Unit-3

Software Design Principles and Best Practices

Modularization

- Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s)independently. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Advantages of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be reused again
- Concurrent execution can be made possible

Coupling and Cohesion

- When a software program is modularized, its tasks are divided into several modules based on some characteristics. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called **coupling and cohesion.**

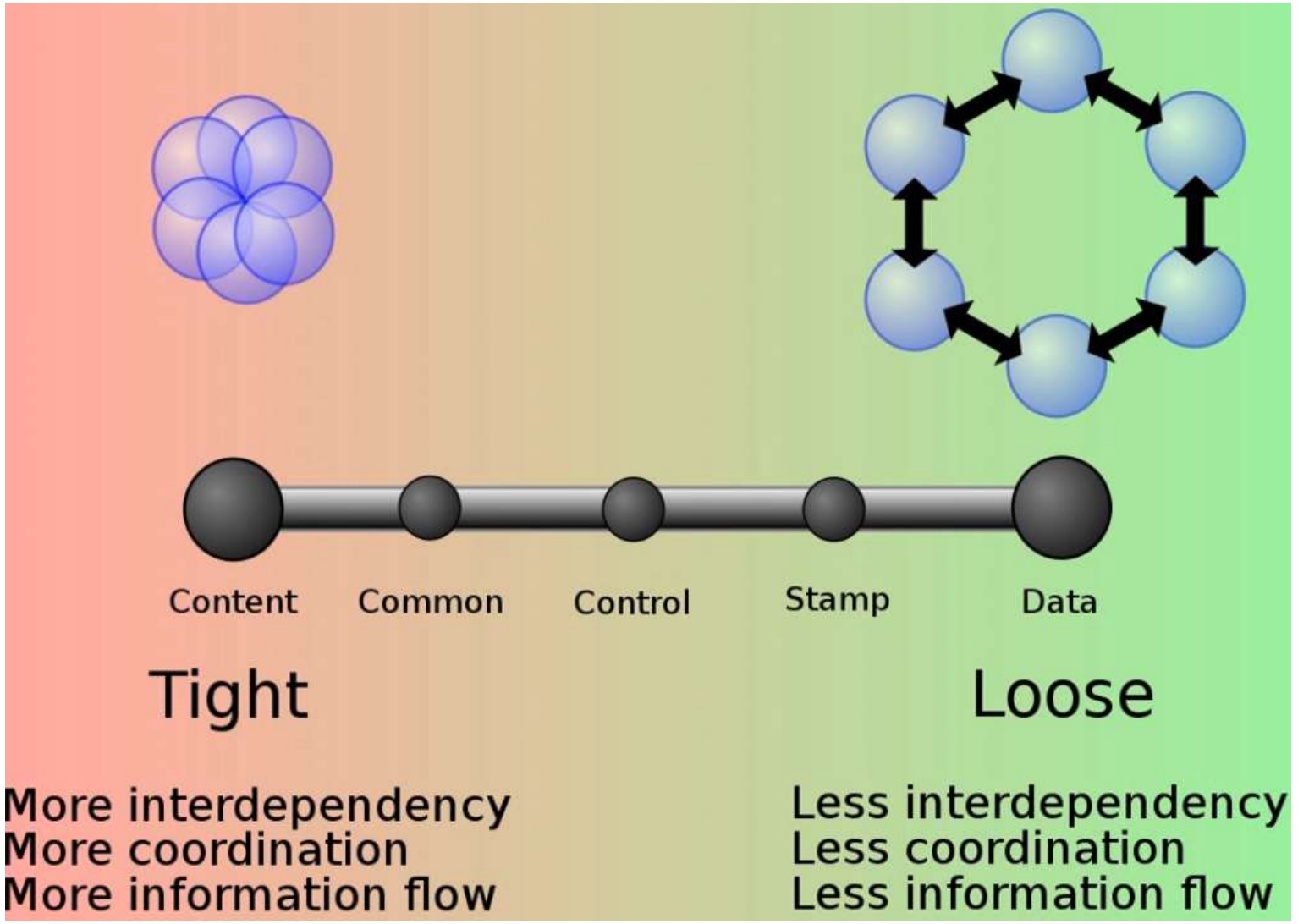
Coupling and Cohesion

- **Coupling -** concerns relationships between modules
- **Cohesion -** concerns relationships within a module
- **Goal:** To have loosely coupled modules with high internal cohesion

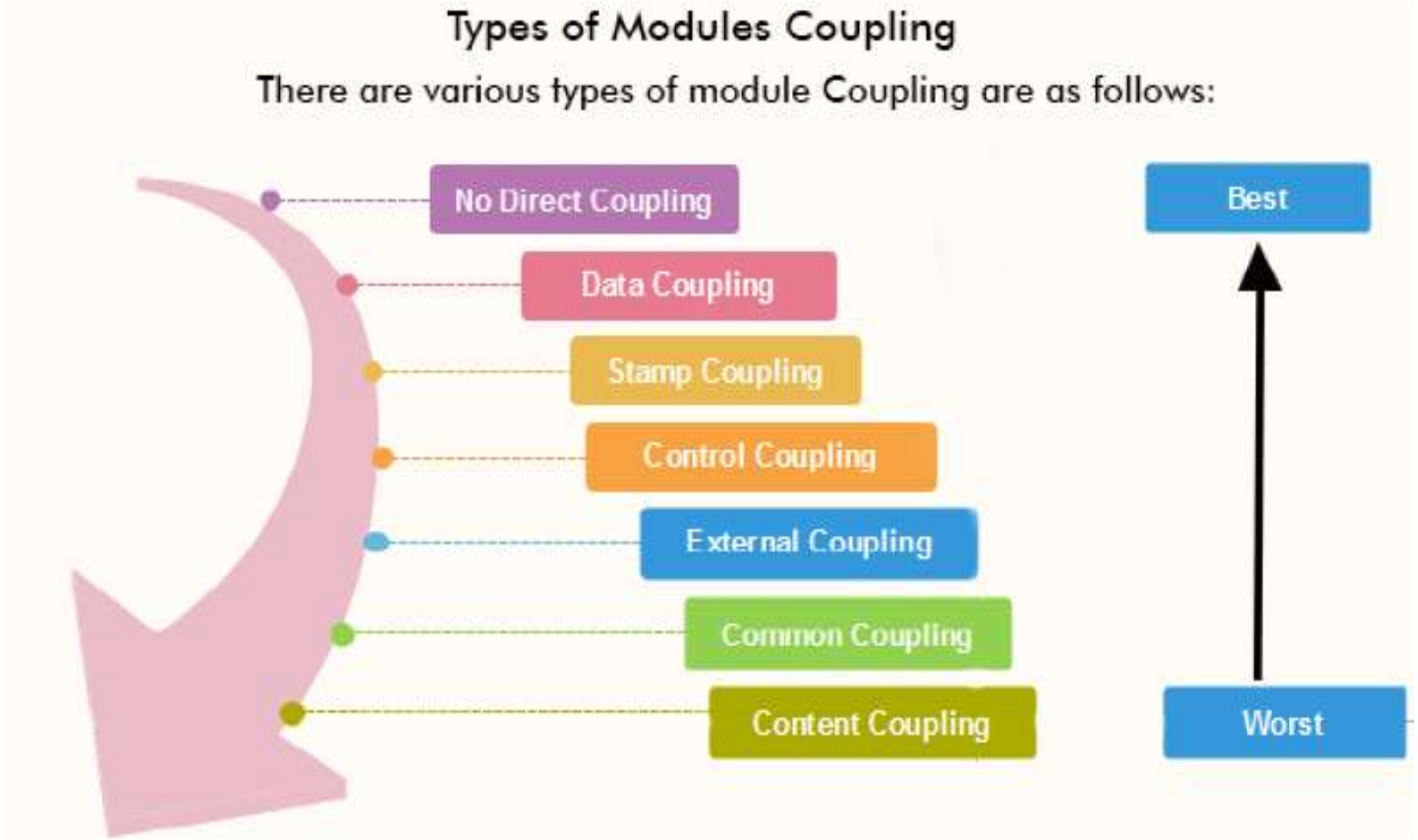
Coupling

- Coupling is defined as the extent to which a system, subsystem, method or module connects with (depends on) others. In other words, it measures interdependency. The lower the coupling, the better the program.
- Coupling increases as the number of calls between modules increase or the amount of shared data is large.

Coupling

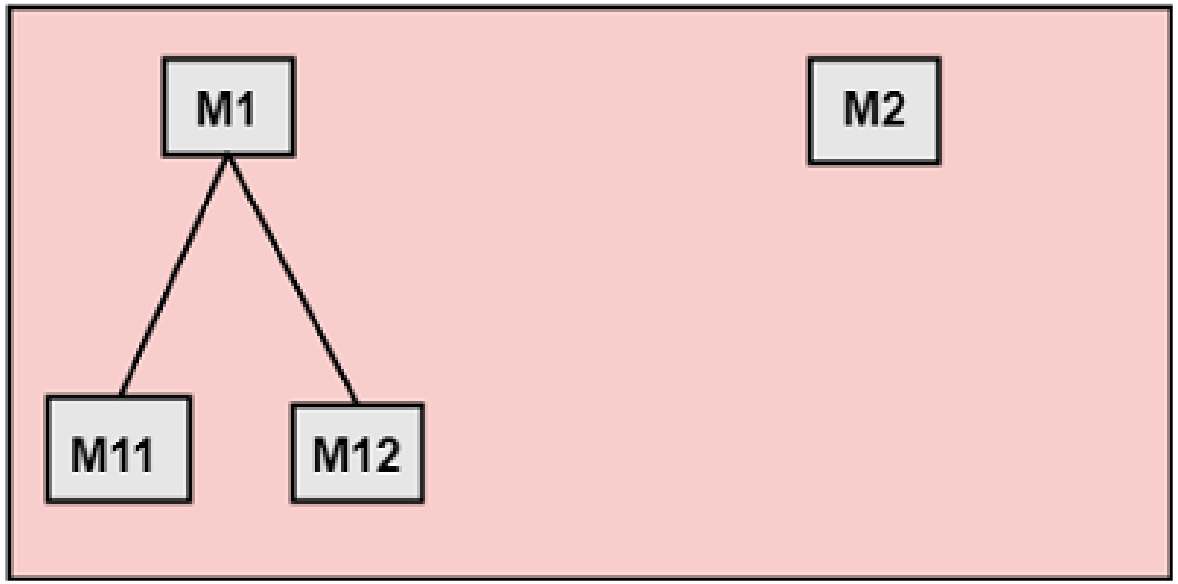

Content
Common
Control
Stamp
Data
Tight
Loose
More interdependency
More coordination
More information flow
Less interdependency
Less coordination
Less information flow

Coupling


Types of Modules Coupling
There are various types of module Coupling are as follows:
No Direct Coupling
Data Coupling
Stamp Coupling
Control Coupling
External Coupling
Common Coupling
Content Coupling
Best
Worst

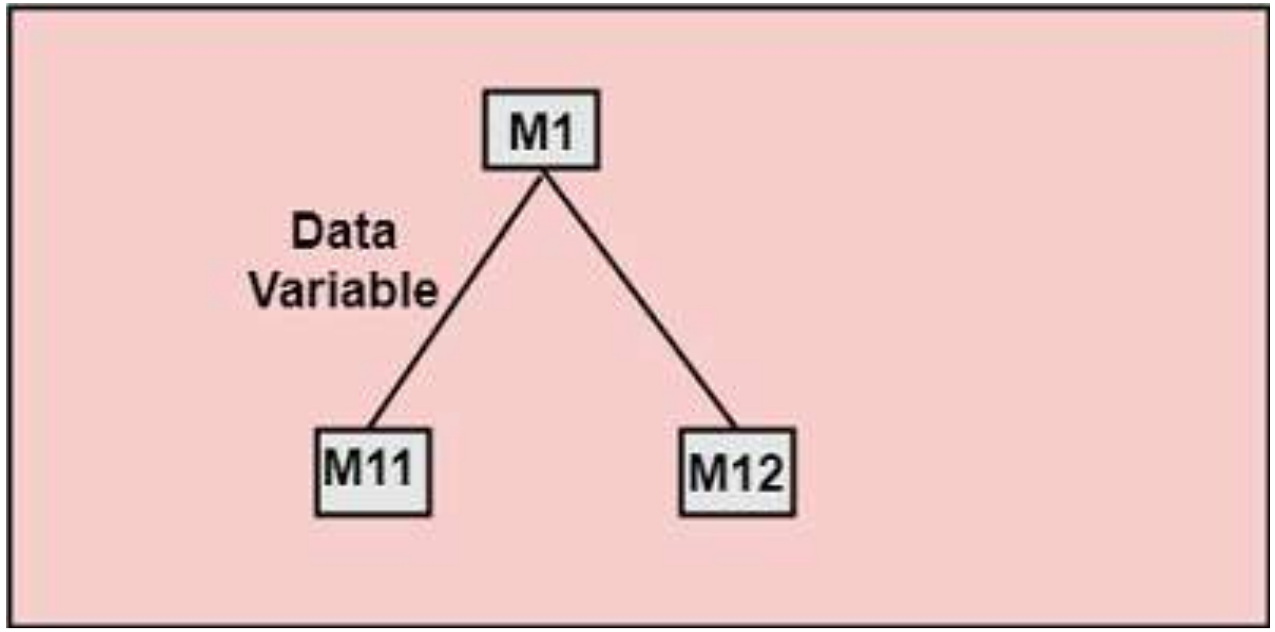
1. No Direct Coupling

- There is no direct coupling between M1 and M2.
- Modules are subordinates to different modules. Therefore, no direct coupling.



2. Data Coupling

- When data of one module is passed to another module, this is called data coupling.
- Example-customer billing system.



3. Stamp Coupling

- Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled.
- For example, passing structure variable in C or objects in C++ language to a module.

4. Control Coupling

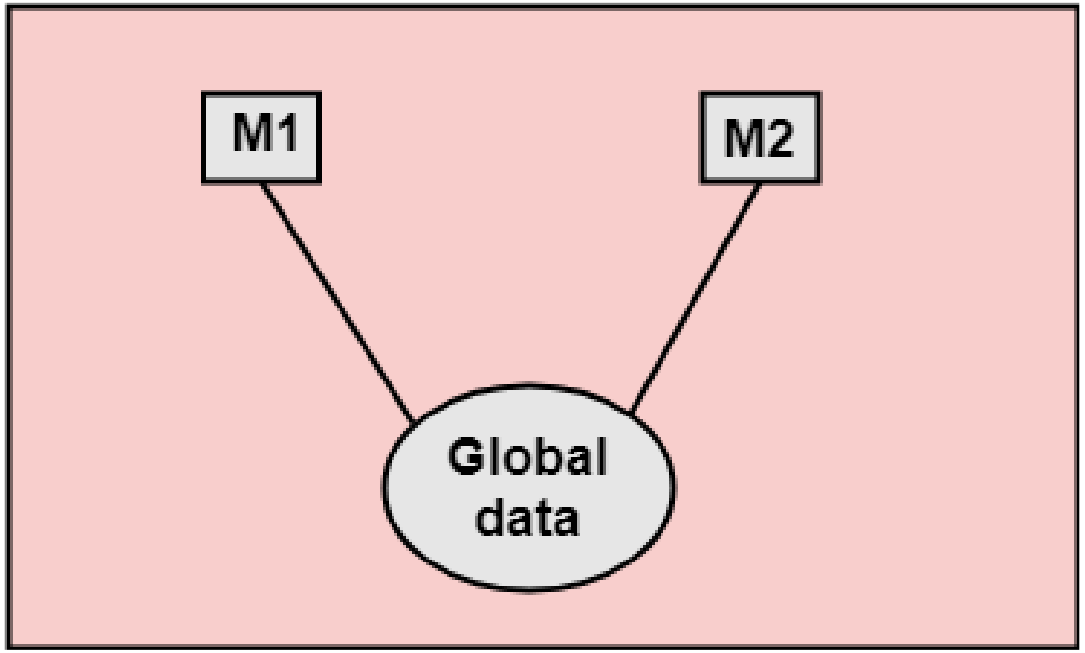
- Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. External Coupling

- External Coupling arises when two modules share an externally imposed data Format, communication protocols, or device interface. This is related to communication to external tools and devices.
- Ex- protocol, external file, device format, etc.

6. Common Coupling

- Two modules are common coupled if they share information through some global data items.



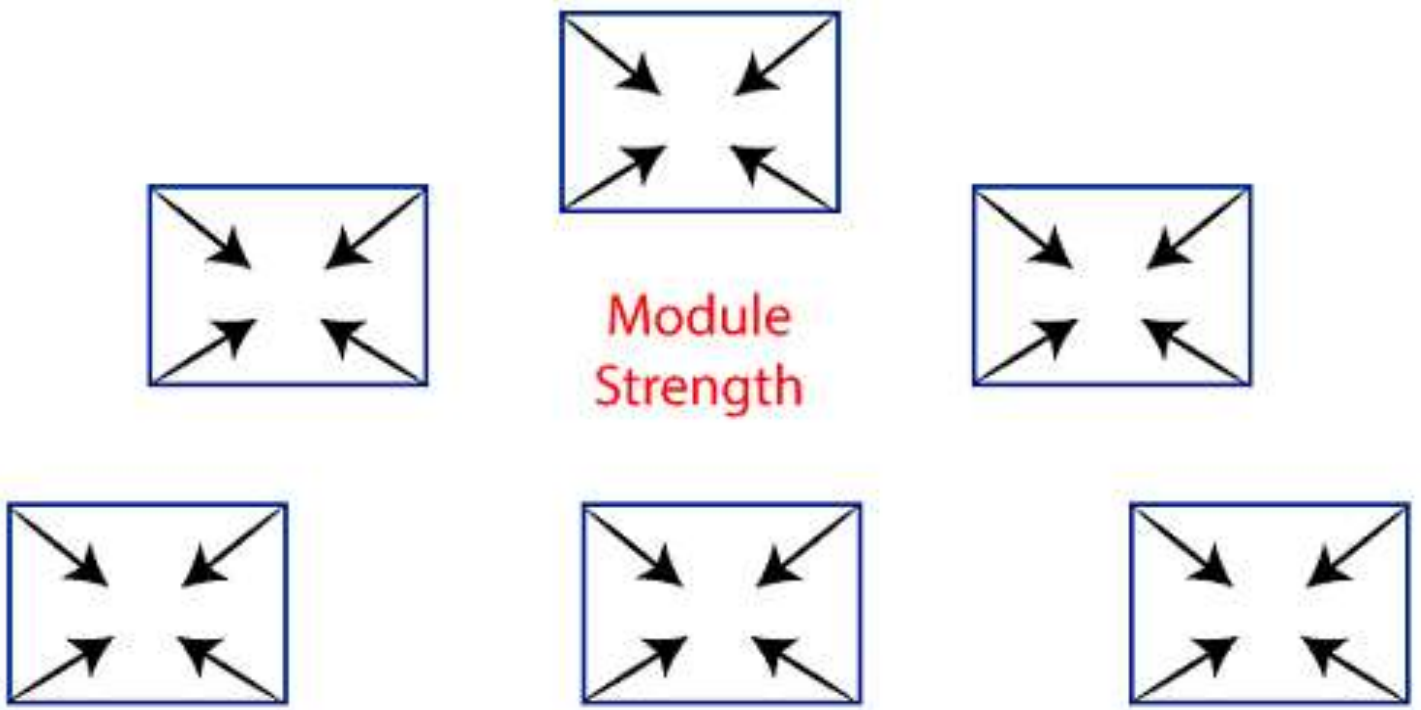
7. Content Coupling

- Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Module Cohesion

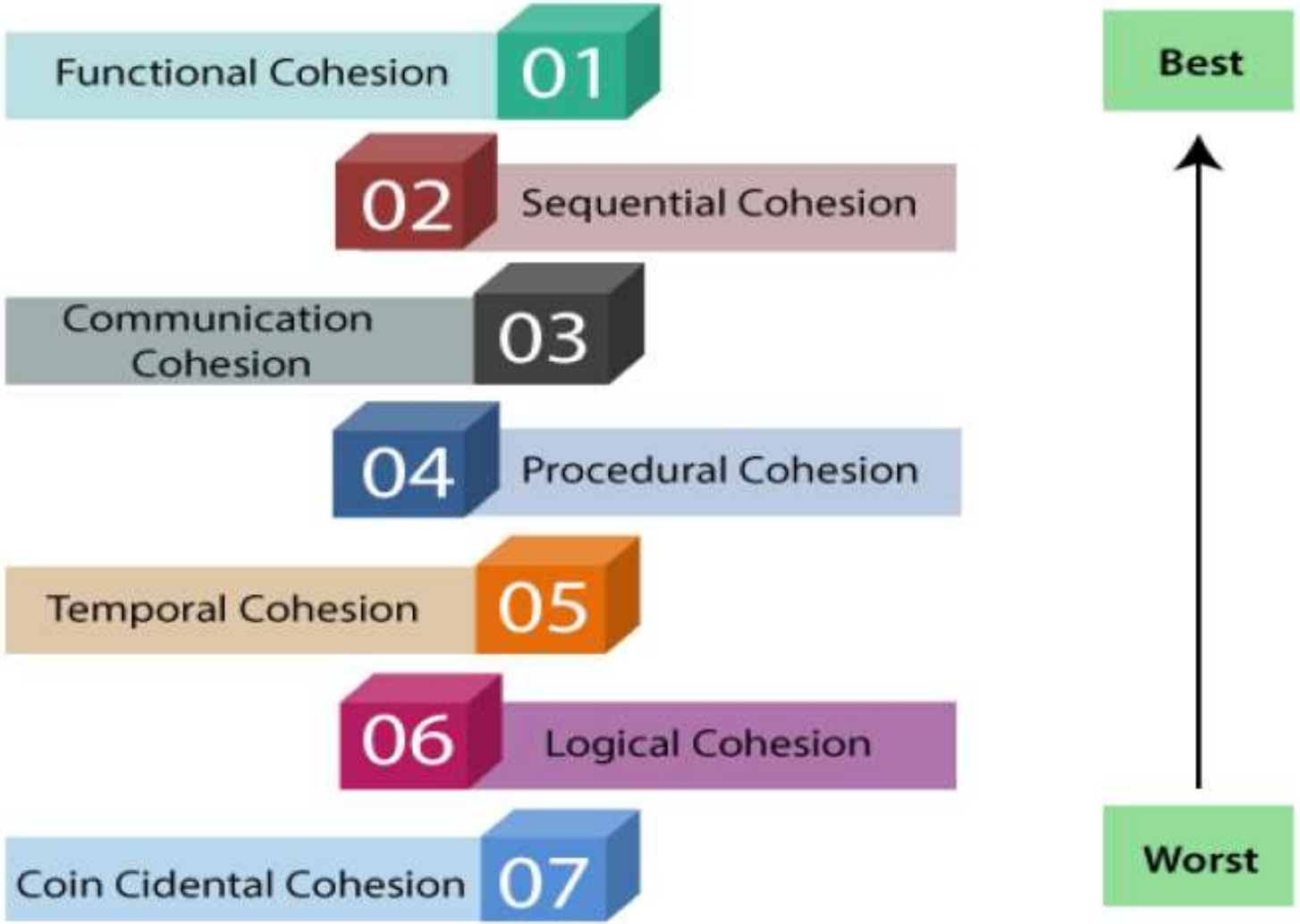
- In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.
- A good software design will have high cohesion.
- Cohesion is an ordinal type of measurement and is generally described as "**high cohesion" or "low cohesion**."

Cohesion



Cohesion= Strength of relations within Modules

Types of Modules Cohesion



1. **Functional Cohesion**

- Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

2. **Sequential Cohesion**

- A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

3. **Communicational Cohesion**

- A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.

4. **Procedural Cohesion**

- A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5. **Temporal Cohesion**

- When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

6. **Logical Cohesion**

- A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

7. **Coincidental Cohesion**

- A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Aspect	Cohesion	Coupling
Definition	Refers to the degree to which the elements of a module belong together.	Refers to the degree of interdependence between modules.
Focus	Focuses on the internal consistency of a module.	Focuses on the interconnections between different modules.
Goal	High cohesion is desirable to ensure modularity and clarity within a single module.	Low coupling is desirable to ensure that changes in one module have minimal impact on others.
Dependency	Does not involve dependency on other modules.	Involves dependency between modules.
Impact on Maintenance	Easier to maintain, test, and reuse highly cohesive modules.	Low coupling leads to easier maintenance and scalability.
Measurement	Measured within a module.	Measured between modules.
Ideal Level	High cohesion is desirable.	Low coupling is desirable.
Example	A class that focuses solely on performing file operations (e.g., reading and writing files).	Two modules communicate through a well-defined API rather than directly accessing each other's data.

Tight vs Loose Coupling

Tight Coupling

- Tight coupling occurs when modules are highly dependent on each other. Changes in one module often require changes in the other, leading to reduced flexibility and maintainability.

Characteristics:

1. **High Dependency**: Modules rely heavily on each other's internal implementation.
2. **Low Flexibility**: Modifying one module may cause a ripple effect in dependent modules.
3. **Hard to Test**: Individual modules are difficult to test in isolation.
4. **Low Reusability**: Tight coupling makes modules less reusable in different contexts.

Example of Tight Coupling:

- **Scenario**: A class directly creates an instance of another class, making it dependent on that class.

```
class Engine {
    public void start() {
        System.out.println("Engine started.");
    }
}

class Car {
    private Engine engine = new Engine(); // Tight
coupling: Direct dependency

    public void startCar() {
        engine.start(); // Car is tightly dependent on
Engine
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar();
    }
}
```

Tight vs Loose Coupling…

- **Loose Coupling**
  - Loose coupling occurs when modules are independent of each other, interacting only through well-defined interfaces. This reduces the impact of changes in one module on the other.

Characteristics:

1. **Low Dependency**: Modules interact via interfaces or abstractions.
2. **High Flexibility**: Changes in one module have minimal impact on others.
3. **Easy to Test**: Modules can be tested independently.
4. **High Reusability**: Modules are more reusable in different systems.

Example of Loose Coupling:

- **Scenario**: A class interacts with another class through an interface, reducing direct dependency.

```
interface Engine {
    void start();
}

class PetrolEngine implements Engine {
    public void start() {
        System.out.println("Petrol engine started.");
    }
}

class DieselEngine implements Engine {
    public void start() {
        System.out.println("Diesel engine started.");
    }
}
```

```
class Car {
    private Engine engine; // Loose coupling:
Dependency via abstraction (interface)

    public Car(Engine engine) {
        this.engine = engine;
    }

    public void startCar() {
        engine.start(); // Engine behavior depends on the
implementation provided
    }
}
```

Example of Loose Coupling…

```
public class Main {
    public static void main(String[] args) {
        Engine petrolEngine = new PetrolEngine();
        Car car1 = new Car(petrolEngine);
        car1.startCar();

        Engine dieselEngine = new DieselEngine();
        Car car2 = new Car(dieselEngine);
        car2.startCar();
    }
}
```

Fundamental Concepts

- A set of fundamental software design concepts has evolved over the history of software engineering. They span both traditional and object-oriented software development.

- M.A. Jackson [Jac75] once said: " the beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right."

- Fundamental software design concepts provide the necessary framework for "getting it right".

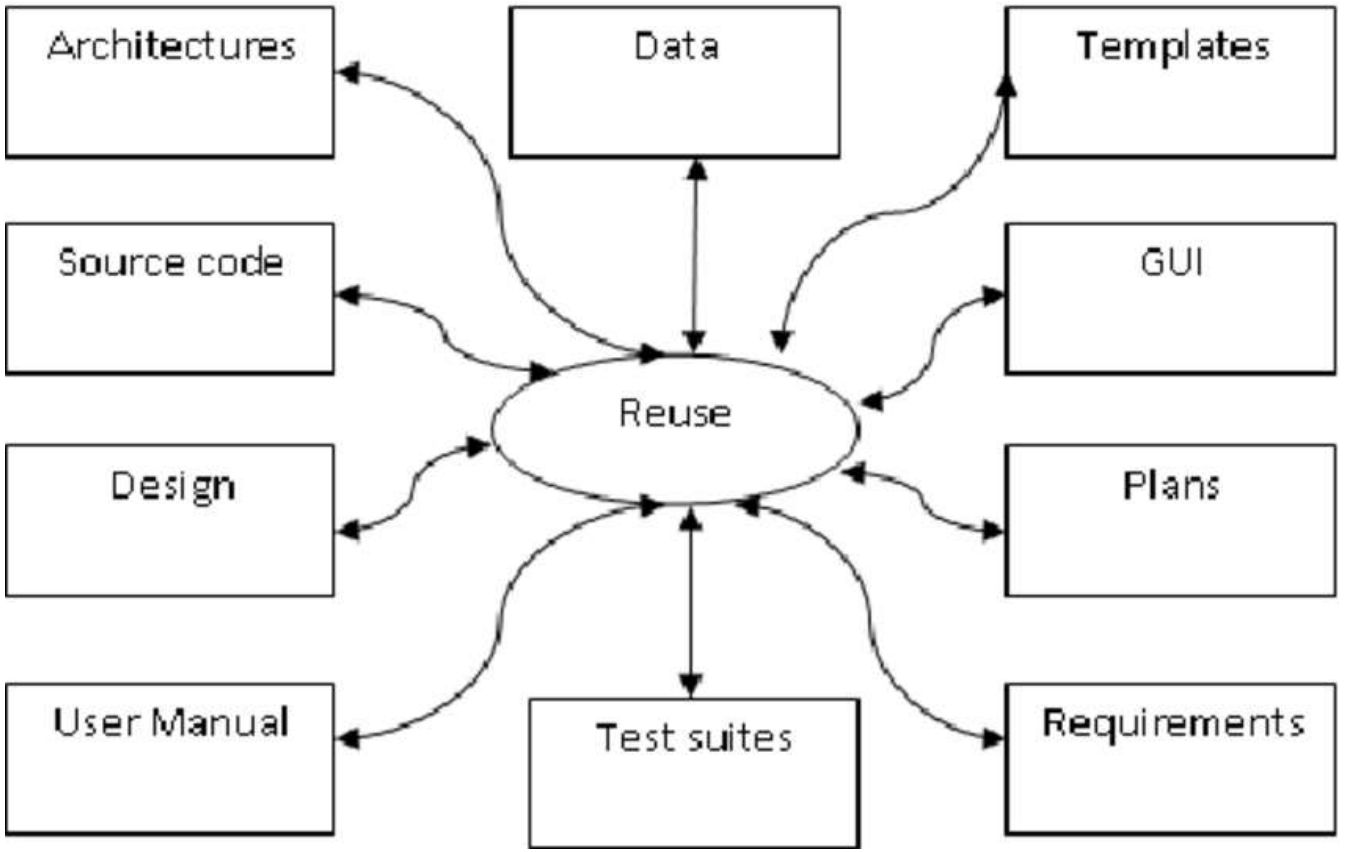
Fundamental Concepts

- Abstraction—data, procedure, control
- Architecture—the overall structure of the software
- Patterns—”conveys the essence” of a proven design solution
- Separation of Concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Functional independence—single-minded function and low coupling
- Refinement—elaboration of detail for all abstractions
- Aspects—a mechanism for understanding how global requirements affect design
- Refactoring—a reorganization technique that simplifies the design
- OO design concepts—
- Design Classes—provide design detail that will enable analysis classes to be implemented

Reusability

- Reusability is the likelihood a segment of components (source code) that can be used again to add new functionalities with slight or no modification.
- Reusable modules and classes reduce implementation time, increase the likelihood that prior testing and use has eliminated bugs and localizes code modifications when a change in implementation is required.
- **Benefits**:
  - Reduces development cost and time.
  - Improves quality and reliability.

Reusability…



TYPES OF REUSE

- **Opportunistic reuse -** While getting ready to begin a project, the team realizes that there are existing components that they can reuse.
- **Planned reuse -** A team strategically designs components so that they'll be reusable in future projects.

TYPES OF REUSE…

Opportunistic reuse can be categorized further:

- **Internal reuse -** A team reuses its own components. This may be a business decision, since the team may want to control a component critical to the project.
- **External reuse -** A team may choose to license a third-party component. Licensing a third-party component typically costs the team 1 to 20 percent of what it would cost to develop internally. The team must also consider the time it takes to find, learn and integrate THE COMPONENT.

REUSE APPROACHES

- **Design patterns:-** Generic abstractions that occur across applications are represented as design patterns that show abstract and concrete objects and interactions.
- **Component-based development:-** Systems are developed by integrating components (collections of objects) that conform to component-model standards.
- **Application frameworks:-**Collections of abstract and concrete classes that can be adapted and extended to create application systems.
- **Legacy system wrapping:-**Legacy systems that can be “wrapped” by defining a set of interfaces and providing access to these legacy systems through these interfaces.
- **Service-oriented systems:-**Systems are developed by linking shared services that may be externally provided.

REUSE APPROACHES…

- **Application product lines:-** An application type is generalized around a common architecture so that it can be adapted in different ways for different customers.
- **COTS integration**:- Systems are developed by integrating existing application systems.
- **Configurable vertical applications:-**A generic system is designed so that it can be configured to the needs of specific system customers**.**
- **Program libraries:-**Class and function libraries implementing commonly-used abstractions are available for reuse.
- **Program generators**:- A generator system embeds knowledge of a particular types of application and can generate systems or system fragments in that domain.
- **Aspect-oriented software development:-**Shared components are woven into an application at different places when the program is compiled.

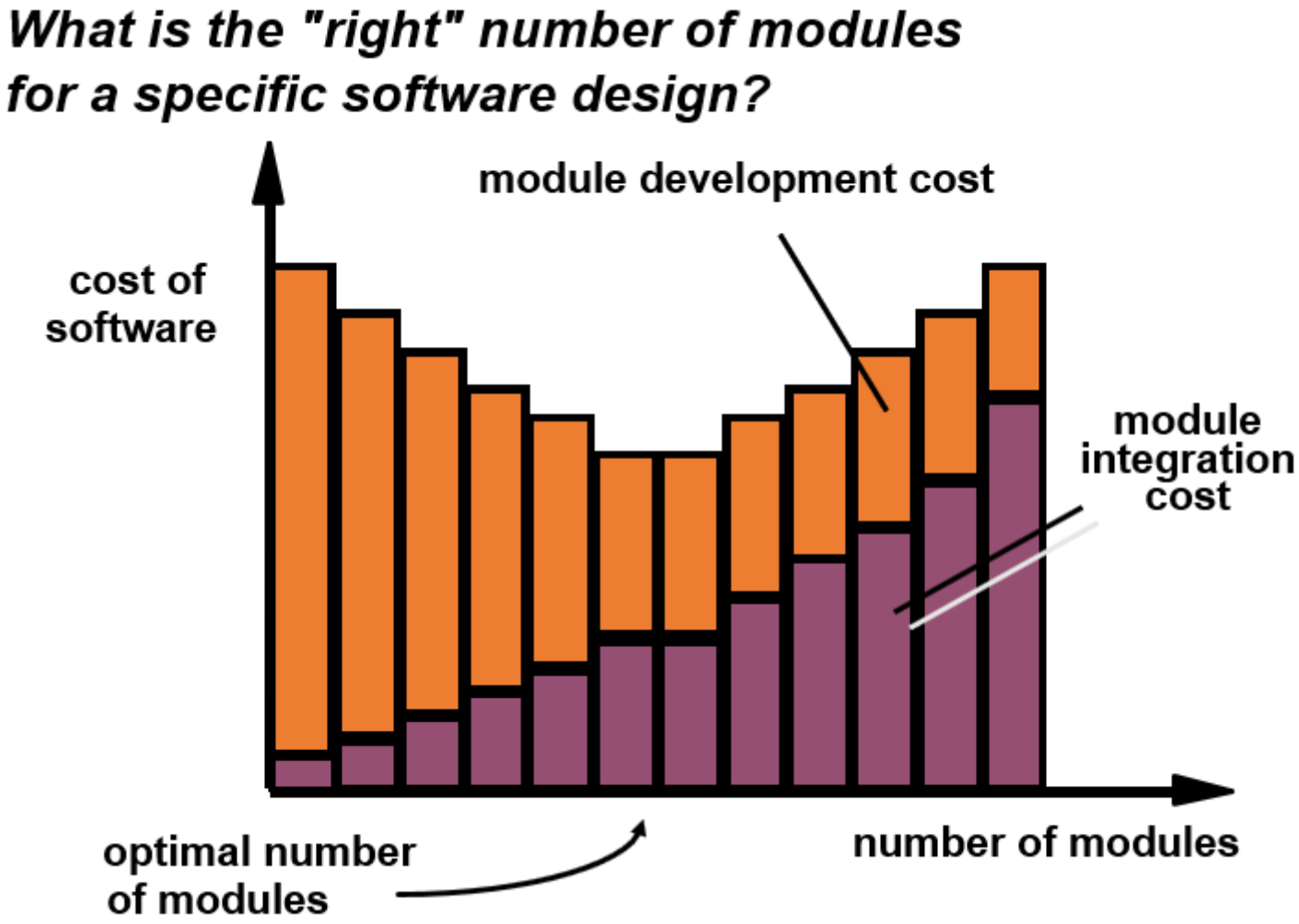
Separation of Concerns

- 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.

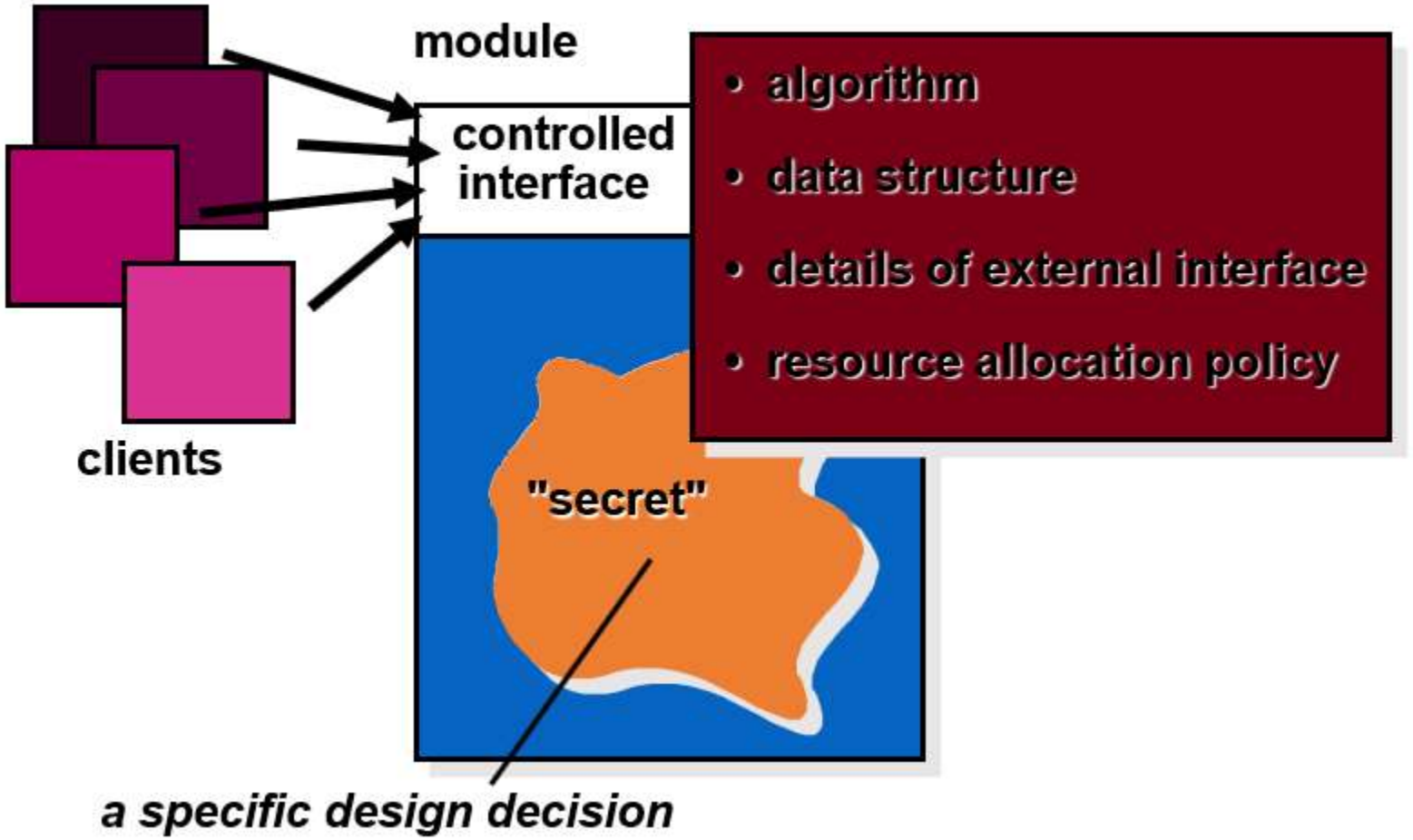
Modularity

- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall 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.

Modularity: Trade-offs



Information Hiding



Why Information Hiding?

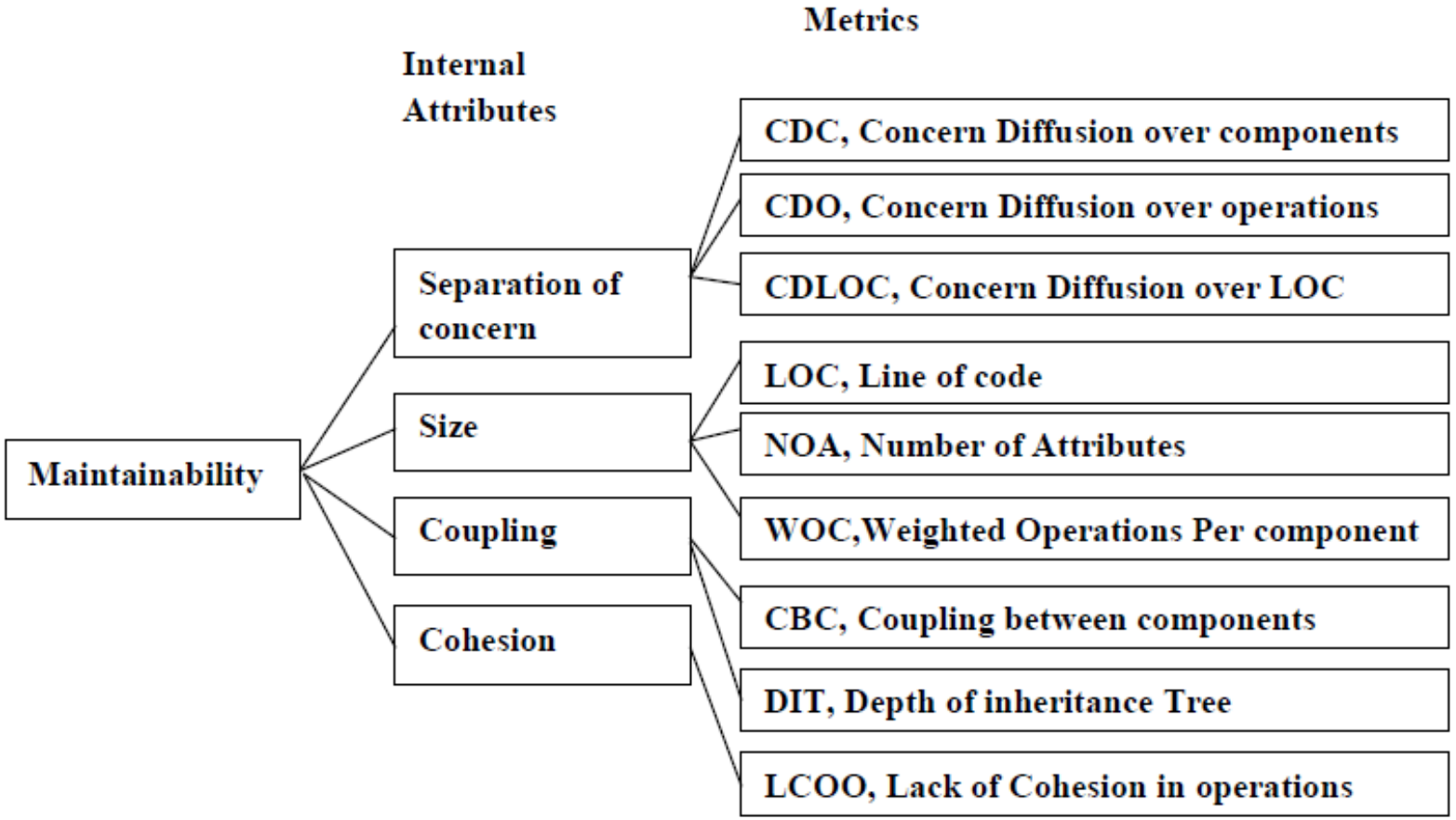
- reduces the likelihood of “side effects”
- limits the global impact of local design decisions
- emphasizes communication through controlled interfaces
- discourages the use of global data
- leads to encapsulation—an attribute of high quality design
- results in higher quality software

Maintainability

- Maintainability refers to the degree to which a software or component of a software can be easily modified in order to correct bugs, add quality attributes or adjust the operating environment and then improve the efficiency of the entire software. Generally, software maintenance phase demands that needed changes are made to the existing system (Michura, Capretz, & Wang, 2013).
- As a result of inevitable increases in the size and complexity of software products, software maintenance duties have also become increasingly herculean.
- Therefore, an urgent concern in the computing industry is the need to maintain and enhance software products as cost effective as possible and within a short time. To meet this objective, concepts and techniques which lead to designing more maintainable solution should be given serious consideration.
- In short, software maintenance should no longer be a design afterthought; that is, it should be easy for software maintainers to enhance the quality of the product without compulsorily tearing down and rebuilding the substantial parts of the code.

Maintainability…

- The following illustrates maintainability and the external versus internal attributes:



Maintainability…

Separation of Concern:

- This attribute refers to the division of a software system into distinct sections, each handling a specific responsibility.
- **Metrics**:
  - **CDC (Concern Diffusion over Components)**: Measures how concerns (functional or non-functional requirements) are distributed across components. Higher diffusion can complicate maintenance.
  - **CDO (Concern Diffusion over Operations)**: Tracks the spread of concerns within operations (methods or functions). A low CDO implies better encapsulation and maintainability.
  - **CDLOC (Concern Diffusion over Lines of Code)**: Evaluates the spread of concerns over the lines of code, where a higher value suggests potential maintenance challenges.

Maintainability…

Size:

- Larger software components are harder to understand, test, and maintain.
- **Metrics**:
  - **LOC (Lines of Code)**: Total lines in a component. Fewer lines often indicate simplicity but must be balanced with functionality.
  - **NOA (Number of Attributes)**: Counts the number of attributes (variables) in a class. Fewer attributes typically make a class easier to maintain.

Maintainability…

Coupling:

- Refers to the degree of dependency between software modules. Lower coupling improves modularity and maintainability.
- **Metrics**:
  - **WOC (Weighted Operations per Component)**: Assesses the complexity of operations in a component. Higher weights often mean higher coupling.
  - **CBC (Coupling Between Components)**: Measures direct dependencies between components. A low CBC is desirable for easier maintenance.
  - **DIT (Depth of Inheritance Tree)**: Represents the inheritance levels for a class. Higher depth can make the system harder to maintain.

Maintainability…

Cohesion:

- Indicates how closely related the functionalities within a module are. Higher cohesion makes components easier to understand and maintain.
- **Metrics**:
  - **LCOO (Lack of Cohesion in Operations)**: Measures how unrelated operations are within a class. Higher LCOO indicates low cohesion, which negatively impacts maintainability.