第 5 章 软 件 实 现

软件实现是把软件设计的结果“翻译”成某种程序设计语言，并通过软件测试后，能正确运行且得到符合结果的程序。软件实现包括编码、测试、调测、优化等一系列工作。鉴于软件测试本身是一个较大的主题，因此将在第 6 章专门介绍。

本章内容主要涉及程序设计语言，但不具体介绍如何编写程序，而是从软件工程的角度在更广泛的范围来讨论程序及编码，包括程序设计语言的分类、特性、准则及程序编写规范等内容。

编码实现设计的过程，编码质量的好坏，将直接导致用户体验和软件维护。

5.1 程序设计语言

程序设计语言是机器按照人的指令完成相应任务的工具。遗憾的是，人类使用的自然语言，计算机目前还不能完全识别和理解，因而人们设计出人与机器都能理解的结构化语言。由于应用领域和设计思想的千差万别，不同的结构化语言有很大差别，因而不同的结构化语言在体现人的设计过程和实现的方式上各有千秋，对代码的编写、测试、修改、维护等都产生了深远的影响。

5.1.1 程序设计语言的分类

从程序设计语言出现、发展至今，出现了数百种不同的程序设计语言。特别是 20 世纪 60 年代以后，程序设计语言随着软件工程思想的不断发展，经历了从低级到高级、从简单到复杂、从非结构化到结构化程序设计再到面向对象程序设计的发展过程。因此，从软件工程角度来看，同时结合程序设计语言的发展历史，可以将程序设计语言大致分为如下四个阶段。

1. 第一代计算机语言——机器语言

自从有了计算机，就有了计算机语言。不过最早的语言与机器的硬件系统有着紧密联系。由机器指令组成的代码不能随意在不同机器上执行。因为这些机器指令都用二进制编写，指令地址是以绝对地址（物理地址）的形式出现的。此外，二进制的编码方式不仅将绝大多数人挡在计算机程序设计的门外，而且即使是计算机专家（当时也是数学家、逻辑学家）编写的二进制指令也经常出错，且难以改正。

2. 第二代计算机语言——汇编语言

汇编语言也是与系统硬件直接交互的机器语言，但它已经有了一定的符号指令。符

号指令增加了对编码的理解，增强了对编码的记忆和使用，降低了程序的出错率，可以提高对程序的修改效率，从而增加程序的可靠性。

第一、二代语言不利于计算机应用的推广，更不具备软件工程中提出的设计、维护等过程，它们已逐渐退出历史舞台。但它们无须计算机对程序语言作更多的“理解”（即无须编译过程），就能执行，且消耗资源少，运算效率高。更重要的是，它们具有现代高级程序设计语言不具备或难以完成的系统操作，因而在一些有特殊要求的领域还有一定的应用。

3. 第三代计算机语言——高级语言

从 20 世纪 60 年代后期开始，随着计算机应用从科学计算逐步转向商业应用，直至现在的家庭、个人娱乐、工作和学习，高级程序设计语言逐步得到发展，并走上计算机研发的大舞台。

早期的高级程序设计语言，如 ALGOL、FORTRAN、BASIC 等，现在看上去它们对应用领域的支持还较弱，但它们已具备高级程序设计语言的基本特征：结构化设计、数据结构的定义和表示、控制逻辑的支持以及与机器硬件的无关性等。

从 20 世纪 80 年代开始，面向对象程序设计语言开始崭露头角，C++、Java、VB、C#等高级程序设计语言相继出现，定义类、对象、封装性、继承性、多态性、消息机制等面向对象程序设计技术也不断涌现。它们具有良好的可扩展性、可移植性、可维护性等，为软件质量的提高提供了可靠的工程技术支持。

与这些较为通用的高级程序设计语言相对应的，还有一些专用于某个领域的程序语言。如 Lisp 和 Prolog 语言主要应用于人工智能领域的符号推理，Mathlap 用于数学工程运算等。专用语言因为有较强的应用针对性，因而有简洁的语法和高效的运算特性，但它们的可移植性、可维护性等较差。

4. 第四代计算机语言——4GL

第四代语言（Fourth Generation Language，4GL）是过程描述语言。相比第三代语言详细定义数据结构和实现过程，4GL 只需要数据结构的定义和将要实现的功能，而实现的过程被隐藏起来。最典型的 4GL 应用是数据库的结构化查询语言（Structure Query Language，SQL）。对数据库的操作只需提供计划要完成的任务命令，而无须考虑实现过程。

目前 4GL 得到了一些商业方面的发展，如报表生成、多窗口生成、菜单、工具条等的生成，都无须考虑编码。此外，用形式化定义的结构化需求描述、设计方案等都能通过 4GL 生成相应代码，并经过人工修改后，得到实际应用。

5.1.2 程序设计语言的特性

不同程序设计语言的特性会影响到整个软件系统的效率和质量。由于不同程序设计语言在语法上和技术上都有一些限制，会影响到设计描述和处理活动，因而要考虑程序设计语言特性对系统实现所带来的影响。

（1）一致性。程序设计语言的一致性是指语言中所用符号的兼容程度，以及对语言用法规定的限制程度。如在 FORTRAN 语言中，括号可以用作下标标记、表达式优先

级、子程序的参数表分隔符等。这样“一个符号，多种用途”的表示方法容易出错。但在面向对象程序设计中，由于引入了重载的概念，使运算符可以有其他含义。这是为了使自定义类与原有数据类型保持操作上的一致，让使用者在调用函数时有记忆的一致性。

（2）二义性。程序设计语言的二义性是指符合语言语法定义的语句，却出现了多种理解方式，而计算机只能用机械方式理解其中的一类，因而出现二义性。如对语句Y=X++与Y = ++ X，人们理解上就会产生错误。在面向对象程序中，由于提供了运算符重载机制，因而也会有符号的多重理解。但这不会造成理解上的混乱，因为重载是需要用户自定义才能实现的。用户自己定义的部分，对其理解不会出现二义性。

（3）局部性。局部性是指程序设计语言对模块化支持的程度。在程序设计语言中，对模块的定义提供了语法，如函数定义，以及用大括号“{ }”或“begin…end”描述语句的一个片段。通过这些符号，支持结构化编程，支持各类数据结构在有效范围内使用，体现了信息隐藏特征。

（4）易编码性。程序设计语言是要将设计方案转换为代码。采用的设计方案应支持对复杂数据结构表示、文件操作的便利、类对象的定义以及对常用算法、常用数学计算能力等的操作，便于将设计转换为代码，更好地体现设计者的思路。

（5）可移植性。随着软件工程的发展，技术的更新及网络的日益普及，软件系统全球开发已成为现实，并将成为软件研发的趋势。因此，对源代码跨平台的支持，逐渐成为优先考虑的问题。国际标准化组织（ISO）、美国国家标准协会（ANSI）和国际电子电气工程师协会（IEEE）不断修订代码标准，以促进代码的可移植性。但由于种种原因，如技术要求、企业知识产权保护、商业考虑等，各软件公司的编译器在支持代码的可移植性上都存在着不足。

（6）可维护性。没有不需要维护的软件系统。无论软件过程管理如何及时、有效，最终都要定位在代码的修改和完善上。因此，代码在变量命名（支持长字符串）、自动缩进排版等要求上，都要支持可维护性特征。

（7）配套的开发工具。优秀的开发支撑环境，不仅便于良好的代码编写，而且为语法纠错、测试、调试、多文件组合、代码库建设、代码的逆向工程等提供强大功能。

5.1.3 选择程序设计语言

程序设计语言的选择不是在编码时才选择。早在软件设计前就必须确定选择何种语言。从技术上说，只有提前确定程序语言，才能更好地支持设计的思路，才能更好地展现设计方案。从经济和法律上说，功能越强大的开发平台，其成本也较大。因而应选用与当前设计相符的软件开发工具，并避免由此发生的法律风险。

不同的程序语言机制，对设计的支持不尽相同，目前被广泛采用的是结构化程序设计语言和面向对象语言。

1. 结构化程序设计语言机制

结构化程序设计语言的机制基本上有如下几项：

（1）数据结构（变量和常量）的显示表示。不同数据结构的定义，会导致算法过程效率的不同。如链表与数组，在对元素的排序、插入、删除和查询的操作就完全不同。

结构化语言支持复杂数据结构的定义，并能提供语法纠错。但有的语言，如 BASIC，就支持不定义数据类型而直接使用变量，容易造成编码在理解和使用上的混乱。

（2）模块化编程。结构化语言支持模块独立编译。模块包括自身数据结构和算法，数据的输入和输出。它通常具备以下三个部分：

- 接口定义：模块所需数据的输入、输出。
- 模块实现：包括自身数据结构和算法过程。
- 调用方式：以何种方式运行模块。

（3）控制结构。几乎所有的高级程序设计语言都支持顺序、分支（选择）和循环结构。有时为了提高运行效率或技术上的需要，有的语言提供 goto 语句，以及用 if…goto…构成循环结构。对于模块间调用控制，提供模块间相互调用和模块自身的递归调用。递归调用运行效率低，且容易陷入“死循环”调用而无法结束调用过程，但递归调用算法实现简洁。

2. 面向对象程序设计语言机制

面向对象设计语言除了结构化语言所支持的机制之外，还增加了面向对象特征和机制。

（1）类。局部化设计原则是将数据结构与操作数据结构的行为集中在一起。类就很好地支持了这一原则，并且类的内部结构还提供外部访问类内部的权限（public、protected 和 private）。类的封装性很好地体现了模块化的信息隐藏原则。

（2）继承性。继承性是使得类自动具有其他类的属性（数据结构）和方法（功能）的机制。通过继承，使得现有系统在原有代码基础上，不加修改或稍作修改，就能实现新的需求。这不仅提高了开发效率，更保证了软件质量。

（3）多态性。多态性是指相同的模块接口定义，却有完全不同的实现过程。这样，使得具有相同语义而算法不同的模块可以共享相同的接口定义，减少调用模块时的理解和记忆负担。如鸟和兽都有“吃”这一行为，但显然它们“吃的方式”不同，因而可以各自定义如下接口：Bird_Eat()与 Beast_Eat()。这样定义的结果导致将来扩展有关生物“吃”的操作时，需要不断增加关于“吃”的新的接口定义。这不仅不利于系统的扩展和维护，而且也给设计和使用带来困难。借助多态性机制，可以把所有关于“吃”的行为统一定义为：Eat()，并借助继承性来实现不同的操作过程，这样就自然地反映了不同生物的“吃”在行为上的差异。

（4）消息机制。消息是实现多态性的重要机制之一。如前所述，鸟与兽关于“吃”都用 Eat()来统一定义，如何区别调用两类不同的 Eat()呢？关键在于对象。消息（如“吃”）是由对象发送的。如定义“鸟”的对象“麻雀”，则“麻雀”发送出“吃”的请求，显然应该调用“鸟”类中定义的“吃”的行为，而不会错误的调用“兽”类中定义的“吃”的行为。由此可以得出，消息由对象、方法、参数共同构成。此外，更广义的消息结构还包括消息的发送者、接收者和消息编号等。

3. 选择程序语言的准则

了解程序语言各自不同的机制，结合软件设计方案的要求，综合考虑可测性、维护性，程序语言开发环境的支撑，以及开发过程的管理和成本等问题，使得理想的程序语

言选择标准有时是困难的。因此，结合实际的可操作性及实用性，应考虑以下程序设计程序语言选择的准则：

（1）工程项目规模。程序语言是用于实现工程的。工程规模的大小，需要程序语言结构的灵活性支持。因为项目规模越大，其不可预测性的因素也越多，因而需要程序语言在修改性、适应性、灵活性等方面给予更大支持。

（2）用户需求。一是用户需求的易变性；二是软件维护中用户的参与性。如果用户参与到开发、维护过程中，则应听取用户对程序语言选择的意见。

（3）开发和维护成本。这与程序语言及程序语言开发环境都密切相关。程序语言开发环境自身也是软件系统，也需要维护和技术支持。这些都将构成项目成本。

（4）编程人员对程序语言的熟悉程度。选择编程人员熟悉的程序语言，不仅开发效率高，而且也能保证软件质量。

（5）项目的领域背景。有一些应用领域（如工程计算），有本领域专用程序设计语言。这样，使得所选语言不仅有针对性，还能提高开发效率。即使采用通用程序语言，也要与应用领域相结合，并进一步考虑该领域将来的发展情形。

5.2 程序设计风格

根据软件工程观点，在选定程序语言，完成设计方案后，程序设计的风格在很大程度上影响程序的可理解性、可测试性和可维护性。程序设计风格是指在程序设计过程中，设计人员所表现出来的编程习惯、编程特点和逻辑思维。

从软件工程发展中人们认识到，程序的阅读过程是软件实现、测试和维护过程中一个重要组成部分。因此，一个良好的程序设计风格，是在不影响程序功能、性能前提下，系统地、有效地组织程序，增强代码的易读性、易理解性。

5.2.1 程序编排和组织的准则

源代码按照一定准则编排，使得逻辑清晰、易读易懂，这已成为良好程序设计的标准。程序的编排和组织，将按照源程序文件、数据说明、语句结构和输入输出来综合体现。

1. 源程序文件

源程序文件中包含了标识符命名、注释以及排版格式。

（1）标识符命名。标识符包括常量、变量、函数名称、宏定义。这些符号命名除了遵循语言自身规定的语法外，还应尽量做到：

- 以具有实际意义的词或短语命名，使读者能望文生义。如求和用 Sum，表示是否有效用 isValid。函数的命名最好能体现函数功能，如从 XML 文件中获得记录，可以命名为 GetRecordFromXML(string strXMLFileName)。这比用 Record()、GetRecord()语义更明确，并能增加代码的可读性。
- 命名方式在整个程序文件中做到统一规范。一是统一用英文或汉语拼音命名；二是分类命名。如与文件操作有关的函数，可加上 file 作为前缀标识符；与字符串

操作有关的标识符，可以加上 string 或 str 作为前缀标识符；三是尽量使用领域词汇或用户的习惯用语。

（2）代码中的注释。注释不是程序代码，但却起着正确、有效理解代码的关键作用。注释允许用自然语言来编写，书写内容要言简意赅，无须冗长。对于代码中的注释，主要包括：

- 程序文件整体的叙述，简述本文件所定义的内容。
- 程序主要的数据结构、常量、静态量、枚举量的定义说明。
- 函数接口说明，包括函数参数、返回类型、简要功能描述及代码编写者、编写日期。

【例 5.1】 下面是一个用 C#语言编写的函数接口说明的实例，“///”是 C#语言注释的 XML 表示。

```
/// <summary>
/// 根据给定的键（关键字）查找对应的权值
/// </summary>
/// <param name="strKey">键（关键词）</param>
/// <param name="Value">值（权重）</param>
/// <returns>true：正确查找到键，并给出对应的值；false：键不存在</returns>
public bool TryGetValue(string strKey, out double Value)
{
    for (int i = 0; i < m_iCount; i++)
    {
        if (strKey == m_KeyWeightSet[i].m_strKey)    // 查找成功
        {
            Value = m_KeyWeightSet[i].m_dWeight;
             return true;
         }
    }
    Value = 0;
    return false;         // 查找失败
 }
```

（3）编排格式。代码的编排是在不影响程序功能和性能的前提下，加入换行、空行、空格等内容，使得源代码富有层次感，更易阅读和理解。下面是用 C#语言编写的不同排版风格的相同代码，读者自能体会其优劣。

【例 5.2】 不同编排风格的相同代码，对程序可理解性的影响。

代码一：没有层次感的代码

```
/// <summary>
/// 将 a 中整数数组按照从小到大的顺序排序
/// </summary>
/// <param name="a[]">输入将要进行排序的数组</param>
/// <param name="size">数组大小</param>
void BubbleSort(int a[], int size)
```

```
{
for (i=size-1; i>=1; i--)
{
for (j=0; j<i; j++)
if (a[j] > a[j+1])
{
t = a[j];
a[j] = a[j+1];
a[j+1]= t;
break;}}}
```

代码二：有层次感的代码

```
/// <summary>
/// 将 a 中整数数组按照从小到大的顺序排序
/// </summary>
/// <param name="a[]">输入将要进行排序的数组</param>
/// <param name="size">数组大小</param>
void BubbleSort(int a[], int size)
{
    for (i=size-1; i>=1; i--)
    {
        for (j=0; j<i; j++)
            if (a[j] > a[j+1])
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1]= t;
                break;
            }
    }
}
```

2. 数据说明

作为加深对程序代码理解的重要手段之一，数据说明是首要工作，尤其对必要数据的说明显得更为重要。

（1）变量和常量的命名应遵循匈牙利命名法。即在阅读代码过程中通过变量名称，不仅知道变量的含义，还能知道变量类型。如标识符 m_iStackSize，该变量表示栈的容量大小（整型），以及它是类的成员变量。简单地说，匈牙利命名法就是将标识符的命名规范为：“数据类型＋标识符”。在数据类型中，各种前缀及含义如下：i 表示整型，f 表示单精度浮点型，d 表示双精度浮点型，b 表示布尔型，p 表示指针，const 表示常量，ch 表示字符型，str 表示字符串。struct 表示结构型，C 表示类类型。

（2）对于复杂的数据结构，以及模块中操作的文件类型，首先对数据结构的整体进行说明，再对复杂结构中的各数据进行说明，做到整体结构、主要数据都应有注释。

【例 5.3】 用类模板实现保存不同数据类型元素的数组，并内嵌迭代器访问数组元素。下面的代码给出了该类模板的接口定义以及注释。

```
//定义数组类模板，实现保存不同数据类型元素，并内嵌迭代器来访问数组元素
template <class T>
class Array {
public:
      Array( unsigned sz );  // 类模板的构造函数，并设定能保存的数组元素个数
     ~Array();
     T& operator[ ]( unsigned i );
      Array<T>& operator=(const Array<T>&);
      friend ostream& operator<<(ostream& os, const Array<T>& arr);
      class iterator;    // 向前引用申明
      friend iterator; // 友元类申明，用于类中成员函数对外部类私有数据的直接访问
      // iterator 是一个嵌套类，用于指向 Array<T>类中的一个元素
      class iterator
      {
      public:
        // 构造函数,参数 isEnd 确定迭代器的起始位置是数组的首个还是最后一个元素
        iterator(Array<T>& arr, bool isEnd = false);
        T* operator++(int);             // 迭代器指向下一个元素
        bool operator<(const iterator& it);  // 比较数组元素位置
        T& operator*();                 // 获取迭代器指向的当前数组元素
       private:
           T* p;                        // 指向当前数组元素的指针
       };
       iterator Begin();                // 获取当前数组元素的起始位置
       iterator End();                  // 获取数组最后一个元素的位置
private:
       T * values;                      // 数组元素列表
       unsigned size;                   // 数组容量
};
```

（3）变量定义尽可能与变量的使用物理地组织在一起，便于查阅和增强理解，正如例 5.3 所示，将类及相关接口、结构放在一起定义。

3. 语句结构的处理

程序语句的组织，以行为单位。语句的结构除了特殊性能要求之外，应该力求表达简单、直接。可能产生歧义的语句都应重写或拆分成多条语句，以使其语义明晰。

（1）每行语句只表达一个语义信息，如赋值、运算、函数调用、判断等。不要同时具有多个表达式。如下代码：

```
a=0;
push(stack，a++);
```

由于编译器在函数参数入栈、表达式求值的顺序上会略有不同。因此，对于上述的 push

语句，入栈时参数是自左向右还是自右向左，会造成 a 的值是 0 还是 1 的混乱。这样，对于不同编译器的不同“翻译”，不仅带来理解上的歧义，也给软件测试和维护带来困扰。

（2）现阶段对编码的标准，已是可理解性第一、效率第二。随着硬件存储空间的不断扩大，运算速度越来越快，而硬件成本却又在不断下降，因此，牺牲较少的效率或通过提升硬件性能换来代码可理解性的增加是值得的。看下面代码：

```
void swap(int x, int y)
{
   x = x + y;
   y = x - y;
   x = x - y;
}
```

这段代码所实现的功能难以理解（实现两个整数互换），甚至还需要纸和笔进行演算才能帮助理解。同样的功能，换成如下形式的代码：

```
void swap(int x, int y)
{
   int t = x;
   x = y;
   y = t;
}
```

上述代码仅增加一个整型的临时变量，但对两个整数交换的过程一目了然。

（3）尽量使用开发环境提供的各种类库、函数库、中间件等，以减少出错。

4. 输入输出设计准则

输入输出的信息和操作直接面对用户。它不仅给出数据运算的结果，还给出系统在运行过程中的一些有用提示，甚至需要与用户交互才能完成任务。因而对输入输出的设计应该做到：

（1）输入输出的格式在整个系统中应该统一。

（2）对用户的输入要进行必要的限制和检查，使整个系统能得到有效控制。

（3）对输入数据应该有必要的缺省值。

（4）给用户输出的反馈信息要及时、准确。

（5）对输出的信息要有解释、说明。

（6）异常引发的系统问题，需要有数据恢复机制和用户选择操作。

5.2.2 程序设计的效率

首先需要说明的是，强调程序编码的编排是为了增加可阅读性和理解性，甚至可以牺牲部分效率。但这并不意味着不考虑算法效率问题。

1. 设计逻辑结构清晰、高效的算法是提高程序设计效率的关键。

下面通过对已排序的数组进行关键词检索，来看算法效率对程序设计效率的影响。

算法一：顺序检索关键词

```
/// <summary>
///从已排序的整数数组中检索关键词
/// </summary>
/// <param name="a[]">输入将要进行排序的数组</param>
/// <param name="Size">数组大小</param>
/// <param name="Key">待检索的关键词</param>
/// <returns>如果检索成功，则返回关键词对应的位置，否则返回－1 </return>
int OrderRetrievd(int a[], int Size, int Key)
{
    for (int i = 0; i < Size; i++)
    {
        if (a[i] == Key) return i;   // 检索成功
    }
    return -1;                       // 检索失败
}
```

算法二：折半法检索关键词

```
/// <summary>
/// 从已排序的整数数组中检索关键词
/// </summary>
/// <param name="a[]">输入将要进行排序的数组</param>
/// <param name="Size">数组大小</param>
/// <param name="Key">待检索的关键词</param>
/// <returns>如果检索成功，则返回关键词对应的位置，否则返回－1 </return>
int DichotomyRetrievd(int a[], int Size, int Key)
{
    int low = 0, high = Size- 1; mid;
    while (low <= high)
    {
        mid = (low+high)/2;
        if (a[mid] == key) return mid;   // 检索成功
        if (a[mid] > key)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;                           // 检索失败
}
```

直观上看，算法一比算法二要简单、易懂。但很容易得出它们的算法效率却大不同。算法一是线性检索，因此它的时间复杂度为 O(Size)，算法二是用折半法检索，因此它的时间复杂度是 $O(\log_2 Size)$，效率明显提高，特别是当 Size 特别大时，算法效率更为明显。

假设 Size＝100 000，由于 Size 是 $2^{16} < 100\ 000 < 2^{17}$，用折半法查找，最多查找 17