

CS 130 SOFTWARE ENGINEERING

DESIGN PATTERNS

TEMPLATE METHOD, STATE
FLYWEIGHT

Professor Miryung Kim

UCLA Computer Science

Based on Materials from Miryung Kim,
Christine Julien and Adnan Aziz

All copyrights are reserved by Professor Kim and UCLA.

AGENDA

- ▶ Template Method
- ▶ State
- ▶ Flyweight
- ▶ Move onto the lecture on Build Management

RECAP

- ▶ Strategy

- ▶ Encapsulate a family of algorithms and allow to vary them

- ▶ Factory Method:

- Separate creation of varying products from other shared uses

- ▶ Abstract Factory

- Create varying families of products

- Separate creation of varying families from other shared uses

- Reduce duplication of parallel product classes

RECAP

- ▶ Singleton ensures a single object creation.
- ▶ Command decouples a receiver object's actions from invokers.
- ▶ Adaptor adapts legacy code to a target interface.
- ▶ Façade simplifies complex interfaces of multiple subsystems.



TEMPLATE METHOD

Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

- (1) Boil some water
- (2) Brew coffee in boiling water
- (3) Pour coffee in cup
- (4) Add sugar and milk

Starbuzz Tea Recipe

- (1) Boil some water
- (2) Steep tea in boiling water
- (3) Pour tea in cup
- (4) Add lemon

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

← The recipe for coffee looks a lot like the recipe for tea, doesn't it?
←

MOTIVATION FOR TEMPLATE METHOD

- ▶ Varying objects want to do similar things but slightly different ways
- ▶ The overall work flow is similar
- ▶ Certain steps are identical but certain steps differ

NOW LET'S TAKE A LOOK AT SKELETON CODE

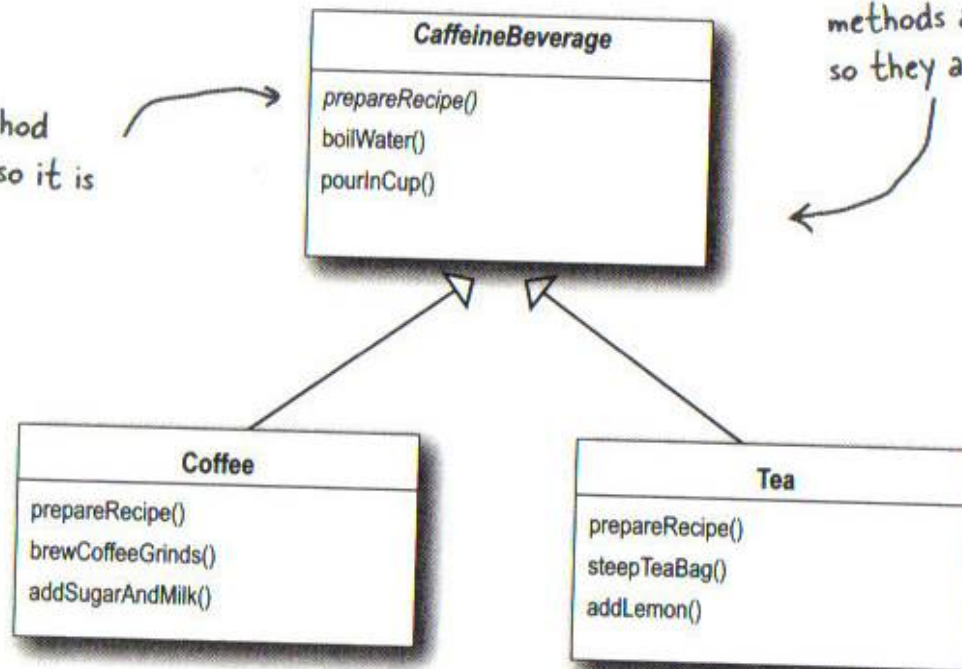
► [Handout-DesignPatternSkeletons](#)



The `boilWater()` and `pourInCup()` methods are shared by both subclasses, so they are defined in the superclass.

The `prepareRecipe()` method differs in each subclass, so it is defined as abstract.

Each subclass implements its own recipe.



Each subclass overrides the `prepareRecipe()` method and implements its own recipe.

The methods specific to `Coffee` and `Tea` stay in the subclasses.

ABSTRACTED RECIPE METHOD

```
void prepareRecipe() {  
    boilWater();  
    brew();  
    pourInCup();  
    addCondiments();  
}
```

ABSTRACT BASE CLASS

```
public abstract class CaffeineBeverage {  
    final void prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
    abstract void brew();  
    abstract void addCondiments();  
    void boilWater() {  
        System.out.println("Boiling water");  
    }  
    void pourInCup() {  
        System.out.println("Pouring into cup");  
    }  
}
```

COFFEE AND TEA IN TERMS OF THE ABSTRACT CAFFEINE CLASS

```
public class Coffee extends CaffeineBeverage {  
    public void brew() {  
        System.out.println("Dripping Coffee through filter");  
    }  
    public void addCondiments() {  
        System.out.println("Adding Sugar and Milk");  
    }  
}
```

```
public class Tea extends CaffeineBeverage {  
    public void brew() {  
        System.out.println("Steeping the tea");  
    }  
    public void addCondiments() {  
        System.out.println("Adding Lemon");  
    }  
}
```

TEMPLATE METHOD PATTERN DEFINED

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

THINK PAIR SHARE (HAND OUT)

(A) The following is a code snippet using Swing frame. Where can you find the use of a template method pattern?

(B) The following program is a sorting program for Ducks. Where can you find the use of a template method pattern?

THINK PAIR SHARE

- ▶ What is the difference between Template Method and Strategy?

TEMPLATE METHOD VS. STRATEGY

- ▶ Template Method defines the outline of algorithms and let subclasses do some of the work.
- ▶ Template Method keeps control over the algorithm's structure.
- ▶ Template Method puts all duplicated code into a super class and subclasses share the common code.
- ▶ Strategy defines a family of algorithms and make them interchangeable.
- ▶ Strategy does not control the algorithm's structure.
- ▶ Strategy uses composition so it's more flexible to switch with a new strategy.



STATE

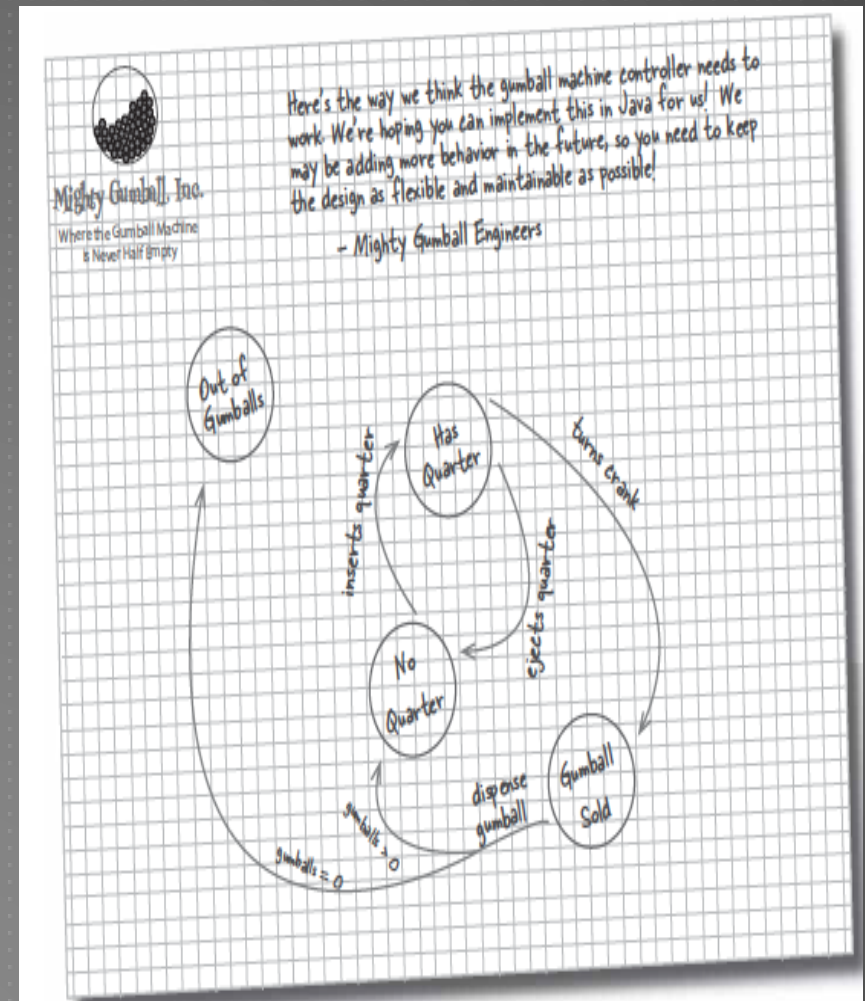
EXAMPLE: GUMBALL MACHINE

Gumball Machine inc. needs a JAVA application
for controlling Gumball Machine

Gumball Machine requirements are
represented as a State Transition diagram

STATE TRANSITION REQUIREMENTS

- ▶ States :
- ▶ No Quarter
- ▶ Has Quarter
- ▶ Gumball Sold
- ▶ Out Of Gumballs
- ▶



TRANSFORMING REQUIREMENTS INTO CODE

Team transform the state diagram using following steps:

Step 1: Gather the states

Step 2: Create an instance variable to hold the current state & define values for each state

Step 3: Gather up all the actions that can happen in the system

Step 4: Create a class that acts as state machine & map each action to a method

NAÏVE DESIGN

```
public class GumballMachine {  
  
    final static int SOLD_OUT = 0;  
    final static int NO_QUARTER = 1;  
    final static int HAS_QUARTER = 2;  
    final static int SOLD = 3;  
  
    int state = SOLD_OUT;  
    int count = 0;
```

NAÏVE DESIGN

```
public GumballMachine(int count) {  
    this.count = count;  
    if (count > 0) {  
        state = NO_QUARTER;  
    }  
}  
  
public void insertQuarter() {  
    if (state == HAS_QUARTER) {  
        System.out.println("You can't insert another quarter");  
    } else if (state == NO_QUARTER) {  
        state = HAS_QUARTER;  
        System.out.println("You inserted a quarter");  
    } else if (state == SOLD_OUT) {  
        System.out.println("You can't insert a quarter, the machine is s  
    } else if (state == SOLD) {  
        System.out.println("Please wait, we're already giving you a gumball");  
    }  
}
```

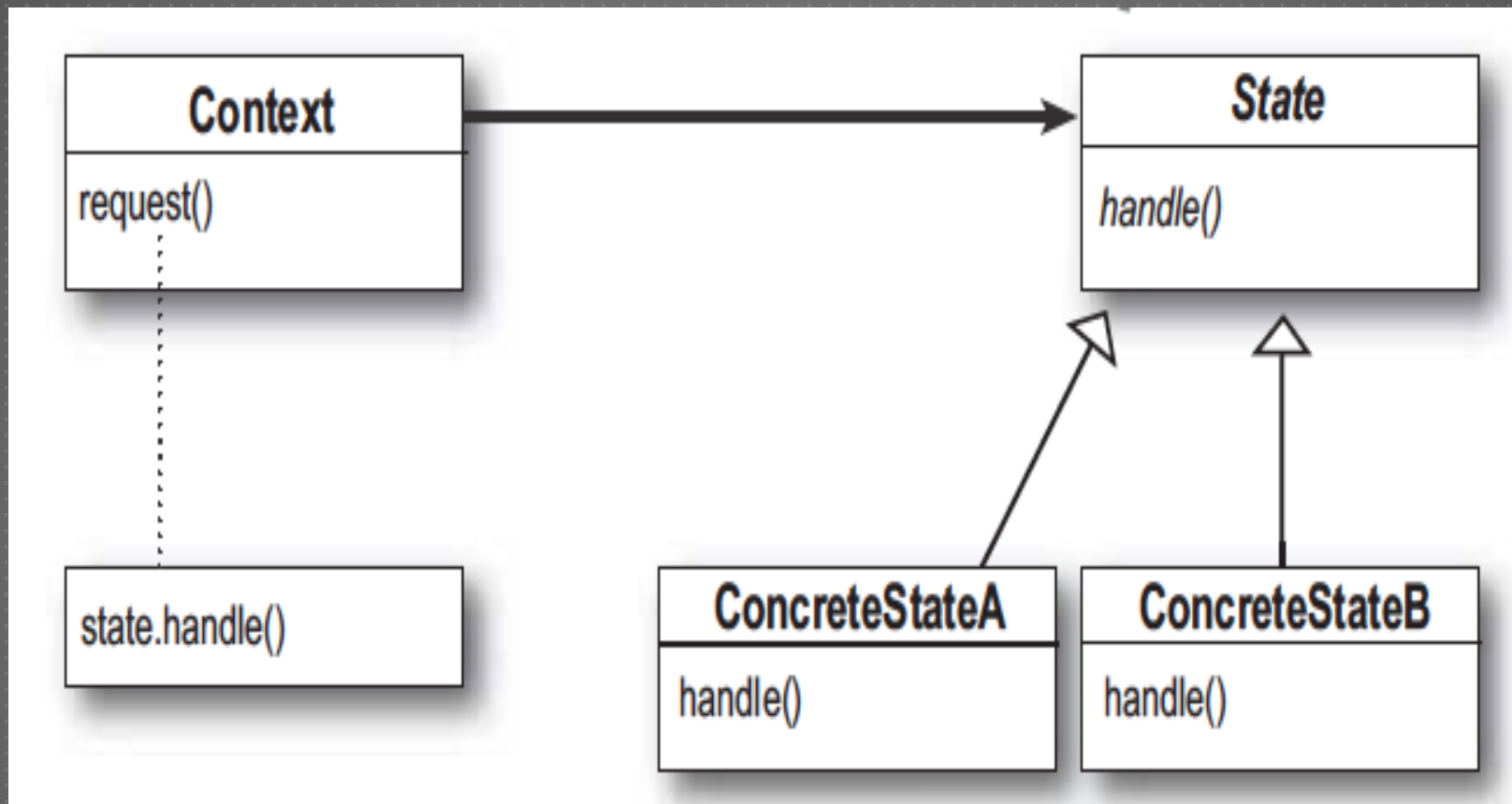

MOTIVATION FOR STATE

- ▶ You'd like to code a complex state transition
- ▶ You'd like to not forget “defining state transitions” for each state and check this at compile time!

STATE PATTERN MECHANICS

- ▶ Encapsulate the varying behaviour
- ▶ Create a new class for every State
- ▶ Localize the behavior for each State
- ▶ Adding new State => adding a new class
- ▶ Handling new requirements is easy

CLASS DIAGRAM



```
public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State state = soldOutState;
    int count = 0;
    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }
    public void insertQuarter() {
        state.insertQuarter();
    }
    public void ejectQuarter() {
        state.ejectQuarter();
    }
    public void turnCrank() {
        state.turnCrank();
    }
}
```



```
public class SoldState implements State {
    GumballMachine gumballMachine;
    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }
    public void insertQuarter() {
        System.out.println("Please wait, we're already giving you a gumball");
    }
    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank");
    }
    public void turnCrank() {
        System.out.println("Turning twice doesn't get you another gumball!");
    }
    public void dispense() {
        gumballMachine.releaseBall();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
    public String toString() {
        return "dispensing a gumball";
    }
}
```

IMPLEMENTING STATE PATTERN

- ▶ Define a State interface
- ▶ Implement a State class for every state of the machine
- ▶ Eliminate the conditional code by delegating the work to the State class

PROS AND CONS

- ▶ Encapsulates all the behavior of a State in one object
- ▶ Helps avoiding inconsistent States Changes
- ▶ Make the code bulky and increases the number of objects
- ▶ Class Explosion problem
- ▶ Brittle Interfaces

THE STATE PATTERN

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

NOW LET'S TAKE A LOOK AT SKELETON CODE

▶ [Handout-DesignPatternSkeletons](#)

THINK-SHARE-PAIR QUESTION 1

How is the Strategy Pattern different from the State Pattern?

THINK-SHARE-PAIR QUESTION 1

- ▶ They share similarity in declaring a field state and delegating work to the field. State is a specific case of using strategy, where actions have consequences on the machine's state.
- ▶ Strategy Pattern is a flexible alternative to subclassing, with strategy you can change the behavior by composing with different object.
- ▶ State Pattern is an alternative to putting lots of conditions; by encapsulating the behavior within state objects, you can simply change the state object in context to change its behavior

MATCH EACH PATTERN WITH ITS DESCRIPTION

1. State

2. Strategy

3. Template Method

- A. Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
- B. Subclasses decide how to implement steps in an algorithm
- C. Encapsulate state-based behavior and delegate behavior to the current state

QUESTION 2

Transform the TV Remote Application using the
State Design Pattern

QUESTION 3

1. Identify the class the Context class
2. Identify the State Interface
3. Identify the ConcreteState classes



FLYWEIGHT PATTERN

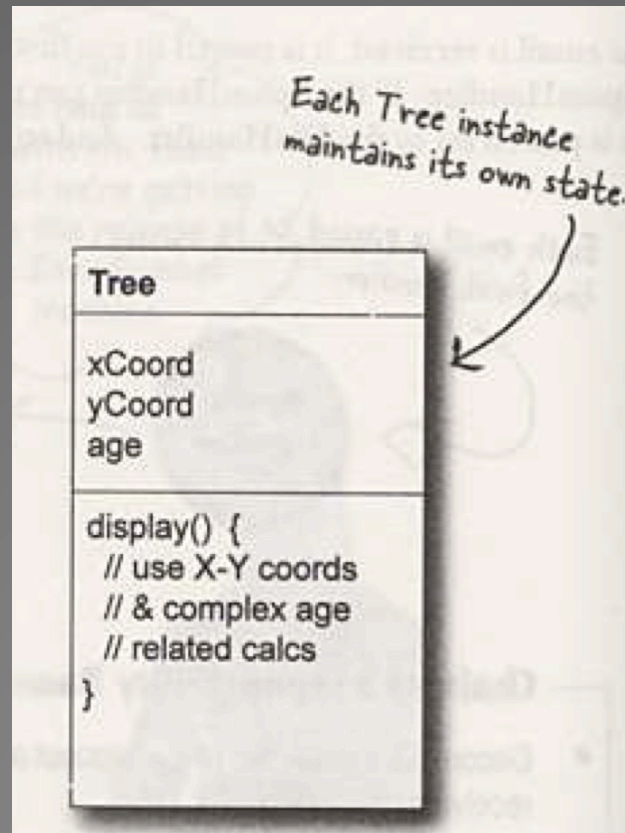
THE SCENARIO

- ▶ You want to add trees to a landscape design application
 - ▶ Trees are pretty dumb; they have x-y locations, they can draw themselves dynamically (depending on an age attribute)
 - ▶ A user might want to add lots and lots of trees
- ▶ A user of your application is complaining that, when he creates large groves of trees, the app gets sluggish
 - ▶ Yikes! There are THOUSANDS of tree objects!
- ▶ When you have tons of objects that are basically the same
 - ▶ Create a single instance of the object
 - ▶ A single client object that manages the state of all of those objects

MOTIVATION FOR FLYWEIGHT

- ▶ Save resources by sharing objects when one instance can be used to provide many virtual instances

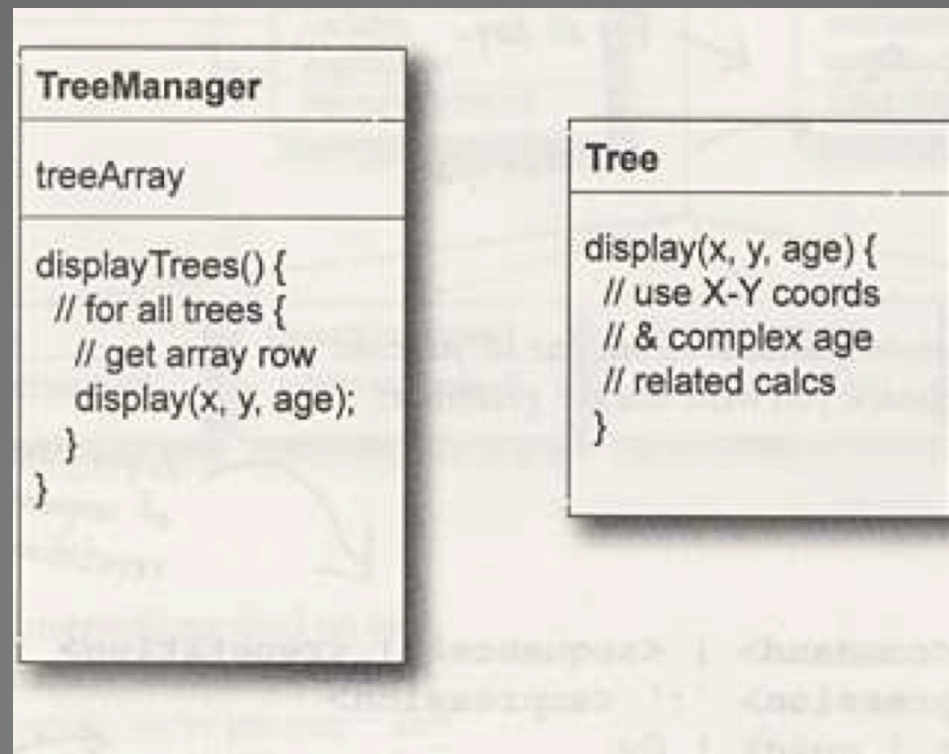
WITHOUT FLYWEIGHT



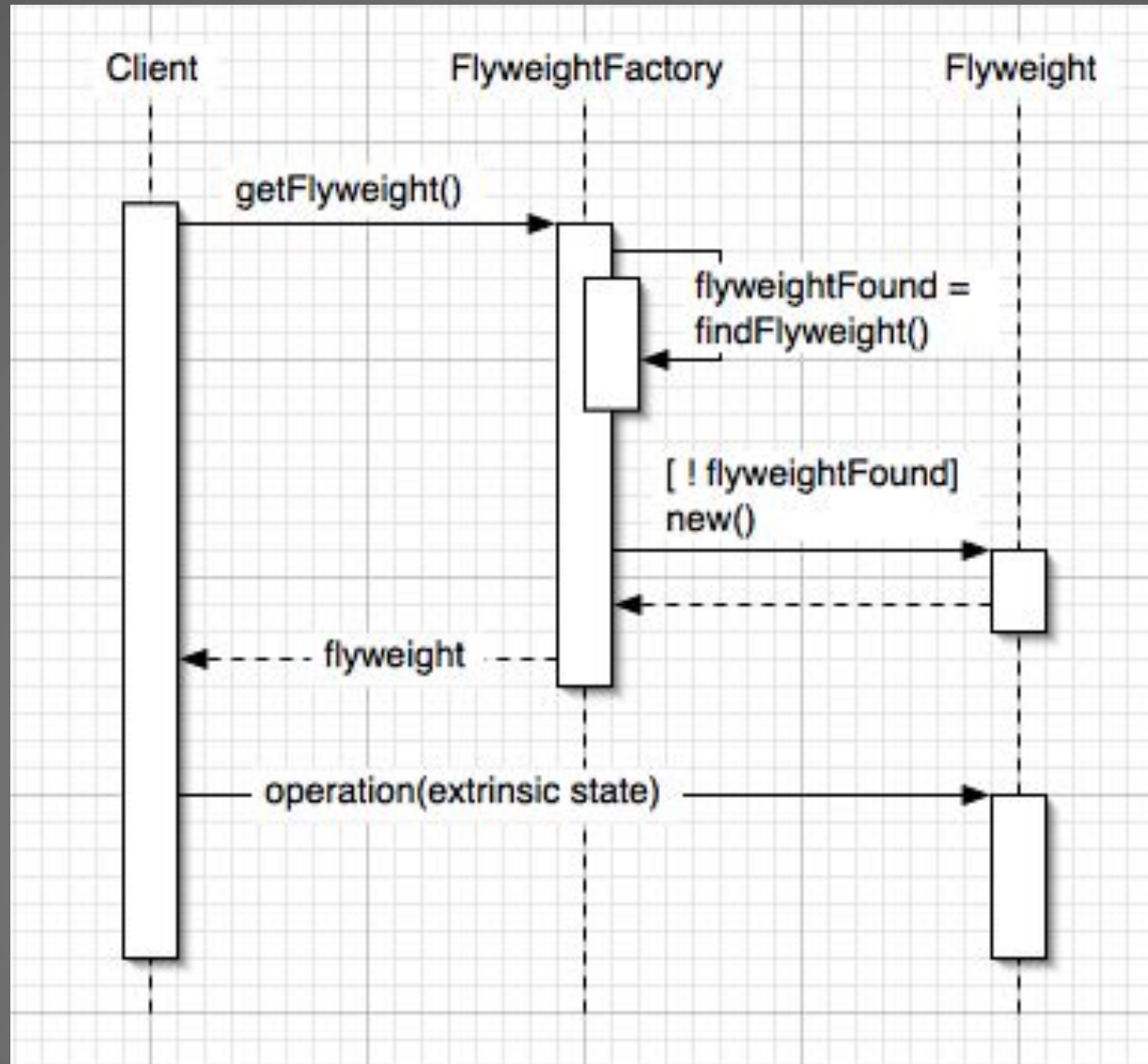
WITH FLYWEIGHT



- ▶ Have only one instance of Tree
- ▶ A client object maintains the state of All trees.



THE FLYWEIGHT SEQUENCE DIAGRAM



ANOTHER EXAMPLE

```
class ColorBox extends Canvas implements Runnable {  
    private int    pause;  
    private Color  curColor = getColor();  
    private static Color[] colors = { Color.black, Color.blue, Color.cyan,  
        Color.darkGray, Color.gray, Color.green, Color.lightGray, Color.red,  
        Color.magenta, Color.orange, Color.pink, Color.white, Color.yellow };  
  
    public ColorBox( int p ) {  
        pause = p;  
        new Thread( this ).start();  
    }  
    private static Color getColor() {  
        return colors[ (int)(Math.random() * 1000) % colors.length ];  
    }
```



```
public void run() {  
    while (true) {  
        curColor = getColor();  
        repaint();  
        try { Thread.sleep( pause ); } catch( InterruptedException e ) { }  
    } }  
public void paint( Graphics g ) {  
    g.setColor( curColor );  
    g.fillRect( 0,0,getWidth(),getHeight() );  
} }  
  
public class ColorBoxes {  
    public static void main( String[] args ) {  
        int size = 8, pause = 10;  
        if (args.length > 0) size = Integer.parseInt( args[0] );  
        if (args.length > 1) pause = Integer.parseInt( args[1] );  
        Frame f = new FrameClose( "ColorBoxes - 1 thread per ColorBox" );  
        f.setLayout( new GridLayout( size, size ) );  
        for (int i=0; i < size*size; i++) f.add( new ColorBox( pause ) );  
        f.setSize( 500,400 );  
        f.setVisible( true );  
    } }  
}
```


PROS AND CONS

- ▶ Reduces the number of object instances at runtime
- ▶ Centralizes state for many virtual objects into a single location
- ▶ Many instances must be controlled similarly
- ▶ Once you implemented Flyweight, a single logical instance will not be able to behave independently from other instances

THE FLYWEIGHT PATTERN

The Flyweight Pattern allows one instance of a class can be reused to provide many virtual instances.

RECAP

- ▶ Template Method allows to set a common workflow where sub steps may vary.
- ▶ State allows to encode complex state transitions.
- ▶ Flyweight allows to share common resources by separating usage contexts from used objects.

LET'S SKETCH SKELETON CODE OF OBSERVER

DIFFERENCE BETWEEN OBSERVER AND MEDIATOR

- ▶ Both are communication abstractions
- ▶ Observer 1:n relationship between subject and observers.
- ▶ It's easy to add a new observer in Observer.
- ▶ Mediator abstracts n:n relationship among multiple objects.
- ▶ Every object only knows about Mediator and trusts Mediator to call other objects that I want to communicate.

LET'S CONSIDER MEDIATOR EXAMPLE

```
class Mediator {  
    private boolean slotFull = false;  
    private int number;  
    public synchronized void storeMessage(  
        int num ) {  
        // no room for another message  
        while (slotFull == true) {  
            try {  
                wait();  
            }  
            catch (InterruptedException e ) { }  
        }  
        slotFull = true;  
        number = num;  
        notifyAll();  
    }  
}
```

```
    public synchronized int  
    retrieveMessage() {  
        // no message to retrieve  
        while (slotFull == false)  
            try {  
                wait();  
            }  
            catch (InterruptedException e ) { }  
        slotFull = false;  
        notifyAll();  
        return number;  
    }  
}
```

```
class Consumer extends Thread {  
  private Mediator med;  
  private int id;  
  private static int num = 1;  
  public Consumer( Mediator m ) {  
    med = m;  
    id = num++;  
  }  
  public void run() {  
    while (true) {  
      System.out.print("c" + id + "-" + med.retrieveMessage() + " ");  
    }  
  }  
}
```

```
class Producer extends Thread {  
    // 2. Producers are coupled only to the Mediator  
    private Mediator med;  
    private int id;  
    private static int num = 1;  
    public Producer( Mediator m ) {  
        med = m;  
        id = num++;  
    }  
    public void run() {  
        int num;  
        while (true) {  
            med.storeMessage( num = (int)(Math.random()*100) );  
            System.out.print( "p" + id + "-" + num + " " );  
        }  
    }  
}
```



```
class MediatorDemo {  
  public static void main( String[] args ) {  
    Mediator mb = new Mediator();  
    new Producer( mb ).start();  
    new Producer( mb ).start();  
    new Consumer( mb ).start();  
    new Consumer( mb ).start();  
    new Consumer( mb ).start();  
    new Consumer( mb ).start();  
  }  
}
```

QUESTIONS?