PREGRADO

UNIDAD 1: OVERVIEW

GOF CREATIONAL DESIGN PATTERNS

SI720 | Diseño y Patrones de Software

Al finalizar la unidad, el estudiante elabora y
comunica artefactos de diseño de software aplicando
principios básicos y
patrones de diseño para un dominio y contexto
determinados

AGENDA

INTRO
BUILDER
FACTORY METHOD
SINGLETON

GoF design patterns

Creational

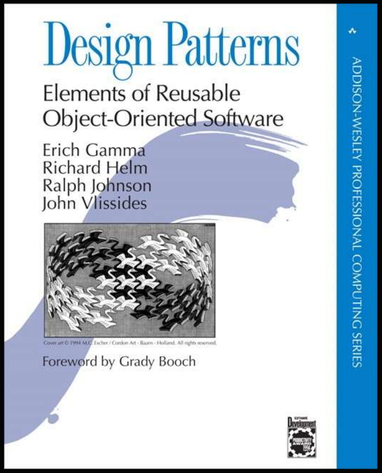
Builder
Factory Method
Prototype
Singleton

Structural

Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

Behavioral

Chain of
responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
States
Strategy
Template Method
Visitor


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Creational Design Patterns

- Enfrentan una de las tareas más comúnmente realizadas en una aplicación OO, la creación de objetos.
- Admiten un mecanismo uniforme, simple y controlado para crear objetos.
- Permiten la encapsulación de los detalles sobre qué clases se instancian y cómo se crean estas instancias.
- Fomentan el uso de interfaces, lo que reduce el acoplamiento.

Creational Design Patterns

Pattern	Description
Abstract Factory	Allows the creation of an instance of a class from a suite of related classes without having a client object to specify the actual concrete class to be instantiated.
Builder	Allows creation of a complex object by providing the information on only its type and content , keeping the details of the object creation transparent to the client. This allows the same construction process to produce different representations of the object.
Factory Method	Subclass decides which concrete class to instantiate. When a client object does not know which class to instantiate , it can make use of the factory method to create an instance of an appropriate class from a class hierarchy or a family of related classes.
Prototype	Provides a simpler way of creating an object by cloning it from an existing (prototype) object.
Singleton	Provides a controlled object creation mechanism to ensure that only one instance of a given class exists.

AGENDA

INTRO
BUILDER
FACTORY METHOD
SINGLETON

Builder

Intent

“Separate construction of complex objects from their representation.”

Motivation

When creation logic for a complex object is independent of assembling its parts; different representations are possible.

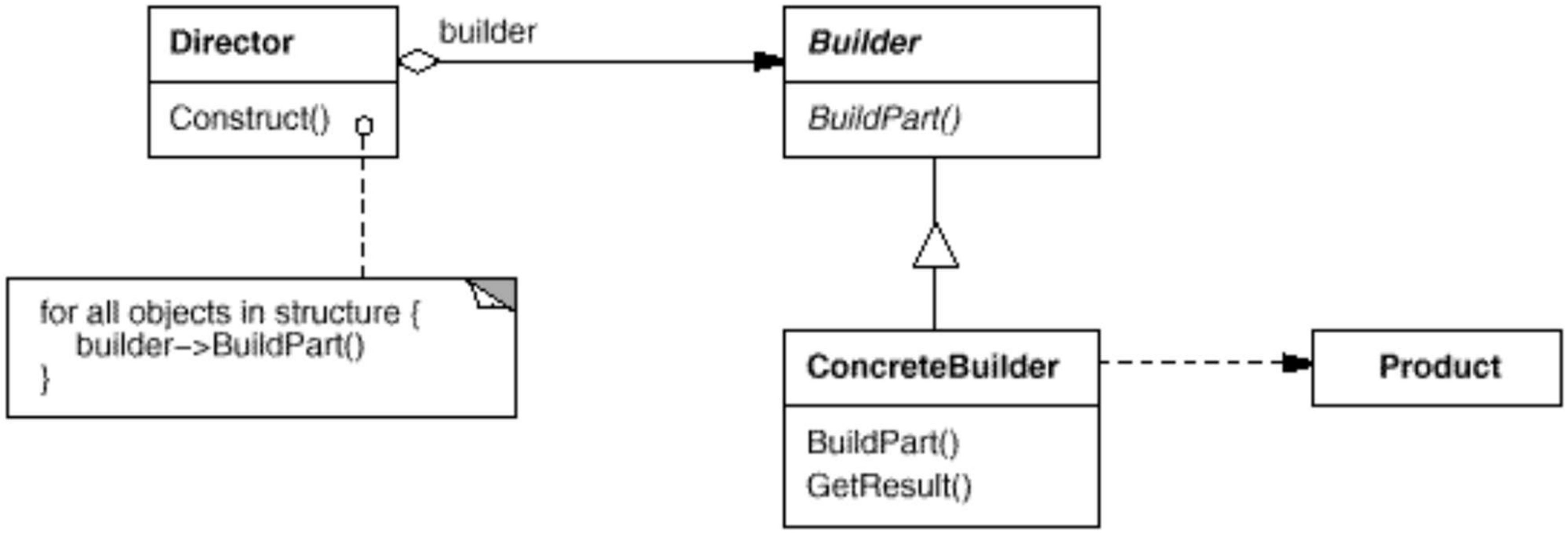
Participants

Builder, ConcreteBuilder, Director, Product

Applicability

Algoritmos de creación de un objeto complejo pueden ser independientes de las partes que forman el objeto y cómo se ensamblan.

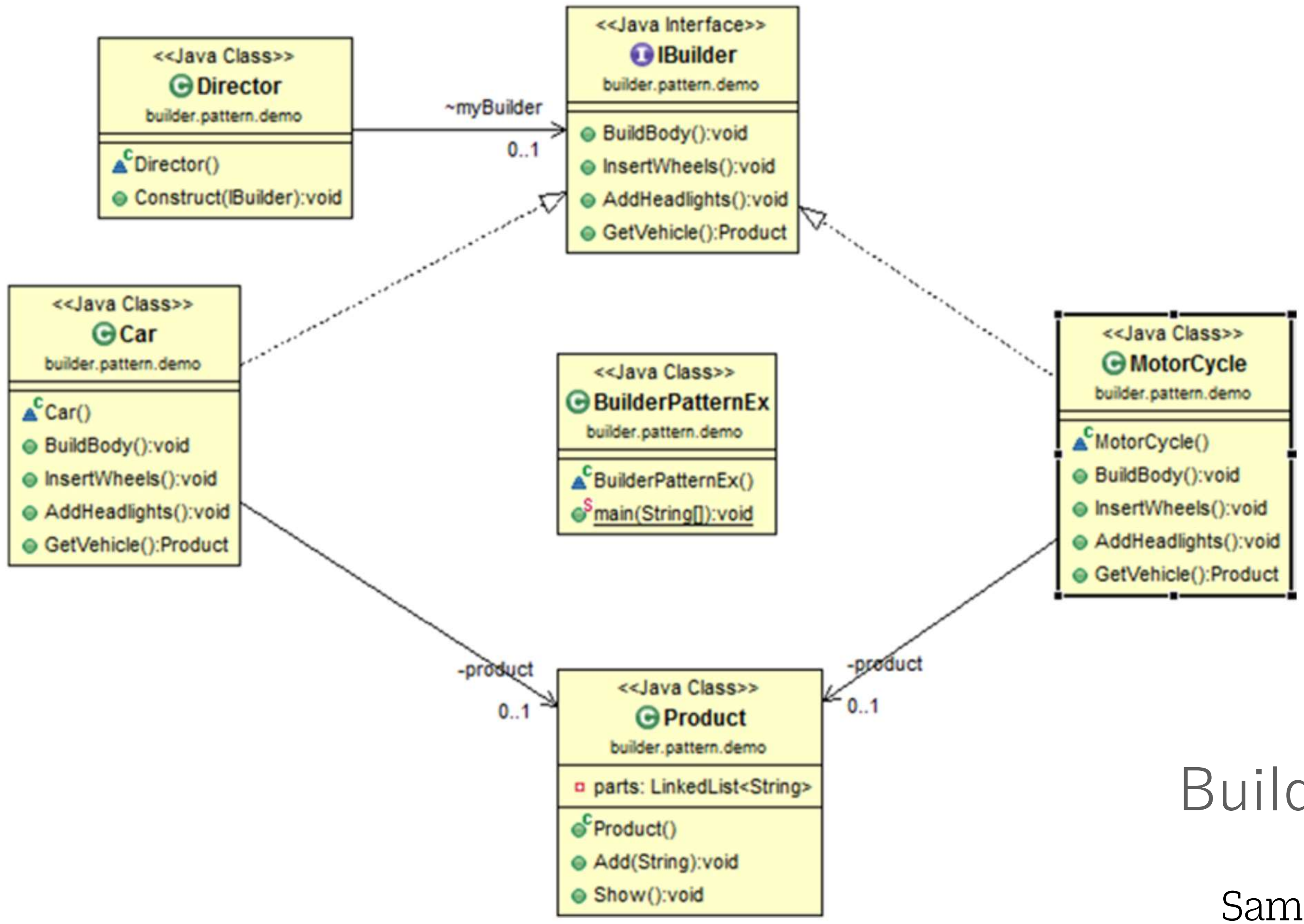
El proceso permite diferentes representaciones para el objeto construido.



```
for all objects in structure {
    builder->BuildPart()
}
```

Builder

Structure



Builder

Sample

```
package builder.pattern.demo;

import java.util.LinkedList;

// Builders common interface
interface IBuilder {
    void buildBody();
    void insertWheels();
    void addHeadlights();
    Product getVehicle();
}
```

Builder

Sample

```
// Car is ConcreteBuilder
class Car implements IBuilder {
	private Product product = new Product();
	@Override
	public void buildBody() {
		product.add("This is a body of a Car");
	}
	@Override
	public void insertWheels() {
		product.add("4 wheels are added");
	}
	@Override
	public void addHeadlights() {
		product.add("2 Headlights are added");
	}
	@Override
	public Product getVehicle() {
		return product;
	}
}
```

Builder

Sample

```
// Motorcycle is a ConcreteBuilder
class MotorCycle implements IBuilder {
	private Product product = new Product();
	@Override
	public void BuildBody() {
		product.add("This is a body of a Motorcycle");
	}
	@Override
	public void insertWheels() {
		product.add("2 wheels are added");
	}
	@Override
	public void addHeadlights() {
		product.add("1 Headlights are added");
	}
	@Override
	public Product getVehicle() {
		return product;
	}
}
```

Builder

Sample

```
// "Product"
class Product {
	// We can use any data structure that you prefer.
	// We have used LinkedList here.
	private LinkedList<String> parts;
	public Product() {
		parts = new LinkedList<String>();
	}

	public void add(String part) {
		//Adding parts
		parts.addLast(part);
	}
	public void show() {
		System.out.println("\n Product completed as below :");
		for(int i=0;i<parts.size();i++) {
			System.out.println(parts.get(i));
		}
	}
}
```

Builder

Sample

```
// "Director"
class Director {
        IBuilder myBuilder;
        // A series of steps—for the production
        public void construct(IBuilder builder) {
                myBuilder = builder;
                myBuilder.buildBody();
                myBuilder.insertWheels();
                myBuilder.addHeadlights();
        }
}
```

Builder

Sample

```
class BuilderPatternEx {
	public static void main(String[] args) {
		System.out.println("***Builder Pattern Demo***\n");
		Director director = new Director();
		IBuilder carBuilder = new Car();
		IBuilder motorBuilder = new MotorCycle();
		// Making Car
		director.construct(carBuilder);
		Product p1 = carBuilder.getVehicle();
		p1.show();
		//Making MotorCycle
		director.construct(motorBuilder);
		Product p2 = motorBuilder.getVehicle();
		p2.Show();
	}
}
```

Builder

Sample

Builder

Consequences

Permite variar la representación interna de un producto.

Aisla el código para construcción y para representación.

Permite un control más fino sobre el proceso de construcción.

AGENDA

INTRO
BUILDER
FACTORY METHOD
SINGLETON

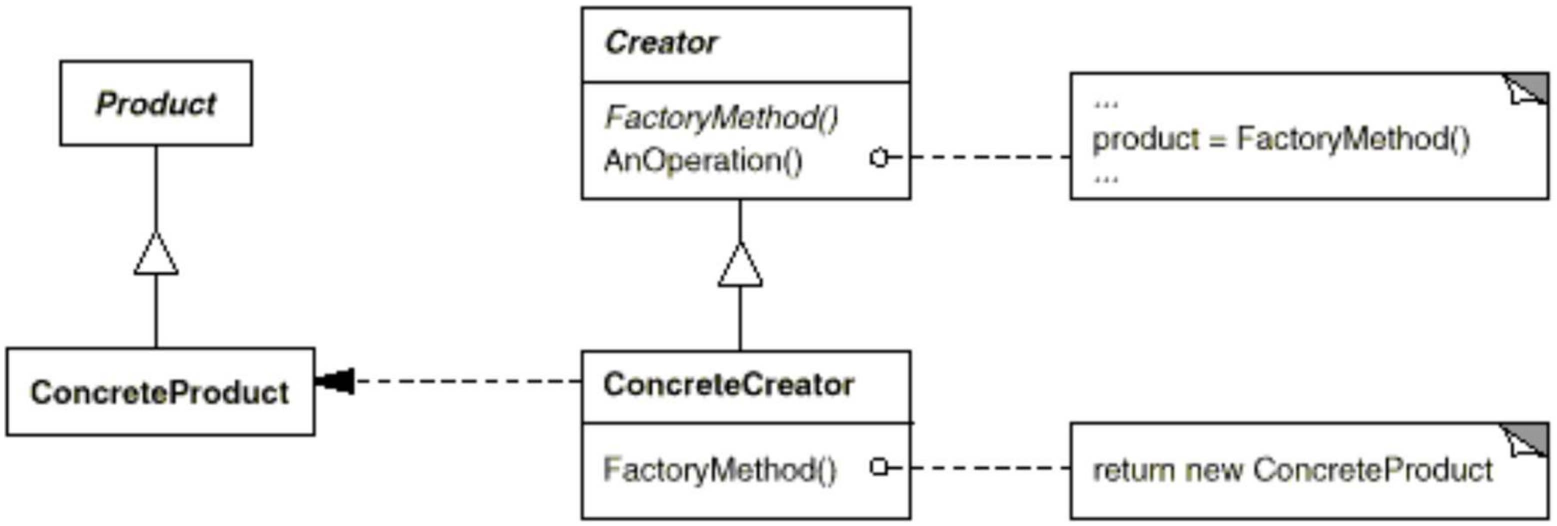
Factory Method

Intent
“Define interface for creating object, let subclasses decide which class to instantiate.”

Motivation
Escenarios donde se requiere encapsular el conocimiento de qué clase debe ser un objeto, delegando ello a subclases.

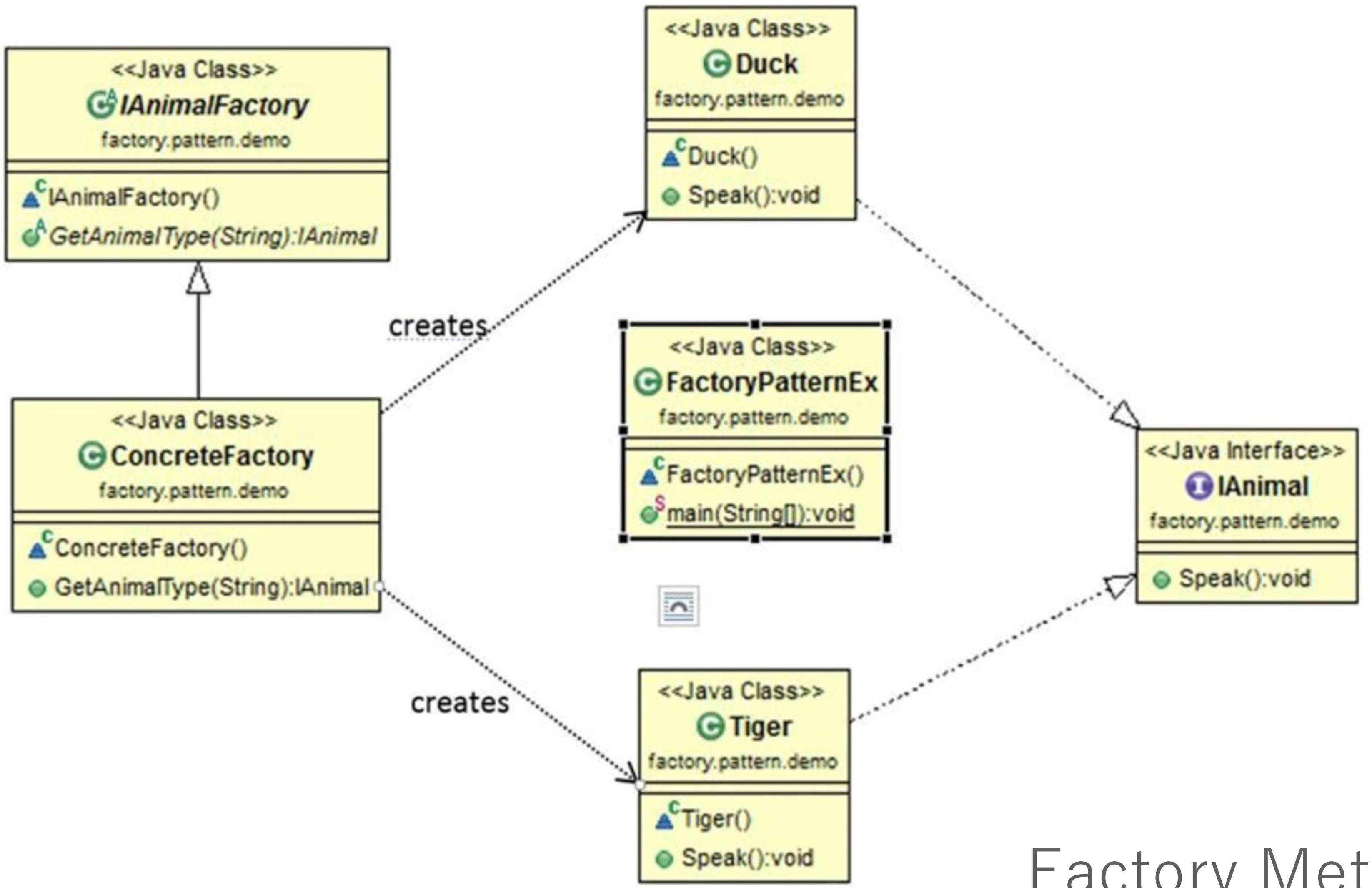
Participants
Product, ConceteProduct, Creator, ConcreteCreator

Applicability
Caching de data con alto costo de carga.
Compartir data para acceso global.
Propósitos de Single-point-of-contact (por ejemplo Logging).



Factory Method

Structure



Factory Method

Sample

```java
package refactoring_guru.factory_method.example.buttons;

public interface Button {
    void render();
    void onClick();
}

/**
 * HTML button implementation.
 */
public class HtmlButton implements Button {

    public void render() {
        System.out.println("<button>Test Button</button>");
        onClick();
    }

    public void onClick() {
        System.out.println("Click! Button says - 'Hello World!'");
    }
}
```

Factory Method

Sample

```
 * Windows button implementation.
 */
public class WindowsButton implements Button {
   JPanel panel = new JPanel();
   JFrame frame = new JFrame();
   JButton button;


   public void render() {
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JLabel label = new JLabel("Hello World!");
      label.setOpaque(true);
      label.setBackground(new Color(235, 233, 126));
      label.setFont(new Font("Dialog", Font.BOLD, 44));
      label.setHorizontalAlignment(SwingConstants.CENTER);
      panel.setLayout(new FlowLayout(FlowLayout.CENTER));
      frame.getContentPane().add(panel);
      panel.add(label);
      onClick();
      panel.add(button);


      frame.setSize(320, 200);
      frame.setVisible(true);
      onClick();
   }

…
```

Factory Method

Sample

...

```
public void onClick() {
    button = new JButton("Exit");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            frame.setVisible(false);
            System.exit(0);
        }
    });
```

Factory Method

Sample

```
package refactoring_guru.factory_method.example.factory;
import refactoring_guru.factory_method.example.buttons.Button;
/**
* Base factory class. Note that "factory" is merely a role for the class. It
* should have some core business logic which
needs different products to be
•   created.
•   */
public abstract class Dialog {
    public void renderWindow() {
    // ... other code ...
    Button okButton = createButton();
    okButton.render();
    }
    /** * Subclasses will override this method in order
     to create specific button * objects. */
    public abstract Button createButton();
}
```

Factory Method

Sample

```java
package refactoring_guru.factory_method.example.factory;

import refactoring_guru.factory_method.example.buttons.Button;
import refactoring_guru.factory_method.example.buttons.HtmlButton;

/**
 * HTML Dialog will produce HTML buttons.
 */
public class HtmlDialog extends Dialog {

    @Override
    public Button createButton() {
        return new HtmlButton();
    }
}
```

Factory Method

Sample

```java
package refactoring_guru.factory_method.example.factory;

import refactoring_guru.factory_method.example.buttons.Button;
import refactoring_guru.factory_method.example.buttons.WindowsButton;

/**
 * Windows Dialog will produce Windows buttons.
 */
public class WindowsDialog extends Dialog {

    @Override
    public Button createButton() {
        return new WindowsButton();
    }
}
```

Factory Method

Sample

```java
package refactoring_guru.factory_method.example;


import refactoring_guru.factory_method.example.factory.Dialog;
import refactoring_guru.factory_method.example.factory.HtmlDialog;
import refactoring_guru.factory_method.example.factory.WindowsDialog;


/**
 * Demo class. Everything comes together here.
 */
public class Demo {
   private static Dialog dialog;


   public static void main(String[] args) {
      configure();
      runBusinessLogic();
   }


…
```

Factory Method

Sample

```java
...
    /**
     * The concrete factory is usually chosen depending on configuration or
     * environment options.
     */
    static void configure() {
        if (System.getProperty("os.name").equals("Windows 10")) {
            dialog = new WindowsDialog();
        } else {
            dialog = new HtmlDialog();
        }
    }


    /**
     * All of the client code should work with factories and products through
     * abstract interfaces. This way it does not care which factory it works
     * with and what kind of product it returns.
     */
    static void runBusinessLogic() {
        dialog.renderWindow();
    }
}
```

Factory Method

Sample

Factory Method

Consequences

Elimina la necesidad de vincular clases específicas de la aplicación en el código.
El Código solo trata con la interfaz del Producto, pudiendo trabajar con cualquier clase Concrete Product definida por el usuario.

AGENDA

INTRO

BUILDER

FACTORY METHOD

SINGLETON

Singleton

Intent

“Ensure a class only has one instance, and provide a global point of access to it.”

Motivation

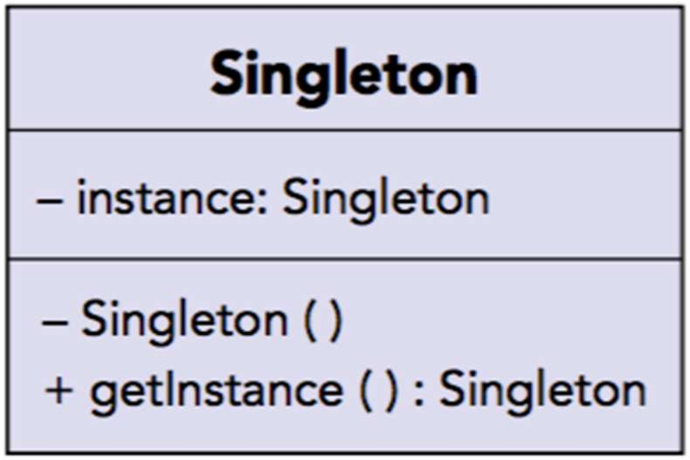
Escenarios donde es importante que una clase tenga una sola instancia.

Participants

Singleton.

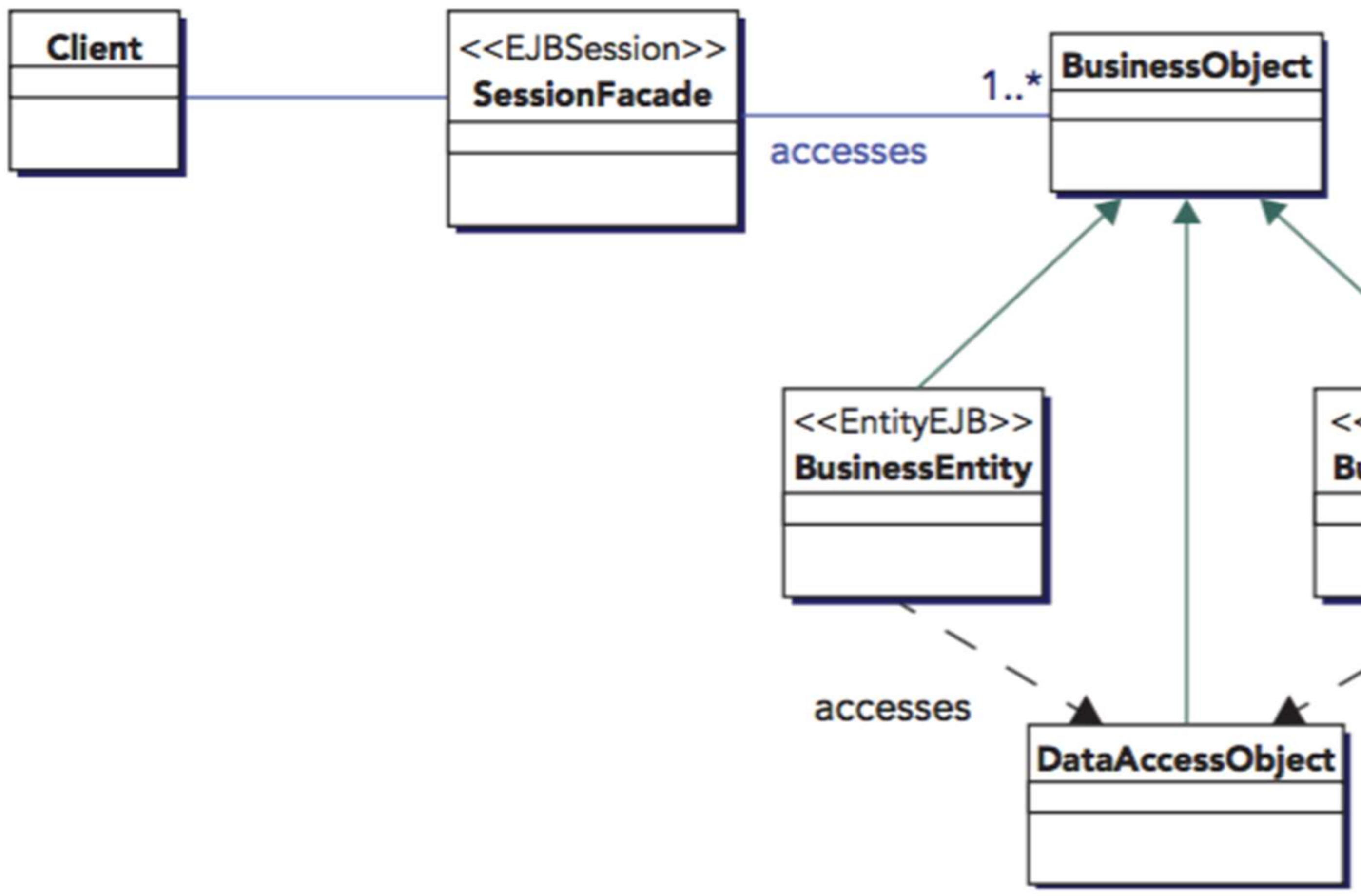
Applicability

Caching de data con alto costo de carga.
Compartir data para acceso global.
Propósitos de Single-point-of-contact (por ejemplo Logging).



Singleton

Structure



Singleton

Sample

```java
// approach with static method
package com.designpatterns.singleton;

public class MySingleton {
  private static MySingleton instance;
  private MySingleton() {}

  public static MySingleton getInstance() {
    if (instance==null){
    // 1
    instance=new MySingleton();
    }
    return instance;
  }
}

// approach with static and synchronized method
package com.designpatterns.singleton;

public class MySingleton {
  private static MySingleton instance;
  private MySingleton() {}

  public static synchronized MySingleton getInstance() {
    if (instance==null){
    // 1
    instance=new MySingleton();
    }
    return instance;
  }
}
```

Singleton

Sample

```
// approach with creation when class loading
package com.designpatterns.singleton;

public class MySingleton {
  private final static MySingleton instance = new MySingleton();
  private MySingleton() {}
  public static MySingleton getInstance() {
    return instance;
  }
}


// approach with static block
package com.designpatterns.singleton;

public class MySingleton {
  private static MySingleton instance=null;
  static {
    instance=new MySingleton();
  }
  private MySingleton() {}
  public static MySingleton getInstance() {
    return instance;
  }
}
```

Singleton

Sample

```
// approach with creation double check
package com.designpatterns.singleton;

public class MySingleton {
  private volatile MySingleton instance;
  private MySingleton() {}
    public MySingleton getInstance() {
      if (instance == null) {
        // 1
        synchronized (MySingleton.class) {
          if (instance == null) {
            // 2
            instance = new MySingleton();
          }
        }
      }
      return instance;
    }
  }
```

Singleton

Sample

```java
// approach with creation when class loading
package com.designpatterns.singleton;

public enum MySingletonEnum { INSTANCE;
  public void doSomethingInteresting(){
  }
}

…
// obtaining reference to instance
MySingletonEnum mse = MySingletonEnum.INSTANCE;
```

Singleton

Sample

```java
package com.designpatterns.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import java.util.logging.Logger;

@Singleton
public class CacheSingletonBean {
  private Map<Integer, String> myCache;

  @PostConstruct
  public void start(){
    Logger.getLogger("MyGlobalLogger").info("Started!");
    myCache = new HashMap<Integer, String>();
  }

  public void addUser(Integer id, String name){
    myCache.put(id, name);
  }

  public String getName(Integer id){
    return myCache.get(id);
  }
}
```

Singleton

Sample

```java
package com.designpatterns.singleton;

@Startup
@DependsOn("MyLoggingBean")
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class CacheSingletonBean {
  // Implementation code here.
}
```

Singleton

Sample

```
package com.designpatterns.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.DependsOn;
import javax.ejb.EJB;
import javax.ejb.Lock;
import javax.ejb.LockType;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Startup
@DependsOn("MyLoggingBean")
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
public class CacheSingletonBean {
  private Map<Integer, String> myCache;
  @EJB
MyLoggingBean loggingBean;
  @PostConstruct
  public void start(){
      loggingBean.logInfo("Started!");
      myCache = new HashMap<Integer, String>();
  }
  @Lock(LockType.WRITE)
public void addUser(Integer id, String name){
    myCache.put(id, name);
  }
  @Lock(LockType.READ)
  public String getName(Integer id){ return myCache.get(id);
  }
}
```

Singleton

Sample

```java
package com.designpatterns.singleton;

import javax.annotation.PostConstruct;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import java.util.logging.Logger;

@Startup
@Singleton
public class MyLoggingBean {
  private Logger logger;
  @PostConstruct
  public void start(){
    logger = Logger.getLogger("MyGlobalLogger");
    logger.info("Well, I started first!!!”);
  }
  public void logInfo(String msg){
    logger.info(msg);
  }
}
```

Singleton

Sample

```
package com.designpatterns.singleton;

import java.util.HashMap;
import java.util.Map;
import javax.annotation.PostConstruct;
import javax.ejb.ConcurrencyManagement;
import javax.ejb.ConcurrencyManagementType;
import javax.ejb.DependsOn;
import javax.ejb.EJB;
import javax.ejb.Lock;
import javax.ejb.LockType;
import javax.ejb.Singleton;
import javax.ejb.Startup;

@Startup
@DependsOn("MyLoggingBean")
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
@Singleton
@AccessTimeout(value=120000) // default in milliseconds
public class CacheSingletonBean {
  private Map<Integer, String> myCache;
  @EJB
MyLoggingBean loggingBean;
  @PostConstruct public void start(){
      loggingBean.logInfo("Started!");
      myCache = new HashMap<Integer, String>();
  }
  @AccessTimeout(value=30, unit=TimeUnit.SECONDS)
  @Lock(LockType.WRITE)
  public void addUser(Integer id, String name){
    myCache.put(id, name);
  }
  @Lock(LockType.READ)
    public String getName(Integer id){
      return myCache.get(id);
  }
}
```

Singleton

Sample

```ruby
# Singleton standard implementation
class SimpleLogger
  @@instance = SimpleLogger.new

  def self.instance
     return @@instance
  end

  private_class_method :new
end
```

```ruby
# Using Singleton Module from Ruby Standard Library
require ‘singleton’

class SimpleLogger
  include Singleton
  # Class code here
end
```

Singleton

Sample

Singleton

Consequences

Acceso controlado a una sola instancia.

Evita que en el name space existan variables globales conteniendo una sola instancia.

Permite refinamiento de las operaciones y la representación.

Single-point-of-change en caso se desee permitir más de una instancia.

RESUMEN

Recordemos

- Los Creational Design Patterns se enfocan en la forma de crear objetos.
- Son útiles cuando la creación de instancias está sujeta a ciertas decisiones.
- Incluyen a

  Factory Method,

  Abstract Factory,

  Singleton,

  Prototype y

  Builder.

REFERENCIAS

Para profundizar

Design Patterns- Libro de Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm.

http://www.blackwasp.co.uk/gofpatterns.aspx

http://www.w3sdesign.com/

https://refactoring.guru/design-patterns/creational-patterns

https://www.javatpoint.com/creational-design-patterns

PREGRADO

Ingeniería de Software
Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova

LAUREATE
INTERNATIONAL
UNIVERSITIES