软件科学基础概论

软件的基本结构

1. 顺序结构
2. 分支结构
3. 循环结构
4. 函数调用框架
5. 继承和对象组合

面向对象的真正威力是封装。在大多数情况下，应该用对象组合替代继承来实现相同的目标。

软件中的一些特殊机制

1. 回调函数
2. 多态
3. 闭包
4. 异步调用
5. 匿名函数

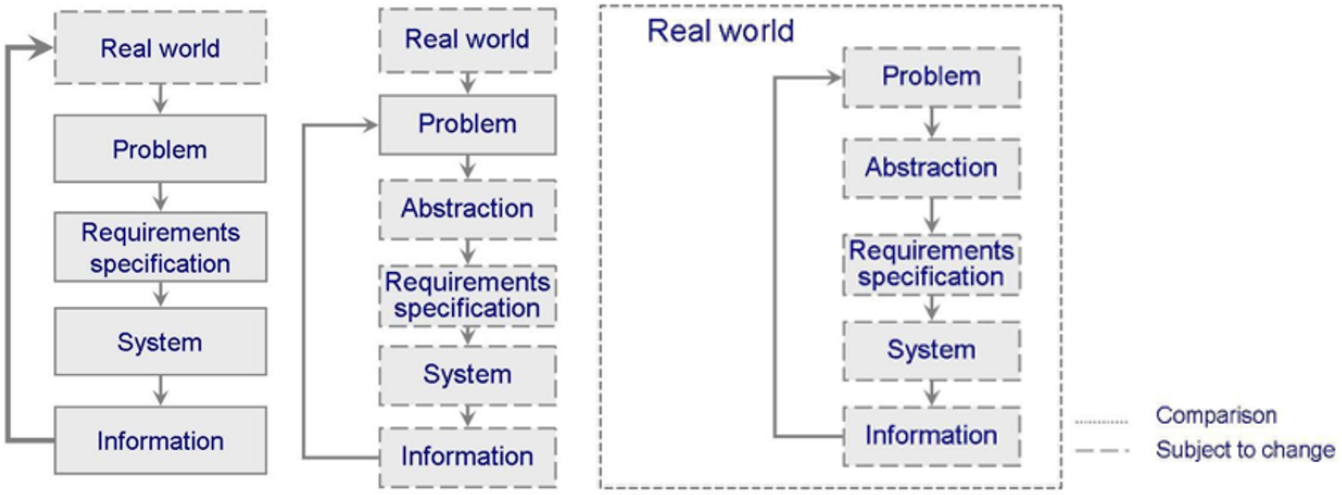
软件的复杂性和易变性

三类系统

唯一不变的就是变化本身

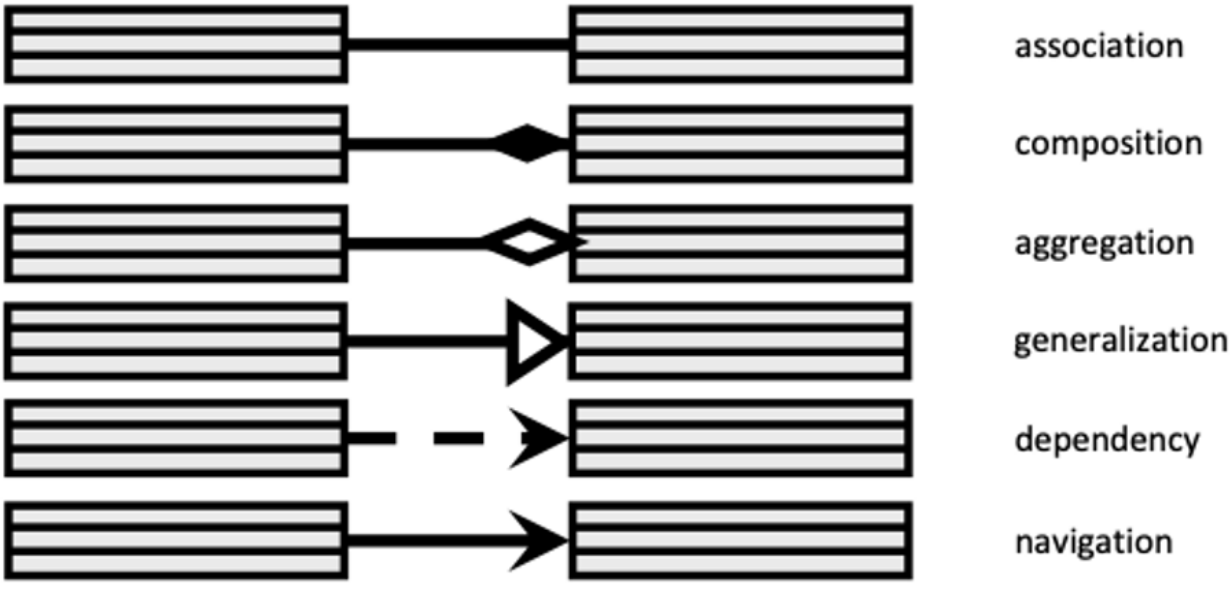
- Lehman将系统分成三种类型：S系统、P系统和E系统。E系统很好地解释了软件易变性的本质原因。我们依次看看这三种系统类型。
- S-system: formally defined, derivable from a specification
  - Matrix manipulation矩阵运算
- P-system: requirements based on approximate solution to a problem, but real-world remains stable
  - Chess program
- E-system: embedded in the real 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



设计模式

关系



设计模式是在某一情境下的问题解决方案

设计模式四个部分

1. 名称
2. 目的
3. 解决方案

4. 解决方案的约束和限制条件

设计模式分类

类模式：用于处理类和子类之间的关系，是静态的，在编译时刻就确定下来了。如模板方法。
对象模式：处理对象之间的关系，可以通过组合或聚合来实现，运行时刻可以变化。组合或聚合关系比继承关系耦合度低，多数设计模式是对象模式。

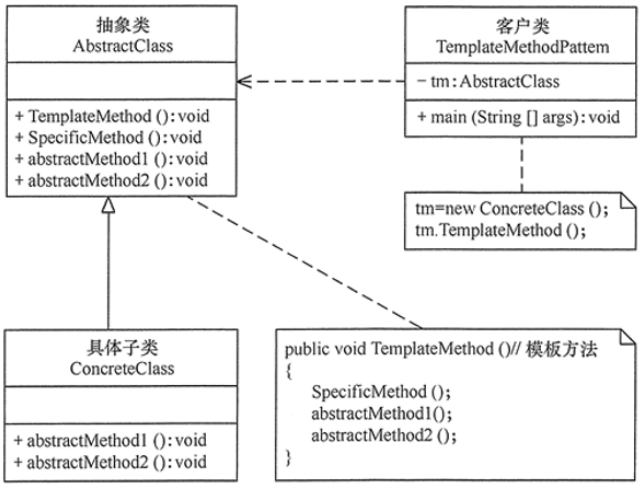
创建型模式：描述"怎样创建对象"，特点是"将对象的创建与使用分离"
结构型模式：将类或对象布局组成更大的结构。分为类结构型模式和对象结构型模式。
行为型模式：描述程序在运行时复杂的流程控制，多个类或对象之间怎样相互协作共同完成单个对象都无法完成的任务。

常用设计模式

单例：
原型：将一个对象为原型，对其复制克隆多个和其类似的新实例。
建造者：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。
代理：不要和陌生人说话原则。
适配器：将一个类的接口转换成客户希望的另外一个接口。
装饰：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。用对象组合地方式扩展功能。 外观：为复杂的子系统提供一致的接口。
享元：用共享技术有效支持大量细粒度对象的复用。
策略：封装一系列算法，算法的改变不影响使用算法的用户。是多态和对象组合的综合应用。
命令：将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：避免请求者与多个请求处理者耦合在一起，所有处理者通过前一对象记住其下一个对象引用而形成一条链。
中介者：定义一个中介对象简化原有对象之间的交互关系，降低系统中对象之间的耦合度。 在MVC框架中，控制器（C）就是模型（M）和视图（V）的中介者。

设计模式背后的原则

开闭原则

软件应当对扩展开放，对修改关闭。

Liskov替换原则

子类可以扩展父类的功能，但是不能修改父类的功能。 **依赖倒置原则**

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。要面向接口编程，不要面向实现编程。

单一职责原则

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）。

迪米特法则

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。

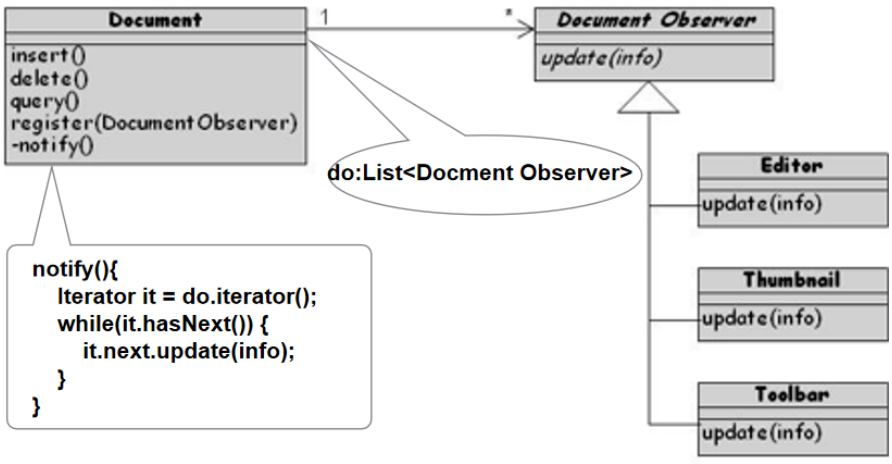
合成复用原则

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。

观察者：

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));
        }
    }
}
```

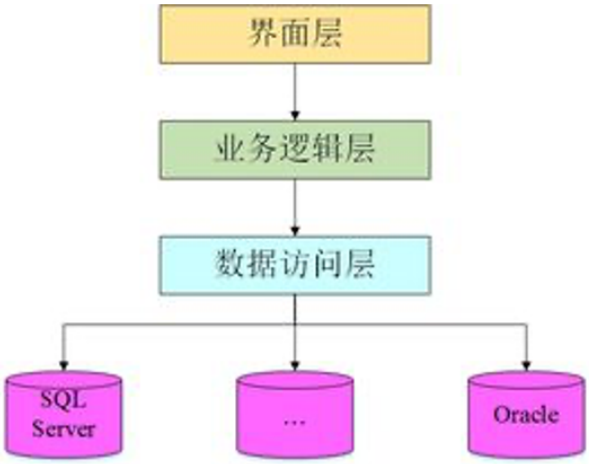
```java
package Observer;
import java.util.Iterator;
import java.util.Vector;

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;
    }
}
```

常见的软件架构举例

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。

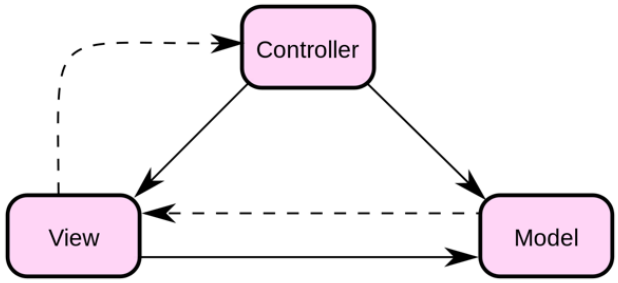


MVC

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。

应用MVC架构的软件其基本的实现过程

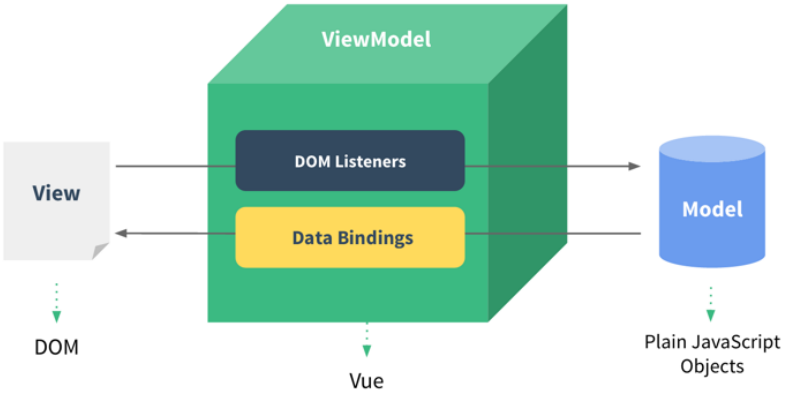


- 控制器创建模型；
- 控制器创建一个或多个视图，并将它们与模型相关联；
- 控制器负责改变模型的状态；
- 当模型的状态发生改变时，模型会通知与之相关的视图进行更新。
- 可以看到这与抽象简化的MVC设计模式已经有了明显的区别，变得更为复杂，但这与实际软件结构相比还是极其简化的模型，实际情况可能会有更多合理的和不合理的复杂联系，要保持软件结构在概念上的完整性极为困难。

MVVM

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

Vue.js是一个前端构建数据驱动的Web界面的库，主要特色是响应式的数据绑定。

Vue.js机制

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

```
//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;
```

- 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？

Watcher

```
//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;
        }
    }
};
```

- 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。

尽管有"=",但这时"="操作并没有发生，可以理解为只是对父类重载了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;
        }
```

软件架构风格与描述方法

两种不同层级软件架构复用方法

克隆

重构

Popular Design Methods

1. functional decomposition
2. feature-oriented decomposition
3. data-oriented decomposition
4. process-oriented decomposition
5. event-oriented decomposition
6. object-oriented decomposition

软件架构风格与策略

1. 管道-过滤器
2. C/S
3. P2P
4. 发布-订阅
5. CRUD
6. 层次化

软件架构的描述方法

分解视图

呈现较为明晰的分解结构。

依赖视图

展现软件模块之间的依赖关系

泛化视图

展现模块间一般化和具体化的关系。如继承关系。

执行视图

系统运行时时许结构特点。流程图、时序图等。

一般将执行视图转换为伪代码之后，再进一步转换为实现代码。 **实现视图**

软件架构与源文件之间的关系。 ~ 源文件目录树

有助于在源代码文件中找到具体软件单元的是实现。

部署视图

执行实体和计算机资源建立映射关系。

有助于设计人员分析一个设计的质量属性。

工作分配图

分解成可独立完成的任务，分配给各项目团队和成员。

高质量软件

用户角度 产品角度

McCall软件质量模型

三个角度

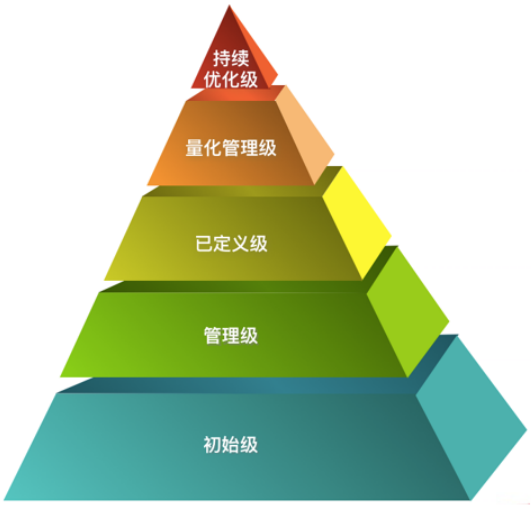
1. Product revision (ability to change)
2. Product transition (adaptability to new environments)
3. Product operations (basic operational characteristics)
   三个角度11个质量要素。

过程角度 CMM/CMMI 能力成熟度5个等级

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。

价值角度

软件质量属性

1. 可修改性
2. 性能
3. 安全性
4. 可靠性
5. 健壮性
6. 易用性
7. 商业目标