




FACTORY PATTERN

David Cerdas Alvarado C02001
Ignacio Robles Mayorga B96549



FACTORY METHOD

Permite que una clase
difiera la creación de sub
clases.

Por lo que a nivel de
código solo se trabaja con
la interfaz o clase
abstracta



Also known as

Virtual Creator





CLASE INTERFAZ

Clase Interface

```
public interface IAnimal {  
    public String hacerSonido();  
  
    public boolean nacer();  
  
    public void mover();  
  
    public boolean respirar();  
}
```


Clase Interface



```
public class Perro implements Animal {  
    public String hacerSonido() {  
        return "Guau!";  
    }  
}
```

Clase Interface



```
public class Gato implements Animal {  
    public String hacerSonido() {  
        return "Miau!";  
    }  
}
```


Clase Interface

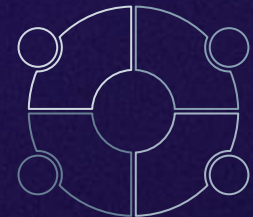
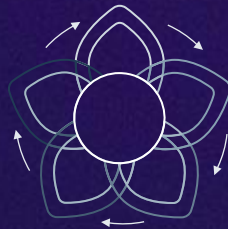
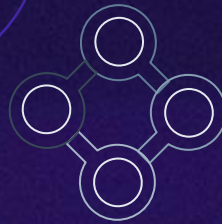


```
public static void main(String[] args) {  
    Animal perro = new Perro();  
    System.out.println(perro.hacerSonido()); // Salida: "Guau!"  
  
    Animal gato = new Gato();  
    System.out.println(gato.hacerSonido()); // Salida: "Miau!"  
}
```


¿Cuándo usar Factory Pattern?

Es una vía fácil de implementar el
clase u objetos que cumplen con la
definición:

Similar, pero no el mismo



Aplicaciones



Anticipación

Cuando la clase no puede anticipar la clase de objetos que debe crear
[Ejemplo Globos]



Especificación

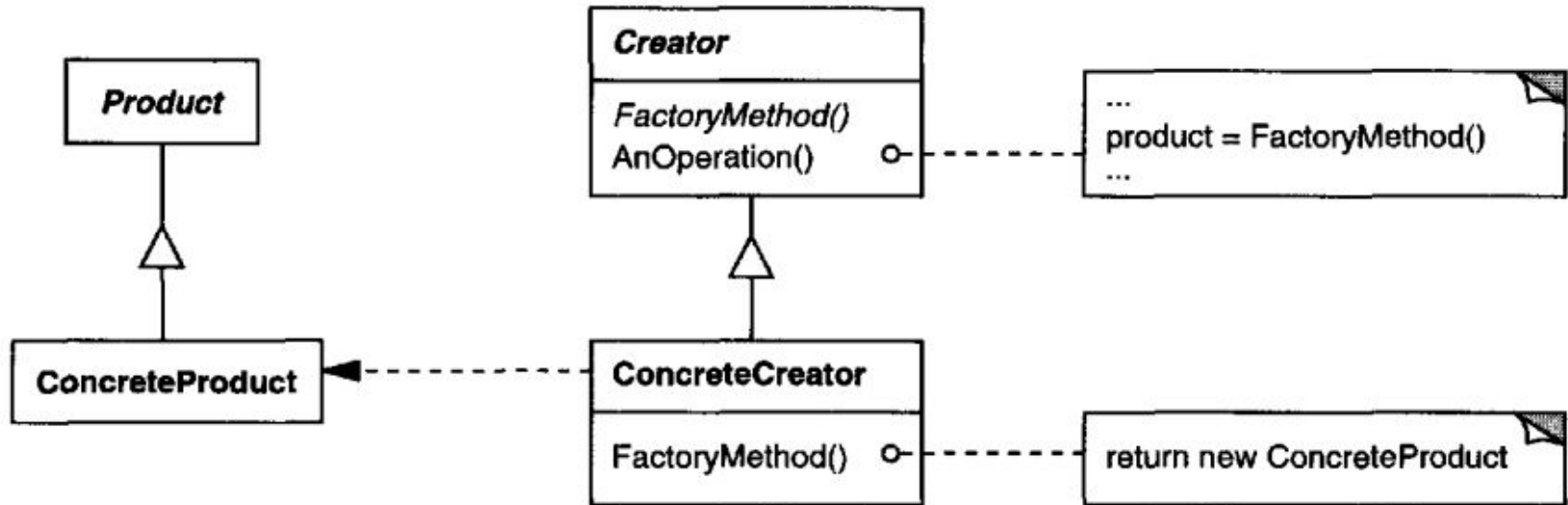
Cuando una clase quiere sus subclases para especificar los objetos que crea
[Ejemplo Receta]



Delegar tareas

Cuando las clases delegan la responsabilidad a subclases auxiliares y desea saber cuál clase auxiliar es la delegada
[Ejemplo Asistente]

Estructura



Product



Creator



ConcreteProduct



ConcreteCreator

Animals
Factory



Ejemplo #1



Resources
Factory



Ejemplo #2



Laboratorio: Arquitecturas



Proyecto



Foro de consultas



Referencias



Carta al estudiante

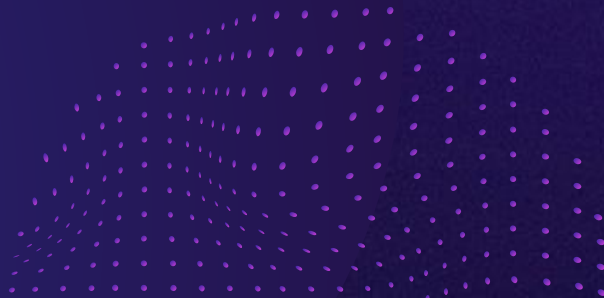


Ejemplos Fibonacci y Factorial



IMPLEMENTACIÓN

A la hora de implementar una aplicación con un patrón de diseño factory method, se deben tomar en cuenta las siguientes consideraciones



HAY DOS VARIANTES DE IMPLEMENTACIÓN

CREATOR ABSTRACTO

```
class MazeGame {
public:
    Maze* CreateMaze();

    // factory methods:
    virtual Maze* MakeMaze() const;

    virtual Room* MakeRoom(int n) const;

    virtual Wall* MakeWall() const;

    virtual Door* MakeDoor(Room* rl, Room* r2) const;
}
```

CREATOR CON MÉTODOS DEFINIDOS

```
class MazeGame {
public:
    Maze* CreateMaze();

    // factory methods:
    virtual Maze* MakeMaze() const
    { return new Maze; }

    virtual Room* MakeRoom(int n) const
    { return new Room(n); }

    virtual Wall* MakeWall() const
    { return new Wall; }

    virtual Door* MakeDoor(Room* rl, Room* r2) const
    { return new Door(rl, r2); }
}
```

FACTORY METHODS CON PARAMETROS

```
Question* Trivia::createQuestion(const std::string& type) {  
    if (type == "textual") {  
        return new TextualQuestion();  
    } else if (type == "numeric") {  
        return new NumericQuestion();  
    } else {  
        return nullptr; // opción por defecto  
    }  
}
```


VARIANTES Y PROBLEMAS DEPENDIENDO DEL LENGUAJE





TEMPLATES VS SUBCLASSES

```
class Creator {  
    public:  
    virtual Product* CreateProduct() = 0 ;  
};
```

```
template <class TheProduct>  
class StandardCreator: public Creator {  
    public:  
    virtual Product* CreateProduct();  
};
```

```
template <class TheProduct>  
    Product* StandardCreator::CreateProduct () {  
    return new TheProduct;  
}
```


CONVENCIONES DE NOMBRADO

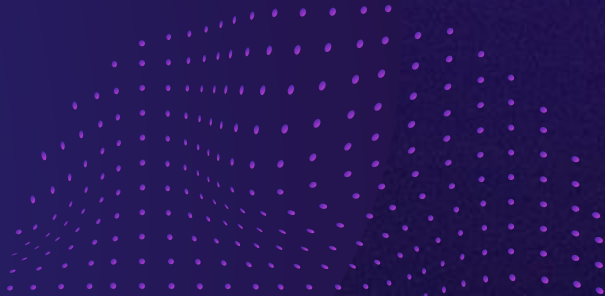
CREAR (create) + RECURSO (resource) = createResource()





EJEMPLO

Supongamos que se quiere crear un juego llamado Maze Game. Este tiene varios elementos que lo componen (Maze, Room, Wall, Door). Ahora bien, también existen varias versiones de este juego, y con esto, también varias versiones de sus elementos.



CREADOR Y PRODUCTOS



MazeGame.cpp

```
class MazeGame {  
public:  
    Maze* CreateMaze();  
    // factory methods:  
    virtual Maze* MakeMaze() const  
    { return new Maze; }  
    virtual Room* MakeRoom(int n) const  
    { return new Room(n); }  
    virtual Wall* MakeWall() const  
    { return new Wall; }  
    virtual Door* MakeDoor(Room* r1, Room* r2) const  
    { return new Door(r1, r2); }  
};
```

CREADOR CONCRETO Y PRODUCTOS CONCRETOS

```
BombedMazeGame.cpp

class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const {
        return new BombedWall;
    }

    virtual Room* MakeRoom(int n) const {
        return new RoomWithABomb(n);
    }
};
```


CONSECUENCIAS

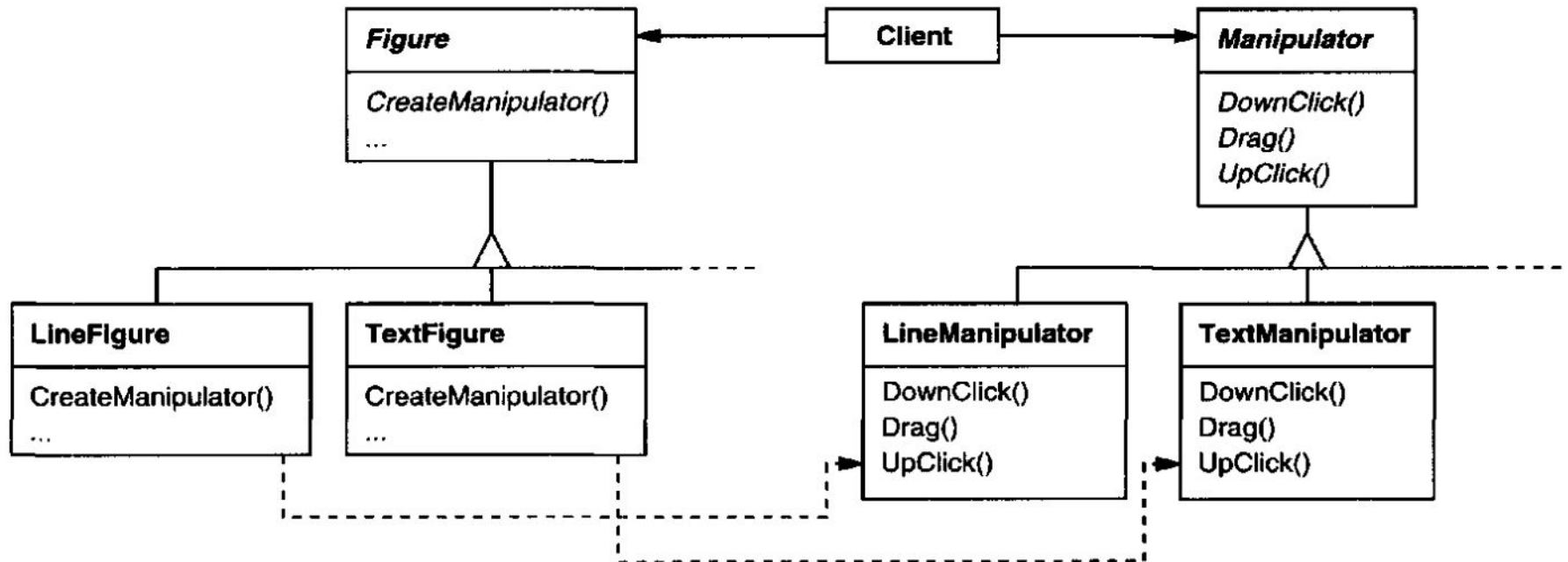


- Necesidad de crear una clase extra por cada producto concreto (si es que no se planeaba crear una de antemano)



- No hay necesidad de mucha lógica de ifs
- Más flexibilidad de código
- Se conectan jerarquías de clases paralelas

JERARQUÍA DE CLASES PARALELAS





PATRONES RELACIONADOS

Abstract factory

Usualmente es implementado con factory methods.

Template methods

Se encuentran factory methods dentro de templates



Gracias!

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon** and infographics & images by **Freepik**

Bibliografía

Gamma, Erich et.al. “Design Patterns”. Addison-Wesley, 1995.

Refactoring.Guru. (2023). Factory Method. *Refactoring.Guru*.

<https://refactoring.guru/design-patterns/factory-method>

Actividad de la presentación

Los pasos para la actividad son los siguientes:

1. Reúnanse en los grupos de proyecto, tengan UNA hoja blanca y lápiz
2. En los grupos, creen un diagrama de clases acerca de cómo implementarían "Factory Pattern" en la creación de recursos de Mediación virtual. (Solo los componentes vistos en la presentación).
3. Este primer diagrama debe ser utilizando ConcreteCreators
4. Ahora, haga lo mismo pero sin usar los ConcreteCreators (pista: asuman que el factory method utiliza parametrización)
5. El primer grupo en hacer ambos diagramas de forma correcta se gana una bolsa de chocolates 🍫🍫.