CHAPTER 5:
PRINCIPLES OF DETAILED DESIGN

1. Detailed Design Fundamentals
2. Structural and Behavioral Design of Components

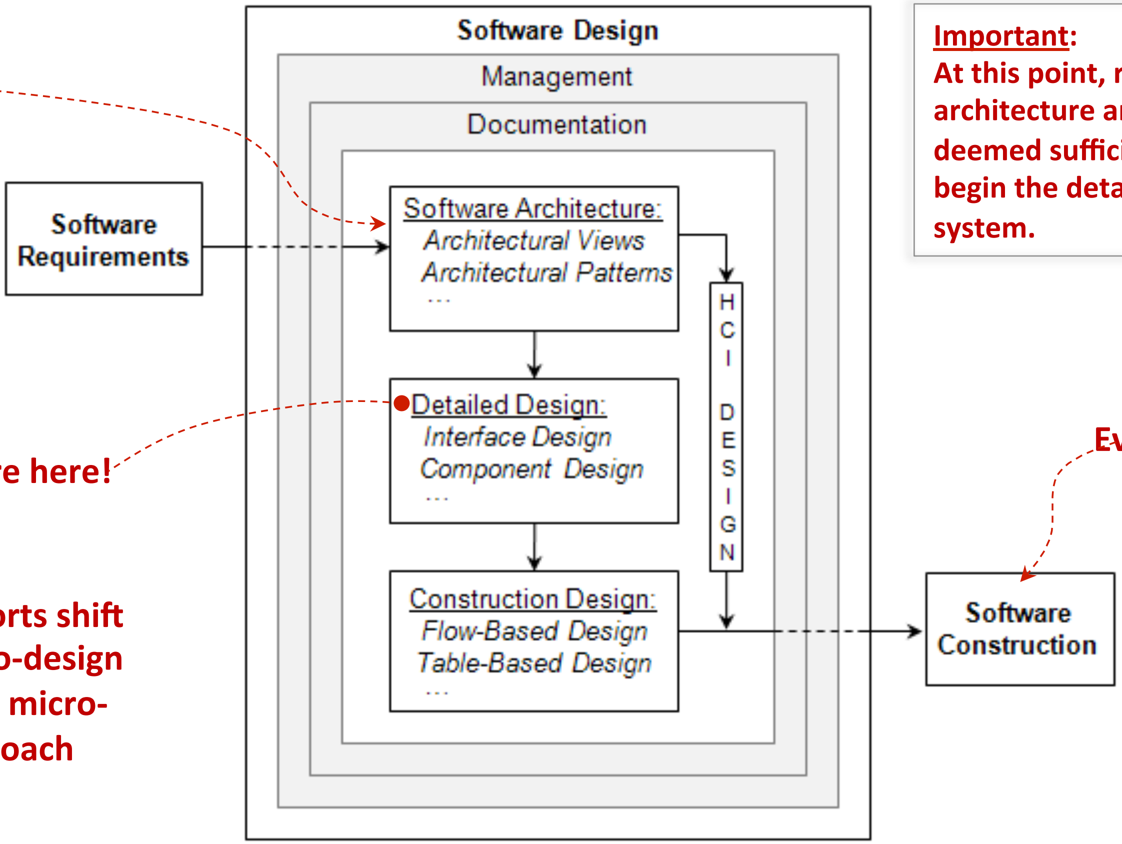
Session I: Detailed Design Fundamentals

SESSION'S AGENDA

1. **Overview of Detailed Design**
    a. **What is detailed design**?
    b. Where does it **fit**?

2. **Key Tasks in Detailed Design**
    a. Understanding architecture and requirements – Topic 3 & 4
    b. Creating detailed designs
    c. Evaluating detailed designs
    d. Documenting detailed designs
    e. Monitoring and controlling implementation

FIRST, LET'S THINK ABOUT WHERE WE ARE…



We started here

We are here!

Eventually, we want to get here!

Important:
At this point, requirements and architecture are specified; they are deemed sufficiently complete to begin the detailed design of the system.

Our design efforts shift from the macro-design approach to a micro-design approach

We now seek to further decompose and refine system components into one or more fine-grained elements, functions, and data variables.

WHAT IS DETAILED DESIGN?

- According to the IEEE [1],
  1. The process of refining and expanding the **preliminary design phase** of a system or component to the extent that the design is sufficiently complete to be implemented .
  2. The result of the process in 1.
- To keep terminology consistent, we'll use the following definition:
  1. The **process of refining and expanding** the ***software architecture*** of a **system or component** to the extent that the **design is sufficiently complete to be implemented .**
  2. The result of the process in 1.
- During *Detailed Design* designers **go deep** into each **component** to define its **internal structure and behavioral capabilities**, and the resulting design leads to natural and efficient construction of software.

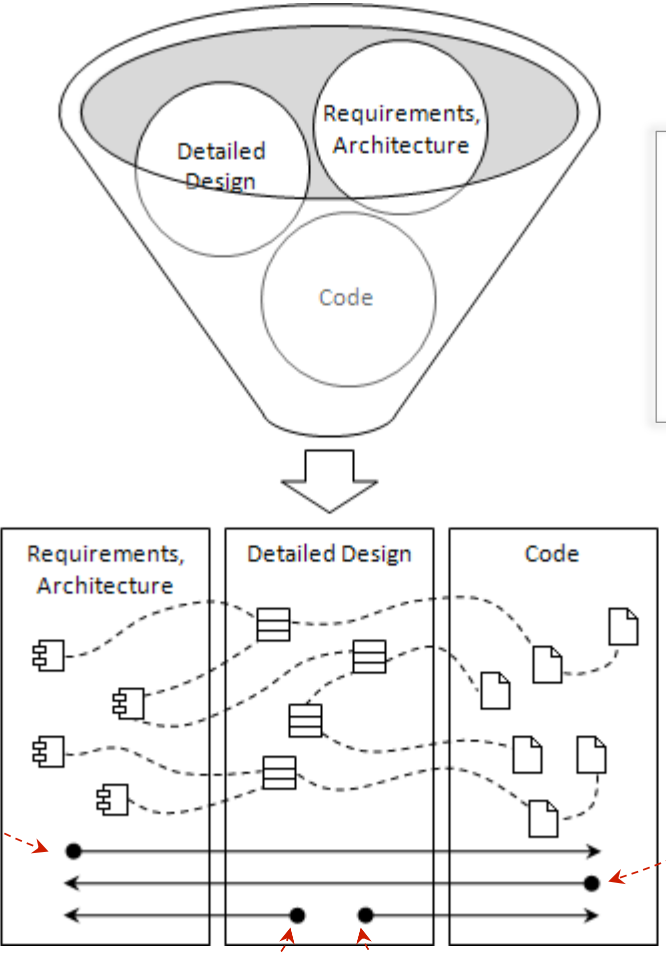
WHAT IS DETAILED DESIGN?

- Clements et al. [2] differentiate between architectural and detailed design as follows:
  - ✓ *"Architecture is design, but not all design is architecture. That is, many design decisions are left unbound by the architecture and are happily left to the discretion and good judgment of downstream designers and implementers. The architecture establishes constraints on downstream activities, and those activities must produce artifacts—finer-grained design and code—that are compliant with the architecture, but architecture does not define an implementation."*

- Detailed design is closely related to architecture and construction; therefore successful designers (during detailed design) are required to have or acquire full understanding of the system's requirements and architecture.
  - ✓ They must also be proficient in **particular design strategies** (e.g., **object-oriented**), **programming languages**, and **methods and processes for software quality control.**
  - ✓ Just as architecture provides the bridge between requirements and design, **detailed design provides the bridge between design and code.**

WHAT IS DETAILED DESIGN?

Designer's Mental Model During Detailed Design!



Important:
During detailed design, the use of industry-grade development tools are essential for modeling, code generation, compiling generated code, reverse engineering, software configuration management, etc.

If given requirements and architecture, detailed designers must move the project forward all the way to code

If given code, detailed designers must be able to reverse engineer the code to produce detailed and architectural designs.

When starting at detailed design, designers must be able to produce both code and architectural designs

KEY TASKS IN DETAILED DESIGN

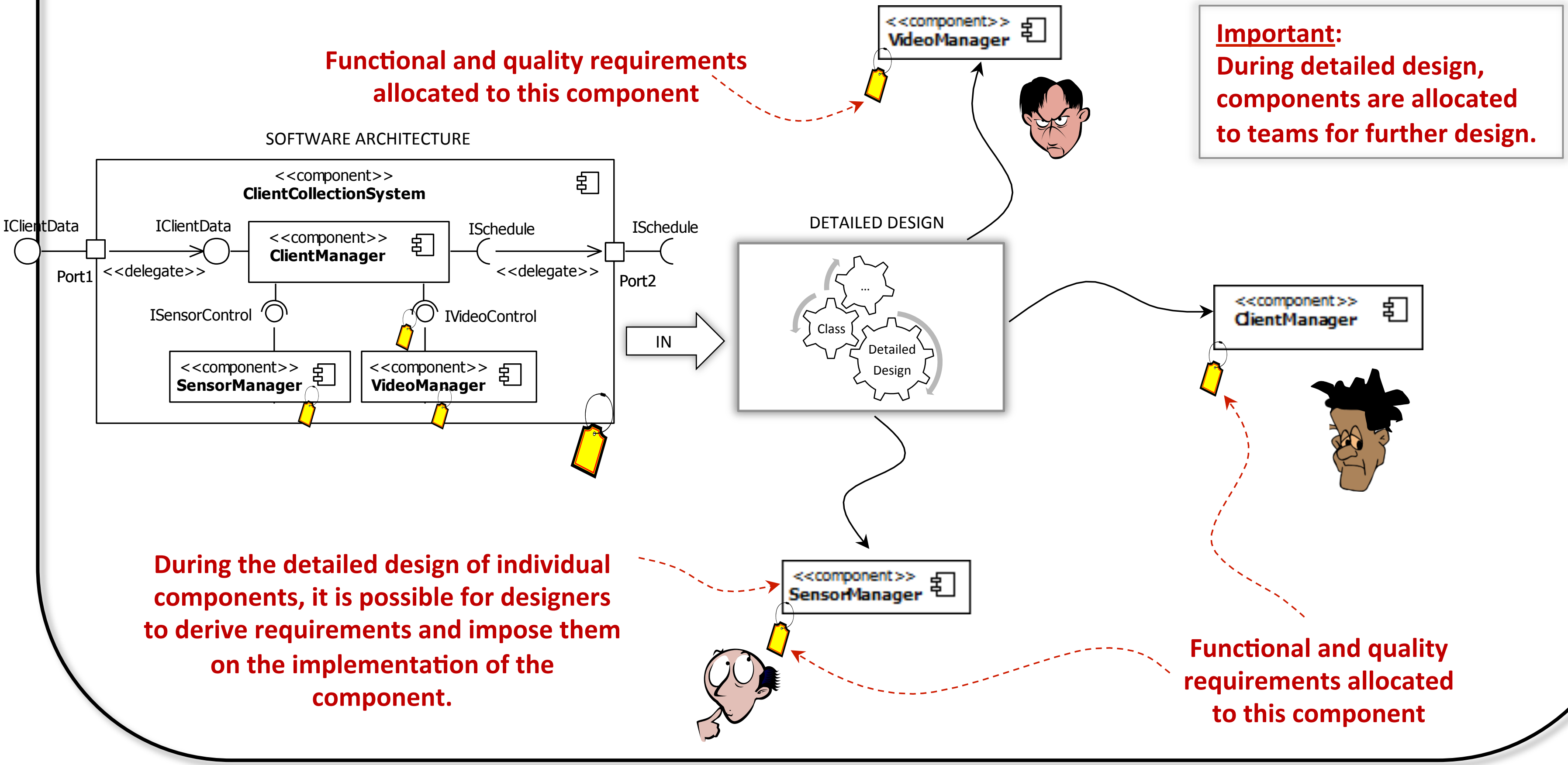
- In practice, it can be argued that the detailed design phase is where most of the problem-solving activities occur. Consider the case in which a formal process is followed, so that the requirements is followed by architecture and detailed design.
  - ✓ In many practical applications, the architectural design activity defers complex problem solving to detailed design, mainly through abstraction.
  - ✓ In some cases, even specifying requirements is deferred to detailed design!

- For these reasons, detailed design serves as the gatekeeper for ensuring that the system's specification and design are sufficiently complete before construction begins.
  - ✓ This can be especially tough for large-scale systems built from scratch without experience with the development of similar systems.

- The major tasks identified for carrying out the detailed design activity include:
  1. Understanding the architecture and requirements
  2. **Creating detailed designs**
  3. Evaluating detailed designs
  4. Documenting software design
  5. Monitoring and controlling implementation

1. UNDERSTANDING THE ARCHITECTURE AND REQUIREMENTS

- Unlike the **software architecture**, where the **complete set of requirements** are **evaluated** and **well understood**, designers during **detailed design** activity focus on **requirements allocated to their specific components**.



Functional and quality requirements allocated to this component

Important:
During detailed design, components are allocated to teams for further design.

During the detailed design of individual components, it is possible for designers to derive requirements and impose them on the implementation of the component.

Functional and quality requirements allocated to this component

2. CREATING DETAILED DESIGNS

- After the architecture and requirements for assigned components are well understood, the detailed design of software components can begin.
  - ✓ Detailed design consist of both **structural and behavioral designs**.

- When creating detailed designs, focus is placed on the following:
  1. **Interface Design** - Internal & External
  2. **Graphical User Interface** (GUI) Design *(Chapter 9)*
     - This may be a continuation of designs originated during architecture.
  3. **Internal Component** Design *(Chapter 7)*
     - Structural
     - Behavioral
  4. **Data Design** ~ *Database ; data dictionary*

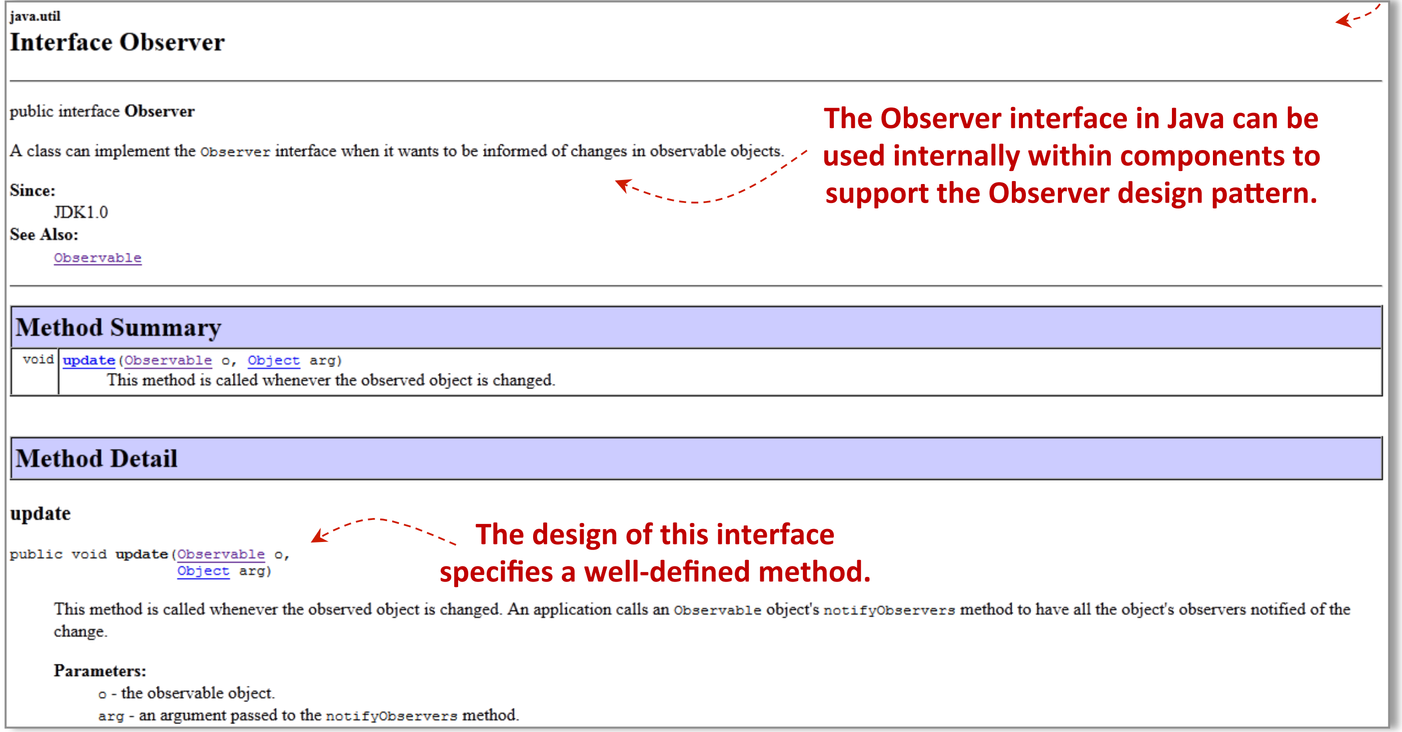
2. CREATING DETAILED DESIGNS

1. Interface Design

- ✓ Refers to the design task that deals **with specification of interfaces between components in the design** [3]. It can be focused on:
  - Interfaces **internally** within components
  - Interfaces used **external**ly across components

➢ An example of an **internal interface design** can be seen below:

Note:
We can model this interface easily with UML.



java.util

Interface Observer

public interface **Observer**

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

Since:
JDK1.0

See Also:
`Observable`

Method Summary

`void`	`update(Observable o, Object arg)` This method is called whenever the observed object is changed.

Method Detail

update

```
public void update(Observable o,
                   Object arg)
```

This method is called whenever the observed object is changed. An application calls an `Observable` object's `notifyObservers` method to have all the object's observers notified of the change.

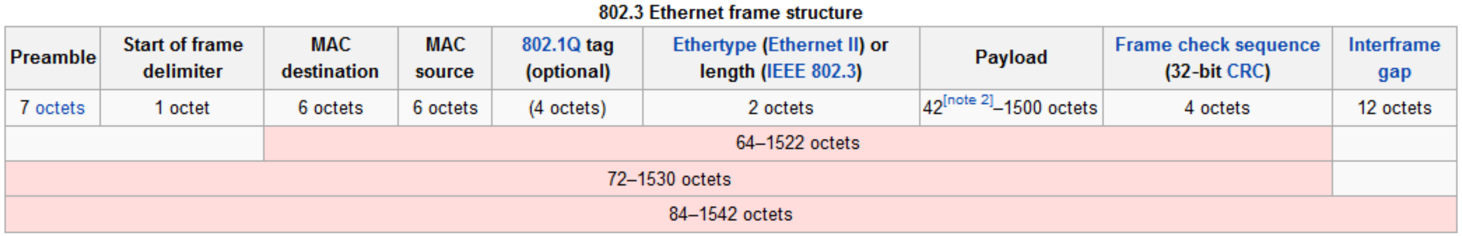
Parameters:
`o` - the observable object.
`arg` - an argument passed to the `notifyObservers` method.

The Observer interface in Java can be used internally within components to support the Observer design pattern.

The design of this interface specifies a well-defined method.

2. CREATING DETAILED DESIGNS

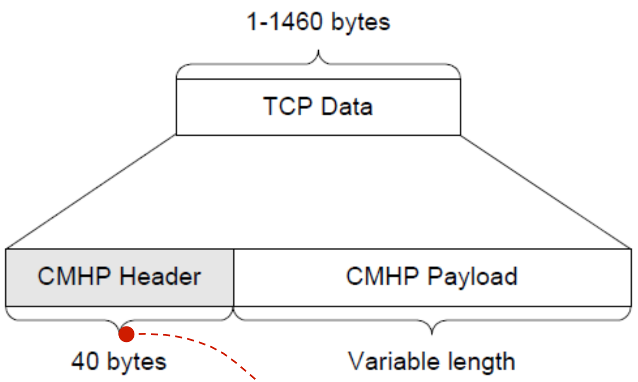
- Example of **external interface design** (from Wikipedia)



802.3 Ethernet frame structure

Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interframe gap
7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	42[note 2]–1500 octets	4 octets	12 octets
		64–1522 octets						
72–1530 octets								
84–1542 octets								

This is already specified by the 802.3 standard, but, you may design your own application-specific messaging specification at the application level. When you do so, you end up with an Interface Design Document containing all the information about the messaging format. For example, see below



The CMHP Header is designed as seen in the table to the right.

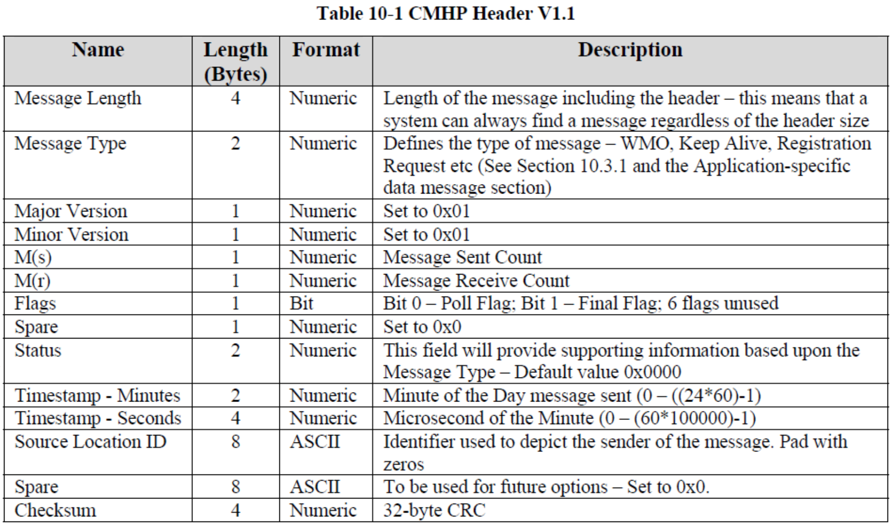


Table 10-1 CMHP Header V1.1

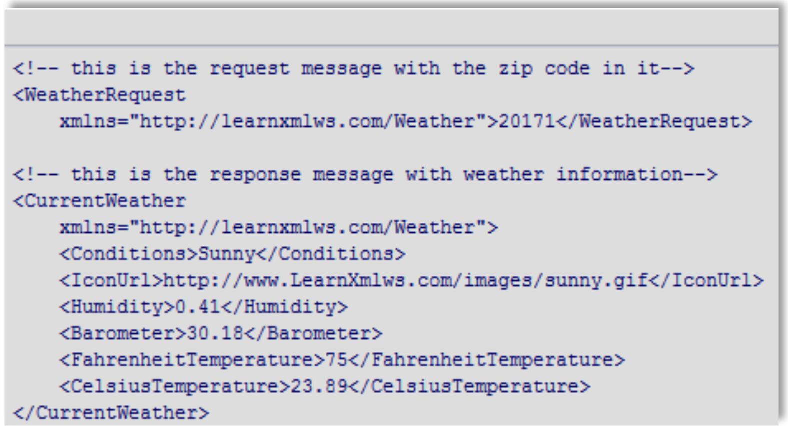
Name	Length (Bytes)	Format	Description
Message Length	4	Numeric	Length of the message including the header – this means that a system can always find a message regardless of the header size
Message Type	2	Numeric	Defines the type of message – WMO, Keep Alive, Registration Request etc (See Section 10.3.1 and the Application-specific data message section)
Major Version	1	Numeric	Set to 0x01
Minor Version	1	Numeric	Set to 0x01
M(s)	1	Numeric	Message Sent Count
M(r)	1	Numeric	Message Receive Count
Flags	1	Bit	Bit 0 – Poll Flag; Bit 1 – Final Flag; 6 flags unused
Spare	1	Numeric	Set to 0x0
Status	2	Numeric	This field will provide supporting information based upon the Message Type – Default value 0x0000
Timestamp - Minutes	2	Numeric	Minute of the Day message sent (0 – ((24*60)-1)
Timestamp - Seconds	4	Numeric	Microsecond of the Minute (0 – (60*100000)-1)
Source Location ID	8	ASCII	Identifier used to depict the sender of the message. Pad with zeros
Spare	8	ASCII	To be used for future options – Set to 0x0.
Checksum	4	Numeric	32-byte CRC

Example extracted from : https://faaco.faa.gov/attachments/5B9EDCCD-D566-F405-67840C21B590D68C.pdf

2. CREATING DETAILED DESIGNS

- Another **example of external interface design in XML**

```
<xs:schema targetName_pace="http://learnxmlws.com/Weather"
elementFormDefault="qualified"
  xmlns="http://learnxmlws.com/Weather"
  xmlns:mstns="http://learnxmlws.com/Weather"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="WeatherRequest" type="xs:string"/>
    <xs:element name="CurrentWeather">
        <xs:complexType>
            <xs:sequence>
               <xs:element name="Conditions"
                            type="xs:string" />
               <xs:element name="IconUrl" type="xs:string" />
               <xs:element name="Humidity"
                            type="xs:float" />
               <xs:element name="Barometer"
                            type="xs:float" />
               <xs:element name="FahrenheitTemperature"
                            type="xs:float" />
                <xs:element name="CelsiusTemperature"
                            type="xs:float" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```



```
<!-- this is the request message with the zip code in it-->
<WeatherRequest
    xmlns="http://learnxmlws.com/Weather">20171</WeatherRequest>

<!-- this is the response message with weather information-->
<CurrentWeather
    xmlns="http://learnxmlws.com/Weather">
    <Conditions>Sunny</Conditions>
    <IconUrl>http://www.LearnXmlws.com/images/sunny.gif</IconUrl>
    <Humidity>0.41</Humidity>
    <Barometer>30.18</Barometer>
    <FahrenheitTemperature>75</FahrenheitTemperature>
    <CelsiusTemperature>23.89</CelsiusTemperature>
</CurrentWeather>
```

Example extracted from link below. For more details of this example, please navigate to the link below

http://msdn.microsoft.com/en-us/magazine/cc188900.aspx#S2

3. EVALUATING DETAILED DESIGNS

- Logical designs are verified using static techniques; that is, through non-execution of the software application.
  - ✓ This makes sense since at this point, the software has not been constructed!
- The most popular technique for evaluating detailed designs involves ***Technical Reviews.*** When conducting technical reviews, keep in mind the following:
  - ✓ Send a review notice with enough time for others to have appropriate time to thoroughly review the design.
  - ✓ Include a **technical expert** in the review team, as well as **stakeholders** of your design.
  - ✓ Include a **member of the software quality assurance** or **testing team** in the review.
  - ✓ During the review, focus on the important aspects of your designs; those that **show how your design helps meet functional and non-functional requirements.**
  - ✓ Document the review process.
    - Make sure that any action items generated during the review are captured and assigned for processing.

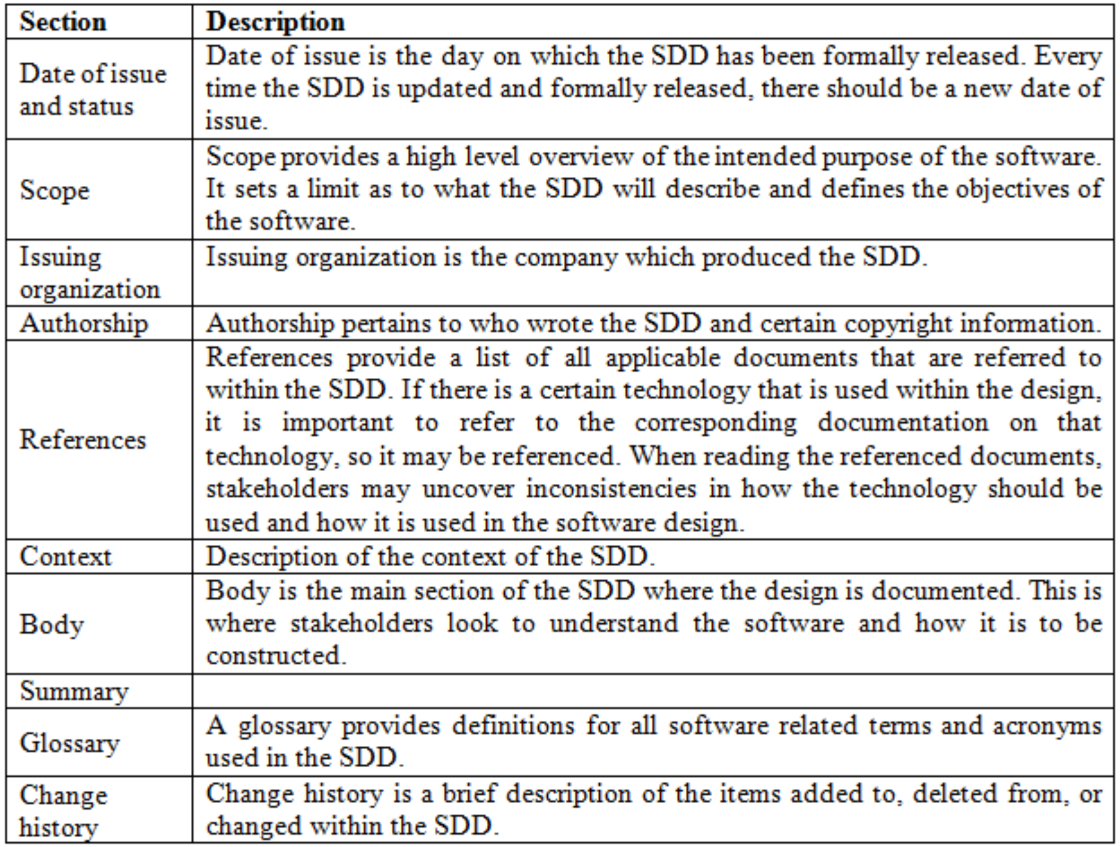
4. DOCUMENTING DETAILED DESIGNS

- Documentation of a project's software design is mostly captured in the **software design document (SDD)**, also known as software design description. The SDD is used widely throughout the development of the software.
  - ✓ Used by programmers, testers, maintainers, systems integrators, etc.

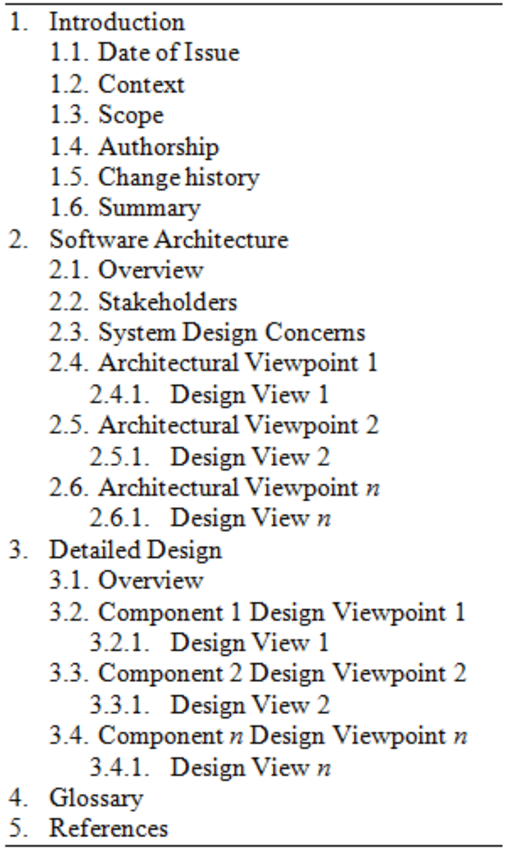
- Other forms of documentation include:
  - ✓ ***Interface Control Document***
    - Serves as written contract between components of the system software as to how they communicate.
  - ✓ ***Version Control Document***
    - Contains information about what is included in a software release, including different files, scripts and executable. Different versions of the design depend on specific software release.

4. DOCUMENTING DETAILED DESIGNS

- The sections of the SDD and sample table of contents:



Section	Description
Date of issue and status	Date of issue is the day on which the SDD has been formally released. Every time the SDD is updated and formally released, there should be a new date of issue.
Scope	Scope provides a high level overview of the intended purpose of the software. It sets a limit as to what the SDD will describe and defines the objectives of the software.
Issuing organization	Issuing organization is the company which produced the SDD.
Authorship	Authorship pertains to who wrote the SDD and certain copyright information.
References	References provide a list of all applicable documents that are referred to within the SDD. If there is a certain technology that is used within the design, it is important to refer to the corresponding documentation on that technology, so it may be referenced. When reading the referenced documents, stakeholders may uncover inconsistencies in how the technology should be used and how it is used in the software design.
Context	Description of the context of the SDD.
Body	Body is the main section of the SDD where the design is documented. This is where stakeholders look to understand the software and how it is to be constructed.
Summary	
Glossary	A glossary provides definitions for all software related terms and acronyms used in the SDD.
Change history	Change history is a brief description of the items added to, deleted from, or changed within the SDD.



1. Introduction
    1.1. Date of Issue
    1.2. Context
    1.3. Scope
    1.4. Authorship
    1.5. Change history
    1.6. Summary
2. Software Architecture
    2.1. Overview
    2.2. Stakeholders
    2.3. System Design Concerns
    2.4. Architectural Viewpoint 1
        2.4.1. Design View 1
    2.5. Architectural Viewpoint 2
        2.5.1. Design View 2
    2.6. Architectural Viewpoint *n*
        2.6.1. Design View *n*
3. Detailed Design
    3.1. Overview
    3.2. Component 1 Design Viewpoint 1
        3.2.1. Design View 1
    3.3. Component 2 Design Viewpoint 2
        3.3.1. Design View 2
    3.4. Component *n* Design Viewpoint *n*
        3.4.1. Design View *n*
4. Glossary
5. References

5. MANAGING IMPLEMENTATION

- **Monitor and control detailed design synchronicity**
- Detailed design synchronicity is concerned with the degree of **how well detailed designs adhere to the software architecture** and **how well software code adheres to the detailed design.**
  - ✓ **Forward & backward traceability**
  - ✓ **Low degree** of synchronicity points to a **flaw in the process** and can lead to **software project failure**.
- Particular attention needs to be paid when projects enter the maintenance phase or when new engineers are brought into the project.
- Processes must be in place to ensure that overall synchronicity is high

SUMMARY…

- In this session, we presented fundamentals concepts of the detailed design activity, including:
  - ✓ What is detailed design?
  - ✓ Key tasks in detailed design

REFERENCES

- [1] IEEE. "IEEE Standard Glossary of Software Engineering Terminology." IEEE, 1990, p.34.
- [2] Clements, Paul, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures. Boston*, MA: Addison Wesley, 2001.
- [3] Sommerville, Ian. Software Engineering, 9th ed. *Boston*, MA: Addison Wesley, 2010.

Session 2: Structural and Behavioral Design of Components

SESSION'S AGENDA

1. Overview of Component Design

2. Designing Internal Structure of Components (OO Approach)
    - ✓ Classes and objects
    - ✓ Interfaces, types, and subtypes
    - ✓ Dynamic binding
    - ✓ Polymorphism

3. Design Principles for Internal Component Design
    - ✓ The open-closed principle
    - ✓ The Liskov Substitution principle
    - ✓ The interface segregation principle

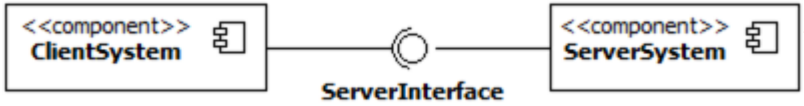
4. Designing Internal Behavior of Components

OVERVIEW OF COMPONENT DESIGN

- Component design (also referred as component-level design) refers to the detailed design task of **defining the internal logical structure and behavior of components.**
  - ✓ That is, refining the structure of components identified during the software architecture activity.
  - ✓ In OO, the internal structure of components identified during architecture can be designed as a single class, numerous classes, or sub components.

During Architecture

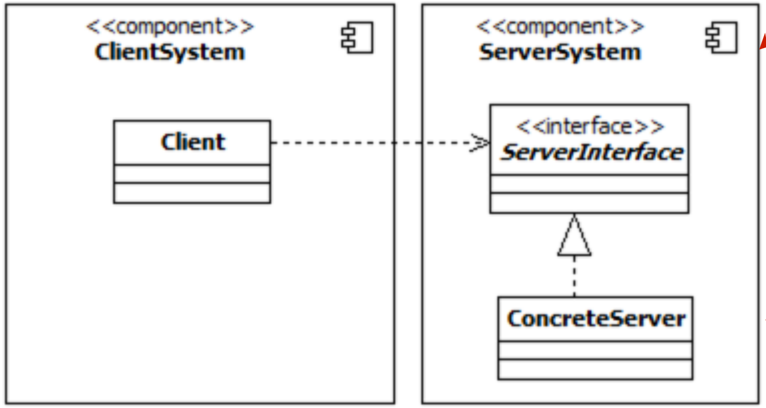
Consider the following components and interface identified during architecture



This is the same as that

During Detailed Design

During the component design task of the detailed design activity, these components are refined to fully define how they realize the component's services



Not quite enough details, but you get the idea, right?

Important:
In OO, during detailed design, we shift away from the more abstract UML component and begin to think in terms of classes, interfaces, types, etc.

OVERVIEW OF COMPONENT DESIGN

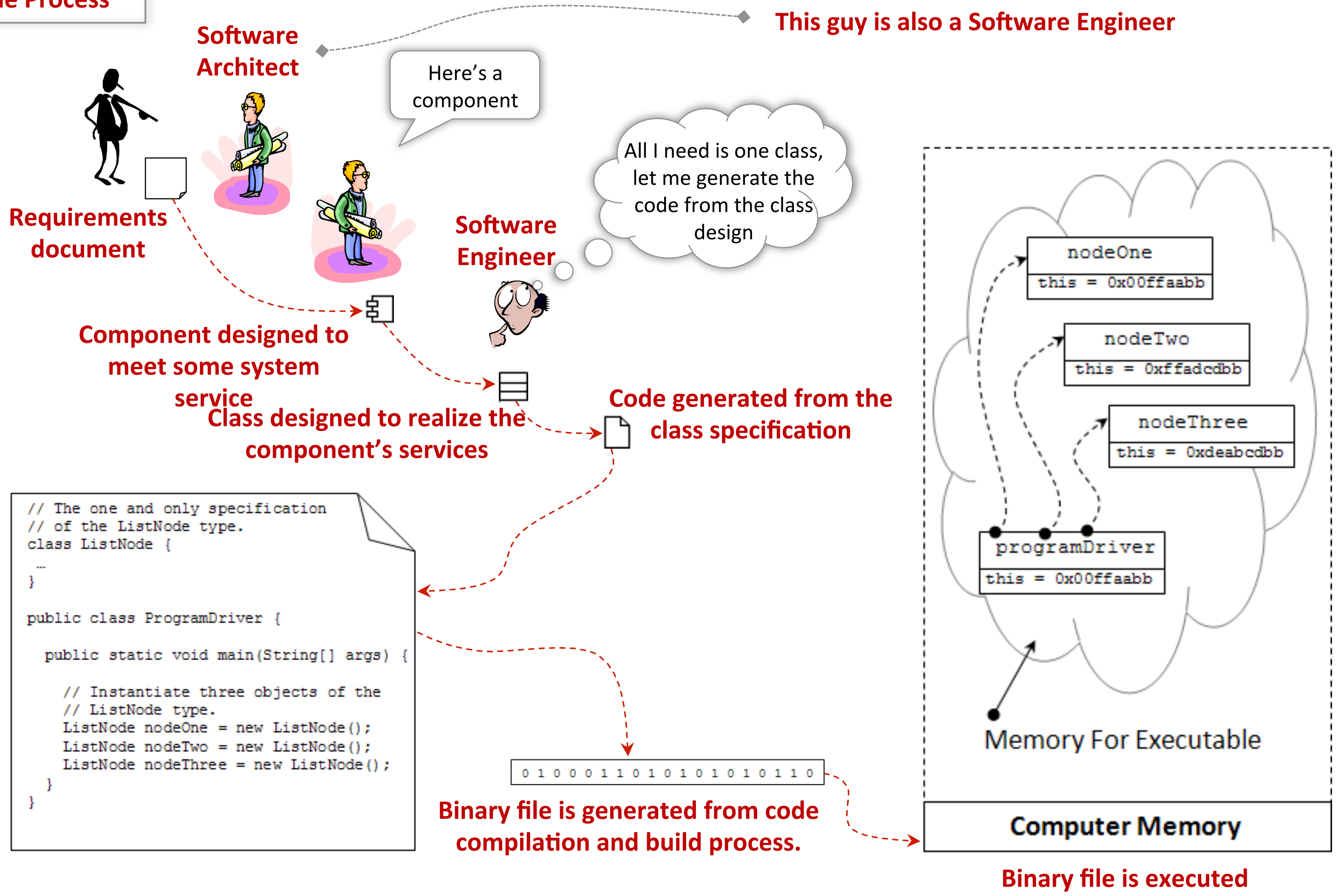
- In **object-oriented systems**, the **internal structure of components** is typically modeled using UML through **one or more class diagrams**.

- During component design, **the internal data structures, algorithms, interface details, and communication mechanisms for all components are defined**.
  - ✓ For this reason, structural and behavioral modes created as part of detailed design provide the most significant mechanism for determining the **functional correctness** of the software system.
  - ✓ This allows us to evaluate alternative solutions before construction begins.

- The work produced during component design contributes significantly to the functional success of the system. In OO, before we can become expert component designers, we must understand the following:
  1. ***Classes and objects***
  2. ***Interfaces, types, and subtypes***
  3. ***Dynamic binding***
  4. ***Polymorphism***

Conceptual View of the Process

OVERVIEW OF COMPONENT DESIGN



This guy is also a Software Engineer

Software Architect

Here's a component

All I need is one class, let me generate the code from the class design

Requirements document

Software Engineer

Component designed to meet some system service

Class designed to realize the component's services

Code generated from the class specification

```
// The one and only specification
// of the ListNode type.
class ListNode {
 …
}

public class ProgramDriver {

  public static void main(String[] args) {

    // Instantiate three objects of the
    // ListNode type.
    ListNode nodeOne = new ListNode();
    ListNode nodeTwo = new ListNode();
    ListNode nodeThree = new ListNode();
  }
}
```

Binary file is generated from code compilation and build process.

Binary file is executed

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN

- In previous modules *(Chp.4- Architecture Styles & Patterns)*, we introduced the concept of quality and discussed several important ones, such as modifiability, performance, etc.

- Let's focus on **modifiability**; what does this mean at the detailed design level? ***Minimizing the degree of complexity involved when changing the system to fit current or future needs***.
  - ✓ This is hard when working with the level of detail that is required during the detailed design activity!
  - ✓ Modifiability cannot be met alone with sound architectural designs; detailed design is crucial to meet this quality attribute.

- Component designs that evolve gracefully over time are hard to achieve.
  - ✓ Therefore, when designing software at the component-level, several principles have to be followed *to create designs that are reusable, easier to modify, and easier to maintain*.

- OO Design principles for internal component design include:
  1. ***The Open-Closed Principle (OCP)***
  2. ***The Liskov Substitution Principle (LSP)***
  3. ***The Interface-Segregation Principle (ISP)***

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)

- The ***Open-Closed principle (OCP)*** is an essential principle for **creating reusable and modifiable** systems that evolve gracefully with time.

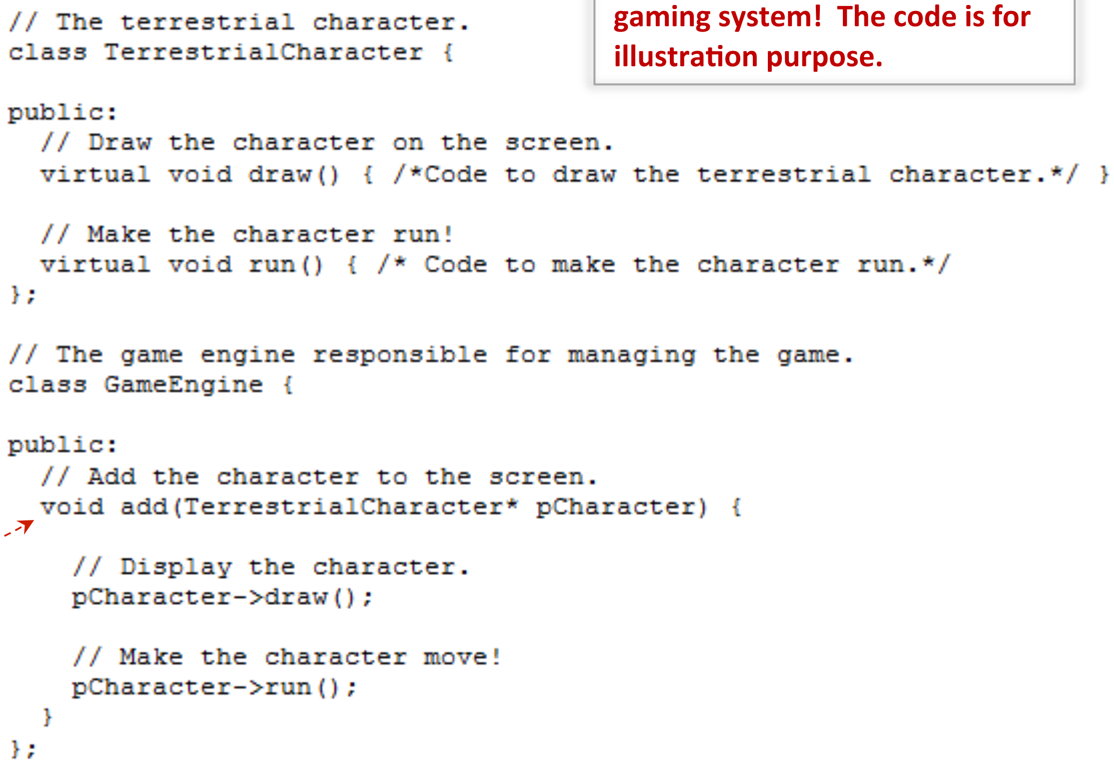
- The OCP was originally coined by Bertrand Meyer [1] and it states that *software designs should be open to extension but closed for modification*.
  - ✓ The main idea behind the OCP is that code that works should remain untouched and that new additions should be extensions of the original work.

- That sounds contradictory, how can that be?
  - ✓ Being close to modifications does not mean that designs cannot be modified; it means that modifications should be done by adding new code, and incorporating this new code in the system in ways that does not require old code to be changed!

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)

Consider a fictional **gaming system** that includes **several types of terrestrial characters**, ones that can roam freely over land. *It is anticipated that new characters will be added in the future.*

Note:
This is really not the code for a gaming system! The code is for illustration purpose.



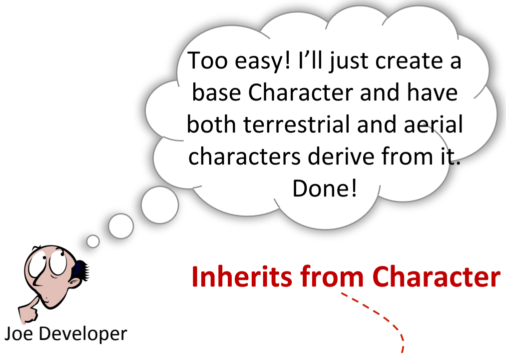
```
// The terrestrial character.
class TerrestrialCharacter {

public:
  // Draw the character on the screen.
  virtual void draw() { /*Code to draw the terrestrial character.*/ }

  // Make the character run!
  virtual void run() { /* Code to make the character run.*/
};

// The game engine responsible for managing the game.
class GameEngine {

public:
  // Add the character to the screen.
  void add(TerrestrialCharacter* pCharacter) {

    // Display the character.
    pCharacter->draw();

    // Make the character move!
    pCharacter->run();
  }
};
```

What can you tell me about the add(…) function?

What happens if we add a new requirement to support other types of characters, e.g., an AerialCharacter that can fly?

Yes, that is right, we would have to change the code inside the add(…) method. This violates the OCP ! Let's see an improved version in the next slide…

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)



```
class Character {

public:
  // Get the type of character.
  virtual string getType() = 0;

  // Draw the character on the screen.
  virtual void draw() = 0;
};
```

Joe Developer decided to abstract the Character concept and separate it from more specific Character types

Inherits from Character

```
class AerialCharacter : public Character {

public:
  // Get the type of character.
  virtual string getType() {

    // Return the type of character.
    return "aerial";
  }

  // Draw the character on the screen.
  virtual void draw() {

    // Code to draw the aerial character.
    cout<<"drawing aerial character!\n";
  }

  // Make the character fly!
  virtual void fly() {

    // Code to make the character fly.
    cout<<"character flying!\n";
  }
};
```

Since Terrestrial characters run and Aerial ones fly, Joe decided to delegate creation of these functions to subtypes, namely, TerrestrialCharacter and AerialCharacter

Are we done? Not really!
The getType(…) function should give you an indication why we're still violating the OCP.
Let's take a closer look in the next slide…

Inherits from Character

```
class TerrestrialCharacter : public Character {

public:
  // Get the type of character.
  virtual string getType() {

    // Return the type of character.
    return "terrestrial";
  }

  // Draw the character on the screen.
  virtual void draw() {

    // Code to draw the terrestrial character.
    cout<<"drawing terrestrial character!\n";
  }

  // Make the character run!
  virtual void run() {

    // Code to make the character run.
    cout<<"character running!\n";
  }
};
```

Note: Character is really an interface, so instead of "Inherits from Character" it (more precisely) realizes the Character interface.

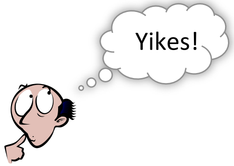
DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)

Design Principle:
Encapsulate Variation

```
class GameEngine {

public:
  // Add a character to the game.
  void add( Character* pCharacter ) {

    // Draw the character on the screen.
    pCharacter->draw();

    // If aerial, make it fly, otherwise, make it run.
    if( pCharacter->getType().compare("aerial") == 0 ) {

      // Downcast the pointer to an aerial character.
      AerialCharacter* pAerial = dynamic_cast<AerialCharacter*>(pCharacter);

      // Assume a valid pointer and make the character fly!
      pAerial->fly();
    }
    else {

      // Downcast the pointer to a terrestrial character.
      TerrestrialCharacter* pTerrestrial =
                           dynamic_cast<TerrestrialCharacter*>(pCharacter);

      // Make the character run!
      pTerrestrial->run();

    } // end if statement.
  } // end add function.
};
```

This code will always vary, depending on the characters in the game!

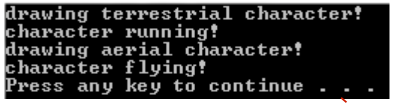
Notice how the GameEngin client needs to know the typ of Character before it can activate it. This is a side-effe of a violation of the OCP



Sample test driver code

```
int _tmain(int argc, _TCHAR* argv[])
{
   // create the mad rabbit character.
   TerrestrialCharacter madRabbit;
   // create the killer bee character.
   AerialCharacter killerBee;

   // create the game engine.
   GameEngine engine;

   // add characters to the game.
   engine.add(&madRabbit);
   engine.add(&killerBee);
   system("pause");

   return 0;
}
```

Sample output



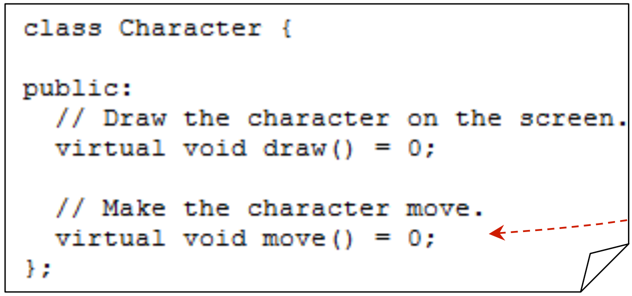
```
drawing terrestrial character!
character running!
drawing aerial character!
character flying!
Press any key to continue . . .
```

It works! We're done!
Not really, we've improved the design, but are we OCP-Compliant?

The Character design still requires clients to know too much about Characters.
What would happen if we now need to support an <u>Aquatic Character</u>?

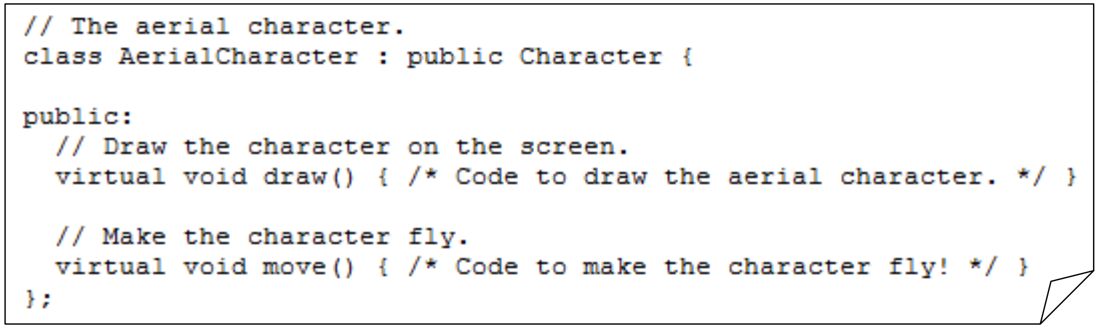
Let's see in the next slide how to make this design OCP-Compliant…

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)



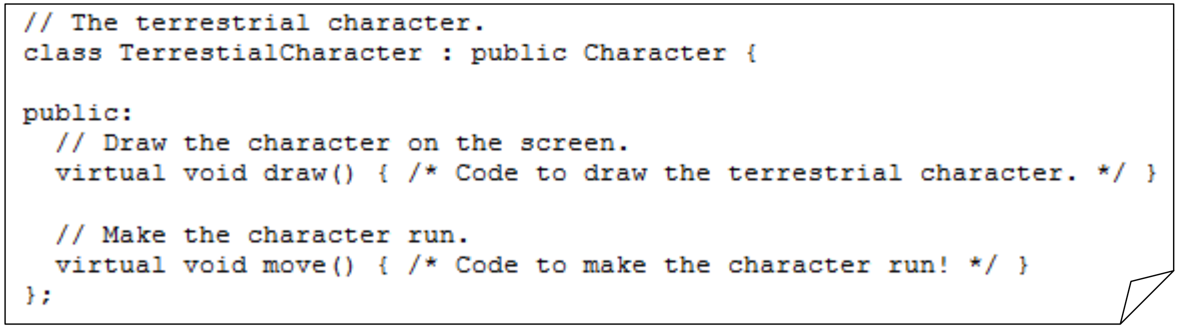
```
class Character {

public:
  // Draw the character on the screen.
  virtual void draw() = 0;

  // Make the character move.
  virtual void move() = 0;
};
```

Encapsulate the movement behavior, so that move(…) works for all characters in the game!



```
// The aerial character.
class AerialCharacter : public Character {

public:
  // Draw the character on the screen.
  virtual void draw() { /* Code to draw the aerial character. */ }

  // Make the character fly.
  virtual void move() { /* Code to make the character fly! */ }
};
```

Per the interface contract, these must provide the implementation for both draw and fly services

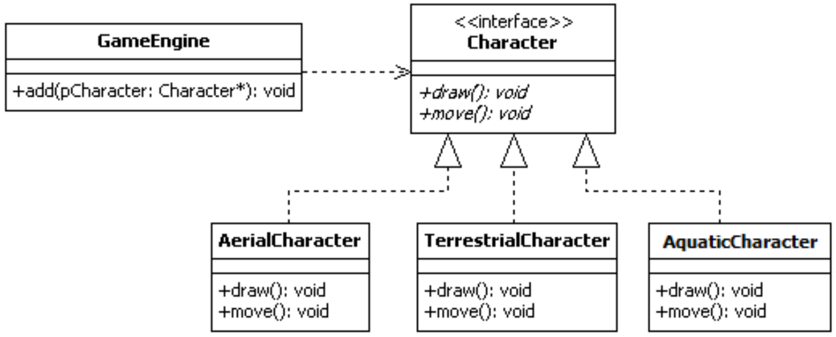


```
// The terrestrial character.
class TerrestialCharacter : public Character {

public:
  // Draw the character on the screen.
  virtual void draw() { /* Code to draw the terrestrial character. */ }

  // Make the character run.
  virtual void move() { /* Code to make the character run! */ }
};
```

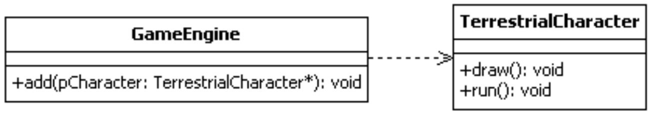
In the next slide, let's see how the code for the GameEngine class looks now based on this new design…

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)

New redesign! Adheres to OCP!



Old design! Violates OCP!



New Aquatic Character added by extension and not by modifying existing working code!

```
// The game engine responsible for managing the game.
class GameEngine {

public:
  // Add the character to the screen.
  void add(Character* pCharacter) {

    // Display the character.
    pCharacter->draw();

    // Activate the character... make it move!
    pCharacter->move();

  } // end add function.
};
```

With this design, GameEngine can draw and activate current and future Characters in the game without modification!

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE OPEN-CLOSED PRINCIPLE (OCP)

One final note about the OCP:
No design will be 100% closed for modification. At some point, some code has to be readily-available for tweaking in any software system. The idea of the OCP is to locate the areas of the software that are likely to vary and the variations can be encapsulated and implemented through polymorphism.

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE LISKOV SUBSTITUTION PRINCIPLE (LSP)

- The LSP was originally proposed by Barbara Liskov and serves as basis for creating designs that **allows clients that are written against derived classes to behave just as they would have if they were written using the corresponding base classes.**

- The LSP requires
  1. **Signatures** between base and derived classes to be maintained
  2. **Subtype specification** supports reasoning based on the super type specification

- In simple terms, LSP demands that "**any class derived from a base class must honor any implied contract between the base class and the components that use it**.“ [2]

- To adhere to the LSP, designs must conform to the following rules:
  1. **The Signature Rule**
  2. **The Methods Rule**

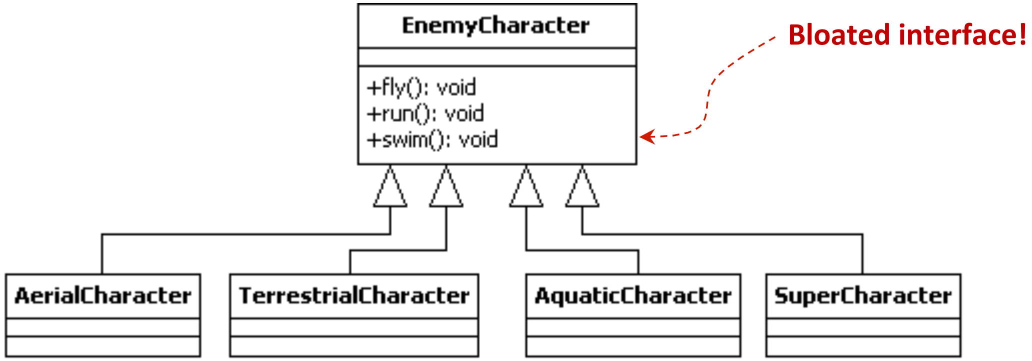
DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
THE LISKOV SUBSTITUTION PRINCIPLE (LSP)

- *The Signature Rule* ensures that if a program is type-correct based on the super type specification, it is also type-correct with respect to the subtype specification.

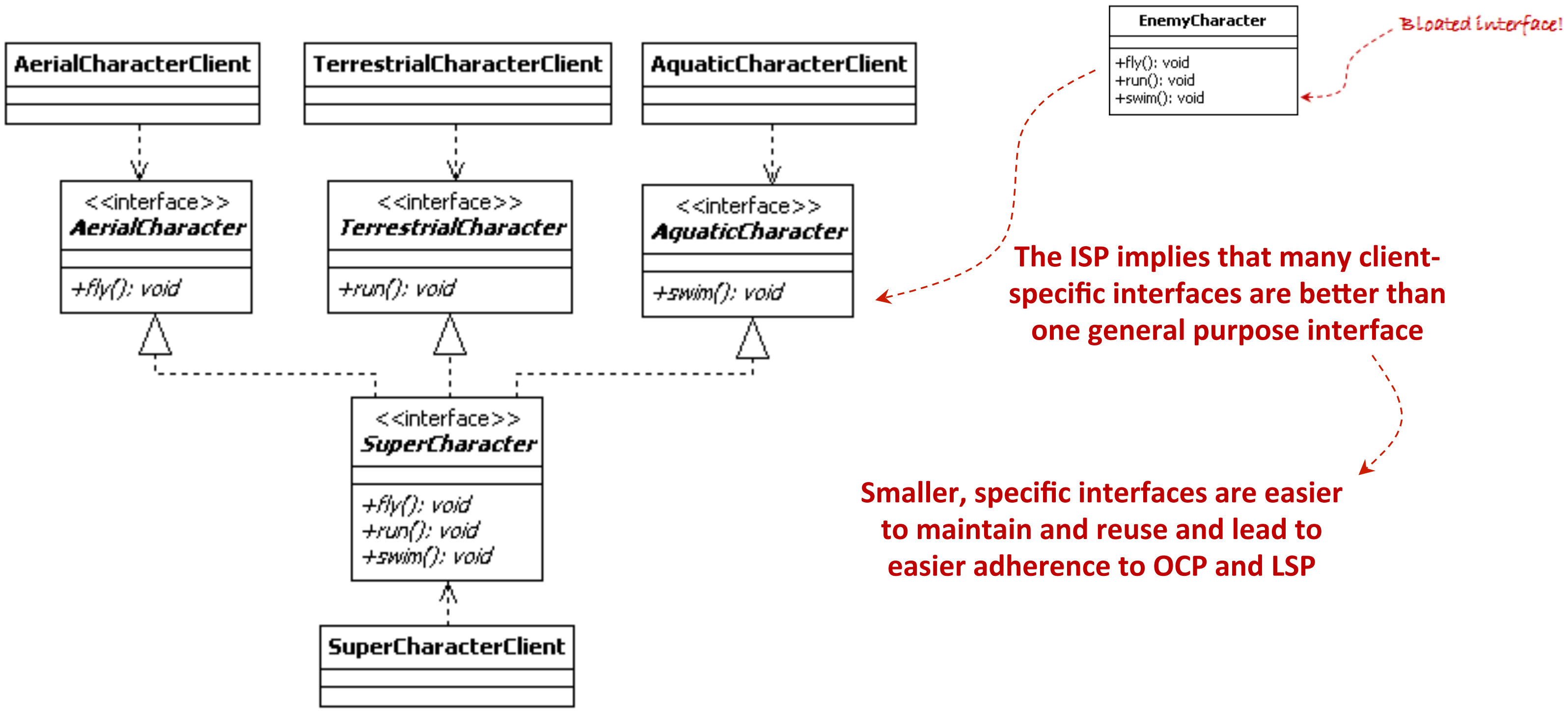
- *The Method Rule* ensures that reasoning about calls of super type methods is valid even though the calls actually go to code that implements a subtype.
  - ✓ Subtype methods can weaken pre-conditions, not strengthen them (i.e., require less, not more).
  - ✓ Subtype methods can strengthen post-conditions, not weaken them (i.e., provide more, not less).

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
INTERFACE SEGREGATION PRINCIPLE (ISP)

- Well designed classes should have one (and only one) reason to change.
- The interface segregation principle (ISP) states that "clients should not be forced to depend on methods that they do not use“ [3].
- Consider a gaming system that supports an advanced enemy character that is able to roam over land, fly, and swim. The game also supports other enemy characters that can either roam over land, fly, or swim.
  - ✓ Some would be tempted to design the system as seen below.

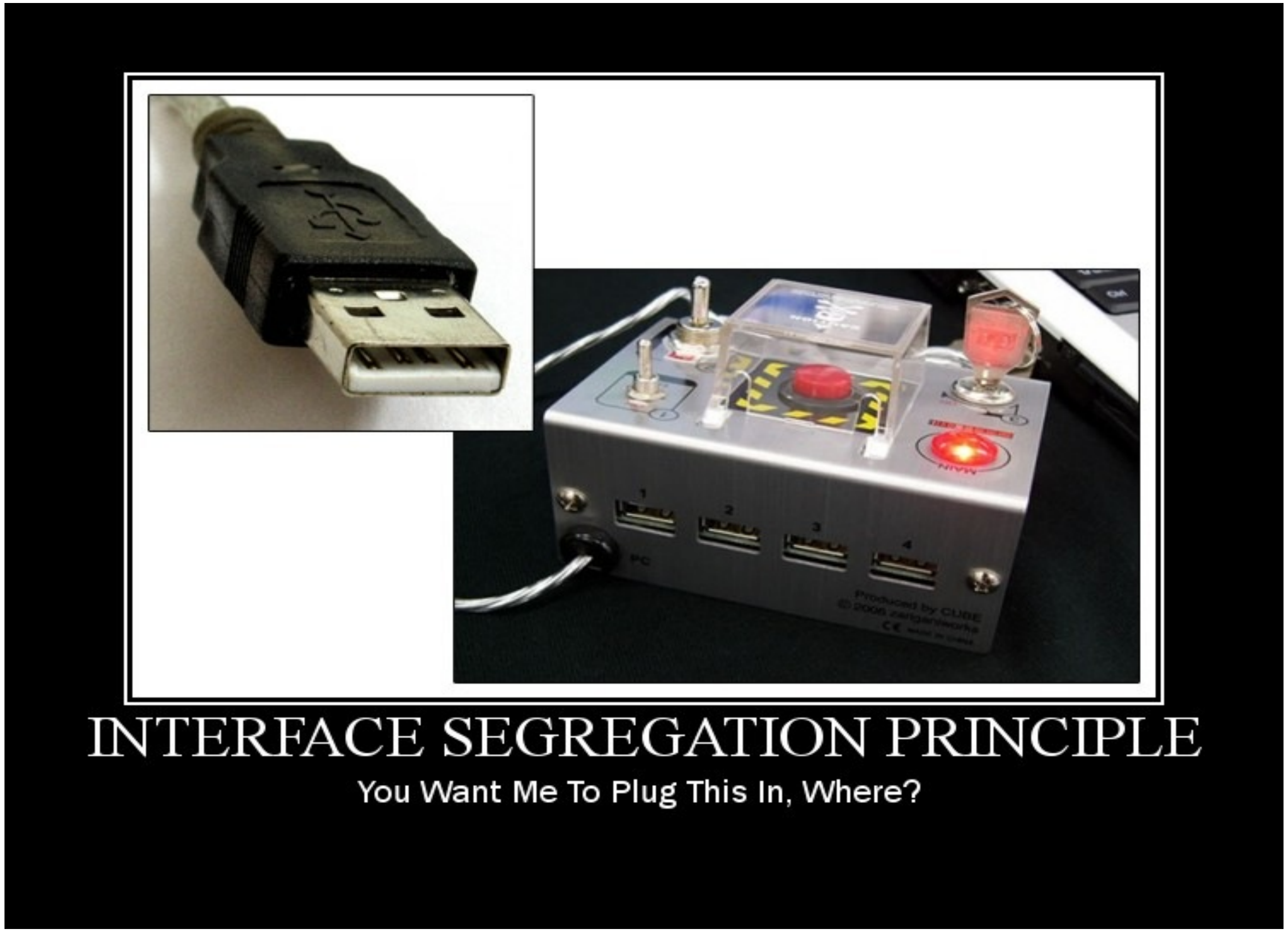


DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN
INTERFACE SEGREGATION PRINCIPLE (ISP)



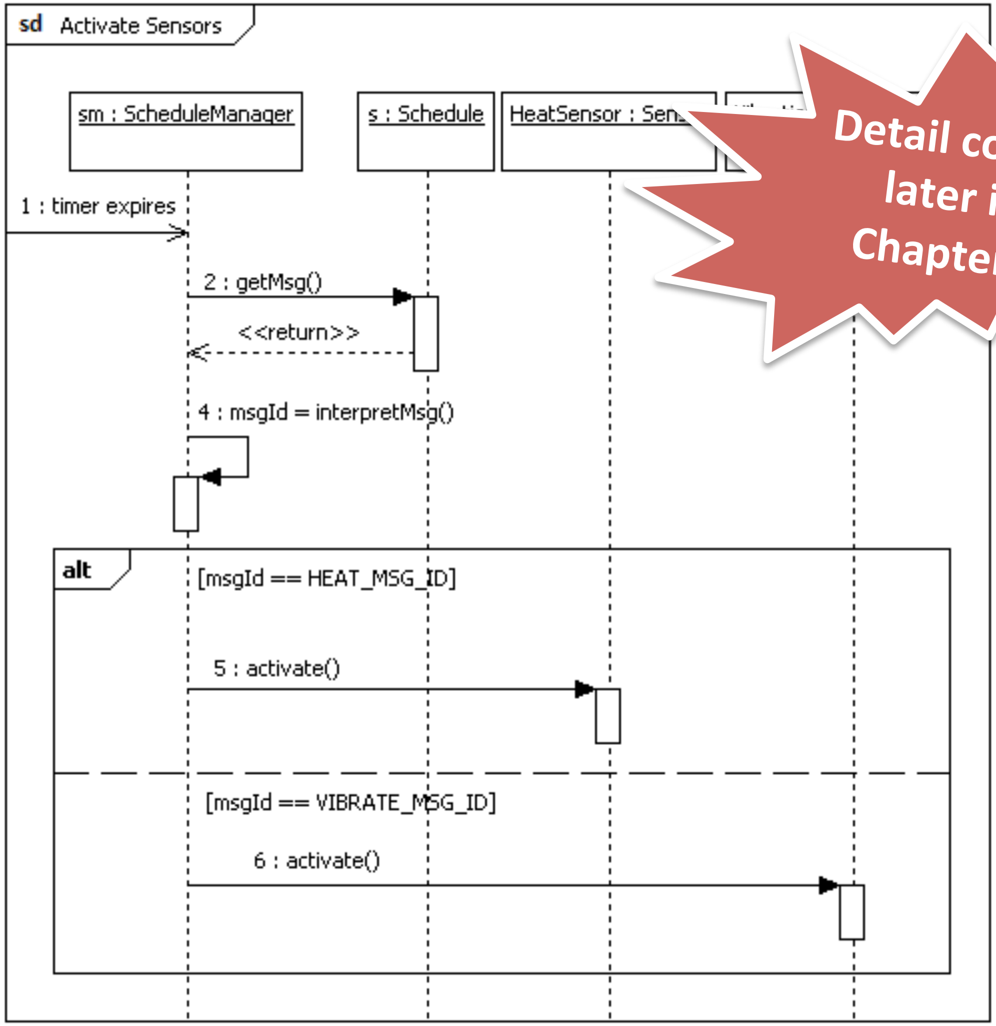
The ISP implies that many client-specific interfaces are better than one general purpose interface

Smaller, specific interfaces are easier to maintain and reuse and lead to easier adherence to OCP and LSP



create smaller well defined interfaces instead of a larger one with many features.

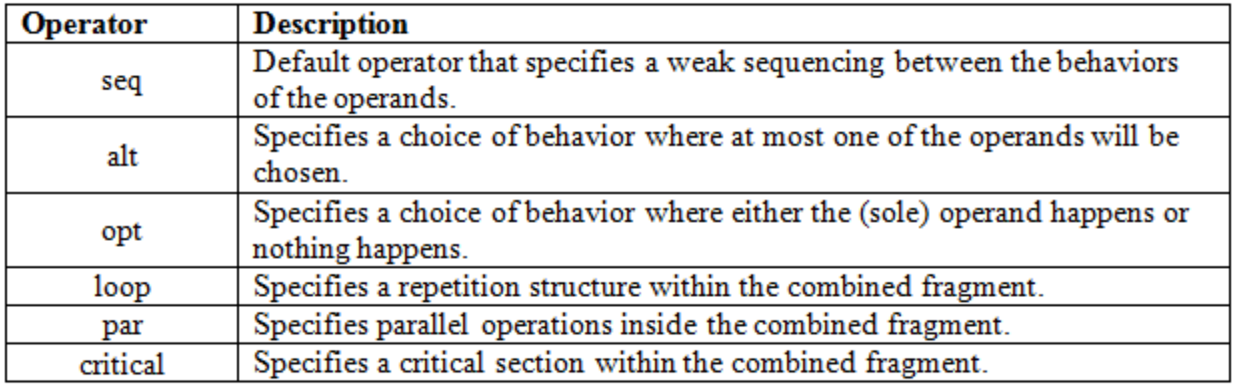
MODELING INTERNAL BEHAVIOR OF COMPONENTS



Detail covers later in Chapter 6

MODELING INTERNAL BEHAVIOR OF COMPONENTS

- Common interaction operators used in sequence diagrams include:



Operator	Description
seq	Default operator that specifies a weak sequencing between the behaviors of the operands.
alt	Specifies a choice of behavior where at most one of the operands will be chosen.
opt	Specifies a choice of behavior where either the (sole) operand happens or nothing happens.
loop	Specifies a repetition structure within the combined fragment.
par	Specifies parallel operations inside the combined fragment.
critical	Specifies a critical section within the combined fragment.

SUMMARY

- In this session, we presented fundamentals concepts of the component design, including:
  - ✓ Overview of Component Design
  - ✓ Designing Internal Structure of Components (OO Approach)
    - Classes and objects
    - Interfaces, types, and subtypes
    - Dynamic binding
    - Polymorphism

  - ✓ Design Principles for Internal Component Design
    - The open-closed principle
    - The Liskov Substitution principle
    - The interface segregation principle

  - ✓ Designing Internal Behavior of Components

REFERENCES

- [1] Meyer, Bertrand. Object-oriented Software Construction, 2d ed. Upper Saddle River, NJ: Prentice Hall, 1997.
- [2] Pressman, Roger S. Software Engineering: A Practitioner’s Approach, 7th ed. Chicago: McGraw-Hill, 2010.
- [3] Marin, Robert C. Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River, NJ: Prentice Hall, 2003.