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

第一讲　软件质量与人机关系

> The purpose of software engineering is to control complexity, not to create it.
>
> — 帕梅拉 • 扎维（Pamela Zave）

人类文明的每一次跃迁，都伴随着工具革命。石器之于采集，青铜之于农耕，蒸汽机之于工业——每一种工具的出现，都不仅是技术的进步，更是人与世界关系的重新定义。今日之电子设计自动化，即 Electronic Design Automation，缩写为 EDA，正是信息时代最为关键的工具之一。它不直接面对终端消费者，不出现在手机屏幕上，也不占据新闻头条。然而，若没有 EDA，便没有芯片；没有芯片，则我们习以为常的一切，从手机到卫星，从医疗设备到人工智能，皆无从谈起。

本讲作为课程的开篇，将从三个层面展开讨论：何为 EDA，何为软件质量，以及人与软件之间应当建立怎样的关系。这三个问题看似独立，实则一以贯之——它们共同回答一个根本性的追问：**在一个复杂性不断膨胀的世界里，我们如何设计出值得信赖的软件？**

1.1　EDA：一部自动化的文明史

1.1.1　从手工到自动化：芯片设计的范式变迁

理解 EDA，不能从定义开始，而要从历史开始。正如钱穆先生论中国史，强调” 不知来路，无以知去处”，理解一个工程领域的当下状态，必须先理解它如何走到今天。

1958 年，杰克 • 基尔比（Jack Kilby）在德州仪器的实验室里，用一块锗片制造出了第一块集成电路，上面只有一个晶体管、三个电阻和一个电容。那一年，所谓” 设计”，就是一个工程师拿着铅笔和尺子，在坐标纸上画电路版图。每一条线的位置，每一个器件的摆放，都靠人脑计算、人手绘制。

这种手工模式维持了将近二十年。1965 年，仙童半导体的戈登 • 摩尔（Gordon Moore）发表了一篇四页的短文，观察到集成电路上的元件数量大约每年翻一倍。这一经验规律后来被称为” 摩尔定律”。摩尔定律不是物理定律——它没有数学证明，也没有

第一性原理的推导——它更像是一种工程上的” 自我实现的预言”：因为整个产业都相信它，所以整个产业都在向着它努力。

这里有必要澄清一个常见的误读。*摩尔在 1965 年原文中的表述是：集成电路上的元件数量约每年翻一倍。他在 1975 年修正为每两年翻一倍。需要强调的是，摩尔定律描述的是***经济最优***的集成度增长速率——即在单位成本最低的前提下，每块芯片能容纳的晶体管数量。它不是说技术上不能更快，而是说更快的增速在经济上不划算。*这一区分对于理解 EDA 的价值至关重要：EDA 的核心使命不仅是让设计” 能做”，更是让设计” 做得起”。

摩尔定律带来的直接后果是：芯片上的晶体管数量以指数速度增长，但人脑的处理能力并没有同步增长。1971 年 Intel 4004 有 2,300 个晶体管，一个工程师可以完全掌控。1985 年 Intel 80386 有 27.5 万个晶体管，已经需要团队协作。2017 年 NVIDIA V100 有 210 亿个晶体管，黄仁勋（Jensen Huang）公开表示其开发成本约 30 亿美元，动用了数千名工程师工作数年。

如果没有自动化工具的进步，摩尔定律早在 1980 年代就会失效——不是因为物理极限，而是因为设计复杂性将超越人类的管理能力。EDA 不是芯片产业的附属品，而是当代芯片产业存在的前提条件。

1.1.2　EDA 的定义与边界

EDA 的正式定义是：

> *EDA consists of a collection of methodologies, algorithms and tools, which assist and automate the design, verification, and testing of electronic systems.*
>
> EDA 是用来辅助和自动化电子系统之设计、验证和测试的方法学、算法和工具的集合。

这个定义有三个关键词，值得逐一展开。

方法学是指导设计过程的系统化方法。例如” 自顶向下设计” 就是一种方法学：先定义系统级规格，再逐步细化到模块级、电路级、版图级。方法学回答的问题是” 按什么顺序做”。**算法**是解决具体计算问题的数学过程。例如布局问题可以建模为一个带约束的优化问题，求解算法包括模拟退火、力导引方法、二次规划等。算法回答的问题是” 怎么算”。**工具**是将方法学和算法封装成可操作的软件产品。例如 Synopsys 的 Design Compiler 是一款综合工具，它将 RTL 描述转换为门级网表。工具回答的问题是” 用什么做”。

一个常见的误解是将 EDA 等同于” 画版图的软件”。事实上，现代 EDA 覆盖的子领域远比这广阔。IEEE 的 CEDA 将 EDA 分为八大方向：SoC 与 3D/2.5D SiP 的设计方法学，设计验证与确认，时序分析与功耗优化，RTL 与逻辑级及高层次综合，模拟

CAD、仿真、验证与测试，物理设计与验证，可制造性与可靠性设计，以及测试、确认与硅生命周期管理。这八个方向中的任何一个，都足以支撑数十年的研究工作。本课程不可能覆盖所有方向，但会从软件工程的视角，讨论设计这些工具时共通的方法论。

1.1.3 EDA 工具的技术演进

EDA 的技术演进可以分为五个时代。每个时代的特征不仅体现在工具本身，更体现在工具背后**设计哲学**的转变。

手工时代：1950–1965

这一时期的集成电路设计完全依赖人工。工程师使用红宝石刻刀在 Mylar 薄膜上手工切割版图，每一层掩模版都是物理实体。设计一块仅有几十个晶体管的芯片，需要数周的精细劳动。FORTRAN 语言于 1957 年问世，但计算机主要用于科学计算，尚未进入设计领域。电路分析依赖手工计算和查表。

编辑器时代：1965–1975

随着晶体管数量突破千级，手工设计的极限开始显现。这一时期出现了第一批版图编辑器和简单的设计规则检查工具。加州大学伯克利分校的 SPICE 于 1973 年发布，成为电路仿真的事实标准，至今仍在使用。

SPICE 的意义不仅在于技术层面——它确立了一个重要原则：**仿真先于制造**。在 SPICE 之前，工程师设计完电路就直接流片，错了就重来。SPICE 使得” 在计算机上先试一遍” 成为可能，这是 EDA 核心理念的萌芽。

SPICE 的前身是 1971 年的 CANCER，即 Computer Analysis of Nonlinear Circuits, Excluding Radiation。据开发者拉里 • 内格尔（Larry Nagel）回忆，导师唐纳德 • 彼德森（Donald Pederson）教授对这个名字不满意，认为用一种疾病命名软件不太妥当，于是改名为 SPICE。这个名字沿用至今，彼德森教授也被尊为”SPICE 之父”。值得一提的是，SPICE 是开源的——伯克利从未对其收取授权费。这一决策对整个 EDA 生态产生了深远影响：它让所有公司站在同一个起跑线上，也为后来的商业 EDA 工具提供了基准和灵感。

算法时代：1975–1990

这是 EDA 从” 辅助工具” 升级为” 核心基础设施” 的关键时期。

1980 年，卡弗 • 米德（Carver Mead）和林恩 • 康威（Lynn Conway）合著的 *Introduction to VLSI Systems* 出版，系统地阐述了如何用结构化方法设计大规模集成电路。这本书的影响力类似于高德纳（Donald Knuth）的 *The Art of Computer Programming* 之于算法学——它奠定了一个学科的基本框架。

同一时期，三家将主导 EDA 市场数十年的公司相继成立。1981 年 Mentor Graphics 在俄勒冈州成立，创始人汤姆·布鲁格（Tom Bruggere）等人从泰克公司出走，专注于 PCB 和系统级设计。1986 年 Synopsys 在北卡罗来纳州成立，次年迁至加州山景城，阿尔特·德格斯（Aart de Geus）从通用电气带出逻辑综合技术，发展出业界首款商业化逻辑综合工具 Design Compiler。1988 年 SDA Systems 和 ECAD 合并为 Cadence，后来发展成为 IC 设计实现和模拟/混合信号设计领域的领导者。

在算法层面，这一时期最重要的突破是自动布局布线的实用化。加州大学伯克利分校的卡尔·塞肯（Carl Sechen）在 Alberto Sangiovanni-Vincentelli 教授指导下开发的 TimberWolf 使用模拟退火算法进行标准单元布局，首次在质量上接近甚至超越人工布局，这一成果发表于 1985 年的 *IEEE Journal of Solid-State Circuits*，是 EDA 史上的里程碑事件之一。

模拟退火算法的引入值得深入讨论。模拟退火借鉴了冶金学中金属退火的物理过程：将金属加热到高温使原子自由移动，然后缓慢冷却，让原子有时间找到能量最低的稳定排列。在布局问题中，”温度”控制着算法接受劣解的概率——高温时大胆探索，低温时精细调优。这种”先粗后细”的策略至今仍是组合优化的核心思路。

模拟退火的成功揭示了一个深刻的道理：好的算法不是找到最优解，而是在可接受的时间内找到足够好的解。布局问题是 *NP-hard* 的，理论上不存在多项式时间的精确算法。工程上的智慧在于接受”够用即可”的哲学，这与软件质量中”功能正确”与”性能精确”之间的权衡一脉相承。

平台化时代：1990–2010

随着芯片复杂度突破百万门级，单点工具已不足以支撑设计流程。EDA 进入”平台化”阶段：各家厂商开始构建涵盖从 RTL 到 GDSII 的完整设计流程。Synopsys 的 Galaxy 平台、Cadence 的 Innovus 平台，都是这一趋势的产物。

这一时期有四项技术变革尤为深刻。静态时序分析取代动态仿真成为时序签核的主要手段，其核心思想是将时序验证从”穷举所有输入组合”简化为”分析所有路径的最坏情况”，将指数复杂度降为多项式复杂度。物理综合打破了传统的”先综合后布局”的线性流程，将逻辑综合和物理信息交叉优化——这是一个方法学的变革：承认抽象层次之间的信息壁垒会导致次优解，从而主动打破这些壁垒。时序驱动布局布线将时序约束直接嵌入布局布线的目标函数中，使得物理实现结果能直接满足时序需求。光学邻近效应校正即 OPC 和可制造性设计 DFM 标志着 EDA 开始向制造端延伸——当特征尺寸接近光刻波长时，版图上画的形状和硅片上实际转印的形状不再一致，OPC 通过预畸变版图来补偿光学效应，这本质上是一个逆问题的求解。

智能化时代：2010 至今

2010 年代开始，机器学习逐步渗透 EDA 的各个环节。早期应用集中在参数调优——用机器学习模型替代计算昂贵的物理仿真来预测设计质量，例如用神经网络预测布局后的时序结果，将原本需要数小时的静态时序分析缩短到秒级。

2021 年，Google DeepMind 在 *Nature* 上发表了一篇引发广泛争议的论文[1]，声称其强化学习算法可以在 6 小时内完成芯片布局，质量超过人类专家数月的工作。这篇论文引发了学术界的激烈辩论：支持者认为它标志着 AI 在 EDA 中的突破，批评者则指出其实验设置存在问题——用于比较的人类基线并非行业最佳实践，且算法在更复杂的设计上表现不稳定。2023 年，多位学者对此提出了系统性质疑。加州大学圣迭戈分校的钟冠寰（Chung-Kuan Cheng）和安德鲁 • 卡恩（Andrew B. Kahng）等人在 ISPD 2023 上发表了评估论文[2]，密歇根大学教授伊戈尔 • 马尔科夫（Igor Markov）也在 *Communications of the ACM* 上发表了系统性反驳[3]。Google 随后发表了回应，承认部分批评合理但坚持核心结论。这场学术论战至今未有定论。

这个争论本身就很有启发性。*AI 能否替代 EDA 工程师？这个问题的答案取决于你如何定义”替代”。如果”替代”意味着在所有设计场景下都能独立完成从规格到 GDSII 的全部工作，那么答案在可预见的未来是否定的。但如果”替代”意味着将工程师从重复性的调参和迭代中解放出来，让他们专注于架构决策和创新，那么这种替代已经在发生。更本质的问题是：当 AI 越来越擅长”执行”时，人类工程师的不可替代性在哪里？*这个问题我们在本讲的第三部分会深入讨论。

近年来值得关注的另一个趋势是开源 EDA 的崛起。以 DARPA 资助的 OpenROAD 项目为代表，开源工具链正在从学术实验走向可用的工业级流程。OpenROAD 的目标是实现 RTL 到 GDSII 的完全自动化、无人干预设计流，已经在若干测试芯片上验证了可行性。Yosys 作为开源综合器、KLayout 作为开源版图编辑和 DRC 工具、Magic 作为开源版图工具、ngspice 作为开源 SPICE 仿真器，共同构成了日益成熟的开源生态。开源 EDA 的意义不仅在于降低成本——商业 EDA 工具的年度许可费通常在数十万到数百万美元的量级——更在于它使得 EDA 教育和研究不再受限于商业工具的黑盒化。学生可以阅读、修改、扩展这些工具的源代码，真正理解算法是如何工作的。

1.1.4 EDA 的知识体系

理解 EDA 的知识结构，需要一张” 地图”。

[1]Mirhoseini, A. et al., “A graph placement methodology for fast chip design,” *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[2]Cheng, C.-K., Kahng, A. B., Kundu, S., Wang, Y., and Wang, Z., “Assessment of reinforcement learning for macro placement,” *Proc. ISPD*, 2023.

[3]Markov, I. L., “Reevaluating Google’s reinforcement learning for IC macro placement,” *Communications of the ACM*, 2024.

顶层是面向用户的 EDA 工具：提取、仿真、静态时序分析、布局布线、综合、等价性检查、形式验证。这些是工程师日常使用的产品。中间层是支撑工具的核心算法：参数提取、电路分析、时序分析、布局、布线、逻辑优化、模型检查，每一个算法都对应一类数学问题——图着色、整数线性规划、布尔可满足性、约束求解等，近年来机器学习和搜索算法也作为新的交叉技术进入了这一层。底层是数学基础，分为两大分支：连续数学支撑着模拟电路仿真、参数提取等需要处理连续变量的问题，涉及非线性求解器、快速线性求解器和连续优化；离散数学支撑着数字电路综合、验证、测试等处理离散结构的问题，涉及离散优化、组合算法、形式语言与自动机理论、数理逻辑与语义学。

这三层结构揭示了 *EDA* 的一个本质特征：它是一个数学密集型的软件领域。与 Web 开发、移动应用等主要处理数据流和用户交互的软件不同，EDA 工具的核心竞争力在于算法——更快的求解速度、更高的解质量、更好的收敛性。这意味着 EDA 软件的质量标准与一般软件有显著差异，我们将在下一节详细讨论。

1.1.5　芯片设计的当代全景

为了让抽象的 EDA 概念落地，我们来看看当代芯片设计的实际面貌。

表 1.1: 芯片设计四大领域的特征比较

	模拟设计	数字设计	板级设计	封装设计
输入类型	原理图	Verilog 网表	原理图	Excel 表格
信号规模	1K–100K	10M–1B	1–10K	∼1K
自动化程度	EDA 辅助人工	99% 自动化	EDA 辅助人工	100% 人工
团队规模	2–4 人	1–100 人	2–4 人	2–4 人
设计周期	3–18 月	3–18 月	3–6 月	2–8 周

这张表中隐含着几个值得深思的事实。

数字设计的自动化程度最高，达到 99%，但团队规模也最大，最多可达 100 人。这看似矛盾——如果 99% 都自动化了，为什么还需要这么多人？答案是规模。一个 10 亿门级的设计，即便 99% 自动化，剩下的 1% 仍然对应 1000 万个需要人工关注的信号。自动化解决了” 能不能做” 的问题，但” 做得好不好” 仍需人工把控。

模拟设计的自动化程度远低于数字设计。这不是因为模拟设计工程师抵制自动化，而是因为模拟设计的本质更难以形式化：一个运算放大器的设计涉及数十个相互耦合的性能指标——增益、带宽、噪声、线性度、功耗——它们之间的关系通常是非线性的，难以用简单的目标函数描述。这是 EDA 研究的前沿方向之一。

封装设计仍然是 100% 人工，输入甚至是 Excel 表格。这提醒我们：即便在高度自动化的芯片产业中，仍然存在自动化的洼地，等待被填补。

1.2 软件质量：在矛盾中寻求平衡

1.2.1 为什么软件和硬件不一样

在讨论软件质量之前，我们需要先理解一个根本性问题：**软件为什么这么难做好？**

这个问题不是修辞。汽车工业经过一百多年的发展，一辆出厂的汽车基本可以保证” 按预期工作”——方向盘向左转车就向左拐，踩刹车车就停下来。但一款出厂的软件呢？Windows 操作系统每次大版本更新几乎必然伴随大量 bug；大型 EDA 工具的每个 release 版本都附带一份已知问题列表；即便是 Google、Amazon 这样拥有顶尖工程团队的公司，也会因为软件故障导致全球性的服务中断。

1996 年 6 月 4 日，欧洲航天局的阿丽亚娜 5 号火箭在发射 37 秒后爆炸，直接经济损失超过 3.7 亿美元。事故原因是飞行控制软件中的一个整数溢出错误：一个 64 位浮点数被转换为 16 位有符号整数时溢出，导致导航系统输出错误数据，飞行姿态失控。

这起事故有几个令人深思的细节。首先，那段导致溢出的代码来自阿丽亚娜 4 号——它在阿丽亚娜 4 号上运行了多年，从未出过问题。问题在于阿丽亚娜 5 号的飞行参数范围超出了阿丽亚娜 4 号的设计假设：5 号的水平速度更大，导致变量超出了 16 位整数的表示范围。其次，这段代码在发射后其实已经不需要了——它属于发射前的校准程序，在起飞后应当被关闭，但为了” 保持与阿丽亚娜 4 号的一致性”，工程师决定让它继续运行。第三，异常处理程序确实捕获了这个溢出，但它的处理方式是关闭整个惯性导航系统——因为在阿丽亚娜 4 号的设计中，这种异常被认为不可能在正常飞行中发生，所以最安全的处理方式是停止导航并切换到备份系统。然而，备份系统运行的是同一份代码，于是也崩溃了。

这起事故揭示了软件工程中三个深层问题。其一是复用的风险：经过验证的代码在新环境下可能失效，因为隐含假设改变了。其二是防御性编程的悖论：异常处理代码本身可能成为故障源。其三是系统思维的缺失：单独看每个组件都是合理的，但组合在一起就出了问题。

IT 业界有一个广为流传的段子，常被归于比尔 • 盖茨（Bill Gates）：” 如果通用汽车像计算机产业一样进步，我们现在开的车售价 25 美元，每加仑油跑 1000 英里。” 这话的言外之意是计算机产业的进步速度远超传统工业。但通用汽车如果真的像软件产业一样运作，情况可能是这样的：你的车可能毫无预兆地每天抛锚两次；每次公路标线重画你就得换一辆新车；气囊弹出前会弹出对话框问你” 确定要弹出吗？”；你需要按” 启动” 按钮来关闭发动机。

这个类比虽然幽默，但它指向一个严肃的事实：**软件的本质与物理产品有根本区别，这些区别决定了软件质量问题的独特性。**

1.2.2　软件的本质特殊性

软件之所以与硬件不同，根源在于四个本质特征。

零边际成本与非磨损性

软件一旦开发完成，复制的边际成本几乎为零——复制一份软件只需要几秒钟的磁盘操作。这与物理产品截然不同：每多生产一辆汽车，都需要钢铁、橡胶、能源和人工。同时，软件不会”磨损”。一辆汽车开十万公里后，发动机、轮胎、制动系统都会老化；但一份软件运行十年后，它的代码不会变——如果第一天能正确运行，第十年还是能正确运行，前提是环境不变。

这听起来是优点，但它导致了一个反直觉的后果：**软件的故障模式与硬件截然不同。**硬件的故障曲线是经典的”浴缸曲线”：早期故障率高，中间阶段稳定，后期故障率再次升高。软件没有磨损期，理论上故障率应该随着 bug 的修复而单调递减。但现实中的软件故障曲线是锯齿状的——每次修改，无论是修 bug 还是加功能，都可能引入新的故障，导致故障率跳升。这就是著名的软件退化现象。

复杂性不可约简

弗雷德·布鲁克斯（Fred Brooks）在 1987 年的经典论文 *No Silver Bullet* 中区分了两种复杂性。**本质复杂性**源于问题本身——例如布局布线问题是 NP-hard 的，这种复杂性无法通过更好的工具或语言来消除，它是问题的内在属性。**偶然复杂性**源于工具、语言或流程的不完善——例如 C 语言中的手动内存管理带来的 bug，可以通过使用带垃圾回收的语言来消除。布鲁克斯认为，软件工程在过去几十年的大部分进步都是在减少偶然复杂性。但本质复杂性是不可约简的——它随着问题规模的增长而增长，而且增长方式通常不是线性的。

对于 EDA 软件而言，本质复杂性尤其显著。一个时序分析引擎需要同时处理数十亿条路径，需要考虑工艺变异、电压降、串扰、温度效应等多种物理效应，这些因素相互耦合，构成了一个极其复杂的多维优化问题。*这种复杂性不是软件写得不好造成的，而是问题本身就那么复杂。*

不可见性

布鲁克斯还指出了软件的另一个本质属性：不可见性。一栋建筑有蓝图，一块电路板有原理图，一辆汽车有三维模型——这些物理系统都有直观的空间表示。但软件没有自然的空间结构。代码是文本，架构图是人为抽象，运行时状态是动态变化的。这意味着软件系统的全貌无法被一张图完整表达。我们可以画类图、序列图、状态图、部署图——但没有任何一种图能够完整地描述一个软件系统的所有方面。这使得软件设计的”

概念完整性” 极其难以维持，尤其是在大规模团队开发中。

可变性

软件之所以叫” 软件”，就是因为它” 软”——容易修改。这既是优势也是诅咒。优势在于我们可以快速迭代、快速响应需求变化；诅咒在于每次修改都可能破坏已有的功能。在 EDA 工具的开发实践中，一个典型的场景是：客户 A 报告了一个时序分析的 bug，工程师修复后，客户 B 发现他的设计结果变差了——因为那个”bug” 恰好让客户 B 的设计受益。这种情况在 EDA 行业比比皆是，被戏称为” 一个人的 bug 是另一个人的 feature”。

1.2.3 软件发展的四个历史阶段

理解了软件的本质特殊性之后，我们可以更好地理解软件发展的历史轨迹。这段历史可以分为四个阶段，每个阶段都是对前一个阶段所暴露问题的回应。

第一阶段是 1957 年之前的**程序设计阶段**。工程师使用汇编语言甚至机器码编程，没有操作系统，没有调试工具，程序运行在穿孔卡片上。这一时期的” 软件” 通常是一次性的——为解决一个特定计算问题而编写，用完即弃。质量不是一个独立的关注点，因为程序足够小，一个工程师可以完全掌控。

第二阶段是 1958 年到 1967 年的**软件设计阶段**。FORTRAN、COBOL、LISP 等高级语言的出现标志着这一阶段的开始。程序设计从翻译机器码升级为描述解题过程：建模、求解方法、实现。操作系统的雏形出现了，程序开始有了” 用户” 的概念——不再只是编写者自己使用，而是需要交付给他人。软件的” 易用性” 问题首次浮出水面。

第三阶段是 1968 年到 1985 年的**软件工程阶段**。1968 年，NATO 在德国加尔米施召开了一次具有历史意义的会议，议题只有一个：软件危机。与会者面对的现实是：软件项目的成本和进度几乎无法准确估计，用户对” 已完成的” 软件系统经常不满意，软件质量难以保证，软件维护困难且昂贵，软件缺少完整的文档，软件成本在整个计算机系统中的占比急剧上升，软件生产效率低下、供不应求。令人感慨的是，将近六十年后的今天，这些问题中的大多数仍然存在——只是程度有所缓解，表现形式有所变化。

正是在这次会议前后，玛格丽特 • 汉密尔顿（*Margaret Hamilton*）——就是课件上那位站在阿波罗导航代码打印件旁边的女士——推动了*”software engineering”* 这一术语的确立。她当时负责 *MIT* 仪器实验室的阿波罗飞行软件开发，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。在那个年代，”编程”被许多人视为次要的辅助工作，汉密尔顿通过这个命名为整个学科争取了应有的地位和尊重。她为阿波罗计划编写的软件以最大程度防止崩溃为目标。*1969* 年阿波罗 *11* 号登月时，交会雷达开关被放在了错误的位置，导致计算机接收到大量虚假数据，处理器负载超出正常 *15%*。汉密尔顿设计的优先级调度系统自动剥离了低优先级

任务，确保导航计算能够继续进行。如果没有这一容错设计，登月极有可能失败。

第四阶段是 1985 年至今的**软件产业阶段**。标志是软件从” 项目” 走向” 产品”、从” 手工作坊” 走向” 工业化生产”。这一时期的主要进展包括面向对象编程的普及、开源运动的兴起、敏捷方法论的提出、DevOps 文化的形成，以及最近的 AI 辅助编程。对于 EDA 领域而言，这一阶段的关键趋势是 Tcl 脚本化——几乎所有商业 EDA 工具都以 Tcl 作为内嵌脚本语言，工程师通过编写 Tcl 脚本来驱动工具、定制流程、自动化重复操作。这种” 工具可编程化” 的设计理念，本身就体现了软件质量中” 可扩展性” 和” 泛化性” 的追求。

1.2.4　与软件质量相关的三个角色

在讨论具体的质量特征之前，我们需要先明确：**谁关心质量？不同的人关心的是同一种质量吗？**

答案是否定的。软件生态中至少存在三个核心角色，他们对” 质量” 的理解和优先级截然不同。

甲方是软件需求的提出者，关心的是软件的外部表现：功能是否正确？性能是否达标？是否易用？是否能满足市场需求？在 EDA 行业，甲方通常是芯片设计公司的管理层，他们关心的是这款综合工具能不能在给定的时间和成本内，生成满足性能指标的网表。

乙方是软件的开发者，关心的是软件的内部质量：代码是否可维护？架构是否可扩展？文档是否完整？测试是否充分？EDA 工具的乙方——即 Synopsys、Cadence、Siemens EDA 等公司的工程师——面对的挑战是：工具需要支持每一代新工艺，从 28nm 到 16nm 到 7nm 到 5nm 到 3nm 到 2nm，每次工艺更新都意味着新的物理效应、新的设计规则、新的约束条件。如果软件架构不具备足够的可扩展性，每次工艺迭代都需要大规模重写，研发成本将不可控。

用户是软件的使用者，关心的是使用体验：软件稳不稳定？响应快不快？学习曲线陡不陡？遇到问题有没有清晰的错误提示？EDA 工具的用户——芯片设计工程师——经常抱怨的问题包括：工具启动慢，因为要等待 license 检查；日志输出不可读，充满只有开发者才看得懂的内部代码；crash 后没有有用的错误信息；不同版本之间的结果不一致。

这三方的利益并不总是一致的。甲方想要尽快交付，可能接受”够用就行”的质量水平；乙方想要充分测试和重构，这需要更多时间；用户想要最好的体验和性能。软件工程师的核心挑战不是把某一个质量维度做到极致，而是在三方诉求之间找到可持续的平衡点。

1.2.5 软件质量的九个核心特征

有了角色的认知框架，我们可以系统地讨论软件质量的具体特征。经典的软件工程理论将软件质量分为九个维度，我们逐一展开，并结合 EDA 领域的实例加以说明。

功能正确性

功能正确性是指：根据输入数据，能够得到正确的运算结果。在 EDA 领域，功能正确性的标准极为严苛。以静态时序分析为例：如果 STA 工具报告一条路径的延迟为 5.2 纳秒，但实际硅片上的延迟是 6.1 纳秒，那么这条路径可能在时序上违约，导致芯片功能错误。更微妙的情况是：STA 工具的结果可能在数学上是" 正确的"——即算法实现没有 bug——但与物理现实不符，因为工具使用的延迟模型可能没有充分考虑某些物理效应。这种情况下，" 功能正确" 和" 物理准确" 是两个不同的标准。

1994 年的 Intel Pentium FDIV bug 是功能正确性缺失的经典案例。Pentium 处理器中的浮点除法单元使用了一种基于查找表的 SRT 除法算法，该查找表有 1066 个条目，但其中 5 个是错误的。这导致在极少数特定操作数组合下，在最坏情况下，除法结果的有效数字从第四位开始出错。

Intel 的 SRT 除法器使用一个包含 1066 个有效条目的查找表，实现在一个 2048 单元的 PLA 中，在每个除法周期中用部分余数和除数的高位查表确定商的下一位。由于生成查找表过程中的错误，5 个应当包含正确商值的 PLA 条目缺失，对应的输出默认为零。这个 bug 的发现者托马斯 • 尼斯利（Thomas Nicely）是林奇堡学院的数学教授，他在计算 Brun 常数时注意到了不一致的结果。Intel 最初试图淡化问题，时任 CEO 安迪 • 格罗夫（Andy Grove）声称该 bug" 平均 90 亿次除法才会出现一次"。但公众不买账——IBM 甚至发表技术报告声称该 bug 在某些科学计算场景下可能每 24 天就会遇到一次。最终 Intel 被迫无条件召回所有有缺陷的处理器，损失达 4.75 亿美元。此后，Intel 建立了世界上最大的形式验证团队，并将浮点运算单元的验证提升到了空前的严格程度。从 EDA 的角度来看，Pentium FDIV bug 的根因不在芯片设计，而在设计流程中的一个辅助脚本——这提醒我们，软件质量问题可能出现在流程的任何环节，包括那些被认为" 不重要" 的辅助工具。

性能精确性

性能精确性衡量的不是结果是否" 正确"，而是是否" 足够精确"。以 SPICE 仿真为例：一个电路仿真器可能在算法上完全正确，但由于使用了过于简化的晶体管模型，仿真结果与实际硅片行为的偏差可能超出可接受的范围。精度和速度之间存在根本性的权衡。高精度仿真可能需要数小时甚至数天来分析一个大规模电路；低精度但快速的仿真器可以在分钟级完成，但精度降低到 5–10% 的误差范围。工程师必须根据设计阶段选

择合适的精度级别：早期架构探索允许 10–20% 的误差以追求速度，详细设计阶段要求 1–5% 的误差，最终签核阶段要求误差在 1% 以内甚至需要与硅片测量数据对比验证。

易用性

易用性反映用户学会和使用软件的难易程度。EDA 工具在这方面历来受到批评。商业 EDA 工具的用户手册通常厚达数千页，命令行选项多达数百个，学习曲线极其陡峭。一个新入职的芯片设计工程师通常需要 6 到 12 个月才能熟练使用一套 EDA 工具链。这背后有客观原因：EDA 处理的问题本身极其复杂，工具的参数空间必然庞大。但也有主观原因：许多 EDA 工具的界面设计是" 工程师给工程师用的"，缺乏系统的用户体验设计，错误信息晦涩难懂，日志输出冗长且不分优先级，文档更新滞后于工具版本。近年来 EDA 行业开始重视这一问题，例如 Cadence 推出了自然语言驱动的设计助手 JedAI，Synopsys 也在探索将大语言模型集成到设计流程中。方向是明确的：**降低使用门槛，让工程师把精力花在设计本身而非与工具搏斗上。**

运行效率

运行效率在 EDA 中是一个生死攸关的指标。一个大规模数字设计的完整流程——从 RTL 综合到最终的 GDSII——可能需要数天到数周的计算时间。每一次迭代都需要重新跑流程。如果单次运行时间从 12 小时增加到 24 小时，意味着一天只能做一次迭代而非两次——在紧张的项目进度下，这可能导致错过流片窗口，直接造成数周甚至数月的延迟和数百万美元的损失。EDA 工具对效率的追求体现在多个层面：在算法层面使用更高效的求解策略，在数据结构层面使用紧凑的内存表示——一个 10 亿门级设计的网表可能占用数十 GB 内存——在并行化层面利用多核 CPU 和分布式计算。

安全性

EDA 工具的安全性有两个维度。第一个维度是工具本身的安全——防止未授权使用。商业 EDA 工具使用复杂的 license 管理系统，按功能、按用户数、按时间段收费。第二个维度更为重要：设计数据的安全。芯片设计涉及大量知识产权，包括电路架构、工艺参数、设计约束等。随着云端 EDA 的兴起，将芯片设计数据上传到第三方云服务器进行计算，在信息安全上面临巨大挑战。特别是在当前的国际形势下，芯片设计数据的跨境传输涉及出口管制和国家安全问题。

健壮性

健壮性是指软件在操作错误、无效输入等压力环境下仍能正常运行的能力。芯片设计工程师在使用工具时，经常需要尝试非标准的操作——例如导入一个不完全符合格式规范的网表、使用一组实验性的约束条件、或者在工具运行中途修改参数。一个健壮的

EDA 工具应当能够优雅地处理这些异常情况，给出清晰的错误提示，而不是直接 crash 并丢失所有中间结果。一个典型的反面案例是：某些 EDA 工具在读入包含特殊字符的 Verilog 文件时会崩溃，但错误信息只是一个含义不明的 segmentation fault，工程师可能需要花费数小时的排查才能定位原因。

泛化性

泛化性是指软件在离开原开发环境或原应用环境后，不经修改就能在其他环境下使用的能力。EDA 工具需要支持多种操作系统、多种硬件架构、多种工艺节点。一个为 28nm 工艺节点开发的布局算法，能否直接用于 5nm 节点？通常不能——因为先进工艺引入了新的设计规则，如多重图形化、自对准接触孔等，这些规则从根本上改变了布局问题的约束空间。

可扩展性

可扩展性是指能否以极小的代价扩展软件的功能。EDA 工具的可扩展性通常通过脚本接口、插件架构和开放 API 来实现。几乎所有商业 EDA 工具都支持 Tcl 脚本，允许用户自定义流程和添加新功能；某些工具支持用户编写 C/C++ 插件，直接调用工具内部的数据结构和算法。可扩展性的反面是" 大泥球" 架构——代码高度耦合，添加任何新功能都需要修改大量现有代码。EDA 行业的一些老牌工具由于经过数十年的增量开发，确实面临这种架构退化问题。

可维护性

可维护性反映的是修改一个软件的难易程度。这是长期运营的 EDA 工具面临的最大挑战之一。Synopsys 的 Design Compiler、Cadence 的 Virtuoso 等核心产品已经持续开发了数十年，代码库规模庞大，通常在千万行量级，经历了无数次的功能添加、bug 修复和架构重构。一个代码库的可维护性取决于代码的模块化程度、命名的清晰度、注释和文档的完整性、测试覆盖率以及开发团队的知识传承。当一个 EDA 工具的核心开发者离职时，如果他负责的模块缺乏文档和测试，接手的工程师可能需要数月才能理解代码逻辑。

这九个特征并非独立的——它们之间存在复杂的权衡关系。追求运行效率可能牺牲精确性和可维护性；追求可扩展性可能牺牲运行效率和易用性；追求安全性可能牺牲易用性；追求泛化性可能牺牲运行效率。**好的软件工程师不是把九个特征都做到满分——那是不可能的——而是根据具体项目的目标和约束，有意识地选择每个维度的优先级，并清楚地记录这些决策及其理由。**

1.3　人机关系：从分工到共生

1.3.1　人机交互研究的学科基础

软件不是孤立存在的——它存在于人与机器的交互之中。理解软件质量，必须理解使用软件的人。这把我们引向了一个跨学科的领域。

研究人机关系涉及至少三个学科。

人类工程学研究人和机器及环境的相互作用。它的起源可以追溯到第二次世界大战时期——当时发现许多飞机事故不是因为机械故障，而是因为座舱设计不符合飞行员的认知和操作习惯。一个经典案例是 B-17 轰炸机的起落架和襟翼控制杆长得几乎一模一样，放在相邻位置，导致飞行员在降落时经常误操作，收起起落架而非放下襟翼。解决方案不是" 训练飞行员更仔细"，而是把两个控制杆设计成不同的形状和触感——这就是人类工程学的核心理念：**适应人，而非要求人适应机器**。

计算机用户工程原理将人类工程学的理念应用于软件设计，研究软件系统设计对用户的影响及其成本，以及如何向用户提供友好、方便的接口。唐纳德 • 诺曼（Donald Norman）的 *The Design of Everyday Things* 是这一领域的经典著作——他提出的" 可供性"" 映射"" 反馈" 等概念，至今仍是用户界面设计的基本原则。

软件心理学则关注软件开发者自身——研究程序员的认知模式、错误倾向、协作行为。杰拉尔德 • 温伯格（Gerald Weinberg）在 1971 年出版的 *The Psychology of Computer Programming* 开创了这一领域。他的核心洞察之一是：**程序不仅是写给计算机执行的，更是写给其他程序员阅读的**。因此，代码的可读性不是" 锦上添花"，而是质量的基本要素。

1.3.2　人的行为模型与认知局限

人在与计算机交互时的信息处理过程可以建模为三个阶段：感知——通过感觉器官接收计算机输出的信息；认知——通过大脑对信息进行分析、判断和决策；行动——通过运动系统向计算机输入指令。计算机的输出构成人的输入，人的操作构成计算机的输入——这形成一个闭合的控制回路。好的人机界面设计的目标是让这个回路转得快且准确，差的设计则在每个环节制造摩擦和误解。

人的认知存在几个重要的局限，直接影响软件设计。

乔治 • 米勒（George Miller）在 1956 年的经典论文 *The Magical Number Seven, Plus or Minus Two* 中指出，**人的短时记忆容量约为 7 ± 2 个信息块**。这意味着如果一个软件界面同时呈现超过 7 个需要关注的信息元素，用户将开始遗漏或混淆。EDA 工具的界面设计需要特别注意这一点——一个布局编辑器的屏幕上可能同时显示数百万个标准单元，但用户在任何时刻真正关注的只有少数几个关键区域。好的工具应当帮助用户聚焦，而非用信息淹没用户。

注意力是一种稀缺资源，人不可能同时关注所有信息。心理学中的” 鸡尾酒会效应” 说明，人能在嘈杂的环境中选择性地关注一个声源，但代价是忽略其他声源中的信息。软件设计的启示是：关键信息必须足够醒目，次要信息应当适当隐藏。EDA 工具中，时序违约的路径应当用醒目的颜色标记，已满足约束的路径则可以淡化处理。

确认偏误是人倾向于关注与自己预期一致的信息、忽略不一致信息的认知倾向。这在芯片设计中是一个危险的倾向：当工程师” 确信” 某个模块不会有时序问题时，他可能不会仔细检查 STA 报告中该模块的详细结果——而恰好就是这种被忽略的角落，隐藏着最致命的 bug。好的 EDA 工具应当主动高亮异常结果，而非仅仅被动地等待用户查看。

1.3.3 人机分工的哲学

人和计算机各有所长。

表 1.2: 人与计算机的能力对比

人的优势	计算机的优势
适应能力和应急处理能力	计算速度快
学习能力和预见能力	记忆容量大且精确
高情商和创造能力	不受情绪影响
理解模糊和不完整的信息	精确执行明确定义的任务
跨领域类比和联想	持续工作不疲劳
伦理判断和责任担当	可并行处理海量数据

由这一对比可以导出任务分工的核心原则：人承担最小化但最关键的工作量——意外判断、应急处理、前景预判、需要合作和妥协的决策；机器承担最大化的工作量——有规律而单调的重复性工作、体力密集型计算、需要高精度的数值运算。用八个字概括：**人尽其才，机尽其用**。

但这一传统框架在大语言模型时代需要重新审视。GPT-4、Claude 等模型已经展现了在某些维度上接近甚至超越人类的能力——代码生成、文本理解、模式识别。传统认为” 机器不擅长” 的学习能力和创造力，正在被 AI 快速填补。那么，当机器在越来越多的维度上追平人类时，人类的不可替代性究竟在哪里？

第一是**定义问题**。AI 可以解决问题，但” 什么是值得解决的好问题” 仍然需要人来定义。一个 EDA 工具团队的产品经理需要判断：是投入资源优化布局算法的运行时间，还是提高布线的可制造性，还是开发全新的电热协同分析功能？这种战略性决策涉及市场判断、技术预见、资源约束，不是算法可以替代的。

第二是**承担责任**。当一颗芯片因为设计缺陷而导致产品召回时，签字负责的是人，

不是 AI。责任的本质是道德和法律上的主体性——AI 不具备法律人格，不能承担责任。这意味着在所有涉及安全性和可靠性的关键决策点上，必须有人类做最终判断。

第三是**理解系统性上下文**。为什么这个客户的需求和上一个客户不一样？背后的商业逻辑、供应链关系、行业趋势、政策环境——这些构成了决策的" 上下文"，而 AI 至少在目前还不能可靠地理解和整合如此多维度的信息。

1.3.4　软件质量与人机关系的统一

回到本讲的核心论题：软件质量和人机关系看似是两个独立的话题，但它们在更深的层面上是统一的。

软件质量的终极评判者是人。无论我们定义了多少技术指标——代码覆盖率、圈复杂度、响应时间、内存占用——最终决定一款软件是" 好" 还是" 坏" 的，是使用它的人。一款在所有技术指标上都达标的 EDA 工具，如果用户觉得" 用起来太痛苦了"，那它就不是一款好软件。

人机关系的优化目标是提升人的效能。计算机不是目的，它是手段。EDA 工具存在的意义不是" 自动化" 本身，而是让芯片设计工程师能够在有限的时间和资源内，设计出更好的芯片。如果一个" 自动化" 工具让工程师花了更多时间去调参和排错，那么这种自动化就是失败的。

统一这两个主题的，是一个贯穿全课程的底层哲学：DRY——Don't Repeat Yourself。安迪 • 亨特（Andy Hunt）和戴夫 • 托马斯（Dave Thomas）在 *The Pragmatic Programmer* 中正式提出这一原则：" 每一条知识在系统中都应当有且仅有一个明确的、权威的表示。"

这个原则的精神远不止于代码层面。在代码层面，不要复制粘贴，封装成函数。在设计层面，不要让同一个约束在两个地方定义，使用单一数据源。在流程层面，不要让同一个验证步骤被手工执行两次，自动化它。在团队层面，不要让两个人各自维护同一个模块的不同版本，使用版本控制。在 *EDA* 层面，不要让时序约束在 *SDC* 文件和 *Tcl* 脚本里各写一份，一处定义、处处引用。***DRY* 不仅是编程原则，它是管理复杂性的核心策略**。本课程后续每一讲教授的方法，本质上都是 *DRY* 原则在不同层面的具体应用——寻找不重复劳动的系统化解决方案。

1.4　本讲总结

本讲从三个维度建立了课程的基本框架。

EDA 是什么？它是方法学、算法和工具的集合，使得现代芯片设计成为可能。从手工版图到 99% 自动化，EDA 的演进史本质上是人类管理复杂性能力的演进史。没有 EDA，摩尔定律很可能在 1980 年代就会终结。

何为好的软件？软件具有与物理产品本质不同的特征——零边际成本复制、不可见

性、高可变性、本质复杂性不可约简。这些特征决定了软件质量不能简单套用制造业的标准。软件质量包含九个核心维度，它们之间存在复杂的权衡关系，好的工程决策在于找到适合具体场景的平衡点。

人与软件应当建立怎样的关系？人机分工的核心原则是" 人尽其才，机尽其用"——人做决策、机做执行。在 AI 时代，这个分界线正在移动，但人类在问题定义、责任担当和系统性判断上的不可替代性并未消失。软件质量的终极评判者是人，好的软件设计应当以提升人的效能为目标。

贯穿以上三个主题的底层哲学是 DRY——不断寻求不重复劳动的解决方案。面对复杂性，我们的第一反应不应是" 投入更多人力"，而是" 寻找系统化的解决方案"。

课后思考

1. 如果你要从零开始设计一款 EDA 布局工具，你会把九个软件质量特征按什么优先级排列？为什么？不同的应用场景——学术研究、商业产品、开源社区——会改变你的排序吗？

2. 回顾 Ariane 5 和 Pentium FDIV 两个案例，它们在" 软件质量九个特征" 中分别暴露了哪些维度的缺陷？如果你是当时的项目负责人，你会如何预防？

3. 在当前大语言模型快速发展的背景下，你认为人机分工的边界在未来 5 年会如何移动？EDA 工程师需要培养什么新的能力来保持不可替代性？

下讲预告：第二讲" 软件设计平台" 将讨论支撑软件开发的基础设施——版本控制系统、构建系统、持续集成——这些工具为什么长成今天这个样子，以及它们的设计如何体现本讲讨论的质量原则和 DRY 哲学。

参考文献

[1] Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[2] Hamilton, M., "What the Errors Tell Us," *IEEE Software*, vol. 35, no. 5, pp. 32–37, 2018.

[3] Kahng, A. B., Lienig, J., Markov, I. L., and Hu, J., *VLSI Physical Design: From Graph Partitioning to Timing Closure*, 2nd ed., Springer, 2022.

[4] Lavagno, L., Markov, I. L., Martin, G., and Scheffer, L. K., eds., *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*, 2nd ed., CRC Press, 2016.

[5] Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[6] Miller, G. A., "The Magical Number Seven, Plus or Minus Two," *Psychological Review*, vol. 63, no. 2, pp. 81–97, 1956.

[7] Mirhoseini, A., et al., "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[8] Moore, G. E., "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[9] Norman, D. A., *The Design of Everyday Things*, Revised and Expanded Edition, Basic Books, 2013.

[10] Weinberg, G. M., *The Psychology of Computer Programming*, Silver Anniversary Edition, Dorset House, 1998.

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

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

[15] Nagel, L. W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Ph.D. dissertation, University of California, Berkeley, 1975.