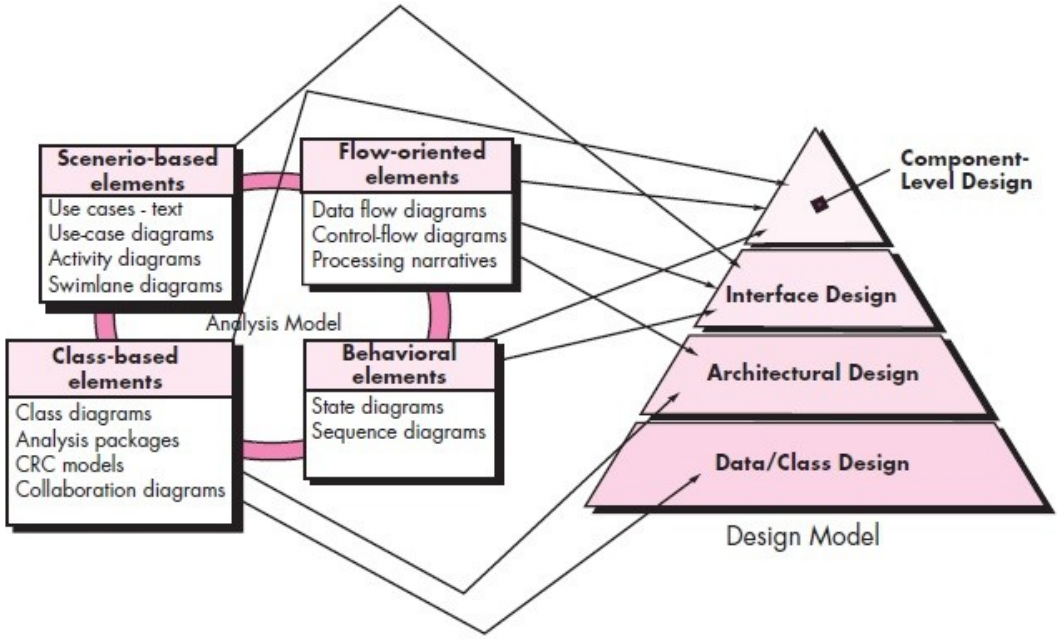
Design within the Context of Software Engineering

- Beginning once software requirements have been analyzed and modeled, **software design is the last software engineering action within the modeling activity and sets the stage for construction**



- Each of the elements of the **requirements model provides information that is necessary to create the four** design models required for a complete specification of design.

- The requirements model, **manifested by scenario-based, class-based,**
flow-oriented, and behavioral elements, feed the design task.

- Using design notation and design methods, **design produces a data/class design, an architectural design, an interface design, and a component design**.

- The data/class design **transforms class models into design class** realizations and the requisite data structures required to implement the software.

- Part of **class design may occur in conjunction with the design of software architecture**.

- More d**etailed class design occurs as each software component is designed**.

- The architectural design **defines the relationship between major structural elements of the software**, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

- The architectural design representation—the framework of a computer-based system—is derived from the requirements model.

- The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it.

Design Process

- **Software design is an iterative process through which requirements are translated into a "blueprint"** for constructing the software.

- Initially, **it depicts a holistic view of software means** design is represented at a **high level of abstraction**—

- A level that can be directly traced to the specific system objective and more **detailed data, functional, and behavioural** requirements.

- As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction.

Software Quality Guidelines and Attributes

Quality Guidelines.

In order to evaluate the quality of a design representation, **members of the software team must establish technical criteria for good design**.

1. A design should **exhibit an architecture** that
(1) has been created **using recognizable architectural styles** or patterns, (2) is **composed of components that exhibit good design** characteristics
(3) can be **implemented in an evolutionary fashion**,2 thereby
facilitating implementation and testing.

2. A **design should be modular**; that is, the software should be logically **partitioned into** elements or subsystems.

3. A **design should contain distinct representations of data**, architecture,
interfaces, and components.

4. A **design should lead to data structures that are appropriate for the classes** to be implemented and are drawn from recognizable data patterns.

5. A design should **lead to components that exhibit independent functional characteristics**.

6. A design should **lead to interfaces that reduce the complexity** of connections between components and with the external environment.

7. A design **should be derived using a repeatable method** that is driven by information obtained during software requirements analysis.

8. A design **should be represented using a notation that effectively communicates** its meaning.

Quality Attributes.
Set of software quality attributes that has been given the acronym **FURPS—functionality, usability, reliability, performance, and supportability**.

• ***Functionality*** is assessed by evaluating the **feature set and capabilities** of the
program, the generality of the functions that are delivered, and the security of
the overall system

• ***Usability*** is assessed by **considering human factors** overall
aesthetics, consistency, and documentation.

• *R**eliability* **is evaluated by measuring the frequency and severity of failure**, the accuracy of output results, the mean-time-to-failure (MTTF), **the ability to
recover from failure, and the predictability of the program**.

• ***Performance*** is measured by considering **processing speed, response time,
resource consumption, throughput, and efficiency**.

• ***Supportability*** combines the ability to extend the program (extensibility),
adaptability,
serviceability—these three attributes represent a more common term, ***maintainability***—and in addition, testability, compatibility, configurability

The Evolution of Software Design

- The **evolution of software design is a continuing process** that has now spanned almost six decades.

- Early design work **concentrated on criteria for the development of modular programs and methods for refining software structures** in a topdown manner

- **Procedural aspects** of design definition evolved into a philosophy

called ***structured programming***

- **Later work proposed methods for the translation of data flow or data structure** into a design definition.

- **Newer design approaches proposed an object-oriented approach** to design derivation.

- **More recent emphasis in software design has been on software architecture and the design patterns** that can be used to implement software architectures and lower levels of design abstractions.

- Growing emphasis on **aspect-oriented methods, model-driven development , and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created**.

- A number of design methods, are being applied throughout the industry.

  Ex: (1) a mechanism for the **translation of the requirements model into a design representation**,
  (2) a notation **for representing functional components and their interfaces**,
  (3) heuristics for refinement and partitioning, and
  (4) **guidelines for quality assessment.**

DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the history of software engineering.

Abstraction

- **When we consider a modular solution to any problem, many levels of abstraction can be posed**.

- At the **highest level of abstraction, a solution is stated in broad terms**
  using the language of the problem environment.

- **At lower levels of abstraction, a more detailed description** of the solution is provided.

- As **different levels of abstraction** are developed, it have work to create **both procedural and data abstractions**.

- A ***procedural abstraction*** **refers to a sequence of instructions**
  that have a specific and limited function.
  - **The name of a procedural abstraction implies these functions, but specific details are suppressed**.
  - An **example** of a procedural abstraction would be the word ***open*** **for a door**. ***Open*** **implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)**

- **A *data abstraction*** is a named collection of data that describes a data object.

- In the context of the procedural abstraction *open,* we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

- It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door.**

Architecture

- *Software architecture* means to "**the overall structure of the software and the ways in which that structure provides conceptual integrity for a system**"

- Architecture is the structure or organization of program components (modules**), the manner in which these components interact, and the structure of data that are used by the components**.

- components can be **generalized to represent major system elements and their interactions.**

- **One goal of software design is to derive an architectural version of a system**. **This version serves as a framework from which more detailed design activities are conducted**.

- **A set of architectural patterns enables a software engineer to solve**
  common design problems.

- A set of **properties** that should be specified as part of an architectural design:
  - **Structural properties**
  - **Extra-functional properties**
  - **Families of related systems**

.

- **He architectural design can be represented using various models**

- **Structural Models**
- **Framework model**
- **Dynamic models**
- **Process Model**

- **Architectural Discretional Languages (ADLs) have been developed**
  to represent these models

- Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another

Patterns

- A pattern conveys the essence of **a proven solution to a recurring problem** within a certain context

- Stated in another way, a design pattern **describes a design structure that solves a particular design problem within a specific context**

- It **enables a designer to determine**
  (1) **whether the pattern is applicable to the current work**,
  (2) **whether the pattern can be reused** (hence, saving design time), and
  (3) **whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern**

Separation of Concerns

- Separation of concerns is a design concept, **suggests that any complex**
  problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.

- **A concern is a feature or behavior that is specified as part of the requirements model for the software.**

- By **separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve**.

- Ex:For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2.

- As by using this, **it does take more time to solve a difficult problem**.

- It also follows that **the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately**.

- It is nothing but leads to **a divide-and-conquer strategy**—it's easier to solve a complex problem when you break it into manageable pieces.

- This has important implications with regard to software modularity.

- Separation of concerns is manifested in other related design concepts: modularity, aspects, functional independence, and refinement.

Modularity

- Modularity **is the most common manifestation of separation of concerns**.

- **Software is divided into separately named and addressable components**, sometimes called modules, that are integrated to satisfy problem requirements.

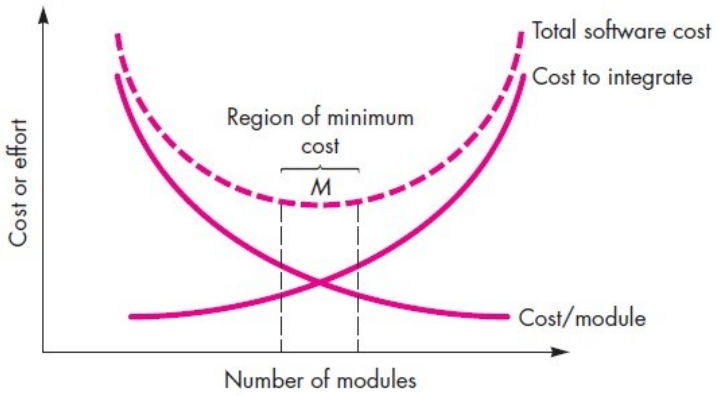
- It has been stated that **"modularity is the single attribute of software that allows a program to be intellectually manageable"**.

- **Monolithic software (a single module) cannot be easily grasped by a software engineer. complexity would make understanding close to impossible**.

- **In almost all instances, you should break the design into many modules**, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.

- **If subdivide software indefinitely the effort required to develop it will become negligibly small**!

- Referring to Figure the effort (cost) to develop an individual
  software module does decrease as the total number of modules increases.



Modularity and software cost

- **However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.**

Information Hiding

- The principle of **information hiding suggests that modules be "characterized by design decisions that each hides from all others**.

- The **modules should be specified and designed so that information contained within a module is inaccessible to other modules** that have no need for such information.

- Hiding implies that **effective modularity can be achieved by defining a set of independent modules** that communicate with one another only that information necessary to achieve software function.

- **Abstraction helps to define the procedural (or informational) entities that make up the software**.

- **Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module**

- The use of **information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance**.

Functional Independence

- The concept of functional independence **is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding**.

- **Functional independence is achieved by developing modules with single minded function and an avoiding excessive interaction with other modules**.

- **Design software so that each module addresses a specific subset of requirements** and has a simple interface when viewed from other parts of the program structure.

- **Software with independent modules, is easier to develop because** function can be compartmentalized and **interfaces are simplified**

- **Independent modules are easier to maintain because secondary effects caused by design or code modification are limited**

- **To summarize, functional independence is a key to good design,** and design is the key to software quality.

- **Independence is assessed using two qualitative criteria: cohesion and coupling**.
  Cohesion is an indication of the **relative functional strength of a module**.
  Coupling is an indication of the **relative interdependence among modules**.

Refinement(decomposition)

- **Stepwise refinement is a top-down design** strategy
- A **program is developed by successively refining levels** of procedural detail.
- A **hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached**.
- Refinement **is actually a process of *elaboration***.
- The statement **describes function conceptually but provides no information about the internal workings of the function**s
- **Elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs**.
- **Abstraction and refinement are complementary to the each other**.
- Refinement helps to reveal low-level details as design progresses.

Aspects

- **As design begins, requirements are refined into a modular design representation**.
- **It is important to identify aspects so that the design can properly accommodate them as refinement and modularization occur**.
- In an ideal context, **an aspect is implemented as a separate module rather than as software fragments that are "scattered" or "tangled" throughout many components**.
- To accomplish this, the design architecture should support a mechanism for defining an aspect
- A module that enables the concern to be implemented across all other concerns that it crosscuts

Refactoring

- Refactoring is **a reorganization technique that simplifies the design of a component without changing its function or behavior**.

- Fowler defines refactoring in the following manner "**Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."**

- **When software is refactored, the existing design is examined for redundancy, unused design elements,**

inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

- **For example**, **a first design iteration might yield a component that exhibits low cohesion** .
- **After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion**.
- **The result will be software that is easier to integrate, easier to test, and easier to maintain**.
- Intent of refactoring is to modify the code in a manner that does not alter its external behavior.

Object-Oriented Design Concepts and Design Classes

- The object-oriented (OO) paradigm **is widely used in modern software engineering.**
- It has been **provided with design concepts such as classes and objects, inheritance, messages, and polymorphism, among others**
- **The requirements model defines a set of analysis classes**.
- **Each describes some element of the problem domain, focusing on aspects of the problem that are user visible**.
- **The level of abstraction of an analysis class is relatively high.**

Five different types of design classes, each representing a different layer of the
design architecture,

- ***User interface classes*** define all abstractions that are necessary for **human computer interaction (HCI**). In many cases, HCI occurs within the context of
- ***Business domain classes*** are often refinements of the analysis classes defined earlier. **The classes identify the attributes and services (methods) that are required to implement some element of the business domain**.
- ***Process classes* implement lower-level business abstractions** required to fully manage the business domain classes.
- ***Persistent classes* represent** data stores (e.g., a **database**) **that will persist beyond the execution of the software.**
- ***System classes*** implement software management and control functions that **enable the system to operate and communicate within its computing environment and with the outside world**

  Characteristics of a design classes
  - ✓ **Complete and sufficient**
  - ✓ **Primitiveness**
  - ✓ **High cohesion**
  - ✓ **Low coupling**

Design Model

- The design model can be viewed in two different dimensions

- The ***process dimension* indicates the evolution of the design model as design tasks are executed as part of the software process**.

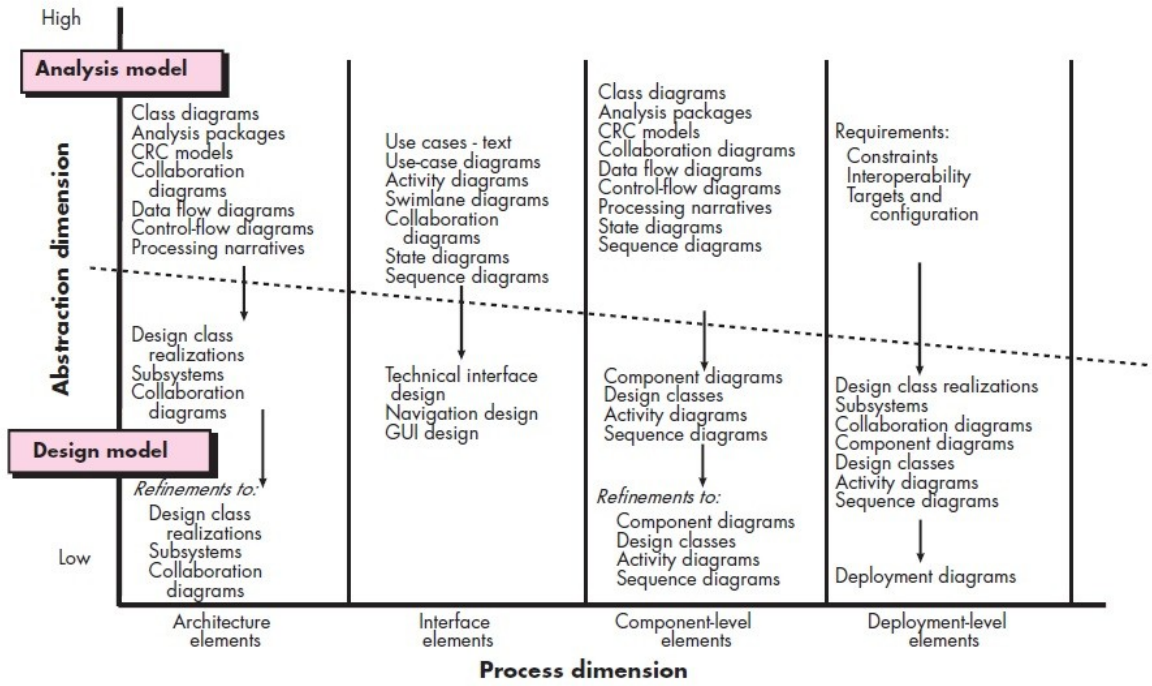
- The ***abstraction dimension* represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively**. As shown fig

- The **dashed line** indicates the boundary between the analysis and design models.

- **In some cases, a clear distinction between the analysis and design models is possible. In other cases, the analysis model slowly blends into the design and a clear distinction is less obvious**.

- **The elements of the design model use many of the same UML diagrams that were used in the analysis model**.

- The difference is that these diagrams are refined and elaborated as part of design; more implementation-specific detail is provided,



Data Design Elements:

- Like other software engineering activities, **data design creates a model of data and/or information that is represented at a high level of abstraction**

- This data model refined into progressively more implementation-specific representations that can be processed by the computer-based system.

- In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

- **The structure of data has always been an important part of software design**.

- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.

- **At the application level, the translation of a data model-into a database is pivotal to achieving the business objectives of a system**.

- At the business level, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining
  or knowledge discovery that can have an impact on the success of the business itself.

Architectural Design Elements

- The architectural design for software is the equivalent to the **floor plan of a house.**
- The floor plan **depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms**.

- The floor plan gives us **an overall view of the house**. Architectural design **elements give us an overall view of the software**.

- The architectural model **is derived from three sources**:
  (1) **information** about the application domain for the software to be built;
  (2) specific **requirements model elements** such as data flow diagrams or
  analysis classes, their relationships and collaborations for the problem at hand; and
  (3) **the availability of architectural styles and patterns**

- **The architectural design element is usually depicted as a set of interconnected subsystems**, often derived from analysis packages within the requirements model.

- **Each subsystem may have it's own architecture** (e.g., a graphical user interface might be structured according to a pre existing architectural style for user interfaces).

Interface Design Elements

- **The interface design for software is analogous to a set of detailed drawings**.
  Example: Interface design for House, a set of detailed drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone)come into the house and are distributed among the rooms depicted in the floor plan.

- **The interface design elements for software depict information flows into and out of the system** and how it is communicated among the components defined as part of the architecture.

- There are **three important elements** of **interface design**:
  (1) the **user interface (UI);**
  (2) **external interfaces to other systems**, devices, networks, or other
  producers or consumers of information; and
  (3) **internal interfaces between various design components**.

- **It allows the software to communicate externally and enable internal communication and collaboration among the components**

- UI design (increasingly called usability design) is a major software engineering action, the

- Usability design **incorporates aesthetic elements e.g., layout, color, graphics, interaction mechanisms**,

- Usability design incorporates **ergonomic elements** e.g., information layout and placement, metaphors, UI navigation,

- Usability design incorporates **technical elements** (e.g., UI patterns, reusable components).

- In general, the UI is a unique subsystem within the overall application architecture.

- The design of external interfaces requires definitive information about the entity to which information is sent or received.

- In every case, this information should be collected during requirements engineering and verified once the interface design commences.

- The design of external interfaces **should incorporate error checking and some security features**.

- The **design of internal interfaces is closely aligned with component-level design**

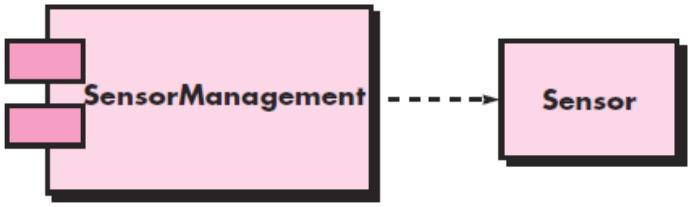
- Design realizations of analysis classes represent all operations and the messaging schemes required to enable communication and collaboration between operations in various classes.

Component-Level Design Elements

**The component-level design for software fully describes the internal
detail of each software component**.

the component-level design **defines data structures for all local data objects** and algorithmic detail

Within the context of object-oriented software engineering, a component is represented in UML



diagramatic form

In this figure, a component named Sensor Management (part of the SafeHome security function) is represented

A dashed arrow connects the component to a class named Sensor that is assigned to it

The design details of a component can be modeled at many different levels of abstraction

A **UML activity diagram can be used to represent processing logic**

Detailed procedural flow for a component or some other diagrammatic form e.g., flowchart or box diagram

Data structures, selected based on the nature of the data objects to be processed, are usually modeled using the programming language to be used for implementation

> **Example-** The component-level design for software is the equivalent to a set of detailed drawings and specifications for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room.

Deployment- Level Design Elements

- It **Indicates, how software functionality and subsystems will be allocated within the physical computing environment that will support the software**.

- For example, the elements of the any software product are configured
  to operate within three primary computing environments

- During design, a UML deployment diagram is developed and then refined as Subsystems (functionality) housed within each computing element

- An external access subsystem has been designed to manage all attempts to access the main system from an external source.

- Each subsystem would be elaborated to indicate the components that it implements.

- **The deployment diagram shows the computing environment but does not explicitly indicate configuration details**.

- **These details are provided when the deployment diagram is revisited in instance form during the latter stages of design or as construction begins**.

- **Each instance of the deployment is identified**

Designing Class-Based Components

Basic Design Principles

- Four basic design principles are applicable to component-level design and have been widely adopted

- These principles can use as a guide as each software component is developed.

The Open-Closed Principle (OCP).

- "A **module should be open for extension but closed for modification**" This represents one of the most important characteristics of a good component-level design.

- Specify the component in a way that **allows it to be extended without the need to make internal modifications to the component itself**.

- Create abstractions that serve as a buffer between the functionality and the design class itself.

The Liskov Substitution Principle (LSP).

- "**Subclasses should be substitutable for their base classes**"

- This design principle, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.

- LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it.

- A **precondition that must be true before the component uses a base class**

- A **postcondition that should be true after the component uses a base class**.

- To create derived classes, be sure they conform to the pre- and postconditions.

Dependency Inversion Principle (DIP).

- “**Depend on abstractions Do not depend on concretions**”.

- **The more a component depends on other concrete components the more difficult it will be to extend**.

The Interface Segregation Principle (ISP).

- “**Many client-specific interfaces are better than one general purpose interface**” There are many instances in which multiple client components use the operations provided by a server class.

- ISP suggests that you should create a specialized interface to serve each major category of clients.

- **Only those operations that are relevant to a particular category of clients should be specified in the interface for that client**.

- **If multiple clients require the same operations, it should be specified in each of the specialized interfaces**.

The Release Reuse Equivalency Principle (REP).

- “**The granule of reuse is the granule of release**”

- When classes or components are designed for reuse, there is an implicit contract that is established between the **developer of the reusable entity and the people who will use i**t.

The Common Closure Principle (CCP).

- **“Classes that change together belong together.”**

- Classes should be packaged cohesively.

- when classes are packaged as part of a design, they should address the same functional or behavioural area.

- When some characteristic of that area must change, it is likely that only those classes within the package will require modification.

The Common Reuse Principle (CRP).

- **“Classes that aren’t reused together should not be grouped together”**

- When one or more classes within a package changes, the release number of the package changes.

- All other classes or packages that rely on the package that has been changed must now update to the most recent release of the package and be tested

- If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed.

.

Component-Level Design Guidelines

Components.

- Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.

- Infrastructure components level classes should be named to reflect
  implementation-specific meaning.

- can choose to use stereotypes to identify the nature of components at
  the detailed design level.
  For example, <<infrastructure>> to identify an infrastructure component,
  <<database>> could be used to identify a database
  <<table>> can be used to identify a table within a database.

Interfaces.

- Interfaces provide important information about communication and
  collaboration

- Interface helping us to achieve the OCP.

- However, unfettered representation of interfaces tends to complicate component diagrams.

- Interface are intended to simplify the visual nature of UML component diagrams.

Dependencies and Inheritance.

- For improved readability, it is a good idea to model dependencies from left to right And inheritance from bottom (derived classes) to top (base classes).

- **The component interdependencies should be represented via interfaces**,

- It will help to make the system more maintainable.

Cohesion

- cohesion as the “single-mindedness” of a component.

- **Within the context of component-level design for object-oriented systems, cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another**

Different types of cohesion

Functional.

- Exhibited primarily by **operations**,

- this level of cohesion occurs when a component **performs a targeted computation and then returns a result**.

Layer.

- Exhibited by **packages, components, and classes**,

- this type of cohesion occurs when **a higher layer accesses the services of a lower layer, but lower layers do not access higher layers**.

Communicational.

- **All operations that access the same data are defined within one class**.
- Such classes focus solely on the data in question, accessing and storing it.
- Classes and components that **exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain**.

Coupling

- Coupling is a qualitative measure of the degree to which classes are connected to one another.
- **As classes become more interdependent, coupling increases**.
- **An important objective in component-level design is to keep coupling as low as is possible**

Class coupling can manifest itself in a variety of ways.

- **Content coupling**.
  It Occurs when one component modifies data that is internal to another component
- **Common coupling**.

It occurs when a number of components all make use of a global variable.

Although this is sometimes necessary common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

- **Control coupling**.
  It occur when operation A() invokes operation B() and passes control flag to B. The control flag then “directs” logical flow within B.

  The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes.

- **Stamp coupling**.
  It occurs when ClassB is declared as a type for an argument
  of an operation of ClassA.

  Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

- **Data coupling.**
  Occurs when operations pass long strings of data arguments.
  The “bandwidth” of communication between classes and components
  grows and the complexity of the interface increases.
  Testing and maintenance are more difficult.

- **Routine call coupling**.
  It occurs when one operation invokes another.

  This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

- **Type use coupling**.

It Occurs when component A uses a data type defined in component B

If the type definition changes, every component that uses the definition must also change.

- **Inclusion or import coupling**.
  It occurs when component A imports or includes a package or the content of component B.

- **External coupling**.
  It occurs when a component communicates or collaborates with infrastructure components

  Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.