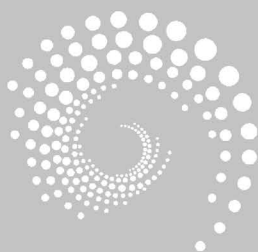


## 软件设计工程

### CHAPTER 5



任何工程从拟定计划到投入使用都要遵循一定的过程,通常包含编制任务书、分析、设计、施工,以及交付使用等阶段,设计工作是其中比较关键的环节。软件设计是软件工程的技术核心,是建模活动的最后一个软件工程行动。通过设计,把计划中有关设计任务的文字资料转换为表达建设实体的全套设计图,为施工构造奠定基础。本章介绍软件设计的基本概念和原则、设计模型和设计过程以及架构设计、设计模式等。

## 5.1 设计工程概述

软件设计会随着新的方法、更好的分析和更广泛的理解而不断变化。虽然大多数软件设计方法学依然缺乏传统工程设计学所具有的渊博、灵活以及定量等特性,但也有自己的方法、质量标准和建模符号。设计阶段的关注点转移到系统划分原则、从软件的概念表示提取功能或数据结构细节、定义软件设计的技术质量标准等方面,与此相关的软件设计概念层出不穷,为软件设计师提供了解决复杂问题的基础思想和原则。本节探讨设计的重要性、设计的要求和步骤以及设计的目标和原则,为理解架构设计、设计模式等建立基础。

### 5.1.1 设计的重要性

任何大中型系统的建设都是一个复杂的物质生产过程,从立项、设计、施工、验收到使用,涉及政策、法规、资金、材料、设备、规划等多方面因素,建设周期随系统的规模、复杂度及条件不同而不同,短则数月,长则数年。因此,在施工之前综合考虑各种因素、划分必要的设计阶段、做出完整的设计方案等,对多快好省地建造系统有着极为重要的作用。

流行的建设过程一般可划分为常规与非常规两大类。常规过程的基本模式是以“建设单位→设计师→承包商”三边关系为基础,基本过程是“设计→发包→建造”。非常规过程有些不同,例如,软件工程发展早期的主程序员开发模式和后来出现的敏捷开发模式等。但不管是哪种类型,模式如何变化,设计工程都居于核心地位。

对于大中型系统的开发,较为基本的建设过程大致可以分为7个阶段。首先是提出项目建议书阶段,即提出拟建项目的轮廓设想,拟订总的要求,做出工作计划,以及在特殊情况下成立筹建机构、聘用设计师等。其次是编制可行性报告阶段,即对建设项目在技术、工程和经济上的合理性进行全面分析、论证和多方案比较,包括研究用户的需要、具备的条件,向建设单位提出评价意见及有关建议以确保项目的实用性、经济性及技术可靠性。第三是项目评估阶段,即在可行性研究报告的基础上对项目进行社会、经济、环境等方面的效益评估。第四是编制设计文件阶段,即按要求进行设计并编制设计文件。第五是施工前准备阶段,即开工前的各项准备工作,包括场地、材料、设备,以及可能的施工招标等。第六是组织施工阶段,即进行项目建设施工(对应软件开发的编程实现)。最后是验收和使用阶段,即进行项目验收,投入使用。在这些阶段中,设计是重中之重,包括大纲设计、方案设计、详细设计、施工设计等工程活动。其中,大纲设计指进一步研究用户的需要以及有关的经济问题,提出总的方案;方案设计指提出总体、外观、构造、施工、概算等全面设计;详细设计指组织各专业咨询师对全部设计及预算做出最后决定;施工设计指组织各专业咨询师编制施工所需要的模型、说明及项目进度表等施工文件及可能的工程招标所需要的工程量表及招标文件。

### 5.1.2 设计的要求和步骤

系统由构件体系、结构体系和基础设施体系构成。其中,构件体系指构成系统的元素,例如构成软件的程序语句、过程、类、组件、服务或子系统;结构体系指构件之间的关系;基础设施体系指对构件及其关系起支撑作用的基本配置,根据系统的重要性的使用性质的



视频讲解



视频讲解

不同而有所相同。因此,系统的设计包括构件设计、结构设计和基本设施配置设计等部分。

设计工程受到多方因素的制约,包括与环境的关系、经济实用、美观易用、技术方案等。第一,任何系统都从属于更大的系统,是总体规划的一个构成部分,要符合总体规划提出的要求。因此在进行设计时,要注意它与环境的关系,例如原有系统的状况、使用者、与其他系统的关系等,即要充分考虑是否符合总体规划的要求。第二,设计工程要满足经济实用性。满足使用功能要求是设计的主要任务,也就是要解决实用性问题。建设工程需要大量人力、物力和资金,在设计和建造中,要因地制宜、就地取材,尽量做到节省劳动力,节约材料和资金,即要考虑具有良好的经济效果。第三,在满足使用要求的同时,还需要考虑人们在美观易用方面的要求,考虑使用者在精神上的感受和使用上的方便性,例如要考虑软件界面的美观要求。第四,应该采用合理的技术措施,包括正确选用构成材料,根据系统特点选择合适的结构、施工方案,使系统易于构造和维护,坚固耐久。这些要求并无顺序关系,要综合考虑,有的要求之间存在冲突,要进行权衡和取舍。

设计工程依据的文件一般包括有关建设任务的要求和总投资等文件、工程设计任务书、委托设计工程项目表等。有的工程经常采用投标方式,委托其他单位进行设计。设计师根据这些文件,通过调查研究,收集必要的原始数据和设计资料,综合考虑总体规划、当前环境、功能要求、框架选项、元素材料、工程经济以及建造技艺等多方面的问题进行设计,绘制设计图,编写主要设计意图的说明书,编制各工种的计算书、说明书以及概算和预算书。这些设计图和相关文件是后续施工的依据。

作为设计师,一般要经历设计前的准备工作、初步设计、技术设计、施工图设计等阶段。在具体着手设计前,要熟悉计划任务书,以明确建设项目的设计要求。计划任务书的内容一般有建设项目总的要求和建造目的的说明、系统的具体使用要求、建设项目的总投资、待建系统的范围和规模描述及调研报告、基础设施方面的要求、设计期限和项目的建设进程要求等。要认真熟悉计划任务书,在设计过程中严格掌握建设标准、功能范围、性能指标等有关约束。必要时可对任务书中的一些内容提出补充或修改意见,但应征得建设单位的同意。由于建设单位主要是从使用要求、建设规模、造价和建设进度等方面提出计划任务,系统的设计和构造还需要收集有关的原始数据和设计资料,在设计前做好调查研究工作。例如,认真调查同类已有系统的实际使用情况,通过分析和总结,对所设计的系统有一定的了解;了解现有构件的种类和规格,掌握新型构件的性能、价格以及获得的可能性;结合使用要求和构件特点,了解并分析不同结构方案的选型,现有施工技术和设备条件;了解组织遗留系统的设计布局、构造经验和使用习惯,结合待建系统的具体情况创造他们喜闻乐见的使用形式。有了充分准备后进入初步设计阶段。

初步设计阶段是系统设计的第一阶段,主要任务是提出设计方案,即在明确的范围内,按照设计要求,综合技术和艺术要求,提出设计方案。初步设计的模型和设计文件有系统架构、概算书,以及说明设计方案的主要意图、主要结构方案及构造特点、主要技术经济指标等的说明书。下一步是初步设计具体化的阶段,即技术设计阶段。

技术设计阶段的主要任务是在初步设计的基础上,进一步确定各设计工种之间的技术问题(对于不太复杂的工程可省去该阶段)。各工种的设计图要标明与具体技术工种有关的详细信息,编制构成部分的技术说明书。例如,结构工种应有系统结构布置方案图并附有相关说明、基础施工工种应提供相应的设施图及其说明书。

施工图设计是系统设计的最后阶段。在这个阶段中,应确定全部工程规模和用料,绘制构件、结构、基础设施等全部施工图,编制工程说明书、结构说明书和预算书。施工图设计的模型和设计文件有总架构、构件、结构、构件连接点,以及各工种相应配套的施工图、说明书、工程预算书等。

### 5.1.3 设计的目标和原则

设计涉及两个世界,一个是人类的欲望世界,一个是实现人类欲望的技术世界。设计师的任务就是尽量把这两个世界结合在一起。例如,人们都希望有一个良好的生活和工作环境,期待建筑物具有稳定、实用和舒适等特性。稳定性指的是不存在影响功能的漏洞,实用性指的是能满足预期的目的,舒适性指的是带给使用者的愉悦感。设计的目标就是获得具有稳定、实用和舒适等特性的模型或表示。为此,设计师需要具备发散思维和综合能力,即先考虑各种可能的备选方案,再从众多的可选方案中选择最合适的。这有赖于凭借个人经验的直觉以及基于建模原则或启发式、质量评价标准、迭代过程等进行的判断。

一个好的设计方案应该具备的特征包括:既实现了需求模型中的所有显式需求也考虑到了利益相关者的所有隐式需求,从实现的角度提供目标物在数据、功能和行为方面的完整描述,便于施工人员和维护人员阅读和理解。这些特性就是设计过程的目标。

要达到这些目标,需要考虑一些准则。较为常用的准则有:使用有辨识度的建造风格或模式创建便于用演化方式实现的架构,使用模块化思想在逻辑上分解系统,明确地表示架构、构件、接口和数据,用便于实现的类来表示数据结构,构件独立且具有良好的设计特征,使用接口以降低连接复杂性,用需求分析所获得的信息驱动设计,使用能有效表达含义的符号,充分考虑现有的实现技术。

设计包括概念、原则和实践。设计工作需要遵循一定的设计原则,设计实践的结果是待建系统的各种表示,这些表示是接下来的施工活动的依据。但在实践之前,应该充分理解与设计相关的各种概念。其中,“分而治之”和“复用”是设计原则的重中之重,与许多设计概念相关。

“分而治之”原则是指一个系统的模块化的过程,即把复杂问题分解为若干便于管理的模块来降低问题求解的难度。模块化概念与系统的抽象性、信息的隐蔽性、功能的独立性等密切相关。系统的抽象性涉及问题抽象的级别,高层次抽象使用面向问题的语言和术语描述解决方案,低层次的抽象使用面向实现的语言和术语描述解决方案。在系统的抽象过程中,人们的第一感觉是分解得到的模块越细越容易理解和实现,即抽象级别越多越好。实际上,总的开发工作量或成本并不会随着模块数量的增多而减少。模块化是一个逐步求精的过程,也是一个由小到大逐步集成的过程。模块数量越多,模块间的联系越复杂,相应的集成工作量或成本也越大。所以,在抽象时,既要避免不足的模块化也要避免过度的模块化。信息的隐蔽性涉及信息封装的级别,可根据用户需求设计为全公开、半公开或不公开。这类似于一台电视机,内部实现被严密封装起来,对外提供各种接口,外部用户可以使用遥控器通过这些接口对电视机进行操作。功能的独立性涉及功能实现的宽泛程度。良好的模块设计应该具有高内聚低耦合的特征。内聚性指的是一个模块内部各功能之间的关联程度,当然是越强越好;耦合性指的是模块之间的关联程度,当然是越低越好。正如一个家庭影院系统,可以分解为电视机和录像机两部分。其中,电视机和录像机各自独立,各自内部的芯



视频讲解

片之间的关联程度很高,一个完成影像解码任务,一个完成影像显示任务;两者之间的关联很弱,把连接线插入两边对应的接口即可。

“复用”原则是指一个模块在若干系统的重复使用过程,旨在提高设计和实现效率。复用分为思想复用和实体复用。侧重思想复用的有架构与设计模式。其中,架构是一个系统的风格和结构规划,设计模式是针对某些经常出现的问题而提出的行之有效的设计解决方案。两者都不是实体,具有抽象和普适性,前者属于战略思想复用,后者属于战术思想复用。侧重实体复用的有工具和框架。在软件行业,诸如 C 语言的函数库、C++ 语言的类库等属于工具复用,Java 领域的 Struts、Spring、Hibernate 等属于框架复用。工具和框架都是实体(例如代码的集合),都提供一些应用接口,但设计理念却截然不同。在软件行业,工具的意义在于使开发人员摆脱底层编码,专注特定问题和业务逻辑,而框架则是一组协同工作的过程或类,利用控制反转机制(由框架代码调用业务代码)实现对各模块的统一调度。没有规矩不成方圆,正如行军打仗的排兵布阵,框架是预先布置的阵,业务代码就是阵中要排的兵。因此,工具因提供“武器装备”而为编程人员带来自由,框架则是在语言语法规则之上再添一层“枷锁”而为编程人员带来约束。对于较为复杂的软件实现,通常的设计原则是:在宏观管理上选取合适的框架以控制整体的结构和流程,在微观实现上利用工具来解决具体的细节问题。

#### 5.1.4 构件设计原则

软件设计不能僵硬、脆弱、固定和黏糊,否则其可维护性就会很差。可复用的构件对软件的可维护性有良好的支持作用。基于面向对象方法的构件由若干的类构成,在设计这类构件时应该遵循一些设计原则,例如开闭、依赖倒转、代换、组合、接口分离、保持距离等基本原则,以及公共封装、公共复用、发布复用等价等打包原则。使用这些原则有利于创建易于变更的设计,降低变更发生时波及的范围。

开闭原则是指“对扩展开放,对修改关闭”,即在对构件内部不做任何改变的情况下拓展系统的功能。这是面向对象可复用设计的第一块基石,即设计的首要目标。“对扩展开放”强调的是“拥抱变化”,使软件具有较强的适应性和灵活性而得以进化;“对修改关闭”强调的是“封装变化”,使进化中的软件具有一定的稳定性和延续性。对一个系统进行分解,越分解越具体。从分解的层次来说,越往上层越抽象,越往下层越具体。发生变化的层次越抽象,向下波及的范围越大。所以通常应该尽量把“可变性”封装在上层以使得系统具有更好的稳定性。因此,实现开闭原则的关键是抽象化以及从抽象化导出具体化。

代换原则是开闭原则的补充,是对实现抽象化的具体步骤的规范。代换原则是指“用派生类代换基类,不影响使用基类的构件”。代换原则是继承复用的基石,强调“任何基类可以出现的地方,其派生类一定可以出现;派生类代换基类时软件功能不受影响,基类才能被真正复用,派生类才能够在基类的基础上增加新的功能”。满足代换原则的设计可以保证使用基类的构件在基类被其派生类代换时保持不变并继续正常工作。该原则要求从基类派生的任何类都必须遵守基类与其使用者之间的约定。例如,构件 A 要使用构件 B,只要与构件 B 的基类约定使用方式即可。由基类派生的任何类只要不违背这个使用约定,就可以随时代换基类并与构件 A 进行交互。当然这种代换只是在构件 B 的内部完成,不会影响构件 A 的使用方式。换句话说,构件 A 并不需要“知道”构件 B 内部的代换情况。

在一个面向对象系统里,类与类之间存在零耦合、具体耦合、抽象耦合等关系。其中,零耦合就是指两者之间没有耦合关系;具体耦合指的是一个类对另一个类的直接引用;抽象耦合指的是两个类通过接口进行交互。显然,抽象耦合具有更好的灵活性。依赖倒转原则是指“依赖于抽象,不要依赖于具体”,强调系统内构件之间关系的灵活性,是达到开闭原则的途径和手段。依赖倒转是针对结构化方法而言的。在结构化方法中,上层模块被分解为下层模块,上层模块调用下层模块,导致抽象依赖于具体,模块之间是具体耦合关系。依赖倒转就是把这种关系“颠倒”过来:从构件内部来看,强调下层的实现依赖于上层的抽象定义;从构件外部使用者角度来看,强调针对接口编程而不要针对实现编程,构件之间是抽象耦合关系。

构件之间的关系除了要强调抽象耦合以实现开闭原则之外,还应该遵循对接口进行分离的原则,以及双方保持一定距离的原则。接口分离原则是指“多个特定客户接口比一个通用接口更好”。通常来说,一个构件对外提供若干操作服务,可以只开设一个统一的接口为所有使用者所共用,也可以针对不同类别的使用者分别提供接口。接口分离原则指的是后者,即根据使用者的类别对接口进行分离,为每类使用者创建相应的专门接口以提供特定的操作服务。保持距离原则也称“最少知识原则”,强调构件之间应该尽可能少“了解”对方的细节,类似于生活中的“不要和陌生人说话”“只与好朋友通信”等处事原则,避免“你跳(海),我也跳(海)”式的相互影响。保持一定的距离可使得系统更加安全和稳定。

构件内部的关系涉及如何打包的问题,也应该遵循一定的原则。例如,“一起变化的类应该放在一起”(称为公共封装原则)“没有一起重用的类不应该放在一起”(称为公共复用原则)“复用的粒度就是发布的粒度”(称为发布复用等价原则)等。满足公共封装原则的构件,其中的类处理相同的功能域或行为域,域一旦发生变化,只修改该构件即可。不满足公共复用原则的构件,其中的类一旦发生变化,整个构件都会重新打包、测试和发布,导致无关类也进行不必要的集成和测试。发布复用等价原则使得版本演进、变更控制和发布管理更为自然和有效。

## 5.2 软件设计过程

在需求工程阶段建立的模型可能有场景模型、行为模型、流程模型、结构模型等,这些模型提供了在设计工程阶段进行架构、构件、界面、数据等设计所需要的信息。软件设计是一个迭代过程,设计的结果会成为构造软件的“蓝图”。这些“蓝图”从高度抽象的总体设计开始,经过多次迭代,产生更低抽象的详细设计。本节介绍与软件设计过程相关的概念。

### 5.2.1 从需求到设计

在需求工程阶段,会开展类似“瞎子摸象”“庖丁解牛”等活动。“象”“牛”等实体是待开发的软件或拟建设的系统。“瞎子摸象”就是从实体外部感知实体的界面、观察实体的行为、了解实体所具有的功能,活动的主体主要是客户方的利益相关者。“庖丁解牛”就是从实体内部查看实体的构成和结构、分析实体功能的实现流程,活动的主体主要是开发方的利益相关者。这些活动所产生的视图就是需求模型,包括在“瞎子摸象”活动中描绘的场景模型、行



视频讲解

为模型,以及在“庖丁解牛”活动中描绘的流程模型和结构模型。其中,场景模型是系统“参与者”的功能视图;行为模型是系统对外部“事件”的响应视图;流程模型是数据变换视图,描述了系统具有的功能是如何实现的;结构模型是系统的构成视图,描述了构成系统的元素及元素之间的关系。

另外要注意的是数据模型和类模型。其中,数据模型表示的是问题的信息域,它是其他模型的基础;类模型表示的是“类”粒度的模块以及类的协作方式等,“类”封装的是编程级别的数据结构和算法,抽象程度较低,易于用编程语言实现。这两种模型都很重要。

设计建模建立在需求模型的基础上,包括架构、界面、构件和数据等设计。需求模型主要关注“做什么”,设计模型主要关注“怎么做”。但是,两者并不是截然分开的。系统处理什么东西、执行什么功能、表现什么行为,系统有什么界面、什么交互、什么约束,这些是需求工程阶段要回答的问题。但是,正如“瞎子摸象”,客户也许并不明了系统在某些方面需要什么,开发者也可能并不确定所采用的方法是否适合完成客户说明的功能或性能。在需求和设计这两个重要的工程之间明确地划分分析和设计任务显然是不现实的。有的设计会作为分析的一部分出现,有的分析也会在设计过程中进行。因此,经常会采用迭代的方式进行分析和设计。

需求模型要实现的目标包括描述客户需要什么、定义一组在构建后可以验证的需求、为设计工程打基础。因此,可以把需求工程活动划分为需求获取、需求分析、系统分析等活动。其中,需求获取活动主要收集资料;需求分析活动面向客户,分析人的“需要”,专注“要什么”;系统分析活动面向系统,分析系统的“需求”,要考虑“做什么”。对于设计工程活动,也可以划分为系统分析、总体设计、详细设计三大活动。其中,系统分析活动就是需求工程中的系统分析活动,要考虑“怎么做”;总体设计活动对系统进行高层次的抽象;详细设计活动对系统进行低层次的抽象。经过这样的划分,显然系统分析介于两大工程之间,既要分析做什么,也要设计怎么做。分析模型是系统级描述和设计级描述之间的桥梁。其中,系统级描述是指用软硬件、数据、人及其他系统元素等对整个系统或业务功能进行的描述;设计级描述是指对应用程序的架构、用户界面、构件结构进行的描述。因此,系统分析人员可以采用迭代的方法对已知内容建模,这些模型作为增量设计的基础,经过增量式迭代,逐步完成全部设计。

需求模型的所有元素都可以直接成为设计模型的某些部分,它们的关系如图 5-1 所示。

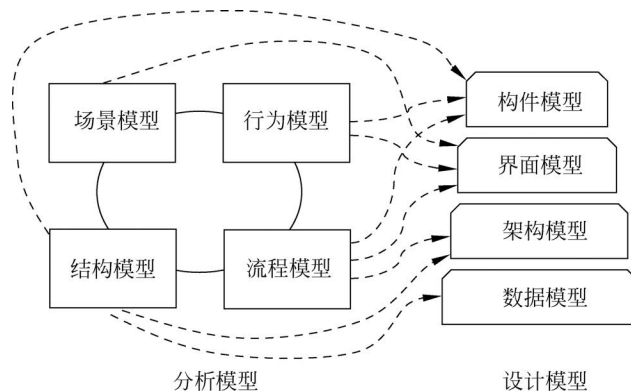


图 5-1 从分析模型到设计模型

软件开发方法有很多,都有着各自的特点和适用场合。例如,结构化方法具有功能模块化、自顶向下逐步求精等特点,强调使用数据流或数据结构来定义设计;面向对象方法具有类型化、自底向上逐步集成等特点,强调从分析类等概念模型自然导出设计;基于组件的开发方法的组件模块比类模块粒度大,面向服务的开发方法的服务模块比组件粒度大,粒度越大越适合更高层次的抽象设计。不过,这些方法都有一些共同的特征,如将需求模型转换为设计表示、表示构件及构件之间的接口、分解和细化、评估质量等。因此,在开发过程中适当对这些方法进行组合以适应实际开发,组合示例如表 5-1 所示。其中,组合 6 使用的都是面向对象的术语,不存在术语转换问题,各阶段的过渡和迭代较为自然,是当前较为常用的软件开发方法。

表 5-1 开发方法组合

阶段	组合 1	组合 2	组合 3	组合 4	组合 5	组合 6
分析	SA	SA	OOA	OOA	OOA	OOA
设计	SD	OOD	SD	OOD	OOD	OOD
实现	OOP	OOP	3GL/4GL	3GL/4GL	SP&OOP	OOP

注:字母 S 表示结构化方法,OO 表示面向对象方法,A 表示分析,D 表示设计,P 表示编程,3GL/4GL 表示第三代和第四代编程语言。

如果采用的是结构化方法,数据设计是将 E-R 图等概念模型转换为物理模型的过程,架构设计是把 DFD 等流程模型转化为架构或结构模型的过程,构件设计是将状态图等模型转换为过程或函数的过程。如果采用的是面向对象方法,数据设计是将分析类等概念模型转换为设计类的过程,架构设计是将协作图、活动图等模型转换为架构或结构模型的过程,构件设计是将状态图、时序图等模型转换为组件的过程。界面设计是对系统与外界的关系进行设计的过程,包括系统与其他系统如何通信、用户与系统如何交互等。图 5-2 从抽象和过程两个视角对采用面向对象方法建立的模型进行观察。其中,抽象维表示详细程度,过程维表示工作阶段。

## 5.2.2 从抽象到具体

设计包括架构设计、构件设计、界面设计及数据设计。这些设计都是一个从抽象逐步到具体的过程。

作为设计师,“架构”是一个绕不开的概念,因为设计基本上就是围绕它展开的。这个术语来自英文单词“architecture”。它有许多含义,可以译为建筑学、建筑式样、建筑风格、体系结构、结构、架构等。从字典解释来看,它有三个层次的含义。首先,它表示规划、设计和建造建筑物的技艺(art),即建筑学或建筑术;其次,它表示建筑物的风格(style),即建筑式样或建筑风格,如现代建筑或摩洛哥建筑;最后,它表示事物的结构(structure),即事物由部件构成,这些部件以一种有序的方式连接起来,如知识结构、肌肉纤维结构等。

在我国软件行业,一般将“architecture”译为“架构”或“体系结构”,将“structure”译为“结构”。因此,将设计划分为较为宏观的“架构设计”和较为微观的“结构设计”。本质上,架构设计包含了结构设计,即架构设计涉及风格设计和结构设计。风格涉及样式样貌,结构涉及分解和细化。高层结构对应行业术语“架构”,细化后的结构对应行业术语“结构”。结构分解的层数视问题复杂度而定,哪些层属于“架构”范畴,哪些层属于“结构”范畴,并没有明



视频讲解

确的界限。因此,有时也没必要把这两个术语截然区分开来。架构设计包括风格设计和结构设计,而结构是分层的,每一层的构件都是上一次构件的分解。例如,系统可以分解为子系统,子系统可以分解为服务,服务可以分解为组件,组件可以分解为类,类可以分解为过程,过程可以分解为语句块等,构件的粒度随着分解的进行越来越小,直至可以直接编程实现。

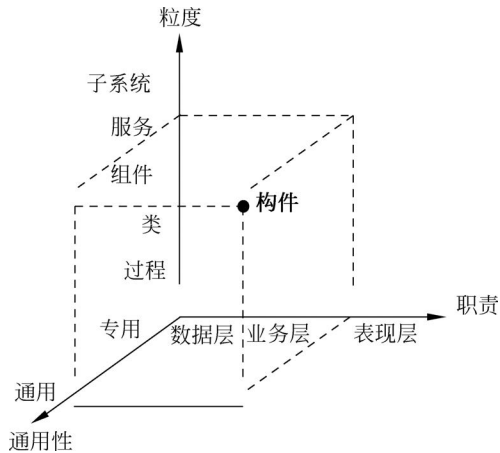


图 5-2 架构设计的维度

对于结构中的构件,既可以按粒度进行分类,也可以按职责、通用性等进行分类,如图 5-2 所示。

构件按粒度从小到大可划分为过程、类、组件、服务、子系统等,按通用性从专用到通用可划分为若干级别,按职责从数据到表现可划分为若干层次。在实践中,人们归纳提炼了很多实用的实现技术,例如,对类进行组合的设计模式和组件技术、面向服务的架构(简称 SOA)、包含基础设施的框架技术等。

以人体系统为例,可把人体分解为运动、神经、内分泌、循环、呼吸、消化、泌尿、生殖等子系统。这些子系统协调配合,完成各种复杂的生命活动。

这些子系统及其关系就是人体高度抽象的架构。这些子系统进一步分解为各种器官。例如直观的眼、耳、鼻、舌、身等感觉器官,内在的肝、心、脾、肺、肾、胆、小肠、胃、大肠、膀胱等脏腑器官。这些器官及其关系是子系统级抽象的架构。这个分解过程还可以继续进行。例如,人们把由多种组织构成的能行使一定功能的结构单位称为器官,器官的组织结构特点与它的功能相适应。也就是说,器官可分解为组织。而组织则是介于细胞和器官之间的细胞架构,由许多形态相似的细胞及细胞间质所构成,有上皮组织、结缔组织、神经组织、肌肉组织之分。可见,组织又可分解为细胞。组织及其关系、细胞及其关系已是人体较为具体的架构了。从各层架构的构件粒度维看,如果人体子系统对应软件子系统,那么,器官可以与服务对应,组织可以与组件对应,细胞可以与类对应。

构件设计指的是对构成程序的基本构件的内部细节进行描述。这与开发方法有关。例如,面向服务的开发方法的基本构件是服务,基于组件的开发方法的基本构件是组织,面向对象开发方法的基本构件是类。构件设计定义了数据的结构和处理过程。在类这一级,就是定义构成类的的数据域和数据处理算法。以人体细胞为例,细胞是人体结构和生理功能的基本单位,是生长和发育的基础。人体细胞形态多样,大小各异,但其结构基本相同。细胞一般由细胞膜、细胞质和细胞核构成。细胞膜是包围在细胞最外面的一层薄膜,将细胞与外界环境隔开,使细胞具有相对独立和稳定的内环境。通过细胞膜,细胞与环境之间可以进行物质运输、能量转换及信号传导。细胞核由核膜、核仁、染色质和核基质构成,是调节细胞生命活动、控制分裂、分化、遗传、变异的控制中心,在细胞的代谢、生长、发育、繁殖和分化中起着重要作用。细胞质是存在于细胞膜和细胞核之间的物质,包括基质、细胞器和包含物等,是细胞进行物质代谢的场所。从面向对象开发方法的角度看,细胞质与类构件的数据结构对应,细胞核与类构件的算法集合对应,细胞膜体现了类构件的封装性。

界面设计描述信息如何进入和流出系统,以及如何在构成架构的构件之间通信。前者是外部界面设计,例如用户界面,与其他系统、设备、网络及其他信息生产者或消费者的接口等。后者是内部接口设计,也就是各构件之间的接口。例如,人体的眼、耳、鼻、口等是人体的外部界面,细胞膜是细胞之间的内部接口,这些界面或接口用于物质运输、能量转换或信号传导等。

数据设计也称为数据架构,数据的架构是软件设计的重要组成部分。数据设计也是先从客户和用户的视角创建一个高度抽象的数据或信息模型,再逐步细化成能被计算机系统处理的更加具体化的实现表示。对于信息系统类的应用程序,数据的架构对处理它的软件架构会产生很大的影响。例如,从业务的角度看,采集各数据库中存储的数据并重组为数据仓库以促进数据挖掘或知识发现,会对业务的成功产生影响;从系统的角度看,将需求工程导出的数据模型转换为数据库,会对业务目标的实现产生影响;从构件的角度看,数据结构和算法的设计对高质量系统的构建会产生影响。

在从需求向设计的转换过程中,采用面向对象方法,由于使用同一套术语,各阶段模型之间的转换相对比较容易。有时,结构化方法或组合方法会因各阶段使用的术语差异会让人困惑。请回顾第三章的DFD模型到软件结构模型的转化、E-R模型到数据模型转换实例,结合本节的内容进一步理解需求工程与设计工程之间的内在逻辑联系,体会需求工程对设计工程的导向作用。

## 5.3 软件的分合与框架

网络经济时代,信息量非常庞大,快速反应是反映企业竞争力的关键因素。为适应市场的变化,企业的分工与合作机制也会追踪改变,支撑企业提升竞争力的软件系统同样需要随之调整。在软件设计时应该充分考虑这种变化,理解领域概念的稳定性和业务过程的易变性,以领域知识架构主导分解过程,以业务流程引导集成过程。软件的“分”以知识架构作为依据,从领域概念设计软件构件;软件的“合”以业务流程为依据,把软件构件组合为各种应用服务。以这种模式设计软件系统,可以增强企业的快速反应能力。本节介绍软件的分合原则。

### 5.3.1 模块与组件

各行各业都存在一些易于引起混淆的概念,软件技术领域同样如此。人们在描述一个概念术语时,有时是泛指,有时是特指,这两种情况可能有很大的不同。例如,对于一个事物的构成部分,可以简称为构件(构成部分)、组件(组成部件)、元件、元素、模块等。从泛指的角度,这些术语的含义是一样的,经常互换使用。但是,从特指的角度,有时差异会非常大。对于软件的构成部分来说,结构化方法用“过程”特指功能,用“模块”特指相关“过程”的集合;面向对象方法用“类”特指字段(数据域)和方法(数据操控,类似于结构化方法的“过程”)的封装体,用“包”特指相关“类”的集合;基于组件的方法用“组件”特指软件的构成部件,面向服务的方法则用“服务”特指软件的构成部件。因此,从特指的角度,可以说“模块化”方法代表传统,“组件化”方法代表现代。



视频讲解

模块译自英文单词“module”,组件译自英文单词“component”。两者看上去颇为接近,其实不然,后者体现了更好的稳定性、灵活性和多样性。用组件构建的软件能节省大量的管理和维护成本。因为涉及范式(思维习惯)的变迁,一个模块化高手转到组件化的门槛很高,软件团队整体转型更是工程浩大,项目管理、系统分析与设计、编程、测试等各阶段都存在很大的差异,不可等闲视之。

软件的目标是对企业的管理制度形成支撑作用,因此软件开发技术会随着企业的管理制度而演变。20世纪20年代到80年代,主流管理制度是面向任务的层级式组织模式,任务导向的软件架构是主流,如财会、采购等系统都是相互独立的应用程序。80年代到90年代中期的主流管理制度是面向过程的扁平式组织模式,进行业务过程再造(BPR),任务导向的应用程序难以促进过程的顺畅而无能为力,软件设计也就逐渐转向了过程思维,模块化方法迅速普及开来。90年代中期到21世纪初的主流管理制度是面向网络和知识工作者的价值网络组织模式,强调快速组装、量身定做,过程导向的构成应用程序的模块因灵活性不足而难以支持快速的流程变迁和产品组装。软件设计随即转向组件思维,知识工作者和组件融为一体(知识工作者是企业里的组件,组件是软件里的知识工作者),组件可快速组装以支持知识工作者,知识工作者可快速组合以支持过程,为顾客提供快速反应和服务。

模块和组件本质相同,模块化和组件化的目的也差不多,但基本思维却相去甚远。模块和组件本质上都是一些数据和对数据进行处理的数据的集合体。模块化和组件化也都采用“治大国如烹小鲜”式的“分而治之”思想来化解软件固有的复杂性。但是,两者背后的理念几乎没有交集,“分而治之”的“手艺”也并不相同:用传统“手艺”切分出来的软件部件被称为功能模块(简称模块),用新“手艺”切分出来的软件部件被称为组件模块(简称组件)。组件化思维类似“庖丁解牛”,依循牛的架构而游刃有余,模块化思维则不然,切块较为随意。由此可知,作为系统分析师,既要确定企业工作流程及用户怎样使用系统,也要像“庖丁”那样解析企业和软件的架构,依循架构切出组件。组件化的软件稳定而又不失灵活性,能快速组装以支持多样化的服务,边际成本也低,促成组件化思维成为软件项目成功的关键。



视频讲解

### 5.3.2 三位一体

进行模块化思维的系统分析师专注于业务部门的客户需求而较少关注生产部门需要的部件,模块由开发者随心所欲的切分而得(从上至下把应用程序分割为树状的功能模块),进而使用这些模块组合并装配出多样化的应用程序。这种思维把业务流程作为企业的架构,从流程来切分应用程序,再从应用程序切分组件。但是,流程是经常变化的,从不稳定的流程切分出模块会导致模块不稳定,用不稳定的模块是难以组合出稳定的软件系统的。显然,如果“分”得不好,“合”就难快,规格和品质也会经常出现问题,从而导致维护成本的增加。

传统企业的目标是批量生产,基本组件是部门。现代企业的目标是快速量身定做,组件是知识工作者。目标变了,支持企业组件的切分方法也就要改变。改变切分方法的目的是更容易组合以达成企业目标。企业分合方式的改变促使软件的分合的改变,即依循流程分出模块要转变到根据架构切分组件、根据流程组合组件。也就是说,组件化思维的核心是:企业领域架构主导“分”,企业业务流程引导“合”。因此,它与模块化思维的本质区别是:模块化是把流程切成模块,组件化是组件组合以支持流程的变化。“分”得好才能“合”得好,既能缩短上市时间,也能提高组件的复用性。把客户需求当成“合”的引导,就不会在刚获得需

求或流程时就想着“分”，就可能发现一些意想不到的组合搭配，大幅降低成本、改善品质、提升经济效益。

软件是用户需要的由开发人员组合装配出来的产品。软件“合”的目标由用户决定，软件的“分”及如何“合”则是由开发者决定。由开发者决定组件的组织，组织不受使用者的限制，组件的组织就有了调整空间，使得组件化软件具有很高的灵活性。

企业领域架构就是领域知识架构，领域知识包含许多基本概念，这些基本概念是知识架构的组件。概念是稳定的，是企业的基因。用这些稳定的概念作为软件系统的基因就是组件化思维的出发点。企业领域知识架构作为“分”的依据，概念一一对应程序组件和数据实体，使得企业、数据和程序因相同的基因而具有高度的一致性和统一性，达成三位一体，如图 5-3 所示。

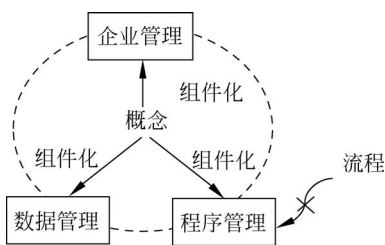


图 5-3 组件化思维的一致性和统一性

概念代表一群实体，是抽象的类别，具有符号、内涵、外延等要素，内涵清晰而稳定。从企业管理的角度看，它是人们日常沟通的基础，具有相同概念易于达成共识。

从数据管理的角度看，数据的切分一直是根据知识领域的概念进行的，也具有一致、清晰、稳定等特性。在程序管理方面，模块化是根据流程等进行切分的，各模块切分与命名因人而异，难以达成共识，导致模块难以复用。用概念来切分具有高度共识且易于沟通、复用和维护。程序架构与企业知识工作者的概念架构相同还会促进知识的分享，组件化的软件更易于支持企业的独立、连结、共享与组合。

寻找领域知识概念就是找出软件系统的组件。例如，快餐企业有薯条、特餐、订餐、顾客、职工、分店等概念，对应软件系统，就会有相同类别的组件。需要指出的是，切分组件的目的是支持企业的工作流程，流程顺畅性是检验切分质量的基础。软件架构只要符合企业领域架构，就能保证装配出高品质的软件。当然，领域知识架构虽然稳定，但也会存在变化，导致软件复杂性的增加。在进行领域分析、组件化、模式创造时也要考虑这种复杂性，“分”的过程多些艺术感和文化性，“合”的过程尽量工程化和自动化，尽量使切分出的组件及其关系可以随着领域架构或流程的改变而能稳定地调整，对各种创新性流程形成有力的支撑。

### 5.3.3 组件框架

当前，软件业界最为流行的开发方法是用 OOAD 方法开发基于组件或组件式的系统。采用组件化技术的系统具有很强的竞争力。

首先，组件化方法能大幅降低系统的成本，如图 5-4 所示。传统的非分布式模块化系统的复杂度不高使得其成本不高，分布式模块化系统因为复杂度较高而导致成本居高不下。基于组件的分布式系统的复杂度降低，成本大幅下降。

其次，应用系统具有高度灵活性的关键因素包括组件化而不是模块化、逐步演进而不是大规模再造、动态配置而不是静态配置系统、多策略外包和伙伴关系管理等。其中，组件化是最为关键性的因素，组件式系统具有极高的灵活性；利用动态配置可以按需装配系统，支持产品的多样化。

在激烈的市场竞争中能有效降低成本、增强灵活性的还有企业组件框架。企业组件框



视频讲解

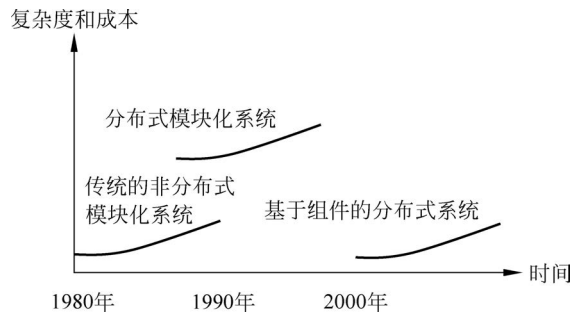


图 5-4 软件的成本

架是软件组件化之后的必然产物。这是因为同行业领域的软件系统有相当部分的架构是一致而稳定的,其他部分是易变而多样化的。稳定部分的架构只需设计一次,留下稳定的接口,以便容纳进易变部分进行组合并提供多样化服务。组件框架思想源于应用框架,两者的基本设计理念和技术是一致的。但应用框架注重于应用程序的快速开发,组件框架则着重于可复用业务组件的开发。

组件框架由稳定的组件组成,提供“插座”(接口)以“插入”为用户量身定做的可变组件。如果说组件是树叶,框架就是树干;组件是水,框架就是杯子。组件框架是应用系统的基础设施,提供稳定的基础架构,具有高度的灵活性与延展性,支持多样化的应用系统。

组件化是基于变与不变的思维,组件框架是组件化思维技术下的架构设计产品。设计组件框架的基本思路是:分清稳定与可变的边界,分为两类不同的组件,稳定的组件具有一致性,可变的组件具有特殊性,两者通过接口连接,可根据用户的多样性需求组合成各种服务或产品。组件框架能与多样化的可变组件搭配以满足各种用户的需要,因此组件框架具有很好的复用性,经济价值很高。总之,多样化的高品质组件能有效降低生产成本,稳定的框架能有效降低管理成本,框架接口提供了容纳其他组件的弹性空间,组件易于更换,使得整个系统能快速地升级换代,用户更容易享受到高品质的服务。

从框架的建构技术来看,组件框架可分为基于类继承的白盒框架、基于对象组合和委托的黑盒框架两类,各有优缺点:前者容易开发,但使用者要了解其内部结构和操作才能设计出适当的子类,后者难以开发,但是易于使用;前者依赖编程语言的继承机制来容纳多样性的类,框架与多样性的类之间的依赖性较高,后者依赖接口容纳多样性的组件,框架与多样性组件之间的独立性较高。应用框架大多属于白盒框架。组件框架大多属于黑盒框架。这是因为在网络环境下,组件一般是分布的,基于接口的黑盒框架独立性较高,灵活性大,更适合于这样的开放和分布式环境。无论是开发组件框架还是购买组件框架,都需要搞清楚需要哪种类型的组件框架。

## 5.4 架构风格

架构设计的特征之一是系统组织的惯用模式的使用。系统组织的惯用模式也就是架构风格。在实践过程中,设计师发现一些特定的组织原则和软件结构的价值,开发了不少的架构风格。本节介绍一些架构风格,旨在展示架构选择的丰富性以及如何选择。

### 5.4.1 经典架构风格

在工程实践中,设计风格和模式的使用都非常普遍。设计风格包括设计术语、结构模式、处理模型、不变项、使用样例、优缺点、具体化等。结构模式是对风格进行结构化描述。一个特定系统的结构由各个部分构成,这些部分以一定的方式连接在一起。架构风格依据一种结构化组织的模式定义一类系统。具体来说,就是定义构件和连件的类型的术语和一组如何构成的约束。很多风格还有语义模型。这些模型用于说明如何从系统的构件属性决定系统的总体属性。

软件也有其组织风格。常见的架构风格如表 5-2 所示。有的风格与开发方法有关,如面向对象组织、数据流组织等。有的风格与系统类型有关,如编译器的传统组织、ISO 的 OSI 模型等。

表 5-2 常见的软件架构风格

类 别	风 格
数据流系统	批处理、管道-过滤器等
调用-返回系统	主程序-子例程、面向对象系统、分层系统
独立组件	通信处理、事件系统等
虚拟机	解释器、基于规则的系统等
仓储式系统	数据库、超文本系统、黑板等

数据流系统类的风格关注数据的流动过程。其中,管道-过滤器风格类似于自来水厂对水的处理过程。在这个过程中,水在管道中流动,在水池进行杂质过滤处理,如图 5-5 所示。

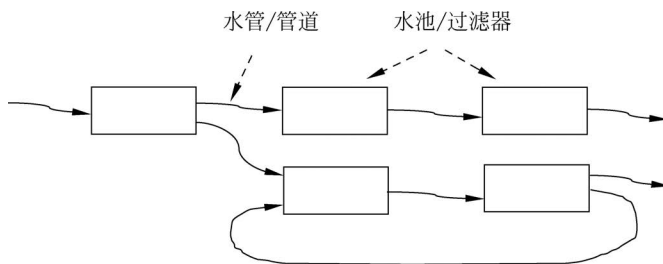


图 5-5 管道-过滤器组织

管道-过滤器风格中的管道类似于水管,过滤器类似于水池。每个过滤器有输入端和输出端。过滤器从其输入端读取数据流,对输入流做变换,在输出端产生数据流。管道连接过滤器,把一个过滤器的输出端流出的数据传输到另一个过滤器的输入端。这种风格的主要特征,一是过滤器必须是独立的实体,即一个过滤器不应该与其他过滤器共享状态,二是过滤器不知道其上游和下游过滤器的情况。在规格说明中,可以对管道中的数据流加以限制,但可以不用了解管道两端的过滤器的情况。进一步讲,管道-过滤器网络输出的正确性不依赖于过滤器的顺序。较为常见的具体化管道-过滤器风格是管线。在管线架构中,过滤器的拓扑结构是线性序列,管道中驻留的数据流量有限制,两个过滤器之间流动的数据的类型要进行清晰的定义。当每个过滤器作为单个实体处理其所有的输入数据时,管线风格就退化为批处理系统了。

调用-返回系统类的风格关注构件之间的互动过程。例如,主程序-子例程就是最为传

统的软件风格之一。这种风格的系统围绕一个主程序和若干子例程进行组织。其中,主程序以控制循环的方式按序驱动各子例程。面向对象的组织风格是基于数据抽象的,即把数据表示和数据操作封装在抽象数据类型(ADT)中。这种风格的构件是抽象数据类型的实例,也就是对象。对象因为负责保存资源可视为管理者,对象之间通过过程调用实现交互,如图 5-6 所示。面向对象组织风格的主要特征是由对象保存数据、隐藏数据(其他对象看不见)。

分层系统也属于调用-返回系统类的风格。这种风格按层级组织系统,每层使用下一层的服务并为其上一层提供服务。除了相邻的层级,每层对其他层都是不可见的。每层的下一层可以视为一个虚拟机(构件)。各层之间的连件用协议来定义,协议确定了层级之间的互动。分层风格的拓扑结构如图 5-7 所示。这种风格的典型例子是 ISO 的 OSI 通信协议模型。在该协议中,每层按一定的抽象级别为通信提供一个子层,各层定义了相应层次的交互,最底层的交互通常是硬件连接。

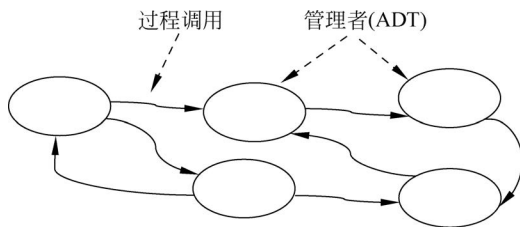


图 5-6 面向对象组织

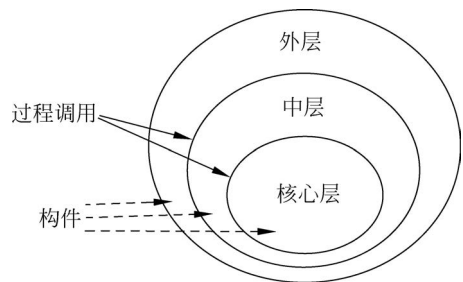


图 5-7 分层组织

独立组件类组织风格强调系统构件的独立性。在这类系统中,构件接口提供有过程的集合,一般是以明显的方式调用(简称显示调用)接口中的过程。事件系统风格则是以隐晦的方式调用(简称隐式调用)构件接口中的过程,还具有反应式集成、选择式广播等特征。这种风格源于那些基于参与者、满足约束、守护进程和包交换网的系统。简而言之,它类似于生活中的杂志订阅系统,读者就是参与者,要订杂志(约束),杂志社的工作就是守护进程,杂志刊印后(事件),通过邮局等网络(包交换网)分发给订阅者(满足约束)。事件发生后对事件的响应就是反应式集成,只分发给订阅者就是选择式广播。也就是说,在一个事件系统中,有的构件可以宣布或广播一些事件,其他构件可以注册(订阅)感兴趣的事件。某事件一旦发生,系统自己会调用所有注册了该事件的构件的过程,相当于把事件广播给感兴趣的那些构件。由此可知,这不是构件之间的直接调用,而是系统以广播的形式调用,所以称为隐式调用。事件处理模型如图 5-8 所示。

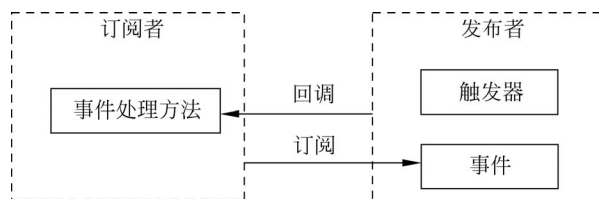


图 5-8 事件系统组织

虚拟机类的组织指的是把一个系统看成一个“虚拟”的机器。例如,操作系统、微软的公共语言运行环境(CLR)等都可以视为虚拟机,如图 5-9 所示。对于工作在操作系统一层的人或其他软件系统来说,操作系统就是虚拟机;对于用 C# 等 .NET 语言编写的应用程序来说,它们运行在 CLR 上,CLR 相当于操作系统之上的又一层虚拟机。CLR 对程序执行的细节进行了包装,程序员无须关注程序的执行环境,只需专注于程序的业务逻辑和功能流程,从而提高开发效率。

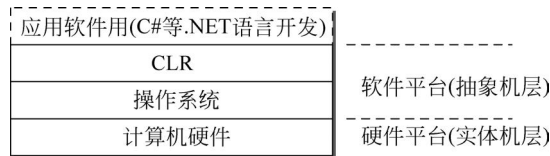


图 5-9 基于 CLR 的主机计算环境

广义而言,任何能为上层提供服务的软件平台都可以视为虚拟机,解释器风格就属于虚拟机类的组织方式。解释器由解释引擎、解释器状态、程序和程序状态 4 种构件构成,如图 5-10 所示。其中,解释引擎负责做解释工作;解释器状态表示解释引擎的内部控制状态;程序是需要解释的伪代码;程序状态表示被解释的程序的当前状态。解释器通常用于构建虚拟机,以弥合程序的语义要求的计算引擎与可用的硬件计算引擎之间的差距。

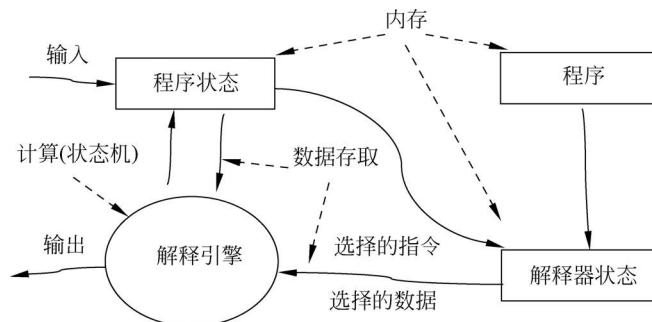


图 5-10 解释器组织

仓储式系统的典型特征是以数据为中心。它有两类明显的构件,一是表示当前状态的中心数据结构,二是对中心数据存储进行操作的独立构件集合。黑板就是一类仓储式组织的风格,如图 5-11 所示。

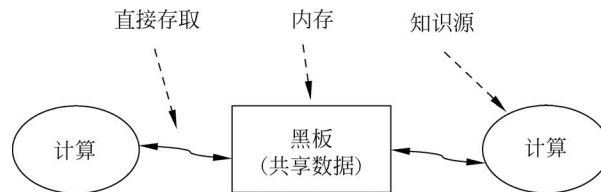


图 5-11 黑板组织

黑板风格由知识源、黑板数据结构和控制三部分构成。其中,知识源指可以直接存取黑板共享数据的独立计算构件;黑板数据结构指的是中心数据,包括解决问题的状态数据等,可被知识源改变;控制指的是由黑板状态直接驱动知识源响应。这种风格本质上就是传统的基于黑板的课堂授课模式,教师、学生等就是知识源,黑板由这些共享,黑板状态会随着教

师的讲解和学生的提问而变化,也会对学生接受知识产生直接影响(驱动学生学习)。



视频讲解

## 5.4.2 经典架构风格的应用

假设要开发一个索引系统,该系统要求接收的所有的数据都按关键字排好序,即所有的行、每行的单词、每个单词的字母都是字母表顺序的有序集合。

从软件架构的角度考虑,不同的问题分解方法在应变能力方面存在较大的差异。在设计时应充分考虑各种可能存在的变化。例如,在排序方面,既可以一边读取一边排序,也可以全部读完后排序。这是属于处理算法方面的变化。再如,行、单词和字符可以按各种方式存储,可以显式或隐式地存储循环位移的结果。这里,循环移位指的是在对每行进行处理时把行首单词移动到行尾,隐式存储指的是按“索引-偏移量”对进行存储。这是属于数据表示方面的变化。还应该考虑的变化有:是否消除虚词、系统交互方式、是否允许用户进行删除操作等增强系统功能的变化,空间、时间等性能的变化,组件的复用性等。

根据一个索引系统的基本要求,可以把问题分解为输入、移位、排序、输出等基本功能。其中,输入是指从输入媒介接收字符序列;移位是循环移位;排序是按字母顺序排列;输出是向输出媒介输出排序后的结果。

管道-过滤器解决方案如图 5-12 所示。



图 5-12 管道-过滤器解决方案

该解决方案使用的是管线方法。其中,输入、移位、排序和输出是过滤器,每个过滤器处理数据并发送到下一个过滤器。不管什么时候,只要有需要计算的数据,过滤器都可以运行。过滤器之间的数据共享就是管道上传输的数据。这种处理流程非常直观,每个过滤器都可以独立地运行,可以很方便地在系统中增加和修改。但是,这种解决方案的交互性较差,空间使用效率也较低。

可以使用一种基于共享数据的主程序-子例程风格来解决这个问题,如图 5-13 所示。

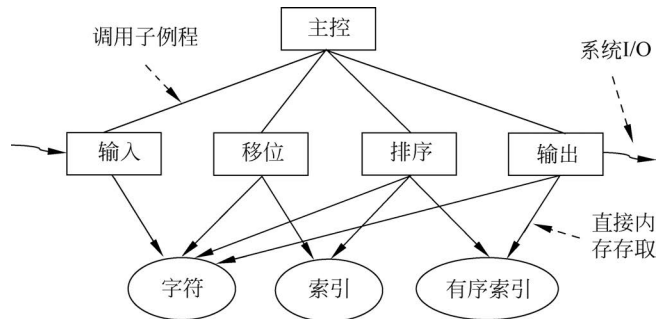


图 5-13 基于共享数据的主程序-子例程解决方案

该解决方案由一个主控程序、四个子例程(输入、移位、排序、输出)和三个共享数据区(字符、索引、有序索引)构成。主控程序协调各子例程,即主程序依次调用它们进行排序。子例程之间没有直接关系,而是通过共享存储通信,即子例程直接存取内存。由于有主程序协调,子例程与共享数据之间的通信无须施加限制,数据可以进行有效的表示。但也要注意

到它的应变能力的不足。例如,如果数据存储格式发生变化会影响到所有访问它的模块,这种分解对复用性的支持度也不高。

也可以使用 ADT 风格来解决这个问题,如图 5-14 所示。

这种解决方案把系统分解为输入、字符、循环移位、按字母移位、输出等构件,每个构件提供了一系列接口,允许其他组件通过调用该接口中的过程来访问其数据,数据不再直接共享。看上去,ADT 解决方案与主程序-子例程对处理模块的逻辑分解相同,但它在设计上的应变能力却比后者更强。例如,数据表示和处理算法的改变是在单个构件中进行而不影响其他构件,它对复用性的支持度也更高。当然,该方案也有一定的局限性。例如,要增加新功能,要么修改已有构件,要么添加新的构件。前者影响现有构件的简单性和完整性,后者可能导致性能下降。

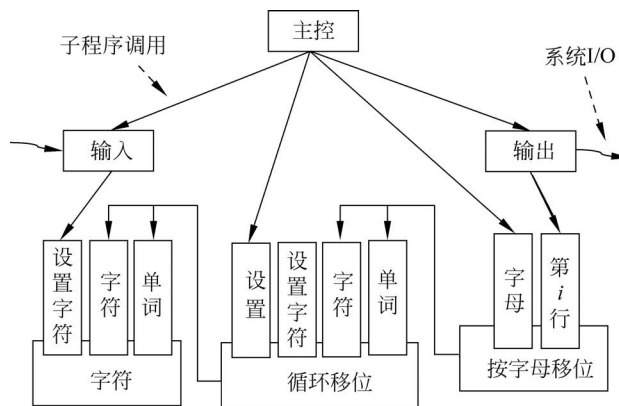


图 5-14 ADT 解决方案

最后再来看一种基于共享数据的隐式调用风格解决方案,如图 5-15 所示。

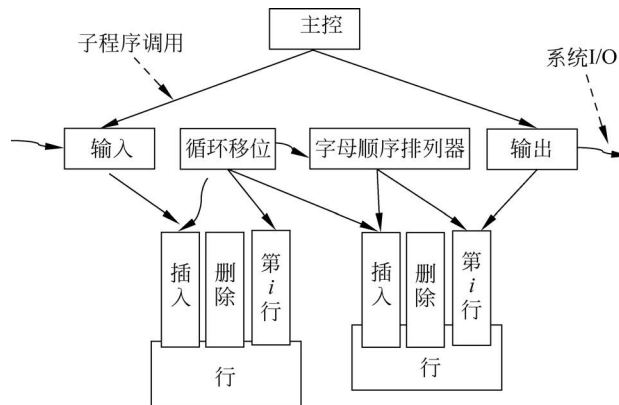


图 5-15 隐式调用解决方案

这种解决方案与共享数据的主程序-子例程方案类似,但数据抽象级别更高,修改数据时的调用方式也不一样。首先,它以诸如列表或集合等形式抽象地访问数据而不会为其他构件公开其存储格式,所以其存储数据的接口更加抽象。其次,它基于“活动数据”模型对数据进行修改。例如,增加新行到行存储的动作使得一个事件被发送到循环移位构件,该事件导致循环移位,即隐式调用移位功能。显然,该方案的扩展性较好。例如,新构件以注册到

事件的方式添加到系统,可很容易增强系统的功能。另外,由于数据访问的抽象性,该方案可以把计算与数据表示中的变化分隔开来,而基于隐式调用的构件仅依赖于某些外部触发的事件的存在,也使得这种分解对复用性有一定的支持度。不过,这种方案的缺点也很明显。例如,隐式调用构件的处理顺序难以控制、数据驱动调用使得这种分解的最自然的实现比其他分解需要更多的存储空间。

上述几种解决方案的对比如表 5-3 所示。

表 5-3 解决方案对比

对比项	管道-过滤器	主程序-子例程	ADT	隐式调用
算法变化	+	-	-	+
数据表示变化	-	-	+	-
功能变化	+	+	-	+
性能	-	+	+	-
复用性	+	-	+	-

管道-过滤器解决方案允许在数据处理流中放置新的过滤器,因此支持处理算法的变化、功能的变化和复用,但数据表示的变化会影响到沿管道传输的数据类型的定义,交换格式的不同也会对解析数据到管道的开销产生影响。基于共享数据的主程序-子程序解决方案对总体处理算法和数据表示的变化及复用的支持很弱,但却因为直接共享数据而导致性能较高,也易于添加新的处理构件。ADT 解决方案对数据表示的变化和复用的支持较好,也不影响性能,但构件之间的交互通过互连实现却可能会导致改变总体处理算法或添加新功能时大量更改现有系统。基于共享数据的隐式调用解决方案非常适合于添加新的功能,但共享数据方法的某些问题却可能导致对数据表示的变化和复用的支持不足,还可能会引起一些额外的运行开销。

从这个例子可知,每种风格都有其自身的优缺点,在进行设计时要充分考虑其适用场合,不能盲目选择。

### 5.4.3 多角度视图架构

软件开发涉及软件系统的可视化、说明、构造及文档化,用户、分析师、设计师、程序员、测试者、项目经理、文档作者等相关人员会在不同的阶段以不同的方式来看待系统,形成多角度的视图架构,如图 5-16 所示。其中,用例视图展现系统的行为,这是用户、分析师、测试者所关心的方面;设计视图支持系统的功能需求,关注系统应该为用户提供什么样的服务,它包含类、接口、协作等,形成了问题及其解决方案的词汇;交互视图主要针对性能、可伸缩性、吞吐量等,展示系统组成元素之间的控制流,包括可能的同步与并发机制,着重于控制系统的活动类及它们之间流动的消息;实现视图主要针对系统发布的配置管理,包括用于装配与发布物理系统的成品,它由一些独立的、可用各种方法装配以产生运行系统的文件构成,关注从逻辑构件到物理成品的映射;部署视图用于描述构成物理系统的构件的分布、交付和安装,包含形成系统硬件拓扑结构的节点。

在视图的静态方面,用例视图可以使用用例图进行展示,设计视图和交互视图可以使用类图、对象图等进行展示,实现视图可以使用组件图进行展示,部署视图可以使用部署图进行展示。在视图的动态方面,这些视图均可以使用交互图、状态图、活动图等进行展示。这

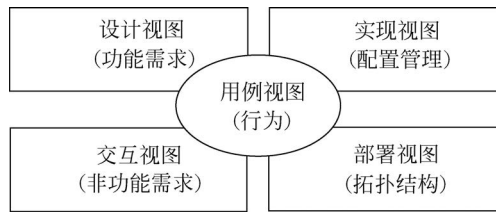


图 5-16 多角度视图架构

些视图可以独立使用,也要注意其关联性。例如,部署视图中的节点有实现视图的构件,这些构件又是设计视图和交互视图中的类、接口、协作、活动类的物理实现等。

不同的人可以专注于各自最为关心的问题,但作为架构设计师,不仅要关心系统的结构和行为,还要考虑其用法、功能、性能、弹性、复用、可理解性、经济与技术约束及其折中,以及审美的考虑。

## 🔑 习题

本书提供在线测试习题,扫描下面的二维码,可以获取本章习题。



在线测试