MAPÚA MALAYAN COLLEGES LAGUNA

COLLEGE OF COMPUTER AND INFORMATION SCIENCE
MAPÚA MALAYAN COLLEGES LAGUNA

CS154-1P
SOFTWARE ENGINEERING

Designing the Architecture

WEEK 5

Topics:

- **Designing the Architecture**
  - The Design Process
  - Modeling Architectures
  - Decomposition and Views
  - Architectural Styles and Strategies

Design Process

Software Design

- Deriving a solution which satisfies software requirements

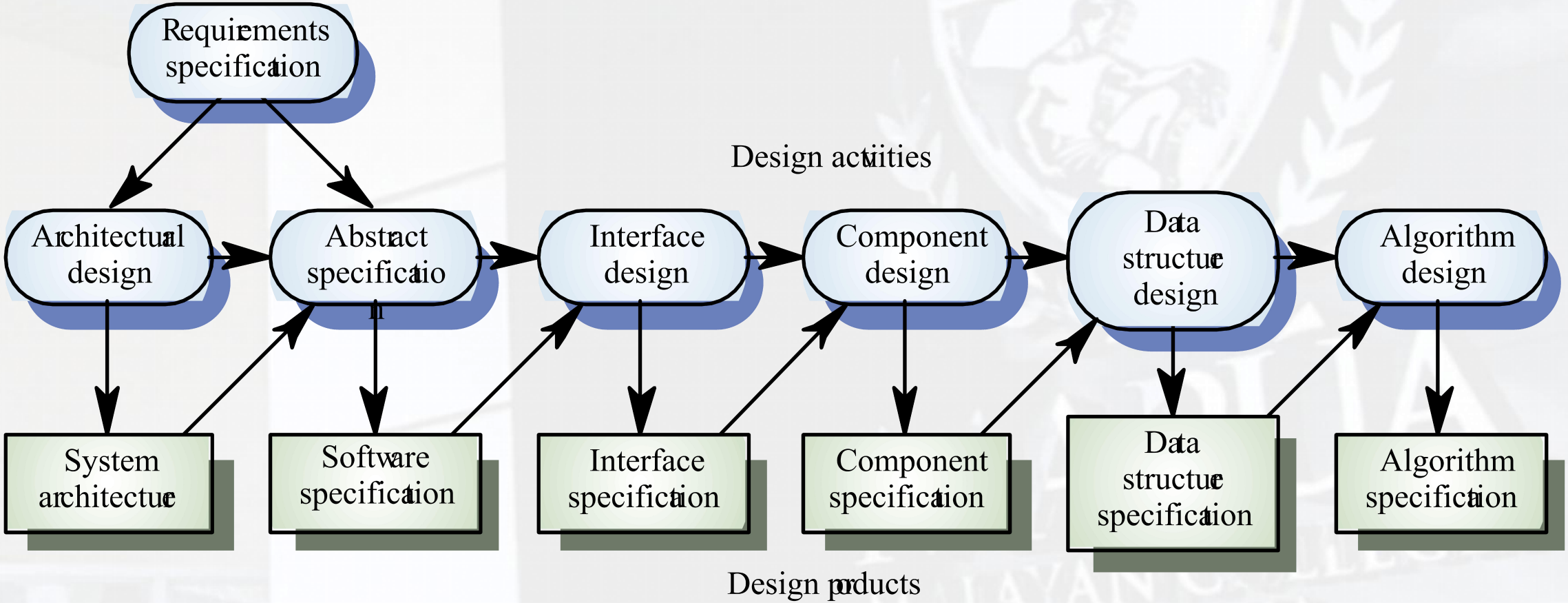
The **Design Process**

- **Any design may be modelled as a directed graph made up of entities with attributes which participate in relationships.**
- The system should be described at several different levels of abstraction.
- Design takes place in overlapping stages. It is artificial to separate it into distinct phases but some separation is usually necessary.

Design Phases

- ***Architectural design:*** Identify sub-systems.
- ***Abstract specification:*** Specify sub-systems.
- ***Interface design:*** Describe sub-system interfaces.
- ***Component design:*** Decompose sub-systems into components.
- ***Data structure design:*** Design data structures to hold problem data.
- ***Algorithm design:*** Design algorithms for problem functions.

Phases in the Design Process



Design

- **Computer systems are not monolithic:** they are usually composed of multiple, **interacting modules.**
- Modularity has long been seen as a key to cheap, high quality software.
- The **goal of system design** is to decode:
  - What the modules are;
  - What the modules should be;
  - How the modules interact with one-another

Modular programming

- In the early days, modular programming was taken to mean constructing programs out of small pieces: “subroutines”
- But modularity cannot bring benefits unless the modules are
  - autonomous,
  - coherent and
  - robust

Procedural Abstraction

- The most obvious design methods involve functional decomposition.
- This **leads to programs in which procedures represent distinct logical functions in a program.**
- Examples of such functions:
  - "Display menu"
  - "Get user option"
- This is called procedural abstraction

Object-oriented design

- The system is viewed as a collection of interacting objects.
- The system state is decentralized and each object manages its own state.
- Objects may be instances of an object class and communicate by exchanging methods.

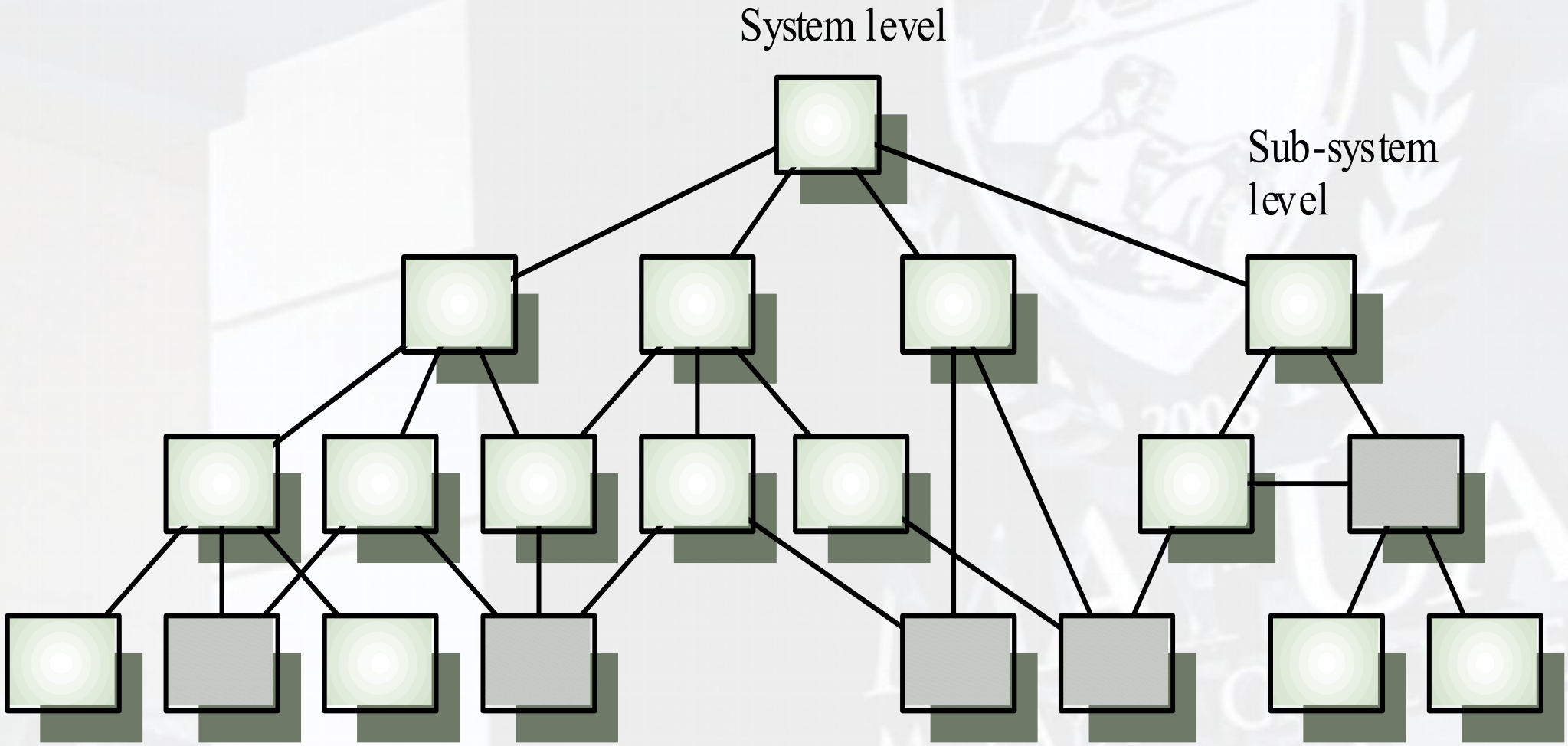
Five Criteria for Design Methods

- We can identify **five criteria** to help evaluate modular design methods:
  1. Modular decomposability;
  2. Modular composability;
  3. Modular understandability;
  4. Modular continuity;
  5. Modular protection.

1. Modular Decomposability

- **This criterion is met by a design method if the method supports the decomposition of a problem into smaller sub-problems, which can be solved independently.**
- In general method will be repetitive: sub-problems will be divided still further
- Top-down design methods fulfil this criterion; stepwise refinement is an example of such method

Hierarchical Design Structure



2. Modular Composability

- **A method satisfies this criterion if it leads to the production of modules that may be freely combined to produce new systems.**
- Composability is directly related to the issue of reusability
- Note that composability is often at odds with decomposability; top-down design,
  - for example, tends to produce modules that may not be composed in the way desired
- This is because top-down design leads to modules which fulfil a specific function, rather than a general one

3. Modular Understandability

- **A design method satisfies this criterion if it encourages the development of modules which are easily understandable.**
  - COUNTER EXAMPLE 1. Take a thousand lines program, containing no procedures; it's just a long list of sequential statements. Divide it into twenty blocks, each fifty statements long; make each block a method.

4. Modular Continuity

- **A method satisfies this criterion if it leads to the production of software such that a small change in the problem specification leads to a change in just one (or a small number of) modules.**
- EXAMPLE. Some projects enforce the rule that no numerical or textual literal should be used in programs: only symbolic constants should be used
- COUNTER EXAMPLE. Static arrays (as opposed to open arrays) make this criterion harder to satisfy.

5. Modular Protection

- **A method satisfied this criterion if it yields architectures in which the effect of an abnormal condition at run-time only effects one (or very few) modules**
- EXAMPLE. Validating input at source prevents errors from propagating throughout the program.
- COUNTER EXAMPLE. Using int types where subrange or short types are appropriate.

Software Architecture

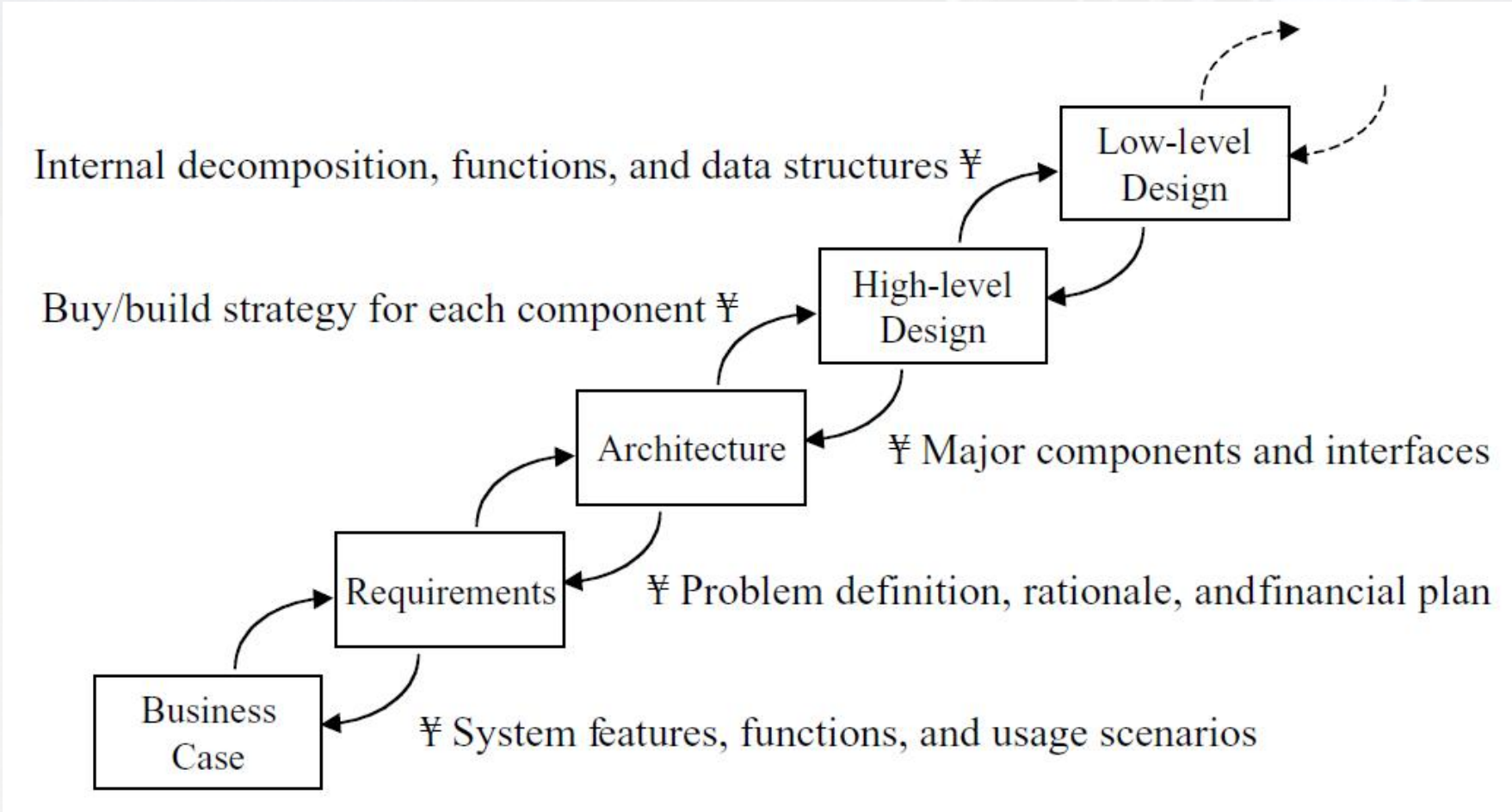
Architecture
(simplest definition)

- A **representation of a system** in which there is a:
  - mapping of functionality onto hardware and software components
  - mapping of the software architecture onto the hardware architecture,
  - and human interaction with these components.
- Maybe existing or to be created

Software Architecture

- The study of software components includes their **properties, relationships, and patterns of combination**
- Also, a particular set of software components as combined in a particular software system

The Role of Architecture



System Architecture and Platform

Platform

- framework, either in hardware or software, which allows software to run.

- Typical platforms include a computer's architecture, operating system, or programming languages their runtime libraries, and other “enableware”

Common Platforms

- IBM-Mainframe (MVS-APPC-DB/2-CICS)
- Java
  - java application and JVM, Java ME for remote
- .Net
- PC
  - Win
  - Mac
- Smartphones

Designing your architecture

- Technical issues (environmental requirements)
  - Existing and technology direction
  - Response time
  - Security
- Distribution
  - Fat or thin client
  - Multi-tier application
- Location and timing

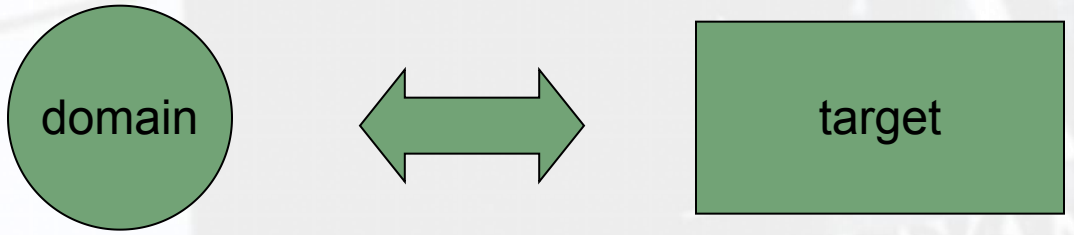
Basically

- Volume
  - transactions/process
  - data
  - users
- Location
- Roles

Mapping

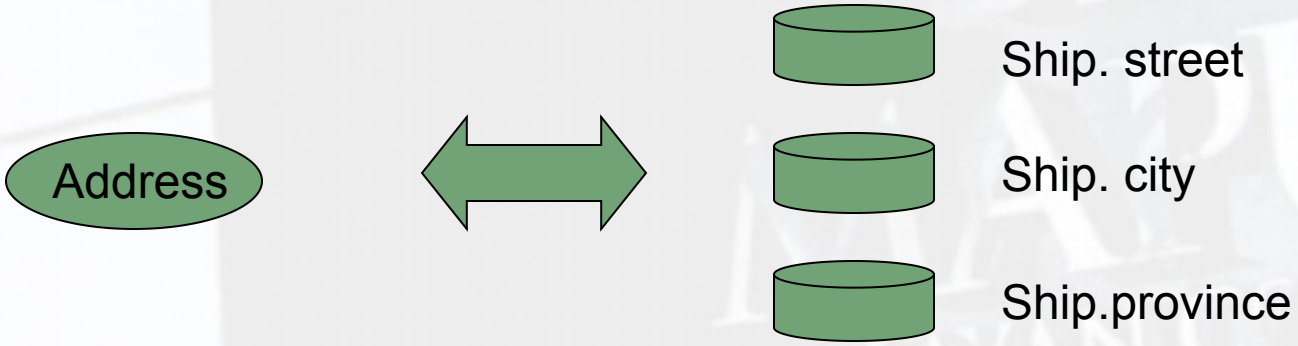
Mapping

- **One to One**
  - function/component/data element has a one to one correspondence with the target domain
  - A very ideal situation that present no problem of conversion/formatting



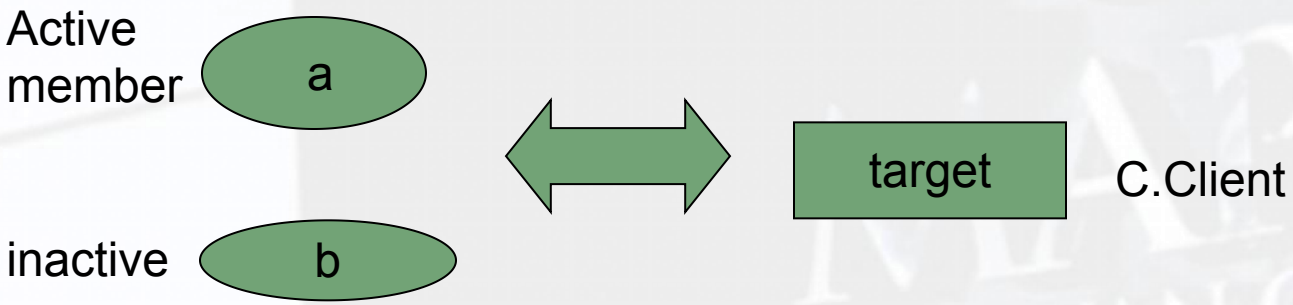
Mapping

- **One to Many**
  - The component will be split into different new components
  - A characterization rule should be consistent or defined
  - No major exposure here



Mapping

- **Many to One**
  - Several components will be merged into a single new component
  - Again, a characterization rule should be defined.
  - Historical data or functionality will be at risk since, reversing the process need extra steps/rule



Logical Design

Completing the functional specification of an application

describes required behavior in terms of required activities, such as reactions to inputs, and the state of each entity before and after an activity occurs.

System Functional Specification

- **Transform system requirements into detailed functional specification**
  - What the system will do?
  - How the system will be implemented through the creation of an appropriate application
- Creating the logical design, application architecture, or application domain specification

Cohesion

level of strength and unity with which different components of a software program are inter-related with each other

Types of Cohesion model

- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion
- Sequential cohesion
- Functional cohesion

Types of Cohesion model

- **Coincidental cohesion:**
  A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, **the module contains a random collection of functions. It is likely that the functions have been put in the module out of pure coincidence without any thought or design.**
    - For example, in a transaction processing system (TPS), the get-input, print-error, and summarize-members functions are grouped into one module. The grouping does not have any relevance to the structure of the problem.
- **Logical cohesion:**
  A module is said to be logically cohesive, if all elements of the module **perform similar operations, e.g. error handling, data input, data output, etc.**
    - An example of logical cohesion is the case where a set of print functions generating different output reports are arranged into a single module.

Types of Cohesion model

- **Temporal cohesion:**
  When a **module contains functions that are related by the fact that all the functions must be executed in the same time span**, the module is said to exhibit temporal cohesion.
    - The set of functions responsible for initialization, start-up, shutdown of some process, etc. exhibit temporal cohesion.
- **Procedural cohesion:**
  A module is said to possess procedural cohesion, if the set of functions of the **module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective**
    - e.g. the algorithm for decoding a message.

Types of Cohesion model

- **Communicational cohesion:**
  A module is said to have communicational cohesion, if all functions of **the module refer to or update the same data structure**
    - e.g. the set of functions defined on an array or a stack.
- **Sequential cohesion:**
  A module is said to possess sequential cohesion, if the elements of **a module form the parts of sequence, where the output from one element of the sequence is input to the next.**
    - For example, in a TPS, the get-input, validate-input, sort-input functions are grouped into one module.
- **Functional cohesion:**
  Functional cohesion is said to exist, if **different elements of a module cooperate to achieve a single function.** For example, a module containing all the functions required to manage employees' pay-roll exhibits functional cohesion.
    - Suppose a module exhibits functional cohesion and we are asked to describe what the module does, then we would be able to describe it using a single sentence

Coupling

Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong)

Some types of coupling
In order of highest to lowest coupling

- Content coupling (high)
- Common coupling
- External coupling
- Control coupling
- Data coupling
- Message coupling (low)
- No coupling

Types of Coupling

- **Content coupling (high)**
  Content coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module).
    - Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.
- **Common coupling**
  Common coupling is when two modules share the same global data (e.g., a global variable).
    - Changing the shared resource implies changing all the modules using it.
- **External coupling**
  External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

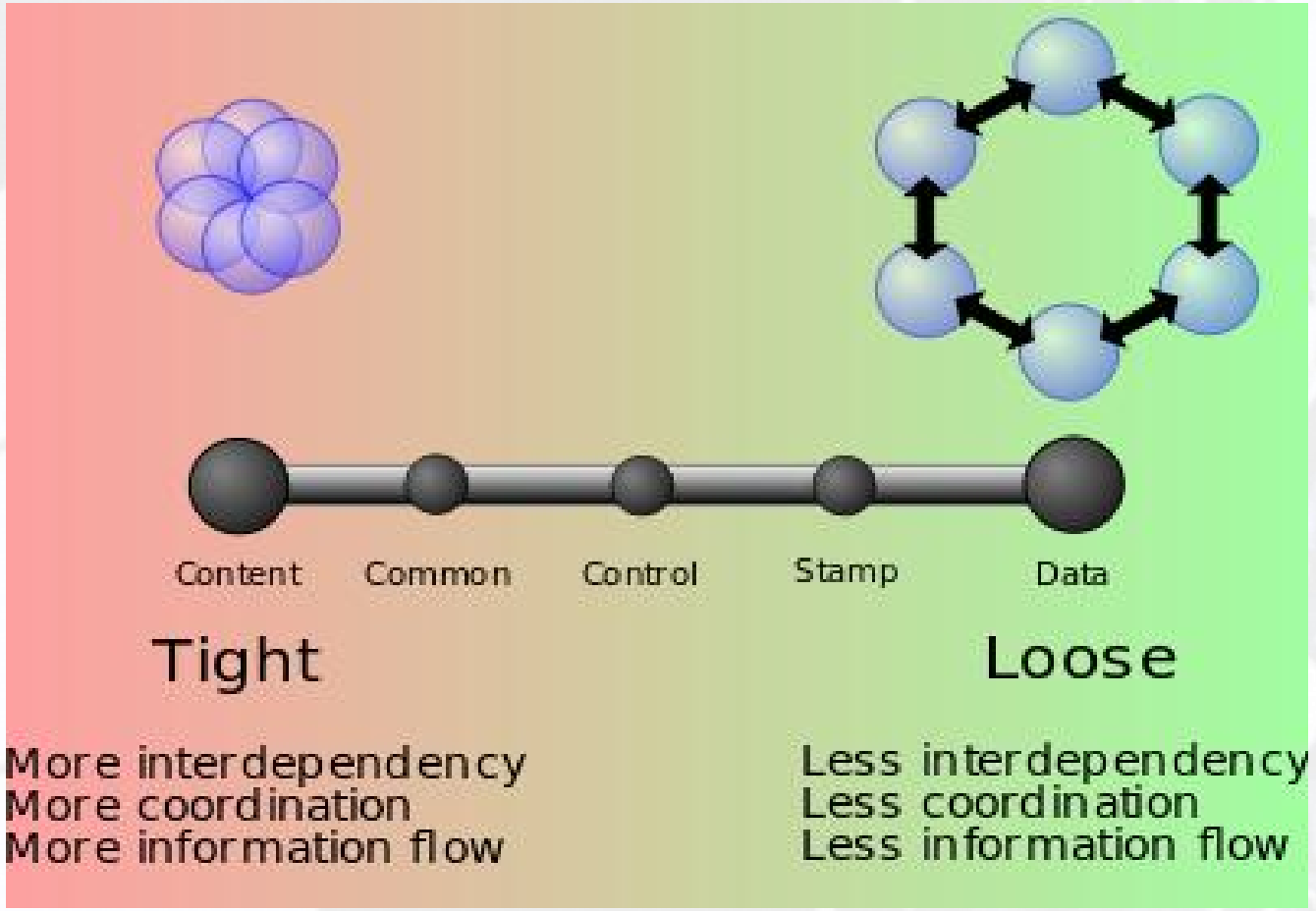
Types of Coupling

- **Control coupling**
  Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).
  - Stamp coupling (Data-structured coupling). Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it). This may lead to changing the way a module reads a record because a field, which the module doesn't need, has been modified.
- **Data coupling**
  Data coupling is when modules share data through, for example, parameters.
  - Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Types of Coupling

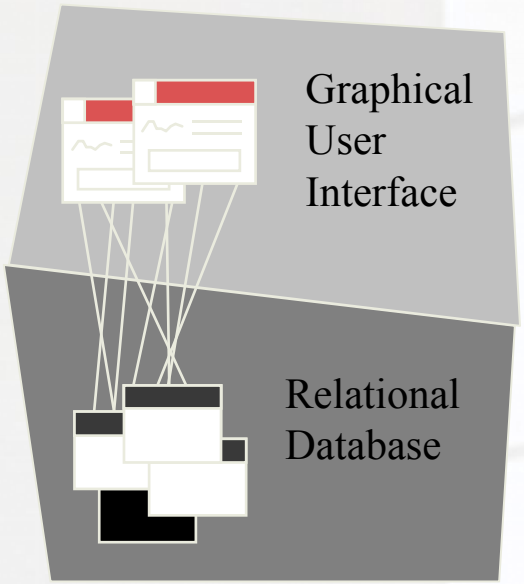
- **Message coupling (low)**
  This is the loosest type of coupling. It can be achieved by state decentralization(as in objects)and component communication is done via parameters or message passing.
- **No coupling**
  Modules do not communicate at all with one another.

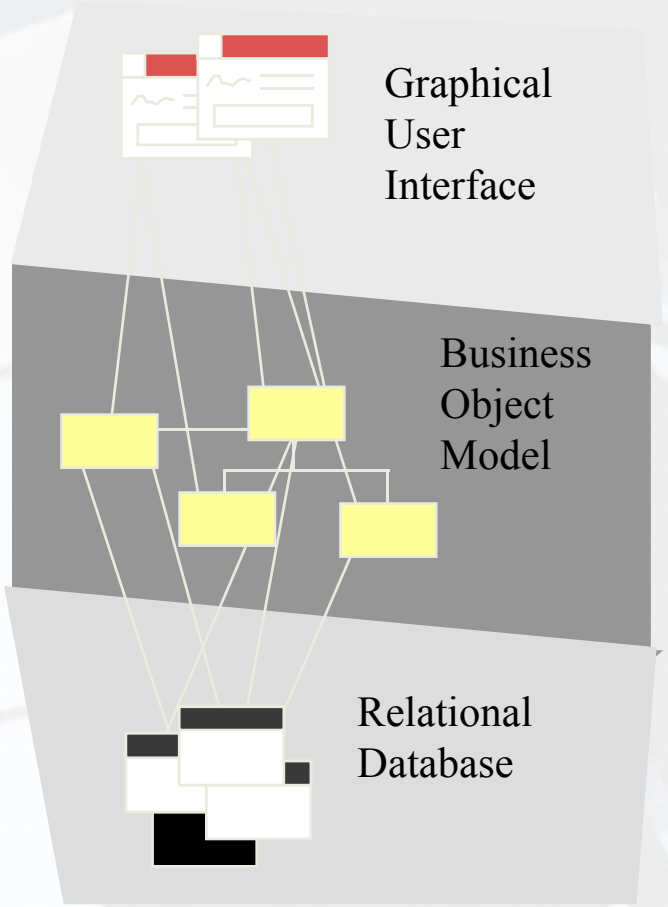
Conceptual model of coupling

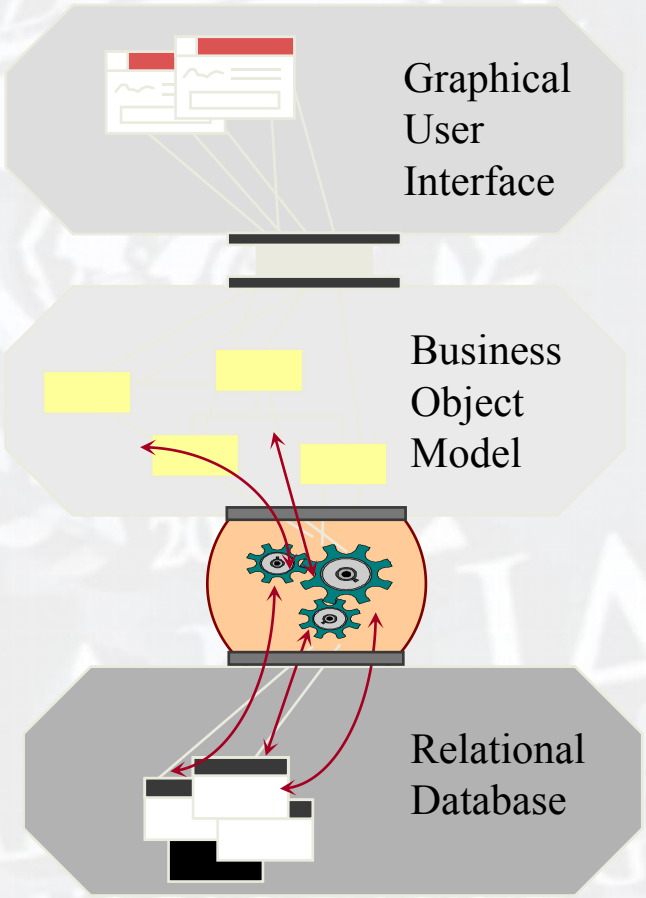


Software Architecture Diagrams

Logical Application Architecture







End