Taller Modelado UML con Patrones de Diseño

Por:

Jairo Yeison Sánchez López

Docente:

Christian David Jaimes Acevedo

Facultad de Ingenierías

Ingeniería de Sistemas

Tecnológico de Antioquia - Institución Universitaria

Los Patios , Colombia

2025

Contenido

Introducción 4
PATRON BUILDER 5
Identificación del patrón 5
Justificación 7
PATRÓN SINGLETON 7
Identificación del patrón 7
Justificación 7
Justificación 8
PATRÓN FACTORY METHOD (O CREACIÓN CONTROLADA) 9
Identificación 9
Justificación 10
Explicación del Código – Proyecto CookMaster 11
CLASE ARCHIVORECETASUNIFICADO (SINGLETON) 11
Explicación 11
INTERFAZ CONSTRUCTORRECETASPASOAPASO (PARTE DEL BUILDER) 12

```java
public interface ConstructorRecetasPasoAPaso {

    void comenzarNuevaReceta();
    void incorporarComponente(String etiqueta, String cantidad);
    void anotarPaso(String descripcion);
    RecetaGeneral finalizarReceta();
}
```

Explicación 12
CLASE CONSTRUCTORPOSTRECASERO (IMPLEMENTACIÓN CONCRETA DEL BUILDER) 12
Explicación 12
CLASE COORDINADORELABORACIONCULINARIA (DIRECTOR DEL BUILDER) 12
Explicación 13
CLASE FABRICARECETASSIMPLE (FACTORY METHOD) 13
Explicación 13
CLASE RECETAGENERAL (SUPERCLASE) 13

Explicación 14
CLASES HIJAS (POLIMORFISMO Y SOBRESCRITURA) 14
Explicación 14
CLASE INSTRUCCIONPROCESO Y ELEMENTOINGREDIENTE (COMPOSICIÓN) 15
Explicación 15
CLASE INICIADORCULINARIO (CAPA DE EJECUCIÓN) 15
Explicación 15
CONCLUSION 20

Introducción

El presente documento analiza una arquitectura desarrollada en Java cuyo propósito es modelar un sistema orientado a la elaboración de recetas culinarias. El proyecto integra diversas entidades, jerarquías y procesos que permiten crear recetas mediante pasos estructurados, manipular ingredientes, estandarizar tipos de recetas y centralizar la lógica de construcción. A partir del código proporcionado, se identifican los patrones de diseño aplicados —implícita o explícitamente— y se justifica su pertinencia técnica para garantizar extensibilidad, reutilización de código y claridad en la construcción de nuevas funcionalidades.

Además, se presentan los diagramas UML asociados (diagrama de clases y diagramas derivados) que permiten visualizar las relaciones entre las entidades, así como abstraer los componentes esenciales del sistema. Este análisis facilita comprender la estructura, responsabilidades y colaboraciones entre las clases, apoyando una documentación clara y un diseño mantenible.

PATRON BUILDER

Identificación del patrón

El Builder aparece en el código a través de clases como:

- ConstructorRecetasPasoAPaso (interfaz / director)
- ConstructorPostreCasero (builder concreto)
- El proceso de construcción paso a paso:
  - comenzarNuevaReceta()
  - incorporarComponente()
  - anotarPaso()
  - finalizarReceta()

```
public interface ConstructorRecetasPasoAPaso {

    void comenzarNuevaReceta();
    void incorporarComponente(String etiqueta, String cantidad);
    void anotarPaso(String descripcion);
    RecetaGeneral finalizarReceta();
}
```

Builder concreto.

```java
public class ConstructorPostreCasero implements ConstructorRecetasPasoAPaso {

    private RecetaDulceTradicional recetaActual;

    @Override
    public void comenzarNuevaReceta() {
        recetaActual = new RecetaDulceTradicional("Postre Casero Artesanal");
    }

    @Override
    public void incorporarComponente(String etiqueta, String cantidad) {
        recetaActual.agregarComponente(new ElementoIngrediente(etiqueta, cantidad));
    }

    @Override
    public void anotarPaso(String descripcion) {
        recetaActual.agregarPaso(new InstruccionProceso(descripcion));
    }

    @Override
    public RecetaGeneral finalizarReceta() {
        return recetaActual;
    }
}
```

"Director" que usa el Builder:

```java
public class CoordinadorElaboracionCulinaria {

    public RecetaGeneral generarPostreClasico(ConstructorRecetasPasoAPaso constructor) {

        constructor.comenzarNuevaReceta();

        constructor.incorporarComponente("Azucar blanca", "200g");
        constructor.incorporarComponente("Harina suave", "300g");
        constructor.incorporarComponente("Huevos frescos", "2 unidades");

        constructor.anotarPaso("Integrar todos los componentes en un tazon amplio.");
        constructor.anotarPaso("Mezclar hasta lograr una textura homogenea.");
        constructor.anotarPaso("Hornear durante 30 minutos a temperatura media.");

        return constructor.finalizarReceta();
    }
}
```

Justificación

El patrón Builder se usa porque la construcción de una receta:

- requiere **varios pasos definidos**
- debe permitir **diferentes tipos de recetas**
- implica **variaciones en los ingredientes, pasos y proceso general**

Builder evita tener constructores gigantes o métodos con 12 parámetros, y además permite que el proceso cambie según el tipo de receta, sin modificar el coordinador.

Justificación formal:
Se usa Builder debido a que el objeto RecetaGeneral requiere un proceso de construcción complejo y secuencial. Este patrón permite separar la lógica de armado del producto final, manteniendo el código extensible y evitando duplicación en la creación de distintas recetas.

PATRÓN SINGLETON

Identificación del patrón

El Singleton aparece en la clase encargada de almacenar todas las recetas:

Justificación

Este patrón se usa porque:

- solo debe existir **una única fuente oficial de recetas**
- varios objetos necesitan acceso simultáneo al archivo
- se requiere consistencia y evitar múltiples colecciones no sincronizadas

Justificación formal:
Se emplea el patrón Singleton para garantizar que exista una única instancia global que gestione el almacenamiento de recetas. Esto asegura integridad en los datos y evita inconsistencias que podrían generarse si cada clase mantuviera su propio repositorio.

```java
import java.util.ArrayList;
import java.util.List;

public class ArchivoRecetasUnificado {

    private static ArchivoRecetasUnificado referenciaCompartida;
    private List<RecetaGeneral> coleccionRecetasGuardadas;

    private ArchivoRecetasUnificado() {
        coleccionRecetasGuardadas = new ArrayList<>();
    }

    public static ArchivoRecetasUnificado accederArchivo() {
        if (referenciaCompartida == null) {
            referenciaCompartida = new ArchivoRecetasUnificado();
        }
        return referenciaCompartida;
    }

    public void registrarReceta(RecetaGeneral receta) {
        coleccionRecetasGuardadas.add(receta);
    }

    public List<RecetaGeneral> consultarRecetas() {
        return coleccionRecetasGuardadas;
    }
}

```

Justificación

Este patrón se usa porque:

- solo debe existir **una única fuente oficial de recetas**
- varios objetos necesitan acceso simultáneo al archivo
- se requiere consistencia y evitar múltiples colecciones no sincronizadas

Justificación formal:
Se emplea el patrón Singleton para garantizar que exista una única instancia global que gestione el almacenamiento de recetas. Esto asegura integridad en los datos y evita inconsistencias que podrían generarse si cada clase mantuviera su propio repositorio

.

PATRÓN FACTORY METHOD (O CREACIÓN CONTROLADA)

Identificación

El patrón aparece cada vez que:

- un objeto no se crea directamente mediante `new`
- la creación depende de una subclase o un método especializado

Ejemplo:

```
public abstract class RecetaGeneral {

   private String titulo;


   public RecetaGeneral(String titulo) {

      this.titulo = titulo;

   }


   public abstract String tipoReceta();

}
```

Y subclases:

```
public class RecetaPreparacionCarnes extends RecetaGeneral {


   public RecetaPreparacionCarnes(String titulo) {

      super(titulo);

   }


   @Override
```

```
    public String tipoReceta() {

        return "Carnes";

    }

}
```

Aquí tipoReceta() es un **método que decide el tipo**, sustituyendo un switch gigante o instancias manuales.

Justificación

El Factory Method se utiliza para:

- permitir que cada receta defina su propio tipo sin condicionales
- delegar en las subclases el comportamiento particular
- permitir extensión sin modificar el código base (Open/Closed Principle)

Justificación formal:
La jerarquía de clases utiliza el patrón Factory Method al delegar en cada subclase la creación del comportamiento específico como tipoReceta(). Esto facilita agregar nuevos tipos de recetas sin alterar las clases ya existentes, asegurando bajo acoplamiento.

Explicación del Código – Proyecto CookMaster

A continuación se presenta una explicación detallada de cada parte del código desarrollado en este ejercicio. El objetivo es comprender el propósito de cada clase, cómo interactúan entre sí y qué patrones de diseño se aplicaron.

CLASE ARCHIVORECETASUNIFICADO (SINGLETON)

```java
import java.util.ArrayList;
import java.util.List;

public class ArchivoRecetasUnificado {

    private static ArchivoRecetasUnificado referenciaCompartida;
    private List<RecetaGeneral> coleccionRecetasGuardadas;

    private ArchivoRecetasUnificado() {
        coleccionRecetasGuardadas = new ArrayList<>();
    }

    public static ArchivoRecetasUnificado accederArchivo() {
        if (referenciaCompartida == null) {
            referenciaCompartida = new ArchivoRecetasUnificado();
        }
        return referenciaCompartida;
    }
```

Explicación

Esta clase implementa el patrón **Singleton**, lo que significa que solo puede existir **una única instancia** del archivo de recetas en toda la aplicación.
Con este patrón se garantiza que todas las recetas creadas se guarden en el mismo repositorio central.

El método accederArchivo() controla la creación de la instancia y asegura que siempre se retorne la misma.

INTERFAZ CONSTRUCTORRECETASPASOAPASO (PARTE DEL BUILDER)

```java
public interface ConstructorRecetasPasoAPaso {

    void comenzarNuevaReceta();
    void incorporarComponente(String etiqueta, String cantidad);
    void anotarPaso(String descripcion);
    RecetaGeneral finalizarReceta();
}
```

Explicación

Esta interfaz define los pasos necesarios para construir una receta. Sirve como base del **patrón Builder**, que permite construir objetos complejos paso a paso sin depender de una estructura rígida.

Cualquier clase que implemente esta interfaz podrá construir recetas siguiendo el mismo proceso general.

CLASE CONSTRUCTORPOSTRECASERO (IMPLEMENTACIÓN CONCRETA DEL BUILDER)

Código (fragmento)

```java
    @Override
    public void comenzarNuevaReceta() {
        recetaActual = new RecetaDulceTradicional("Postre Casero Artesanal");
    }
```

Explicación

Esta clase implementa los métodos de la interfaz Builder y define cómo se construye paso a paso una receta dulce:

- Crea una instancia de RecetaDulceTradicional
- Agrega ingredientes
- Agrega pasos
- Al final devuelve la receta terminada

Gracias al Builder, podemos crear recetas paso a paso sin depender de un constructor gigante ni de configuraciones rígidas.

CLASE COORDINADORELABORACIONCULINARIA (DIRECTOR DEL BUILDER)

```java
public class CoordinadorElaboracionCulinaria {

    public RecetaGeneral generarPostreClasico(ConstructorRecetasPasoAPaso constructor) {

        constructor.comenzarNuevaReceta();

        constructor.incorporarComponente("Azucar blanca", "200g");
        constructor.incorporarComponente("Harina suave", "300g");
        constructor.incorporarComponente("Huevos frescos", "2 unidades");

        constructor.anotarPaso("Integrar todos los componentes en un tazon amplio.");
        constructor.anotarPaso("Mezclar hasta lograr una textura homogenea.");
        constructor.anotarPaso("Hornear durante 30 minutos a temperatura media.");

        return constructor.finalizarReceta();
    }
}
```

Explicación

Esta clase funciona como **Director** del Builder, es decir, coordina el procedimiento para construir una receta completa.

No sabe cómo se construye internamente la receta; solo llama en orden a los pasos del builder. Esto permite cambiar la receta (dulce, salada, infusión…) simplemente cambiando el constructor que se le pasa.

CLASE FABRICARECETASSIMPLE (FACTORY METHOD)

Explicación

se usa el **Factory Method**, un patrón que encapsula la lógica de creación de objetos. En lugar de que el usuario decida qué clase instanciar, la fábrica lo hace según el tipo solicitado.

Esto facilita:

- agregar nuevos tipos de recetas
- centralizar la lógica de creación
- no depender de múltiples new en el código principal

CLASE RECETAGENERAL (SUPERCLASE)

Código(Fragmento)

```java
import java.util.ArrayList;
import java.util.List;

public class RecetaGeneral {

    protected String tituloReceta;
    protected List<ElementoIngrediente> listaComponentes;
    protected List<InstruccionProceso> secuenciaPreparacion;
    protected String tiempoTotal;
```

Explicación

Es la clase base de todas las recetas. Contiene:

- título
- lista de ingredientes
- lista de pasos
- tiempo estimado

Incluye métodos para agregar componentes y pasos, mostrar la receta y sobrecargas para mostrarla de diferentes maneras.

Las demás recetas (dulce, carnes, infusión) heredan de esta clase y pueden personalizar su comportamiento.

CLASES HIJAS (POLIMORFISMO Y SOBRESCRITURA)

Ejemplo:

```java
    @Override
    public void mostrarCompleta() {
        System.out.println("=== Receta (DULCE) ===");
        super.mostrarCompleta();
        System.out.println("(Esta receta es ideal para postres caseros)");
    }
}
```

Explicación

Cada tipo de receta sobrescribe métodos como mostrarCompleta() para agregar mensajes específicos.
Esto demuestra **polimorfismo**, ya que cada receta se comporta diferente al llamarse el mismo método.

CLASE INSTRUCCIONPROCESO Y ELEMENTOINGREDIENTE (COMPOSICIÓN)

```
public void agregarPaso(InstruccionProceso paso) {
   this.secuenciaPreparacion.add(paso);
}
```

Explicación

Una receta **contiene** ingredientes y pasos.
Estos elementos no existen por sí mismos fuera de la receta, lo cual representa una **composición**.

Además, ambas clases tienen sobrecarga de constructores y del método toString.

CLASE INICIADORCULINARIO (CAPA DE EJECUCIÓN)

```
public class IniciadorCulinario {

    public static void main(String[] args) {

        // aca puse el Builder y el Director
        ConstructorRecetasPasoAPaso constructor = new ConstructorPostreCasero();
        CoordinadorElaboracionCulinaria coordinador = new CoordinadorElaboracionCulinaria();
        RecetaGeneral recetaBuilder = coordinador.generarPostreClasico(constructor);
```

Explicación

Es la clase main. Aquí se demuestra cómo todo el sistema trabaja junto:

1. Se usa Builder + Director para crear una receta completa.
2. Se usa Factory Method para crear otra receta.
3. Se guarda todo en el Singleton ArchivoRecetasUnificado.
4. Se muestran las recetas almacenadas.

Actúa como punto de entrada del programa.

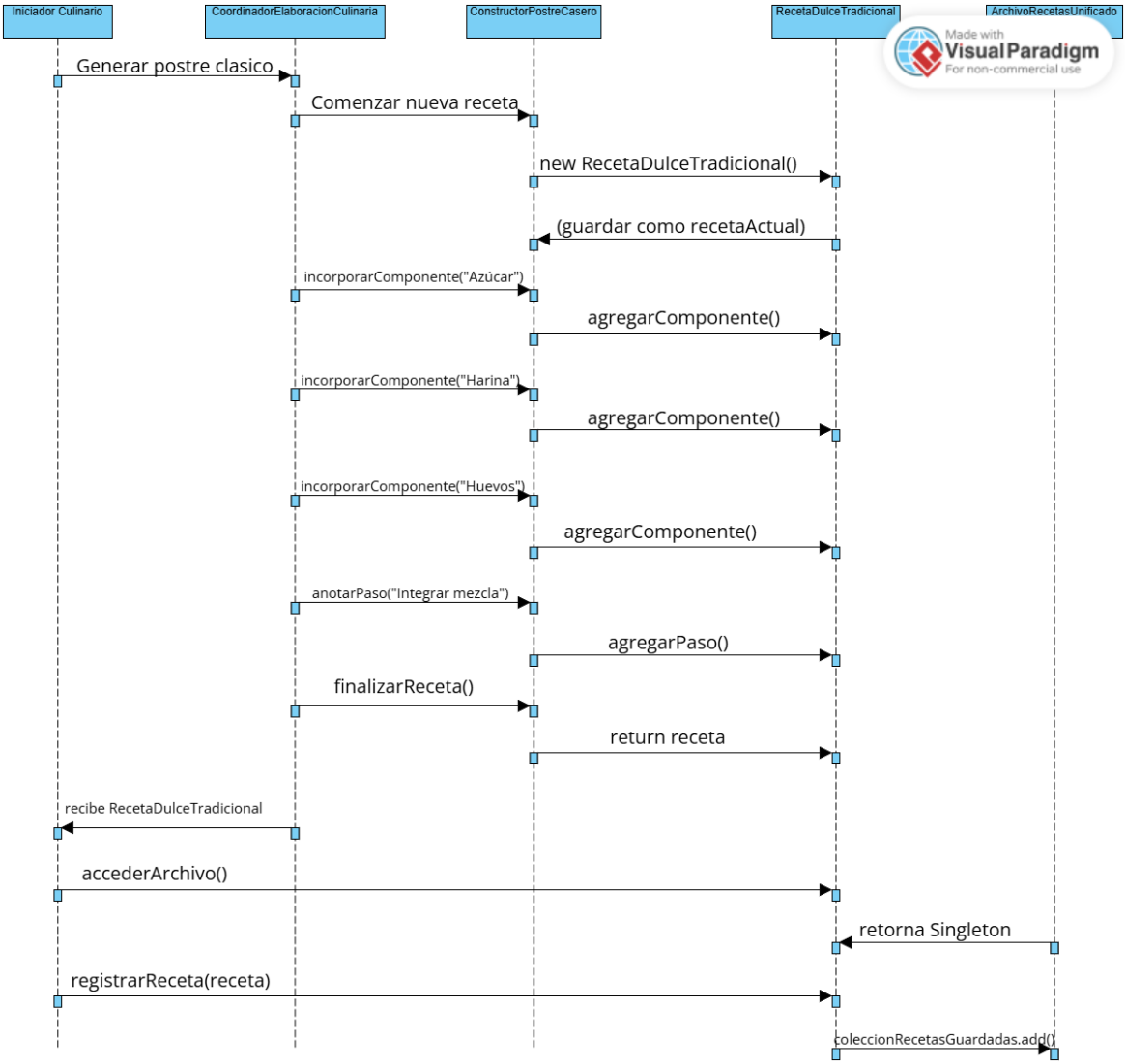

Ilustración 1. Diagrama UML Diagrama de secuencia

NOTA: Diagrama propio elaborado en Visual Paradigm

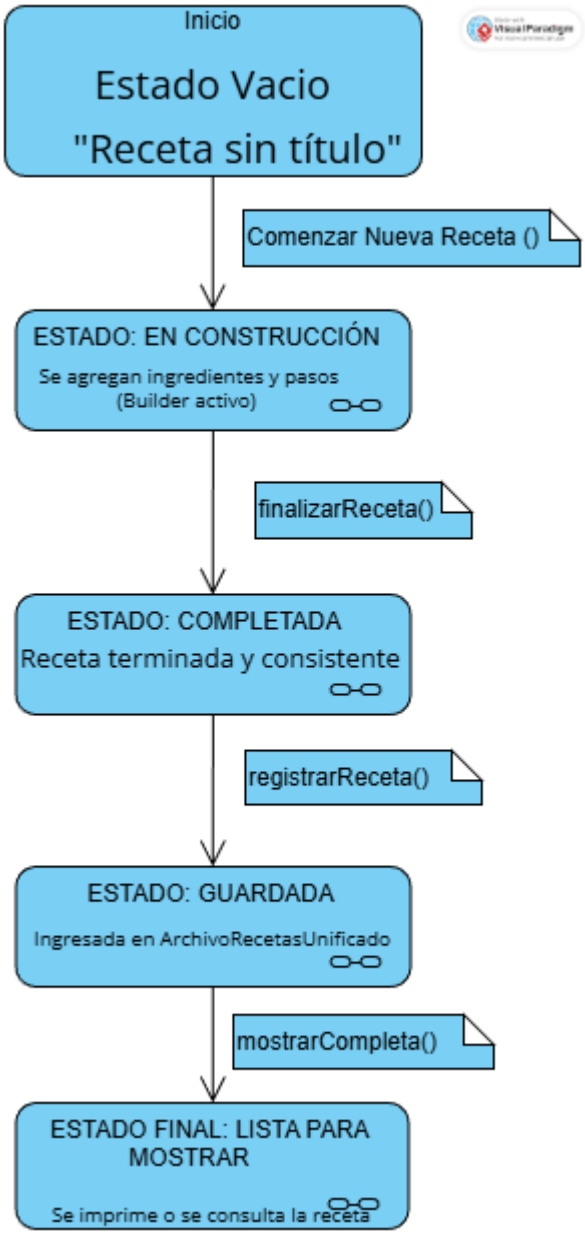

Ilustración 2. Diagrama UML Diagrama de estado

NOTA: Diagrama propio elaborado en visual paradigm

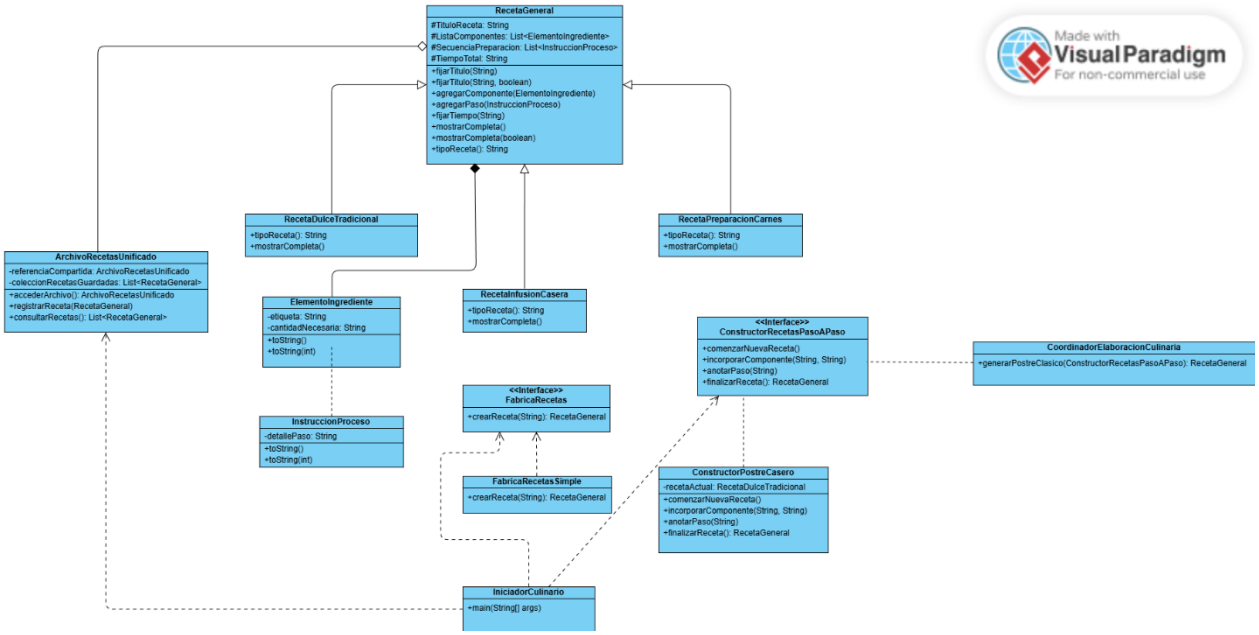

Ilustración 3. Diagrama UML Diagrama de clases

NOTA: Diseño propio elaborado en visual paradigm

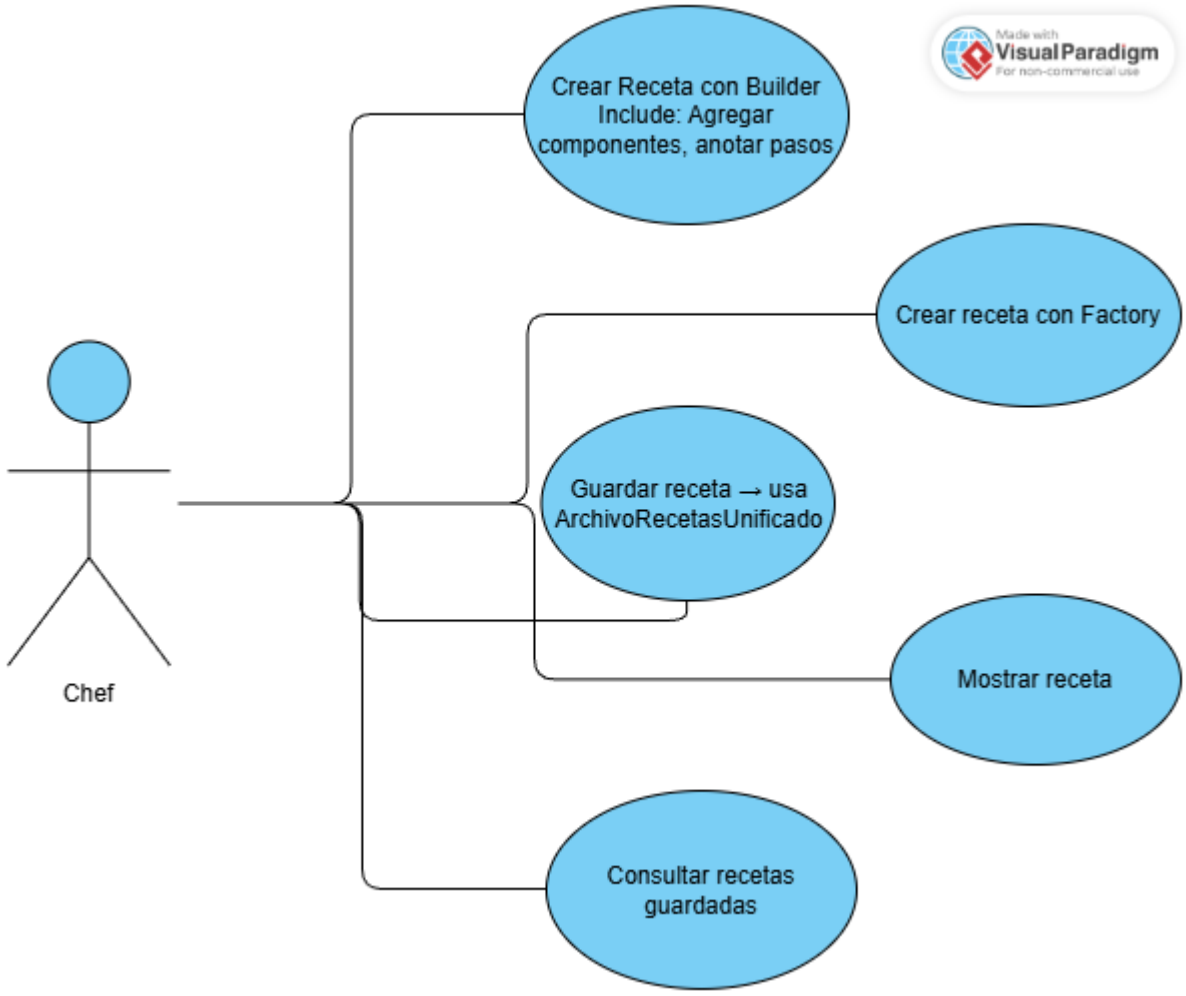

Ilustración 4. Diagrama UML Diagrama de caso de uso

NOTA: Diseño propio elaborado en visual paradigm

CONCLUSION

El desarrollo del código y la elaboración de los diagramas UML permitieron comprender de manera más concreta cómo se aplican los principios de la programación orientada a objetos dentro de un proyecto real. El uso de patrones como Builder, Factory Method y Singleton mostró cómo es posible estructurar un sistema que sea entendible, modular y fácil de ampliar sin modificar su funcionamiento principal.
Asimismo, los diagramas facilitaron visualizar la interacción entre las clases y el flujo completo para crear y registrar una receta, lo que contribuyó a entender mejor la lógica del programa y la importancia de una buena arquitectura. En conjunto, este trabajo reforzó conocimientos esenciales y permitió apreciar cómo un diseño ordenado mejora la claridad, la organización y la capacidad de crecimiento del software.