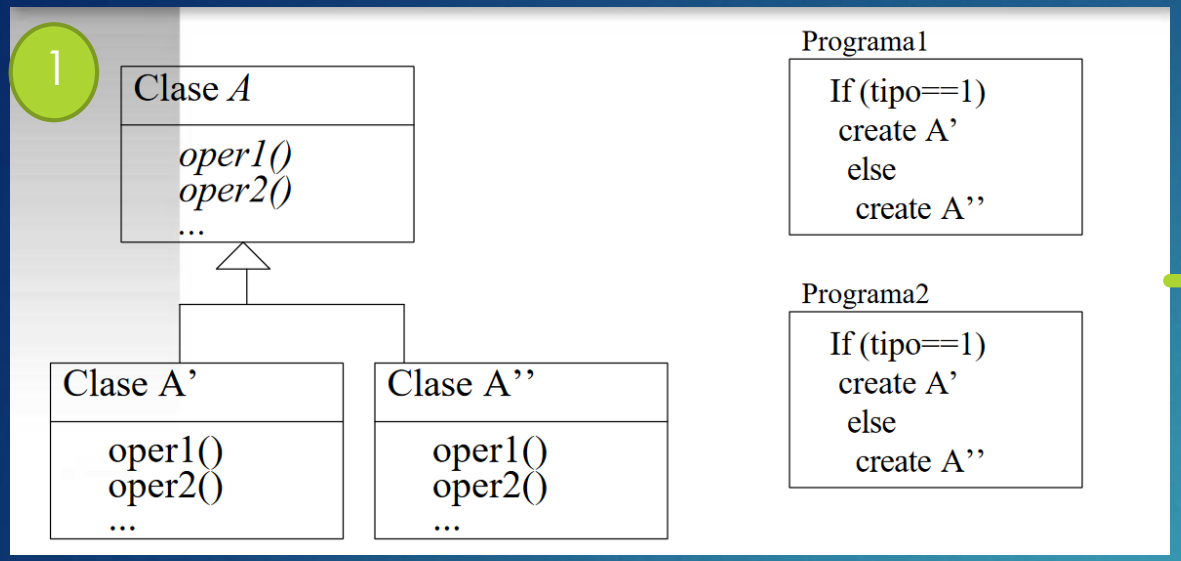
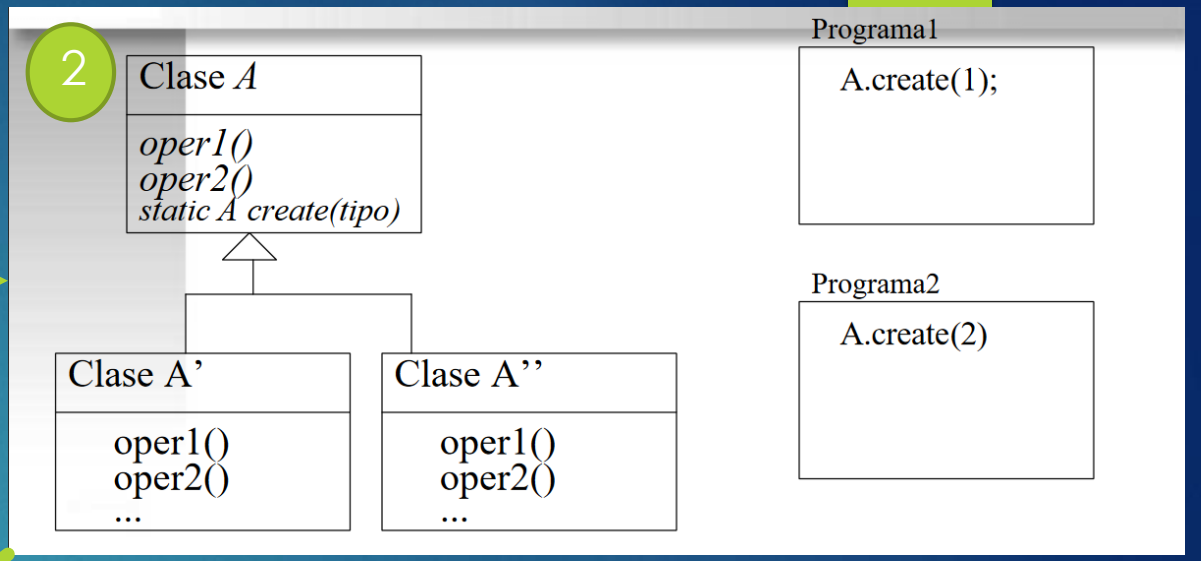
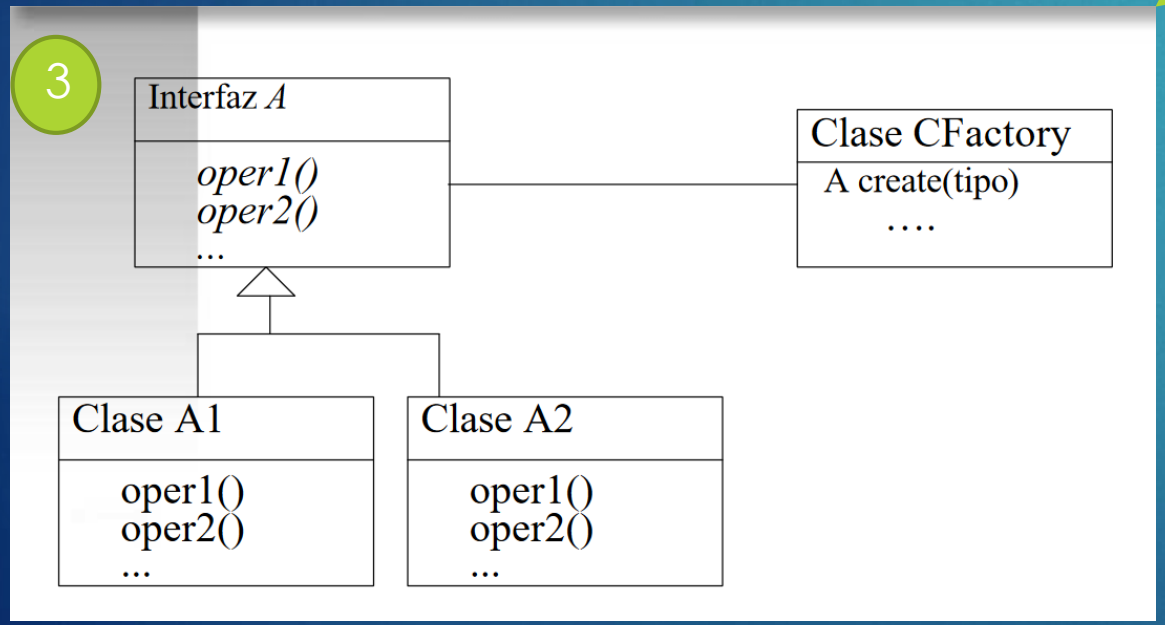
Factory Method

PATRONES DE DISEÑO **TIPO CREACIONAL**.

Integrantes
Gardiola Joaquin
Giachero Ezequiel







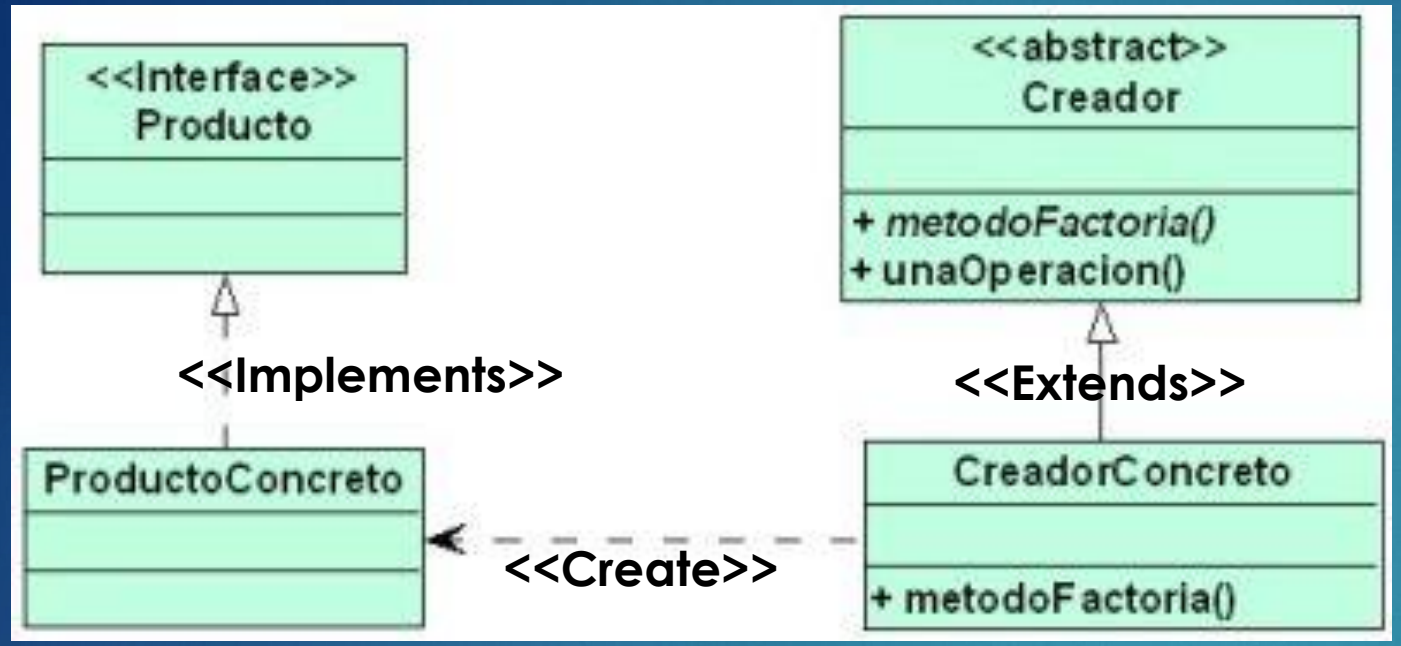
El problema

Qué sucede si queremos añadir A''' en 1)?
Una solución seria recompilar todas las clases que heredan de A y crear un método de creación dentro de A como se muestra en 2), pero puede que no se tenga acceso al código de esta clase.
La solución final a este problema la brinda el patrón Factory Method, donde se separa el creador de instancias de la propia clase. Estas instancias se crean en la clase CFactory como se muestra en 3).

Propósito y Usabilidad

- Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan cual clase instanciar. Es decir, Factory Method delega la creación de objetos a las subclases.
- Este método debe usarse cuando:
  - Cuando una clase no puede anticipar la clase de objetos que debe crear.
  - Cuando se desea que las subclases especifique el objeto que se crea.
  - Cuando una clase delega la responsabilidad a una de las subclases auxiliares y se quiere saber cual es la subclase delegada.

Estructura.



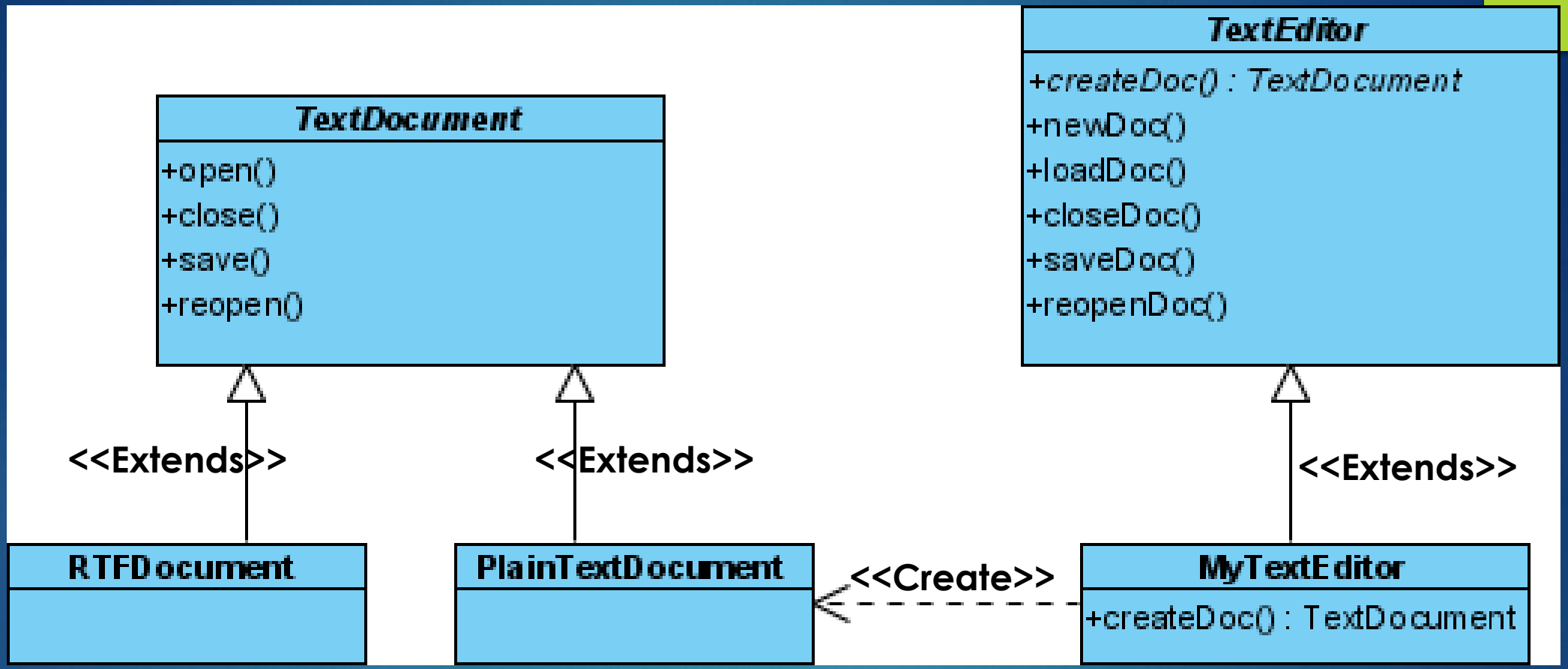
Producto: Interfaz de los objetos que crea el método de fabricación

ProductoConcreto: Implementa la interfaz "Producto"

Creador: Declara el método de fabricación, el cual devuelve un objeto del tipo Producto. Puede llamar al método de fabricación para crear un objeto Producto

CreadorConcreto: Redefine el método de fabricación para devolver una instancia de un ProductoConcreto

Ejemplo



NOTA: Si no existiera la clase TextEditor con los métodos de creación, y estos estuvieran incluidos en la clase TextDocument, si quisiéramos añadir otro tipo de TextDocument habría que recompilar todas las clases que heredan de esta para realizar la modificación de la superclase y puede que incluso no se tenga acceso al código de esta. La solución a esto es separar el creador de las instancias de la propia clase, como lo indica el Factory Method

```java
public class TrianguloFactory implements TrianguloFactoryMethod {

    public Triangulo createTriangulo (int ladoA, int ladoB, int ladoC){

        if ((ladoA == ladoB) && (ladoA == ladoC)){
            return new Equilatero(ladoA, ladoB, ladoC);
        }
        else if ((ladoA != ladoB) && (ladoA != ladoC) && (ladoB != ladoC)){
            return new Escaleno(ladoA, ladoB, ladoC);
        }
        else{
            return new Isoceles(ladoA, ladoB, ladoC);
        }
    }
}
```

```java
public interface TrianguloFactoryMethod {
    public Triangulo createTriangulo (int ladoA, int ladoB, int ladoC);
}
```

```java
public abstract class Triangulo {
    private int ladoA;
    private int ladoB;
    private int ladoC;

    public Triangulo(int ladoA, int ladoB, int ladoC){
        setLadoA(ladoA);
        setLadoB(ladoB);
        setLadoC(ladoC);
    }

    //Escribe el tipo de triangulo que es.
    public abstract String getDescripcion();

    //Definiciones de Get y Set para cada Lado

    public int getLadoA(){
        return ladoA;
    }

    public void setLadoA(int aux){
        ladoA = aux;
    }

    public int getLadoB(){
        return ladoB;
    }

    public void setLadoB(int aux){
        ladoB = aux;
    }

    public int getLadoC(){
        return ladoC;
    }

    public void setLadoC(int aux){
        ladoC = aux;
    }
}
```

```java
public class Escaleno extends Triangulo{

    publ

    }

    publ

    }
}
```

```java
public class Isoceles extends Triangulo{

    public Isoceles (int ladoA, int ladoB, int ladoC) {

    }

    publ

    }
}
```

```java
public class Equilatero extends Triangulo{

    public Equilatero (int ladoA, int ladoB, int ladoC) {
        super(ladoA, ladoB, ladoC);
    }

    public String getDescripcion() {
        return "Soy un Triangulo Equilatero";
    }

}
```

```java
public class Main {
    public static void main(String[] args){
        TrianguloFactoryMethod factory = new TrianguloFactory();
        Triangulo triangulo = factory.createTriangulo(Integer.parseInt(args[0]),
        Integer.parseInt(args[1]),Integer.parseInt(args[2]));
        System.out.println(triangulo.getDescripcion());

    }
}
```

Consecuencias

- Elimina la necesidad de incluir clases especificas de la aplicación en código mas general (potencial reutilización del framework).
- Conecta jerarquías paralelas (una clase delega alguna de sus responsabilidades a otra clase aparte).
- La creación de objetos a través de Factory Method es mas flexible.
- El cliente debe crear subclases de la clase Creator para cada tipo de producto concreto.