

Patrones de Diseño.

Definición:

Los patrones de diseño son soluciones estandarizadas y generales reutilizables para problemas comunes en el diseño de software. Son plantillas o pautas que ayudan a los desarrolladores a crear software sólido y mantenible. Estos patrones no son algoritmos o bloques de código específicos solo para utilizar, sino mejores prácticas y conceptos para resolver problemas de diseño.

¿Por qué utilizarlos?

Los patrones de diseño ofrecen soluciones reutilizables a problemas comunes de diseño de software, lo que ayuda a los desarrolladores a evitar reinventar la rueda. Mejoran la legibilidad y el mantenimiento del código al proporcionar enfoques estandarizados para los desafíos de diseño, lo que fomenta una comunicación más clara entre los miembros del equipo. Al incorporar las mejores prácticas, los patrones de diseño promueven la resolución eficaz de problemas y mejoran la calidad general y la solidez del software.

Categorías o tipos:

1. Patrones creacionales o patrones de creación: se centran en los mecanismos de creación de objetos. Abstraen el proceso de creación de instancias, haciéndolo más flexible y dinámico. Ejemplos: Singleton, Abstract factory, Factory Method, Builder, Prototype.

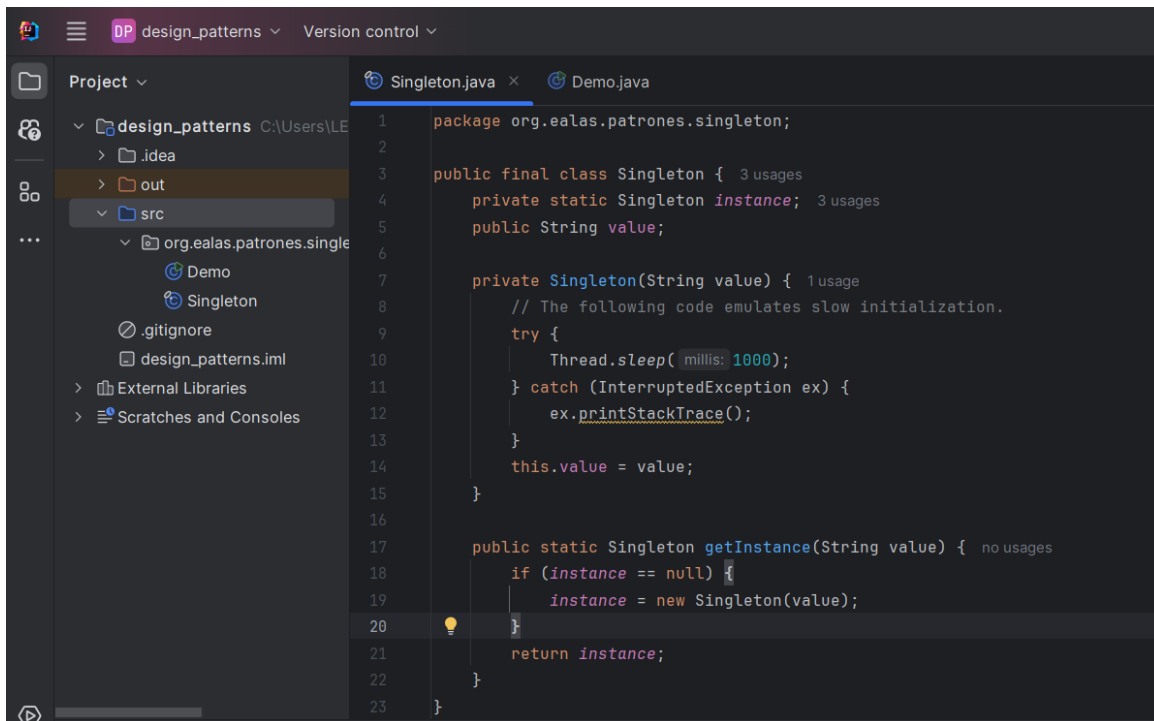
2. Patrones estructurales: se encargan de la composición de los objetos, asegurando que los componentes encajen bien y puedan cambiarse o reemplazarse fácilmente. El concepto de herencia se utiliza para componer interfaces y definir formas de componer objetos para obtener nuevas funcionalidades. Ejemplos: Adapter, Bridge, Composite, Decorator, Facade, Proxy, Flyweight.

3. Patrones de comportamiento: se centran en la comunicación entre objetos y en cómo se distribuyen las responsabilidades entre ellos. Ejemplos: Chain of responsibility, Command, Iterator, Observer, Strategy, Mediator, etc.

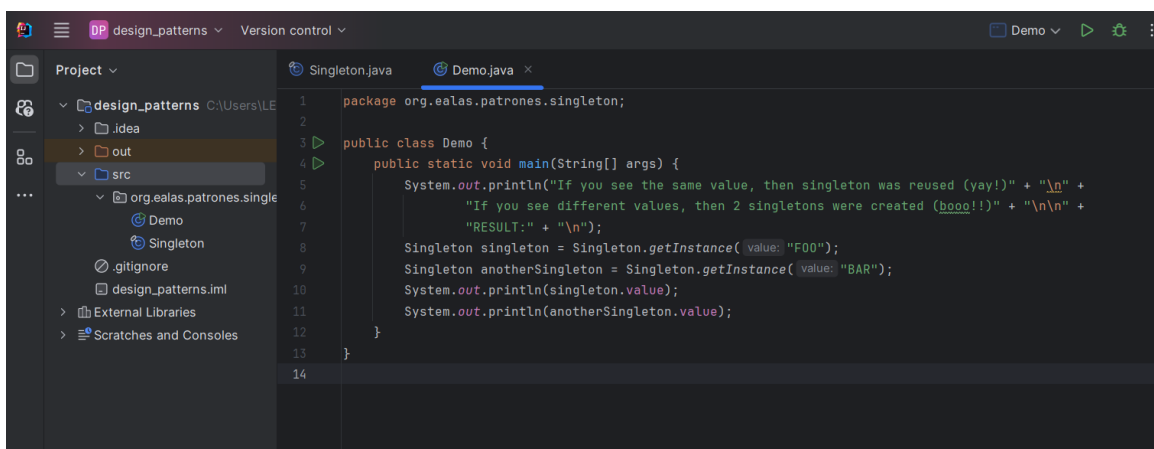
Patrones más utilizados:

1. Patrón Singleton.

Garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella. Comúnmente utilizado para administrar recursos como ajustes de configuración.



```
1 package org.ealas.patrones.singleton;
2
3 public final class Singleton { 3 usages
4     private static Singleton instance; 3 usages
5     public String value;
6
7     private Singleton(String value) { 1 usage
8         // The following code emulates slow initialization.
9         try {
10             Thread.sleep( millis: 1000);
11         } catch (InterruptedException ex) {
12             ex.printStackTrace();
13         }
14         this.value = value;
15     }
16
17     public static Singleton getInstance(String value) { no usages
18         if (instance == null) {
19             instance = new Singleton(value);
20         }
21         return instance;
22     }
23 }
```



```
1 package org.ealas.patrones.singleton;
2
3 public class Demo {
4     public static void main(String[] args) {
5         System.out.println("If you see the same value, then singleton was reused (yay!) " + "\n" +
6             "If you see different values, then 2 singletons were created (boo!!)" + "\n\n" +
7             "RESULT: " + "\n");
8         Singleton singleton = Singleton.getInstance( value: "FOO");
9         Singleton anotherSingleton = Singleton.getInstance( value: "BAR");
10        System.out.println(singleton.value);
11        System.out.println(anotherSingleton.value);
12    }
13 }
14
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (boooo!!)

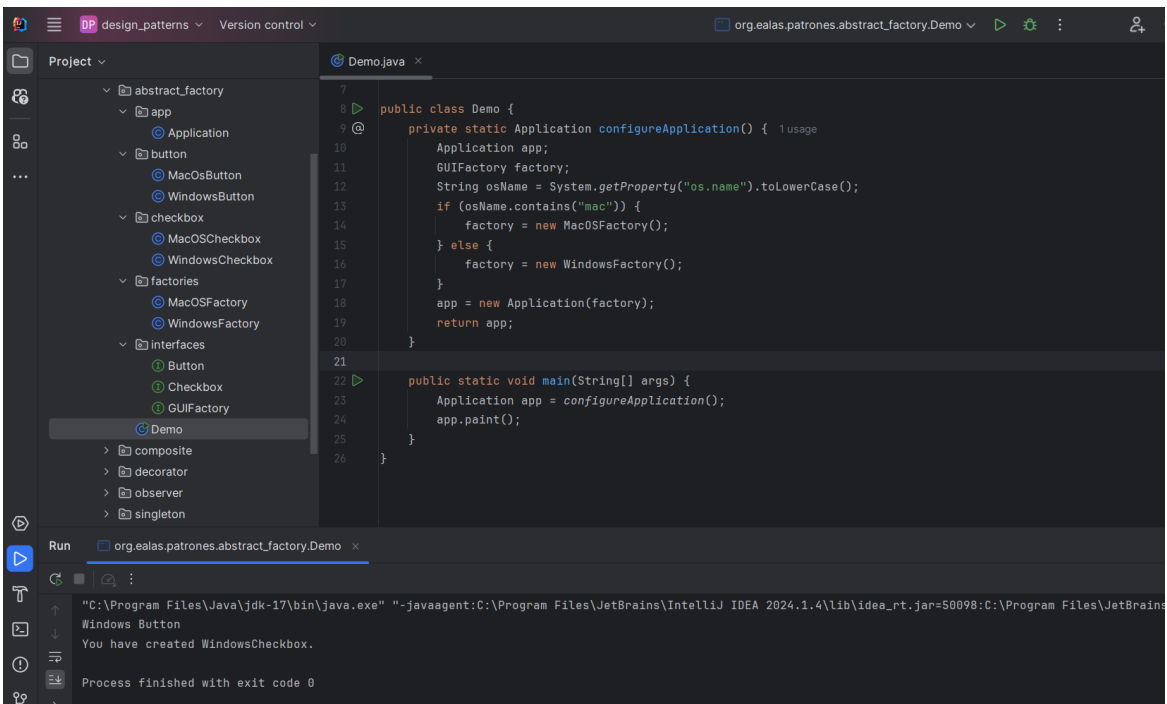
RESULT:

F00
F00

Process finished with exit code 0
```

2. Patrón Abstract Factory.

Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.



```
public class Demo {
    private static Application configureApplication() { 1 usage
        Application app;
        GUIFactory factory;
        String osName = System.getProperty("os.name").toLowerCase();
        if (osName.contains("mac")) {
            factory = new MacOsFactory();
        } else {
            factory = new WindowsFactory();
        }
        app = new Application(factory);
        return app;
    }

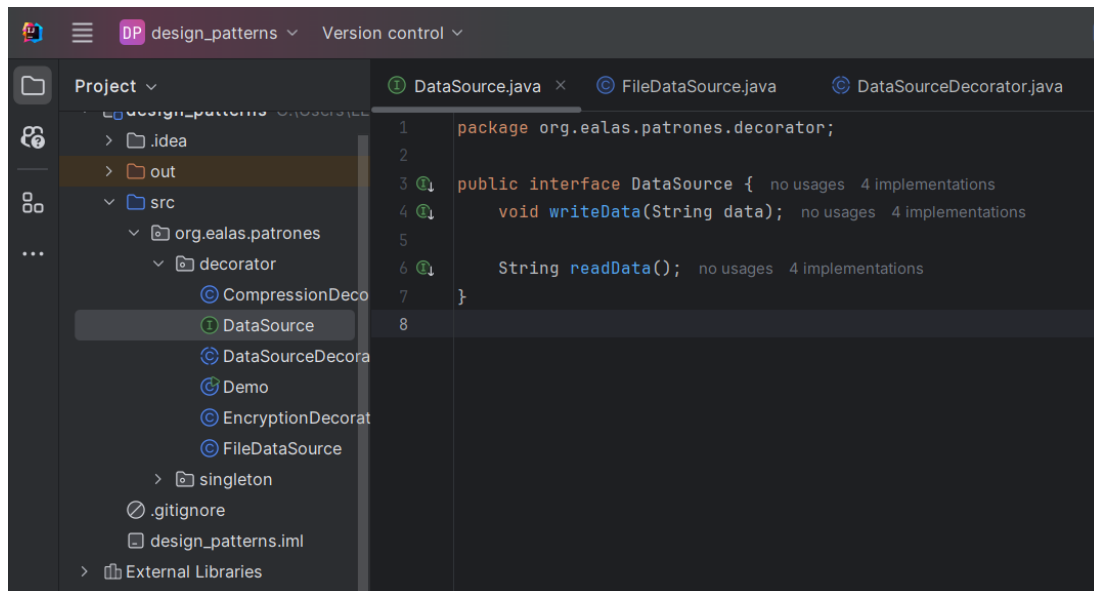
    public static void main(String[] args) {
        Application app = configureApplication();
        app.paint();
    }
}
```

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=50098:C:\Program Files\JetBrains
Windows Button
You have created WindowsCheckbox.

Process finished with exit code 0
```

3. Patrón Decorator.

Permite agregar nuevos comportamientos a los objetos de forma dinámica colocándolos dentro de objetos envoltentes especiales, llamados decoradores. Comúnmente se usa para ampliar funcionalidades de una manera flexible y reutilizable, como agregar funciones a los componentes de la interfaz de usuario.



```
5 public class FileDataSource implements DataSource { no usages
6     private String name; 3 usages
7
8     public FileDataSource(String name) { no usages
9         this.name = name;
10    }
11
12    @Override no usages
13    public void writeData(String data) {
14        File file = new File(name);
15        try (OutputStream fos = new FileOutputStream(file)) {
16            fos.write(data.getBytes(), off: 0, data.length());
17        } catch (IOException ex) {
18            System.out.println(ex.getMessage());
19        }
20    }
21
22    @Override no usages
23    public String readData() {
24        char[] buffer = null;
25        File file = new File(name);
26        try (FileReader reader = new FileReader(file)) {
27            buffer = new char[(int) file.length()];
28            reader.read(buffer);
29        } catch (IOException ex) {
30            System.out.println(ex.getMessage());
31        }
32        return new String(buffer);
33    }
34 }
```

Run org.ealas.patrones.decorator.Demo

design_patterns > src > org > ealas > patrones > decorator > FileDataSource > readData

```
1 package org.ealas.patrones.decorator;
2
3 public abstract class DataSourceDecorator implements DataSource { no usages 2 inheritors
4     private DataSource wrappee; 3 usages
5
6     DataSourceDecorator(DataSource source) { no usages
7         this.wrappee = source;
8     }
9
10    @Override 1 usage 2 overrides
11    public void writeData(String data) {
12        wrappee.writeData(data);
13    }
14
15    @Override 1 usage 2 overrides
16    public String readData() {
17        return wrappee.readData();
18    }
19 }
```

design_patterns > src > org > ealas > patrones > decorator > DataSourceDecorator > writeData

The screenshot shows the IntelliJ IDEA IDE with the 'design_patterns' project open. The 'src' directory is expanded, showing the 'org.ealas.patrones.decorator' package. The 'EncryptionDecorator' class is selected in the package explorer. The main editor displays the implementation of the 'EncryptionDecorator' class, which extends 'DataSourceDecorator'. The code includes methods for 'writeData', 'readData', 'encode', and 'decode', all utilizing 'Base64' for data transformation. The 'run' button at the bottom indicates the current run configuration is 'org.ealas.patrones.decorator.Demo'.

```
5 public class EncryptionDecorator extends DataSourceDecorator { no usages
7     public EncryptionDecorator(DataSource source) { no usages
8         super(source);
9     }
10
11     @Override 2 usages
12     public void writeData(String data) {
13         super.writeData(encode(data));
14     }
15
16     @Override 2 usages
17     public String readData() {
18         return decode(super.readData());
19     }
20
21     private String encode(String data) { 1 usage
22         byte[] result = data.getBytes();
23         for (int i = 0; i < result.length; i++) {
24             result[i] += (byte) 1;
25         }
26         return Base64.getEncoder().encodeToString(result);
27     }
28
29     private String decode(String data) { 1 usage
30         byte[] result = Base64.getDecoder().decode(data);
31         for (int i = 0; i < result.length; i++) {
32             result[i] -= (byte) 1;
33         }
34         return new String(result);
35     }
}
```

The screenshot shows the IntelliJ IDEA IDE with the 'design_patterns' project open. The 'src' directory is expanded, showing the 'org.ealas.patrones.decorator' package. The 'CompressionDecorator' class is selected in the package explorer. The main editor displays the implementation of the 'CompressionDecorator' class, which extends 'DataSourceDecorator'. The code includes imports for various Java I/O and utility classes, and methods for 'getCompressionLevel', 'setCompressionLevel', and 'writeData'. The 'run' button at the bottom indicates the current run configuration is 'org.ealas.patrones.decorator.Demo'.

```
1 package org.ealas.patrones.decorator;
2
3 import java.io.ByteArrayInputStream;
4 import java.io.ByteArrayOutputStream;
5 import java.io.IOException;
6 import java.io.InputStream;
7 import java.util.Base64;
8 import java.util.zip.Deflater;
9 import java.util.zip.DeflaterOutputStream;
10 import java.util.zip.InflaterInputStream;
11
12 public class CompressionDecorator extends DataSourceDecorator { no usages
13     private int compLevel = 6; 3 usages
14
15     public CompressionDecorator(DataSource source) { no usages
16         super(source);
17     }
18
19     public int getCompressionLevel() { no usages
20         return compLevel;
21     }
22
23     public void setCompressionLevel(int value) { no usages
24         compLevel = value;
25     }
26
27     @Override 3 usages
28     public void writeData(String data) {
29         super.writeData(compress(data));
30     }
}
```

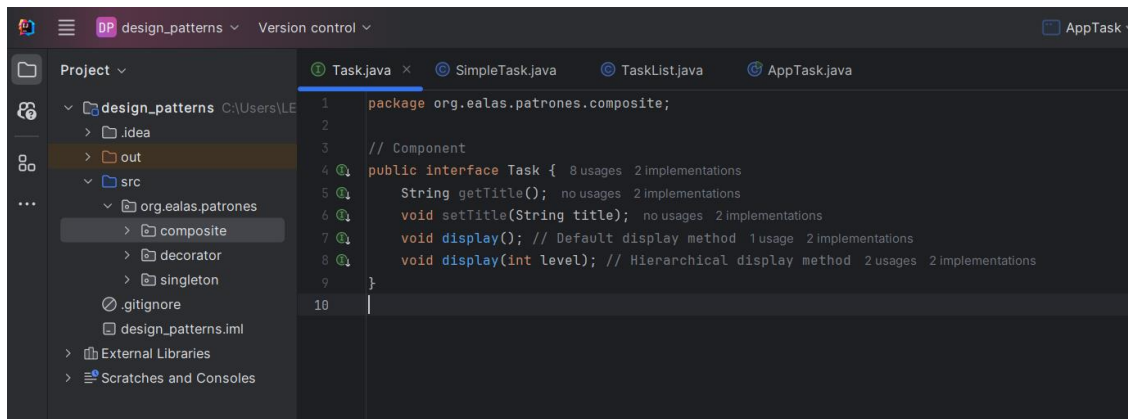
```
12 public class CompressionDecorator extends DataSourceDecorator {
13     @Override
14     public String readData() {
15         return decompress(super.readData());
16     }
17
18     private String compress(String stringData) {
19         byte[] data = stringData.getBytes();
20         try {
21             ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
22             DeflaterOutputStream dos = new DeflaterOutputStream(bout, new Deflater(comLevel));
23             dos.write(data);
24             dos.close();
25             bout.close();
26             return Base64.getEncoder().encodeToString(bout.toByteArray());
27         } catch (IOException ex) {
28             return null;
29         }
30     }
31
32     private String decompress(String stringData) {
33         byte[] data = Base64.getDecoder().decode(stringData);
34         try {
35             InputStream in = new ByteArrayInputStream(data);
36             InflaterInputStream iin = new InflaterInputStream(in);
37             ByteArrayOutputStream bout = new ByteArrayOutputStream(512);
38             int b;
39             while ((b = iin.read()) != -1) {
40                 bout.write(b);
41             }
42         }
43     }
44 }
```

```
1 package org.ealas.patrones.decorator;
2
3 public class Demo {
4     public static void main(String[] args) {
5         String salaryRecords = "Name,Salary\nJohn Smith,100000\nSteven Jobs,912000";
6         DataSourceDecorator encoded = new CompressionDecorator(
7             new EncryptionDecorator(
8                 new FileDataSource( "name: \"out/OutputDemo.txt\""));
9         encoded.writeData(salaryRecords);
10        DataSource plain = new FileDataSource( "name: \"out/OutputDemo.txt\"");
11
12        System.out.println("- Input -----");
13        System.out.println(salaryRecords);
14        System.out.println("- Encoded -----");
15        System.out.println(plain.readData());
16        System.out.println("- Decoded -----");
17        System.out.println(encoded.readData());
18    }
19 }
20 }
```

```
Run org.ealas.patrones.decorator.Demo
C:\Program Files\Java\jdk-17\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1.4\lib\idea_rt.jar=64517:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1
- Input -----
Name,Salary
John Smith,100000
Steven Jobs,912000
- Encoded -----
Zkt7e1Q5eU8yUm1qe0ZsdHJ2VXp6dDBKYNhrUhtUe0sxRUYxQkJIjdJyLTVZDdVlSQ2IwOXFISmVUMU5rcENCQmdxRLByaD4+
- Decoded -----
Name,Salary
John Smith,100000
Steven Jobs,912000
Process finished with exit code 0
```

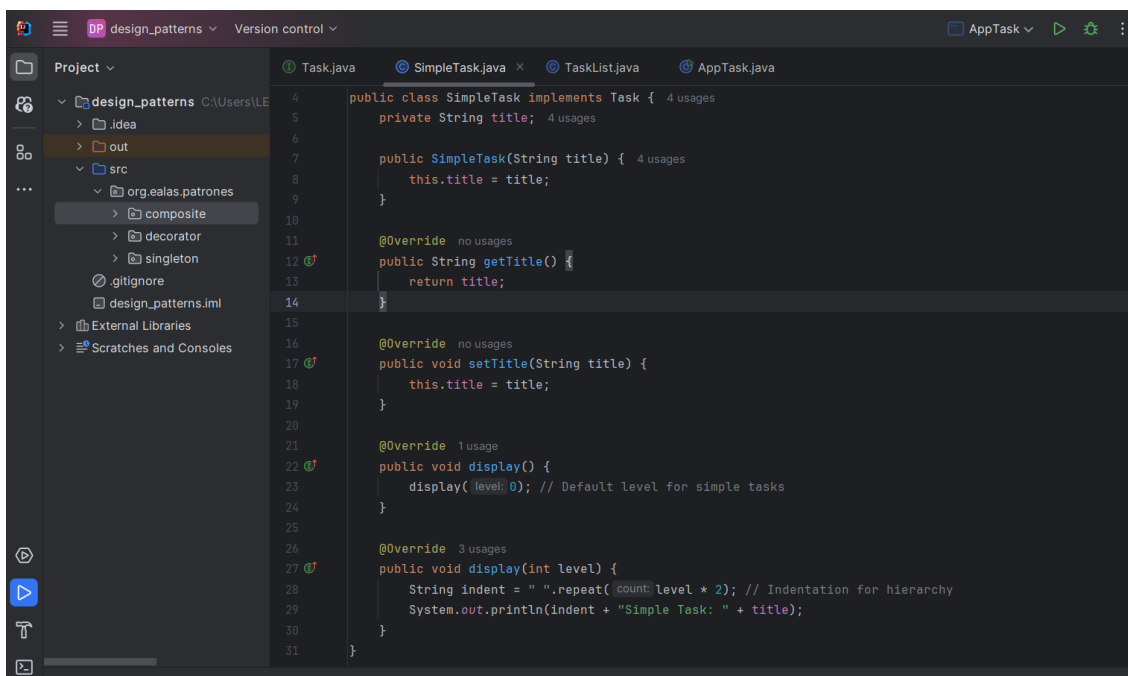
4. Patrón Composite.

Permite componer objetos en estructuras de árbol y luego trabajar con estas estructuras como si fueran objetos individuales.



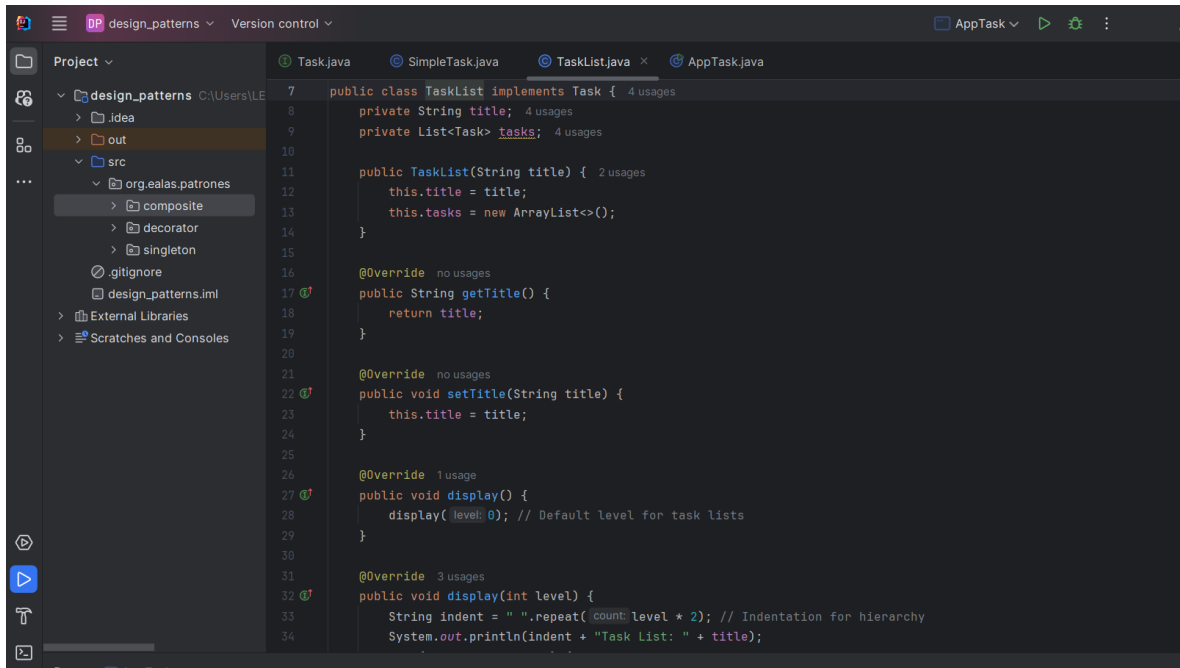
The screenshot shows the IntelliJ IDEA interface with the 'design_patterns' project open. The 'Project' view on the left shows the package structure: 'org.ealas.patrones' containing 'composite', 'decorator', and 'singleton'. The 'Task.java' file is open in the editor, showing the following code:

```
1 package org.ealas.patrones.composite;
2
3 // Component
4 public interface Task {
5     String getTitle();
6     void setTitle(String title);
7     void display(); // Default display method
8     void display(int level); // Hierarchical display method
9 }
10
```

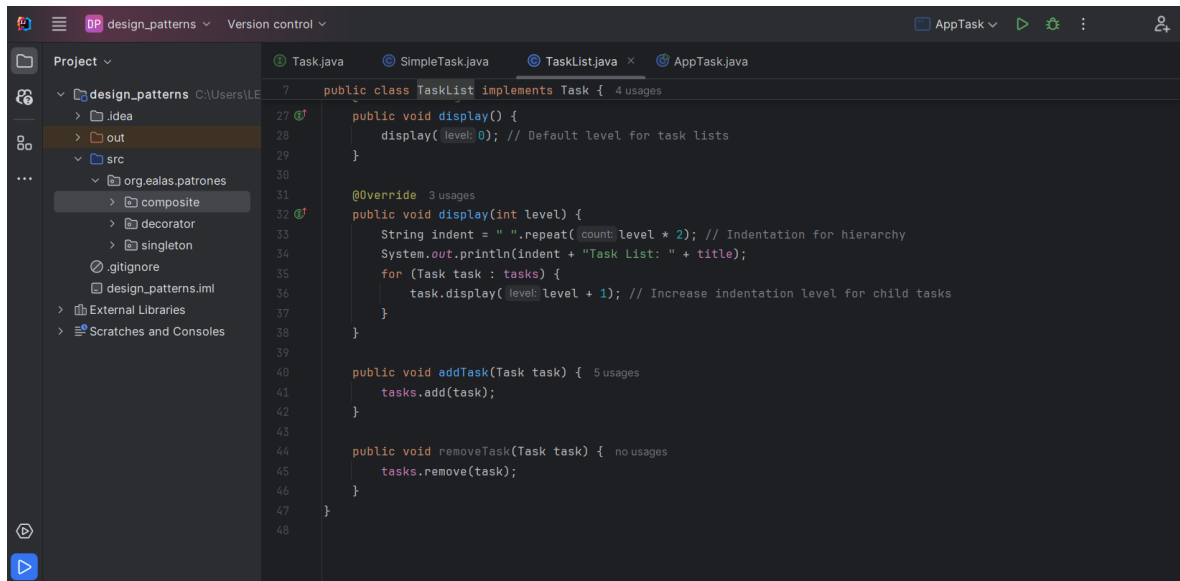


The screenshot shows the IntelliJ IDEA interface with the 'design_patterns' project open. The 'Project' view on the left shows the package structure: 'org.ealas.patrones' containing 'composite', 'decorator', and 'singleton'. The 'SimpleTask.java' file is open in the editor, showing the following code:

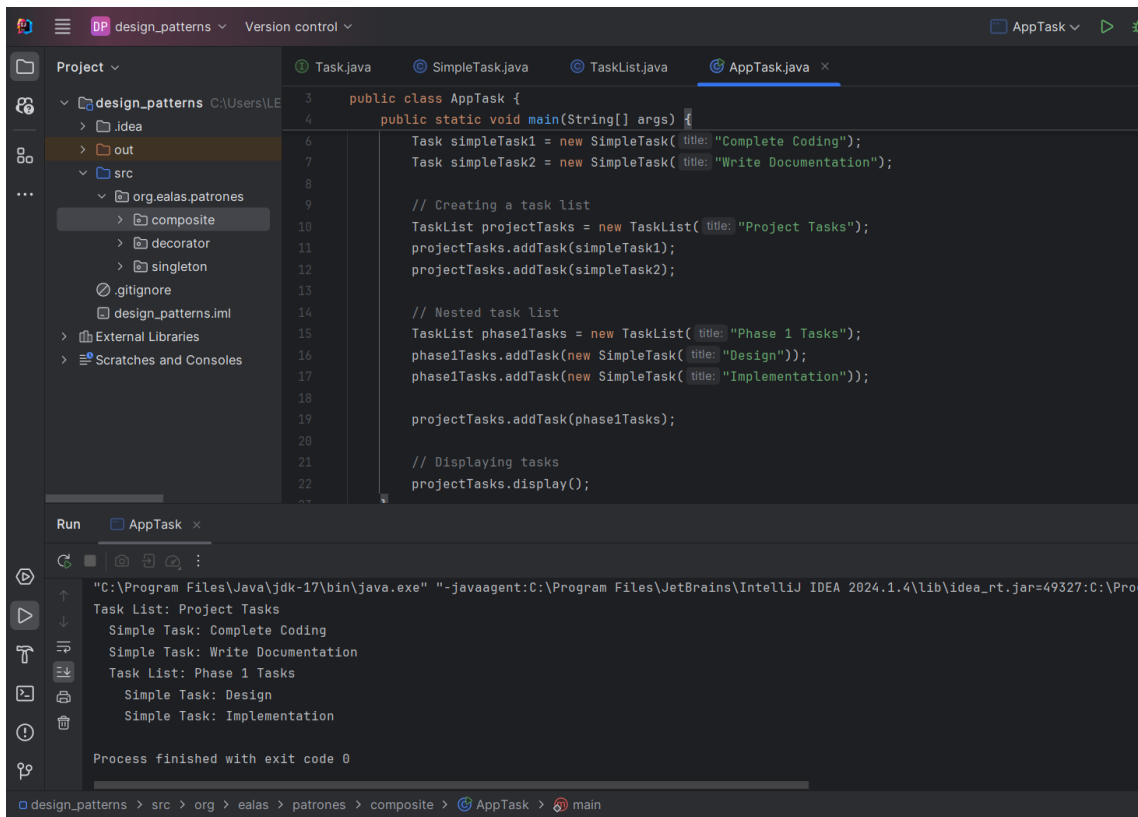
```
4 public class SimpleTask implements Task {
5     private String title;
6
7     public SimpleTask(String title) {
8         this.title = title;
9     }
10
11     @Override
12     public String getTitle() {
13         return title;
14     }
15
16     @Override
17     public void setTitle(String title) {
18         this.title = title;
19     }
20
21     @Override
22     public void display() {
23         display(level: 0); // Default level for simple tasks
24     }
25
26     @Override
27     public void display(int level) {
28         String indent = " ".repeat(count: level * 2); // Indentation for hierarchy
29         System.out.println(indent + "Simple Task: " + title);
30     }
31 }
```

```
7 public class TaskList implements Task { 4 usages
8     private String title; 4 usages
9     private List<Task> tasks; 4 usages
10
11     public TaskList(String title) { 2 usages
12         this.title = title;
13         this.tasks = new ArrayList<>();
14     }
15
16     @Override no usages
17     public String getTitle() {
18         return title;
19     }
20
21     @Override no usages
22     public void setTitle(String title) {
23         this.title = title;
24     }
25
26     @Override 1 usage
27     public void display() {
28         display(level: 0); // Default level for task lists
29     }
30
31     @Override 3 usages
32     public void display(int level) {
33         String indent = " ".repeat(count: level * 2); // Indentation for hierarchy
34         System.out.println(indent + "Task List: " + title);
35     }
36 }
```

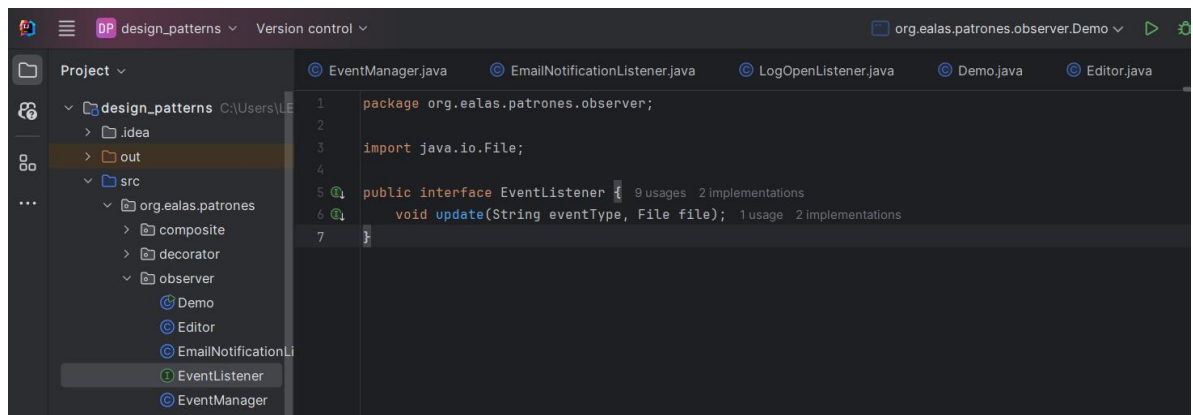


```
7 public class TaskList implements Task { 4 usages
8
9     public void display() {
10         display(level: 0); // Default level for task lists
11     }
12
13     @Override 3 usages
14     public void display(int level) {
15         String indent = " ".repeat(count: level * 2); // Indentation for hierarchy
16         System.out.println(indent + "Task List: " + title);
17         for (Task task : tasks) {
18             task.display(level: level + 1); // Increase indentation level for child tasks
19         }
20     }
21
22     public void addTask(Task task) { 5 usages
23         tasks.add(task);
24     }
25
26     public void removeTask(Task task) { no usages
27         tasks.remove(task);
28     }
29 }
30 }
```



5. Patrón Observer.

Observer es un patrón de diseño de comportamiento que permite que algunos objetos notifiquen a otros objetos sobre cambios en su estado. Se utiliza comúnmente en sistemas controlados por eventos y en la implementación de modelos de publicador-suscriptor.



This screenshot shows the IntelliJ IDEA IDE with the 'EmailNotificationListener.java' file open. The project structure on the left includes 'design_patterns' with subfolders like 'src' and 'observer'. The code in the editor is as follows:

```
1 package org.ealas.patrones.observer;
2
3 import java.io.File;
4
5 public class EmailNotificationListener implements EventListener { 1 usage
6     private String email; 2 usages
7
8     public EmailNotificationListener(String email) 1 usage
9         this.email = email;
10
11
12     @Override 1 usage
13     public void update(String eventType, File file) {
14         System.out.println("Email to " + email + ": Someone has performed " + eventType
15             + " operation with the following file: " + file.getName());
16     }
17 }
```

This screenshot shows the IntelliJ IDEA IDE with the 'EventManager.java' file open. The project structure on the left is the same as the previous screenshot. The code in the editor is as follows:

```
4 import java.util.*;
5
6 public class EventManager { no usages
7     Map<String, List<EventListener>> listeners = new HashMap<>(); 4 usages
8
9     @
10     public EventManager(String... operations) { no usages
11         for (String operation : operations) {
12             this.listeners.put(operation, new ArrayList<>());
13         }
14
15     public void subscribe(String eventType, EventListener listener) { no usages
16         List<EventListener> users = listeners.get(eventType);
17         users.add(listener);
18     }
19
20     public void unsubscribe(String eventType, EventListener listener) { no usages
21         List<EventListener> users = listeners.get(eventType);
22         users.remove(listener);
23     }
24
25     public void notify(String eventType, File file) 1 usage
26         List<EventListener> users = listeners.get(eventType);
27         for (EventListener listener : users) {
28             listener.update(eventType, file);
29         }
30
31 }
```

This screenshot shows the IntelliJ IDEA IDE with the 'LogOpenListener.java' file open. The project structure on the left is the same as the previous screenshots. The code in the editor is as follows:

```
1 package org.ealas.patrones.observer;
2
3 import java.io.File;
4
5 public class LogOpenListener implements EventListener{ 1 usage
6
7     private File log; 2 usages
8
9     public LogOpenListener(String fileName) { 1 usage
10         this.log = new File(fileName);
11     }
12
13     @Override 1 usage
14     public void update(String eventType, File file) {
15         System.out.println("Save to log " + log + ": Someone has performed " + eventType
16             + " operation with the following file: " + file.getName());
17     }
18
19 }
```

