
Introduzione

Problema Fondamentale

Nelle applicazioni con **interfaccia utente (UI)**, gestire insieme:

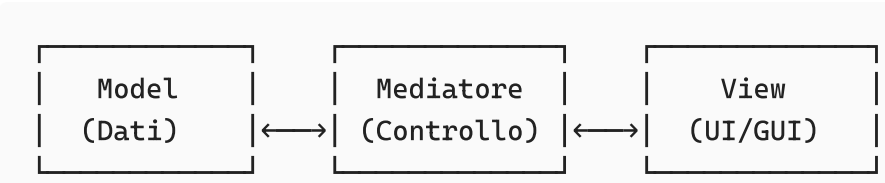
- **Logica di business** (elaborazione dati)
- **Presentazione** (visualizzazione)
- **Interazione utente** (input)

porta rapidamente a codice:

- **Monolitico** e difficile da testare
- **Accoppiato** strettamente
- **Non riutilizzabile**

Soluzione: Separazione delle Responsabilità

I **pattern architetturali MV*** (Model-View-*) risolvono questo problema separando:



Vantaggi:

- **Testabilità**: logica di business testabile senza UI
- **Manutenibilità**: modifiche UI non impattano la logica
- **Riusabilità**: stesso model per più view
- **Separazione competenze**: designer UI vs sviluppatori backend

---

Model-View-Controller (MVC)

Definizione

MVC è il pattern architetturale fondamentale per applicazioni con interfaccia utente. Separa l'applicazione in tre componenti interconnesse.

Componenti

1. Model

Responsabilità: rappresentare i **dati** e la **logica di business**.

Caratteristiche:

- Contiene lo **stato** dell'applicazione
- Implementa le **regole di business**
- **Indipendente** dalla presentazione
- Notifica i cambiamenti agli **observer** (tipicamente le View)

Esempio:

```
public class Studente {
    private String nome;
    private int età;
    private String matricola;

    // Business logic
    public boolean isIdoneoEsame() {
        return età >= 18;
    }

    // Notifica cambiamenti (Observer Pattern)
    private List<Observer> observers = new ArrayList<>();

    public void setNome(String nome) {
        this.nome = nome;
        notifyObservers();  // Notifica la View
    }
}
```

2. View

Responsabilità: visualizzare i dati e catturare **input** utente.

Caratteristiche:

- Presenta i dati del Model
- Cattura gli input dell'utente
- Delega al Controller l'elaborazione
- Si aggiorna quando il Model cambia (**Observer**)
- Mantiene un **riferimento al Model** (MVC classico)

Esempio:

```
public class StudenteView implements Observer {
    private JTextField nomeField;
    private JLabel etaLabel;

    @Override
    public void update(Observable o, Object arg) {
        Studente s = (Studente) o;
        nomeField.setText(s.getNome());
        etaLabel.setText(String.valueOf(s.getEtà()));
    }

    public void onButtonClick() {
        // Delega al Controller
        controller.handleUpdate(nomeField.getText());
    }
}
```

3. Controller

Responsabilità: coordinare Model e View.

Caratteristiche:

- Riceve input dalla View
- Elabora la logica applicativa
- Aggiorna il Model
- Può modificare quale View visualizzare

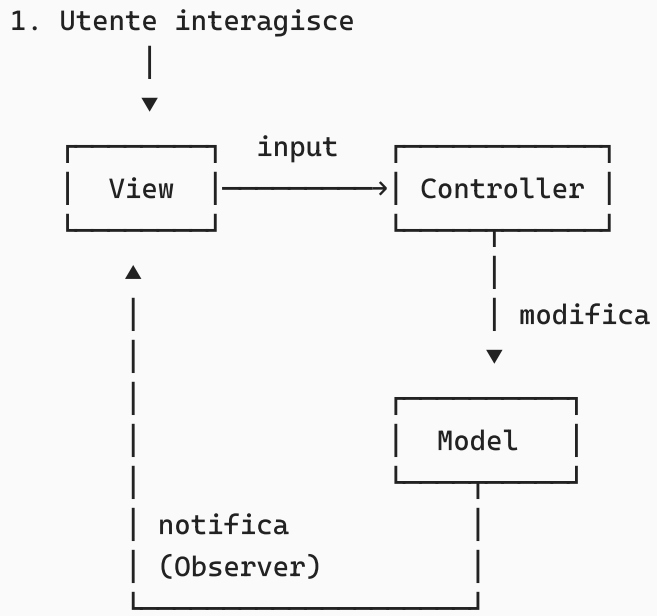
Esempio:

```
public class StudenteController {
    private Studente model;
    private StudenteView view;

    public StudenteController(Studente model, StudenteView view) {
        this.model = model;
        this.view = view;
    }

    public void handleUpdate(String nuovoNome) {
        // Validazione
        if (nuovoNome != null && !nuovoNome.isEmpty()) {
            model.setNome(nuovoNome);  // Il Model notifica la View
        }
    }
}
```

Flusso di Interazione MVC



Sequenza:

1. **Utente** interagisce con la **View**
2. **View** delega al **Controller**
3. **Controller** aggiorna il **Model**
4. **Model** notifica la **View** (Observer Pattern)
5. **View** si aggiorna leggendo dal Model

Varianti di Aggiornamento

Push Model

Caratteristica: la View viene **costantemente** aggiornata.

Quando usare:

- Applicazioni in **un solo ambiente** di esecuzione (es. JavaScript client-side)
- Real-time UI (dashboard, monitoring)
- Single-page applications

Meccanismo:

```
// Model notifica attivamente tutti gli observer
model.setValore(x);  // Trigger automatico
```

Pull Model

Caratteristica: la View richiede aggiornamenti **solo quando necessario**.

Quando usare:

- Applicazioni **distribuite** (client-server)
- Web applications tradizionali (JSP, Spring MVC)
- Overhead di notifica elevato

Meccanismo:

```
// View richiede esplicitamente i dati
String value = model.getValore();  // Esplicito
```

Design Pattern Coinvolti

Observer Pattern

Scopo: disaccoppiare Model e View.

```
┌─────────────┐             ┌──────────────────┐
│  Model      │ notifica    │     View         │
│ (Subject)   │────────────→│   (Observer)     │
└─────────────┘             └──────────────────┘
```

Il Model è il **Subject**, la View è l'**Observer**.

Strategy Pattern

Scopo: cambiare dinamicamente il comportamento del Controller.

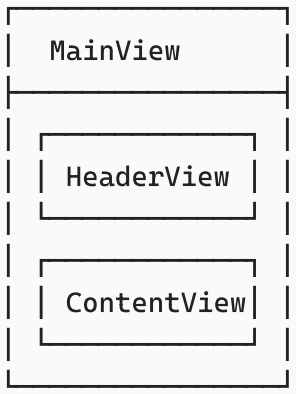
```
public interface InputStrategy {
    void handleInput(String input);
}

public class Controller {
    private InputStrategy strategy;

    public void setStrategy(InputStrategy strategy) {
        this.strategy = strategy;
    }
}
```

Composite Pattern

Scopo: costruire View complesse come composizione di View semplici.



Esempio Completo: Spring MVC (Pull Model)

1. Model (Service Layer)

```
@Service
public class StudenteService {
    @Autowired
    private StudenteRepository repository;

    public Studente getStudente(Long id) {
        return repository.findById(id).orElse(null);
    }

    public void saveStudente(Studente s) {
        repository.save(s);
    }
}
```

2. Controller

```
@Controller
@RequestMapping("/studenti")
public class StudenteController {
    @Autowired
    private StudenteService service;

    @GetMapping("/{id}")
    public String viewStudente(@PathVariable Long id, Model model) {
        Studente studente = service.getStudente(id);
        model.addAttribute("studente", studente);
        return "studente-detail";  // Nome della View
    }

    @PostMapping("/update")
```

```
    public String updateStudente(@ModelAttribute Studente studente) {
        service.saveStudente(studente);
        return "redirect:/studenti/" + studente.getId();
    }
}
```

3. View (JSP)

```
<%@ page contentType="text/html;charset=UTF-8" %>
<html>
<body>
    <h1>Dettaglio Studente</h1>
    <form action="/studenti/update" method="post">
        <input type="hidden" name="id" value="${studente.id}"/>
        Nome: <input type="text" name="nome" value="${studente.nome}"/>
        <button type="submit">Salva</button>
    </form>
</body>
</html>
```

4. DispatcherServlet (Front Controller)

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Flusso:

1. **HTTP Request** → **DispatcherServlet** (Front Controller)
2. **DispatcherServlet** → **Controller** (handler mapping)
3. **Controller** → **Service** (business logic)
4. **Controller** → **Model** (dati per la View)
5. **DispatcherServlet** → **View** (view resolving)
6. **View** → **HTTP Response**

Model-View-Presenter (MVP)

Definizione

MVP è una variante di MVC dove la View è **passiva** (dumb) e tutta la logica risiede nel **Presenter**.

Differenze chiave con MVC

Aspetto	MVC	MVP
View	Intelligente (legge dal Model)	Passiva (solo template)
Riferimento Model	View → Model	NO
Mediatore	Controller	Presenter
Logica UI	Nella View	Nel Presenter

Componenti

1. Model

Identico a MVC: logica di business indipendente.

2. View (Passive View)

Caratteristica fondamentale: la View è **stupida**.

- Non ha logica
- Non conosce il Model
- È tipicamente un **template dichiarativo** (XML, HTML)
- Espone un'**interfaccia** per il Presenter

Esempio (Android):

```
<!-- layout.xml - View passiva -->
<LinearLayout>
    <TextView android:id="@+id/nomeText"/>
    <EditText android:id="@+id/nomeInput"/>
    <Button android:id="@+id/salvaButton"/>
</LinearLayout>
```

```
public interface StudenteViewInterface {
    void showNome(String nome);
    void showError(String message);
    String getNomeInput();
}
```

3. Presenter (Man in the Middle)

Responsabilità: tutta la logica applicativa.

- Osserva il Model
- Aggiorna la View manualmente
- Contiene **tutta** la logica di presentazione

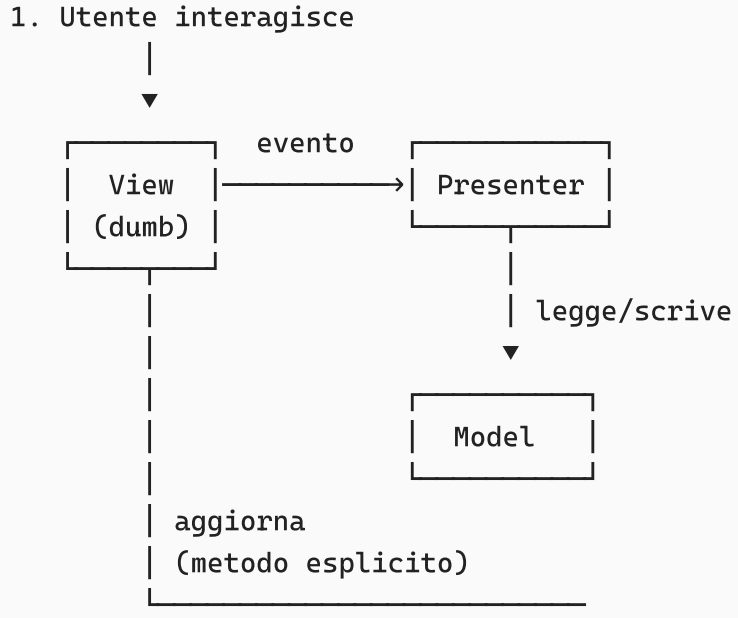
Esempio:

```
public class StudentePresenter {
    private StudenteViewInterface view;
    private Studente model;

    public StudentePresenter(StudenteViewInterface view, Studente model) {
        this.view = view;
        this.model = model;
    }

    public void onSalvaClicked() {
        String nome = view.getNomeInput();
        if (nome.isEmpty()) {
            view.showError("Nome obbligatorio");
            return;
        }
        model.setNome(nome);
        view.showNome(model.getNome());
    }
}
```

Flusso di Interazione MVP



Sequenza:

1. **View** notifica evento al **Presenter**
2. **Presenter** legge dal **Model**
3. **Presenter** elabora
4. **Presenter** aggiorna **esplicitamente** la View

Quando Usare MVP

Contesti ideali:

- **Android development** (framework nativo)
- View è un file dichiarativo (XML, XAML)
- Testabilità massima richiesta (mock della View)
- Logica UI complessa

Vantaggi:

- View completamente testabile con mock
- Separazione netta tra UI e logica
- Facile cambio di tecnologia UI

Svantaggi:

- Boilerplate code elevato
- Presenter può diventare molto grande (God Object)

Model-View-ViewModel (MVVM)

Definizione

MVVM separa completamente lo sviluppo UI dalla business logic tramite **data binding** bidirezionale.

Componenti

1. Model

Identico a MVC/MVP: dati e business logic.

2. View

Caratteristica: dichiarativa con **data binding**.

- Costruita con linguaggi di markup (XAML, HTML)
- **Binding automatico** con ViewModel
- Non contiene logica

Esempio (WPF/XAML):

```
<Window>
    <TextBox Text="{Binding Nome, Mode=TwoWay}"/>
    <TextBlock Text="{Binding Età}"/>
    <Button Command="{Binding SalvaCommand}"/>
</Window>
```

3. ViewModel

Responsabilità: proiezione del Model per la View.

- Espone **proprietà** per il binding
- Espone **comandi** per le azioni
- Converte dati dal Model al formato View
- Contiene la **validazione**

Esempio:

```
public class StudenteViewModel : INotifyPropertyChanged {
    private Studente model;

    public string Nome {
        get => model.Nome;
        set {
            model.Nome = value;
            OnPropertyChanged(nameof(Nome));
        }
    }

    public ICommand SalvaCommand { get; }

    public StudenteViewModel(Studente model) {
        this.model = model;
        SalvaCommand = new RelayCommand(Salva);
    }

    private void Salva() {
        // Business logic
        repository.Save(model);
    }
}
```

Data Binding

One-Way Binding

```
ViewModel ———→ View
```

La View si aggiorna automaticamente quando cambia il ViewModel.

Two-Way Binding

```
ViewModel ←———→ View
```

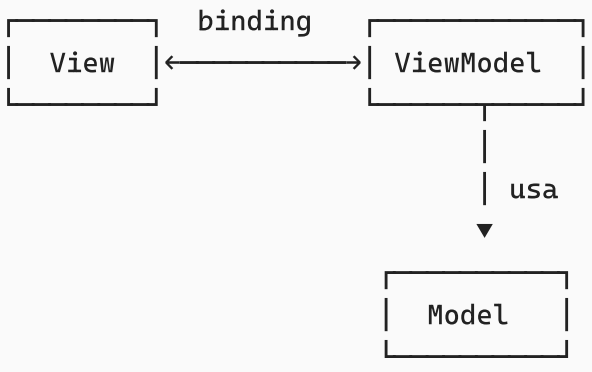
Cambiamenti in View o ViewModel si propagano automaticamente.

Esempio (Angular):

```
<!-- Two-way binding -->
<input [(ngModel)]="studente.nome">

<!-- One-way binding -->
<span>{{ studente.età }}</span>
```

Flusso di Interazione MVVM



Caratteristiche:

- La View **non** conosce il Model
- **Sincronizzazione automatica** tramite binding
- ViewModel espone solo ciò che serve alla View

Quando Usare MVVM

Contesti ideali:

- **WPF / Xamarin / Avalonia** (framework .NET)
- **Angular / Vue / React** (con binding library)
- UI complessa con molti input
- Necessità di testabilità massima

Vantaggi:

- **Nessun codice** per sincronizzazione View-ViewModel
- Testabilità eccellente (ViewModel puro)
- Designer e developer lavorano in parallelo
- Riutilizzo ViewModel per più View

Svantaggi:

- Richiede framework con binding nativo
- Curva di apprendimento elevata
- Debugging più complesso (binding nascosto)

Confronto tra Pattern MV*

Tabella Comparativa

Aspetto	MVC	MVP	MVVM
View conosce Model	✅ Sì	❌ No	❌ No
Logica nella View	Media	Minima	Nessuna
Sincronizzazione	Observer	Manuale	Binding
Testabilità View	Bassa	Alta	Alta
Complessità	Bassa	Media	Alta
Boilerplate	Basso	Alto	Medio
Uso tipico	Web tradizionale	Mobile (Android)	Desktop, SPA

Scelta del Pattern

Usa MVC quando:

- Sviluppi web application tradizionali
- Push/pull model sufficiente

- Semplicità preferita a testabilità assoluta

Usa MVP quando:

- View è dicharativa (XML, XAML)
- Testabilità critica
- Logica UI complessa
- Mobile development (Android nativo)

Usa MVVM quando:

- Framework supporta data binding nativo
- Desktop application (WPF)
- SPA moderne (Angular, Vue)
- Riuso ViewModel cruciale

Implementazioni Pratiche

Esempio JavaScript (MVC)

```
// Model
class StudenteModel {
    constructor(nome, età) {
        this.nome = nome;
        this.età = età;
        this.observers = [];
    }

    subscribe(observer) {
        this.observers.push(observer);
    }

    setNome(nome) {
        this.nome = nome;
        this.notify();
    }

    notify() {
        this.observers.forEach(obs => obs.update(this));
    }
}

// View
class StudenteView {
    constructor(model, controller) {
```

```
        this.model = model;
        this.controller = controller;
        this.model.subscribe(this);
        this.render();
    }

    render() {
        document.getElementById('nome').textContent = this.model.nome;
    }

    update(model) {
        this.render();
    }

    handleInput(value) {
        this.controller.updateNome(value);
    }
}

// Controller
class StudenteController {
    constructor(model) {
        this.model = model;
    }

    updateNome(nome) {
        if (nome && nome.length > 0) {
            this.model.setNome(nome);
        }
    }
}
```

Esempio React (Simplified MVVM)

```
// ViewModel (React Hook)
function useStudenteViewModel(initialStudente) {
    const [studente, setStudente] = useState(initialStudente);

    const updateNome = (nome) => {
        setStudente({ ...studente, nome });
    };

    const salva = () => {
        // Business logic / API call
        api.saveStudente(studente);
    };
```

```
    return { studente, updateNome, salva };
}

// View (Component)
function StudenteView({ initialStudente }) {
    const { studente, updateNome, salva } =
useStudenteViewModel(initialStudente);

    return (
        <div>
            <input
                value={studente.nome}
                onChange={(e) => updateNome(e.target.value)}
            />
            <button onClick={salva}>Salva</button>
        </div>
    );
}
```

Best Practices

1. Separazione Responsabilità

- **Model**: solo business logic, nessuna UI
- **View**: solo presentazione, minima logica
- **Mediatore**: coordinamento, validazione

2. Testabilità

```
// Test del Controller senza UI
@Test
public void testUpdateStudente() {
    Studente model = new Studente();
    StudenteController controller = new StudenteController(model, mockView);

    controller.handleUpdate("Mario");

    assertEquals("Mario", model.getNome());
}
```

3. Disaccoppiamento

- Usa **interfacce** per View e Model
- Dependency Injection per wiring

- Observer Pattern per notifiche

4. Granularità

- Un Controller/Presenter per View
- View composte da sotto-view
- Model può essere aggregato

5. Evita God Objects

Se Controller/Presenter supera 300 righe:

- Decomponi la View
- Estrai business logic in Service
- Usa Mediator Pattern

Design Pattern Correlati

Front Controller

Scopo: punto di ingresso unico per tutte le richieste.

Esempio: `DispatcherServlet` in Spring MVC

```
HTTP Request → Front Controller → Routing → Controller
```

Service Layer

Scopo: incapsulare business logic riutilizzabile.

```
Controller → Service Layer → Repository → Database
```

Repository Pattern

Scopo: astrarre l'accesso ai dati.

```
public interface StudenteRepository {
    Studente findById(Long id);
    void save(Studente s);
}
```

Conclusione

I **pattern MV*** sono fondamentali per costruire applicazioni scalabili e manutenibili. La scelta tra MVC, MVP e MVVM dipende da:

1. **Piattaforma**: web, mobile, desktop
2. **Framework disponibile**: supporto binding, dependency injection
3. **Requisiti di testabilità**: quanto critica è la copertura di test
4. **Complessità UI**: quanta logica di presentazione serve
5. **Competenze team**: familiarità con i pattern

Principio unificante: separare la presentazione dalla logica di business è sempre vincente, indipendentemente dal pattern scelto.

Riferimenti

- **Design Patterns: Elements of Reusable Object-Oriented Software**, GoF, Addison-Wesley, 1995
- **GUI Architectures**, Martin Fowler - http://martinfowler.com/eaaDev/uiArchs.html
- **Core J2EE Patterns**, Sun Microsystems
- **Learning JavaScript Design Patterns**, Addy Osmani
- **Developing Backbone.js Applications**, Addy Osmani
- **Angular Fundamentals** - https://angular.io/guide/architecture