Architectural Styles
架构风格

Foundations, Theory, and Practice
基础、理论和实践
2
Architectural Styles
架构风格
● Certain design choices regularly result in solutions with superior
properties
某些设计选择通常会导致具有卓越性能的解决方案
Compared to other possible alternatives, solutions such as this
are more elegant, effective, efficient, dependable, evolvable,
scalable, and so on
与其他可能的替代方案相比，这类解决方案更加优雅、有效、高效、可靠、可演变、可扩展等等
● Definition
定义
An architectural style is a named collection of architectural
design decisions that
架构风格是一组命名的架构设计决策，这些决策
● are applicable in a given development context
适用于特定的开发上下文
● constrain architectural design decisions that are specific to a
particular system within that context
限制在该上下文中特定系统的架构设计决策
● elicit beneficial qualities in each resulting system
引出每个生成系统中的有益特性
Architectural styles are a primary way of characterizing lessons from
experience in software system design. Foundations, Theory, and Practice
架构风格是从软件系统设计经验中总结教训的主要方法。基础、理论和实践
3
Some Common Styles
一些常见的风格
● Traditional, language influenced styles
传统的、受语言影响的风格
Main program and
subroutines
主程序和子程序
Object-oriented
面向对象
● Layered
分层
Virtual machines

虚拟机

Client-server

客户端-服务器

- Data-flow styles

数据流风格

Batch sequential

批处理顺序

Pipe and filter

管道和过滤器

- Shared memory

共享内存

Blackboard

黑板

Rule based

基于规则

- Interpreter

解释器

Mobile code

移动代码

- Implicit invocation

隐式调用

Event-based

基于事件

Publish-subscribe

发布-订阅

- Peer-to-peer

点对点

- “Derived” styles

“衍生”的风格

C2

CORBAFoundations, Theory, and Practice

C2

CORBA 基础、理论和实践

- Summary: Decomposition based upon separation of functional processing steps.

总结：基于功能处理步骤的分解。

- Components: ・Main program and subroutines.

组件：・主程序和子程序。

- Connectors: Function/procedure calls.

连接器：函数/过程调用。

- Data elements: Values passed in/out of subroutines.

数据元素：传递给/传递出子程序的值。

- Topology: Static organization of components is hierarchical; full structure is a directed graph.

拓扑结构：组件的静态组织是分层的；完整结构是有向图。
● Additional constraints imposed: None.
施加的其他约束：无。
4
Main Program and SubroutinesFoundations, Theory, and Practice
主程序和子程序基础、理论和实践
● Qualities yielded: modularity. Subroutines may be
replaced with different implementations long as
interface semantics are unaffected.
产生的特性：模块化。只要接口语义不受影响，子程序可以用不同的实现替换。
● Typical uses: Small programs, pedagogical purposes.
典型用途：小型程序，教育目的。
● Cautions: Typically fails to scale to large applications;
inadequate attention to data structures;
Unpredictable effort required to accommodate new
requirements.
注意事项：通常不适用于大型应用程序；对数据结构的关注不足；
适应新需求需要不可预测的努力。
● Relations to programming languages or environments:
与编程语言或环境的关系：
Traditional imperative programming languages, such as
BASIC, Pascal, or C.
传统的命令式编程语言，如 BASIC、Pascal 或 C。

Main Program and Subroutines
主程序和子程序

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Some Common Styles
一些常见的风格

Traditional, language-influenced styles
传统的、受语言影响的风格

Main program and subroutines
主程序和子程序

Object-oriented
面向对象

Layered
分层

Virtual machines
虚拟机

Client-server
客户端-服务器

Data-flow styles
数据流风格

Batch sequential
批处理顺序

Pipe and filter
管道和过滤器

Shared memory
共享内存

Blackboard
黑板

Rule based
基于规则的

Interpreter
解释器

Mobile code
移动代码

Implicit invocation
隐式调用

Event-based
基于事件

Publish-subscribe
发布-订阅

Peer-to-peer

点对点

"Derived" styles
派生的风格

C2
C2

CORBA
CORBA

Object-Oriented Style
面向对象风格

Summary: State encapsulated with functions that operate on that state as objects; object must be instantiated before objects’ methods can be called
总结：状态通过在对象上操作的函数封装；必须在调用对象方法之前实例化对象

Components: objects
组件：对象

Connectors: method invocations (procedure calls to manipulate states)
连接器：方法调用（用于操作状态的过程调用）

Data elements: arguments to methods
数据元素：方法的参数

Qualities yielded: integrity of data operations - data manipulated only by appropriate functions; Abstraction
产生的特性：数据操作的完整性 - 数据只能由适当的函数操作；抽象

Typical use:
典型用途：

- Applications where the designer wants a close correlation between entities in the physical world and entities in the program
- applications involving complex, dynamic data structures.

应用场景包括设计者希望物理世界实体与程序中实体之间存在密切关联的情况，以及涉及复杂、动态数据结构的应用。

Cautions:
注意事项：

- Distributed applications require extensive middleware to provide access to remote objects.
- Inefficient for high-performance applications with large, numeric data structures, such as scientific computing.

分布式应用需要大量的中间件来提供对远程对象的访问。在具有大型数值数据结构（如科学计算）的高性能应用中效率低下。

Relations to PL:
与编程语言的关系：

- Java, C++
- Java，C++

Components are objects
组件是对象

Connectors are messages and method invocations
连接器是消息和方法调用

Style invariants:
风格不变性：

- Objects are responsible for their internal representation integrity
- Internal representation is hidden from other objects

对象对其内部表示的完整性负责，内部表示对其他对象是隐藏的。

Advantages:
优点：

- "Infinite malleability" of object internals
- System decomposition into sets of interacting agents

对象内部的“无限可塑性”，系统分解为一组相互作用的代理。

Disadvantages:
缺点：

- Objects must know identities of servers
- Side effects in object method invocations

对象必须知道服务器的身份，对象方法调用中可能会产生副作用。

Layered Style

分层风格

Hierarchical system organization "Multi-level client-server" Each layer exposes an interface (API) to be used by above layers Each layer acts as a Server: service provider to layers "above" Client: service consumer of layer(s) "below" Connectors are protocols of layer interaction Example: operating systems Virtual machine style results from fully opaque layers
分层系统组织“多层客户端服务器”每个层都暴露一个接口（API）供上面的层使用每个层都充当服务提供者对“上面”的层而言，是服务的消费者对“下面”的层而言，是服务的提供者连接器是层之间交互的协议示例：操作系统虚拟机风格源自完全不透明的层

Layered Systems/Virtual Machines
分层系统/虚拟机

Disk drivers and volume management File manipulations User applications Operating systems designs
磁盘驱动程序和卷管理文件操作用户应用程序操作系统设计

Layered LL
分层（LL）

OO/LL in UML
在 UML 中的面向对象/分层

Layered Style (cont’d)
分层风格（续）

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Layered Style (cont’d)
分层风格（续）

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Layered Style (cont’d)
分层风格（续）

- Summary
由有序的层序列组成；
每个层或虚拟机提供一组服务，可以被驻留在其上的程序访问。
- Components
层，通常包括多个程序
- Connectors
通常是过程调用
- Data elements
在层之间传递的参数
- Topology
线性，对于严格的虚拟机；有向无环图
- Qualities
清晰的依赖结构；软件对下层的变化具有上层的免疫性；下层独立于上层

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Layered Style (cont’d)
分层风格（续）

- Advantages
提高抽象级别
可演变性
对一个层的更改最多影响相邻的两个层
- Reuse
只要保留接口，允许层的不同实现
为库和框架提供标准化的层接口

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Layered Style (cont’d)
分层风格（续）

- Disadvantages
不是普遍适用
性能
可能需要跳过一些层

确定正确的抽象级别

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Some Common Styles
一些常见的风格

- Traditional, language-influenced styles
传统的、受语言影响的风格
- Main program and subroutines
主程序和子程序
- Object-oriented
面向对象
- Layered
分层
- Virtual machines
虚拟机
- Client-server
客户端-服务器
- Data-flow styles
数据流风格
- Batch sequential
批处理顺序
- Pipe and filter
管道和过滤器
- Shared memory
共享内存
- Blackboard
黑板
- Rule based
基于规则的
- Interpreter
解释器
- Mobile code
移动代码
- Implicit invocation
隐式调用
- Event-based
基于事件
- Publish-subscribe

发布-订阅
- Peer-to-peer
点对点
- "Derived" styles
派生的风格
- C2
C2
- CORBA
CORBA

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Client-Server Style
客户端-服务器风格

- Simply, is understood as a two-layer virtual machine with network connections.
简单地说，它被理解为具有网络连接的两层虚拟机。
- The server is the virtual machine below the clients, each of which accesses the virtual machine's interfaces via remote procedure calls or equivalent network access methods.
服务器是位于客户端下方的虚拟机，每个客户端通过远程过程调用或等效的网络访问方法访问虚拟机的接口。
- Typically, there are multiple clients that access the same server; the clients are mutually independent.
通常，有多个客户端访问同一个服务器；客户端之间相互独立。
- The obligation of the server is to provide the specific services requested by each client.
服务器的责任是提供每个客户端请求的特定服务。

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Client-Server Style
客户端-服务器风格

- Components are clients and servers
组件是客户端和服务器
- Servers do not know the number or identities of clients
服务器不知道客户端的数量或身份

- Clients know the server’s identity
客户端知道服务器的身份
- Connectors are RPC-based network interaction protocols
连接器是基于 RPC 的网络交互协议

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Client-Server LL
客户端-服务器（LL）

Other cases?
其他案例？

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Client-Server LL
客户端-服务器（LL）

- Summary: Clients send service requests to the server, which performs the required functions and replies as needed with the requested information. Communication is initiated by the clients.
总结：客户端向服务器发送服务请求，服务器执行所需的功能，并根据需要以请求的信息回复。通信由客户端发起。
- Components: Clients/servers;
组件：客户端/服务器；
- Connectors: Remote procedure call, network protocols.
连接器：远程过程调用，网络协议。
- Data elements: Parameters and return values as sent by the connectors.
数据元素：由连接器发送的参数和返回值。
- Topology: Two levels, with multiple clients making requests to the server.
拓扑结构：两层，多个客户端向服务器发出请求。

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Client-Server LL
客户端-服务器（LL）

- Additional constraints imposed: Client-to-client communication prohibited.
强加的附加约束：禁止客户端之间的通信。
- Qualities yielded: Centralization of computation and data at the server; A single powerful server can service many clients.
产生的特性：在服务器上集中计算和数据；一个强大的服务器可以为多个客户端提供服务。
- Typical uses: Centralization of computation and data are required.
典型用途：需要集中计算和数据。
- Cautions: A large number of client requests.
注意事项：大量的客户端请求。

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Some Common Styles
一些常见的风格

- Traditional, language-influenced styles
传统的、受语言影响的风格
- Main program and subroutines
主程序和子程序
- Object-oriented
面向对象
- Layered
分层
- Virtual machines
虚拟机

- Client-server
客户端-服务器
- Data-flow styles
数据流风格
- Batch sequential
批处理顺序
- Pipe and filter
管道和过滤器
- Shared memory
共享内存
- Blackboard
黑板
- Rule based
基于规则的
- Interpreter
解释器
- Mobile code
移动代码
- Implicit invocation
隐式调用
- Event-based
基于事件
- Publish-subscribe
发布-订阅
- Peer-to-peer
点对点
- "Derived" styles
派生的风格
- C2
C2
- CORBA
CORBA

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Data-Flow Styles
Batch Sequential
- Separate programs are executed in order; data is passed as an aggregate from one program to the next.
独立的程序按顺序执行；数据作为聚合从一个程序传递到下一个。

- Connectors: “The human hand” carrying tapes between the programs, a.k.a. “sneaker-net ”
连接器：“人手”在程序之间携带磁带，又称“运动鞋网络”。
- Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution.
数据元素：显式的、在生成程序执行完成后从一个组件传递到下一个组件的聚合元素。
- Typical uses: Transaction processing in financial systems. “The Granddaddy of Styles” concerns the movement of data between independent processing elements.
典型用途：金融系统中的事务处理。“风格之祖”涉及独立处理元素之间的数据传递。

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Batch-Sequential: A Financial Application
批处理顺序：金融应用程序

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Batch-Sequential LL
批处理顺序（LL）

Not a recipe for a successful lunar mission!
这不是一次成功的月球任务的配方！

Not a highly interactive, real-time game!
不是一款高度交互、实时的游戏！

Foundations, Theory, and Practice
基础理论与实践

Software Architecture
软件架构

Data-Flow Styles
Batch Sequential
- Summary: Separate programs are executed in order; data is passed as an aggregate from one program to the next.
总结：独立的程序按顺序执行；数据作为聚合从一个程序传递到下一个。
- Components: Independent programs.
组件：

Data-Flow Styles
数据流风格

Batch Sequential
批处理顺序

Additional constraints imposed:
额外的约束：

One program runs at a time, to completion.
一次只运行一个程序，直至完成。

Qualities yielded:
产生的特性：

Simplicity.
简单性。

Typical uses:
典型用途：

Transaction processing in financial systems.
金融系统中的事务处理。

Cautions:
注意事项：

When interaction between the components is required; when concurrency between components is possible or required.
当组件之间需要交互时；当组件之间可能或需要并发时。

Relations to programming languages or environments:
与编程语言或环境的关系：

None.

无。

Some Common Styles
一些常见风格

Traditional, language-influenced styles
传统的、受语言影响的风格

Main program and subroutines
主程序和子程序

Object-oriented
面向对象

Layered
分层

Virtual machines
虚拟机

Client-server
客户端-服务器

Data-flow styles
数据流风格

Batch sequential
批处理顺序

Pipe and filter
管道和过滤器

Shared memory
共享内存

Blackboard
黑板

Rule-based
基于规则的

Interpreter
解释器

Interpreter
解释器

Mobile code
移动代码

Implicit invocation
隐式调用

Event-based
基于事件

Publish-subscribe
发布-订阅

Peer-to-peer
点对点

Derived styles
派生风格

C2
C2

CORBA
CORBA

Shared Memory Style
共享内存风格

The essence of shared state styles (sometimes colloquially referred to as shared memory styles) is that multiple components have access to the same data store and communicate through that data store.
共享内存风格的本质（有时口头上称为共享内存风格）是多个组件可以访问相同的数据存储并通过该数据存储进行通信。

This corresponds roughly to the ill-advised practice of using global data in C or Pascal programming.
这大致对应于在 C 或 Pascal 编程中使用全局数据的不良做法。

The difference is that with shared state styles, the center of design attention is explicitly on these structured, shared repositories, and consequently, they are well-ordered and carefully managed.
不同之处在于，共享内存风格将设计关注点明确放在这些结构化的共享存储库上，因此它们是井然有序且经过精心管理的。

Blackboard Style
黑板风格

- Two kinds of components
两种组件

Central data structure — blackboard
中央数据结构 — 黑板

Components operating on the blackboard
在黑板上操作的组件

- System control is entirely driven by the blackboard state
系统控制完全由黑板状态驱动

- Examples
示例

Typically used for AI systems
通常用于人工智能系统

Integrated software environments (e.g., Interlisp)
集成软件环境（例如，Interlisp）

Compiler architecture
编译器架构

Blackboard LL
黑板风格（LL）

- Summary: Independent programs access and communicate exclusively through a global data repository, known as a blackboard.
摘要：独立程序通过全局数据存储库（称为黑板）进行访问和通信。

- Components: Independent programs, sometimes referred to as "knowledge sources," blackboard.
组件：独立程序，有时称为“知识源”，黑板。

- Connectors: Access to the blackboard may be by direct memory reference or can be through a procedure call or a database query.
连接器：对黑板的访问可以通过直接内存引用，也可以通过过程调用或数据库查询。

- Data elements: Data stored in the center.

数据元素：存储在中心的数据。

- Topology: Star topology, with the blackboard at the center.
拓扑结构：星型拓扑结构，黑板位于中心。

- Qualities yielded: Complete solution strategies to complex problems do not have to be preplanned. Evolving views of data/problem determine the strategies that are adopted.
产生的特性：对于复杂问题，不必事先规划完整的解决策略。数据/问题的不断演变决定了采用的策略。

- Typical uses: Heuristic problem solving in artificial intelligence applications.
典型用途：启发式问题解决在人工智能应用中。

- Cautions: When a well-structured solution strategy is available; when interactions between the independent programs require complex regulation; when representation of the data on the blackboard is subject to frequent change.
注意事项：当存在良好结构的解决策略时；当独立程序之间的交互需要复杂的调节时；当黑板上的数据表示经常发生变化时。

Rule-Based/Expert Style
基于规则/专家风格

Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query. The shared memory is a so-called knowledge base.
推理引擎解析用户输入并确定其是否为事实/规则或查询。如果是事实/规则，它将此条目添加到知识库。否则，它会查询适用的规则，并尝试解决查询。共享内存被称为知识库。
Rule-Based Style (cont’d)
基于规则的风格（续）

- Components: User interface, inference engine, knowledge base
组件：用户界面、推理引擎、知识库

- Connectors: Components are tightly interconnected, with direct procedure calls and/or shared memory.
连接器：组件之间紧密相连，通过直接过程调用和/或共享内存。

- Data Elements: Facts and queries
数据元素：事实和查询

- Behavior of the application can be very easily modified through addition or deletion of rules from the knowledge base.
通过向知识库添加或删除规则，可以非常容易地修改应用程序的行为。

- Caution: When a large number of rules are involved, understanding the interactions between multiple rules affected by the same facts can become very difficult.

注意事项：当涉及大量规则时，理解受相同事实影响的多个规则之间的交互可能变得非常困难。

Rule Based LL
基于规则的风格（LL）

IF fact1 & fact2
THEN results
如果 fact1 和 fact2 那么 结果

Interpreter Style
解释器风格

- The distinctive characteristic of interpreter styles is dynamic, on-the-fly interpretation of commands.

解释器风格的独特特点是对命令进行动态、即时的解释。

- Commands are explicit statements, possibly created moments before they are executed, possibly encoded in human-readable and editable text.

命令是明确的语句，可能是在执行前的瞬间创建的，可能以人类可读和可编辑的文本编码。

- Interpretation proceeds by starting with an initial execution state, obtaining the first command to execute, executing the command over the current execution state, thereby modifying that state, then proceeding to execute the next command.

解释是通过从初始执行状态开始，获取要执行的第一个命令，执行该命令以修改当前执行状态，然后继续执行下一个命令进行的。

Interpreter Style
解释器风格

- Excel ’ s formulas are in fact commands interpreted by the Excel execution engine-the interpreter.

实际上，Excel 的公式是由 Excel 执行引擎（解释器）解释的命令。

- Excel's macros are interpreted by the Visual Basic interpreter.

Excel 的宏由 Visual Basic 解释器解释。

Interpreter Style
解释器风格

- Summary: Interpreter parses and executes input commands, updating the state maintained

by the interpreter
摘要：解释器解析并执行输入命令，更新由解释器维护的状态。

- Components: Command interpreter, program/interpreter state, user interface.

组件：命令解释器、程序/解释器状态、用户界面。

- Connectors: Typically very closely bound with direct procedure calls and shared state.

连接器：通常与直接过程调用和共享状态紧密相连。

- Quality yielded: Highly dynamic behavior possible, where the set of commands is dynamically modified. System architecture may remain constant while new capabilities are created based upon existing primitives.

产生的特性：可能具有高度动态的行为，其中命令集是动态修改的。系统架构可能保持不变，同时基于现有原语创建新的能力。

- Uses: Superb for end-user programmability; supports dynamically changing set of capabilities

用途：非常适用于最终用户的可编程性；支持动态更改的能力集

- Lisp and Scheme

Lisp 和 Scheme

Interpreter Style
解释器风格

- Caution:

注意：

When fast processing is needed;
当需要快速处理时；

it takes longer to execute interpreted code than executable code;
执行解释的代码比可执行的代码需要更长的时间；

memory management may be an issue, especially when multiple interpreters are invoked simultaneously.
内存管理可能是一个问题，特别是当同时调用多个解释器时。

Interpreter LL
解释器风格（LL）

"BurnRate(50)": the interpreter takes BurnRate as the command and the parameter as the amount of fuel to burn. It then calculates the necessary updates to the altitude, fuel level, and velocity. "CheckStatus": the user will receive the current state of altitude, fuel, time, and velocity.
“BurnRate(50)”: 解释器将 BurnRate 视为命令，参数为要燃烧的燃料量。然后，它计算了

更新海拔、燃料水平和速度所需的更新。 "CheckStatus": 用户将收到海拔、燃料、时间和速度的当前状态。

Some Common Styles (continued)
一些常见的风格（续）

- Traditional, language-influenced styles

传统的、受语言影响的风格
- Main program and subroutines
主程序和子程序
- Object-oriented
面向对象的
- Layered
分层
- Virtual machines
虚拟机
- Client-server
客户端-服务器

- Data-flow styles

数据流风格
- Batch sequential
批量顺序
- Pipe and filter
管道和过滤器
- Shared memory
共享内存
- Blackboard
黑板
- Rule based
基于规则
- Interpreter
解释器
- Mobile code
移动代码
- Implicit invocation
隐式调用
- Event-based
基于事件
- Publish-subscribe
发布-订阅
- Peer-to-peer
对等网络

- "Derived" styles

“衍生”风格

- C2

C2

- CORBA

CORBA

Mobile-Code Style

移动代码风格

- Mobile code styles enable code to be transmitted to a remote host for interpretation.

移动代码风格允许代码被传输到远程主机进行解释。

- This may be due to a lack of local computing power, lack of resources, or due to large data sets remotely located.

这可能是由于缺乏本地计算能力、缺乏资源，或者由于远程位置上的大型数据集。

Mobile-Code Style

移动代码风格

- Summary: a data element (some representation of a program) is dynamically transformed into a data processing component.

总结：数据元素（程序的某种表示）被动态地转化为数据处理组件。

- Components: “ Execution dock ” , which handles receipt of code and state; code compiler/interpreter.

组件：“执行码坞”，负责处理代码和状态的接收；代码编译器/解释器。

- Connectors: Network protocols and elements for packaging code and data for transmission.

连接器：网络协议和用于封装代码和数据以进行传输的元素。

- Data Elements: Representations of code as data; program state; data.

数据元素：代码的表示作为数据；程序状态；数据。

- Variants: Code-on-demand, Remote evaluation, and Mobile agent.

变体：按需代码、远程评估和移动代理。

Mobile Code LL

移动代码风格（LL）

Scripting languages (i.e., JavaScript, VBScript), ActiveX control, embedded Word/Excel macros.

脚本语言（例如 JavaScript、VBScript）、ActiveX 控件、嵌入的 Word/Excel 宏。

Code-on-demand

从服务器下载脚本代码；游戏逻辑移动到客户端。每个客户端都独立地维护游戏状态。

Some Common Styles (continued)

一些常见的风格（续）

- Traditional, language-influenced styles

传统的、受语言影响的风格

- Main program and subroutines

主程序和子程序

- Object-oriented

面向对象的

- Layered

分层

- Virtual machines

虚拟机

- Client-server

客户端-服务器

- Data-flow styles

数据流风格

- Batch sequential

批量顺序

- Pipe and filter

管道和过滤器

- Shared memory

共享内存

- Blackboard

黑板

- Rule based

基于规则

- Interpreter

解释器

- Mobile code

移动代码

- Implicit invocation

隐式调用

- Publish-subscribe

发布-订阅

- Event-based

基于事件

- Peer-to-peer

对等网络

Implicit Invocation Style

隐式调用风格

- Event announcement instead of method invocation

使用事件公告而不是方法调用

- “Listeners” register interest in and associate methods with events

“侦听器”注册对事件的兴趣并将方法与事件关联

- System invokes all registered methods implicitly

系统隐式调用所有注册的方法

● Component interfaces are methods and events

组件接口是方法和事件

● Two types of connectors

两种类型的连接器

- Invocation is either explicit or implicit in response to events

调用是对事件的响应中明确的还是隐式的

● Style invariants

风格不变性

- “Announcers” are unaware of their events’ effects

“广播者”不知道它们的事件的影响

- No assumption about processing in response to events

对事件响应中的处理没有假设

Implicit Invocation (cont’d)

隐式调用（续）

● Advantages

优势

- Component reuse; loosely coupled components

组件重用；松散耦合的组件

- System evolution

系统演进

● Both at system construction-time & run-time

既在系统构建时又在运行时

● Disadvantages

劣势

- Counter-intuitive system structure

反直觉的系统结构

- Components relinquish computation control to the system

组件将计算控制权交给系统

- No knowledge of what components will respond to event

不知道哪些组件将响应事件

- No knowledge of order of responses

不知道响应的顺序

Publish-Subscribe

发布-订阅

● Subscribers register/deregister to receive specific messages or specific content.

订阅者注册/注销以接收特定消息或特定内容。

- Publishers broadcast messages to subscribers either synchronously or asynchronously.

发布者向订阅者同步或异步地广播消息。

Publish-Subscribe (cont’d)
发布-订阅（续）

● Components: Publishers, subscribers, proxies for managing distribution
组件：发布者、订阅者、用于管理分发的代理
● Connectors: Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
连接器：通常需要网络协议。基于内容的订阅需要复杂的连接器。
● Data Elements: Subscriptions, notifications, published information
数据元素：订阅、通知、发布的信息
● Topology: Subscribers connect to publishers either directly or

may receive notifications via a network protocol from intermediaries
拓扑结构：订阅者直接连接到发布者，或通过网络协议从中间人那里接收通知
● Qualities yielded: Highly efficient one-way dissemination of information with very low-coupling of components
产生的特性：高效的单向信息传播，组件之间的耦合非常低

Event-Based Style
基于事件的风格

● Independent components asynchronously emit and receive events communicated over event buses.
独立的组件异步地发出和接收通过事件总线传递的事件。
● Components: Independent, concurrent event generators and/or consumers.
组件：独立的、并发的事件生成器和/或消费者。
● Connectors: Event buses (at least one).
连接器：事件总线（至少一个）。
● Data Elements: Events – data sent as a first-class entity over the event bus.
数据元素：事件 - 作为第一类实体通过事件总线发送的数据。
● Topology: Components communicate with the event buses, not directly to each other.
拓扑结构：组件通过事件总线进行通信，而不是直接相互通信。
● Variants: Component communication with the event bus may either be push or pull based.
变体：组件与事件总线的通信可以是推送或拉取方式。
● Highly scalable, easy to evolve, effective for highly distributed applications.
高度可扩展，易于演变，对于高度分布式应用非常有效。

Event-Based Style
基于事件的风格

● Event-based vs. publish-subscribe
基于事件与发布-订阅的区别

- There is no classification of components into pub. and sub.
没有将组件分类为发布者和订阅者。
- All components potentially both emit and receive events.
所有组件可能既发出事件又接收事件。
- Event-based style is suited to strongly decoupled concurrent components, where at any given time a component either may be creating information of potential interest to others or may be consuming information.
基于事件的风格适用于强烈解耦的并发组件，其中在任何给定时间，组件可能正在创建对其他组件有潜在兴趣的信息，也可能正在消耗信息。

Peer-to-Peer Style
点对点风格

- State and behavior are distributed among peers which can act as either clients or servers.
状态和行为分布在可以充当客户端或服务器的对等体之间。
- Peers: independent (autonomous) components, having their own state and control thread.
对等体：独立（自主）组件，具有自己的状态和控制线程。
- Connectors: Network protocols, often custom.
连接器：网络协议，通常是定制的。
- Data Elements: Network messages.
数据元素：网络消息。
- State and logic are decentralized on peers.
状态和逻辑在对等体上分散。
- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically.
拓扑结构：网络（对等体之间可能存在冗余连接）；可以任意和动态变化。
- Supports decentralized computing with flow of control and resources distributed among peers.
支持分散计算，控制流和资源在对等体之间分布。
- Highly robust in the face of failure of any given node.
在任何给定节点失败的情况下具有高度鲁棒性。
- Scalable in terms of access to resources and computing power.
在资源访问和计算能力方面具有可伸缩性。

But caution on the protocol!
但是需要注意协议！
Peer-to-Peer LL
点对点 LL

LL3 what to obtain
some information from
the group of LLs:
LL3->LL1->LL2->LL5-
>LL2->LL1->LL3
LL3 想要从 LL 组中获取一些信息：

LL3->LL1->LL2->LL5-
>LL2->LL1->LL3
Foundations, Theory, and Practice
Software Architecture
78

Some Common Styles (continued)
一些常见的风格（续）

- Traditional, language-influenced styles
传统的、受语言影响的风格
- Main program and subroutines
主程序和子程序
- Object-oriented
面向对象的
- Layered
分层
- Virtual machines
虚拟机
- Client-server
客户端-服务器

- Data-flow styles
数据流风格
- Batch sequential
批量顺序
- Pipe and filter
管道和过滤器
- Shared memory
共享内存
- Blackboard
黑板
- Rule based
基于规则
- Interpreter
解释器
- Mobile code
移动代码
- Implicit invocation
隐式调用
- Publish-subscribe
发布-订阅
- Event-based
基于事件

- Peer-to-peer
对等网络

The C2 Style
C2 风格

● A general multi-tier architectural style
一种通用的多层架构风格
- Asynchronous message-based integration of independent components
独立组件的异步消息集成
● Based on past experience with the model-view-controller design pattern
基于对模型-视图-控制器设计模式的过去经验
- Generalizes architectures for user interface management systems
概括了用户界面管理系统的体系结构
● Notable feature: flexible connectors
显著特点：灵活的连接器
- Accommodate arbitrary numbers of components
适应任意数量的组件
- Facilitate runtime architectural change
促进运行时架构变更
Foundations, Theory, and Practice
Software Architecture

The C2 Style
C2 风格

● Principle of substrate independence
基质独立原则
- A component need not know anything about components beneath it
组件无需了解其下面的组件
- We’ll define the layering rules and the notion of “beneath” shortly…
我们将很快定义分层规则和“下方”的概念…
● Rich design environment for composition & analysis
用于构成和分析的丰富设计环境
● Autonomous components
自主组件
- Own thread and non-shared address space
拥有独立的线程和非共享的地址空间
● All communication is by asynchronous events which must pass through a connector
所有通信都通过必须经过连接器的异步事件
- Connectors may be complex and powerful
连接器可以是复杂且强大的
● Some additional rules to promote reuse
一些额外的规则以促进重用

The C2 Style (continued)
C2 风格（续）

● Asynchronous, event-based communication among autonomous components, mediated by active connectors
异步的、基于事件的通信在自主组件之间，由主动连接器中介
● No component-component links
没有组件与组件之间的链接
● Event-flow rules
事件流规则
● Hierarchical application
层次应用
- Notifications fall
通知下降
- Requests rise
请求上升
- Component
组件
- Connector
连接器
- “Push”
“推”
- “Pull”
“拉”
Connector
连接器
The C2 Style
C2 风格

Some Common Styles (continued)
一些常见的风格（续）

● Traditional, language-influenced styles
传统的、受语言影响的风格
- Main program and subroutines
主程序和子程序
- Object-oriented
面向对象的

- Layered
分层
- Virtual machines
虚拟机
- Client-server
客户端-服务器

- Data-flow styles

数据流风格
- Batch sequential
批量顺序
- Pipe and filter
管道和过滤器
- Shared memory
共享内存
- Blackboard
黑板
- Rule based
基于规则
- Interpreter
解释器
- Mobile code
移动代码

- Implicit invocation

隐式调用
- Publish-subscribe
发布-订阅
- Event-based
基于事件
- Peer-to-peer
对等网络

Distributed Objects
分布式对象

- The fundamental vocabulary comes, of course, from the simple object-oriented style.

基本的词汇当然来自简单的面向对象风格。

- OO style is augmented with the client-server style to provide the notion of distributed objects, with access to those objects from, potentially, different processes executing on different computers.

面向对象风格与客户端-服务器风格相结合，以提供分布式对象的概念，可以从潜在地在不同计算机上执行的不同进程中访问这些对象。

- Objects are instantiated on different hosts, each exposing a public interface.

对象在不同的主机上实例化，每个对象都暴露一个公共接口。

● The interface: all parameters and return values must be serializable so they can go over the network.
接口：所有参数和返回值都必须是可序列化的，以便它们可以通过网络传输。
● Interaction between objects: synchronous procedure call; asynchronous forms such as in CORBA.
对象之间的交互：同步过程调用；异步形式，如在 CORBA 中。
Foundations, Theory, and Practice
Software Architecture

Distributed Objects (continued)
分布式对象（续）

● CORBA is a standard for implementing middleware that supports the development of applications composed of distributed objects.
CORBA 是一个用于实现中间件的标准，支持由分布式对象组成的应用程序的开发。
● The basic idea behind CORBA is that

an application is broken up into objects, which are effectively software components that expose one or more provided interfaces.
CORBA 背后的基本思想是将应用程序分解为对象，这些对象实际上是公开一个或多个提供的接口的软件组件。
● These provided interfaces are specified in terms of a programming language- and platform-neutral notation called the Interface Definition Language (IDL).
这些提供的接口是使用称为接口定义语言（IDL）的与编程语言和平台无关的符号来指定的。
Foundations, Theory, and Practice
Software Architecture

Distributed Objects (continued)
分布式对象（续）

● Remote vs. local procedure call
远程调用 vs. 本地调用
- Remote call: all data must be serializable.
远程调用：所有数据必须可序列化。
- Remote call suffers from a much wider variety of potential failures than local calls.
远程调用受到比本地调用更广泛的潜在故障的影响。
Foundations, Theory, and Practice
Software Architecture