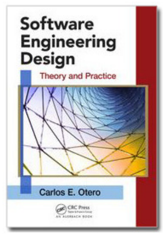
CHAPTER 5: PRINCIPLES OF DETAILED DESIGN

SESSION II: STRUCTURAL AND BEHAVIORAL DESIGN OF COMPONENTS

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



SESSION’S AGENDA

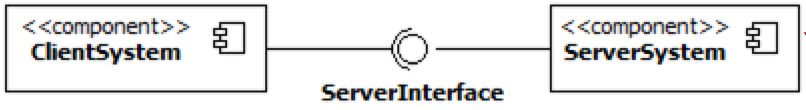
- Overview of Component Design
- Designing Internal Structure of Components (OO Approach)
  - ✓ Classes and objects
  - ✓ Interfaces, types, and subtypes
  - ✓ Dynamic binding
  - ✓ Polymorphism
- Design Principles for Internal Component Design
  - ✓ The open-closed principle
  - ✓ The Liskov Substitution principle
  - ✓ The interface segregation principle
- Designing Internal Behavior of Components
- What’s next…

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

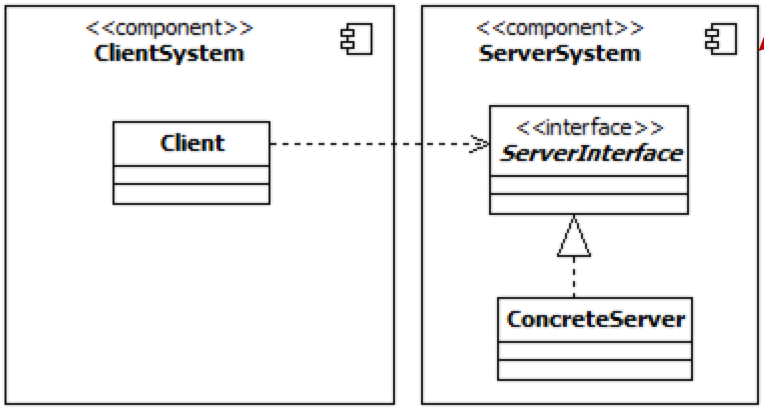
Consider the following components and interface identified during architecture



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.

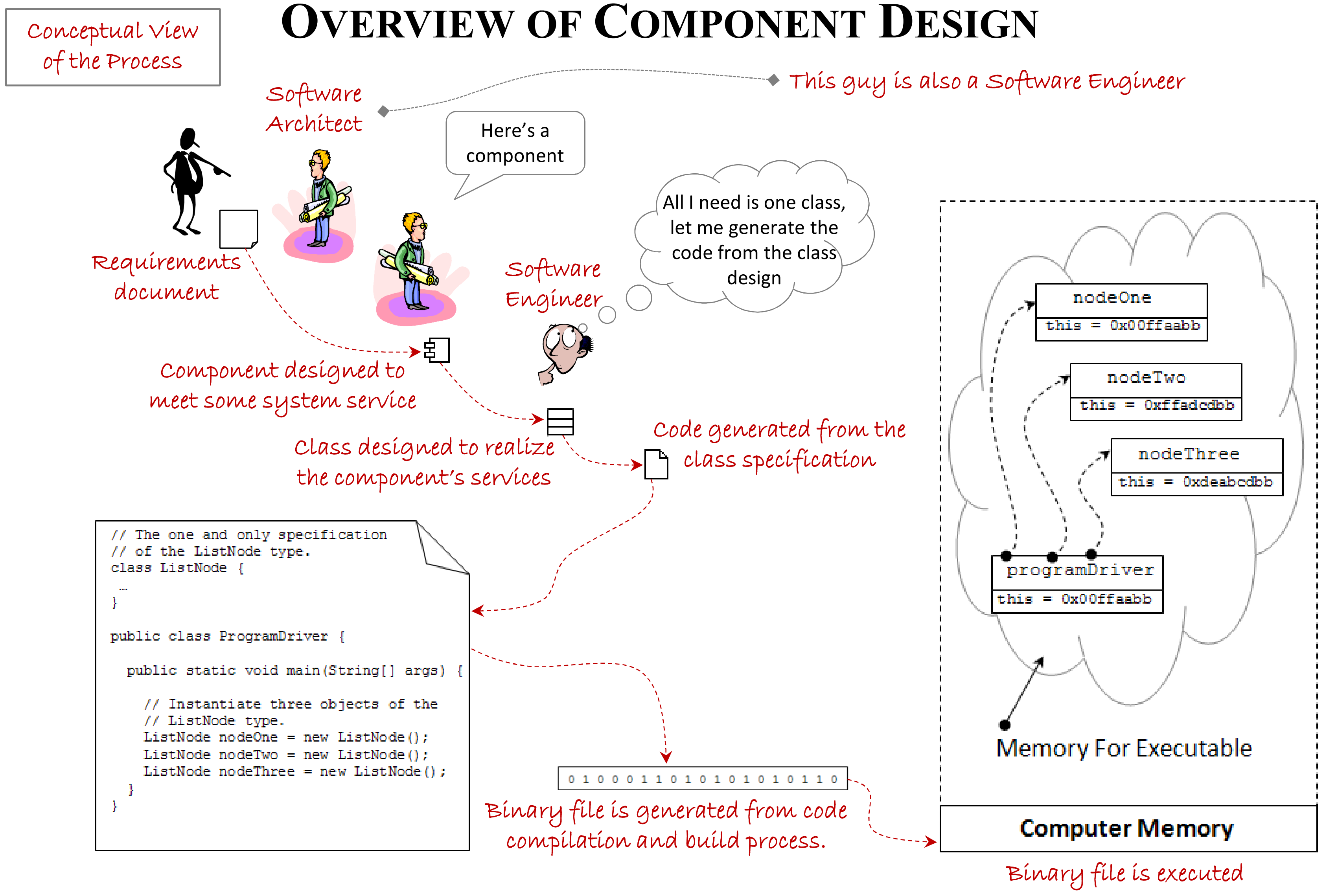
OVERVIEW OF COMPONENT DESIGN

- 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:
  - ✓ Classes and objects
  - ✓ Interfaces, types, and subtypes
  - ✓ Dynamic binding
  - ✓ Polymorphism

Overview of Component Design


Conceptual View of the Process
Software Architect
This guy is also a Software Engineer
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
0 1 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 1 0
Binary file is generated from code compilation and build process.
nodeOne
this = 0x00ffaabb
nodeTwo
this = 0xffadcdbb
nodeThree
this = 0xdeabcdbb
programDriver
this = 0x00ffaabb
Memory For Executable
Computer Memory
Binary file is executed

```
// 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();
  }
}
```

DESIGN PRINCIPLES FOR INTERNAL COMPONENT DESIGN

- In previous modules, 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:
  - ✓ The Open-Closed Principle
  - ✓ The Liskov Substitution Principle
  - ✓ The Dependency-Inversion Principle
  - ✓ The Interface-Segregation Principle

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)

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

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.*

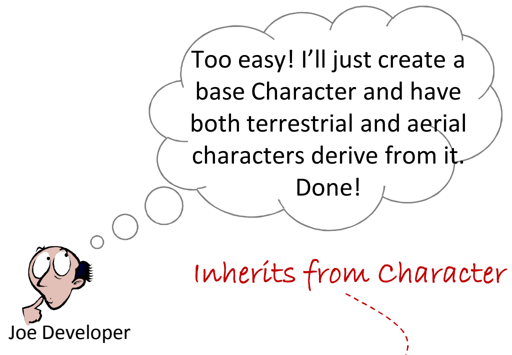
```
// The terrestrial character.
class TerrestrialCharacter {

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)


Too easy! I'll just create a base Character and have both terrestrial and aerial characters derive from it. Done!
Joe Developer

```
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.

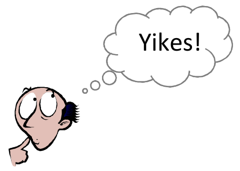
Design Principles for Internal Component Design
The Open-Closed Principle (OCP)

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!

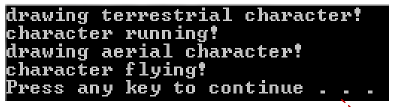
Notice how the GameEngin client needs to know the ty of Character before it can activate it. This is a side-effect of a violation of the O



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! */ }
};
```

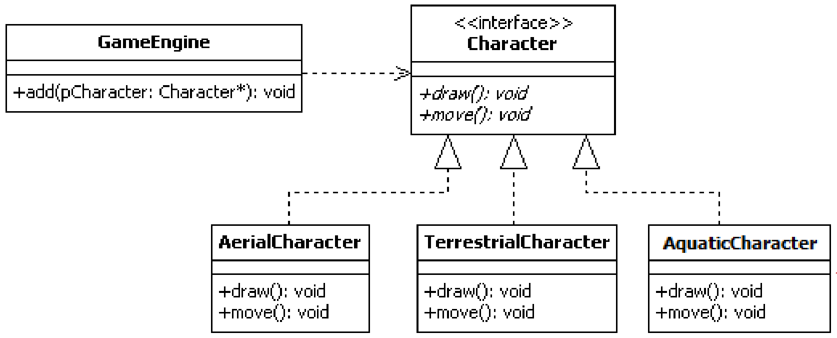
Per the interface contract, these must provide the implementation for both draw and fly services

```
// 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! */ }
};
```

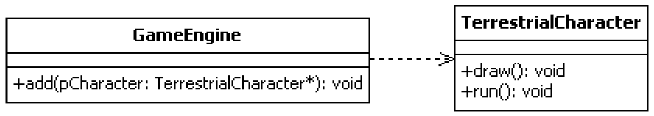
In the next slide, let's see how the code for the GameEngine class looks now based on this new design…

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
  - ✓ Signatures between base and derived classes to be maintained
  - ✓ 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:
  - ✓ The Signature Rule
  - ✓ The Methods Rule

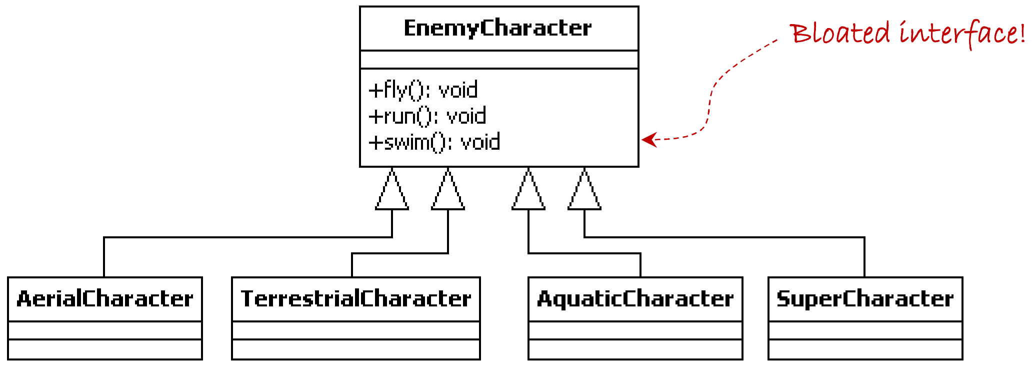
Design Principles for Internal Component Design
The Liskov Substitution Principle (LSP)

- *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.

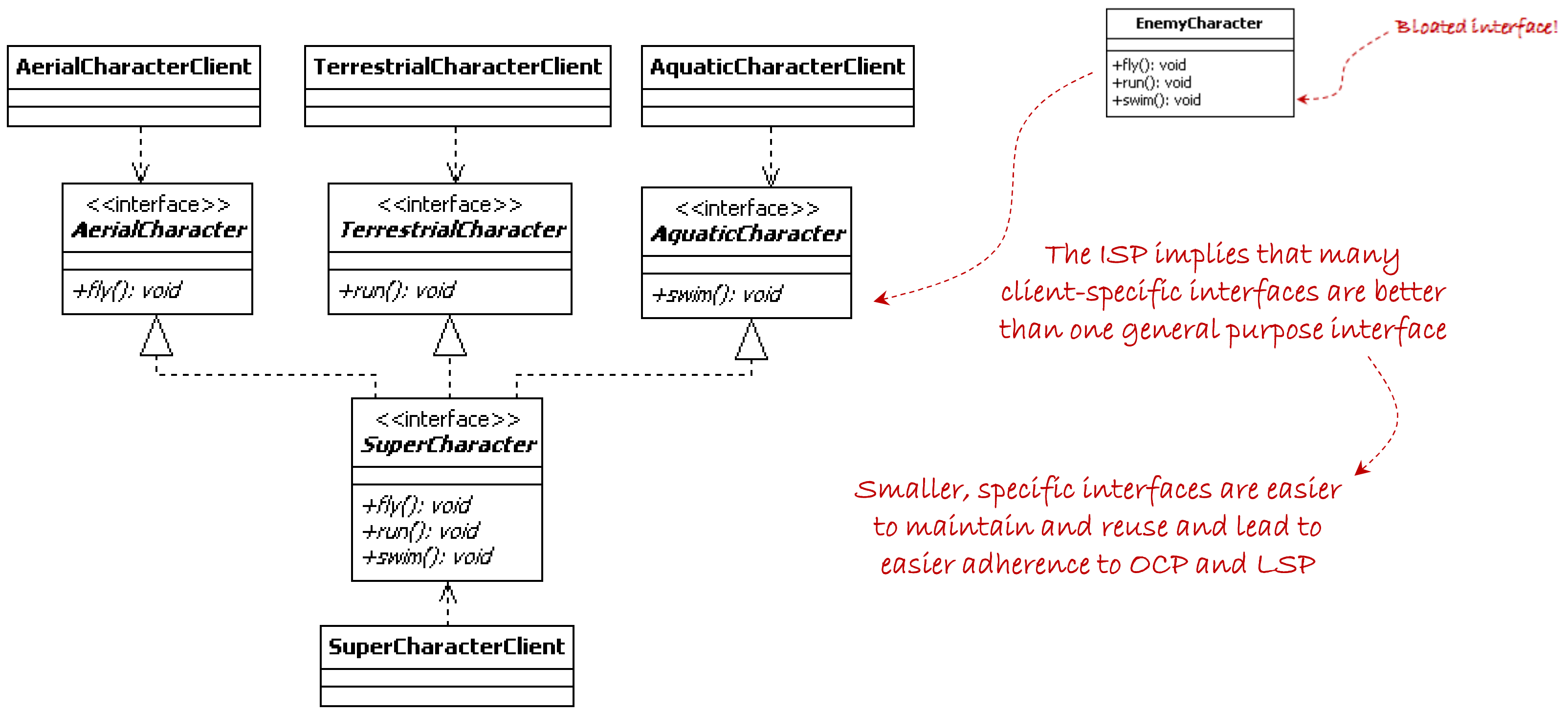
- *The Method Rule* ensures that reasoning about calls of super type methods is valid even though the calls actually go to code that implements a subtype.
  - ✓ Subtype methods can weaken pre-conditions, not strengthen them (i.e., require less, not more).
  - ✓ Subtype methods can strengthen post-conditions,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



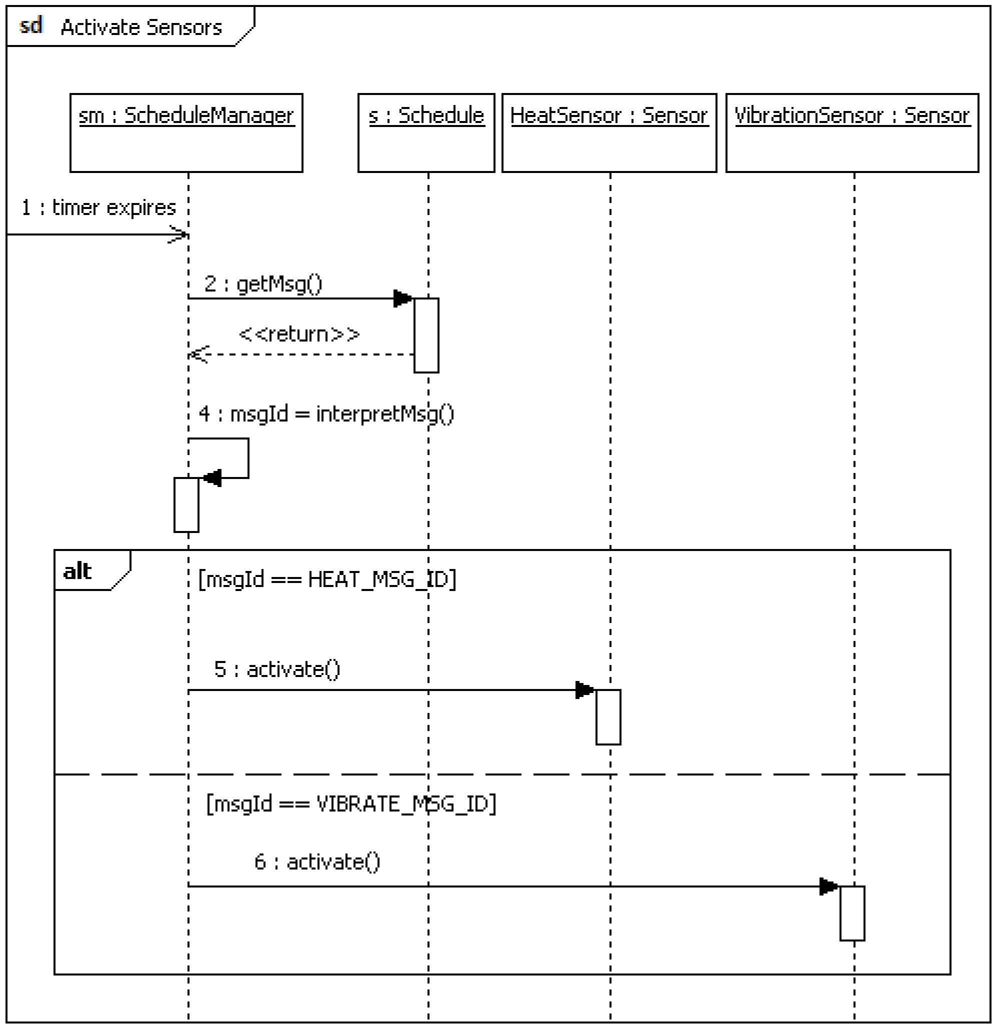
Design Principles for Internal Component Design
Interface Segregation Principle (ISP)



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

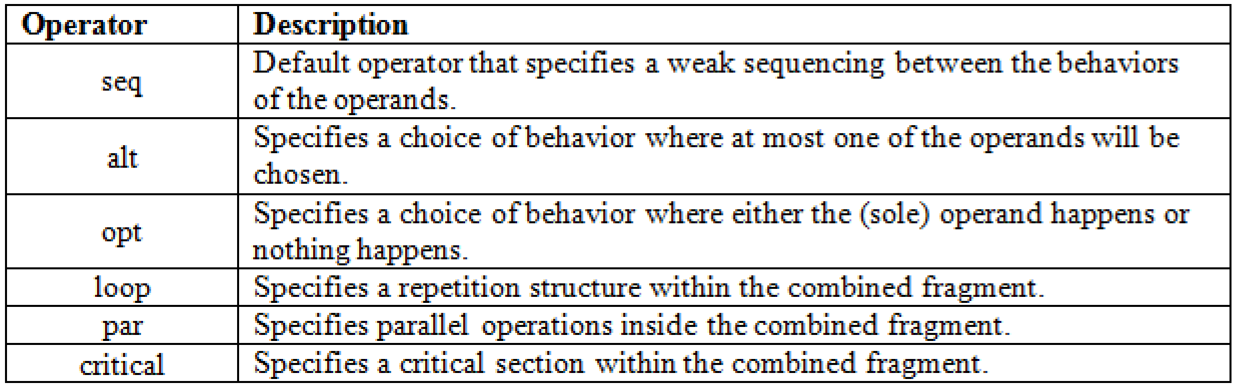
Smaller, specific interfaces are easier to maintain and reuse and lead to easier adherence to OCP and LSP

MODELING INTERNAL BEHAVIOR OF COMPONENTS



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.

What's Next…

- 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
- In the next module, we will present in detail the concept of design patterns, which include :
  - ✓ Creational
  - ✓ Behavioral
  - ✓ Structural

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.