Elements

* Objects have state
  + defined by the value of its attributes and its relationships with other objects at a particular point in time.
* Each object has methods
* Specifying what processes the object can perform.
* In order to get an object to perform a method,
  + a message is sent to the object.
* A message
  + a function or procedure call from one object to another object.

Encapsulation

* Combine processes and data into a single object
* Hide the content of the object from the outside view
* Communication only through object methods
* Key to reusability
  + It shields internal workings of an object from changes in the outside system
  + It keeps the system from being affected when changes are made to an object

Polymorphism

* Same message triggers different methods in different objects
* Dynamic binding means specific method is selected at run time
* Need to be very careful about run time errors
* Need to ensure semantic consistency

Inheritance

* Single inheritance -- one parent class
* Multiple inheritance -- multiple parent classes
* Redefinition and inheritance conflict
* Most inheritance conflicts are due to poor

classification

* Inheritance conflicts and redefinition can cause all kinds of problems with interpreting the final design and implementation.
* Most inheritance conflicts are due to poor classification of the subclass in the inheritance hierarchy
  + i.e., the generalization A-Kind-Of semantics are violated,
* or the actual inheritance mechanism violates the encapsulation principle
  + i.e., subclasses are capable of directly addressing the attributes or methods of a superclass.

Requirement Analysis

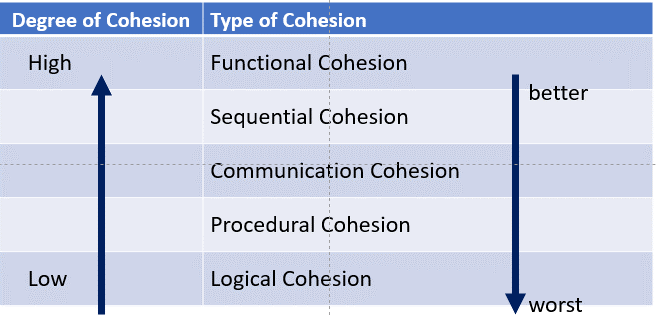
* Requirement can be divided into two major types:
  + Functional
  + Non-functional
  + Functional requirement describes what the system should do.
  + Functional requirement discusses the functionality required by the users from the system
  + Functional requirement should include:
  + Everything that a user would need to know regarding what the system does
  + Everything that would concerns any other system that has interface to this system
  + Functional requirement can be:
  + What inputs the system should accept, and under what conditions
  + What outputs the system should produce, and under what conditions
  + What data the system should store that other systems might use
  + What communication channel the system might relay on
  + Functional requirement can be:
  + What inputs the system should accept, and under what conditions
* What outputs the system should produce, and under what conditions
* What data the system should store that other systems might use
* What communication channel the system might relay onNon-Functional requirement is a statement of how a system must behave
* Quality constraint:
* Maintainability and Enhancement
* Reusability
* Reliability
* Performance constraint
* Response time
* Throughput
* Utilization
* Non-Functional requirement is a statement of how a system must behave
* Design constraint
* Availability (should not be down for more than 5mn)
* Recovery from failure (recover within a certain amount of time and minimal loss of data)
* Environmental / platform constraint
* Platform (i.e iOS, andriod, Linux)
* Technology to be used (i.e programing language)
* Process Constraint
* Development process to be used [waterfall](https://en.wikipedia.org/wiki/Waterfall_model), [prototyping](https://en.wikipedia.org/wiki/Software_prototyping), [iterative and incremental development](https://en.wikipedia.org/wiki/Iterative_and_incremental_development)
* Cost and delivery date
* Problem Specification
* Start a problem specification with the following steps:
* List Nouns and Verbs
* Identify Things Outside the Scope of the System
* Identify Synonyms
* Identify Potential Classes
* Identify Potential Attributes
* Identify Potential Methods
* Identify Common Characteristics
* Refine Design using CRC Cards
* Elaborate Classes

**Modularization**

* In Software Engineering with OO, modularization is important to divide a software system into multiple independent modules where each module works independently
* Advantages of Modularization in software engineering
  + Easy to understand the system.
  + System maintenance is easy.
  + A module can be used many times as their requirements.
  + No need to write it again and again
* Coupling
* Cohesion

**Cohesion**

* The togetherness of a module’s elements
* In object oriented design, cohesion refers all about how a single class is designed to meet its purpose
* The more focused a class is, the more cohesive the class is
* The cohesion actually helps to measure the strength of bonding between the different methods and data of a class
* Cohesion deals with the interconnection between the elements of the same module
* Low Cohesion – functionalities of a module are independent of each other
* High Cohesion – functionalities of a module are strongly related
* We have a class that multiply two numbers, but the same class creates a pop up window displaying the result.
* This is the example of low cohesive class because the window and the multiplication operation don’t have much in common
* To make it high cohesive, we would have to create a class Display and a class Multiply.
* The Display will call Multiply’s method to get the result and display it. This way to develop a high cohesive solution.
* In low cohesion only one class is responsible to execute lots of job which are not in common and reduces the chance of re-usability and maintenance.
* In high cohesion there is a separate class for all the jobs to execute a specific job, which result better usability and maintenance
* Advantages of High Cohesion
  + Increase module reusability – developer can easily search module and reuse
  + Increase system maintainability
  + More robustness and reliability – module is smaller and simpler
  + More readability and less complex



Functional Cohesion

A functionally cohesive module performs one and only one problem related task

Module with functional cohesion perform exactly one action

Considered as the best cohesion

Why functional cohesion is good?

More reusable

Corrective maintenance easier

Easier to extend product

Examples:

Calculate sale commission

Verify customer number

Compute net salary

Sequential Cohesion

An element (method) of a module (class) produce some data which become the input of other element of the module

It is easy maintenance and provides a good coupling

It is not good for reusability because the activities generally only meet the requirements of that one module.

Example

-> retrieve customer

--> retrieve customer order

---> generate invoice

Communication Cohesion

A communicationally cohesive module is one which performs several functions on the same input or output data

It is an acceptable level of cohesion although it is not as good as functional or sequential cohesion

Other modules may require only part of the input or output data of the communicationally cohesive module, making the unnecessary data redundant

Difficult to make changes part of the code without affecting the functionality of another part of the code

Procedural cohesion

A *procedurally cohesive* module is one whose elements are involved in different activities, but the activities are sequential.

Activities in a procedurally cohesive module are related by flow of execution rather than

by one problem-related function

Not reusable Not maintainable Logical cohesion

A *logically cohesive* module is one whose elements perform similar activities and in which the activities to be executed are chosen from outside the module.

Explanation: The actions in the modules are all logically “transaction processing”. The “which” flag tell the module what part of its internal logical to apply to particular transaction data coming in for each specific call.

* In object oriented design, Coupling refers to the degree of direct knowledge that one element has of another.

necessary to perform functions

Never pass a data structure containing many fields to a module that only needs a few

Strength: it simplifies interfaces between modules

Weakness: It promotes the creation of artificial data structures which some data are not used

Control Coupling

Control coupling occurs between modules when data are passed that influence the internal logic of another module (e.g., flags and switches)

A module that retrieves either a customer name or an address depending on the value of a flag is illustrated

Common Coupling

Common coupling occurs when modules communicate using global data areas (i.e., universal common data areas). C allows the developer to declare a data element as external, enabling it to be accessed by all modules. Avoid using global data because global data may be updated by any module at any time. Modules referencing global data are tied to specific data names, unlike modules that reference data through parameters. Thus limits the module's re-usability

It takes time for programmer to determine what module updated the global data

Content Coupling

Two modules are content coupled if:

one module changes a statement in another

one module references or alters data contained inside another module

one module branches into another module

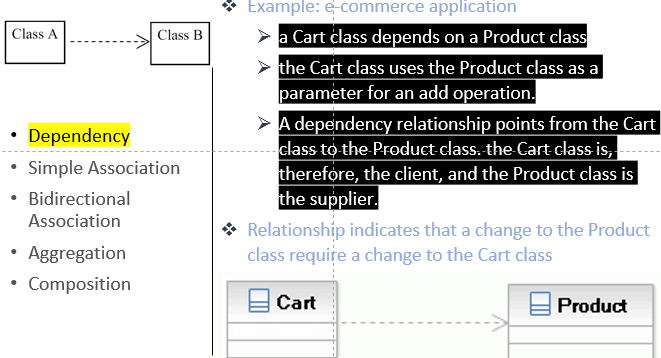
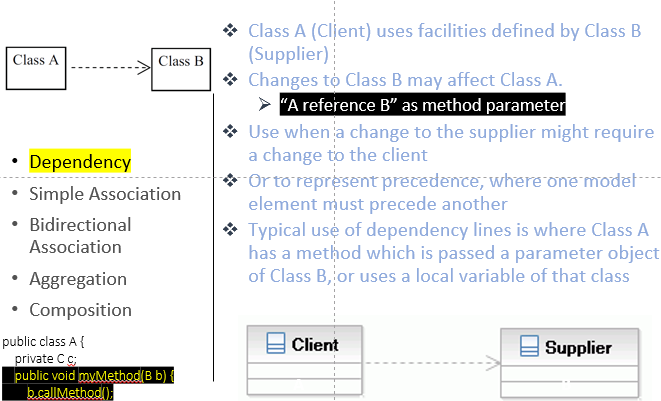
Avoid Content Coupling

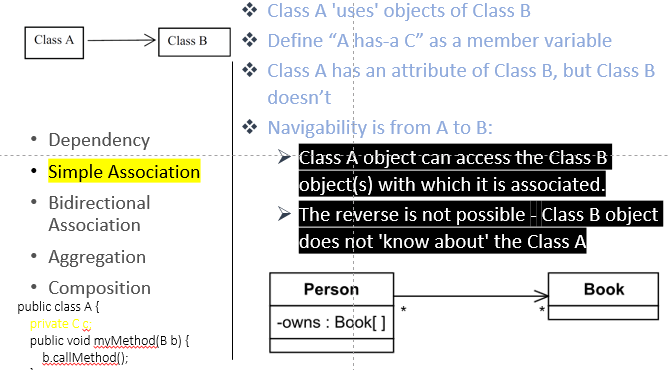
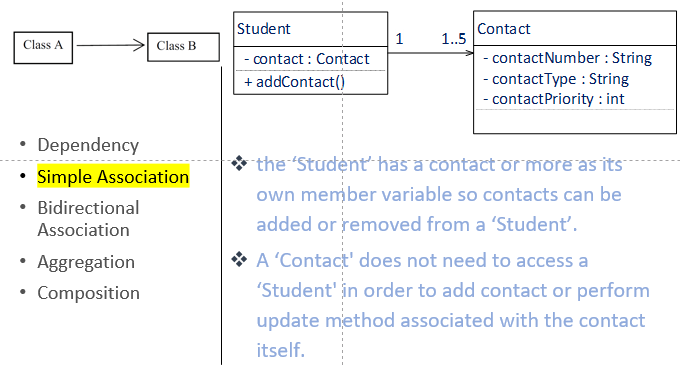
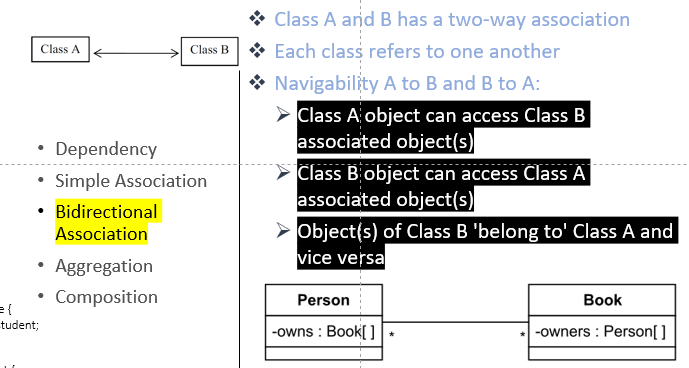
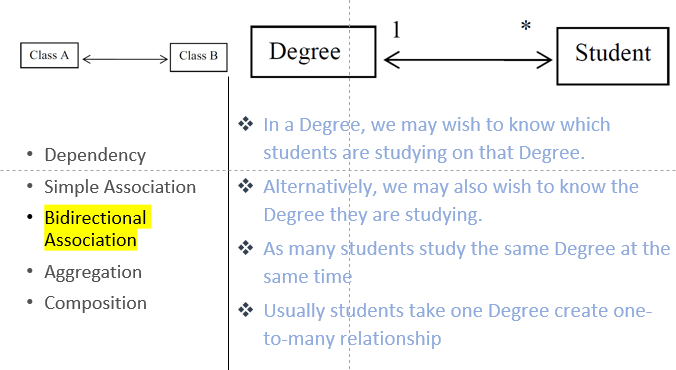
* + UML stands for “Unified Modelling Language”
  + UML is a standard language for specifying, visualizing, constructing and documenting the artifacts of software systems.
  + UML is different from the other programming languages like Java, C++ but tools can be used to generate code in various languages using UML diagrams.

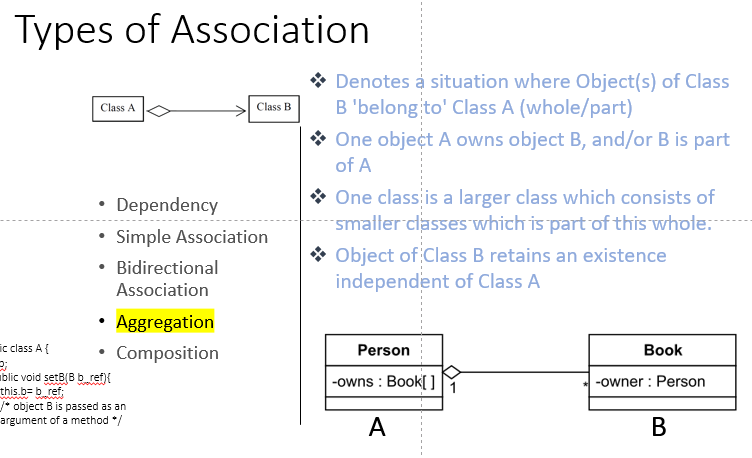
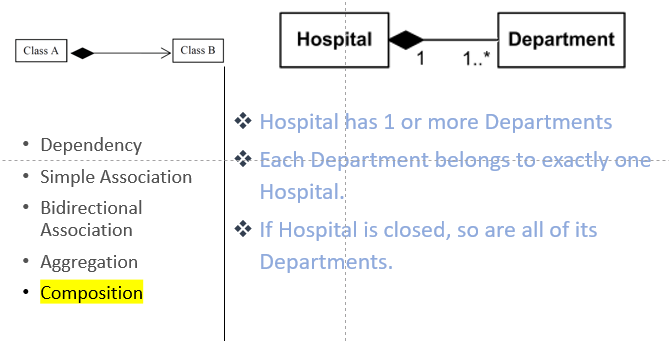
Modelling describe a system at a high level of abstraction. Used for requirement and specification. A model of system.

Dependency



* In other words, how often do changes in class A force related changes in class B
* Low coupling minimizes the "ripple effect" where changes in one module cause errors in other modules.
* Coupling deals with the interdependence between software modules
* There are two types of coupling:
  + Tight coupling (high coupling)
  + Loose coupling (low coupling)
* Use Case Diagram is a graphical representation of what a system must do
* It models the functionality of a system using actors and use-cases
* Uses simplified term so to be easily understood by all stakeholders
  + To visualize the functional requirements of a system that will translate into design and development
  + To identify any internal or external factors that may influence the system
  + To provide a high level analysis from outside the system
* Use case diagram specifies how the system interacts with actors without worrying about the details of how that functionality is implemented.
* Systems (system domain):
  + the program needs to be developed comprising use cases
* Actors
  + Specify a role played by someone or some other system that interacts with the subject
* Use Cases (goal):
  + set of actions, services, and functions that the system needs to perform
* Relationships
  + to indicate interaction between actors and use cases



* Tight coupling:
  + the two classes often change together
  + if A knows more than it should about the way in which B was implemented, then A and B are tightly coupleds
* Loose coupling:
  + loose coupling means they are mostly independent
  + If the only knowledge that class A has about class B, is

what class B has exposed

through its interface, then

class A and class B are said

to be loosely coupled

there is no dependency between both the classes. If we change anything in the Box classes then we dont have to change anything in Volume class

Which is better tight coupling or loose coupling?

* Tight Coupling is bad
  + it reduces flexibility and re-usability of code,
  + it makes changes much more difficult,
  + it prevents test ability
* Loose coupling is a better choice
  + A loosely coupled will help you when your application need to change or grow.
  + If you design with loosely coupled architecture, only a few parts of the application should be affected when requirements change.

Data Coupling

Two modules are data coupled if they communicate by passing parameters.

This is a "good design principle" for your programming.

Use data coupling when the argument list is small.

Limit the argument list to three items.

Strengths: A module sees only the data elements it requires

Weakness: A module is difficult to maintain if many data elements are passed. Too many parameters indicate poorly design

Stamp Coupling

Two modules communicate via a passed data structure that contains more information than