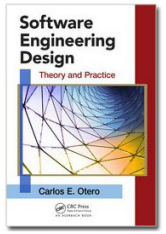
# CHAPTER 2: SOFTWARE DESIGN WITH THE UNIFIED MODELING LANGUAGE

## SESSION II: UML STRUCTURAL MODELING

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



# SESSION'S AGENDA

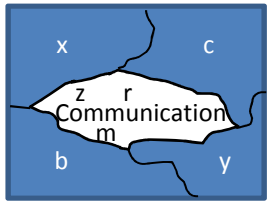
- UML structural modeling
  - ✓ What is structural modeling?
  - ✓ Why is it important?
- Component diagrams
  - ✓ Interfaces
  - ✓ Assembly connectors
  - ✓ Other common relationships
- Class diagrams
  - ✓ Details of UML classes
  - ✓ Common relationships
  - ✓ The meaning (in code) of class diagrams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

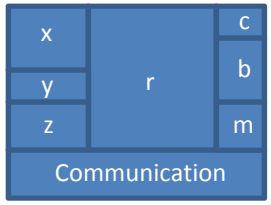
Remember modularization
From previous sessions?

- 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?

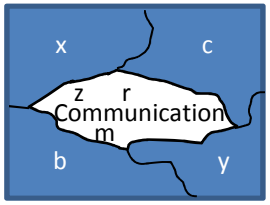
- We have not talk about the concept of **Quality** yet, but we will… For now, assume that quality is a function of reusability and maintainability. What can you say about the quality of these systems?
  - ✓ Under these assumptions structure drives quality AND quality drives structure.

How can this be true?

# FUNDAMENTALS OF UML STRUCTURAL MODELING

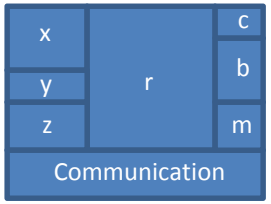
- In the previous slide, under the established assumptions, we mentioned that structure drives quality AND quality drives structure. What do we mean by that? Consider again the two structures for the same software application.
  - ✓ Assume that these are structures for a message processing application, 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!

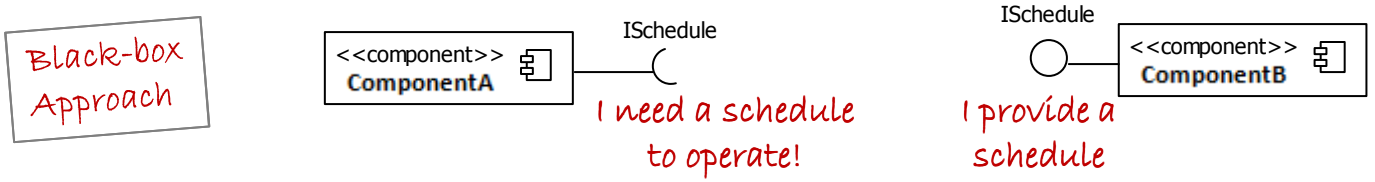
# UML Component Diagrams

- 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).

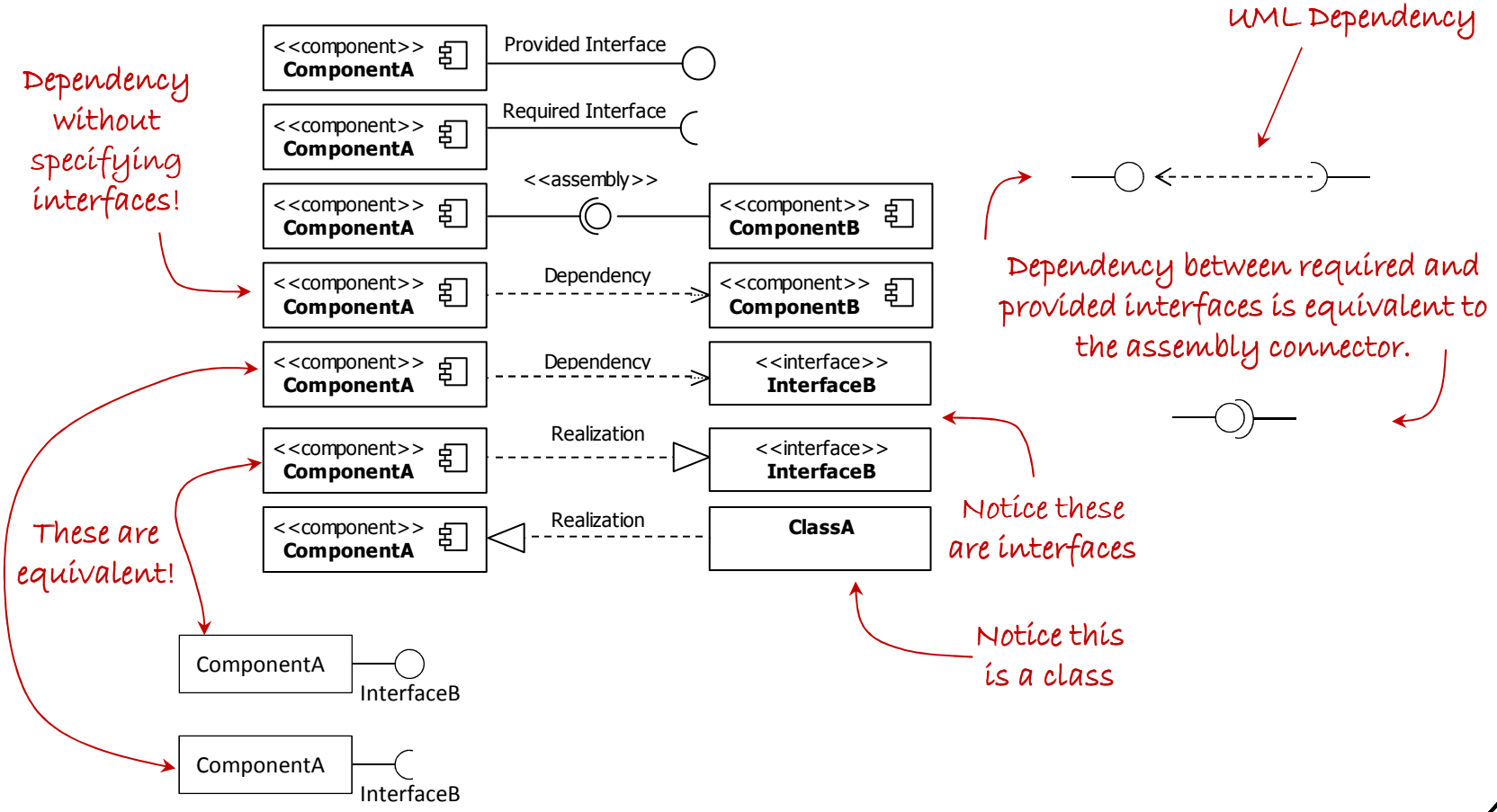
- Component interfaces are classified as *provided* or *required* interfaces.
  - ✓ Required interfaces are those the components need to carry out their functions.
  - ✓ Provided interfaces are used by other external components to interact with the component providing the services.




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

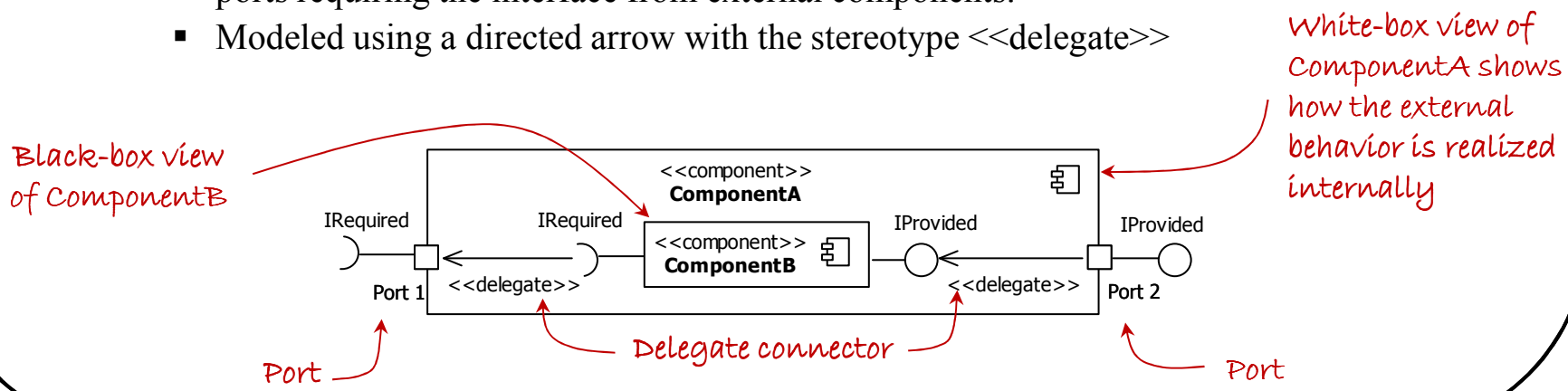
# UML Component Diagram

- 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!

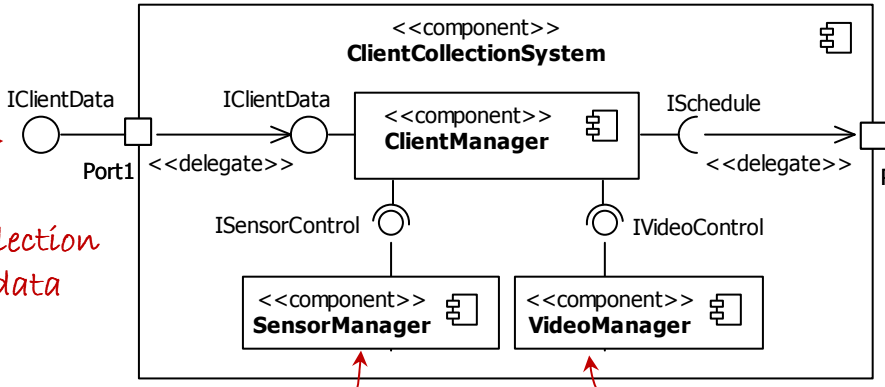
Before moving on, can you model this system using the component notation discussed so far? Give it a try using paper and pencil!

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!!!

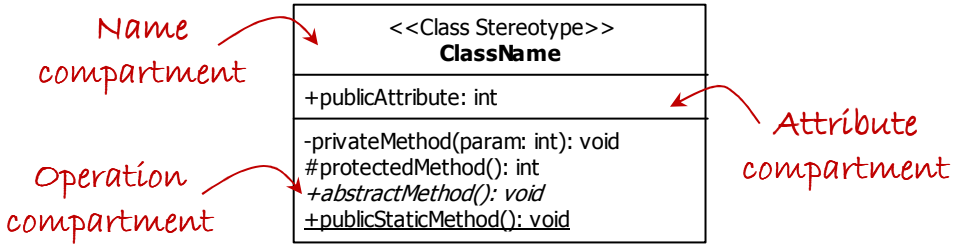
  - ✓ To eliminate confusion, UML 2.x supports the specification of these physical components as artifacts.
  - ✓ This new paradigm allows designers to model physical deployment aspects of components using artifacts classifier deployed on a node.


This is how UML 1.x components look like

# UML Class Diagram

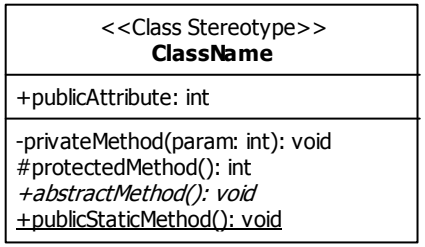
- Class diagrams exist at a lower level of abstraction than component diagrams.
  - ✓ Models consisting of classes and relationships between them necessary to achieve a system's functionality.
  - ✓ Weather a class diagram is created or not, the code of an object-oriented systems will always reflect 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.

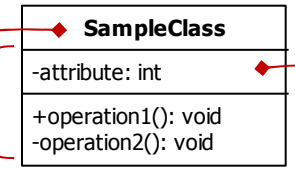
```
//  Generated by StarUML(tm) C++ Add-In
//
//  @ Project : Code Generation Tutorial
//  @ File Name : SampleClass.h
//  @ Date : 9/1/2012
//  @ Author : Carlos E. Otero
//
//


#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

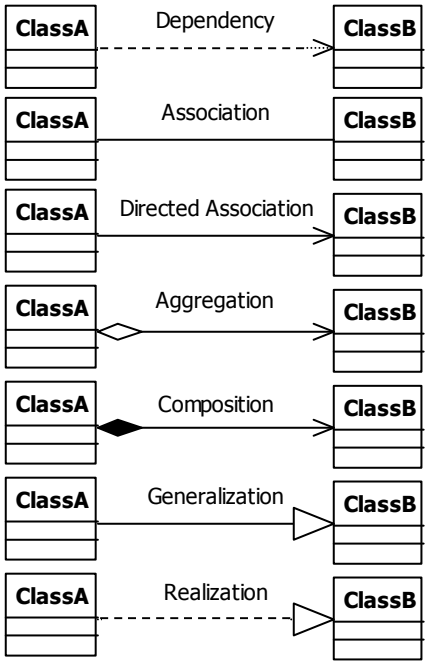
- In the previous slide, we presented two different types of UML visibility specification.
  - ✓ Visibility types specify policies on how attributes and operations are accessed by clients.
  - ✓ Common types of visibility are presented below.

| 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!

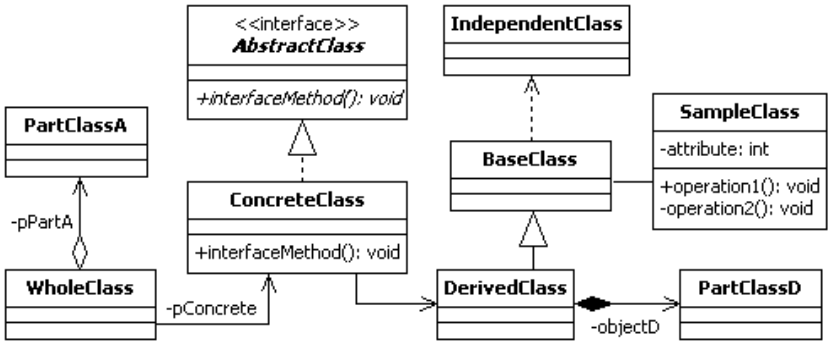
# UML Class Diagram

- UML relationships applied to the class classifier



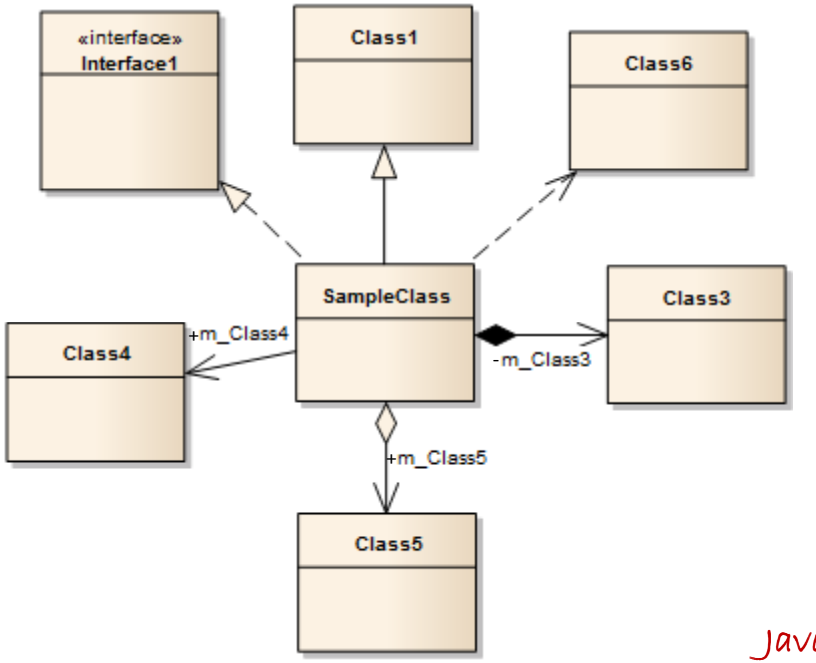

Important:
All of these relationships mean something in code, so that when you define these relationships, you're actually beginning to structure your code!

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;
}
```

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

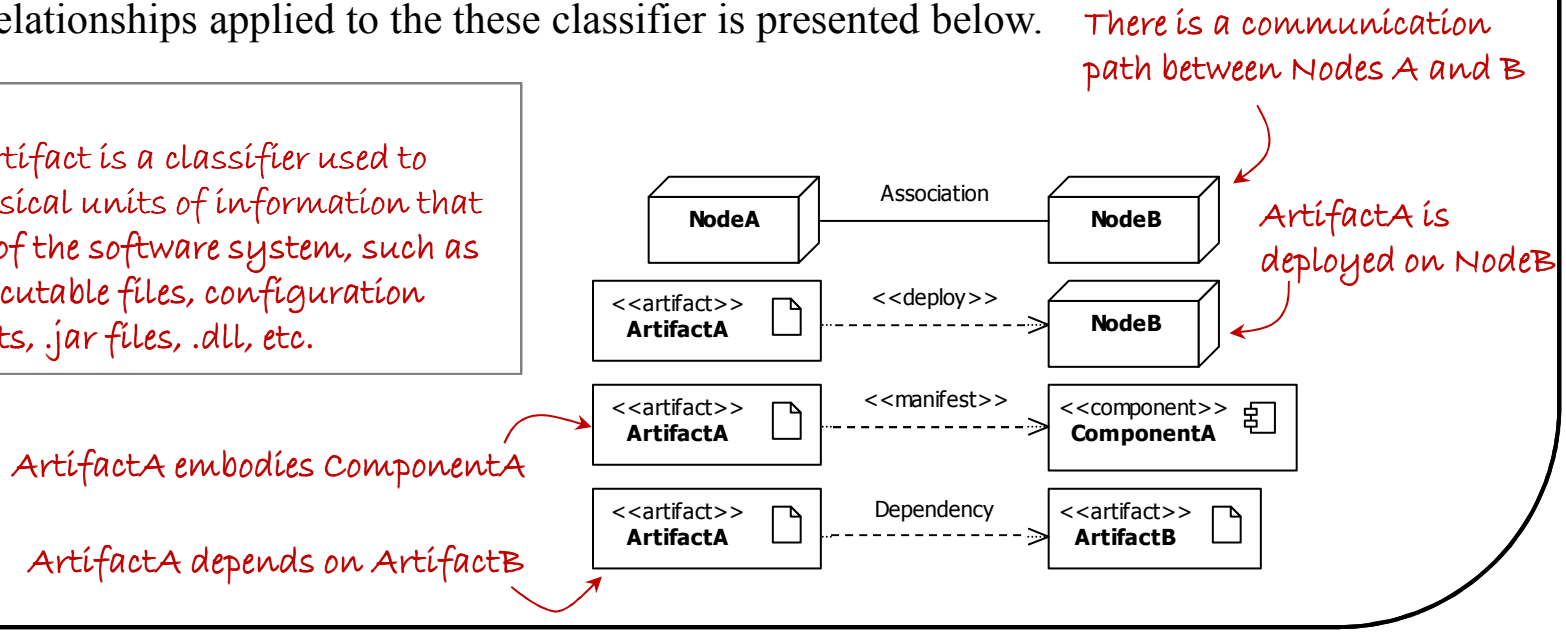
# UML Deployment Diagrams

- 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.

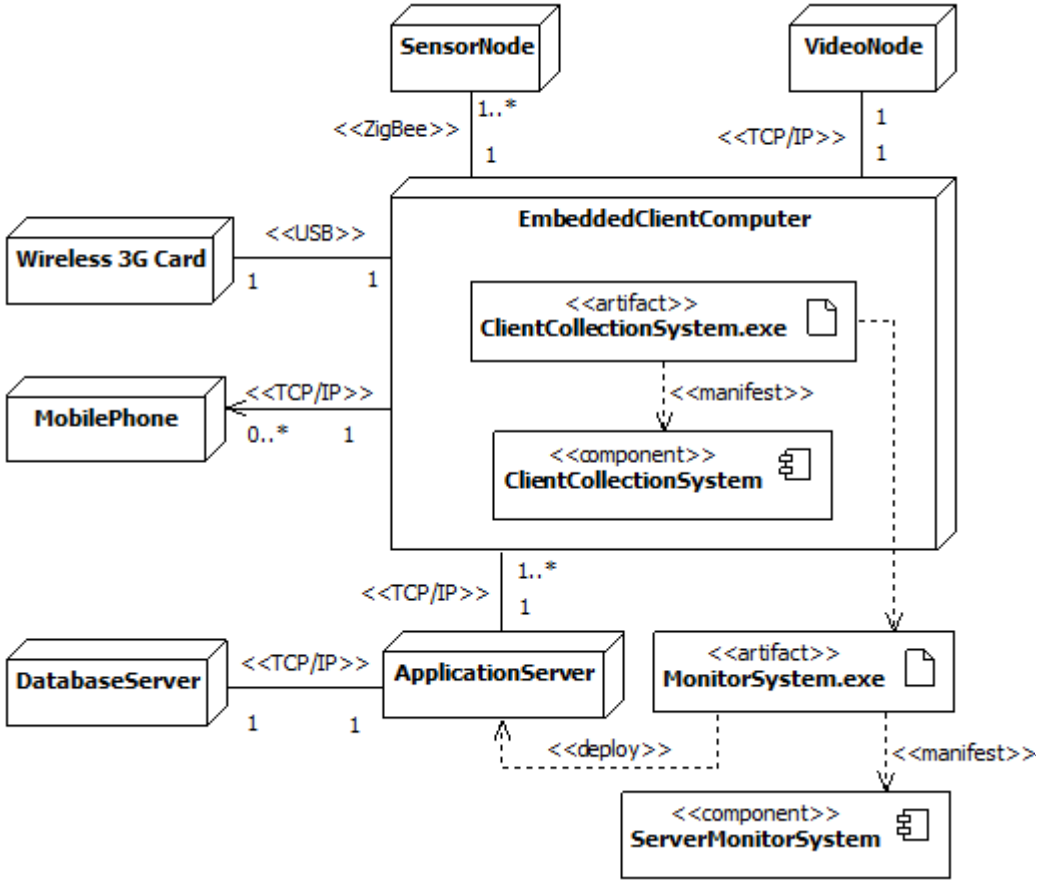
- UML relationships applied to the these classifier is presented below.

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