UNIVERSIDAD DE LAS FUERZAS ARMADAS (ESPE)

SEGUNDO SEMESTRE

CARRERAS TECNICAS

DEPARTAMENTO DE CIENCIAS EXACTAS

SEGUNDO PARCIAL

“MODELO VISTA CONTROLADOR (MVC) Y PATRONES DE DISEÑO ”

AUTORES:

- Guerrero Steven.
- Sislema Gabriel.
- Shaffik Samir.

NRC:

1322

DOCENTE

Luis Enrique Jaramillo Monaño

Introducción

En el desarrollo de software, una buena estructuración del código es esencial para garantizar que las aplicaciones sean escalables, mantenibles y eficientes. Para lograr este objetivo, se han diseñado diversos patrones de arquitectura que permiten organizar mejor el código y distribuir adecuadamente las responsabilidades dentro del sistema.

Uno de los enfoques más utilizados es el Modelo Vista Controlador (MVC), el cual segmenta una aplicación en tres partes fundamentales: el modelo, encargado de la gestión de datos y la lógica de negocio; la vista, que representa la interfaz con la que interactúa el usuario; y el controlador, que actúa como intermediario entre ambos, gestionando la comunicación y la interacción. Gracias a esta división, se facilita la reutilización del código y se mejora su mantenimiento.

Además del MVC, existen otros patrones de diseño que optimizan la arquitectura del software, como el patrón Singleton, el Observador y el Fábrica. Estas estrategias ofrecen soluciones efectivas a problemas recurrentes en la programación.

Este documento analizará en profundidad el patrón MVC y otros patrones de diseño relevantes, resaltando sus beneficios y presentando ejemplos prácticos que ilustran su implementación en distintos contextos de desarrollo.

1. OBJETIVOS

1.1 OBJETIVO GENERAL

Analizar el patrón de diseño Modelo Vista Controlador (MVC) y otros patrones de diseño relevantes, con el fin de comprender su importancia en la organización del código, mejorar la estructuración del software y optimizar su mantenimiento y escalabilidad a través de ejemplos prácticos de implementación.

1.2 OBJETIVOS ESPECÍFICO

Evaluar la aplicabilidad de los patrones de diseño en distintos escenarios de desarrollo de software, destacando su impacto en la escalabilidad y reutilización del código.

Implementar ejemplos prácticos que demuestren el uso de MVC y otros patrones de diseño, facilitando su comprensión y aplicación en proyectos reales.

2. MARCO TEÓRICO

- **UML**

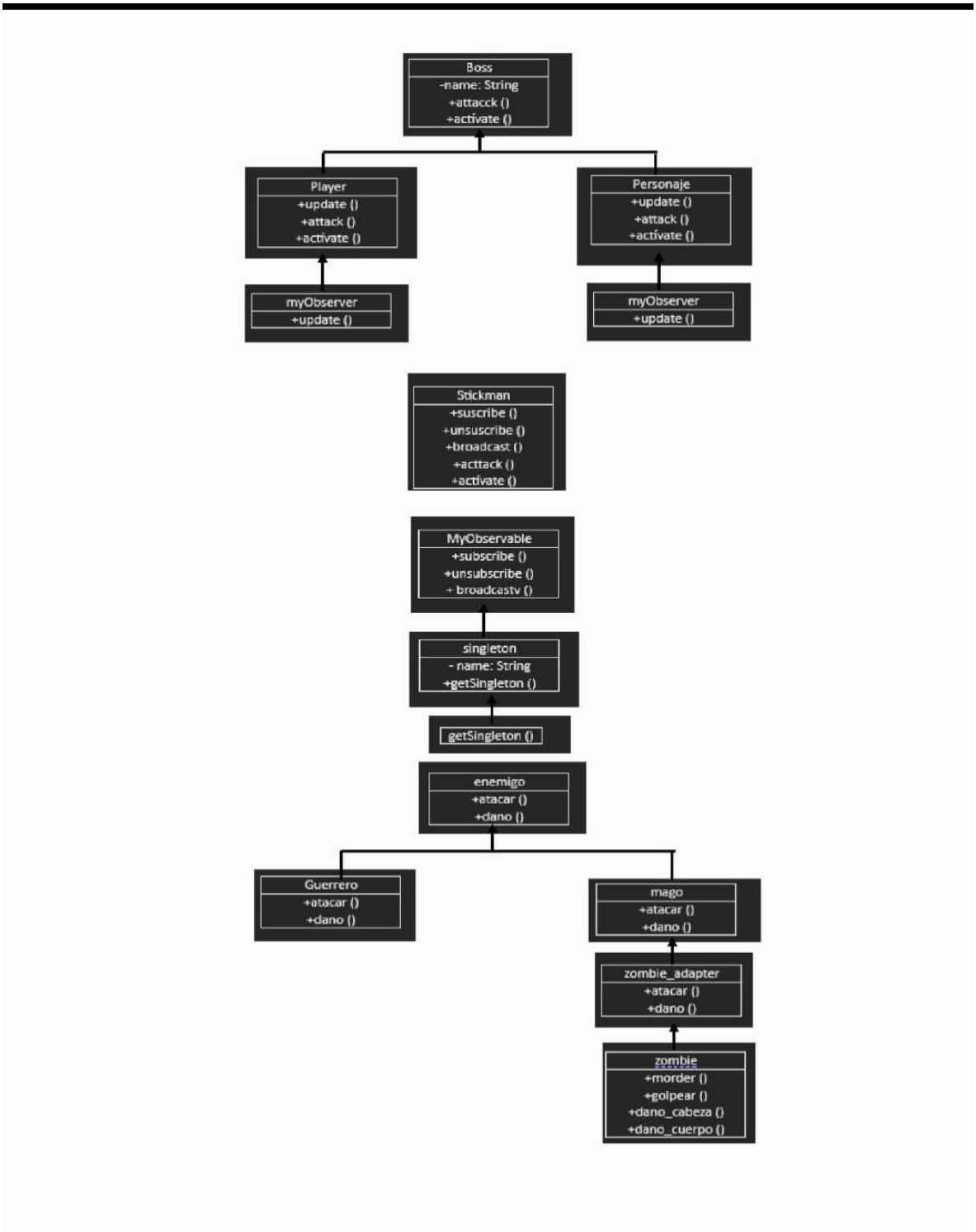


Ilustración 1

Clase Principal - Boss

- Tiene un atributo name: String.
- Métodos: attack(), activate().
- Es la clase base de la jerarquía.

Subclases de Boss

- **Player y Personaje:**
    - Ambos tienen los métodos update(), attack(), y activate().
    - Cada uno tiene una asociación con myObserver, que implementa update().
    - Sugiere que siguen el patrón Observer, donde Player y Personaje son observadores.

Clase Stickman

- Métodos clave: subscribe(), unsubscribe(), broadcast(), attack(), activate().
- Indica que esta clase podría ser un sujeto observable que gestiona suscriptores.

Clase MyObservable

- Tiene subscribe(), unsubscribe(), y broadcast().
- Refuerza el uso del patrón Observer.

Clase singleton

- **Atributo name:** String.
- Método estático getSingleton().
- Implementa el patrón Singleton, asegurando una única instancia.

Clase enemigo

- Métodos: atacar(), daño().
- Se divide en subclases:
    - Guerrero y Mago: implementan atacar() y daño().
    - zombie_adapter: adapta la clase zombie, sugiriendo el patrón Adapter.

Clase zombie_adapter y zombie

- zombie_adapter permite que zombie, con métodos morder(), golpear(), daño_cabeza(), y daño_cuerpo(), sea compatible con enemigo.
- Implementa el patrón Adapter, facilitando la conversión de métodos incompatibles.

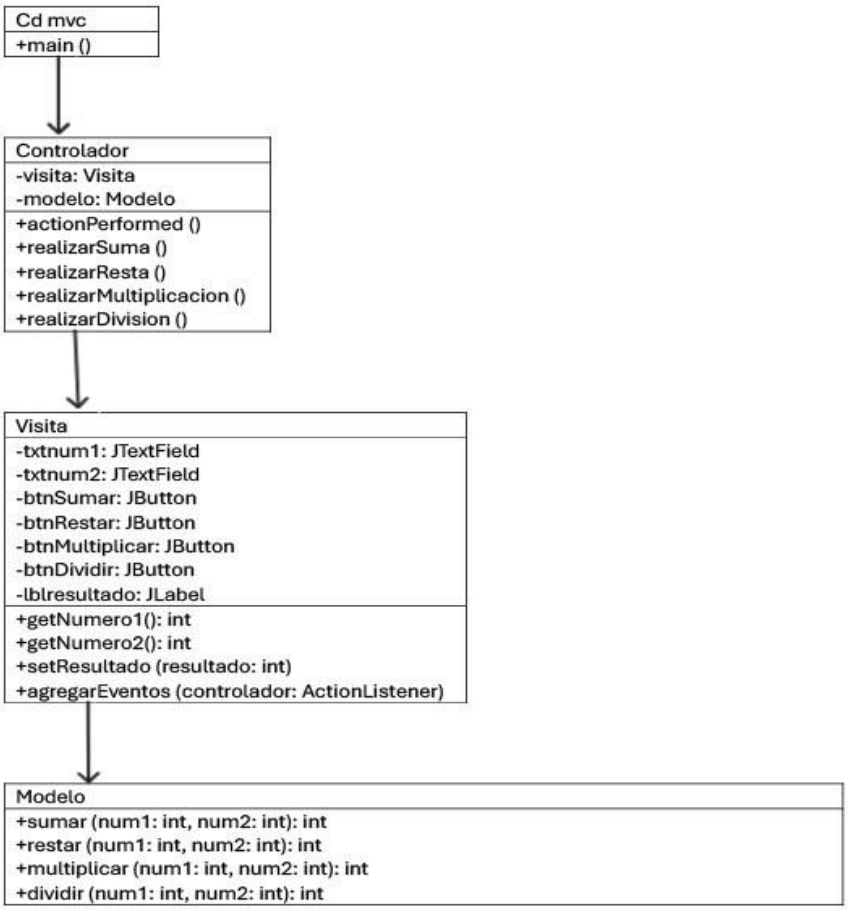


Ilustración 2

Cd MVC (Clase Principal)

- Contiene el método main(), que inicia la aplicación.
- Probablemente crea instancias de Controlador, Vista y Modelo, y las vincula.

Controlador

- Atributos:
    - **visita:** Visita (Referencia a la vista).
    - **modelo:** Modelo (Referencia al modelo).
- Métodos:
    - **actionPerformed():** Maneja eventos de la interfaz gráfica.
    - Métodos para operaciones matemáticas:
        - realizarSuma()
        - realizarResta()
        - realizarMultiplicacion()
        - realizarDivision()
- Se encarga de gestionar la lógica de negocio y actualizar la vista con los resultados.

Visita (Vista - Interfaz Gráfica)

- Atributos (Componentes de la UI):
    - **txtnum1, txtnum2:** Campos de texto (JTextField) para ingresar números.
    - **Botones (JButton):** btnSumar, btnRestar, btnMultiplicar, btnDividir.
    - **lblresultado:** Etiqueta (JLabel) para mostrar el resultado.
- Métodos:
    - **getNumero1(), getNumero2():** Obtienen los valores ingresados.
    - **setResultado(resultado: int):** Muestra el resultado en la etiqueta.
    - **agregarEventos(controlador: ActionListener):** Conecta los botones con el controlador.

Modelo (Lógica de Negocio)

- Métodos matemáticos:
    - sumar(num1, num2), restar(num1, num2), multiplicar(num1, num2), dividir(num1, num2).
- Contiene la lógica de cálculo independiente de la interfaz.
    - **MVC**

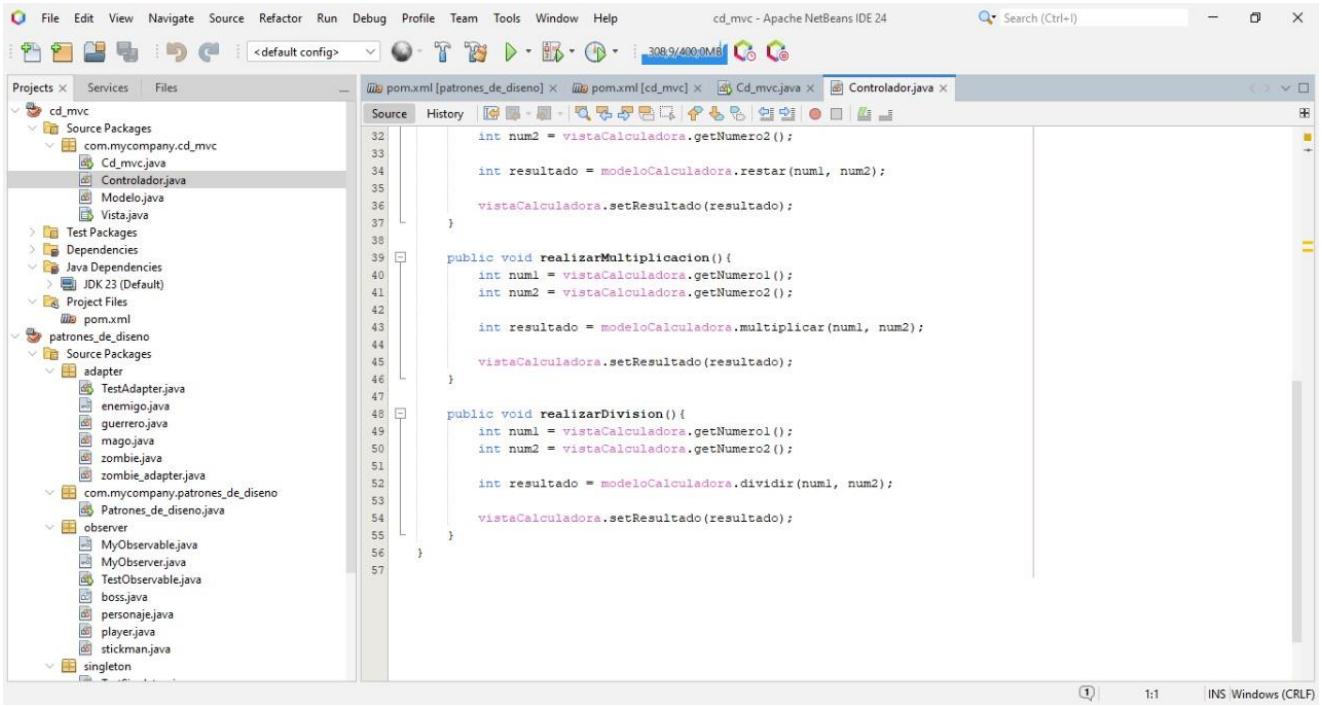


Ilustración 3

Variables de Instancia

- **vistaCalculadora**:Referencia a la vista, que permite obtener los valores ingresados y mostrar resultados.
- **modeloCalculadora**:Referencia al modelo, que contiene la lógica de las operaciones matemáticas.

Variables Locales :dentro de los Métodos

- **num1**:Almacena el primer número obtenido desde la vista.
- **num2**:Almacena el segundo número obtenido desde la vista.

- **Resultado:** Guarda el resultado de la operación matemática realizada (suma, resta, multiplicación o división).

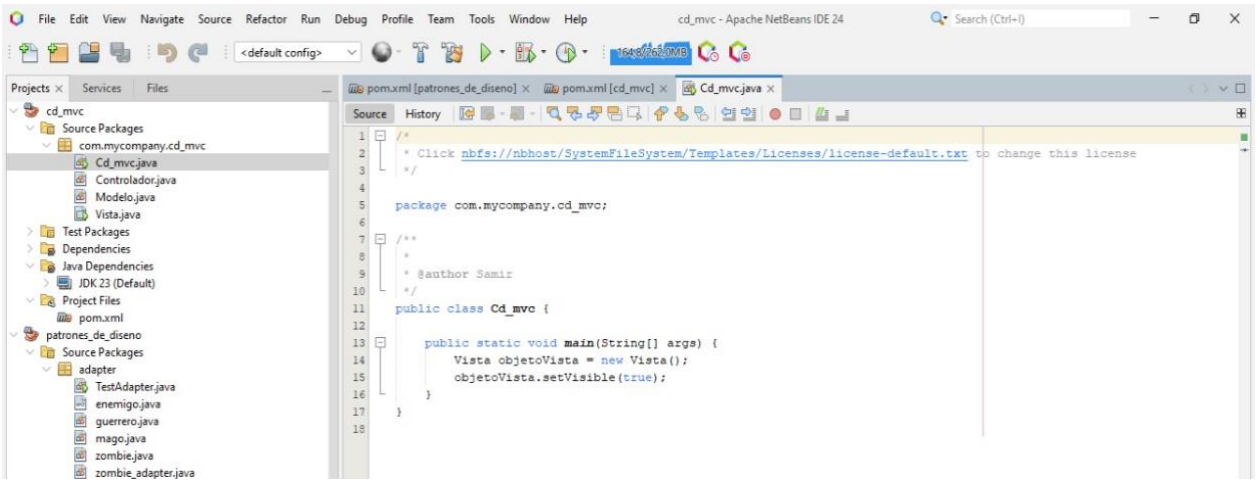


Ilustración 4

Variables Locales dentro del Método main

- **args**: Es un arreglo de tipo String que recibe los argumentos de la línea de comandos al ejecutar el programa.
- **objetoVista**: Es una variable local que almacena una instancia de la clase Vista. Se usa para inicializar la interfaz gráfica y hacerla visible con setVisible(true).

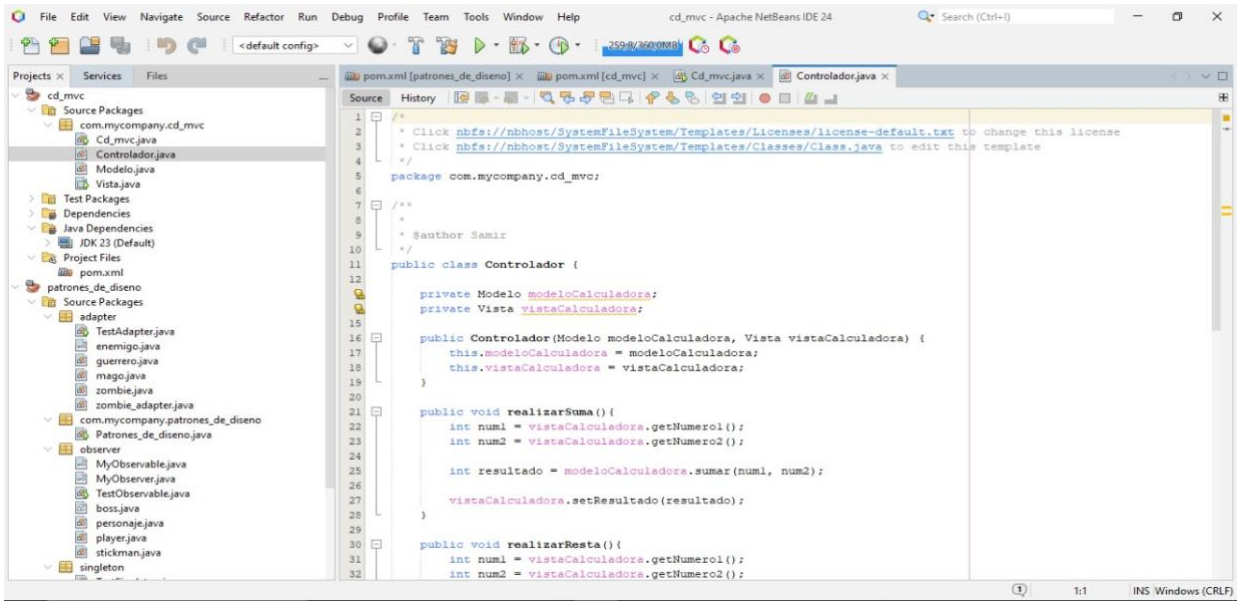


Ilustración 5

Variables de Instancia (Atributos de la Clase)

- **modeloCalculadora** : Variable privada que almacena una referencia a la clase Modelo, la cual contiene la lógica de las operaciones matemáticas.
- **vistaCalculadora** : Variable privada que almacena una referencia a la clase Vista, la cual se encarga de la interfaz gráfica del usuario.

Variables Locales dentro de los Métodos

- **num1**:Almacena el primer número ingresado por el usuario a través de la vista.
- **num2** :Almacena el segundo número ingresado por el usuario a través de la vista
- **resultado**:Guarda el resultado de la operación matemática realizada (suma, resta, multiplicación o división).

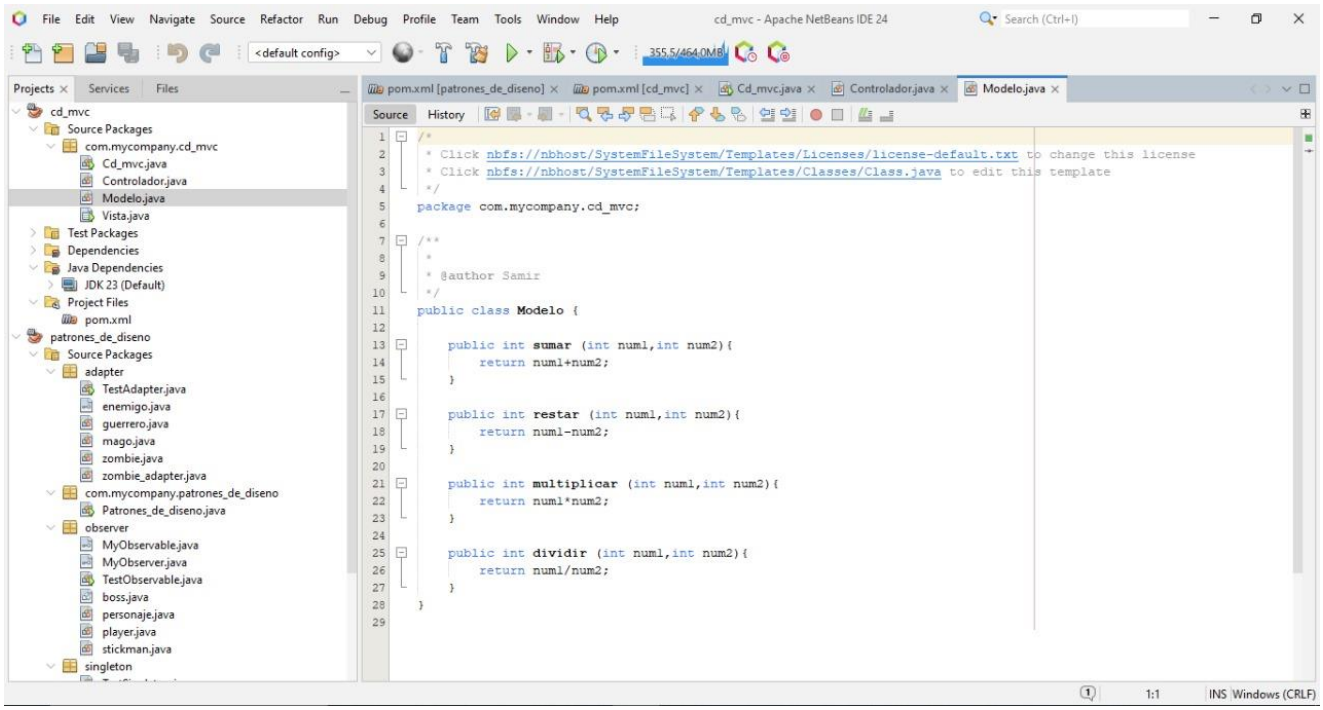


Ilustración 6

sumar(int num1, int num2)

- **Parámetros:** num1, num2 (dos números enteros)
- **Operación:** Retorna la suma de num1 + num2.

restar(int num1, int num2)

- **Parámetros:** num1, num2
- **Operación:** Retorna la resta de num1 - num2.

multiplicar(int num1, int num2)

- **Parámetros:** num1, num2
- **Operación:** Retorna el producto de num1 * num2.

dividir(int num1, int num2)

- **Parámetros:** num1, num2
- **Operación:** Retorna la división de num1 / num2.

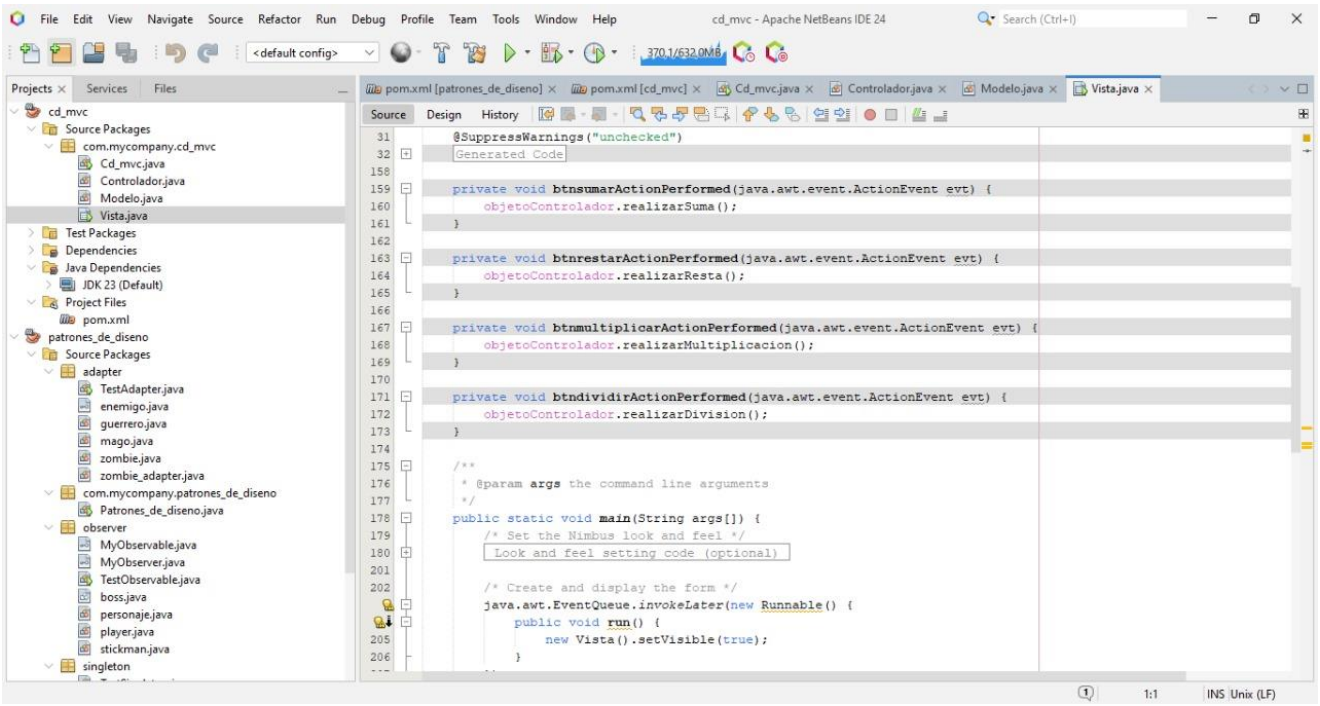


Ilustración 7

- **objetoControlador**: Es una instancia del controlador, utilizada para delegar las operaciones matemáticas.
- **btnsumarActionPerformed(evt)**: Método que se ejecuta cuando el usuario presiona el botón de sumar.
- **btnrestarActionPerformed(evt)**: Método que se ejecuta al presionar el botón de restar.
- **btnmultiplicarActionPerformed(evt)**: Método que se ejecuta al presionar el botón de multiplicar.
- **btndividirActionPerformed(evt)**: Método que se ejecuta al presionar el botón de dividir.
- **main(String args[])**: Método principal que inicia la aplicación y muestra la interfaz gráfica.

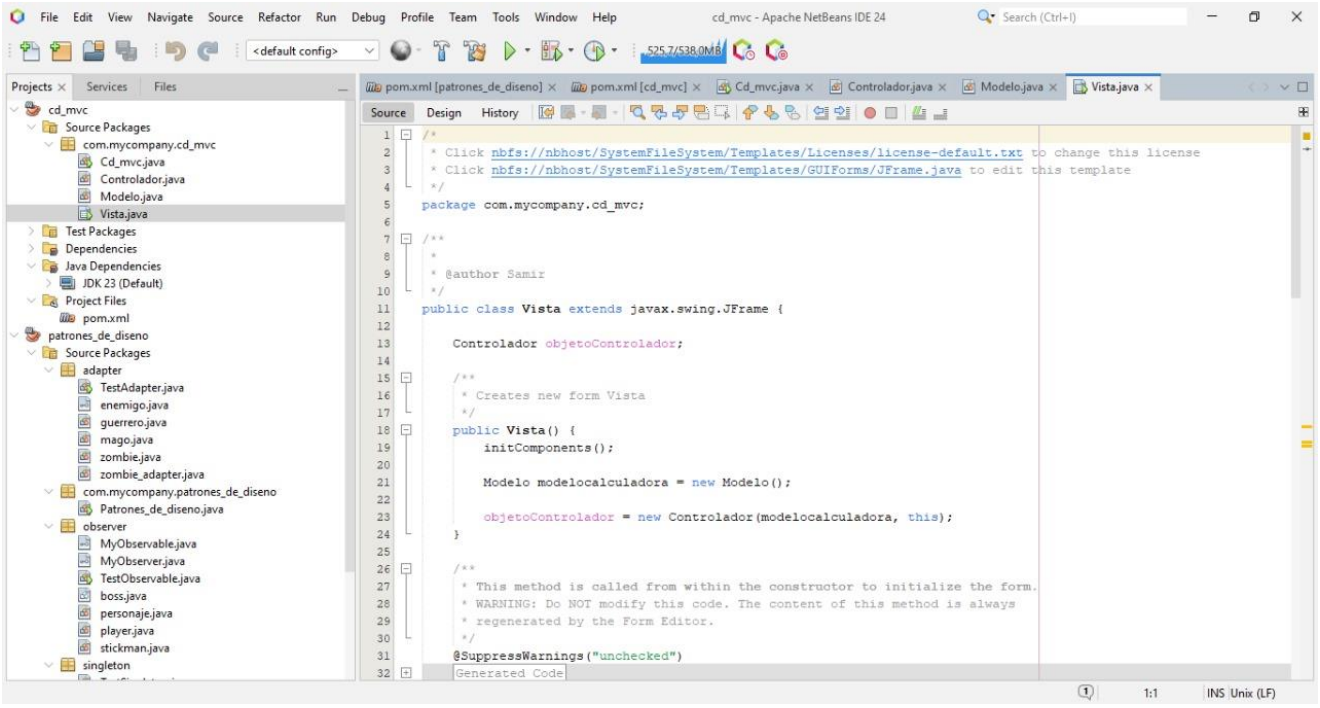


Ilustración 8

- **objetoControlador**: Variable de tipo Controlador, que se usará para manejar la lógica de los eventos en la interfaz.
- **modelocalculadora**: Variable local de tipo Modelo, que representa la parte lógica de los cálculos.

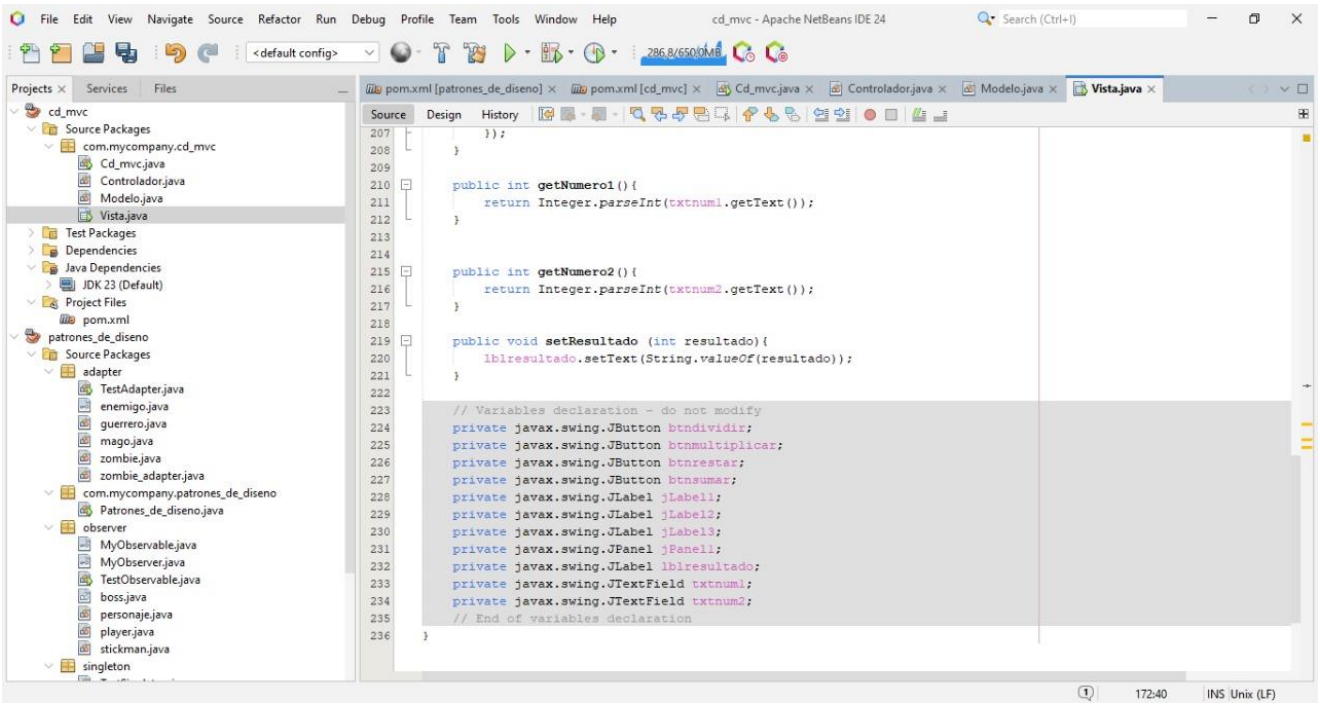


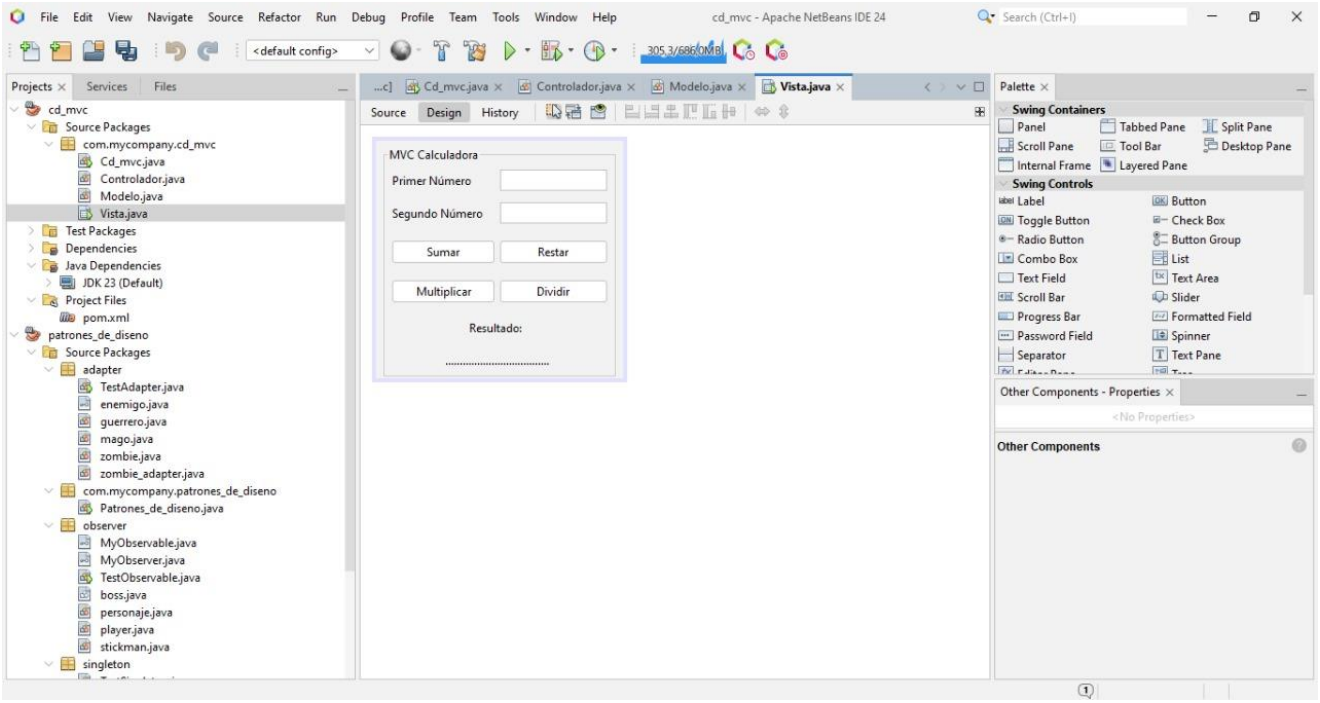
Ilustración 9

Botones (JButton)

- **btndividir:** Botón para dividir dos números.
- **btnmultiplicar:** Botón para multiplicar dos números.
- **btnrestar:** Botón para restar dos números.
- **btnsumar:** Botón para sumar dos números.

Etiquetas (JLabel)

- **jlabel1, jlabel2, jlabel3, jlabel4:** Etiquetas de la interfaz gráfica.



- **Etiqueta (JLabel)**:
    - **"MVC Calculadora":**Es el título de la ventana.
    - **"Primer Número":** Indica el primer campo de entrada.
    - **"Segundo Número" :** Indica el segundo campo de entrada.
    - **"Resultado":** Muestra el resultado de la operación.
- **Campos de texto (JTextField)**:
    - Dos cajas de texto donde el usuario puede ingresar números.
- **Botones (JButton)**:
    - **Sumar** :Realiza la suma de los dos números.
    - **Restar** :Realiza la resta de los dos números.
    - **Multiplicar** :Realiza la multiplicación de los dos números.
    - **Dividir**: Realiza la división de los dos números.

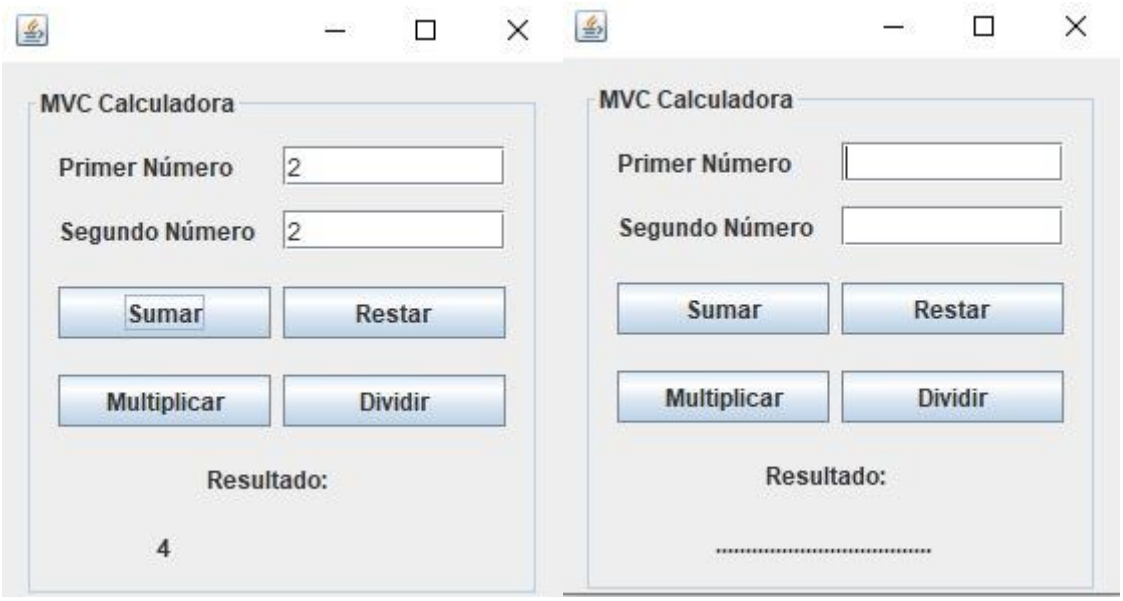


Ilustración 10

Ventana izquierda (calculadora en uso)

- Se han ingresado valores en los campos de texto:
  - **Primer número:** 2
  - **Segundo número:** 2

  Se ha presionado el botón "Sumar", por lo que el resultado mostrado es **4**.

Ventana derecha (calculadora sin uso)

- No se han ingresado valores en los campos de texto.
- El resultado aún no se ha calculado, por lo que se muestra la línea de puntos.

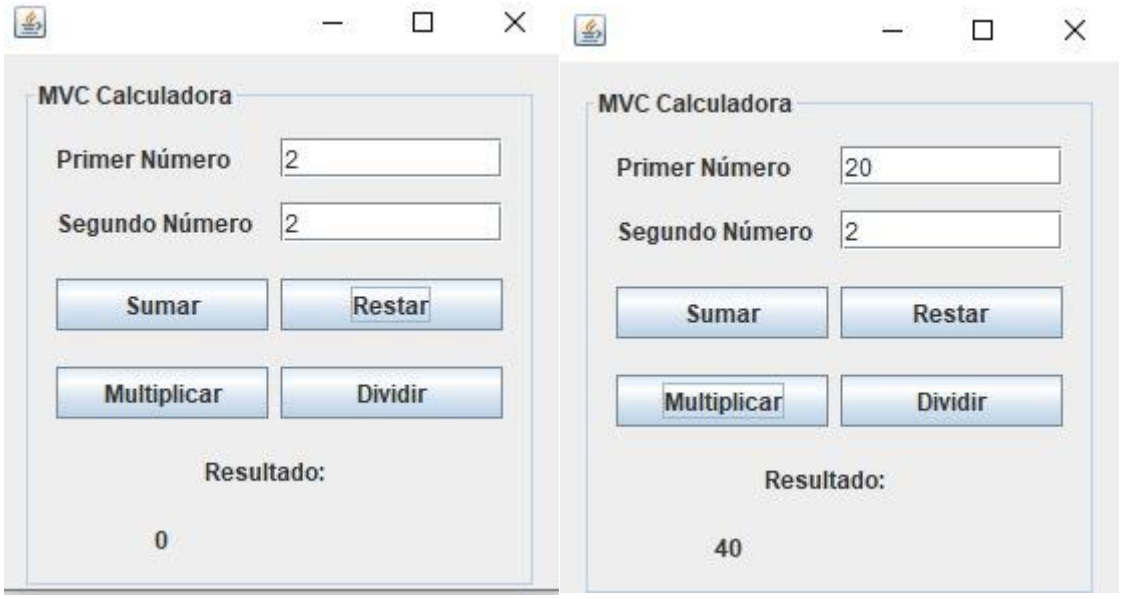


Ilustración 11

- **En la primera ventana:**
  - Se ingresaron los valores 2 y 2.
  - El resultado mostrado es 0, lo que sugiere que se realizó una resta (2 - 2 = 0).
  - El botón Restar parece haber sido presionado.
- **En la segunda ventana:**
  - Se ingresaron los valores 20 y 2.
  - El resultado mostrado es 40, lo que indica que se realizó una multiplicación (20 × 2 = 40).
  - El botón Multiplicar parece haber sido presionado o deshabilitado después de su uso.

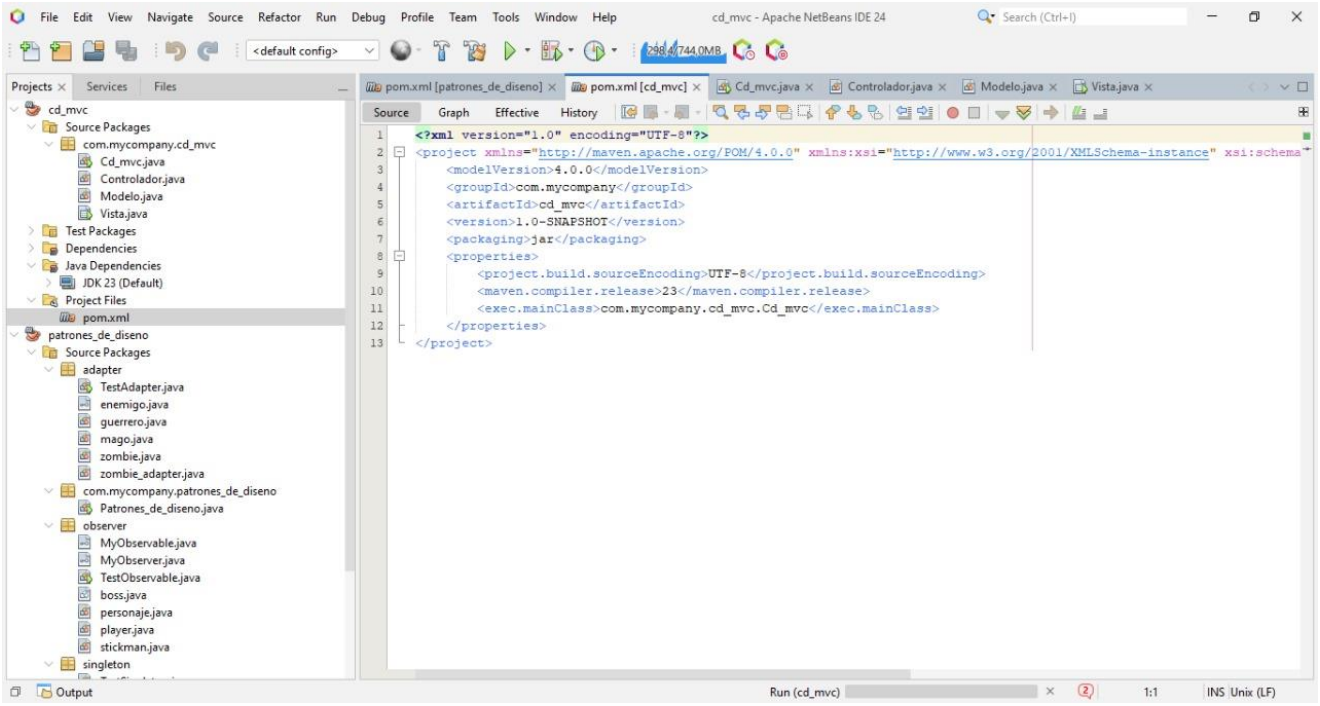


Ilustración 12

- **groupId**: "com.mycompany" :Identifica el grupo de proyectos al que pertenece.
- **artifactId**: "cd_mvc" : Nombre del artefacto del proyecto.
- **version**: "1.0-SNAPSHOT" : Versión del proyecto.
- **packaging**: "jar" :Indica que el resultado de la compilación será un archivo JAR.
- **maven.compiler.release**: "23" :Define que el proyecto se compila con Java 23.
- **exec.mainClass**: "com.mycompany.cd_mvc.Cd_mvc": Especifica la clase principal del proyecto.

- **PATRÓN DE DISEÑO ADAPTER**

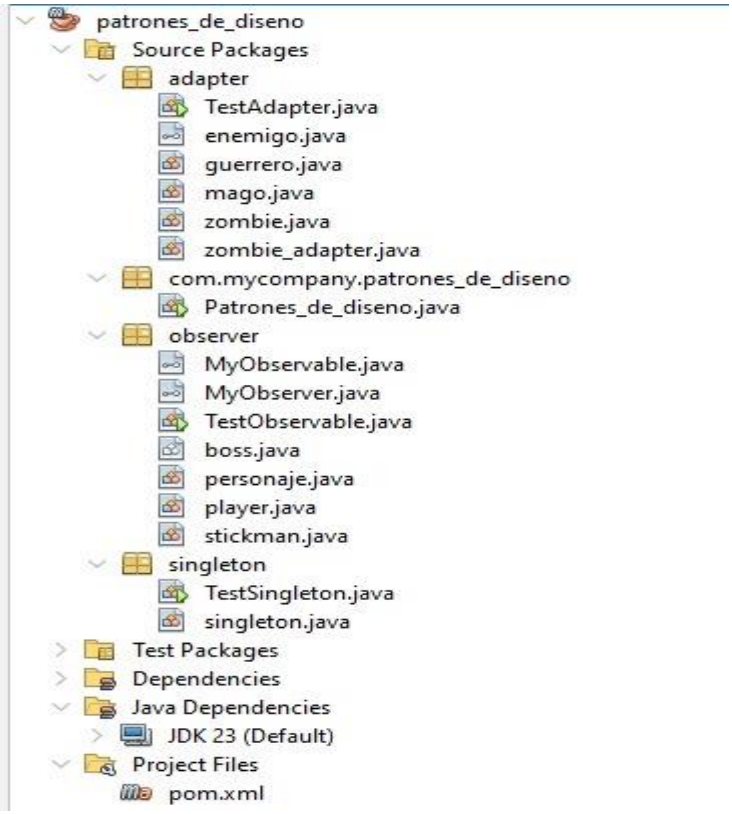


Ilustración 13

- **adapter**: Aquí parece que se está trabajando con la idea de adaptar diferentes tipos de personajes o entidades. Hay clases como enemigo, guerrero, mago, zombie y zombie_adapter, lo que sugiere que algunos personajes necesitan ser convertidos o adaptados para interactuar con otros de manera más flexible. También hay una clase TestAdapter, probablemente usada para probar la implementación.
- **observer**: Este paquete parece estar manejando el patrón Observer, que se usa para notificar cambios entre objetos. Clases como MyObservable y MyObserver refuerzan esta idea. Además, hay personajes como boss, player, personaje y stickman, lo que sugiere que el sistema podría estar monitoreando eventos o cambios en el estado de estos personajes en tiempo real.

- **singleton**: Aquí se está aplicando el patrón Singleton, que asegura que solo haya una única instancia de una clase en toda la ejecución del programa. singleton.java es la clase que probablemente lo implementa, mientras que TestSingleton.java parece estar diseñada para probar que realmente funciona como se espera.

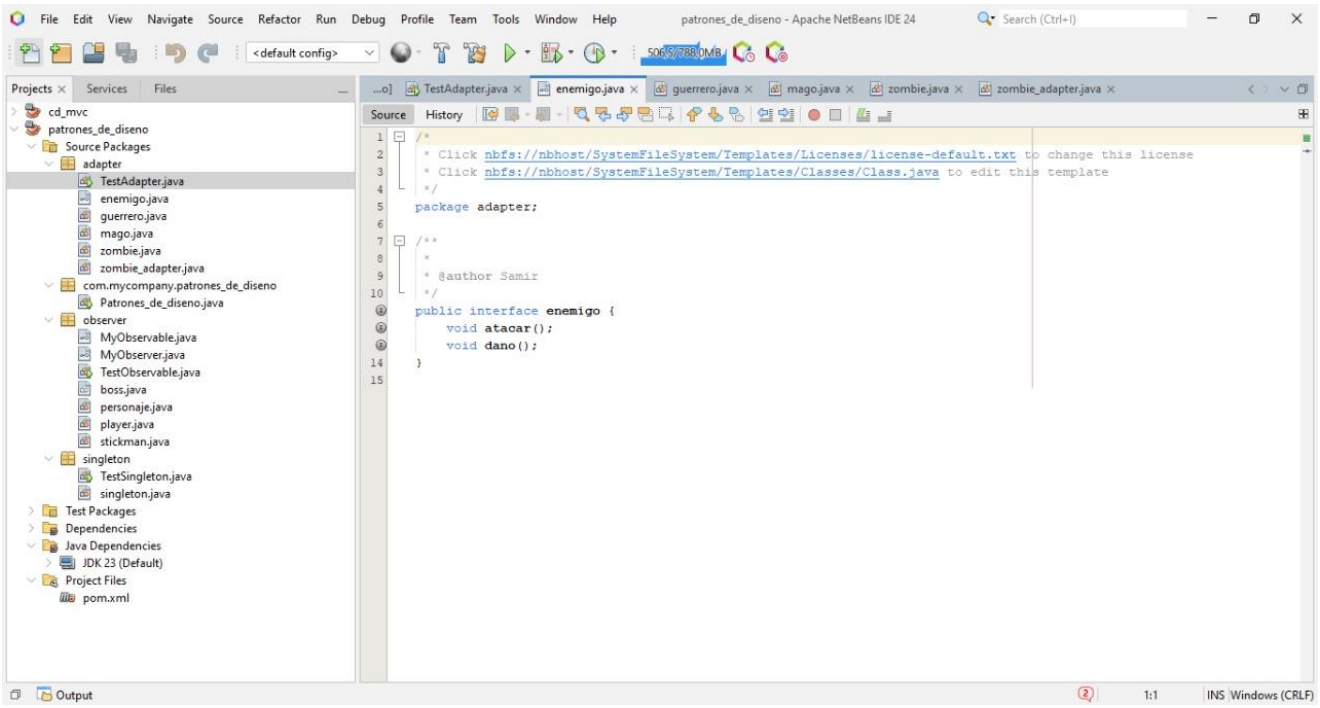


Ilustración 14

package adapter;

- Indica que esta interfaz pertenece al paquete adapter, lo que sugiere que forma parte del patrón de diseño Adapter.

public interface enemigo {}

- Declara una interfaz llamada enemigo.
- En Java, una interfaz define un contrato que otras clases deben implementar.

void atacar();

- Es un método abstracto (sin implementación).
- Obliga a cualquier clase que implemente la interfaz enemigo a definir su propia versión del método atacar(), que probablemente represente una acción ofensiva.

void dano();

- También es un método abstracto.
- Cualquier clase que implemente la interfaz enemigo deberá definir dano(), que probablemente represente el daño recibido o causado en un juego.

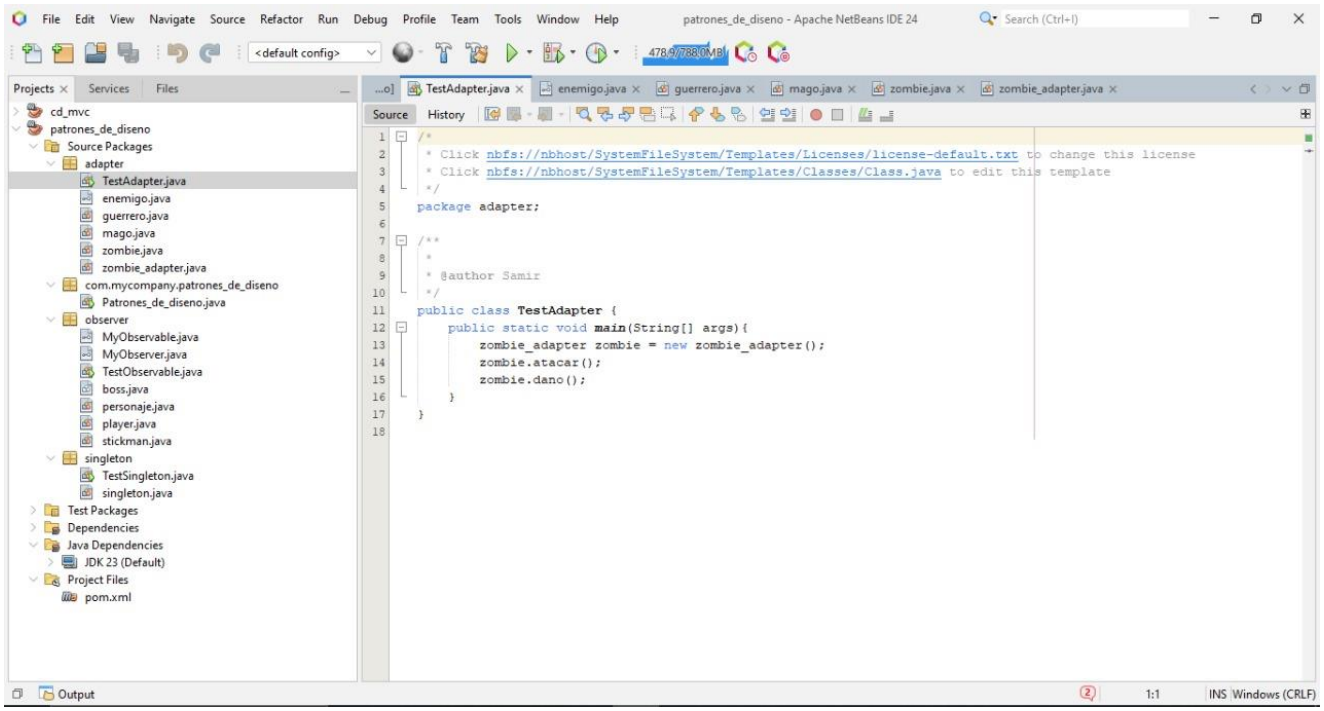


Ilustración 15

zombie_adapter zombie = new zombie_adapter();

- Crea una instancia del adaptador zombie_adapter.

zombie.atacar();

- Llama al método atacar() del adaptador.

zombie.dano();

- Llama al método dano() del adaptador.

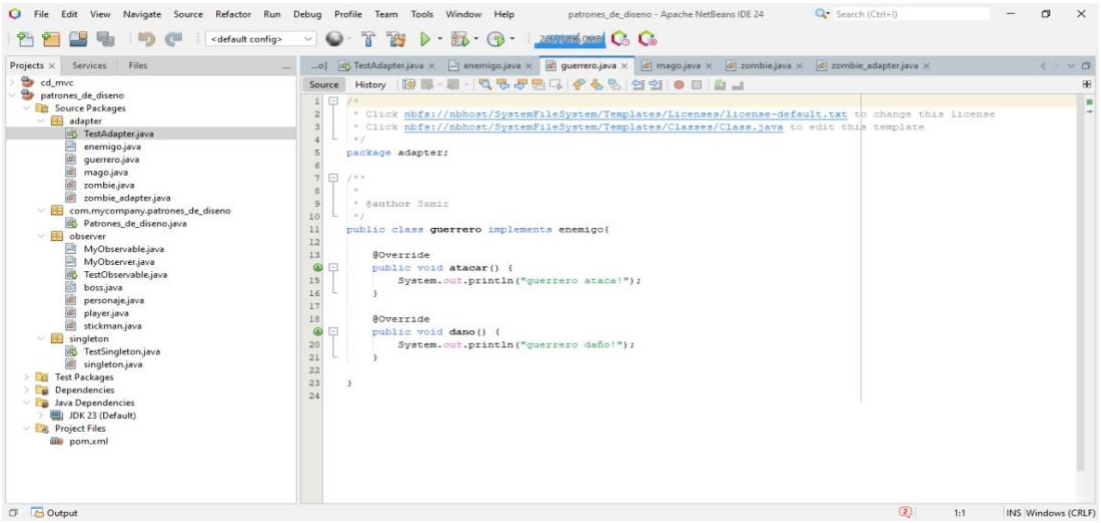


Ilustración 16

- **atacar()**: Imprime "guerrero ataca!".
- **dano()** :Imprime "guerrero daño!".

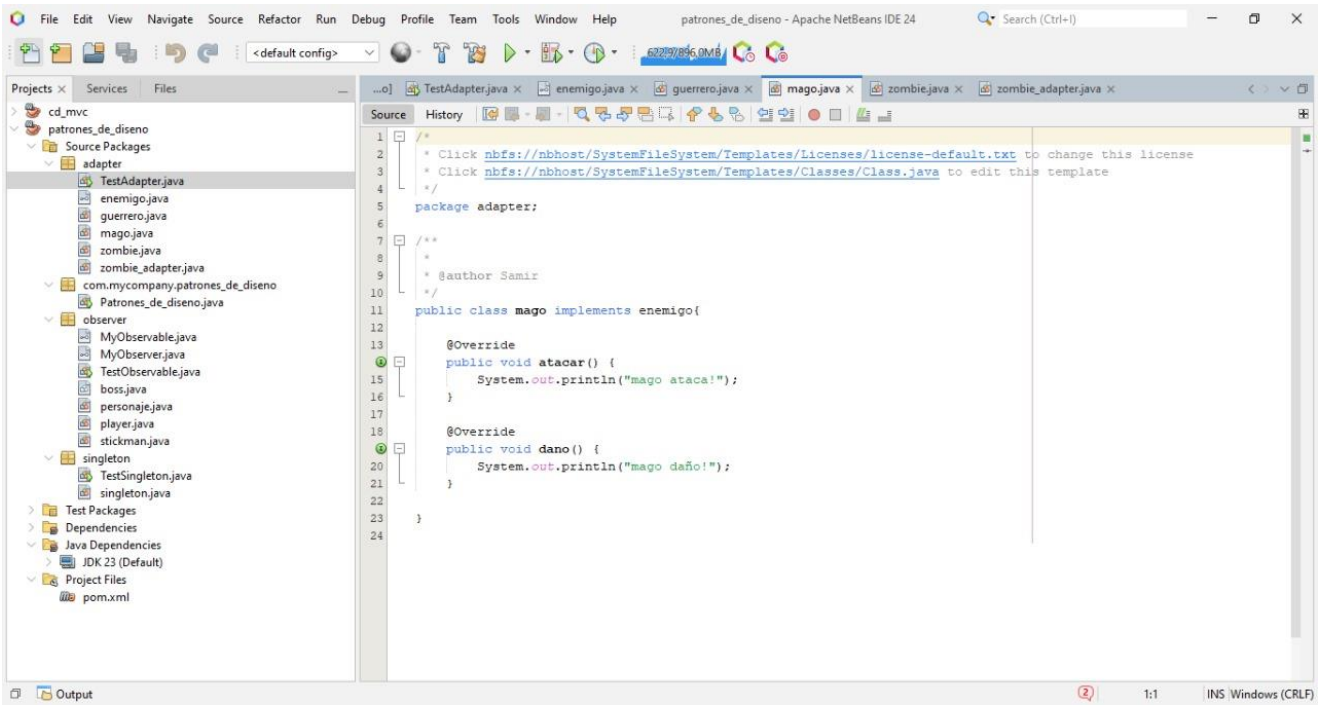


Ilustración 17

- **public void atacar():** Imprime "mago ataca!" en la consola.
- **public void dano():** Imprime "mago daño!" en la consola.

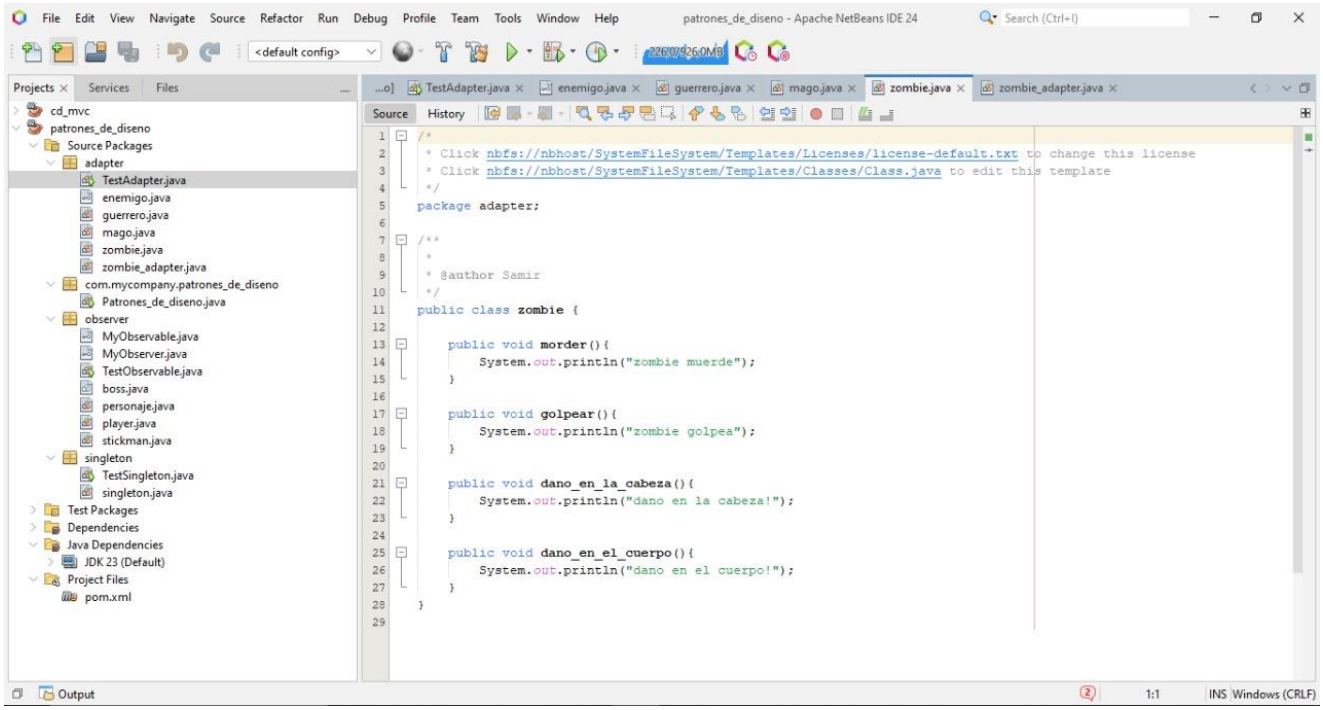


Ilustración 18

- **morder()** : Imprime "zombie muerde".
- **golpear()** : Imprime "zombie golpea".
- **dano_en_la_cabeza()** : Imprime "daño en la cabeza!".
- **dano_en_el_cuerpo()** : Imprime "daño en el cuerpo!".

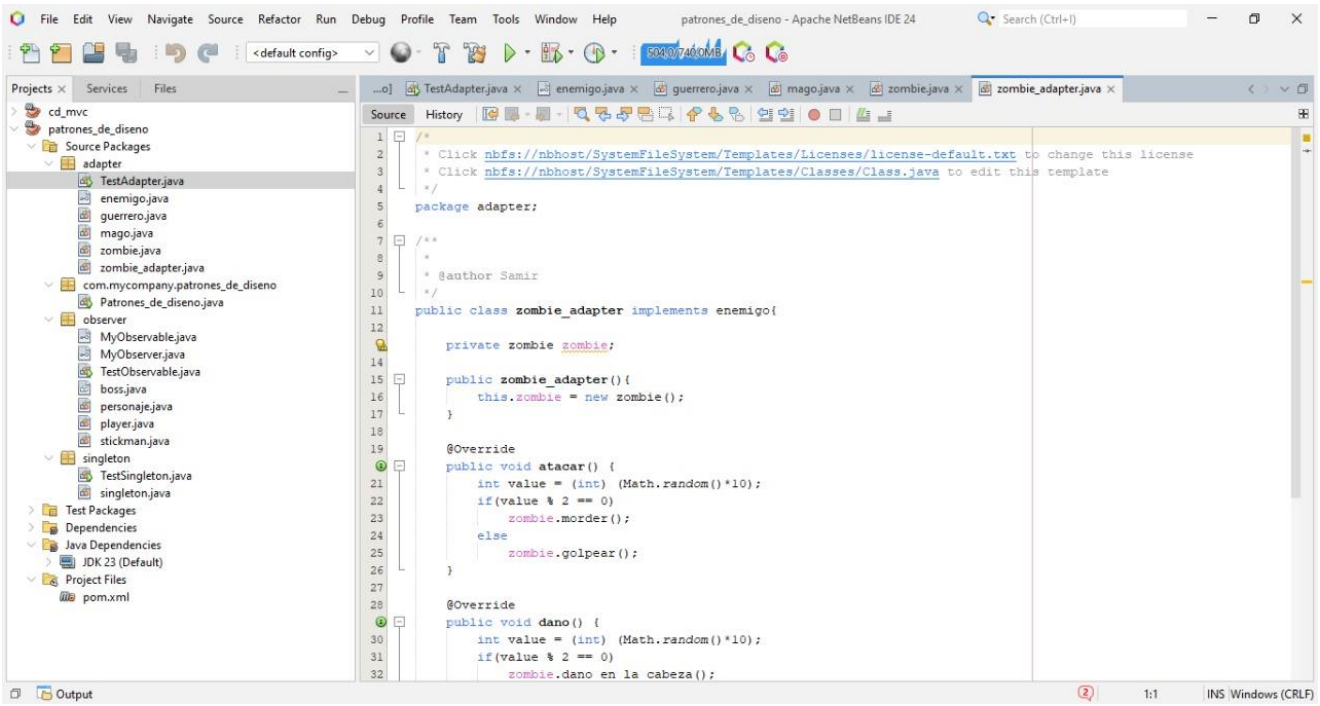


Ilustración 19

private zombie zombie;

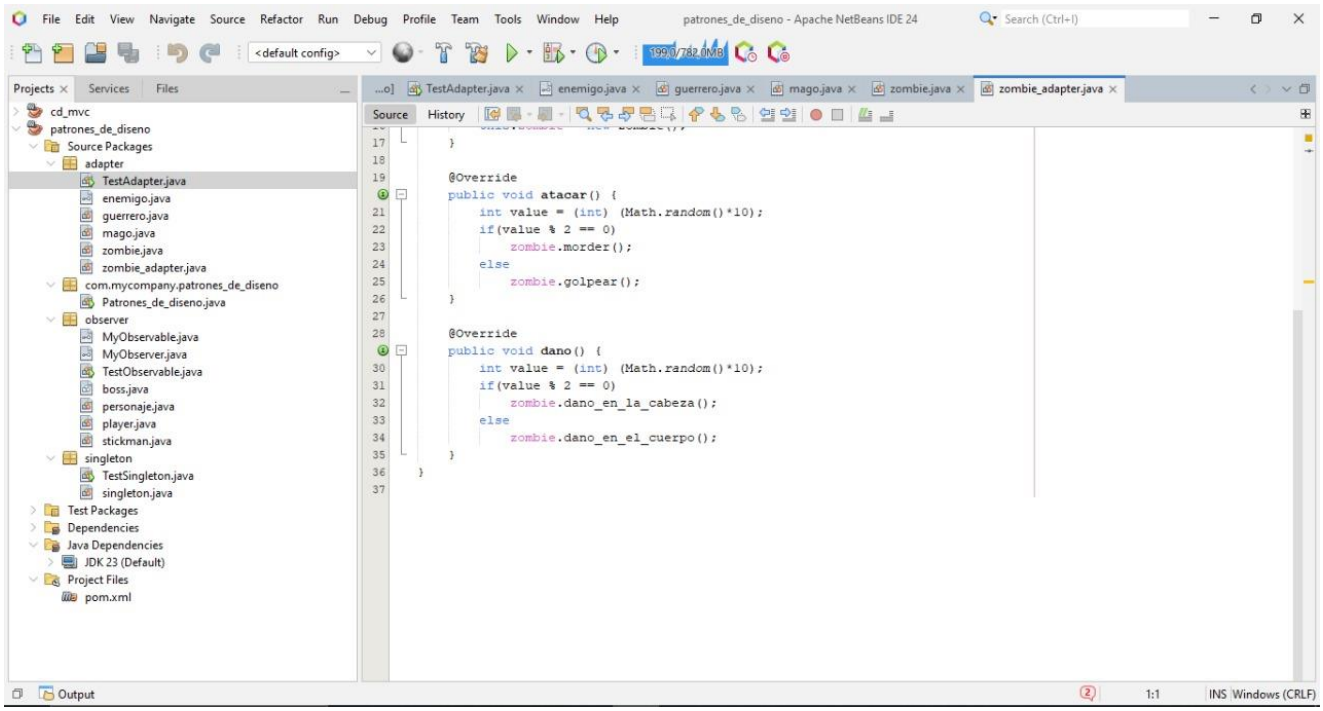
- Es una variable de instancia de tipo zombie.
- Se usa para crear un objeto zombie dentro del adaptador.

public zombie_adapter()

- Es el constructor de la clase.
- Inicializa this.zombie con una nueva instancia de zombie.

int value = (int) (Math.random() * 10);

- Se encuentra en los métodos atacar() y dano().
- Genera un número aleatorio entre 0 y 9.
- Se usa para decidir qué acción tomará el zombie (morder/golpear o causar daño en la cabeza/cuerpo).



Ilustración

value en atacar()

- Si el número es par, el zombie ejecuta una acción de morder.
- Si el número es impar, el zombie realiza un golpe.

Cómo se usa value en dano()

- Si el número es par, el daño se produce en la cabeza del objetivo.
- Si el número es impar, el daño se aplica en el cuerpo.

```
zombie muerde
dano en el cuerpo!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.507 s
Finished at: 2025-02-21T10:38:08-05:00
------------------------------------------------------------------------
```

Ilustración 20

- **zombie muerde:**Esto significa que el número aleatorio fue par, activando la acción de "morder".
- **daño en el cuerpo :**El número aleatorio fue impar, por lo que el daño se aplicó en el cuerpo.
- **Tiempo total:** 2.507 segundos

```
zombie golpea
dano en el cuerpo!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.349 s
Finished at: 2025-02-21T10:38:34-05:00
------------------------------------------------------------------------
```

Ilustración 21

- **zombie golpea :** El número aleatorio fue impar, lo que hizo que el zombie golpee en lugar de morder.
- **daño en el cuerpo :** El número aleatorio fue impar, causando daño en el cuerpo.
- **Tiempo total:** 2.349 segundos

```
zombie golpea
dano en la cabeza!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.535 s
Finished at: 2025-02-21T10:39:27-05:00
------------------------------------------------------------------------
```

Ilustración 22

- **zombie golpea**: Él número aleatorio fue impar, activando la acción de "golpear".
- **daño en la cabeza:** El número aleatorio fue par, causando daño en la cabeza.
- **Tiempo total:** 2.535 segundos

```
zombie muerde
dano en la cabeza!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.535 s
Finished at: 2025-02-21T10:39:27-05:00
------------------------------------------------------------------------
```

Ilustración 23

- **zombie muerde :** El número aleatorio fue par, por lo que el zombie mordió.
- **daño en la cabeza :** El número aleatorio fue par, causando daño en la cabeza.
- **Tiempo total:** 2.535 segundos

- **PATRÓN DE DISEÑO OBSERVER**

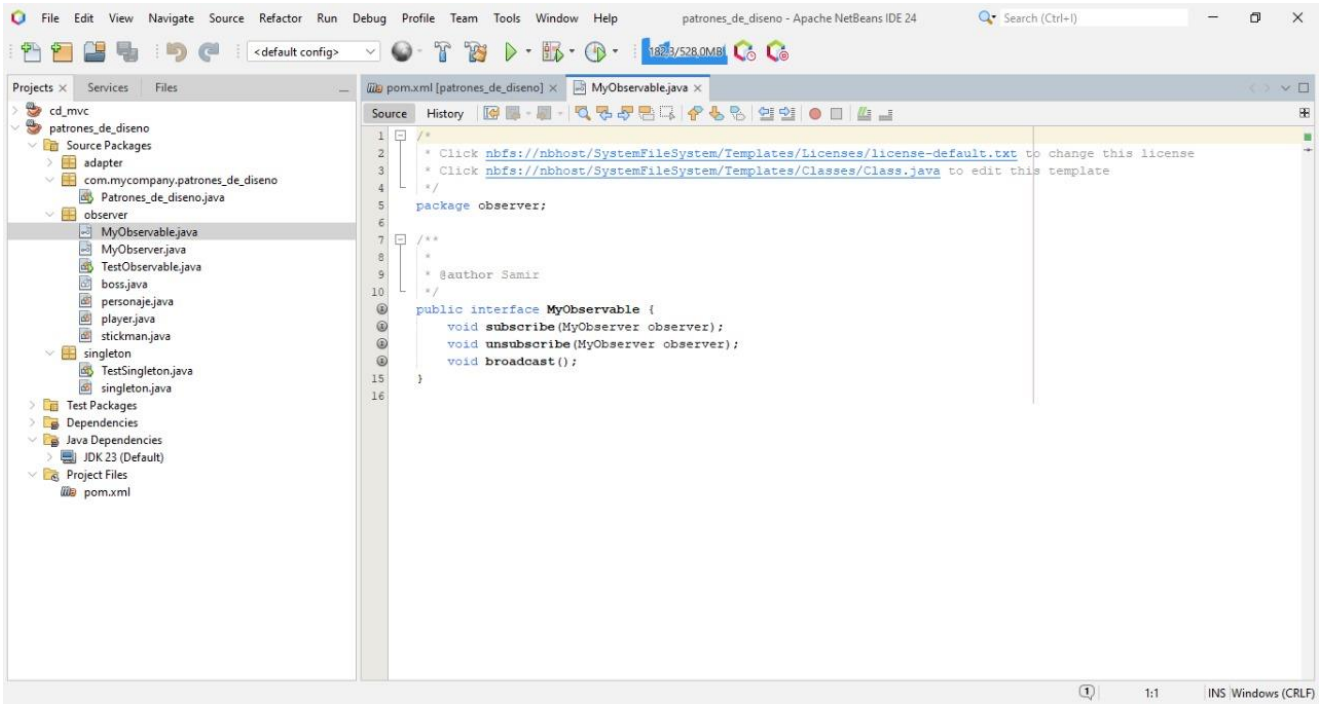


Ilustración 24

- **subscribe(MyObserver observer):**
  - Permite que un observador (de tipo MyObserver) se registre en el observable.
  - Esencialmente, agrega el observador a la lista de los que recibirán notificaciones.
- **unsubscribe(MyObserver observer):**
  - Permite que un observador deje de recibir notificaciones.
  - Básicamente, lo elimina de la lista de observadores.

- **broadcast():**
    - Se encarga de notificar a todos los observadores registrados cuando ocurre un cambio en el estado del observable.
    - Llamará a un método en cada observador para informarle de los cambios.

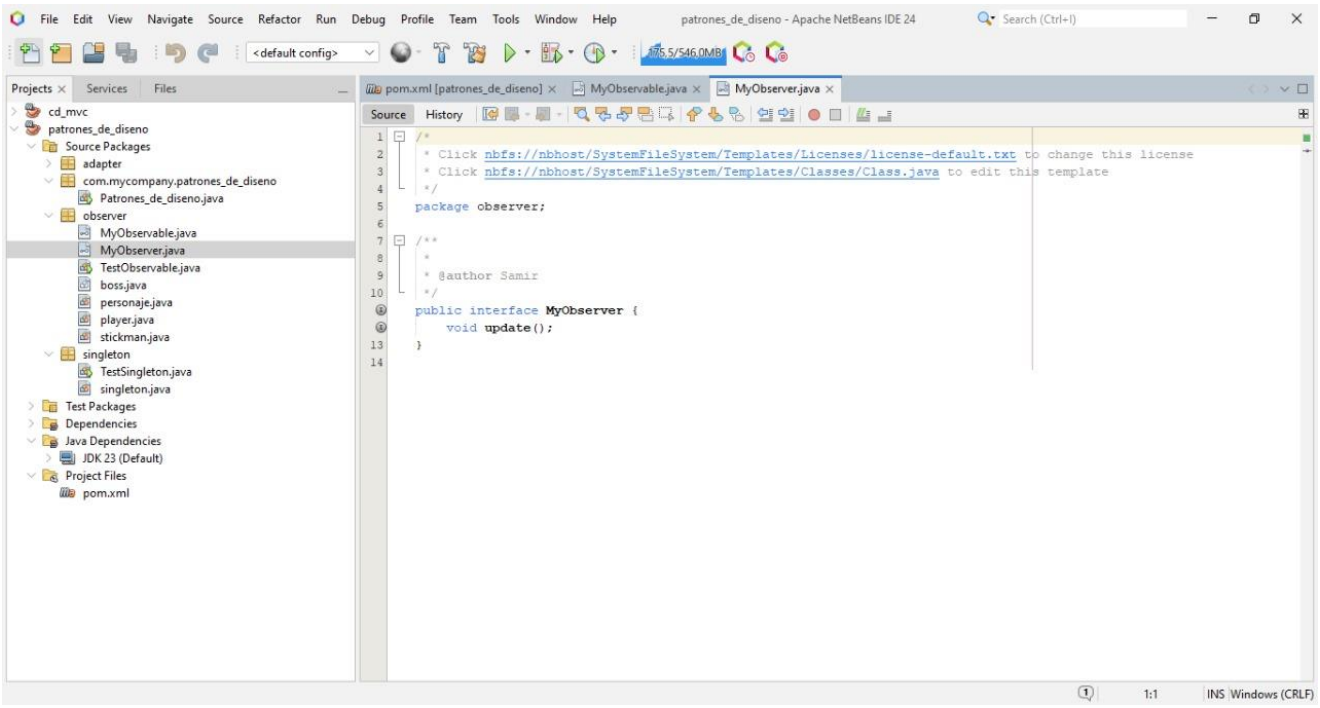


Ilustración 25

- **MyObserver**: Es una **interfaz** que establece un contrato para los observadores. Cualquier clase que implemente esta interfaz debe definir el método update().
- **void update()**: Es un método **abstracto** que las clases observadoras deben implementar. Su propósito es definir la acción a ejecutar cuando el sujeto observado cambia de estado.

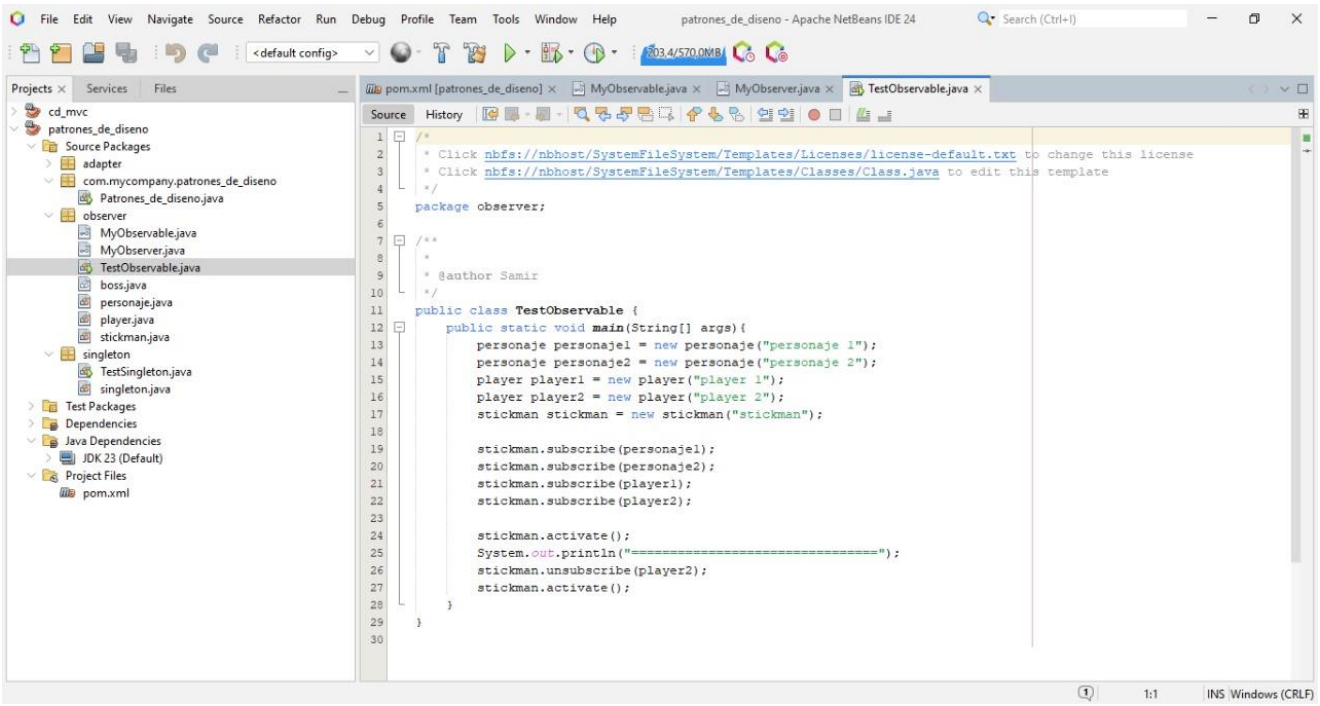


Ilustración 26

personaje personaje1 = new personaje("personaje 1");

- Crea un objeto personaje1 con el nombre "personaje 1".
- personaje probablemente implementa la interfaz MyObserver, lo que significa que puede ser notificado de cambios.

personaje personaje2 = new personaje("personaje 2");

- Igual que personaje1, pero con el nombre "personaje 2".

player player1 = new player("player 1");

- Crea un objeto player1 con el nombre "player 1".
- player también parece ser un observador.

player player2 = new player("player 2");

- Igual que player1, pero con el nombre "player 2".

stickman stickman = new stickman("stickman");

- Crea un objeto stickman.
- stickman parece ser el sujeto observable, que notificará a los observadores cuando cambie su estado.

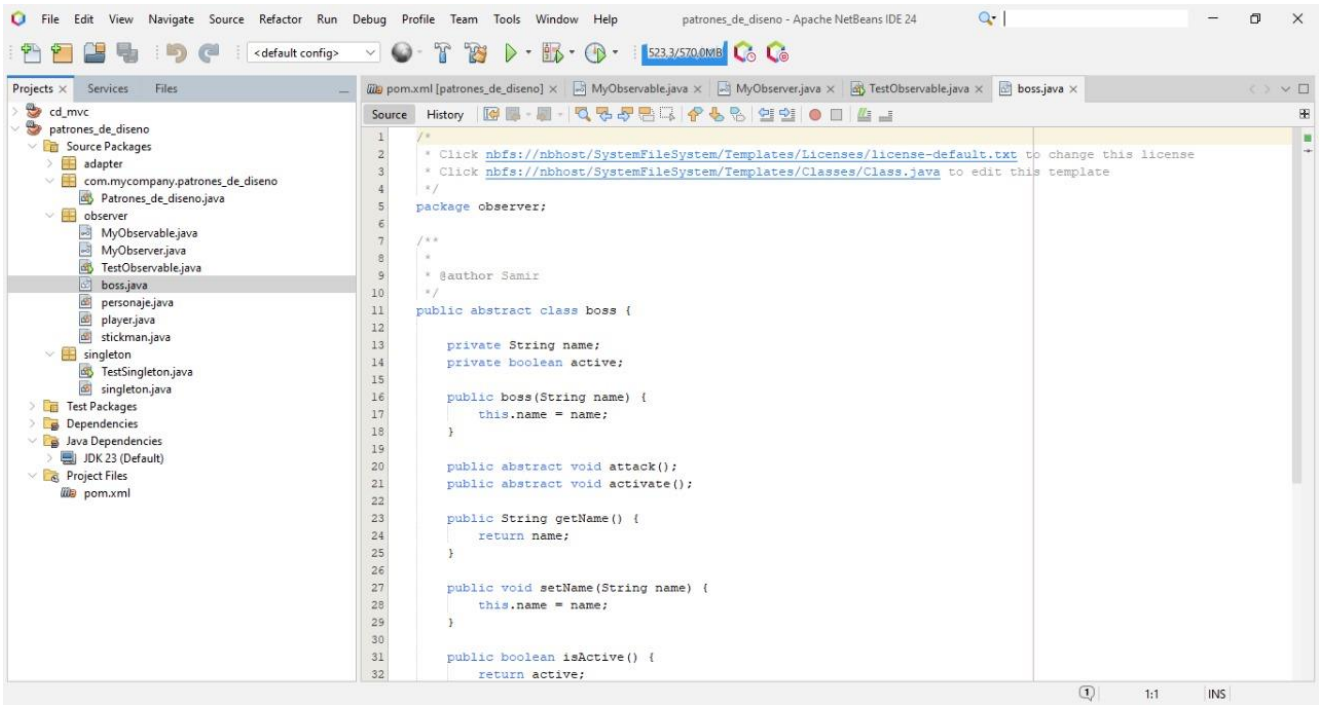


Ilustración 27

- **name**: Almacena el nombre del jefe.
- **active**: Indica si el jefe está activo o no.

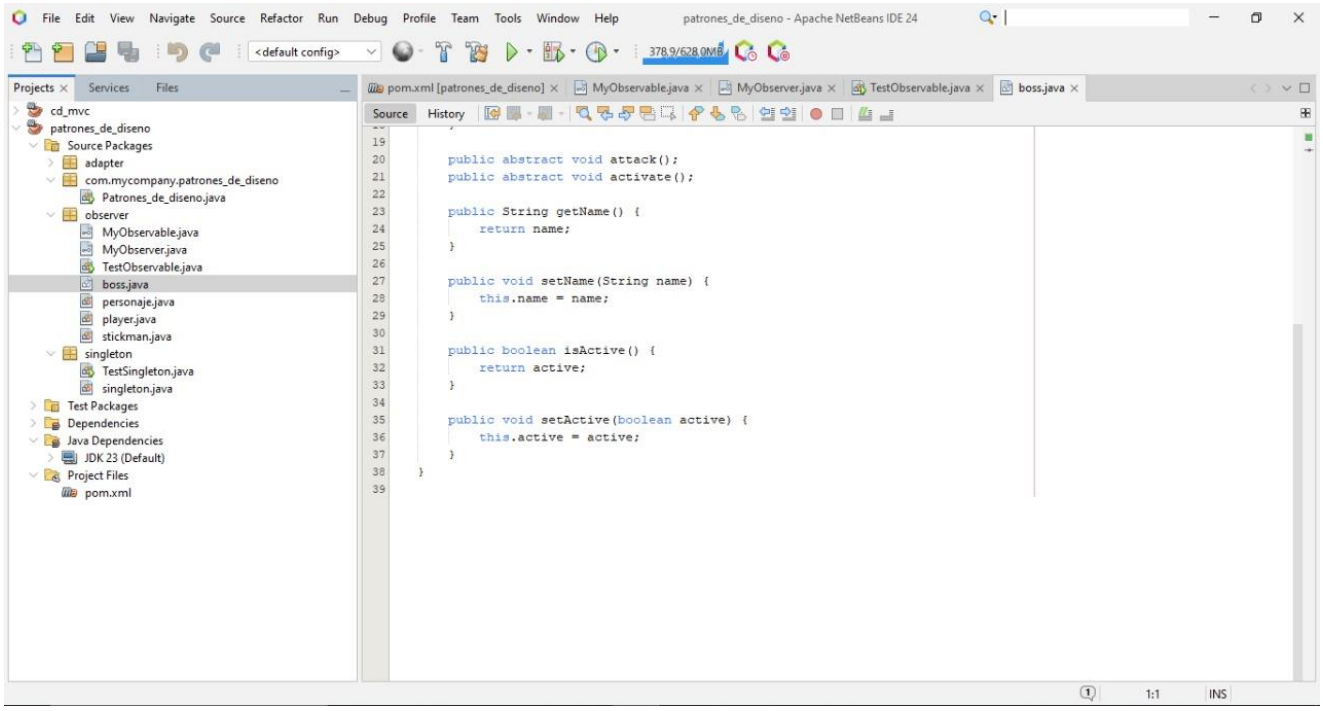


Ilustración 28

- **attack() y activate()**: Son métodos abstractos que deben ser implementados en las clases hijas.
- **getName() y setName(String name)**: Permiten obtener y modificar el nombre del jefe.
- **isActive() y setActive(boolean active)**: Permiten consultar y modificar el estado activo del jefe.

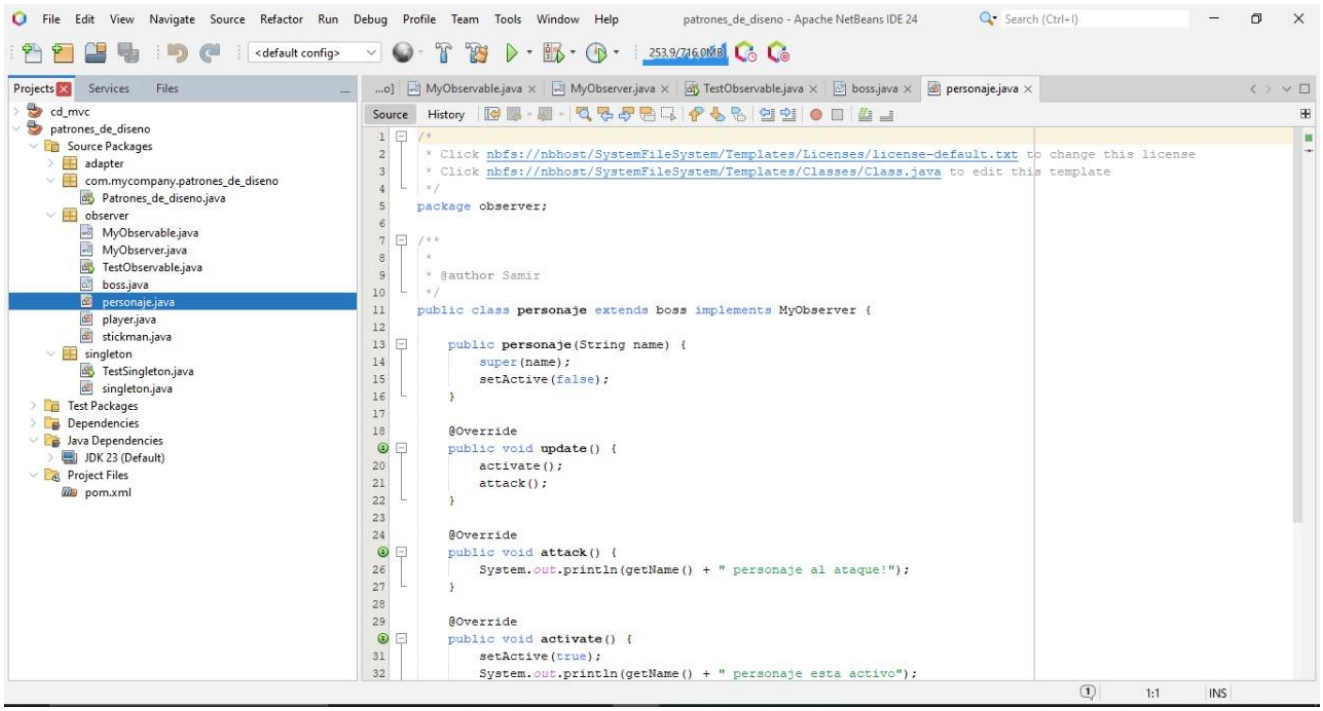


Ilustración 29

- **Constructor (personaje(String name))**: Llama al constructor de boss con el nombre del personaje y lo inicia como inactivo.
- **update()**: Se ejecuta cuando el objeto observado cambia, activando al personaje y ordenándole atacar.
- **attack()**: Imprime un mensaje indicando que el personaje está atacando.
- **activate()**: Activa al personaje e imprime un mensaje indicando que ahora está activo.

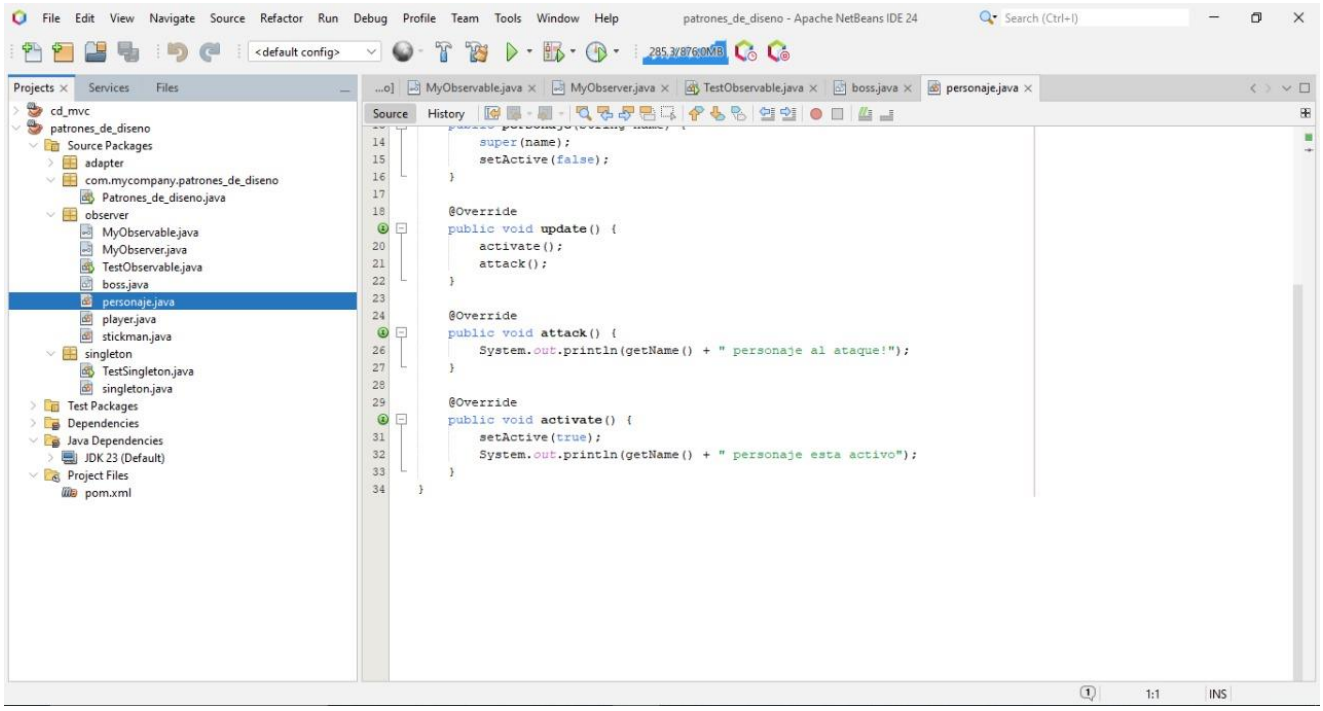


Ilustración 30

update()

- Se invoca cuando el objeto observado notifica un cambio.
- Llama internamente a activate() y a attack(), lo que significa que, ante una notificación, el personaje se activa y ataca.

attack()

- Imprime un mensaje indicando que el personaje, con su nombre, está atacando.
- Representa la acción ofensiva del personaje.

activate()

- Cambia el estado del personaje a activo (setActive(true)) y muestra un mensaje informando que el personaje se ha activado.
- Indica que el personaje está listo para realizar acciones.

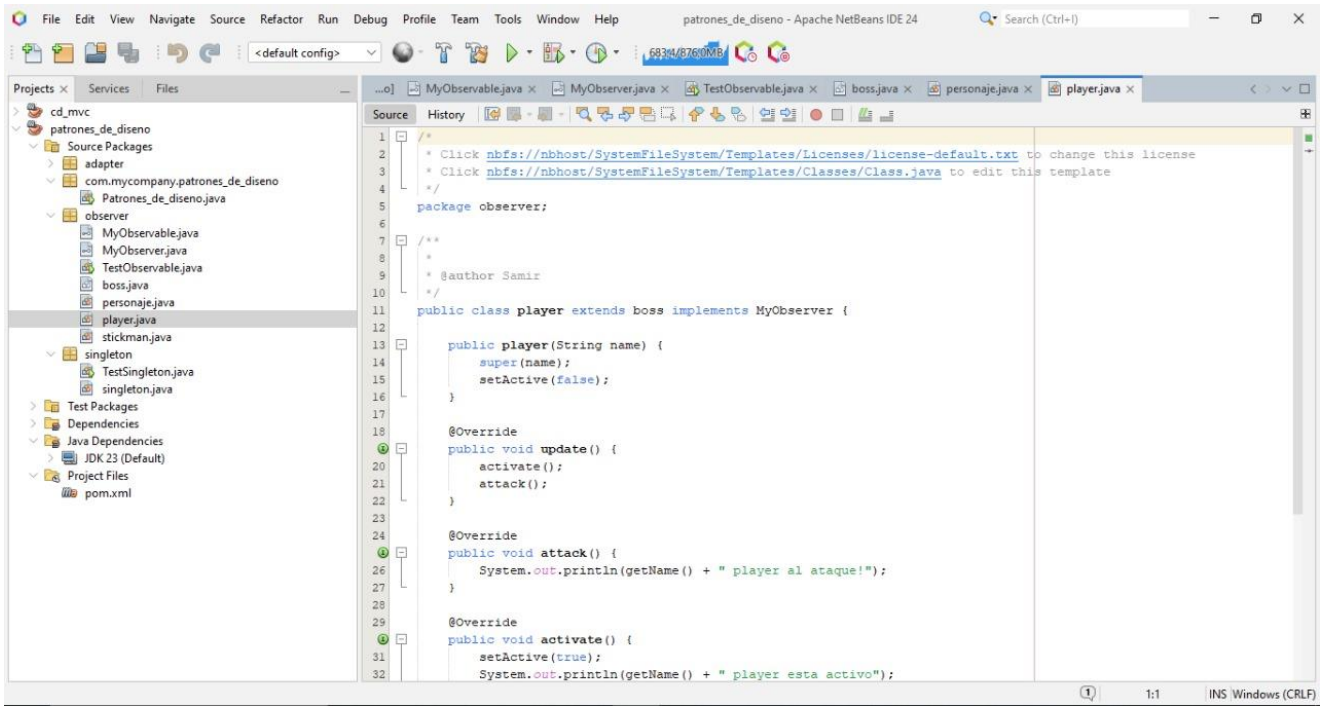


Ilustración 31

Extiende boss e implementa MyObserver

- **boss:** Parece ser una clase base con ciertos comportamientos comunes para los personajes/jugadores.
- **MyObserver:** Indica que player es un observador que reaccionará a eventos de un **observable**.

Constructor

- Recibe un name como parámetro y llama al constructor de boss con super(name).
- Llama a setActive(false), lo que significa que los jugadores inician desactivados.

Método update() (Sobrescrito de MyObserver)

- Este método se ejecuta cuando el observable notifica un cambio.
- Llama a activate() y attack(), lo que hace que el jugador se active y ataque cuando recibe una actualización.

Método attack()

- Imprime un mensaje indicando que el jugador está atacando.
- " + player al ataque!", lo que lo diferencia de personaje.java, que imprimía " + personaje al ataque!".

Método activate()

- Cambia el estado del jugador a activo (setActive(true)).
- Muestra un mensaje de activación (getName() + " player esta activo").

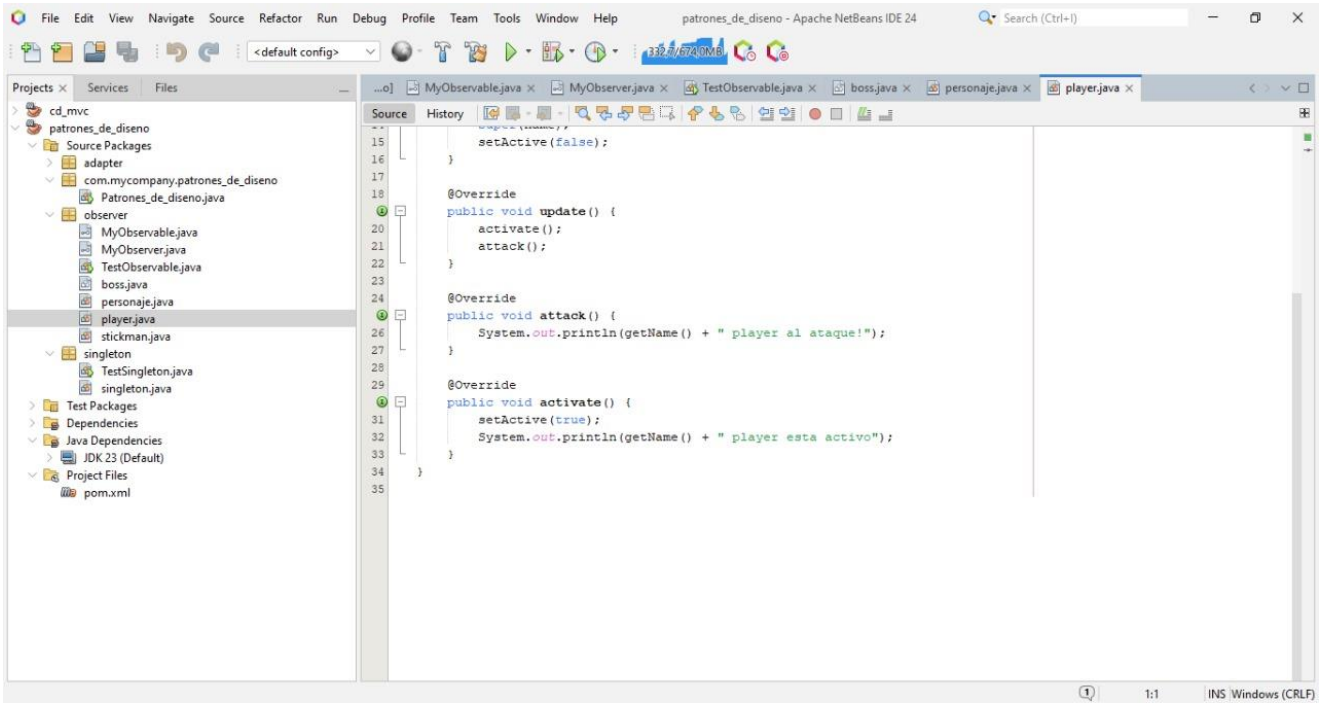


Ilustración 32

Crear una clase base Character

- player y personaje podrían extender Character, que tendría los métodos comunes (update(), attack(), activate()).
- La diferencia en el mensaje ("player" o "personaje") podría manejarse con una variable.

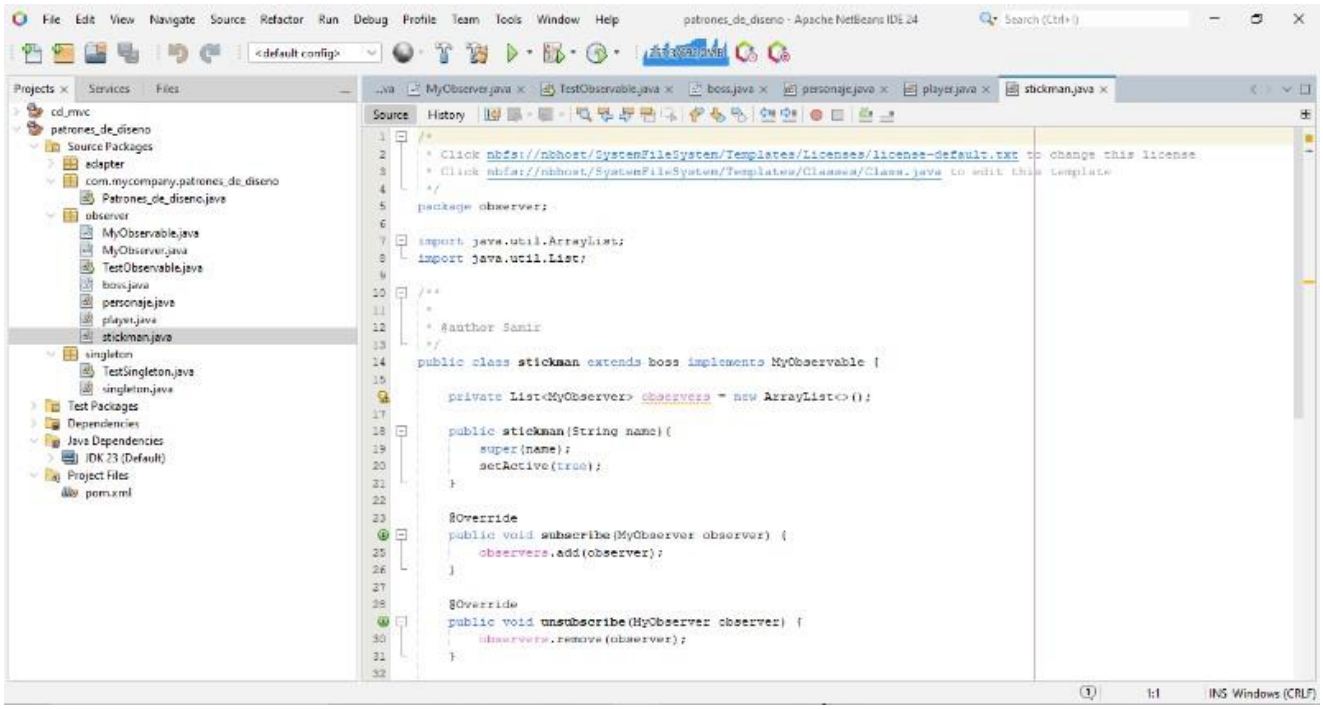


Ilustración 33

name (definida en la superclase boss)

- Almacena el nombre del stickman.
- Se usa para identificarlo dentro del programa.

active (definida en la superclase boss)

- Indica si stickman está activo o inactivo.
- Se usa para controlar su estado en la lógica del juego o aplicación.

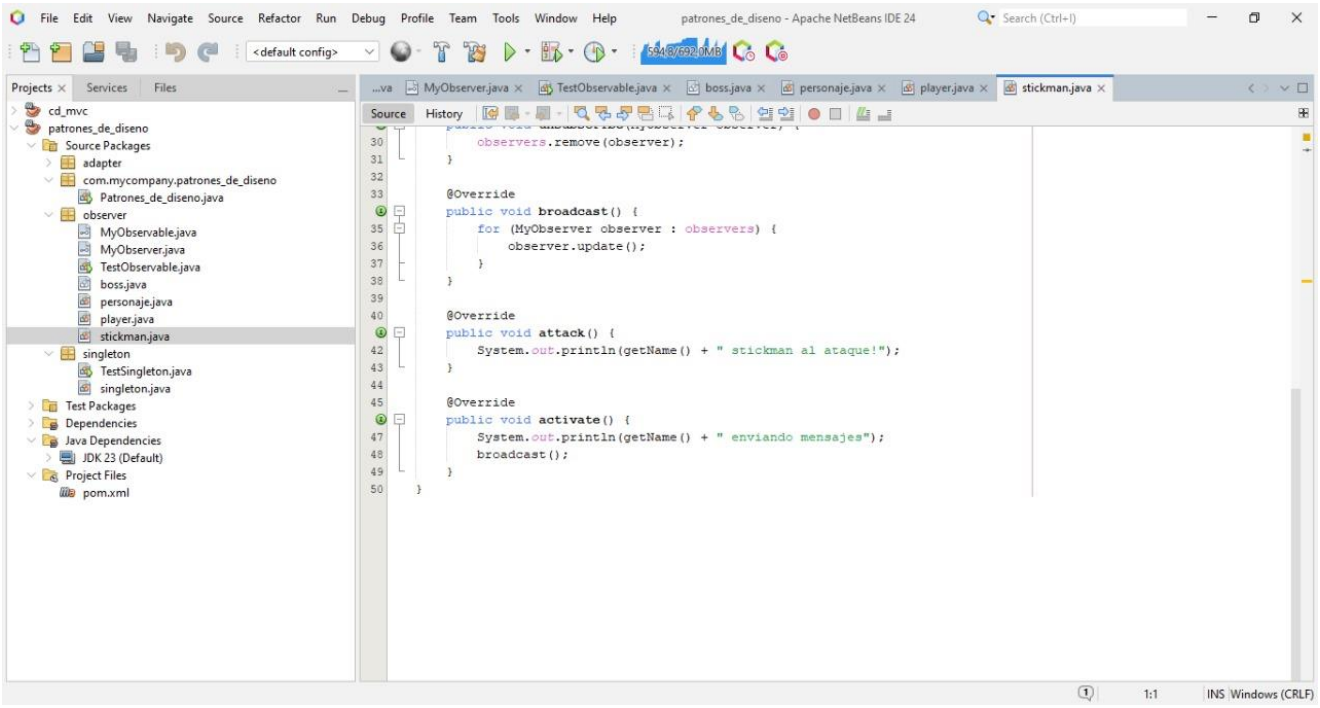


Ilustración 34

observers (List<MyObserver> observers)

- Lista que almacena los observadores suscritos a stickman.
- Se usa para notificar a todos los observadores cuando hay un cambio.

observer (variable dentro del método broadcast())

- Representa cada observador dentro del ciclo for.
- Se usa para llamar al método update() de cada observador.

name (heredada de boss)

- Contiene el nombre del stickman.
- Se usa en los mensajes impresos en System.out.println().

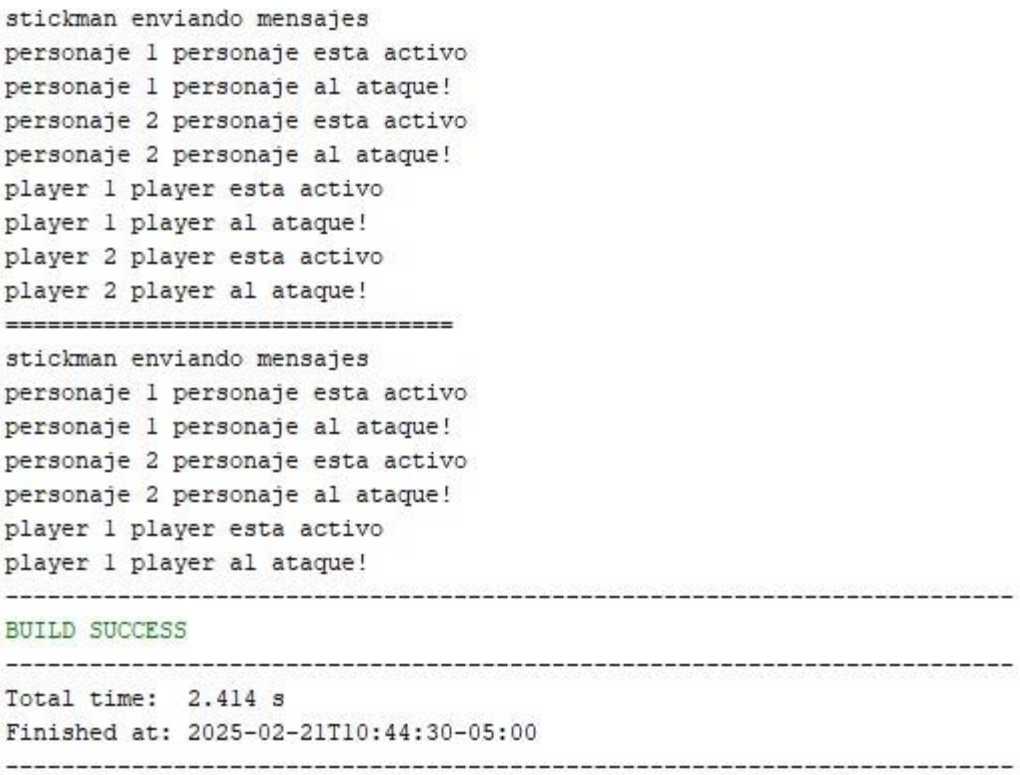


Ilustración 35

"stickman enviando mensajes" (indicando que el sistema está enviando información).

- "personaje 1 personaje está activo" y "personaje 1 personaje al ataque!" (señalando que el personaje 1 está activo y atacando).
- "personaje 2 personaje está activo" y "personaje 2 personaje al ataque!" (lo mismo para el personaje 2).
- "player 1 player está activo" y "player 1 player al ataque!" (jugador 1 está activo y ataca).
- "player 2 player está activo" y "player 2 player al ataque!" (jugador 2 hace lo mismo).

Tiempo total de ejecución: 2.414 segundos

Finalización: 21 de febrero de 2025 a las 10:44:30 (-05:00 GMT)

- **PATRÓN DE DISEÑO SINGLETON**

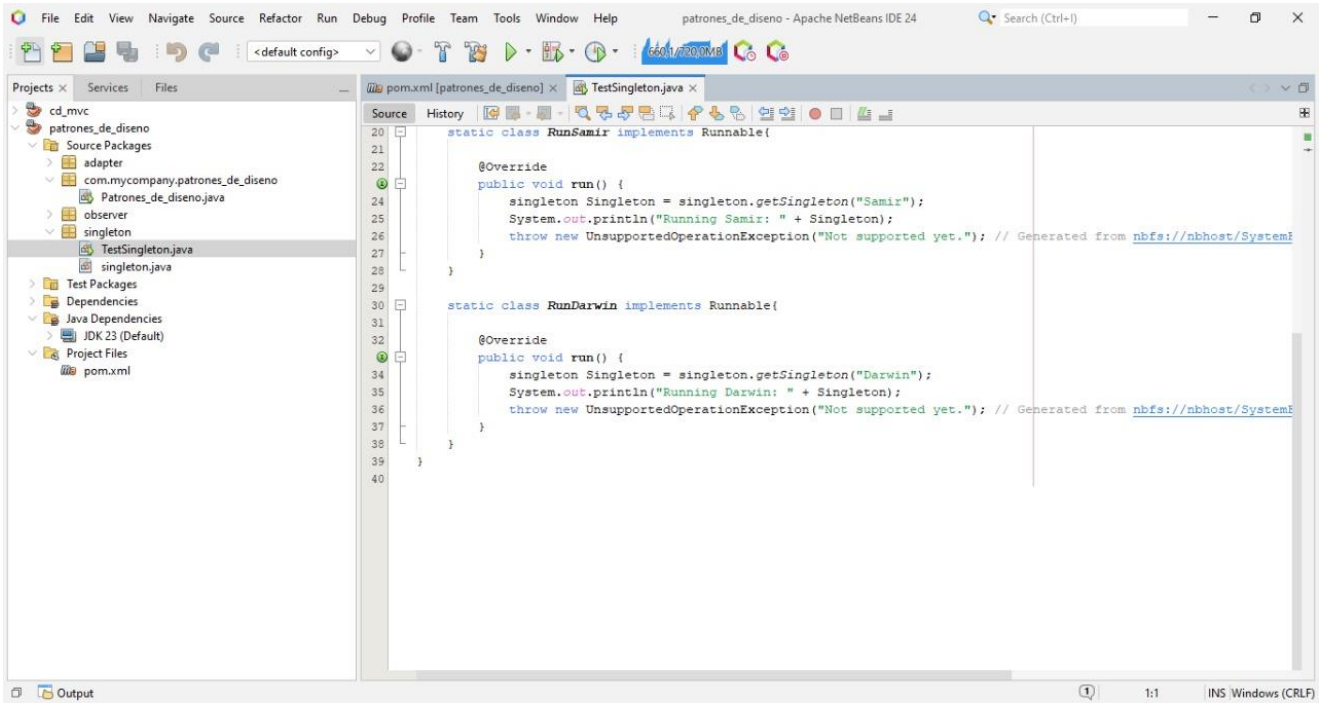


Ilustración 36

- **RunSamir**
  - Sobrescribe el método run().
  - Llama al método singleton.getSingleton("Samir");.
  - Imprime el objeto obtenido.
  - Lanza una excepción UnsupportedOperationException("Not supported yet.").
- **RunDarwin**
  - Similar a RunSamir, pero usa "Darwin" como parámetro en getSingleton().

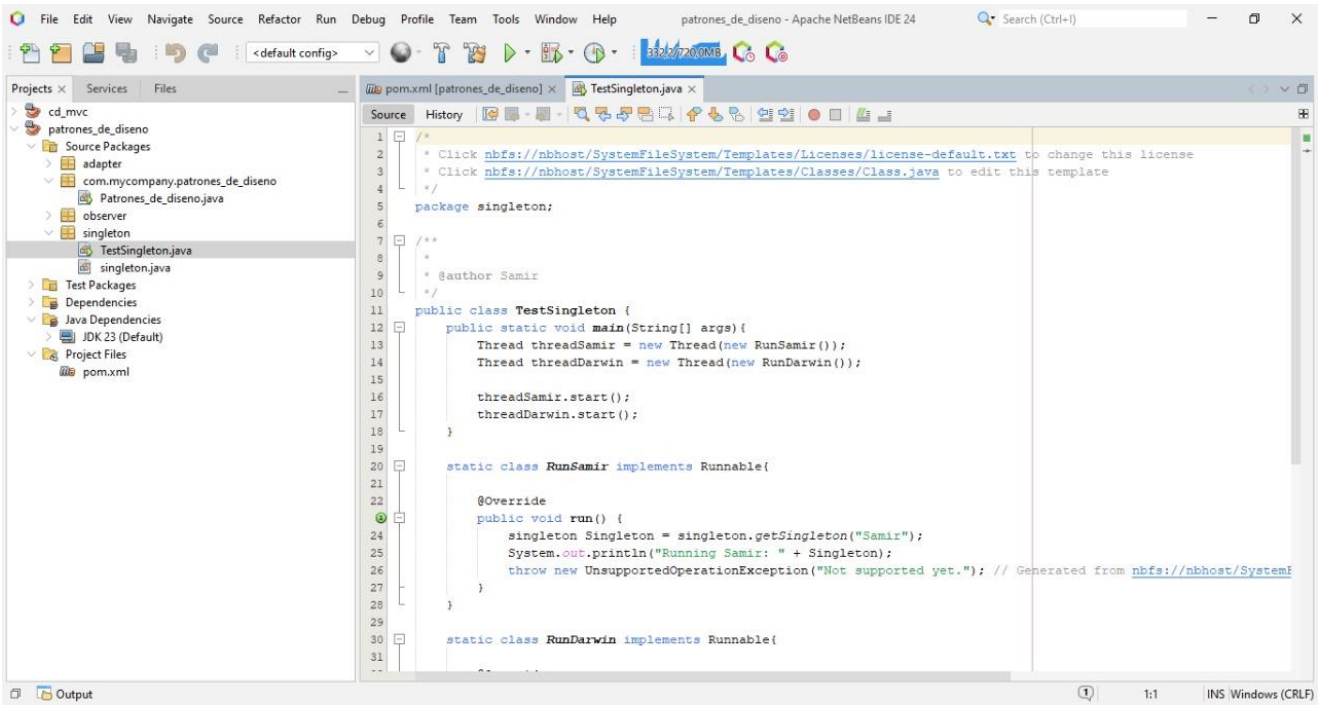


Ilustración 37

Clase TestSingleton:

- Contiene el método main(), donde se crean y ejecutan dos hilos (Thread):
    - threadSamir ejecuta la clase RunSamir.
    - threadDarwin ejecuta la clase RunDarwin.
- Ambos hilos son iniciados con .start(), lo que ejecuta su método run() en paralelo.

Clases internas RunSamir y RunDarwin:

- Ambas implementan Runnable, permitiendo que sean ejecutadas en hilos.
- En el método run():
    - Se intenta obtener una instancia de singleton mediante singleton.getSingleton("Samir") o singleton.getSingleton("Darwin").
    - Se imprime la instancia obtenida con System.out.println("Running Samir: " + Singleton); o "Running Darwin: " + Singleton);
    - Se lanza una excepción UnsupportedOperationException("Not supported yet."), lo que detiene la ejecución.

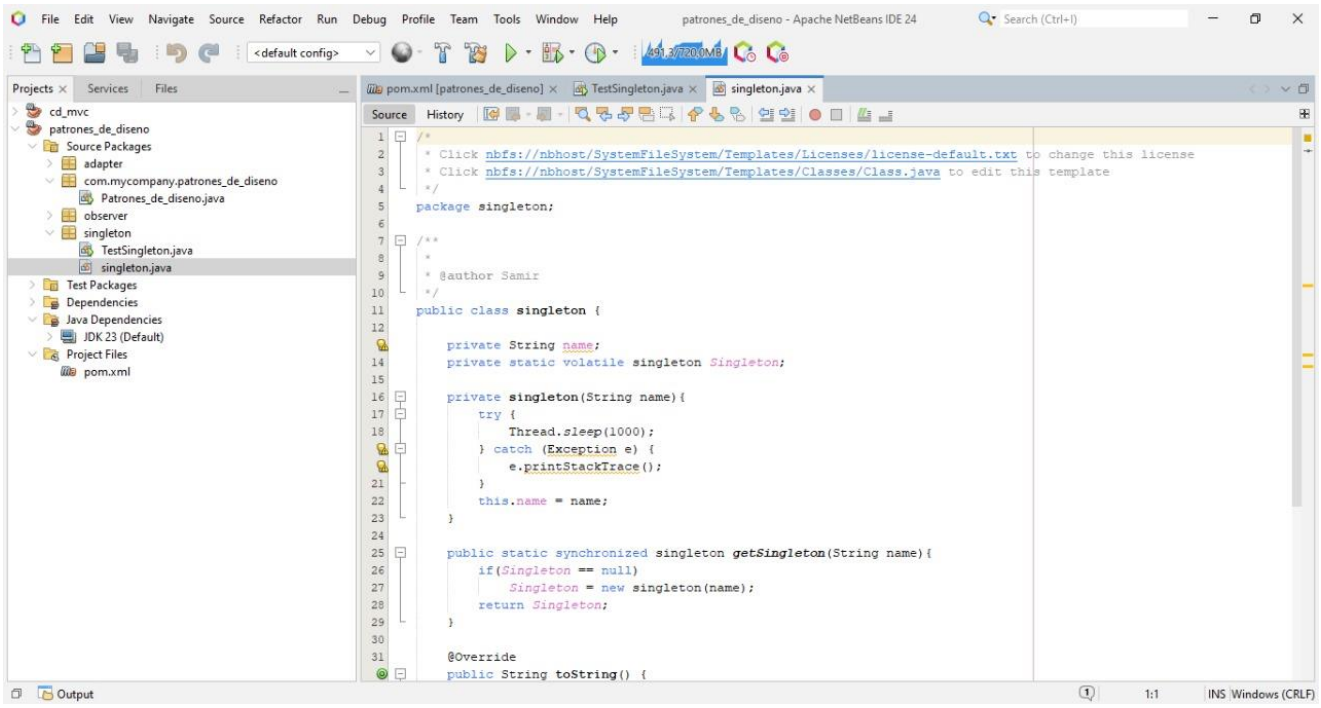


Ilustración 38

Atributos:

- Tiene un nombre (name) y una variable estática que almacena la única instancia de la clase.

Constructor privado:

- Evita que otras clases creen objetos directamente.
- Incluye una pausa de 1 segundo (sleep) para simular un proceso lento.

Método getSingleton(String name):

- Si la instancia no existe, la crea.
- Si ya existe, devuelve la misma instancia.

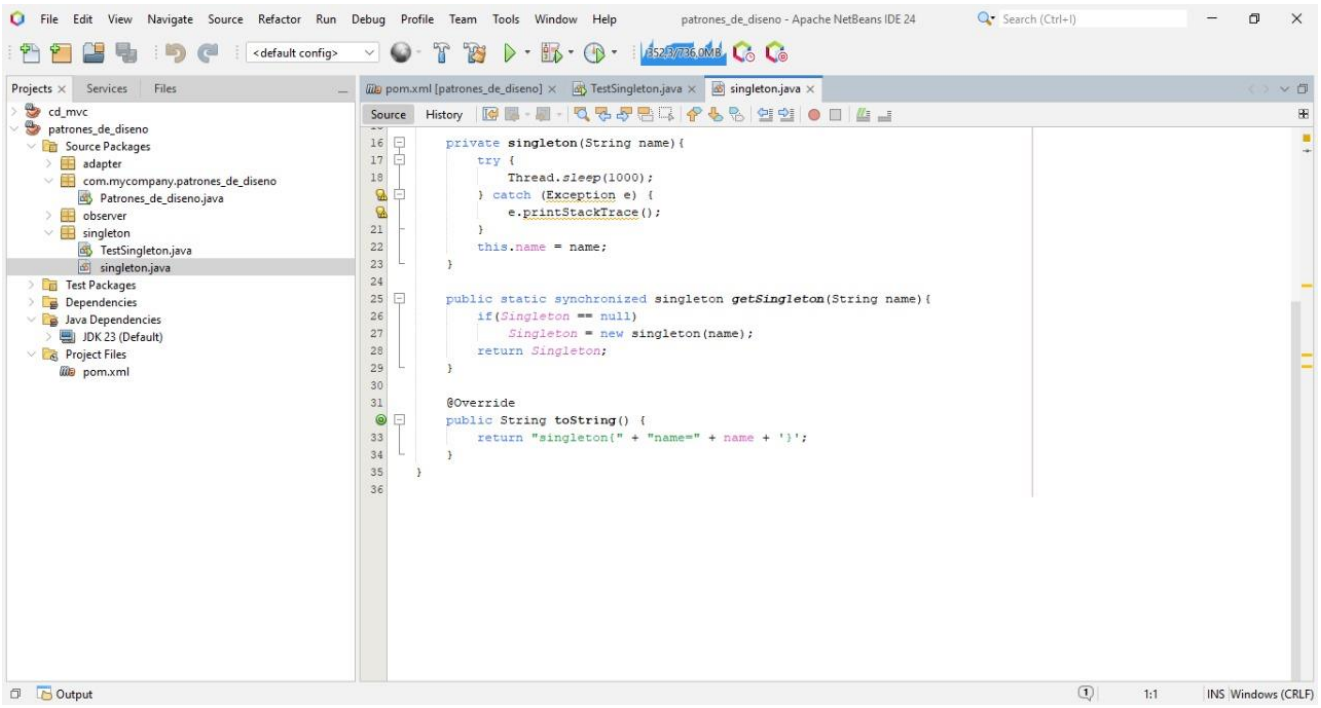


Ilustración 39

Controla la creación de instancias

- La clase tiene un constructor privado para evitar que se creen objetos desde fuera.
- Solo se puede obtener una instancia llamando al método getSingleton(String name).

Sincronización para entornos multihilo

- Usa synchronized en getSingleton() para evitar que múltiples hilos creen instancias simultáneamente.

Simulación de retardo

- En el constructor, hay un Thread.sleep(1000); que introduce una pausa de 1 segundo, probablemente para simular un proceso costoso.

Salida de texto personalizada

- El método toString() devuelve el nombre del Singleton en formato singleton{name=nombre} cuando se imprime.

Ilustración 40

Se crearon dos hilos (Thread-0 y Thread-1).

- **Ambos hilos accedieron a la instancia Singleton**, obteniendo el mismo objeto con el nombre "Samir".

Se imprimieron los mensajes:

- Running Darwin: singleton{name=Samir}
- Running Samir: singleton{name=Samir}
- Se lanzó una excepción UnsupportedOperationException en ambos hilos (RunDarwin y RunSamir).
- El código tiene una línea que lanza manualmente esta excepción.

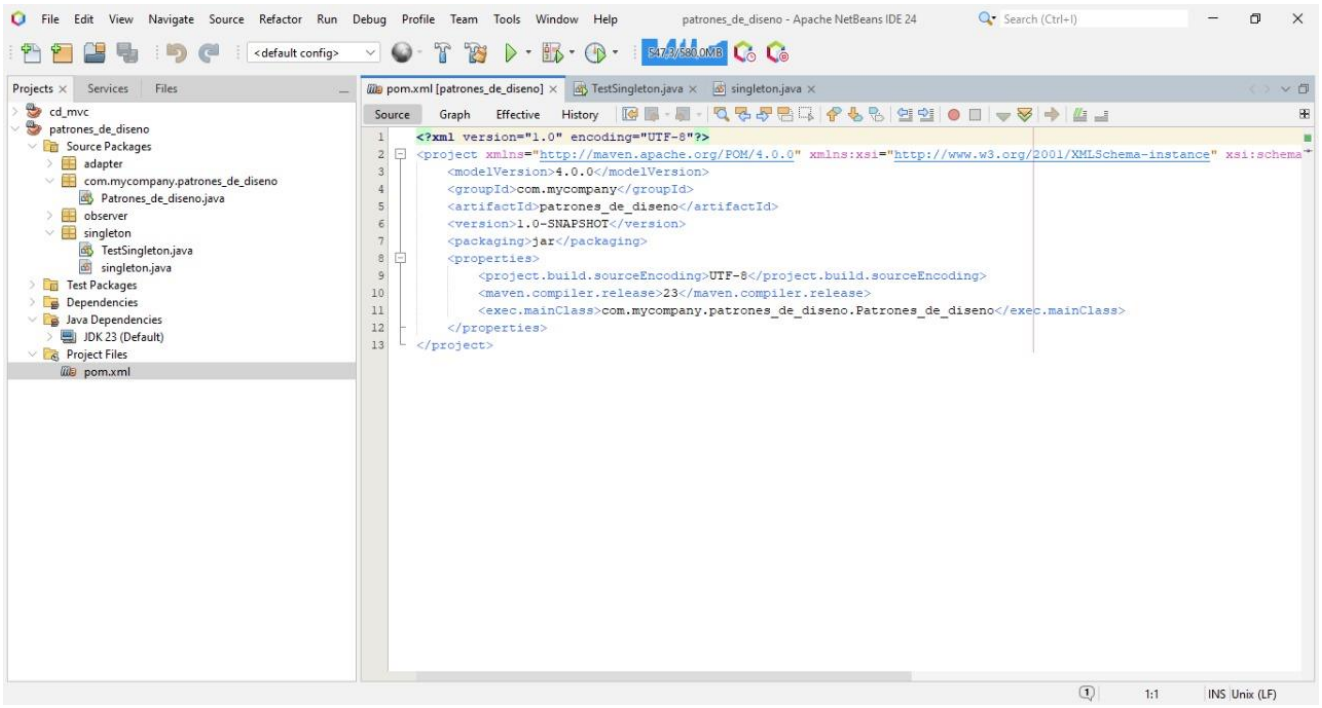


Ilustración 41

- **Definición del proyecto Maven** con:
    - groupId: com.mycompany (Identificador del grupo)
    - artifactId: patrones_de_diseño (Nombre del proyecto)
    - version: 1.0-SNAPSHOT
    - packaging: jar (Se compilará como un archivo .jar)
- **Configuraciones adicionales:**
    - **Codificación en UTF-8**
    - **Uso de Java 23** (maven.compiler.release = 23)
    - **Clase principal (mainClass) definida:**

      com.mycompany.patrones_de_diseno.Patrones_de_diseño

3. CONCLUSIONES

El Modelo Vista Controlador (MVC) es un enfoque clave en la arquitectura de software, ya que permite dividir el sistema en capas independientes, lo que facilita su escalabilidad y mantenimiento a lo largo del tiempo.

Además del MVC, existen otros patrones de diseño como Singleton, Observador y Fábrica, los cuales ayudan a resolver problemas comunes en la programación al mejorar la reutilización del código y la eficiencia de las aplicaciones

La implementación de patrones de diseño adecuados no solo favorece una mejor estructuración del código, sino que también facilita el trabajo en equipo, ya que proporciona una base más clara y comprensible para la colaboración y evolución del software.

4. RECOMENDACIONES

Es recomendable aplicar patrones de diseño como MVC en el desarrollo de software, ya que contribuyen a una mejor organización del código, lo que facilita su mantenimiento y optimiza la estructura del sistema.

Es fundamental estar en constante actualización sobre nuevas metodologías y patrones de diseño, ya que la evolución tecnológica brinda soluciones más eficientes y adecuadas para distintos proyectos de desarrollo.

5. BIBLIOGRAFÍA

Martínez, J. (2020). *Fundamentos del diseño de software*. Alfaomega.

Gómez, L. (2019). *Arquitectura de software avanzada* (2ª ed.). McGraw-Hill. https://www.ejemplo.com/libro

Pérez, M., & López, R. (2021). Desarrollo de software seguro. *Revista de Ingeniería de Software, 35*(2), 45-60. https://doi.org/10.xxxx/ejemplo

Organización Mundial de la Salud. (2022). *Guía sobre salud digital*. OMS. https://www.who.int/es/publications

Smith, J. (2023, marzo 15). La inteligencia artificial en la vida cotidiana. *The New York Times*. https://www.nytimes.com/ai-life

Universidad Nacional Autónoma de México. (2020). *Manual de desarrollo tecnológico*. UNAM Press.

González, C. (2018, septiembre). Métodos de aprendizaje automático. *Conferencia en el Congreso Latinoamericano de Tecnología*, Buenos Aires, Argentina.