

# Patrones de diseño

## Patrones estructurales:

### Factory:

Generamos un objeto base para nuestros personajes.

```
1 package factory;
2
3 // Clase base para los personajes que se crean.
4 public abstract class Character {
5     protected String name;
6     protected int health;
7     protected int strength;
8
9     public Character(String name, int health, int strength) {
10         this.name = name;
11         this.health = health;
12         this.strength = strength;
13     }
14
15     public String getName() {
16         return name;
17     }
18
19     public int getHealth() {
20         return health;
21     }
22
23     public int getStrength() {
24         return strength;
25     }
26
27     public abstract String action();
28 }
29
```

Y una clase para definir cada tipo de personaje (arqueros, magos y guerreros):

```
Source History
1 package factory;
2
3 // Clase específica que extiende de la principal de personajes para devolver los valores que ocupa para crear uno.
4 public class Warrior extends Character {
5
6     private int armor;
7
8     public Warrior(String name) {
9         super(name, health:200, strength: 50);
10        this.armor = 30;
11    }
12
13    @Override
14    public String action() {
15        return name + " se tiene " + armor + " puntos de armadura.";
16    }
17 }
```

Definimos nuestra fábrica de personajes para cada que se quiera crear uno nuevo:

```
Main.java X CharacterFactory.java X
Source History
1 package factory;
2
3 // Implementamos nuestra fabrica de personajes para crear uno según lo que elijamos.
4 public class CharacterFactory {
5
6     public Character createCharacter(String type, String name) {
7         //Se crea el personaje de la clase o type que se haya recibido.
8         switch (type.toLowerCase()) {
9             case "guerrero":
10                 return new Warrior(name);
11             case "mago":
12                 return new Mage(name);
13             case "arquero":
14                 return new Archer(name);
15             default:
16                 throw new IllegalArgumentException("Tipo de personaje no válido: " + type);
17         }
18     }
19 }
20
```

Creamos nuestro personaje:

```
package factory;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        CharacterFactory factory = new CharacterFactory();
        Scanner scanner = new Scanner(System.in);

        //Pedimos el tipo o clase del personaje que queremos crear.
        System.out.println("Ingresa el tipo de personaje (Guerrero, Mago, Arquero):");
        String type = scanner.nextLine();

        //Le asignamos un nombre que irá como parametro (1 de los 3 que ocupa un personaje).
        System.out.println("Ingresa el nombre del personaje:");
        String name = scanner.nextLine();

        try {
            // Manejamos la creación de nuestro personaje mandando el tipo/clase que queremos crear.
            Character character = factory.createCharacter(type, name);
            System.out.println("Personaje creado: " + character.getName());
            System.out.println("Acción: " + character.action());
        } catch (IllegalArgumentException e) {
            System.err.println("Error: " + e.getMessage());
        }

        scanner.close();
    }
}
```

Output - DesignModels (run) X

```
run:
Ingresa el tipo de personaje (Guerrero, Mago, Arquero):
Guerrero
Ingresa el nombre del personaje:
Aaron
Personaje creado: Aaron
Acción: Aaron se tiene 30 puntos de armadura.
BUILD SUCCESSFUL (total time: 12 seconds)
```

## Abstract Factory:

Clase base para personajes:

```
Character.java X
Source History
1 package abstractFactory;
2
3 // Clase base para los personajes que se crean.
4 public abstract class Character {
5     protected String name;
6     protected int health;
7     protected int strength;
8
9     public Character(String name, int health, int strength) {
10         this.name = name;
11         this.health = health;
12         this.strength = strength;
13     }
14
15     public String getName() {
16         return name;
17     }
18
19     public int getHealth() {
20         return health;
21     }
22
23     public int getStrength() {
24         return strength;
25     }
26
27     public abstract String action();
28 }
29
30
```

Clases para cada tipo de personaje y su fabrica:

```
Character.java X CharacterFactory.java X Armor.java X ArcherFactory.java X Archer.java X
Source History
1 package abstractFactory;
2
3 public class Archer extends Character {
4
5     private Weapon weapon;
6     private Armor armor;
7
8     public Archer(String name, Weapon weapon, Armor armor) {
9         super(name, health:160, strength: 60);
10         this.weapon = weapon;
11         this.armor = armor;
12     }
13
14     @Override
15     public String action() {
16         return name + " usa " + weapon.useWeapon() + " y su equipamiento es " + armor.useArmor();
17     }
18 }
19
```

```
Character.java X CharacterFactory.java X Armor.java X ArcherFactory.java X Archer.java X
Source History
1 package abstractFactory;
2
3 // Creador especifico para los personajes arqueros.
4 public class ArcherFactory implements CharacterFactory {
5     @Override
6     public Character createCharacter(String name) {
7         return new Archer(name, weapon:createWeapon(), armor: createArmor());
8     }
9
10     @Override
11     public Weapon createWeapon() {
12         return new Bow();
13     }
14
15     @Override
16     public Armor createArmor() {
17         return new LightArmor();
18     }
19 }
20
```

Creamos una clase para cada item que tengamos:

```
Character.java x CharacterFactory.java x Armor.java x ArcherFacto
Source History
1 package abstractFactory;
2
3 public interface Armor {
4     String useArmor();
5 }
6
7
```

Creamos personajes:

```
Source History
1 package abstractFactory;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // Creamos nuestros personajes y sus accesorios
7         CharacterFactory warriorFactory = new WarriorFactory();
8         Character warrior = warriorFactory.createCharacter(name: "Aaron");
9         System.out.println(x: warrior.action());
10
11         CharacterFactory mageFactory = new MageFactory();
12         Character mage = mageFactory.createCharacter(name: "Diego");
13         System.out.println(x: mage.action());
14
15         CharacterFactory archerFactory = new ArcherFactory();
16         Character archer = archerFactory.createCharacter(name: "Roger");
17         System.out.println(x: archer.action());
18     }
19 }
```

```
run:
Aaron usa espada afilada y ssu equipamiento es armadura pesada
Diego usa bastón mágico y ssu equipamiento es armadura ligera
Roger usa arco con flechas y su equipamiento es armadura ligera
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Singleton:

Creamos nuestra clase junto con una instancia única, para asegurarnos de siempre apuntar al mismo objeto y así mismo declaramos los métodos para obtener usuarios, agregar y/o eliminar:

```
UserManager.java X Main.java X
Source History
10 // Lista de los usuarios conectados.
11 private List<String> connectedUsers;
12
13 private UserManager() {
14     connectedUsers = new ArrayList<>();
15 }
16
17 // Método para obtener la instancia única.
18 public static UserManager getInstance() {
19     if (instance == null) {
20         synchronized (UserManager.class) {
21             if (instance == null) {
22                 instance = new UserManager();
23             }
24         }
25     }
26     return instance;
27 }
28
29 // Metodo para conectar más usuarios
30 public void addUser(String username) {
31     connectedUsers.add(e: username);
32     System.out.println(username + " se ha conectado.");
33 }
34
35 // Desconectar usuarios
36 public void removeUser(String username) {
37     connectedUsers.remove(e: username);
38     System.out.println(username + " se ha desconectado.");
39 }
40
41 // Lista de usuarios.
42 public void showConnectedUsers() {
```

```

43     if (connectedUsers.isEmpty()) {
44         System.out.println(x: "No hay usuarios conectados.");
45     } else {
46         System.out.println("Usuarios conectados: " + connectedUsers);
47     }
48 }
49 }
```

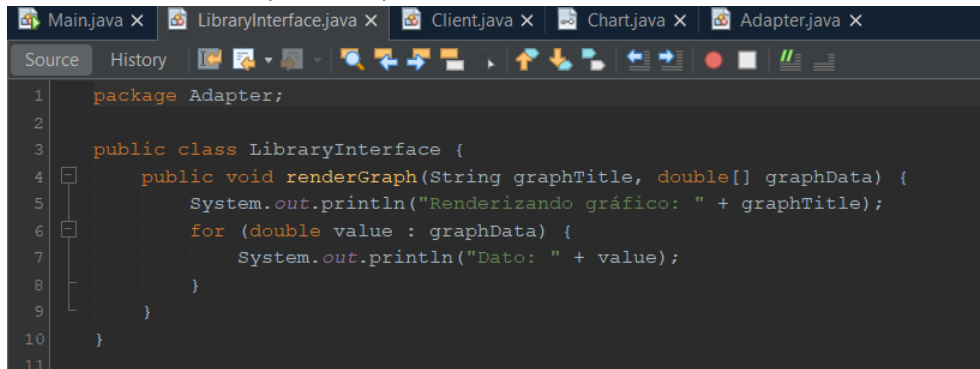
Creamos usuarios y los buscamos utilizando diferentes constructores que apuntan a la misma instancia:

```
UserManager.java x Main.java x
Source History
1 package Singleton;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // Hacemos 2 constructores de nuestra instancia unica de userManager.
7         UserManager userManager = UserManager.getInstance();
8         UserManager userManager2 = UserManager.getInstance();
9
10        // Agregamos usuarios con el primer constructor
11        userManager.addUser(username: "Aaron");
12        userManager.addUser(username: "Diego");
13
14        // Mostramos los usuarios con el otro constructor, debe haber 2.
15        userManager2.showConnectedUsers();
16
17        // Desconectamos un usuario con un constructor
18        userManager2.removeUser(username: "Diego");
19
20        // Mostramos los usuarios de nuevo con el otro constructor.
21        userManager.showConnectedUsers();
22    }
23 }
24
```

```
Output - DesignModels (run) x
run:
Aaron se ha conectado.
Diego se ha conectado.
Usuarios conectados: [Aaron, Diego]
Diego se ha desconectado.
Usuarios conectados: [Aaron]
BUILD SUCCESSFUL (total time: 0 seconds)
```

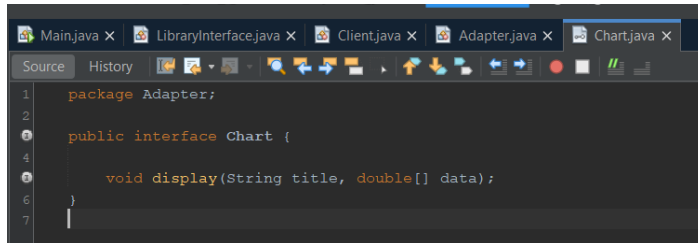
## Adapter:

Definimos la interfaz que no se puede modificar:



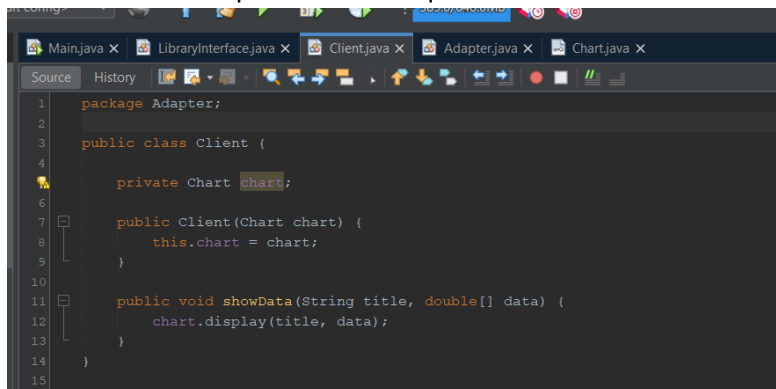
```
1 package Adapter;
2
3 public class LibraryInterface {
4     public void renderGraph(String graphTitle, double[] graphData) {
5         System.out.println("Renderizando gráfico: " + graphTitle);
6         for (double value : graphData) {
7             System.out.println("Dato: " + value);
8         }
9     }
10 }
11
```

Definimos la que vamos a usar:



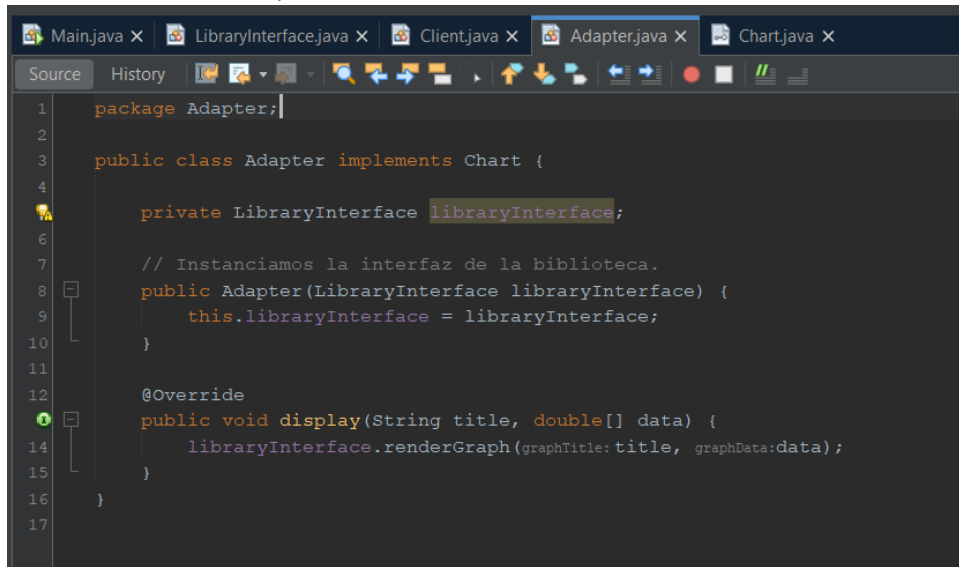
```
1 package Adapter;
2
3 public interface Chart {
4     void display(String title, double[] data);
5 }
6
7
```

Creamos una clase para el usuario que usará dicha interfaz:



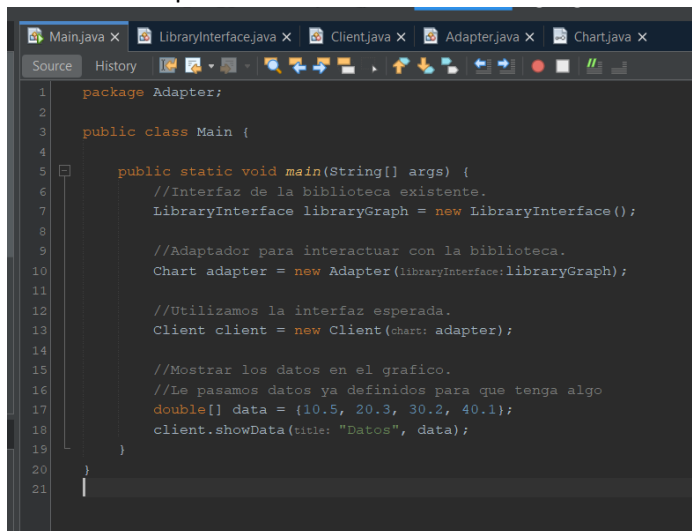
```
1 package Adapter;
2
3 public class Client {
4     private Chart chart;
5
6     public Client(Chart chart) {
7         this.chart = chart;
8     }
9
10    public void showData(String title, double[] data) {
11        chart.display(title, data);
12    }
13 }
14
15
```

Definimos nuestro adaptador:

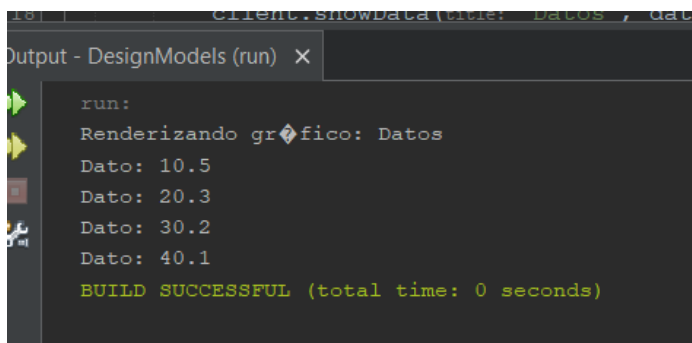


```
1 package Adapter;
2
3 public class Adapter implements Chart {
4
5     private LibraryInterface libraryInterface;
6
7     // Instanciamos la interfaz de la biblioteca.
8     public Adapter(LibraryInterface libraryInterface) {
9         this.libraryInterface = libraryInterface;
10    }
11
12    @Override
13    public void display(String title, double[] data) {
14        libraryInterface.renderGraph(graphTitle: title, graphData: data);
15    }
16 }
17
```

Realizamos el proceso:



```
1 package Adapter;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         //Interfaz de la biblioteca existente.
7         LibraryInterface libraryGraph = new LibraryInterface();
8
9         //Adaptador para interactuar con la biblioteca.
10        Chart adapter = new Adapter(libraryInterface: libraryGraph);
11
12        //Utilizamos la interfaz esperada.
13        Client client = new Client(chart: adapter);
14
15        //Mostrar los datos en el grafico.
16        //Le pasamos datos ya definidos para que tenga algo
17        double[] data = {10.5, 20.3, 30.2, 40.1};
18        client.showData(title: "Datos", data);
19    }
20 }
21
```

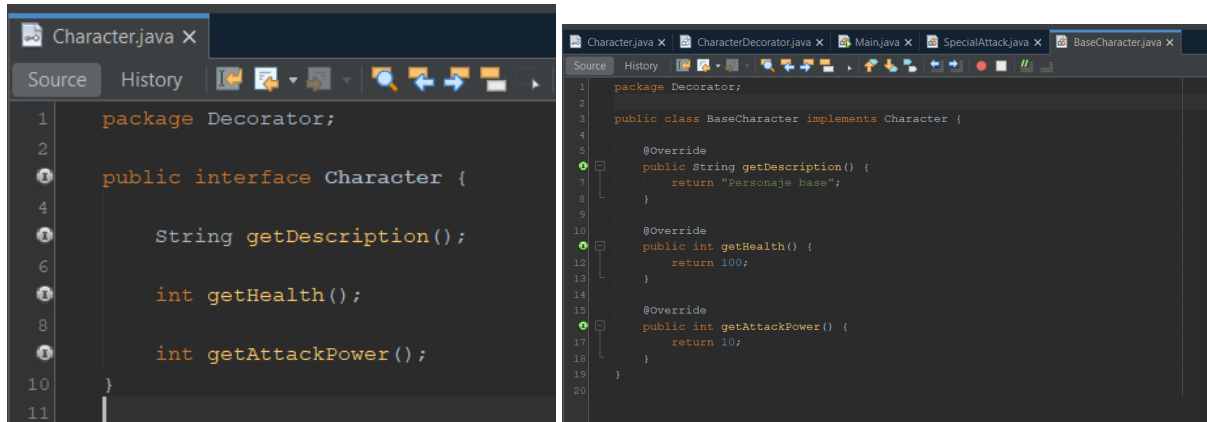


```
Output - DesignModels (run) X
run:
Renderizando gráfico: Datos
Dato: 10.5
Dato: 20.3
Dato: 30.2
Dato: 40.1
BUILD SUCCESSFUL (total time: 0 seconds)
```



## Decorator:

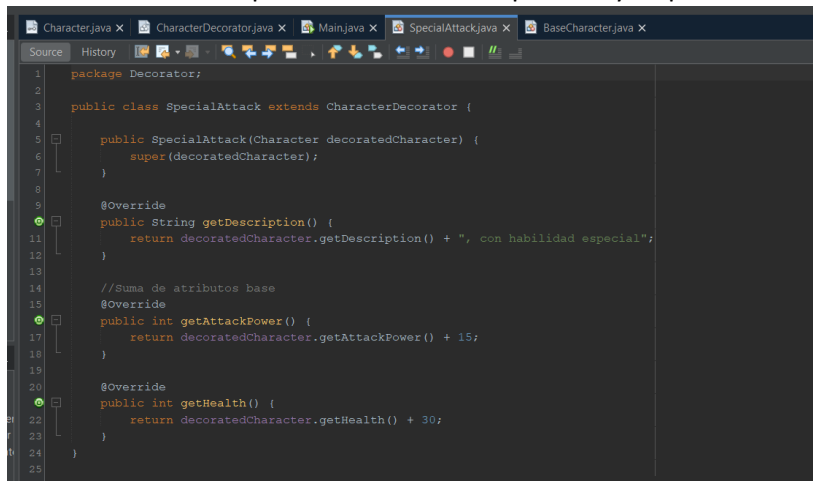
Creamos una clase base para los personajes y una clase a para sus atributos base:



```
1 package Decorator;
2
3 public interface Character {
4
5     String getDescription();
6
7     int getHealth();
8
9     int getAttackPower();
10 }
11
```

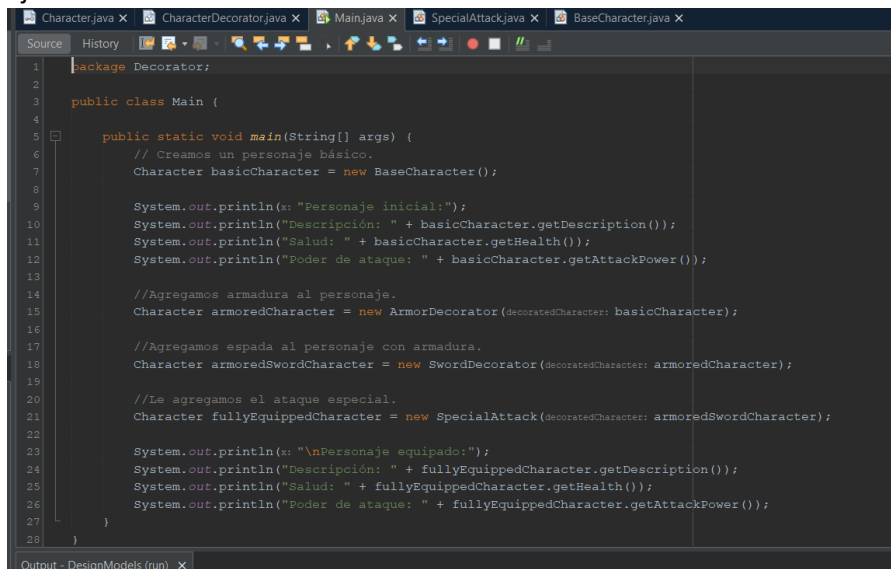
```
1 package Decorator;
2
3 public class BaseCharacter implements Character {
4
5     @Override
6     public String getDescription() {
7         return "Personaje base";
8     }
9
10    @Override
11    public int getHealth() {
12        return 100;
13    }
14
15    @Override
16    public int getAttackPower() {
17        return 10;
18    }
19 }
20
```

Definimos una clase para cada decorador que se vaya a poder añadir a nuestro personaje:



```
1 package Decorator;
2
3 public class SpecialAttack extends CharacterDecorator {
4
5     public SpecialAttack(Character decoratedCharacter) {
6         super(decoratedCharacter);
7     }
8
9     @Override
10    public String getDescription() {
11        return decoratedCharacter.getDescription() + ", con habilidad especial";
12    }
13
14    //Suma de atributos base
15    @Override
16    public int getAttackPower() {
17        return decoratedCharacter.getAttackPower() + 15;
18    }
19
20    @Override
21    public int getHealth() {
22        return decoratedCharacter.getHealth() + 30;
23    }
24 }
25
```

## Ejecución:



```
1 package Decorator;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // Creamos un personaje básico.
7         Character basicCharacter = new BaseCharacter();
8
9         System.out.println("Personaje inicial:");
10        System.out.println("Descripción: " + basicCharacter.getDescription());
11        System.out.println("Salud: " + basicCharacter.getHealth());
12        System.out.println("Poder de ataque: " + basicCharacter.getAttackPower());
13
14        //Agregamos armadura al personaje.
15        Character armoredCharacter = new ArmorDecorator(decoratedCharacter: basicCharacter);
16
17        //Agregamos espada al personaje con armadura.
18        Character armoredSwordCharacter = new SwordDecorator(decoratedCharacter: armoredCharacter);
19
20        //Le agregamos el ataque especial.
21        Character fullyEquippedCharacter = new SpecialAttack(decoratedCharacter: armoredSwordCharacter);
22
23        System.out.println("\nPersonaje equipado:");
24        System.out.println("Descripción: " + fullyEquippedCharacter.getDescription());
25        System.out.println("Salud: " + fullyEquippedCharacter.getHealth());
26        System.out.println("Poder de ataque: " + fullyEquippedCharacter.getAttackPower());
27    }
28 }

```

Output - DesignModels (run) X

Output - DesignModels (run) X



```
run:
Personaje inicial:
Descripción: Personaje base
Salud: 100
Poder de ataque: 10

Personaje equipado:
Descripción: Personaje base, con armadura, con espada, con habilidad especial
Salud: 180
Poder de ataque: 45
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Patrones de comportamiento:

### Observer:

Definimos a nuestros observadores (en mi caso usuarios a notificar) y la clase que los gestionará:

```
User.java X
Source History
1 package Observer;
2
3 public class User implements Observer {
4
5     private String name;
6
7     public User(String name) {
8         this.name = name;
9     }
10
11     @Override
12     public void update(String message) {
13         System.out.println(name + " ha recibido la notificación: " + message);
14     }
15 }
16
```

```
User.java X Subject.java X NotificationService.java X Observer.java X Main.java X
Source History
4 import java.util.List;
5
6 public class NotificationService implements Subject {
7
8     //Lista de usuarios a notificar
9     private List<Observer> observers;
10
11     private String notification;
12
13     public NotificationService() {
14         this.observers = new ArrayList<>();
15     }
16
17     @Override
18     public void registerObserver(Observer observer) {
19         observers.add(observer);
20     }
21
22     @Override
23     public void notifyObservers() {
24         for (Observer observer : observers) {
25             observer.update(notification);
26         }
27     }
28
29     // Método para actualizar el estado y notificar a los observadores
30     public void publishNotification(String message) {
31         this.notification = message;
32         System.out.println("Publicando notificación: " + message);
33         notifyObservers();
34     }
35 }
36
Output - DesignModels (run) X
```

```
User.java X Subject.java X NotificationService.java X Observer.java X Main.java X
Source History
1 package Observer;
2
3 //Esta clase gestiona a los observadores que tengamos
4 public interface Subject {
5
6     void registerObserver(Observer observer);
7
8     void notifyObservers();
9 }
10
```

## Ejecución:

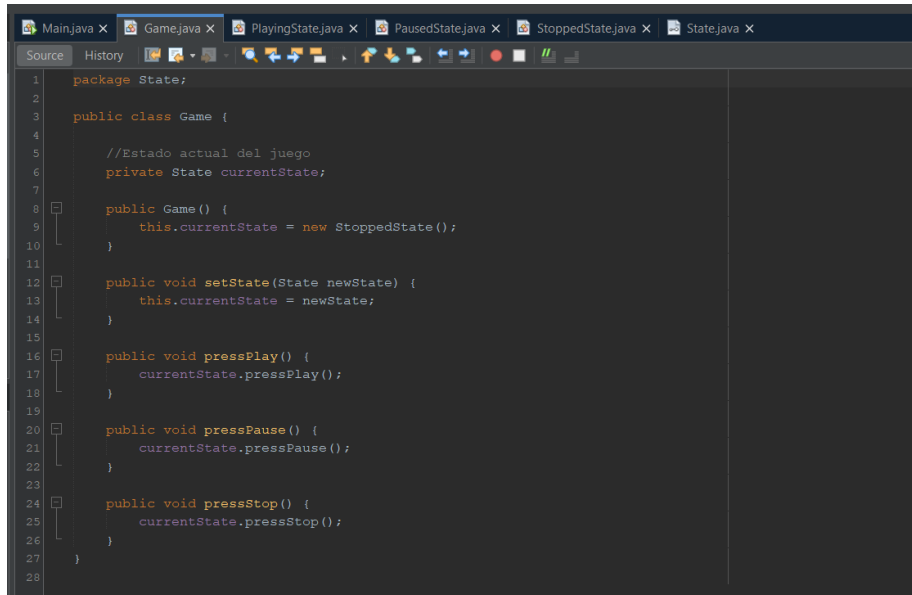
```
User.java X Subject.java X NotificationService.java X Observer.java X Main.java X
Source History
1 package Observer;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // Constructor de nuestro servicio de notificaciones
7         NotificationService notificationService = new NotificationService();
8
9         // Creamos nuestros usuarios a notificar
10        User user1 = new User(name: "Aaron");
11        User user2 = new User(name: "Diego");
12
13        // Registramos los usuarios
14        notificationService.registerObserver(observer: user1);
15        notificationService.registerObserver(observer: user2);
16
17        // Publicamos una notificación
18        notificationService.publishNotification(message: "Nuevo evento disponible.");
19
20        // Publicar otra notificación
21        notificationService.publishNotification(message: "Actualización del sistema programada para mañana.");
22    }
23 }
24
```

## Output - DesignModels (run) X

```
run:
Publicando notificación: Nuevo evento disponible.
Aaron ha recibido la notificación: Nuevo evento disponible.
Diego ha recibido la notificación: Nuevo evento disponible.
Publicando notificación: Actualización del sistema programada para mañana.
Aaron ha recibido la notificación: Actualización del sistema programada para mañana.
Diego ha recibido la notificación: Actualización del sistema programada para mañana.
BUILD SUCCESSFUL (total time: 0 seconds)
```

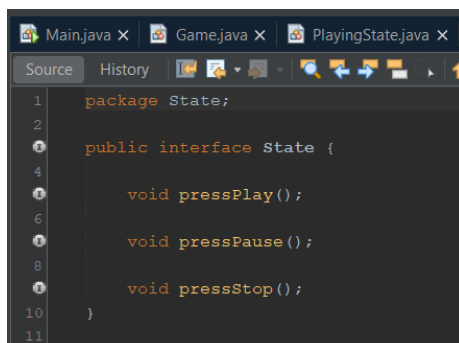
## State:

Definimos Nuestro juego y sus posibles estados:



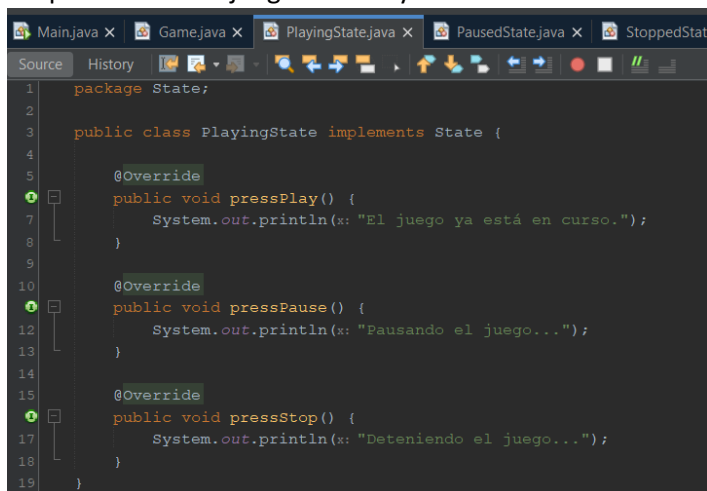
```
1 package State;
2
3 public class Game {
4
5     //Estado actual del juego
6     private State currentState;
7
8     public Game() {
9         this.currentState = new StoppedState();
10    }
11
12    public void setState(State newState) {
13        this.currentState = newState;
14    }
15
16    public void pressPlay() {
17        currentState.pressPlay();
18    }
19
20    public void pressPause() {
21        currentState.pressPause();
22    }
23
24    public void pressStop() {
25        currentState.pressStop();
26    }
27 }
28
```

Nuestra clase de estado base:



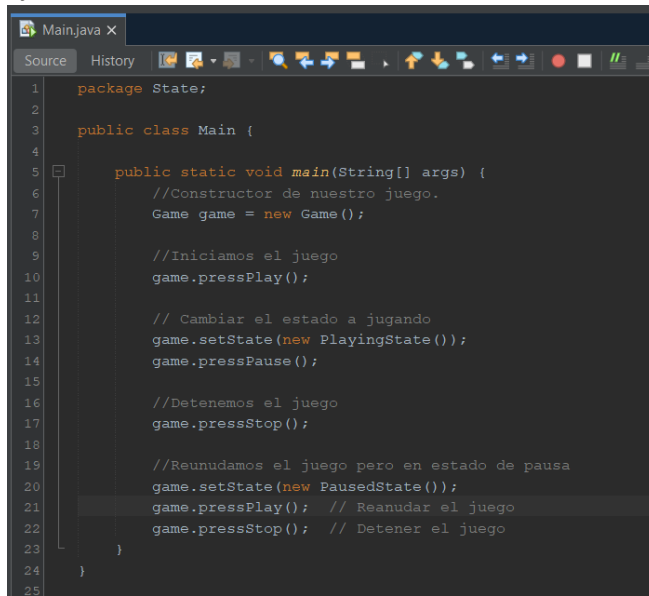
```
1 package State;
2
3 public interface State {
4
5     void pressPlay();
6
7     void pressPause();
8
9     void pressStop();
10 }
11
```

Para cada estado definimos sus posibilidades de que ya se encuentre en otro (esto para los casos donde se quiere iniciar el juego cuando ya está en estado iniciado o jugando por ejemplo):

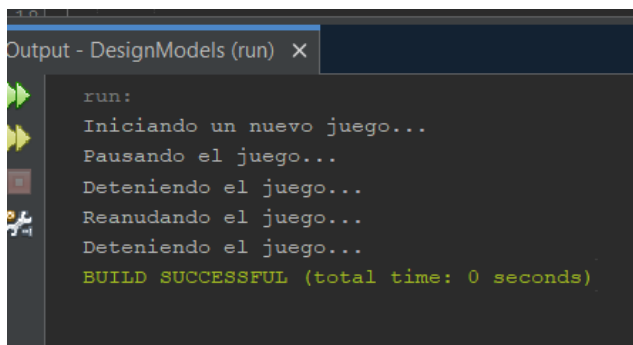


```
1 package State;
2
3 public class PlayingState implements State {
4
5     @Override
6     public void pressPlay() {
7         System.out.println(x: "El juego ya está en curso.");
8     }
9
10    @Override
11    public void pressPause() {
12        System.out.println(x: "Pausando el juego...");
13    }
14
15    @Override
16    public void pressStop() {
17        System.out.println(x: "Deteniendo el juego...");
18    }
19 }
```

## Ejecución:



```
1 package State;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         //Constructor de nuestro juego.
7         Game game = new Game();
8
9         //Iniciamos el juego
10        game.pressPlay();
11
12        // Cambiar el estado a jugando
13        game.setState(new PlayingState());
14        game.pressPause();
15
16        //Detenemos el juego
17        game.pressStop();
18
19        //Reanudamos el juego pero en estado de pausa
20        game.setState(new PausedState());
21        game.pressPlay(); // Reanudar el juego
22        game.pressStop(); // Detener el juego
23    }
24 }
25
```



```
Output - DesignModels (run) X
run:
Iniciando un nuevo juego...
Pausando el juego...
Deteniendo el juego...
Reanudando el juego...
Deteniendo el juego...
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Strategy:

Definimos un personaje base:

```
Main.java X MagicalAttack.java X Character.java X AttackStrategy.java X PhysicalAttack.java X
Source History
1 package Strategy;
2
3 public class Character {
4
5     private AttackStrategy attackStrategy; // Referencia a la estrategia de ataque
6
7     // Constructor de personajes para asignar una estrategia de ataque inicial
8     public Character(AttackStrategy attackStrategy) {
9         this.attackStrategy = attackStrategy;
10    }
11
12    // Metodo para cambiar la estrategia de ataque.
13    public void setAttackStrategy(AttackStrategy attackStrategy) {
14        this.attackStrategy = attackStrategy;
15    }
16
17    // Metodo para atacar con la estrategia usada
18    public void performAttack() {
19        attackStrategy.attack();
20    }
21 }
22
```

Definimos cada estrategia de cada tipo de ataque:

```
Main.java X MagicalAttack.java X Character.java X AttackStrategy.java X PhysicalAttack.java X
Source History
1 package Strategy;
2
3 public class PhysicalAttack implements AttackStrategy {
4
5     @Override
6     public void attack() {
7         System.out.println("Realizando un ataque fisico!");
8     }
9
10 }
```

Ejecución:

Nuestro personaje cambia de estrategia en medio de la ejecución

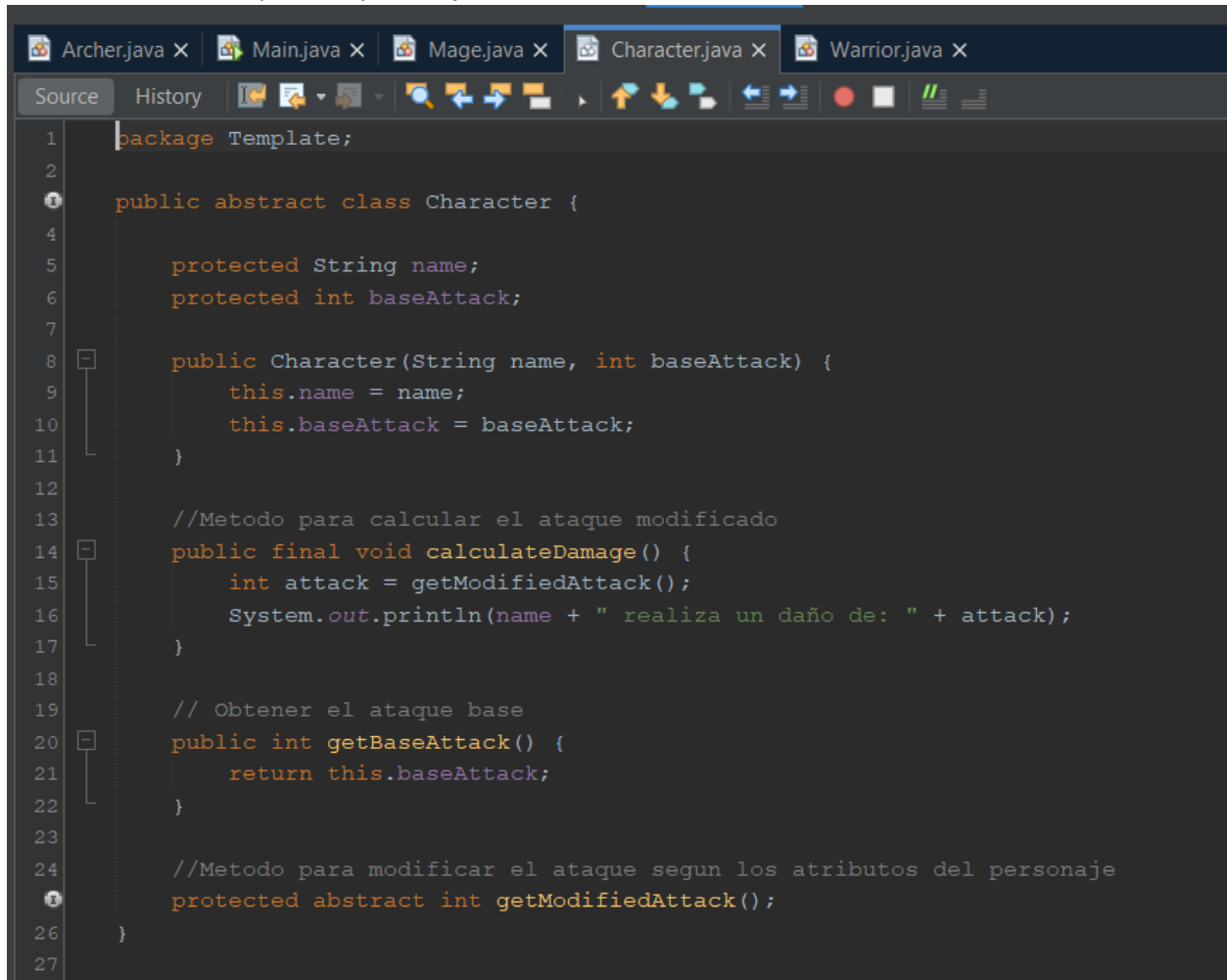
```
Main.java X
Source History
1 package Strategy;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // Creamos un personaje y le asignamos una estrategia de ataque inicial
7         Character warrior = new Character(new PhysicalAttack());
8
9         // El guerrero realiza el ataque fisico
10        warrior.performAttack();
11
12        // Cambiamos la estrategia de ataque porque se le rompió el arma
13        warrior.setAttackStrategy(new MagicalAttack());
14
15        // Atacamos
16        warrior.performAttack();
17    }
18 }
19
```

```
Output - DesignModels (run) X
run:
Realizando un ataque fisico!
Realizando un ataque magico!
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Patrones creacionales:

### Template:

Creamos nuestro template de personajes:



```
1 package Template;
2
3 public abstract class Character {
4
5     protected String name;
6     protected int baseAttack;
7
8     public Character(String name, int baseAttack) {
9         this.name = name;
10        this.baseAttack = baseAttack;
11    }
12
13    //Metodo para calcular el ataque modificado
14    public final void calculateDamage() {
15        int attack = getModifiedAttack();
16        System.out.println(name + " realiza un daño de: " + attack);
17    }
18
19    // Obtener el ataque base
20    public int getBaseAttack() {
21        return this.baseAttack;
22    }
23
24    //Metodo para modificar el ataque segun los atributos del personaje
25    protected abstract int getModifiedAttack();
26 }
27
```



Creamos las clases de cada tipo de personaje junto con su método de calcular su ataque:

```
Archer.java X Main.java X Mage.java X Character.java X
Source History
1 package Template;
2
3 public class Archer extends Character {
4
5     private int dexterity;
6
7     public Archer(String name, int baseAttack, int dexterity) {
8         super(name, baseAttack);
9         this.dexterity = dexterity;
10    }
11
12    //Calculo de su daño dependiendo de su destreza
13    @Override
14    protected int getModifiedAttack() {
15        return getBaseAttack() + (dexterity / 2);
16    }
17 }
```

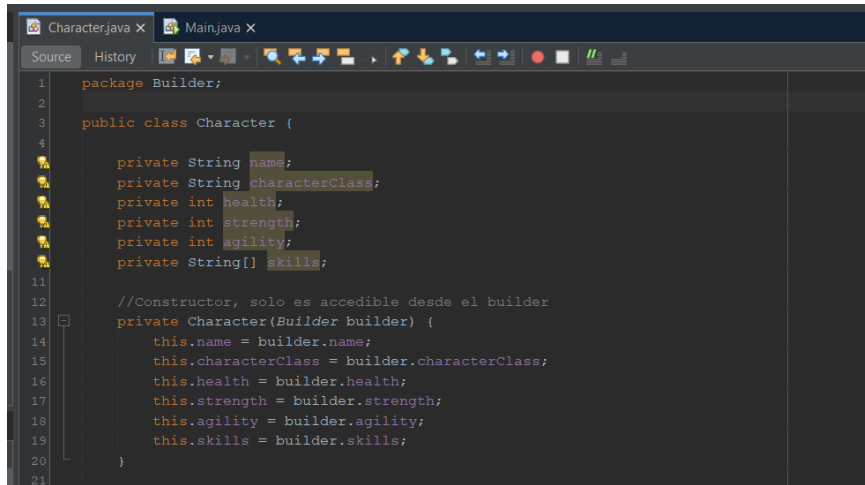
Ejecución:

```
Main.java X
Source History
1 package Template;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         //Creamos personajes con diferentes atributos
7         Character warrior = new Warrior(name: "Guerrero", baseAttack: 50, strength: 30);
8         Character mage = new Mage(name: "Mago", baseAttack: 40, intelligence: 25);
9         Character archer = new Archer(name: "Arquero", baseAttack: 45, dexterity: 20);
10
11         //Calculamos el daño para cada uno y lo imprimimos
12         warrior.calculateDamage();
13         mage.calculateDamage();
14         archer.calculateDamage();
15     }
16 }
17 }
```

```
Output - DesignModels (run) X
run:
Guerrero realiza un daño de: 80
Mago realiza un daño de: 90
Arquero realiza un daño de: 55
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Builder:

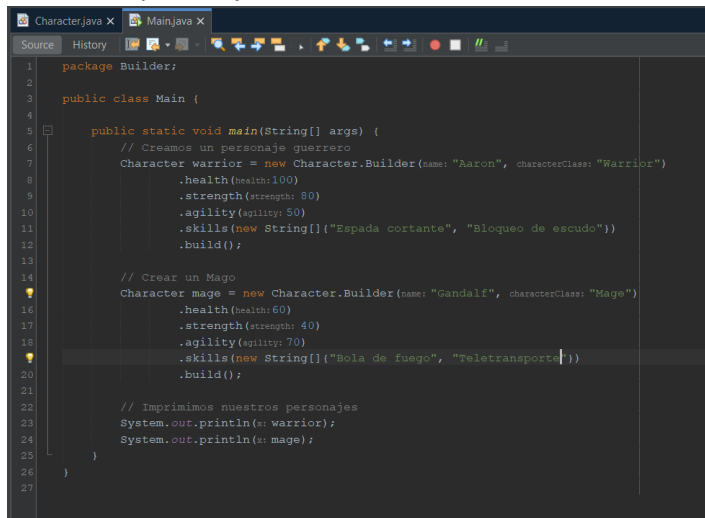
Creamos nuestra clase de personaje, además de sus getters para sus atributos y setters para asignar valores al crear uno nuevo:



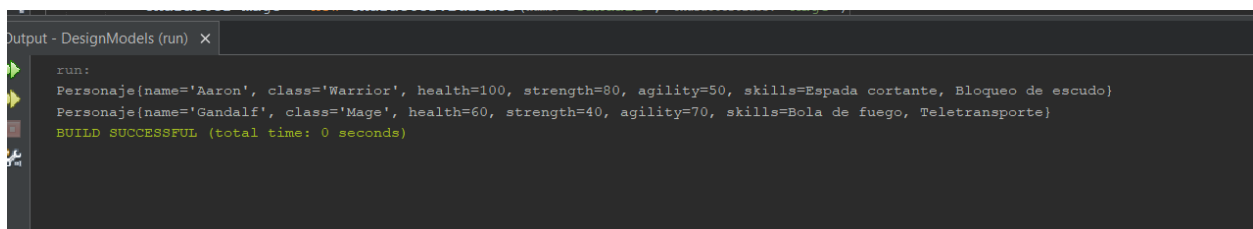
```
1 package Builder;
2
3 public class Character {
4
5     private String name;
6     private String characterClass;
7     private int health;
8     private int strength;
9     private int agility;
10    private String[] skills;
11
12    //Constructor, solo es accesible desde el builder
13    private Character(Builder builder) {
14        this.name = builder.name;
15        this.characterClass = builder.characterClass;
16        this.health = builder.health;
17        this.strength = builder.strength;
18        this.agility = builder.agility;
19        this.skills = builder.skills;
20    }
21 }
```

## Ejecución:

Creamos 2 personajes:



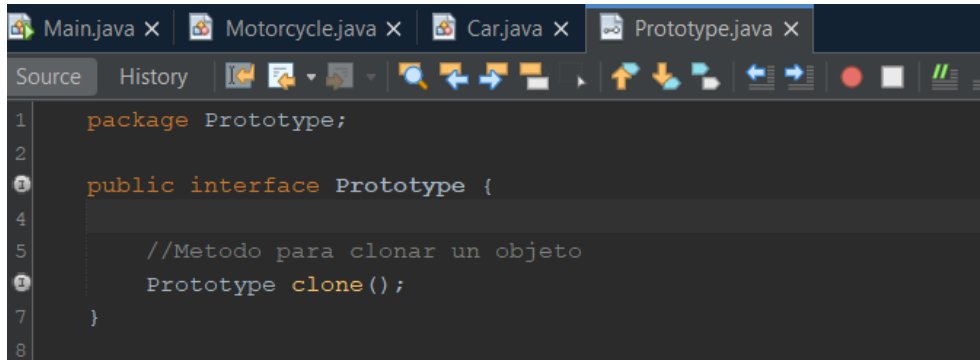
```
1 package Builder;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         // Creamos un personaje guerrero
7         Character warrior = new Character.Builder(name: "Aaron", characterClass: "Warrior")
8             .health(health: 100)
9             .strength(strength: 80)
10            .agility(agility: 50)
11            .skills(new String[]{"Espada cortante", "Bloqueo de escudo"})
12            .build();
13
14        // Crear un Mago
15        Character mage = new Character.Builder(name: "Gandalf", characterClass: "Mage")
16            .health(health: 60)
17            .strength(strength: 40)
18            .agility(agility: 70)
19            .skills(new String[]{"Bola de fuego", "Teletransporte"})
20            .build();
21
22        // Imprimimos nuestros personajes
23        System.out.println(s: warrior);
24        System.out.println(s: mage);
25    }
26 }
27 }
```



```
run:
Personaje{name='Aaron', class='Warrior', health=100, strength=80, agility=50, skills=Espada cortante, Bloqueo de escudo}
Personaje{name='Gandalf', class='Mage', health=60, strength=40, agility=70, skills=Bola de fuego, Teletransporte}
BUILD SUCCESSFUL (total time: 0 seconds)
```

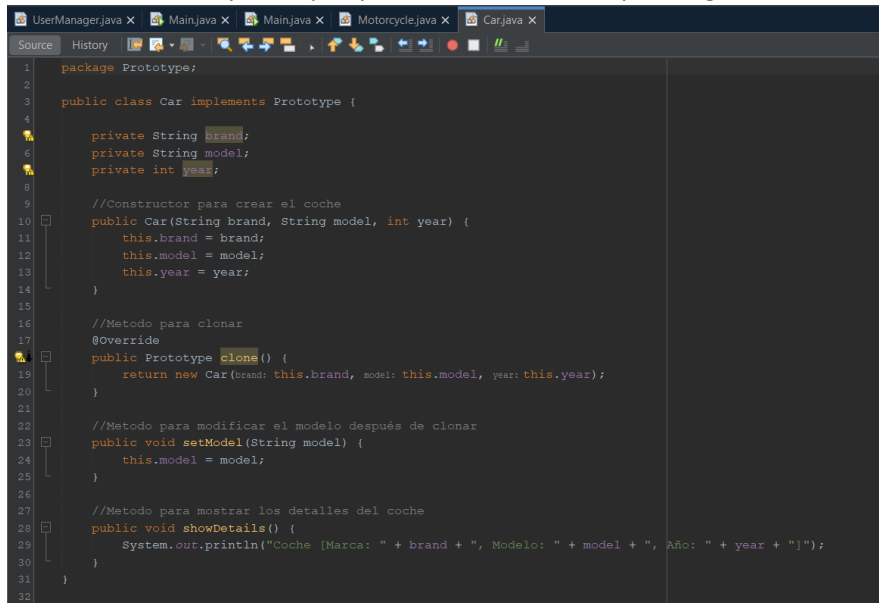
## Prototype:

Definimos nuestra clase protipo:



```
1 package Prototype;
2
3 public interface Prototype {
4
5     //Metodo para clonar un objeto
6     Prototype clone();
7 }
8
```

Creamos nuestros prototipos para los automóviles que tengamos.



```
1 package Prototype;
2
3 public class Car implements Prototype {
4
5     private String brand;
6     private String model;
7     private int year;
8
9     //Constructor para crear el coche
10    public Car(String brand, String model, int year) {
11        this.brand = brand;
12        this.model = model;
13        this.year = year;
14    }
15
16    //Metodo para clonar
17    @Override
18    public Prototype clone() {
19        return new Car(brand: this.brand, model: this.model, year: this.year);
20    }
21
22    //Metodo para modificar el modelo después de clonar
23    public void setModel(String model) {
24        this.model = model;
25    }
26
27    //Metodo para mostrar los detalles del coche
28    public void showDetails() {
29        System.out.println("Coche [Marca: " + brand + ", Modelo: " + model + ", Año: " + year + "]");
30    }
31 }
32
```

Hacemos pruebas de creación, clonado y modificación de prototipos:

```
1 package Prototype;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         //Creamos un coche e imprimimos sus detalles
7         Car car1 = new Car(brand: "Toyota", model: "Corolla", year: 2020);
8         car1.showDetails();
9
10        //Clonamos el coche que acabamos de crear
11        Car car2 = (Car) car1.clone();
12        car2.showDetails();
13
14        //Modificamos el modelo del coche clonado
15        // Cambiar el modelo en el coche clonado
16        car2.setModel(model: "Camry");
17        System.out.println("Después de cambiar el modelo del coche clonado:");
18        car2.showDetails();
19
20        //Creamos una moto
21        Motorcycle moto1 = new Motorcycle(brand: "Yamaha", type: "Sport", engineCapacity: 600);
22        moto1.showDetails();
23
24        //Clonamos la moto
25        Motorcycle moto2 = (Motorcycle) moto1.clone();
26        moto2.showDetails();
27
28        //Modificamos el tipo de la moto clonada
29        // Cambiar el tipo de moto en el objeto clonado
30        moto2.setType(type: "Cruiser");
31        System.out.println("Después de cambiar el tipo de la moto clonada:");
32        moto2.showDetails();
33    }
34 }
```

Salida:

```
Output - DesignModels (run) X
run:
Coche [Marca: Toyota, Modelo: Corolla, Año: 2020]
Coche [Marca: Toyota, Modelo: Corolla, Año: 2020]
Después de cambiar el modelo del coche clonado:
Coche [Marca: Toyota, Modelo: Camry, Año: 2020]
Moto [Marca: Yamaha, Tipo: Sport, Capacidad: 600cc]
Moto [Marca: Yamaha, Tipo: Sport, Capacidad: 600cc]
Después de cambiar el tipo de la moto clonada:
Moto [Marca: Yamaha, Tipo: Cruiser, Capacidad: 600cc]
BUILD SUCCESSFUL (total time: 0 seconds)
```