

SYLLABUS

UNIT I: ENVISIONING ARCHITECTURE

What is software Architecture-What is Software Architecture, Other Points of View, Architectural Patterns, Reference Models, and Reference Architectures, Importance of Software Architecture, Architectural Structures and views.

ENVISIONING ARCHITECTURE: Architecture Business Cycle- Architectures influences, Software Processes and the Architecture Business Cycle, Making of —Good Architecture.

UNIT II: DESIGNING THE ARCHITECTURE WITH STYLES

Designing the Architecture: Architecture in the Life Cycle, Designing the Architecture, Formatting the Team Structure, Creating a Skeletal System.

Architecture Styles: Architectural Styles, Pipes and Filters, Data Abstraction and ObjectOriented Organization, Event-Based, Implicit Invocation, Layered Systems, Repositories, Interpreters.2013-2014

UNIT III: CREATING AN ARCHITECTURE-I

Creating an Architecture: Understanding Quality Attributes – Functionality and Architecture, Architecture and Quality Attributes, System Quality Attributes, Quality Attribute. Scenarios in Practice, Other System Quality Attributes, Business Qualities, Architecture Qualities.

Achieving Qualities: Introducing Tactics, Availability Tactics, Modifiability Tactics, Performance Tactics, Security Tactics, Testability Tactics, Usability Tactics.

UNIT IV: CREATING AN ARCHITECTURE-II

Documenting Software Architectures: Use of Architectural Documentation, Views, Choosing the Relevant Views, Documenting a view, Documentation across Views.

Reconstructing Software Architecture: Introduction, Information Extraction, Database Construction, View Fusion, and Reconstruction.

UNIT V: ANALYZING ARCHITECTURES

The ATAM: Participants in the ATAM, Outputs of The ATAM, Phases Of the ATAM. **The CBAM:** Decision-Making Context, The Basis for the CBAM, Implementing the CBAM. **The World Wide Web:** A Case study in Interoperability- Relationship to the Architecture Business Cycle, Requirements and Qualities, Architecture Solution, Achieving Quality Goals.

TEXT BOOKS:

1. Software Architectures in Practice , Len Bass, Paul Clements, Rick Kazman, 2nd Edition, Pearson Publication.
2. Software Architecture , Mary Shaw and David Garlan, First Edition, PHI Publication, 1996.

UNIT-1

ENVISIONING ARCHITECTURE

1.WHAT IS SOFTWARE ARCHITECTURE ?

WHAT SOFTWARE ARCHITECTURE IS AND WHAT IT ISN'T

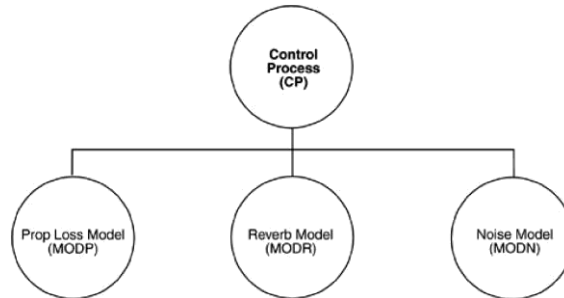


Fig taken from a system description for an underwater acoustic simulation, purports to describe that system's "top-level architecture" and is precisely the kind of diagram most often displayed to help explain an architecture. Exactly what can we tell from it?

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 each other.

All of the elements apparently have some sort of relationship with each other, since the diagram is fully connected.

Is this an architecture? 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 something else?

What are the responsibilities of the elements?

What is it they do? What is their function in the system?

What is the significance of the connections?

Do the connections mean that the elements 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 call the 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?

This diagram does not show a software architecture. 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.

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.

The definition is indifferent as to whether the architecture for a system is a good one or a bad one, meaning that it will allow or prevent the system from meeting its behavioral, performance, and life-cycle requirements

2. 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.

3. 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.

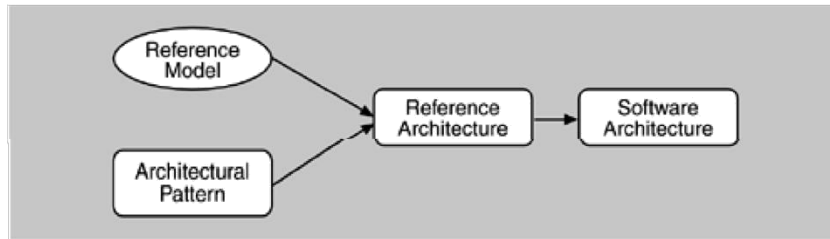
For ex: client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients.

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

4. 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.

5. 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.

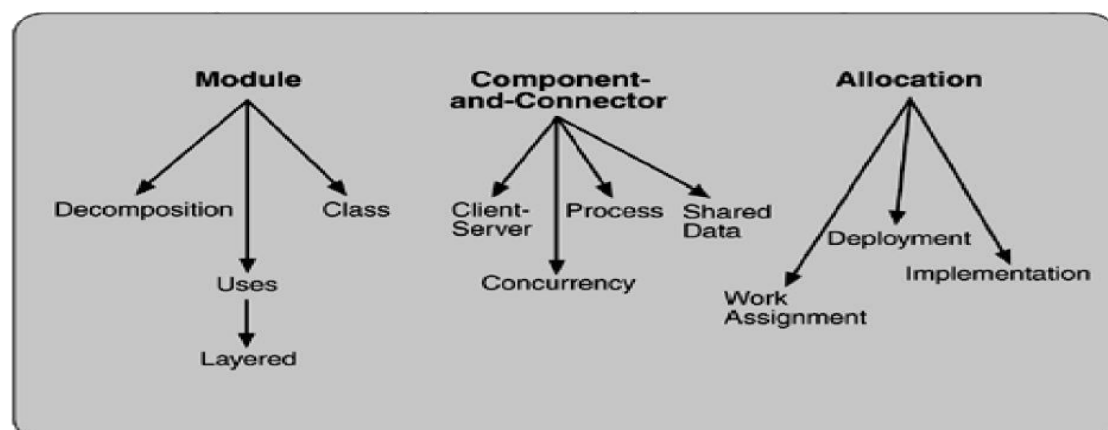
Module structures.

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? How can the system's structure change as it executes?

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

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

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

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

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.

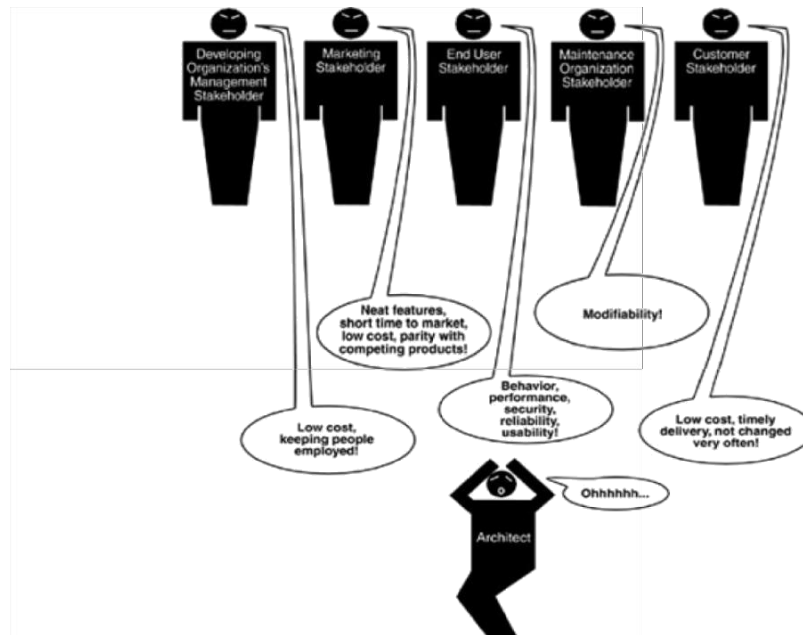
1.1 WHERE DO ARCHITECTURES COME FROM?

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

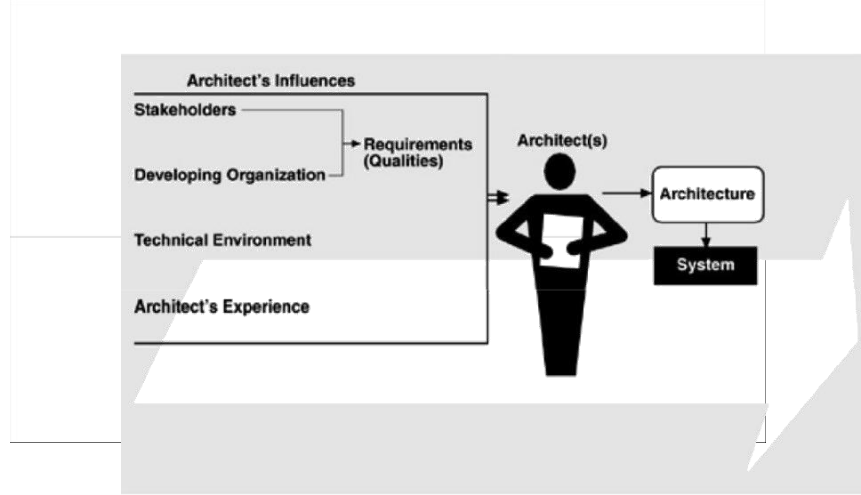
A special case of the architect's background and experience is reflected by the *technical environment*.

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.

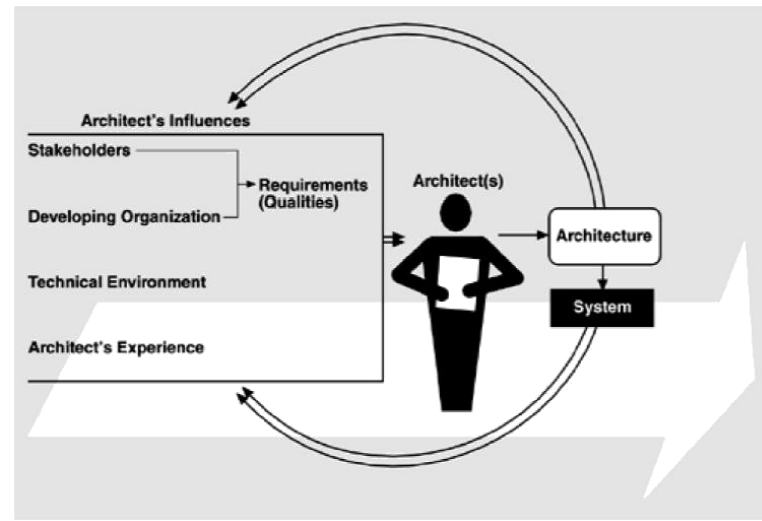
THE ARCHITECTURE AFFECTS THE FACTORS THAT INFLUENCE THEM

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:

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.

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.

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.

The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base.

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.

3 .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 product (or structural) recommendations.

Process recommendations 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.

Product (structural) recommendations 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.

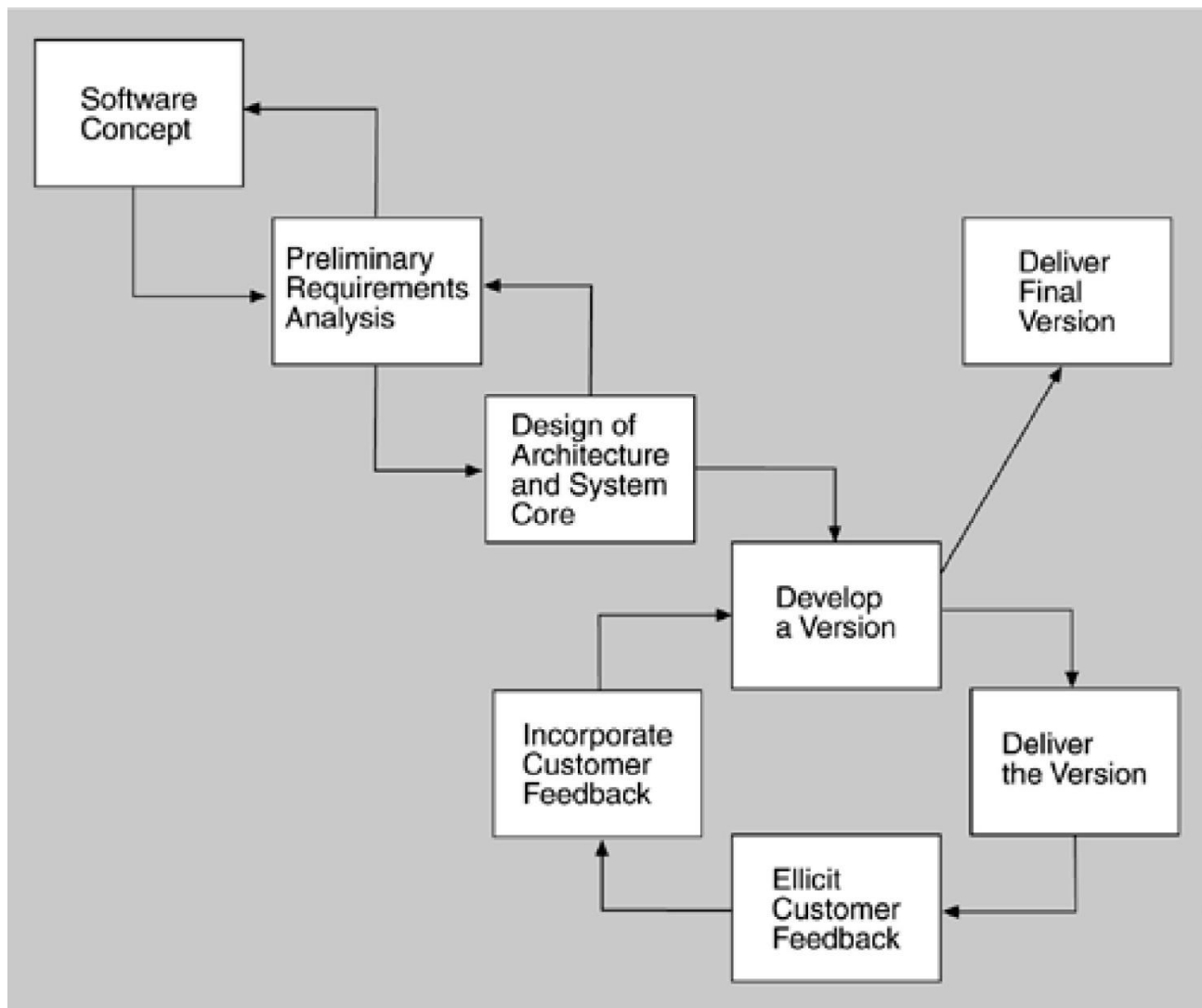
UNIT- II

DESIGNING THE ARCHITECTURE WITH STYLES

1. 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



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.
- The architecture of the A-7E discussed shaped by its modifiability and performance requirements.
- The architecture for the air traffic control system is shaped by its availability requirements, we will see an architecture shaped by performance and modifiability 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
- From this list, choose the ones that will have the most impact on the architecture. These are the architectural drivers, and there should be fewer than ten.
- The Architecture Tradeoff Analysis Method uses a *utility tree* to help turn the business drivers into quality scenarios.
- Once the architectural drivers are known, the architectural design can begin. The requirements analysis process will then be influenced by the questions generated during architectural design—one of the reverse-direction arrows.

Designing the Architecture

- In this section we describe a method for designing an architecture to satisfy both quality requirements and functional requirements.
- We call this method 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.
- 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 and implementation. Incorporating ADD into it involves modifying the steps dealing with the high-level design of the architecture and then following the 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.

- There are a number of different design processes that could be created using the general scenarios and the tactics and pattern.
- Each process assumes different things about how to "chunk" the design work and about the essence of the design process.
- We discuss ADD in some detail to illustrate how we are applying the general scenarios and tactics, and hence how we are "chunking" the work, and what we believe is the essence of the design process.
- We demonstrate the ADD method by using it to design a product line architecture for a garage door opener within a 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.

Sample Input

- The input to ADD is a set of requirements. ADD assumes functional requirements (typically expressed as use cases) and constraints as input, as do other design methods.
- However, in ADD, we differ from those methods in our treatment of *quality requirements*. ADD mandates that quality requirements be expressed as a set of system-specific quality scenarios
- System-specific scenarios should be defined to the detail necessary for the application. In our examples, we omit several portions of a fully fleshed scenario since these portions do not contribute to the design process.

For our garage door example, the quality scenarios include the following:

- The device and controls for opening and closing the door are different for the various products in the product line, as already mentioned.
- They may include controls from within a home information system.
- The product architecture for a specific set of controls should be directly derivable

from the product line architecture.

- The processor used in different products will differ. The product architecture for each specific processor should be directly derivable from the product line architecture.
- If an obstacle (person or object) is detected by the garage door during descent, it must halt (alternately re-open) within 0.1 second.
- The garage door opener should be accessible for diagnosis and administration from within the home information system using a product-specific diagnosis protocol. It should be possible to directly produce an architecture that reflects this protocol.

Beginning ADD

We have already introduced architectural drivers. ADD depends on the identification of the drivers and can start as soon as all of them are known. Of course, during the design the determination of which architectural drivers are key may change either as a result of better understanding of the requirements or as a result of changing requirements. Still, the process can begin when the driver requirements are known with some assurance.

In the following section we discuss ADD itself.

ADD Steps

We begin by briefly presenting the steps performed when designing an architecture using the ADD method. We will then discuss the steps in more detail

Choose the module to decompose. The module to start with is usually the whole system. All required inputs for this module should be available (constraints, functional requirements, quality requirements).

Refine the module according to these steps:

Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.

Choose an architectural pattern that satisfies the architectural drivers. Create (or select) the pattern based on the tactics that can be used to achieve the drivers. Identify child modules required to implement the tactics.

Instantiate modules and allocate functionality from the use cases and represent using multiple views.

Define interfaces of the child modules. The decomposition provides modules and constraints on the types of module interactions. Document this information in the interface document for each module.

Verify and refine use cases and quality scenarios and make them constraints for the child modules. This step verifies that nothing important was forgotten and prepares the child modules for further decomposition or implementation.

Repeat the steps above for every module that needs further decomposition.

Choose the Module to Decompose

The following are all modules: system, subsystem, and sub module. The decomposition typically starts with the system, which is then decomposed into subsystems, which are further decomposed into sub modules.

In our example, the garage door opener is the system. One constraint at this level is that the opener must interoperate with the home information system.

Choose the Architectural Drivers

- As we said, architectural drivers are the combination of functional and quality requirements that "shape" the architecture or the particular module under consideration.
- The drivers will be found among the top-priority requirements for the module.
- In our example, the four scenarios we have shown are architectural drivers. In the

systems on which this example is based, there were dozens of quality scenarios. In examining them, we see a requirement for real-time performance, and modifiability to support product lines.

- We also see a requirement that online diagnosis be supported. All of these requirements must be addressed in the initial decomposition of the system.
- The determination of architectural drivers is not always a top-down process. Sometimes detailed investigation is required to understand the ramifications of particular requirements.
- For example, to determine if performance is an issue for a particular system configuration, a prototypical implementation of a piece of the system may be required. In our example, determining that the performance requirement is an architectural driver requires examining the mechanics of a garage door and the speed of the potential processors.
- We will base our decomposition of a module on the architectural drivers.

Other requirements apply to that module, but, by choosing the drivers, we are reducing the problem to satisfying the most important ones.

- We do not treat all of the requirements as equal; the less important requirements are satisfied within the constraints of the most important.
- This is a significant difference between ADD and other architecture design methods.

Choose an Architectural Pattern

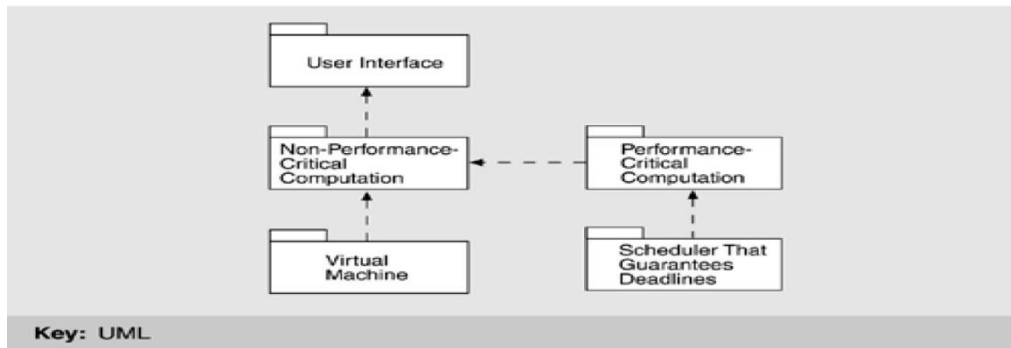
- For each quality there are identifiable tactics (and patterns that implement these tactics) that can be used in an architecture design to achieve a specific quality.
- Each tactic is designed to realize one or more quality attributes, but the patterns in which they are embedded have an impact on other quality attributes. In an architecture design, a composition of many such tactics is used to achieve a balance between the required multiple qualities.
- Achievement of the quality and functional requirements is analyzed during the refinement step.

- The goal of step 2b is to establish an overall architectural pattern consisting of module types. The pattern satisfies the architectural drivers and is constructed by composing selected tactics.
- Two main factors guide tactic selection. The first is the drivers themselves. The second is the side effects that a pattern implementing a tactic has on other qualities.
- For example, a classic tactic to achieve modifiability is the use of an interpreter. Adding an interpreted specification language to a system simplifies the creation of new functions or the modification of existing ones. Macro recording and execution is an example of an interpreter. HTML is an interpreted language that specifies the look-and-feel of Web pages.
- An interpreter is an excellent technique for achieving modifiability at runtime, but it has a strong negative influence on performance. The decision to use one depends on the relative importance of modifiability versus performance. A decision may be made to use an interpreter for a portion of the overall pattern and to use other tactics for other portions.
- ." We choose one example of each: "increase computational efficiency" and "choose scheduling policy." This yields the following tactics:

Semantic coherence and information hiding. Separate responsibilities dealing with the user interface, communication, and sensors into their own modules. We call these modules *virtual machines* and we expect all three to vary because of the differing products that will be derived from the architecture. Separate the responsibilities associated with diagnosis as well.

Increase computational efficiency. The performance-critical computations should be made as efficient as possible.

Schedule wisely. The performance-critical computations should be scheduled to ensure the achievement of the timing deadline.



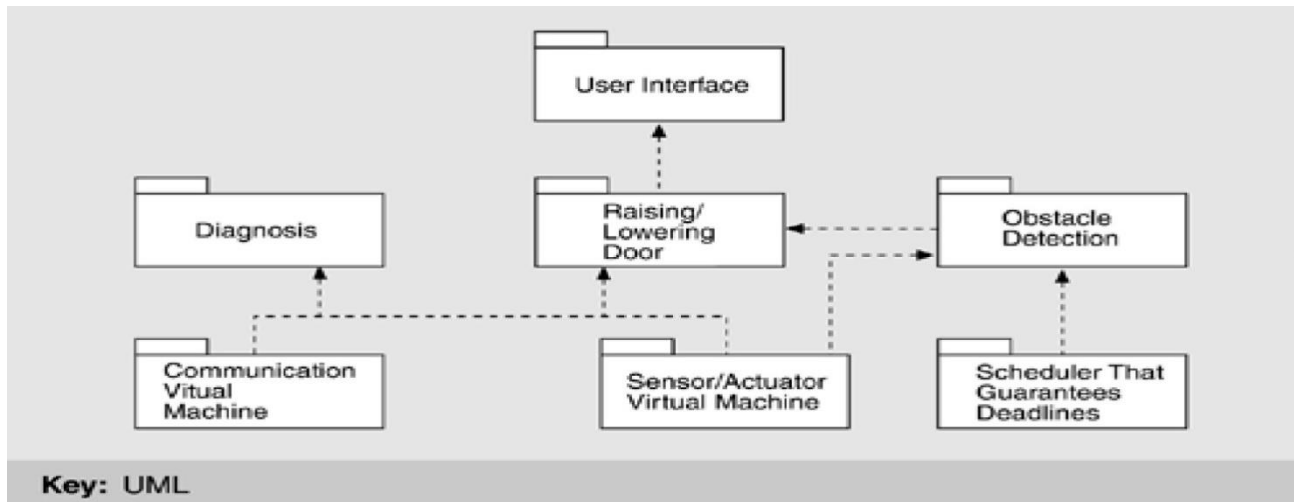
Instantiate Modules and Allocate Functionality Using Multiple Views

- In the preceding section, we discussed how the quality architectural drivers determine the decomposition structure of a module via the use of tactics.
- As a matter of fact, in that step we defined the module types of the decomposition step. We now show how those module types will be instantiate.

Instantiate modules

- we identified a non-performance-critical computation running on top of a virtual machine that manages communication and sensor interactions.
- The software running on top of the virtual machine is typically an application. In a concrete system we will normally have more than one module.
- There will be one for each "group" of functionality; these will be instances of the types shown in the pattern. Our criterion for allocating functionality is similar to that used in functionality-based design methods, such as most object-oriented design methods.

- For our example, we allocate the responsibility for managing obstacle detection and halting the garage door to the performance-critical section since this functionality has a deadline.
- The management of the normal raising and lowering of the door has no timing



deadline, and so we treat it as non-

We also identify several responsibilities of the virtual machine: communication and sensor reading and actuator control.

The result of this step is a plausible decomposition of a module. The next steps verify how well the decomposition achieves the required functionality.

Allocate functionality

- Applying use cases that pertain to the parent module helps the architect gain a more detailed understanding of the distribution of functionality.
- This also may lead to adding or removing child modules to fulfill all the functionality required. At the end, every use case of the parent module must be representable by a sequence of responsibilities within the child modules.
- Assigning responsibilities to the children in a decomposition also leads to the discovery of necessary information exchange. This creates a producer/consumer relationship between those modules, which needs to be recorded. At this point in the design, it is not important to define how the information is exchanged.

- Is the information pushed or pulled? Is it passed as a message or a call parameter? These are all questions that need to be answered later in the design process.
- At this point only the information itself and the producer and consumer roles are of interest.
- This is an example of the type of information left unresolved by ADD and resolved during detailed design.

- Some tactics introduce specific patterns of interaction between module types. A tactic using an intermediary of type publish-subscribe
- for example, will introduce a pattern, "Publish" for one of the modules and a pattern "Subscribe" for the other. These patterns of interaction should be recorded since they translate into responsibilities for the affected modules

Represent the architecture with views

- we introduced a number of distinct architectural views. In our experience with ADD, one view from each of the three major groups of views (module decomposition, concurrency, and deployment) have been sufficient to begin with.
- The method itself does not depend on the particular views chosen, and if there is a need to show other aspects, such as runtime objects, additional views can be introduced. 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.

The concurrency view is one of the component-and-connector views. The components are

instances of the modules in the module decomposition view, and the connectors are the carriers of *virtual threads*.

A "virtual thread" describes an execution path through the system or parts of it. This should not be confused with operating system threads (or processes), which implies other properties like memory/processor allocation.

Those properties are not of interest on the level at which we are designing. Nevertheless, after the decisions on an operating system and on the deployment of modules to processing units are made, virtual threads have to be mapped onto operating system threads. This is done during detailed design.

The connectors in a concurrency view are those that deal with threads such as "synchronizes with," "starts," "cancels," and "communicates with." A concurrency view shows instances of the modules in the module decomposition view as a means of understanding the mapping between those two views.

It is important to know that a synchronization point is located in a specific module so that this responsibility can be assigned at the right place.

To understand the concurrency in a system, the following use cases are illuminating:

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 a switch.

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 the door.

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 and closing?

Shutting down the system. This helps to uncover issues of cleaning up, such as achieving and saving a consistent system state.

In our example, we can see a point of synchronization in the sensor/actuator virtual machine. The performance-critical section must sample the sensor, as must the raising/lowering door section. It is plausible that the performance-critical section will interrupt the sensor/actuator virtual machine when it is performing an action for the raising/lowering door section.

We need a synchronization mechanism for the sensor/actuator virtual machine. We see this by examining the virtual thread for the performance-critical section and the virtual thread for the raising/lowering door section, and observing that these two threads both involve the sensor/actuator virtual machine.

The crossing of two virtual threads is an indication that some synchronization mechanism should be employed.

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.

The deployment view also helps in deciding if multiple instances of some modules are needed. For example, a reliability requirement may force us to duplicate critical functionality on different processors.

A deployment view also supports reasoning about the use of special-purpose hardware.

The derivation of the deployment view is not arbitrary. As with the module decomposition and concurrency views, the architecture drivers help determine the allocation of components to hardware.

Tactics such as replication offer a means to achieve high performance or reliability by deploying replicas on different processors. Other tactics such as a real-time scheduling mechanism actually prohibit deployment on different processors.

Functional considerations usually guide the deployment of the parts that are not predetermined by the selected tactics.

The crossing of a virtual thread from one processor to another generates responsibilities for different modules.

It indicates a communication requirement between the processors. Some module must be responsible for managing the communication; this responsibility must be recorded in the module decomposition view.

Forming the Team Structure

- Once the first few levels of the architecture's module decomposition structure are fairly stable, those modules can be allocated to development teams. The result is the work assignment view .
- This view will either allocate modules to existing development units or define new ones.
- Take any two nodes x and y of the system. Either they are joined by a branch or they are not. (That is, either they communicate with each other in some way meaningful to the operation of the system or they do not.)
- If there is a branch, then the two (not necessarily distinct) design groups X and Y which designed the two nodes must have negotiated and agreed upon an interface specification to permit communication between the two corresponding nodes of the design organization.
- If, on the other hand, there is no branch between x and y , then the subsystems do not communicate with each other, there was nothing for the two corresponding design groups to negotiate, and therefore there is no branch between X and Y .

The impact of an architecture on the development of organizational structure is clear.

Once an 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 (or a system-wide set of practices is adopted).
- For large systems, the teams may belong to different subcontractors. The work practices may include items such as bulletin boards and Web pages for communication, naming conventions for files, and the configuration control system. All of these may be different from group to group, again especially for large systems. Furthermore, quality assurance and testing procedures are set up for each group, and each group needs to establish liaisons and coordinate with the other groups.
- Why does the team structure mirror the module decomposition structure? Information hiding, the design principle behind the module decomposition structure of systems, holds that modules should encapsulate, or hide, changeable details by putting up an interface that abstracts away the changeable aspects and presents a common, unified set of services to its users (in this case, the software in other system modules).
- This implies that each module constitutes its own small domain; we use *domain* here to mean an area of specialized knowledge or expertise.

This makes for a natural fit between teams and modules of the decomposition structure, as the following examples show.

- The module is a user interface layer of a system. The application programming interface that it presents to other modules is independent of the particular user interface devices (radio buttons, dials, dialog boxes, etc.) that it uses to present information to the human user, because those might change.
- The domain here is the repertoire of such devices.
- The module is a process scheduler that hides the number of available processors and the scheduling algorithm. The domain here is process scheduling and the list of appropriate algorithms.

- It encapsulates the equations that compute values about the physical environment.
- The domain is numerical analysis (because the equations must be implemented to maintain sufficient accuracy in a digital computer) and avionics.
- Recognizing modules as mini-domains immediately suggests that the most effective use of staff is to assign members to teams according to their expertise. Only the module structure permits this.
- As the sidebar Organizational and Architectural Structures discusses, organizations sometimes also add specialized groups that are independent of the architectural structures.

Creating a Skeletal System

- Once an architecture is sufficiently designed and teams are in place to begin building to it, a skeletal system can be constructed.
- The idea at this stage is to provide an underlying capability to implement a system's functionality in an order advantageous to the project.
- Classical software engineering practice recommends "stubbing out" sections of code so that portions of the system can be added separately and tested independently.
- However, which portions should be stubbed? By using the architecture as a guide, a sequence of implementation becomes clear.
- First, implement the software that deals with the execution and interaction of architectural components.
- This may require producing a scheduler in a real-time system, implementing the rule engine (with a prototype set of rules) to control rule firing in a rule-based system, implementing process synchronization mechanisms in a multi-process system, or implementing client-server coordination in a client-server system.
- Often, the basic interaction mechanism is provided by third-party middleware, in which case the job becomes ones of installation instead of implementation.
- On top of this communication or interaction infrastructure, you may wish to install the simplest of functions, one that does little more than instigate some rote behavior.

- At this point, you will have a running system that essentially sits there and hums to itself—but a running system nevertheless.
- This is the foundation onto which useful functionality can be added.
- Even the stubbed-out parts help pave the way for completion. These stubs adhere to the same interfaces that the final version of the system requires, so they can help with understanding and testing the interactions among components even in the absence of high-fidelity functionality.
- These stub components can exercise this interaction in two ways, either producing hardcoded canned output or reading the output from a file.
- They can also generate a synthetic load on the system to approximate the amount of time the actual processing will take in the completed working version.
- This aids in early understanding of system performance requirements, including performance interactions and bottlenecks.

PART-2

ARCHITECTURAL STYLES

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 FILTER

- 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.

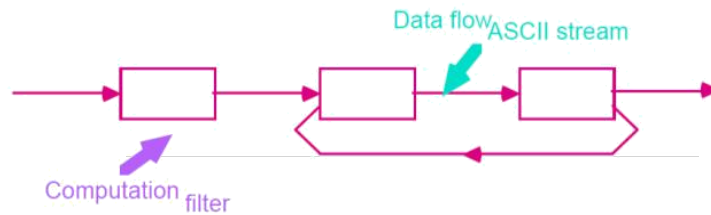


Figure 1: Pipes and Filters

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 existing 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.

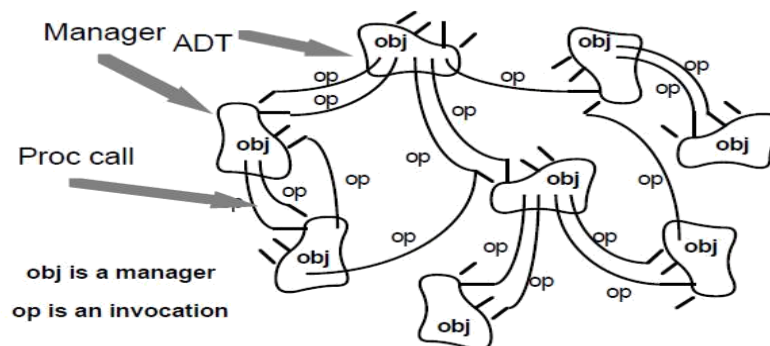


Figure 2: Abstract Data Types and Objects

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 issue

LAYERED SYSTEMS:

- A layered system is organized hierarchically
- Each layer provides service to the layer above it.

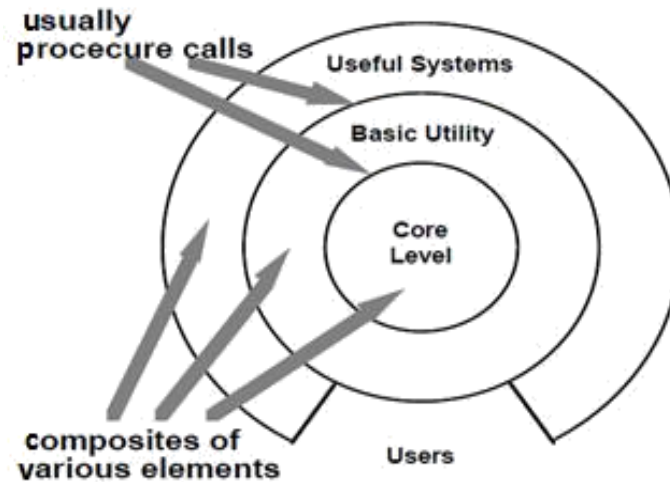


Figure 3: Layered Systems

- 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.

Examples

- Layered communication protocol
- Operating system
- Database system.

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 centered 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

Blackboard:

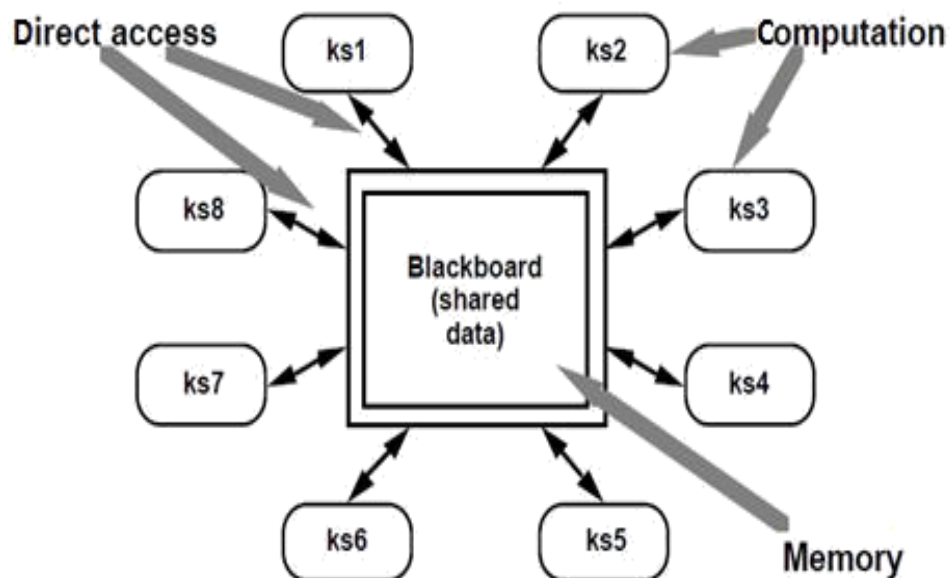


Figure 4: The Blackboard

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.
- 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

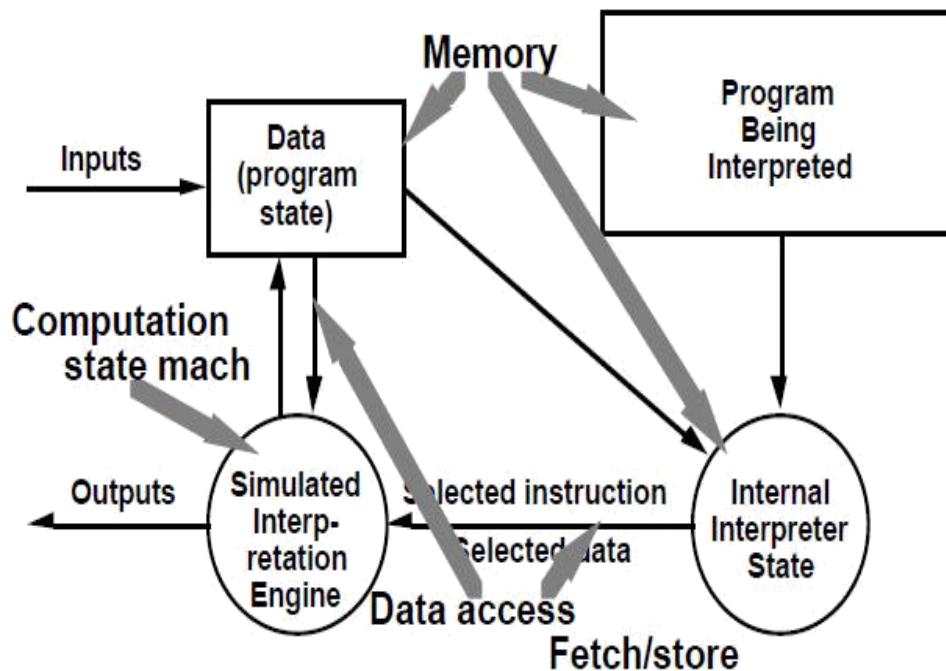


Figure 5: Interpreter

- 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”

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

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

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.

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) that generated the stimulus.

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

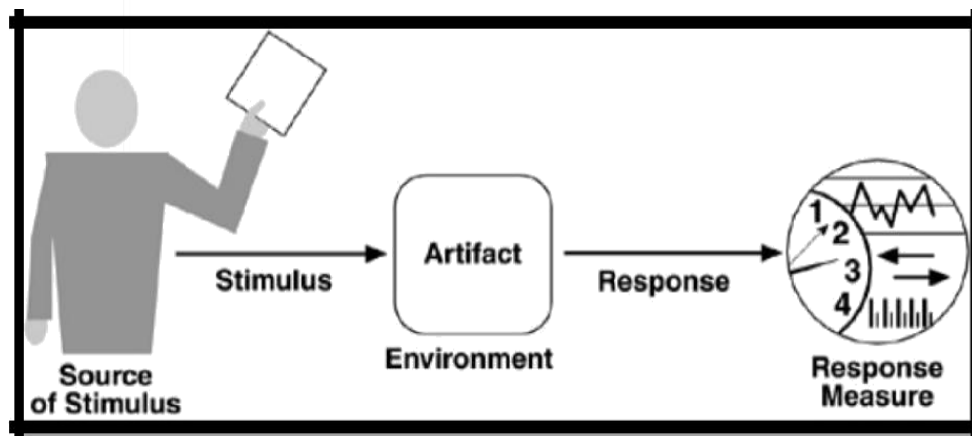
Environment. The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.

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 4.1 shows the parts of a quality attribute scenario.



Quality attribute scenarios in practice

AVAILABILITY SCENARIO

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

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

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

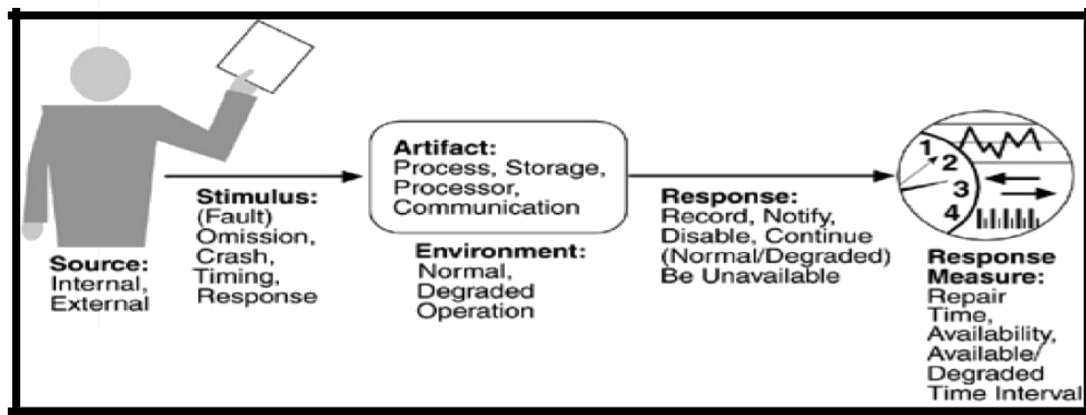
Stimulus. A fault of one of the following classes occurs.

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 system response. 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.

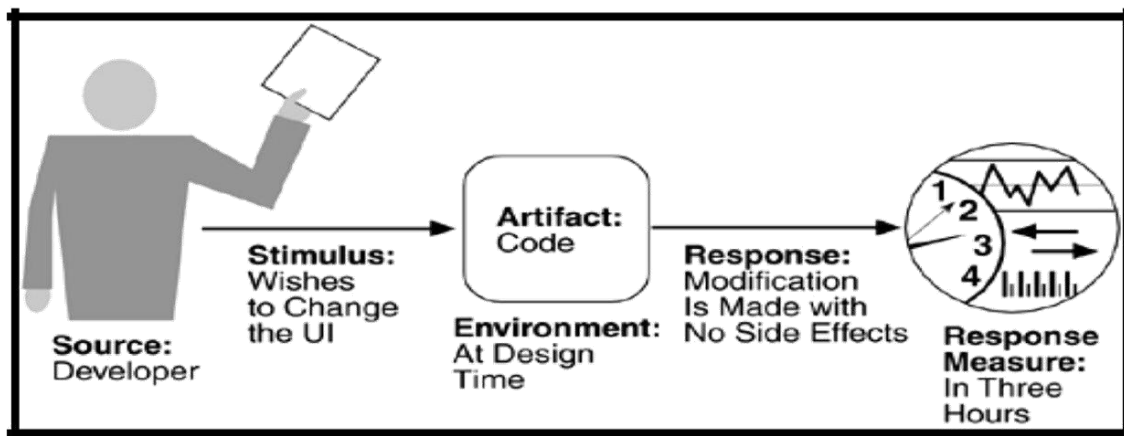
Response measure. The response measure can specify an availability percentage, or it can specify a time to repair, times during which the system must be available, or the duration for which the system must be available

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, or an 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. 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, the modification 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.

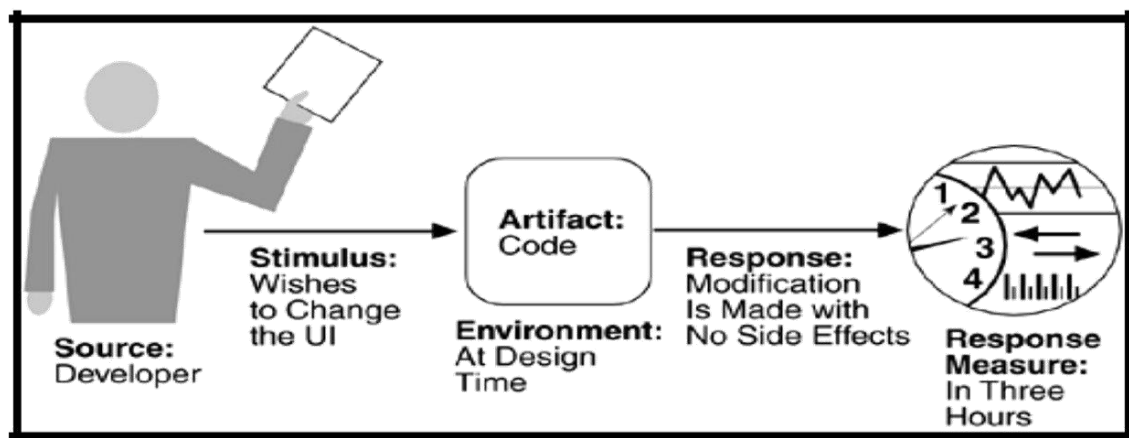
Artifact. This portion specifies what is to be changed—the functionality of a system, its platform, its user interface, its environment, or another system with which it interoperates. The modification is to the user interface.

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 most desirable 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.



Source of stimulus. This portion specifies who makes the changes—the developer, a system administrator, or an 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, the modification 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 user interface, 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.

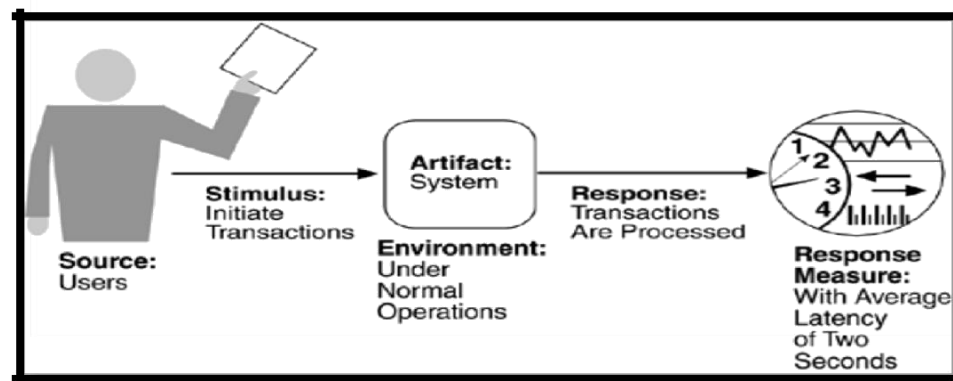
Response. Whoever makes the change must understand how to make it, and then make it, test it and deploy it.

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 most desirable 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 our example, 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, or sporadic. 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 our example, 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 a deadline 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. This means 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 your grade has not been changed since your instructor assigned it.

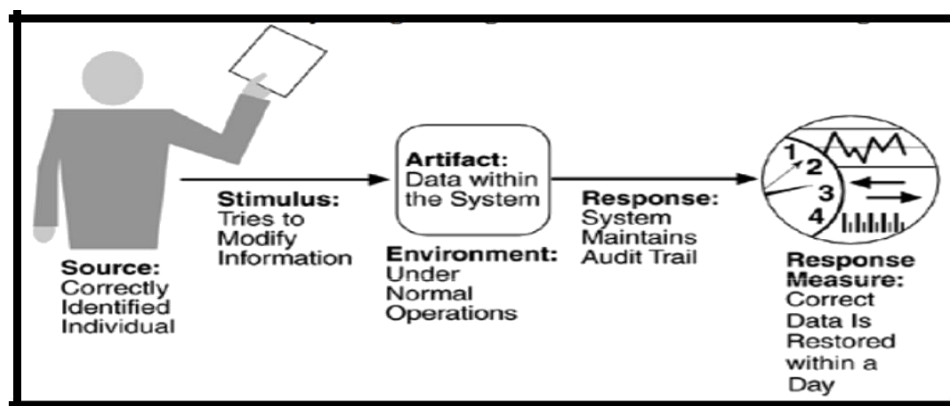
Assurance is the property that the parties to a transaction are who they purport to be. This means that, when a customer sends a credit card number to an Internet merchant, the merchant is who the customer thinks they are.

Availability is the property that the system will be available for legitimate use. This means that a denial-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 reconstruct them. 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 been previously 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 unauthorized person 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.

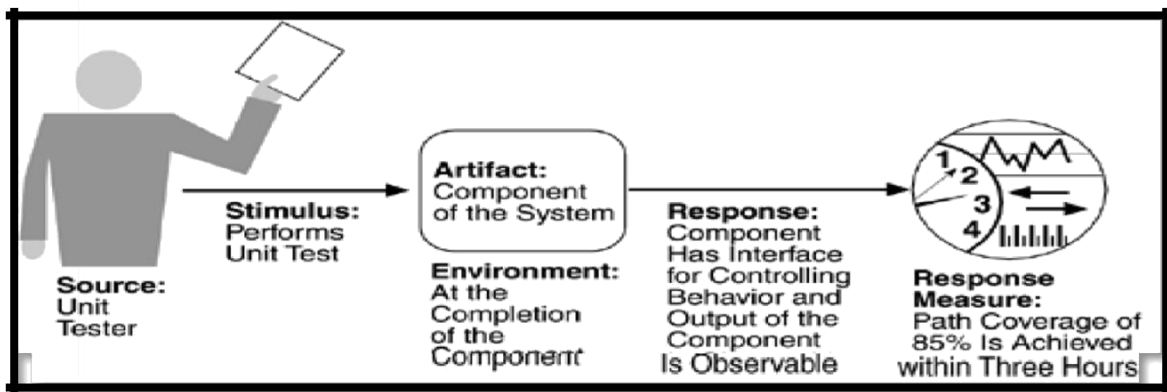
Environment. The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to the network.

Response. Using services without authorization or preventing legitimate users from using services is a different goal 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 the difficulty 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. A test 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 the completion 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 code is 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 the system 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 statement.

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 can the system do to make the task of learning easier?

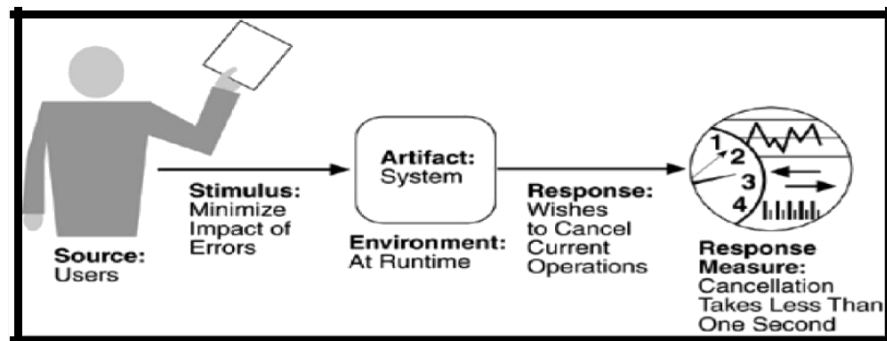
Using a system efficiently. What can the system do to make the user more efficient in its operation?

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 correct action 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, minimize the 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 system configuration 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. In our 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. It 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.

OTHER SYSTEM QUALITY ATTRIBUTES

SCALABILITY

PORTABILITY

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.

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's requirements 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 open to certain changes as development progresses.

PART-2 ACHIEVING QUALITIES

INTRODUCING TACTICS

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

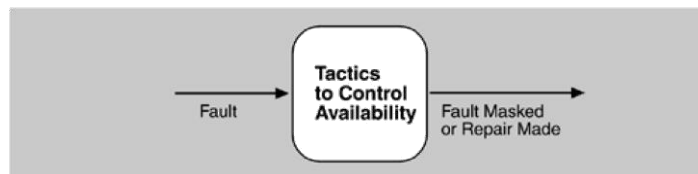


Figure 5.1. Tactics are intended to control responses to stimuli.

Tactics can refine other tactics. For each quality attribute that we discuss, we organize the tactics as a hierarchy.

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

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 and another 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 simple output 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. The response 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 (the primary) 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 to make 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 in response 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

Removal from service. This tactic removes a component of the system from operation to undergo some activities to prevent anticipated failures.

Transactions. A transaction is the bundling of several sequential steps such that the entire bundle can be 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 the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

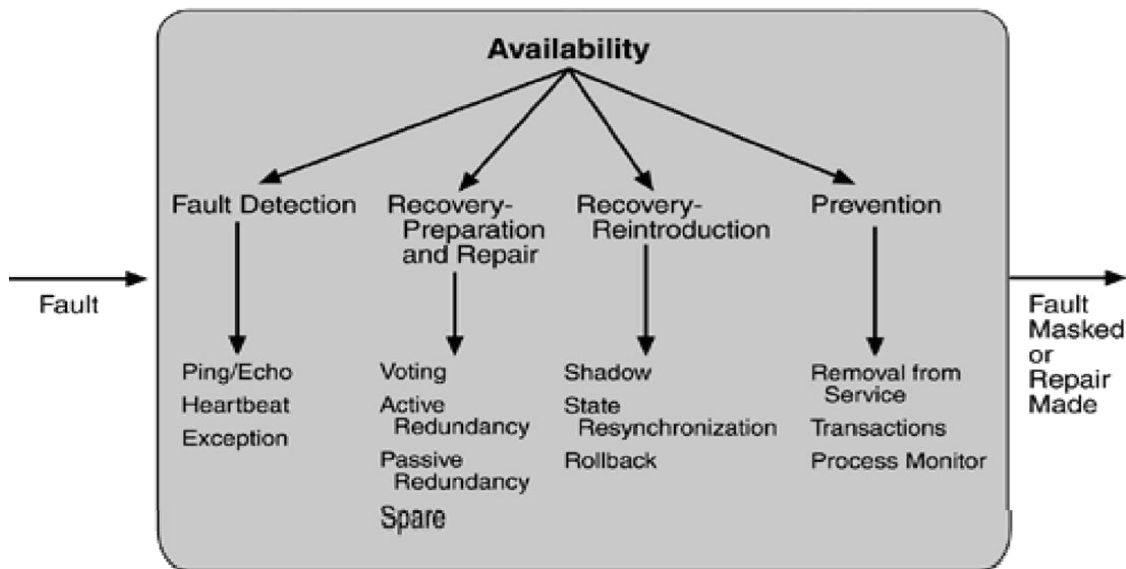
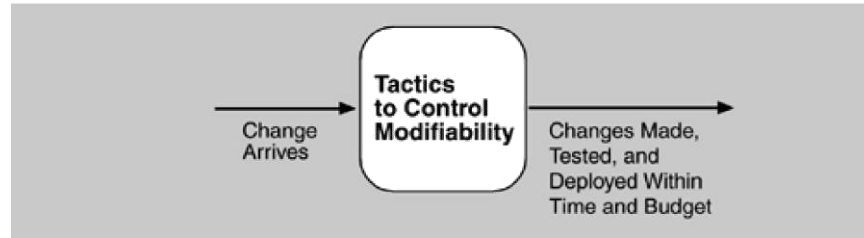


Figure 5.3. Summary of availability tactics

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/consumption introduces 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.

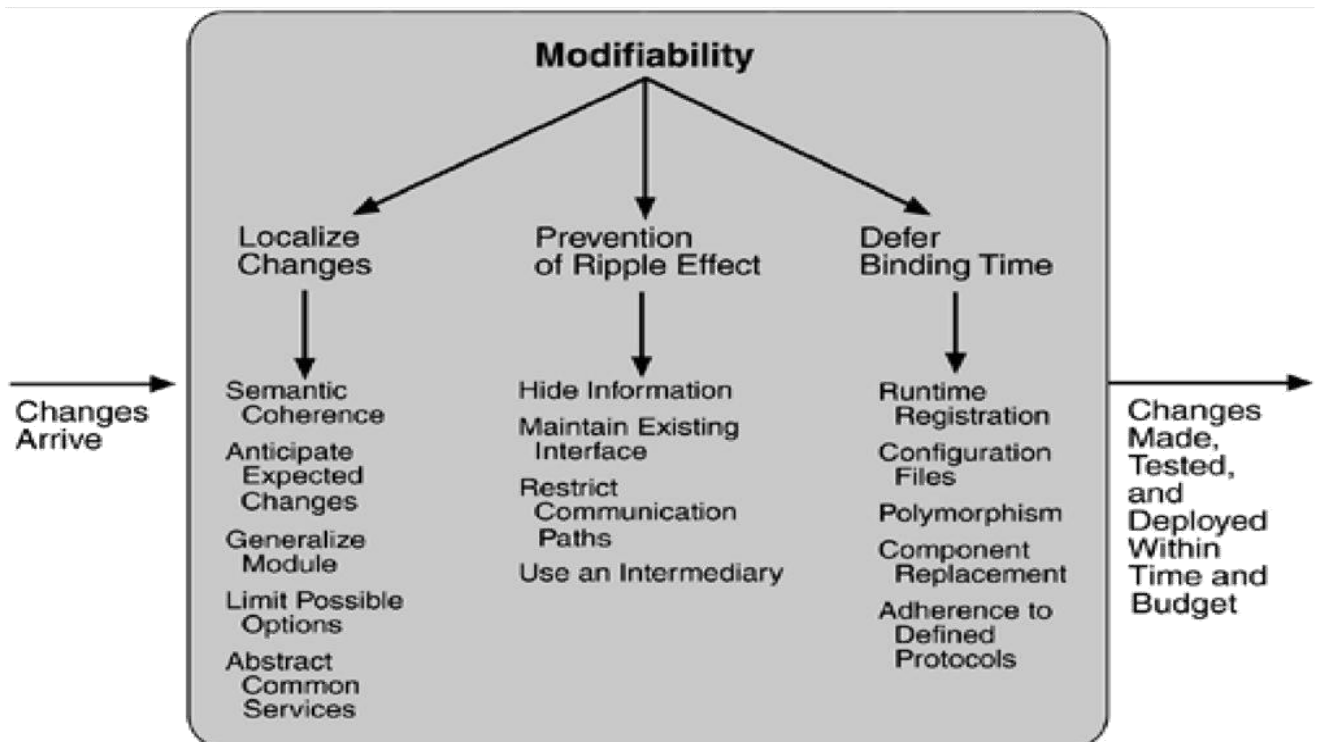
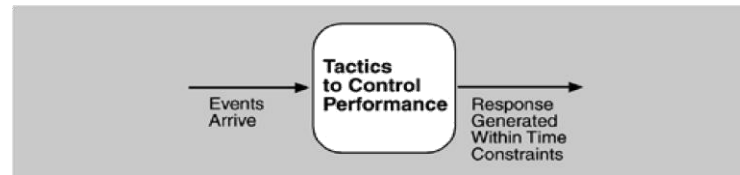


Figure 5.5. Summary of modifiability tactics

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 the network, 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, because the 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 the resources 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 faster networks 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

semantic importance. Each stream is assigned a priority statically according to some domain characteristic 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 every assignment possibility, assigns the resource to the next request in that order.

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

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

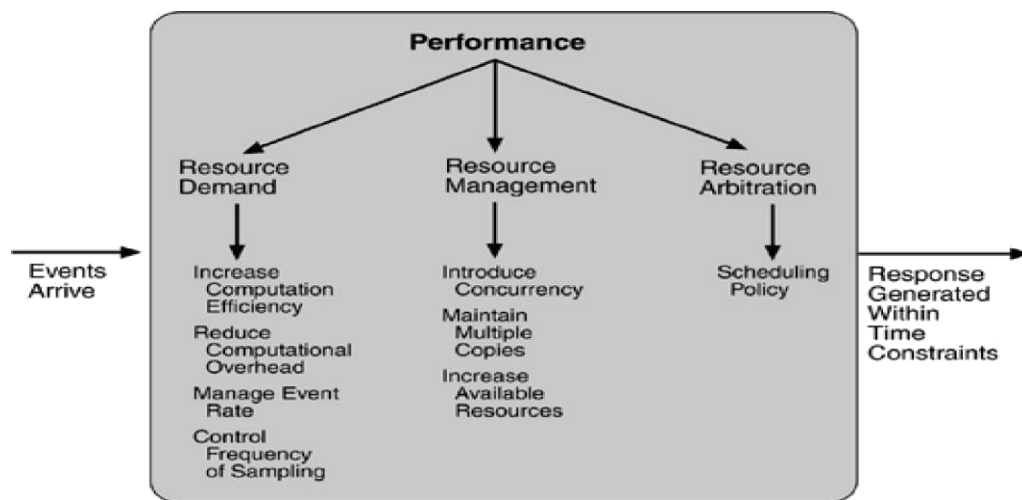
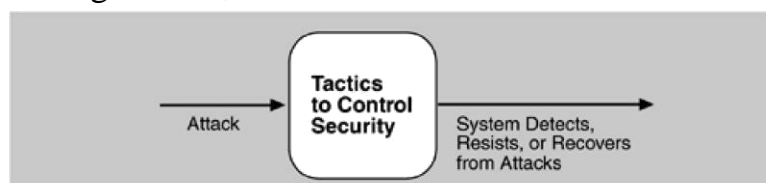


Figure 5.7. Summary of performance tactics

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 and modify 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 is usually 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 encoded in 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 services on 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 from unknown 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 trail*.



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.

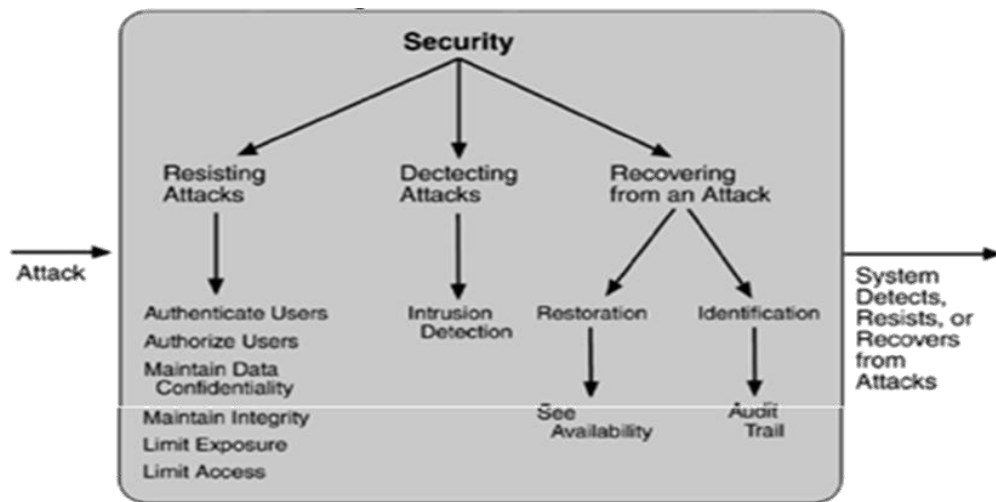
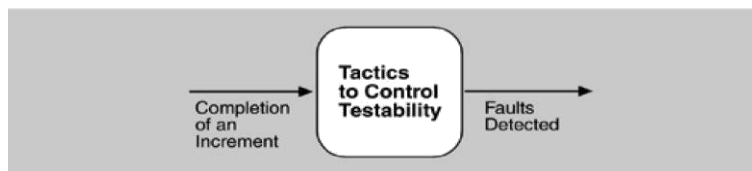


Figure 5.9. Summary of tactics for security

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 and using 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 allows substitution 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 or specification 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 other information 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.

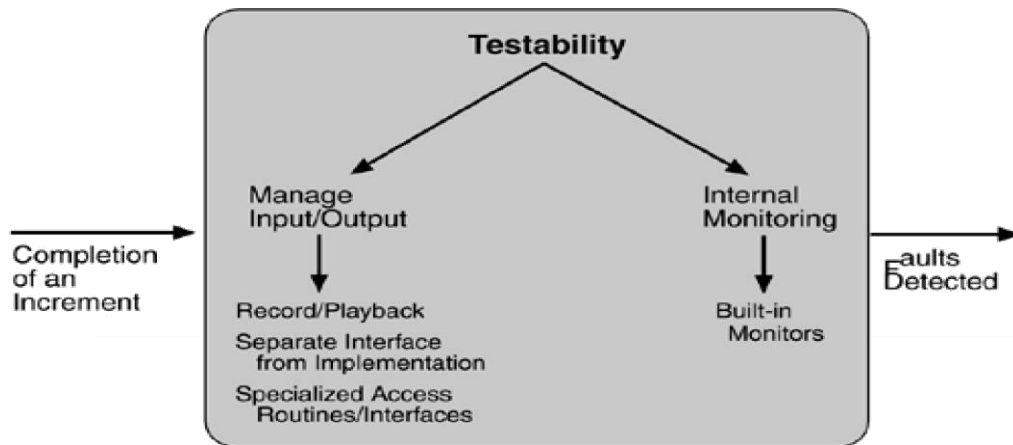
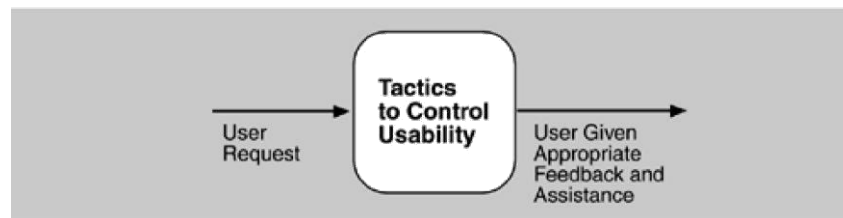


Figure 5.11. Summary of testability tactics

USABILITY TACTICS



RUNTIME TACTICS

Maintain a model of the task. In this case, the model maintained is that of the task. The task model is used 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's knowledge 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 determines the 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 the rationale 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
Seeheim

Arch/Slinky

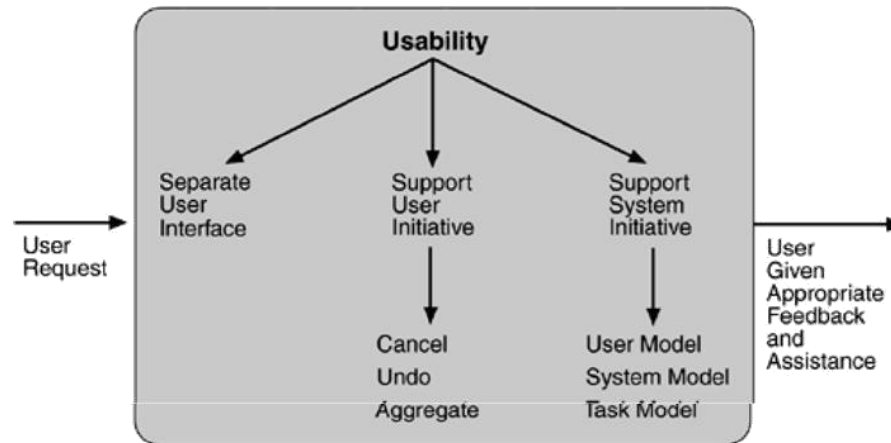


Figure 5.13. Summary of runtime usability tactics

RELATIONSHIP OF TACTICS TO ARCHITECTURAL PATTERNS

The pattern consists of six elements:

a **proxy**, which provides an interface that allows clients to invoke publicly accessible methods on an active object;

a **method request**, which defines an interface for executing the methods of an active object;

an **activation list**, which maintains a buffer of pending method requests;

a **scheduler**, which decides what method requests to execute next;

a **servant**, which defines the behavior and state modeled as an active object; and

a **future**, which allows the client to obtain the result of the method invocation.

The tactics involves the following:

Information hiding (modifiability). Each element chooses the responsibilities it will achieve and hides their achievement behind an interface.

Intermediary (modifiability). The proxy acts as an intermediary that will buffer changes to the method invocation.

Binding time (modifiability). The active object pattern assumes that requests for the object arrive at the object at runtime. The binding of the client to the proxy, however, is left open in terms of binding time.

Scheduling policy (performance). The scheduler implements some scheduling policy.

UNIT-IV

CREATING AN ARCHITECTURE

DOCUMENTING SOFTWARE ARCHITECTURE

USE OF ARCHITECTURAL DOCUMENTATION:

Architecture documentation is both prescriptive and descriptive. That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true by recounting decisions already made about a system's design.

All of this tells us that different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.

One of the most fundamental rules for technical documentation in general, and software architecture documentation in particular, is to write from the point of view of the reader. Documentation that was easy to write but is not easy to read will not be used, and "easy to read" is in the eye of the beholder—or in this case, the stakeholder.

Documentation facilitates that communication. Some examples of architectural stakeholders and the information they might expect to find in the documentation are given in [Table 9.1](#).

In addition, each stakeholders come in two varieties: seasoned and new. A new stakeholder will want information similar in content to what his seasoned counterpart wants, but in smaller and more introductory doses. Architecture documentation is a key means for educating people who need an overview: new developers, funding sponsors, visitors to the project, and so forth.

_VIEWS

The concept of a view, which you can think of as capturing a structure, provides us with the basic principle of documenting software architecture

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

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 relevant views

Documenting view

Documenting information that applies to more than one view

CHOOSING THE RELEVANT VIEWS

A view simply represents a set of system elements and relationships among them, so whatever elements and relationships you deem useful to a segment of the stakeholder community constitute a valid view. Here is a simple 3 step procedure for choosing the views for your project.

Produce a candidate view list:

Begin by building a stakeholder/view table. Your stakeholder list is likely to be different from the one in the table as shown below, but be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views apply to every system, while others only apply to systems designed that way. Once you have rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

Table 9.2. Stakeholders and the Architecture Documentation They Might Find Most Useful

Stakeholder	Module Views				C&C Views	Allocation Views	
	Decomposition	Uses	Class	Layer	Various	Deployment	Implementation
Project Manager	s	s		s		d	
Member of Development Team	d	d	d	d	d	s	s
Testers and Integrators		d	d		s	s	s
Maintainers	d	d	d	d	d	s	s
Product Line Application Builder		d	s	o	s	s	s
Customer					s	o	
End User					s	s	
Analyst	d	d	s	d	s	d	
Infrastructure Support	s	s		s		s	d

Combine views:

The candidate view list from step 1 is likely to yield an impractically large number of views. To reduce the list to a manageable size, first look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger consistency. Next, look for the views that are good candidates to be combined- that is, a view that gives information from two or more views at once. For small and medium projects, the implementation view is often easily overlaid with the module decomposition view. The module decomposition view also pairs well with users or layered views. Finally, the deployment view usually combines well with whatever component-and-connector view shows the components that are allocated to hardware elements.

Prioritize:

After step 2 you should have an appropriate set of views to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific project. But, remember that you don't have to complete one view before starting another. People can make progress with overview-level information, so a breadth-first approach is often the best.

DOCUMENTING A VIEW

Primary presentation- elements and their relationships, contains main information about these system , usually graphical or tabular.

Element catalog- details of those elements and relations in the picture,

Context diagram- how the system relates to its environment

Variability guide- how to exercise any variation points a variability guide should include

documentation about each point of variation in the architecture, including

- o The options among which a choice is to be made

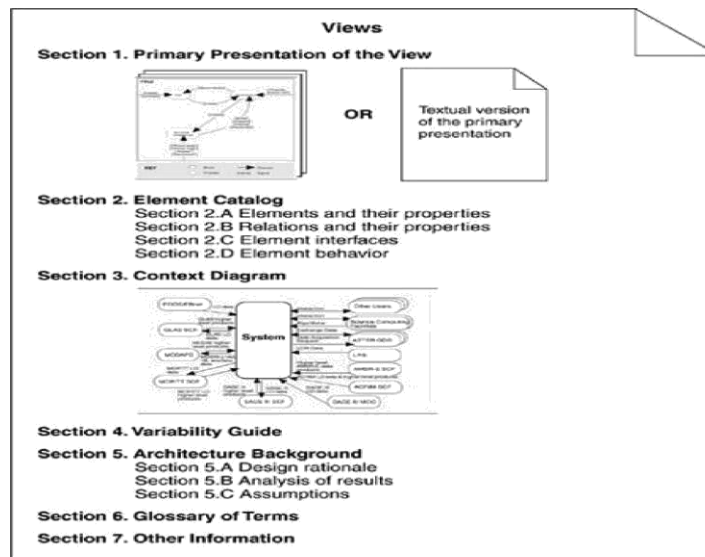
- o The binding time of the option. Some choices are made at design time, some at build time, and others at runtime.

Architecture background –why the design reflected in the view came to be? an architecture background includes

- o rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected
- o analysis results, which justify the design or explain what would have to change in the face of a modification
- o assumptions reflected in the design

Glossary of terms used in the views, with a brief description of each.

Other information includes management information such as authorship, configuration control data, and change histories. Or the architect might record references to specific sections of a requirements document to establish traceability



DOCUMENTING BEHAVIOR

Views present structural information about the system. However, structural information is not sufficient to allow reasoning about some system properties .behavior description add information that reveals the ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions.

Behavior can be documented either about an ensemble of elements working in concert. Exactly what to model will depend on the type of system being designed.

Different modeling techniques and notations are used depending on the type of analysis to be performed. In UML, sequence diagrams and state charts are examples of behavioral descriptions. These notations are widely used.

DOCUMENTING INTERFACES

An interface is a boundary across which two independent entities meet and interact or communicate with each other.

1. Interface identify

When an element has multiple interfaces, identify the individual interfaces to distinguish them. This usually means naming them. You may also need to provide a version number.

2. Resources provided:

The heart of an interface document is the resources that the element provides.

Resource syntax – this is the resource’s signature

Resource Semantics:

Assignment of values of data

Changes in state

Events signaled or message sent

how other resources will behave differently in future

humanly observable results

Resource Usage Restrictions

initialization requirements

limit on number of actors using resource

Data type definitions:

If used if any interface resources employ a data type other than one provided by the underlying programming language, the architect needs to communicate the definition of that type. If it is defined by another element, then reference to the definition in that element’s documentation is sufficient.

4. Exception definitions:

These describe exceptions that can be raised by the resources on the interface. Since the same exception might be raised by more than one resource, if it is convenient to simply list each resource's exceptions but define them in a dictionary collected separately.

5. Variability provided by the interface.

Does the interface allow the element to be configured in some way? These configuration parameters and how they affect the semantics of the interface must be documented.

6. Quality attribute characteristics:

The architect needs to document what quality attribute characteristics (such as performance or reliability) the interface makes known to the element's users

7. Element requirements:

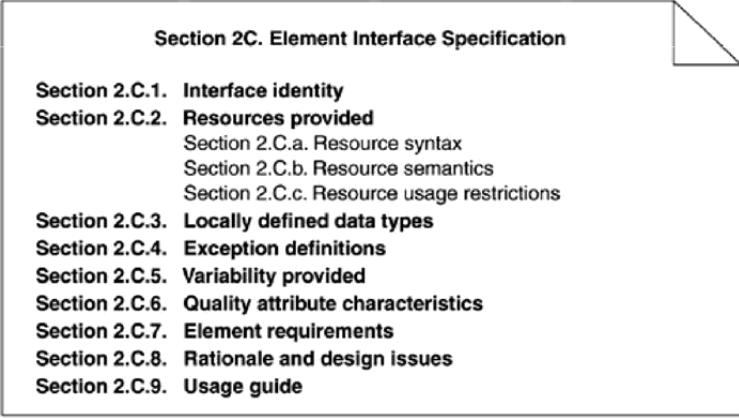
What the element requires may be specific, named resources provided by other elements. The documentation obligation is the same as for resources provided: syntax, semantics, and any usage restrictions.

8. Rationale and design issues:

Why these choices the architect should record the reasons for an elements interface design. The rationale should explain the motivation behind the design, constraints and compromises, what alternatives designs were considered.

9. Usage guide:

Item 2 and item 7 document an element's semantic information on a per resource basis. This sometimes falls short of what is needed. In some cases semantics need to be reasoned about in terms of how a broad number of individual interactions interrelate.

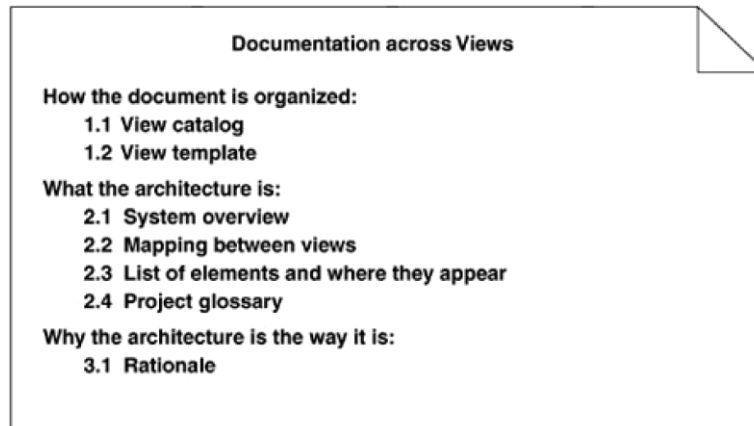


Section 2C. Element Interface Specification	
Section 2.C.1. Interface identity	
Section 2.C.2. Resources provided	
Section 2.C.a. Resource syntax	
Section 2.C.b. Resource semantics	
Section 2.C.c. Resource usage restrictions	
Section 2.C.3. Locally defined data types	
Section 2.C.4. Exception definitions	
Section 2.C.5. Variability provided	
Section 2.C.6. Quality attribute characteristics	
Section 2.C.7. Element requirements	
Section 2.C.8. Rationale and design issues	
Section 2.C.9. Usage guide	

Figure 9.2. The nine parts of interface documentation

DOCUMENTATION ACROSS VIEWS

Cross-view documentation consists of just three major aspects, which we can summarize as how-what-why:



Source: Adapted from [Clements 03].

Figure 9.3. Summary of cross-view documentation

HOW THE DOCUMENTATION IS ORGANIZED TO SERVE A STAKEHOLDER

Every suite of architectural documentation needs an introductory piece to explain its organization to a novice stakeholder and to help that stakeholder access the information he or she is most interested in. There are two kinds of "how" information:

View Catalog

A view catalog is the reader's introduction to the views that the architect has chosen to include in the suite of documentation.

There is one entry in the view catalog for each view given in the documentation suite. Each entry should give the following:

- The name of the view and what style it instantiates

- A description of the view's element types, relation types, and properties

- A description of what the view is for

Management information about the view document, such as the latest version, the location of the view document, and the owner of the view document

View Template

A view template is the standard organization for a view. It helps a reader navigate quickly to a section of interest, and it helps a writer organize the information and establish criteria for knowing how much work is left to do.

WHAT THE ARCHITECTURE IS

This section provides information about the system whose architecture is being documented, the relation of the views to each other, and an index of architectural elements.

System Overview

This is a short prose description of what the system's function is, who its users are, and any important background or constraints. The intent is to provide readers with a consistent mental model of the system and its purpose. Sometimes the project at large will have a system overview, in which case this section of the architectural documentation simply points to that.

Mapping between Views

Since all of the views of an architecture describe the same system, it stands to reason that any two views will have much in common. Helping a reader of the documentation understand the relationships among views will give him a powerful insight into how the architecture works as a unified conceptual whole. Being clear about the relationship by providing mappings between views is the key to increased understanding and decreased confusion.

Element List

The element list is simply an index of all of the elements that appear in any of the views, along with a pointer to where each one is defined. This will help stakeholders look up items of interest quickly.

Project Glossary

The glossary lists and defines terms unique to the system that have special meaning. A list of acronyms, and the meaning of each, will also be appreciated by stakeholders. If an appropriate glossary already exists, a pointer to it will suffice here.

WHY THE ARCHITECTURE IS THE WAY IT IS: RATIONALE

Cross-view rationale explains how the overall architecture is in fact a solution to its requirements. One might use the rationale to explain:

The implications of system-wide design choices on meeting the requirements or satisfying constraints.

The effect on the architecture when adding a foreseen new requirement or changing an existing one.

The constraints on the developer in implementing a solution.

Decision alternatives that were rejected.

In general, the rationale explains why a decision was made and what the implications are in changing it.

PART-2

RECONSTRUCTING SOFTWARE ARCHITECTURE

INTRODUCTION:

- 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 also important.

Architecture reconstruction has been used in a variety of projects ranging from MRI scanners to public telephone switches and from helicopter guidance systems to classified NASA systems. It has been used

- to redocument architectures for physics simulation systems.
- to understand architectural dependencies in embedded control software for mining machinery.

- to evaluate the conformance of a satellite ground system's

-

implementation to its reference architecture . to understand different systems in the automotive industry.

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 a selection 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

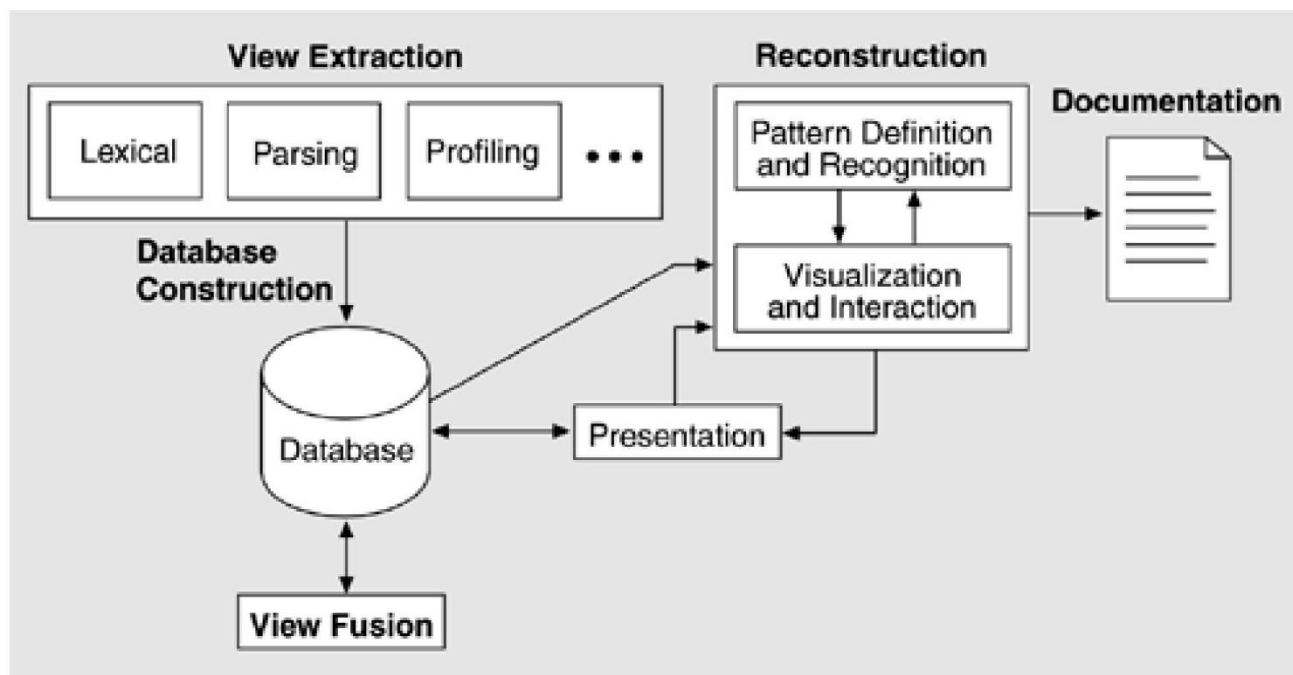
Software architecture reconstruction comprises the following activities, carried out iteratively:

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 the form of relationship <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 the architecture.
4. *Reconstruction.* The reconstruction activity is where the main work of building abstractions and various representations of the data to generate an architecture representation takes place.

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

Figure 10.1. Architecture reconstruction activities. (The arrows show how information flows among the activities.)



The reconstruction process needs to have several people involved. These include the person doing the reconstruction (reconstructor) and one or more individuals who are familiar with the system being reconstructed (architects and software engineers).

The reconstructor extracts the information from the system and either manually or

with the use of tools abstracts the architecture from it. The architecture is obtained by the reconstructor through a set of hypotheses about the system. These hypotheses reflect the inverse mappings from the source artifacts to the design (ideally the opposite of the design mappings). They are tested by generating the inverse mappings and applying them to the extracted information and validating the result. To most effectively generate these hypotheses and validate them, people familiar with the system must be involved, including the system architect or engineers who have worked on it (who initially developed it or who currently maintain it).

In the following sections, the various activities of architecture reconstruction are outlined in more detail along with some guidelines for each. Most of these guidelines are not specific to the use of a particular workbench and would be applicable even if the architecture reconstruction was carried out manually.

Information Extraction

Information extraction involves analyzing a system's existing design and implementation artifacts to construct a model of it. The result is a set of information placed in a database, which is used in the view fusion activity to construct a view of the system.

Information extraction is a blend of the ideal—what information do you want to discover about the architecture that will most help you meet the goals of your reconstruction effort—and the practical—what information can your available tools actually extract and present. From the source artifacts (e.g., code, header files, build files) and other artifacts (e.g., execution traces), you can identify and capture the elements of interest within the system (e.g., files, functions, variables) and their relationships to obtain several base system views. [Table 10.1](#) shows a typical list of the elements and several relationships among them that might be extracted.

Table 10.1. Typical Extracted Elements and Relations

Source Element	Relation	Target Element	Description
File	"includes"	File	C preprocessor #include of one file b another
File	"contains"	Function	Definition of a function in a file
File	"defines_var"	Variable	Definition of a variable in a file
Directory	"contains"	Directory	Directory contains a subdirectory
Directory	"contains"	File	Directory contains a file
Function	"calls"	Function	Static function call
Function	"access_read"	Variable	Read access on a variable
Function	"access_write"	Variable	Write access on a variable

Each of the relationships between the elements gives different information about the system. The calls relationship between functions helps us build a call graph. The includes relationship between the files gives us a set of dependencies between system files. The access_read and access_write relationships between functions and variables show us how data is used. Certain functions may write a set of data and others may read it. This information is used to determine how data is passed between various parts of the system. We can determine whether or not a global data store is used or whether most information is passed through function calls.

If the system being analyzed is large, capturing how source files are stored within the directory structure may be important to the reconstruction process. Certain elements or subsystems may be stored in particular directories, and capturing relations such as dir_contains_file and dir_contains_dir is useful when trying to identify elements later.

The set of elements and relations extracted will depend on the type of system being analyzed and the extraction support tools available. If the system to be reconstructed is object oriented, classes and methods are added to the list of elements to be extracted, and relationships such as class_is_subclass_of_class and class_contains_method are extracted and used.

To extract information, a variety of tools are used, including these:

● **Parsers** (e.g., Imagix, SNiFF+, CIA, rigiparse)

● **Abstract syntax tree (AST) analyzers** (e.g., Gen++, Refine) **Lexical analyzers** (e.g., L~~S~~ME)

● **Profilers** (e.g., gprof)

● **Code instrumentation tools Ad hoc** (e.g., grep, perl)

Parsers analyze the code and generate internal representations from it (for the purpose of generating machine code). Typically, however, it is possible to save this internal representation to obtain a view. **AST analyzers** do a similar job, but they build an explicit tree representation of the parsed information. We can build analysis tools that traverse the AST and output selected pieces of architecturally relevant information in an appropriate format.

Lexical analyzers examine source artifacts purely as strings of lexical elements or tokens. The user of a lexical analyzer can specify a set of code patterns to be matched and output. Similarly, a collection of ad hoc tools such as grep and perl can carry out pattern matching and searching within the code to output some required information. All of these tools—code-generating parsers, AST-based analyzers, lexical analyzers, and ad hoc pattern matchers—are used to output static information.

Profiling and code coverage analysis tools can be used to output information about the code as it is being executed, and usually do not involve adding new code to the system. On the other hand, code instrumentation, which has wide applicability in the field of testing, involves adding code to the system to output specific information while the system is executing. These tools generate dynamic system views.

Tools to analyze design models, build files, makefiles, and executables can also be used to extract further information as required. For instance, build files and makefiles include information on module or file dependencies that exist within the system and may not be reflected in the source code.

Much architecture-related information may be extracted statically from source code, compile-time artifacts, and design artifacts. Some architecturally relevant information, however, may not exist in the source artifacts because of late binding. Examples of late binding include the following:

Polymorphism Function pointers

Runtime parameterization

The precise topology of a system may not be determined until runtime. For example, multi-process and multi-processor systems, using middleware such as J2EE, Jini, or .NET, frequently establish their topology dynamically, depending on the availability of system resources. The topology of such systems does not live in its source artifacts and hence cannot be reverse engineered using static extraction tools.

For this reason, it may be necessary to use tools that can generate dynamic information about the system (e.g., profiling tools). Of course, this requires that such tools be available on the platform on which the system executes. Also, it may be difficult to collect the results from code instrumentation. Embedded systems often have no way to output such information.

Database Construction

The extracted information is converted into a standard format for storage in a database during database construction. It is necessary to choose a database model. When doing so, consider the following:

- It should be a well-known model, to make replacing one database
- implementation with another relatively simple. It should allow for
- efficient queries, which is important given that source models can be quite large.

- It should support remote access of the database from one or more
- geographically distributed user interfaces. It supports view fusion by combining information from various tables.

- It supports query languages that can express architectural patterns.

- Checkpointing should be supported by implementations, which

means that intermediate results can be saved. This is important in an interactive process in that it gives the user the freedom to explore with the comfort that changes can always be undone.

The Dali workbench, for example, uses a relational database model. It converts the extracted views (which may be in many different formats depending on the tools used to extract them) into the Rigi Standard Form. This format is then read in by a perl script and output in a format that includes the necessary SQL code to build the relational tables and populate them with the extracted information. [Figure 10.2](#) gives an outline of this process.

Figure 10.2. Conversion of the extracted information to SQL format



An example of the generated SQL code to build and populate the relational tables is shown in [Figure 10.3](#).

When the data is entered into the database, two additional tables are generated: *elements* and *relationships*. These list the extracted elements and relationships, respectively.

Here, the workbench approach makes it possible to adopt new tools and techniques, other than those currently available, to carry out the conversion from whatever format(s) an extraction tool uses. For example, if a tool is required to handle a new language, it can be built and its output can be converted into the workbench format.

In the current version of the Dali workbench, the POSTGRES relational database provides functionality through the use of SQL and perl for generating and manipulating the architectural views (examples are shown in [Section 10.5](#)). Changes can easily be made to the SQL scripts to make them compatible with other SQL implementations.

Example:

```
create table calls( caller text, callee text );
create table access( func text, variable text );
create table defines_var( file text, variable text );
```

...

```
insert into calls values( 'main', 'control' ); insert into calls values( 'main', 'clock' );
```

...

```
insert into accesses values( 'main', 'stat 1' );
```

View Fusion

View fusion involves defining and manipulating extracted information (now stored in a database) to reconcile, augment, and establish connections between the elements. Different forms of extraction should provide complementary information. Fusion is illustrated using the examples given in the following sections.

IMPROVING A VIEW

Consider the two excerpts shown in [Figure 10.4](#), which are from the sets of methods (each shown preceded by its respective class) extracted from a system implemented in C++. These tables include static and dynamic information about an object-oriented segment of code. We can see from the dynamic information that, for example, `List::getnth` is called. However, this method is not included in the static analysis because the static extractor tool missed it. Also, the calls to the constructor and destructor methods of `InputValue` and `List` are not included in the static information and need to be added to the class/method table that reconciles both sources of information.

Figure 10.4. Static and dynamic data information about the `class_contains_method` relation

Static Extraction

```
InputValue::GetValue  
InputValue::SetValue  
List::[]  
List::length  
List::attachr  
List::detachr  
PrimitiveOp::Compute
```

Dynamic Extraction

```
InputValue::GetValue  
InputValue::SetValue  
InputValue::~~InputValue  
InputValue::InputValue  
List::[]  
List::length  
List::getnth  
List::List  
ArithmeticOp::Compute  
AttachOp::Compute  
...  
StringOp::Compute
```

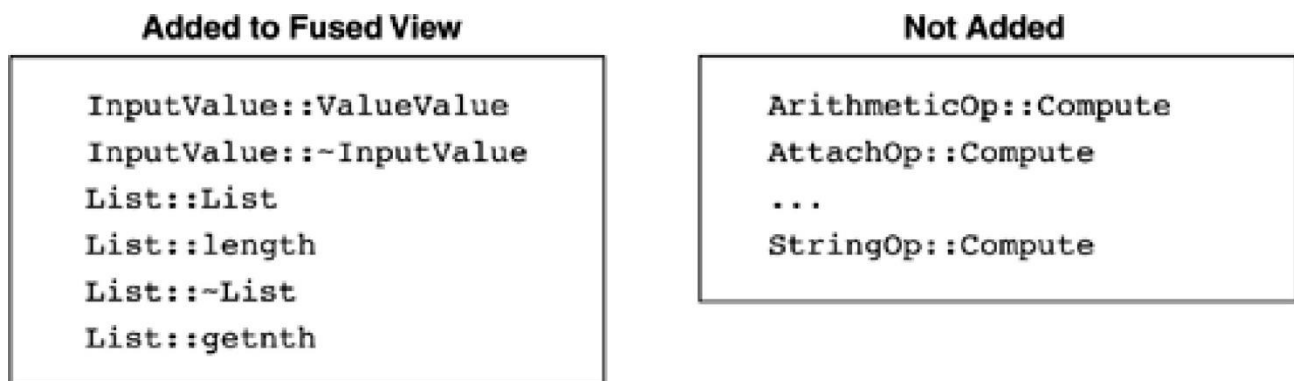
In addition, the static extraction in this example shows that the `PrimitiveOp` class has a method called `Compute`. The dynamic extraction results show no such class, but

they do show classes, such as ArithmeticOp, AttachOp, and StringOp, each of which has aCompute method and is in fact a subclass of PrimitiveOp. PrimitiveOp is purely a superclass and so never actually called in an executing program. But it is the call to PrimitiveOp that a static extractor sees when scanning the source code, since the polymorphic call to one of PrimitiveOp's subclasses occurs at runtime.

To get an accurate view of the architecture, we need to reconcile the PrimitiveOp static and dynamic information. To do this, we perform a fusion using SQL queries over the extracted calls, *actually_calls*, and *has_subclass* relations. In this way, we can see that the calls to Compute (obtained from the static information) and to its various subclasses (obtained from the dynamic information) are really the same thing.

The lists in [Figure 10.5](#) show the items added to the fused view (in addition to the methods that the static and dynamic information agreed upon) and those removed from it (even though included in either the static or the dynamic information).

Figure 10.5. Items added to and omitted from the overall view



DISAMBIGUATING FUNCTION CALLS

In a multi-process application, name clashes are likely to occur. For example, several processes might have a procedure called main. It is important that clashes be identified and disambiguated within the extracted views. Once again, by fusing information that can be easily extracted, we can remove this potential ambiguity. In this case, we need to fuse the static calls table with a "file/function containment" table (to determine which functions are defined in which source files) and a "build dependency" table (to determine which files are compiled to produce which executables). The fusion of these information sources allows potentially ambiguous procedure or method names to be made unique and hence unambiguously referred to in the architecture reconstruction process. Without view fusion, this ambiguity would persist into the architecture reconstruction.

Re Construction:

At this point, the view information has been extracted, stored, and refined or augmented to improve its quality. The reconstruction operates on views to reveal broad, coarse-grained insights into the architecture. 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. An example of an architectural view is shown in [Figure 10.6](#).

