P.1 A Short Overview of Software Architecture

P.1.1 Overview

image

The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

[Software architecture](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_47) has emerged as an important subdiscipline of software engineering. Architecture is roughly the prudent partitioning of a whole into parts, with specific relations among the parts. This partitioning is what allows groups of people—often separated by organizational, geographical, and even time-zone boundaries—to work cooperatively and productively together to solve a much larger problem than any of them could solve individually. Each group writes software that interacts with the other groups’ software through carefully crafted interfaces that reveal the minimal and most stable information necessary for interaction. From that interaction emerges the functionality and quality attributes—security, modifiability, performance, and so forth—that the system’s stakeholders demand. The larger and more complex the system, the more critical is this partitioning—and hence, architecture. And as we will see, the more demanding those quality attributes are, the more critical the architecture is.

A single system is almost inevitably partitioned simultaneously in a number of different ways. Each partitioning results in the creation of an architectural structure: different sets of parts and different relations among the parts. Each is the result of careful design, carried out to satisfy the driving quality attribute requirements and the most important business goals behind the system.

image

Many projects make the mistake of trying to impose a single partition in multiple component domains, such as equating threads with objects, which are equated with modules, which in turn are equated with files. Such an approach never succeeds fully, and adjustments eventually must be made, but the damage of the initial intent is often hard to repair. This invariably leads to problems in development and occasionally in final products.

—Jazayeri, Ran, and van der Linden ([2000](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_091), pp. 16–17)

Architecture is what makes the sets of parts work together as a coherent and successful whole. Architecture documentation help architects make the right decisions; it tells developers how to carry them out; and it records those decisions to give a system’s future caretakers insight into the architect’s solution.

P.3 Architecture Views

Perhaps the most important concept associated with software architecture documentation is that of the [view](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_66). A software architecture is a complex entity that cannot be described in a simple one-dimensional fashion. Our analogy with the bird wing proves illuminating. If you are interested in any but the most superficial understanding, then no single rendition of a bird wing will do. Instead, you need many: feathers, skeleton, circulation, muscular views, and many others. Which of these views *is* the “architecture” of the wing? None of them. Which views *convey* the architecture? All of them.

image

A view is a representation of a set of system elements and the relationships associated with them.

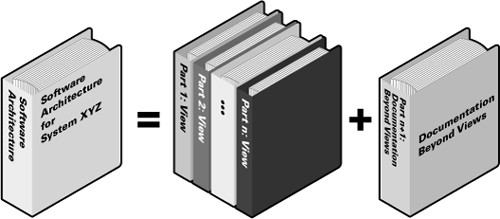
image

For more information about the bird wing analogy, see “[About the Cover](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/pre02.html#pre02)” on page [xxi](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/pre02.html#pre02).

In this book, we use the concept of views to give us the most fundamental principle of architecture documentation, illustrated in [Figure P.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch00.html#ch00fig01):

Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.

Figure P.1 A documentation package for a software architecture can be composed of one or more view documents and documentation that explains how the views relate to one another, introduces the package to its readers, and guides them through it.



image

[Chapter 9](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch09.html#ch09) shows how to choose the relevant views. [Section 10.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10sec1lev1) shows how to document a view, and [Section 10.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10sec1lev2) shows how to document the information that applies to more than one view.

What are the relevant views? It depends on your goals. As we saw previously, architecture documentation can serve many purposes: a mission statement for implementers, a basis for analysis, the specification for automatic code generation, the starting point for system understanding and asset recovery, or the blueprint for project planning.

Different views also expose different quality attributes to different degrees. Therefore, the quality attributes that are of most concern to you and the other stakeholders in the system’s development will affect the choice of what views to document. For instance, a *layered view* will tell you about your system’s portability, a *deployment view* will let you reason about your system’s performance and reliability, and so forth.

image

Layered views are covered in [Section 2.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sec1lev4). Deployment views are covered in [Section 5.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch05.html#ch05sec1lev2).

Different views support different goals and uses. This is fundamentally why we do not advocate a particular view or collection of views. The views you should document depend on the uses you expect to make of the documentation. Different views will highlight different system elements and/or relations.

image

An object-oriented program’s runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program’s runtime structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.

—Gamma et al. ([1995](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_058), p. 22)

It may be disconcerting that no single view can fully represent an architecture. Additionally, it feels somehow inadequate to see the system only through discrete, multiple views that may or may not relate to one another in any straightforward way. The essence of architecture is the suppression of information not necessary to the task at hand, and so it is somehow fitting that the very nature of architecture is such that it never presents its whole self to us but only a facet or two at a time. This is its strength: Each view emphasizes certain aspects of the system while deemphasizing or ignoring other aspects, all in the interest of making the problem at hand tractable. Nevertheless, no one of these individual views adequately documents the software architecture for the system. That is accomplished by the complete set of views along with information that transcends them.

The documentation for a view contains

• A primary presentation, usually graphical, that depicts the primary elements and relations of the view

• An element catalog that explains and defines the elements shown in the view and lists their properties

• A specification of the elements’ interfaces and behavior

• A variability guide explaining any built-in mechanisms available for tailoring the architecture

image

[Section 10.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10sec1lev1) substantially elaborates this outline.

• Rationale and design information

The documentation that applies to all of the views contains

• An introduction to the entire package, including a reader’s guide that helps a stakeholder find a desired piece of information quickly

image

[Section 10.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10sec1lev2) substantially elaborates this outline.

• Information describing how the views relate to one another, and to the system as a whole

• Constraints and rationale for the overall architecture

• Such management information as may be required to effectively maintain the whole package

Coming to Terms: A Short History of Architecture Views

Nearly all modern approaches to designing and documenting architectures rely on the concept of an architectural view. Where did this concept come from?

image

More than three decades ago, David Parnas ([1974](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_114)) observed that software consists of many structures, which he defined as partial descriptions showing a system as a collection of parts and showing some relations among the parts. This definition largely survives in architecture papers today. Parnas identified several structures prevalent in software. A few were fairly specific to operating systems, such as the structure that defines what process owns what memory segment, but others are more generic and broadly applicable. These include the *module structure*, in which the units are work assignments and the relation is *is-a-part-of* or *shares-part-of-the-same-secret-as;* the *uses structure*, in which the units are programs, and the relation is *depends on the correctness of*; and the *process structure*, in which the units are processes, and the relation is *gives computational work to*.

image

Quite a bit later, DeWayne Perry and Alexander Wolf recognized that, similar to building architecture, a variety of views of a system are required. Each view emphasizes certain architectural aspects that are useful to different stakeholders or for different purposes ([Perry and Wolf 1992](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_121)).

image

Later, Philippe Kruchten ([1995](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_095)) of the Rational Software Corporation wrote an influential paper describing four main views of software architecture (logical, process, development, physical) that can be used to great advantage in system building, along with a distinguished fifth view that ties the other four together by showing how they satisfy key use cases: the “4+1” approach to architecture. The 4+1 approach has since been embraced as a foundation piece of the Rational Unified Process.

image

To see how the 4+1 views correspond to views described in this book, see [Section E.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/app01.html#app01sec1lev2) of the epilogue.

image

At about the same time, Dilip Soni, Robert Nord, and Christine Hofmeister of Siemens Corporate Research made a similar observation about views of architecture they found in use in industrial practice ([Soni, Nord, and Hofmeister 1995](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_143)). They wrote about the conceptual view, module interconnection view, execution view, and code view. These views, which correspond more or less to Kruchten’s four views, have become known as the Siemens Four View model for architecture.

image

The Siemens Four View model is explained in the book by Hofmeister, Nord, and Soni ([2000](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_073)).

Other “view sets” have emerged since these. In their book *Software Systems Architecture*, Rozanski and Woods ([2005](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_124)) advocate using functional, information, concurrency, development, deployment, and operational views. Philips Research, the R&D arm of the giant Dutch electronics company, has created the “CAFCR” model of architecture, which calls for five views: the *c*ustomer, *a*pplication, *f*unctional, *c*onceptual, and *r*ealization views.

image

IEEE 1471-2000 is now known as ISO/IEC 42010:2007. We describe this standard in [Section E.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/app01.html#app01sec1lev1) of the epilogue.

In the year 2000, the IEEE adopted a standard (IEEE 1471-2000) for architecture descriptions. Unlike approaches that prescribe a fixed set of views, this standard advocates creating your own views that best serve the stakeholders and their concerns associated with your system. (The Views and Beyond approach also advises flexibility in choosing your view set.)

I.1 Three Categories of Styles

[Chapters 1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01)–[5](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch05.html#ch05) are organized along the lines of the three categories of styles we discussed in the prologue: module styles ([Chapters 1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01) and [2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02)), component-and-connector (C&C) styles ([Chapters 3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03) and [4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04)), and allocation styles ([Chapter 5](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch05.html#ch05)). Plan for your documentation package to include at least one module view, at least one component-and-connector view, and at least one allocation view.

Modules are the primary elements of [module styles](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_35). A module is an implementation unit that provides a coherent set of responsibilities. A module might take the form of a class, a collection of classes, a layer, an aspect, or any decomposition of the implementation unit. Every module has a collection of properties assigned to it. These properties are intended to express the important information associated with the module, as well as constraints on the module. Sample properties are responsibilities, visibility information, and author or owner. The relations that modules have to one another include *is part of*, *depends on*, and *is a*.

image

A module style is a kind of style that introduces a specific set of module types and specifies rules about how elements of those types can be combined.

Module styles are described in [Chapters 1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01) and [2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02).

[Component-and-connector styles](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_13) express runtime behavior. They are described in terms of components and connectors. A component is one of the principal processing units of the executing system. Components might be services, processes, threads, filters, repositories, peers, or clients and servers, to name a few. A connector is the interaction mechanism among components. Connectors include pipes, queues, request/reply protocols, direct invocation, event-driven invocation, and so forth. Components and connectors can be decomposed into other components and connectors. The decomposition of a component may include connectors and vice versa.

image

A component-and-connector style is a kind of style that introduces a specific set of component and connector types and specifies rules about how elements of those types can be combined. Additionally, given that C&C views capture runtime aspects of a system, a C&C style is typically also associated with a computational model that prescribes how data and control flow through systems designed in that style.

C&C styles are described in [Chapters 3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03) and [4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04).

[Allocation styles](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_02) describe the mapping of software units to elements of an environment in which the software is developed or executes. The environment might be the hardware, the file systems supporting development or deployment, or the development organization(s).

1.2.2 Relations

Module views have the following types of relations:

• *Is part of.* The *is-part-of* relation defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole. In its most general form, the *is-part-of* relation simply indicates aggregation, with little implied semantics.

image

In [Chapter 2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02), the *is-part-of* relation is refined to a decomposition relation in the decomposition style.

• *Depends on.* A *depends on* B defines a dependency relation between A and B. Many different specific forms of dependency can be used in module views. Later, we look at four in particular: *uses, allowed to use, crosscuts*, and data entity *relationships*, in the module uses, layered, aspect, and data model styles, respectively. The logical association between classes (in a UML class diagram, for example) also depicts a dependency between the classes.

image

In [Chapter 2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02), the *depends-on* relation is refined to “uses” in the uses style, “allowed to use” in the layered style, and “crosscut” in the aspect style.

• *Is a.* The *is-a* relation defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent. The child is able to be used in contexts in which the parent is used. Later, we look at this relation in more detail in the generalization style. Object-oriented inheritance and interface realization are special cases of the *is-a* relation.

1.4.2 Unified Modeling Language

Software modeling notations, such as UML, provide a variety of constructs that can be used to represent modules. [Figure 1.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01fig02) shows some examples for modules using UML notation. [Figure 1.3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01fig03) shows how the three basic relations native to module views are denoted using UML.

Figure 1.2 Examples of module notation in UML. A module may be represented as a class or a package. More specific types of modules can be indicated with stereotypes (as in [Figure 1.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01fig04)).

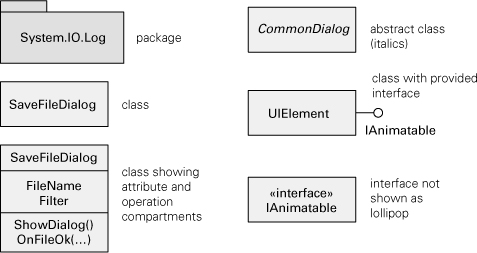
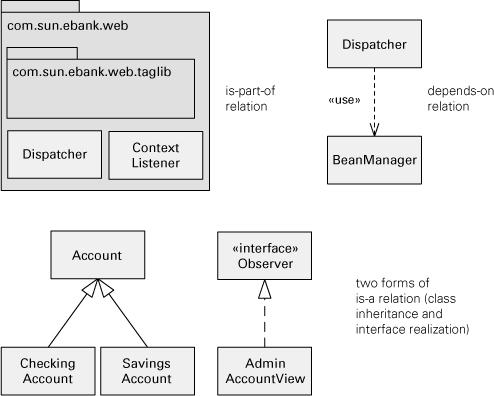


Figure 1.3 Examples of module relations in UML



image

[Appendix A](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/app02.html#app02) describes how UML can be used to show different module views, as well as C&C and allocation styles.

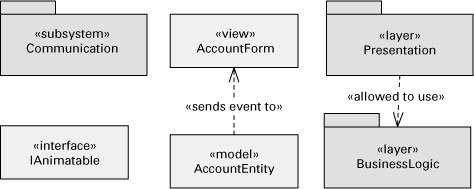
UML has a class construct, which is the object-oriented specialization of a module as described here. UML packages are used to represent an aggregation of modules. UML packages can represent, for example, layers, subsystems, and collections of implementation units that live together in the implementation namespace.

image

Stereotype is a UML extension mechanism that allows the definition of a new type of modeling element or relation based on an existing UML element or relation.

UML was originally created to model object-oriented systems. It is now considered a general-purpose modeling language. As a result, UML elements and relations are generic; that is, they are not specific to implementation technologies or platforms. But you can define stereotypes to specialize the UML symbols. A [stereotype](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_51) is a UML extension mechanism and is represented in diagrams as a label in guillemets («stereotype label»). [Figure 1.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01fig04) shows some examples. If used correctly, stereotypes make your UML diagrams more expressive. The UML specification provides a number of standard stereotypes, but you can also create your own.

Figure 1.4 Examples of UML elements and relations with stereotypes



2.1.1 Overview

By taking the elements and the properties of module views and focusing on the *is-part-of* relation, we get the decomposition style. A decomposition view describes the organization of the code as modules and submodules and shows how system responsibilities are partitioned across them. Almost all architects begin with the decomposition style. Architects tend to attack a problem with divide-and-conquer techniques, and a decomposition view records their campaign.

The criteria used for decomposing a module into smaller modules include:

• *Achievement of certain quality attributes.* For example, to support modifiability, the information-hiding design principle calls for encapsulating changeable aspects of a system in separate modules, so that the impact of any one change is localized.

• *Build-versus-buy decisions.* Some modules may be bought in the commercial marketplace, reused intact from a previous project, or obtained as open-source software. These modules already have a set of responsibilities implemented. The remaining responsibilities then must be decomposed around those established modules.

• *Product line implementation.* To support the efficient implementation of products of a product family, it is essential to distinguish between common modules, used in every or most products, and variable modules, which differ across products.

• *Team allocation*. To allow implementation of different responsibilities in parallel, separate modules that can be allocated to different teams should be defined. The skills of developers also influence the decomposition. For example, if specialized Web developers are available, modules that handle the Web UI should be kept separate.

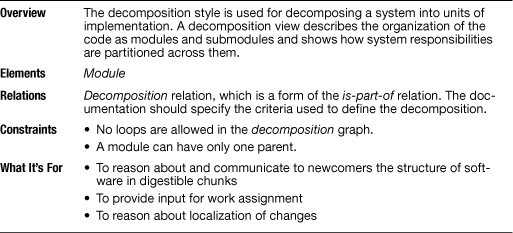
A useful design heuristic holds that a module is small enough if it could be discarded and begun again if the programmer(s) assigned to implement it left the project.

A decomposition view may represent the first pass at a detailed architecture design; the architect may subsequently introduce other types of relations and module specializations. The decomposition view defines the modules that may appear in uses, layered, generalization, and other module-based views.

2.1.2 Elements, Relations, and Properties

[Table 2.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02tab01) summarizes the characteristics of the decomposition style. Elements of the decomposition style are modules, as described in [Section 1.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01sec1lev2). Some modules that aggregate other modules can be called [subsystems](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_54). The principal relation, the *decomposition* relation, is a form of the *is-part-of* relation and has as its primary constraint the guarantee that an element can be a part of at most one aggregate.

Table 2.1 Summary of the decomposition style



image

See “[Coming to Terms: Subsystem](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sb01)” on page [73](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sb01), in this chapter.

image

The element catalog of an architecture view provides various information about the elements in that view. Element catalogs are described in [Section 10.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10sec1lev1).

The module decomposition may define whether the submodules are visible within only the aggregate module—the parent—or also to other modules. The visibility of submodules can be described in the view’s element catalog or conveyed graphically, for example by showing interface lollipops inside or outside the aggregate module, as in [Figure 1.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01fig01).

2.1.6

Adventure Builder

The example software architecture document that accompanies this book online contains an example of a decomposition view for the Adventure Builder ([2010](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_003)) system. See [wiki.sei.cmu.edu/sad](http://wiki.sei.cmu.edu/sad).

The ATIA-M System

Army Training Information Architecture-Migrated (ATIA-M) is a large Web-based, Java EE application that supports training in the U.S. Army. It has “thick clients”: Windows desktop applications developed using .NET (C#) that communicate with the server-side Java EE components using Web services technology.

[Figure 2.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig02) shows the top-level module decomposition for the entire ATIA-M system, itself a module. The code is divided into three large modules:

• *Windowsapps* contains the code of the thick clients. The three submodules correspond to Training and Doctrine Development Tool (TDDT), Unit Training Management Configuration (UTMC), and a separate submodule with common code used by the different Windows applications. TDDT and UTMC were the two Windows applications originally planned, but others could be added.

image

[Figure 2.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig02) is the first of many examples of architecture documentation fragments from real systems. When examining these examples, keep in mind the considerations stated in [Section I.5](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/part01.html#part01sec1lev5), in the introduction to [Part I](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/part01.html#part01). The descriptions of the elements we provide cannot be derived from the figures; rather, they rely on additional documentation that would accompany the diagrams in an architecture document.

• *ATIA server-side Web modules* contains all non-Java modules that would be deployed to server machines. The Web modules include JavaServer Pages (JSP) files, JavaScript and HTML code, and applets.

• *ATIA server-side Java modules* contains all Java source code in ATIA that would run on application servers. This module does not include JSP, JavaScript, HTML, applet, or thick-client code.

The decomposition of Windowsapps into three submodules is shown in [Figure 2.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig02). The decomposition of ATIA server-side Java modules, on the other hand, was captured in another module view diagram, shown in [Figure 2.3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig03).

2.2 Uses Style

2.2.1 Overview

image

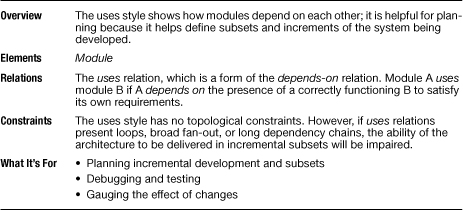
*Uses* is a form of dependency that can exist between two modules. A *uses* B if the correctness of A depends on the presence of a correct implementation of B.

The uses style results when the *depends-on* relation is specialized to [uses](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_61). A module *uses* another module if its correctness depends on the correctness of the other. Whereas the module decomposition style shows only the organization of the implementation units as modules and submodules, a uses style goes one step further to reveal which modules use which other modules. This style tells developers what other modules must exist for their portion of the system to work correctly. This style enables incremental development and the deployment of useful subsets of full systems.

2.2.2 Elements, Relations, and Properties

[Table 2.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02tab02) summarizes the characteristics of the uses style. The elements of this style are the modules as described in [Section 1.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch01.html#ch01sec1lev2). We define a specialization of the *depends-on* relation to be the *uses* relation, whereby one module requires the correct implementation of another module for its own correct functioning. This view makes explicit which modules use which other modules to achieve their responsibilities.

Table 2.2 Summary of the uses style



2.2.3 What the Uses Style Is For

This style is useful for planning incremental development, system extensions and subsets, debugging and testing, and gauging the effects of specific changes. [Figure 2.6](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig06) shows the primary presentation of a uses view and how it can help with incremental development. To define incremental subsets, modules should be defined at the right level of granularity. In the example, admin.core may not need the entire dao package, only a submodule of it; the diagram should then show the submodules of dao.

Figure 2.6 In this uses view, suppose the incremental development plan called for module admin.client in the next release. Based on the *uses* relation, the diagram highlights what other modules need to be present: admin.core, dao, and util.

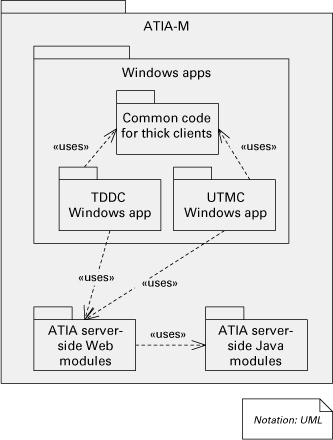
Adventure Builder

The example software architecture document accompanying this book online contains an example of a uses view for the Adventure Builder ([2010](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_003)) system. See [wiki.sei.cmu.edu/sad](http://wiki.sei.cmu.edu/sad).

The ATIA-M System

[Figure 2.10](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig10) shows the diagram from a top-level uses view for the ATIA-M system (it also shows decomposition). In the architecture documentation, it could have superseded the decomposition view (see [Figure 2.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig02)) for the same system.

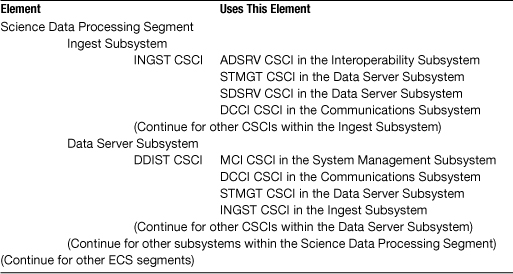
Figure 2.10 Top-level uses view for the ATIA-M system



ECS

EOSDIS Core System (ECS) is a NASA system. A constellation of satellites collect measurements about Earth and send the data to ground stations. ECS controls spacecraft and instruments, processes data, and produces refined data that are stored in several distributed data centers and made available to scientists around the world. [Figure 2.11](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig11) is a small excerpt of a uses view’s primary presentation from the ECS system. The notation is textual, using the tabular format mentioned earlier. Like most primary presentations, this one names only the elements; they are defined in the view’s supporting documentation (not shown here).

Figure 2.11 Excerpt of the ECS system uses view, documented as a table. The left column mirrors the system’s module decomposition structure.



Coming to Terms: Uses

Two of the module styles that we present in this book—the uses style and the layered style—are based on one of the most underutilized relations in software engineering: *uses*. The *uses* relation is a form of the *depends-on* relation. A unit of software P1 is said to use another unit P2 if P1’s correctness depends on a correct implementation of P2 being present.

The *uses* relation resembles, but is decidedly not, the simple *calls* relation provided by most programming languages. Here’s why.

• A program P1 can use program P2 without calling it. P1 may assume, for example, that P2 has left a shared device in a usable state when it finished with it. Or P1 may expect P2 to leave a computed result that it needs in a shared variable. Or P1 may be a process that sleeps until P2 signals an event to awaken it.

• A program P1 might call program P2 but not use it. If P2 is an exception handler that was passed as a parameter[1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/footnotes.html#ch02fn01) for P1 to call when it detects an error, P1 will usually not care what P2 does. P1 does not use P2 because its own correctness does not depend on P2.

So *uses* is not *calls* or *invokes*. Likewise, *uses* is different from other *depends-on* relations, such as *includes*, which deals with compilation dependencies but need not influence runtime correctness.

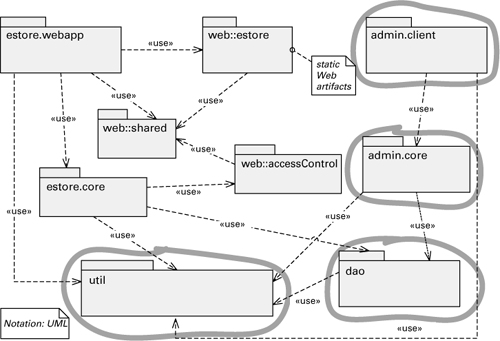
Because the *uses* relation takes many forms, a uses view usually cannot be automatically derived from other architecture views nor extracted from source code. To enjoy its benefits, the architect must engineer the relations and document the uses view explicitly.

The careful engineering of the *uses* relation imparts a powerful capability to a development team: It enables the building of small subsets of a total system. Early in the project, this allows incremental development, a development paradigm that allows early prototyping, early integration, and early testing. At every step along the way, the system carries out part of its total functionality, even if far from everything, and does it correctly. Fred Brooks ([1995](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_027)) writes about the “electrifying effect” on team morale when the system first succeeds at doing something. Absent incremental development, nothing works until everything works, and we are reduced to the waterfall model of development. Subsets of the total system are also useful beyond development. They provide a safe fallback in the event of slipped schedules: It is much better for the project manager to offer the customer a working subset of the system at delivery time rather than apologies and promises.

Here’s how it works. Choose a program that is to be in a subset; call it P1. In order for P1 to work correctly in this subset, correct implementations of the programs it uses must also be present. So include them in the subset. For them to work correctly, their used programs must also be present, and so forth. The subset consists of the transitive closure of P1’s uses.[2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/footnotes.html#ch02fn02) Conceptually, you pluck P1 out from the uses graph and then see what programs come dangling beneath it. There’s your subset.

Loops in the relation—that is, for example, where P1 uses P2, P2 uses P3, and P3 uses P1—are the enemy of simple subsets. A large uses loop necessitates bringing in a large number of programs—every member of the loop—into any subset joined by any member. “Bringing in a program” means, of course, that it must be implemented, debugged, integrated, and tested. But the point of incremental development is that you’d like to bring in a small number of programs to each new increment, and you’d like to be able to choose which ones you bring in and not have them choose themselves. Generally speaking, any long list of used programs (caused by long dependency chains or broad fan-out in the relation) detracts from the ability to field small increments. They also decrease modifiability, because a change to a module could very well ripple into modules that it uses.

Besides managing subsets, the *uses* relation is also a helpful tool for debugging and integration testing. If you discover a program that’s producing incorrect results, the problem is going to be either in the program itself or in the programs that it uses. The *uses* relation lets you instantly narrow the list of suspects. In a similar way, you can employ the relation to help you gauge the effects of proposed changes. If a program’s external behavior changes as the result of a planned modification, you can backtrack through the *uses* relation to see what other programs may be affected by that modification.



The uses view also helps in managing the dependencies of a system that is being built or maintained. The goal of this task is to keep complexity under control and avoid degradation in the modifiability of the system due to the addition of undesirable dependencies.

2.4 Layered Style

2.4.1 Overview

The layered style, like all module styles, reflects a division of the software into units. In this case, the units are [layers](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_33). Each layer represents a grouping of modules that offers a cohesive set of services. There are constraints on the *allowed-to-use* relationship among the layers: the relations must be unidirectional. The layered view of architecture, shown with a layer diagram, is one of the most commonly used views in software architecture. However, it often is poorly defined, and so often misunderstood. Because true layered systems promote modifiability and portability, architects have an incentive to show their systems as layered, even if they are not.

image

A layer is a grouping of modules that together offer a cohesive set of services to other layers. The layers are related to each other by the strictly ordered relation *allowed to use*.

Layers completely partition a set of software, and each partition—through a public interface—provides a cohesive set of services. But that’s not all. [Figure 2.16](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig16), which is intentionally vague about what the units are and how they interact, shows three divisions of software—you’ll have to take our word that each division provides a cohesive set of services—but none of them constitutes a layering. What’s missing?

Figure 2.16 Three different divisions of software. Is any of them layered?



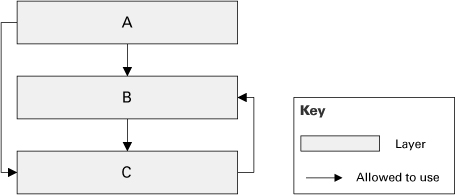
Layering has one more fundamental property: The layers are created to interact according to a strict ordering relation. Herein lies the conceptual heart of layers. If (A, B) is in this relation, we say that the implementation of layer A is allowed to use any of the public facilities provided by layer B.

image

Element A uses element B if A’s correctness depends on a correct implementation of B being present.

By [uses](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_61), we mean the very specific term defined in [Section 2.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sec1lev2) for the uses style, but the definition has some loopholes. If A is implemented using the facilities in B, is it implemented using only B? Maybe or maybe not. For example, assume that layers are depicted horizontally, one on top of the other. Some layering schemes allow a layer to use the public facilities of *any* lower layer, not just the nearest lower layer. Other layering schemes have so-called layers that are collections of utilities and can be used by any layer. *But no architecture that can be validly called layered allows a layer to use, without restriction, the facilities of a higher layer.* Allowing unrestricted upward usage destroys the desirable properties that layering brings to an architecture; this will be discussed shortly. Usage in layers generally flows downward. A small number of well-defined special cases may be permitted, but these should be few and regarded as exceptions to the rule. Hence, the architecture in [Figure 2.17](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig17) *resembles* a layering *but is not*.

Figure 2.17 There may be three layers here, but this is not a design in the layered style, which forbids upward uses.



[Figure 2.17](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig17) shows why layers have been a source of ambiguity for so long: architects have been calling such diagrams layered when they are not. There is more to layers than the ability to draw separate parts on top of each other.

image

Remember that a system with a *uses* relation from a lower layer to a higher layer is not a layered system, strictly speaking.

In some cases, modules in a very high layer might be required to directly use modules in a very low layer where normally only next-lower-layer uses are allowed. The layer diagram or an accompanying document will have to show these exceptions. The case of software in a higher layer using modules in a lower layer that is not just the next lower layer is called *layer bridging*. If many of these are present, the system is poorly structured, at least with respect to the portability and modifiability goals that layering helps to achieve. Systems with upward usages are not, strictly according to the definition, layered. However, in such cases, the layered style may represent a close approximation to reality and also conveys the ideal design that the architect was trying to achieve.

Layers cannot be derived by examining source code. Layers are logical groupings that are wonderful aids in creating and communicating the architecture, but often they are not explicitly delimited in the source code. The source code may disclose what uses what, but the relation in layers is *allowed to use*.

Some of the criteria used in defining the layers of a system are an expectation that they will evolve independently on different time scales, that different people with different sets of skills will work on different layers, and that different levels of reuse are expected of the different layers.

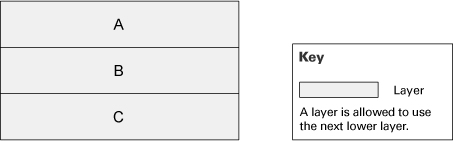
2.4.4 Notations for the Layered Style

Informal Notations

*Stack*

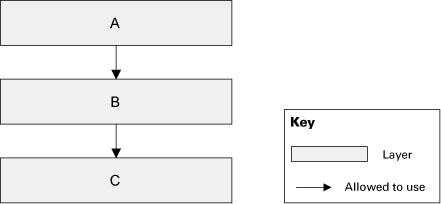
Layers are almost always drawn as a stack of boxes. The *allowed-to-use* relation is denoted by geometric adjacency and is read from the top down, as in [Figure 2.18](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig18) (note that the key could have said, “A layer is allowed to use any lower layer”).

Figure 2.18 Stack of boxes notation for layered designs



Layering is thus one of the few architecture styles in which connection among components is shown by geometric adjacency and not an explicit symbology, such as an arrow, although arrows can be used, as in [Figure 2.19](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig19).

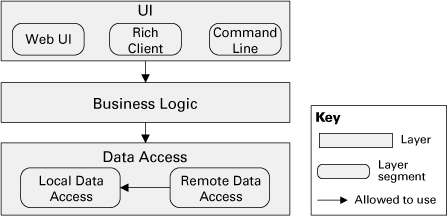
Figure 2.19 Layered design with *allowed-to-use* relations shown with arrows



*Segmented Layers*

Sometimes layers are divided into segments denoting a finer-grained aggregation of the modules. Often, this occurs when a preexisting set of units, such as imported modules, share the same *allowed-to-use* relation. When this happens, the creator of the diagram must specify what usage rules are in effect among the segments. Many usage rules are possible, but they must be made explicit. In [Figure 2.20](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig20), the top and the bottom layers are segmented. Segments of the top layer are not allowed to use each other, but segments of the bottom layer are. If you draw the same diagram without the arrows, it will be harder to differentiate the usage rules within segmented layers. Layered diagrams are often a source of ambiguity because the diagram does not make explicit the *allowed-to-use* relations.

Figure 2.20 Layered design with segmented layers

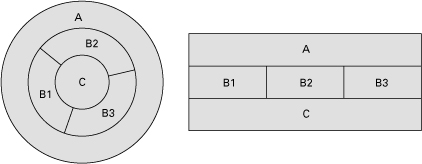


*Rings*

A notational variation is to show layers as a set of concentric circles, or rings. The innermost ring corresponds to the lowest layer; the outermost ring, the highest layer. A ring may be subdivided into sectors, meaning the same thing as the corresponding layer being segmented.

There is no semantic difference between a layer diagram that uses a stack of rectangles and one that uses the rings paradigm, except when segmented layers have restrictions on the *allowed-to-use* relation within the layer. In [Figure 2.21](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig21), assume that ring segments that touch are allowed to use one another and that layer segments that touch are allowed to use one another. You cannot “unfold” the ring diagram to produce a stack diagram, such as the one on the right, with exactly the same meaning, because circular arrangements allow more adjacencies than do linear arrangements. (In the layer diagram, B1 and B3 are separate; in the ring diagram they are adjacent.) Cases like this are the only ones in which a ring diagram can show a geometric adjacency that a stack picture cannot.

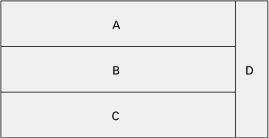
Figure 2.21 A layered design shown as concentric rings and as a stack of boxes. Are these two representations equivalent?



*Layers with a Sidecar*

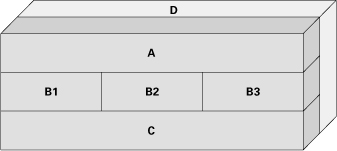
Many architectures that are described as layered look something like [Figure 2.22](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig22). This type of notation could mean one of two things: (1) Modules in D can use modules in A, B, or C. (2) Modules in A, B, or C can use modules in D. (Technically, the diagram might mean that both are true, although this would arguably be a poor layered architecture.) The creator of the diagram must specify which usage rules pertain. A variation like this makes sense only for single-level usage rules in the main stack, that is, when A can use only B and nothing below. Otherwise, D could simply be made the bottommost layer in the main stack, and the “sidecar” geometry would be unnecessary.

Figure 2.22 Layers with a “sidecar.” The key should make clear what is allowed to use and be used by software in the box on the side.



In some cases, the layered architecture is depicted as a three-dimensional figure, to represent a layer that is accessible to all other layers, as shown in [Figure 2.23](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig23).

Figure 2.23 Three-dimensional layered diagram trying to show that layer D can be used by all other layers. The picture could just as well be showing that D can use all other layers. The ambiguity should be resolved by an annotation, or in the key.



Such layers on the side often represent utility libraries or platform services (such as the operating system or runtime environment).

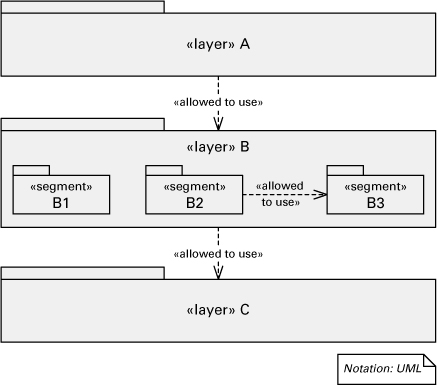
*Size and Color*

Sometimes layers are colored to denote which team is responsible for them or to denote another distinguishing feature. Sometimes layers use different colors just to improve readability. Size is sometimes used to give a vague idea of the relative size of the modules constituting the various layers. If they carry meaning, size and color should be explained in the key accompanying the layer diagram.

*UML*

UML has no built-in primitive corresponding to a layer. However, layers can be represented in UML as stereotyped packages, as shown in [Figure 2.24](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig24). A package is a general-purpose mechanism for organizing elements into groups, and it suits the notion of layers. The *allowed-to-use* relation can be a stereotyped dependency between layer packages.

Figure 2.24 Documenting segmented layers in UML. If segments in a layer are allowed to use each other, then <<allowed to use>> dependencies must be added among them as well.



image

[Appendix A](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/app02.html#app02) discusses how to use UML classes and packages to represent layers and more.

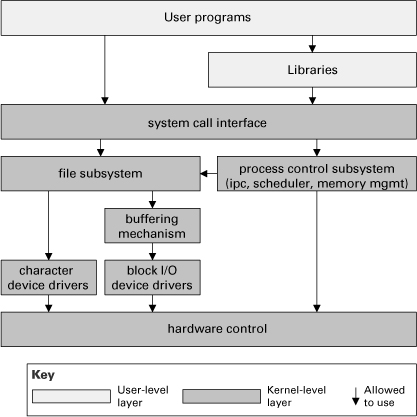
Access dependencies are not transitive. If package 1 can access package 2 and package 2 can access package 3, it does not automatically follow that package 1 can access package 3.

2.4.6 Examples Using the Layered Style

UNIX System V

A classic layered design is the UNIX System V operating system, as shown in [Figure 2.26](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig26). The lower layers form the system kernel; top layers are user programs or libraries that access the kernel through system calls. The system call interface layer isolates the kernel implementation details and provides a virtual machine to user programs. The file subsystem is responsible for managing files (devices are treated as files), administering free space, controlling access, and reading/writing data. The process control subsystem is responsible for process scheduling, interprocess communication, process synchronization, and memory management. The hardware control layer is responsible for handling interrupts and communicating with the machine.

Figure 2.26 The primary presentation of a layered view of the UNIX System V operating system implementation (adapted from [Bach 1986](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_015))



image

This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level.

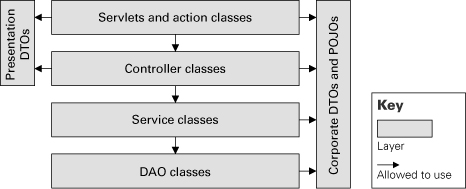
—H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs* ([1996](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_001))

This design is presented in [Chapter 2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02) of the classic book by Maurice Bach, *The Design of the UNIX Operating System* ([Bach 1986](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_015)), where a candid observation is made: “The diagram serves as a useful logical view of the kernel, although in practice the kernel deviates from the model because some modules interact with the internal operations of others.” All such exceptions should be noted in your documentation.

Java EE Application

[Figure 2.27](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig27) is the primary presentation of the layered view of a set of integrated, multi-tier, Web-based applications that use the Java EE platform. All user operations in these applications follow this layered design. The topmost layer has presentation classes, which are servlets and JavaServer Faces (JSF) action classes. Servlet and JSF are Java component technologies for developing Web components. The second layer has controller classes, which implement the sequence of steps to carry on the functionality of a use case. An example of a controller class is CtlRetrievePtoDays. Controller classes interact with business service classes, which encapsulate the core business logic associated with domain objects. An example of a service class is SvcFullTimeEmployee. The lowermost layer has data access objects. These modules handle all interaction with the relational database.

Figure 2.27 Part of the layered view of a set of Java EE applications. The top layer has servlets and JSF action classes responsible for the user interface. Controller classes handle the user operations by interacting with business service classes. Access to the database is done in the lowermost layer with the data access objects. Sidecar layers contain DTOs and POJOs that are used by the other layers to hold and transfer data.



There are two sets of auxiliary modules that are presented as sidecar layers. On the left are presentation data transfer objects (DTOs). They are simple classes that contain basic attributes corresponding to data elements required in different user screens. The right sidecar layer has the corporate DTOs and plain old Java objects (POJOs). Like presentation DTOs, these classes have a set of attributes to hold data. In this design, DTOs have attributes required by a particular transaction, whereas POJOs correspond to data entities stored in the database.

The key drivers for this layered design are modifiability and portability, which is achieved with separation of concerns. On top is the presentation layer. Changes to the user interface are addressed in that layer. If the technology used to implement the UI has to change from servlet and JSF to, say, Google Web Toolkit and Flash, this layer has to be rewritten, but the other layers should remain unchanged. The second layer implements the logic to handle the user actions by wiring the calls to services in the third layer, which is the core business logic layer. The bottom layer isolates database access operations and also enhances portability. If the application is migrated to a different database management system with a different SQL dialect, all modifications required would be confined to that layer.

Coming to Terms: Virtual Machines

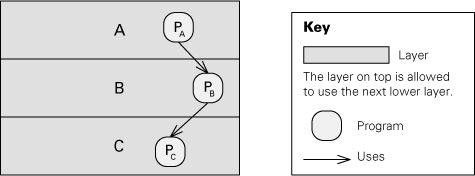
A virtual machine, sometimes called an *abstract machine*, is a collection of modules that form an isolated, cohesive set of services that can execute programs. Early use of the term referred to a more abstract stand-in for a real computer, but current use includes virtual machines that have no direct correspondence to any real machine. Interpreters are good examples of virtual machines. The Common Language Runtime (CLR) of the Microsoft .NET platform is an example of a virtual machine. It provides services to execute bytecode produced by compiling C# or other .NET programming languages. The CLR converts the bytecode into code that is native to the operating system underneath. The Java Virtual Machine (JVM) does the same thing for the Java language. An operating system itself is a virtual machine that allows the execution of native code on the underlying hardware. Thus, a virtual machine is a software layer that can execute “programs,” which can be sequences of calls to facilities of the virtual machine’s interface. Hence some authors regard layers and virtual machines as synonyms.

Perspectives: Calling Higher Layers

We have been emphatic in saying that upward uses invalidate layering. We made allowances for documented exceptions but implied that too many of those would get you barred from the Software Architect’s Hall of Fame.

Seasoned designers, however, know that in many elegantly designed layered systems, all kinds of control and information flow upward along the chain of layers, with no loss of portability, reusability, modifiability, or any of the other qualities associated with layers. In fact, one of the purposes of layers is to allow for the “bubbling up” of information to the units of software whose scope makes them the appropriate handlers of the information. One approach to error handling illustrates this upward flow. Suppose that we have a simple three-layer system, as in [Figure 2.28](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig28). Say that program PA in A uses program PB in B, which uses program PC in C. If PC is called in a way that violates its specification, PC needs a way to tell PB, “Hey! You called me incorrectly!” At that point, (1) PB can either recognize its own mistake and call PC again, this time correctly, or take another action; or (2) PB can realize that the error resulted because it was called incorrectly—perhaps it received bad data—by PA. In the latter case, PB needs a way to tell PA, “Hey! You called me incorrectly!”

Figure 2.28 Layered design showing programs inside and their usage dependencies



Callbacks are a mechanism to manifest the protestation. We do not want PC written with knowledge about programs in B or PB written with knowledge about programs in A, as this would limit the portability of layers C and B. Therefore, the names of higher-level programs to call in case of error are passed downward as parameters. Then the specification for, say, PB includes the promise that in case of error, it will invoke the program whose name has been made available to it.

Other situations where callbacks can be used include:

• When PA uses PB to obtain data to present in the user interface but PA also wants PB to announce future changes to the data. In other words, PA subscribes to events that can be emitted by PB and provides to PB the name of the operation that will handle the events.

• When PA uses PB and the interaction is asynchronous, but PA needs to receive a response once PB is done processing the request. In this case PA provides PB the name of the operation to call.

So there we have it: data and control flowing downward and upward in an elegant error-handling scheme that preserves the best qualities of layers. So much for our prohibition about upward uses. Right?

Wrong. Upward uses are still a bad idea, but the scheme we just described doesn’t have any. It has upward data flow and upward invocation but not uses. The reason is that once a program calls its error handler, its obligation is discharged. The program does not *use* the error handler, because its own correctness depends not a whit on what the error handler does. This is how the callback mechanisms, built in to some programming languages, work and still allow true layered systems to be written in those languages.

Although this may sound like a mere technicality, it is an important distinction. *Uses* is the relation that determines the ability to reuse and to port a layer; “calls” or “sends data to” is not. Architects need to know the difference and need to convey the precise meaning of the relations in their architecture documentation.

—P.C. and P.M.

Perspectives: Using a DSM to Maintain a Layered Architecture

Tools based on the dependency structure matrix claiming to be the solution to managing complexity in large software projects have recently been capturing the attention of program analysts and software architects. The DSM concept has been adopted for use in software engineering from its origins with Donald Steward as the Design Structure System ([Steward 1981](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_146)), which he devised in 1967 to help manage complexity in the nuclear power industry. Over the past 15 years the DSM has been used in a wide variety of industries to aid in systems engineering and analysis as well as project planning and management.

When layer A depends on layer B and layer B depends on layer A, there is a codependence between these two layers, a situation that is forbidden in a layered architecture. In a DSM, circular dependencies are immediately visible as marked cells on both sides of the matrix’s diagonal. A layered architecture is clearly discernable because the corresponding DSM is a lower triangular matrix (that is, one in which all the marked cells are below the diagonal). For example, consider the layered architecture in [Figure 2.29](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig29). The key indicates that a layer is allowed to use only the next lower layer, so it’s a strictly layered design. The corresponding DSM is shown in [Figure 2.30(a)](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig30). If a layer were allowed to use any lower layer, the DSM would be similar to [Figure 2.30(b)](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig30). When cells above the diagonal are marked, the architect can see the circular dependency and focus on what to change to reach the goal of a layered architecture.

Figure 2.29 Simple layered architecture

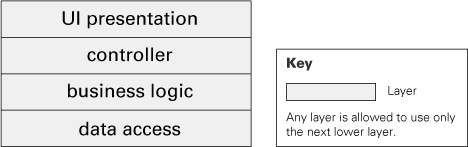
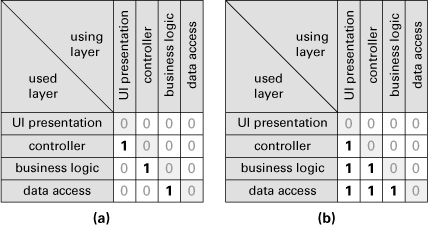


Figure 2.30 DSM showing (a) strictly layered design and (b) layered design



In practice, layered designs are more complex. [Figure 2.31](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig31) shows the layered design that was introduced in [Figure 2.27](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig27), now with Java packages added for each layer. The DSM for this design is shown in [Figure 2.32](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig32). In a DSM tool, the architect can mark the dependencies that violate the layered design: the highlighted cells above and below the diagonal in [Figure 2.32](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig32). During the implementation of the system, the tool can create a DSM from the code and highlight any violations. If other constraints on interdependencies have been indicated by the architect, those will also be visible using the DSM representation. With good tool support, continuous integration builds can be subjected to DSM analysis, and architecture violations can be caught immediately. DSM tools also generally allow the user to perform “what-if” analysis by simulated restructuring of the system, providing immediate insight into the impact that a suggested change would have on the system’s structure.

Figure 2.31 Layered design showing Java packages for each layer

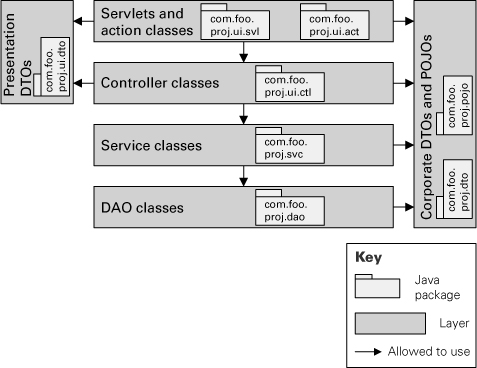
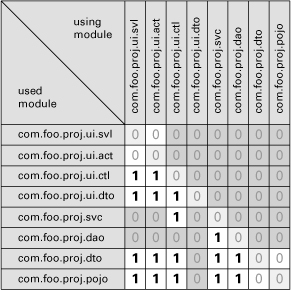


Figure 2.32 DSM for a layered design. The highlighted cells above and below the diagonal represent dependencies that are not allowed



—J.S. and P.M.

2.6 Data Model

2.6.1 Overview

Data modeling is a common activity in the software development process of information systems. The output of this activity is the data model, which describes the static information structure in terms of data entities and their relationships. For example, in a banking system, entities typically include Account, Customer and Loan. Account has several attributes, such as account number, type (savings or checking), status, and current balance. A relationship may dictate that one customer can have one or more accounts, and one account is associated to one or two customers. The data model is often represented graphically in entity-relationship diagrams (ERDs) or UML class diagrams.

The first draft of an architecture view typically has very little detail. Over time, as design decisions are made, the view is elaborated until the architect considers there’s enough information captured in that architecture view. The same thing happens with the data model. Data modeling spans the evolution of the high-level model that displays the data entities in a given business domain into a model that shows details of how the data is stored, for example, in a relational database management system. As a result, different organizations focus the modeling and documentation effort on different stages of the data model evolution. Thus organizations sometimes use qualifiers to the data model to distinguish these stages. Examples of qualifiers include:

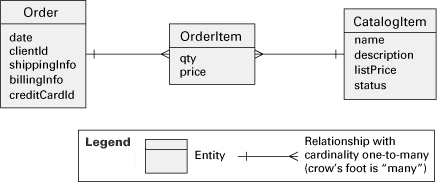
• *Conceptual.* The conceptual data model abstracts implementation details and focuses on the entities and their relationships as perceived in the problem domain. [Figure 2.38](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig38) shows a fragment of a conceptual data model.

Figure 2.38 First draft of a conceptual data model. This and the next two diagrams are fragments of an online order-processing system at different stages.



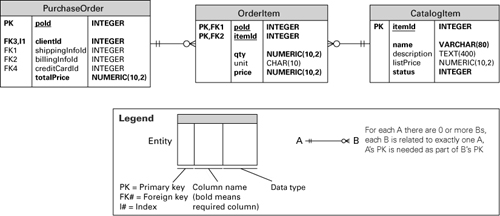
• *Logical*. The logical data model is an evolution of the conceptual data model toward a data management technology (such as relational databases). It is typically the subject of normalization (see [Section 2.6.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sec2lev32)). [Figure 2.39](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig39) shows an example of a logical data model.

Figure 2.39 Logical data model that has evolved from the conceptual data model in [Figure 2.38](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig38)



• *Physical*. The physical data model is concerned with the implementation of the data entities. It incorporates optimizations that may include partitioning or merging entities, duplicating data, and creating identification keys and indexes. For example, in [Figure 2.40](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig40) a column named total-Price was likely added to the entity Order as a performance optimization, since the total price could also be obtained by reading all order items and adding up their prices.

Figure 2.40 Physical data model that was created by adding implementation details and optimizations to the logical data model in [Figure 2.39](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig39)

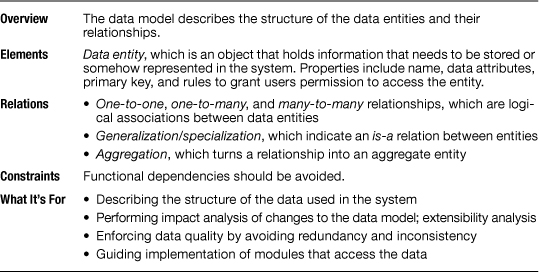


In an early stage, the architecture documentation may contain the data model with the key entities and important relationships. Later on, this initial model is superseded by the detailed model approved by the data administrators.

2.6.2 Elements, Relations, and Properties

[Table 2.6](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02tab06) summarizes the characteristics of the data model style.

Table 2.6 Summary of the data model style



image

See “[Coming to Terms: Entity](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sb07)” on page [118](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sb07), in this chapter.

The elements in a data model are called data entities or simply entities. Any distinguishable object that contains information to be stored or represented in the system can be an entity.

Properties of entities may include:

• Name of the entity.

• Description of the meaning and significance of the entity.

• List of data attributes of the entity. For example, a Car entity may have attributes year, manufacturer, model, mileage, price, and license. Each attribute may have properties, such as data type, size, and whether it’s a required attribute or not.

• The attribute (or attributes) used to uniquely identify an entity (that is, the primary key).

• Whether an entity is weak. A weak entity, also known as a dependent entity, depends on the existence of another entity to exist. For example, an OrderItem requires the existence of a PurchaseOrder in [Figure 2.40](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig40).

• Constraints and invariants on the values of individual or combined attributes. For example, “Returning date cannot be prior to arrival date.”

• Rules that will be used to grant permissions to users or user groups to access the entity.

• Expected number of entity instances and growth rate.

Other properties concern the physical data model and are specific to the target implementation platform of the data model. Examples include:

• List of attributes that should be indexed to optimize access time.

• List of attributes that should be encrypted or compressed.

• Whether the entity should become a database view instead of a table. A view is a virtual table that is defined by a SQL query command on one or more tables.

• Whether the entity should become a materialized view, which means it will be implemented as a database table that stores a subset of the data copied from a master table. Like a regular view, the subset is defined by a query command.

• List of database triggers that will be implemented for that entity. A trigger is a special procedure that is automatically executed by the database management system when data is inserted, updated, or deleted.

There are three types of relations found in data models:

• *Relationship*. Used to designate a logical association between entities. It is usually qualified by the cardinality of the participant entities: one-to-one, one-to-many, or many-to-many. In addition, a relationship can be *identifying* or *nonidentifying*. An identifying relationship from A to B means that the existence of B depends on the existence of A; that is, the primary key of B contains the primary key of A.

• *Generalization/specialization.* Indicates an *is-a* relation between entities. For example, entity Insurance is a generalization of different types of insurances; at the same time, entities Car Insurance and House Insurance are specializations of entity Insurance.

• *Aggregation*. An abstraction that turns a relationship between entities into an aggregate entity ([Smith and Smith 1977](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_141)). For example, a relationship between a patient, a physician, and a date can be abstracted as an aggregate entity called Appointment. In practice, this relation is rarely used.

image

For an explanation of the normalization technique and description of the various normal forms, refer to the classic book by C. J. Date, *An Introduction to Database Systems* ([1999](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_041)).

Conceptually, there are no topological constraints with respect to the relations in a data model. However, the database normalization technique imposes restrictions on the data model based on the dependencies between entity attributes. Normalization is used by data administrators to avoid duplication of information, in order to safeguard the consistency (integrity) of the data. [Figures 2.41](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig41) and [2.42](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02fig42) show an example of normalization.

Figure 2.41 Entity ProjectAssignment before normalization, along with sample data (adapted from [Ponniah 2007](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/bib.html#bib01_122)). The attributes that uniquely identify a project assignment (that is, the primary key) are EmpId and ProjNo.

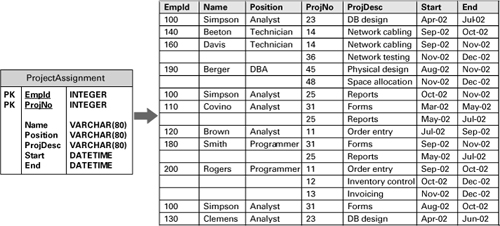
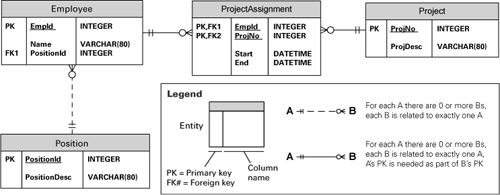


Figure 2.42 Data model for ProjectAssignment after normalization. One of the rules of normalization is that non-key attributes should have functional dependencies to the whole primary key only. Attribute ProjDesc has a functional dependency to ProjNo, which is not the whole primary key. After this and other violations of the normalization rules were fixed, this is the resulting data model diagram.



2.6.3 What the Data Model Is For

The data model facilitates stakeholder communication during domain analysis and requirements elicitation. But foremost, the data model is the blueprint for the implementation of the data entities, for example, in a relational database.

A carefully created data model also helps to achieve performance requirements in a software system. In data-centric applications, access to the data usually represents a significant amount of the time to process user requests. The architect and the data administrator should understand what kinds of data access operations will be more critical to the system and what their performance requirements are. Driven by these requirements, denormalizations, optimizations, and other design decisions are applied to the data model, aiming at improved system performance. Examples of these design decisions include:

• Merging two entities to avoid an expensive outer join or union operation in a query

• Adding a derived attribute to avoid scanning an entire data table to obtain the derived value

• Creating an index on attributes that are often parameters in a query

• Changing the granularity (such as table row or page) and type (such as optimistic) of locks on certain entities to avoid contention and deadlocks

After the software system is implemented, even when the data model is carefully created, it’s common to find performance bottlenecks in data access operations. To remove these bottlenecks, the data model comes in handy once again, in a task called query optimization.

In information systems, the data model is essential input to modifiability analysis. To analyze the impact of required modifications to a system, one cannot look exclusively at the code structure. Many modifications require altering the data model and hence its physical implementation. Modifications to the data model can be costly, as they may require changing the code of multiple applications that share the same data. A simple change such as making a certain attribute of an entity mandatory (for example, requiring a customer’s date of birth) may require changes to all screens and functions that allow creating or updating that information. Versioning and redeployment of applications is more complicated when data model changes are involved. Moreover, larger data model modifications, such as merging with the data model of a legacy system, may also require the implementation of extract, transform, and load (ETL) operations to fix the data itself. Indeed, the data model is an important input to data warehouse projects and to the integration of data schemas required by some business partnerships (for example, an airline company needs to share data with a car rental company).

The data model is an architecture view that should ideally be created with a thorough understanding of incremental development plans, future extensions, and integration of data across information systems. Data is a valuable asset, and the existence of an enterprise data model and a data administration group helps to enforce [data integrity](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_17). If a new system needs to retrieve sales information, the enterprise data model may already contain that information. The architect of the new system may not be aware of the data entities that hold sales information, but the data administrator should and can point out those entities instead of creating new ones in the database. Disparate, redundant data contribute to poor data quality.

image

Data integrity refers to the consistency and accuracy of the data shared across all applications in a system.

Based on the data model, data modeling tools can generate scripts to create the physical database. Some tools can also generate application code to access the data tables, classes to hold the data, forms for end users to enter data, message schemas, and simple reports.

Finally, the data model can help application developers to write code to access the database. It is easier to understand an entity-relationship diagram than to browse through the table creation commands or the database management system dictionary.

3.1 Overview

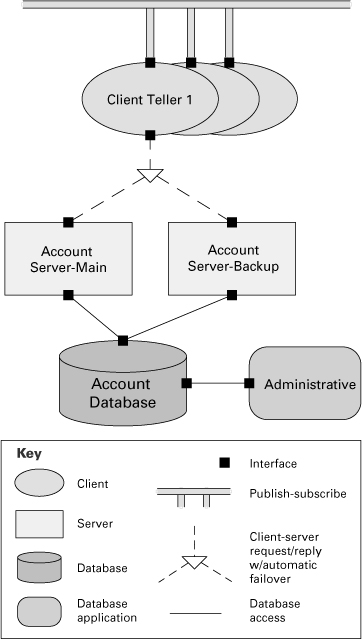
In this chapter we discuss C&C views in their most general form, and we look at notations for representing C&C views. In [Chapter 4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04), we explore some important C&C styles.

A C&C view shows elements that have some runtime presence, such as processes, objects, clients, servers, and data stores. These elements are called [components](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_14). Additionally, component-and-connector views include as elements the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage. Such interactions are represented as [connectors](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_15) in C&C views.

Component-and-connector views are ubiquitous in practice; indeed, box-and-line diagrams depicting these views are often the graphical medium of choice as a principal first-look explanation of the architecture of a system. But such informal C&C views can be misleading, ambiguous, and inconsistent. Some problems follow from the usual pitfalls of visual documentation and are equally applicable to any of the view types discussed in this book. Other problems derive specifically from the use of components and connectors to portray a system’s execution structure. In this chapter, we provide guidelines for documenting C&C views, and we highlight common pitfalls.

[Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) illustrates the primary presentation of a C&C view of a system’s runtime architecture. What is this diagram (and the documentation that explains it) attempting to convey? It shows a picture of the system as it appears at runtime. The system contains a shared repository of customer accounts (Account Database) accessed by two servers and an administrative component. A set of client tellers can interact with the account repository servers, embodying a client-server style. These client components communicate among themselves by publishing and subscribing to events. The purpose of the two servers is to enhance availability: If the main server goes down, the backup can take over. Finally, an administrative component allows an administrator to access and maintain the shared-data store.

Figure 3.1 A bird’s-eye-view of a system as it appears at runtime. This system contains a shared repository that is accessed by servers and an administrative component. A set of client tellers can interact with the account servers and communicate among themselves through a publish-subscribe connector.



image

The primary presentation is the (typically) graphical portion of an architecture view. Documentation that explains the primary presentation is called supporting documentation. Both are described in [Chapter 10](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10).

Each of the three types of connectors shown in [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) represents a different form of interaction among the connected parts.

• Client-server connectors allow a set of concurrent clients to retrieve data synchronously via service requests. This variant of the client-server style supports transparent failover to a backup server.

image

The system illustrated in [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) is built from an amalgamation of different styles: client-server is described in [Section 4.3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04sec1lev3); the shared-data style is described in [Section 4.5](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04sec1lev5); and publish-subscribe is described in [Section 4.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04sec1lev4). This picture is a result of combining views, which is discussed in [Section 6.6](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch06.html#ch06sec1lev6).

• The database access connector supports transactional, authenticated access for reading, writing, and monitoring the database.

• The publish-subscribe connector supports asynchronous event announcement and notification.

Each of these connectors represents a complex form of interaction and will likely require nontrivial implementation mechanisms. For example, the client-server connector type represents a protocol of interaction that prescribes how clients initiate a client-server session, how and when failover is achieved, and how sessions are terminated. Implementation of this connector will probably involve runtime mechanisms that detect when a server has gone down, queue client requests, handle attachment and detachment of clients, and so on.

Connectors need not be binary. Two of the three connector types in [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) can involve more than two participants: the publish-subscribe bus and the failover client-server connectors.

It may also be possible to carry out both qualitative and quantitative analyses of system properties such as performance, reliability, and security based on this view. For instance, the design decision that causes the administrative user interface to be the only way to change the database schema would improve the security of the system. But that decision also might affect serviceability or availability. For example, does the use of the administrative interface lock out the servers? Similarly, by knowing properties about the reliability of the individual servers and the database, you might be able to produce numeric estimates of the overall reliability of the system, using some form of reliability analysis.

Here are some things to note about the nature of C&C graphical documentation, as illustrated in [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01):

• It acts as a key to the associated supporting documentation (not shown here), where details about the elements, relations, and their properties can be found.

image

Supporting documentation is discussed in [Section 10.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch10.html#ch10sec1lev1).

• It’s restricted to information that can be simply presented in—and comprehended from—a single diagram.

• It’s explicit about its vocabulary of component-and-connector types in the diagram’s key.

• It indicates the number and kind of interfaces on its components and connectors.

• It uses component-and-connector abstractions that may have rich semantics and complex implementations.

The documentation explaining the diagram should elaborate on the elements shown. Supporting documentation should explain, for example, how Account Server-Backup improves the availability of the system. Some of the elements of this [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) may themselves represent subsystems that have their own subarchitectures, shown elsewhere.

The combination of C&C diagrams and their supporting documentation provide an essential vehicle for communicating an architect’s design intent, supporting reasoning about the runtime behavior of the system, and justifying design decisions in terms of their impact on relevant quality attributes.

3.2.1 Elements

The elements of a C&C view are components and connectors. Each element in a C&C view of a system has a runtime manifestation, consuming execution resources and contributing to the execution behavior of that system. Attachment relations of a C&C view associate components with connectors (via their respective ports and roles) to form a graph that represents a runtime system configuration.

Components

[Components](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_14) represent the principal computational elements and data stores that are present at runtime. Each component in a C&C view has a name. The name should indicate the intended function of the component. The name also allows you to relate the graphical element with any supporting documentation for that component.

image

Components are the principal computational elements and data stores that execute in a system.

Components have interfaces called [ports](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_38). A port defines a specific point of potential interaction of a component with its environment. A port usually has an explicit type, which defines the kind of behavior that can take place at that point of interaction. A component may have many ports of the same type. In this respect, ports differ from interfaces of modules, whose interfaces are never replicated. For example, a filter might have several input ports of the same type to handle multiple input streams, or a server might provide a number of request ports for client interactions. The database in [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) has two ports for two kinds of access.

image

A port is an interface of a component. A port defines a point of interaction of a component with its environment.

You can annotate a port with a number or range of numbers to indicate replication. For example, a port annotated with “[3]” stands for three occurrences of that port. A port annotated with “[0..10]” means that there are from 0 to 10 instances of that port. That form is useful when defining component types, allowing component instances to bind the exact number, or for components that dynamically create new points of interaction.

A component’s ports should be explicitly documented, by showing them in the diagram and defining them in the diagram’s supporting documentation.

image

To indicate multiple ports of the same type in a diagram using an informal notation, you can draw each one separately or you can show a single port but append a bracketed number (for example, [5]) after the port’s name to indicate its degree of replication. UML provides a similar convention.

A component in a C&C view may represent a complex subsystem, which itself can be described as a C&C subarchitecture. This subarchitecture can be depicted graphically *in situ* when the substructure is not too complex, by showing it as nested inside the component that it refines. Often, however, it is documented separately. A component’s subarchitecture may be in a style different from the one in which the component appears.

image

See [Chapter 7](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch07.html#ch07) for a more complete discussion of types of information that can be used to define a port.

[Section 6.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch06.html#ch06sec1lev1) contains more detail on guidelines for documenting hierarchical relationships and refinement.

See [Section 3.2.3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03sec2lev3) for more information on how to document substructure using an interface delegation relation.

When a component has such a substructure, you should also document the relationship between the “internal” and “external” ports. As we describe later, this relationship is captured using an interface delegation relation.

Connectors

[Connectors](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_15) are the other kind of element in a C&C view. Simple examples of connectors are service invocation, asynchronous message queues, event multicast, and pipes that represent asynchronous, order-preserving data streams. But as we noted earlier, connectors often represent much more complex forms of interaction, such as a transaction-oriented communication channel between a database server and a client, or an enterprise service bus that mediates interactions between collections of service users and providers.

image

A connector is a runtime pathway of interaction between two or more components.

Connectors have [roles](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_46), which are its interfaces, defining the ways in which the connector may be used by components to carry out interaction. For example, a client-server connector might have *invokes-services* and *provides-services* roles. A pipe might have *writer* and *reader* roles. Like component ports, connector roles differ from module interfaces in that they can be replicated, indicating how many components can be involved in its interaction. A publish-subscribe connector might have many instances of the *publisher* and *subscriber* roles.

image

A role is an interface of a connector. A role defines a point of interaction of a connector and indicates how components may use a connector in interactions.

A role typically defines the expectations of a participant in the interaction. For example, an *invokes-services* role might require that the service invoker initialize the connection before issuing any service requests. The semantics of the interaction represented by a connector is often documented as a protocol specification prescribing what patterns of events or actions are allowed to take place over the connector.

image

A protocol specification or a pattern of events can be described using behavioral notations, described in [Chapter 8](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch08.html#ch08).

Like components, complex connectors may in turn be decomposed into collections of components and connectors that describe the architectural substructure of those connectors. For example, a decomposition of the failover client-server connector of [Figure 3.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03fig01) would probably include components that are responsible for buffering client requests, determining when a server has failed, and rerouting requests.

image

Refinement is described in [Section 6.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch06.html#ch06sec1lev1).

8.2.2 Step 2: Determine What Types of Information Are Available or Can Be Constrained

Types of Communication

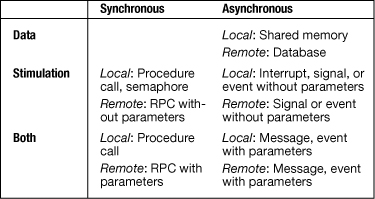
Looking at a structural diagram that depicts two interrelated elements, users of the documentation often ask “What does the line connecting the elements mean? Is it showing flow of data or control?” The answer should be in the diagram key. A behavioral diagram provides a place to describe aspects of the transfer of information and the stimulation of actions from one element to another in more detail than you include in diagram keys.

image

See “[Perspectives: Quivering at Arrows](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch00.html#ch00sb10)” on page [41](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch00.html#ch00sb10), in the prologue.

[Table 8.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch08.html#ch08tab01) shows some common examples of various types of communication. In this table we identify three different important characteristics of a type of communication. The first characteristic is the general purpose of the communication. In some cases, the primary purpose is to exchange data. In others, the primary purpose is to stimulate another element to signal that a task is completed or that a service is required. Often, however, a combination of the two is the main idea, as is the case when an element stimulates another to deliver data or when information is passed in messages or as parameters of events.

Table 8.1 Types of communication



A second characteristic indicates whether elements communicate via synchronous or asynchronous means. Remote procedure call (RPC) is an example of synchronous communication. The sender calls the receiver and is blocked until the receiver responds. Messaging is an example of asynchronous communication. The sender does not concern itself with the state of the receiver when sending a message or posting an event. Right after the message is sent, the sender continues its execution and is not blocked waiting for a response. In fact, the sender and receiver may not be aware of each other’s identity.

Consider the telephone and e-mail as examples. If you make a phone call to someone, the person has to be at the phone in order for it to achieve its full purpose. That is synchronous communication. If you send an e-mail message and go on to other business, perhaps without concern for a response, the communication is asynchronous. The distinction between synchronous and asynchronous communication has implications for the behavior of the transaction. An asynchronous call introduces concurrency and is more suitable for loosely coupled elements. The distinction also affects modifiability. Asynchronous interactions are usually more complicated, especially when the transaction needs a callback, which may require establishing a callback end point and a mechanism for correlating the original call to the callback message.

A third characteristic of the type of communication is whether the call is local (within the same container or machine) or remote. If it’s remote, the performance is worse, because of the network overhead (even if the remote call reaches a component within the same machine, there’s the overhead of going through the stack of network layers). Remote calls are also less reliable. A call or its response may not be delivered, may get corrupted, or may arrive in the wrong order.

Constraints on Ordering

In the case of synchronous communication, you probably want to say more than that there is two-way communication from A to B. For instance, you may want to say whether the target of the original message uses the assistance of other elements before it can respond to the original request.

You may want to be more specific about certain aspects of the way an element reacts to its inputs. You may want to note whether an element requires all or just some of its inputs to be present before it begins calculating. Also, you may want to say whether it can provide intermediate outputs or only final outputs. If a specific collection of events must take place before an action of an element is enabled, that should be specified, as should the circumstances (such as ordering) in which the events or element interactions will be triggered. These types of constraints on interactions provide information that is useful for analyzing the design for functional correctness, as well as for quality attributes.

Time-Based Stimulation

If any activities are specified to take place at specific times or after certain intervals of time, some concept of time needs to be introduced into your documentation. Time can be specified as either a point in time (that is, calendar based) or as a duration (timer based). Duration can be based on either wall time or task time. As an example of using a point in time, you may specify that certain behavior is different on weekends or holidays. As an example of using wall-time duration, you may specify that every five minutes, the system should determine how many people are logged in. As an example of task-based duration, you may specify that a task can use one minute of CPU time before being temporarily interrupted.

4.1 An Introduction to C&C Styles

A component-and-connector (C&C) style introduces a specific set of component-and-connector types and specifies rules about how elements of those types can be combined. Additionally, given that C&C views capture runtime aspects of a system, a C&C style is typically also associated with a computational model that prescribes how data and control flow through systems designed in that style.

The choice of a C&C style (or styles) will usually depend on the nature of the runtime structures in the system. For example, if the system will need to access a set of legacy databases, the style will likely be based on a shared-data style. Alternatively, if a system is intended to perform data stream transformation, a data flow style will likely be chosen.

image

In [Section 4.9](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04sec1lev9) we provide references for reading about dozens of C&C styles.

The choice of style will also depend on the intended use of the documentation. For example, if high performance is a critical property, the style will likely be chosen to enable analysis of performance, so that trade-offs affecting that system quality can be assessed.

Many C&C styles exist. To make sense of the space of these styles, we begin by describing some broad categories of commonly used C&C styles, and then we consider in more detail one or more example styles in each category.

The space of C&C styles is quite large. For example, C&C styles can differ dramatically in terms of the types of the connectors that they support. Styles based on asynchronous event broadcast (such as publish-subscribe) are quite different from those based on synchronous service invocation. Similarly, styles may differ in terms of the types of components that they permit or require. For instance, some styles require a database component to be present. Other styles may require a registry component to enable components to find others at runtime. Styles may differ in terms of topological restrictions, such as whether the components are assigned to tiers. They may also differ in terms of their level of domain specificity. For example, a style to support automotive control systems will likely involve connectors that represent specific protocols for real-time coordination. Similarly, there exist dozens of client-server styles that differ in subtle (or not-so-subtle) ways, depending on the nature of the application domain they are addressing. For example, some client-server styles allow late binding of requests for services, where the recipient of a request is determined dynamically; others insist on a static configuration determined when a system is built or deployed.

image

[Section 6.1.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch06.html#ch06sec2lev4) discusses how styles can be progressively specialized from generic styles to domain-specific styles and product line.

One way to impose some conceptual order on the space of C&C styles is to consider several broad categories of styles, differentiated primarily by their underlying computational model. In this chapter we consider examples in four such categories.

• *Call-return styles.* Styles in which components interact through synchronous invocation of capabilities provided by other components.

• *Data flow styles.* Styles in which computation is driven by the flow of data through the system.

• *Event-based styles.* Styles in which components interact through asynchronous events or messages.

• *Repository styles.* Styles in which components interact through large collections of persistent, shared data.

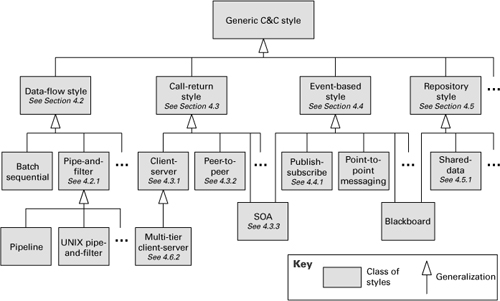
Additionally we consider several crosscutting style issues, such as the imposition of a tiered topology, and augmentations that allow one to reason about concurrency.

image

[Section 4.6.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04sec2lev7) describes communicating processes, which is a way to add concurrency to a C&C style. [Section 4.6.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04sec2lev8) describes the notion of tiers, which are common in some C&C architectures.

[Figure 4.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04fig01) provides a birds-eye view of part of the terrain. This figure can be interpreted as a kind of C&C style specialization hierarchy. At the top is the most general and unconstrained form of C&C view: namely, one that uses generic components and connectors, with no particular constraints on topology, behavior, and element properties. Below this are the general categories of C&C styles distinguished largely by their underlying computational model. Below these are specializations of these general styles. Note that a specific style may specialize more than one general category, as is the case of the service-oriented architecture (SOA) style.

Figure 4.1 A partial representation of the space of C&C styles



Naturally this is only a partial representation of the space of C&C styles: there are other general categories, and there are many styles that are specializations of these categories. Additionally, in most real systems several styles may be used together, often from across categories. For example, enterprise IT applications are frequently a combination of client-server and shared-data styles.

4.2.1 Pipe-and-Filter Style

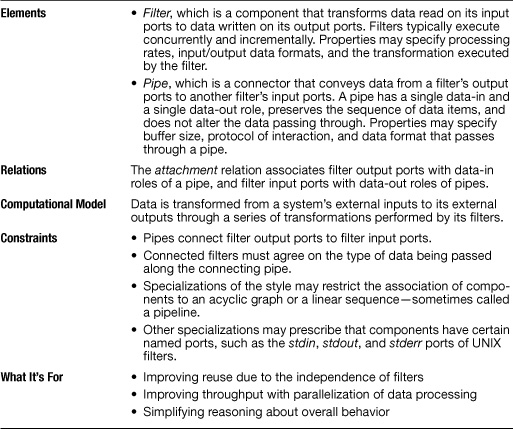
Overview

The pattern of interaction in the pipe-and-filter style is characterized by successive transformations of streams of data. Data arrives at a filter’s input ports, is transformed, and then is passed via its output ports through a pipe to the next filter. A single filter can consume from, or produce data to, multiple ports. Modern examples of such systems are signal-processing systems, systems built using UNIX pipes, the request-processing architecture of the Apache Web server, the map-reduce paradigm for search engines, Yahoo! Pipes for processing RSS feeds, and many scientific computation systems that have to process and analyze large streams of experimental data.

Elements, Relations, and Properties

The basic form of pipe-and-filter style, summarized in [Table 4.1](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04tab01), provides a single type of component—the [filter](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_24)—and a single type of connector—the [pipe](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/gloss.html#gloss01_37). A filter transforms data that it receives through one or more pipes and transmits the result through one or more pipes. Filters typically execute concurrently and incrementally. A pipe is a connector that conveys streams of data from the output port of one filter to the input port of another filter. Pipes act as unidirectional conduits, providing an order-preserving, buffered communication channel to transmit data generated by filters. In the pure pipe-and-filter style, filters interact only through pipes.

Table 4.1 Summary of the pipe-and-filter style



Because pipes buffer data during communication, filters can act asynchronously and concurrently. Moreover, a filter need not know the identity of its upstream or downstream filters. For this reason, pipe-and-filter systems have the nice formal property that the overall computation can be treated as the functional composition of the computations of the filters, allowing the architect to reason about the end-to-end behavior as a simple composition of the behaviors of the parts.

Advice

Typical properties to document for pipes include

• Pipe capacity (that is, buffer size)

• How end-of-data is signaled

• What form of blocking occurs when writing to a pipe whose buffer is full or reading from a pipe that is empty

Properties of filters can include

• Whether or not each filter is a separate process

• The data stream transformation each performs

What the Pipe-and-Filter Style Is For

Systems conforming to a pipe-and-filter style are typically used in data transformation systems, where the overall processing can be broken down into a set of independent steps, each responsible for an incremental transformation of its input data. The independence of the processing done by each step supports reuse, parallelization, and simplified reasoning about overall behavior.

Often such systems constitute the front end of signal-processing applications. These systems typically receive sensor data at a set of initial filters; each of these filters compresses the data and performs initial filtering. “Downstream” filters reduce the data further and do synthesis across data derived from different sensors. The final filter typically passes its data to an application, for example, providing input to modeling or visualization tools.

Analyses associated with pipe-and-filter systems include deriving the aggregate transformation provided by a graph of filters and reasoning about system performance: input/output stream latency, pipe buffer requirements, and throughput.

Relation to Other Styles and Models

A pipe-and-filter view of a system is not the same as a data flow model. In the pipe-and-filter style, lines between components represent connectors, which have a specific computational meaning: They transmit streams of data from one filter to another. In data flow models, the lines represent relations, indicating the communication of data between components. Flows in a data flow model have little computational meaning: They simply indicate that data flows from one element to the next. This flow might be realized by a connector, such as a procedure call, the routing of an event between a publisher and a subscriber, or data transmitted via a pipe. The reason that these views might be confused is that the data flow model of a pipe-and-filter style looks almost identical to the original pipe-and-filter view.

image

Data flow models are discussed in “[Perspectives: Data Flow and Control Flow Models](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03sb08),” on page [146](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03sb08), in [Chapter 3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch03.html#ch03).

Data flow styles are often combined with other styles by using them to characterize a particular subsystem. A good example of this is the filter processing chains of the Apache Web server.

Example of the Pipe-and-Filter Style: Yahoo! Pipes

“Rewire the Web” is the motto of Yahoo! Pipes, a composition tool that lets Web users combine simple functions quickly and easily into pipe-and-filter applications that aggregate and manipulate content from around the Web.

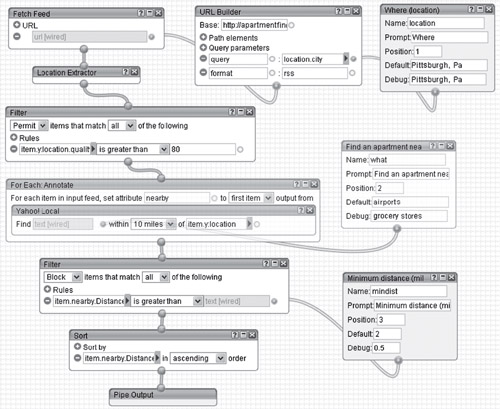
The basis of Yahoo! Pipes is the many RSS feeds available from sites on the Internet. These data streams form the input to the applications that users build, applications that combine and manipulate the data in the streams to form useful results. Many of the building blocks to perform general-purpose filtering and manipulation of the data streams are made available in the composition environment itself, rather like library functions.

For example, you can take an RSS stream from a financial news site and filter it so that only news items related to stocks that you own are shown. Or you can take an RSS stream from a sports site and filter it so that you see news about your favorite teams or athletes.

Yahoo! Pipes uses terminology not quite the same as that in this book. It calls a complete application a pipe; the building blocks are called modules. A filter is a special kind of module that removes values from a stream based on given comparison criteria.

[Figure 4.2](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04fig02) shows an application that finds an apartment for rent that is near a given type of business, such as a movie theater. This is based on one of the teaching examples on the Yahoo! Pipes Web site.

Figure 4.2 A Yahoo! Pipes application for finding apartments for rent near a given location (shown using the notation of the Yahoo! Pipes editor). The pipe-and-filter flow runs from top to bottom through the seven “modules” down the left-hand side (each representing what our pipe-and-filter style calls a filter); this is indicated by the thick solid lines (the pipes) connecting the output port of one to the input port of the next. The other “modules” supply inputs to the mainline components; this is indicated by the thinner, hollow lines. The Fetch Feed component uses the RSS output from an apartment-finder search; it is fed the search site URL and the search parameters by the helper modules to its right. The Location Extractor and the Filter component extract high-quality (well-formed) addresses from the apartment-finder search. That stream feeds Yahoo! Local, which finds businesses of a given type (supplied by its helper module) near a given location. (The For Each component applies the function shown in its interior to every item in the input stream.) The second Filter removes listings that aren’t a minimum distance from our search term. The Sort component orders the stream in ascending order of distance for viewing via the Pipe Output component.



4.1.2 REST   
REST was proposed by Roy Fielding [Fielding 2000]. It avoids the complexity and processing overhead of the Web services protocols by using bare http. As an example, consider a weather forecast service that is publicly available and is provided by http://www.weather.com. One important REST concept is a resource, which is a piece of information that has a unique identifier (e.g., a uniform resource identifier (URI)). For the weather service, examples of resources include  
 • current weather for zip code 15213  
 • weather forecast for tomorrow for the city of Pittsburgh  
 • 10-day weather forecast for zip code 15213  
 • temperature averages for the city of Pittsburgh in October   
In this example, there are three types of resources: current weather, weather forecast, and temperature averages. We can structure the URIs of the resources based on these three types. Parameters can be represented by elements in the URI hierarchical path or [key]=[value] pairs. The URIs corresponding to the resources we listed above could be  
 • <http://www.weather.com/current/zip/15213>  
 • <http://www.weather.com/forecast/tomorrow/city/Pittsburgh>  
 • <http://www.weather.com/forecast/tenday/zip/15213>  
 • <http://www.weather.com/avg/city/Pittsburgh?month=10>  
  
It is no coincidence that these URIs look like what we type in a Web browser. REST relies on the http protocol for the interaction between service users and providers. The http protocol has four basic operations: POST, GET, PUT, and DELETE. In a REST design, the application of these operations to resource URIs correspond to create, retrieve, update, and delete (CRUD) operations commonly used in information systems. Thus, if the service user sends a POST request on http://www.weather.com/current/zip/15213, it is asking the service provider to create the data for the current weather in zip code 15213 using the data passed along with the request. A GET request on the same URI tells the service provider to retrieve the data for the current weather in zip code 15213 and return it in the response. A PUT request indicates that the service provider should replace the data it has with the data sent in the request. A DELETE request indicates that the service user wants the service provider to delete the data. The http protocol also defines the status codes that can be returned: 200 for OK, 201 for created, 401 for unauthorized, and so forth.   
A unique characteristic of REST is that it prescribes a uniform interface—the service is exposed as information resources upon which a fixed set of operations can be applied, rather than a set of methods with different parameters. In a REST solution, for each resource we have to define a representation. In most cases, basic XML is the format used. Also, REST services are necessarily stateless—they don’t store the conversational state across multiple requests from the same service user.   
REST advocates claim several benefits over SOAP-based Web services:  
 • REST results in improved modifiability. For a service user to interact with a non-REST Web Service, the service user has to understand the specifics of the data contract (i.e., how data is structured) and the interface contract (i.e., what operations can be performed). Because of the uniform interface, to invoke a REST service, the service user only has to understand the data contract, because the interface contract is uniform for all services [Vinoski 2007].  
 • REST is easy to implement and yields high interoperability, since it only requires standard http support from both the service user and provider. It doesn’t require SOAP toolkits to implement the code or an application server that supports Web services.  
 • REST has better performance due to its ability to cache the responses (when applicable) and to the absence of the intermediaries, message wrapping, and serialization that are required by Web services.   
Web services and REST represent different paradigms to implement SOA. One is centered on the operations to be executed by the service provider. The other is focused on access to resources. In the architecture evaluation of an SOA system, the evaluation team can question which approach would be more appropriate for each service. REST is a good option for accessing static or nearly static resources. It is also useful for portable devices with limited bandwidth, because REST messages are less verbose than SOAP messages. The Web services technology offers better support for security, reliable messaging, and transaction management [MacVittie 2006]. As a result of widespread adoption, plenty of knowledge on Web services is provided on the Web and in the professional community. There is also better tool support for developing Web services, although APIs for easy development of REST solutions are being created, such as the Java API for RESTful Web services [Sun 2007b]. If the application is going to provide services to multiple users and business partners, an alternative is to build both SOAP and REST interfaces for the same services like Amazon.com and eBay do.  
  
4.3.2 Peer-to-Peer Style

Overview

In the peer-to-peer style, components directly interact as peers by exchanging services. Peer-to-peer communication is a kind of request/reply interaction without the asymmetry found in the client-server style. That is, any component can, in principle, interact with any other component by requesting its services. Each peer component provides and consumes similar services, and sometimes all peers are instances of the same component type. Connectors in peer-to-peer systems may involve complex bidirectional protocols of interaction, reflecting the two-way communication that may exist between two or more peer-to-peer components.

image

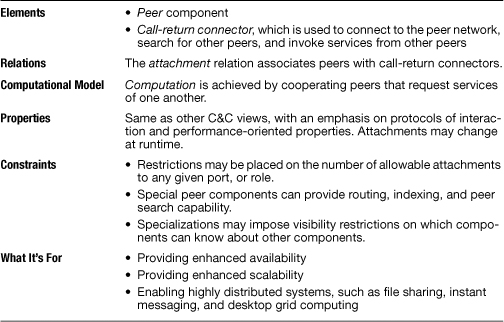
The peer-to-peer architecture style has inspired new models for industrial production, community knowledge, political movement, property ownership, and an economic alternative to capitalism. See [en.wikipedia.org/wiki/Peer-to-peer\_(meme)](http://en.wikipedia.org/wiki/Peer-to-peer_(meme)).

Examples of peer-to-peer systems include file-sharing networks, such as BitTorrent and eDonkey; instant messaging and VoIP applications, such as Skype; and desktop grid computing systems.

Elements, Relations, and Properties

[Table 4.3](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04tab03) summarizes the peer-to-peer style. The component types in this style are peers, which are typically independent programs running on network nodes. The principal connector type is the call-return connector. Unlike in the client-server style, the interaction may be initiated by either party: each peer component acts as both client and server. Peers have interfaces that describe the services they request from other peers and the services they provide. The computational flow of peer-to-peer systems is symmetric: Peers first connect to the peer-to-peer network and then initiate actions to achieve their computation by cooperating with their peers by requesting services from one another.

Table 4.3 Summary of the peer-to-peer style



Often a peer’s search for another peer is propagated from one peer to its connected peers for a limited number of hops. A peer-to-peer architecture may have special peer nodes (called ultrapeers, ultranodes, or supernodes) that have indexing or routing capability and allow a regular peer’s search to reach a larger number of peers.

Constraints on the use of the peer-to-peer style might limit the number of peers that can be connected to a given peer or impose a restriction about which peers know about which other peers.

What the Peer-to-Peer Style Is For

Peers interact directly among themselves and can play the role of both service caller and service provider, assuming whatever role is needed for the task at hand. This partitioning provides flexibility for deploying the system across a highly distributed platform. Peers can be added and removed from the peer-to-peer network with no significant impact, resulting in great scalability for the whole system.

Typically multiple peers have overlapping capabilities, such as providing access to the same data. Thus, a peer acting as client can collaborate with multiple peers acting as servers to complete a certain task. If one of these multiple peers becomes unavailable, the others can still provide the services to complete the task. The result is improved overall availability. The load on any given peer component acting as a server is reduced, and the responsibilities that might have required more server capacity and infrastructure to support it are distributed. This can decrease the need for other communication for updating data and for central server storage, but at the expense of storing the data locally.

Peer-to-peer computing is often used in distributed computing applications, such as file sharing, instant messaging, and desktop grid computing. Using a suitable deployment, the application can make efficient use of CPU and disk resources by distributing computationally intensive work across a network of computers and by taking advantage of the local resources available to the clients. The results can be shared directly among participating peers.

Relation to Other Styles

The absence of hierarchy means that peer-to-peer systems have a more general topology than client-server systems.

Examples of the Peer-to-Peer Style

Gnutella is a peer-to-peer network that supports bidirectional file transfers. The topology of the system changes at runtime as peer components connect and disconnect to the network. A peer component is a running copy of a Gnutella client program connected to the Internet. Upon startup, this program establishes a connection with a few other peers. The Web addresses of these peers are kept in a local cache.

image

In late 2007, [Gnutella] was the most popular file sharing network on the Internet with an estimated market share of more than 40%.

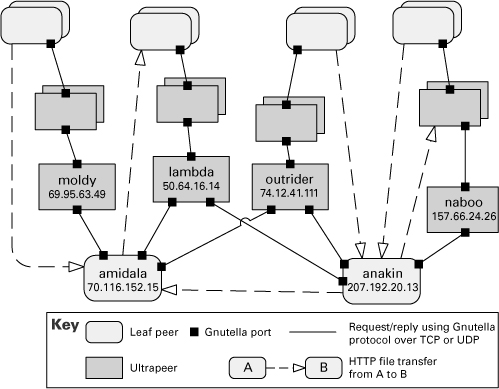
—Wikipedia  
([en.wikipedia.org/wiki/Gnutella](http://en.wikipedia.org/wiki/Gnutella))

The Gnutella protocol supports request/reply messages for peers to connect to other peers and search for files. Peers are identified by their IP address, and the Gnutella protocol messages are carried over dedicated UDP and TCP ports. To perform a search, a Gnutella peer requests information from all of its connected peers, which respond with any information of interest. The connected peers also pass the request to their peers successively, up to a predefined number of “hops.” All the peers that have positive results for the search request reply directly to the requester, whose IP address and port number go along with the request. The requester then establishes a connection directly with the peers that have the desired file and initiates the data transfer using HTTP (outside the Gnutella network).

Later versions of Gnutella differentiate between leaf peers and ultrapeers. An ultrapeer runs on a computer with a fast Internet connection. A leaf peer is usually connected to a small number (say, three) of ultrapeers, and an ultrapeer is connected to a large number of other ultrapeers and leaf peers. The ultrapeers are responsible for routing search requests and responses for all leaf peers connected to them.

[Figure 4.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04fig04) shows part of a peer-to-peer view of a Gnutella network using an informal C&C notation. For brevity, only two leaf peers and four ultrapeers are identified. Each of the identified leaf peers uploads and downloads files directly from other peers.

Figure 4.4 A C&C diagram of a Gnutella network, using informal notation

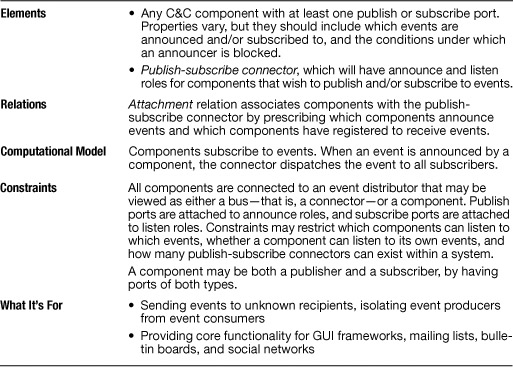


4.4.1 Publish-Subscribe Style

Overview

In the publish-subscribe style, summarized in [Table 4.5](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04tab05), components interact via announced events. Components may subscribe to a set of events. It is the job of the publish-subscribe runtime infrastructure to make sure that each published event is delivered to all subscribers of that event. Thus the main form of connector in this style is a kind of event bus. Components place events on the bus by announcing them; the connector then delivers those events to the components that have registered an interest in those events.

Table 4.5 Summary of the publish-subscribe style



The computational model for the publish-subscribe style is best thought of as a system of independent processes or objects, which react to events generated by their environment, and which in turn cause reactions in other components as a side effect of their event announcements.

Examples of systems that employ the publish-subscribe style are the following:

• Graphical user interfaces, where a user’s low-level input actions are treated as events that are routed to appropriate input handlers

• Applications based on the model-view-controller (MVC) pattern, where view components are notified when the state of a model object changes

• Extensible programming environments, in which tools are coordinated through events

• Mailing lists, where a set of subscribers can register interest in specific topics

• Social networks, where “friends” are notified when changes occur to a person’s Web site

Elements, Relations, and Properties

The publish-subscribe style can take several forms. In one common form, called *implicit invocation*, the components have procedural interfaces, and a component registers for an event by associating one of its procedures with each subscribed type of event. When an event is announced, the associated procedures of the subscribed components are invoked in an order usually determined by the runtime infrastructure. Graphical user-interface frameworks, such as Visual Basic, are often driven by implicit invocation: User code fragments are associated with predefined events, such as mouse clicks.

In another publish-subscribe form, events are simply routed to the appropriate components. It is the component’s job to figure out how to handle the event. Such systems put more of a burden on individual components to manage event streams, but also permit a more heterogeneous mix of components than implicit invocation systems do.

In some publish-subscribe systems, an event announcer may block until an event has been fully processed by the system. For example, some user-interface frameworks require that all views be updated when the data they depict has been changed. This is accomplished by forcing the component that announces a “changed-data” event to block until all subscribing views have been notified.

Advice

Useful properties to document for components include these:

• Which events a component announces or subscribes to

• Conditions under which an announcer is blocked

• Whether components can change their subscriptions dynamically

• Whether new event types can be created dynamically, or the event vocabulary is fixed at build or deployment time

• Whether one can add new publishers to the system dynamically

Connector properties often describe the semantics of the event dispatch mechanism:

• Can a subscriber queue up new events when it’s busy handling an event?

• Is the connector synchronous or asynchronous?

• Do events have priorities?

• Is temporal or causal ordering enforced?

• Is event delivery reliable?

• What are the semantics of each event?

• Does the connector support other distributed component management, such as starting and stopping publish-subscribe components at the same time?

What the Publish-Subscribe Style Is For

The publish-subscribe style is used to send events and messages to an unknown set of recipients. Because the set of event recipients is unknown to the event producer, the correctness of the producer cannot depend on those recipients. Thus new recipients can be added without modification to the producers.

Publish-subscribe styles are often used to decouple user interfaces from applications. They may also be used to integrate tools in a software development environment: tools interact by announcing events that trigger invocation of other tools. Other applications include systems such as bulletin boards, social networks, and message lists, where some dynamically changing set of users are notified when the content that they care about is modified.

Relation to Other Styles

The publish-subscribe style is similar to a blackboard repository style, because in both styles components are automatically triggered by changes to some component. However, in a blackboard system, the database is the only component that generates such events; in a publish-subscribe system, any component may generate events.

Implicit invocation is often combined with call-return in systems in which components may interact either synchronously by service invocation or asynchronously by announcing events. For example, many service-oriented architectures and distributed object systems (such as CORBA and Java EE) support both synchronous and asynchronous communication. In other object-based systems, synchronous procedure calls are used to achieve asynchronous interaction using the MVC pattern or the observer pattern.

Example of the Publish-Subscribe Style

[Figure 4.6](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch04.html#ch04fig06) is a publish-subscribe view of the SEI ArchE tool. There are three different publish-subscribe interactions in this architecture:

1. Eclipse UI event manager acts as an event bus for user-interface events (such as button clicks). Subscription information—that is, what UI events are relevant to the system and what components handle them—is defined at load time when the event manager reads the SEI.ArchE.UI plug-in config XML file. From then on a UI event generated by the user working on a view or editor is dispatched via implicit invocation to the action handler objects that subscribe to that event.

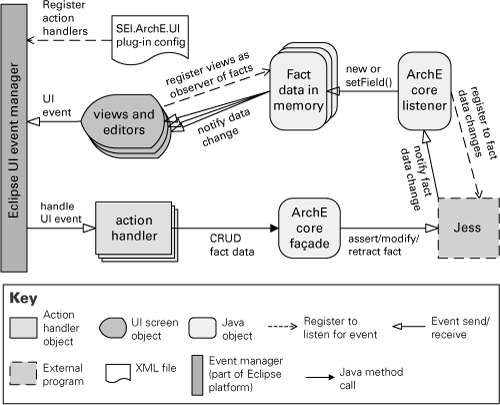
image

[Sections 2.3.6](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch02.html#ch02sec2lev18) and [6.6.4](https://learning.oreilly.com/library/view/documenting-software-architectures/9780132488617/ch06.html#ch06sec2lev21) have more information about the ArchE tool.

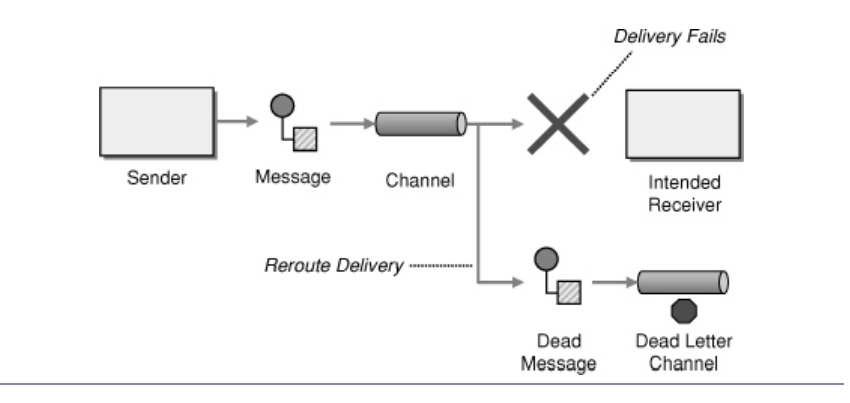
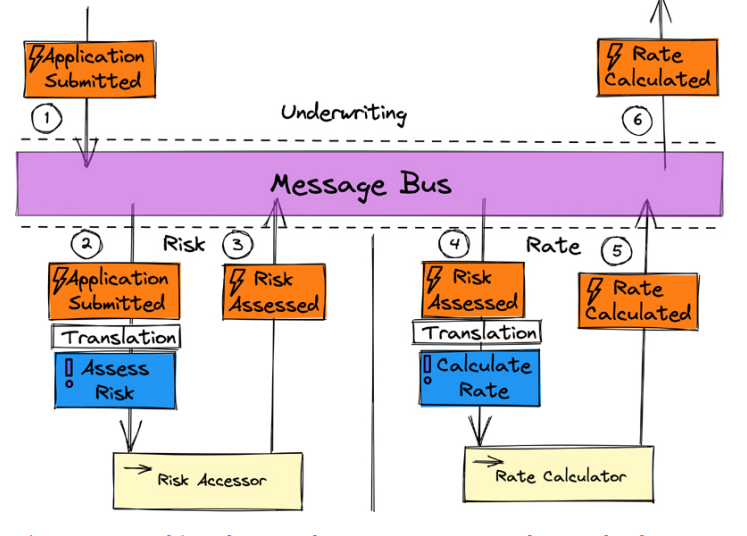
2. The data manipulated in ArchE is stored using a rule engine called Jess. Data elements are called facts. When a user action creates, updates, or deletes a fact, that action generates respectively an assert, modify, or retract fact event that is sent to Jess. When Jess processes that event, changes to many other facts may be triggered. Jess also acts as an event bus that announces changes to facts. In the ArchE architecture, there is one component that subscribes to all data changes: ArchE core listener.

3. ArchE keeps in memory copies of the fact data elements persisted in the rule engine. These copies are observable Java objects. User-interface screens (that is, views) that display those elements are observers of the fact data objects. When facts in memory are created or updated, the views are notified.

Figure 4.6 Diagram for a publish-subscribe view of the SEI ArchE tool



Request-Reply  
  
When a caller performs a Remote Procedure Invocation (50), the caller’s thread must block while it waits for the response. With Request-Reply, the requestor has two approaches for receiving the reply.  
1. Synchronous Block—A single thread in the caller sends the request  
message, blocks (as a Polling Consumer [494]) to wait for the reply mes-  
sage, and then processes the reply. This is simple to implement, but if the  
requestor crashes, it will have difficulty reestablishing the blocked  
thread. The request thread awaiting the response implies that there is  
only one outstanding request or that the reply channel for this request is  
private for this thread.  
2. Asynchronous Callback—One thread in the caller sends the request  
message and sets up a callback for the reply. A separate thread listens for  
reply messages. When a reply message arrives, the reply thread invokes  
the appropriate callback, which reestablishes the caller’s context and pro-  
cesses the reply. This approach enables multiple outstanding requests to  
share a single reply channel and a single reply thread to process replies  
for multiple request threads. If the requestor crashes, it can recover by  
simply restarting the reply thread. An added complexity, however, is the  
callback mechanism that must reestablish the caller’s context.  
  
Dead Letter Channel  
An enterprise is using Messaging (53) to integrate applications.   
What will the messaging system do with a message it cannot deliver?   
If a receiver receives a message it cannot process, it should move the invalid message to an Invalid Message Channel (60). But what if the messaging system cannot deliver the message to the receiver in the first place?   
There are a number of reasons the messaging system may not be able to deliver a message. The messaging system may not have the message’s channel configured properly. The message’s channel may be deleted after the message is sent but before it can be delivered or while it is waiting to be received. The message may expire before it can be delivered (see Message Expiration [176]). A message without an explicit expiration may nevertheless time out if it cannot be delivered for a very long time. A message with a selection value that all Selective Consumers (515) ignore will never be read and may eventually die. A message could have something wrong with its header that prevents it from being delivered successfully.   
Once the messaging system determines that it cannot deliver a message, it has to do something with the message. It could just leave the message wherever it is, cluttering up the system. It could try to return the message to the sender, but the sender is not a receiver and cannot detect deliveries. It could just delete the message and hope no one misses it, but this may well cause a problem for the sender that has successfully sent the message and expects it to be delivered (and received and processed). When a messaging system determines that it cannot or should not deliver a message, it may elect to move the message to a Dead Letter Channel.

  
The specific way a Dead Letter Channel works depends on the specific messaging system’s implementation, if it provides one at all. The channel may be called a “dead message queue” [Monson-Haefel] or “dead letter queue” [MQSeries], [Dickman]. Typically, each machine the messaging system is installed on has its own local Dead Letter Channel so that whatever machine a message dies on, it can be moved from one local queue to another without any networking uncertainties. This also records what machine the message died on. When the messaging system moves the message, it may also record the original channel on which the message was supposed to be delivered. The difference between a dead message and an invalid one is that the messaging system cannot successfully deliver what it then deems a dead message, whereas an invalid message is properly delivered but cannot be processed by the receiver. Determining if a message should be moved to the Dead Letter Channel is an evaluation of the message’s header performed by the messaging system. On the other hand, the receiver moves a message to an Invalid Message Channel (60) because of the message’s body or particular header fields the receiver is interested in. To the receiver, determination and handling of dead messages seem automatic, whereas the receiver must handle invalid messages itself. A developer using a messaging system is stuck with whatever dead message handling the messaging system provides, but she can design her own invalid message handling, including handling for seemingly dead messages that the messaging system doesn’t handle.  
In a stock trading system, an application that wishes to perform a trade can send a trade request. To make sure that the trade is received in a reasonable amount of time (less than five minutes, perhaps), the requestor sets the request’s Message Expiration (176) to five minutes. If the messaging system cannot deliver the request in that amount of time, or if the trading application does not receive the message (e.g., read it off of the channel) in time, then the messaging system will take the message off of the trade request channel and put the message on the Dead Letter Channel. The trading system may wish to monitor the system’s Dead Letter Channels to determine if it is missing trades.  
  
Chapter 9 Message- and Event-Driven Architectures  
 A message-driven architecture is one that emphasizes sending and receiving messages as playing a prominent role throughout the system. In general, message-driven architectures have been chosen less often compared to REpresentational State Transfer (REST) and remote procedure calls (RPC). This is because REST and RPC seem more similar to general-purpose programming language paradigms than does messaging; the former approaches provide abstractions that give the impression of procedure calls and method invocations, with which many programmers are already familiar.  
 Yet, REST and RPC are brittle mechanisms in comparison to general-purpose programming languages. It’s highly unlikely that a procedure call or method invocation will fail due to brittle mechanisms within a programming language. With the REST-over-HTTP and RPC approaches, it is very likely that failures will occur due to network and remote service failures. When failure does occur, the temporal coupling between one remote service and another will tend to cause a complete failure of the client service. The more remote services or subsystems that are involved in the given use case, the worse the problem can become. As Leslie Lamport, a distributed systems expert, described it:   
 A distributed system is one that prevents you from working because of the failure of a machine that you had never heard of.   
 That sort of cascading failure tends to be avoided when systems use asynchronous messaging, because the requests and responses are all temporally decoupled. Figure 9.1 highlights the relaxed temporal dependencies across subsystems involved in a choreographed event-driven process. To be clear, events capturing and communicating business interests are (generally) a form of message, and message-driven processes are a superset of event-driven processes.   
  
Figure 9.1 Event-driven choreography: events over a message bus translated to commands. 1  
 Some use a strict definition of a message, specifying that it must be directly sent peer-to-peer from sender to receiver. They may also constrain events to only those sent via pub-sub. The authors hold the opinion that this is an overly limiting perspective. The next section discusses using poll-based REST as a means to read event logs. Although many consumers might read such a feed of events, it is not the same as the push model of pub-sub. Of course, anyone can hold an opinion, so neither view is necessarily wrong.  
Choreographed and Orchestrated Processes  
These are two primary styles of process management: choreography and orchestration. Choreography comprises a decentralized style of process where, for example, events are published using messaging, and each subsystem context must determine whether the event is relevant to it, and if so, apply the event to its state. One or more events emitted in response by the subsystem in context will be relevant to one or more other subsystems. Choreography is relatively simple to understand and is most practical when processes have only a few steps. One disadvantage of this process management style is that when the process stalls somewhere, it can be difficult to determine where it went wrong and why. Another disadvantage is that event dependencies become coupled to subsystems that don’t own the events, and must be subjectively interpreted and applied to their own purpose. And, of course, dependencies on events can become quite tangled as system and process complexities increase.  
 Orchestration, in contrast, features a centralized style of process manager (i.e., Saga) that receives events emitted by any number of subsystems involved in the process, and then creates command messages that drive subsequent steps of the process to relevant subsystems. The advantages of using orchestrated processes include reduced dependencies across subsystems involved in a given process because the orchestrator takes on the entire responsibility of translating from event to command. The orchestrator can be a central point of failure, but that’s generally not a significant concern given the scalability and failover strategies common to well-designed distributed systems. Typically, an orchestrator is designed and implemented by the team most interested in its ultimate outcome. They might become a blocker when changes are made across subsystems involved in the process that must be applied inside the orchestrator. An orchestrator might be too complex for controlling processes with less complexity. The orchestrator must not become a dungeon for business logic; it must be used only to drive steps of the process.  
 The overall system works by means of the events that occur in a given subsystem being made available to other subsystems via a message bus. Message-driven architecture is also a Reactive architecture because the components, as seen in Figure 9.1, are passive until message stimuli occur, and then the appropriate components react to those stimuli. In contrast, imperative code drives responses by means of REST and RPC procedure calls or method invocations. Reactive is defined as having four primary characteristics: responsive, resilient, elastic, and message-driven [Reactive].  
 In Figure 9.1, the six steps executed across three subsystem contexts (Underwriting, Risk, and Rate) collectively provide the calculated rate needed to quote a policy premium for an applicant over the Web. The Underwriting subsystem context is blissfully unaware of the details involved in reaching the result. At some time in the future after the Application Submitted event occurs, Underwriting will be informed that a quotable premium rate is available.  
 The desired Underwriting outcome could require 2 seconds or 12 seconds to achieve. There is no failure in Underwriting because of the infrastructure being preconfigured to time out after 5 seconds, which would hold sway over the response to a REST request. It’s not that 12 seconds is an acceptable longterm Service Level Agreement (SLA)—but it is perfectly acceptable in the face of full failure of the Risk or Rate subsystem, followed by full recovery on a different cloud infrastructure, and possibly even in another region. Neither ordinary REST nor RPC would survive that pathway.