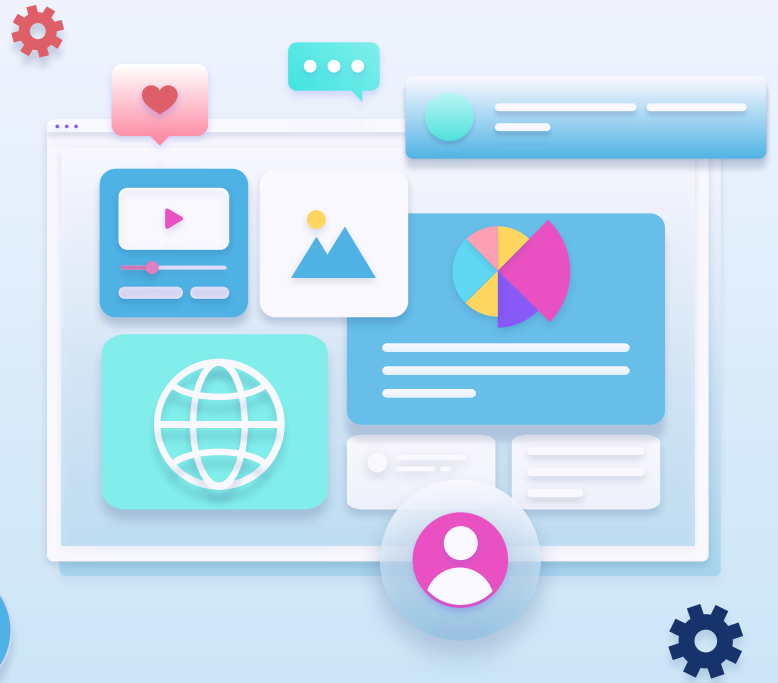
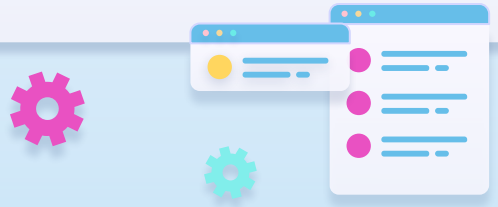
Patrón Builder



Ángel Chaves Chinchila C12113
Camilo Suarez Sandí C17811

Tabla de contenidos

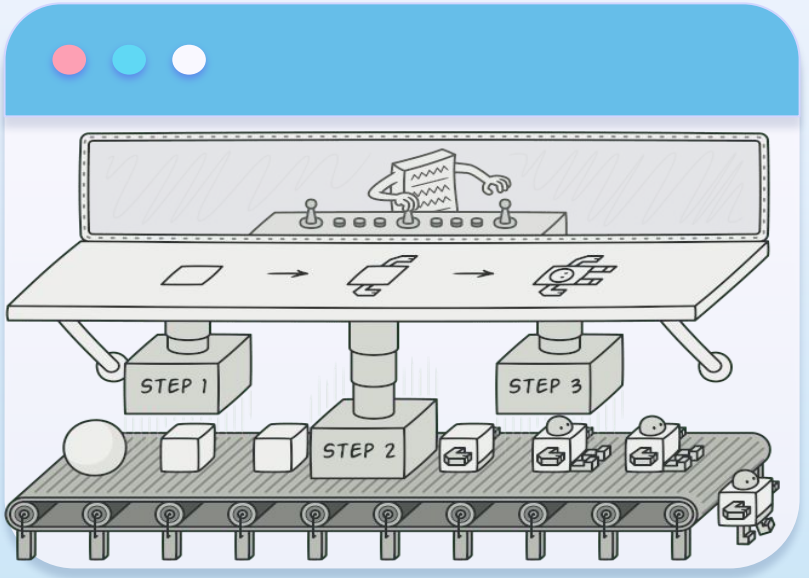


Descripción

- Patrón Creacional

- Separa la construcción de un objeto complejo de su uso

- Permite producir distintos tipos y representaciones de un objeto usando el mismo código de construcción



Problema

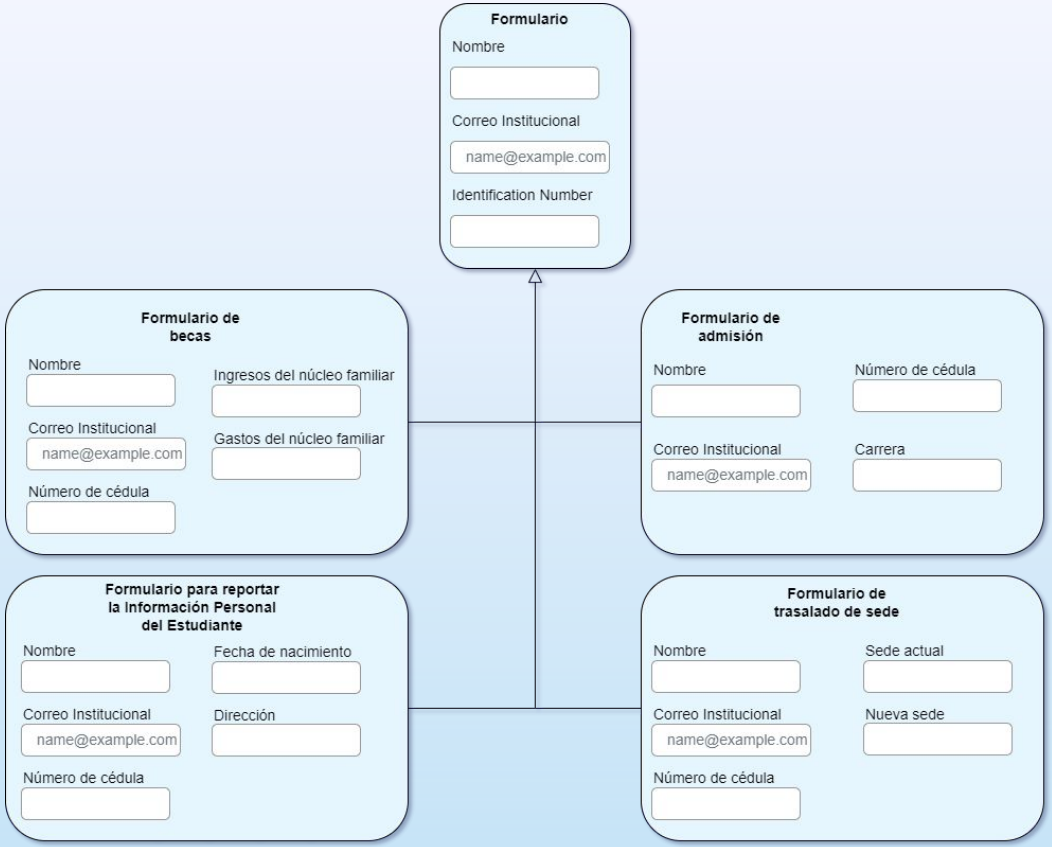
Flexibilidad en construcción

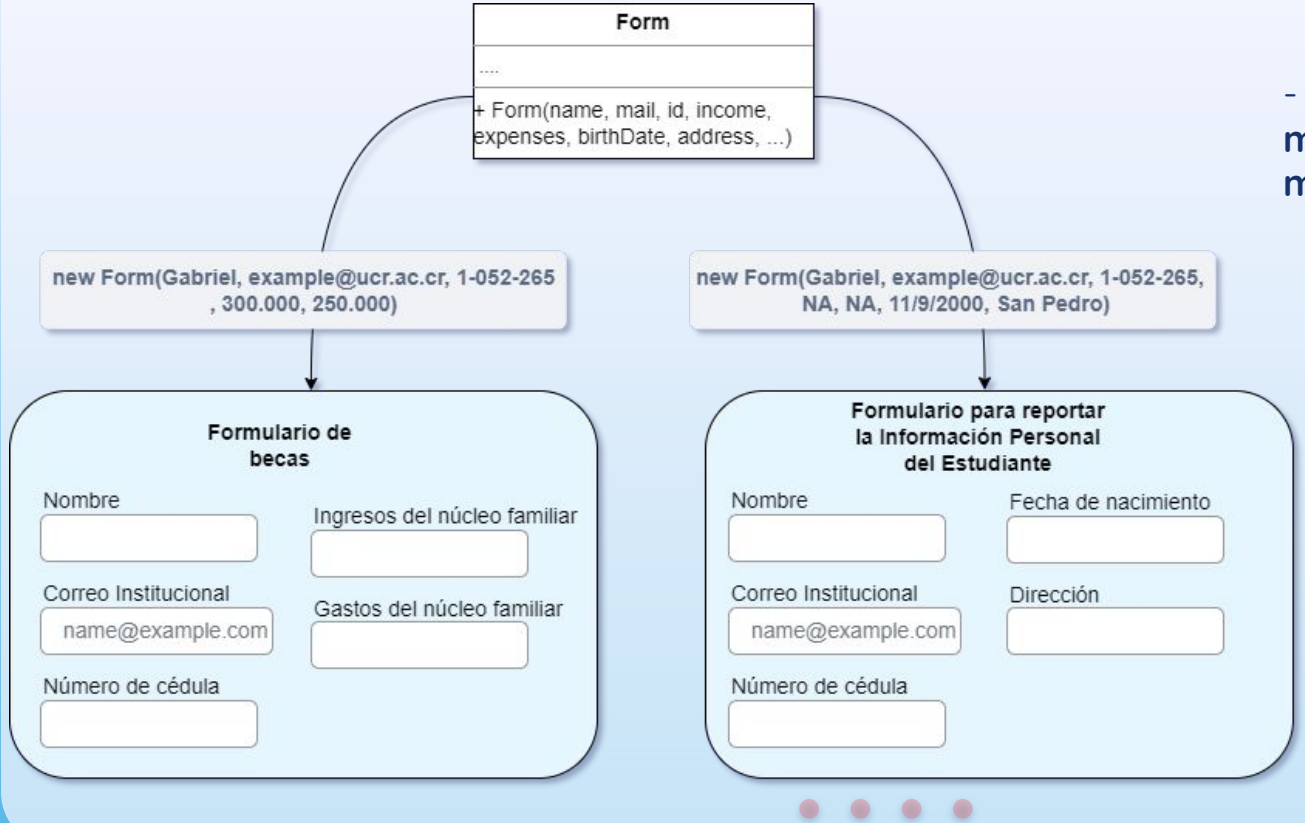
Constructor monstruoso

Gran cantidad de atributos

Ejemplo

- Solución **rápida** pero eventualmente **incómoda**; creación de muchas **subclases**

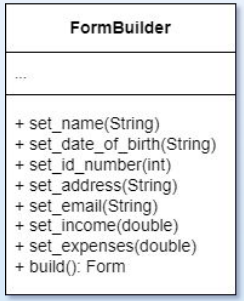


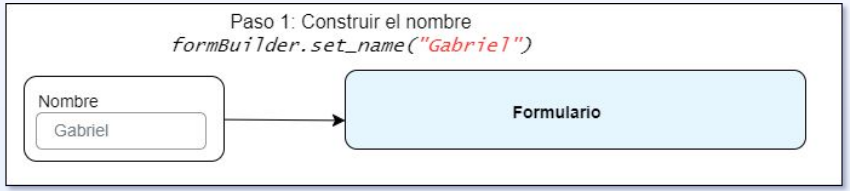


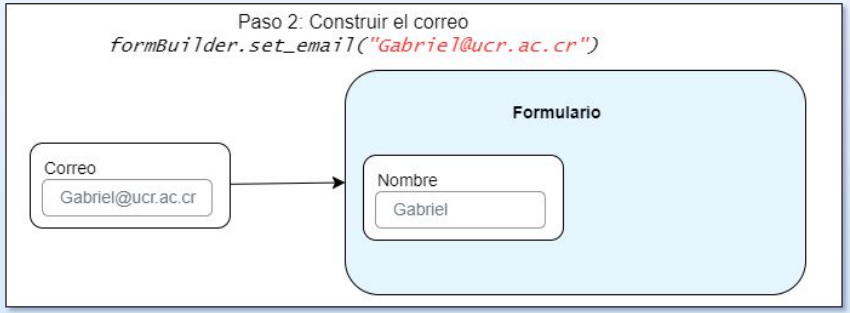
- Solución sencilla pero **mala práctica; constructor monstruoso**

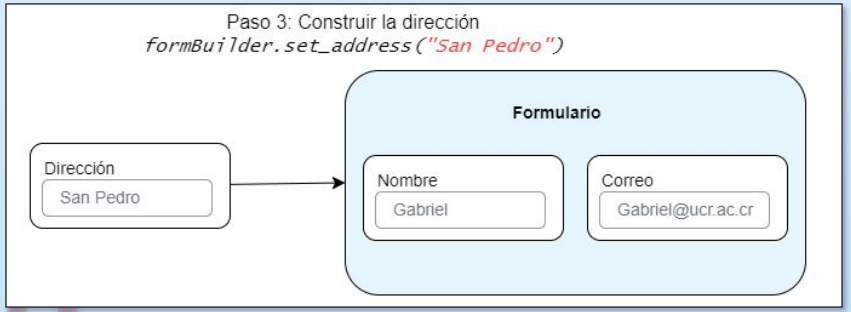
Solución

- Clases builders
- Paso a paso
- No requiere todos los pasos









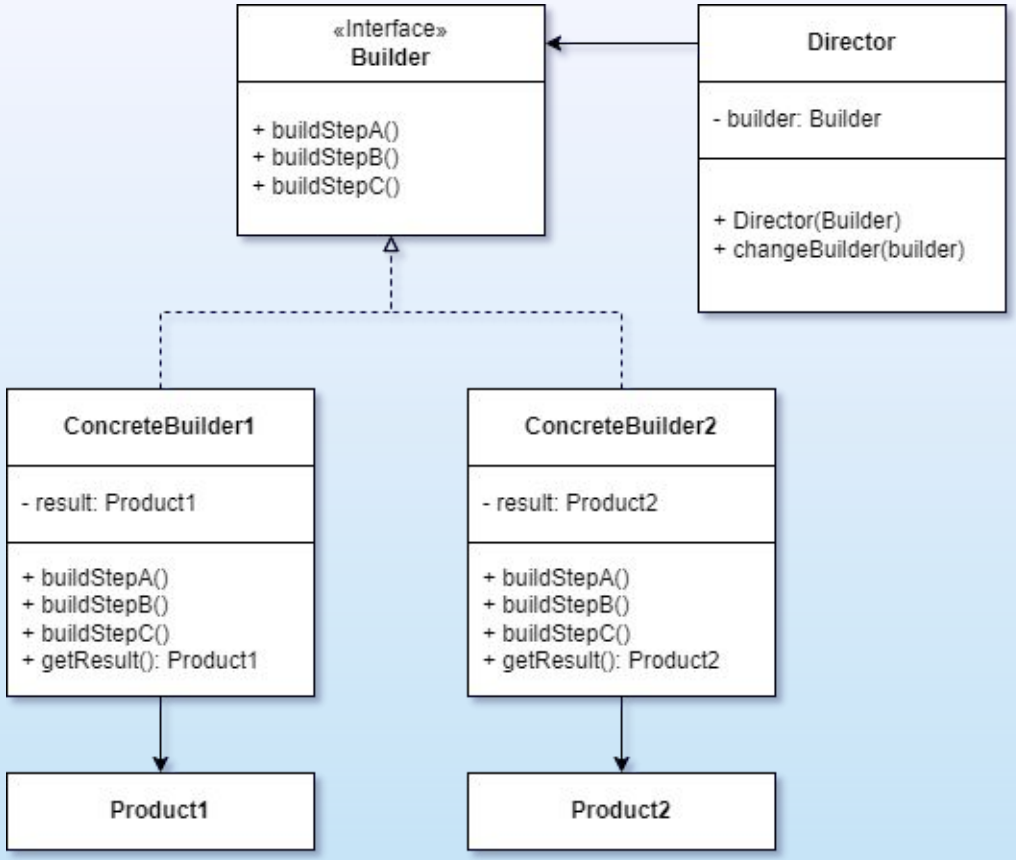
Estructura

- Builder interface: declara los pasos de construcción

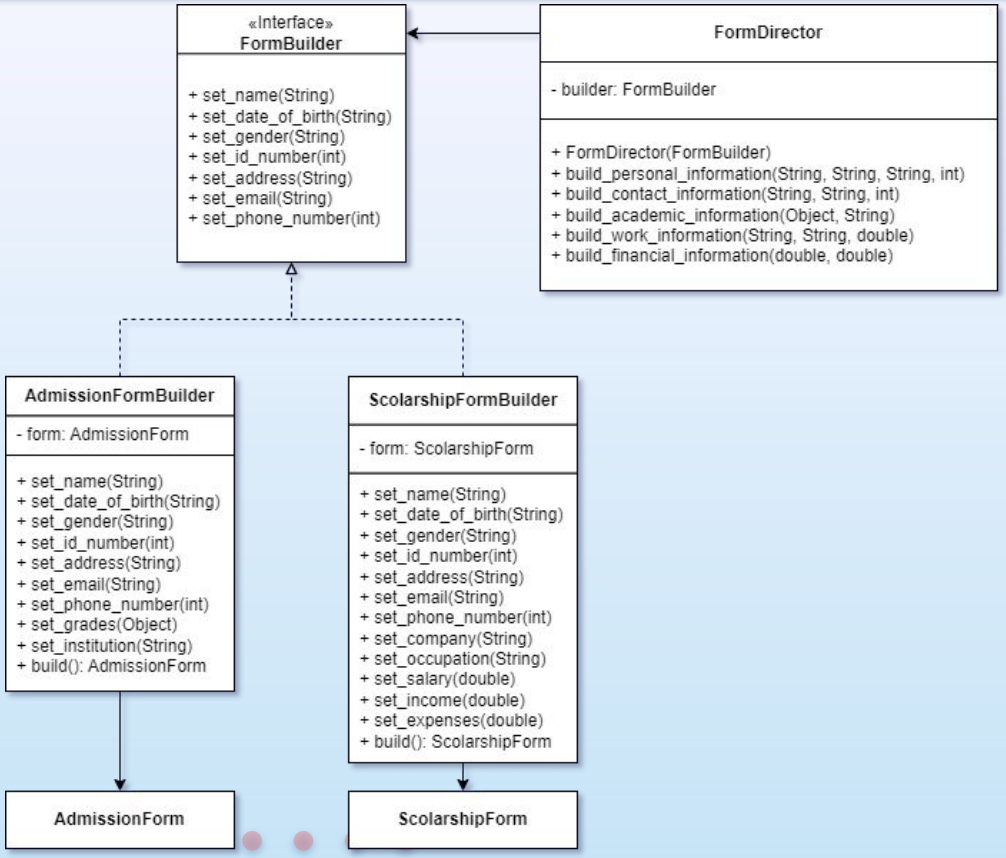
- ConcreteBuilder: implementan la interfaz Builder con diferentes pasos de construcción

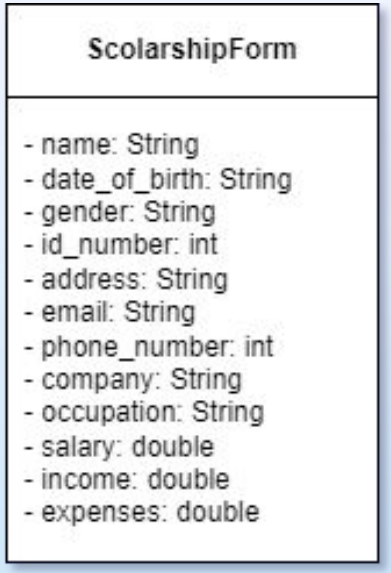
- Product: producido por el builder, pueden no tener una interfaz en común. Son los objetos resultantes

- Director; define el órden en el que se ejecutan los pasos de construcción



Ejemplo de estructura





Ejemplo de estructura

Ejemplo en código

FormBuilder Interface

FormBuilder .java

```java
public interface FormBuilder {
  public FormBuilder setName(String name);

  public FormBuilder setDateOfBirth(String date_of_birth);

  public FormBuilder setGender(String gender);

  public FormBuilder setIdNumber(int idNumber);

  public FormBuilder setAddress(String address);

  public FormBuilder setEmail(String email);

  public FormBuilder setPhoneNumber(int phoneNumber);
}

```

AdmissionFormBuilder Class

AdmissionFormBuilder.java

```java
public class AdmissionFormBuilder implements FormBuilder {

  private AdmissionForm form;

  public AdmissionFormBuilder() {
    this.form = new AdmissionForm();
  }

  public AdmissionFormBuilder setName(String name) {
    this.form.setName(name);
    return this;
  }

  public AdmissionFormBuilder setDateOfBirth(String date_of_birth) {
    this.form.setDateOfBirth(date_of_birth);
    return this;
  }

  public AdmissionFormBuilder setGender(String gender) {
    this.form.setGender(gender);
    return this;
  }

  public AdmissionFormBuilder setIdNumber(int id_number) {
    this.form.setIdNumber(id_number);
    return this;
  }

  public AdmissionFormBuilder setAddress(String address) {
    this.form.setAddress(address);
    return this;
  }

  public AdmissionFormBuilder setEmail(String email) {
    this.form.setEmail(email);
    return this;
  }

  public AdmissionFormBuilder setPhoneNumber(int phone_number) {
    this.form.setPhoneNumber(phone_number);
    return this;
  }

  public AdmissionFormBuilder setGrades(Object grades) {
    this.form.setGrades(grades);
    return this;
  }

  public AdmissionFormBuilder setInstitution(String institution) {
    this.form.setInstitution(institution);
    return this;
  }

  public AdmissionForm build() {
    return this.form;
  }
}
```

FormDirector Class

FormDirector.java

```java
public class FormDirector {

  private FormBuilder builder;
  // Constructor
  public FormDirector(FormBuilder builder) {
    this.builder = builder;
  }

  public FormBuilder getBuilder() {
    return this.builder;
  }

  public FormDirector buildPersonalInformation(
    String name,
    String date_of_birth,
    String gender,
    int id_number
  ) {
    this.builder.setName(name)
      .setDateOfBirth(date_of_birth)
      .setGender(gender)
      .setIdNumber(id_number);
    return this;
  }

  public FormDirector buildContactInformation(
    String address,
    String email,
    int phone_number
  ) {
    this.builder.setAddress(address).setEmail(email).setPhoneNumber(phone_number);
    return this;
  }

  public FormDirector buildAcademicInformation(Object grades, String institution) {
    if (this.builder instanceof AdmissionFormBuilder this_builder) {
      this_builder.setGrades(grades).setInstitution(institution);
    }
    return this;
  }

  public FormDirector buildWorkInformation(
    String company,
    String occupation,
    double salary
  ) {
    if (this.builder instanceof ScolarshipFormBuilder this_builder) {
      this_builder.setCompany(company).setOccupation(occupation).setSalary(salary);
    }
    return this;
  }

  public FormDirector buildFinancialInformation(double income, double expenses) {
    if (this.builder instanceof ScolarshipFormBuilder this_builder) {
      this_builder.setIncome(income).setExpenses(expenses);
    }
    return this;
  }
}
```

AdmissionForm Class

AdmissionForm.java

```java
public class AdmissionForm {

  // Attributes
  private String name;
  private String date_of_birth;
  private String gender;
  private int id_number;
  private String address;
  private String email;
  private int phone_number;
  private Object grades;
  private String institution;

  // Default Constructor
  public AdmissionForm() {
    this.name = "";
    this.date_of_birth = "";
    this.gender = "";
    this.id_number = 0;
    this.address = "";
    this.email = "";
    this.phone_number = 0;
    this.grades = null;
    this.institution = "";
  }
```

```java
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getDateOfBirth() {
    return date_of_birth;
  }

  public void setDateOfBirth(String date_of_birth) {
    this.date_of_birth = date_of_birth;
  }
  // set and get methods for each attribute...
}
```

Aplicación

```
1  AdmissionFormBuilder builder = new AdmissionFormBuilder();
2
3
4  FormDirector director = new FormDirector(builder)
5          .buildPersonalInformation("Martin Fowler", "1963/12/18", "Male", 442076792)
6          .buildContactInformation("Chicago, IL", "martin@martinfowler.com",
   5551234);
7
8
9  AdmissionForm form = builder.build();
10
```

Ventajas

- Permite construir objetos paso a paso y diferir los pasos de construcción o ejecución recursivamente.

- Código de construcción reutilizable para varias representaciones de los productos.

- **Single Responsibility Principle**: aísla el código complejo de construcción de la lógica de negocio del producto

- Los usuarios no necesitan preocuparse por los detalles completos de la construcción del objeto.

Desventajas

- La complejidad del código puede aumentar debido a que el patrón requiere la creación de nuevas clases.

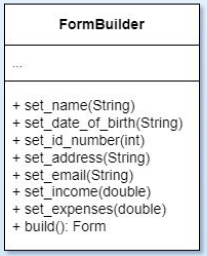
- Puede requerir la implementación de interfaces adicionales para los diferentes componentes del objeto

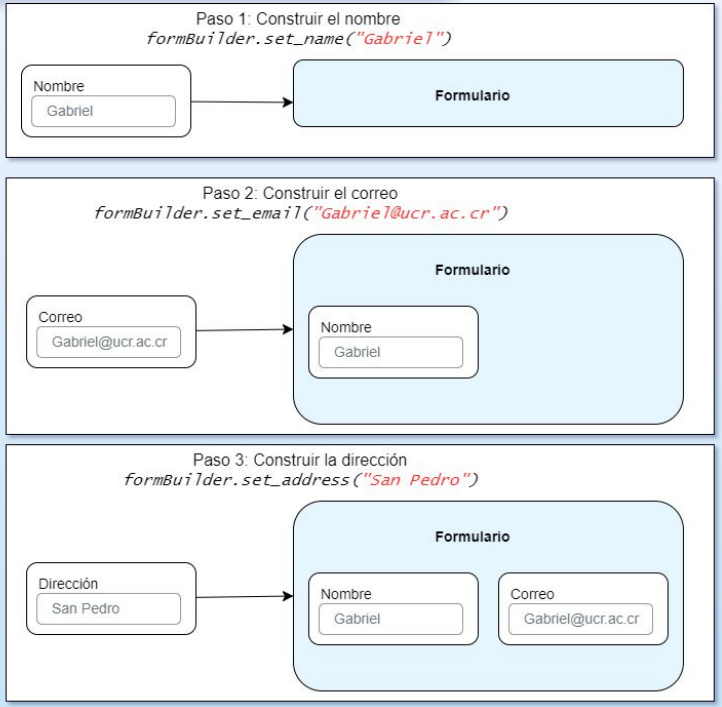
- Si el objeto que se está construyendo es simple, podemos estar agregando complejidad en lugar de evitarla.

- Puede ser menos eficiente que otros enfoques de construcción de objeto

Sugerencias de Implementación

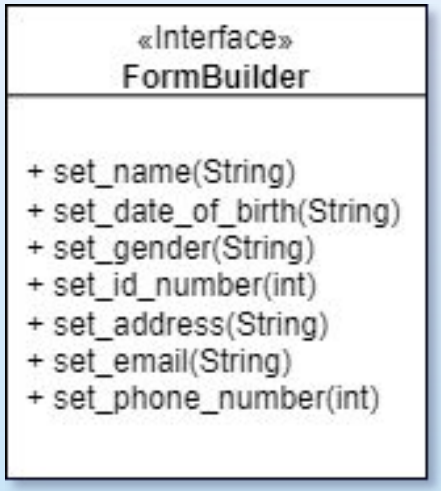
Se debe definir de forma adecuada los pasos de construcción de todas las representaciones de productos disponibles.





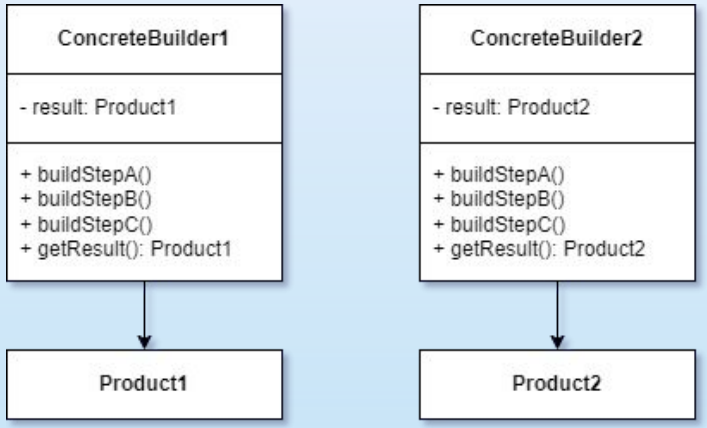
Sugerencias de Implementación

Estos pasos deben estar declarados en la interfaz del builder (Builder interface)



Sugerencias de Implementación

- Cree una clase constructora concreta (***ConcreteBuilder***) para cada una de las representaciones de los productos e implemente los pasos de construcción
- Se debe implementar un método para obtener el resultado de la construcción (***getResult()***) en cada clase *ConcreteBuilder*.



Sugerencias de Implementación

- Aunque no es necesario, considere implementar la clase ***Director***, puede ayudar a encapsular varias formas de construir un objeto *Product* utilizando el mismo objeto *ConcreteBuilder*.
- El cliente crea el objeto ***Director*** y el ***ConcreteBuilder***. Antes de iniciar la construcción el cliente pasa el *ConcreteBuilder* al *Director* mediante el constructor del *Director*.

```
1  AdmissionFormBuilder builder = new AdmissionFormBuilder();
2
3
4  FormDirector director = new FormDirector(builder)
5          .buildPersonalInformation("Martin Fowler", "1963/12/18", "Male", 442076792)
6          .buildContactInformation("Chicago, IL", "martin@martinfowler.com",
   5551234);
7
8
9  AdmissionForm form = builder.build();
10
```

Relación con otros patrones

Factory Method

Menos complicado y evoluciona a Builder

Abstract Factory

Crea objetos relacionados y retorna el producto inmediatamente. Builder crea objetos complejos paso a paso

Composite

El patrón Builder se puede utilizar para crear árboles complejos del patrón Composite.

Bridge

En este caso, la clase Director actúa como abstracción mientras que los diferentes builders actúan como implementaciones.

Bibliografía

Refactoring Guru. (s.f.). Builder. Recuperado el 14 de abril de 2023, de https://refactoring.guru/design-patterns/builder

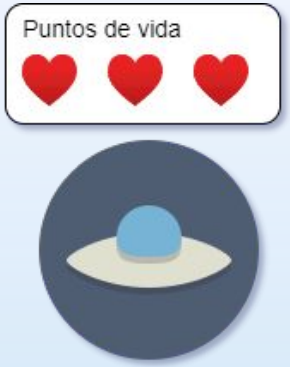
Hunt, J. (2013). Scala Design Patterns (1a ed.). Springer Publishing. https://doi-org.ezproxy.sibdi.ucr.ac.cr/10.1007/978-3-319-02192-8

¡Muchas gracias!

Actividad

Nave humana

Nave alienígena



Escudo: Permite esquivar (saltar) una pregunta. Sólo se puede usar una vez.

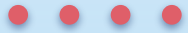
Cañón: Permite contra-atacar (lanzar) una pregunta. Sólo se puede usar una vez

1. ¿Qué es el patrón Builder?

A) Un patrón que define una familia de algoritmos, encapsula cada uno y los hace intercambiables.
B) Un patrón que proporciona una interfaz simplificada para un conjunto de interfaces más complejas.
C) Un patrón que separa la construcción de un objeto complejo de su representación y permite su creación paso a paso.
D) Un patrón que define una estructura de objetos en árbol para representar una jerarquía de clases.

2. ¿En qué tipo de proyecto es más útil utilizar el patrón Builder?

A) Proyectos con objetos simples y pocos atributos.
B) Proyectos con objetos complejos y muchos atributos.
C) Proyectos que no requieren la creación de objetos.
D) Proyectos con una arquitectura basada en microservicios.



3. ¿Qué ventaja proporciona el patrón Builder?

A) Simplifica la creación de objetos complejos.
B) Hace que la construcción de objetos sea más rápida.
C) Permite la creación de objetos sin necesidad de un constructor.
D) Mejora el rendimiento de la aplicación.

4. ¿Qué componentes principales tiene el patrón Builder?

A) Director, Constructor, Producto.
B) Interfaz, Implementación, Abstracción.
C) Cliente, Servidor, Base de datos.
D) Fachada, Adaptador, Decorador.

5. ¿Cuál es el papel de la interfaz Builder en el patrón Builder?

A) Define la estructura básica del objeto complejo.
B) Crea y devuelve el objeto complejo construido.
C) Define una interfaz para la construcción de partes del objeto complejo.
D) Define las partes individuales del objeto complejo y cómo se ensamblan.

6. ¿Qué otros patrones de diseño pueden trabajar en conjunto con el patrón Builder?

A) Singleton y Visitor.
B) Prototype y Abstract Factory.
C) Decorator y Adapter.
D) Strategy y Chain of Responsibility.

7. ¿Cuál es el beneficio de usar el patrón Builder en lugar de un constructor estándar?

A) El patrón Builder reduce el acoplamiento entre los objetos.
B) El patrón Builder reduce la complejidad del código de la aplicación.
C) El patrón Builder proporciona una interfaz más fácil de usar para crear objetos.
D) El patrón Builder permite la creación de objetos sin la necesidad de un constructor.

8. ¿Qué ocurre si se omite el uso del patrón Builder en la construcción de objetos complejos?

A) El código se vuelve más difícil de leer y mantener.
B) La creación de objetos puede ser más lenta y menos eficiente.
C) El acoplamiento entre objetos puede aumentar.
D) Es imposible crear objetos complejos sin el patrón Builder.

9. ¿Qué es el objeto Producto en el patrón Builder?

A) El objeto que el constructor construye.
B) El objeto que el Director utiliza para construir el objeto final.
C) El objeto que el Cliente utiliza para acceder al objeto final.
D) El objeto que el patrón Builder utiliza para definir la interfaz de construcción.

10. ¿Cuándo es apropiado utilizar el patrón Builder?

A) Cuando se desea crear objetos complejos paso a paso.
B) Cuando se desea crear una interfaz simplificada para un conjunto de interfaces más complejas.
C) Cuando se desea encapsular una familia de algoritmos y hacerlos intercambiables.
D) Cuando se desea mejorar el rendimiento de la aplicación.

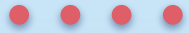
11. ¿Cómo se puede utilizar el patrón Builder para crear diferentes variantes de un objeto complejo?

A) Implementando diferentes constructores con diferentes parámetros.
B) Creando diferentes subclases (builders) de la clase Builder.
C) Definiendo diferentes métodos build() en la clase Builder.
D) Utilizando diferentes implementaciones de las partes del objeto en la clase Builder.

11. ¿Cómo se puede utilizar el patrón Builder para crear diferentes variantes de un objeto complejo?

A) Implementando diferentes constructores con diferentes parámetros.
B) Creando diferentes subclases (builders) de la clase Builder.
C) Definiendo diferentes métodos build() en la clase Builder.
D) Utilizando diferentes implementaciones de las partes del objeto en la clase Builder.






12. ¿Qué sucede si se llama al método build() de un objeto Builder antes de que se hayan definido todas sus partes?

A) Se produce un error de compilación.
B) Se devuelve un objeto incompleto con valores predeterminados.
C) Se devuelve un objeto nulo.
D) El método build() espera hasta que se definan todas las partes antes de devolver el objeto.

13. ¿Cuál es el papel del Director en el patrón Builder?

A) Define la estructura de objetos en árbol para representar una jerarquía de clases.
B) Define una interfaz simplificada para un conjunto de interfaces más complejas.
C) Coordina la construcción del objeto complejo utilizando los Constructores.
D) Encapsula una familia de algoritmos y los hace intercambiables.

14. ¿Cómo se diferencia el patrón Builder del patrón Abstract Factory?

A) El patrón Builder permite crear objetos complejos paso a paso, mientras que el patrón Abstract Factory crea familias de objetos relacionados.
B) El patrón Builder encapsula una familia de algoritmos y los hace intercambiables, mientras que el patrón Abstract Factory define una interfaz para crear una familia de objetos relacionados.
C) El patrón Builder se enfoca en la construcción de objetos complejos, mientras que el patrón Abstract Factory se enfoca en la creación de objetos relacionados.
D) El patrón Builder proporciona una interfaz simplificada para un conjunto de interfaces más complejas, mientras que el patrón Abstract Factory separa la creación de objetos de su implementación.

15. ¿Cuál es una desventaja del patrón Builder?

A) Puede aumentar el acoplamiento entre los objetos.
B) Puede ser más difícil de entender y utilizar que un constructor estándar.
C) Genera un código más complejo y difícil de mantener.

16. ¿Qué método se utiliza en el patrón Builder para agregar partes a un objeto complejo?

A) addPart()
B) buildPart()
C) createPart()
D) setPart()

17. ¿Cómo se llama la clase que representa el objeto complejo que se está construyendo?

A) Builder
B) Product
C) Director
D) ConcreteBuilder

18. ¿Cómo puede el patrón Builder ayudar a evitar la creación de objetos con valores inválidos o inconsistentes?

A) Validando cada parte del objeto en el método build().
B) Utilizando un objeto Director para definir todas las partes del objeto de forma consistente.
C) Utilizando un objeto Singleton para garantizar que las partes del objeto sean coherentes.
D) No puede evitar la creación de objetos con valores inválidos o inconsistentes.

19. ¿En qué consiste el proceso de construcción en el patrón Builder?

A) Se definen todas las partes del objeto complejo y se agregan al objeto.
B) Se construyen las partes del objeto complejo de forma independiente y se agregan al objeto.
C) Se construyen las partes del objeto complejo de forma secuencial, agregándolas al objeto a medida que se completan.
D) Se agregan las partes del objeto complejo de forma independiente y se definen sus propiedades.