GUIDA OPERATIVA ESAME - PRATICA CARDIN

STRUTTURA ESAME

Formato: 3 esercizi

- Esercizio 1 (5-7 punti): Diagramma classi + pattern multipli da testo descrittivo
- Esercizio 2 (2-3 punti): Diagramma sequenza oppure pattern specifico
- Esercizio 3 (2-3 punti): Pattern specifico oppure validità sintattica

Tempo: 9-10 minuti totali
Valutazione: **Sintassi UML corretta = prerequisito** (errore = 0 punti), poi completezza pattern

---

STRATEGIA LETTURA TESTO

FASE 1: Prima Lettura (1 minuto)

Cercare **PAROLE CHIAVE** che indicano pattern:

Parola Chiave	Pattern Indicato
"remoto", "remota", "come se fosse locale"	**PROXY** (remote proxy)
"informato", "notificato", "avvisato", "quando disponibile"	**OBSERVER**
"algoritmo", "strategia", "scelto", "modalità"	**STRATEGY**
"formati diversi", "filesystem differenti", "adattare"	**ADAPTER**
"decorare", "aggiungere funzionalità", "wrapper"	**DECORATOR**
"interfaccia unificata", "semplificare accesso"	**FACADE**
"una sola istanza", "singola istanza"	**SINGLETON**
"famiglia prodotti", "configurabile"	**ABSTRACT FACTORY**
"promise", "futuro", "asincrono", "callback"	**OBSERVER** (promessa=subject)
"chunk", "pacchetti", "stream"	**OBSERVER** (stream)
"virtuale", "astrazione", "rappresentazione astratta"	**PROXY** (virtual) o **FACADE**
"incapsula richiesta", "azione come oggetto", "undo/redo"	**COMMAND**
"algoritmo scheletro", "passi comuni + specifici"	**TEMPLATE METHOD**

Parola Chiave	Pattern Indicato
"Model View Controller", "separazione UI", "notifica View"	**MVC**

FASE 2: Seconda Lettura (1 minuto)

Identificare **COMPONENTI** e **RELAZIONI**:

- Chi fa cosa? (classi principali)
- Chi dipende da chi? (relazioni)
- Chi crea chi? (composizione vs aggregazione)
- Chi viene sostituito? (strategy, adapter)

PATTERN DA TESTO: RICONOSCIMENTO OPERATIVO

OBSERVER (80% probabilità esame)

SEGNALI NEL TESTO:

- "viene **informato** quando..."
- "**notifica** quando disponibile"
- "**avvisato** in caso di..."
- "**stream** di eventi"
- "**promise** / futuro"
- "**asincrono**"

ESEMPIO TESTO:

> "Ogniqualvolta un nuovo chunk è disponibile durante streaming, l'applicazione client ne viene opportunamente **informata**."

IDENTIFICAZIONE RUOLI:

- **Subject**: chi genera eventi (es. Server, StreamSource, EventEmitter)
- **Observer**: chi riceve notifiche (es. Client, Listener, Handler)

CHECKLIST DISEGNO:

```
[Subject]
<<interface>>
+attach(Observer)        ← OBBLIGATORIO
+detach(Observer)        ← OBBLIGATORIO
+notify()                ← OBBLIGATORIO
    △
```

```
    |
[ConcreteSubject]
-observers: List<Observer>  ← lista observer
+notify() {
  for (o : observers)
    o.update()
}

[Observer]
<<interface>>
+update()                ← OBBLIGATORIO
    △
    |
[ConcreteObserver]
+update() {
  // reazione
}
```

ERRORE COMUNE: Usare ◆ (composizione) tra Subject e Observer
CORRETTO: Usare ◇ (aggregazione) o semplice → (associazione)

PROXY (70% probabilità esame)

SEGNALI NEL TESTO:

- "**remoto** / remota"
- "**come se fosse locale**"
- "**dislocati** su nodi diversi"
- "**rappresenta** oggetto in altro spazio"
- "**tramite** tra X e Y"

ESEMPIO TESTO:

> "L'invio del file viene effettuato verso il server, **come se esso stesse eseguendo sul medesimo host**."

IDENTIFICAZIONE RUOLI:

- **RealSubject**: oggetto reale (es. RemoteServer, RealService)
- **Proxy**: intermediario (es. ServerProxy, LocalProxy)
- **Subject**: interfaccia comune

CHECKLIST DISEGNO:

COMPOSIZIONE: Proxy ◆→ RealSubject (Proxy crea/gestisce real)

VARIANTI:

- Remote Proxy: comunicazione rete
- Virtual Proxy: lazy initialization
- Protection Proxy: controllo accessi

STRATEGY (60% probabilità esame)

SEGNALI NEL TESTO:

- "**algoritmo** scelto"
- "diverse **strategie**"
- "**modalità** / quality"
- "**decoder** / encoder"
- "**sconti** applicati"

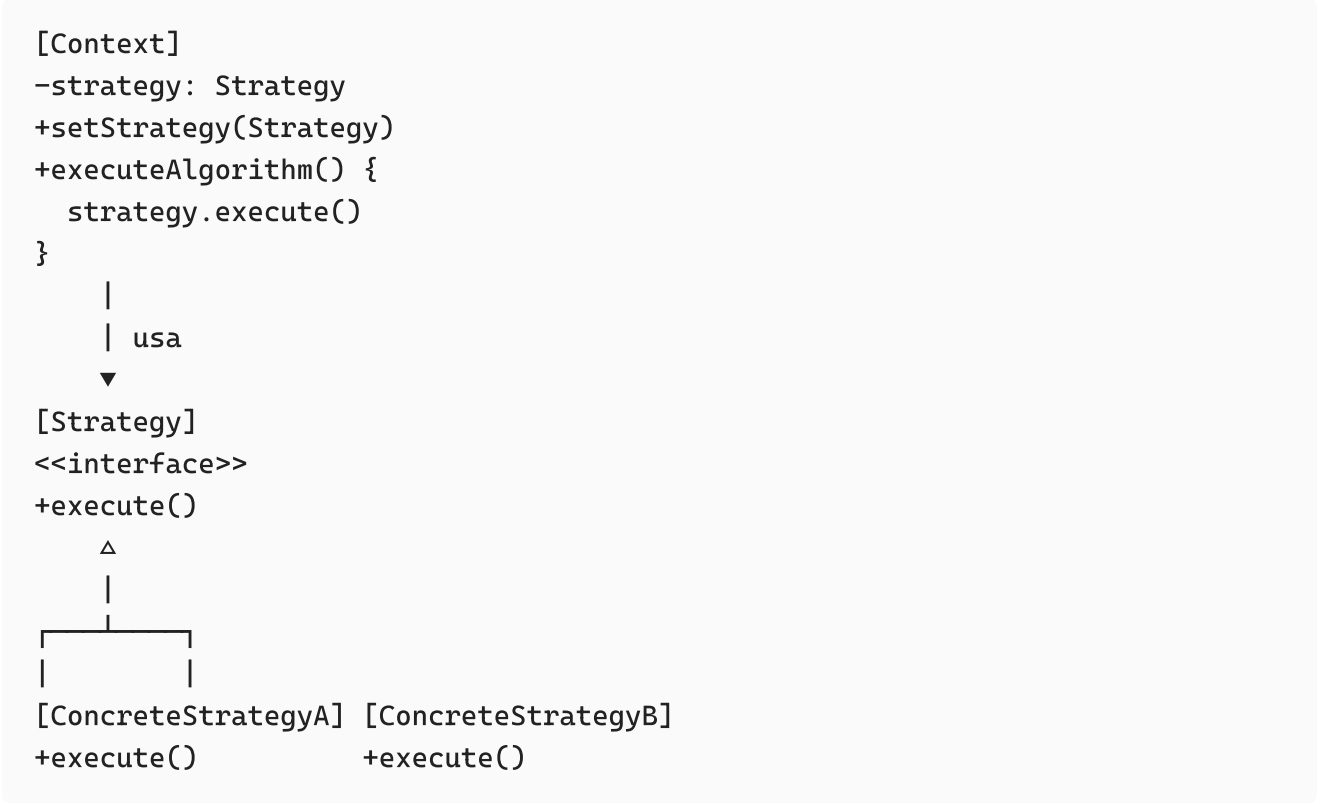
ESEMPIO TESTO:

> "I chunk vengono decodificati in istruzioni per scheda audio da un **algoritmo** opportunamente **scelto**."
> "Un brano può essere riprodotto con qualità **standard** (192Kbps) oppure **elevata** (320Kbps)."

IDENTIFICAZIONE RUOLI:

- **Strategy**: interfaccia algoritmo (es. Decoder, DiscountStrategy)
- **ConcreteStrategy**: implementazioni (es. StdQualityDecoder, HighQualityDecoder)
- **Context**: chi usa strategy (es. AudioPlayer, CashRegister)

CHECKLIST DISEGNO:



COMPOSIZIONE: Context ◇→ Strategy (aggregazione, strategy passata)

ADAPTER (50% probabilità esame)

SEGNALI NEL TESTO:

- "**adattare** / adatta"
- "**formati diversi**" (MP4, MKV, MOV)
- "**filesystem** differenti" (ext3, NTFS)
- "**libreria** utilizza X, applicazione usa Y"
- "**array di byte** → chunk"

ESEMPIO TESTO:

> "La libreria utilizza un semplice **array di byte**, l'applicazione **adatta** tale informazione alla modalità lettura a **chunk**."

IDENTIFICAZIONE RUOLI:

- **Adaptee**: classe da adattare (es. FileReader, ByteArrayReader)
- **Target**: interfaccia desiderata (es. ChunkReader)
- **Adapter**: converte (es. ByteToChunkAdapter)

CHECKLIST DISEGNO (Object Adapter - preferito):

```
[Client] --> [Target]
              <<interface>>
              +request()
                  △
                  |
              [Adapter]  ◇----> [Adaptee]
              +request() {       +specificRequest()
                adaptee.specificRequest()
              }
```

COMPOSIZIONE: Adapter ◇→ Adaptee (aggregazione)

Class Adapter (se richiesto esplicitamente):

```
[Client] --> [Target]
              <<interface>>
                  △
                  |
              [Adapter]
                  |
                  | extends
                  ▼
              [Adaptee]
```

DECORATOR (30% probabilità esame)

SEGNALI NEL TESTO:

- Framework con "wrapping" (es. Java I/O)
- "**aggiungere** funzionalità"
- Catena oggetti: `new A(new B(new C(...)))`

ESEMPIO TESTO:

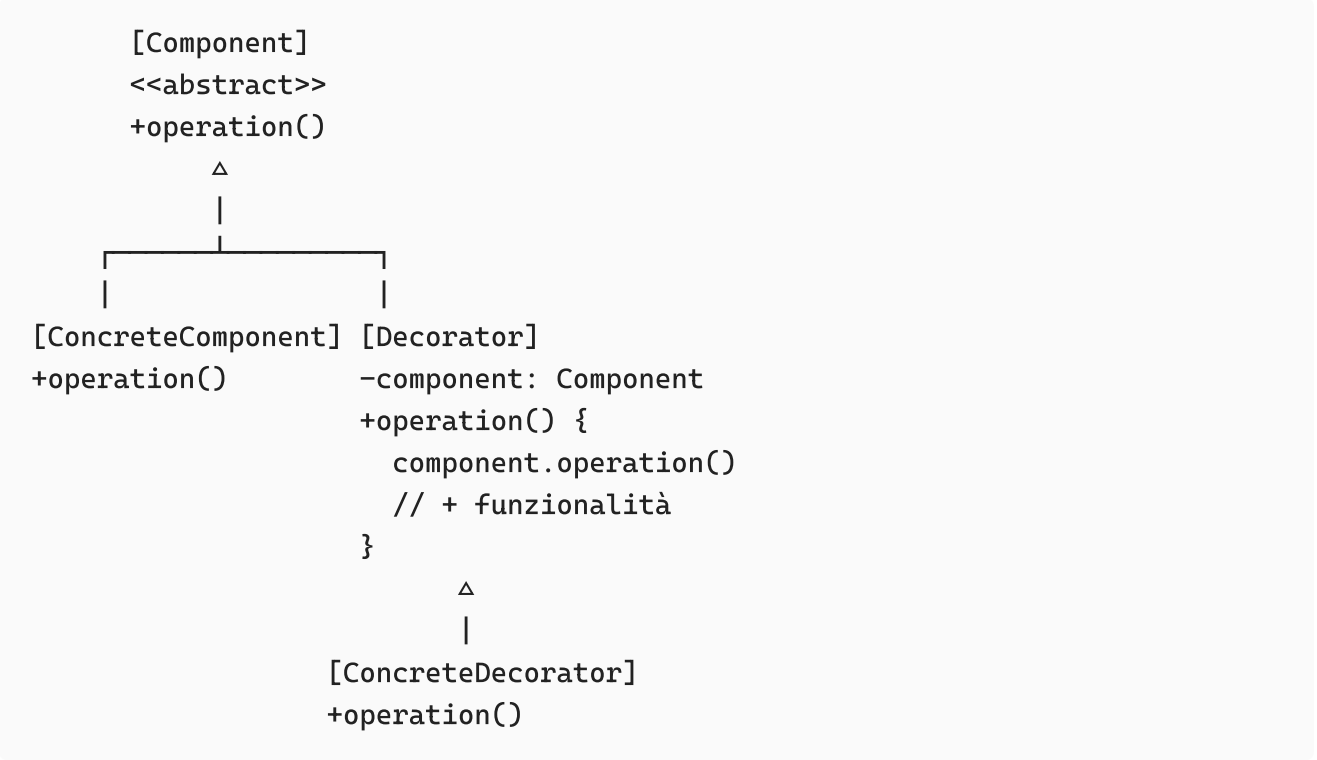
> ```
> FileInputStream fis = new FileInputStream("file.gz");
> BufferedInputStream bis = new BufferedInputStream(fis);
> GzipInputStream gis = new GzipInputStream(bis);
> ObjectInputStream ois = new ObjectInputStream(gis);
> ```

IDENTIFICAZIONE RUOLI:

- **Component**: interfaccia base (InputStream)
- **ConcreteComponent**: implementazione base (FileInputStream)
- **Decorator**: classe decoratore (FilterInputStream)

- **ConcreteDecorator**: decoratori specifici (BufferedInputStream, GzipInputStream)

CHECKLIST DISEGNO:



COMPOSIZIONE: Decorator ◆⟶ Component (composizione, decorator contiene component)

COMMAND (20% probabilità esame)

SEGNALI NEL TESTO:

- "**incapsula** richiesta"
- "**azione** come oggetto"
- "**queue** / coda richieste"
- "**undo** / redo"
- "**macro** / batch comandi"
- "**invoker** esegue comando"

ESEMPIO TESTO:

> "Il sistema deve supportare operazioni di undo/redo. Ogni azione utente viene **incapsulata** come comando che può essere **eseguito** e **annullato**."

IDENTIFICAZIONE RUOLI:

- **Command**: interfaccia comando (es. Command, Action)
- **ConcreteCommand**: comandi specifici (es. CopyCommand, PasteCommand)

- **Receiver**: chi fa realmente l'azione (es. TextEditor)
- **Invoker**: chi esegue comando (es. Button, MenuItem)
- **Client**: crea comando e lo associa a invoker

CHECKLIST DISEGNO:

```
[Client] --> [Invoker]
              -command: Command
              +setCommand(Command)
              +executeCommand() {
                command.execute()
              }
                   |
                   | usa
                   ▼
              [Command]
              <<interface>>
              +execute()
              +undo()  ← opzionale se richiesto
                   △
                   |
            [ConcreteCommand]
            -receiver: Receiver
            -state: Object  ← per undo se necessario
            +execute() {
              receiver.action()
            }
            +undo() {
              receiver.undoAction()
            }
                   |
                   | usa
                   ▼
              [Receiver]
              +action()
              +undoAction()
```

COMPOSIZIONE: ConcreteCommand ◇→ Receiver (aggregazione, receiver iniettato)

VARIANTI:

1. **Command semplice** (solo execute):

```
interface Command {
    void execute();
}

class PrintCommand implements Command {
```

```
    private Printer printer;
    private String text;

    public PrintCommand(Printer p, String t) {
        this.printer = p;
        this.text = t;
    }

    public void execute() {
        printer.print(text);
    }
}
```

2. **Command con undo**:

```
interface Command {
    void execute();
    void undo();
}

class InsertTextCommand implements Command {
    private TextEditor editor;
    private String text;
    private int position;

    public void execute() {
        editor.insertAt(position, text);
    }

    public void undo() {
        editor.deleteAt(position, text.length());
    }
}
```

3. **Macro Command** (composto):

```
class MacroCommand implements Command {
    private List<Command> commands = new ArrayList<>();

    public void add(Command c) {
        commands.add(c);
    }

    public void execute() {
        for (Command c : commands) {
            c.execute();
        }
    }
```

```
    public void undo() {
        for (int i = commands.size()-1; i >= 0; i--) {
            commands.get(i).undo();
        }
    }
}
```

QUANDO SI USA:

- Undo/redo operations
- Transazioni (batch operations)
- Queue di richieste
- Logging operazioni
- Callback decoupling (button → azione)

ERRORE COMUNE: Confondere Command con Strategy

- **Strategy**: diversi ALGORITMI, stesso obiettivo
- **Command**: diverse AZIONI, obiettivi diversi

TEMPLATE METHOD (15% probabilità esame)

SEGNALI NEL TESTO:

- "**algoritmo scheletro**"
- "**passi comuni** + passi specifici"
- "**sequenza fissa**, dettagli variabili"
- "**hook methods**"
- "**abstract** operations"

ESEMPIO TESTO:

> "Il processo di importazione dati ha passi comuni (apertura file, validazione, chiusura), ma la **fase di parsing** varia per formato (CSV, JSON, XML)."

IDENTIFICAZIONE RUOLI:

- **AbstractClass**: definisce template + operazioni astratte
- **ConcreteClass**: implementa operazioni specifiche

CHECKLIST DISEGNO:

```
[AbstractClass]
+templateMethod() {          ← FINAL (non override)
  step1()
  step2()  ← abstract
  step3()
  hook()   ← hook (default vuoto)
}
+step1()
#step2(): abstract            ← protected abstract
+step3()
#hook()                       ← protected hook (opzionale)
    △
    |
[ConcreteClass]
#step2() {
  // implementazione specifica
}
#hook() {
  // override se necessario
}
```

EREDITARIETÀ: ConcreteClass ——▷ AbstractClass

CODICE ESEMPIO:

```
abstract class DataImporter {
    // Template Method - FINAL
    public final void importData(String file) {
        openFile(file);
        validateFormat();
        parseData();      // abstract - varia
        transformData();  // abstract - varia
        saveToDatabase();
        closeFile();
        afterImport();    // hook - opzionale
    }

    private void openFile(String file) { /* comune */ }
    private void validateFormat() { /* comune */ }
    protected abstract void parseData();      // DEVE implementare
    protected abstract void transformData();  // DEVE implementare
    private void saveToDatabase() { /* comune */ }
    private void closeFile() { /* comune */ }

    protected void afterImport() {  // Hook - default vuoto
        // Sottoclasse può override se serve
    }
}
```

```
class CSVImporter extends DataImporter {
    @Override
    protected void parseData() {
        // parsing CSV specifico
    }

    @Override
    protected void transformData() {
        // transform CSV → domain objects
    }

    @Override
    protected void afterImport() {
        // log CSV import
    }
}

class JSONImporter extends DataImporter {
    @Override
    protected void parseData() {
        // parsing JSON specifico
    }

    @Override
    protected void transformData() {
        // transform JSON → domain objects
    }
}
```

DUE VARIANTI:

1. CLASSICA (con ereditarietà):

```
[AbstractAlgorithm]
+templateMethod() { ... }
#primitiveOp1(): abstract
    △
    |
[ConcreteAlgorithm]
#primitiveOp1() { ... }
```

Usa: **ereditarietà** (——▷)

2. CON COMPOSIZIONE (se richiesto esplicitamente):

```
[Algorithm]
-strategy: Strategy
```

```
+algorithm() {
  // passi comuni
  strategy.specificStep()
  // altri passi comuni
}
    |
    | usa
    ▼
[Strategy]
<<interface>>
+specificStep()
```

Usa: **composizione** (◇—>)

Differenza:

- Ereditarietà: sottoclassi cambiano SOLO passi specifici, algoritmo fisso
- Composizione: algoritmo riceve strategy, più flessibile

QUANDO SI USA:

- Algoritmo con sequenza fissa ma passi variabili
- Codice comune da riusare (DRY)
- Framework dove utente estende
- Hollywood Principle: "Don't call us, we'll call you"

ESEMPI REALI:

- Java: `InputStream.read()` template, sottoclassi implementano read specifico
- Testing frameworks: `setUp()` → `runTest()` → `tearDown()`
- Game loops: `initialize()` → `update()` → `render()`

ERRORE COMUNE: Confondere con Strategy

- **Template Method**: sequenza FISSA, passi variabili, **ereditarietà**
- **Strategy**: algoritmo INTERO variabile, **composizione**

DIFFERENZA VISIVA:

```
Template Method:
abstract class A {
  final run() {   ← fisso
    step1()
    step2()       ← variabile
    step3()
  }
}
```

```
Strategy:
class A {
  strategy: S
  run() {
    strategy.execute()  ← tutto variabile
  }
}
```

MVC PATTERN (15% probabilità esame)

SEGNALI NEL TESTO:

- "**Model View Controller**"
- "**separazione UI da business logic**"
- "**push model** / **pull model**"
- "View **notificata** quando Model cambia"

ESEMPIO TESTO:

> "Il sistema usa architettura MVC. Quando il **Model** cambia, la **View** viene **aggiornata** automaticamente. Il **Controller** gestisce input utente."

IDENTIFICAZIONE RUOLI:

- **Model**: business logic e dati
- **View**: presentazione UI
- **Controller**: gestisce input, aggiorna Model

CHECKLIST DISEGNO (Push Model - più comune):

```
        [Model]  ◄──────────┐
        -data               │ modifica
        +getData()          │
        +setData()          │
           △                │
           | (Subject)      │
           |                │
    [Observer pattern]      │
           |                │
           ▼                │
        [View]          [Controller]
        +update()       -model: Model
        +display()      -view: View
                        +handleInput() {
```

```
                        model.setData(...)
                    }
```

RELAZIONI:

- Model è **Subject** (Observer pattern)
- View è **Observer** (Observer pattern)
- Controller **modifica** Model (→)
- View **legge** Model (→)

DUE VARIANTI MVC:

1. PUSH MODEL (Model spinge dati a View):

```
[Model]
+notifyObservers(data) {
  for (observer : observers) {
    observer.update(data)  ← passa dati
  }
}

[View]
+update(data) {
  this.data = data  ← riceve dati
  display()
}
```

Pro: View sempre aggiornata
Contro: Coupling Model→View (Model sa cosa mandare)

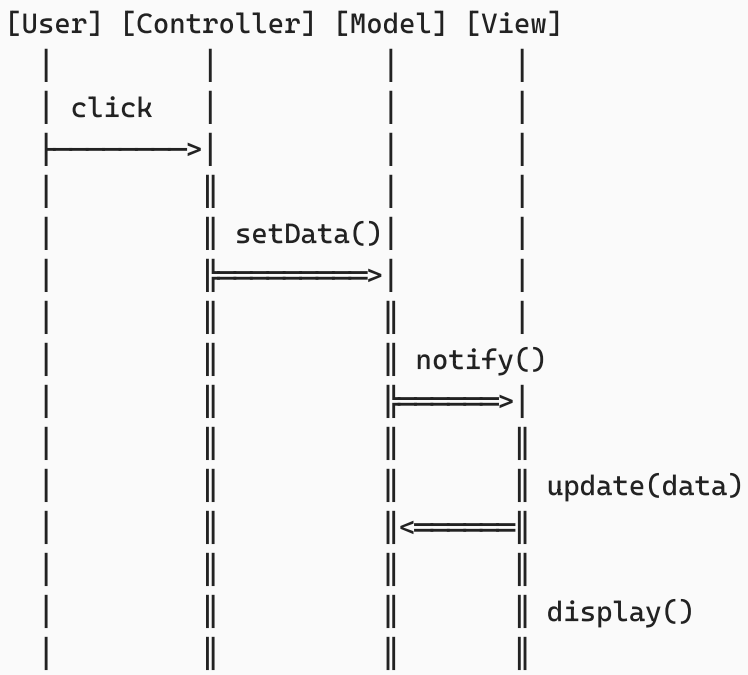
2. PULL MODEL (View chiede dati a Model):

```
[Model]
+notifyObservers() {
  for (observer : observers) {
    observer.update()  ← NO dati
  }
}

[View]
+update() {
  data = model.getData()  ← chiede dati
  display()
}
```

Pro: Decoupling (Model non sa cosa View serve)
Contro: View deve sapere cosa chiedere

DIAGRAMMA SEQUENZA TIPICO (Push Model):



DOPPIA ISTANZA OBSERVER:

MVC classico usa Observer **DUE VOLTE**:

1. **Model** → **View**: quando Model cambia, View aggiorna
2. **Model** → **Controller**: quando Model cambia, Controller aggiornato (meno comune)

Oppure:

1. **Model** → **View**: dati cambiano
2. **View** → **Controller**: input utente (ma qui Controller è listener, non observer tipico)

CODICE ESEMPIO:

```
// Model (Subject)
class Model extends Observable {
    private int counter = 0;

    public void increment() {
        counter++;
        setChanged();
        notifyObservers(counter);  // Push
    }

    public int getCounter() { return counter; }
}

// View (Observer)
class View implements Observer {
```

```
    private Model model;

    public View(Model m) {
        this.model = m;
        m.addObserver(this);
    }

    @Override
    public void update(Observable o, Object data) {
        // Push model - data passato
        System.out.println("Counter: " + data);
    }
}

// Controller
class Controller {
    private Model model;
    private View view;

    public Controller(Model m, View v) {
        this.model = m;
        this.view = v;
    }

    public void userClickedButton() {
        model.increment();  // Model notifica View automaticamente
    }
}
```

QUANDO SI USA:

- Applicazioni GUI (desktop, web)
- Separazione presentation logic da business logic
- Multiple views per stesso Model
- Testabilità (Model isolato da UI)

VARIANTI MODERNE:

- **MVP** (Model-View-Presenter): Presenter media tutto, View passiva
- **MVVM** (Model-View-ViewModel): Data binding bidirezionale
- **MVC Web**: Controller riceve HTTP, View è template engine

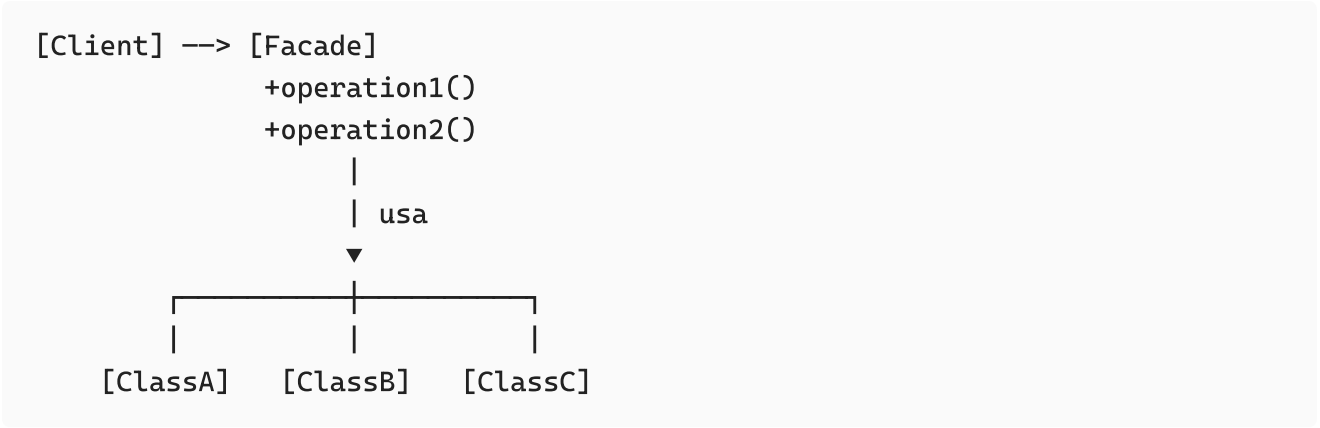
ERRORE COMUNE: View modifica Model direttamente
❌ `view.onButtonClick() → model.setData()`
✓ `view.onButtonClick() → controller.handleClick() → model.setData()`

FACADE (20% probabilità esame)

SEGNALI NEL TESTO:

- "interfaccia **unificata**"
- "**semplificare** accesso"
- "console **amministrazione**"
- Subsystem complesso → interfaccia semplice

CHECKLIST DISEGNO:



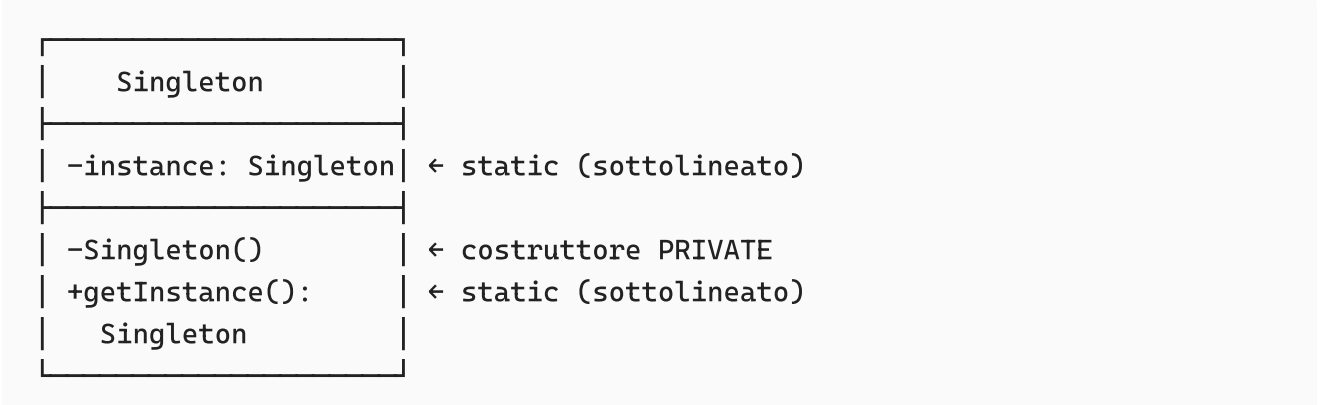
COMPOSIZIONE: Facade ◇→ Subsystems (aggregazione o dipendenza)

SINGLETON (20% probabilità esame)

SEGNALI NEL TESTO:

- "**singola istanza**"
- "**una sola istanza**"
- Abstract Factory spesso è Singleton

CHECKLIST DISEGNO:



ERRORI COMUNI:

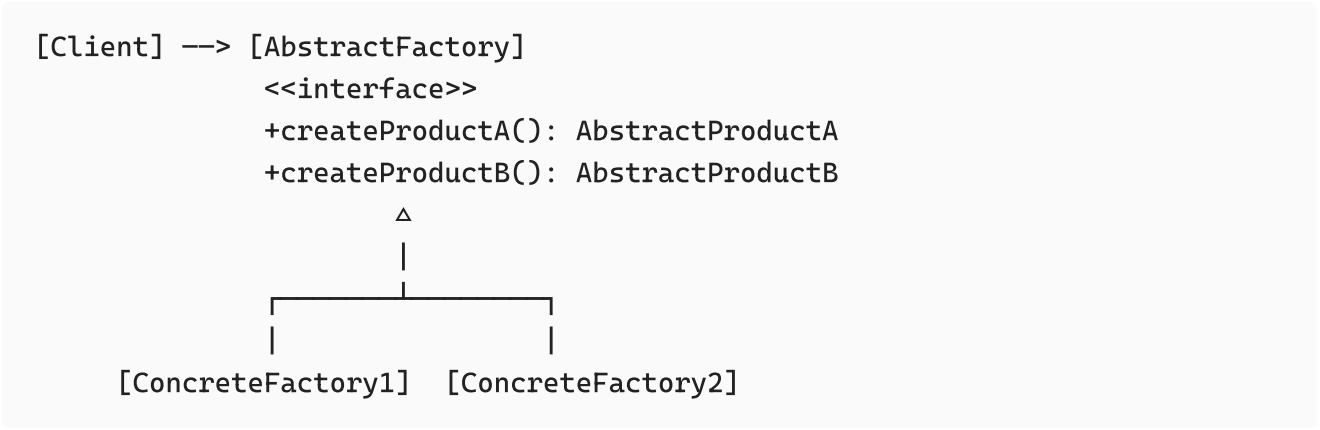
- Costruttore pubblico → SBAGLIATO
- instance non static → SBAGLIATO
- getInstance non static → SBAGLIATO

ABSTRACT FACTORY (15% probabilità esame)

SEGNALI NEL TESTO:

- "famiglia **prodotti**"
- "tipi **diversi**" + "concretizzazioni"
- Sistema **configurabile**

CHECKLIST DISEGNO:



COMBO CON SINGLETON: Factory può essere singleton

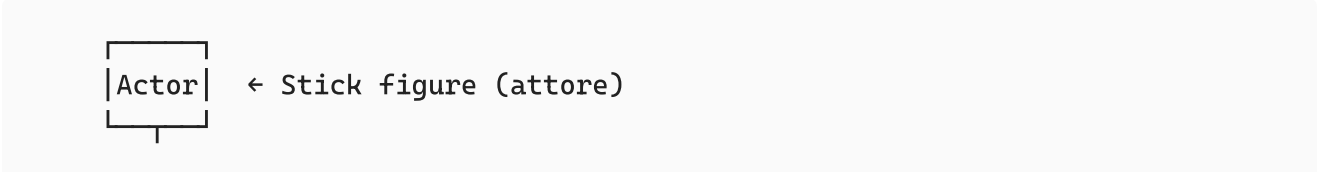
DIAGRAMMI UML AGGIUNTIVI

DIAGRAMMI CASI D'USO (10% probabilità esame)

QUANDO RICHIESTO:

- "Modellare casi d'uso del sistema"
- "Diagramma UML per requisiti funzionali"
- "Interazioni attore-sistema"

ELEMENTI BASE:



```
      │
      │ usa
      ▼
  ┌──────────┐
  │Use Case  │  ← Ellisse (caso d'uso)
  └──────────┘
```

COMPONENTI:

1. **ATTORI** (Actors):
   - Rappresentano: utenti, sistemi esterni, hardware
   - Disegno: Stick figure o box con `<<actor>>`
   - Primari: iniziano use case
   - Secondari: forniscono servizi
2. **CASI D'USO** (Use Cases):
   - Rappresentano: funzionalità sistema
   - Disegno: Ellisse con nome
   - Nome: verbo + oggetto ("Registra Utente", "Genera Report")
3. **RELAZIONI**:

   a) Associazione (——):

   ```
   [Actor] —— (Use Case)
   ```

   Attore usa/partecipa al caso d'uso

   b) Include (- - - ->):

   ```
   (Base UC) - - - -> (Included UC)
            <<include>>
   ```

   Base UC SEMPRE include Included UC
   Esempio: "Acquista Prodotto" include "Verifica Disponibilità"

   c) Extend (<- - - -):

   ```
   (Extension UC) - - - -> (Base UC)
                 <<extend>>
   ```

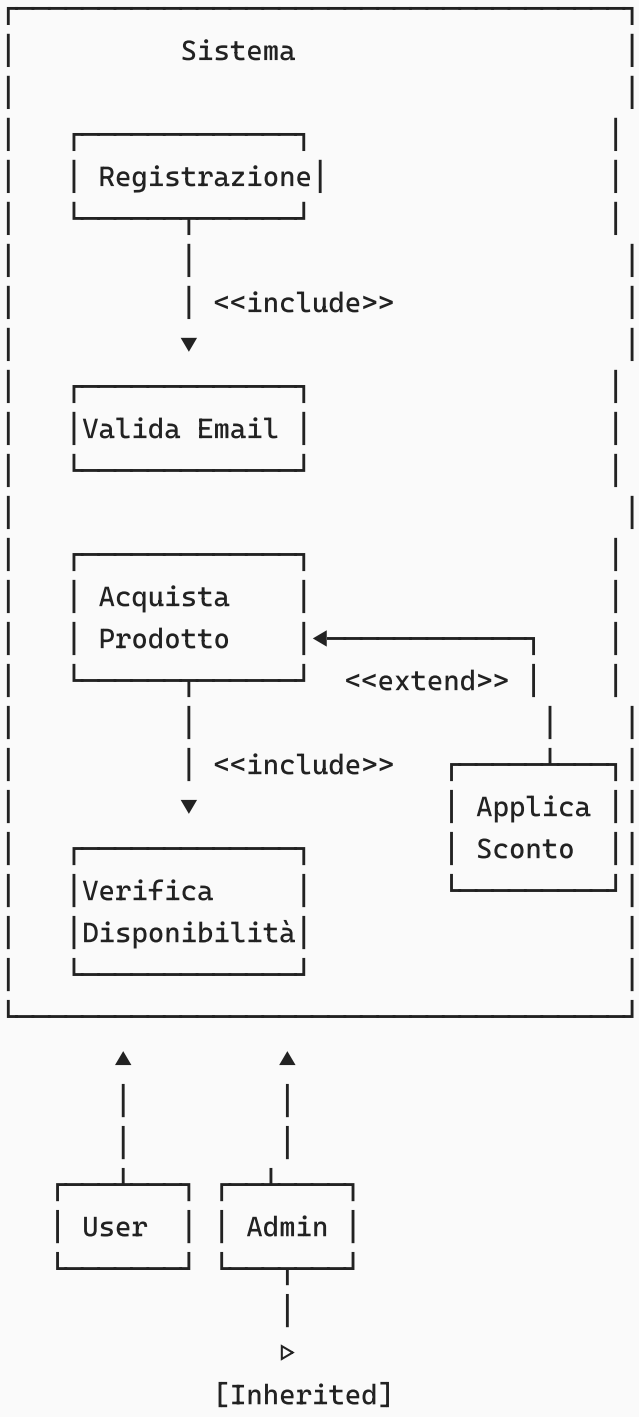
   Extension UC estende Base UC OPZIONALMENTE
   Esempio: "Applica Sconto" extend "Acquista Prodotto"

   d) Generalizzazione (——▷):

```
[Specialized Actor] --▷ [General Actor]
```

Ereditarietà tra attori
Esempio: "Admin" ——▷ "User"

TEMPLATE DIAGRAMMA:



REGOLE PRATICHE:

1. **Granularità**: UC non troppo piccolo, non troppo grande
   - ✓ "Registra Utente" (giusto)
   - ❌ "Clicca Button" (troppo piccolo)

- ❌ "Gestisci Sistema" (troppo grande)

2. **Include vs Extend**:
   - **Include**: comportamento SEMPRE eseguito, fattorizzazione
   - **Extend**: comportamento OPZIONALE, variante
3. **Attori**: rappresentano RUOLI, non persone fisiche
   - ✓ "Cliente", "Admin"
   - ❌ "Mario Rossi"

DIAGRAMMI DI ATTIVITÀ (10% probabilità esame)

QUANDO RICHIESTO:

- "Modellare flusso di lavoro"
- "Diagramma attività per processo"
- "Rappresentare algoritmo/procedura"

ELEMENTI BASE:

1. NODI INIZIALE/FINALE:

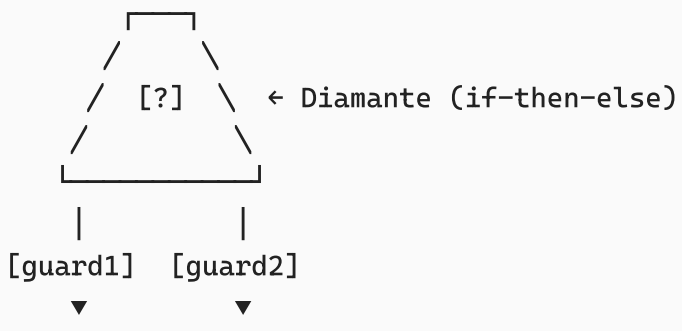
```
● ← Nodo iniziale (start)

⊙ ← Nodo finale (end)
```

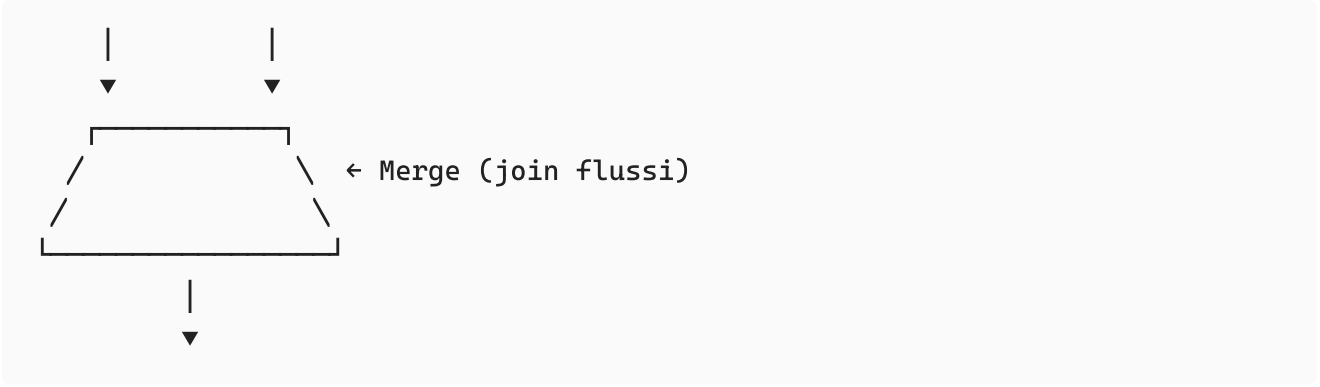
2. ATTIVITÀ:

```
┌──────────────┐
│  Attività    │  ← Rettangolo arrotondato
└──────────────┘
```

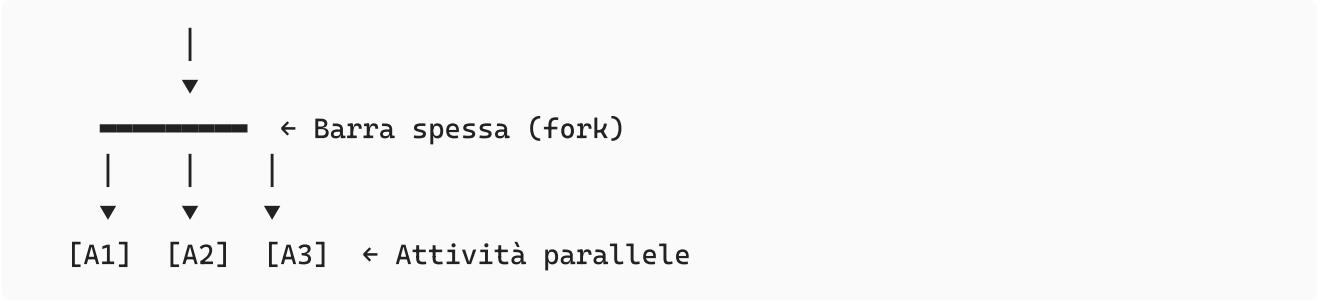
3. DECISIONE (BRANCH):



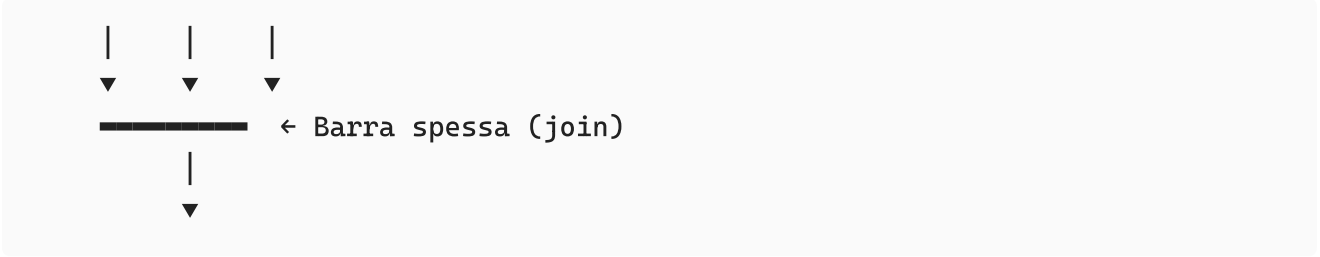
4. MERGE:



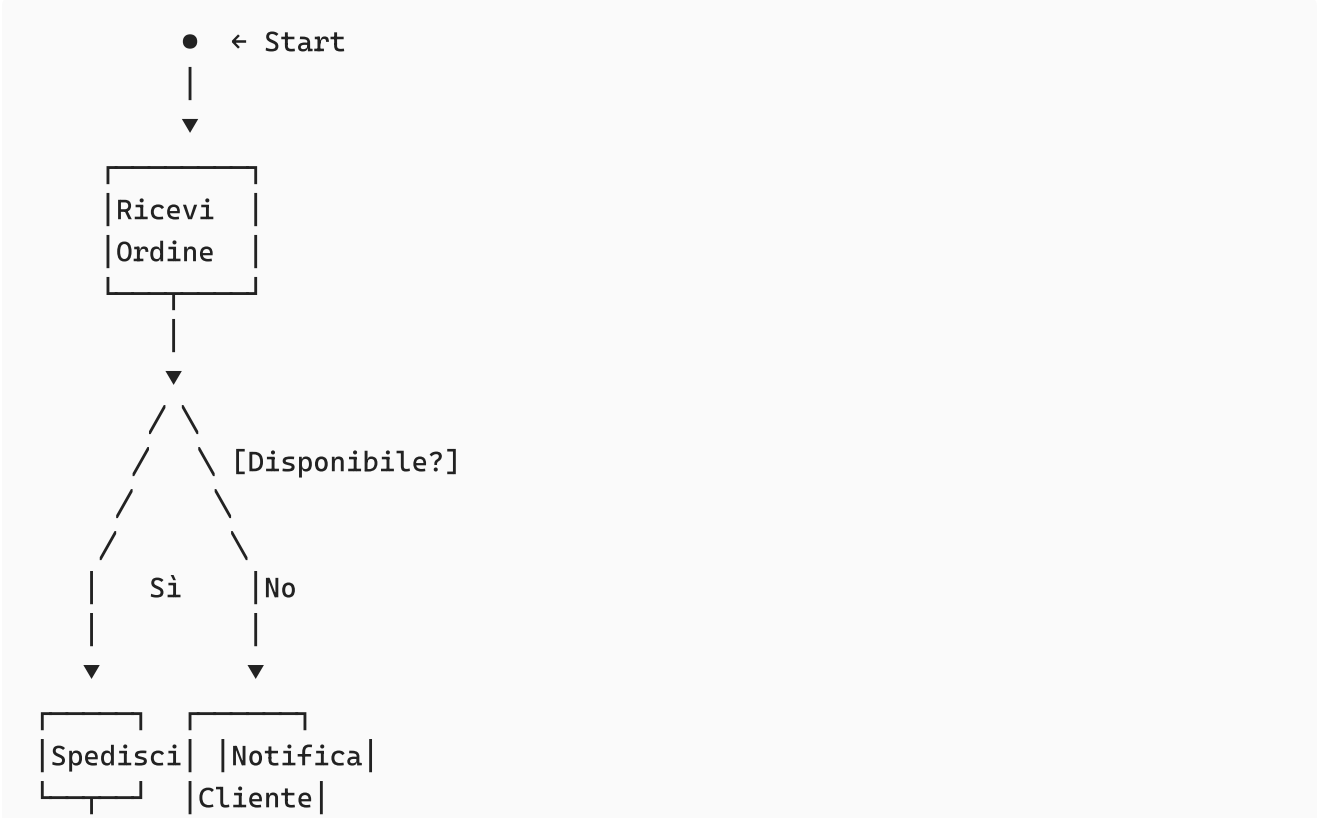
5. FORK (PARALLELISMO):

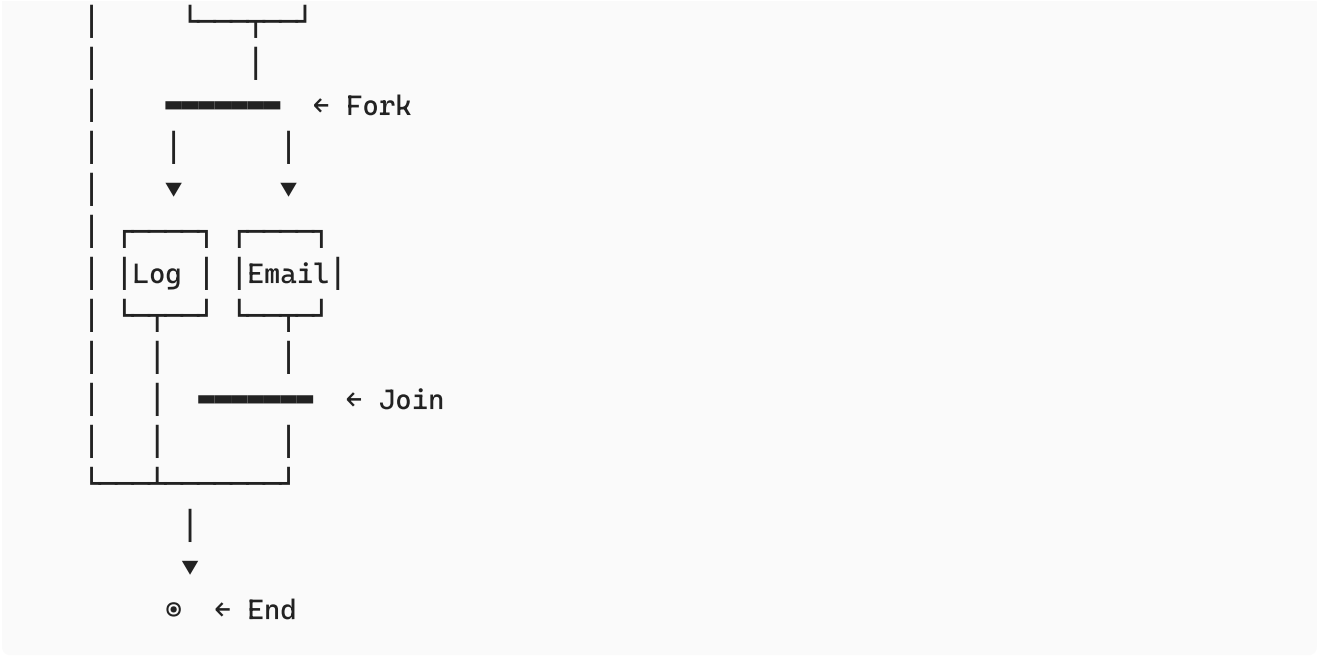


6. JOIN (SINCRONIZZAZIONE):



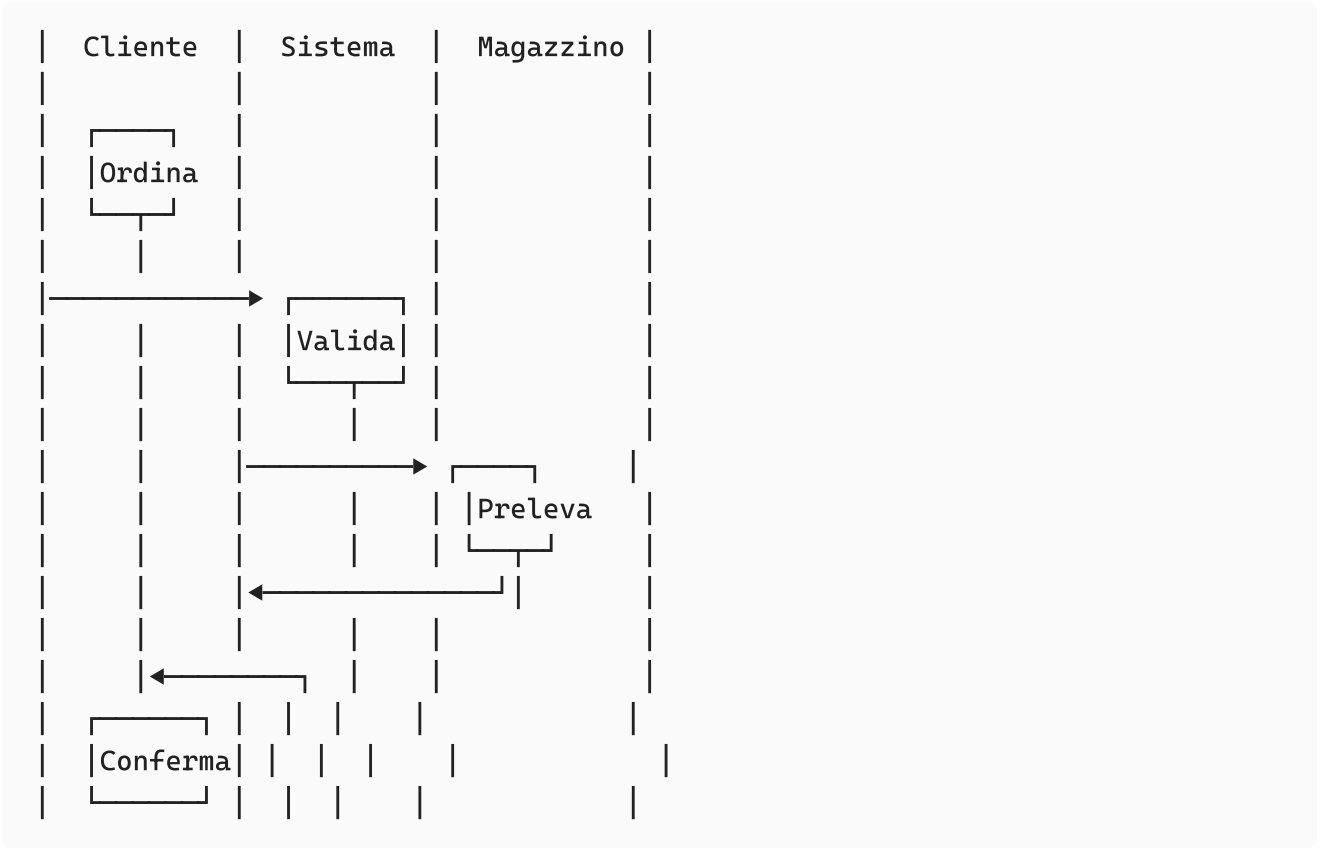
TEMPLATE DIAGRAMMA:





SWIMLANES (CORSIE):

Organizzare attività per responsabilità:

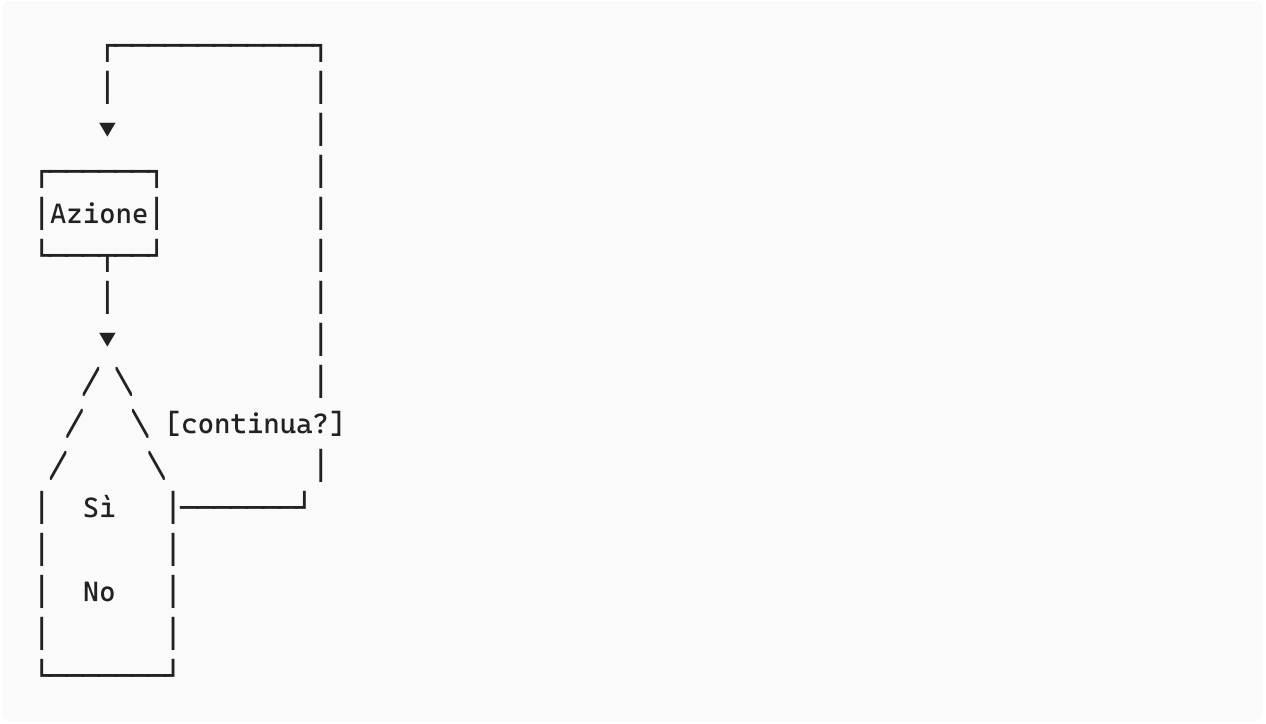


REGOLE PRATICHE:

1. **Ogni fork DEVE avere un join corrispondente**
   - Parallelismo deve convergere
2. **Guard conditions** in decisioni:

```
[x > 0]  ← Condizione esplicita
[else]   ← Ramo alternativo
```

3. **Loop**:




```
**DIFFERENZA ACTIVITY vs SEQUENCE**:

| Aspetto | Activity | Sequence |
|---------|----------|----------|
| Focus | Flusso controllo | Interazione oggetti |
| Tempo | Implicito | Esplicito (verticale) |
| Parallelismo | Fork/join | Messaggi asincroni |
| Quando | Algoritmi, processi | Collaborazioni classi |

**ESEMPIO USO**:
- Activity: "processo ordine" (decisioni, loop, parallelo)
- Sequence: "come Controller chiama Model" (messaggi tra classi)

---

## STRATEGIA DISEGNO (Esercizio 1 - 5-7 punti)

### STEP 1: Identificare Pattern (2 min)
1. Leggi testo 2 volte
2. Sottolinea parole chiave
3. Lista pattern identificati (di solito 2-4)

**ESEMPIO**:
Testo parla di:
- "remoto" → PROXY ✓
- "informato quando chunk disponibile" → OBSERVER ✓
- "algoritmo decoder scelto" → STRATEGY ✓
- "adatta array byte → chunk" → ADAPTER ✓
```

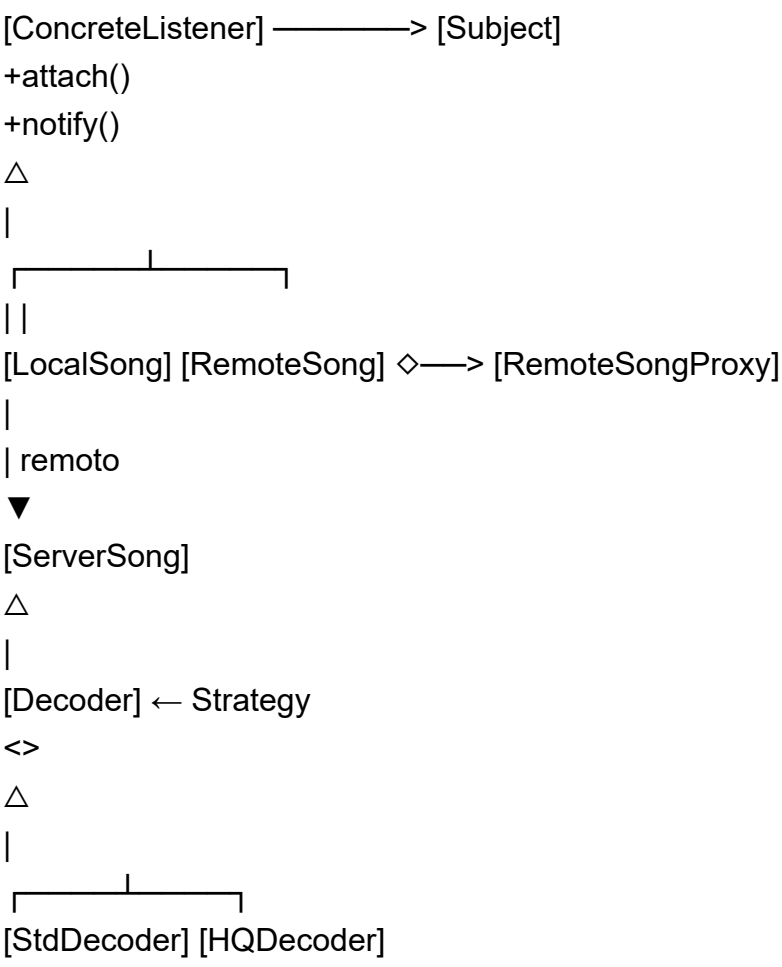
```
### STEP 2: Identificare Classi Principali (1 min)
- Client/User
- Server/Service
- Intermediari
- Dati (Chunk, File, Item...)

### STEP 3: Disegno Schema (4 min)

**ORDINE DISEGNO**:
1. Disegna pattern più complesso (es. Observer) al centro
2. Aggiungi altri pattern intorno
3. Collega con relazioni corrette

**TEMPLATE SPOTIFY (esempio reale esame)**:
```

```
      [Client]
         |
         | usa
         ▼
[ChunkListener]  ◄──────┐
<<interface>>           | osserva
+update()               |
    △                   |
    |                   |
```



[ByteArrayReader] ◀——◇ [ChunkAdapter] ——▷ [ChunkReader]
(Adapter pattern) <>

```
### STEP 4: Verifica Sintassi (1-2 min)

**CHECKLIST**:
- [ ] Observer: attach, detach, notify, update presenti
- [ ] Proxy: Subject interface, RealSubject, Proxy
- [ ] Strategy: Strategy interface, ConcreteStrategies
- [ ] Adapter: Target, Adapter, Adaptee
- [ ] Frecce corrette (◇ aggregazione, ◆ composizione, → associazione, ---▷
realizzazione, —▷ ereditarietà)
- [ ] Visibilità corretta (- private, + public, # protected)
- [ ] Metodi obbligatori pattern presenti
- [ ] Interfacce marcate `<<interface>>`

---

## COME DECIDERE COMPOSIZIONE VS AGGREGAZIONE

### USA ◆ COMPOSIZIONE quando:
- Classe crea (new) l'oggetto internamente
- Oggetto non esiste senza contenitore
- Lifecycle controllato da contenitore

**Codice**:
```java
class University {
    private Department math = new Department(); // ◆
}
```

Nel diagramma: `[University] ◆—> [Department]`

USA ◇ AGGREGAZIONE quando:

- Oggetto passato dall'esterno (injection)
- Oggetto può essere condiviso
- Oggetto esiste indipendentemente

Codice:

```java
class Playlist {
    private List<Song> songs;
    public Playlist(List<Song> s) { // ◇
        this.songs = s;
```
```

```
    }
}
```

Nel diagramma: `[Playlist] ◇–> [Song]`

CASI PATTERN

Pattern	Relazione Tipica	Esempio
Observer	Subject ◇—> Observer	Lista observer gestita
Proxy	Proxy ◆—> RealSubject	Proxy crea real
Strategy	Context ◇—> Strategy	Strategy iniettata
Adapter	Adapter ◇—> Adaptee	Adaptee iniettato
Decorator	Decorator ◆—> Component	Decorator contiene
Facade	Facade ◇—> Subsystems	Dipendenza

REGOLA PRATICA:

- Se nel codice vedi `new` → ◆
- Se vedi parametro costruttore/setter → ◇
- Se usi temporaneamente → ----> (dipendenza)

DIAGRAMMA DI SEQUENZA (Esercizio 2)

ELEMENTI BASE

Attori/Classi:

```
┌─────────┐
│  Actor  │
└─────────┘
     │
     │ lifeline (linea vita)
     │
     ↓
```

Messaggi:

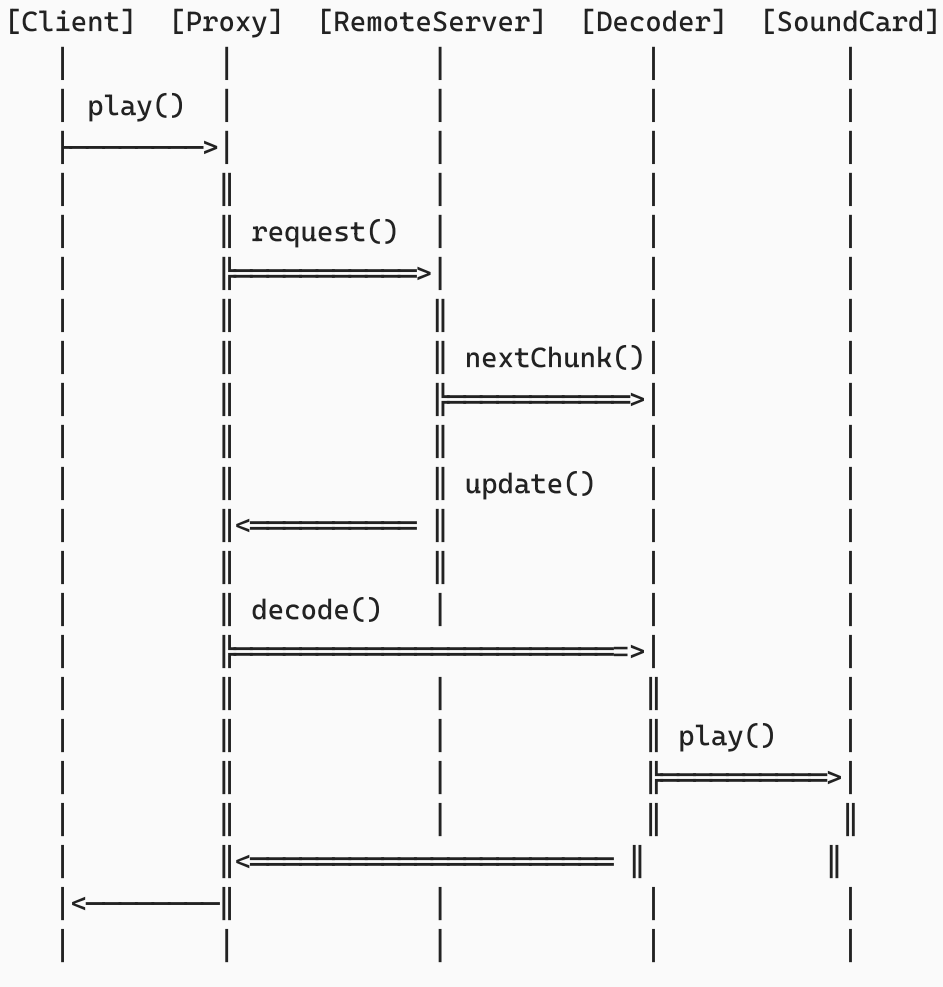
- Sincrono: `------>` (linea continua)
- Asincrono: `- - - ->` (linea tratteggiata)
- Risposta: `<------` (linea tratteggiata indietro)

Attivazione:

```
      │
      ├──> metodo()
      ‖         ← barra verticale = attivazione
      ‖
      ╚═══> return
      │
```

TEMPLATE STANDARD

Scenario: "Arrivo chunk remoto e processamento"



STEP DISEGNO SEQUENZA

1. **Identifica partecipanti** (3-6 classi coinvolte)
2. **Identifica trigger** (chi inizia? spesso Client)
3. **Segui flusso logico** del testo
4. **Aggiungi chiamate remote** (se Proxy)
5. **Aggiungi notifiche** (se Observer)
6. **Aggiungi scelta algoritmo** (se Strategy)

PATTERN SPECIFICI (Esercizio 3 - 2-3 punti)

TIPO A: Disegnare Pattern Dato Nome

Richiesta: "Si disegni diagramma classi del design pattern Observer"

STRATEGIA:

1. Ricorda struttura base pattern (studiata)
2. Disegna tutte classi obbligatorie
3. Aggiungi tutti metodi obbligatori
4. Verifica relazioni corrette

PATTERN PIÙ RICHIESTI:

1. Observer (80%)
2. Abstract Factory (60%)
3. Singleton (40%)
4. Decorator (30%)
5. Proxy (20%)

TIPO B: Validare Sintassi Diagramma

Richiesta: "Il seguente diagramma è sintatticamente valido?"

ERRORI COMUNI OBSERVER:

- ❌ Manca `notify()` in Subject
- ❌ Manca `update()` in Observer
- ❌ Composizione ◆ invece di aggregazione ◇
- ❌ Freccia generalizzazione tratteggiata
- ❌ Subject non ha lista observers

CHECKLIST VALIDAZIONE:

1. Tutti metodi obbligatori presenti?
2. Relazioni corrette (tipo freccia)?
3. Visibilità corretta?
4. Interfacce marcate?
5. Attributi necessari presenti?

TIPO C: Adapter Varianti

Richiesta: "Indicare quale diagramma rappresenta Object Adapter"

RICONOSCIMENTO:

- **Object Adapter**: Adapter ◇—> Adaptee (composizione)
- **Class Adapter**: Adapter ——▷ Adaptee (ereditarietà)

ERRORI CHE FANNO PERDERE PUNTI

ERRORI SINTATTICI (= 0 punti)

1. **Freccia composizione invertita**
   - ❌ `[A] <–◆ [B]`
   - ✓ `[A] ◆–> [B]`
2. **Diamante su entrambi i lati**
   - ❌ `[A] ◇–◇ [B]`
   - ✓ `[A] ◇–> [B]`
3. **Linea tratteggiata per ereditarietà**
   - ❌ `[A] – – ▷ [B]`
   - ✓ `[A] ––▷ [B]`
4. **Freccia aperta per ereditarietà**
   - ❌ `[A] ––> [B]`
   - ✓ `[A] ––▷ [B]` (freccia chiusa vuota)
5. **Metodi pattern mancanti**
   - Observer DEVE avere: attach, detach, notify, update
   - Singleton DEVE avere: -instance (static), -costruttore, +getInstance (static)

ERRORI CONCETTUALI (perdita parziale punti)

1. **Pattern sbagliato per scenario**
   - Testo dice "remoto" → devi usare Proxy
   - Se non lo usi = pattern mancante
2. **Relazioni sbagliate tra pattern**
   - Observer: Subject deve avere LIST di Observer (non singolo)
   - Proxy: Proxy e RealSubject implementano STESSA interfaccia
3. **Nomi classi nonsense**
   - Se testo parla di "chunk" e "server" → usa quei nomi
   - Evita nomi generici (Class1, Class2) quando testo è specifico

---

GESTIONE TEMPO ESERCIZIO PRATICA

TOTALE: 9-10 minuti

Esercizio 1 (5-7 punti): 5-6 minuti

- 2 min: Lettura, identificazione pattern
- 3-4 min: Disegno diagramma

Esercizio 2 (2-3 punti): 2-3 minuti

- 1 min: Identificare flusso
- 1-2 min: Disegno sequenza

Esercizio 3 (2-3 punti): 1-2 minuti

- Pattern specifico: 1.5 min disegno
- Validazione: 30 sec check

Se sei indietro:

- Priorità: Esercizio 1 (vale di più)
- Meglio diagramma incompleto sintatticamente corretto che completo sbagliato
- Se non hai tempo per Esercizio 3, almeno disegna struttura base

---

STUDIO PRE-ESAME

COSA MEMORIZZARE

1. **Struttura 6 pattern principali** (disegno a memoria):
   - Observer (attach, detach, notify, update)
   - Proxy (Subject, Proxy, RealSubject)
   - Strategy (Context, Strategy, ConcreteStrategy)
   - Adapter (Target, Adapter, Adaptee)
   - Decorator (Component, Decorator, ConcreteDecorator)
   - Singleton (instance, getInstance, costruttore privato)
2. **Tabella parole chiave → pattern**
   - Memorizza associazioni (vedi tabella sopra)
3. **Regole composizione vs aggregazione**
   - new interno → ◆
   - injection → ◇
4. **Sintassi frecce UML**
   - ◆ composizione
   - ◇ aggregazione
   - → associazione
   - ----> dipendenza

- ——▷ ereditarietà
- ---▷ realizzazione interfaccia

PRATICA PRE-ESAME

ESERCIZI DA FARE (ultimi 3 giorni):

1. Disegna a memoria i 6 pattern principali (timer 2 min ciascuno)
2. Fai almeno 5 esercizi vecchi appelli completi
3. Per ogni esercizio vecchio: cronometra tempo
4. Identifica errori sintattici in diagrammi sbagliati

SIMULAZIONE FINALE (giorno prima):

- Prendi 3 esercizi vecchi mai visti
- Timer 9 minuti NETTI
- Correggi con soluzioni
- Identifica pattern errori ricorrenti

CHECKLIST FINALE PRIMA DI CONSEGNARE

ESERCIZIO 1 (Diagramma Classi + Pattern)

- [ ] Ho disegnato TUTTI i pattern identificati nel testo?
- [ ] Ogni pattern ha TUTTI i metodi obbligatori?
- [ ] Le frecce sono CORRETTE (tipo e direzione)?
- [ ] I diamanti sono nel lato GIUSTO?
- [ ] Le interfacce sono marcate `<<interface>>` ?
- [ ] Visibilità corretta (- private costruttore Singleton)?
- [ ] Le classi hanno nomi sensati dal testo?

ESERCIZIO 2 (Diagramma Sequenza)

- [ ] Tutti partecipanti necessari presenti?
- [ ] Flusso segue logica testo?
- [ ] Chiamate sincrone (linea continua) vs asincrone (tratteggiata)?
- [ ] Attivazioni (barre verticali) presenti?
- [ ] Pattern rispettati (es. Proxy chiama RealSubject)?

ESERCIZIO 3 (Pattern Specifico)

- [ ] Struttura pattern corretta?
- [ ] Tutti metodi obbligatori?

- [ ] Relazioni corrette?

MINDSET FINALE

CARDIN VALUTA:

1. **Correttezza sintattica UML** (50%): Frecce, diamanti, linee corrette = prerequisito
2. **Completezza pattern** (30%): Tutti metodi obbligatori presenti
3. **Applicazione corretta** (20%): Pattern usati dove richiesto dal testo

NON VALUTA:

- Estetica disegno
- Dimensioni classi
- Colori
- Ordine elementi nel diagramma

ERRORE = 0 PUNTI:

- Sintassi UML sbagliata
- Pattern richiesto mancante
- Metodi obbligatori mancanti

IN SINTESI: Cardin vuole PRECISIONE TECNICA. Sintassi corretta è VITALE. Pattern incompleto ma corretto > pattern completo ma sbagliato sintatticamente.

TABELLA REFERENCE RAPIDA

RELAZIONI UML

Relazione	Simbolo	Quando Usare	Codice
Dipendenza	`----->`	Parametro metodo	`void m(B b)`
Associazione	`-->`	Campo permanente	`B campo;`
Aggregazione	`◇–>`	Injection	`A(B b){this.b=b}`
Composizione	`◆–>`	Crea interno	`B b=new B()`
Ereditarietà	`--▷`	Extends	`class A extends B`
Realizzazione	`---▷`	Implements	`class A implements I`

PATTERN OBBLIGATORI

Pattern	Metodi OBBLIGATORI	Note
Singleton	-instance (static), -costruttore(), +getInstance() (static)	Costruttore PRIVATE
Observer	Subject: attach, detach, notify; Observer: update	Subject ha LIST observer
Proxy	Subject interface, Proxy+RealSubject implementano	Stessa interfaccia
Strategy	Strategy: execute; Context: setStrategy	Strategy iniettata
Adapter	Target interface, Adapter implements Target	Adapter usa Adaptee
Decorator	Component: operation; Decorator: operation (chiama super)	Decorator contiene Component
Command	Command: execute, [undo]; ConcreteCommand usa Receiver	execute() obbligatorio
Template Method	AbstractClass: templateMethod (final), primitiveOps (abstract)	templateMethod NON override
MVC	Model (Subject), View (Observer), Controller	Doppia istanza Observer
Facade	Facade: metodi semplificati; usa Subsystems	Nasconde complessità
Abstract Factory	Factory: createProductA/B; ConcreteFactory implementa	Spesso Singleton

RELAZIONI PATTERN

Pattern	Relazione Tipica	Quando
Observer	Subject ◇—> Observer (lista)	Subject gestisce lista
Proxy	Proxy ◆—> RealSubject	Proxy crea/controlla real
Strategy	Context ◇—> Strategy	Strategy iniettata
Adapter (object)	Adapter ◇—> Adaptee	Adaptee iniettato
Adapter (class)	Adapter ——▷ Adaptee	Ereditarietà
Decorator	Decorator ◆—> Component	Decorator contiene
Command	ConcreteCommand ◇—> Receiver	Receiver iniettato
Template Method	ConcreteClass ——▷ AbstractClass	Ereditarietà
MVC	Model (Subject), View (Observer)	Observer pattern
Facade	Facade ◇—> Subsystems	Aggregazione/dipendenza

ULTIMA RACCOMANDAZIONE: Studia disegnando. 30 minuti al giorno disegnando pattern a memoria > 3 ore leggendo teoria. Cardin vuole PRATICA, non teoria.