Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Brideg	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Composite: Intent

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

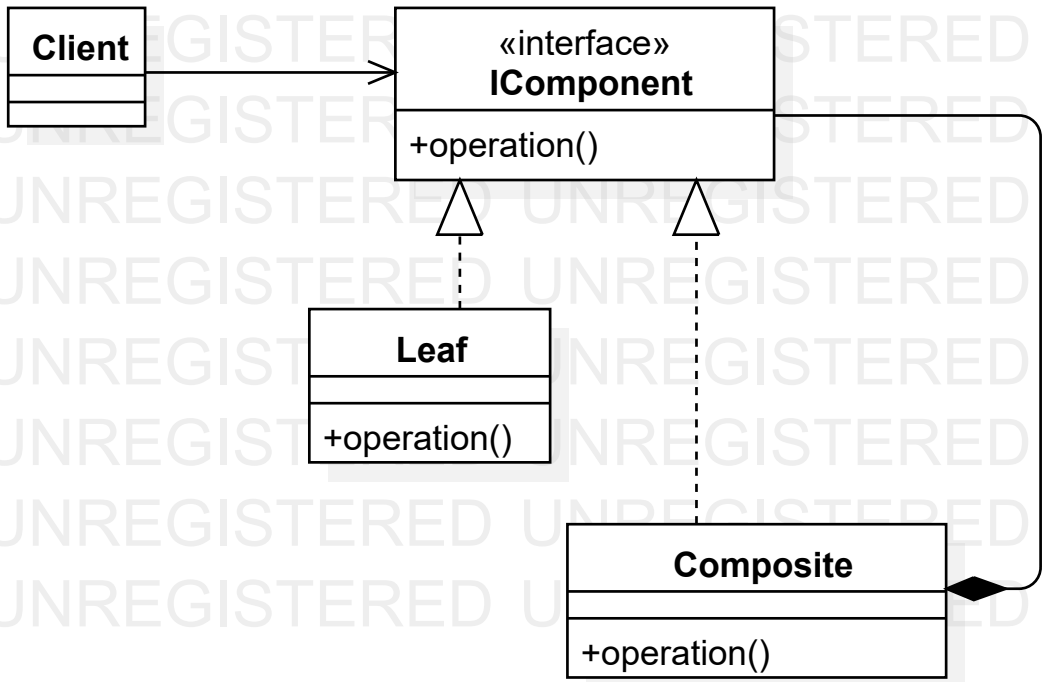
Composite: Applicability

- Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite Pattern

- A composite design pattern is meant to achieve two goals:
  - To compose nested structures of objects, and
  - To deal with the classes for theses objects uniformly.

Composite Pattern: Structure



Composite Pattern (contd.)

- In this design, a component interface serves as the supertype for a set of classes.
- Using polymorphism, all implementing classes conform to the same interface, allowing them to be dealt with uniformly.

The Leaf Class and the Composite Class

- A **composite class** is used **to aggregate any class** that implements the component interface.
- The composite class allows you to "traverse through" and "potentially manipulate" the component objects that the composite object contains.
- A **leaf class** represents a non-composite type. It is not composed of other components.

The Leaf Class and the Composite Class (contd.)

- The Leaf class and the Composite class implement the Component interface, unifying them with a single type.
- This allows us to deal with non-composite and composite objects **uniformly**—the Leaf class and the Composite class are now considered subtypes of Component.
- You may have other composite or leaf classes in practice, but there will only be one overall component **interface or abstract superclass**.

Recursive Composition

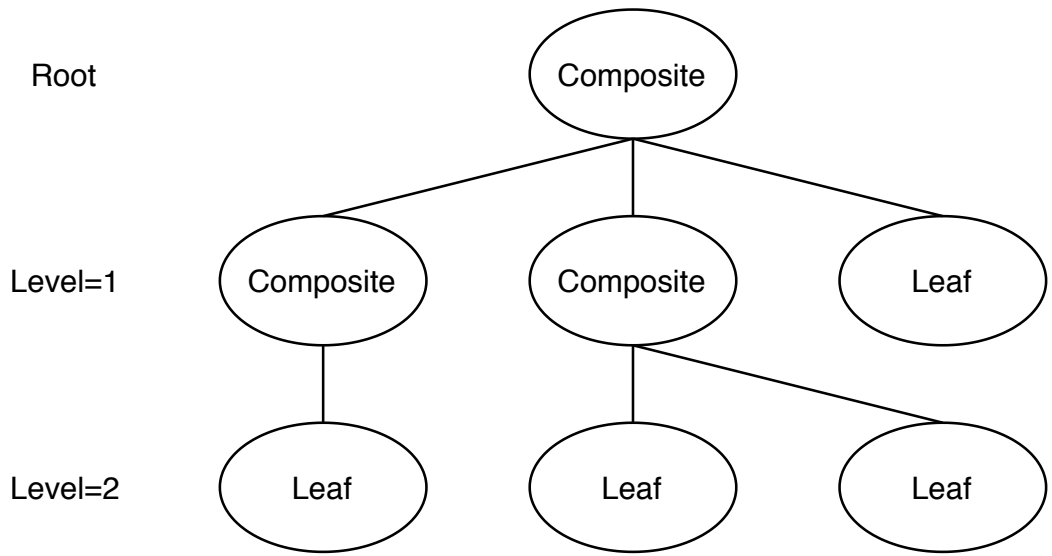
- A composite object can contain other composite object, since the composite class is a subtype of component. This is known as **recursive composition**.

- This term is also a synonym for composite design patterns.

Composite Design Patterns as Trees

- This design pattern has a composite class with a cyclical nature, which may make it difficult to visualize.

- Instead, it is easier to think of composite design patterns as trees:



Composite Design Patterns as Trees (contd.)

- The main composite object, which is made up of other component objects, is at the root level of the tree.
- At each level, more components can be added below each composite object, like another composite or a leaf.
- Leaf objects cannot have components added to them.
- Composites therefore have the ability to "grow" a tree, while a leaf ends the three where it is.

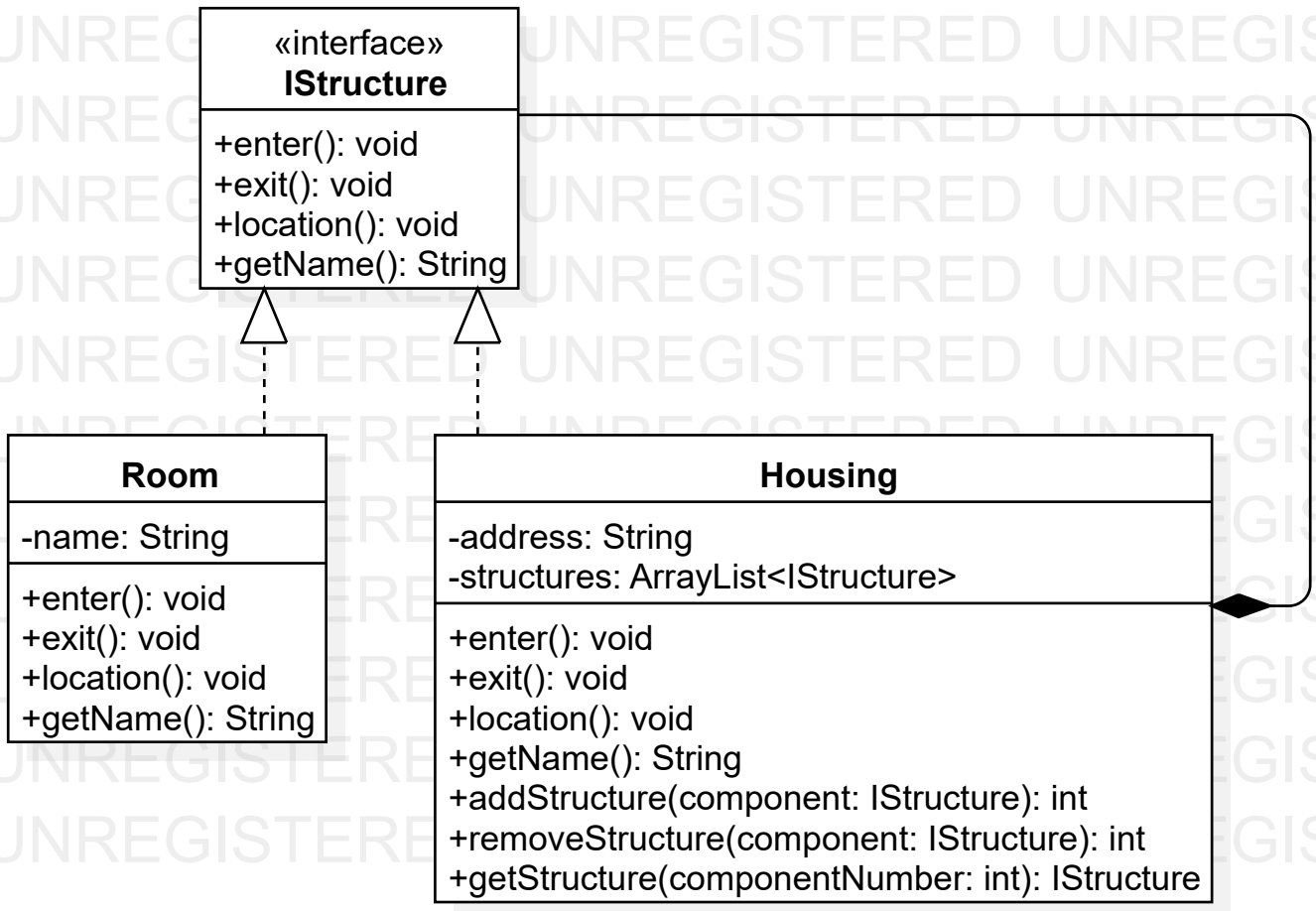
Composite Design Patterns as Trees (contd.)

- At the beginning of this lecture, we mentioned two issues that composite design patterns are meant to address.
- This tree helps us understand how those goals are met. Individual types of objects, in particular, composite class objects, can aggregate component classes, which creates a tree-like structure.
- As well, each individual class is a subtype of an interface or superclass, and will be able to conform to a set of shared behaviors.

Composite Pattern (contd.)

- Expressing this in Java can be broken down into steps.
    1. Design the interface that defines the overall type.
    2. Implement the composite class.
    3. Implement the leaf class
- Let us examine each of these steps using a specific example.

Composite Pattern: Example



Composite Pattern: Example (contd.)

- Step 1: Design the interface that defines the overall type

```
public interface IStructure {
    public void enter();
    public void exit();
    public void location();
    public String getName();
}
```

Composite Pattern: Example (contd.)

- Step 2: Implement the composite class

```
public class Housing implements IStructure {
    private ArrayList<IStructure> structures;
    private String address;

    public Housing (String address) {
        this.structures = new ArrayList<IStructure>();
        this.address = address;
    }

    public void enter() {
        System.out.println("You have entered the " +
            this.address);
    }
    public void exit() {
        System.out.println("You have left the " +
            this.address);
    }
```

Composite Pattern: Example (contd.)

- Step 2: Implement the composite class (contd.)

```
//Housing (contd.)

    public void location() {
        System.out.println("You are currently in " +
            this.getName() + ". It has ");
        for (IStructure struct : this.structures)
            System.out.println(struct.getName());
    }

    public String getName() {
        return this.address;
    }

    public int addStructure(IStructure component) {
        this.structures.add(component);
        return this.structures.size() - 1;
    }
```

Composite Pattern: Example (contd.)

- Step 2: Implement the composite class (contd.)

```
//Housing (contd.)

    public int removeStructure(IStructure component) {
        this.structures.remove(component);
        return this.structures.size() - 1;
    }

    public IStructure getStructure(int componentNumber) {
        return this.structures.get(componentNumber);
    }
}
```

Composite Pattern: Example (contd.)

- Step 3: Implement the leaf class

```
public class Room implements IStructure {
    private String name;

    Room(String name) {
        this.name = name;
    }
    public void enter() {
        System.out.println("You have entered the " +
            this.name);
    }
    public void exit() {
        System.out.println("You have left the " + this.name);
    }
    public void location() {
        System.out.println("You are currently in the " +
            this.name);
    }
    public String getName() {
        return this.name;
    }
}
```

Composite Pattern: Example (contd.)

```
public class CompositeMain {

    public static void main(String[] args) {
        Housing building = new Housing("123 Street");

        Housing floor1 = new Housing("123 Street - First Floor");

        int firstFloor = building.addStructure(floor1);

        Room washroom1m = new Room("1F Men's Washroom");
        Room washroom1w = new Room("1F Women's Washroom");
        Room common1 = new Room("1F Common Area");

        int firstMens = floor1.addStructure(washroom1m);
        int firstWomans = floor1.addStructure(washroom1w);
        int firstCommon = floor1.addStructure(common1);

        building.enter(); // Enter the building
        building.location();
```

Composite Pattern: Example (contd.)

```
//CompositeMain (contd.)

        Housing currentfloor =
            (Housing)building.getStructure(firstFloor);
        currentfloor.enter(); // Walk into the first floor
        currentfloor.location();

        Room currentRoom = (Room)
            currentfloor.getStructure(firstMens);
        currentRoom.enter(); // Walk into the men's room
        currentRoom.exit(); // Exit from the men's room

        currentRoom = (Room)
            currentfloor.getStructure(firstCommon);
        currentRoom.enter(); // Walk into the common area
        currentRoom.exit(); // Exit from the common area
        currentfloor.exit(); // Exit from the first floor

        floor1.removeStructure(common1);
        floor1.removeStructure(firstMens);
        currentfloor.enter(); // Walk into the first floor
        currentfloor.location();
    }
```

Composite Pattern: Summary

- A composite design pattern allows you to build a tree-like structure of objects, and to treat individual types of those objects **uniformly**. This is achieved by:
  - Enforcing polymorphism across each class through implementing an interface (or inheriting from a superclass).
  - Using a technique called recursive composition which allows objects to be composed of other objects that are of a common type.
  - Composite design patterns apply the design principles of decomposition and generalization.
  - They break a whole into parts, but have the whole and parts both conform to a common type.
  - Complex structures can be built using composite objects and leaf objects which belong to a unified component type.