11

Using Styles and Patterns

Historically, the software industry hasn't had a very good record of learning from experience. Software designers often ignore existing, proven design solutions and instead develop their own solutions to complex problems. The same can also be said of software architects, who can end up creating new designs for systems containing very familiar challenges.

One of the reasons for this state of affairs used to be a lack of easily accessible, standard solutions for common software architecture and design problems. However, during the 1990s, the design patterns movement emerged with the aim of addressing this problem. Proponents of software patterns, inspired by Christopher Alexander's work on patterns for building architecture,[1] started identifying and cataloging widely used solutions to common design problems. Today, this work has culminated in an ever-growing number of patterns available for general use.

Software Patterns

The purpose of a software pattern is to share a proven, widely applicable solution to a particular design problem in a standard form that allows it to be easily reused. Software patterns should provide the following five important pieces of information.

1. *Name*: A pattern needs a memorable and meaningful name to allow us to clearly identify and discuss the pattern and, more importantly, to use its name as part of our design language when discussing possible solutions to design problems.

1. *A Pattern Language: Towns, Buildings, Construction,* by Christopher Alexander, Sara Ishikawa, and Murray Silverstein (Oxford: Oxford University Press, 1977). Alexander is a building architect, and his books look at building architecture, but his ideas have inspired virtually everyone in the design patterns community.

2. *Context*: This sets the stage for the pattern and describes the situations in which the pattern may apply.
3. *Problem*: Each pattern is a solution to a particular problem, so part of the pattern's definition must be a clear statement of the problem that the pattern solves and any conditions that need to be met in order for the pattern to be effectively applied. A common way to describe the problem that a pattern solves is to describe the design *forces* it aims to resolve, each force being a goal, requirement, or constraint that informs or influences the solution (such as a particular sort of flexibility needed or a particular type of interelement decoupling you want to achieve).
4. *Solution*: The core of the pattern is a description of the solution to the problem that the pattern addresses. This is usually some form of design model, explaining the elements of the design and how they work together to solve the problem.
5. *Consequences*: The definition of a software pattern should include a clear statement of the results and tradeoffs that will result from its application, to allow you to decide whether it is a suitable solution to the problem. Consequences may be positive (benefits) or negative (costs).

Let's look at a simple example of what goes into a software pattern.

EXAMPLE The widely used *Adapter* pattern is so named because it adapts the interface of a system element to a form needed by one of its clients. An outline definition of the pattern could be as follows.

The *context* of the Adapter pattern is a number of heterogeneous elements that need to be connected. The pattern solves the *problem* that arises when one system element (the client) could use the services of another system element (the target) except that the interface offered by the target element is unsuitable for use by the client. An example is a system with a .NET client that wishes to access a calculation service offered by a Java-based target. The calculation service is perfectly suitable for the client's requirements, but the client cannot call a Java-based interface and thus cannot use the service.

The *forces* include the following.

- Service interfaces should be decoupled from the underlying physical data structures and implementation algorithms.
- Services should be exposed in a way that is independent of their implementation technology.

- The adapter should provide translation only and should not perform any inherently useful functionality (this is the responsibility of the invoked service).
- Use of the adapter must not adversely affect the quality properties of the underlying service (security, resilience, performance, scalability, and so on).

The *solution* to the problem is to introduce a third system element, the adapter, sitting between the client and the target, such that it is called by the client and calls the target. The role of the adapter is simply to interpret the request from the client, transform it into the form required by the target, call the target, and transform the response into the form expected by the client. A real-world example of an adapter is an international power-plug adapter that allows, for example, electrical equipment with French plugs to be used in Denmark with Danish power sockets.

The *consequences* of using this pattern include the following.

- Decoupling of the client and target implementations allows each implementation to be varied without impacting the other. (+)
- The target can be used by different types of clients simultaneously (possibly via different adapters). (+)
- A possible reduction in efficiency could result due to the additional level of indirection between the client and target. (–)
- An increase in maintenance overhead could occur if the services provided and used change because the adapter must be changed as well as the client and target. (–)

You can find a much fuller definition of this pattern in the well-known "Gang of Four" book referenced in the Further Reading section at the end of this chapter.

STYLES, PATTERNS, AND IDIOMS

Software patterns are generally organized into three groups: *architectural styles* that record solutions for system-level organization, *design patterns* that record solutions to detailed software design problems, and *language idioms* that capture useful solutions to language-specific problems. All three types of patterns can be useful to you, although you use them in different places in the lifecycle. Before considering how to use styles, patterns, or idioms, let's define these three terms more formally. The definitions we present are all based

on those defined by Frank Buschmann and his colleagues in their book *Pattern-Oriented Software Architecture*.

Architectural Styles

Architectural styles are probably the software pattern type that will be of most immediate interest to you when designing a system because they apply to system-level structures.

DEFINITION An **architectural style** expresses a fundamental structural organization schema for software systems. It provides a set of predefined element types, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

The key point about an architectural style is that it provides a set of organizational principles for the system as a whole, rather than for the details of one piece of the system. The solution described by an architectural style is usually defined in terms of types of architectural elements and their interfaces, types of connectors, and constraints on how the elements and connectors should be combined.

Design Patterns

A design pattern is a solution to a much more specific problem related to the structure of a particular part of a system.

DEFINITION A **design pattern** provides a scheme for refining the elements of a software system or the relationships between them. It describes a commonly recurring structure of interconnected design elements that solves a general design problem within a particular context.

A design pattern forms an input to the detailed software design of the system and guides a software designer to organize her software design units (such as classes and procedures) appropriately. The solution presented by a design pattern is defined in terms of design-level elements (such as procedures, classes, and data structures) and the structure they form when combined.

Language Idioms

Language idioms are the most specific type of software pattern, applying to situations where a particular programming language is in use.

DEFINITION A **language idiom** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of elements or the relationships between them by using the features of a given language.

A language idiom provides guidance to the programmer when implementing software in a specific language and is normally written to help prevent a common pitfall with the language or to illustrate a unique feature that needs to be learned. The solutions presented by language idioms are defined in terms of programming language constructs.

Patterns of all three varieties can play several helpful roles including the following.

- *A store of knowledge*: Patterns are a store of knowledge about solving a particular type of problem in a particular domain. Documenting this knowledge allows it to be shared among people solving similar problems. People can move between specialist areas more easily and work more effectively within a particular area by sharing knowledge about success and failure.
- *Examples of good practices*: A set of patterns provides examples of good design practices. You can use these examples directly, but they can also act as a guide and provide inspiration when you're solving somewhat different design problems.
- *A language*: Patterns allow designers to create and share a common language for discussing design problems. This common language helps designers relate ideas to each other easily and analyze alternative solutions to a problem. This allows for more effective communication among participants in the design process.
- *An aid to standardization*: The use of patterns encourages designers to choose standard solutions to recurring problems rather than searching for novel solutions in each case. This has obvious efficiency benefits for the design process, and reliability is also likely to increase because of the reuse that results from the application of an already proven solution.
- *A source of constant improvement*: Because patterns are generally in the public domain, you can quickly amass a lot of experience about their use. This allows rapid feedback into the pattern definition and promotes improvement over time, reflecting the experiences of its users.
- *Encouragement of generality*: Good patterns are usually generic, flexible, and reusable in a number of situations. Providing flexible and generic solutions to problems is often a goal for architects as well. Using patterns as inputs to the design process and thinking in terms of identifying design patterns within the design process can help you create flexible, generic solutions to the problems within your system.

From our point of view as architects, the real utility of design patterns in software development can be summarized in a single phrase: *reduction of risk*. The use of patterns (and ideally reusable pattern implementations) has the potential to increase productivity, standardization, and quality while reducing risk and repetition.

AN EXAMPLE OF AN ARCHITECTURAL STYLE

Having considered architectural styles in the abstract, let's continue by considering an example of a specific architectural style.

EXAMPLE Here is a summary of the Pipes and Filters architectural style. (This is just a summary; you can find a much fuller definition in *Pattern-Oriented Software Architecture* [BUSC96].)

The *context* of the Pipes and Filters style is a system that needs to process data streams.

The style solves the *problem* of implementing a system that must process data in a sequence of steps, where using a single process is not possible and where the requirements for the processing steps may change over time.

The problem has the following primary *forces*.

- Future changes should be possible by changing or recombining steps.
- Small processing steps are easier to reuse than large ones.
- Nonadjacent steps in the process do not share information.
- Different possible sources of input data exist.
- Explicit storage of intermediate results should be avoided.
- Multiprocessing between steps should not be ruled out.

The *solution* to this problem is to divide the task into a number of sequential steps and to connect the steps by the system's data flow.

The processing is performed by filter components, which consume and process data incrementally. The input data to the system is provided by a data source while the output flows into a data sink. The data source, data sink, and filter components are connected by pipes. The pipe implements data flow between two adjacent components. The pipe is the only permitted way to connect the other components, and it defines a simple, standard format for data that passes through it, allowing filters to be combined without prior knowledge of each other's existence.

The sequence of filters combined by pipes is called a *processing pipeline*. An example of a processing pipeline appears in the informal diagram in Figure 11–1, which shows how the pieces of the Pipes and Filters style are combined. As indicated by the UML-style comments, the boxes represent the filters and the arrows represent unidirectional pipes linking the filters. Each filter performs a single task—for example, the NPV filter calculates net present value (the present value of an investment's future cash flows less the initial investment).

The *consequences* of using this style are as follows.

- No intermediate files are necessary, but they are possible. (+)
- Filter implementation can be easily changed without affecting other system elements. (+)
- Filter recombination makes creating new pipelines from existing filters easy. (+)
- Filters can be easily reused in different situations. (+)
- Parallel processing can be supported with multiple filters running concurrently. (+)
- Sharing state information is difficult. (–)
- The data transformation required for a common interfilter data format adds overhead. (–)
- Error handling is difficult and needs to be implemented consistently. (–)

You are almost certainly familiar with this architectural style from the UNIX operating system. However, it has been applied in a number

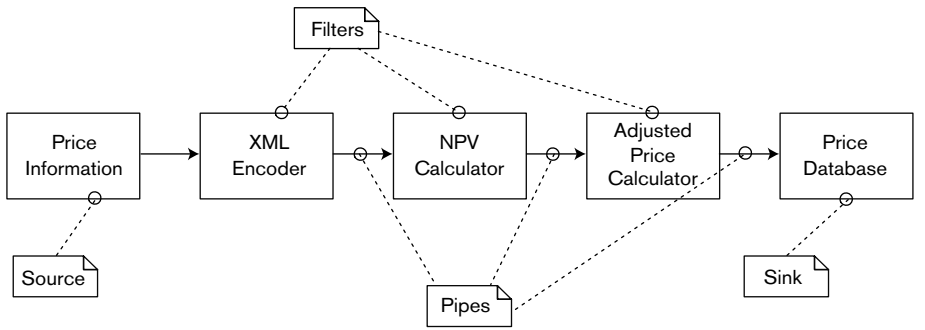

FIGURE 11–1 PROCESSING PIPELINE

of other software systems, such as Enterprise Application Integration (EAI) systems, and it is a useful system organization for a specific type of data processing problem that may occur across a number of application domains.

What does this architectural style definition tell us about systems based on it? Some of the important points are listed here.

- The system processes streams of data, rather than transactions.
- The processing can be broken into a series of independent steps.
- There is just one sort of architectural element (the filter) and one type of unidirectional connector (the pipe), and the filters must form a continuous path through the system, connected by pipes, without any cycles.
- The system doesn't need a central persistent data store.
- It should be easy to replace and reuse the system's filter elements.
- It is likely to be difficult to modify the system to address a situation where elements need to maintain or share state.
- The architect of the system will need to define and enforce an error-handling strategy because the style makes this something of a challenge.

An architectural style tells us what sort of structure a system based on it will have, in terms of the types of system elements and the structure they combine to form. The style also explains the design concerns (or forces) that led to its development and the positive and negative implications to be considered when using it.

THE BENEFITS OF USING ARCHITECTURAL STYLES

Basing your architecture on a recognizable style can have two immediate benefits. First, using a style allows you to select a proven, well-understood solution to your problems and defines the organizing principles for the system. Second, if people know that the architecture is based on a familiar style, it helps them understand its important characteristics.

In our experience, most architects do reuse good ideas they've seen before and do match previously successful solutions to the characteristics of the current problem. It's simply that we do this with varying degrees of formality and may not consider ourselves to be using architectural styles as we do it.

Most of the books documenting architectural styles assume that the styles will be used directly as generic blueprints to guide the design process. However, styles can be used during the architecture definition process in a number of ways.

- *Solution for a system design*: One of the styles you encounter may be a good solution to the particular problems you are trying to solve. In this case, you can simply adopt the style as one of the core structures of your architecture and enjoy the benefits of understanding its likely strengths and weaknesses immediately.
- *Basis for adaptation*: When considering existing styles, you may find that none of them really solve your problem, but they partially address it or address it with some limitations. In these situations, the style forms a starting point for the design process but acts as a base to be adapted to the particular constraints of the current situation. This identifies a candidate variant of the original style.
- *Inspiration for a related solution*: You may also find that none of the styles actually address the problem you are trying to solve. However, simply reading about previously identified styles and the problems they address often helps you understand the current problem in more depth so you can find a solution related in some way to the existing styles. This identifies a related candidate style.
- *Motivation for a new style*: Sometimes you are faced with a problem that does not seem to be addressed by any of the styles you have found. In this case, perhaps you are solving a problem that hasn't been widely solved before, or if it has, its solution hasn't been documented as a style. These situations often act as a spur to solve the problem in a general way and to define a new candidate architectural style capturing the resulting design knowledge.

Architectural styles tend to be quite one-dimensional, focused on solving a very specific type of problem. This means that, in all but the simplest systems, you usually need to combine a number of styles to meet the varied design problems that most systems place before you.

EXAMPLE Consider a financial trading system that needs to allow users to perform transactions but also needs to broadcast information (such as news or prices) across the system. At least two architectural styles immediately suggest themselves: the Client/Server style to allow transaction processing via central servers and the Publisher/Subscriber style to allow news and price information to be broadcast throughout the

system. Another style may be needed too, such as Layered Implementation, in order to achieve portability and common use of the underlying platform. In this example, each style is needed for a distinct reason.

- The Client/Server style is present to allow secure, scalable, available transaction processing that performs well.
- The Publisher/Subscriber style is present to allow efficient, flexible, asynchronous distribution of information.
- The Layered Implementation style is present to ensure portability across deployment platforms, to ensure a common approach to the use of underlying technology, and to achieve a good level of development productivity by hiding low-level details of the underlying technology from most of the system developers.

Situations in which we need to combine styles present us with a twofold problem. First, how do we use a number of styles together and retain overall coherence in our architecture? Second, how do we know that the styles will work together and not conflict with each other?

With this problem, as with so many others in software architecture, there really is no substitute for the blend of experience, knowledge, and sound judgment that you bring to the project. Our experience with system architectures has taught us that combining styles is difficult and needs to be done carefully. In particular, the key to success is to choose one of the styles as the dominant organizing style for the system and to structure the architecture around it, introducing the other styles as subsidiary styles where they are needed to solve particular problems that the primary style cannot address by itself. Such an approach helps retain overall coherence and prompts you to consider the compatibility of the styles as you try to add the elements of the subsidiary styles to the structure imposed by the overall system organization defined by the primary style.

STYLES AND THE ARCHITECTURAL DESCRIPTION

You are likely to adopt a particular architectural style in your system because you want the system to exhibit certain qualities that the style claims to provide. Once you've chosen one or more styles to use, it is easy to forge ahead and create an architectural design based on the styles while forgetting to explain the styles you've used in the AD. When this happens, the knowledge about the styles used and why they were selected gets lost, so it is likely that, over time, the architecture will diverge from the structure suggested by the style.

In addition, many architectural styles focus on the overall functional structure of the system and will primarily affect the Functional view in the AD. However, in principle, a style could affect any of the architectural views, so explaining the styles you've used can be important to allow readers of the AD to understand the impact a style has had across the different aspects of your architecture. For these reasons, it is useful to explicitly explain the styles you have used in the AD.

We've found that there are a couple of effective ways to do this: textual commentary and model annotation. Textual commentary simply involves adding to the AD document a brief discussion of the styles you've used and why you used them, probably as an early subsection to help set the scene for the reader. Annotating your models takes this a step further and draws the reader's attention to the relationship between the generic style in use and the specific elements in the architecture. If you are using UML as your modeling notation, you can achieve this by using stereotypes to mark model elements as corresponding to particular abstract elements of the style in use. However, you need to avoid cluttering the models with a lot of extra notation, which prevents people from understanding them easily. If this starts to happen, remove the annotation from the model diagrams and just add textual notes in your explanation of the model.

COMMON ARCHITECTURAL STYLES

A few of the currently documented architectural styles that are particularly relevant to information systems architecture are briefly summarized below. You can find more information on most of these styles in the books we reference in the Further Reading section or from Internet sources.

Pipes and Filters

We defined this style as an example earlier in this chapter. The Pipes and Filters style is characterized by a single, simple element type (the filter) that processes a data stream, with instances of this type connected by simple connectors known as pipes.

Example of Use. Refer to the example illustrated in Figure 11–1.

Advantages. The style allows filter implementation to easily be changed without affecting other system elements, and it makes creating new pipelines by recombining existing filters easy. Parallel processing can also be supported with multiple filters running concurrently.

Disadvantages. Sharing state information is difficult with this style, and the data transformation required for a common interfilter data format adds

overhead. Error handling is difficult and needs to be implemented consistently down the pipeline.

Common Variants. Pipes can execute in parallel rather than in sequence, or filters can have more than one entrance or exit channel.

Client/Server

This very widely used style defines a system structure comprised of two types of elements: a server that provides one or more services via a well-defined interface and a client that uses the services as part of its operation. The client and server are typically assumed to reside on different machines in a network (although this is not a requirement of the style—the client and server could be in the same operating system process).

Example of Use. This style is probably very familiar to you from mainstream IT technologies such as client/server databases.

Advantages. The advantages of the style include the centralization of complex or sensitive processing (e.g., allowing specialized hardware to be shared easily).

Disadvantages. Making the request and receiving the response between clients and servers that do not reside on the same machine introduces some unavoidable inefficiency.

Common Variants. Variants of the Client/Server style include Stateful Server (where the server is responsible for keeping track of conversational state) and Stateful Client, Stateless Server (where the client is responsible for keeping track of the state).

Tiered Computing

A development of the Client/Server style, the Tiered Computing style is widely used in enterprise information systems. A tiered system is considered to contain a number of tiers of computation, which combine to offer a service to an ultimate consumer (e.g., a human user). Each tier acts as a server for its caller and as a client to the next tier in the architecture. A key architectural principle is that a tier can communicate in this way only with the tiers immediately on either side of it; a tier is not aware of the existence of other tiers in the system apart from its neighbors. Common tiers in enterprise information systems include the following:

- Presentation (user interface)
- Business process (combining a sequence of business transactions into a service)
- Business transactions (the fundamental business operations in the system that act on business entities)

- Data access (providing governed access to business entities)
- Data storage (the databases in which the enterprise's data resides)

Example of Use. Most large enterprise information systems are organized into tiers, and a number of common application development technologies (such as J2EE and .NET) encourage this organization.

Advantages. This style allows a clear separation of concerns between tiers and provides the potential for reusability of the simpler tiers (data storage, data access, and business transactions) across a number of systems.

Disadvantages. Disadvantages of the style include the overhead of communication between the tiers and additional development complexity arising from the number of system elements that need to be developed and integrated across the tiers.

Peer-to-Peer

Often referred to as P2P, this architectural style defines a single type of system element (the peer) and a single type of connector that is a network connection (an interpeer connection). The characteristics of the connector are not important to the style, and the style has been used with a number of types of network connections.

The central organizational principle of the system is that any peer is free to communicate directly with any other peer, without needing to use a central server. Peers typically locate each other by automatically exchanging lists of known peers (although central peer lists are also used in some cases). Each peer is capable of acting as both client (when making requests) and server (when servicing requests), often being both concurrently.

Example of Use. Well-known examples of the P2P style include Internet file-swapping applications and distributed computation systems.

Advantages. P2P systems eliminate the possible point of failure that a central server represents, can be made very scalable, and are resilient to partial failures in the underlying network.

Disadvantages. Disadvantages of P2P systems include the possible partitioning of the network if no central peer list is available and the difficulty of guaranteeing a particular response from the system at any point in time.

Layered Implementation

The Layered Implementation style identifies a single type of system element: the layer. The style organizes a system's implementation into a stack of layers,

with each layer providing a service to the layer above it and requesting services from the one below it. The layers are ordered by the level of abstraction they represent, with the most abstract (e.g., organization-specific operations) at the top of the stack and the least abstract (e.g., operating system–specific libraries) at the bottom. Depending on the implementation of the style, a layer may be able to communicate directly with any of the layers below it (relaxed layering) or only with the layer directly below it (strict layering).

Layers can be contrasted with the tiers in the Tiered Computing style because layers are organized based on the level of abstraction they deal with, whereas tiers are organized based on the type of service they provide. All of the layers in a particular implementation are concerned with providing a single service, but each layer is concerned with a different level of abstraction involved in providing the service. In contrast, the tiers in an implementation all operate at a broadly similar level of abstraction but are each concerned with providing a different type of service, which, when the services are combined, creates a useful system. Given this difference, tiers are often visualized as running horizontally, while layers are often visualized as running vertically. Indeed, the two styles are often combined, with each tier in the system using a stack of layers within it to organize its implementation into different levels of abstraction.

Example of Use. Communication stacks are the classic example of layered organization, but most information system technologies are organized in this way too (e.g., a locally developed utility library layered above a third-party library layered above the operating system).

Advantages. Likely advantages of this style of organization include the reuse of layers, good separation of concerns, and relatively easy maintenance due to the isolation of each layer's implementation.

Disadvantages. Disadvantages include a reduction in implementation flexibility, a reduction in efficiency when many layers need to be traversed, and the constraints that the style places on the development process (layers often need to be developed in order).

Publisher/Subscriber

The Publisher/Subscriber style grew out of a realization that client/server interactions are not suitable for all types of distributed system problems. The style defines a single system element (the publisher) that creates information of interest to any number of system elements (the subscribers) that may wish to consume it. A single type of connector, a reliable network link, is used to link the publisher and the subscribers.

The subscribers register their interest in certain information with the publisher, and when the publisher creates or changes information that subscribers have registered their interest in, the publisher notifies the relevant subscribers

of the change. Depending on the implementation of the style, the notification may contain the new or changed information, or it may just be a notification of a relevant change, leaving the subscribers to query the publisher for changes themselves.

Example of Use. The Publisher/Subscriber style is widely implemented in enterprise messaging systems.

Advantages. Advantages include the flexibility to add new subscribers dynamically, the relatively loose coupling between publisher and subscribers, and the increased efficiency that comes from the subscribers not having to poll the publisher repeatedly to find new and changed information.

Disadvantages. The main disadvantage of the style is its relatively complex implementation (particularly if reliable delivery of messages is required).

Asynchronous Data Replication

While Publisher/Subscriber is normally considered to be a style that allows functional elements to exchange information, a variant of it, Asynchronous Data Replication, is a style used where information in two data stores needs to be kept synchronized (e.g., where it is replicated for performance reasons). The style has three element types: the data source, the data replica, and the replicator. The data source is a data store that owns a particular type of information, while the data replica is a separate data store that wishes to maintain a synchronized copy of some subset of the information in the source. The replicator is the element responsible for recognizing changes or additions to information in the source and performing the synchronization of the replica data store.

Example of Use. This style is widely implemented in enterprise data replication technologies, such as those supplied by the major database vendors.

Advantages. The advantages of the style include the ability to synchronize two data stores automatically and efficiently, without needing to complicate the application logic.

Disadvantages. Common problems include the latency that can occur between source update and replica update as well as the complexity of dealing with updates at the replica data store.

Distribution Tree

Another architectural style that relates to data distribution is the Distribution Tree style. This style defines three types of system elements: publishers, distributors, and consumers. The elements of the system are arranged into a tree, connected by network link connectors. A single publisher forms the root of the tree, the distributors are connected to form the intermediate nodes, and

the consumers form the leaves of the tree. The publishers publish new or changed information, which the distributors then cache and distribute to their immediate child nodes. If a child node is added or restarted, it can refresh its view of the information from its parent node's cache. When the information reaches the leaf nodes of the tree, the consumer nodes consume it.

Example of Use. This style is implemented by a number of Internet push-client products.

Advantages. Distribution Tree can be scaled from small implementations to very large ones by increasing the size of the tree and its support for intermittently connected consumer nodes (such as mobile devices).

Disadvantages. Disadvantages of the style include the amount of storage required for caching and the potentially high update latency when the distribution tree becomes large.

Integration Hub

The Integration Hub style is another data-oriented architectural style, extending Asynchronous Data Replication to situations where information needs to be synchronized between a number of different systems (rather than between replica data stores).

This style defines four types of system elements: the data source, the data destination, the hub, and the adapter. The elements are organized into a cartwheel form, with the hub at the center of the wheel and the data sources and destinations at the outer edges of the spokes. Along every spoke, between hub and source or destination is an adapter (so each spoke is an adapter connected to a source or destination, radiating out from the hub). The nature of the connectors is not that important; the main characteristic to note is that the connector allows unidirectional data flow, from one end to the other.

The implementation of the hub includes a common data model for data entities of interest in the system. The adapters are responsible for translating between the common data model and the specific data models of the source or destination they connect. The sources and destinations simply supply data to or receive data from their adapters.

Information can flow between any source and destination by the data being transformed from the source's form into the common form (by the source's adapter) and then from the common form to the destination's form (by the destination's adapter). This allows routine data transfer and synchronization between systems that do not share common data models, formats, and encodings.

Example of Use. This style is widely implemented by EAI products that allow data integration between applications.

Advantages. Integration Hub allows new data sources and destinations to be added easily to the system without disrupting the existing implementation. It

can also integrate sources and destinations of practically any form because a dedicated adapter hides the specifics of a source or destination.

Disadvantages. Data movement between applications will be relatively inefficient due to the amount of translation, and the imposition of a common model may mean that some information that the common model does not accommodate is lost in translation. The design of the common model can also be quite difficult.

The hub itself can become a central point of failure or a performance bottleneck if it is not designed properly. This risk can be mitigated to some extent by extending the cartwheel topology into a snowflake which links multiple hubs together.

Tuple Space

The Tuple Space style is a type of repository that allows a number of pieces of a distributed system to collaborate to share information. The two types of system elements in the style are the clients (computational elements that create and consume information) and the tuple space itself (a storage area where clients can read and write typed information tuples or records). The clients and the tuple space are connected by a client/server network connection. Clients interact with the tuple space by writing new tuples to it or requesting tuples that match simple search criteria. Clients do not interact directly. Typically the tuple space can also call back to the clients when objects they are interested in change.

Example of Use. Examples of this style include the original Linda research system (from which the style was derived) and more recent implementations such as JavaSpaces for the Java language. An application of the approach might be a price source in a banking system that allows various types of price information to be published by any number of price sources to a single tuple space and accessed from the space by any number of applications that need price information.

Advantages. The Tuple Space style provides a simple computational model, encourages loose coupling between clients, and provides good support for evolution due to the easy evolvability of the tuple space itself.

Disadvantages. Disadvantages can include the scalability limits of the tuple space element and the limited types of interaction available between system elements.

Common Variants. A more specialized version of this style known as Blackboard organizes the clients so that they collaborate to use the shared data to solve a particular problem, reading intermediate results from the blackboard, processing them, and writing the results back for other clients to process further.

DESIGN PATTERNS AND LANGUAGE IDIOMS IN ARCHITECTURE

Although it's fairly clear how you can use architectural styles because they act as a source of proven architectural design ideas, it's not necessarily as clear how design patterns and language idioms fit into the architectural design process. Given that detailed design and coding isn't your key focus, how do design patterns and language idioms contribute to your main area of work?

The answer is that design patterns and language idioms are a very important written communication path from architects to software developers. It's crucial to communicate directly by leading the team and talking face-to-face, but there are also many situations where it's important to get design constraints and guidelines on paper so that everyone can understand and consider them thoroughly. Design patterns and language idioms are a perfect mechanism for communicating design advice and constraints to a development team.

EXAMPLE Here are some typical examples of using design patterns and language idioms.

Examples of Using Design Patterns

- If you are developing a system that requires internationalization, this is an important system-wide design constraint. In order to ensure the use of a common approach to internationalization across the systems' modules, adopt or define a design pattern that illustrates how this part of a module's implementation should be performed.
- Many database applications need to use specific approaches to locking (e.g., the choice of using optimistic or pessimistic locks depending on data integrity and concurrency needs). The locking approach to use in certain situations may be an important design constraint resulting from the architectural design. Where this is the case, use a design pattern to define how database locking must be implemented.
- The evolutionary needs of the system may require that new code can be easily introduced to handle new types of data. In order to guide the design process to achieve the required flexibility, you could suggest the use of relevant design patterns like Chain of Responsibility, Reflection, or Visitor to help explain to the developers concerned the type of flexibility you need.

Examples of Using Language Idioms

- Many modern programming languages such as Java, C++, and C# include exception-handling facilities. These facilities can be used in a number of ways, so an important architectural constraint is to standardize the exception handling. Adopt or create a language idiom that defines how the programming language's exception-handling facilities should be used, and ensure that the idiom is used throughout the system.
- To allow a system to be easily instrumented, it can be very useful for each element to be able to return a string containing its state so that this can be written to a debug log. This is possible in most programming languages, but the mechanism available and the best way to use it varies. You can standardize this across your system by defining or adopting an idiom to be used when implementing each system element.
- Many languages have features that need careful use to avoid subtle problems creeping in later (such as the advice to override either both or neither of Java's `equals()` and `hashCode()` methods or the need to define a copy constructor in C++ to avoid problems when using object assignment). You can help make sure that such language-specific problems don't emerge with your system by working with your senior developers to define or adopt language idioms to provide guidance in potentially problematic areas.

The identification and capture of design patterns and language idioms that are important for your system would normally be part of the activity of creating the Development view and they are normally captured as part of the common design model in that view. Practically, because this documentation can be quite large, it usually makes sense to capture patterns and idioms as part of a development standards document referenced from the AD.

CHECKLIST

- Have you considered existing architectural styles as solutions for your architectural design problems?
- Have you clearly indicated in your AD where you have used architectural styles?
- Have you reviewed likely sources for possible new styles, patterns, and idioms that may be relevant to your system?

- Do you understand the design forces addressed by the patterns you use and the strengths and weaknesses of each pattern?
- Have you defined patterns and idioms to document all important design constraints for your system?
- Have you considered using design patterns and idioms to provide design guidance where relevant?

SUMMARY

Architectural styles, design patterns, and language idioms (collectively known as *patterns*) are all ways to reuse proven software design knowledge, and all three are valuable during the architectural design process. Patterns provide a reusable store of knowledge, help develop a language for discussing design, and encourage standardization and generality in design.

Becoming familiar with a range of architectural styles helps you build your design vocabulary and provides you with a library of options to consider when you meet new architectural design problems. Styles can also form the basis for further refinement and the inspiration for entirely new solutions, as well as simply being design blueprints. A good selection of relevant architectural styles for information systems already exists, and part of an architect's training is getting to know these styles and the strengths and weaknesses of each.

Patterns and idioms also help expand your knowledge of proven design solutions for more detailed problems, but they also are a valuable mechanism for recording the design constraints and guidelines that are important to achieving architectural integrity in the system's implementation.

FURTHER READING

The original book on design patterns is *Design Patterns* [GAMM95], often referred to as the "Gang of Four" or "GoF" book, which is still a definitive source of basic design patterns. A good place to start reading about patterns from the perspective of an architect is Buschmann et al., *Pattern-Oriented Software Architecture* [BUSC96] (generally known as "POSA1"). Our definitions of style, pattern, and idiom come from this book. Shaw and Garlan [SHAW96] is one of the original descriptions of architectural styles.

The Pattern Languages of Program Design conference series [PLOP95–99] has produced a large number of design-level patterns during the years it has been running. These references present the results of pattern-writing workshops at the conferences and are a rich source of useful design patterns.

More recently, Fowler has lead the creation of a book [FOWL03] containing a large number of patterns found in enterprise information systems that are likely to be of use to most information systems architects. Another recent book that contains a very valuable set of patterns focusing on the deployment aspects of large information systems is Dyson and Longshaw [DYSO04].

There are too many books of language idioms to list here, but one of the originals is Coplien's book of C++ idioms [COPL91], while more recent examples include Bloch [BLOC01] for Java.

A fair number of Web sites have appeared that contain design patterns. The speed of evolution of Web-based information sources makes it futile to attempt to present a list here, but a few of the original pattern resources are the Hillside Group, which organizes the Pattern Languages of Program Design conferences (www.hillside.net), and the patterns area of Ward Cunningham's C2 Wiki site (http://c2.com/cgi-bin/wiki?PatternIndex).