南京大学
NANJING UNIVERSITY

《软件工程与计算II》

Ch9 软件体系结构基础

大纲

- 软件体系结构的发展
- 理解软件体系结构
- 体系结构风格初步

软件体系结构必要

- 小规模编程
  - 模块内部的程序结构非常依赖程序设计语言提供的编程机制
- 大规模编程
  - 将众多模块组织起来实现需求，需求特别关注模块之间的关系，不依赖于程序设计语言的另外一项技术

First reference to the phrase software architecture 1969

Software Architecture in 1969

Ian P. Sharp made these comments at the 1969 NATO Conference on Software Engineering Techniques. They still resonate well 37 years later.

I think that we have something in addition to software engineering: something that we have talked about in small ways but which should be brought out into the open and have attention focused on it. This is the subject of software architecture. Architecture is different from engineering.

As an example of what I mean, take a look at OS/360. Parts of OS/360 are extremely well coded. Parts of OS, if you go into it in detail, have used all the techniques and all the ideas which we have agreed are good programming practice. The reason that OS is an amorphous lump of program is that it had no architect. Its design was delegated to a series of groups of engineers, each of whom had to invent their own architecture. And when these lumps were nailed together they did not produce a smooth and beautiful piece of software.

I believe that a lot of what we construe as being theory and practice is in fact architecture and engineering; you can have theoretical or practical architects: you can have theoretical or practical engineers. I don't believe, for instance, that the majority of what Dijkstra does is theory—I believe that in time we will probably refer to the "Dijkstra School of Architecture."

What happens is that specifications of software are regarded as functional specifications. We only talk about what it is we want the program to do. It is my belief that anybody who is responsible for the implementation of a piece of software must specify more than this. He must specify the design, the form; and within that framework programmers or engineers must create something. No engineer or programmer, no programming tools, are going to help us, or help the software business, to make up for a lousy design. Control, management, education and all the other goodies that we have talked about are important; but the implementation people must understand what the architect had in mind.

Probably a lot of people have experience of seeing good software, an individual piece of software which is good. And if you examine why it is good, you will probably find that the designer, who may or may not have been the implementer as well, fully understood what he wanted to do and he created the shape. Some of the people who can create shape can't implement and the reverse is equally true. The trouble is that in industry, particularly in the large manufacturing empires, little or no regard is being paid to architecture. —Software Engineering Techniques: Report of a Conference Sponsored by the NATO Science Committee, *B. Randell and J.N. Buxton, eds., Scientific Affairs Division, NATO, 1970, p. 12.*

.

Some of our field’ s most prestigious pioneers, including:
Tony Hoare,
Edsger Dijkstra,
Alan Perlis,
Per Brinch Hansen, Friedrich Bauer,
and Niklaus Wirth,
attended this meeting

Until the late 1980

- From then until the late 1980s, the word “architecture” was used mostly in the sense of system architecture (meaning a computer system’s physical structure) or sometimes in the narrower sense of a given family of computers’ instruction set.
- Key sources about a soft- ware system’s organization came from Fred Brooks in 1975, Butler Lampson in 1983, David Parnas from 1972 to 1986, and John Mills in 1985 (whose article looked more into the process and pragmatics of architecting).

1992

- In 1992, Dewayne Perry and Alexander Wolf published their seminal article "Foundations for the Study of Software Architecture."
- This article introduced the famous formula "{elements, forms, rationale} = software architecture," to which Barry Boehm added "constraints" shortly thereafter.
- For many researchers, the "elements" in the formula were components and connectors.These were the basis for a flurry of architecture description languages (ADLs), including C2, Rapide, Darwin,Wright, ACME, and Unicon, which unfortunately haven't yet taken much root in industry.

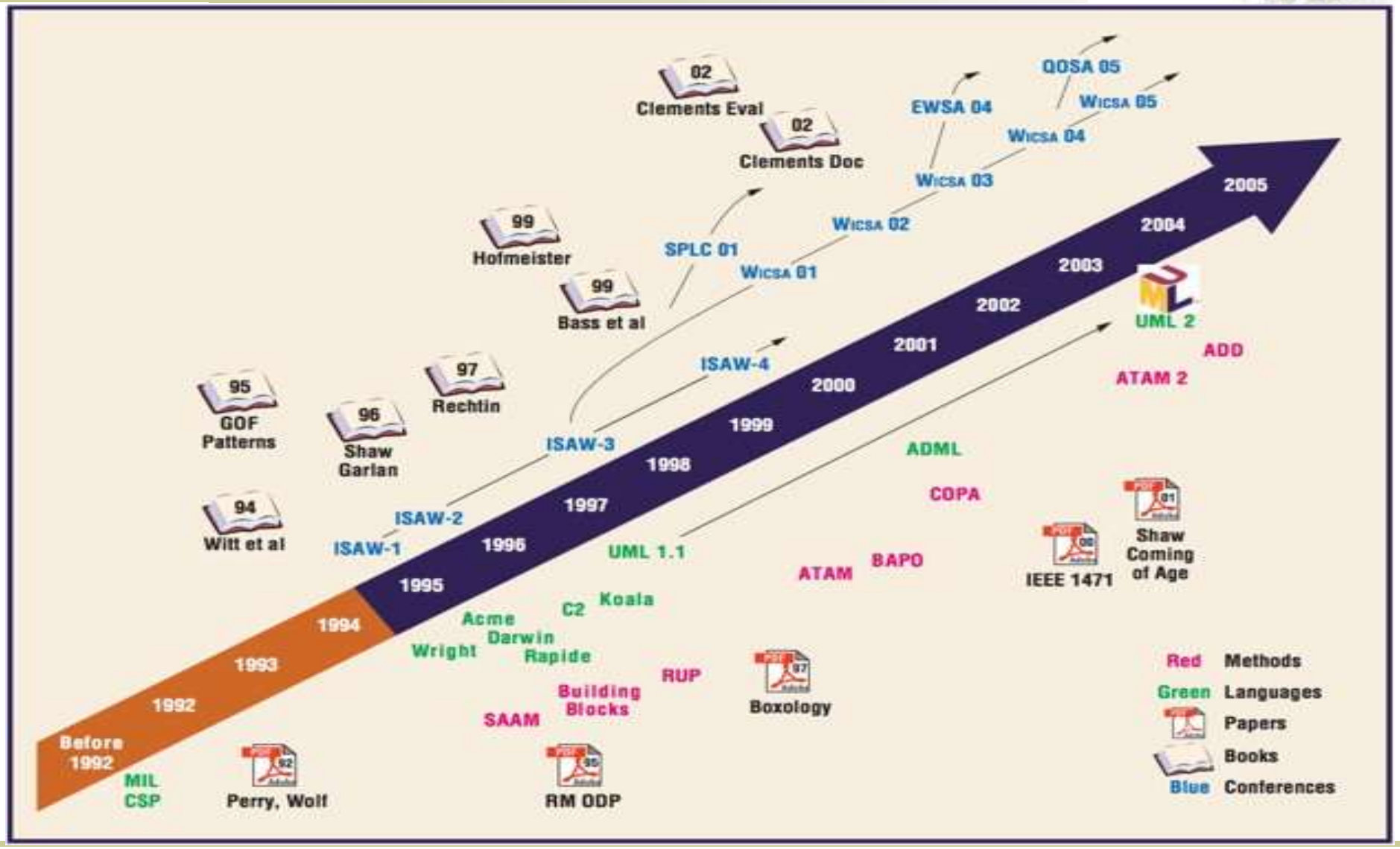
Importance

- “Architecture is the linchpin for the highly complex, massively large-scale, and highly interoperable systems that we need now and in the future"
- — Rolf Siegers, Raytheon
- “Fundamentally, there are three reasons:
  - Mutual communication
  - Early design decisions
  - Transferable abstraction of a system”

  — [Clements1996]

软件体系结构的十年

-- Philippe Kruchten



IEEE 1471-2000

- IEEE Recommended Practice for Architectural Description of Software- Intensive Systems

Books

Starting Your Software Architecture Library

We recommend the following 12 books to any budding software architect. They cover a vast range of issues and provide the necessary foundation for further study, research, and application.

The first book

- M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996. This book put software architecture firmly on the world map as a discipline distinct from software design or programming, and it's still a worthwhile read. The authors tried to define what software architecture is—a difficult task. We still haven't reached consensus 10 years later. Much of the book is dedicated to the concept of architectural styles, and there's a useful chapter on educating software architects.

The SEI trilogy

- L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003. Originally published in 1998, this book expanded many aspects of software architecture: process and method, representation, techniques, tools, and business implications. It provides a good introduction to several SEI architectural methods.
- P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002. Focused solely on software architecture documentation and representation, this book constitutes a de facto application guide to the rather abstract *IEEE Standard 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems*.
- P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architecture*, Addison-Wesley, 2002. How good is this architecture? The third book in the SEI trilogy (this productive group actually wrote more than three) focuses on reviewing and evaluating various aspects of "goodness" and qualities of an architecture, existing or to be built. A good complement to the *Software Architecture Review and Assessment (SARA) Report* (SARA Working Group, 2002).

Ammunition for architects

- C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999. The authors offer a systematic, detailed architectural-design method and a representation of software architecture based on their work at the Siemens Research Center.
- I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997. As the title indicates, this book bridges the software reuse community (which was thriving but running a bit out of air in the mid-1990s) with the architecture community, showing how the two can leverage each other. It presents elements of the architectural method the Rational Unified Process embodies. If reuse and product lines are important to you, we'll suggest more reading later.
- F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996. Branching off from the design pattern work of the Gang of Four, this "Gang of Five" assembled a useful catalog of architectural-design patterns. Unfortunately, they haven't continued what they had started so well.

Pragmatics

- R.C. Malveau and T.J. Mowbray, *Software Architect Bootcamp*, 2nd ed., Prentice Hall, 2000. This "how to get started" guide is directed at practitioners.
- D.M. Dikel, D. Kane, and J.R. Wilson, *Software Architecture: Organizational Principles and Patterns*, Prentice Hall, 2001. The authors have captured the dynamics of what happens in a software architecture team—the constraints, tensions, and dilemmas—in their VRAPS (vision, rhythm, anticipation, partnering, and simplification) model.
- E. Rechtin and M. Maier, *The Art of Systems Architecting*, CRC Books, 1997. Rechtin's initial book in 1991 was about systems and very little about software, although software architects could transpose and interpret many of the principles presented. By teaming up with Mark Maier, Rechtin was able to cover software aspects more specifically and deeply. However, beginners will find it hard to read, so they should start with the first two books in this group.

Software product lines

- J. Bosch, *Design and Use of Software Architecture: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000. This book and the next represent the branching off of software architecture into its application for software product lines.
- M. Jazayeri, A. Ran, F. van der Linden, and P. van der Linden, *Software Architecture for Product Families: Principles and Practice*, Addison-Wesley, 2000.

Papers

Great Papers on Software Architecture

If you're not a great fan of books (see the "Starting Your Software Architecture Library" sidebar), you can get a quick introduction to many of the underlying concepts from this collection of key papers.

Foundations

- M. Shaw and D. Garlan, "An Introduction to Software Architecture," V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, vol. 2, World Scientific Publishing, 1993, pp. 1–39. Shortly preceding their book, this paper brought together what we knew about software architecture in the beginning of the 1990s.
- D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM Software Eng. Notes*, vol. 17, no. 4, 1992, pp. 40–52. This seminal paper will be always remembered for giving us this simple but insightful formula: {elements, form, rationale} = software architecture.

Precursors

- D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 12, 1972, pp. 1053–1058. Software architecture didn't pop up out of the blue in the early 1990s. Although David Parnas didn't use the term "architecture," many of the underlying concepts and ideas owe much to his work. This article and the next two are the most relevant in this regard.
- D.L. Parnas, "On the Design and Development of Program Families," *IEEE Trans. Software Eng.*, vol. 2, no. 1, 1976, pp. 1–9.
- D.L. Parnas, P. Clements, and D.M. Weiss, "The Modular Structure of Complex Systems," *IEEE Trans. Software Eng.*, vol. 11, no. 3, 1985, pp. 259–266.
- F. DeRemer and H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," *Proc. Int'l Conf. Reliable Software*, ACM Press, 1975, pp. 114–121. Their Module Interconnection Language (MIL 75) is in effect the ancestor of all ADLs, and its design objectives are still valid today. The authors had a clear view of architecture as distinct from design and programming at the module level but also at the fuzzy, abstract, "high-level design" level.

Architectural views

- D. Soni, R. Nord, and C. Hofmeister, "Software Architecture in Industrial Applications," *Proc. 17th Int'l Conf. Software Eng.* (ICSE 95), ACM Press, 1995, pp. 196–207. This article introduced Siemens' five-view model, which the authors detailed in their 1999 book *Applied Software Architecture* (see the "Architecture Library" sidebar).
- P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 45–50. Part of the Rational Approach—now known as the Rational Unified Process—this set of views was used by many Rational consultants on large industrial projects. Its roots are in the work done at Alcatel and Philips in the late 1980s.

Process and pragmatics

- B.W. Lampson, "Hints for Computer System Design," *Operating Systems Rev.*, vol. 15, no. 5, 1983, pp. 33–48; reprinted in *IEEE Software*, vol. 1, no. 1, 1984, pp. 11–28. This article and the next gave one of us (Kruchten), a budding software architect in the 1980s, great inspiration. They haven't aged and are still relevant.
- J.A. Mills, "A Pragmatic View of the System Architect," *Comm. ACM*, vol. 28, no. 7, 1985, pp. 708–717.
- W.E. Royce and W. Royce, "Software Architecture: Integrating Process and Technology," *TRW Quest*, vol. 14, no. 1, 1991, pp. 2–15. This article articulates the connection between architecture and process very well—in particular, the need for an iterative process in which early iterations build and validate an architecture.

Two more for the road

Where do we stop? We're tempted to add many more articles on such ADLs as Rapide, Wright, and C2 as well as on model-driven architecture. We'll just add two more.

- M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proc. 21st Int'l Computer Software and Applications Conf.* (COMPSAC 97), IEEE CS Press, 1997, pp. 6–13.
- M. Shaw, "The Coming-of-Age of Software Architecture Research," *Proc. 23rd Int'l Conf. Software Eng.* (ICSE 01), IEEE CS Press, 2001, pp. 656–664a.

Community

A Software Architecture Community

Here are a dozen places to go for more information on software architecture, to participate in or attend conferences, or to join a group of peers.

Resources

- The Software Engineering Institute's Software Architecture for Software-Intensive Systems Web site (www.sei.cmu.edu/architecture) contains many definitions, papers on their methods, and further pointers. The software architecture practice group at the Software Engineering Institute maintains this portal.
- The Gaudi System Architecting Web page (www.gaudisite.nl), named after the famous Spanish architect, deals with system architecture. Gerrit Muller from Philips Research and the Embedded Systems Institute in Eindhoven maintains the page.
- The Bredemeyer architecture portal (www.bredemeyer.com), called "Software Architecture, Architects and Architecting," is maintained by Dana Bredemeyer and Ruth Malan. It contains not only their own writings but also a well-organized collection of other resources and announcements.
- The Software Product Lines page (http://softwareproductlines.com) focuses on product lines and large-scale reuse.
- SoftwareArchitectures.com (http://softwarearchitectures.com) is another portal to architecture resources.
- Grady Booch, from IBM, is spearheading an effort to establish a handbook for software architects and is building a repository of example architectures and case studies (www.booch.com/architecture/index.jsp).

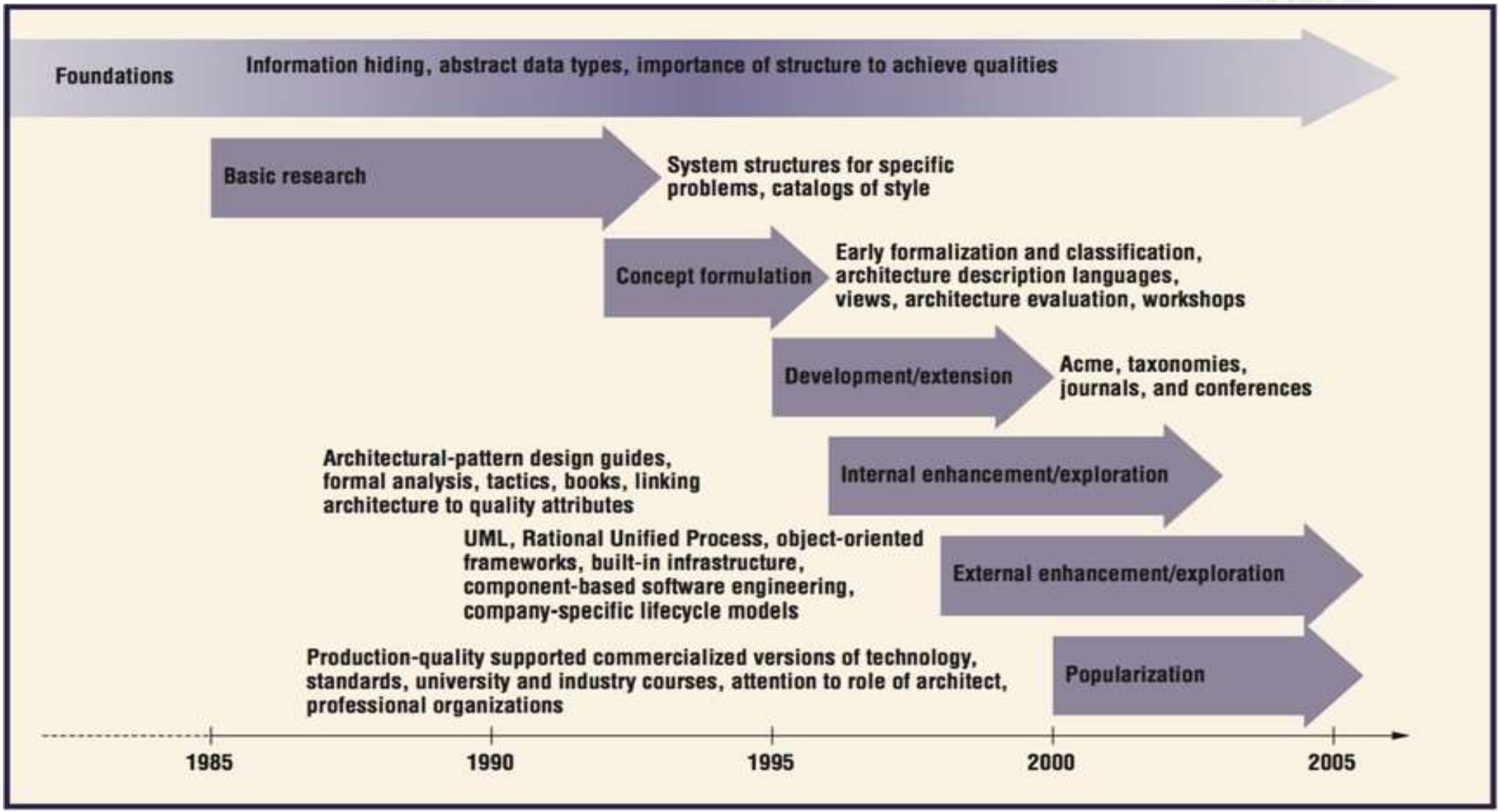
Conferences

- Working IEEE/IFIP Conferences on Software Architecture (www.softwarearchitectureportal.org/WICSA/conferences). Since 1999, WICSA has attracted a lot of contributions from industry and academia and many interesting debates and advances. Its location alternates between North America and another part of the world (only Europe so far). WICSA subsumed the International Software Architecture Workshop series which ran from 1995 to 2000.
- European Workshop on Software Architecture (www.arch-ware.org/ewsa). Begun in 2004, EWSA is mainly driven by the participants in the European project ArchWare—Architecting Evolvable Software.
- Software Product Line Conference (http://softwareproductlines.com). Since 2000, this subcommunity of software architecture has organized a successful meeting series. SPLC subsumed the older PFE (Product Family Engineering) conference series in Europe. Start from this site to access the most recent SPLC.
- The Conference on the Quality of Software Architectures, or QOSA (http://se.informatik.uni-oldenburg.de/qosa) was a new conference in 2005.
- Software architecture is also present—often in the form of a specific session or a distinct track—in other conferences, including ICSE; ECOOP (European Conference on Object-Oriented Programming); OOPSLA (Object-Oriented Programming Systems, Languages, and Applications); FSE (Foundation of Software Engineering); APSEC (Asia-Pacific Software Engineering Conference); and now MODELS (ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems), which subsumed the UML conference series.

Associations and working groups

- IFIP WG 2.10 Software Architecture (www.ifip.org/bulletin/bulltcs/memtc02.htm#wg210). Founded at the first WICSA conference in 1999, the 13 or so members of the IFIP's Working Group 2.10 meet face to face twice a year and a few more times by telephone and the Internet. They are the driving force behind the WICSA conference series and this special issue of *IEEE Software*. They also maintain the www.softwarearchitectureportal.org portal.
- The Worldwide Institute of Software Architects, or WWISA (www.wwisa.org) was founded by Mark and Laura Sewell in 1999.
- The International Association of Software Architects, or IASA (www.iasarchitects.org) is an association of IT architects focusing on social networking, advocacy, ethics, and knowledge sharing.
- IEEE Standards Association WG 1471 (http://standards.ieee.org). The working group that created *IEEE Std 1471-2000* is now resurrecting itself to tackle a revision of the standard.
- SARA—Software Architecture Review and Assessment. This informal group of architects from industry (Philips, Siemens, Rational, Nokia, IBM, and Lockheed Martin) met regularly from 1998 to 2001 to share software evaluation practices. They produced a report in 2001 (www.philippe.kruchten.com/architecture/SARAv1.pdf).
- Research and education. Many academics and researchers maintain pages with pointers to their research and other resources. We picked just two examples: Nenad Medvidovic, University of Southern California, http://sunset.usc.edu/research/software_architecture/index.html; and Gert Florijn, Software Engineering Research Center in the Netherlands, www.serc.nl/people/florijn/interests/arch.html.

The golden age of software architecture -- Mary Shaw



大纲

- 软件体系结构的发展
- 理解软件体系结构
  - 概念和定义
  - 区分物理与逻辑
  - 高层抽象
- 体系结构风格初步

What is software architecture?

Community Software Architecture Definitions

http://www.sei.cmu.edu/architecture/start/glossary/community.cfm

Kruchten

- Software architecture involves:
  - the structure and organization by which modern system components and subsystems interact to form systems, and
  - the properties of systems that can best be designed and analyzed at the system level.

Kruchten

Philippe Kruchten (Director of Process Development, Rational Software Corporation, Vancouver, B.C.): Software Architecture encompasses the significant decisions about

- the organization of a software system,
- the selection of the structural elements and their interfaces by which the system is composed together with - their behavior as specified in the collaboration among those elements,
- the composition of these elements into progressively larger subsystems,
- the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition.

- Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technological constraints and tradeoffs, and aesthetics. (With G. Booch and R. Reitman)
- Citation: Rational Unified Process 5.0, Rational, Cupertino, 1998; PBK, The Rational Unified Process--An Introduction, Addison-Wesley-Longman (1999).

Shaw

[Shaw1995]将软件体 系结构模型定义为:

- 软件体系结构={部件(Component),连接件(Connector),配置(Configuration)}
  - “部件”是软件体系结构的基本组成单位之一,承载系统的主要功能,包括处理与数据;
  - “连接件”是软件体系结构的另一个基本组成单位,定义了部件间的交互,是连接的抽象表示;
  - “配置”是对“形式”的发展,定义了“部件”以及“连接件”之间的关联方式,将它们组织成系统的总体结构。

按照这个模型,[Shaw1996]给出了一个简洁的软件体系结构定义:

- “一个软件系统的体系结构规定了系统的计算部件和部件之间的交互.”

Perry

- Software Architecture = { Elements, Form, Rationale }
  [Perry1992]
- what how why
- Process+Data+Connection

Bass

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." [Bass1998]

- Every software has one or more architecture structures

Wiki

Software architecture is the realization of non-functional requirements, while software design is the realization of functional requirements.

What is software architecture?

- High level abstraction
- Concern of stakeholders
- Design decision

大纲

- 软件体系结构的发展
- 理解软件体系结构
  - 概念和定义
  - 区分物理与逻辑
  - 高层抽象
- 体系结构风格初步

Logical vs Physical

- High-level vs low-level
- Abstraction vs Implementation

- Logical implies a higher view than the physical.

Example

- Logically From Phoenix to Boston A message transmitted from Phoenix to Boston logically goes from one city to the other;

- However, the physical telephone circuit could be Phoenix to Chicago to Philadelphia to Boston. Over the Internet, the message could traverse switching points in many other locations.

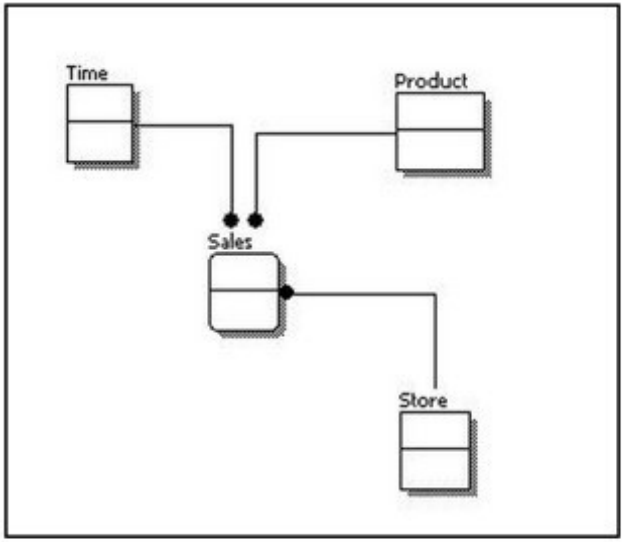
Example

- Logical Drive C: - Physical Drive 0 In a Windows PC, a single physical hard drive is drive 0; however, it may be partitioned into several logical drives, such as C:, D: and E:.
- Virtualization Various virtualization methods create a logical "abstraction layer" for dealing with the physical hardware. For example, virtual machines enable multiple operating systems to run in the computer, each accessing the hardware via a logic layer rather than direct physical contact.

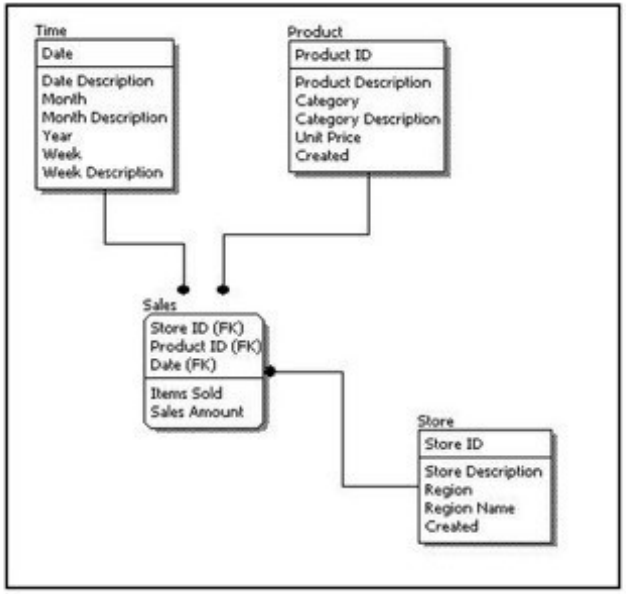
Example

- Users relate to data logically by data element name; however, the actual fields of data are physically located in sectors on a disk.
- To find out which customers ordered how many of a particular product, the logical view is customer name and quantity. Its physical organization might have customer name in a customer file and quantity in an order file cross referenced by customer number.The physical sequence of the customer file could be indexed, while the sequence of the order file could be sequential.

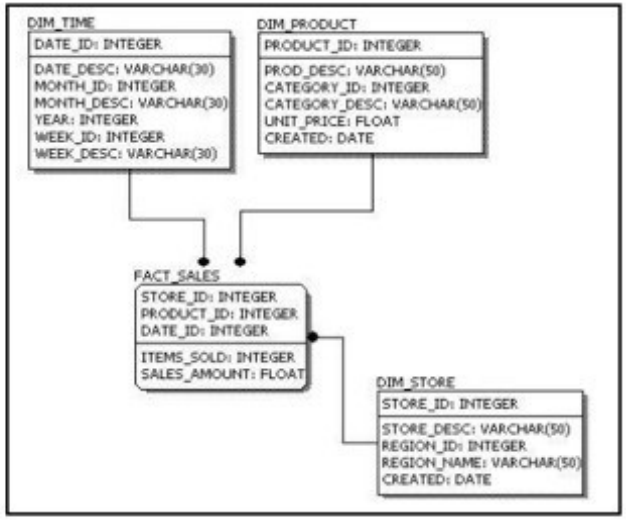
Conceptual Model Design



Logical Model Design



Physical Model Design



Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

Conceptual vs Logical vs Physical

模块

逻辑：一个模块调用另一个模块

- 物理实现
  - 基本：接口调用
  - 需要传递数据对象怎么办？

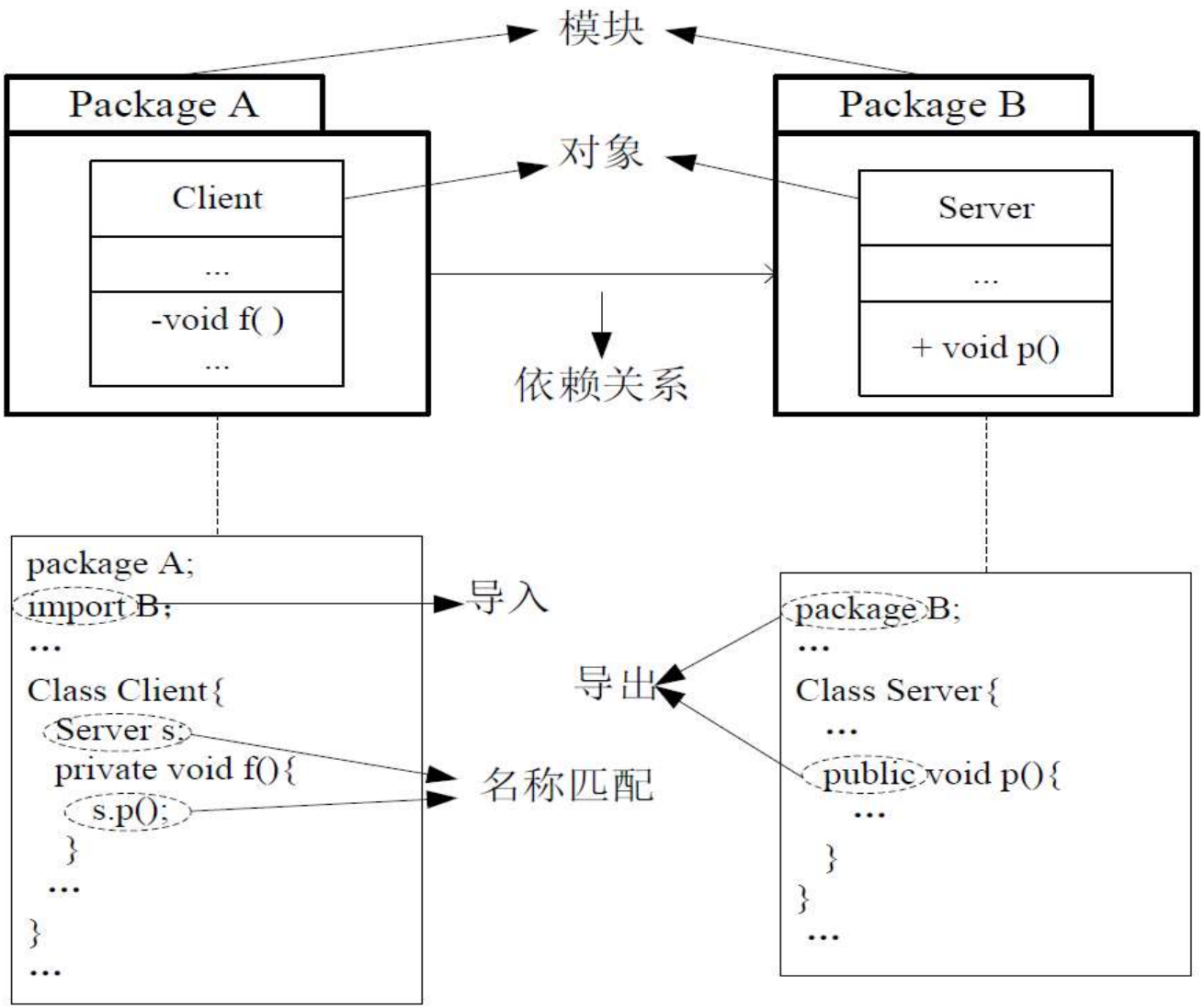
逻辑：一个模块给另一个模块传递数据流

- 物理实现
  - 读写共享数据、pipe…

物理实现的载体

- 低层：基本类型+基本控制结构
- 中层：OO编程语言机制
  - 类声明、实例创建与撤销、实例生命期管理
  - 类权限控制机制
  - 复杂机制：继承…
- 高层
  - 导入导出和名称匹配

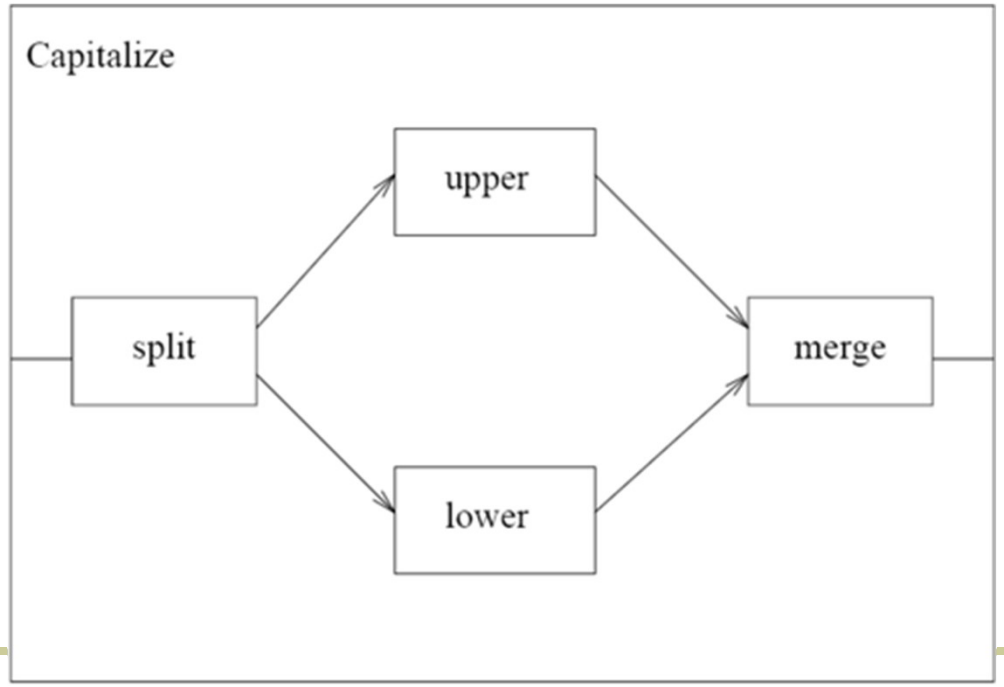
导入导出机制

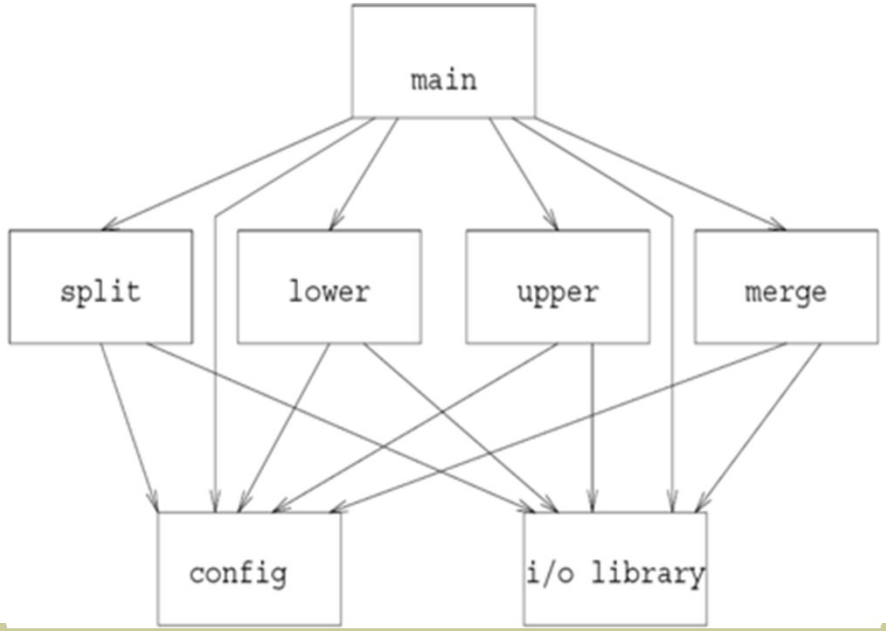

模块
Package A
Client
...
-void f( )
...
对象
依赖关系
Package B
Server
...
+ void p()
package A;
import B;
...
Class Client{
Server s;
private void f(){
s.p();
}
...
}
...
导入
导出
名称匹配
package B;
...
Class Server{
...
public void p(){
...
}
}
...

抽象vs实现

- Architecture as-design
- Functional organizations

- Architecture as-implementation
- implemented mechanisms to software





大纲

- 软件体系结构的发展
- 理解软件体系结构
  - 概念和定义
  - 区分物理与逻辑
  - 高层抽象
- 体系结构风格初步

高层抽象

- Components are the locus of computation and state
- Connectors are the locus of relations among components

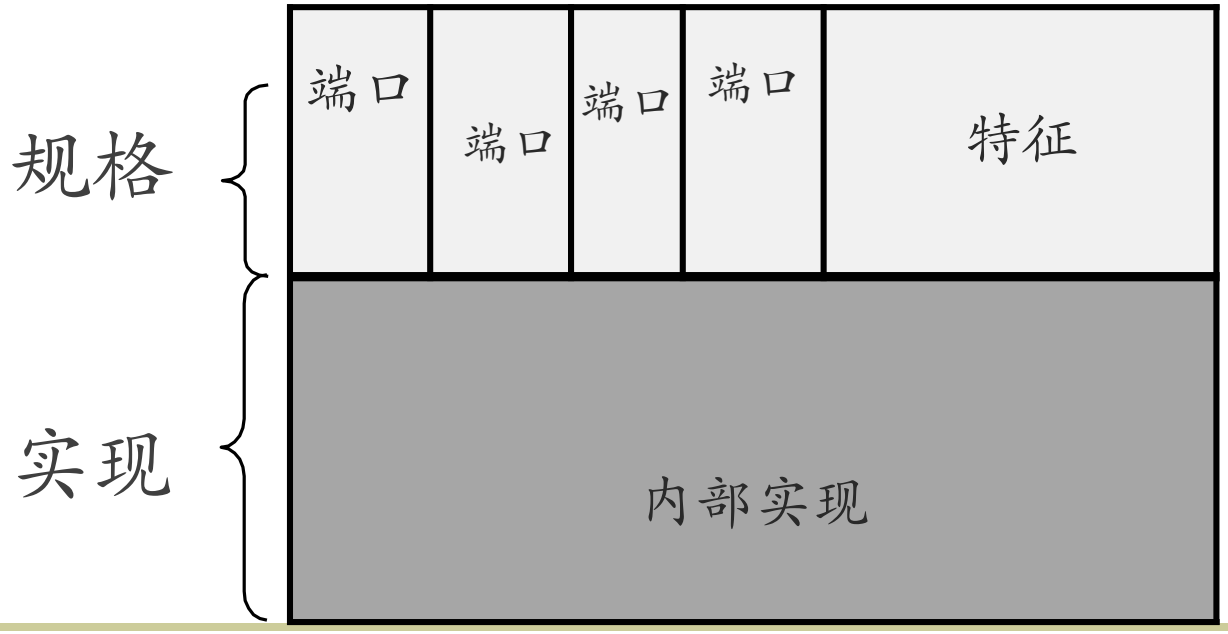
软件体系结构设计	部件、连接件、配置
软件详细设计	过程、调用；类、协作；模块、导入/导出
软件底层设计	类型、语句；数据结构、算法

理解高层抽象

- 连接件是一个与部件平等的单位。
- 部件与连接件是比类、模块等软件单位更高层次的抽象。

部件（Components）

- Elements that encapsulate processing and data in a system’s architecture are referred to as software components
- Components typically provide application-specific services



原始部件和复合部件

- 部件可以分为原始(Primitive)和复合(Composite) 两种类型。
- 原始类型的部件可以直接被实现为相应的软件实现机制。
- 复合部件则由更细粒度的部件和连接件组成,复合部件通过局部配置将其内部的部件和连接件连接起来,构成一个整体。

原始部件常用的实现机制

表 9-1 原始部件常用的软件实现机制

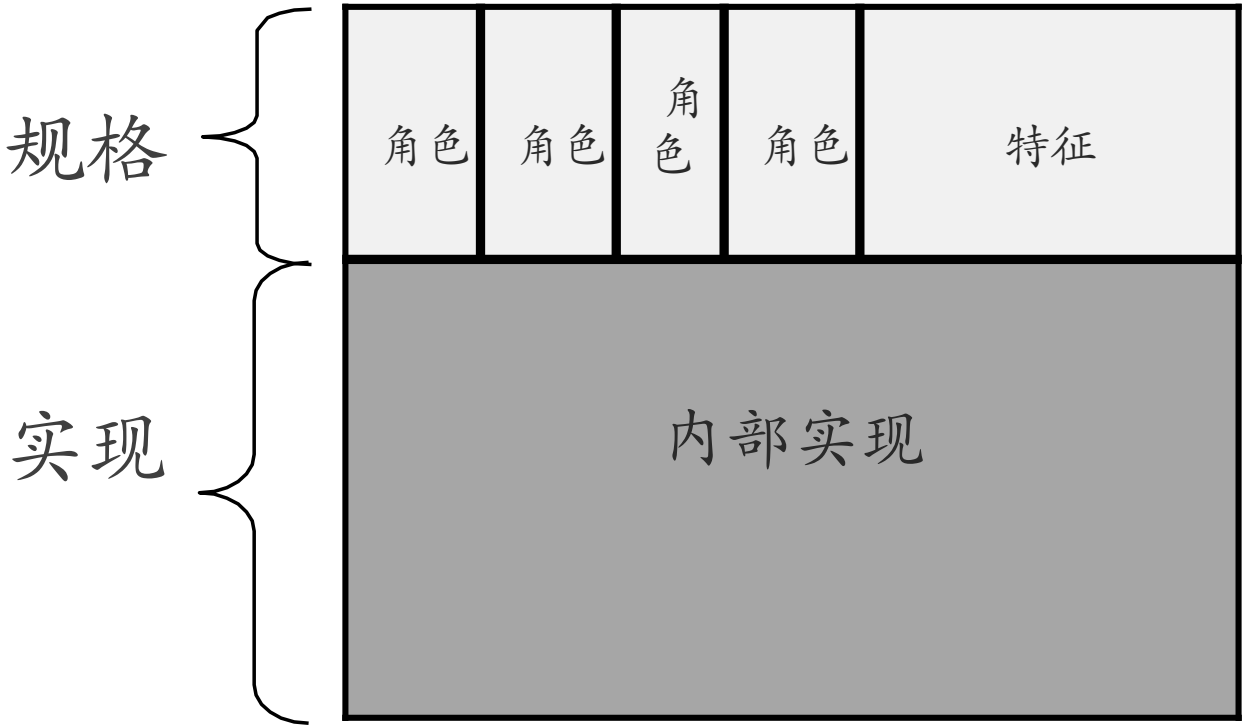
软件实现机制	示例
模块(Module)	Routine, SubRoutine, Filter
层(Layer)	View, Logical, Model
文件(File)	DLL, EXE, DAT
数据库(Database)	Repository, Center Data
进程(Process)	Sender, Receiver
网络节点(Physical Unit)	Client, Server

Component in ACME

```
Component Type Split ={Port alternateC;
                       Port othersC;
                       };
Component Type Upper={Port originalC;
                      Port upperedC;
                      };
Component Type Lower={Port originalC;
                      Port loweredC;
                      };
Component Type Merge={Port upperedAlternateC;
                      Port loweredOthersC;
                      };
```

连接件（Connectors）

- In complex systems interaction may become more important and challenging than the functionality of the individual components



原始连接件和复合连接件

- 与部件相似,在实现上连接件也可以分为原始(Primitive)和复合(Composite)两种类型。原始类型的连接件可以直接被实现为相应的软件实现机制。

- 复合连接件则由更细粒度的部件和连接件组成,复合连接件通过局部配置将其内部的部件和连接 件连接起来,构成一个整体。

原始连接件常用的软件实现机制

连接件

表 9-2 原始部件常用的软件实现机制

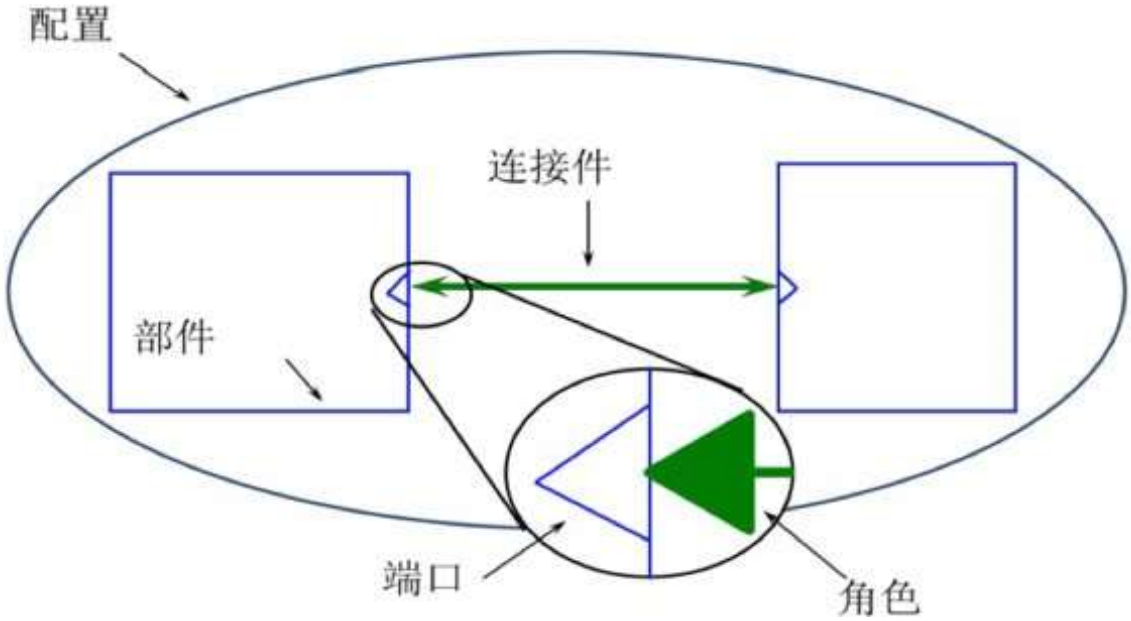
实现类型	软件实现机制	提供方
隐式（Implicit）	程序调用（Procedure Call）	编程语言机制
	共享变量（Shared variable）	
	消息（Message）	平台、框架或高级语言机制
	管道（Pipe）	
	事件（Event）	
	远程过程调用（RPC）	
	网络协议（Network Protocol）	
	数据库访问协议（Database Access Protocol）	
显式（Explicit）	适配器（Adaptor）	复杂逻辑实现
	委托（Delegator）	
	中介（Intermediate）	

Connector in ACME

```
Connector Type Pipe ={ Role source;
                       Role destination;
                       };
```

Configurations

- Components and connectors are composed in a specific way in a given system’s architecture to accomplish that system’s objective



Configuration in ACME

```
System Capitalize = {
        Component split : Split;
        Component upper : Upper;
        Component lower : Lower;
        Component merge : Merge;
        Connector {stou : Pipe; stol:Pipe; utom : Pipe; ltom:Pipe;}

         Attachments = {
                split.alternateC to stou.source;
                split.othersC to stol.source;
                upper.originalC to stou.destination;
                lower.originalC to stol.destination;
                upper.upperedC to utom.source;
                lower.loweredC to ltom.source;
                merge.upperedAlternateC to utom. destination;
                merge.loweredOtherC to ltom. destination;
        };
};
```

Wright

```
System Capitalize
    component Split =
        port in [ in protocol]
        port toUpper [toUpper protocol]
        port toLower [toLower protocol]
        spec [Split specification]
    component Upper =
        port fromSplit [fromSplit protocol]
        port toMerge [toMerge protocol]
        spec [Upper specification]
    component Lower =
        port fromSplit [fromSplit protocol]
        port toMerge[toMerge protocol]
        spec [Lower specification]
    component Merge =
        port out [ out protocol]
        port fromUpper [fromUpper protocol]
        port fromLower [fromLower protocol]
        spec [Merge specification]
    connector pipe-connector =
        role producer [producer protocol]
        role consumer [consumer protocol ]
        glue [ glue protocol]
Instances
    s: Split
    u: Upper
    l: Lower
    m: Merge
    pipe1: pipe-connector
    pipe2: pipe-connector
    pipe3: pipe-connector
    pipe4: pipe-connector
Attachments
    s.toUpper as pipe1.producer ;
    u.fromSplit as pipe1.consumer ;
    s.toLower as pipe2.producer ;
    l.fromSplit as pipe2.consumer ;
    u.toMerge as pipe3.producer ;
    m.fromUpper as pipe3.consumer ;
    l. toMerge as pipe4.producer ;
    m.fromLower as pipe4.consumer ;
end Capitalize
```

部件类型

连接件类型

实例

配置

Component & Connector in CSP
(Communicating Sequential Processes)

```
Component Split =
    port In = read?x→In □ read-eof→close→ √
    port Left, Right = write!x→Out ⊓ close→ √
    comp spec =
        let Close = In.close→Left.close→Right.close→ √
        in Close □
           In.read?x→Left.write!x → (Close □ In.read?x→Right.write!x → computation)
```

Figure 4: The Filter *Split*

```
connector Pipe =
    role Writer = write!x→Writer ⊓ close→ √
    role Reader =
      let ExitOnly = close→ √
      in let DoRead = (read?x→Reader □ read-eof→ExitOnly)
      in DoRead ⊓ ExitOnly
    glue = let ReadOnly = Reader.read!y→ReadOnly □ Reader.read-eof→Reader.close→ √
                        □ Reader.close→ √
           in let WriteOnly = Writer.write?x→WriteOnly □ Writer.close→ √
           in Writer.write?x→glue
              □ Reader.read!y→glue
              □ Writer.close→ReadOnly
              □ Reader.close→WriteOnly
    spec ∀ Reader.read_i!y • ∃ Writer.write_j?x • i = j ∧ x = y
         ∧ Reader.read-eof ⇒ (Writer.close ∧ # Reader.read = # Writer.write)
```

Figure 5: A Pipe Connector

高层抽象的好处

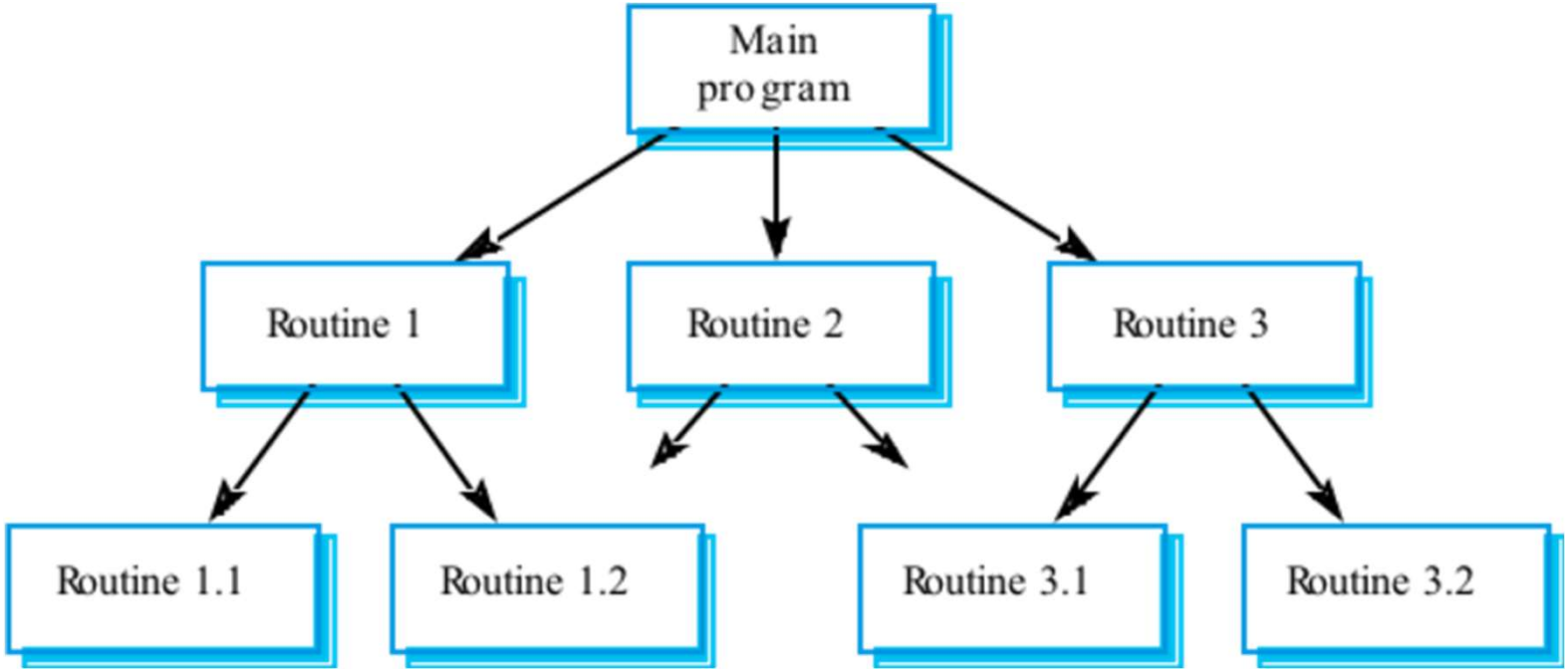
- 直观，便于理解
- 验证正确性
- 关注度分离，降低复杂度

Outline

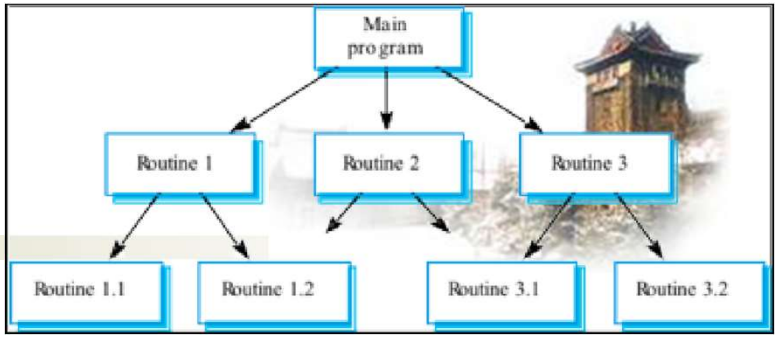
- 软件体系结构的发展
- 理解软件体系结构
- 体系结构风格初步
  - 主程序与子程序
  - 面向对象
  - 分层
  - MVC

Main Program and Subroutine Style

- Components:
  - procedures , functions and module
- Connectors:
  - calls between them



设计决策与约束



- 基于声明-使用(程序调用)关系建立连接件,以层次分解的方式建立系统部件,共同组成层次结构。
- 每一个上层部件可以“使用”下层部件,但下层部件不能“使用”上层部件,即不 允许逆方向调用。
- 系统应该是单线程执行。主程序部件拥有最初的执行控制权,并在“使用”中将控 制权转移给下层子程序。
- 子程序只能够通过上层转移来获得控制权,可以在执行中将控制权转交给下层的子 子程序,并在自身执行完成之后必须将控制权还交给上层部件。

实现

- 功能分解
- 集中控制
- 每个构件一个模块实现
  - 主要是单向依赖
- 使用utility或tools等基础模块

效果

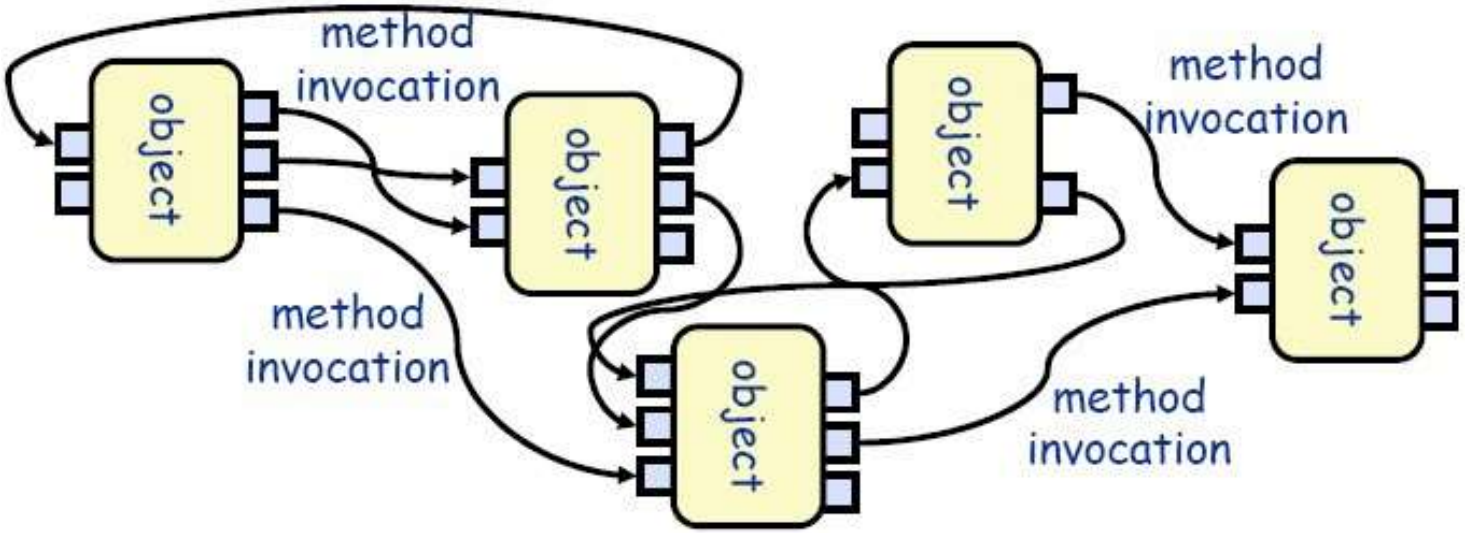
- 主程序/子程序风格的优点有:
  - 流程清晰,易于理解。
  - 强控制性。
- 主程序/子程序风格的缺点有:
  - 程序调用是一种强耦合的连接方式,非常依赖交互方的接口规格,这会使得系统难 以修改和复用。
  - 程序调用的连接方式限制了各部件之间的数据交互,可能会使得不同部件使用隐含 的共享数据交流,产生不必要的公共耦合,进而破坏它的“正确性”控制能力。

应用

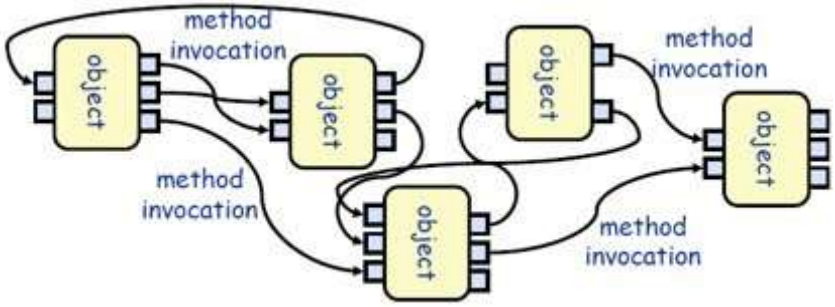
- 主程序/子程序风格主要用于能够将系统功能依层次分解为多个顺序执行步骤的系统。
- [Shaw1996]发现,在很多受到限制的编程语言环境下,这些编程语言没有模块化支持, 系统通常也会使用主程序/子程序风格,这时主程序/子程序风格的实现是程序实现,即主程 序和子程序都被实现为单独的程序。
- 一些使用结构化方法(自顶向下或自底向上)建立的软件系统也属于主程序/子程序风 格。

Object-Oriented Style

- Components:
  - object or module
- Connectors:
  - function or invocations (methods).



设计决策及约束



- 依照对数据的使用情况,用信息内聚的标准,为系统建立对象部件。每个对象部件基于内部数据提供对外服务接口,并隐藏内部数据的表示。
- 基于方法调用(Method Invocation)机制建立连接件,将对象部件连接起来。
- 每个对象负责维护其自身数据的一致性与完整性,并以此为基础对外提供“正确”的服务。
- 每个对象都是一个自治单位,不同对象之间是平级的,没有主次、从属、层次、分解等关系。

实现

- 任务分解
- (委托式)分散式控制

- 每个构件一个模块实现
  - 使用接口将双向依赖转换为单向依赖
  - 将每个构件分割为多个模块，以保证单向依赖
  - 每个模块内部可以是基于面向对象方法，也可以基于结构化
- 使用utility或tools等基础模块

效果

- 面向对象式风格的优点有:
  - 内部实现的可修改性。
  - 易开发、易理解、易复用的结构组织。

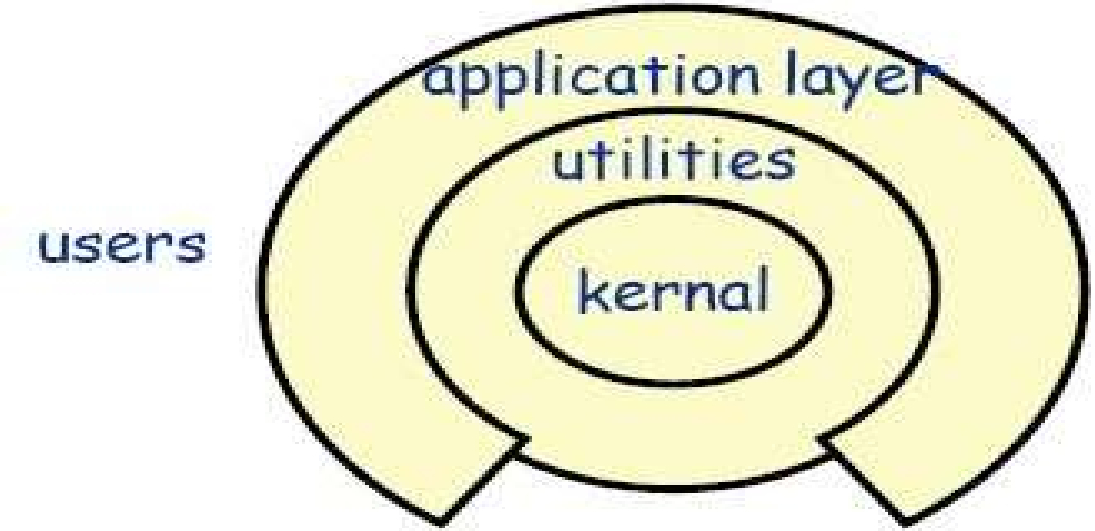
- 面向对象式风格的缺点有:
  - 接口的耦合性。
  - 标识(Identity)的耦合性。
  - 副作用。
    - 面向对象式风格借鉴了面向对象的思想,也引入了面向对象的副作用,因 此更难实现程序的“正确性”。例如,如果A 和 B 都使用对象 C,那么 B 对 C 的修改可能会 对A 产生未预期的影响。再例如对象的重入 (Reentry)问题:如果 A 的方法 f()调用了 B 的 方法 p(),而 p()又调用了 A的另一方法 q(),那么就可能使得 q()失败,因为在 q()开始执行时,A 正处于 f()留下的执行现场,这个现场可能是数据不一致的。

应用

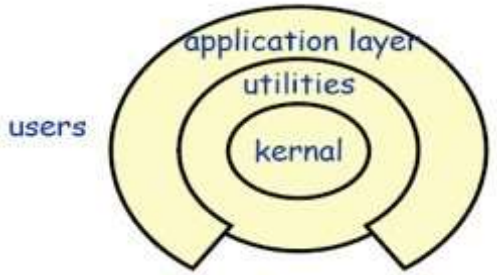
- 面向对象式风格适用于那些能够基于数据信息分解和组织的软件系统,这些系统:
  - 主要问题是标识和保护相关的数据信息;
  - 能够将数据信息和相关操作联系起来,进行封装。
- 实践中,基于抽象数据类型建立的软件系统大多属于面向对象式风格。

Layered Style

- Components: typically collections of procedures or objects.
- Connectors: typically procedure calls or methods invocations under restricted visibility.



设计决策与约束



- 从最底层到最高层,部件的抽象层次逐渐提升。每个下层为邻接上层提供服务, 每个上层将邻接下层作为基础设施使用。也就是说,在程序调用机制中上层调用下层。
- 两个层次之间的连接要遵守特定的交互协议,该交互协议应该是成熟、稳定和 标准化的。也就是说,只要遵守交互协议,不同部件实例之间是可以互相替换的。
- 跨层次的连接是禁止的,不允许第 I 层直接调用 I+N(N>1)
- 层的服务。
- 逆向的连接是禁止的,不允许第 I 层调用第 J(J<I)层的服务。

效果

- 关注点分离（每层逐次抽象）
- 层间接口使用固定协议（固定控制）

- 每层一或多个模块实现
  - 单向依赖
  - 层间数据传递建立专门模块
- 使用utility或tools等基础模块

效果

- 分层风格的优点有:
  - 设计机制清晰,易于理解。
  - 支持并行开发。
  - 更好的可复用性与内部可修改性。
- 分层风格的缺点有:
  - 交互协议难以修改。
  - 性能损失。
  - 难以确定层次数量和粒度。

应用

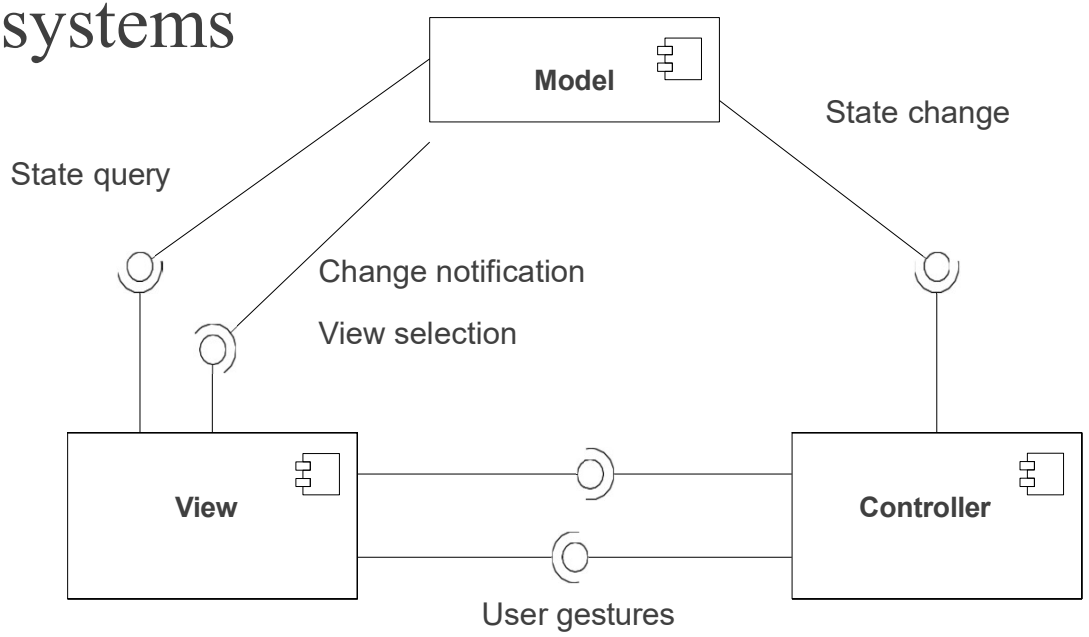
- 分层风格适用于具备下列特性的系统:
  - 主要功能是能够在不同抽象层次上进行任务分解的复杂处理;
  - 能够建立不同抽象层次之间的稳定交互协议;
  - 没有很高的实时性能要求,能够容忍稍许的延迟

- 此外,那些需要进行并行开发的软件系统也可能会使用分层风格,以便于任务分配和工作开展。

- 在现有的软件系统中,分层风格是一种经常被用到的体系结构风格,像网络通信、交互系统、硬件控制系统、系统平台等都会使用分层风格。例如,ISO网络通信模型、TCP/IP 的 网络通信模型等都使用了分层风格。

Model-View-Controller Style

- The model subsystems are designed such that they do not depend on any view or controller subsystems.
- Changes in their states are propagated to the view subsystems



Model-View-Controller Style

- Components
  - Model components are responsible for maintaining domain knowledge and notify view of changes
  - View components are responsible for displaying information to the user and send user gestures to controller
- Controller
  - Changes the state of the model
  - Maps user actions to model updates Selects view for response
- Connectors:
  - Procedure calls, Messages, Events

设计决策和约束

- 模型、视图、控制是分别是关于业务逻辑、表现和控制的三种不同内容抽象。
- 如果视图需要持续地显示某个数据的状态,那么它首先需要在模型中注册对该数据的兴趣。如果该数据状态发生了变更,模型会主动通知视图,然后再由视图查询数据的更新 情况。
- 视图只能使用模型的数据查询服务,只有控制部件可以调用可能修改模型状态的程 序。
- 用户行为虽然由视图发起,但是必须转交给控制部件处理。对接收到的用户行为, 控制部件可能会执行两种处理中的一种或两种:调用模型的服务,执行业务逻辑;提供下一 个业务展现。
- 模型部件相对独立,既不依赖于视图,也不依赖于控制。 虽然模型与视图之间存在 一个“通知变更”的连接,但该连接的交互协议是非常稳定的,可以认为是非常弱的依赖。

实现

- 模型-视图-控制风格需要为模型、视图和控制的每个部件实例建立模块实现,各模块间存在导入/导出关系,程序调用连接件不需要显式的实现。
- 特定技术实现，通常专用于WEB
  - Model与Controller单向
  - Controller与View双向
  - Model与View双向
- 典型实现
  - View：JSP，HTML
  - Controller：Servlet
  - Model：JavaBean

效果

- 模型-视图-控制风格的优点有:
  - 易开发性。视图和控制的可修改性。
  - 模型封装了系统的业务逻辑,所以是三种类型中最为复杂的系统部件。MVC 中模型是相对独立的,所以对视图实现和控制实现的修改不会影响到模 型实现。再考虑到业务逻辑通常比业务表现和控制逻辑更加稳定,所以 MVC 具有一定的可 修改性优势。
  - 适宜于网络系统开发的特征。MVC 不仅允许视图和控制的可修改性,而且其对业务 逻辑、表现和控制的分离使得一个模型可以同时建立并保持多个视图,这非常适用于网络系统开发。
- 模型-视图-控制风格的缺点有:
  - 复杂性。MVC 将用户任务分解成了表现、控制和模型三个部分,这增加了系统的复杂性,不利于理解任务实现。
  - 模型修改困难。视图和控制都要依赖于模型,因此,模型难以修改。

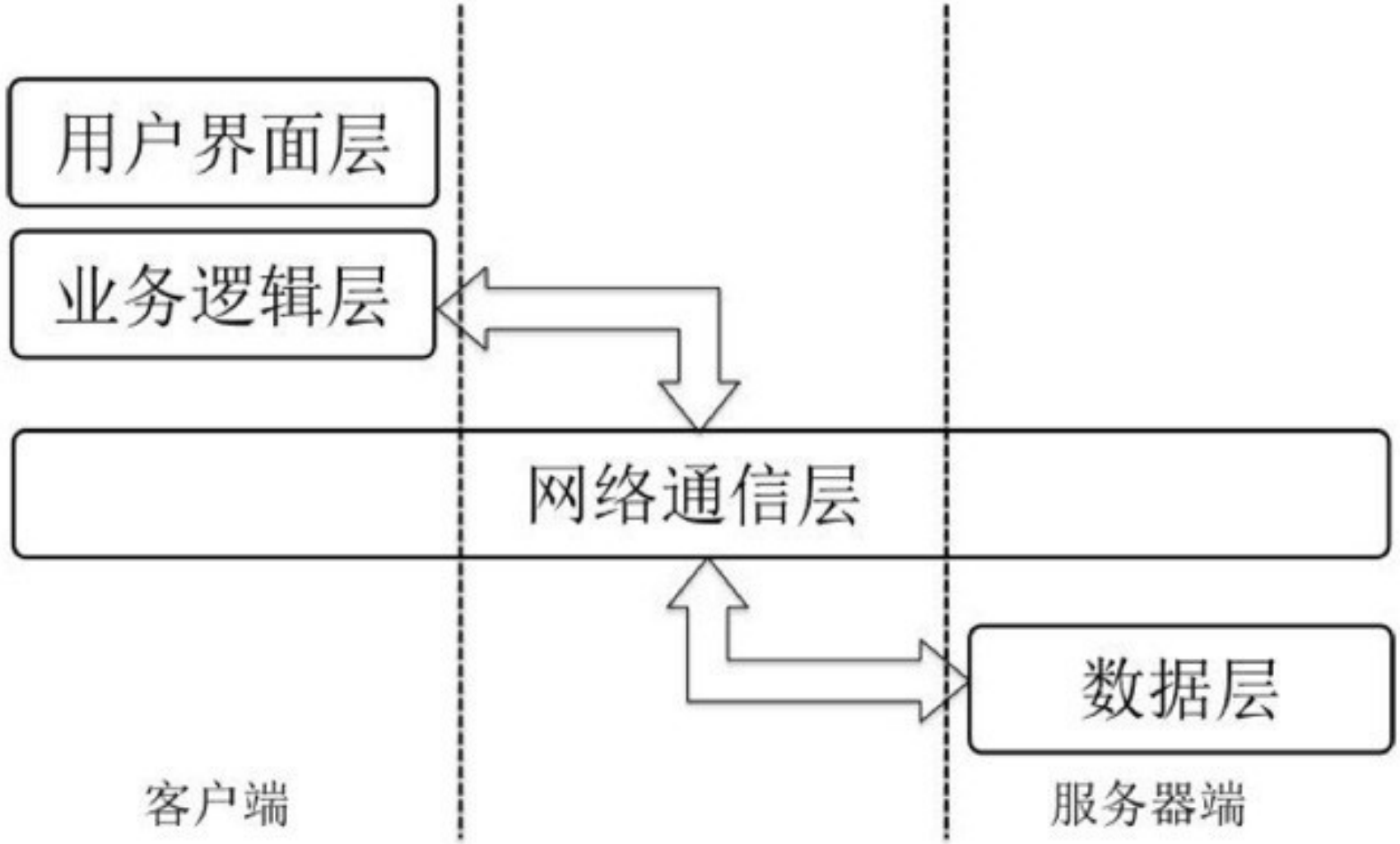
应用

- Suitable for applications:
  - Changes to user interface should be easy and possible at run-time.
  - Adapting or porting the user interface should not impact the design or code in the functional part of the application.
- Examples:
  - Web applications

分层与MVC

	分层	MVC
界面响应	需要逐层调用，每一层都有性能损耗	View 可以直接从 Model 中读取数据
可修改性	如果发生接口修改，每一层的修改只会影响上一层，Prensentation 层可以随意修改。	如果发生接口修改，Model 会影响到 Controller 和 View，Controller 会影响到 View，View 通常不会影响另外 2 个
Web 条件	受到 Web 技术（HTTP）限制，难以实现 Logic 向 Prensentation 的数据传递	比较适合
大概模块对应	Presentation	View
		Controller
	Logical	
		Model
	Data	
关联方向	上层拥有下层模块的引用	互相持有引用

连锁超市系统
分层体系结构风格方案



连锁超市系统
MVC体系结构风格方案