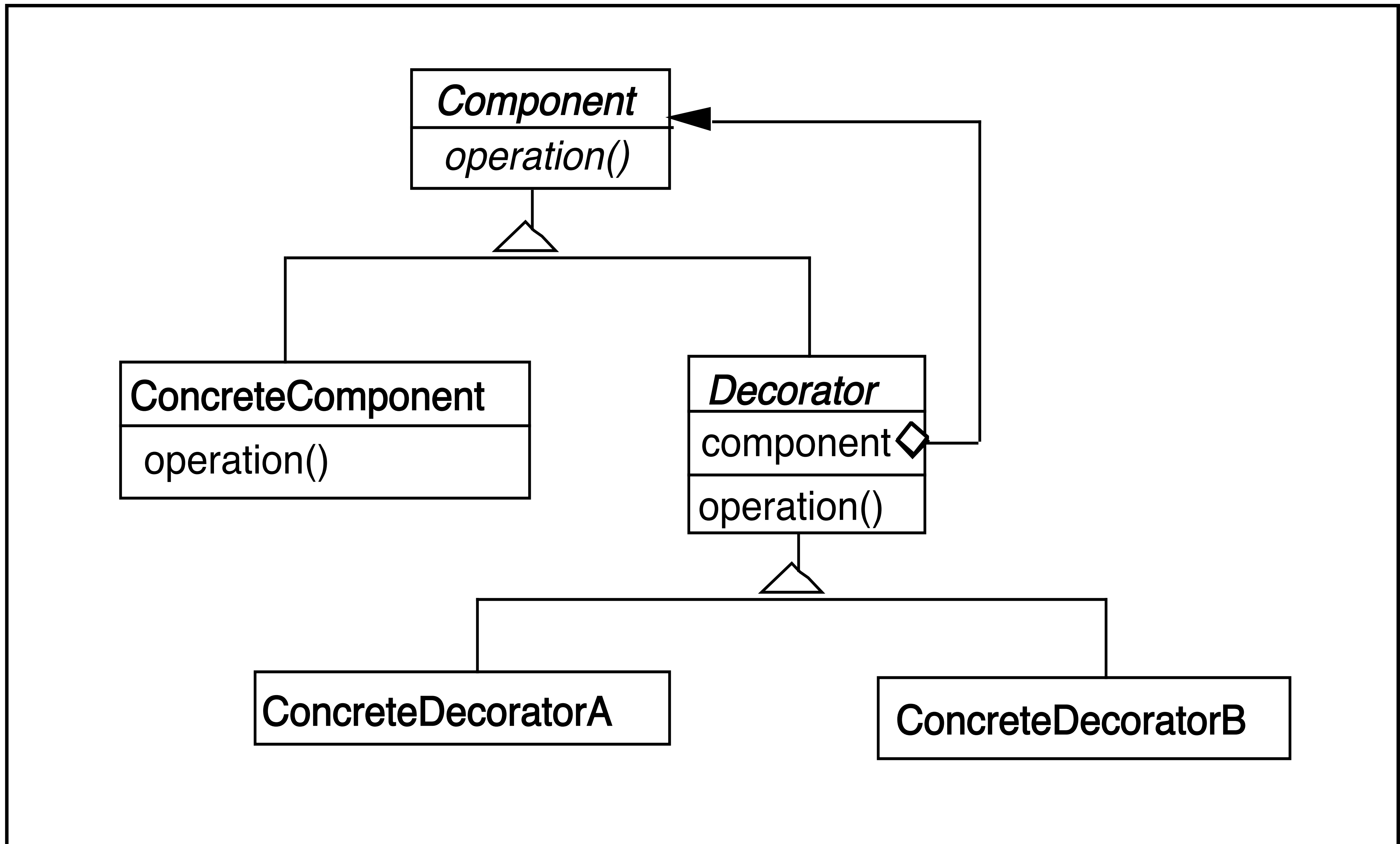
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 06 Decorator,Null Object
Sep 10, 2020

Copyright ©, All rights reserved. 2020 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.

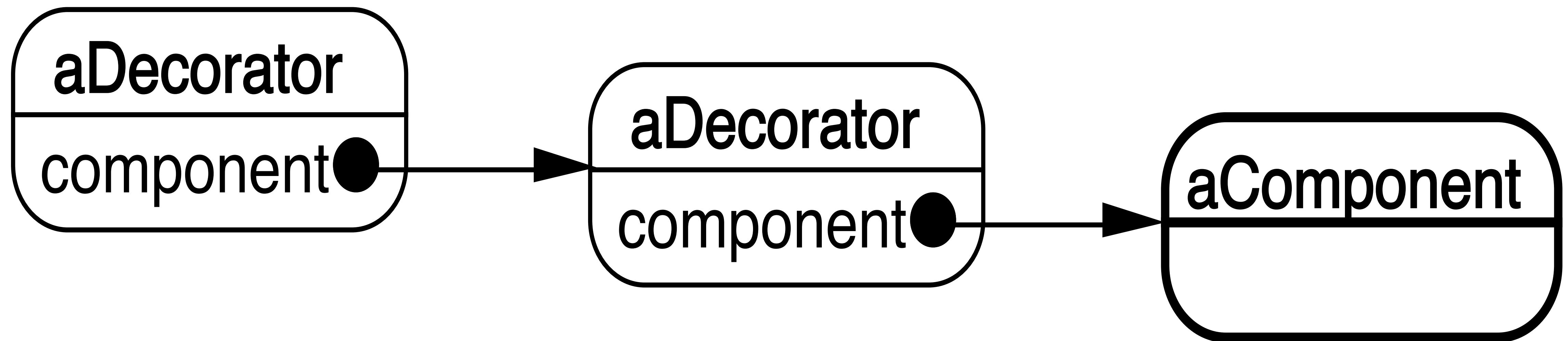
Decorator Pattern

Adds responsibilities to individual objects

- Dynamically
- Transparently


Component
operation()
ConcreteComponent
operation()
Decorator
component
operation()
ConcreteDecoratorA
ConcreteDecoratorB

Decorator forwards all component operations



Coffee Example from Wikipedia

Compute the cost of coffee

- Base Price $1
- Cost of Milk $0.50
- Cost of Sprinkles $0.20

```
public interface Coffee {
    public double getCost();
    public String getIngredients();
}

public class SimpleCoffee implements Coffee {
    public double getCost() {  return 1;  }

    public String getIngredients() { return "Coffee"; }
}
```

Abstract Decorator

```
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

Milk

```
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() {
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}
```

```java
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

Simple Examples

Good for explaining basic concept

But

- Prices change
- New extras
- Seasonal extras

System driven by data

Download information to cash register

Simple Example & Functional Programming

Just compose functions

with-sprinkles(with-milk(base-price(amount)))

(with-sprinkles(with-milk(base-price amount)))

(def sprinkles-milk (**comp** with-sprinkles with-milk base-price))

(sprinkles-milk amount)

Functional Programming

```
(defn sprinkles-milk2
    [amount]
    (-> amount
        base-price
        with-milk
        with-sprinkles
        compute-tax))
```

Simple Example & Functional Programming

Use maps

prices

```
{:large 4.0
 :medium 3.5
 :small 3.0
 :milk 0.5
 :sprinkles 0.2 }
```

order

```
{:size :medium
 :extras [:milk :sprinkles]}
```

```
(compute-cost prices order)
```

Favor Composition over Inheritance

Benefits & Liabilities

Benefits

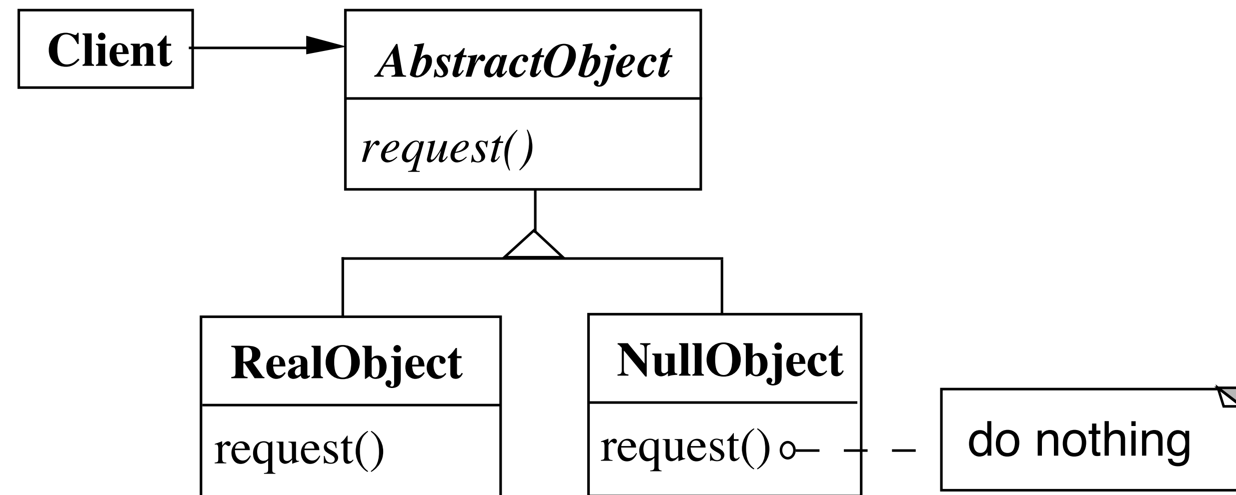
Simplifies a class
Distinguishes a classes core responsibilities from embellishments

Liabilities

Changes the object identity of a decorated object
Code harder to understand and debug
Combinations of decorators may not work correctly together

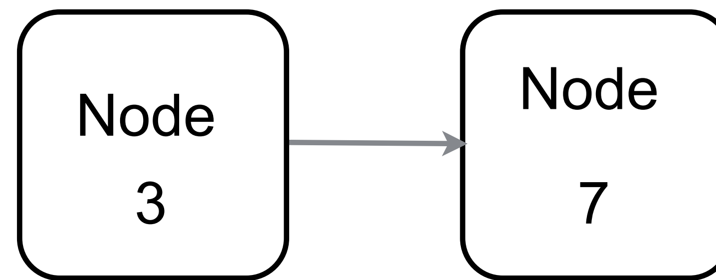
Null Object

Null Object



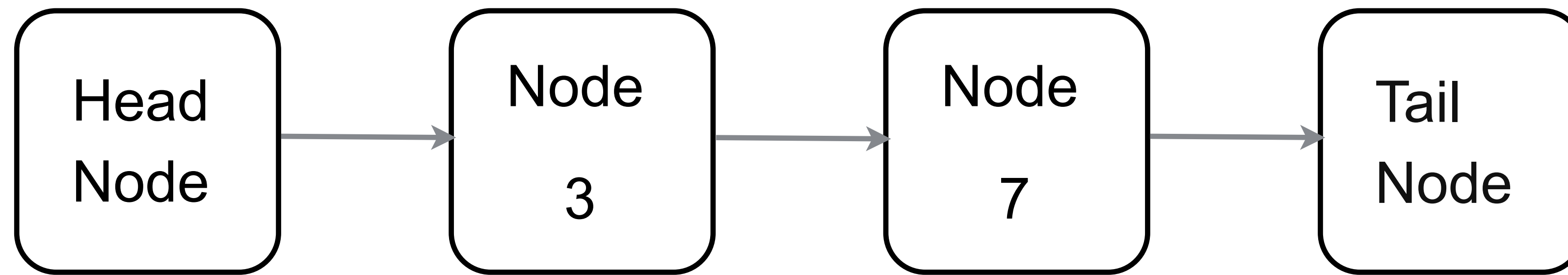
NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing




```
class LinkedList {
  Node head;

  public toString() {
    if (head == nil) {
      return “()”;
    }
    String listAsString = “(;
    Node current = head;
    while (current != null) {
      listAsString += current.value() + “, “;
      current = current.next;
    }
    listAsString = removetail(listsString, 2);
    return listAsString + “)”;
  }
```




```
class LinkedList {
  Node head;

  public toString() {
    return head.toString();
  }
```

```
class Node {
  public String toString() {
    return " " + element + next.toString();
  }
}
```

```
class HeadNode {
  public String toString() {
    return "(" + next.toString();
  }
}
```

```
class TailNode {
  public String toString() {
    return " )";
  }
}
```

Applicability - When to use Null Objects

Some collaborator instances should do nothing

You want clients to ignore the difference between a collaborator that does something and one that does nothing

- Client does not have to explicitly check for null or some other special value

You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way

Applicability -When not to use Null Objects

Very little code actually uses the variable directly

The code that does use the variable is well encapsulated

The code that uses the variable can easily decide how to handle the null case and will always handle it the same way

Consequences

Advantages

Uses polymorphic classes

Simplifies client code

Encapsulates do nothing behavior

Makes do nothing behavior reusable

Disadvantages

Forces encapsulation

- Makes it difficult to distribute or mix into the behavior of several collaborating objects

May cause class explosion

Forces uniformity

Is non-mutable

Implementation

- Too Many classes
- Multiple Do-nothing meanings
  - Try Adapter pattern
- Transformation to RealObject
  - Try Proxy pattern

Refactoring: Introduce Null Object

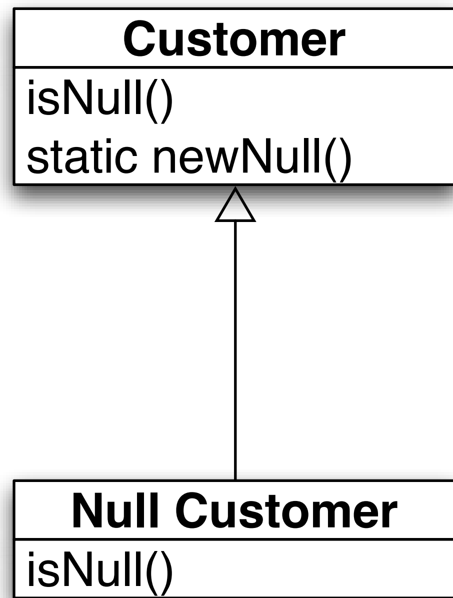
You have repeated checks for a null value

Replace the null value with a null object

```
if (customer == null)
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

Create Null Subclass




```
public boolean isNull() { return false;}
public static Customer newNull() { return new NullCustomer();}
```

```
boolean isNull() { return true;}
```

Compile

Replace all nulls with null object

```
class SomeClassThatReturnCustomers {

  public Customer getCustomer() {
    if (_customer == null )
      return Customer.newNull();
    else
      return _customer;
  }
  etc.
}
```

Compile

Replace all null checks with isNull()

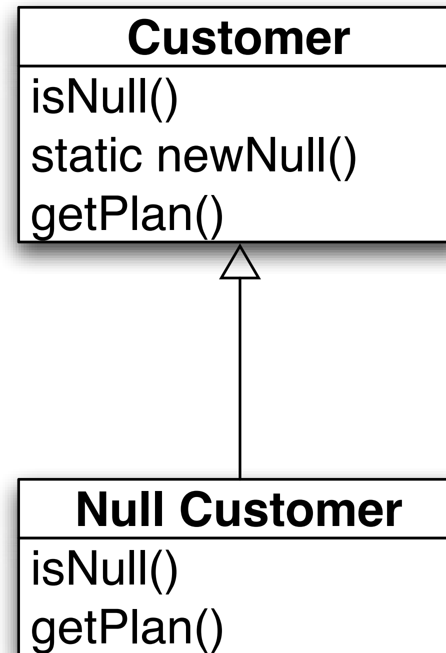
```
if (customer == null)
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```

```
if (customer.isNull())
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```

Compile and test

Find an operation clients invoke if not null

```
if (customer.isNull())
  plan = BillingPlan.basic();
else
  plan = customer.getPlan();
```




```
class NullCustomer {
  public BillingPlan getPlan() {
    return BillingPlan.basic();
  }
```

Remove the Condition Check

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

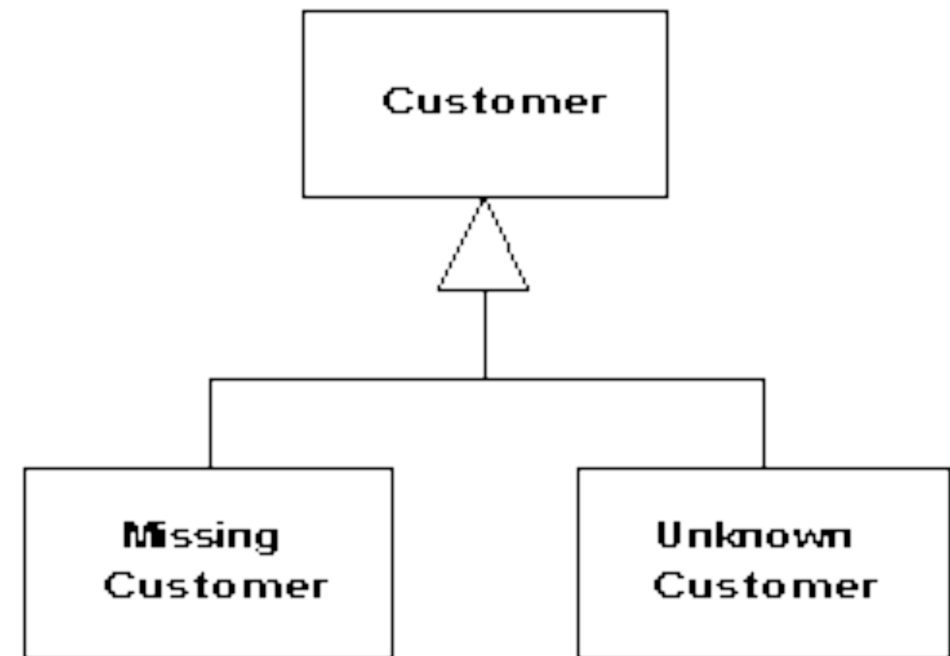
```
plan = customer.getPlan();
```

Compile & Test

Repeat last two slides for each operation
clients check if null

Special Case

Special Case



Represent special cases by a subclass

Use when multiple places that have same behavior

After conditional check for particular class instance

Or same behavior after a null check