Indice

---

1. Observer Pattern - Le Basi

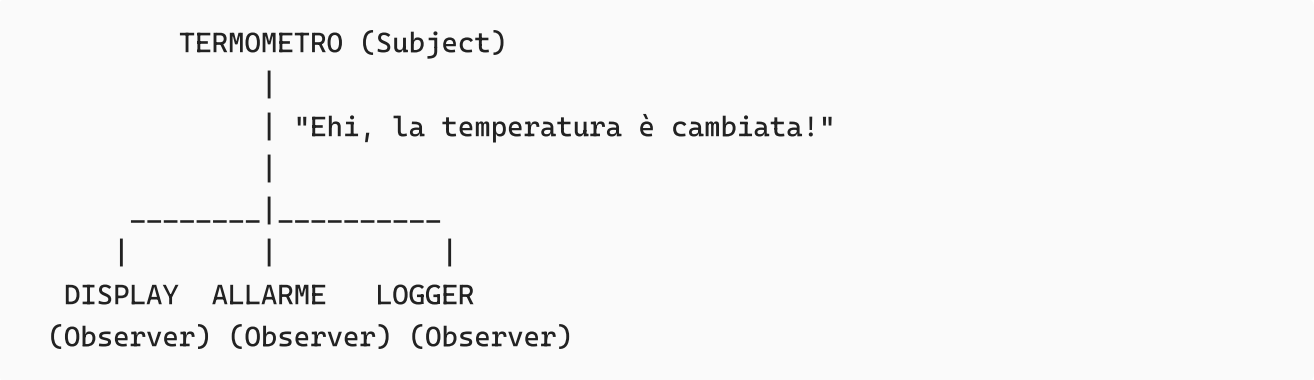
🎯 Il Problema

Immagina di avere un **termometro** che misura la temperatura. Ci sono diversi dispositivi interessati a sapere quando la temperatura cambia:

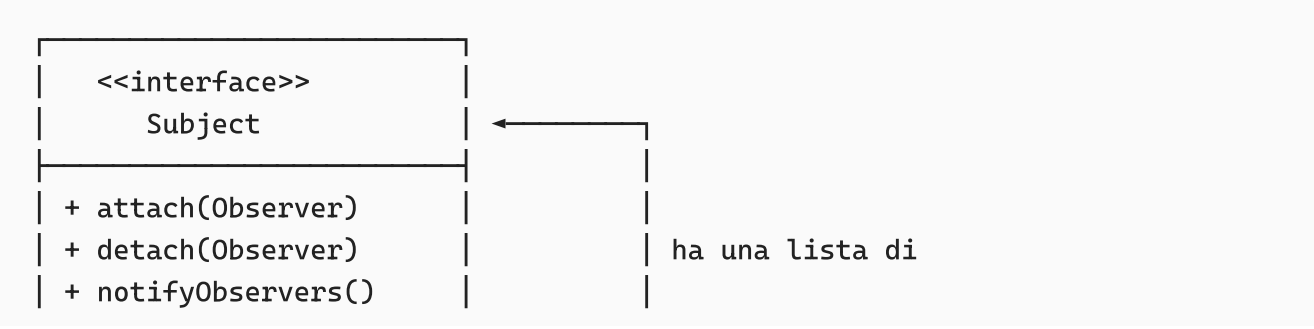
- Un **display** che mostra la temperatura
- Un **sistema di allarme** che suona se fa troppo caldo
- Un **logger** che salva i dati su un file

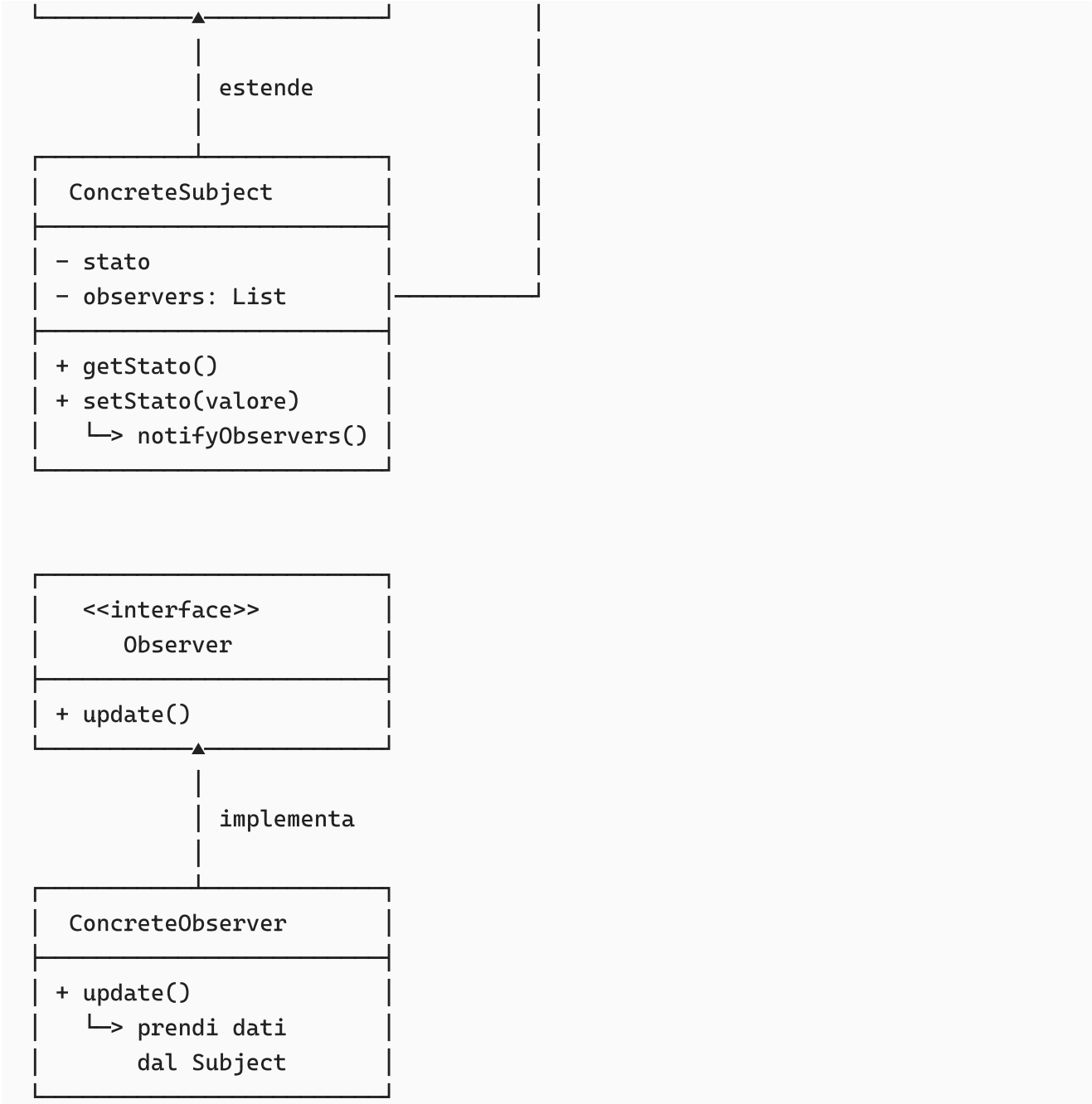
Problema: Come fa il termometro a notificare tutti questi dispositivi **senza conoscerli direttamente**?

💡 La Soluzione: Observer Pattern



📐 Struttura del Pattern





🔑 Componenti Chiave

1. **Subject (Soggetto Osservato)**
   - Mantiene una lista di Observer
   - Fornisce metodi per aggiungere/rimuovere Observer
   - Notifica tutti gli Observer quando il suo stato cambia
2. **Observer (Osservatore)**
   - Definisce un'interfaccia di aggiornamento
   - Implementa la logica di reazione ai cambiamenti

📝 Flusso di Esecuzione

```
└──────────────────────────────────────────────────────────┘

subject.attach(observerA)
subject.attach(observerB)
subject.attach(observerC)

┌──────────────────────────────────────────────────────────┐
│  PASSO 2: Cambiamento di Stato                           │
└──────────────────────────────────────────────────────────┘

subject.setStato(nuovoValore)
   ↓
   └─> notifyObservers()
          ↓
          ├─> observerA.update()
          ├─> observerB.update()
          └─> observerC.update()

┌──────────────────────────────────────────────────────────┐
│  PASSO 3: Aggiornamento Observer                         │
└──────────────────────────────────────────────────────────┘

Ogni Observer:
1. Riceve la notifica tramite update()
2. Legge il nuovo stato dal Subject
3. Esegue la propria logica specifica
```

---

2. Observer Pattern - Esempio Concreto

🌡️ Sistema di Monitoraggio Temperatura

Implementiamo il termometro dell'esempio precedente.

Codice Java

```java
// ========================================
// INTERFACCE
// ========================================

/**
 * Observer - Chi vuole essere notificato
 */
interface TemperatureObserver {
    void update(float temperatura);
}
```

```java
// ========================================
// SUBJECT (SOGGETTO OSSERVATO)
// ========================================

/**
 * Il termometro che misura la temperatura
 */
class Termometro {
    // Lista degli osservatori registrati
    private List<TemperatureObserver> observers = new ArrayList<>();

    // Lo stato interno: la temperatura corrente
    private float temperaturaCorrente;

    /**
     * Registra un nuovo osservatore
     */
    public void attach(TemperatureObserver observer) {
        observers.add(observer);
        System.out.println("✓ Nuovo observer registrato");
    }

    /**
     * Rimuove un osservatore
     */
    public void detach(TemperatureObserver observer) {
        observers.remove(observer);
        System.out.println("✗ Observer rimosso");
    }

    /**
     * Imposta la temperatura e notifica tutti
     */
    public void setTemperatura(float temp) {
        System.out.println("\n🌡️  Temperatura cambiata: " + temp + "°C");
        this.temperaturaCorrente = temp;
        notifyObservers();
    }

    /**
     * Notifica TUTTI gli observer registrati
     */
    private void notifyObservers() {
        System.out.println("📢 Notifica in corso a " + observers.size() + "
observer...\n");
        for (TemperatureObserver observer : observers) {
            observer.update(temperaturaCorrente);
        }
    }
```

```
    public float getTemperatura() {
        return temperaturaCorrente;
    }
}

// ========================================
// OBSERVER CONCRETI
// ========================================

/**
 * Display che mostra la temperatura
 */
class DisplayTemperatura implements TemperatureObserver {
    @Override
    public void update(float temperatura) {
        System.out.println("📺 DISPLAY: Temperatura attuale = " +
temperatura + "°C");
    }
}

/**
 * Sistema di allarme
 */
class SistemaAllarme implements TemperatureObserver {
    private static final float SOGLIA_MASSIMA = 30.0f;

    @Override
    public void update(float temperatura) {
        if (temperatura > SOGLIA_MASSIMA) {
            System.out.println("🚨 ALLARME: Temperatura CRITICA! (" +
temperatura + "°C)");
        } else {
            System.out.println("✅ ALLARME: Temperatura nella norma");
        }
    }
}

/**
 * Logger che salva i dati
 */
class TemperatureLogger implements TemperatureObserver {
    @Override
    public void update(float temperatura) {
        String timestamp = LocalDateTime.now().format(
            DateTimeFormatter.ofPattern("HH:mm:ss")
        );
        System.out.println("💾 LOGGER: [" + timestamp + "] Temp = " +
temperatura + "°C");
    }
}
```

```java
// ========================================
// UTILIZZO
// ========================================

public class EsempioTermometro {
    public static void main(String[] args) {
        // 1. Creo il Subject (termometro)
        Termometro termometro = new Termometro();

        // 2. Creo gli Observer
        DisplayTemperatura display = new DisplayTemperatura();
        SistemaAllarme allarme = new SistemaAllarme();
        TemperatureLogger logger = new TemperatureLogger();

        // 3. Registro gli Observer
        System.out.println("=== FASE DI REGISTRAZIONE ===");
        termometro.attach(display);
        termometro.attach(allarme);
        termometro.attach(logger);

        // 4. Cambio la temperatura (tutti vengono notificati!)
        System.out.println("\n=== PRIMA MISURAZIONE ===");
        termometro.setTemperatura(22.5f);

        System.out.println("\n=== SECONDA MISURAZIONE ===");
        termometro.setTemperatura(28.0f);

        System.out.println("\n=== TERZA MISURAZIONE (ALLARME!) ===");
        termometro.setTemperatura(35.0f);

        // 5. Rimuovo un observer
        System.out.println("\n=== RIMOZIONE DISPLAY ===");
        termometro.detach(display);

        System.out.println("\n=== QUARTA MISURAZIONE ===");
        termometro.setTemperatura(25.0f);
    }
}
```

Output del Programma

```
=== FASE DI REGISTRAZIONE ===
✓ Nuovo observer registrato
✓ Nuovo observer registrato
✓ Nuovo observer registrato

=== PRIMA MISURAZIONE ===
🌡️  Temperatura cambiata: 22.5°C
```

```
📢 Notifica in corso a 3 observer...

📺 DISPLAY: Temperatura attuale = 22.5°C
✅ ALLARME: Temperatura nella norma
💾 LOGGER: [14:35:22] Temp = 22.5°C

=== SECONDA MISURAZIONE ===
🌡️  Temperatura cambiata: 28.0°C
📢 Notifica in corso a 3 observer...

📺 DISPLAY: Temperatura attuale = 28.0°C
✅ ALLARME: Temperatura nella norma
💾 LOGGER: [14:35:22] Temp = 28.0°C

=== TERZA MISURAZIONE (ALLARME!) ===
🌡️  Temperatura cambiata: 35.0°C
📢 Notifica in corso a 3 observer...

📺 DISPLAY: Temperatura attuale = 35.0°C
🚨 ALLARME: Temperatura CRITICA! (35.0°C)
💾 LOGGER: [14:35:22] Temp = 35.0°C

=== RIMOZIONE DISPLAY ===
✗ Observer rimosso

=== QUARTA MISURAZIONE ===
🌡️  Temperatura cambiata: 25.0°C
📢 Notifica in corso a 2 observer...

✅ ALLARME: Temperatura nella norma
💾 LOGGER: [14:35:22] Temp = 25.0°C
```

✅ Vantaggi Dimostrati

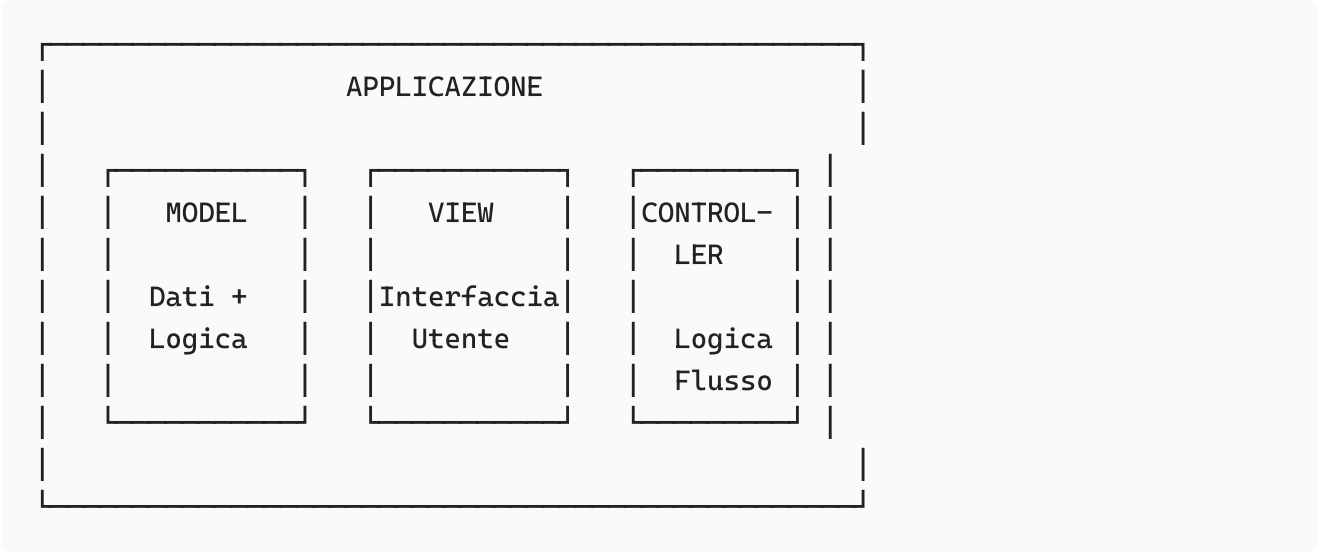
1. **Disaccoppiamento**: Il `Termometro` non sa nulla di Display, Allarme o Logger
2. **Estensibilità**: Posso aggiungere nuovi observer (es. `CloudUploader`) senza modificare il termometro
3. **Dinamicità**: Posso aggiungere/rimuovere observer a runtime

---

3. MVC Pattern - Introduzione

🎯 Cos'è MVC?

MVC (Model-View-Controller) è un **pattern architetturale** che separa un'applicazione in **tre componenti principali**:



📦 I Tre Componenti

1. **MODEL** (Il Cervello)

- **Cosa fa**: Gestisce i dati e la logica di business
- **Responsabilità**:
  - Memorizza lo stato dell'applicazione
  - Esegue operazioni sui dati
  - Notifica le View quando i dati cambiano (tramite Observer!)
- **Caratteristica**: **Indipendente dalla UI** (può funzionare senza interfaccia grafica)

```
// Esempio: Model di un contatore
class ContatorModel {
    private int valore = 0;

    public void incrementa() {
        valore++;
        // Notifica le View (Observer Pattern!)
    }

    public int getValore() {
        return valore;
    }
}
```

2. **VIEW** (Gli Occhi)

- **Cosa fa**: Mostra i dati all'utente
- **Responsabilità**:
  - Visualizza lo stato del Model
  - Cattura l'input dell'utente (click, tasti, etc.)
  - Si aggiorna automaticamente quando il Model cambia

- **Caratteristica**: **Passiva** (non contiene logica di business)

```
// Esempio: View grafica
class ContatoreView {
    public void mostraValore(int valore) {
        System.out.println("╔═══════════════╗");
        System.out.println("║ Valore: " + valore + " ║");
        System.out.println("╚═══════════════╝");
    }
}
```

3. **CONTROLLER** (Il Coordinatore)

- **Cosa fa**: Gestisce il flusso dell'applicazione
- **Responsabilità**:
  - Interpreta gli input dell'utente
  - Traduce gli input in operazioni sul Model
  - Decide quale View mostrare
- **Caratteristica**: **Intermediario** tra View e Model

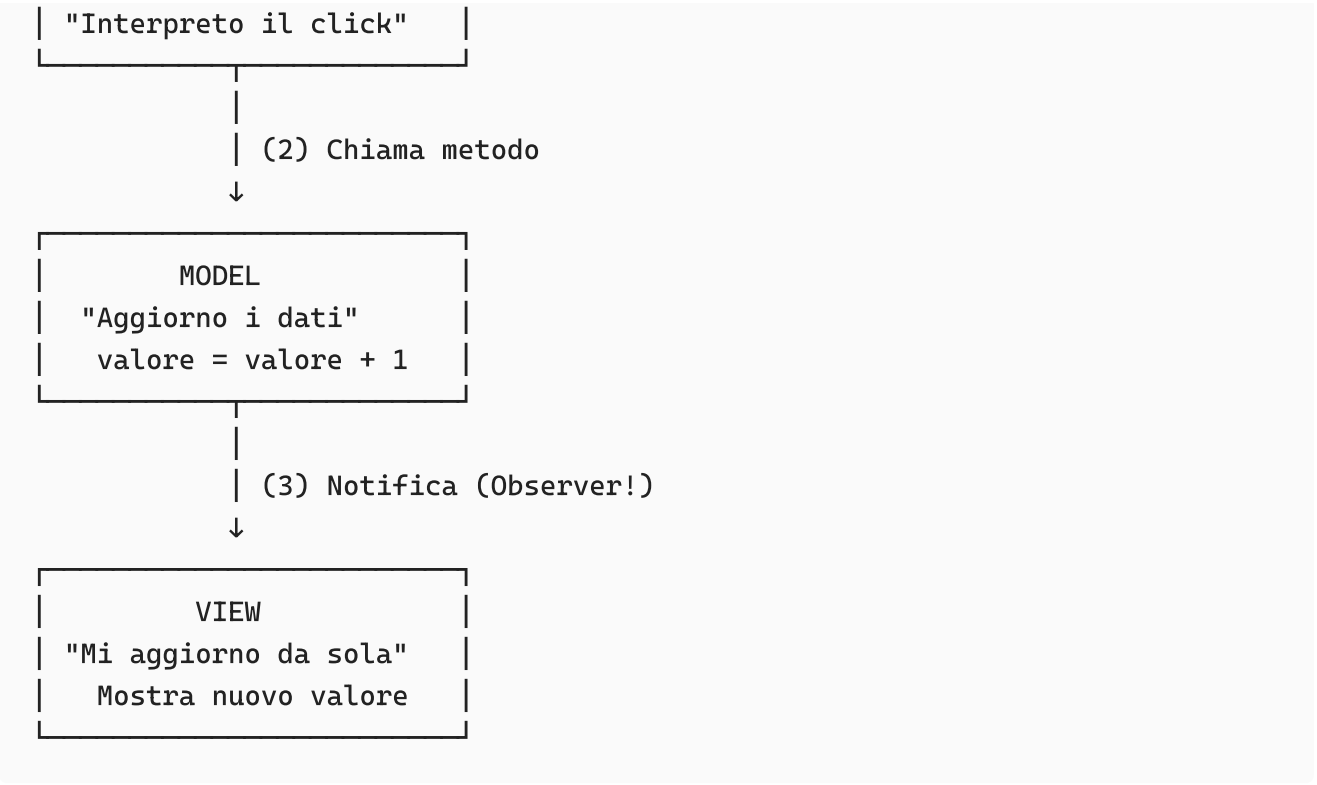
```
// Esempio: Controller
class ContatoreController {
    private ContatorModel model;

    public void handleClickIncrementa() {
        model.incrementa();
    }
}
```

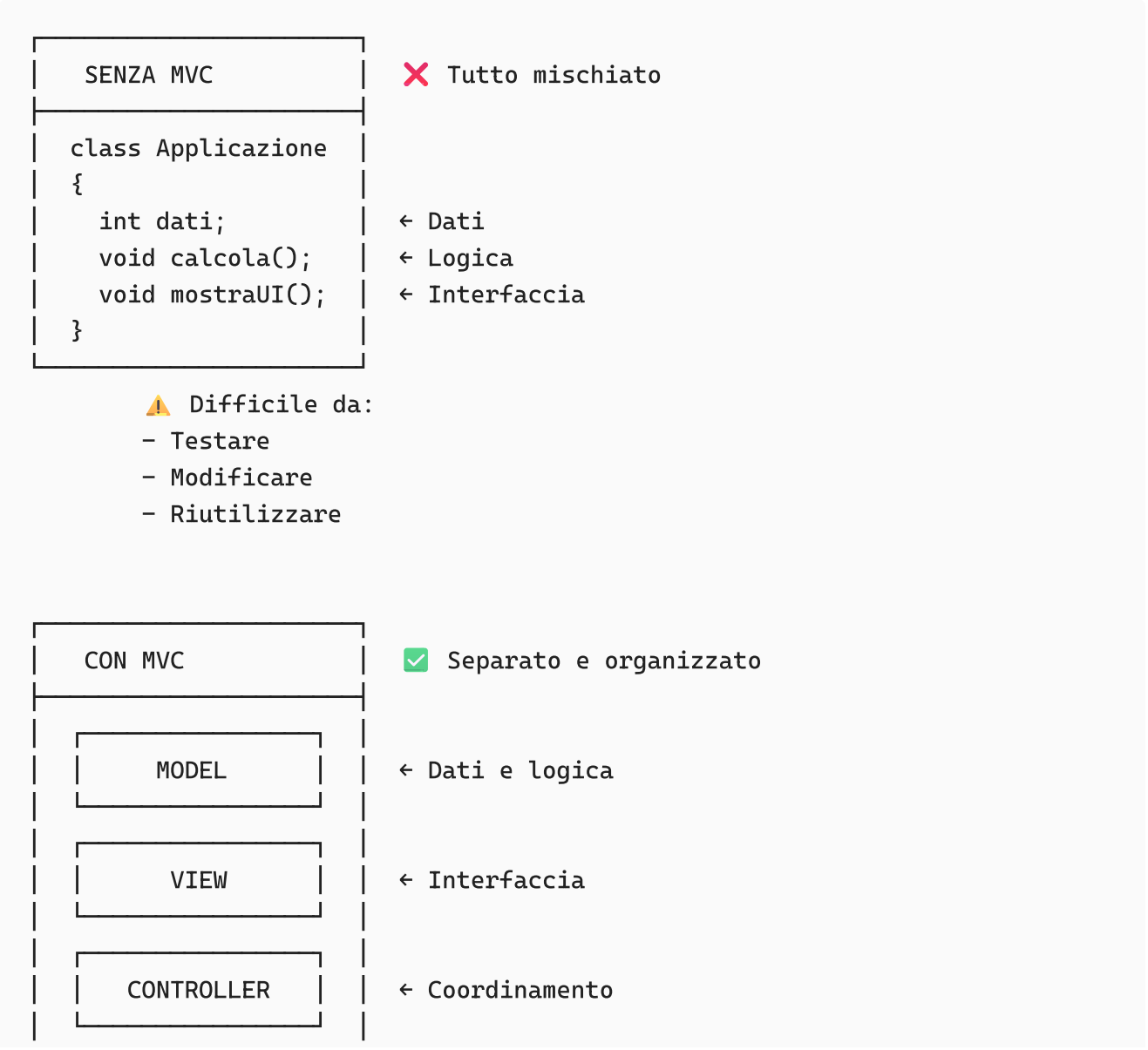
🔄 Flusso di Comunicazione

```
┌─────────────────────────────────────────────────────────┐
│  1. UTENTE INTERAGISCE                                  │
└─────────────────────────────────────────────────────────┘

    Utente clicca pulsante "+" sulla VIEW
              ↓
┌──────────────────────────┐
│         VIEW             │
│   "Ho ricevuto click"    │
└──────────┬───────────────┘
           │
           │ (1) Invio evento
           ↓
┌──────────────────────────┐
│      CONTROLLER          │
```



🎨 Separazione delle Responsabilità



```
└──────────────────────┘

        ✅ Facile da:
        - Testare (ogni parte separata)
        - Modificare (cambio solo ciò che serve)
        - Riutilizzare (Model indipendente)
```

---

4. MVC + Observer - La Combinazione

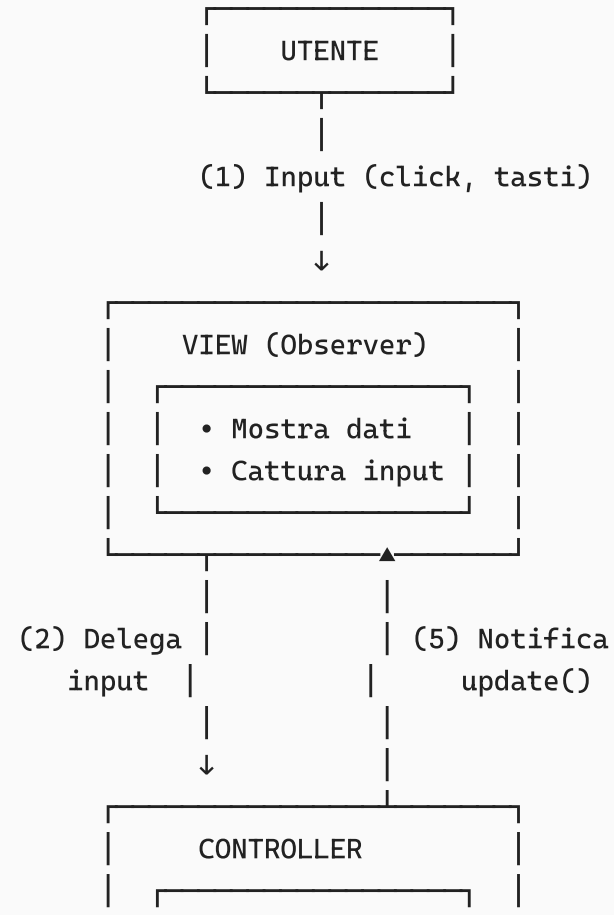
🔗 La Connessione Magica

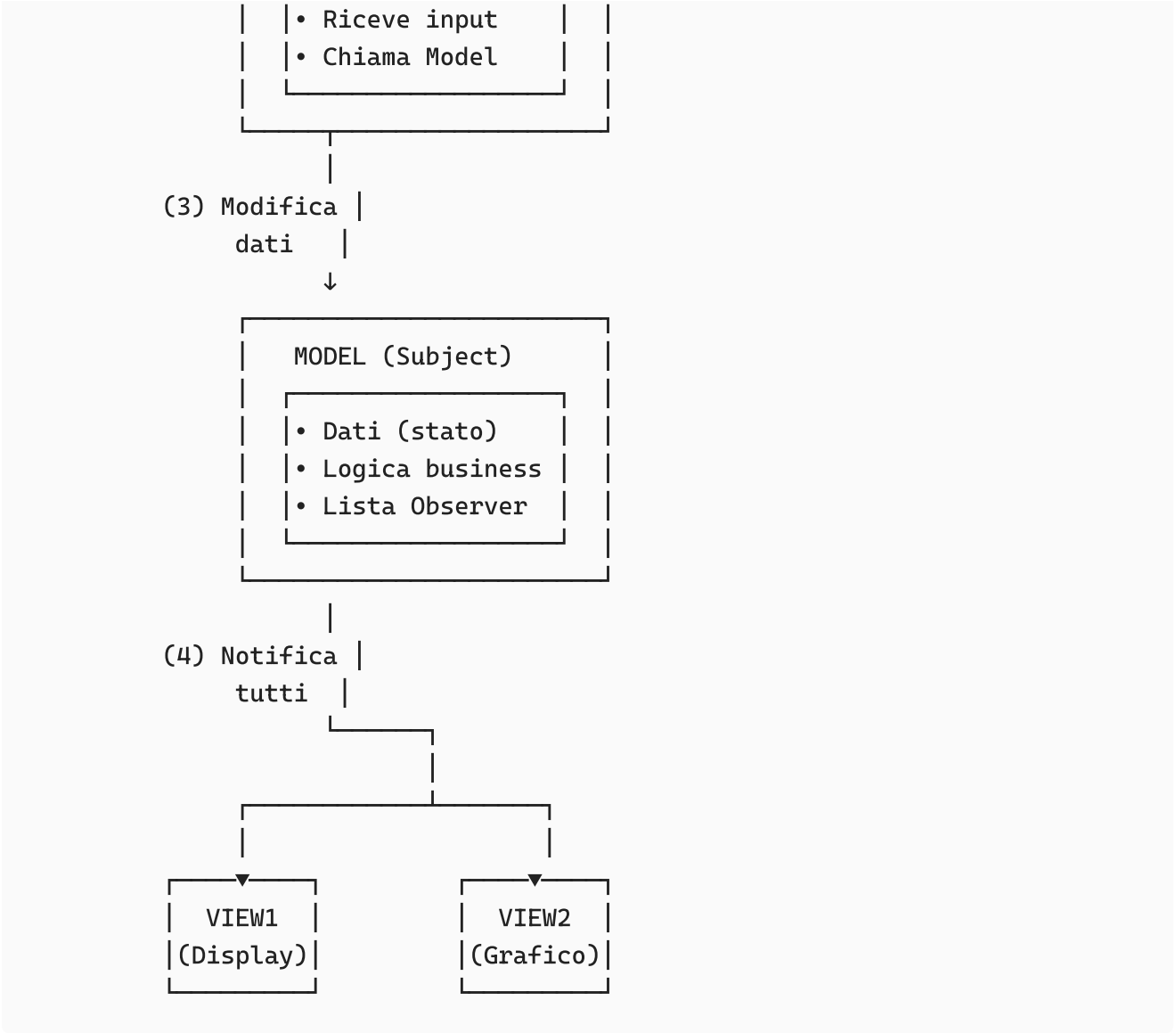
Domanda chiave: Come fa la View a sapere quando il Model è cambiato?

Risposta: Usando l'**Observer Pattern**!

```
┌─────────────────────────────────────────────────┐
│         MVC + Observer                          │
│                                                 │
│   MODEL = Subject    (chi viene osservato)      │
│   VIEW  = Observer   (chi osserva)              │
│                                                 │
└─────────────────────────────────────────────────┘
```

📐 Architettura Completa





🎯 Mappatura Esatta

MVC Component	Observer Pattern Role	Responsabilità
Model	**Subject**	Mantiene lista di View, notifica cambiamenti
View	**Observer**	Implementa `update()`, si aggiorna quando notificata
Controller	*Nessun ruolo diretto*	Modifica il Model, che poi notifica le View

🔄 Sequenza Completa

```
SCENARIO: Utente clicca pulsante "Incrementa"

┌─────────────────────────────────────────────────────────┐
│ STEP 1: Click del pulsante                              │
└─────────────────────────────────────────────────────────┘

Utente: *click*
        ↓
```

```
View: "Ho ricevuto un evento click sul pulsante +"


┌─────────────────────────────────────────────────────────┐
│ STEP 2: View delega al Controller                       │
└─────────────────────────────────────────────────────────┘

View → Controller
"controller.handleIncrementa()"


┌─────────────────────────────────────────────────────────┐
│ STEP 3: Controller modifica il Model                    │
└─────────────────────────────────────────────────────────┘

Controller → Model
"model.incrementa()"
         ↓
Model: valore cambia da 5 a 6


┌─────────────────────────────────────────────────────────┐
│ STEP 4: Model notifica TUTTI gli Observer (Pattern!)    │
└─────────────────────────────────────────────────────────┘

Model.notifyObservers()
         ↓
    ┌────┴────┐
    │         │
    ↓         ↓
View1       View2
Observer    Observer


┌─────────────────────────────────────────────────────────┐
│ STEP 5: Ogni View si aggiorna                           │
└─────────────────────────────────────────────────────────┘

View1.update()
  → legge model.getValore()  (= 6)
  → mostra "Valore: 6"

View2.update()
  → legge model.getValore()  (= 6)
  → disegna grafico con valore 6


┌─────────────────────────────────────────────────────────┐
│ RISULTATO: Tutte le View mostrano il nuovo valore       │
└─────────────────────────────────────────────────────────┘
```

💡 Perché Funziona Così Bene?

```
┌──────────────────────────────────────────────┐
│  SENZA Observer                              │
├──────────────────────────────────────────────┤
│                                              │
│  Controller:                                 │
│    model.incrementa()                        │
│    view1.aggiorna()  ← Controller deve conoscere  │
│    view2.aggiorna()  ← tutte le view!        │
│    view3.aggiorna()  ← Accoppiamento forte ❌  │
│                                              │
└──────────────────────────────────────────────┘

┌──────────────────────────────────────────────┐
│  CON Observer                                │
├──────────────────────────────────────────────┤
│                                              │
│  Controller:                                 │
│    model.incrementa()  ← Solo questo!        │
│      └─> Model notifica automaticamente      │
│          tutte le View registrate            │
│                                              │
│  ✅ Controller non sa niente delle View      │
│  ✅ Posso aggiungere View senza toccare Controller │
│  ✅ Disaccoppiamento totale                  │
│                                              │
└──────────────────────────────────────────────┘
```

---

5. Esempio Completo Integrato

🎮 Applicazione: Contatore con MVC + Observer

Creiamo un'applicazione completa che dimostra come MVC e Observer lavorano insieme.

Codice Completo

```java
import java.util.*;

// ============================================================
// OBSERVER PATTERN - INTERFACCIA
// ============================================================

/**
 * Observer - Le View implementano questa interfaccia
 */
interface ContaOsservatore {
    void update(int nuovoValore);
```

```
}

// ============================================================
// MODEL (Subject)
// ============================================================

/**
 * Il Model contiene i dati e la logica di business.
 * È anche il Subject nel pattern Observer.
 */
class ContaModel {
    // Stato interno
    private int valore = 0;

    // Lista degli osservatori (le View!)
    private List<ContaOsservatore> osservatori = new ArrayList<>();

    // ------------------------------------------------
    // METODI OBSERVER PATTERN (Subject)
    // ------------------------------------------------

    public void registraOsservatore(ContaOsservatore osservatore) {
        osservatori.add(osservatore);
        System.out.println("✓ View registrata. Totale view: " +
osservatori.size());
    }

    public void rimuoviOsservatore(ContaOsservatore osservatore) {
        osservatori.remove(osservatore);
        System.out.println("✗ View rimossa. Totale view: " +
osservatori.size());
    }

    /**
     * Notifica TUTTI gli osservatori registrati
     */
    private void notificaOsservatori() {
        System.out.println("  📢 Notifica " + osservatori.size() + "
view...");
        for (ContaOsservatore osservatore : osservatori) {
            osservatore.update(valore);
        }
    }

    // ------------------------------------------------
    // LOGICA DI BUSINESS
    // ------------------------------------------------

    public void incrementa() {
        valore++;
```

```
        System.out.println("  [MODEL] Valore incrementato a: " + valore);
        notificaOsservatori(); // ← Observer Pattern!
    }

    public void decrementa() {
        valore--;
        System.out.println("  [MODEL] Valore decrementato a: " + valore);
        notificaOsservatori(); // ← Observer Pattern!
    }

    public void reset() {
        valore = 0;
        System.out.println("  [MODEL] Valore azzerato");
        notificaOsservatori(); // ← Observer Pattern!
    }

    public int getValore() {
        return valore;
    }
}

// ============================================================
// VIEW 1: Visualizzazione Grafica (Observer)
// ============================================================

class ViewGrafica implements ContaOsservatore {
    @Override
    public void update(int nuovoValore) {
        System.out.println("  📺 [VIEW GRAFICA]");
        System.out.println("     ╔═══════════════╗");
        System.out.println("     ║  Contatore: " + String.format("%2d",
nuovoValore) + " ║");
        System.out.println("     ╚═══════════════╝");
    }
}

// ============================================================
// VIEW 2: Visualizzazione Testuale (Observer)
// ============================================================

class ViewTestuale implements ContaOsservatore {
    @Override
    public void update(int nuovoValore) {
        System.out.println("  📄 [VIEW TESTUALE] Valore attuale = " +
nuovoValore);
    }
}

// ============================================================
// VIEW 3: Grafico a Barre (Observer)
```

```java
// ============================================================

class ViewGraficoBarre implements ContaOsservatore {
    @Override
    public void update(int nuovoValore) {
        System.out.print("  📊 [GRAFICO] ");

        // Mostra barre positive o negative
        if (nuovoValore > 0) {
            for (int i = 0; i < nuovoValore; i++) {
                System.out.print("▓");
            }
            System.out.println(" (+" + nuovoValore + ")");
        } else if (nuovoValore < 0) {
            for (int i = 0; i < Math.abs(nuovoValore); i++) {
                System.out.print("▒");
            }
            System.out.println(" (" + nuovoValore + ")");
        } else {
            System.out.println("(0)");
        }
    }
}

// ============================================================
// CONTROLLER
// ============================================================

/**
 * Il Controller gestisce l'input dell'utente e aggiorna il Model.
 * Non sa NULLA delle View (disaccoppiamento!)
 */
class ContaController {
    private ContaModel model;

    public ContaController(ContaModel model) {
        this.model = model;
    }

    // ------------------------------------------------
    // GESTIONE INPUT UTENTE
    // ------------------------------------------------

    /**
     * Chiamato quando l'utente clicca "+"
     */
    public void handleIncrementa() {
        System.out.println("\n[CONTROLLER] Richiesta incremento");
        model.incrementa();
    }
```

```java
    /**
     * Chiamato quando l'utente clicca "-"
     */
    public void handleDecrementa() {
        System.out.println("\n[CONTROLLER] Richiesta decremento");
        model.decrementa();
    }

    /**
     * Chiamato quando l'utente clicca "Reset"
     */
    public void handleReset() {
        System.out.println("\n[CONTROLLER] Richiesta reset");
        model.reset();
    }
}

// ============================================================================
// MAIN - CONFIGURAZIONE SISTEMA
// ============================================================================

public class MVCObserverCompleto {
    public static void main(String[] args) {
        System.out.println("╔═══════════════════════════════════════════╗");
        System.out.println("║   SISTEMA MVC + OBSERVER PATTERN          ║");
        System.out.println("╚═══════════════════════════════════════════╝\n");

        // -------------------------------------------
        // STEP 1: Creo il MODEL (Subject)
        // -------------------------------------------
        System.out.println("▶ Fase 1: Creazione Model");
        ContaModel model = new ContaModel();

        // -------------------------------------------
        // STEP 2: Creo le VIEW (Observer)
        // -------------------------------------------
        System.out.println("\n▶ Fase 2: Creazione View");
        ViewGrafica viewGrafica = new ViewGrafica();
        ViewTestuale viewTestuale = new ViewTestuale();
        ViewGraficoBarre viewGrafico = new ViewGraficoBarre();

        // -------------------------------------------
        // STEP 3: Registro le View come Observer
        // -------------------------------------------
        System.out.println("\n▶ Fase 3: Registrazione View al Model");
        model.registraOsservatore(viewGrafica);
        model.registraOsservatore(viewTestuale);
        model.registraOsservatore(viewGrafico);
```

```
        // ------------------------------------------
        // STEP 4: Creo il CONTROLLER
        // ------------------------------------------
        System.out.println("\n▶ Fase 4: Creazione Controller");
        ContaController controller = new ContaController(model);
        System.out.println("✓ Controller creato e collegato al Model");

        // ------------------------------------------
        // STEP 5: Simulazione interazioni utente
        // ------------------------------------------
        System.out.println("\n╔═══════════════════════════════════════╗");
        System.out.println("║      SIMULAZIONE INTERAZIONI          ║");
        System.out.println("╚═══════════════════════════════════════╝");

        // Utente clicca "+"
        controller.handleIncrementa();

        // Utente clicca "+" altre 2 volte
        controller.handleIncrementa();
        controller.handleIncrementa();

        // Utente clicca "-"
        controller.handleDecrementa();

        // Utente clicca "Reset"
        controller.handleReset();

        // Utente clicca "-" per andare in negativo
        controller.handleDecrementa();
        controller.handleDecrementa();

        // ------------------------------------------
        // STEP 6: Dimostro la dinamicità
        // ------------------------------------------
        System.out.println("\n╔═══════════════════════════════════════╗");
        System.out.println("║   RIMOZIONE DINAMICA DI UNA VIEW      ║");
        System.out.println("╚═══════════════════════════════════════╝");

        System.out.println("\n▶ Rimuovo la View Grafica");
        model.rimuoviOsservatore(viewGrafica);

        System.out.println("\n▶ Incremento dopo rimozione");
        controller.handleIncrementa();

        System.out.println("\n╔═══════════════════════════════════════╗");
        System.out.println("║         FINE DIMOSTRAZIONE            ║");
        System.out.println("╚═══════════════════════════════════════╝");
    }
}
```

Output del Programma

```
╔═══════════════════════════════════════╗
║   SISTEMA MVC + OBSERVER PATTERN      ║
╚═══════════════════════════════════════╝

▶ Fase 1: Creazione Model

▶ Fase 2: Creazione View

▶ Fase 3: Registrazione View al Model
✓ View registrata. Totale view: 1
✓ View registrata. Totale view: 2
✓ View registrata. Totale view: 3

▶ Fase 4: Creazione Controller
✓ Controller creato e collegato al Model

╔═══════════════════════════════════════╗
║      SIMULAZIONE INTERAZIONI          ║
╚═══════════════════════════════════════╝

[CONTROLLER] Richiesta incremento
  [MODEL] Valore incrementato a: 1
  📢 Notifica 3 view...
  📺 [VIEW GRAFICA]
     ╔═══════════════╗
     ║  Contatore:  1 ║
     ╚═══════════════╝
  📄 [VIEW TESTUALE] Valore attuale = 1
  📊 [GRAFICO] █ (+1)

[CONTROLLER] Richiesta incremento
  [MODEL] Valore incrementato a: 2
  📢 Notifica 3 view...
  📺 [VIEW GRAFICA]
     ╔═══════════════╗
     ║  Contatore:  2 ║
     ╚═══════════════╝
  📄 [VIEW TESTUALE] Valore attuale = 2
  📊 [GRAFICO] ██ (+2)

[CONTROLLER] Richiesta incremento
  [MODEL] Valore incrementato a: 3
  📢 Notifica 3 view...
  📺 [VIEW GRAFICA]
     ╔═══════════════╗
     ║  Contatore:  3 ║
     ╚═══════════════╝
```

```
   📄 [VIEW TESTUALE] Valore attuale = 3
   📊 [GRAFICO] ███ (+3)

[CONTROLLER] Richiesta decremento
  [MODEL] Valore decrementato a: 2
   📢 Notifica 3 view...
   📺 [VIEW GRAFICA]
      ╔═══════════════════╗
      ╟  Contatore:  2 ╢
      ╚═══════════════════╝
   📄 [VIEW TESTUALE] Valore attuale = 2
   📊 [GRAFICO] ██ (+2)

[CONTROLLER] Richiesta reset
  [MODEL] Valore azzerato
   📢 Notifica 3 view...
   📺 [VIEW GRAFICA]
      ╔═══════════════════╗
      ╟  Contatore:  0 ╢
      ╚═══════════════════╝
   📄 [VIEW TESTUALE] Valore attuale = 0
   📊 [GRAFICO] (0)

[CONTROLLER] Richiesta decremento
  [MODEL] Valore decrementato a: -1
   📢 Notifica 3 view...
   📺 [VIEW GRAFICA]
      ╔═══════════════════╗
      ╟  Contatore: -1 ╢
      ╚═══════════════════╝
   📄 [VIEW TESTUALE] Valore attuale = -1
   📊 [GRAFICO] ▒ (-1)

[CONTROLLER] Richiesta decremento
  [MODEL] Valore decrementato a: -2
   📢 Notifica 3 view...
   📺 [VIEW GRAFICA]
      ╔═══════════════════╗
      ╟  Contatore: -2 ╢
      ╚═══════════════════╝
   📄 [VIEW TESTUALE] Valore attuale = -2
   📊 [GRAFICO] ▒▒ (-2)

╔═══════════════════════════════════════════╗
╟   RIMOZIONE DINAMICA DI UNA VIEW       ╢
╚═══════════════════════════════════════════╝

▶ Rimuovo la View Grafica
✗ View rimossa. Totale view: 2
```

```
▶ Incremento dopo rimozione

[CONTROLLER] Richiesta incremento
  [MODEL] Valore incrementato a: -1
    📢 Notifica 2 view...
    📄 [VIEW TESTUALE] Valore attuale = -1
    📊 [GRAFICO] ▒ (-1)

╔═══════════════════════════════════════════╗
║         FINE DIMOSTRAZIONE                ║
╚═══════════════════════════════════════════╝
```

🎓 Riepilogo Finale

Observer Pattern

```
┌─────────────────────────────────────────────────┐
│  OBSERVER PATTERN                               │
├─────────────────────────────────────────────────┤
│  • Problema: Notificare molti oggetti quando    │
│    uno cambia stato                             │
│  • Soluzione: Subject mantiene lista di Observer│
│  • Quando: 1-a-molti dipendenze, disaccoppiamento│
│  • Componenti:                                  │
│    - Subject: chi viene osservato               │
│    - Observer: chi osserva e reagisce           │
└─────────────────────────────────────────────────┘
```

MVC Pattern

```
┌─────────────────────────────────────────────────┐
│  MVC PATTERN                                    │
├─────────────────────────────────────────────────┤
│  • Problema: Separare dati, UI e logica         │
│  • Soluzione: Tre componenti distinti           │
│  • Quando: Applicazioni con interfaccia utente  │
│  • Componenti:                                  │
│    - Model: dati + business logic               │
│    - View: interfaccia utente                   │
│    - Controller: coordinamento                  │
└─────────────────────────────────────────────────┘
```

L'Unione Perfetta

```
┌─────────────────────────────────────────────────────┐
│  MVC + OBSERVER                                     │
├─────────────────────────────────────────────────────┤
│                                                     │
│  Model = Subject    (notifica cambiamenti)          │
│  View  = Observer   (si aggiorna automaticamente)   │
│                                                     │
│  Flusso:                                            │
│  1. Utente interagisce con View                     │
│  2. View delega al Controller                       │
│  3. Controller modifica Model                       │
│  4. Model notifica tutte le View (Observer!)        │
│  5. View si aggiornano automaticamente              │
│                                                     │
│  Vantaggio: Disaccoppiamento totale!                │
│  • Controller non conosce le View                   │
│  • Posso aggiungere View dinamicamente              │
│  • Model indipendente da UI                         │
│                                                     │
└─────────────────────────────────────────────────────┘
```

Push vs Pull Model

```
┌─────────────────────────────────────────────────────┐
│  PUSH MODEL (usato negli esempi)                    │
├─────────────────────────────────────────────────────┤
│  Model → Observer: "Ecco il nuovo valore!"          │
│                                                     │
│  void update(int nuovoValore) {                     │
│      mostra(nuovoValore);                           │
│  }                                                  │
│                                                     │
│  ✅ Più semplice                                    │
│  ❌ Observer riceve dati che forse non servono      │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│  PULL MODEL                                         │
├─────────────────────────────────────────────────────┤
│  Model → Observer: "Sono cambiato!"                 │
│  Observer → Model: "Dammi i dati che mi servono"    │
│                                                     │
│  void update() {                                    │
│      int valore = model.getValore();                │
│      mostra(valore);                                │
│  }                                                  │
│                                                     │
│  ✅ Più flessibile                                  │
```

```
│  ❌ Observer deve conoscere il Model                │
└────────────────────────────────────────────────────┘
```
