Chapter 12 Architecture

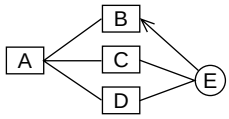
Architecture is a word used loosely to mean many different things; yet it is one of the most important aspects of the design of a system and has far-reaching effects on its success.

An architecture is, first, an abstraction of a system's implementation. There are many different architectural models that help you understand the system: process, module, usage dependencies, and so on. These models help you analyze certain *qualities* of the system: runtime qualities, such as performance, security, or reliability; and development-time qualities, such as modifiability and portability. These qualities are important to different system stakeholders: not only the end user but also the system administrator, developer, customer, maintainer, and so on. Different kinds of usage scenarios, including system modifications and deployment scenarios, can help you to evaluate architectures against such qualities.

You should clearly define the vocabulary of element types that can be used to describe an architecture: processes, replicators, buffers, caches, and events. The same is true of the roles played by the different components: controllers, mediators, routers, and so on. A good architecture exhibits a coherence and simplicity by being based on a small number of such elements and patterns that are used consistently throughout the design.

In practice, it is useful to distinguish the application architecture—how the business logic is split across components and how they interact—from the technical architecture: all the infrastructure and other domain-independent pieces that support that collaboration. The four-tier Web-enabled architecture presents a typical case for making this distinction.

12.1 *What Is Architecture?*



Because the term *architecture* is often used quite loosely, let us start with what it is not.

A neat-looking drawing of boxes, circles, and lines, laid out nicely in Powerpoint or Word, does *not* constitute an architecture.

Such a drawing leaves many critical questions unanswered. What is the nature of the blocks A–E: Are they objects, modules, libraries, or processes? Based on the shapes drawn, are A–D of a similar kind, and is E somehow different? Does the layout imply that

A plays a special role relative to B–D (for example, it is created before them)? What do the lines between these blocks mean: interprocess communication, compile-time dependencies, data flow?

In short, if given a complete implemented system—source, design documents, development structure of the project, and running installation—how would you determine whether it conformed to this architectural drawing?

© ***architecture*** The architecture of a system consists of the structure(s) of its parts (including design-time, test-time, and runtime hardware and software parts), the nature and relevant externally visible properties of those parts (modules with interfaces, hardware units, objects), and the relationships and constraints between them (there are a great many possibly interesting such relationships).

An architecture must define the parts, the essential external characteristics of each part, and the relationships between the parts.

12.1.1 Architecture Imposes Decisions and Constraints

In our view, architecture is not only about Gothic-scale structures but is also about all structures and relationships used down to the level of code.[1] The decision to use a four-tier structure, with a thin client, a Web server, a business application server, and a database, is architectural. But, in the extreme, we consider a consistent use of getX() and setX(x) methods also to be part of the (detailed) architecture. This view leads to a somewhat less formal definition of architecture.

© ***architecture*** The set of design decisions about any system (or smaller component) that keeps its implementors and maintainers from exercising needless creativity.

Here is a sampling of such decisions across a range of granularity:

1. Use a three-tier client-server architecture. All business logic must be in the middle tier, presentation and dialog on the client, and data services on the server. In this way you can scale the application server processing independently of persistent storage.
2. Use CORBA for all distribution, using CORBA event channels for notification and the CORBA relationship service.
3. Use Collection Galore's collections for representing any collections; by default, use its List class or else document your reason for not using it.
4. Use Model-View-Controller with an explicit ApplicationModel object to connect any UI to the business logic and objects.
5. Start every access method with get_ and set_.
6. Every computation component must have update_data and re_compute operations, to be invoked by the scheduler.

1. Provided that it influences the externally visible properties of the parts.

This does not mean that there is no creativity in implementation. Rather, it means that any level of refinement involves a set of constraining design decisions that define a limited and consistent toolbox of techniques for downstream work. To continue our lower-level example, use your creativity to address the problems that matter rather than come up with your own convention for dealing with get and set methods.

We use the same philosophy in type modeling: When stating behavior requirements, you have a limited set of terms and their definitions, defined by the type model, that you can refer to. When implementing a system at any level, your design vocabulary is defined by a limited set of well-defined constructs.[2]

12.1.2 Architectural Models

The architecture of a system frequently remains undocumented. If documentation exists, it may be hopelessly out of date with the implementation; or it may be so fuzzy and ambiguous that there is no way to tell whether it is accurate. To be useful, architecture must be described in a clear, explicit way to serve as the basis for understanding, implementation, reuse, and evolution of the system.

Unfortunately, in some contexts, the word *architecture* is dropped into a discussion to lend instant credibility to someone's position: If it is about architecture, it must be abstract, and surely it cannot be bothered with the difficult questions about what it all means to an implementation. This leaves us with ill-defined terms and diagrams of boxes and lines. As with any model, their value is severely degraded if the underlying terms and notations do not have a clearly defined meaning or if the number and complexity of design elements render them incomprehensible.

12.1.3 Many Architectural Views

An architectural description is an abstraction; there are many such abstractions that contribute to understanding a system, each one focused on one aspect and omitting other details. As with any model, there is some definition of conformance—that is, does a given implementation conform to that architecture? Some views are more focused on the design and development-time activities; others are relevant when you're testing or running the system; still others focus on deployment and upgrade activities. Table 12.1 shows some useful architectural views and the system parts each one focuses on.

Table 12.1 Various Architectural Views

View	Parts	Properties and Relations
Domain	Types, actions, subject areas	Attributes, associations, refinements, import relations between subject areas

2. Down to the primitive constructs of your programming language; you could almost certainly be more creative in assembler.

Logical	Components, connectors	How business logic is logically partitioned in software; how parts collaborate
Process	Process, thread, component	Synchronization relations: precedes, excludes, controls; assignment of software components to processes and threads
Physical	Hardware units, networks	Processing and communication capabilities (speed, latency, resources), communication relations, topology, physical containment
Distribution	Software components, processes, hardware	Deployed on or runs on CPU, can dynamically move to, network protocols
Calls	Methods, classes, objects, programs, procedures	Call invocations, parameters, returns, synchronous versus asynchronous, data volumes, processing time
Uses	Packages	Imports, uses, needs the presence of (more general than "calls")
Data flow	Actions	Provides or sets up data for (independent of call-specific protocols)
Modules	Design-time units of development work	Design decisions hidden in work units (packages); refinements used for submodule decompositions; justifications, including rationale for choices made and choices rejected

A small project may need only one of these views. The domain view can map directly into classes, which also constitutes the units of work without any higher-level component structuring; and the domain view can run in a single process on a single machine. A large project can use all these views and can even introduce new ones. These views are not independent and it is critical to know how they relate to each other.

In Catalysis, we can use refinement to model abstract objects that do not necessarily correspond to an instance of an OOP class and abstract actions that may not correspond to a single OOP message send. Based on this, we can use all the modeling techniques—including attributes, types, collaborations, refinement, justifications, interactions, snapshots, and states—to describe architectural models and their rationale.

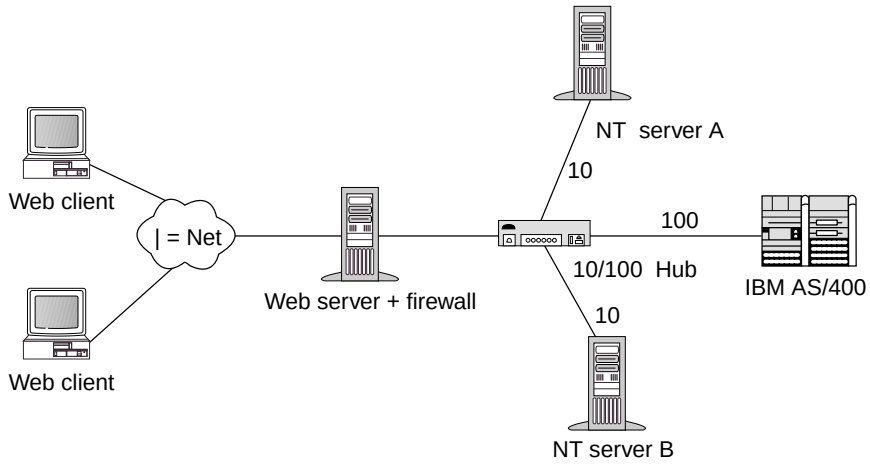
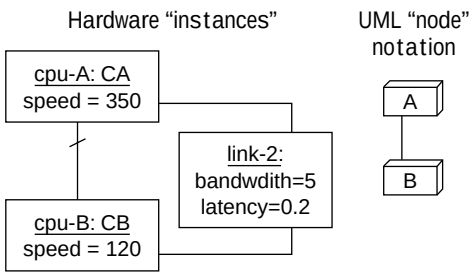


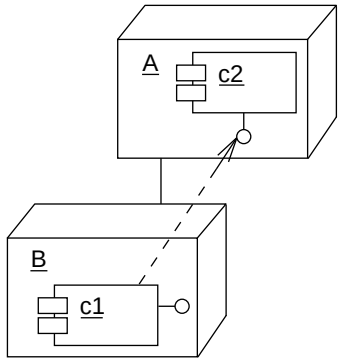
Figure 12.1 Networking symbols.

12.1.3.1 Physical Architecture



The processors in a physical model can be modeled as objects, their states modeled as attributes, their capabilities modeled as attributes, and communication links shown as explicit objects. It is useful to make visual distinctions between categories using stereotypes or a distinguished notation such as the one UML provides; or, you can use traditional network diagram symbols for the different hardware objects. Base operating systems can be shown as part of this hardware architecture (see Figure 12.1).

12.1.3.2 Software Distribution



Deployment of software is shown here against this physical model, with different software components shown on different hardware nodes; for example, the software components in the four-tier system are shown in Figure 12.5. Note that software component requirements, such as memory and storage space, can be modeled as attributes and matched against the corresponding attributes of the hardware. If needed, you can also explicitly specify the effect of actions, such as node failures (failover requirements) or network load, against such a model.

12.1.3.3 Process Architecture

Processes and threads can be introduced into a design to deal with essential concurrency in the problem itself or to handle performance requirements. Both can be modeled as objects with suitable stereotypes «process» and «thread», and all the tools of object modeling applied. Processes are mapped to CPUs.

Abstract actions can proceed concurrently with others provided they satisfy concurrency constraints. When specifying essential concurrency, we use rely and guarantee clauses on actions to implicitly constrain what can happen concurrently, or we can explicitly define concurrent actions by multiplicities and other explicit constraints.

12.2 *Why Architect?*

Assuming that *to architect* means to go through the analyses and reasoning required to come up with an architecture, why bother?

12.2.1 Multiple Stakeholders and Their Requirements

The architecture largely determines how well the system will meet its requirements. Much of our earlier focus in this book has been on describing the functional behavior that an end user would perceive; however, the overall requirements and conflicting objectives are frequently much broader and vary among the different *stakeholders*—roles of people who will be involved in the construction of the system (see Table 12.2).

Table 12.2 Concerns and Requirements of Various Stakeholders

Stakeholder	Concerns and Requirements
End user	Intuitive and correct behavior, helps do tasks, performance, reliability
System administrator	Intuitive behavior, tools to aid monitoring and administration
Marketer	Competitive features, time to market, fits into larger market positioning with other products
Customer	On time, low cost, stable
Developer	Clear requirements, simple and consistent design approach
Architect	Familiar domain, infrastructure, architecture; buildable; meet others' requirements
Development manager	Predictability and tracking of project, schedule, productive use of resources including existing or familiar code, cost
Maintainer	Understandable, consistent and documented design approach, easy to make commonly required modifications

A clearly defined architecture defines constraints on the implementation: The components must have the required external characteristics, and their interactions and relationships must conform to the prescribed constraints. Architecture describes some of the most

far-reaching design decisions about a system, having the greatest impact on how well the system meets its diverse requirements.

The architectural models are the primary vehicle for communication among all these stakeholders and for formal review of the models against the system requirements. They form the basis for an early prototype, against which many qualities can be evaluated even though there is minimal end-user functionality implemented.

The architecture directly influences the work-breakdown structure of the system and influences the package structure for the project, project planning and scheduling, team structure and communication needs, configuration management, testing, and deployment—in other words, all organizational aspects of the project. On a larger scale, a shared architecture for a family of products strongly influences the business that owns it.

12.2.2 Many Qualities Are Affected by Architecture

The quality of a system is a measure of how closely it meets all its requirements; it is an attribute of the design itself. Some qualities can be observed only during runtime because they relate to the dynamic behavior of the deployed system; others must be observed during development or maintenance activities because they relate to the design structure itself and how it can be manipulated.

Achieving system qualities is an engineering task: Frequently they conflict with one another. Some of the main qualities—often traditionally partitioned into functional and nonfunctional requirements—are briefly discussed next:

12.2.2.1 Development-Time Qualities

The following qualities most directly concern the development and maintenance teams; certain aspects of the architecture can make their life easier.

Modifiability: Is the system design structure amenable to effective modification? Are frequently requested changes achieved by changes that are fully localized within a single module or few modules without affecting public interfaces? Modifiability has become the single most important quality in most systems. Careful separation of concerns, grouping and hiding of design decisions that are likely to change together, separation of business functions from infrastructure technology, isolating computation from data and control transfer strategies, and interface-based specification are all important techniques to support modifiability.

Reusability: Are there implementation or design units in the system that are good candidates for reuse in another system? Does the system make good use of existing standard designs, interfaces, or implementations? Interestingly, systems that are designed to be easily modifiable for expected changes are also those that are the most reusable and the most maintainable.

Portability: Does the system design permit easy porting to other platforms? Are hardware and infrastructure dependencies localized in the implementation? Portability is gov-

erned almost completely by how well any idiosyncrasies of its computing environment have been encapsulated.

Buildability: Will the system as designed be easy to implement and build? What third-party components or libraries does it take advantage of? What tools will be used to assist in the construction process? Do these tools adequately support working with the components or libraries that will be used? Although this is rarely a stated requirement for the end user or customer, it is an important factor for the development team. Buildability is affected by the complexity of the design; whether it is counting on new, unproven technologies and tools; the appropriateness of the tools for the components being used; and the experience of the development team

Testability*:* How easy is it to demonstrate defects in the system by stimulating it with test data? Is it clear how to systematically define the test data based on the documented system architecture? At a technical level, testability is determined by how easy it is to access the internal state and inputs of the component so that they can be stimulated and observed. What's more, testing is an effort to show conformance (or, more properly, lack of conformance) of an implementation to a specification; so testability is determined to a great extent by clear specifications and rules about how to map from an implementation to the specification (see Chapter 6). Tests are an important part of any refinement relation in Catalysis. At least as much attention should be paid to specification and testing as to actual implementing. For large systems, it is worth explicitly documenting a *test architecture*, which can include hardware and software configurations, test tools used, and packages containing the test cases and results.

System qualities that do not correspond to runtime behaviors—modifiability, portability, buildability, and testability—must also be captured as part of requirements-gathering activities, although evaluating them can be more difficult (see Section 12.3, Architecture Evaluation with Scenarios).

12.2.2.2 Runtime Qualities

Many requirements can be measured only against a running system. The following most obvious ones—the requirements of the end user—fall into this category.

Functionality: Does the system, when deployed and running, assist its users in their tasks? Our previous discussion of business models and system type specification addresses this quality.

Usability: Does the system at runtime provide an intuitive interface and easily support the users' tasks? Does this apply to all categories of users? Building good business models so that system operations are designed as refinements of business tasks, and getting early user input on user-interfaces, are both important aspects of usability. There are also other, deeper issues of designing human-computer interactions that are outside our scope.

Performance: Does the system perform adequately when running—response time, number of events processed per second, number of concurrent users, and so on? In most large

systems, communication and synchronization costs between components, particularly across a network, dominate performance bottlenecks. Perceived response time can often be addressed by using multiple threads of control for asynchronous processing of a request. Performance can be modeled using the arrival rates of different stimuli to the system, the latency for different kinds of requests (processing time), and an approximation of delay caused by interference due to resource contention. The model can be checked as a back-of-the-envelope calculation, fed into a stochastic queueing model or simulation, or used to build a load driver for an architectural prototype. For many systems, raw performance is no longer the single most dominant quality.

Security: Does the system at runtime prevent unauthorized access or (mis)use? Security concerns typically include at least authentication—ensuring that the apparent source of a request is, in fact, who it claims to be—and authorization: ensuring that the authenticated user is permitted to access the needed set of resources. Security should be modeled as any other behavioral requirement is modeled. If an existing security mechanism or product is being used (for example, Kerberos), use a model of that mechanism as a part of your design models.

Reliability and availability: Does the running system reliably continue to perform correctly over extended periods of time? What proportion of time is the system up and running? In the presence of failure, does it degrade gracefully rather than shut down completely? Reliability is measured as the mean time to system failure; availability is the proportion of time the system is functioning. Both qualities are typically dealt with by making the architecture fault-tolerant: using duplicated hardware and software resources.

Scalability: Can the system as designed and deployed be scaled up to handle greater usage demands (volume of data, numbers of users, rate of requests)? Scalability is achieved primarily by replicating resources—processors, memory, storage media—and their software processing counterparts. Component-based designs in which the components can be deployed on separate processors, and where the overhead of cross-component coordination is proportionately small, enhances scalability.

Upgradability: Can the system at runtime be upgraded with new features or versions of software without bringing operations to a halt? Some systems must be operational continuously, and shutting them down for maintenance or upgrades is a serious matter. Systems that have a reflective core—the runtime keeps an explicit representation of the software structure, classes, interfaces, and so on and permits operations on that structure itself—can be the easiest to upgrade in this manner.

All the qualities that relate to runtime behavior should be captured as part of the behavioral specification of the system,[3] including requirements about security, availability, and performance, because they contribute to the definition of acceptable behavior. For example, the need to authenticate users before permitting them to access certain operations should be

3. They can often be in a separate section of the documents; our facility of joining combines all specifications of the same actions.

captured as part of the state transition model of the system. Similarly, the system may need to complete processing of certain requests in a timely manner for the components around it to function correctly.

12.2.2.3 A Single Key Quality

Other qualities also influence the design and development of a system. They include time to market, customizability to different products in a product family, development organization structure, distributed teams and their areas of expertise, and so on.

But despite this large number of different aspects of the "goodness" of an architecture, there is one single quality that dominates all others: the *conceptual integrity* of an architecture. This ephemeral quality summarizes an architecture's balance, simplicity, elegance, and practicality. A clean unifying architectural vision, and a consistency of design structures, can never be achieved by accident or by committee.

12.3 *Architecture Evaluation with Scenarios*

Although we can broadly state that an architecture that is simple is preferred to one that is not, we may want a more systematic way to evaluate architectural alternatives.

In Section 12.2.2, Many Qualities Are Affected by Architecture, we explained that there are many different qualities, both runtime and design-time qualities, that are affected by the architecture. Quantifying these qualities can be extremely difficult; for example, a design may easily permit one kind of modification but be resistant to another modification.

Hence, we must recognize that most "qualities" of a system are not absolutes but rather are meaningful only in specific contexts. A system is efficient only with respect to particular resources being consumed under particular usage profiles; it is modifiable under certain classes of requested changes.

Because each quality attribute corresponds to a stakeholder performing an action on the system either at design time or at runtime, we can use scenarios of such interactions to explore and evaluate an architecture. The main difference here from our original use of the word *scenario* (see Section 4.7.4, Scenarios) is that now we are not restricted to only runtime behaviors but include scenarios of system modifications, reuse, and so on.

The scenarios are used to rank the architecture qualitatively based on how well it handles the requirement, such as the number of components changed. Also, if scenarios that are largely independent affect common components, the responsibilities of those components may need reconsideration. Scenario-based evaluation is a relatively new technique, and we merely mention it here.

Here are some sample scenarios for the less obvious quality attributes.

- Make a batch program operate in an interactive mode and vice versa.
- Change an internal representation.
- Change an external interface that is known to be unstable.

- Add a new user function.
- Reuse a component in another system.
- Encrypt data being transmitted across a communication link.
- Integrate with a variety of e-mail systems.

12.4 *Architecture Builds on Defined Elements*

Simple digital systems are assembled from well-known libraries of parts: combinational logic gates (and, or, inverters, tri-state buffers); storage elements (flip-flops, registers, RAM); synchronization parts (clocks, dividers). From these parts are assembled a huge variety of systems, but all of them can be understood in terms of these basic parts.

Similarly, an architectural model is built from some number of elements: processors, modules, components, objects, class libraries, threads, and so on. A good architecture is based on a small set of design elements and uses them in a regular and consistent manner so that the system substructures are simple and similar.

The starting point for describing an architecture is to define the kinds of elements that constitute it. At the simplest level, interfaces and implementations are the elements of architecture. Beyond that, concrete implemented elements—specific kinds of buffers, synchronization primitives, coordinators, kits of parts to be assembled—can be specified as types or interfaces; more-abstract ones—design patterns, patterns of connectors and components—can be described using model frameworks. It is even possible for certain architectural qualities to be quantified, in a parameterized form, on the framework level. After these elements have been defined, the architecture itself can be described using these as "primitives."

12.4.1 Components and Connectors

As explained in Chapter 10, Components and Connectors, a component kit defines a set of components that are designed to work naturally together. The underlying idea is quite general, and frameworks let us define our own new kinds of components, ports, and connectors.

A sample architecture might be based on Cat One (see Section 10.8.1, Cat One: An Example Component Architecture. Its categories of ports include <<Event>>, an outgoing notification from a component to all other registered components; <<Property>>, a value that can be kept constantly in sync with another; and <<Transfer>>, wherein an object is moved from an output port to an input port. Using these ports, we can build a general-purpose set of components, such as those shown in Figure 12.2. An architecture can now be described as a configuration of such elements and their interconnections using the defined connectors.

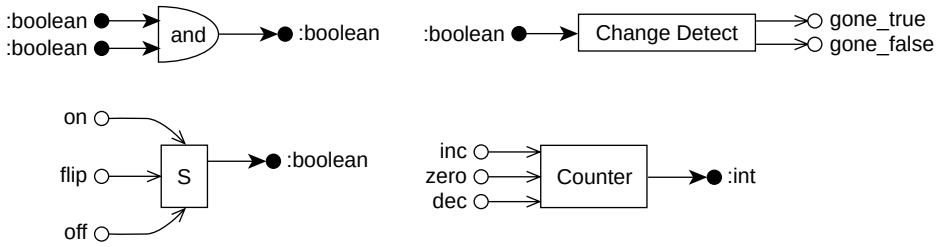


Figure 12.2 Sample low-level component kit.

12.4.2 Concurrency and Threads

A *thread* is a totally ordered set of events, or a sequential trace of execution, that can run concurrently with other threads. A thread is conceptually orthogonal to an object,[4] and its trace can involve the methods of many objects. Programming multithreaded systems is a tricky business, because you constantly must worry about the effects of concurrent activities; specifically

- Interference, in which one thread inadvertently interferes with state information that is being used by another
- Cooperation, in which two threads that are progressing concurrently at their own pace need to coordinate their activities at specific points in their execution

Just as programming language constructs relieve you of the tedium of assembler, multithreaded systems benefit from a library of concurrency and synchronization primitives.

- Bounded buffer, a thread-safe buffer with bounded capacity.
- Future, a "promise" of a return value that can be returned eagerly and passed around. When its value is finally accessed, it may block the accessor if the value is not yet available.
- Semaphore, an integer variable with an associated queue of waiting processes.
- Condition variables, state variables that a thread can synchronize on. A waiting thread is notified precisely when those variables change.
- Timers, which wake up at programmable times, take an action, and then go back to sleep.

12.4.3 Pipes and Filters

The well-known *pipe-and-filter* architecture can be described as another set of components and connectors. The components are filters: stateless components that transform their inputs into outputs; they have input and output ports corresponding to each input and output. The connectors are the pipes the interconnect filter outputs to inputs.

4. Except in the "active object model," which associates a thread with each active object.

12.4.4 Third-Party Libraries and Tools

Packages and their import structure define a development-time architectural model. In Catalysis, the imports define definitional and usage dependencies.

One architectural view includes all third-party libraries that will be used and import relations from packages that make use of these libraries. This view also explicitly describes the packages for tools (compilers, UI builders, and so on) that are used to populate other packages (containing object code, UI screens, and so on), documenting all project and module usage dependencies.

12.5 *Architecture Uses Consistent Patterns*

Having a set of design elements—components, connectors, object types—provides a higher-level language for describing an architecture. The use of these elements should also follow consistent patterns, and many of these patterns can themselves be formalized using frameworks. This section outlines a few such patterns; the details of formalizing them as frameworks is omitted. These examples are only illustrative not intended to serve as general prescriptions for architecture.

12.5.1 Event Notification Design Pattern

An event is an interesting change in state. There are many ways to design with events; this pattern defines one consistent style. To publish an event, E, from a component to interested subscribers, follow these steps.

1. Define the signature of E on an interface named E_Listener.
2. Add a pair of methods add_E_Listener and remove_E_Listener to manage the set of registered listeners on the component.
3. Document the event information in the parameters passed with E.

12.5.2 Subsystem Controller Pattern

Subsystems can interact with each other in many ways. This pattern defines a consistent scheme governing those interactions. For every subsystem, you may choose to uniformly have a distinguished *head object* that controls the connections between its children's ports and those in other subsystems based on a naming scheme. The head object also mediates all control and asynchronous communication between the subsystem and its parent system and coordinates the activities of its child components (see Figure 12.3). This arrangement gives a consistent structure for every subsystem: a head object, a defined role relative to its children, and a consistent protocol regardless of actual subsystem function.

12.5.3 Interface Packages Pattern

Packages can be structured in many different ways. The separation of an interface from classes in separate packages provides a pattern for setting up the project package structure. Combined with consistent naming conventions, it makes certain aspects of the architecture very visible in the project development structure (see Section 7.4.1, Role-based Decoupling of Classes).

12.5.4 Enterprise JavaBeans Pattern

The 1.0 specs of Enterprise JavaBeans are a good example of standard architectural patterns and how they can be used to define a simple and consistent architecture even for large-scale business systems.

12.5.5 Architectural Rules and Styles

In addition to the patterns themselves—event notification, subsystem controller—the architecture often dictates the rules that govern when these patterns must be applied and

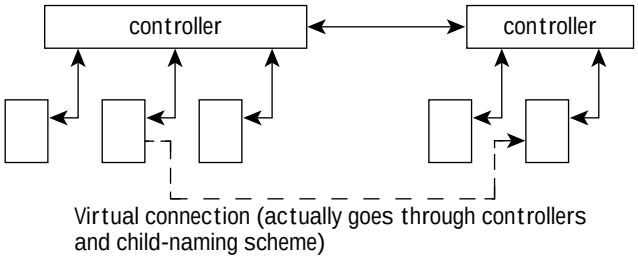


Figure 12.3 Consistent structure on subsystems.

when they should not be used. Thus, we may decide that every single subsystem must follow the subsystem controller pattern (CP). We can document this in an architectural package (see Figure 12.4).

Architectural rules can be documented using frameworks and other modeling constructs, including the ability to "say more" about a modeling element in another package. You can introduce shortcut notations to *tag* design elements as belonging to certain architectural categories, using stereotypes or customized notations. If formalizing the rules seems too heavyweight for some rules, describe them in concise prose or use whatever mechanism is appropriate. Regardless of how they are documented, all such rules should be explicitly placed into an architecture package that is imported by all relevant detailed design packages.

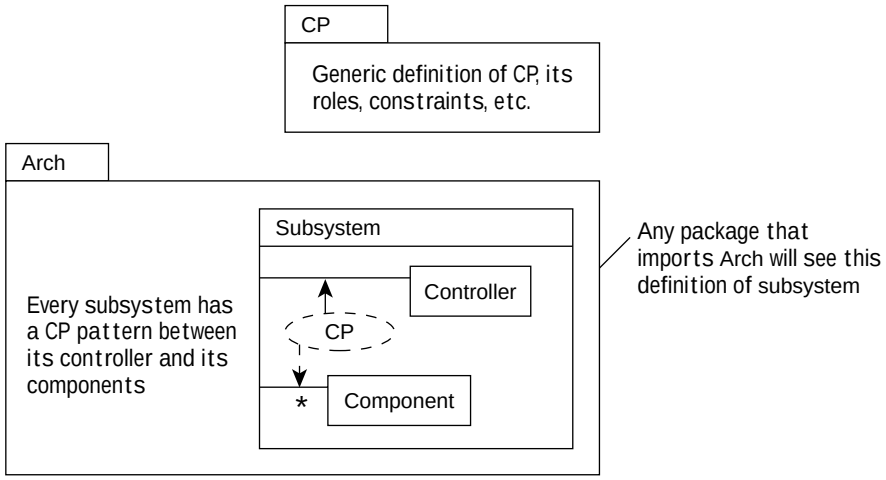
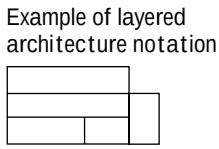


Figure 12.4 Defining the architectural rule for subsystem controllers.

© ***architectural style*** An architectural style (or *type*) is defined by a package (with nested packages) that has a consistent set of architectural elements, patterns, rules for using them, and stereotype or other notational shorthands for expressing their usage.

For example, *layered* architectures frequently use a special notation. A catalog of such architectural styles can be built in a way that's much like the use of frameworks in Section 9.8.2, Stereotypes and Dialects.



12.6 *Application versus Technical Architecture*

It is useful to separate the application architecture from the infrastructure parts (which we will call the *technical* architecture).

12.6.1 Application Architecture Partitions Business Logic

Application architecture is the partitioning of business logic across components and the design of their collaborations to meet the specified behavioral requirements. Application architecture is high-level logical design. In one sense it is entirely technology-independent, or at least the models are. However, how well a design fulfills other quality

attributes, most notably performance, depends quite dramatically on the underlying technology: machines, networking, middleware, and so on.

Hence, even though logical application architecture can be started early, it should piggyback on the physical and technical architecture to determine feasibility.

12.6.2 Technical Architecture Is Domain-Independent

The technical architecture includes all domain-independent design decisions, including the communication middleware that is used to enable communication between application architectural components; the patterns and rules that are followed in using the middleware; all domain-independent libraries that may be needed to build the system; and the rules for using them. The domain-independent facilities include the following:

- Communication middleware, the infrastructure that mediates between distributed components
- Command-line parser, a component to read commands and parse them against a command grammar
- State-machine interpreter, a means to interpret a reified representation of a state machine regardless of what the machine does
- Event logging, a facility to record all notable events, including communication, exceptions, and alarms
- Data access services, a layer that centralizes services to access the databases
- Exception signaling, the mechanism to signal, handle, and recover from various kinds of exceptions
- Start-up, monitor, shutdown, and failover, the mechanisms to start up the system, monitor its operations for things such as time-outs, shut the system down gracefully, and deal with failures
- Query processor, a means to deal with general ad hoc queries against any data model

12.6.3 Implement the Technical Architecture Early

Most project risk comes from two sources: business requirements and technology infrastructure. It is common for a project team to evaluate complexity based mostly on the business requirements—the problem domain itself—and vastly underestimate the effort it will take to implement all the plumbing and supporting pieces that are not domain-specific.

Unfortunately, until component-based development becomes the norm and until project managers understand the economics of buying the kinds of domain-independent components described earlier, we will still be building many of them. If most of the elements of the technical architecture are already *implemented*, then estimating the development time for business functions becomes much less like a black art.

Hence, you should implement the technical architecture as early as possible. Have all communication paths working—from user input and command-line parsing to the data

access services—even with the minimum of end-user functionality; and test that architecture for performance.

12.7 *Typical Four-Tier Business Architecture*

A typical Web-enabled business system might adopt an architecture with four physical tiers, as shown informally in Figure 12.5. The four tiers are as follows.

- *Clients:* Browsers and traditional GUIs can connect using HTTP to a Web server.

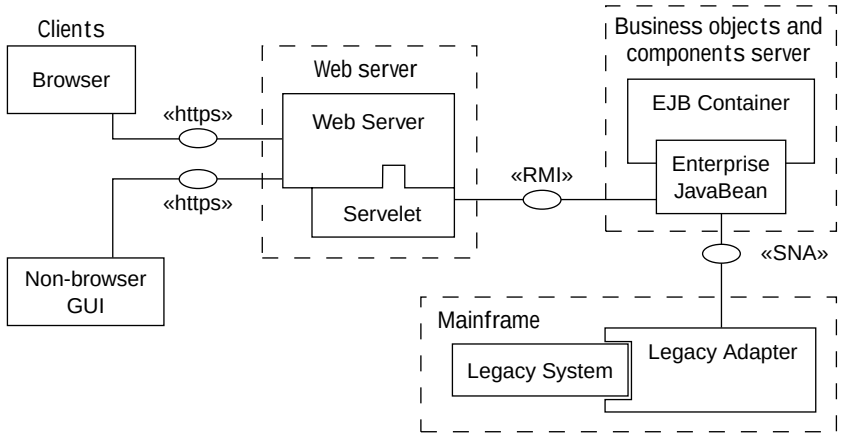


Figure 12.5 A Web-enabled business system with four physical tiers.

- *Web server:* This tier forwards appropriate requests to a *servelet*, a request handler that uses a standard API to plug in to the Web server. It translates CGI-encoded requests into proper method calls using distributed object invocations on an enterprise component. Then it formats and returns the results to the client.
- *Enterprise JavaBean:* This business component provides a meaningful business service using an uncluttered business type model via method invocation protocols such as Java RMI, or CORBA IIOP, or COM+ and communicating via a proprietary protocol to a legacy system and database. This component is also plugged in to an EJB container supplied by the legacy system vendor to provide security, transaction, threading, and other infrastructure services. This tier adds load balancing and failover behaviors, mostly provided by the container.
- *Legacy adapter:* This thin layer shields business objects from idiosyncrasies of the legacy system and its representations.

We can describe this architecture with a combination of a physical model, a component deployment model, and frameworks for the protocols used between these components. Moreover, the four-tier structure itself can be described as a framework.

12.8 *User Interfaces*

The Model-View-Controller (MVC) architecture (or some variant) is commonly used in the design of user interfaces. A user-interface element is associated with a single model—the object that acts as the source of the information presented, and the recipient of user requests. Each element is designed as a pair of a view, which deals with the presentation aspects of information from the model; and a controller, which is responsible for interpreting user inputs and gestures for the view and the model. The view and controller roles are combined into a single object in some variants.

When initially describing the system context, we avoid describing UI specifics, because these can vary. Detailed interactions via the user interface are better treated as a refinement of the abstract use cases that are being carried out. The basic MVC architecture applies to many forms of user interfaces, including graphical, character-based, touch screen, and voice-response.

At a higher level, it is useful to document the dialog flow of the UI, shown as a state transition diagram, where each state corresponds to a particular window that is active at that point in the interaction. At a more detailed level, we need uniform mechanisms to assemble the user interfaces and respond to their events.

There are various ways to build and connect user interfaces to business objects. The scheme you use depends largely on your user-interface frameworks. Following is a simple scheme broken into two main phases: configuration (creates and connects the appropriate objects) and run (user interacts with the widgets).

12.8.1 Configuration

For each window (or major panel within a window), follow these steps to configure the UI.

1. Create one application object. This object coordinates and mediates among the UI widgets and between the widgets and the business objects. The application object is created at the appropriate point in the dialog flow; in contrast, the business objects are created from persistent storage.
2. The application object creates its corresponding UI panel and populates it with its widgets.
3. The application object registers the widgets as its observer; later, a single changed method will inform all these widgets to update themselves.
4. The application object registers itself as the listener for events from each relevant widget; if necessary, it creates intermediate adapter objects to listen directly for these

events and to translate them into callbacks to the application object (for example, if you have two buttons, both of which generate a pressed event).

5. The application object registers itself as an observer for the relevant business objects. When they change they update the application object, which updates the widgets if necessary (see Figure 12.6).

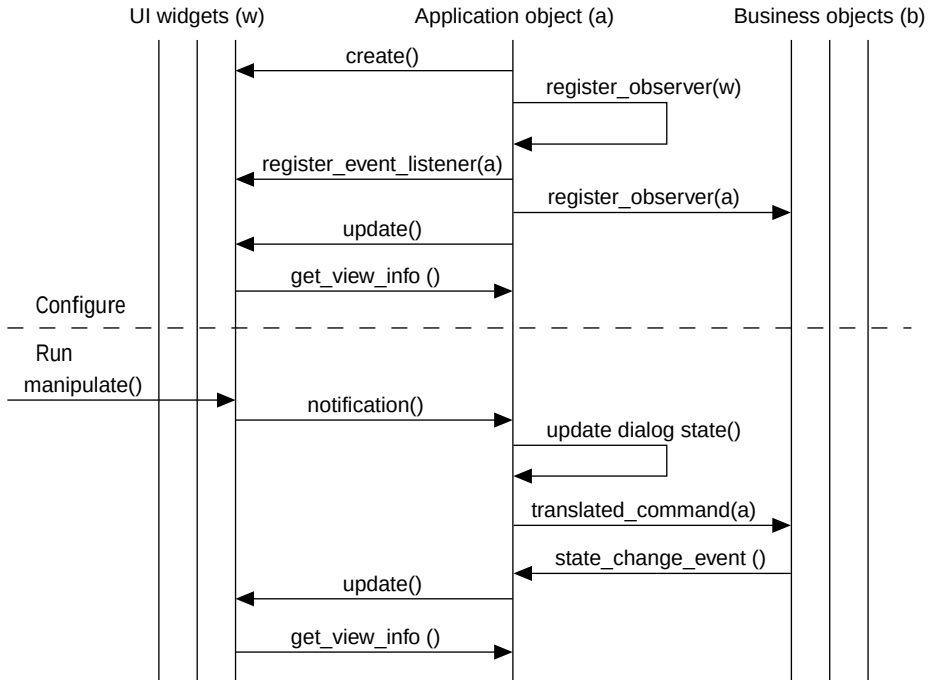


Figure 12.6 A typical UI interaction.

12.8.2 Run

During the run phase, the following actions occur.

1. The user manipulates a widget.
2. The widget generates the appropriate event to the application object (possibly through an intermediate adapter object).
3. The application object makes necessary changes to its state to reflect the progress of the interaction dialog; then it generates a command to the business object; then it propagates updates to the widgets.
4. The widgets update themselves by calling back to the application object for the state information they need (again, possibly translated by the intermediate adapter).

12.8.3 Reuse and the UI

You can sometimes make the user interface elements and the application object reusable so that you can create larger panels and application objects such as the following:

- video_title_selector: application object plus panel to help select a video title
- video_copy_scanner: application object plus panel to scan in a video copy
- member_selector: application object plus panel to select a member

Follow these steps.

1. Devise an event protocol to make these elements composable. For example, each application object checks with its container before it sends commands to its business objects.
2. If the application object represents an action in progress, name it as such.
   - check_out_video_application_object
   - return_video_application_object
3. Use typed event channels instead of generic subject-observer where necessary (see Section 12.5.1).
4. Use a scheme to batch the updates as needed.

12.8.4 Patterns in the UI

Define standard patterns to be followed in constructing the user interfaces. Many of these patterns can be formalized as frameworks. Examples are a master-slave listbox for navigating any 1-N association and a drill-down into data details using a tree panel.

12.9 *Objects and Databases*

In most applications the state of the business objects must persist even when the process or application that created them exits. There are three main approaches to providing this persistence.

12.9.1 Flat Files

Each object has the ability to serialize itself and also to initialize itself from a serialized representation. If the programming language has a reflective facility, you can write a single piece of code to determine the structure of the object and perform serialization and initialization. Java serialization works this way. Of course, flat files do not provide any of the multi-user, concurrency, meta-data, schema evolution, transaction, and recovery facilities that a database provides.

12.9.2 Relational Databases

In using a relational database, the interlinked object structure is translated into a representation based on tables and rows. Each object typically is a row in a table; one table is defined for each class (sans inheritance).

- The object identity is a primary key, usually generated by the database.
- Simple attributes are stored directly (if they correspond to built-in database types).
- Single-valued "object" attributes are stored as a foreign key referring to another table.
- Multivalued associations cannot be stored directly. Instead, they are stored as an inverse foreign key, and a join is performed to traverse the association.
- Inheritance cannot be mapped directly. There are several alternative schemes, well documented elsewhere, that are systematic and make different trade-offs.

The overall disadvantages of relational databases are as follows.

- Because of the different models of programming and persistence, impedence mismatch becomes a major development cost. It is alleviated by products that help automate the mapping.
- The cost of multiple joins can become prohibitive for a highly structured data model. However, some relational databases have recently shown tremendous improvements in this area.
- The data access is usually not integrated with the programming language.

Conversely, relational databases have a number of advantages.

- Relational technology has been around for a while. It is mature and well understood at both a formal and a practical level.
- Query languages include declarative access and querying using SQL.
- Relational technology permits the use of multiple views: virtual "schema" that can be treated like the real schema, at least for query purposes.
- Relational databases are scalable and easily handle the terabyte range and hundreds of users.

12.9.3 Object-Oriented Databases

With object-oriented databases, the interlinked object model is transparently carried over to persistence, and the developer is concerned with logical things—transaction boundaries, concurrency, how far to propagate locks—rather than mapping between two dissimilar models.

The disadvantages of this approach include the following.

- Object-oriented databases scale to very large datasets but they do not have the maturity of relational products.
- OODBs traditionally have not supported declarative query access, choosing explicit navigation instead. This situation is starting to change, with standards such as the ODMG.

- Commercial OODBs do not provide a flexible mechanism for multiple views of a common underlying schema.
- The simplicity of the mapping can be deceptive. You still have to carefully design the volume of persistent data, granularity of objects that are made persistent, and transaction boundaries.

The advantages are as follows.

- OODBs present no major conceptual barrier for persistence.
- Within their scaling limits, OODBs traditionally have vastly outperformed relational databases.

12.9.4 Hybrid Object-Relational Databases

These purport to provide the best of both worlds. They are a relatively new technology and are worth watching as they mature. The growth of the World Wide Web has spurred on this segment significantly, because WWW demands better support for complex structured documents and other data.

12.10 *Summary*

The architecture of a system is the set of design decisions that constrain its implementors and maintainers. A good architecture is one that defines a small set of elements to be used in a consistent and regular way, and one in which conformance to the architecture can be clearly defined.

The architecture of a system has a direct impact on its qualities—performance, funtionality, understandability, maintainability, portability, testability, functionality, scalability, and so on. Scenarios are a promising way to evaluate an architecture against these qualities.

There are many different architectural views of a system, calling for different kinds of elements. These elements include hardware and networks, packages and their structures, object types and relationships, concurrent processes and threads, tables and columns, and the patterns that dictate how they are to be used. An architectural style, or type, defines a consistent set of elements and rules for their use.