Chapter 5

第5章 总体设计

需求分析的主要任务是回答系统“做什么”，在对软件需求有了完整、准确、具体的理解之后，接下来的工作就是着手进行软件设计，设计阶段的任务是回答系统“怎么做”的问题。

传统的软件工程方法学采用结构化设计(Structured Design，SD)技术，完成软件设计工作，通常把软件设计工作划分为总体设计和详细设计两个阶段。

5.1 软件设计基础

5.1.1 软件设计的目标

传统的结构化方法将软件设计划分为体系结构设计、数据设计、接口设计和过程设计4个部分。软件设计的过程和目标，就是根据用信息域表示的软件需求，以及功能和性能要求进行数据设计、体系结构设计、接口设计和过程设计。图5.1描绘了软件设计过程中的信息流。

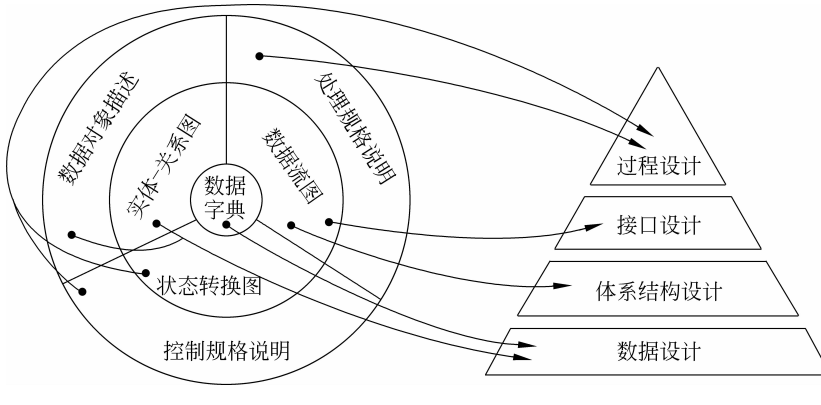

图5.1 把分析模型转换成软件设计

(1) 数据设计：数据设计将分析阶段创建的信息模型转变成实现软件所需的数据结构。在实体关系(E-R)图中定义的数据对象和关系，以及数据字典中描述的详细数据内容，提供了数据设计活动基础。部分数据设计可能和软件体系结构的设计同时发生，但更详细的数据设计活动则发生在每个具体的软件构件(或模块)设计的时候。

(2) 体系结构设计：体系结构设计定义软件的主要结构元素及其之间的关系。体系结构设计表示可以从系统规格说明、分析模型(如数据流图)以及分析模型所定义子系统的交互中导出。

(3) 接口设计：接口设计描述用户界面、软件和其他硬件设备、其他软件系统及使用人员的外部接口，以及各种构件之间的内部接口。

(4) 过程设计：过程设计的主要工作是确定软件各个组成部分内的算法及内部数据结构，并选定某种过程的表达形式描述各种算法。

5.1.2 软件设计的任务

从工程管理的角度来看，传统的软件设计任务通常分为两个阶段完成，如图 5.2 所示。

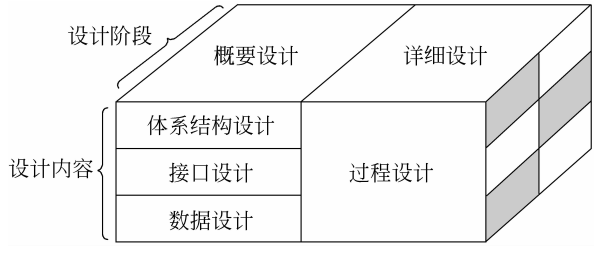

图 5.2 软件设计过程结构图

第一个阶段是概要设计，又称为总体设计或初步设计，将软件需求转化为数据结构或软件的系统结构，包括结构设计和接口设计，并编写概要设计文档。这一阶段主要确定实现目标系统的总体思想和设计框架。系统分析员使用系统流程图或其他工具，描述每种可能的系统，估计每种方案的成本和效益，推荐一个较好的系统方案(最佳方案)，并且制定实现所推荐系统的详细计划。在用户确认后，系统分析员就要设计软件的整体结构和框架，确定程序由哪些模块组成，以及模块与模块之间的关系，最后提出总体设计说明书。

第二阶段是详细设计阶段，即过程设计或构件级设计，其任务是通过对结构表示进行细化，确定各个软件构件的详细数据结构和算法，产生描述各软件构件的详细设计文档，详细设计或构件级设计的根本目标是确定应该怎样具体地实现所要求的系统。

5.1.3 总体设计过程

总体设计的目标是综合采用各种技术手段，将系统需求转换为模块结构、数据结构(或对象/类结构)的表达式，并实现系统的性能、安全性、可靠性要求。

软件结构包括两部分：程序的模块结构和数据的结构。软件的体系结构通过一个划分过程来完成，该划分过程从需求分析确立的目标系统的模型出发，对整个问题进行分割，使其每个部分用一个或几个软件成分加以解决，整个问题就解决了，这个过程可以形象地用图 5.3 表示。该图表明了从软件需求分析到软件设计的过渡。

总体设计过程要解决系统的模块结构，确定系统模块的层次结构。在软件设计过程中，总体设计是关键，它决定了系统结构、数据结构以及软件的质量，反映了系统的概貌。

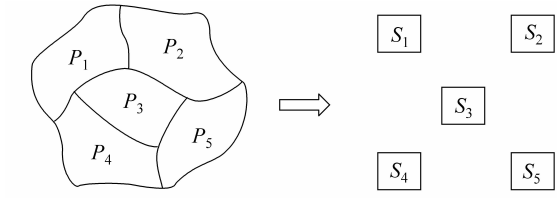

图 5.3　软件解决方案

5.2　软件设计的基本原理

软件设计经过多年的发展，已经形成一套基本的软件设计概念与原则，这些概念与原则经历了时间的考验，已经成为软件设计人员完成复杂设计问题的基础。主要内容如下。

(1) 把软件划分成若干独立成分的依据；

(2) 表示不同的成分内的功能细节和数据结构；

(3) 统一衡量软件设计技术质量。

5.2.1　模块化设计原理

所谓模块，是指具有相对独立性，由数据说明、执行语句等程序对象构成的集合。程序中的每个模块都需要单独命名，通过名字可实现对指定模块的访问。在高级语言中，模块具体表现为函数、子程序、过程等。

在软件的体系结构中，模块是可组合、分解和更换的单元，具有以下基本属性。

(1) 接口，指模块的输入与输出。

(2) 功能，指模块实现什么功能，有什么作用。

(3) 逻辑，描述内部如何实现要求的功能及所需要的数据。

(4) 状态，指模块的运行环境，即模块间调用与被调用关系。

功能、状态与接口反映模块的外部特征，逻辑则反映模块的内部特征。

模块是构成程序的基本构件。对于其他模块而言，只需了解被调用模块的外部特性就足够了，不必了解它的内部特性。在软件设计时，通常先确定模块的外部特性，然后再确定它的内部特性。前者是总体设计的任务，后者是详细设计的任务。

模块化是指将整个程序划分为若干个模块，每个模块用于实现一个特定的功能。把这些模块集成起来构成一个整体，可以完成指定的功能满足用户的需求。划分模块对于解决大型复杂的问题是非常必要的，可以大大降低解决问题的难度。

一个大的软件，由于控制路径多、设计范围广、变量多以及总体的复杂性，比一个较小的软件更不容易被人理解。把一个大而复杂的问题分解成一些独立的、易于处理的小问题，解决起来就容易得多。模块化是软件的一个重要属性，模块化的特性给人们提供了处理复杂问题的一种方法，同时也使得软件能够被有效地管理。这种“分而治之”的思想提供了模块

化的根据：把复杂的问题分解成许多容易解决的小问题，原来的问题也就容易解决了。下面根据人类解决问题的一般规律，论证上面的结论。

设函数 $C(X)$ 定义问题 X 的复杂程度，函数 $E(X)$ 确定解决问题 X 需要的工作量(时间)。对于两个问题 P_1 和 P_2，如果

$$C(P_1) > C(P_2)$$

显然

$$E(P_1) > E(P_2)$$

根据人类解决一般问题的经验，另一个有趣的规律是

$$C(P_1 + P_2) > C(P_1) + C(P_2)$$

也就是说，如果一个问题由 P_1 和 P_2 两个问题组合而成，那么它的复杂程度大于分别考虑每个问题时的复杂程度之和。

综上所述，得到下面的不等式

$$E(P_1 + P_2) > E(P_1) + E(P_2)$$

这个不等式证实了“分而治之”的结论，这就是模块化的根据。

同理，若能将问题 P 分解成若干个独立子问题 $P_1, P_2, \cdots, P_n$ 时，则下列公式成立：

$$E(P) > E(P_1) + E(P_2) + \cdots + E(P_n)$$

若能将问题 P 分解成若干个相关联的子问题 $P_1, P_2, \cdots, P_n$ 时，相关因子分别为 $i_1, i_2, \cdots, i_n$ 时，则下列公式成立：

$$E(P) > E(P_1) + E(P_2) + \cdots + E(P_n)$$

由上面的不等式似乎还能得出下述结论：如果无限地分割软件，最后为了开发软件而需要的工作量也就小得可以忽略了。事实上，还有另一个因素在起作用，从而使得上述结论不能成立。

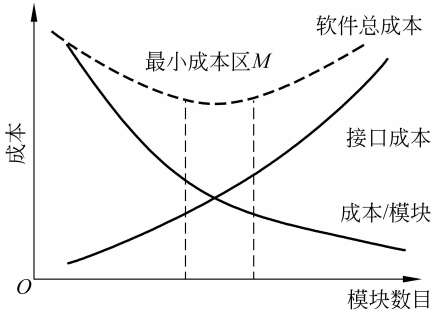

图 5.4　模块化和软件成本

如图 5.4 所示，当模块数目增加时，每个模块的规模将减小，开发单个模块需要的成本(工作量)确实减少了。但是，随着模块数目增加，设计模块间接口所需要的工作量也将增加。根据这两个因素，得出了图中的总成本曲线。每个程序都相应地有一个最适当的模块数目 M，使得系统的开发成本最小。

虽然目前还不能精确地预测出 M 的数值，但是在考虑模块化解决方案的时候，总成本曲线确实是有用的指南。

当考虑模块化解决方案时，除了模块数目之外，还有另外一个重要问题需要回答：怎样定义一个给定大小的模块？用来定义系统内模块的方法，决定了定义出的是什么样的模块。Meyer 提出了 5 条标准，可以用这 5 条标准来评价一种设计方法定义有效的模块系统的能力。下面列出这 5 条标准。

1. 模块可分解性

如果一种设计方法提供了把问题分解为子问题的系统化机制，它就能降低整个问题的复杂性，从而可以实现一种有效的模块化解决方案。

2. 模块可组装性

如果一种设计方法能把现有的(可重用的)设计构件组装成新系统,它就能提供一种并非一切都从头开始做的模块化解决方案。

3. 模块可理解性

如果可以把一个模块作为一种独立单元(无须参考其他模块)来理解,那么,这样的模块是易于构造和易于修改的。

4. 模块连续性

如果对系统需求的微小修改只导致对个别模块而不是对整个系统的修改,则修改所引起的副作用将最小。

5. 模块保护性

如果在一个模块内出现异常情况时,它的影响局限在该模块内部,则由错误引起的副作用将最小。

采用模块化原理可以使软件结构清晰,不仅容易设计也容易阅读和理解。因为程序错误通常局限在有关的模块及它们之间的接口中,所以模块化使软件容易测试和调试,因而有助于提高软件的可靠性,因为变动往往只涉及少数几个模块,所以模块化能够提高软件的可修改性;模块化也有助于软件开发工程的组织管理,一个复杂的大型程序可以由许多程序员分工编写不同的模块,并且可以进一步分配技术熟练的程序员编写困难的模块。

5.2.2 抽象和逐步求精

人们在认识复杂现象的过程中,使用的最强有力的思维工具就是抽象。所谓抽象就是将事务相似的方面集中和概括起来暂时忽略它们之间的差异。或者说,抽象就是抽出事务本质特性而暂时不考虑它们的细节。

当人们面对复杂的事务时,很难一下给出问题的答案。比较有效的方法是按照层次结构将事务的本质特性抽象在层次结构的最高层,高层的特性又可以用一些较具体的特性描述,如此继续下去,直至结构层次的最底层元素。

抽象的概念被广泛用于计算机软件领域,软件工程实施中的每一步都可以看作是对软件抽象层次的一次细化。在软件定义阶段,软件作为整个计算机系统的一个元素来对待;在需求分析阶段,软件解法是使用在问题环境内熟悉的方式描述的;从总体设计到详细设计阶段,抽象的程度逐渐降低,将面向问题的术语与面向实现的术语结合起来描述解法;最后当程序编写出来,也就到了抽象的最底层,完全用实现的术语来描述。

在软件设计阶段,又有不同的抽象层次。软件结构每一层中的模块,表示了对抽象层次的一次精化。软件结构顶层的模块,控制了系统的主要功能并影响全局,底层的模块,完成一个具体的处理。用自顶向下由抽象到具体地分析和构造软件的层次结构,不仅可以简化软件的设计,还可提高软件的可理解性。因此,逐步求精和模块化的概念,与抽象是紧密相关的。

逐步求精是一种先总体、后局部的思维原则,也就是一种逐层分解、分而治之的方法。在面对一个复杂的大问题时,它采用自顶向下、逐步细化的方法,将一个大问题逐层分解成许多小问题,然后每个小问题再分解成若干个更小的问题,经过多次逐层分解,每个最低层

问题都足够简单,最后再逐个解决。

逐步求精最初是由 Niklaus Wirth 提出的自顶向下的设计策略。“将软件的体系结构按自顶向下的方式,对各个层次的过程细节和数据细节逐层细化,直到用程序设计语言的语句能够实现为止,从而最后确立整个体系结构。”

求精实际上是细化过程。我们从在高抽象级别定义的功能陈述(或信息描述)开始,也就是说,该陈述仅概念性地描述了功能和信息,但是并没有提供功能的内部工作情况或信息的内部结构。求精要求设计者细化原始陈述,随着每个后续求精步骤的完成而提供越来越多的细节。

抽象与逐步求精是一对互补的概念。抽象使得设计者能够说明过程和数据,同时却忽略低层细节,求精有助于设计者在数据处理过程中揭示低层的细节。两个概念均能帮助设计者在设计演化中,构造出完整的设计模型。

5.2.3 信息隐蔽和局部化

信息隐蔽是指在设计和确定模块时,使一个模块内包含的信息(过程和数据),对于不需要这些信息的其他模块来说,是不可访问的。“隐蔽”意味着有效的模块化可以通过定义一组相互独立的模块来实现,这些独立的模块彼此间仅交换那些为了完成系统功能而必须交换的信息,而将自身的实现细节与数据“隐藏”起来。

局部化和信息隐蔽是密切相关的。所谓局部化,就是使一些关系密切的软件元素彼此靠近些。在模块中使用局部数据元素是局部化的一个例子。显然,局部化有助于实现信息隐蔽。

将信息隐蔽用作模块化系统的设计标准。一个软件系统在整个生存期内要经过多次修改,信息隐蔽为软件系统的修改、测试及以后的维护都带来了好处,因为绝大多数数据和过程对于软件的其他部分而言是隐蔽的,在修改期间由于疏忽而引入的错误就很少能被传播到其他部分。

5.2.4 模块独立性

模块独立性是指每个模块只完成系统要求的独立的子功能,并且与其他模块的联系最少且接口简单。模块独立性的概念是模块化、抽象和信息隐蔽这些软件设计基本原理的直接结果。只有符合和遵守这些原则才能得到高度独立的模块。良好的模块独立性能使开发的软件具有较高的质量,并能有效提高软件的生产率。这是因为有效模块化(即具有独立的模块)的软件比较容易开发,功能分割明确并且接口简单,当多人合作开发时,这点尤其重要。同时,独立的模块也容易测试和维护。所以模块独立是好的设计的关键。

模块独立性可以由两个定性标准度量,反映模块外部特征的标准是耦合,反映模块内部特征的则是内聚。

1. 耦合

耦合也称块间联系,是对一个软件结构内不同模块间相互联系紧密程度的度量。模块间联系越紧密,耦合性就越强,模块的独立性就越差,因此软件设计中应追求尽可能松散的

耦合系统。模块间耦合强度取决于模块间接口的复杂程度、调用的方式及传递的信息。模块的耦合有以下几种类型，它们的耦合性高低如图 5.5 所示。

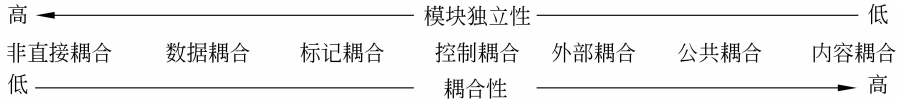

图 5.5 耦合性由低到高

(1) 非直接耦合。指两个模块间没有直接的关系，它们之间的联系完全通过主模块的控制与调用，它们间不传递任何信息。因此这是最弱的耦合，模块独立性最强。

(2) 数据耦合。两个模块彼此间通过参数交换信息，而且交换的信息是简单的数据值，则这两个模块是数据耦合。这种耦合的耦合程度较低，模块独立性较高。

(3) 标记耦合。如果两个模块间传递的是数据结构，如高级语言中的数组名、记录名和文件名等这些名字即为标记，其实传递的是数据结构的地址。两个模块必须清楚这些数据结构，并按结构要求对其进行操作，这样就降低了可理解性。可采用“信息隐蔽”的方法，把该数据结构及对它的操作都集中在一个模块内，就可消除这种耦合，但有时因为其他功能的缘故，标记耦合是不可避免的。

(4) 控制耦合。如果两个模块间传递的是控制信息，如开关、标志等，则称这两个模块是控制耦合的。被调用函数通过控制信息有选择地执行块内某一功能，即被调用模块内应具有多个功能，哪个功能被执行受调用模块的控制。因此，调用模块必须知道被调用模块内部的逻辑关系，即被调用模块不能实现“信息隐蔽”，降低了模块的独立性。模块间的控制耦合不是必需的，可通过将被调用模块内的判定上移到调用模块中同时将被调用模块按其功能分解为若干单一功能的模块，将控制耦合改变为数据耦合。

(5) 外部耦合。如果两个模块都访问同一个全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量信息，则称这两个模块属于外部耦合。

(6) 公共耦合。如果一组模块都访问同一个公共数据环境，则这组模块之间的耦合就称为公共耦合。公共数据环境可以是全局数据结构、共享的通信区、内存的公共覆盖区等。由于多个模块共享同一个公共数据环境，如果其中一个模块对数据进行了修改，则会影响到所有相关模块。另外无法控制各个模块对公用数据的存取，使得软件的可靠性和可维护性比较差。

如果一个模块只是向公共数据环境里传送数据，而另一个模块只是从公共数据环境中取数据，这种耦合就叫松散公共耦合。如果两个模块都向公共数据环境里传送数据，同时也都从公共数据环境中取数据，这种耦合就叫紧密公共耦合。

如果模块间共享的数据很多，通过参数的传递很不方便时，可使用公共耦合。

(7) 内容耦合。如果一个模块直接使用另一模块的内部数据，或通过非正常入口而转入另一个模块内部，这种耦合称为内容耦合。这是最高程度的耦合，也是最差的耦合，一般出现在汇编程序设计中，高级程序设计语言已经不允许出现任何形式的内容耦合。

模块化设计的目标是建立模块间尽可能松散的耦合，为此应尽量使用数据耦合，避免或少用控制耦合，慎用或有控制地使用公共耦合，完全不用内容耦合。

2. 内聚

内聚也称块内联系，是指一个模块内部各个元素之间关系的紧密程度。若一个模块内

各元素(语句间、程序段间)联系得越紧密,则内聚性就越高,模块的独立性就越好。软件设计中应追求尽可能紧密的内聚,理想内聚的模块只做一件事。模块的内聚有以下几种类型,它们的内聚性高低如图5.6所示。

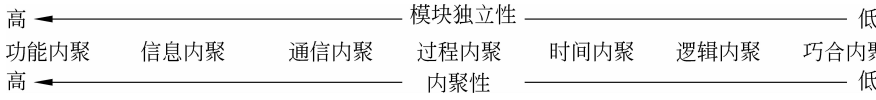

图5.6 内聚性由低到高

(1) 巧合内聚。巧合内聚又称偶然内聚,指一个模块内的各处理元素之间没有任何联系,或者即使有联系,这种联系也很松散,则称这种模块为巧合内聚模块。例如一个模块执行的活动有:打印一行文字,将输入参数的字符串反转,计算一个数组的平均值。这样的模块显然没有主要的功能,而是一些无关指令序列的集合。巧合内聚模块的出现通常是由于对模块大小的限制造成的,例如,为了节省存储空间,将多个模块中出现的重复语句提取出来,组成一个新的模块。这样的模块可维护性差,并且不可能被复用。

(2) 逻辑内聚。这种模块把几种相关的功能组合在一起,每次被调用时,由传送给模块的判定参数来确定该模块应执行哪一种功能。例如,根据输入的控制信息从文件中读一个记录,或者向文件中写一个记录。这种模块是单入口多功能模块,类似的有错误处理模块,它接收出错信号,对不同类型的错误打印出不同的出错信息。

逻辑内聚模块比巧合内聚模块的内聚程度要高,因为它表明了各部分之间功能上的相关关系,但是它所执行的不是一种功能,而是执行若干功能中的一种,因此它不易修改。另外,在调用时需要进行控制参数传递,这就增加了模块间的耦合程度。另外,将未用的部分也调入内存会降低系统的效率。

(3) 时间内聚。时间内聚又称经典内聚。这种模块通常为多功能模块,但模块的各个功能的执行与时间有关,通常要求所有功能必须在同一时间内执行,例如程序设计中的初始化模块。时间内聚在一定程度上反映了程序的某些实质,所以比逻辑内聚模块的内聚程度又稍高一些。因为时间内聚模块中所有各部分都要在同一时间段内执行,而且在一般情形下,各部分可以以任意的顺序执行,所以它的内部逻辑更简单,存在的开关(或判定)转移更少。但是时间内聚把很多功能、独立的任务组合在一起,这给维护与修改造成了困难。

(4) 过程内聚。如果一个模块内的处理是相关的,而且必须以特定顺序执行,则称这个模块为过程内聚模块,这类模块的内聚程度比时间内聚模块的内聚程度更强一些。由于过程内聚模块可能包括按特定顺序执行的多个功能,它的内聚程度仍然比较低。

(5) 通信内聚。如果一个模块内所有处理元素都在同一个数据结构上操作,或指各处理使用相同的输入数据或者产生相同的输出数据,则称之为通信内聚模块。通信内聚模块各部分都紧密相关于同一数据或数据结构,所以内聚性要高于前面几种。

(6) 信息内聚。模块能完成多个功能,各个功能都在同一数据结构上操作,每一项功能有一个唯一的入口点。这个模块将根据不同的要求,确定该模块执行哪一个功能。由于这个模块的所有功能都是基于同一个数据结构(符号表),因此,它是一个信息内聚的模块。

(7) 功能内聚。指模块内所有元素共同完成一个功能,缺一不可。这是最强的内聚,模块不可分割。功能内聚的模块易理解,易修改,也易于实现软件重用,是最理想的模块内聚。

耦合性和内聚性是衡量模块独立性的两个定性标准，在划分软件模块时，没有必要精确确定模块内聚的级别和模块间的耦合级别。重要的是设计时力争做到高内聚，并且能够辨认低内聚模块，有能力通过修改设计提高模块的内聚程度，从而获得较高的模块独立性。

5.2.5　软件结构设计优化原则

软件总体设计的主要任务是软件结构的设计，为了提高设计质量，必须根据软件设计原理改进软件设计，并提出以下软件结构设计的优化原则。

(1) 模块的独立性原则。尽量建立高内聚、低耦合的模块结构，保持模块相对独立性，并以此为原则优化初始的软件结构。如果若干模块之间耦合度过高，每个模块内功能不复杂，可将它们合并，以减少信息的传递和对公共区的引用；若多个模块共有一个子功能，应对它们的功能进行分析，把重复的功能分割成一个独立的模块，再由这些模块调用。通过分解或合并模块减少控制信息的传递及对全程数据的引用，并降低接口的复杂程度。

(2) 保持适中的模块规模和复杂度。一个模块的规模不应过大，模块内的语句行数通常不超过 60 行，以便于阅读和研究。过大的模块往往由于分解不够充分，增加阅读理解的难度。但模块的进一步分解，必须符合问题结构，分解后不应降低模块独立性。过小的模块也会使模块之间的联系变得复杂，可以进行恰当的合并，避免模块过多使系统接口复杂。如图 5.7 所示，为模块分解的实例。

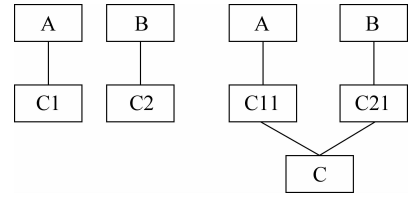

图 5.7　模块分解实例

(3) 软件层次的深度、宽度、扇入、扇出要适当。

一个模块的深度表示软件结构中控制的层数，能粗略地标志系统的大小和复杂度，如图 5.8 所示，该模块结构的深度为 5。模块的深度和程序长度之间应该有简单的对应关系，如果层数过多，则应考虑是否存在许多管理模块过分简单，需要适当合并。

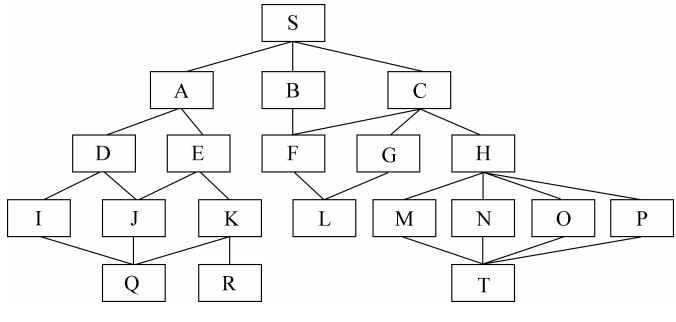

图 5.8　软件结构图

宽度是指软件结构一层中最大的模块个数，图中模块结构的宽度为 8。一般来说，宽度越大系统就越复杂。

扇出是指一个模块直接下属模块的个数，图中模块 M 的扇出为 3。扇出数衡量一个模块直接控制其他模块的数量，扇出数过大意味着模块过于复杂，需要控制和协调过多的下级模块；扇出数太小可以把下级模块进一步分解成若干个子功能模块或者合并到它的上级模

块中去。当然,分解或合并模块必须符合问题结构,不能违背模块独立性原则。

扇入是指一个模块有多少个上级模块直接调用它。图中模块 T 的扇入为 4。扇入越大则共享该模块的上级模块数目越多,这是有好处的,但是,不能违背模块独立性原则。

(4) 模块的作用范围应在其控制域内。模块的控制域是这个模块本身以及所有直接或间接从属它的模块的集合。模块的作用范围是受该模块内的一个判定影响的所有模块。

如图 5.9 所示,模块 A 的控制域是 A、B、C、D、E 模块的集合。若 A 模块内的判定只影响 B 模块,符合上述原则;若 A 模块的一个判定影响 F 模块,则 A 的作用范围不在它的控制域内。

为使模块的作用范围在其控制域内,可以通过以下两种方法。其一是将模块 A 中影响模块 F 的判定上移至模块 W 中,此时,模块 F 处于模块 W^c 的控制域中。

其二是在不违反模块独立性原则的基础上,将模块 F 下移,使其接受模块 A^c 的调用。两种解决方法如图 5.10 所示。

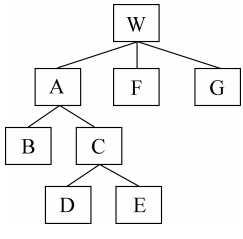

图 5.9 作用范围与其控制域关系

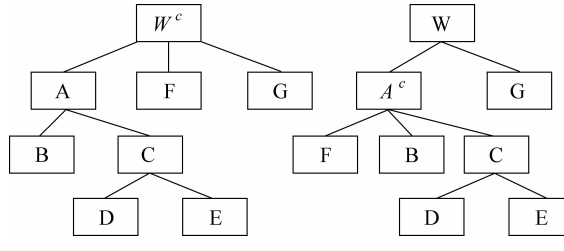

图 5.10 调整模块结构示意图

(5) 降低模块接口的复杂程度。模块接口复杂是软件发生错误的一个主要原因。应该仔细设计模块接口,使得信息传递简单而且和模块的功能一致。

(6) 定义单入口、单出口模块。尽量不使模块之间出现内容耦合,不随便使用转向语句。从顶部进入模块并且从底部退出模块时,软件是比较容易理解的,因此也比较容易维护。

(7) 模块内的数据结构尽可能做到对外部隐蔽和具备灵活的扩充机制,并且模块的功能应该做到可以预测。

5.3 表示软件结构的图形

结构化设计方法(SD)通常采用层次图(H 图)、HIPO 图和结构图(Structure Chart)描述软件结构。

5.3.1 层次图和 HIPO 图

通常使用层次图(H 图)用来描绘软件的层次结构,层次图适合于在自顶向下设计软件的过程中使用。在层次图中一个矩形表示一个模块,矩形框之间的连线表示调用关系,位于