California State University, Sacramento
Computer Science Department

# CSC- 131

Fall 2022

Lecture # 6

# Design Concepts and Principles

SACRAMENTO STATE
Redefine the Possible

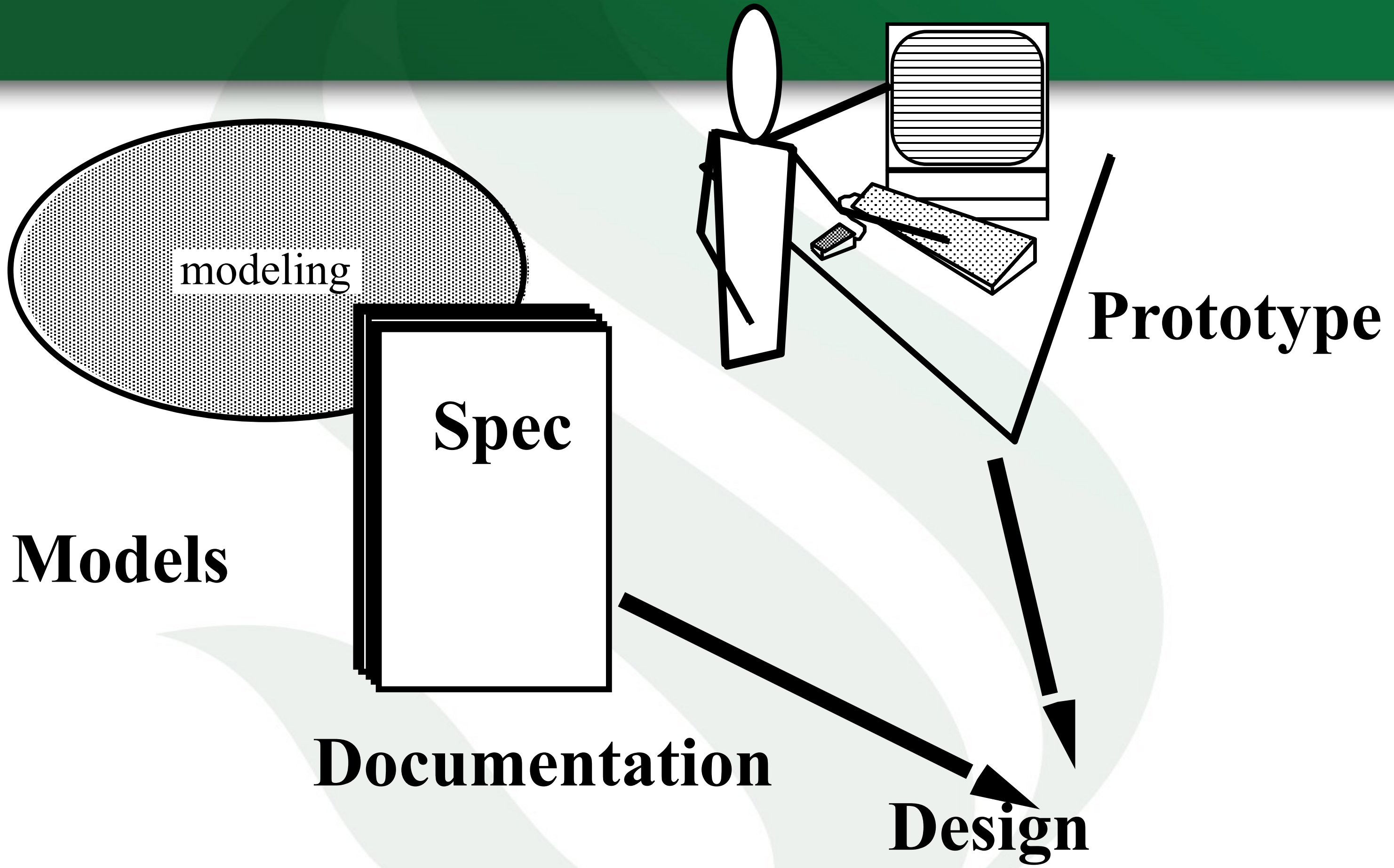
# Software Design

Goal:

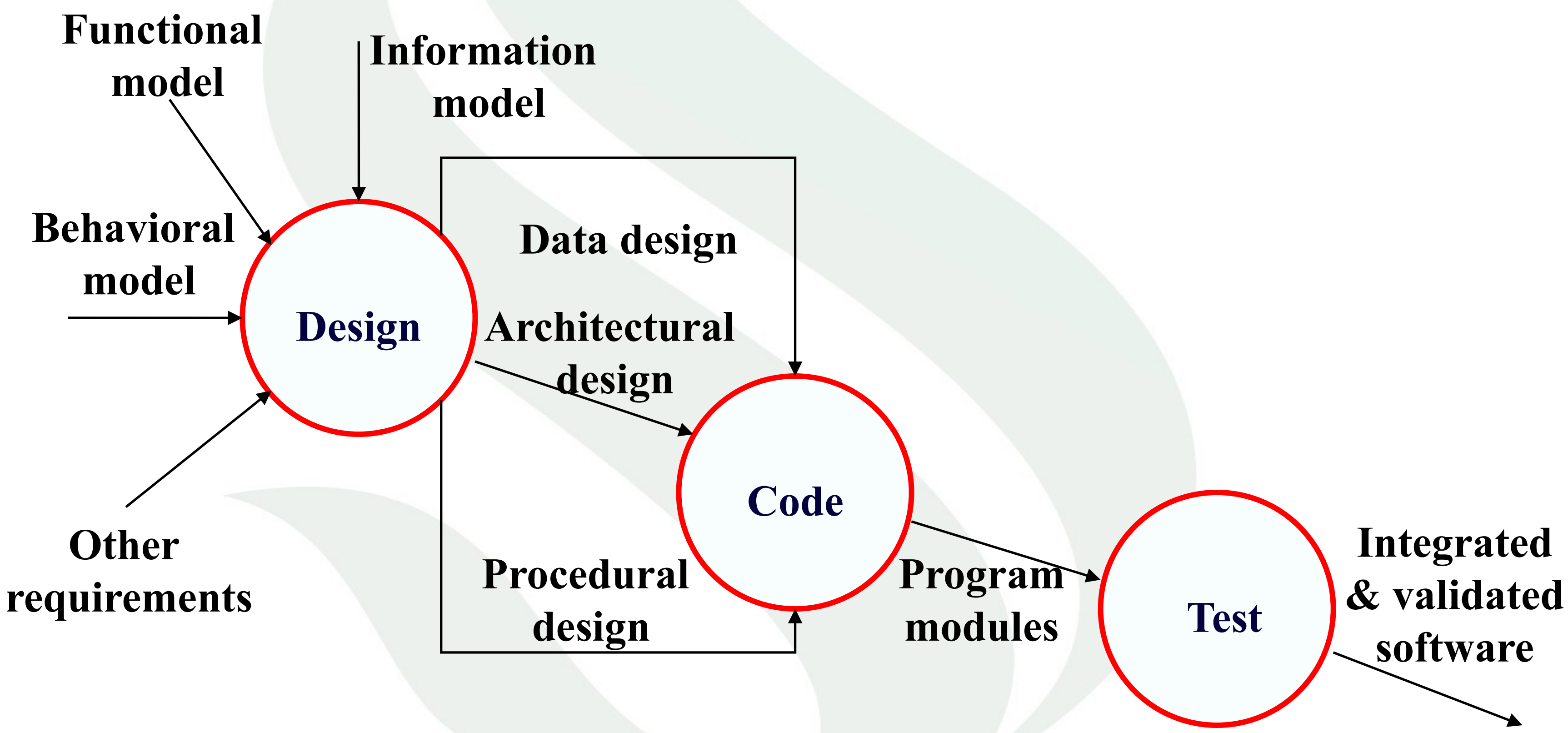
- To produce a model or representation that will later be built

- Software design is the first of three technical activities (Design, Implementation, and Test)

# Where Do We Begin?



# Software Design Model



# Software Design

## Design

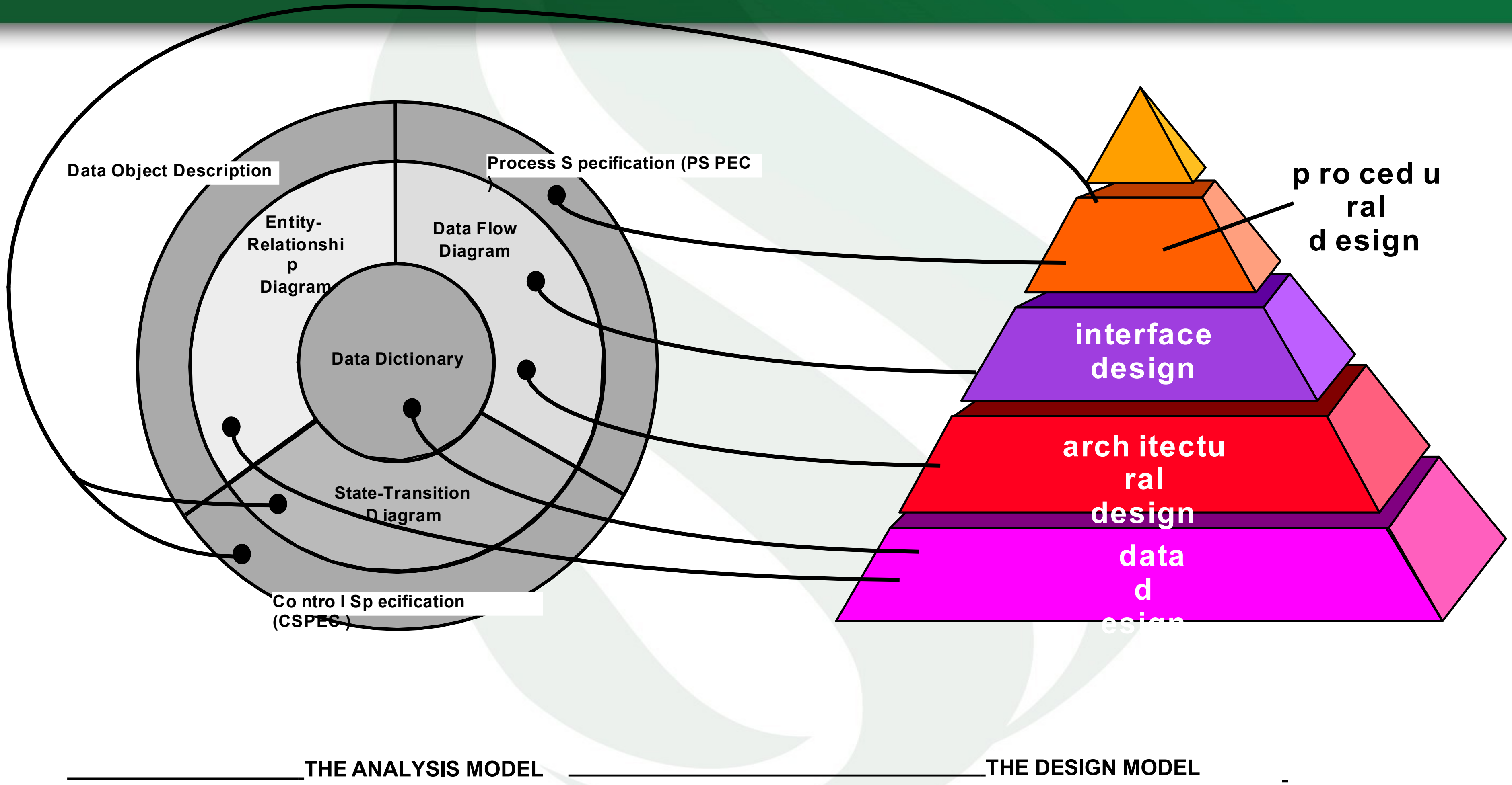
- Top-down approach: breaking a system into a set of smaller subsystems
- Object-oriented approach: identification of a set of objects and specification of their interactions
- UML diagrams are a design tool to illustrate the interactions between
  - Classes
  - Classes and external entities

# Design Models

- Analysis models are used to build design models.

- Four design models required to build the product:

  - Data design
  - Architectural design
  - Interface design
  - Component / Procedural design


Data Object Description
Process S pecification (PS PEC )
Entity-Relationshi p Diagram
Data Flow Diagram
Data Dictionary
State-Transition D iagram
Co ntro l Sp ecification (CSPEC )
p ro ced u ral d esign
interface design
arch itectu ral design
data d esign
THE ANALYSIS MODEL
THE DESIGN MODEL

# Data Design

- Transform the model created during analysis into the data structures that will be required to implement the software.

- ERD and Data Dictionary are used to build this model.

- Created by transforming the analysis information model (data dictionary and ERD) into data structures required to implement the software.

- Part of the data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

SACRAMENTO STATE
Redefine the Possible

# Architectural Design

- Objective is to develop a modular program structure and represent the control relationships between modules. DFDs are used for this design.

- Defines the relationships among the major structural elements of the software, the design patterns that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which the architectural patterns can be applied.

- It is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD).

# Interface Design

- Describes how the software communicates within itself, with other systems and with users.
- DFDs may be used to develop the interface. Describes how the software elements communicate with each other, with other systems, and with human users. Use Case model can be used as well.
- The data flow and control flow diagrams provide much of the necessary information required.

# Component / Procedural Design

- After data & program structure have been established, it become necessary to specify procedural detail without ambiguity

- Graphical design notation
  - Flow-charts (draw sequence, if-then, selection, repetition)
- Program Design Language(PDL) = pseudocode
- 
- Created by transforming the structural elements defined by the software architecture into procedural descriptions of software components using information obtained from the process specification (PSPEC), control specification (CSPEC), and state transition diagram (STD).

# Software Design Fundamentals

Good design is not accomplished by chance

*Fundamental concepts provide the framework for "**getting it right**"*

# Design Fundamentals

| | | |
|---|---|---|
| Abstraction | Refinement | Modularity |
| Control Hierarchy | Information Hiding | Refactoring |
| Patterns | Functional Independence | Architecture |

# Design Fundamentals

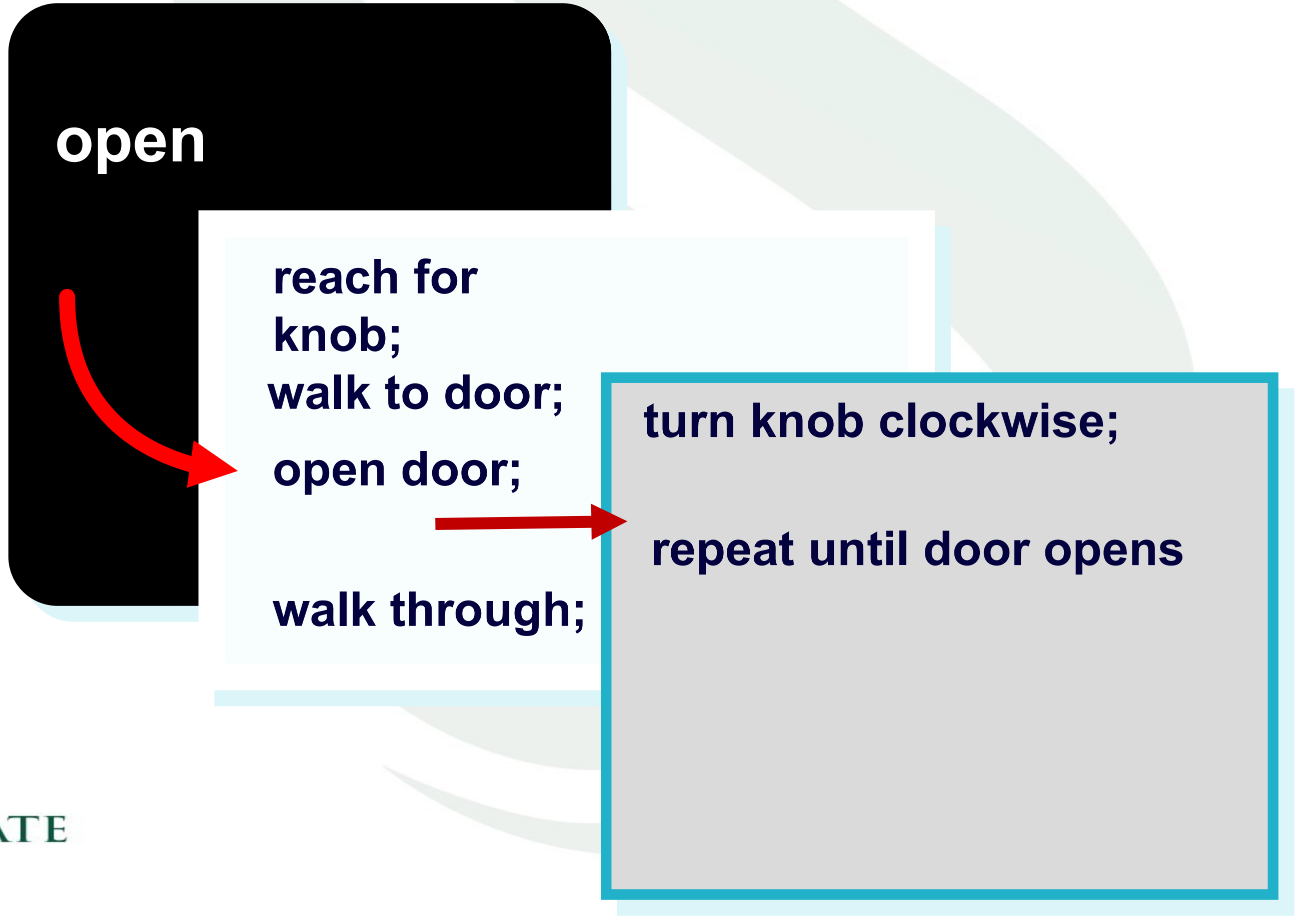
Abstraction

- Levels of detail/language used to describe a problem
- There are two different types of Abstraction namely:
  - Data Abstraction
  - Procedural Abstraction

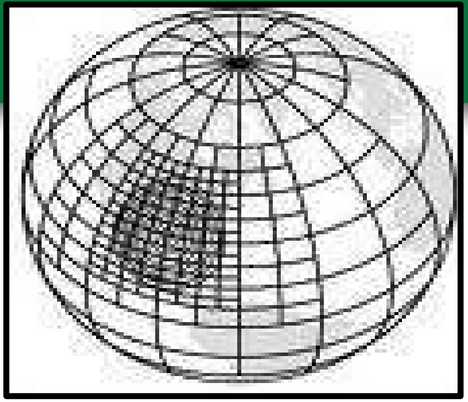
# Design Fundamentals (cont.)

- Refinement
  - Top-down strategy

## Stepwise Refinement



# Refinement



Refinement is a process of **elaboration**

- It is a top-down design strategy
- A program is developed by successfully refining levels of procedural details.

# Modular Design

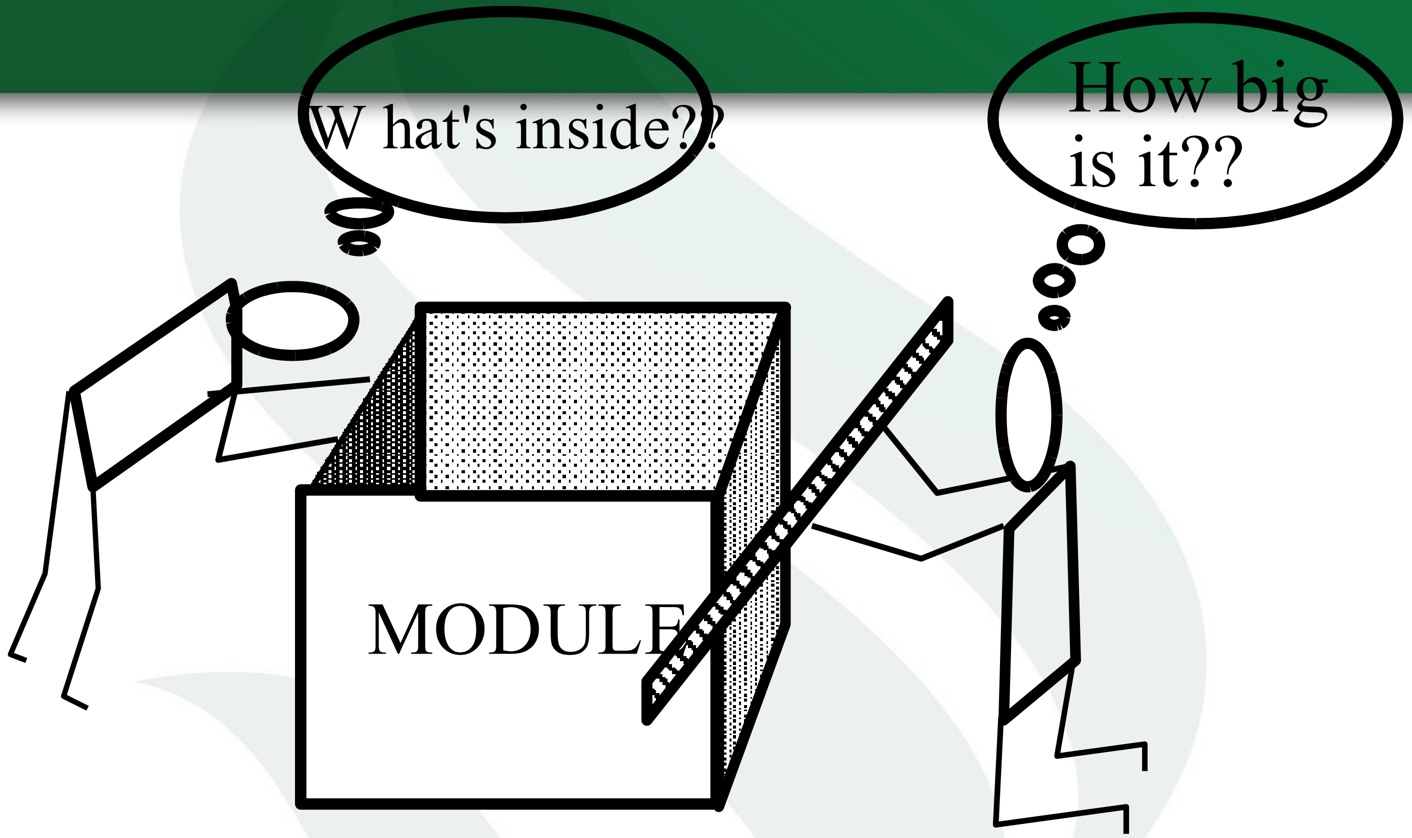
*Software is divided into separately named components **(Modules)***

Benefits

- Reduces **complexity**
- Facilitates **change**
- Easier **implementation**

# Sizing Modules

## Two Views



# Control Hierarchy

Control Hierarchy also called **Program Structure**

- Organization of modules that implies a **hierarchy of control**
- It does not represent **procedural** aspects of software.

# Structural Partitioning

## Horizontal Partitioning

- Defines separate branches for each major program function.
- Reasons & Benefits
  - to reduce module size
  - to avoid duplication of a function in more than one module
  - to provide more reusable modules
  - to simplify implementation

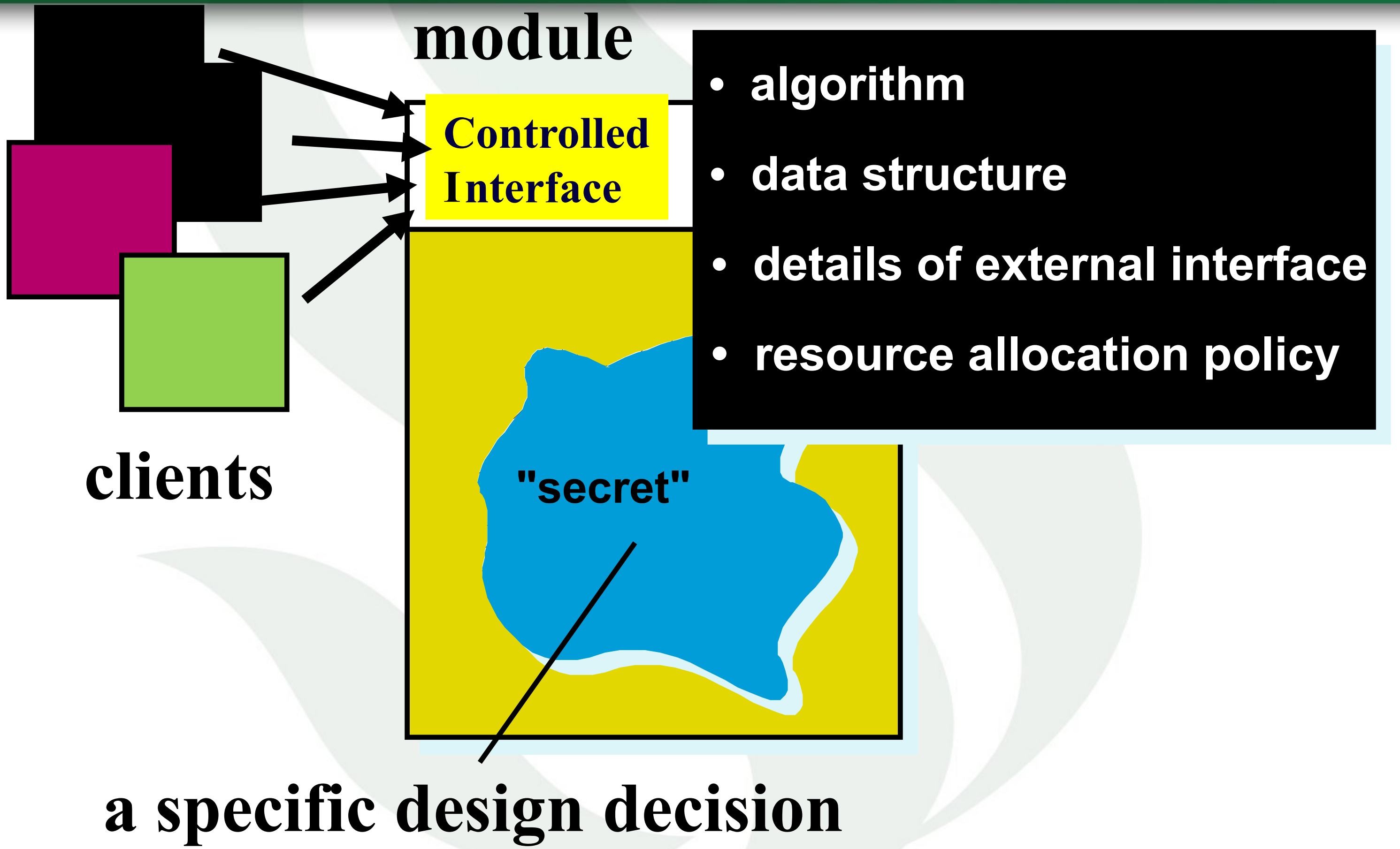
# Vertical Partitioning

- Vertical Partitioning (factoring). Implies that control (decision making) should be distributed **top-down** in the program.

- Top level should perform control function and do-little **processing work**. Lower-level module performs all types of input /output, and **computation tasks**

- Change in the **low-level** modules are less likely to cause **side effects**.

# 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**

- The greatest benefit is when modifications are required (during testing & maintenance); less **propagation of errors**

# Information Hiding

- Principle of information hiding says that a **good split** of modules is when **modules communicate** with one another with only the information necessary to **achieve the s/w function**.

- So, information hiding enforces access constraints to both **procedural detail** with a module, and **local data structure** used by that module.

- Data hiding is **a CRITERION for modular design**. How to know what modules to create.

SACRAMENTO STATE
Redefine the Possible

# Information Hiding (Benefits)

- 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 and results in **higher quality software**.

# Functional Independence

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

Benefits

- Easier to develop
- Easier to maintain & test

Measures of Independence

- Coupling
- Cohesion

# Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which modules in the system is "connected" to one other

# Coupling

- Coupling indicates the degree of interdependence between two modules

- Aim for low coupling by
  - **Eliminating** unnecessary **relationships**
  - **Reducing** the number of **necessary relationships**
  - **Easing** the "tightness" of **necessary relationships**

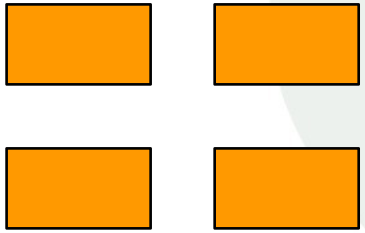
# Principles of Coupling

- Narrow is better than Broad
- Direct is better than Indirect
- Local is better than Remote
- Obvious is better than Obscure
- Flexible is better than Rigid

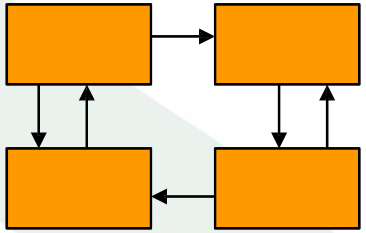
SACRAMENTO STATE
Redefine the Possible

# Coupling cont'd…

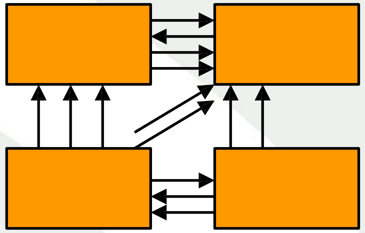
Degree of dependence among components.



No dependencies



Loosely coupled-some dependencies

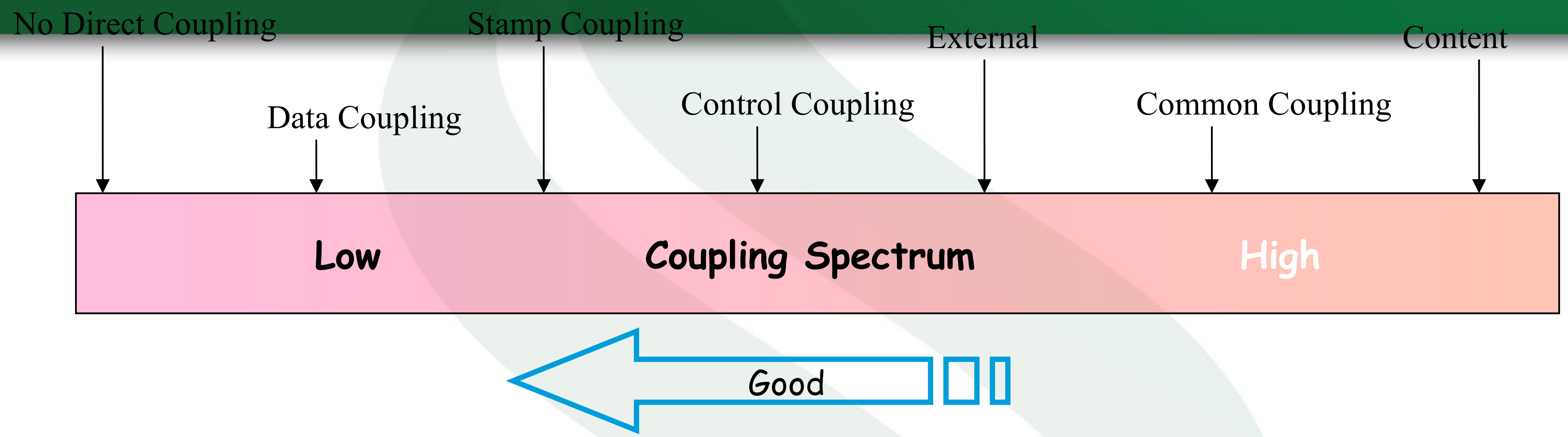


Highly couples-many dependencies

# Ways Components can be dependent

- References made from one to another
- Component A invokes B
- A depends on B for completion of its function or process
- Amount of data passed from one to another
- Component A passes to B: a parameter, contents of an array, data
- Amount of control one has over the other
- Component passes a control flag to B
- Value of flag tells B the state of some resource or subsystem, process to invoke, or whether to invoke a process
- Degree of complexity in the interface between components
- Components C and D exchange values before D can complete execution

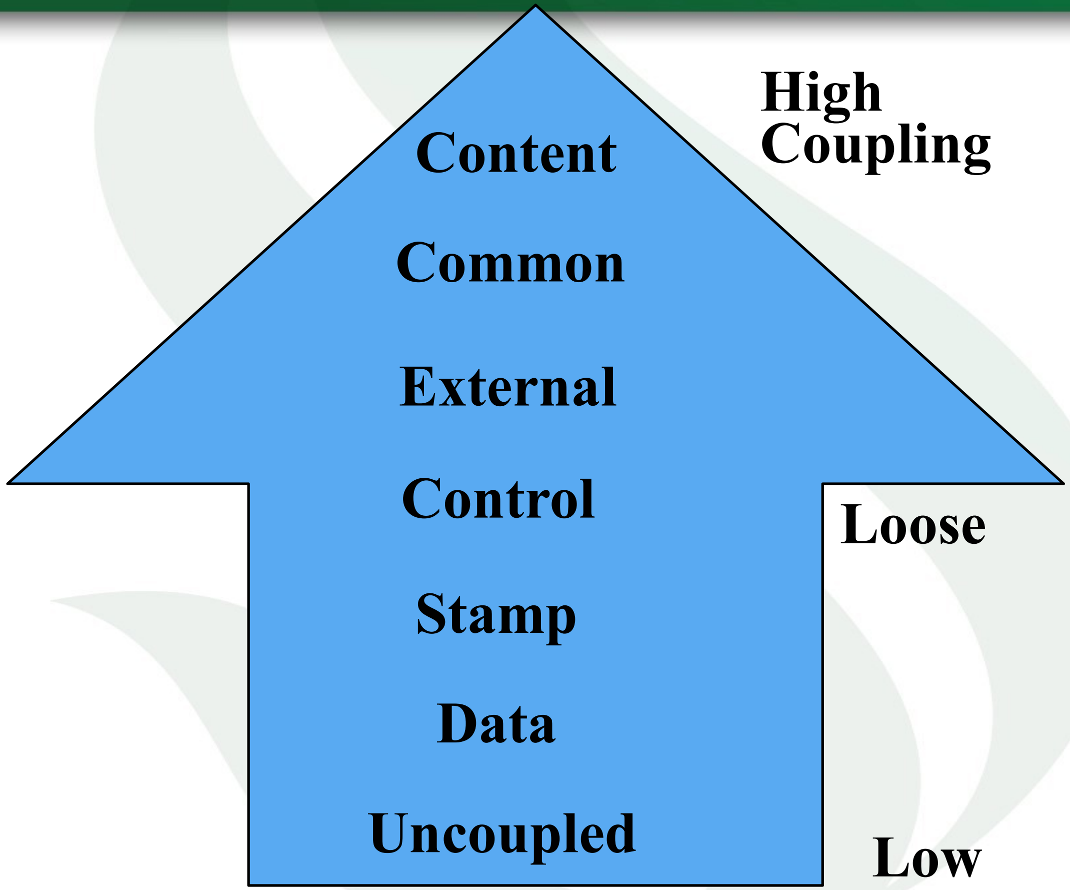
# Types of Coupling




A measure of the interdependence among software modules

Data: Simple argument passing
Stamp: Data structure passing
Control: One module passes the element of control (flag) to another.
External: Modules are tied to environment external to software (device)
Common: Modules have access to the same global data.
Content: One module directly references the content of another

# Range of Coupling




# Types of Coupling

- **Content Coupling : (worst)** When a module uses/alters data in another module
- **Common Coupling:** 2 modules communicating via global data
- **External Coupling:** Modules are tied to an environment external to the software
- **Control Coupling:** 2 modules communicating with a control flag

# Types of Coupling

- ✓**Stamp Coupling:** Communicating via a data structure passed as a parameter. The data structure holds more information than the recipient needs.

- ✓**Data Coupling : (best)** communicating via parameter passing.
- ✓The parameters passed are only those that the recipient needs.

- ✓**No data Coupling**: Independent modules

# Advantages of Low Coupling

- The **fewer connections** between modules, the less chance of a defect in one **causing a defect in another**
- The risk of having to change other modules as a result of changing one module is reduced
- The need to know about the **internals of other modules** is reduced when **maintaining the details** of other modules.
- Some coupling is needed…!

# Cohesion

- Is the measure of the strength of **functional relatedness** of elements within a module

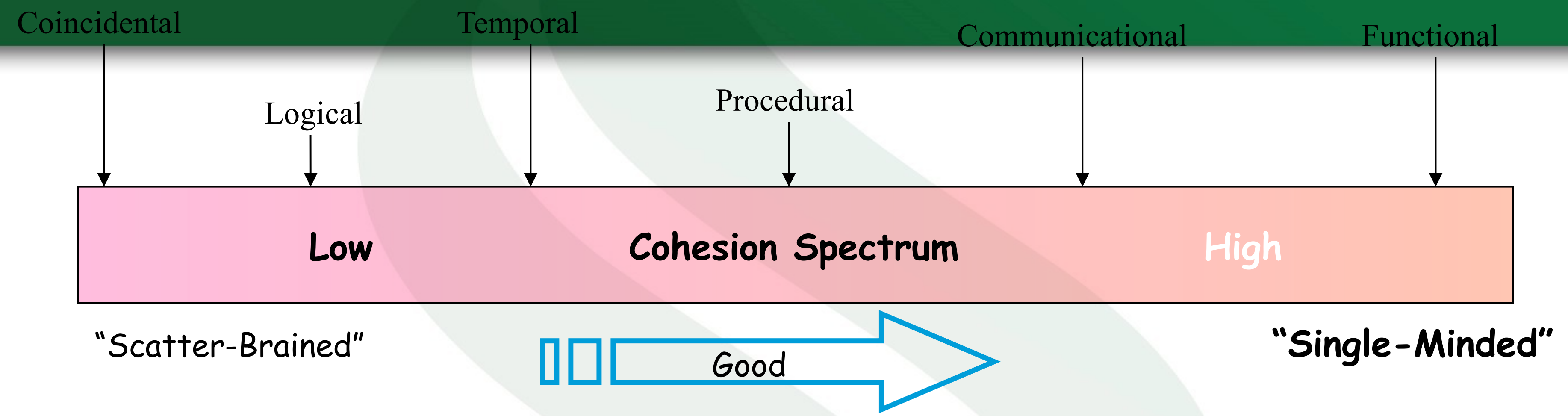
- "Element" means
  - an instruction, a group of instructions
  - **a data definition**
  - **a call to another module**

- We aim for **strong cohesion**: modules whose elements are functionally related

# Cohesion

- Measure of how well we have **partitioned the system.**
- Internal glue with which **component is constructed**.
- All elements of component are **directed toward essential** for performing the **same task**.

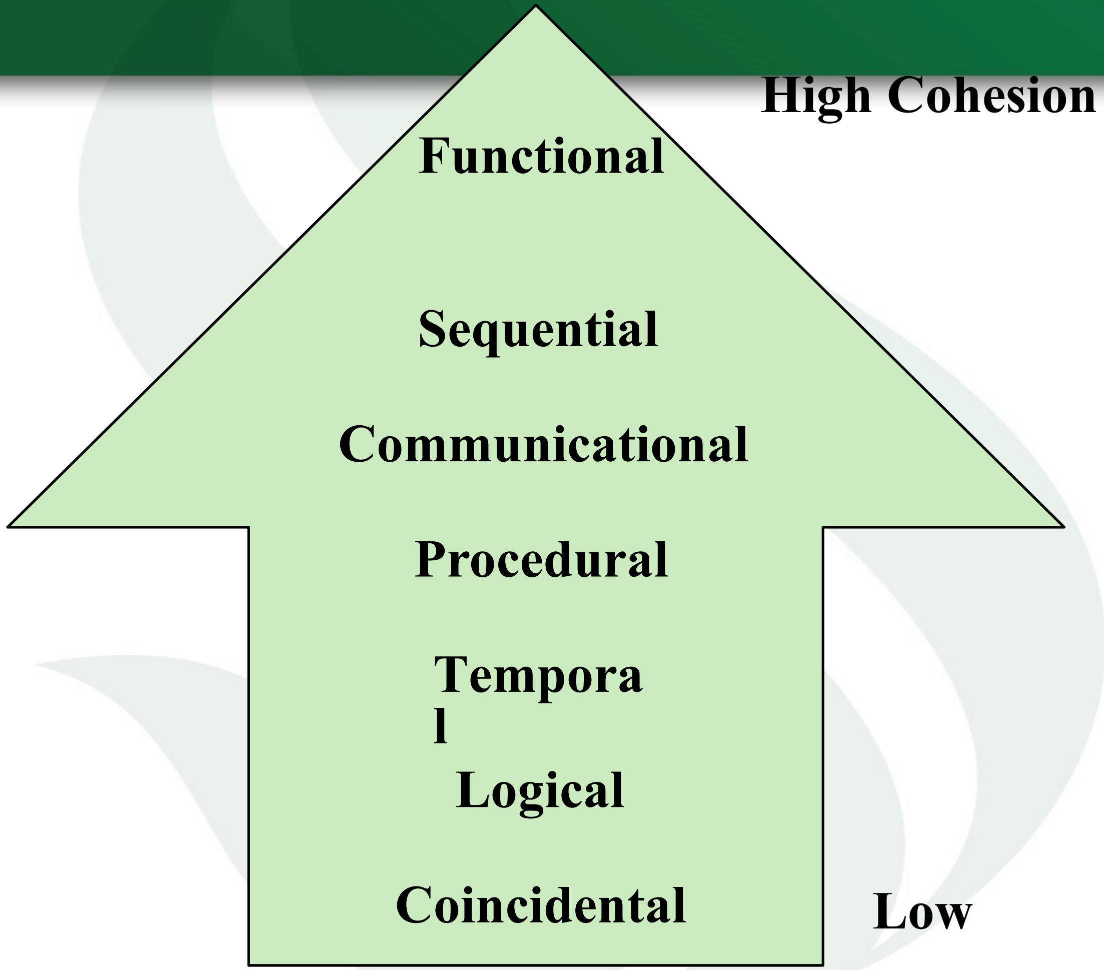
# Types of Cohesion




A measure of the relative functional strength of a software module

Coincidental: multiple, completely unrelated actions or components
Logical: series of related actions or components (e.g. library of IO functions)
Temporal: series of actions related in time (e.g. initialisation modules)
Procedural: series of actions sharing sequences of steps.
Communicational: procedural cohesion but on the same data.
Functional: one action or function

# Range of Cohesion



# Examples of Cohesion-1

| Function A | |
|---|---|
| Function B | Function C |
| Function D | Function E |

logic

Coincidental
Parts unrelated

| Function A |
|---|
| Function A’ |
| Function A’’ |

Logical
Similar functions

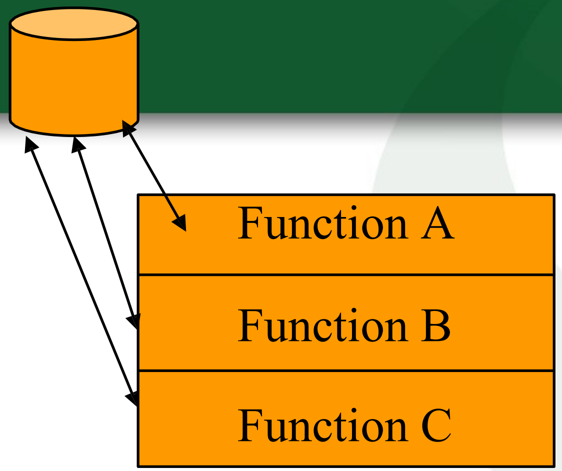
| Time $t_0$ |
|---|
| Time $t_0 + X$ |
| Time $t_0 + 2X$ |

Temporal
Related by time

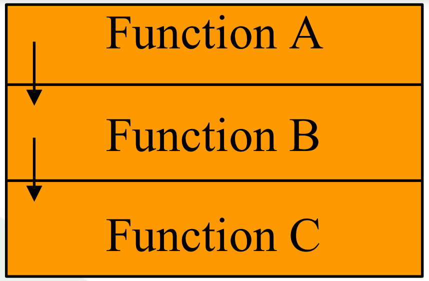
| Function A |
|---|
| Function B |
| Function C |

Procedural
Related by order of functions

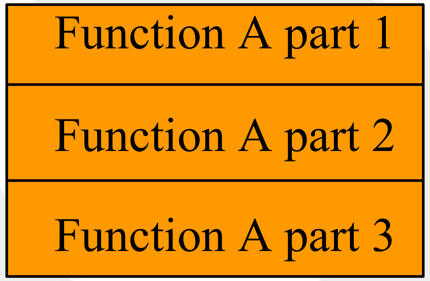
# Examples of Cohesion-2




Communicational
Access same data




Sequential
Output of one is input to another




Functional
Sequential with complete, related functions

# Differences between Cohesion and Coupling

## COHESION

- The measure of **strength** of the **association** of elements within a module.
- It is the **degree** to which the **responsibility** of a single component form a **meaningful unit**
- It is a **property or characteristic of an individual module**

## COUPLING

- The measure of **interdependence** of one **module to another**.
- It describes the **relationship** between **software components**
- It is a **property of a collection of** modules

# Patterns

A pattern is —a **common solution to a common problem** in each context. While **architectural styles** can be viewed as patterns describing the high-level organization of software (their macro architecture), other design patterns can be used to describe details at a lower, level (their microarchitecture).

- Creational patterns (example: builder, factory, prototype, and singleton)
- Structural patterns (example: adapter, bridge, composite, decorator, façade, flyweight, and proxy)
- Behavioral patterns (example: command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor)

# Design Pattern

Design Pattern enables a designer to determine whether the pattern :

- is applicable to the **current work**
- can be **reused**
- can serve as a guide for **developing a similar but functionally** or structurally **different pattern**.

# Singleton Pattern

- The **singleton** pattern is one of the simplest **design patterns**: it involves only one class which is responsible to **instantiate itself**, to make sure it creates not more than **one instance**.

- Sometimes it's important to have only **one instance for a class**. For example, in a system there should be only one **window manager** (or only a file system or only a print spooler).

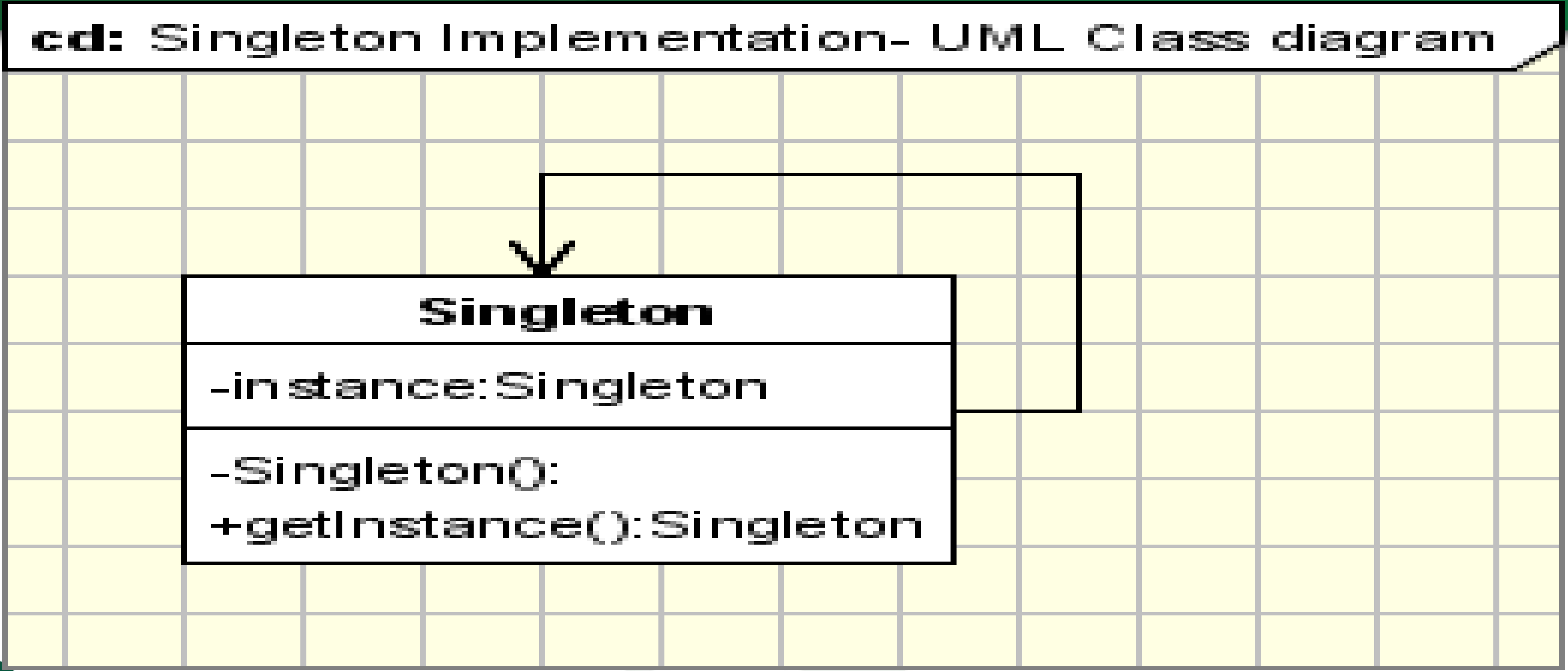
- Usually, singletons are used for **centralized management** of **internal or external resources**, and they provide a global point of access to themselves.

# Singleton Pattern

## Implementation

- The implementation involves a **static member in the "Singleton" class**, a private constructor and a static public method that returns a reference to the static member.

- The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.


cd: Singleton Implementation- UML Class diagram
Singleton
-instance:Singleton
-Singleton():
+getInstance():Singleton

# Some Design Principles

- 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 be structured to **accommodate change**.
- Design is not **coding**, coding is not **design**.

# What is Next…???

- Continue on Software Design
- Object-Oriented Design - UML Class & Sequence Diagrams
- User Interface Design
- Prototype session # 3

# Questions …..