UNIT-I
ENVISIONING ARCHITECTURE

CHAPTER 1

WHAT IS SOFTWARE ARCHITECTURE :

Software Architecture Definition:

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.

Let's look at some of the implications of this definition in more detail.

- Architecture defines software elements
- 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.
- 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.
- 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.

OTHER POINTS OF VIEW

The study of software architecture is an attempt to abstract the commonalities inherent in system design, and as such it must account for a wide range of activities, concepts, methods, approaches, and results.

- ***Architecture is high-level design.*** Other tasks associated with design are not architectural, such as deciding on important data structures that will be encapsulated.
- ***Architecture is the overall structure of the system.*** The different structures provide the critical engineering leverage points to imbue a system with the quality attributes that will render it a success or failure. The multiplicity of structures in an architecture lies at the heart

of the concept.

- ***Architecture is the structure of the components of a program or system, their interrelationships, and the principles and guidelines governing their design and evolution over time.*** Any system has an architecture that can be discovered and analyzed independently of any knowledge of the process by which the architecture was designed or evolved.
- ***Architecture is components and connectors.*** Connectors imply a runtime mechanism for transferring control and data around a system. When we speak of "relationships" among elements, we intend to capture both runtime and non-runtime relationships.

ARCHITECTURAL PATTERNS, REFERENCE MODELS & REFERENCE ARCHITECTURES

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.

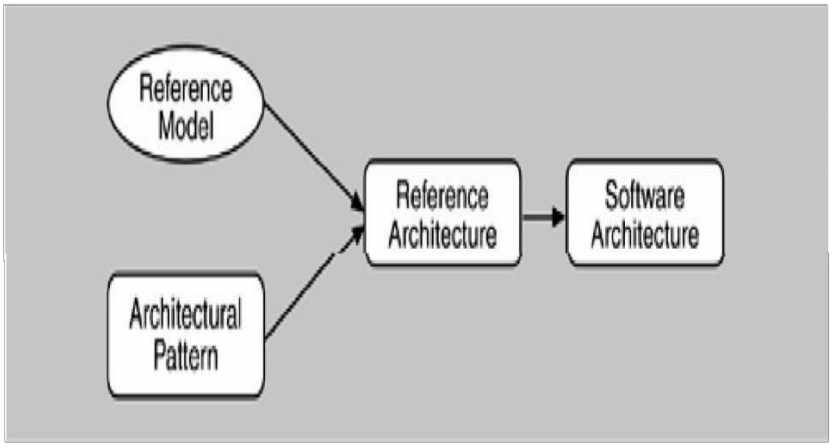
For ex: client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients.

A reference model is a division of functionality together with data flow between the pieces.

A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem.

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.

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.)



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

WHY IS SOFTWARE ARCHITECTURE IMPORTANT?

There are fundamentally three reasons for software architecture's importance from a technical perspective.

- ***Communication among stakeholders***: software architecture represents a common abstraction of a system that most if not all of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus and communication.
- ***Early design decisions***: Software architecture manifests the earliest design decisions about a system with respect to the system's remaining development, its deployment, and its maintenance life.
- It is the earliest point at which design decisions governing the system to be built can be analyzed.
- ***Transferable abstraction of a system***: software architecture model is transferable across systems. It can be applied to other systems exhibiting similar quality attribute and functional attribute and functional requirements and can promote large-scale re-use.

We will address each of these points in turn:

❖ ARCHITECTURE IS THE VEHICLE FOR STAKEHOLDER COMMUNICATION

- Each stakeholder of a software system – customer, user, project manager, coder, tester and so on - is concerned with different system characteristics that are affected by the architecture.
- For ex. The user is concerned that the system is reliable and available when needed; the customer is concerned that the architecture can be implemented on schedule and to budget; the manager is worried that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.
- Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems.

❖ ARCHITECTURE MANIFESTS THE EARLIEST SET OF DESIGN DECISIONS

Software architecture represents a system's earliest set of design decisions. These early decisions are the most difficult to get correct and the hardest to change later in the development process, and they have the most far-reaching effects.

- **The architecture defines constraints on implementation**
  - This means that the implementation must be divided into the prescribed elements, the elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the others as dictated by the architecture.
- **The architecture dictates organizational structure**
  - The normal method for dividing up the labor in a large system is to assign different groups different portions of the system to construct. This is called the work breakdown structure of a system.
- **The architecture inhibits or enables a system's quality attributes**
  - Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.
  - However, the architecture alone cannot guarantee functionality or quality.
  - Decisions at all stages of the life cycle—from high-level design to coding and implementation—affect system quality.
  - Quality is not completely a function of architectural design. To ensure quality, a good architecture is necessary, but not sufficient.
- **Predicting system qualities by studying the architecture**
  - Architecture evaluation techniques such as the architecture tradeoff analysis method support top-down insight into the attributes of software product quality that is made possible (and constrained) by software architectures.
- **The architecture makes it easier to reason about and manage change**
  - Software systems change over their lifetimes.
  - Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.
  - A local change can be accomplished by modifying a single element.
  - A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact.

- **The architecture helps in evolutionary prototyping**
    - The system is executable early in the product's life cycle. Its fidelity increases as prototype parts are replaced by complete versions of the software.
    - A special case of having the system executable early is that potential performance problems can be identified early in the product's life cycle.
- **The architecture enables more accurate cost and schedule estimates**
    - Cost and schedule estimates are an important management tool to enable the manager to acquire the necessary resources and to understand whether a project is in trouble.

❖ ARCHITECTURE AS A TRANSFERABLE, RE-USABLE MODEL

The earlier in the life cycle re-use is applied, the greater the benefit that can be achieved. While code re- use is beneficial, re-use at the architectural level provides tremendous leverage for systems with similar requirements.

- **Software product lines share a common architecture**

A software product line or family is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

- **Systems can be built using large. Externally developed elements**

Whereas earlier software paradigms focused on programming as the prime activity, with progress measured in lines of code, architecture-based development often focuses on composing or assembling elements that are likely to have been developed separately, even independently, from each other.

- **Less is more: it pays to restrict the vocabulary of design alternatives**

We wish to minimize the design complexity of the system we are building. Advantages to this approach include enhanced re-use more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability.

- **An architecture permits template-based development**

An architecture embodies design decisions about how elements interact that, while reflected in each element's implementation, can be localized and written just once.

Templates can be used to capture in one place the inter-element interaction mechanisms.

- **An architecture can be the basis for training**

The architecture, including a description of how elements interact to carry out the required behavior, can serve as the introduction to the system for new project members.

ARCHITECTURAL STRUCTURES AND VIEWS

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

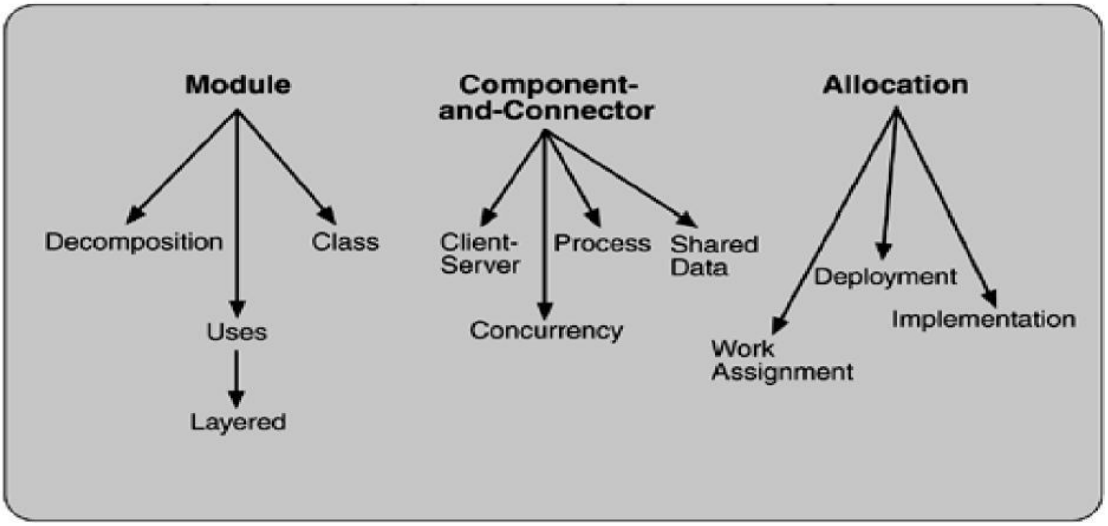
- ***Module structures***.

  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 resulting software 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 are the 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?

Figure 2-3. Common software architecture structures



- ***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 elements to development teams?

SOFTWARE STRUCTURES :

Module

Module-based structures include the following structures.

- ✓ ***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.
- ✓ ***Uses:*** 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.
- ✓ ***Layered:*** Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.
- ✓ ***Class or generalization:*** The class structure allows us to reason about re-use and the incremental addition of functionality.

- ***Component-and-connector***

Component-and-connector structures include the following structures

- ✓ ***Process or communicating processes***: The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations.
- ✓ ***Concurrency:*** 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
- ✓ ***Client-server:*** 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 structures

- ✓ ***Deployment:*** This view allows an engineer to reason about performance, data integrity, availability, and security
- ✓ ***Implementation:*** This is critical for the management of development activities and builds processes.
- ✓ ***Work assignment:*** This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams.

RELATING STRUCTURES TO EACH OTHER

Each of these structures provides a different perspective and design handle on a system, and each is valid and useful in its own right. In general, mappings between structures are many to many. Individual structures bring with them the power to manipulate one or more quality attributes. They represent a powerful separation-of- concerns approach for creating the architecture.

WHICH STRUCTURES TO CHOOSE?

Kruchten's four views follow:

- ✓ ***Logical.*** The elements are "key abstractions," which are manifested in the object-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- connector view.
- ✓ ***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

CHAPTER 2

THE ARCHITECTURE BUSINESS CYCLE

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 environment to the architecture and back to the environment, the ***Architecture Business Cycle (ABC).*** This chapter introduces the ABC in detail and examine the following:

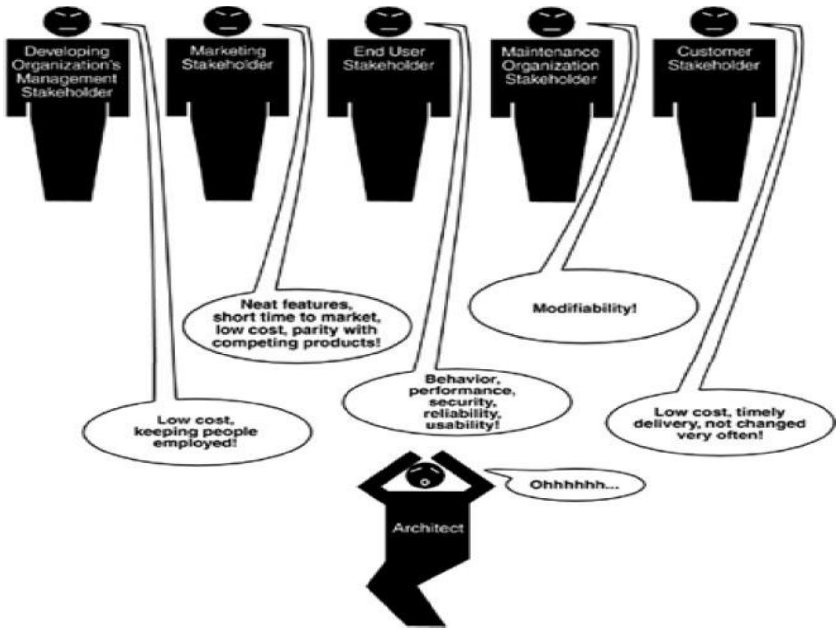
- ✓ How organizational goals influence requirements and development strategy.
- ✓ How requirements lead to architecture.
- ✓ How architectures are analyzed.
- ✓ How architectures yield systems that suggest new organizational capabilities and requirements.

WHERE DO ARCHITECTURES COME FROM?

An architecture is the result of a set of business and technical decisions. There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform. Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.

❖ ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS

- ✓ Many people and organizations interested in the construction of a software system are referred to as stakeholders. E.g. customers, end users, developers, project manager etc.
- ✓ Figure below shows the architect receiving helpful stakeholder “suggestions”.



- ✓ Having an acceptable system involves properties such as performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior.
- ✓ The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory.
- ✓ The reality is that the architect often has to fill in the blanks and mediate the conflicts.

❖ ARCHITECTURES ARE INFLUENCED BY THE DEVELOPING ORGANIZATIONS.

- ✓ Architecture is influenced by the structure or nature of the development organization.
- ✓ There are three classes of influence that come from the developing organizations: immediate business, long-term business and organizational structure.
  - An organization may have an immediate business investment in certain assets, such as existing architectures and the products based on them.
  - An organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may review the proposed system as one means of financing and extending that infrastructure.
  - The organizational structure can shape the software architecture.

❖ ARCHITECTURES ARE INFLUENCED BY THE BACKGROUND AND EXPERIENCE OF THE ARCHITECTS.

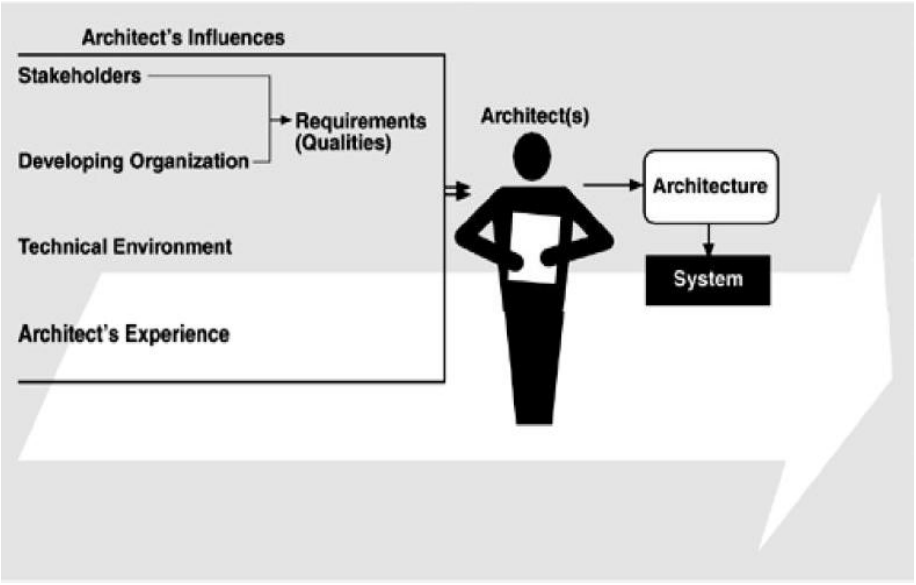
- ✓ If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort.
- ✓ Conversely, if their prior experience with this approach was disastrous, the architects may be reluctant to try it again.
- ✓ Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well.
- ✓ The architects may also wish to experiment with an architectural pattern or technique learned from a book or a course.

❖ ARCHITECTURES ARE INFLUENCED BY THE TECHNICAL ENVIRONMENT

- ✓ A special case of the architect's background and experience is reflected by the *technical environment*.
- ✓ The environment that is current when an architecture is designed will influence that architecture.
- ✓ It might include standard industry practices or software engineering prevalent in the architect's professional community.

❖ RAMIFICATIONS OF INFLUENCES ON AN ARCHITECTURE

- ✓ The influences on the architect, and hence on the architecture, are shown in Figure 1.3.

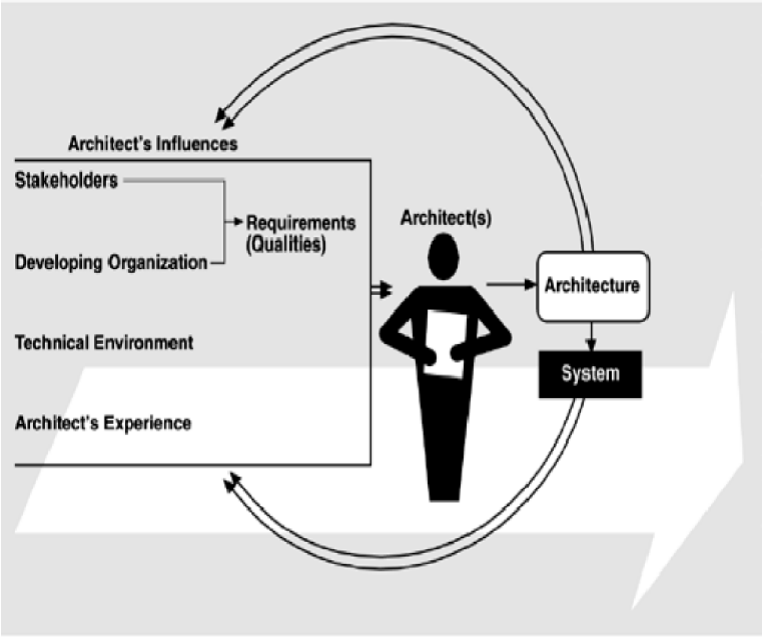


- ✓ Influences on an architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict.
- ✓ Architects need to know and understand the nature, source, and priority of constraints on the project as early as possible.
- ✓ Therefore, *they must identify and actively engage the stakeholders to solicit their needs and expectations*.
- ✓ Architects are influenced by the requirements for the product as derived from its stakeholders, the structure and goals of the developing organization, the available technical environment, and their own background and experience.

❖ THE ARCHITECTURE AFFECTS THE FACTORS THAT INFLUENCE THEM

- ✓ Relationships among business goals, product requirements, architects experience, architectures and fielded systems form a cycle with feedback loops that a business can manage.
- ✓ A business manages this cycle to handle growth, to expand its enterprise area, and to take advantage of previous investments in architecture and system building.
- ✓ Figure 1.4 shows the feedback loops. Some of the feedback comes from the architecture itself, and some comes from the system built from it.

Figure 1.4. The Architecture Business Cycle



Working of architecture business cycle:

1) The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system it particularly prescribes the units of software that must be implemented and integrated to form the system. Teams are formed for individual software units; and the development, test, and integration activities around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units. Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization.
2) The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient production and deployment of the similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.
3) The architecture can affect customer requirements for the next system by giving the customer the opportunity to receive a system in a more reliable, timely and economical manner than if the subsequent system were to be built from scratch.
4) The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base.
5) A few systems will influence and actually change the software engineering culture. i.e, The technical environment in which system builders operate and learn.

SOFTWARE PROCESSES AND THE ARCHITECTURE BUSINESS CYCLE

Software process is the term given to the organization, ritualization, and management of software development activities.

The various activities involved in creating software architecture are:

- **Creating the business case for the system**
    - It is an important step in creating and constraining any future requirements.
    - How much should the product cost?
    - What is its targeted market?
    - What is its targeted time to market?
    - Will it need to interface with other systems?

- Are there system limitations that it must work within?
- These are all the questions that must involve the system's architects.
- They cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, it may be impossible to achieve the business goals.

- **Understanding the requirements**
  - There are a variety of techniques for eliciting requirements from the stakeholders.
  - For ex:
    - ✓ Object oriented analysis uses scenarios, or "use cases" to embody requirements.
    - ✓ Safety-critical systems use more rigorous approaches, such as finite-state-machine models or formal specification languages.
  - Another technique that helps us understand requirements is the creation of prototypes.
  - Regardless of the technique used to elicit the requirements, the desired qualities of the system to be constructed determine the shape of its structure.
- **Creating or selecting the architecture**
  - In the landmark book *The Mythical Man-Month*, Fred Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture.
- **Documenting and communicating the architecture**
  - For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders.
  - Developers must understand the work assignments it requires of them, testers must understand the task structure it imposes on them, management must understand the scheduling implications it suggests, and so forth.
- **Analyzing or evaluating the architecture**
  - Choosing among multiple competing designs in a rational way is one of the architect's greatest challenges.
  - Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders needs.
  - Use scenario-based techniques or architecture tradeoff analysis method (ATAM) or cost benefit analysis method (CBAM).

- **Implementing the system based on the architecture**
    - This activity is concerned with keeping the developers faithful to the structures and interaction protocols constrained by the architecture.
    - Having an explicit and well-communicated architecture is the first step toward ensuring architectural conformance.
- **Ensuring that the implementation conforms to the architecture**
    - Finally, when an architecture is created and used, it goes into a maintenance phase.
    - Constant vigilance is required to ensure that the actual architecture and its representation remain to each other during this phase.

WHAT MAKES A "GOOD" ARCHITECTURE?

Given the same technical requirements for a system, two different architects in different organizations will produce different architectures, how can we determine if either one of them is the right one?

We divide our observations into two clusters: process recommendations and Structural recommendations.

Process rules of thumb are as follows:

- The architecture should be the product of a single architect or a small group of architects with an identified leader.
- The architect (or architecture team) should have the functional requirements for the system and an articulated, prioritized list of quality attributes that the architecture is expected to satisfy.
- The architecture should be well documented, with at least one static view and one dynamic view, using an agreed-on notation that all stakeholders can understand with a minimum of effort.
- The architecture should be circulated to the system's stakeholders, who should be actively involved in its review.
- The architecture should be analyzed for applicable quantitative measures (such as maximum throughput) and formally evaluated for quality attributes before it is too late to make changes to it.

- The architecture should lend itself to incremental implementation via the creation of a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality. This skeletal system can then be used to "grow" the system incrementally, easing the integration and testing efforts.
- The architecture should result in a specific (and small) set of resource contention areas, the resolution of which is clearly specified, circulated and maintained.

Structural rules of thumb are as follows:

- The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns.
- Each module should have a well-defined interface that encapsulates or "hides" changeable aspects from other software that uses its facilities. These interfaces should allow their respective development teams to work largely independent of each other.
- Quality attributes should be achieved using well-known architectural tactics specific to each attribute.
- The architecture should never depend on a particular version of a commercial product or tool.
- Modules that produce data should be separate from modules that consume data. This tends to increase modifiability.
- For parallel processing systems, the architecture should feature well-defined processors or tasks that do not necessarily mirror the module decomposition structure.
- Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of simple interaction patterns

UNIT II

DESIGNING THE ARCHITECTURE WITH STYLES

DESIGNING THE ARCHITECTURE

ARCHITECTURE IN THE LIFE CYCLE

Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the evolutionary delivery life cycle model shown in figure 7.1.

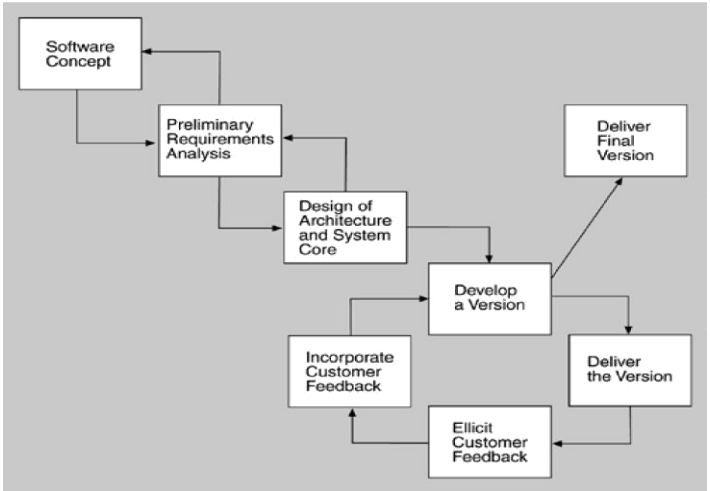


Figure 7.1. Evolutionary Delivery Life Cycle

The intent of this model is to get user and customer feedback and iterate through several releases before the final release. The model also allows the adding of functionality with each iteration and the delivery of a limited version once a sufficient set of features has been developed.

WHEN CAN I BEGIN DESIGNING?

The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. Clearly, you cannot begin the design until you have some idea of the system requirements. On the other hand, it does not take many requirements in order for design to begin.

An architecture is "shaped" by some collection of functional, quality, and business requirements. We call these shaping requirements *architectural drivers* and we see examples of them in our case studies like modifiability, performance requirements availability requirements and so on.

To determine the architectural drivers, identify the highest priority business goals.

There should be relatively few of these. Turn these business goals into quality scenarios or use cases.

DESIGNING THE ARCHITECTURE

A method for designing an architecture to satisfy both quality requirements and functional requirements is called attribute-driven design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between quality attribute achievement and architecture in order to design the architecture.

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.

The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate.

Garage door opener example

Design a product line architecture for a garage door opener with a larger home information system the opener is responsible for raising and lowering the door via a switch, remote control, or the home information system. It is also possible to diagnose problems with the opener from within the home information system.

Input to ADD: a set of requirements

Functional requirements as use cases

- Constraints
- Quality requirements expressed as system specific quality scenarios

ADD Steps:

Steps involved in attribute driven design (ADD)

Choose the module to decompose o Start with entire system

- o Inputs for this module need to be available
- o Constraints, functional and quality requirements

Refine the module

Choose architectural drivers relevant to this decomposition

Choose architectural pattern that satisfies these drivers

Instantiate modules and allocate functionality from use cases representing using multiple views

Define interfaces of child modules

Verify and refine use cases and quality scenarios

Repeat for every module that needs further decomposition

Discussion of the above steps in more detail:

Choose The Module To Decompose

- o the following are the modules: system->subsystem->submodule
- o Decomposition typically starts with system, which then decompose into subsystem and then into sub-modules.
- o In our Example, the garage door opener is a system
- o Opener must interoperate with the home information system

Refine the module

Choose Architectural Drivers:

o choose the architectural drivers from the quality scenarios and functional requirements o The drivers will be among the top priority requirements for the module.

o In the garage system, the 4 scenarios were architectural drivers.

FORMING THE TEAM STRUCTURES

Once the architecture is accepted we assign teams to work on different portions of the design and development.

Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules and a work breakdown structure is created that reflects those teams.

Each team then creates its own internal work practices.

For large systems, the teams may belong to different subcontractors.

Teams adopt “work practices’ including

Team communication via website/bulletin boards

o Naming conventions for files

o Configuration/revision control system

o Quality assurance and testing

procedure

The teams within an organization work on modules, and thus within team high level of communication is necessary

CREATING A SKELETAL SYSTEM

- Develop a skeletal system for the incremental cycle.
- Classical software engineering practice recommends -> “stubbing out”
- Use the architecture as a guide for the implementation sequence
- First implement the software that deals with execution and interaction of architectural components
- Communication between components. Sometimes this is just install third-party middleware
- Then add functionality

  By risk-lowering Or by availability of staff

ARCHITECTURAL STYLES

List of common architectural styles:

Dataflow systems:
- ✓ Batch sequential
- ✓ Pipes and filters

Call-and-return systems:
- ✓ Main program and subroutine
- ✓ OO systems
- ✓ Hierarchical layers.

Independent components:
- ✓ Communicating processes
- ✓ Event systems

Virtual machines:
- ✓ Interpreters
- ✓ Rule-based systems

Data-centered systems:
- ✓ Databases
- ✓ Hypertext systems
- ✓ Blackboards.

PIPES AND FILTERS

Each components has set of inputs and set of outputs. A component reads streams of data on its input and produces streams of data on its output. By applying local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence, components are termed as filters. Connectors of this style serve as conducts for the streams transmitting outputs of one filter to inputs of another. Hence, connectors are termed pipes.

Conditions (invariants) of this style are:

Filters must be independent entities. They should not share state with other filter . Filters do not know the identity of their upstream and downstream filters. Specification might restrict what appears on input pipes and the result that appears on the output pipes.

Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

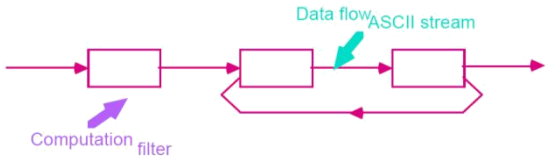


Figure 1: Pipes and Filters

Common specialization of this style includes :

Pipelines:

Restrict the topologies to linear sequences of filters.

Bounded pipes:

Restrict the amount of data that can reside on pipe.

Typed pipes:

Requires that the data passed between two filters have a well-defined type.

Batch sequential system:

A degenerate case of a pipeline architecture occurs when each filter processes all of its input data as a single entity. In these systems pipes no longer serve the function of providing a stream of data and are largely vestigial.

Example 1:

Best known example of pipe-and-filter architecture are programs written in UNIX-SHELL. Unix supports this style by providing a notation for connecting components [Unix process] and by providing run-time mechanisms for implementing pipes.

Example 2:

Traditionally compilers have been viewed as pipeline systems. Stages in the pipeline include lexical analysis parsing, semantic analysis and code generation other examples of this type are:

Signal processing domains

Parallel processing

Functional processing

Distributed systems.

Advantages:

- They allow the designer to understand the overall input/output behavior of a system as a simple composition of the behavior of the individual filters.
- They support reuse: Any two filters can be hooked together if they agree on data.
- Systems are easy to maintain and enhance: New filters can be added to exciting systems.
- They permit certain kinds of specialized analysis eg: deadlock, throughput.
- They support concurrent execution.

Disadvantages:

- They lead to a batch organization of processing.
- Filters are independent even though they process data incrementally.
- Not good at handling interactive applications When incremental display updates are required.
- They may be hampered by having to maintain correspondences between two separate but related streams.
- Lowest common denominator on data transmission.
- This can lead to both loss of performance and to increased complexity in writing the filters.

OBJECT-ORIENTED AND DATA ABSTRACTION

In this approach, data representation and their associated primitive operations are encapsulated in the abstract data type (ADT) or object. The components of this style are- objects/ADT's objects interact through function and procedure invocations.

Two important aspects of this style are:

Object is responsible for preserving the integrity of its representation.

Representation is hidden from other objects.

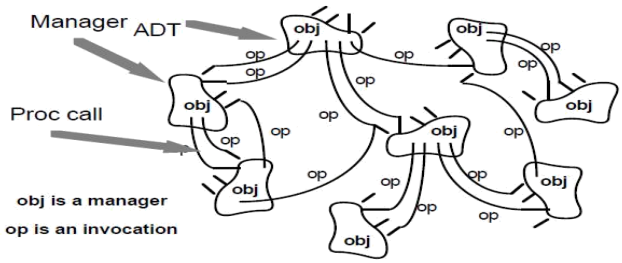


Figure 2: Abstract Data Types and Objects

Advantages

- It is possible to change the implementation without affecting the clients because an object hides its representation from clients.
- The bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

Disadvantages

- To call a procedure, it must know the identity of the other object.
- Whenever the identity of object changes it is necessary to modify all other objects that explicitly invoke it.

EVENT-BASED, IMPLICIT INVOCATION

Instead of invoking the procedure directly a component can announce one or more events.

Other components in the system can register an interest in an event by associating a procedure to it.

When the event is announced, the system itself invokes all of the procedure that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.

Architecturally speaking, the components in an implicit invocation style are modules whose interface provides both a collection of procedures and a set of events.

Advantages:

It provides strong support for reuse

Any component can be introduced into the system simply by registering it for the events of that system.

Implicit invocation eases system evolution.

Components may be replaced by other components without affecting the interfaces of other components.

Disadvantages:

- Components relinquish control over the computation performed by the system.
- Concerns change of data.
- Global performance and resource management can become artificial issues.

LAYERED SYSTEMS:

A layered system is organized hierarchically.

Each layer provides service to the layer above it.

Inner layers are hidden from all except the adjacent layers. Connectors are defined by the protocols that determine how layers interact each other.

Goal is to achieve qualities of modifiability portability.

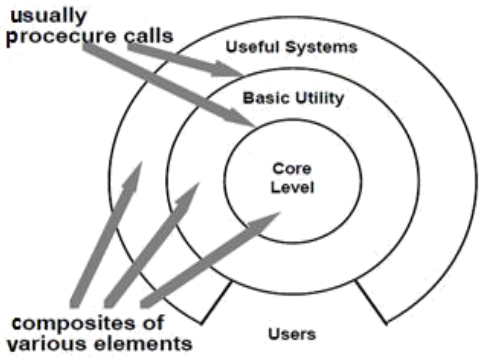


Figure 3: Layered Systems

Examples:

Layered communication protocol

Operating systems

Database systems

Advantages:

- They support designs based on increasing levels abstraction.
- Allows implementers to partition a complex problem into a sequence of incremental steps.
- They support enhancement.
- They support reuse.

Disadvantages:

- Not easily all systems can be structures in a layered fashion.
- Performance may require closer coupling between logically high-level functions and their lower-level implementations.

- Difficulty to mapping existing protocols into the ISO framework as many of those protocols bridge several layers.
- Layer bridging: functions is one layer may talk to other than its immediate neighbor.

REPOSITORIES:[data cantered architecture]

Goal of achieving the quality of integrability of data. In this style, there are two kinds of components. Central data structure- represents current state. Collection of independent components which operate on central data store. The choice of a control discipline leads to two major sub categories.

- Type of transactions is an input stream trigger selection of process to execute
- Current state of the central data structure is the main trigger for selecting processes to execute.

Active repository such as blackboard.

Blackboard:

Three major parts:

- **Knowledge sources:**

  Separate, independent parcels of application – dependents knowledge.

- **Blackboard data structure:**

  Problem solving state data, organized into an application-dependent hierarchy

- **Control:**

  Driven entirely by the state of blackboard

  Invocation of a knowledge source (ks) is triggered by the state of blackboard.

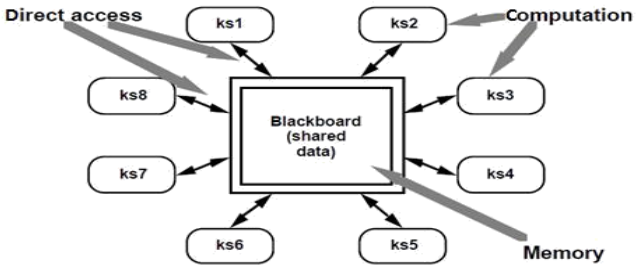


Figure 4: The Blackboard

The actual focus of control can be in

- knowledge source
- blackboard
- Separate module or
- combination of these

Blackboard systems have traditionally been used for application requiring complex interpretation of signal processing like speech recognition, pattern recognition.

INTERPRETERS

An interpreter includes pseudo program being interpreted and interpretation engine.

Pseudo program includes the program and activation record.

Interpretation engine includes both definition of interpreter and current state of its execution.

Interpretation engine: to do the work

Memory: that contains pseudo code to be interpreted.

Representation of control state of interpretation engine

Representation of control state of the program being simulated.

Ex: JVM or “virtual Pascal machine”

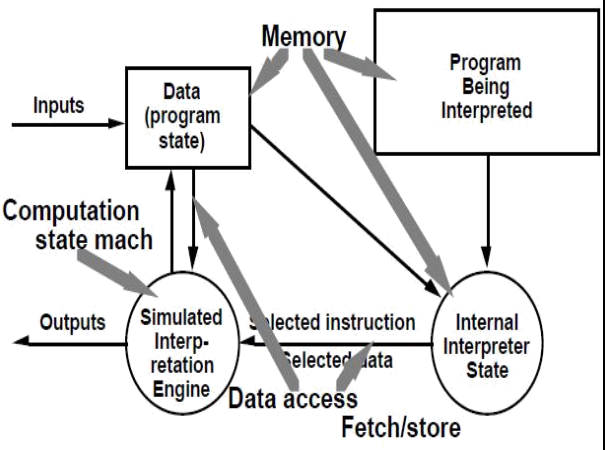


Figure 5: Interpreter

Advantages:

Executing program via interpreters adds flexibility through the ability to interrupt and query the program

Disadvantages:

Performance cost because of additional computational involved

UNIT III

CREATING AN ARCHITECTURE-I

3.1 FUNCTIONALITY AND ARCHITECTURE

Functionality: It is the ability of the system to do the work for which it was intended.

A task requires that many or most of the system's elements work in a coordinated manner to complete the job, just as framers, electricians, plumbers, drywall hangers, painters, and finish carpenters all come together to cooperatively build a house.

Software architecture constrains its allocation to structure when *other* quality attributes are important.

3.2 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. For example:

Usability involves both architectural and non-architectural aspects.

Modifiability is determined by how functionality is divided (architectural) and by coding techniques within a module (non-architectural).

Performance involves both architectural and non-architectural dependencies

The message of this section is twofold:

- 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 architectural level
- Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality.

3.3 SYSTEM QUALITY ATTRIBUTES

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) thatgenerated the stimulus.

Stimulus.The stimulus is a condition that needs to be considered when it arrives at a system.

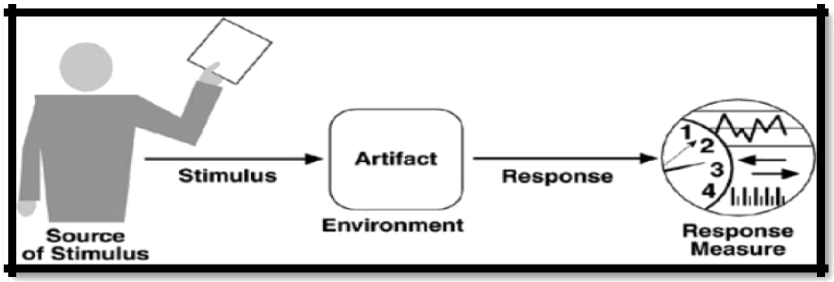
Environment. The stimulus occurs within certain conditions. The system may be in an overloadcondition or may be running when the stimulus occurs, or some other condition may be true.

Artifact.Some artifact is stimulated. This may be the whole system or some pieces of it.

Response.The response is the activity undertaken after the arrival of the stimulus.

Response measure. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Figure 3.1 shows the parts of a quality attribute scenario.



QUALITY ATTRIBUTE SCENARIOS IN PRACTICE

AVAILABILITY SCENARIO

Availability is concerned with system failure and its associated consequencesFailures are usually a result of system errors that are derived from faults in the system. It is typically defines as

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure + mean time to repair}}$$

Source of stimulus.We differentiate between internal and external indications of faults or failure since thedesired system response may be different. In our example, the unexpected message arrives from outside the system.

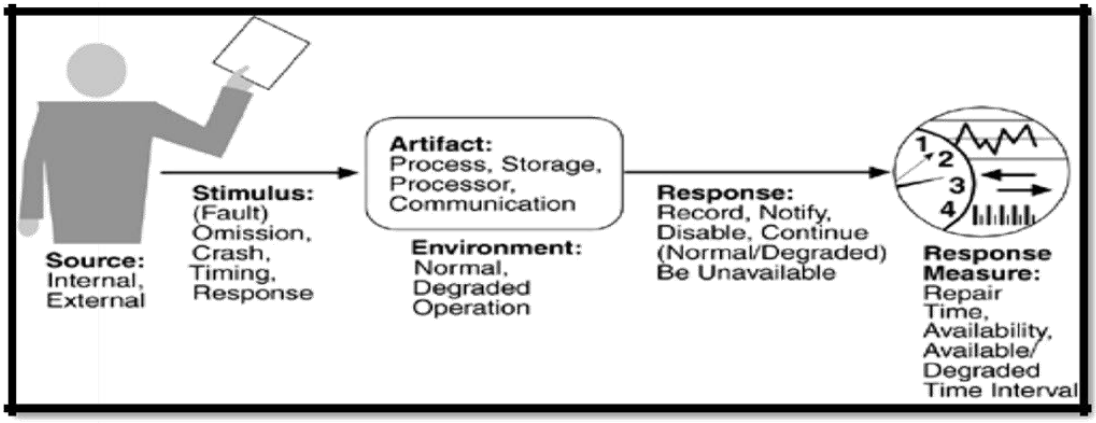
Stimulus. A fault of one of the following classes occurs.

omission. A component fails to respond to an input.

crash. The component repeatedly suffers omission faults.

timing. A component responds but the response is early or late.

response. A component responds with an incorrect value.



Artifact. This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.

Environment.The state of the system when the fault or failure occurs may also affect the desired systemresponse. For example, if the system has already seen some faults and is operating in other than normal mode, it may be desirable to shut it down totally. However, if this is the first fault observed, some degradation of response time or function may be preferred. In our example, the system is operating normally.

Response. There are a number of possible reactions to a system failure. These include logging the failure,notifying selected users or other systems, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair. In our example, the system should notify the operator of the unexpected message and continue to operate normally.

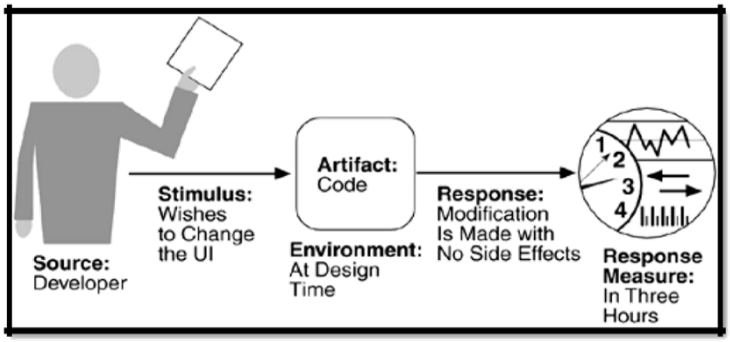
Response measure. The response measure can specify an availability percentage, or it can specify a time torepair, times during which the system must be available, or the duration for which the system must be available.

MODIFIABILITY SCENARIO

Modifiability is about the cost of change. It brings up two concerns.

What can change (the artifact)?

When is the change made and who makes it (the environment)?



Source of stimulus.This portion specifies who makes the changes—the developer, a system administrator, oran end user. Clearly, there must be machinery in place to allow the system administrator or end user to modify a system, but this is a common occurrence. In Figure 4.4, the modification is to be made by the developer.

Stimulus. This portion specifies the changes to be made. A change can be the addition of a function, themodification of an existing function, or the deletion of a function. It can also be made to the qualities of the system—making it more responsive, increasing its availability, and so forth. The capacity of the system may also change. Increasing the number of simultaneous users is a frequent requirement. In our example, the stimulus is a request to make a modification, which can be to the function, quality, or capacity.

Artifact. This portion specifies what is to be changed—the functionality of a system, its platform, its userinterface, its environment, or another system with which it interoperates. In Figure 4.4, the modification is to the user interface.

Environment. This portion specifies when the change can be made—design time, compile time, build time,initiation time, or runtime. In our example, the modification is to occur at design time.

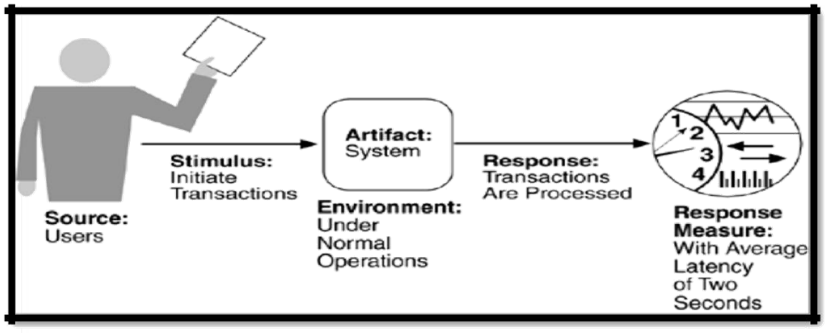
Response. Whoever makes the change must understand how to make it, and then make it, test it and deploy it.

In our example, the modification is made with no side effects.

Response measure.All of the possible responses take time and cost money, and so time and cost are the mostdesirable measures. Time is not always possible to predict, however, and so less ideal measures are frequently used, such as the extent of the change (number of modules affected). In our example, the time to perform the modification should be less than three hours.

PERFORMANCE SCENARIO:

Performance is about timing. Events (interrupts, messages, requests from users, or the passage of time) occur, and the system must respond to them. There are a variety of characterizations of event arrival and the response but basically performance is concerned with how long it takes the system to respond when an event occurs.



Source of stimulus.The stimuli arrive either from external (possibly multiple) or internal sources. In ourexample, the source of the stimulus is a collection of users.

Stimulus. The stimuli are the event arrivals. The arrival pattern can be characterized as periodic, stochastic, orsporadic. In our example, the stimulus is the stochastic initiation of 1,000 transactions per minute.

Artifact.The artifact is always the system's services, as it is in our example.

Environment.The system can be in various operational modes, such as normal, emergency, or overload. In ourexample, the system is in normal mode.

Response.The system must process the arriving events. This may cause a change in the system environment(e.g., from normal to overload mode). In our example, the transactions are processed.

Response measure.The response measures are the time it takes to process the arriving events (latency or adeadline by which the event must be processed), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate, data loss). In our example, the transactions should be processed with an average latency of two seconds.

SECURITY SCENARIO

Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users. An attempt to breach security is called an attack and can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

Security can be characterized as a system providing non-repudiation, confidentiality, integrity, assurance, availability, and auditing. For each term, we provide a definition and an example.

Non-repudiation is the property that a transaction (access to or modification of data or services)cannot be denied by any of the parties to it. This means you cannot deny that you ordered that item over the Internet if, in fact, you did.

Confidentiality is the property that data or services are protected from unauthorized access. Thismeans that a hacker cannot access your income tax returns on a government computer.

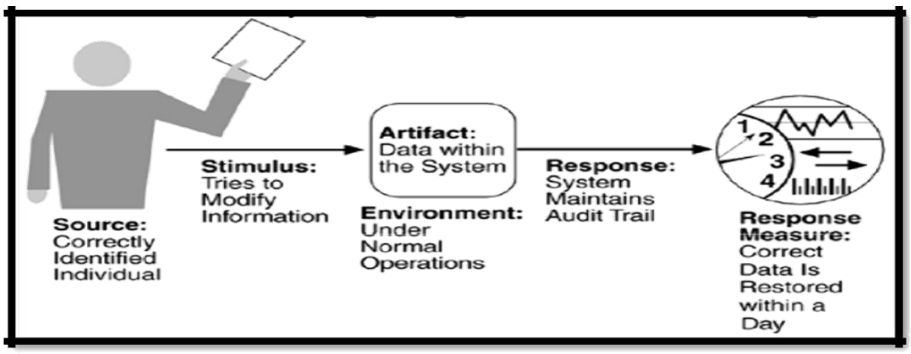
Integrity is the property that data or services are being delivered as intended. This means that yourgrade has not been changed since your instructor assigned it.

Assurance is the property that the parties to a transaction are who they purport to be. This means that,when a customer sends a credit card number to an Internet merchant, the merchant is who the customer thinks they are.

Availability is the property that the system will be available for legitimate use. This means that adenial-of-service attack won't prevent your ordering *this* book.

Auditing is the property that the system tracks activities within it at levels sufficient to reconstructthem. This means that, if you transfer money out of one account to another account, in Switzerland, the system will maintain a record of that transfer.

Each of these security categories gives rise to a collection of general scenarios.



Source of stimulus.The source of the attack may be either a human or another system. It may have beenpreviously identified (either correctly or incorrectly) or may be currently unknown.

Stimulus. The stimulus is an attack or an attempt to break security. We characterize this as an unauthorizedperson or system trying to display information, change and/or delete information, access services of the system, or reduce availability of system services. In Figure, the stimulus is an attempt to modify data.

Artifact.The target of the attack can be either the services of the system or the data within it. In our example,the target is data within the system.

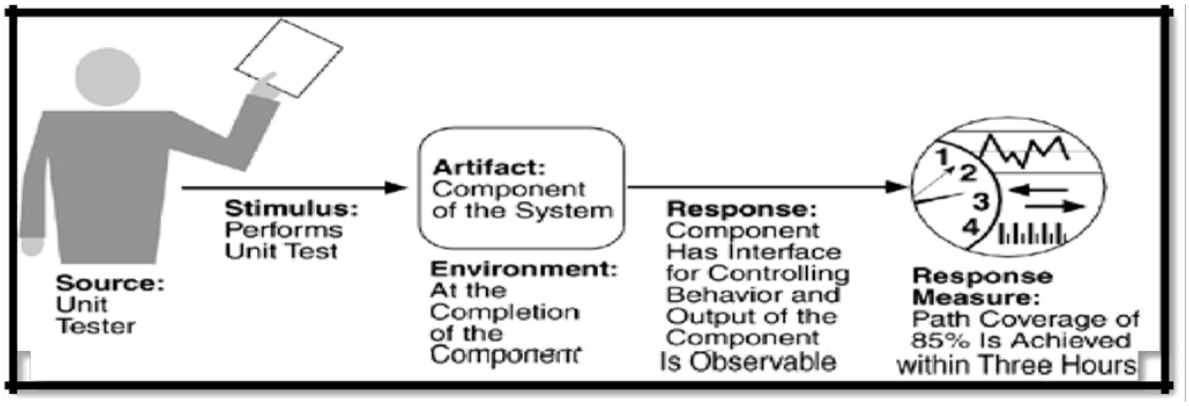
Environment. The attack can come when the system is either online or offline, either connected to ordisconnected from a network, either behind a firewall or open to the network.

Response.Using services without authorization or preventing legitimate users from using services is a differentgoal from seeing sensitive data or modifying it. Thus, the system must authorize legitimate users and grant them access to data and services, at the same time rejecting unauthorized users, denying them access, and reporting unauthorized access

Response measure.Measures of a system's response include the difficulty of mounting various attacks and thedifficulty of recovering from and surviving attacks. In our example, the audit trail allows the accounts from which money was embezzled to be restored to their original state.

TESTABILITY SCENARIO:

Software testability refers to the ease with which software can be made to demonstrate its faults through testing. In particular, testability refers to the probability, assuming that the software has at least one fault that it will fail on its *next* test execution. Testing is done by various developers, testers, verifiers, or users and is the last step of various parts of the software life cycle. Portions of the code, the design, or the complete system may be tested.



Source of stimulus.The testing is performed by unit testers, integration testers, system testers, or the client. Atest of the design may be performed by other developers or by an external group. In our example, the testing is performed by a tester.

Stimulus. The stimulus for the testing is that a milestone in the development process is met. This might be thecompletion of an analysis or design increment, the completion of a coding increment such as a class, the completed integration of a subsystem, or the completion of the whole system. In our example, the testing is triggered by the completion of a unit of code.

Artifact.A design, a piece of code, or the whole system is the artifact being tested. In our example, a unit of codeis to be tested.

Environment. The test can happen at design time, at development time, at compile time, or at deployment time.

In Figure, the test occurs during development.

Response.Since testability is related to observability and controllability, the desired response is that thesystem can be controlled to perform the desired tests and that the response to each test can be observed. In our example, the unit can be controlled and its responses captured.

Response measure.Response measures are the percentage of statements that have been executed in some test,the length of the longest test chain (a measure of the difficulty of performing the tests), and estimates of the probability of finding additional faults. In Figure, the measurement is percentage coverage of executable statements.

USABILITY SCENARIO

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides. It can be broken down into the following areas:

Learning system features.If the user is unfamiliar with a particular system or a particular aspect of it, what canthe system do to make the task of learning easier?

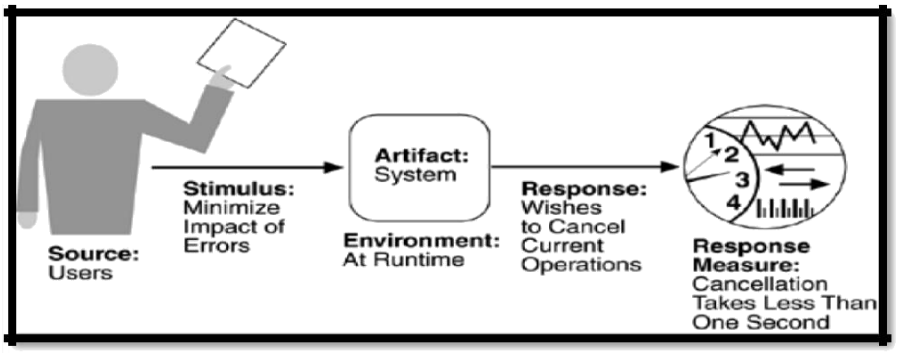
Using a system efficiently.What can the system do to make the user more efficient in its operation?

Minimizing the impact of errors. What can the system do so that a user error has minimal impact?

Adapting the system to user needs.How can the user (or the system itself) adapt to make the user's task easier?

Increasing confidence and satisfaction.What does the system do to give the user confidence that the correctaction is being taken?

A user, wanting to minimize the impact of an error, wishes to cancel a system operation at runtime; cancellation takes place in less than one second. The portions of the usability general scenarios are:



Source of stimulus.The end user is always the source of the stimulus.

Stimulus.The stimulus is that the end user wishes to use a system efficiently, learn to use the system, minimizethe impact of errors, adapt the system, or feel comfortable with the system. In our example, the user wishes to cancel an operation, which is an example of minimizing the impact of errors.

Artifact.The artifact is always the system.

Environment.The user actions with which usability is concerned always occur at runtime or at systemconfiguration time. In Figure, the cancellation occurs at runtime.

Response.The system should either provide the user with the features needed or anticipate the user's needs. Inour example, the cancellation occurs as the user wishes and the system is restored to its prior state.

Response measure.The response is measured by task time, number of errors, number of problems solved,user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time/data lost when an error occurs. In Figure, the cancellation should occur in less than one second.

COMMUNICATING CONCEPTS USING GENERAL SCENARIOS

One of the uses of general scenarios is to enable stakeholders to communicate. Table 4.7 gives the stimuli possible for each of the attributes and shows a number of different concepts. Some stimuli occur during runtime and others occur before. The problem for the architect is to understand which of these stimuli represent the same occurrence, which are aggregates of other stimuli, and which are independent.

Table 4.7. Quality Attribute Stimuli

Quality Attribute	Stimulus
Availability	Unexpected event, nonoccurrence of expected event
Modifiability	Request to add/delete/change/vary functionality, platform, quality attribute, or capacity
Performance	Periodic, stochastic, or sporadic
Security	Tries to display, modify, change/delete information, access, or reduce availability to system services
Testability	Completion of phase of system development
Usability	Wants to learn system features, use a system efficiently, minimize the impact of errors, adapt the system, feel comfortable

3.5 OTHER SYSTEM QUALITY ATTRIBUTES

SCALABILITY

PORTABILITY

3.6 BUSINESS QUALITIES

Time to market.

If there is competitive pressure or a short window of opportunity for a system or product, development time becomes important. This in turn leads to pressure to buy or otherwise re-use existing elements.

Cost and benefit.

The development effort will naturally have a budget that must not be exceeded. Different architectures will yield different development costs. For instance, an architecture that relies on technology (or expertise with a technology) not resident in the developing organization will be more expensive to realize than one that takes advantage of assets already inhouse. An architecture that is highly flexible will typically be more costly to build than one that is rigid (although it will be less costly to maintain and modify).

Projected lifetime of the system.

If the system is intended to have a long lifetime, modifiability, scalability, and portability become important. On the other hand, a modifiable, extensible product is more likely to survive longer in the marketplace, extending its lifetime.

Targeted market.

For general-purpose (mass-market) software, the platforms on which a system runs as well as its feature set will determine the size of the potential market. Thus, portability and functionality are key to market share. Other qualities, such as performance, reliability, and usability also play a role.

Rollout schedule.

If a product is to be introduced as base functionality with many features released later, the flexibility and customizability of the architecture are important. Particularly, the system must be constructed with ease of expansion and contraction in mind.

Integration with legacy systems.

If the new system has to *integrate* with existing systems, care must be taken to define appropriate integration mechanisms. This property is clearly of marketing importance but has substantial architectural implications.

3.7 ARCHITECTURE QUALITIES

Conceptual integrity is the underlying theme or vision that unifies the design of the system at all levels.The architecture should do similar things in similar ways.

Correctness **and** ***completeness*** are essential for the architecture to allow for all of the system'srequirements and runtime resource constraints to be met.

Buildability allows the system to be completed by the available team in a timely manner and to be opento certain changes as development progresses.

3.8 INTRODUCING TACTICS

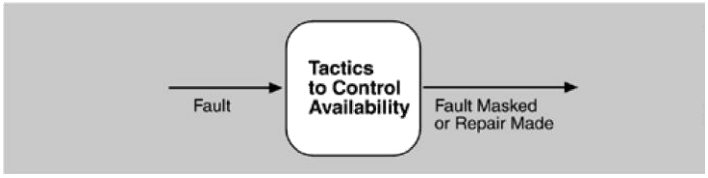
A **tactic** is a design decision that influences the control of a quality attribute response.

Figure 3.8 Tactics are intended to control responses to stimuli.

- *Tactics can refine other tactics*. For each quality attribute that we discuss, we organize the tactics as ahierarchy.

Patterns package tactics. A pattern that supports availability will likely use both a redundancy tactic anda synchronization tactic.

3.9 AVAILABILITY TACTICS



The above figure depicts goal of availability tactics. 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. In some cases, the monitoring or recovery is automatic and in others it is manual.

FAULT DETECTION

- ***Ping/echo***.One component issues a ping and expects to receive back an echo, within a predefined time,from the component under scrutiny. This can be used within a group of components mutually responsible for one task

- ***Heartbeat (dead man timer).*** In this case one component emits a heartbeat message periodically andnother component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. The heartbeat can also carry data.
- ***Exceptions****.*The exception handler typically executes in the same process that introduced the exception.

FAULT RECOVERY

- ***Voting.*** Processes running on redundant processors each take equivalent input and compute a simpleoutput value that is sent to a voter. If the voter detects deviant behavior from a single processor, it fails it.
- ***Active redundancy (hot restart).*** All redundant components respond to events in parallel. Theresponse from only one component is used (usually the first to respond), and the rest are discarded. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs
- ***Passive redundancy (warm restart/dual redundancy/triple redundancy)****.*One component (theprimary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services.
- ***Spare****.*A standby spare computing platform is configured to replace many different failed components.It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs.
- ***Shadow operation****.*A previously failed component may be run in "shadow mode" for a short time tomake sure that it mimics the behavior of the working components before restoring it to service.
- ***State resynchronization****.*The passive and active redundancy tactics require the component being restored to have its state upgraded before its return to service.
- ***Checkpoint/rollback****.*A checkpoint is a recording of a consistent state created either periodically or inresponse to specific events. Sometimes a system fails in an unusual manner, with a detectably inconsistent state. In this case, the system should be restored using a previous checkpoint of a consistent state and a log of the transactions that occurred since the snapshot was taken.

FAULT PREVENTION

- **R*emoval from service*.**This tactic removes a component of the system from operation to undergo someactivities to prevent anticipated failures.
- ***Transactions*.**A transaction is the bundling of several sequential steps such that the entire bundle canbe undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and also to prevent collisions among several simultaneous threads accessing the same data.

***Process monitor*.**Once a fault in a process has been detected, a monitoring process can delete thenonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

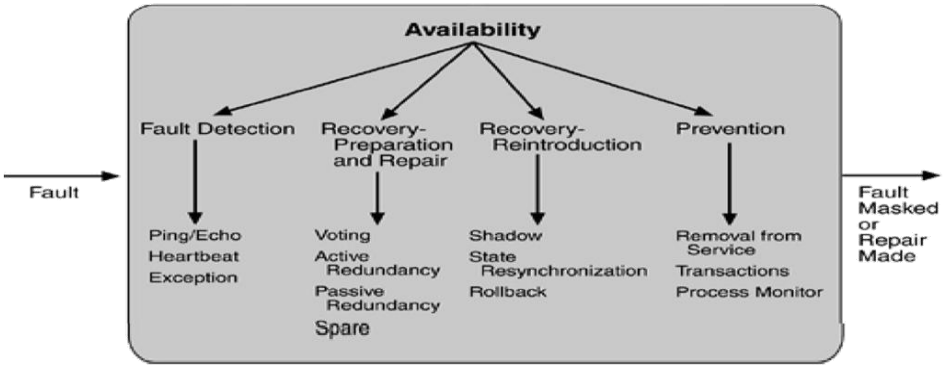
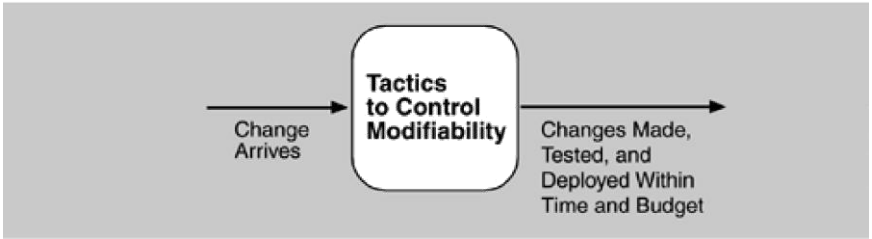


Figure 3.9. Summary of availability tactics

3.10 MODIFIABILITY TACTICS

The figure below represent goal of modifiability tactics.



LOCALIZE MODIFICATIONS

- ***Maintain semantic coherence*.** Semantic coherence refers to the relationships among Responsibilities in a module. The goal is to ensure that all of these responsibilities work together without excessive reliance on other modules.

- ***Anticipate expected changes.*** Considering the set of envisioned changes provides a way to evaluate a particular assignment of responsibilities. In reality this tactic is difficult to use by itself since it is not possible to anticipate all changes.
- ***Generalize the module.*** Making a module more general allows it to compute a broader range Of functions based on input
- ***Limit possible options.*** Modifications, especially within a product line, may be far ranging and Hence affect many modules. Restricting the possible options will reduce the effect of these modifications

PREVENT RIPPLE EFFECTS

We begin our discussion of the ripple effect by discussing the various types of dependencies that one module can have on another. We identify eight types:

Syntax of *data service.*

Semantics of *data service.*

Sequence of *data control.*

Identity of an interface of A

Location of A (runtime).

Quality of service/data provided by A.

Existence of A

Resource behaviour of A.

With this understanding of dependency types, we can now discuss tactics available to the architect for preventing the ripple effect for certain types.

- ***Hide information.*** Information hiding is the decomposition of the responsibilities for an entity Into smaller pieces and choosing which information to make private and which to make public. The goal is to isolate changes within one module and prevent changes from propagating to others
- ***Maintain existing interfaces*** it is difficult to mask dependencies on quality of data or quality of service,resource usage, or resource ownership. Interface stability can also be achieved by separating the interface from the implementation. This allows the creation of abstract interfaces that mask variations.
- ***Restrict communication paths.*** This will reduce the ripple effect since data production/consumptionintroduces dependencies that cause ripples.

- ***Use an intermediary*** If B has any type of dependency on A other than semantic, it is possible to insert an intermediary between B and A that manages activities associated with the dependency.

DEFER BINDING TIME

Many tactics are intended to have impact at loadtime or runtime, such as the following.

- ***Runtime registration*** supports plug-and-play operation at the cost of additional overhead *to* Manage the registration.
- ***Configuration files*** are intended to set parameters at startup.
- ***Polymorphism*** allows late binding of method calls.
- ***Component replacement*** allows load time binding.
- ***Adherence to defined protocols*** allows runtime binding of independent processes.

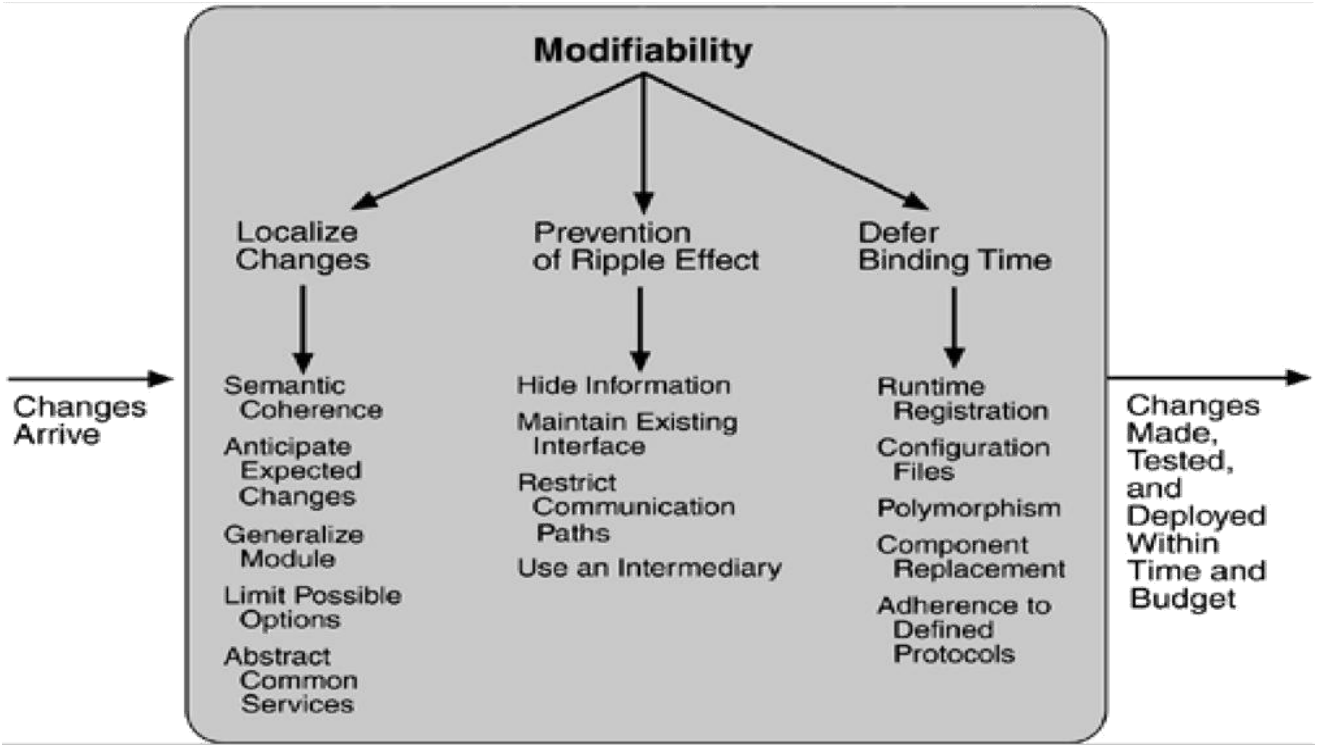
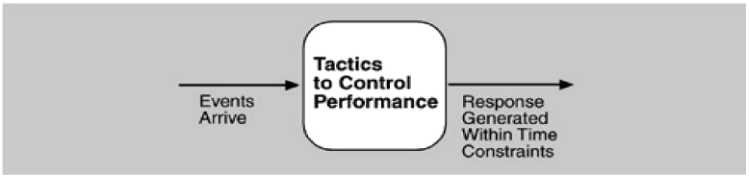


Figure 3.10. Summary of modifiability tactics

3.11 PERFORMANCE TACTICS

Performance tactics control the time within which a response is generated. Goal of performance tactics is shown below:



After an event arrives, either the system is processing on that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: resource consumption and blocked time.

Resource consumption. For example, a message is generated by one component, is placed on thenetwork, and arrives at another component. Each of these phases contributes to the overall latency of the processing of that event.

Blocked time. A computation can be blocked from using a resource because of contention for it, becausethe resource is unavailable, or because the computation depends on the result of other computations that are not yet available.

Contention for resources

Availability of resources

Dependency on other computation.

RESOURCE DEMAND

One tactic for reducing latency is to reduce the resources required for processing an event stream.

- ***Increase computational efficiency.*** One step in the processing of an event or a message is applying some algorithm. Improving the algorithms used in critical areas will decrease latency. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk.
- ***Reduce computational overhead.*** If there is no request for a resource, processing needs are reduced. The use of intermediaries increases the resources consumed in processing an event stream, and so removing them improves latency.

Another tactic for reducing latency is to reduce the number of events processed. This can be done in one of two fashions.

- ***Manage event rate.*** If it is possible to reduce the sampling frequency at which environmental Variables are monitored, demand can be reduced.
- ***Control frequency of sampling.*** If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.

Other tactics for reducing or managing demand involve controlling the use of resources.

- ***Bound execution times.*** Place a limit on how much execution time is used to respond to an event. Sometimes this makes sense and sometimes it does not.

- ***Bound queue sizes***. This controls the maximum number of queued arrivals and consequently there sources used to process the arrivals.

RESOURCE MANAGEMENT

- ***Introduce concurrency***. If requests can be processed in parallel, the blocked time can be reduced

- ***Maintain multiple copies of either data or computations***. Clients in a client-server pattern are replicas of the computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a central server.

  Increase available resources.Faster processors, additional processors, additional memory, and fasternetworks all have the potential for reducing latency.

RESOURCE ARBITRATION

- ***First-in/First-out***. FIFO queues treat all requests for resources as equals and satisfy them in turn.

  Fixed-priority scheduling. Fixed-priority scheduling assigns each source of resource requests A particular priority and assigns the resources in that priority order. Three common prioritization strategies are

  semantic importance. Each stream is assigned a priority statically according to some domaincharacteristic of the task that generates it.

  deadline monotonic. Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines.

  *rate monotonic.*Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods.

  round robin. Round robin is a scheduling strategy that orders the requests and then, at everyassignment possibility, assigns the resource to the next request in that order.

  earliest deadline first. Earliest deadline first assigns priorities based on the pending requests withthe earliest deadline.

- ***Static scheduling***. A cyclic executive schedule is a scheduling strategy where the pre-emption pointsand the sequence of assignment to the resource are determined offline.

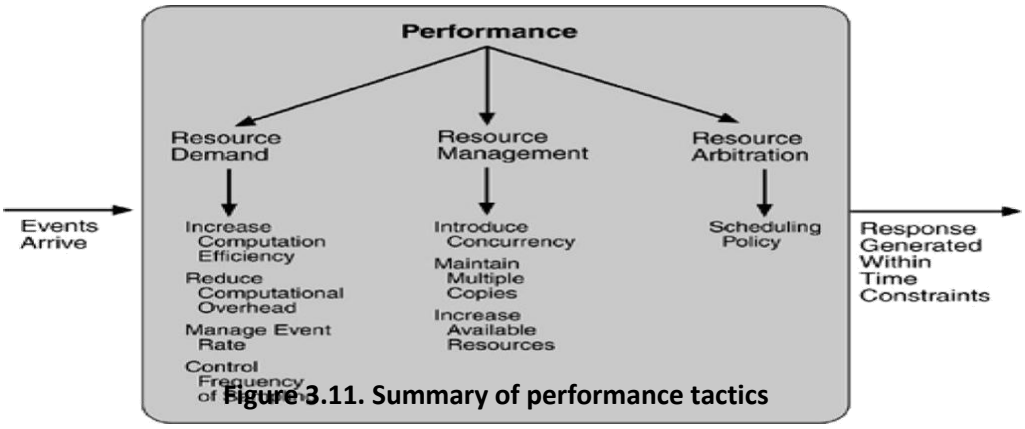
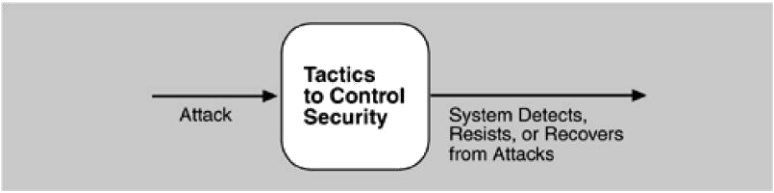


Figure 3.11. Summary of performance tactics

3.12 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.



RESISTING ATTACKS

- ***Authenticate users.*** Authentication is ensuring that a user or remote computer is actually who It purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication.
- ***Authorize users.*** Authorization is ensuring that an authenticated user has the rights to access andmodify either data or services. Access control can be by user or by user class.
- ***Maintain data confidentiality***.Data should be protected from unauthorized access. Confidentiality isusually achieved by applying some form of encryption to data and to communication links. Encryption provides extra protection to persistently maintained data beyond that available from authorization.
- ***Maintain integrity.*** Data should be delivered as intended. It can have redundant information encodedin it, such as checksums or hash results, which can be encrypted either along with or independently from the original data.

  Limit exposure. Attacks typically depend on exploiting a single weakness to attack all data and serviceson a host. The architect can design the allocation of services to hosts so that limited services are available on each host.

- ***Limit access.*** Firewalls restrict access based on message source or destination port. Messages fromunknown sources may be a form of an attack. It is not always possible to limit access to known sources.

DETECTING ATTACKS

- The detection of an attack is usually through an *intrusion detection* system.
- Such systems work by comparing network traffic patterns to a database.
- In the case of misuse detection, the traffic pattern is compared to historic patterns of known attacks.
- In the case of anomaly detection, the traffic pattern is compared to a historical baseline of itself. Frequently, the packets must be filtered in order to make comparisons.
- Filtering can be on the basis of protocol, TCP flags, payload sizes, source or destination address, or port number.
- Intrusion detectors must have some sort of sensor to detect attacks, managers to do sensor fusion, databases for storing events for later analysis, tools for offline reporting and analysis, and a control console so that the analyst can modify intrusion detection actions.

RECOVERING FROM ATTACKS

- Tactics involved in recovering from an attack can be divided into those concerned with restoring state and those concerned with attacker identification (for either preventive or punitive purposes).
- The tactics used in restoring the system or data to a correct state overlap with those used for availability since they are both concerned with recovering a consistent state from an inconsistent state. One difference is that special attention is paid to maintaining redundant copies of system administrative data such as passwords, access control lists, domain name services, and user profile data.
- The tactic for identifying an attacker is to *maintain an audit trai.l*

- **An** audit trail is a copy of each transaction applied to the data in the system together with identifying information.
- Audit information can be used to trace the actions of an attacker, support nonrepudiation (it provides evidence that a particular request was made), and support system recovery.

- Audit trails are often attack targets themselves and therefore should be maintained in a trusted fashion.

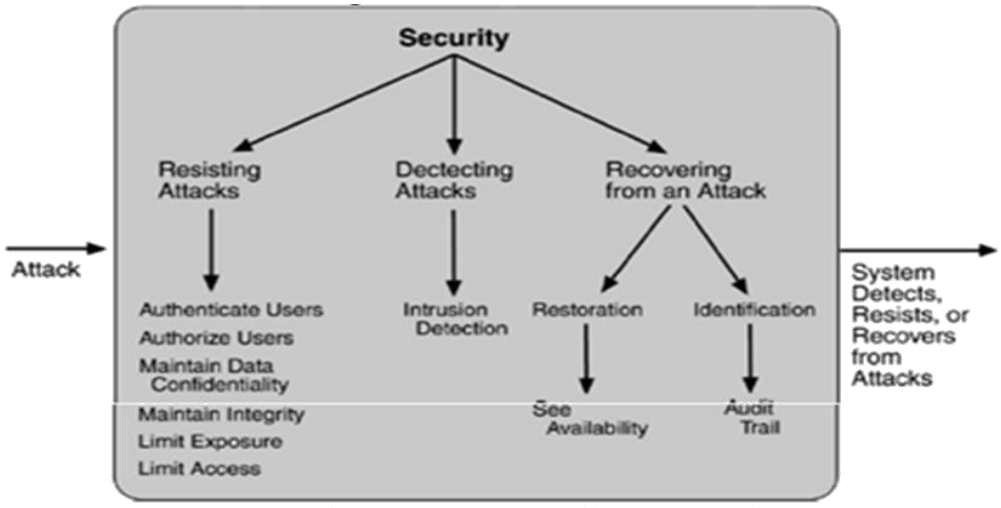
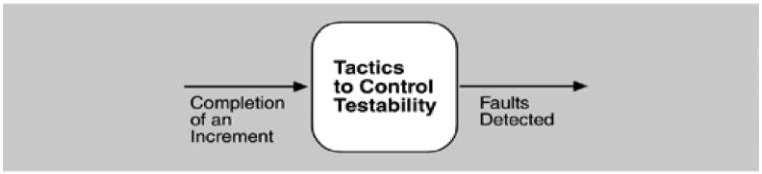


Figure 5.9. Summary of tactics for security

3.13 TESTABILITY TACTICS

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.



INPUT/OUTPUT

- ***Record/playback***.Record/playback refers to both capturing information crossing an interface andusing it as input into the test harness. The information crossing an interface during normal operation is saved in some repository. Recording this information allows test input for one of the components to be generated and test output for later comparison to be saved.
- ***Separate interface from implementation.*** Separating the interface from the implementation allowssubstitution of implementations for various testing purposes. Stubbing implementations allows the remainder of the system to be tested in the absence of the component being stubbed.
- ***Specialize access routes/interfaces.*** Having specialized testing interfaces allows the capturing orspecification of variable values for a component through a test harness as well as independently from its normal execution. Specialized access routes and interfaces should be kept separate from the access routes and interfaces for required functionality.

INTERNAL MONITORING

- ***Built-in monitors***.The component can maintain state, performance load, capacity, security, or otherinformation accessible through an interface. This interface can be a permanent interface of the component or it can be introduced temporarily. A common technique is to record events when monitoring states have been activated. Monitoring states can actually increase the testing effort since tests may have to be repeated with the monitoring turned off. Increased visibility into the activities of the component usually more than outweigh the cost of the additional testing.

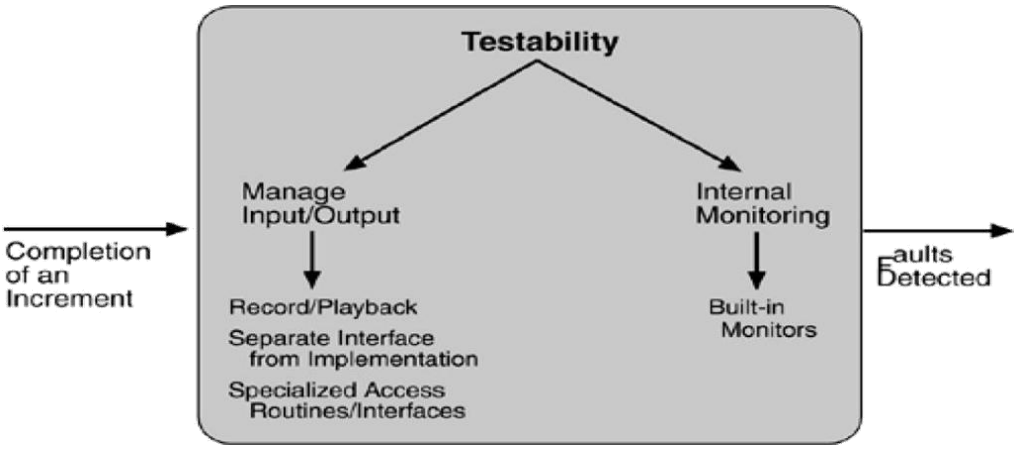
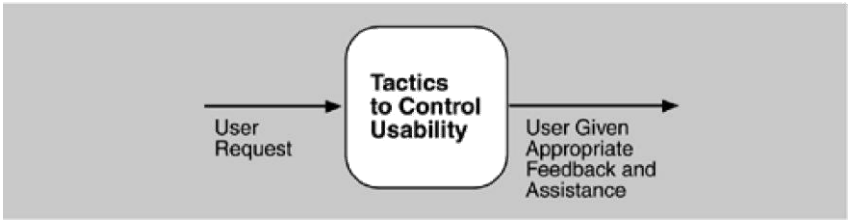


Figure 3.13. Summary of testability tactics

3.14 USABILITY TACTICS



RUNTIME TACTICS

- ***Maintain a model of the task.*** In this case, the model maintained is that of the task. The task model isused to determine context so the system can have some idea of what the user is attempting and provide various kinds of assistance. For example, knowing that sentences usually start with capital letters would allow an application to correct a lower-case letter in that position.

Maintain a model of the user.In this case, the model maintained is of the user. It determines the user'sknowledge of the system, the user's behavior in terms of expected response time, and other aspects specific to a user or a class of users. For example, maintaining a user model allows the system to pace scrolling so that pages do not fly past faster than they can be read.

Maintain a model of the system.In this case, the model maintained is that of the system. It determinesthe expected system behavior so that appropriate feedback can be given to the user. The system model predicts items such as the time needed to complete current activity.

DESIGN-TIME TACTICS

- ***Separate the user interface from the rest of the application***. Localizing expected changes is therationale for semantic coherence. Since the user interface is expected to change frequently both during the development and after deployment, maintaining the user interface code separately will localize changes to it. The software architecture patterns developed to implement this tactic and to support the modification of the user interface are:

  Model-View-Controller

  Presentation-Abstraction-Control

  Arch/Slinky

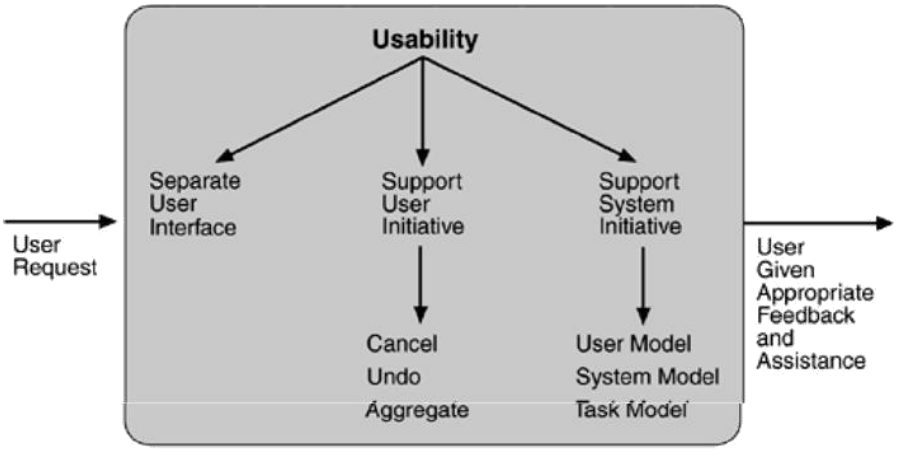


Figure 3.14. Summary of runtime usability tactics