Designing Architectures
设计架构

- **Architectural Styles**
  - 架构风格
    - **Named collection of design decisions**（命名的设计决策集合）
    - **Reflection of experience**（经验的反映）
    - **Less domain-specific than patterns**（比模式更少领域特定）

- **Definitions of Architectural Style**
  - 架构风格定义
    - **Collection of decisions in a context**（特定上下文中的决策集合）
    - **Constrain system-specific decisions**（限制系统特定的决策）
    - **Elicit beneficial qualities**（引出有益的特质）
    - **Shared understanding of forms**（共享的形式理解）

- **Basic Properties of Styles**
  - 风格基本属性
    - **Design elements vocabulary**（设计元素词汇）
    - **Configuration rules**（配置规则）
    - **Semantic interpretation**（语义解释）
    - **Possible system analyses**（可能的系统分析）

- **Benefits of Using Styles**
  - 使用风格的好处
    - **Design and code reuse**（设计和代码重用）
    - **Understandability**（易理解性）
    - **Interoperability**（互操作性）
    - **Style-specific analyses**（特定风格的分析）
    - **Visualizations**（可视化）

- **Style Analysis Dimensions**
  - 风格分析维度
    - **Design vocabulary**（设计词汇）
    - **Structural patterns**（结构模式）
    - **Computational model**（计算模型）
    - **Essential invariants**（基本不变量）
    - **Examples of use**（使用示例）
    - **(Dis)advantages**（优缺点）
    - **Specializations**（特化）

Designing Architectures
设计架构

Where do architectures come from?
架构从哪里来？

- Method
  方法
  - Efficient in familiar terrain
    在熟悉的领域中高效
  - Not always successful
    不总是成功的
  - Predictable outcome (+ & -)
    可预测的结果（正面和负面）
  - Quality of methods varies
    方法的质量各异

- Creativity
  创造力
  - Fun!
    有趣！
  - Fraught with peril
    充满危险
  - May be unnecessary
    可能是不必要的
  - May yield the best result
    可能得到最佳结果

How Do You Design?Foundations, Theory, and Practice
你如何设计？基础、理论与实践

Software Architecture
软件架构

Objectives
目标

- Creativity
  创造力
  - Enhance your skill-set
    提升你的技能集
  - Provide new tools
    提供新工具
- Method
  方法

- Focus on highly effective techniques
  专注于高效的技术
- Develop judgment: when to develop novel solutions, and when to follow established method
  培养判断力：何时开发新的解决方案，何时遵循已建立的方法

Engineering Design Process
工程设计过程

- Feasibility stage: identifying a set of feasible concepts for the design as a whole.
  可行性阶段：确定整体设计的一组可行概念。
- Preliminary design stage: selection and development of the best concept.
  初步设计阶段：选择和开发最佳概念。
- Detailed design stage: development of engineering descriptions of the concept. (work in parallel)
  详细设计阶段：开发概念的工程描述。（并行工作）
- Planning stage: evaluating and altering the concept to suit the requirements of production, distribution, consumption, and product retirement. (work in parallel)
  计划阶段：评估和修改概念，以满足生产、分发、消费和产品退役的要求。（并行工作）

Potential Problems
潜在问题

- If the designer is unable to produce a set of feasible concepts, progress stops.
  如果设计者无法提出一组可行的概念，进展将停滞。
- As problems and products increase in size and complexity, the probability that any one individual can successfully perform the first steps decreases.
  随着问题和产品的规模和复杂性增加，任何一个个体成功完成第一步的可能性减小。
- The standard approach does not directly address the situation where system design is at stake, i.e. when the relationship between a set of products is at issue.
  标准方法并没有直接解决系统设计处于危险境地的情况，即当一组产品之间的关系成为问题时。
- As complexity increases or the experience of the designer is not sufficient, alternative approaches to the design process must be adopted.
  随着复杂性的增加或设计者的经验不足，必须采用设计过程的替代方法。

Alternative Design Strategies
替代设计策略

- Standard
  标准
  - Linear model described above
    上述的线性模型
- Cyclic
  循环
  - Process can revert to an earlier stage
    过程可以回到较早的阶段
- Parallel

并行

- Independent alternatives are explored in parallel

  独立的替代方案并行探索

- Adaptive (“lay tracks as you go”)

自适应（“边走边铺轨”）

- The next design strategy of the design activity is decided at the end of a given stage

  设计活动的下一个设计策略在给定阶段结束时决定

- Incremental

增量

- Each stage of development is treated as a task of incrementally improving the existing design

  发展的每个阶段被视为逐步改进现有设计的任务

Identifying a Viable Strategy

识别可行的策略

- Use fundamental design tools

: abstraction and modularity.

使用基本设计工具：抽象和模块化。

But how?

但是如何？

- Inspiration, where inspiration is needed. Predictable techniques elsewhere.

灵感，需要灵感的地方。在其他地方使用可预测的技术。

But where is creativity required?

但创造力在哪里需要？

- Applying own experience or experience of others.

应用自己的经验或他人的经验。

万事开头难： how to identify that set of viable arrangement

The Tools of “Software Engineering 101”

“软件工程 101”的工具

- Abstraction

抽象

- Abstraction(1): look at details, and abstract “up” to concepts

  抽象（1）：查看细节，向概念抽象“上升”

- Abstraction(2): choose concepts, then add detailed substructure, and move “down”

  抽象（2）：选择概念，然后添加详细的子结构，并向“下”移动

- Example: design of a stack class

示例：栈类的设计

- Separation of concerns

关注点分离

A Few Definitions… from the OED

一些定义...来自 OED

Online

在线

- Abstraction: “The act or process of separating in thought, of considering a thing independently of its associations; or a substance independently of its attributes; or an attribute or quality independently of the substance to which it belongs.”

抽象：“在思想中分离的行为或过程，独立考虑一件事物，或独立考虑物质而不考虑其属性，或独立考虑属性或品质而不考虑其所属的物质。”

- Reification: “The mental conversion of … [an] abstract concept into a thing.”

具体化：“将抽象概念转化为事物的心理过程。”

- Deduction: “The process of drawing a conclusion from a principle already known or assumed; spec. in Logic, inference by reasoning from generals to particulars; opposed to INDUCTION.”

推断：“从已知或假定的原则中得出结论的过程；特别是在逻辑学中，由从概括到特殊的推理得出的推断；与归纳相对。”

- Induction: “The process of inferring a general law or principle from the observation of particular instances (opposed to DEDUCTION, q.v.).”

归纳：“通过观察特定实例推断一般法则或原则的过程（与推断相对）。”

Abstraction and the Simple Machines

抽象和简单机器

- What concepts should be chosen at the outset of a design task?

设计任务开始时应选择哪些概念？

One technique: Search for a “simple machine” that serves as an abstraction of a potential system that will perform the required task

一种技术：寻找作为潜在系统抽象的“简单机器”，该系统将执行所需的任务

For instance, what kind of simple machine makes a software system embedded in a fax machine?

例如，嵌入传真机中的软件系统使用了哪种简单机器？

- At core, it is basically just a little state machine.

从本质上说，它基本上只是一个小的状态机。

- Simple machines provide a plausible first conception of how an application might be built.

简单机器提供了一个合理的第一构想，即应用程序可能如何构建。

- Every application domain has its common simple machines. (experience)

每个应用领域都有其常见的简单机器。（经验）

Simple Machines

简单机器

Domain

领域

Simple Machines

简单机器

Graphics

图形

Pixel arrays

像素阵列

Transformation matrices

变换矩阵
Widgets
小部件
Abstract depiction graphs
抽象描绘图
Word processing
文字处理
Structured documents
结构化文档
Layouts
布局

Process control
过程控制
- Finite state machines
  有限状态机
- Income Tax Software
  所得税软件
- Hypertext
  超文本
- Spreadsheets
  电子表格
- Form templates
  表单模板
- Web pages
  网页
- Hypertext
  超文本
- Composite documents
  复合文档
- Scientific computing
  科学计算
- Matrices
  矩阵
- Mathematical functions
  数学函数
- Banking
  银行业
- Spreadsheets
  电子表格
- Databases
  数据库
- Transactions
  事务

Choosing the Level and Terms of Discourse
选择层次和论述术语

- Any attempt to use abstraction as a tool must choose a level of discourse, and once that is chosen, must choose the terms of discourse.
  任何尝试使用抽象作为工具的企图必须选择一种层次的论述，并且一旦选择了，必须选择论述的术语。
- Alternative 1: initial level of discourse is one of the application as a whole (step-wise refinement).
  替代方案 1：初始层次的论述是整个应用程序的一个（逐步细化）。
- Alternative 2: work, initially, at a level lower than that of the whole application. Once several such sub-problems are solved they can be composed together to form an overall solution.
  替代方案 2：最初在低于整个应用程序的层次上工作。一旦解决了几个这样的子问题，它们可以组合在一起形成整体解决方案。
- Alternative 3: work, initially, at a level above that of the desired application. E.g. handling simple application input with a general parser.
  替代方案 3：最初在高于所需应用程序的层次上工作。例如，使用通用解析器处理简单的应用程序输入。

Separation of Concerns
关注点分离
- Separation of concerns is the subdivision of a problem into (hopefully) independent parts.
  关注点分离是将问题划分为（希望是）独立的部分。
- The difficulties arise when the issues are either actually or apparently intertwined.
  当问题实际上或表面上相互交织时，困难就会产生。
- Separations of concerns frequently involve many tradeoffs.
  关注点的分离通常涉及许多权衡。
- Total independence of concepts may not be possible.
  概念的完全独立可能是不可能的。
- Key example from software architecture: separation of components (computation) from connectors (communication).
  软件架构的关键示例：将组件（计算）与连接器（通信）分离。

The Grand Tool: Refined Experience
大器：精炼经验

- Experience must be reflected upon and refined.
  经验必须被反思和精炼。
- The lessons from prior work include not only the lessons of successes but also the lessons arising from failure.
  从先前工作中得出的教训不仅包括成功的教训，还包括从失败中得出的教训。
- Learn from the success and failure of other engineers.
  向其他工程师的成功和失败学习。

- Literature, Conferences
  文献，会议
- Experience can provide that initial feasible set of "alternative arrangements for the design as a whole."
  经验可以提供设计整体的初始可行的“替代安排集”。

马丁·福勒（Martin Fowler）
- 敏捷开发方法论
- 企业应用架构模式
- 微服务架构
- 重构

Robert C. Martin（罗伯特·C·马丁）
- 极限编程
- 软件设计原则
- Clean code(r)

Patterns, Styles, and DSSAs
模式、风格和 DSSAs（Design Space of Software Architecture）

Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.
软件架构：基础、理论和实践；Richard N. Taylor，Nenad Medvidovic 和 Eric M. Dashofy；© 2008 John Wiley & Sons，Inc. 经许可再版。

Domain-Specific Software Architectures
领域特定软件架构

- DSSAs encode substantial knowledge, acquired through extensive experience, about how to structure complete applications within a particular domain.
  DSSAs 编码通过广泛经验获取的大量知识，关于如何在特定领域内构建完整的应用程序。
- A useful operational definition of DSSAs is the combination of
  有用的 DSSAs 的操作定义是以下组合
  - (1) a reference architecture for an application
    一个应用程序的参考架构
  - (2) a library of software components for that architecture containing reusable chunks of domain expertise
    该架构的软件组件库，包含可重用的领域专业知识块
  - (3) a method of choosing and configuring components to work within an instance of the reference architecture.
    选择和配置组件以在参考架构的实例中工作的方法。
- A DSSA represents the most valuable type of experience useful in identifying a feasible set of "alternative arrangements for the design as a whole."
  DSSA 代表了在识别整体设计的可行替代方案集时最有价值的经验类型。

- A DSSA is an assemblage of software components specialized for a particular type of task (domain), generalized for effective use across that domain, and composed in a standardized structure (topology) effective for building successful applications.

DSSA 是一组专为特定类型任务（领域）而设计的软件组件，广泛用于该领域的有效使用，并以标准化结构（拓扑）组成，适用于构建成功应用程序。

- Since DSSAs are specialized for a particular domain, they are only of value if one exists for the domain wherein the engineer is tasked with building a new application.

由于 DSSAs 专为特定领域而设计，只有在工程师负责构建新应用程序的领域存在时才有价值。

- DSSAs are the preeminent means for maximal reuse of knowledge and prior development and hence for developing a new architectural design.

DSSAs 是最大化知识和先前开发重用的卓越手段，因此是开发新架构设计的最佳途径。

- Eg. FACE

例如：FACE

- 美国国防部正在持续推动开放式架构解决方案的应用，以便让更好的航空电子硬件更加快速低成本地进入该领域。

- 未来机载能力环境（FACE）联盟是此类解决方案的提供者之一，成立于 2010 年，目的是开发一套开放架构的技术标准以及用于在系统中实施 FACE 标准的商业模型。

- FACE 技术标准是开放式的航空电子标准，旨在使军事计算系统运作更加稳健、更具互操作性、更加便携以及更为安全。

- 该标准使开发人员可以通过一个通用的操作环境来创建和部署门类广泛的应用系统，以便应用于整个军用航空系统。

- FACE 标准的概念旨在提供一种基于片段的参考架构，这些片段可以组合起来以满足最终系统的需求。各层内容的变化，包括应用代码的变化，使系统设计师可以灵活地设计和构建最终系统。FACE 在这些层之间提供逻辑接口，以实现便携性和重用性。

Architectural Patterns

架构模式

- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.

架构模式是一组适用于重复设计问题的架构设计决策，可以进行参数化以适应该问题出现的不同软件开发环境。

- Architectural patterns are similar to DSSAs but applied "at a lower level" and within a much narrower scope.

架构模式类似于 DSSAs，但应用于“较低层次”和较窄的范围内。

Architectural Patterns

架构模式

- State-Logic-Display: three-tiered pattern

- State-Logic-Display: 三层模式
- Model-View-Controller (MVC)
  - Model-View-Controller (MVC)
- Sense-Compute-Control
  - Sense-Compute-Control

State-Logic-Display: Three-Tiered Pattern
状态-逻辑-显示：三层模式

- Application Examples
  - 业务应用程序
  - 多人游戏
  - 基于 Web 的应用程序
- business apps: the data store is usually a large database server.
  - 业务应用程序：数据存储通常是大型数据库服务器。
- Multi-player games: each player has his own display component.
  - 多人游戏：每个玩家都有自己的显示组件。
- Web-based apps: the display component is the user's Web browser
  - 基于 Web 的应用程序：显示组件是用户的 Web 浏览器

Model-View-Controller (MVC)
模型-视图-控制器（MVC）

- Objective: Separation between information, presentation, and user interaction.
  - 目标：在信息、演示和用户交互之间进行分离。
- When a model object value changes, a notification is sent to the view and to the controller. Thus, the view can update itself and the controller can modify the view if its logic so requires.
  - 当模型对象的值发生变化时，通知将发送到视图和控制器。因此，视图可以更新自己，而控制器如果逻辑需要，可以修改视图。
- When handling input from the user, the windowing system sends the user event to the controller. If a change is required, the controller updates the model object.
  - 处理来自用户的输入时，窗口系统将用户事件发送到控制器。如果需要更改，控制器会更新模型对象。

- Since its invention in the 1980s, the model-view-controller (MVC) pattern has been a dominant influence in the design of graphical user interfaces.
  - 自 20 世纪 80 年代发明以来，模型-视图-控制器（MVC）模式在图形用户界面设计中一直占主导地位。
- Besides as a pattern, it can also be viewed as providing a structure for applications at a higher level.
  - 除了作为一种模式外，它还可以被视为在更高层次为应用程序提供结构。
- MVC promotes separation, and thus independent development paths, between information manipulated by a program and depictions of user interactions with that information.
  - MVC 促进了由程序操纵的信息与用户与该信息互动的描绘之间的分离，从而促进了独立

的开发路径。

The model component encapsulates the information used by the application;
The view component encapsulates the information chosen and necessary for the graphical depiction of that information;
The controller component encapsulates the logic necessary to maintain consistency between the model and the view, and to handle inputs from the user as they relate to the depiction.
模型-视图-控制器

- 在许多情况下，视图和控制器组件合并在一起；这是 Java 的 Swing 框架体系结构的情况。
- 在比编程更高的抽象层次上，MVC 范 paradigm 可以在全球网络中发挥作用。
- Web 资源对应于模型对象；在浏览器内的 HTML 呈现代理对应于查看器；
- 在最简单的情况下，控制器对应于作为浏览器一部分响应用户输入并导致与 Web 服务器的交互或修改浏览器显示的代码。

Model-View-Controller
模型-视图-控制器

Sense-Compute-Control
感知-计算-控制

- Objective: Structuring embedded control applications
  - 目标：构建嵌入式控制应用程序
- Kitchen appliance applications; Automotive applications; Robotic control.
  - 厨房电器应用程序；汽车应用程序；机器人控制。
- such a cycle is keyed to a clock
  - 这样的周期与时钟同步

The Lunar Lander: A Long-Running Example
月球着陆器：一个长期运行的例子

- A simple computer game that first appeared in the 1960’s
  - 一款最早出现在 20 世纪 60 年代的简单电脑游戏
- Simple concept: You (the pilot) control the descent rate of the Apollo-era Lunar Lander
  - 简单的概念：您（飞行员）控制阿波罗时代的月球着陆器的下降速率
- Throttle setting controls descent engine
  - 油门设置控制下降引擎
- Limited fuel
  - 有限燃料
- Initial altitude and speed preset
  - 初始高度和速度预设
- If you land with a descent rate of < 5 fps: you win (whether there’s fuel left or not)
  - 如果您以<5 fps 的下降速率着陆：您赢了（无论燃料是否剩余）
- “Advanced” version: joystick controls attitude & horizontal motion

- “高级”版本：操纵杆控制姿态和水平运动

Sense-Compute-Control LL
感知-计算-控制 LL

Architectural Styles
架构风格

- An architectural style is a named collection of architectural design decisions…
  - 架构风格是一组命名的架构设计决策...
- A primary way of characterizing lessons from experience in software system design
  - 是从软件系统设计经验中描述教训的主要方法
- Reflect less domain specificity than architectural patterns
  - 反映比架构模式更少的领域特定性
- Useful in determining everything from subroutine structure to top-level application structure
  - 有助于确定从子程序结构到顶级应用程序结构的一切
- Many styles exist, and we have discussed them in detail in previous classes
  - 许多风格存在，我们在之前的课程中已经详细讨论过

Definitions of Architectural Style
架构风格的定义

- Definition: An architectural style is a named collection of architectural design decisions that
  - 定义：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
    - 在特定的开发上下文中适用
  - constrain architectural design decisions that are specific to a particular system within that context
    - 约束特定于该上下文中的特定系统的架构设计决策
  - elicit beneficial qualities in each resulting system.
    - 在每个生成的系统中引出有益的特质。
- Recurring organizational patterns & idioms
  - 重复出现的组织模式和习语
- Established, shared understanding of common design forms
  - 共同设计形式的确立的、共享的理解
- Mark of mature engineering field.
  - 成熟工程领域的标志。
- Shaw & Garlan
- Abstraction of recurring composition & interaction characteristics in a set of architectures
  - 在一组架构中抽象了重复出现的组成和交互特性
- Medvidovic & Taylor

Basic Properties of Styles
架构风格的基本属性

- A vocabulary of design elements
  - 设计元素的词汇表
  - Component and connector types; data elements
    - 组件和连接器类型；数据元素
    - e.g., pipes, filters, objects, servers
      - 例如，管道、过滤器、对象、服务器
- A set of configuration rules
  - 一组配置

规则

  - Topological constraints that determine allowed compositions of elements
    - 确定元素允许的组合的拓扑约束
    - e.g., a component may be connected to at most two other components
      - 例如，一个组件最多可以连接到其他两个组件
- A semantic interpretation
  - 语义解释
  - Compositions of design elements have well-defined meanings
    - 设计元素的组合具有明确定义的含义
- Possible analyses of systems built in a style
  - 在风格中构建的系统的可能分析

Benefits of Using Styles
使用风格的好处

- Design reuse
  - 设计重用
  - Well-understood solutions applied to new problems
    - 应用于新问题的易理解的解决方案
- Code reuse
  - 代码重用
  - Shared implementations of invariant aspects of a style
    - 风格不变方面的共享实现
- Understandability of system organization
  - 系统组织的可理解性
  - A phrase such as “client-server” conveys a lot of information
    - 诸如“客户端-服务器”之类的短语传达了大量信息
- Interoperability
  - 互操作性
  - Supported by style standardization
    - 由风格标准化支持
- Style-specific analyses
  - 风格特定的分析
  - Enabled by the constrained design space

    - 通过受限的设计空间启用
- Visualizations
  - 可视化
  - Style-specific depictions matching engineers’ mental models
    - 与工程师的心智模型相匹配的风格特定描绘

Style Analysis Dimensions
风格分析维度

- What is the design vocabulary?
  - 设计词汇是什么？
  - Component and connector types
    - 组件和连接器类型
- What are the allowable structural patterns?
  - 允许的结构模式是什么？
- What is the underlying computational model?
  - 底层计算模型是什么？
- What are the essential invariants of the style?
  - 风格的基本不变性是什么？
- What are common examples of its use?
  - 它的常见用法有哪些？
- What are the (dis)advantages of using the style?
  - 使用该风格的（不）利弊是什么？
- What are the style’s specializations?
  - 风格的专业化有哪些？