CMR
INSTITUTE OF
TECHNOLOGY

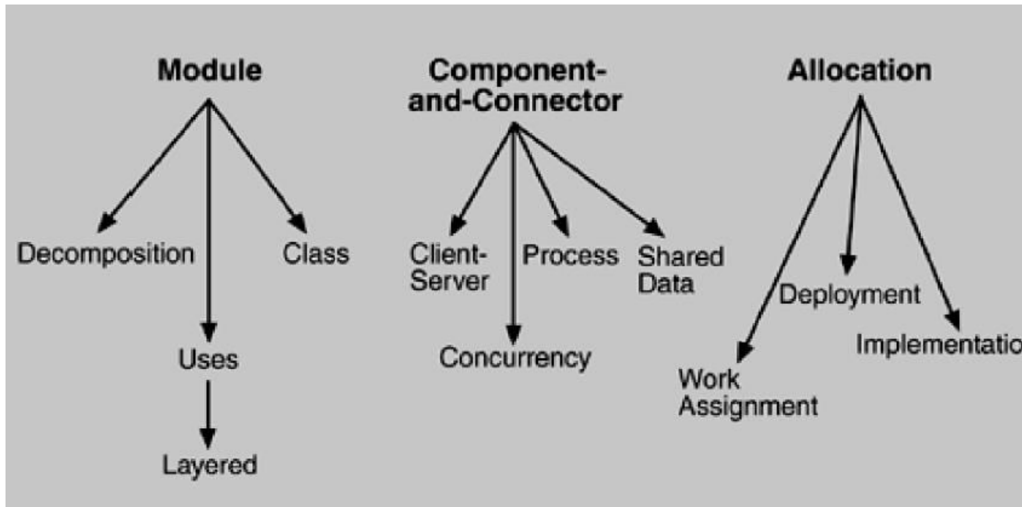
USN | | | | | | | | | |

Internal Assessment Test - I

Sub:	**Software Architectures**							Code:	10IS81
Date:	28 / 03 / 2017	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	ISE
Answer Any FIVE FULL Questions									

		Marks	OBE CO	OBE RBT
1 (a)	Explain common software architectures.	[10]	CO1	L1

Soln.



1. **Module**

Module-based structures include the following.

Decomposition: The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood. Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent design and eventual implementation. The decomposition structure provides a large part of the system's modifiability, by ensuring that likely changes fall within the purview of at most a few small modules.

Uses: The units of this important but overlooked structure are also modules procedures or resources on the interfaces of modules. The units are related by the uses relation. One unit uses another if the correctness of the first requires the presence of a correct version of the second. The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be easily extracted.

Layered: When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality. In a strictly layered structure, layer n may only use the services of layer n – 1.

Class, or generalization: The module units in this structure are called classes. The relation is "inherits-from" or "is-an-instance-of." This view supports reasoning about collections of similar behavior or capability and parameterized differences which are captured by subclassing. The class structure allows us to reason about re-use and the incremental addition of functionality.

2. Component-and-Connector

These structures include the following.

Process, or communicating processes: Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system. The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations. The relation in this is attachment, showing how the components and connectors are hooked together. The process structure is important in helping to engineer a system's execution performance and availability.

Concurrency: This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are "logical threads." A logical thread is a sequence of computation that can be allocated to a separate physical thread later in the design process. The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.

Shared data, or repository: This structure comprises components and connectors that create, store, and access persistent data. If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate. It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and data integrity.

Client-server: If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate. The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work. This is useful for separation of concerns, for physical distribution, and for load balancing.

3. Allocation

Allocation structures include the following.

Deployment. The deployment structure shows how software is assigned to hardware-processing and communication elements. The elements are software, hardware entities (processors), and communication pathways. Relations are "allocated-to," showing on which physical units the software elements reside, and "migrates-to," if the allocation is dynamic. This view allows an engineer to reason about performance, data integrity, availability, and security. It is of particular interest in distributed or parallel systems.

Implementation. This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments. This is critical for the management of development activities and build processes.

Work assignment. This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams. Having a work assignment structure as part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications. The architect will know the expertise required on each team. Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning them to a single team, rather than having them implemented by everyone who needs them.

2 (a) Explain Process Control view of cruise control. [8] CO3 L2

Soln. A control loop architecture is embedded in a physical system that involves continuing behavior, especially when the system is subject to external perturbations. The system is supposed to maintain constant speed in an automobile despite variations in terrain, vehicle load, air resistance, fuel quality, etc. To develop a control loop architecture for this system, we begin by identifying the essential system elements.

Computational elements

- *Process definition:* Since the cruise control software is driving a mechanical device (the engine), the details are not relevant. For our purposes, the process receives a **throttle** setting and turns the car's wheels. There may in fact be more computers involved, for example in controlling the fuel-injection system. From the standpoint of the cruise control subsystem, however, the process takes a throttle setting as input and controls the speed of the vehicle.
- *Control algorithm:* This algorithm models the **current speed** based on the **wheel pulses**, compares it to the **desired speed**, and changes the throttle setting. The **clock** input is needed to model **current speed** based on intervals between **wheel pulses**. Since the problem requires an exact **throttle** setting rather than a change, the current **throttle** setting must be maintained by the control algorithm. The policy decision about how much to change the **throttle** setting for a given discrepancy between **current speed** and **current speed** is localized in the control algorithm.

Data elements

- *Controlled variable:* For the cruise control, this is the **current speed** of the vehicle.
- *Manipulated variable:* For the cruise control, this is the **throttle** setting.
- *Set point:* The **desired speed** is set and modified by the **accelerator** input and the **increase/decrease speed** input, respectively. Several other inputs help control whether the cruise control is currently controlling the car: **System on/off**, **engine on/off**, **brake**, and **resume**. These interact: **resume** restores automatic control, but only if the entire system is on. These inputs are provided by the human driver (the operator, in process terms).
- *Sensor for controlled variable:* For cruise control, the current state is the **current speed**, which is modeled on data from a sensor that delivers **wheel pulses** using the **clock**. However, see discussion below about the accuracy of this model.

The restated control task was, "Whenever the system is active determine the desired speed and control the engine throttle setting to maintain that speed." Note that only the **current speed** output, the **wheel pulses** input, and the **throttle** manipulated variable are used outside the set point and active/inactive determination. This leads immediately to two subproblems: the interface with the driver, concerned with "whenever the system is active determine the desired speed" and the control loop, concerned with "control the engine throttle setting to maintain that speed."

The latter is the actual control problem; we'll examine it first. Figure 7 shows a suitable architecture for the control system. The first task is to model the **current speed** from the **wheel pulses**; the designer should validate this model carefully. The model could fail if the wheels spin; this could affect control in two ways. If the **wheel pulses** are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving. Even worse, if the **wheel pulses** are being taken from a non-drive wheel and the drive wheels are spinning, the controller will be misled to believe that the current speed is too slow and will continually increase the **throttle** setting. The designer should also consider whether the controller has full control authority over the process. In the case of cruise control, the only manipulated variable is the **throttle**: the brake is not available.

The controller also receives two inputs from the set point computation: the **active/inactive toggle**, which indicates whether the controller is in charge of the **throttle**, and the **desired speed**, which only needs to be valid when the vehicle is under automatic control. All this information should be either state or continuously updated data, so all lines in the diagram represent data flow. The controller is implemented as a continuously-evaluating function that matches the dataflow character of the inputs and outputs. Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines. Each of these has a parameter that controls how quickly and tightly the control tracks the set point; analysis of these characteristics is discussed in Section 3.4. As noted above, the engine is of little interest here; it might very well be implemented as an object or as a collection of objects.

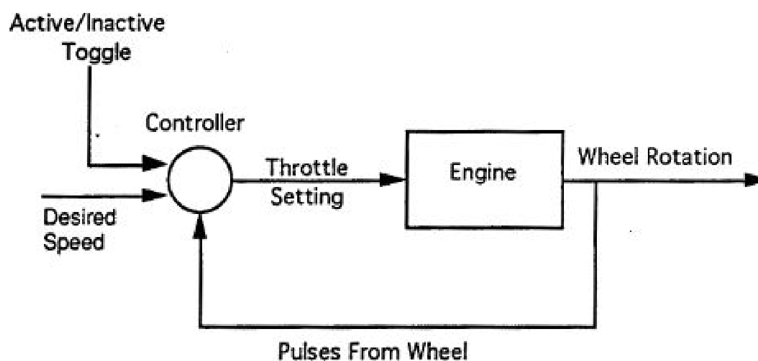


Figure 7: Control Architecture for Cruise Control

The set point calculation divides naturally into two parts: (a) determining whether or not the automatic system is active—in control of the throttle and (b) determining the **desired speed** for use by the controller in automatic mode.

The active/inactive toggle is triggered by a variety of events, so a state transition design is natural. It's shown in Figure 8. The system is completely off whenever the engine is off. Otherwise there are three inactive and one active states. In the first inactive state no set point has been established. In the other two, the previous set point must be remembered: When the driver accelerates to a speed greater than the set point, the manual accelerator controls the throttle through a direct linkage (note that this is the only use of the accelerator position in this design, and it relies on relative effect rather than absolute position); when the driver uses the brake the control system is inactivated until the resume signal is sent. The **active/inactive toggle** input of the control system is set to active exactly when this state machine is in state Active.

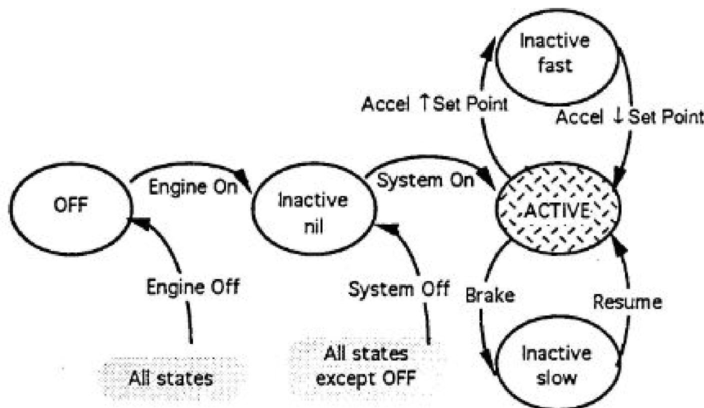


Figure 8: State Machine for Activation

(b)	Explain the problem statement of cruise control with input parameters.	[2]	CO3	L2

Soln. A cruise control system exists to maintain the speed of a car, even over varying terrain.

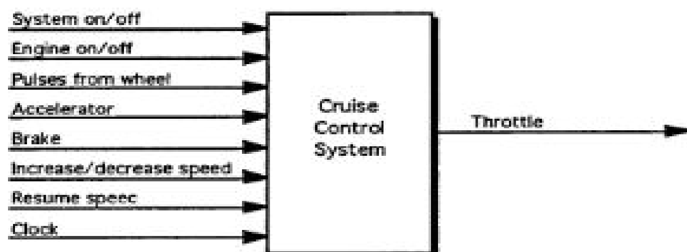


Figure 5: Booch block diagram for cruise control

There are several inputs:

- **System on/off** — If on, denotes that the cruise-control system should maintain the car speed.
- **Engine on/off** — If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
- **Pulses from wheel** — A pulse is sent for every revolution of the wheel.
- **Accelerator** — Indication of how far the accelerator has been pressed.
- **Brake** — On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
- **Increase/Decrease Speed** — Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
- **Resume** — Resume the last maintained speed; only applicable if the cruise-control system is on.
- **Clock** — Timing pulse every millisecond.

There is one output from the system:

- **Throttle** — Digital value for the engine throttle setting.

3 (a)	Explain the following Architectural Styles i) Event Based Implicit Invocation ii) Pipes and Filters	[10]	CO3	L2

Soln. i) Event Based Implicit Invocation

- In a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines.
- The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement ``implicitly" causes the invocation of procedures in other modules.
- For example, tools such as editors and variable monitors register for a debugger's breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke methods in those registered tools.
- These methods might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools (if any) are concerned with that event, or what they will do when that event is announced.
- The main invariant of this style is that announcers of events do not know which components will be affected by those events.
- Components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events.
- One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system.
- A second benefit is that implicit invocation eases system evolution. Components may be replaced by other components without affecting the interfaces of other components in the system.
- In contrast, in a system based on explicit invocation, whenever the identity of a that provides some system function is changed, all other modules that import that module must also be changed.
- The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it has no idea what other components will respond to it. Worse, even if it does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked. Nor can it know when they are finished.
- Another problem concerns exchange of data. Sometimes data can be passed with the event. But in other situations, event systems must rely on a shared repository

for interaction.

- Reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked.

ii) Pipes and Filters

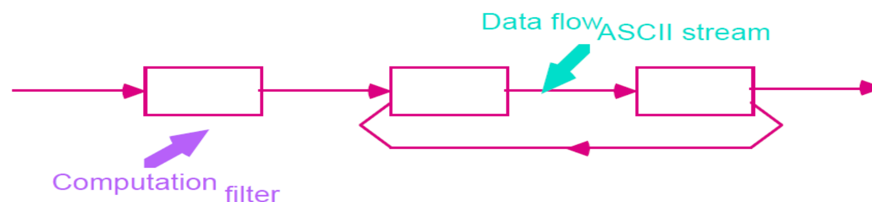


Figure 1: Pipes and Filters

- In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order.

- This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. Hence components are termed “filters”.

- The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”.

- Filters must be independent entities: in particular, they should not share state with other filters.

- Filters do not know the identity of their upstream and downstream filters.

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

- Common specializations of this style include pipelines, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters have a well-defined type.

- The best-known examples of pipe and filter architectures are programs written in the Unix shell.

- Pipe and filter systems have a number of nice properties. First, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters.

- Second, they support reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them.

- Third, systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones.

- Fourth, they permit certain kinds of specialized analysis, such as throughput and deadlock analysis.

- Finally, they naturally support concurrent execution. Each filter can be implemented as a separate task and potentially executed in parallel with other filters.

But these systems also have their disadvantages.

- First, pipe and filter systems often lead to a batch organization of processing. Although filters can process data incrementally, since filters are inherently independent, the designer is forced to think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character, pipe and filter systems are typically not good at handling interactive applications.

- Second, they may be hampered by having to maintain correspondences between two separate, but related streams.

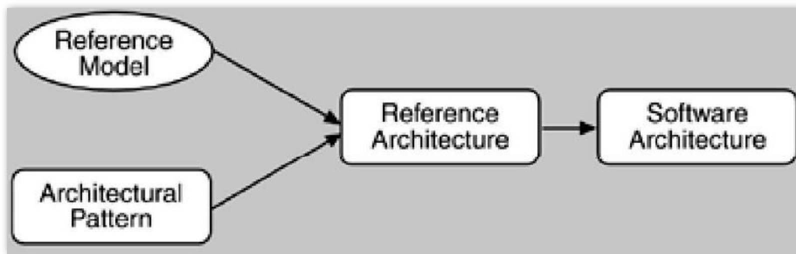
- Third, depending on the implementation, they may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparse its data. This, in turn, can lead both to loss of performance and to increased complexity in writing the filters themselves.

4 (a)	Explain the following concepts i) Architectural Pattern ii) Reference Model iii) Reference Architecture	[5]	CO1	L1

Soln. An ***architectural pattern*** is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of as a set of constraints on an architecture—on the element types and their patterns of interaction—and these constraints define a set or family of architectures that satisfy them. For example, client-server is a common architectural pattern. Client 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. Use of the term client-server implies only that multiple clients exist; the clients themselves are not identified, and there is no discussion of what functionality, other than implementation of the protocols, has been assigned to any of the clients or to the server.
An architectural pattern is not an architecture, then, but it still conveys a useful image of the system—it imposes useful constraints on the architecture and, in turn, on the system.
One of the most useful aspects of patterns is that they exhibit known quality attributes. This is why the architect chooses a particular pattern and not one at random. Some patterns represent known solutions to performance problems, others lend themselves well to high-security systems, still others have been used successfully in high-availability systems.

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. Arising from experience, reference models are a characteristic of mature domains.

A ***reference architecture*** is a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them. Whereas a reference model divides the functionality, a reference architecture is the mapping of that functionality onto a system decomposition. The mapping may be, but by no means necessarily is, one to one. A software element may implement part of a function or several functions.



(b)	Explain Layered System Architectural Style.	[5]	CO3	L2

Soln.

- A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below.
- The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers.

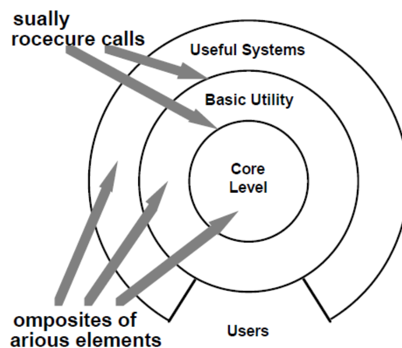


Figure 3: Layered Systems

- The most widely known examples of this kind of architectural style are layered communication protocols. In this application area, each layer provides a substrate for communication at some level of abstraction.

Layered systems have several desirable properties.

- First, they support design based on increasing levels of abstraction. This allows implementers to partition a complex problem into a sequence of incremental steps. Second, they support enhancement. Like pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers.
- Third, they support reuse. Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers. This leads to the possibility of defining standard layer interfaces to which different implementers can build.

But layered systems also have disadvantages.

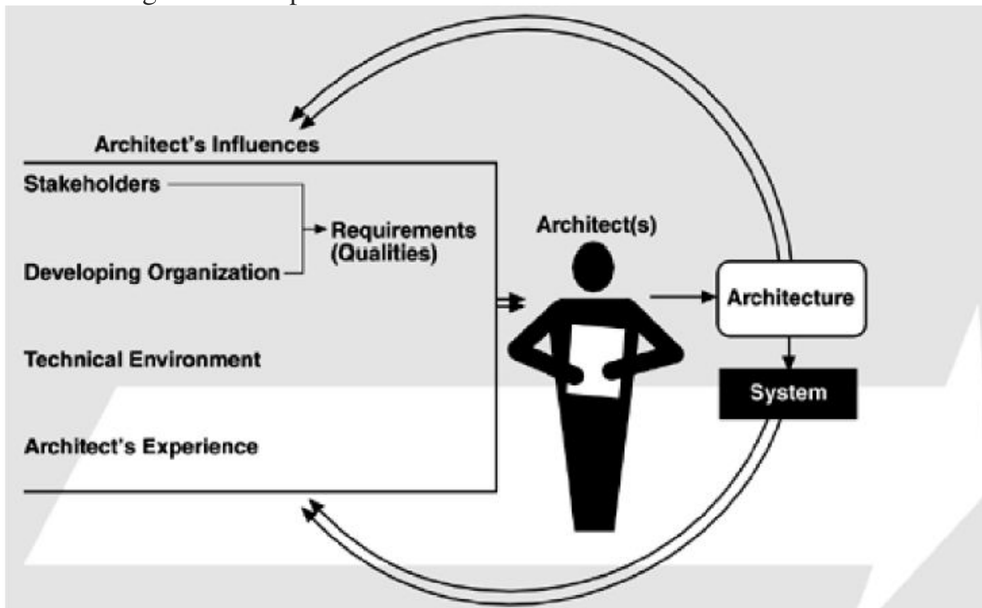
- Not all systems are easily structured in a layered fashion. And even if a system can logically be structured as layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations. Additionally, it can be quite difficult to find the right levels of abstraction. This is particularly true for standardized layered models.
- In one sense this is similar to the benefits of implementation hiding found in abstract data types. However, here there are multiple levels of abstraction and implementation.

- They are also similar to pipelines, in that components communicate at most with one other component on either side. But instead of simple pipe read/write protocol of pipes, layered systems can provide much richer forms of interaction.

5 (a)	Explain ABC with respect to software process and activities involved in creating software architecture.	[10]	CO1	L1

Soln. Software process is the term given to the organization, ritualization, and management of software development activities. These activities include the following:

- Creating the business case for the system
- Understanding the requirements
- Creating or selecting the architecture
- Documenting and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture



ARCHITECTURE ACTIVITIES

1. Creating the Business Case for the System

Creating a business case is broader than simply assessing the market need for a 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 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.

2. Understanding the Requirements

There are a variety of techniques for eliciting requirements from the stakeholders. For example, 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.

One fundamental decision with respect to the system being built is the extent to which it is a variation on other systems that have been constructed. Since it is a rare system these days that is not similar to other systems, requirements elicitation techniques extensively involve understanding these prior systems' characteristics.

Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly catalyze decisions on the system's design and the design of its user interface.

Regardless of the technique used to elicit the requirements, the desired qualities of the system to be constructed determine the shape of its architecture. Specific tactics have long been used by architects to achieve particular quality attributes. An architectural design embodies many tradeoffs, and not all of these tradeoffs are apparent when specifying requirements. It is not until the architecture is created that some tradeoffs among requirements become apparent and force a decision on requirement priorities.

3. Creating or Selecting the Architecture

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.

4. 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. Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds.

5. Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these 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. Becoming

more widespread are analysis techniques to evaluate the quality attributes that an architecture imparts to a system. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture.

6. Implementing 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. Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture is better.

7. Ensuring Conformance to an 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 faithful to each other during this phase. Although work in this area is comparatively immature, there has been intense activity in recent years.

6 (a)	Explain briefly the properties of a good software architecture design.	[6]	CO1	L1

Soln.

Process recommendations and product (or structural) recommendations.

Process recommendations are as follows:

1. The architecture should be the product of a single architect or a small group of architects with an identified leader.
2. 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.
3. 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.
4. The architecture should be circulated to the system's stakeholders, who should be actively involved in its review.
5. The architecture should be analyzed for applicable quantitative measures and formally evaluated for quality attributes before it is too late to make changes to it.
6. 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.

Structural rules of thumb are as follows:

1. The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns. The information-hiding modules should include those that encapsulate idiosyncrasies of the computing infrastructure, thus insulating the bulk of the software from change should the infrastructure change.
2. Each module should have a well-defined interface that encapsulates or "hides"

changeable aspects from other software that uses its facilities.

3. The architecture should never depend on a particular version of a commercial product or tool. If it depends upon a particular commercial product, it should be structured such that changing to a different product is straightforward and inexpensive.

4. Modules that produce data should be separate from modules that consume data. This tends to increase modifiability because changes are often confined to either the production or the consumption side of data.

5. For parallel-processing systems, the architecture should feature well-defined processes or tasks that do not necessarily mirror the module decomposition structure.

6. Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.

(b) Explain basic requirements of Mobile Robot's Architecture. [4] CO3 L2

Soln. **Typical software functions.**

- Acquiring and interpreting input provided by sensors.
- Controlling the motion of wheels and other movable parts.
- Planning future paths.

Examples of complications.

- Obstacles may block path.
- Sensor input may be imperfect.
- Robot may run out of power.
- Mechanical limitations may restrict accuracy of movement.
- Robot may manipulate hazardous materials.
- Unpredictable events may demand a rapid (autonomous) response.

Evaluation criteria for a given architecture

1. **Accommodation of deliberate and reactive behavior.** Robot must coordinate actions to achieve assigned objectives with the reactions imposed by the environment.

2. **Allowance for uncertainty.** Robot must function in the context of incomplete, unreliable and contradictory information.

3. **Accounting of dangers in the robot's operations and its environment.** Relating to fault tolerance, safety and performance, problems like reduced power supply, unexpectedly open doors, etc., should not lead to disaster.

4. **Flexibility.** Support for experimentation and reconfiguration.

7 (a)	Explain KWIC with problem statement & following architectural style i) Abstract data types ii) Main Program/Subroutine with shared data	[10]	CO3	L2

Soln. Problem Statement:

"The KWIC [Key Word in Context] 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 repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."

ii) Main Program/Subroutine with shared data

- The solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output. These computational components are coordinated as subroutines by a main program that sequences through them in turn. Data is communicated between the components through shared storage.

- Communication between the computational components and the shared data is an unconstrained read write protocol. This is made possible by the fact that the coordinating program guarantees sequential access to the data.

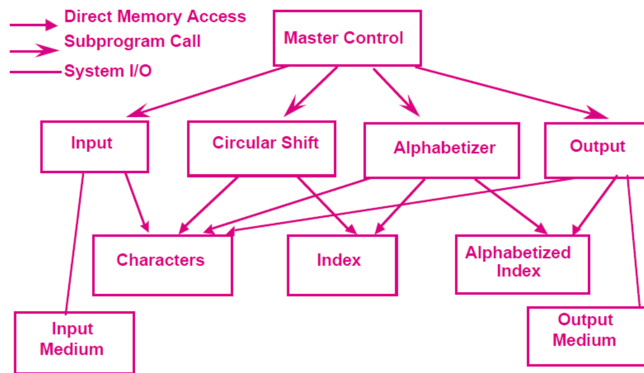


Figure 6: KWIC – Shared Data Solution

- Using this solution data can be represented efficiently, since computations can share the same storage. It has a number of serious drawbacks in terms of its ability to handle changes.
- In particular, a change in data storage format will affect almost all of the modules. Similarly changes in the overall processing algorithm and enhancements to system function are not easily accommodated.
- Finally, this decom-position is not particularly supportive of reuse.

i) *Abstract Data Types*

- In this case data is no longer directly shared by the computational components. Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface.
- This solution provides the same logical decomposition into processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered. In particular, both algorithms and data representations can be changed in individual modules without affecting others. Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.

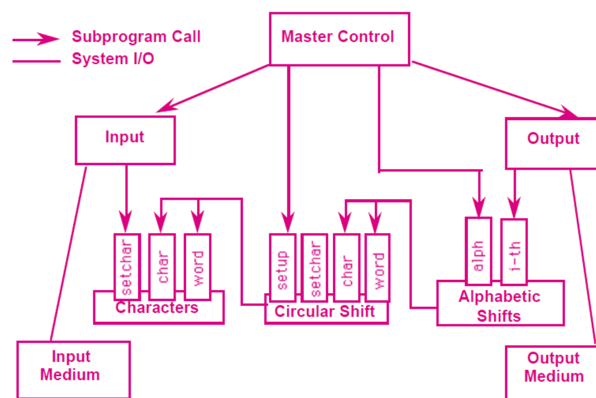


Figure 7: KWIC – Abstract Data Type Solution

- The main problem is that to add new functions to the system, the implementer must either modify the existing modules—compromising their simplicity and integrity—or add new modules that lead to performance penalties.

Scheme of Evaluation

Q No.		Description	Distribution of Marks		Total Marks
1.		Explain common software architectures.	3 M (Diagram) 2 M (Module) 3 M (Component & Connector) 2 M (Allocation)	10 M	10 M
2.	a.	Explain Process Control view of cruise control.	2 M (Diagram) 3 M (Computational Elements) 3 M (Data Elements)	2 M 6 M	10 M
	b.	Explain the problem statement of cruise control with input parameters.	1 M (Problem Statement) 1 M (Input Parameters)	2 M	
3.		Explain the following Architectural Styles i) Event Based Implicit Invocation ii) Pipes and Filters	5 M (EBII) 1 M (P & F Diagram) 4 M (P & F)	5 M 5 M	10M
4.	a	Explain the following concepts i) Architectural Pattern ii) Reference Model iii) Reference Architecture	1 M (Diagram) 2 M (AP) 1 M (RM) 1 M (RA)	5 M	10 M
	b	Explain Layered System Architectural Style.	1 M (Diagram) 4 M (Layered System)	5 M	
5.		Explain ABC with respect to software process and activities involved in creating software architecture.	1 M (Diagram) 1 M (Listing Activities) 8 M	10 M	10 M
6.	a.	Explain briefly the properties of a good software architecture design.	3 M (Process Recommendation) 3 M (Product Recommendation)	6 M	10 M
	b.	Explain basic requirements of Mobile Robot's Architecture.	2 M (Problem Statement) 2 M (4 Requirements)	4 M	
7.		Explain KWIC with problem statement & following architectural style i) Abstract data types ii) Main Program/Subroutine with shared data	2 M (Problem Statement) 1 M (Diagram) 3 M (ADT) 1 M (Diagram) 3 M (MP/S)	10M	10M