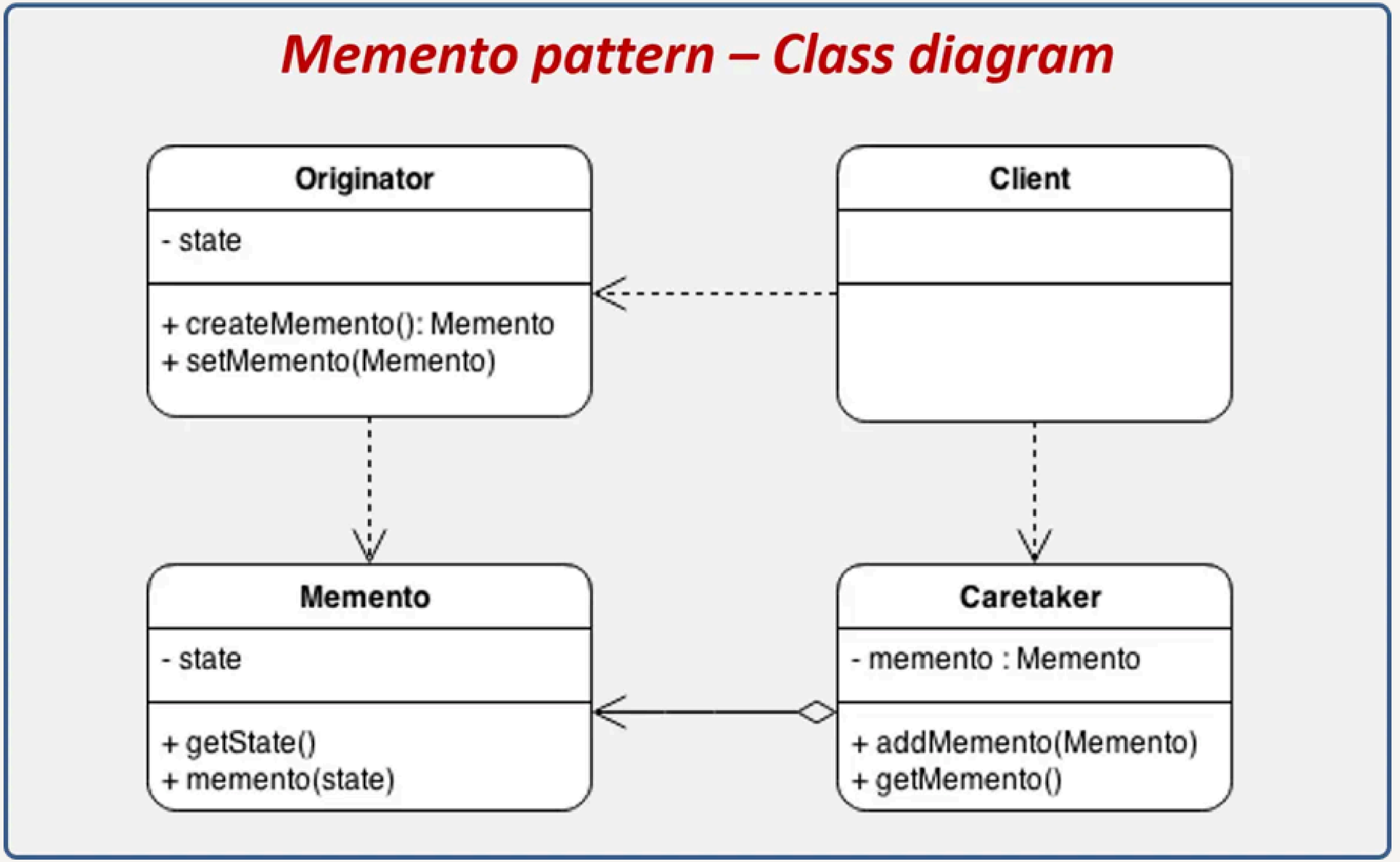
Fredd Badilla Víquez

DISEÑO DE SOFTWARE

PATRONES DE DISEÑO

MEMENTO

Memento pattern – Class diagram



[1] [2] [3]

[”Hola”, “Hola mundo”, “mundo mundo”]

Nueva ---> “mundo”

```java
// Clase Editor que tiene un contenido y puede guardar/restaurar estados
class Editor {
    private String content = "";

    public void type(String words) {
        content += words;
    }

    public String getContent() {
        return content;
    }

    public EditorState save() {
        return new EditorState(content);
    }

    public void restore(EditorState state) {
        content = state.getContent();
    }
}
```

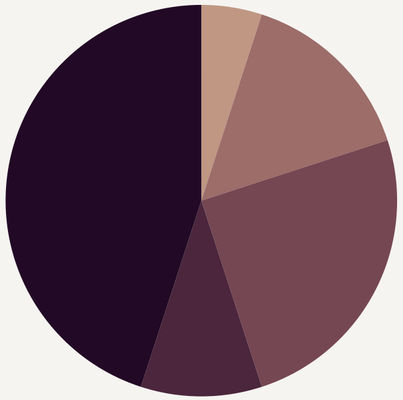
```java
// Clase que representa un estado guardado del editor
class EditorState {
    private final String content;

    public EditorState(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

```java
// Clase que mantiene un historial de estados del editor
class History {
    private final List<EditorState> states = new ArrayList<>();

    public void push(EditorState state) {
        states.add(state);
    }

    public EditorState undo() {
        int lastIndex = states.size() - 1;
        EditorState lastState = states.get(lastIndex);
        return lastState;
    }

    public EditorState redo() {
        int las
        EditorS
        return lastState;
    }
}
```

```
EditorState lastState - History.redo()
```

```java
// Uso del patrón Memento
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Editor editor = new Editor();
        History history = new History();

        editor.type(words:"Este es el primer contenido. ");
        history.push(editor.save());

        editor.type(words:"Este es el segundo contenido. ");
        history.push(editor.save());

        editor.type(words:"Este es el tercer contenido. ");
        System.out.println("Contenido actual: " + editor.getContent());

        // Restaurar a una versión anterior
        editor.restore(history.undo());
        System.out.println("Contenido restaurado: " + editor.getContent());

        editor.restore(history.redo());
        System.out.println("Contenido restaurado: " + editor.getContent());
    }
}
```

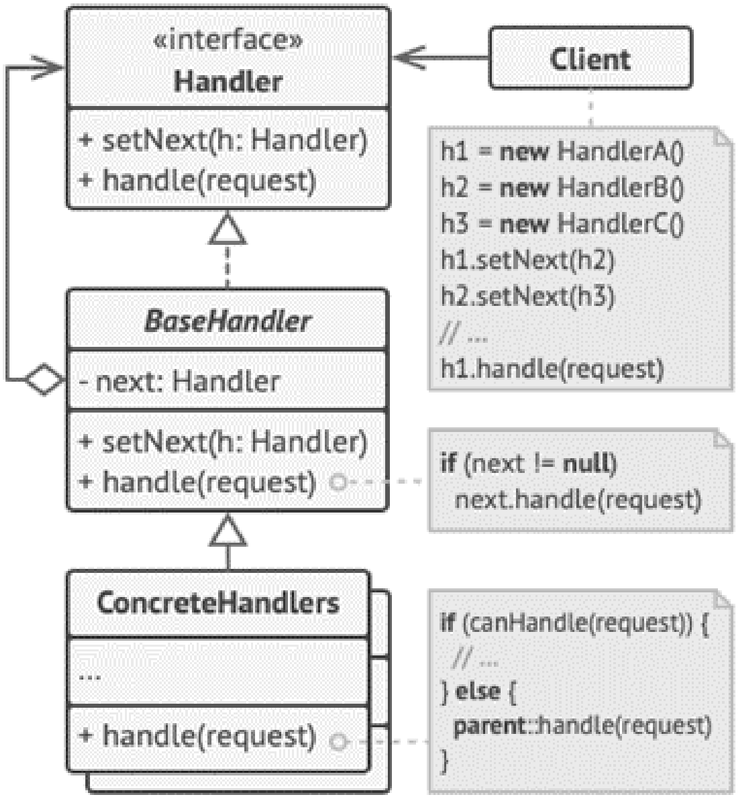
CHAIN OF RESPONSABILITY

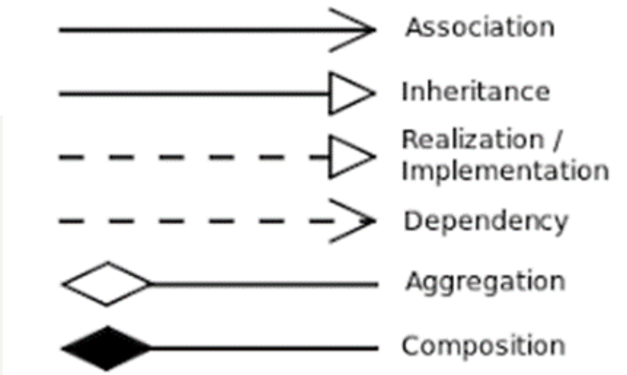


1 La clase **Manejadora** declara la interfaz común a todos los manejadores concretos. Normalmente contiene un único método para manejar solicitudes, pero en ocasiones también puede contar con otro método para establecer el siguiente manejador de la cadena.

2 La clase **Manejadora Base** es opcional y es donde puedes colocar el código boilerplate (segmentos de código que suelen no alterarse) común para todas las clases manejadoras.

Normalmente, esta clase define un campo para almacenar una referencia al siguiente manejador. Los clientes pueden crear una cadena pasando un manejador al constructor o modificador (*setter*) del manejador previo. La clase también puede implementar el comportamiento de gestión por defecto: puede pasar la ejecución al siguiente manejador después de comprobar su existencia.

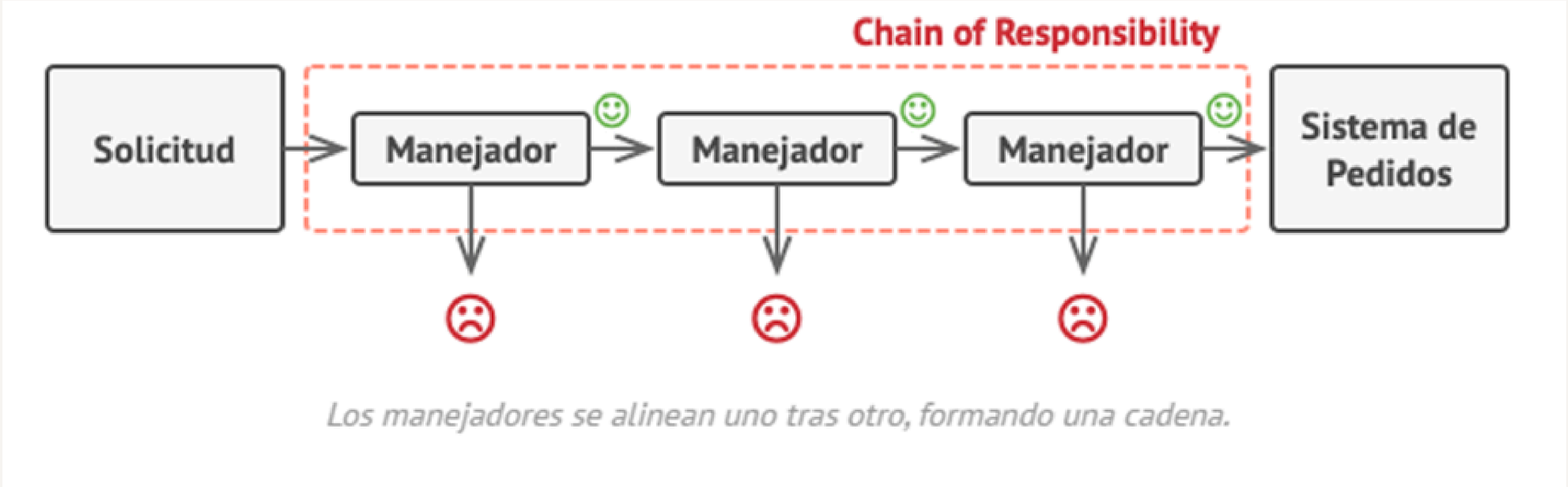




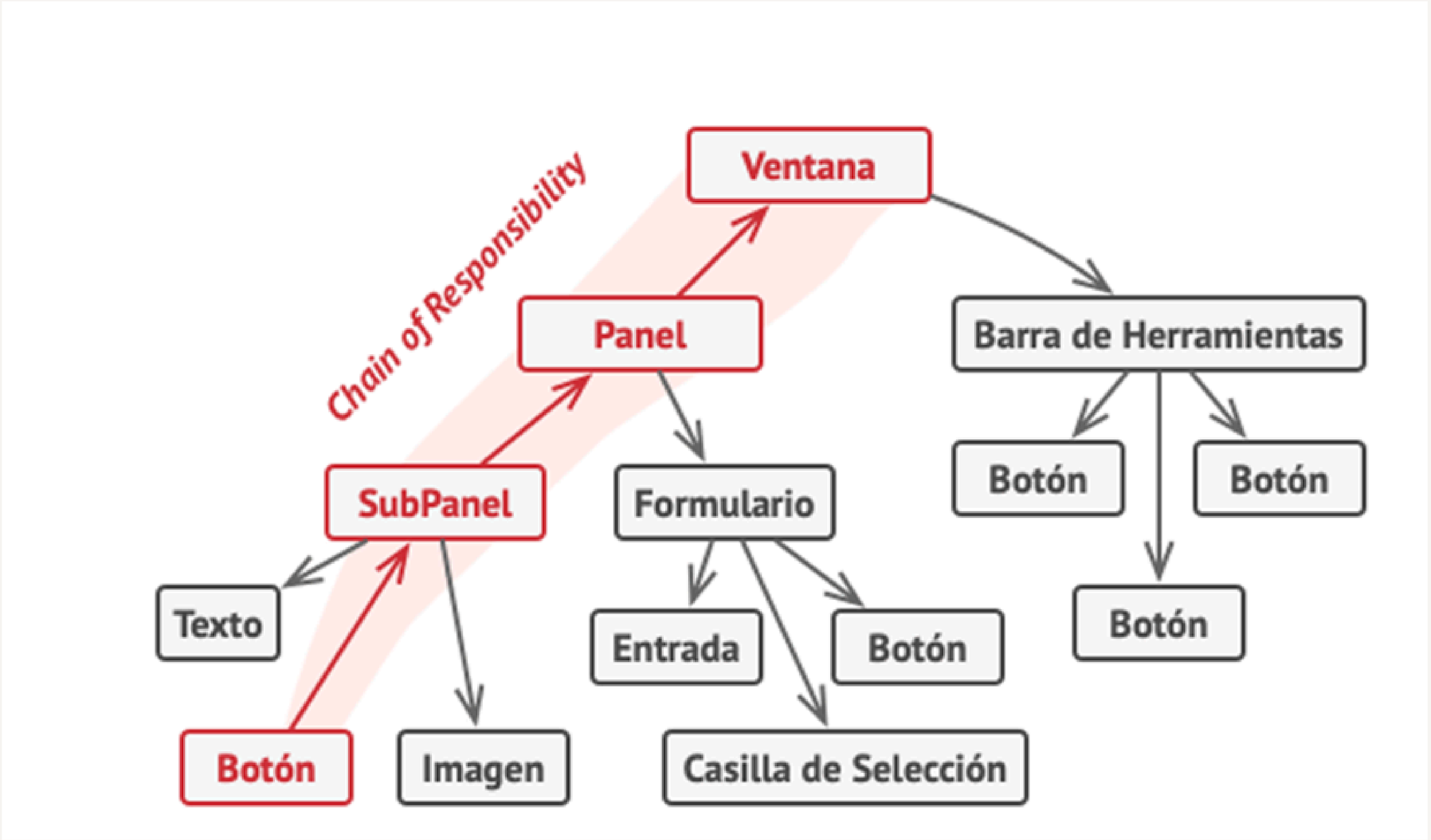
4 El **Cliente** puede componer cadenas una sola vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación. Observa que se puede enviar una solicitud a cualquier manejador de la cadena; no tiene por qué ser al primero.

3 Los **Manejadores Concretos** contienen el código para procesar las solicitudes. Al recibir una solicitud, cada manejador debe decidir si procesarla y, además, si la pasa a lo largo de la cadena.

Habitualmente los manejadores son autónomos e inmutables, y aceptan toda la información necesaria únicamente a través del constructor.



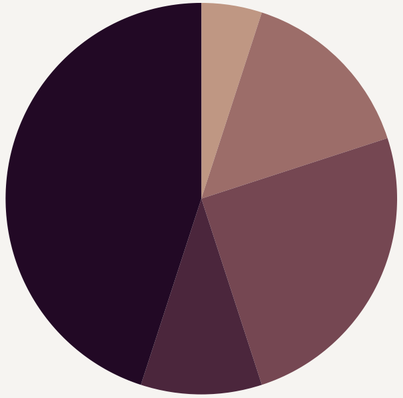
Los manejadores se alinean uno tras otro, formando una cadena.

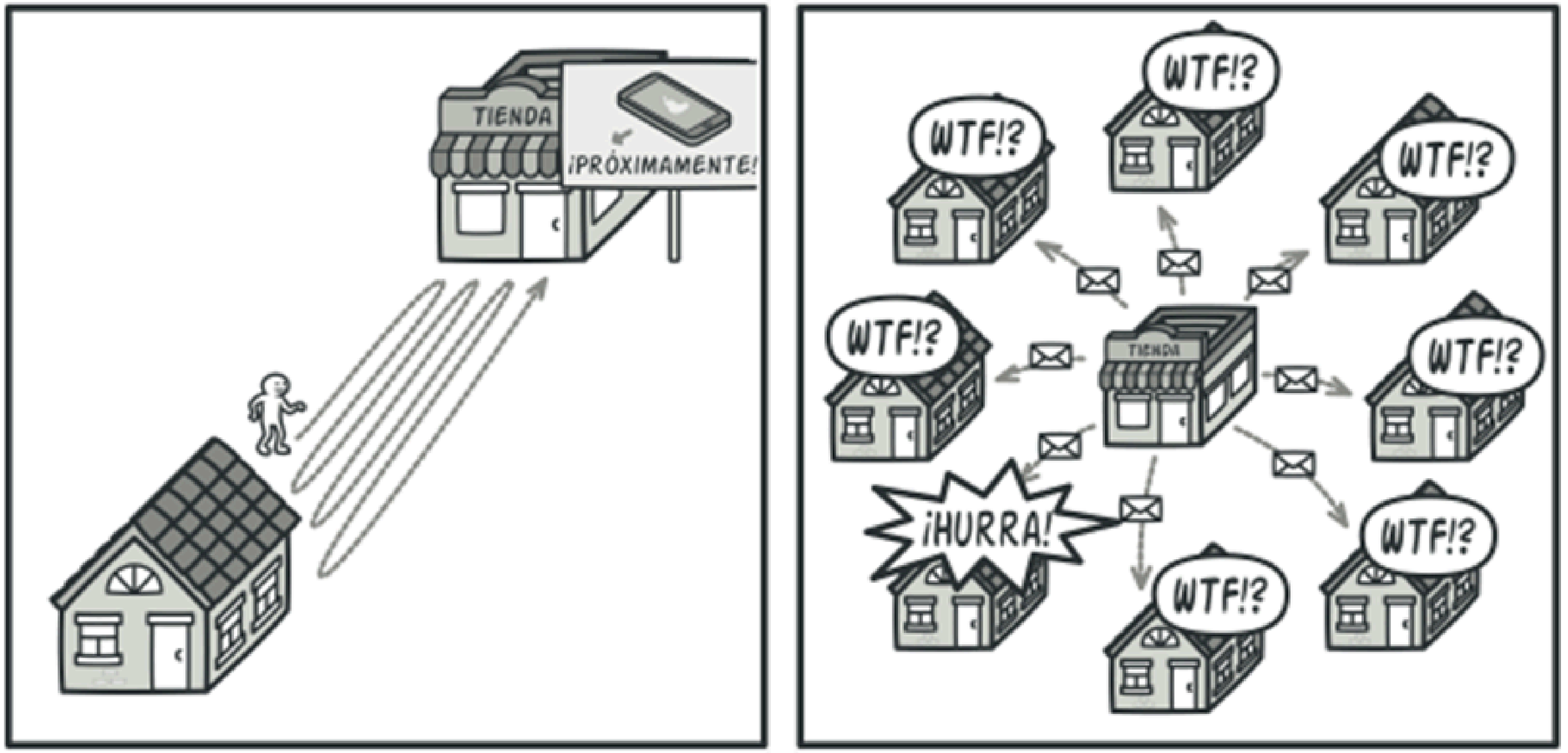

Chain of Responsibility
Ventana
Panel
Barra de Herramientas
SubPanel
Formulario
Botón
Botón
Texto
Entrada
Botón
Botón
Botón
Imagen
Casilla de Selección

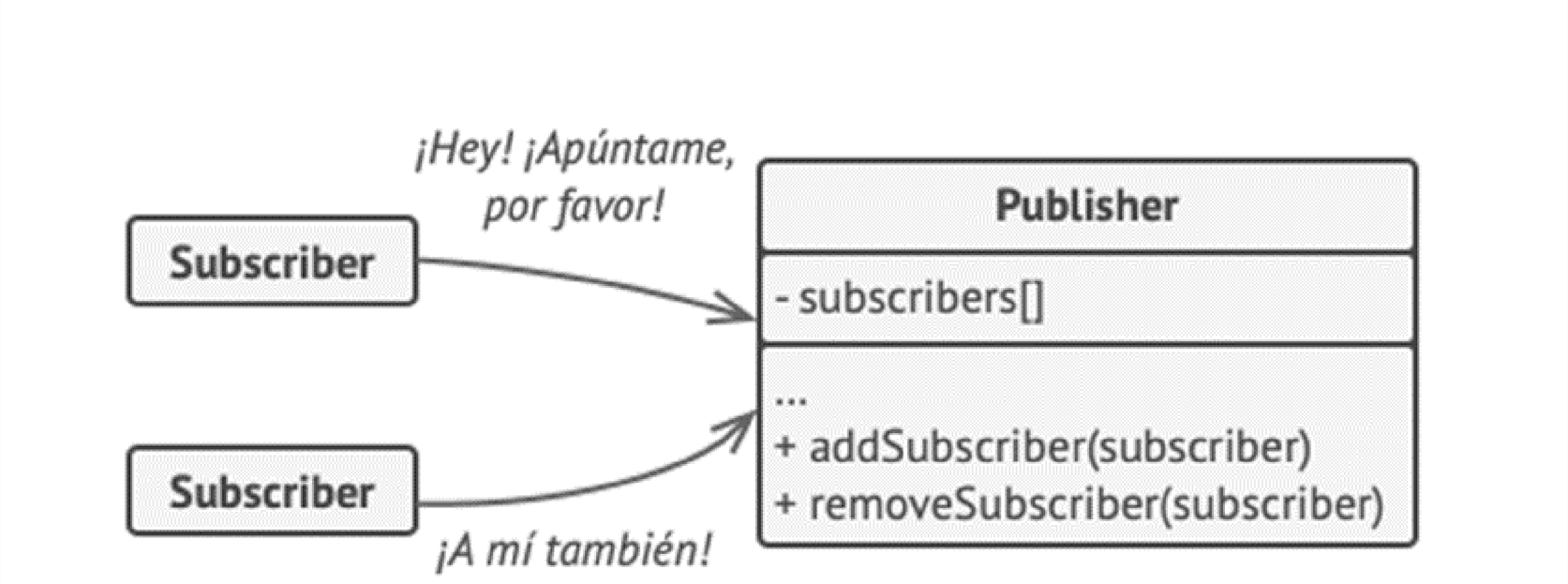
```
function InputTask (props) {
  const [input, setInput] = useState('');

  const handleChange = e => {
    setInput(e.target.value);
  }

  const handleCommit = e => {
    e.preventDefault();

    const newTask = {
      id: uuidv4(),
      text: input,
      completada: false
    }

    props.onSubmit( newTask );
}
```

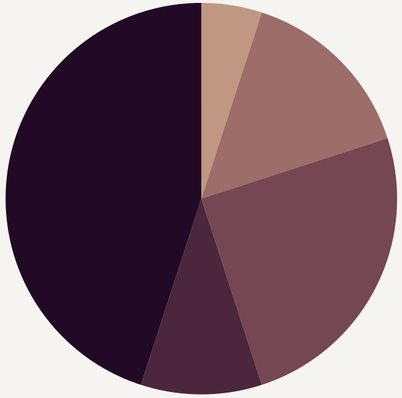
```
return (
  <form
  onSubmit={handleCommit}
  className="InputTask">
    <input className="Input"
    type="text"
    placeholder="Add a new task"
    name="text"
    onChange={handleChange}
    />
    <button className="TaskButton">
      Add task
    </button>
  </form>
);
```

OBSERVER




TIENDA
¡PRÓXIMAMENTE!
WTF!?
WTF!?
WTF!?
WTF!?
TIENDA
WTF!?
¡HURRA!
WTF!?
WTF!?


¡Hey! ¡Apúntame, por favor!
Subscriber
Subscriber
¡A mí también!
Publisher
- subscribers[]
...
+ addSubscriber(subscriber)
+ removeSubscriber(subscriber)



```java
// Interfaz Observable que define los métodos para agregar, eliminar y notificar observadores
interface Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Interfaz Observer que define el método update() para que los observadores reciban actualizaciones
interface Observer {
    void update();
}
```

```java
class Subject implements Observable {
    private int state;
    private List<Observer> observers = new ArrayList<>();

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyObservers();
    }

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

```java
// Clase Observer concreta que implementa la interfaz Observer
class ConcreteObserver implements Observer {
    private final Subject subject;

    public ConcreteObserver(Subject subject) {
        this.subject = subject;
        this.subject.addObserver(this);
    }

    @Override
    public void update() {
        System.out.println("Estado actualizado: " + subject.getState());
    }
}
```

```java
// Ejemplo de uso del patrón Observer
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Subject subject = new Subject();

        // Creamos dos observadores
        ConcreteObserver observer1 = new ConcreteObserver(subject);
        ConcreteObserver observer2 = new ConcreteObserver(subject);

        // Cambiamos el estado del sujeto
        subject.setState(state:5);
        subject.setState(state:10);

        // Removemos un observador
        subject.removeObserver(observer2);

        // Cambiamos el estado del sujeto nuevamente
        subject.setState(state:15);
    }
}
```

“MUCHAS GRACIAS”