CptS 487
Software Design and Architecture

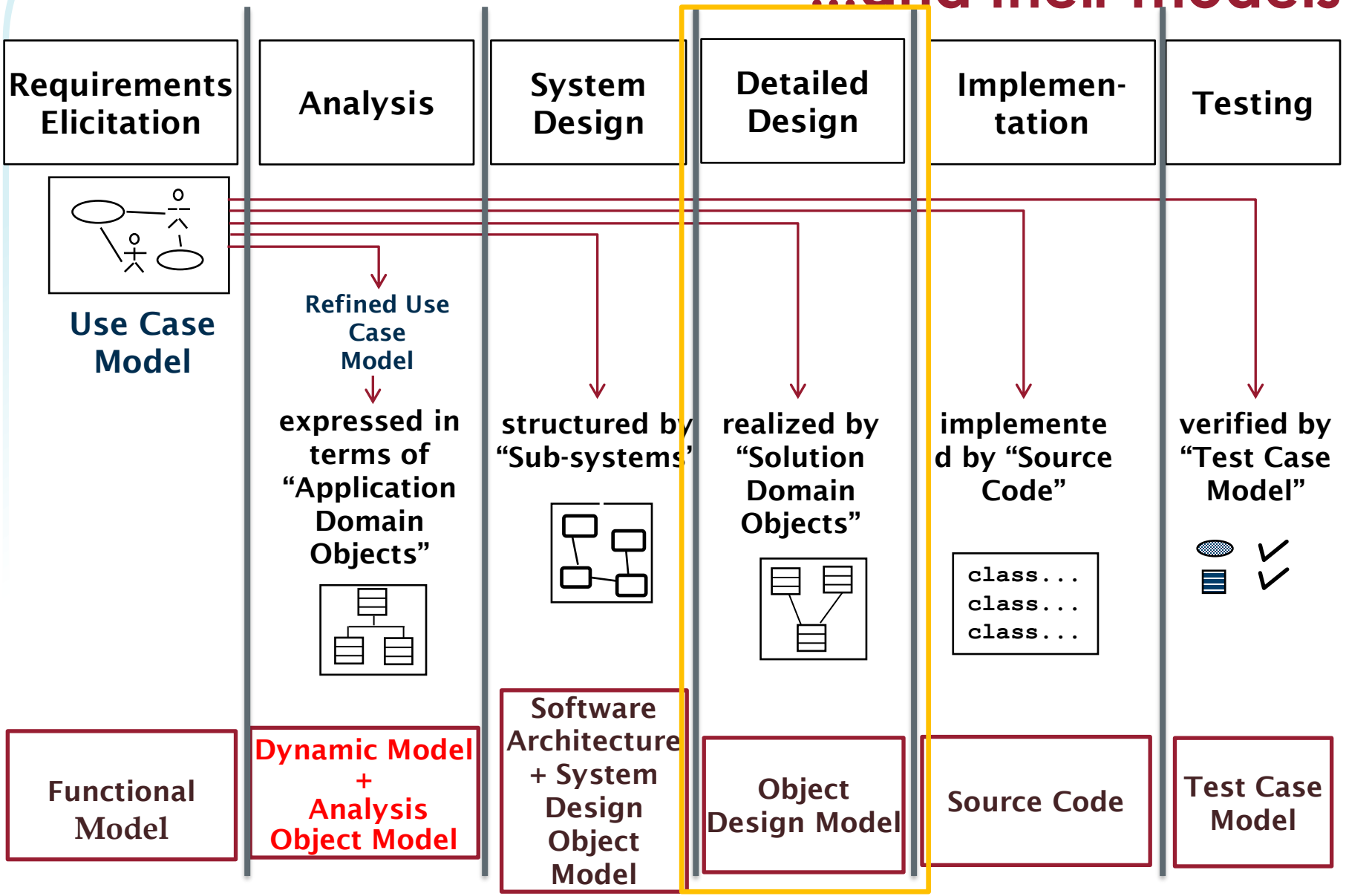
Lesson 29

Object Design

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

Instructor: Bolong Zeng

Software Lifecycle Activities ...and their models



Outline of Today

- Object Design Model
- Object Design Activities
- Reuse examples
  - Whitebox Reuse (Inheritance)
  - Blackbox Reuse (Composition)
- Implementation vs Specification Inheritance
- Inheritance vs Delegation

Object Design

- Purpose of object design:
  - Prepare for the implementation of the system model based on design decisions
  - Transform the system model (optimize it)
- Investigate alternative ways to implement the system model
  - Use design goals: minimize execution time, memory and other measures of cost.
- Object design serves as the basis of implementation.

Object Design Activities Consists of 4 Activities

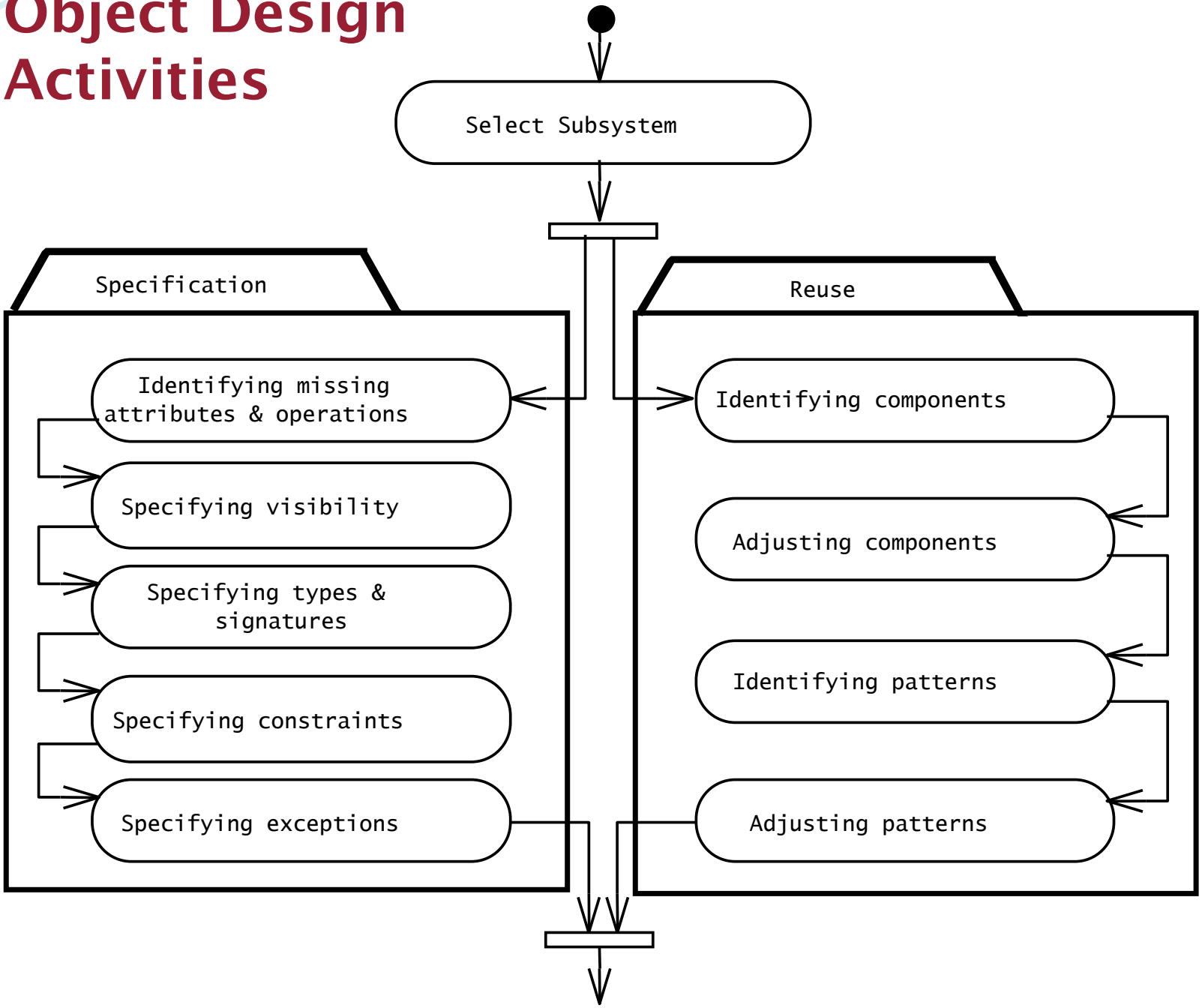
1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Use of delegation (additional solution objects)
   - Use of design patterns
2. Interface specification
   - Describes precisely each class interface

Focus on Reuse and Specification

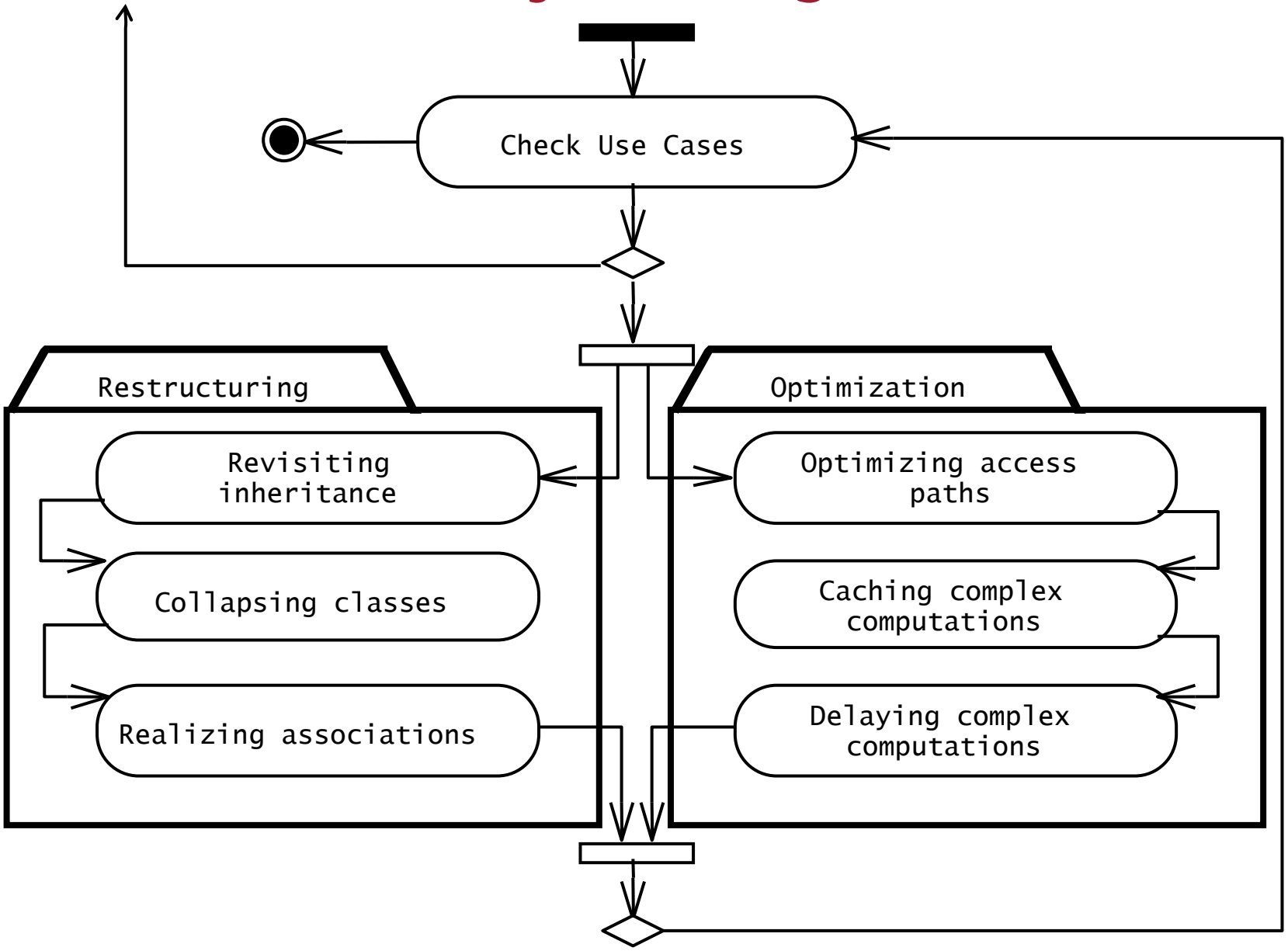
3. Object model restructuring
   - Transforms the object design model to improve its maintainability, understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Towards Mapping Models to Code

Object Design Activities



Detailed View of Object Design Activities (cont.)



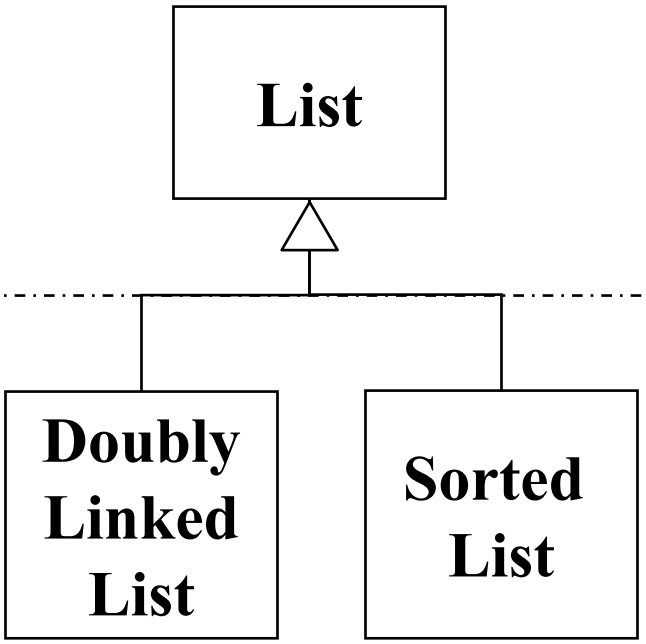
Reuse of Existing Classes

- I have an implementation for a list of elements of type int.
  - Can I reuse this list to build
    - a list of customers?
    - a spare parts catalog?
    - a flight reservation schedule?
- I have developed a class “Addressbook” in a previous project.
  - Can I add it as a subsystem to my e-mail program which I purchased from a vendor (replacing the vendor-supplied address book)?
  - Can I reuse this class in the billing software of my dealer management system?

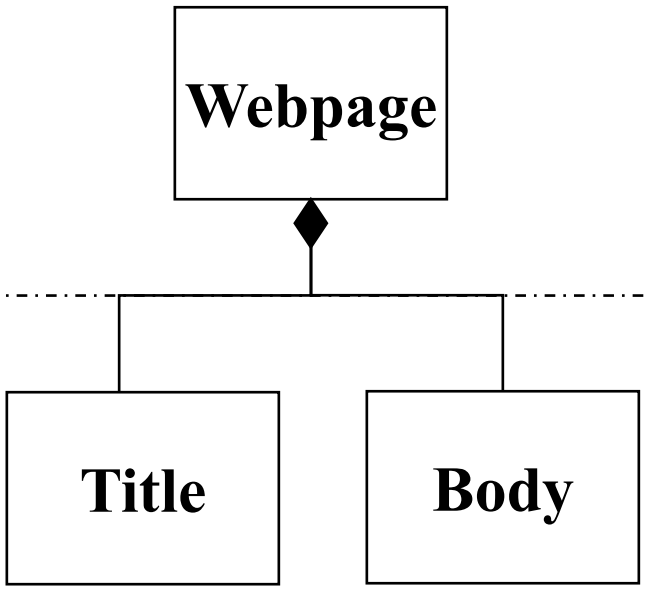
Customization: Build Custom Objects

- During object design we refine and detail the objects identified during analysis and identify additional solution objects.
  - Reuse functionality already available
  - Use design knowledge (from previous experience)
- The two most common techniques for reusing functionality:
  - Inheritance (also called White-box Reuse)
    - The new functionality is obtained by inheritance.
  - Composition (also called Black Box Reuse)
    - The new functionality is obtained by aggregation
    - The new object with more functionality is an aggregation of existing objects

Example of Inheritance



Example of Composition



White Box and Black Box Reuse

- White box reuse (Inheritance)
  - The term white-box refers to visibility: with inheritance the internals of parent classes are visible to subclasses.
  - Access to the development artifacts (analysis model, system design, object design, source code) must be available
- Black box reuse (Composition)
  - Requires that the objects being composed have well defined interfaces.
  - The term black-box implies that no internal details of objects are visible.
    - Access to models and designs is not available, or models do not even exist
      - Worst case: Only executables (binary code) are available
      - Better case: A specification of the system interface is available.

Design Patterns – Putting Reuse Mechanism to Work

- Design patterns show how to apply object oriented principles (such as inheritance and composition/delegation) to build flexible , reusable software.

The use of Inheritance

1. Organization (during analysis):
   - Inheritance helps us with the construction of taxonomies to deal with the application domain
     - Activity: identify application domain objects that are hierarchically related
     - Goal: make the analysis model more understandable
2. Reuse (during object design):
   - Inheritance helps us to reuse models and code to deal with the solution domain
     - Activity:
       - reuse code quickly by subclassing and existing class and refining its behavior,
       - classify concepts into type hierarchies
     - Goal: increase reusability, enhance modifiability and extensibility

Reusing Functionality – Inheritance

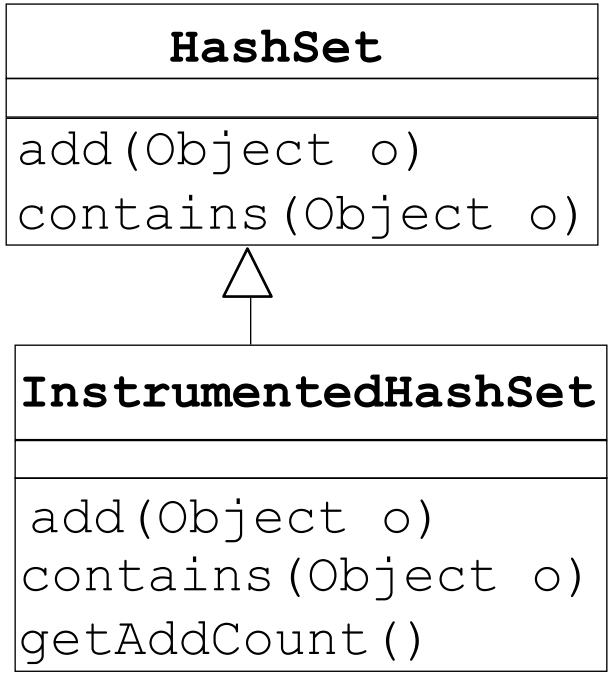
- Method of reuse in which new functionality is obtained by extending the implementation of an existing object
- The generalization class (the superclass) explicitly captures the common attributes and methods
- The specialization class (the subclass) extends the implementation with additional attributes and methods

Advantages/Disadvantages Of Inheritance

- Advantages:
  - New implementation is easy, since most of it is inherited
  - Easy to modify or extend the implementation being reused
- Disadvantages:
  - Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - "White-box" reuse, since internal details of superclasses are often visible to subclasses
  - Subclasses may have to be changed if the implementation of the superclass changes
  - Implementations inherited from superclasses can not be changed at runtime

Inheritance Example

- `java.util.HashSet` class implements the Set interface, backed by a hash table.
- Suppose we want a variant of HashSet class that keeps track of the number of attempted insertions. So we subclass HashSet as follows:



- This example comes from the book *Effective Java by Joshua* Bloch

Inheritance Example (cont.)

```
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet(Collection c) {super(c);}
    public InstrumentedHashSet(int initCap, float loadFactor)
    {
    super(initCap, loadFactor);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

Inheritance Example (cont.)

- Looks good, right. Let's test it!

```
public static void main(String[] args) {
      InstrumentedHashSet s = new InstrumentedHashSet();
      s.addAll(Arrays.asList(new String[]
                           {"Snap","Crackle","Pop"}));
      System.out.println(s.getAddCount());
}
```

- We get a result of 6, not the expected 3. Why?

Inheritance Example (cont.)

- Looks good, right. Let's test it!

```
public static void main(String[] args) {
        InstrumentedHashSet s = new InstrumentedHashSet();
        s.addAll(Arrays.asList(new String[]
                              {"Snap","Crackle","Pop"}));
        System.out.println(s.getAddCount());
}
```

- We get a result of 6, not the expected 3. Why?
  - It's because the internal implementation of addAll() in the HashSet superclass itself invokes the add() method. So first we add 3 to addCount in InstrumentedHashSet's addAll(). Then we invoke HashSet's addAll(). For each element, this addAll() invokes the add() method, which as overridden by InstrumentedHashSet adds one for each element.
  - The result: each element is double counted.

Inheritance Example (cont.)

`addAll` implementation in **java.util.AbstractCollection**

```
318     public boolean addAll(Collection<? extends E> c) {
319         boolean modified = false;
320         Iterator<? extends E> e = c.iterator();
321         while (e.hasNext()) {
322             if (add(e.next()))
323                 modified = true;
324         }
325         return modified;
326     }
```

Inheritance Example (cont.)

- The implementation of the subclass is bound up with the implementation of it's parent class
  - because `InstrumentedHashSet` calls functions of superclass `HashSet`
  - any change in `HashSet` 's implementation will force the `InstrumentedHashSet` to change.

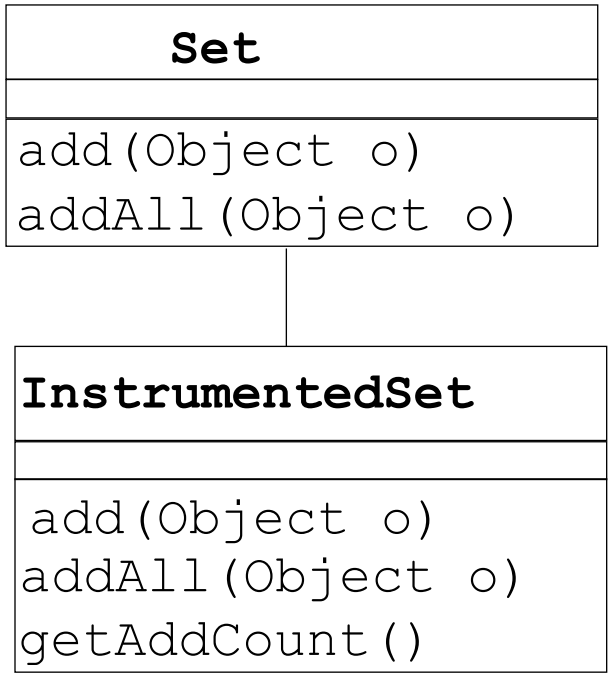
- Implementations inherited from superclasses can not be changed at runtime
  - Because inheritance is defined at compile time

Principle #1

- ***Favor Composition Over Inheritance***
  - ***Roughly implied Open-Closed Principle (OCP)***

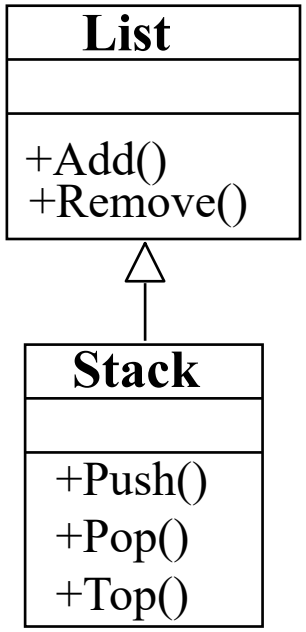
Inheritance vs Composition

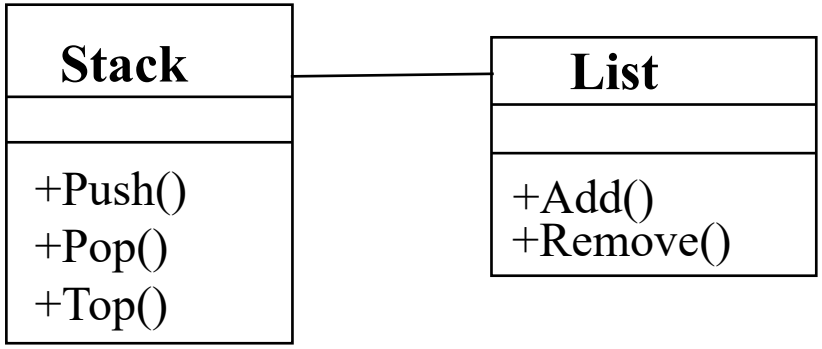
- There are several ways to fix this.
- The best way to fix this is to use composition. Let's write an `InstrumentedSet` class that is composed of a `Set` object.
  - all `Set` operations will actually be forwarded to the contained `Set` object.
- This is an example of delegation through composition!



Delegation Instead of Inheritance

- Inheritance: Extending a parent class by a new operation or overwriting an operation
- Delegation: Implements an operation by sending a message to another class

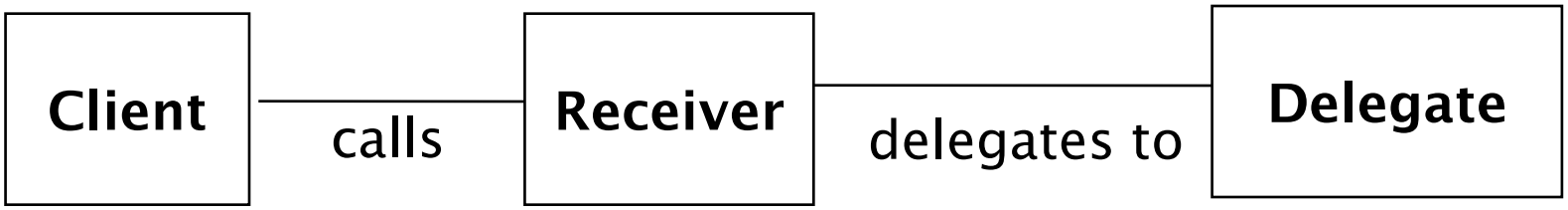




Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance
- In delegation two objects are involved in handling a request from a client

  •The Receiver object delegates operations to the Delegate object
  •The Receiver object makes sure, that the client does not misuse the Delegate object.



Delegation Example (cont.)

```
public class InstrumentedSet {

    Private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {this.s = s;}

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }

    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {return addCount;}
}
```

Delegation Example (cont.)

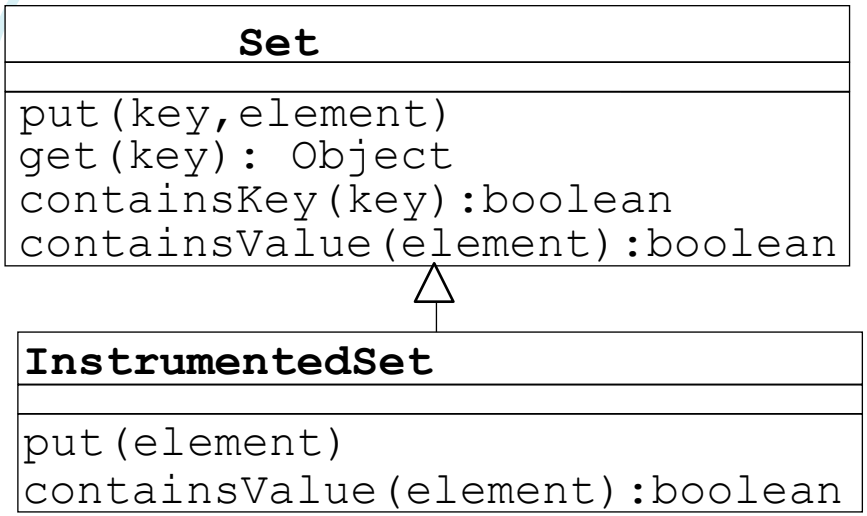
- Note several things about `InstrumentedSet` class:
  - It has one constructor whose argument is a `Set`
  - The contained `Set` object can be an object of any class that implements the `Set` interface (and not just a HashSet)
  - This class is very flexible and can wrap any preexisting `Set` object

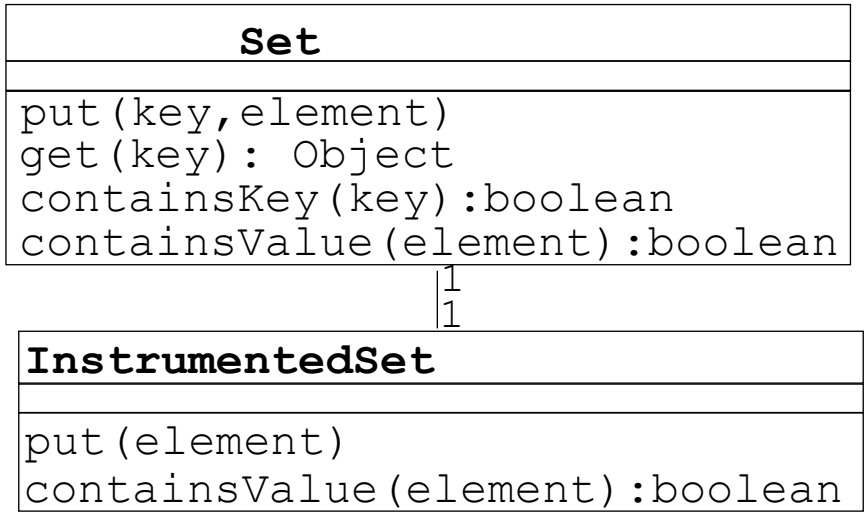
```
List list = new ArrayList();
InstrumentedSet s1 = new InstrumentedSet(new TreeSet(list));

int capacity = 7;
float loadFactor = .66f;
InstrumentedSet s2 = new InstrumentedSet(new HashSet(capacity,
loadFactor));
```

- Both `TreeSet` and `HashSet` classes implement the `Set` interface

Another Inheritance vs Composition Example





Implementation of MySet using inheritance

```
class MySet extends Hashtable {
  MySet() {}
  void put(Object element) {
      if (!containsKey(element)){
           put(element, this);
      }
  }
  boolean containsValue(Object element){
      return containsKey(element)
}
/* Other methods omitted */
}
```

Implementation of MySet using delegation

```
class MySet {
  private Hashtable table;
  MySet() {
       table = Hashtable();
  }
  void put(Object element) {
      if (!containsValue(element)){
               table.put(element,this);
      }
  }
  boolean containsValue(Object element){
      return(table.containsKey(element));
  }
}
```

Comparison: Delegation vs Implementation Inheritance

- Delegation
  - ☺ Flexible: any object can be replaced at run time by another one (as long as it has the same type)
  - ☹ Harder to understand
  - ✓ Delegation is a good design choice when it simplifies more than it complicates
- Inheritance
  - ☺ Straightforward to use
  - ☺ Supported by many programming languages
  - ☺ Easy to implement new functionality in the subclass
  - ☹ Inheritance exposes a subclass to the details of its parent class
  - ☹ Any change in the parent class implementation forces the subclass to change (which requires recompilation of both).

Inheritance/Delegation Summary

- Both composition and inheritance are important methods of reuse
- Inheritance was overused in the early days of OO development
- Over time we've learned that designs can be made more reusable and simpler by favoring composition/delegation over inheritance
- The available set of composable classes can be enlarged using inheritance
- So composition/delegation and inheritance work together
- **Principle #1**
  - ***Favor Composition Over Inheritance***

Principle #2

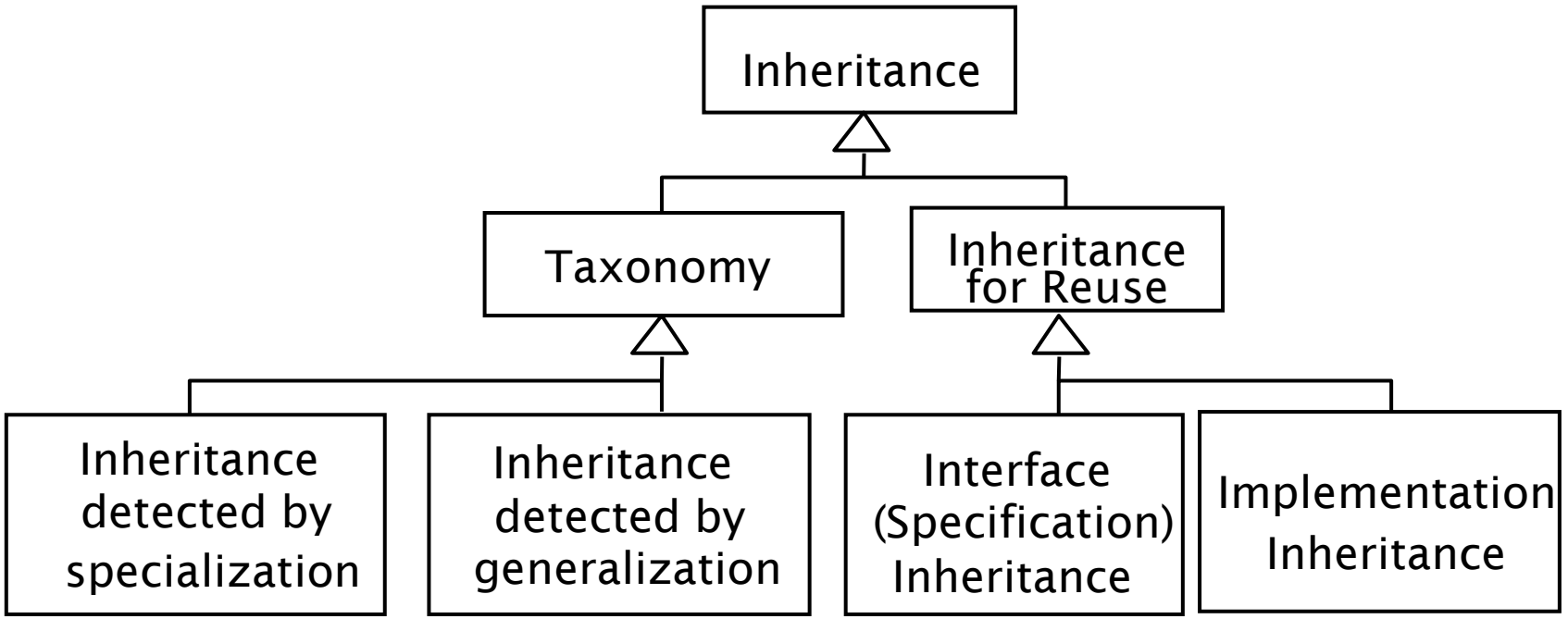
- ***Program To An Interface, Not An Implementation***
  - ***Basically equivalent to Dependency Inversion Principle (DIP)***

Interfaces

- An *interface* *is the set of methods one object knows it can invoke* on another object
- An object can have many interfaces. (Essentially, an interface is a subset of all the methods that an object implements).
- A *type* *is a specific interface of an object*
- Different objects can have the same type and the same object can have many different types
- An object is known by other objects only through its interface
- In a sense, interfaces express "is a kind of" in a very limited way as "is a kind of that supports this interface“
- Interfaces are the key to pluggability!

Inheritance (Revisited)

- In OO analysis and design inheritance is used for achieving several goals:
  - Modeling taxonomies
  - Reusing behavior from abstract classes
    - Interface (specification) inheritance: classification of concepts into type hierarchies (subtyping relationship)
    - Implementation inheritance: reuse code quickly by subclassing an existing class and refining it's behavior



Implementation Inheritance vs Interface (Specification) Inheritance

- *Implementation Inheritance (Class Inheritance) - an object's* implementation is defined in terms of another's objects implementation
- *Interface (Specification) Inheritance - describes when one object can* be used in place of another object
- The C++ inheritance mechanism means both class and interface inheritance
- C++ can perform interface inheritance by inheriting from a pure abstract class
- Java has a separate language construct for interface inheritance - the Java `interface`
- Java's `interface` construct makes it easier to express and implement designs that focus on object interfaces

Interface Example

```
/**
* Interface IManeuverable provides the specification
* for a maneuverable vehicle.
*/
public interface IManeuverable {
 public void left();
 public void right();
 public void forward();
 public void reverse();
 public void climb();
 public void dive();
 public void setSpeed(double speed);
 public double getSpeed();
}
```

Interface Example (cont.)

```
public class Car
  implements IManeuverable { // Code here. }

public class Boat
  implements IManeuverable { // Code here. }

public class Submarine
  implements IManeuverable {// Code here. }
```

Interface Example

- In some other class, the `travel` method can maneuver the vehicle without being concerned about what the actual class is (car, boat, submarine) or what inheritance hierarchy it is in.

```
public void travel(IManeuverable vehicle) {
  vehicle.setSpeed(35.0);
  vehicle.forward();
  vehicle.left();
  vehicle.climb();
}
```

Principle #2

- ***Program To An Interface, Not An Implementation***

Benefits Of Interfaces

- Advantages:
  - Clients are unaware of the specific class of the object they are using
  - One object can be easily replaced by another
  - Object connections need not be hardwired to an object of a specific class, thereby increasing flexibility
  - Loosens coupling
  - Increases likelihood of reuse
  - Improves opportunities for composition since contained objects can be of any class that implements a specific interface
- Disadvantages:
  - Modest increase in design complexity

Principle #3

The Liskov Substitution Principle:

- ***Functions That Use References To Base (Super) Classes Must Be Able To Use Objects Of Derived (Sub) Classes Without Knowing It***

Summary

- Object design adds details to the requirements analysis and makes implementation decisions
- Object design activities:
  - ✓ Identification of Reuse
  - ✓ Identification of Inheritance and Delegation
- Object oriented design principles