

	WWW.STUDENTSFOCUS.COM	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for Design Process and Design concept		
Time:	50 Minutes	
Lesson. No	Unit – III Lesson No: 1,2/11	

1.Content List: Design Process and Design concept

2.Skill addressed:

- Understand the Concepts of Design Process and Design concept

3.Objectives of this Lesson Plan:

To enable students to understand the Concept of basic Design Process and Design concept

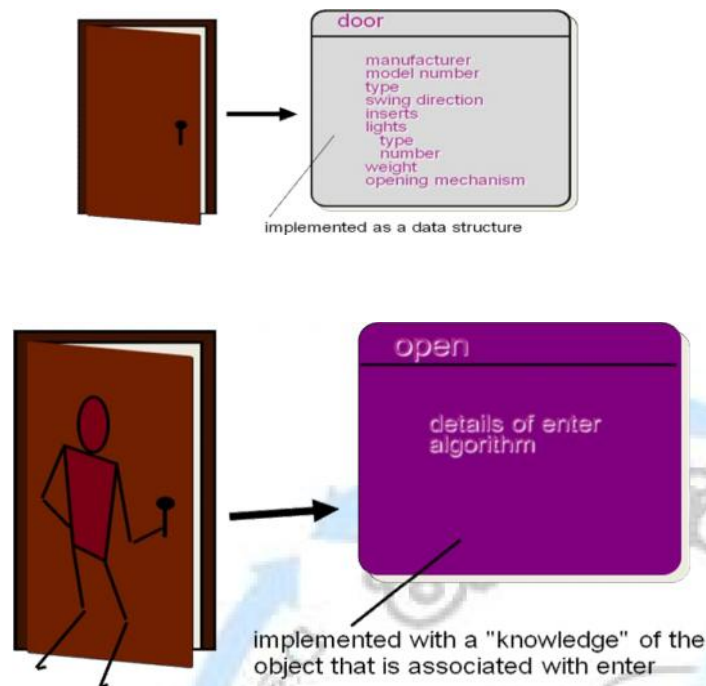
4.Outcome(s):

Understand And Analyse the basic concepts of Design Process and Design concept

5.Link sheet:

1. Mention some of design principles.
2. State procedural abstraction.
3. What does abstraction contains?

6.Evocation: (5 Minutes)



Subject introduction (40 Minutes):

Topics:

- Design process
- Design principle
- Design concepts
- Abstraction
- Refinement

7.Lecture Notes

Design Process

Design process is a sequence of steps carried through which the requirement are translated into a system or software model

Design products

1. In architectural design the subsystem components can be identified.
2. The abstract specification is used to specify the subsystems.
3. The interfaces between the subsystems are designed, which is called interface
4. design.
5. In component design of subsystems components is done ..
6. The data structure is designed to hold the data
7. For performing the required functionality, the appropriate algorithm is designed.

Design Principle

Davis suggested a set of principles for software design as:

- The design process should not suffer from "tunnel vision",
- The design should be traceable to the Analysis model,
- The design should not reinvent the wheel.
- The design should “minimize the intellectual distance” between the software and the problem in the real world
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- Design is not coding.
- The design should be assessed for quality.

The design should be reviewed to minimize conceptual errors.

Design Principles – 1:

- Process should not suffer from tunnel vision – consider alternative approaches
- Design should be traceable to analysis model
- Do not try to reinvent the wheel use design patterns ie reusable components
- Design should exhibit both uniformity and integration
- Should be structured to accommodate changes

Design Principles – 2 :

- Design is not coding and coding is not design
- Should be structured to degrade gently, when bad data, events, or operating conditions are encountered
- Needs to be assessed for quality as it is being created
- Needs to be reviewed to minimize conceptual (semantic) errors

Design Concepts -1 :

- Abstraction allows designers to focus on solving a problem without being concerned about irrelevant lower level details .Procedural abstraction is a named sequence of instructions that has a specific and limited function

e.g open a door

- Open implies a long sequence of procedural steps

data abstraction is collection of data that describes a data object

e.g door type, opening mech, weight,dimensions

Design Concepts -2 :

- Design Patterns description of a design structure that solves a particular design problem within a specific context and its impact when applied

Design Concepts -3 :

- Software Architecture
 - overall structure of the software components and the ways in which that structure
 - provides conceptual integrity for a system

Design Concepts -4 :

- • Information Hiding information (data and procedure) contained within a module is inaccessible to modules that have no need for such information
- • Functional Independence achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models • Objects encapsulate both data and data manipulation procedures needed to describe the content and behavior of a real world entity Class generalized description (template or pattern) that describes a collection of similar objects
- • Inheritance provides a means for allowing subclasses to reuse existing superclass data and procedures; also provides mechanism for propagating changes



Design Concepts – 5:

- Messages the means by which objects exchange information with one another
- Polymorphism
 - a mechanism that allows several objects in a class hierarchy to have different methods with the same name
 - instances of each subclass will be free to respond to messages by calling their own version of the method
 - Abstraction
 - Procedural abstraction – a sequence of instructions that have a specific and limited function
 - Data abstraction – a named collection of data that describes a data object
 - Architecture
 - The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
 - Consists of components, connectors, and the relationship between them
 - Patterns
 - A design structure that solves a particular design problem within a specific context
 - It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns
 - Modularity
 - Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
 - Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
 - Information hiding
 - The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
 - This enforces access constraints to both procedural (i.e., implementation) detail and local data structures
 - Functional independence
 - Modules that have a "single-minded" function and an aversion to excessive interaction with other modules
 - High cohesion – a module performs only a single task
 - Low coupling – a module has the lowest amount of connection needed with other modules

8.Textbook:

T1: Roger S. Pressman, “Software Engineering – A practitioner’s Approach”, Sixth Edition, McGraw-Hill International Edition, 2005

9.Application: Software Process

	Sri Vidya College of Engineering &Technology Department of Computer science & Engineering	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for	design model, Design heuristic	
Time:	50 Minutes	
Lesson. No	Unit – III Lesson No: 3,4/11	

1.Content List: design model, Design heuristic

2.Skill addressed:

- Understand the Concepts of design model, Design heuristic

3.Objectives of this Lesson Plan:

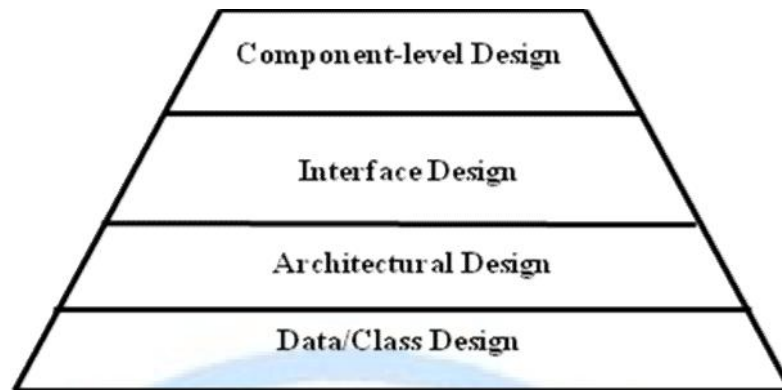
To enable students to understand and remember the Concept of basic design model,
Design heuristic

4.Outcome(s):

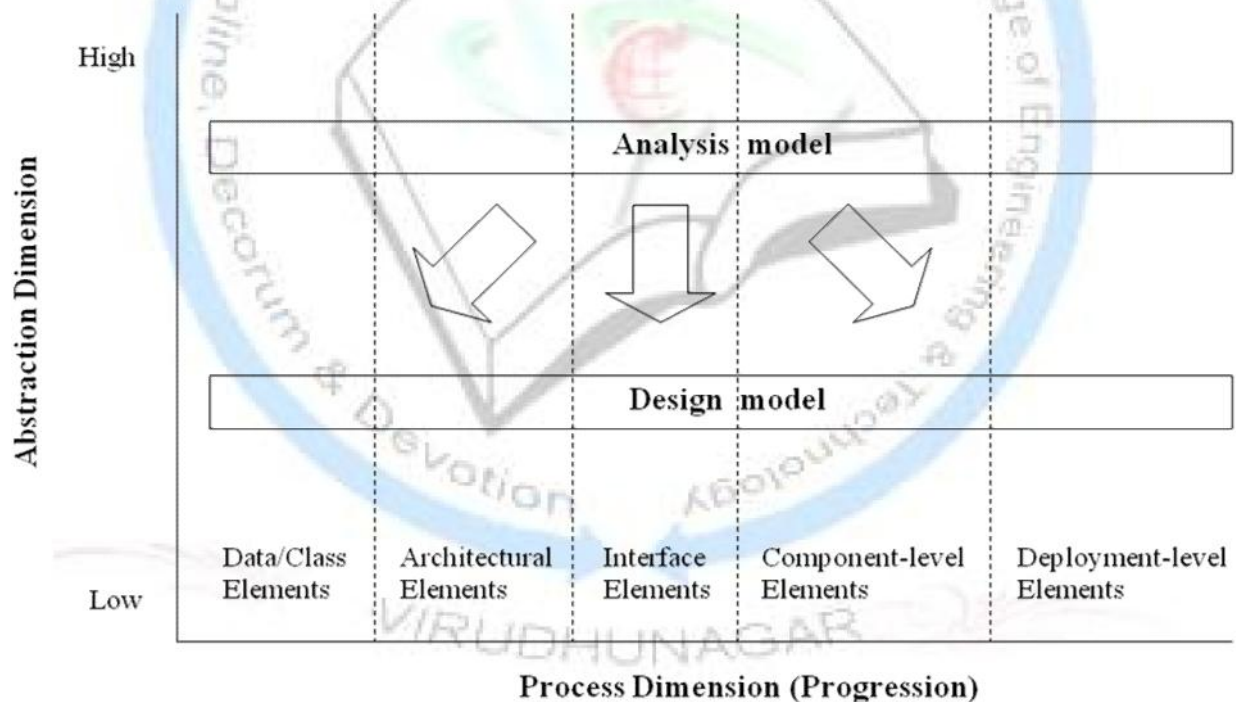
Understand And Analyse the basic concepts of design model, Design heuristic

5.Link sheet:

6.Evocation: (5 Minutes)



Dimensions of the Design Model



Subject introduction (40 Minutes):

Topics: Design Heuristics for Effective Modularity

7.Lecture Notes

Design model

- The design model can be viewed in two different dimensions
 - (Horizontally) The process dimension indicates the evolution of the parts of the design model as each design task is executed

- (Vertically) The abstraction dimension represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined
- Elements of the design model use many of the same UML diagrams used in the analysis model
 - The diagrams are refined and elaborated as part of the design
 - More implementation-specific detail is provided
 - Emphasis is placed on
 - Architectural structure and style
 - Interfaces between components and the outside world
 - Components that reside within the architecture
- Design model elements are not always developed in a sequential fashion
 - Preliminary architectural design sets the stage
 - It is followed by interface design and component-level design, which often occur in parallel
- The design model has the following layered elements
 - Data/class design
 - Architectural design
 - Interface design
 - Component-level design. A fifth element that follows all of the others is deployment-level design

Design Elements

- Data/class design
 - Creates a model of data and objects that is represented at a high level of abstraction
- Architectural design
 - Depicts the overall layout of the software
- Interface design
 - Tells how information flows into and out of the system and how it is communicated among the components defined as part of the architecture
 - Includes the user interface, external interfaces, and internal interfaces
- Component-level design elements
 - Describes the internal detail of each software component by way of data structure definitions, algorithms, and interface specifications
- Deployment-level design elements
 - Indicates how software functionality and subsystems will be allocated within the physical computing environment that will support the software



Design Heuristics for Effective Modularity

- Evaluate the first iteration of the program structure to reduce coupling and improve cohesion.
- Attempt to minimize structures with high fan-out; strive for fan-in as structure depth increases.
- Keep the scope of effect of a module within the scope of control for that module.
- Evaluate module interfaces to reduce complexity, reduce redundancy, and improve consistency.
- Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single subfunction).
- Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

8.Textbook:

T1: Roger S. Pressman, “Software Engineering – A practitioner’s Approach”, Sixth Edition, McGraw-Hill International Edition, 2005

9.Application: Software Process

	<div>Sri Vidya College of Engineering &Technology</div> <div>Department of Computer science & Engineering</div>	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for Architectural design, Architectural styles,		
Time:	50 Minutes	
Lesson. No	Unit – III Lesson No: 5,6/11	

1.Content List: Architectural design, Architectural styles,

2.Skill addressed:

Understand the Concepts of Architectural design, Architectural styles,

3.Objectives of this Lesson Plan:

To enable students to understand the Concept of basic Architectural design,
Architectural styles,

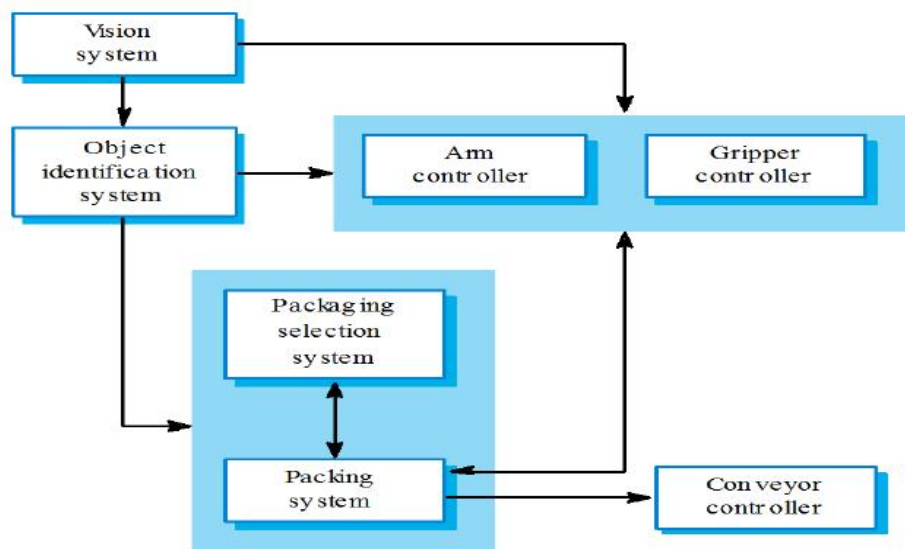
4.Outcome(s):

Understand And Analyse the basic concepts of Architectural design, Architectural styles

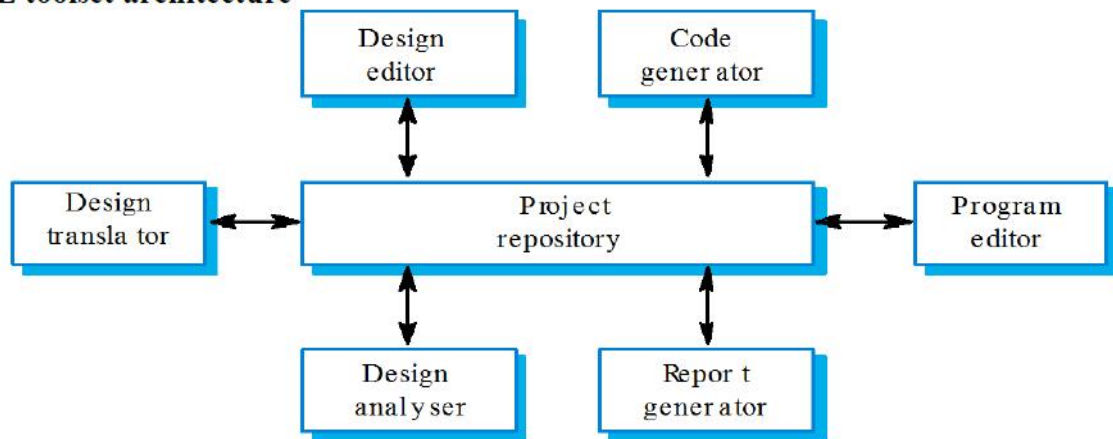
5.Link sheet:

4. Define architectural design

6.Evocation: (5 Minutes)



CASE toolset architecture



Repository model characteristics

Subject introduction (40 Minutes):

Topics:

- Introduction
- Functional Requirements
- Types of Requirements

7.Lecture Notes

Architectural Design

- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

Advantages of explicit architecture

- Stakeholder communication
 - ❖ Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
 - ❖ Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
 - ❖ The architecture may be reusable across a range of systems.

Architecture and system characteristics

- Performance
 - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
 - Use a layered architecture with critical assets in the inner layers.
- Safety
 - Localise 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 conflicts

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localising safety-related features usually means more communication so degraded performance.

System structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

Packing robot control system

Box and line diagrams

- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes.
- 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?

Architecture reuse

- 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.

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style.
- An awareness of these styles can simplify the problem of defining system architectures.

- However, most large systems are heterogeneous and do not follow a single architectural style.

Architectural models

- Used to document an architectural design.

Static structural model that shows the major system components.

- Dynamic process model that shows the process structure of the system.
- Interface model that defines sub-system interfaces.
- Relationships model such as a data-flow model that shows sub-system relationships.
- Distribution model that shows how sub-systems are distributed across computers.

System organisation

- Reflects the basic strategy that is used to structure a system.
- Three organisational styles are widely used:
 - A shared data repository style;
 - A shared services and servers style;
 - An abstract machine or layered style.

The repository model

- Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
 - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

CASE toolset architecture

➤ Repository model

Advantages

- Efficient way to share large amounts of data;
- Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
- Sharing model is published as the repository schema.

Disadvantages

Sub-systems must agree on a repository data model. Inevitably a compromise;

- Data evolution is difficult and expensive;
- No scope for specific management policies;
- Difficult to distribute efficiently.

Client-server model

- Distributed system model which shows how data and processing is distributed across a range of components.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.

Client-server characteristics

Advantages

- Distribution of data is straightforward;
- Makes effective use of networked systems. May require cheaper hardware;
- Easy to add new servers or upgrade existing servers.

Disadvantages

- No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
- Redundant management in each server;
- No central register of names and services - it may be hard to find out what servers and services are available.

Abstract machine (layered) model

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.

Modular decomposition styles

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organisation and modular decomposition.

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.
- Modular decomposition
- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered
 - An object model where the system is decomposed into interacting object;
 - A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

Object model

- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations

Object model advantages

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-oriented pipelining

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

Common Architectural Styles of American Homes

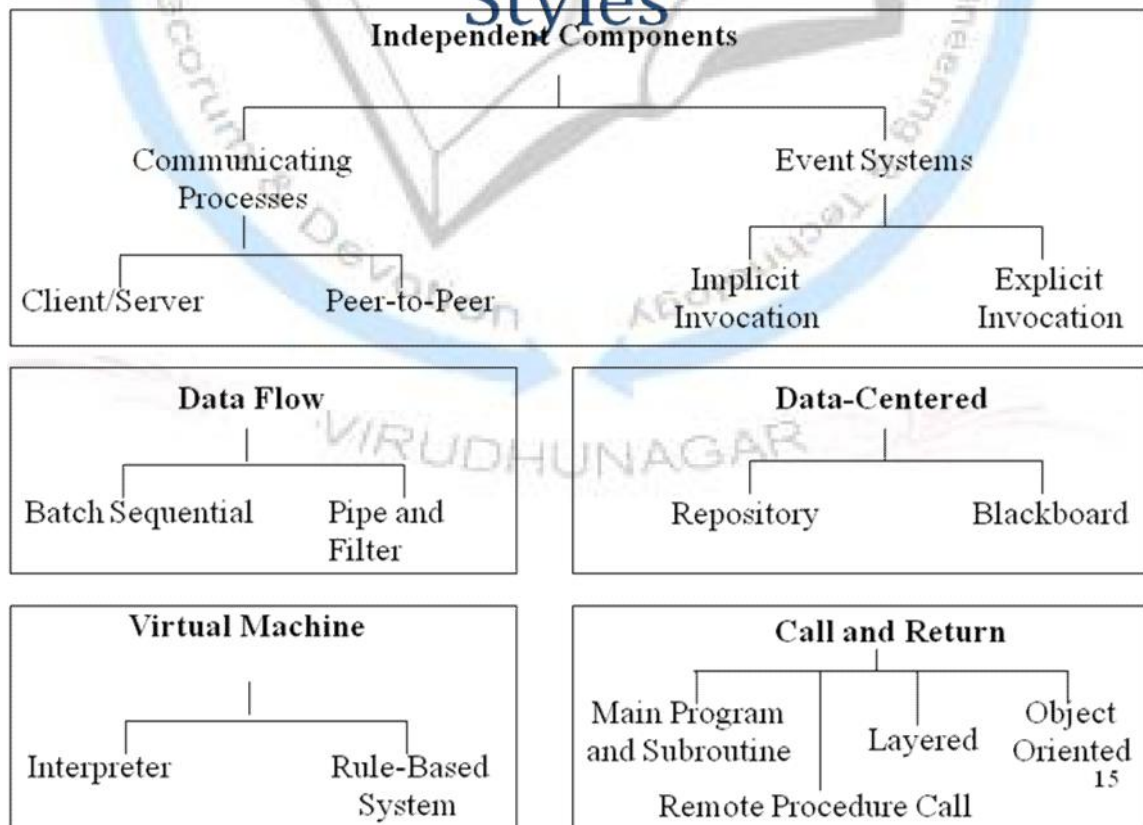


Common Architectural Styles of American Homes



13

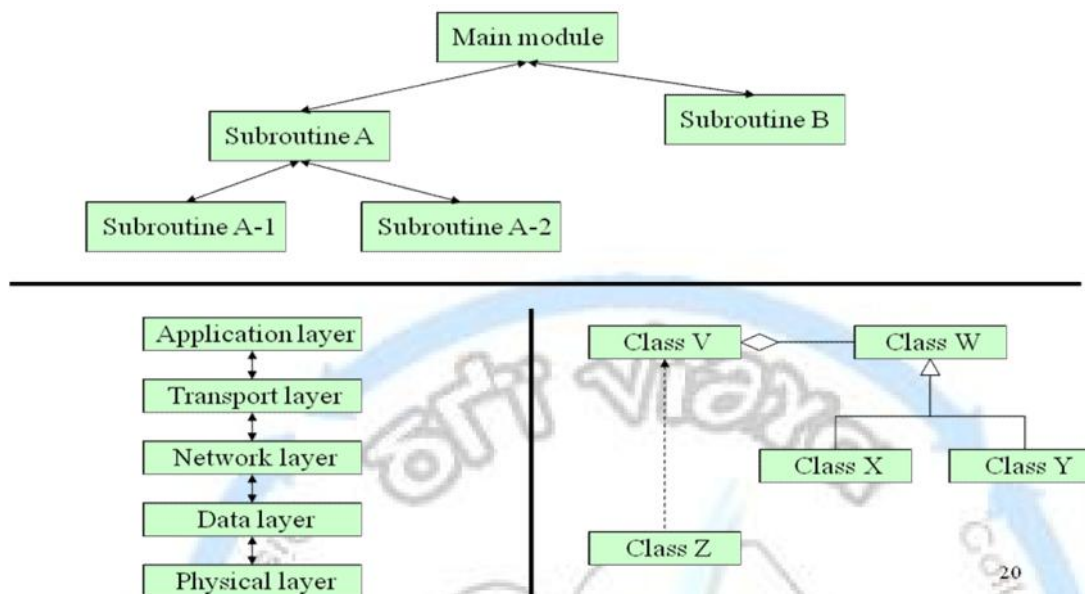
A Taxonomy of Architectural Styles



Data Flow style

- Has the goal of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- Batch sequential style
 - The processing steps are independent components
 - Each step runs to completion before the next step begins
- Pipe-and-filter style
 - Emphasizes the incremental transformation of data by successive components
 - The filters incrementally transform the data (entering and exiting via streams)
 - The filters use little contextual information and retain no state between instantiations
 - The pipes are stateless and simply exist to move data between filters
- Advantages
 - Has a simplistic design in the limited ways in which the components interact with the environment
 - Consists of no more and no less than the construction of its parts
 - Simplifies reuse and maintenance
 - Is easily made into a parallel or distributed execution in order to enhance system performance
- Disadvantages
 - Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
 - Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
 - Exhibits poor performance
 - Filters typically force the least common denominator of data representation (usually ASCII stream)
 - Filter may need unlimited buffers if they cannot start producing output until they receive all of the input
 - Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time
- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
 - The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style





- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
 - Decomposes a program hierarchically into small pieces (i.e., modules)
 - Typically has a single thread of control that travels through various components in the hierarchy
- Remote procedure call style
 - Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
 - Strives to increase performance by distributing the computations and taking advantage of multiple processors
 - Incurs a finite communication time between subroutine call and response
- Object-oriented or abstract data type system
 - Emphasizes the bundling of data and how to manipulate and access data
 - Keeps the internal data representation hidden and allows access to the object only through provided operations
 - Permits inheritance and polymorphism
- Layered system
 - Assigns components to layers in order to control inter-component interaction
 - Only allows a layer to communicate with its immediate neighbor
 - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
 - Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
 - Is compromised by layer bridging that skips one or more layers to improve runtime performance

- Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

8.Textbook:

T1: Roger S. Pressman, “Software Engineering – A practitioner’s Approach”, Sixth Edition, McGraw-Hill International Edition, 2005

9.Application: Software design

	<div>Sri Vidya College of Engineering &Technology</div> <div>Department of Computer science & Engineering</div>	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for Interface analysis, Interface Design		
Time:	50 Minutes	
Lesson. No	Unit – II Lesson No: 8,9/11	

1.Content List: Interface analysis, Interface Design

2.Skill addressed:

- Understand the Concepts of Interface analysis, Interface Design

3.Objectives of this Lesson Plan:

To enable students to understand the Concept of Interface analysis, Interface Design

4.Outcome(s):

Understand And remember the basic concepts of Interface analysis, Interface Design

5.Link sheet:

5. Define user interface design

6.Evocation: (5 Minutes)

Subject introduction (40 Minutes):

Topics:

- User Interface Analysis

7.Lecture Notes

Elements of the User Interface

To perform user interface analysis, the practitioner needs to study and understand four elements

- The users who will interact with the system through the interface
- The tasks that end users must perform to do their work
- The content that is presented as part of the interface
- The work environment in which these tasks will be conducted

User Analysis

- The analyst strives to get the end user's mental model and the design model to converge by understanding
 - The users themselves
 - How these people use the system
- Information can be obtained from
 - User interviews with the end users
 - Sales input from the sales people who interact with customers and users on a regular basis
 - Marketing input based on a market analysis to understand how different population segments might use the software
 - Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use A set of questions should be answered during user analysis .

User Analysis Questions:

- Are the users trained professionals, technicians, clerical or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?
- Are the users expert typists or are they keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform or are they volunteers?
- Do users work normal office hours, or do they work whenever the job is required?
- Is the software to be an integral part of the work users do, or will it be used only occasionally?

- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Task Analysis and Modeling

- Task analysis strives to know and understand
 - The work the user performs in specific circumstances
 - The tasks and subtasks that will be performed as the user does the work
 - The specific problem domain objects that the user manipulates as work is performed
 - The sequence of work tasks (i.e., the workflow)
 - The hierarchy of tasks
- Use cases
 - Show how an end user performs some specific work-related task
 - Enable the software engineer to extract tasks, objects, and overall workflow of the interaction
 - Helps the software engineer to identify additional helpful features

Content Analysis

- The display content may range from character-based reports, to graphical displays, to multimedia information
- Display content may be
 - Generated by components in other parts of the application
 - Acquired from data stored in a database that is accessible from the application
 - Transmitted from systems external to the application in question
- The format and aesthetics of the content (as it is displayed by the interface) needs to be considered

Analysis of display content

- A set of questions should be answered during content analysis
- Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?
- Are users able to customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- Can large reports be partitioned for ease of understanding?

- Are mechanisms available for moving directly to summary information for large collections of data?
- Is graphical output scaled to fit within the bounds of the display device that is used?
- How is color used to enhance understanding?
- How are error messages and warnings presented in order to make them quick and easy to see and understand?

Work Environment Analysis

- Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use
- Factors to consider include
 - Type of lighting
 - Display size and height
 - Keyboard size, height and ease of use
 - Mouse type and ease of use
 - Surrounding noise
 - Space limitations for computer and/or user
 - Weather or other atmospheric conditions
 - Temperature or pressure restrictions
 - Time restrictions (when, how fast, and for how long)

User Interface Design

- User interface design is an iterative process, where each iteration elaborate and refines the information developed in the preceding step
- General steps for user interface design
 - 1) Using information developed during user interface analysis, define user interface objects and actions (operations)
 - 2) Define events (user actions) that will cause the state of the user interface to change; model this behavior
 - 3) Depict each interface state as it will actually look to the end user
 - 4) Indicate how the user interprets the state of the system from information provided through the interface
- During all of these steps, the designer must
 - 1) Always follow the three golden rules of user interfaces

- 2) Model how the interface will be implemented
- 3) Consider the computing environment (e.g., display technology, operating system, development tools) that will be used

Interface Objects and Actions

- Interface objects and actions are obtained from a grammatical parse of the use cases and the software problem statement
- Interface objects are categorized into types: source, target, and application
 - 1) A source object is dragged and dropped into a target object such as to create a hardcopy of a report
 - 2) An application object represents application-specific data that are not directly manipulated as part of screen interaction such as a list
- After identifying objects and their actions, an interface designer performs screen layout which involves
 - 1) Graphical design and placement of icons
 - 2) Definition of descriptive screen text
 - 3) Specification and titling for windows
 - 4) Definition of major and minor menu items
 - 5) Specification of a real-world metaphor to follow

Design Issues

- Four common design issues usually surface in any user interface
 - 1) System response time (both length and variability)
 - 2) User help facilities
 - When is it available, how is it accessed, how is it represented to the user, how is it structured, what happens when help is exited
 - 3) Error information handling (more on next slide)
 - How meaningful to the user, how descriptive of the problem
 - 4) Menu and command labeling (more on upcoming slide)
 - Consistent, easy to learn, accessibility, internationalization
- Many software engineers do not address these issues until late in the design or construction process
 - 1) This results in unnecessary iteration, project delays, and customer frustration

Guidelines for Error Messages



- The message should describe the problem in plain language that a typical user can understand
- The message should provide constructive advice for recovering from the error
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)
- The message should be accompanied by an audible or visual cue such as a beep, momentary flashing, or a special error color
- The message should be non-judgmental .The message should never place blame on the user

Questions for Menu Labeling and Typed Commands

- Will every menu option have a corresponding command?
- What form will a command take? A control sequence? A function key? A typed word?
- How difficult will it be to learn and remember the commands?
- What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

8.Textbook:

T1: Roger S. Pressman, “Software Engineering – A practitioner’s Approach”, Sixth Edition, McGraw-Hill International Edition, 2005

	Sri Vidya College of Engineering & Technology Department of Computer science & Engineering	
Class	II Year / CSE (04 Semester)	
Subject Code	CS6403	
Subject	SOFTWARE ENGINEERING	
Prepared By	P.Ramya	
Lesson Plan for	Designing Class based components, traditional Components.	
Time:	50 Minutes	
Lesson. No	Unit – III Lesson No: 10,11/11	

1.Content List: Designing Class based components, traditional Components.

2.Skill addressed:

- Understand the Concepts of Designing Class based components, traditional Components.

3.Objectives of this Lesson Plan:

To enable students to understand the Concept of basic Designing Class based components, traditional Components

4.Outcome(s):

Understand And Analyse the basic concepts of Designing Class based components, traditional Components.

5.Link sheet:

6. Define Components .

6.Evocation: (5 Minutes)

Subject introduction (40 Minutes):

Topics:

- Component-level Design Principles
- Component Packaging Principles
- Component-Level Design Guidelines
- Cohesion,Coupling

7.Lecture Notes

Component-level Design Principle

- Open-closed principle
 - A module or component should be open for extension but closed for modification
 - The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
- Liskov substitution principle
 - Subclasses should be substitutable for their base classes
 - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- Dependency inversion principle
 - Depend on abstractions (i.e., interfaces); do not depend on concretions
 - The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
 - Interface segregation principle
 - Many client-specific interfaces are better than one general purpose interface
 - For a server class, specialized interfaces should be created to serve major categories of clients
 - Only those operations that are relevant to a particular category of clients should be specified in the interface

Component Packaging Principles

- Release reuse equivalency principle
 - The granularity of reuse is the granularity of release
 - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle
 - Classes that change together belong together

- Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle
 - Classes that aren't reused together should not be grouped together
 - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

Component-Level Design Guidelines

- Components
 - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
 - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
 - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
 - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)

Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component

Cohesion

- Cohesion is the “single-mindedness’ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)

Functional

- A module performs one and only one computation and then returns a result

Layer

- A higher layer component accesses the services of a lower layer component

Communicational

- All operations that access the same data are defined within one class

Kinds of cohesion

➤ Sequential

Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations

➤ Procedural

Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them

➤ Temporal

Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected

➤ Utility

Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
 - Data coupling

- Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
- Stamp coupling
 - A whole data structure or class instantiation is passed as a parameter to an operation
- Control coupling
 - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
 - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
- Common coupling
 - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
- Content coupling
 - One component secretly modifies data that is stored internally in another component
- Other kinds of coupling (unranked)
 - Subroutine call coupling
 - When one operation is invoked it invokes another operation within side of it
 - Type use coupling
 - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
 - If/when the type definition changes, every component that declares a variable of that data type must also change
 - Inclusion or import coupling
 - Component A imports or includes the contents of component B
 - External coupling
 - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component-Level Design

- Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
- Identify all design classes that correspond to the infrastructure domain
 - These classes are usually not present in the analysis or architectural models
 - These classes include GUI components, operating system components, data management components, networking components, etc.
- Elaborate all design classes that are not acquired as reusable components
 - Specify message details (i.e., structure) when classes or components collaborate
 - Identify appropriate interfaces (e.g., abstract classes) for each component
 - Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
- Describe persistent data sources (databases and files) and identify the classes required to manage them
- Develop and elaborate behavioral representations for a class or component
 - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
- Elaborate deployment diagrams to provide additional implementation detail
 - Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments
- Factor every component-level design representation and always consider alternatives
 - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
 - The final decision can be made by using established design principles and guidelines
 - Describe processing flow within each operation in detail by means of pseudo code or UML activity diagrams

8.Textbook:

T1: Roger S. Pressman, “Software Engineering – A practitioner’s Approach”, Sixth Edition, McGraw-Hill International Edition, 2005