软 件 工 程

计算机科学与技术工程系

2003 年 1 月

目 录

第 1 章
软件工程引论

1.1 关键概念

以下列出本章的关键概念与技巧：

- 软件开发的复杂性根源
- 软件开发项目为什么失败以及怎样失败
- 为什么人与人之间的交流是困难的
- 为什么维护需要大量人力资源
- 问题的概念化
- 问题刻画
- 结构化程序设计
- 功能分解
- 抽象层次
- 结构化分析与设计
- 抽象数据类型
- 继承
- 数据建模
- 软件开发范型的要素
- 软件开发范型的目标

1.2 为什么要建造软件

假想你已经大学毕业，正在为一家国际性软件开发公司工作。你乘坐一架飞机，跨越这个国家飞往一个用户所在地。航班机舱光线很暗，大部分乘客都在打盹，机舱外面很黑，由于浓厚的云层，可见度很低。突然，飞机向左转向，你看到外面是另一架飞机的机翼，它危险地贴近你乘坐飞机的机翼飞过。飞机的突然转向惊醒了大部分乘客，机舱中立即喧哗起来。机组人员要么不知道，要么不说到底发生了什么事情。当你在一个半小时后最终着陆，你感到幸运，但不知道能不能产生足够的安全感再去乘坐飞机。

据我们目前所知，最近没有像刚刚描述的情景发生。我们感觉只是幸运而不是好的计划防止了此类事故。实际上，在1998年12月初的一个星期中，报道了3个相互独立的事故，在这些事故中由于计算机暂停导致两架飞机非常接近地飞行[17]。美国大部分空域中目前控制空中交通的系统已使用25年以上。软件非常陈旧，预计到2003年仅能处理一小部分交通问题[72]。软件同时运行在已有25年历史的硬件上，找替代部件的困难不断增加。目前，此类系统故障开始多了起来[33]。整个系统需要更新。

自20世纪80年代早期，更换空中交通控制系统已成为美国联邦航空管理局（FAA）非常优先的任务。1989年IBM联邦系统公司获得更换该系统的合同，截止期为2001年，预计投入25亿美元[21]。由于面临着极苛刻的需求，因此该软件项目是已进行的最复杂的项目之一。例如，空中

交通控制系统必须具备全局完整性并且每周7天，每天24小时不能停止工作，甚至在升级时或正常维护时，也不允许有停顿时间。任何错误的数据都会引起重大伤亡，任何停机均会导致世界范围出行的延误及潜在的危险。该系统的反应时间不能超过2到3秒。另外，该系统设计时必须考虑到允许小飞机的私人驾驶员和拥有者继续使用其旧设备，并且允许软件可以移植到将来开发出的新硬件上[30]。

在本书中，作为一个可怕故事描述的空中交通控制系统升级是失控的软件开发的一个例子。当IBM获得了该合同后，该系统的花费主要用于软件开发，而仅有8万美元用于硬件[78]。在1993年，IBM联邦系统公司这家负责合同的IBM的子公司被卖给了Loral公司[7]。1994年年初，该系统已花费了23亿美元，但尚未提交系统的任何程序段。预计到1994年年底，整个系统花费将增至50亿美元[21]。1994年晚些时候，FAA局长David Hinson调查了该项目惨败的原因。在调查中，Hinson解释说该系统的问题出在过度急躁的开发和实现计划（包括费用与进度的估计），尤其是给定的软件系统的复杂性，没有充分考虑IBM公司的生产力，特别是在开发的早期阶段需要投入的生产力。最终，实际上FAA并没有明确掌握该系统的某些基本需求[3]。另外，FAA代表与承包商的代表说过，技术的发展比使用这些技术的能力要快得多[3]。作为本次调查的结果，FAA取消或修改该系统四个主要部分[63]。由于已经认识到当前空中交通控制系统中的计算机非常落伍而使许多旅客处于危险之中，因此FAA订购了一套作为权宜之计的系统，将由Formation公司开发，它是一家位于美国新泽西州穆尔镇的软件开发公司[33]。1996年1月，Loral公司被Lockheed Martin收购[71]。Lockheed Martin公司那时失去了1996年9月与Raytheon公司签订的合同中的主要部分[72]。

这个故事并没有结束。我们仍不清楚是否一个可靠的空中交通控制系统能够在我们所描述的混乱中诞生。解决复杂的问题是困难的，但并不是不可能的。软件工程——本书的主题——是找到一条途径，避免软件开发过程走进迷宫和尽可能地节约资源。

在过去的30年中，软件开发的状况可被描述成一种社会性的苦恼[88]。大规模系统软件开发好像一条陷入沥青坑的恐龙[18]。这样的描述表明，虽然有一些成功的例子，但大规模系统软件开发都有一段漫长而痛苦的失败历史。一些成功的例子包括给人印象深刻的当前一些办公应用软件。然而，每一个成功的软件开发项目都存在许多已经渐渐被遗忘的不为人们注意的初期失败。由于那些制造软件的公司想避免负面影响，因此这样的失败很少被公之于众。一般说来，大规模系统软件的开发是一项具有冒险性的工作[9]。

核心的问题是，为什么会有这么多的软件开发项目失败？答案只有一个词，即**复杂性**（complexity）。许多软件项目在某些层面上是非常复杂的。例如，在某些应用领域，软件的操作依赖于专业知识，这种软件往往非常复杂。另外，懂得应用领域专业知识的人很可能不是实际编写软件的人。因此，这些应用领域的专家必须将他们的需求告诉软件开发的技术人员。在处理完理解问题和将来软件用户的需求这个复杂过程后，就要进行开发软件。任何写过程序的人都知道，开发实际上用于完成特殊任务的软件是一个复杂过程。最终，许多已着手进行的软件开发项目的规模都非常大，太大，事实上已经不可能由一个人完成。因而开发技术小组必须将开发任务分成易于管理的模块，一旦每个部分全部完成，小组要将它们组装起来使它们作为一个协同的整体来工作。将大项目分割成几个小部分，每个小部分由不同的个人来开发，并且

保证这些部分可以在一起工作，这样的过程已成为软件开发过程复杂性的另一个来源。

考虑一下创建一个遗传信息库的工作。你作为一个技术人员来创建管理该库的软件，很可能对遗传信息的所有可能形式与应用只有一个肤浅的了解。你的信息收集过程从哪里开始？你能够仅简单地存储与生物体有关的遗传序列吗？应该考虑分子的排序类型吗？你如何表示一个三维分子？科学家是否想在三维空间处理遗传信息？当你考虑你所面对的任务时，很可能还有别的成百上千的问题出现。

为了使你的软件开发成功，你必须与懂得这些问题的人交谈，这些人应该是遗传学领域的专家。你可以通过与顶尖的遗传学家交谈而开始你的信息收集工作，这样你很快就会发现遗传学家的专业词汇与你自己所掌握的词汇完全不同。当她试图解释三维分子结构的相关特性时，你对大多数词汇感到不熟悉。当你鼓励她刻画她脑子中的特别体系时，往往她不能够用你所能理解的方式讲清楚这个体系。这种不同领域专业技术词汇的不相容性被称为**阻抗不匹配**（impedance mismatch）[14]。在某个科学领域工作的人员没有必要去体验阻抗不匹配。许多行业已经开发了他们自己的词汇。

口头、书面语言中所固有的含糊性为领域专家与开发软件的技术人员之间的交流又增添了一层复杂性。例如，或许权威遗传学家表述了如下系统需求“在计算机屏幕上再现三维结构，这样科学家就可以围绕图形表示来操纵分子”。你可能认为你已明白了这个需求，因此你开发了相关软件。假如显示屏不能完全显示一个分子，则用户可以四处卷动画面来查看超出显示屏边缘的部分。然而遗传学家在过去已经使用了一种允许用户从分子内部叠合部分产生三维视图的软件。那位遗传学家将会喜欢这个新软件具备相同的功能。使人与人之间的交流变得复杂的真正原因在于我们都拥有自己不同的**背景知识**（background knowledge），这种差异会使我们基于以往的经验对相同的现象做出不同的解释。

对于未来的用户，不论他们是否是领域专家，都应该让他们参与系统开发的每个阶段。在软件开发过程中，软件开发的计划有必要允许最终用户来评价该系统。不幸的是，许多软件企业的人认为只需要在系统开发早期阶段和开发完成时，向用户了解专业知识。这种态度很可能会导致许多软件开发项目的失败。

在构建软件时，应用本身同样具有复杂性。大部分软件开发项目要求几个不同的开发者共同努力使该项目按时完成。这些关系意味着为了一致地、有效地使一个中等规模的系统概念化，我们需要一种能书写其结构并在几个不同的人之间交流该结构的机制。系统的这种表示方法可以降低它在我们头脑中的复杂性，同时方便相互协作人员之间的交流，包括某个领域的专家与最终用户之间的交流。请见要点栏1-1。

要点1-1 为什么开发软件是困难的

1) 应用领域复杂。
2) 不同的背景知识和不同的专业术语使人与人之间的有效交流变得困难。
3) 自然语言的含糊性。
4) 深刻领会大的开发项目的细节比较困难。

本章所提到的大部分内容涉及到软件开发项目的失效。关于**软件失效**（**software failure**）这个词汇本身，有些问题值得澄清。软件失效可以有许多原因。

- 软件开发过程能够导致没有功能的软件。
- 如果开发出有功能的软件，它也许没有充分地针对某些领域专家或用户的需要。该软件可能在很大程度上达不到预期的要求。
- 最终软件可能表面上满足了需求，但底层的运算也可能不正确。这种软件的运行结果可能会发生错误。
- 软件可能已满足了用户需求，其计算也是正确的，但可能由于用户的错误而产生故障。对用户来说，这种软件的使用方法可能并不直观，因此用户不知道如何正确使用它。
- 软件能够完成用户的一切需要，但由于响应时间太慢而失去了实用价值。

软件企业所关注的另一个主要问题是人们花在软件维护（maintenance）上的精力。一个成功开发出的软件将被长期应用。该软件要定时地进行修改以适应使用该软件的组织的需求变化，加强其功能，修改其缺陷。与开发新软件相比，人们花在软件维护上的精力是非常大的[9]。

由于软件系统的维护是不可避免的，实际上也是值得做的，因此建造软件的一个主要目的是减少维护所需的人力。使用模块化构件构建的结构良好的系统需要的维护精力就比较少，因为一组模块或单个模块可以在不影响系统其他功能的情况下被修改。随着修改的进行，新的错误来源充其量将被局限在少数几个模块。另外，如果软件用模块方式来书写，那么即使某种操作要在软件中多个地方执行，也只需写一段代码。当需要执行这种操作时可以从软件中各个地方调用。模块被修改后，整个系统发生了相同的改变。在系统中多个地方利用同一代码的这种行为被称为**代码重用**（**code reuse**）。代码重用的基本思想是开发通用模块使其在系统中充当多重角色。

图1-1展示了企业中一个典型的场景，即过多的技术人员从事结构不好的系统的维护工作，而留下从事新系统开发工作的人却不够。

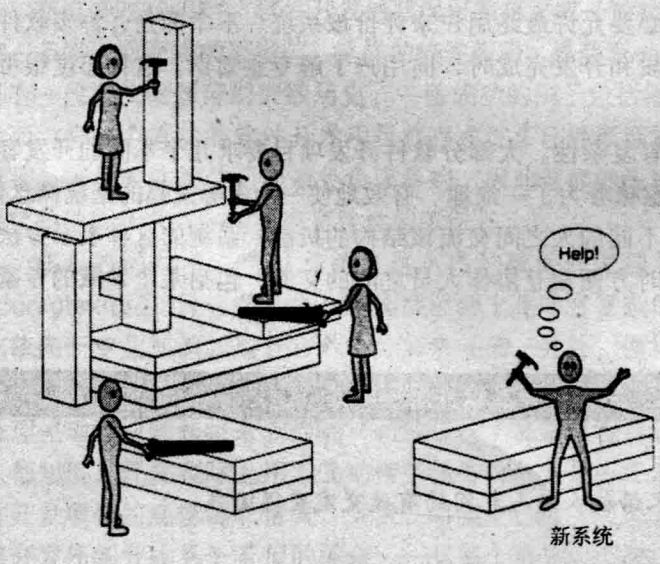

图1-1　失控的维护

因此，软件业在开发新软件时面临着非常复杂的任务。数百万美元的软件开发启动费常常产生不出任何有价值的软件。我们该怎么办？简单地用一个词来回答就是**组织**（organization）。通过组织，我们能够克服软件开发过程中的复杂性。软件工程的基本课题就是控制开发过程并生产出结构良好的、准确的软件解决方案。我们用于开发软件和进行有组织地开发的各种技术确定了软件开发的**范型**（paradigm）。

习题1.1 思考和四个朋友一起建立一个工具房的过程。如果你简单地让你的朋友拿起工具和木料并开始锯木、锤打直到一个工具房出现，会发生什么情况？你怎样着手来确保完成一个可靠的、结构合理和坚固的小屋？

1.3 软件开发范型要素

一个软件开发范型是一个用来指导软件开发过程的技术集合。从大的方面来说，可以把软件开发过程视为由以下三个部分组成：

- 概念化
- 表示
- 实现

理想情况下，这三部分应该是兼容的，从而使开发过程十分顺利。然而对一个项目来说，如果你采用某一个范型中的表示方法而采用另一个范型的实现方法，也是有可能成功开发出有用软件的。这样的范型转换将给我们看到的已十分复杂的过程再增加另外一层困难。为了排除这层不必要的复杂，所有这三个构件应该是兼容的。下面我们将详细分析每个构件。

1.3.1 项目概念化

项目的**概念化**（conceptualization）关注软件开发者怎样考虑待解决的问题。用来考虑和讨论待编程系统的各种要素决定了它的概念化。例如，在一个面向过程的范型下，用过程（函数、过程和子程序）来刻画系统。对图书馆图书跟踪系统的面向过程的刻画如图1-2所示。系统的过程以诸如借书、还书、放回书架等形式来表示。这些要素代表了系统的活动并与真实世界中系统用户承担的活动相对应。

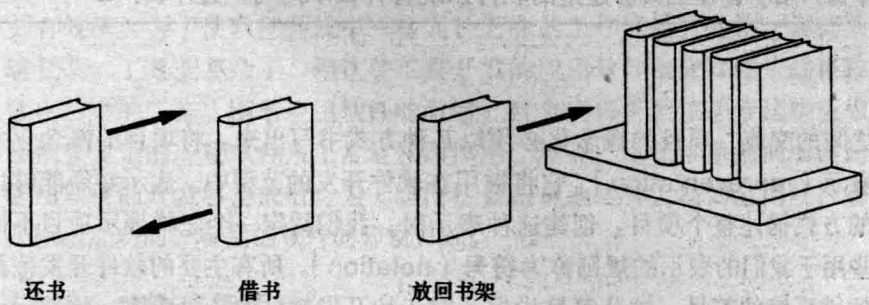

图1-2 图书跟踪系统的面向过程概念化

与此相反，对同一系统面向对象的概念化必须明确构成系统的对象（或事物）与思想。在这种概念化中，我们关心的是书、书架、顾客与图书管理员以及这些事物之间的关系。图1-3表示了这种概念化。例如，顾客来借书与还书。在这种概念化中，借书、还书是存在于书对象与顾客对象之间的关系。在概念化中，每个对象通过某种关系与其他对象关联。

系统的概念化描述了开发者用来组织他们对项目的思考和分析的思维结构。对大部分人来说，不论他们是技术员还是该领域的专家，在讨论软件系统时，对象会比过程更直观[57]。因为一个高效的软件开发范型可以使开发过程中一个阶段到另一个阶段进展更加顺利，所以利用概念化来转换成专家们容易理解和易于评估的形式是值得做的事情。这种简单的转换可以让系统的未来用户更容易地参与到系统的创建当中。

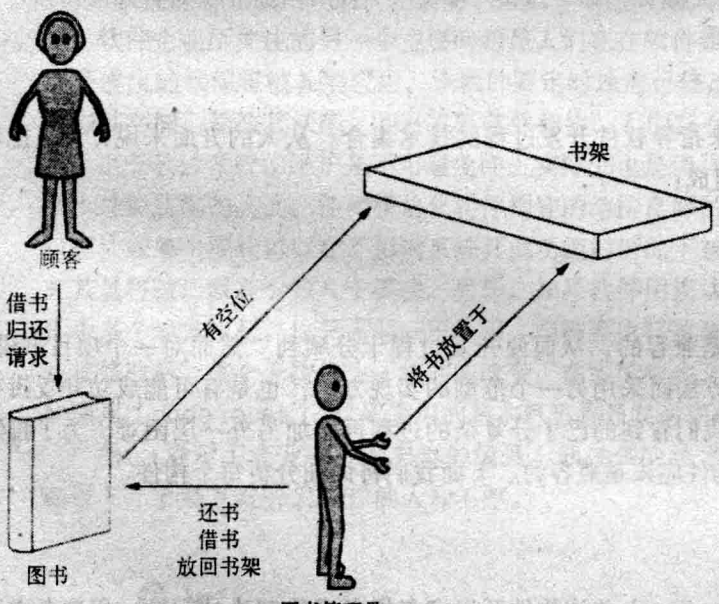

图1-3　图书跟踪系统的面向对象概念化

习题1.2　你将如何分别用面向对象的方法和面向过程的方法概念化一个工具房的建造过程？两者之间有区别吗？为什么？用于管理工具房建造的软件系统有什么不同吗？为什么？

1.3.2　项目表示

为了便于人与人之间的交流，项目的概念化必须以某种方式书写出来。将项目的概念化书写出来就叫做项目的**表示**（representation），它将被用在软件开发的范型中。表示必须能够以一种有效的、无二义的方式描述整个项目。创建这种表示时，我们约定一个怎样表示项目不同部分的规则集合。这些用于我们的表示的规则称为**符号**（notation）。所有主要的软件开发范型都有若干个用来表达软件系统的符号。这些符号指出了对于待开发软件的一个看法。软件行业中受欢迎的开发范型往往使用图形表示方法来交流项目的概念化。例如，面向过程的范型中的

符号可能会用椭圆表示进程，用有向边（箭头）表示进程之间的数据流。与此相反，面向对象的开发范型中可能用矩形表示对象，用有向边表示对象之间的关系。

为什么符号对软件开发如此重要呢？使用符号的一个基本目的是以如下方式表示系统，即以一种无二义的而且在观察特征时不受人的背景知识影响的方式来表示系统。理想情况下，符号应该导致系统的这样一种表示，即它在领域专家和技术人员看来是同一事物。符号这个主题将在下一章做进一步的详细论述。

习题1.3 为了确保工具房的建造是一个有序的过程，你将让建造者们遵守什么形式的指令？你怎样保证每个人都用相同的方式来解释指令？

1.3.3 项目实现

项目的**实现**（implementation）关注如何构造组成软件的源代码。所有的程序设计语言均为创建程序设计语言指令单元提供了某种机制。这些单元是在最终软件系统中共同作用以满足用户需求的模块。回忆前面几节所说的，在良好结构的软件系统中模块化是所希望的特征。在面向对象范型流行之前，大部分企业系统是依据面向过程范型构造的。与面向过程范型相比，面向对象系统使用类（对象框架）作为它的模块化机制。理想情况下，项目实现中的模块化机制与构成项目概念化和表示的要素相同。如果某一项目的概念化和表示用对象的形式来表示，则应该用面向对象的语言来实现该项目。

1.4 软件工程技术简史

历史上，面向过程范型是业界中选用的范型。最近，越来越多的项目已用面向对象的方式来开发。虽然许多人认为采用面向对象的范型是对以前技术的根本性背离，但是我们把这种改变看做是对在此之前的软件工程技术的一种逻辑延伸。因此，我们感觉理解出现在面向对象范型之前的软件工程技术是十分重要的。

图1-4将软件工程技术置于计算机行业中其他开发环境中。我们可以把软件开发的费用分成两部分：软件运行所需的硬件费用和软件开发人员的工资。随着用于硬件的开发费用不断减少，工资的费用将不断增长。在这两者之间的关系转换后，开发出的工具使人在软件开发中的工作更有效率。为了使软件开发中一些冗长乏味的工作能够自动化，人们不断努力，而工具也越来越复杂。工具变复杂了，用这些工具开发的应用软件也变得越来越精致，并承担了越来越多日常生活中的工作。随着应用软件的激增，社会变得更加依赖于这些应用软件。当前的情况是社会依赖复杂的应用软件来正常地发挥作用。然而，由于本章前面所讨论的一些问题，这些复杂应用软件的开发容易失败。为了能使社会正常地运转，新的应用软件的开发应采取一种最易于产生出成功的、有用的软件的开发方式。

1.4.1 结构化编程

随着20世纪60年代**结构化编程**（structured programming）的不断发展，*goto*语句正式从软

件中消除[32]。消除这种语句的动机在于改进源代码的结构，从而改进目标系统的健壮性和可靠性。*goto*语句的使用导致程序结构非常糟糕。当程序中使用*goto*语句时，流程控制、指令执行次序理解起来就很困难，从而导致很难控制。在结构化编程中，*goto*语句被函数调用所替代，从而简化了程序结构。当一个函数被调用时，控制流不再是严格线性的，但控制将（很可能）实际上从该函数段返回到函数调用的下一行代码。

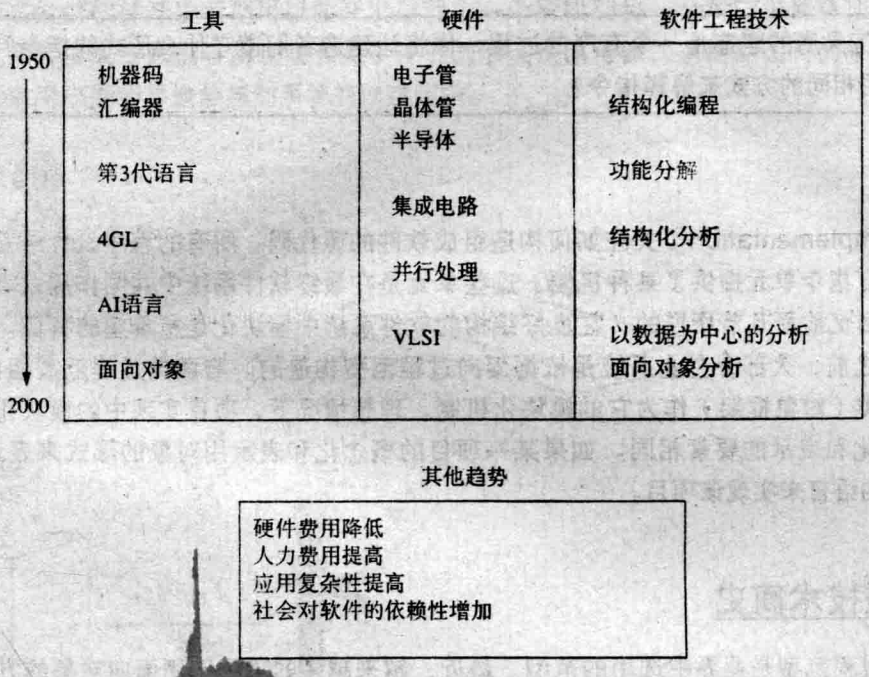

图1-4　计算机行业中的一般趋势

随着系统的实现变得越来越复杂，愈加显现出这样一个事实，即只用结构化编程并不能有效地确保高质量软件的交付。即使采用了结构化编程的技术，生产的软件也会难以理解和使用。这种现实导致了功能分解技术的发展。

1.4.2　功能分解

功能分解（functional decomposition）是一个把待实现的系统分解成一系列逐步增加细节的概念化的过程。这种概念化可以利用一种称为**结构图**（structure chart）的表示来通信。这种方法用方框表示创建的过程，用箭头表示子过程，图1-5给出了结构图的例子，我们将在后面进一步详细讨论结构图。因为逐项分解后可以用模块来表示并实现，所以系统的迭代分解产生了高度的模块化。因为每个模块的主要作用是完成最终系统中的一项功能或一个活动，所以功能分解用于面向过程的范型。待开发系统的这样一种概念化可借助面向过程定义。

为了能够理解模块化是怎样从不断细化的概念化中产生的，让我们探讨一下这些概念化意味着什么。我们以图1-2中所示的图书跟踪系统为例。当我们谈到“图书跟踪系统”时，我们是

用尽可能抽象的方式来描述待实现的系统。该系统从而可以被**分解**成它的要素，即面向过程角度下的更加详细的模块。在一个图书跟踪系统中需要起作用的功能是借书、还书和放回书架。这一模块集合是**第二层抽象**，它比“图书跟踪系统”这个词能够更加详细地描述这个待实现的图书跟踪系统。

第二层抽象中的每一要素能够被进一步分解，产生第三层抽象。例如，借书过程可以进一步分解成以下模块：检查逾期图书、更改图书状态、更新顾客信息和更新图书管理员工作信息。还书过程可以分解成以下模块：更改图书状态、更新顾客信息、更新书架列表以及更新图书管理员工作信息。最后，将书放回书架过程可以分解成以下模块：检查图书状态、更新顾客信息和更新书架列表。把第二层抽象中的每一个元素分解的结果组合起来就形成了第三层抽象。图1-5给出了图书跟踪系统功能分解的结构图。在该图中，公共模块被较高抽象层中的模块所共享。应当注意的是，这并不是一个完全的功能分解，因为其中某些过程仍然十分抽象。例如，第二层抽象中的所有三个模块共享了更改图书状态过程。而且，在所有情况下均没有定义图书状态是如何改变的细节，这一点在以后单个过程的进一步分解中必须定义。

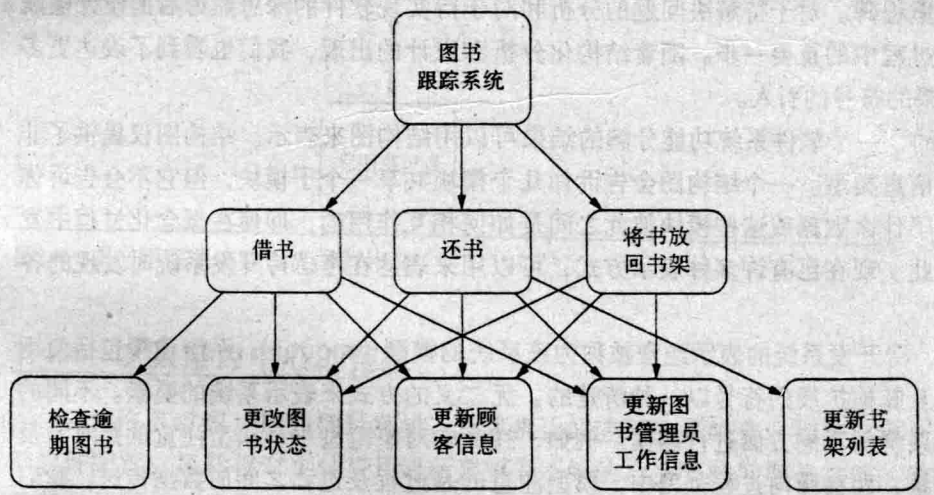

图1-5 图书跟踪系统结构图

这个例子阐明了所有软件开发范型中最重要的内容之一：**抽象**（abstraction）。当一个软件开发小组开始他们的工作时，他们首要的和最紧急的任务是研究待开发系统。他们对系统的最初理解是不精确的，同时也是含糊的，仅能非常抽象地概念化待开发系统。例如，软件开发小组的知识可能仅限于知道这个系统将被图书馆用来管理图书。于是，小组使用一个符号来代表图书跟踪系统。当小组仔细研究手头的问题时，另外的细节将会出现，这些细节将组合成一体形成对后来的系统概念化的表示。

功能分解的目的在于提供一个有序的机制通过抽象来理解待开发系统，以及产生一个结构良好的软件系统。在实际系统的实现中，我们可以把处于不同抽象层次的模块转换成函数、过程或子程序。这样系统的概念化和表示与它们实际的源代码结构是兼容的。这种从系统概念化抽象开始逐步推进到更详细层次抽象的技术就是**逐步求精**（stepwise refinement）。逐步求精的

技术今天仍然在继续使用，但从早期功能分解任务中发展出来的结构图往往不能提供足够的信息来确保一个结构良好的、精确的解决方案。为了开始增添一些必需的信息，结构化分析和设计技术已经开发出来。

习题1.4　你将怎样对建立一个工具房的过程进行功能分解？你怎样利用功能分解的结果来安排建造工具房的人员的工作？

1.4.3　结构化分析和设计

软件工程中下一个主要的开发阶段是**结构化分析和设计**（structured analysis and design）。结构化分析和设计的出现标志着第一个软件工程**方法学**（methodology）的引入。方法学是一个共同描述整个软件开发过程的技术集合。结构化分析是在其前面的技术——结构化编程和功能分解基础上建立起来的。因此它也使用抽象的方法来产生一个模块化结果。

随着结构化分析和设计的引入，最终实现系统的提交将成为软件开发过程中多个里程碑之一，而不是惟一的里程碑。对于待解决问题的分析和对于待实现软件的深思熟虑后的设计逐渐被认为是软件设计过程中的重要一步。随着结构化分析和设计的出现，我们也看到了表达更多分析与设计阶段结果的符号的引入。

回想前面所讲的，一个软件系统功能分解的结果可以用结构图来表示。结构图仅提供了非常有限的信息量和信息类型。一个结构图会告诉你几个模块共享一个子模块，但它不会告诉你这些模块之间共享了什么数据或这些模块彼此之间是如何相互作用的，即使在概念化过程中发现这些信息也是如此。现在已有许多种表示方式，可以用来表达在考虑待开发系统时发现的各类信息。

在软件工程中，待开发系统的表示经常被称为该系统的**模型**（model）。系统建模包括使用几何符号、标记以及其他扩展的符号以一种清楚的、无二义的方式来表示系统的要素。不同的软件开发范型将对系统的不同方面进行建模。例如，在面向对象范型中，模型注重的是对象及它们之间的相互关系，而在面向过程范型中，模型注重的是过程及过程之间的数据传送。我们将在下一章进一步详细讨论建模的重要性和模型中使用的符号。

在面向过程范型中，我们使用**数据流图**（data-flow diagram）来作为分析和设计阶段结果的表示方式。这些图代表了在最终的软件实现中不同过程之间数据移动的方式。功能分解是面向过程范型中所使用的一种主要技术之一。功能分解的结果用数据流图来表示，每个图的集合都表达了某一层的抽象。

图1-6是一个数据流图，用来表示图书跟踪系统的借书过程的一部分。该图表示了借书过程的结束方式。这个图特别表示了哪一个子过程被调用了以及过程之间传送了什么信息。例如，借书过程与检查逾期图书过程之间的联系表明顾客ID是由借书过程传送到检查逾期图书过程。检查逾期图书过程返回一个布尔值（真或假）到借书过程。检查逾期图书过程也通过顾客ID调用另一个过程，即读顾客记录。读顾客记录过程或者返回一个顾客记录或者在找不到顾客记录时返回一个空值。

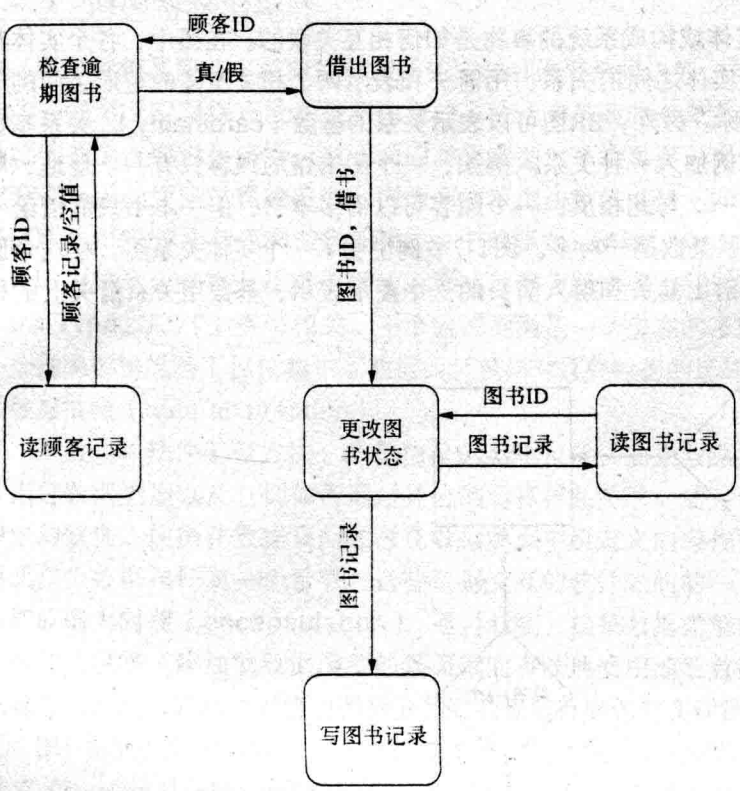

图1-6 图书跟踪系统数据流图的片段

1.4.4 以数据为中心的范型

随着许多组织长期使用计算机，计算机化的应用越来越多，如果想得到的结果是一个有效的软件配置，显然仅对个体应用进行建模是不够的。主要的问题是在同一组织内的不同应用通常需要共同的数据元素，并且这些数据元素在每个应用中被典型地复制。因为一个组织每天使用多个应用程序，所以这些复制的数据元素很快产生了冲突和不正确的值。在某个应用中有些数据值被改变，但是并非每个数据元素的复制版本都被更新，因此就导致了错误的数据值。例如，为公司工作的一名员工移动和归档了一个地址表格的变化。这个表格总是导致职员工资单应用文件的更新，与此同时，现在已是不正确的地址（原地址）将仍然存在于奖金或退休应用文件中。

以数据为中心的方法学的贡献在于用一种称为**数据建模**（data modeling）的技术扩展了结构化分析。数据建模发生在单个应用开发之前。数据建模的目的在于确定整个组织所需的数据，从而创建一个类似于关系数据库的、集中的、完整的数据储存库，进而可以开发单个应用并从这个集中的数据库中提取其所需的数据。这个数据模型用一种称为**实体关系**（Entity Relationship，ER）**图**的图形技术来表示[24]。ER图的基本用途是用于关系数据库设计。在数据建模的最初阶段之后，单个应用可以采用集中来自中心数据储存库的数据进行结构化分析和设

计的方法来开发。

在ER图中，我们设法展示实体或构成系统的事物是如何相互关联的。在图中，各个实体都用标注有实体名的矩形来表示。实体之间的关系，用箭头和表示两个相关实体的矩形之间的菱形框表示。菱形框上标有关系名称。另外，ER图可以表示关系的**基数**（cardinality）。关系基数描述了每一类实体可能有多少实例加入一种关系。例如，一个图书馆的顾客只有一个地址，所以顾客与地址之间的基数是一对一。与此相反，一个顾客可以借多本书，但一本书只能借给一个顾客，所以顾客与书之间的关系基数是一对多。图1-7举例说明了一个实体关系图，在这个图中实体用矩形表示，关系用带有射出箭头和射入箭头的一个菱形表示，基数用关系箭头上的标记来表示。

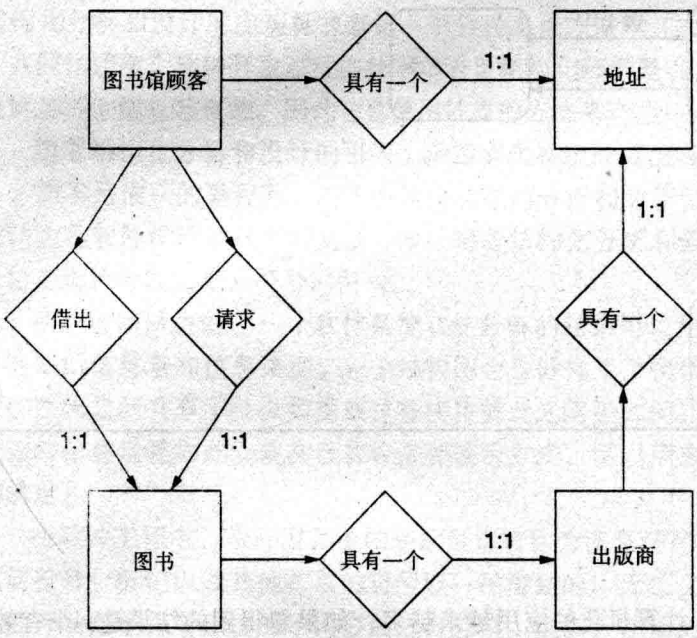

图1-7 图书跟踪系统ER图

以数据为中心的范型不应与**面向数据结构的设计方法**（data-structure-oriented design method）相混淆[40、53、80、99]。在面向数据结构的设计方法中，软件是围绕所设想系统的数据结构来构建的。Warnier-Orr设计方法包括构图技术，这种构图技术有助于数据结构的向下层次分解。**Jackson程序设计系统**[153]是从数据结构开始，导出程序结构的另一种方法。在此，我们仅给出面向数据结构的设计方法的很少细节，原因在于这些方法中没有一个方法针对数据结构设计这个重要问题。因为它们没有针对这一问题，所以这些方法也没有针对完整的软件开发过程，因此这些方法不是合格的成熟的范型。

不幸的是，以数据为中心的范型并不广为人知，也没有在业界普遍应用。在以数据为中心的范型在业界中立足之前，面向对象范型已被开发出来了。面向对象范型把数据分析与待开发应用程序的分析综合成一个对象的结构，它既包含数据也包含过程。