南京大学
NANJING UNIVERSITY

软件开发的方法和过程

张 天
SEG - Software Engineering Group
ztluck@nju.edu.cn
2025年 秋季

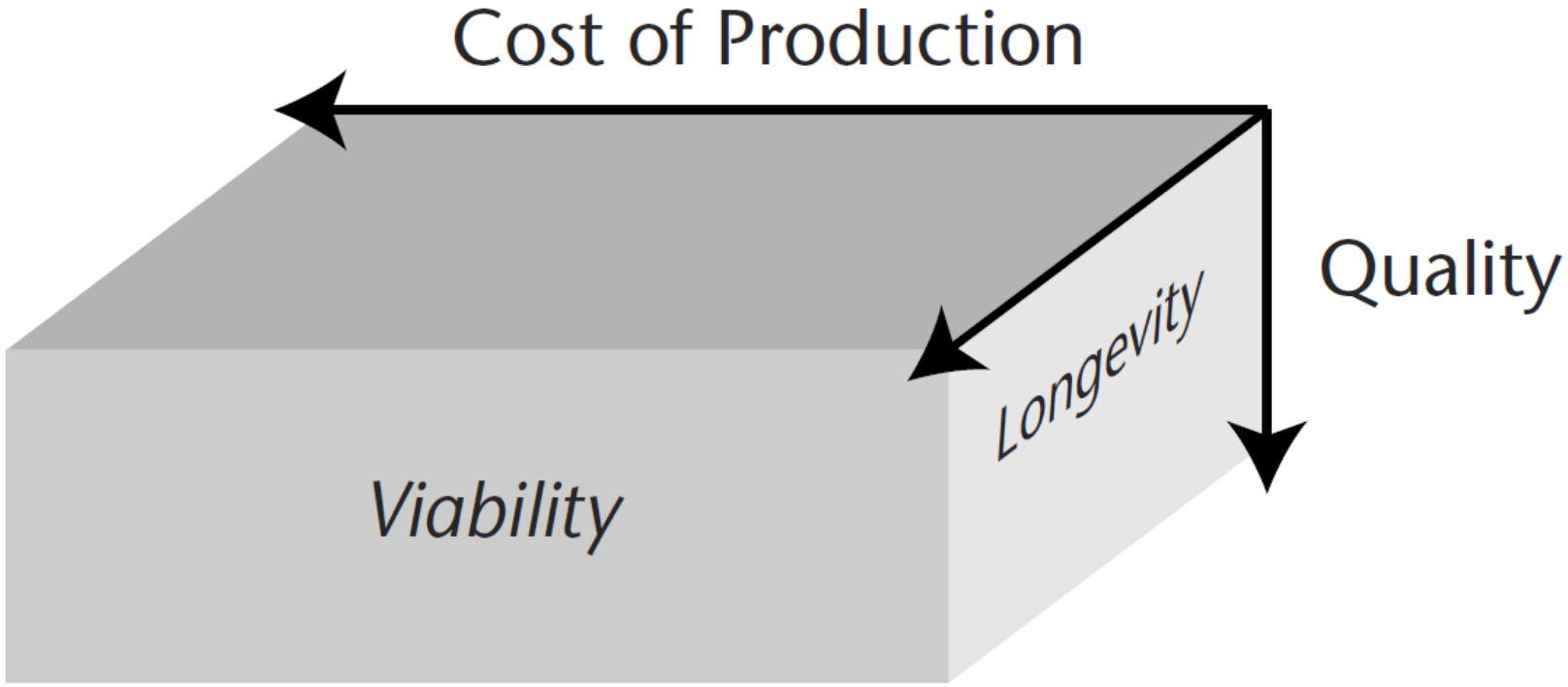
关键因素

- 压力与进步：我们如何来到这里？

如何高效、低成本地开发优质的软件产品一直是计算机软件领域重点关注和研究的问题[1]。

[1] Shari Lawrence Pfleeger. Software Engineering: Theory and Practice. 2nd Edition. Upper Saddle River, New Jersey: Prentice Hall, 2001.

软件开发的可行性变量



- 经济可行性取决于：
  - 我们能够在何种程度上开发出质量和生命周期与开发成本相匹配的软件系统。

软件产业的发展

- **提高整体的可行性**
  - 软件开发整体的可行性是对生产成本、质量和生命周期进行权衡和平衡的结果。
- **软件产业的发展**
  - 当现有开发方法不能适应软件产业不断增长的需求时，新的开发方法被提出，并推动质量、生命周期和生产成本形成新的平衡。
- **软件开发方法的发展**
  - 各种不同的新开发方法从不同的角度入手，以不同的方式来实现同一个目标

提高抽象层次

- **与软件技术发展密切相关的三个要素**
  - 计算机平台、人的思维模式和问题的基本特征[1]。
- **提高解决问题的抽象层次**
  - 在软件开发的长期摸索过程中，人们逐渐认识到“提高解决问题的抽象层次”是有效利用抽象手段解决软件开发问题的一个非常具体而实用的途径。
  - 提高抽象层次即是想着符合人的思维模式的方向提高

[1] 杨芙清, 梅宏, 吕建, 金芝. 浅论软件技术发展.《电子学报》, 2002, 30(12A): 1901-1906.

两个较为显著的进展

- Stephen J. Mellor在21世纪初指出[1]，过去的50多年里，人们利用“提高解决问题的抽象层次”处理软件开发的问题已经取得了两个较为显著的进展：
  1. 开发出了具有较高抽象层次的程序设计语言；
  2. 能够在更高抽象层次上实现软件复用。

[1] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise. MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley, 2004.

简单回顾软件开发的方法和过程

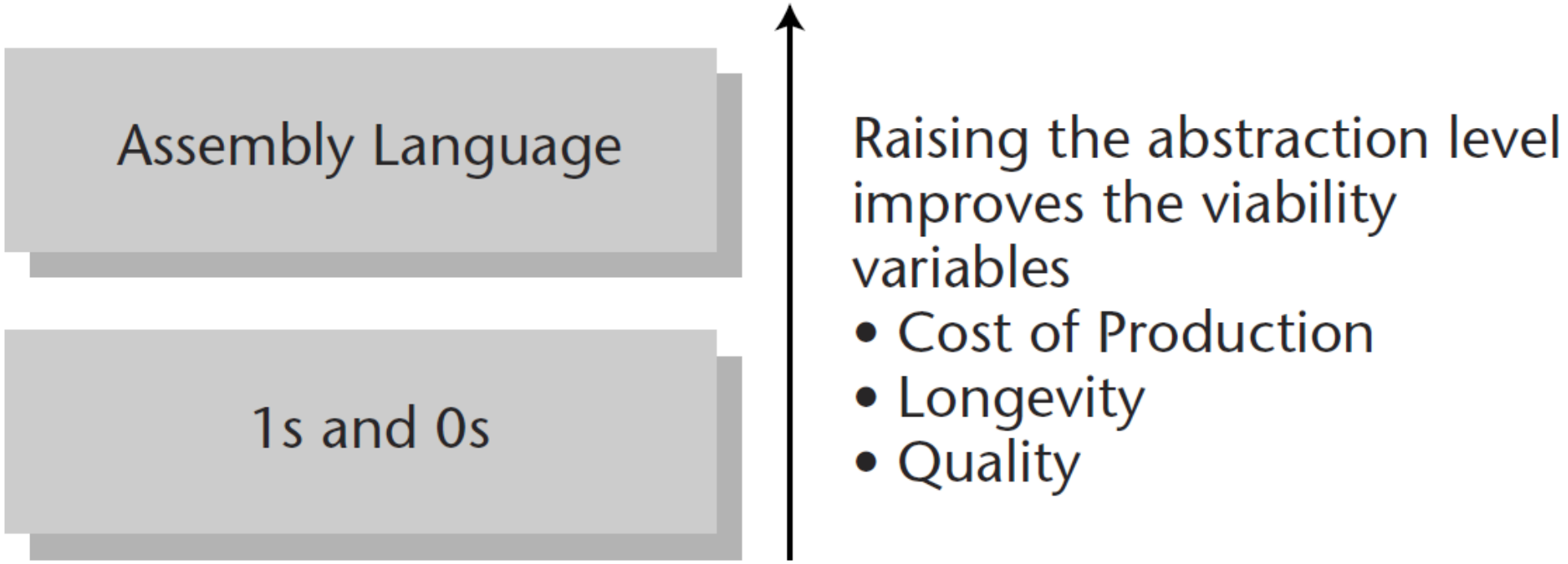
- 从提高抽象层次的角度简单回顾软件开发的方法和过程：
  - 以机器为中心的计算
  - 以应用为中心的计算
  - 以企业为中心的计算
    - 新的压力……

以机器为中心的计算

- **最早期的程序员以0和1编写机器指令进行开发**
  - 完全面向机器进行开发的方式
- **汇编语言通过助记符提高效率**
  - 程序员仍需要针对物理地址、寄存器和CPU编写代码
  - 汇编语言的出现并没有改变以机器为中心开发的方式
  - 汇编语言延长了以机器为中心的计算方法的寿命

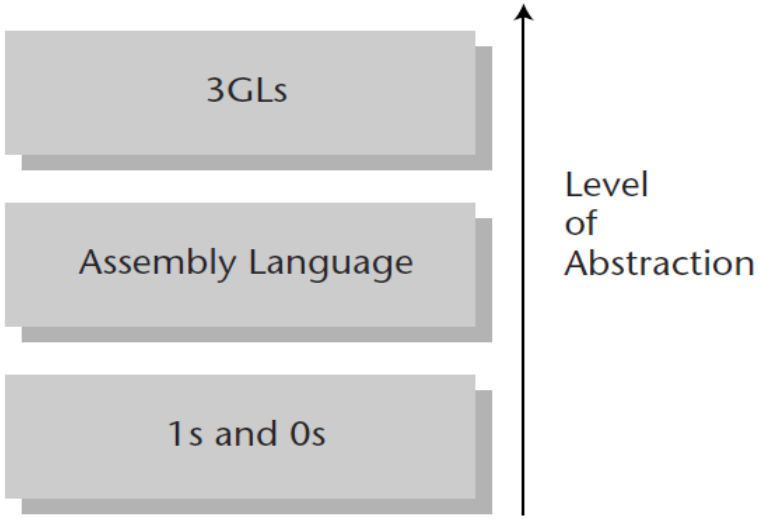
从机器指令到汇编语言

- 从基于0和1指令的开发到基于汇编语言的开发，实际上是提高了开发的抽象层次



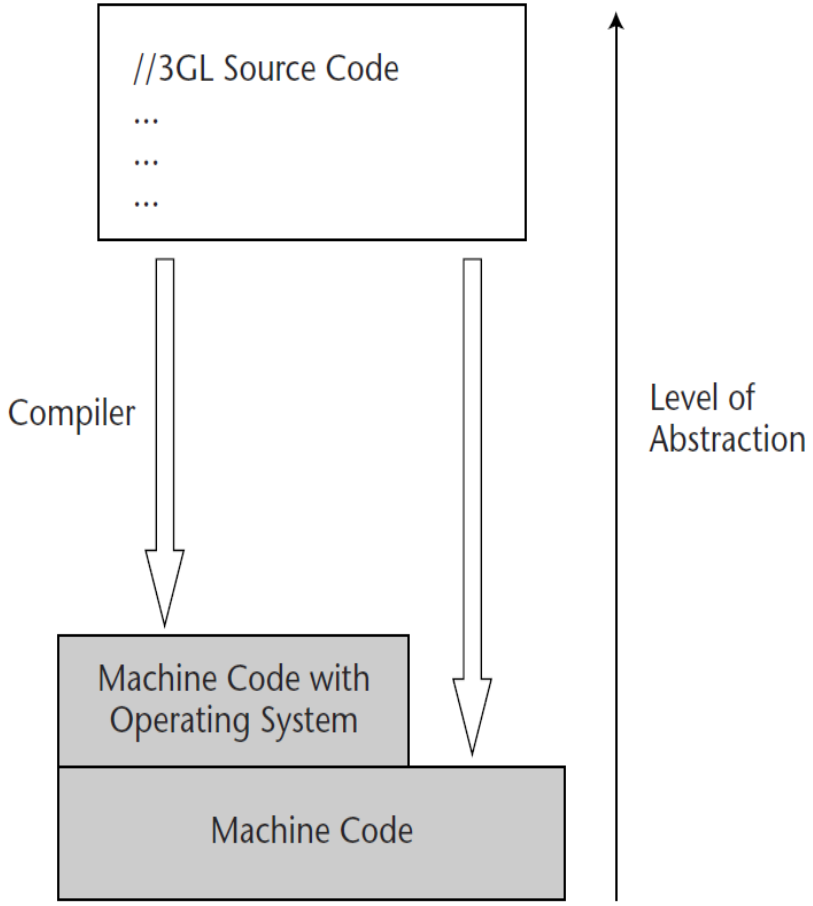
以应用为中心的计算(1)

- 从汇编到3GL
  - 第三代编程语言（3GL）的出现大大提高了生产力
    - 最早的3GL，如FORTRAN和CORBOL
    - 结构化的3GL，如C和PASCAL
    - 面向对象的3GL，如Smalltalk和C++



以应用为中心的计算(2)

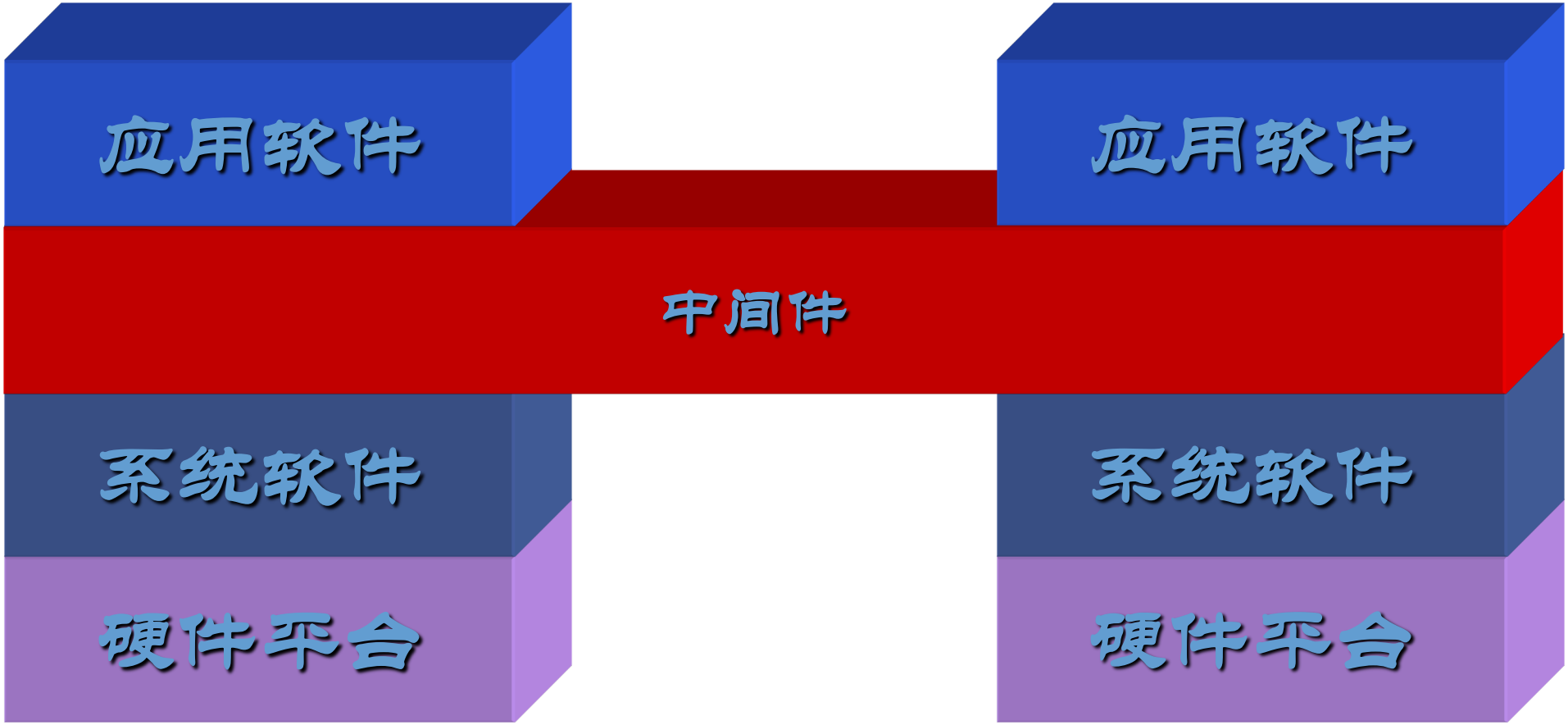
- 操作系统
  - 3GL提高了编程语言的抽象层次，而操作系统则提升了计算平台的抽象层次
- 虚拟机
  - 虚拟机执行语言编译器产生的中间代码
  - 中间代码可以独立于操作系统



以企业为中心的计算

- **基于组件开发**
  - 基于组件开发吸取了工业生产过程的经验，致力于创建一个用于可交换组件组装应用程序的世界。
- **分布式计算**
  - 随着网络的产生，计算从单一CPU、机器发展到了分布在网络节点各处的新形态。
- **中间件**
  - 分布式计算的发展使通过底层网络协议在不同处理器之间传递消息成为新的需求，即屏蔽底层的异构性
  - 中间件便是在这样的需求下产生出来的

中间件



中间件：提升抽象层次

- **提升平台的抽象层次**
  - 中间件提供了一个位于传统操作系统层之上的计算抽象层
  - 逐渐形成了三大中间件平台：CORBA、J2EE和.NET
- **提升编程抽象层次**
  - 一些中间件还提供了独立于操作系统和编程语言的服务
  - 事实上每一种中间件平台都提供了大量的服务，用于延伸自身的能力，并最大限度的锁定用户
- **中间件平台很好的实现了对底层异构环境的屏蔽**
  - 中间件自身又带来了新的挑战

中间件分化带来了异构性

- **中间件分化（proliferation）现象**
  - 上个世纪90年代是软件中间件高速发展的一个时期，支持不同技术标准的中间件产品并存，没有“最终赢家”
  - 企业将为不同中间件应用系统的集成付出昂贵代价
  - 企业商业逻辑与特定中间件平台实现技术的“绑定”，提升了信息系统的平台移植难度，加大了企业业务发展受制于某种平台技术发展的风险。
- **新的挑战**
  - 如何解决企业信息系统建设的互操作性、可移植性、可重用性等问题，成为软件开发领域的重要课题。

OMG的标准

- 从1997年起，OMG以UML为核心陆续颁布了几个重要的技术无关建模标准
  - 统一建模语言UML
  - 元对象设施MOF
  - XML元数据交换XMI
  - 公共仓库元模型CWM
- 2001年，OMG正式推出了框架规范MDA
  - MDA整合了OMG在建模领域发布的一些列标准，成为首个模型驱动软件开发的框架性标准

Who Are OMG?

AT&T	Glaxo SmithKline	Microsoft	Rational
BEA	Hewlett Packard	MITRE	SAGA Software
Borland	Hitachi	MSC.Software	SAP
Boeing	Hyperion	NASA	SAS Institute
CA	IBM	NEC	Secant
Citigroup	IONA	NetGenics	Siemens
Compaq	io Software	NTT	Sprint
Ericsson	Kabira	OASIS	Sun
Ford	Kennedy Carter	Oracle	Unisys
Fujitsu	John Deere	Pfizer	Vertel

OMG (Object Management Group)

- OMG是世界上最大的计算机工业联盟，于1989年4月由8个公司发起，目前有800多家成员。
- OMG属于非盈利性标准化组织
- OMG推出的主要标准：
  - CORBA
  - UML
  - MOF、XMI、CWM
  - MDA

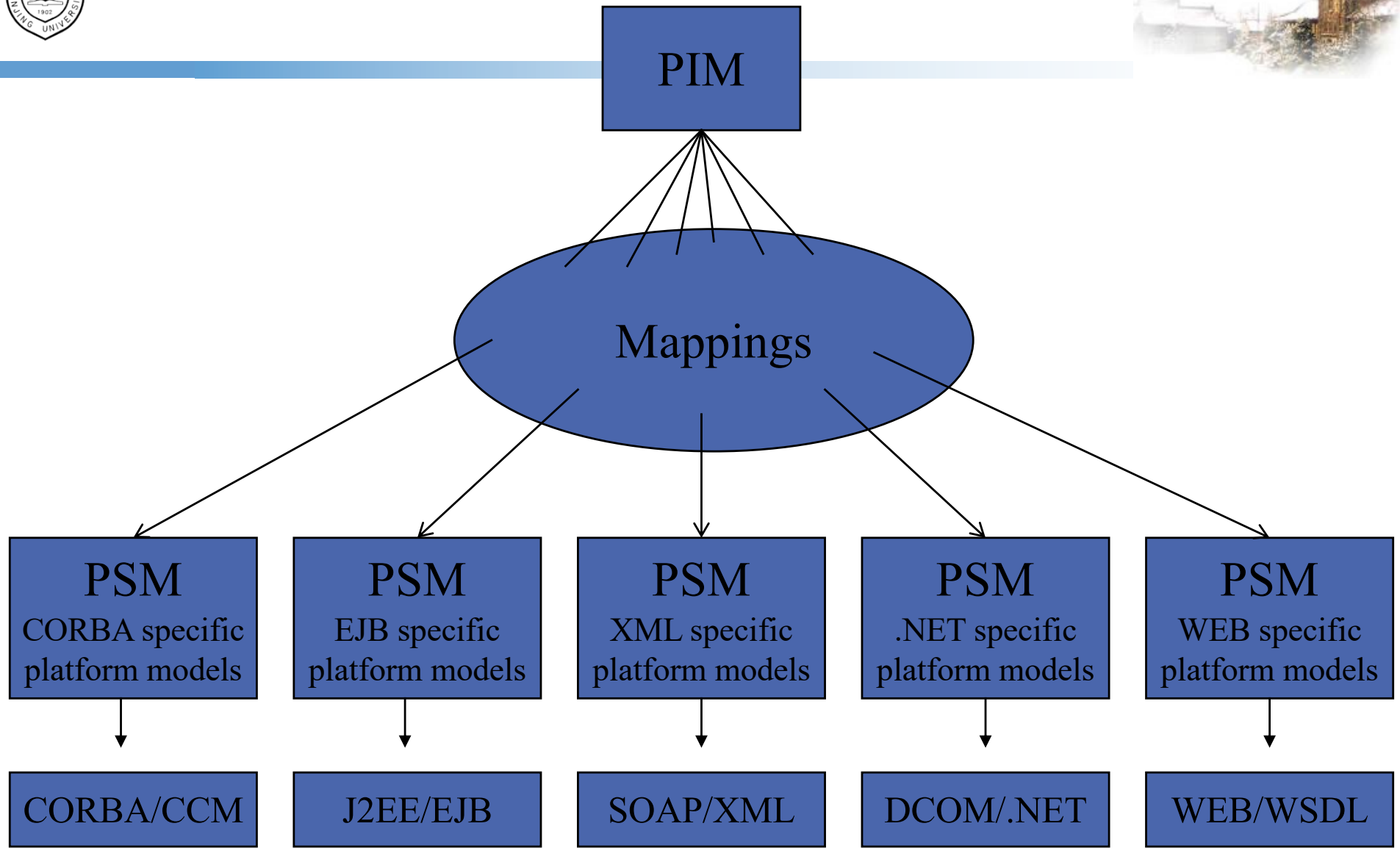
MDA的主要思想

- MDA的主要思想是分离业务功能分析与设计和实现技术与平台之间紧耦合的关系，从而将技术与平台变化对系统的影响降低到最小程度。

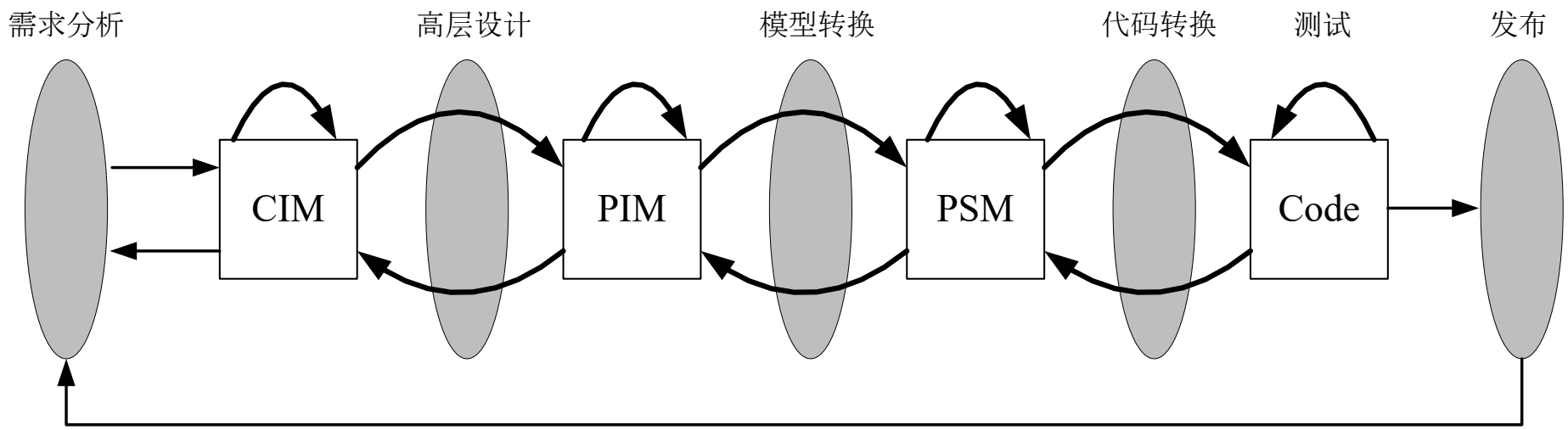
- MDA极大地加强了应用模型与领域模型在整个软件生命周期中的复用。

MDA对抽象层次的划分

- 与实现技术和平台无关、描述系统设计层次的模型（Platform-Independent Model, PIM）
- 与具体实现技术和平台相关的应用模型（Platform-Specific Model, PSM）
- MDA将PIM抽象出来，针对不同实现技术与平台制订多个映射规则，然后通过这些映射规则及辅助工具将PIM转换成PSM，再将PSM不断求精直至形成最后代码。


PIM
Mappings
PSM
CORBA specific platform models
PSM
EJB specific platform models
PSM
XML specific platform models
PSM
.NET specific platform models
PSM
WEB specific platform models
CORBA/CCM
J2EE/EJB
SOAP/XML
DCOM/.NET
WEB/WSDL

MDA的软件开发生命周期

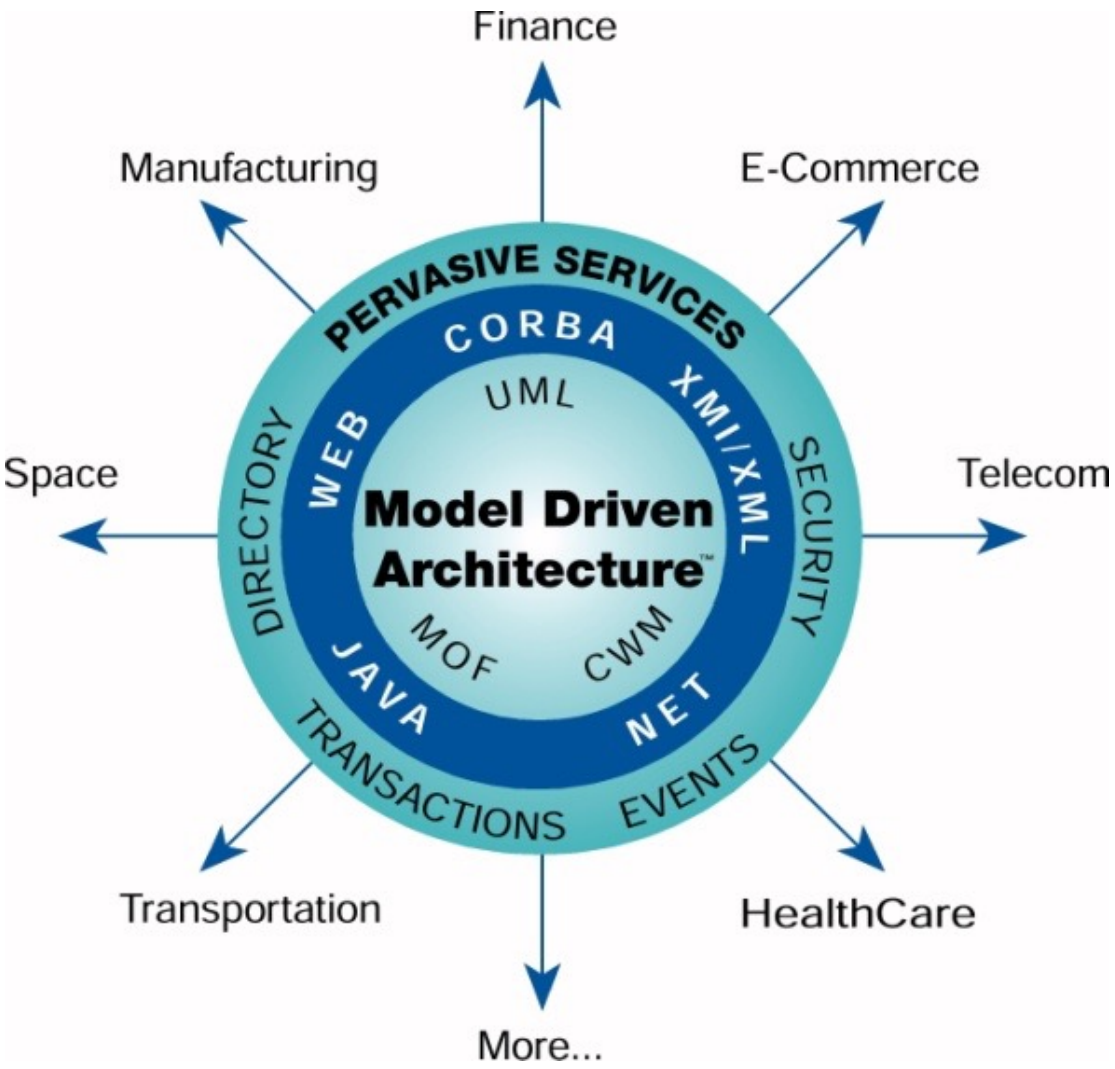


- MDA软件开发生命周期也可以分为需求分析、设计、实现、测试和发布几个阶段，每个阶段都是一个迭代的过程
  - 需求分析阶段的输出是CIM；
  - 高层设计阶段通过模型转换从CIM生成PIM；
  - 低层设计阶段通过模型转换从PIM生成PSM；
  - 代码实现阶段通过模型转换从PSM生成基于特定平台的系统实现代码；
  - 迭代测试结束后发布系统。

MDA带来的好处

- 增强软件复用性
- 增强软件可移植性
- 提高软件开发效率、降低成本
- 降低软件维护成本
- 推动软件自动化进程

MDA的构成


Finance
Manufacturing
E-Commerce
PERVASIVE SERVICES
CORBA
UML
XMI/XML
WEB
DIRECTORY
SECURITY
Space
Model Driven Architecture
Telecom
MOF
CWM
JAVA
.NET
TRANSACTIONS
EVENTS
Transportation
HealthCare
More...

MDA的核心

- 以下标准或规范构成了MDA的核心：
  - 统一建模语言（Uniform Modeling Language, UML），建模工具
  - 元对象设施（Meta Object Facility, MOF），标准的建模与交换结构
  - 公共仓库元模型（Common Warehouse Metamodel , CWM），数据仓库的标准
  - 基于XML的元数据交换（XML Metadata Interchange , XMI），信息交换的标准格式等。
  - 此外，MDA还将标准化少数通用领域的PIM、基于特定于中间件标准的PSM、以及PIM与PSM之间的映射规则，为设计到代码的自动生成提供基础。

模型驱动工程（MDE）

- **模型驱动工程(Model Driven Engineering, MDE)**
  - 2005年，模型驱动软件开发领域最重要的年会之一UML series (International Conference on the Unified Modeling Language)正式更名为MoDELS/UML series (International Conference on Model Driven Engineering Languages and Systems)。
  - 2007年，模型驱动工程这个术语正式出现在第29届ICSE年会的FoSE (the Future of Software Engineering)分会上，可以看做是MDE成为成为模型驱动软件开发领域代名词的标志。

MDE的核心思想

- 模型驱动工程以模型为首要软件制品（software artifact），通过建模为问题域构造软件系统的业务模型（business model），然后依靠模型转换（model transformation）驱动软件开发，（半）自动地产生最终完备的应用程序[1]。
- 有学者认为模型驱动的思想，或者说以模型为中心的软件开发思想正是抽象层次提高到一定程度的一个自然而然的结果[2]。

---

[1] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer. IEEE CS, 2006, 39: 25-31.
[2] David S. Frankel. Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, 2003

总结：两个趋势

- 回顾软工发展的历史，有两个明显的趋势：
  - 关注点从小规模编程向大规模编程转变
  - **语言和基础设施**的不断演进
  - *注：这里的语言从机器语言一直到建模语言，但其中最重要的就是程序设计语言*

从语言的演化看开发方法

- 从汇编语言到高级语言
  - 在相当长的一段时间内，很多复杂系统仍由汇编语言编写（为什么）
- 高级语言的演进速度和丰富程度极快
  - 从FORTRAN开始，高级程序设计语言的发展基本经历了六个过程
  - 主要的范型：过程式、函数式、逻辑语言、面向对象语言
- 在这一系列的语言换代中，每种语言所支持的抽象机制发生了变化

语言设计上的两种基本思路

- 每一种语言都带有设计者明确的目标和定位，并主导了该语言的特点（起码是最初时）
  - 第一种：偏向于机器运行性能的考虑（**运行快**）
  - 第二种：偏向人的思维方式与习惯（**开发快**）

尽管从数学的角度来看，绝大多数高级语言都是“图灵完备”的，但不同的设计理解会使它们长成完全不同的样子！

两个“一定”

- 有研究表明（90年代）：
  - 开发程序时使用的编程语言和生产力没有关系，无论用什么编程语言，一定时间内所开发的代码行数是相对固定的[Matz09]
  - 无论用什么样的编程语言，同样的代码行数，其中所隐藏的Bug量是相对固定的[UnixArt05]

最主要的发展脉络

- 从计算机程序设计语言的演化来看，其背后最主要或最本质的发展脉络：

抽象层次由低到高，从面向机器到面向开发人员！

汇编语言

- 汇编语言（assembly language）
  - 特定于机器的底层语言，可以和CPU指令直接对应，优势是使用助记符封装了指令
- 与最初的高程相比
  - 没有语言级的结构化控制
  - 没有类型系统
  - 对函数的支持不够
  - **但程序的运行速度可以发挥机器的极致！**

对应指令而言，已经是巨大的进步，可以进行较复杂的程序开发了（事实上可以非常复杂）！

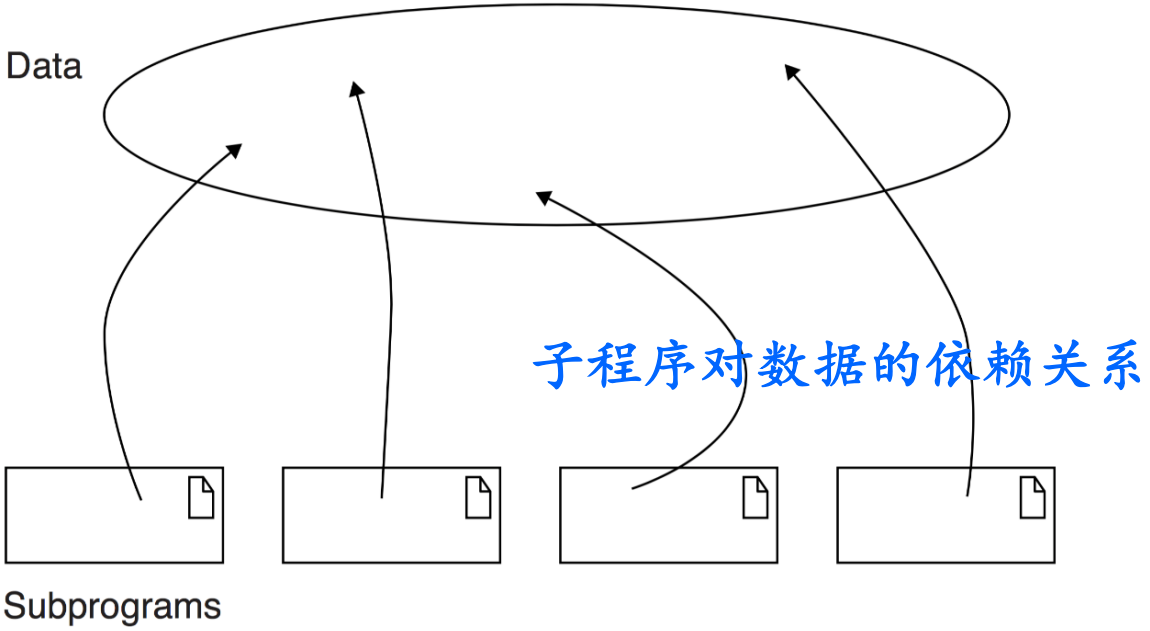
第一代语言

- First-generation languages (1954–1958)
  - FORTRAN I　　数学表达式
  - ALGOL 58　　数学表达式
  - Flowmatic　　数学表达式
  - IPL V　　数学表达式
- 第一代语言主要应用于科学和工程应用，这个问题领域的词汇几乎全是数学
- 典型代表FORTRAN I，让程序员可以写成数学公式，从而不用面对汇编语言或机器语言中的一些复杂问题

注意：这个时期汇编语言仍在广泛使用！

早期语言的拓扑

- 所谓语言的“拓扑结构”，指语言的基本构成单元（constructs），以及这些单元是如何连接的
- 早期这些语言的拓扑表现出相对较平的结构：只包含全局数据和子程序



注意：这个时候，子程序的价值还没有被充分发掘！

第一代和第二代早期程序设计语言的结构

主要问题

- 从拓扑上，没有机制保证不同类型数据的区分
  - 虽然设计者可以在逻辑上将不同类型的数据分开，但语言层面并没有机制来强制保证
  - 某个局部子程序的错误可能给其他部分带来毁灭性的影响，因为全局数据结构对于所有子程序都是可见的
- 当编写大型系统时，子程序之间的大量交叉耦合非常难以维护

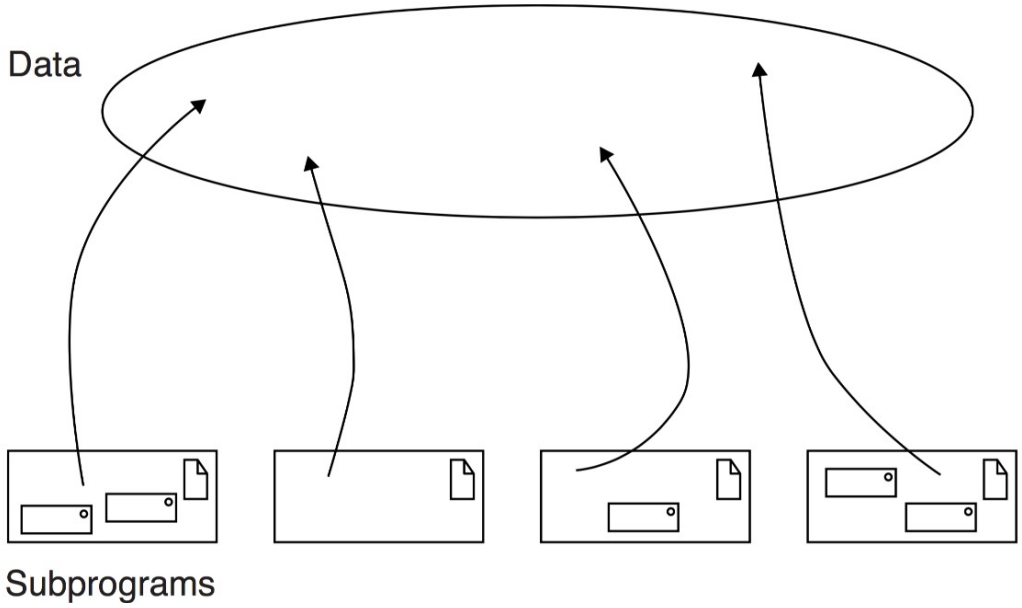
第二代语言

- Second-generation languages (1959–1961)
  - FORTRAN II　　子程序、单独编译
  - ALGOL 60　　块结构、数据类型
  - COBOL　　数据描述、文件处理
  - **Lisp**　　列表处理、指针、垃圾收集（注：J. McCarthy设计并实现了第一个以lambda演算为模型的语言）

- 第二代语言中，重点是**算法抽象**
- 关注的焦点主要是**告诉计算机做什么**：先读入一些记录，然后进行排序，最后打印报表。
- 向着问题空间有靠近一步，离底层计算机又远了一步

二代和三代早期的特点

- 子程序作为抽象程序功能的作用被重视
  - 人们开始发明语言，支持参数传递机制
  - 奠定了结构化程序设计的基础，语言上支持嵌套的子程序，在声明的可见性范围方面开始探索



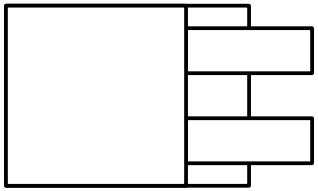
第一代和第二代早期程序设计语言的结构

第三代语言

- Third-generation languages (1962–1970)
  - PL/1　　FORTRAN + ALGOL + COBOL
  - ALGOL 68　　ALGOL 60的严格继承者
  - Pascal　　ALGOL 60的简单继承者
  - Simula　　类、数据抽象（但没有继承）
- **大事记：68年Dijkstra论“goto有害”的提出促使了结构化编程的出现，进而产生“结构化分析与设计”，进而形成“传统软降工程”**
- 演进到了支持数据抽象
- 这时，程序员可以描述相关数据的意义（它们的类型），并让程序设计语言强制确保这些设计决策

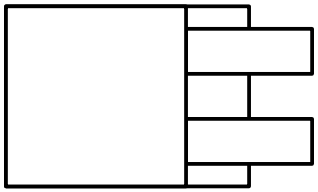
强类型的出现及ADT

- 在这一代语言中，模块化的思想已经相对普遍，但很少有规则要求模块间接口的语义一致性



模块1

参数列表：浮点数，10个元素的数组，布尔



模块2

参数列表：整数，5个元素的数组，负数

两个模块有各自不同的假定，但语言无法检查出来，直到运行时才发现这个错误！（后期出现了强类型语言Pascal）

里程碑：Pascal

- 高级语言发展过程中，Pascal是一个重要的里程碑
  - 71年苏黎世工学院Niklaus Wirth教授发表了Pascal语言
  - 第一个系统地体现了E.W.Dijkstra和C.A.R.Hoare定义的结构化程序设计概念的语言
- 著名公式：“算法+数据结构=程序”
- Wirth的学生Philipe Kahn毕业后和Anders Hejlsberg创办了Borland公司靠Turbo Pascal起家
  - Anders又进一步创建了Objective Pascal （Delphi）

断代

- The generation gap (1970–1980)

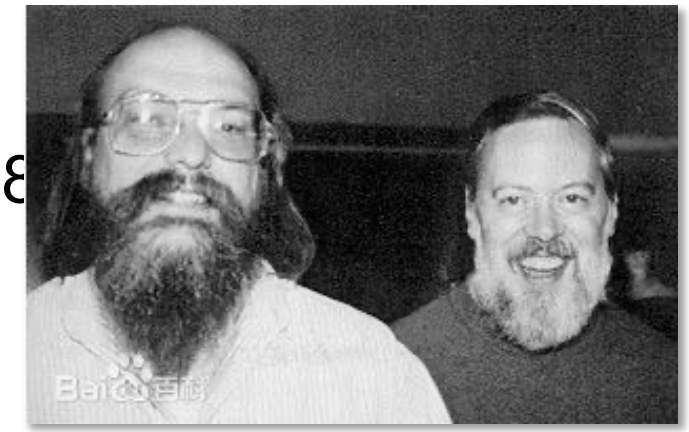
  人们发明了许多不同语言，但很少存活下来。但是，下面的语言值得一提：

  - C　　Efficient; small executables
  - FORTRAN 77　　ANSI standardization

- 以C语言为代表，从语言机制上并没有创新，但面向实际应用需求，更具有实用性
  - 值得一提的是，C语言是和UNIX系统共同发展的

C及其标准化

- C的祖先语言
  - CPL、BCPL、B和ALGOL68语言
  - BCPL和B都是无类型语言（意味着所有数据都被认为是机器字）
  - 引入类型的B语言：NB（New B）随后命名为C语言
- C的标准化
  - 89年ANSI C（ISO 1989），即C8
  - 99年更新为C99

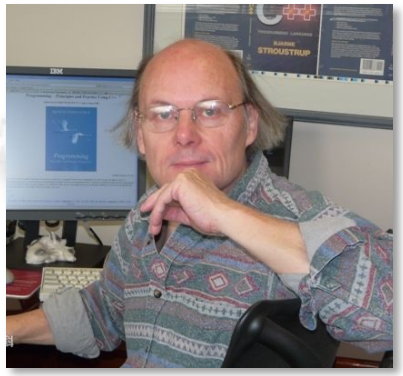


Ken Thompson & Dennis Ritchie

面向对象兴盛

- Object-orientation boom (1980–1990, but few languages survive)
  - Smalltalk 80　纯面向对象语言
  - C++　从C和Simula发展而来
  - Ada83　强类型，受到Pascal的很大影响
  - Eiffel　从Ada和Simula发展而来

混合风格C++



- 最初阶段“C with Class”
  - 80-83年贝尔实验室Bjarne Stroustrup提出
  - 在C语言的基础上加进去的特征主要有：类及派生类、共有和私有成员的区分、类的构造函数和析构函数、友元、内联函数、赋值运算符的重载等
- 84年命名为C++
  - 扩展了虚函数（动态绑定）、方法名和运算符重载，以及引用类型
- 85年第一个可用实现Cfront系统
  - 将C++程序翻译为C语言程序

C++语言是个颇具争议的语言，因为支持太多的范型而异常复杂！

后C++时代

- 严格的说，并不是C++之后的时代，而是C++退出“一统江湖”的时代
- C以及C++退出垄断的根本原因[UnixArt05]：
  - **机器性能的提升是编程语言的选择开始转向“人”的因素（好用）！**
  - 使用脚本语言的性能损失对真实世界的程序来讲经常微不足道，因为真实世界的程序往往受I/O事件等待、网络延迟以及缓存列填充等限制，而非CPU的自身效率。

关于术语“脚本语言”

- [UNIX05]14.2 为什么不是C
  - 其中最重要的是讲解了C以及C++退出垄断的根本原因：机器性能的提升是编程语言的选择开始转向“人”的因素（好用）！
    - *注意这个说法的时间是2005年*
  - 使用脚本语言的性能损失对真实世界的程序来讲经常微不足道，因为真实世界的程序往往受I/O事件等待、网络延迟以及缓存列填充等限制，而非CPU的自身效率

框架的出现

- Emergence of frameworks (1990–2000)

  出现了许多语言活动、新版本和标准化工作，导致了程序设计框架的出现.

  - Visual Basic　简化了Windows应用的图形用户界面(GUI)开发
  - Java　Oak的后续版本，其设计意图是实现可移植性
  - Python　Object-oriented scripting language
  - J2EE　基于Java的企业计算框架
  - .NET　Microsoft's object-based framework
  - Visual C#　.NET架构下Java的竞争者
  - Visual Basic　针对微软.NET框架的Visual Basic

- 中间件（middle-ware）的概念逐渐登上历史舞台，并成为一个重要的竞争领域

新的发展趋势

- 动态脚本类语言随着Web应用的发展受到越来越多的关注
  - Javascript、Python
  - 典型的有松本行弘的Ruby语言，随着Rails的推广而受到关注（Martin Fowler给予非常高的评价）

- 函数式编程随着并行技术的发展得到更多关注
  - Scala语言、Google Go语言
  - Java 8的lambda表达式

函数式的新趋势

- 更多的面向对象语言中加入“函数式编程”的支持
  - Java 8加入了lambda表达式
  - JVM语言Scala中的函数
  - Ruby语言中的块

附录：编程与面向对象

- 主要围绕面向对象的发展、机制和自身的问题展开介绍和讨论，涉及如下：
  - 多态性
  - 数据抽象与继承（多继承的讨论）
  - 静态语言和动态语言
  - 鸭子类型
  - 元编程

面向对象的“窘境”

- 说的面向对象，最让人感到痛苦的莫过于各种不同的解释和实现
  - 从设计到实现，存在众多关于面向对象的具体解释，至今仍没有所谓的“官方解释”和“统一”的定义
  - 尤其是各种OOP的语言，都有自己对面向对象的理解和支持方式，造成语言上的“巴别塔”现象

深刻理解“对象模型”的本质，是根本，也是难点！

单看面向对象的历史

- 自20世纪60年代末到现在，面向对象已经经历了近50年的发展
  - 从类型支持的ADT，进一步形成封装、继承和多态
  - 面向对象的概念，面向对象模型及其在各种语言中的具体实现也经历了逐步发展的过程

- 沿着面向对象的发展一步步去了解面向对象，更容易让我们深刻理解其中的奥妙

Simula的发明

- Simula被普遍认为是OOP的起源
  - 面向对象编程（OOP）的思想起源于60年代后期发展起来的Simula语言
  - 在1962到1964年间，挪威人Ole-Johan Dahl和Kristen Nygaard在挪威计算机中心NCC开发了SIMULA I，随后在67年完成扩展并正式发布SIMULA 67
- Simula是**基于对象**的语言
  - 在SIMULA 67中，数据和处理数据的方法结合在一起作为“抽象数据类型”——类
  - 但最初的SIMULA中，只有类及其实例，并没有类继承的概念

Smalltalk的发展

- 第一个广泛使用的OOPL
  - 20世纪70年代到80年代前期，由Alan Kay领导的团队在美国施乐公司的帕洛阿尔托研究中心（PARC）开发
  - Smalltalk被公认为历史上第二个面向对象的程序设计语言和第一个真正的集成开发环境 (IDE)，并得到了广泛使用
  - Smalltalk对其它众多的程序设计语言的产生起到了极大的推动作用，主要有：Objective-C，Actor， Java 和Ruby等
  - 90年代的许多软件开发思想得利于Smalltalk，例如Design Patterns， Extreme Programming(XP)和Refactoring等
- Smalltalk是动态纯面向对象语言

OO示例：Smalltalk的纯面向对象

在Smalltalk中所有的东西都是对象，或者应该被当作对象处理。

例如下面的表达式：

2 + 3

应当被理解为：向对象“2” 发送消息“+”，参数为对象“3”。

思考：如果对象“2” 无法接受消息“+”，怎么办？

从C上成长

- C++的面向对象之花是生长于C土壤之上
  - Bjarne Stroustrup最初名之为“C with Class”
- C++的目标
  - 首要目标是提供一种语言，像SIMULA 67那样用类和实例来组织程序
  - 另一个重要目标是相对于C来讲，完全没有性能上的损失

从目标上可以看出，C++没有采用Smalltalk动态类型的思路而是沿用C和SIMULA的静态类型！

Java的出现

- Java是由Sun 公司James Gosling主要设计和开发的纯面向对象程序设计语言
  - Java最初被称为Oak，是1991年为消费类电子产品的嵌入式芯片而设计的。
  - 1995年更名为Java，并重新设计用于开发Internet应用程序。
- 现在，Java 作为在 20 世纪 90 年代诞生的最成功的语言，被全世界广泛应用

Java最大的成果在于其简单易用和“一次编写到处运行”的跨平台性！

思考与讨论

- 从 C、Pascal 到 C++、Java 思考方式的差异？
  - 结构化开发方法
  - 面型对象开发方法

- 语言和方法的底层思维方式是一致的
  - 语言的背后是编程模型
  - 方法的背后是分解方式

Reference

[Courtois85] Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28(6), p. 596.

[Ibid] Ibid., p. 221.

[Gall86] Gall, J. 1986. Systemantics: How Systems Really Work and How They Fail. Second Edition. Ann Arbor, MI: The General Systemantics Press, p. 65.

[Matz09]松本行弘,松本行弘的程序世界

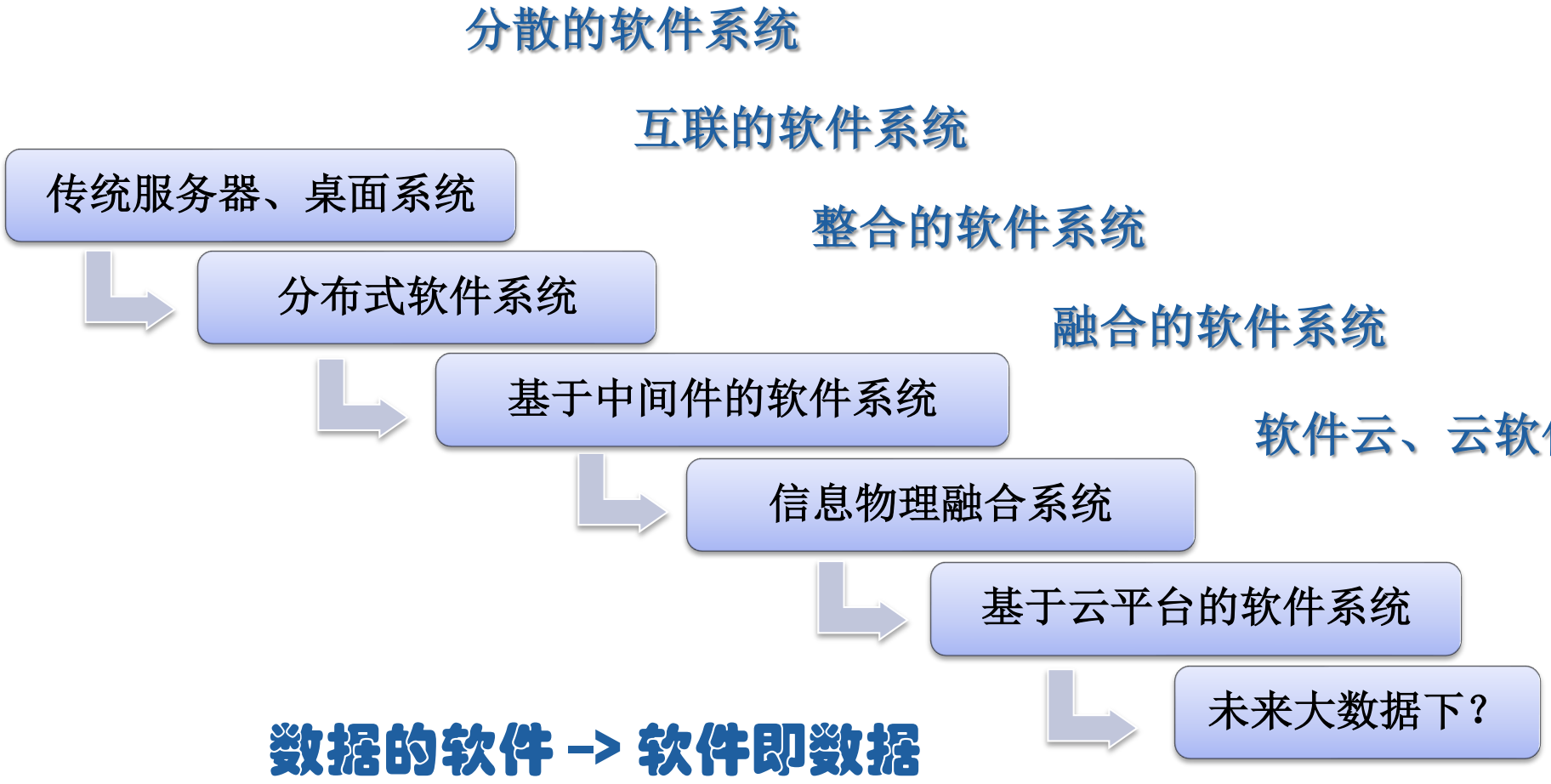
[UnixArt05]Eric Raymond, The Art of UNIX Programming

附录一：程序员

- Ken Thompson、Dijkstr、Knuth和弗洛伊德都曾在图灵奖演说或其他公开场合称自己为“程序员”
- 但现在为什么“程序员”慢慢变成了“码农”？
  - 从高智商的代名词、公司的“白领”，慢慢变成“蓝领”，最后沦落为“码农”
  - 奇怪的是**软件自身的复杂性**却是日益加剧，按道理应该程序员更加重要才对，这两者不成正比呀？
  - 个人感觉，原因应该在于软件工程的发展，相关理论、技术、架构、工具的成熟，使得软件行业也在“社会化分工”！

- 程序员的角色在分化
  - 其实现在的程序员中，仍有如Anders、Kent、Flower这样的顶级人物，但大家已经习惯于称他们为架构师、咨询师或其它
  - 这说明Dijsktra那个时代的“程序员”的概念，由于行业发展而“社会化分工”为多种不同的职业角色了

软件本身形态的演化


分散的软件系统
互联的软件系统
传统服务器、桌面系统
整合的软件系统
分布式软件系统
融合的软件系统
基于中间件的软件系统
软件云、云软件
信息物理融合系统
基于云平台的软件系统
未来大数据下？
数据的软件 -> 软件即数据

软件开发方法日新月异

人月神话：没有“银弹”！

结构化分析与设计

数据驱动的分析与设计

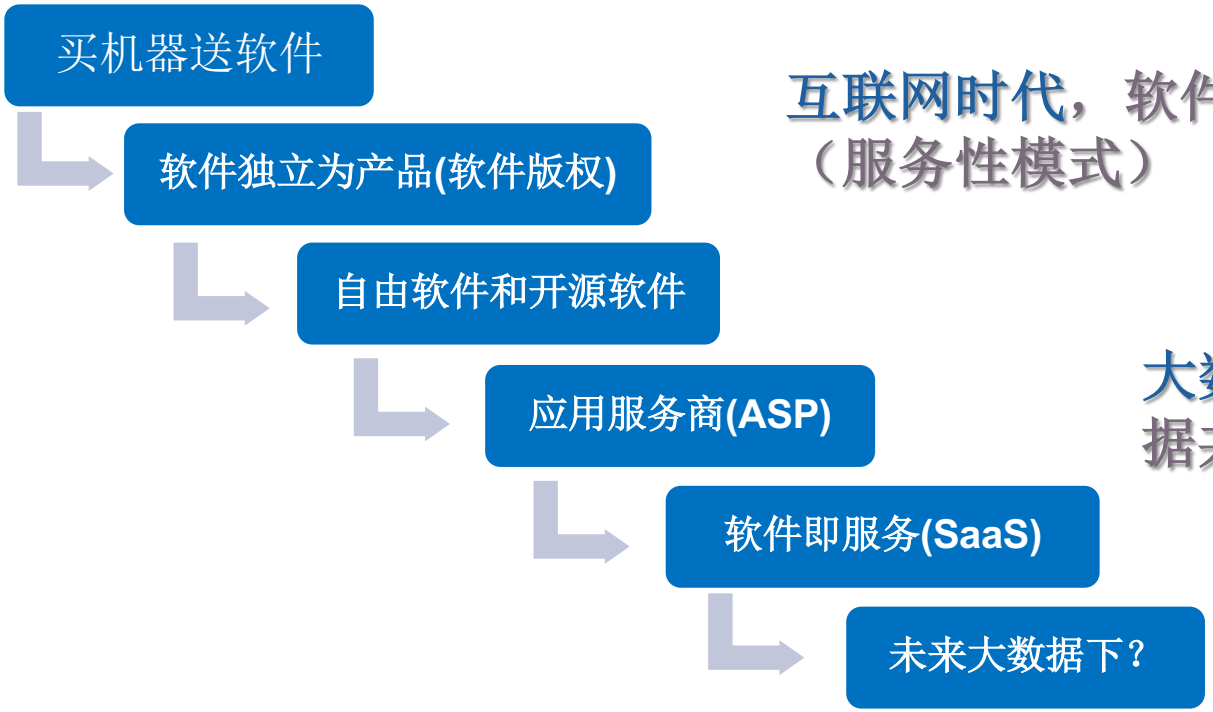
面向对象分析与设计

软件行业的规范难制定

- 不像传统产业，软件行业的规范很难制定
  - **建筑公司不会自己造钢铁厂，为新的大楼提供定制的钢梁，但在软件行业，这种情况时有发生**
  - 软件提供了巨大的灵活性，它诱使开发者打造几乎所有的初级构建模块
  - 建筑行业对原材料的品质有着统一的编码和标准，但软件行业能通用的标准相对而言实在太少

商业模式不断挑战软件的极限

PC时代，软件产业主要与硬件产业共生（传统工业模式）



互联网时代，软件业主要与云计算共生（服务性模式）

大数据时代，软件业主要与数据共生（数据资产型模式）