Thakur Educational Trust's (Regd.)

THAKUR RAMNARAYAN COLLEGE OF ARTS & COMMERCE

ISO 21001:2018 Certified

PROGRAMME: B.Sc (I.T)

CLASS: S.Y.B.Sc (I.T)

SUBJECT NAME: SOFTWARE ENGINEERING

SEMESTER: IV

FACULTY NAME: Ms. SMRITI DUBEY

UNIT III

Chapter 1 – Architectural Design

Concepts:

Introduction

Architectural Design Decisions

Organization Styles of Architectural Design

Modular decomposition styles

Control Styles

Introduction

Requirements of the software should be transformed into an architecture that describes the software's top-level structure and identifies its components. This is accomplished through architectural design (**also called system design**), which acts as a preliminary 'blueprint' from which software can be developed. **IEEE defines architectural design as** '**the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.'**

Architectural design is a process for identifying the sub-systems making up a system and the framework for sub-system control and communication. The output of this design process is a description of the software architecture. Architectural design is an early stage of the system design process. It represents the link between specification and design processes and is often carried out in parallel with some specification activities. It involves identifying major system components and their communications**.**

Software architectures can be designed at **two levels of abstraction**:

- **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Three **advantages** of explicitly designing and documenting software architecture:

- **Stakeholder communication**: Architecture may be used as a focus of discussion by system stakeholders.
- **System analysis**: Well-documented architecture enables the analysis of whether the system can meet its non-functional requirements.
- **Large-scale reuse**: The architecture may be reusable across a range of systems or entire lines of products.

Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Systems in the same domain often have **similar architectures** that reflect domain concepts. Application product lines are built around a core architecture with variants that satisfy particular customer requirements. The architecture of a system may be designed around one of more **architectural patterns/styles**, which capture the essence of an architecture and can be instantiated in different ways.

The particular architectural style should depend on the **non-functional system requirements**:

- **Performance**: localize critical operations and minimize communications. Use large rather than fine-grain components.
- **Security**: use a layered architecture with critical assets in the inner layers.
- **Safety**: localize safety-critical features in a small number of sub-systems.
- **Availability**: include redundant components and mechanisms for fault tolerance.
- **Maintainability**: use fine-grain, replaceable components.

Architectural views

Each architectural model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios. The views that he suggests are:

1. **A logical view**, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. **A process view**, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about nonfunctional system characteristics such as performance and availability.

3. **A development view**, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.

4. **A physical view**, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers

planning a system deployment.

Organization Styles of Architectural Design

An architectural pattern should describe a system organization that has been successful in previous systems. It should include information of when it is and is not appropriate to use that pattern, and the pattern's strengths and weaknesses.

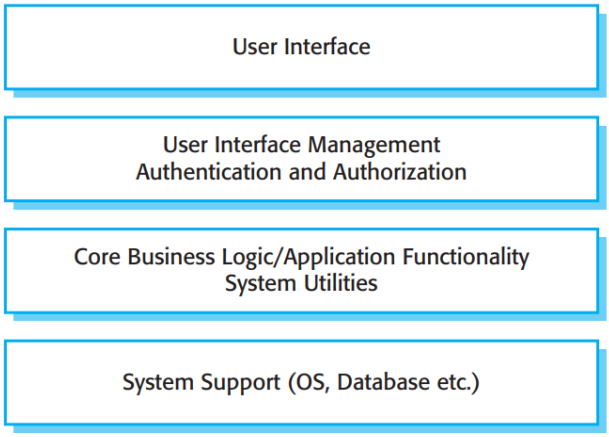
Patterns are a means of representing, sharing and reusing knowledge. An architectural pattern is a **stylized description of a good design practice**, which has been tried and tested in different environments. Patterns should include information about when they are and when they are not useful. Patterns may be represented using tabular and graphical descriptions.

Layered architecture

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. The layered architecture pattern is another way of achieving separation and independence. This pattern is shown in figure. Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected.

Figure 6.6 is an example of a layered architecture with four layers. The **lowest layer** includes system support software—typically database and operating system support. The **next layer** is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components. The **third layer** is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities. Of course, the number of layers is arbitrary. Any of the layers in Figure 6.6 could be split into two or more layers.



Repository Architecture

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, CAD systems, and interactive development environments for software.

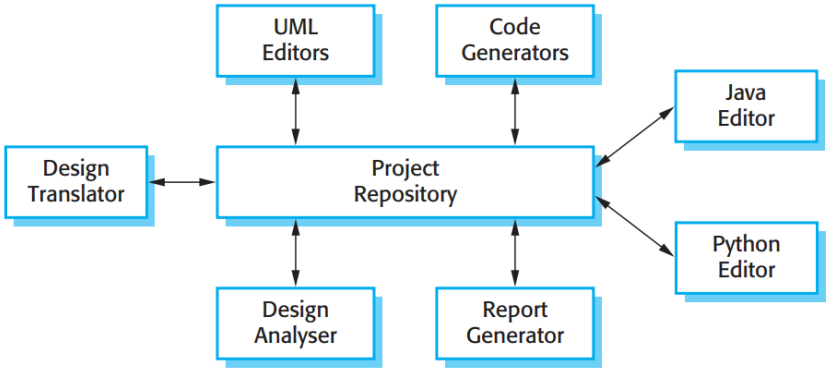


Figure is an illustration of a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development. The repository in this case might be a version-controlled environment that keeps track of changes to software and allows rollback to earlier versions. Organizing tools around a repository is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one component to another. However, components must operate around an agreed repository data model. In the

example shown in Figure, the repository is passive and control is the responsibility of the components using the repository.

An alternative approach, which has been derived for AI systems, uses a 'blackboard' model that triggers components when particular data become available. This is appropriate when the form of the repository data is less well structured. Decisions about which tool to activate can only be made when the data has been analyzed. This model is introduced by Nii (1986). Bosch (2000) includes a good discussion of how this style relates to system quality attributes.

Client Server Model

A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services.

2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.

3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

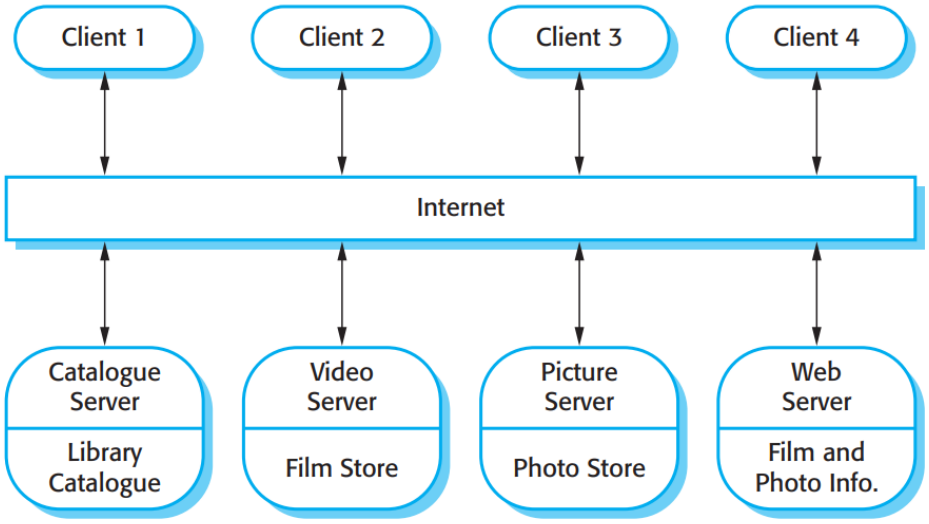
Client–server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer.

Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

Figure is an example of a system that is based on the client–server model. This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution.

They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.

The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clip. The client program is simply an integrated user interface, constructed using a web browser, to access these services.

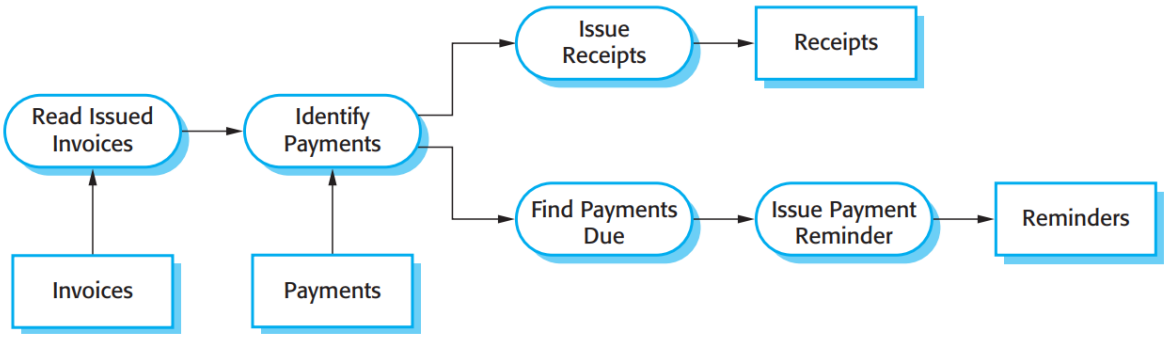


Pipe Filter Architecture

This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The name ‘pipe and filter’ comes from the original Unix system where it was possible to link processes using ‘pipes’. These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term ‘filter’ is used because a transformation ‘filters out’ the data it can process from its input data stream.

An example of this type of system architecture, used in a batch processing application, is shown in Figure. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.



Modular Decomposition Styles

Modular decomposition is a process of decomposing subsystems into modules. After decomposition of the system into subsystems, subsystems must be decomposed into modules.

1) Object oriented decomposition

In object-oriented model of decomposition, system is decomposed into a set of communicating objects. An object-oriented, architectural model structures the system into a set of loosely coupled objects with well-defined interfaces. Objects call on the services offered by other objects.

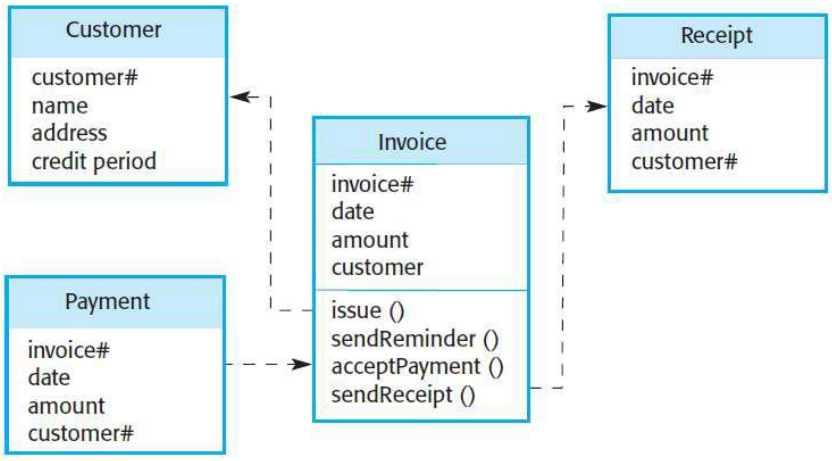


Figure: An object model of an invoice processing system

This system can issue invoices to customers, receive payments, and issue receipts for these payments and reminders for unpaid invoices.

Operations, if any, are defined in the lower part of the rectangle representing the object. Dashed arrows indicate that an object uses the attributes or services provided by another object.

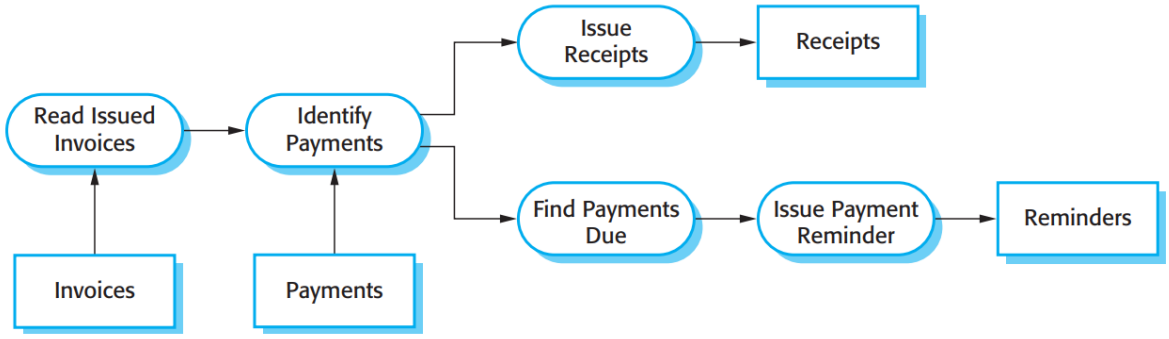
An object-oriented decomposition is concerned with object classes, their attributes and their operations.

When implemented, objects are created from these classes and some control model is used to coordinate object operations.

The Invoice class has various associated operations that implement the system functionality.

2) Function oriented decomposition

In function-oriented model of decomposition, system is decomposed into functional modules. A function-oriented, architectural model accepts input data and transforms it into output data. In a functional oriented pipeline functional transformations process their inputs and produces output. Data flows from one to another and is transformed as it moves through the sequence. The transformations may execute sequentially or in parallel. When the transformations are represented as separate processes, this model is sometimes called the pipe.



An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

Control Styles

To work as a system, sub-systems must be controlled so that their services are delivered to the right place at the right time. Control models at the architectural level are concerned with the control flow between sub-systems.

There are two generic control styles that are used in software systems:

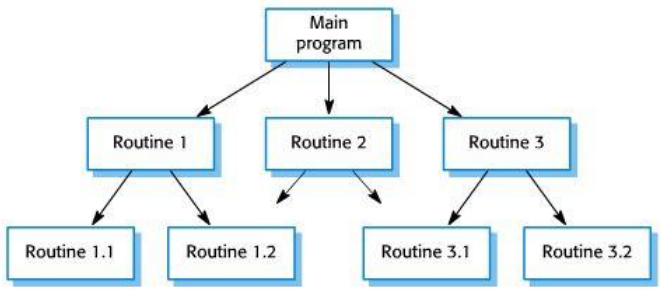
1) **Centralized control:** Centralized model is a formulation of centralized control in which one subsystem has overall responsibility for control and starts and stops other subsystems. It is a control subsystem that takes responsibility for managing the execution of other subsystems.

2) **Event based control**: Event-based models are those in which each sub-system can respond to externally generated events from other subsystems or the system's environment. It is a system driven by externally generated events where the timing of the events is out with the control of the subsystems which process the event.

TYPES OF CENTRALIZED MODELS

Call-return Model

In call-return model, it is a model which has top-down subroutine architecture where control starts at the top of a subroutine hierarchy and moves downwards. It is applicable to sequential systems. This familiar model is embedded in programming languages such as C, Ada and Pascal. Control passes from a higher-level routine in the hierarchy to lower-level routine. This call-return model may be used at the module level to control functions or objects



The call-return model is illustrated in Figure 3.1. The main program calls routines 1, 2 and 3 whilst Routine 1 can call Routines 1.1 or 1.2. This is a model of program dynamics. It is not a structural model; there is no need for Routine1.1, for example, to be a part of Routine 1.

Manager Model

Manager model is applicable to concurrent systems. One system component controls the stopping, starting and coordination of other system processes. It can be implemented in sequential systems as a case system.

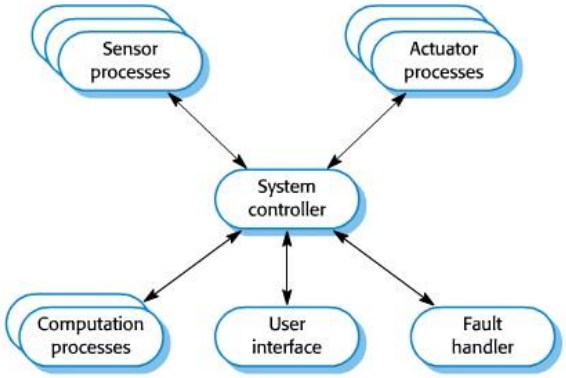
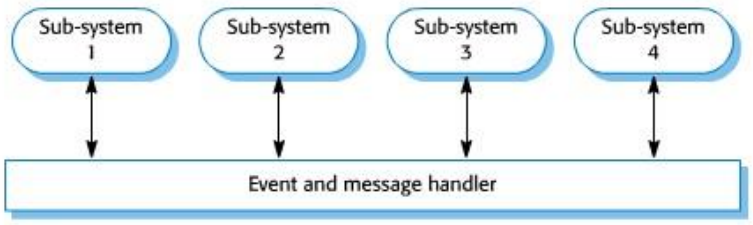


Figure is an illustration of centralized management model for a concurrent system. It is often used in real time systems which do not have very tight constraints. The central controller manages the execution of a set of processes associated with sensors and actuators. The system controller process decides when processes should be started or stopped depending on system state variables. The controller usually loops continuously, polling sensors and other processes for events or state changes. For this reason, this model is called an event-loop model.

TYPES OF EVENT-BASED MODELS

Broadcast Model

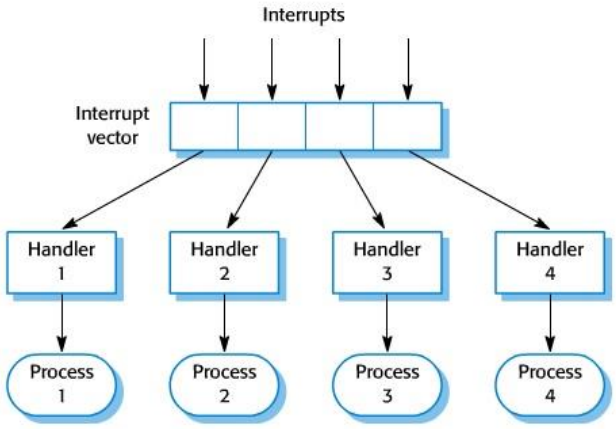
It is a model in which an event is broadcast to all subsystems. Any subsystem which can handle the broadcasting event may behave as a broadcast model. These are effective in integrating components distributed across different computers on a network. The advantage of this model is that evolution is simple. This distribution is transparent to other components. The disadvantage is that the components don't know if the event will be handled.



In Figure, components register an interest in specific events. When these events occur, control is transferred to the component that can handle the event.

Interrupt-driven Model

Interrupt-driven model is used in real time systems where interrupts are detected by and interrupt handler and passed to some other component for processing. This model is used in real-time systems where immediate response to some event is necessary. The advantage is that it allows very fast responses to events to be implemented. The disadvantage is that it is complex to program a difficult to validate.



In Figure, there are known number of interrupt types with a handler for each type. Each type of interrupt is associated with the memory location where its handler's address is stored. The interrupt handler may start or stop the processes in response to the event signaled by the interrupt.