PATTERNS FOR SOFTWARE ARCHITECTURE

BY PANKAJ KAMTHAN

1. INTRODUCTION

> [The] statement, "Those who cannot remember the past are condemned to repeat it," is only half true. The past also includes **successful histories**.
> — Barry Boehm

The reliance on the knowledge garnered from past experience is important for any development [Booch, 2007; Kamthan, 2008]. If software architecture is about **topology**, a pattern is means of creating 'high-quality' topologies **effectively**.

This document explores the notion of a pattern and its applications in software architecture.

CONVENTION/NOTATION

In this document, a **pattern for software architecture**, **software architecture pattern**, and **architectural pattern** are considered synonymous (and therefore interchangeable).

A pattern is usually referred to by its name. To distinguish a pattern from surrounding text, its name is usually presented in some unique manner. In this document, the name of a pattern is expressed in **uppercase** to distinguish it from surrounding text.

2. DEFINITION OF PATTERN

There are a number of definitions of a pattern, including the following:

Definition [Pattern]. An **empirically proven solution** to a **recurring problem** in a **particular context**.

The previous definition leads to the following:

Definition [Software Architecture Pattern]. An empirically proven solution to a recurring architectural problem in a particular architectural context.

3. THE INTERRELATIONSHIP BETWEEN THEORY AND PRACTICE

Theory without practice is empty; practice without theory is blind.
— John Dewey

There is nothing so practical as a good theory.
— Kurt Lewin

In theory, there is no difference between theory and practice. But, in practice, there is.
— J. L. A. van de Snepscheut

It has been pointed out [Falessi, Babar, Cantone, Kruchten, 2010] that "it is necessary to have **significant experience** as an **implementationist** to become a good **abstractionist**".

In other words, **theory**, especially in (software) engineering, is often motivated by **practice**. For example, many concepts have been inspired and introduced in programming languages [Sebesta, 2012, Section 1.1] as a result of developing non-trivial applications.

EXPERIENTIAL KNOWLEDGE AND PATTERNS

There can be no doubt that all our knowledge begins with experience.
— Immanuel Kant

Patterns are nothing if they don't capture expertise.
— John Vlissides

There are different kinds of **experiential knowledge** for software architecture [Babar, Dingsøyr, Lago, Vliet, 2009, Section 2.2; Capilla, Jansen, Tang, Avgeriou, Babar, 2016]. The interest is especially in **time-invariant** knowledge, and patterns are one such type of knowledge [Kamthan, Fancott, 2011; Fehling, Leymann, Retter, Schupeck, Arbitter, 2014].

There are patterns for, essentially, **all** aspects of software engineering, as illustrated in Figure 1.

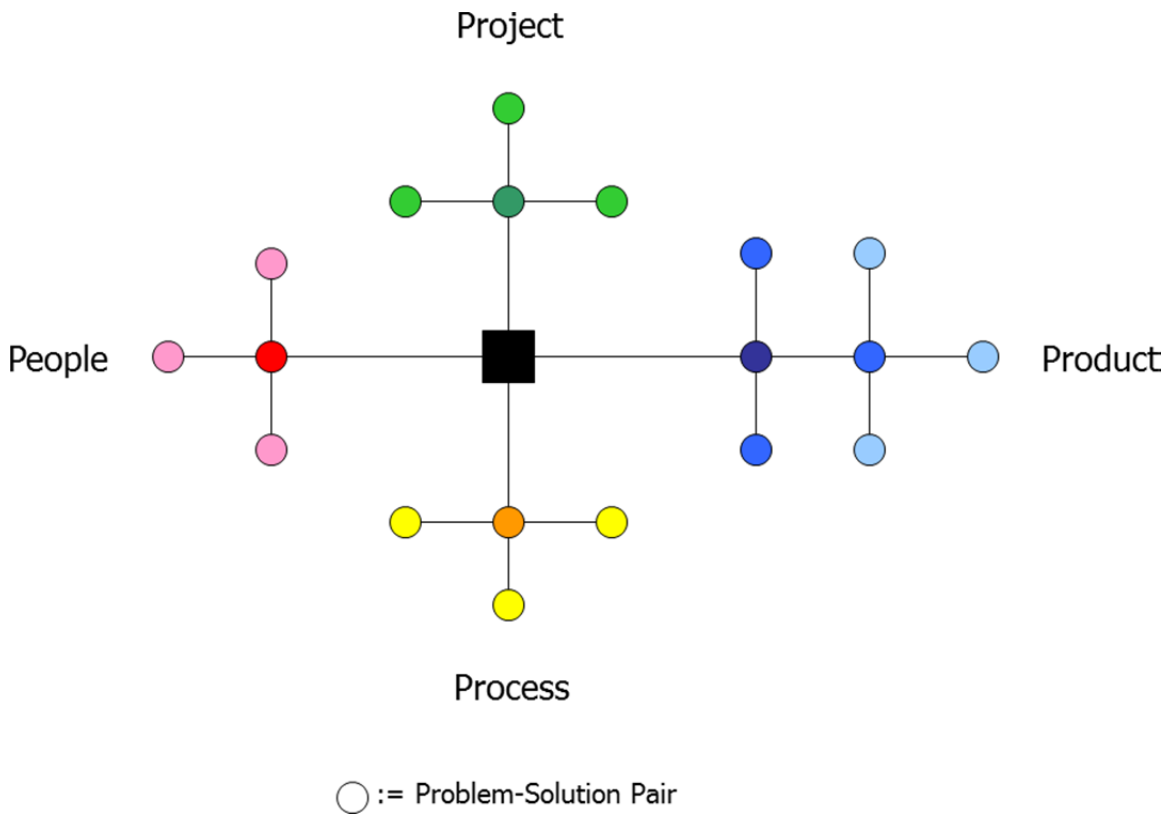


Figure 1. The discipline of software engineering from the perspective of patterns at different scales.

4. ON TERMINOLOGY

The notion of pattern is related to number of other, similar but **not same**, concepts. This has led to **better understanding** of the notion of pattern [Vogel, Arnold, Chughtai, Kehrer, 2011, Section 6.3], but has also led to **misunderstandings**.

4.1. PATTERN VERSUS 'X' FOR SOFTWARE ARCHITECTURE: "NOT EVEN WRONG"

It is not only not right, it is not even wrong.
— Wolfgang Pauli

The **synonymization** (and, thereby, **obfuscation**) of the notion of pattern with other concepts seems to be **prevalent** in software engineering, and neither the books [Larman, 2005], nor prominent organizations, are immune.

For example, the following is mentioned at **Microsoft Developer Network (MSDN)**[1]:

> "An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems."

This is not even nonsense.

4.2. THE RELATIONSHIP OF PATTERN TO OTHER, RELATED, CONCEPTS

The **candidate** concepts included here are:

- Decision
- Tactic
- Principle
- Guideline
- Style
- Framework

The concepts not discussed here, but of interest, include **anti-pattern, application programming interface (API), and framework**.

PATTERN AND DECISION

In any moment of decision the best thing you can do is the right thing, the next best thing is the wrong thing, and the worst thing you can do is nothing.
— Theodore Roosevelt

The life of a software architect is a long (and sometimes painful) **succession** of **suboptimal decisions** made partly in the dark.
— Grady Booch

It could be said that software architecture is a **product** (or outcome) of a number of **decisions** [Tyree, Akerman, 2005; Babar, Dingsøyr, Lago, Vliet, 2009, Section 3.4; Vliet, Tang, 2016]. Indeed, one of the definitions of software architecture is given in terms of decisions [Medvidović, Dashofy, Taylor, 2007].

Figure 2 illustrates the placement of an architectural decision relative to several other elements.

[1] URL: http://msdn.microsoft.com/en-us/library/ee658117.aspx .

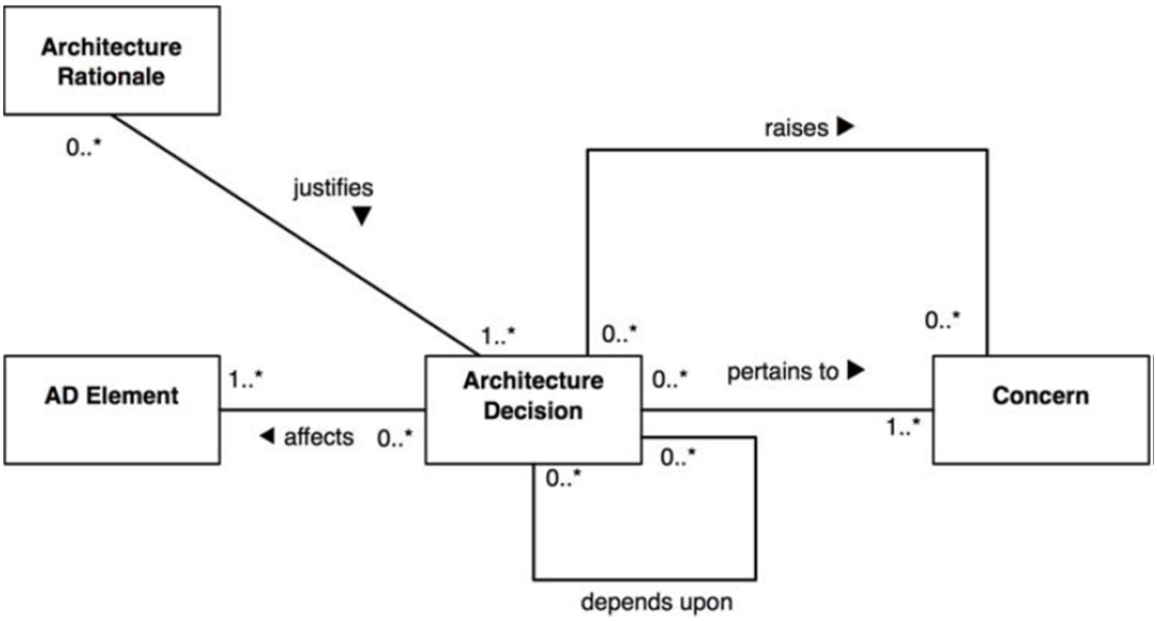


Figure 2. The conceptual model of architectural decisions and rationale. (Source: [ISO/IEC/IEEE, 2011].)

A pattern is a **pre-defined decision**. Then, by reference, a software architecture pattern is a **pre-defined architectural decision**.

Therefore, a commitment to a pattern entails the following: it is used **as-is**, or **not at all**.

For example, it is **not possible** to **change** the problem, or the solution of a pattern, to suit a situation. This is relevant to the **selection** of that pattern.

PATTERN VERSUS TACTIC

Definition [Tactic] [Bass, Clements, Kazman, 2003]. A **design decision** that influences the control of a **quality attribute** response.

A tactic is a decision at a level **higher** than a pattern. A software architect usually selects a pattern, or a collection of patterns, to **realize** one or more tactics. In other words, **patterns follow (succeed) tactics**.

A **difference** between pattern and tactic is that the notion of pattern has been applied to many disciplines, not only software architecture.

There are various **collections of tactics** for software architecture, available in different avenues, including print publishing: [Bass, Clements, Kazman, 2003, Chapter 5; Sangwan, 2015, Chapter 4].

A collection of tactics is an **architectural strategy**.

PATTERN VERSUS PRINCIPLE

A principle is **relatively more general** than a pattern. A principle usually does **not** outline the problem or the solution to any specific problem. It may also be devoid of any rationale of its existence or use.

The solution of a pattern is often based on established principles, **implicitly or explicitly**.

For example, the MODEL-VIEW-CONTROLLER pattern, which is an architectural pattern, is based on the principle of **separation of concerns** [Ghezzi, Jazayeri, Mandrioli, 2003].

PATTERN VERSUS GUIDELINE

A guideline is **relatively more general** than a pattern. A guideline usually suggests a solution **without** outlining any problem. It may also be devoid of any **rationale** of its existence or use.

PATTERN VERSUS STYLE

The notion of **architectural style** has been part of the discipline of software architecture since its inception [Garlan, Shaw, 1994].

In [Bass, Clements, Kazman, 2003, Section 2.3; Fairbanks, 2010, Section 14.4], architectural pattern and architectural style are treated as **synonymous**. However, an architectural style is **not structured** like an architectural pattern. The focus in a style is also **more on the solution than on the problem**.

There are certain architectural styles that have been **'recast'** as architectural patterns [Fairbanks, 2010, Section 14.5]. For example, such styles include Client/Server, N-Tier, and 3-Tier, among others.

PATTERN AND FRAMEWORK

Definition [Framework] [ISO/IEC/IEEE, 2010].
(1) A **reusable** design (models and/or code) that can be refined (specialized) and extended to provide some portion of the overall functionality of many applications.
(2) A **partially completed software subsystem** that can be extended.

Definition [Architecture Framework] [ISO/IEC/IEEE, 2010]. [A collection of] conventions, principles, and practices for the description of architectures established within a specific domain of application or community of stakeholders.

A pattern in itself is not a framework, although a pattern could be used to construct a framework [Christensen, 2010, Chapter 32].

5. CHARACTERISTICS OF A PATTERN

There are a number of characteristics that make a pattern **unique**. These characteristics are intimately related to the structure of the description of a pattern.

5.1. STRUCTURE OF THE DESCRIPTION OF A PATTERN

It is the structure of the description of a pattern [Buschmann, Henney, Schmidt, 2007b; Kamthan, 2011; Meszaros, Doble, 1998] that makes it **unique** among other kinds of experiential knowledge.

If patterns can lead to software systems of 'high-quality', then the **quality** of patterns that are candidate for the software architecture is important. This is because, in case of changes to requirements, it is desirable that the software architecture remains **unchanged** [Nuseibeh, 2001].

The **quality of the description of a pattern** is important in the **selection** of that pattern [Zhang, Budgen, 2012]. The structure of the description of a pattern is an integral aspect of the quality of the description of that pattern. For the patterns that are currently available, there is, unfortunately, stark variation in the quality of their descriptions [Kamthan, 2009].

There is currently no 'standard' for the structure of the description of a pattern. However, it is expected [Meszaros, Doble, 1998] that a proper description of a pattern has (at least) the following elements:

- Name
- Context
- Problem
- Forces
- Solution
- Rationale
- Consequences
- Example

For example, one type of **structure of the description of an architectural pattern** is given in [Gomaa, 2011, Appendix A].

5.2. RELATIONSHIPS BETWEEN PATTERNS

The patterns in given collection, or even otherwise, can be related in different ways [Buschmann, Henney, Schmidt, 2007b]. The patterns could **complement** each other, they could **compete** with each other, they could be relatively more **general/specific** with respect to each other, they could be **variations** of each other, and so on. These relationships are apparent in a **pattern language** [Buschmann, Henney, Schmidt, 2007b], a special kind of collection of related patterns.

6. PATTERNS AND SOFTWARE ARCHITECTURE

There are two important aspects in patterns for software architecture, namely patterns and concerns, and patterns and views.

6.1. PATTERNS AND CONCERNS (QUALITY ATTRIBUTES)

There are a number of concerns that need to be addressed by the software architecture. These concerns include, but are **not** limited to, **quality attributes**. There is an **intrinsic** relationship between patterns and concerns, through tactics and (software) quality.

SECURITY

The **protection of assets** is a concern that can be related to **security**. Next, security can be **decomposed** into a number of quality attributes (such as **authentication, authorization, integrity, confidentiality, and auditability**).

These quality attributes can then be related to tactics, and again those tactics can be related to patterns [Yoder, Barcalow, 1997; Babar, Wang, Gorton, 2005; Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, Sommerlad, 2006].

For example, for a File Transfer Protocol (FTP) server application, concerns and patterns can be related, as shown in Table 1.

FTP Server			
Concern	**Tactics**	**Quality Attribute**	**Patterns**
Security Management	• Check passwords • Have time-outs for spoofing • Track security violations • Consider security levels • Consider failure actions • . . .	Authentication	SINGLE ACCESS POINT
		Confidentiality	LIMITED VIEW
		Auditability	CHECK POINT

Table 1. A concern-pattern connection.

MAINTAINABILITY

For example, for an income tax application, concerns and patterns can be related, as shown in Table 2.

Software Type	Income Tax
Stakeholder	Government, Accountant, Tax Payer
Goal	Maximize Electronic Submission of Tax Returns (or Use of Software by Tax Payers)
Concern	• Social: Accommodate New Tax Payers Each Year • Technical: Avoid Propagation of Change
Quality Attribute	[ISO/IEC, 2001] Maintainability: Stability and Testability
Tactics	[Bass, Clements, Kazman, 2003] • Hide Information • Maintain Existing Interfaces • Restrict Communication Paths
Patterns	[Fowler, Rice, Foemmel, Hieatt, Mee, Stafford, 2003] • MODEL-VIEW-CONTROLLER • TRANSFORM VIEW

Table 2. A concern-pattern connection.

6.1.1. PATTERNS AND TRIZ

теория решения изобретательских задач Teoriya Resheniya Izobretatelskikh Zadatch

The solution of a pattern is a result of optimally **balancing** a number of forces.

It is **unlikely** that any given pattern for software architecture satisfies **all concerns equally**, especially if the concerns are quality attributes. This is because of the **nature** of relationships among the quality attributes.

In other words, there is **no 'perfect' pattern for software architecture**.

Therefore, the **selection** of a pattern for software architecture, inevitably, leads to **trade-offs** among the candidate concerns [Monson-Haefel, 2009].

6.2. PATTERNS AND VIEWS

A concern can lead to the selection of a particular view, and a view can lead to the selection of **one or more** patterns for software architecture [Avgeriou, Zdun, 2005].

7. EXAMPLES OF PATTERNS FOR SOFTWARE ARCHITECTURE

There is currently **no single collection or index of patterns** for software architecture. The **Handbook of Software Architecture**[2] is an ongoing effort in that direction.

There are various **collections of patterns** for software architecture, available in different avenues, including print publishing: [Buschmann, Meunier, Rohnert, Sommerlad, Stal, 1996; Fowler, Rice, Foemmel, Hieatt, Mee, Stafford, 2003; Dyson, Longshaw, 2004; **Gomaa, 2004, Appendix B**; Hohpe, Woolf, 2004; Avgeriou, Zdun, 2005; Lethbridge, Laganière, 2005; **Buschmann, Henney, Schmidt, 2007a**; Fairbanks, 2010, Chapter 14; Dooley, 2011, Chapter 5; **Gomaa, 2011, Appendix A**; Sangwan, 2015, Chapter 4].

The patterns discussed in the following include LAYERS, CLIENT-SERVER, SHARED REPOSITORY, MODEL-VIEW-CONTROLLER, and PIPES AND FILTERS. The selection is based on their availability, broad use, and the diversity of views that they correspond to.

[2] URL: http://www.handbookofsoftwarearchitecture.com/ .

The patterns **not** discussed in this document, but are of interest nevertheless, are BROKER, PUBLISH-SUBSCRIBE, and VIRTUAL MACHINE.

The coverage of the patterns in the following is **intentionally minimal**.

7.1. THE LAYERS PATTERN

View: Layered View.

Epigrammatic Description: The LAYERS pattern is useful for separating higher-level from lower-level responsibilities. Figure 3 provides an illustration.

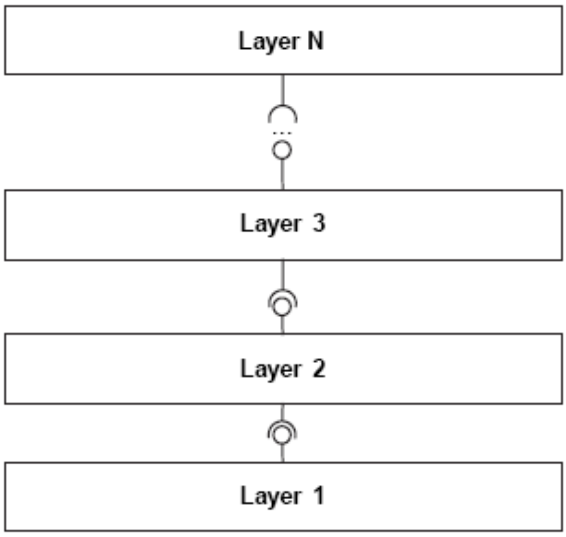


Figure 3. The LAYERS pattern. (Source: [Avgeriou, Zdun, 2005].)

The system is structured into **layers**: each layer provides a set of services to the layer above and uses the services of the layer below.

In **each** layer, all constituent components[3] work at the **same level of abstraction**, and can interact through connectors[4]. However, between two adjacent layers a clearly defined **interface** is provided.

[3] For the sake of this document, a **component** is defined as "the principal computational element and data store that execute in a system" [CMU/SEI Software Architecture Glossary].
[4] For the sake of this document, a **connector** is defined as "a runtime pathway of interaction between two or more components" [CMU/SEI Software Architecture Glossary].

A higher-level layer is allowed to access only the lower-level layer immediately below. In general, **bypassing a layer is not permitted**.

The **Unified Modeling Language (UML)** [Booch, Jacobson, Rumbaugh, 2005] **does not** have a built-in primitive for layering [Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010, Section A.2.4]. The UML Package Diagram could be used to model a layer, a stereotype **`<<layer>>`** could be used to denote a layer, and a stereotype could be used for the direction of dependency of the layer. Figure 4 provides an illustration.

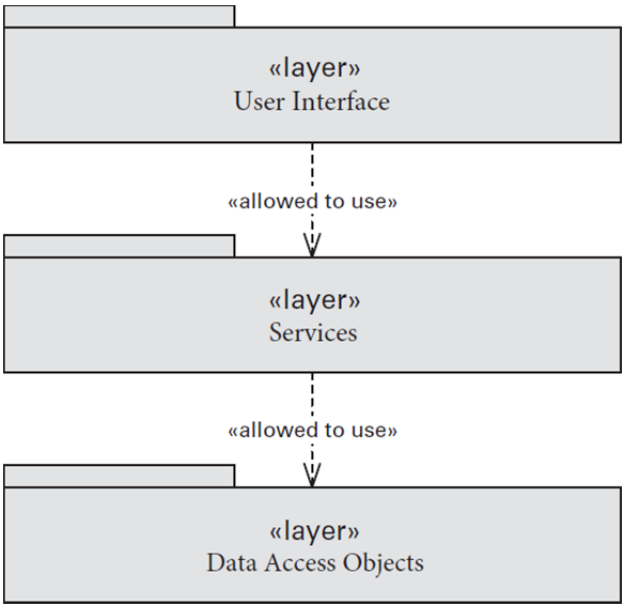


Figure 4. The use of UML Package Diagram with stereotypes to represent the LAYERS pattern. (Source: [Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010].)

Example: The software architecture of an **operating system** has a number of objectives, including (1) to centralize control of the limited hardware resources, and (2) to protect users from each other [Dooley, 2011, Chapter 5]. The LAYERS pattern enables accomplishing both of these things, and is therefore common in basic operating system services [Sangwan, 2015, Section 4.3.1]. For instance, the user interface to a command processor (that is, shell) is layered on the top of the shell, which, in turn, is layered on the top of input/output services, which, again in turn, are layered on the top of the kernel. Figure 5 provides an illustration.

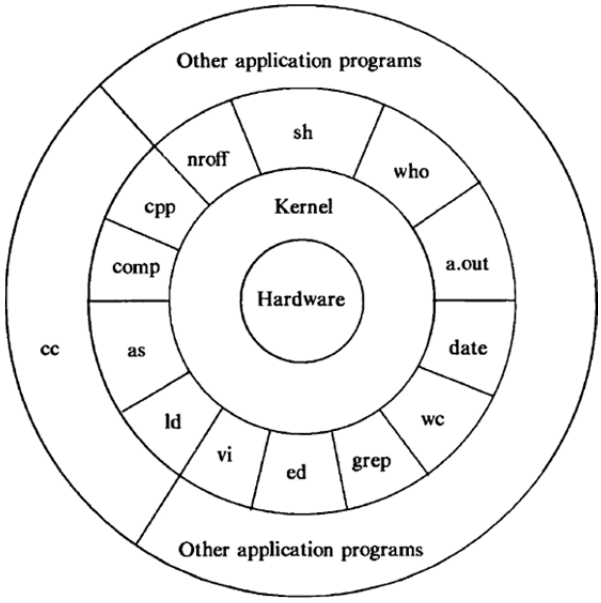


Figure 5. The use of the LAYERS pattern in the UNIX system architecture. (Source: [Bach, 1986].)

Example: The LAYERS pattern is common in the implementation of **communication protocols** [Qin, Xing, Zheng, 2009, Section 2.5; Dooley, 2011, Chapter 5; Sangwan, 2015, Section 4.3.1]. The **Open Systems Interconnection (OSI)** model is "a **conceptual model** that characterizes and standardizes the internal functions of a communication system by partitioning it into seven abstraction layers" [Wikipedia]. Figure 6 provides an illustration.

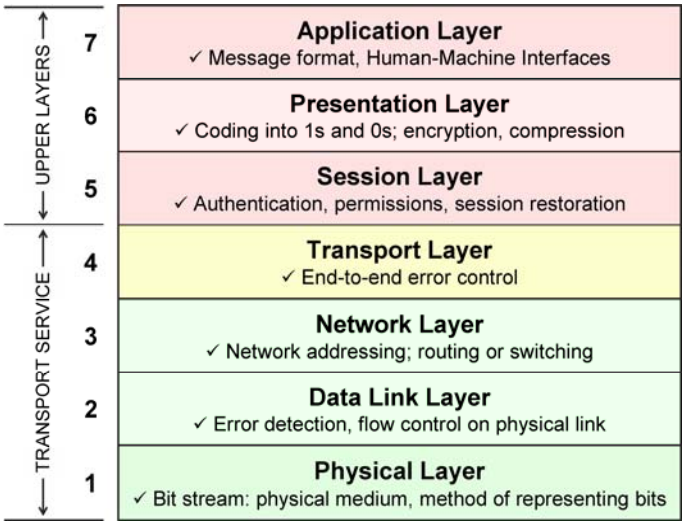


Figure 6. The use of the LAYERS pattern in the OSI model protocol stack. (Source: Cisco Systems.)

7.2. THE CLIENT-SERVER PATTERN

View: Component Interaction View.

Epigrammatic Description: The CLIENT-SERVER pattern distinguishes two kinds of components: clients and servers.

1. **Client.** The client requests information or services from a server. To do so it needs the **address** of the server and knowledge of the **interface** of the server.

2. **Server.** The server responds to the requests of the client, and processes each client request on its own. It **does not know** about the address of the client before the interaction takes place.

The type of tasks for which a client or a server is responsible can either be **individual or collective**.

The clients are optimized for their application task; the servers are optimized for serving multiple clients. Also, both client and server must implement collective tasks, such as security management, transaction management, and systems management.

A client or server can be **thick or thin**, depending on the **number** and **type** of **responsibilities**. The accepted combinations are:

- **Thin Client.**
  - Example: Web User Agent.

The support for **Rich Internet Applications (RIA)**, such as Social Web Applications, has increased the 'thicknesses' of clients. This is because a client has to support a myriad of technologies (for markup, for scripting, for style sheet processing, and so on).

- **Thick Client.**
  - Example: Typical Desktop Client-Server Application.

Let the set of responsibilities be some (irreducible) constant. The **Tesler's Law of the Conservation of Complexity** (or Tesler's Law for short) [Saffer, 2006] could be used to explain the difference between thin and thick clients. Figure 7 provides an illustration.

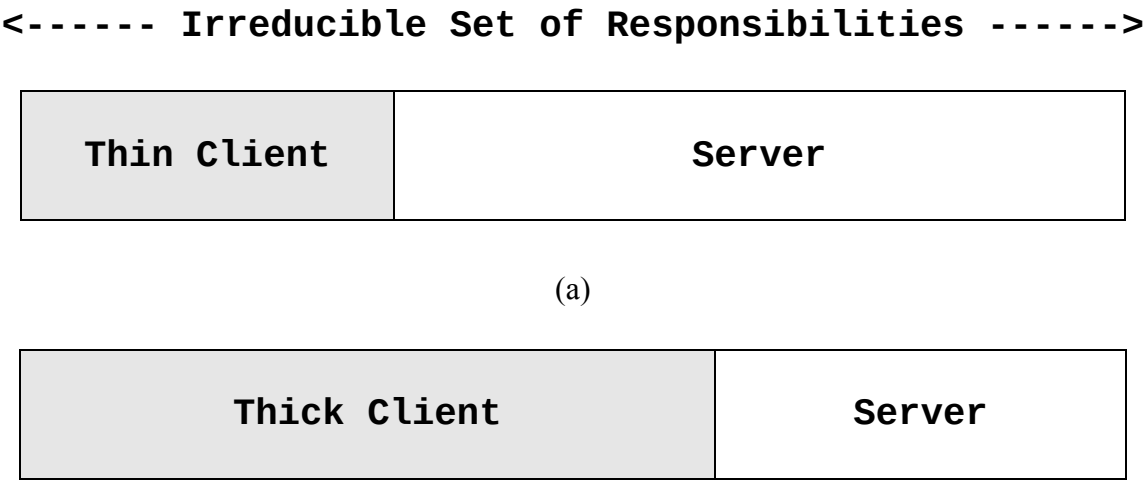


Figure 7. (a) A thin client takes upon relatively fewer number of responsibilities; (b) A thick client takes upon relatively more responsibilities.

Example: The CLIENT-SERVER pattern can be used to build arbitrarily complex **(n-tier) architectures** by introducing multiple client-server relationships: a server can itself act as a client to other servers. Figure 8 shows an example of **3-tier architecture**, which is common in Web Applications.

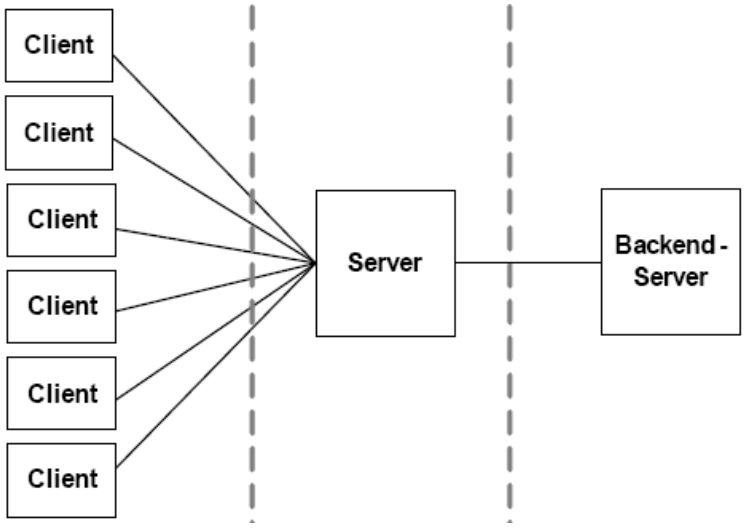


Figure 8. A 3-tier architecture that consists of: (1) a **client tier**, responsible for the presentation of data, receiving user events, and controlling the user interface, (2) an **application logic tier**, responsible for implementing the application logic (also known as business logic), and (3) a **backend tier**, responsible for providing backend services, such as data storage in a database or access to a legacy system.

Example: The **X Window System**[5] is a graphical windowing system for UNIX and Linux operating systems. (It is also available for Apple Macintosh and Microsoft Windows as an add-on windowing system.) The X Window System uses the CLIENT-SERVER pattern, where both the client programs and the server typically reside on the same computer. The examples of client programs include **`xterm`** (a windowed terminal program that provides a user interface to a command processor in UNIX) and **`xdm`** (the X Window System display manager).

Example: The CLIENT-SERVER pattern is common in software systems that require **multiple clients** accessing a server simultaneously. For instance, Cloud Applications [Marinescu, 2013, Section 4.3], Domain Name System (DNS), multimedia library systems, and Web Applications, belong to this category.

Remark: There is a **difference between a layer and a tier**. A layer is about **structure (or space)**, whereas a tier is about **behavior (or time)**. An n-layer is about how components are organized (or grouped). An n-tier is about how the processes corresponding to the components are run.

Remark: It is possible to **combine** the CLIENT-SERVER pattern with the LAYERS pattern in order to design a system where the client and the server components are layered individually.

Remark: The PEER-TO-PEER pattern is closely related to the CLIENT-SERVER pattern. The difference being that each component has identical responsibilities (and, therefore, can act as **both** a client and a server).

[5] URL: http://www.xfree86.org/ .

7.3. THE SHARED REPOSITORY PATTERN

View: Data-Centered View.

Epigrammatic Description: In the SHARED REPOSITORY pattern, **one** component of the system is used as a **central data store**, accessed by **all** other independent components. This SHARED REPOSITORY offers suitable means for accessing the data such as a query API or language. The SHARED REPOSITORY might also introduce transaction mechanisms. Figure 9 provides an illustration.

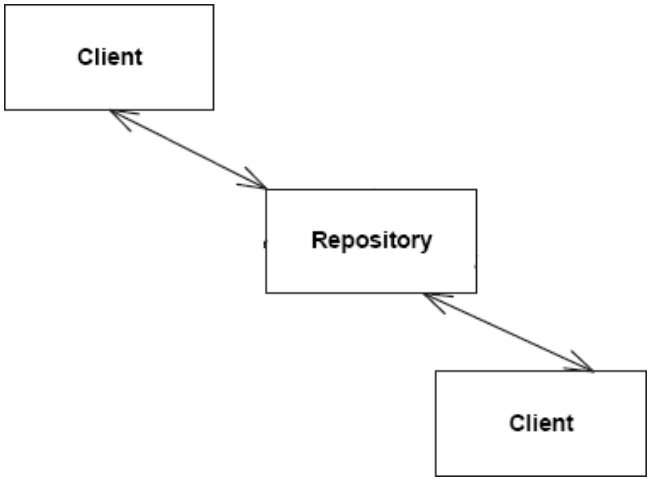


Figure 9. The SHARED REPOSITORY pattern. (Source: [Avgeriou, Zdun, 2005].)

The SHARED REPOSITORY must have a number of properties:

- It must be **scalable** to meet the clients' requirements, that is, it should be possible to add more data and add more clients to the existing architecture.
- It must ensure data **consistency**.
- It must handle problems of **resource contention**, for example, by **locking accessed data**.

Example: This pattern is common in software systems that act upon a **common data set**. For instance, a **computer-aided software engineering (CASE) toolkit** may have a project repository and several clients, such as a **program editor, source code generator, modeler, and metrics analyzer**.

7.4. THE MODEL-VIEW-CONTROLLER PATTERN

View: User Interaction View

Epigrammatic Description: There **representation**, **presentation** (displaying or, more generally, rendering), and **processing** of information are semantically **different** things. This philosophy has motivated the MODEL-VIEW-CONTROLLER pattern.

The system is divided into three different parts, and each part is assigned a different responsibility:

1. **Model.** There is a **model** that encapsulates some application data and the logic that manipulates that data, **independently** of the user interfaces.

2. **View.** There is a **view** that displays a specific portion of the data to the user. The 'user' may be a **human** or a **machine**.

3. **Controller.** There is a **controller** associated with each view that receives user input and translates it into a request to the **model**.

Figure 10 provides an illustration.

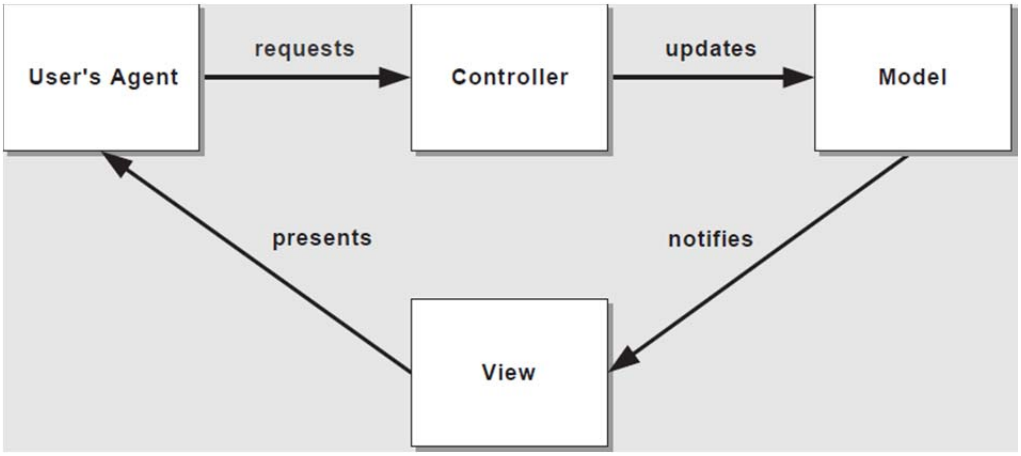


Figure 10. The Box-and-Lines Diagram of the MODEL-VIEW-CONTROLLER pattern. (Source: [Brambilla, Fraternali, 2015, Section 10.2.1].)

There can be **one or more views**, and **one or more controllers**. In general, there is only **one model**.

The user interface is composed of views and controllers. The users interact strictly through the views and their controllers, **independently** of the model. The model in turn **notifies** all different user interfaces about updates.

The three different parts can be modeled as components and connectors. Figure 11 provides an illustration.

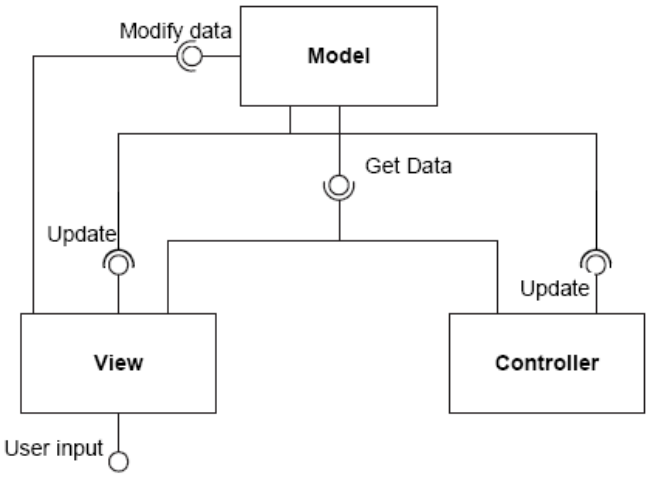


Figure 11. The MODEL-VIEW-CONTROLLER pattern. (Source: [Avgeriou, Zdun, 2005].)

The MODEL-VIEW-CONTROLLER pattern has implications towards **maintainability** [Dooley, 2011, Chapter 5; Crookshanks, 2014, Chapter 7]:

- The view can be changed without impacting the model, as the view is **separated** from the model and the model does **not** depend on the view.

- The model and controller can be tested without a view, through unit tests and mock controllers.

MODEL-VIEW-CONTROLLER INCARNATIONS

There are **'incarnations'** of the MODEL-VIEW-CONTROLLER pattern[6] [Gamma, Helm, Johnson, Vlissides, 1995; Fowler, Rice, Foemmel, Hieatt, Mee, Stafford, 2003; Buschmann, Henney, Schmidt, 2007; Plakalović, Simić, 2010].

These variations are **motivated** by one or more of the following factors:

[6] URL: http://martinfowler.com/eaaCatalog/ .

- **Dynamic Delivery/Personalizability.** To invoke **different user interface styles** in a Web Application, there may be a need to have **multiple controllers**.

- **Minimize Redundancy.** An increase in the number of controllers can lead to **redundancy** (in form of duplicated source code across controllers) that, in turn, is **prohibitive to maintainability**. (In such a case, the common logic could be extracted via the APPLICATION CONTROLLER pattern.)

- **Implementability.** There is a need for programming language-specific customizations. For instance, **Java**-specific customization is known as **Model 2** [Seshadri, 1999]: JavaBeans (model), JavaServer Pages (view) and Java Servlets (controller). There are implementations in **ASP.NET, F#, Perl, PHP: Hypertext Preprocessor (PHP)**, and so on.

  Definition [Framework] [ISO/IEC/IEEE, 2010].
  (1) [A] reusable design (models and/or code) that can be refined (specialized) and extended to provide some portion of the overall functionality of **many applications**.
  (2) [A collection of] conventions, principles and practices for the description of architectures established within a specific domain of application.

  There are also certain frameworks that implement this pattern. These frameworks include **Apache Jakarta Struts, Spring MVC Framework**, and **AngularJS**.

  It could be noted that certain **JavaScript** implementations may **not** clearly separate controller from the view; in fact, certain frameworks, such as **Backbone.js**, merge the two. Therefore, such frameworks are labeled as supporting the **MV*** pattern (meaning, there is likely to be a model and a view, but a distinct controller might not be present).

Example: This pattern is common in software systems that need to provide multiple user interfaces for the same data [Crookshanks, 2014, Chapter 7]. For instance, these can include **Mobile Applications** [Plakalović, Simić, 2010] and **Web Applications** [Kamthan, 2008; Brambilla, Fraternali, 2015, Section 10.2.1].

Example: This pattern has been used in the software architecture of **Emergency Remote Pre-Hospital Assistance (ERPHA)**, a Mobile Application for assisting paramedics while attending victims of traffic accidents [Lavariega, Avila, Gómez-Martínez, 2015].

Remark: The PRESENTATION-ABSTRACTION-CONTROL pattern [Bass, Clements, Kazman, 2003; Avgeriou, Zdun, 2005; Sangwan, 2015] is closely related to the MODEL-VIEW-CONTROLLER pattern. A **comparison** between the two patterns is given in [Plakalović, Simić, 2010].

7.5. THE PIPES-AND-FILTERS PATTERN

View: Data Flow View.

Epigrammatic Description: A pipe or a filter is about data processing, specifically, transformation.

In the PIPES AND FILTERS pattern, a **complex task** is divided into several **sequential subtasks**. This division eases **refactoring** and enables **reuse** of filters [Qin, Xing, Zheng, 2009, Section 2.2; Homer, Sharp, Brader, Narumoto, Swanson, 2014, Page 100]. Figure 12 provides an illustration. There is a flow of data from left to right.

General:

Specific:

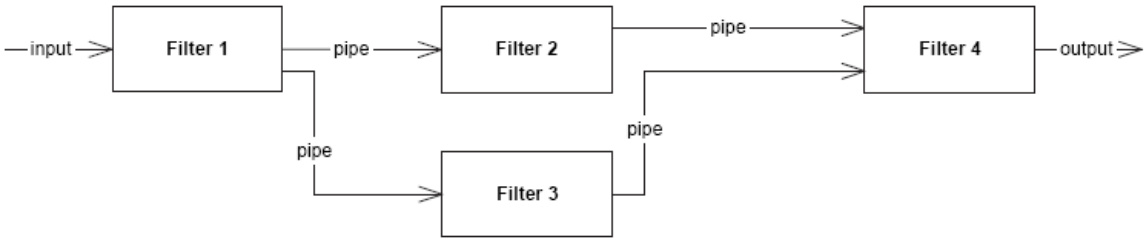


Figure 12. The PIPES AND FILTERS pattern. (Source: [Avgeriou, Zdun, 2005].)

SUB-TASKS AND FILTERS

Each sub-task is implemented by a separate, **independent** component, a **filter**, which handles **only** this task. The filters have a number of inputs and a number of outputs and they are **connected** flexibly using pipes, but they are **not aware** of the **identity** of adjacent filters. The **mutual independence** of filters means that **different combination** of filters can yield **different results**, as implied by Figure 12.

The performance of the pipeline (that is, the time it takes to process a single request) depends on the performance of slowest filter in the pipeline.

DATA AND PIPES

Each pipe realizes a **stream of data** between two components. Each filter consumes and delivers data **incrementally**. This maximizes the throughput of each individual filter since filters can potentially work in parallel. The pipes act as **data buffers** between adjacent filters.

Example: The **UNIX** operating system [Bach, 1986] and its operating systems, including **Linux**, have sophisticated **piping and filtering** mechanism via the use of one of the **shells**. For instance, the sequence

```
ls | grep '.sh' | sort > shlist.txt
```

does the following: lists the contents of the current directory, filters the output to only contain lines that contain `.sh`, sorts the resulting output lexicographically, and places the final output in `shlist.txt`.

Example: The PIPES AND FILTERS pattern is common in software systems that need a sequential transformation pipeline. For instance, a **programming language implementation system** (such as a **compiler**) is a realization of this pattern, as illustrated in Figure 13.

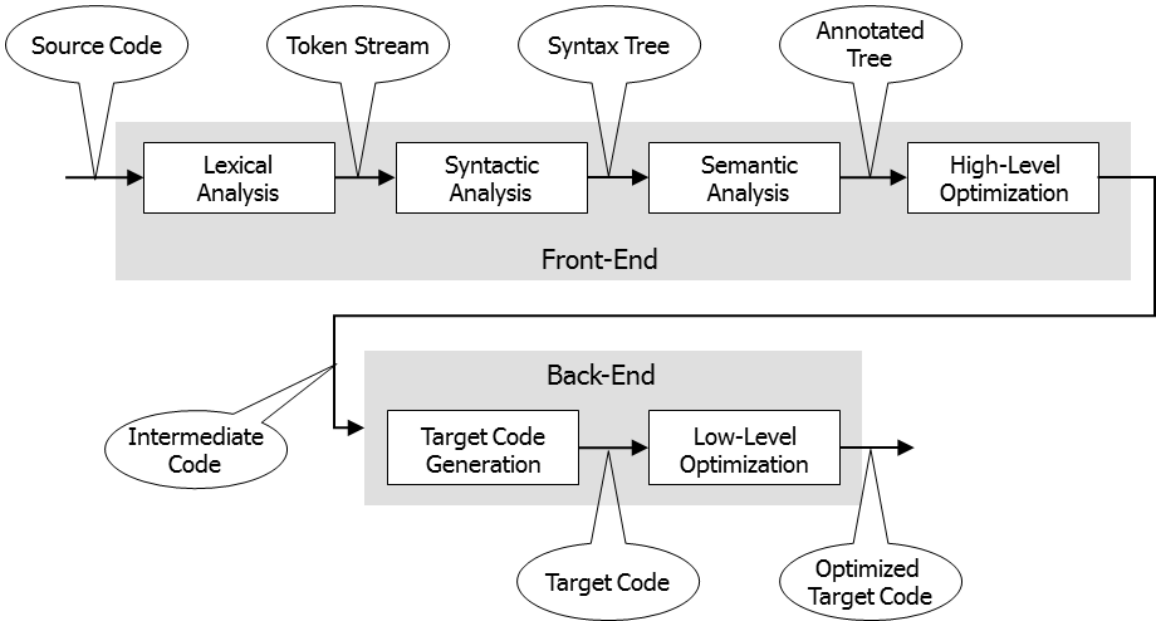
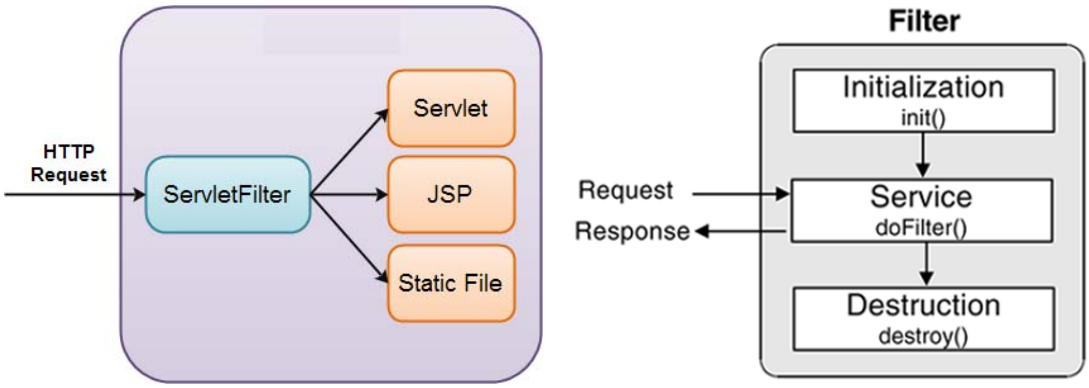


Figure 13. The process of program transformation is a realization of the PIPES AND FILTERS pattern.

Example: The **Java Servlet Application Programming Interface (API)** is part of the **Java Platform, Enterprise Edition (Java EE)**, and uses the PIPES AND FILTERS pattern. A **Java Servlet filter** takes an input, and may make use of one or more filters to produce different outputs, as shown in Figure 14. For instance,

$$\mathbf{I \rightarrow Filter\ 1 + Filter\ 2 \rightarrow O_1}$$
$$\mathbf{I \rightarrow Filter\ 2 + Filter\ 3 \rightarrow O_2}$$
$$\mathbf{I \rightarrow Filter\ 1 + Filter\ 3 \rightarrow O_3}$$



(a) (b)

Figure 14. (a) The data flow view of the architecture of a typical Java Servlet filter. (b) The Java Servlet filter life cycle of initialization, service, and destruction. (Source: Google Images.)

Example: There are a number of advantages of expressing non-binary, structured, data in the **Extensible Markup Language (XML)**, including the ease of transformation. In general, an **XML document transformation system** transformation is a realization of the PIPES AND FILTERS pattern.

Remark: In the PIPES AND FILTERS pattern, a filter acts like a **transducer** (input → one or more algorithmic transformations → output) [Dooley, 2011, Chapter 5].

Remark: The use of the PIPES AND FILTERS pattern is recommended if **little contextual information** needs to be maintained between the filter components and filters retain **no state between invocations**. However, due to **latency** in communication, there are **performance overheads** for transferring data in pipes and data transformations.

Remark: The use of the PIPES AND FILTERS pattern is **not** recommended for **interactive programs** (or software systems) that need to respond to events or interrupts.

Remark: The PIPES AND FILTERS pattern necessitates that **only two adjacent filters can share data** through their pipe, but **not non-adjacent filters**.

Remark: In **cloud computing and service-oriented computing**, the PIPES AND FILTERS pattern is used as part of **larger architectural patterns**, such as the MESSAGE-ORIENTED MIDDLEWARE pattern and the DISTRIBUTED APPLICATION pattern [Buyya, Vecchiola, Selvi, 2013, Section 2.4.3.2; Fehling, Leymann, Retter, Schupeck, Arbitter, 2014; Homer, Sharp, Brader, Narumoto, Swanson, 2014].

Remark: The PIPES AND FILTERS is an **alternative** to the LAYERS pattern, as well as to the SHARED REPOSITORIES pattern, **only if** data sharing between non-adjacent processing tasks is **not** needed.

ACKNOWLEDGEMENT

The inclusion of images from external sources is only for non-commercial educational purposes, and their use is hereby acknowledged.

REFERENCES

[Avgeriou, Zdun, 2005] Architectural Patterns Revisited - A Pattern Language. By P. Avgeriou, U. Zdun. The Tenth European Conference on Pattern Languages of Programs (EuroPLoP 2005). Irsee, Germany. July 6-10, 2005.

[Babar, Dingsøyr, Lago, Vliet, 2009] Software Architecture Knowledge Management: Theory and Practice. By M. A. Babar, T. Dingsøyr, P. Lago, H. van Vliet. Springer-Verlag. 2009.

[Babar, Wang, Gorton, 2005] Supporting Security Sensitive Architecture Design. By M. A. Babar, X. Wang, I. Gorton. The First International Conference on the Quality of Software Architectures (QoSA 2005). Erfurt, Germany. September, 20-22, 2005.

[Bach, 1986] The Design of the UNIX Operating System. By M. J. Bach. Prentice-Hall. 1986.

[Bass, Clements, Kazman, 2003] Software Architecture in Practice. By L. Bass, P. Clements, R. Kazman. Second Edition. Addison-Wesley. 2003.

[Booch, 2007] It Is What It Is Because It Was What It Was. By G. Booch. IEEE Software. Volume 24. Issue 1. 2007. Pages 14-15.

[Booch, Jacobson, Rumbaugh, 2005] The Unified Modeling Language Reference Manual. By G. Booch, I. Jacobson, J. Rumbaugh. Second Edition. Addison-Wesley. 2005.

[Brambilla, Fraternali, 2015] Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. By M. Brambilla, P. Fraternali. Morgan Kaufmann. 2015.

[Buschmann, Henney, Schmidt, 2007a] Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing. By F. Buschmann, K. Henney, D. C. Schmidt. John Wiley and Sons. 2007.

[Buschmann, Henney, Schmidt, 2007b] Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages. By F. Buschmann, K. Henney, D. C. Schmidt. John Wiley and Sons. 2007.

[Buschmann, Meunier, Rohnert, Sommerlad, Stal, 1996] Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. By F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. John Wiley and Sons. 1996.

[Buyya, Vecchiola, Selvi, 2013] Mastering Cloud Computing: Foundations and Applications Programming. By R. Buyya, C. Vecchiola, S. T. Selvi. Morgan Kaufmann. 2013.

[Capilla, Jansen, Tang, Avgeriou, Babar, 2016] 10 Years of Software Architecture Knowledge Management: Practice and Future. By R. Capilla, A. Jansen, A. Tang, P. Avgeriou, M. A. Babar. The Journal of Systems and Software. Volume 116. 2016. Pages 191-205.

[Christensen, 2010] Flexible, Reliable Software: Using Patterns and Agile Development. By H. B. Christensen. CRC Press. 2010.

[Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, Stafford, 2010] Documenting Software Architectures: Views and Beyond. By P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford. Second Edition. Addison-Wesley. 2010.

[Crookshanks, 2014] Practical Software Development Techniques: Tools and Techniques for Building Enterprise Software. By E. Crookshanks. Apress. 2014.

[Dooley, 2011] Software Development and Professional Practice. By J. Dooley. Apress. 2011.

[Dyson, Longshaw, 2004] Architecting Enterprise Solutions: Patterns for High-Capability Internet-Based Systems. By P. Dyson, A. Longshaw. John Wiley and Sons. 2004.

[Fairbanks, 2010] Just Enough Software Architecture: A Risk-Driven Approach. By G. Fairbanks. Marshall and Brainerd. 2010.

[Falessi, Babar, Cantone, Kruchten, 2010] Applying Empirical Software Engineering to Software Architecture: Challenges and Lessons Learned. By D. Falessi, M. A. Babar, G. Cantone, P. Kruchten. Empirical Software Engineering. Volume 15. Number 3. 2010. Pages 250-276.

[Fehling, Leymann, Retter, Schupeck, Arbitter, 2014] Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. By C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. Springer-Verlag. 2014.

[Fowler, Rice, Foemmel, Hieatt, Mee, Stafford, 2003] Patterns of Enterprise Application Architecture. By M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, R. Stafford. Addison-Wesley. 2003.

[Gamma, Helm, Johnson, Vlissides, 1995] Design Patterns: Elements of Reusable Object-Oriented Software. By E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley. 1995.

[Ghezzi, Jazayeri, Mandrioli, 2003] Fundamentals of Software Engineering. By C. Ghezzi, M. Jazayeri, D. Mandrioli. Second Edition. Prentice-Hall. 2003.

[Gomaa, 2004] Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. By H. Gomaa. Addison-Wesley. 2004.

[Gomaa, 2011] Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. By H. Gomaa. Cambridge University Press. 2011.

[Hohpe, Woolf, 2004] Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. By G. Hohpe, B. Woolf. Addison-Wesley. 2004.

[Homer, Sharp, Brader, Narumoto, Swanson, 2014] Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. By A. Homer, J. Sharp, L. Brader, M. Narumoto, T. Swanson. Microsoft. 2014.

[ISO/IEC, 2001] ISO/IEC 9126-1:2001. Software Engineering -- Product Quality -- Part 1: Quality Model. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC). 2001.

[ISO/IEC/IEEE, 2010] ISO/IEC/IEEE 24765:2010. Systems and Software Engineering -- Vocabulary. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC)/The Institute of Electrical and Electronics Engineers (IEEE) Computer Society. 2010.

[ISO/IEC/IEEE, 2011] ISO/IEC/IEEE 42010:2011. Systems and Software Engineering -- Architecture Description. The International Organization for Standardization (ISO)/The International Electrotechnical Commission (IEC). 2011.

[Kamthan, 2008] A Situational Methodology for Addressing the Pragmatic Quality of Web Applications by Integration of Patterns. By P. Kamthan. Journal of Web Engineering. Volume 7. Number 1. 2008. Pages 70-92.

[Kamthan, 2009] On the Symbiosis between Quality and Patterns. By P. Kamthan. The Third International Workshop on Software Patterns and Quality (SPAQu 2009). Orlando, U.S.A. October 25, 2009.

[Kamthan, 2011] Towards Understanding the Use of Patterns in Software Engineering. By P. Kamthan. In: Knowledge Engineering for Software Development Life Cycles: Support Technologies and Applications. M. Ramachandran (Editor). IGI Global. 2011. Pages 115-135.

[Kamthan, Fancott, 2011] A Knowledge Management Model for Patterns. By P. Kamthan, T. Fancott. In: Encyclopedia of Knowledge Management. D. G. Schwartz, D. Te'eni (Editors). Second Edition. IGI Global. 2011. Pages 694-703.

[Larman, 2005] Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. By C. Larman. Third Edition. Prentice-Hall. 2005.

[Lavariega, Avila, Gómez-Martínez, 2015] Software Architecture for Emergency Remote Pre-hospital Assistance Systems. By J. C. Lavariega, A. Avila, L. G. Gómez-Martínez. In: Mobile Health: A Technology Road Map. S. Adibi (Editor). Springer International Publishing. 2015. Pages 453-472.

[Lethbridge, Laganière, 2005] Object-Oriented Software Engineering: Practical Software Development using UML and Java. By T. C. Lethbridge, R. Laganière. Second Edition. McGraw-Hill. 2005.

[Marinescu, 2013] Cloud Computing: Theory and Practice. By D. C. Marinescu. Morgan Kaufmann. 2013.

[Medvidović, Dashofy, Taylor, 2007] Moving Architectural Description from Under the Technology Lamppost. By N. Medvidović, E. M. Dashofy, R. N. Taylor. Information and Software Technology. Volume 49. Issue 1. 2007. Pages 12-31.

[Meszaros, Doble, 1998] A Pattern Language for Pattern Writing. By G. Meszaros, J. Doble. In: Pattern Languages of Program Design 3. R. C. Martin, D. Riehle, F. Buschmann (Editors). Addison-Wesley. 1998. Pages 529-574.

[Monson-Haefel, 2009] 97 Things Every Software Architect Should Know: Collective Wisdom from the Experts. By R. Monson-Haefel (Editor). O'Reilly Media. 2009.

[Nuseibeh, 2001] Weaving Together Requirements and Architectures. By B. Nuseibeh. Computer. Volume 34. Issue 3. 2001. Pages 115-117.

[Plakalović, Simić, 2010] Applying MVC and PAC Patterns in Mobile Applications. By D. Plakalović, D. Simić. Journal of Computing. Volume 2. Issue 1. 2010. Pages 65-72.

[Qin, Xing, Zheng, 2009] Software Architecture. By Z. Qin, J. Xing, X. Zheng. Springer-Verlag. 2009.

[Saffer, 2009] Designing Gestural Interfaces. By D. Saffer. O'Reilly Media. 2009.

[Sangwan, 2015] Software and Systems Architecture in Action. By R. S. Sangwan. CRC Press. 2015.

[Schumacher, Fernandez-Buglioni, Hybertson, Buschmann, Sommerlad, 2006] Security Patterns: Integrating Security and Systems Engineering. By M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad. John Wiley and Sons. 2006.

[Sebesta, 2012] Concepts of Programming Languages. By R. W. Sebesta. Tenth Edition. Addison-Wesley. 2012.

[Seshadri, 1999] Understanding JavaServer Pages Model 2 Architecture. By G. Seshadri. Java World. December 29, 1999.

[Tyree, Akerman, 2005] Architecture Decisions: Demystifying Architecture. By J. Tyree, A. Akerman. IEEE Software. Volume 22. Issue 2. 2005. Pages 19-27.

[Vliet, Tang, 2016] Decision Making in Software Architecture. By H. van Vliet, A. Tang. The Journal of Systems and Software. Volume 117. 2016. Pages 638-644.

[Vogel, Arnold, Chughtai, Kehrer, 2011] Software Architecture: A Comprehensive Framework and Guide for Practitioners. By O. Vogel, I. Arnold, A. Chughtai, T. Kehrer. Springer-Verlag. 2011.

[Yoder, Barcalow, 1997] Architectural Patterns for Enabling Application Security. By J. Yoder, J. Barcalow. The Fourth Conference on Pattern Languages of Programs (PLoP 1997). Monticello, U.S.A. September 3-5, 1997.

[Zhang, Budgen, 2012] What Do We Know about the Effectiveness of Software Design Patterns? By C. Zhang, D. Budgen. IEEE Transactions on Software Engineering. Volume 38. Issue 5. 2012. Pages 1213-1231.