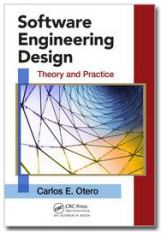
CHAPTER 2: SOFTWARE DESIGN WITH THE UNIFIED MODELING LANGUAGE

SESSION I: UML FUNDAMENTALS

Software Engineering Design: Theory and Practice
by Carlos E. Otero

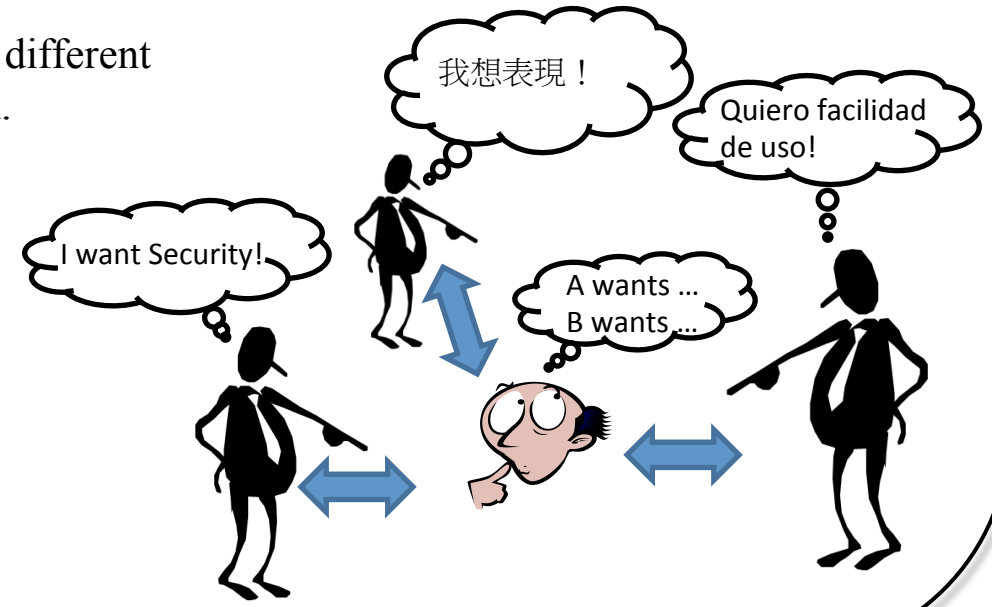


SESSION'S AGENDA

- UML Fundamentals
  - ✓ History, goals, etc.
  - ✓ Classifiers
  - ✓ Relationships
  - ✓ Enhancing features
- UML Diagrams
  - ✓ Structural diagrams
  - ✓ Behavioral diagrams
- UML Summary
  - ✓ What's next…

UNIFIED MODELING LANGUAGE FUNDAMENTALS

- Communication is an essential, critical skill for engineers.
  - ✓ Throughout a project's life-cycle, designers spent a great deal of time and effort communicating with other members of the project.
    - Oral
    - Written
  - ✓ In previous sessions, we discussed the importance of managing design influences.
    - Imagine, if stakeholders used different languages for communication.
  - ✓ Different languages of communication would decrease efficiency and effectiveness.



UNIFIED MODELING LANGUAGE FUNDAMENTALS

- The UML is the result of years of collaborative work spent in devising a unified approach for modeling systems.
  - ✓ The first efforts focused on unifying three popular modeling methods:
    - The Booch method, by Grady Booch
    - The object-oriented software engineering method (OOSE), by Ivar Jacobson
    - The object modeling technique (OMT) by James Rumbaugh

- The goals of the unification project were specified by Booch, Rumbaugh, and Jacobson as follows:
  1. To model systems, from concept to executable artifact, using object-oriented techniques.
  2. To address the issues of scale inherent in complex, mission-critical systems.
  3. To create a modeling language usable by both humans and machines.

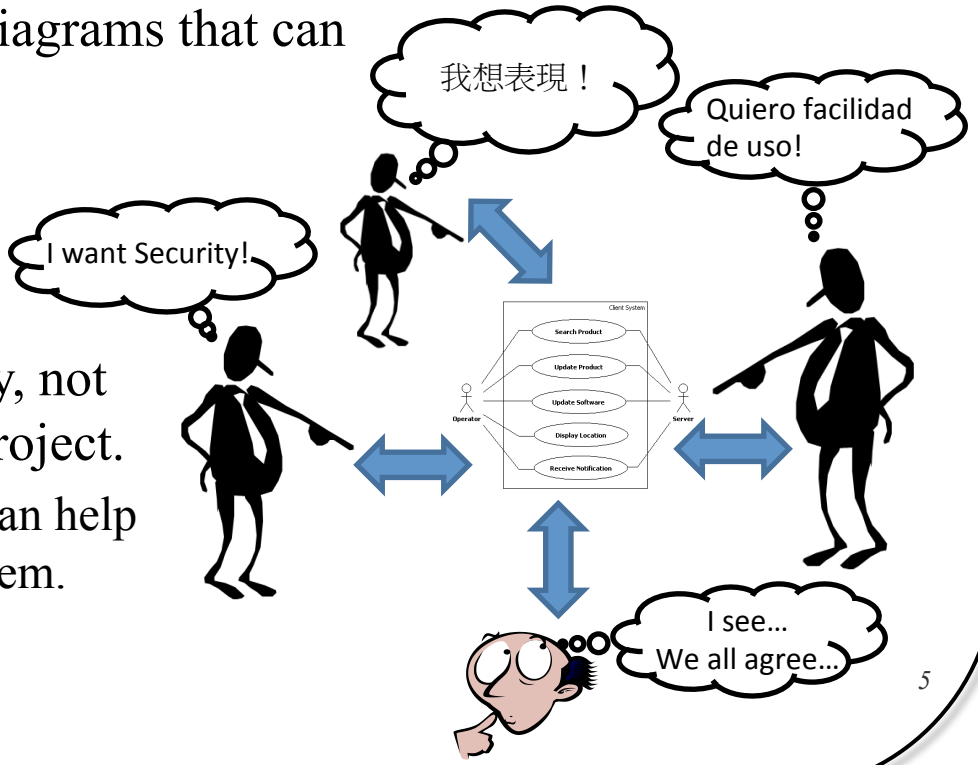
- The development of early UML versions generated interest among numerous influential organizations.
  - ✓ Microsoft, Oracle, IBM, Rational, etc. This collaborative effort resulted in UML 1.0.
  - ✓ After revisions, it was accepted by the OMG as UML 1.1.
  - ✓ Since then, UML has evolved and accepted heavily in industry.

UNIFIED MODELING LANGUAGE FUNDAMENTALS

- Formally, UML can be defined as a visual language for specifying, analyzing, and documenting design elements essential for modeling the dynamic and static aspects of software systems before construction.

- UML 2.3 provides 14 types of diagrams that can be used for modeling both
  - ✓ Structural
  - ✓ Behavioral

- Because projects vary drastically, not all diagrams are used in every project.
  - ✓ Designers select the ones that can help them effectively model the system.



UNIFIED MODELING LANGUAGE FUNDAMENTALS

- Officially, UML Structural Diagrams
  - ✓ Concerned with capturing and specifying static elements and their interrelationships required for supporting the solution to a given problem, within a given context.

- Behavioral Diagrams
  - ✓ Concerned with capturing and specifying the dynamic behavior and the inherent complexities present in the behavioral aspects of software systems.

UNIFIED MODELING LANGUAGE FUNDAMENTALS

- To model almost any aspect of today's modern system, UML was developed with flexibility and extensibility in mind.
  - ✓ However, the fundamental building blocks are well defined and must be understood before applying UML effectively.

- UML's building blocks are grouped into:
  - ✓ Classifiers
    - Structural things that represent conceptual or physical elements of a model.
    - They are typically the main elements in UML diagrams.
    - Each type of UML diagram, uses specific type of classifiers, so that not all classifiers are relevant to all UML diagrams.
  - ✓ Relationships
    - Defines and provides visualization of the interconnections that exists among classifiers.
  - ✓ Enhancing features
    - Provides flexibility that allow designers to enhance and evolve modeling capabilities so that they become appropriate for particular systems.

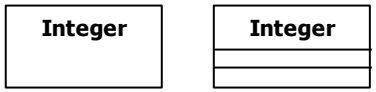
- Let's cover the building blocks in more detail…

UML 2.3 CLASSIFIERS

- UML 2.3 classifiers include:
  - ✓ Use Case
    - Classifier used to model a single required system behavior; represented with icons of elliptical shape.

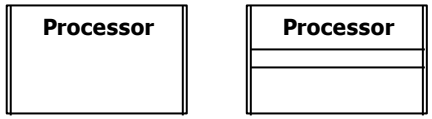
  - ✓ Component
    - Classifier used to represent a modular and replaceable part of the system; modeled using a box with the keyword <<component>> and optional component icon on the top right corner.

  - ✓ Class
    - Classifier used to model a type in terms of operations, attributes, relationships, and other semantics; modeled using a rectangular box. Typically, a class is split into compartments for attributes and operations.

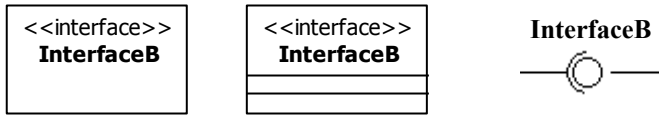


UML 2.3 CLASSIFIERS

- UML 2.3 common classifiers include (continued):
  - ✓ Active Class
    - Classifier used to model a class that owns an independent flow of execution and can initiate control activity; modeled as a class with double lines on each side.



  - ✓ Interface
    - Classifier that models the set of operations that specify the services provided by a class or component; represented as stereotyped classes or using the ball-and-socket notation.

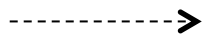


  - ✓ Node
    - Classifier used to model a physical element (e.g., a computer), its processing capabilities, and other characteristics; modeled using a cube.

  - ✓ Artifact
    - Classifier that models a physical deployable information element (e.g., .dll, .exe, .jar, .script, etc.); modeled using a rectangle with the keyword <<artifact>>,

UML 2.3 RELATIONSHIPS

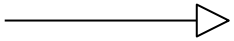
- Relationships apply to all UML Classifiers. UML relationships include:
  - ✓ Dependency
    - Dashed line (typically directed with a stick arrow) used to model the relationship between two UML classifiers indicating that changes to one element affect the other.

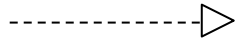
  - ✓ Association
    - Line used to model the relationship between two UML classifiers indicating that a connection exists between them; associations can be directed using a stick arrow.

  - ✓ Generalization
    - Line with a hollow arrowhead used to model the relationship between two UML classifiers indicating that one (child) inherits from another (parent).

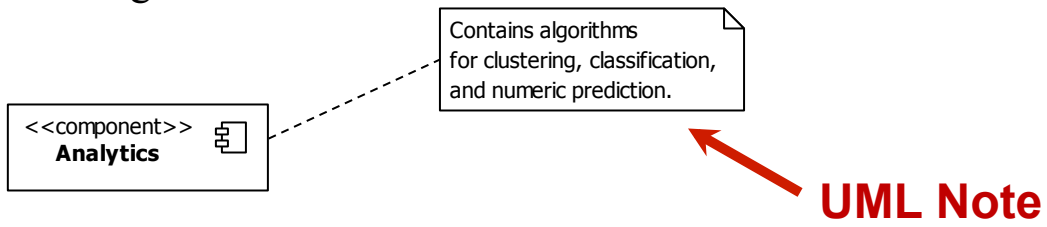
      

  - ✓ Realization
    - Relationship between two UML classifiers indicating that one element realizes a specified interface; modeled using a dashed line with hollow arrowhead.

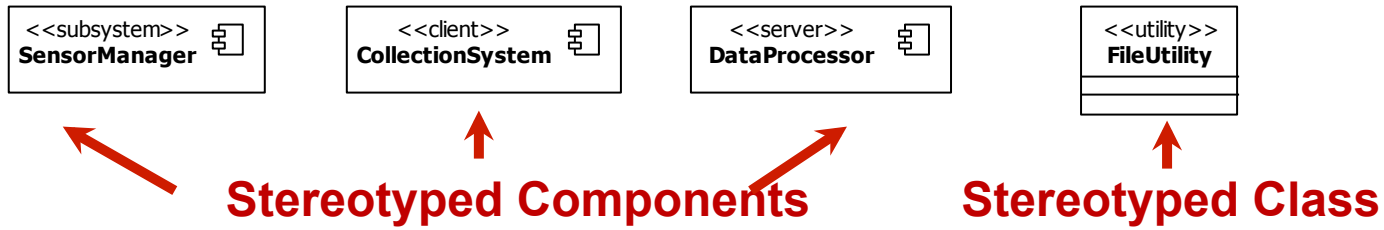
      

UML 2.3 Enhancing Features

- Common UML mechanisms for enhancement:
  - ✓ Notes
    - Mechanism for adding descriptive information to UML elements (both classifiers and relationships) and diagrams; modeled using a rectangle with a dog-eared corner and can be connected using a dashed line.



  - ✓ Stereotypes
    - Mechanism for extending UML by adding information that gives existing UML elements (both classifiers and relationships) a different meaning, therefore creating a semantically different element for modeling application-specific concepts; modeled as existing UML elements with the <<stereotype>> mechanism.



UML 2.3 Enhancing Features

- Common UML mechanisms for enhancement (continued):
  - ✓ Tagged values
    - Mechanism for adding new properties to a stereotype; modeled by adding the tagged value in the form of *property* = *value* to existing stereotyped UML elements.

  - ✓ Constraints
    - Mechanism for specifying constraints to design elements (both classifiers and relationships); associated with specific design elements in the form of *{constraint description}*

UML 2.3 Diagrams - Structural

- Together, UML classifiers, relationships, and enhancement features can be used together in 14 diagrams to model systems from different perspectives and different concerns.

- The most common UML *structural diagrams* are presented below.

Structural Diagram	Description
Component Diagram	Used to model software as group of components connected to each other through well-defined interfaces.
Class Diagram	Used to model software as a set of classes, including their operations, attributes, and relationships.
Object Diagram	Used to model an instant snapshot of the life of an object during execution, including its state and attribute values.
Deployment Diagram	Used to model the physical realization of software systems, including physical nodes where software is deployed, interfaces between nodes, executable software artifacts, and the manifestation of software components.
Package Diagram	Used to model the decomposition of software as a set of packages, including relationships between packages.

- Other structural diagrams include:
  - ✓ Composite structure diagram
  - ✓ Profile diagram

UML 2.3 Diagrams - Behavioral

- The most common UML *behavioral diagrams* are presented below.

Behavioral Diagram	Description
Use Case Diagram	Used to capture, specify, and visualize required system behavior .
Sequence Diagram	Used to capture, specify, and visualize system interactions with emphasis on the time-order sequence of messages exchanged.
Communication Diagram	Used to capture, specify, and visualize system interactions with emphasis on the structural order of entities participating in the message exchange.
State Machine Diagram	Used to capture, specify, and visualize system behavior as a set of discrete states and the transitions between them.
Activity Diagram	Used to capture, specify, and visualize system behavior; provide mechanisms for modeling that includes conditional statements, repetition, concurrency, and parallel execution and thus can be used at many different levels of abstraction, from modeling business work flows to code.

- Other behavioral diagrams include:
  - ✓ Timing diagram
  - ✓ Interaction overview diagram

UML SUMMARY

- UML is essential to enhancing system analysis, specification, and communication among a project's stakeholders. Specifically, UML
  - ✓ Provides a common language for analyzing, evaluating, and specifying systems.
  - ✓ Models can be created at higher levels and transferred downstream, for subsequent, finer-grained analysis, evaluation, and specification.
  - ✓ Models serve as the main tool for transferring knowledge and enhancing communication among stakeholders, including customers, managers, and programmers.
  - ✓ Enables visualization of complex systems and enables more efficient reasoning about the problem, therefore enhancing the problem-solving process.
  - ✓ Enhances design documents and enables reusability of solutions, which can be applied in future projects.

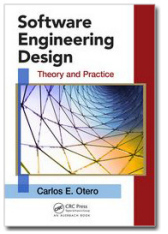
WHAT'S NEXT…

- In the next session we will discuss how UML classifiers, relationships, and enhancement features can be used to create structural diagrams. Specifically, we will focus on:
  - ✓ UML structural modeling
    - What is structural modeling?
    - Why is it important?
  - ✓ Component diagrams
    - Interfaces
    - Assembly connectors
    - Other common relationships
  - ✓ Class diagrams
    - Details of UML classes
    - Common relationships
    - The meaning (in code) of class diagrams
  - ✓ Deployment diagrams

CHAPTER 2: SOFTWARE DESIGN WITH THE UNIFIED MODELING LANGUAGE

SESSION II: UML STRUCTURAL MODELING

Software Engineering Design: Theory and Practice
by Carlos E. Otero



SESSION'S AGENDA

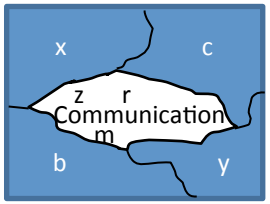
- UML structural modeling
  - ✓ What is structural modeling?
  - ✓ Why is it important?
- Component diagrams
  - ✓ Interfaces
  - ✓ Assembly connectors
  - ✓ Other common relationships
- Class diagrams
  - ✓ Details of UML classes
  - ✓ Common relationships
  - ✓ The meaning (in code) of class diagrams
- Deployment diagrams

Fundamentals of UML Structural Modeling

- What do we mean by *structural*? What is *structure*?
  - ✓ When we talk about structure, we talk about:
    - Parts arranged together in some way to compose some product.
    - In this conversation, the process for composition of the parts is important as well as the specification of relationships that glue these parts together.
  - ✓ From the software profession's point of view, what does this mean?
    - Stop and think about this!
    - Structure of code? Structure of computers in the system?

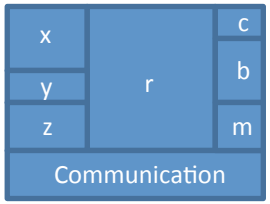
Remember modularization From previous sessions?

- Let's *assume that structure refers to the structure of code*. Consider the following conceptual structure of the same software application.



Is there any structure in these applications?

Sure, there is structure here. Different parts are arranged together to compose some product!



- Consider what would happen if we needed to reuse the communication function in some other project.
  - ✓ What can you say about the reusability of these systems?
  - ✓ What can you say about the maintainability of these systems?

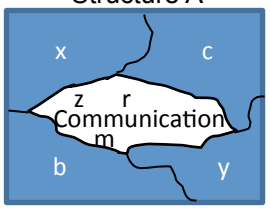
- We have not talk about the concept of **Quality** yet, but we will… For now, assume that quality is a function of reusability and maintainability. What can you say about the quality of these systems?
  - ✓ Under these assumptions structure drives quality AND quality drives structure.

How can this be true?

FUNDAMENTALS OF UML STRUCTURAL MODELING

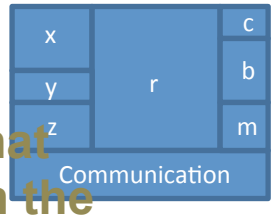
- In the previous slide, under the established assumptions, we mentioned that structure drives quality AND quality drives structure. What do we mean by that? Consider again the two structures for the same software application.
  - ✓ Assume that these are structures for a message processing application, where messages are sent and processed by the software.
  - ✓ Furthermore, assume the following:
    - Messages **need to be processed and executed in less than .5 seconds**.
      - Assume that ONLY *Structure A* leads to a system that meets this requirement.
    - Based on the customer, quality is a function of performance and NOT reusability and maintainability.

Structure A



What can you say about the quality of these systems?

Structure B



Reusability 

Maintainability 

Performance 

Quality 

Structure drives quality in the sense that Quality goes up or down depending on the Structure.

Quality drives structure in the sense that structures are designed to meet quality goals.

Reusability 

Maintainability 

Performance 

Quality 

FUNDAMENTALS OF UML STRUCTURAL MODELING

- Quality is one of the most important topics for software designers.
  - ✓ Since structural designs affect quality, it is important that we have tools that can help us efficiently and effectively model structural aspects of software systems!
  - ✓ We will cover quality in more depth during the software architecture portion of the course and we will see these concepts throughout the rest of the course.

- In previous slides, we stated the assumption that *structure referred to the structure of code.*
  - ✓ It turns out that *structure* applies also to design units that exists at higher levels of abstractions than code. That is, units used to encapsulate functions, algorithms, classes, etc.
    - This structure is relevant to the software architecture and has significant impact on quality.
  - ✓ It also applies to other important aspects of software systems, e.g., the structure of the system as a whole (hardware, software, and interfaces)
    - This structure also drives quality!

- UML structural diagrams provide efficient tools that allows designers to create, evaluate, and analyze all of these structures.
  - ✓ They help us design for certain quality goals!

UML Component Diagrams

- A component represents a *modular* part of a system that *encapsulates its contents* and whose manifestation is *replaceable* within its environment.

- Component diagrams are used to model software as group of components connected to each other through well-defined interfaces. They help decompose systems and represent their structural architecture from a logical perspective.

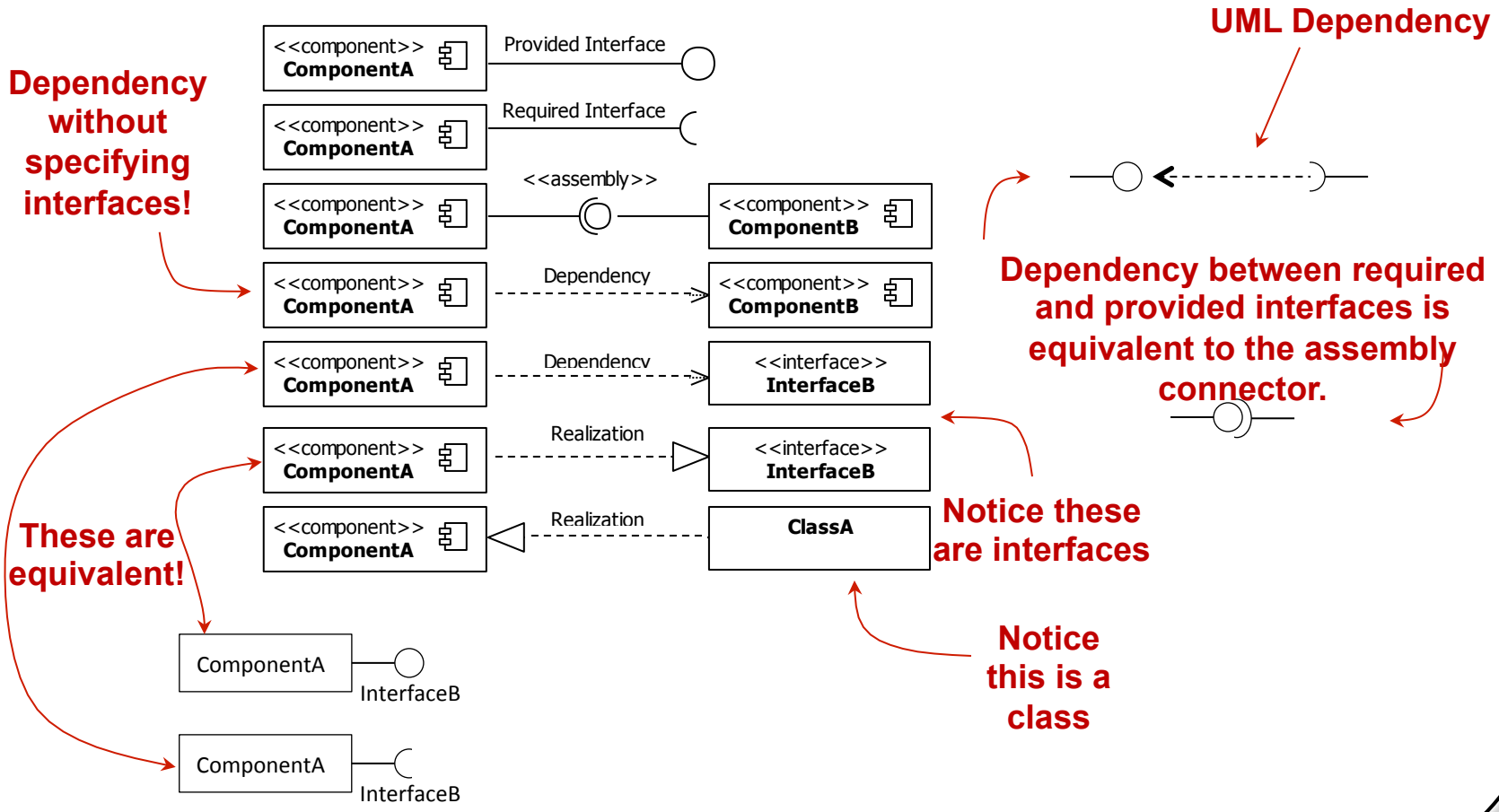
- Components can be modeled using an external black-box or internal white-box approach.
  - ✓ Black-box approach hides the component's internal structure.
    - Components interact with each other only through identified interfaces.
  - ✓ White-box approach shows the component's internal structure (e.g., realizing classifiers).

- Component interfaces are classified as *provided* or *required* interfaces.
  - ✓ Required interfaces are those the components need to carry out their functions.
  - ✓ Provided interfaces are used by other external components to interact with the component providing the services.

- Let's formally see how the UML relationships defined in previous sessions apply to the component classifier…

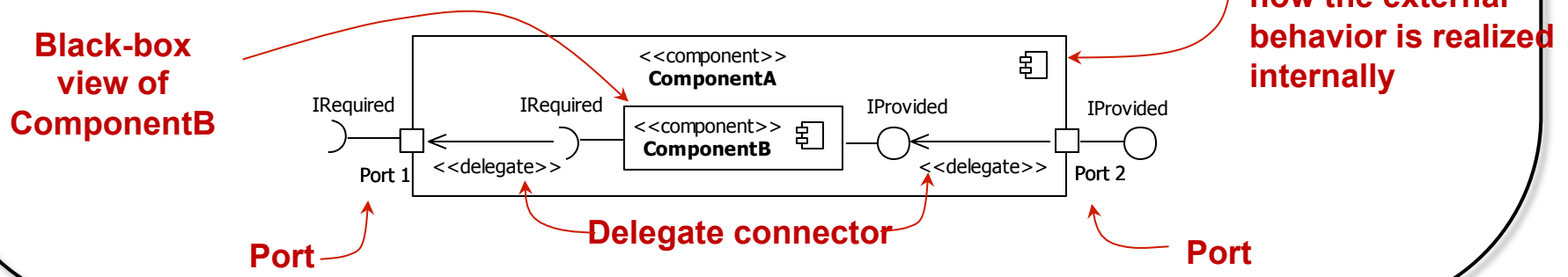
UML Component Diagram

- UML relationships applied to the component classifier.



UML Component Diagram

- Two more important concepts used in component diagrams are:
  - ✓ Ports
    - Abstraction used to model access points for allowing the external environment to access the component's services and for allowing components to interact with their external environment.
    - Modeled using a small square at the boundary of a *classifier*, in this case, a component.
    - Ports can be named, e.g., port names below are Port 1 and Port 2.
  - ✓ Delegation connectors
    - Used to model the link between the external provided interfaces of a component to the realization of those interfaces internally within the component.
    - Similarly, delegation connectors model the link between internally required interfaces to ports requiring the interface from external components.
    - Modeled using a directed arrow with the stereotype <<delegate>>



UML Component Diagram

- Consider a system with the following desired properties:
  - ✓ A data collection system equipped with:
    - Sensors
    - Video capture capabilities
  - ✓ Automatic collection at specific times of the day.
    - Collection schedules need to be provided to the system.
  - ✓ It is expected that the technology used for collection can improve, therefore:
    - Different sensors technology can be incorporated.
    - Different video capture capabilities can be incorporated.
    - This is important to the customers!
  - ✓ The system must make available the data collected.
    - Both sensor and video data.
    - Also, health data about the system
      - Events, problems, etc.

Warning:
There is not enough information to actually build this system.

Stop!

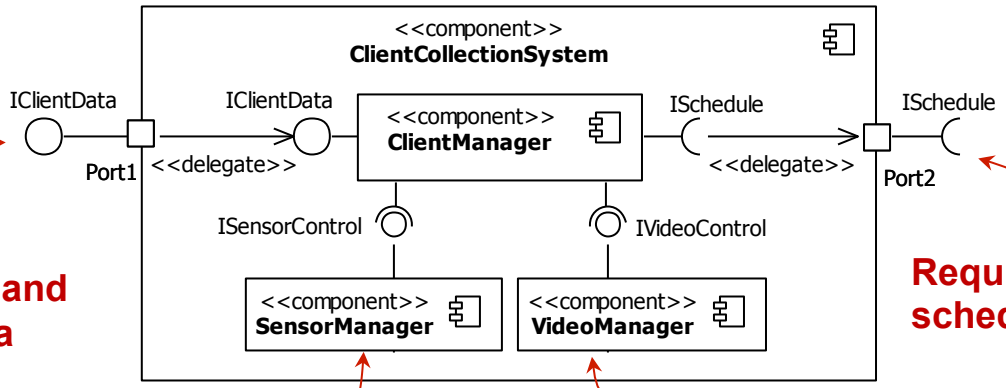
Before moving on, can you model this system using the component notation discussed so far? Give it a try using paper and pencil!

Remember this Definition?

Component = a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

Therefore, we really can't say at this point what the SensorManager will be! It could end up being one class, a bunch of classes, one function, etc.

The same applies to all other components!



Provides collection and health data

Requires collection schedule

Sensor functions

Video capture functions

Abstraction Principle:
Focuses on essential characteristics of entities—in their active context—while deferring unnecessary details

Everything that you see in the component diagram is an abstraction!

Although a good start, too many details are still missing to be able to build this system!

UML Component Diagrams

- A few last comments on UML components:
  - ✓ In previous versions of UML (i.e., 1.x), components were reserved exclusively for modeling deployable physical entities.
    - So, if you read an older book covering UML 1.x, discussions about components will differ from what we've presented so far.
  - ✓ A clear distinction between physical and logical components can be made by identifying the context in which they are relevant.
  - ✓ Both physical and logical components are modular parts of a system that encapsulate their contents and whose manifestation are replaceable within their *environment*.

A major difference is that physical components exist in a run-time environment , whereas logical components exist in a design-time environment!

This can be confusing because the design of a physical component may include one or more logical components!!!

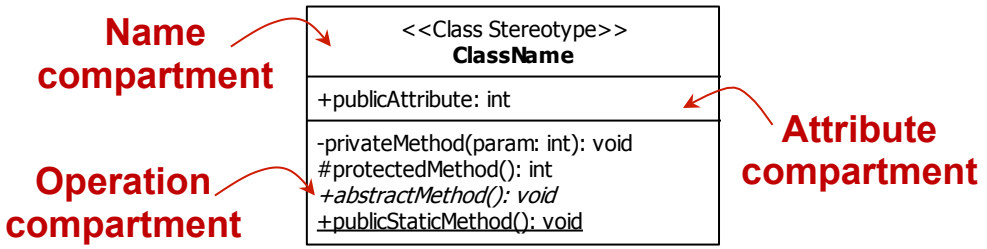
  - ✓ To eliminate confusion, UML 2.x supports the specification of these physical components as artifacts.
  - ✓ This new paradigm allows designers to model physical deployment aspects of components using artifacts classifier deployed on a node.

This is how UML 1.x components look like

UML Class Diagram

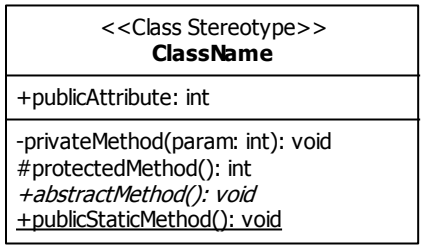
- Class diagrams exist at a lower level of abstraction than component diagrams.
  - ✓ Models consisting of classes and relationships between them necessary to achieve a system's functionality.
  - ✓ Weather a class diagram is created or not, the code of an object-oriented systems will always reflect some class design.
  - ✓ Therefore, there is two-way relationship between class diagrams and object-oriented code.
    - Class diagrams can be transformed to code (i.e., forward engineering)
    - Code can be transformed into class diagrams (i.e., reverse engineering)
  - ✓ This makes class diagrams the most powerful tool for designers to model the characteristics of object-oriented software before the construction phase.

- To become an effective designer, it is essential to understand the direct mapping between class diagrams and code. Let's take a closer look at the fundamental unit of the UML class diagram: the class.



UML Class Diagram

- Name compartment
  - ✓ Reserved for the class name and its stereotype
  - ✓ Class names can be qualified to show the package that they belong to in the form of Owner::ClassName.
  - ✓ Commonly used stereotypes include:
    - <<interface>>
      - Used to model interfaces.
    - <<utility>>
      - Used to model static classes.
- Attribute compartment
  - ✓ Reserved for the class' attribute specification.
    - Including name, type, visibility, etc.
- Operation compartment
  - ✓ Reserved for the class' operations specification.
    - Including name, return type, parameters, visibility, etc.
- Everything specified in the UML class can be directly translated to code… let's see an example in the next slide…



UML Class Diagram

- Example of the *forward engineering* of a UML class to C++ and Java.

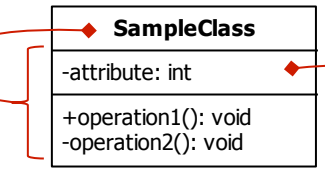
```
//  Generated by StarUML(tm) C++ Add-In
//
//  @ Project : Code Generation Tutorial
//  @ File Name : SampleClass.h
//  @ Date : 9/1/2012
//  @ Author : Carlos E. Otero
//
//


#if !defined(_SAMPLECLASS_H)
#define _SAMPLECLASS_H


class SampleClass {
public:
    void operation1();
private:
    int attribute;
    void operation2();
};

#endif  //_SAMPLECLASS_H
```

Code generated by free open source Star UML tool.



Class name

Private and public attributes and operations

Attribute name, type, and visibility

```
//  Generated by StarUML(tm) Java Add-In
//
//  @ Project : Code Generation Tutorial
//  @ File Name : SampleClass.java
//  @ Date : 9/1/2012
//  @ Author : Carlos E. Otero
//
//


public class SampleClass {
    private int attribute;
    public void operation1() {

    }

    private void operation2() {

    }
}
```

Important:
Notice how the modeled visibility {-, +} next to attribute and operations translate to code!

UML Class Diagram

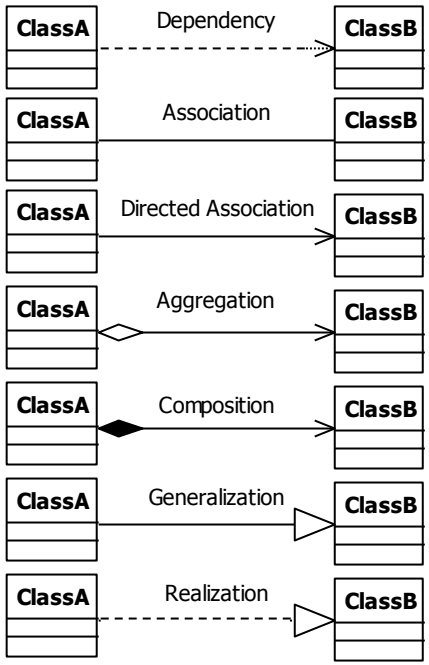
- In the previous slide, we presented two different types of UML visibility specification.
  - ✓ Visibility types specify policies on how attributes and operations are accessed by clients.
  - ✓ Common types of visibility are presented below.

Visibility	Symbol	Description
Public	+	Allows access to external clients.
Private	-	Prevents access to external clients. Accessible only internally within the class.
Protected	#	Allows access internally within the class and to derived classes.
Package	~	Allows access to entities within the same package.

Important:
Visibility allows us to apply the Encapsulation principle in our designs!

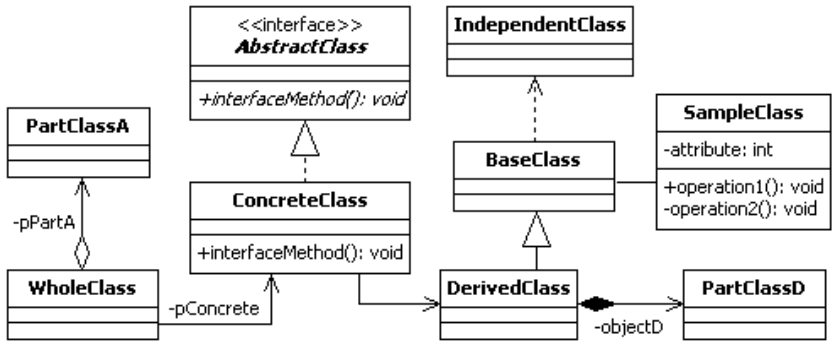
UML Class Diagram

- UML relationships applied to the class classifier



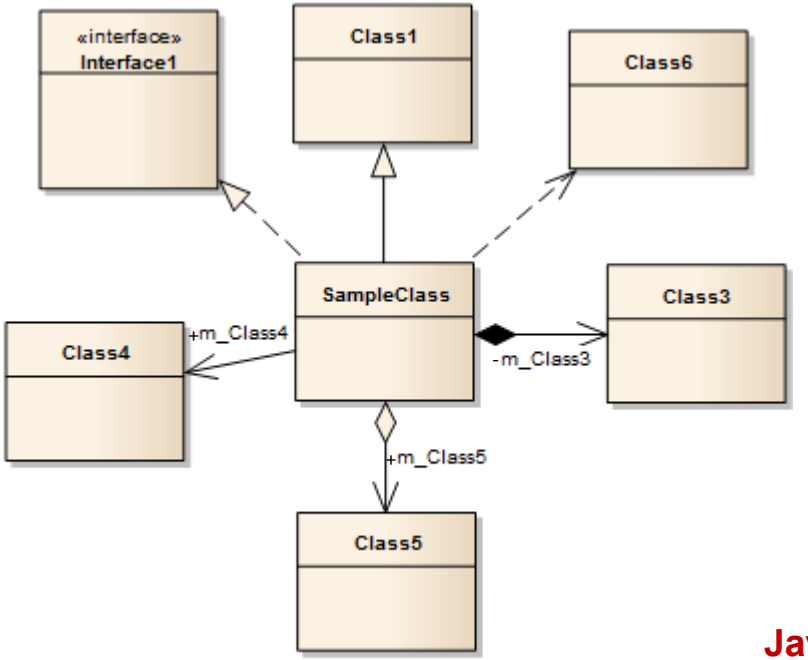
Important:
All of these relationships mean something in code, so that when you define these relationships, you're actually beginning to structure your code!

This is how a sample class diagram would look like



UML Class Diagram – Code Generation

Model and Code generated by commercial Enterprise Architect UML tool.



Notice that dependency on Class6 is not generated!

```
#include "Class1.h"
#include "Class3.h"
#include "Interface1.h"
#include "Class4.h"
#include "Class5.h"

class SampleClass : public Class1, public Interface1
{

public:
    SampleClass();
    virtual ~SampleClass();
    Class4 *m_Class4;
    Class5 *m_Class5;

private:
    Class3 m_Class3;

};
```

C++ code generation of model

Java code generation of same model

```
public class SampleClass extends Class1 implements Interface1 {
    private Class3 m_Class3;
    public Class4 m_Class4;
    public Class5 m_Class5;
}
```

<u>Important</u>:
Code generation varies from tool-to-tool. Some need to be configured appropriately to be useful in production environments!

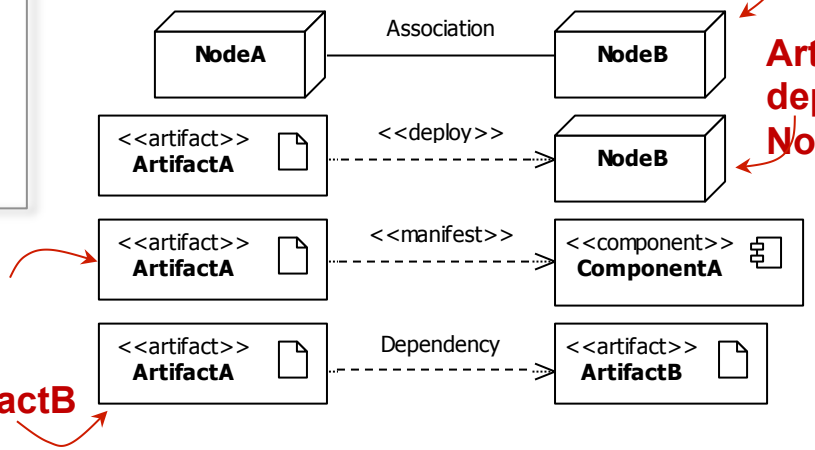
UML Deployment Diagrams

- Deployment diagrams are structural diagrams used to model the physical realization of software systems.
  - ✓ They provide the means to visualize and evaluate the environment in which software executes.
  - ✓ They model nodes and the interfaces between them.
    - A *Node* is a computational resource that host software artifacts for execution.
    - As seen before, in UML, a *Node* is a named classifier modeled as a cube.

- Deployment diagrams also include artifact and components and depicts how all of these work together from a system deployment perspective.

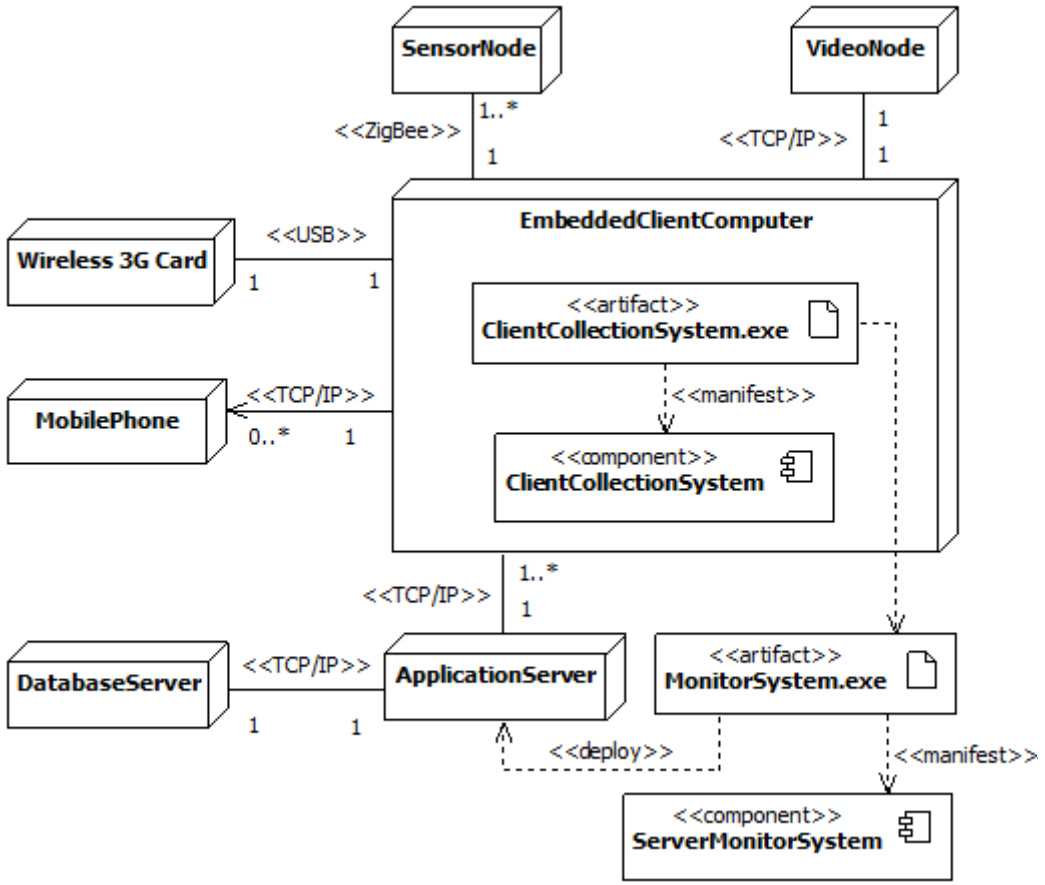
- UML relationships applied to the these classifier is presented below.

Important:
A UML Artifact is a classifier used to model physical units of information that form part of the software system, such as binary executable files, configuration files, scripts, .jar files, .dll, etc.



UML Deployment Diagrams

- Example of UML Deployment Diagram.



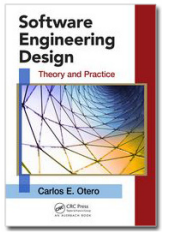
WHAT'S NEXT…

- In the next session we will discuss how UML classifiers, relationships, and enhancement features can be used to create behavioral diagrams. Specifically, we will focus on:
  - ✓ UML behavioral modeling
    - What is behavioral modeling?
    - Why is it important?
  - ✓ Use case diagrams
    - Actors
    - System boundary
    - Common relationships
  - ✓ Interaction diagrams
    - Communication diagrams
    - Sequence diagrams
    - Concurrency modeling
  - ✓ Summary and conclusion of UML coverage

CHAPTER 2: SOFTWARE DESIGN WITH THE UNIFIED MODELING LANGUAGE

SESSION III: UML BEHAVIORAL MODELING

Software Engineering Design: Theory and Practice
by Carlos E. Otero



SESSION'S AGENDA

- UML behavioral modeling
  - ✓ What is behavioral modeling?
  - ✓ Why is it important?
- Use case diagrams
  - ✓ Actors
  - ✓ System boundary
  - ✓ Common relationships
- Interaction diagrams
  - ✓ Communication diagrams
  - ✓ Sequence diagrams
  - ✓ Concurrency modeling
- Summary and conclusion of UML coverage

UML Behavioral Modeling

- In the previous session, we presented structural modeling and made a case for how it is essential to evaluate, characterize, and visualize the structural design of software systems from various perspective. Specifically, we presented:
  - ✓ Logical structural designs
    - At different levels of abstraction
  - ✓ Physical structural designs

- We also presented (very vaguely) the concept of **Quality** and discussed how structural modeling can be used to evaluate and access quality in terms of some quality goals, such as *reusability* and *maintainability*.

- Although structural designs work well for evaluating some quality attributes of systems, they are inadequate for others, such as *performance*.
  - ✓ Structural designs also provide poor techniques for evaluating the dynamic aspects and interactions of systems.

- UML provides several diagrams to model and reason about the dynamic aspects and of systems.
  - ✓ These can be used to model almost any behavioral aspect of modern software systems
  - ✓ They are used in many practical development projects.

UML Behavioral Modeling

- From previous sessions, we learned that the most common UML *behavioral diagrams* include:

Behavioral Diagram	Description
Use Case Diagram	Used to capture, specify, and visualize required system behavior .
Sequence Diagram	Used to capture, specify, and visualize system interactions with emphasis on the time-order sequence of messages exchanged.
Communication Diagram	Used to capture, specify, and visualize system interactions with emphasis on the structural order of entities participating in the message exchange.
State Machine Diagram	Used to capture, specify, and visualize system behavior as a set of discrete states and the transitions between them.
Activity Diagram	Used to capture, specify, and visualize system behavior; provide mechanisms for modeling that includes conditional statements, repetition, concurrency, and parallel execution and thus can be used at many different levels of abstraction, from modeling business work flows to code.

- Other behavioral diagrams include:
  - ✓ Timing diagram
  - ✓ Interaction overview diagram

UML Behavioral Modeling – Use Case Diagram

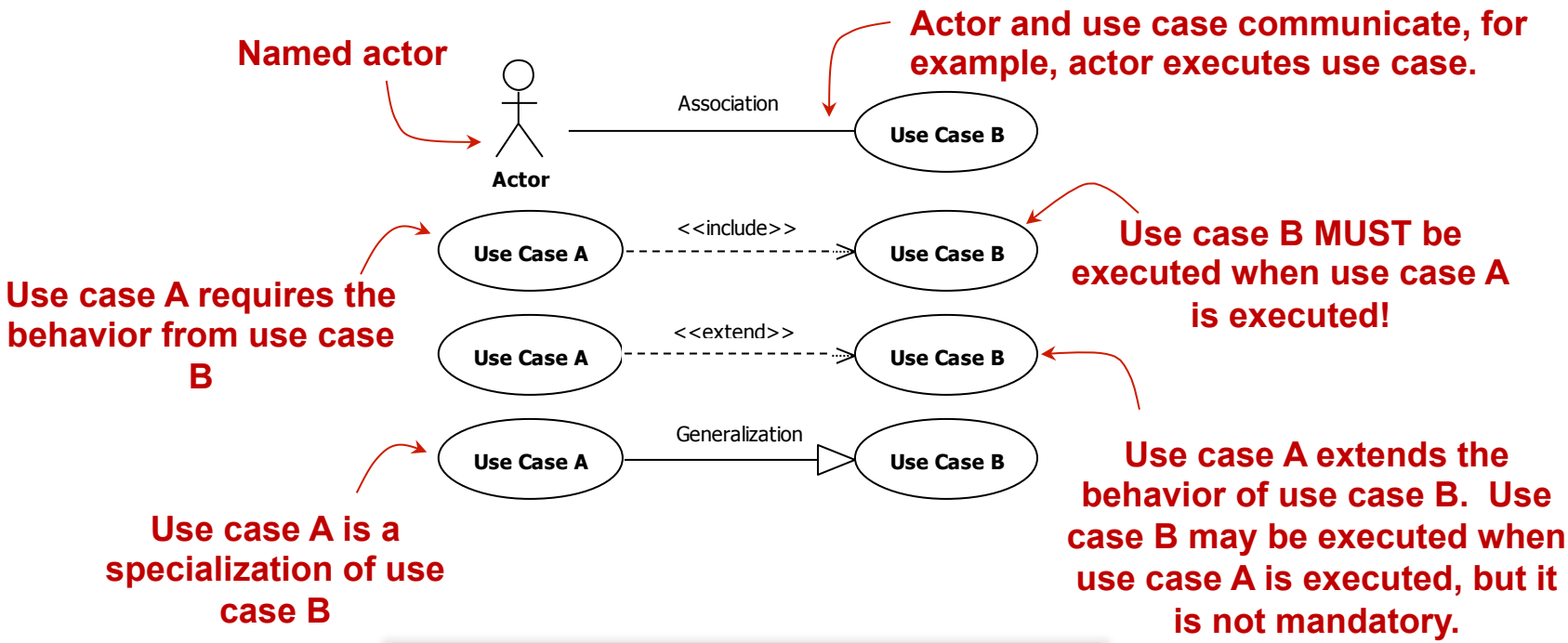
- Use case diagram
  - ✓ Behavioral diagram used to capture, specify, and visualize required system behavior.
    - Required system behavior are just requirements!
  - ✓ The main elements of use case diagrams are *actors*, *use cases*, and the *relationships* connecting them together.

- Actors are entities used to model users or other systems that interact with the system being modeled (i.e., the subject). Examples include:
  - ✓ Operators
  - ✓ Sensors
  - ✓ Client computers

- Use cases are entities used in use case diagrams to specify the required behavior of a system.
  - ✓ They provide the means to capture, model, and visualize the systems' required behavior.
  - ✓ They do this without any knowledge of programming technology, so that different stakeholders with different backgrounds can reason about the system.

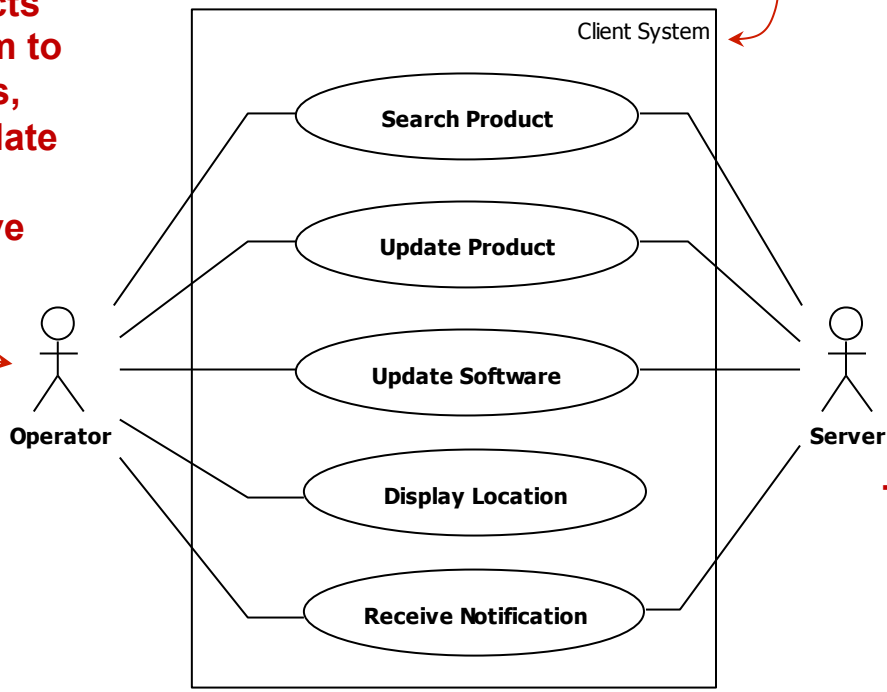
UML Behavioral Modeling – Use Case Diagram

- Common UML relationships applied in use case diagrams.



Important:
include relationship specifies a mandatory inclusion
Extend relationship specifies an optional inclusion

UML Behavioral Modeling – Use Case Diagram



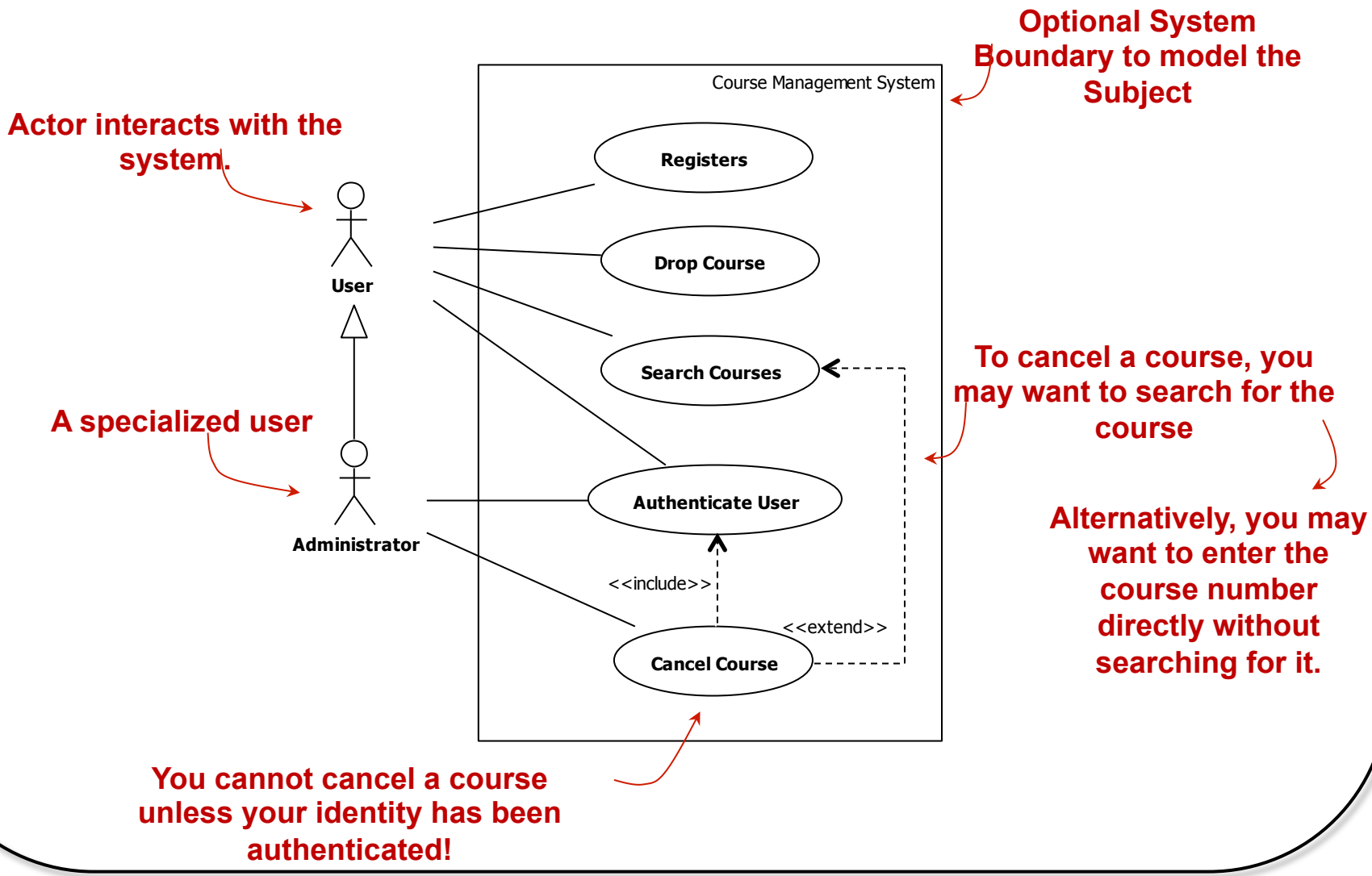
Optional System Boundary to model the Subject

The Operator interacts with the Client System to search for products, update products, update software, display location, and receive notifications.

The Server also interacts with the Client System

This models the behavior of a client system. An Operator and Server both interact with the Client System, denoted by the System Boundary.

UML Behavioral Modeling – Use Case Diagram



UML Behavioral Modeling – Interaction Diagrams

- Interaction diagrams can be used to model complex interactions between design units together with the messages exchanges and the type of the exchange.
- Interaction diagrams can be used at the architectural level.
  - ✓ For example, they allow us to model interaction among software components.
- Interaction diagrams can be used at the detailed design level.
  - ✓ For example, they allow us to model interactions among objects at run-time
- Interaction diagrams can be used at the construction design level.
  - ✓ For example, they allow us to model conditional and repetition structures.
- In many situations, interaction diagrams reveal many important issues related to the quality of the system.
- Two types of interaction diagrams are:
  - ✓ Communication diagrams
  - ✓ **Sequence diagrams**

UML Behavioral Modeling – Communication Diagram

- Communication diagrams
  - ✓ Behavioral diagrams used to capture, specify, evaluate, and visualize system interactions with emphasis on the structural order of entities participating in the message exchange.

- When using communication diagrams, entities can be modeled as
  - ✓ Objects, representing instances of classes and components.
  - ✓ Roles, representing a prototypical instance

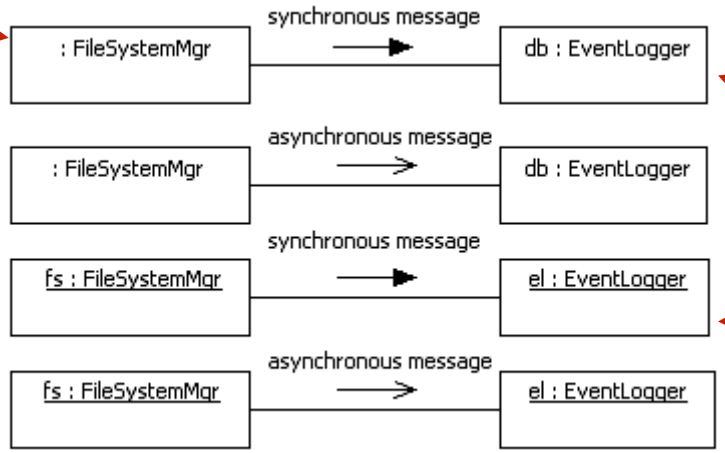
- Both objects and roles can be connected to model the exchange of messages using:
  - ✓ Links to connect objects
  - ✓ Connectors to connect roles
  - ✓ Both links and connectors look exactly alike, as a solid line. They only differ semantically.

- The type of message exchanges include:
  - ✓ Asynchronous
  - ✓ Synchronous

UML Behavioral Modeling – Communication Diagram

FileSystemMgr Role.

In synchronous messaging, sender and receiver work in lock-step to achieve an operation.



In asynchronous messaging, sender continues execution while receiver completes operation independent from the sender

EventLogger Object.

Asynchronous messaging between two run-time object instances

Important:
Synchronous and Asynchronous messaging are are important when reasoning about concurrency issues!

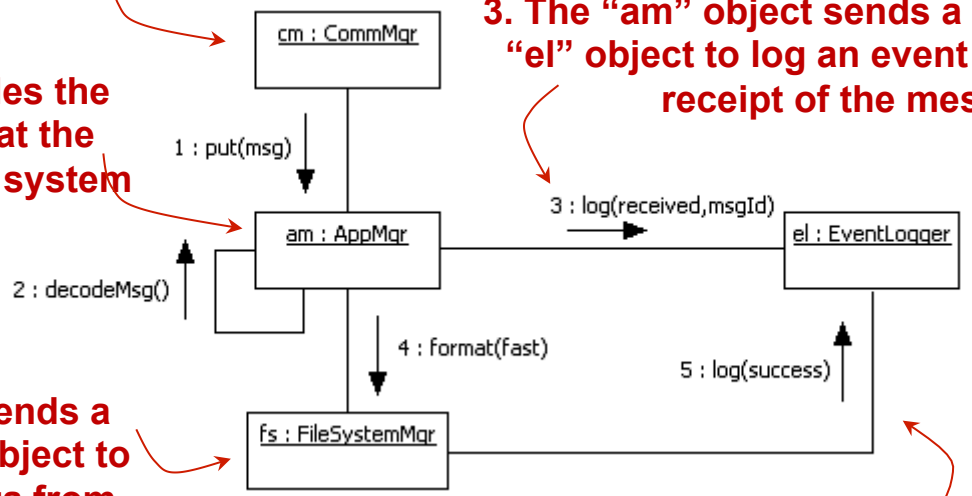
UML Behavioral Modeling – Communication Diagram

Communication diagram of objects at run-time using synchronous messaging

1. The "cm" object of type CommMgr sends a message to another object of type AppMgr.

2. The "am" object decodes the message. If finds out that the message commands a file system format operation

3. The "am" object sends a message to the "el" object to log an event indicating the receipt of the message



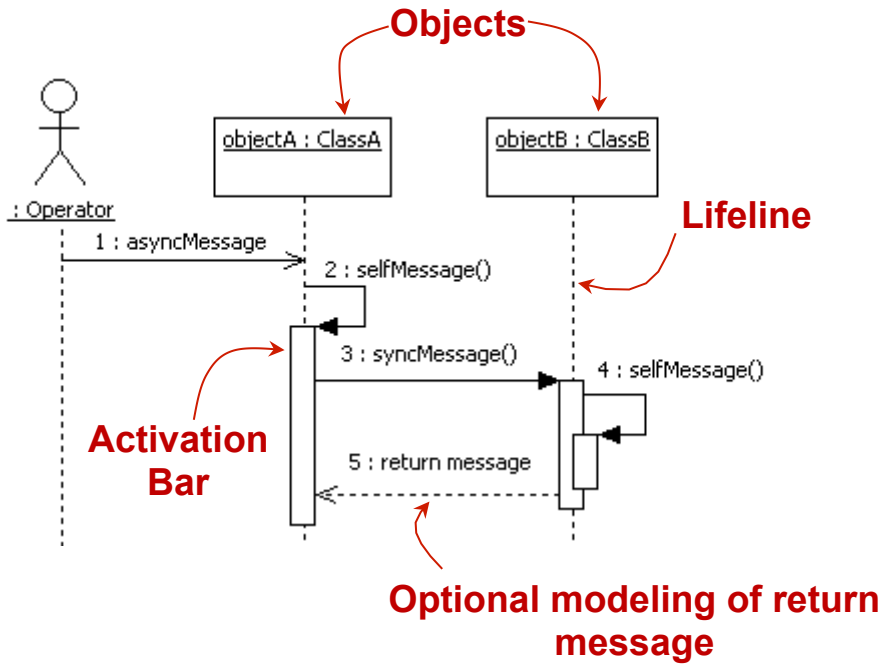
4. The "am" object sends a message to the "fs" object to begin erasing the data from the file system

5. Once complete, the "fs" object sends a message to the "el" object to log an event indicating success

Given this model, can "cm" begin other operations before the file system format is complete?

UML Behavioral Modeling – Sequence Diagram

- Sequence diagrams
  - Similar to Communication diagrams, but, they put emphasis on the time-order sequence of messages exchanged.

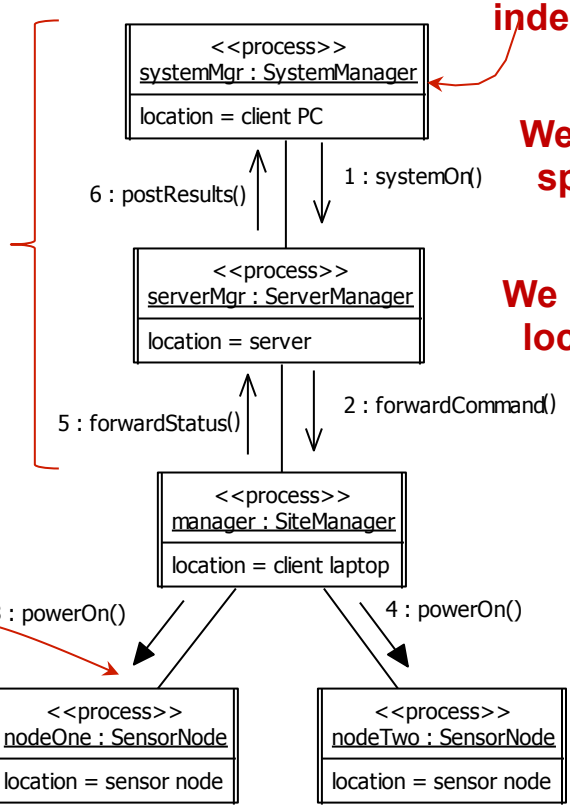


UML Behavioral Modeling

- Finally, a word about concurrency…

SystemManager process, located on client PC does not have to wait until Sensor Nodes are powered on to begin doing other work

Asynchronous messaging



We use Active Objects to model independent flows of control, e.g., processes.

We use stereotype <<process>> to specify these are actual run-time processes.

We use Tagged Values to specify the location of the running processes.

Synchronous messaging

SiteManager works in lock-step with nodeOne. Once nodeOn is powered on, then SiteManager initiates power on sequence of nodeTwo

WHAT'S NEXT…

- Now that we are equipped with the necessary UML tools, we can now explore how to used them to design a software architecture.

- In the next session, we will present Software Architecture in more detail, including :
  - ✓ Understanding the role of software architecture within the design phase.
  - ✓ Explore in more detail the architectural tasks and problem-solving during architecture.