
# SOFTWARE DESIGN PRINCIPLES

**Dr Faiyaz Doctor**

**M19COM Software Development & Design**

# Topics Covered

- The role of Design
- Architectural Design
- Functional Design
- Functional Independence
- Heuristics
- Meyer's Criteria
- Meyer's Rules
- Meyer's Principles
- Introduction to UML

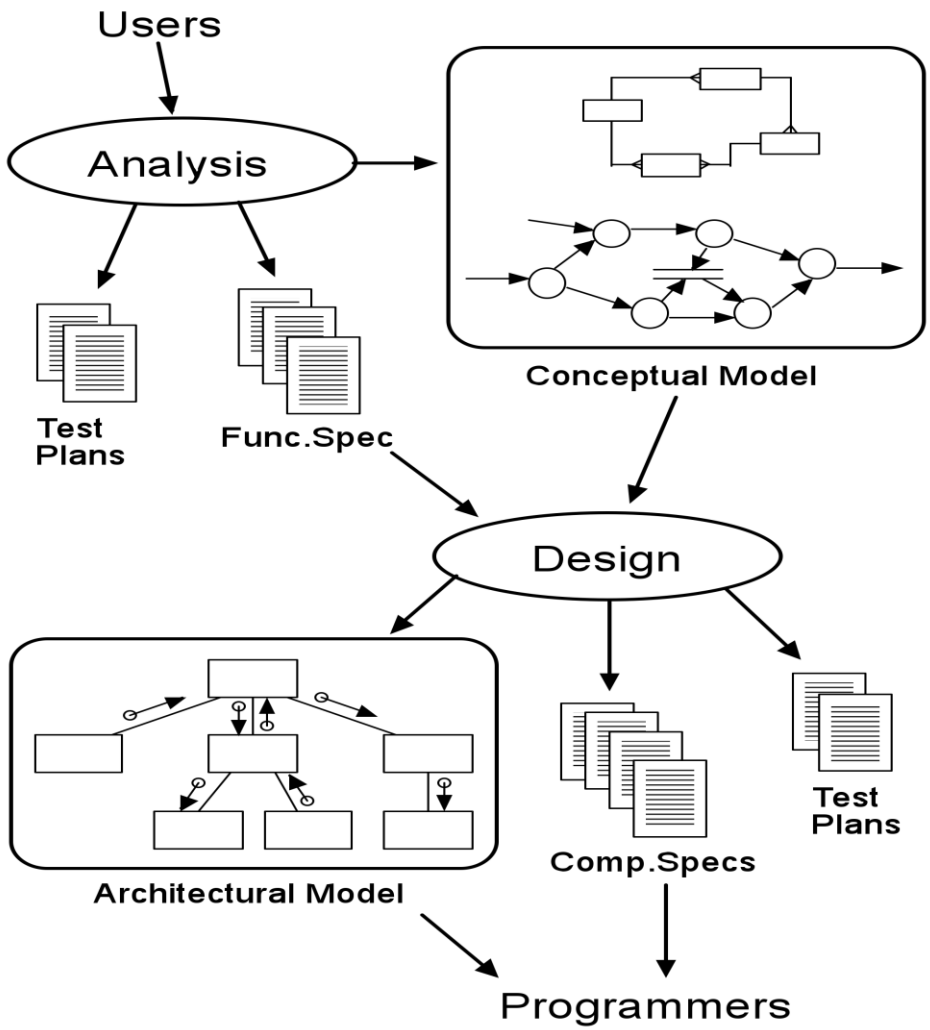
# The Role of Design

- A ***Design*** phase is at the centre of any software development process, whatever the lifecycle model.
- It is normally preceded by an ***Analysis*** phase and is followed by an ***Implementation*** phase.
- The Analysis phase determines **what** the system must do, producing a *Software Requirements Specification* which will normally include:
  - a *Conceptual Model* (system structure)
  - a *Functional Specification* (system behaviour)
  - user-level *Validation Criteria & Test Plans*

# The Role of Design

- The Design phase determines **how** the required functionality is to be achieved.
- The starting point is the specifications and models produced during Analysis. The final output will normally include:
  - an *Architectural Model* (software structure)
  - Design Class Diagrams, Interaction diagrams
  - detailed *Component Specifications*
  - system and component-level *Test Plans*

# The Role of Design




This diagram should not be taken too literally

# The Role of Design

## Analysis products:

- A Data Dictionary
- Entity Relationship Diagrams (ERDs)
- Data Flow Diagrams (DFDs)
- State Transition Diagrams (STDs)
- Process & Control Specs (PSPECs & CSPECs)

## Design products:

- A Control Structure Chart
- Interface Specifications

Architectural Model (systems modular / interaction composition defined as packages containing data and procedures).

- Data Structure Specifications
- Procedure Specifications

Important data using declarative notation, and algorithms using pseudo code.

# Architectural Design

- Determining software structure is the most important goal of the Design phase. A good architecture will make the system:
  - easier to understand, easier to implement, easier to test, easier to maintain, easier to modify

- ***Abstraction***
  - This refers to extracting essential features and ignoring (or delaying consideration of) inessential or lower-level detail. Can be applied to both data and procedures.

- ***Refinement***
  - This is the process of moving to a lower level of abstraction where new features are uncovered and elaborated. Applies to both data and procedures.

# Modularity

- This is the “purposeful structuring” of software in terms of components, e.g. Modula-2 *modules*, Ada *packages*, Java *classes*. Partitioning a large system into modules helps to reduce complexity (if done properly).

- A module should reflect a useful abstraction, i.e. have a well-defined role and contain the data and procedures relating to that role. This is called *Encapsulation*.

- A module should hide its internal details and provide an external interface which reflects its role and function. This is *Information Hiding*.

- A module should carry out its function with reference to input parameters and its own state; it may be fully self-contained or use subordinate modules, but should have no unexpected or hidden side-effects.

- A module should be capable of being developed and tested separately, or at least in conjunction with its subordinate modules in the control hierarchy.

# Functional Independence

- Abstraction, encapsulation and information hiding make modules “functionally independent”. Such modules have a well-defined single role/function and make limited use of other modules.

- Functional independence helps achieve the Architectural Design goals (see above). But how do we measure it? There are two related concepts:

- ***Cohesion***

  This is a measure of “connectedness” of software elements (data and procedures) within the module. A module which provides a single function and where all elements contribute to that function is *highly cohesive.*

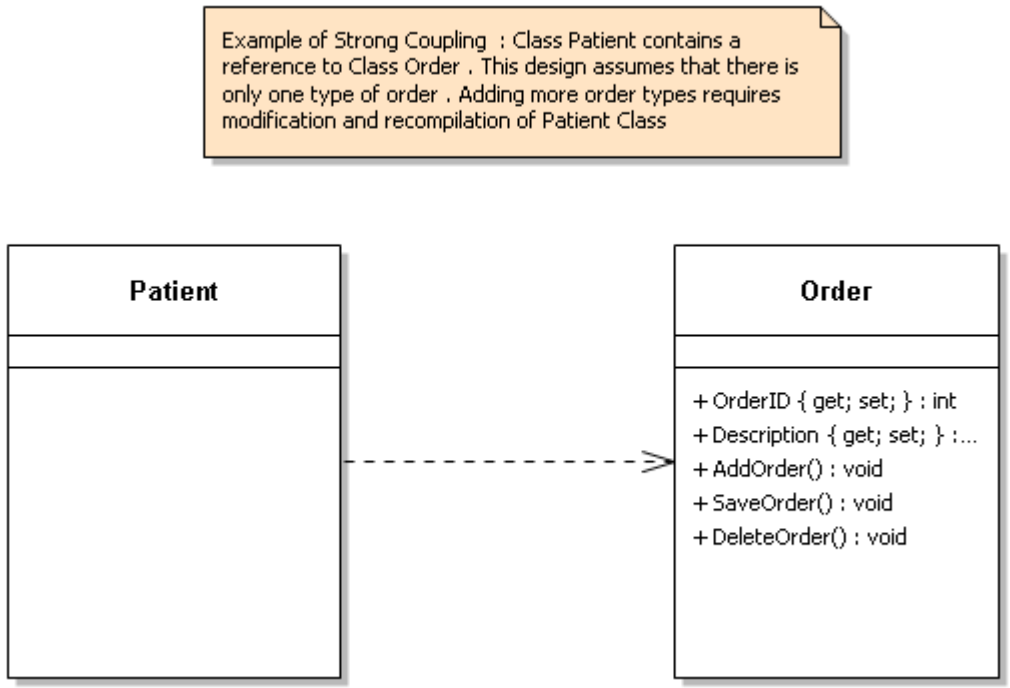
  Indicate the degree to which a class has a single, well-focused purpose.

- ***Coupling***

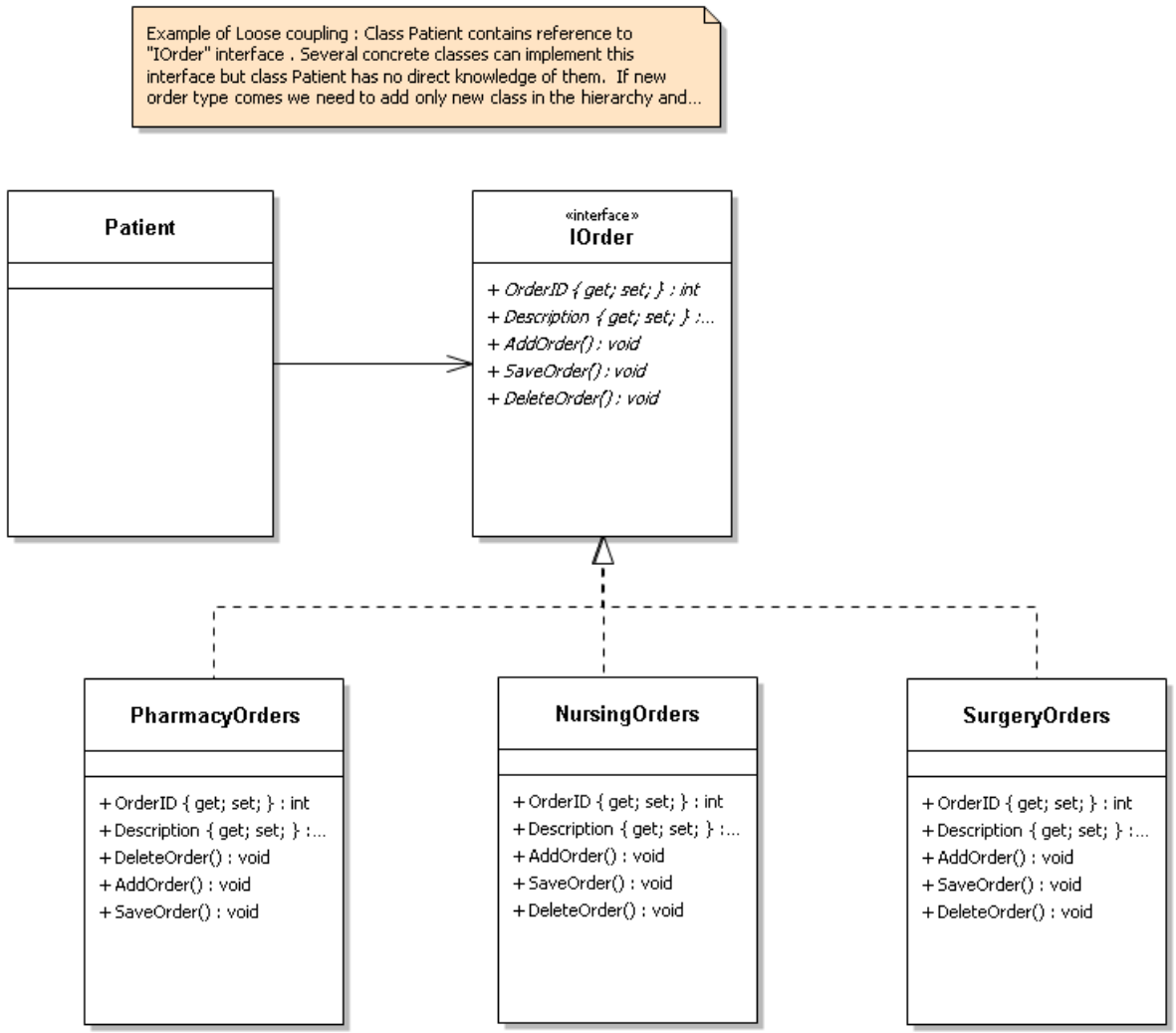
  This is a measure on connections between modules. A module which has a small interface allowing calls with simple parameter values is *loosely coupled,* whereas a module which allows direct access to its internal data is *tightly coupled.*

# Cohesion and Coupling Examples…

# Using Interfaces to Reduce Coupling



# Using Interfaces to Reduce Coupling



# Cohesion and Coupling

- Cohesion and Coupling are *categorised* rather than measured numerically (although that's also possible).

- Types of ***Cohesion*** include (from worst to best):

  - *Coincidental* — what's there by chance!
  - *Logical* — "similar" elements (e.g. all input functions)
  - *Temporal* — tasks done at same time (e.g. initialisation)
  - *Procedural* — tasks performed in some order
  - *Functional* — single-function module (e.g. CaIc_Tax)
  - *Abstract* — all features / functionality relate to a single abstraction (e.g. ADT)

# Cohesion and Coupling

- Types of ***Coupling*** include (from worst to best):
  - *Content*— direct access to internal data or code
  - *Common* — data access via common/global area
  - *External* — connection via files or environment
  - *Control* — parameters influence module decisions
  - *Stamp* — parameters are modifiable records
  - *Data* — parameters are simple values
  - *Abstract* — tightly-controlled ADT/class interface
- The use of abstraction leading to modules representing Abstract Data Types, or to classes encapsulating a data structure, leads to the “best” type of modularity.

# Meyer's Criteria

Bertrand Meyer has proposed the following criteria by which design methods can be judged:

- ***Decomposability*** — the facility with which a design method helps the designer to decompose a large problem into subproblems that are easier to solve
- ***Composability*** — the degree with which a design method enables program components (modules), once built, to be reused in creating new systems
- ***Understandability*** — the ease with which a each component can be understood without reference to other information or other modules
- ***Continuity*** — the ability to deal with a small change in a problem specification by making changes in just one or very few modules
- ***Protection*** — the ability to limit the propagation of side effects if an error occurs in a given module

# Meyer's Rules

From the preceding criteria, Meyer suggests the following design rules:

- ***Direct Mapping*** — the modular structure should reflect the main entities in the analysis model
- ***Few Interfaces*** — every module should communicate with as few others as possible
- ***Small Interfaces*** — if two modules communicate, they should exchange as little information as possible
- ***Explicit Interfaces*** — if two modules communicate, this must be obvious from the text of at least one of them
- ***Information Hiding*** — the designer must decide which features of a module are its "public" interface; the rest should be hidden and inaccessible

# Meyer's Principles (based on Eiffel language)

From the preceding Meyer derives the following design principles:

- ***Linguistic Modularity*** — "modules" must be directly representable in the modeling or programming language to be used
- ***Self-Documentation*** — the module's text should contain all information about the module
- ***Uniform Access*** — the notation through which a module's services are accessed should be uniform, and not betray its implementation (e.g. data/function)
- ***Open-Closed*** — modules should be "open" to extension but "closed" to unauthorized change
- ***Single Choice*** — if a set of alternatives is supported, only one module should have complete information

# Approaches to Software Design

- ***Object-Oriented***

  These methods are quickly gaining ground over “structured” approaches because of several important benefits they bring:

  - scalability (can be applied to small or large problems)
  - stability (models cope with requirements changes)
  - distribution (easier to map to networked systems)
  - reusability (software is easily reused or adapted)
  - standardisation (UML has unified modelling notations)

# Unified Modelling Language: UML

- Proposed and developed by the “three amigos”:

  Grady Booch (“the Booch method”)
  James Rumbaugh (OMT)
  Ivar Jacobson (OOSE)

- Booch and Rumbaugh started the work at Rational in 1994, trying to combine the Booch and OMT methods into a “Unified Method”. Jacobson joined them in 1995, but the three then concentrated on defining a *modelling language* rather than a single method.

- Version 1.0 of UML was submitted to OMG in 1997. now version 2.0.

- UML has been enthusiastically adopted by developers and is already supported by many CASE tools Visual Paradigm and Rational Rose.

# UML cont…

**Goals of UML**

- to model systems (and not just software) using object-oriented concepts
- to establish an explicit coupling to conceptual as well as executable artifacts
- to address the issues of scale inherent in complex, mission-critical systems
- to create a modelling language usable by both humans and machines

***But note that UML is not a software development method!***

- Rather, it provides a set of well-defined notations for representing the analysis and design models produced using almost any object-oriented method.
- Knowing UML allows you to understand such models; but it does not by itself help you create new models. To do that you need to apply some appropriate method.

# Generic View of Object Oriented Software Development with UML

- The OO approach works best with an *iterative* lifecycle (process) model. This type of model is also described as “incremental”, “evolutionary”, or “spiral” (although the words emphasise different aspects of the model).
- Other descriptive phrases are “prototyping cycles” and “rapid application development” (RAD).
- The basic tool of UML that drives iterations is the *Use Case*: each iteration should attempt to realise a specific Use Case (or small set of related Use Cases) from the requirements or analysis models.
- The initial set of high-level Use Cases is derived during the requirements and/or prototyping phases.
- Each iteration will itself have a “mini-lifecycle” involving suitable planning, analysis, design, coding and testing activities. Ideally, the outcome from each iteration will be a usable increment (new functionality) of the system.

# Selection of Object Oriented Methods

- **CRC (=Class/Responsibility/Collaborators):** an early approach to defining class roles & relationships
- **Coad & Yourdon:** another early OOA/OOD method
- **HOOD (=Hierarchical Object-Oriented Design):** based on successive refinement of objects
- **Fusion:** combining and improving earlier methods
- **BON (=Business Object Notation):** uses static & dynamic models and CRC-style cards
- **Booch:** an incremental/evolutionary approach to system modelling using several different views
- **OMT (=Object Modeling Technique):** Rumbaugh's work uses techniques for describing static, dynamic & functional models of a system
- **OOSE (=Object-Oriented Software Engineering):** Jacobson's method based on Use Case analysis