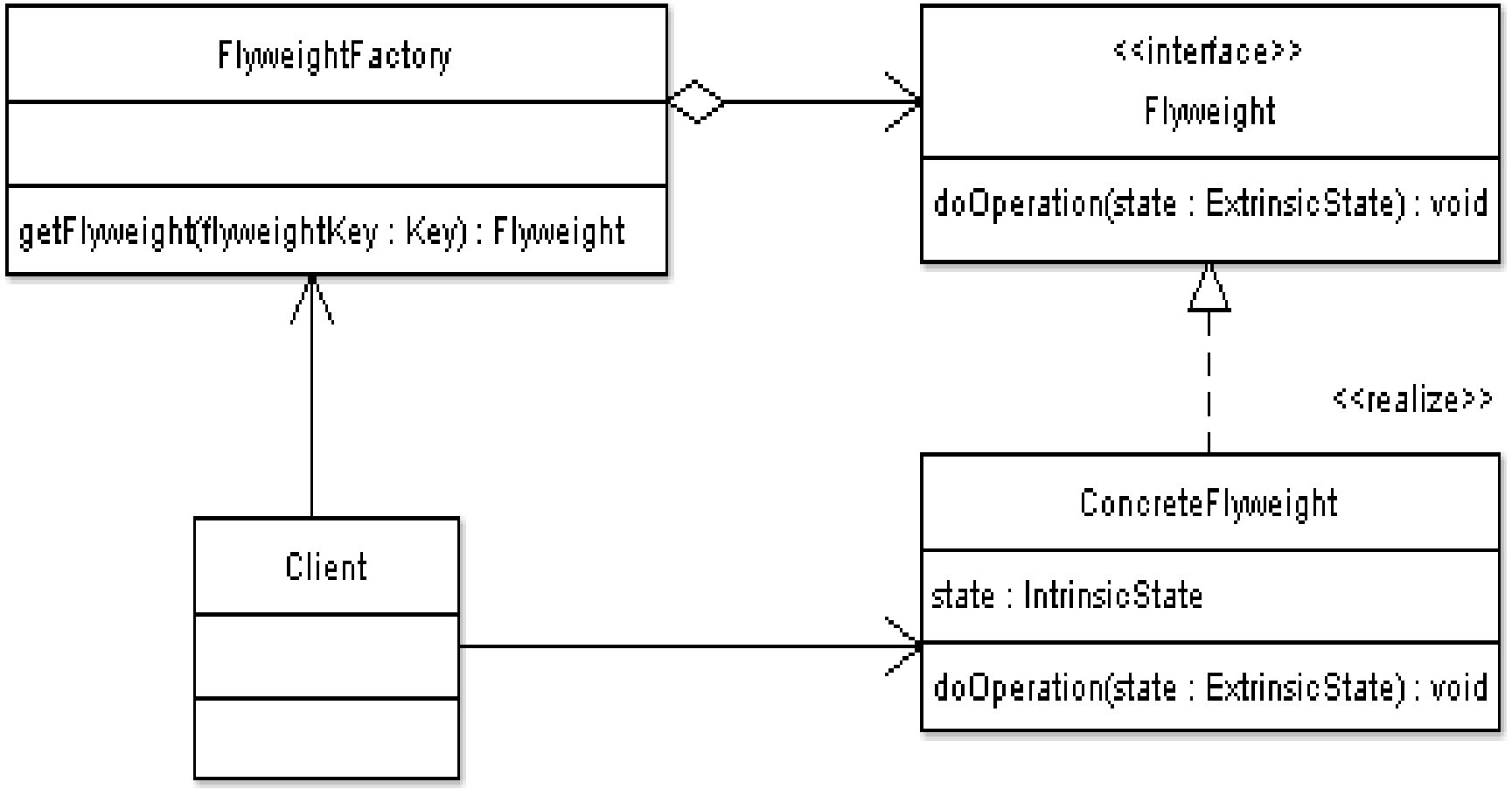
Flyweight

- Creational pattern that describes how to support a large number of fine grained objects efficiently, by sharing commonalities in state
- Minimizes memory use by sharing as much data as possible with other similar objects
- If parts of object **state** can be shared, you can put them in external data structures and pass them to the flyweight objects temporarily (when the flyweight objects are used)

Flyweight

- You can greatly reduce memory requirements by not requiring that 'heavy-weight' objects be created in large numbers when dealing with systems that contain many things that are **mostly** the same.
- Yes, Flyweight is named after the lightest general category in boxing. It lightens the way a set of related objects is represented in memory via shared state

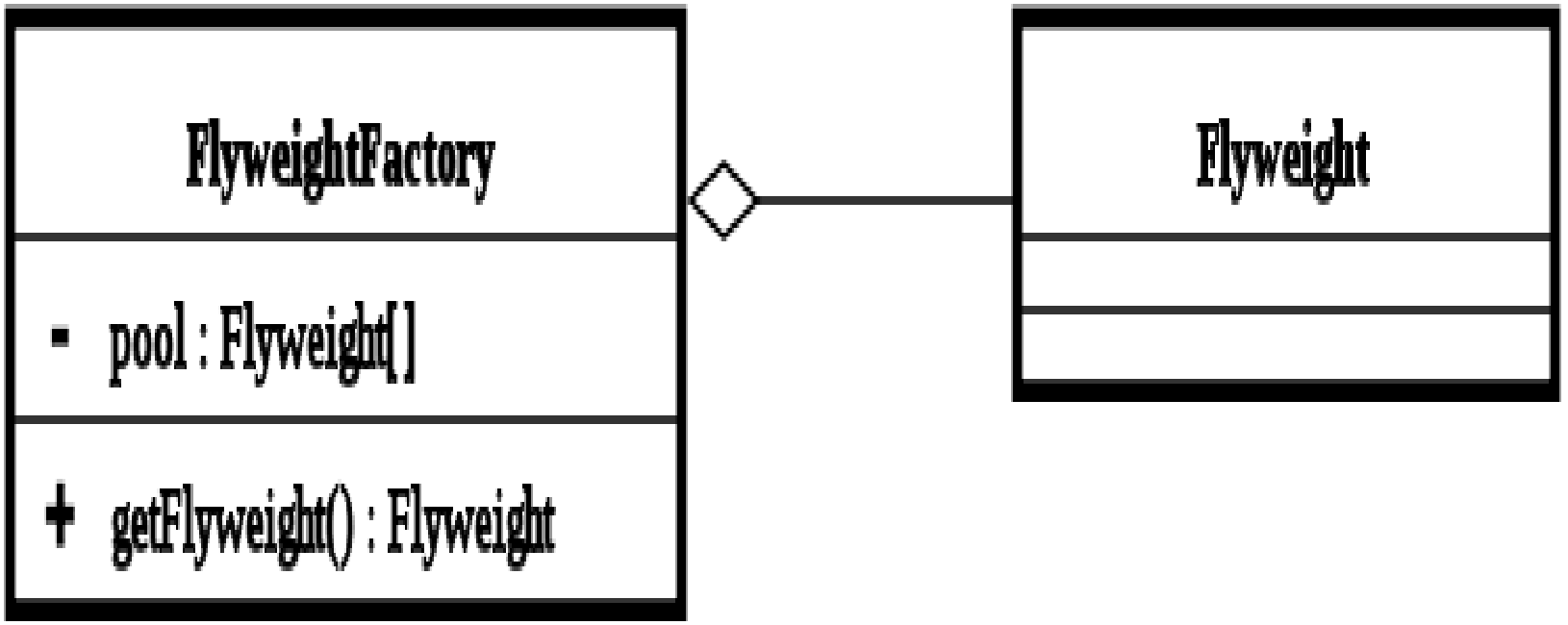
UML


FlyweightFactory
getFlyweight(flyweightKey : Key) : Flyweight
<<interface>>
Flyweight
doOperation(state : ExtrinsicState) : void
<<realize>>
ConcreteFlyweight
state : IntrinsicState
doOperation(state : ExtrinsicState) : void
Client

Participants in UML Diagram

- Flyweight: interface through which flyweights can receive and act on **extrinsic** state
- ConcreteFlyweight:
  - stores intrinsic state
  - must be sharable
  - must be able to manipulate state that is extrinsic
- FlyweightFactory:
  - creates and manages the Flyweight objects
  - maintains a pool of objects and returns necessary object if it exists, otherwise it creates a new object, adds it to the pool, then returns it
- Client: maintains references to flyweights in addition to computing and maintaining extrinsic state

UML 2


FlyweightFactory
- pool : Flyweight[ ]
+ getFlyweight() : Flyweight
Flyweight

Code Example (Wikipedia)

```
// Flyweight object
public interface CoffeeOrder {
    public void serveCoffee(CoffeeOrderContext
context);
}
```

ConcreteFlyweight

```
// ConcreteFlyweight object that creates ConcreteFlyweight
public class CoffeeFlavor implements CoffeeOrder {
    private String flavor;

    public CoffeeFlavor(String newFlavor) {
        flavor = newFlavor;
    }

    public String getFlavor() {
        return this.flavor;
    }

    public void serveCoffee(CoffeeOrderContext context) {
        System.out.println("Serving Coffee flavor " + flavor + " to table
number " + context.getTable());
    }
}
```

Context for Flyweight

```
public class CoffeeOrderContext {
   private int tableNumber;

   public CoffeeOrderContext(int tableNumber) {
      this.tableNumber = tableNumber;
   }

   public int getTable() {
      return this.tableNumber;
   }
}
```

Factory for Flyweight

```
import java.util.HashMap;
import java.util.Map;

//FlyweightFactory object
public class CoffeeFlavorFactory {
    private Map<String, CoffeeFlavor> flavors = new HashMap<String,
CoffeeFlavor>();

    public CoffeeFlavor getCoffeeFlavor(String flavorName) {
        CoffeeFlavor flavor = flavors.get(flavorName);
        if (flavor == null) {
            flavor = new CoffeeFlavor(flavorName);
            flavors.put(flavorName, flavor);
        }
        return flavor;
    }

    public int getTotalCoffeeFlavorsMade() {
        return flavors.size();
    }
}
```

Test the Flyweight

```
public class TestFlyweight {
   /** The flavors ordered. */
   private static CoffeeFlavor[] flavors = new
CoffeeFlavor[100];
   /** The tables for the orders. */
   private static CoffeeOrderContext[] tables = new
CoffeeOrderContext[100];
   private static int ordersMade = 0;
   private static CoffeeFlavorFactory flavorFactory;

   public static void takeOrders(String flavorIn, int table) {
      flavors[ordersMade] =
flavorFactory.getCoffeeFlavor(flavorIn);
      tables[ordersMade++] = new CoffeeOrderContext(table);
   }
```

Test the Flyweight

```
public static void main(String[] args) {
        flavorFactory = new CoffeeFlavorFactory();

        takeOrders("Cappuccino", 2);
        takeOrders("Cappuccino", 2);
        takeOrders("Frappe", 1);
        takeOrders("Frappe", 1);
        takeOrders("Xpresso", 1);
        takeOrders("Frappe", 897);
        takeOrders("Cappuccino", 97);
        takeOrders("Cappuccino", 97);
        takeOrders("Frappe", 3);
        takeOrders("Xpresso", 3);
        takeOrders("Cappuccino", 3);
        takeOrders("Xpresso", 96);
        takeOrders("Frappe", 552);
        takeOrders("Cappuccino", 121);
        takeOrders("Xpresso", 121);

        for (int i = 0; i < ordersMade; ++i) {
            flavors[i].serveCoffee(tables[i]);       }
        System.out.println(" ");
        System.out.println("total CoffeeFlavor objects made: " +
flavorFactory.getTotalCoffeeFlavorsMade());   }}
```

Example Usage

- When designing a word processor application, you might create an object for each character typed.
- Each Character object might contain information such as the font face, size and weight of each character. The problem here is that a lengthy document might contain tens of thousands of characters, and objects - quite a memory killer!
- The Flyweight pattern addresses the problem by creating a new object to store such information, which is shared by all characters with the same formatting. So, if I had a ten-thousand word document, with 800 characters in Bold Times-New-Roman, these 800 characters would contain a reference to a flyweight object that stores their common formatting information.
- The key here is that you only store the information once, so memory consumption is greatly reduced.

Uses

- Flyweight is often combined with Composite to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes
- State and Strategy objects are many times best represented as flyweights
- Often times are created using a Factory of some form and a Singleton is applied so that for each type of category of flyweights, a single instance is returned
- Any time you have numerous objects that are essentially the same, and thus can have shared state, are candidates for Flyweight. Games with massive numbers of creatures can really use this to great advantage.