



UNIVERSIDAD
Popular del Cesar

Ingeniería de Sistemas

Patrones de Diseño de Software

Profesor: Deivis Martínez Acosta



Agenda

Patrones Creacionales

Singleton

Factory Method

Abstract factory

Prototype


Builder



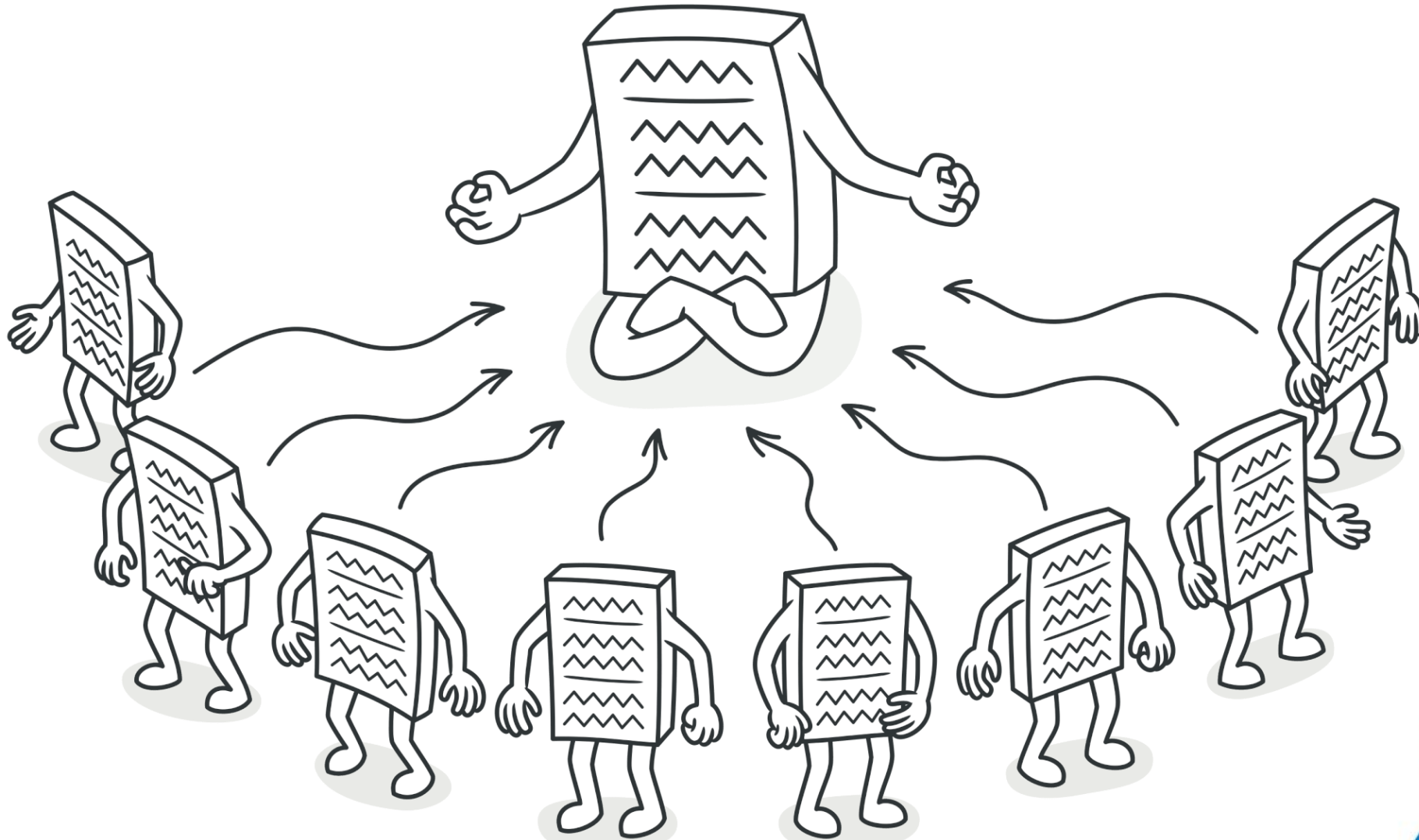


Patrones Creacionales

Como habíamos mencionado los patrones creacionales nos facilitan la creación de nuevos objetos de una forma que podamos cumplir el principio de bajo acoplamiento. Vamos a analizar algunos de ellos.



Singleton



Singleton

Este patrón también se le conoce con el nombre de instancia única, lo utilizamos para no permitir que existan múltiples objetos de una clase sin necesidad, sino solamente una. Es fácil de implementar en el código y de diseñar en UML.

Singleton

Permite tener una única instancia para ser utilizado durante toda la aplicación, esto nos permite tener un mayor rendimiento en memoria del procesador. No se encarga de la creación de objetos en sí, sino que se enfoca en la restricción en la creación de un objeto.

Singleton

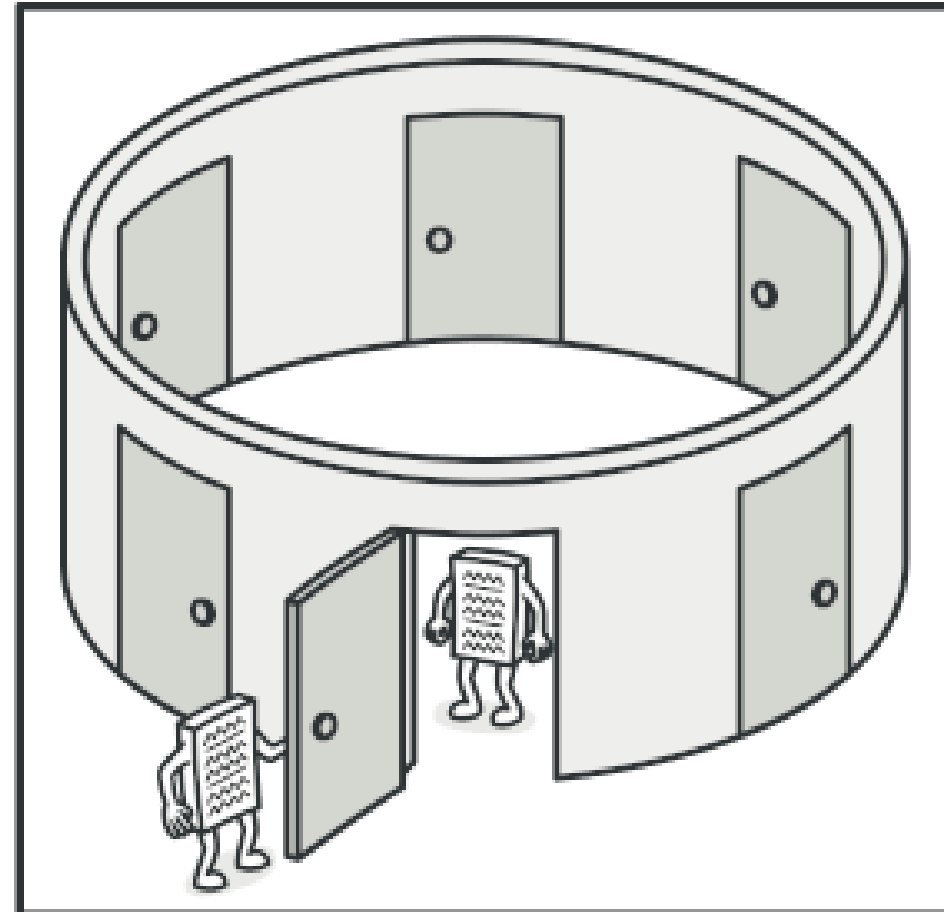
Reduce el espacio de nombres.

El patrón es una mejora sobre las variables globales. Ya no se reservan nombres para las variables globales, ahora solo existen instancias.

Singleton

Controla el acceso a la instancia única, porque la clase Singleton encapsula la única instancia. Así se obtiene control sobre cómo y cuándo se accede a ella. Permite el refinamiento de las operaciones y la representación.

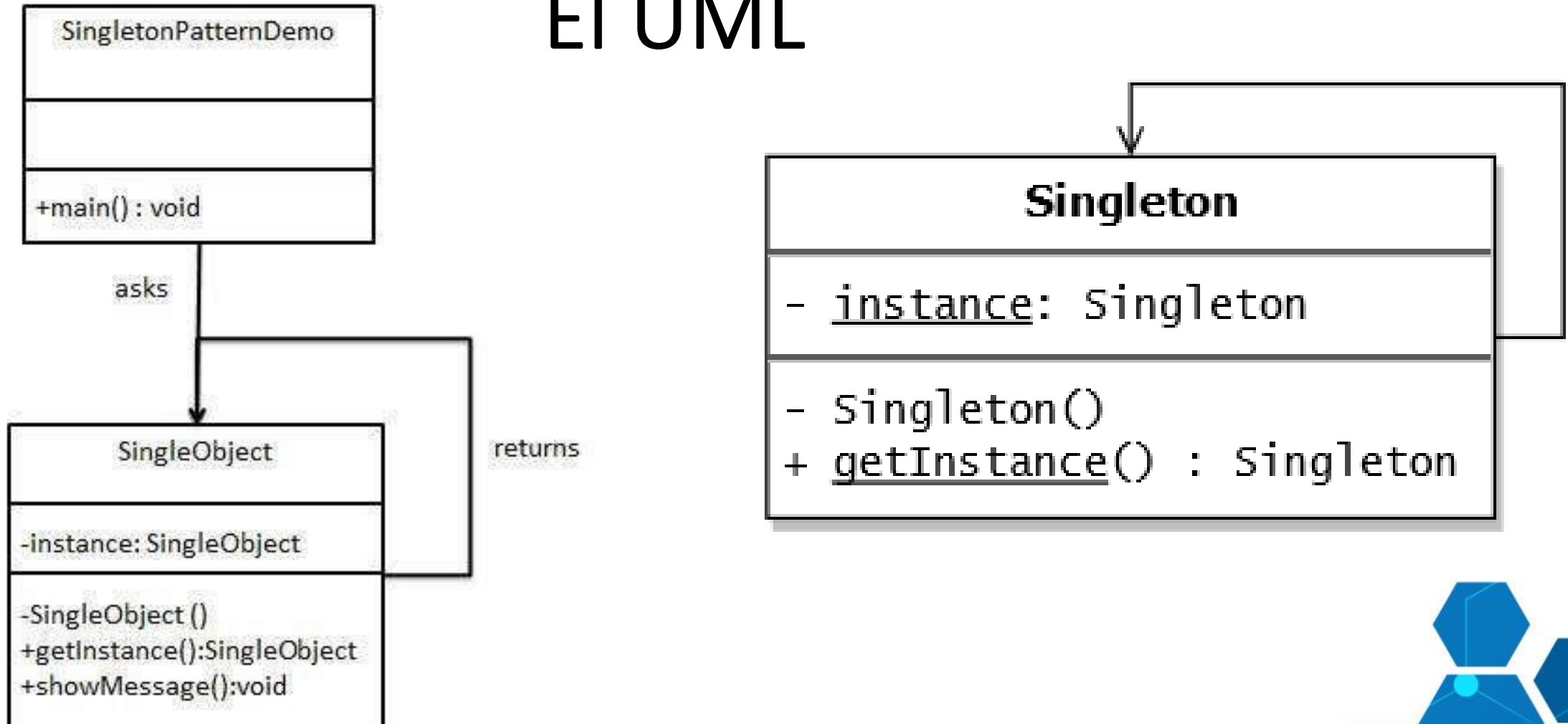
Singleton



<https://refactoring.guru/es/design-patterns/>

Singleton

El UML

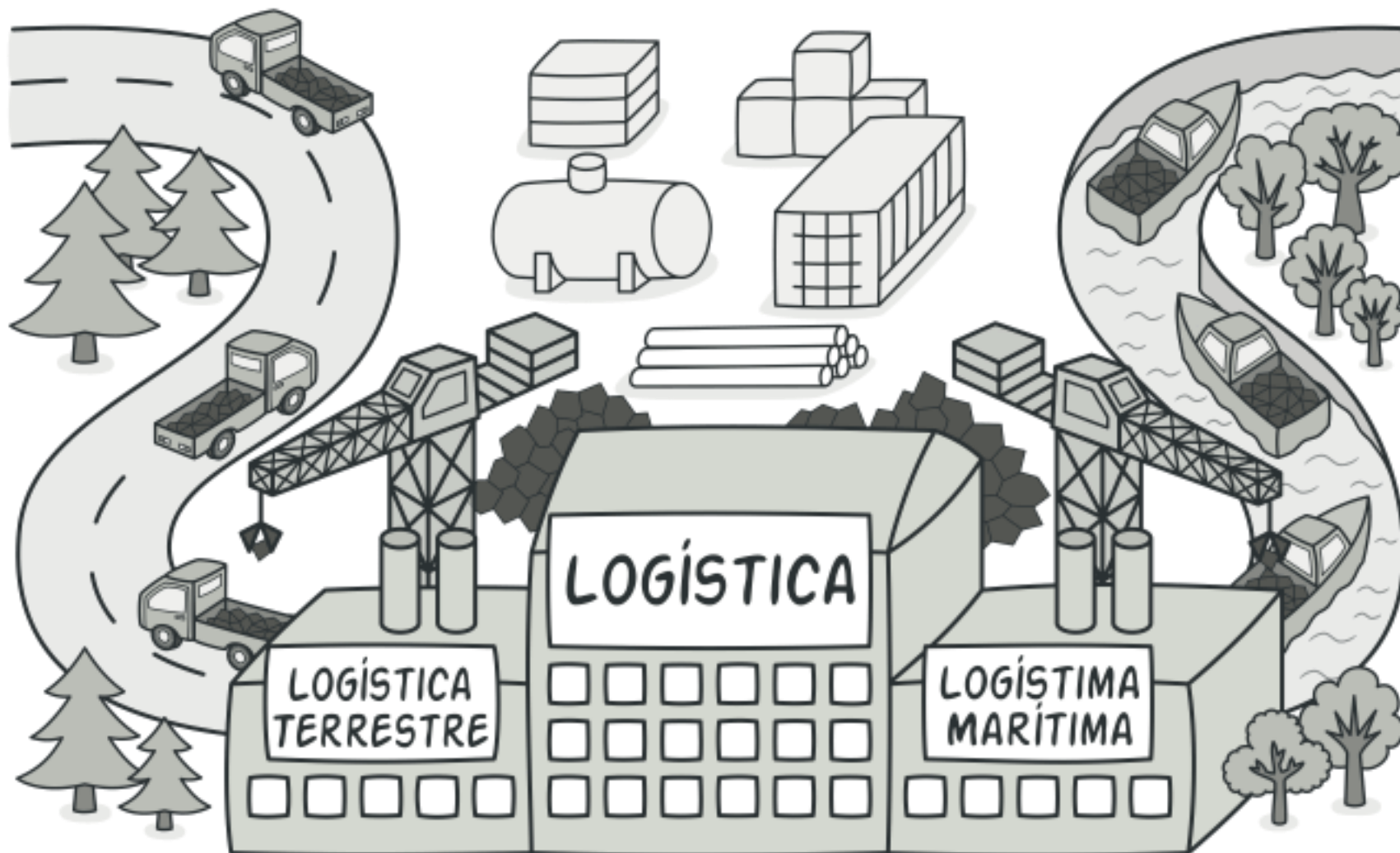


Singleton

```
public class Singleton {  
    private static Singleton objetoUPC;  
    private Singleton() {}  
  
    public static Singleton getInstancia() {  
        if(objetoUPC == null) {  
            objetoUPC = new Singleton();  
            System.out.println("Creamos el objeto en la especialización");  
        } else {  
            System.out.println("El mismo objeto ...");  
        }  
        return objetoUPC;  
    }  
}
```

El Código fuente


Factory Method





Factory Method


Este patrón permite la creación de una clase que fungirá como fabrica de objetos y los retornará al módulo, clase u objeto que se lo solicite, para ello hace uso del polimorfismo e interfaces.





Factory Method


Factory Method permite la creación de objetos de un subtipo determinado a través de una clase Factory. Esto es especialmente útil cuando no sabemos, en tiempo de diseño, el subtipo que vamos a utilizar o cuando queremos delegar la lógica de creación de los objetos a una clase Factory.





Factory Method

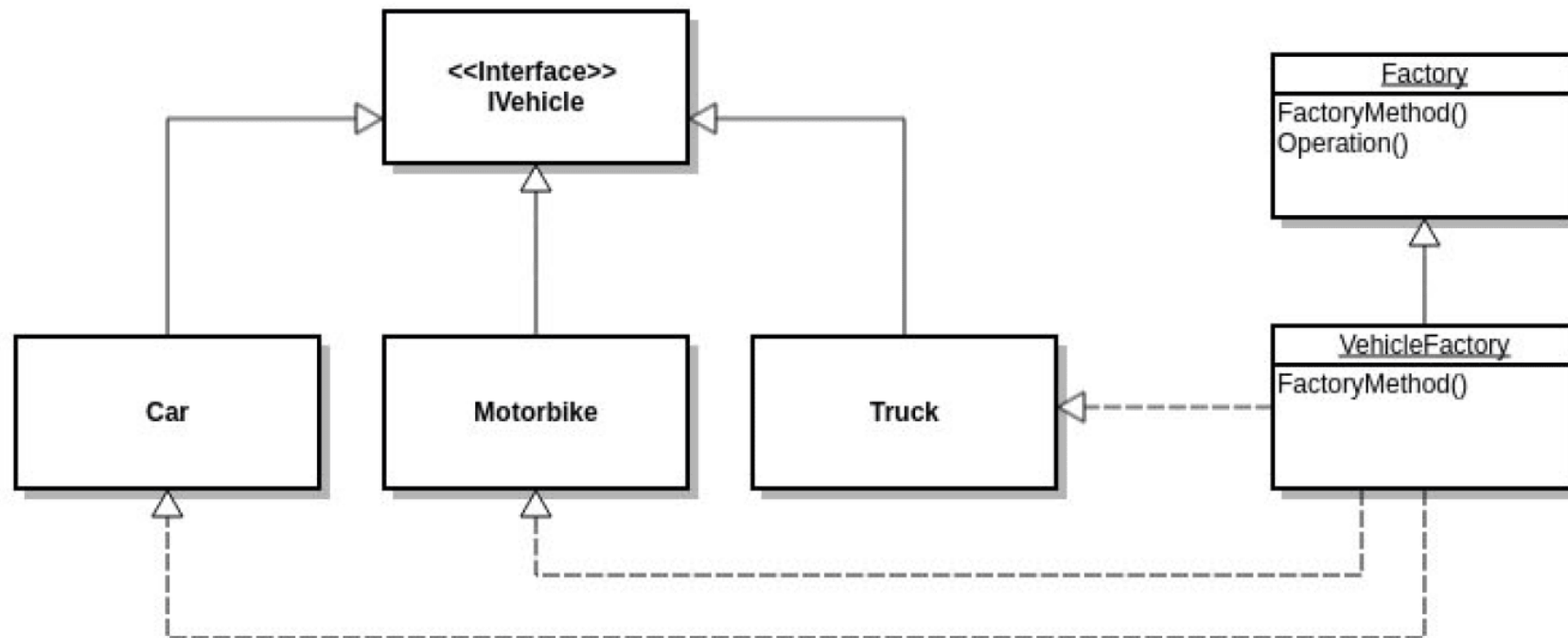
Utilizando este patrón podemos crear instancias dinámicamente mediante la configuración, estableciendo cual será la implementación a utilizar en un archivo de texto, XML, properties o mediante cualquier otra estrategia.



Factory Method

El UML

How to use Factory Method Design Pattern in C#



Factory Method

El Código

```
public interface IFigura {  
    double calcularArea(double valor, double valor1);  
}  
  
public class Triangulo implements IFigura{  
  
    @Override  
    public double calcularArea(double base, double altura) {  
        return base * altura / 2;  
    }  
}
```

Factory Method

El Código

```
public class Rectangulo implements IFigura{

    @Override
    public double calcularArea(double base, double altura) {
        return base * altura;
    }
}

public class OtraFigura implements IFigura{

    @Override
    public double calcularArea(double base, double altura) {
        return 0;
    }
}
```

Factory Method

El Código

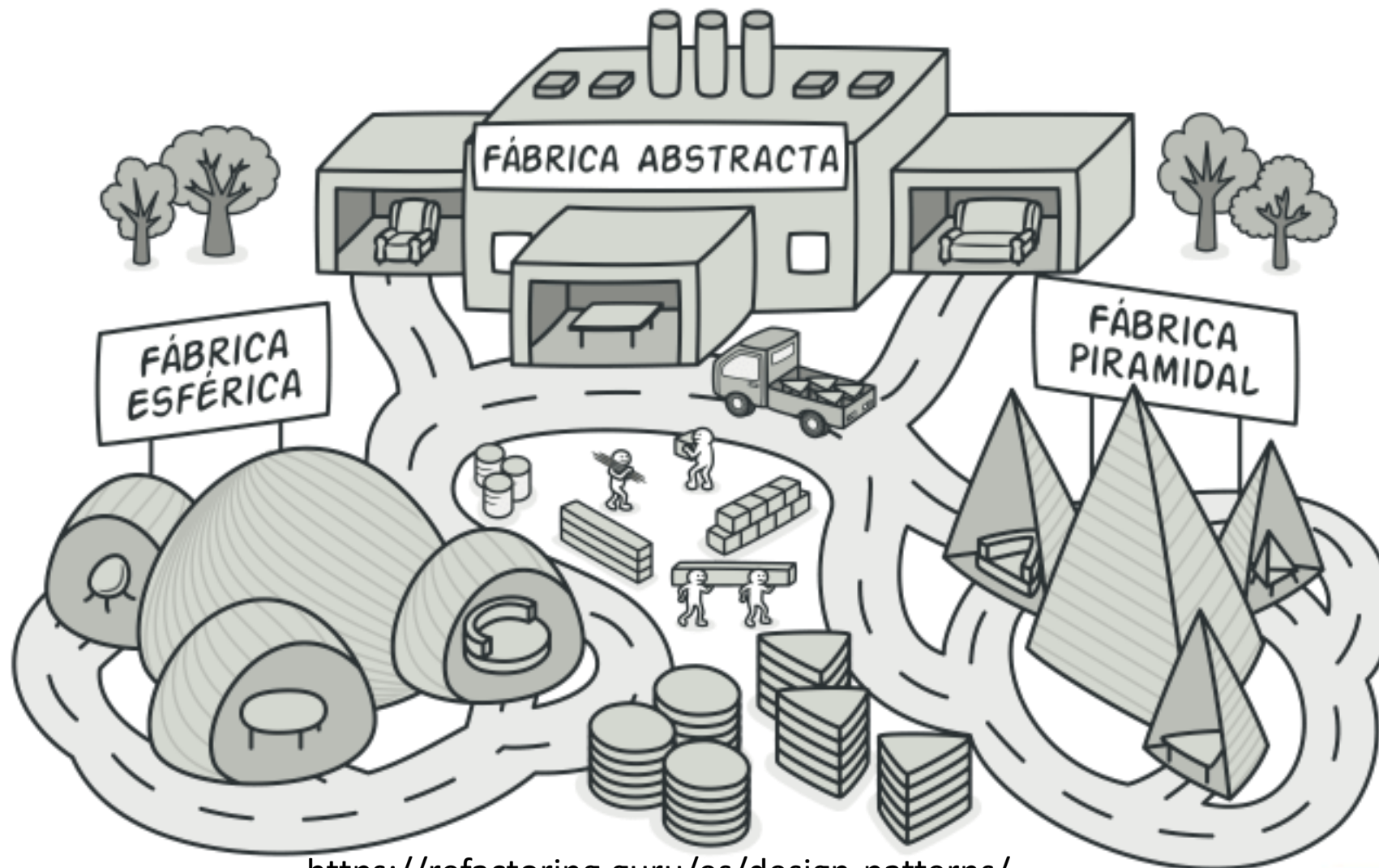
```
public class FabricaFigura {  
  
    public IFigura getFigura(String tipo) {  
        if (tipo.toUpperCase().equals("TRIANGULO")) {  
            return new Triangulo();  
        } else if (tipo.toUpperCase().equals("RECTANGULO")) {  
            return new Rectangulo();  
        }  
        return new OtraFigura();  
    }  
}
```

Factory Method

El Código

```
public class Main {  
  
    public static void main(String[] args) {  
        FabricaFigura fabrica = new FabricaFigura();  
        IFigura figura = fabrica.getFigura("triangulo");  
        System.out.println("El área es: "+figura.calcularArea(12, 10));  
    }  
}
```


Abstract Factory




<https://refactoring.guru/es/design-patterns/>



Abstract Factory

Este patrón “provee una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta” (Gamma, Helm, Johnson, & Vlissides, 1994).



Abstract Factory

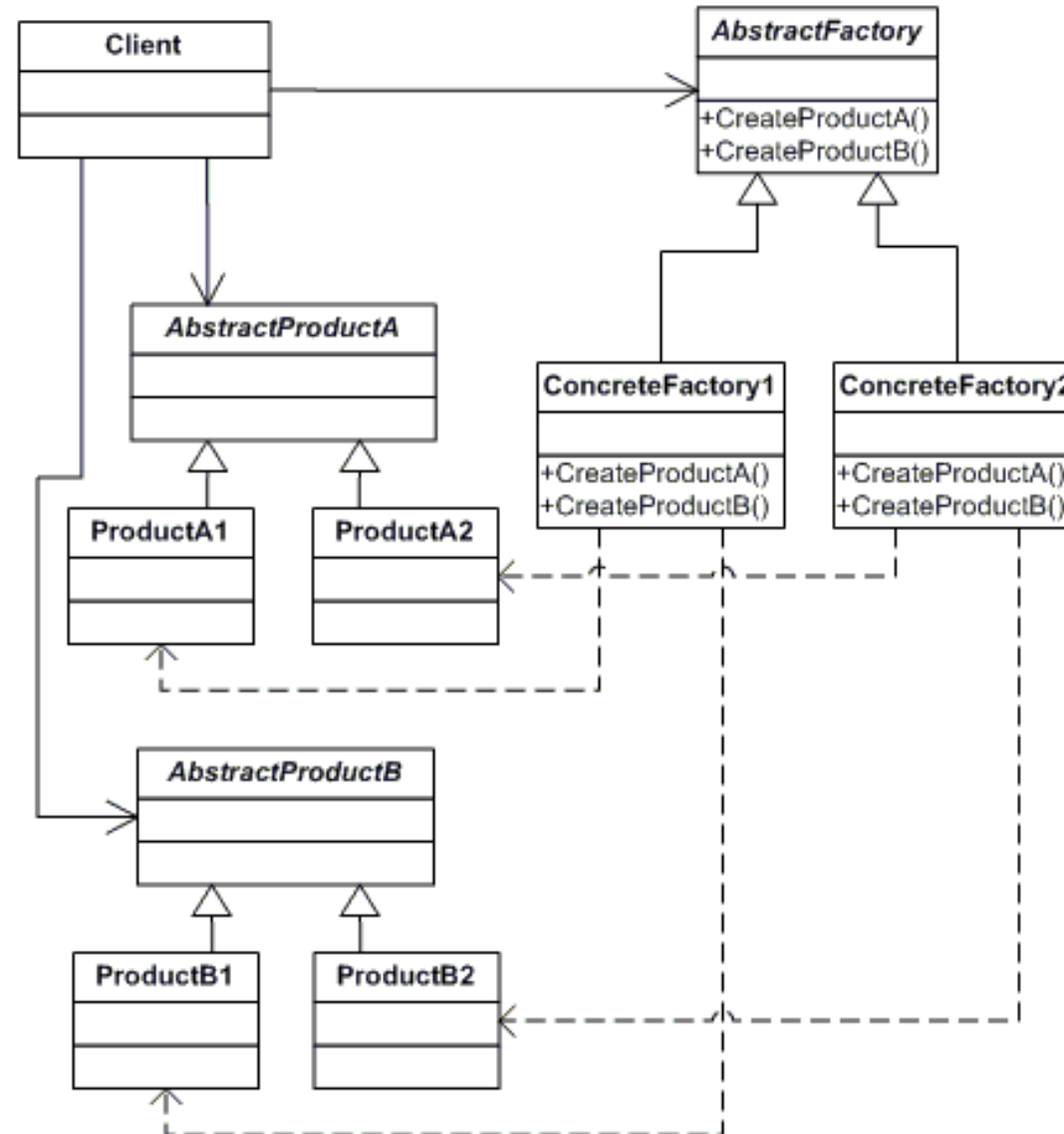


Abstract Factory



Abstract Factory

EI UML



Abstract Factory

El polimorfismo

```
public interface Boton {  
    public String getNombre();  
}  
  
public class BotonEscritorio implements Boton{  
    @Override  
    public String getNombre() {  
        return "Un Botón de Escritorio";  
    }  
}  
  
public class BotonWeb implements Boton{  
    @Override  
    public String getNombre() {  
        return "Un Botón de Web";  
    }  
}
```


Abstract Factory

Otro polimorfismo

```
public interface CajaTexto {  
    public String getNombre();  
}  
  
public class CajaEscritorio implements CajaTexto{  
    @Override  
    public String getNombre() {  
        return "Una Caja de Texto de Escritorio";  
    }  
}  
  
public class CajaWeb implements CajaTexto{  
    @Override  
    public String getNombre() {  
        return "Una Caja de Texto de Web";  
    }  
}
```

Abstract Factory

La Fabbrica Abstracta

```
public interface FabricaAbstracta {  
    Boton getBoton();  
    CajaTexto getCaja();  
}
```

Abstract Factory

```
public class FabricaWeb implements FabricaAbstracta{  
  
    @Override  
    public Boton getBoton() {  
        return new BotonWeb();  
    }  
  
    @Override  
    public CajaTexto getCaja() {  
        return new CajaWeb();  
    }  
  
}
```

Fabrica Concreta

Abstract Factory

```
public class FabricaEscritorio implements FabricaAbstracta{
```

```
    @Override
```

```
    public Boton getBoton() {  
        return new BotonEscritorio();  
    }
```

```
    @Override
```

```
    public CajaTexto getCaja() {  
        return new CajaEscritorio();  
    }
```

```
}
```

Otra Fabrica Concreta


Abstract Factory

```
public static void main(String[] args) {  
    probarAbstractFactory();  
}  
private static void probarAbstractFactory() {  
    FabricaAbstracta fe = new FabricaEscritorio();  
    FabricaAbstracta fw = new FabricaWeb();  
    CajaTexto cajaTextoE = fe.getCaja();  
    CajaTexto cajaTextoW = fw.getCaja();  
    Boton botonE = fe.getBoton();  
    Boton botonW = fw.getBoton();  
    Boton boton1W = new FabricaWeb().getBoton();  
    CajaTexto cajaTexto1W = new FabricaWeb().getCaja();  
    Boton boton1E = new FabricaEscritorio().getBoton();  
    CajaTexto cajaTexto1E = new FabricaEscritorio().getCaja();  
    System.out.println(cajaTextoE.getNombre());  
    System.out.println(cajaTextoW.getNombre());  
    System.out.println(cajaTexto1E.getNombre());  
    System.out.println(cajaTexto1W.getNombre());  
    System.out.println(botonE.getNombre());  
    System.out.println(botonW.getNombre());  
    System.out.println(boton1E.getNombre());  
    System.out.println(boton1W.getNombre());  
}
```

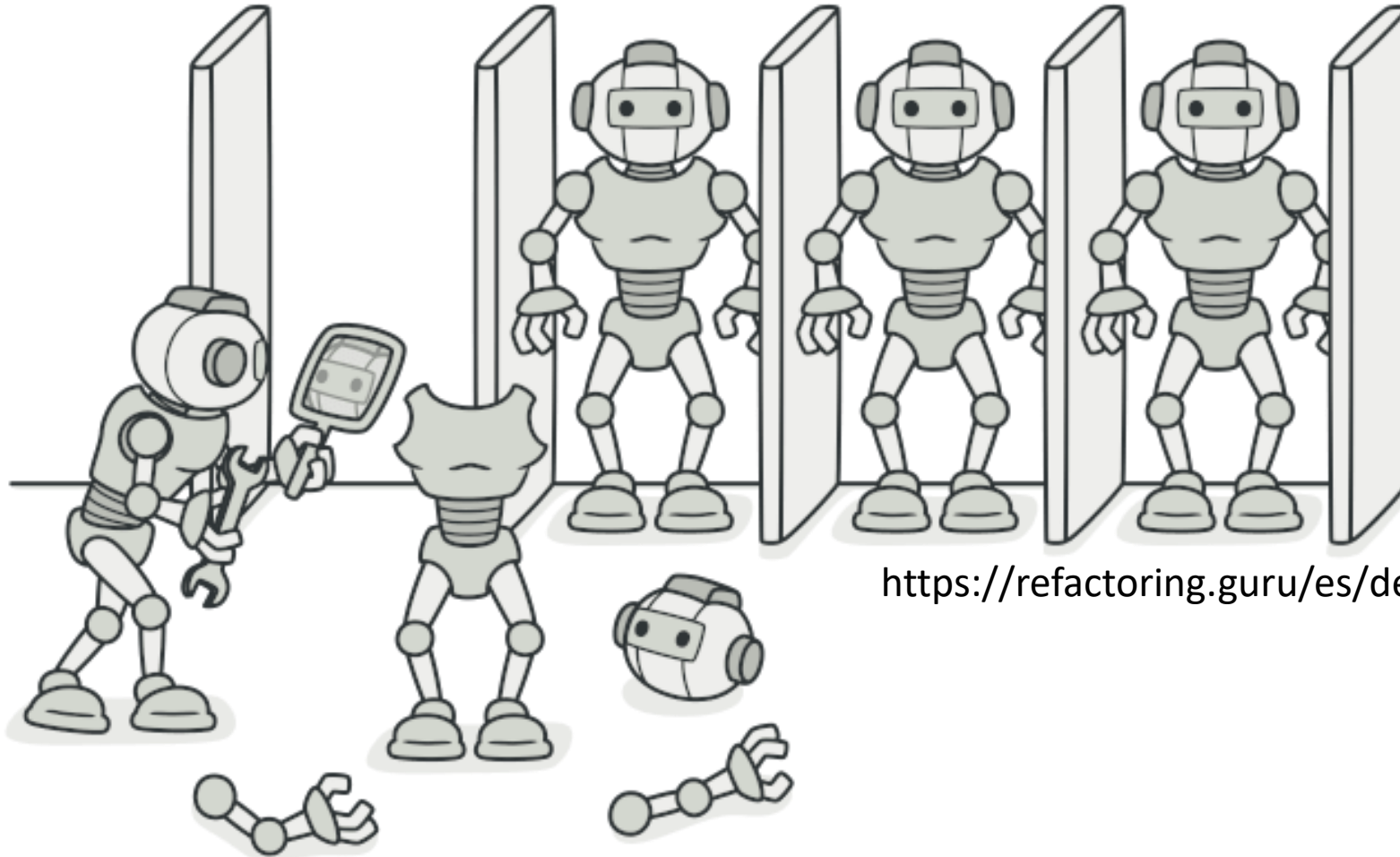
La prueba del código

La salida del programa

```
Una Caja de Texto de Escritorio  
Una Caja de Texto de Web  
Una Caja de Texto de Escritorio  
Una Caja de Texto de Web  
Un Botón de Escritorio  
Un Botón de Web  
Un Botón de Escritorio  
Un Botón de Web
```



Prototype



<https://refactoring.guru/es/design-patterns/>

Prototype



<https://refactoring.guru/es/design-patterns/>


Prototype





Prototype


Este patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz permite clonar un objeto sin acoplar el código a la clase de ese objeto. Generalmente, dicha interfaz contiene un único método clonar.





Prototype

El método clonar crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo.



```
public class Persona{  
    private String nombre, telefono, direccion;  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    public String getTelefono() {  
        return telefono;  
    }  
    public void setTelefono(String telefono) {  
        this.telefono = telefono;  
    }  
    public String getDireccion() {  
        return direccion;  
    }  
    public void setDireccion(String direccion) {  
        this.direccion = direccion;  
    }  
}
```

La Clase para heredar

Prototype

La Interface propia para que nuestros objetos se puedan clonar.

```
public interface Clonable extends Cloneable{  
    public Clonable clonar();  
}
```


Prototype

```
public class Empleado extends Persona implements Clonable{  
    private double salario;  
    public double getSalario() {  
        return salario;  
    }  
    public void setSalario(double salario) {  
        this.salario = salario;  
    }  
    @Override  
    public Clonable clonar() {  
        try{  
            return (Clonable) clone();  
        } catch (CloneNotSupportedException e) {  
            return null;  
        }  
    }  
}
```

La Clase que hereda e
implementa interface

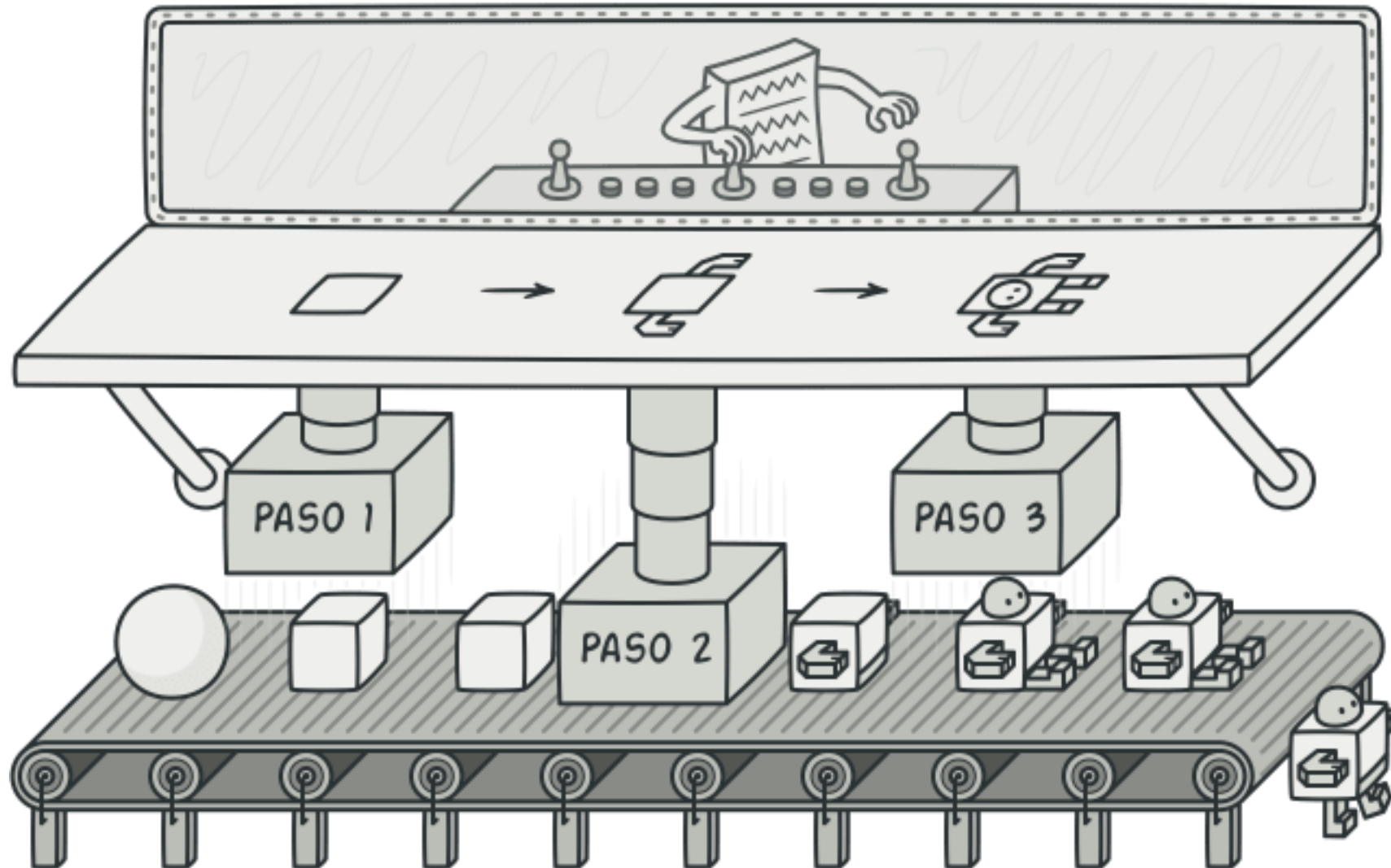
Prototype

```
public static void main(String[] args) {  
    probar();  
}
```

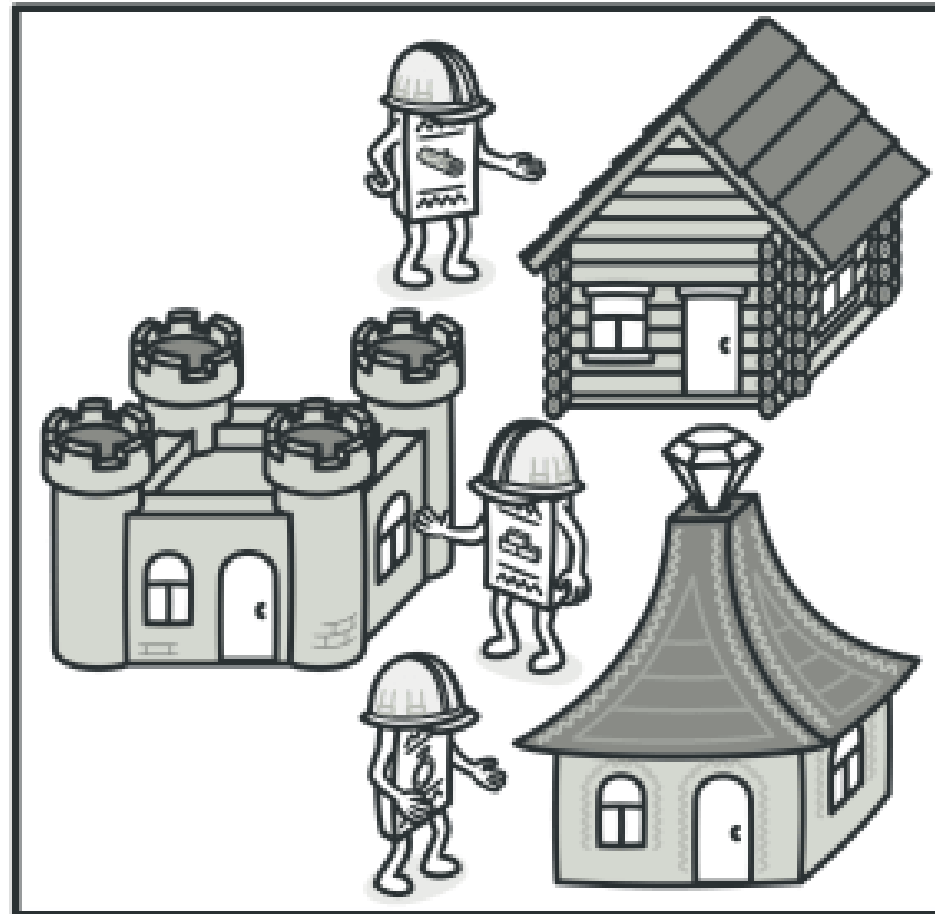
La Clase para probar

```
private static void probar() {  
    Empleado empleado = new Empleado();  
    empleado.setDireccion("Calle 15");  
    empleado.setNombre("Braulio");  
    empleado.setTelefono("3214567890");  
    empleado.setSalario(3100000);  
    Empleado empleado1 = (Empleado)empleado.clonar();  
    empleado1.setSalario(200);  
    empleado1.setNombre("Patiño");  
    System.out.println(empleado.getNombre());  
    System.out.println(empleado.getSalario());  
}
```

Builder



Builder




Builder



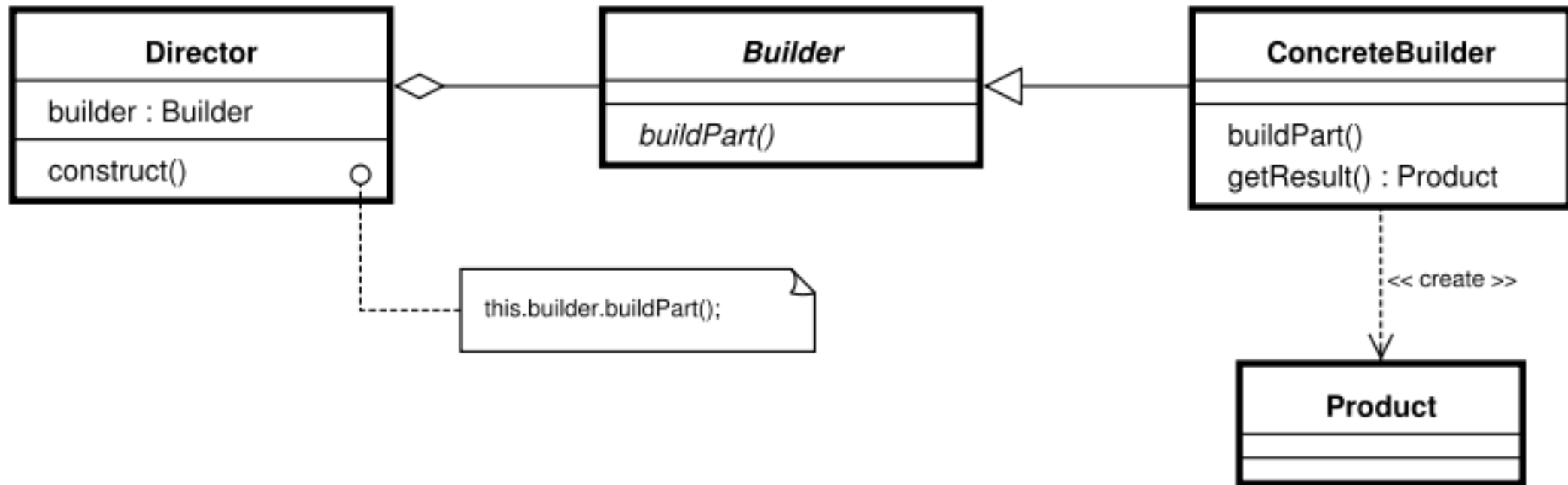


Builder

Este patrón nos permite tener especie de una política o directriz general para la creación de objetos y esto lo hace centralizando en una clase constructora que se encarga de proporcionar un punto centralizado.



Builder



Builder

```
import java.util.ArrayList;
public class Usuario {
    private String login, nombre, identidad, password, telefono;
    private String direccion, ultimoCambio;
    private ArrayList<String> roles;
    public ArrayList<String> getRoles() {
        return roles;
    }
    public void setRoles(ArrayList<String> roles) {
        this.roles = roles;
    }
    public String getLogin() {
        return login;
    }
}
```

La Clase original

Builder

```
import java.util.ArrayList;
public class UsuarioBuilder implements IConstructor{
    private String login, nombre, identidad, password, telefono;
    private String direccion, ultimoCambio;
    private ArrayList<String> roles;

    public UsuarioBuilder(String login){
        this.login = login;
    }
    public UsuarioBuilder nombre(String nombre){
        this.nombre = nombre;
        return this;
    }
    public UsuarioBuilder identidad(String identidad){
        this.identidad = identidad;
        return this;
    }
}
```

La Clase constructora

Builder

```
public UsuarioBuilder roles(String rol){  
    if(this.roles==null){  
        this.roles = new ArrayList<>();  
    }else{}  
    this.roles.add(rol);  
    return this;  
}
```

@Override

```
public Usuario build(){  
    Usuario usuario = new Usuario();  
    usuario.setDireccion(direccion);  
    usuario.setIdentidad(identidad);  
    usuario.setNombre(nombre);  
    usuario.setPassword(password);  
    usuario.setTelefono(telefono);  
    usuario.setUltimoCambio(ultimoCambio);  
    usuario.setRoles(roles);  
    return usuario;  
}
```

El método que
construye

La interface que establece el contrato

```
public interface IConstructor {  
    public Usuario build();  
}
```

Builder

La Clase para probar

```
public class Main {  
    public static void main(String[] args) {  
        UsuarioBuilder builder = new UsuarioBuilder("deivis");  
        Usuario usuario = builder.nombre("Deivis Martinez Acosta")  
            .roles("administrativo")  
            .roles("docente")  
            .roles("estudiante").build();  
  
        System.out.println(usuario.getNombre());  
        usuario.getRoles().forEach((rol) -> {  
            System.out.println(rol);  
        });  
    }  
}
```

Bibliografía

Design Patterns: Elements of Reusable Object-Oriented Software (GoF)

Código Limpio, Robert C. Martin

ingeniería de software ian sommerville 9 edición

Head First Design Patterns (A Brain Friendly Guide) (Inglés)

Tapa blanda – Ilustrado, 4 noviembre 2004

Design Patterns: Elements of Reusable Object-Oriented Software (Addison Wesley professional computing series) (Inglés) Tapa dura – 31 octubre 1994



UNIVERSIDAD
Popular del Cesar

Vamos a la práctica