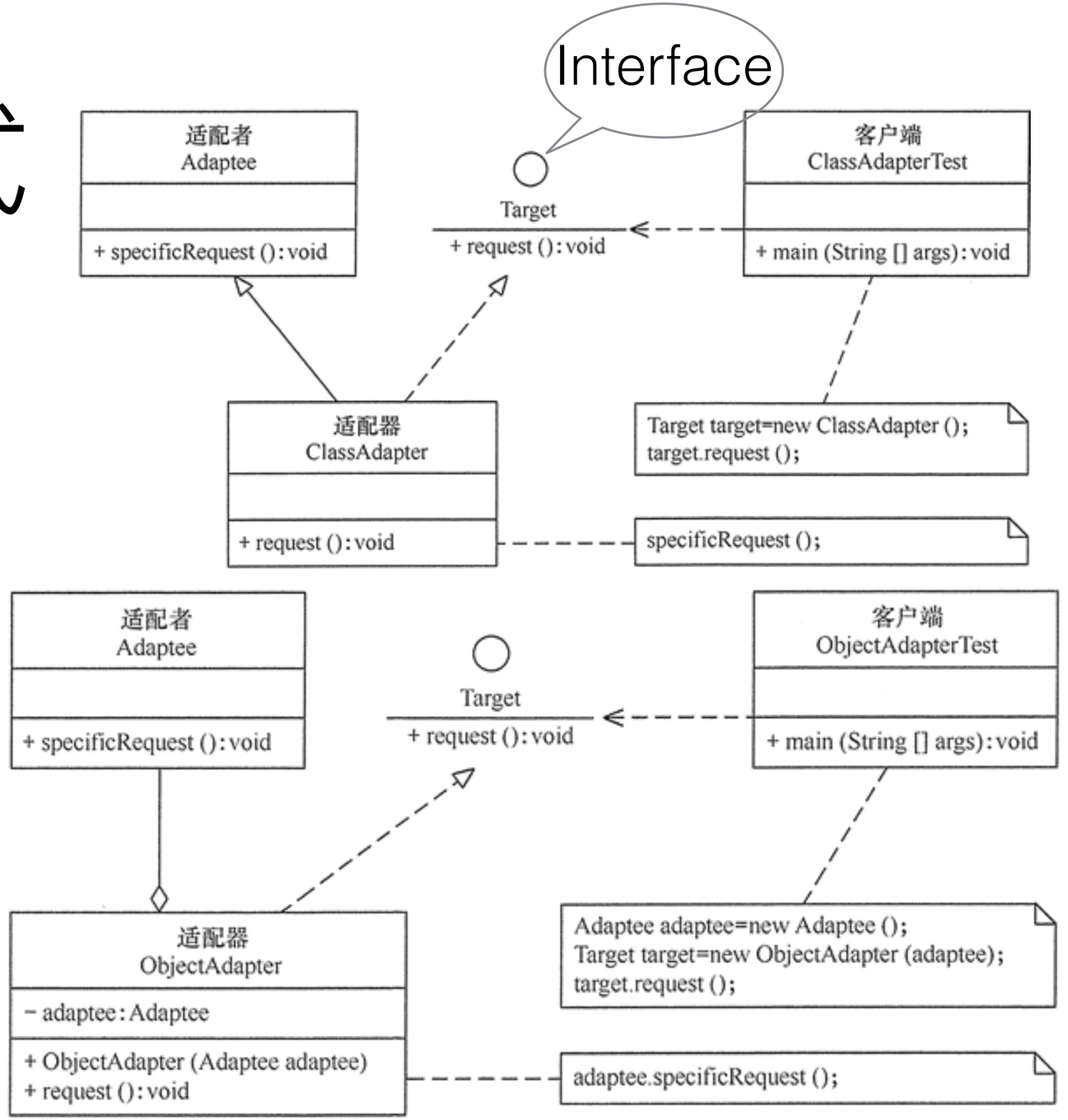
常用的设计模式

- • 代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。代理模式是不要和陌生人说话原则的体现，典型的应用如外部接口本地化将外部的输入和输出封装成本地接口，有效降低模块与外部的耦合度。



常用的设计模式

- 适配器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。继承和对象组合都可以实现适配器模式，但由于组合关系或聚合关系比继承关系耦合度低，所以对象组合方式的适配器模式比较常用。



常用的设计模式

- • 装饰（Decorator）模式：在不改变现有对象结构的情况下，动态地给对象增加一些职责，即增加其额外的功能。装饰模式实质上是用对象组合的方式扩展功能，因为比继承的方式扩展功能耦合度低。装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

- • 外观（Facade）模式：为复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。

- • 享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。比如线程池、固定分配存储空间的消息队列等往往都是该模式的应用场景。

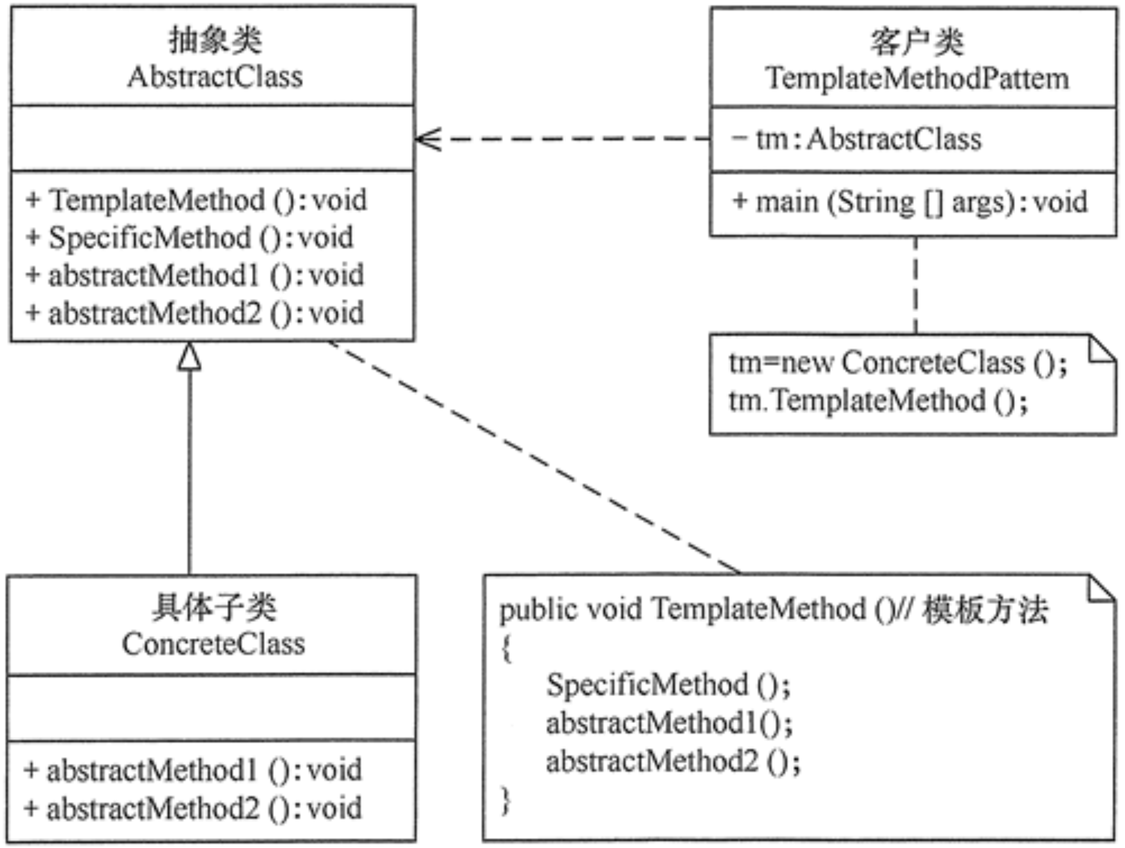
常用的设计模式

- • 策略（Strategy）模式：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。策略模式是多态和对象组合的综合应用。

- • 命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。

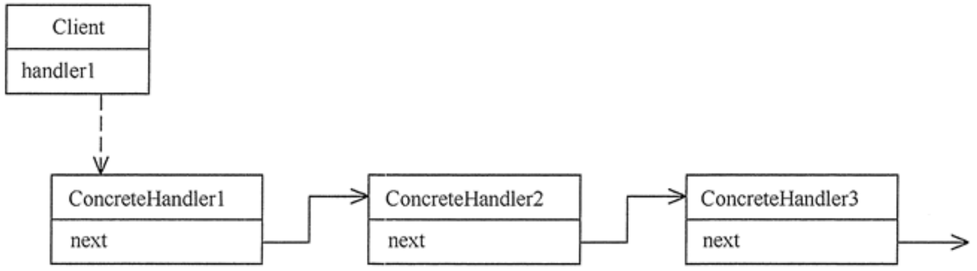
模板方法（TemplateMethod）

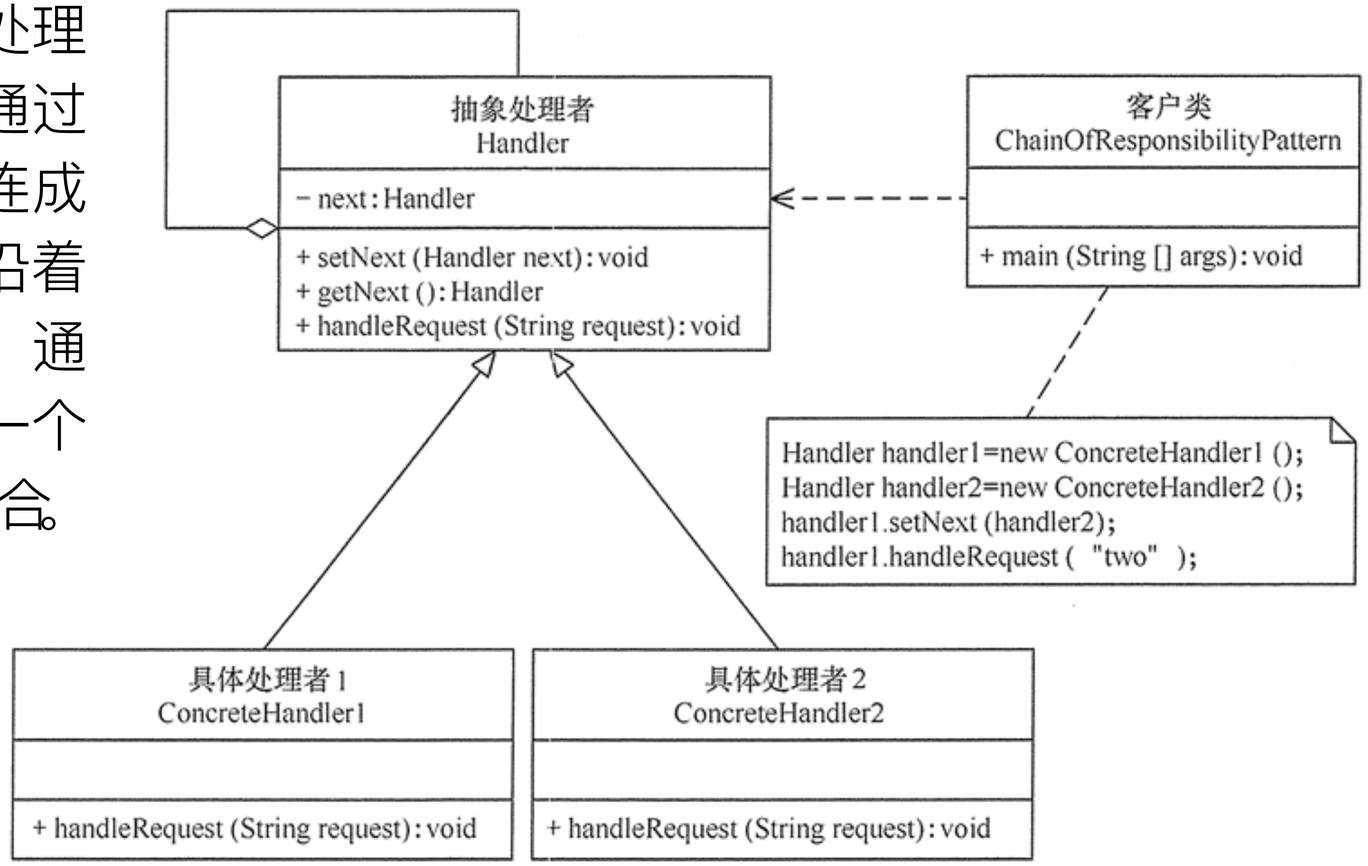
- 模板方法（TemplateMethod）模式：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。模版方法是继承和重载机制的应用，属于类模式。



职责链（Chain of Responsibility）

- 职责链（Chain of Responsibility）模式：为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。通过这种方式将多个请求处理者串联为一个链表，去除请求发送者与它们之间的耦合。





中介者（Mediator）

- 中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。在现实生活中，常常会出现好多对象之间存在复杂的交互关系，这种交互关系常常是“网状结构”，它要求每个对象都必须知道它需要交互的对象。如果把这种“网状结构”改为“星形结构”的话，将大大降低它们之间的“耦合性”，这时只要找一个“中介者”就可以了。在软件的开发过程中，这样的例子也很多，例如，在 MVC 框架中，控制器（C）就是模型（M）和视图（V）的中介者，采用“中介者模式”大大降低了对象之间的耦合性，提高系统的灵活性。

设计模式背后的设计原则

- 4.5.1.开闭原则（Open Closed Principle，OCP）
- 4.5.2.Liskov替换原则（Liskov Substitution Principle，LSP）
- 4.5.3.依赖倒置原则（Dependence Inversion Principle，DIP）
- 4.5.4.单一职责原则（Single Responsibility Principle，SRP）
- 4.5.5.迪米特法则（Law of Demeter，LoD）
- 4.5.6.合成复用原则（Composite Reuse Principle，CRP）

开闭原则（Open Closed Principle，OCP）

- 开闭原则（Open Closed Principle，OCP）由勃兰特·梅耶（Bertrand Meyer）提出，他在 1988 年的著作《面向对象软件构造》（Object Oriented Software Construction）中提出：软件应当对扩展开放，对修改关闭（Software entities should be open for extension，but closed for modification），这就是开闭原则的定义。

- 遵守开闭原则使软件拥有一定的适应性和灵活性的同时具备稳定性和延续性。统一过程以架构为中心增量且迭代的过程和开闭原则具有内在的一致性，它们都追求软件结构上的稳定性。当我们理解了软件结构模型本质上具有不稳定性的时候，一个小小的需求变更便很可能会触动软件结构的灵魂，瞬间让软件结构崩塌，即便通过破坏软件结构的内在逻辑模型打上丑陋的补丁程序，也会使得软件内在结构恶化加速软件的死亡。因此开闭原则在基本需求稳定且被充分理解的前提下才具有一定的价值。

- 在设计上包容软件结构本身的变化，以利于软件设计结构上的不断重构（Refactoring），才能适应软件结构本质上的不稳定性特点。

Liskov替换原则（Liskov Substitution Principle，LSP）

- Liskov替换原则（Liskov Substitution Principle，LSP）由麻省理工学院计算机科学实验室的Liskov女士在 1987 年发表的一篇文章Data Abstraction and Hierarchy里提出来的，她提出：继承必须确保超类所拥有的性质在子类中仍然成立（Inheritance should ensure that any property proved about supertype objects also holds for subtype objects）。

- Liskov替换原则主要阐述了继承用法的原则，也就是什么时候应该使用继承，什么时候不应该使用继承，以及其中蕴含的原理。通俗来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。也就是说儿子和父母要在DNA基因上一脉相承，尽管程序员是自己的代码的上帝，但也不能胡来，要做一个遵守自然规律的上帝。

- Liskov替换原则告诉我们，子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。

- Liskov替换原则如今看来其价值也大大折扣，因为为了降低耦合度我们往往使用对象组合来替代继承关系。反而是Liskov替换原则不推荐的多态成为诸多设计模式的基础。

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

- 依赖倒置原则（Dependence Inversion Principle，DIP）是 Object Mentor 公司总裁罗伯特·马丁（Robert C.Martin）于 1996 年在 C++ Report 上发表的文章。

- 依赖倒置原则的原始定义为：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象（High level modules shouldnot depend upon low level modules.Both should depend upon abstractions.Abstractions should not depend upon details. Details should depend upon abstractions）。其核心思想是：要面向接口编程，不要面向实现编程。

- 由于在软件设计中，细节具有多变性，而抽象层则相对稳定，因此以抽象为基础搭建起来的架构要比以细节为基础搭建起来的架构要稳定得多。这里的抽象指的是接口或者抽象类，而细节是指具体的实现类。

- 依赖倒置原则在模块化设计中降低模块之间的耦合度和加强模块的抽象封装提高模块的内聚度上具有普遍的指导意义，但是“依赖倒置”这个名称体现的是抽象层次结构上的依赖倒置，并不能直观展现它在模块化设计中的重要价值，我觉得应该给它起个更好的名字，您可以试试重新给它起个名字。

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

- 单一职责原则（Single Responsibility Principle，SRP）又称单一功能原则，由罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分（There should never be more than one reason for a class to change）。

- 单一职责原则的核心就是控制类的粒度大小、提高其内聚度。如果遵循单一职责原则可以降低类的复杂度，因为一个类只负责一项职责，其逻辑肯定要比负责多项职责简单得多；同时可以提高类的内聚度，符合模块化设计的高内聚低耦合的设计原则。

- 单一职责原则是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，再封装到不同的类或模块中。而发现类的多重职责需要设计人员具有较强的抽象分析设计能力和相关重构经验。

迪米特法则（Law of Demeter，LoD）

- 迪米特法则（Law of Demeter，LoD）又叫作最少知识原则（Least Knowledge Principle，LKP)，产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被 UML 创始者之一的布奇（Booch）普及，后来又因为在经典著作《程序员修炼之道》（The Pragmatic Programmer）提及而广为人知。

- 迪米特法则的定义是：只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

合成复用原则（Composite Reuse Principle，CRP）

- 合成复用原则（Composite Reuse Principle，CRP）又叫组合/聚合复用原则（Composition/Aggregate Reuse Principle，CARP）。它要求在软件复用时，要尽量先使用组合或者聚合关系来实现，其次才考虑使用继承关系来实现。如果要使用继承关系，则必须严格遵循Liskov替换原则。

- 通常类的复用分为继承复用和对象组合复用两种。

继承复用的缺点

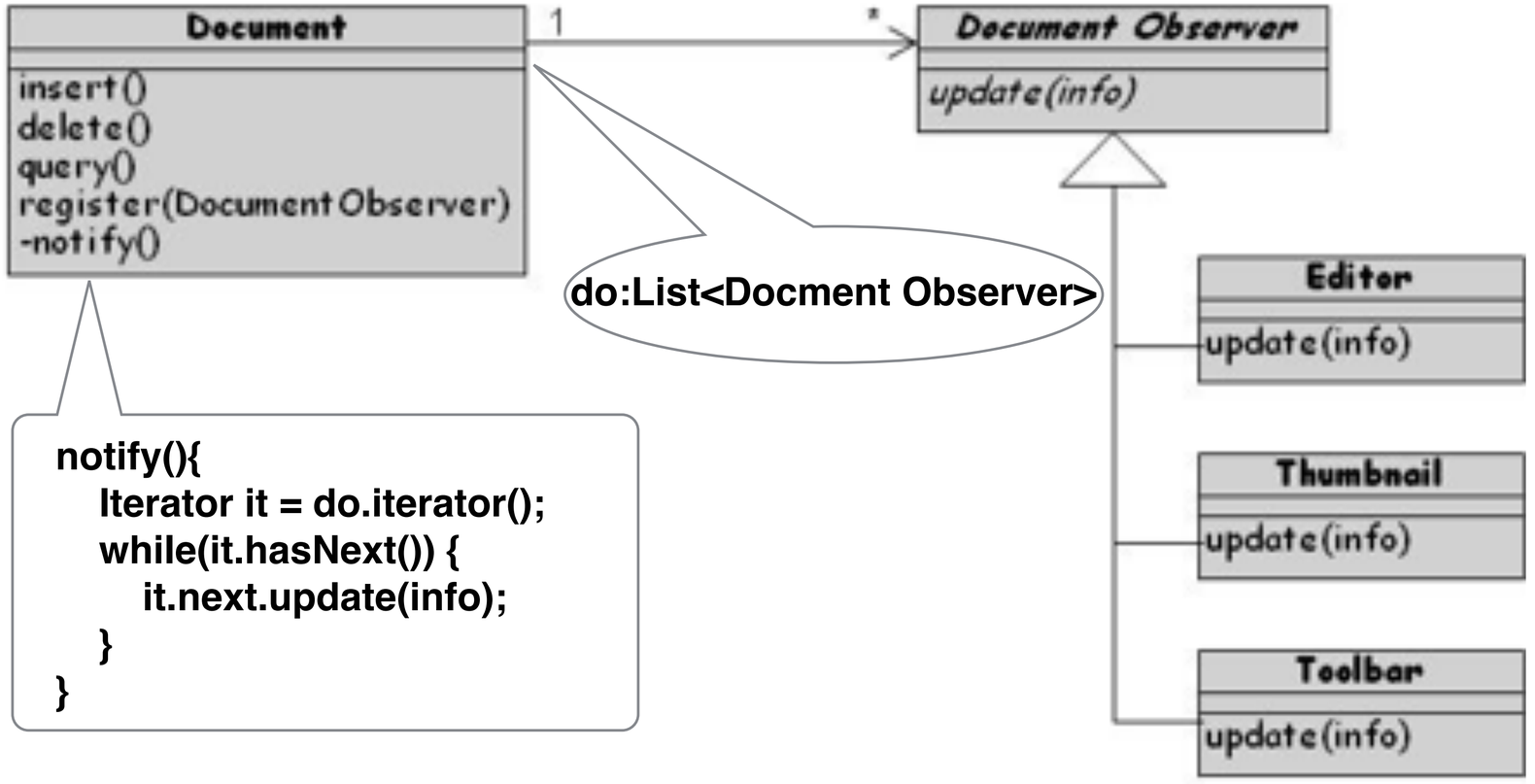
- 继承复用虽然有简单和易实现的优点，但它也存在以下缺点。
- ● 继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。
- ● 子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。
- ● 继承复用限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。

合成复用的优点

- 采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点。
- • 组合或聚合复用维持了类的封装性。因为属性对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。
- • 新旧类之间的耦合度低。这种复用所需的依赖较少，新对象存取属性对象的唯一方法是通过属性对象的接口。
- • 复用的灵活性高。这种复用可以在运行时动态进行，新对象可以动态地引用与属性对象类型相同的对象。

观察者（Observer）

- 观察者（Observer）模式：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为这样所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式。



高温预警系统

- 我们以高温预警系统为例来看看观察者模式的范例代码。高温预警系统是气象部门根据气象卫星获得相关的天气温度信息，当温度超过某一阈值时，向各个单位和个人发出高温警报通知，以及时做好高温防护措施。高温预警过程分析总结为如下几点：

- • 想要得到高温警报通知的对象订阅高温预警服务。

- • 高温预警系统需要知道哪些对象是需要通知的。这需要预警系统维护一个订阅对象列表。

- • 高温预警系统在温度达到阈值的时候，通知所有订阅服务的对象。如何通知呢？这需要订阅服务的对象具备接收高温警报通知的功能。

```java
package Observer;
import java.util.Random;

public class Test {
    public static void main(String[] args) {
        Subject s = new ForcastSystem();
        Government g = new Government(s);
        Company c = new Company(s);
        Person p = new Person(s);

        Random temperature = new Random();
        int i = 0;
        while (true) {
            s.setTemperature(temperature.nextInt(45));
        }
    }
}
```

```java
package Observer;
import java.util.Iterator;
import java.util.Vector;

public interface Subject {
    public boolean add(Observer observer);
    public boolean remove(Observer observer);
    public void notifyAllObserver();
    public String report();
}

public class ForcastSystem implements Subject {
    private float temperature;
    private String warningLevel;
    private final Vector<Observer> vector;

    public ForcastSystem() {
        vector = new Vector<Observer>();
    }

    public boolean add(Observer observer) {
        if (observer != null && !vector.contains(observer)) {
            return vector.add(observer);
        }
        return false;
    }
    public boolean remove(Observer observer) {
        return vector.remove(observer);
    }
    public void notifyAllObserver() {
        Iterator<Observer> iterator = vector.iterator();
        while (iterator.hasNext()) {
            (iterator.next()).update(this);
        }
    }

    private void invoke() {
        if (this.temperature >= 35) {
            if (this.temperature >= 35 && this.temperature < 37) {
                this.warningLevel = "Yellow";
            } else if (this.temperature >= 37 && this.temperature < 40) {
                this.warningLevel = "Orange";
            } else if (this.temperature >= 40) {
                this.warningLevel = "Red";
            }
            this.notifyAllObserver();
        }
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        this.invoke();
    }
    public String report() {
        return this.warningLevel + this.temperature;
    }
}
```

- Observer和Subject是抽象接口；
- Government、Company、Person及其子子孙孙只与Observer和Subject有关系，可以独立演化；
- ForcastSystem及同类系统只与Observer和Subject有关系；
- 具体的观察者和被观察者之间没有直接耦合关系，各自独立演化。

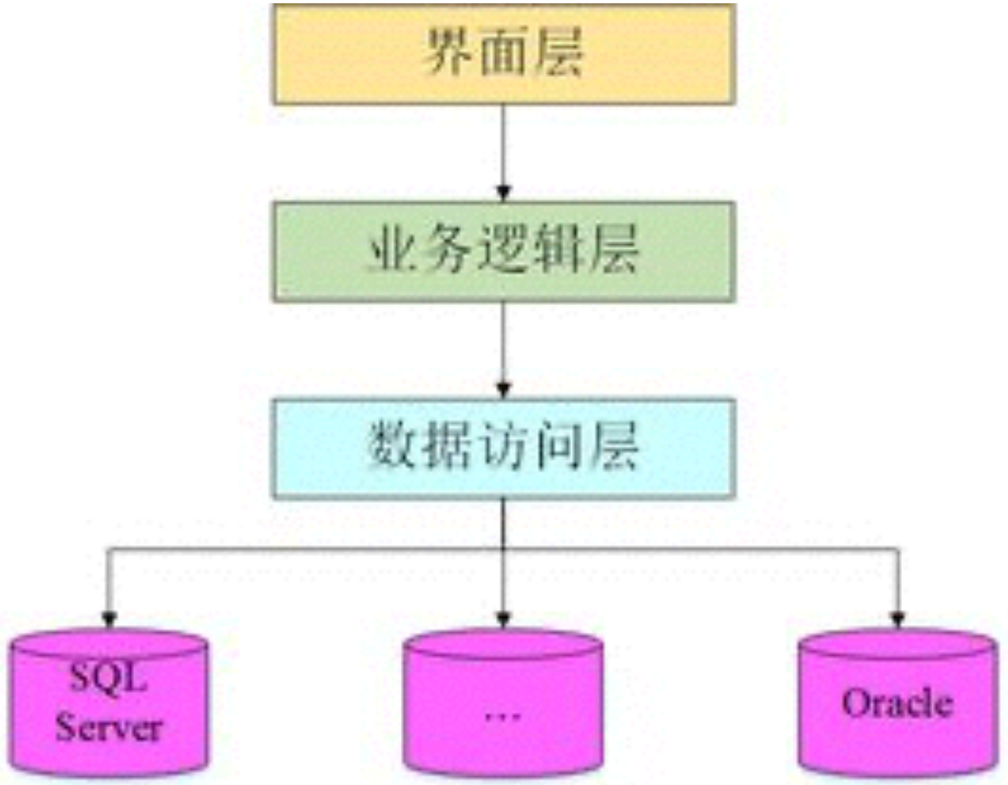
```
public interface Observer {
    public void update(Subject subject);
}

public class Government implements Observer {
    public Government(Subject subject) {
        subject.add(this);
    }
    public void update(Subject subject) {
        System.out.println(subject.report());
    }
}

public class Company implements Observer {
    public Company(Subject subject) {
        subject.add(this);
    }

    public void update(Subject subject) {
        System.out.println(subject.report());
    }
}

public class Person implements Observer {
    public Person(Subject subject) {
        subject.add(this);
    }

    public void update(Subject subject) {
        System.out.println(subject.report());
    }
```

三、常见的软件架构举例

- 三层架构
- MVC架构
- MVVM架构
- 微服务架构

三层架构

- 层次化架构是利用面向接口编程的原则将层次化的结构型设计模式作为软件的主体结构。比如三层架构是层次化架构中比较典型的代表，如下图所示我们以层次化架构为例来介绍。



MVC架构

- MVC即为Model-View-Controller（模型-视图-控制器），MVC是一种设计模式，以MVC设计模式为主体结构实现的基础代码框架一般称为MVC框架，如果MVC设计模式决定了整个软件的架构，不管是直接实现了MVC模式还是以某一种MVC框架为基础，只要软件的整体结构主要表现为MVC模式，我们就称为该软件的架构为MVC架构。

- MVC中M、V和C所代表的含义如下：

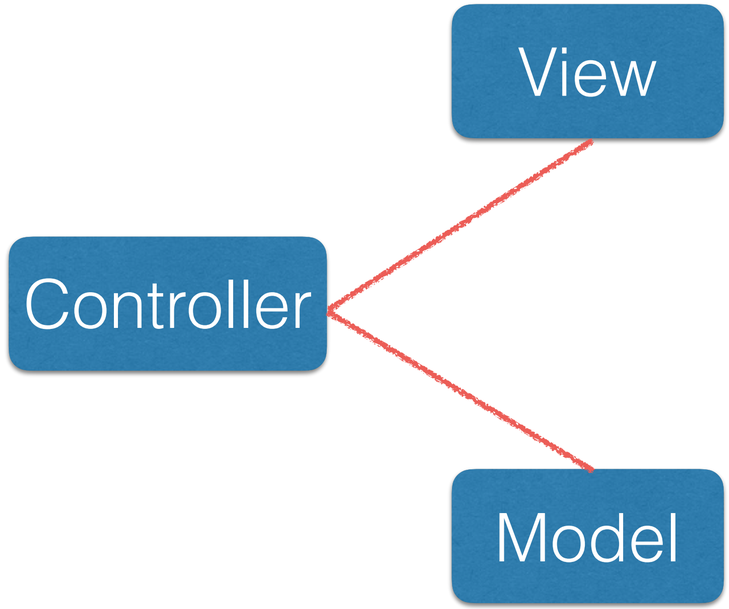
  - Model（模型）代表一个存取数据的对象及其数据模型。

  - View（视图）代表模型包含的数据的表达方式，一般表达为可视化的界面接口。

  - Controller（控制器）作用于模型和视图上，控制数据流向模型对象，并在数据变化时更新视图。控制器可以使视图与模型分离开解耦合。

MVC架构 vs. 三层架构

- 模型和视图有着业务层面的业务数据紧密耦合关系，控制器的核心工作就是业务逻辑处理，显然MVC架构和三层架构有着某种对应关系，但又不是层次架构的抽象接口依赖关系，因此为了体现它们的区别和联系，我们在MVC的结构示意图中将模型和视图上下垂直对齐表示它们内在的业务层次及业务数据的对应关系，而将控制器放在左侧表示控制器处于优先重要位置，放在模型和视图的中间位置是为了与三层架构对应与业务逻辑层处于相似的层次。



MVC模式/中介者模式范例

- 为了理解MVC的工作机制是如何区别于三层架构，我们先来看看MVC模式的一个简要范例。

- 我们将创建一个作为模型的 Student 对象。StudentView 是一个把学生详细信息输出到控制台的视图类，StudentController 是负责存储数据到 Student 对象中的控制器类，并相应地更新视图 StudentView。

```java
public class Student {
   private String rollNo;
   private String name;
   public String getRollNo() {
      return rollNo;
   }
   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Model（模型）Student类

View（视图）StudentView类

```java
public class StudentView {
   public void printStudentDetails(String studentName, String studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }
}
```

Controller（控制器）StudentController类

```java
public class StudentController {
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view){
      this.model = model;
      this.view = view;
   }

   public void setStudentName(String name){
      model.setName(name);
   }

   public String getStudentName(){
      return model.getName();
   }

   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }

   public String getStudentRollNo(){
      return model.getRollNo();
   }

   public void updateView(){
      view.printStudentDetails(model.getName(), model.getRollNo());
   }
}
```

MVCPatternDemo 来演示 MVC 模式的用法

```java
public class MVCPatternDemo {
   public static void main(String[] args) {
      //从数据库获取学生记录
      Student model  = retrieveStudentFromDatabase();
      //创建一个视图：把学生详细信息输出到控制台
      StudentView view = new StudentView();
      StudentController controller = new StudentController(model,
view);

      controller.updateView();
      //更新模型数据
      controller.setStudentName("John");
      controller.updateView();
   }

   private static Student retrieveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
}
```

MVC模式/中介者模式范例

- 如果从类的角度看，StudentController类应该在上面，它下面依赖Student类和StudentView类，这也是为什么我们看到的大多数MVC模式的示意图大多习惯于将Controller（控制器）放在上面中间位置的原因。从面向对象的设计来看，MVC是对象组合的综合应用；从设计模式的角度看，Controller（控制器）是Model（模型）和View（视图）之间的中介者（Mediator），是典型的中介者模式。

MVC架构

- MVC模式通常用开发具有人机交互界面的软件，这类软件的最大特点就是用户界面容易随着需求变更而发生改变，例如，当你要扩展一个应用程序的功能时，通常需要修改菜单和添加页面来反映这种变化。如果用户界面和核心功能逻辑紧密耦合在一起，要扩展功能通常是非常困难的，因为任何改动很容易在其他功能上产生错误。

- 为了包容需求上的变化而导致的用户界面的修改不会影响软件的核心功能代码，可以采用将模型（Model）、视图（View）和控制器（Controller）相分离的思想。采用MVC设计模式的话往往决定了整个软件的主体结构，因此我们称该软件为MVC架构。

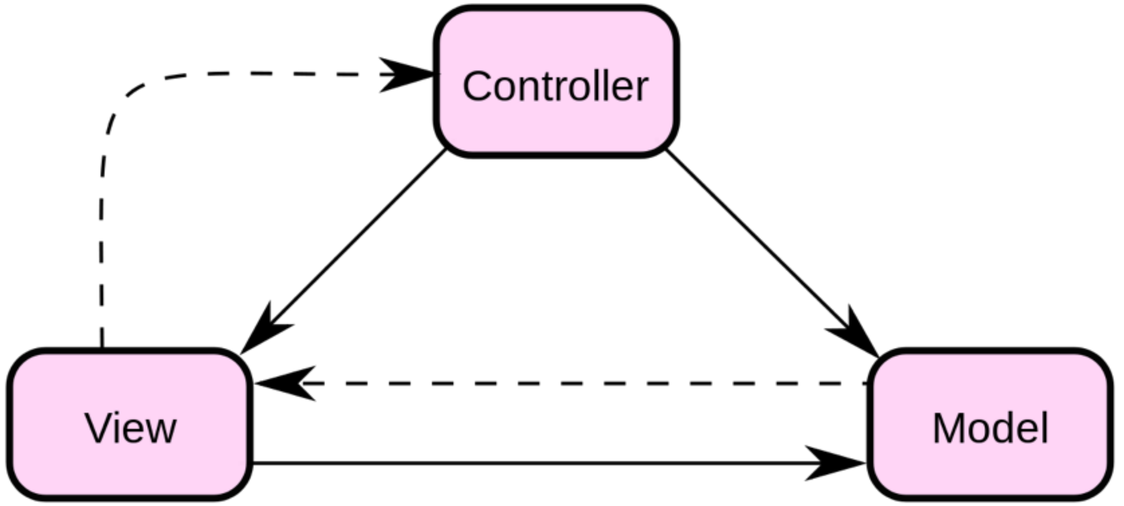
MVC架构中的模型、视图与控制器

- 在MVC架构下，模型用来封装核心数据和功能，它独立于特定的输出表示和输入行为，是执行某些任务的代码，至于这些任务以什么形式显示给用户，并不是模型所关注的问题。模型只有纯粹的功能性接口，也就是一系列的公开方法，这些方法有的是取值方法，让系统其它部分可以得到模型的内部状态，有的则是写入更新数据的方法，允许系统的其它部分修改模型的内部状态。

- 在MVC架构下，视图用来向用户显示信息，它获得来自模型的数据，决定模型以什么样的方式展示给用户。同一个模型可以对应于多个视图，这样对于视图而言，模型就是可重用的代码。一般来说，模型内部必须保留所有对应视图的相关信息，以便在模型的状态发生改变时，可以通知所有的视图进行更新。

- 在MVC架构下，控制器是和视图联合使用的，它捕捉鼠标移动、鼠标点击和键盘输入等事件，将其转化成服务请求，然后再传给模型或者视图。软件的用户是通过控制器来与系统交互的，他通过控制器来操纵模型，从而向模型传递数据，改变模型的状态，并最后导致视图的更新。

应用MVC架构的软件其基本的实现过程



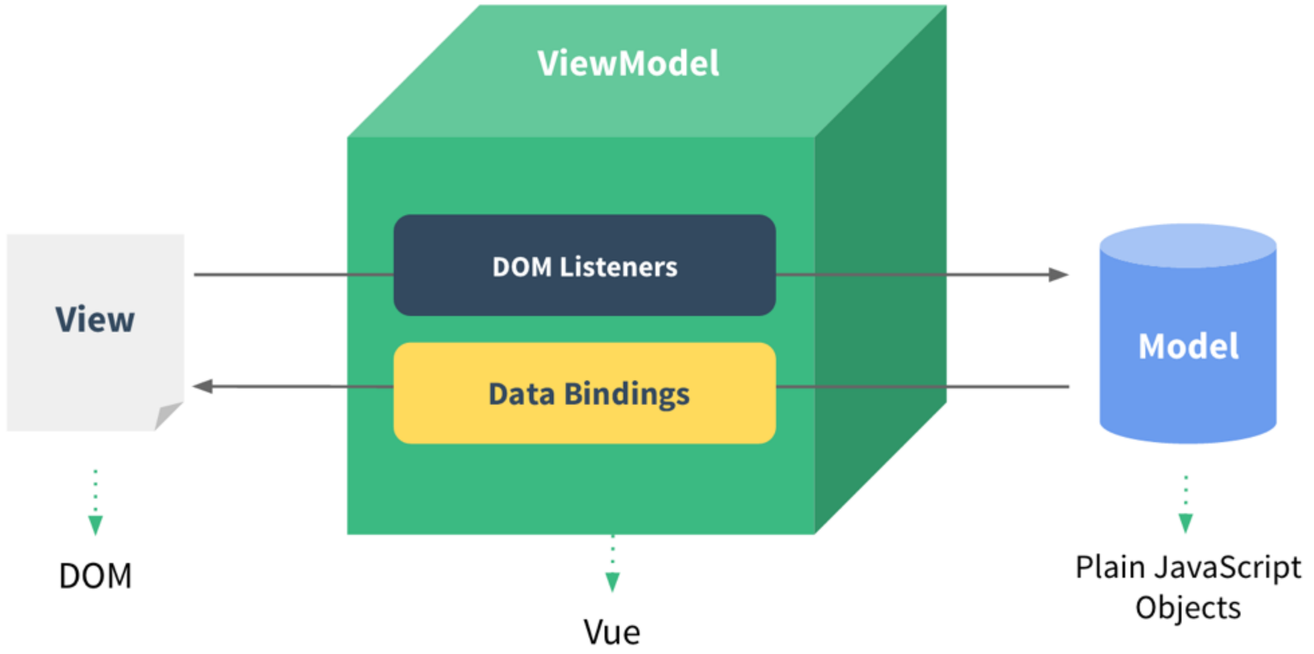
- 控制器创建模型；
- 控制器创建一个或多个视图，并将它们与模型相关联；
- 控制器负责改变模型的状态；
- 当模型的状态发生改变时，模型会通知与之相关的视图进行更新。
- 可以看到这与抽象简化的MVC设计模式已经有了明显的区别，变得更为复杂，但这与实际软件结构相比还是极其简化的模型，实际情况可能会有更多合理的和不合理的复杂联系，要保持软件结构在概念上的完整性极为困难。

什么是MVVM?

- MVVM即 Model-View-ViewModel，最早由微软提出来，借鉴了桌面应用程序的MVC模式的思想，是一种针对WPF、Silverlight、Windows Phone的设计模式，目前广泛应用于复杂的Javacript前端项目中。
- 随着前端页面越来越复杂，用户对于交互性要求也越来越高，jQuery是远远不够的，于是MVVM被引入到Javascript前端开发中。

Vue.js框架的MVVM架构

- 在前端页面中，把Model用纯JavaScript对象表示，View负责显示，两者做到了最大限度的分离。把Model和View关联起来的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。以比较流行的Vue.js框架为例，MVVM架构示意图如下：



MVVM的优点

- MVVM模式和MVC模式一样，主要目的是分离视图（View）和模型（Model），有几大优点：
  - 低耦合。视图（View）可以独立于Model变化和修改，一个ViewModel可以绑定到不同的"View"上，当View变化的时候Model可以不变，当Model变化的时候View也可以不变。
  - 可重用性。你可以把一些视图逻辑放在一个ViewModel里面，让很多View重用这段视图逻辑。
  - 独立开发。开发人员可以专注于业务逻辑和数据的开发（ViewModel），设计人员可以专注于页面设计。
  - 可测试。界面素来是比较难于测试的，测试可以针对ViewModel来写。

Vue.js的基本用法

- 为了展现MVVM的威力，我们先用Vue.js写一个简单的范例，从用户开发者的角度来体会MVVM的良好特性。

- 我们现在的目标是尽快用起来，所以最简单的方法是直接在HTML代码中像引用jQuery一样引用Vue.js。

```
<!-- 引用网上的Vue.js -->
<script src="https://unpkg.com/vue"></script>
<!-- 或者下载下来本地引用Vue.js -->
<script src="/static/js/vue.js"></script>
```

Vue.js的基本用法

- 我们的Model是一个JavaScript对象，
- 负责显示的是DOM节点可以用{{ message }}来引用Model的属性，也就是View了。
- 其中v-on:click="reverseMessage"用来跟踪DOM的点击事件调用reverseMessage方法。

```
{
    message: 'Hello Vue!'
}
```

```
methods: {
  reverseMessage: function () {
    this.message = this.message.spl
  }
}
```

```
<div id="app">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

```
<!DOCTYPE html>
<html>
<head>
  <title>My first Vue app</title>
  <script src="https://unpkg.com/vue"></script>
</head>
<body>
  <div id="app">
    <p>{{ message }}</p>
    <button v-on:click="reverseMessage">Reverse Message</button>
  </div>

  <script>
    var app = new Vue({
      el: '#app',
      data: {
        message: 'Hello Vue.js!'
      },
      methods: {
        reverseMessage: function () {
          this.message = this.message.split('').reverse().join('')
        }
      }
    })
  </script>
</body>
</html>
```

Vue.js的基本用法

- 我们在创建Vue对象的时候将View的id="app"与Model（JavaScript对象定义的data）绑定起来，这样'Hello Vue!'就会自动更新到View DOM元素中。View DOM元素button上的事件click绑定Vue对象的方法reverseMessage，这样点击button按钮就能触发reverseMessage，reverseMessage方法只是修改了Model中JavaScript对象定义的message，而页面却能神奇地自动更新message。就是模型数据绑定和DOM事件监听。如下完整代码来自https://vuejs.org/v2/guide/，该页面上也有如下代码的运行演示。

Vue.js背后MVVM模型的秘密

- Vue.js是一个前端构建数据驱动的Web界面的库，主要的特色是响应式的数据绑定，区别于以往的命令式用法。也就是在this.message = this.message.split('').reverse().join('')的过程中，拦截'='的过程，从而实现模型和视图自动同步更新的功能。而不需要显式地使用命令更新视图。Vue.js如何做到这一点的呢?

Object.defineProperty

- 首先把一个普通对象作为参数创建Vue对象时，Vue.js将遍历data的属性，用Object.defineProperty将要观察的对象“=”操作转化为getter/setter，以便拦截对象赋值与取值操作，称之为Observer；

```
//遍历data用Object.defineProperty 将要观察的对象“=”操作转化为getter/setter
Observer.prototype.transform = function(data){
    for(var key in data){
        var value = data[key];
        Object.defineProperty(data, key, {
            enumerable:true,
            configurable:true,
            get:function(){
                return value;
            },
            set:function(newVal){
                if(newVal == value){
                    return;
                }
                //遍历newVal
                this.transform(newVal);
                data[key] = newVal;//赋值还会调用set方法死循环了
            }
        });

        //递归处理
        this.transform(value);
    }
```

```
//客户端把一个普通对象作为参数创建Vue对象时
    var app = new Vue({
      el: '#app',
      data: {
        message: 'Hello Vue.js!'
      },
      ...
```

编译视图模板的主要工作

- 将DOM解析，提取其中的事件指令与占位符/表达式，并赋与不同的操作创建Watcher在模型中监听视图中出现的占位符/表达式，以及根据事件指令绑定监听事件和method，这是编译视图模板的主要工作，我们称之为Compiler；

```
//DOM中的指令与占位符
...
    <p>{{ message }}</p>
    <button v-on:click="reverseMessage">Reverse Message</button>
...
//创建Watcher在模型中监听视图中出现的占位符/表达式的每一个成员
var watcher = new Watcher("message"):
//绑定监听事件和method
node.addEventListener("click", "reverseMessage"):
```

```
//观察者模式中的被观察者的核心部分
var Dep = function(){
    this.subs = {};
};
Dep.prototype.addSub = function(target){
    if(!this.subs[target.uid]) {
        //防止重复添加
        this.subs[target.uid] = target;
    }
};
Dep.prototype.notify = function(newVal){
    for(var uid in this.subs){
        this.subs[uid].update(newVal);
    }
};
Dep.target = null;
```

观察者模式

- 将Compiler的解析结果，与Observer所观察的对象连接起来建立关系，在Observer观察到对象数据变化时，接收通知，同时更新DOM，称之为Watcher；

- 如何将Watcher与被观察者的核心部分联系起来的呢?

```
//创建Watcher，观察者模式中的观察者
var Watcher = function(exp, vm, cb){
    this.exp = exp; // 占位符/表达式的一个成员
    this.cb = cb; //更新视图的回调函数
    this.vm = vm; //ViewModel
    this.value = null;
    this.getter = parseExpression(exp).get;
    this.update();
};

Watcher.prototype = {
    get : function(){
        Dep.target = this;
        var value = this.getter?this.getter(this.vm):'';
        Dep.target = null;
        return value;
    },
    update :function(){
        var newVal = this.get();
        if(this.value != newVal){
            this.cb && this.cb(newVal, this.value);
            this.value = newVal;
        }
    }
};
```

Watcher

- 将Compiler的解析结果，与Observer所观察的对象连接起来建立关系，在Observer观察到对象数据变化时，接收通知，同时更新DOM，称之为Watcher；

```
Observer.prototype.defineReactive = function(data, key, value){
    var dep = new Dep();
    Object.defineProperty(data, key ,{
        enumerable:true,
        configurable:false,
        get:function(){
            if(Dep.target){
                //添加观察者
                dep.addSub(Dep.target);
            }
            return value;
        },
        set:function(newVal){
            if(newVal == value){
                return;
            }
            //data[key] = newVal;//死循环！赋值还会调用set方法
            value = newVal;//为什么可以这样修改？闭包依赖的外部变量
            //遍历newVal
            this.transform(newVal);
            //发送更新通知给观察者
            dep.notify(newVal);
        }
    });

    //递归处理
    this.transform(value);
};
```

```
Observer.prototype.transform = function(data){
    for(var key in data){
        this.defineReactive(data,key,data[key]);
    }
};
```

- 最后，我们将前面遍历data用Object.defineProperty将要观察的对象转为getter/sette的伪代码和观察者模式结合起来如下伪代码，这样逻辑完整Vue.js内部实现的MVVM框架实现机制就呈现出来了。

几个关键点

- 尽管如上关键伪代码逻辑结构完整，但是理解起来却并不容易，为了便于您理解它的运行机制我们简要提示几点：

- 在创建Vue对象时，遍历data用Object.defineProperty将要观察的对象“=”操作转化为getter/setter，但是这时要观察的对象“=”操作并没有发生，可以理解为只是只是对父类重载了getter/setter方法；

- 当编译视图模版创建Watcher对象时，也就是观察者模式中的观察者，其中通过触发getter方法将Watcher对象自身添加到观察者列表中。这一过程几句关键代码按照执行顺序罗列如下：

```
//创建Watcher对象时，获取它要监听占位符/表达式的一个成员对应的getter方法
this.getter = parseExpression(exp).get;
this.update();
//执行该getter方法，如果没有通过Object.defineProperty定义则没有该getter方法
var newVal = this.get();
Dep.target = this;
var value = this.getter?this.getter(this.vm):'';
//该getter方法中添加观察者到观察者列表
dep.addSub(Dep.target);
```

```
//在method中修改message
    methods: {
      reverseMessage: function () {
        this.message = this.message.split('').reverse().join('')
      }
    }
//触发setter方法
      set:function(newVal){
          if(newVal == value){
              return;
          }
          //data[key] = newVal;
//死循环！赋值还会调用set方法
          value = newVal;
//为什么可以这样修改？闭包依赖的外部变量
          //遍历newVal
          this.transform(newVal);
          //发送更新通知给观察者
          dep.notify(newVal);
      }
```

- 当模型中的数据对象被修改时，如何自动更新到视图页面中的呢？我们也简要罗列一下代码执行过程

```
//发送更新通知给观察者
Dep.prototype.notify = function(newVal){
    for(var uid in this.subs){
        this.subs[uid].update(newVal);
    }
};
//观察者更新视图
    update :function(){
        var newVal = this.get();
        if(this.value != newVal){
            this.cb && this.cb(newVal, this.value);
            this.value = newVal;
        }
```

- 弄清了以上几点相信能够理解其中的数据绑定和事件通知工作机制，但这只是一个伪代码示意，实际上Vue.js要复杂的多，有兴趣的话可以参阅Vue.js的源代码https://github.com/vuejs/vue。

软件架构风格与描述方法

- 构建软件架构模型的基本方法
- 软件架构模型的描述方法
- 软件架构风格与策略

软件架构设计是一项非常有挑战性的工作

- 既要考虑满足数量众多的各种系统功能需求，
- 也需要完成诸如系统的易用性、系统的可维护性等非功能性的设计目标，
- 还要遵从各种行业标准和政策法规。
- 不过并不是每一个项目我们都需要从头开始进行完全创新性的设计，更多的是通过研究借鉴优秀的设计方案，来逐步改进我们的设计。换句话说，大多数的设计工作都是通过复用（Reuse）相似项目的解决方案，或者采用一些优秀设计方案的方法，这让看起来非常有挑战性的软件架构设计工作变得有例可循。

两种不同层级的软件架构复用方法

- 克隆（Cloning），完整地借鉴相似项目的设计方案，甚至代码，只需要完成一些细枝末节处的修改适配工作。

- 重构（Refactoring），构建软件架构模型的基本方法，通过指引我们如何进行系统分解，并在参考已有的软件架构模型的基础上逐步形成系统软件架构的一种基本建模方法。

- 这两种软件架构复用方法与生物世界的无性繁殖和有性繁殖极为相似，比如单细胞生物通过细胞分裂实现增殖，实际上就是克隆；而有性繁殖重构了DNA双螺旋结构。复杂系统方面的研究表明在不同层次上的复杂系统常常表现出来相似的结构，这又是一个例证。

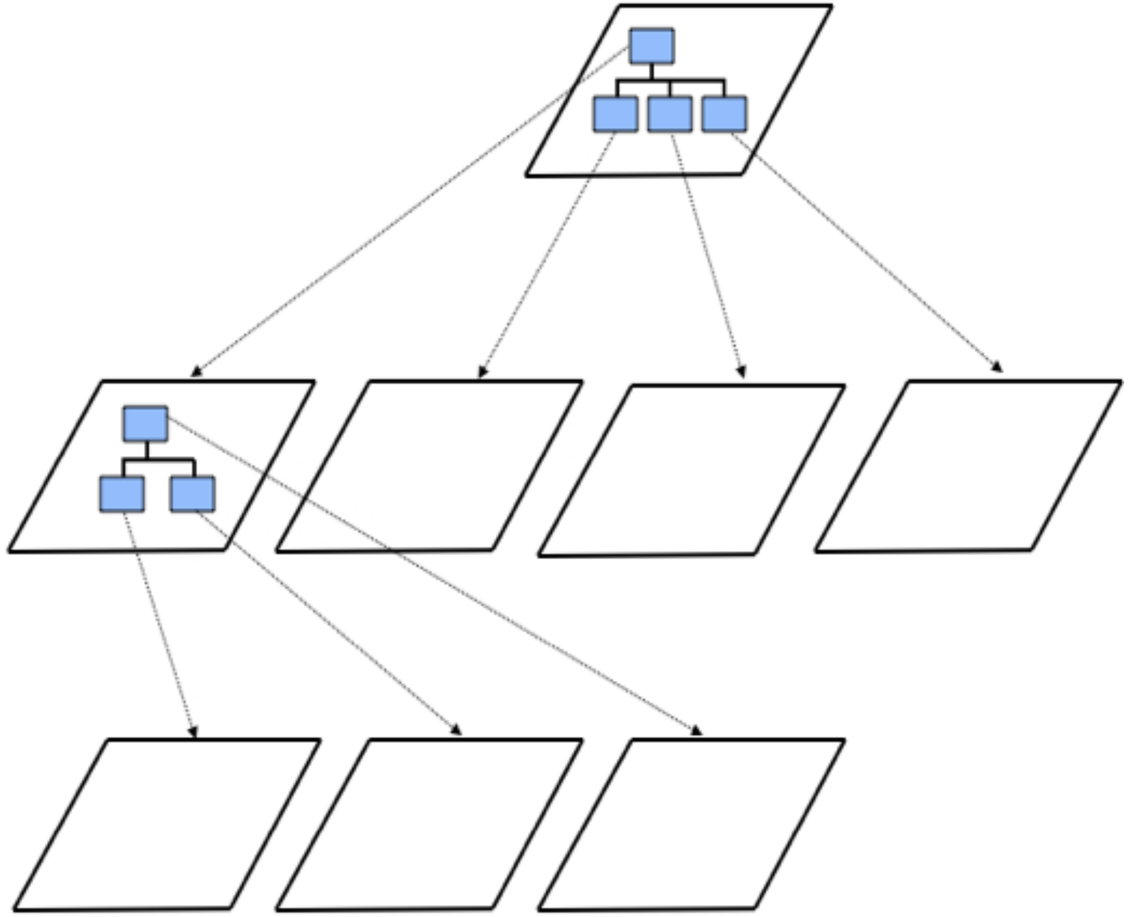
软件架构模型至关重要的作用

- 大型软件系统的软件架构模型在整个项目中起到至关重要的作用。
- 首先，软件架构模型有助于项目成员从整体上理解整个系统；
- 其次，给复用提供了一个高层视图，既可以辅助决定从其他系统中复用设计或组件，也给我们构建的软件架构模型未来的复用提供了更多可能性；
- 再次，软件架构模型为整个项目的构建过程提供了一个蓝图，贯穿于整个项目的生命周期；
- 最后，软件架构模型有助于理清系统演化的内在逻辑、有助于跟踪分析软件架构上的依赖关系、有助于项目管理决策和项目风险管理等。

软件架构模型

- 软件架构模型是在高层抽象上对系统中关键要素的描述，而且表现出抽象的层次结构。

- 构建软件架构模型的基本方法就是在不同层次上分解（decomposition）系统并抽象出其中的关键要素。



分解的常见方法

- 面向功能的分解方法，用例建模即是一种面向功能的分解方法；
- 面向特征的分解方法，根据数量众多的某种系统显著特征在不同抽象层次上划分模块的方法；
- 面向数据的分解方法，在业务领域建模中形成概念业务数据模型即应用了面向数据的分解方法；
- 面向并发的分解方法，在一些系统中具有多种并发任务的特点，那么我们可以将系统分解到不同的并发任务中（进程或线程），并描述并发任务的时序交互过程；
- 面向事件的分解方法，当系统中需要处理大量的事件，而且往往事件会触发复杂的状态转换关系，这时系统就要考虑面向事件的分解方法，并内在状态转换关系进行清晰的描述；
- 面向对象的分解方法，是一种通用的分析设计范式，是基于系统中抽象的对象元素在不同抽象层次上分解的系统的方法。

Popular Design Methods

- Functional decomposition
  - partitions functions or requirements into modules
  - begins with the functions that are listed in the requirements specification
  - lower-level designs divide these functions into subfunctions, which are then assigned to smaller modules
  - describes which modules (subfunctions) call each other

Popular Design Methods

- Feature-oriented decomposition
  - assigns features to modules
  - high-level design describes the system in terms of a service and a collection of features
  - lower-level designs describe how each feature augments the service and identifies interactions among features

Popular Design Methods

- Data-oriented decomposition
  - focuses on how data will be partitioned into modules
  - high-level design describes conceptual data structures
  - lower-level designs provide detail as to how
    - data are distributed among modules
    - distributed data realize the conceptual models

Popular Design Methods

- Process-oriented decomposition
  - partitions the system into concurrent processes
  - high-level design:
    - identifies the system's main tasks
    - assigns tasks to runtime processes
    - explains how the tasks coordinate with each other
  - Lower-level designs describe the processes in more detail

Popular Design Methods

- Event-oriented decomposition
  - focuses on the events that the system must handle and assigns responsibility for events to different modules
  - high-level design catalogues the system's expected input events
  - lower-level designs decompose the system into states and describe how events trigger state transformations

Popular Design Methods

- Object-oriented decomposition
  - assigns objects to modules
  - high-level design identifies the system's object types and explains how objects are related to one another
  - lower-level designs detail the objects' attributes and operations

分解和视图

- 在合理的分解和抽象基础上抽取系统的关键要素，进一步描述关键要素之间的关系，比如面向数据分解之后形成的数据关系模型、面向事件分析之后总结出的状态转换图、面向并发分解之后总结出的并发任务的时序交互过程等，都是软件架构模型中的某种关键视图（Views）。

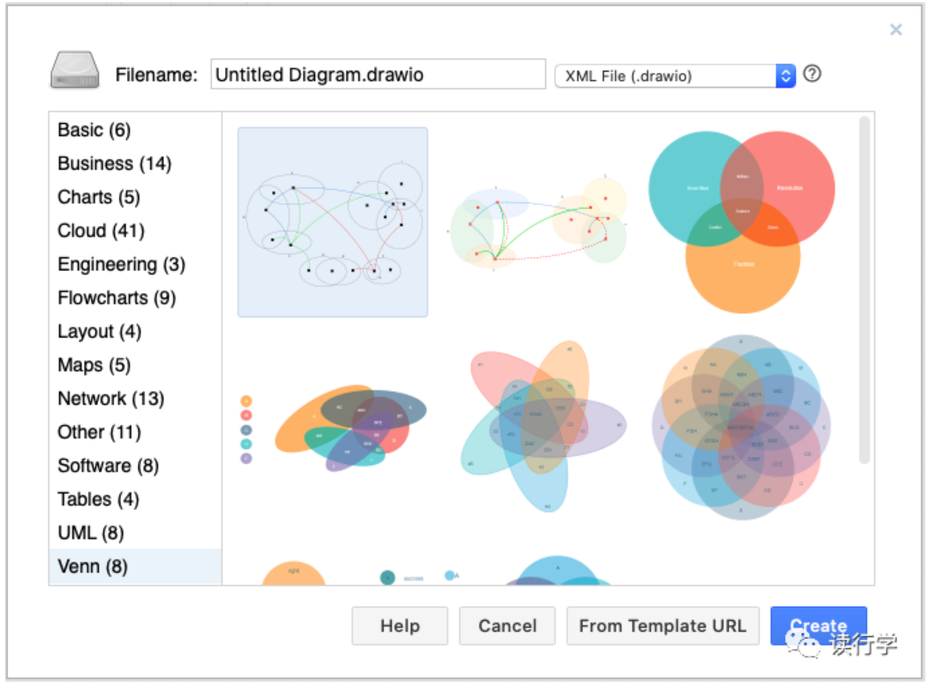
- 软件架构模型是通过一组关键视图来描述的，同一个软件架构，由于选取的视角不同（Perspective）可以得到不同的视图，这样一组关键视图搭配起来可以完整地描述一个逻辑自洽的软件架构模型。一般来说，我们常用的几种视图有分解视图、依赖视图、泛化视图、执行视图、实现视图、部署视图和工作任务分配视图。

软件架构模型的描述方法

- 分解视图 Decomposition View
- 依赖视图 Dependencies View
- 泛化视图 Generalization View
- 执行视图 Execution View
- 实现视图 Implementation View
- 部署视图 Deployment View
- 工作任务分配视图 Work-assignment View

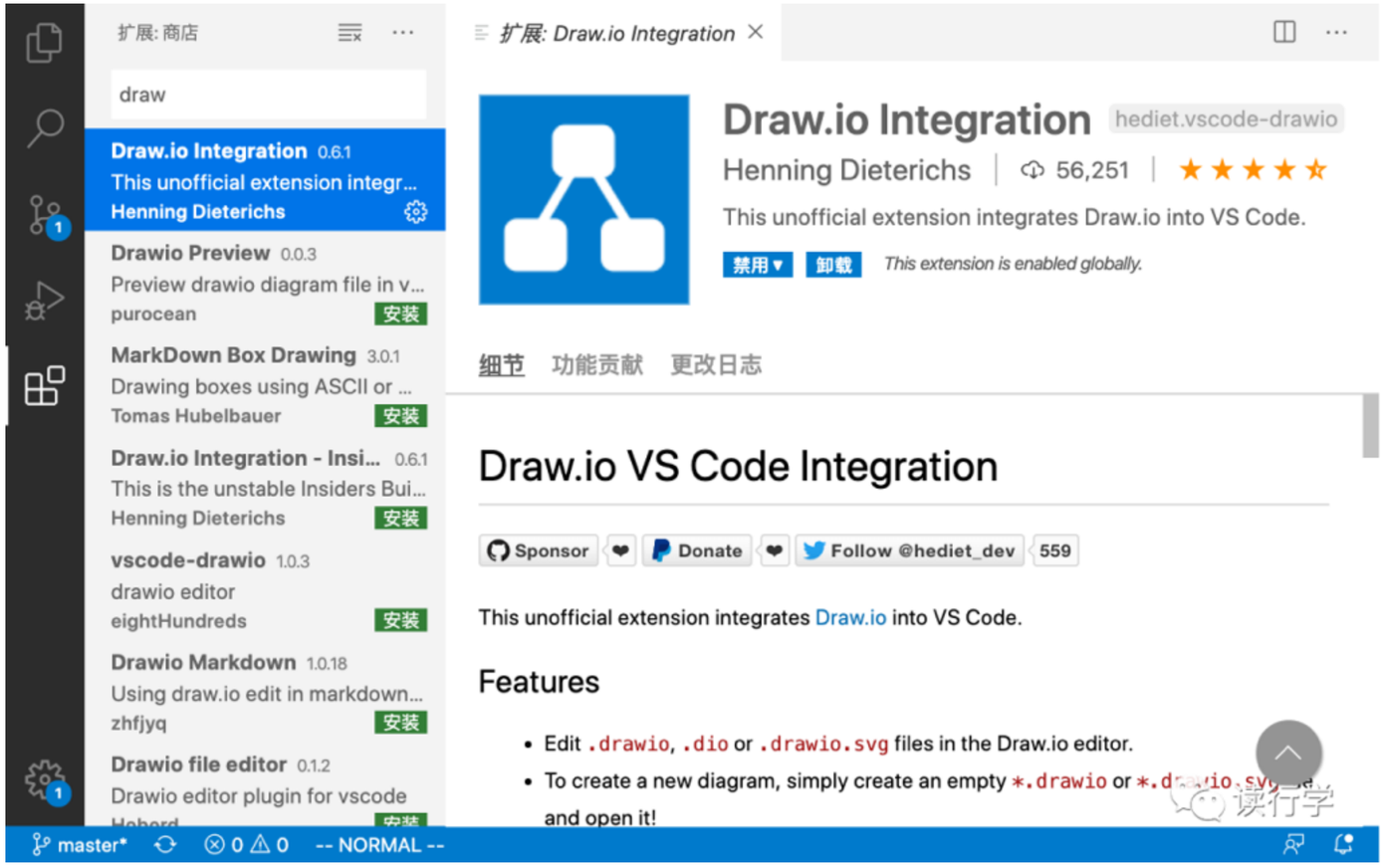
在线绘图工具draw.io

- draw.io是一个在线绘图工具，因其界面简洁直观，功能丰富强大而受到不少用户喜爱。在浏览器中输入draw.io即可访问使用。
- draw.io提供了各类丰富的图形模版
- VS Code中有draw.io插件，可以使用VS Code画图



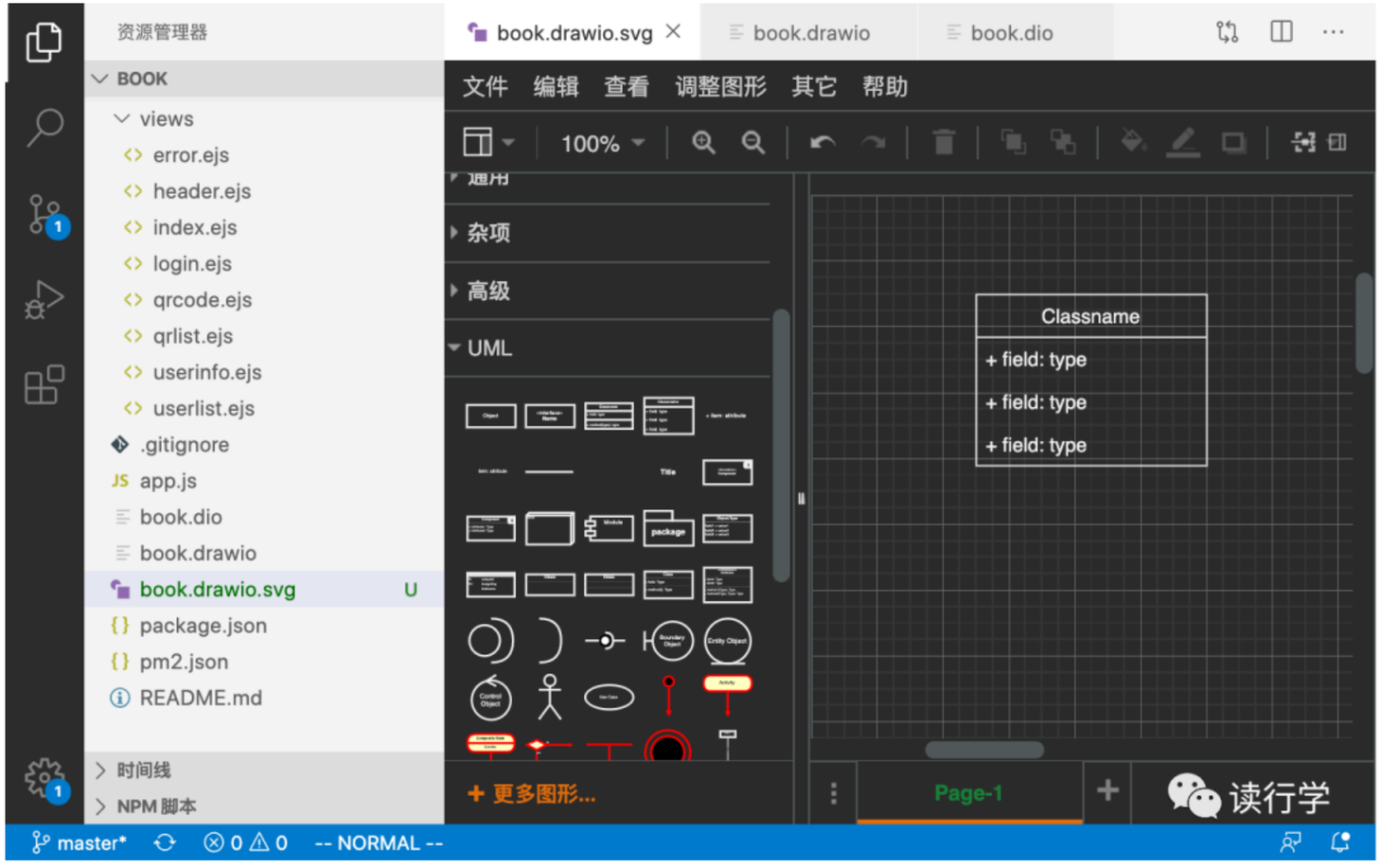
安装VS Code和draw.io插件

- Ctrl+Shift+X进入插件市场，搜索draw已经可以搜到不少插件，我选用了这个下载量比较大、小星星比较多的Draw.io Integration，直接安装就OK了。



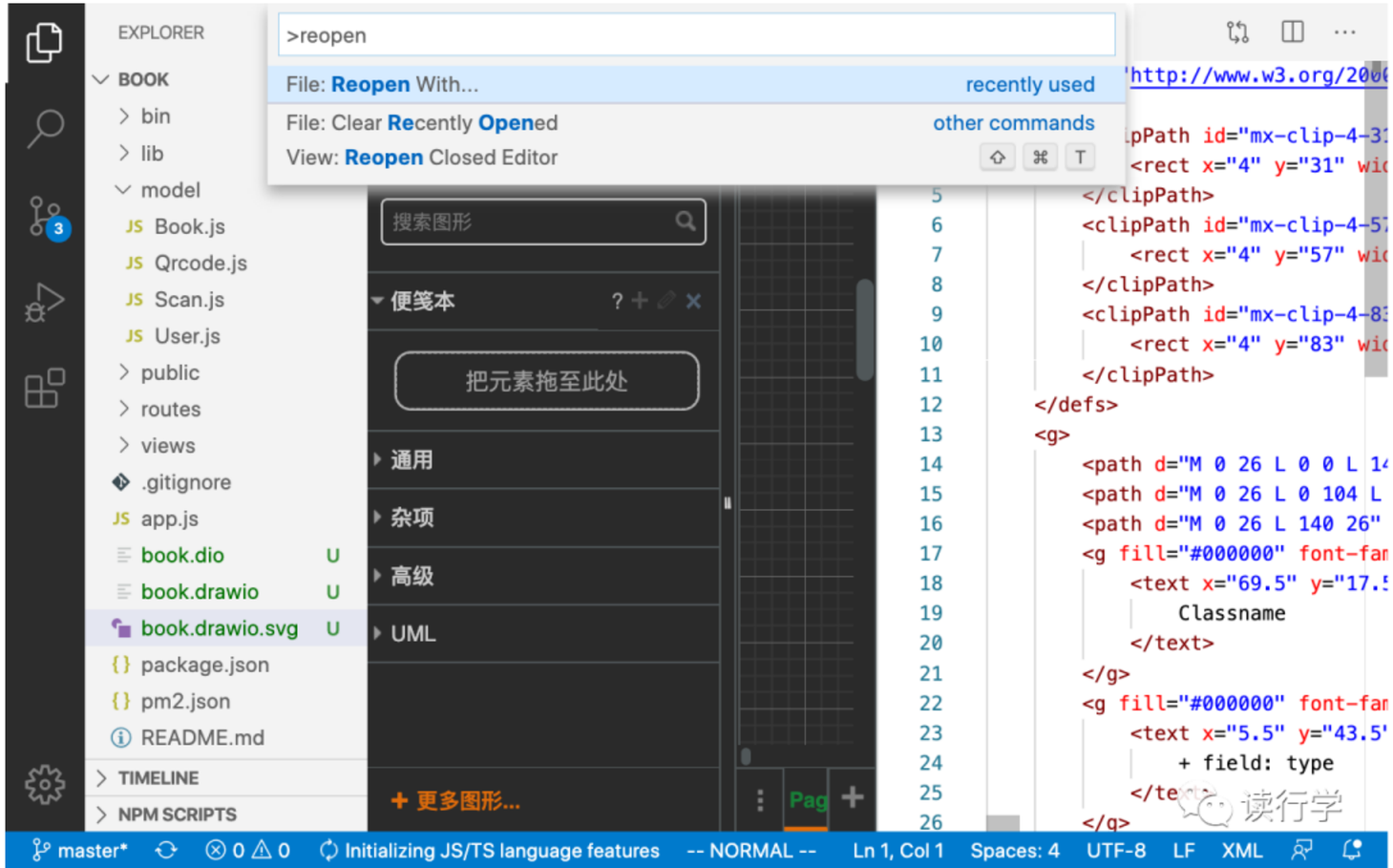
快速入门开始画图

- 只要在创建.drawio.svg、.drawio或.dio文件，然后打开就是所见即所得的画图工具了



通过编辑xml文件修改图形

- Ctrl+Shift+P调出VS Code命令行工具搜索Reopen找到File: Reopen With...就可以选择是以Text Editor编辑器打开



Decomposition View

- The decomposition view portrays the system as programmable units
- This view is likely to be hierarchical
- May be represented by multiple models

Dependencies View

- The dependencies view shows dependencies among software units
- This view is useful in project planning
- Also useful for assessing the impact of making a design change to some software unit

Generalization View

- The generalization view shows software units that are generalizations or specializations of one another
- This view is useful when designing abstract or extendible software units

Execution View

- The execution view is the traditional box-and-arrow diagram that software architects draw, showing the runtime structure of a system in terms of its components and connectors
- Each **component** is a distinct executing entity, possibly with its own program stack
- A **connector** is some intercomponent communication mechanism, such as a communication channel, shared data repository, or remote procedure call

Implementation View

- The implementation view maps code units to the source file that contains their implementation
- Helps programmers find the implementation of a software unit within a maze of source-code files

Deployment View

- The deployment view maps runtime entities, such as components and connectors, onto computer resources, such as processors, data stores, and communication networks
- It helps the architect analyze the quality attributes of a design, such as performance, reliability, and security

Work-assignment View

- The work-assignment view decomposes the system's design into work tasks that can be assigned to project teams
- Helps project managers plan and allocate project resources, as well as track each team's progress

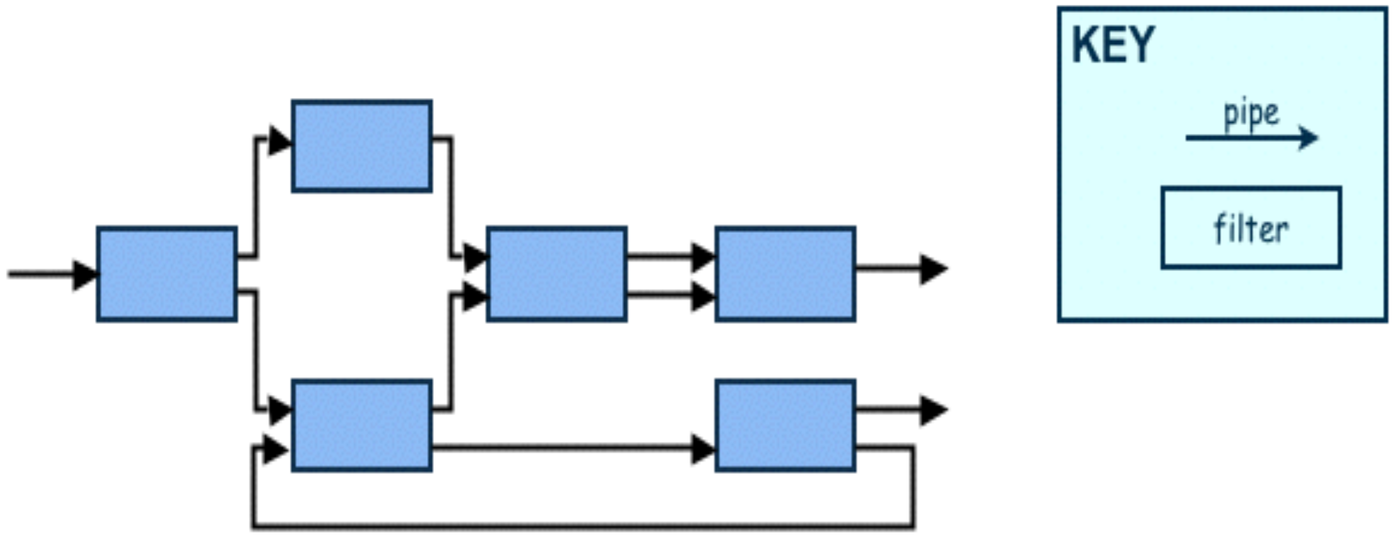
软件架构风格与策略

软件架构风格与策略

- Pipes-and-Filter
- Client-Server
- Peer-to-Peer
- Publish-Subscribe
- Repositories
- Layering

Pipes-and-Filter

- The system has
  - Streams of data (pipe) for input and output
  - Transformation of the data (filter)

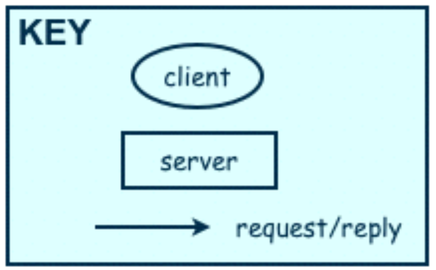


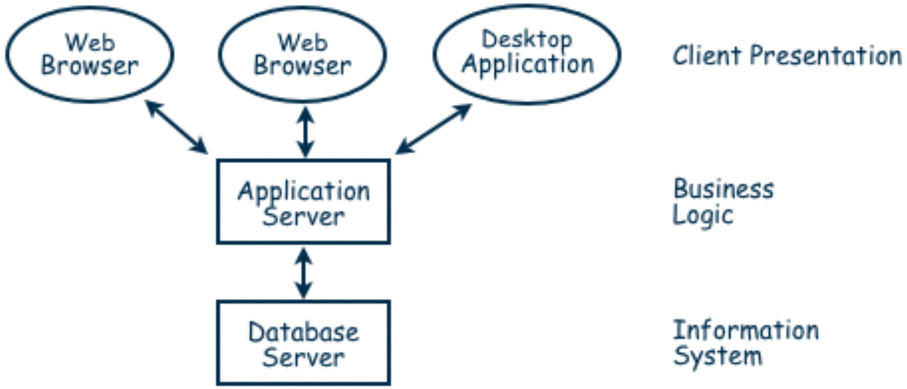
Pipes-and-Filter

- Several important properties
  - The designer can understand the entire system's effect on input and output as the composition of the filters
  - The filters can be reused easily on other systems
  - System evolution is simple
  - Allow concurrent execution of filters
- Drawbacks
  - Encourages batch processing
  - Not good for handling interactive application
  - Duplication in filters functions

Client-Server

- Two types of components:
  - Server components offer services
  - Clients access them using a request/reply protocol
- Client may send the server an executable function, called a callback
  - The server subsequently calls under specific circumstances





Peer-to-Peer (P2P)

- Each component acts as its own process and acts as both a client and a server to other peer components.
- Any component can initiate a request to any other peer component.
- Characteristics
  - Scale up well
  - Increased system capabilities
  - Highly tolerant of failures
- Examples?

Publish-Subscribe

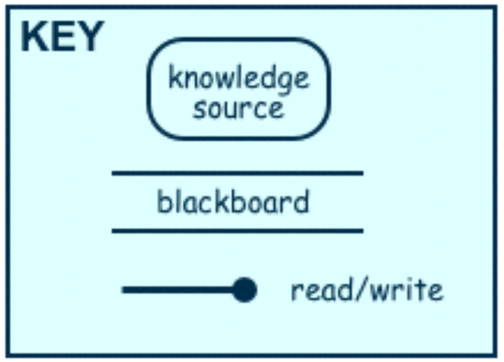
- Components interact by broadcasting and reacting to events
  - Component expresses interest in an event by subscribing to it
  - When another component announces (publishes) that event has taken place, subscribing components are notified
  - Implicit invocation is a common form of publish-subscribe architecture
    - Registering: subscribing component associates one of its procedures with each event of interest (called the procedure)
- Characteristics
  - Strong support for evolution and customization
  - Easy to reuse components in other event-driven systems
  - Need shared repository for components to share persistent data
  - Difficult to test

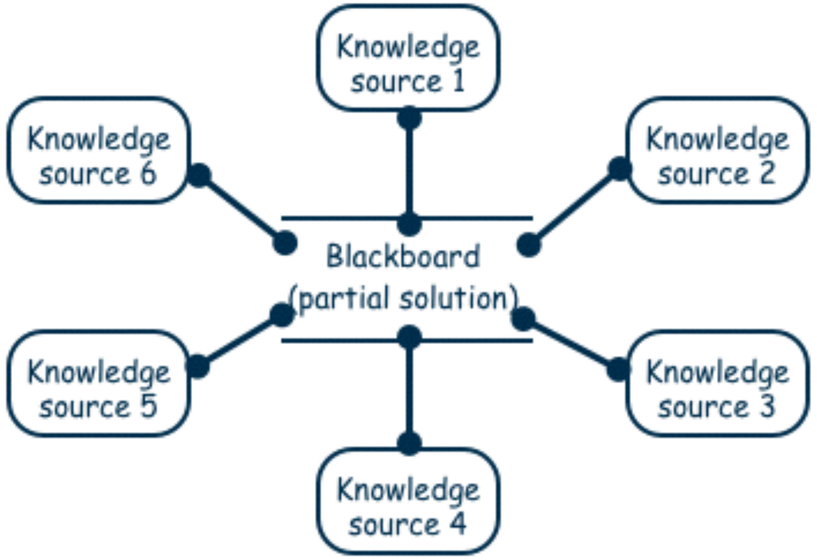
Repositories

- Two components
  - A central data store
  - A collection of components that operate on it to store, retrieve, and update information
- The challenge is deciding how the components will interact
  - A traditional database: transactions trigger process execution
  - A blackboard: the central store controls the triggering process
  - Knowledge sources: information about the current state of the system's execution that triggers the execution of individual data accessors

Repositories

- Major advantage: openness
  - Data representation is made available to various programmers (vendors) so they can build tools to access the repository
  - But also a disadvantage: the data format must be acceptable to all components



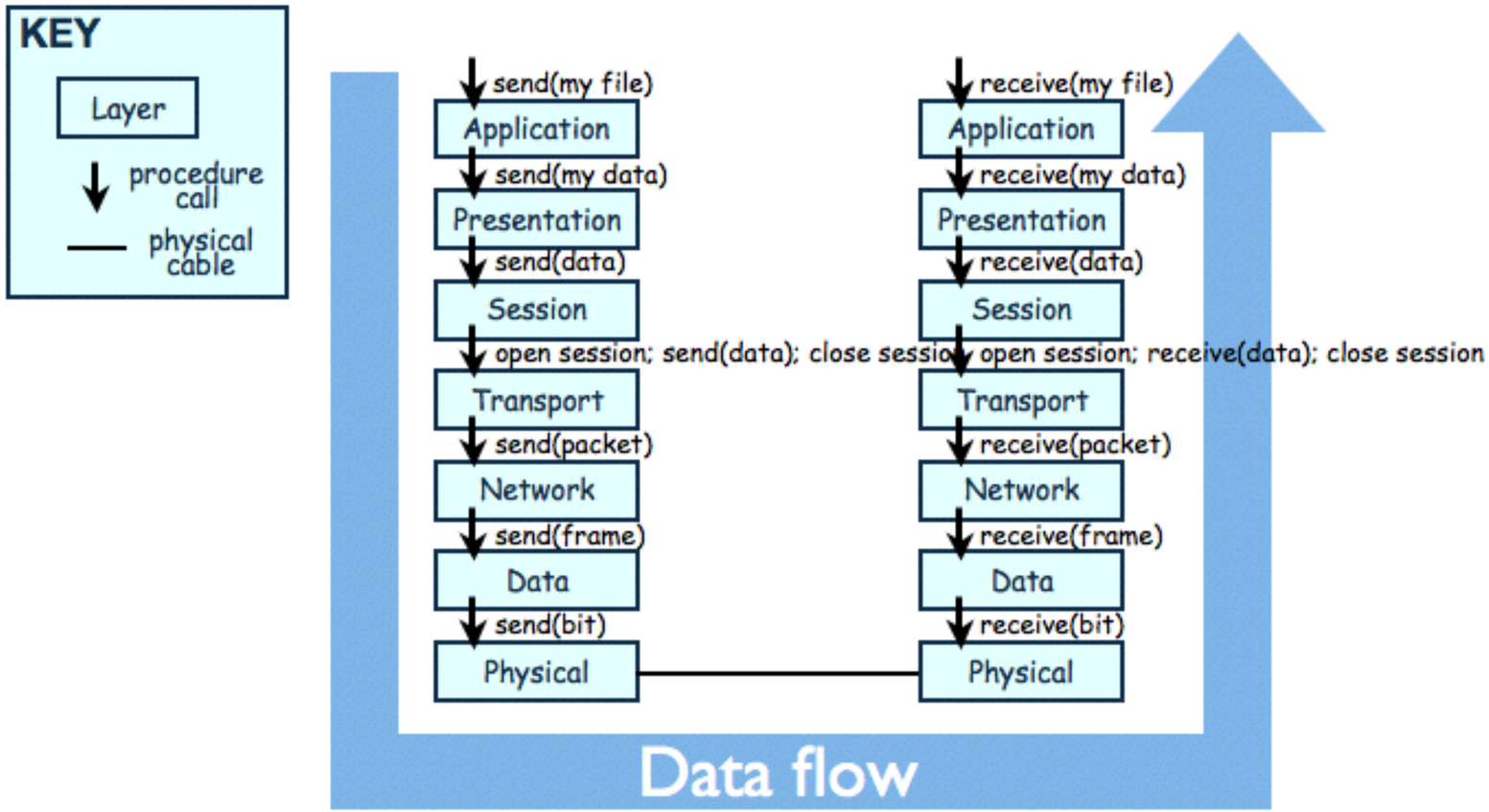


Layering

- Layers are hierarchical
  - Each layer provides service to the one outside it and acts as a client to the layer inside it
  - Layer bridging: allowing a layer to access the services of layers below its lower neighbor
- The design includes protocols
  - Explain how each pair of layers will interact
- Advantages
  - High levels of abstraction
  - Relatively easy to add and modify a layer
- Disadvantages
  - Not always easy to structure system layers
  - System performance may suffer from the extra coordination among layers

Example of Layering System

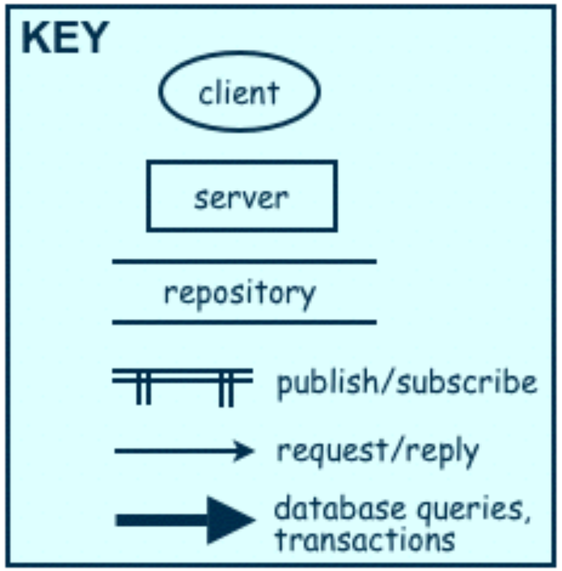
- The OSI Model

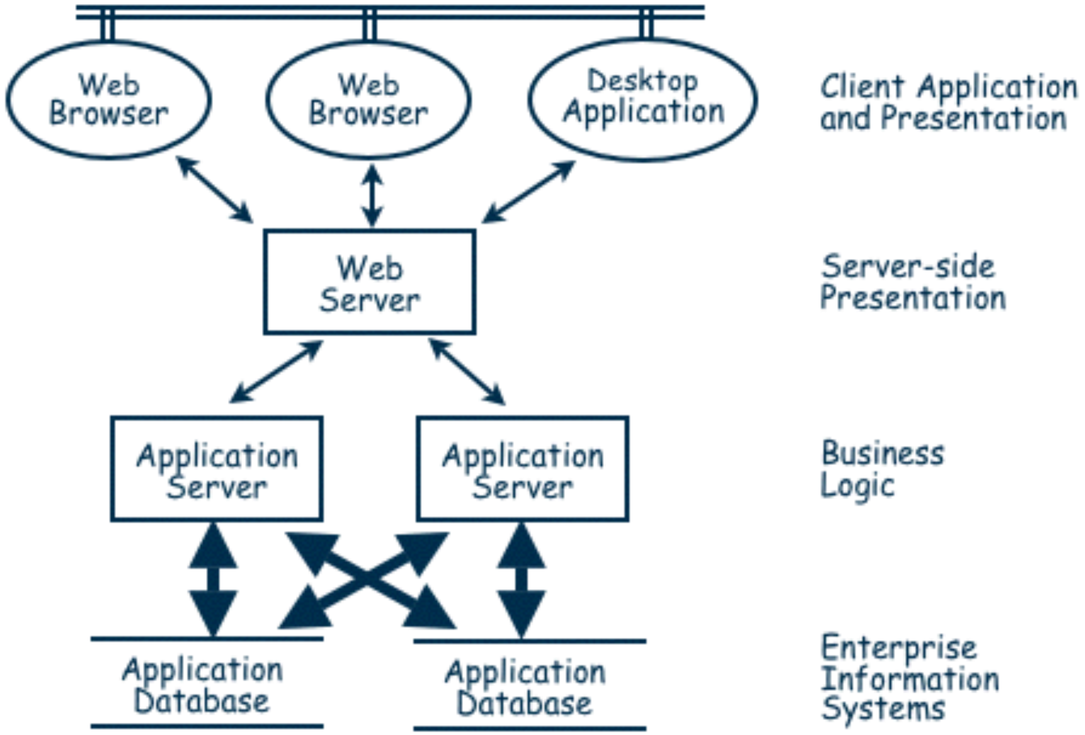


Combining Architectural Styles

- Actual software architectures rarely based on purely one style
- Architectural styles can be combined in several ways
  - Use different styles at different layers (e.g., overall client-server architecture with server component decomposed into layers)
  - Use mixture of styles to model different components or types of interaction (e.g., client components interact with one another using publish-subscribe communications
- If architecture is expressed as collection of models, documentation must be created to show relation between models

Combination of Publish-Subscribe, Client-Server, and Repository Architecture Styles





什么是高质量软件?

- 软件质量的三种视角
- 几种重要的软件质量属性

软件质量的含义

- 生产商：产品符合标准规范
- 消费者：产品适于使用且带来益处
- 按照我们一般的常识理解，质量是指和其他竞争者相比产品或服务有更高的标准，也就是所谓，不怕不识货就怕货比货，质量是在比较中衡量的。
- 按照字典的解释，质量是指较好的一类或优秀的等级。

IEEE的软件质量定义

- IEEE将软件质量定义为，一个系统、组件或过程符合指定要求的程度，或者满足客户或用户期望的程度。
- 软件质量是许多质量属性的综合体现，各种质量属性反映了软件质量的方方面面。人们通过改善软件的各种质量属性，从而提高软件的整体质量 。

不同的视角下的质量

- 质量能够被识别，但是很难准确定义，是一个非常复杂的概念，由许多质量属性综合起来构成，而且在不同的视角下所关注的质量属性也有很大的差异。
- 从用户的角度看，高质量就是恰好满足或超出了用户的预期目标。
- 从工业生产的角度看，高质量就是符合标准规范的程度。
- 从产品的角度看，高质量意味着产品具有良好的产品内在特性。
- 从市场价值的角度看，高质量意味着客户愿意为此付费的数量。
- 从软件工程过程的角度看，我们希望通过高质量的过程打造高质量产品，从而实现产品价值。接下来我们从产品、过程和价值的视角分别讨论软件质量问题。

产品视角下的软件质量

- 用户看到的产品质量和开发者看到的产品质量是不同的，我们将用户看到的产品质量称为外部质量，比如正确的功能、发生故障的数量等；开发者看到的产品质量称为内部质量。比如代码缺陷等。

- 将外部质量和内部质量之间的依赖关系总结出了很多成熟的质量模型，比较常见的质量模型有Jim McCall 软件质量模型（1977 年）、Barry W. Boehm 软件质量模型（1978 年）、FURPS/FURPS+ 软件质量模型、R. Geoff Dromey 软件质量模型、ISO/IEC 9126 软件质量模型（1993 年）和 ISO/IEC 25010 软件质量模型（2011 年）等，其中以 Jim McCall 软件质量模型最具基础性和奠基性。

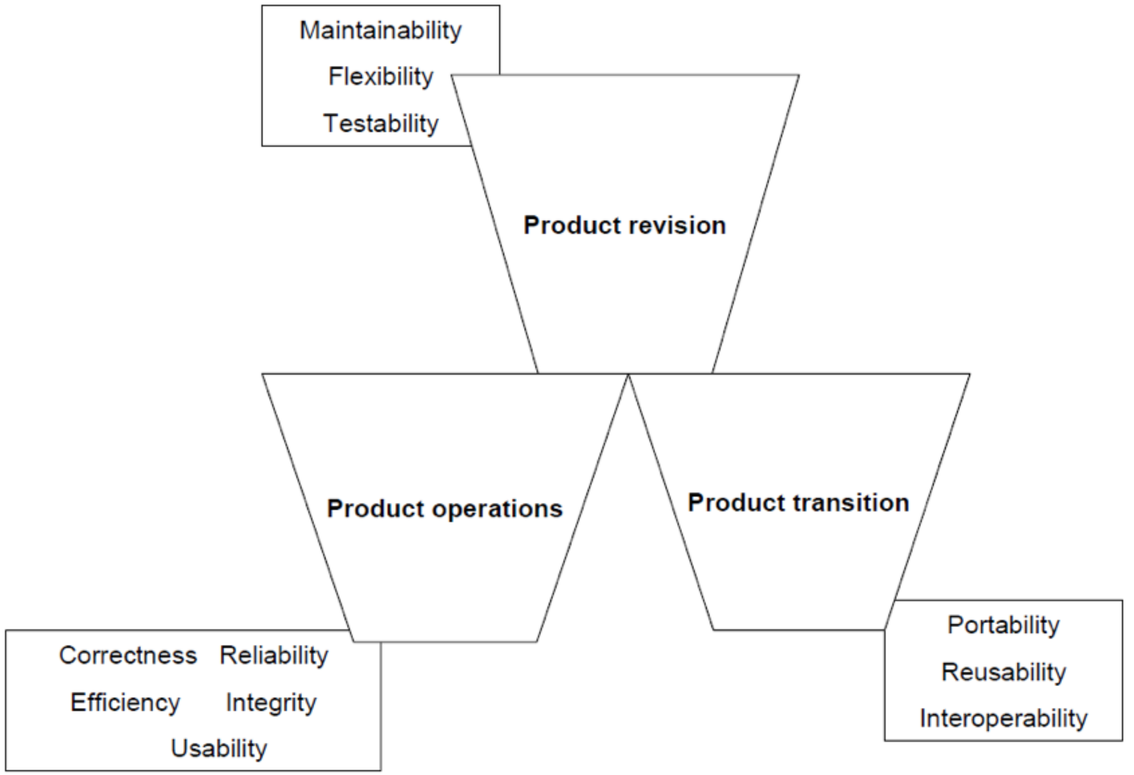
McCall软件质量模型

- Jim McCall软件质量模型，最初起源于美国空军，主要面向的是系统开发人员和系统开发过程。Jim McCall 软件质量模型通过一系列的软件质量属性指标来弥合开发人员与最终用户之间的鸿沟。McCall 质量模型从三个角度来定义和识别软件产品的质量：

- • Product revision (ability to change).

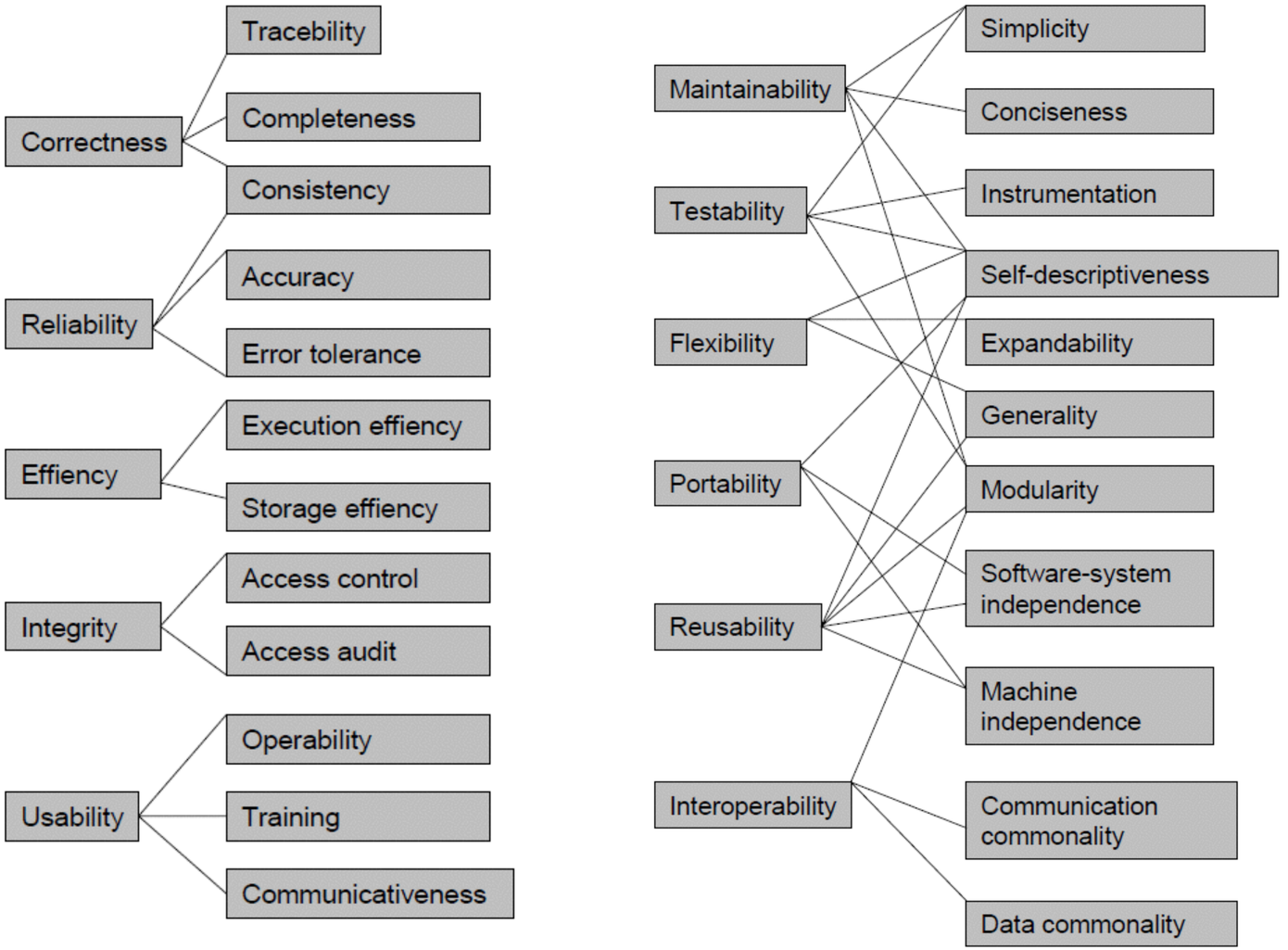
- • Product transition (adaptability to new environments).

- • Product operations (basic operational characteristics).



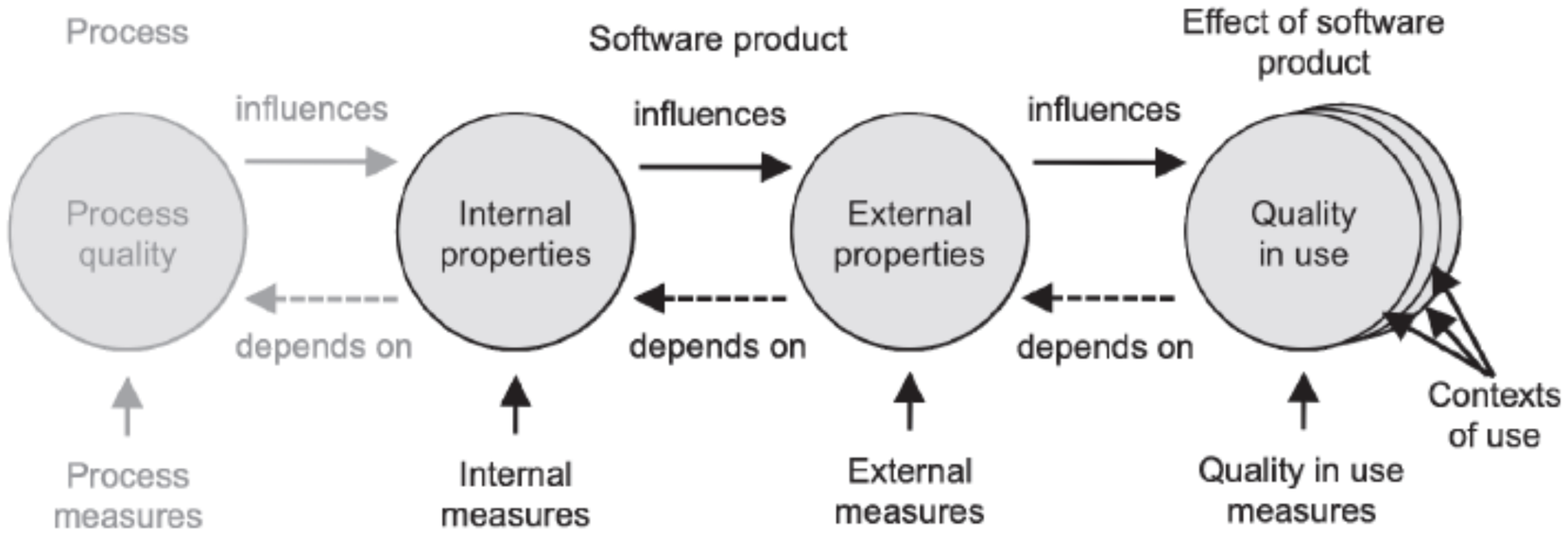
McCall软件质量模型

- 三个角度总结出了11个质量要素，用来描述软件的外部视角，也就是客户或使用者的视角；这11个质量要素又关联着23个质量标准，用来描述软件的内部视角，也就是开发人员的视角。下图中，左侧为 11 个质量要素，右侧为 23 个质量标准。



质量的生命周期

- 从Jim McCall 软件质量模型开始，软件质量模型一路演化分化，为了适应不同的软件类型逐渐变得繁复，在此略过，不过值得一提的是ISO/IEC 25010 软件质量模型（2011年）在外部和内部的软件产品质量基础上给出了质量的生命周期，向前是过程质量，向后是使用质量。使用质量是在特定的使用场景中，软件产品使得特定用户能达到有效性、生产率、安全性和满意度的特定目标的能力。过程质量将是我们下一小节重点讨论的内容。



过程视角下的软件质量

- 软件过程主要是指开发和维护的过程，过程质量和产品质量同样重要，因此也需要对此仔细地建模、研究和分析。过程质量主要需要研究如下几个问题：

- • 特定类型的缺陷在哪里常常出现?
- • 如何尽早发现缺陷?
- • 如何构建容错机制健全质量保证体系?
- • 如何组织安排过程活动可以改善软件质量?

过程改进模型

- 过程质量不像产品质量强调结果性评价，过程质量更关注持续不断的过程改进，因此过程质量模型一般称为过程改进模型，常见的有CMM/CMMI、ISO 9000和Software Process Improvement and Capability dEtermination (SPICE)等。
- 其中以CMM/CMMI最为著名。

CMM/CMMI

- CMM/CMMI的全称为能力成熟度模型（Capability Maturity Model）或能力成熟度模型集成（Capability Maturity Model Integration），CMM/CMMI是美国卡内基-梅隆大学研制的一种用于评价软件生产能力并帮助其改善软件质量的方法，也就是评估软件能力与成熟度的一套标准，它侧重于软件开发过程的管理及工程能力的提高与评估，是国际软件业的质量管理标准。CMM模型自20世纪80年代末推出，并于20世纪90年代广泛应用于软件过程的改进以来，促进了软件过程质量和软件质量的提高，为软件产业的发展和壮大做出了贡献，CMMI是CMM模型的新版本。

- CMMI共有5个级别，代表软件团队能力成熟度的5个等级，数字越大，成熟度越高，高成熟度等级表示有比较强的软件综合开发能力。

能力成熟度的5个等级

- CMMI一级，初始级。在初始级水平上，软件组织对项目的目标与要做的努力很清晰，项目的目标可以实现。但是由于任务的完成带有很大的偶然性，软件组织无法保证在实施同类项目时仍然能够完成任务。项目实施能否成功主要取决于实施人员。

- CMMI二级，管理级。在管理级水平上，所有第一级的要求都已经达到，另外，软件组织在项目实施上能够遵守既定的计划与流程，有资源准备，权责到人，对项目相关的实施人员进行了相应的培训，对整个流程进行监测与控制，并联合上级单位对项目与流程进行审查。二级水平的软件组织对项目有一系列管理程序，避免了软件组织完成任务的偶然性，保证了软件组织实施项目的成功率。

能力成熟度的5个等级

- CMMI三级，已定义级。在已定义级水平上，所有第二级的要求都已经达到，另外，软件组织能够根据自身的特殊情况及自己的标准流程，将这套管理体系与流程予以制度化。这样，软件组织不仅能够在同类项目上成功，也可以在其他项目上成功。科学管理成为软件组织的一种文化，成为软件组织的财富。

- CMMI四级，量化管理级。在量化管理级水平上，所有第三级的要求都已经达到，另外，软件组织的项目管理实现了数字化。通过数字化技术来实现流程的稳定性，实现管理的精度，降低项目实施在质量上的波动。

- CMMI五级，持续优化级。在持续优化级水平上，所有第四级的要求都已经达到，另外，软件组织能够充分利用信息资料，对软件组织在项目实施的过程中可能出现的问题予以预防。能够主动地改善流程，运用新技术，实现流程的优化。

- 由上述的5个级别可以看出，每一个级别都是更高一级的基石。要上高层台阶必须首先踏上所有下层的台阶。

价值视角下的软件质量

- 软件的价值与软件的技术同样重要，在某种程度上决定着软件技术因素的取舍。价值视角下的软件质量表现为软件产品投入运营后的效果，一般在商业环境下软件投入使用后的效果是通过投资回报ROI（return on investment）来量化评价的。投资回报在不同情形下可能被表述为降低成本、预期收益或提高生产力等。

- 对于一个已经投入运营的软件项目来讲，我们可以通过前期投入、运营成本、运营收入、毛利率、税后纯利润以及它们的增长率等来衡量和预测项目的价值，这都有比较成熟的估值方法来衡量软件项目的价值。

新项目的估值

- 对于一个还没有投入运营的项目，还停留在想法或纸面上的软件项目方案，该如何估算资本投入、运营成本、运营收入、毛利率、税后纯利润以及它们的增长率等？换句话如何给一个待开发的或正在开发的软件项目估值是一个非常有挑战性的工作，因为对软件的商业价值的预估决定着对软件开发过程以及软件产品本身的投资强度，以实现可以接受的价值视角下的软件质量。

- 一般从两个角度来估值，一个预估市场规模，一个是预估开发成本。

新项目的估值

- 根据项目的预期市场规模及其所能带来营业收入，可以得到一个项目的理想价值，根据达成理想价值的难度系数不同得出项目的估值。

- 我们也常常用软件开发的成本来衡量软件的价值，软件开发成本的主要部分是人力成本，只要估算出项目所需投入的人月数就可以大致估算出项目的开发成本。一般情况下软件的价值应该高于软件的开发成本。

- 当然软件价值并不仅仅是指商业价值，在某些情况下可能关注的是科研价值，有的情况下关注的是社会效益，不管是何种价值取向，最终都会转化为对软件的投资这一量化指标，这里的投资意味着人力、资金、时间等资源的投入。

几种重要的软件质量属性

- 易于修改维护（Modifiability）
- 良好的性能表现（Performance）
- 安全性（Security）
- 可靠性（Reliability）
- 健壮性（Robustness）
- 易用性（Usability）
- 商业目标（Business goals）

易于修改维护（Modifiability）

- 包容变化是软件设计质量的关键需求。如何才能包容变化？理想的情况是设计和代码能够应对变化而本身不需要修改和维护，但现实是软件难以避免需要修改和维护，那么设计和代码易于修改和维护就能有效地包容变化。那什么样的设计和代码易于修改和维护的呢？那就回到了我们从代码编写、需求分析和系统设计等贯穿始终的一个主题——高内聚低耦合的模块化设计。

高内聚低耦合

- 软件模块的高内聚可以将变化的需求局限于单个软件模块内部，从而易于修改维护；

- 软件模块之间的低耦合可以将修改维护所产生的直接影响和间接影响明确地限定在特定的范围内，从而大大降低修改维护对整体软件系统带来的系统性扰乱。

高内聚和通用性

- 需求变更（变化）直接影响的软件模块会带来模块职责（功能内聚）发生改变，因此直接影响的软件模块需要根据需要进行修改；需求变更（变化）间接影响的软件模块不会造成模块职责（功能内聚）的改变，因此间接影响的软件模块仅需要做适应性的修改，一般是软件模块接口及实现的修订。

- 最小化受变化直接影响的软件模块数量的策略，需要在模块化设计过程中预测预期的需求变更（变化），从而确定最有可能更改的设计决策，并将每个决策封装在独立软件模块中。

- 在模块化设计中保持软件模块的高内聚，使得变化带来的修改和维护仅限于分配相应职责的少数软件模块。

- 软件模块越通用就越有可能适应变化，即通过修改软件模块接口的输入而无需修改软件模块本身。

低耦合和接口

- 最小化受变化间接影响的软件模块数量的策略侧重于减少依赖关系，即降低耦合度会减小一个软件模块的更改所波及到的其他软件模块的数量。

- 如果软件模块仅通过接口与其他软件模块交互，则更改的影响不会超出一个软件模块的边界，除非软件模块之间的接口也发生了改变。

- 即便需要接口也发生改变，我们可以为修改的软件模块增加新的接口，而不去修改软件模块任何现有接口，从而减小所波及到的其他软件模块的数量。

良好的性能表现（Performance）

- 良好的性能表现主要体现在系统的速度和容量上。比较典型的几个性能指标有：

- • 响应时间（Response time），就是软件响应请求的速度有多快。在响应时间的分析中，在底层受到操作系统根据优先级、FIFO或时间片的进程调度策略的影响；在上层受到网络延时、业务处理、数据库访问等应用处理过程的影响。为了追求响应时间性能的提高，需要根据软件类型的不同，比如实时系统、Web服务等，在不同的层次上分析其中影响响应时间的关键因素并加以改进。

- • 吞吐量（Throughput），是单位时间内能处理的请求数量，

- • 负载（Load），用系统能同时支持的用户数量来衡量负载能力，一般是在响应时间和吞吐量受到影响之前的负载能力。

提高性能的策略

- 提高性能的策略包括：
- - 提高资源的利用率
- - 更有效地管理资源分配
- ○ 先到先得：按收到请求的顺序处理
- ○ 显式优先级：按其分配优先级的顺序处理请求
- ○ 最早截止时间优先：按收到请求的截止时间的顺序处理请求
- - 减少对资源的需求

安全性（Security）

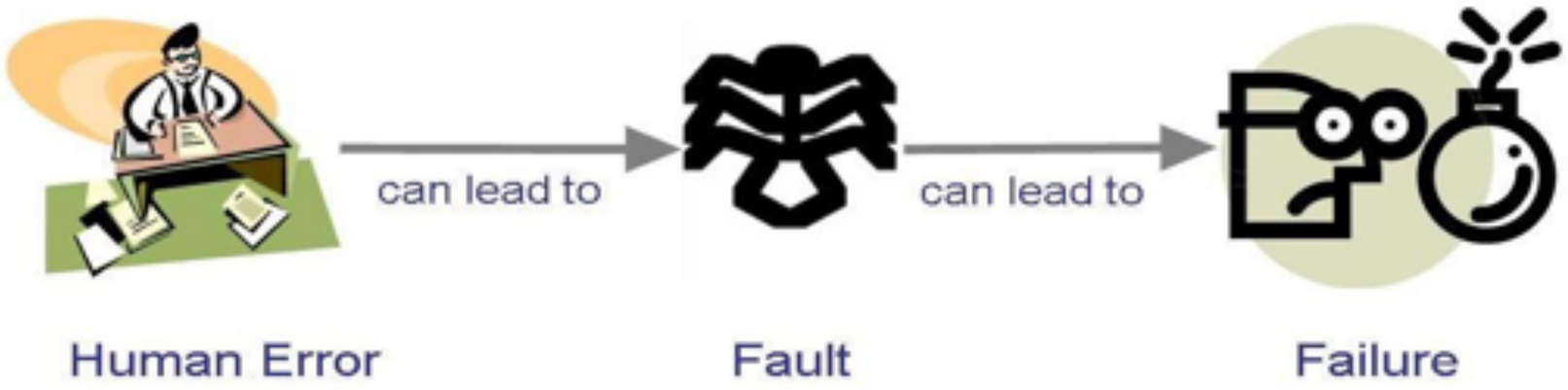
- 与安全性特别相关的两个关键软件架构特征：系统的免疫力（Immunity）和系统的自我恢复能力（Resilience）。

- 系统的免疫力（Immunity）是挫败未遂攻击的能力。一般在软件设计中要确保所有安全需求都包含在设计中得到考虑，并尽可能减少可利用的安全漏洞。

- 系统的自我恢复能力（Resilience）是指可以快速轻松地从攻击中恢复的能力。在软件设计中要包含异常检测机制，一旦发现攻击导致的异常能启动自我恢复程序。

可靠性（Reliability）

- 如果软件系统在假定的条件下能正确执行所要求的功能，则软件系统是可靠的（reliable）。可靠性与软件内部是否有缺陷（Fault）密切相关，而软件内部的缺陷产生的前因后果大致如下图所示：



可靠性（Reliability）

- 与故障（Failure）相比，缺陷是人为错误（Human Error）的结果，而故障是可观察到的偏离要求的行为表现，是缺陷在某种条件下造成的系统失效。

- 可以通过防止或容忍缺陷的存在使软件更加可靠，一般通过缺陷检测和故障恢复两类方法来提高软件的可靠性。缺陷检测主要包括被动缺陷检测、主动缺陷检测和异常处理等，故障恢复的方法则是根据系统设计中创建的应急方案执行撤销、回退、备份、服务降级、修复或报告等。

- • 被动缺陷检测，等待执行期间发生故障。

- • 主动缺陷检测：定期检查症状或尝试预测故障何时发生。

健壮性（Robustness）

- 如果软件能够很好地适应环境或具有故障恢复等机制，则软件是健壮的（robust）。健壮性是比可靠性更高的软件质量要求。可靠的软件只说明它在假定的条件下能够正确地执行，也就是输入和环境是符合要求的，而健壮的软件在不正确的输入或异常环境条件下还能正确执行。也就是说，可靠性和软件内部是否有缺陷有关，而健壮性与软件容忍错误或外部环境异常时的表现有关。

“防人之心不可无，害人之心不可有”

- 在软件设计和编码过程中提高健壮性的一个重要策略是“防人之心不可无，害人之心不可有”。“防人之心不可无”要求我们对所有的外部输入、返回值和其他上下文环境条件进行检测，只允许软件或软件模块在正确的输入、返回值和特定的环境条件下才会继续执行；“害人之心不可有”则要求我们在调用访问外部软件或软件模块时，也要确保自己提供的输入和其他上下文环境条件是符合规格要求的。

- 这种相互怀疑的策略，通过假定外部软件有缺陷或者有恶意攻击的可能，可以有效提高软件整体的健壮性。

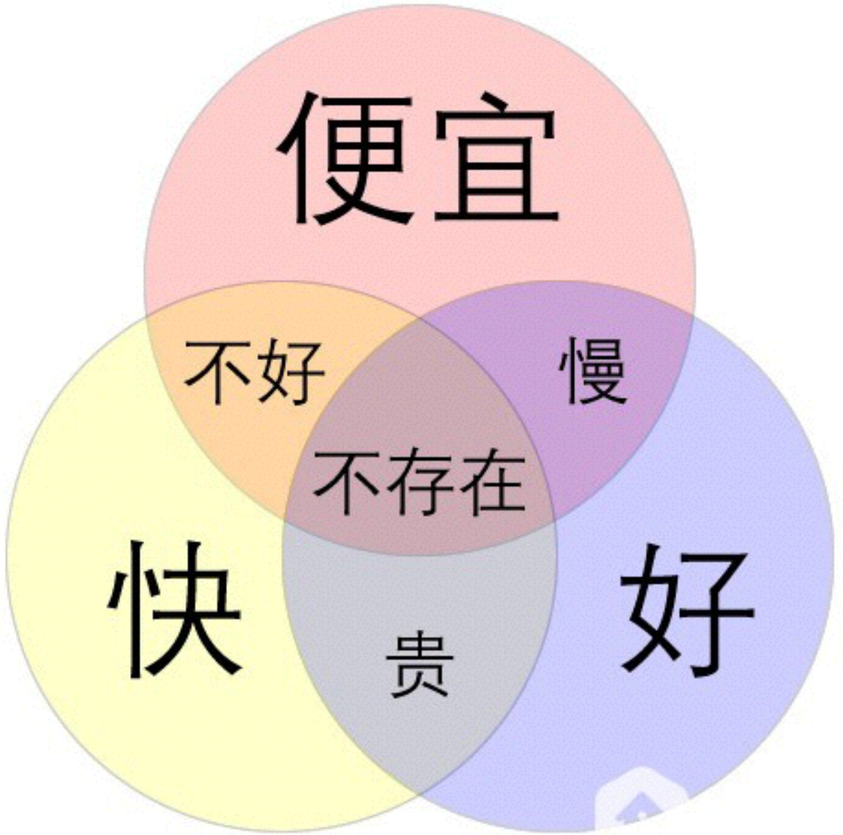
- 异常处理和故障恢复的基本方法在可靠性和健壮性上是通用的，都是根据系统设计中创建的应急方案执行撤销、回退、备份、服务降级、修复或报告等。

易用性（Usability）

- 易用性反映了用户操作软件的容易程度。易用性更多地体现在软件的人性化交互设计中，其中做到交互逻辑与业务逻辑自然统一，符合用户操作习惯和思维习惯非常关键。除此之外，软件架构上需要支持易用性设计，比如支持多种语言扩展、不同操作方式的自定义、撤销/重做、自动保存和错误恢复等。

商业目标（Business goals）

- 商业目标是客户希望软件系统表现出的某些质量属性，其中最普遍的目标是在软件质量有保障的情况下将开发成本降到最低、完成时间尽量提前，简而言之就是又好又快又便宜。



一些关键商业决策

- 又好又快又便宜是不存在的，因此我们需要考虑软件设计中的一些关键商业决策，
- 比如对于软件或软件模块是从外部购买还是独立构建，
- 还有将开发成本和维护成本结合起来估算综合成本，
- 再比如涉及相关软件技术选型时如何决定是采用新技术还是采用已知的成熟技术？

从外部购买还是独立构建?

- 先来讨论软件或软件模块是从外部购买还是独立构建。
- 从外部购买和独立构建哪种方式更节省开发时间和资金成本?
- 从外部购买和独立构建哪种方式更可靠?
- 独立构建软件或模块有哪些制约因素?
- 从外部购买易受供应商的影响，有哪些影响?

开发成本和维护成本

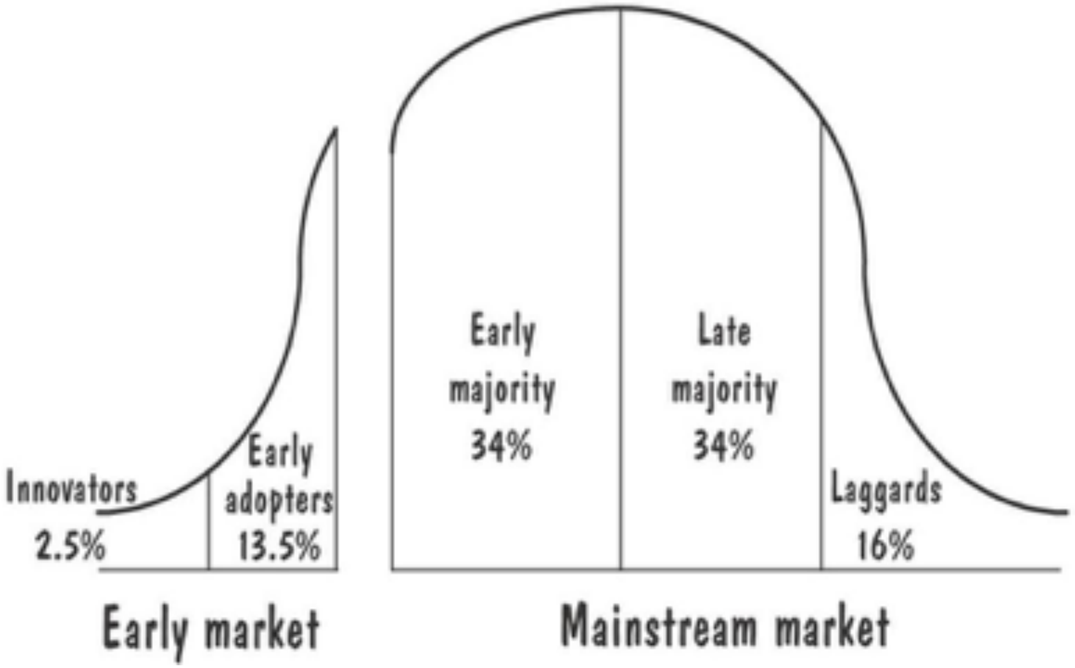
- 再来讨论将开发成本和维护成本结合起来估算综合成本。
- • 通过使软件易于修改维护来节省总体成本，也就是通过降低维护成本来减少综合成本；
- • 使软件易于修改维护可能会增加软件的复杂性，造成延迟发布，使得开发成本上升，并有可能会因为延迟发布而输给竞争对手丢掉部分市场。

采用新技术还是采用已知的成熟技术?

- 最后来讨论是采用新技术还是采用已知的成熟技术。

- • 采用新技术需要额外的资源投入，并可能会延迟产品发布。因为要么学习如何使用新技术，要么雇佣新员工，最终必须自己掌握新技术

- • 新技术和已知的成熟技术各自处于进入主流市场的哪个阶段？综合评估采用新技术对于未来市场的竞争优势，以及采用已知的成熟技术对于未来市场的竞争劣势。



参考资料

- Software Engineering: Theory and Practice (Fourth Edition),Shari Lawrence Pfleeger,Joanne M. Atlee
- https://github.com/fwing1987/MyVue
- https://vuejs.org
- http://c.biancheng.net/design_pattern/