CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Object Oriented Software Engineering

LECTURE 11

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Outline

Designing class based components

User interface analysis and design

Interface analysis and Interface design steps

Designing class based components

- **Software Components**
- A **software component** is a self-contained piece of software that contains or encapsulates a known set of operational units, relationships, and behaviors. Think of them as the building blocks for a system. You combine any number of these components to get the capabilities you desire. Often, they can be combined in more than one way. For example, consider a common bicycle. The wheels are considered a component, the handlebars are considered a component, and the seat is considered another component. Each provides an important piece of the whole.

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

What is a component?

- A component is a basic building block for the computer software.
- It is a higher level abstractions defined by their interfaces.
- It helps in achieving the objectives & requirements of system to be built

Definition for a component

- A famous definition proposed by Council and Heinemann::

"Component is a software element that conforms to a standard component model and can be independently deployed and composed without modification according to a composition standard."

Characteristics of component

- Standardized
- Independent
- Composable
- Deployable
- Documented

STANDARDIZED

- A component used in a CBSE process has to conform to a standardized component model.
- The model may define: component interfaces, component metadata, documentation and deployment.

INDEPENDENT

- It should be possible to compose and deploy it without having to use other specific components.
- In case of need of external service these should be explicitly set out in a 'requires' interface specification.

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

COMPOSABLE

- All external interactions must take place through publicly defined interfaces.
- Also it must provide external access to information about itself such as its methods and attributes.

DEPLOYABLE

- A component has to be self contained.
- It should be able to stand alone entity on a component platform that provides an implementation of component model.

DOCUMENTED

- Components have to be fully documented so that the users can decide whether or not the components meet their needs.
- Each and every details of the component interfaces should be specified clearly.

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Software Component

- An individual software component is a software package, a Web service, or a module that encapsulates a set of related functions (or data).
- Software components are modular and cohesive.
- Components communicate with each other via interfaces.
- Component based development embodies good software engineering practice.

- Component level design establishes the algorithmic detail required to manipulate data structures, effect communication between software components via their interfaces, and implement the processing algorithms allocated to each component

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Essentials of a CBSE

- Independent components that are completely specified by their interfaces.
- Component standards that facilitates the integration of components.
- Middleware that provides support for integration of components.

Software Component

- Defined
- A software component is a modular building block for computer software

– It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces

• A component communicates and collaborates with

– Other components

– Entities outside the boundaries of the system

• Three different views of a component

– An object-oriented view

– A conventional view

– A process-related view

Object-oriented View

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
- *This also involves defining the interfaces that enable classes to communicate and collaborate*

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Object-oriented View contd..

- This elaboration activity is applied to every component defined as part of the architectural design.
- Once this is completed, the following steps are performed 1) Provide further elaboration of each attribute, operation, and interface
- 2) Specify the data structure appropriate for each attribute
- 3) Design the algorithmic detail required to implement the processing logic associated with each operation
- 4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects

Conventional View

- A component is viewed as a functional element (i.e., a module) of a program that incorporates

– The processing logic

– The internal data structures that are required to implement the processing logic

– An interface that enables the component to be invoked and data to be passed to it

Conventional View contd…

- A component serves one of the following roles

– A control component that coordinates the invocation of all other problem domain components

– A problem domain component that implements a complete or partial function that is required by the customer

– An infrastructure component that is responsible for functions that support the processing required in the problem domain

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Conventional View contd…

- software components are
- Conventional derived from the data flow diagrams (DFDs) in the analysis model

– Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy

– Control components reside near the top

– Problem domain components and infrastructure components migrate toward the bottom

– Functional independence is strived for between the transforms

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Conventional View contd…

- Once this is completed, the following steps are performed for each transform

1) Define the interface for the transform (the order, number and types of the parameters)
2) Define the data structures used internally by the transform
3) Design the algorithm used by the transform (using a stepwise refinement approach)

Process-related View

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:

– A complete description of their interface – The functions they perform

– The communication and collaboration they require

Designing Class-Based Components

- Component-level Design Principles
- 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

Designing Class-Based Components

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

– This principle says that inheritance should be well designed and well behaved. In any case a user should be able to instantiate an object as a subclass and use all the base class functionality invisibly.

Component-level Design Principles

- 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

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

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

- – 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., Floor plan)
- – Use infrastructure component names that reflect their implementation-specific meaning (e.g., Linked List)
- – Use stereotypes to identify the nature of components at the detailed design level (e.g., <<database>>)

Cohesion

- Cohesion “single-mindedness’ of a
- Cohesion is the 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

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

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Designing Conventional Components

- Conventional design constructs emphasize the Introduction maintainability of a functional/procedural program

– Sequence, condition, and repetition

• Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow

• Various notations depict the use of these constructs – Graphical design notation

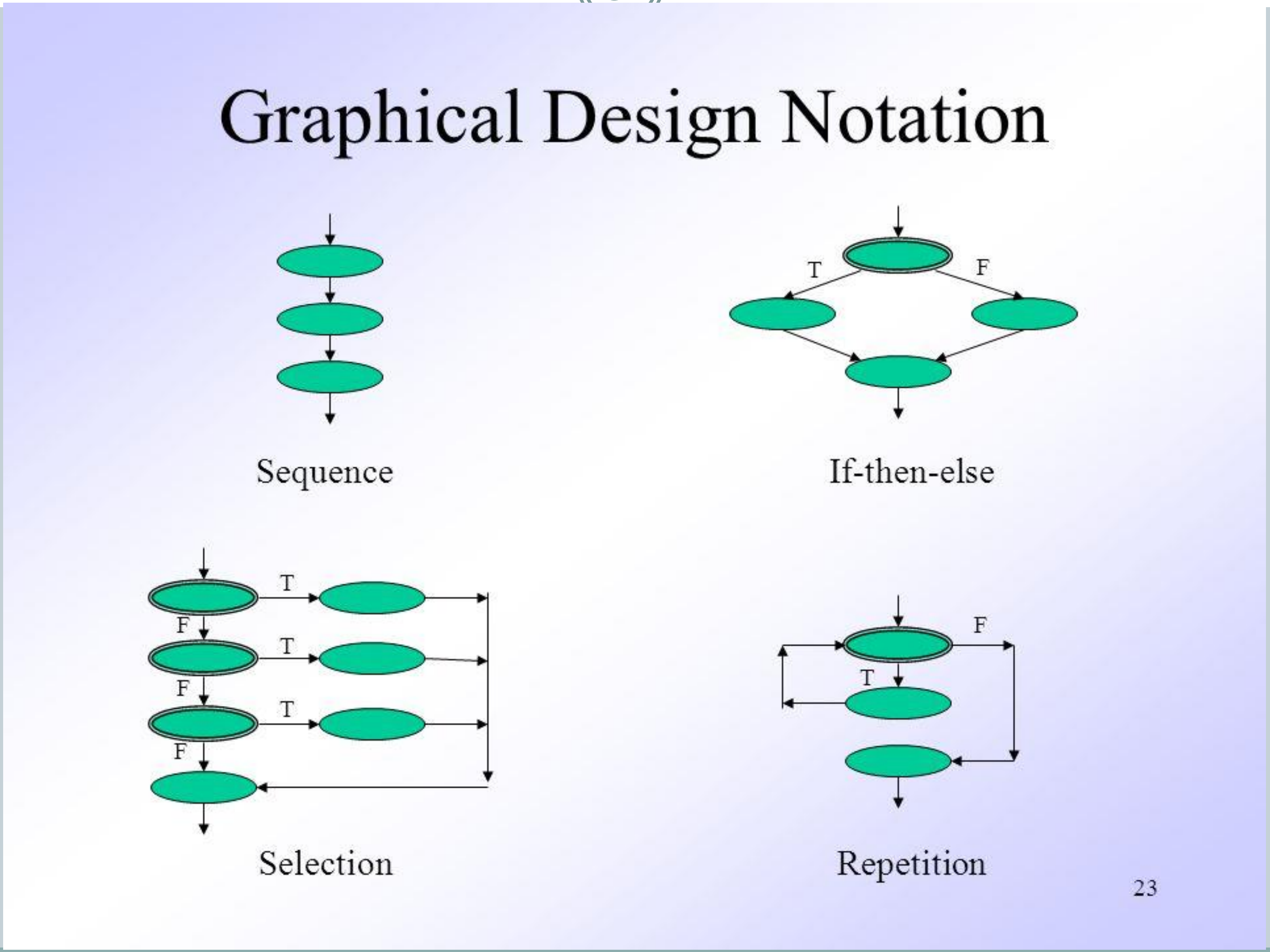
• Sequence, if-then-else, selection, repetition

– Tabular design notation

– Program design language

• Similar to a programming language; however, it uses narrative text embedded directly within the program statements

Graphical Design Notation


Graphical Design Notation

Sequence

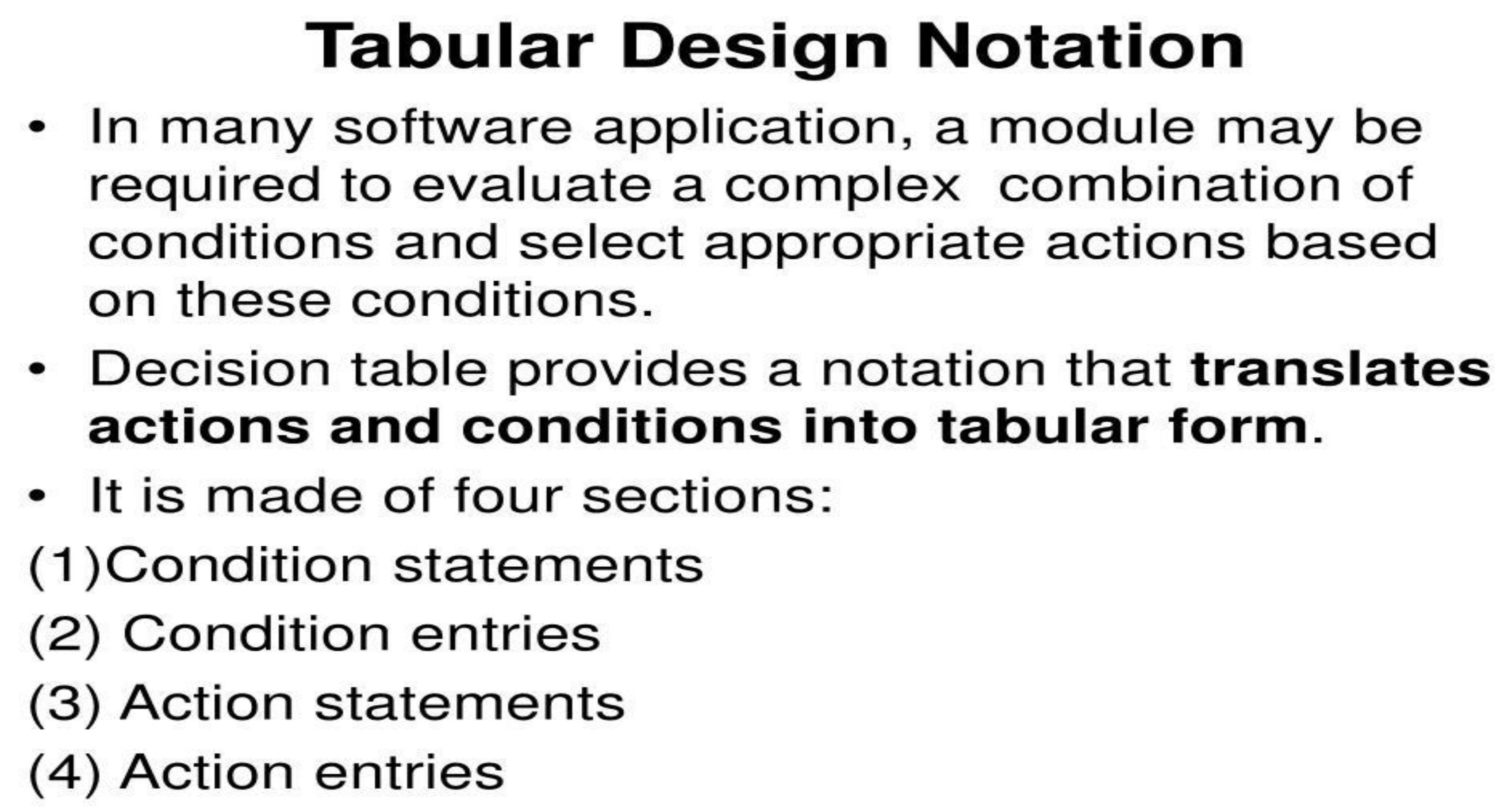
If-then-else

Selection

Repetition

23

Tabular Design Notation



Tabular Design Notation

- In many software application, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions.
- Decision table provides a notation that **translates actions and conditions into tabular form**.
- It is made of four sections:

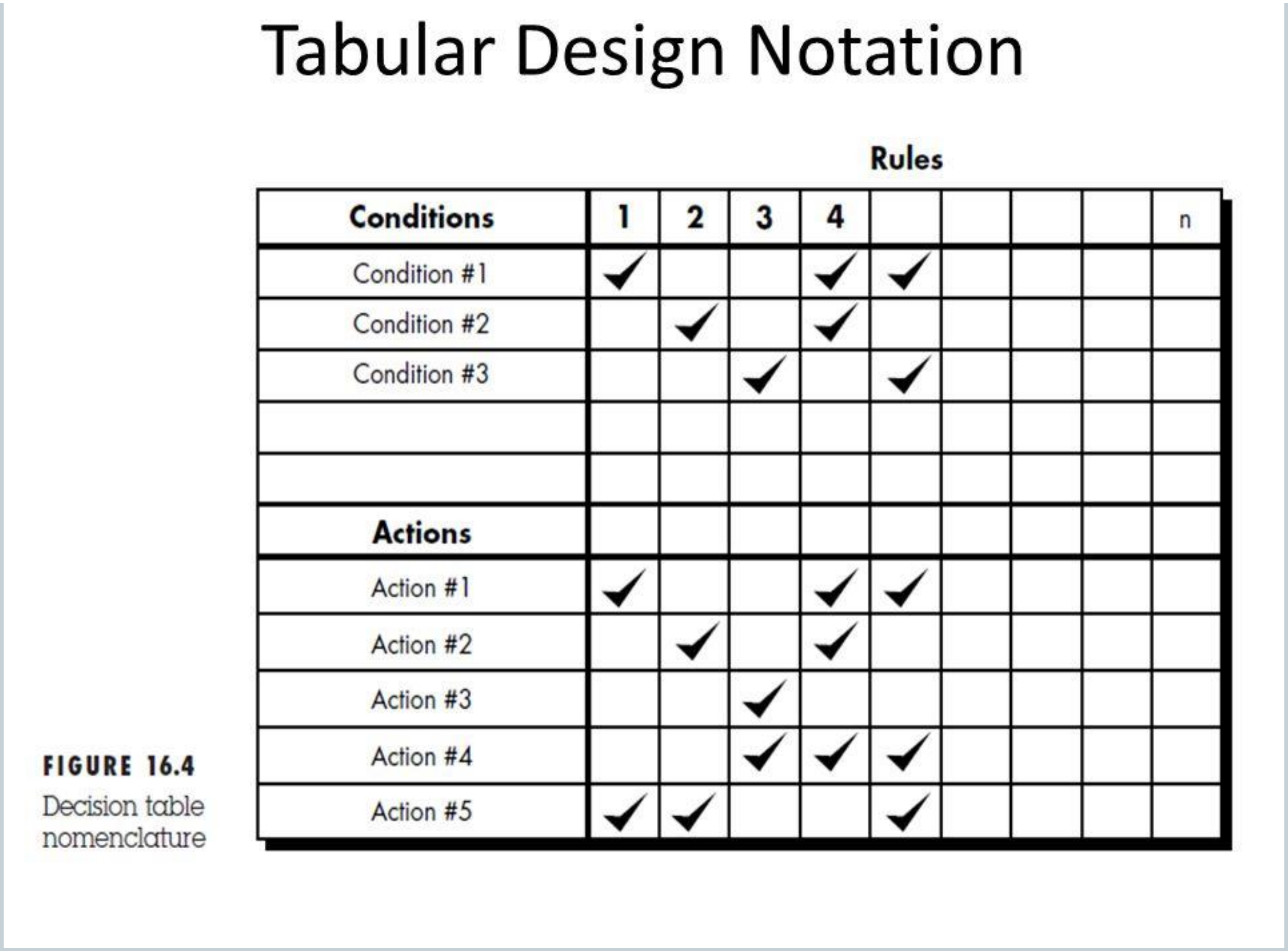
(1)Condition statements

(2) Condition entries

(3) Action statements

(4) Action entries

Tabular Design Notation



Tabular Design Notation

Rules

Conditions	1	2	3	4					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

FIGURE 16.4
Decision table nomenclature

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

COMPONENT-BASED DEVELOPMENT

- Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components."
- **Domain Engineering:** The intent of domain engineering is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

COMPONENT-BASED DEVELOPMENT

- **Component Qualification, Adaptation, and Composition:** Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in-house, others can be extracted from existing applications, and still others may be acquired from third parties. Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

COMPONENT-BASED DEVELOPMENT

- Analysis and Design for Reuse:
- Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, software quality assurance (SQA), and correctness verification methods all contribute to the creation of software components that are reusable.

COMPONENT-BASED DEVELOPMENT

- Classifying and Retrieving Components: Consider a large component repository. Tens of thousands of reusable software components reside in it.
- A reusable software component can be described in many ways, but an ideal description encompasses the 3C model—concept, content, and context. The concept of a software component is “a description of what the component does”. The interface to the component is fully described and the semantics—represented within the context of pre- and post conditions— is identified. The content of a component describes how the concept is realized. The context places a reusable software component within its domain of applicability.

Class-Based Component Design

- **Class-based component design** is a method for designing software components. It uses a class or classes (a collection of related data items, and the operations needed to manipulate those items) to represent the component in question. Consider for a moment the email application on your personal computer. One component of this system might be an email message. If we define a class called 'EmailMessage', then a data item might be 'message', and an operation that can be performed on 'message' might be 'SendMessage'. It's likely that this will have more data and operations defined, but you get the idea.

Designing Class based components, traditional Components

- An individual **software component** is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data).
- All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are *modular* and *cohesive*.

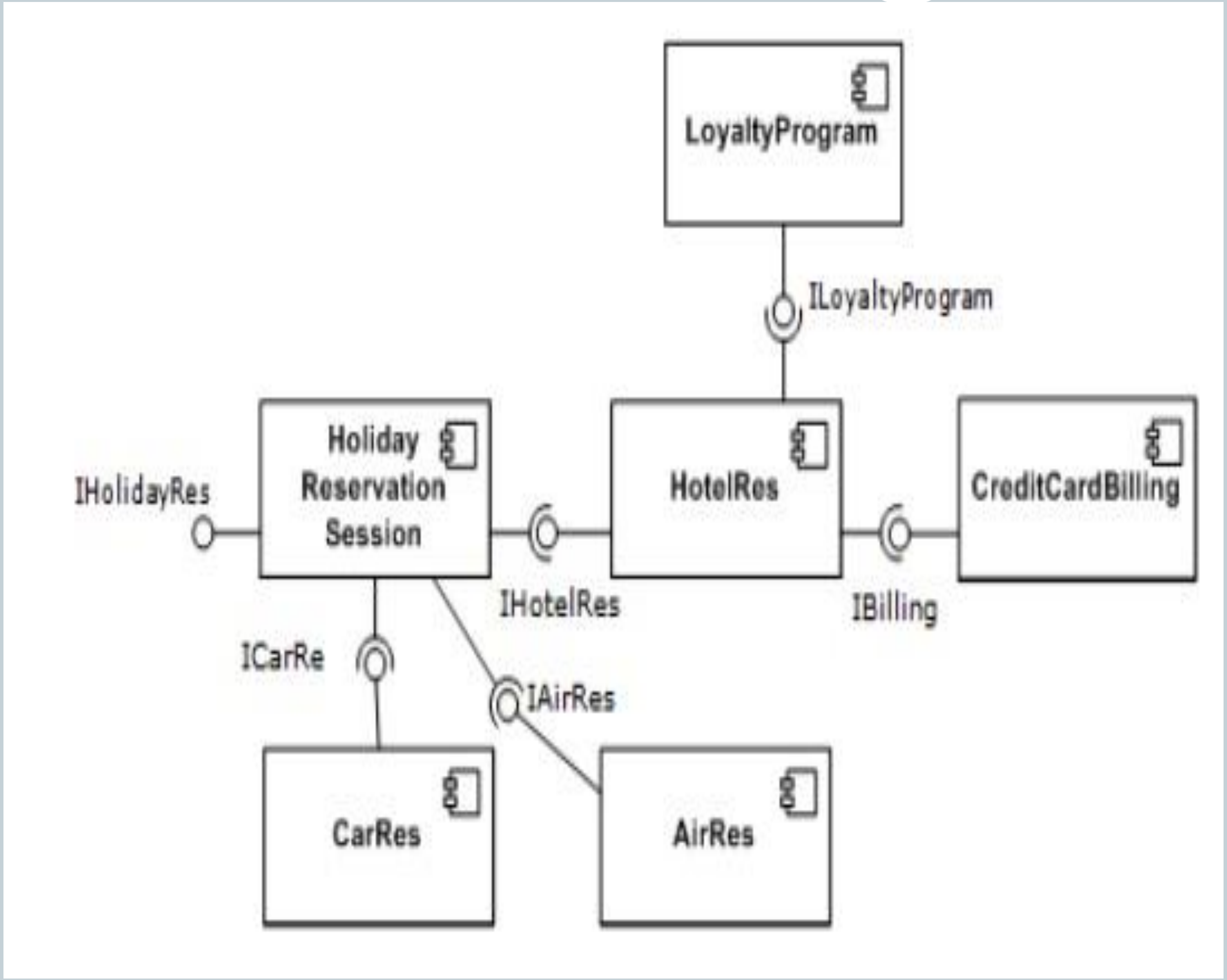
CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Designing Class based components, traditional Components

- With regard to system-wide co-ordination, components communicate with each other via *interfaces*. When a component offers services to the rest of the system, it adopts a *provided* interface that specifies the services that other components can utilize, and how they can do so. This interface can be seen as a signature of the component - the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components referred to as *encapsulated*. The UML illustrations within this article represent provided interfaces by a lollipop-symbol attached to the outer edge of the component.

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Designing Class based components, traditional Components



A simple example of several software components - pictured within a hypothetical holiday-reservation system represented in UML

Designing Class based components, traditional Components

- Another important attribute of components is that they are *substitutable*, so that a component can replace another (at design time or run-time), if the successor component meets the requirements of the initial component (expressed via the interfaces). Consequently, components can be replaced with either an updated version or an alternative without breaking the system in which the component operates.

Designing Class based components, traditional Components

- As a general rule of thumb for engineers substituting components, component B can immediately replace component A, if component B provides at least what component A provided and uses no more than what component A used.
- Software components often take the form of objects (not classes) or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some interface description language (IDL) so that the component may exist autonomously from other components in a computer.

Designing Class based components, traditional Components

- Reusability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them. Furthermore, component-based usability testing should be considered when software components directly interact with users.

Class-based design components Summary

- **The principles for class-based design component are as follows:**
- **Open Closed Principle (OCP)**
  Any module in OCP should be available for extension and modification.

  The Liskov Substitution Principle (LSP)
- The subclass must be substitutable for their base class.
- This principle was suggested by Liskov.
- **Dependency Inversion Principle (DIP)**
- It depends on the abstraction and not on concretion.
- Abstraction is the place where the design is extended without difficulty.
- **The Interface Segregation Principle (ISP)**
  Many client specific interfaces is better than the general purpose interface.

Class-based design components Summary

- **The principles for class-based design component are as follows:**

The Release Reuse Equivalency Principle (REP)

- A fragment of reuse is the fragment of release.
- The class components are designed for reuse which is an indirect contract between the developer and the user.
- **The common closure principle (CCP)**
  The classes change and belong together i.e. the classes are packaged as part of design which should have the same address and functional area.

  The Common Reuse Principle (CRP)
  The classes that are not reused together should not be grouped together.

Practice Question

Draw the Level 1 of DFD of Result management system

CHITKARA UNIVERSITY
HIMACHAL PRADESH - NAAC Accredited

Solution

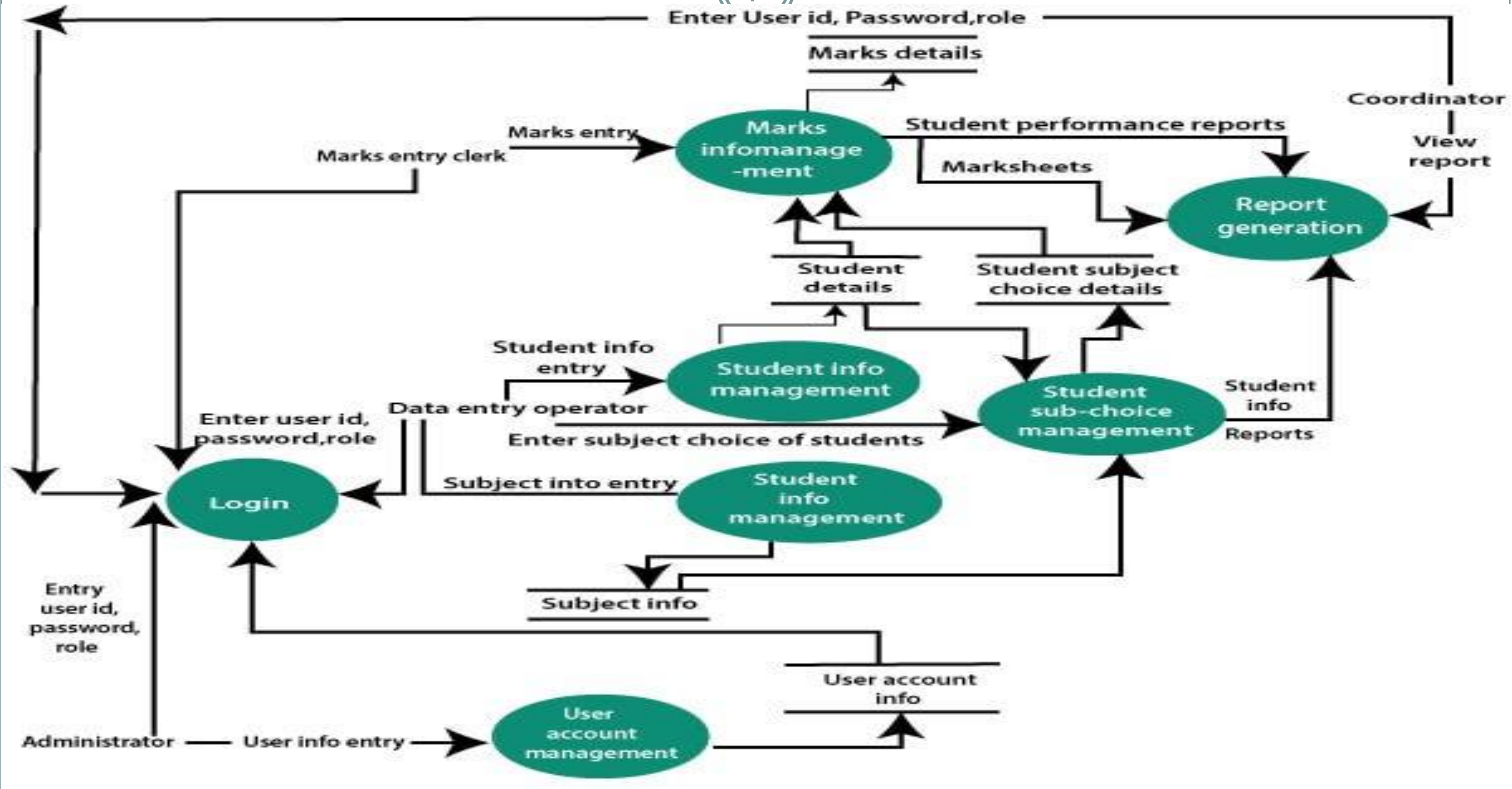


Fig: Level-1 DFD of result management system

THANK YOU