体系结构设计的总结与补充

1. 设计的思想

产品线

体系结构设计

框架

大规模软件设计

重构

构件

为复用而设计

信息隐藏

设计模式

结构化设计

面向对象设计

抽象数据类型

独立于软件实现

软件设计

与过程的关系

自顶向下逐
步求精

函数

程序设计

与人的关系

软件

过程

程序

软件设计思想的发展

什么是设计?

- [wiki]Design is the planning that lays the basis for the making of every object or system.
- [Brooks 2010]认为上述定义的精髓在于计划、思维和后续执行。
- [Ralph 2009]将设计定义为:
    - 设计(名词):一个对象的规格说明。它由人创造,有明确的目标,适用于特殊的环境,由一些基础类型构件组成,满足一个需求集合,受一定的限制条件约束。
    - 设计(动词):在一个环境中创建对象的规格说明。

什么是设计?

- Designing often requires a designer to consider the aesthetic, functional, and many other aspects of an object or a process, which usually requires considerable research, thought, modeling, interactive adjustment, and re-design.

- 设计经常需要一个设计师考虑一个对象或过程的审美、功能以及其他方面，这通常需要进行相当的研究、 思考、 建模、 交互调整和重新设计。

Engineering design or Art design?

工程设计与艺术设计

- [Faste2001]认为软件设计要:时刻保持以用户为中心,为其建造有 用的软件产品;将设计知识科学化、系统化,并能够通过职业教育产生合格的软件设计师; 能够进行设计决策与折中,解决设计过程中出现的不确定性、信息不充分、要求冲突等复杂 情况。

- Vitruvius(《De Architectura》,公元前 22 年)认为好的建筑架构要满足“效用(Useful)、 坚固(Solid)与美感(Beautiful)”三个方面的要求。

- [Brooks2010]认为重要的美感因素包括:简洁、结构 清晰和一致。

- [Smith1996]认为在软件设计(尤其是人机交互设计)中艺术始终都处于中心地位,比工程性更加重要,为此设计师需要学会:发散性思维和创新;与用户共情,体会他们的内心感受;进行相关因素的评价和平衡,例如可靠性与时尚(Fashion)、简洁性与可修改性等;构思与想象,设计软件产品的可视化外观(Visualization)。

Design = Engineering +Art

Rational or Pragmatic?
理性主义or经验主义？

理性主义代表

- 理性主义更看重设计的工程性,希望以科学化知识为基础,利用模型语言、建模方法、 工具支持,将软件设计过程组织成系统、规律的模型建立过程[McPhee1996]。

- 在考虑到人的 因素时,理想主义认为人是优秀的,虽然会犯错,但是可以通过教育不断完善自己 [Brooks2010]。

- 设计方法学的目标就是不断克服人的弱点,持续完善软件设计过程中的不足,最终达到完美[McPhee1996]。

- 形式化软件工程的支持者是典型的理想主义。

经验主义代表

- 经验主义者则在重视工程性的同时,也强调艺术性,要求给软件设计过程框架添加一些 灵活性以应对设计中人的因素。

- [Parnas1986]曾指出没有过程指导和完全依赖个人的软件设 计活动是不能接受的,因为不能保证质量和工程性。但是[Parnas1986]也指出一些人的因素 决定了完全理性的设计过程是不存在的:○1 用户并不知道他们到底想要怎样的需求;○2 即使 用户知道需要什么,仍然有些事情需要反复和迭代才能发现或理解;○3 人类的认知能力有限; ○4 需求的变更无法避免;○5 人类总是会犯错的;○6 人们会固守一些旧有的设计理念;○7 不合 适复用。

- 所以,[Parnas1986]认为软件设计需要使用一些方法弥补人的缺陷,以建立一个尽 可能好的软件设计过程。文档化、原型、尽早验证、迭代式开发等都被实践证明能够有效弥 补人类的缺陷[Parnas1986, Brooks 2010]

2. 软件设计的分层

程序设计的建立

- 1950s
  - 第一代语言，第二代语言
  - 语句为最小单位
- 1960s
  - 第三代语言
  - 类型与函数：第一次复杂系统分割
- 1970s
  - 类型与函数的成熟：形式化方法
  - 数据结构+算法=程序

低层设计

- 将基本的语言单位（类型与语句），组织起来，建立高质量的 数据结构+算法
- 常见设计场景：
  - 数组的使用，链表的使用，内存的使用，遍历算法，递归算法…
- 经典场景：
  - 堆栈，队列，树，排序算法，查找算法…
- 数据结构与算法审美：
  - 简洁、结构清晰，坚固（可靠、高效、易读）
  - 数据结构与算法　课程
  - <计算机程序设计艺术>

低层设计的本质

- 屏蔽程序中复杂数据结构与算法的实现细节！

抽象层 接口层	数据结构的含义与使用	算法的语义与复杂度
精化层 实现层	结构的类型定义 存储空间的使用与更改	书写控制语句 分支的处理技巧 对数据的操纵

低层设计：代码设计

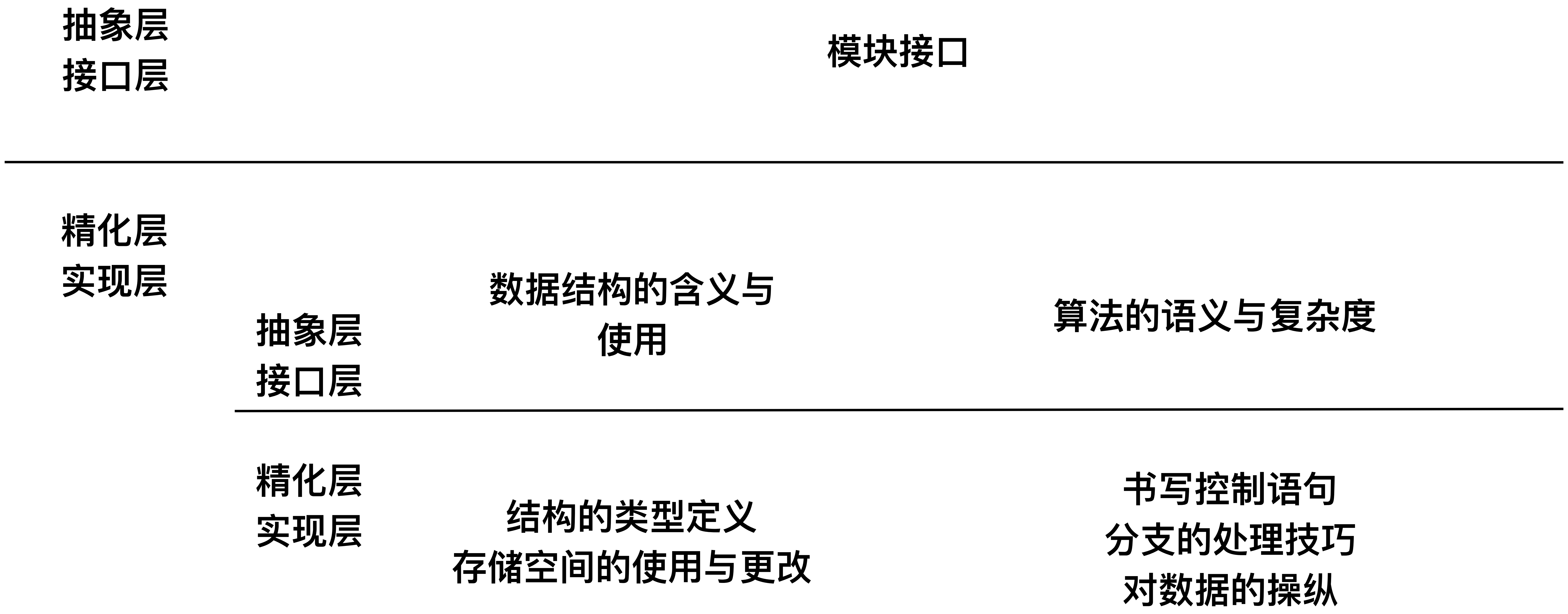
- 对一个方法/函数的内部代码进行设计
- 又被称为软件构造“software construction”，通常由程序员独立完成
- 依赖于语言提供的机制（程序设计课程）
    - OO or Structural
    - Reference or Address Point
    - …

模块划分

- 1970s
    - 函数的成熟与模块的出现
- 模块划分
    - 将系统分成简单片段
        - 片段有名字，可以被反复使用
- 名字和使用方法称为模块的抽象与接口
- 模块内部的程序片段为精化与实现

中层设计的开始

- 模块划分隐藏一些程序片段（数据结构+算法）的细节，暴露接口于外界



抽象层
接口层

模块接口

精化层
实现层

抽象层
接口层

数据结构的含义与使用

算法的语义与复杂度

精化层
实现层

结构的类型定义
存储空间的使用与更改

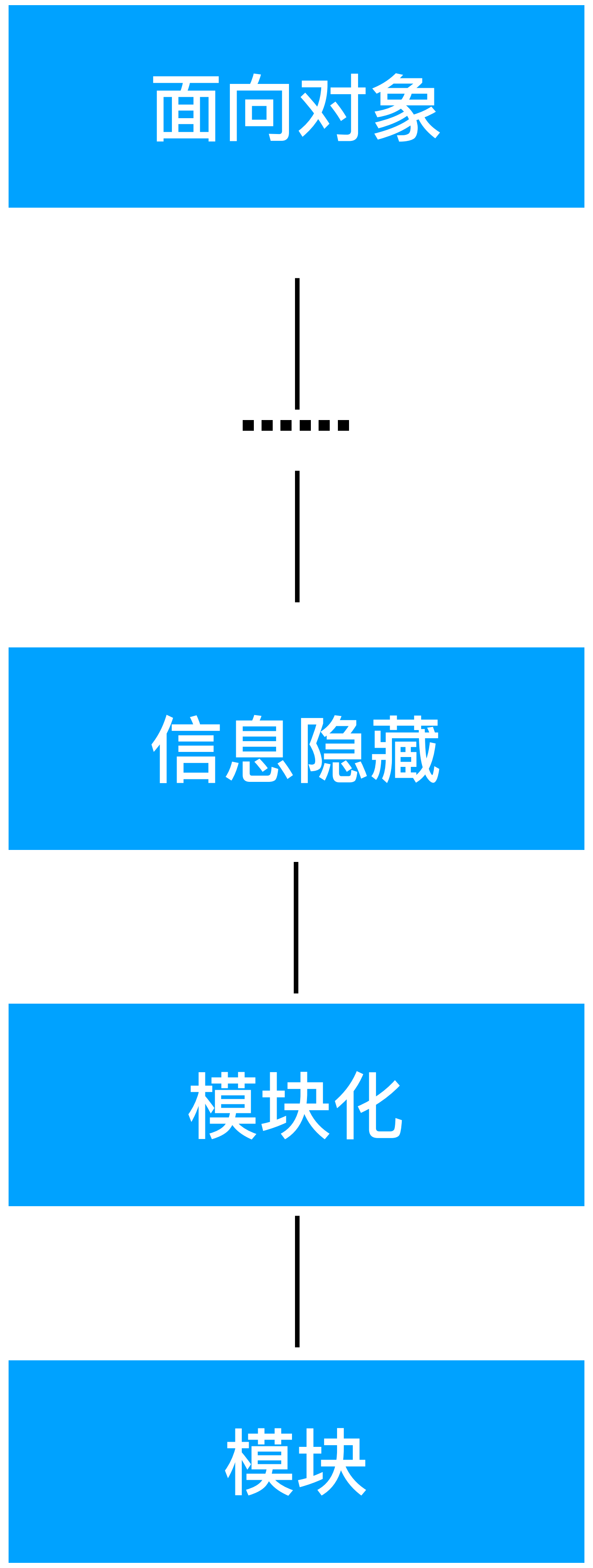
书写控制语句
分支的处理技巧
对数据的操纵

模块化的目标：完全独立性

- 完全独立有助于
    - 理解
    - 使用与复用
    - 开发
    - 修改

模块化的问题与困难

- 程序片段之间不可能是完全独立的
- 方法：实现尽可能的独立
    - 模块化
    - 信息隐藏
    - 抽象数据类型
    - 封装
    - …



中层设计总结——设计目标

- 最终审美目标：简洁性、结构清晰、一致性、质量（可修改、易开发（易理解、易测试、易调试）、易复用）

- 直接评价标准：模块化；信息隐藏；OO原则

抽象层 接口层		对象、模块	
精化层 实现层	抽象层 接口层	数据结构的含义与使用	算法的语义与复杂度
	精化层 实现层	结构的类型定义 存储空间的使用与更改	书写控制语句 分支的处理技巧 对数据的操纵

中低层设计的问题

- 《Programming-in-the-Small VS Programming-in-the-Large》
- 过于依赖细节
  - 连接与依赖，接口与实现
- 忽略的关键因素:无法有效抽象部件的整体特性
  - 总体结构
  - 质量属性

大型软件开发的一个根本不同是它更关注如何将大批独立模块组织形成一个“系统”，也就是说更重视系统的总体组织

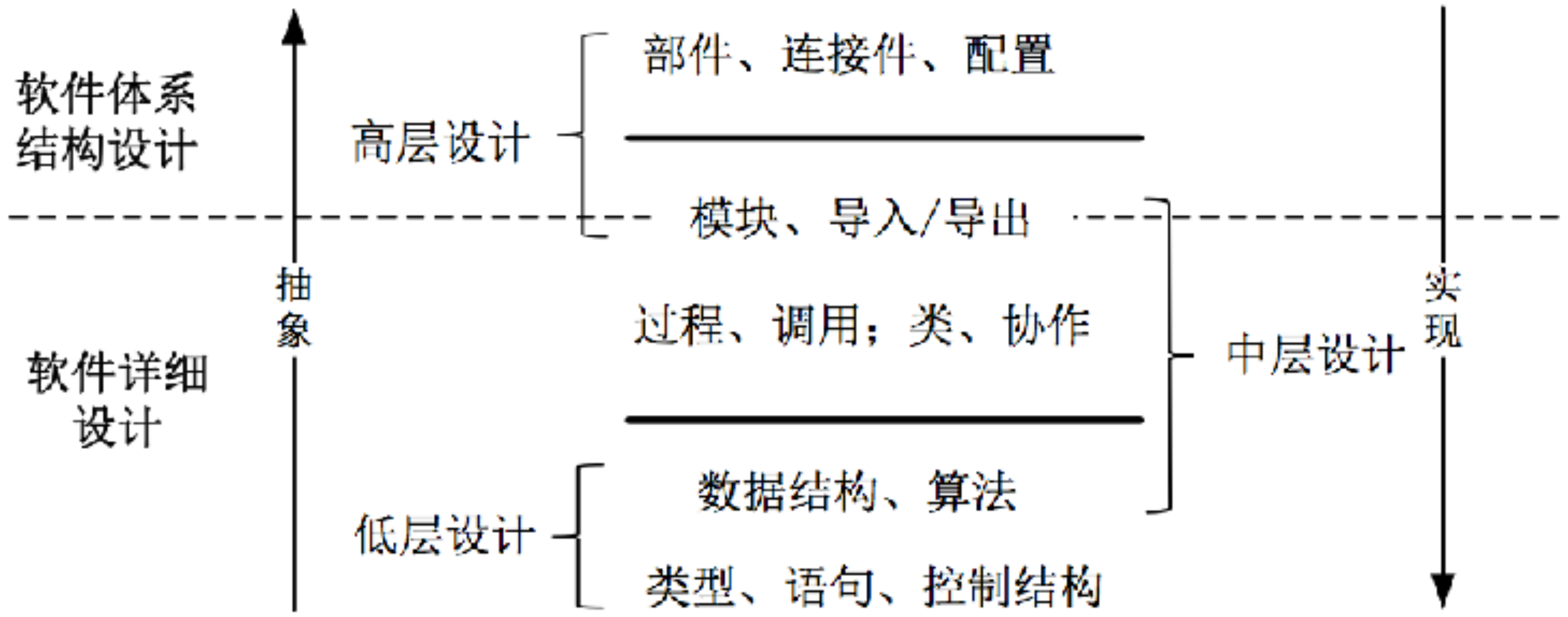
高层设计：体系结构

- 部件承载了系统主要的计算与状态

- 连接件承载部件之间的交互

- 部件与连接件都是抽象的类型定义（就像类定义），它们的实例（就像类的对象实例）组织构成软件系统的整体结构，配置将它们的实例连接起来

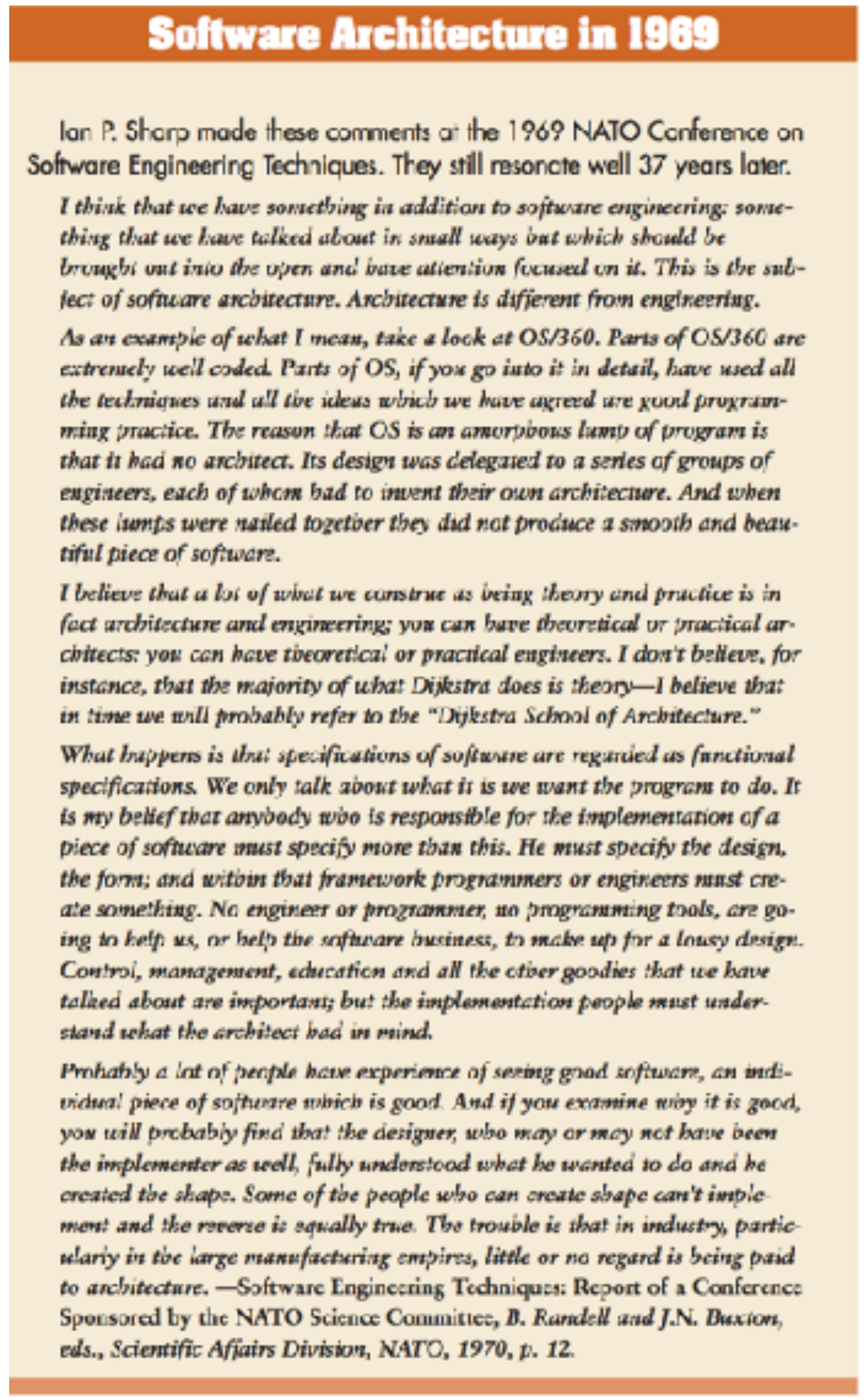
- 连接件是一个与部件平等的单位



敏捷视点

- 只有高层设计良好，底层设计才可能良好
- 只有写完并测试代码之后，才能算是完成了设计！
- 源代码可能是主要的设计文档，但它通常不是唯一一个必须的。

3 History of Architecture Design



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.

First reference to the phrase software architecture 1969

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 software 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.

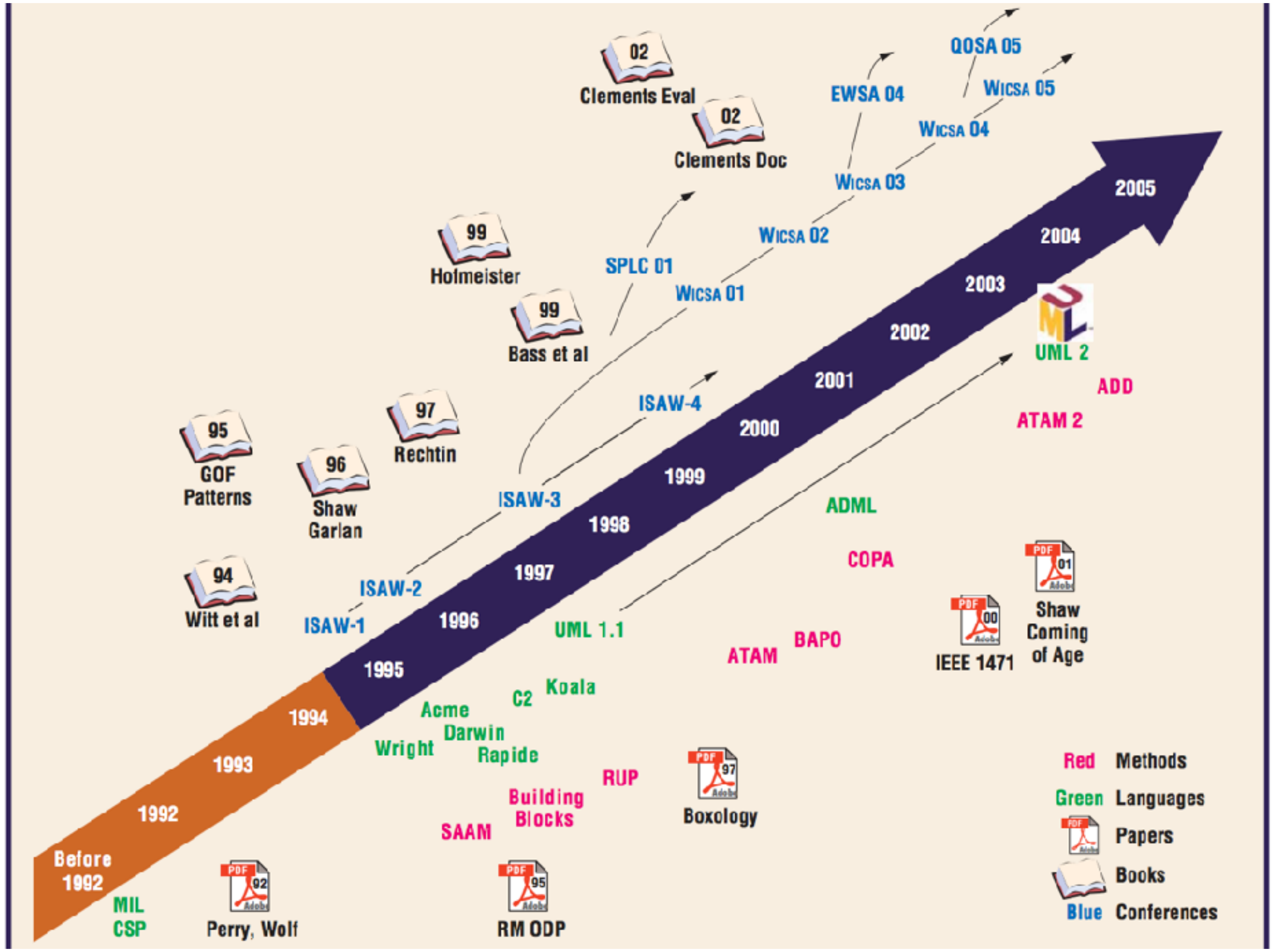
- For many researchers, the “elements” in the formula were components and connectors. These were the basis for a flurry of architecture description languages (ADLs), including C2, Rapide, Darwin, Wright, ACME, and Unicon, which unfortunately haven’t yet taken much root in industry.

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]

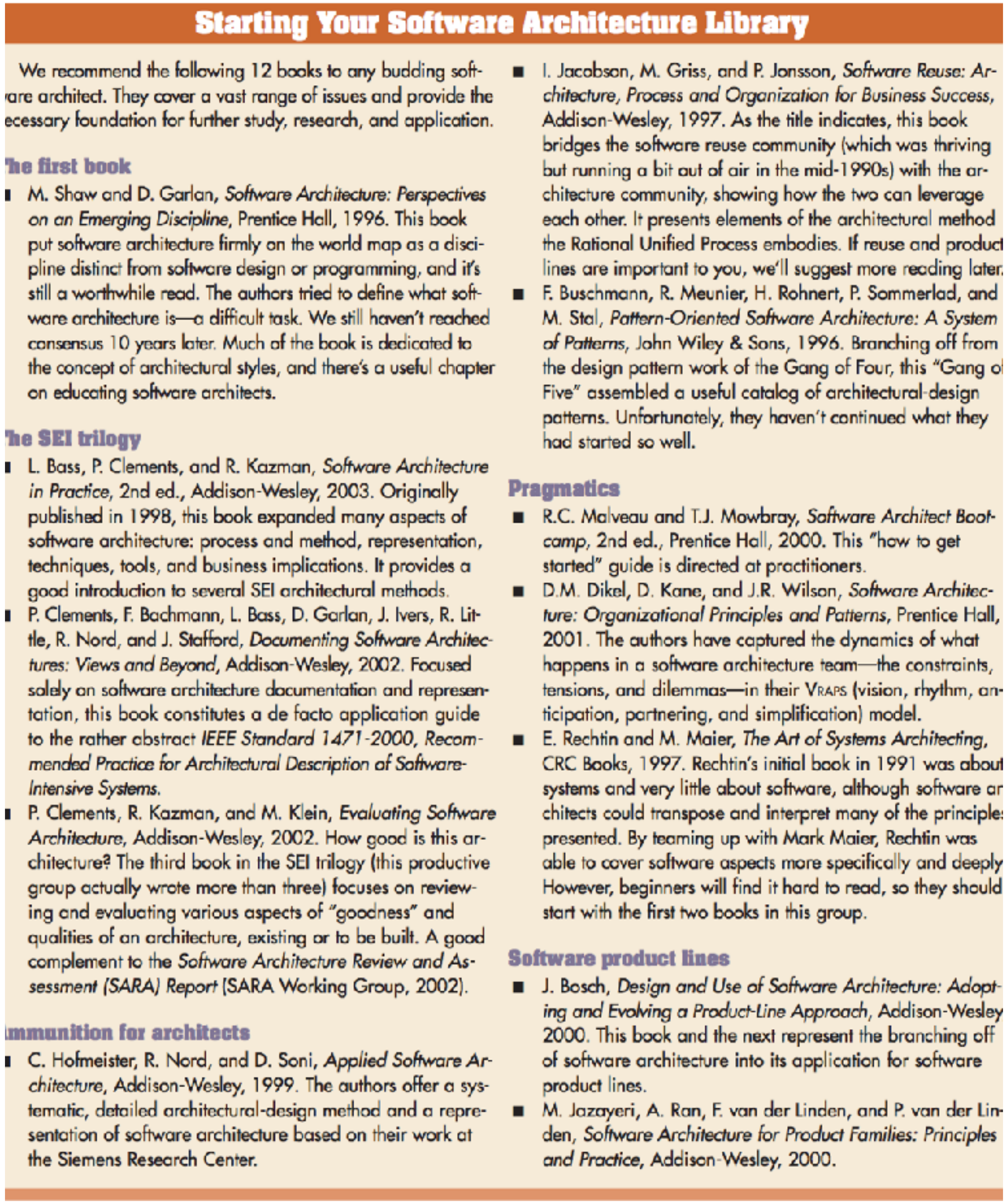


软件体系结构的十年 -- Philippe Kruchten

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 ecessary 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.

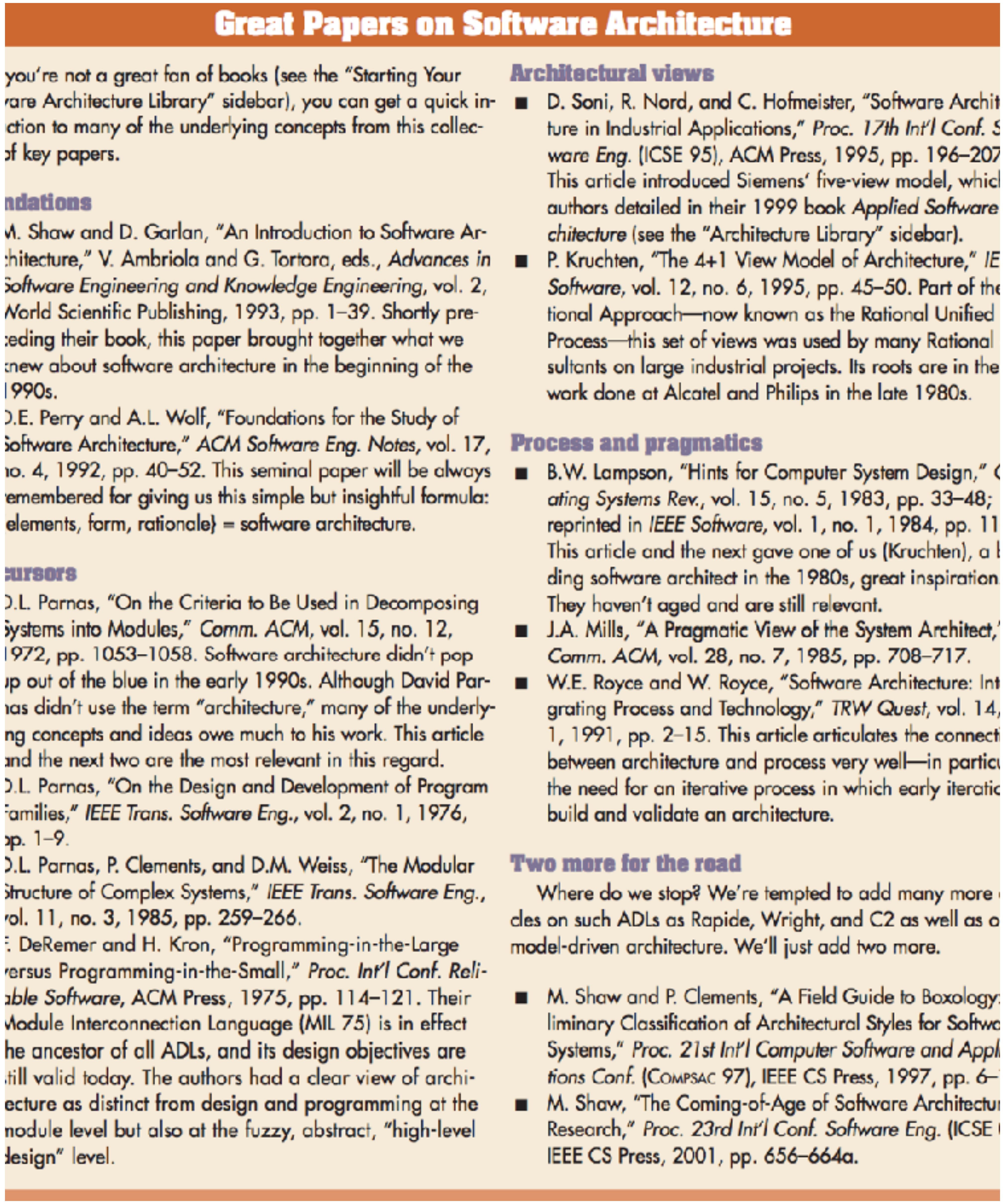
Pragmatics

- 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

you're not a great fan of books (see the "Starting Your
vare Architecture Library" sidebar), you can get a quick in-
ction to many of the underlying concepts from this collec-
of key papers.

ndations

M. Shaw and D. Garlan, "An Introduction to Software Ar-
chitecture," V. Ambriola and G. Tortora, eds., *Advances in
Software Engineering and Knowledge Engineering*, vol. 2,
World Scientific Publishing, 1993, pp. 1–39. Shortly pre-
ceding 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.

cursors

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 Par-
nas didn't use the term "architecture," many of the underly-
ing 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. Reli-
able 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 archi-
tecture 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 Archit
ture in Industrial Applications," *Proc. 17th Int'l Conf. S
ware Eng.* (ICSE 95), ACM Press, 1995, pp. 196–207
This article introduced Siemens' five-view model, whic
authors detailed in their 1999 book *Applied Software
chitecture* (see the "Architecture Library" sidebar).
- P. Kruchten, "The 4+1 View Model of Architecture," *IE
Software*, vol. 12, no. 6, 1995, pp. 45–50. Part of the
tional Approach—now known as the Rational Unified
Process—this set of views was used by many Rational
sultants on large industrial projects. Its roots are in the
work done at Alcatel and Philips in the late 1980s.

Process and pragmatics

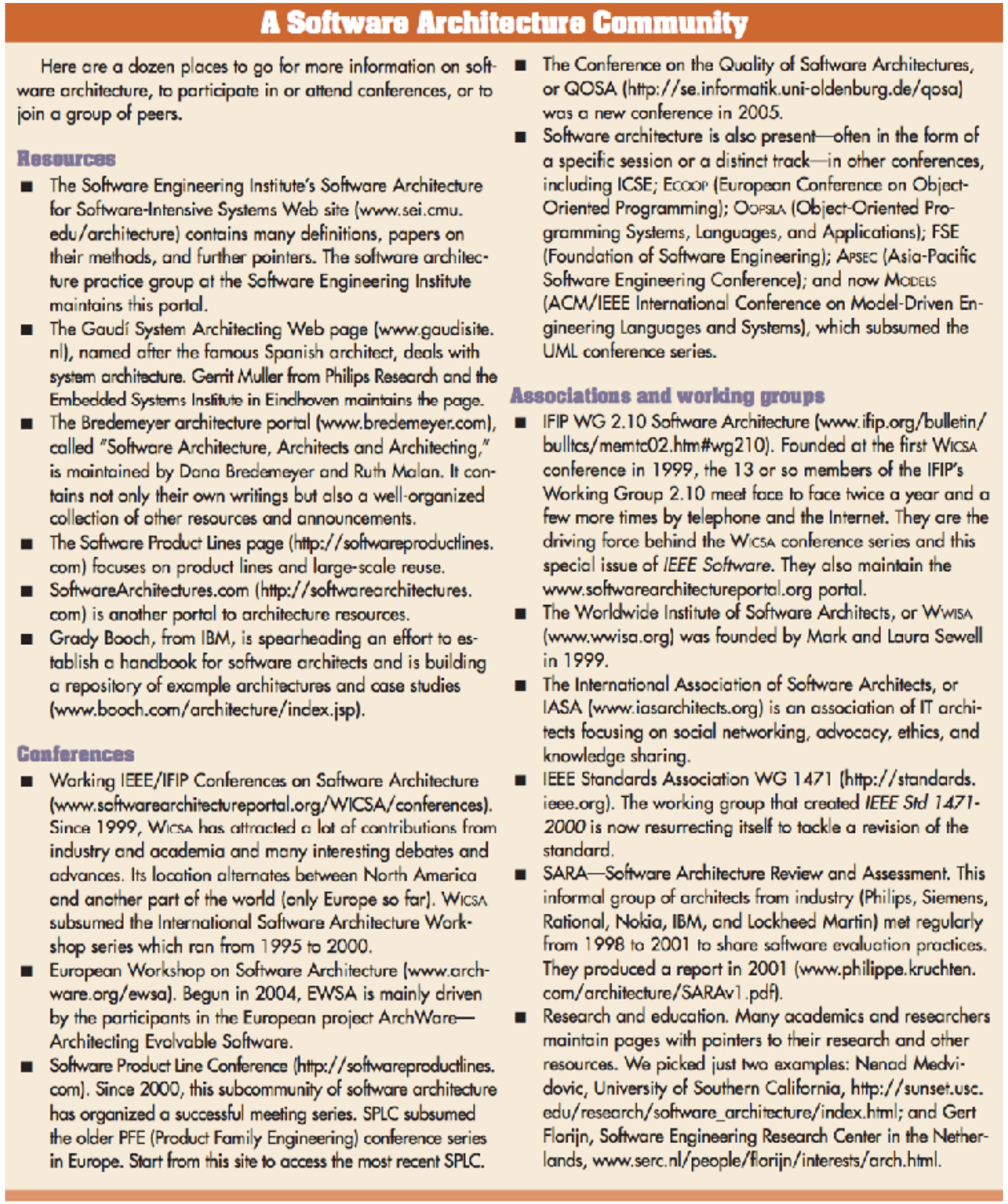
- B.W. Lampson, "Hints for Computer System Design,"
ating Systems Rev., vol. 15, no. 5, 1983, pp. 33–48;
reprinted in *IEEE Software*, vol. 1, no. 1, 1984, pp. 11
This article and the next gave one of us (Kruchten), a
ding software architect in the 1980s, great inspiration
They haven't aged and are still 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: Int
grating Process and Technology," *TRW Quest*, vol. 14,
1, 1991, pp. 2–15. This article articulates the connect
between architecture and process very well—in partic
the need for an iterative process in which early iteratio
build and validate an architecture.

Two more for the road

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

- M. Shaw and P. Clements, "A Field Guide to Boxology:
liminary Classification of Architectural Styles for Softwa
Systems," *Proc. 21st Int'l Computer Software and Appl
tions Conf.* (COMPSAC 97), IEEE CS Press, 1997, pp. 6–
- M. Shaw, "The Coming-of-Age of Software Architectur
Research," *Proc. 23rd Int'l Conf. Software Eng.* (ICSE
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

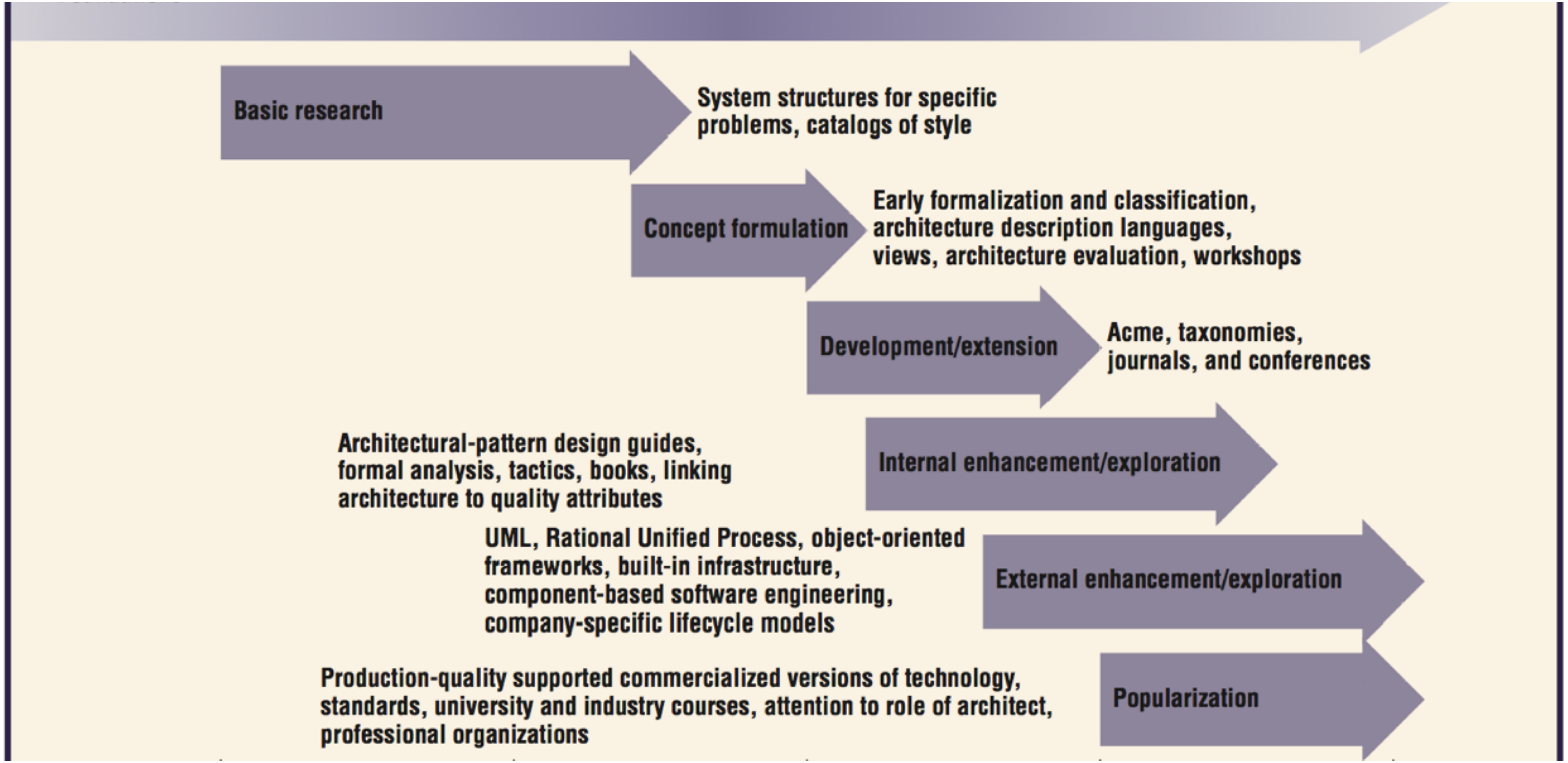
- The Software Engineering Institute's Software Architecture for Software-Intensive Systems Web site (www.sei.cmu.edu/architecture) contains many definitions, papers on their methods, and further pointers. The software architecture practice group at the Software Engineering Institute maintains this portal.
- The Gaudí 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.archware.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.



Basic research

System structures for specific problems, catalogs of style

Concept formulation

Early formalization and classification, architecture description languages, views, architecture evaluation, workshops

Development/extension

Acme, taxonomies, journals, and conferences

Internal enhancement/exploration

Architectural-pattern design guides, formal analysis, tactics, books, linking architecture to quality attributes

External enhancement/exploration

UML, Rational Unified Process, object-oriented frameworks, built-in infrastructure, component-based software engineering, company-specific lifecycle models

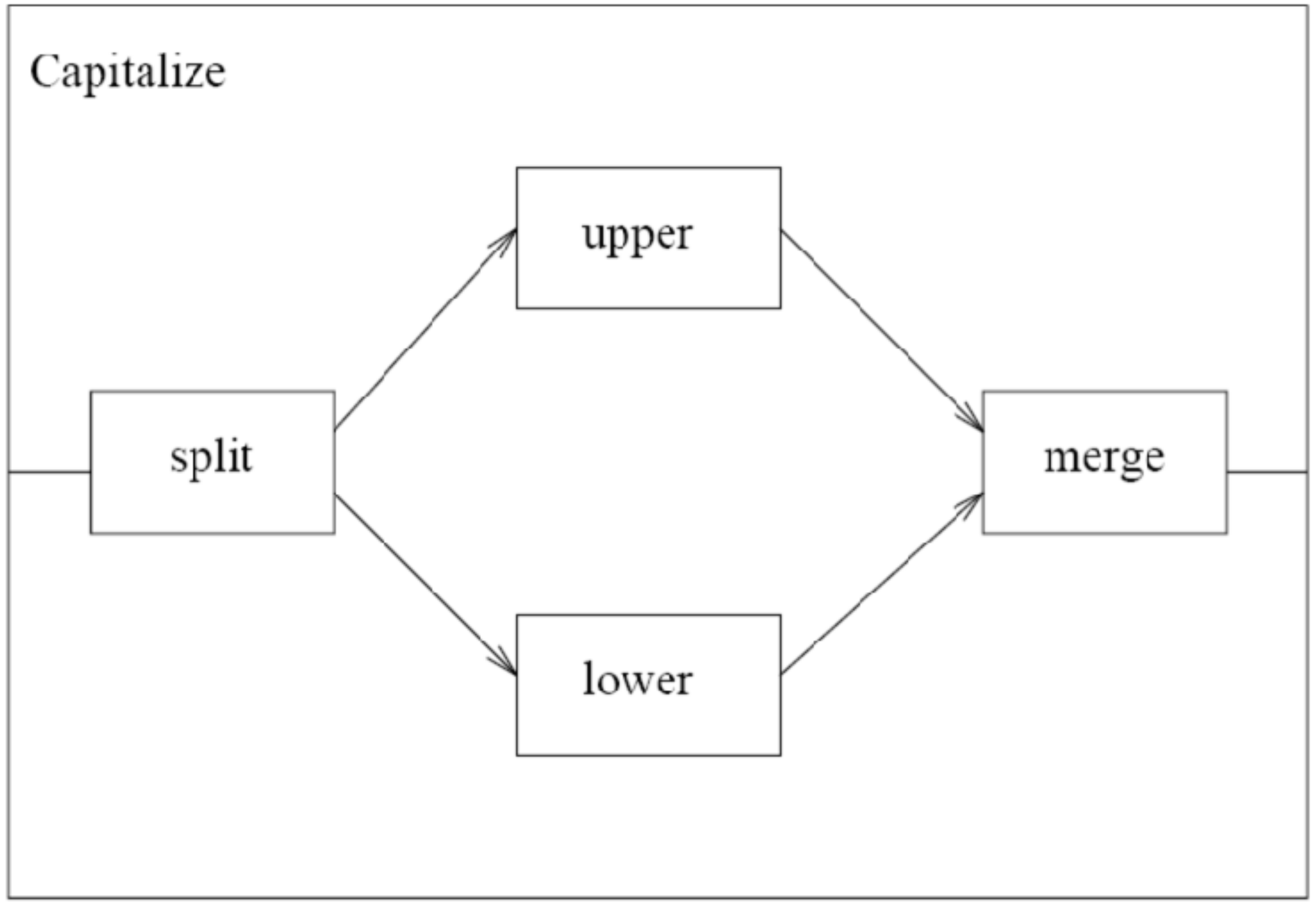
Popularization

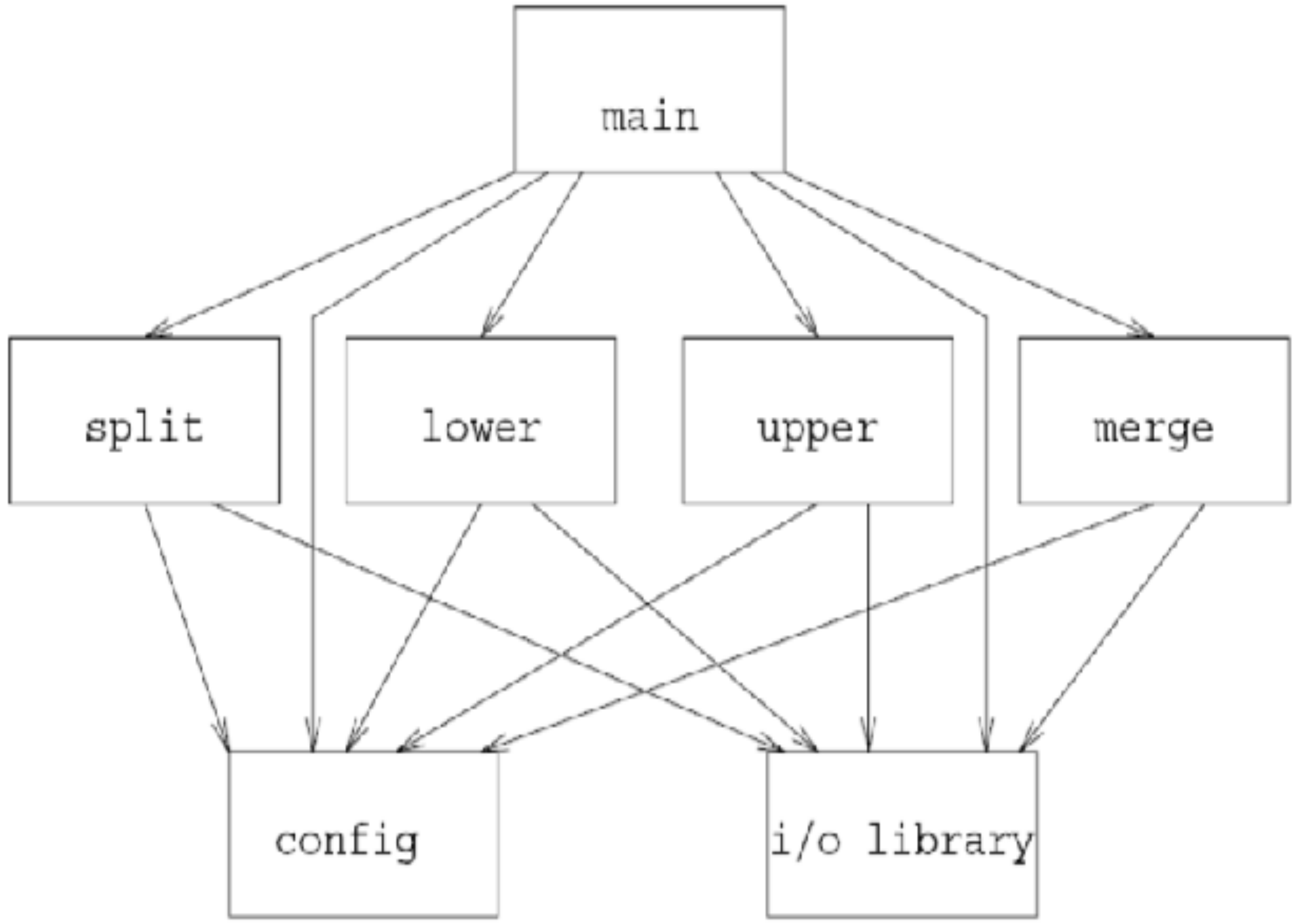
Production-quality supported commercialized versions of technology, standards, university and industry courses, attention to role of architect, professional organizations

The golden age of software architecture -- Mary Shaw

4. Logic vs Physical

抽象vs实现





- Architecture as-design
- Functional organizations

- Architecture as-implementation
- implemented mechanisms to software

5. 体系结构集成与测试

集成的策略

- 当体系结构中原型各个模块的代码都编写完成并经过单元测试之后,需要将所有模块组 合起来形成整个软件原型系统,这就是集成。集成的目的是为了逐步让各个模块合成为一个 系统来工作,从而验证整个系统的功能、性能、可靠性等需求。对于被集成起来的系统一般 主要是通过其暴露出来的接口,伪装一定的参数和输入,进行黑盒测试。

- 根据从模块之间集成的先后顺序,一般有下列几种常见的集成策略:
  - 大爆炸式
  - 增量式
    - 自顶向下式
    - 自底向上式
    - 三明治式
    - 持续集成

自顶向下

- 自顶向下集成的优点:
    - 按深度优先可以首先实现和验证一个完整的功能需求;
    - 只需最顶端一个驱动(Driver);
    - 利于故障定位。
- 自顶向下集成的缺点:
    - 桩的开发量大;
    - 底层验证被推迟,且底层组件测试不充分。

自底向上

- 自底向上集成的优点:
    - 是对底层组件行为较早验证;
    - 底层组件开发可以并行;
    - 桩的工作量少;
    - 利于故障定位。
- 自底向上集成的缺点:
    - 驱动的开发工作量大;
    - 对高层的验证被推迟,设计上的高层错误不能被及时发现。

持续集成

- 一种增量集成方法,但它提倡尽早集成和频繁集成。

- 尽早集成是指不需要总是等待一个模块开发完成才把它集成起来,而是在开发之初就利 用 Stub 集成起来。

- 频繁集成是指开发者每次完成一些开发任务之后,就可以用开发结果替换 Stub 中的相 应组件,进行集成与测试。一般来说,每人每天至少集成一次,也可以多次。

- 结合尽早集成和频繁集成的办法,持续集成可以做到:

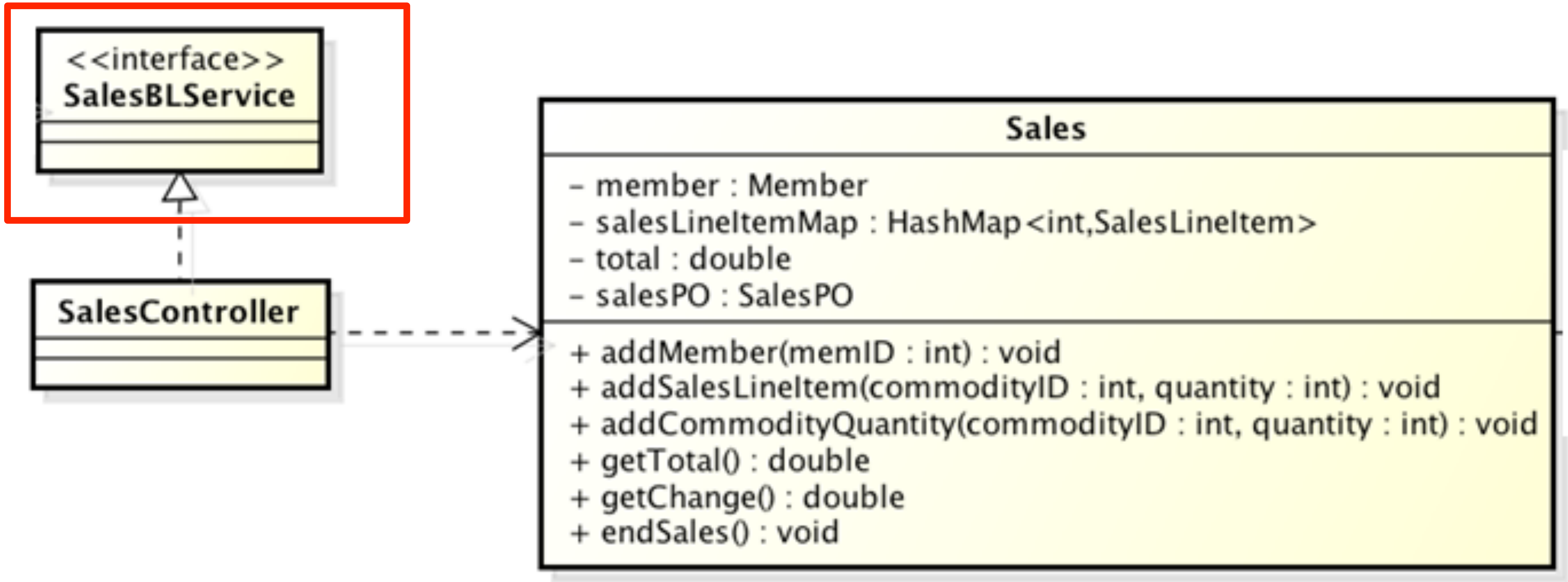
  - 防止软件开发中出现无法集成与发布的情况。因为软件项目在任何时刻都是可以集成和发布的。

  - 有利于检查和发现集成缺陷。因为最早的版本主要集成了简单的 Stub,比较容易做到没有错误。后续代码逐渐开发完成后,频繁集成又使得即使出现集成问题也能 够尽快发现、尽快解决。

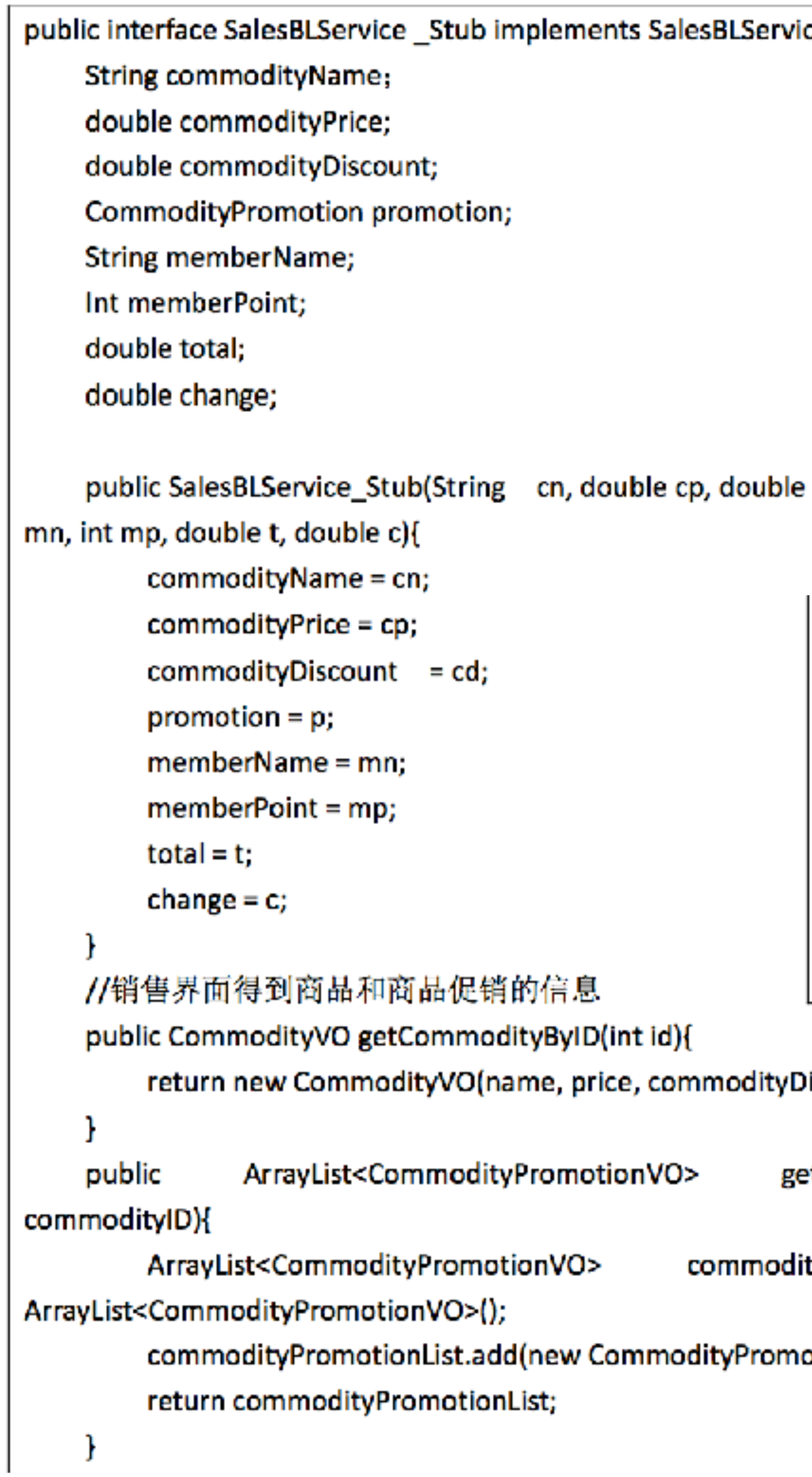
- 持续集成的频率很高,所以手动的集成对软件工程师来说是无法接受的,必须利用版本 控制工具和持续集成工具。如果程序员可以仅仅使用一条命令就完成一次完整的集成,开发 团队才有动力并且能够坚持进行频繁的集成。

依据模块接口建立桩程序Stub

- Stub
    - 为了完成程序的编译和连接而使用的暂时代码
    - 对外模拟和代替承担模块接口的关键类
    - 比真实程序简单的多，使用最为简单的逻辑

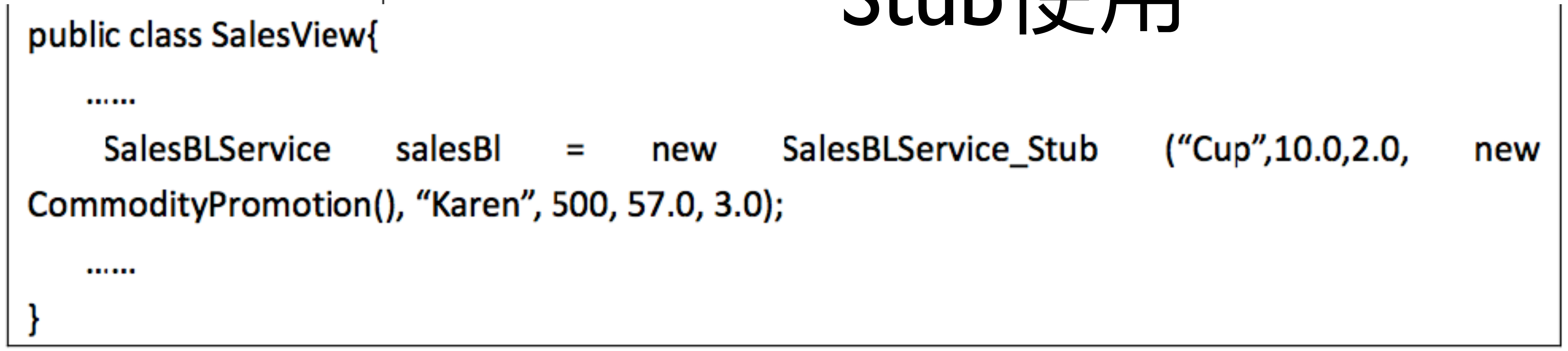

<<interface>>
SalesBLService
SalesController
Sales
- member : Member
- salesLineItemMap : HashMap<int,SalesLineItem>
- total : double
- salesPO : SalesPO
+ addMember(memID : int) : void
+ addSalesLineItem(commodityID : int, quantity : int) : void
+ addCommodityQuantity(commodityID : int, quantity : int) : void
+ getTotal() : double
+ getChange() : double
+ endSales() : void

Stub定义



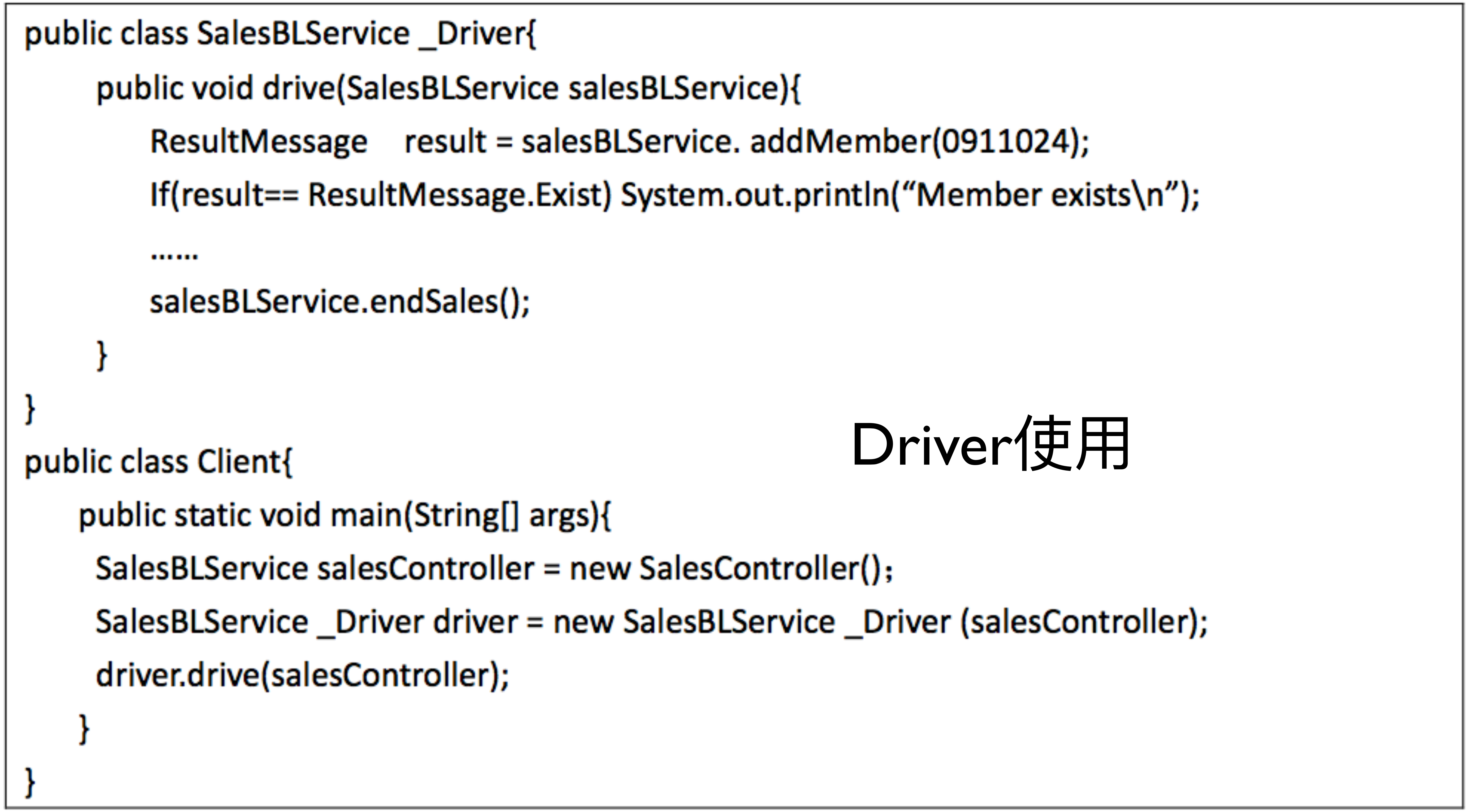
```
public interface SalesBLService _Stub implements SalesBLService {
    String commodityName;
    double commodityPrice;
    double commodityDiscount;
    CommodityPromotion promotion;
    String memberName;
    Int memberPoint;
    double total;
    double change;

    public SalesBLService_Stub(String   cn, double cp, double cd, CommodityPromotion p,String
mn, int mp, double t, double c){
        commodityName = cn;
        commodityPrice = cp;
        commodityDiscount   = cd;
        promotion = p;
        memberName = mn;
        memberPoint = mp;
        total = t;
        change = c;
    }
    //销售界面得到商品和商品促销的信息
    public CommodityVO getCommodityByID(int id){
        return new CommodityVO(name, price, commodityDiscount );
    }
    public        ArrayList<CommodityPromotionVO>        getCommodityPromotionListByID(int
commodityID){
        ArrayList<CommodityPromotionVO>        commodityPromotionList        =        new
ArrayList<CommodityPromotionVO>();
        commodityPromotionList.add(new CommodityPromotionVO(promotion));
        return commodityPromotionList;
    }
```

Stub使用



```
public class SalesView{
    ……
    SalesBLService    salesBl    =    new    SalesBLService_Stub    (“Cup”,10.0,2.0,    new
CommodityPromotion(), “Karen”, 500, 57.0, 3.0);
    ……
}
```

Driver定义



```
public class SalesBLService _Driver{
        public void drive(SalesBLService salesBLService){
              ResultMessage    result = salesBLService. addMember(0911024);
              If(result== ResultMessage.Exist) System.out.println("Member exists\n");
              ……
              salesBLService.endSales();
        }
}
public class Client{
      public static void main(String[] args){
        SalesBLService salesController = new SalesController();
        SalesBLService _Driver driver = new SalesBLService _Driver (salesController);
        driver.drive(salesController);
      }
}
```

Driver使用

集成与构建

- 集成
  - 使用桩程序辅助集成
  - 编译与链接
- 持续集成
  - 逐步编写各个模块内部程序，替换相应的桩程序
  - 真实程序不仅实现业务逻辑，而且会使用其他模块的接口程序（真实程序或者桩程序）
  - 每次替换之后都进行集成与测试

编写驱动程序，在桩程序帮助下进行集成测试

项目实践

- 客户端
    - View层：HCI特殊测试
    - Logical层：
        - 驱动模拟来自View层的请求
        - 桩程序模拟远程Data层
- 服务器端（Data层）
    - 驱动模拟来自客户端的请求
    - 桩程序模拟未完成的Data模块接口

6. MVC vs Multilayer

Reference

- https://msdn.microsoft.com/en-us/library/ff649643.aspx

- [Burbeck92] Burbeck, Steve. "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)."University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive. Available at: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html.

- [Fowler03] Fowler, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley, 2003.

- [Gamma95] Gamma, Helm, Johnson, and Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Context

- The purpose of many computer systems is to retrieve data from a data store and display it for the user. After the user changes the data, the system stores the updates in the data store. Because the key flow of information is between the data store and the user interface, you might be inclined to tie these two pieces together to reduce the amount of coding and to improve application performance. However, this seemingly natural approach has some significant problems. One problem is that the user interface tends to change much more frequently than the data storage system. Another problem with coupling the data and user interface pieces is that business applications tend to incorporate business logic that goes far beyond data transmission.

Problem

- How do you modularize the user interface functionality of a Web application so that you can easily modify the individual parts?

Force

- User interface logic tends to change more frequently than business logic, especially in Web-based applications. For example, new user interface pages may be added, or existing page layouts may be shuffled around. After all, one of the advantages of a Web-based thin-client application is the fact that you can change the user interface at any time without having to redistribute the application. If presentation code and business logic are combined in a single object, you have to modify an object containing business logic every time you change the user interface. This is likely to introduce errors and require the retesting of all business logic after every minimal user interface change.

- In some cases, the application displays the same data in different ways. For example, when an analyst prefers a spreadsheet view of data whereas management prefers a pie chart of the same data. In some rich-client user interfaces, multiple views of the same data are shown at the same time. If the user changes data in one view, the system must update all other views of the data automatically.

- Designing visually appealing and efficient HTML pages generally requires a different skill set than does developing complex business logic. Rarely does a person have both skill sets. Therefore, it is desirable to separate the development effort of these two parts.

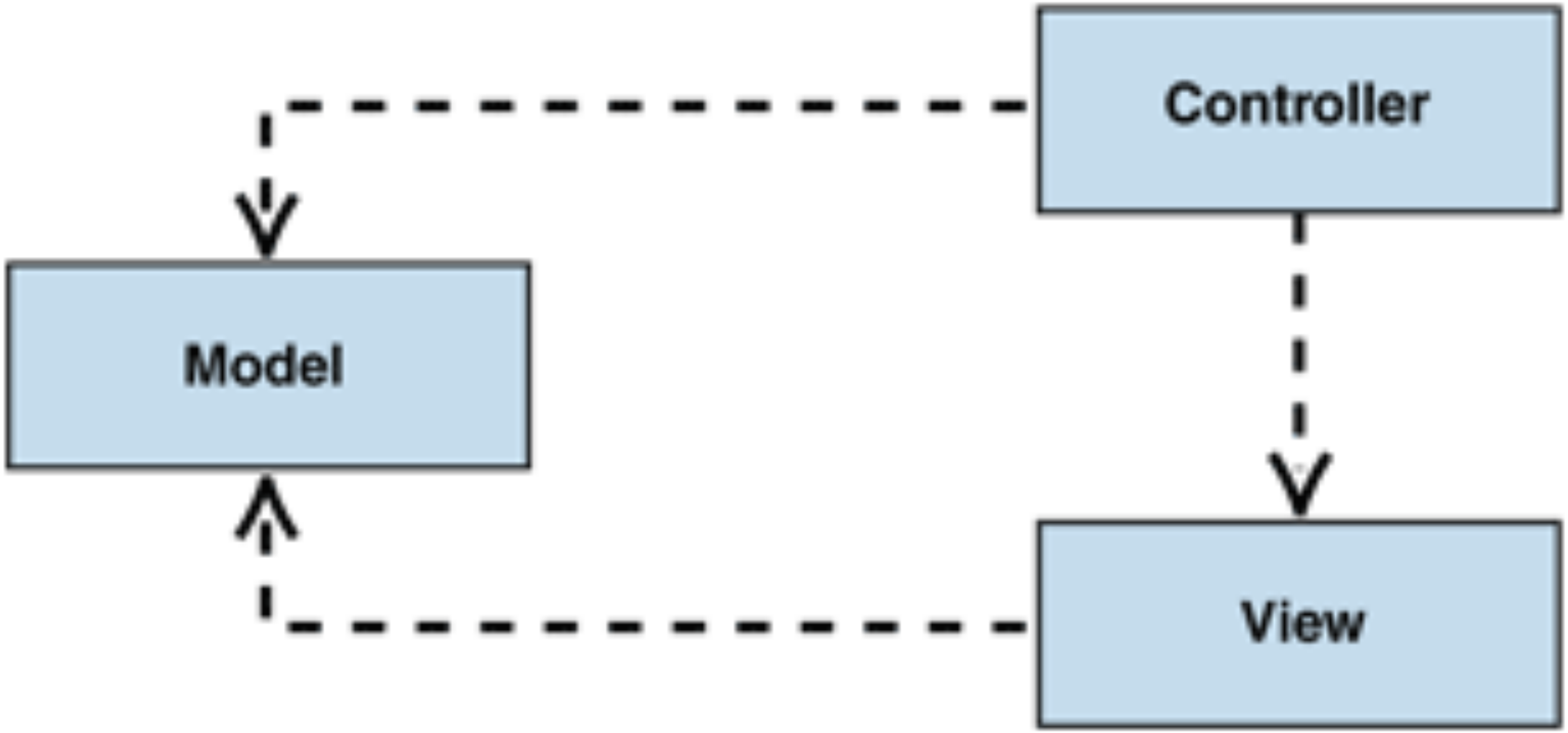
- User interface activity generally consists of two parts: presentation and update. The presentation part retrieves data from a data source and formats the data for display. When the user performs an action based on the data, the update part passes control back to the business logic to update the data.

Force

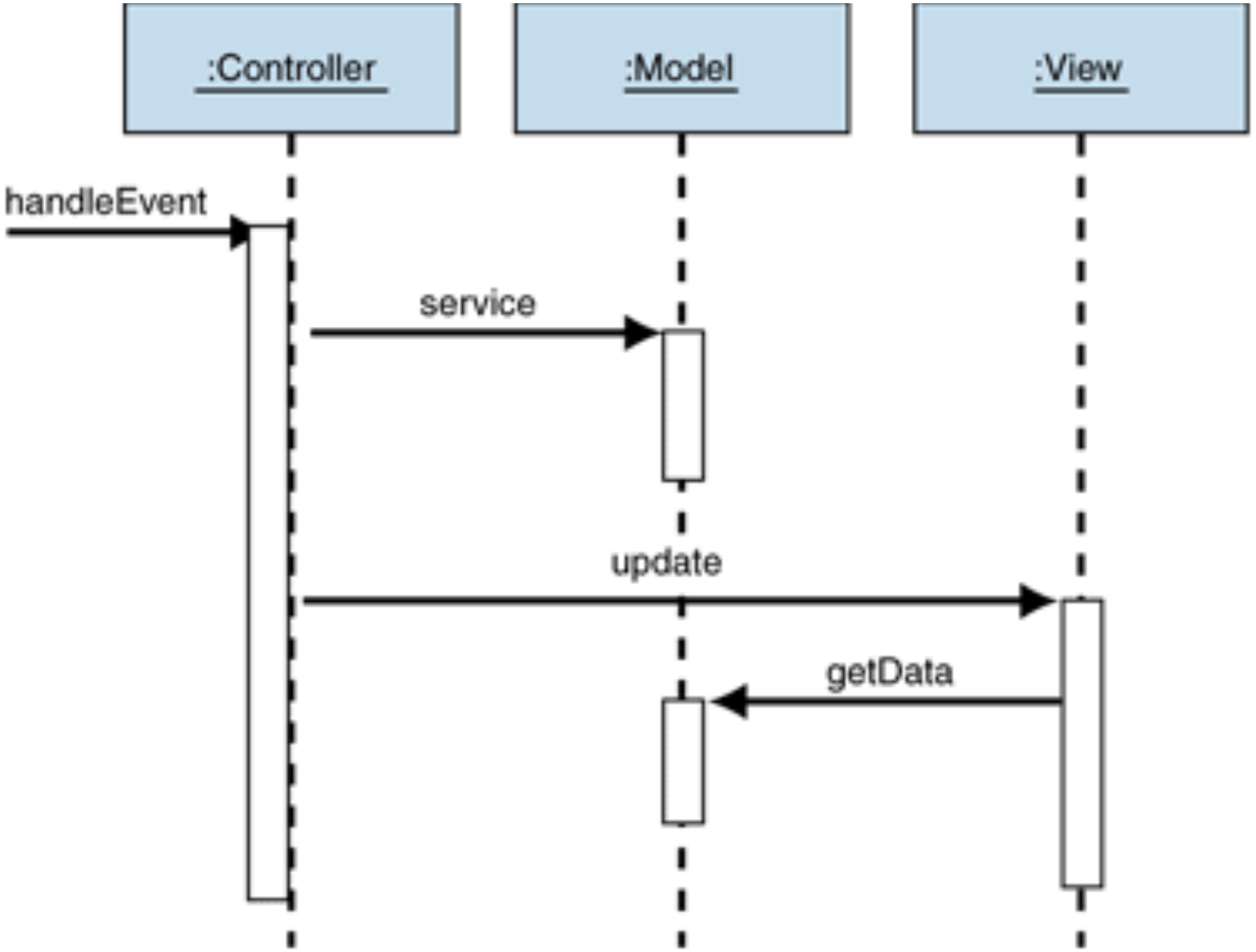
- In Web applications, a single page request combines the processing of the action associated with the link that the user selected with the rendering of the target page. In many cases, the target page may not be directly related to the action. For example, imagine a simple Web application that shows a list of items. The user returns to the main list page after either adding an item to the list or deleting an item from the list. Therefore, the application must render the same page (the list) after executing two quite different commands (adding or deleting)-all within the same HTTP request.

- User interface code tends to be more device-dependent than business logic. If you want to migrate the application from a browser-based application to support personal digital assistants (PDAs) or Web-enabled cell phones, you must replace much of the user interface code, whereas the business logic may be unaffected. A clean separation of these two parts accelerates the migration and minimizes the risk of introducing errors into the business logic.

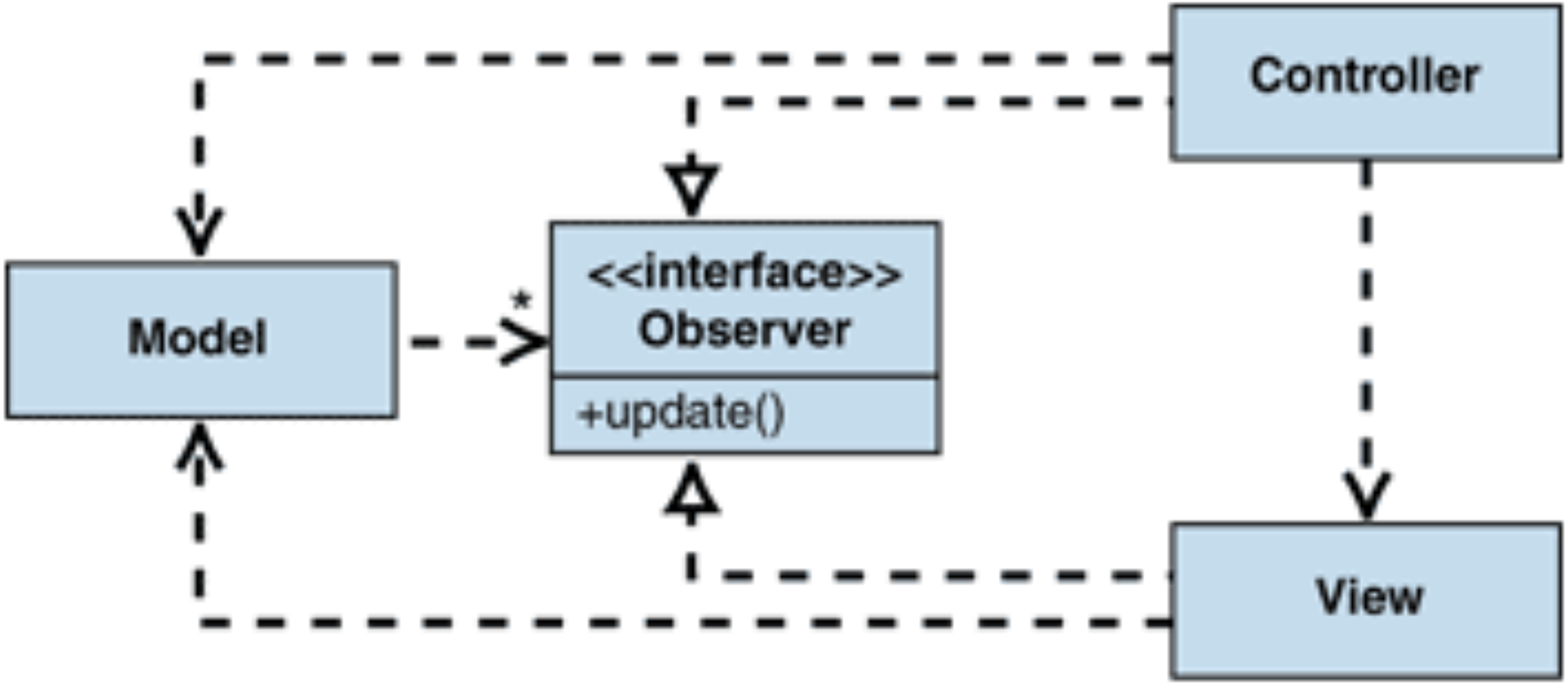
- Creating automated tests for user interfaces is generally more difficult and time-consuming than for business logic. Therefore, reducing the amount of code that is directly tied to the user interface enhances the testability of the application.



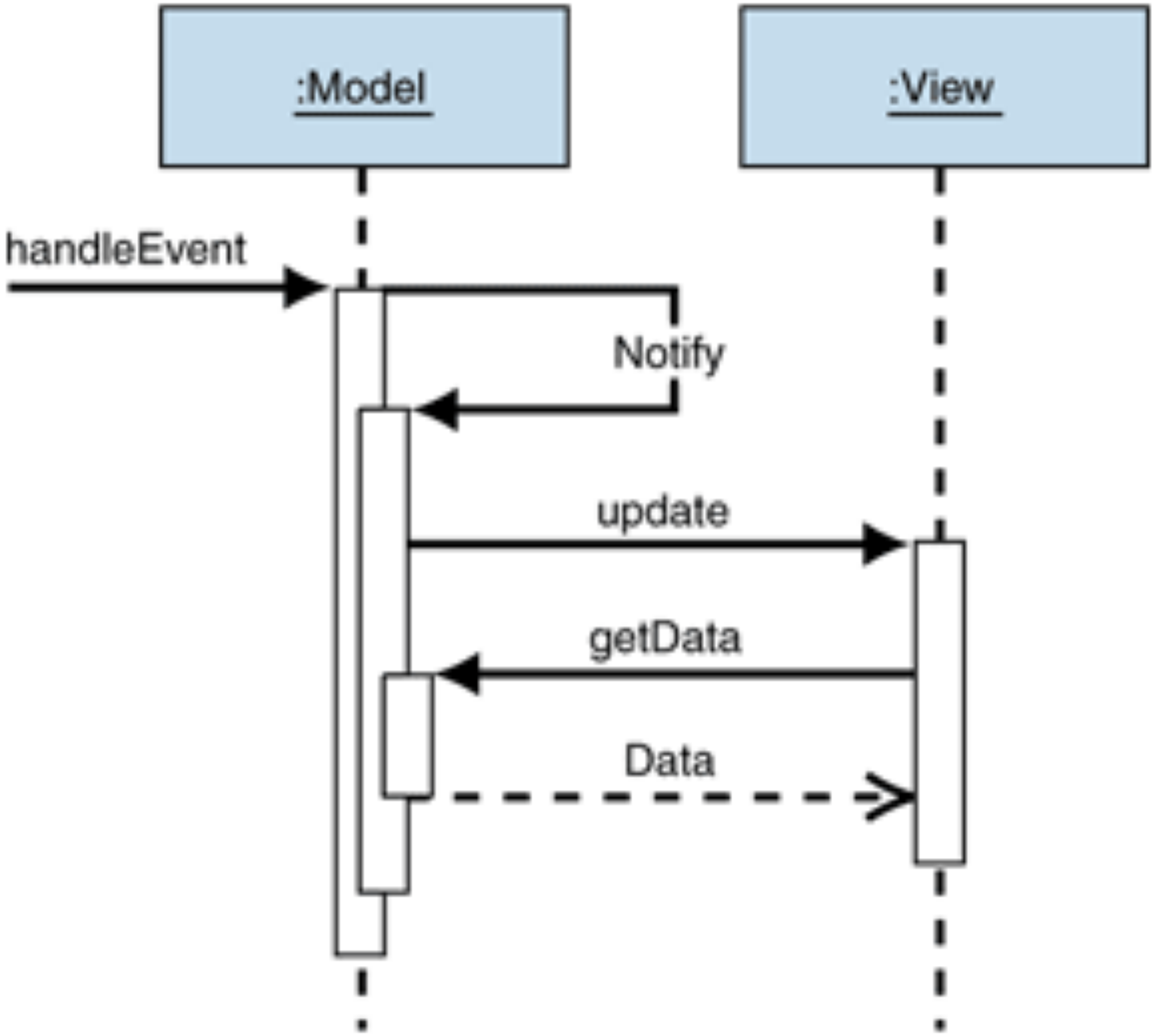
MVC



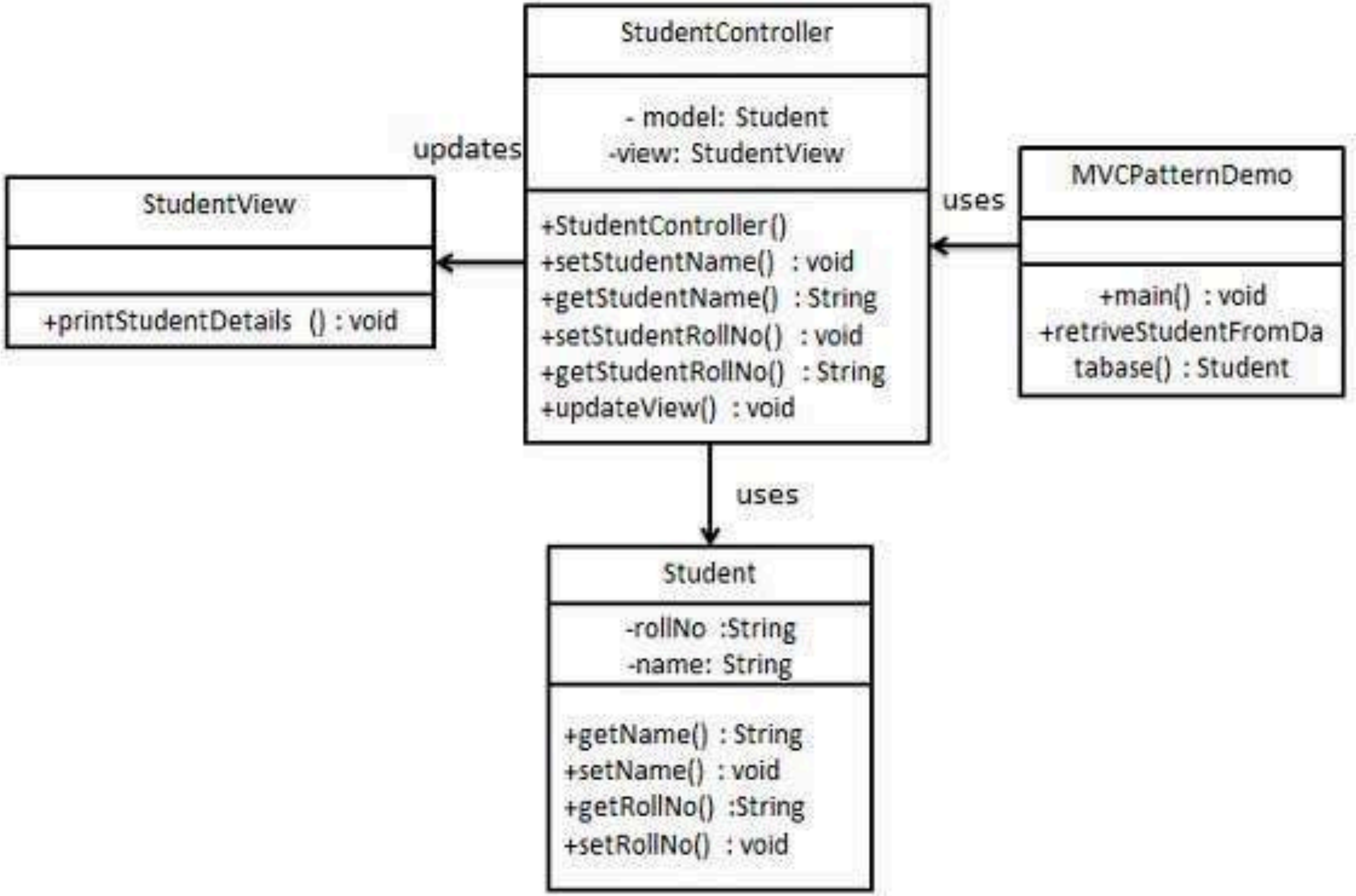
Behavior of the passive model



Using Observer to decouple the model from the view in the active model



Behavior of the active model



StudentController

- model: Student
-view: StudentView

+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void

StudentView

+printStudentDetails () : void

MVCPatternDemo

+main() : void
+retriveStudentFromDatabase() : Student

Student

-rollNo :String
-name: String

+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void

MVC

Model

```
public class Student {
   private String rollNo;
   private String name;

   public String getRollNo() {
      return rollNo;
   }

   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

View

```
•  public class StudentView {
•      public void printStudentDetails(String studentName, String studentRollNo){
•          System.out.println("Student: ");
•          System.out.println("Name: " + studentName);
•          System.out.println("Roll No: " + studentRollNo);
•      }
•  }
```

Controller

```
public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view){
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name){
        model.setName(name);
    }

    public String getStudentName(){
        return model.getName();
    }

    public void setStudentRollNo(String rollNo){
        model.setRollNo(rollNo);
    }

    public String getStudentRollNo(){
        return model.getRollNo();
    }

    public void updateView(){
        view.printStudentDetails(model.getName(), model.getRollNo());
    }
}
```

MVCPatternDemo

```
• public class MVCPatternDemo {
•    public static void main(String[] args) {

•       //fetch student record based on his roll no from the database
•       Student model  = retriveStudentFromDatabase();

•       //Create a view : to write student details on console
•       StudentView view = new StudentView();

•       StudentController controller = new StudentController(model, view);

•       controller.updateView();
•       //update model data
•       controller.setStudentName("John");

•       controller.updateView();
•    }

•    private static Student retriveStudentFromDatabase(){
•       Student student = new Student();
•       student.setName("Robert");
•       student.setRollNo("10");
•       return student;
•    }
• }
```

Multilayer

https://github.com/sunfan314/SE-DemoV2