R N S INSTITUTE OF TECHNOLOGY

CHANNASANDRA, BANGALORE - 98

SOFTWARE ARCHITECTURE

NOTES FOR 7TH SEMESTER INFORMATION SCIENCE

SUBJECT CODE: 06IS72

PREPARED BY

DIVYA K

1RN09IS016

7th Semester

Information Science

divya.1rn09is016@gmail.com

SHWETHA SHREE

1RN09IS050

7th Semester

Information Science

shwethashree.1rn09is050@gmail.com

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

CONTENTS:

UNIT 1, UNIT 2, UNIT 3, UNIT 4, UNIT 5, UNIT 6, UNIT 7, UNIT 8

ALONG WITH CHAPTER-WISE PREVIOUS VTU QUESTION PAPERS

UNIT 1
INTRODUCTION

CHAPTER 1 :

THE ARCHITECTURE BUSINESS CYCLE

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

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

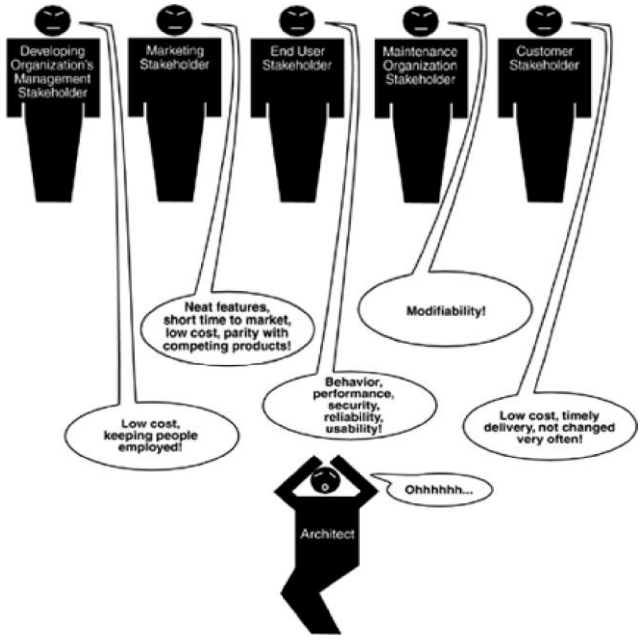
- ✓ How organizational goals influence requirements and development strategy.
- ✓ How requirements lead to architecture.
- ✓ How architectures are analyzed.
- ✓ How architectures yield systems that suggest new organizational capabilities and requirements.

1.1 WHERE DO ARCHITECTURES COME FROM?

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

❖ ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS

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



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

❖ ARCHITECTURES ARE INFLUENCED BY THE DEVELOPING ORGANIZATIONS.

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

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

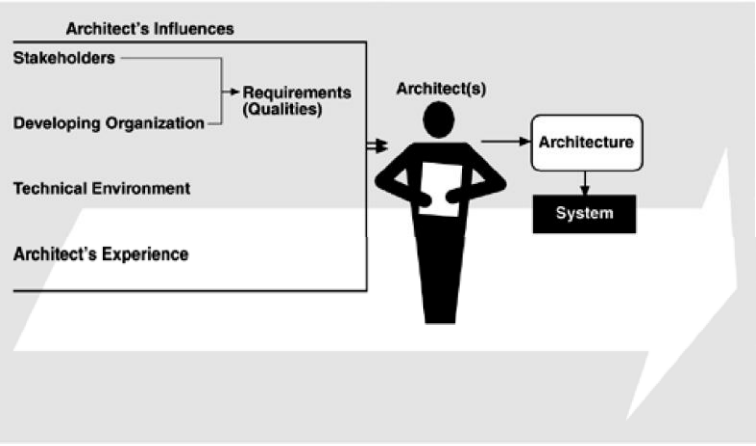
- ✓ If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort.
- ✓ Conversely, if their prior experience with this approach was disastrous,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

❖ ARCHITECTURES ARE INFLUENCED BY THE TECHNICAL ENVIRONMENT

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

❖ RAMIFICATIONS OF INFLUENCES ON AN ARCHITECTURE

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

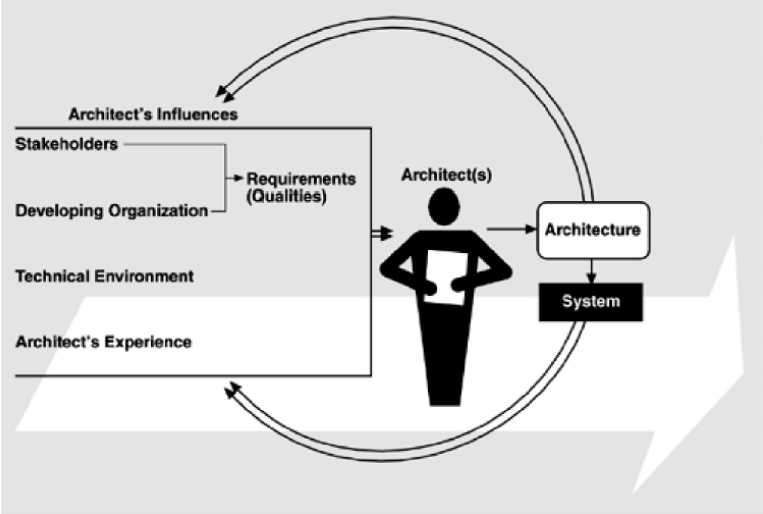


- ✓ Influences on an architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict.
- ✓ Architects need to know and understand the nature, source, and priority of constraints on the project as early as possible.
- ✓ Therefore, *they must identify and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

❖ THE ARCHITECTURE AFFECTS THE FACTORS THAT INFLUENCE THEM

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

Figure 1.4. The Architecture Business Cycle



Working of architecture business cycle:

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

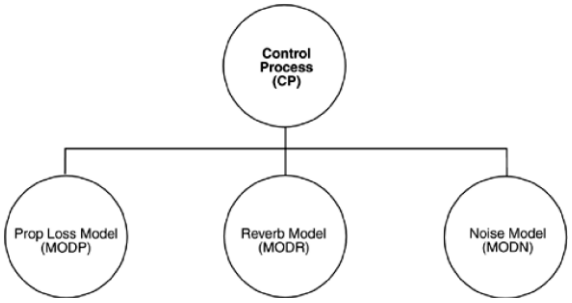


Figure 2.1 : Typical, but uninformative, presentation of a software architecture

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

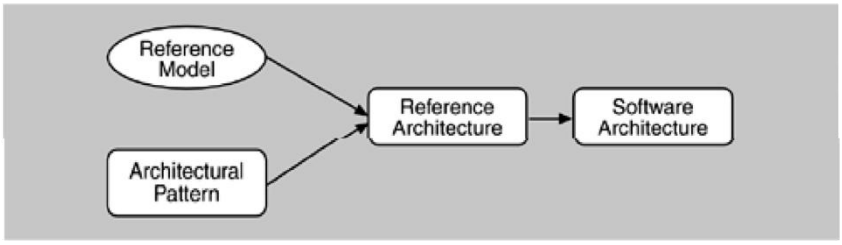
An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.

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**

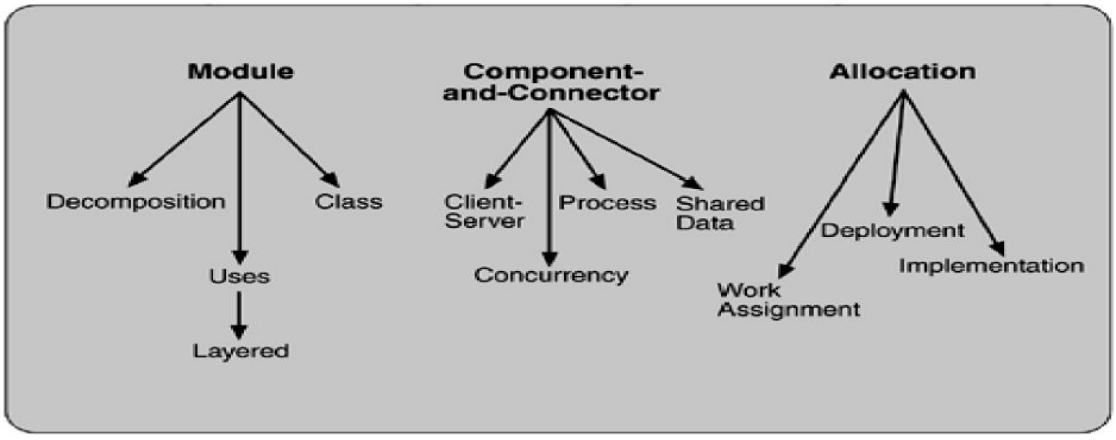
The architecture, including a description of how elements interact to carry out the required behavior, can serve as the introduction to the system for new project members.

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	5
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	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.

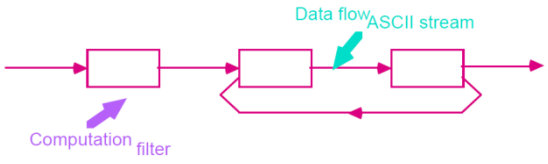


Figure 1: Pipes and Filters

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.

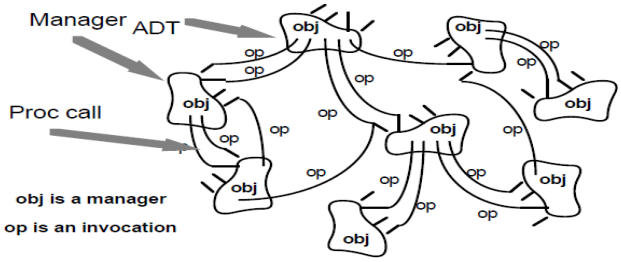


Figure 2: Abstract Data Types and Objects

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.

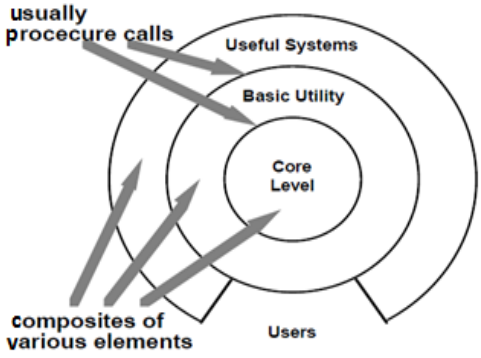


Figure 3: Layered Systems

Examples:

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

Advantages:

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

Disadvantages:

- o 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

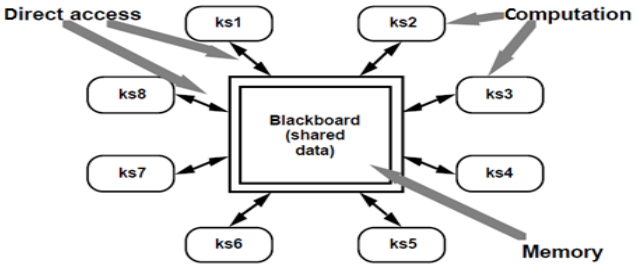


Figure 4: The Blackboard

- ✓ 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"

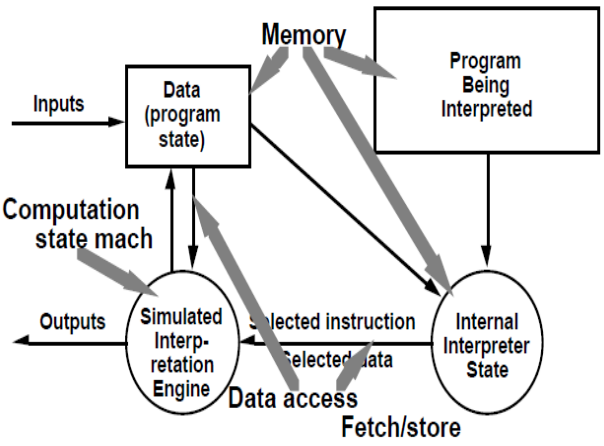


Figure 5: Interpreter

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.*

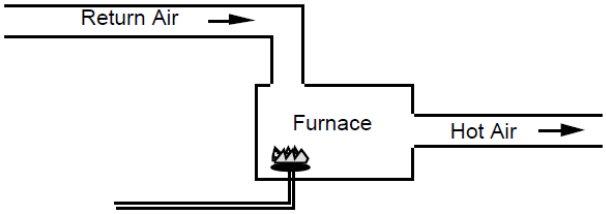


Figure 2.8 open-loop temperature control

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.

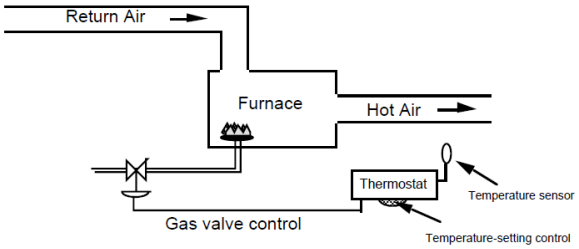


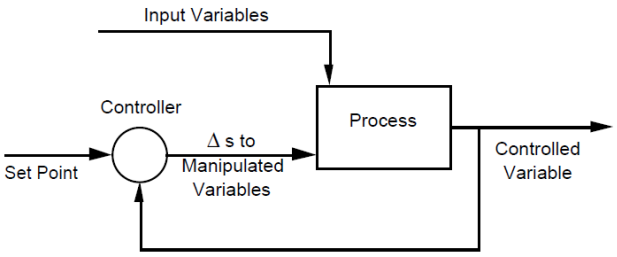
Figure 2.9 closed-loop temperature control

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.

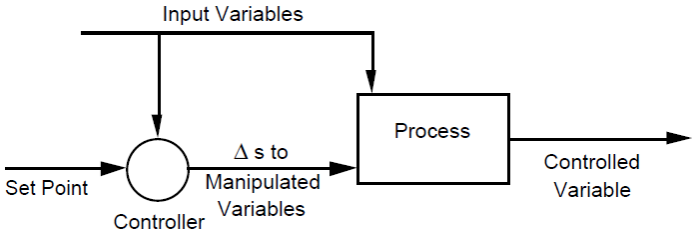


Figure 2.11 feedforward control

- 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.

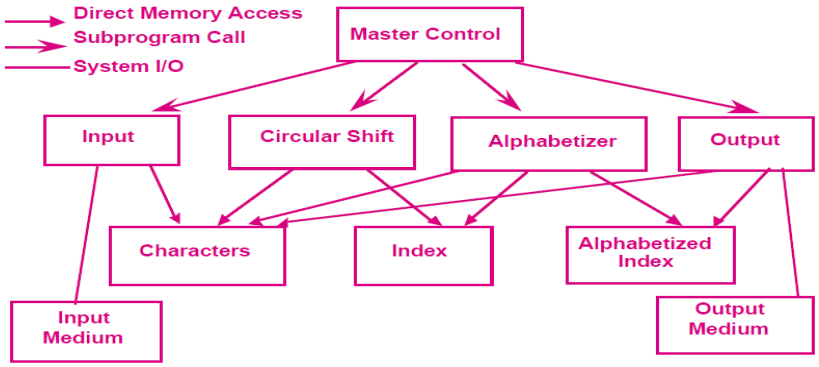


Figure 6: KWIC – Shared Data Solution

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.

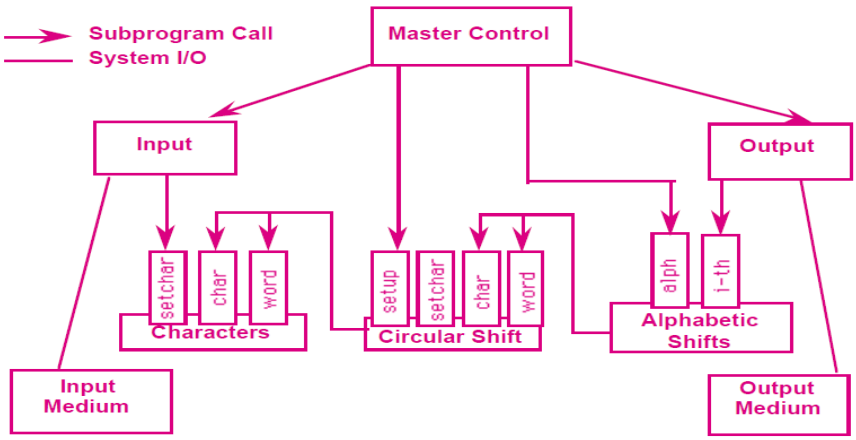


Figure 7: KWIC – Abstract Data Type Solution

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.

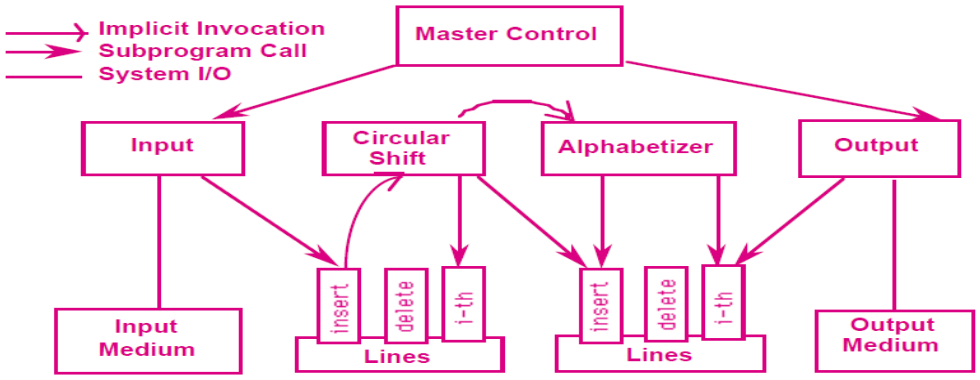


Figure 8: KWIC – Implicit Invocation Solution

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.

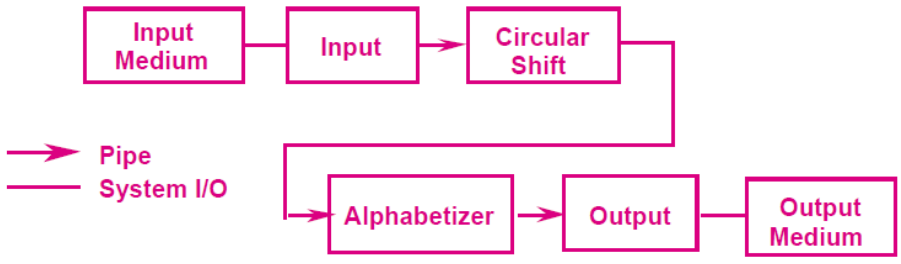


Figure 9: KWIC – Pipe and Filter Solution

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.

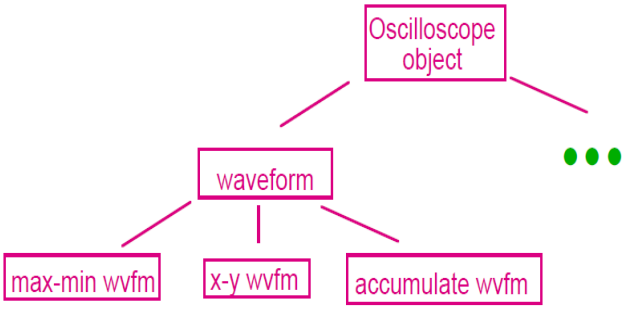


Figure 11: Oscilloscopes – An Object-oriented Model

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.

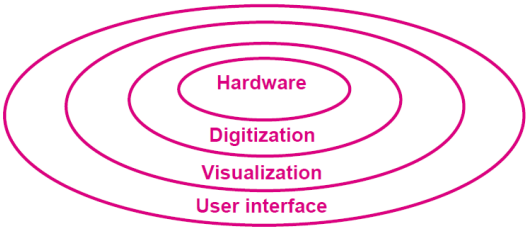


Figure 12: Oscilloscopes – A Layered Model

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.

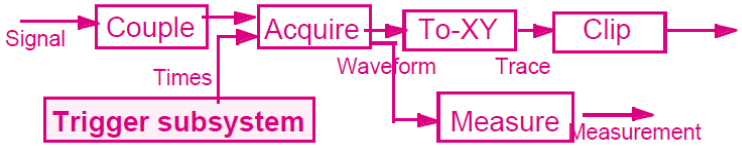


Figure 13: Oscilloscopes – A Pipe and Filter Model

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.

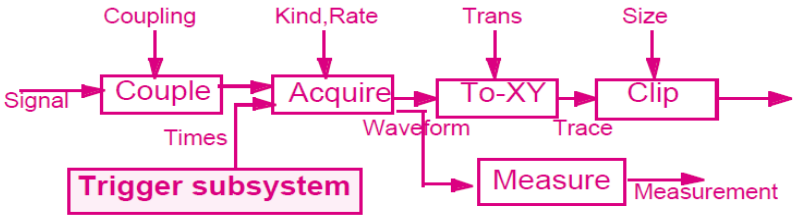


Figure 14: Oscilloscopes – A Modified Pipe and Filter Model

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

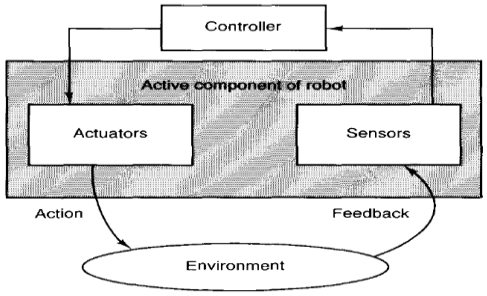
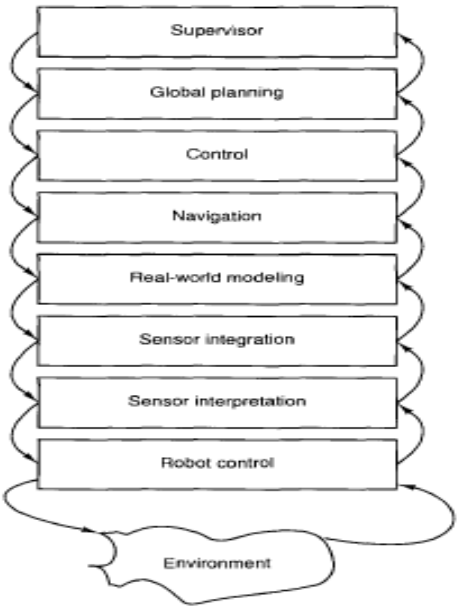


FIGURE 3.10 A Control-Loop Solution for Mobile Robots

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

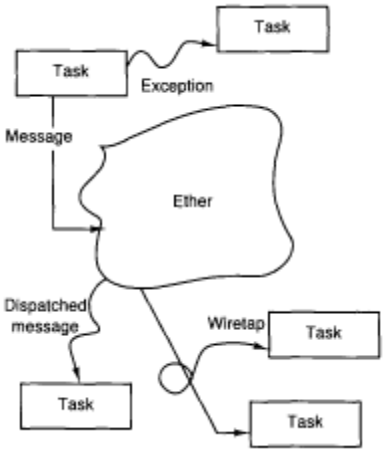


FIGURE 3.12 An Implicit Invocation Architecture for Mobile Robots

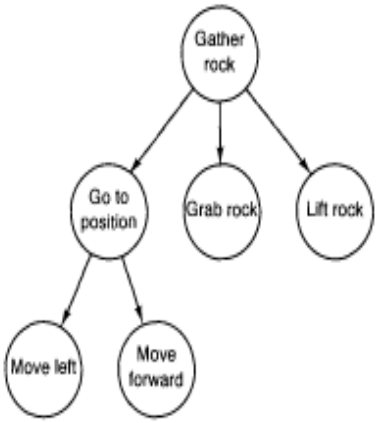


FIGURE 3.13 A Task Tree for Mobile Robo

- ★ **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

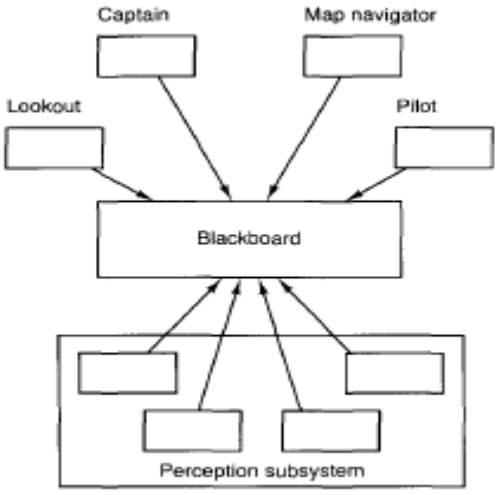


Figure: blackboard solution for mobile robots

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	+–	–	+	+

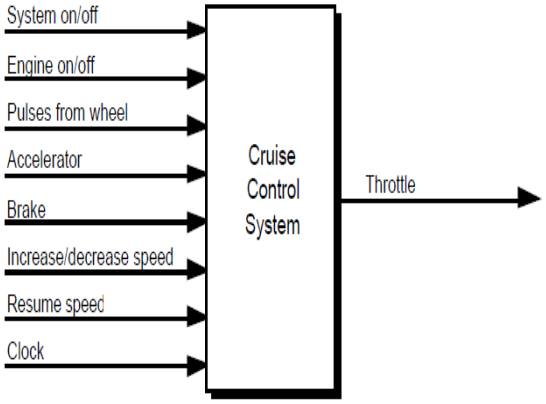
Table 2.2.1. Strengths and Weaknesses of Robot Architectures

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



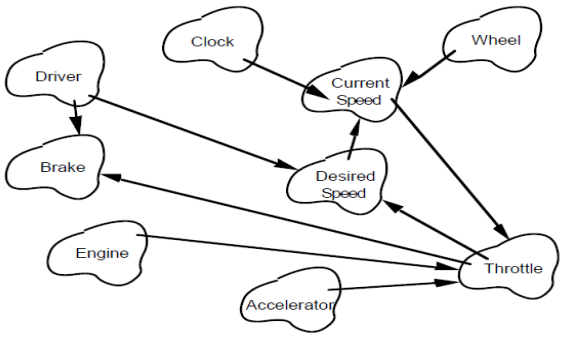
Outputs:

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

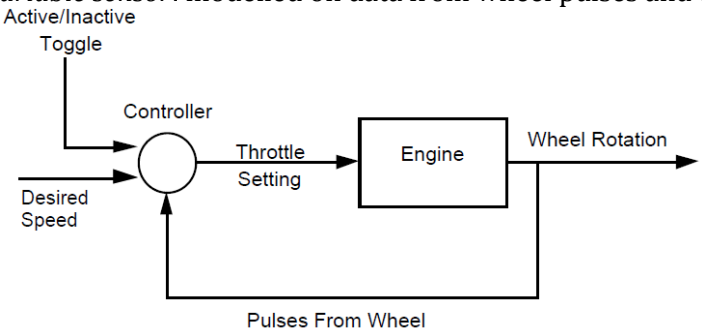


Figure 3.18 control architecture for cruise control

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.

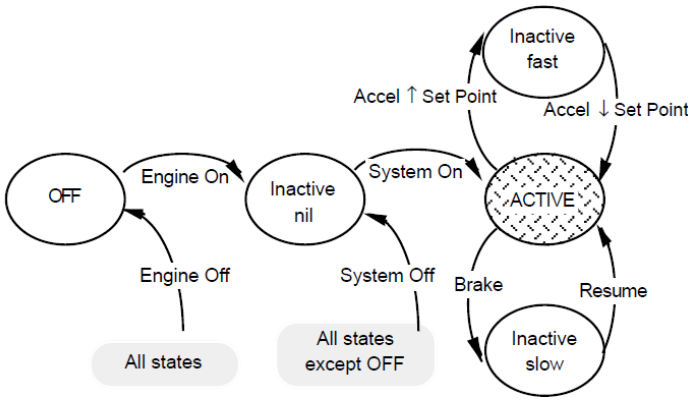


Figure 3.19 state machine for activation

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.

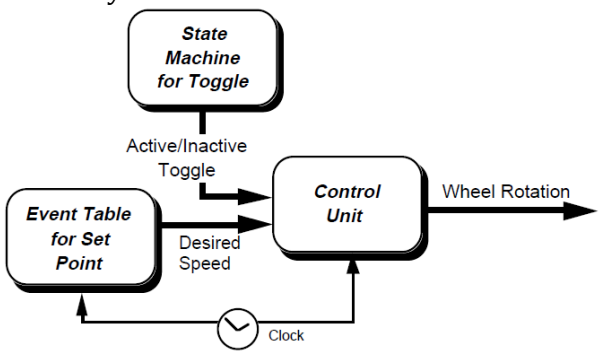


Figure 3.21 complete cruise control system

ANALYSIS AND DISCUSSION

[Interested students can refer text book since this has not been asked in exam till date]

THREE VIGNETTES IN MIXED STYLE

A LAYERED DESIGN WITH DIFFERENT STYLES FOR THE LAYERS

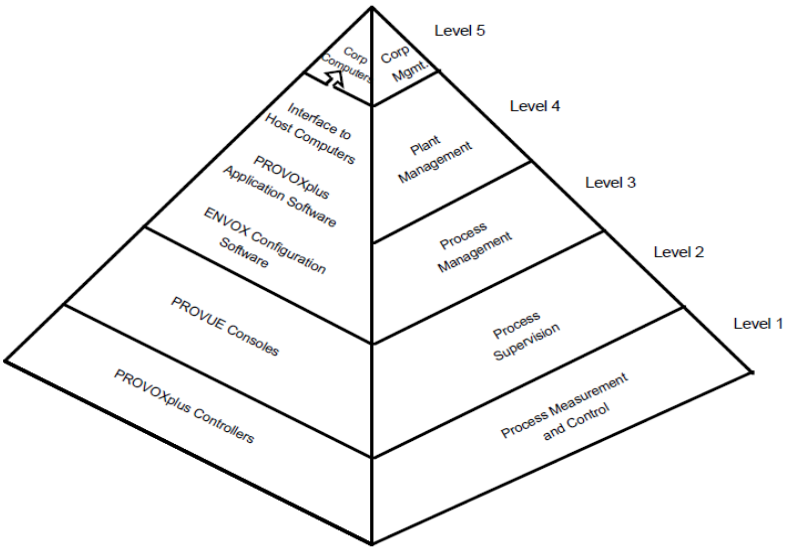


Figure 19: PROVOX – Hierarchical Top Level

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.

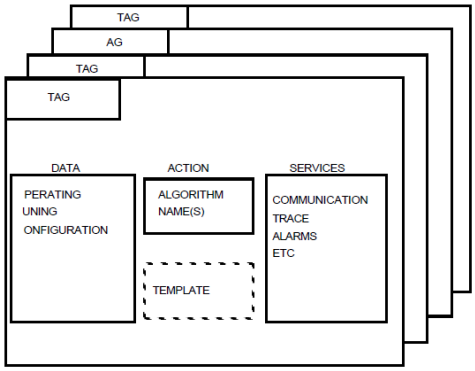


Figure 20: PROVOX – Object-oriented Elaboration

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.

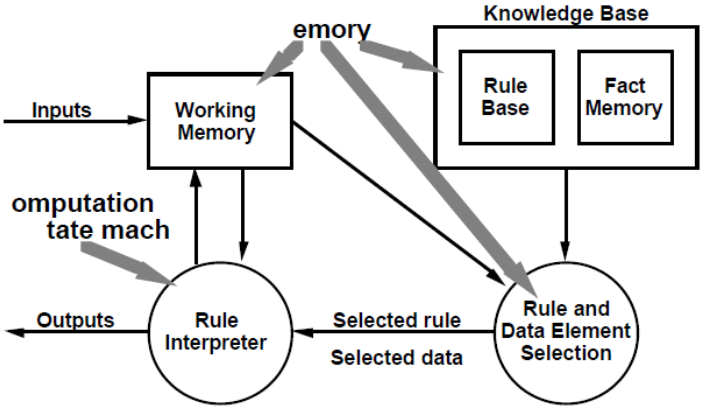


Figure 21: Basic Rule-Based System

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.

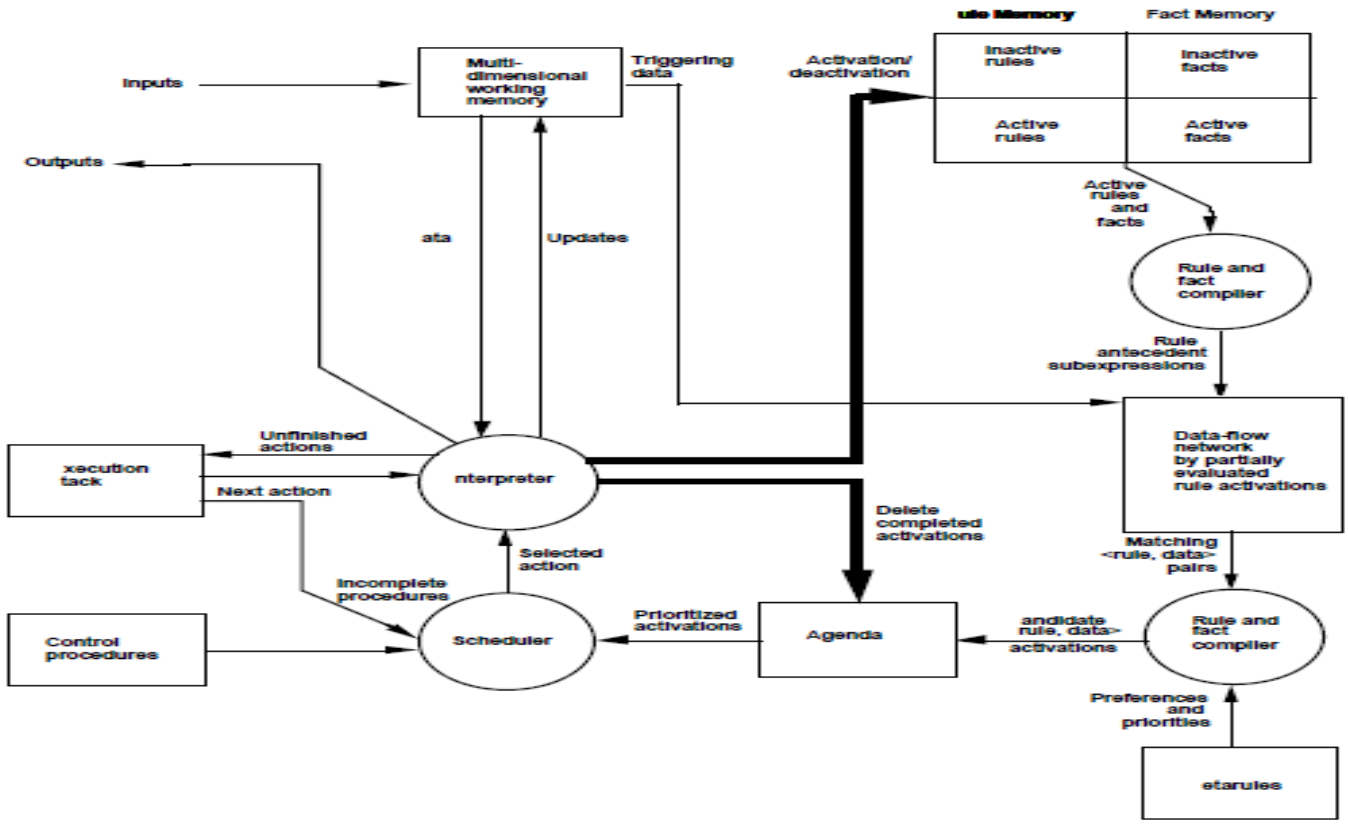


Figure 23: Sophisticated Rule-Based System

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.

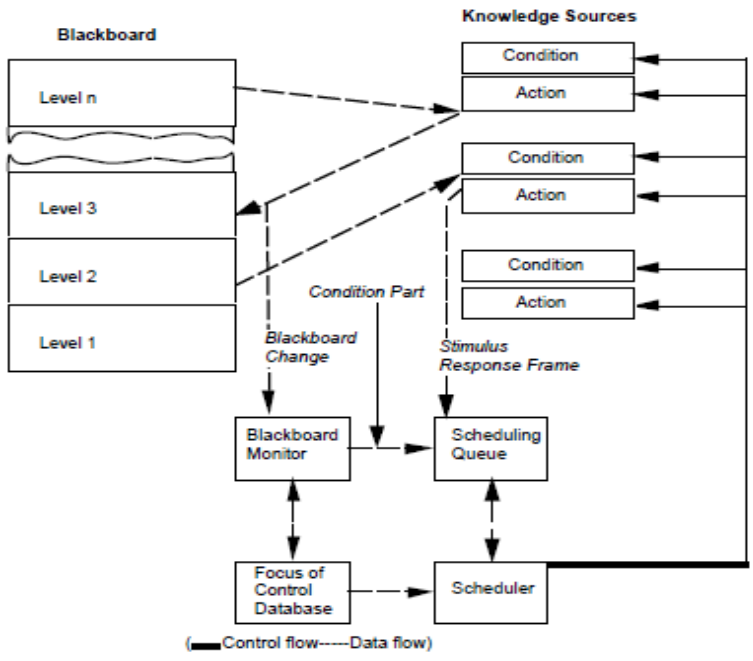


Figure 24: Hearsay-II

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.

Interested students can refer text book for blackboard and interpreter view of HEARSAY II which is similar to above figure

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 a 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	Dec 10	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	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.

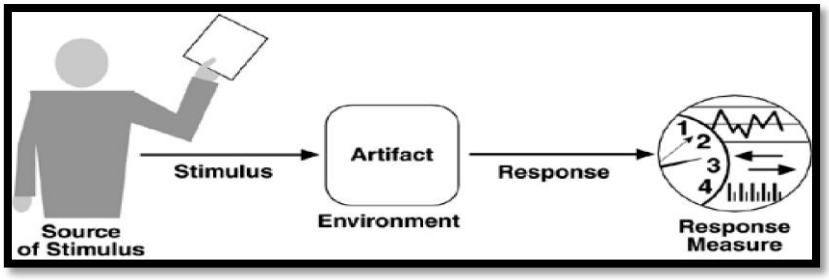
4.3 SYSTEM QUALITY ATTRIBUTES

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

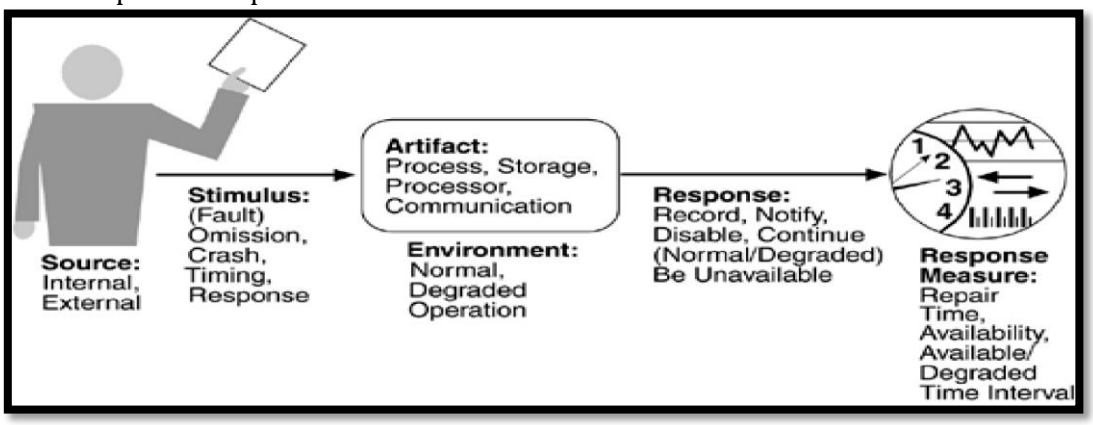
AVAILABILITY SCENARIO

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.

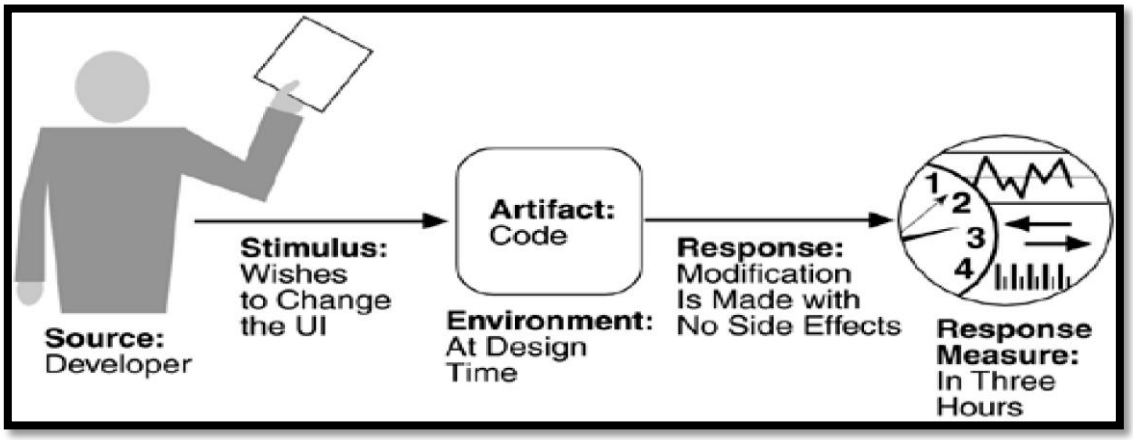


Artifact. This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.
Environment. The state of the system when the fault or failure occurs may also affect the desired system response. For example, if the system has already seen 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.

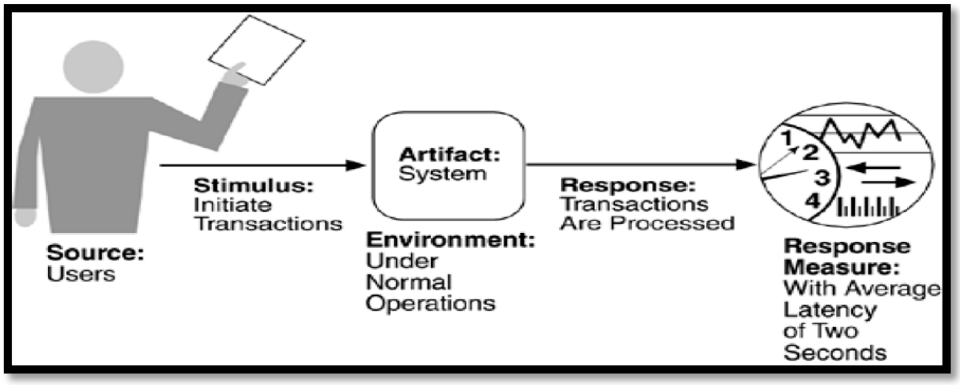
Environment. This portion specifies when the change can be made—design time, compile time, build time, initiation time, or runtime. In our example, the modification is to occur at design time.

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.

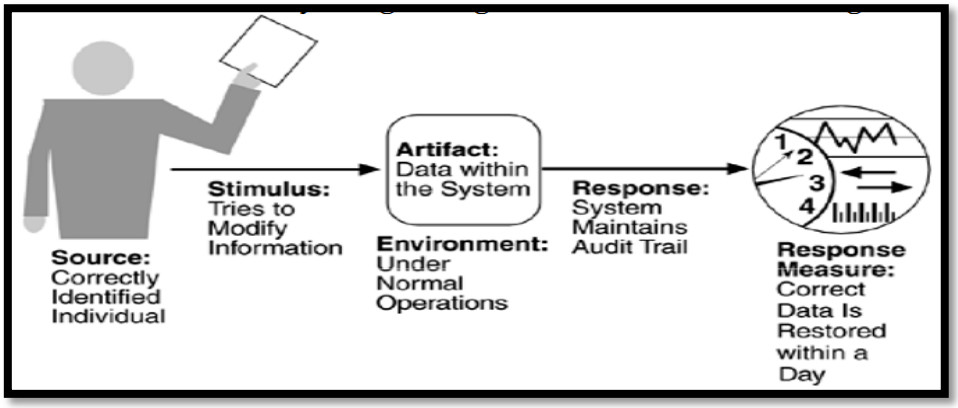
Environment. The system can be in various operational modes, such as normal, emergency, or overload. In our example, the system is in normal mode.
Response. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode). In our example, 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.

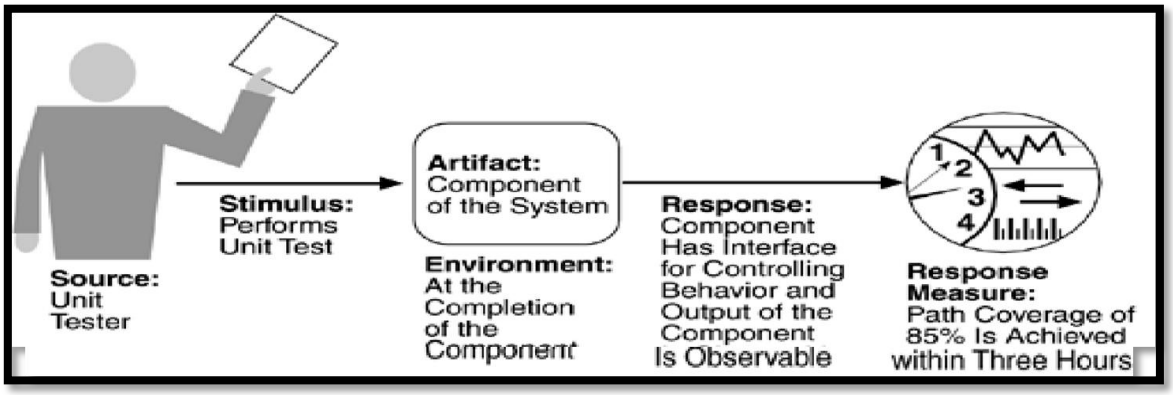


Source of stimulus. The source of the attack may be either a human or another system. It may have been previously identified (either correctly or incorrectly) or may be currently unknown.
Stimulus. The stimulus is an attack or an attempt to break security. We characterize this as an unauthorized person 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.

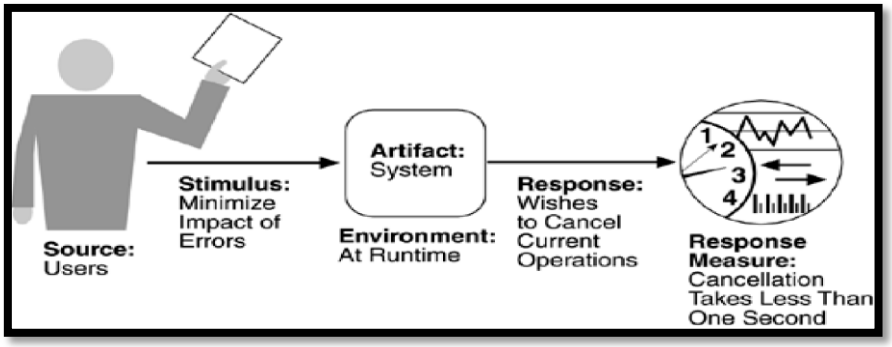
USABILITY SCENARIO

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

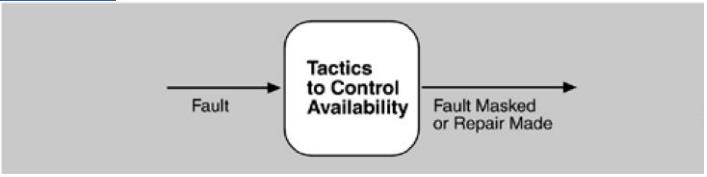
A **tactic** is a design decision that influences the control of a quality attribute response.

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.

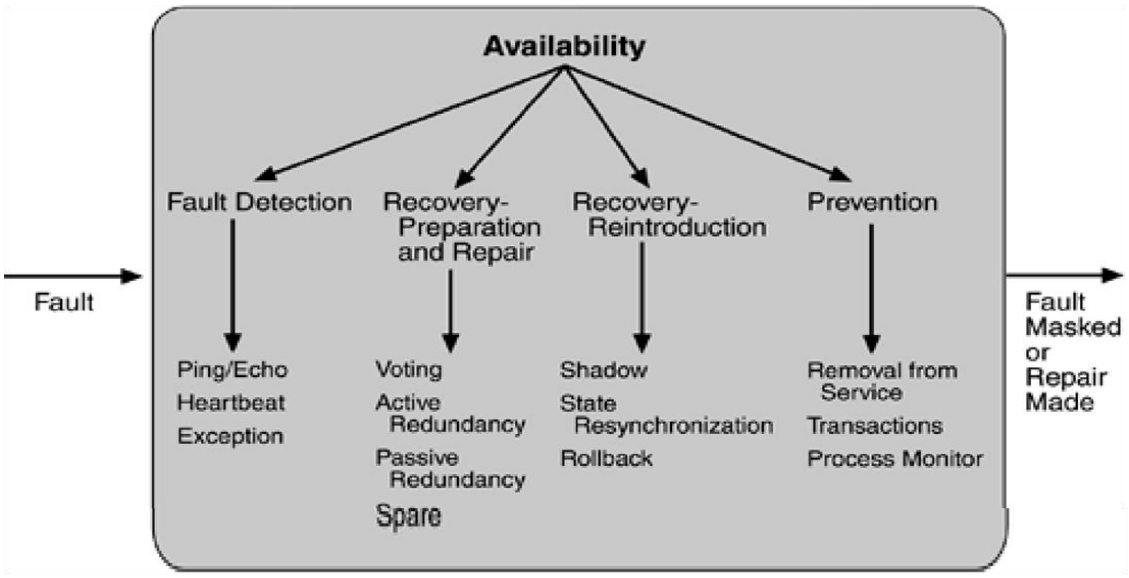
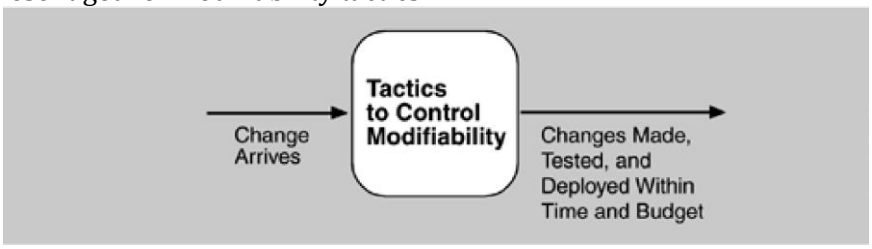


Figure 5.3. Summary of availability tactics

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.

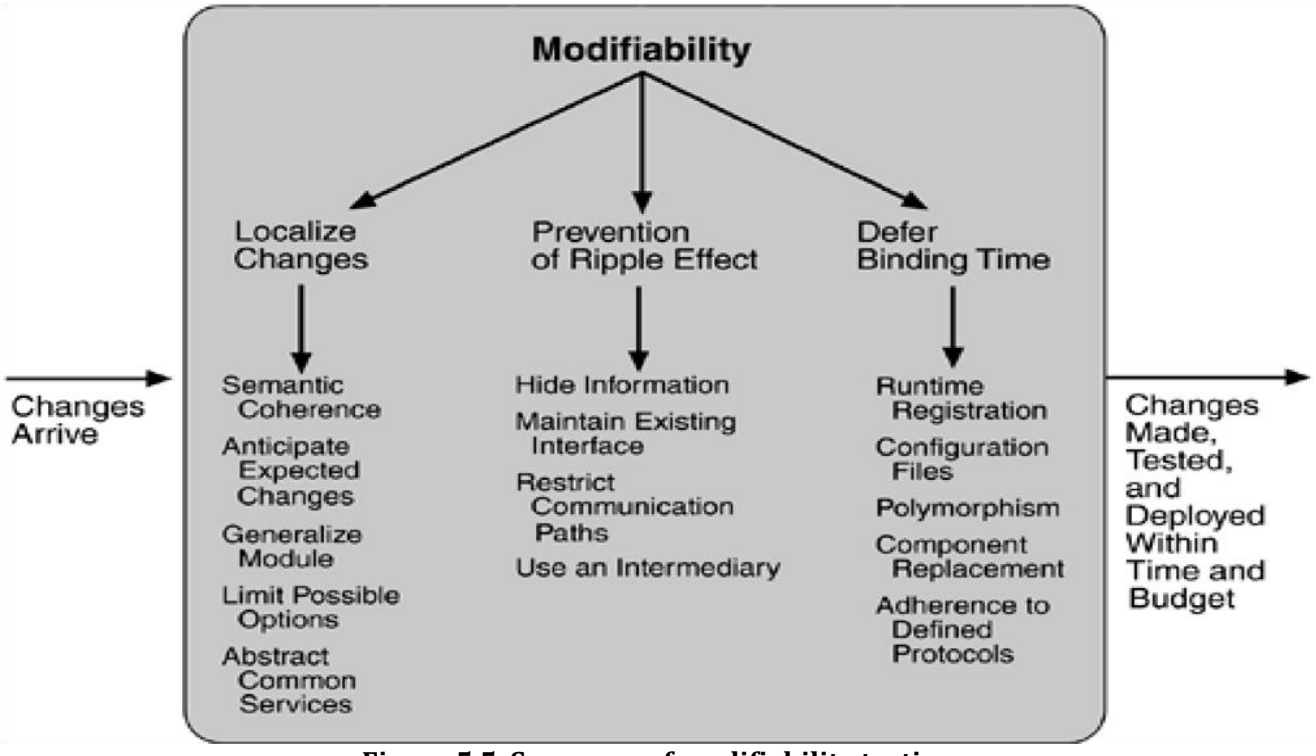
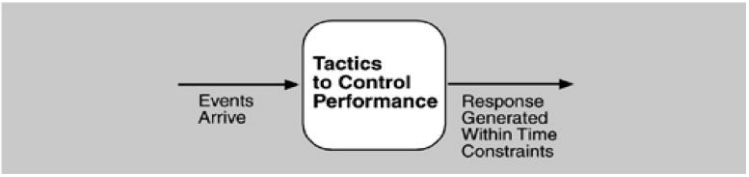


Figure 5.5. Summary of modifiability tactics

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.

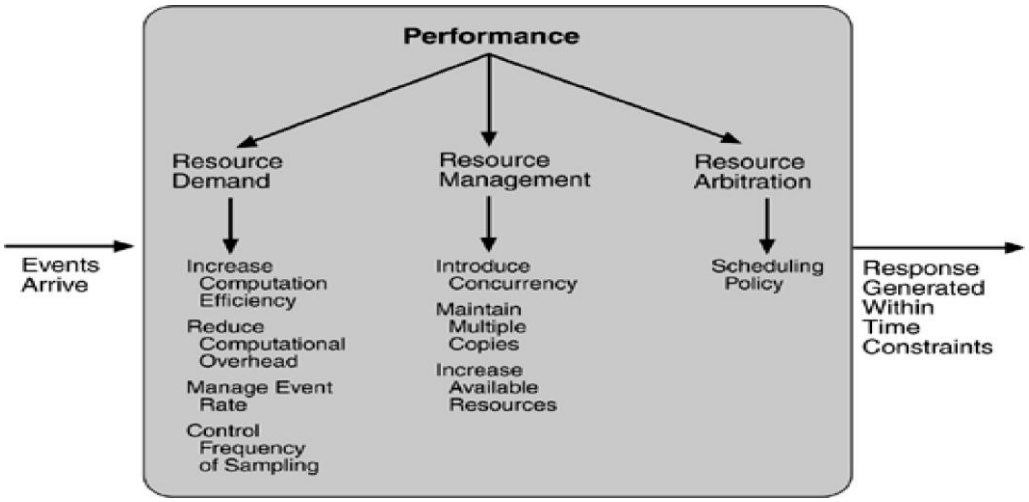
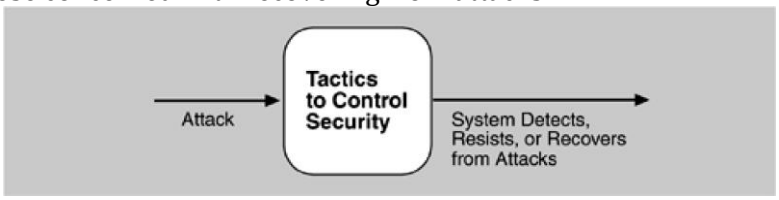


Figure 5.7. Summary of performance tactics

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.

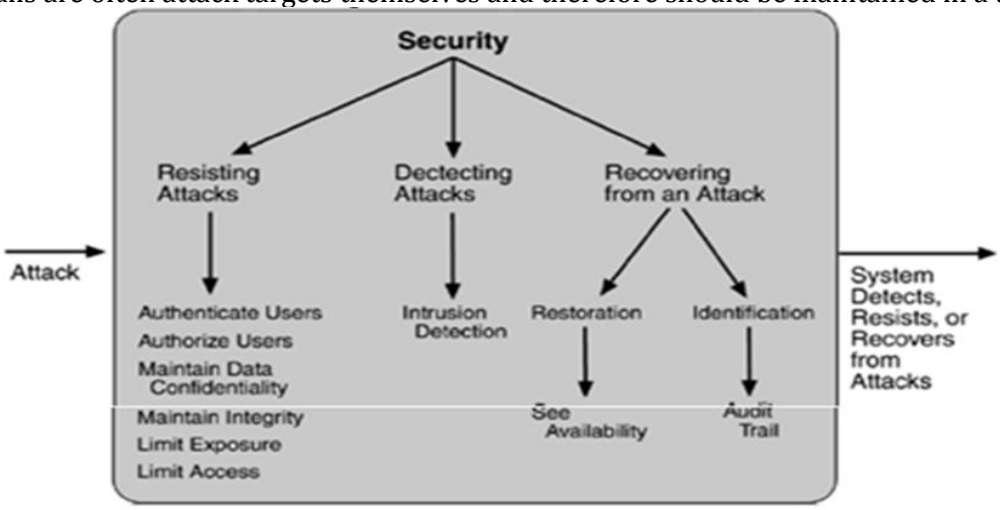
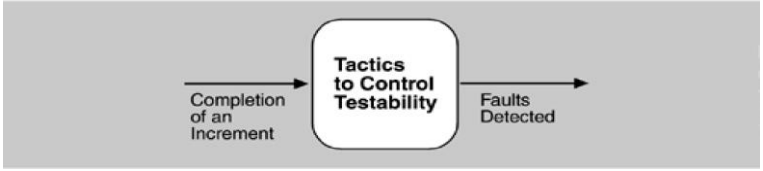


Figure 5.9. Summary of tactics for security

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.

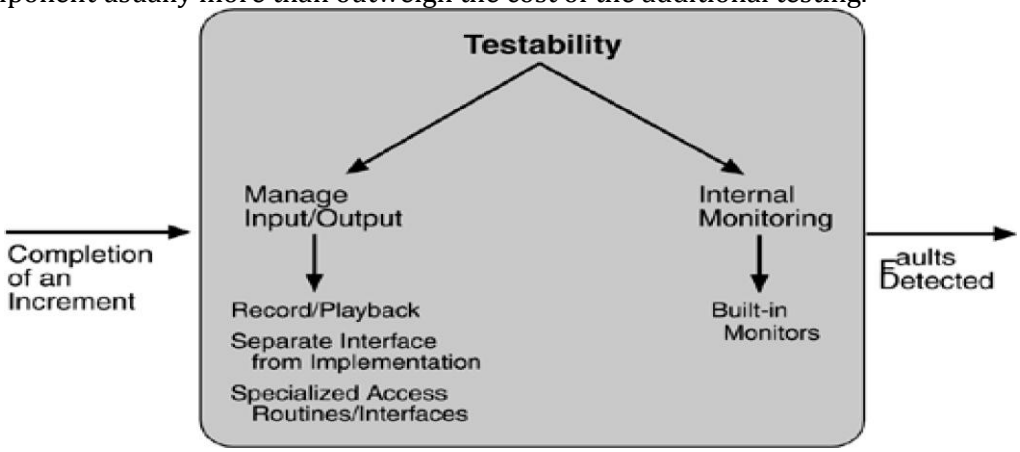
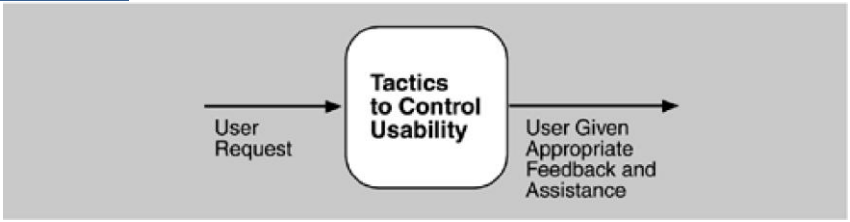


Figure 5.11. Summary of testability tactics

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

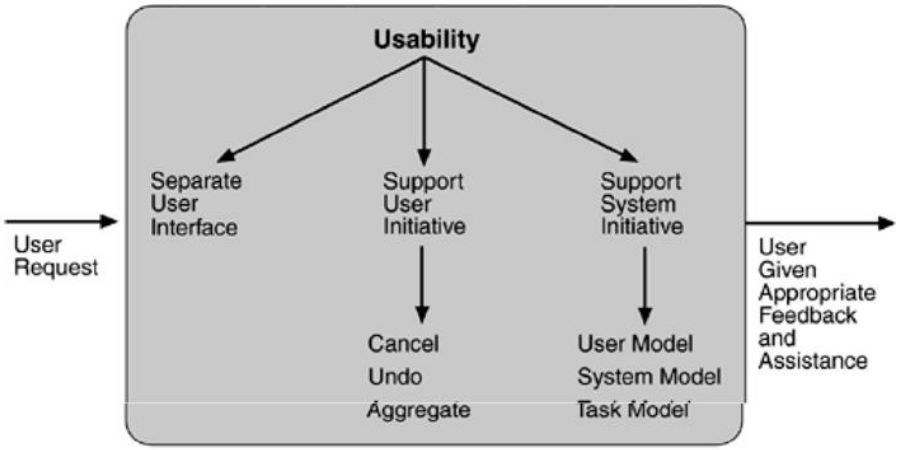


Figure 5.13. Summary of runtime usability tactics

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.

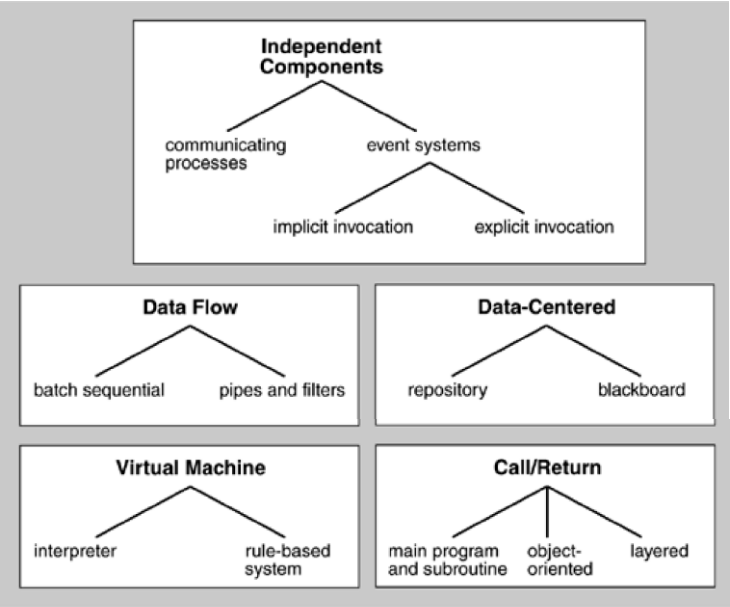


Figure 5.14. A small catalog of architectural patterns, organized by is-a relations

UNIT 3 - QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	What is availability? Explain the general scenario for availability.	Dec 09	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	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.

[If mud to structure question is asked for 10 marks in exam, summary of full unit about layers, pipes & filters and blackboard pattern should be written]

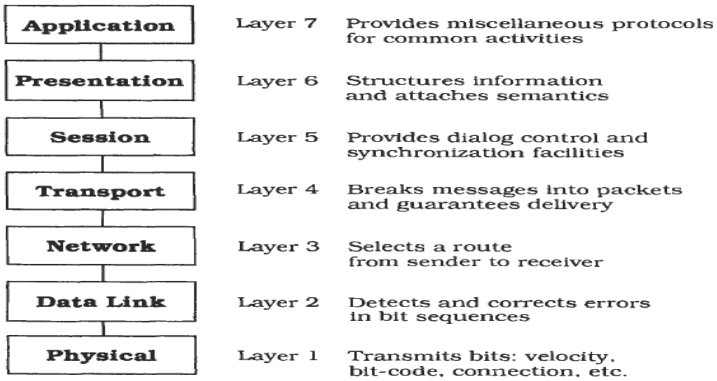
LAYERS

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:

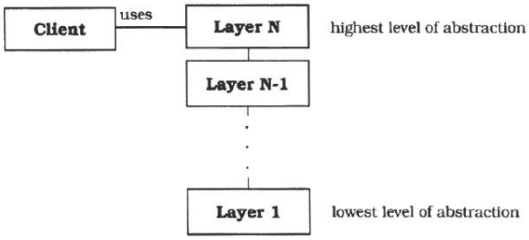
- Structure your system into an appropriate number of layers and place them on top of each other.
- Lowest layer is called 1 (base of our system), the highest is called layer N. i.e, Layer 1, ……. Layer J-1, Layer J, ….. Layer N.
- Most of the services that layer J Provides are composed of services provided 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:

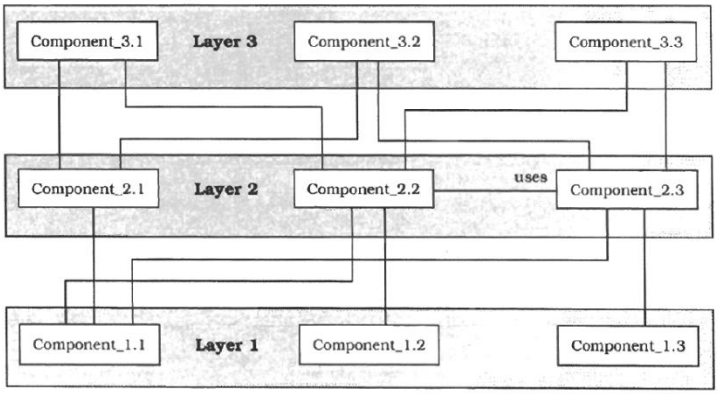
- An individual layer can be described by the following CRC card:

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.

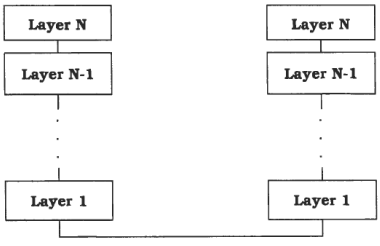


- In more detail, it might look something like this:



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.

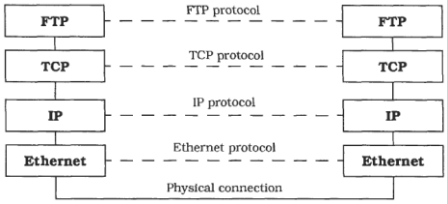
- ★ **Define the abstraction criterion for grouping tasks into layers.**
  - This criterion is often the conceptual distance from the platform (sometimes, we encounter other abstraction paradigm as well).
  - In the real world of software development we often use a mix of abstraction criterions. For 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.

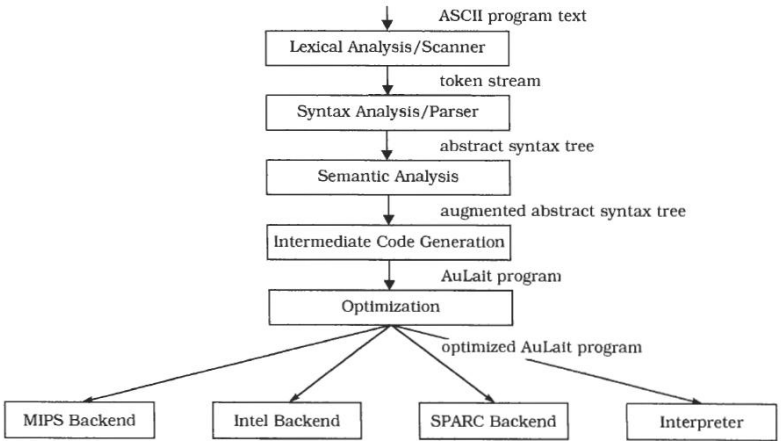
PIPES AND FILTERS

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:**
  - o Filter components are the processing units of the pipeline.
  - o 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.
  - o 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.	

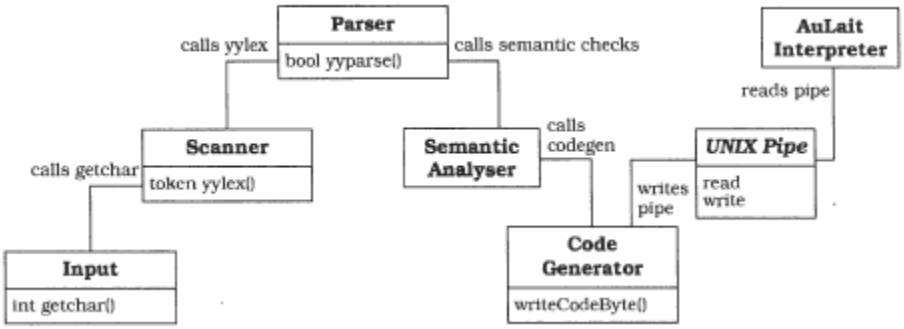
★ **Data source component:**
- The data source represents the input to the system, and provides a sequence of data values of the same structure or type.
- It is responsible for delivering input to the processing pipeline.

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.	

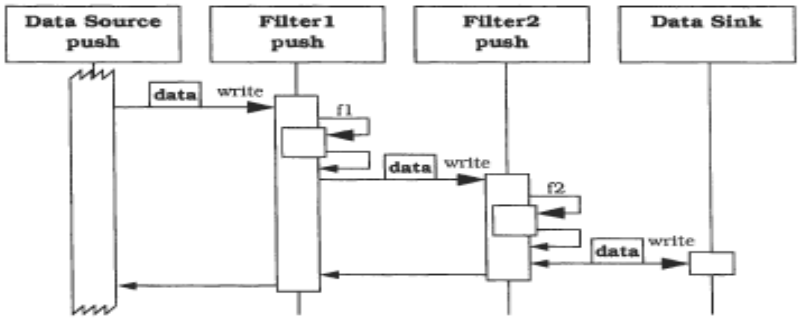
In our Mocha compiler we use the UNIX tools lex and yacc to implement the first two stages of the compiler. The connection to the other frontend stages consists of many procedure calls embedded in the grammar action rules, and not just simple data flow. The backends and interpreter run as separate programs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



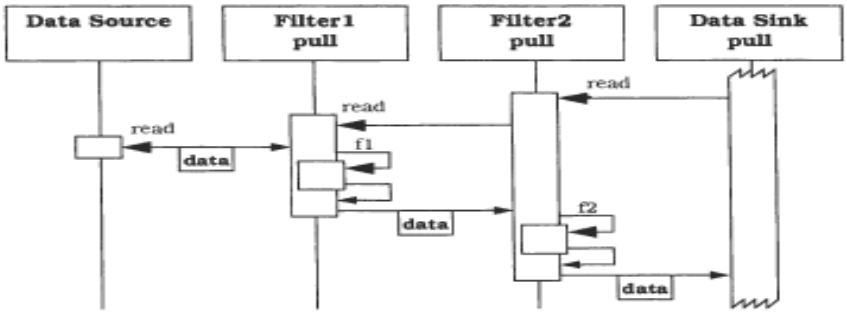
Dynamics:

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

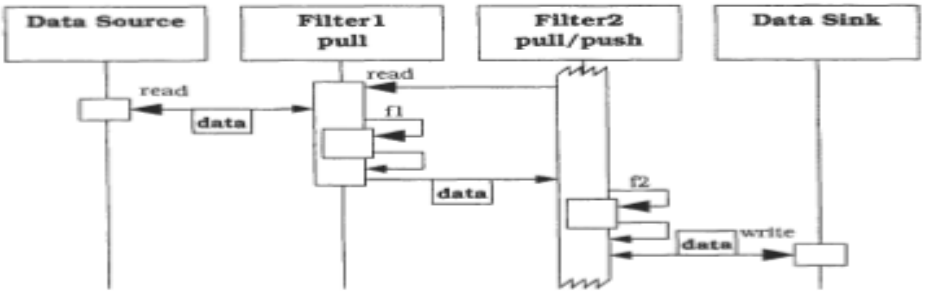
- ★ **Scenario 1** shows a push pipeline in which activity starts with the data source.



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

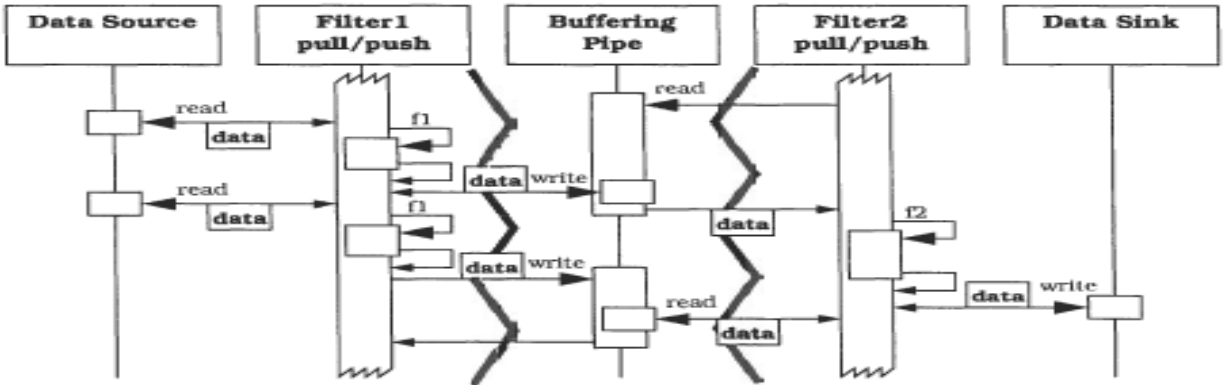


- ★ **Scenario 3** shows a mixed push-pull pipeline with passive data source and sink. Here second filter plays the active role and starts the processing.



- ★ **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.



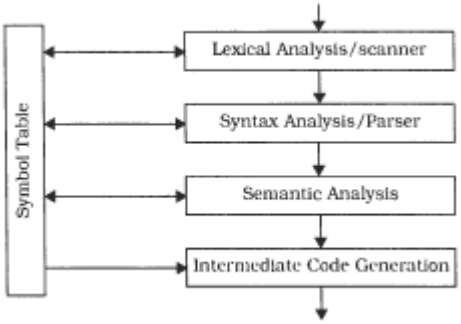
Implementation:

- ★ **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

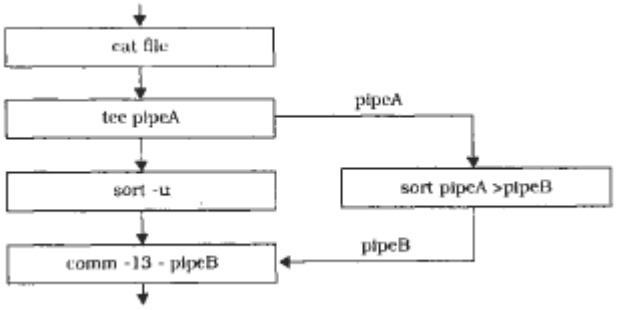
single program because they all access and modify the symbol table. This allows us to implement the pipe connection between the scanner and the parser as a simple function call.



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**

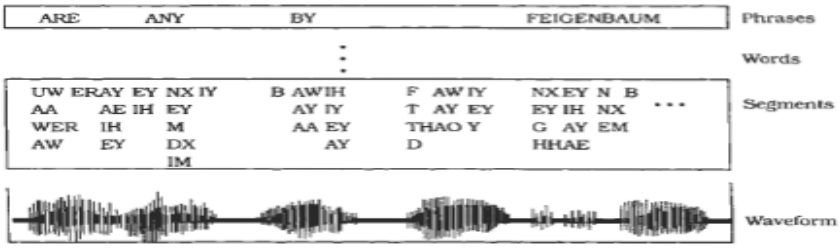
- ★ **Sharing state information is expensive or inflexible**
  - This pattern is inefficient if your processing stage needs to share a large amount of global data.
- ★ **Efficiency gain by parallel processing is often an illusion**, because:
  - The cost for transferring data between filters is relatively 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:**
  - o Blackboard is the central data store.
  - o Elements of the solution space and control data are stored here.
  - o Set of all data elements that can appear on the blackboard are referred to as vocabulary.
  - o Blackboard provides an interface that enables all knowledge sources to read form and write to it.
  - o We use the terms hypothesis or blackboard entry for solutions that are constructed during the problem solving process and put on the blackboard.
  - o A hypothesis has several attributes, ex: its abstraction level.

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

- ★ **Knowledge source**:
  - o Knowledge sources are separate, independent subsystem that solve specific aspects of the overall problem.
  - o Knowledge sources do not communicate directly they only read from and write to the blackboard.
  - o Often a knowledge source operates on two levels of abstraction.
  - o 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.

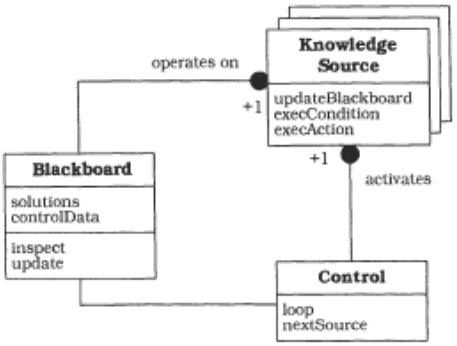
Class Knowledge Source	Collaborator • Blackboard
Responsibility • Evaluates its own applicability • Computes a result • Updates Blackboard	

★ **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 Control	Collaborators • Blackboard • Knowledge Source
Responsibility • Monitors Blackboard • Schedules Knowledge Source activations	

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

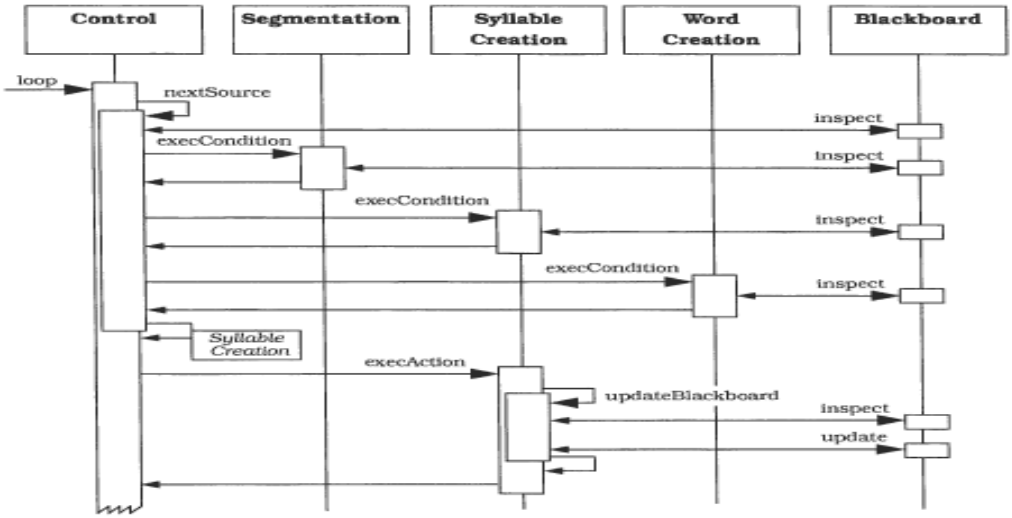


- Knowledge source calls `inspect()` to check the current solutions on the blackboard
- `Update()` is used to make changes to the data on the blackboard
- Control component runs a loop that monitors changes on the blackboard and decides what actions to take next. `nextSource()` is responsible for this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

[Refer text if you need more details]

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.

[Refer text if you need more details]

Consequences:

Blackboard approach has the following **Benefits:**

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

Blackboard approach has the following **Liabilities:**

- ★ **Difficulty of testing**
  - o 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	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	8
15	Write a note on the HEARSAY – II system	June 12	4

UNIT 5
ARCHITECTURAL PATTERNS-2

DISTRIBUTED SYSTEMS

What are the advantages of distributed systems that make them so interesting?
Distributed systems allow better sharing and utilization of the resources available within the network.

- Economics: computer network that incorporates both pc's and workstations offer a better price/performance ratio than mainframe computer.
- Performance and scalability: a huge increase in performance can be gained by using the combine computing power of several network nodes.
- Inherent distribution: some applications are inherently distributed. Ex: database applications that follow a client-server model.
- Reliability: A machine on a network in a multiprocessor system can crash without affecting the rest of the system.

Disadvantages
They need radically different software than do centralized systems.

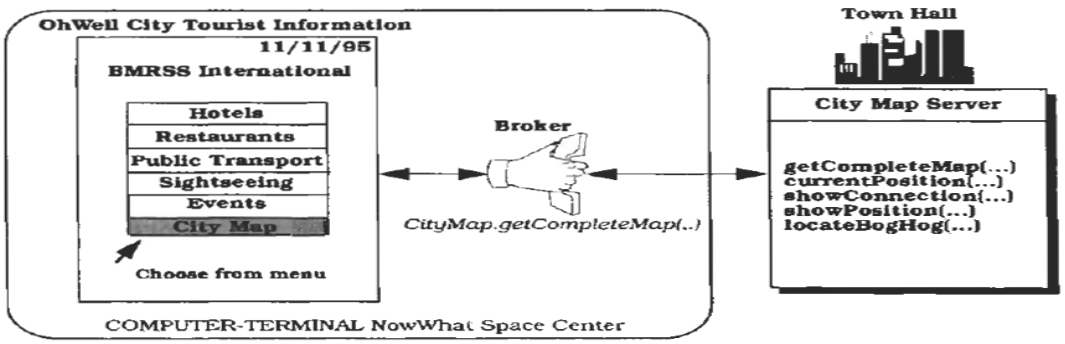
We introduce three patterns related to distributed systems in this category:

- 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.
- The **Microkernel pattern** applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts
- The **Broker pattern** can be used to structure distributed software systems with decoupled components that interact by remote service invocations.

BROKER

The broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as requests, as well as for transmitting results and exceptions.

Example:



Suppose we are developing a city information system (CIS) designed to run on a wide area network. Some computers in the network host one or more services that maintain information about events, restaurants, hotels, historical monuments or public transportation. Computer terminals are connected to the network. Tourists throughout the city can retrieve information in which they are interested from the terminals using a

World Wide Web (WWW) browser. This front-end software supports the on-line retrieval of information from the appropriate servers and its display on the screen. The data is distributed across the network, and is not all maintained in the terminals.

Context:
Your environment is a distributed and possibly heterogeneous system with independent co operating components.

Problem:
Building a complex software system as a set of decoupled and interoperating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentially distributable and scalable. Services for adding, removing, exchanging, activating and locating components are also needed. From a developer's viewpoint, there should essentially be no difference between developing software for centralized systems and developing for distributed ones.
We have to balance the following forces:

- Components should be able to access services provided by other through remote, location-transparent service invocations.
- You need to exchange, add or remove components at run time.
- The architecture should hide system and implementation-specific details from the users of component and services.

Solution:

- Introduce a broker component to achieve better decoupling of clients and servers.
- Servers registers themselves with the broker make their services available to clients through method interfaces.
- Clients access the functionality of servers by sending requests via the broker.
- A broker's tasks include locating the appropriate server, forwarding the request to the server, and transmitting results and exceptions back to the client.
- The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer.

Structure:
The broker architectural pattern comprises six types of participating components.

★ **Server:**
- Implements objects that expose their functionality through interfaces that consists of operations and attributes.
- These interfaces are made available either through an interface definition language (IDL) or through a binary standard.
- There are two kind of servers:
  - ♠ Servers offering common services to many application domains.
  - ♠ Servers implementing specific functionality for a single application domain or task.

★ **Client:**
- Clients are applications that access the services of at least one server.
- To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.
- Interaction b/w servers and clients is based on a dynamic model, which means that servers may also act as clients.

Class: Client	Collaborators
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

Class: Server	Collaborators
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	• Server-side Proxy • Broker

★ **Brokers:**

- It is a messenger that is responsible for transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client.
- It offers API'S to clients and servers that include operations for registering servers and for invoking server methods.
- When a request arrives from server that is maintained from local broker, the broker passes the request directly to the server. If the server is currently inactive, the broker activates it.
- If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request this route.
- Therefore there is a need for brokers to interoperate through bridges.

Class: Broker	Collaborators
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge

★ **Client side proxy:**

- They represent a layer b/w client and the broker.
- The proxies allow the hiding of implementation details from the clients such as
- The inter process communication mechanism used for message transfers b/w clients and brokers.
- The creation and deletion of blocks.
- The marshalling of parameters and results.

★ **Server side proxy:**

- Analogous to client side proxy. The difference that they are responsible for receiving requests, unpacking incoming messages, un marshalling the parameters, and calling the appropriate service.

Class: Client-side Proxy	Collaborators
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	• Client • Broker

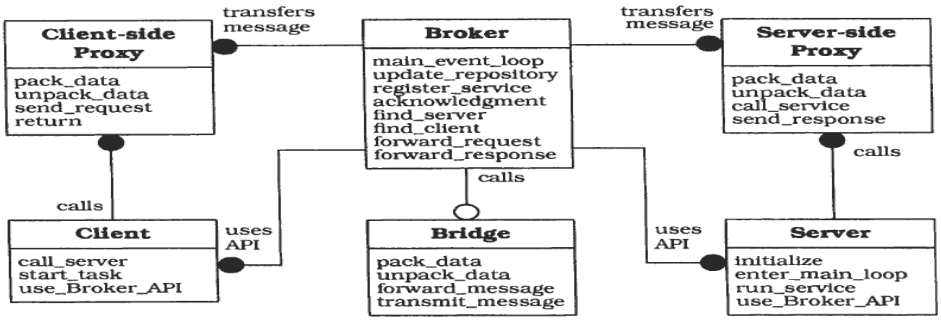
Class: Server-side Proxy	Collaborators
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	• Server • Broker

★ **Bridges:**

- These are optional components used for hiding implementation details when two brokers interoperate.

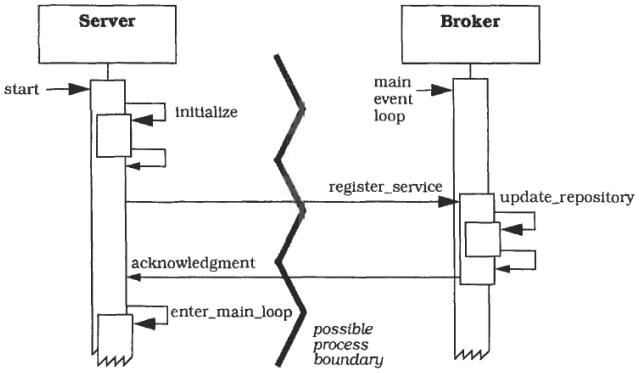
Class: Bridge	Collaborators
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	• Broker • Bridge

The following diagram shows the objects involved in a broker system.

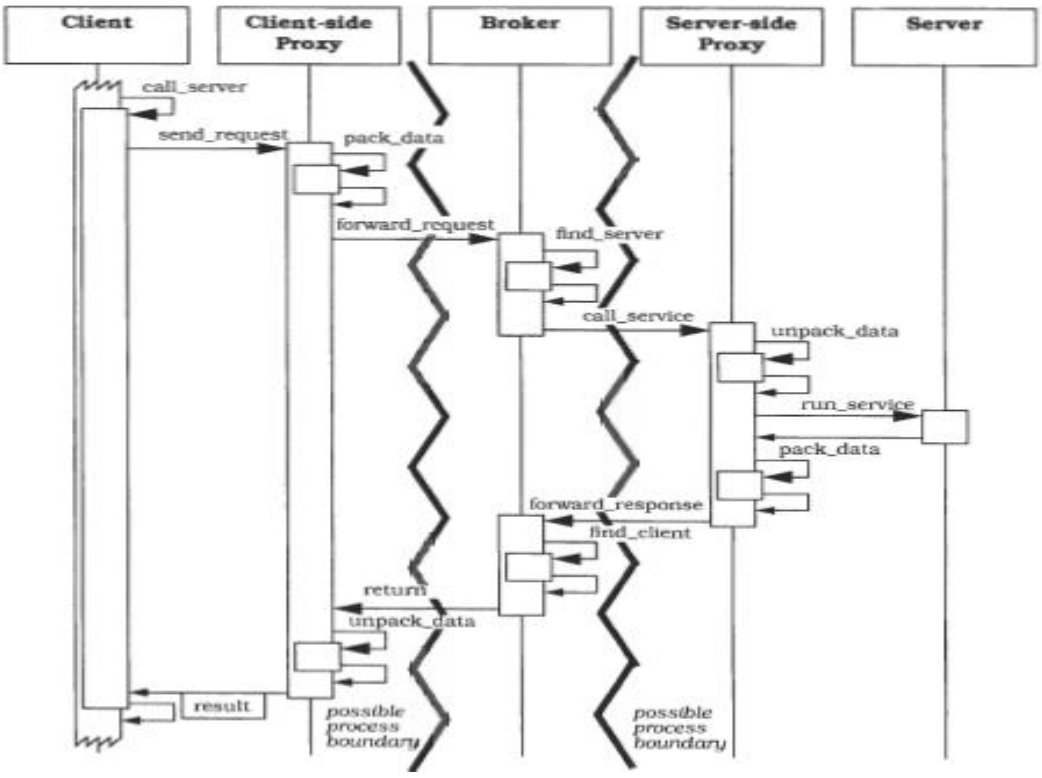


Dynamics:

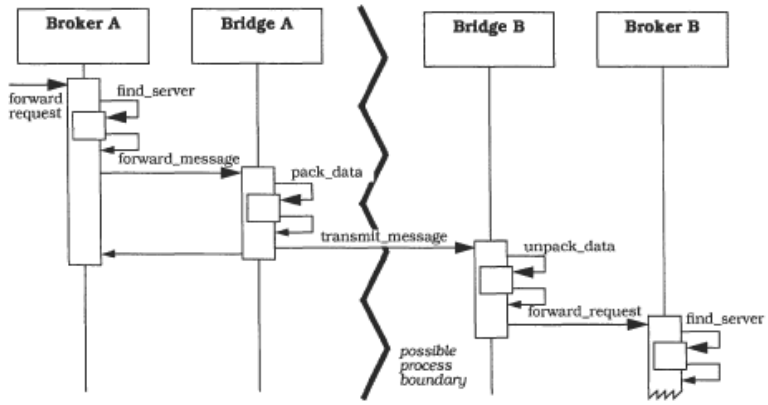
- **Scenario 1.** illustrates the behaviour when a server registers itself with the local broker component:



- **Scenario II** illustrates the behaviour when a client sends a request to a local server. In this scenario we describe a synchronous invocation, in which the client blocks until it gets a response from the server. The broker may also support asynchronous invocations, allowing clients to execute further tasks without having to wait for a response.



- **Scenario III** illustrates the interaction of different brokers via bridge components:



[Please refer text book if you need detailed explanation of scenarios]

Implementation:

1) **Define an object existing model, or use an existing model.**
   Each object model must specify entities such as object names, requests, objects, values, exceptions, supported types, interfaces and operations.
2) **Decide which kind of component-interoperability the system should offer.**
   - You can design for interoperability either by specifying a binary standard or by introducing a high-level IDL.
   - IDL file contains a textual description of the interfaces a server offers to its clients.
   - The binary approach needs support from your programming language.
3) **Specify the API'S the broker component provides for collaborating with clients and servers.**
   - Decide whether clients should only be able to invoke server operations statically, allowing clients to bind the invocations at complete time, or you want to allow dynamic invocations of servers as well.
   - This has a direct impact on size and no. of API'S.
4) **Use proxy objects to hide implementation details from clients and servers.**
   - Client side proxies package procedure calls into message and forward these messages to the local broker component.
   - Server side proxies receive requests from the local broker and call the methods in the interface implementation of the corresponding server.
5) **Design the broker component in parallel with steps 3 and 4**
   During design and implementations, iterate systematically through the following steps
   5.1 Specify a detailed on-the-wire protocol for interacting with client side and server side proxies.
   5.2 A local broker must be available for every participating machine in the network.
   5.3 When a client invokes a method of a server the broker system is responsible for returning all results and exceptions back to the original client.
   5.4 If the provides do not provide mechanisms for marshalling and un marshalling parameters results, you must include functionality in the broker component.
   5.5 If your system supports asynchronous communication b/w clients and servers, you need to provide message buffers within the broker or within the proxies for temporary storage of messages.
   5.6 Include a directory service for associating local server identifiers with the physical location of the corresponding servers in the broker.
   5.7 When your architecture requires system-unique identifiers to be generated dynamically during server registration, the broker must offer a name service for instantiating such names.
   5.8 If your system supports dynamic method invocation the broker needs some means for maintaining type information about existing servers.
   5.9 Plan the broker's action when the communication with clients, other brokers, or servers fails.
6) **Develop IDL compliers**
   An IDL compiler translates the server interface definitions to programming language code. When many programming languages are in use, it is best to develop the compiler as afrarnework that allows the developer to add his own code generators.

Example resolved:

Our example CIS system offers different kinds of services. For example, a separate server workstation provides all the information related to public transport. Another server is responsible for collecting and publishing information on vacant hotel rooms. A tourist may be interested in retrieving information from several hotels, so we decide to provide this data on a single workstation. Every hotel can connect to the workstation and perform updates.

Variants:

- **Direct communication broker system:**
  - ★ We may sometime choose to relax the restriction that clients can only forward requests through the local brokers for efficiency reasons
  - ★ In this variant, clients can communicate with server directly.
  - ★ Broker tells the clients which communication channel the server provides.
  - ★ The client can then establish a direct link to the requested server
- **Message passing broker system:**
  - ✓ This variant is suitable for systems that focus on the transmission of data, instead of implementing a remote produce call abstraction.
  - ✓ In this context, a message is a sequence of raw data together with additional information that specifies the type of a message, its structure and other relevant attributes.
  - ✓ Here servers use the type of a message to determine what they must do, rather than offering services that clients can invoke.
- **Trader system**:
  - ✓ A client request is usually forwarded to exactly one uniquely – identified servers.
  - ✓ In a trader system, the broker must know which server(s) can provide the service and forward the request to an appropriate server.
  - ✓ Here client side proxies use service identifiers instead of server identifiers to access server functionality.
  - ✓ The same request might be forwarded to more than one server implementing the same service.
- **Adapter broker system:**
  - ✓ Adapter layer is used to hide the interfaces of the broker component to the servers using additional layer to enhance flexibility
  - ✓ This adapter layer is a part of the broker and is responsible for registering servers and interacting with servers.
  - ✓ By supplying more than one adapter, support different strategies for server granularity and server location.
  - ✓ Example: use of an object oriented database for maintaining objects.
- **Callback broker system:**
  - ✓ Instead of implementing an active communication model in which clients produce requests and servers consume then and also use a reactive model.
  - ✓ It's a reactive model or event driven, and makes no distinction b/w clients and servers.
  - ✓ Whenever an event arrives, the broker invokes the call back method of the component that is registered to react to the event
  - ✓ The execution of the method may generate new events that in turn cause the broker to trigger new call back method invocations.

Known uses:

- ♠ CORBA
- ♠ SOM/DSOM
- ♠ OLE 2.x
- ♠ WWW
- ♠ ATM-P

[Please refer text book if you need detailed explanation of these uses]

Consequences:

The broker architectural pattern has some important *Benefits*:

- **Location transparency:**
  Achieved using the additional 'broker' component
- **Changeability and extensibility of component**
  If servers change but their interfaces remain the same, it has no functional impact on clients.
- **Portability of a broker system:**
  Possible because broker system hides operating system and network system details from clients and servers by using indirection layers such as API'S, proxies and bridges.
- **Interoperability between different broker systems.**
  Different Broker systems may interoperate if they understand a common protocol for the exchange of messages.
- **Reusability**
  When building new client applications, you can often base the functionality of your application on existing services.

The broker architectural pattern has some important *Liabilities:*

- ♠ **Restricted efficiency:**
  Broker system is quite slower in execution.
- ♠ **Lower fault tolerance:**
  Compared with a non-distributed software system, a Broker system may offer lower fault tolerance.

Following aspect gives benefits as well as liabilities.

- ★ **Testing and debugging:**
  Testing is more robust and easier itself to test. However it is a tedious job because of many components involved.

INTERACTIVE SYSTEMS

These systems allow a high degree of user interaction, mainly achieved with the help of graphical user interfaces.

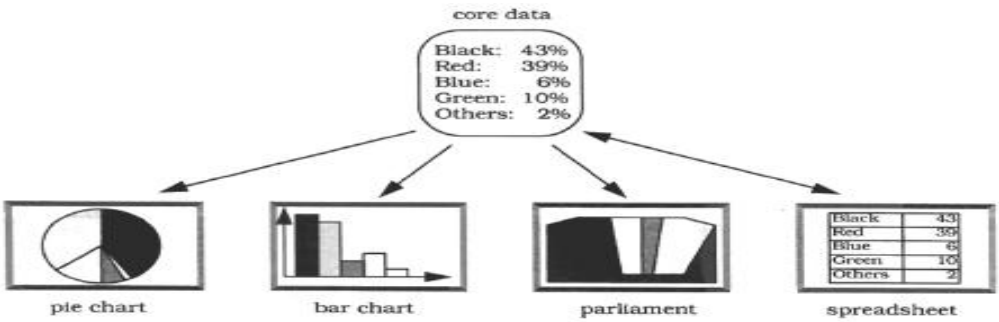
Two patterns that provide a fundamental structural organization for interactive software systems are:

- Model-view-controller pattern
- Presentation-abstraction-control pattern

MODEL-VIEW-CONTROLLER (MVC)

- *MVC architectural pattern divides an interactive application into three components.*
  - ✓ *The model contains the core functionality and data.*
  - ✓ *Views display information to the user.*
  - ✓ *Controllers handle user input.*
- *Views and controllers together comprise the user interface.*
- *A change propagation mechanism ensures consistence between the user interface and the model.*

Example:



Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting the current results. Users can interact with the system via a graphical interface. All information displays must reflect changes to the voting data immediately.

Context:
Interactive applications with a flexible human-computer interface

Problem:
Different users place conflicting requirements on the user interface. A typist enters information into forms via the keyboard. A manager wants to use the same system mainly by clicking icons and buttons. Consequently, support for several user interface paradigms should be easily incorporated. How do you modularize the user interface functionality of a web application so that you can easily modify the individual parts?
The following forces influence the solution:

- ✓ Same information is presented differently in different windows. For ex: In a bar or pie chart.
- ✓ The display and behavior of the application must reflect data manipulations immediately.
- ✓ Changes to the user interface should be easy, and even possible at run-time.
- ✓ Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Solution:

- ★ MVC divides an interactive application into the three areas: processing, output and input.
- ★ Model component encapsulates core data and functionality and is independent of o/p and i/p.
- ★ View components display user information to user a view obtains the data from the model. There can be multiple views of the model.
- ★ Each view has an associated controller component controllers receive input (usually as mouse events) events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.
- ★ The separation of the model from view and controller components allows multiple views of the same model.

Structure:

- ★ **Model component:**
  - o Contains the functional core of the application.
  - o Registers dependent views and controllers
  - o Notifies dependent components about data changes (change propagation mechanism)

Class Model	Collaborators • View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

- ★ **View component:**
  - o Presents information to the user
  - o Retrieves data from the model
  - o Creates and initializes its associated controller
  - o Implements the update procedure

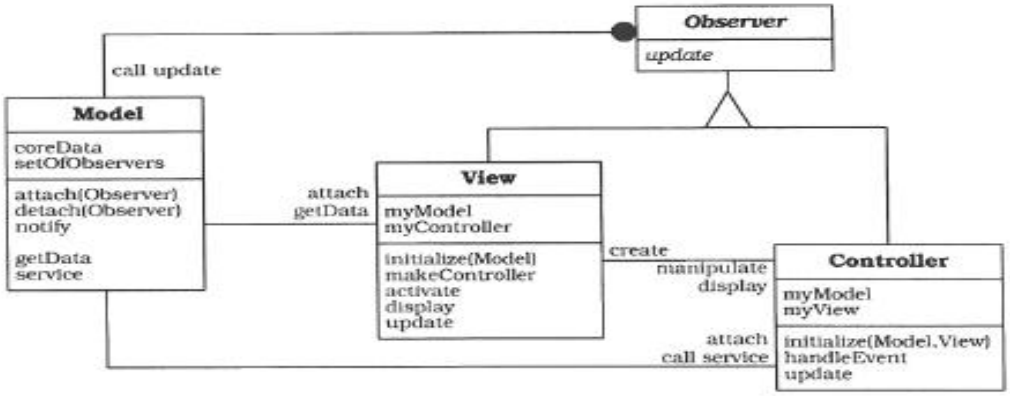
- ★ **Controller component:**
  - o Accepts user input as events (mouse event, keyboard event etc)
  - o Translates events to service requests for the model or display requests for the view.

- The controller registers itself with the change-propagation mechanism and implements an update procedure.

Class: View	Collaborators
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	• Controller • Model

Class: Controller	Collaborators
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	• View • Model

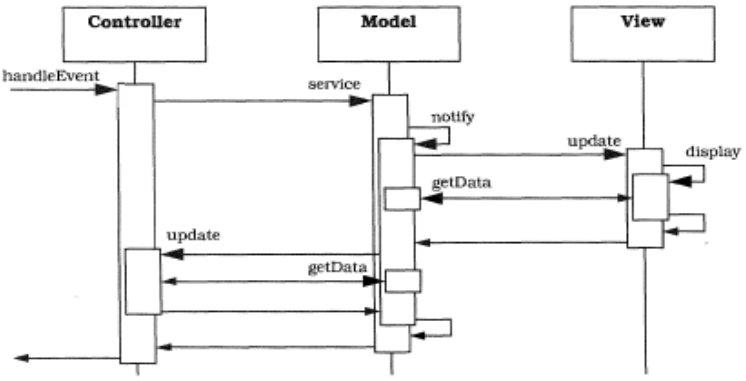
An object-oriented implementation of MVC would define a separate class for each component. In a C++ implementation, view and controller classes share a common parent that defines the update interface. This is shown in the following diagram.



Dynamics:

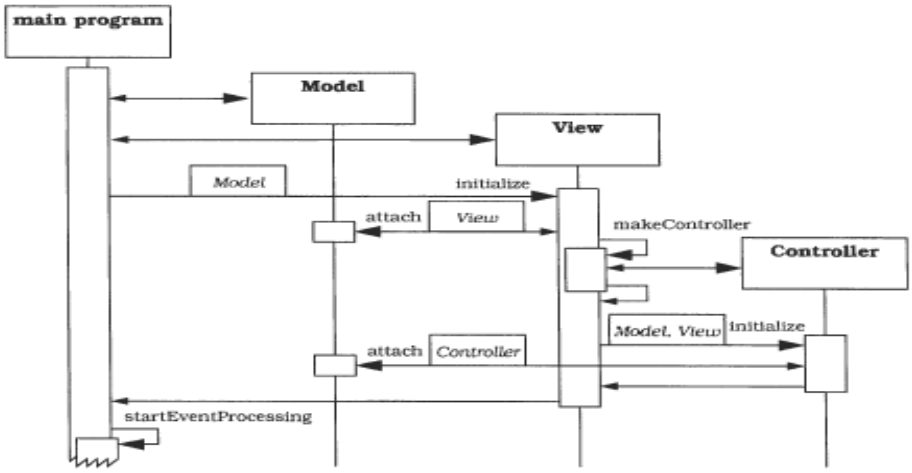
The following scenarios depict the dynamic behavior of MVC. For simplicity only one view-controller pair is shown in the diagrams.

- **Scenario** I shows how user input that results in changes to the model triggers the change-propagation mechanism:
  - The controller accepts user input in its event-handling procedure, interprets the event, and activates a service procedure of the model.
  - The model performs the requested service. This results in a change to its internal data.
  - The model notifies all views and controllers registered with the change-propagation mechanism of the change by calling their update procedures.
  - Each view requests the changed data from the model and redisplays itself on the screen.
  - Each registered controller retrieves data from the model to enable or disable certain user functions..
  - The original controller regains control and returns from its event handling procedure.



- **Scenario II** shows how the MVC triad is initialized. The following steps occur:
  - The model instance is created, which then initializes its internal data structures.
  - **A** view object is created. This takes a reference to the model as a parameter for its initialization.

- The view subscribes to the change-propagation mechanism of the model by calling the attach procedure.
- The view continues initialization by creating its controller. It passes references both to the model and to itself to the controller's initialization procedure.
- The controller also subscribes to the change-propagation mechanism by calling the attach procedure.
- After initialization, the application begins to process events.



Implementation:

1) Separate human-computer interaction from core functionality
- Analysis the application domain and separate core functionality from the desired input and output behavior

2) Implement the change-propagation mechanism
- Follow the publisher subscriber design pattern for this, and assign the role of the publisher to the model.

3) Design and implement the views
- design the appearance of each view
- Implement all the procedures associated with views.

4) Design and implement the controllers
- For each view of application, specify the behavior of the system in response to user actions.
- We assume that the underlying pattern delivers every action of and user as an event. A controller receives and interprets these events using a dedicated procedure.

5) Design and implement the view controller relationship.
- A view typically creates its associated controller during its initialization.

6) Implement the setup of MVC.
- The setup code first initializes the model, then creates and initializes the views.
- After initialization, event processing is started.
- Because the model should remain independent of specific views and controllers, this set up code should be placed externally.

7) Dynamic view creation
- If the application allows dynamic opening and closing of views, it is a good idea to provide a component for managing open views.

8) 'pluggable' controllers
- The separation of control aspects from views supports the combination of different controllers with a view.
- This flexibility can be used to implement different modes of operation.

9) Infrastructure for hierarchical views and controllers
- Apply the composite pattern to create hierarchically composed views.
- If multiple views are active simultaneously, several controllers may be interested in events at the same time.

10) Further decoupling from system dependencies.

- Building a framework with an elaborate collection of view and controller classes is expensive. You may want to make these classes platform independent. This is done in some Smalltalk systems

Variants:

Document View - This variant relaxes the separation of view and controller. In several GUI platforms, window display and event handling are closely interwoven. You can combine the responsibilities of the view and the controller from MVC in a single component by sacrificing exchangeability of controllers. This kind of structure is often called Document-View architecture. The view component of Document-View combines the responsibilities of controller and view in MVC, and implements the user interface of the system.

Known uses:

- **Smalltalk** - The best-known example of the use of the Model-View-Controller pattern is the user-interface framework in the Smalltalk environment
- **MFC** - The Document-View variant of the Model-View-Controller pattern is integrated in the Visual C++ environment-the Microsoft Foundation Class Library-for developing Windows applications.
- **ET++** - ET++ establishes 'look and feel' independence by defining a class **`Windowport`** that encapsulates the user interface platform dependencies.

Consequences:

Benefits:

- **Multiple views of the same model:**
  It is possible because MVC strictly separates the model from user interfaces components.
- **Synchronized views:**
  It is possible because of change-propagation mechanism of the model.
- **'pluggable views and controller:**
  It is possible because of conceptual separation of MVC.
- **Exchangeability of 'look and feel'**
  Because the model is independent of all user-interface code, a port of MVC application to a new platform does not affect the functional core of the application.
- **Framework potential**
  It is possible to base an application framework on this pattern.

Liabilities:

- **Increased complexity**
  Following the Model-View-Controller structure strictly is not always the best way to build an interactive application
- **Potential for excessive number of updates**
  If a single user action results in many updates, the model should skip unnecessary change notifications.
- **Intimate connection b/w view and controller**
  Controller and view are separate but closely-related components, which hinders their individual reuse.
- **Closed coupling of views and controllers to a model**
  Both view and controller components make direct calls to the model. This implies that changes to the model's interface are likely to break the code of both view and controller.
- **Inevitability of change to view and controller when porter**
  This is because all dependencies on the user-interface platform are encapsulated within view and controller.
- ★ **Difficulty of using MVC with modern user interface tools**
  If portability is not an issue, using high-level toolkits or user interface builders can rule out the use of MVC.

PRESENTATION-ABSTRACTION-CONTROL

PAC defines a structure for interactive s/w systems in the form of a hierarchy of cooperating agents.

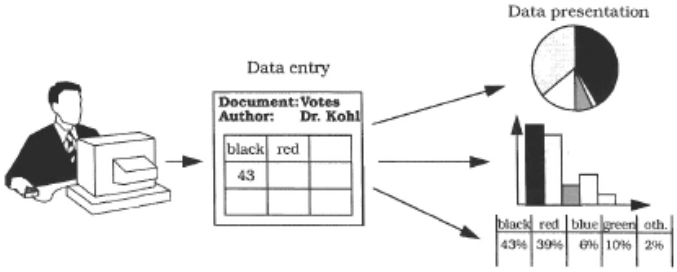
Every agent is responsible for a specific aspect of te applications functionality and consists of three components

- ✓ *Presentation*
- ✓ *Abstraction*
- ✓ *Control*

The subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Example:

Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting current standings. Users interact with the software through a graphical interface.



Context:

Development of an interactive application with the help of agents

Problem:

Agents specialized in human-comp interaction accept user input and display data. Other agents maintain the data model of the system and offer functionality that operates on this data. Additional agents are responsible for error handling or communication with other software systems

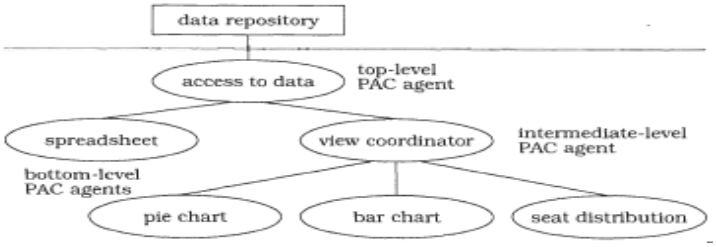
The following forces affect solution:

- ★ Agents often maintain their own state and data however, individual agents must effectively co operate to provide the overall task of the application. To achieve this they need a mechanism for exchanging data, messages and events.
- ★ Interactive agents provide their own user interface, since their respective human-comp interactions often differ widely
- ★ Systems evolve over time. Their presentation aspect is particularly prone to change. The use of graphics, and more recently, multimedia features are ex: of pervasive changes to user interfaces. Changes to individual agents, or the extension of the system with new agents, should not affect the whole system.

Solution:

- ♠ Structure the interactive application as a tree-like hierarchy f PAC agents every agent is responsible for a specific agent of the applications functionality and consists of three components:
  - ▶ Presentation
  - ▶ Abstraction
  - ▶ Control
- ♠ The agents presentation component provides the visible behavior of the PAC agent
- ♠ Its abstraction component maintains the data model that underlies the agent, and provides functionality that operates on this data.
- ♠ Its control component connects the presentation and abstraction components and provides the functionality that allow agent to communicate with other PAC agents.
- ♠ The top-level PAC agent provides the functional core of the system. Most other PAC agents depend or operate on its core.

- ♠ The bottom-level PAC agents represent self contained semantic concepts on which users of the system can act, such as spreadsheets and charts.
- ♠ The intermediate-level PAC agents represent either combination of, or relationship b/w. lower level agent.



Structure:

★ **Top level PAC agent:**
- Main responsibility is to provide the global data model of the software. This is maintained in the abstraction component of top level agent.
- The presentation component of the top level agent may include user-interface elements common to whole application.
- The control component has three responsibilities
  - Allows lower level agents to make use of the services of manipulate the global data model.
  - It co ordinates the hierarchy of PAC agents. It maintains information about connections b/w top level agent and lower-level agents.
  - It maintains information about the interaction of the user with system.

★ **Bottom level PAC agent:**
- Represents a specific semantic concept of the application domain, such as mail box in a n/w management system.
- The presentation component of bottom level PAC agents presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it.
- The abstraction component has a similar responsibility as that of top level PAC agent maintaining agent specific data.
- The control component maintains consistency b/w the abstraction and presentation components, by avoiding direct dependencies b/w them. It serves as an adapter and performs both interface and data adaption.

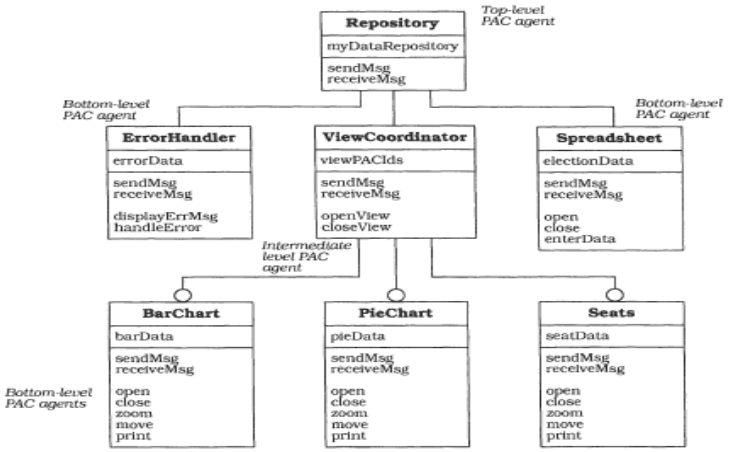
★ **Intermediate level PAC agent**
- Can fulfill two different roles: composition and co ordination.
- It defines a new abstraction, whose behavior encompasses both the behavior of its component, and the new characteristics that are added to the composite object.
- Abstraction component maintains the specific data of the intermediate level PAC agent.
- Presentation component implements its user interface.
- Control component has same responsibilities of those of bottom level and top level PAC agents.

Class	Collaborators
Top-level Agent	• Intermediate-level Agent • Bottom-level Agent
Responsibility	
• Provides the functional core of the system. • Controls the PAC hierarchy.	

Class	Collaborators
Interm. -level Agent	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent
Responsibility	
• Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	

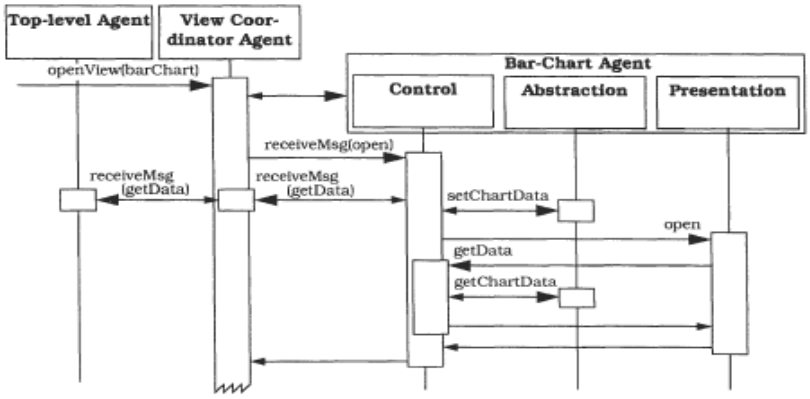
Class	Collaborators
Bottom-level Agent	• Top-level Agent • Intermediate-level Agent
Responsibility	
• Provides a specific view of the software or a system service, including its associated human-computer interaction.	

The following OMT diagram illustrates the PAC hierarchy of the information system for political elections

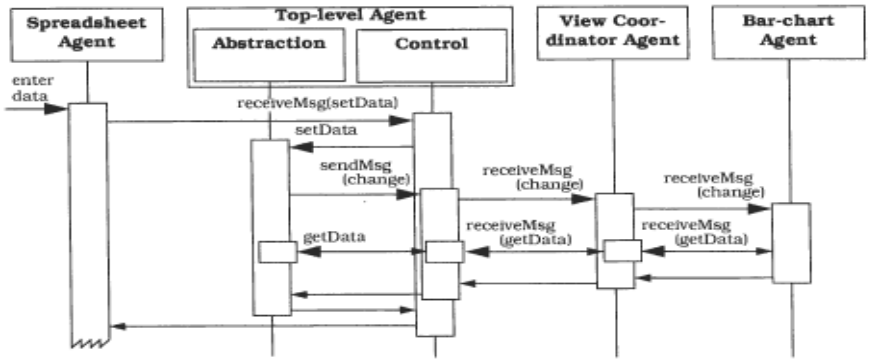


Dynamics:

- **Scenario I** describes the cooperation between different PAC agents when opening a new bar-chart view of the election data. It is divided into five phases:
  - A user asks the presentation component of the view coordinator agent to open a new bar chart.
  - The control of the view coordinator agent instantiates the desired bar-chart agent.
  - The view coordinator agent sends an 'open' event to the control component of the new bar-chart agent.
  - The control component of the bar-chart agent first retrieves data from the top-level PAC agent. The view coordinator agent mediates between bottom and top-level agents. The data returned to the bar-chart agent is saved in its abstraction component. Its control component then calls the presentation component to display the chart.
  - The presentation component creates a new window on the screen, retrieves data from the abstraction component by requesting it from the control component, and finally displays it within the new window.



- **Scenario II** shows the behavior of the system after new election data is entered, providing a closer look at the internal behavior of the toplevel PAC agent. It has five phases:
  - The user enters new data into a spreadsheet. The control component of the spreadsheet agent forwards this data to the toplevel PAC agent.
  - The control component of the top-level PAC agent receives the data and tells the top-level abstraction to change the data repository accordingly. The abstraction component of the top-level agent asks its control component to update all agents that depend on the new data. The control component of the top-level PAC agent therefore notifies the view coordinator agent.
  - The control component of the view coordinator agent forwards the change notification to all view PAC agents it is responsible for coordinating.
  - As in the previous scenario, all view PAC agents then update their data and refresh the image they display.



Implementation:

1) **Define a model of the application**
   Analyze the problem domain and map it onto an appropriate s/w structure.
2) **Define a general strategy for organizing the PAC hierarchy**
   Specify general guidelines for organizing the hierarchy of co operating agents.
   One rule to follow is that of "lowest common ancestor". When a group of lower level agents depends on the services or data provided by another agent, we try to specify this agent as the root of the sub tree formed by the lower level agents. As a consequence, only agents that provide global services rise to the top of the hierarchy.
3) **Specify the top-level PAC agent**
   Identify those parts of the analysis model that represent the functional core of the system.
4) **Specify the bottom-level PAC agent**
   Identify those components of the analysis model that represent the smallest self-contained units of the system on which the user can perform operations or view presentations.
5) **Specify bottom-level PAC agents for system service**
   Often an application includes additional services that are not directly related to its primary subject. In our example system we define an error handler.
6) **Specify intermediate level PAC agents to compose lower level PAC agents**
   Often, several lower-level agents together form a higher-level semantic concept on which users can operate.
7) **Specify intermediate level PAC agents to co ordinate lower level PAC agents**
   Many systems offer multiple views of the same semantic concept. For example, in text editors you find 'layout' and 'edit" views of a text document. When the data in one view changes, all other views must be updated.
8) **Separate core functionality from human-comp interaction.**
   For every PAC agent, introduce presentation and abstraction component.
9) **Provide the external interface to operate with other agents**
   Implement this as part of control component.
10) **Link the hierarchy together**
    Connect every PAC agent with those lower level PAC agents with which it directly co operates

Variants:

- ★ **PAC agents as active objects** - Every PAC agent can be implemented as an active object that lives in its own thread of control.
- ★ **PAC agents as processes** - To support PAC agents located in different processes or on remote machines, use proxies to locally represent these PAC agents and to avoid direct dependencies on their physical location.

Known uses:

- **Network traffic management**
  - Gathering traffic data from switching units.
  - Threshold checking and generation of overflow exceptions.

- Logging and routing of network exceptions.
- Visualization of traffic flow and network exceptions.
- Displaying various user-configurable views of the whole network.
- Statistical evaluations of traffic data.
- Access to historic traffic data.
- System administration and configuration.

- **Mobile robot**
  - Provide the robot with a description of the environment it will work in, places in this environment, and routes between places.
  - Subsequently modify the environment.
  - Specify missions for the robot.
  - Control the execution of missions.
  - Observe the progress of missions.

Consequences:

Benefits:-

- **separation of concerns**
  - Different semantic concepts in the application domain are represented by separate agents.
- **Support for change and extension**
  - Changes within the presentation or abstraction components of a PAC agent do not affect other agents in the system.
- **Support for multi tasking**
  - PAC agents can be distributed easily to different threads, processes or machines.
  - Multi tasking also facilitates multi user applications.

Liabilities:

- ✓ **Increased system complexity**
  Implementation of every semantic concept within an application as its own PAC agent may result in a complex system structure.
- ✓ **Complex control component**
  - Individual roles of control components should be strongly separated from each other. The implementations of these roles should not depend on specific details of other agents.
  - The interface of control components should be independent of internal details.
  - It is the responsibility of the control component to perform any necessary interface and data adaptation.
- ✓ **Efficiency:**
  - Overhead in communication between PAC agents may impact system efficiency.
  - Example: All intermediate-level agents are involved in data exchange. if a bottom-level agent retrieve data from top-level agent.
- ✓ **Applicability:**
  - The smaller the atomic semantic concepts of an applications are, and the greater the similarly of their user interfaces, the less applicable this pattern is.

UNIT 5 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the variants of broker architecture	Dec 09	10
2	Depict the dynamic behavior of MVC, with any one scenario	Dec 09	5
3	Give the CRC cards for top level, intermediate level and bottom level PAC-agents	Dec 09	5
4	What do you mean by broker architecture? What are the steps involved in implementing distributed broker architecture patterns?	June 10	10
5	Explain with a neat diagram, the dynamic scenarios of MVC	June 10	10
6	What is the necessity of proxies and bridge components in a broker system? Explain	Dec 10	6
7	Explain the possible dynamic behavior of MVC pattern, with suitable sketches	Dec 10	9
8	Highlight the limitations of PAC pattern	Dec 10	5
9	Give detailed explanation on the different steps involved in the implementation of the broker pattern	June 11	15
10	Propose the description of a scenario that depicts the dynamic behavior of MVC in detail. Support the description with appropriate pictorial representation	June 11	5
11	Discuss the most relevant scenario, illustrating the dynamic behavior of a broker system	Dec 11	10
12	Discuss the consequences of PAC architectural pattern	Dec 11	10
13	Define broker architectural pattern. Explain with a diagram the objects involved in a broker system	June 12	7
14	Depict the dynamic behavior of MVC, with any one scenario	June 12	7
15	Give the CRC cards for top level, intermediate level and bottom level PAC-agents	June 12	6

UNIT 6
ARCHITECTURAL PATTERNS-3

ADAPTABLE SYSTEMS

The systems that evolve over time - new functionality is added and existing services are changed are called adaptive systems.
They must support new versions of OS, user-interface platforms or third-party components and libraries.
Design for change is a major concern when specifying the architecture of a software system.
We discuss two patterns that helps when designing for change.

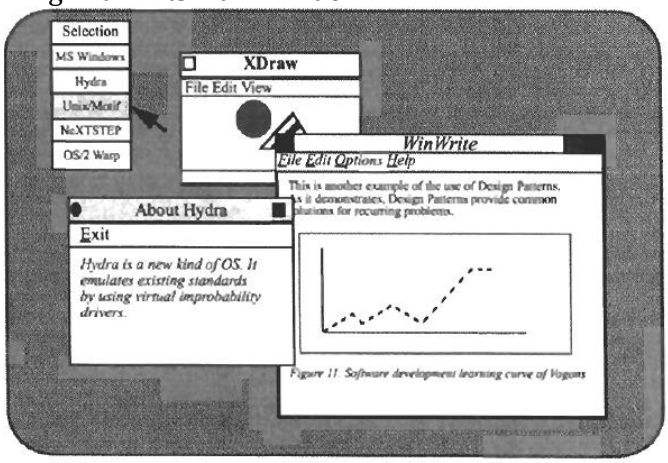
- ***Microkernel pattern*** → applies to software systems that must be able to adapt to changing system requirements.
- ***Reflection pattern*** → provides a mechanism for changing structure and behavior of software systems dynamically.

MICROKERNEL

The microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts the microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

Example:

Suppose we intend to develop a new operating system for desktop computers called Hydra. One requirement is that this innovative operating system must be easily portable to the relevant hardware platforms, and must be able to accommodate future developments easily. It must also be able to run applications written for other popular operating systems such as Microsoft Windows and UNIX System V. A user should be able to choose which operating system he wants from a pop-up menu before starting an application. Hydra will display `all` the applications currently running within its main window:



Context:

The development of several applications that use similar programming interfaces that build on the same care functionality.

Problem:

Developing software for an application domain that needs to cope with a broad spectrum of similar standards and technology is a nontrivial task well known. Ex: are application platform such as OS and GUI'S.
The following forces are to be considered when designing such systems.

- The application platform must cope with continuous hardware and software evolution.

- ♣ The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies.

Application platform such as an OS should also be able to emulate other application platforms that belong to the same application domain. This leads to following forces.

- ♠ The applications in your domain need to support different but, similar application platforms.
- ♠ The applications may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.

Additional force must be taken to avoid performance problem and to guarantee scalability.

- The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

Solution:

- Encapsulates the fundamental services of your applications platform in a microkernel component.
- It includes functionality that enables other components running in different process to communicate with each other.
- It provides interfaces that enable other components to access its functionality.
  - Core functionality should be included in ***internal servers.***
  - ***External servers*** implement their own view of the underlying microkernel.
  - ***Clients*** communicate with external servers by using the communication facilities provided by microkernel.

Structure:

Microkernel pattern defines 5 kinds of participating components.

- ♠ Internal servers
- ♠ External servers
- ♠ Adapters
- ♠ Clients
- ♠ Microkernel

❖ **Microkernel**

- The microkernel represents the main component of the pattern.
- It implements central services such as communication facilities or resource handling.
- The microkernel is also responsible for maintaining system resources such as processes or files.
- It controls and coordinates the access to these resources.
- A microkernel implements atomic services, which we refer to as mechanisms.
- These mechanisms serve as a fundamental base on which more complex functionality called policies are constructed.

Class Microkernel	***Collaborators*** • Internal Server
Responsibility • Provides core mechanisms. • Offers communication facilities. • Encapsulates system dependencies. • Manages and controls resources.	

❖ **An internal server (subsystem)**

- Extends the functionality provided by microkernel.
- It represents a separate component that offers additional functionality.
- Microkernel invokes the functionality of internal services via service requests.
- Therefore internal servers can encapsulates some dependencies on the underlying hardware or software system.

Class Internal Server	Collaborators • Microkernel
Responsibility • Implements additional services. • Encapsulates some system specifics.	

- **An external server (personality)**
  - Uses the microkernel for implementing its own view of the underlying application domain.
  - Each external server runs in separate process.
  - It receives service requests from client applications using the communication facilities provided by the microkernel, interprets these requests, executes the appropriate services, and returns its results to clients.
  - Different external servers implement different policies for specific application domains.

Class External Server	Collaborators • Microkernel
Responsibility • Provides programming interfaces for its clients.	

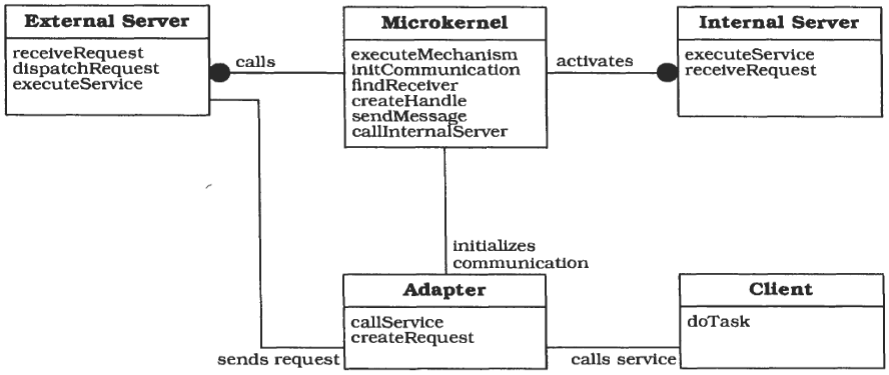
- **Client:**
  - It is an application that is associated with exactly one external server. It only accesses the programming interfaces provided by the external server.
  - Problem arises if a client accesses the interfaces of its external server directly (direct dependency)
    - ✓ Such a system does not support changeability
    - ✓ If ext servers emulate existing application platforms clients will not run without modifications.

- **Adapter** (emulator)
  - Represents the interfaces b/w clients and their external servers and allow clients to access the services of their external server in a portable way.
  - They are part of the clients address space.
  - The following OMT diagram shows the static structure of a microkernel system.

Class Client	Collaborators • Adapter
Responsibility • Represents an application.	

Class Adapter	Collaborators • External Server • Microkernel
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	

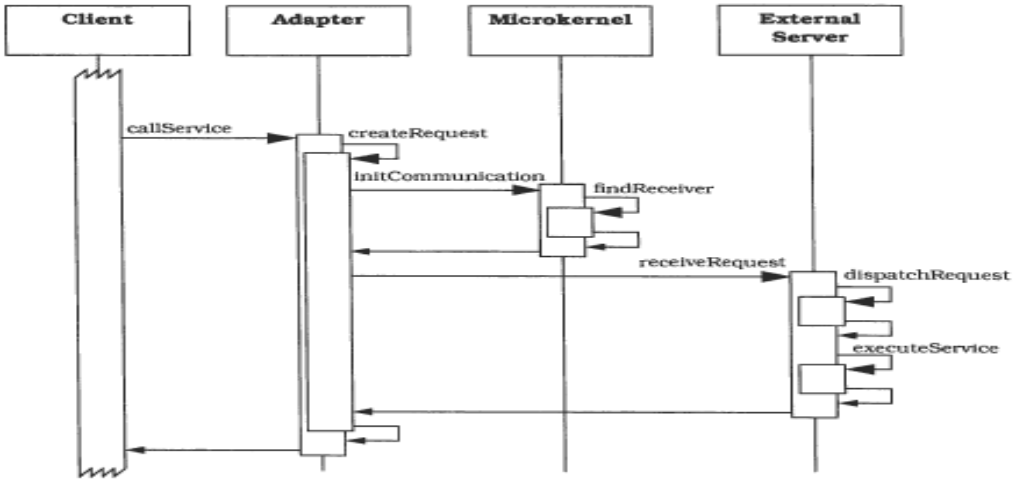
The following OMT diagram shows the static structure of a Microkernel system.



Dynamics:

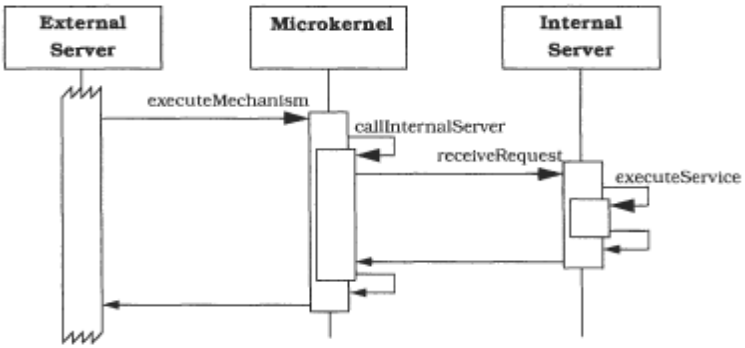
Scenario I demonstrates the behavior when a client calls a service of its external server:

- ★ At a certain point in its control flow the client requests a service from an external server by calling the adapter.
- ★ The adapter constructs a request and asks the microkernel for a communication link with the external server.
- ★ The microkernel determines the physical address of the external server and returns it to the adapter.
- ★ After retrieving this information, the adapter establishes a direct communication link to the external server.
- ★ The adapter sends the request to the external server using a remote procedure call.
- ★ The external server receives the request, unpacks the message and delegates the task to one of its own methods. After completing the requested service, the external server sends all results and status information back to the adapter.
- ★ The adapter returns to the client, which in turn continues with its control flow.



Scenario II illustrates the behavior of a Microkernel architecture when an external server requests a service that is provided by an internal server

- ★ The external server sends a service request to the microkernel.
- ★ A procedure of the programming interface of the microkernel is called to handle the service request. During method execution the microkernel sends a request to an internal server.
- ★ After receiving the request, the internal server executes the requested service and sends all results back to the microkernel.
- ★ The microkernel returns the results back to the external server.
- ★ Finally, the external server retrieves the results and continues with its control flow.



Implementation:

1. **Analyze the application domain:**
   Perform a domain analysis and identify the core functionality necessary for implementing ext servers.
2. **Analyze external servers**
   That is polices ext servers are going to provide
3. **Categorize the services**
   Group all functionality into semantically independent categories.
4. **Partition the categories**
   Separate the categories into services that should be part of the microkernel, and those that should be available as internal servers.
5. **Find a consistent and complete set of operations and abstractions**
   for every category you identified in step 1.
6. **Determine the strategies for request transmission and retrieval.**
   - Specify the facilities the microkernel should provide for communication b/w components.
   - Communication strategies you integrate depend on the requirements of the application domain.
7. **Structure the microkernel component**
   Design the microkernel using the layers pattern to separate system-specific parts from system-independent parts of the kernel.
8. **Specify the programming interfaces of microkernel**
   To do so, you need to decide how these interfaces should be accessible externally.
   You must take into an account whether the microkernel is implemented as a separate process or as a module that is physically shared by other component in the latter case, you can use conventional method calls to invoke the methods of the microkernel.

9. The microkernel is responsible for **managing all system resources** such as memory blocks, devices or **device contexts** - a handle to an output area in a GUI implementation.

10. **Design and implement the internal servers as separate processes or shared libraries**
    - Perform this in parallel with steps 7-9, because some of the microkernel services need to access internal servers.
    - It is helpful to distinguish b/w active and passive servers
      - ✓ Active servers are implemented as processes
      - ✓ Passive servers as shared libraries.
    - Passive servers are always invoked by directly calling their interface methods, where as active server process waits in an event loop for incoming requests.

11 **Implement the external servers**
- Each external server is implemented as a separate process that provide its own service interface
- The internal architecture of an external server depends on the policies it comprises
- Specify how external servers dispatch requests to their internal procedures.

12. **Implement the adapters**
- Primary task of an adapter is to provide operations to its clients that are forwarded to an external server.
- You can design the adapter either as a conventional library that is statically linked to client during complication or as a shared library dynamically linked to the client on demand.

13. **Develop client applications**
or use existing ones for the ready-to-run microkernel system.

Example resolved:
Shortly after the development of Hydra has been completed, we are asked to integrate an external server that emulates the Apple MacOS operating system. To provide a MacOS emulation on top of Hydra, the following activities are necessary:

- ✓ *Building an external server* on top of the Hydra microkernel that implements all the programming interfaces provided by MacOS, including the policies of the Macintosh user interface.
- ✓ *Providing an adapter* that is designed as a library, dynamically linked to clients.
- ✓ *Implementing the internal servers* required for MacOS.

Variants:

- *Microkernel system with indirect client-server communications.*
  In this variant, a client that wants to send a request or message to an external server asks the microkernel for a communication channel.
- *Distributed microkernel systems.*
  In this variant a microkernel can also act as a message backbone responsible for sending messages to remote machines or receiving messages from them.

Known uses:

- The **Mach** microkernel is intended to form a base on which other operating systems can be emulated. One of the commercially available operating systems that use Mach as its system kernel is NeXTSTEP.
- The operating system **Amoeba** consists of two basic elements: the microkernel itself and a collection of servers (subsystems) that are used to implement the majority of Amoeba's functionality. The kernel provides four basic services: the management of processes and threads, the low-level-management of system memory, communication services, both for point-to-point communication as well as group-communication, and low-level I/O services.
- **Chorus** is a commercially-available Microkernel system that was originally developed specifically for real-time applications.
- **Windows NT** was developed by Microsoft as an operating system for high-performance servers.
- **MKDE** (Microkernel Datenbank Engine) system introduces an architecture for database engines that follows the Microkernel pattern.

Consequences:
The microkernel pattern offers some important *Benefits:*

- **Portability:**
  High degree of portability
- **Flexibility and extensibility:**
  - ✓ If you need to implement an additional view, all you need to do is add a new external server.
  - ✓ Extending the system with additional capabilities only requires the additional or extension of internal servers.
- **Separation of policy and mechanism**
  The microkernel component provides all the mechanisms necessary to enable external servers to implement their policies.

Distributed microkernel has further benefits:

- ♠ **Scalability**
  A distributed Microkernel system is applicable to the development of operating systems or database systems for computer networks, or multiprocessors with local memory
- ♠ **Reliability**
  A distributed Microkernel architecture supports availability, because it allows you to run the same server on more than one machine, increasing availability. Fault tolerance may be easily supported because distributed systems allow you to hide failures from a user.

- **Transparency**
  In a distributed system components can be distributed over a network of machines. In such a configuration, the Microkernel architecture allows each component to access other components without needing to know their location.

The microkernel pattern also has *Liabilities:*

- **Performance:**
  Lesser than monolithic software system therefore we have to pay a price for flexibility and extensibility.
- **Complexity of design and implementation:**
  Develop a microkernel is a non-trivial task.

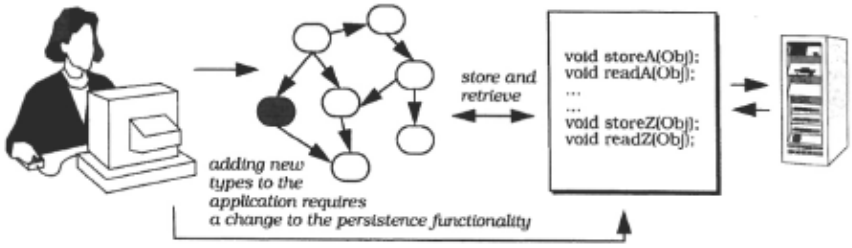
REFLECTION

The reflection architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as the type structures and function call mechanisms. In this pattern, an application is split into two parts:

- *A Meta level provides information about selected system properties and makes the s/w self aware.*
- *A base level includes application logic changes to information kept in the Meta level affect subsequent base-level behavior.*

Example:

Consider a C++ application that needs to write objects to disk and read them in again. Many solutions to this problem, such as implementing type-specific store and read methods, are expensive and error-prone. Persistence and application functionality are strongly interwoven. Instead we want to develop a persistence component that is independent of specific type structures



Context:

Building systems that support their own modification a prior

Problem:

- Designing a system that meets a wide range of different requirements a prior can be an overwhelming task.
- A better solution is to specify an architecture that is open to modification and extension i.e., we have to design for change and evolution.
- Several forces are associated with the problem:
  - ✓ Changing software is tedious, error prone and often expensive.
  - ✓ Adaptable software systems usually have a complex inner structure. Aspects that are subject to change are encapsulated within separate components.
  - ✓ The more techniques that are necessary for keep in a system changeable the more awkward and complex its modifications becomes.
  - ✓ Changes can be of any scale, from providing shortcuts for commonly used commands to adapting an application framework for a specific customer.
  - ✓ Even fundamental aspects of software systems can change for ex. communication mechanisms b/w components.

Solution:

- ★ Make the software self-aware, and make select aspects of its structure and behavior accessible for adaptation and change.
  - This leads to an architecture that is split into two major parts: A Meta level
  - A base level
- ★ Meta level provides a self representation of the s/w to give it knowledge of its own structure and behavior and consists of so called *Meta objects* (they encapsulate and represent information about the software). Ex: type structures algorithms or function call mechanisms.
- ★ Base level defines the application logic. Its implementation uses the Meta objects to remain independent of those aspects that are likely to change.
- ★ An interface is specified for manipulating the Meta objects. It is called the *Meta object protocol* (MOP) and allows clients to specify particular changes.

Structure:

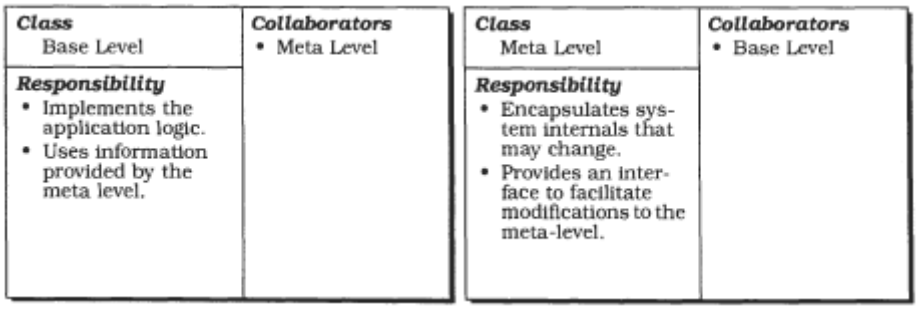
- ♥ Meta level
- ♥ Meta objects protocol(MOP)
- ♥ Base level

❖ **Meta level**

- ✓ Meta level consists of a set of Meta objects. Each Meta object encapsulates selected information about a single aspect of a structure, behavior, or state of the base level.
  There are three sources for such information.
  - It can be provided by run time environment of the system, such as C++ type identification objects.
  - It can be user defined such as function call mechanism
  - It can be retrieved from the base level at run time.
- ✓ All Meta objects together provide a self representation of an application.
- ✓ What you represent with Meta objects depends on what should be adaptable only system details that are likely to change r which vary from customer to customer should be encapsulated by Meta objects.
- ✓ The interface of a Meta objects allows the base level to access the information it maintains or the services it offers.

❖ **Base level**

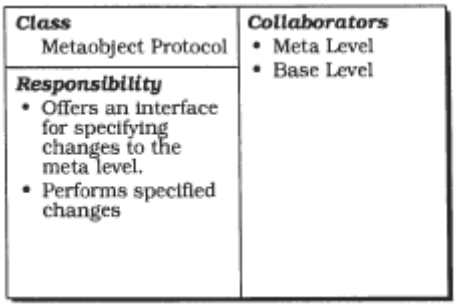
- ✓ It models and implements the application logic of the software. Its component represents the various services the system offers as well as their underlying data model.
- ✓ It uses the info and services provided by the Meta objects such as location information about components and function call mechanisms. This allows the base level to remain flexible.
- ✓ Base level components are either directly connected to the Meta objects and which they depend or submit requests to them through special retrieval functions.



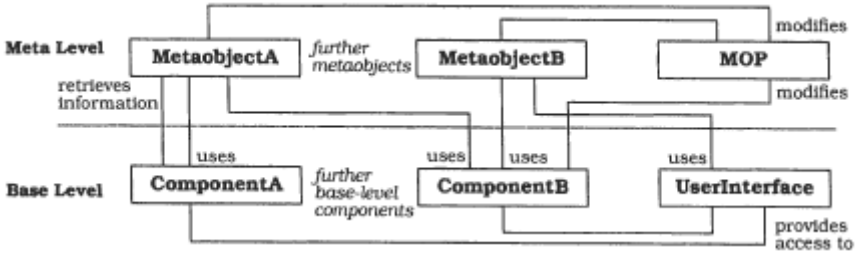
❖ **Meta object protocol (MOP)**

- ✓ Serves an external interface to the Meta level, and makes the implementation of a reflective system accessible in a defined way.
- ✓ Clients of the MOP can specify modifications to Meta objects or their relationships using the base level

- ✓ MOP itself is responsible for performing these changes. This provides a reflective application with explicit control over its own modification.
- ✓ Meta object protocol is usually designed as a separate component. This supports the implementation of functions that operate on several Meta objects.
- ✓ To perform changes, the MOP needs access to the internals of Meta objects, and also to base level components (sometimes).
  One way of providing this access is to allow the MOP to directly operate on their internal states. Another way (safer, inefficient) is to provide special interface for their manipulation, only accessible by MOP.



The general structure of a reflective architecture is very much like a Layered system



Dynamics:

Interested students can refer text book for scenarios since the diagrams are too hectic & can't be memorized

Implementation:

Iterate through any subsequence if necessary.

1. **Define a model of the application**
   Analyze the problem domain and decompose it into an appropriate s/w structure.
2. **Identify varying behavior**
   - ✓ Analyze the developed model and determine which of the application services may vary and which remain stable.
   - ✓ Following are ex: of system aspects that often vary
     - o Real time constraints
     - o Transaction protocols
     - o Inter Process Communication mechanism
     - o Behavior in case of exceptions
     - o Algorithm for application services.
3. **Identify structural aspects** of the system, which when changed, should not affect the implementation of the base level.
4. **Identify system services that support both the variation of application services identified**
   In step 2 and the independence of structural details identified in step 3
   Eg: for system services are
   - ✓ Resource allocation
   - ✓ Garbage allocation
   - ✓ Page swapping
   - ✓ Object creation.
5. **Define the meta objects**
   - ✓ For every aspect identified in 3 previous steps, define appropriate Meta objects.

- ✓ Encapsulating behavior is supported by several domain independent design patterns, such as objectifier strategy, bridge, visitor and abstract factory.

6. Define the MOP

- ✓ There are two options for implementing the MOP.
  - Integrate it with Meta objects. Every Meta object provides those functions of the MOP that operate on it.
  - Implement the MOP as a separate component.
- ✓ Robustness is a major concern when implementing the MOP. Errors in change specifications should be detected wherever possible.

7. Define the base level

- ✓ Implement the functional core and user interface of the system according to the analysis model developed in step 1.
- ✓ Use Meta objects to keep the base level extensible and adaptable. Connect every base level component with Meta objects that provide system information on which they depend, such as type information etc.
- ✓ Provide base level components with functions for maintaining the relationships with their associated Meta objects. The MOP must be able to modify every relationship b/w base level and Meta level.

Example resolved:

Unlike languages like CLOS or Smalltalk. C++ does not support reflection very well-only the standard class type_info provides reflective capabilities: we can identify and compare types. One solution for providing extended type information is to include a special step in the compilation process. In this, we collect type information from the source files of the application, generate code for instantiating the' metaobjects, and link this code with the application. Similarly, the 'object creator' metaobject is generated. Users specify code for instantiating an 'empty' object of every type, and the toolkit generates the code for the metaobject. Some parts of the system are compiler-dependent, such as offset and size calculation.

Variants:

- ♥ *Reflection with several Meta levels*
  Sometimes metaobjects depend on each other. Such a software system has an infinite number of meta levels in which each meta level is controlled by a higher one, and where each meta level has its own metaobject protocol. In practice, most existing reflective software comprises only one or two meta levels.

Known uses:

- ❖ **CLOS.**
  This is the classic example of a reflective programming language [Kee89]. In CLOS, operations defined for objects are called generic functions, and their processing is referred to as generic function invocation. Generic function invocation is divided into three phases:
  - ♣ The system first determines the methods that are applicable to a given invocation.
  - ♣ It then sorts the applicable methods in decreasing order of precedence.
  - ♣ The system finally sequences the execution of the list of applicable methods.
- ❖ **MIP**
  It is a run-time type information system for C++. The functionality of MIP is separated into four layers:
  - ♣ The first layer includes information and functionality that allows software to identify and compare types.
  - ♣ The second layer provides more detailed information about the type system of an application.
  - ♣ The third layer provides information about relative addresses of data members, and offers functions for creating 'empty' objects of user-defined types.
  - ♣ The fourth layer provides full type information, such as that about friends of a class, protection of data members, or argument and return types of function members.
- ❖ **PGen**
  It allows an application to store and read arbitrary C++ object structures.

- **NEDIS**
  NEDIS includes a meta level called run-time data dictionary. It provides the following services and system information:
  - Properties for certain attributes of classes, such as their allowed value ranges.
  - Functions for checking attribute values against their required properties.
  - Default values for attributes of classes, used to initialize new objects.
  - Functions specifying the behavior of the system in the event of errors
  - Country-specific functionality, for example for tax calculation.
  - Information about the 'look and feel' of the software, such as the layout of input masks or the language to be used in the user interface.
- **OLE 2.0**
  It provides functionality for exposing and accessing type information about OLE objects and their interfaces.

Consequences:

The reflection architecture provides the following *Benefits:*

- **No explicit modification of source code:**
  You just specify a change by calling function of the MOP.
- **Changing a software system is easy**
  MOP provides a safe and uniform mechanism for changing s/w. it hides all specific techniques such as use of visitors, factories from user.
- **Support for many kind of change:**
  Because Meta objects can encapsulate every aspect of system behavior, state and structure.

The reflection architecture also has *Liabilities:*

- **Modifications at meta level may cause damage:**
  - ✓ Incorrect modifications from users cause serious damage to the s/w or its environment. Ex: changing a database schema without suspending the execution of objects in the applications that use it or passing code to the MOP that includes semantic errors
  - ✓ Robustness of MOP is therefore of great importance.
- **Increased number of components:**
  It includes more Meta objects than base level components.
- **Lower efficiency:**
  Slower than non reflective systems because of complex relnp b/w base and meta level.
- **Not all possible changes to the software are supported**
  Ex: changes or extensions to base level code.
- **Not all languages support reflection**
  Difficult to implement in C ++

UNIT 6 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the benefits and liabilities of microkernel pattern	Dec 09	10
2	Enumerate the implementation steps of reflection pattern	Dec 09	10
3	What are the steps involved in implementing the microkernel system?	June 10	12
4	What are the benefits and liabilities of reflection architecture pattern?	June 10	8
5	List and explain the participating components of a microkernel pattern	Dec 10	10
6	Explain the known uses of reflection pattern	Dec 10	10
7	Discuss the benefits and liabilities of microkernel pattern	June 11	10
8	Give the detailed explanation on the different known applications offered by the reflection pattern	June 11	10
9	Explain in brief, the components comprising the structure of microkernel architectural pattern	Dec 11	10
10	With an example, explain when the reflection architectural pattern is used. What are its benefits?	Dec 11	10
11	What are the steps involved in implementing the microkernel system?	June 12	8
12	Explain the known uses of reflection pattern	June 12	6
13	Explain the advantages and disadvantages of reflection architectural pattern	June 12	6

UNIT 7
SOME DESIGN PATTERNS

INTRODUCTION:

Design patterns are medium scale patterns. They are smaller in scale than architectural patterns, but are at a higher level than the programming language specific idioms.
We group design patterns into categories of related patterns, in the same way as we did for architectural patterns:

- **Structural Decomposition**
  This category includes patterns that support a suitable decomposition of subsystems and complex components into co-operating parts. The Whole-Part pattern is the most general pattern we are aware of in this category.
- **Organization of Work.**
  This category comprises patterns that define how components collaborate together to solve a complex problem. We describe the Master-Slave pattern, which helps you to organize the computation of services for which fault tolerance or computational accuracy is required.
- **Access Control.**
  Such patterns guard and control access to services or components. We describe the Proxy pattern here.
- **Management.**
  This category includes patterns for handling homogenous collections of objects, services and components in their entirety. We describe two patterns: the Command Processor pattern addresses the management and scheduling of user commands, while the View Handler pattern describes how to manage views in a software system.
- **Communication.**
  Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern deals with peer-to-peer communication, while the Client Dispatcher-Server pattern describes location-transparent communication in a Client-Server structure.

STRUCTURAL DECOMPOSITION:

Subsystems and complex components are handled more easily if structured into smaller independent components, rather than remaining as monolithic block of code.
We discuss whole-part design pattern that supports the structural decomposition of the component.

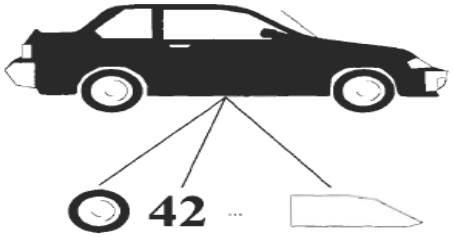
WHOLE-PART

Whole-part design pattern helps with the aggregation of components that together form a semantic unit.
An aggregate component, the whole, encapsulates its constituent components, the parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the parts is not possible.

Example:

A computer-aided design (CAD) system for 2-D and 3-D modelling allows engineers to design graphical objects interactively. For example, a car object aggregates several smaller objects such as wheels and windows, which

themselves may be composed of even smaller objects such as circles and polygons. It is the responsibility of the car object to implement functionality that operates on the car as a whole, such as rotating or drawing.



Context:
Implementing aggregate objects

Problem:

- In almost every software system objects that are composed of other objects exists. Such aggregate objects do not represent loosely-coupled set of components. Instead, they form units that are more than just a mere collection of their parts.
- The combination of the parts makes new behavior emerge- such behavior is called emergent behavior.
- We need to balance following forces when modeling such structures;
  - ✓ A complex object should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of the constituent objects in other types of aggregate.
  - ✓ Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts.

Solution:

- Introduce a component that encapsulates smaller objects and prevents clients from accessing theses constitutes parts directly.
- Define an interface for the aggregate that is the only means of access to the functionalities of the encapsulated objects allowing the aggregate to appear as semantic unit.
- The principle of whole-part pattern is applicable to the organization of three types of relationship
  - ✓ An *assembly-parts* relationship which differentiation b/w a product and its parts or subassemblies.
  - ✓ A *container-contents* relationship, in which the aggregated object represents a container.
  - ✓ The *collection-members* relationship, which helps to group similar objects.

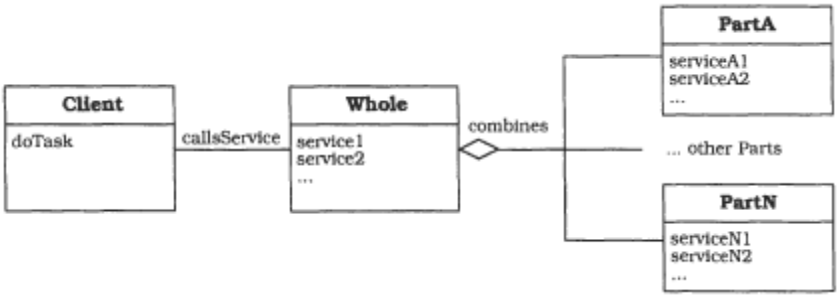
Structure:
The Whole-Part pattern introduces two types of participant:

- ❖ **Whole**
  - Whole object represents an aggregation of smaller objects, which we call parts.
  - It forms a semantic grouping of its parts in that it co ordinates and organizes their collaboration.
  - Some methods of whole may be just place holder for specific part services when such a method is invoked the whole only calls the relevant part services, and returns the result to the client.
- ❖ **Part**
  - Each part object is embedded in exactly one whole. Two or more parts cannot share the same part.
  - Each part is created and destroyed within the life span of the whole.

Class Whole	Collaborators • Part
Responsibility • Aggregates several smaller objects. • Provides services built on top of part objects. • Acts as a wrapper around its constituent parts.	

Class Part	Collaborators -
Responsibility • Represents a particular object and its services.	

Static relationship between whole and its part are illustrated in the OMT diagram below



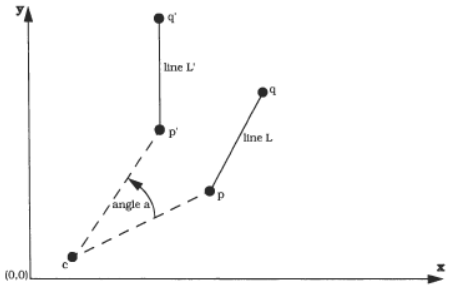
Dynamics:

The following scenario illustrates the behavior of a Whole-Part structure. We use the two-dimensional rotation of a line within a CAD system as an example. The line acts as a Whole object that contains two points p and **q** as Parts. A client asks the line object to rotate around the point c and passes the rotation angle as an argument.

The rotation of a point **p** around a center c with an angle a can be calculated using the following formula:

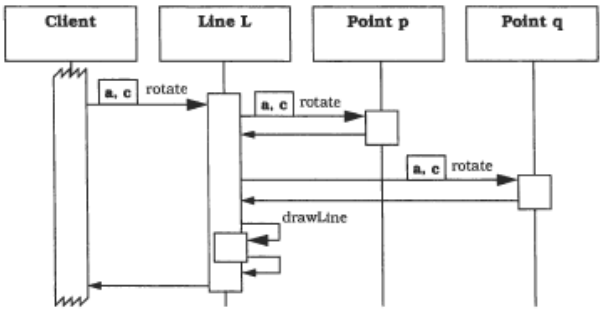
$$\vec{p}' = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix} \cdot (\vec{p} - \vec{c}) + \vec{c}$$

In the diagram below the rotation of the line given by the points p and q is illustrated.



The scenario consists of four phases:

- A client invokes the rotate method of the line L and passes the angle a and the rotation center c as arguments.
- The line L calls the rotate method of the point p.
- The line L calls the rotate method of the point q.
- The line L redraws itself using the new positions of p_1 and q_1 as endpoints.



Implementation:

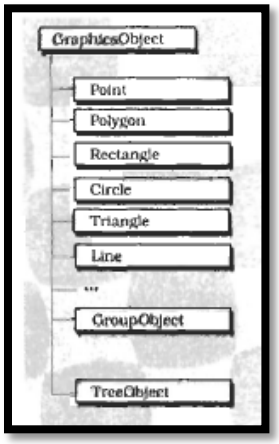
1. **Design the public interface of the whole**
   - Analyze the functionality the whole must offer to its clients.
   - Only consider the clients view point in this step.
   - Think of the whole as an atomic component that is not structured into parts.
2. **Separate the whole into parts, or synthesize it from existing ones.**
   - There are two approaches to assembling the parts either assemble a whole 'bottom-up' from existing parts, or decompose it 'top-down' into smaller parts.
   - Mixtures of both approaches is often applied
3. **If you follow a bottom up approach**, use existing parts from component libraries or class libraries and specify their collaboration.
4. **If you follow a top down approach, partition the Wholes services into smaller collaborating services** and map these collaborating services to separate parts.
5. **Specify the services of the whole in terms of services of the parts.**
   Decide whether all part services are called only by their whole, or if parts may also call each other.
   Two are two possible ways to call a Part service:
   @ If a client request is forwarded to a Part service, the Part does not use any knowledge about the execution context of the Whole, relying on its own environment instead.
   @ A delegation approach requires the Whole to pass its own context information to the Part.
6. **Implement the parts**
   If parts are whole-part structures themselves, design them recursively starting with step1 . if not reuse existing parts from a library.
7. **Implement the whole**
   Implement services that depend on part objects by invoking their services from the whole.

Variants:

- **Shared parts:**
  This variant relaxes the restriction that each Part must be associated with exactly one Whole by allowing several Wholes to share the same Part.
- **Assembly parts**
  In this variant the Whole may be an object that represents an assembly of smaller objects.
- **Container contents**
  In this variant a container is responsible for maintaining differing contents
- **Collection members**
  This variant is a specialization of Container-Contents, in that the Part objects all have the same type.
- **Composite pattern**
  It is applicable to Whole-Part hierarchies in which the Wholes and their Parts can be treated uniformly-that is, in which both implement the same abstract interface.

Example resolved:

In our CAD system we decide to define a Java package that provides the basic functionality for graphical objects. The class library consists of atomic objects such as circles or lines that the user can combine to form more complex entities. We implement these classes directly instead of using the standard Java package awt (Abstract Windowing Toolkit) because awt does not offer all the functionality we need.



Known uses:

- The key abstractions of many **object-oriented applications** follow the Whole-Part pattern.
- Most **object-oriented class libraries** provide collection classes such as lists, sets and maps. These classes implement the Collection- Member and Container-Contents variants.
- **Graphical user interface toolkits** such as Fresco or **ET++** use the Composite variant of the Whole-Part pattern.

Consequences:

The whole-part pattern offers several *Benefits:*

- **Changeability of parts:**
  Part implementations may even be completely exchanged without any need to modify other parts or clients.
- **Separation of concerns:**
  Each concern is implemented by a separate part.
- **Reusability in two aspects:**
  - Parts of a whole can be reused in other aggregate objects
  - Encapsulation of parts within a whole prevents clients from 'scattering' the use of part objects all over its source code.

The whole-part pattern suffers from the following *Liabilities:*

- **Lower efficiency through indirection**
  Since the Whole builds a wrapper around its Parts, it introduces an additional level of indirection between a client request and the Part that fulfils it.
- **Complexity of decomposition into parts.**
  An appropriate composition of a Whole from different Parts is often hard to find, especially when a bottom-up approach is applied.

ORGANIZATION OF WORK

The implementation of complex services is often solved by several components in co operation. To organize work optimally within such structures you need to consider several aspects.
We describe one pattern for organizing work within a system → maser-slave pattern.

MASTER-SLAVE

*The **Master-Slave** design pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results these slaves return.*

Example:

Travelling salesman problem

Context:

Portioning work into semantically identical subtasks

Problem:

Divide and conquer: here work is partitioned into several equal subtasks that are processed independently. The result of the whole calculation is computed from the results provided by each partial process.

Several forces arise when implementing such a structure

- Clients should not be aware that the calculation is based on the 'divide and conquer' principle.
- Neither clients nor the processing of subtasks should depend on the algorithms for partitioning work and assembling the final result.
- It can be helpful to use different but semantically identical implementations for processing subtasks.
- Processing of subtasks sometimes need co ordination for ex. In simulation applications using the finite element method.

Solution:

- Introduce a co ordination instance b/w clients of the service and the processing of individual subtasks.
- A master component divides work into equal subtasks, delegates these subtasks to several independent but semantically identical slave components and computes a final result from the partial results the slaves return.
- The general principle is found in three application areas
  - Fault tolerance → Failure of service executions can be detected and handled
  - Parallel computing → A complex task is divided into a fixed number of identical sub-tasks that are executed in parallel.
  - Computational accuracy → Inaccurate results can be detected and handled.

Structure:

- **Master component:**
- ✓ Provides the service that can be solved by applying the 'divide and conquer' principle.
- ✓ It implements functions for partitioning work into several equal subtasks, starting and controlling their processing and computing a final result from all the results obtained.
- ✓ It also maintains references to all slaves instances to which it delegates the processing of subtasks.

❖ **Slave component:**

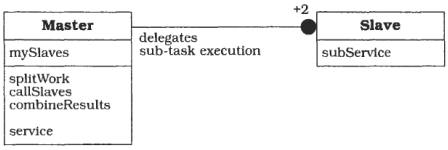
✓ Provides a sub-service that can process the subtasks defined by the master

✓ There are at least two instances of the slave component connected to the master.

Class Master	***Collaborators*** • Slave
Responsibility • Partitions work among several slave components • Starts the execution of slaves • Computes a result from the sub-results the slaves return.	

Class Slave	***Collaborators*** -
Responsibility • Implements the sub-service used by the master.	

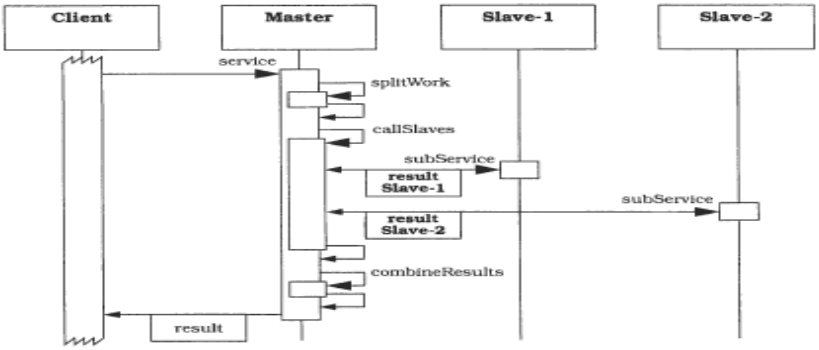
The structure defined by the Master-Slave pattern is illustrated by the following OMT diagram.



Dynamics:

The scenario comprises six phases:

- A client requests a service from the master.
- The master partitions the task into several equal sub-tasks.
- The master delegates the execution of these sub-tasks to several slave instances, starts their execution and waits for the results they return.
- The slaves perform the processing of the sub-tasks and return the results of their computation back to the master.
- The master computes a final result for the whole task from the partial results received from the slaves.
- The master returns this result to the client.



Implementation:

1. **Divide work:**
   Specify how the computation of the task can be split into a set equal sub tasks.
   Identify the sub services that are necessary to process a subtask.
2. **Combine sub-task results**
   Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.
3. **Specify co operation between master and slaves**
   - Define an interface for the subservice identified in step1 it will be implemented by the slave and used by the master to delegate the processing of individual subtask.

- One option for passing subtasks from the master to the slaves is to include them as a parameter when invoking the subservice.
  Another option is to define a repository where the master puts sub tasks and the slaves fetch them.

4. **Implement the slave component**s according to the specifications developed in previous step.
5. **Implement the master** according to the specifications developed in step 1 to 3
   - There are two options for dividing a task into subtasks.
     - The first is to split work into a fixed number of subtasks.
     - The second option is to define as many subtasks as necessary or possible.
   - Use strategy pattern to support dynamic exchange and variations of algorithms for subdividing a task.

Variants:

- There are 3 application areas for master slave pattern.
  - **Master-slave for fault tolerance**
    In this variant the master just delegates the execution of a service to a fixed number of replicated implementations, each represented by a slave.
  - **Master-slave for parallel computation**
    In this variant the master divides a complex task into a number of identical sub-tasks, each of which is executed in parallel by a separate slave.
  - **Master-slave for computational concurrency.**
    In this variant the execution of a service is delegated to at least three different implementations, each of which is a separate slave.

- Other variants
  - **Slaves as processes**
    To handle slaves located in separate processes, you can extend the original Master-Slave structure with two additional components
  - **Slaves as threads**
    In this variant the master creates the threads, launches the slaves, and waits for all threads to complete before continuing with its own computation.
  - **Master-slave with slave co ordination**
    In this case the computation of all slaves must be regularly suspended for each slave to coordinate itself with the slaves on which it depends, after which the slaves resume their individual computation.

Known uses:

- **Matrix multiplication**. Each row in the product matrix can be computed by a separate slave.
- **Transform-coding** an image, for example in computing the discrete cosine transform (DCT) of every 8 x 8 pixel block in an image. Each block can be computed by a separate slave.
- Computing the **cross-correlation** of two signals
- The **Workpool model** applies the master-slave pattern to implement process control for parallel computing
- The concept of **Gaggles** builds upon the principles of the Master-Slave pattern to handle 'plurality' in an object-oriented software system. **A *gaggle*** represents a set of replicated service objects.

Consequences:

The Master-Slave design pattern provides several *Benefits:*

- **Exchangeability and extensibility**
  By providing an abstract slave class, it is possible to exchange existing slave implementations or add new ones without major changes to the master.
- **Separation of concerns**

The introduction of the master separates slave and client code from the code for partitioning work, delegating work to slaves, collecting the results from the slaves, computing the final result and handling slave failure or inaccurate slave results.

- **Efficiency**
  The Master-Slave pattern for parallel computation enables you to speed up the performance of computing a particular service when implemented carefully

The Master-Slave design pattern has certain *Liabilities:*

- **Feasibility**
  It is not always feasible
- **Machine dependency**
  It depends on the architecture of the m/c on which the program runs. This may decrease the changeability and portability.
- **Hard to implement**
  Implementing Master-Slave is not easy, especially for parallel computation.
- **Portability**
  Master-Slave structures are difficult or impossible to transfer to other machines

ACCESS CONTROL

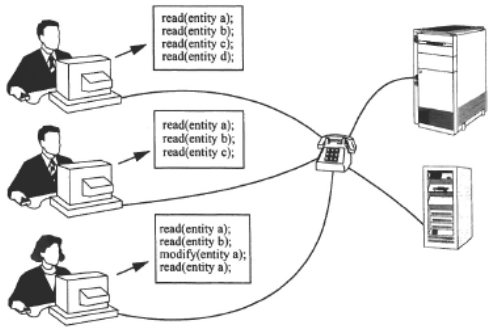
Sometimes a component or even a whole subsystem cannot or should not be accessible directly by its clients. Here we describe one design pattern that helps to protect access to a particular component: → *The proxy design pattern*

PROXY

Proxy design pattern makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a place holder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.

Example:

Company engineering staff regularly consults databases for information about material providers, available parts, blueprints, and so on. Every remote access may be costly, while many accesses are similar or identical and are repeated often. This situation clearly offers scope for optimization of access time and cost.



Context:

A client needs access to the services of another component direct access is technically possible, but may not be the best approach.

Problem:

It is often inappropriate to access a component directly.
A solution to such a design problem has to balance the following forces.

- Accessing the component should be runtime efficient, cost effective and safe for both the client and the component
- Access to component should be transparent and simple for the client. The client should particularly not have to change its calling behavior and syntax from that used to call any other direct access component.
- The client should be well aware of possible performance or financial penalties for accessing remote clients. Full transparency can obscure cost differences between services.

Solution:

- Let the client communicate with a representative rather than the component itself.
- This representative called a 'proxy' offers the interface of the component but performs additional pre and post processing such as access control checking or making read only copies of the 'original'.

Structure:

❖ **Original**

- Implements a particular service

❖ **Client**

- Responsible for specific task
- To do this, it involves the functionality of the original in an indirect way by accessing the proxy.

❖ **Proxy**

- Offers same interface as the original, and ensures correct access to the original.
- To achieve this, the proxy maintains a reference to the original it represents.
- Usually there is one-to-one relationship b/w the proxy and the original.

❖ **Abstract original**

- Provides the interface implemented by the proxy and the original. i.e, serves as abstract base class for the proxy and the original.

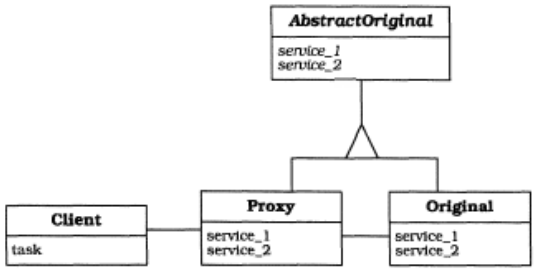
Class: Client	Collaborators: • Proxy
Responsibilities: • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task.	

Class: AbstractOriginal	Collaborators: -
Responsibilities: • Serves as an abstract base class for the proxy and the original.	

Class: Proxy	Collaborator: • Original
Responsibilities: • Provides the interface of the original to clients. • Ensures a safe, efficient and correct access to the original.	

Class: Original	Collaborators: -
Responsibilities: • Implements a particular service.	

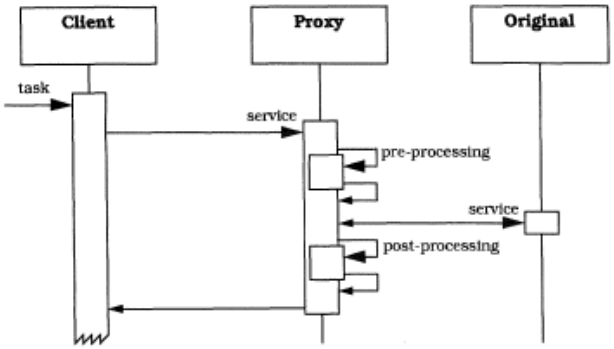
The following **OMT** diagram shows the relationships between the classes graphically:



Dynamics:

The following diagram shows a typical dynamic scenario of a Proxy structure.

- ♣ While working on its task the client asks the proxy to carry out a service.
- ♣ The proxy receives the incoming service request and pre-processes it.
- ♣ If the proxy has to consult the original to fulfill the request, it forwards the request to the original using the proper communication protocols and security measures.
- ♣ The original accepts the request and fulfills it. It sends the response back to the proxy.
- ♣ The proxy receives the response. Before or after transferring it to the client it may carry out additional post-processing actions such as caching the result, calling the destructor of the original or releasing a lock on a resource.



Implementation:

1. **Identify all responsibilities for dealing with access control to a component**
   Attach these responsibilities to a separate component the proxy.
2. **If possible introduce an abstract base class that specifies the common parts of the interfaces of both the proxy and the original.**
   Derive the proxy and the original from this abstract base.
3. **Implement the proxy's functions**
   To this end check the roles specified in the first step
4. **Free the original and its client** from responsibilities that have migrated into the proxy.
5. **Associate the proxy and the original** by giving the proxy a handle to the original. This handle may be a pointer a reference an address an identifier, a socket, a port, and so on.
6. **Remove all direct relationships between the original and its client**
   Replace them by analogous relationships to the proxy.

Variants:

- ➢ **Remote proxy:**
  Clients of remote components should be scheduled from network addresses and IPC protocols.
- ➢ **Protection proxy:**
  Components must be protected from unauthorized access.
- ➢ **Cache proxy:**
  Multiple local clients can share results from remote components.
- ➢ **Synchronization proxy:**
  Multiple simultaneous accesses to a component must be synchronized.
- ➢ **Counting proxy:**
  Accidental deletion of components must be prevented or usage statistics collected
- ➢ **Virtual proxy:**
  Processing or loading a component is costly while partial information about the component may be sufficient.
- ➢ **Firewall proxy:**

Local clients should be protected from the outside world.

Known uses:

- **NeXT STEP**
  The Proxy pattern is used in the NeXTSTEP operating system to provide local stubs for remote objects. Proxies are created by a special server on the first access to the remote object.
- **OMG-COBRA**
  It uses the Proxy pattern for two purposes. So called 'client-stubs', or IDL-stubs, guard clients against the concrete implementation of their servers and the Object Request Broker.
- **OLE**
  In Microsoft OLE servers may be implemented as libraries dynamically linked to the address space of the client, or as separate processes. Proxies are used to hide whether a particular server is local or remote from a client.
- **WWW proxy**
  It gives people inside the firewall concurrent access to the outside world. Efficiency is increased by caching recently transferred files.
- **Orbix**
  It is a concrete OMG-CORBA implementation, uses remote proxies. A client can bind to an original by specifying its unique identifier.

Consequences:

The Proxy pattern provides the following *Benefits:*

- **Enhanced efficiency and lower cost**
  The Virtual Proxy variant helps to implement a 'load-on-demand' strategy. This allows you to avoid unnecessary loads from disk and usually speeds up your application
- **Decoupling clients from the location of server components**
  By putting all location information and addressing functionality into a Remote Proxy variant, clients are not affected by migration of servers or changes in the networking infrastructure. This allows client code to become more stable and reusable.
- **Separation of housekeeping code from functionality.**
  A proxy relieves the client of burdens that do not inherently belong to the task the client is to perform.

The Proxy pattern has the following *Liabilities:*

- **Less efficiency due to indirection**
  All proxies introduce an additional layer of indirection.
- **Over kill via sophisticated strategies**
  Be careful with intricate strategies for caching or loading on demand they do not always pay.

UNIT 7 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1.	Give the structure of whole port design pattern with CRC	Dec 09	5
2.	What are the applications of master slave pattern	Dec 09	10
3.	What are the variants of proxy pattern?	Dec 09	5
4.	Discuss the five steps implementation of master slave pattern	June 10	10
5.	Define proxy design pattern. Discuss the benefits and liabilities of the same	June 10	10
6.	Briefly explain the benefits of master slave design pattern	Dec 10	6
7.	List and explain the steps to implement a whole-part structure	Dec 10	8
8.	With a neat sketch, explain the typical dynamic scenario of a proxy structure	Dec 10	6
9.	Enumerate with explanation the different steps, which constitute the implementation of the whole part structure for a CAD system for 2D modeling.	June 11	14
10.	Briefly comment on the different steps carried out to realize the implementation of the proxy pattern	June 11	6
11.	Explain the variants of whole-part design pattern, in brief	Dec 11	10
12.	Explain the dynamics part of master slave design pattern	Dec 11	8
13.	Mention any two benefits of proxy design pattern	Dec 11	2
14.	Briefly explain the benefits of master slave design pattern	June 12	6
15.	What are the variants of proxy pattern?	June 12	6
16.	List and explain the steps to implement a whole-part structure	June 12	8

UNIT 8
DESIGNING AND DOCUMENTING SOFTWARE ARCHITECTURE

CHAPTER 7:
DESIGNING THE ARCHITECTURE

ARCHITECTURE IN THE LIFE CYCLE

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

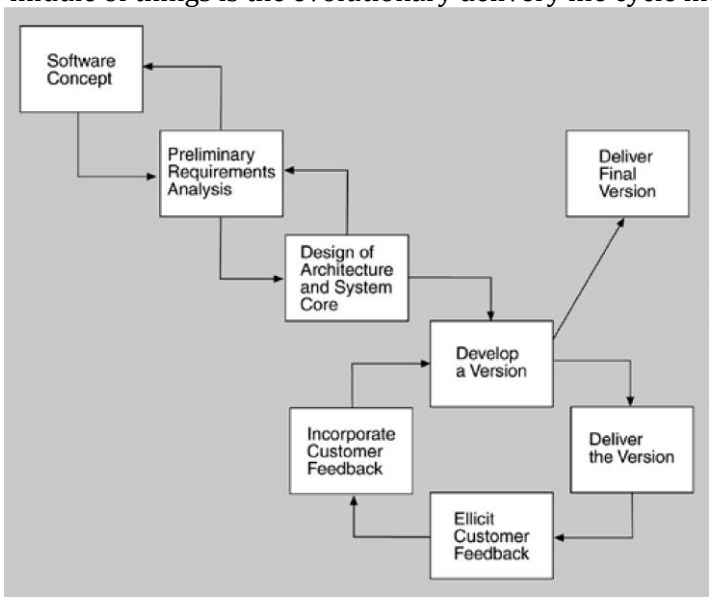


Figure 7.1. Evolutionary Delivery Life Cycle

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

WHEN CAN I BEGIN DESIGNING?

- The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. Clearly, you cannot begin the design until you have some idea of the system requirements. On the other hand, it does not take many requirements in order for design to begin.
- An architecture is "shaped" by some collection of functional, quality, and business requirements. We call these shaping requirements *architectural drivers* and we see examples of them in our case studies like modifiability, performance requirements availability requirements and so on.
- To determine the architectural drivers, identify the highest priority business goals. There should be relatively few of these. Turn these business goals into quality scenarios or use cases.

7.2 DESIGNING THE ARCHITECTURE

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

ATTRIBUTE DRIVEN DESIGN

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern.
The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate.

Garage door opener example

- Design a product line architecture for a garage door opener with a larger home information system the opener is responsible for raising and lowering the door via a switch, remote control, or the home information system. It is also possible to diagnose problems with the opener from within the home information system.
- **Input** to ADD: a set of requirements
  - Functional requirements as use cases
  - Constraints
  - Quality requirements expressed as system specific quality scenarios
- **Scenarios** for garage door system
  - Device and controls for opening and closing the door are different for the various products in the product line
  - The processor used in different products will differ
  - If an obstacle is (person or object) is detected by the garage door during descent, it must stop within 0.1 second
  - The garage door opener system needs to be accessible from the home information system for diagnosis and administration.
  - It should be possible to directly produce an architecture that reflects this protocol

ADD Steps:

Steps involved in attribute driven design (ADD)

1. *Choose the module to decompose*
   - Start with entire system
   - Inputs for this module need to be available
   - Constraints, functional and quality requirements
2. *Refine the module*
   a) Choose architectural drivers relevant to this decomposition
   b) Choose architectural pattern that satisfies these drivers
   c) Instantiate modules and allocate functionality from use cases representing using multiple views
   d) Define interfaces of child modules
   e) Verify and refine use cases and quality scenarios
3. *Repeat for every module that needs further decomposition*

Discussion of the above steps in more detail:

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

2. **Refine the module**

1. Choose Architectural Drivers:

- choose the architectural drivers from the quality scenarios and functional requirements
- The drivers will be among the top priority requirements for the module.
- In the garage system, the 4 scenarios were architectural drivers,
- By examining them, we see
  - Real-time performance requirement
  - Modifiability requirement to support product line
- Requirements are not treated as equals
- Less important requirements are satisfied within constraints obtained by satisfying more important requirements
- This is a difference of ADD from other architecture design methods

2. Choose Architectural Pattern

- For each quality requirement there are identifiable tactics and then identifiable patterns that implement these tactics.
- The goal of this step is to establish an overall architectural pattern for the module
- The pattern needs to satisfy the architectural pattern for the module tactics selected to satisfy the drivers
- Two factors involved in selecting tactics:
  - ✓ Architectural drivers themselves
  - ✓ Side effects of the pattern implementing the tactic on other requirements
- This yields the following tactics:
  - *Semantic coherence* and *information hiding.* Separate responsibilities dealing with the user interface, communication, and sensors into their own modules.
  - *Increase computational efficiency.* The performance-critical computations should be made as efficient as possible.
  - *Schedule wisely.* The performance-critical computations should be scheduled to ensure the achievement of the timing deadline.

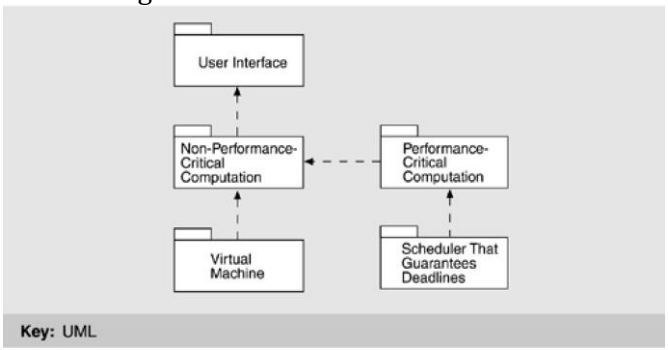


Figure 7.2. Architectural pattern that utilizes tactics to achieve garage door drivers

3. Instantiate Modules And Allocate Functionality Using Multiple Views

- *Instantiate modules*

  The non-performance-critical module of Figure 7.2 becomes instantiated as diagnosis and raising/lowering door modules in Figure 7.3. We also identify several responsibilities of the virtual machine: communication and sensor reading and actuator control. This yields two instances of the virtual machine that are also shown in Figure 7.3.

- *Allocate functionality*

  Assigning responsibilities to the children in a decomposition also leads to the discovery of necessary information exchange. At this point in the design, it is not important to define how the information is exchanged. Is the information pushed or pulled? Is it passed as a message or a call parameter? These are all questions that need to be answered later in the design process. At this point only the information itself and the producer and consumer roles are of interest

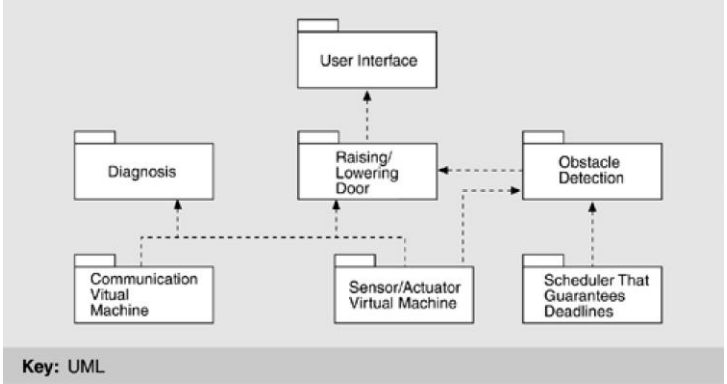


Figure 7.3. First-level decomposition of garage door opener

♥ *Represent the architecture with multiple views*
- ♣ *Module decomposition view*
- ♣ *Concurrency view*
  - Two users doing similar things at the same time
  - One user performing multiple activities simultaneously
  - Starting up the system
  - Shutting down the system
- ♣ *Deployment view*

4. Define Interfaces Of Child Modules
- It documents what this module provides to others.
- Analyzing the decomposition into the 3 views provides interaction information for the interface
  - *Module view:*
    - ✓ Producers/consumers relations
    - ✓ patterns of communication
  - *Concurrency view:*
    - ✓ Interactions among threads
    - ✓ Synchronization information
  - *Deployment view*
    - ✓ Hardware requirement
    - ✓ Timing requirements
    - ✓ Communication requirements

5. Verify And Refine Use Cases And Quality Scenarios As Constraints For The Child Modules
- *Functional requirements*

Using functional requirements to verify and refine
- ➢ Decomposing functional requirements assigns responsibilities to child modules
- ➢ We can use these responsibilities to generate use cases for the child module
  - ✓ *User interface:*
    - ♣ Handle user requests
    - ♣ Translate for raising/lowering module
    - ♣ Display responses
  - ✓ *Raising/lowering door module*
    - ♣ Control actuators to raise/lower door
    - ♣ Stop when completed opening or closing
  - ✓ *Obstacle detection:*
    - ♣ Recognize when object is detected
    - ♣ Stop or reverse the closing of the door
  - ✓ *Communication virtual machine*
    - ♣ Manage communication with house information system(HIS)

- ✓ *Scheduler*
  - ♣ Guarantee that deadlines are met when obstacle is detected
- ✓ *Sensor/actuator virtual machine*
  - ♣ Manage interactions with sensors/actuators
- ✓ *Diagnosis:*
  - ♣ Manage diagnosis interaction with HIS

- *Constraints:*
  - ✓ The decomposition satisfies the constraint
    - OS constraint-> satisfied if child module is OS
  - ♥ The constraint is satisfied by a single module
    - Constraint is inherited by the child module
  - ♥ The constraint is satisfied by a collection of child modules
    - E.g., using client and server modules to satisfy a communication constraint

- *Quality scenarios:*
  - Quality scenarios also need to be verified and assigned to child modules
  - A quality scenario may be satisfied by the decomposition itself, i.e, no additional impact on child modules
  - A quality scenario may be satisfied by the decomposition but generating constraints for the children
  - The decomposition may be "neutral" with respect to a quality scenario
  - A quality scenario may not be satisfied with the current decomposition

7.3 FORMING THE TEAM STRUCTURES

- ♥ Once the architecture is accepted we assign teams to work on different portions of the design and development.
- ♥ Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules and a work breakdown structure is created that reflects those teams.
- ♥ Each team then creates its own internal work practices.
- ♥ For large systems, the teams may belong to different subcontractors.
- ♥ Teams adopt "work practices' including
  - Team communication via website/bulletin boards
  - Naming conventions for files
  - Configuration/revision control system
  - Quality assurance and testing procedure

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

7.4 CREATING A SKELETAL SYSTEM

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

Once the elements providing the next increment of functionality have been chosen, you can employ the uses structure to tell you what additional software should be running correctly in the system to support that functionality. This process continues, growing larger and larger increments of the system, until it is all in place.

CHAPTER 9: DOCUMENTING SOFTWARE ARCHITECTURES

9.1 USES OF ARCHITECTURAL DOCUMENTATION

- Architecture documentation is both prescriptive and descriptive. That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true by recounting decisions already made about a system's design.
- All of this tells us that different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.
- One of the most fundamental rules for technical documentation in general, and software architecture documentation in particular, is to write from the point of view of the reader. Documentation that was easy to write but is not easy to read will not be used, and "easy to read" is in the eye of the beholder—or in this case, the stakeholder.
- Documentation facilitates that communication. Some examples of architectural stakeholders and the information they might expect to find in the documentation are given in Table 9.1.
- In addition, each stakeholders come in two varieties: seasoned and new. A new stakeholder will want information similar in content to what his seasoned counterpart wants, but in smaller and more introductory doses. Architecture documentation is a key means for educating people who need an overview: new developers, funding sponsors, visitors to the project, and so forth.

Table 9.1. Stakeholders and the Communication Needs Served by Architecture

Stakeholder	Use
Architect and requirements engineers who represent customer(s)	To negotiate and make tradeoffs among competing requirements
Architect and designers of constituent parts	To resolve resource contention and establish performance and other kinds of runtime resource consumption budgets
Implementors	To provide inviolable constraints (plus exploitable freedoms) on downstream development activities
Testers and integrators	To specify the correct black-box behavior of the pieces that must fit together
Maintainers	To reveal areas a prospective change will affect
Designers of other systems with which this one must interoperate	To define the set of operations provided and required, and the protocols for their operation
Quality attribute specialists	To provide the model that drives analytical tools such as rate-monotonic real-time schedulability analysis, simulations and simulation generators, theorem provers, verifiers, etc. These tools require information about resource consumption, scheduling policies, dependencies, and so forth. Architecture documentation must contain the information necessary to evaluate a variety of quality attributes such as security, performance, usability, availability, and modifiability. Analyses for each attributes have their own information needs.
Managers	To create development teams corresponding to work assignments identified, to plan and allocate project resources, and to track progress by the various teams
Product line managers	To determine whether a potential new member of a product family is in or out of scope, and if out by how much
Quality assurance team	To provide a basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions

Source: Adapted from [Clements 03]

9.2 VIEWS

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

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

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

- Choosing the relevant views
- Documenting view
- Documenting information that applies to more than one view

9.3 CHOOSING THE RELEVANT VIEWS

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

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

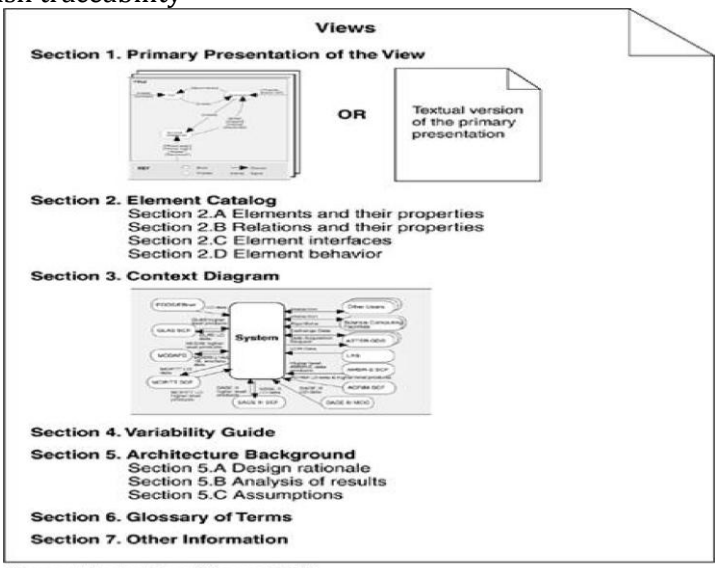
Table 9.2. Stakeholders and the Architecture Documentation They Might Find Most Useful

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

2. **Combine views:**
   The candidate view list from step 1 is likely to yield an impractically large number of views. To reduce the list to a manageable size, first look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger consistency. Next, look for the views that are good candidates to be combined- that is, a view that gives information from two or more views at once. For small and medium projects, the implementation view is often easily overlaid with the module decomposition view. The module decomposition view also pairs well with users or layered views. Finally, the deployment view usually combines well with whatever component-and-connector view shows the components that are allocated to hardware elements.
3. **Prioritize:**
   After step 2 you should have an appropriate set of views to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific project. But, remember that you don't have to complete one view before starting another. People can make progress with overview-level information, so a breadth-first approach is often the best. Also, some stakeholders' interests supersede others.

9.4 DOCUMENTING A VIEW

- **Primary presentation**- elements and their relationships, contains main information about these system , usually graphical or tabular.
- **Element catalog**- details of those elements and relations in the picture,
- **Context diagram**- how the system relates to its environment
- **Variability guide**- how to exercise any variation points a variability guide should include documentation about each point of variation in the architecture, including
  - The options among which a choice is to be made
  - The binding time of the option. Some choices are made at design time, some at build time, and others at runtime.
- **Architecture background** –why the design reflected in the view came to be? an architecture background includes
  - rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected
  - analysis results, which justify the design or explain what would have to change in the face of a modification
  - assumptions reflected in the design
- **Glossary of terms** used in the views, with a brief description of each.
- **Other information** includes management information such as authorship, configuration control data, and change histories. Or the architect might record references to specific sections of a requirements document to establish traceability



Source: Adapted from [Clements 03].

DOCUMENTING BEHAVIOR

- ★ Views present structural information about the system. However, structural information is not sufficient to allow reasoning about some system properties .behavior description add information that reveals the ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions.
- ★ Behavior can be documented either about an ensemble of elements working in concert. Exactly what to model will depend on the type of system being designed.
- ★ Different modeling techniques and notations are used depending on the type of analysis to be performed. In UML, sequence diagrams and state charts are examples of behavioral descriptions. These notations are widely used.

DOCUMENTING INTERFACES

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

1. **Interface identify**

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

2. Resources provided:

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

- ★ *Resource syntax* – this is the resource's signature
- ★ *Resource Semantics:*
  - Assignment of values of data
  - Changes in state
  - Events signaled or message sent
  - how other resources will behave differently in future
  - humanly observable results
- ★ *Resource Usage Restrictions*
  - initialization requirements
  - limit on number of actors using resource

3. Data type definitions:

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

4. Exception definitions:

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

5. Variability provided by the interface.

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

6. Quality attribute characteristics:

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

7. Element requirements:

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

8. Rationale and design issues:

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

9. Usage guide:

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

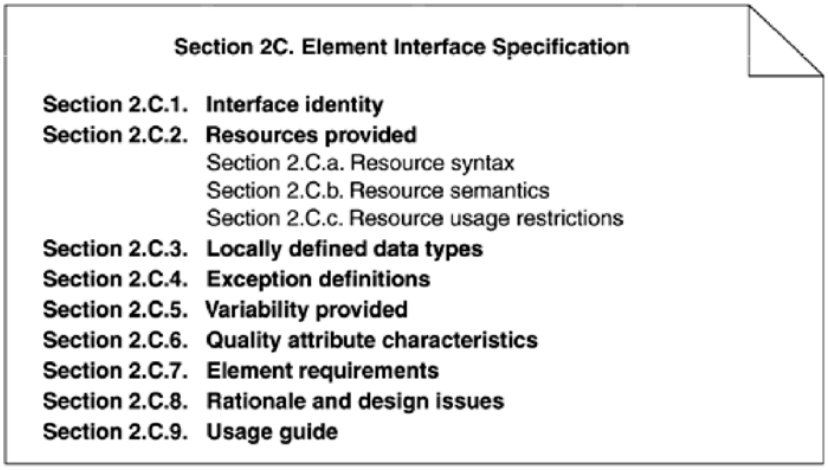
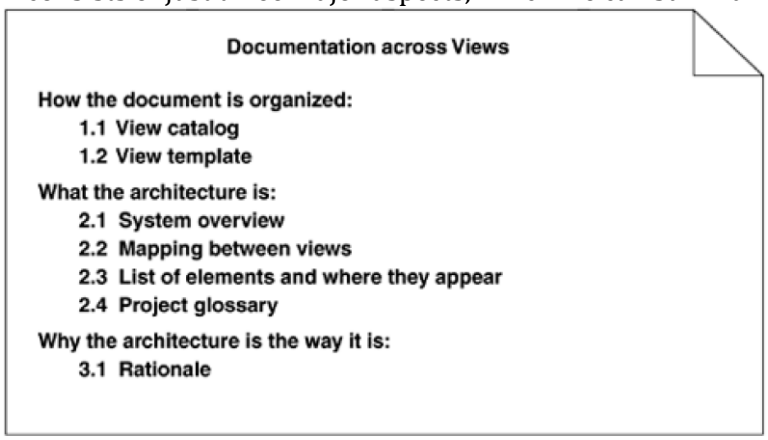


Figure 9.2. The nine parts of interface documentation

9.5 DOCUMENTATION ACROSS VIEWS

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



Source: Adapted from [Clements 03].

Figure 9.3. Summary of cross-view documentation

HOW THE DOCUMENTATION IS ORGANIZED TO SERVE A STAKEHOLDER

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

★ **View Catalog**

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

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

- The name of the view and what style it instantiates
- A description of the view's element types, relation types, and properties
- A description of what the view is for
- Management information about the view document, such as the latest version, the location of the view document, and the owner of the view document

★ **View Template**

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

WHAT THE ARCHITECTURE IS

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

♣ **System Overview**

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

♣ **Mapping between Views**

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

♣ **Element List**

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

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

WHY THE ARCHITECTURE IS THE WAY IT IS: RATIONALE

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

- The implications of system-wide design choices on meeting the requirements or satisfying constraints.
- The effect on the architecture when adding a foreseen new requirement or changing an existing one.
- The constraints on the developer in implementing a solution.
- Decision alternatives that were rejected.

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

UNIT 8 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1.	What are the three steps for choosing views for a project?	Dec 09	6
2.	Write a note on view catalog	Dec 09	4
3.	What are the options for representing connectors and systems in UML? ★ **OUT OF SYLLABUS**	Dec 09	10
4.	Explain with a neat diagram, the evolutionary delivery life cycle model	June 10	8
5.	What are the suggested standard organization points for interface documentation?	June 10	12
6.	List the steps of ADD	Dec 10	4
7.	Write a note on creating a skeletal system	Dec 10	6
8.	What are the uses of architectural documentation? Bring out the concept of view as applied to architectural documentation.	Dec 10	10
9.	Briefly explain the different steps performed while designing an architecture using the ADD method	June 11	10
10.	write short notes on: i)forming team structures ii)documenting across views iii)documenting interfaces	June 11	10
11.	Explain the steps involved in designing an architecture, using the attribute driven design	Dec 11	10
12.	"Architecture serves as a communication vehicle among stakeholders. Documentation facilitates that communication." Justify.	Dec 11	10
13.	List the steps of ADD method of architectural design	June 12	6
14.	Explain with a neat diagram, the evolutionary delivery life cycle model	June 12	6
15.	What are the suggested standard organization points for view documentation?	June 12	8