upon the number of concurrently executing modules, each of which is likely executing on a separate physical processor. In other words, scalability is achieved through sheer distribution.

One limitation, of course, is that not all problems can be easily parallelized in this manner. Unless the *problem* is naturally, easily parallelized, increasing the scale can happen at the significant expense of efficiency.

### Place the Data Sources Close to the Data Consumers

If components that need to access some data are distributed across a network, the decision to have a single, remotely located data storage component will result in a lot of network traffic. This, in turn, will affect the system's performance and its ability to accommodate additional client components that may yet further increase the data traffic. One solution is to always keep the data sources close to data consumers, minimizing the resulting network traffic, and even co-locating them on the same host whenever possible.

Clearly, this guideline will be difficult to apply in many distributed systems. Moving the data closer to the processing components can be done virtually through techniques such as caching and prefetching, as discussed earlier. However, in such cases the system needs to ensure that multiple copies of the same data are synchronized, which may add to the system's processing and communications load beyond the savings incurred by making data access local.

Another possibility is to co-locate the processing components. However, the application's location constraints may limit the architect's options. Furthermore, even if a given component could be redeployed to another host (that is, closer to the data), its interactions with other parts of the system may suffer as a result.

### Try to Make Distribution Transparent

Relocating either processing or data to improve a system's performance or scalability in the manner discussed above requires some degree of location transparency. We have discussed the potential negative impact of a component's location transparency on efficiency. At the same time, location transparency can aid a system's scalability.

In a distributed setting, scaling up a system will often result in changing the deployment view of the system's architecture. For example, new users may need to be allowed to access the system via new hardware hosts. Processing or data components may need to be added in specific places in the architecture, or they may need to be relocated across hosts, to enable the new users. The outcome will be increased processing and data traffic in the system.

Redeployment may cause significant disturbances to a system. However, if connectors are capable of handling both local and remote interactions in a manner that is transparent to the involved components, then the system can be reconfigured in several ways, such as by redeploying consumer components closer to the needed data sources, replicating remote functionality locally for performance, or off-loading components to capacious remote hosts.

The impact of such adaptations always needs to be assessed carefully before they are effected.

### Use Appropriate Architectural Styles

The architectural styles selected for a given system will have a significant influence on that system's scalability. As has been the case with other system properties, even if we do not

Sony Exhibit 1018 Page 494
Sony v. Quantum IPR2023-00954

take into account specific details of the system under development, certain architectural styles will be more appropriate for achieving scalability than others. Thus, for example, publish-subscribe and event-based architectures have been demonstrated to scale to very large numbers of components, users, devices, and data volumes. Other styles typically result in systems that scale well in one or more dimensions. For example, interpreter-based systems cannot easily accommodate growing numbers of users, but can usually handle addition of new functional operators. Likewise, pipe-and-filter may be unable to support increasingly large volumes of data, but can support increasing numbers of components, since arbitrarily long pipelines can be constructed.

## 12.4 ADAPTABILITY

***Definition.*** ***Adaptability*** **is a software system's ability to satisfy new requirements and adjust to new operating conditions during its lifetime.**

Adaptability can be manual or automated. A software system's architecture has an impact on either type of adaptability. Chapter 14 is dedicated in its entirety to architectural adaptation, so we will make only brief general observations here.

### 12.4.1 Software Components and Adaptability

Architecturally relevant adaptability occurs at the level of system components, their interfaces, and their composition. In other words, if adaptation is required entirely *within* an individual component or connector, that adaptation is not considered to be architectural. Such adaptations can still be effected with the aid of the system's architecture, for example, by replacing an entire component with its newer version. This observation informs several guidelines for designing for adaptability.

#### Give Each Component a Single, Clearly Defined Purpose

This guideline has been discussed above—twice—in the context of efficiency and complexity. Since architecture-level adaptation occurs at the level of entire components, it is imperative to separate different system concerns into multiple components. This allows the architects and engineers to minimize the amount of degradation the system experiences during adaptation.

#### Minimize Component Interdependencies

Adapting a complex system is difficult. Each modification may impact multiple parts of the system. For example, Figure 12-5 indicates that modifying any major subsystem of Linux will, in principle, have an effect on every other subsystem. Defining the system's components to have simple interfaces, in a manner that precludes unnecessary interdependencies, can help to control the effects of adaptation.

#### Avoid Burdening Components with Interaction Responsibilities

Again, this guideline was discussed previously in the context of other NFPs. In the context of adaptability, the objective is to separate the system's functionality and data from interaction.

Sony Exhibit 1018 Page 495
Sony v. Quantum IPR2023-00954

#### Separate Processing from Data

Adaptations to the system's processing components should be handled independently of adaptations to its data.

#### Separate Data from Meta-Data

Changes to the data in a large, long-lived software system will occur regularly. If the data is separated from the meta-data (that is, the data about the data), then in principle each can be adapted independently of the other. Furthermore, the processing components will be able to adapt more easily to the changes in both the data and the meta-data.

### 12.4.2 Software Connectors and Adaptability

Connectors are the key enablers of architectural adaptability. Components should be insulated from their particular context to the greatest extent possible in order to appropriately separate system concerns and maximize their reuse potential. It is the task of connectors to provide the necessary facilities that enable a given component to operate appropriately in its environment. Several guidelines for adaptability stem from this observation.

#### Give Each Connector a Clearly Defined Responsibility

If a connector is in charge of enabling the interactions of a given type among a specific set of components, it will be easier for architects and engineers to manage the required adaptations to those components or their interactions. However, this requires one additional property of connectors, discussed next.

#### Make the Connectors Flexible

At the least, connectors must be able to support different numbers of components, and possibly component types. If a connector is unable to do so by itself, composing it with other connectors may produce the desired effect, as elaborated next.

#### Support Connector Composability

To enable interactions among heterogeneous components, which may be added to a system during run time, connectors must be composable with other connectors. For example, Figure 12-6 shows a connector composed from three separate object request brokers (ORBs)—recall the discussion on CORBA from Chapter 4. In the architecture depicted in Figure 12-6, each component exchanges information only with the connector to which it is directly attached; in turn, the connector will (re)package that information and deliver it to its recipients using one or more middleware technologies—whatever is necessary to achieve the desired communication. Each such middleware-enabled connector exposes the interface expected by its attached components. The connector may change the underlying mechanism for marshaling and delivering messages, but externally appears unchanged.

Such a composite connector has the added advantage that it can also preserve the topological and stylistic constraints of the application. For example, the application in Figure 12-6 was designed according to the C2 style (recall Chapter 4). This means that, for example, Component A and Component B can interact with one another, while Component A and

Sony Exhibit 1018 Page 496
Sony v. Quantum IPR2023-00954

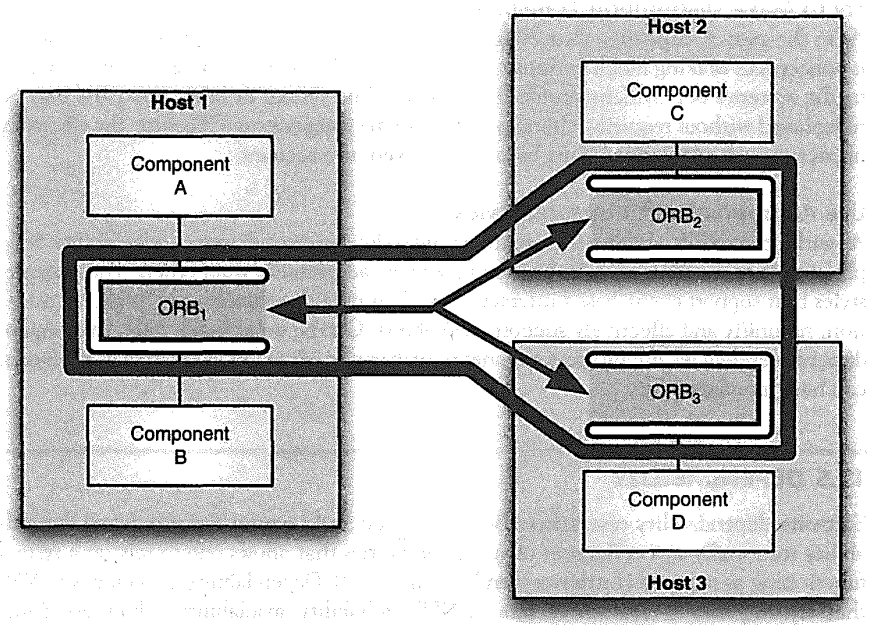



Figure 12-6. *Distributed components interact through a single conceptual connector, which is actually composed from multiple interacting ORBs. Each ORB exports the interfaces expected by its attached components. To do so, the ORB may need to include an adaptor, depicted as a highlighted ORB edge.*

Component C cannot. Using a single ORB to connect the components—even if it were possible—would potentially violate these stylistic constraints.

Another advantage of such composite connectors is that they are independent of other connectors in the system, and can optimize aspects of communication in a homogeneous setting (for instance, if the interaction between Component A and Component B is local). Maintaining a connector's interface and semantics, coupled with its composability, allows much more flexibility in application development and deployment. For instance, if it were later decided that Component C and Component D should indeed run in the same process, it would be possible to make this change by simply reconfiguring the composite connector. The component code would not have to be changed at all.

#### Be Aware of Differences Between Direct and Indirect Dependencies

This guideline has been discussed above in the context of scalability. Much of the argument is applicable for adaptability. In principle, direct and explicit dependencies result in more efficient systems, while indirect and implicit dependencies allow the given system to adapt more easily.

### 12.4.3 Architectural Configurations and Adaptability

#### Leverage Explicit Connectors

A software system in which the connectors are implicit (that is, in which components are endowed with interaction capabilities) will be difficult to adapt since individual concerns may be distributed across multiple system elements.

Sony Exhibit 1018 Page 497
Sony v. Quantum IPR2023-00954

### Try to Make Distribution Transparent

As in the case of scalability, distribution transparency has its advantages when adaptability is concerned: Making modifications to a system is much easier if components are oblivious to the system's deployment profile. For example, the system or one of its parts may be redeployed without requiring changes to the system's components. Of course, the efficiency impact of such adaptations must be carefully taken into account.

### Use Appropriate Architectural Styles

As with other NFPs, architectural patterns and styles can directly impact adaptability. Simply put, certain styles are better suited to support adaptability than others. For example, styles that support event-based interaction, such as publish-subscribe and implicit invocation, naturally and effectively support adaptability. On the other hand, styles that require direct dependencies among the components, such as virtual machines or distributed objects, can hamper adaptability.

## 12.5 DEPENDABILITY

Software dependability researchers Bev Littlewood and Lorenzo Strigini define dependability informally as a collection of system properties that allows one to rely on a system functioning as required (Littlewood and Strigini 2000). Dependability is a composite NFP that encompasses several other, related NFPs: reliability, availability, robustness, fault-tolerance, survivability, safety, and security. Security is discussed in detail in Chapter 13; here, we focus on the design guidelines required to ensure the remaining facets of dependability in a system's software architecture. The following definitions illustrate the interrelated nature of these different facets of dependability. The role of software architecture in ensuring a software system's dependability is then discussed.

> ***Definition.*** **A software system's *reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure, over a given time period.**

> ***Definition.*** **A software system's *availability* is the probability that the system is operational at a particular time.**

Unlike reliability, which is a statistical measure of the system's overall health over a continuous period of time, availability represents the system's health at discrete snapshots in time.

> ***Definition.*** **A software system is *robust* if it is able to respond adequately to unanticipated run time conditions.**

Robustness deals with those run time circumstances that have not been captured in a system's requirements specification. It can be said that responding to conditions specified in the requirements is a matter of system correctness, while responding to all other conditions is a matter of robustness. The adequacy of the response will vary depending on different factors, such as application domain, user, and execution context.

> ***Definition.*** **A software system is *fault-tolerant* if it is able to respond gracefully to failures at run time.**

The failures affecting a system can be outside the system itself, whether caused by hardware malfunctions or system software defects, such as network or device driver failures. Alter-

Sony Exhibit 1018 Page 498
Sony v. Quantum IPR2023-00954

natively, the system's own components may fail due to internal defects. What constitutes a graceful reaction to failure will depend on the system context. For example, one system may continue operating in a degraded mode; another one may introduce, possibly only temporarily, new copies of the failed components; yet another may offer the "blue screen of death" to the consternation of its users, requiring a reboot of the entire computer, after which the user might be able to continue with normal operation.

From a software architectural perspective, faults can be classified into the following.

1. Faults in the system's environment (that is, outside the software architecture).
2. Faults in components.
3. Faults in connectors.
4. Component-connector mismatches.

An NFP closely related to fault-tolerance is survivability.

***Definition. Survivability* is a software system's ability to resist, recognize, recover from, and adapt to mission-compromising threats.**

Survivability must address three basic kinds of threats, cited below.

1. *Attacks*, examples of which are intrusions, probes, and denials of service.
2. *Failures*, which can be due to system deficiencies or defects in external elements on which the system depends.
3. *Accidents*, which are randomly occurring but potentially damaging events such as natural disasters.

Therefore, fault tolerance and survivability are related because the goal of both is to effectively combat threats, and system faults can be viewed as one kind of threat.

It should be noted that the distinctions among the three categories of threats may not matter in the context of recovery from the threats. The key to survivability is to try to recover from these threats as gracefully as possible. In order to make a system survivable, the basic activities that can be performed are the following.

1. Resistance to threats.
2. Recognition of threat and extent of damage.
3. Recovery of full and essential services after a threat.
4. Adaptation and evolution to reduce the impact of future attacks.

The final facet of dependability that will be discussed here is software system safety. Several definitions of software safety are currently in use, and they usually are variations on the definition provided by software safety expert Nancy Leveson (Leveson 1995). We provide a similar definition.

***Definition. Safety* denotes the ability of a software system to avoid failures that will result in loss of life, injury, significant damage to property, or destruction of property.**

It should be noted that different degrees of property damage and destruction, and even human injury, may be considered acceptable in the context of different systems and/or system usage scenarios. The elaboration of such circumstances is outside the scope of this book, however.

Sony Exhibit 1018 Page 499
Sony v. Quantum IPR2023-00954

### 12.5.1 Software Components and Dependability

More dependable software components will, on the average, result in more dependable software systems—though it should be noted that dependable components need not always result in dependable systems, and that highly dependable systems may comprise undependable individual components (recall the "Honey-Baked Ham" sidebar in Chapter 8). Practice has shown that it is seldom possible to develop components that are completely reliable, robust, and fault-tolerant. However, engineers can follow certain practices to help with achieving these properties. In particular, the guidelines below are targeted at the outwardly visible facets of a component.

#### Carefully Control External Component Interdependencies

Changes in the behavior of a given component, including anomalous behavior and failures, should have a minimal impact on the remaining components in the system. This can be achieved by properly insulating components from one another. One specific guideline is to restrict all intercomponent dependencies to be explicit and only at the level of the components' public interfaces. Recall from the discussion earlier in this chapter that following this guideline may compromise a system's scalability and adaptability. Another guideline is to minimize, or completely disallow, side effects from component operations.

#### Provide Reflection Capabilities in Components

While a given component in a system may be unable to provide certain desired dependability guarantees, it should be possible to partially mitigate that by enabling querying of the internal state of the component. This will allow other parts of the system to assess the health of the component at times that are deemed important.

#### Provide Suitable Exception Handling Mechanisms

When an individual component fails, the rest of the system may be able to adjust to that failure. In order for that adjustment to happen quickly and gracefully, the failing component should be able to expose the necessary information to the rest of the system. One way of doing this is by providing exception reporting mechanisms via the component's public interface. This will allow the other components and connectors in the system to adjust their operation accordingly.

#### Specify the Components' Key State Invariants

Architects can explicitly state conditions that must hold at different times during the component's execution. Such invariants will allow the component's users to establish best-case, normative, and worst-case guarantees when interacting with that component. Different actions taken in response to that information will then have the potential to improve the overall system's dependability.

### 12.5.2 Software Connectors and Dependability

#### Employ Connectors that Strictly Control Component Dependencies

Unintended or implicit dependencies among a system's components will likely harm a system's dependability. Explicit, first-class connectors can be used to manage all dependencies among system components. If necessary, connectors can completely insulate components

Sony Exhibit 1018 Page 500
Sony v. Quantum IPR2023-00954

from one another. It should be noted that widely used connectors such as shared memory and procedure calls give developers too much leeway in creating component interdependencies. Many distributed systems employ connectors that, by necessity, strictly enforce component boundaries and control remote interactions. However, recall from Chapter 5 that even within a single address space it is possible to leverage connectors, such as event buses, that will enforce any component interaction requirements.

#### Provide Appropriate Component Interaction Guarantees

In certain scenarios, it is imperative that a component receive the data sent to it, even if that means sending the information multiple times. In other scenarios, the system's hardware resources may be overtaxed to the point that, regardless of other needs, the system cannot afford to have the same data transmitted or processed more than once. In yet other circumstances, the recipient components may process each event, even though these are replicas of the same event. In that case, the system's state may erroneously change and the results produced may be wrong. This could have disastrous consequences in a safety-critical system: Consider receiving, multiple times, a command to change the pitch, roll, or yaw of an aircraft by a specified value.

In large, distributed systems—especially those composed with heterogeneous, possibly off-the-shelf components—the individual components may, for instance, make different assumptions about each other, the desired interaction profiles, or the state of the hardware resources. In such situations, it is the responsibility of connectors to ensure that all interaction guarantees (for example, at least once, at most once, or exactly once) are provided, and possibly adapted in response to changing circumstances during the system's execution.

#### Support Dependability Techniques via Advanced Connectors

A number of dependability concerns and needs in a system cannot be met simply by constructing solid components, properly composing them, and ensuring that they interact only in the prescribed ways. Large, complex, distributed, embedded, and mobile systems need to respond to many external stimuli, and can suffer failures induced by human users, hardware malfunctions, or interference—whether accidental or malicious—from other software systems.

To deal with such situations, connectors may need to support advanced interaction facilities. Examples include providing replicas of failing or failed components on the fly, run time replacement of components (also referred to as "hot swaps"), and support for multiple versions of the same functionality, for example, to ensure the correctness of calculations. These facilities will often need to be provided such that the system's existing, still correctly functioning components are unaware of them. In other words, connectors should support *seamless* dependability.

### 12.5.3 Architectural Configurations and Dependability

#### Avoid Single Points of Failure

In many large systems, a large portion of a system may depend on the services provided by one component or a small number of components. If those services are critical to the system's mission, the failure of the component, or components, providing them will significantly or completely incapacitate the system.

Sony Exhibit 1018 Page 501
Sony v. Quantum IPR2023-00954

There are numerous examples of such systems. For instance, any data-intensive system with a centralized repository cannot afford to lose the repository. Likewise, a spacecraft with a centralized controller will be lost if the controller fails. Yet another example is a swarm of robots relying on a centralized planning engine; the robots may end up roaming aimlessly if the planner malfunctions.

There are several techniques that can be employed to deal with such concerns, including replicating the component in question, clearly separating into multiple components the different concerns it may embody, and employing connectors with advanced capabilities such as those discussed immediately above. At the same time, it should be acknowledged that in some situations it may be impossible to avoid creating a design that contains one or more single points of failure. Regardless of the circumstances (that is, whether the single point of failure can be avoided or not), software architects need to be aware of this issue throughout the design process.

### Provide Back-Ups of Critical Functionality and Data

This guideline should be obvious. It also is not universally applicable. For example, it is possible that a lack of system resources or hard real-time constraints effectively prevent back-ups. However, as with one's personal computer or PDA, back-ups are ultimately the best way of ensuring that critical functionality and data are not lost. Different techniques may be used to implement this guideline, including employing advanced connectors such as those discussed above.

### Support Nonintrusive System Health Monitoring

It may be too late to try to deal with system malfunctions once they have already occurred. Frequently, however, a system will exhibit anomalous behavior and give hints that something is wrong for some time before it, or a part of it, actually fails. Periodically monitoring the system for certain events that indicate its health status can help engineers to anticipate, and possibly eliminate, any unexpected behavior.

Nonintrusive monitoring can be achieved in a number of different ways. One possibility is to employ a component model that supports the inclusion of explicit monitors at the level of application components. Another, even less intrusive possibility is to include monitoring facilities within the system's connectors. Component containers used in many middleware platforms present another possible target for placing system health monitors.

### Support Dynamic Adaptation

A dynamically adaptable system is able to respond to events at run time; a static system may not. Dynamically adaptable systems are able to support the addition, removal, replacement, and reconnection of both components and connectors while the system is running. In decentralized settings they frequently also support dynamic discovery of available services. A thorough treatment of the role of software architecture in dynamic adaptation is provided in Chapter 14.

It should be noted that dynamic adaptation can harm a system's dependability, at least temporarily. Recall the discussion of mobility in Chapter 10 and, in particular, the diagram in Figure 10-12. In addition to rendering the system at least temporarily undependable *during* the dynamic adaptation, it is possible that the result of adaptation will be a system whose critical properties, including its dependability, are compromised. This is a reason why the stakeholders of many highly safety-critical systems are leery of dynamically modifying

Sony Exhibit 1018 Page 502
Sony v. Quantum IPR2023-00954

their systems. In other words, while dynamic adaptation can be a very useful tool at an architect's disposal, it should be applied with caution.

## 12.6 END MATTER

Engineering a software system to provide the required functionality is often quite challenging. This is why historically a majority of the software design and implementation techniques have focused on supporting and improving a system's functional capabilities. Ultimately, however, good engineers will usually manage to produce almost any functionality, no matter how complex. What they will still struggle with are the non-functional aspects of their systems.

Recall the discussion of the "better-cheaper-faster — pick any two" software engineering maxim from Chapter 10. It suggests that, so long as the system stakeholders are willing to sacrifice one or more of the system's non-functional characteristics, such as schedule, cost, or performance, developers will be able to produce the desired functional capabilities.

Producing those capabilities *along with* the desired non-functional system characteristics is significantly more challenging. There is much less guidance available, in particular to software system architects, for how such characteristics are to be "designed into" the system in question. One reason is that many of these characteristics are loosely defined and qualitative in nature. The challenge becomes even more daunting when system stakeholders have to consider multiple, possibly clashing non-functional characteristics simultaneously.

This chapter has distilled a large body of experience and good software engineering practices into a set of guidelines [or "tactics," as they have been referred to elsewhere in literature, for instance, (Rozanski and Woods 2005)] that software architects can follow in achieving specific non-functional properties. The guidelines should work most of the time, but they should be applied with caution. Whenever appropriate, each of the guidelines in this chapter has been coupled with specific admonitions that reflect the difficulties a software architect may encounter in certain development scenarios.

Both the guidelines and the accompanying admonitions should be kept in mind when designing software systems. The reader should also realize that this chapter did not, and could not, provide a complete enumeration of all possible architectural design scenarios, or all possible trade-offs among the choices made in achieving the desired non-functional characteristics. Chapter 13, which follows, is a companion to this chapter, in that it provides in-depth treatment of security, an NFP of tremendous importance in modern software systems. Together, these two chapters are intended to serve as a rich foundation for software designers targeting NFPs.

**The Business Case**

Achieving desired non-functional properties often makes the difference between a successful product and a failure. There are many examples from the software development industry. Just recall the example of Microsoft Word 6.0, released in the mid-1990s. The Windows version of the product performed well and was successful. The Macintosh version, with roughly the same functionality, was large and slow, and resulted in a fair amount of bad publicity for Microsoft. This example is telling also in that attempts at patching the product proved unsuccessful, so that Microsoft eventually had to revert to Word 5.0 for its Macintosh user base.

While Microsoft was able to weather this particular storm, a smaller company in its place may not have been able to recover as easily. Inserting a non-functional property into a system after

Sony Exhibit 1018 Page 503
Sony v. Quantum IPR2023-00954

the system has been designed, and possibly implemented, is rarely a successful strategy, and it is fraught with dangers. A more appropriate approach is to plan for the desired properties from the project's inception and to ensure that the necessary architectural design decisions are made. In many situations, even knowing what *not* to do, or at least which pitfalls to avoid, in designing the system can be quite valuable. The guidelines provided in this chapter can serve as a useful tool in that regard.

## 12.7 REVIEW QUESTIONS

1. Define the following NFPs:
   a. Efficiency
   b. Complexity
   c. Adaptability
   d. Scalability
2. What are the different dimensions of dependability?
3. How is each of the above NFPs manifested in software architectures?
4. How can each of the above NFPs be addressed at the architectural level?
5. Identify at least three guidelines for achieving each of the above NFPs. Elaborate on the manner in which each of those guidelines supports the NFP in question.
6. Can multiple NFPs be achieved simultaneously? If so, under what conditions? If not, why not?

## 12.8 EXERCISES

1. It is often difficult to simultaneously satisfy multiple non-functional properties (NFPs). Discuss the trade-offs between the following NFPs with respect to their impact on a system's architecture. For each trade-off, provide the answer in three parts: Discuss how your architectural choices can help to maximize each of the two individual properties (possibly at the expense of the other property); then discuss architectural choices that can help you maximize both properties in tandem. Make sure to provide your answer at least in terms of the role and specific characteristics of components and connectors that impact the NFPs.
   a. Performance versus complexity
   b. Safety versus efficiency
   c. Reliability versus adaptability
2. Recall the example Lunar Lander architectures discussed in Chapter 4. Select any two of the architectures from Chapter 4. For each NFP discussed in this chapter, analyze what can and cannot be determined about the selected architectures. Can you recognize any of the NFP design guidelines in either of the two architectures? What is your assessment of the architectures' respective support for the NFP under consideration?
3. Now consider an ADL model of Lunar Lander from Chapter 6. Does the model aid or hamper your ability to answer the questions from the preceding exercise? Be sure to justify your answer.
4. Discuss how one of the architectures selected in the two preceding exercises can be adapted to improve:
   a. Efficiency
   b. Scalability
   c. Adaptability
5. The list of guidelines for achieving the different NFPs provided in this chapter is incomplete. Add at least one additional guideline for each existing NFP.
6. Devise a set of design guidelines for achieving additional properties, such as:
   a. Heterogeneity
   b. Compositionality
   c. Security

   If necessary, you should locate the definitions and any other necessary explanation of these properties. Note that security is discussed in Chapter 13.
7. Analyze the pair–wise trade-offs among one of the properties you devised in the previous exercise and those introduced in this chapter.

Sony Exhibit 1018 Page 504
Sony v. Quantum IPR2023-00954

## 12.9 FURTHER READING

Several software engineering textbooks provide useful overviews of non-functional properties (NFPs). Carlo Ghezzi, Mendi Jazayeri, and Dino Mandrioli (Ghezzi, Jazayeri, and Mandrioli 2003) are particularly thorough in their treatment of NFPs. They divide the NFPs into internal (relevant to architects and developers) and external (relevant to customers and users). Furthermore, they divide NFPs into those relevant to the developed product and those relevant to the development process. The NFPs studied in this chapter are the internal product NFPs.

Ghezzi et al. do not focus on the design guidelines for accomplishing the various NFPs. A recent book by Nick Rozanski and Eoin Woods (Rozanski and Woods 2005) attempts to do just that. The authors identify a set of NFPs relevant to software architects. These include performance, scalability, security, and availability. They also propose a number of guidelines, called tactics, targeted at achieving the NFPs. Several of their tactics are relatively general (such as "capture the availability requirements"), and are not targeted separately at different architectural facets such as components, connectors, and configurations.

A volume on the *Future of Software Engineering (FoSE)* (Finkelstein 2000), accompanying the proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), provided a set of useful overviews and research roadmaps for several NFPs, including reliability and dependability (Littlewood and Strigini 2000), performance (Pooley 2000), and safety (Lutz 2000). These topics have been revisited in the *FoSE* volume (Briand and Wolf 2007) accompanying the 29th International Conference on Software Engineering (ICSE 2007) by Michael Lyu on reliability (Lyu 2007), Murray Woodside et al. on performance (Woodside, Franks, and Petriu 2007), and Mats Heimdahl on safety (Heimdahl 2007).

Many of the architectural guidelines advocated in this chapter targeted at accomplishing NFPs emerged over time from general software engineering principles. For example, modularity and separation of concerns was articulated by David Parnas more than thirty years ago (Parnas 1972). More recently, Robert DeLine has argued for a decoupling of a component's "essence" from its "packaging" (DeLine 2001). Robert Allen, David Garlan, and John Ockerbloom have shared a very useful experience on the effects of off-the-shelf component integration on a system's NFPs, and the inherent architectural causes (Garlan, Allen, and Ockerbloom 1995; Garlan, Allen, and Ockerbloom 1995).

Sony Exhibit 1018 Page 505
Sony v. Quantum IPR2023-00954

# 13

# Security and Trust

The preceding chapter introduced several non-functional properties, such as efficiency, scalability, and dependability, and described architectural design strategies to help achieve those properties. Security is another non-functional property; its increasing, critical importance warrants the separate, in-depth look provided in this chapter. As with many other non-functional properties, it is most effectively addressed while designing a system's architecture.

Consider the example of building architectures introduced in Chapter 1. A building is designed with various structural properties and the owner's requirements in mind. If such requirements and design do not encompass security needs, problems can arise. For instance, if a building has windows or doors that are easy to access from the outside, or its structure prevents the installation of security alarms, the building may be vulnerable to unwanted visitors. If these considerations are addressed during the building's design, however, a secure structure at a reasonable price is achievable.

If the building is not designed from the outset with security in mind, it may still be possible to add external reinforcements to improve security when such demands later occur. For example, thin walls can be reinforced by adding extra layers; doors and windows that represent potential points of entry can be safeguarded using suitable lock mechanisms. In some cases, a building may not be securable by itself but it might be housed within a gated enclosure that can safeguard an entire community against external intruders.

The caveat of adding security afterward, though, is that it is generally more expensive than taking the proper measures from the beginning. Imagine opening up the walls, installing extra wires and cameras, then closing the walls. This is more expensive than envisioning the requirements from the start and designing the building around those requirements. The same is true for software. It is therefore imperative that security be considered and treated when developing a system's architecture. Using a software architecture-based approach for security allows developers to leverage experience and



Sony Exhibit 1018 Page 506
Sony v. Quantum IPR2023-00954

achieve desired security properties. Software architecture also provides a sound basis for reasoning about security properties.

Note that while external reinforcements can be used to provide a certain degree of security post-development, it still entails that the software be designed to allow addition of such reinforcements without compromising required functionalities. This makes reasoning about security at the architectural level even more important. Further, since software systems often go through an extended round of releases as new functionalities are added, a security-based architectural approach can provide guidance through the various software evolution cycles and help ensure that essential security properties continue to be achieved through each release.

Security, as important as it is, is only a part of the overall system. It has to be balanced with other non-functional properties. For example, encryption is generally used in software systems to keep data secret, but using encryption can be computationally expensive, and the performance of the system might be adversely impacted by such operations. Even more importantly, security, along with other non-functional properties, must be balanced against a system's general functional requirements. For example, a browser that displays and executes all types of content would provide the richest experience for users, but such indiscreet execution is almost doomed to bring malicious software into a user's computer. When facing such choices, uninformed stakeholders, such as end-users and product-planning teams, might choose functionality over other critical properties. A software architecture approach can ameliorate this condition by providing the necessary abstraction and tools that will help stakeholders to make sound decisions.

The chapter begins with introduction of the different aspects of security, including confidentiality, integrity, and availability. Section 13.2 discusses several general design principles for security. These principles have been developed by theoreticians and practitioners over the years and have been applied in many systems. We illustrate how these principles can be applied to software architectures. In some contrast to the preceding chapter, however, the design guidance provided is not as straightforward. Properties such as efficiency and dependability have been important to software designers since the beginning of software engineering, hence it is not surprising that the preceding chapter was replete with crisp techniques for achieving them. Security is of more recent prominence, and is arguably more subtle, so a broad understanding and application of general design principles is needed. In Section 13.3, a technique for architectural access control is presented that complements other design techniques with capabilities to specify and regulate intercomponent communication. The chapter concludes with presentation in section 13.4 of an architectural approach for constructing trust-enabled decentralized applications. These types of applications play an important role

Sony Exhibit 1018 Page 507
Sony v. Quantum IPR2023-00954

in the emerging collaborative Web world, where autonomous users communicate and collaborate in a community environment and require trust management to protect themselves from malicious users.

**Outline of Chapter 13**

13. Security and Trust
13.1 Security
13.2 Design Principles
13.3 Architectural Access Control
13.3.1 Access Control Models
13.3.2 Connector-centric Architectural Access Control
13.4 Trust Management
13.4.1 Trust
13.4.2 Trust Model
13.4.3 Reputation-Based Systems
13.4.4 Architectural Approach to Decentralized Trust Management
13.5 End Matter
13.6 Review Questions
13.7 Exercises
13.8 Further Reading

## 13.1 SECURITY

The National Institute of Standards and Technology defines computer security as, "The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)" (Guttman and Roback 1995). According to this definition, there are three main aspects of security: confidentiality, integrity, and availability. We briefly introduce them here; for a comprehensive treatment, see the references in the "Further Reading" section at the end of the chapter.

### Confidentiality

Preserving the confidentiality of information means preventing unauthorized parties from accessing the information or perhaps even being aware of the existence of the information. Confidentiality is also referred to as secrecy.

This concept is as applicable in the domain of building architectures as it is to computer systems. For example, in a large office complex having two buildings, if the office management does not want others to know when certain items are moved between the two buildings, the management could build a covered passage between them so that others outside the building are unable to see what is moved within the passage. Even better, the passage could be built underground so people would not even be aware of the existence of the passage.

Applying this concept to software architectures, software systems should take proper measures while exchanging information to protect confidential information from being intercepted by rogue parties. Likewise, systems should store sensitive data in a secure way so unauthorized users cannot discover the content or even the existence of such data.

Sony Exhibit 1018 Page 508
Sony v. Quantum IPR2023-00954

## Cryptography

A well-known measure to support secrecy in software systems is use of encryption techniques. Cryptography itself has been used since the beginning of civilization, and the wide use of computer systems facilitates the wide adoption of computer cryptography. A basic encryption mechanism can be described with the following equations:

$$Cipher = EncryptionFunction(Encryption_Key, ClearText)$$

$$ClearText = DecryptionFunction(DecryptionKey, Cipher)$$

When information needs to be encrypted to preserve confidentiality during communication or storage, the information, called clear text, is fed into an encryption algorithm, along with an encryption key, to produce a cipher. To retrieve the original information stored in the cipher, the cipher can be fed into a decryption algorithm, along with the decryption key, to produce the original clear text.

An encryption function has a corresponding decryption function. After using a specific encryption function, only the corresponding decryption function can be used to decrypt the cipher.

There are two forms of cryptography functions. With *shared-key cryptography* the same key is used by both the encryption and decryption functions. This type of scheme has been used since the earliest days of cryptography. Management of the shared key presents problems, however. Since the key is shared between the sender and the receiver, the key must be delivered to both the sender and the receiver, which can be awkward if a sender wants to communicate with many receivers. Moreover since best security practices demand that keys not be reused (in order to defeat cryptanalysis efforts by opponents), these key deliveries must be performed every time a key is changed.

To solve the key management problem with the shared-key scheme, *public key cryptography* was invented in 1970s. These systems use a key pair, which consists of two related keys that are generated at the same time. One key, the private key, is retained and kept secret by an individual party. The corresponding key, the public key, is published by the individual and shared with everyone. To send information secretly, the initiator encrypts the clear text with the public key of the *recipient*, and sends the cipher text. The recipient's private key is used by the individual to decrypt the message. If the keys are appropriately generated it is computationally infeasible for a third party to determine the individual's private key from the known public key, hence confidential communication is ensured.

Besides secrecy, public key cryptography can also be used to provide nonrepudiation. The private key can be used to generate a digital signature. Such a signature is similar to a regular handwritten signature in that only the person who possesses the private key can generate the signature. Thus, receiving information having such a digital signature is a sure sign that the sender has sent the information, since no one can generate the signature without the private key. Recipients can use the sender's public key to verify signatures, thus establishing that the information was indeed sent by the sender (nonrepudiation) and that it is authentic and was not modified along the way.

While a public key mechanism is flexible and mathematically sound, it suffers from a performance problem. It is generally orders of magnitude slower than a shared-key algorithm. Thus, a choice exists between private key and public key: the former is fast, but is inflexible, and requires complex key management mechanism; the latter provides scalability and ease of management, but is much slower. An architectural solution that combines both of them is to use a public key mechanism to negotiate a shared key in the beginning of the communication session, and then apply the negotiated key with a shared-key encryption algorithm in the ensuing communications.

Sony Exhibit 1018 Page 509
Sony v. Quantum IPR2023-00954

An important principle while designing a cryptography based system is to avoid designing yet another encryption algorithm. Such algorithm design requires extensive mathematical skills, which is outside the scope of most projects. If you think you can design a secret encryption algorithm, then the admonition, "Do not depend on secrecy for security," is applicable here. It is almost inevitable that a determined hacker will expose the operational details of the newly proposed algorithm, and the "security" based on secrecy is lost. Instead, to design secure software based on cryptography to achieve secrecy and nonrepudiation, a designer should evaluate performance, architecture, and security requirements, choose a suitable public algorithm, and use frequently changing keys as the primary secrecy mechanism.

### Integrity

Maintaining the integrity of information means that only authorized parties can manipulate the information and do so only in authorized ways. For example, in a building protected by a door that can be opened by entering an access code on a numeric panel, only the owner should be able to change the code. Moreover, when the code is changed, it should only be changeable to another numeric code—not to an alphanumeric one.

Analogous constructs exist at the programming language level, such as access modifiers of member fields and functions. For example, in Java, member fields and functions are given access levels as `private`, `package`, `protected`, and `public`, such that only certain parts of the class hierarchy can access those members. A `private` field of a class can only be accessed by the methods of the same class, whereas a `package` method can be called by all classes of the same package. Likewise, in C++, `const` pointers can be defined that enforce the condition that data accessed through those pointers cannot be changed.

At the software component and architectural level, similar protective mechanisms can be applied to the interfaces of components. For example, if a particular interface of a component changes the most critical information belonging to that component, invocation of that interface should be ensured to be limited to only authorized components. Therefore, such an interface should be designated and separated from the others and receive more scrutiny during design.

To establish the identity of a user—and hence to determine whether a user is authorized or not—an authentication process is used to verify that the user is really who the user claims to be. The most common form of authentication is the user name/password pair: If a user can correctly supply the password associated with a user name, then the user is authenticated as that specific user. This is a form of authentication that relies on what a user knows; other forms of authentication include checking who the user is (for example, by scanning the iris of the user and comparing it to a set of authorized irises) and what the user has. (For example, a user must posses a security token that can generate the correct number at the correct time.)

Depending on security requirements, different levels of authentication may be used for software components and connectors. For example, in the Microsoft DCOM middleware technology, authentication may be bypassed completely. However, if needed, authentication can be performed at the beginning of a communication session, for each method, or even for each communication packet. The most secure authentication level is, of course, the packet authentication level, but it is also the most computationally expensive. The

Sony Exhibit 1018 Page 510
Sony v. Quantum IPR2023-00954

authentication requirements of a communicating client and server determine what the DCOM middleware connector must do. In particular, the middleware connector ensures that both parties operate on the chosen authentication level.

To deter potential intruders, a software system can maintain an audit trail that records important historical information. By analogy, the security guard of a gated housing community can record every visitor's name, license plate, and visiting time, so any security incident could be correlated to possible suspects. Likewise, security cameras can be deployed to record the activities of residents. (Of course, such measures have to be balanced against privacy requirements!) Correspondingly, in the case of software components, audit trails can be maintained internally; that is, a component may log requests and responses from an authenticated user and then produce an audit trail of those requests and responses at a later time. Connectors may also be used to log component invocations that pass through the connector. Further, since the architecture provides a systemwide view of the configuration of components and connectors, an audit trail can be captured recording patterns of access through the system.

### Availability

Resources are *available* if they are accessible by authorized parties on all appropriate occasions. In contrast, if a system cannot deliver its services to its authorized users because of the activities of malicious users, then its services are unavailable; it is said to be the victim of a denial of service (DoS) attack.

Applications that are distributed across a network, such as the Internet, may be susceptible to a distributed denial of service (DDoS) attack. Such attacks try to bring down the many distributed elements of an application. For example, the Domain Name System (DNS), which is in charge of resolving URLs to IP addresses and is distributed across different levels of operation, has occasionally been the target of such attacks. When such attacks succeed, access to Web resources, for example, may be denied.

## 13.2 DESIGN PRINCIPLES

Security aspects of software systems should be considered from a project's start. During system conception the security requirements should be identified and corresponding security measures designed. Patching security problems after a system is built can be prohibitively expensive, if not technically infeasible. Security requirements also evolve with other requirements. Thus, an architect should anticipate possible changes and design flexibility into the security architecture. The architecture of the system is the place for software developers to identify the security requirements, design the security solutions, and design to accommodate future changes.

This section highlights several design principles that help guide the design of secure software. These principles emerged from the research community and have since been applied in many commercial software systems. Such principles are by no means sufficient by themselves for the design of secure software, but do play an important role in guiding designers and architects through possible alternatives and choosing an appropriate solution.

**Design Principles for Computer Security**

- *Least privilege:* Give each component only the privileges it requires.
- *Fail-safe defaults:* Deny access if explicit permission is absent.
- *Economy of mechanism:* Adopt simple security mechanisms.
- *Complete mediation:* Ensure every access is permitted.
- *Open design:* Do not rely on secrecy for security.
- *Separation of privilege:* Introduce multiple parties to avoid exploitation of privileges.
- *Least common mechanism:* Limit critical resource sharing to only a few mechanisms.
- *Psychological acceptability:* Make security mechanisms usable.
- *Defense in depth:* Have multiple layers of countermeasures.

These principles adapted from (Bishop 2003), (Saltzer and Schroeder 1975), and (Gasser 1988).

### Principle of Least Privilege

The principle of least privilege states that a subject should be given only those privileges it needs to complete its task. The rationale is that even if a subject is compromised, the attacker has access only to a limited set of privileges, which limits the damage to certain specific parts of the system.

Currently, many less-informed Windows users browse the Internet using an account with many administrative privileges. This is not only unnecessary for the simple task of browsing the Web but is potentially dangerous since it opens paths for malicious software to take control of the user's computer. This practice owes its origin to early versions of Windows that were generally shipped with only one account, the administrator account. Based on the principle of least privilege, a minimally privileged account should be used for daily simple activities such as browsing and e-mail. Embodying this principle, Internet Explorer 7, shipped in late 2006, can lower its privileges during execution to below those of the launching user's privileges.

Software architecture makes it easier to determine the least privileges components should have since explicit models of the architecture enable analysis of communication and control paths to determine the necessary attributes. A component should not be given more privileges than are necessary for it to interact with other appropriate components.

### Principle of Fail-Safe Defaults

The principle of fail-safe defaults states that unless a subject is granted explicit access to an object, it should be denied access to that object. This scheme might deny some safe requests that otherwise would have been granted, but it assures that each granted access is a safe access.

A simple illustration of this principle is the case of Internet browsers requesting a resource ("GETing a URL") on behalf of a user. Fetching and displaying a resource is a form of granting permissions based on the user-selected URL. Since a URL can be expressed and encoded in different forms (such as using absolute paths versus relative paths), it is not always straightforward to list and reject *all* invalid URLs. Thus, this rule suggests that accesses to all URLs should be denied unless their form can be verified as belonging to a known, valid kind. Based on this principle, a connector connecting

Sony Exhibit 1018 Page 512
Sony v. Quantum IPR2023-00954

two components should only allow the specific communications that satisfy some approval criterion, rejecting all others.

### Principle of Economy of Mechanism

The principle of economy of mechanism states that security mechanisms should be as simple as possible—also referred as the KISS principle (Keep it Simple and Small). While this rule generally is useful with any type of design, it is especially important for security systems. Complexity is the enemy of security because complex interactions make verifying the security of software systems more difficult and hence could possibly lead to a security breach.

One way to apply this principle is to isolate, consolidate, and minimize security controls. Redundant security mechanisms should be simplified. For example, in Internet Explorer prior to Version 7, there were multiple places where URLs were analyzed and results of these analyses were used to make decisions. Such redundancy and inconsistency led to security vulnerabilities. This issue was corrected in Internet Explorer Version 7 by centralizing the handling of URLs. An architecture description provides a suitable abstraction to apply this principle more generally. It allows architects to analyze the locations of security controls, identify potential redundancy, and evaluate alternatives to choose a suitable place for the control.

### Principle of Complete Mediation

The principle of complete mediation requires that all accesses to entities be checked to ensure that they are allowed, irrespective of who is accessing what. The check should also ensure that the attempted access does not violate any security properties.

Applying this principle to a software system requires all communication to be checked thoroughly. Such an inspection is greatly facilitated through the systematic view of the system provided by an accurate architectural model. A security architect can evaluate each possible interaction among the components in all types of configurations to make sure that none of the interactions and configurations violate the intended security rules.

The principle of economy of mechanism helps achieve complete mediation. Where there are only a limited number of security control mechanisms it is easier to apply security control, verifying that each access actually goes through these mechanisms.

### Principle of Open Design

The principle of open design states that the security of a mechanism should not depend upon the secrecy of its design or implementation. While secrecy is a desired security property, secrecy itself should not be used as a mechanism. A secure design should not rely on the fact that an intruder does not know the internal operations of the software system. While keeping the internals secret might initially make it more difficult for an attacker to break into a system, simply relying on such secrecy is unreliable. It is inevitable that such information will be discovered by malicious users in a world where many different types of information and computational resources are available to attackers. Trivially, employees could leak the secret, either intentionally or unintentionally. Among other options, the attacker can also try clever reverse engineering or simple brute force attacks.

Revealing the internals of a system can actually increase its security. In early stages of design other security reviewers can inspect and evaluate the design and provide insights.

Sony Exhibit 1018 Page 513
Sony v. Quantum IPR2023-00954

Further, during its operation and evolution phases, the system can be studied and refined accordingly to make it more secure. For instance, a system's security should not rely upon a software connector implementing a proprietary (secret) communication protocol. The likelihood of that idiosyncratic communication protocol having flaws is very high. Rather, using a protocol that has passed extensive external scrutiny is far more likely to provide the security desired.

### Principle of Separation of Privilege

The principle of separation of privilege states that a system should not grant permission based on a single condition. It suggests that sensitive operations should require the cooperation of more than one key party. For example, a purchase order request generally should not be approvable solely by the requestor; otherwise, an unethical employee could keep requesting and approving inappropriate purchase orders without immediate detection by others.

Software architecture descriptions facilitate the checking of this principle. If an architect discovers that some component possesses multiple privileges that should be separated, the architect should redesign the system and the component so that the privileges are partitioned amongst multiple components.

### Principle of Least Common Mechanism

The principle of least common mechanism states that mechanisms used to access separate resources should not be shared. The objective of the principle is to avoid the situation where errors or compromises of the mechanism while accessing one resource allow compromise of all resources accessible by the mechanism. For instance, use of separate machines, separate networks, or virtual machines can help fulfill this principle and avoid cross-contamination.

In the context of software architectures, this implies the need for careful scrutiny when certain software architectural styles are used. For example, in the case of the blackboard style, where all data is maintained on the shared blackboard and access to it is mediated by the blackboard component, the architect must ensure that the existence of the shared store and common mechanism does not introduce unintended security problems.

### Principle of Psychological Acceptability

The principle of psychological acceptability states that security mechanisms should not make the resource more difficult to access for legitimate users than if the security mechanisms were not present. Likewise, the human interface of security mechanisms should be designed to match the mental model of the users and should be easily usable. Otherwise, the users will either attempt to bypass the security measure because it is too difficult to use, or use it incorrectly because the user interface is error-prone.

This principle did not receive much attention in the past, but now that software has become a mainstream phenomenon and most computer users are not technically savvy, it is increasingly important to design security mechanisms keeping users' psychological acceptability in mind.

By analogy, a building may have several security capabilities to safeguard it, such as specially designed door and security alarms. Yet if the building owner does not use them because they are too cumbersome or error-prone, then essentially the building becomes as vulnerable to potential threats as if the safeguards did not exist. With regard to software

Sony Exhibit 1018 Page 514
Sony v. Quantum IPR2023-00954

systems, an application may support security techniques such as digital authentication and cryptography, but if the end users do not use those techniques because they do not understand the mechanisms or cannot use the mechanisms correctly, the resulting system may become vulnerable to security attacks such as impersonation and repudiation.

**Bruce Schneier, on Security Systems, rather than Security Technologies**

The principle of psychological acceptability has been emphasized by Bruce Schneier, a security expert and the author of *Applied Cryptography* (Schneier 1995). In the preface to his subsequent book, *Secrets and Lies* (Schneier 2000) he writes,

> Cryptography is a branch of mathematics. And like all mathematics, it involves numbers, equations, and logic. Security, palpable security that you or I might find useful in our lives, involves people: things people know, relationships between people, people and how they relate to machines. Digital security involves computers: complex, unstable, buggy computers.
>
> Mathematics is perfect; reality is subjective. Mathematics is defined; computers are ornery. Mathematics is logical; people are erratic, capricious, and barely comprehensible.
>
> The error of *Applied Cryptography* is that I didn't talk at all about the context. I talked about cryptography as if it were The Answer™. I was pretty naïve.
>
> The result wasn't pretty. Readers believed that cryptography was a kind of magic security dust that they could sprinkle over their software and make it secure. That they could invoke magic spells like "128-bit key" and "public-key infrastructure." A colleague once told me that the world was full of bad security systems designed by people who read *Applied Cryptography*.
>
> Since writing the book, I have made a living as a cryptography consultant: designing and analyzing security systems. To my initial surprise, I found that the weak points had nothing to do with the mathematics. They were in the hardware, the software, the networks, and the people. Beautiful pieces of mathematics were made irrelevant through bad programming, a lousy operating system, or someone's bad password choice. I learned to look beyond the cryptography, at the entire system, to find weaknesses. I started repeating a couple of sentiments you'll find throughout this book: "Security is a chain; it's only as secure as the weakest link." "Security is a process, not a product."
>
> Any real-world system is a complicated series of interconnections. Security must permeate the system: its components and connections.

### Principle of Defense in Depth

The principle of defense in depth states that a system should have multiple defensive countermeasures to discourage potential attackers. Since an attacker will have to break through each of these countermeasures, it increases the likelihood of being able to identify and prevent an attack from occurring.

This principle requires each component in a path that leads to a critical component to implement proper security measures in its own context. This ensures that the security of the whole system will not be violated just because of one component's failure to implement proper security control.

A good example is the way Microsoft Internet Information Service (IIS) Version 6 (a Web server) handles WebDAV requests. By re-architecting IIS, utilizing the underlying support provided by the operating system, and applying appropriate security measures at

| POTENTIAL PROBLEM | PROTECTION MECHANISM | DESIGN PRINCIPLES |
|---|---|---|
| The underlying dll (ntdll.dll) was not vulnerable because... | Code was made more conservative during the Security Push. | Check precondition |
| Even if it were vulnerable... | Internet Information Services (IIS) 6.0 is not running by default on Windows Server 2003. | Secure by default |
| Even if it were running... | IIS 6.0 does not have WebDAV enabled by default. | Secure by default |
| Even if Web-based Distributed Authoring and Versioning (WebDAV) had been enabled... | The maximum URL length in IIS 6.0 is 16 Kbytes by default (>64 Kbytes needed for the exploit). | Tighten precondition, secure by default |
| Even if the buffer were large enough... | The process halts rather than executes malicious code due to buffer-overrun detection code inserted by the compiler. | Tighten postcondition, check precondition |
| Even if there were an exploitable buffer overrun... | It would have occurred in w2wp.exe, which is running as a network service (rather than as administrator). | Least privilege<br><br>(Data courtesy of David Aucsmith) |

Figure 13-1. *Security for Microsoft IIS. Table data from Table 1 in (Wing 2003) © IEEE 2003.*

multiple points along the access path, IIS has become a far more secure system than its previous versions. The different mechanisms applied by different components along the WebDAV access path are shown in Figure 13-1.

This principle does not contradict the principle of economy of mechanism because it does not duplicate identical security checks, or worse, implement similar but inconsistent checks. Instead, each component provides unique security safeguards that are most appropriate in its local context and thus helps to collectively form a more secure system.

## 13.3 ARCHITECTURAL ACCESS CONTROL

Having introduced the design principles for building secure software, we now present one technique, architectural access control, to demonstrate how software architects can follow the above-described principles in designing secure software systems. We define the basic access control models in security, illustrate how these models can be applied during architectural design, introduce software tools that facilitate the utilization of these models, and, through examples, show how these concepts and techniques can be practiced.

Sony Exhibit 1018 Page 516
Sony v. Quantum IPR2023-00954

### 13.3.1 Access Control Models

The most basic security mechanism used to enforce secure access is a *reference monitor*. A reference monitor controls access to protected resources and decides whether access should be granted or denied. The reference monitor must intercept every possible access from external subjects to the secured resources and ensure that the access does not violate any policy. Widely accepted practices require a reference monitor to be tamper-proof, non-bypassable, and small. A reference monitor should be tamper-proof so that it cannot be altered. It should be non-bypassable so that each access is mediated by the reference monitor. It should be small so that it can be thoroughly verified.

Two dominant types of access control models are discretionary access control (DAC) models and mandatory access control (MAC) models. In a discretionary model, access is based on the identity of the requestor, the accessed resource, and whether the requestor has permission to access the resource. This permission can be granted or revoked at the resource owner's discretion. In contrast, in a mandatory model, the access decision is made according to a policy specified by a central authority.

#### Classic Discretionary Access Control

The Access Matrix Model is the most commonly used discretionary access control model. It was first proposed by Butler Lampson (Lampson 1974) and later formalized by Michael Harrison, Walter Ruzzo, and Jeffrey Ullman (Harrison, Ruzzo, and Ullman 1976). In this model, a system contains a set of subjects (also called principals) that have privileges (also called permissions) and a set of objects on which these privileges can be exercised. An access matrix specifies the privilege a subject has on a particular object. The rows of the matrix correspond to the subjects, the columns correspond to the objects, and each cell lists the allowed privileges that the subject has on the object. The access matrix can be implemented directly resulting in an authorization table. More commonly, it is implemented as an access control list (ACL), where the matrix is stored by column, and each object has one column that specifies the privileges each subject has over the object. A less common implementation is a capability system, where the access matrix is stored by rows, and each subject has a row that specifies the privileges (capabilities) that the subject has over all objects.

#### Role-Based Access Control

A role-based access control (RBAC) model is a more recent extension of the classic access control model. In this model, an extra level of indirection, called a role, is introduced. Roles become the entities that are authorized with permissions. Instead of authorizing a user's access to an object directly, the authorization is expressed as a role's permissions to an object and the user can be assigned to the corresponding role. RBAC allows roles to form a hierarchy. In such a hierarchical RBAC model, a senior role can inherit from a junior role. Every user that takes the senior role can also take the junior role, thus obtaining all the permissions associated with the junior role. The RBAC model, thus, eases management of access control in large-scale organizations. Instead of granting and revoking permissions individually to many users, all relevant users can be assigned a single role, and the permissions can be granted and revoked to this role. Role-based access control allows a clear specification of the roles that cannot be performed simultaneously by a user.

Sony Exhibit 1018 Page 517
Sony v. Quantum IPR2023-00954

### Mandatory Access Control

Mandatory access control models are less common and more stringent than discretionary models. They can prevent both direct and indirect inappropriate access to a resource. The most common types of mandatory models work in a multilevel security (MLS) environment, which is typical in a military setting. In that environment, each subject (denoting a user) and each object are assigned a security label. These labels have a dominance relationship between them. For example, the top-secret label dominates the classified information label. A subject can only access information whose label is dominated by the label of the subject. Thus, a subject with only classified information clearance cannot access top secret information, but a subject with top secret clearance is able to access content that is labeled classified information.

## 13.3.2 Connector–Centric Architectural Access Control

This section presents a connector-centric approach that describes one way in which the above-described access control models can be applied and enforced at the architectural level. Specifically, we describe how an architectural description can be extended to model security and how the resultant description can be checked to examine whether the architecture successfully addresses the security needs of the system.

### Basic Concepts

The core concepts that are necessary to model access control at the architecture level are *subject*, *principal*, *resource*, *privilege*, *safeguard*, and *policy*.

***Subject.*** A *subject* is the user on whose behalf a piece of software executes. The concept of subject is key in security, but is typically missing from software architectural models. Many software architectures assume that (a) all of its components and connectors execute under the same subject, (b) this subject can be determined at design-time, (c) the subject generally will not change during run time, either inadvertently or intentionally, and (d) even if there is a change, it will have no impact on the software architecture. As a result, there is typically no modeling facility to capture the allowed subjects of architectural components and connectors. Consequently, the allowed subjects cannot be checked against actual subjects at execution time to ensure security conformance. In order to address these needs for architectural access control, basic component and connector constructs must be extended with the subject for which they perform, thus enabling architectural design and analysis based on different security subjects.

***Principal.*** A subject can take upon it multiple *principals*. Essentially, principals encapsulate the credentials that a subject possesses to acquire permissions. There are different types of credentials. In the classic access control model, the principal is synonymous with the subject and directly denotes the identity of the subject. But there exist other types of principals that provide indirection and abstraction necessary for more advanced access control models. In a role-based access control model, each principal can denote one role that the user adopts. The results for accessing resources will vary depending on the different principals a subject possesses.

***Resource.*** A *resource* is an entity for which access should be protected. Example resources and access controls on them are files that should be read-only, password databases that

should only be modified by administrators, and ports that should only be opened by the root user. Traditionally, resources are *passive* and accessed by active software components operating for different subjects. However, in the case of software architecture, resources can also be *active*. Specifically, software components and connectors may also be considered resources, access to which should be protected. Such an active view is lacking in traditional architectural modeling. Explicitly enabling this view can give architects more analysis and design power to improve security assurance.

***Permission, Privilege, and Safeguard.*** *Permissions* describe operations on a resource that a component may perform. A *privilege* describes what permissions a component possesses depending upon the executing subject. Privilege is an important security concept that is missing from traditional architecture description languages. Most current modeling approaches take a maximum privilege route wherein a component's interfaces list all the privileges that that component could possibly need. This could become a source for privilege escalation vulnerabilities that are caused when a less privileged component is given more privileges than it properly should be granted in a particular usage context. A more disciplined modeling of privileges is therefore needed to reduce such vulnerabilities.

There are two types of privileges corresponding to the two types of resources. The first type handles passive resources and enumerates, for instance, which subject has read/write access to which files. The second type deals with active resources. These privileges include architecturally important privileges such as instantiation and destruction of architectural elements, connection of components with connectors, execution through message routing or procedure invocation, and reading and writing architecturally critical information. These privileges are pivotal in ensuring secure execution of software systems.

A notion corresponding to privilege is *safeguard*, which describes conditions that are required to access the interfaces of protected components and connectors. A safeguard attached to a component or a connector specifies the privileges that other components and connectors must possess before they can access the protected component or connector.

***Policy.*** A *policy* ties together the concepts above. It specifies what privileges a subject, with a given set of principals, should have in order to access resources that are protected by safeguards. It is the foundation needed by the architectural elements to make access control decisions. Components and connectors consult the policy to decide whether an architectural access should be granted or denied.

### The Central Role of Architectural Connectors

Architectural access control is centered on connectors because connectors propagate privileges that are necessary for access control decisions. They regulate communication between components and can also support secure message routing.

***Components: Supply Security Contract.*** A security *contract* specifies the privileges and safeguards of an architectural element.

In the ensuing discussion, for purposes of specific illustration, we will utilize and refer to modeling architectures using the xADL language (Dashofy, van der Hoek, and Taylor 2005), as presented in Chapter 6. For component types, the above modeling constructs are modeled as extensions to the base xADL types. The extended security modeling constructs

Sony Exhibit 1018 Page 519
Sony v. Quantum IPR2023-00954

describe the subject the component type acts for, the principals this component type can take, and the privileges the component type possesses.

The base xADL component type supplies interface signatures that describe the basic functionality of components of this type. These signatures become the active resources that should be protected. Thus, each interface signature is augmented with safeguards that specify the necessary privileges an accessing component must possess before the interfaces can be accessed.

***Connectors: Regulate and Enforce Contract.*** Connectors play a key role in regulating and enforcing the security contract specified by components. They can determine the subjects for which the connected components are executing. For example, in a normal SSL (secure socket layer) connector, the server authenticates itself to the client, thus the client knows the executing subject of the server. A stronger SSL connector can also require client authentication, thus both the server component and the client component know the executing subjects of each other.

Connectors also determine whether components have sufficient privileges to communicate through the connectors. For example, a connector can use the information about the privileges of connected components to decide whether a component executing under a certain subject can deliver a request to the serving component. This regulation is subject to the policy specification of the connector. The recent version of DCOM, for example, introduces such regulation on local and remote connections.

Connectors can also potentially serve to provide secure interaction between insecure components. Since many components in component-based software engineering can only be used "as is" and many of them do not have corresponding security descriptions, a connector is a suitable place to assure appropriate security. A connector decides which communications are secure and should be allowed, which communications are dangerous and should be rejected, or which communications are potentially insecure and require close monitoring.

***A Secure Architecture Description Language: Secure xADL.*** Secure xADL is a software architecture description language that describes security properties of a software architecture. Secure xADL combines the xADL language with the architectural access control concepts defined in the preceding paragraphs. Figure 13-2 depicts the core syntax of Secure xADL. The central construct is `SecurityPropertyType`, which is a collection of the subject, the principals, the privileges, and the policies of an architectural element. The `SecurityPropertyType` can be attached to component and connector types in xADL. Figure 13-2 illustrates that it is attached to a connector type to make a secure connector type. The `SecurityPropertyType` can also be attached to components and connectors, making them secure components and connectors. Finally, the `SecurityPropertyType` can also be attached to the specifications of subarchitectures and the description of the global software architecture.

An access control policy describes what access control requests should be permitted or denied. The policies for Secure xADL are embedded in the xADL syntax and written with the eXtensible Access Control Markup Language (XACML) (OASIS 2005). XACML is an open standard from OASIS (Organization for the Advancement of Structured Information Standards) to describe access control policies for different types of applications. It is utilized in an environment where a policy enforcement point (PEP) asks a policy decision point (PDP) whether a request, expressed in XACML, should be permitted. The PDP consults

Sony Exhibit 1018 Page 520
Sony v. Quantum IPR2023-00954

Figure 13-2. *Secure xADL schema.*

```
<complexType name="SecurityPropertyType">
  <sequence>
    <element name="subject"
          type="Subject"/>
    <element name="principals"
          type="Principals"/>
    <element name="privileges"
          type="Privileges"/>
    <element name="policies"
          type="Policies"/>
  </sequence>
</complexType>

<complexType name="SecureConnectorType">
  <complexContent>
    <extension base="ConnectorType">
      <sequence>
        <element mame="security"
          type="SecurityPropertyType"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="SecureSignature">
  <complexContent>
    <extension base="Signature">
      <sequence>
        <element name="safeguards"
          type="Safeguards"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<!-- similar constructs for component, structure, and instance -->
```

its policy, also expressed in XACML and makes a decision. The decision can be one of the following: `permit`, `deny`, `not applicable` (when the PDP cannot find a policy that clearly gives a permit or a deny answer), and `indeterminate` (when the PDP encounters other errors).

The core XACML is based on the classic discretionary access control model where a request for performing an action on an object by a subject is permitted or denied. In XACML, an object is termed a resource. Syntactically, a PDP has a `PolicySet`, which consists of a set of `Policy`. Each `Policy` in turn consists of a set of `Rule`. Each `Rule` decides whether a request from a subject for performing an action on a resource should be permitted or denied. When a PDP receives a request that contains attributes of the requesting subject, action, and resource, it tries to find a *matching* `Rule`, whose attributes match those of the request, from the `Policy` and `PolicySet`, and uses the matching rule to make a decision about permitting or denying access to the resource.

### An Algorithm to Check Architectural Access Control

In xADL, each component and connector has a set of interfaces that represent externally accessible functionalities. An interface can be either an incoming interface, denoting functionality the element provides, or an outgoing interface, denoting functionality that the element requires. Each incoming interface can be protected by a set of safeguards that

specify the permissions that components or connectors must possess before they can access that interface. Each outgoing interface can also possess a set of privileges that is generally the same as those of the owning element, that is, the privileges of the element having that outgoing interface.

The interfaces are connected to form a complete architecture topology. A pair of connected interfaces has one outgoing interface and one incoming interface. Such a connection specifies that the element with the outgoing interface accesses the element at the incoming interface. Each such connection defines an *architectural access*. For example, in the C2 architecture style, a component sends a notification from its bottom interface to a top interface of a connector if the component has sufficient privileges. Architectural access is not limited to direct connections between interfaces. Two components could be connected through a connector. Thus, a meaningful architectural access might involve two components that only indirectly communicate through a connector.

At the architecture level, the concerning decision is whether an architectural access in a software architecture description should be granted or denied. More precisely, *given a software architecture description written in Secure xADL, for a pair of components (A, B), should A be allowed to access B?* Finding the answer to this question can help an architect design secure software from two different perspectives. First, the answer helps the architect decide whether the given architecture allows intended access control. If there is some access that is intended by the architect but is not allowed by the description, the description should be changed to accommodate the access. Second, the answer can help the architect decide whether there are architectural vulnerabilities that introduce undesired access. If some undesired access is allowed, then the architect must modify the architecture and architectural description to eliminate such vulnerabilities.

From an architectural modeling viewpoint, the security-related decisions made by components and connectors might be based on factors, or contexts, other than the decision maker and the protected resource. The four most common types of contexts that can affect access control decisions are the neighboring components and connectors, the type of components and connectors, the subarchitecture containing components and connectors, and the global architecture.

Given knowledge of the executing subjects, an algorithm can be used to decide whether the outgoing interface of an accessing component carries sufficient privileges to satisfy the safeguards of the incoming interface of an accessed component. The accessing component can acquire privileges from multiple sources. The component may itself possess some privileges. It can also get privileges from its type, the containing subarchitecture, and the complete architectural model. Further, privileges can also propagate to the accessing component through connected components and connectors, subject to the privilege propagation capability of the connectors. The accessed components can acquire safeguards from similar sources. One notable difference in acquiring safeguards is that this process does not involve the connected element context, and thus does not go through a propagation process.

The simplest approach to make a decision whether to allow such access is to check whether the accumulated privileges of the accessing element covers the accumulated permissions of the accessed element. However, the accessed element can choose to use a different policy, and the sources of the policy can be from the accessed element, the type of the element, the subarchitecture containing the element, and the complete architecture.

Sony Exhibit 1018 Page 522
Sony v. Quantum IPR2023-00954

Figure 13-3. *Access control check algorithm.*

```
Input: an outgoing interface, Accessing,
    and an incoming interface, Accessed

Output: grant if the Accessing can access
    the Accessed, deny if the Accessing
    cannot access the Accessed

Begin
  if (there is no path between Accessing and Accessed)
    return deny;
  if (Accessing and Accessed are connected directly)
    DirectAccessing = Accessing;
  else
    DirectAccessing = the element nearest to Accessed in the path;
  Get AccumulatedPrivileges for
    DirectAccessing from the owning element, the type, the containing
    sub-architecture, the complete architecture, and the
    connected elements;
  Get AccumulatedSafeguards for Accessed from the owning element, the
    type, the containing sub-architecture, and the complete architecture;
  Get AccumulatedPolicy for Accessed from similar sources;
  if (AccumulatedPolicy exists)
    if (AccumulatedPolicy grants access)
      return grant;
    else
      return deny;
  else
    if (AccumulatedPrivileges contains AccumulatedSafeguards)
      return grant;
    else
      return deny;
End;
```

A simple architectural access control check algorithm is sketched in Figure 13-3. The algorithm first checks whether the accessing interface and the accessed interface are connected in the architecture topology. If not, the algorithm denies the architectural access. However, if they are connected, the algorithm proceeds to find the interface in the path that is nearest to the accessed interface, namely the direct accessing interface. If the accessing interface and the accessed interface are directly connected, this direct accessing interface is the same as the accessing interface. Then, the privileges of the direct accessing interface are accumulated using various contexts. Similarly, the safeguards and policies of the accessed interface are also collected. If a policy is explicitly specified by the architect, then the policy is consulted to decide whether the accumulated privileges are sufficient for the access. If there is no explicit policy, then the access is granted if the accumulated privileges contain the accumulated safeguards as a subset. This simple algorithm assumes a known, fixed assignment of subjects on whose behalf the architecture operates. Changing subjects requires re–analysis. Dynamic contexts require a more sophisticated approach.

The algorithm in Figure 13-3 checks architectural access control for a pair of interfaces. Extending it to the global system architecture can be achieved by enumerating each pair of interfaces and then applying the algorithm to each pair. If the global architecture contains subarchitectures, then a completely flattened architecture graph, where containers' privileges are propagated to the contained elements, is first constructed. Afterward, the algorithm is used to check architectural access control between relevant pairs of interfaces belonging to this architecture graph.

Sony Exhibit 1018 Page 523
Sony v. Quantum IPR2023-00954

We now examine how the models and techniques for architectural access control can be applied to two applications, one based on secure cooperation and Firefox as the other.

### Example: Secure Cooperation

The first example of architectural access control is a simplistic, notional application that requires secure cooperation between its participants. The software architecture of the application is expressed in the C2 architectural style. The application allows two parties to share data with each other, but these two parties do not necessarily fully trust each other; thus, the data shared must be subject to the control of each party.

The two parties participating in this hypothetical application are an insurance company and a hospital. Each can operate independently and display the messages they receive from their own information sources. For example, the insurance company may internally exchange messages about an insured person's policy status and the hospital sends a patient's medical history among its departments. The two parties also need to share some messages so the insurance company can pay for the service the hospital provides to patients. To accomplish this, the hospital sends a message to the insurance company, including the patient's name and the service performed. After verifying the policy, the insurance company sends a message back to the hospital, authorizing paying a certain amount from a certain account. While the two parties need to exchange information, such sharing is limited to certain types of messages. Governing laws, such as the United States' Health Insurance Portability and Accountability Act (HIPAA), might prohibit one party from sending certain information to the other, such that the hospital cannot send a person's full medical report to the insurance company. Moreover, maintaining business competitiveness also requires each party to not disclose unnecessary information.

Figure 13-4 depicts the application architecture that uses a secure connector on each side that securely routes messages between the insurance company and the hospital. When the insurance-to-hospital connector receives a notification message, it inspects the message, and if the message can be delivered to both the company and the hospital, such as a payment authorization message, then the message is forwarded to both sides. Otherwise, the message is only transferred within the insurance company. The hospital-to-insurance connector operates in a similar fashion.

The data sharing can be controlled in a number of ways by setting different policies on the connectors. For instance, each of the connectors can be denied instantiation that will prevent any sharing to occur. Even if both connectors are instantiated, the connections with other components and connectors can still be rejected to prevent message delivery and data sharing. When the connectors are instantiated and properly connected with other elements, each of them can use its own policy on internal message routing to control the messages that can be delivered to its own and the other side.

This architecture also promotes understanding and reuse: Only two secure connectors are used; these connectors perform a single task of secure message routing, and they can be used in other cases by adopting a different policy.

### Example: Firefox

Firefox is an open-source Web browser first released in November 2004. It uses three key platform technologies: XPCOM, a cross-platform component model; JavaScript, the Web development programming language that is also used to develop front-end

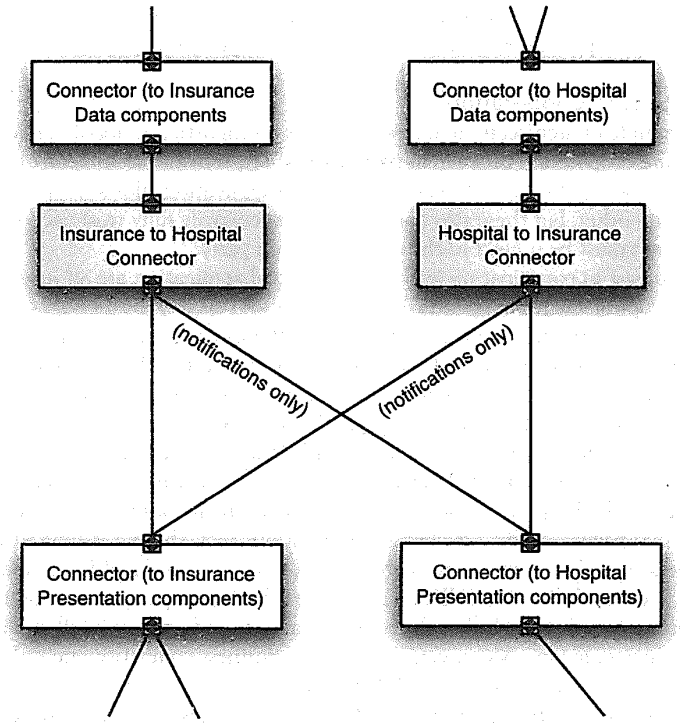



Figure 13-4. *Insurance-hospital interorganization information routing.*

components of Firefox; and XPConnect, the bidirectional connection between native XPCOM components and JavaScript objects.

***Trust Boundary Between Chrome and Content.*** When a user uses the Firefox browser to browse the Web, the visible window contains two areas. The chrome, which consists of decorations of the browser window, such as the menu bar, the status bar, and the dialogs, are controlled by the browser. The browser is trusted to perform arbitrary actions to accomplish the intended task. Borrowing the chrome term that originally refers to the user interface elements, the browser's code is called the *chrome code*. Such code can perform arbitrary actions. Any installed third-party extensions also become a part of the chrome code.

The other area, the content area, is contained within the browser chrome. The content area contains content coming from different sources that are not necessarily trustworthy. Some of this content may contain active JavaScript code. Such *content code* should not be allowed to perform arbitrary actions unconditionally and must be restricted accordingly. Otherwise, such code could abuse privileges to cause damage or harm to the users. This boundary between the chrome code and the content code is the most important trust boundary in Firefox.

Because of the architectural choice of using XPCOM, JavaScript, and XPConnect to develop the Firefox browser and extensions, both chrome code and content code written

in JavaScript can use XPConnect to access interfaces of XPCOM components that interact with the underlying operating system services. The XPCOM components are represented as the global `Components` collection in JavaScript.

XPConnect, as the connector between the possibly untrustworthy accessing code and the accessed XPCOM components, should protect the XPCOM interfaces and decide whether the access to those interfaces should be permitted.

***Trust Boundary Between Contents from Different Origins.*** Another trust boundary is between contents having different origins. The origin of content is determined by the protocol, the host name, and the port used to retrieve the content. Contents differing in either the protocol, the host name, or the port would be considered to have different origins. Users may browse many different sites, and any page can load content from different origins. The content coming from one source should only be able to read or write content originating from the same source. This is called the *same-origin* policy. Otherwise, a malicious page from one source could use this cross-domain access to retrieve or modify sensitive information from another origin, such as the password that the user uses for authentication with the other origin. This is an architectural access control process where interfaces of a content component from one origin should not be inappropriately accessed by another content component from another origin.

***Principals.*** Since the JavaScript language does not specify how security should be handled, the Firefox JavaScript implementation defines a principal-based security infrastructure to support enforcing the trust boundaries. There are two types of principals. When a script is accessing an object, the executing script has a *subject principal* and the object being accessed has an *object principal*.

Firefox uses principals to identify code and content coming from different origins. Each unique origin is represented by a unique principal. The principal in Firefox corresponds to the Subject construct in Secure xADL. Such Subjects are used to regulate architectural access control.

***XPConnect: Secure Connector.*** The security manager within the XPConnect architectural connector coordinates critical architectural operations. It regulates the access by scripts running as one principal to objects owned by another principal (if the subject principal is not the system principal, then both principals should be the same for the access to be allowed), it decides whether a native service can be created, obtained, and wrapped (one type of architectural instantiation operation), and it also arbitrates whether a URL can be loaded into a window (another type of architectural instantiation operation).

Figure 13-5 depicts the Firefox component security architecture. Interfaces of the native XPCOM components executing with the chrome role are accessible from other chrome components but should be protected from other content components. The XPConnect connector maintains this boundary between content code and chrome code. The content components from one origin, including the containing window or frame and the DOM nodes contained within them, form a subarchitecture. Their interfaces can be manipulated by chrome components, but should be protected from content components from other origins. The XPConnect connector maintains this boundary of same origin and helps achieve the needed protection.

Sony Exhibit 1018 Page 526
Sony v. Quantum IPR2023-00954

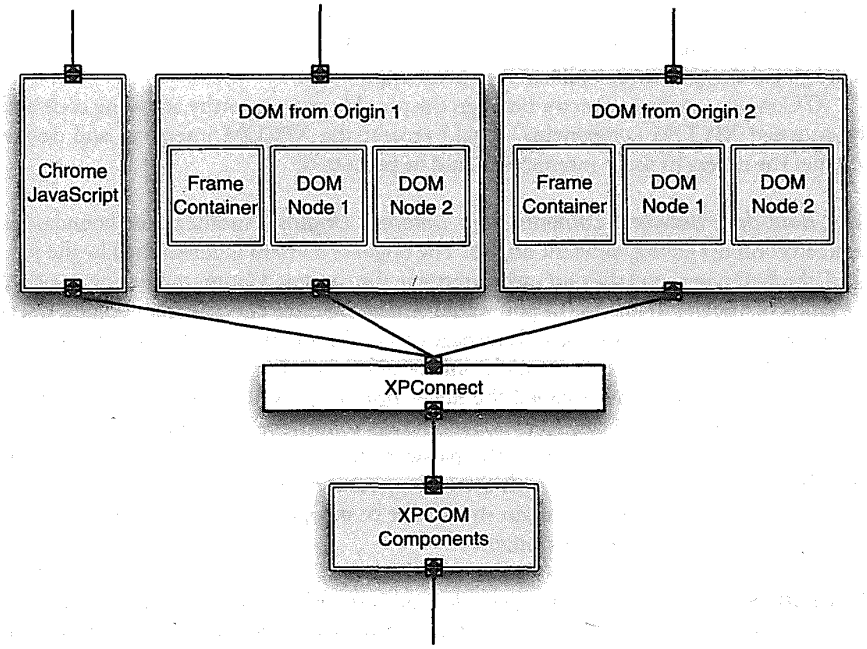



Figure 13-5. *Firefox component security architecture.*

To briefly summarize this section, we first defined the access control models needed to enforce security and illustrated how they can be applied at the architecture level through the use of concepts such as *subject*, *principal*, *resource*, *privilege*, *safeguard*, and *policy*. We next demonstrated how these concepts can be incorporated into an architecture description using the Secure xADL architecture description language as an example. The resulting architecture description can be checked to verify that the architectural accesses occur only as intended. The concepts, languages and algorithms allow an architect to evaluate the security properties of alternative architectures and choose designs that suit secure requirements. We also discussed two example applications to illustrate these benefits.

In the next section, we discuss another security notion—trust—and show how an architecture approach can be successfully used to integrate trust management within environments where participants are decentralized and make independent trust decisions in the absence of a centralized authority.

## 13.4 TRUST MANAGEMENT

Trust management concerns how entities establish and maintain trust relationships with each other. Trust management plays a significant role in decentralized applications (such as the peer-to-peer architectures discussed in Chapter 11) where entities do not have complete information about each other and thus must make local decisions autonomously. Entities must account for the possibility that other entities may be malicious and may

Sony Exhibit 1018 Page 527
Sony v. Quantum IPR2023-00954

indulge in attacks with an intention to subvert the system. In the absence of a centralized authority that can screen the entry of entities in the system, manage and coordinate the peers, and implement suitable trust mechanisms, it becomes the responsibility of each decentralized entity to adopt suitable measures to safeguard itself. Trust relationships, in such applications, help entities gauge the trustworthiness of other entities and thus make informed decisions to protect themselves from potential attacks.

It is therefore critical to choose an appropriate trust management scheme for a decentralized application. This by itself is not enough, however. Consider the analogy of a house, access to which is restricted by a lock on the front door. The owner may be worried that the lock may be easily picked by thieves and so may explore new kinds of locks that are harder to break; however, the owner may not realize that the windows are unsecured and can be easily penetrated. Thus, it is important to focus on the lock as well as the windows to ensure the integrity of the house. Similarly, it is important to focus not only on a reliable trust management scheme but also on fortifying each entity in a decentralized application. This is possible through a software architecture approach that guides the integration of a suitable trust model within the structure of each entity and includes additional security technologies to address the pervasive concerns of security and trust.

Our focus in this section is primarily on incorporating reputation-based trust management systems within the architecture of decentralized entities. Reputation-based trust management systems are those that use an entity's past reputation to determine its trustworthiness. However, before delving deeper into reputation-based systems, we present a brief introduction to the concepts of trust and reputation.

### 13.4.1 Trust

The concept of trust is not new to humans and it is not limited only to electronic entities. Trust is an integral part of our social existence: Our interactions in society are influenced by the perceived trustworthiness of other entities. Thus, in addition to computer scientists, researchers from other fields such as sociology, history, economics, and philosophy have devoted significant attention to the issue of trust (Marsh 1994). Given the fact that trust is a multidisciplinary concept, several definitions of trust exist in the literature. However, since the discussion here is in the context of software development, we adopt a definition of trust coined by Diego Gambetta that has been widely used by computer scientists. He defines trust as

> ... a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both *before* he can monitor such action (or independently of his capacity ever to be able to monitor it) *and* in a context in which it affects his own action. (Gambetta 2000)

This definition notes that trust is subjective and depends upon the view of the individual; the perception of trustworthiness may vary from person to person. Further, trust can be multidimensional and depends upon the context in which trust is being evaluated. For example, A may trust B completely to repair electronic devices but may not trust B to repair cars. The concept of context is thus critical since it can influence the nature of trust relationships significantly.

Sony Exhibit 1018 Page 528
Sony v. Quantum IPR2023-00954

Gambetta also introduced the concept of using values for trust. These values may express trust in several different ways. For example, trust may be expressed as a set of continuous real values, binary values, or a set of discrete values. The representation and expression of trust relationships depends upon the application requirements. For example, binary values for trust may be used in an application that needs only to establish whether an entity can be trusted. If instead the application requires entities to compare trustworthiness of several entities, a richer expression of trust values is required, motivating the need for continuous trust values.

Trust is conditionally transitive. This means that if an entity A trusts entity B and entity B trusts entity C, it may not necessarily follow that entity A can trust entity C. There are a number of parameters that influence whether entity A can trust entity C. For example, entity A may trust entity C only if certain possibly application-specific conditions are met, or if the context of trust is the same.

### 13.4.2 Trust Model

A trust model describes the trust relationships between entities. Realizing the immense value of managing trust relationships between entities, a number of trust models have been designed. These models are geared at different objectives and targeted at specific applications and hence embody different definitions of "trust model." For some, the model may mean just a trust algorithm and a way of combining different trust information to compute a single trust value; for others, a trust model may also encompass a trust-specific protocol to gather trust information from other entities. Yet others may want a trust model to also specify how and where trust data is stored.

> ***Definition.*** **A *trust model* describes the trust information that is used to establish trust relationships, how that trust information is obtained, how that trust information is combined to determine trustworthiness, and how that trust information is modified in response to personal and reported experiences.**

This definition of a trust model identifies three important components of a trust model. The first component specifies the nature of trust information used and the protocol used to gather that information. The second component dictates how the gathered information is analyzed to compute a trust value. The third component determines not only how an entity's experiences can be communicated to other entities but also how it can be incorporated back into the trust model.

### 13.4.3 Reputation-Based Systems

Related to trust is the concept of *reputation*. Alfarez Abdul-Rahman and Stephen Hailes (Abdul-Rahman and Hailes 2000) define reputation *as an expectation about an individual's behavior based on information about or observations of its past behavior*. In online communities, where an individual may have little information to determine the trustworthiness of others, reputation information is typically used to determine the extent to which they can be trusted. An individual who is more reputed generally is considered to be more trustworthy.

Reputation may be determined in several ways. For example, a person may rely on his direct experiences, the experiences of other people, or a combination of both to determine the reputation of another person. Trust management systems that use reputation to

determine the trustworthiness of an entity are termed reputation-based systems. There are several applications, such as Amazon.com and eBay, that employ such reputation-based systems.

Reputation-based systems can be either centralized or decentralized. A decentralized reputation-based system is one where every entity directly evaluates other entities, maintains those evaluations locally, and interacts directly with other entities to exchange trust information. A centralized reputation-based system, on the other hand, relies on a single centralized authority to either facilitate evaluations and interactions between entities or to store relevant trust information. Amazon.com and eBay provide a central repository to store reputation information provided by their users while XREP, a trust model for P2P file-sharing applications, is an example of a decentralized reputation-based system. Next, we next take a deeper look at eBay and XREP.

### eBay

eBay is an electronic marketplace where diverse users sell and buy goods. Sellers advertise items and buyers place bids for those items. After an auction ends, the winning bidder pays the seller for the item. Both buyers and sellers rate each other after the completion of a transaction. A positive outcome results in a $+1$ rating and a negative outcome results in a $-1$ rating. These ratings form the reputation of buyers and sellers. This reputation information is stored and maintained by eBay instead of by its users. eBay is, thus, not a purely decentralized reputation system. If eBay's centralized data stores were to become unavailable, eBay users would have no access to the trust information of other users.

eBay allows this trust information to be viewed through feedback profiles. A user can click on the feedback profile of buyers or sellers to view their past interaction histories and trust information. The profile includes the number of total interactions a user has been involved in along with his total trust score. This score, called the feedback score, is computed as follows: A positive rating increases the feedback score by 1, a negative rating decreases the feedback score by 1, and a neutral rating leaves the feedback score unaffected. A user can only affect another user's feedback score by one point per week. For example, if one user were to leave three positive ratings for another user, the feedback score would only increase by 1. Similarly, even if a user were to leave five negative ratings and two positive ratings, the feedback score would only decrease by 1. The profile also lists the user's total number of positive, negative, and neutral ratings. The profile also displays the aggregate of the most recent ratings received in the last one month, six months, and twelve months. A user viewing a profile can also choose to read all the comments written about a particular buyer or seller.

Such a system can be manipulated to defeat its purpose, of course. The case of eBay in 2000 is a classic example of a set of peers engaging in fraudulent actions (Dingledine et al. 2003). A number of peers first engaged in multiple successful eBay auctions (to establish strong trust ratings). Once their trust ratings were sufficiently high to engage in high-value deals, they used their reputations to start auctions for high-priced items, received payment for those items, and then disappeared, leaving the buyers defrauded.

### XREP

XREP, proposed by Ernesto Damiani and colleagues (Damiani et al. 2002), is a trust model for decentralized peer-to-peer (P2P) file-sharing applications. Development of trust models

is an active area of research and XREP is chosen here to serve as an example. P2P file-sharing applications consist of a distributed collection of entities, also called peers, that interact and exchange resources, such as documents and media, directly with other peers in the system. A decentralized P2P file-sharing application, such as one based on Gnutella, is characterized by the absence of a centralized authority that coordinates interactions and resource exchanges among peers. Instead, each peer directly queries its neighboring peers for files and this query subsequently is forwarded to other peers in the system. Each queried peer responds positively if it has the requested resource. Upon receiving these responses, the query originator can choose to download the resource from one of the responding peers.

While such decentralized file-sharing applications offer significant benefits, such as no single point of failure and increased robustness in addition to allowing users at the edge of the network to directly share files with each other, they are also prone to several attacks by malicious peers. This is because such decentralized P2P file-sharing applications are also open, implying that anyone can join and leave the system at any time without any restrictions. Peers with malicious intent may offer tampered files or may even disguise Trojan horses and viruses as legitimate files and make them available for download. In the January 2004 issue of *Wired* magazine, an article by Kim Zetter, "Kazaa Delivers More Than Tunes" (Zetter 2004) mentions a study in January 2004 that reported that 45 percent of 4,778 executable files downloaded through the Kazaa file-sharing application contained malicious code such as viruses and Trojan horses. When unsuspecting users download such files, they may not only harm their own computers but also unknowingly spread the malicious files to other users.

Clearly, there is a need for mechanisms that will help determine the trustworthiness of both peers and the resources offered by them. Decentralized reputation-based trust schemes offer a potential solution to this problem by using reputation to determine the trustworthiness of peers and resources. XREP is an example of such a reputation-based scheme for decentralized file-sharing applications. XREP includes a distributed polling algorithm to allow reputation values to be shared among peers, so that a peer requesting a resource can assess the reliability of both the peer and the resource offered by a peer.

The XREP distributed protocol consists of the following phases: resource searching, resource selection and vote polling, vote evaluation, best servent check, and resource downloading, as illustrated in Figure 13-6. Resource searching is similar to that in Gnutella and involves a servent (that is, a Gnutella peer; *servent* is a neologism formed from *ser*ver and cli*ent*) broadcasting to all its neighbors a *Query* message containing search keywords. When a servent receives a *Query* message, it responds with a *QueryHit* message. In the next phase, upon receiving *QueryHit* messages, the originator selects the best matching resource among all possible resources offered. At this point, the originator polls other peers using a *Poll* message to ask their opinions about the resource or the servent offering the resource. Upon receiving a *Poll* message, each peer may respond by communicating its votes on the resource and servents using a *PollReply* message. These messages help distinguish reliable from unreliable resources and trustworthy from fraudulent servents.

In the third phase, the originator collects a set of votes on the queried resources and their corresponding servents. Then it begins a detailed checking process that includes verification of the authenticity of the *PollReply* messages, guarding against the effect of a group of malicious peers acting in tandem by using cluster computation, and sending

Sony Exhibit 1018 Page 531
Sony v. Quantum IPR2023-00954

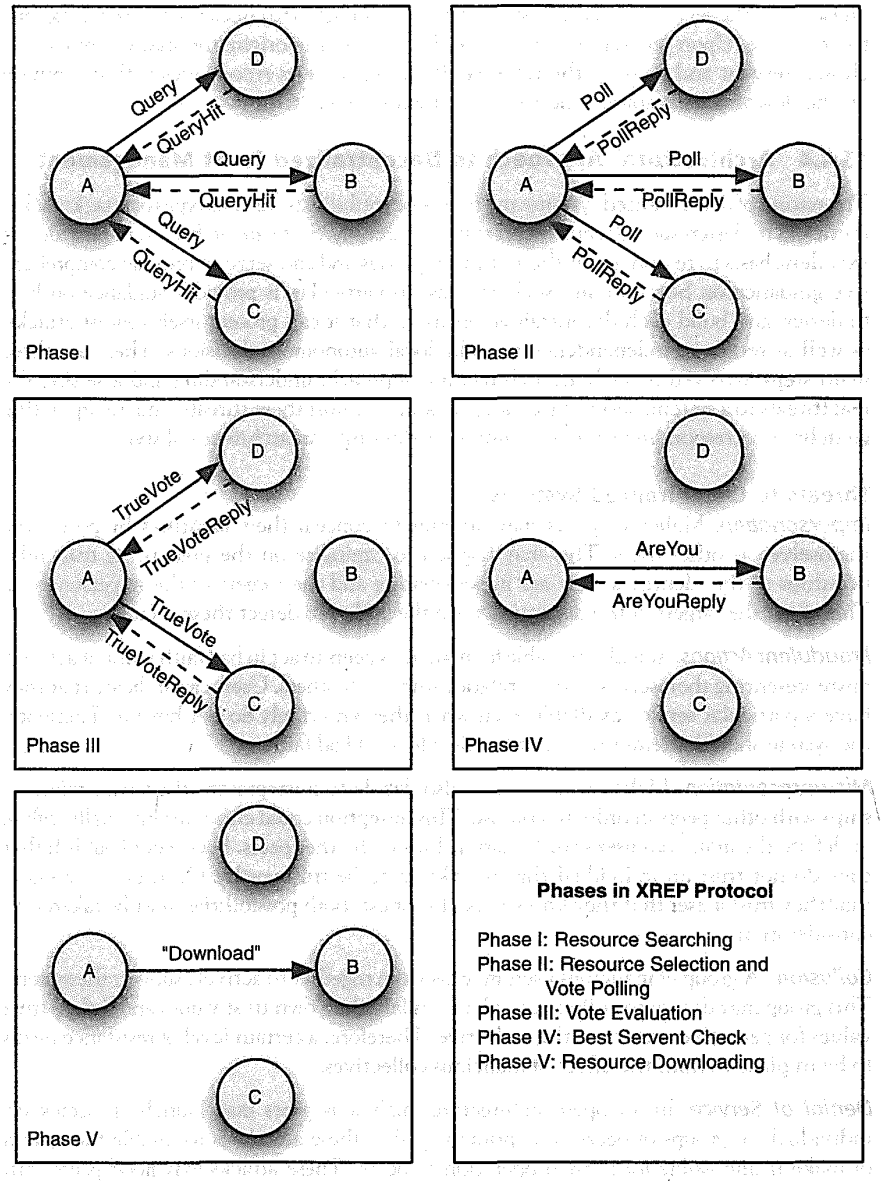



Figure 13-6.
*Phases in XREP.*

*TrueVote* messages to peers that request confirmation on the votes received from them. At the end of this checking process, based on the trust votes received, the peer may decide to download a particular resource. However, since multiple servents may be offering the same resource, the peer still needs to select a reliable servent. This is done in the fourth

phase when the servent with the best reputation is contacted to check the fact that it exports the resource. Upon receiving a reply from the servent, the originator finally contacts the chosen servent and requests the resource. It also updates its repositories with its opinion on the downloaded resource and the servent who offered it.

### 13.4.4 Architectural Approach to Decentralized Trust Management

The nature of decentralized systems and their susceptibility to various types of attacks makes it critical to design such decentralized systems carefully. Software architecture provides an excellent basis to reason about these trust properties and can serve to provide comprehensive guidance on how to build such systems. In particular, it provides guidance on how to design and build each decentralized entity so that it can protect itself against attacks, as well as retain its independence to make local autonomous decisions. There are three main steps involved in such an architectural approach: understanding and assessing the real threats to a system, designing countermeasures against these threats, and incorporating guidelines corresponding to these countermeasures into an architectural style.

#### Threats to Decentralized Systems

***Impersonation.*** Malicious peers may attempt to conceal their identities by portraying themselves as other users. This may happen to capitalize on the preexisting trust relationships of the identities they are impersonating and the targets of the impersonation. Therefore, the targets of the deception need the ability to detect these incidents.

***Fraudulent Actions.*** It is also possible for malicious peers to act in bad faith without actively misrepresenting themselves or their relationships with others. Users can indicate that they have a particular service available even when they knowingly do not have it. Therefore, the system should attempt to minimize the effects of bad faith.

***Misrepresentation.*** Malicious users may also decide to misrepresent their trust relationships with other peers in order to confuse. This deception could either intentionally inflate or deflate the malicious user's trust relationships with other peers. Peers could publish that they do not trust an individual that they know to be trustworthy. Or, they could claim that they trust a user that they know to be dishonest. Both possibilities must be taken into consideration.

***Collusion.*** A group of malicious users may also join together to actively subvert the system. This group may decide to collude in order to inflate their own trust values and deflate trust values for peers that are not in the collective. Therefore, a certain level of resistance needs to be in place to limit the effect of malicious collectives.

***Denial of Service.*** In an open architecture, malicious peers may launch an attack on individuals or groups of peers. The primary goal of these attacks is to disable the system or make it impossible for normal operation to occur. These attacks may flood peers with well-formed or ill-formed messages. In order to compensate, the system requires the ability to contain the effects of denial of service attacks.

***Addition of Unknowns.*** In an open architecture, the cold start situation arises: Upon initialization, a peer does not know anything about anyone else on the system. Without any trust information present, there may not be enough knowledge to form relationships

until a sufficient body of experience is established. Therefore, the ability to bootstrap relationships when no prior relationships exist is essential.

***Deciding Whom to Trust.*** In a large-scale system, certain domain-specific behaviors may indicate the trustworthiness of a user. Trust relationships generally should improve when good behavior is perceived of a particular peer. Similarly, when dishonest behavior is perceived, trust relationships should be downgraded accordingly.

***Out-of-Band Knowledge.*** Out-of-band knowledge occurs when there is data not communicated through normal channels. While trust is assigned based on visible in-band interactions, there may also exist important invisible interactions that have an impact on trust. For example, Alice could indicate in person to Bob the degree to which she trusts Carol. Bob may then want to update his system to adjust for Alice's out-of-band perception of Carol. Therefore, ensuring the consideration of out-of-band trust information is essential.

### Measures to Address Threats

***Use of Authentication.*** To prevent impersonation attacks, it is essential to use some form of authentication so that message senders can be uniquely identified. For instance, entities sign outgoing messages and receiving entities verify those signatures to validate the authenticity of those messages. Signature-based authentication, such as that discussed at the beginning of this chapter, also helps protect against potential repudiation attacks—attacks where an entity may falsely claim that it never sent the message.

### Separation of Internal Beliefs and Externally Reported Information

In a decentralized system, each entity has its own individual goals, which may conflict with those of other entities. It is therefore important to model externally reported information separately from internal beliefs. This separation helps resolve conflicts between externally reported information and internal perceptions. For example, a peer may favor information it has perceived directly and believes to be accurate over information reported by others. A peer may also not want to disclose sensitive data, so it must have the ability to report information that differs from what it actually believes. ("What I've heard is . . .")

***Making Trust Relationships Explicit.*** Without a controlling authority that governs the trust process, peers require information to make decisions whether or not to trust what they perceive. Active collaboration between peers may provide enough knowledge for peers to reach their local decisions. Thus it is important that information about trust relationships be explicit and exchangeable between peers. There is a possibility that exposing trust information may be misused by malicious peers to take advantage of certain peers; however, it should be remembered that exchanged information may not truly reflect the trust perceptions of the entities.

***Comparable Trust.*** Ideally, published trust values should be syntactically and semantically comparable; that is, equivalent representations in one implementation should have the same structure and meaning in another. If the same value has different meanings across implementations, then accurate comparisons across peers cannot be made.

Sony Exhibit 1018 Page 534
Sony v. Quantum IPR2023-00954

### Corresponding Guidelines to Incorporate into an Architectural Style

***Digital Identities.*** Without the ability to associate identity with published information, it is a challenge to develop meaningful trust relationships. Thus, the concept of identities, both physical and digital, is necessary to facilitate meaningful relationships. However, it is important to understand the limitations of digital identities with respect to physical identities.

There may not be a one-to-one mapping between digital and physical identities as one person may utilize multiple digital identities or multiple people may share the same digital identity. Additionally, anonymous users may be present who resist digital identification. Therefore, it is not always possible to tie a digital identity to one physical individual and make accurate evaluations of a person. Instead, a critical criterion of trust relationships in decentralized applications should be the actions performed by digital identities, not by physical identities. The architectural style should therefore consider trust relationships only between digital identities.

***Separation of Internal and External Data.*** Explicit separation of internal and external data supports the separation of internal beliefs from externally reported information within a peer. Therefore, the architectural style should adopt the explicit separation of internal and external data.

***Making Trust Visible.*** Trust information received externally from entities is used within the peer architecture to make local decisions. In order to process this trust information internally across the architecture, trust cannot be localized to only one component. Each component responsible for making local decisions needs the ability to take advantage of this perceived trust. If the perceived trust is not visible, then accurate assessments may not be made. Therefore, the architectural style should require trust relationships to be visible to the components in the peer's architecture as well as be published externally to other peers.

***Expression of Trust.*** There has been no clear consensus in the trust literature as to which trust semantics provide the best fit for applications, therefore it is believed that indiscriminately enforcing a constraint at the architectural level to use a particular trust semantic is inappropriate. While trust values should be semantically comparable, a generic architectural style might impose only the constraint that trust values must at least be syntactically comparable. For example, this can be done by enforcing that trust values be represented numerically.

### Resultant Architectural Style

The principles and constraints identified above can be combined to create an architectural style for decentralized trust management. In addition to these constraints, based on the common elements of trust models, four functional units of a decentralized entity are first identified. These are Communication, Information, Trust, and Application. The Communication unit handles interaction with other entities, the Information unit is responsible for persistently storing trust and application-specific information, the Trust unit is responsible for computing trustworthiness and guides trust-related decisions, and the Application unit includes application-specific functionality and is responsible for enabling local decision making. The Communication unit does not depend upon any other units while the Information unit depends upon information received from other entities and thus depends upon interaction with them. The Trust unit depends upon the Communication and Information units and the Application unit builds upon all the other three units.

Sony Exhibit 1018 Page 535
Sony v. Quantum IPR2023-00954

Given this interplay between the four units, adopting a layered architectural style enables a natural structuring of these units according to their interdependencies and also offers several benefits such as reusability of components. Since decentralized entities are autonomous, they have the privilege of refusing to respond to requests of other entities. As a result, decentralized applications typically employ asynchronous event-based communication protocols. In order to reflect this communication paradigm within the internal architecture of an entity, an event-based architectural style for the architecture of an entity is natural. Moreover, event-based architectural styles have been known to successfully facilitate loose coupling among components. This can, for instance, allow for the replacement of trust models and protocol handlers in the architecture.

C2 is one such event-based layered architectural style. As discussed in Chapter 4, C2 includes specific visibility rules—components belonging to a layer are only aware of components in layers above them and are unaware of components below them. C2 thus naturally fits in with the constraints of a trust–centric architectural style. Further, C2 also has existing tool support that can be leveraged by an architectural style based on C2. Therefore, the PACE architectural style, (Practical Architectural Style for Composing Egocentric applications) described next, extends the C2 style.

### PACE Architectural Style

The PACE architectural style includes all the above-described guidelines and constraints and provides guidance on the components that must be included within the architecture of an entity and how they should interact with each other. The style is described here to illustrate one way of combining insights from the preceding discussion into a coherent architecture. Corresponding to the four functional units, PACE divides the architecture of a decentralized entity into four layers: Communication, Information, Trust, and Application. Each of these layers, along with their components, is illustrated in Figure 13-7.

The Communication layer is responsible for handling communication with other peers in the system. It consists of several components designed to support various standard communication protocols, the Communications Manager, and the Signature Manager. The communication protocol components are responsible for translating internal events to external communications. The Communications Manager instantiates the protocol components while the Signature Manager signs requests and verifies notifications.

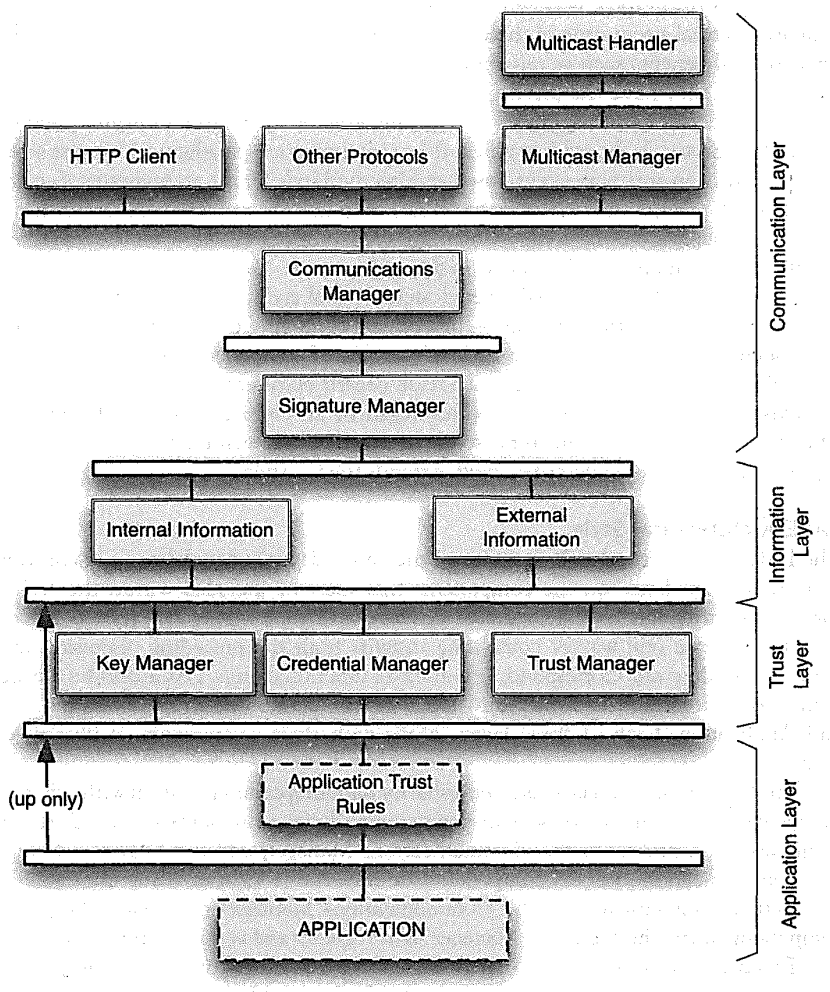
To separate the internal trust beliefs of a peer from those received from other peers, the Information layer consists of two components: the Internal Information component that stores self-originating messages and the External Information component, which stores messages received from others.

The Trust layer incorporates the components that enable trust management. This layer consists of the Key Manager, which generates the local PKI keypair; the Credential Manager, which manages the credentials of other peers; and the Trust Manager, which computes trust values for messages received from other peers.

The Application layer encapsulates all application-specific components. The Application Trust Rules component encapsulates the chosen rules for assigning trust values based on application-specific semantic meanings of messages, and supports different dimensions of trust relationships. The Application subarchitecture represents the local behavior of a peer. While components in the other layers can be reused across different applications, components in the Application layer are application-dependent and hence not reusable

Sony Exhibit 1018 Page 536
Sony v. Quantum IPR2023-00954

Figure 13-7. *PACE components.*




across domains. The application developer is thus expected to implement components for this layer depending upon the application's needs; its internal architecture may be in an entirely different style. All external communication must go through the PACE stack, however.

### PACE-Induced Benefits

PACE's guiding principles induce properties that act as countermeasures against threats to a decentralized system. Some of these properties are effected by the PACE architectural style and canonical implementations of its standard components; others are application specific and so involve the Application layer. We now take a look at some of the common threats

and the way PACE helps address them. It should be noted that it is not mandatory for all peers participating in a decentralized system to be built in the PACE architectural style in order to function and interact with PACE-based peers. However, those peers cannot avail themselves of the benefits of the PACE style.

***Impersonation.*** Impersonation refers to the threat caused by a malicious peer posing as another in order to misuse the peer's privileges or reputation. PACE addresses this threat through the use of digital signatures and message authentication. All external communication in the PACE architecture is constrained to the Communication layer, thus it offers a single point where impersonation can be detected. A malicious peer that tries to impersonate a user without the correct private key or does not digitally sign the message can be detected by verifying signatures. Additionally, if a private key has been compromised, a revocation for that key can be transmitted. PACE components can then refuse to assign trust values to revoked public keys.

***Fraudulent Actions.*** Malicious peers may engage in fraudulent behavior including advertising false resources or services and not fulfilling commitments. Since PACE is designed for open, decentralized architectures, there is little that may be done to prevent the entry of malicious peers. However, malicious actions may be detected by the user or through the Application layer. Explicit warnings can then be issued concerning those malicious peers, which may help others in their evaluations of these peers.

***Misrepresenting Trust.*** A malicious peer may misrepresent its trust with another in order to positively or negatively influence opinion of a specific peer. Since PACE facilitates explicit communication of comparable trust values, a peer can incorporate trust relationships of others. By using a transitive trust model in the Trust Manager, if Alice publishes that she distrusts Bob, then Carol can use that information to determine if she should trust Bob's published trust relationships.

***Collusion.*** Collusion refers to the threat caused by a group of malicious peers that work in concert to actively subvert the system. It is thus of greater concern than a single peer misrepresenting trust. It has been proven that explicitly signed communication between peers can overcome a malicious collective in a distributed setting. Adapting and combining these results with efficient schemes to identify noncooperative groups in a decentralized setting [such as in NICE (Lee, Sherwood, and Bhattacharjee 2003)] with PACE's ability to detect impersonation allows collusion to be addressed.

***Denial of Service.*** Malicious peers may also launch attacks against peers by flooding them with well-formed or ill-formed messages. The separation of the Communication layer allows isolation and response to the effects of such denial of service attacks. Incorrectly formed messages can be disposed of by the protocol handlers. The Communications Manager can also compensate for well-formed message floods by introducing rate limiting or interacting with neighboring firewalls to prevent further flooding.

***Addition of Unknowns.*** When the system is first initialized, there can be a cold-start problem because there are no existing trust relationships. Even though a peer may not have previously interacted with another peer or a message may be known to be forged, PACE's Application layer can still receive these events. Without enough information to make an evaluation, the message will not be assigned a trust value by the Trust Manager.

Sony Exhibit 1018 Page 538
Sony v. Quantum IPR2023-00954

However, the user can still make the final decision to trust the contents of the message based on out-of-band knowledge that is not captured explicitly.

***Deciding Whom To Trust.*** In a large-scale system, certain domain-specific behaviors may indicate the user's trustworthiness. Trust relationships should generally improve when good behavior is perceived of a particular peer and vice versa. In PACE, the application trust rules component allows for automated identification of application-dependent patterns. The detection of good or bad behavior by this component can cause the trust level of the corresponding peer to be increased or decreased respectively along a particular trust dimension.

***Out-of-Band Knowledge.*** It is essential to ensure that out-of-band information is also considered in establishing trust relationships. While PACE confines all electronic communication to the Communication layer, out-of-band trust information can originate as requests from the user through the Application layer.

### Building a PACE-Based Trust-Enabled Decentralized File-Sharing Application

In this section, we present a walk-through of how PACE can be used to design and construct applications. Specifically, we explore how the PACE architectural style can be used to guide the construction of a trust-enabled decentralized file-sharing application. Since the XREP trust model for file-sharing applications was presented earlier, it will be used as the candidate trust model for integration within the PACE style.

The first step in designing an appropriate architecture for each file-sharing entity is to identify the components in the four layers. Since the PACE architectural style already specifies components for the Communication, Information, and Trust layers, the main task here is to identify the components of the Application layer. For the file-sharing application, the Application layer can be decomposed, for example, into eight different components organized into three sublayers as shown in Figure 13-8.

The top sublayer contains only the Application Trust Rules component while the bottom sublayer comprises the User Interface. The middle sublayer consists of the components: Library, Search Manager, Poll Manager, File Exchanger, Evaluator, and Preferences. The Library component maintains the list of files that have been downloaded and that can be shared with other peers; the files themselves are persistently stored in the Internal Information. The Search Manager component is responsible for issuing *Query*, *Poll*, and *TrueVote* messages and displaying received responses to those messages through the user interface.

The Poll Manager component responds to *Poll* messages by sending *PollReply* messages. The File Exchanger component is responsible for uploading and downloading files, displaying uploaded and downloaded files to the user interface, saving downloaded files to the Internal Information storage, and deleting files from the Internal Information. The Evaluator component is responsible for checking the authenticity of *PollReply* messages and analyzing peer votes received about resources. The Preferences component manages login information for the user and enables the user to specify preferences including whether to automatically connect to the P2P network, the number of hops, the number of permissible uploads and downloads, and the destination for the library folder.

Once the components of the Application layer are identified, it is important to determine the interactions between the components in order to identify the relevant request and

Sony Exhibit 1018 Page 539
Sony v. Quantum IPR2023-00954

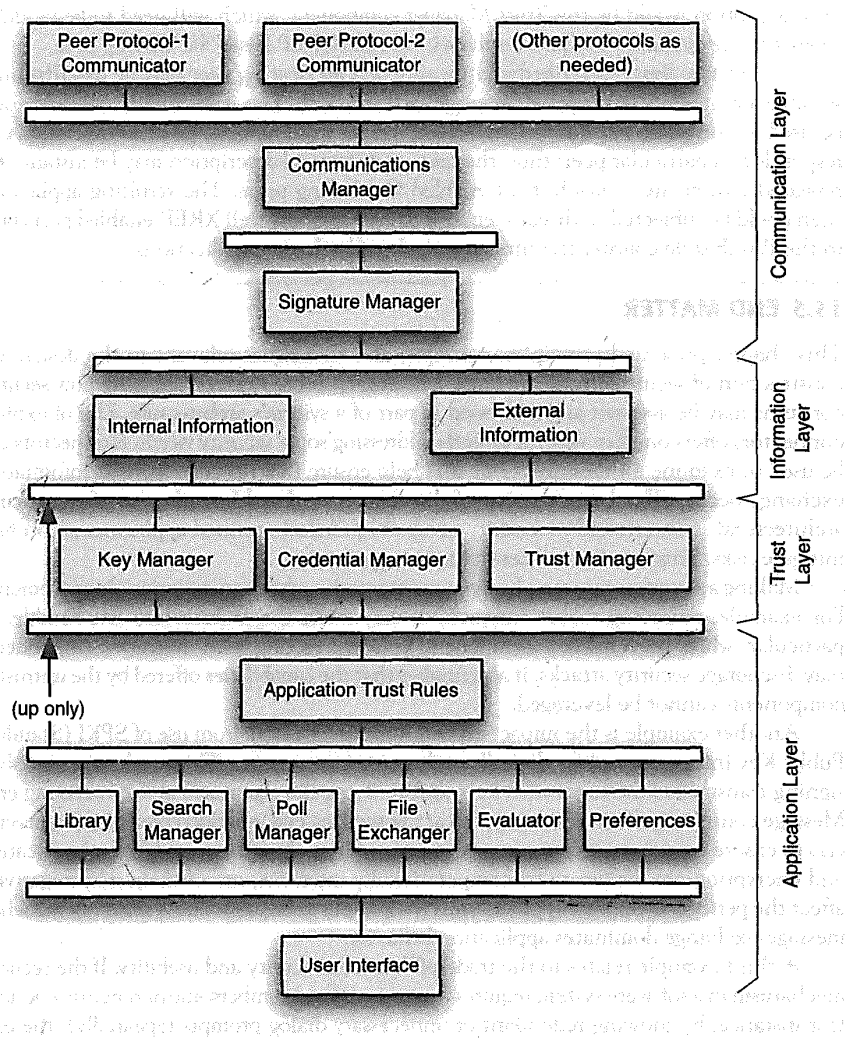



Figure 13-8. *XREP-based architecture of file-sharing peer.*

notification messages that traverse the architecture. This includes modeling the different kinds of trust messages exchanged between peers as dictated by the XREP trust model so the relevant components can appropriately react to them.

In the next step, components belonging to each layer must be appropriately implemented. If existing implementations for any of these components exist, the application developer can choose to reuse them as long as the PACE style is not violated. Since PACE prototypes in other application domains already exist, it is possible to reuse all components from the Communication, Information, and Trust layers without any modifications. The

Sony Exhibit 1018 Page 540
Sony v. Quantum IPR2023-00954

only exception would be the Trust Manager component, which will need to be modified to enable the evaluation of poll results within each XREP-based file-sharing peer.

Finally, the above-described architecture of a file-sharing peer may be described in a suitable architecture description language such as xADL. The ArchStudio tool suite could be used to both describe the architecture as well as instantiate it. Each instantiation corresponds to a particular peer; thus, the same architectural description may be instantiated repeatedly to create multiple trust-enabled file-sharing peers. The resulting application then could be subjected to threat scenarios to evaluate how well XREP-enabled peers built in the PACE style counter the threats typical to file-sharing applications.

## 13.5 END MATTER

This chapter presented principles, concepts, and techniques relevant to the design and construction of secure software systems. The emphasis has been on showing how security concerns may be assessed and addressed as part of a system's architecture. Use of explicit connectors offers one key way for directly addressing some security needs. Connectors can be used to examine all message traffic and help ensure that no unauthorized information exchange occurs. The latter portion of the chapter explored how the use of a particular architectural style, attuned to security needs in open decentralized applications, can help mitigate risks. Trust models are central to this approach.

Making a system secure may, however, compromise other non-functional requirements. For example, enforcing secure interactions may make the architecture less flexible. In particular, while preventing a component from interacting with unknown components may discourage security attacks, it also implies that the capabilities offered by the untrusted components cannot be leveraged.

Another example is the impact on performance resulting from use of SPKI (Standard Public Key Infrastructure) for digitally authenticating messages. This mechanism involves signing transmitted messages with keys and verifying the signatures at the receiving end. Message content itself may be encrypted at the sending end and decrypted at the receiving end to ensure confidentiality of message content. Since algorithms used for authentication and encryption are known to be computationally intensive, use of them may negatively affect the performance of an application. This is likely to be particularly troublesome when message-exchange dominates application behavior.

A third example relates to the trade-off between security and usability. If the security mechanism in a software system requires a user to perform cumbersome or repetitive actions (for instance, by showing redundant or unnecessary dialog prompts repeatedly), the user may choose to completely turn off the security mechanism.

These examples illustrate the importance of properly considering such trade-offs. It is critical for software architects and application developers to apply careful analysis while designing and constructing secure software systems.

**The Business Case**

In a television advertisement for a motor oil additive, auto racing legend Andy Granatelli said, "You can pay me now, or you'll pay me later," referring to the benefits of regular engine maintenance. The implication was that failure to tend to the preventive needs of an engine in a timely manner could lead to expensive, if not catastrophic, failure later on. So it is with software security.

Sony Exhibit 1018 Page 541
Sony v. Quantum IPR2023-00954

As with other software qualities, it is far more cost effective, and technically easier, to plan for and design in security from a project's outset, rather than waiting until a security breach either lands a company unfavorably in the public eye or causes a significant financial loss. Moreover, an architecture-based approach focuses on a comprehensive analysis and solution strategy, not a "plug this leak and wait for the next disaster" reactive response.

The serendipity that accompanies architectural design for security and trust is that the discipline of analyzing a design and introducing security-focused separations and mechanisms at the outset can yield a clean design able to accommodate other, nonsecurity focused, demands for change that arise over time.

Trust management for open, decentralized systems will continue to rise in importance as Web services and other forms of open commerce increase in adoption. As the main text of this chapter has shown, the techniques available have numerous limitations. These limitations are no different from those that have always existed in business, long before the advent of electronic commerce. Should you trust another vendor? Will your invoice be paid? How do you know? The legal remedies that society provides, the out-of-band communication people depend upon—these can apply in e-commerce as well. The increased difficulty, and hence the need for trust management techniques such as those described here, arises from the speed with which transactions—both legitimate and fraudulent—can take place.

Chapter 12 began exploration of architecturally-based techniques for achieving a variety of non-functional properties. This chapter continued that theme, focusing on the specific NFPs of security and trust. Chapter 13 completes the theme, by focusing in-depth on the NFP of adaptability.

## 13.6 REVIEW QUESTIONS

1. What is security? What are the properties that a software system must exhibit in order to be called secure?
2. List and briefly describe each design principle explained in this chapter.
3. What challenges to security are introduced by decentralized applications? What is trust management?

## 13.7 EXERCISES

1. Identify and describe any trade-offs between the design principles explained in the chapter.
2. What are the security benefits and the security risks associated with the following architectural styles from Chapter 4?
   a. Pipe-and-filter
   b. Blackboard
   c. Event-based
   d. C2

   In your answer, consider how malicious actions in one component can or cannot affect the actions of other components.
3. Choose any software application, such as Firefox, and use UML sequence diagrams to show how the security mechanisms within the application operate.
4. Describe at least two applications, other than online auctions and file sharing, where trust or reputation management may prove useful, and explain the rationale for your choices.
5. Evaluate whether the PACE architectural style could be used to build the participants in the above applications.

Sony Exhibit 1018 Page 542
Sony v. Quantum IPR2023-00954

6. Study the architecture of Internet Explorer 7 and evaluate whether IE 7 has any architectural security deficiencies. If so, how can these vulnerabilities be addressed? Use xACML and Secure xADL to propose a more secure architecture for IE 7.

## 13.8 FURTHER READING

An excellent introduction to the field of computer security is given by Bruce Schneier in (Schneier 2000). This highly readable book explores the interplay of technical and nontechnical issues in achieving system security. Matt Bishop's text (Bishop 2003) provides details of the technologies involved. Pfleeger and Pfleeger's book (Pfleeger and Pfleeger 2003) is another comprehensive reference.

Research frontiers for software engineering for security are presented in Devanbu and Stubblebine's roadmap paper (Devanbu and Stubblebine 2000) from the 2000 International Conference on Software Engineering.

Further details on the PACE architectural style can be found in (Suryanarayana et al. 2006; Suryanarayana, Erenkrantz, and Taylor 2005). Further details on the connectorcentric approach to security are available in (Ren et al. 2005; Ren and Taylor 2005).

More recently, new results have been presented at the International Workshop on Software Engineering for Secure Systems, beginning with the 2005 workshop.

Sony Exhibit 1018 Page 543
Sony v. Quantum IPR2023-00954

# 14

# Architectural Adaptation

Few applications are designed, built, used, and ultimately discarded without undergoing change. Change is endemic to software: Both the perceived and actual malleability of the medium, coupled with the ease of altering code, induces everyone associated with an application to initiate changes. Users change their minds about what they want and hence what they require of an application. Designers seek to improve their designs with respect to performance, or appearance, or some other property. The application's usage environment can change. Whatever the reason, software developers are faced with the challenges of coping with the need to modify an application. We group all of these types of changes under the term *adaptation*: modification of a software system to satisfy new requirements and changing circumstances.

Adaptability was introduced in Chapter 12 as one of several non-functional properties for which architects must frequently design. Several basic architectural techniques supporting adaptation were presented. Just as Chapter 13 was devoted to in-depth exploration of the non-functional properties of security and trust, so this chapter delves further into adaptability. Nonetheless, this chapter does not aspire to present techniques for dealing with all possible types of adaptation at all levels of granularity and abstraction—to do so would require a book of its own. Consistent with the focus of the book, our discussion is centered on facilitating change in the context of software architectures: We focus on adaptation to a system's principal design decisions—its architecture—and on changes that proceed from a strong architectural foundation. We consider processes for effecting change that are conceptualized in terms of the system's architecture, and show the value and role of explicit architectural models in supporting and effecting adaptation. We also consider one of the more difficult types of change: modifying applications on-the-fly, showing how this can be achieved using an architecture- and connector-centric approach.



Sony Exhibit 1018 Page 544
Sony v. Quantum IPR2023-00954

The goals of the chapter are to:

- Characterize adaptation, showing what changes, why, and who the players are.
- Characterize the central role software architecture plays in system adaptation.
- Present techniques for effectively supporting adaptation, based on an architecture-centric perspective.

This chapter reveals the particular power of connectors in supporting adaptation, especially in those situations where the connectors remain first-class entities in the implementation and where communication is event-based. Highly dynamic applications are possible in such situations and offer the opportunity for creating novel, flexible, highly adaptable applications.

**Outline of Chapter 14**

14 Architectural Adaptation
14.1 Concepts of Architecture-centric Adaptation
14.1.1 Sources and Motivations for Change
14.1.2 Shearing Layers
14.1.3 Structural Elements Subject to Change
14.1.4 Change Agents and Context
14.1.5 Architecture: The Central Abstraction
14.2 A Conceptual Framework for Architectural Adaptation
14.3 Techniques for Supporting Architecture-centric Change
14.3.1 Basic Techniques Corresponding to Activities of the Conceptual Framework
14.3.2 Architectures/Styles that Support Adaptation
14.3.3 The Special Problems of On-the-Fly and Autonomous Adaptation
14.4 End Matter
14.5 Review Questions
14.6 Exercises
14.7 Further Reading

## 14.1 CONCEPTS OF ARCHITECTURE-CENTRIC ADAPTATION

### 14.1.1 Sources and Motivations for Change

Several motivations for change are familiar to developers. Perhaps the most common is *corrective change*, where the application is adapted to conform to the existing requirements. Put in other words, corrective change means bringing the descriptive architecture into conformance with the prescriptive architecture. Put most simply, bug fixing. Whether bug fixing is easy or highly disruptive, however, may depend on the root cause of an error: Did the observed problem with the application stem from a small, localized error in the code, or result from a profound mistake made at the highest level of system structure? Understanding the root cause is essential for determining how to effect the fix; attempting to patch a deep problem with localized change usually is not effective.

Sony Exhibit 1018 Page 545
Sony v. Quantum IPR2023-00954

A second common motivation for change is *modification to the functional requirements* for a system—new features are needed, existing ones modified, perhaps some must be removed. Rather than thinking that the existence of such needs for change reflect an inadequate system planning or analysis process, functional change requests often proceed from success with the existing application. Once applications are in use they inspire users to think of their task in new ways, leading to new ideas for what might be accomplished. Applications in use may change the usage context, or the usage context may change for extrinsic reasons. Whatever the source, feature change requests typically represent changes to the application's architecture.

A third motivation for change is the requirement to satisfy *new or changed non-functional system properties*. Such properties include security, performance, scale, and so on, as discussed in Chapters 12 and 13. New non-functional properties may induce profound changes to a system's architecture. One example is the World Wide Web. Prior to 1994 the Web's governing protocol was HTTP/1.0, a point-to-point protocol with no provision for proxies or caching. The change to HTTP/1.1 was accomplished by the adoption of a new architectural style for the Web (viz., REST), enabling the Web to scale from thousands of servers and tens of thousands of clients, to many millions of each. Similarly, as security attacks on Microsoft's IIS Web server increased, a significant redesign was required to meet the new threats. Another non-functional property that may cause significant architectural change is "better organization"—restructuring an application in *anticipation* of future change requests. For instance, a new middleware platform (that is, connector technology) may be put in place as a proactive step toward supporting potential future modifications.

A fourth common motivation for system adaptation is the need to conform to a *changed operating environment*. In this situation, neither the application's functions, nor its non-functional properties are altered, but the application's structure must be adapted to run, for example, on new hardware or to accommodate changes in interfaces to external systems.

More obviously germane to the software architect are motivations for change that *arise from product-line forces*. Strictly speaking, such changes can be classified under one of the preceding types of change, but such classification would obscure the insights.

The first of these product-line motivations is the desire to *create a new variant* within a product line. The notion here is that the architects have previously identified a branch point within a product-line architecture and at that branch point a new variant is identified. For example, in consumer electronics, a branch point within the design for integrated television systems is the interface to a high-capacity storage/replay subsystem—a hard disk, a DVD recorder/player, or a tape system perhaps. If it becomes desirable to interface with a different technology, perhaps Blu-ray, a new variant at that branch point is created. In our long-running example of the Lunar Lander video game, a new variant might be based upon the desire to utilize a new wireless joy-stick for controlling the lander or to incorporate a new three-dimensional graphics engine having improved performance.

Another of the product-line motivations is *creation of a new branch point*, thereby identifying a new opportunity for product marketplace segmentation. For instance, an electronics manufacturer may decide to adapt the design of its television products to enable them to work with home security systems, perhaps by using picture-in-picture technology to allow continuous display of surveillance camera data while watching broadcast TV. The existence of many types of external security systems implies many possible variants

Sony Exhibit 1018 Page 546
Sony v. Quantum IPR2023-00954

within this branch point; other variants are implied by the many ways of integrating surveillance information into the television display environment. New branch points in the Lunar Lander design might focus on multiplayer environments or on extensibility to allow missions to other solar system objects.

Yet another product-line motivation comes from the desire to *merge product (sub-)lines*, thereby rationalizing their architectures. A successful effort of this type will yield a common product-line architecture without requiring the complete creation of a new architecture. The goal, ostensibly, is to preserve as much as possible from the contributing architectures.

One type of adaptation that will be discussed later in this chapter is on-the-fly adaptation. The difficulty of supporting such change is typically greater than for non-real time adaptation, so a few words are needed here to prompt its consideration. A simple but perhaps extreme example motivating this type of change comes from planetary exploration. The Cassini/Huygens spacecraft were launched from Cape Canaveral in October 1997; they did not reach their intended destination, Saturn, until July 2004. During transit to Saturn the onboard software systems were upgraded several times. As the Cassini Web site (NASA Jet Propulsion Laboratory) puts it, "This development phase will never really be complete, since new command sequences are continually being created throughout Cassini's operation. New flight software will also be developed and loaded aboard the spacecraft periodically, to run in the Command Data Subsystem, Attitude and Articulation Control Subsystem, as well as in the physical science instruments onboard." As a *system* it is clear that Cassini could not be stopped and reloaded with the new software; rather the updates had to take place while the system was in continuous operation.

**Spacecraft Software Updates**

The updating of spacecraft software while in space light is an extreme case of adaptation on the fly. The needs for updating reflect many sources, from new directions or priorities for a mission based upon recently discovered phenomena to bug fixes. Another source is time pressure and astrophysics: Because of the relative orientation of the planets it may be essential to launch an exploration mission in a fairly narrow time frame. If the mission software is not ready by the time of launch, a workable strategy is to launch with a skeletal system and then upgrade while the craft is en route to the mission's destination.

All of these factors have arisen in the Cassini mission to Saturn. Needless to say, the stakes are high in on-flight software updates, hence a very conservative, patch-oriented update strategy is applied. A few documents are available online that describe the processes involved [such as (Paczkowski and Roy 2004)].

A more prosaic example is the extension of Web browser functionality. If when surfing the Web a resource is encountered that requires a novel type of display software (such as for a new type of audio encoding, perhaps), the browser must be extended. If the browser requires the user to download the new extension, perform some installation steps, and then restart the browser—or worse yet, the computer—before being able to continue, the user is inconvenienced. After restart the user will have to renavigate to the resource that demanded the extension in the first place. A more pleasant strategy, from the user's perspective, is to support the download and installation of the new software

Sony Exhibit 1018 Page 547
Sony v. Quantum IPR2023-00954

(that is, adaptation of the browser) *dynamically*, so that the user may continue usage in an uninterrupted manner.[1]

Motivations for supporting online (dynamic) change thus include:

- Nonstop applications: Ones in which the software cannot be stopped because the application cannot be stopped—the services of the software are continually required or safety-critical.
- Maintaining user or application state: Stopping the software would cause the user to lose mental context or because saving and/or re-creating the software's application state would be difficult or costly.
- Re-installation difficulty: Applications with complex installation properties, such as software embedded in an automobile.

Finally, all software adaptation is motivated by *observation and analysis*. While perhaps this is an obvious point, the role of observation and analysis is critical in the process of supporting software adaptation, and will be discussed later in the chapter. The behavior and properties of the extant system are observed and compared with, for instance, goals and objectives for the system, and to the extent that analysis reveals a discrepancy, adaptation activities are initiated.

### 14.1.2 Shearing Layers

Before going further in consideration of the adaptation of software systems it is worth a moment to consider change in physical, building architecture. Stewart Brand's insightful book, *How Buildings Learn—What Happens After They're Built* (Brand 1994), examines how, and why, buildings change over time. We have all seen this process, of course: We rearrange the furniture in our offices and homes, we install new cabling in our homes to accommodate new audio systems, we install new windows to remove drafts and make a building more energy-efficient. On infrequent occasions we remodel our homes, adding new space or reconfiguring existing space. On very rare occasions we might even see a house physically moved—for instance, if it is of historical significance but is in the way of a new civic development.

Brand categorizes the types of change that can be made to a building in terms of "shearing layers." Building upon earlier work by Frank Duffy, Brand cites six layers.

> Site. This is the geographical setting, the urban location, and the legally defined lot, whose boundaries and context outlast generations of ephemeral buildings.
>
> Structure. The foundation and load-bearing elements are perilous and expensive to change, so people don't. These *are* the building.
>
> Skin. Exterior surfaces now change every twenty years or so, to keep up with fashion or technology, or for wholesale repair.
>
> Services. These are the working guts of a building: communications wiring, electrical wiring, plumbing, sprinkler systems, . . .
>
> Space Plan. The interior layout—where walls, ceilings, floors, and doors go.

[1] Strictly speaking, any browser that supports JavaScript is dynamically adaptable. Whenever JavaScript is downloaded from a Web site and executed in the browser the user is taking advantage of dynamic extensibility. Most contemporary browsers are so good at this that the experience is completely transparent to the user.

Sony Exhibit 1018 Page 548
Sony v. Quantum IPR2023-00954

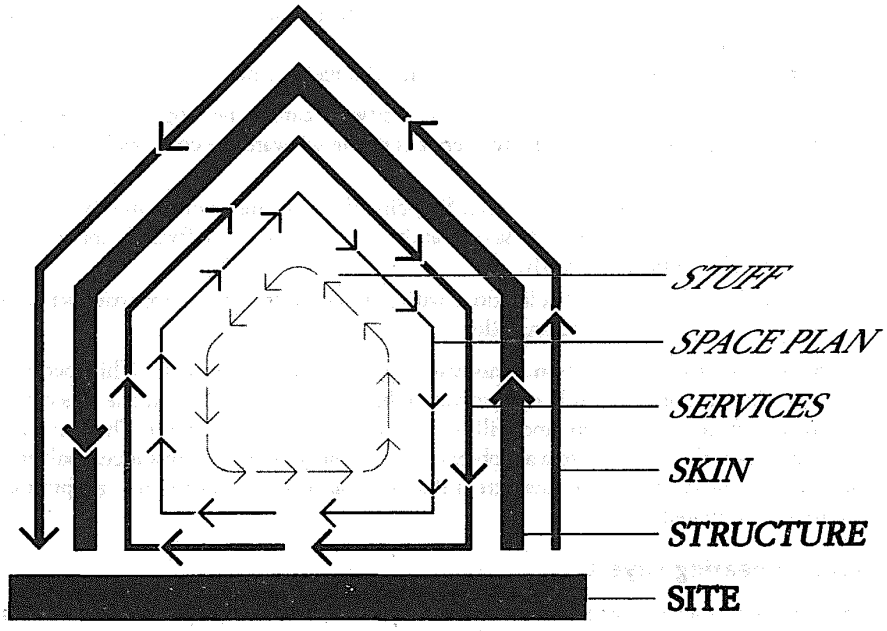



Figure 14-1. *Shearing layers of change [from Stewart Brand's* How Buildings Learn *(Brand 1994)].*

> Stuff. Chairs, desks, phones, pictures, kitchen appliances, lamps, hairbrushes; all the things that twitch around daily to monthly.
>
> (Brand 1994)

These six shearing layers are illustrated in Figure 14-1, with the arrangement of the layers in the diagram corresponding to where, approximately, the layers appear in a physical structure.[2]

Software is like this too. Though software is intrinsically more malleable, the constraints and dependencies that we impose during the process of design and implementation make it behave much like a building. As we have seen, any software system has an architecture, which determines its load-bearing elements. It has a site in its installation, that is, the usage context, whether it is a home business, a research laboratory, or a bank. Similarly, software has a "skin" with its user interfaces; further analogies can be found for the other layers. The difficult thing about software, however, is that we cannot examine a software system by looking at the source code and readily distinguish one of the system's load-bearing elements from a mere incidental—an item as incidental as the placement of a hairbrush in a house.

[2]Layers, as used by Brand and shown in Figure 14-1 do not correspond exactly to layers as used in software. In particular, the relative juxtaposition of layers in a software system, such as found in the layered style of Chapter 4, differs from the ordering of the shearing layers. As shown in this diagram, for instance, the relatively immutable structure layer is adjacent to the relatively mutable skin.

Sony Exhibit 1018 Page 549
Sony v. Quantum IPR2023-00954

One value of Brand's observations is the categorization of types of change according to the nature and cost of making a change within those dimensions. Put another way, our understanding of building layers, either as sophisticated housing developers or simply as occupants, informs us regarding what can be changed, how quickly, and roughly for what cost. For instance, as Brand noted, changes to a load-bearing element of a building—such as a steel girder—are just not done; to do so would likely endanger the structural integrity of the building. If we could make a similar categorization for software, one that would allow us to dependably identify the load-bearing elements of an application, it would help us understand the necessary techniques to effect a given change, and allow us to effectively estimate the time and cost to achieve the goals. Our intuition as to understanding the costs, difficulty, and time to make changes in any of these aspects to a building is pretty good, due at least in part to our long experience with buildings and their visible, material nature. Unfortunately, we do not usually have that kind of intuition with software, for we often cannot associate a change request with a particular type of layer in the software.

One of Brand's objectives in identifying shearing layers with their different properties is that it provides design guidance. Recognizing that the layers change at different rates, change is facilitated by limiting the coupling between the parts of a building that correspond to different shearing layers. (A simple example admonition that comes from this is, "Don't inextricably associate the appearance of a building with its services"—an admonition famously contravened by Paris's Pompidou Center, as illustrated in Figure 14-2. By making the services—such as escalators—part of the skin, changing one requires changing the other. Maintenance costs for the services now must also include costs for maintaining their very public appearance.)

These layer notions can be applied to software. In particular, the observation of layers and the admonition to decouple them is consistent with David Parnas's dictum that software engineers should "design for change." The layer concept goes beyond that simple platitude,

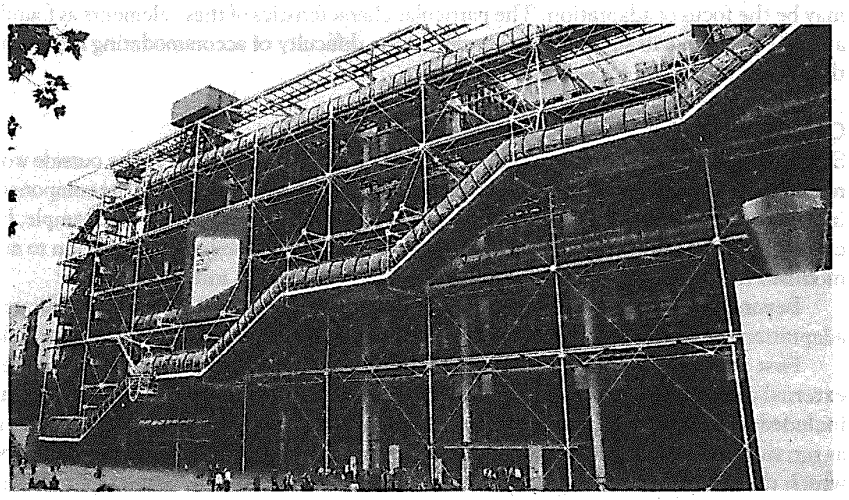


Figure 14-2. *The Pompidou Center: A good lesson in why appearance should not be inextricably linked with services.*

Sony Exhibit 1018 Page 550
Sony v. Quantum IPR2023-00954

however, to suggest specific ways of assessing whether a proposed connection between two elements of a system is appropriate or not. Are the elements proposed for connection in separate layers? In layers that are widely separated? Is the proposed connection one that maintains the essential aspects of independence for those layers?

In Brand's analysis, structure plays a somewhat distinguished role. Its relative immutability in building architecture means that it must be understood well, for it provides the bounds to the types of changes that can be applied. In the case of software, with its relative mutability, we have the opportunity for accommodating a great range of change—but only under certain conditions. We can achieve effective adaptation if we make the structure of software explicit (that is, the system's architectural configuration of components and connectors), and provide ways for manipulating that structure. At a minimum, understanding a software system's architecture enables us to assess the potential for future changes.

Finally, Brand makes a trenchant observation about building architectures that provides key insight into software adaptation: "Because of the different rates of change of its components, a building is always tearing itself apart." As introduced in Chapter 3, a concern with software change is architectural erosion, the regressive deviance of an application from its original intended architecture resulting from successive changes. If a programmer modifies an application based only upon local knowledge of the source code, it can be quite difficult to determine if the changes made cut across the software's shearing layers. Put another way, the changes to code may not respect the principal design decisions that govern the application. To the extent that the boundaries are ignored while making changes, the structure is degraded. The preventative strategy for avoiding architectural erosion of this type is to base change decisions not on local analysis of the code, but on the basis of understanding and modifying the architecture, and then flowing those changes down into the relevant code, a theme we explore below.

### 14.1.3 Structural Elements Subject to Change

Any of a system's structural elements, its components, connectors, and their configuration, may be the focus of adaptation. The particular characteristics of these elements as found in a specific system will significantly determine the difficulty of accommodating change and the techniques useful for achieving it.

#### Components

Some changes may be confined to a component's interior: Its interface to the outside world remains fixed, but, perhaps to achieve a particular non-functional property, the component's interior must be altered. A component's performance may be improved, for example, by a change to the algorithm that it uses internally. In many cases, however, adaptation to meet modified functional properties entails changing a component's interface.

Beyond this simple dichotomy a component may possess capabilities that facilitate its adaptation, or its role in the adaptation of the larger architecture within which it resides.

First among these capabilities is knowledge of self and exposure of this knowledge to external entities. A component may be built such that its own specifications are explicit and included within the component. A straightforward form and use of such specifications are as run time assertions that validate that incoming parameters satisfy the assumptions upon which the component's design depends. In a verified, static (that is, non-adaptive) world,

Sony Exhibit 1018 Page 551
Sony v. Quantum IPR2023-00954

such run time assertion checking may well be considered wasteful, but in an uncertain, adaptive world, such checking may be a useful adjunct in maintaining a robust application. A less obvious use of such assertions is in gathering information that may motivate adaptations. Assertions may repeatedly detect violation of input assumptions. If a component retains a history of such violations and exposes that history to a monitoring agent, that information may guide the architect in identifying ways in which the architecture needs to be changed. More generally, a component may make its full specifications available for inspection by external entities. Such a capability enables dynamic selection of components to achieve system goals and hence is strongly supportive of adaptation. While not supported in all programming languages, component reflection is supported in Java.

Second is (self-)knowledge of the component's role in the larger architecture. For instance, a component could be built to monitor the behavior of other components in an architecture and to change its behavior depending upon the failure of an external component. A component may be able to query whether other components exist in an architecture that could be used to obtain an equivalent service. Similarly, if a component monitors its own behavior and realizes, for example, that it is not performing fast enough to satisfy its client components, it could request an external agent to perform some load leveling or other remedial activity.

Third is the ability to proactively engage other elements of a system in order to adapt. Continuing the previous example, rather than asking an external agent to reduce the load, a component might query whether external (sub-) components are available that could be used within the component to enable it to improve its own performance. While such a scenario is currently pretty far-fetched, such a capability is only a natural progression from components that are not self-aware and not reflective, to components that are reflective but passive, to components that not only are reflective but (pro)active in supporting system adaptation.

The capabilities listed above prefigure the general adaptation techniques discussed later in this chapter. The common theme is explicit knowledge and representation of a component's specifications and knowledge of its role within the wider architecture.

***Component Interfaces.*** Changes to component interfaces are often inevitable when modifying functionality. Changing a component's interface, however, requires changes to the interface's clients as well: The input and output parameters must match at both ends of the interaction. Many client components may not be "interested" in the new functionality represented by the new interface, yet must be modified nonetheless.

This undesirable ripple effect of change has motivated creation of techniques for mitigating such impacts. A popular technique is the creation of adaptors, whereby components wanting to use the original interface invoke an adaptor component, which then passes along the call to the new interface, as illustrated in Figure 14-3. A client needing the new interface calls the modified component directly.

Use of a technique such as this clearly has architectural implications: New components are introduced and tracing interactions to understand system functioning becomes more involved. Subsequent changes to one of the previously unmodified methods become even more complex: Should yet another adaptor be introduced? Should the method be changed in the root component and in all the previously created adaptors? Suffice it to say for now that "local fixes" that attempt to mitigate changes to component interfaces can have

Sony Exhibit 1018 Page 552
Sony v. Quantum IPR2023-00954

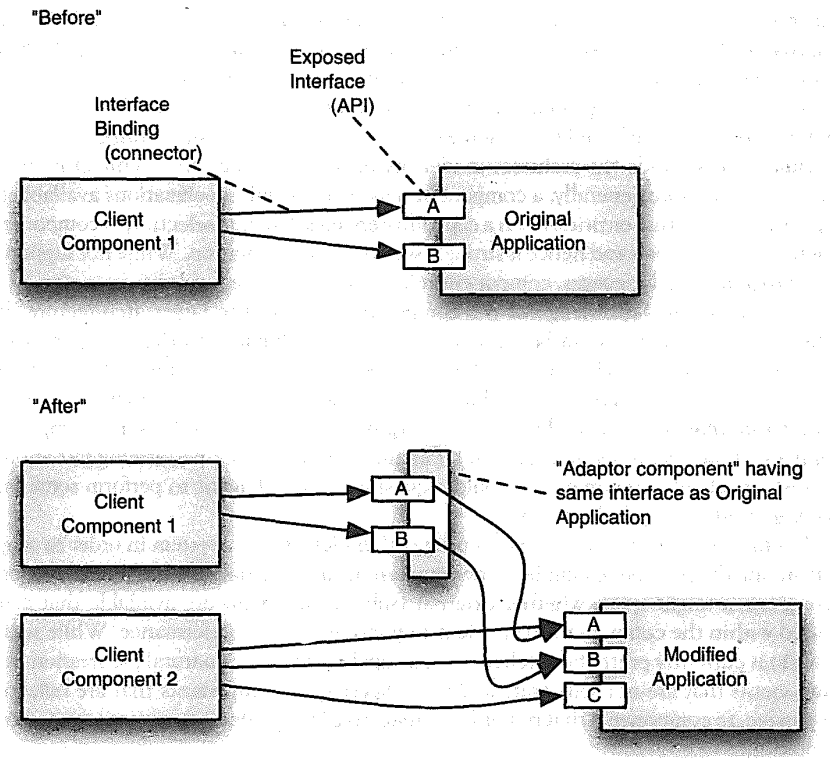



Figure 14-3. *Use of adaptor components to preserve interfaces.*

serious repercussions; the architect is wise to consider, in advance, a range of techniques for accommodating change, as discussed later in this chapter.

### Connectors

Just as components may be required to change, so may connectors, and depending on the type of change, the architectural implications may be profound or may be negligible, similarly for the resulting changes in the implementation. Typically, the motivation for changes to connectors is the desire to achieve altered non-functional properties, such as distribution of components, heightened independence of subarchitectures in the face of potential failures, increased performance, and so on.

Changes to connectors may be along any of the dimensions of connector design discussed in detail in Chapter 5. As the latter part of this chapter will show, choice of connector type is a major determinant of the effort involved in supporting architectural evolution. Roughly speaking, the more powerful the connector, the easier the architectural change. Connectors that remain explicit in the implementation code and that support very strong decoupling between components (such as those connectors supporting event-based communication) are superior for supporting adaptation.

Sony Exhibit 1018 Page 553
Sony v. Quantum IPR2023-00954

### Configurations

Changes to the configuration of components and connectors—that is, how they are arranged and linked to one another—represent fundamental changes to a system's architecture. Such changes may occur at any level of design abstraction, and may be motivated by the need to meet new functional requirements or to achieve new non-functional properties. For instance, a common, large-scale configuration change is to move from an integrated business-to-business application model to an open, service-oriented architecture. The custom application may enable two businesses to communicate and transact business very efficiently, but a move to an open, service-oriented architecture may allow each party to more easily add new functionality, or support interactions with other business partners. Many of the core components may remain unchanged, but the way the components interact with each other and the connectors through which they communicate may be profoundly altered.

Effectively supporting such a modification requires working from an explicit model of the architecture. Many dependencies between components will exist, and the architectural model is the basis for managing and preserving such relationships. Most importantly, the constraints that governed creation of the initial architecture must be maintained if the architecture is to retain the same style. If maintaining the governing style constraints is not possible, then deliberative analysis should take place to determine an appropriate style for the new application. The move from custom, tightly integrated business-to-business applications to service-oriented architectures is precisely such a change: The fundamental architectural style of the application is altered.

The architectural model will need to manage deployment characteristics too, as basic component-connector relationships, in keeping with the discussion of Chapter 10.

## 14.1.4 Change Agents and Context

Techniques for effecting change are determined not only by what is ultimately changed, but also by the *context* for the change and the character of the *agent(s)* performing the change. Important aspects of the context for change include:

- The factors behind the change motivation: how a system is observed and what analysis can be performed on those observations.
- The timing of the change: when in the development/deployment/use time line the change occurs.

The agents for achieving change include both people and software. Important attributes of the change agents include:

- The location of the agent with respect to the system being changed.
- The knowledge possessed by the change agent about the system, whether complete or partial.
- The degree of freedom possessed by the change agent.

Each of the above items is considered in the following paragraphs.

Sony Exhibit 1018 Page 554
Sony v. Quantum IPR2023-00954

### Motivations, Observations, and Analysis

Several types of motivation for change were discussed earlier. Depending on the particular motivation that is at work, different problems and opportunities for dealing with the change arise. All motivations spring from a combination of observations about a system and analysis of those observations. For example, a system may not perform some newly desired function (as determined by comparing its current behavior to the newly sought behavior), or it may need to be fixed (as determined by comparing its current behavior to the original specification). The key issue is where the observations about a system originate.

Some observations may be drawn from the engineer's review of the system's architecture. More likely is the situation where a user, or some external agent, observes the run-time behavior of the application. To the extent that these observations are tied or related to information about the architecture of the application, the adaptation task is eased. If the only observations are those that come from seeing an application's external functional behavior, then determining the related structure is a task akin to debugging: The offending behavior has to be (possibly laboriously) examined to see what parts of the application are responsible. If the application has, or can have, probes included within it that are designed to assist the analyst determine which internal elements are relevant, the adaptation can proceed more quickly. Every beginning programmer is familiar with this technique in its simplest form: the inclusion of dozens of `println` functions, to repeatedly indicate where the program is executing and the values that particular variables have.

As architectures become more complex the beginning programmer's print line technique becomes inadequate. The concept, though, is still valuable. Monitors of the architecture can be included in a system (typically in the connectors) to enable high-quality monitoring of the behavior. The more precise the information and the analysis based on that information, the better the change can be planned and managed. In extreme cases, the absence of a good technique for identifying the components responsible for a particular behavior may cause an engineer to replace an entire subsystem. With good information, however, a much more localized change may suffice. We return to the topic of embedded probes in Section 14.3.1 below.

### Timing

The positioning of change within a project's time line greatly affects the techniques available for accommodating the change. In short, the earlier the better. Rather than just repeating the old maxim that fixing problems in requirements is substantially cheaper than fixing them in code, we expand the time line to include system deployment and system use, for indeed some applications may need to be changed in situ, on the fly. A notional graph showing the relationship between cost and timing is found in Figure 14-4.

In Figure 14-4, the steep rise in cost is associated with accommodating change once a system is deployed; the costs top out when a deployed system must be changed on the fly. Deployment adds the costs of dealing with each individual installation of a system—possibly behind firewalls and customized at time of installation. On-the-fly adaptation requires special mechanisms and circumstances, a topic we treat on its own in Section 14.3.3.

Accommodating change early in a project's life is not the same, however, as fixing problems in the requirements. Identification, early in a system's design, of types of potential changes enables insertion of mechanisms specifically intended to facilitate those changes when they happen later. Far beyond a banal "design interfaces to be resilient in the face

Sony Exhibit 1018 Page 555
Sony v. Quantum IPR2023-00954

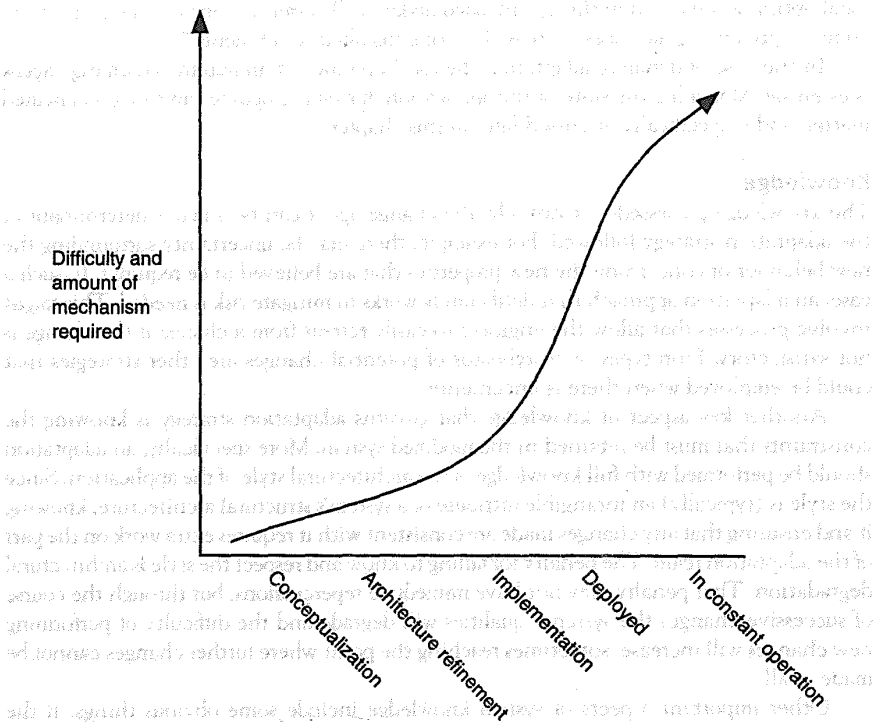



Figure 14-4. *Notional relationship between the timing of the identification of a change and the difficulty of and mechanism required for supporting that change.*

of potential, identifiable changes to the internals of a module," numerous sophisticated techniques designed to enable broad adaptation have been created. These include plug-in architectures, scripting languages, and event interfaces. These and others will be described in Section 14.3.2.

### Change Agents: Their Identity and Location

The processes that carry out adaptation may be performed by human or automated agents, or a combination thereof. Some trade-offs between these two types of agents are obvious, such as noting that humans can deal with a much wider range of problems and types of change or that automated agents can act with greater speed. Other issues may not be so obvious. One particular value of automated change agents is in the domain of deployed and continuously operating systems. If an adaptation change agent is part of a deployed application from the outset, the potential is present for an effective adaptation process: Getting to the deployed application is not a question. Moreover, the agent may have access to contextual information, whereby details of any local modifications and customizations are available. This potentially enables the change agent to act differently in every individual deployment. Users of modern desktop operating systems are familiar with such agents: Periodically a user is notified when an OS or application upgrade is available. If the agent is at all sophisticated, the advice given regarding the updates to be performed will be based upon knowledge of any

Sony Exhibit 1018 Page 556
Sony v. Quantum IPR2023-00954

local optimizations. (Often the downloaded update will perform a repeat or supplementary analysis, presenting messages such as "Locating installed components.")

In the case of dynamic adaptation the local presence of an automated change agent is essential. Managing the state of the application during the update can be a complicated matter, and is specifically discussed later in this chapter.

### Knowledge

The knowledge possessed—or not—by the change agent can be a major determinant of the adaptation strategy followed. For example, there may be uncertainty surrounding the new behavior or concerning the new properties that are believed to be required. In such a case, an adaptation approach that deliberately works to mitigate risk is needed. This might involve processes that allow the engineer to easily retreat from a change if the change is not satisfactory. Prototypes or storyboards of potential changes are other strategies that could be employed when there is uncertainty.

Another key aspect of knowledge that governs adaptation strategy is knowing the constraints that must be retained in the modified system. More specifically, an adaptation should be performed with full knowledge of the architectural style of the application. Since the style is (typically) an intangible attribute of a system's structural architecture, knowing it and ensuring that any changes made are consistent with it requires extra work on the part of the adaptation team. The penalty for failing to know and respect the style is architectural degradation. That penalty may not have immediate repercussions, but through the course of successive changes the system's qualities will degrade and the difficulty of performing new changes will increase, sometimes reaching the point where further changes cannot be made at all.

Other important aspects of system knowledge include some obvious things. If the system's architecture has not been retained, then architectural recovery will be required. If a system component is a purchased binary, for which no information is available concerning its adaptation properties (if any), then large granularity changes will be needed ("component transplants"). If no analysis tools are available to help assess properties of a proposed change, then perhaps a more iterative approach to the adaptation should be followed. When it comes to adaptation, the more knowledge of the system and the change, the better.

### Degree of Freedom

The final aspect of the context of change is the degree of freedom that the engineer has in designing the changes. One might think that greater freedom is always better. Unfortunately, greater freedom comes at a significant cost: The search space for solutions is larger, there is less immediate guidance for the engineer on how to proceed, and assessing all of the consequences and properties of a possible change will take more effort. The assumption of this statement, of course, is that the engineer is attempting to do a high-quality job; the cost can always be kept down in the short term by simply adopting whatever adaptation scheme first comes to mind, and doing the work in the same manner or style as the engineer always does. The price is paid in the future, when, for example, it is discovered that the change inhibits further growth or otherwise degrades the system's architectural quality. The alternate approach is to know or learn the constraints of the application and work within those constraints to satisfy the needs. With this approach, coherency

Sony Exhibit 1018 Page 557
Sony v. Quantum IPR2023-00954

is retained and the engineer's energies are directed toward solutions within the current architectural style.

### 14.1.5 Architecture: The Central Abstraction

The final concept of architecture-centric adaptation is the central one: the architectural model. An explicit architecture is assumed in much of the preceding discussion. In the absence of an explicit architecture, the engineer is left to reason about adaptation from memory and from source code—neither of which has proven very effective in managing change. The presence of an explicit architecture provides a sufficient basis for the planning and execution of system adaptation. Since the architecture *is* the set of principal design decisions governing the system, adaptation based on the architecture proceeds from knowledge of those things that are most intrinsic to the system's design.

The critical need, of course, is to maintain the integrity of the architecture and its explicit representation throughout the whole course of adaptation. As discussed earlier, in Chapters 2 and 9, the most important and most difficult aspect of this task is maintaining consistency between the architecture and the implementation. As long as the implementation is faithful to the architecture, the architecture can serve as the primary focus of reasoning about potential adaptations. As soon as the architectural model fails to be a reliable guide to the code it loses its value for guiding changes; the engineer is left with just the code—the "software engineering" of the 1970s.

Note that apropos of the discussion above regarding the location of change agents, if the agent is an automated part of a deployed system, then for architecture to be the central abstraction governing adaptation, the architectural model must either be deployed with the application or communication with an outside reference must be possible. If the model is not available, the agent does not have any basis for architecture-based adaptation.

The next section places all the concepts discussed above into a comprehensive framework for architecture-based adaptation.

## 14.2 A CONCEPTUAL FRAMEWORK FOR ARCHITECTURAL ADAPTATION

The first comprehensive conceptual framework for architecture-based adaptation was presented in 1999 (Oreizy et al. 1999). The framework, shown in Figure 14-5, shows the principal activities and entities and the key relationships among them. The diagram is simple enough to explain how a very static, requirements-based adaptation process proceeds, but sophisticated enough to indicate how a highly dynamic, automated adaptation process can work. Quoting from Peyman Oreizy and his colleagues' article,

> The upper half of the diagram, labeled "adaptation management," describes the lifecycle of adaptive software systems. The lifecycle can have humans in the loop or be fully autonomous. "Evaluate and monitor observations" refers to all forms of evaluating and observing an application's execution, including, at a minimum, performance monitoring, safety inspections, and constraint verification. "Plan changes" refers to the task of accepting the evaluations, defining an appropriate adaptation, and constructing a blueprint for executing that adaptation. "Deploy change descriptions" is the coordinated conveyance of change descriptions,

Sony Exhibit 1018 Page 558
Sony v. Quantum IPR2023-00954

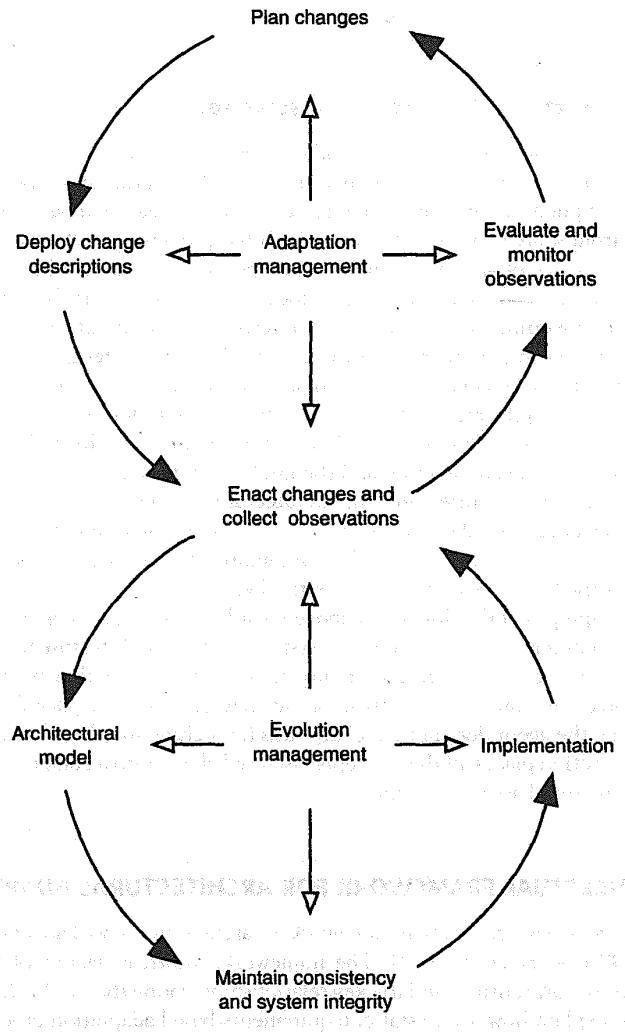



Figure 14-5. *Conceptual architecture for adaptation. From (Oreizy et al. 1999). © IEEE 1999.*

components, and possibly new observers or evaluators to the implementation platform in the field. Conversely, deployment might also extract data, and possibly components, from the running application and convey them to some other point for analysis and optimization.

Adaptation management and consistency maintenance play key roles in our approach. Although mechanisms for runtime software change are available in operating systems (for example, dynamic-link libraries in Unix and Microsoft Windows), component object models, and programming languages, these facilities all share a major shortcoming: they do not ensure the consistency, correctness, or other desired properties of runtime change. Change management is a critical

Sony Exhibit 1018 Page 559
Sony v. Quantum IPR2023-00954

aspect of runtime-system evolution that identifies what must be changed; provides the context for reasoning about, specifying, and implementing change; and controls change to preserve system integrity. Without change management, the risks engendered by runtime modifications might outweigh those associated with shutting down and restarting a system. . . .

The lower half of [the diagram], labeled "evolution management," focuses on the mechanisms employed to change the application software. Our approach is architecture-based: changes are formulated in, and reasoned over, an explicit architectural model residing on the implementation platform. Changes to the architectural model are reflected in modifications to the application's implementation, while ensuring that the model and the implementation are consistent with one another. Monitoring and evaluation services observe the application and its operating environment and feed information back to the diagram's upper half.

Figure 14-6 presents a further refined framework. Rather than the somewhat arbitrarily labeled "adaptation management" and "evolution management" of Figure 14-5, this figure

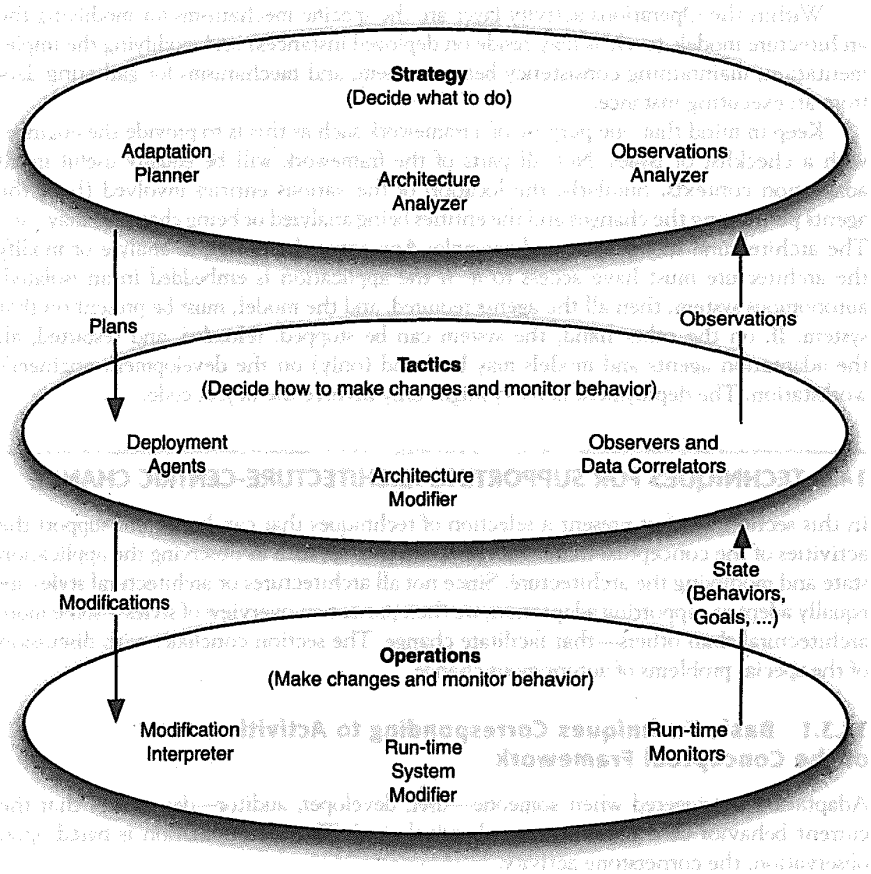



Figure 14-6. *Activities, agents, and entities in architecture-based adaptation.*

separates the activities into three types and shows the entities and agents involved in carrying out the activities. The essential insights of the earlier framework are present, but now with additional detail and precision.

The three activities are characterized as strategic, tactical, and operational. Strategy refers to determining what to do, tactics to developing detailed plans for achieving the strategic goals, and operations to the nuts-and-bolts of carrying out the detailed plans.

Within the Strategy activity layer are agents for analyzing information about the system and the architecture, then planning adaptations based on that analysis. These agents may simply be tasks that a human user performs, or may be automated programs that carry out such tasks.

Within the Tactics activity layer are agents for making specific architectural modifications based upon the plans received from the strategy layer and deployment agents for seeing that those modifications are conveyed to all deployed instances of the architecture to be modified. Also found here are agents for determining how to correlate raw monitoring information received from deployed instances of the system such that the collected data is suitable for analysis at the strategy level.

Within the Operations activity layer are the specific mechanisms for modifying the architecture models (such as may reside on deployed instances) and modifying the implementation, maintaining consistency between them, and mechanisms for gathering data from an executing instance.

Keep in mind that one purpose of a framework such as this is to provide the engineer with a checklist of issues. Not all parts of the framework will be equally useful in all adaptation contexts. Similarly, the location of the various entities involved (both the agents performing the changes and the entities being analyzed or being changed) may vary. The architectural model is a good example: Any agent that needs to analyze or modify the architecture must have access to it. If the application is embedded in an isolated, autonomous system, then all the agents required, and the model, must be present on that system. If, on the other hand, the system can be stopped, reloaded, and restarted, all the adaptation agents and models may be found (only) on the development engineer's workstation. The deployment activity might only involve the object code.

## 14.3 TECHNIQUES FOR SUPPORTING ARCHITECTURE-CENTRIC CHANGE

In this section, we first present a selection of techniques that can be used to support the activities of the conceptual framework presented above, such as observing the application state and modifying the architecture. Since not all architectures or architectural styles are equally adept at supporting adaptation, we then present an overview of styles—some more architectural than others—that facilitate change. The section concludes with discussion of the special problems of autonomous change.

### 14.3.1 Basic Techniques Corresponding to Activities of the Conceptual Framework

Adaptation is triggered when someone—user, developer, auditor—determines that the current behavior of a system is not what is desired. That determination is based upon observation, the cornerstone activity.

### Techniques for Observing and Collecting State

Determining the techniques that are most useful for observing and collecting state information presupposes knowing what "state" is. In its most obvious form, state consists of the run time values of a program's objects. Since values are continually changing, state is relative to time, and since only a small subset of a program's state is likely to be of use in adaptation, the observed state is usefully a time-stamped sequence of a subset of the run time values of the program's objects.

Other information beyond the values of a program's variables may also be important parts of state used for adaptation purposes. For instance, properties of the program's environment that are not represented in the program may be essential in determining how an application should be adapted. It may be, for example, that the behavior of a system should be a function of some currently unrepresented attribute of the external environment. An automated teller machine, for instance, should perhaps enter a safe mode and wirelessly signal an alarm whenever the ATM is subjected to G-forces in any direction (such as might be caused by an earthquake—or by someone attempting to abscond with the machine). Gathering this enhanced notion of a program's state would entail monitoring more than just the program.

A more prosaic example of external information that may be critical in determining an adaptation strategy is deployment information. The behavior of a system may be impacted by the presence of certain platform-specific options—the presence of certain device drivers, the amount of memory available, or the version of the operating system, for instance. Such information is usually easy to gather (for instance, from "registries" such as MS Windows registry database or Mac OS X's library files) and may be essential in the analysis phase.

System specifications and the system's architectural model may also be aspects of state that need to be observed on the target system and brought to the analysis context. One might expect this information to be static and already known to the adaptation analyst/planner, but in some contexts that may not be true. For example, if an application is highly configurable, each running instance of the application may be unique—perhaps because of customization to reflect installed hardware options. This information is thus akin to the deployment information mentioned above: In that case, the site-specific information concerns properties external to the application to be adapted; in this case, the information is about the application itself. The observed architecture, therefore, is the architecture of the application as it exists in its fielded state. Its set of components, connectors, and their current configuration may be unique. In some more esoteric situations, such as autonomous adaptation, the fielded system may also contain explicit and dynamically changing goal specifications for the application. These would also constitute part of the state subject to observation and reporting to the adaptation analyst.

***Gathering the State.*** Numerous techniques exist for gathering the state of a program's objects; the numerous techniques developed to support program debugging all apply. Most primitively, the analyst may manually observe the program's user interface. More usefully, observation code may be part of a special run time system that enables monitoring of a program without requiring any modification to it. In some contexts, such as specialized embedded systems, "bolt-on" hardware analyzers may allow inspection of program state without any disturbance to the run time software environment.

Sony Exhibit 1018 Page 562
Sony v. Quantum IPR2023-00954

If modification of the subject software system is feasible, a variety of more specialized and targeted approaches are possible. Again, at the most primitive level, the beginning programmer's technique of seeding a program with print statements may be sufficient. As program and problem complexity increase, more powerful and thoughtful approaches are required. Customized monitors may be inserted in the application code and function similarly to assertions. Such assertions can check for specified conditions and, if satisfied, record some value or otherwise emit an observation of the program's state. Whether a given programming language's assert statement is appropriate for this use depends on the language semantics: If satisfaction of the assert's condition necessarily interrupts the program's flow of control following the assertion, then use of that language feature would not be appropriate. Monitoring should allow execution to proceed unimpeded.

To the extent that creation of custom monitors is a manual process, it is an expensive process. Moreover, the use of custom monitors such as described above is only feasible when the source code is available and the application can be recompiled. One alternative is to focus on capturing information only at component boundaries, and to do so by capturing information that passes through an application's connectors. This technique can be based upon commercial middleware when used for connectors, or may take other forms. If the information gathering is automatic and does not filter "uninteresting" data, then it must be coupled with other tools to support filtering after the raw data has been gathered.

The concept is illustrated by the MTAT system (Hendrickson, Dashofy, and Taylor 2005), which creates a log of all messages sent between components in an architecture during its execution. This log is created by automatically implanting trace connectors in an architecture (see Figure 14-7). Trace connectors intercept all messages passing through them, make a copy of each message, and send each copy to a distinguished component that logs the messages in a relational database. The original messages are passed on unmodified. The trace connectors are first-class connectors. Figure 14-8 shows an architecture before modification as well as the architecture after inserting trace connectors.

The MTAT analysis system includes a tool that examines an xADL description of a system's structure and inserts trace connectors into that description automatically. The system's structure is modified so that each link in the original architecture is split into two links, with a trace connector between. Because the infrastructure provided by xADL and its supporting ArchStudio tool set instantiates architectures directly from their descriptions,

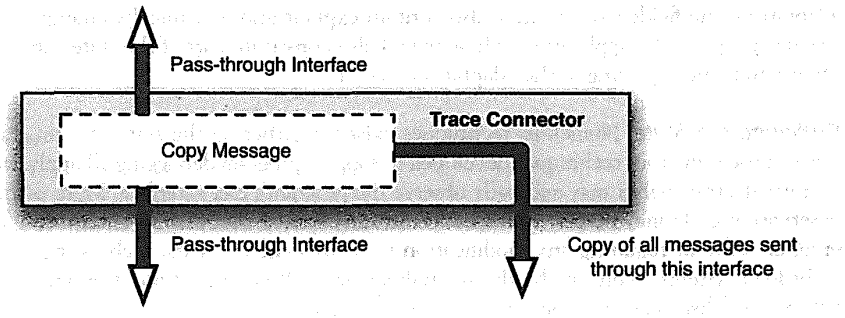



Figure 14-7. *A trace connector.*

Sony Exhibit 1018 Page 563
Sony v. Quantum IPR2023-00954

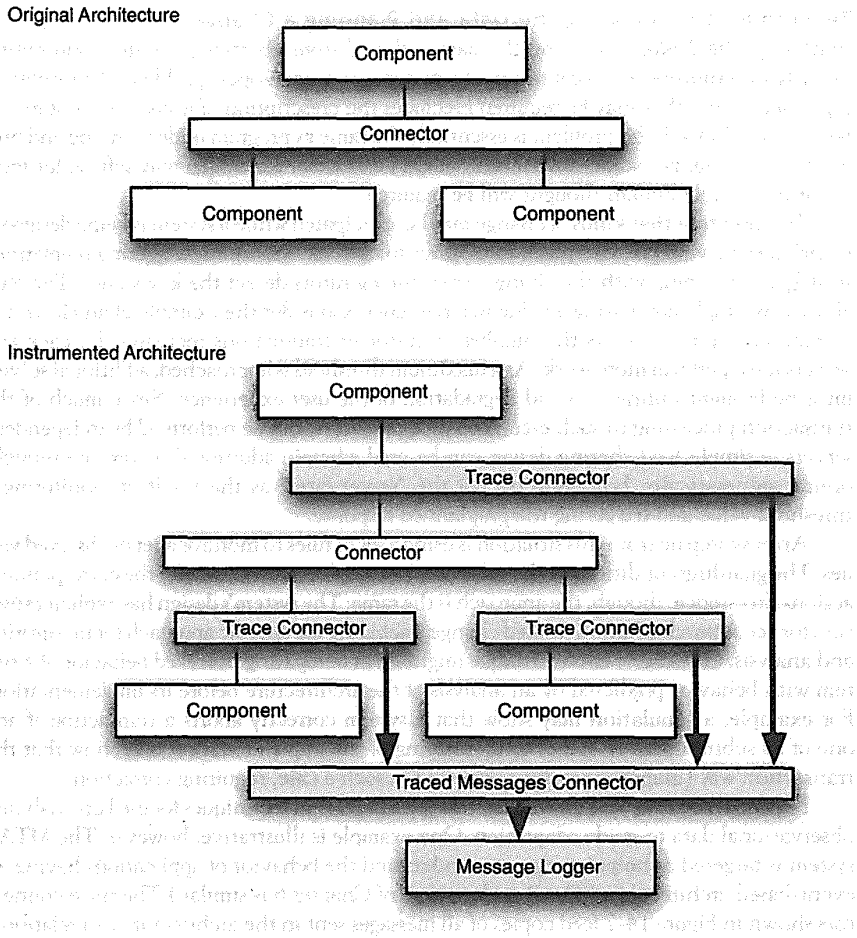



Figure 14-8. *Instrumented architecture and original architecture.*

no recoding of components is needed for instrumentation. Consequently, components in the application remain unaware of any architectural changes or the presence of the trace connectors once the architecture is modified. The approach works in both single-process and distributed systems. In distributed systems, connectors that bridge process boundaries are broken into two halves—one in each process they connect. In terms of instrumentation, distributed connectors are treated the same way single-process connectors are; a trace connector is placed on every link in every process.

Finally, depending on the adaptation context, the technique used for gathering the observations may also need to be responsible for transmitting them from the platform where the target application is running to the host where the adaptation analysis will take place. To the extent that the target is an embedded system this off-board transmission is likely to be a necessity.

Sony Exhibit 1018 Page 564
Sony v. Quantum IPR2023-00954

## Techniques for Analyzing the Data and Planning a Change

***Analyzing the Data.*** Analyzing the data gathered from observing a system and consequently determining the adaptations to perform is a wide-open problem. The range of types of changes that may be required precludes the prescription of a small set of standard techniques. Indeed, the problem is essentially the same as program understanding and program debugging. For some situations, a simple, canonical technique may suffice; for most situations, ample human thought will be required.

To the extent that kinds of change may be anticipated while a system is being designed, techniques may be put into place to monitor for specific situations, triggering preplanned strategies for coping with the change when the monitors detect the key values. The data analysis is simply comparing a value to a reference. Consider the example of an electronic commerce application. As the number of customer transactions increases the back-end servers must perform more work. As a maximum threshold is approached, additional servers must be brought online to avoid degradation of the user experience. Since much of the transaction processing in such e-commerce applications can be performed by independent servers, a simple load-sharing design can be used wherein additional servers are brought online automatically, becoming part of the "server farm" as the result of monitoring a threshold value and triggering the preplanned response.

An easy step up from this situation is using a set of rules to monitor a set of observed values. The guarding conditions in the rules may be complex; so, too, may be the corresponding actions. In essence, though, the approach is the same: The system's design has explicit provision for accommodating anticipated change, including the specific approach for monitoring and analysis. One use of this technique might be in comparing observed behavior of a system with behavior predicted by an analysis of the architecture before its implementation. For example, a simulation may show that a system correctly aborts a transaction if any one of its subtransactions fails; but monitoring of the implementation may show that the transaction was (inappropriately) completed in such a case, requiring correction.

Beyond these simple approaches there are few general techniques focused on analyzing observational data to guide adaptation. One example is illustrative, however. The MTAT system is targeted at helping engineers understand the behavior of applications having an event-based architecture. (The Rapide system of Chapter 6 is similar.) The trace connectors shown in Figure 14-7 feed copies of all messages sent in the architecture to a relational database. This database can then be examined in various ways to determine, for instance, communication patterns in the application or causality chains. Causality chains are particularly useful: When an engineer understands that message *a* from component *A* causes message *b* to be sent from component *B* and so on, changes can be made that reflect a deep understanding of the interrelationships between elements of a system's architecture. Note that, in essence, this approach is similar to techniques employed in debugging source code; the difference is in bringing the analytic approach to bear at the level of architectural concepts, and focusing on the particular kind of communication used at the architectural level.

***Analyzing the Architecture and Proposing a Change.*** Assuming that observations of a system have been obtained and analyzed, it remains to develop a modification to the architecture to meet the needs with which the architect is concerned. Again, the challenge may be very simple or very complex.

If the architecture was originally designed to accommodate certain types of change, and analysis of the data shows that the needed change is within the expected scope, then

Sony Exhibit 1018 Page 565
Sony v. Quantum IPR2023-00954

proposing a responsive change is straightforward. For instance, if a component is shown to be a performance bottleneck, and a faster replacement component having the same interfaces can be produced (that is, the change is confined to the component's internal "secrets"), then simply swapping the original component for the improved one is all that is required. If a system's architecture is in the client-server style and a new client is needed, modification is simple. If a system's architecture is in the publish-subscribe style and a new publisher is available, then again the path to change is straightforward. The pattern here is clear: If the needed change fits within the type of change anticipated and accommodated by the application's architectural style, then the approach to take is clear.

If analysis shows that the needed change does not fall within an anticipated pattern, then the architect has to revert to the general analysis and design techniques discussed earlier in Chapters 4 and 8.

**Patches, Service Packs, Upgrades, and Releases**

Commercial systems use a variety of techniques and terminology in supporting "upgrades" of deployed software products. The terms *upgrades*, *releases*, *service packs*, *patches*, and others are all used. There is little consistent technical meaning for these terms. Patches were initially used to refer to small changes to a file, but some have used the term for what amounts to wholesale replacement of the original file. Patches can be applied to binary as well as textual files. The term *service pack* is usually used to denote a collection of changes to an application, but there is no consistency on this either.

The means of deployment and installation of patches (whichever term is used) can vary significantly. Some deployments are initiated by the deployed software itself—that is, an instance of a system checks with the home site for available upgrades and, if found, proceeds to initiate a download. In other occasions, the vendor will initiate the deployment, attempting to notify all user sites of the need for the upgrade. Once deployed the patch may be installed automatically or may require active user involvement.

The installation activity may be unsophisticated or may involve checking the context—determining the other components or applications that are installed that might determine the particular change to make.

All of these activities echo or embody some of the techniques involved in a robust architecture-based adaptation process, such as described in this chapter.

***Analyzing the Proposed Change.*** After a proposed change to an architecture has been developed it should be checked to determine if it is complete and appropriate for deployment to its target platforms. Checking a proposed change seems so obvious that it should go without saying, but consider some of the possible items that may need to be checked:

- Are there any unattached ("broken") links in the new architecture?
- Have all interface links been type-checked for consistency?
- Are all components properly licensed for use in the target environments? (For example, if the target environment is a commercial product, do any of the new components have GPL licenses attached?)
- Are communications between the components performed consistently with the application's security requirements? (For example, if two components communicate over an open network, are the messages encrypted?)

Sony Exhibit 1018 Page 566
Sony v. Quantum IPR2023-00954

- Is every component and connector in the revised architecture mapped to an implementation?
- Is everything in the deployment package that the change agent on the target platform requires? (For example, license keys and configuration parameters.)

The number of issues that may need to be checked is large, and hence an explicit process step to verify them is usually warranted.

One situation that calls for special attention is when each target system to which the revised architecture is to be deployed may be unique. That is, due to the possibility of local modifications, each target needs, in essence, to be individually checked. If the number of targets is at all significant, the analysis will have to be automated.

### Techniques for Deploying Change Descriptions and Modifying the Architecture

***Describing Changes to an Architecture.*** To effect a repair on a running software system, the changes to the system that will occur because of the repair must be specified and machine-readable. There are three primary ways that concrete architectural changes can be expressed, as follows.

- *Change scripts:* Change scripts are executable programs that operate on a system to make changes. In general, these scripts leverage underlying adaptation APIs provided by the implemented system (or its architecture implementation framework or its middleware). Adaptation APIs provide high-level functions, for instance, for instantiating and removing components and connectors, creating and destroying links, exposing or withdrawing provided services, and so on. Sometimes, change scripts will be accompanied by implementation artifacts (for example, new source or binary files) that are merged with the implemented system.
- *Architectural differences:* Rather than executable programs or scripts, architectural changes can be expressed as a set of differences between one architecture and another. In general, these differences, also known as "diffs," contain a list of additions, removals, and (optionally) modifications to architectural elements. For example, a diff might indicate the addition of two components, new interfaces on those components, and new architectural links to hook those components into the architecture, as well as the removal of links that are no longer needed. Given two concrete architectural models of the system—as it is and the system as it should be—tools can create a diff automatically.
- *New architectural models:* In some cases, especially cases where the current state of the system to be adapted is unknown, a complete architectural model is used to describe the target adaptation architecture. In this case, an architectural difference is generally created by examining the elements in the current system as well as those in the new model and determining what to add and what to remove to bring the current system in line with the new architecture. Whether or not this architectural difference is expressed as a new artifact or is simply a side effect of determining the changes depends on the approach.

***Applying Changes to an Architecture.*** In any of these three cases, two worlds are involved—the world of architecture and the world of implementation. In general, for

any of these change artifacts to be effective in adapting a real, implemented system there must be a tight correspondence between elements in the architectural model and elements in the implementation. The use of architecture implementation frameworks or flexible middleware (see Chapter 9) can be used to facilitate this.

In the absence of tight bindings between architecture and implementation, it is conceivable to employ a human as the change agent on the target system. That is, the change artifact is reviewed by a person, and that person is responsible for making the corresponding changes to the system. Certain software systems [for example, enterprise resource planning (ERP) systems] are so complex to deploy, install, and configure that customers simply cannot do it themselves. In general, an organization that purchases such software also hires a team of consultants or product experts to deploy, install, and configure the software for their own organization. In this case, architectural changes might be deployed from the vendor to the consulting team, which could then make the configuration changes to the system manually.

Once changes to a system's architecture are described concretely in one of the above forms, they must be actually applied to that system. In the case of change scripts, the change script is executed in some run time environment co-located with the target application. Software underlying the target application, usually the architecture implementation framework or middleware, is called upon to actually make the changes to the system. In the case of architectural diffs, a change agent running alongside the target application must interpret the diff and make the changes specified therein, a process known as merging. In the case of new architecture specifications, a change agent must determine the difference between the current and intended application configuration and make changes as needed.

***Issues with Deployment.*** No matter what form of change artifact is used—change scripts, architectural diffs, or new architectural specifications—that artifact needs to be deployed to the target system (or its associated change agent). The issues involved with deploying these change artifacts are similar to those found in software deployment in general, and so most of the advice in Chapter 10 on deployment applies here as well. If the target system is running on a single remote host, deployment is generally straightforward: The change artifact can be sent over any number of network protocols such as HTTP or FTP to the change agent, which then executes the changes. Distributed and decentralized systems can use push-based or pull-based approaches to distribute change artifacts.

In any of these approaches, a change agent must be located on the target machine to actually effect the changes on the architecture—to execute the change script, to parse the architectural diff, and so on. This agent can be deployed with the original system, or it can be sent along with the change artifact. For example, a change script could be compiled into executable form (as an "installer" or "patcher") and then sent over the network to the target system, where it is executed by the operating system directly. This is convenient as it avoids the problem of having to update change agents themselves. This strategy carries some security risk—running arbitrary executables on target platforms is an easy way to spread malicious software if the code is not trusted. Code signing and other strategies can be used to mitigate this risk.

***Issues with Applying Changes.*** Making on-the-fly architectural changes (see Section 14.3.3 below) is more difficult. All issues identified below with respect to ensuring that the application is in an appropriate quiescent state to be adapted apply with these strategies.

Sony Exhibit 1018 Page 568
Sony v. Quantum IPR2023-00954

Making on-the-fly changes to a distributed system is even more difficult, since the change must first be deployed to all appropriate parts of a distributed system, and then the system as a whole must be put into a quiescent state for update. This involves additional risk, since network and host failures during the update process can make it difficult to get the system into a consistent state.

### An Example: ArchStudio

The ArchStudio environment supports architecture-based adaptation with a combination of techniques listed above. Specifically, it uses an architecture-implementation framework to ensure a mapping between architecture and implementation. Then, architectural diffs are used as change descriptions to evolve a system at run time. The process for evolving a system occurs in four steps.

Step 1 of the process is shown in Figure 14-9. An architecture description in xADL is provided to a tool in ArchStudio, the Architecture Evolution Manager (AEM). The AEM reads through the model and uses the Flexible C2 Framework, described in Chapter 9, to instantiate the system. Implementation bindings in the xADL model facilitate the mapping from architectural components and connectors to Java components.

Step 2 of the process is shown in Figure 14-10. A new architectural model is created in the environment, possibly by making a copy of and modifying the old model. Then, the old model and the new model are both provided as input to another ArchStudio tool, the Differencing Engine, "ArchDiff." ArchDiff creates a third document, an architectural diff, that describes the differences between the two.

Step 3 in the process is shown in Figure 14-11. Here, the new model is no longer needed. The original model, still describing the running system, and the diff are both provided as input to another ArchStudio tool, the Merging Engine, "ArchMerge." The merging engine, acting as a change agent, modifies the original model with the changes in the diff.

Step 4 in the process is shown in Figure 14-12. Here, the AEM notices that changes were made to the original model (by way of events emitted from the model repository

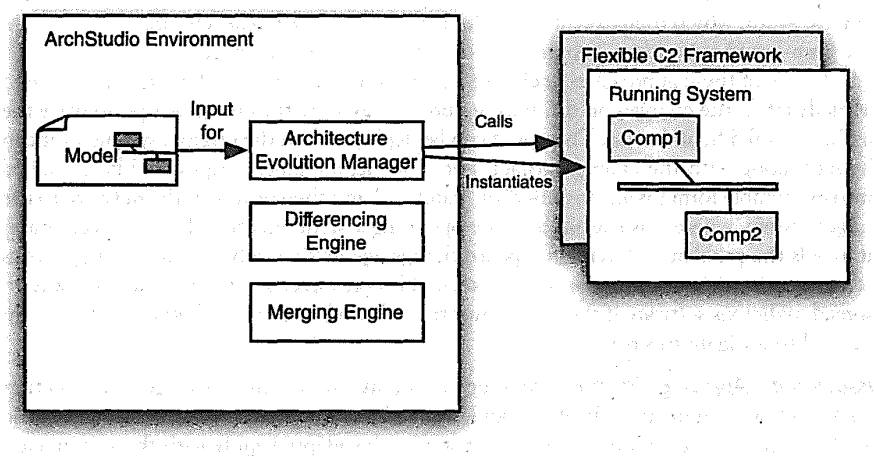



Figure 14-9. *ArchStudio adaptation step1: instantiation.*

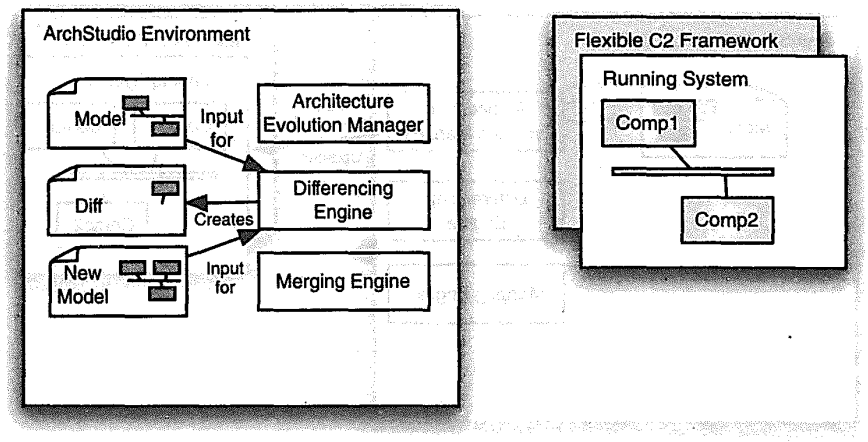



Figure 14-10. *ArchStudio adaptation step 2: differencing.*

within ArchStudio). AEM then makes the corresponding changes to the running architecture by making more calls to the underlying Flexible C2 Framework, instantiating new elements and changing the application as necessary. Although only additions are shown in these simple diagrams, ArchStudio supports additions, modifications, and removals of elements.

### 14.3.2 Architectures/Styles that Support Adaptation

The preceding sections focused on generic techniques that support parts of the conceptual framework for architecture-based adaptation presented in Figure 14-6. The difficulty of making changes—even using these techniques—will vary enormously from one application

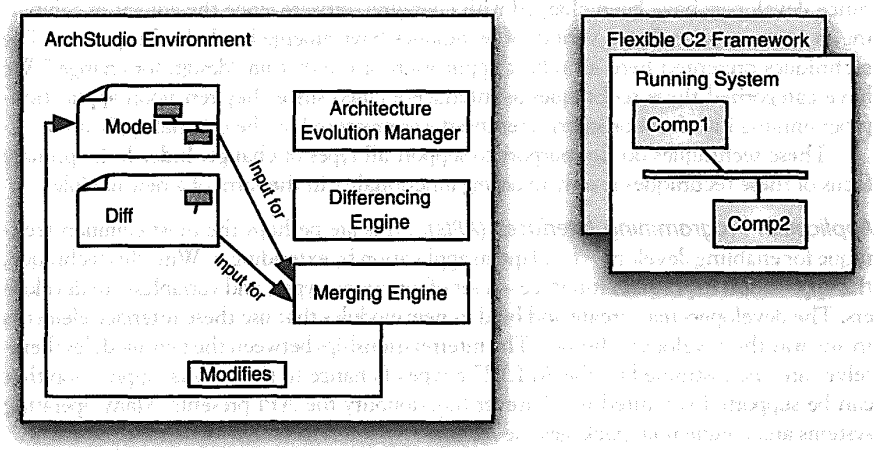



Figure 14-11. *ArchStudio adaptation step 3: merging.*

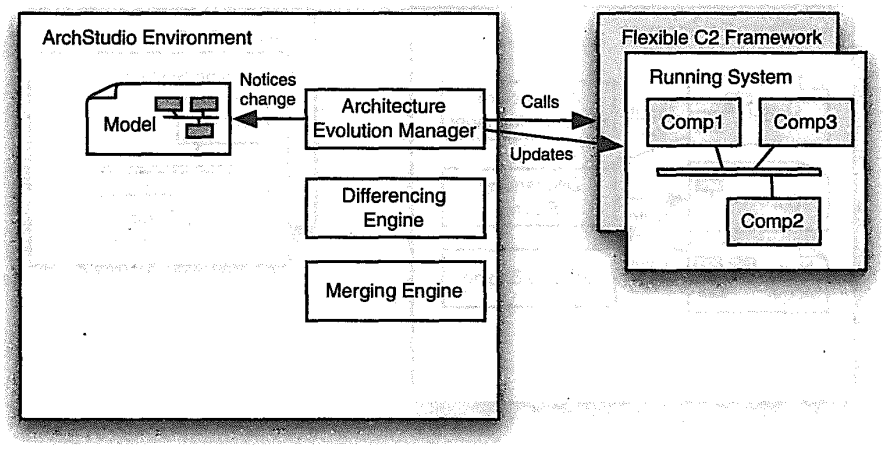



Figure 14-12. *ArchStudio adaptation step 4: applying changes.*

to another, depending on the type of change required and the nature of the architecture to be changed. The discussion of shearing layers early in this chapter indicated, in generic terms, why some changes are more difficult to effect than others. The discussion of the many architectural styles in Chapter 4 showed in particular how some styles are more adept at handling particular types of change than others. Client-server architectures, for example, are designed to easily accommodate the addition or deletion of clients.

In the subsections below we revisit the topic of styles, introducing some specific interface-focused architectural approaches to facilitating change, and then briefly reconsider some concepts from Chapter 4 that are particularly effective in supporting change.

### Interface-Focused Architectural Solutions

Since developers have been charged with changing software since the advent of computing, it is no surprise that a variety of techniques have emerged to help this process. The techniques presented here all reflect application of the dictum "design for change." We have categorized these techniques as interface-focused since they rest upon application-programming interfaces or interpreter interfaces presented by the original application.

These techniques do not purport to support all types of change. Indeed, the primary focus of these techniques is only in *adding* functionality in the form of a new module.

***Application Programming Interfaces (APIs).*** APIs are perhaps the most common technique for enabling developers to adapt an application by extending it. With this technique, the application exposes an interface—a set of functions, types, and variables—to developers. The developers may create and bind in new modules that use these interface elements in any way the developers choose. (The interrelationships between the new modules themselves are unconstrained by the API.) The type of change to the original application that can be supported is limited to whatever functionality the API presents. Many operating systems and commercial packages use APIs.

Sony Exhibit 1018 Page 571
Sony v. Quantum IPR2023-00954

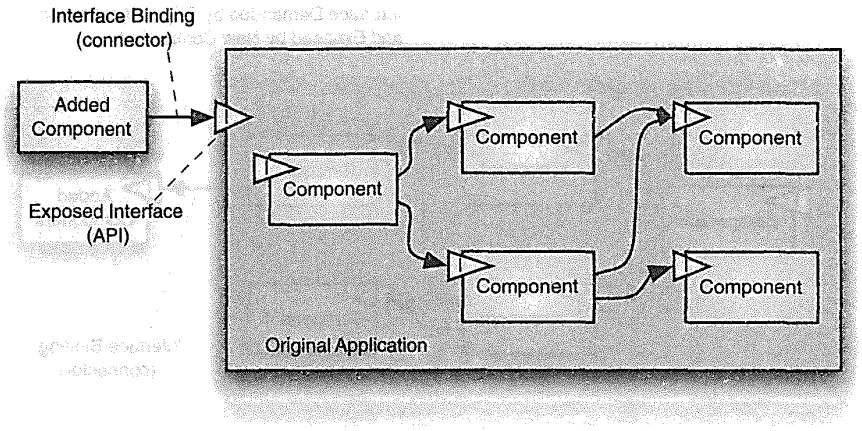



Figure 14-13. *Using an API to extend an application with a new module.*

This approach is illustrated in Figure 14-13, which presents a notional picture of an application to which one new component has been added. The added component can make use of any of the features of the original application that are exposed by the API, but it cannot access the interfaces of any of the components internal to the application, nor can it modify any of the connections internal to the application. The added component makes calls to the original application; nothing in the original application can make calls to the added component.

***Plug-Ins.*** Plug-ins are sort of a mirror-image of APIs: Instead of the added component calling the original application, the application calls out to the added component. To achieve this, the original application predefines an interface that third-party add-ons or plug-ins must implement. The application uses and invokes the plug-in's interface, thereby altering its own behavior. To initiate the process, the original application must be made aware of the existence of the plug-ins; they must become registered with the application. Typically, the registration happens on application start-up, when the application inspects predefined file directories. Components found in those directories that meet the interface specification are registered and may then be called. The technique is illustrated in Figure 14-14.

Adobe Acrobat and Adobe Photoshop are applications that historically have made significant use of the plug-in mechanism. Direct interactions between multiple plug-ins are possible, but such interaction is not part of the plug-in architecture as such.

***"Component/object Architectures".*** With this technique, the host application exposes its internal entities and their interfaces to third-party developers so that they can be used during the adaptation process. This is in contrast, for example, with the API and plug-in approaches that view the original application as a monolithic entity whose internal structure is opaque and immutable. Here, add-ons alter the behavior of the host application by adding new components that interact with existing components. Exposing the internal entities also allows a component to be replaced by one that exposes a compatible interface. This approach is illustrated in Figure 14-15.

Sony Exhibit 1018 Page 572
Sony v. Quantum IPR2023-00954

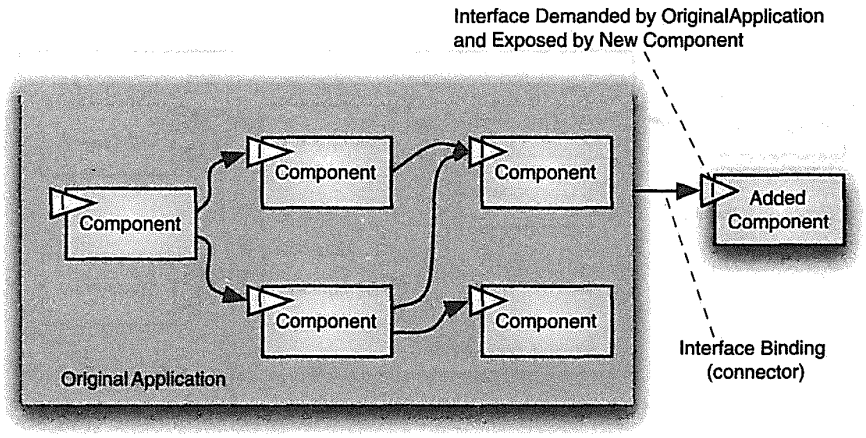



Figure 14-14. *Using a plug-in interface to extend an application.*

This approach to adaptation is more powerful than either APIs or plug-ins, since more interfaces are exposed, and the approach seems architecture-centric because the key items in the vocabulary are a system's components. Typically missing in common application of this technique, however, is an explicit architectural model. The developer is (just) confronted with the system's source code; nothing at a higher level of granularity can be manipulated to achieve the desired changes.

CORBA-based systems, discussed in Chapter 4, are members of this category, as are systems built with Microsoft's Component Object Model (COM). Such systems can be very fluid, but management of them can be difficult because there is no explicit model capable of serving as the fundamental abstraction.

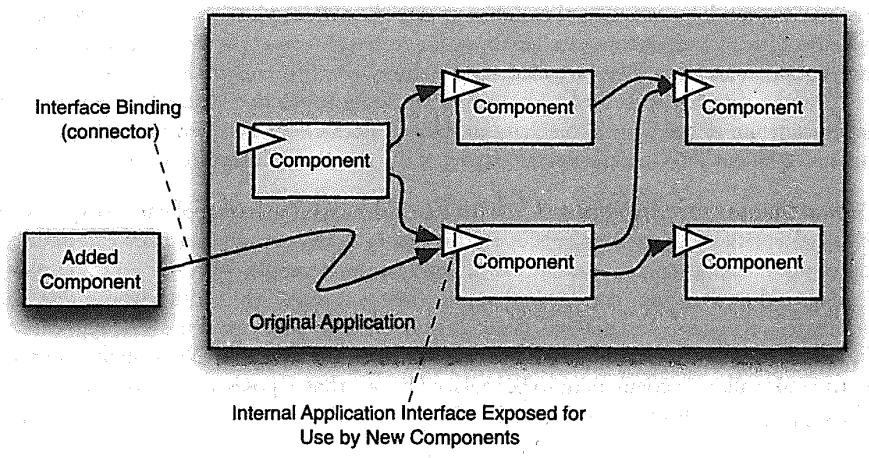



Figure 14-15. *Application modification via the component/object architecture approach.*

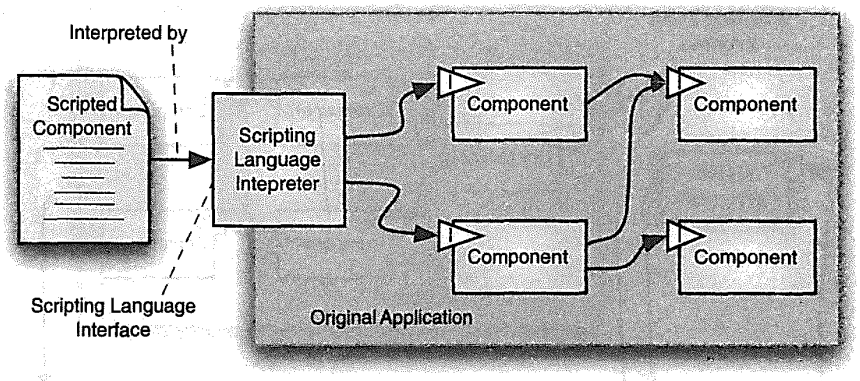



Figure 14-16. *Application adaptation by means of a scripting language.*

***Scripting Languages.*** With this approach, the application provides its own programming language (the "scripting language") and run time environment that the architect uses to implement add-ons. In essence, this is a use of the interpreter style from Chapter 4. Add-ons, when executed by the interpreter, alter the behavior of the application. Scripting languages commonly provide domain-specific language constructs and built-in functions that facilitate the implementation of add-ons, especially for users who lack programming expertise. Spreadsheet formula languages, macro systems, and programming-by-demonstration systems are essentially scripting languages optimized for specialized needs. Microsoft Excel is a prime example of an application that provides this means of extension. The technique is illustrated in Figure 14-16.

***Event Interfaces.*** With this technique—a simple application of the event-based architectural style of Chapter 4—the original application exposes two distinct interfaces to third-party developers. The first is an incoming event interface, specifies the messages it can receive and act on; the second, an outgoing event interface, which specifies the messages it generates. Messages are exchanged via an event mechanism. Add-ons alter the behavior of the application by sending messages to it or by acting on the messages they receive from it. As discussed in Chapter 5, event mechanisms commonly provide a message broadcast facility, which sends a message to every add-on attached to the event mechanism. The event mechanism acts as an intermediary, encapsulating and localizing program binding and communication decisions. As a result, these decisions can be altered independently of the original application or add-on components. This is in sharp contrast to the other techniques, in which programs directly reference, bind to, and use each other's interfaces. The technique is illustrated in Figure 14-17.

### (Strong) Architecture-Based Approaches

Should changes to a system be required that involve more than just adding a new module, a richer approach to adaptation is needed than what the interface-focused solutions above offer. The core of any architecture-based solution, as we have discussed, is an explicit architectural model, faithful to the implementation, which can serve as the basis for reasoning about changes.

Sony Exhibit 1018 Page 574
Sony v. Quantum IPR2023-00954

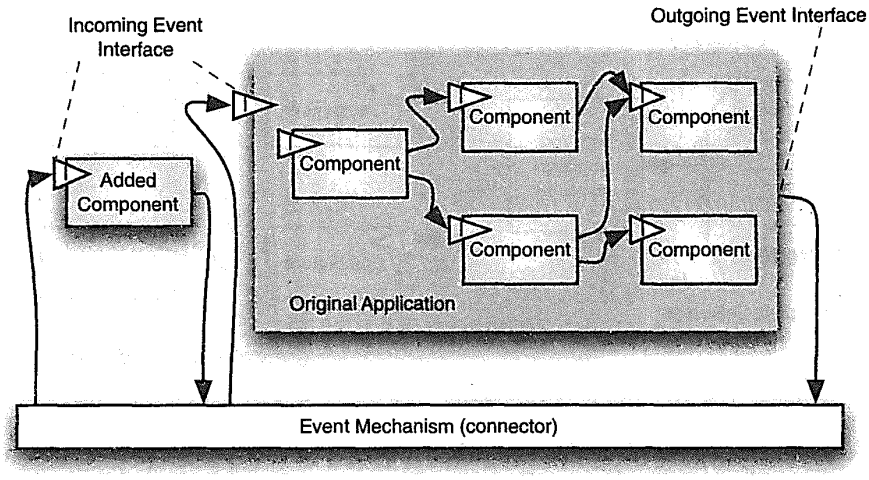



Figure 14-17. *An event architecture used to support extension.*

Simply having an explicit architecture is no panacea, of course. Complex dependencies between architectural elements and inflexible connectors may render difficult any adaptation problem. Consistent use of any of the architectural styles of Chapter 4 can remove some of the inherent difficulties. Ill-conceived custom styles that combine simpler styles in complex ways can vitiate those benefits, however.

The fundamental issue is *bindings*. To the extent that an architectural style enables easy attachment and detachment of components from one another, that style facilitates adaptation. The consistent emphasis of this text on the use of connectors stems from the benefits that explicit connectors can provide in supporting such attachments and detachments. The wide range of connector techniques discussed in Chapter 5 reveals that not all connectors are equally supportive of adaptation, however. Since a direct procedure call is one kind of connector it is obviously possible for the internals of one component to be strongly tied (via a procedure-call connector) to the details of an interface provided by another component. In the extreme (and hence rather useless) case of treating shared memory as a connector, arbitrary interdependencies between components may be achieved. These negative examples do serve to indicate what makes a good connector and what makes a style supportive of adaptation: independence between components. At a minimum, this means lack of dependency on interfaces. (As we shall see later, however, this is not a sufficient condition.) Use of particular types of connectors can achieve this independence, wherein one of the roles of the connector is to facilitate communication between two components while insulating them from dependence.

Perhaps the best example of this kind of connector and the independence provided is the event-based connector discussed in Chapter 4 in conjunction with the implicit invocation architectural styles. Components send events to connectors that route them to other components. The events are capsules of information; the sending component may not know which components receive the sent event; a receiving component may not fully understand all the information in the event, but still can process it such that its own

Sony Exhibit 1018 Page 575
Sony v. Quantum IPR2023-00954