软件体系结构 学习笔记

组员：杨伟钦、吕策、李奕辰、方羿阳、徐伟

0 概述

本笔记系统学习了软件体系结构设计。

如文献 [1] 所述，“A critical issue in the design and construction of any complex software system is its architecture: that is, its gross organization as a collection of interacting components. ”，可见，软件体系结构研究的是软件系统的整体结构和各组件之间的联系，而不是具体的组件实现细节。

什么是软件体系结构设计的目标呢？文献 [1] 告诉我们，"A good architecture can help ensure that a system will satisfy key requirements in such areas as performance, reliability, portability, scalability, and interoperability. A bad architecture can be disastrous."。文献 [2] 告诉我们，软件体系结构设计的目的是为了实现软件系统的可修改性、性能、安全性、可靠性、健壮性、易使用性以及商业目标等质量属性。我们将对这些质量属性的需求作简要分析。

软件体系结构设计的方法有很多，文献 [2] 给出了一些常用的软件体系结构设计风格：管道和过滤器、客户端-服务器、对等网络、发布-订阅、信息库、分层以及组合体系结构等。我们将关注于各种软件体系结构风格所适用的范围。

最后，我们还将简要介绍体系结构的评估与改进方法，如文献 [2] 中介绍的一些方法，包括测量设计质量、故障树分析、安全性分析、权衡分析、成本效益分析、原型化等方法。

本笔记在尽可能全面地介绍软件体系结构设计的同时，立足于软件体系结构设计的发展历程，将更为关注于其未来的发展方向。

1 软件体系结构的定义与作用

如文献 [3] 所述，软件体系结构是与组件具体实现（包括算法和数据结构）完全不同的。

> "As the size of software systems increases, the algorithms and data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organization of the overall system—the software architecture—presents a new set of design problems. "

对软件体系结构的准确定义，能够同时指明其重要性和目标。如文献 [1] 中定义的：

> "While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top level design decisions, includeing things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal. "

我们必须要指明组成系统的部件、部件间交互的方式，还需要给出分析和评估的方法。可见，软件体系结构充当着需求和具体代码的桥梁，起到了一种封装的作用。

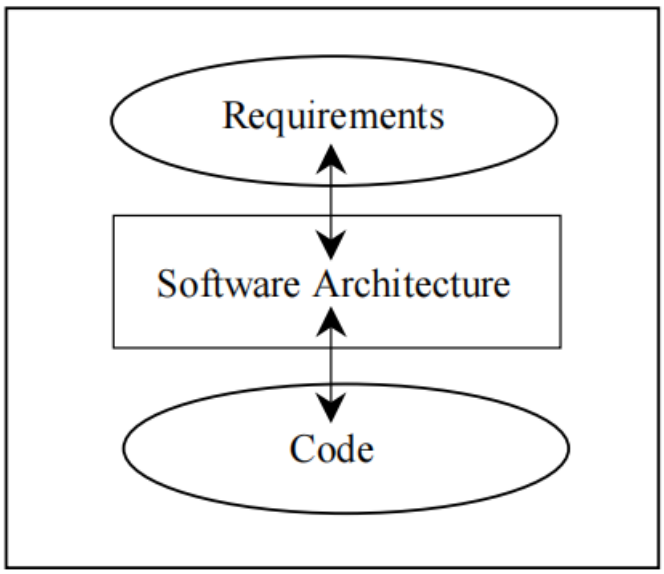

Figure 1: Software Architecture as a Bridge

为什么软件体系结构设计那么重要？我们知道，一件事物是重要的，大概率是因为它发挥了某些关键的作用。文献 [1] 指出，软件体系结构有以下重要的作用，包括软件系统理解与设计、组件重用和集成、组件组成与交互、改进计划与成本评估、更先进的软件分析和管理：

- *Understanding*: Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's high-level design can be easily understood. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices.
- *Reuse*: Architectural descriptions support reuse at multiple levels. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components and also frameworks into which components can be integrated. Existing work on domain-specific software architectures, reference frameworks, and architectural design patterns has already begun to provide evidence for this.
- *Construction*: An architectural description provides a partial blueprint for development by indicating the major components and dependencies between them. For example, a layered view of an architecture typically documents abstraction boundaries between parts of a system's implementation, clearly identifying the major internal system interfaces, and constraining what parts of a system may rely on services provided by other parts.
- *Evolution*: Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications. More over, architectural descriptions separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components, by clearly distinguishing between components and mechanisms that allow them to interact. This separation permits one to more easily change connection mechanisms to handle evolving concerns about performance interoperability, prototyping, and reuse.
- *Analysis*: Architectural descriptions provide new opportunities for analysis, including system consistency checking, conformance to constraints imposed by an architectural style, conformance to quality attributes, dependence analysis, and domain-specific analyses for architectures built in specific styles.
- *Management*: Experience has shown that successful projects view achievement of a viable software architecture as a key milestone in an industrial software development process.

Critical evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies, and potential risks.

文献 [3] 也指出软件体系结构有如下重要作用：

- Inhibiting or Enabling a System's Quality Attributes
- Reasoning About and Managing Change
- Predicting System Qualities
- Enhancing Communication among Stakeholders
- Carrying Early Design Decisions
- Defining Constraints on an Implementation
- Influencing the Organizational Structure
- Enabling Evolutionary Prototyping
- Improving Cost and Schedule Estimates
- Supplying a Transferable，Reusable Model
- Allowing Incorporation of Independently Developed Components
- Restricting the Vocabulary of Design Alternatives
- Providing a Basis for Training

2 软件体系结构设计的目标：质量属性

在概述中，我们提到：软件体系结构设计的目标，就是提高（指定的）质量属性。

首先，我们要搞懂什么是质量属性（Quality Attributes）。如文献 [4] 中定义的：

> "A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders. You can think of a quality attribute as measuring the "goodness" of a product along some dimension of interest to a stakeholder. "。

这一定义表明了质量属性是系统“好”的一种度量。

那么，有哪些比较典型的质量属性呢？我们依据文献 [2] 来抛砖引玉地列举几个：

- 可修改性：在对系统做出一个特定改变的情况下，软件单元可以被分为受直接影响和受间接影响的软件单元。受直接影响的软件单元需要为了适应系统改变而改变自身职责；受间接影响的软件单元只需要修改其实现以适应受直接影响的软件单元的影响，而不需要修改自身职责。对于上述两种受到影响的软件单元，我们都要减少其数量以保证可修改性，做法包括公开接口、降低耦合性等。
- 性能：性能描述了系统的速度和容量，包括响应时间、吞吐量、负载等。提高系统性能要求提高资源利用率、有效管理资源分配、降低对资源的需求等。
- 安全性：大多数的安全性需求阐述了什么是需要被保护的，以及谁不可以访问它。体系结构的安全性有两个方面：高免疫力，指系统能够阻挡攻击企图；高弹性，指系统能够快速容易地从成功的攻击中恢复。体系结构可以通过功能分段、减少安全性弱点等方法提高免疫力。
- 可靠性与健壮性：如果一个软件系统可以在假设的环境下正确地实现所要求的功能，那么我们就说这个软件系统是可靠的；如果在不正确或意外的环境下仍然能够正确地工作，那么它就是健壮的，也即——可靠性与软件本身内部是否有错误有关，健壮性与软件容忍错误或外部环境异常时的表现有关。为了提高可靠性，我们需要实现主动故障检测、异常处理以及故障恢复；为了提高健壮性，我们可以采用“互相怀疑”的设计策略。
- 易使用性：易使用性反应了用户能够操作系统的容易程度。
- 商业目标：商业目标包括开发成本最小化、维护成本最小化、产品上市时间最小化等。

3 软件体系结构设计的方法

在介绍具体的软件体系结构设计方法之前，我们首先给出文献 [1] 中的一段描述：

> "Within industry, two trends highlighted the importance of architecture.
>
> The first was the recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems. For example, the box-and-line-diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a "pipeline,'' a "blackboard-oriented design,'' or a "client-server system.'' Although these terms were rarely assigned precise definitions, they permitted designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provided significant semantic content about the kinds of properties of concern, the expected paths of evolution, the overall computational paradigm, and the relationship be tween this system and other similar systems.
>
> The second trend was the concern with exploiting com monalities in specific domains to provide reusable frame works for product families. Such exploitation is based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by "instantiating'' the shared design. Familiar examples include the standard decomposition of a compiler (which permits undergraduates to construct a new compiler in a semester), standardized communication protocols (which allow vendors to interoperate by providing services at different layers of abstraction), fourth generation languages (which exploit the common patterns of business information processing), and user interface toolkits and frameworks (which provide both a reusable framework for developing interfaces and sets of reusable components, such as menus and dialogue boxes)."

可见，软件体系结构设计方法需要着重于两点：一是对构建复杂软件系统的共同方法、技术、模式和范式的认识；二是注开发特定领域的共性，为产品族提供可重用的框架。因此，我们给出的软件体系结构设计要有足够高的抽象性和复用性，又要有足够的针对性。

我们将基于文献 [2] 和 [3] 简略列举几种常用的软件体系结构：

- Pipes and Filters

> In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. Hence components are termed “filters”. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”.

使用管道和过滤器的典型软件系统是编译器，各个过滤器（如词法分析器、语法分析器等）会接收上一个过滤器的输出，输出直接传送到到下一个过滤器，过滤器间彼此独立，并具有严格的输入输出格式。

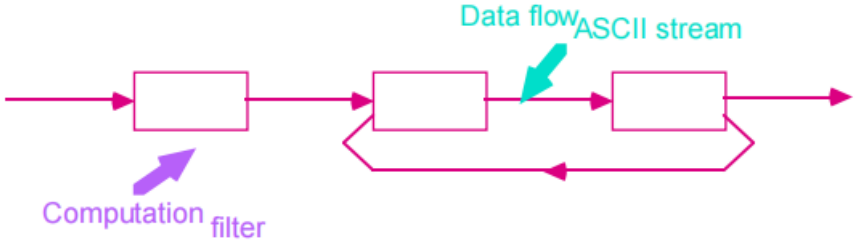

Figure 1: Pipes and Filters

- Data Abstraction and Object-Oriented Organization

> In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects—or, if you will, instances of the abstract data types. Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and (b) that the representation is hidden from other objects.

数据抽象和面向对象是当前软件设计最主流的范式，其对于对象细节的隐藏、对于数据的保护等特性对软件体系结构设计的重要性不言而喻。

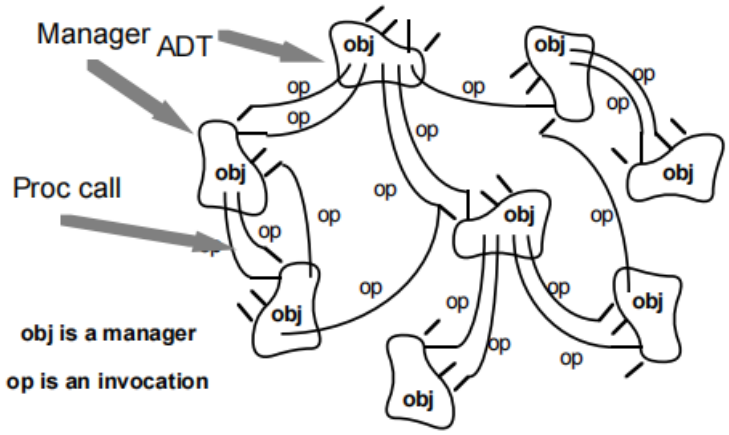

Figure 2: Abstract Data Types and Objects

- Event-based, Implicit Invocation

> The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement ``implicitly'' causes the invocation of procedures in other modules.

很多系统都使用了隐式调用机制，例如在数据库管理系统中用于确保一致性约束，通过语法制导编辑器来支持增量语义检查等。隐式调用的一个重要好处是它为重用提供了强大的支持。只要将任何组件注册为该系统的事件，就可以将其引入到系统中。第二个好处是隐式调用简化了系统进化，在不影响系统其他部件接口的情况下，部件可以被其他部件替换。

- Layered Systems

> A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers.

典型的分层系统包括：计算机网络协议、早年的层次化操作系统等。

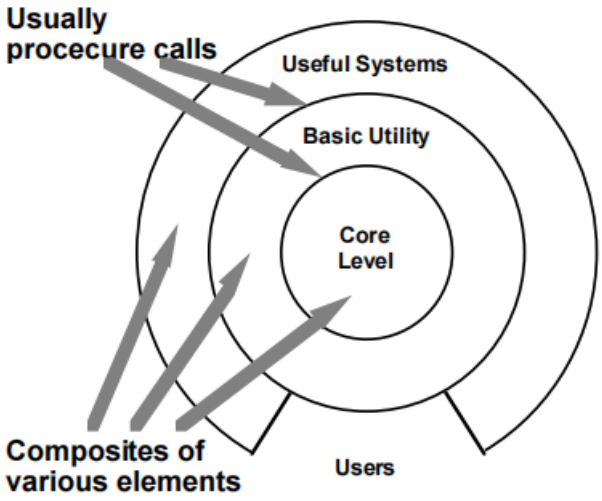

Figure 3: Layered Systems

- Repositories

> In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store.
>
> The knowledge sources: separate, independent parcels of application-dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.
>
> The blackboard data structure: problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
>
> Control: driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

信息库系统基于共享的数据，例如编译器之于符号表等即是典例。

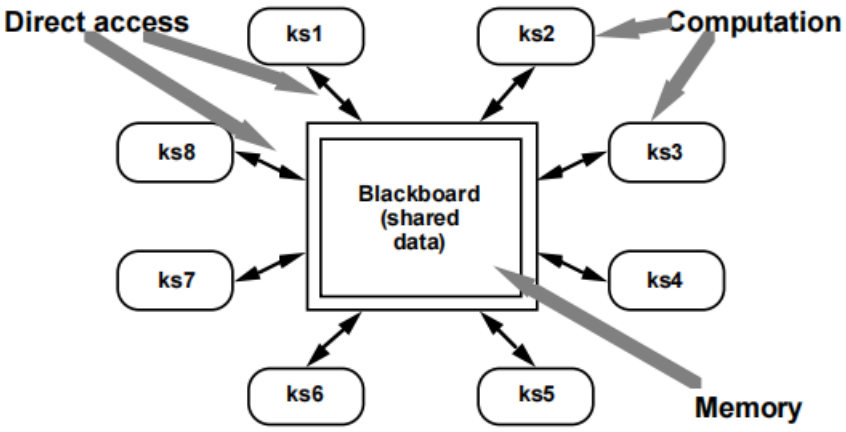

Figure 4: The Blackboard

- Table Driven Interpreters

> In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of its execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo-

> code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated.

显然，解释器架构很适合虚拟机的构建。

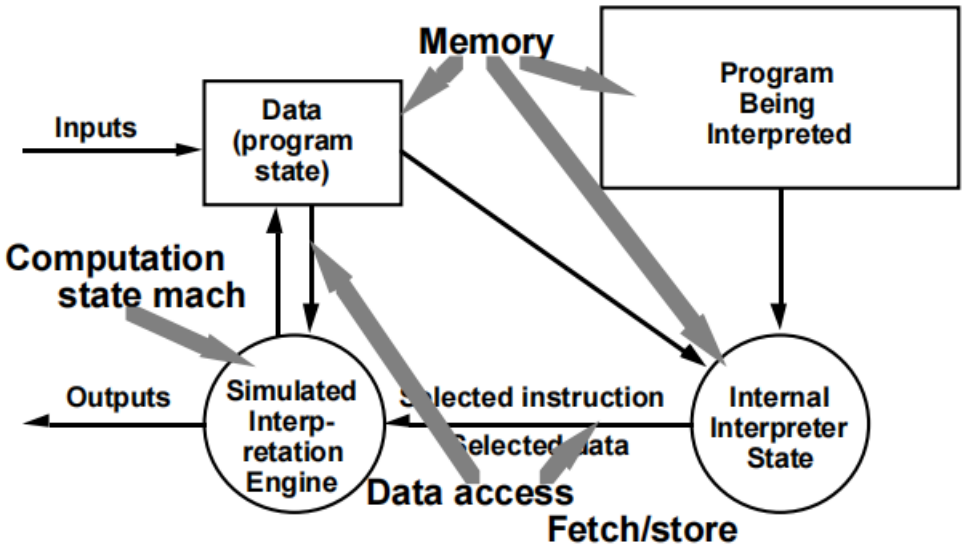

Figure 5: Interpreter

- C/S架构和P2P架构

文献 [3] 中缺少了一些十分重要的体系结构——C/S架构和P2P架构。我们知道，C/S架构（客户-服务器）是基于请求/应答协议的，是最主要的服务器架构；而P2P架构则文件共享网络中广泛应用。两者在计算机网络中有非常广泛且重要的应用。

4 软件系统结构设计的评估与改进

- 故障树

软件系统结构的评估与改进，首要的对故障的分析和处理。一种有利的工具是故障树。

故障树的根节点是我们想分析的失效，而其他节点表示事件或导致根节点失效发生的故障，边是节点间的逻辑关系（表示为逻辑门），表示父节点的事件发生是由子节点的事件的某一发生情况导致的。

一旦建立了故障树，我们就可以查找设计中的弱点。特别地，我们可以将故障树转化为割集树，它解释了哪些事件的组合可以引起失效，以简化对故障树的分析。在割集树中，每个节点表示一个割集，即一组事件的组合。割集树可以从故障树的根节点逻辑门开始，根据逻辑门不断分裂或合并，直到所有子节点都是基本事件为止。

通过故障树/割集树，我们可以找到导致失效的事件组合，然后对这些事件组合进行分析，找到导致失效的原因，从而改进软件系统结构。为此，我们可以改正故障、预防故障、增加检测故障或故障恢复。

- 安全性分析

安全性分析可以被描述为六个步骤：

1. 软件特征化
2. 威胁分析
3. 漏洞评估
4. 风险可能性决策
5. 风险影响决策
6. 风险缓解计划

除此以外，软件系统结构设计的评估与改进还包括：

- 权衡分析：根据质量属性比较多个设计候选项，利用比较表等工具。
- 成本效益分析：根据经济效益和经济成本来综合分析系统的成本效益，一般包括：计算效益，即改进某个质量属性所带来的增值；计算投资回报，即ROI = 效益/成本；计算投资回收期等。

5 软件系统结构设计的今天

文献 [1] 中准确地描述了现代软件系统结构设计的发展表现在三个方面：

> In addition, the technological basis for architectural design has improved dramatically. Three of the important advancements have been the development of architecture description languages and tools, the emergence of product line engineering and architectural standards, and the codification and dissemination of architectural design expertise.

- the development of architecture description languages and tools

体系结构描述语言（ADLs）为描述软件体系结构提供了概念框架和具体语法，例如Adage, Aesop, C2, Darwin, Rapide, SADL, UniCon, Meta-H, and Wright.

> Languages explicitly designed with software architecture in mind are not the only approach. There has been considerable interest in using general-purpose object design notations for architectural modeling [8, 24]. Moreover, recently there have been a number of proposals that attempt to show how the concepts found in ADLs can be mapped directly into an object-oriented notation like UML.

现在也有很多工作，将ADL中的软件结构直接映射到UML这样的更一般的面向对象建模语言中。

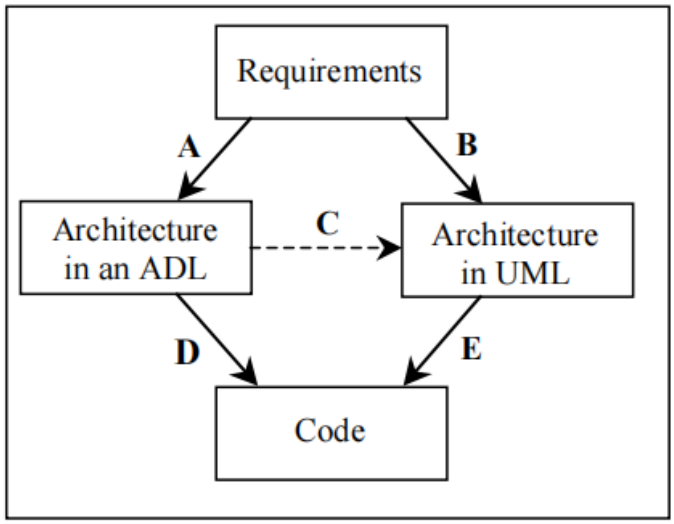

Figure 2: Software Architecture as a Bridge

- the emergence of product line engineering

> As noted earlier, one of the important trends has been the desire to exploit commonality across multiple products. Two specific manifestations of that trend are improvements in our ability to create product lines within an organization and the emergence of cross-vendor integration standards.

产品线方法必须要考虑到系统族的需求，还需要考虑到这些需求和与每个特定实例相关的需求之间的联系，开发出可重用的架构。一些典型的例子包括High Level Architecture (HLA) for Distributed Simulation, Sun's Enterprise Java Beans (EJB) architecture。

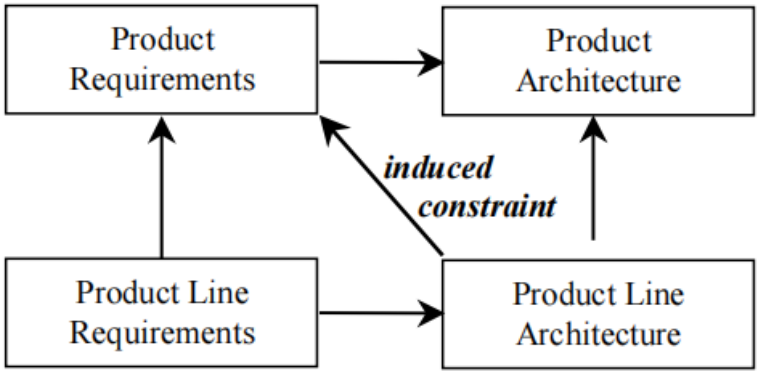

Figure 3: Product Line Architectures

- the codification and dissemination of architectural design expertise

一些关于软件体系结构设计的书籍、 课程、标准、文档的出版，使得软件体系结构的基本范式得以广泛传播快速发展。

6 软件系统结构设计的明天

文献 [1] 中介绍了软件系统结构设计的三个未来方向：

- Changing Build-Versus Buy Balance

> Throughout the history of software engineering a critical issue in the development of systems has been the decision about what parts of the system to obtain elsewhere and what parts to build in-house.
>
> While the issue itself has not changed, economic pressures to reduce time-to-market are drastically changing the balance. For an increasing number of products, introducing a product a month early may be the difference between success and failure. In such situations building systems using software that others have written becomes the only feasible choice.
>
> First, this trend heightens the need for industry-wide standards. Today's component technologies work at a fairly low level of architectural abstraction – essentially at the level of procedure call between objects. To obtain more significant integration will require higher-level architectural standards. This is likely to lead us from "component-based" engineering to "architecture-based" engineering, a shift that will emphasize the role of domain-specific integration architectures (such as EJB or HLA, mentioned earlier) in promoting composability. Thus, the trend toward architectural standards, noted earlier, is likely to become even more pronounced.
>
> Second, this trend is leading to new software subcontracting processes. When integration is the critical issue, we can expect that externally contracted software will be held to much higher standards of architectural conformance. In many cases the standards will be commercial or governmental standards. In others, it is likely that sub-contractors will need to guarantee compatibility with product line architectures.
>
> Third, this trend is leading toward standardization of notations and tools across vendors. When down-stream integrators are recombining systems, it is no longer acceptable to have in-house, idiosyncratic architecture descriptions and tools. This has led many to adopt languages like UML and XML for architectural modeling.

可见，软件之间的互相集成已经成为了未来软件开发的发展趋势，这一趋势强调了对软件体系结构的行业标准的需求，包括软件的体系结构一致性标准、供应商之间的符号和工具的标准化等等。

- Network-Centric Computing

> Historically, software systems have been developed as closed systems, developed and operated under control of individual institutions. The constituent components may be acquired from external sources, but when incorporated in a system they come under control of the system designer. Architectures for such systems are either completely static – allowing no run time restructuring – or permit only a limited form of run time variation.
>
> However, within the world of pervasive services available over networks, systems may not have such centralized control. The Internet is an example of such an open system. For such systems a new set of software architecture challenges emerges [41]. First, is the need for architectures that scale up to the size and variability of the Internet. While many of the same architectural paradigms will likely apply, the details of their implementation and specification will need to change.
>
> For example, one attractive form of composition is implicit invocation – sometimes termed ''publish-subscribe.'' Within this architectural style components are largely autonomous, interacting with other components by broadcasting messages that may be "listened to" by other components. Most systems that use this style, however, make many assumptions about properties of its use. For example, one typically assumes that event delivery is reliable, that centralized routing of messages will be sufficient, and that it makes sense to define a common vocabulary of events that are understood by all of the components. In an Internet-based setting all of these assumptions are questionable.
>
> Second, is the need to support computing with dynamically-formed, task-specific, coalitions of distributed autonomous resources. The Internet hosts a wide variety of resources: primary information, communication mechanisms, applications that can be invoked, control that coordinates the use of resources, and services such as secondary (processed) information, simulation, editorial selection, or evaluation. These resources are independently developed and independently supported; they may even be transient. They can be composed to carry out specific tasks set by a user; in many cases the resources need not be specifically aware of the way they are being used, or even whether they are being used. Such coalitions lack direct control over the incorporated resources. Selection and composition of resources is likely to be done afresh for each task, as resources appear, change, and disappear. Unfortunately, it is hard to automate the selection and composition activity because of poor information about the character of services and hence with establishing correctness. Composition of components in this setting is difficult because it is hard to determine what assumptions each component makes about its operating context, let alone whether a set of components will interoperate well (or at all) and whether their combined functionality is what you need. Moreover, many useful resources exist but cannot be smoothly integrated because they make incompatible assumptions about component interaction. For example, it is hard to integrate a component packaged to interact via remote procedure calls with a component packaged to interact via shared data in a proprietary representation. These problems will provide new challenges for architecture description and analysis. In particular, the need to handle dynamically-evolving collections of components obtained from a variety of sources will require new techniques for managing architectural models at run time, and for evaluating the properties of those ensembles.
>
> Third, is the related need to find architectures that flexibly accommodate commercial application service providers. In the future, applications will be composed out of a mix of local and remote computing capabilities on each desktop, requiring architectural support that supports billing, security, etc.

> Fourth, is the need to develop architectures that permit end users to do their own system composition. With the rapid growth of the Internet, an increasing number of users are in a position to assemble and tailor services. Such users may have minimal technical expertise, and yet will still want strong guarantees that the parts will work together in the ways they expect.

可见，在以网络为中心的软件系统中，软件系统并没有集中控制的方式，许多软件对资源或事件的假设都需要进行修改，对于体系结构的分析与评估需要重新进行。未来，需要构建能够灵活适应商业应用服务提供商的架构，包括远程通信、安全支持、分布式和云端计算等。此外，还可能需要开发允许用户定制自己的系统组合的体系结构。

- Pervasive Computing

> A third related trend is toward pervasive computing in which the computing universe is populated by a rich variety of heterogeneous computing devices: toasters, home heating systems, entertainment systems, smart cars, etc. This trend is likely to lead to an explosion in the number of devices in our local environments – from dozens of devices to hundreds or thousands of devices. Moreover these devices are likely to be quite heterogeneous, requiring special considerations in terms of their physical resources and computing power. There are a number of consequent challenges for software architectures.
>
> First, we will need architectures that are suited to systems in which resource usage is a critical issue. For example, it may be desirable to have an architecture that allows us to modify the fidelity of computation based on the power reserves at its disposal.
>
> Second, architectures for these systems will have to be more flexible than they are today. In particular, devices are likely to come and go in an unpredictable fashion. Handling reconfiguration dynamically, while guaranteeing uninterrupted processing, is a hard problem.
>
> Third, is the need for architectures that will better handle user mobility. Currently, our computing environments are configured manually and managed explicitly by the user. While this may be appropriate for environments in which we have only a few, relatively static, computing devices (such as a couple of PCs and laptops), it does not scale to environments with hundreds of devices. We must therefore find architectures that provide much more automated control over the management of computational services, and that permit users to move easily from one environment to another.

在普适计算阶段，我们需要让系统适配多种不同的计算设备，这需要合适的资源评估、分配和使用。因此，未来需要一种更加灵活的系统架构，允许不可预测的设备出现、移动或消失，实现对计算服务的自动化控制与迁移。

参考文献

[1] Garlan, David. "Software architecture: a roadmap." *Proceedings of the Conference on the Future of Software Engineering*. 2000.

[2] Pfleeger, Shari Lawrence, and Joanne M. Atlee. *Software engineering: theory and practice*. Pearson Education India, 2010.

[3] Garlan, David, and Mary Shaw. "An introduction to software architecture." *Advances in software engineering and knowledge engineering*. 1993. 1-39.

[4] Bass, Len, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.