



Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern
University



10. Flyweight Pattern



Intent

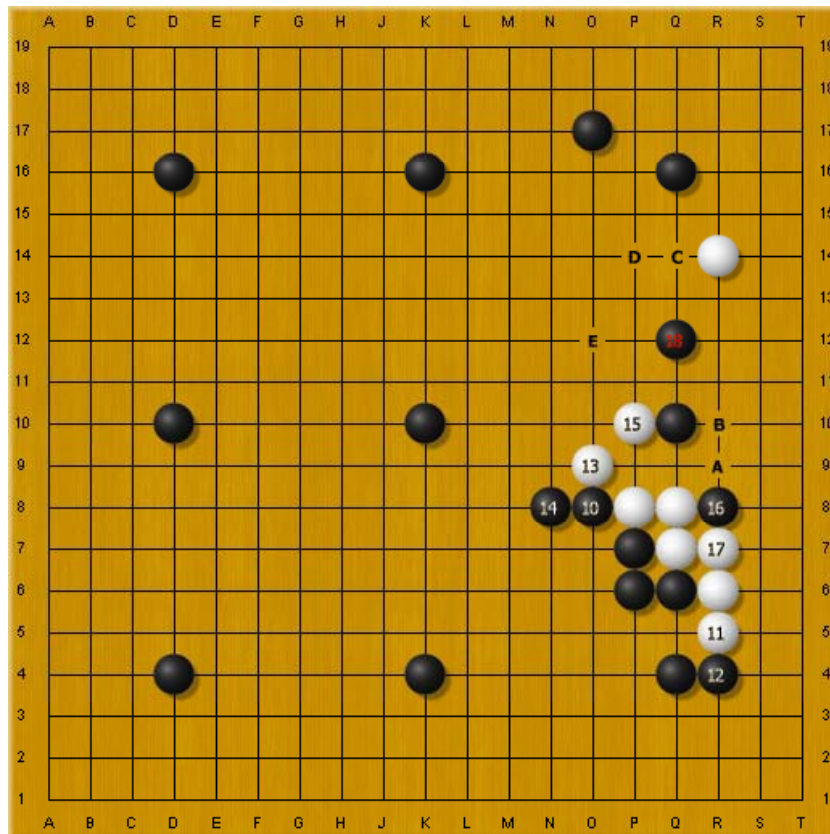
- Use sharing to support large numbers of **fine-grained** objects efficiently.
 - 享元模式以共享的方式高效地支持大量的细粒度对象。
-




Intent

- The shared flyweight is separated the **extrinsic state** from **intrinsic state**:
 - **Intrinsic state** is stored in the flyweight, sharable, not changed according to the context.
 - **Extrinsic state** is stored outside the flyweight, unsharable, changed according to the context.
 - For sharing the instances, **extrinsic state** of the **flyweight** (object) is stored by the clients, outside the **flyweight**. **Extrinsic state** is passed when **flyweight** need it.
 - In one word, **extrinsic state** of **flyweight** is independent from **flyweight**.
-


Example: Go Game



- Shared flyweight
 - Stone
(game piece)
- Intrinsic state
 - Black and white.
- Extrinsic state
 - Point
(position of Stone)



```
interface Stone {  
    public boolean isBlack();  
    public boolean isWhite();  
    public void place(int x, int y);  
}  
  
class GoStone implements Stone {  
    private boolean black;  
    public GoStone(boolean isBlack) {  
        this.black = isBlack;  
    }  
    public boolean isBlack() {  
        return black;  
    }  
    public boolean isWhite() {  
        return !black;  
    }  
    public void place(int x, int y){  
    }  
}
```

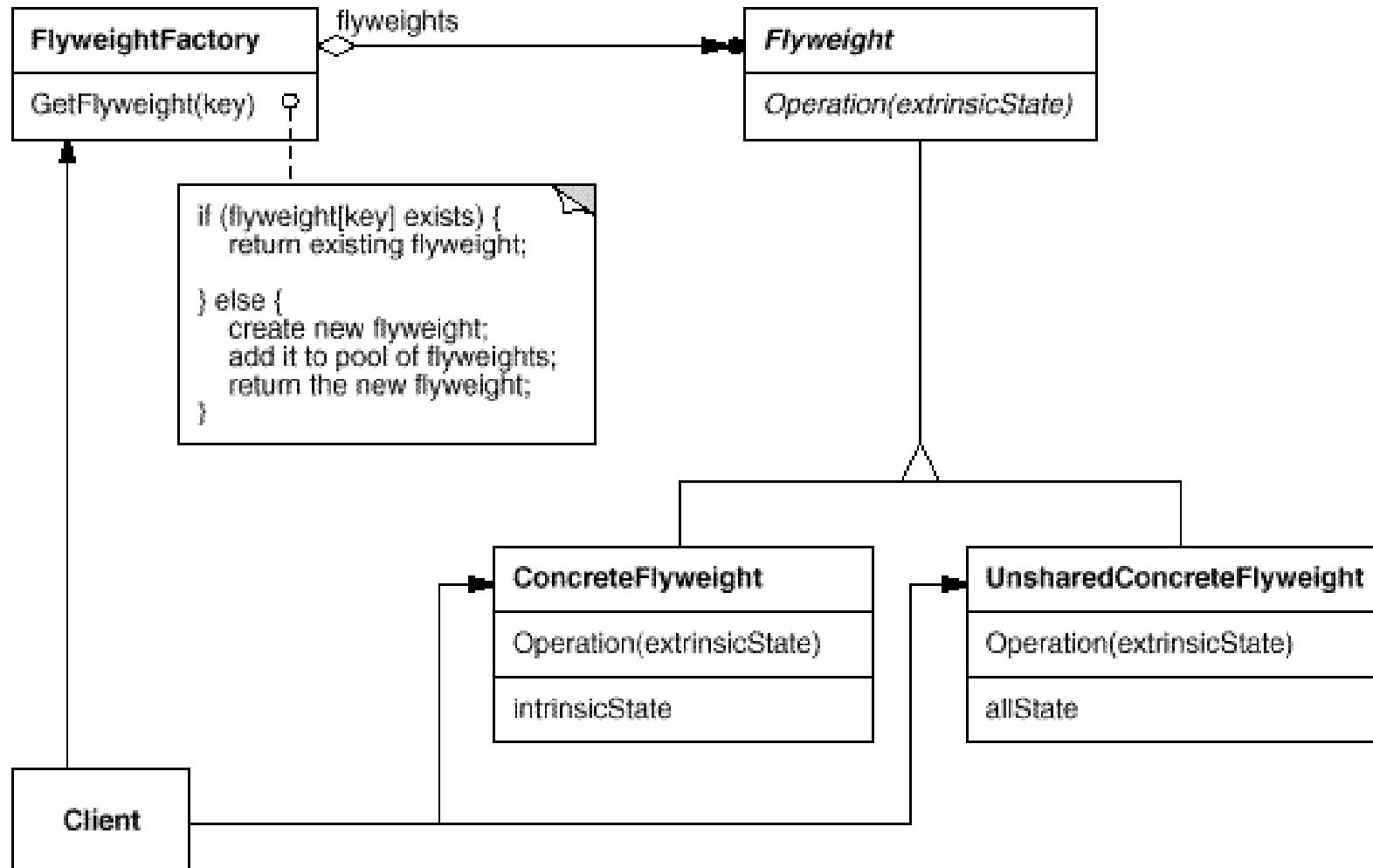


```
class StoneFactory{
    private static Stone white = new GoStone(false);
    private static Stone black = new GoStone(true);

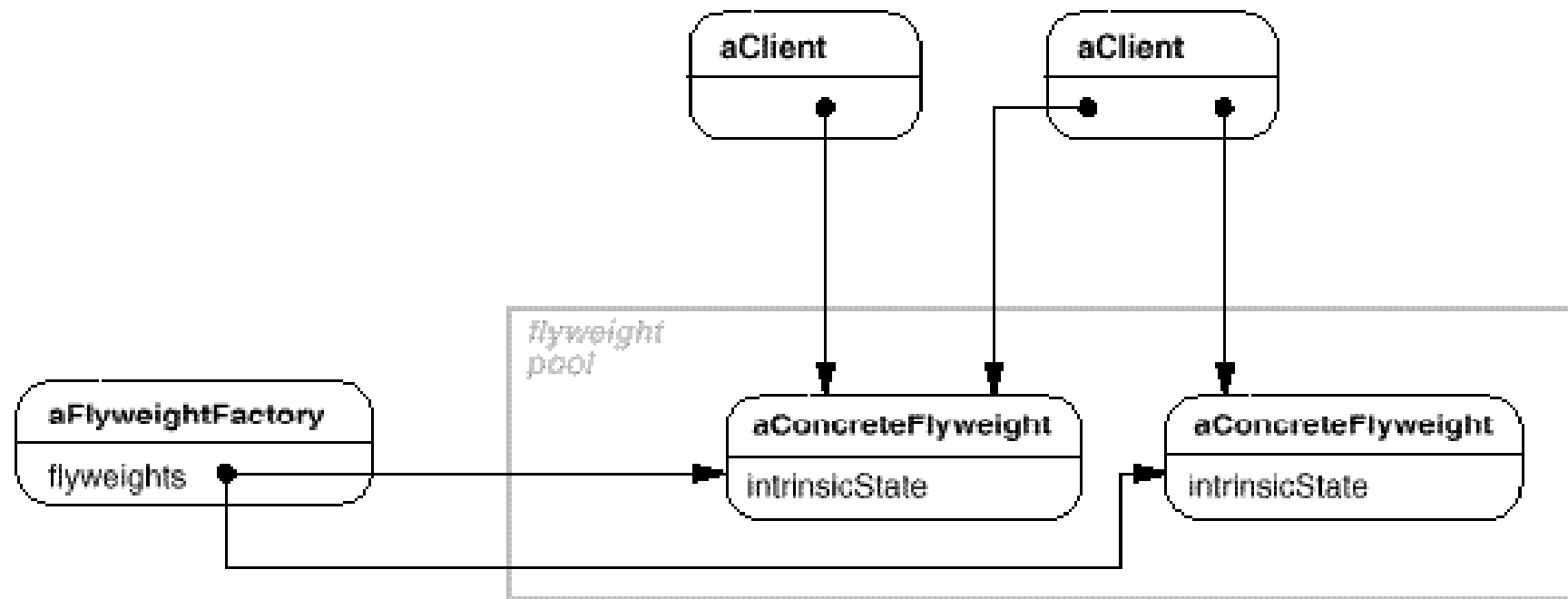
    public static Stone getWhite(){
        return white;
    }
    public static Stone getBlack(){
        return black;
    }
}

class FlyweightClient {
    public static void main(String[] args) {
        StoneFactory.getBlack().place(9, 9);
        StoneFactory.getWhite().place(9, 10);
        StoneFactory.getBlack().place(1, 1);
        StoneFactory.getWhite().place(9, 8);
    }
}
```

Structure



Structure





Participants

■ Flyweight:

- Declares an interface through which flyweights can receive and act on extrinsic state.

■ ConcreteFlyweight

- Implements the Flyweight interface and adds storage for intrinsic state, if any.
- A ConcreteFlyweight object must be sharable.
- Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.

■ UnsharedConcreteFlyweight

- Not all Flyweight subclasses need to be shared.
 - The Flyweight interface enables sharing; it doesn't enforce it.
-



Participants

■ FlyweightFactory

- Creates and manages **flyweight** objects, and ensures that **flyweights** are shared properly.
- When a client requests a **flyweight**, the **FlyweightFactory** object supplies an existing instance or creates one, if none exists.

■ Client

- Maintains a reference to **flyweight(s)**.
 - Computes or stores the extrinsic state of **flyweight(s)**.
-




Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic.
 - Intrinsic state is stored in the **ConcreteFlyweight** object;
 - Extrinsic state is stored or computed by **Client** objects. Clients pass this state to the flyweight when they invoke its operations.
 - Clients should not instantiate **ConcreteFlyweights** directly. Clients must obtain **ConcreteFlyweight** objects exclusively from the **FlyweightFactory** object to ensure they are shared properly.
-



Consequences

- Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.
-



Consequences -Storage savings

- Storage savings are a function of several factors:
 - The reduction in the total number of instances that comes from sharing
 - The amount of intrinsic state per object
 - Whether extrinsic state is computed or stored.
 - The more flyweights are shared, the greater the storage savings.
 - The greatest savings occur when the extrinsic state can be computed rather than stored. Then you save on storage in two ways: Sharing reduces the cost of intrinsic state, and you trade extrinsic state for computation time.
-



Applicability

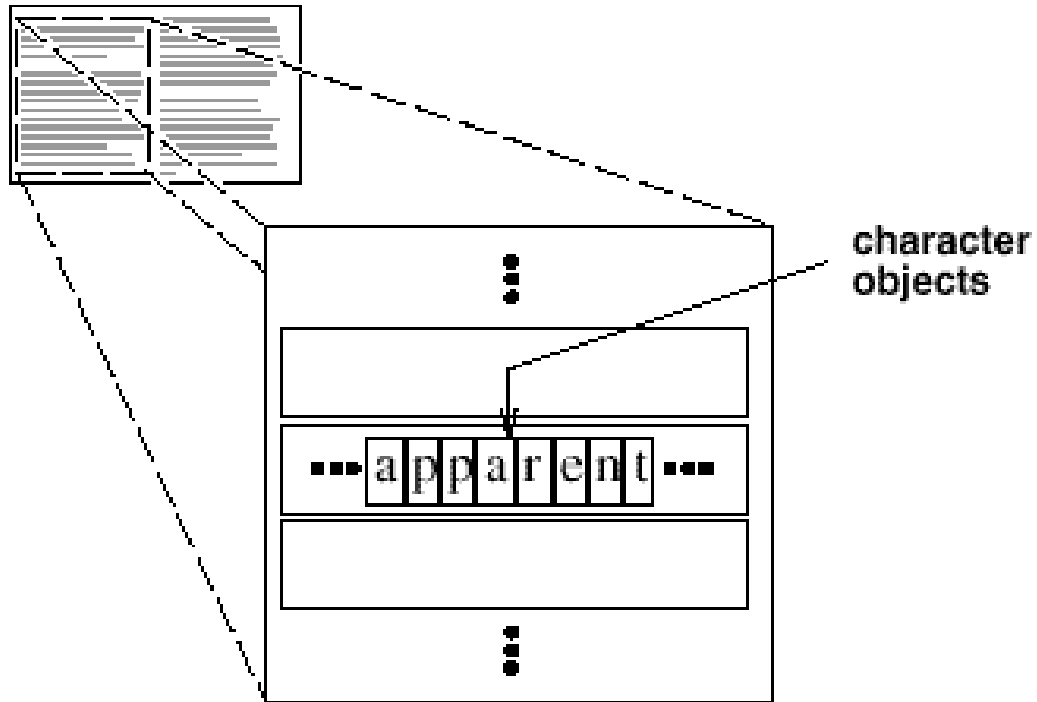
- An application uses a large number of objects.
 - Storage costs are high because of the larger quantity of objects.
 - Unsharable states could be extrinsic.
 - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
 - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.
-



Implementation 1: Removing extrinsic state

- The pattern's applicability is determined largely by **how easy it is to identify extrinsic state and remove it from shared objects.**
-

Example




■ Characters

■ Font

□ Color


□ Size

□ Space




Implementation 2: How to removing extrinsic state

- (Best) The clients store and pass the extrinsic state to the flyweight by parameters; OR
 - Extrinsic state should not be tiny.
 - (Better) Remove the extrinsic state and associated behaviors (codes) from the flyweight **to clients** together; OR
 - Encapsulate the extrinsic state and corresponding behaviors to an independent structure, that is, building new classes.
 - Extrinsic state should not be huge.
-




Implementation 3: Managing shared objects

- Because objects are shared, clients shouldn't instantiate them directly. **FlyweightFactory** lets clients locate a particular flyweight.
 - **FlyweightFactory** objects often use an associative store to let clients look up flyweights of interest. The manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.
 - Sharability also implies some form of **reference counting** or **garbage collection** to reclaim a flyweight's storage when it's no longer needed. However, neither is necessary if the number of flyweights is fixed and small.
-



Example 1: An **FlyweightFactory** with registered pool and unshared **Flyweight**

```
abstract class Flyweight {  
    protected String identity;  
    public String getIdentity() {  
        return identity;  
    }  
}  
  
class UnsharedFlyweight extends Flyweight {  
    public UnsharedFlyweight(String identity) {  
        this.identity = identity;  
    }  
}  
  
class SharedFlyweight extends Flyweight {  
    public SharedFlyweight(String identity) {  
        this.identity = identity;  
    }  
}
```




```
class FlyweightFactory {
    private static Map<String, Flyweight> flyweightPool
        = new HashMap<String, Flyweight>();

    static {
        // can initialize the flyweightPool with default flyweights here
    }

    public static Flyweight getUnsharedFlyweight(String identity) {
        return new UnsharedFlyweight(identity);
    }

    public static Flyweight getSharedFlyweight(String identity) {
        if (!flyweightPool.containsKey(identity)) {
            flyweightPool.put(identity, new SharedFlyweight(identity));
        }
        return flyweightPool.get(identity);
    }

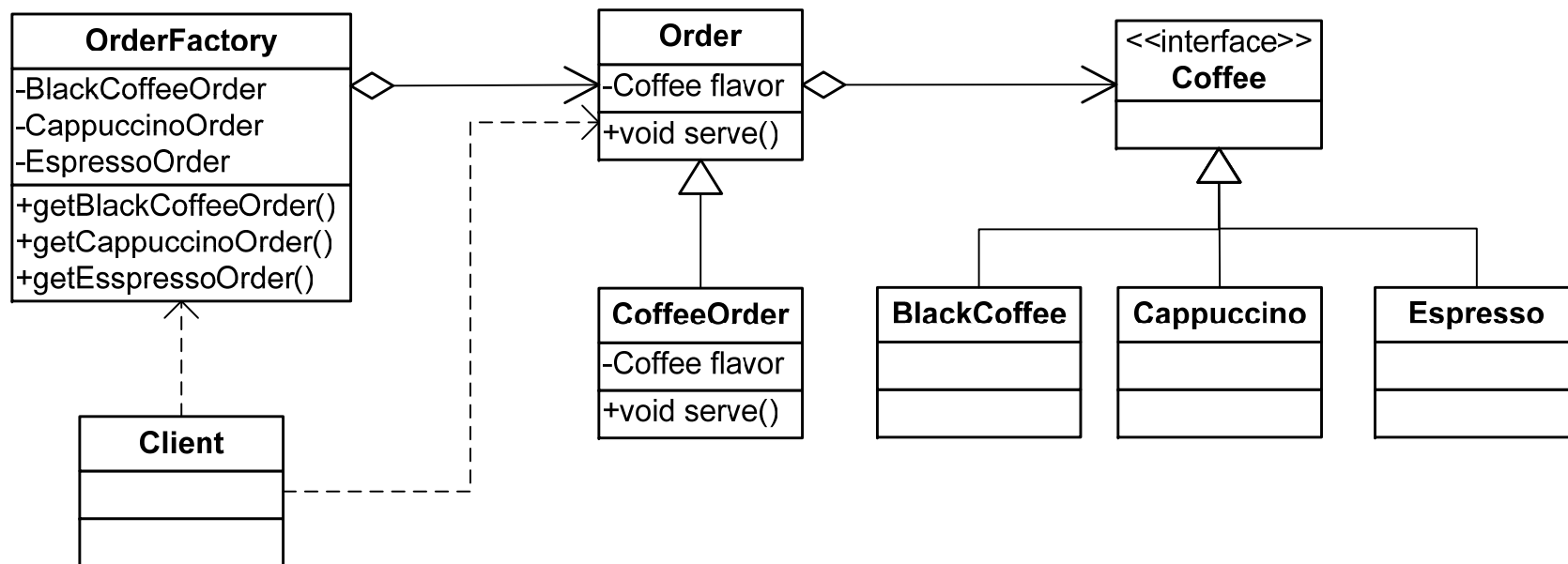
    public static void clearPool() {
        flyweightPool.clear();
    }
}
```



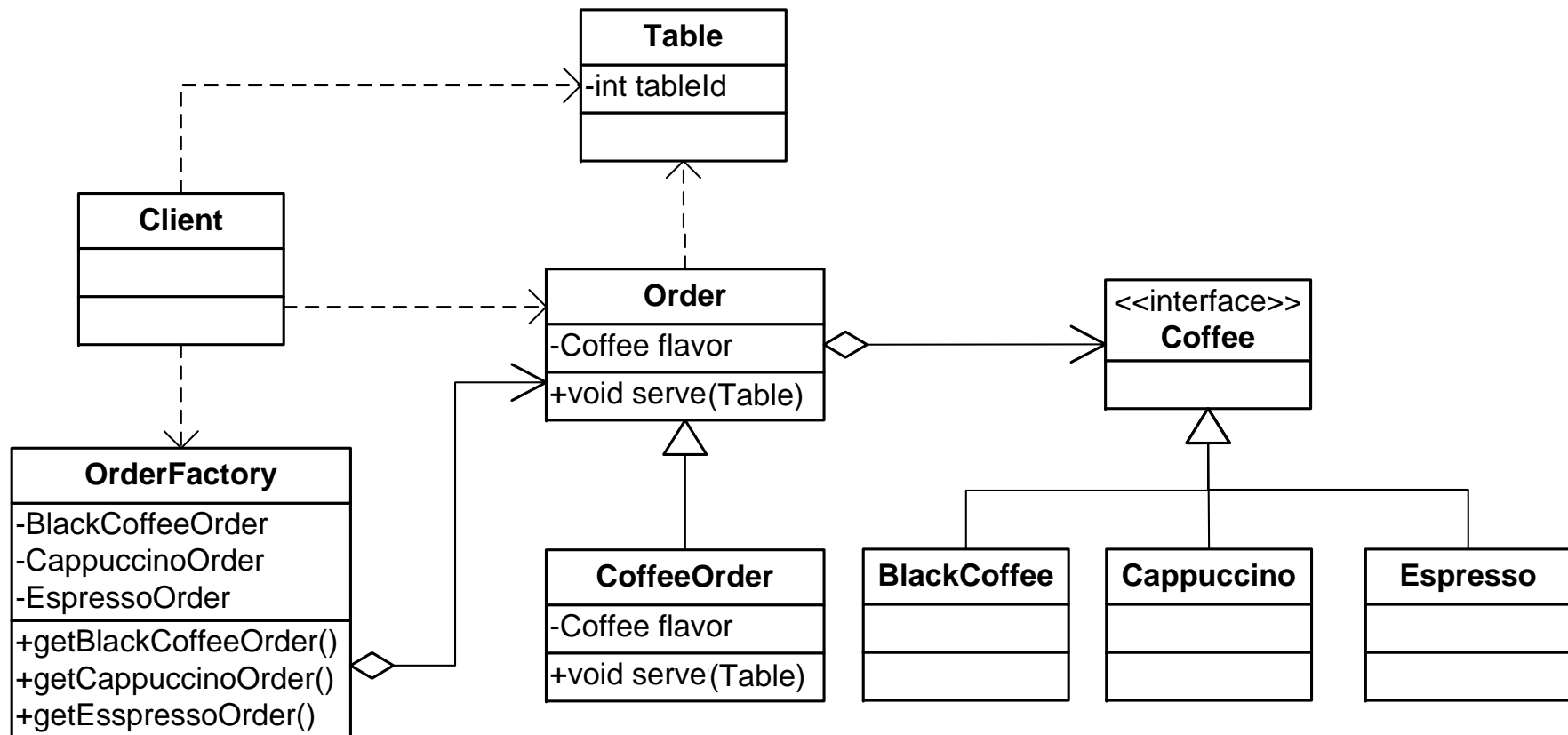
Example 2: Coffee Booth

Coffee : Black coffee, Cappuccino, Espresso

Order : serve()



Example 2: Coffee Bar

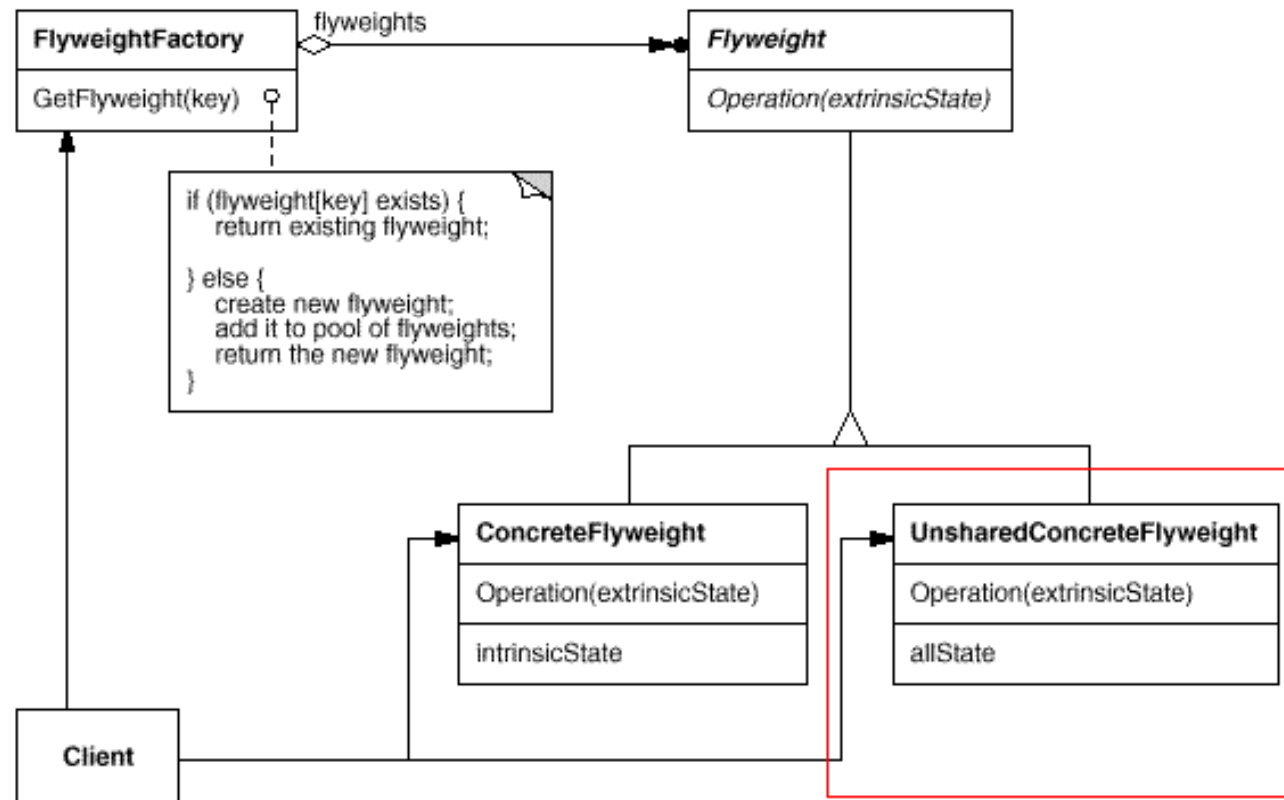




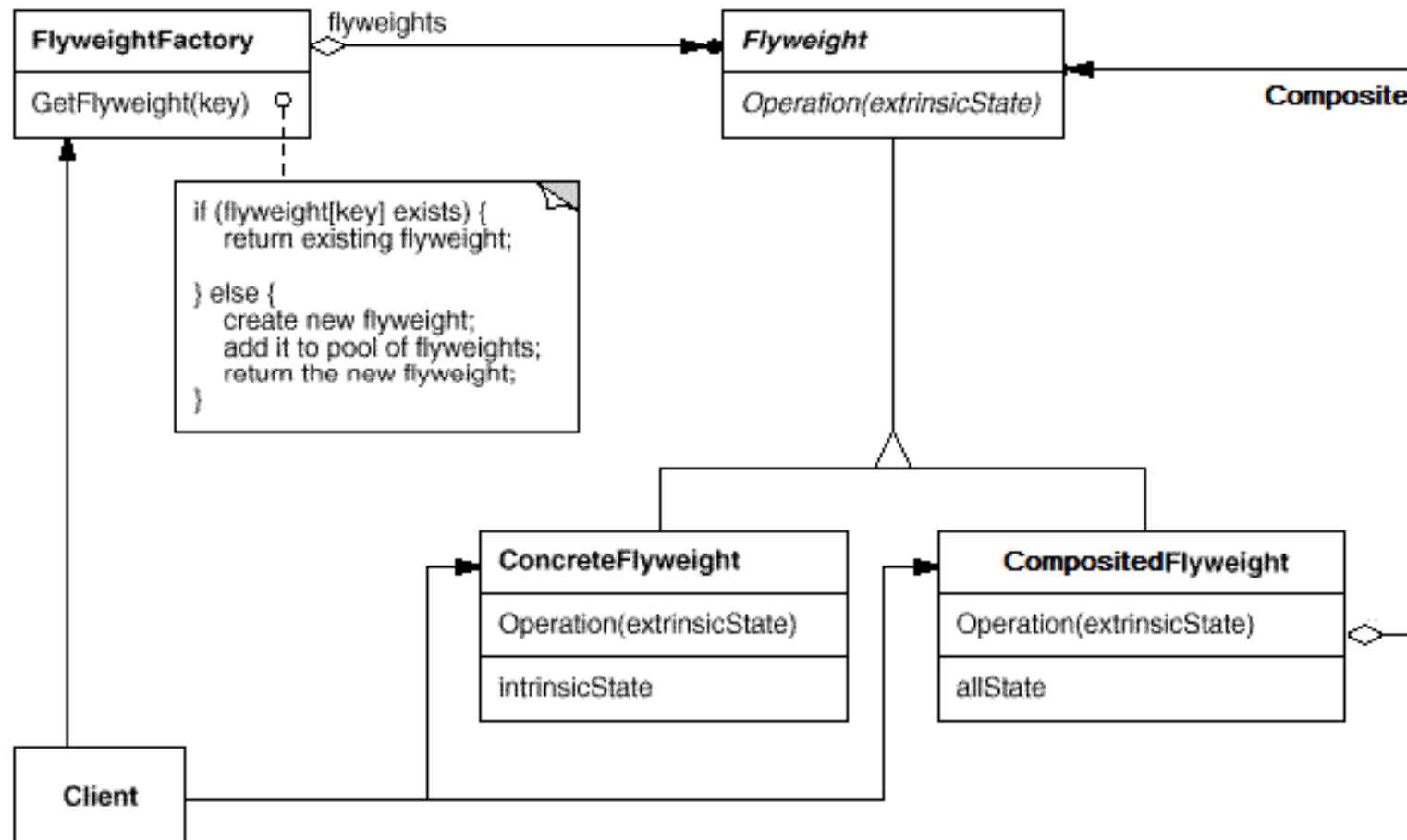
Extension: Shared Object Pattern

- The intent of flyweight pattern is saving storage by sharing larger quantity of **fine granularity** objects;
 - Sometimes, we share object but not adopt the flyweight pattern because the shared object is **coarse granularity**.
 - Saving storage by sharing the bigger, heavy and less quantity objects.
-

Variation 1: Constrainedly shared flyweight



Variation 2: Composited flyweight





Let's go to next...