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

第三讲　软件工程方法

> Simplicity is prerequisite for reliability.
>
> — 艾兹赫尔 • 迪杰斯特拉（Edsger W. Dijkstra）

上一讲我们从底层向上，逐层剖析了软件设计平台的构成——从计算机硬件到操作系统，从编程语言到云计算。我们得出的核心结论是：平台的每一层都是对下层复杂性的封装和抽象，而 YAGNI 原则为这种封装划定了务实的边界。但平台只是地基。有了地基之后，一个根本性的问题浮出水面：**我们用什么样的方法和流程来组织软件开发，才能在有限的时间和资源内交付高质量的软件？**

这个问题并非从来就有。在计算机诞生的头二十年里，根本没有人觉得需要一套**方法**来开发软件——程序足够小，一两个聪明人就能搞定。方法论的需求是被逼出来的：当软件的规模膨胀到超越个体能力的极限时，当项目一个接一个地失败、超支、延期时，人们才意识到，光有好的工具和平台远远不够——还需要一套系统化的**方法**来组织人、管理过程、控制质量。

本讲将沿着历史的轨迹，讲述软件工程方法从混乱走向秩序的过程。第一节追溯软件危机的爆发和软件工程学科的诞生；第二节展示过程模型如何从僵硬的瀑布走向灵活的敏捷；第三节深入模块化设计的核心原则——耦合、内聚与信息隐藏。贯穿这三个主题的底层哲学，是本讲开篇迪杰斯特拉的那句话：**简单是可靠的前提。**

3.1　从危机到工程：软件方法论的诞生

3.1.1　一场迟来的危机

1960 年代中期，一场无声的危机正在整个计算机产业中蔓延。

这场危机不像金融危机那样有明确的爆发时刻，也不像自然灾害那样有清晰的破坏现场。它更像一种慢性病——项目一个接一个地超支，一个接一个地延期，交付的软件一个接一个地让用户不满意。IBM 的 OS/360 操作系统项目是这场危机的标志性事件。弗雷德 • 布鲁克斯（Fred Brooks）在 *The Mythical Man-Month* 中回忆了这个噩梦般的

经历：项目最终投入了超过 5000 人年的工作量，耗资数亿美元，交付时间比计划晚了数年，而交付的系统中包含了大约一千个已知的 bug。布鲁克斯在 IBM 的职位是 OS/360 的项目经理——他后来坦言，这段经历是他一生中最痛苦、也最有教育意义的工作。

OS/360 并非个案。1960 年代末，类似的故事在整个产业中反复上演。美国空军的后勤系统项目花了七年时间才部署，结果发现几乎无法正常使用。一项针对美国政府软件项目的调查显示：只有 2% 的软件能直接使用，3% 可以修改后使用，其余 95% 要么被废弃、要么从未交付。

为什么软件开发如此容易失败？回顾第一讲的讨论，我们已经知道了一部分答案：软件具有不可见性、高可变性、本质复杂性不可约简等特殊属性。但 *1960* 年代的软件危机揭示了一个更深层的原因——人们把软件开发当成了纯粹的**个人创造活动**，而非需要系统化管理的**工程活动**。一个天才程序员可以凭直觉写出一千行精妙的代码，但让一百个普通程序员协作开发一百万行代码，靠直觉是行不通的。

软件错误的积累效应使问题雪上加霜。在软件开发的各个阶段——分析、设计、编码、测试——都可能引入错误，而早期引入的错误会在后续阶段不断积累和放大。一个需求阶段的误解，到了编码阶段可能变成一个架构缺陷，到了测试阶段可能变成一组难以定位的 bug，到了发布之后可能变成一次代价高昂的召回。改正同一个错误，在需求阶段发现只需要花费 1 倍的代价，在设计阶段发现需要 3 倍，在编码阶段需要 5 到 10 倍，在系统测试阶段需要 10 到 15 倍，而在产品发布之后才发现，代价可能高达 10 到 100 倍。这个规律被称为**错误放大效应**，它是推动软件工程方法论发展的最根本的经济驱动力之一。

对 EDA 领域而言，错误放大效应更为致命。一个 EDA 工具中的 bug，其后果不仅是工具本身需要修复——如果这个 bug 导致芯片设计者做出了错误的设计决策，那么代价是一次失败的流片。在先进工艺节点上，一次流片的成本可能超过一千万美元，周期可能长达三到六个月。Synopsys 和 Cadence 等公司的工程师们深知这一点：他们的工具中的每一个 bug，都可能直接转化为客户的经济损失。

3.1.2　软件工程的诞生与七条原理

危机催生了行动。1968 年 10 月，北约（NATO）科学委员会在西德的加尔米施-帕滕基兴召开了一次会议，参会者约五十人，包括来自十一个国家的计算机科学家和产业界代表。会议的议题只有一个：如何解决软件危机？

这次会议的最重要产出不是任何技术方案，而是一个**名词**——“软件工程”（Software Engineering）。会议组织者故意选择了这个带有挑衅意味的名称：当时“编程”被许多人视为一种艺术或手艺，而非工程学科。将“软件”和“工程”放在一起，是要宣告一个立场：**软件开发必须像造桥、修路、建楼一样，遵循系统化的工程原则，而不能继续依赖个人的直觉和灵感。**

我们在第一讲中提到过玛格丽特·汉密尔顿（Margaret Hamilton），她在同一时期为“软件工程”这一术语的确立做出了重要贡献。汉密尔顿当时负责 MIT 仪器实验室的阿波罗飞行软件开发，她坚持用“engineering”来命名这一新兴学科，因为她希望软件开发能像传统工程学一样受到严肃对待。NATO 会议和汉密尔顿的努力共同奠定了软件工程作为一门独立学科的地位。

软件工程的正式定义由 IEEE 在 1993 年给出：把系统的、规范的、可度量的途径用于软件开发、运行和维护过程，也就是把工程应用于软件。这个定义有三个关键词值得注意：**系统的**意味着不是即兴发挥，而是有章法的；**规范的**意味着不是各行其是，而是遵循标准的；**可度量的**意味着不是靠感觉判断，而是用数据说话的。

1983 年，软件工程领域的先驱巴里·伯姆（Barry Boehm）综合了专家们陆续提出的一百多条准则，结合 TRW 公司多年的软件开发经验，提炼出了软件工程的**七条基本原理**。这七条原理至今仍是软件工程实践的基石：用分阶段的生命周期计划严格管理；坚持进行阶段评审；实行严格的产品控制；采用现代程序设计技术；结果应能清楚地审查；开发小组的人员应该少而精；承认不断改进软件工程实践的必要性。

这七条原理中，最容易被忽视的是最后一条——“承认不断改进的必要性”。它暗含了一个深刻的认识：软件工程本身也是一个不断演进的学科，不存在一劳永逸的“最佳实践”。伯姆在 *1983* 年认为正确的方法，到了 *2000* 年代可能需要修正。这种自我进化的意识，使得软件工程与传统的土木工程、机械工程有了本质的区别——后者的基本原理已经相对稳定了上百年，而软件工程的方法论每十年就会经历一次重大变革。本讲接下来要讲述的过程模型演进史，正是这种持续变革的最佳写照。

3.1.3　软件生命周期：一个隐喻的力量

软件工程确立之后，首先需要回答的是一个基本的组织问题：**软件开发的过程应该分为哪些阶段？**

“生命周期”这个隐喻提供了一个直觉上令人满意的答案。软件从诞生开始，经历成长、成熟，最终衰老失效，这段历程被称为**软件生命周期**。生命周期可以分为三个大的阶段：**定义**——确定要解决什么问题；**开发**——设计并实现解决方案；**运行维护**——交付使用并持续改进。

定义阶段包含问题定义、可行性研究和需求分析——搞清楚“需要解决什么问题”和“需要做什么”。开发阶段包含总体设计、详细设计、程序编码和单元测试——解决“如何做”的问题。运行阶段包含集成部署和维护——确保软件持续满足用户需求。在整个生命周期中，从计划、分析、设计、编码，一直到运行，需要不断评估、不断修改，周而复始。

IT 业界有一组广为流传的漫画，用一棵树上的秋千来隐喻软件生命周期中各方的认知偏差：客户描述的是一棵大树上挂着三层秋千——要求很多，但往往相互冲突、不

可得兼；项目经理的理解是一根绳子绑在树枝上——表面上正确，但永远抓不住真正的要点；分析员的设计化简为繁——原本简单的秋千变成了过度工程化的产物；程序员的实现什么都有了就是完全没法用——功能虽全但违反常识；参与测试的用户收到的东西难用至死；而客户真正需要的，其实只是一个轮胎系在树枝上。

这组漫画之所以流传了几十年仍不过时，是因为它揭示了软件开发中一个至今未被彻底解决的根本问题：**信息在传递过程中必然失真**。客户脑中的需求是模糊的、直觉性的；分析员将其转化为规格说明时会引入自己的理解偏差；设计者将规格说明转化为架构时会做出自己的取舍；程序员将设计转化为代码时又会做出自己的解读。每一次转化都是一次信息损耗。如何减少这种损耗？这正是后面要讨论的过程模型试图解决的核心问题。

软件工程学的组成体系也在这一时期逐渐成形。从学科结构上看，软件工程学包含两大分支：软件开发技术和软件工程管理。软件开发技术涵盖软件开发方法学——贯穿开发全过程的技术方法，包括传统的结构化方法和面向对象方法；软件工具——支持分析、设计、编码、测试的工具，如 CASE 工具、集成开发环境、版本控制系统；以及软件平台——第二讲讨论的内容。软件工程管理则涵盖软件管理学——进度安排、人员组织、质量保证；以及软件经济学——效益和成本估算。本讲聚焦的是软件开发方法学中最核心的两个议题：过程模型和模块化设计。

3.2　过程模型的演进：在秩序与灵活之间

软件生命周期告诉我们**要做什么**——定义、开发、维护——但没有告诉我们**怎么组织这些活动**。过程模型回答的正是这个问题：它是一个框架，规定了各项活动的顺序、应交付的文档资料、需采取的管理措施，以及标志各阶段任务完成的里程碑。

过程模型的演进史，本质上是一部**在秩序与灵活之间不断寻找平衡**的历史。太多的秩序导致僵化，太多的灵活导致混乱——好的过程模型需要在两者之间找到恰当的位置。

3.2.1　作坊模式：混乱的起点

在有任何正式的过程模型之前，软件开发遵循的是最原始的模式：实现第一版，修改直到客户满意，然后交付维护。没有需求文档，没有设计评审，没有测试计划——一切凭程序员的直觉和经验。

这种模式在 EDA 的早期历史中并不陌生。回顾第一讲提到的 SPICE 的诞生：拉里·内格尔（Larry Nagel）在伯克利开发 SPICE 时，基本上就是一个人写代码、调试、再写。这在 1970 年代初是可行的，因为 SPICE 的第一个版本大约只有几千行 FORTRAN 代码。但当 SPICE 演化为 SPICE2、SPICE3，代码量增长到数万行甚至更多，参与的开发者从一个人变成一个团队时，作坊模式就开始崩溃了——没有人能完全理解整个代

码库，修改一处往往在另一处引发意想不到的问题。

作坊模式的失败不是因为程序员不够聪明，而是因为它**不可规模化**——这和第二讲讨论的裸机编程的失败原因一模一样。人脑的工作记忆是有限的，当系统的复杂性超越个体认知能力的边界时，必须引入系统化的方法来分担认知负担。

3.2.2　瀑布模型：秩序的代价

1970 年，温斯顿 • 罗伊斯（Winston Royce）发表了一篇后来被视为“瀑布模型”奠基之作的论文[1]。瀑布模型将软件开发过程分为严格的顺序阶段：需求分析、系统设计、详细设计、编码实现、测试、交付维护。每个阶段的工作自顶向下，从抽象定义到具体实现，像川流不息、拾级而下的瀑布。

瀑布模型的核心设计思想可以归结为三点。其一是**控制复杂性**：将一个庞大的开发任务分割成有限的确定步骤，每一步都有明确的输入和输出。其二是**推迟实现**：在没有充分理解需求和设计之前，不急于编码——这与第二讲讨论的 YAGNI 原则一脉相承。其三是**质量保证**：每个阶段完成后都有评审，确保输出的文档完整、准确，只有前一阶段的输出正确，后一阶段的工作才能获得正确的结果。

瀑布模型在软件工程的早期发挥了巨大的积极作用。它第一次为软件开发引入了**纪律**——在此之前，很多项目的管理者甚至不知道项目处于什么阶段、还需要多久才能完成。瀑布模型用明确的里程碑和文档交付物回答了这些问题，使软件开发从“艺术”向“工程”迈出了关键一步。

然而，瀑布模型有一个致命的假设：**需求可以在项目初期被完整、准确地确定，并且在整个开发过程中保持不变**。这个假设在现实中几乎从不成立。

有一个具有讽刺意味的历史细节：罗伊斯在 *1970* 年那篇论文中其实明确指出，纯粹的瀑布模型是**有缺陷的**。他在论文中画出了线性的瀑布流程图之后，紧接着写道：“我相信这个概念是有根本性风险的，它会导致失败。”他随后提出了一个带反馈环的改进版本——允许相邻阶段之间的迭代。但后来的工业界几乎完全忽略了罗伊斯的警告，只采纳了他论文中最简单的线性模型。这是软件工程史上一个经典的“断章取义”案例——人们选择性地接受了他们想要的简单答案，而忽略了发明者自己的保留意见。

瀑布模型在 EDA 行业有着深刻的印记。1980 至 1990 年代，Synopsys、Cadence 和 Mentor Graphics 的早期产品大多采用瀑布式开发——先花数月制定详细的需求规格，再花数月设计架构，然后编码、测试、发布。每个主要版本的开发周期通常是 12 到 18 个月。这种节奏在当时是可以接受的，因为 EDA 工具的用户——芯片设计公司——自身的设计周期也是以年为单位的，对工具更新频率的要求不高。

但瀑布模型的弊端在 EDA 行业也表现得淋漓尽致。一个典型的场景是：市场部门

[1]Royce, W. W., “Managing the Development of Large Software Systems,” *Proc. IEEE WESCON*, pp. 1–9, 1970.

在年初收集客户需求，工程团队花了 12 个月开发出新版本，结果发布时发现客户的工艺节点已经从 28nm 推进到了 16nm，而新版本的某些关键功能是按 28nm 的设计规则开发的。需求在开发过程中已经发生了根本性的变化，但瀑布模型没有提供应对这种变化的机制。

表 3.1: 瀑布模型的优缺点

优点	缺点
使软件开发遵循严格的步骤，清晰定义各阶段的里程碑，有利于项目管理	需求分析和总体设计阶段的完成标准往往模糊不清
为系统产生完整的文档	文档可能给项目进展一种虚假的控制感
在进行下一阶段之前完成本阶段的文档	冻结每个阶段的结果与需求持续变化的现实相悖
仔细的项目计划	在生命周期早期制定的计划往往基于不充分的信息，有错误估计资源的风险

3.2.3　从迭代到螺旋：拥抱不确定性

瀑布模型的困境推动了新模型的探索。1980 年代到 1990 年代，一系列替代模型相继出现，它们的共同特征是：承认需求的不确定性，并将应对变化纳入过程本身。

迭代式开发将整个开发工作组织为一系列短小的、固定长度的小项目——每次迭代通常为 2 到 6 周——每一次迭代都包含需求分析、设计、实现与测试的完整过程。开发工作可以在需求被完整确定之前就启动，在每次迭代中完成系统的一部分功能，再通过客户的反馈来细化需求，开始新一轮的迭代。迭代式开发与瀑布模型的根本区别在于对“完整性”的态度：瀑布模型追求每个阶段的完整性——需求要全部确定才能开始设计；迭代式开发追求每次迭代的完整性——每次迭代交付一个可工作的、虽然不完整但可以被评估的系统。

PPT 中有一组用蒙娜丽莎来类比迭代式开发的图片：先画出整体的线条轮廓，再逐步添加细节和色彩，最终完成精细的作品。而不是先把左上角画到完美，再画右上角——那样做的风险是，画到一半才发现整体比例不对。

增量式开发与迭代式开发密切相关但有所不同。增量式开发将大型系统的开发过程分解为多个小的、可管理的部分，逐步构建和完善系统。每个增量都是系统的一个功能子集。与一次性开发整个系统相比，增量式开发更注重迅速交付部分功能，并允许在每

个增量中反馈和调整。如果说迭代式开发是“先画轮廓再填细节”，那么增量式开发更像是“先建一栋能住的小房子，再逐步扩建加盖”。

1986 年，巴里 • 伯姆提出了**螺旋模型**[2]，试图综合瀑布模型的系统性和迭代模型的灵活性。螺旋模型的独特贡献在于将**风险分析**引入了开发过程。模型以螺旋线的形式展开，每一圈代表一个开发阶段，每个阶段都包含四个象限的活动：明确本迭代阶段的目标、备选方案以及限制条件；对备选方案进行评估，明确并解决存在的风险，建立原型；当风险得到充分分析与解决后，进行本阶段的开发与测试；与客户一起对本阶段进行评审，并对下一阶段进行计划与部署。

螺旋模型的哲学启示在于：它第一次将“不确定性”从一个需要被消除的敌人，重新定义为一个需要被管理的**常态**。瀑布模型的隐含假设是“只要计划得足够好，就可以消除不确定性”；螺旋模型的立场是“不确定性是软件开发的本质属性，与其假装它不存在，不如正面应对它”。这一哲学转向对后来的敏捷运动有着直接的影响。

在 EDA 行业，螺旋模型的风险驱动思维有着天然的土壤。EDA 工具开发面临的最大风险之一是**算法风险**——一个布局布线算法在小规模测试用例上表现良好，但在客户的实际大规模设计上可能完全失败。螺旋模型鼓励在每个迭代的早期就用原型来验证关键技术风险，而不是等到整个系统开发完成后才发现核心算法不可行。

3.2.4　敏捷革命：从方法到文化

2001 年 2 月，十七位软件开发领域的思想领袖聚集在美国犹他州的雪鸟滑雪度假村。他们来自不同的方法学流派——极限编程（XP）、Scrum、水晶方法、自适应软件开发——但共享一个不满：当时主流的软件开发方法过于沉重，文档过于繁琐，流程过于僵化。经过两天的讨论，他们发表了一份只有 68 个英文单词的文件——*Manifesto for Agile Software Development*——敏捷软件开发宣言。

宣言的核心是四条价值观：**个体和交互**高于流程和工具；**可工作的软件**高于详尽的文档；**客户合作**高于合同谈判；**响应变化**高于遵循计划。宣言特别强调：“也就是说，尽管右项有其价值，我们更重视左项的价值。”

敏捷宣言的措辞极为精妙。它没有说文档是坏的、计划是坏的——它说的是在价值排序中，可工作的软件比文档更重要，响应变化比遵循计划更重要。这种“两者都要，但分优先级”的表述，避免了非此即彼的极端主义，也是敏捷运动能够被广泛接受的关键原因之一。

敏捷方法中最具影响力的实践框架是 **Scrum**。Scrum 将开发过程组织为一系列固定长度的“冲刺”（Sprint），通常为 2 到 4 周。每个冲刺开始时，团队从产品待办事项列表中选择一组要完成的功能，在冲刺期间集中精力实现这些功能，冲刺结束时交付一

[2]Boehm, B. W., “A Spiral Model of Software Development and Enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

个可工作的软件增量并进行回顾。Scrum 引入了三个关键角色：产品负责人（Product Owner）代表客户和业务需求；Scrum Master 负责确保团队遵循 Scrum 流程并移除障碍；开发团队是自组织的、跨职能的工作单元。

敏捷的另一个重要流派是肯特 • 贝克（Kent Beck）在 1990 年代末提出的**极限编程**（Extreme Programming，XP）。XP 的核心实践包括测试驱动开发（先写测试再写代码）、结对编程（两个程序员共用一台计算机）、持续集成（每天多次将代码集成到共享代码库）、重构（在不改变外部行为的前提下改善代码结构）。这些实践中的许多已经超越了 XP 本身，成为整个软件行业的通用实践。

敏捷运动的深层意义不仅在于提出了一组新的实践方法，更在于它推动了一场**文化变革**。传统的软件开发组织通常是科层制的——需求从上层传下，代码从下层传上，信息在层级之间逐级过滤和失真。敏捷倡导的是一种扁平化的、以团队为核心的组织文化——开发者直接与客户对话，团队自主决定如何组织工作，管理者的角色从“指挥者”转变为“服务者”。这种文化转型的难度往往远超技术转型的难度——很多组织引入了 *Scrum* 的形式（每日站会、冲刺回顾），但没有真正改变决策机制和权力结构，结果是“形似而神不似”的伪敏捷。

3.2.5　EDA 工具的开发实践：混合模型的现实

理论上的过程模型是清晰的，但现实中的 EDA 工具开发从来不会纯粹地采用某一种模型。更常见的情况是混合多种模型的元素，形成适合自身特点的实践。

EDA 工具开发有几个独特的特征，使得它不能简单地套用任何一种通用模型。第一是**算法密集性**。EDA 工具的核心竞争力在于算法——布局算法、布线算法、时序分析算法、逻辑优化算法。算法的研发往往具有探索性质：一个新算法可能需要数月的实验才能确定是否可行，这种不确定性比一般的业务软件开发高得多。第二是**工艺依赖性**。每一代新工艺节点都可能引入新的物理效应和设计规则，要求 EDA 工具做出相应的适配。工艺节点的更新周期约为两年，EDA 工具的开发节奏必须与之同步。第三是**验证的特殊性**。EDA 工具的正确性验证极其困难——对于一个布局算法，没有“标准答案”可以对照，只能通过与其他算法的比较、与硅片测量数据的对比来评估质量。

在实践中，大型 EDA 公司的开发流程通常呈现这样一幅混合图景：核心算法的研发采用**探索式迭代**——研究团队在小规模测试用例上快速实验多种算法方案，评估质量和性能，选择最优方案；产品化阶段采用**有计划的增量式**开发——按照客户需求的优先级，逐步将算法集成到产品中，每个季度发布一个包含新功能的版本；每个版本的发布前都有严格的**瀑布式**签核流程——回归测试、性能基准测试、客户 beta 测试，确保新版本不会引入退化。

这种混合模型反映了一个务实的认识：不同的活动适合不同的过程模型。算法探索需要灵活和迭代，适合敏捷的精神；产品交付需要纪律和可预测性，适合计划驱动的方

法；质量签核需要严格和全面，适合瀑布式的评审。**好的过程不是选择一种模型然后严格遵循，而是理解每种模型的适用场景，在正确的环节使用正确的方法。**

近年来，持续集成和持续交付（CI/CD）的实践也在 EDA 行业逐渐普及。传统的 EDA 工具发布节奏是每年一到两个主要版本；现代的实践是将代码持续集成到主干分支，每天运行自动化回归测试，使得任何时刻都可以从主干构建出一个可发布的版本。这种实践大幅缩短了从“bug 修复”到“客户可用”的周期，也使得工程师能够更频繁地获得关于自己代码质量的反馈。

RAH-66 科曼奇直升机的故事为需求管理提供了一个发人深省的反面教材。这是 1990 年代美军研制的下一代攻击侦察直升机，一些过于超前甚至相互矛盾的需求使得设计过于复杂——它既要比阿帕奇更容易装载进运输机，又要拥有 1260 海里的超长航程让它能自己飞抵海外战区。经费不断超支，研发一再延期，美军最终在 2004 年取消了整个项目。科曼奇的教训是：**需求不是越多越好，而是越简单越可靠**。无法克制地堆积功能需求，是软件项目和工程项目失败的共同根源。

表 3.2: 各种软件过程模型的比较

模型	技术特点	适用范围
瀑布模型	分阶段,阶段间有因果关系,各阶段完成后有评审，要求预先确定需求	需求明确且不易变更的系统
快速原型	不要求需求预先完备，支持用户参与和需求的渐进式确认	需求复杂、动态变化的系统
迭代模型	逐步获取用户需求、完善软件产品	需求难以确定、不断变更的系统
增量模型	增量式开发，允许开发活动并行和重叠	技术风险较大、需求较稳定的系统
螺旋模型	结合瀑布、迭代思想，引入风险分析活动	需求难以获取、风险较大的系统
敏捷方法	短迭代、自组织团队、持续交付、拥抱变化	需求快速变化、需要频繁反馈的系统

3.3 模块化设计：简单性的工程实现

过程模型告诉我们如何组织开发的时间和流程，但没有告诉我们如何组织软件本身的**结构**。如果说过程模型是“项目管理的方法论”，那么模块化设计就是“技术实现的方法论”。它回答的核心问题是：**一个大型软件系统应该如何分解为更小的单元，使得**

每个单元可以被独立理解、独立开发、独立测试？

3.3.1　分解的艺术：耦合与内聚

模块化的根据朴素而深刻：**把一个大的、复杂的问题分解成许多容易解决的小问题**。过程、函数、子程序、类、包——这些都是模块的不同表现形式。它们有一个共同特征：单独命名，通过名字来访问，各自完成一个子功能。

但“分解”这件事并不像听起来那么简单。同一个系统可以有无数种分解方式，质量却天差地别。用 PPT 中的 CPU 设计来类比：将 CPU 分解为寄存器组、运算器和控制器三个模块，每个模块有清晰的职责和简单的接口，这是高质量的分解——模块独立性好。但如果将 CPU 按“组合逻辑”和“时序逻辑”来分解，虽然逻辑上也说得通，但两个模块之间的信号交互极为复杂，几乎无法独立设计和测试——模块独立性差。

如何衡量一种分解方案的好坏？软件工程给出了两个定性标准：**耦合**和**内聚**。

耦合衡量的是不同模块之间相互依赖的紧密程度。软件设计应该追求尽可能松散耦合的系统。耦合从松散到紧密可以分为四个层次。**数据耦合**是最松散的形式——两个模块之间传递的信息仅仅包含数据，没有控制信息，没有共享的全局状态。**控制耦合**较为松散——两个模块之间传递的信息中包含控制信息，例如一个标志位告诉被调用模块应该执行哪种操作。**公共环境耦合**较为紧密——两个或多个模块通过某个公共数据环境相互作用，典型的例子是多个模块共享同一组全局变量。**内容耦合**是最紧密也最危险的形式——一个模块直接访问另一个模块的内部数据，或者不通过正常入口而跳转到另一个模块内部。耦合设计的准则可以概括为一句话：**尽量使用数据耦合，少用控制耦合，限制公共环境耦合的范围，完全不用内容耦合**。

内聚衡量的是一个模块**内部**各个元素彼此结合的紧密程度。高内聚意味着模块内的所有元素都服务于同一个明确的目标；低内聚意味着模块内塞进了本不相关的功能。内聚从低到高可以分为七个层次：偶然内聚——模块内的任务之间几乎没有关系，只是碰巧被放在了一起；逻辑内聚——模块内的任务在逻辑上相似，例如一个“处理所有输入”的模块；时间内聚——模块内的任务在同一段时间执行，例如一个“初始化所有子系统”的模块；过程内聚——模块内的处理功能存在时序关系；通讯内聚——模块内的任务使用同一个输入数据或产生同一个输出数据；顺序内聚——一个任务的输出直接作为另一个任务的输入；功能内聚——模块内的所有元素属于一个整体，完成一个单一的任务。设计的准则是：**力求做到高内聚，不使用低内聚**。

耦合和内聚看似是两个独立的概念，实际上是同一枚硬币的两面。一个高内聚的模块，由于其内部元素紧密围绕单一目标组织，与外部的交互自然就会减少——因此也倾向于低耦合。反过来，一个低内聚的模块，由于其内部功能杂乱，必然需要与外部进行大量的数据和控制交换——因此也倾向于高耦合。追求“高内聚、低耦合”，本质上就是追求**简单性**——每个模块做一件事，做好它，然后通过简单的接口与外部世界通信。这

正是迪杰斯特拉“简单是可靠前提”这一哲学的工程化表达。

在 EDA 工具的架构中，耦合与内聚的权衡无处不在。以一个物理设计工具为例：布局引擎、布线引擎、时序分析引擎是三个核心模块。理想的设计是让它们通过明确定义的数据接口通信——布局引擎输出标准单元的坐标，布线引擎读取坐标并生成互连线，时序分析引擎读取互连线的寄生参数并计算延迟。但在实际的高性能实现中，为了追求优化质量，这三个引擎往往需要共享大量的中间数据结构——时序驱动的布局需要在布局过程中调用时序分析，布线过程中需要实时更新时序信息。这种紧密的交互提高了优化质量，但也提高了耦合度，使得任何一个引擎的修改都可能影响其他两个。如何在优化质量和架构清晰度之间取得平衡，是 EDA 架构师面临的永恒挑战。

3.3.2 信息隐藏：帕纳斯的洞见

1972 年，大卫 • 帕纳斯（David Parnas）发表了一篇改变软件工程思维方式的论文[3]。在这篇论文中，帕纳斯提出了**信息隐藏**（Information Hiding）原则：每个模块应该将自己的设计决策隐藏在一个明确定义的接口背后，外部模块只能通过接口与之交互，不能依赖于其内部实现的任何细节。

帕纳斯通过一个简单的例子展示了这一原则的威力。他考虑了同一个系统的两种模块化方案：第一种按照处理步骤分解——输入模块、处理模块、输出模块；第二种按照可能变化的设计决策分解——每个模块封装一个可能变化的决策。两种方案都能正确实现系统功能，但在面对变化时表现截然不同。当数据格式发生变化时，第一种方案可能需要修改所有三个模块，因为它们都直接依赖于数据格式；第二种方案只需要修改封装了数据格式决策的那一个模块。

信息隐藏原则与第一讲讨论的 *DRY* 原则有着深层的联系。*DRY* 说的是“每一条知识在系统中应该有且仅有一个权威的表示”——不要让同一个设计决策分散在多个地方。信息隐藏说的是“每个设计决策应该被封装在一个模块中”——不要让一个决策的影响扩散到多个模块。两者从不同的角度表达了同一个核心思想：**将变化的影响范围最小化**。

信息隐藏在 EDA 工具设计中有一个典型的应用场景：**工艺技术文件的处理**。EDA 工具需要读取和处理各种工艺相关的参数——晶体管模型参数、互连线的寄生参数、设计规则等。这些参数的格式和内容随着每一代新工艺而变化。如果工具的各个引擎都直接解析原始的工艺文件，那么每次工艺更新都需要修改大量代码。正确的做法是将工艺文件的解析封装在一个独立的模块中，该模块向其他引擎提供统一的查询接口——“在给定的宽度和间距下，这条互连线的单位长度电阻和电容是多少？”——而不暴露工艺文件的内部格式。这样，当工艺文件格式变化时，只需要修改这一个模块。

[3]Parnas, D. L., “On the Criteria To Be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

3.3.3　从结构化到现代架构思维

1968 年，迪杰斯特拉发表了那封著名的致 *Communications of the ACM* 编辑的信——*Go To Statement Considered Harmful*[4]。他观察到，程序的质量与其中包含的 goto 语句数量成反比。这封信引发了一场持续多年的激烈辩论，但最终推动了**结构化程序设计**的普及：任意复杂的控制逻辑都可以用顺序、选择和循环三种基本结构来表达，程序应该只有一个入口和一个出口。

与迪杰斯特拉同时期，哈兰 • 米尔斯（Harlan D. Mills）在 IBM 推广了结构化编程的工业实践。他用图论的新结果证明了一个重要的定理：任意程序系统的控制逻辑都可以被设计和编码为高度结构化的形式，只需要使用有限的几种基本控制结构。这个定理为结构化程序设计提供了坚实的理论基础。

结构化方法不仅限于编码层面，它贯穿了软件开发的整个过程。在需求分析阶段，汤姆 • 迪马科（Tom DeMarco）和约登（Edward Yourdon）等人于 1978 年提出了**结构化分析**（Structured Analysis，SA）方法，其核心思想是**自顶向下、逐层分解**——将复杂的系统按层次分解，每一层只关注本层的抽象，将细节推迟到下一层处理。在设计阶段，对应的**结构化设计**（Structured Design，SD）方法将需求分析的逻辑模型按照逻辑功能划分模块，面向数据流组织模块之间的联结。在编码阶段，结构化程序设计确保代码只使用三种基本控制结构。

结构化方法的本质是**用纪律换取可理解性**。*goto* 语句之所以有害，不是因为它在机器层面做了什么坏事——*CPU* 的指令集里满是跳转指令——而是因为它破坏了程序的**局部可理解性**。当你看到一段结构化的代码时，你可以从上到下、从左到右地阅读，逻辑是线性展开的。但当代码中充斥着 *goto* 跳转时，你必须在脑中维护一张复杂的控制流图，认知负担急剧上升。还记得第一讲中米勒的 7 ± 2 吗？*goto* 语句的滥用使得程序员需要同时追踪的控制流分支远超工作记忆的容量。结构化编程的胜利，本质上是对人类认知局限性的尊重。

从 1990 年代开始，面向对象编程逐渐取代结构化编程成为主流范式——这将在后续的第五讲“编程语言与面向对象设计”中详细讨论。但结构化方法的核心原则——分解、抽象、模块独立性——并没有过时。面向对象是结构化思想的延伸和深化：类是模块的面向对象表达，封装是信息隐藏的面向对象实现，继承和多态是抽象和复用的面向对象机制。

在现代 EDA 工具的开发实践中，架构设计的关注点已经从单个模块的内聚和耦合，扩展到更大尺度的**系统架构**层面。一个大型 EDA 工具可能包含数百万行甚至上千万行代码，划分为数十个甚至上百个子系统。架构师需要在更高的抽象层次上回答同样的问题：哪些子系统应该是独立的？它们之间的接口应该如何定义？哪些决策可以被推迟？

[4]Dijkstra, E. W., “Go To Statement Considered Harmful,” *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

无论在何种尺度上，好的架构设计都遵循同一条原则：**越简单越可靠**。一个模块做的事情越少、接口越简单，它就越容易被正确实现、被充分测试、被他人理解。当你面对一个设计决策时，如果两种方案都能满足当前需求，选择更简单的那个。这不是偷懒，这是工程智慧——因为今天的简单代码，就是明天的可维护代码，就是后天的可扩展代码。复杂性是软件工程的头号敌人，而简单性是对抗它的最有力武器。

3.4 本讲总结

本讲沿着历史的轨迹，讲述了软件开发从无序走向有序的方法论演进。

软件工程如何诞生？1960 年代的软件危机——项目持续超支、延期、失败——迫使人们认识到，软件开发不能继续依赖个人英雄主义，而必须像传统工程学科一样，引入系统化的方法和纪律。1968 年的 NATO 会议和“软件工程”术语的确立，标志着这一转变的开始。错误放大效应——越早引入的错误，越晚才被发现，修复代价越高——成为推动方法论发展的核心经济动力。

过程模型如何演进？从瀑布模型的严格秩序，到迭代和螺旋模型对不确定性的拥抱，再到敏捷运动从方法到文化的革命——过程模型的历史是一部在秩序与灵活之间不断寻找平衡的历史。EDA 行业的实践表明，现实中的开发流程往往是多种模型的混合：算法探索需要迭代的灵活，产品交付需要计划的纪律，质量签核需要瀑布的严格。

软件如何被正确地分解？模块化设计通过耦合与内聚这两个互补的度量标准，指导我们将复杂系统分解为可独立理解和开发的单元。帕纳斯的信息隐藏原则将模块化从“功能分解”提升到“按变化分解”——每个模块封装一个可能变化的设计决策。结构化编程用纪律换取可理解性，消除了 goto 语句对认知的破坏。

贯穿这三个主题的底层哲学是迪杰斯特拉的信条：**简单是可靠的前提**。过程模型的演进，是在追求更简洁的组织方式——瀑布模型的问题不在于太简单，而在于它试图用一种僵化的流程来应对复杂多变的现实。模块化设计的核心，是将复杂系统分解为简单单元——高内聚、低耦合的本质就是让每个模块尽可能简单。结构化编程的胜利，是用三种简单的控制结构取代了无序的 goto 跳转。**在软件工程的每一个层面，简单性都不是偷懒的借口，而是专业精神的体现——因为只有足够简单的系统，才有可能是可靠的系统。**

如果说第一讲的 DRY 原则回答了“不该做什么”——不要重复自己，第二讲的 YAGNI 原则回答了“什么时候做”——面临确凿需求时才实现，那么本讲的简单性原则回答了“怎么做”——**用最简单的方式做正确的事**。三条原则共同构成了务实软件工程的哲学基础。

课后思考

1. 回顾你参与过的软件项目（课程项目或实习项目），它采用的是哪种过程模型？如果让你重新选择，你会做出不同的选择吗？为什么？
2. 以你熟悉的一个 EDA 工具或软件系统为例，分析它的模块划分是否符合“高内聚、低耦合”的原则。如果不符合，你认为应该如何改进？改进的代价和收益分别是什么？
3. 迪杰斯特拉说“简单是可靠的前提”，但现实中的芯片设计和 EDA 工具都在变得越来越复杂。你认为在复杂性持续增长的趋势下，“追求简单”这一原则还有可操作性吗？如何在不牺牲功能的前提下保持设计的简单性？

参考文献

[1] Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, Addison-Wesley, 1995.

[2] Royce, W. W., “Managing the Development of Large Software Systems,” *Proc. IEEE WESCON*, pp. 1–9, 1970.

[3] Boehm, B. W., “A Spiral Model of Software Development and Enhancement,” *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[4] Boehm, B. W., “Software Engineering Economics,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, pp. 4–21, 1984.

[5] Beck, K., et al., “Manifesto for Agile Software Development,” *agilemanifesto.org*, 2001.

[6] Beck, K., *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison-Wesley, 2004.

[7] Schwaber, K. and Sutherland, J., *The Scrum Guide*, 2020.

[8] Dijkstra, E. W., “Go To Statement Considered Harmful,” *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[9] Parnas, D. L., “On the Criteria To Be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[10] DeMarco, T., *Structured Analysis and System Specification*, Yourdon Press, 1979.

[11] Yourdon, E. and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.

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

[13] Farley, D., *Modern Software Engineering: Doing What Works to Build Better Software Faster*, Addison-Wesley, 2021.

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

[15] Naur, P. and Randell, B., eds., "Software Engineering: Report on a Conference Sponsored by the NATO Science Committee," Garmisch, Germany, 1968.