EDA 系统软件分析和设计方法学

讲义

2026 年春季

毕朝日

复旦大学集成电路与微纳电子创新学院

目录

第五讲　编程语言与面向对象设计

> The limits of my language mean the limits of my world.
>
> — 路德维希·维特根斯坦（Ludwig Wittgenstein）

语言是思维的边界。维特根斯坦这句名言用于哲学，却同样适用于编程。一个只会 FORTRAN 的工程师，看到的世界是由数组和循环构成的；一个精通 Lisp 的程序员，看到的世界是由函数和递归构成的；而一个面向对象的设计者，看到的世界是由对象和消息构成的。编程语言不仅仅是一种“实现工具”——它深刻地塑造着程序员思考问题的方式，正如自然语言塑造着我们对世界的认知。

上一讲我们讨论了软件错误与测试，得出结论：好的工程师不是不犯错的人，而是有能力系统性地发现和消灭错误的人。然而，发现和消灭错误只是“守”的一面。“攻”的一面——如何从一开始就构建出结构清晰、易于理解、易于修改的软件——则取决于我们选择什么语言来思考，用什么范式来组织代码。本讲将从编程语言的历史演进出发，追溯从机器语言到面向对象的抽象阶梯，然后深入面向对象设计的核心思想，最后通过一个完整的重构案例，展示如何在实践中运用这些原则将一团乱麻般的代码逐步变成优雅的设计。

5.1　编程语言：一部抽象的阶梯史

5.1.1　第一个程序员与“语言”的起源

理解编程语言，不能从语法规则开始，而要从人类为什么需要编程语言开始。

1843 年，英国诗人拜伦的女儿阿达·洛夫莱斯（Ada Lovelace）为查尔斯·巴贝奇（Charles Babbage）的分析机编写了一段计算伯努利数的程序。这是人类历史上第一个算法——尽管分析机从未被真正建造出来。洛夫莱斯不仅写出了程序，还提出了一个远超时代的洞见：分析机可以处理的不仅仅是数字，还有任何能用符号表示的事物——音乐、文字、图像。这一预言在一百多年后才被计算机科学证实。

洛夫莱斯面对的问题，也是此后所有程序员面对的问题：**如何把人脑中的算法，翻**

译成机器能执行的指令？这个翻译过程，就是编程语言存在的理由。

在计算机发展的最初阶段，程序员直接用二进制机器码编程。每一条指令都是 0 和 1 的序列，每一个内存地址都需要手工计算。这种方式精确、高效——也极其痛苦。1949 年，剑桥大学的莫里斯·威尔克斯（Maurice Wilkes）和他的团队为 EDSAC 计算机编写程序时，发现即使是最简单的计算任务，用机器码编程也容易出错且难以修改。威尔克斯后来回忆道："我清楚地记得那一刻的幻灭感——我意识到我的余生的很大一部分将花在寻找自己程序中的错误上。"

汇编语言是第一步抽象。它用助记符代替二进制编码——MOV 代替 01001000，ADD 代替 00000001——让程序员至少能读懂自己写的东西。但汇编语言仍然与特定的机器架构绑定：为 IBM 704 写的汇编程序，在 PDP-1 上毫无用处。程序员被锁定在硬件的细节里，无法在更高的层面思考问题。

5.1.2　FORTRAN：第一次解放

真正的突破发生在 1957 年。IBM 的约翰·巴库斯（John Backus）领导的团队发布了 FORTRAN——FORmula TRANslation，公式翻译。FORTRAN 的设计哲学极其务实：它要让科学家和工程师能用接近数学公式的方式编写程序，而不需要理解底层的机器指令。

巴库斯后来在图灵奖演讲中坦承，当初选择开发 FORTRAN 的动机很大程度上是出于"懒惰"——他实在不想继续用汇编语言编程了。这种"懒惰"驱动了计算机科学史上最重要的创新之一。FORTRAN 编译器能够将高级语言自动翻译为机器码，而且生成的代码效率接近手写汇编——这在当时被认为是不可能的。

FORTRAN 的成功证明了一个关键命题：**抽象不一定意味着低效。**好的抽象可以在提升人类生产力的同时，不显著牺牲机器效率。这一命题贯穿了此后所有编程语言的发展。

紧随 FORTRAN 之后，1958 年诞生了 ALGOL（ALGOrithmic Language）。如果说 FORTRAN 是工程实践的胜利，ALGOL 则是理论优雅的典范。ALGOL 引入了块结构、递归、词法作用域等概念，成为现代高级语言的思想源头。虽然 ALGOL 在商业上不如 FORTRAN 成功，但它的影响无处不在——后来的 Pascal、C、Java，都可以追溯到 ALGOL 的设计理念。

ALGOL 的影响还催生了编程史上最激烈的论战之一。1968 年，荷兰计算机科学家艾兹赫尔·迪杰斯特拉（Edsger Dijkstra）向《Communications of the ACM》投稿了一封信，原标题为"A Case against the GO TO Statement"，编辑尼克劳斯·维尔特（Niklaus Wirth）将它改为更具煽动性的"Go To Statement Considered Harmful"。这篇短文在编程界掀起了轩然大波。迪杰斯特拉的论点很简单：GOTO 语句使程序的控制流变得不可预测，应该用结构化的顺序、选择、循环三种控制结构来取代它。这场"结构

化编程”运动深刻地改变了编程语言的设计方向——此后诞生的主流语言，几乎都限制或取消了 GOTO 语句。有趣的是，“Considered Harmful”这个标题格式后来成了计算机科学论文的经典梗——此后出现了“Considered Harmful Considered Harmful”等一系列模仿之作。

5.1.3 C 语言：系统编程的利器

1972 年，贝尔实验室的丹尼斯·里奇（Dennis Ritchie）和肯·汤普森（Ken Thompson）创造了 C 语言。C 语言的诞生与 UNIX 操作系统的开发密不可分——事实上，C 就是为了重写 UNIX 而设计的。

C 语言的独特定位在于它是一种“高级的低级语言”。它提供了结构化编程的所有便利——函数、控制结构、类型系统——同时又允许直接操作内存地址和硬件寄存器。这种双重性格使 C 成为系统编程的首选语言，从操作系统内核到嵌入式固件，从数据库引擎到编译器本身，C 无处不在。

在 EDA 领域，C 语言的地位尤为特殊。早期的 EDA 工具——从 SPICE 电路仿真器到布局布线程序——几乎全部用 C 编写。加州大学伯克利分校的 SPICE 最初是 FORTRAN 代码，后来的商业版本逐渐迁移到 C。C 语言的高效率和对底层硬件的直接控制，使它特别适合处理 EDA 中常见的大规模数值计算和复杂数据结构操作。即便在今天，Synopsys 和 Cadence 的核心引擎中仍有大量 C 代码在运行。

然而，C 语言的强大也是它的危险所在。指针运算的灵活性同时带来了内存错误的温床——缓冲区溢出、悬挂指针、内存泄漏，这些问题困扰了 C 程序员半个世纪。上一讲我们讨论的许多测试和错误检测技术——Valgrind、AddressSanitizer——很大程度上就是为了对付 C 语言的内存安全问题。

一个有趣的细节是 *C* 语言名字的由来。*C* 的前身是汤普森创造的 *B* 语言，而 *B* 又脱胎于 *BCPL*（*Basic Combined Programming Language*）。所以 *C* 只是字母表上 *B* 的下一个。后来比雅尼·斯特劳斯特鲁普（*Bjarne Stroustrup*）在 *C* 的基础上添加面向对象特性时，将新语言命名为“*C++*”——这是 *C* 语言的自增运算符，暗示它是 *C* 的“下一个版本”。不过斯特劳斯特鲁普本人曾半开玩笑地说：“*C++* 这个名字本身就暗示了它的一个问题——自增运算发生在使用之后（后缀自增），意味着你先得到的是旧的 *C*，新的特性要等一等才生效。”

5.1.4 函数式编程：另一条道路

在命令式语言沿着 FORTRAN-ALGOL-C 的路线演进时，另一条完全不同的道路也在展开。

1958 年——与 ALGOL 同年——麻省理工学院的约翰·麦卡锡（John McCarthy）创造了 LISP（LISt Processor，列表处理器）。LISP 的设计灵感不是来自工程实践，而

是来自阿隆佐 • 邱奇（Alonzo Church）的 Lambda 演算——一种比图灵机更古老的计算理论模型。

LISP 的世界观与 FORTRAN 截然不同。在 FORTRAN 的世界里，程序是一系列改变内存状态的指令；在 LISP 的世界里，程序是一系列函数的组合，没有赋值语句，没有可变状态，计算就是函数对数据的变换。这种编程范式后来被称为**函数式编程**（Functional Programming）。

函数式编程的核心特征包括：以数学函数——定义域到值域的映射——为基础，用递归代替循环，用不可变数据结构代替可变状态。这些特征使得函数式程序天然具有某些优良性质：没有副作用的函数更容易推理和测试，不可变数据结构天然是线程安全的。

用一个简单的例子来说明范式的差异。求两个自然数的最大公约数，欧几里得算法的核心思想是：不断从较大数中减去较小数，直至两数相等。在 FORTRAN 66 中，这个算法依赖 GOTO 跳转和算术 IF 语句：

```
10 IF (NA-NB) 20, 40, 30
20 NB = NB - NA
GOTO 10
30 NA = NA - NB
GOTO 10
40 ...
```

在 C 语言中，结构化的控制流让代码变得清晰：

```
while (a != b) {
if (a > b) a -= b;
else b -= a;
}
```

而在 Haskell——LISP 的精神后裔——中，同一算法用模式匹配表达，读起来几乎就是数学定义本身：

```
gcd a b | a > b    = gcd (a - b) b
| a < b    = gcd a (b - a)
| otherwise = a
```

三段代码实现的是完全相同的算法，但它们反映了三种不同的思维方式：FORTRAN 用控制流的跳转来描述计算过程，C 用结构化的循环和分支来组织逻辑，Haskell 用递归的函数定义来表达数学关系。**语言不仅是工具，更是思维的框架。**

LISP 的后裔包括 Scheme（1975）、ML（1973）、Erlang（1986）、Haskell（1990）和 F#（2005）。虽然纯函数式语言从未成为工业主流，但函数式编程的思想深刻地影

响了主流语言的发展。今天的 C++ 有 lambda 表达式，Java 有 Stream API，Python 有 map/filter/reduce，JavaScript 的整个生态都建立在函数作为第一类对象的基础之上——这些都是 LISP 播下的种子。

说到 JavaScript，它的诞生过程本身就是编程语言史上最戏剧性的故事之一。1995 年，网景公司（Netscape）的管理层决定在浏览器中嵌入一种脚本语言，给了布兰登·艾克（Brendan Eich）仅仅十天的时间来设计和实现它。艾克在这十天里创造了一种混血语言——从 Scheme 借来了函数作为第一类对象，从 Self 借来了基于原型的面向对象，从 Java 借来了语法外观，从 Perl 借来了正则表达式。结果是一种充满矛盾和意外的语言——它有自动类型转换导致的奇葩行为（比如 `[] + [] === ""`），有令人困惑的 this 绑定规则，有全局变量污染问题。然而，正是这种“不完美”的语言，凭借其无处不在的浏览器平台，成长为今天使用最广泛的编程语言之一。JavaScript 的故事告诉我们：**一种语言的成功，往往不取决于它的设计优雅性，而取决于它所占据的生态位。**

在 EDA 领域，函数式思想同样有其用武之地。电路网表本质上是一个不可变的图结构，对它的各种分析——时序分析、功耗分析、等价性检验——都可以建模为对这个图的纯函数变换。一些现代 EDA 工具开始在其脚本层引入函数式风格，用声明式的方式描述设计约束和分析流程。

5.1.5 脚本语言：灵活性的胜利

编程语言演进的另一条重要支线，是脚本语言（或称动态语言）的兴起。

1987 年，拉里·沃尔（Larry Wall）创造了 Perl；1988 年，约翰·奥斯特豪特（John Ousterhout）创造了 Tcl（Tool Command Language）；1991 年，吉多·范罗苏姆（Guido van Rossum）创造了 Python；1993 年，罗伯托·耶鲁萨林斯基（Roberto Ierusalimschy）创造了 Lua；1995 年，布兰登·艾克（Brendan Eich）在十天内创造了 JavaScript。

这些语言有一组共同的特征：动态类型、灵活的数据结构、易于嵌入其他程序、支持多种编程范型。它们牺牲了运行效率，换取了开发效率。在脚本语言中，变量没有固定类型——同一个变量可以先存整数、再存字符串、最后存一个列表；数据结构可以在运行时动态增长和改变形状；函数是第一类对象，可以赋值给变量、作为参数传递、在运行时动态创建。

脚本语言的一个核心特性是**动态类型**（dynamic typing）。与 C++ 或 Java 要求在编译时确定每个变量的类型不同，Python 中的变量只是一个名字——它指向的对象有类型，但名字本身没有。这意味着同一个函数可以处理多种类型的输入，只要这些类型支持函数中使用的操作——这就是所谓的**鸭子类型**（duck typing）：“如果它走起来像鸭子，叫起来像鸭子，那它就是鸭子。”

在 EDA 行业中，Tcl 占据了一个独特的位置。几乎所有的商业 EDA 工具——Synopsys 的 Design Compiler、Cadence 的 Innovus、Mentor Graphics 的 Calibre——都

使用 Tcl 作为其脚本接口语言。这一传统始于 1990 年代：当时 EDA 工具需要一种轻量级的嵌入式脚本语言来实现命令行交互和流程自动化，Tcl 凭借其简洁的语法和出色的嵌入能力赢得了这个生态位。Python 在近年来逐渐进入 EDA 领域——Synopsys 的一些新工具开始支持 Python 接口，开源 EDA 生态（如 OpenROAD）也大量使用 Python——但 Tcl 的遗产仍然根深蒂固。对于 EDA 工程师来说，掌握 Tcl 几乎是一项必备技能。

围绕 Tcl 在 EDA 界的统治地位，一个长期的争论是：为什么不换成 Python？答案往往出人意料——不是技术问题，而是“二十年的 Tcl 脚本存量”。一家芯片设计公司可能积累了数万行 Tcl 流程脚本，从综合到布局布线到签核验证，每一步都嵌入了多年调试凝结的工艺经验和特殊处理逻辑。将这些脚本迁移到 Python 的成本和风险远超技术收益。这是软件工程中“路径依赖”（path dependence）的经典案例——最初的技术选择，即使不是最优解，也会因为生态积累而变得越来越难以替换。斯特劳斯特鲁普曾说：“世界上只有两种编程语言：一种是人人抱怨的，一种是没人用的。”Tcl 在 EDA 界的处境，正是这句话的完美注脚。

5.1.6　编程语言的选择：没有银弹

面对数百种编程语言，如何选择？答案取决于问题的性质。

面向特定领域的语言各有所长：科学计算用 FORTRAN，商务应用用 COBOL，人工智能用 Prolog，网络编程用 JavaScript 和 PHP，嵌入式脚本用 Tcl 和 Lua，统计分析用 R。**通用语言**试图在各个领域都表现良好：Ada、C、C++、Java、C#、Python 都属于这一类。**系统编程**——操作系统、工具软件、嵌入式系统——要求极致的运行效率，通常选择 C 或 C++。**快速原型开发**优先考虑开发速度，脚本语言是首选。

TIOBE 编程语言排行榜记录了编程语言流行度的变迁。2024 年的榜单上，Python 位居第一，C 和 C++ 紧随其后，Java 和 C# 分列四五位。值得注意的是，C 语言在 1989 年就位居榜首，三十五年来从未跌出前三——这是一种令人惊叹的生命力。而 Python 的崛起——从 1999 年的第 31 位到 2024 年的第 1 位——则反映了数据科学和人工智能对编程语言格局的深刻重塑。

编程语言的选择从来不是一个纯技术问题。它涉及团队的技能储备、现有代码的遗产、生态系统的成熟度、招聘市场的供给，甚至组织文化的惯性。在 EDA 行业，C++ 是核心引擎的首选，因为它兼顾了性能和抽象能力；Tcl 是流程自动化的事实标准，因为整个工具链都围绕它构建；Python 正在崛起，因为机器学习和数据分析的需求日益增长。一个成熟的 EDA 工程师，往往需要同时掌握这三种语言——用 C++ 写算法核心，用 Tcl 写工具脚本，用 Python 做数据分析和机器学习集成。

5.2 面向对象：从 Simula 到设计模式

5.2.1 面向对象的起源

如果说编程语言的历史是一部抽象阶梯的攀登史，那么面向对象编程（Object-Oriented Programming，OOP）就是这部阶梯上最重要的一级台阶。

面向对象的核心观念出奇地简单：**客观世界是由各种对象组成的，任何事物都是对象，复杂的对象可以由比较简单的对象以某种形式组合而成。**面向对象的软件系统同样由对象组成——软件中的任何元素都是对象，复杂的软件对象由比较简单的对象组合而成。

这种思想的起源可以追溯到 1967 年。那一年，挪威计算中心的奥利-约翰·达尔（Ole-Johan Dahl）和克里斯滕·尼加德（Kristen Nygaard）发布了 Simula 67 语言。Simula 最初是为了编写离散事件模拟程序而设计的——“模拟”（simulation）正是其名字的由来。在模拟程序中，现实世界的实体（汽车、顾客、服务窗口）需要在计算机中有对应的表示，每个实体有自己的状态和行为。达尔和尼加德发现，用“类”和“对象”来建模这些实体是最自然的方式。他们在 Simula 中引入了类（class）、对象（object）、继承（inheritance）和虚方法（virtual method）的概念——这些概念后来成为面向对象编程的基石。

1972 年，施乐帕洛阿尔托研究中心（Xerox PARC）的艾伦·凯（Alan Kay）创造了 Smalltalk。如果说 Simula 是面向对象的先驱，Smalltalk 就是面向对象的布道者。凯提出了一个激进的主张：程序中的**一切都是对象**——整数是对象，字符串是对象，甚至类本身也是对象；对象之间通过**消息传递**（message passing）来通信。凯后来给出了面向对象的经典定义：“OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.”——对我来说，面向对象仅仅意味着消息传递、状态的本地持有与保护和隐藏、以及所有事物的极端延迟绑定。

凯是一个思维天马行空的人，他的名言之一是：“预测未来的最好方式就是发明它。”另一句同样深刻的话是：“大多数声称做面向对象编程的人，其实只是在做面向类编程。”在凯看来，真正的面向对象的灵魂不在于类和继承这些语法糖，而在于对象之间通过消息实现的松耦合协作——每个对象就像一台小型计算机，只通过发送和接收消息与外界交互。这种理念后来在 Erlang 的 Actor 模型和微服务架构中得到了回响。

5.2.2 面向对象的三大支柱

从 Simula 和 Smalltalk 的思想中，逐渐凝练出面向对象编程的三大支柱：**封装**（Encapsulation）、**继承**（Inheritance）和**多态**（Polymorphism）。

封装是最基本的原则。它要求将数据和操作数据的方法绑定在一起，并对外界隐藏

内部实现细节。在 C++ 中，一个类通过 public 和 private 关键字明确区分对外接口和内部实现：

```
class Dog {
public:
Color color() const { return color_; }
void set_color(Color c) { color_ = c; }
void eat();
private:
Color color_;
};
```

外界只能通过公开的接口——`color()` 和 `set_color()`——与 Dog 对象交互，而不能直接触碰内部的 `color_` 成员变量。这种隔离有两重好处：其一，内部实现可以自由变更（比如改用字符串而非枚举来表示颜色），只要接口不变，所有使用者不受影响；其二，对象可以在接口方法中加入逻辑约束（比如在 `set_color` 中检查颜色值的合法性），保证对象始终处于一致的状态。

封装的本质，是对信息的**访问权限控制**。它将一个复杂系统分解为若干个“黑箱”，每个黑箱只通过严格定义的接口与外界交互。这与 EDA 工具中的 IP 核概念异曲同工——一个 IP 核对外提供标准化的接口（端口列表、时序约束），而内部的电路实现对集成者是不可见的。

继承允许新的类（派生类）基于现有的类（基类）来构建，自动获得基类的数据和方法。考虑一个简单的动物类层次：

```
class Animal {
public:
Color color() const;
void set_color(Color c);
void eat();
};
class Dog : public Animal {
public:
void bark();
};
class Bird : public Animal {
public:
void fly();
};
```

Dog 和 Bird 都继承了 Animal 的 `color()`、`set_color()` 和 `eat()` 方法，同时各自添加了特有的行为——`bark()` 和 `fly()`。继承实现了代码的**复用**：公共的行为只需要在基类中实现一次，所有派生类自动获得。

然而，继承不仅仅是代码复用的机制——更重要的是，它建立了类之间的**“是一个”**（is-a）关系。Dog“是一个”Animal，Bird 也“是一个”Animal。这种关系是多态的基础。

多态——字面意思是“多种形态”——使得同一个接口可以对应不同的实现。在 C++ 中，通过虚函数（virtual function）实现运行时多态：

```
class Animal {
public:
virtual void eat() { std::cout << "animal eat."; }
};
class Dog : public Animal {
public:
void eat() override { std::cout << "dog eat."; }
};
class Bird : public Animal {
public:
void eat() override { std::cout << "bird eat."; }
};
```

当我们有一个 Animal 指针数组，其中混合着 Dog 和 Bird 对象时：

```
Animal* zoo[100];
for (int i = 0; i < 100; ++i)
zoo[i]->eat();
```

每个对象会调用自己的 `eat()` 实现——Dog 对象输出“dog eat.”，Bird 对象输出“bird eat.”——尽管调用代码完全相同。这就是多态的力量：**派生类继承了基类的接口，但各自又提供了不同的功能实现**。调用者不需要知道具体对象的类型，只需要通过基类接口发送消息，多态机制会自动将消息路由到正确的实现。

多态在 EDA 软件中有着广泛的应用。以一个版图编辑器为例：图形元素包括矩形、多边形、圆弧、文字等多种类型，它们都继承自一个共同的 Shape 基类。当用户点击“绘制”按钮时，编辑器不需要用一长串 if-else 来判断当前选中的是什么类型——只需调用 `shape->draw()`，多态机制自动调用正确的绘制方法。当需要添加新的图形类型时，只需添加一个新的 Shape 子类并实现 `draw()` 方法，现有的代码完全不需要修改。这就是面向对象设计应对变化的基本策略。

事实上，整个 EDA 工具链的核心数据结构都深度依赖面向对象设计。以电路网表为例：一个芯片设计的网表包含数百万个元素——实例（Instance）、网络（Net）、端口（Port）、引脚（Pin）——它们之间存在复杂的层次和连接关系。在面向对象的建模中，Instance 和 Net 都继承自一个通用的 DesignObject 基类，共享名称管理、属性查询、迭代遍历等公共接口。Net 对象通过 Pin 与 Instance 建立连接关系，Port 定义了模块的外部接口。这种类层次使得上层的分析算法——无论是时序分析、功耗估计还是信号完整性检查——都可以通过统一的接口遍历设计，而不需要关心底层是标准单元、宏单元还是 IO Pad。

另一个典型的例子是时序分析引擎中的延迟模型。不同的工艺节点和不同精度要求下，延迟计算的方法差异巨大：线性延迟模型（Linear Delay Model）适用于早期估算，非线性延迟模型（NLDM）使用查找表插值，电流源模型（CCS/ECSM）则基于电流波形做精确仿真。通过多态，时序引擎定义一个 `DelayCalculator` 基类，不同的延迟模型作为子类各自实现 `calcDelay()` 方法。用户在 SDC 约束中指定精度要求，引擎在运行时选择对应的模型——整个调度过程对上层的时序图遍历算法完全透明。

EDA 软件之所以成为面向对象设计的天然试验场，根本原因在于它所建模的对象——电路——本身就是层次化、组件化、多态的。一个标准单元库中的 AND 门和 OR 门，天然地共享“逻辑门”的接口（输入引脚、输出引脚、延迟特性、功耗参数），却各自有不同的功能实现。面向对象的封装、继承、多态三板斧，几乎是为描述这种结构量身定做的。

5.2.3　面向对象设计：从分析到实现

面向对象不仅是一种编程范式，更是一种完整的软件设计方法学。面向对象的开发过程通常分为三个阶段：**分析**——提取和整理用户需求，建立问题域的精确模型；**设计**——将分析阶段得到的需求转换为系统实现方案；**实现**——编程、测试与调试。

在面向对象方法中，分析和设计没有明显的界限。许多分析结果可以直接映射为设计结果，而在设计过程中又往往会加深和补充对系统需求的理解，从而进一步完善分析结果。这与传统的瀑布模型形成鲜明对比——传统方法严格区分需求分析、总体设计、详细设计和编码实现，每个阶段有明确的交付物和评审节点；面向对象方法则允许在分析、设计和实现之间流畅地迭代。

面向对象设计可以进一步细分为**系统设计**和**对象设计**。系统设计确定实现系统的策略和目标系统的高层结构——将系统分解为若干子系统，定义子系统之间的接口和交互方式。一个典型的分层架构包括表示层、业务逻辑层和数据访问层，各层之间通过简单、明确的接口通信，尽量减少彼此的依赖性，使得每个子系统可以相对独立地设计和实现。对象设计则更加精细——确定类、关联、接口形式及实现服务的算法。

在对象设计中，类之间的**关联**是一个核心问题。在程序中，关联通常用指针或引用

类型的数据成员来实现。以雇员和公司的关系为例：如果只需要从雇员查到公司（单向关联），在 Employee 类中放一个 `Company*` 成员即可；如果还需要从公司查到所有雇员（双向关联），则 Company 类也需要一个 `vector<Employee*>` 成员。当关联本身携带属性——比如雇佣关系有薪资、入职日期等信息——最好的做法是引入一个独立的**关联类**（如 Employ），封装关联的属性和行为。

5.2.4 设计原则与设计模式

面向对象设计的实践中，逐渐总结出一系列指导原则。其中最重要的两条是**开放-封闭原则**（Open/Closed Principle，OCP）和**单一职责原则**（Single Responsibility Principle，SRP）。

开放-封闭原则要求：**软件实体——类、模块、函数等——对于扩展是开放的，对于修改是封闭的。**也就是说，当需求变化时，我们应当能够通过添加新代码来应对，而不是修改已有代码。这个原则之所以重要，是因为对已有代码的修改总是伴随着引入新 bug 的风险，而一个已经测试过、在生产环境中运行良好的模块，任何修改都可能破坏其稳定性。

单一职责原则更加朴素：**一个类应当只有单一的职责。**换言之，一个类应该只有一个“变化的理由”。如果一个类同时负责数据解析和界面渲染，那么解析逻辑的变化和界面需求的变化都会导致这个类被修改——两种互不相关的变化纠缠在同一个类中，增加了理解和维护的难度。

这些原则催生了**设计模式**（Design Patterns）的概念。1994 年，埃里希 • 伽马（Erich Gamma）、理查德 • 赫尔姆（Richard Helm）、拉尔夫 • 约翰逊（Ralph Johnson）和约翰 • 弗利赛兹（John Vlissides）——后来被称为“四人帮”（Gang of Four，GoF）——出版了 *Design Patterns: Elements of Reusable Object-Oriented Software*，提出了 23 种设计模式。设计模式不是发明，而是对以往设计经验的提炼——它们是软件设计中常见问题的**可重用解决方案**。

这里举两个在 EDA 软件中特别常见的设计模式。

Facade（门面）模式：为子系统的一组接口提供统一的、更高层的接口，使子系统更加容易使用。EDA 工具是 Facade 模式的重度用户。以综合工具为例，一次逻辑综合涉及 HDL 解析、技术映射、时序优化、面积优化、功耗优化等数十个子模块，每个子模块又有大量的配置参数。Facade 模式将这些复杂性封装在一个简洁的顶层接口之后——工程师在 Tcl 脚本中只需写 `compile_ultra` 一条命令，Facade 内部自动编排各子模块的执行顺序、传递中间结果、处理异常情况。同样，时序分析引擎内部涉及线延迟计算、单元延迟查表、时钟树分析、串扰分析等模块，Facade 提供 `report_timing` 接口将全部复杂性隐藏。

Template Method（模板方法）模式：在抽象类中定义一个算法的骨架，而将某

些步骤延迟到子类中实现。算法的结构不变，但具体步骤可以由子类重新定义。在 EDA 中，布局布线流程是 Template Method 的经典应用场景。布局算法的骨架是固定的——初始布局、全局布局、合法化、详细布局——但每一步的具体实现可以替换：全局布局可以用力导向算法（Force-Directed）、模拟退火（Simulated Annealing）或解析方法（Analytical Placement），详细布局可以用贪心交换或动态规划。Template Method 将算法骨架与步骤实现分离，使得研究人员可以替换其中一步而不影响整体流程。图形绘制中也有类似结构：Shape 基类定义 `draw()` 方法的骨架——先设置画笔、再绘制线条、最后填充颜色——其中 `draw_lines()` 由 Rectangle 和 Circle 子类各自实现。

除了这两种模式，EDA 软件中还大量使用 **Observer（观察者）模式**——当设计数据发生变化时（比如移动了一个单元），自动通知所有依赖该数据的视图和分析器更新自身状态；以及 **Iterator（迭代器）模式**——提供统一的接口来遍历网表中的实例、网络、引脚等不同类型的集合，而不暴露底层数据结构的实现细节（哈希表、链表或 KD-tree）。

设计模式的真正价值不在于提供现成的代码模板，而在于建立一套共同的设计词汇。当一个工程师说“这里我们用一个 *Facade*”，团队中的每个人立刻理解他的意图——无需冗长的解释。在 *EDA* 公司动辄数百万行代码的代码库中，设计模式是团队沟通和代码组织的基础设施。

5.3　重构：代码的持续进化

5.3.1　什么是重构

面向对象设计的原则和模式，说起来容易做起来难。在实际开发中，代码很少一开始就拥有完美的结构——它是在需求的不断变化和开发者认知的逐步深入中逐渐演化的。**重构**（Refactoring）正是应对这种演化的系统性方法。

重构的定义来自马丁·福勒（Martin Fowler）1999 年的经典著作 *Refactoring: Improving the Design of Existing Code*：重构是对软件内部结构的一种调整，目的是**在不改变软件之可察行为的前提下，提高其可理解性，降低其修改成本**。

注意定义中的关键限定：“不改变可察行为”。重构不是重写，不是添加新功能，不是修复 bug——它是对代码结构的纯粹改善。外部观察者看到的行为完全不变，变的只是内部的组织方式。这种约束使得重构比其他代码修改更加安全，因为它有一个明确的正确性判据：重构前后，所有测试用例的结果必须相同。

这引出了重构的**先决条件**：一个可靠的测试环境。没有测试，重构就像在没有安全网的高空走钢丝——任何结构调整都可能无意中改变行为，而你无法及时发现。这也是为什么敏捷过程提倡**测试驱动开发**（Test-Driven Development，TDD）——先编写测试定义代码的行为，再编写代码使测试通过，最后重构以消除重复和优化结构。测试是重构的安全网，重构是 TDD 的最后一步。

安德鲁 • 亨特（Andrew Hunt）和大卫 • 托马斯（David Thomas）在 *The Pragmatic Programmer* 中提出了一个生动的比喻：**破窗效应**。纽约市在 1990 年代的犯罪治理中发现，修好一栋建筑的第一扇破窗，能有效防止整栋楼的衰败。代码也是如此——一处未经清理的“烂代码”会发出信号：“这个模块已经没人在乎了，随便怎么写都行。”后续的开发者会不自觉地降低标准，在“烂代码”旁边堆砌更多的烂代码。重构的价值不仅在于改善当前代码的结构，更在于维护代码库的“社会契约”——向团队传递这样的信号：这里的代码质量值得被认真对待。

5.3.2 一个完整的重构案例

为了展示重构在实践中的运作方式，我们借用福勒书中的经典案例——一个影碟出租店的计费系统。

需求陈述：出租店向顾客出租电影碟片，需要打印报表，列出顾客的租片情况、租金总额以及积分。租金的计算同影片类型有关：普通片 2 天内 2 元，超出的天数每天 1.5 元；儿童片 3 天内 1.5 元，超出的天数每天 1.5 元；新片每天 3 元。常客积分每次租用 1 分，但新片租借 2 天以上计 2 分。使用文本格式输出报表，今后还会需要 HTML 格式。

系统涉及三个类：Movie（影片，有价格类别）、Rental（租借记录，有租期）、Customer（顾客，有租借记录列表和报表生成方法）。

第一版代码直截了当地实现了需求。Movie 类和 Rental 类是简单的数据容器，所有的业务逻辑——租金计算、积分计算、报表格式化——都塞在 Customer 的 `statement()` 方法中。这个方法用一个循环遍历所有租借记录，内嵌一个 switch 语句按影片类型计算租金，同时累加积分，最后拼接输出字符串。

这种“一个方法做所有事”的代码，在需求简单时工作得很好。但问题在于：如果现在要添加 HTML 格式的报表呢？`statement()` 方法中的租金计算逻辑和格式化逻辑纠缠在一起，你无法复用计算逻辑而只替换格式化部分。要么复制整个方法、修改其中的格式化代码（违反 DRY 原则），要么把这一坨代码拆开来重新组织。

在动手重构之前，第一步必须做的事情是**编写单元测试**。测试必须有自动检测的能力，不需要人工判断。我们为 Customer 的 `statement()` 方法编写测试，构造已知的输入（特定的影片、租期组合），验证输出字符串是否与预期完全一致。这些测试就是重构过程中的安全网——每一步重构之后，都要运行测试确认行为未变。

第一步重构：拆分 statement()。`statement()` 方法太长了，我们首先把计算单笔租金的代码提取为一个独立的方法 `amountFor()`。这是最基本的重构手法——**提取方法**（Extract Method）。提取后，`statement()` 中原来的 switch 语句被一个简洁的方法调用替代：`double thisAmount = amountFor(*each);`。运行测试，通过。

第二步重构：移动方法。仔细审视 `amountFor()` 的代码，它查询的是 Rental 的数据

（租期和影片类型），却放在 Customer 类中——这是一种“特性嫉妒”（Feature Envy）的代码坏味道。方法应该放在它使用数据最多的类中。我们将 `amountFor()` 移动到 Rental 类，改名为 `getCharge()`。Customer 中的调用变为 `each->getCharge()`。类似地，积分计算的代码也从 Customer 移到 Rental，命名为 `getFrequentRenterPoints()`。运行测试，通过。

第三步重构：消除临时变量。`statement()` 中还有两个循环内累加的临时变量——`totalAmount` 和 `frequentRenterPoints`。我们用查询方法替代它们：在 Customer 中添加 `getTotalCharge()` 和 `getTotalFrequentRenterPoints()` 方法，各自遍历租借记录并累加。这看起来增加了循环的次数（从一次变为三次），但换来的是更清晰的结构——`statement()` 方法现在只负责格式化输出，计算逻辑完全委托给了专门的方法。运行测试，通过。

此时，添加 HTML 报表变得轻而易举：

```
void Customer::htmlStatement(ostream& out) const {
out << "<H1>Rentals for <EM>" << getName()
<< "</EM></H1><P>\n";
for (const auto& each : _rentals) {
out << each->getMovie().getTitle() << ": "
<< each->getCharge() << "<BR>\n";
}
out << "<P>You owe <EM>" << getTotalCharge()
<< "</EM><P>\n";
out << "On this rental you earned <EM>"
<< getTotalFrequentRenterPoints()
<< "</EM> frequent renter points<P>\n";
}
```

`htmlStatement()` 和 `statement()` 共享所有的计算逻辑——`getCharge()`、`getTotalCharge()`、`getTotalFrequentRenterPoints()`——只有格式化部分不同。DRY 原则得到了完美的遵守。

第四步重构：将计算逻辑移到 Movie 类。目前，`getCharge()` 和 `getFrequentRenterPoints()` 在 Rental 类中实现，但它们的逻辑完全依赖于影片类型——这意味着如果未来增加新的影片类型或修改计价规则，需要修改 Rental 类。从职责分配的角度看，计价逻辑应该属于 Movie。我们将这两个方法移到 Movie 类，在 Rental 中保留代理方法：

```
double Rental::getCharge() const
{ return _movie.getCharge(_daysRented); }
```

这一步看似微小，实则为最后的关键重构埋下了伏笔。运行测试，通过。

第五步重构：用多态取代条件逻辑。现在 Movie 类中的 `getCharge()` 仍然包含一个 switch 语句，根据价格类别分支计算——这正是上一讲讨论的“不要评论不好的代码——重写它”的典型场景。问题在于，影片的价格类别**不是固定的**——一部新片过了上映期就会变成普通片。如果为每种价格类别创建 Movie 的子类（RegularMovie、NewReleaseMovie、ChildrensMovie），影片的类别变化就意味着对象需要在运行时改变自己的类——这在大多数面向对象语言中是不可能的。

解决方案是 **State（状态）模式**。我们不让 Movie 本身分裂为子类，而是引入一个 Price 抽象类，让不同的计价策略成为 Price 的子类——RegularPrice、NewReleasePrice、ChildrensPrice。Movie 持有一个指向 Price 对象的指针，将 `getCharge()` 和 `getFrequentRenterPoints()` 的调用委托给 Price。当影片类别变化时，只需切换 Price 对象，而 Movie 对象本身不变。

Price 基类定义虚方法，提供默认实现：

```
struct Price {
virtual double getCharge(int daysRented) const = 0;
virtual int getFrequentRenterPoints(int daysRented) const
{ return 1; }
};
```

各子类覆写计算逻辑。NewReleasePrice 不仅有不同的计费公式，还覆写了积分计算——新片租借超过 1 天得 2 分。RegularPrice 和 ChildrensPrice 只需覆写 `getCharge()`，积分计算沿用基类的默认行为。

一个工厂方法 `make_price()` 负责根据价格类别码创建对应的 Price 对象。Movie 的构造函数和 `setPrice()` 方法都通过这个工厂获取 Price 对象。

最终版本的类结构清晰地体现了各自的职责：Customer 负责报表格式化和汇总，Rental 负责租借关系的管理，Movie 负责影片信息和计价策略的持有，Price 层次结构负责具体的计价逻辑。每个类只有一个变化的理由——单一职责原则得到了满足；添加新的影片类型只需添加新的 Price 子类——开放-封闭原则得到了满足；switch 语句被多态调用替代——每一种条件分支变成了一个独立的类，逻辑清晰、易于测试、易于扩展。

这个从第一版到最终版的演变过程，浓缩了面向对象设计的精髓：不是一开始就设计出完美的结构，而是通过不断的小步重构，在测试的保护下，逐步发现并实现更好的设计。每一步重构都是安全的——测试保证行为不变；每一步重构都是有目的的——消除代码坏味道，让职责归位。这种“持续改进”的哲学，与敏捷开发的理念一脉相承，也与我们课程一直强调的 *DRY* 原则高度吻合——重构的终极目标，就是消除代码中一切不必要的重复。

5.3.3　三则重构

重构的节奏有一个朴素的经验法则，被称为**“事不过三，三则重构”**（Three Strikes And You Refactor）。

第一次做某件事时，直接做。第二次做类似的事时，你会对重复感到不适，但还是做了。第三次做类似的事时——停下来，重构。

这条法则的深意在于：它在“过早抽象”和“过度重复”之间找到了平衡点。第一次写某段代码时，你并不知道它是否会被重复——此时提取公共逻辑是过度设计。第二次遇到类似代码时，重复已经出现，但模式还不清晰——急于抽象可能抽象出错误的维度。第三次遇到时，模式已经确立，此时重构的方向最为明确，抽象的成本最低，收益最高。

这条法则也呼应了第二讲讨论的 YAGNI 原则——“You Aren’t Gonna Need It”——不要为假想的未来需求过度设计。让代码先“长”出来，等重复模式自然显现时再重构，而不是一开始就试图预测所有可能的变化方向。

在 EDA 开发实践中，“三则重构”的智慧随处可见。考虑一个布线引擎的开发：第一次实现最短路径搜索时，工程师直接在代码中硬编码了迷宫路由（maze routing）算法；第二次需要支持全局布线时，发现搜索策略不同但图遍历框架相似，忍一忍，复制了一份代码做修改；第三次要添加详细布线的多弯折路径搜索时——停下来，将搜索框架抽象为通用的图搜索引擎，迷宫路由、A* 搜索、协商式布线各自作为策略子类插入。如果在第一次就试图设计“通用搜索框架”，大概率会设计出一个既不通用也不高效的过度抽象；等到第三次，真实的变化维度已经清晰，抽象自然水到渠成。

EDA 工具的演进历史本身就是“三则重构”的宏观版本。SPICE 的第一版（1973 年）是一个单体程序，所有功能——网表解析、矩阵装配、方程求解、输出格式化——混在一起。SPICE 2 开始分离子模块但仍然高度耦合。到了 SPICE 3，经过大规模重构，才形成了清晰的模块化架构——解析器、器件模型、数值求解器各自独立，通过标准接口通信。类似的演化也发生在布局工具上：从 TimberWolf（1984 年）的单一模拟退火引擎，到现代布局工具中解析布局、力导向布局、合法化器等可插拔模块的组合——每一次架构升级都不是一步到位的，而是在多次遭遇类似问题后，才找到正确的抽象维度。

5.4　本讲总结

本讲从编程语言的历史出发，讨论了语言范式的演进、面向对象的核心思想，以及通过重构将思想付诸实践的方法。

编程语言如何塑造思维？从洛夫莱斯为分析机编写的第一个算法，到 FORTRAN 对科学计算的解放，到 C 语言在系统编程中的统治，到 LISP 开创的函数式传统，到脚本语言对灵活性的追求——编程语言的历史是一部抽象阶梯的攀登史。每一代语言都在前

一代的基础上提供更高层次的抽象，让程序员能够在更接近问题本身的层面上思考。语言的选择不是纯技术决策，它关乎团队能力、生态系统、遗产代码和组织文化。在 EDA 行业，C++ 承担核心引擎，Tcl 驱动流程自动化，Python 连接机器学习——三者各司其职，构成了 EDA 工程师的基本武器库。

面向对象为何成为 EDA 软件的主流范式？面向对象编程的三大支柱——封装、继承、多态——提供了管理大规模软件复杂性的有效武器。封装将电路元素的内部表示与外部接口隔离，继承建立起从通用图元到具体器件的类层次，多态使得同一个分析引擎可以无差别地处理不同类型的设计对象。设计原则（OCP、SRP）和设计模式（Facade、Template Method、State 等）将实践经验凝练为可传授、可讨论的设计词汇。

如何让代码持续进化？重构是在不改变外部行为的前提下改善代码内部结构的系统性方法。它的先决条件是可靠的测试环境，它的基本手法包括提取方法、移动方法、消除临时变量、用多态取代条件逻辑。影碟出租店案例从一个臃肿的 `statement()` 方法出发，经过五步重构，最终演化为职责清晰、多态驱动的设计——这个过程本身就是面向对象思想从理论走向实践的缩影。

贯穿本讲的底层哲学，是**“事不过三，三则重构”**。这条法则的深意远超重构本身——它是一种认识论立场：**承认我们无法在一开始就设计出完美的结构，但有信心在实践中逐步发现并实现更好的设计**。第一次写某段代码时直接做，第二次遇到类似代码时容忍重复，第三次再遇到时停下来重构——这个节奏在过早抽象和过度重复之间找到了平衡。它呼应了第二讲 YAGNI 原则的精神——不为假想的未来设计——却补充了 YAGNI 没有回答的后半句：**当重复真的出现时，要有勇气和方法去消除它。**

如果说第一讲的 DRY 原则回答了“不该做什么”，第二讲的 YAGNI 原则回答了“什么时候做”，第三讲的简单性原则回答了“怎么做”，第四讲的“不要评论不好的代码——重写它”回答了面对错误时的态度，那么本讲的**“事不过三，三则重构”**回答了代码演化的节奏——**让代码在使用中生长，在重复中发现模式，在重构中趋向优雅。**

课后思考

1. 你使用过的 EDA 工具（如 Vivado、Design Compiler、Innovus 等）的脚本接口是什么语言？试分析这种语言选择背后的历史和技术原因。如果让你从零开始设计一个 EDA 工具的脚本接口，你会选择什么语言？为什么？

2. 在本讲的影碟出租店重构案例中，最终版本用 State 模式取代了 switch 语句。试想：如果需求变化为“租金不再按影片类型计算，而是按租借时段（工作日/周末/节假日）计算”，当前的设计需要做什么修改？这种修改是否符合开放-封闭原则？

3. 函数式编程强调不可变数据和无副作用，面向对象编程强调封装可变状态。这两种范式看似矛盾，但现代语言（如 Scala、Kotlin、Rust）正在融合两者。你认为在 EDA 软件开发中，哪些模块更适合函数式风格，哪些更适合面向对象风格？

参考文献

[1] Wittgenstein, L., *Tractatus Logico-Philosophicus*, Routledge & Kegan Paul, 1922.

[2] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[3] Ritchie, D. M., "The Development of the C Language," *Proc. ACM SIGPLAN History of Programming Languages Conference*, pp. 201–208, 1993.

[4] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[5] Kay, A. C., "The Early History of Smalltalk," *Proc. ACM SIGPLAN History of Programming Languages Conference*, pp. 69–95, 1993.

[6] Dahl, O.-J. and Nygaard, K., "SIMULA: An ALGOL-Based Simulation Language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.

[7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[8] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[9] Martin, R. C., "The Open-Closed Principle," *The C++ Report*, vol. 8, no. 1, 1996.

[10] Stroustrup, B., *The C++ Programming Language*, 4th ed., Addison-Wesley, 2013.

[11] Ousterhout, J. K., "Scripting: Higher-Level Programming for the 21st Century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

[12] Hunt, A. and Thomas, D., *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.

[13] Knuth, D. E., "Structured Programming with go to Statements," *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, 1974.

[14] 赵文庆, 周学功, 《软件设计和开发》, 复旦大学出版社, 2013.