# Implementación del Patrón Abstract Factory en el Sistema de Citas Médicas

## 1. Introducción

En el desarrollo de nuestro sistema de citas médicas, nos enfrentamos al desafío de manejar diferentes tipos de médicos, pacientes y citas. Para resolver este problema de manera eficiente, decidimos implementar el patrón Abstract Factory. Este patrón nos permite crear familias de objetos relacionados (como médicos, pacientes y citas) sin necesidad de especificar sus clases concretas, lo que hace que el sistema sea más flexible y fácil de mantener.

## 2. ¿Qué es el Patrón Abstract Factory?

El patrón Abstract Factory es un patrón de diseño creacional que proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. En otras palabras, nos permite crear grupos de objetos que están diseñados para trabajar juntos, como médicos generales con pacientes generales y citas generales, o dentistas con pacientes dentales y citas dentales.

## 3. Participantes del Patrón Abstract Factory

En nuestra implementación, los participantes son:

1. **FabricaAbstracta:**
    a. Es la interfaz que define los métodos para crear médicos, pacientes y citas.
2. **FabricaMedicosGenerales y FabricaDentistas:**
    a. Son las implementaciones concretas de FabricaAbstracta. Cada una crea una familia específica de objetos (médicos generales o dentistas, pacientes generales o dentales, y citas generales o dentales).
3. **Medico, Paciente, Cita:**
    a. Son las interfaces que definen los métodos comunes para los médicos, pacientes y citas.
4. **MedicoGeneral, Dentista, PacienteGeneral, PacienteDental, CitaGeneral, CitaDentista:**
    a. Son las implementaciones concretas de las interfaces Medico, Paciente y Cita.
5. **Main (Cliente):**
    a. Es el código que utiliza la FabricaAbstracta para crear y utilizar las familias de objetos.

## 4. ¿Cómo Funciona el Patrón Abstract Factory?

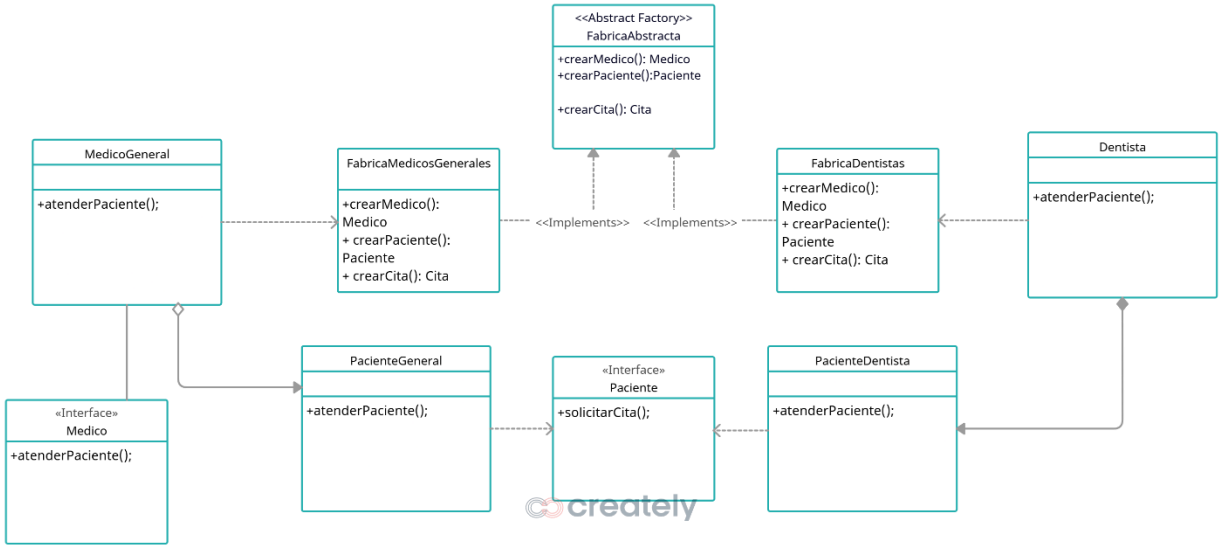
El patrón Abstract Factory funciona de la siguiente manera:

1. El Cliente (Main) utiliza una FabricaAbstracta para crear objetos.
2. La FabricaAbstracta define métodos para crear cada tipo de objeto (médico, paciente, cita).
3. Las fábricas especificas (FabricaMedicosGenerales, FabricaDentistas) implementan estos métodos para crear objetos específicos (médicos generales, dentistas, etc.).

## 5. Implementación en el Proyecto

En nuestro sistema de citas médicas, implementamos el patrón Abstract Factory para manejar dos tipos de médicos: médicos generales y dentistas. Cada tipo de médico tiene su propia familia de objetos relacionados (pacientes y citas).

## 6. Diagrama UML




## 7. Código e implementación

### 1. Clases abstractas

```
public interface FabricaAbstracta {
    Medico crearMedico();
    Paciente crearPaciente();
    Cita crearCita();
}

public interface Medico {
    void atenderPaciente();
```

```
}


public interface Paciente {
    void solicitarCita();
}


public interface Cita {
    void programar();
}
```

2. **Fabricas**

```
public class FabricaMedicosGenerales implements FabricaAbstracta {
    @Override
    public Medico crearMedico() {
        return new MedicoGeneral();
    }

    @Override
    public Paciente crearPaciente() {
        return new PacienteGeneral();
    }

    @Override
    public Cita crearCita() {
        return new CitaGeneral();
    }
}

public class FabricaDentistas implements FabricaAbstracta {
    @Override
    public Medico crearMedico() {
        return new Dentista();
    }

    @Override
    public Paciente crearPaciente() {
        return new PacienteDental();
    }

    @Override
    public Cita crearCita() {
        return new CitaDentista();
    }
}
```

3. **Productos**

```
public class MedicoGeneral implements Medico {
    @Override
    public void atenderPaciente() {
        System.out.println("Médico general atendiendo paciente...");
    }
}

public class Dentista implements Medico {
    @Override
    public void atenderPaciente() {
        System.out.println("Dentista atendiendo paciente...");
    }
}

public class PacienteGeneral implements Paciente {
    @Override
    public void solicitarCita() {
        System.out.println("Paciente general solicitando cita...");
    }
}

public class PacienteDental implements Paciente {
    @Override
    public void solicitarCita() {
        System.out.println("Paciente dental solicitando cita...");
    }
}

public class CitaGeneral implements Cita {
    @Override
    public void programar() {
        System.out.println("Cita general programada.");
    }
}

public class CitaDentista implements Cita {
    @Override
    public void programar() {
        System.out.println("Cita dental programada.");
    }
}
```

4. **Main**

```
public class Main {
    public static void main(String[] args) {
        // Crear una fábrica de médicos generales
        FabricaAbstracta fabricaMedicosGenerales = new
FabricaMedicosGenerales();
        Medico medicoGeneral =
fabricaMedicosGenerales.crearMedico();
        Paciente pacienteGeneral =
fabricaMedicosGenerales.crearPaciente();
        Cita citaGeneral = fabricaMedicosGenerales.crearCita();

        medicoGeneral.atenderPaciente();
        pacienteGeneral.solicitarCita();
        citaGeneral.programar();

        // Crear una fábrica de dentistas
        FabricaAbstracta fabricaDentistas = new FabricaDentistas();
        Medico dentista = fabricaDentistas.crearMedico();
        Paciente pacienteDental = fabricaDentistas.crearPaciente();
        Cita citaDentista = fabricaDentistas.crearCita();

        dentista.atenderPaciente();
        pacienteDental.solicitarCita();
        citaDentista.programar();
    }
}
```

## 8. Salida del Código

```
run:
M�dico general atendiendo paciente...
Paciente general solicitando cita...
Cita general programada.
Dentista atendiendo paciente...
Paciente dental solicitando cita...
Cita dental programada.
BUILD SUCCESSFUL (total time: 1 second)
```

## 9. Conclusiones

El patron factory en nuestro sistema nos permite cosas nuevas como crear familias de objetos relacionados de manera flexible y coherente.

Separar la creación de objetos de su uso, lo que hace que el código sea más modular y fácil de mantener.

Escalar el sistema fácilmente, ya que podemos añadir nuevas familias de objetos (como cardiólogos o pediatras) sin modificar el código existente.