Software Design Pattern

Final Homework Presentation

Serum Storage in stark industries

–using flyweight design pattern

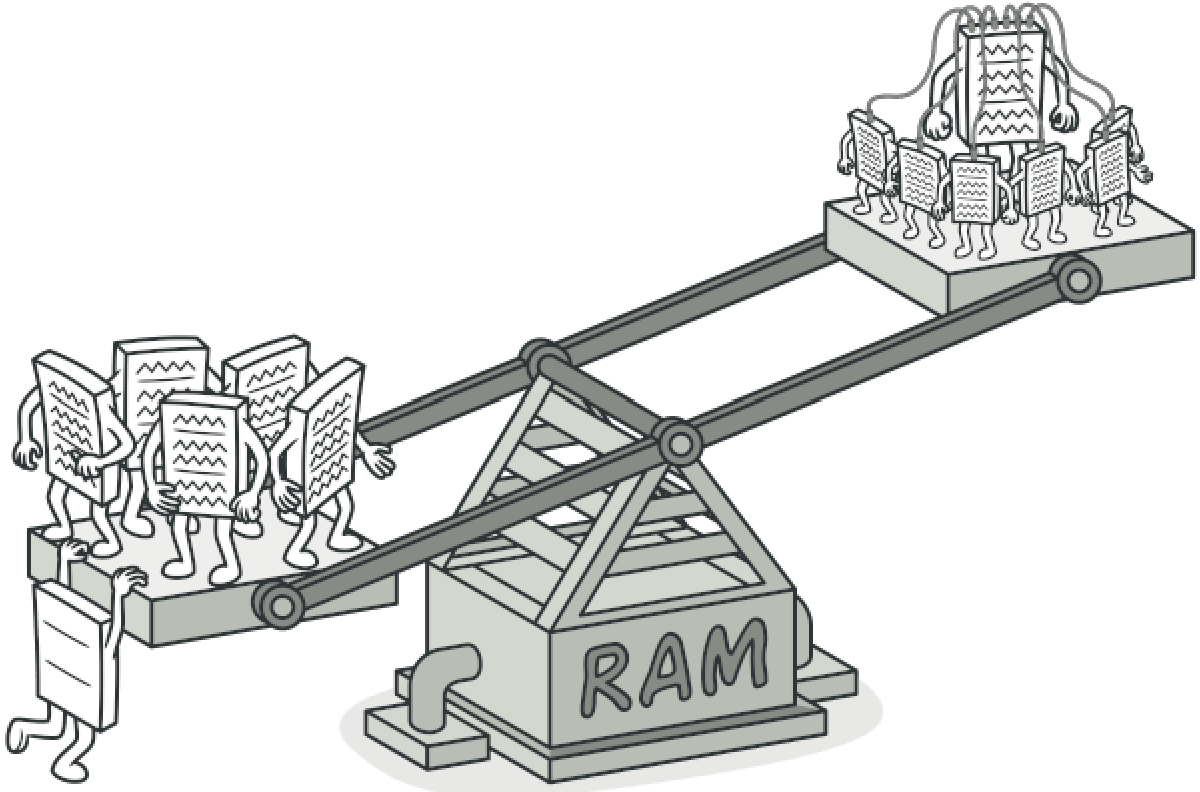
Teacher： 冯立波

Student : MD MAHFUZUR RAHMAN 罗尼

Student ID: 20183290375

Intent

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Explanation

- **Real world example:**

In stark industries lab has some super soldier and super human and some power serum. Many of the serum are the same so there is no need to create new object for each of them. Instead one object instance can represent multiple shelf items so memory footprint remains small.

- **In Plain words**

It is used to minimize memory usage or computational expenses by sharing as much as possible with similar objects.

- **Wikipedia says**

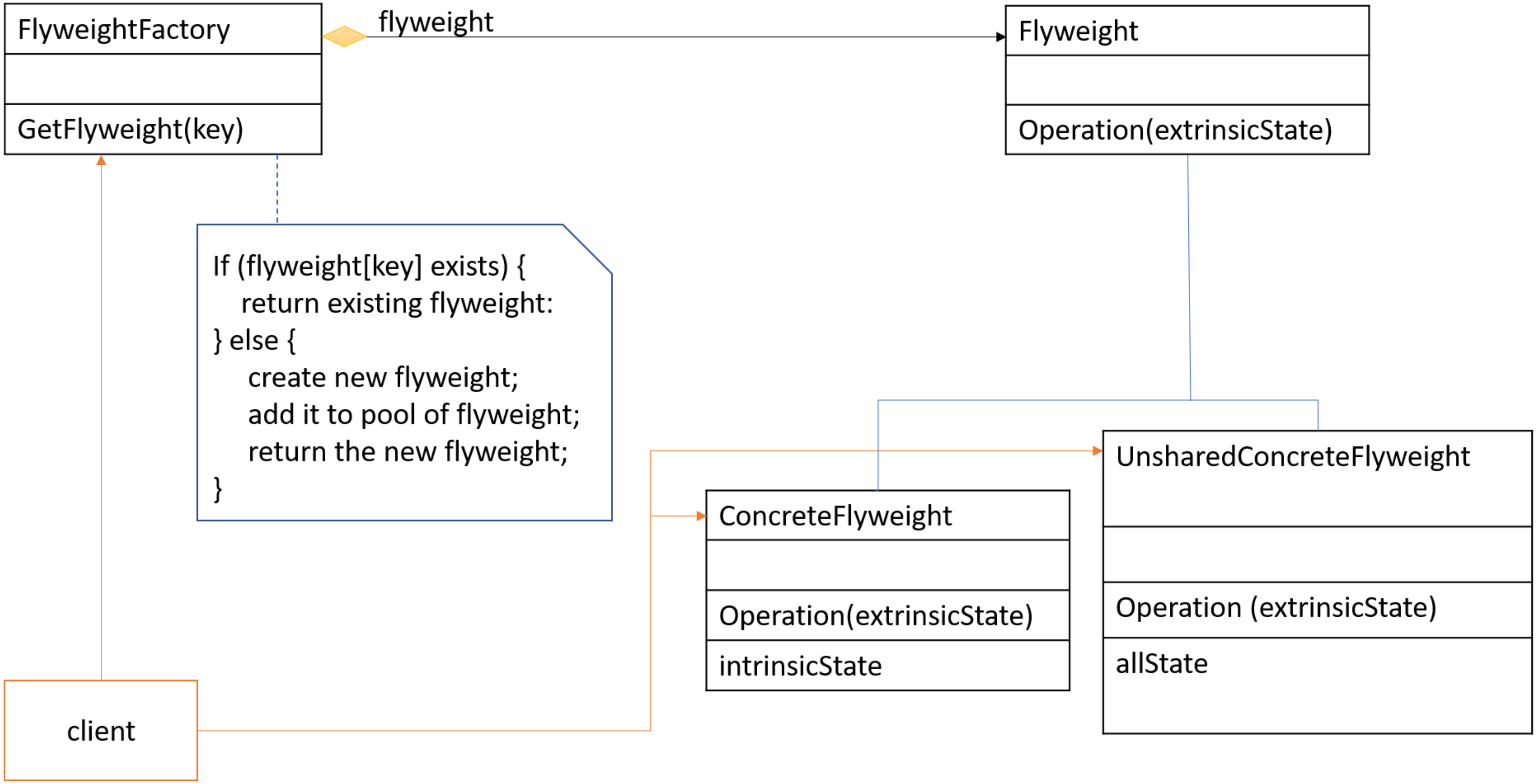
In computer programming, flyweight is a software design pattern. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

Advantage of Flyweight Design Pattern

- It reduces the number of objects
- It reduces the amount of memory and storage device required if the objects are persisted.

Structure

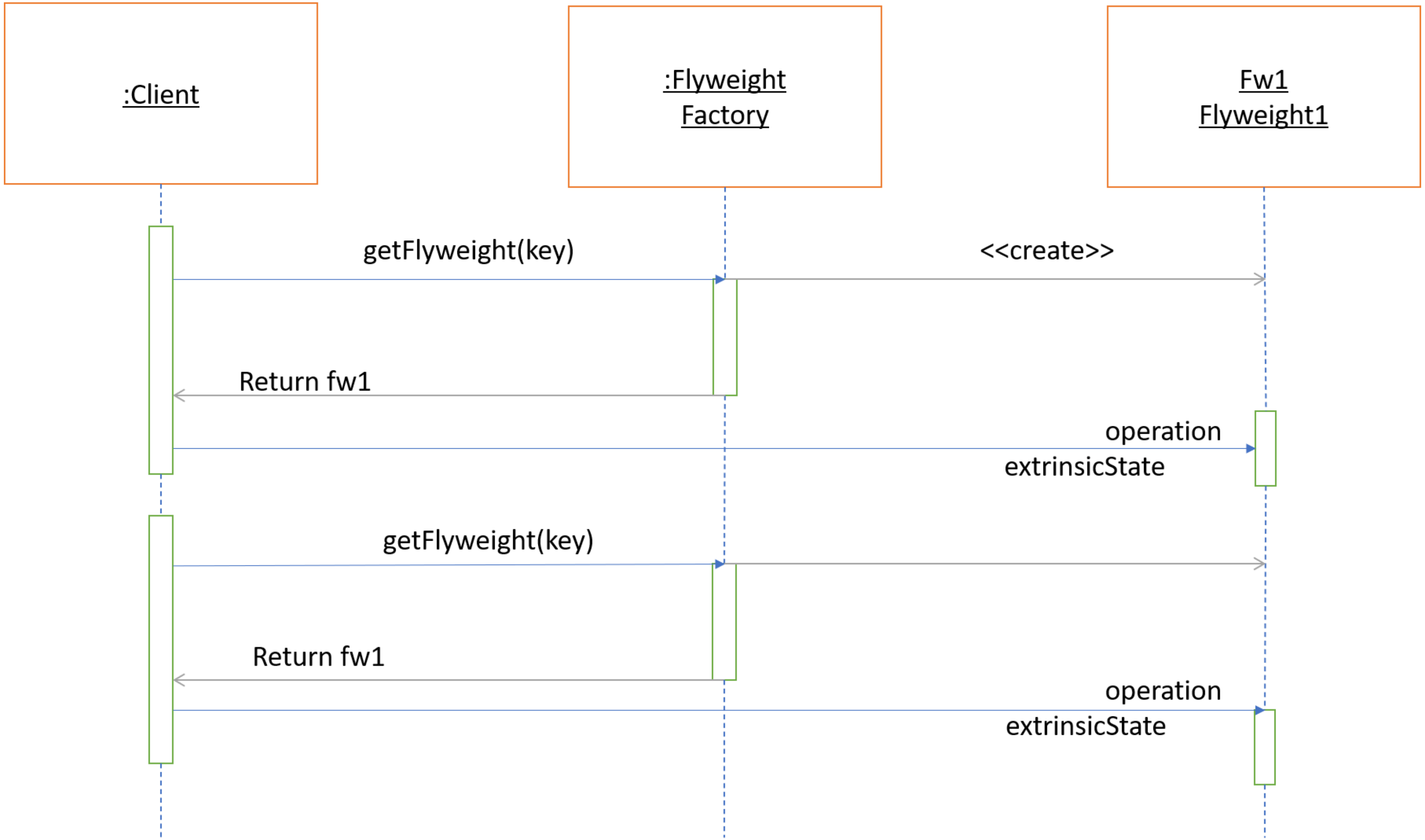
- **Sample class diagram**



FlyweightFactory
GetFlyweight(key)
flyweight
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation (extrinsicState)
allState
client

```
If (flyweight[key] exists) {
   return existing flyweight:
} else {
    create new flyweight;
    add it to pool of flyweight;
    return the new flyweight;
}
```

- **Sample Sequence diagram**



Implementation details

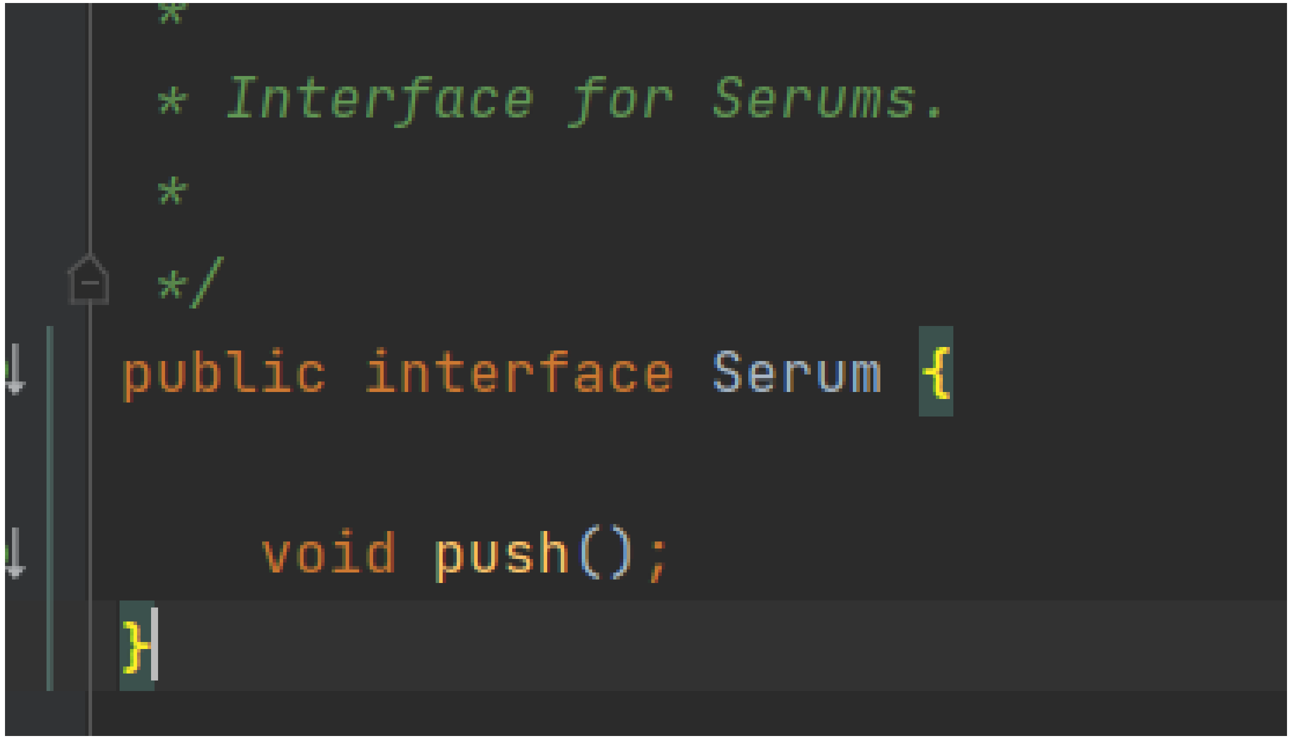
- There are multiple ways to implement the flyweight pattern. One example is **mutability**: whether the objects storing extrinsic flyweight state can change.
- **Immutable** objects are easily shared, but require creating new extrinsic objects whenever a change in state occurs. In contrast, mutable objects can share state. Mutability allows better object reuse via the caching and re-initialization of old, unused objects. Sharing is usually nonviable when state is highly variable.
- Retrieval
- Caching
- Concurrency

Pseudocode

In this example stark industries lab has some super soldier and super human and some power serum. Many of the serum are the same so there is no need to create new object for each of them. Instead one object instance can represent multiple shelf items so memory footprint remains small.

Pseudocode

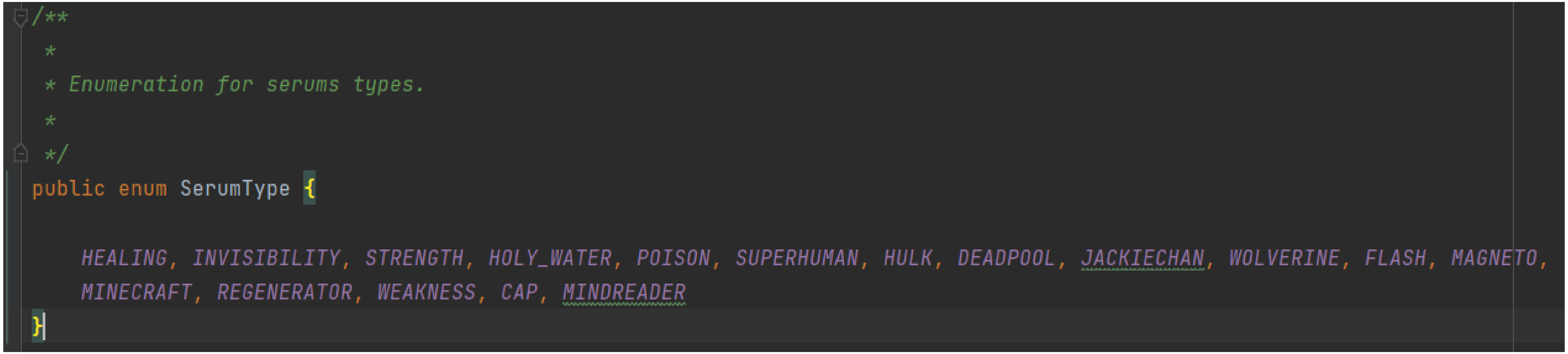
- Let’s create Serum Interface



```
 *
 * Interface for Serums.
 *
 */
public interface Serum {

    void push();
}
```

Pseudocode

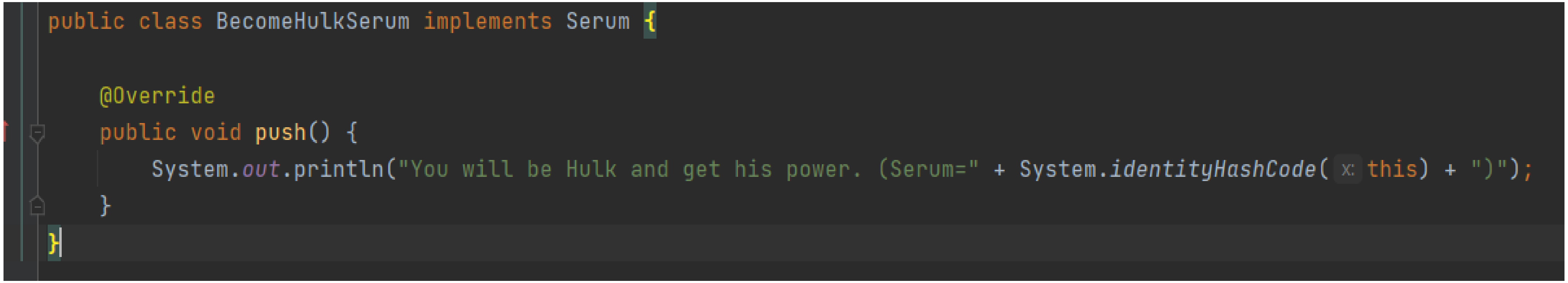
- Now let’s create enumeration for serums types



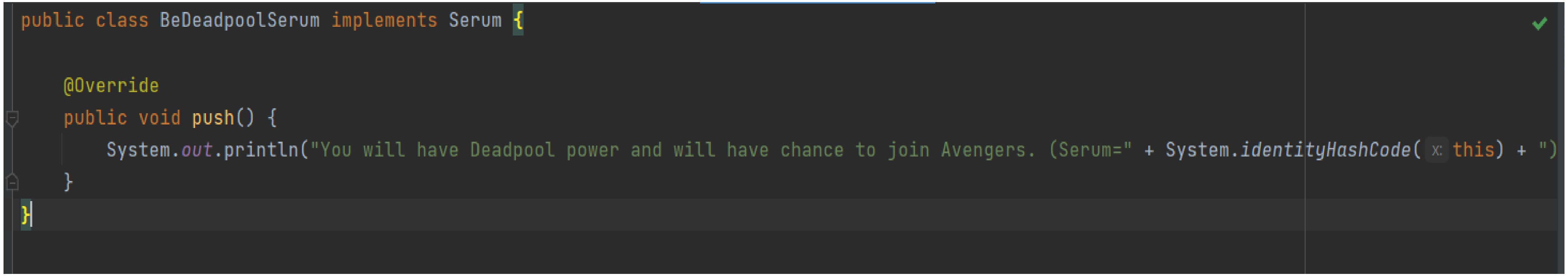
```
/**
 *
 * Enumeration for serums types.
 *
 */
public enum SerumType {

    HEALING, INVISIBILITY, STRENGTH, HOLY_WATER, POISON, SUPERHUMAN, HULK, DEADPOOL, JACKIECHAN, WOLVERINE, FLASH, MAGNETO,
    MINECRAFT, REGENERATOR, WEAKNESS, CAP, MINDREADER
}
```

Pseudocode

- Now let’s create different types of serum class



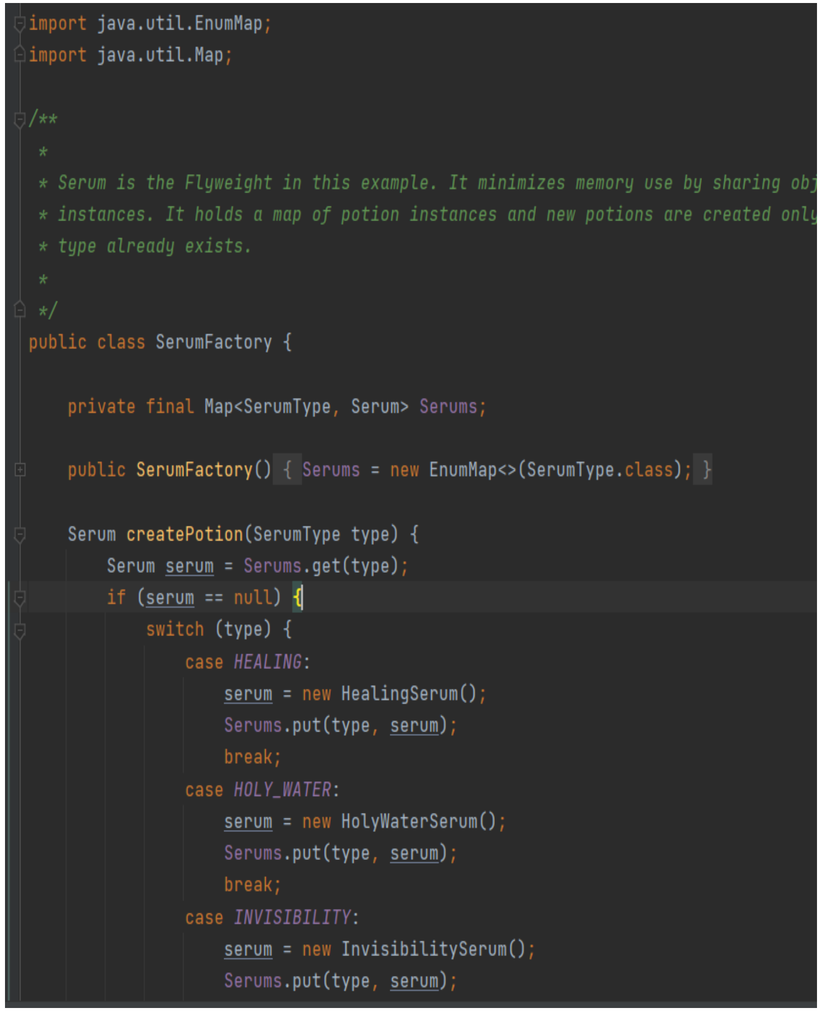
```
public class BecomeHulkSerum implements Serum {

    @Override
    public void push() {
        System.out.println("You will be Hulk and get his power. (Serum=" + System.identityHashCode( x: this) + ")");
    }
}
```



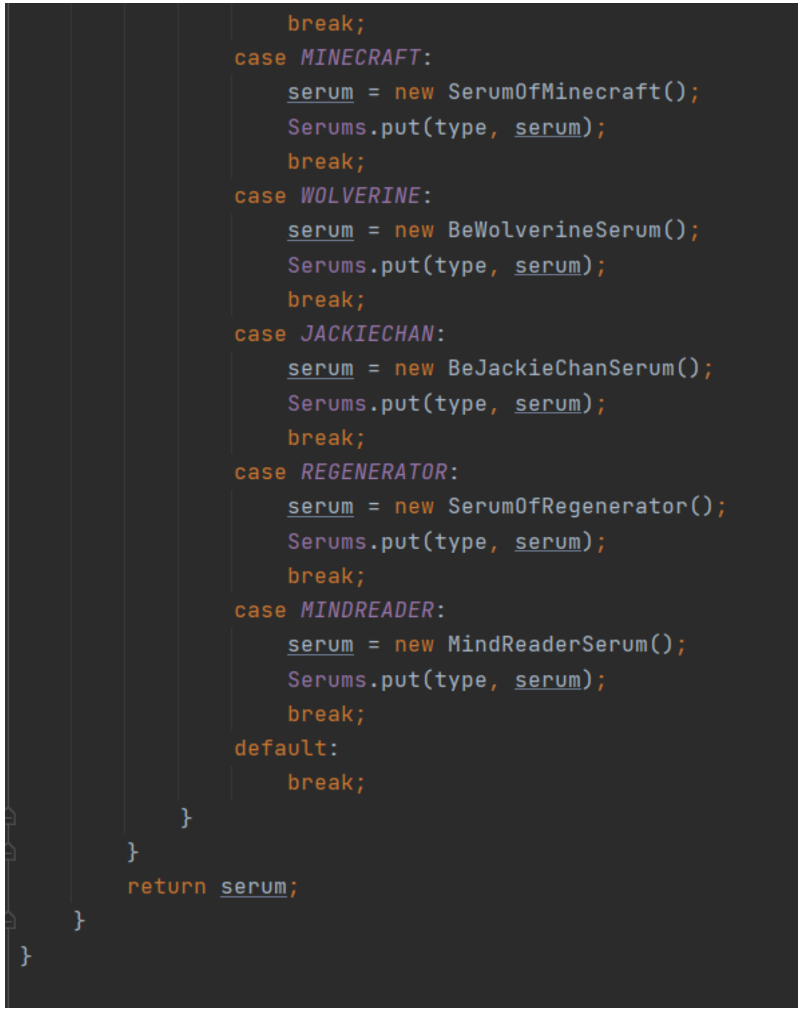
```
public class BeDeadpoolSerum implements Serum {

    @Override
    public void push() {
        System.out.println("You will have Deadpool power and will have chance to join Avengers. (Serum=" + System.identityHashCode( x: this) + ")
    }
}
```

Pseudocode

- Then the actual Flyweight object which is the factory for creating serums



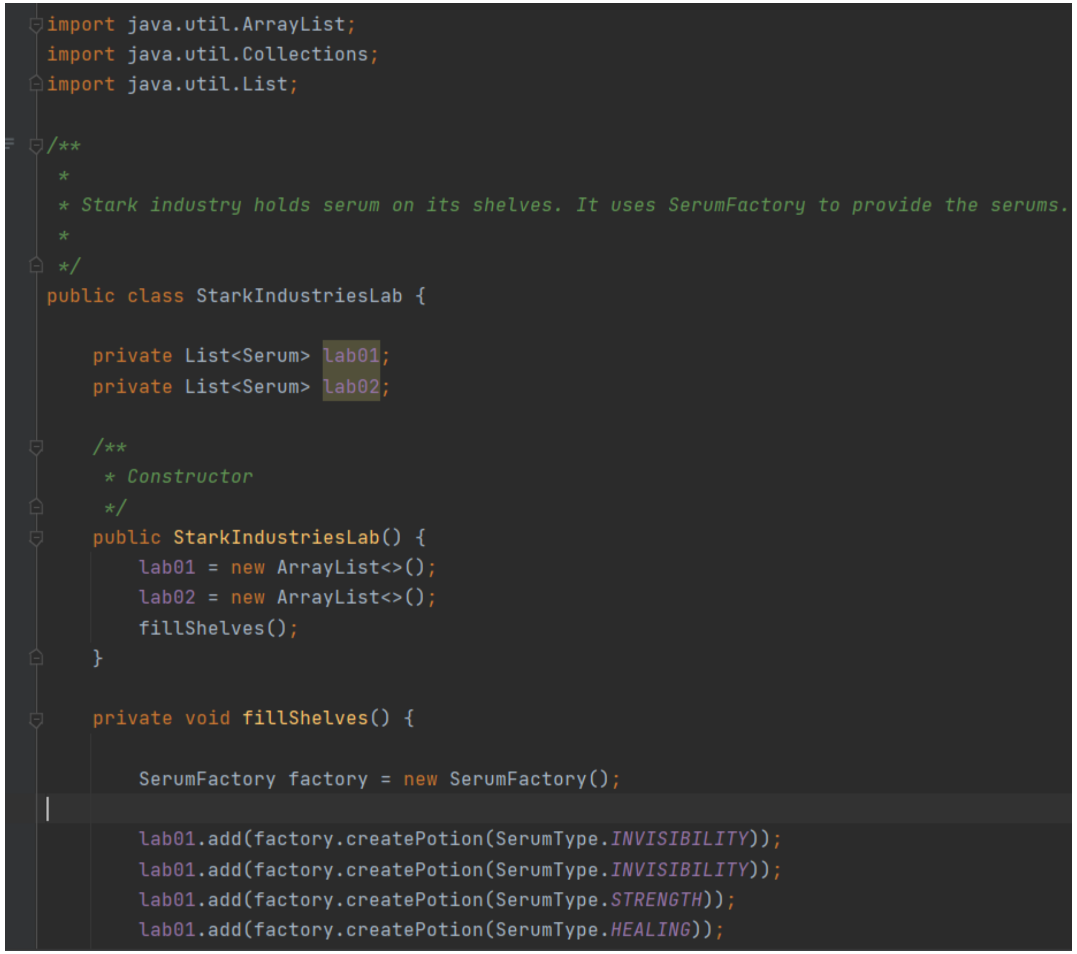
```java
import java.util.EnumMap;
import java.util.Map;

/**
 *
 * Serum is the Flyweight in this example. It minimizes memory use by sharing obj
 * instances. It holds a map of potion instances and new potions are created only
 * type already exists.
 *
 */
public class SerumFactory {

    private final Map<SerumType, Serum> Serums;

    public SerumFactory() { Serums = new EnumMap<>(SerumType.class); }

    Serum createPotion(SerumType type) {
        Serum serum = Serums.get(type);
        if (serum == null) {
            switch (type) {
                case HEALING:
                    serum = new HealingSerum();
                    Serums.put(type, serum);
                    break;
                case HOLY_WATER:
                    serum = new HolyWaterSerum();
                    Serums.put(type, serum);
                    break;
                case INVISIBILITY:
                    serum = new InvisibilitySerum();
                    Serums.put(type, serum);
```



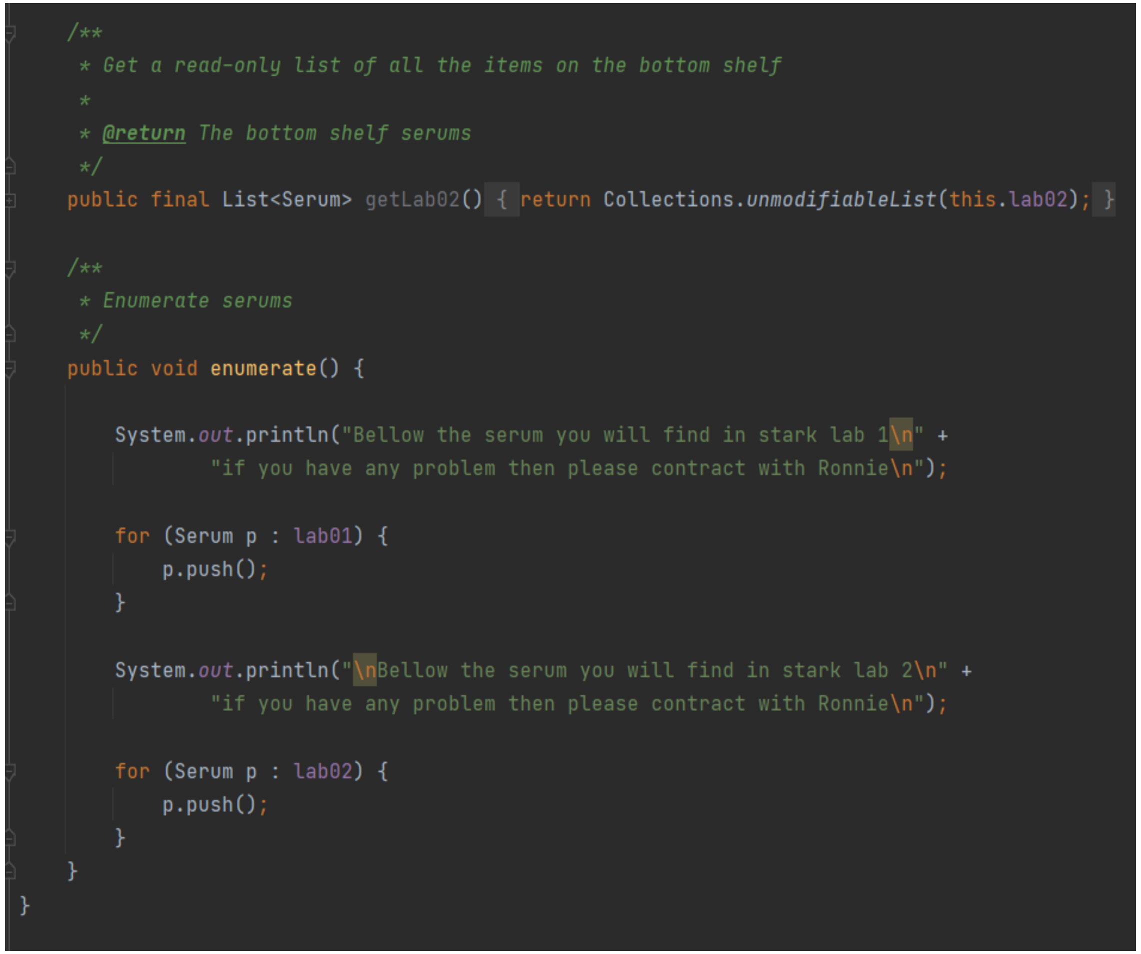
```java
                    break;
                case MINECRAFT:
                    serum = new SerumOfMinecraft();
                    Serums.put(type, serum);
                    break;
                case WOLVERINE:
                    serum = new BeWolverineSerum();
                    Serums.put(type, serum);
                    break;
                case JACKIECHAN:
                    serum = new BeJackieChanSerum();
                    Serums.put(type, serum);
                    break;
                case REGENERATOR:
                    serum = new SerumOfRegenerator();
                    Serums.put(type, serum);
                    break;
                case MINDREADER:
                    serum = new MindReaderSerum();
                    Serums.put(type, serum);
                    break;
                default:
                    break;
            }
        }
        return serum;
    }
}
```

Pseudocode

- Now create stark industries lab to uses Serum Factory to provide the serums



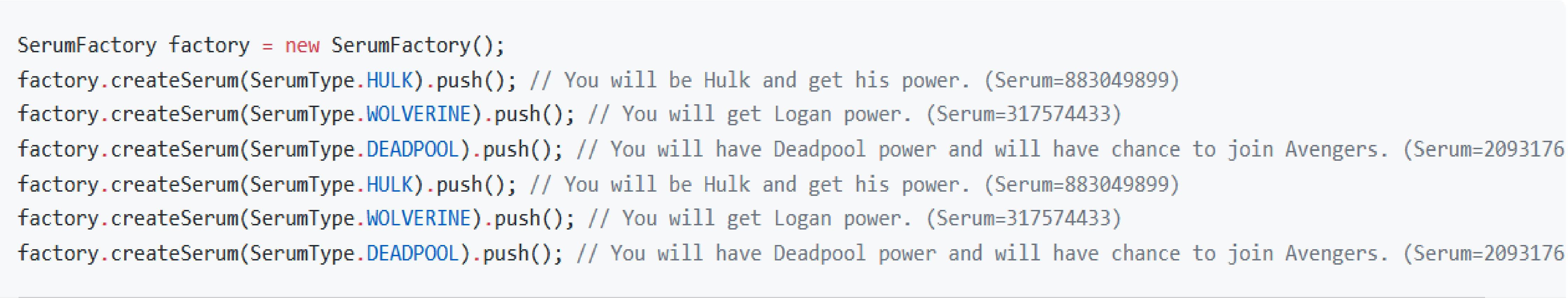
```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 *
 * Stark industry holds serum on its shelves. It uses SerumFactory to provide the serums.
 *
 */
public class StarkIndustriesLab {

    private List<Serum> lab01;
    private List<Serum> lab02;

    /**
     * Constructor
     */
    public StarkIndustriesLab() {
        lab01 = new ArrayList<>();
        lab02 = new ArrayList<>();
        fillShelves();
    }

    private void fillShelves() {

        SerumFactory factory = new SerumFactory();

        lab01.add(factory.createPotion(SerumType.INVISIBILITY));
        lab01.add(factory.createPotion(SerumType.INVISIBILITY));
        lab01.add(factory.createPotion(SerumType.STRENGTH));
        lab01.add(factory.createPotion(SerumType.HEALING));
```



```java
    /**
     * Get a read-only list of all the items on the bottom shelf
     *
     * @return The bottom shelf serums
     */
    public final List<Serum> getLab02() { return Collections.unmodifiableList(this.lab02); }

    /**
     * Enumerate serums
     */
    public void enumerate() {

        System.out.println("Bellow the serum you will find in stark lab 1\n" +
                "if you have any problem then please contract with Ronnie\n");

        for (Serum p : lab01) {
            p.push();
        }

        System.out.println("\nBellow the serum you will find in stark lab 2\n" +
                "if you have any problem then please contract with Ronnie\n");

        for (Serum p : lab02) {
            p.push();
        }
    }
}
```

Pseudocode

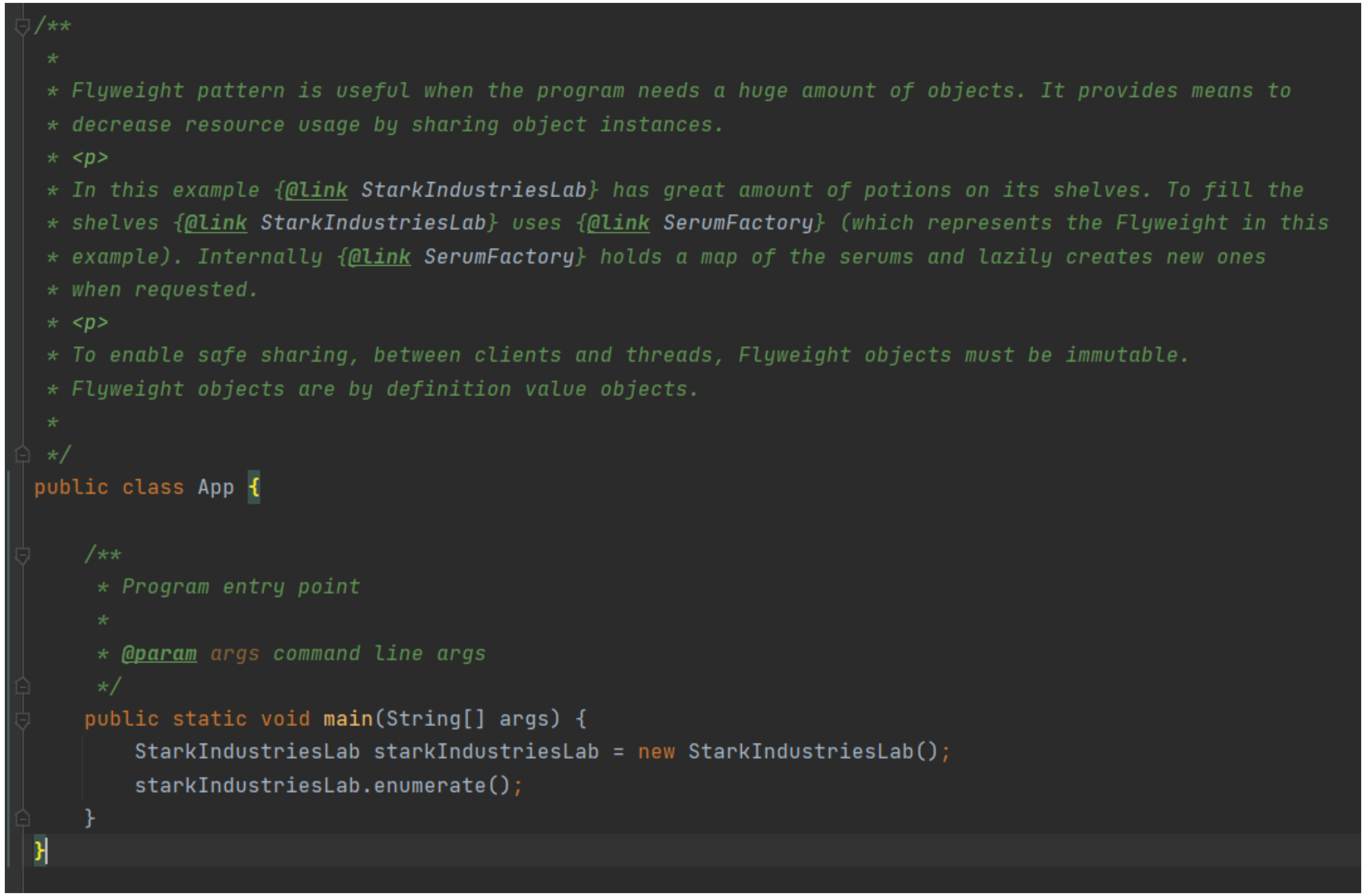
- It will be use like this



```
SerumFactory factory = new SerumFactory();
factory.createSerum(SerumType.HULK).push(); // You will be Hulk and get his power. (Serum=883049899)
factory.createSerum(SerumType.WOLVERINE).push(); // You will get Logan power. (Serum=317574433)
factory.createSerum(SerumType.DEADPOOL).push(); // You will have Deadpool power and will have chance to join Avengers. (Serum=2093176
factory.createSerum(SerumType.HULK).push(); // You will be Hulk and get his power. (Serum=883049899)
factory.createSerum(SerumType.WOLVERINE).push(); // You will get Logan power. (Serum=317574433)
factory.createSerum(SerumType.DEADPOOL).push(); // You will have Deadpool power and will have chance to join Avengers. (Serum=2093176
```

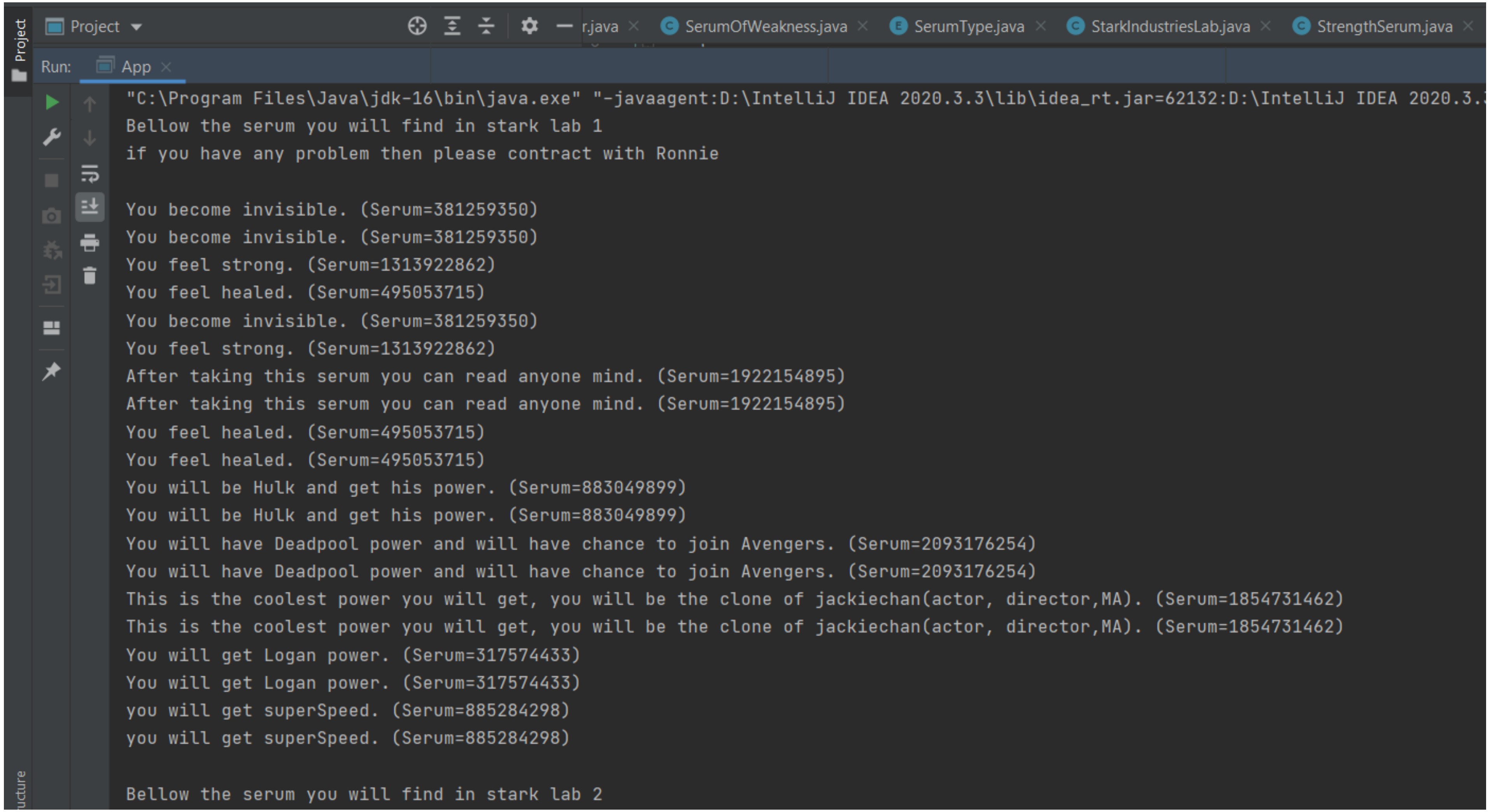
Pseudocode

- Now create a App class and run the program



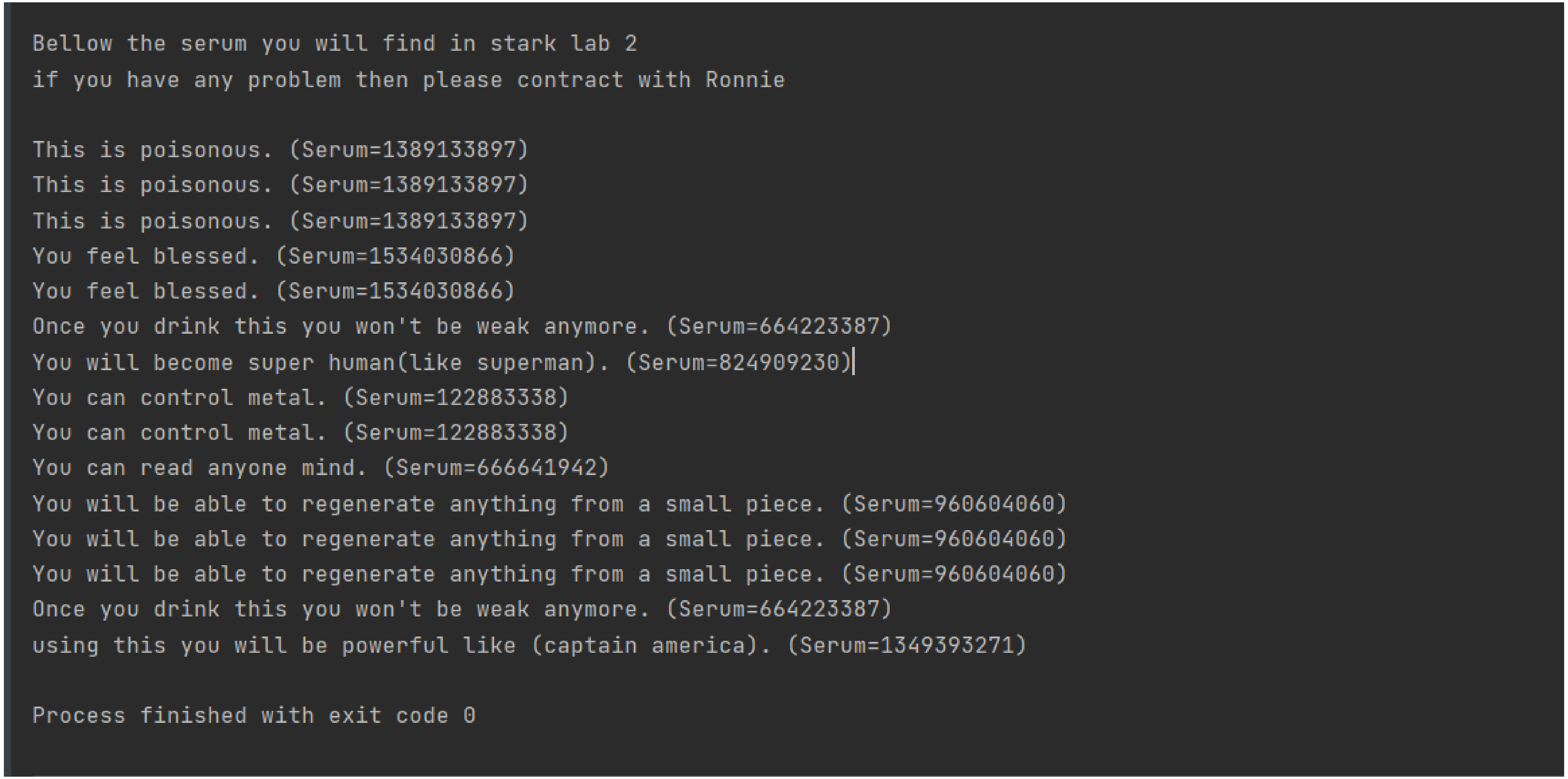
```
/**
 *
 * Flyweight pattern is useful when the program needs a huge amount of objects. It provides means to
 * decrease resource usage by sharing object instances.
 * <p>
 * In this example {@link StarkIndustriesLab} has great amount of potions on its shelves. To fill the
 * shelves {@link StarkIndustriesLab} uses {@link SerumFactory} (which represents the Flyweight in this
 * example). Internally {@link SerumFactory} holds a map of the serums and lazily creates new ones
 * when requested.
 * <p>
 * To enable safe sharing, between clients and threads, Flyweight objects must be immutable.
 * Flyweight objects are by definition value objects.
 *
 */
public class App {

    /**
     * Program entry point
     *
     * @param args command line args
     */
    public static void main(String[] args) {
        StarkIndustriesLab starkIndustriesLab = new StarkIndustriesLab();
        starkIndustriesLab.enumerate();
    }
}
```

Result



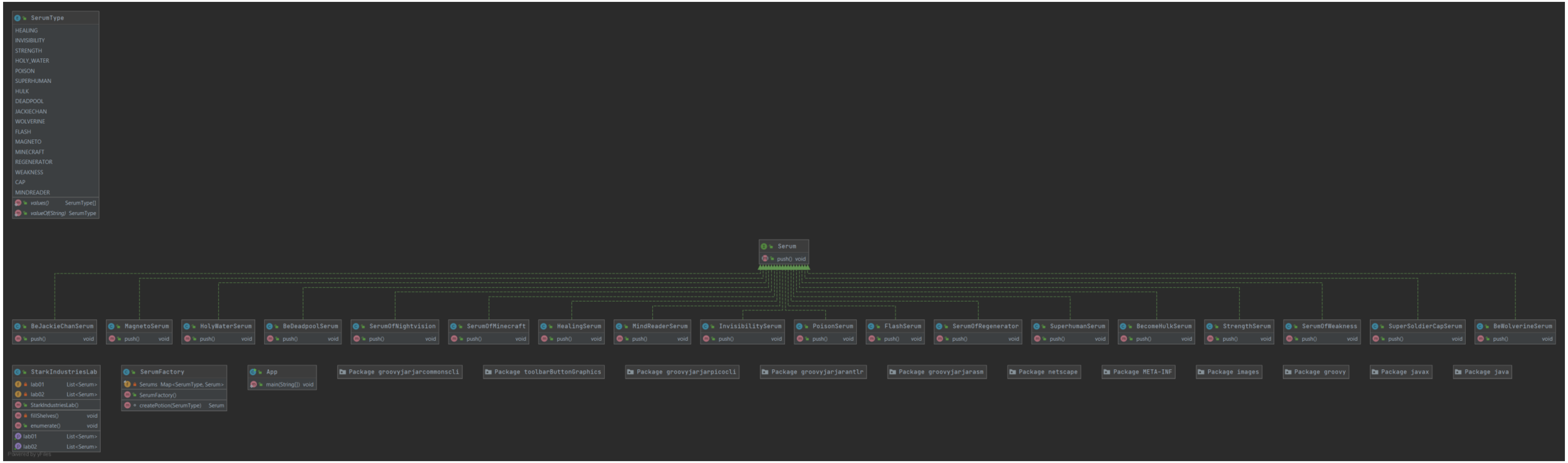
```
"C:\Program Files\Java\jdk-16\bin\java.exe" "-javaagent:D:\IntelliJ IDEA 2020.3.3\lib\idea_rt.jar=62132:D:\IntelliJ IDEA 2020.3.
Bellow the serum you will find in stark lab 1
if you have any problem then please contract with Ronnie

You become invisible. (Serum=381259350)
You become invisible. (Serum=381259350)
You feel strong. (Serum=1313922862)
You feel healed. (Serum=495053715)
You become invisible. (Serum=381259350)
You feel strong. (Serum=1313922862)
After taking this serum you can read anyone mind. (Serum=1922154895)
After taking this serum you can read anyone mind. (Serum=1922154895)
You feel healed. (Serum=495053715)
You feel healed. (Serum=495053715)
You will be Hulk and get his power. (Serum=883049899)
You will be Hulk and get his power. (Serum=883049899)
You will have Deadpool power and will have chance to join Avengers. (Serum=2093176254)
You will have Deadpool power and will have chance to join Avengers. (Serum=2093176254)
This is the coolest power you will get, you will be the clone of jackiechan(actor, director,MA). (Serum=1854731462)
This is the coolest power you will get, you will be the clone of jackiechan(actor, director,MA). (Serum=1854731462)
You will get Logan power. (Serum=317574433)
You will get Logan power. (Serum=317574433)
you will get superSpeed. (Serum=885284298)
you will get superSpeed. (Serum=885284298)

Bellow the serum you will find in stark lab 2
```

Result



```
Bellow the serum you will find in stark lab 2
if you have any problem then please contract with Ronnie

This is poisonous. (Serum=1389133897)
This is poisonous. (Serum=1389133897)
This is poisonous. (Serum=1389133897)
You feel blessed. (Serum=1534030866)
You feel blessed. (Serum=1534030866)
Once you drink this you won't be weak anymore. (Serum=664223387)
You will become super human(like superman). (Serum=824909230)
You can control metal. (Serum=122883338)
You can control metal. (Serum=122883338)
You can read anyone mind. (Serum=666641942)
You will be able to regenerate anything from a small piece. (Serum=960604060)
You will be able to regenerate anything from a small piece. (Serum=960604060)
You will be able to regenerate anything from a small piece. (Serum=960604060)
Once you drink this you won't be weak anymore. (Serum=664223387)
using this you will be powerful like (captain america). (Serum=1349393271)

Process finished with exit code 0
```

Class Diagram



Applicability

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made 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.

THANK YOU

How to Implement

1. Divide fields of a class that will become a flyweight into two parts:
    - the intrinsic state: the fields that contain unchanging data duplicated across many objects
    - the extrinsic state: the fields that contain contextual data unique to each object
2. Leave the fields that represent the intrinsic state in the class, but make sure they’re immutable. They should take their initial values only inside the constructor.
3. Go over methods that use fields of the extrinsic state. For each field used in the method, introduce a new parameter and use it instead of the field.

How to Implement

- 4. Optionally, create a factory class to manage the pool of flyweights. It should check for an existing flyweight before creating a new one. Once the factory is in place, clients must only request flyweights through it. They should describe the desired flyweight by passing its intrinsic state to the factory.
- 5. The client must store or calculate values of the extrinsic state (context) to be able to call methods of flyweight objects. For the sake of convenience, the extrinsic state along with the flyweight-referencing field may be moved to a separate context class.

Pros and Cons

- **Pros:**

1. You can save lots of RAM, assuming your program has tons of similar objects.

- **Cons**

1. You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method.
2. The code becomes much more complicated. New team members will always be wondering why the state of an entity was separated in such a way.

Relations with other Patterns

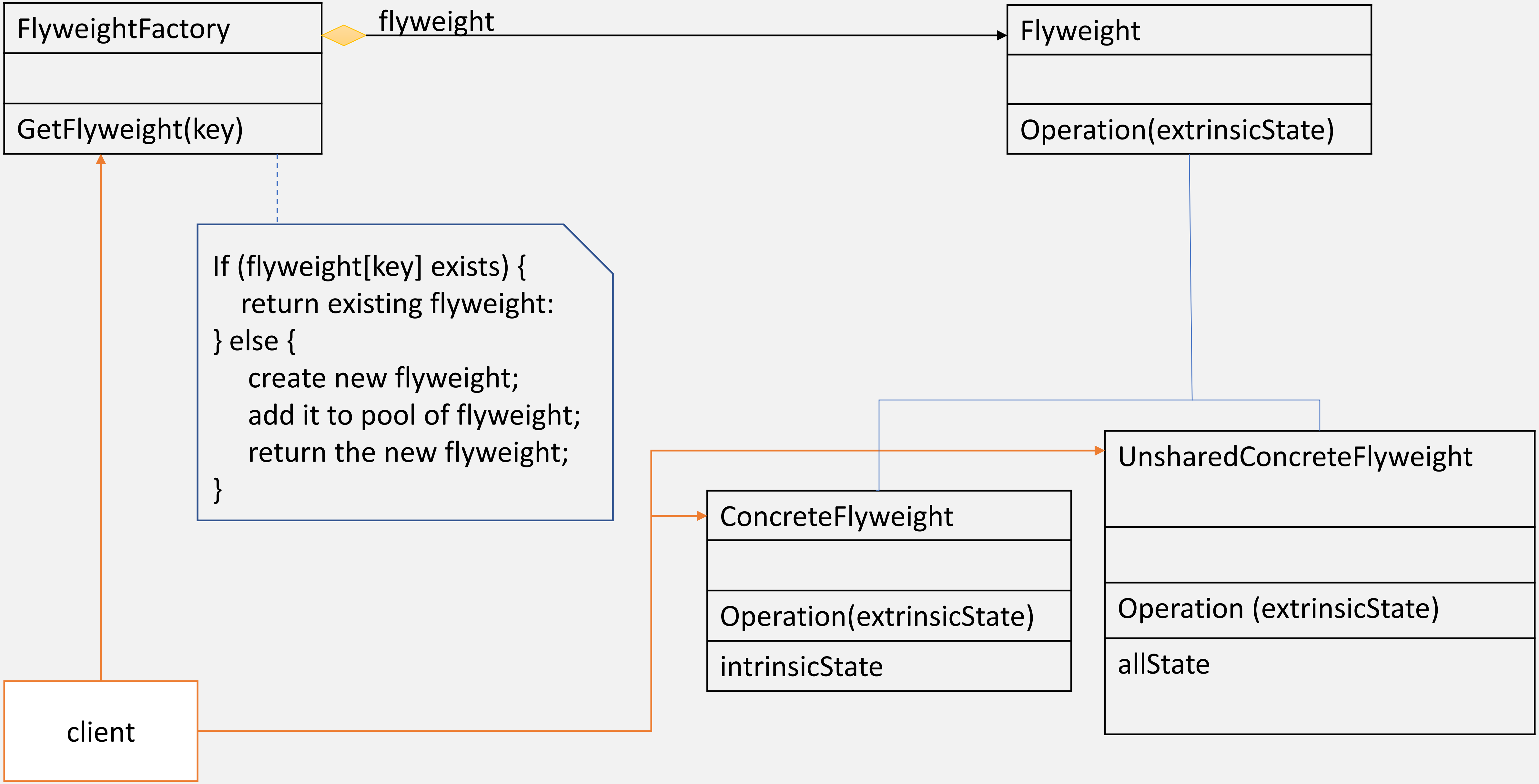
1. You can implement shared leaf nodes of the **Composite** tree as **Flyweights** to save some RAM.

2. **Flyweight** shows how to make lots of little objects, whereas **Facade** shows how to make a single object that represents an entire subsystem.

3. **Flyweight** would resemble **Singleton** if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:

Relations with Other Patterns

- ✓There should be only one Singleton instance, whereas a *Flyweight* class can have multiple instances with different intrinsic states.
- ✓The *Singleton* object can be mutable. Flyweight objects are immutable.



FlyweightFactory

GetFlyweight(key)

flyweight

Flyweight

Operation(extrinsicState)

```
If (flyweight[key] exists) {
   return existing flyweight:
} else {
    create new flyweight;
    add it to pool of flyweight;
    return the new flyweight;
}
```

ConcreteFlyweight

Operation(extrinsicState)

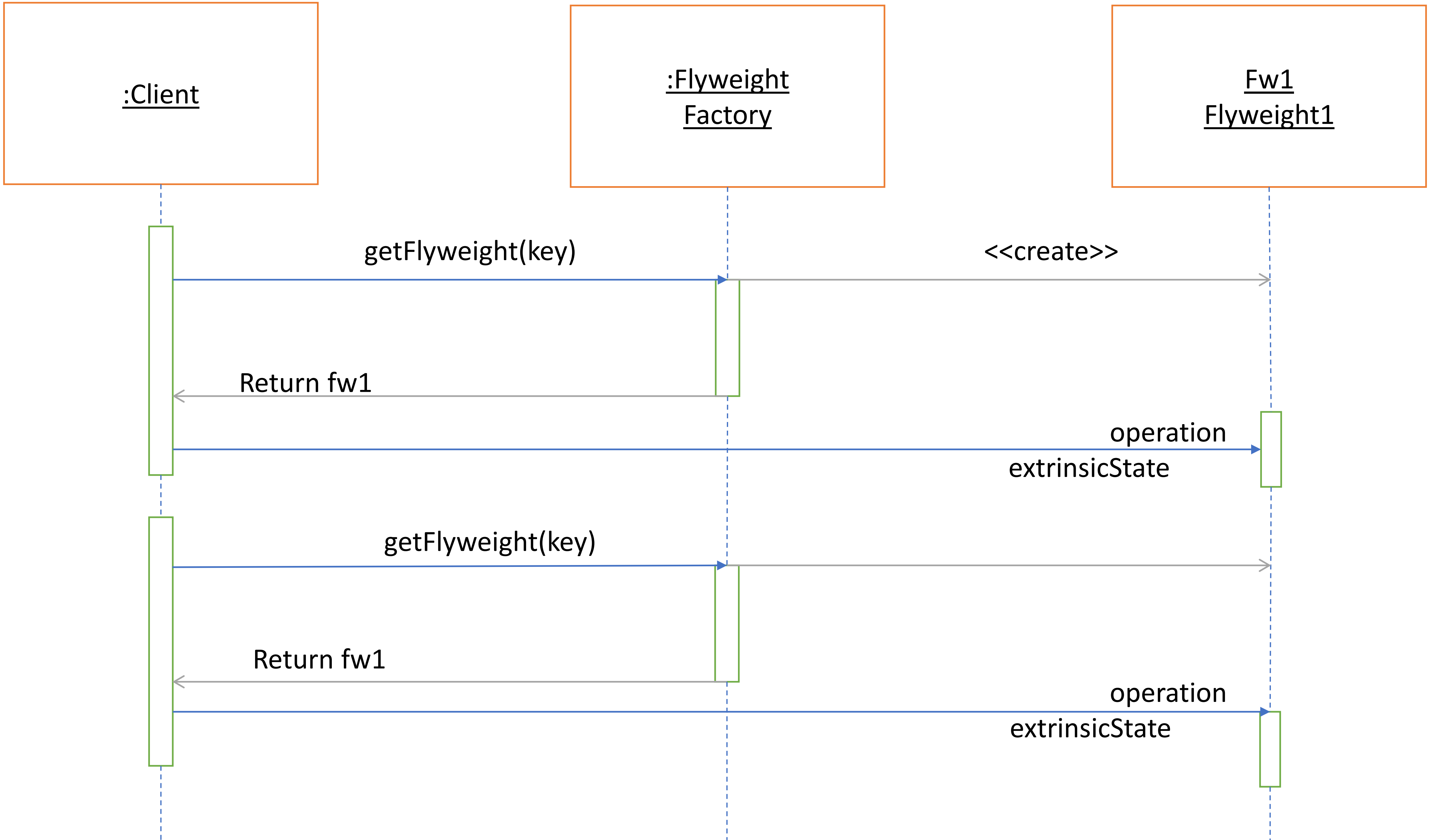
intrinsicState

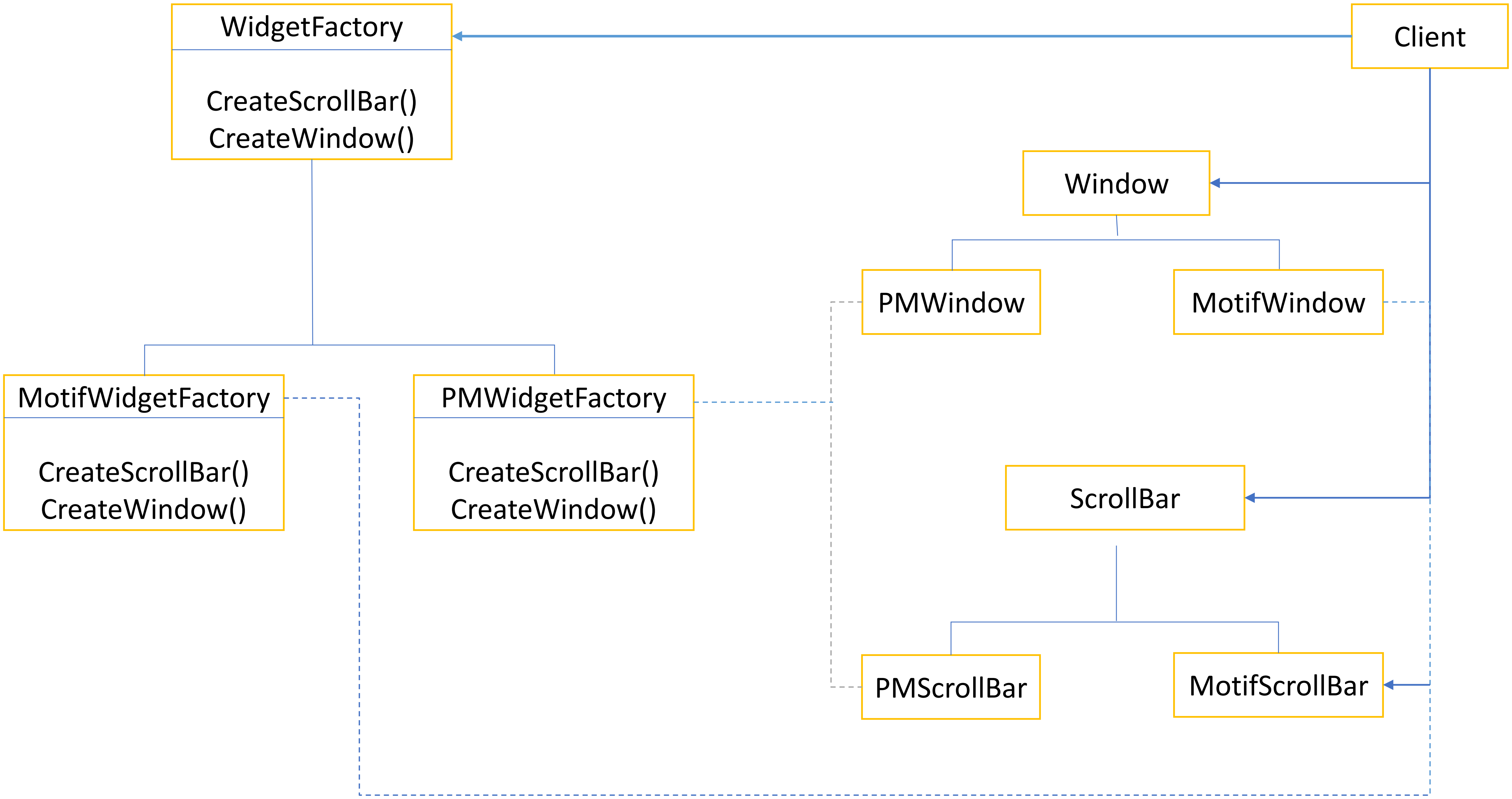
UnsharedConcreteFlyweight

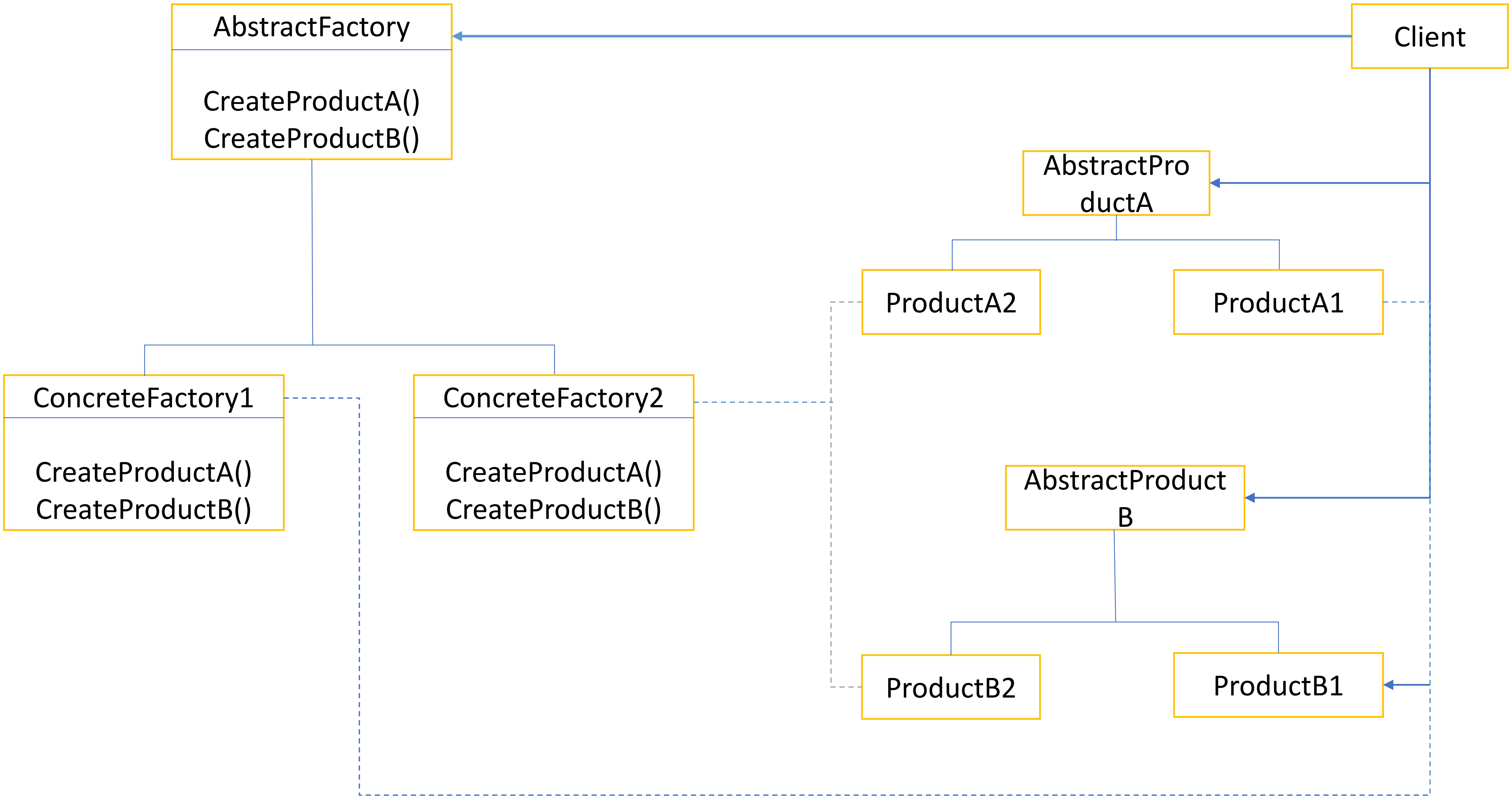
Operation (extrinsicState)

allState

client


:Client
:Flyweight Factory
Fw1 Flyweight1
getFlyweight(key)
<<create>>
Return fw1
operation
extrinsicState
getFlyweight(key)
Return fw1
operation
extrinsicState


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar



AbstractFactory

CreateProductA()
CreateProductB()

Client

AbstractProductA

ProductA2

ProductA1

ConcreteFactory1

CreateProductA()
CreateProductB()

ConcreteFactory2

CreateProductA()
CreateProductB()

AbstractProductB

ProductB2

ProductB1