

9/19/23

# CSCI 3400: OBJECT ORIENTED DESIGN

FALL 2023

LECTURE 05

FAHMIDA HAMID

# So far ...

- SOLID
- Some patterns:
  - Decorator
    - >> how to dynamically create new variants of an object with the use of composition and inheritance
  - Observer
    - >> How different objects can implement a “subscription-notification” relationship among them
    - >> push model vs pull model

9/19/23

# THE FACTORY PATTERN

A **Creational** Design Pattern

# Let's start with an Example

```
class Point
{
    private double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

• • •

```
Scanner in = new Scanner(System.in);
```

```
String type = in.nextLine();
```

```
double val1 = in.nextDouble();
```

```
double val2 = in.nextDouble();
```

```
Point p1 = new Point(val1, val2);
```

# Let's extend a bit ...


The current implementation considers only **Cartesian coordinate**

What if some user have data in **Polar coordinate** where the input is two doubles: **r** and **theta**.

We know that a polar coordinate can be converted to cartesian coordinate system.

```
x = r*cos(theta)
```

```
y = r * sin(theta)
```



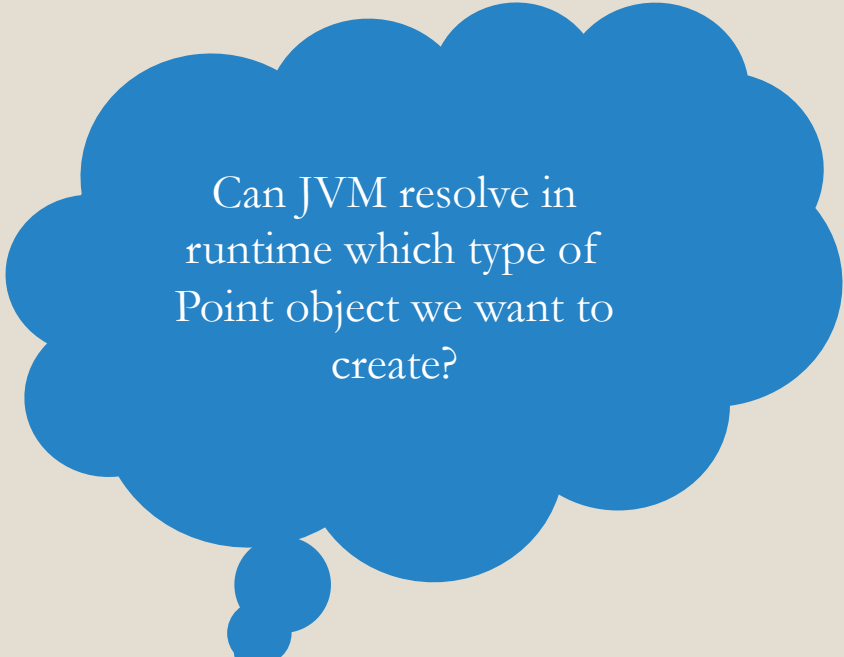
Can we overload a  
constructor to do so  
for us?

# Another Constructor ...

```
class Point
{
    private double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Point(double r, double theta) {
        this.x = r * Math.cos(theta);
        this.y = r * Math.sin(theta);
    }
}
```



Can JVM resolve in runtime which type of Point object we want to create?

```
Scanner in = new Scanner(System.in);

String type = in.nextLine();
double val1 = in.nextDouble();
double val2 = in.nextDouble();
Point p1 = new Point(val1, val2);
```

# What if we ask the client?

```
Scanner in = new Scanner(System.in);
Point p1;
String type = in.nextLine();
double val1 = in.nextDouble();
double val2 = in.nextDouble();

if(type.equals("cartesian"))

    p1 = new Point(val1, val2);

else
    p1 = new Point(
        val1*Math.cos(val2),
        val2* Math.sin(val2));
```



# What if we modify the constructor?

```
/**
 *
 * @param x: x is either a x-coordinate value in cartesian system or a radius
 *           in polar system
 * @param y: y is either a y-coordinate value in cartesian system or an angle
 *           in polar system
 * @param type: a String type, valid entry: cartesian or polar
 */
public Point(double x, double y, String type) {
    if(type.equalsIgnoreCase("cartesian")) {
        this.x = x;
        this.y = y;
    }
    else
    {
        this.x = x * Math.cos(y);
        this.y = x * Math.sin(y);
    }
}
```



Are we happy now?

# What if ...?

```
class AFactory{  
    public static Point CartesianPointFactory(double x, double y) {  
        return new Point(x, y);  
    }  
    public static Point PolarPointFactory(double x, double y) {  
        return new Point(x*Math.cos(y), x*Math.sin(y));  
    }  
}
```

```

class Point
{
    private double x, y;

    private Point(double x, double y) {

        this.x = x;
        this.y = y;

    }

    static class AFactory{

        public static Point CartesianPointFactory(double x, double y) {
            return new Point(x, y);
        }
        public static Point PolarPointFactory(double x, double y) {

            return new Point(x*Math.cos(y), x* Math.sin(y));

        }

    }

}

```

# Client (main) only knows about the two params...

```
Scanner in = new Scanner(System.in);
```

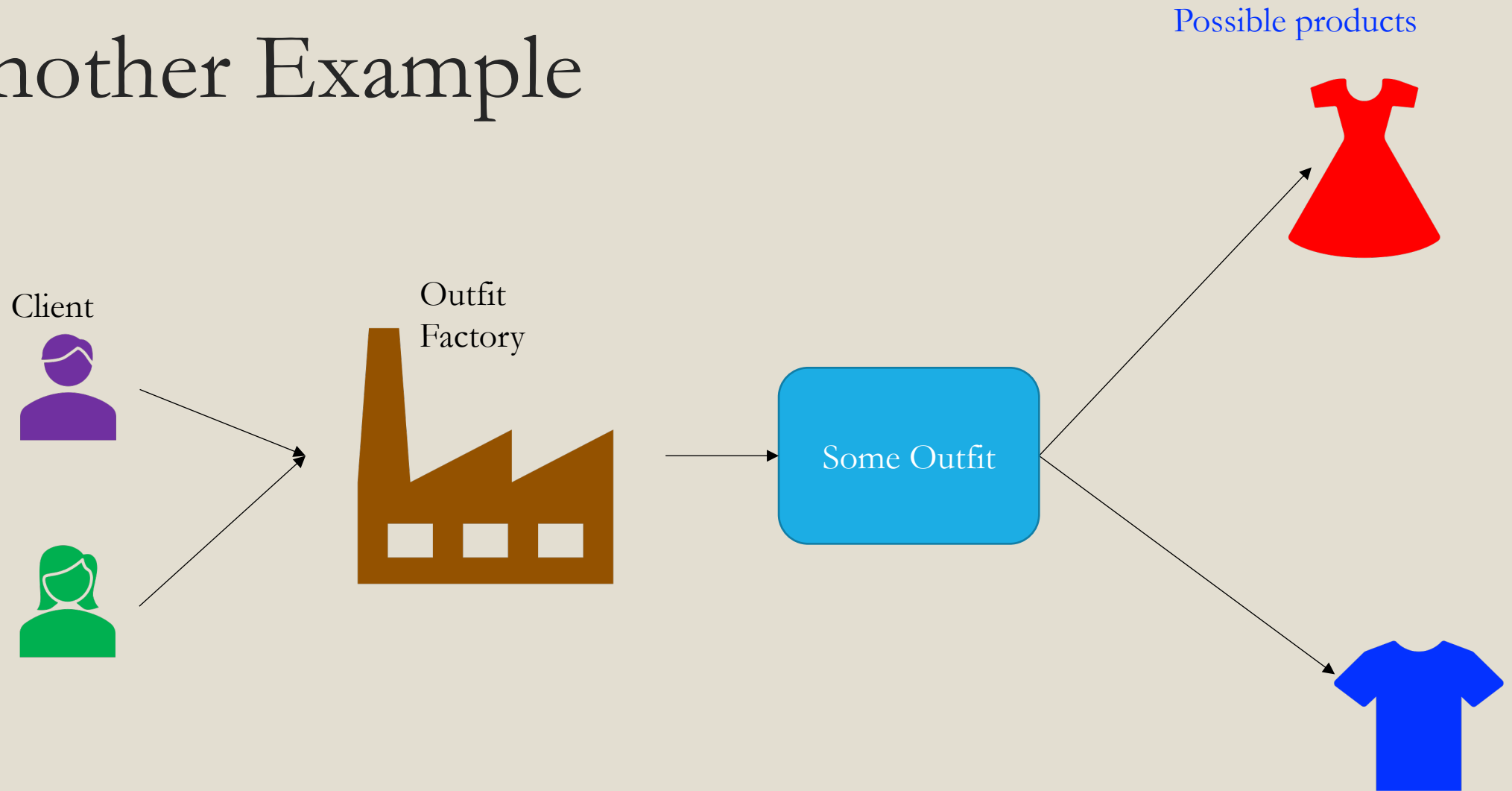
```
String type = in.nextLine();  
double val1 = in.nextDouble();  
double val2 = in.nextDouble();
```

```
//also, how to expand for newer types? The third party has to know about a lot of details  
Point p1 = AFactory.CartesianPointFactory(val1, val2);  
Point p2 = AFactory.PolarPointFactory(val1, val2);
```

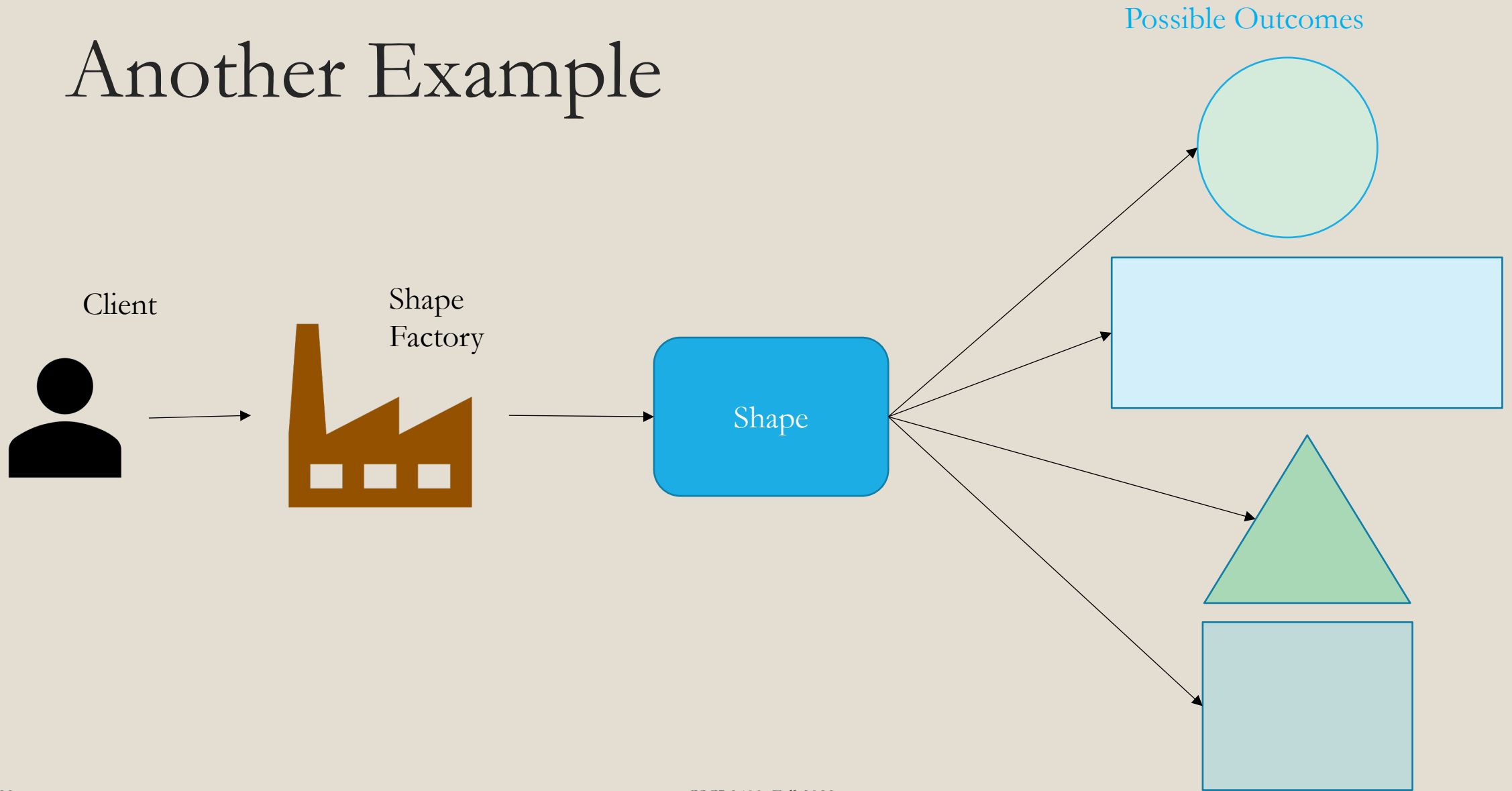
# Another way ...

```
class PointFactory{  
    public Point deliverAPoint(Character c, double x, double y)  
    {  
        if (c == 'C')  
            return new Point (x, y);  
        else if (c == 'P')  
            return new Point (x*Math.cos(y), x* Math.sin(y));  
        else  
            return null;  
    }  
}
```

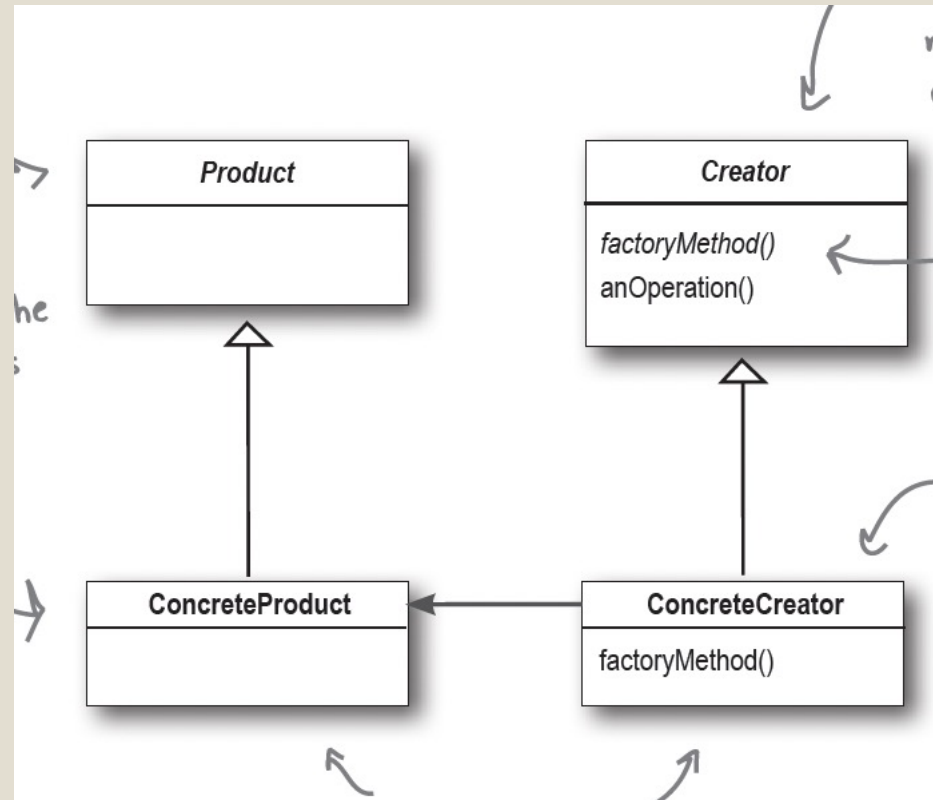
# Another Example



# Another Example



# The Factory Method Pattern





# Summary so far ...

- A simple factory **returns an instance of any one of several possible classes** that have a common parent class.
  - The common parent class can be an abstract class, or an interface.
- The calling program (client: main) typically has a way of telling the factory what it wants, and the factory makes the decision which subclass should be returned to the calling program. It then creates an instance of that subclass, and then returns it to the calling program.
- There are several design patterns that are closely related to the Factory Pattern, including the Factory Method Pattern, and the Abstract Factory.

9/19/23

# CHECK THE SAMPLE CODE ...

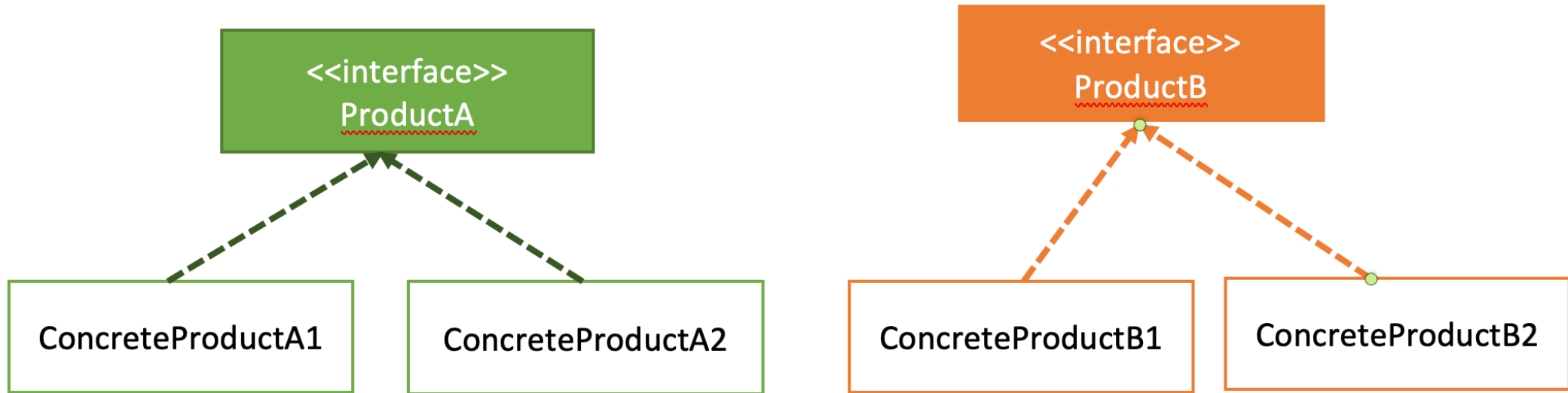
9/19/23

# GIVE 2 MORE EXAMPLES OF FACTORY PATTERN

Think of the applications and benefits of factory pattern

# Another Situation ...

(multiple factories and a combination of deliverable items)



# Situation

- ProductA1 and ProductB1 get along with each other. (can be used to build a good application)
- Similarly, ProductA2 and ProductB2 get along with each other. (can be used to build a good application).
- But (ProductA2, ProductB1) and (ProductA1, ProductB2) are incompatible.
- Can I have a Factory Method Pattern to implement this case?
  - Will that be the best option?

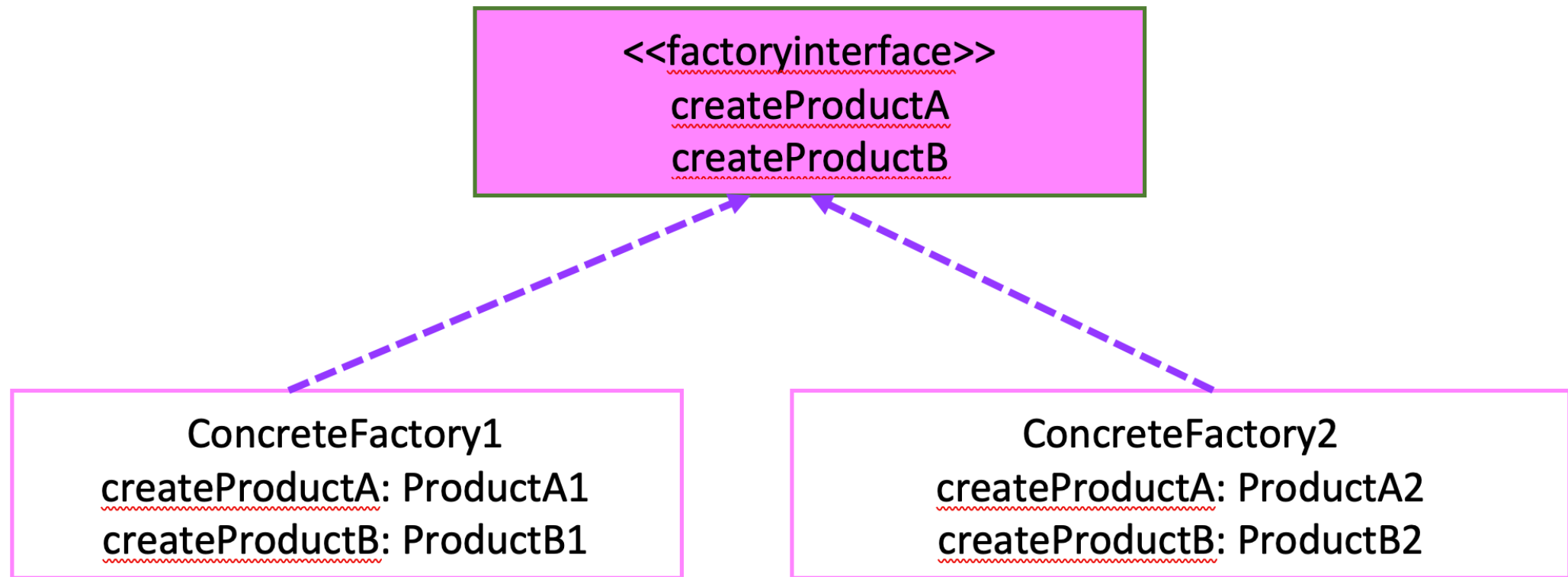


# THE ABSTRACT FACTORY PATTERN

A Creational Design Pattern



# A Factory to deliver ProductA and ProductB together ...



# Client

- The client doesn't want/need to know about this complicated relations.
- All the client wants is to be able to create an app (some object) that can deliver an instance of ProductA and an instance of ProductB.



# Abstract Factory Pattern

// **Abstract Factory** is a creational design pattern, which solves the problem of creating entire product families without specifying their concrete classes.

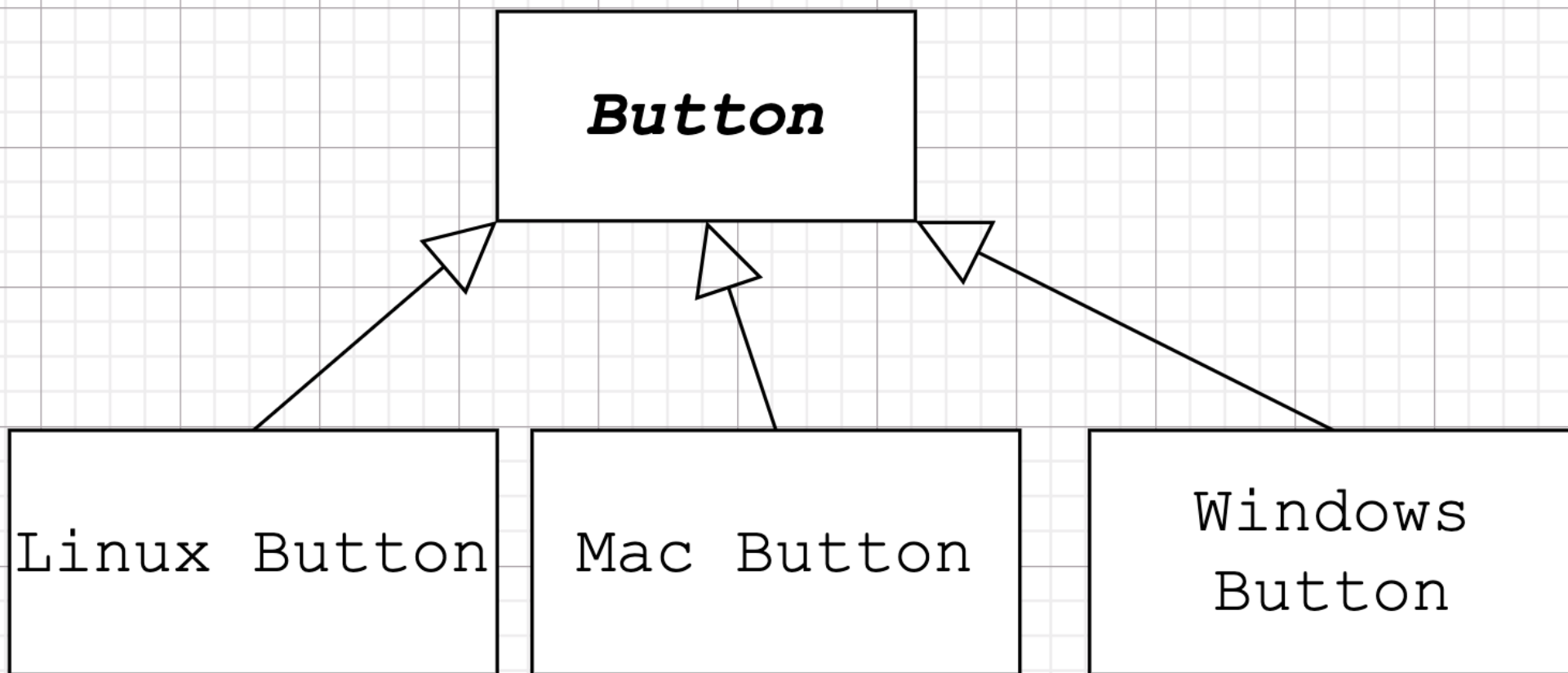
Abstract Factory defines an interface for creating all distinct products but leaves the actual product creation to concrete factory classes. Each factory type corresponds to a certain product variety.

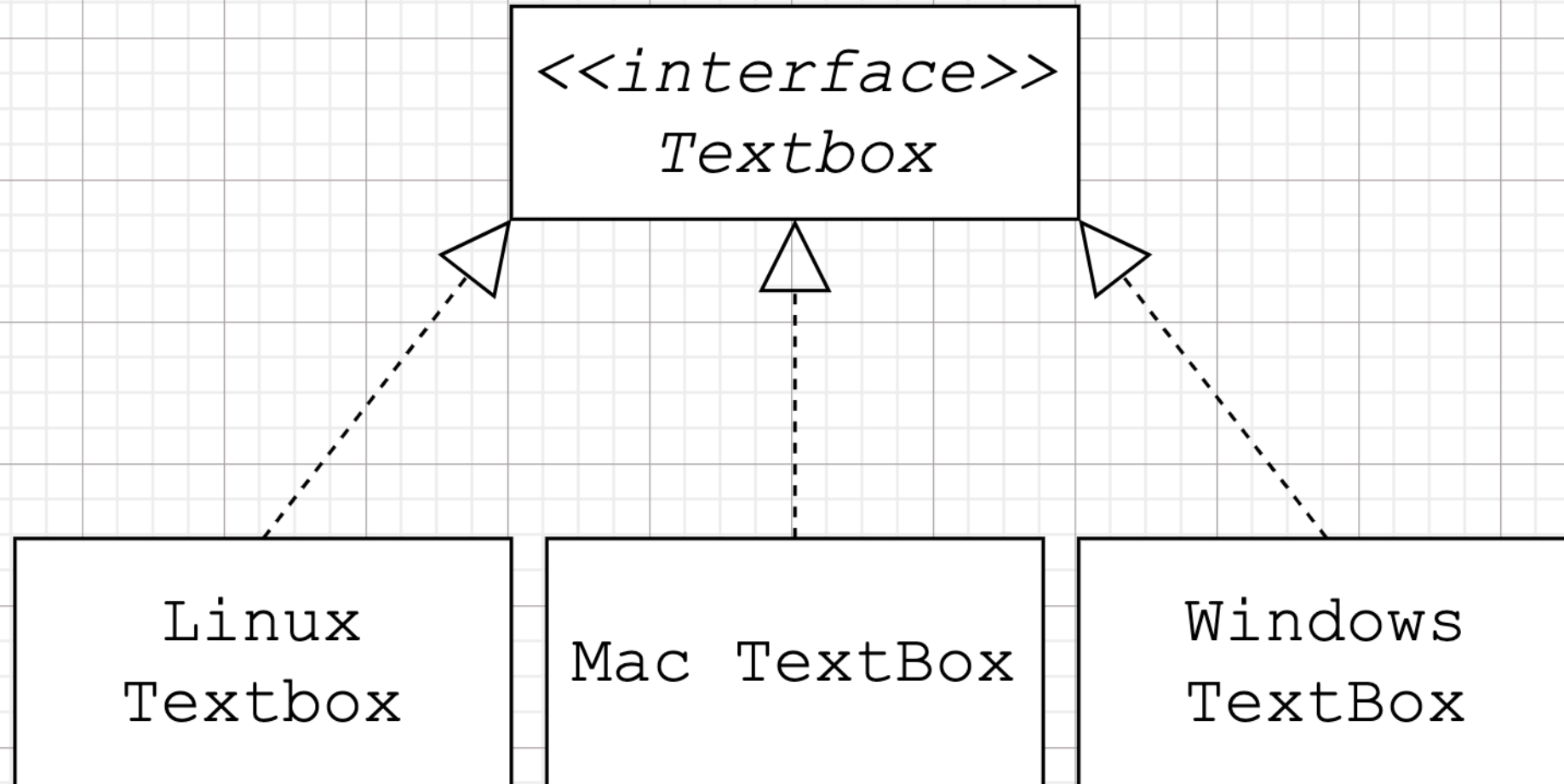
# Abstract Factory Pattern

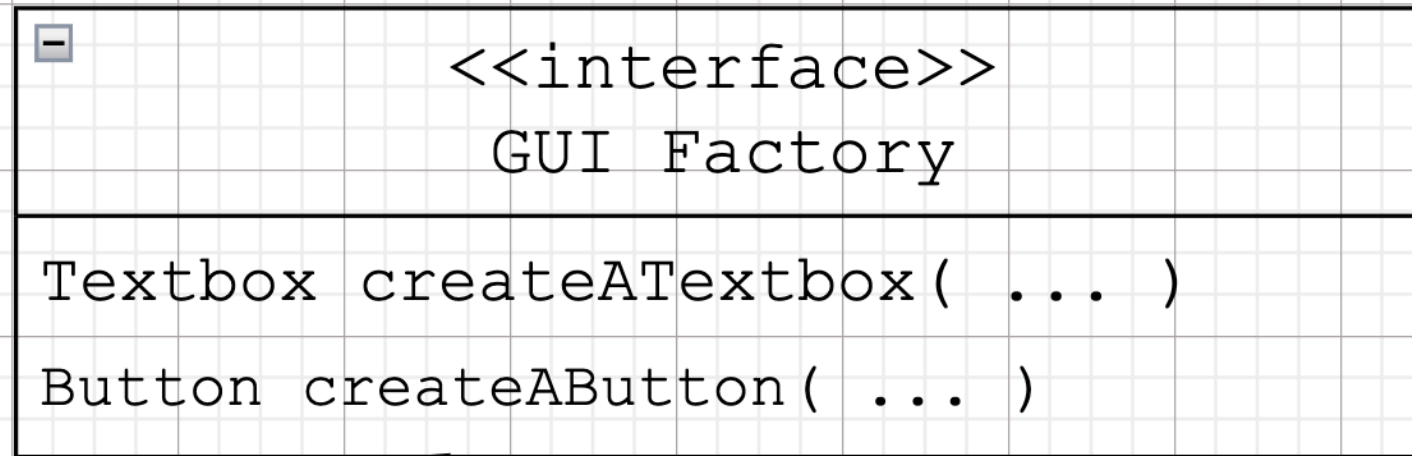
Client code works with factories and products only through their abstract interfaces.

**This lets the client code work with any product variants, created by the factory object.**

You just create a new concrete factory class and pass it to the client code.



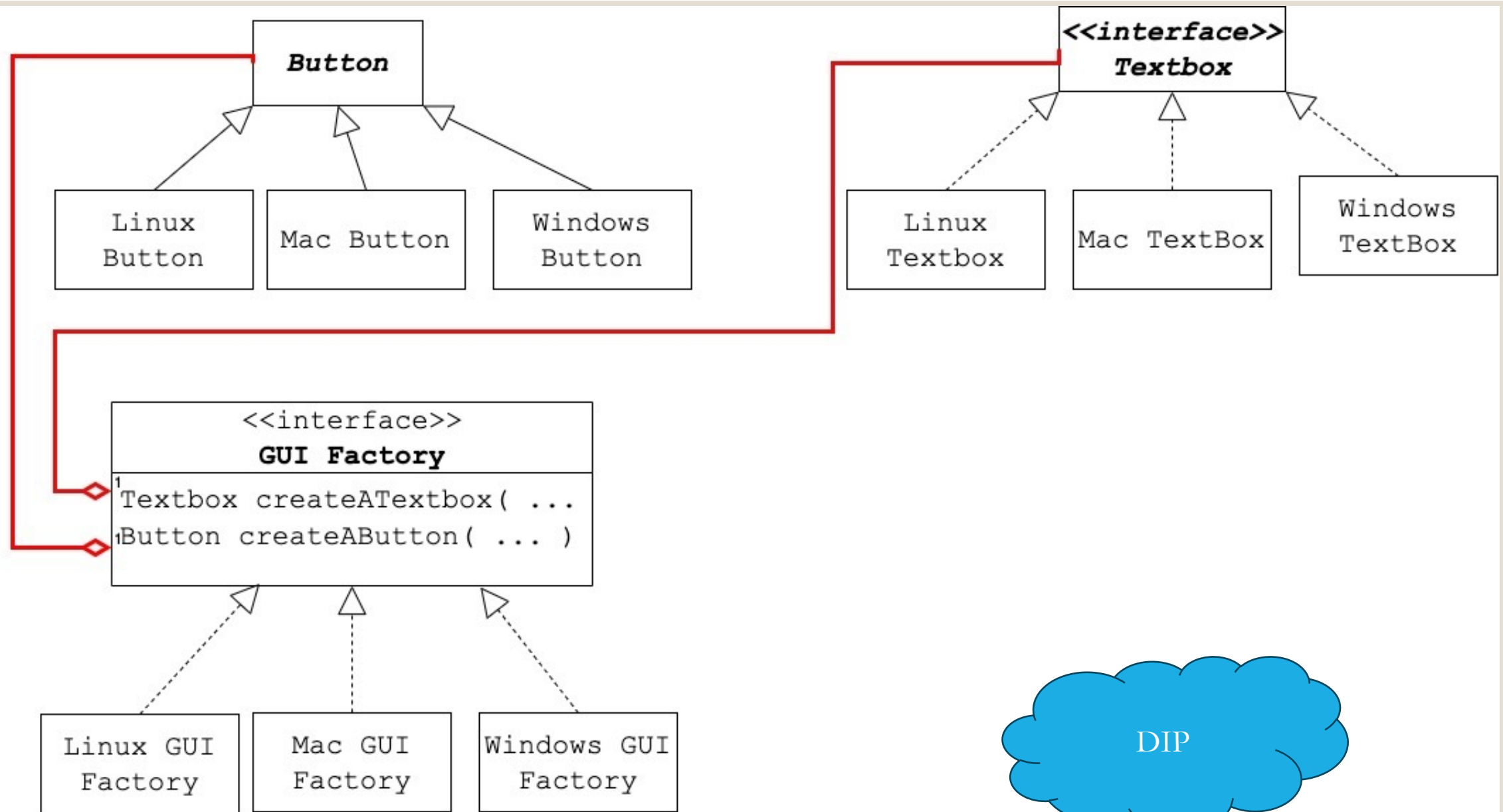




Linux GUI  
Factory

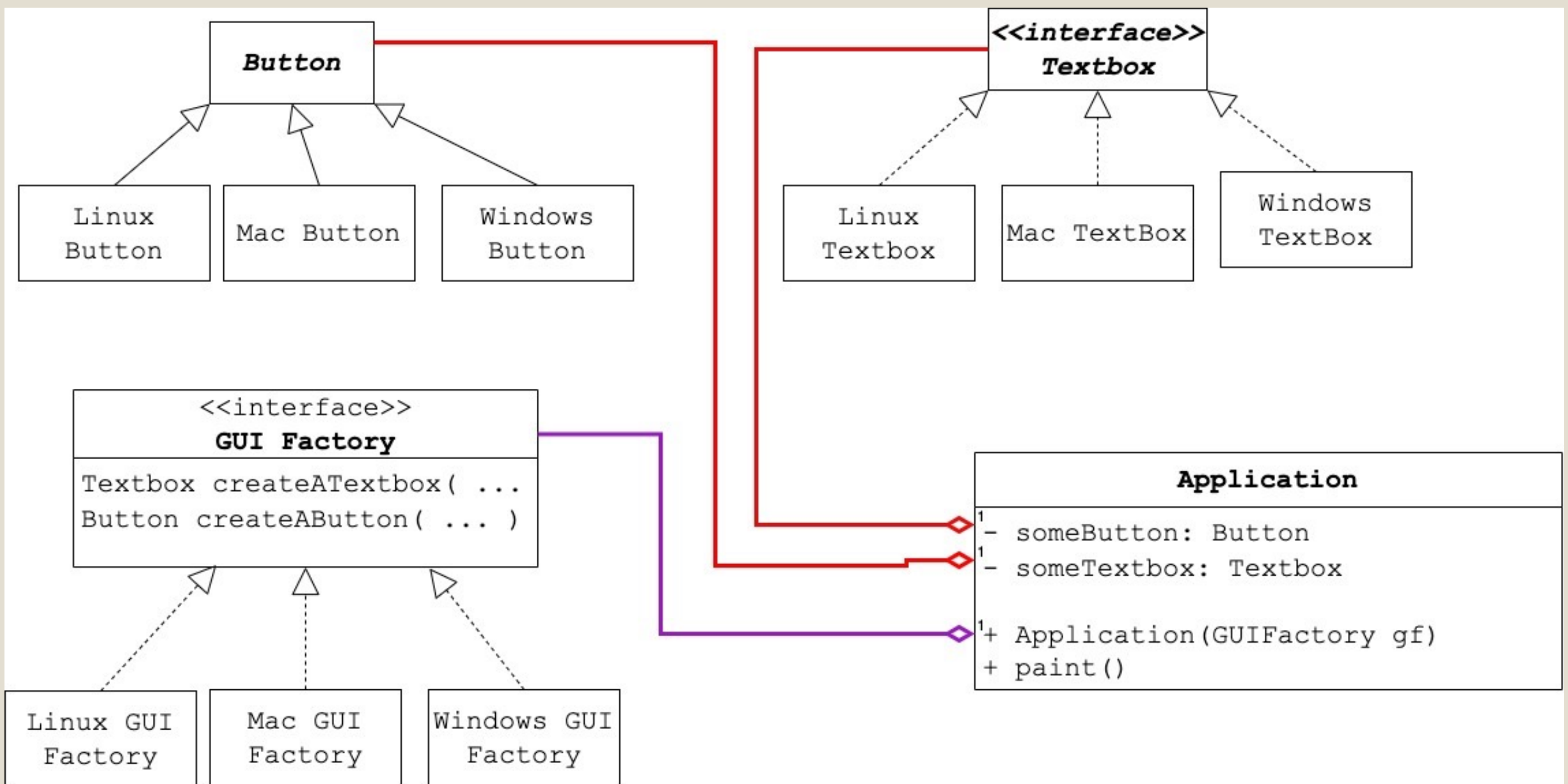
Mac GUI  
Factory

Windows GUI  
Factory



# Our Client: Applicati on

```
public class Application {  
  
    private Button button;  
    private TextBox textbox;  
  
    public Application(GUIFactory factory) {  
        button = factory.createButton();  
        textbox = factory.createTextBox();  
    }  
  
    public void paint() {  
        button.paint();  
        textbox.create();  
    }  
}
```





Time  
to test  
😊

```
private static Application configureApplication(String osName) {  
  
    Application app;  
    GUIFactory factory;  
    //String osName = System.getProperty("os.name").toLowerCase();  
  
    if (osName.contains("mac")) {  
        factory = new MacOSGUI();  
        app = new Application(factory);  
    } else if (osName.contains("linux")) {  
        factory = new LinuxGUI();  
        app = new Application(factory);  
    }  
    else if (osName.contains("windows")) {  
        factory = new WindowsGUI();  
        app = new Application(factory);  
    }  
    else {  
        app = null;  
    }  
    return app;  
}
```

```
public class DemoAbstractFactory{  
    private static Application configureApplication(String osName) {  
  
    public static void main(String[] args) {  
        Application app = configureApplication("MacOS".toLowerCase());  
        app.paint();  
  
        Application app1 = configureApplication("windows".toLowerCase());  
        app1.paint();  
    }  
}
```

### Sample output ...

Creating an app for Mac OS ...  
default is a yellow MacOS button  
This is a MacOS textbox.

default is a blue windows button  
This is a windows textbox.

# Abstract Factory Patterns

- Abstract Factory relies on **object composition**
  - Object creation is implemented in methods exposed in the factory interface.

# Summary so far ...

- A factory method pattern returns an instance of any one of several possible classes that have a common parent class.
- An abstract factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Both of these patterns encapsulate object creation and lead to more decoupled, flexible designs.



# DIP: DEPENDENCY INVERSION PRINCIPLE

# Dependency Inversion Principle

- High-level modules should not depend on low-level modules.
- Both should depend on abstractions (e.g. interfaces).

# A couple of last words ...

- Factories are a powerful technique for coding to abstractions, not concrete classes.

# STUDY ...

- Chapter 4 ....
  - Other resources ...
  - <https://www.gofpatterns.com/design-patterns/module4/intro-creational-patterns.php>
  - <https://alvinalexander.com/java/java-factory-pattern-example/>
  - <https://stackabuse.com/factory-method-design-pattern-in-java/>
- 
- Special thanks to ...
  - <https://refactoring.guru/design-patterns/abstract-factory/java/example>