CS 2340 Objects and Design

Structural Patterns

Christopher Simpkins
chris.simpkins@gatech.edu

Structural Design Patterns

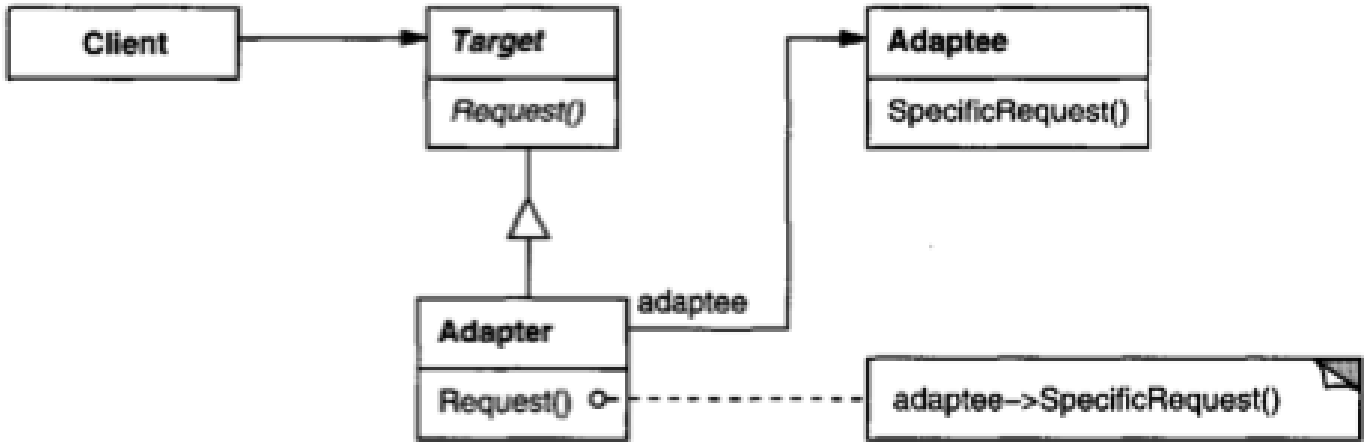
Concerned with how classes and objects are composed to form larger structures.

- Structural class patterns use inheritance to compose interfaces or implementations. (Adapter)
- Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class/interface composition. (Composite)

Adapter (A.K.A Wrapper)

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure

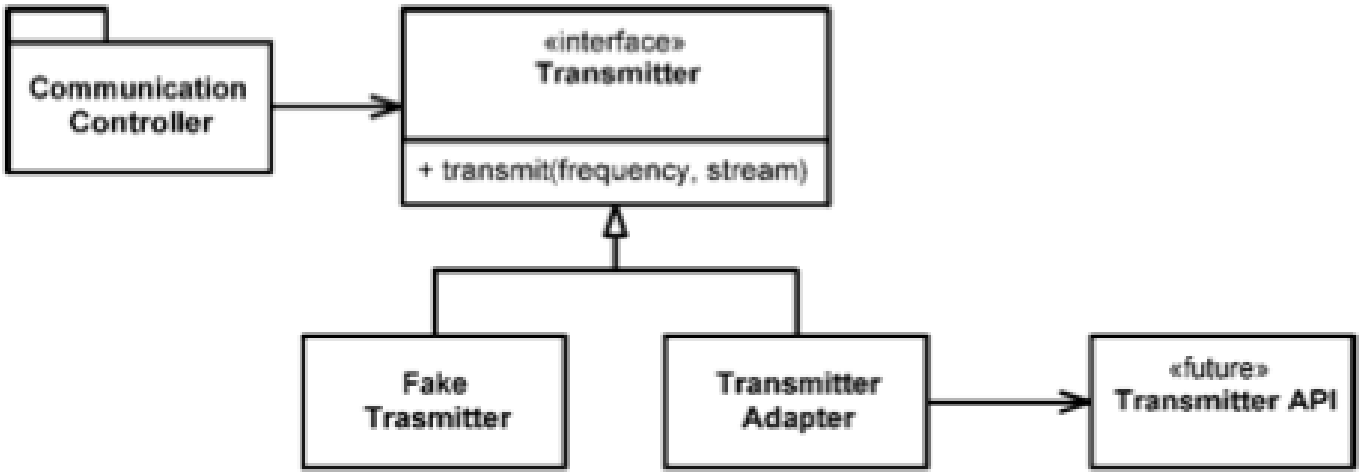


Participants

- **Target** defines the domain-specific interface that Client uses.
- **Client** collaborates with objects conforming to the Target interface.
- **Adaptee** defines an existing interface that needs adapting.
- **Adapter** adapts the interface of Adaptec to the Target interface.

Adapter Example

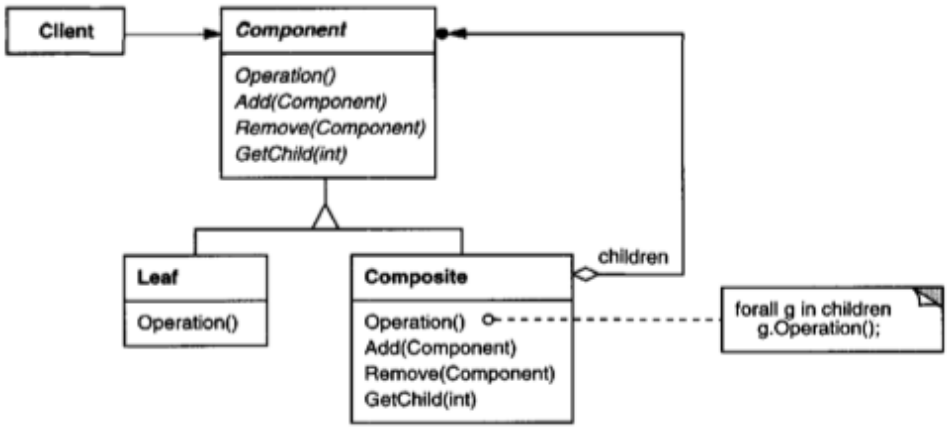
- Imagine we're a team writing an application that will use a hardware transmitter, but the transmitter's software is handled by another team that hasn't defined their software interface.
- We can define our own interface the way *we* want it to work.
- While we're waiting for the transmitter team, we create a fake implementation to work with.
- When the transmitter team finally gives us their interface, we can write an adapter to fit it to our interface.
- The rest of our code is unaffected.



Composite

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

Structure



Participants

- **Component** declares the interface for objects in the composition.
- **Leaf** represents leaf objects (objects with no children).
- **Composite** defines behavior for components having children; stores child components; implements child-related operations.
- **Client** manipulates objects in the composition through the Component interface.

Composite Example: Dive Log (1 of 3)

Say we have a dive log with individual dives. Both represent the concept of dive experience, so we can represent this concept abstractly which allows us to get reports of dive experience in a uniform way whether we have a single dive or a log of several dives:

```
public interface DiveExperience {

    public Date getDateTimeBegin();

    public Date getDateTimeEnd();

    public int getMaxDepthFeet();

    public int getBottomTimeMinutes();
}
```

This interface plays the **Component** role in the composite pattern.

Composite Example: Dive Log (2 of 3)

`Dive` plays the `Leaf` role:

```
public class Dive implements DiveExperience, Comparable<Dive> {
    private Date dateTimeBegin, dateTimeEnd;
    private int maxDepthFeet, bottomTimeMinutes;

    public Dive(Date dateTimeBegin, Date dateTimeEnd,
                int maxDepthFeet, int bottomTimeMinutes) {
        this.dateTimeBegin = dateTimeBegin;
        // ...
    }
    public Date getDateTimeBegin() {return dateTimeBegin; }
    public Date getDateTimeEnd() { return dateTimeEnd; }
    public int getMaxDepthFeet() { return maxDepthFeet; }
    public int getBottomTimeMinutes() { return bottomTimeMinutes; }

    public int compareTo(Dive other) {
        return this.getDateTimeBegin().
                compareTo(other.getDateTimeBegin());
    }
    public boolean equals(Object other) { .. }

    public int hashCode() { ... }
}
```

Composite Example: Dive Log (3 of 3)

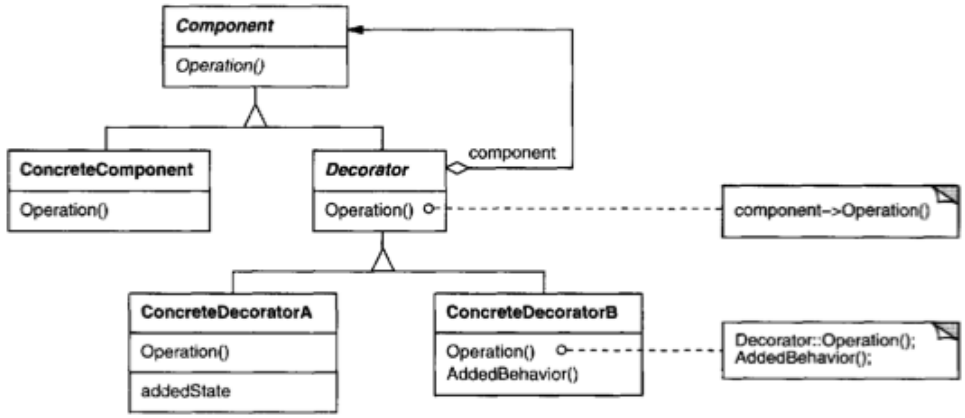
And `DiveLog` plays the **Composite** role:

```
public class DiveLog implements DiveExperience {
    private TreeSet<Dive> dives = new TreeSet<Dive>();
    private int maxDepthFeet = 0;
    public void add(Dive dive) {
        dives.add(dive);
        if (dive.getMaxDepthFeet() > maxDepthFeet)
            maxDepthFeet = dive.getMaxDepthFeet();
    }
    public Date getDateTimeBegin() {
      return dives.first().getDateTimeBegin();
    }
    public Date getDateTimeEnd() {
        return dives.last().getDateTimeEnd();
    }
    public int getMaxDepthFeet() { return maxDepthFeet; }
    public int getBottomTimeMinutes() {
        int sum = 0;
        for (Dive dive: dives) {
            sum += dive.getBottomTimeMinutes();
        }
        return sum;
    }
```

Decorator

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Structure



Participants

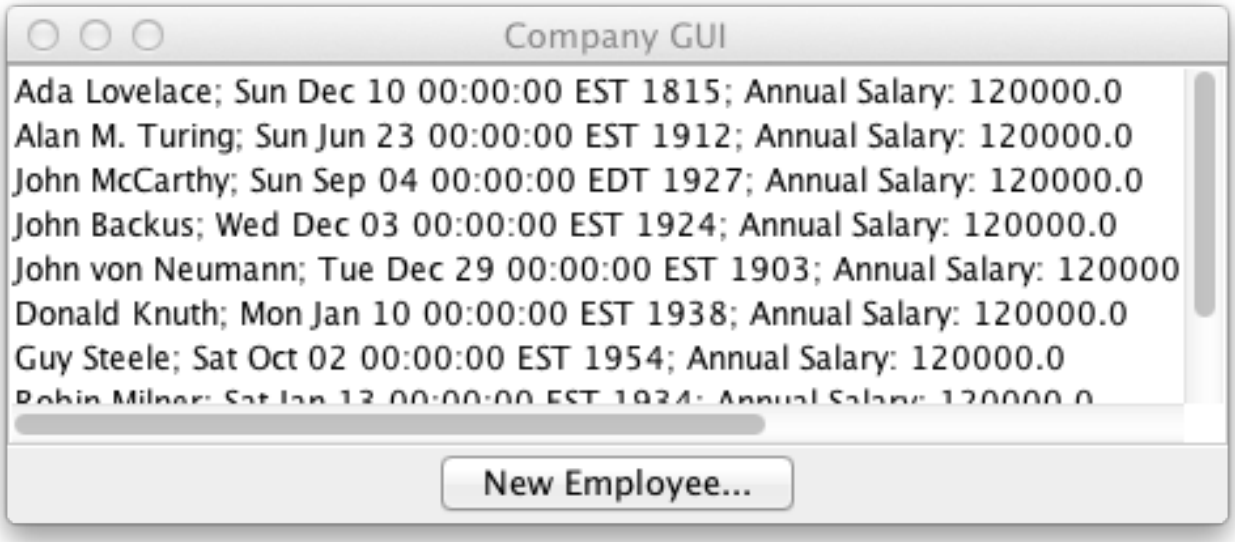
- **Component** defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent** defines an object to which additional responsibilities can be attached.
- **Decorator** maintains a reference to a Component object and defines an interface that conforms to Component's interface.

Decorator Example: `JScrollPane`

The Swing I ibrary provides a scrollbar decorator called `JScrollPane`. Using it is easy:

```
add(new JScrollPane(new JList(...)));
```

By simply wrapping our `JList` in a `JScrollPane` the list will show horizontal and vertical scroll bars as needed.



We've *extended* the functionality of a `JList` without having to *subclass* it.