**软件构造总结**

**概念理解：**

**软件构造**：

是编码和调试，部分的详细设计和单元测试，也涉及规划构建、集成和集成测试等活动。

**软件构造的重要性：**

是开发软件的重要组成部分；

在开发软件中处于枢纽地位；

把主要精力集中于构造活动，可以极大地提高程序员的生产效率；

创建活动的产品，源代码，往往是软件的唯一精确描述；

是唯一一项必不可少的（确保会完成的）工作。

**软件工程领域的隐喻：**

软件书写：写代码；

软件播种：生成系统；

软件珍珠培植法：系统积累；

软件创建：建造软件；

应用软件技术：智能工具箱

复合隐喻：组合各个隐喻

**软件构造的前期准备工作：**

是复查性的活动，中心目标是降低风险。（在实现一个系统之前，我们必须清楚这个系统应该做什么以及如何做到这些）

**软件项目的类型：**

商业系统；（高度迭代的开发方式）

使命攸关的系统；

性命攸关的系统；（序列化的方法）

**序列化开发：**

使用瀑布模型，需求相对稳定

**迭代式开发：**

整个开发工作被组织为一系列的迭代过程（需求分析，设计，实现与测试），需求易于变化

问题定义决定了项目的方向

**非功能需求：**

软件产品在功能以外的服务质量方面的需求

**软件架构：**（**两个要素**：元件划分和设计决定）

是构建计算机软件实践的基础；

定义了一张用于描述整个系统各个方面的草图；

建造一个系统所作出的最高层次的、以后难以更改的，商业的和技术的决定

**设计：**

把需求分析和编码调试连在一起的活动

**包括的活动：**用伪码写出一个类接口；编码前画出类的关系图；关于设计模式的选取

**设计中的挑战：**

设计是险恶的问题；（只有先解决或者部分解决这个问题才能真正明确这个问题，然后重新解决）

设计是了无章法的过程；（会犯很多错误，难以判断何时设计“足够好”）

设计需要取舍和调整顺序；

设计会受到很多限制；

设计是不确定的；

设计是一个启发式过程；

设计是自然而然形成的

**管理复杂度：**

是软件的首要技术使命，是软件开发中最为重要的技术话题（技术目标）（软件开发本质性困难的根源在于复杂性）

**管理复杂度的方法：**

（在一个时刻专注于一个特定的部分，减少在任一时间所要考虑的程序量；在软件架构层次上，把整个系统分解为多个子系统来降低问题的复杂度）

把任何人在同一时间需要处理的本质复杂度的量减到最少；

不让偶然性的复杂度无谓的快速的增长

**高设计的理想特征：**

最小复杂度：简单、容易理解；

易于维护；

松散耦合；

可扩展性；

可重用性；

高扇入（让大量的类使用某个给定的类）；

低扇出（让一个类里少量或适中地使用其他的类）；

可移植性；

精简性：没有多余的代码；

层次性：能在任意层次上观察系统；

使用标准技术

**设计的层次：**

软件系统；

分解为子系统或包：

分解为类；

分解为子程序和数据；

子程序的内部

**第2层：分解为子系统或包**

识别出所有的主要子系统；

定义各子系统如何使用其他子系统，即不同子系统之间的**通信规则**；

子系统之间的**交互关系：**

最简单：让一个子系统去调用另一个子系统的子程序；

稍复杂：一个子系统包含另一个子系统的类；

最复杂：一个子系统的类继承自另一个子系统的类

**基本原则：**系统层设计图应该是无环图，即程序中不应该有任何环形关系

**常见的子系统：**业务规则，用户界面，数据库访问，对系统的依赖性

**第3层：类层次**

识别出系统中所有的类；

对子系统进行分解，确保分解出的细节能够用单个类实现，并明确类的接口（当定义类时，也同时定义了这些类与系统其余部分打交道的细节）

**类：**在程序源码中存在的静态事物

**对象：**程序运行期间实际存在的具体实体

**第4层：子程序层**

把每个类细分为子程序

类层次定义了对外接口，这里细化出类的私有子程序（可能会对类的接口进行修改），通常由程序员个人完成

**第5层：子程序内部设计**

为每个子程序布置详细功能；

通常由负责该子程序的开发人员完成；

编写伪码、选择算法、组织代码块、编写代码；

包括子程序内部的设计

**设计的启发式方法：**

软件设计是非确定性的；

启发式“方法”就是不断试错的过程

**设计1：找出现实的对象**

软件设计首选且流行的方法：面向对象设计方法

步骤（非顺序，可重复）：

辨识对象及其属性；

确定可对各个对象进行的操作；

确定对象能对其他对象进行的操作；

区分哪些部分对其他对象可见（public or private）;

确定公开接口（public interface）

**设计2：形成一致的抽象**

目的：忽略细节，在不同层次处理不同细节

基类、接口都是抽象的例子

优秀程序员要在子程序接口、类接口、包接口层次上进行抽象

**设计3：封装实现细节：**

抽象：从较高层次看待对象；

封装：填补抽象留下的空白（能看到的就是全部得到的）；

**设计4：继承**

概念：对大同小异的对象，定义对象之间的相同点和不同点

好处：能很好地辅佐抽象的概念；

简化编程（基本子程序处理基类属性的事项，另外一些子程序处理依赖特定子类的特定操作）；

面向对象编程中最强大的工具之一，需谨慎使用

**设计5：信息隐藏**

结构化设计和面向对象设计的基础（结构化设计的黑盒子概念，面向对象设计的封装、模块化概念）

开发人员把一个地方的设计和实现隐藏起来，使程序的其他部分看不到

是减少重复工作的强大技术（对类而言：类的接口应尽可能少地暴露其内部工作机制）

隐藏复杂度和变化源

信息隐藏同样有助于设计类的公开接口

**设计6：找出容易变化的区域**

目标：把不稳定的区域隔离出来，把变化所带来的影响限制在一个子程序、类或者包的内部；

容易变化的区域包括：

业务规则；对硬件的依赖性；输入和输出；非标准的语言特性；困难的设计和构造区域；状态变量（使用枚举类型取代布尔类型，使用访问器子程序）；数据量的限制（通过具名常量隐藏细节）

**设计7：保持松散耦合**

耦合度：类之间或子程序之间关系的紧密程度

目标：耦合度小就是美，一个模块能够很容易地被其他模块使用

耦合种类：

简单数据参数耦合：模块（类和子程序）间只传递简单数据类型；

简单对象耦合：一个模块实例化一个对象；

对象参数耦合：对象1要求对象2给它对象3

语义耦合：一个模块不仅使用了另一个模块的语法元素，而且还使用了有关那个模块内部工作细节的语义知识。（危险！！！）

**类：**

是一组数据和子程序构成的集合，这些数据和子程序共同拥有一组内聚的、明确定义的职责。

**ADT:**

指一组数据以及对这些数据所进行的操作的集合（像现实世界一样操作实体而不必关注底层实现细节）

**两者区别：**

从数据角度来看没有区别；

从OO角度来看，类还涉及到面向对象的概念

**ADT的益处：** 在类中对应的体现：

隐藏实现细节； 隐藏私有信息；

改动不会影响整个程序； 封装性

让接口提供更多的信息； 真正意义的接口

更易提高性能； 更易于修改

让程序的正确性显而易见； 易于识别错误

程序更具有自我说明性； 可读性高

无需再程序内到处传递数据； 封装数据和操作

可以像现实世界中那样操作实体 更接近人的思维

**ADT和类：**

ADT构成了类概念的基础；

类可以看作是ADT再加上继承和堕胎两个概念

**类接口的设计：**

类设计的第一步，也是最重要的一步；

**类接口设计建议：**

好的抽象：类的接口应能提供一组明显相关的子程序；

好的封装：不要暴露自身数据和实现细节；

二者相关，或者两者都有，或者两者皆失

好的抽象：

类的接口应该尽量展现一致的抽象层次；

每个类应该实现且只实现一个ADT

当实现类多个ADT时，可以考虑重新组织类

理解类接口应该捕获的抽象到底是哪一个；

提供成对的服务(不要盲目创建相反操作)；

把不想关的信息转移到其他类；

让接口可编程，而不是表达语义；

不要添加与接口抽象不一致的公共成员；

同时考虑抽象性和内聚性；

好的封装：

封装比抽象更严格，二者相辅相成（抽象：提供可以忽略实现细节的模型来管理复杂度；封装：阻止你看到细节）

尽可能限制类和成员的可访问性（采用严格且可行的访问级别，最好地保护接口抽象的完整性）；

不要公开暴露成员数据；

避免把私有的实现细节放到类的接口中去；

不要对类的使用者做任何假设；

避免使用友元类（破坏封装，增加复杂度）；

不要因为一个子程序只使用公共子程序，就把它归入公开接口（考虑接口抽象是否一致）；

让阅读代码比编写代码更方便；

警惕从语义上破坏封装性；

针对接口编程；

留意过于紧密的耦合关系