UNIT-I

1. The Architecture Business Cycle

For decades, software designers have been taught to build systems based exclusively on the technical requirements. Conceptually, the requirements document is tossed over the wall into the designer's cubicle, and the designer must come forth with a satisfactory design.
Requirements beget design, which begets system. Of course, modern software development methods recognize the naïveté of this model and provide all sorts of feedback loops from designer to analyst. But they still make the implicit assumption that design is a product of

the system's technical requirements, period.

Architecture has emerged as a crucial part of the design process and is the subject of this book. *Software architecture* encompasses the structures of large software systems. The architectural view of a system is abstract, distilling away details of implementation, algorithm, and data representation and concentrating on the behavior and interaction of "black box" elements. A software architecture is developed as the first step toward designing a system that has a collection of desired properties.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Software architecture is a result of *technical*, *business*, and *social* influences. Its existence in turn affects the technical, business, and social environments that subsequently influence future architectures. We call this cycle of influences, from the environment to the architecture and back to the environment, the*Architecture Business Cycle* (ABC).

This chapter introduces the ABC . The major parts of the book tour the cycle by examining the following:

- How organizational goals influence requirements and developmentstrategy.
- How requirements lead to anarchitecture.
- How architectures are analyzed.
- How architectures yield systems that suggest new organizational capabilities and requirements.

Architecture Process Advice

1. Architecture should be product of a single architect or small group with identifiedleader

2. Architect should have functional requirements and a prioritized list of quality attributes

3. Architecture should be well-documented with at least one static and one dynamicview

4. Architecture should be circulated to stakeholders, who are active in review

5. Architecture should be analyzed (quantitatively and qualitatively) before it is too late.

6. System should be developed incrementally from an initial skeleton that includes major communication paths

7. Architecture should result in a small number of specific resource contentionareas

" Good" Architecture Rules

1. Use information hiding to hide computing infrastructure 2.Each module should protect its secrets with a good interface

3. Use well-known architecture tactics to achieve qualityattributes

4. Minimize and isolate dependence on a particular version of a commercial product or tool. 5.Separate producer modules from consumermodules.

6. For parallel-processing, use well-defined processes ortasks.

7. Assignment of tasks or processes to processors should be easily changeable (even at runtime) 8.Use a small number of simple interactionpatterns

Note: Linda Northrop is a program director at Carnegie Mellon University's Software

Engineering Institute.

If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and maintenance process.

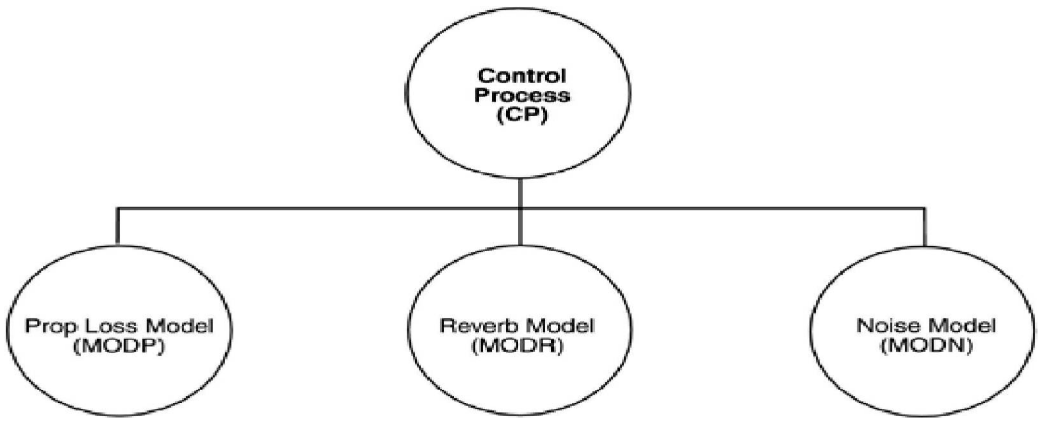
In Chapter 1, we explained that architecture plays a pivotal role in allowing an organization to meet its business goals. Architecture commands a price (the cost of its careful development), but it pays for itself handsomely by enabling the organization to achieve its system goals and expand its software capabilities. Architecture is an asset that holds tangible value to the developing organization beyond the project for which it was created.

In this chapter we will focus on architecture strictly from a software engineering point of view. That is, we will explore the value that a software architecture brings to a development project in addition to the value returned to the enterprise in the ways described in Chapter

What Software Architecture Is and What ItIsn't

Figure 2.1, taken from a system description for an underwater acoustic simulation, purports to describe that system's "top-level architecture" and is precisely the kind of diagram most often displayed to help explain an architecture. Exactly what can we tell from it?

Figure 2.1. Typical, but uninformative, presentation of a software architecture



The system consists of four elements.

- Three of the elements? Prop Loss Model (MODP), Reverb Model (MODR), and Noise Model (MODN)?might have more in common with each other than with the fourth?Control Process (CP)?because they are positioned next to eachother.
- All of the elements apparently have some sort of relationship with each other, since the diagram is fullyconnected.

Is this an architecture? Assuming (as many definitions do) that architecture is a set of components (of which we have four) and connections among them (also present), thisDiagram seems to fill the bill. However, even if we accept the most primitive definition, what can we *not* tell from the diagram?

- *What is the nature of the elements?* What is the significance of their separation? Do they run on separate processors? Do they run at separate times? Do the elements consist of processes, programs, or both? Do they represent ways in which the project labor will be divided, or do they convey a sense of runtime separation? Are they objects, tasks, functions, processes, distributed programs, or somethingelse?
- *What are the responsibilities of the elements?* What is it they do? What is theirfunction in the system?
- *What is the significance of the connections?* Do the connections mean that theelements communicate with each other, control each other, send data to each other, use each other, invoke each other, synchronize with each other, share some information-hiding secret with each other, or some combination of these or other relations? What are the mechanisms for the communication? What information flows across the mechanisms, whatever they may be?
- *What is the significance of the layout?* Why is CP on a separate level? Does it callthe other three elements, and are the others not allowed to call it? Does it contain the other three in an implementation unit sense? Or is there simply no room to put all four elements on the same row in the diagram?

We *must* raise these questions because unless we know precisely what the elements are and how they cooperate to accomplish the purpose of the system, diagrams such as these are not much help and should be regarded skeptically.

This diagram does not show a software architecture, at least not in any useful way. The most charitable thing we can say about such diagrams is that they represent a start. We now define what *does* constitute a software architecture:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

This is a slight change from the first edition. There the primary building blocks were called "components," a term that has since become closely associated with the component- based software engineering movement, taking on a decidedly runtime flavor. "Element" was chosen here to convey something moregeneral.

"Externally visible" properties are those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. Let's look at some of the implications of this definition in more detail.

First, *architecture defines software elements*. The architecture embodies information about how the elements relate to each other. This means that it specifically *omits* certain information about elements that does not pertain to their interaction. Thus, an architecture is foremost an *abstraction* of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements. In nearly all modern systems, elements interact with each other by means of interfaces that partition details

about an element into public and private parts. Architecture is concerned with the public side of this division; private details?those having to do solely with internal implementation?are not architectural.

Second, the definition makes clear that *systems can and do comprise more than one structure* and that no one structure can irrefutably claim to be *the* architecture. For example, all nontrivial projects are partitioned into implementation units; these units are given specific responsibilities and are frequently the basis of work assignments for programming teams. This type of element comprises programs and data that software in other implementation units can call or access, and programs and data that are private. In

large projects, these elements are almost certainly subdivided for assignment to subteams. This is one kind of structure often used to describe a system. It is very static in that it focuses on the way the system's functionality is divided up and assigned to implementation teams.

Other structures are much more focused on the way the elements interact with each other at runtime to carry out the system's function. Suppose the system is to be built as a set of parallel processes. The processes that will exist at runtime, the programs in the various implementation units described previously that are strung together sequentially to form each process, and the synchronization relations among the processes form another kind of structure often used to describe a system.

Are any of these structures alone *the* architecture? No, although they all convey architectural information. The architecture consists of these structures as well as many others. This example shows that since architecture can comprise more than one kind of structure, there is more than one kind of element (e.g., implementation unit and processes), more than one kind of interaction among elements (e.g., subdivision and

Synchronization), and even more than one context (e.g., development time versus runtime). By intention, the definition does not specify what the architectural elements and relationships are. Is a software element an object? A process?A library?A database?A commercial product? It can be any of these things and more.

Third, the definition implies that *every computing system with software has a software architecture* because every system can be shown to comprise elements and the relations among them. In the most trivial case, a system is itself a single element?uninteresting and probably nonuseful but an architecture nevertheless. Even though every system has an architecture, it does not necessarily follow that the architecture is known to anyone. Perhaps all of the people who designed the system are long gone, the documentation has

vanished (or was never produced), the source code has been lost (or was never delivered), and all we have is the executing binary code. This reveals the difference between the *reconstruction* (discussed in Chapter 10).
architecture of a system and the representation of that architecture. Unfortunately, an

Fourth, *the behavior of each element is part of the architecture* insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other, which is clearly part of the architecture. This is

another reason that the box-and-line drawings that are passed off as architectures are not architectures at all. They are simply box-and-line drawings?or, to be more charitable, they serve as cues to provide more information that explains what the elements shown actually do.

Between box-and-line sketches that are the barest of starting points and full-fledged architectures, with all of the appropriate information about a system filled in, lie a host of intermediate stages. Each stage represents the outcome of a set of architectural decisions, the binding of architectural choices. Some of these intermediate stages are very useful in their own right. Before discussing architectural structures, we define three of them.

1. *An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.* A pattern can be thought of as a set of constraints on an architecture?on the element types and their patterns of interaction?and these constraints define a set or family of architectures that satisfy them. For example, client-server is a common architectural pattern. Client andserver are two element types, and their coordination is described in terms of the protocolthat the server uses to communicate with each of its clients. Use of the term *client-server*implies only that multiple clients exist; the clients themselves are not identified, and there is no discussion of what functionality, other than implementation ofthe protocols, has been assigned to any of the clients or to the server. Countless architectures are of the client-server pattern under this (informal) definition, but they are different from each other. An architectural pattern is not an architecture, then, but it still conveys a useful image of the system?it imposes useful constraints on the architecture and, in turn, on the system.

   One of the most useful aspects of patterns is that they exhibit known quality attributes. This is why the architect chooses a particular pattern and not one at random. Some patterns represent known solutions to performance problems, others lend themselves well to high-security systems, still others have been used successfully in high-availability systems. Choosing an architectural pattern is often the architect's first major design choice.

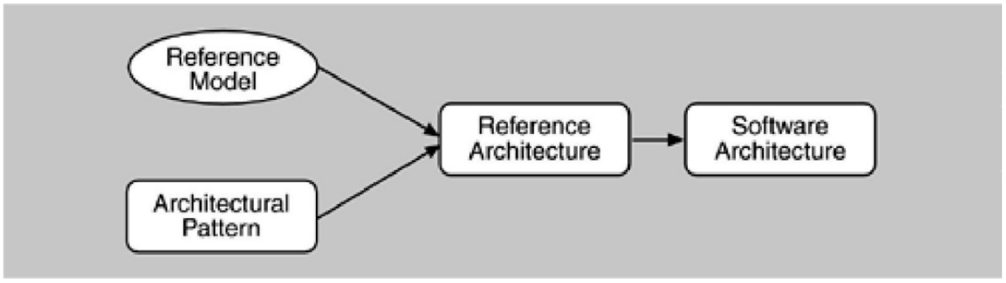
   pieces. A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem. Arising from experience, reference models are a characteristic of mature domains. Can you name the standard parts of a compiler or a database management system? Can you explain in broad terms how the parts work together to accomplish their collective purpose? If so, it is because you have been taught a reference model of these applications.

3. *A reference architecture is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them.* Whereas a reference model divides the functionality, a reference

   Architecture is the mapping of that functionality onto a system decomposition. The mapping may be, but by no means necessarily is, one to one. A software element may implement part of a function or several functions.

Reference models, architectural patterns, and reference architectures are not architectures; they are useful concepts that capture elements of an architure. Each is the outcome of early design decisions. The relationship among these design elements is shown in Figure 2.2.

Figure 2.2.The relationships of reference models, architectural patterns, reference architectures, and software architectures. (The arrows indicate that subsequent concepts contain more design elements.)



People often make analogies to other uses of the word *architecture*, about which they have some intuition. They commonly associate architecture with physical structure (buildings, streets, hardware) and physical arrangement. A building architect must design a building that provides accessibility, aesthetics, light, maintainability, and so on. A software architect must design a system that provides concurrency, portability, modifiability, usability, security, and the like, and that reflects consideration of the tradeoffs among these needs.

Analogies between buildings and software systems should not be taken too far, as they break down fairly quickly. Rather, they help us understand that the viewer's perspective is important and that structure can have different meanings depending on the motivation for examining it. A precise definition of software architecture is not nearly as important as what investigating the concept allows us to do.

development teams?

Architectural Structures and Views

We will be using the related terms *structure* and *view* when discussing architecture representation. A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them. A structure is the set of elements itself, as they exist in software or hardware. For example, a module structure is the set of the system's modules and their organization.

A module view is the representation of that structure, as documented by and used by some system stakeholders. These terms are often used interchangeably, but we will adhere to these definitions.

Architectural structures can by and large be divided into three groups, depending on the broad nature of the elements they show.

- *Module structures.* Here the elements are modules, which are units of implementation. Modules represent a code-based way of considering the system. They are assigned areas of functional responsibility. There is less emphasis on how the resultingsoftware manifests itself at runtime. Module structures allow us to answer questions such as What is the primary functional responsibility assigned to each module? What other software elements is a module allowed to use? What other software does it actually use? What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?
- *Component-and-connector structures.* Here the elements are runtime components (which are the principal units of computation) and connectors (which arethe communication vehicles among components). Component-and-connector structures help answer questions such as What are the major executing components and how do they interact? What are the major shared data stores? Which parts of the system are replicated? How does data progress through the system? What parts of the system can run in parallel? How can the system's structure change as it executes?
- *Allocation structures.* Allocation structures show the relationship between the software elements and the elements in one or more external environments in which the software is created and executed. They answer questions such as What processor does each software

element execute on? In what files is each element stored during development, testing, and system building? What is the assignment of software elementsto

These three structures correspond to the three broad types of decision that architectural design involves:

- How is the system to be structured as a set of code units(modules)?

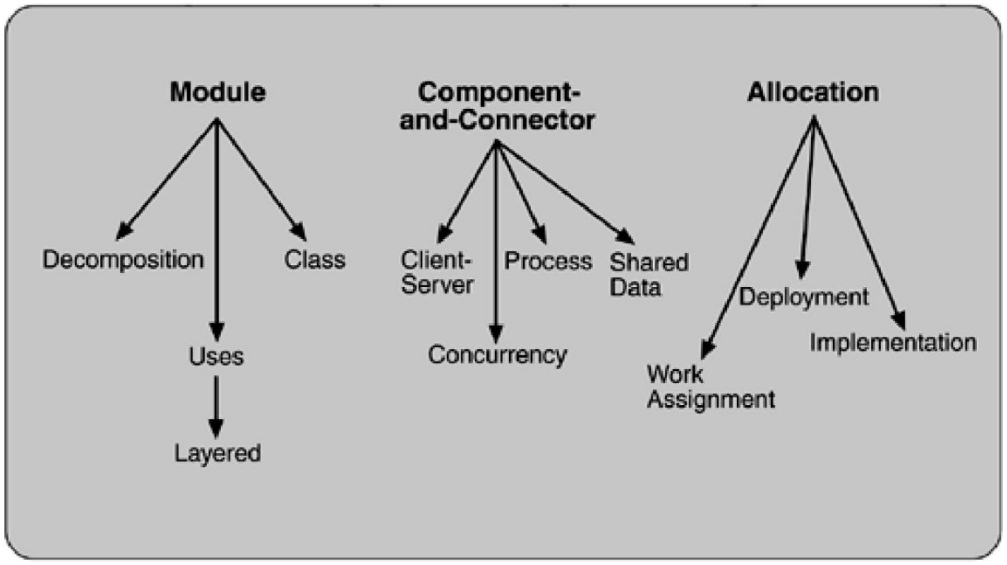
How is the system to be structured as a set of elements that have runtime behavior

- (components) and interactions(connectors)?
- How is the system to relate to nonsoftware structures in its environment (i.e., CPUs, file systems, networks, development teams,etc.)?

SOFTWARE STRUCTURES

Some of the most common and useful software structures are shown in Figure 2.3. These are described in the following sections.

Figure 2-3. Common software architecture structures



Module

Module-based structures include the following.

- *Decomposition.* The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood. Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent(more

  detailed) design and eventual implementation. Modules often have associated products (i.e., interface specifications, code, test plans, etc.). The decomposition structure provides a large part of the system's modifiability, by ensuring that likely changes fall

  within the purview of at most a few small modules. It is often used as the basis for the development project's organization, including the structure of the documentation, and its integration and test plans. The units in this structure often have organization-specific names. Certain U.S. Department of Defense standards, for instance, define Computer Software Configuration Items (CSCIs) and Computer Software Components (CSCs), which are units of modular decomposition. In Chapter 15, we will see system function groups and system functions as the units of decomposition.

- *Uses.* The units of this important but overlooked structure are also modules, or (in circumstances where a finer grain is warranted) procedures or resources on the interfaces of modules. The units are related by the *uses* relation. One unit uses another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second. The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be easily extracted. The ability to easily subset a working system allows for incremental development, a powerful build discipline that will be discussed further in Chapter7.
- *Layered.* When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set ofrelated functionality. In a strictly layered structure, layer *n* may only use the services of layer*n* ? 1. Many variations of this (and a lessening of this structural restriction) occur in practice, however. Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability. We will see layers in the case studies of Chapters 3, 13 and 15.
- *Class*, or *generalization*. The module units in this structure are called classes.The relation is "inherits-from" or "is-an-instance-of." This view supports reasoning about collections of similar behavior or capability (i.e., the classes that other classes inherit from) and parameterized differences which are captured by subclassing. The class structure allows us to reason about re-use and the incremental addition of functionality.

Component-and-Connector

These structures include the following.

- *Process*, or *communicating processes*. Like all component-and-connector structures,this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system. The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations. The relation in this (and in all component-and-connector structures) is *attachment*, showing how the components and connectors are hooked together. The process structure is important in helping to engineer a system's execution performance and availability.
- *Concurrency.* This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are "logical threads." A logical thread is a sequence

of computation that can be allocated to a separate physical thread later inthe design process. The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.

- *Shared data*, or *repository*. This structure comprises components and connectors that create, store, and access persistent data. If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate. It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and dataintegrity.
- *Client-server*. If the system is built as a group of cooperating clients and servers, thisis a good component-and-connector structure to illuminate. The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work. This is useful for separation of concerns (supporting modifiability), for physical distribution, and for load balancing (supporting runtime performance).

Allocation

Allocation structures include the following.

- *Deployment*. The deployment structure shows how software is assigned to hardware-processing and communication elements. The elements are software (usually aprocess from a component-and-connector view), hardware entities (processors), and communication pathways. Relations are "allocated-to," showing on which physical units the software elements reside, and "migrates-to," if the allocation is dynamic. This view allows an engineer to reason about performance, data integrity, availability, and security. It is of particular interest in distributed or parallel systems.
- *Implementation*. This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments. This is critical for the management of development activities and buildprocesses.

Work assignment. This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams. Having a work assignment structure as part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications.WHICH STRUCTURES TO CHOOSE?

There is no shortage of advice. In 1995, Philippe Kruchten [Kruchten 95] published a very influential paper in which he described the concept of architecture comprising separate structures and advised concentrating on four. To validate that the structures were not in conflict with each other and together did in fact describe a system meeting its requirements, Kruchten advised using key use cases as a check. This so-called "Four Plus One" approach became popular and has now been institutionalized as the conceptual basis of the Rational

Unified Process.Kruchten's four views follow:

- *Logical.* The elements are "key abstractions," which are manifested in theobject-oriented world as objects or object classes. This is a module view.
- *Process.* This view addresses concurrency and distribution of functionality. It is a component-and-connectorview.
- *Development.* This view shows the organization of software modules,libraries, subsystems, and units of development. It is an allocation view, mapping software to the development environment.
- *Physical.* This view maps other elements onto processing and communication nodes and is also an allocation view (which others call the deploymentview).

Creating anArchitecture

Quality is often in the eye of the beholder (to paraphrase Booth Tarkington). What this means for the architect is that customers may dislike a design because their concept of quality differs from the architect's. Quality attribute scenarios are the means by which

explore different types of quality that may be appropriate for an architecture. For six important attributes (availability, modifiability, performance, security, testability, and usability), we describe how to generate scenarios that can be used to characterize quality requirements. These scenarios demonstrate what quality means for a particular system, giving both the architect and the customer a basis for judging a design.

Quality Attributes

Architecture and Quality Attributes

Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct. For example:

- Usability involves both architectural and nonarchitectural aspects. Thenonarchitectural aspects include making the user interface clear and easy to use. Should you providea radio button or a check box? What screen layout is most intuitive? What typeface is most clear? Although these details matter tremendously to the end user and influence usability, they are not architectural because they belong to the details of design. Whether a system provides the user with the ability to cancel operations, to undo operations, or to re-use data previously entered is architectural, however. These requirements involve the cooperation of multiple elements.
- Modifiability is determined by how functionality is divided (architectural) and bycoding techniques within a module (nonarchitectural). Thus, a system is modifiable if changes involve the fewest possible number of distinct elements.
- Performance involves both architectural and nonarchitectural dependencies. It depends partially on how much communication is necessary among components (architectural), partially on what functionality has been allocated to each component (architectural), partially on how shared resources are allocated (architectural), partially on the choiceof algorithms to implement selected functionality (nonarchitectural), and partially on how these algorithms are coded (nonarchitectural).

The message of this section is twofold:

1. Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architecturallevel.
2. Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality, but this foundation will be to no avail if attention is not paid to the details.

Let's begin our tour of quality attributes. We will examine the following three classes:

1. Qualities of the system. We will focus on availability, modifiability, performance, security, testability, andusability.
2. Business qualities (such as time to market) that are affected by thearchitecture.
3. Qualities, such as conceptual integrity, that are about the architecture itself although they indirectly affect other qualities, such asmodifiability.

System Quality Attributes

System quality attributes have been of interest to the software community at least since the 1970s. There are a variety of published taxonomies and definitions, and many of them have their own research and practitioner communities. From an architect's perspective, there are three problems with previous discussions of system quality attributes:

- The definitions provided for an attribute are not operational. It is meaningless to say that a system will be modifiable. Every system is modifiable with respect to one set of changes and not modifiable with respect to another. The other attributes aresimilar.
- A focus of discussion is often on which quality a particular aspect belongs to. Is asystem failure an aspect of availability, an aspect of security, or an aspect of usability? All three attribute communities would claim ownership of a system failure.
- Each attribute community has developed its own vocabulary. The performance community has "events" arriving at a system, the security community has "attacks" arriving at a system, the availability community has "failures" of a system, and the usability community has "user input." All of these may actually refer to the same occurrence, but are described using differentterms.

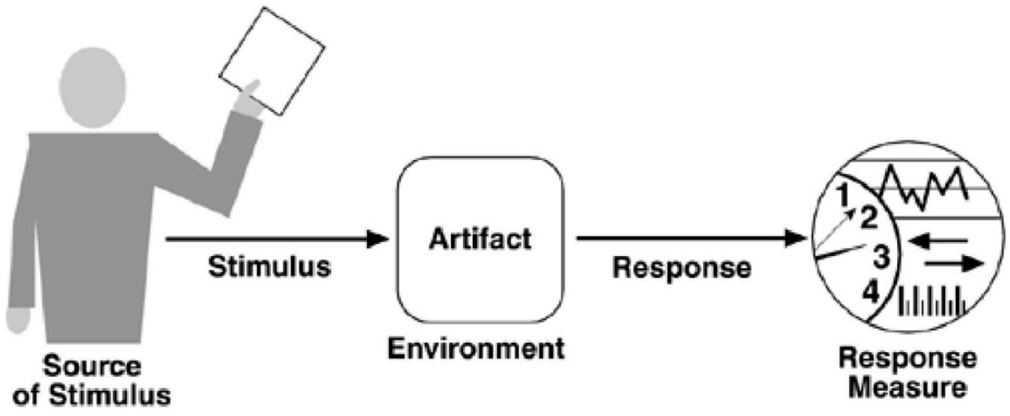
QUALITY ATTRIBUTE SCENARIOS

A quality attribute scenario is a quality-attribute-specific requirement. It consists of six parts.

- *Source of stimulus.* This is some entity (a human, a computer system, or any other actuator) that generated thestimulus.
- *Stimulus.* The stimulus is a condition that needs to be considered when it arrives at a system.

- *Environment.* The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may betrue.
- *Artifact.* Some artifact is stimulated. This may be the whole system or some pieces ofit.
- *Response.* The response is the activity undertaken after the arrival of thestimulus.
- *Response measure.* When the response occurs, it should be measurable in some fashion so that the requirement can betested.

Quality attribute parts



Quality Attribute Scenarios in Practice

General scenarios provide a framework for generating a large number of generic, system-independent, quality-attribute-specific scenarios. Each is potentially but not necessarily relevant to the system you are concerned with. To make the general scenarios useful for a particular system, you must make them system specific.

We now discuss the six most common and important system quality attributes, with the twin goals of identifying the concepts used by the attribute community and providing a way to generate general scenarios for that attribute.

AVAILABILITY

MODIFIABILITY

PERFORMANCESECU

RITYTESTABILITY

USABILITY

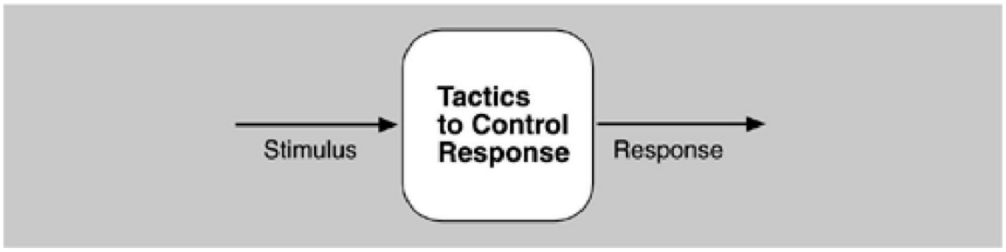
Achieving Qualities

INTRODUCTION TACTICS

A tactic is a design decision that influences the control of a quality attribute response. A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality. The tactics are those that architects have been using for years, and we isolate and describe them. We are not *inventing* tactics here, just capturing

what architects do in practice.

Tactics are intended to control responses to stimuli.



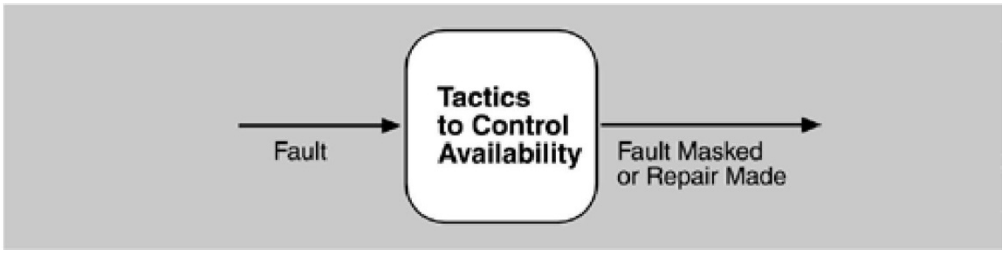
Each tactic is a design option for the architect. For example, one of the tactics introduces redundancy to increase the availability of a system. This is one option the architect has to increase availability, but not the only one. Usually achieving high availability through redundancy implies a concomitant need for synchronization (to ensure that the redundant copy can be used if the original fails). We see two immediate ramifications of this example.

1. *Tactics can refine other tactics.*

2. *Patterns packagetactics.*

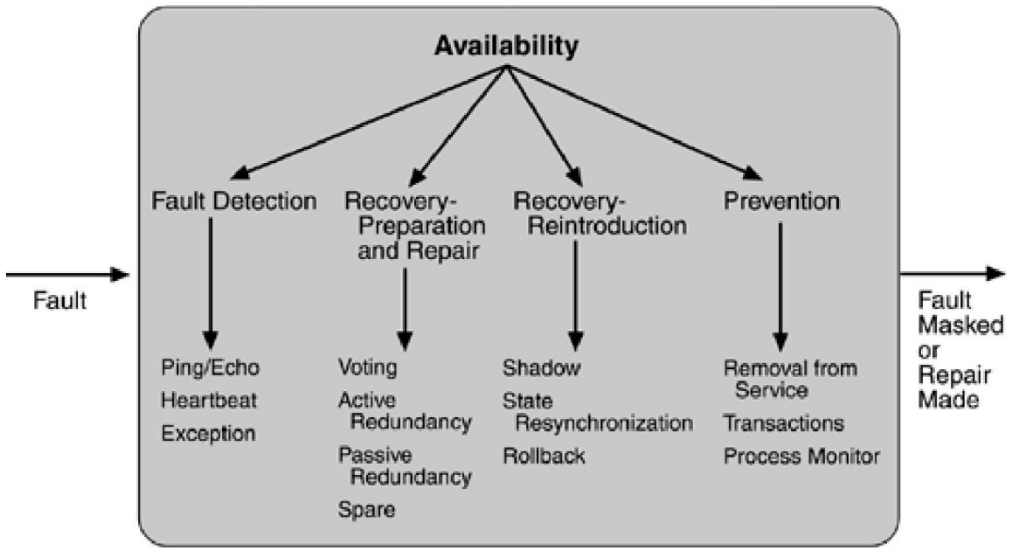
Availability Tactics

Goal of availability tactics



Many of the tactics we discuss are available within standard execution environments such as operating systems, application servers, and database management systems. It is still

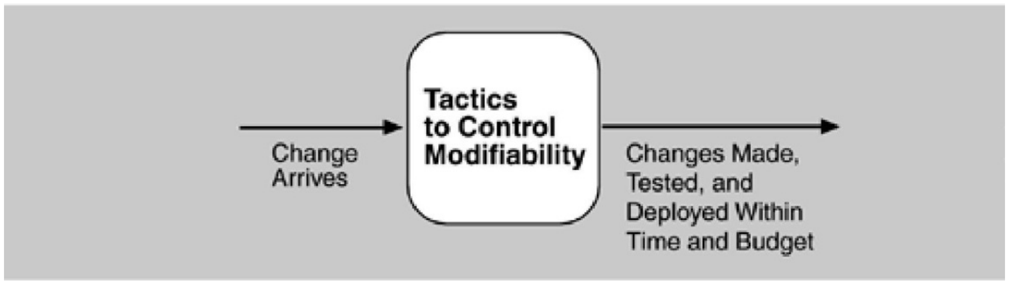
important to understand the tactics used so that the effects of using a particular one can be considered during design and evaluation. All approaches to maintaining availability involve some type of redundancy, some type of health monitoring to detect a failure, and some type of recovery when a failure is detected.Summary of availability tactics

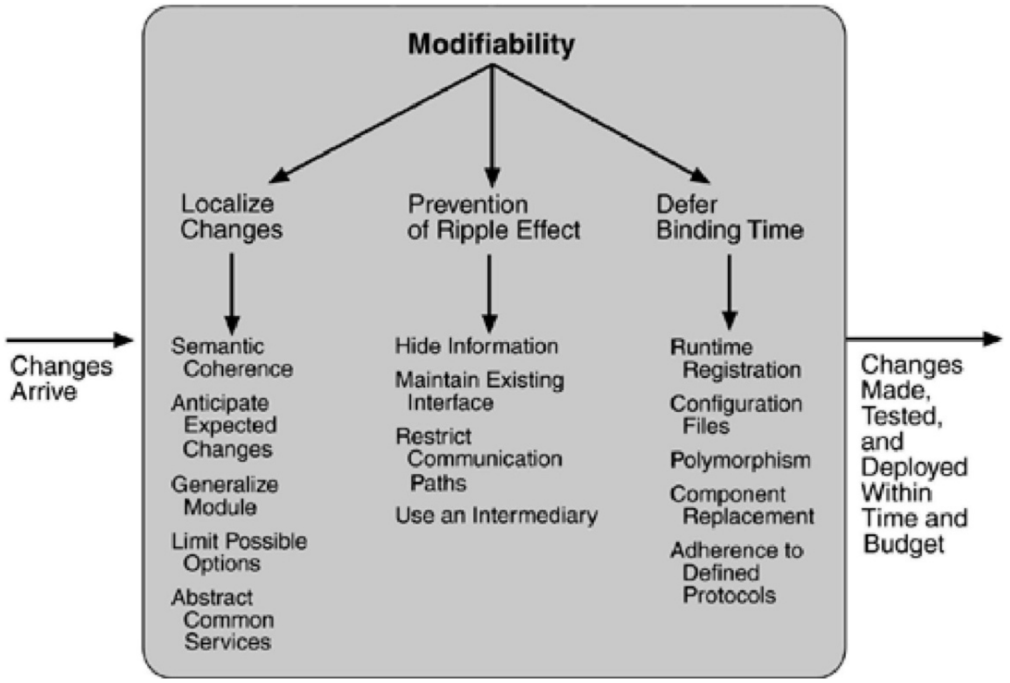


Modifiability Tactics

We organize the tactics for modifiability in sets according to their goals. One set has as its goal reducing the number of modules that are directly affected by a change. We call this set "localize modifications." A second set has as its goal limiting modifications to the localized modules. We use this set of tactics to "prevent the ripple effect." Implicit in this distinction is that there are modules directly affected (those whose responsibilities are adjusted to accomplish the change) and modules indirectly affected by a change (those whose responsibilities remain unchanged but whose implementation must be changed to accommodate the directly affected modules). A third set of tactics has as its goal controlling deployment time and cost. We call this set "defer bindingtime."

Goal of modifiability tactics

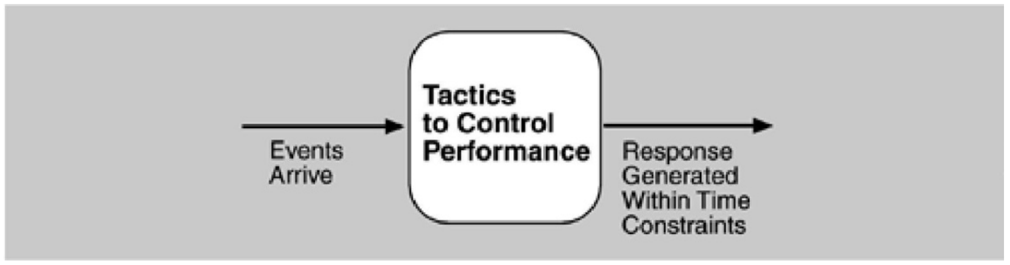

Change Arrives
Tactics to Control Modifiability
Changes Made, Tested, and Deployed Within Time and Budget



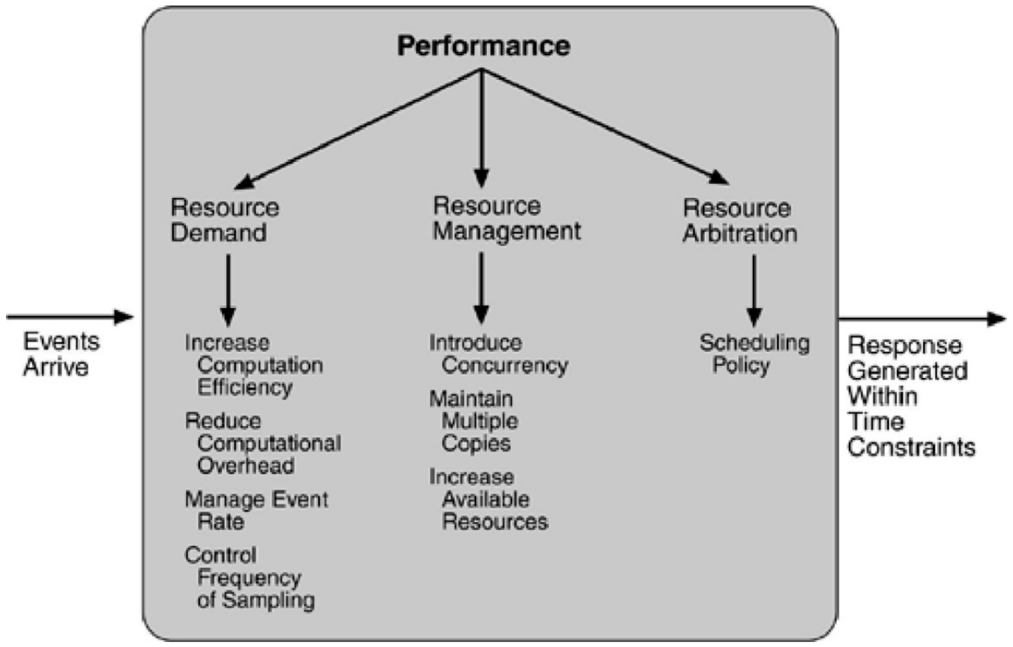
Performance Tactics

the goal of performance tactics is to generate a response to an event arriving at the system within some time constraint. The event can be single or a stream and is the trigger for a request to perform computation. It can be the arrival of a message, the expiration of a time interval, the detection of a significant change of state in the system's environment, and so forth. The system processes the events and generates a response. Performance tactics control the time within which a response is generated. This is shown in Figure 5.6. Latency is the time between the arrival of an event and the generation of a response to it.

Goal of performance tactics


Events Arrive
Tactics to Control Performance
Response Generated Within Time Constraints

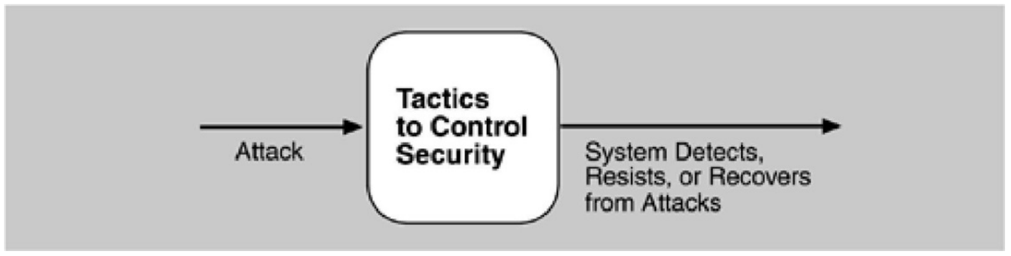
Summary of performance tactics


Performance
Resource Demand
Resource Management
Resource Arbitration
Events Arrive
Increase Computation Efficiency
Reduce Computational Overhead
Manage Event Rate
Control Frequency of Sampling
Introduce Concurrency
Maintain Multiple Copies
Increase Available Resources
Scheduling Policy
Response Generated Within Time Constraints

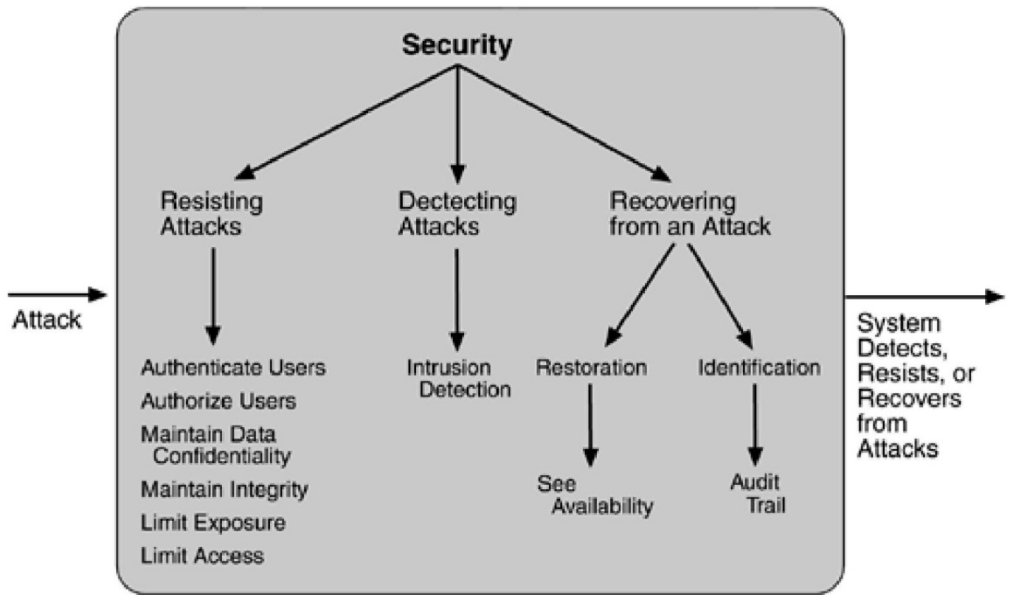
Security Tactics

Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks. All three categories are important. Using a familiar analogy, putting a lock on your door is a form of resisting an attack, having a motion sensor inside of your house is a form of detecting an attack, and having insurance is a form of recovering from an attack. Figure 5.8 shows the goals of the security tactics.

. Goal of security tactics


Attack
Tactics to Control Security
System Detects, Resists, or Recovers from Attacks

Summary of tactics forsecurity

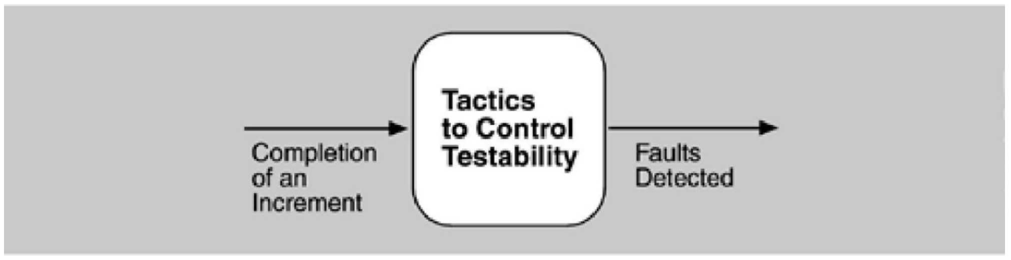


TestabilityTactics

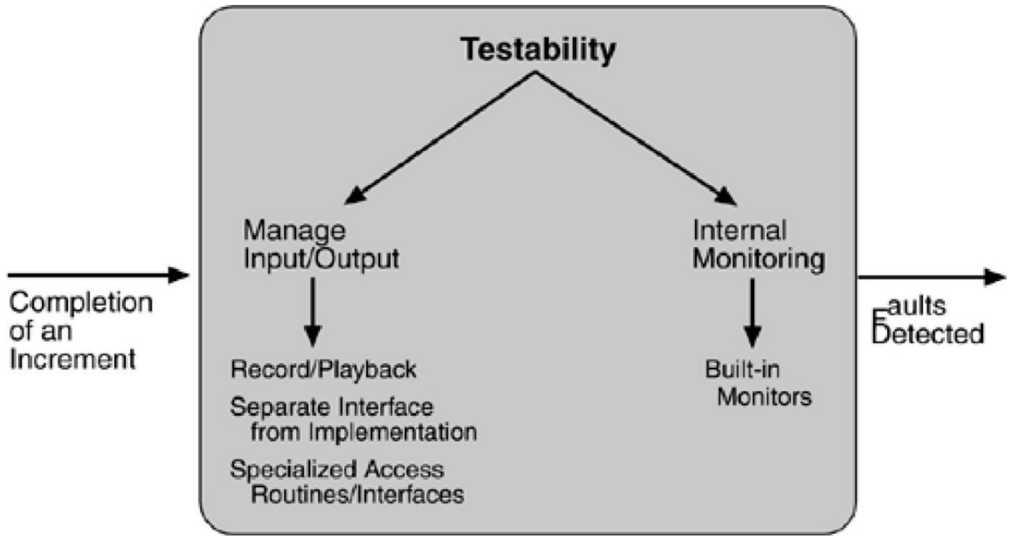
The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. Figure 5.10 displays the use of tactics for testability.

Architectural techniques for enhancing the software testability have not received as much attention as more mature fields such as modifiability, performance, and availability, but, as we stated in Chapter 4, since testing consumes such a high percentage of system development cost, anything the architect can do to reduce this cost will yield a significant benefit.

Goal of testability tactics



Summary of testability tactics

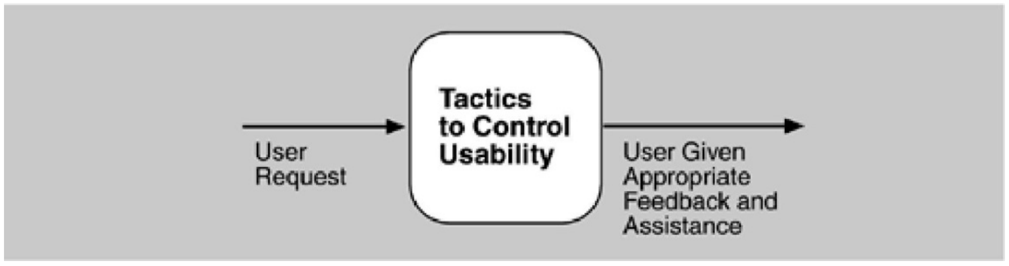

Testability
Completion of an Increment
Manage Input/Output
Record/Playback
Separate Interface from Implementation
Specialized Access Routines/Interfaces
Internal Monitoring
Built-in Monitors
Faults Detected

Usability Tactics

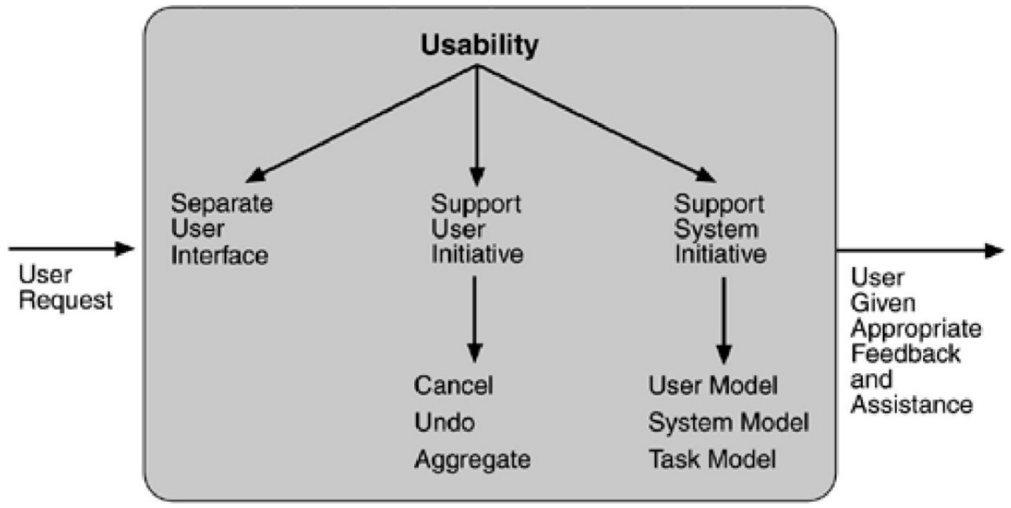
usability is concerned with how easy it is for the user to accomplish a desired task and the kind of support the system provides to the user. Two types of tactics support usability, each intended for two categories of "users." The first category, runtime, includes those that support the user during system execution. The second category is based on the iterative nature of user interface design and supports the interface developer at design time. It is

strongly related to the modifiability tactics already presented.

Goal of runtime usability tactics


User Request
Tactics to Control Usability
User Given Appropriate Feedback and Assistance

Summary of runtime usability tactics



Architectural Patterns and Styles

An architectural pattern in software, also known as an architectural style, is analogous to an architectural style in buildings, such as Gothic or Greek Revival or Queen Anne. It consists of a few key features and rules for combining them so that architectural integrityis preserved. An architectural pattern is determined by:

- A set of element types (such as a data repository or a component that computesa

  mathematical function).
- A topological layout of the elements indicating theirinterrelation-ships.
- A set of semantic constraints (e.g., filters in a pipe-and-filter style are pure datatransducers?they incrementally transform their input stream into an output stream,but do not control either upstream or downstream elements).
- A set of interaction mechanisms (e.g., subroutine call, event-subscriber,blackboard)

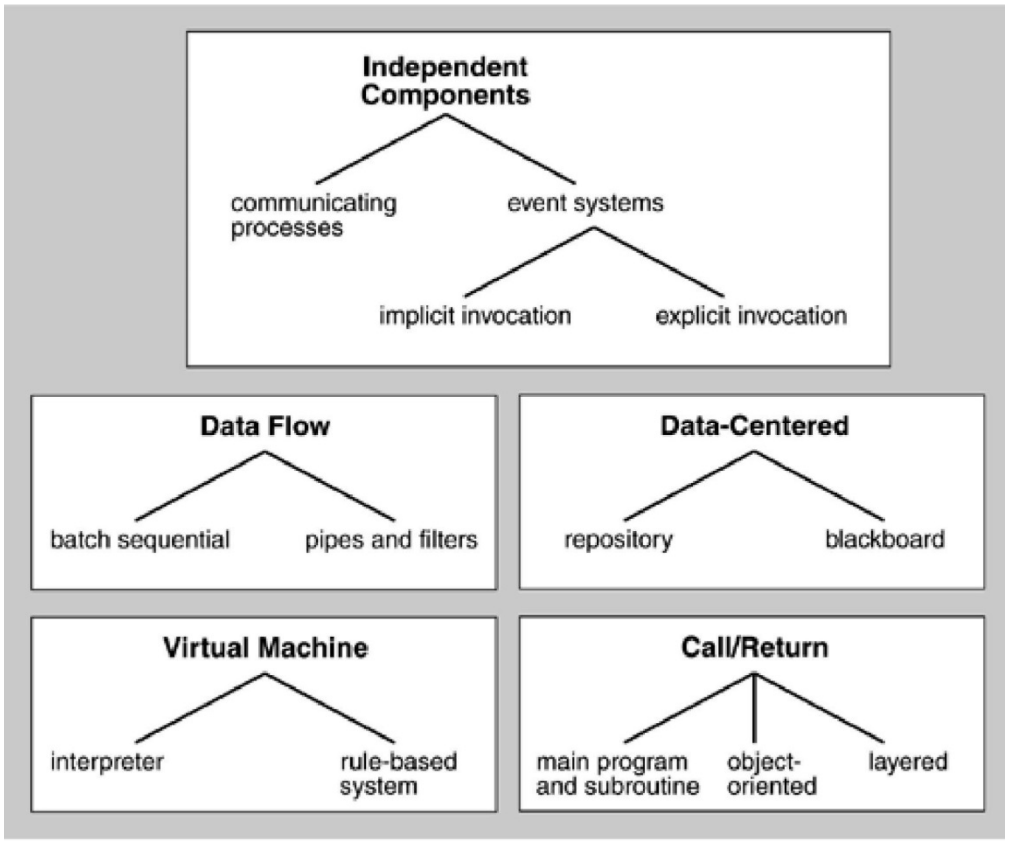
  that determine how the elements coordinate through the allowed topology.

Mary Shaw and David Garlan's influential work attempted to catalog a set of architectural patterns that they called architectural *styles* or *idioms*. This has been evolved by the software engineering community into what is now more commonly known as architectural patterns, analogous to design patterns and code patterns.

The motivation of [Shaw 96] for embarking on this project was the observation that high-level abstractions for complex systems exist but we do not study or catalog them, as is common in other engineering disciplines.

These patterns occur not only regularly in system designs but in ways that sometimes prevent us from recognizing them, because in different disciplines the same architectural pattern may be called different things. In response, a number of recurring architectural patterns, their properties, and their benefits have been cataloged. One such catalog is illustrated in Figure.

A small catalog of architectural patterns, organized by is-a relations



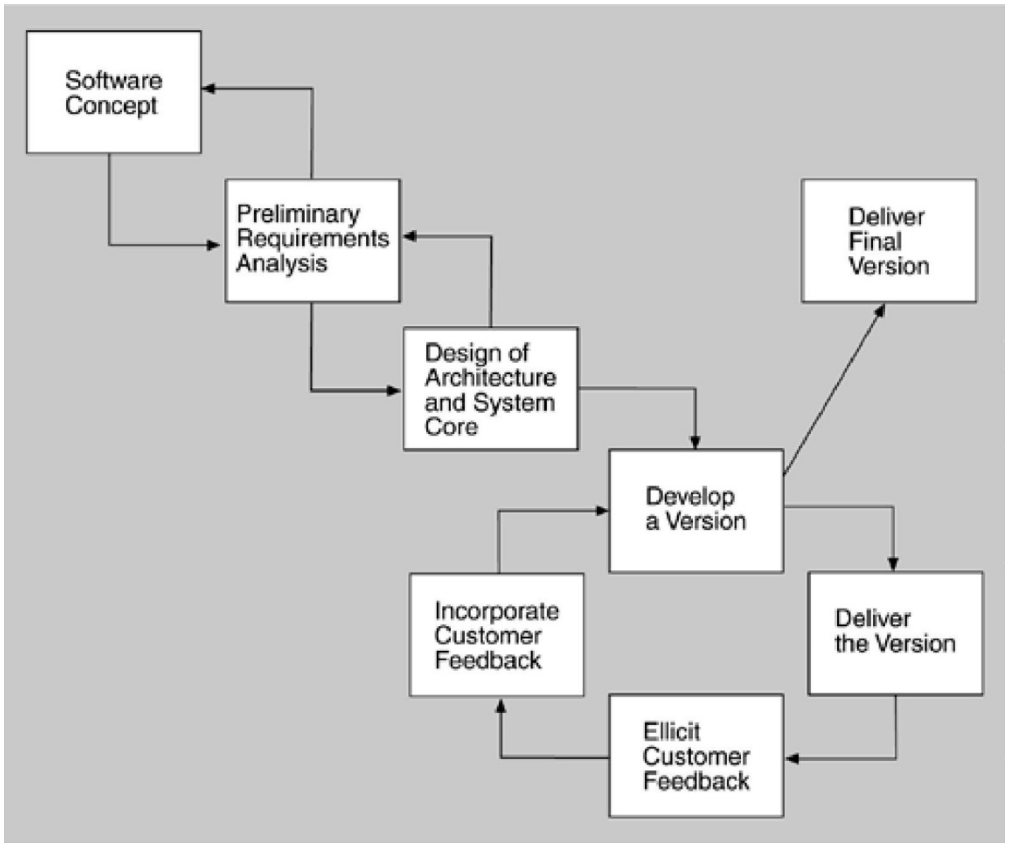
In this figure patterns are categorized into related groups in an inheritance hierarchy. For example, an event system is a substyle of independent elements. Event systems themselves have two subpatterns: implicit invocation and explicit invocation.

Designing the Architecture

We have observed two traits common to virtually all of the successful object-oriented systems we have encountered, and noticeably absent from the ones that we count as failures: the existence of a strong architectural vision and the application of a well-managed iterative and incremental development cycle.

- Architecture in the lifecycle
- Designing thearchitecture
- Forming the team structure and its relationship to thearchitecture
- Creating a skeletal system

Evolutionary Delivery Life Cycle


Software Concept
Preliminary Requirements Analysis
Design of Architecture and System Core
Develop a Version
Deliver Final Version
Deliver the Version
Ellicit Customer Feedback
Incorporate Customer Feedback

Designing an architecture to satisfy both quality requirementsand functional requirements. We call this method Attribute-Driven Design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about therelationbetweenqualityattributeachievementandarchitectureinorderto designthe architecture. The ADD method can be viewed as an extension to most other development methods, such as the Rational Unified Process. The Rational Unified Process has several steps that result in the high-level design of an architecture but then proceeds to detailed design andimplementation.
Incorporating ADD into it involves modifying the steps dealing with the high-level design of the architecture and then followingthe process as described by Rational.

ATTRIBUTE-DRIVEN DESIGN

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern. ADD is positioned in the life cycle after requirements analysis and, as we have said, can begin when the architectural drivers are known with some confidence.

The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate. Not all details of the views result from an application of ADD; the system is described as a set of containers for functionality and the interactions among them. This is the first articulation of architecture during the design process and is therefore necessarily coarse grained. Nevertheless, it is critical for achieving the desired qualities, and it provides a framework for achieving the functionality. The difference between an architecture resulting from ADD and one ready for implementation rests in the more detailed design decisions that need to be made. These could be, for example, the decision to use specific object-oriented design patterns or a specific piece of middleware that brings with it many architectural constraints. The architecture designed by

ADD may have intentionally deferred this decision to be more flexible.

Represent the architecture with views

We now briefly discuss how ADD uses these three common views.

*Module decomposition view.*Our discussion above shows how the module decomposition view provides containers for holding responsibilities as they are discovered. Major data flow relationships among the modules are also identified through this view.

*Concurrency view.*In the concurrency view dynamic aspects of a system such as parallel activities and synchronization can be modeled. This modeling helps to identify resource contention problems, possible deadlock situations, data consistency issues, and so forth. Modeling the concurrency in a system likely leads to discovery of new responsibilities of the modules, which are recorded in the module view. It can also lead to discovery of new modules, such as a resource manager, in order to solve issues of concurrent access to a scarce resource and the like.

- To understand the concurrency in a system, the following use cases areilluminating:
- - *Two users doing similar things at the same time.* This helps in recognizing resource contention or data integrity problems. In our garage door example, one user may be closing the door remotely while another is opening the door from aswitch.
- - *One user performing multiple activities simultaneously*. This helps to uncover data exchange and activity control problems. In our example, a user may be performing diagnostics while simultaneously opening thedoor.
- - *Starting up the system.* This gives a good overview of permanent running activities in the system and how to initialize them. It also helps in deciding on an initialization strategy, such as everything in parallel or everything in sequence or any other model. In our example, does the startup of the garage door opener system depend on the availability of the home information system? Is the garage door opener system always working, waiting for a signal, or is it started and stopped with every door opening andclosing?
- - *Shutting down the system.* This helps to uncover issues of cleaning up, such as achieving and saving a consistent systemstate.

*Deployment view.*If multiple processors or specialized hardware is used in a system, additional responsibilities may arise from deployment to the hardware. Using a deployment view helps to determine and design a deployment that supports achieving the desired qualities. The deployment view results in the virtual threads of the concurrency view being decomposed into virtual threads within a particular processor and messages that travel between processors to initiate the next entry in the sequence of actions. Thus, it is the basis for analyzing the network traffic and for determining potential congestion.

Documenting Software Architectures

The software architecture for a system plays a central role in system development and in the organization that produces it. The architecture serves as the blueprint for both the system and the project developing it. It defines the work assignments that must be carried out by design and implementation teams and it is the primary carrier of system qualities such as performance, modifiability, and security?none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure thatthe design approach will yield an acceptable system. Moreover, architecture holds the key to post-deployment system understanding, maintenance, and mining efforts. In short, architecture is the conceptual glue that holds every phase of the project together for all of its manystakeholders.

Documenting the architecture is the crowning step to crafting it. Even a perfect architecture is useless if no one understands it or (perhaps worse) if key stakeholders misunderstand it. If you go to the trouble of creating a strong architecture, you *must* describe it in sufficent detail, without ambiguity, and organized in such a way that others can quickly find needed information. Otherwise, your effort will have been wasted because the architecture will be unusable.

Uses of Architectural Documentation

Stakeholder	Use
Architect and requirements engineers who represent customer(s)	To negotiate and make tradeoffs among competing requirements
Architect and designers of constituent parts	To resolve resource contention and establish performance and other kinds of runtime resource consumption budgets
Implementors	To provide inviolable constraints (plus exploitable freedoms) on downstream development activities
Testers and integrators	To specify the correct black-box behavior of the pieces that must fit together

Maintainers	To reveal areas a prospective change willaffect
Designers of other	To define the set of operations provided and required, andthe
Stakeholder	Use
systems with which this one must interoperate	protocols for their operation
Quality attribute specialists	To provide the model that drives analytical tools such as rate-monotonic real-time schedulability analysis, simulations and simulation generators, theorem provers, verifiers, etc. These tools require information about resource consumption, scheduling policies, dependencies, and so forth. Architecture documentation must contain the information necessary to evaluate a variety of quality attributes such as security, performance, usability, availability, and modifiability. Analyses for each attributes have their own information needs.

Managers	To create development teams corresponding to work assignments identified, to plan and allocate project resources, and to track progress by the various teams

Views

This principle is useful because it breaks the problem of architecture documentation into more tractable parts, which provide the structure for the remainder of this chapter:

- Choosing the relevantviews
- Documenting aview
- Documenting information that applies to more than oneview

Documentation across Views

1. *How* the documentation is laid out and organized so that a stakeholder of the architecture can find the information he or she needs efficiently and reliably. This part consists of a view catalog and a viewtemplate.
2. *What* the architecture is. Here, the information that remains to be captured beyond the views themselves is a short system overview to ground any reader as to the purpose of the system; the way the views are related to each other; a list of elements andwhere they appear; and a glossary that applies to the entire architecture.
3. *Why* the architecture is the way it is: the context for the system, external constraints that have been imposed to shape the architecture in certain ways, and the rationale for coarse-grained large-scaledecisions.

Reconstructing Software Architectures

Architecture reconstruction is an interpretive, interactive, and iterative process involving many activities; it is not automatic. It requires the skills and attention of both the reverse engineering expert and the architect (or someone who has substantial knowledge of the architecture), largely because architectural constructs are not represented explicitly in the source code. There is no programming language construct for "layer" or "connector" or other architectural elements that we can easily pick out of a source code file. Architectural patterns, if used, are seldom labeled. Instead, architectural constructs are realized by many diverse mechanisms in an implementation, usually a collection of functions, classes, files, objects, and so forth. When a system is initially developed, its high-level design/architectural elements are mapped to implementation elements. Therefore, when we reconstruct those elements, we need to apply the inverses of the mappings. Coming up with those requires architectural insight. Familiarity with compiler construction techniques and utilities such as grep, sed, awk, perl, python, and lex/yacc is alsoimportant.

THE WORKBENCH APPROACH

Architecture reconstruction requires tool support, but no single tool or tool set is always adequate to carry it out. For one thing, tools tend to be language-specific and we may encounter any number of languages in the artifacts we examine. A mature MRI scanner, for example, can contain software written in 15 languages. For another thing, data extraction tools are imperfect; they often return incomplete results or false positives, and so we use aselection of tools to augment and check on each other. Finally, the goals of reconstruction vary, as discussed above. What you wish to do with the recovered documentation will determine what information you need to extract, which in turn will suggest different tools.

Taken together, these have led to a particular design philosophy for a tool set to support architecture reconstruction known as the *workbench*. A *workbench* should be open (easy to integrate new tools as required) and provide a lightweight integration framework whereby tools added to the tool set do not affect the existing tools or data unnecessarily.

RECONSTRUCTION ACTIVITIES

Software architecture reconstruction comprises the following activities, carried out iteratively:

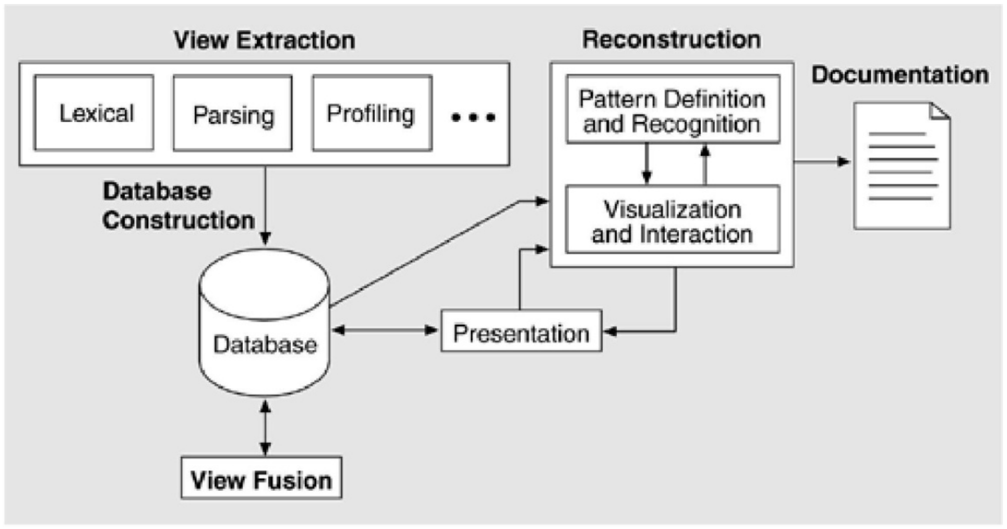
1. *Information extraction*. The purpose of this activity is to extract information from various

sources.

2. *Database construction.* Database construction involves converting this information into a standard form such as the Rigi Standard Form (a tuple-based data format in theform ofrelationship <entity1><entity2>) and an SQL-based database format from which the database is created.
3. *View fusion.* View fusion combines information in the database to produce a coherent view of thearchitecture.
4. *Reconstruction.* The reconstruction activity is where the main work of building abstractions and various representations of the data to generate an architecture representation takesplace.

As you might expect, the activities are highly iterative. Figure 10.1 depicts the architecture reconstruction activities and how information flows among them.

Architecture reconstruction activities. (The arrows show how information flows among the activities.)



Reconstruction consists of two primary activities: *visualization and interaction* and*pattern definition and recognition*. Each is discussed next.

Visualization and interaction provides a mechanism by which the user may interactively visualize, explore, and manipulate views. In Dali, views are presented to the user as a hierarchically decomposed graph of elements and relations, using the Rigi tool.

Pattern definition and recognition provides facilities for architectural reconstruction: the definition and recognition of the code manifestation of architectural patterns. Dali's reconstruction facilities, for example, allow a user to construct more abstract views of a software system from more detailed views by identifying aggregations of elements. Patterns are defined in Dali, using a combination of SQL and perl, which we call *code segments*. An SQL query is used to identify elements from the Dali repository that will contribute to a new aggregation, and perl expressions are used to transform names and perform other manipulations of the query results. Code segments are retained, and users can selectively apply and re-use them.