SOFTWARE ARCHITECTURES

NOTES FOR 8th SEMESTER COMPUTER SCIENCE & ENGINEERING

SUBJECT CODE: 10IS81

PREPARED BY
Shilpa B. Hosagoudra

TEXT BOOKS:

1. **Software Architecture in Practice, Second Edition** by Len Bass, Paul Clements, Rick Kazman
2. **Pattern-Oriented Software Architecture** by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sornmerlad, Michael Stal
3. **An Introduction To Software Architecture** by David Garlan And Mary Shaw

UNIT 1
INTRODUCTION

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 examines the following:

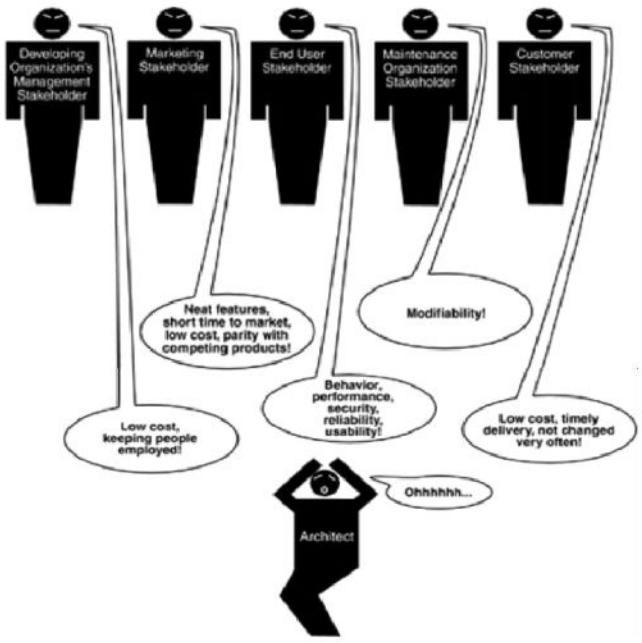
- ✓ How organizational goals influence requirements and development strategy.
- ✓ How requirements lead to architecture.
- ✓ How architectures are analyzed.
- ✓ How architectures yield systems that suggest new organizational capabilities and requirements.

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.

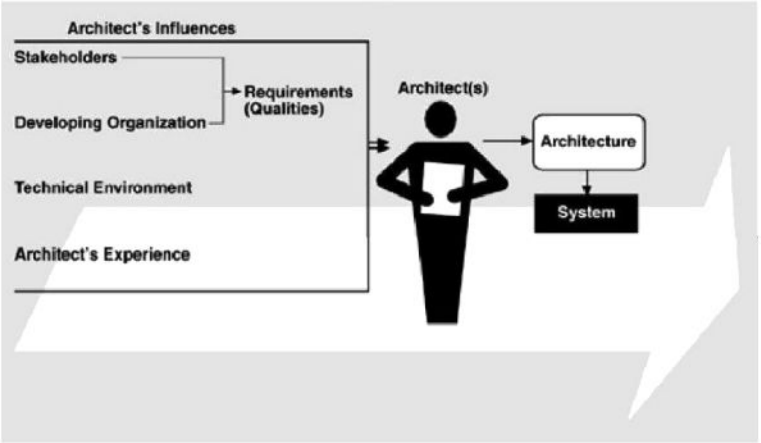
- ✓ If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort.
- ✓ Conversely, if their prior experience with this approach was disastrous, 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

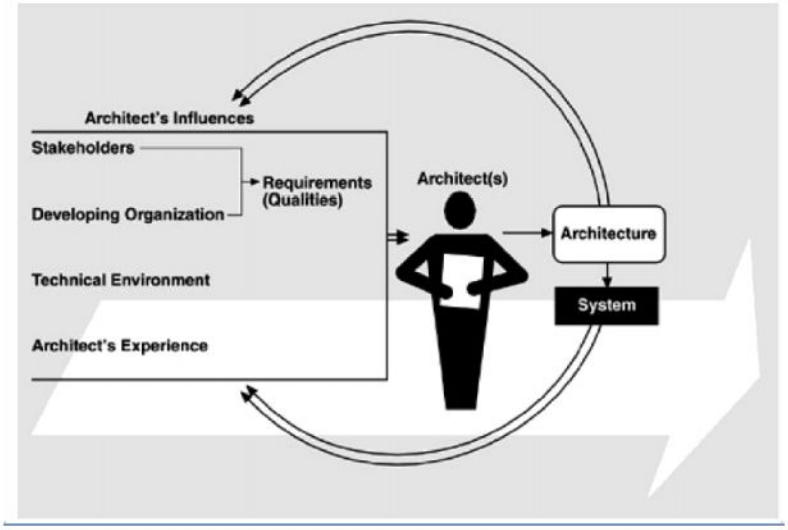


- ✓ Influences on an architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict.
- ✓ Architects need to know and understand the nature, source, and priority of constraints on the project as early as possible.
- ✓ Therefore, *they must identify and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

❖ THE ARCHITECTURE AFFECTS THE FACTORS THAT INFLUENCE THEM

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

Figure 1.4. The Architecture Business Cycle



Working of architecture business cycle:

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

1.2 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.

1.3 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.

CHAPTER 2

WHAT IS SOFTWARE ARCHITECTURE

2.1 WHAT SOFTWARE ARCHITECTURE IS AND WHAT IT ISN'T

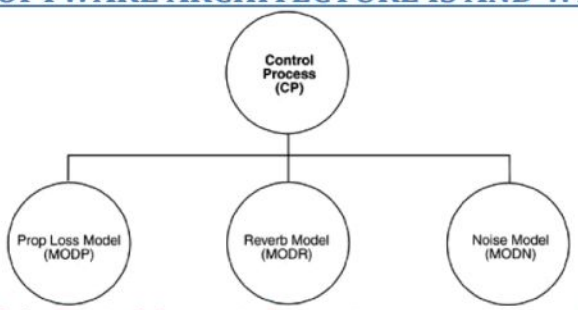


Figure 2.1 : Typical, but uninformative, presentation of a software architecture

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

- 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.

2.2 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.

2.3 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.

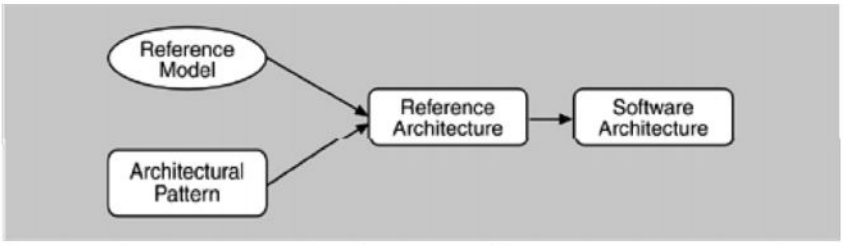
For ex: client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients.

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.

2.4 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.

2.5 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.

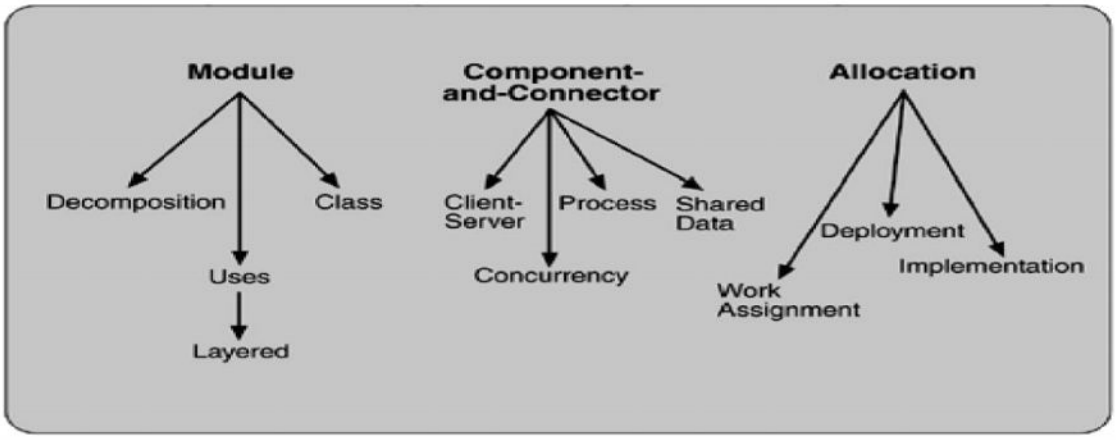
- ***Module structures.***

  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

Module-based structures include the following structures.

- ***Decomposition:*** The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood.
- ***Uses:*** The units are related by the *uses* relation. One unit uses another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second.
- ***Layered:*** Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.
- ***Class or generalization:*** The class structure allows us to reason about re-use and the incremental addition of functionality.

Component-and-connector

Component-and-connector structures include the following structures

- ***Process or communicating processes***: The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations.
- ***Concurrency:*** The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.
- ***Shared data or repository:*** This structure comprises components and connectors that create, store, and access persistent data
- ***Client-server:*** This is useful for separation of concerns (supporting modifiability), for physical distribution, and for load balancing (supporting runtime performance).

Allocation

Allocation structures include the following structures

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

RELATING STRUCTURES TO EACH OTHER

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

WHICH STRUCTURES TO CHOOSE?

Kruchten's four views follow:

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

UNIT 1 - QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	What is a architecture business cycle?	Dec 09	7
2	Why is software architecture important?	Dec 09	3
3	List all the common software architecture structures. Explain the component connector structure.	Dec 09	10
4	Define software architecture. Explain the common software architecture structures.	June 10	10
5	Explain how the architecture business cycle works, with a neat diagram	June 10	10
6	Explain how the software architectures affect the factors of influence. Hence or otherwise explain ABC.	Dec 10	8
7	Briefly explain the technical importance of software architectures. Further elaborate on the fact that architecture is the vehicle for stakeholder communication	Dec 10	7
8	What is an allocation structure as applied to software architectures? Explain the three allocation structures in practice.	Dec 10/ June 14	5/10
9	With the help of a neat block diagram of ABC, explain in detail the different activities which are involved in creating a software architecture	June 11/14	10
10	Enumerate and explain in detail the different groups software architecture structures are categorized into with the help of appropriate pictorial descriptions	June 11	10
11	Define software architecture. Discuss in detail, the implications of the definition	Dec 11	10
12	Define the following terms: i)architectural model ii)reference model iii)reference architecture	Dec 11	6
13	Explain the module based structures	Dec 11	4
14	Define software architecture. What is a architecture business cycle? Explain with a neat diagram	June 12	10
15	Define architectural model, reference model, reference architecture and bring out the relationship between them	June 12	6
16	Explain the various process recommendations as used by an architect while developing software architectures	June 12	4

UNIT 2
ARCHITECTURAL STYLES & CASE STUDIES

ARCHITECTURAL STYLES

List of common architectural styles:

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

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

Independent components:
- ✓ Communicating processes
- ✓ Event systems

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

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

PIPES AND FILTERS

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

Conditions (invariants) of this style are:

- Filters must be independent entities.
- They should not share state with other filter
- Filters do not know the identity of their upstream and downstream filters.
- Specification might restrict what appears on input pipes and the result that appears on the output pipes.
- Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

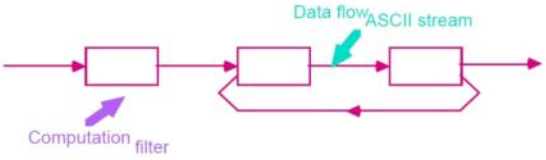


Figure 1: Pipes and Filters

Common specialization of this style includes :

- *Pipelines:*
  Restrict the topologies to linear sequences of filters.
- *Bounded pipes:*
  Restrict the amount of data that can reside on pipe.
- *Typed pipes:*
  Requires that the data passed between two filters have a well-defined type.

Batch sequential system:

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

Example 1:

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

Example 2:

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

- Signal processing domains
- Parallel processing
- Functional processing
- Distributed systems.

Advantages:

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

Disadvantages:

- They lead to a batch organization of processing.
- Filters are independent even though they process data incrementally.
- Not good at handling interactive applications
  - ✓ When incremental display updates are required.
  - ✓ They may be hampered by having to maintain correspondences between two separate but related streams.
  - ✓ Lowest common denominator on data transmission.

This can lead to both loss of performance and to increased complexity in writing the filters.

OBJECT-ORIENTED AND DATA ABSTRACTION

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

Two important aspects of this style are:

- ♥ Object is responsible for preserving the integrity of its representation.
- ♥ Representation is hidden from other objects.

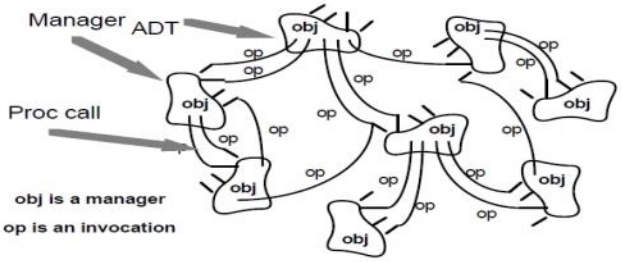


Figure 2: Abstract Data Types and Objects

Advantages

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

Disadvantages

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

EVENT-BASED, IMPLICIT INVOCATION

- ✓ Instead of invoking the procedure directly a component can announce one or more events.
- ✓ Other components in the system can register an interest in an event by associating a procedure to it.
- ✓ When the event is announced, the system itself invokes all of the procedure that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules.
- ✓ Architecturally speaking, the components in an implicit invocation style are modules whose interface provides both a collection of procedures and a set of events.

Advantages:

- ✓ *It provides strong support for reuse*
  Any component can be introduced into the system simply by registering it for the events of that system.
- ✓ *Implicit invocation eases system evolution.*
  Components may be replaced by other components without affecting the interfaces of other components.

Disadvantages:

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

LAYERED SYSTEMS:

- ✓ A layered system is organized hierarchically
- ✓ Each layer provides service to the layer above it.
- ✓ Inner layers are hidden from all except the adjacent layers.
- ✓ Connectors are defined by the protocols that determine how layers interact each other.
- ✓ Goal is to achieve qualities of modifiability portability.

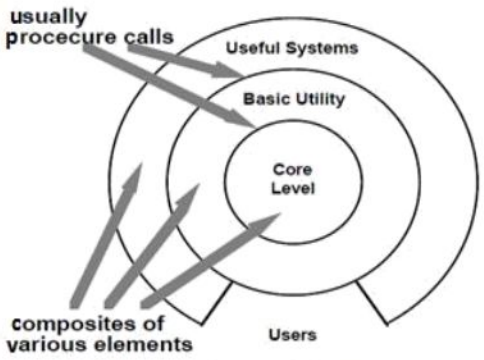


Figure 3: Layered Systems

Examples:

- ✓ Layered communication protocol
- ✓ Operating systems
- ✓ Database systems

Advantages:

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

Disadvantages:

- Not easily all systems can be structures in a layered fashion.

- Performance may require closer coupling between logically high-level functions and their lower-level implementations.
- Difficulty to mapping existing protocols into the ISO framework as many of those protocols bridge several layers.
  Layer bridging: functions is one layer may talk to other than its immediate neighbor.

REPOSITORIES: [data cantered architecture]

✓ Goal of achieving the quality of integrability of data.

✓ In this style, there are two kinds of components.

i. Central data structure- represents current state.

ii. Collection of independent components which operate on central data store. The choice of a control discipline leads to two major sub categories.

♣ Type of transactions is an input stream trigger selection of process to execute

♣ Current state of the central data structure is the main trigger for selecting processes to execute.

- Active repository such as blackboard.

Blackboard:

Three major parts:

- **Knowledge sources:**
  Separate, independent parcels of application – dependents knowledge.
- **Blackboard data structure:**
  Problem solving state data, organized into an application-dependent hierarchy
- **Control:**
  Driven entirely by the state of blackboard

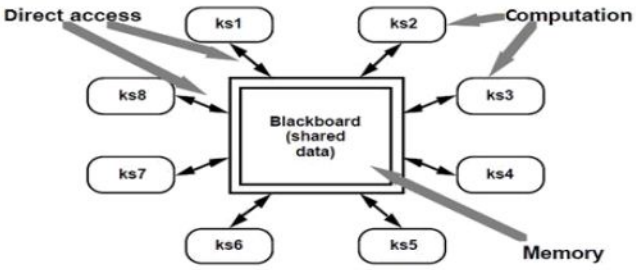


Figure 4: The Blackboard

✓ 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

✓ 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.

Interpreter includes 4 components:

1. Interpretation engine: to do the work
2. Memory: that contains pseudo code to be interpreted.
3. Representation of control state of interpretation engine
4. Representation of control state of the program being simulated.
   Ex: JVM or "virtual Pascal machine"

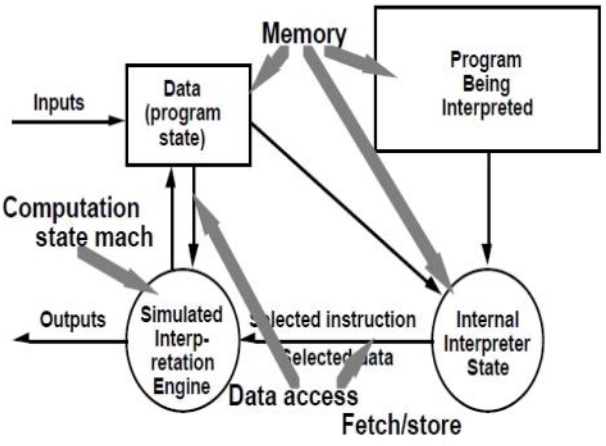


Figure 5: Interpreter

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

Disadvantages:
Performance cost because of additional computational involved

PROCESS CONTROL

PROCESS CONTROL PARADIGMS

Useful definitions:

Process variables → properties of the process that can be measured

Controlled variable → process variable whose value of the system is intended to control

Input variable → process variable that measures an input to the process

Manipulated variable → process variable whose value can be changed by the controller

Set point → the desired value for a controlled variable

Open-loop system → system in which information about process variables is not used to adjust the system

Closed-loop system → system in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions

Feedback control system → the controlled variable is measured and the result is used to manipulate one or more of the process variables

Feed forward control system → some of the process variables are measured, and anticipated disturbances are compensated without waiting for changes in the controlled variable to be visible.

The open-loop assumptions are rarely valid for physical processes in the real world. More often, properties such as temperature, pressure and flow rates are monitored, and their values are used to control the process by changing the settings of apparatus such as valve, heaters and chillers. Such systems are called *closed loop systems.*

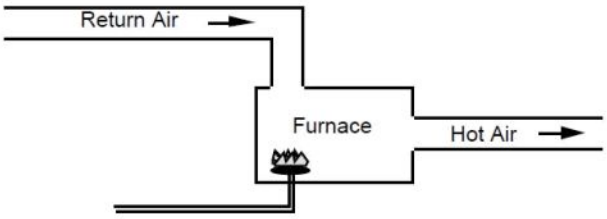


Figure 2.8 open-loop temperature control

A home thermostat is a common example; the air temperature at the thermostat is measured, and the furnace is turned on and off as necessary to maintain the desired temperature. Figure 2.9 shows the addition of a thermostat to convert figure 2.8 to a closed loop system.

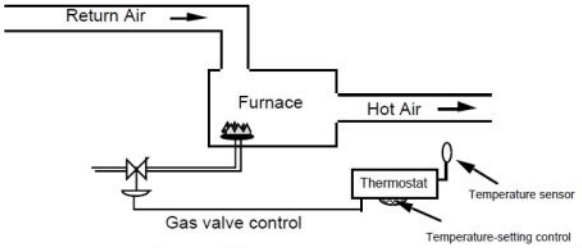


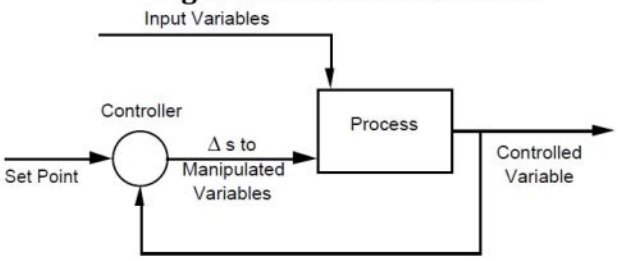
Figure 2.9 closed-loop temperature control

Feedback control:
Figure 2.9 corresponds to figure 2.10 as follows:

- The furnace with burner is the process
- The thermostat is the controller
- The return air temperature is the input variable
- The hot air temperature is the controlled variable

- The thermostat setting is the set point
- Temperature sensor is the sensor

Figure 2.10 feedback control



Feedforward control:

It anticipates future effects on the controlled variable by measuring other process variables and adjusts the process based on these variables. The important components of a feedforward controller are essentially the same as for a feedback controller except that the sensor(s) obtain values of input or intermediate variables.

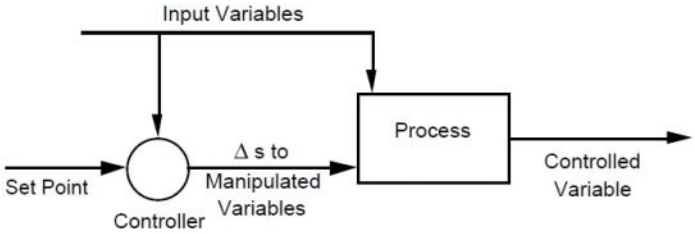


Figure 2.11 feedforward control

- These are simplified models
- They do not deal with complexities - properties of sensors, transmission delays & calibration issues
- They ignore the response characteristics of the system, such as gain, lag and hysteresis.
- They don't show how combined feedforward and feedback
- They don't show how to manipulate process variables.

A SOFTWARE PARADIGM FOR PROCESS CONTROL

An architectural style for software that controls continuous processes can be based on the process-control model, incorporating the essential parts of a process-control loop:

- **Computational elements**: separate the process of interest from the controlled policy
  - *Process definition*, including mechanisms for manipulating some process variables
  - *Control algorithm*, for deciding how to manipulate variables
- **Data element**: continuously updated process variables and sensors that collect them
  - *Process variables*, including designed input, controlled and manipulated variables and knowledge of which can be sensed
  - *Set point*, or reference value for controlled variable
  - *Sensors* to obtain values of process variables pertinent to control
- **The control loop paradigm**: establishes the relation that the control algorithm exercises.

OTHER FAMILIAR ARCHITECTURES

- **Distributed processes:** Distributed systems have developed a number of common organizations for multi-process systems. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in terms of the kinds of inter-process protocols that are used for communication (e.g., heartbeat algorithms).
- **Main program/subroutine organizations:** The primary organization of many systems mirrors the programming language in which the system is written. For languages without support for modularization this often results in a system organized around a main program and a set of subroutines.
- **Domain-specific software architectures:** These architectures provide an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle management

systems. By specializing the architecture to the domain, it is possible to increase the descriptive power of structures.

★ **State transition systems:** These systems are defined in terms a set of states and a set of named transitions that move a system from one state to another.

HETEROGENEOUS ARCHITECTURES

Architectural styles can be combined in several ways:

- One way is through hierarchy. Example: UNIX pipeline
- Second way is to combine styles is to permit a single component to use a mixture of architectural connectors. Example: "active database"
- Third way is to combine styles is to completely elaborate one level of architectural description in a completely different architectural style. Example: case studies

CASE STUDIES

KEYWORD IN CONTEXT (KWIC)

This case study shows how different architectural solutions to the same problem provide different benefits.
Parnas proposed the following problems:

> ***KWIC index system accepts an ordered set of lines. Each line is an ordered set of words and each word is an ordered set of characters. Any line may be circularly shifted by repeated removing the first word and appending it at the end of the line. KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.***

Parnas used the problem to contrast different criteria for decomposing a system into modules.
He describes 2 solutions:

a) Based on functional decomposition with share access to data representation.
b) Based on decomposition that hides design decision.

From the point of view of Software Architecture, the problem is to illustrate the effect of changes on software design. He shows that different problem decomposition vary greatly in their ability to withstand design changes. The changes that are considered by parnas are:

1. **The changes in processing algorithm:**
   Eg: line shifting can be performed on each line as it is read from input device, on all lines after they are read or an demand when alphabetization requires a new set of shifted lines.
2. **Changes in data representation:**
   Eg: Lines, words, characters can be stored in different ways. Circular shifts can be stored explicitly or implicitly

Garlan, Kaiser and Notkin also use KWIC problem to illustrate modularization schemes based on implicit invocation. They considered the following.

3. **Enhancement to system function:**
   Modify the system to eliminate circular shift that starts with certain noise change the system to interactive.
4. **Performance:**
   Both space and time
5. **Reuse:**
   Extent to which components serve as reusable entities

Let's outline 4 architectural designs for KWIC system.

SOLUTION 1: MAIN PROGRAM/SUBROUTINE WITH SHARED DATA

- Decompose the problem according to 4 basic functions performed.
  - Input
  - Shift
  - Alphabetize
  - output
- These computational components are coordinated as subroutines by a main program that sequence through them in turn.
- Data is communicated between components through shared storage.
- Communication between computational component and shared data is constrained by read-write protocol.

Advantages:

- Allows data to be represented efficiently. Since, computation can share the same storage

Disadvantages:

- Change in data storage format will affect almost all of the modules.
- Changes in the overall processing algorithm and enhancement to system function are not easily accommodated.
- This decomposition is not particularly support reuse.

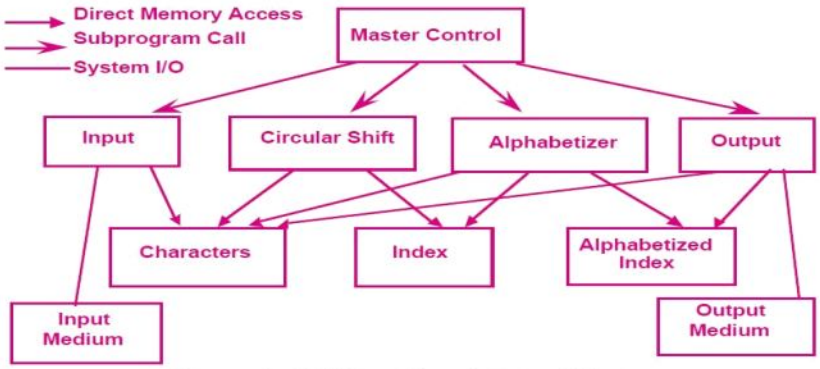


Figure 6: KWIC – Shared Data Solution

SOLUTION 2: ABSTRACT DATA TYPES

- Decomposes The System Into A Similar Set Of Five Modules.
- Data is no longer directly shared by the computational components.
- Each module provides an interface that permits other components to access data only by invoking procedures in that interface.

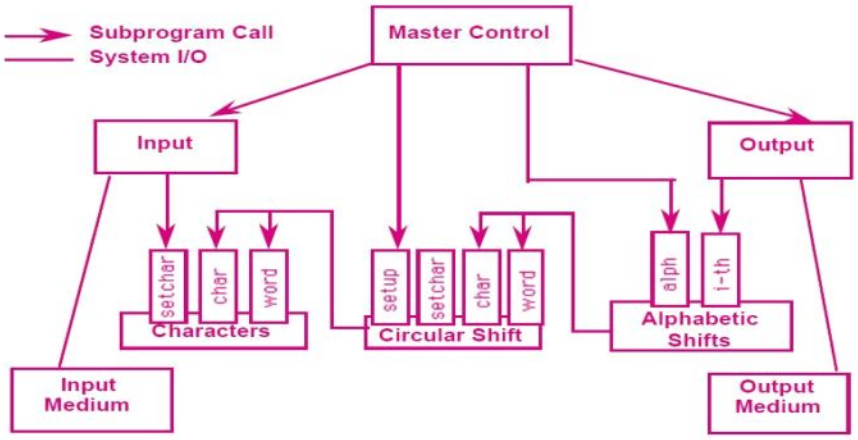


Figure 7: KWIC – Abstract Data Type Solution

Advantage:

- Both Algorithms and data representation can be changed in individual modules without affecting others.
- Reuse is better supported because modules make fewer assumption about the others with which they interact.

Disadvantage:

- Not well suited for functional enhancements
- To add new functions to the system
- To modify the existing modules.

SOLUTION 3: IMPLICIT INVOCATION

- Uses a form of component integration based on shared data
- Differs from 1st solution by these two factors
  - Interface to the data is abstract
  - Computations are invoked implicitly as data is modified. Interactions is based on an active data model.

Advantages:

- Supports functional enhancement to the system
- Supports reuse.

Disadvantages:

- Difficult to control the processing order.
- Because invocations are data driven, implementation of this kind of decomposition uses more space.

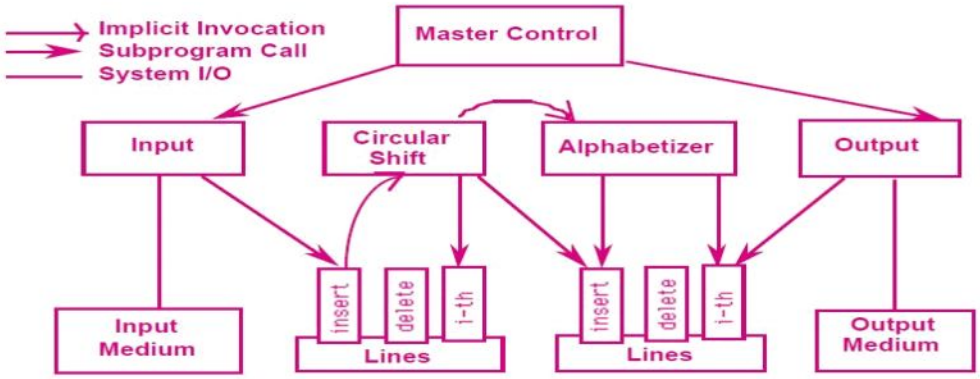


Figure 8: KWIC – Implicit Invocation Solution

SOLUTION 4: PIPES AND FILTERS:

- Four filters: Input, Output, Shift and alphabetize
- Each filter process the data and sends it to the next filter
- Control is distributed
  - Each filter can run whenever it has data on which to compute.
- Data sharing between filters are strictly limited.

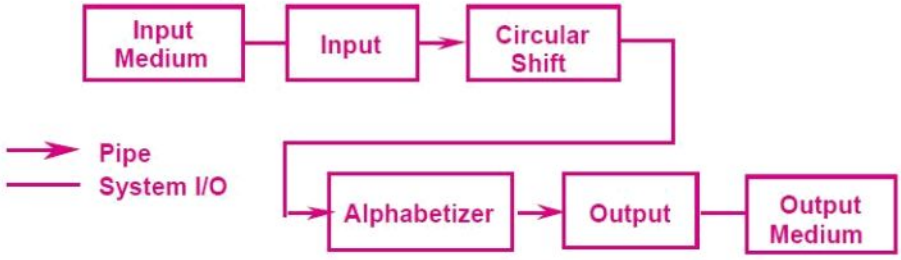


Figure 9: KWIC – Pipe and Filter Solution

Advantages:

- It maintains initiative flow of processing
- It supports reuse
- New functions can be easily added to the system by inserting filters at appropriate level.
- It is easy to modify.

Disadvantages:

- Impossible to modify the design to support an interactive system.
- Solution uses space inefficiently.

COMPARISONS

	Shared Memory	ADT	Events	Dataflow
Change in Algorithm	–	–	+	+
Change in Data Repn	–	+	–	–
Change in Function	+	–	+	+
Performance	+	+	–	–
Reuse	–	+	–	+

Figure 10: KWIC – Comparison of Solutions

INSTRUMENTATION SOFTWARE:

- Describes the industrial development of software architecture.
- The purpose of the project was to develop a reusable system architecture for oscilloscope
- Oscilloscope is an instrumentation system that samples electrical signals and displays pictures of them on screen.
- Oscilloscope also performs measurements on the signals and displays them on screen.
- Modern oscilloscope has to perform dozens of measurements supply megabytes of internal storage.
- Support an interface to a network of workstations and other instruments and provide sophisticated user interface, including touch panel screen with menus, built-in help facilities and color displays.
- Problems faced:
  - Little reuse across different oscilloscope products.
  - Performance problems were increasing because the software was not rapidly configurable within the instrument.
- Goal of the project was to develop an architectural framework for oscilloscope.
- Result of that was domain specific software architecture that formed the basis of the next generation of oscilloscopes.

SOLUTION 1: OBJECT ORIENTED MODEL

Different data types used in oscilloscope are:

- ✓ Waveforms
- ✓ Signals
- ✓ Measurements
- ✓ Trigger modes so on

There was no overall model that explained how the types fit together. This led to confusion about the partitioning of functionality. Ex: it is not clearly defined that measurements to be associated with types of data being measured or represented externally.

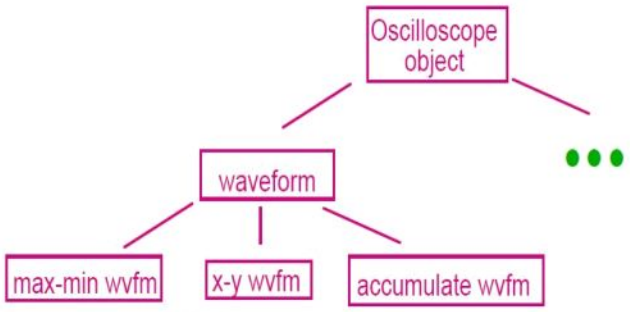


Figure 11: Oscilloscopes – An Object-oriented Model

SOLUTION 2: LAYERED MODEL

- To correct the problems by providing a layered model of an oscilloscope.
- Core-layer: implemented in hardware represents signal manipulation functions that filter signals as they enter the oscilloscope.
- Initially the layered model was appealing since it partitioned the functions of an oscilloscope into well defined groups.
- But, it was a wrong model for the application domain. Because, the problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction among various functions.

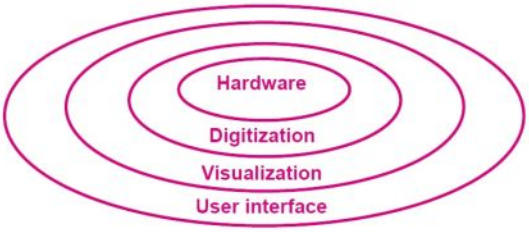


Figure 12: Oscilloscopes – A Layered Model

SOLUTION 3: PIPE-AND-FILTER MODEL:

- In this approach oscilloscope functions were viewed as incremental transformers of data.
  - Signal transformer: to condition external signal.
  - Acquisition transformer: to derive digitized waveforms
  - Display transformers: to convert waveforms into visual data.
- It is improvement over layered model as it did not isolate the functions in separate partition.
- Main problem with this model is that
  - It is not clear how the user should interact with it.

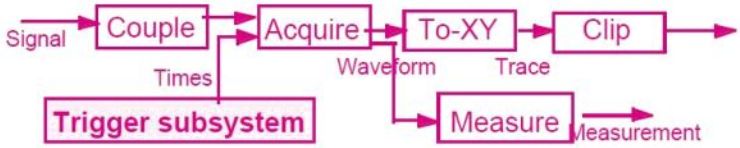


Figure 13: Oscilloscopes – A Pipe and Filter Model

SOLUTION 4: MODIFIED PIPE-AND-FILTER MODEL:

To overcome the above said problem, associate control interface with each filter that allowed external entity to set parameters of operation for the filter.
Introduction of control interface solves a large part of the user interface problem

- ✓ It provides collection of setting that determines what aspect of the oscilloscope can be modified dynamically by the user.
- ✓ It explains how user can change functions by incremental adjustments to the software.

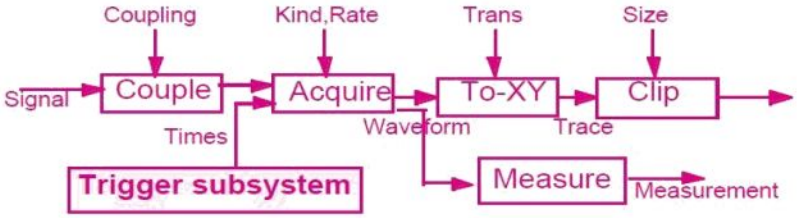


Figure 14: Oscilloscopes – A Modified Pipe and Filter Model

FURTHER SPECIALIZATION

The above described model is greater improvement over the past. But, the main problem with this is the performance.

a. *Because waveform occupy large amount of internal storage*
   It is not practical for each filter to copy waveforms every time they process them.
b. *Different filters run at different speeds*

It is unacceptable to slow one filter down because another filter is still processing its data. To overcome the above discussed problems the model is further specialized.

Instead of using same kind of pipe. We use different "colors" of pipe. To allow data to be processed without copying, slow filters to ignore incoming data.

These additional pipes increased the stylistic vocabulary and allowed pipe/filter computations to be tailored more specifically to the performance needs of the product.

MOBILE ROBOTICS

- **Mobile Robotic systems**
  - Controls a manned or semi-manned vehicle
    - E.g., car, space vehicle, etc
  - Used in space exploration missions
  - Hazardous waste disposal
  - Underwater exploration
- **The system is complex**
  - Real Time respond
  - input from various sensors
  - Controlling the motion and movement of robots
  - Planning its future path/move
- **Unpredictability of environment**
  - Obstacles blocking robot path
  - Sensor may be imperfect
  - Power consumption
  - Respond to hazardous material and situations

DESIGN CONSIDERATIONS

- **REQ1:** Supports deliberate and reactive behavior. Robot must coordinate the actions to accomplish its mission and reactions to unexpected situations
- **REQ2:** Allows uncertainty and unpredictability of environment. The situations are not fully defined and/or predicable. The design should handle incomplete and unreliable information
- **REQ3:** System must consider possible dangerous operations by Robot and environment
- **REQ4:** The system must give the designer flexibility (mission's change/requirement changes)

SOLUTION 1: CONTROL LOOP

- **Req1:** an advantage of the closed loop paradigm is its simplicity → it captures the basic interaction between the robot and the outside.
- **Req2:** control loop paradigm is biased towards one method → reducing the unknowns through iteration
- **Req3:** fault tolerance and safety are supported which makes duplication easy and reduces the chances of errors
- **Req4:** the major components of a robot architecture are separated from each other and can be replaced independently

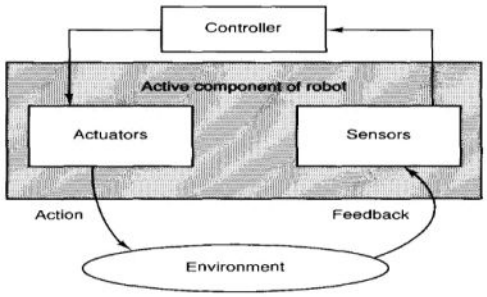
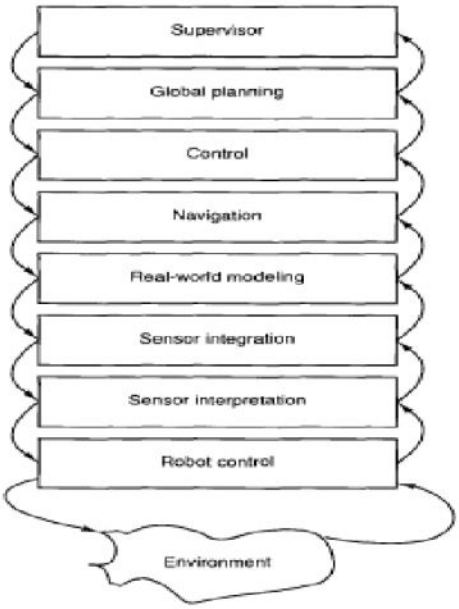


FIGURE 3.10 A Control-Loop Solution for Mobile Robots

SOLUTION 2: LAYERED ARCHITECTURE

Figure shows Alberto Elfes's definition of the layered architecture. ⊕ Level 1 (core) control routines (motors, joints,..),
⊕ Level 2-3 real world I/P (sensor interpretation and integration (analysis of combined I/Ps)
⊕ Level 4 maintains the real world model for robot
⊕ Level 5 manage navigation
⊕ Level 6-7 Schedule & plan robot actions (including exception handling and re-planning)
⊕ Top level deals with UI and overall supervisory functions



- **Req1:** it overcomes the limitations of control loop and it defines abstraction levels to guide the design
- **Req2:** uncertainty is managed by abstraction layers
- **Req3:** fault tolerance and passive safety are also served
- **Req4:** the interlayer dependencies are an obstacle to easy replacement and addition of components.

SOLUTION 3: IMPLICIT INVOCATION

The third solution is based on the form of implicit invocation, as embodied in the Task-Control-Architecture (TCA). The TCA design is based on hierarchies of tasks or task trees

- Parent tasks initiate child task
- Temporal dependencies between pairs of tasks can be defined
  - A must complete A must complete before B starts (selective concurrency)
- Allows dynamic reconfiguration of task tree at run time in response to sudden change(robot and environment)
- Uses implicit invocation to coordinate tasks
  - Tasks communicate using multicasting message (message server) to tasks that are registered for these events

TCA's implicit invocation mechanisms support three functions:

- **Exceptions:** Certain conditions cause the execution of an associated exception handling routines
  - i.e., exception override the currently executing task in the sub-tree (e.g., abort or retry) tasks
- **Wiretapping:** Message can be intercepted by tasks superimposed on an existing task tree
  - E.g., a safety-check component utilizes this to validate outgoing motion commands
- **Monitors**: Monitors read information and execute some action if the data satisfy certain condition
  - E.g. battery check

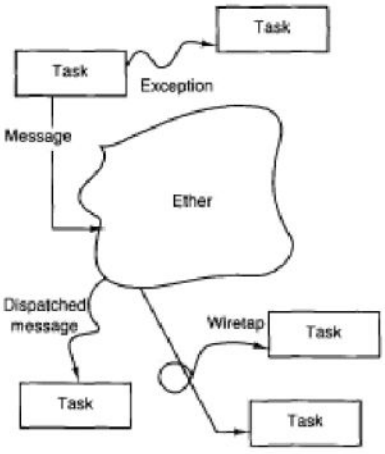


FIGURE 3.12 An Implicit Invocation Architecture for Mobile Robots

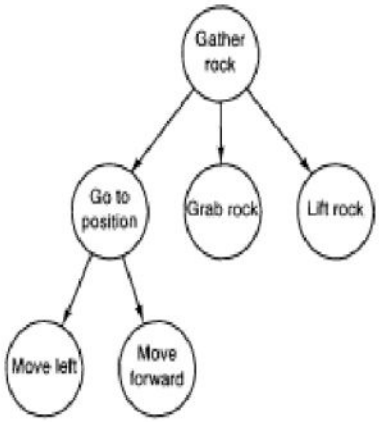


FIGURE 3.13 A Task Tree for Mobile Robo

- **Req1:** permits clear cut separation of action and reaction
- **Req2:** a tentative task tree can be built to handle uncertainty
- **Req3:** performance, safety and fault tolerance are served
- **Req4:** makes incremental development and replacement of components straight forward

SOLUTION 4: BLACKBOARD ARCHITECTURE

The components of CODGER are the following:

- Captain: overall supervisor
- Map navigator: high-level path planner
- Lookout: monitors environment for landmarks
- Pilot: low-level path planner and motor controller
- Perception subsystems: accept sensor input and integrate it into a coherent situation interpretation

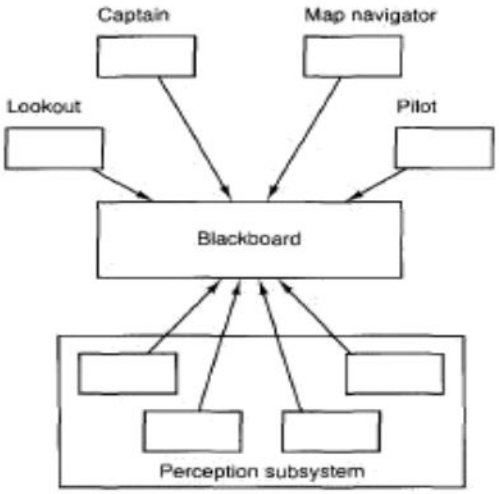


Figure: blackboard solution for mobile robots

The requirements are as follows:

- **Req1:** the components communicate via shared repository of the blackboard system.
- **Req2:** the blackboard is also the means for resolving conflicts or uncertainties in the robot's world view
- **Req3:** speed, safety and reliability is guaranteed
- **Req4:** supports concurrency and decouples senders from receivers, thus facilitating maintenance.

COMPARISONS

	Control Loop	Layers	Impl. Invoc.	Black Board
Task Coordination	+–	–	++	+
Dealing with Uncertainty	–	+–	+–	+
Fault Tolerance	+–	+–	++	+
Safety	+–	+–	++	+
Performance	+–	+–	++	+
Flexibility	+–	–	+	+

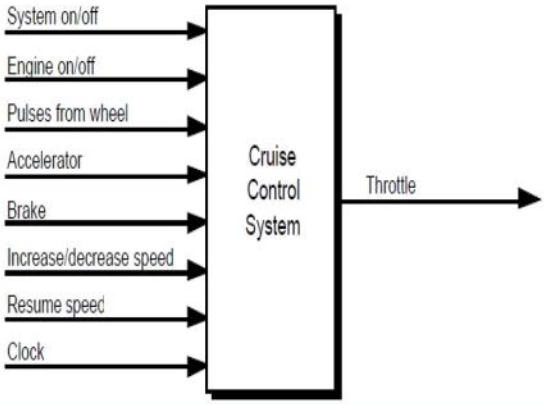
Table 2.2.1. Strengths and Weaknesses of Robot Architectures

CRUISE CONTROL

A cruise control (CC) system that exists to maintain the constant vehicle speed even over varying terrain.

Inputs:

System On/Off: If on, maintain speed
Engine On/Off: If on, engine is on. CC is active only in this state Wheel Pulses: One pulse from every wheel revolution
Accelerator: Indication of how far accelerator is de-pressed
Brake: If on, temp revert cruise control to manual mode
Inc/Dec Speed: If on, increase/decrease maintained speed
Resume Speed: If on, resume last maintained speed
Clock: Timing pulses every millisecond



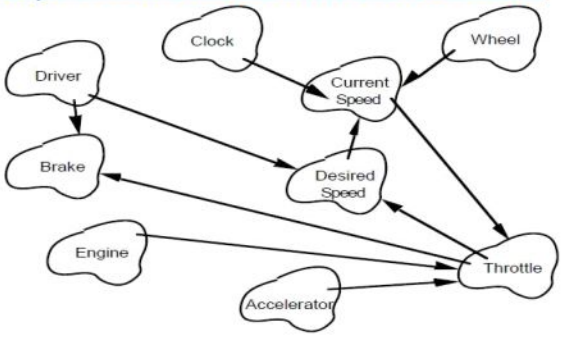
Outputs:

Throttle: Digital value for engine throttle setting

Restatement of Cruise-Control Problem

Whenever the system is active, determine the desired speed, and control the engine throttle setting to maintain that speed.

OBJECT VIEW OF CRUISE CONTROL



⊗ Each element corresponds to important quantities and physical entities in the system
⊗ Each blob represents objects
⊗ Each directed line represents dependencies among the objects

The figure corresponds to Booch's object oriented design for cruise control

PROCESS CONTROL VIEW OF CRUISE CONTROL

❖ **Computational Elements**

- ✓ *Process definition* - take throttle setting as I/P & control vehicle speed
- ✓ *Control algorithm* - current speed (wheel pulses) compared to desired speed ○ Change throttle setting accordingly presents the issue:
  - ○ decide how much to change setting for a given discrepancy

❖ **Data Elements**

- ✓ *Controlled variable:* current speed of vehicle
- ✓ *Manipulated variable:* throttle setting
- ✓ *Set point:* set by accelerator and increase/decrease speed inputs
  - ▪ system on/off, engine on/off, brake and resume inputs also have a bearing
- ✓ *Controlled variable sensor*: modelled on data from wheel pulses and clock

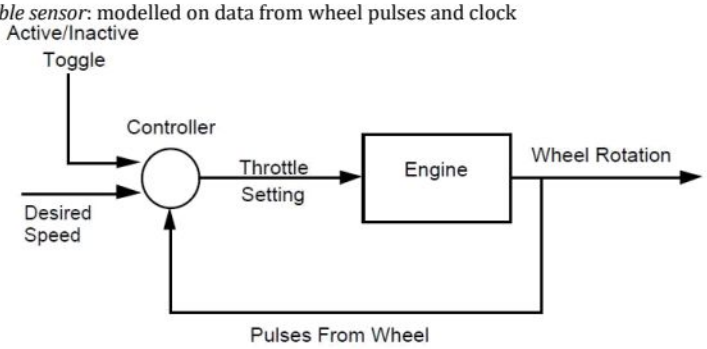


Figure 3.18 control architecture for cruise control

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural. It's shown in Figure 3.19. The system is completely off whenever the engine is off. Otherwise there are three inactive and one active state.

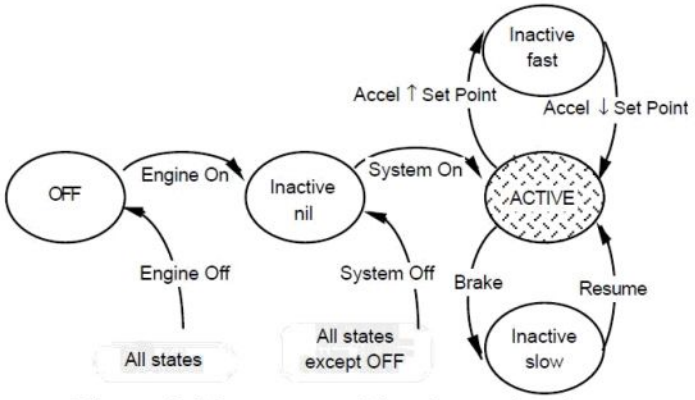


Figure 3.19 state machine for activation

Event	*Effect on desired speed*
Engine off, system off	Set to "undefined"
System on	Set to current speed as estimated from wheel pulses
Increase speed	Increment desired speed by constant
Decrease speed	Decrement desired speed by constant

We can now combine the control architecture, the state machine for activation, and the event table for determining the set point into an entire system.

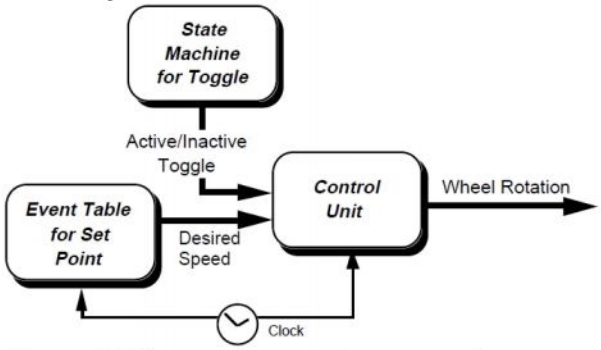


Figure 3.21 complete cruise control system

THREE VIGNETTES IN MIXED STYLE

A LAYERED DESIGN WITH DIFFERENT STYLES FOR THE LAYERS

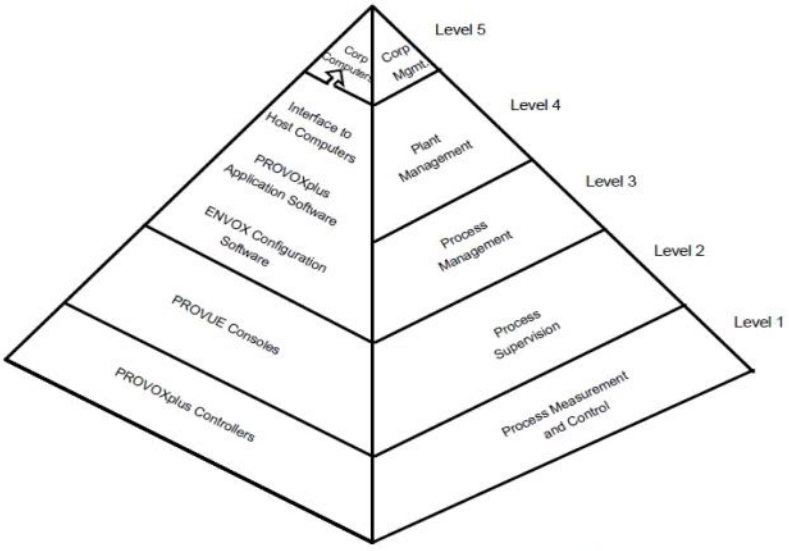


Figure 19: PROVOX – Hierarchical Top Level

Each level corresponds to a different process management function with its own decision-support requirements.

- **Level 1:** Process measurement and control: direct adjustment of final control elements.
- **Level 2:** Process supervision: operations console for monitoring and controlling Level 1.
- **Level 3:** Process management: computer-based plant automation, including management reports, optimization strategies, and guidance to operations console.

- **Levels 4 and 5:** Plant and corporate management: higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

Figure 20 shows the canonical form of a point definition; seven specialized forms support the most common kinds of control. Points are, in essence, object-oriented design elements that encapsulate information about control points of the process. Data associated with a point includes: Operating parameters, including current process value, set point (target value), valve output, and mode (automatic or manual); Tuning parameters, such as gain, reset, derivative, and alarm trip-points; Configuration parameters, including tag (name) and I/O channels.

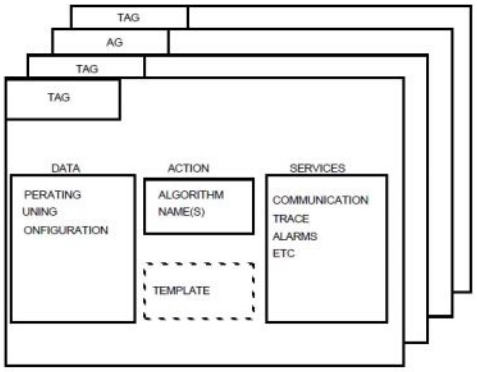


Figure 20: PROVOX – Object-oriented Elaboration

AN INTERPRETER USING DIFFERENT IDIOMS FOR THE COMPONENTS

Rule-based systems provide a means of codifying the problem-solving knowhow of human experts. These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme. Since these rules are not directly executable by available computers, systems for interpreting such rules must be provided. Hayes-Roth surveyed the architecture and operation of rule-based systems.

The basic features of a rule-based system, shown in Hayes-Roth's rendering as Figure 21, are essentially the features of a table-driven interpreter, as outlined earlier.

- The *pseudo-code* to be executed, in this case the knowledge base
- The *interpretation engine,* in this case the rule interpreter, the heart of the inference engine
- The *control state of the interpretation engine,* in this case the rule and data element selector
- The *current state of the program* running on the virtual machine, in this case the working memory.

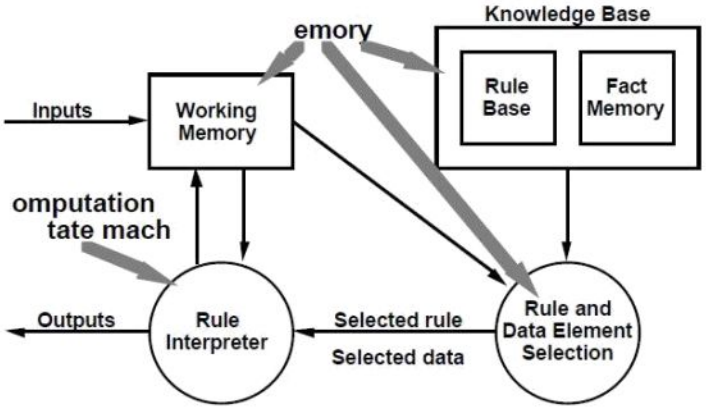


Figure 21: Basic Rule-Based System

Rule-based systems make heavy use of pattern matching and context (currently relevant rules). Adding special mechanisms for these facilities to the design leads to the more complicated view shown in Figure 22.

We see that:

- The knowledge base remains a relatively simple memory structure, merely gaining substructure to distinguish active from inactive contents.
- The rule interpreter is expanded with the interpreter idiom, with control procedures playing the role of the pseudo-code to be executed and the execution stack the role of the current program state.
- "Rule and data element selection" is implemented primarily as a pipeline that progressively transforms active rules and facts to prioritized activations.
- Working memory is not further elaborated.

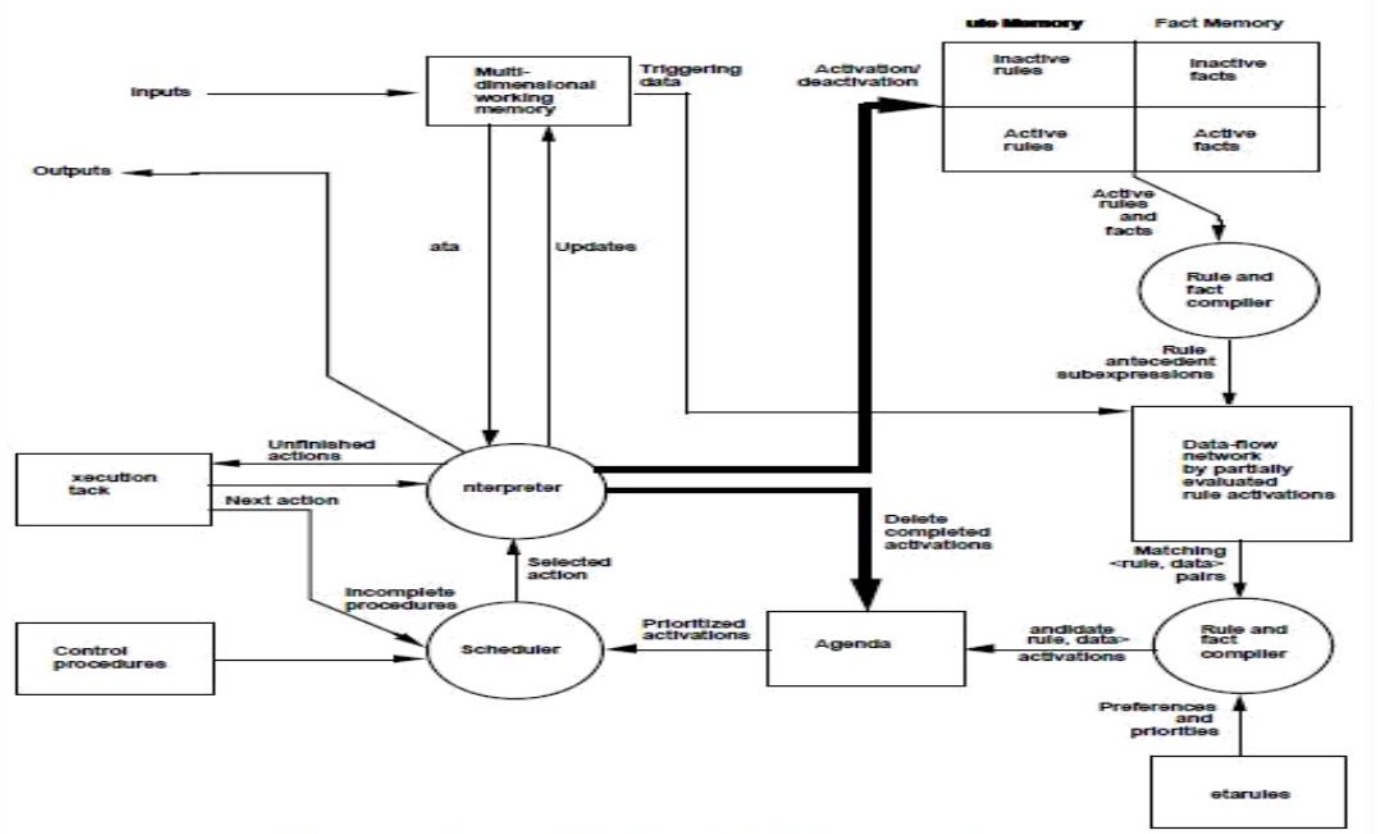


Figure 23: Sophisticated Rule-Based System

A BLACKBOARD GLOBALLY RECAST AS AN INTERPRETER

The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. In this model, the solution space is organized into several application-dependent hierarchies and the domain knowledge is partitioned into independent modules of knowledge that operate on knowledge within and between levels. Figure 24 showed the basic architecture of a blackboard system and outlined its three major parts: knowledge sources, the blackboard data structure, and control.

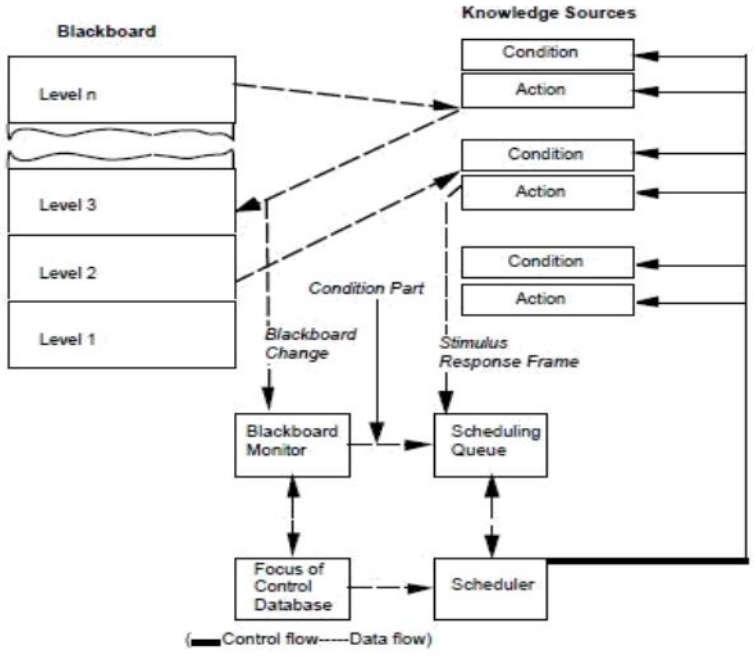


Figure 24: Hearsay-II

The first major blackboard system was the HEARSAY-II speech recognition system. Nii's schematic of the HEARSAY-II architecture appears as Figure 24. The blackboard structure is a six- to eight-level hierarchy in which each level abstracts information on its adjacent lower level and blackboard elements represent hypotheses about the interpretation of an utterance.

HEARSAY-II was implemented between 1971 and 1976; these machines were not directly capable of condition-triggered control, so it should not be surprising to find that an implementation provides the mechanisms of a virtual machine that realizes the implicit invocation semantics required by the blackboard model.

UNIT 2 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the architecture styles based on: (i)Data abstraction and object-oriented organization (ii)event-based, implicit invocation	Dec 09	10
2	What are the basic requirements for mobile robot architecture?	Dec 09	4
3	Explain the control loop solution for a mobile robot	Dec 09	6
4	Define architectural style. Mention any four commonly used styles	June 10	4
5	Consider the case study of building software controlled mobile robot. Describe its challenging problems and design considerations with four requirements. Finally give the solution by layered architecture for all the four requirements.	June 10	16
6	Define the following with an example: i)controlled variable ii)set point iii)open loop system iv)feedback control system v)feed forward control system	Dec 10	10
7	State the problem of KWIC. Propose implicit invocation and pipes and filters style to implement a solution for the same Explain in brief about KWIC (keyword in context) with shared data solution.	Dec 10 June 14	10
8	Discuss the importance and advantages of the following architectural styles with reference to an appropriate application area	June 11	8
9	List out the design considerations for mobile robotics case study. With the help of the design considerations, evaluate the pros and cons of the layered architecture and implicit invocation architecture for mobile robots	June 11	12
10	Discuss the invariants, advantages and disadvantages of pipes and filters architectural style	Dec 11 June 14	9
11	What are the basic requirements for a mobile robot's architecture? How the implicit invocation model handles them?	Dec 11	8
12	Write a note on heterogeneous architectures	Dec 11	3
13	Explain the process control paradigm with various process control definitions	June 12	6
14	List the basic requirements for mobile robot architecture	June 12	4
15	Explain the process control view of cruise control architectural style and obtain the complete cruise control system	June 12	10

UNIT 3
QUALITY

4.1 FUNCTIONALITY AND ARCHITECTURE

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

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

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

4.2 ARCHITECTURE AND QUALITY ATTRIBUTES

Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. For example:

- Usability involves both architectural and non-architectural aspects
- Modifiability is determined by how functionality is divided (architectural) and by coding techniques within a module (non-architectural).
- Performance involves both architectural and non-architectural dependencies

The message of this section is twofold:

- Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architectural level
- Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality.

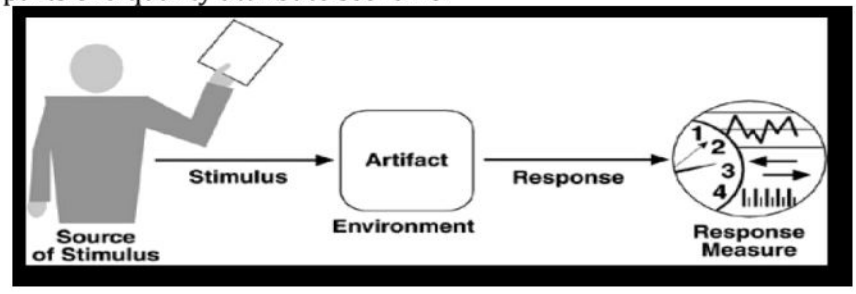
4.3 SYSTEM QUALITY ATTRIBUTES

QUALITY ATTRIBUTE SCENARIOS

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

1) ***Source of stimulus***. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
2) ***Stimulus***. The stimulus is a condition that needs to be considered when it arrives at a system.
3) ***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.
4) ***Artifact***. Some artifact is stimulated. This may be the whole system or some pieces of it.
5) ***Response***. The response is the activity undertaken after the arrival of the stimulus.
6) ***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

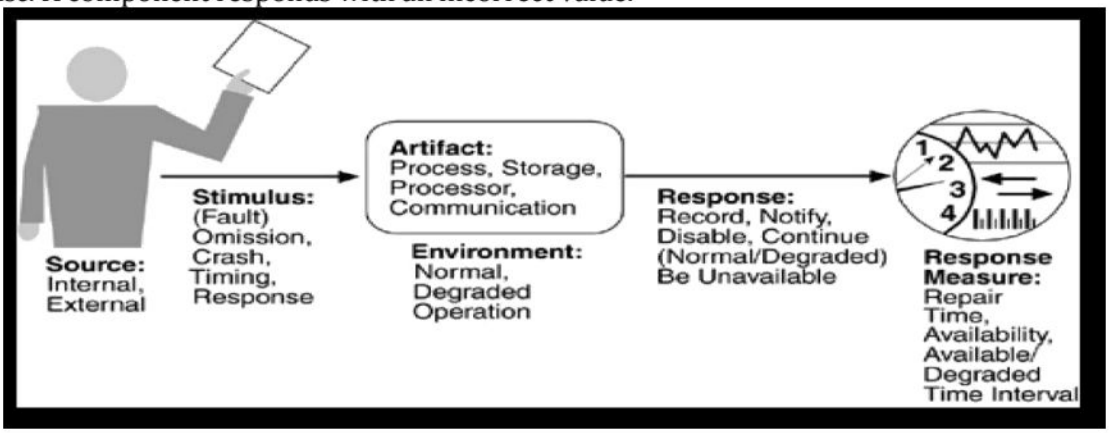
Availability is concerned with system failure and its associated consequences Failures are usually a result of system errors that are derived from faults in the system. It is typically defines as

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

Source of stimulus. We differentiate between internal and external indications of faults or failure since 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.

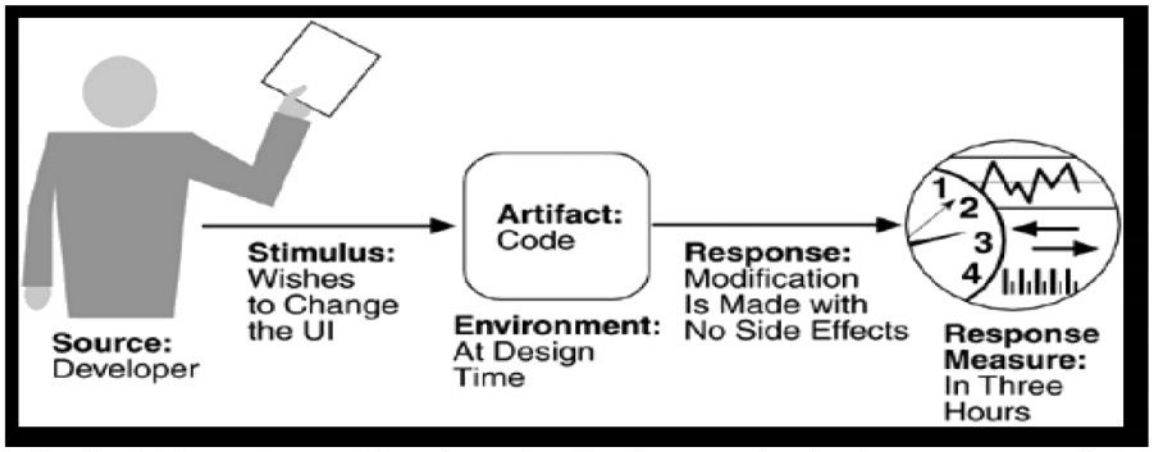
Response. There are a number of possible reactions to a system failure. These include logging the failure, notifying selected users or other systems, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair. In our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

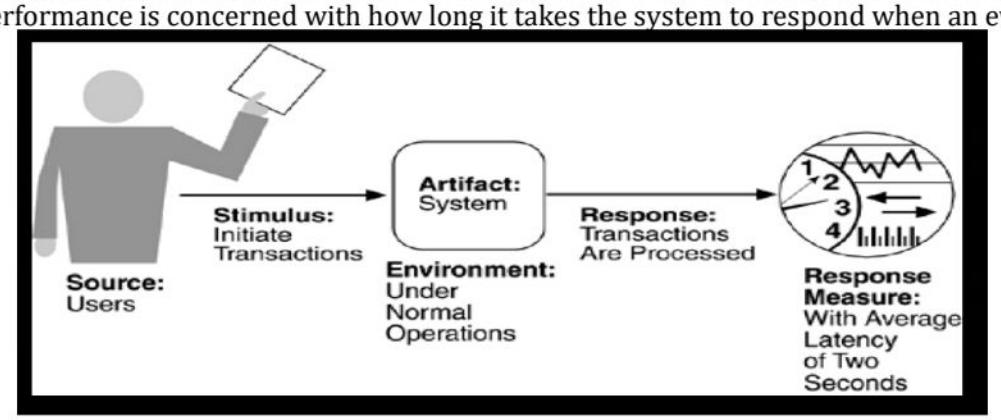
Environment. This portion specifies when the change can be made—design time, compile time, build time, initiation time, or runtime. In our example, the modification is to occur at design time.

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.

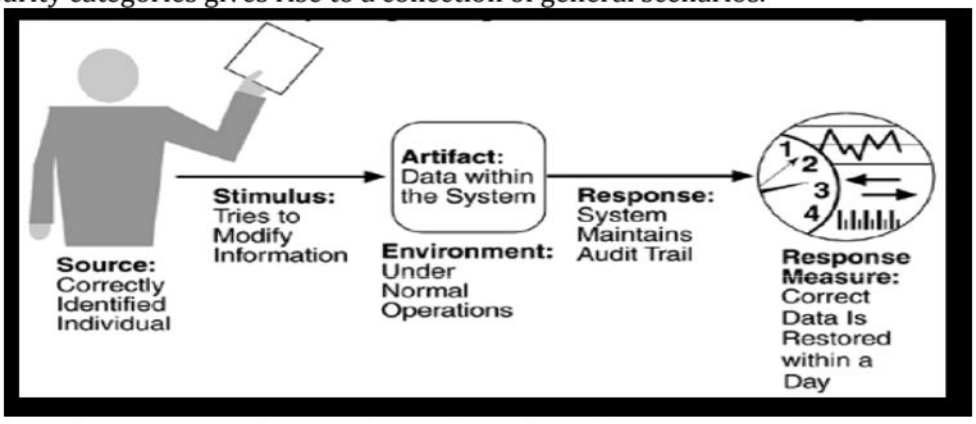
SECURITY SCENARIO

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.

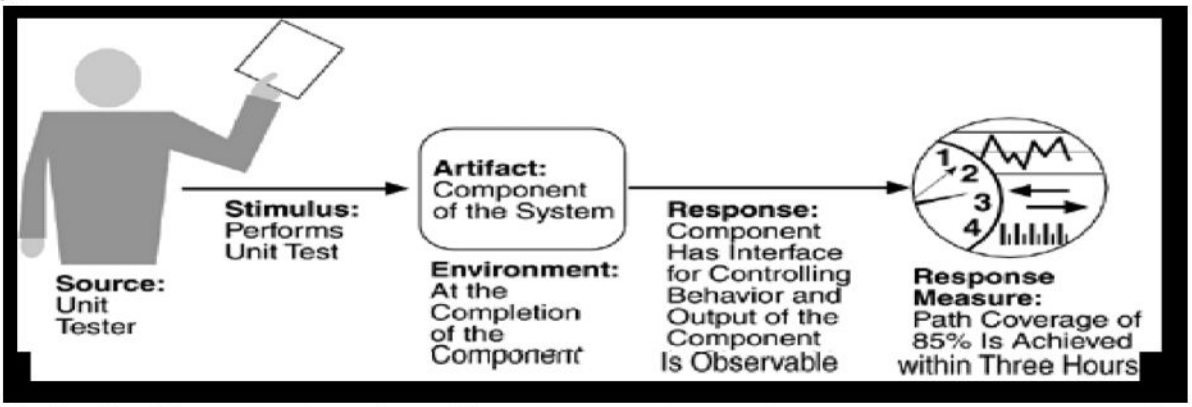
Environment. The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to the network.

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 statements.

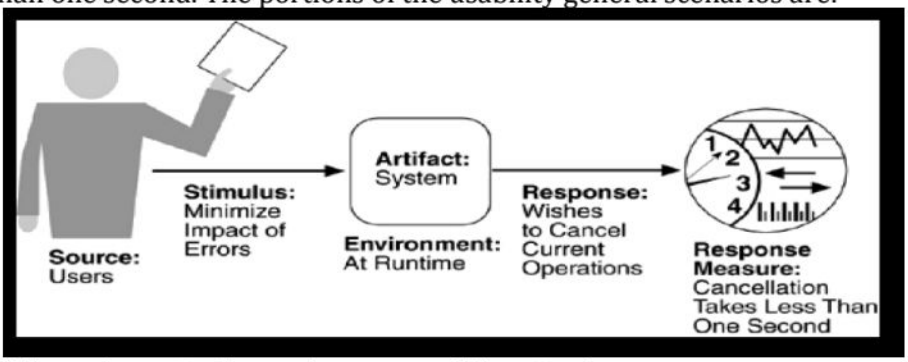
USABILITY SCENARIO

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

- ***Learning system features.*** If the user is unfamiliar with a particular system or a particular aspect of it, what 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. Table 4.7 gives the stimuli possible for each of the attributes and shows a number of different concepts. Some stimuli occur during runtime and others occur before. The problem for the architect is to understand which of these stimuli represent the same occurrence, which are aggregates of other stimuli, and which are independent.

Table 4.7. Quality Attribute Stimuli

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

4.5 OTHER SYSTEM QUALITY ATTRIBUTES

- SCALABILITY
- PORTABILITY

4.6 BUSINESS QUALITIES

- ***Time to market.***

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

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

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

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

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

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

4.7 ARCHITECTURE QUALITIES

- ***Conceptual integrity*** is the underlying theme or vision that unifies the design of the system at all levels. The architecture should do similar things in similar ways.
- ***Correctness*** **and** ***completeness*** are essential for the architecture to allow for all of the system'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.

5.1 INTRODUCING TACTICS

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

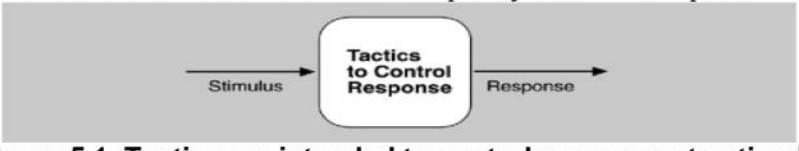
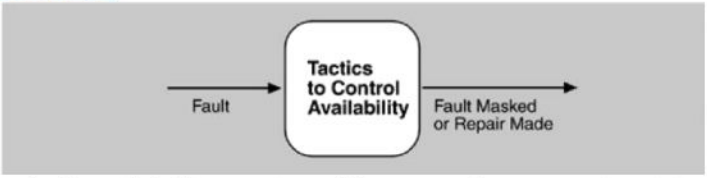


Figure 5.1. Tactics are intended to control responses to stimuli.

- *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.

5.2 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 nother 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

- **R*emoval 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.

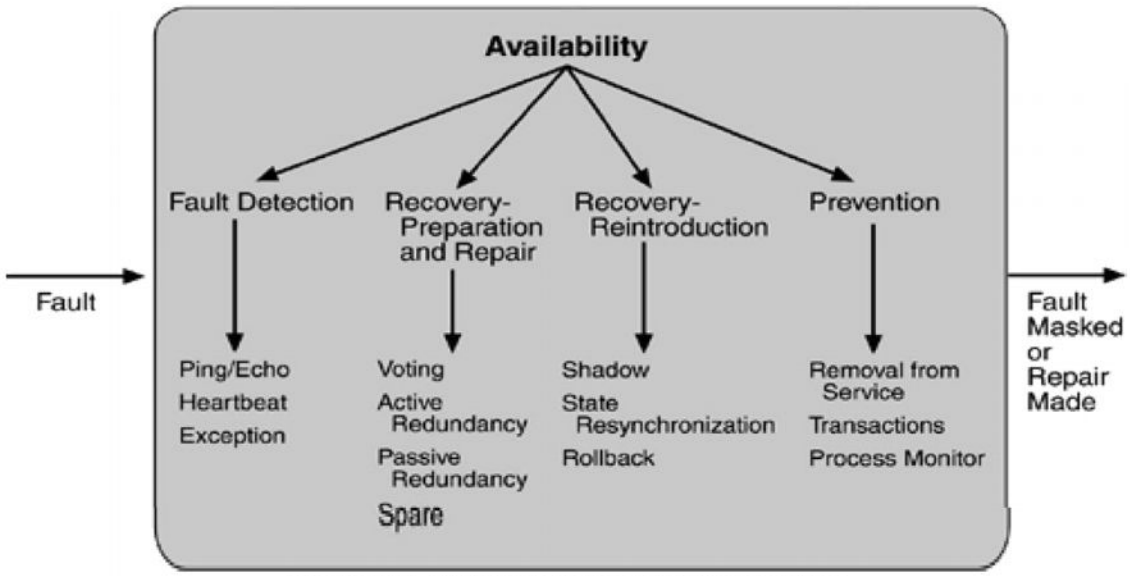
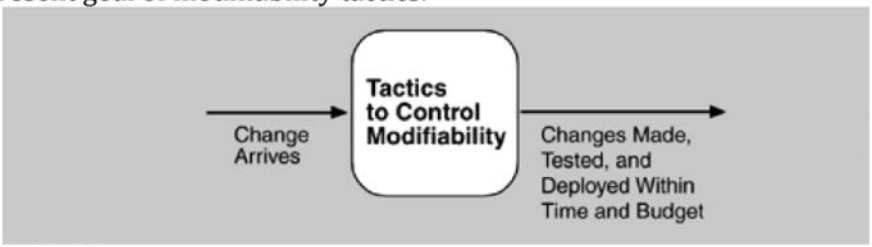


Figure 5.3. Summary of availability tactics

5.3 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:

1. Syntax of
   - *data.*
   - *service.*
2. Semantics of
   - *data.*
   - *service.*

3. Sequence of
   - *data.*
   - *control.*
4. *Identity of an interface of A*
5. *Location of A (runtime).*
6. *Quality of service/data provided by A.*
7. *Existence of A*
8. *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.

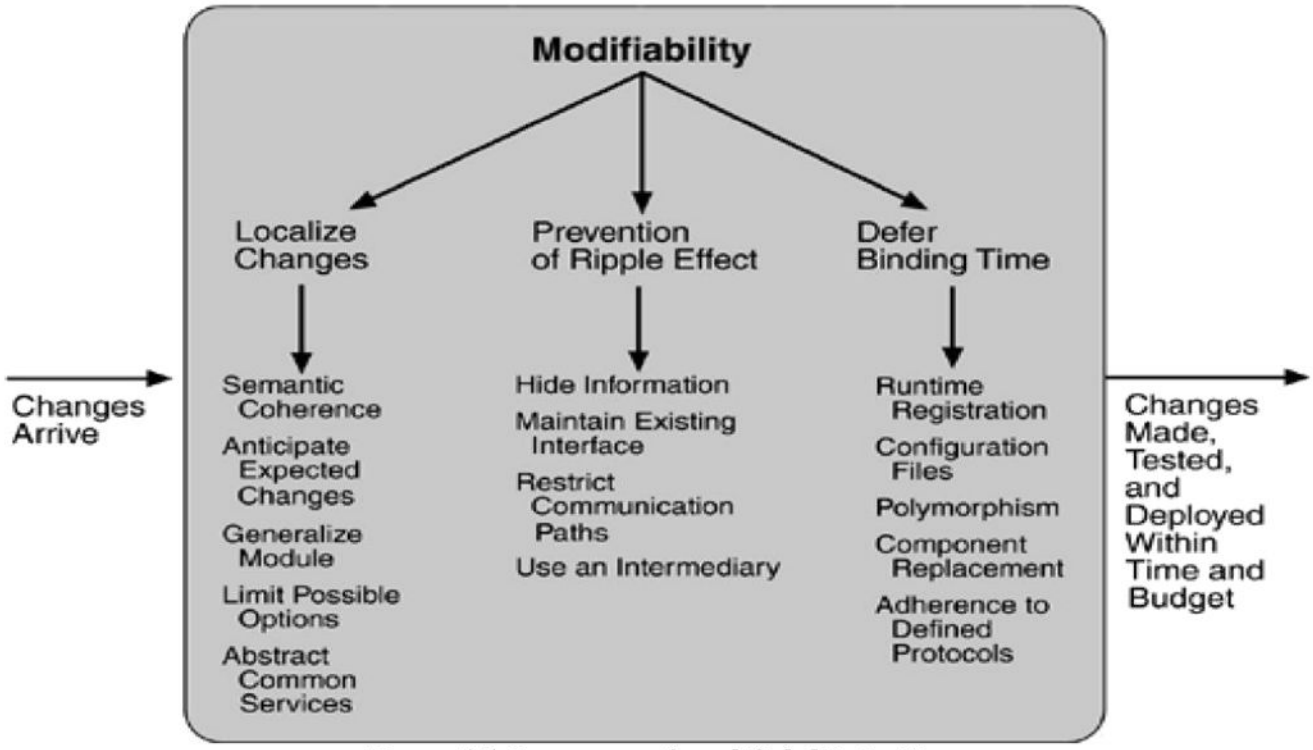
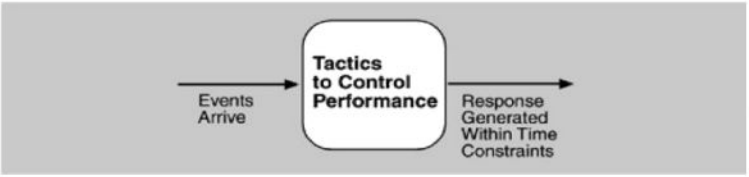


Figure 5.5. Summary of modifiability tactics

5.4 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 strategies are

- *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.

- ***Dynamic priority scheduling:***
  - *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.

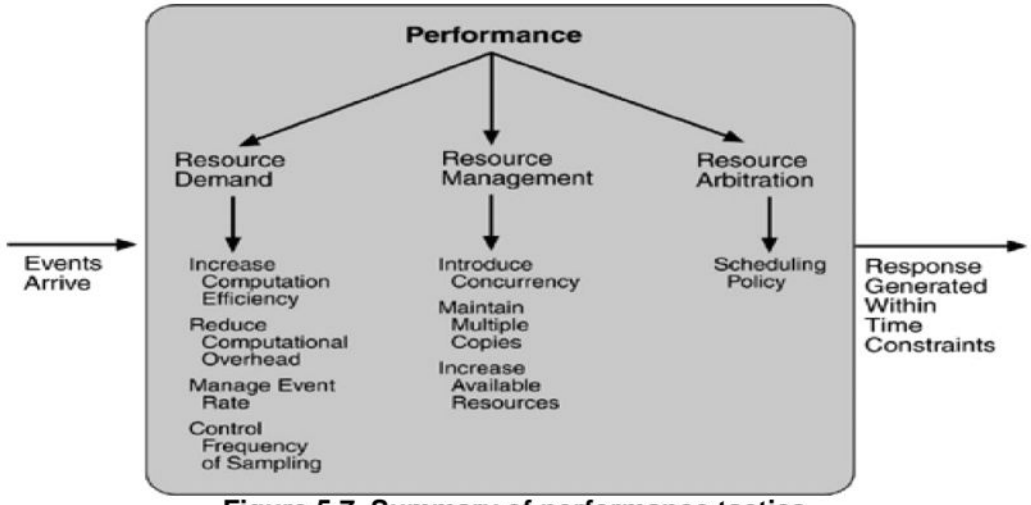
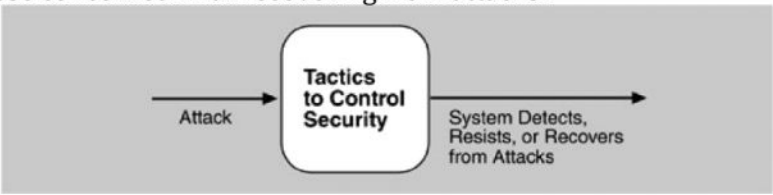


Figure 5.7. Summary of performance tactics

5.5 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.

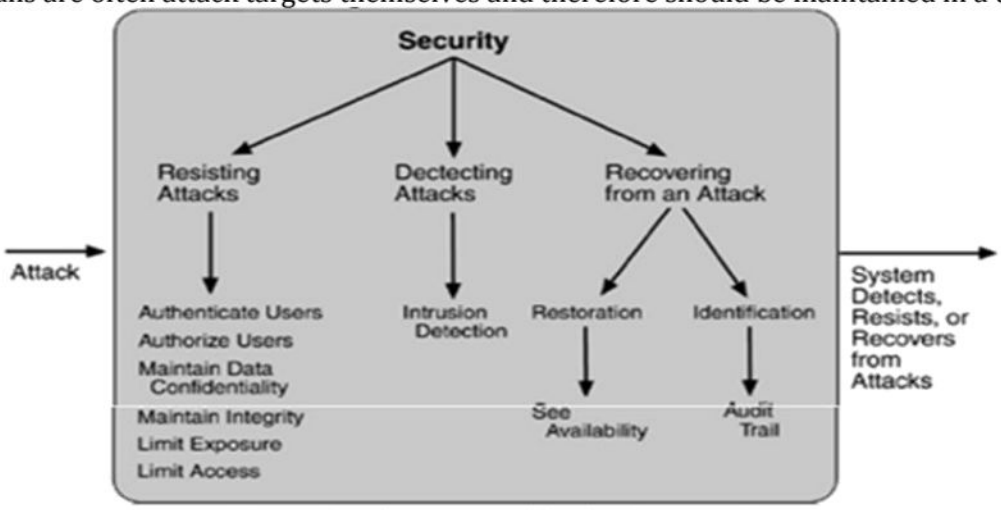
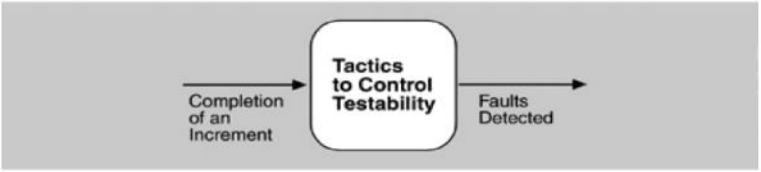


Figure 5.9. Summary of tactics for security

5.6 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.

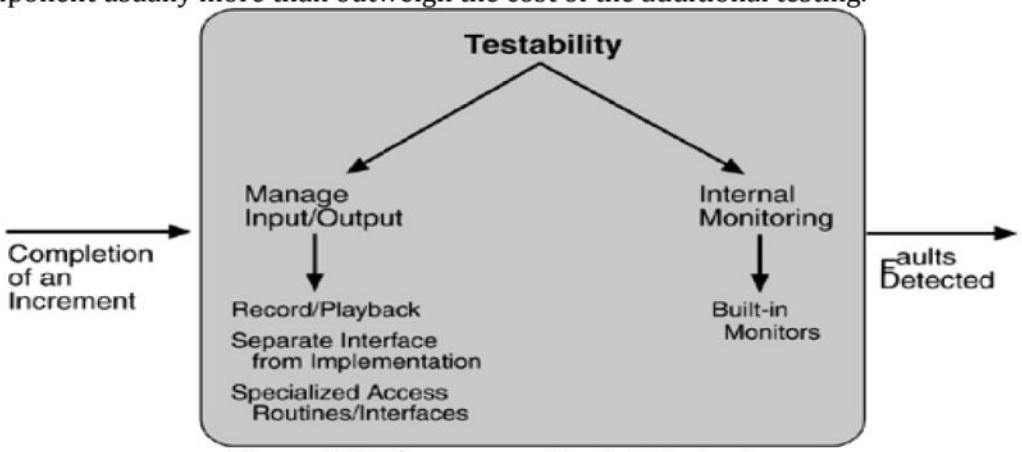
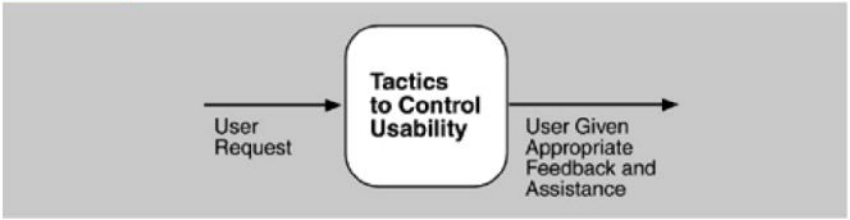


Figure 5.11. Summary of testability tactics

5.7 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

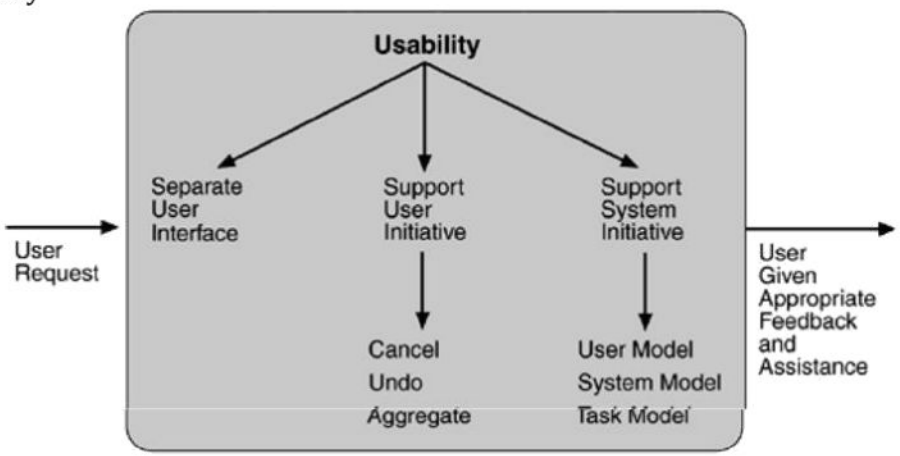


Figure 5.13. Summary of runtime usability tactics

5.8 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.

5.9 ARCHITECTURAL PATTERNS AND STYLES

An architectural pattern is determined by:

- A ***set of element types*** (such as a data repository or a component that computes a mathematical function).
- A **topological layout** of the elements indicating their interrelation-ships.
- A **set of semantic constraints** (e.g., filters in a pipe-and-filter style are pure data transducers—they incrementally transform their input stream into an output stream, but do not control either upstream or downstream elements).
- A **set of interaction mechanisms** (e.g., subroutine call, event-subscriber, blackboard) that determine how the elements coordinate through the allowed topology.

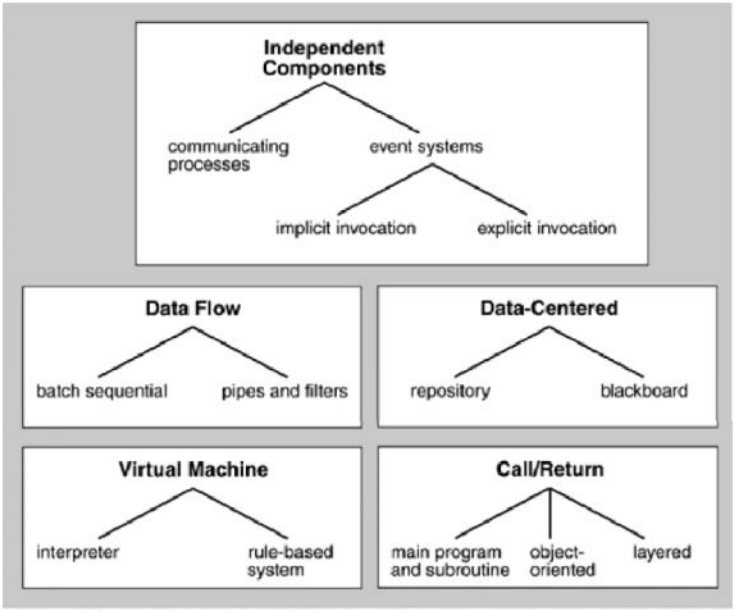


Figure 5.14. A small catalog of architectural patterns, organized by is-a relations

UNIT 3 - QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	What is availability? Explain the general scenario for availability.	Dec 09/ June 14	10
2	Classify security tactics. What are the different tactics for resisting attacks?	Dec 09	10
3	What are the qualities of the system? Explain the modifiability general scenario	June 10	10
4	What do you mean by tactics? Explain the availability tactics, with a neat diagram	June 10/ 14	10
5	What is a quality attribute scenario? List the parts of such a scenario. Distinguish between availability scenarios and modifiability scenarios	Dec 10	8
6	Explain how faults are detected and prevented	Dec 10	8
7	Write a brief note on the design time tactics	Dec 10	4
8	With the help of appropriate diagrams, explain the availability scenario and testability scenario in detail	June 11	12
9	Briefly discuss the various types of dependencies that one module can have on another which forms the basis for prevention of ripple effect	June 11	8
10	What are the qualities that the architecture itself should possess?	Dec 11	6
11	List the parts of quality attribute scenario	Dec 11	4
12	What is the goal of tactics for testability? Discuss the two categories of tactics for testing	Dec 11	10
13	What is quality attribute scenario? List the parts of scenario with an example	June 12	4
14	What is availability? Explain the general scenario for availability	June 12	8
15	Classify security tactics. What are the different tactics for resisting attacks?	June 12	8

UNIT 4
ARCHITECTURAL PATTERNS-1

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

2.1 INTRODUCTION

Architectural patterns represent the highest-level patterns in our pattern system. They help you to satisfy the fundamental structure of an application. We group our patterns into four categories:

- ***From Mud to Structure.*** The category includes the Layers pattern, the Pipes and Filters pattern and the Blackboard pattern.
- ***Distributed systems.*** This category includes one pattern, Broker and refers to two patterns in other categories, Microkernel and Pipes and Filters.
- ***Interactive systems.*** This category comprises two patterns, the Model-View-Controller pattern and the Presentation-Abstraction-Control pattern.
- ***Adaptable systems.*** The Reflection pattern and the Microkernel pattern strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

2.2 FROM MUD TO STRUCTURE

Before we start the design of a new system, we collect the requirements from the customer and transform them into specifications. The next major technical task is to define the architecture of the system. This means finding a high level subdivision of the system into constitute parts.

We describe three architectural patterns that provide high level system subdivisions of different kinds:

- The ***Layers*** pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
- The ***Pipes and Filters*** pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.
- The ***Blackboard*** pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

LAYERS

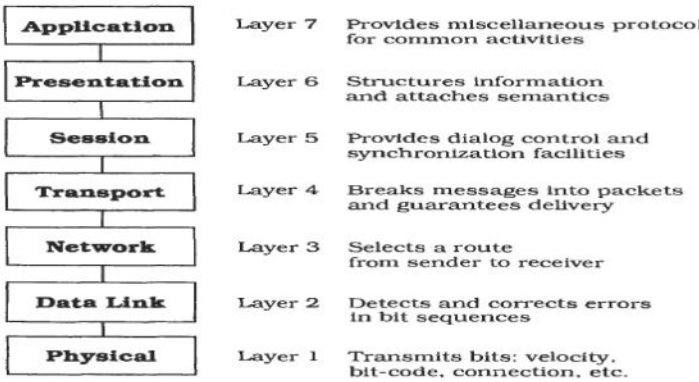
The layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Example: Networking protocols are best example of layered architectures. Such a protocol consists of a set of rules and conventions that describes how computer programmer communicates across machine boundaries. The format, contacts and meaning of all messages are defined. The protocol specifies agreements at a variety of abstraction levels, ranging from the details of bit transmission to high level abstraction logic.

Therefore the designers use secured sub protocols and arrange them in layers. Each layer deals with a specific aspect of communication and users the services of the next lower layer. (see diagram & explain more)

Context: a large system that requires decomposition.

***Problem*: THE SYSTEM WE ARE BUILDING IS DIVIDED BY MIX OF LOW AND HIGH LEVEL ISSUES, WHERE HIGH-LEVEL OPERATIONS RELY ON THE LOWER-LEVEL ONES. FOR EX, HIGH-LEVEL WILL BE INTERACTIVE TO USER AND LOW-LEVEL WILL BE CONCERNED WITH HARDWARE IMPLEMENTATION.**



In such a case, we need to balance the following forces:

- Late source code changes should not ripple through the systems. They should be confined to one component and not affect others.
- Interfaces should be stable, and may even be impressed by a standards body.
- Parts of the system should be exchangeable (i.e, a particular layer can be changed).
- It may be necessary to build other systems at a later date with the same low-level issues as the system you are currently designing.
- Similar responsibilities should be grouped to help understandability and maintainability.
- There is no 'standard' component granularity.
- Complex components need further decomposition.
- Crossing component boundaries may impede performance, for example when a substantial amount of data must be transferred over several boundaries.
- The system will be built by a team of programmers, and works has to be subdivided along clear boundaries.

Solution:

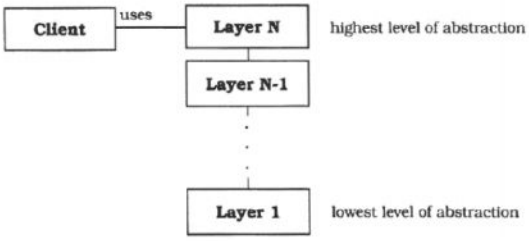
- Structure your system into an appropriate number of layers and place them on top of each other.
- Lowest layer is called 1 (base of our system), the highest is called layer N. i.e, Layer 1, ……. Layer J-1, Layer J, ….. Layer N.
- Most of the services that layer J Provides are composed of services provided by layer J-1. In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, layer J's services may depend on other services in layer J.

Structure:

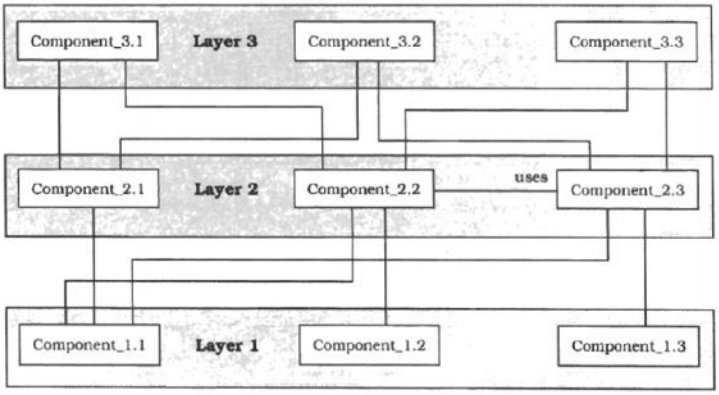
- An individual layer can be described by the following CRC card:

Class Layer J	***Collaborator*** • Layer J-1
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	

- The main structural characteristics of the layers patterns is that services of layer J are only use by layer J+1-there are no further direct dependencies between layers. This structure can be compared with a stack, or even an onion. Each individual layer shields all lower from direct access by higher layers.

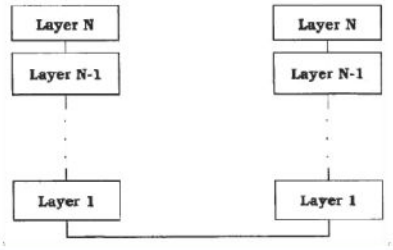


- In more detail, it might look something like this:



Dynamics:

- **Scenario I** is probably the best-known one. A client Issues a request to Layer N. Since Layer N cannot carry out the request on its own. It calls the next Layer N - 1 for supporting subtasks. Layer N - I provides these. In the process sending further requests to Layer N-2 and so on until Layer I is reached. Here, the lowest-level services are finally performed. If necessary, replies to the different requests are passed back up from Layer 1 to Layer 2, from Layer 2 to Layer 3, and so on until the final reply arrives at Layer N.
- **Scenario II** illustrates bottom-up communication-a chain of actions starts at Layer 1, for example when a device driver detects input. The driver translates the input into an internal format and reports it to Layer 2 which starts interpreting it, and so on. In this way data moves up through the layers until it arrives at the highest layer. While top-down information and control flow are often described as 'requests'. Bottom-up calls can be termed 'notifications'.
- **Scenario III** describes the situation where requests only travel through a subset of the layers. A top-level request may only go to the next lower level N- 1 if this level can satisfy the request. An example of this is where level N- 1 acts as a cache and a request from level N can be satisfied without being sent all the way down to Layer 1 and from here to a remote server.
- **Scenario IV** An event is detected in Layer 1, but stops at Layer **3** instead of travelling all the way up to Layer N. In a communication protocol, for example, a resend request may arrive from an impatient client who requested data some time ago. In the meantime the server has already sent the answer, and the answer and the re-send request cross. In this case, Layer **3** of the server side may notice this and intercept the re-send request without further action.
- **Scenario V** involves two stacks of N layers communicating with each other. This scenario is well-known from communication protocols where the stacks are known as 'protocol stacks'. In the following diagram, Layer N of the left stack issues a request. 'The request moves down through the layers until it reaches Layer **1,** is sent to Layer 1 of the right stack, and there moves up through the layers of the right stack. The response to the request follows the reverse path until it arrives at Layer N of the left stack.



Implementation:

The following steps describe a step-wise refinement approach to the definition of a layered architecture.

★ **Define the abstraction criterion for grouping tasks into layers.**
- This criterion is often the conceptual distance from the platform (sometimes, we encounter other abstraction paradigm as well).
- In the real world of software development we often use a mix of abstraction criterions. For ex, the distance from the hardware can shape the lower levels, and conceptual complexity governs the higher ones.
- Example layering obtained using mixed model layering principle is shown below User-visible elements
  - ♠ Specific application modules
  - ♠ Common services level
  - ♠ Operating system interface level
  - ♠ Operating system
  - ♠ Hardware

★ **Determine the number of abstraction levels according to your abstraction criterion.**
- Each abstraction level corresponds to one layer of the pattern.
- Having too many layers may impose unnecessary overhead, while too few layers can result in a poor structure.

★ **Name the layers and assign the tasks to each of them.**
- The task of the highest layer is the overall system task, as perceived by the client.
- The tasks of all other layers are to be helpers to higher layers.

★ **Specify the services**
- It is often better to locate more services in higher layers than in lower layers.
- The base layers should be kept 'slim' while higher layers can expand to cover a spectrum of applicability.
- This phenomenon is also called the '*inverted pyramid of reuse*'.

★ **Refine the layering**
- Iterate over steps 1 to 4.
- It is not possible to define an abstraction criterion precisely before thinking about the implied layers and their services.
- Alternatively it is wrong to define components and services first and later impose a layered structure.
- The solution is to perform the first four steps several times until a natural and stable layering evolves.

★ **Specify an interface for each layer.**
- If layer J should be a 'black box' for layer J+1, design a flat interface that offers all layer J's services.
- 'White box' approach is that in which, layer J+1 sees the internals of layer J.
- 'Gray box' approach is a compromise between black and white box approaches. Here layer J+1 is aware of the fact that layer J consists of three components, and address them separately, but does not see the internal workings of individual components.

★ **Structure individual layers**
- When an individual layer is complex, it should be broken into separate components.
- This subdivision can be helped by using finer-grained patterns.

★ **Specify the communication between adjacent layers.**
- Push model (most often used): when layer J invokes a service of layer J+1, any required information is passed as part of the service call.
- Pull model: it is the reverse of the push model. It occurs when the lower layer fetches available information from the higher layer at its own discretion.

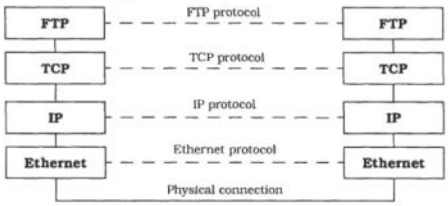
★ **Decouple adjacent layers.**
- For top-down communication, where requests travel top-down, we can use one-way coupling (i.e, upper layer is aware of the next lower layer, but the lower layer is unaware of the identity of its users) here return values are sufficient to transport the results in the reverse direction.

- For bottom-up communication, you can use callbacks and still preserve a top-down one way coupling. Here the upper layer registers callback functions with the lower layer.
- We can also decouple the upper layer from the lower layer to a certain degree using object oriented techniques.

★ **Design an error handling strategy**
- An error can either be handled in the layer where it occurred or be passed to the next higher layer.
- As a rule of thumb, try to handle the errors at the lowest layer possible.

Example resolved:

The most widely-used communication protocol, TCP/IP, does not strictly conform to the OSI model and consists of only four layers: TCP and IP constitute the middle layers, with the application at the top and the transport medium at the bottom. A typical configuration, that for the **UNIX** ftp's utility, is shown below:



Variants:

★ **Relaxed layered system:**
- Less restrictive about relationship between layers. Here each layer may use the services of all layers below it, not only of the next lower layer.
- A layer may also be partially opaque i.e. some of its services are only visible to the next higher layer, while others are visible to all higher layers.
- Advantage: flexibility and performance.
- Disadvantage: maintainability.

★ **Layering through inheritance**
- This variant can be found in some object oriented systems.
- Here, lower layers are implemented as base classes. A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.
- Advantage: higher layers can modify lower-layer services according to their needs.
- Disadvantage: closely ties the higher layer to the lower layer.

Known uses:

★ **Virtual machines**: we can speak of lower levels as a virtual machine that insulates higher levels from low-level details or varying hardware. E.g. Java virtual machine.

★ **API'S**: An API is a layer that encapsulates lower layers of frequently-used functionality. An API is usually a flat collection Specifications, such as the UNIX system calls.

★ **Information system (IS):** IS from the business software domain often use a two-layer architecture. The bottom layer is a database that holds company-specific data. This architecture can be split into four-layer architecture. These are, from highest to lowest:
- ♠ Presentation
- ♠ Application logic
- ♠ Domain layer
- ♠ Database

★ **Windows NT:** This OS is structured according to the microkernel pattern. It has the following layers:
- ♠ System services
- ♠ Resource management layer
- ♠ Kernel
- ♠ HAL(hardware abstraction layer)
- ♠ Hardware

Consequences:

The layers pattern has several ***benefits***:

- **Reuse of layers**
  - If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts.
  - However, developers often prefer to rewrite its functionality in order to avoid higher costs of reusing existing layers.
- **Support for standardization**
  - Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces.
  - Different implementations of the same interface can then be used interchangeably. This allows you to use products from different vendors in different layers.
- **Dependencies are kept local**
  - Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed.
- **Exchangeability**
  - Individual layer implementations can be replaced by semantically-equivalent implementations without too great an effort.

The layers pattern has several ***liabilities:***

- **Cascades of changing behavior**
  - A severe problem can occur when the behavior of the layer changes.
  - The layering becomes a disadvantage if you have to do a substantial amount of rework on many layers to incorporate an apparently local change.
- **Lower efficiency**
  - If high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate layers, and may be transformed several times. Therefore, layered architecture is less efficient than a monolithic structure or a 'sea of objects'.
- **Unnecessary work**
  - If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer this has a negative impact on performance.
  - De-multiplexing in a communication protocol stack is an example of this phenomenon.
- **Difficulty of establishing the correct granularity of layers**
  - The decision about the granularity of layers and the assignment of task to layers is difficult, but is critical for the architecture.

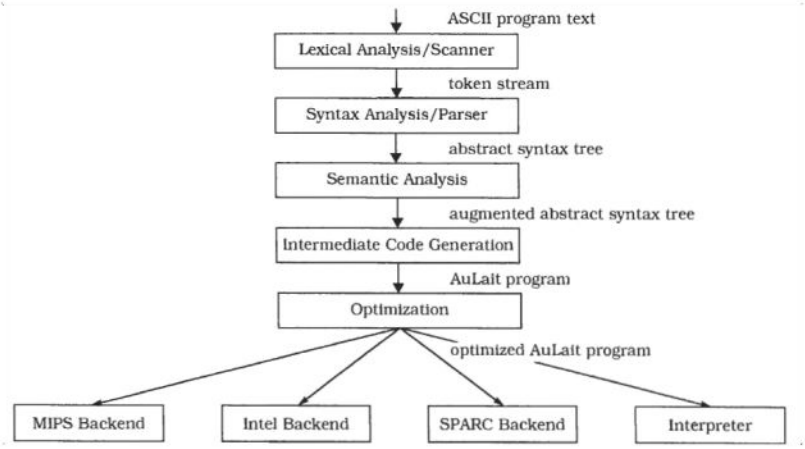
PIPES AND FILTERS

The pipes and filter's architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

Example:

Suppose we have defined a new programming language called **Mocha** [*Modular Object Computation with Hypothetical Algorithms*]. Our task is to build a portable compiler for this language. To support existing and future hardware platforms we define an intermediate language **AuLait** [*Another Universal Language for Intermediate Translation*] running on a virtual machine Cup (Concurrent Uniform Processor).

Conceptually, translation from Mocha to AuLait consists of the phases lexical analysis, syntax analysis, semantic analysis, intermediate-code generation (AuLait), and optionally intermediate-code optimization. Each stage has well-defined input and output data.



Context:

Processing data streams.

Problem:

- Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons: the system has to be built by several developers, the global system task decomposes naturally into several processing stages, and the requirements are likely to change.
- The design of the system-especially the interconnection of the processing steps-has to consider the following forces:
  - Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
  - Small processing steps are easier to reuse in different contexts than larger contexts.
  - Non-adjacent processing steps do not share information.
  - Different sources of input data exists, such as a network connection or a hardware sensor providing temperature readings, for example.
  - It should be possible to present or store the final results in various ways.
  - Explicit storage of intermediate results for further processing the steps, for example running them in parallel or quasi-parallel.
  - You may not want to rule out multi-processing the steps

Solution:

- The pipes and filters architectural pattern divides the task of a system into several sequential processing steps (connected by the dataflow through the system).
- Each step is implemented by a filter component, which consumes and delivers data incrementally to achieve low battery and parallel processing.
- The input to a system is provided by a data source such as a text file.
- The output flows into a data sink such as a file, terminal, and so on.
- The data source, the filters, and the data sink are connected sequentially by pipes. Each pipe implements the data flow between adjacent processing steps.
- The sequence of filters combined by pipes is called a processing pipeline.

Structure:

- **Filter component:**
  - Filter components are the processing units of the pipeline.
  - A filter enriches, refines or transforms its input data. It enriches data by computing and adding information, refines data by concentrating or extracting information, and transforms data by delivering the data in some other representation.
  - It is responsible for the following activities:
    - The subsequent pipeline element pulls output data from the filter. (passive filter)

- The previous pipeline element pushes new input data to the filter. (passive filter)
- Most commonly, the filter is active in a loop, pulling its input from and pushing its output down the pipeline. (active filter)

Class: Filter	Collaborators: • Pipe
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	

★ **Pipe component:**

- Pipes denote the connection between filters, between the data source and the first filter, and between the last filter and the data sink.
- If two active components are joined, the pipe synchronizes them.
- This synchronization is done with a first-in- first-out buffer.

Class: Pipe	Collaborators: • Data Source • Data Sink • Filter
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	

★ **Data source component:**

- The data source represents the input to the system, and provides a sequence of data values of the same structure or type.
- It is responsible for delivering input to the processing pipeline.

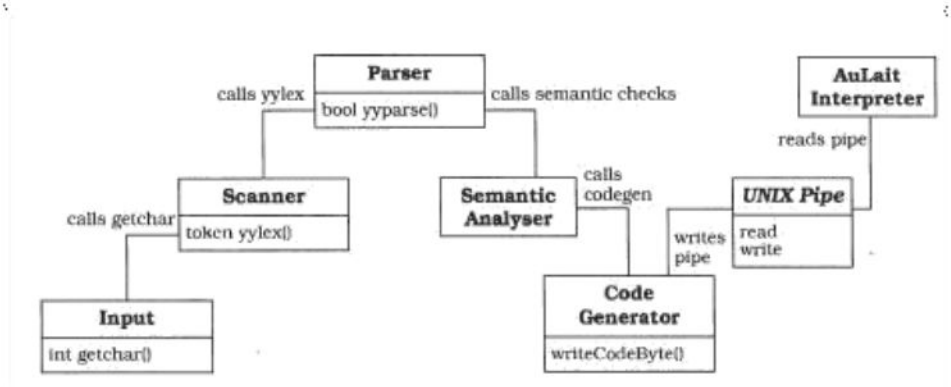
Class: Data Source	Collaborators: • Pipe
Responsibility • Delivers input to processing pipeline.	

★ **Data sink component:**

- The data sink collects the result from the end of the pipeline (i.e, it consumes output).
- Two variants of data sink:
  - An active data sink pulls results out of the preceding processing stage.
  - An passive data sink allows the preceding filter to push or write the results into it.

Class: Data Sink	Collaborators: • Pipe
Responsibility • Consumes output.	

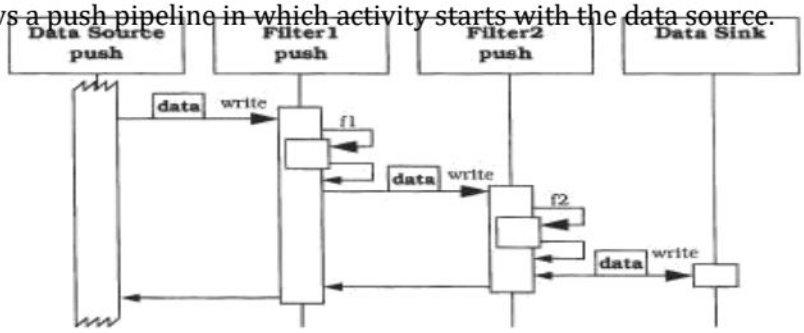
In our Mocha compiler we use the UNIX tools lex and yacc to implement the first two stages of the compiler. The connection to the other frontend stages consists of many procedure calls embedded in the grammar action rules, and not just simple data flow. The backends and interpreter run as separate programs to allow exchangeability. They are connected via a UNIX pipe to the frontend.



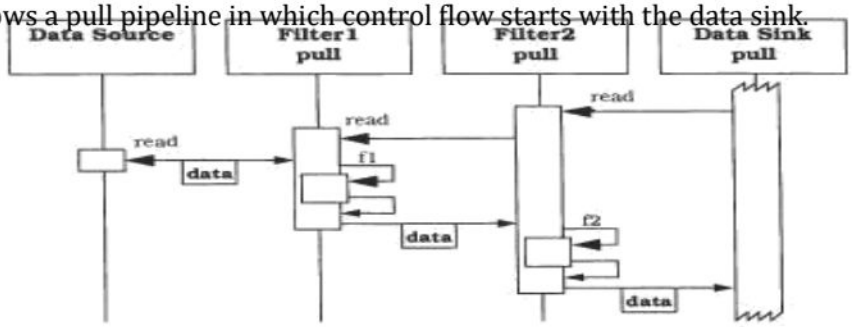
Dynamics:

The following scenarios show different options for control flow between adjacent filters.

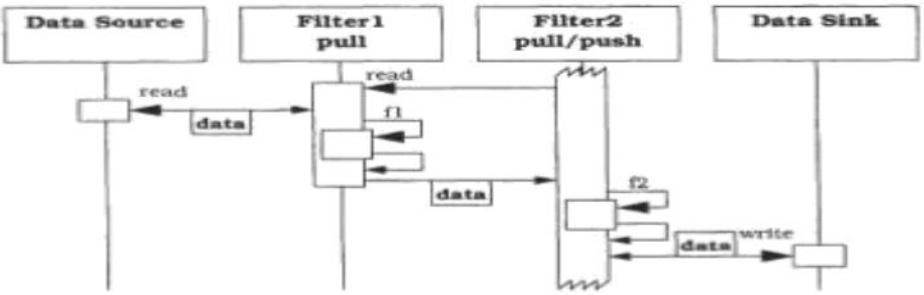
★ **Scenario 1** shows a push pipeline in which activity starts with the data source.



★ **Scenario 2** shows a pull pipeline in which control flow starts with the data sink.



★ **Scenario 3** shows a mixed push-pull pipeline with passive data source and sink. Here second filter plays the active role and starts the processing.

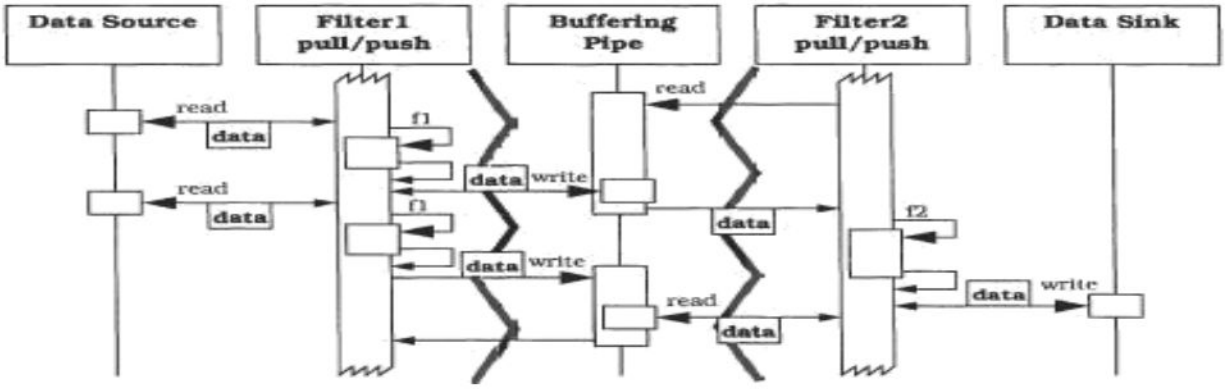


★ **Scenario 4** shows a more complex but typical behavior of pipes and filter system. All filters actively pull, compute, and push data in a loop.

★ The following steps occur in this scenario:

- ♠ Filter 2 tries to get new data by reading from the pipe. Because no data is available the data request suspends the activity of Filter 2-the buffer is empty.
- ♠ Filter 1 pulls data from the data source and performs function f 1.
- ♠ Filter 1 then pushes the result to the pipe.
- ♠ Filter 2 can now continue, because new input data is available.

- Filter l can also continue, because it is not blocked by a full buffer within the pipe.
- Filter 2 computes f 2 and writes its result to the data sink.
- In parallel with Filter 2's activity, Filter 1 computes the next result and tries to push it down the pipe. This call is blocked because Filter 2 is not waiting for data-the buffer is full.
- Filter 2 now reads new input data that is already available from the pipe. This releases Filter 1 so that it can now continue its processing.



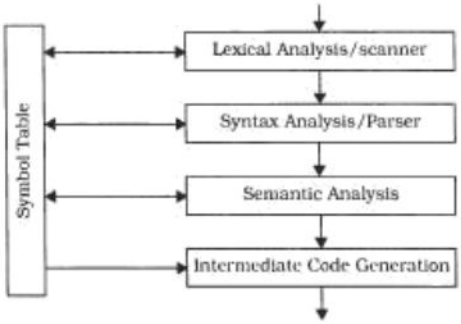
Implementation:

- **Divide the system's tasks into a sequence of processing stages.**
  - Each stage must depend only on the output of its direct predecessor.
  - All stages are conceptually connected by the data flow.
- **Define the data format to be passed along each pipe.**
  - Defining a uniform format results in the highest flexibility because it makes recombination of its filters easy.
  - You must also define how the end of input is marked.
- **Decide how to implement each pipe connection.**
  - This decision directly determines whether you implement the filters as active or passive components.
  - Adjacent pipes further define whether a passive filter is triggered by push or pull of data.
- **Design and implement the filters.**
  - Design of a filter component is based both on the task it must perform and on the adjacent pipes.
  - You can implement passive filters as a function, for pull activation, or as a procedure for push activation.
  - Active filters can be implemented either as processes or as threads in the pipeline program.
- **Design the error handling.**
  - Because the pipeline components do not share any global state, error handling is hard to address and is often neglected.
  - As a minimum, error detection should be possible. UNIX defines specific output channel for error messages, standard error.
  - If a filter detects error in its input data, it can ignore input until some clearly marked separation occurs.
  - It is hard to give a general strategy for error handling with a system based on the pipes and filter pattern.
- **Set up the processing pipeline.**
  - If your system handles a single task you can use a standardized main program that sets up the pipeline and starts processing.
  - You can increase the flexibility by providing a shell or other end-user facility to set up various pipelines from your set of filter components.

Example resolved:

We did not follow the Pipes and Filters pattern strictly in our Mocha compiler by implementing all phases of the compiler as separate filter programs connected by pipes. We combined the first four compiler phases into a

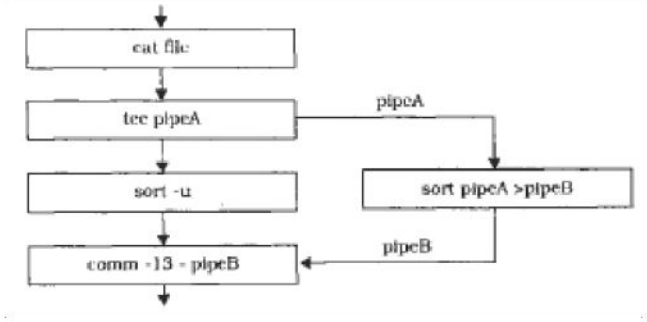
single program because they all access and modify the symbol table. This allows us to implement the pipe connection between the scanner and the parser as a simple function call.



Variants:

- Tee and join pipeline systems:
- The single-input single-output filter specification of the Pipes and Filters pattern can be varied to allow filters with more than one input and/or more than one output.
- For example, to build a sorted list of all lines that occur more than once In a text file. we can construct the following shell program:

```
# first create two auxiliary named pipes to be
used mknod pipeA p
mknod pipeB p
# now do the processing using available UNIX filters
# start side fork of processing in background:
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



Known uses:

★ **UNIX** [Bac86] popularized the Pipes and Filters paradigm. The flexibility of UNIX pipes made the operating system a suitable plafform for the binary reuse of filter programs and for application integration.

★ **CMS Pipelines** [HRV95] is an extension to the operating system of IBM mainframes to support Pipes and Filters architectures. It provides a reuse and integration platform in the same way as UNIX.

★ **LASSPTools** [Set95] is a toolset to support numerical analysis and graphics. The toolset consists mainly of filter programs that can be combined using UNIX pipes.

Consequences:

The Pipes and Filters architectural pattern has the following **benefits**

★ **No intermediate files necessary, but possible.**
- Computing results using separate programs is possible without pipes, by storing intermediate results in pipes.

★ **Flexibility by the filter change**
- Filters have simple interface that allows their easy exchange within a processing pipeline.

★ **Flexibility by recombination**
- This benefit combined with reusability of filter component allows you to create new processing pipelines by rearranging filters or by adding new ones.

- **Reuse of filter components.**
  - Support for recombination leads to easy reuse of filter components.
- **Rapid prototyping of pipelines.**
  - Easy to prototype a data processing system from existing filters.
- **Efficiency by parallel processing**.
  - It is possible to start active filter components in parallel in a multiprocessor system or a network.

The Pipes and Filters architectural pattern has the following **Liabilities**

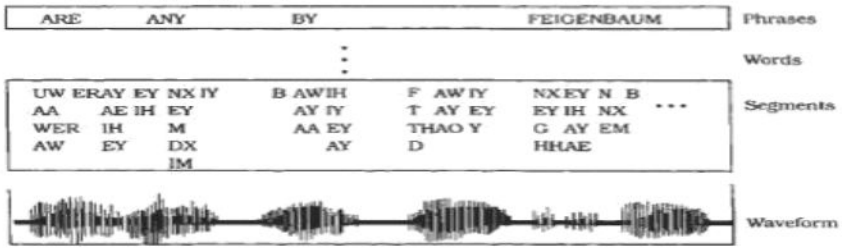
- **Sharing state information is expensive or inflexible**
  - This pattern is inefficient if your processing stage needs to share a large amount of global data.
- **Efficiency gain by parallel processing is often an illusion**, because:
  - The cost for transferring data between filters is relatively high compared to the cost of the computation carried out by a single filter.
  - Some filter consumes all their input before producing any output.
  - Context-switching between threads or processes is expensive on a single processor machine.
  - Synchronization of filters via pipes may start and stop filters often.
- **Data transformation overhead**
  - Using a single data type for all filter input and output to achieve highest flexibility results in data conversion overheads.
- **Error handling**
  - Is difficult. A concrete error-recovery or error-handling strategy depends on the task you need to solve.

BLACKBOARD

The blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. In blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Example:

Consider a software system for speech recognition. The input to the system is speech recorded as a waveform. The system not only accepts single words, but also whole sentences that are restricted to the syntax and vocabulary needed for a specific application, such as a database query. The desired output is a machine representation of the corresponding English phrases.



Context:

An immediate domain in which no closed approach to a solution is known or feasible

Problem:

- A blackboard pattern tackle problems that do not have a feasible deterministic solution for the transformation of raw data into high level data structures, such as diagrams, tables, or English phrases.
- Examples of domains in which such problems occur are:- vision, image recognition, and speech recognition.
- They are characterized by problems that when decomposed into sub problems, spans several fields of expertise.
- The Solution to the partial problems requires different representations and paradigm.

★ The following *forces* influence solutions to problems of this kind:

- ✓ A complete search of the solution space is not feasible in a reasonable time.
- ✓ Since the domain is immature, we may need to experiment with different algorithms for the same subtask. Hence, individual modules should be easily exchangeable.
- ✓ There are different algorithms that solve partial problems.
- ✓ Input as well as intermediate and final results, have different representation, and the algorithms are implemented according to different paradigms.
- ✓ An algorithm usually works on the results of other algorithms.
- ✓ Uncertain data and approximate solutions are involved.
- ✓ Employing dis fact algorithms induces potential parallelism.

Solution:

★ The idea behind the blackboard architecture is a collection of independent programs that work co operatively on a common data structure.

★ Each program is meant for solving a particular part of overall task.

★ These programs are independent of each other they do not call each other, nor there is a predefined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of progress.

★ A central control component evaluates the current state of processing and coordinates these programs.

★ These data directed control regime is referred to as *opportunistic problem solving.*

★ The set of all possible solutions is called *solution space* and is organized into levels of abstraction.

★ The name 'blackboard' was chosen because it is reminiscent of the situation in which human experts sit in front of a real blackboard and work together to solve a problem.

★ Each expert separately evaluates the current state of the solution, and may go up to the blackboard at any time and add, change or delete information.

★ Humans usually decide themselves who has the next access to the blackboard.

Structure:

Divide your system into a component called a *blackboard*, a collection of *knowledge sources*, and a *control component.*

★ **Blackboard:**

- Blackboard is the central data store.
- Elements of the solution space and control data are stored here.
- Set of all data elements that can appear on the blackboard are referred to as vocabulary.
- Blackboard provides an interface that enables all knowledge sources to read form and write to it.
- We use the terms hypothesis or blackboard entry for solutions that are constructed during the problem solving process and put on the blackboard.
- A hypothesis has several attributes, ex: its abstraction level.

Class Blackboard	***Collaborators*** -
Responsibility • Manages central data	

★ **Knowledge source**:

- Knowledge sources are separate, independent subsystem that solve specific aspects of the overall problem.
- Knowledge sources do not communicate directly they only read from and write to the blackboard.
- Often a knowledge source operates on two levels of abstraction.
- Each knowledge source is responsible for knowing the conditions under which it can contribute to a solution. Hence, knowledge sources are split into.

- *Condition part*: evaluates the current state of the solution process, as written on the blackboard, to determine if it can make a contribution.
- *Action part*: produces a result that may causes a change to the blackboard's content.

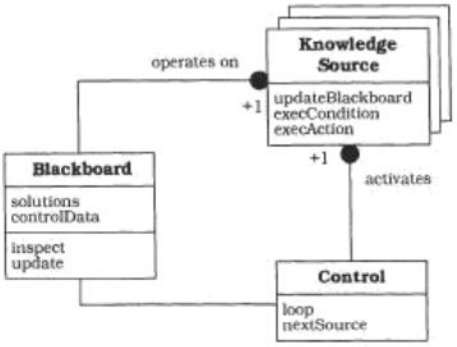
Class	Collaborator
Knowledge Source	• Blackboard
Responsibility • Evaluates its own applicability • Computes a result • Updates Blackboard	

★ **Control component:**

- Runs a loop that monitors the changes on the blackboard and decides what action to take next.
- It schedules knowledge source evaluations and activations according to the knowledge applications strategy. The basis for this strategy is the data on the blackboard.

Class	Collaborators
Control	• Blackboard • Knowledge Source
Responsibility • Monitors Blackboard • Schedules Knowledge Source activations	

- The following figure illustrates the relationship between the three components of the blackboard architecture.

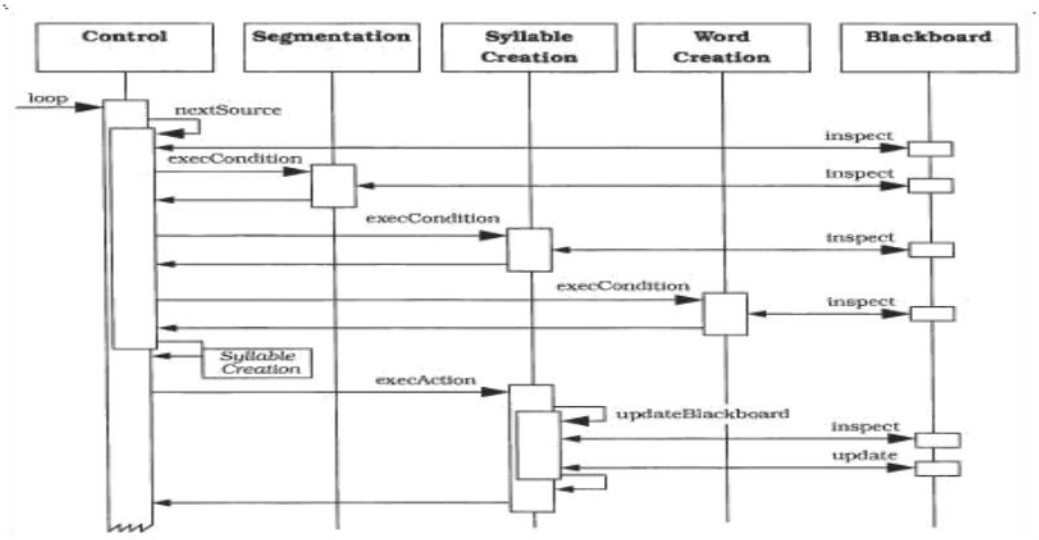


- Knowledge source calls `inspect()` to check the current solutions on the blackboard
- `Update()` is used to make changes to the data on the blackboard
- Control component runs a loop that monitors changes on the blackboard and decides what actions to take next. `nextSource()` is responsible for this decision.

Dynamics:

The following scenario illustrates the behavior of the Blackboard architecture. It is based on our speech recognition example:

- The main loop of the Control component is started.
- Control calls the **`nextsource`**`()` procedure to select the next knowledge source.
- **`nextsource`** `()` first determines which knowledge sources are potential contributors by observing the blackboard.
- **`nextsource()`** invokes the condition-part of each candidate knowledge source.
- The Control component chooses a knowledge source to invoke, and a hypothesis or a set of hypotheses to be worked on.



Implementation:

★ **Define the problem**
- ✓ Specify the domain of the problem
- ✓ Scrutinize the input to the system
- ✓ Define the o/p of the system
- ✓ Detail how the user interacts with the system.

★ **Define the solution space for the problem**
- ✓ Specify exactly what constitutes a top level solution
- ✓ List the different abstraction levels of solutions
- ✓ Organize solutions into one or more abstraction hierarchy.
- ✓ Find subdivisions of complete solutions that can be worked on independently.

★ **Divide the solution process into steps.**
- ✓ Define how solutions are transformed into higher level solutions.
- ✓ Describe how to predict hypothesis at the same abstraction levels.
- ✓ Detail how to verify predicted hypothesis by finding support for them in other levels.
- ✓ Specify the kind of knowledge that can be uses to exclude parts of the solution space.

★ **Divide the knowledge into specialized knowledge**
- ✓ These subtasks often correspond to areas of specialization.

★ **Define the vocabulary of the blackboard**
- ✓ Elaborate your first definition of the solution space and the abstraction levels of your solution.
- ✓ Find a representation for solutions that allows all knowledge sources to read from and contribute to the blackboard.
- ✓ The vocabulary of the blackboard cannot be defined of knowledge sources and the control component.

★ **Specify the control of the system.**
- ✓ Control component implements an opportunistic problem-solving strategy that determines which knowledge sources are allowed to make changes to the blackboard.
- ✓ The aim of this strategy is to construct a hypothesis that is acceptable as a result.
- ✓ The following mechanisms optimizes the evaluation of knowledge sources, and so increase the effectiveness and performance of control strategy.
- ✓ Classifying changes to the blackboard into two types. One type specify all blackboard change that may imply new set of applicable knowledge sources, and the other specifies all blackboard changes that do not.
- ✓ Associating categories of blackboard changes with sets of possibly applicable knowledge sources.
- ✓ Focusing of control. The focus contains either partial results on the blackboard or knowledge sources that should be preferred over others.
- ✓ Creating on queue in which knowledge sources classified as applicable wait for their execution.

★ **Implement the knowledge sources**

- ✓ Split the knowledge sources into condition parts and action-parts according to the needs of the control component.
- ✓ We can implement different knowledge sources in the same system using different technologies

Variants:

- ★ **Production systems:** used in oops language. Here the subroutines are represented as condition-action rules, and data is globally available in working.
- ★ **Repository:** it is a generalization of blackboard pattern the central data structure of this variant is called a repository.

Known uses:

- ★ **HEARSAY-II** → The first Blackboard system was the HEARSAY-II speech recognition system from the early 1970's. It was developed as a natural language interface to a literature database. Its task was to answer queries about documents and to retrieve documents from a collection of abstracts of Artificial Intelligence publications. The inputs to the system were acoustic signals that were semantically interpreted and then transformed to a database query. The control component of HEARSAY-I1 consists of the following:
  - The focus of control database, which contains a table of primitive change types of blackboard changes, and those condition-parts that can be executed for each change type.
  - The scheduling queue, which contains pointers to condition- or action-parts of knowledge source.
  - The monitor, which keeps track of each change made to the blackboard.
  - The scheduler, which uses experimentally-derived heuristics to calculate priorities for the condition- and action- parts waiting in the scheduling queue.
- ★ HASP/SIAP
- ★ CRYSALIS
- ★ TRICERO
- ★ SUS

Example resolved:

- ★ RPOL – runs as a high-priority task & calculates overall rating for the new hypothesis
- ★ PREDICT – works on a phrase and generates predictions of all words
- ★ VERIFY – verify the existence of, or reject, a predicted word
- ★ CONCAT – generates a phrase from verified word & its predicting phrase.

Consequences:

Blackboard approach has the following **Benefits:**

- ★ **Experimentation**
  - A blackboard pattern makes experimentation different algorithms possible.
- ★ **Support for changeability and maintainability**
  - Knowledge sources, control algorithm and central data structure are strictly separated.
- ★ **Reusable knowledge sources**
  - Knowledge source and underlying blackboard system understand the same protocol and data.
  - This means knowledge sources reusable.
- ★ **Support for fault tolerance and robustness**
  - Only those that are strongly supported by data and other hypotheses survive

Blackboard approach has the following **Liabilities:**

- ★ **Difficulty of testing**
  - Because it does not follow a deterministic algorithm
- ★ **No good solution is guaranteed**

- **Difficulty of establishing good control strategy** ○ We require an experimental approach.
- **Low efficiency**
  - Computational overheads in rejecting wrong hypothesis.
- **High development effort**
  - Most blackboard systems take years to evolve.
- **No support for parallelism.**

UNIT 4 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	With neat diagrams, depict the dynamic behaviour of pipes and filters pattern	Dec 09	10
2	What are the benefits of a layered pattern?	Dec 09	4
3	Give the structure of blackboard with CRC cards	Dec 09	6
4	What do you mean by architectural patterns? How is it categorized? Explain the structure part of the solution for ISO layered architecture	June 10/ 14	10
5	Explain with a neat diagram, the dynamic scenario of passive filters	June 10	10
6	List the components of a pipe and filters architectural pattern. With sketches, explain the CRC cards for the same	Dec 10	8
7	Explain the forces that influence the solutions to problems based on blackboard pattern	Dec 10	7
8	Write a note on the HEARSAY – II system	Dec 10	5
9	Discuss the 3-part schema which underlies the layers architectural patterns, with reference to networking protocols	June 11	14
10	Briefly explain the benefits offered by the pipes and filters pattern	June 11	6
11	Discuss the steps involved in the implementation of pipes and filters architecture	Dec 11	12
12	Write the context, problem and solution part of blackboard architectural pattern	Dec 11	8
13	List the components of a pipe and filters architectural pattern. With sketches, explain the CRC cards for the same	June 12	8
14	Define blackboard architectural pattern. Briefly explain the steps to implement the blackboard pattern	June 12/ 14	8
15	Write a note on the HEARSAY – II system	June 12	4