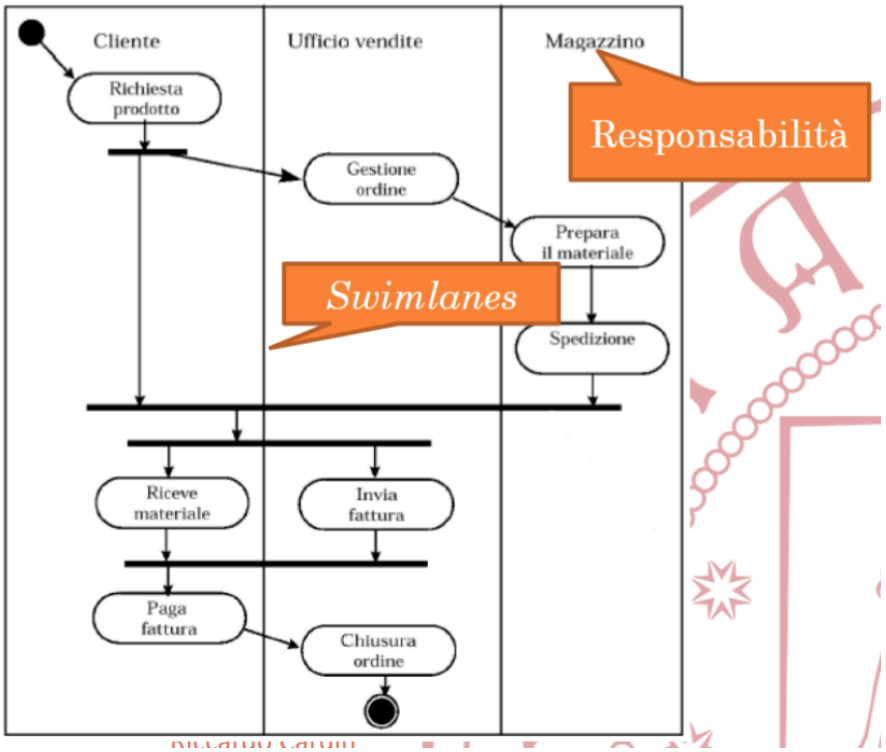
- Forniscono una responsabilità all'esecuzione delle azioni
  - *Swimlanes*



La semantica dei nodi fork/join nei diagrammi di attività UML segue il modello dei token derivato dalle reti di Petri. Analizziamo la situazione specifica che evidenzi.

Semantica Fork/Join:

Un fork node ha un edge in ingresso e multiple uscite, duplicando il token per consentire l'esecuzione concorrente di più percorsi. Un join node ha multiple entrate e un'uscita singola, sincronizzando i flussi concorrenti aspettando che arrivino token da TUTTI gli edge in ingresso.

Il "paradosso" del singolo percorso da fork:

È assolutamente valido avere join e fork combinati nello stesso nodo simbolico. Questo mapping corrisponde a un modello contenente un JoinNode con tutti gli ActivityEdges in ingresso e un edge uscente verso un ForkNode con tutti gli edge uscenti.

Nel tuo diagramma, la sequenza "Richiesta prodotto" → "Riceve materiale"/"Invia fattura" con doppio fork/join ha senso logico preciso:

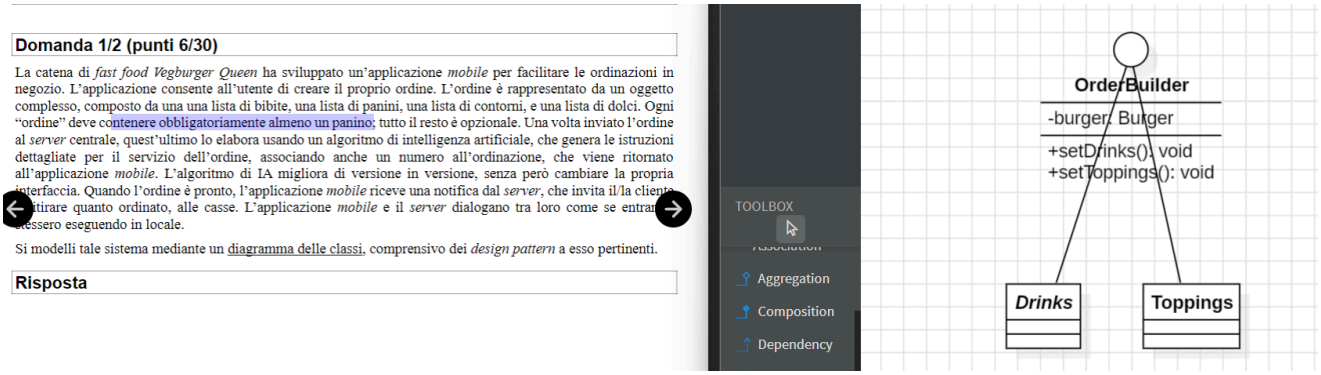
1. **Prima sincronizzazione (join)**: Aspetta il completamento di attività precedenti concorrenti
2. **Fork intermedio**: Crea concorrenza tra "Riceve materiale" e "Invia fattura"
3. **Join finale**: Sincronizza questi due processi prima di procedere

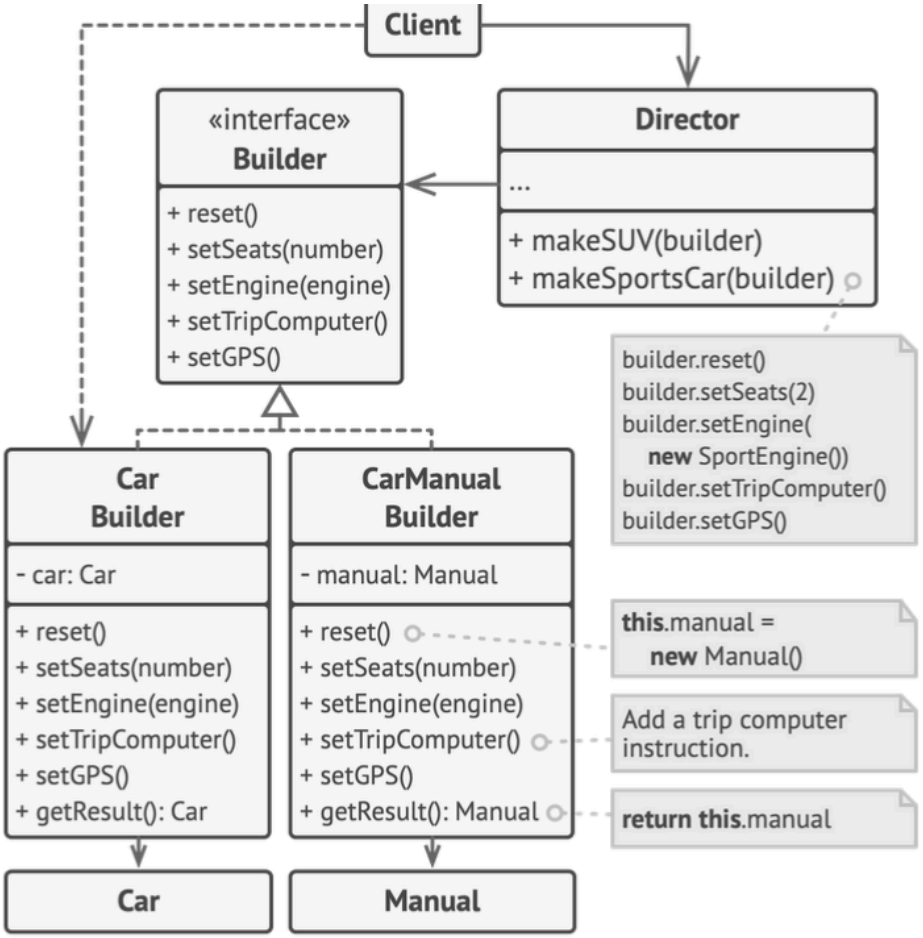
Giustificazione semantica:

Con il fork node, ogni token in ingresso viene duplicato e inizia l'esecuzione su ogni edge uscente. I processi concorrenti vengono poi unificati usando la sincronizzazione (join node).

Il fatto che da un fork parta apparentemente "una sola serie" è un'illusione visiva: in realtà, il fork crea più stati attivi che possono propagarsi indipendentemente, e il join può essere attraversato solo quando tutti gli input sono ricevuti.

La doppia fork/join modella correttamente processi di business dove certe attività devono essere sincronizzate (come preparazione materiale) prima di lanciare processi paralleli (ricezione fisica e fatturazione), che poi richiedono nuova sincronizzazione prima della chiusura ordine.



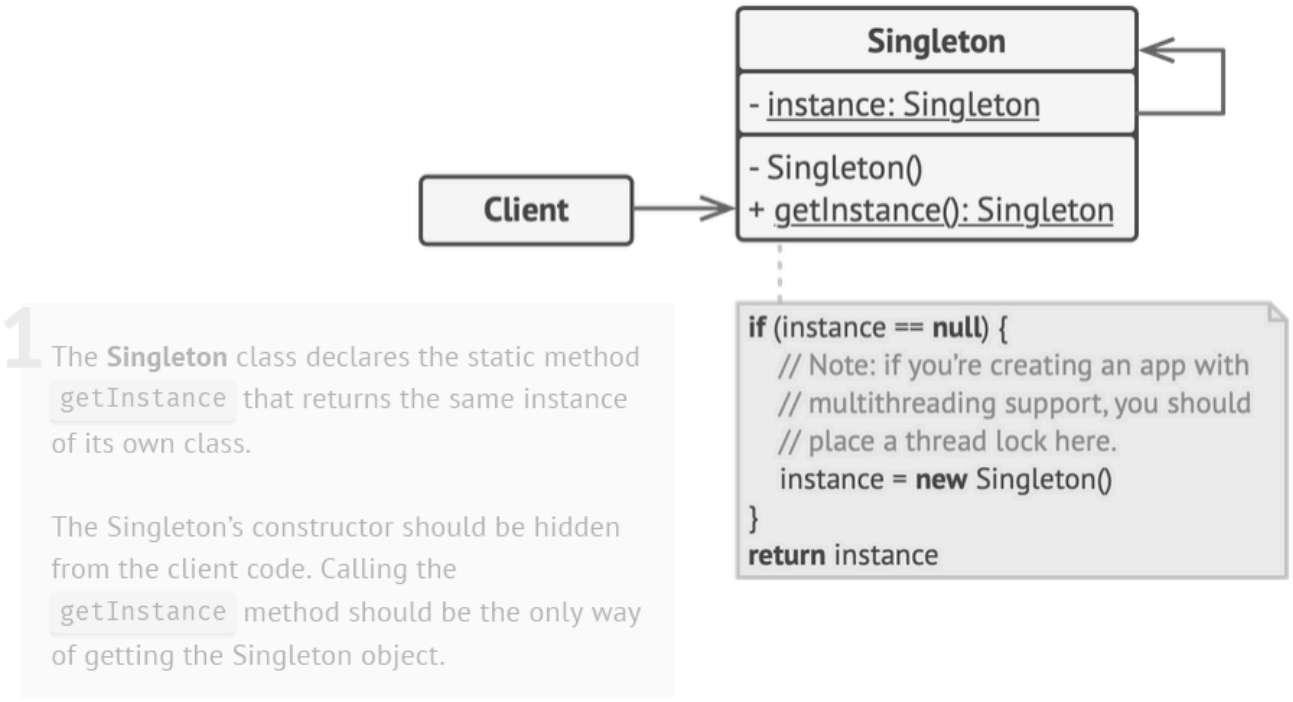


The example of step-by-step construction of cars and the user guides that fit those car models.

Casi strani pattern

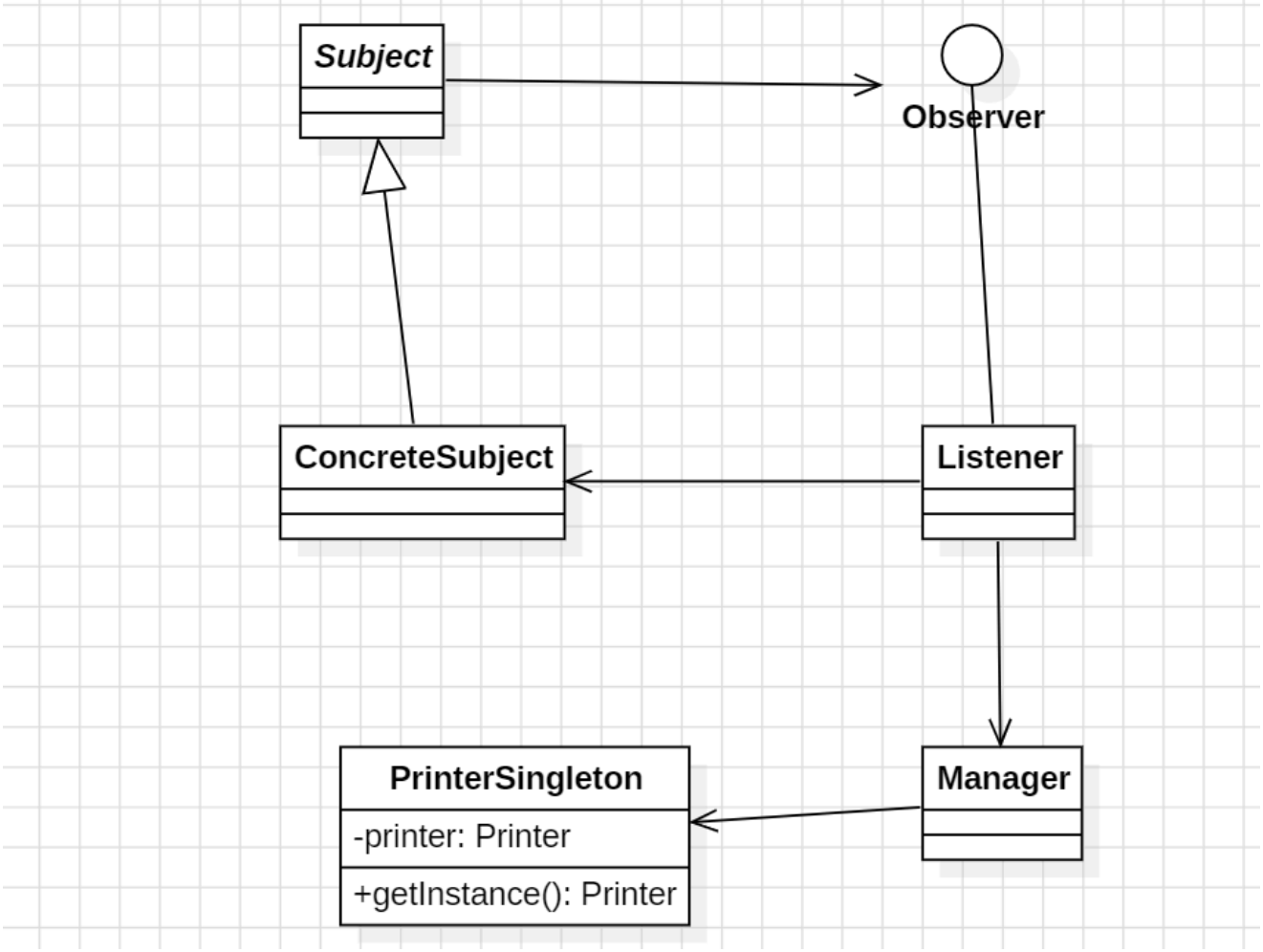
Singleton (Singoletto)

Structure



> Contesto: Applicazione di stampa in cui il driver della stampante non accetta processi in contemporanea ed è instanziato una sola volta globalmente.

Esempio reale:



Classi Template (Esempio VegBurger - Appello 2, 24_25)

Definizione della Classe Template

```
┌──────────────────────────────────┐
│    «TemplateParameter»           │
│         T                        │
└──────────────────────────────────┘
           |
           |
┌──────────────────────────────────┐
│      ListaIngredienti<T>         │
├──────────────────────────────────┤
│ - elementi: T[]                  │
│ - capacitaMax: int               │
│ - obbligatorio: boolean          │
├──────────────────────────────────┤
│ + aggiungi(elemento: T): void    │
│ + rimuovi(indice: int): void     │
│ + ottieniTutti(): T[]            │
```

```
| + isValida(): boolean            |
| + calcolaPrezzoTotale(): double |
└──────────────────────────────────┘
```

Binding Concreti per il Sistema Vegburger Queen

1. Lista Panini (Obbligatoria)

```
ListaIngredienti<Panino>
- obbligatorio = true
- capacitaMax = 10
```

2. Lista Bibite (Opzionale)

```
ListaIngredienti<Bibita>
- obbligatorio = false
- capacitaMax = 5
```

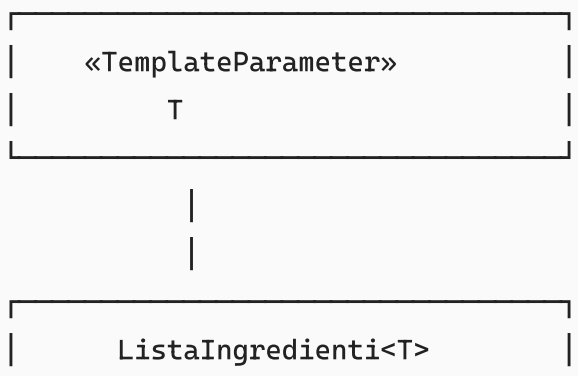
3. Lista Contorni (Opzionale)

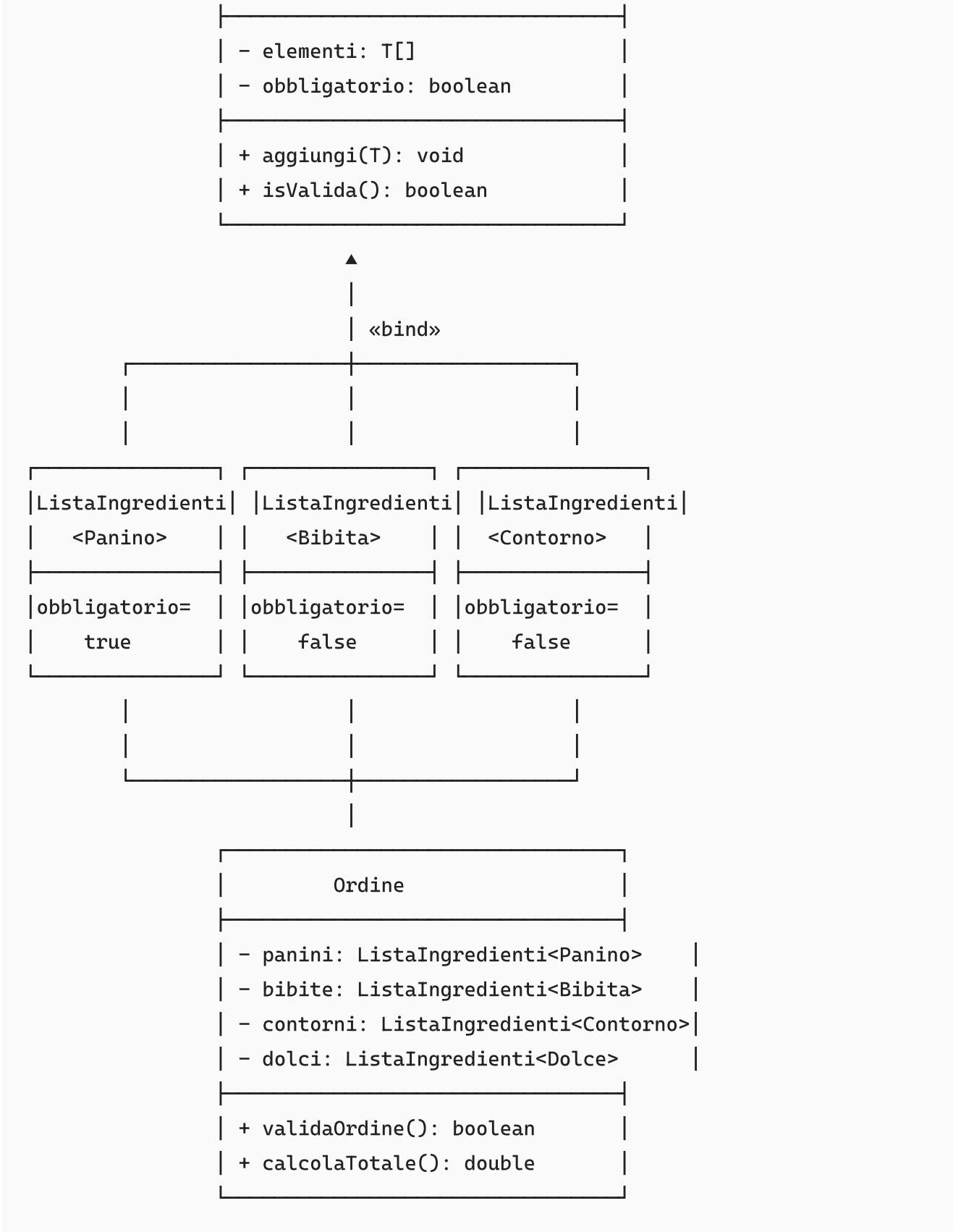
```
ListaIngredienti<Contorno>
- obbligatorio = false
- capacitaMax = 8
```

4. Lista Dolci (Opzionale)

```
ListaIngredienti<Dolce>
- obbligatorio = false
- capacitaMax = 3
```

Diagramma UML Completo del Sistema





Vantaggi del Template nel Contesto Vegburger Queen

Riuso del Codice

Invece di creare quattro classi separate (`ListaPanini` , `ListaBibite` , `ListaContorni` , `ListaDolci`), si usa una singola definizione template che garantisce:

- **Type Safety**: `ListaIngredienti<Panino>` accetta solo oggetti `Panino`
- **Consistenza**: Tutti i tipi di lista hanno la stessa interfaccia
- **Manutenibilità**: Modifiche alla logica di lista si applicano automaticamente a tutti i tipi

Validazione Specifica per Tipo

```
// Esempio di implementazione del metodo isValida()
public boolean isValida() {
    if (obbligatorio && elementi.isEmpty()) {
        return false; // Lista panini deve avere almeno un elemento
    }
    return elementi.size() <= capacitaMax;
}
```

Integrazione con OrderBuilder Pattern

Il template si integra perfettamente con il pattern Builder mostrato nel diagramma originale:

```
OrderBuilder builder = new OrderBuilder();
builder.setPanini(new ListaIngredienti<Panino>(true, 10))
       .setBibite(new ListaIngredienti<Bibita>(false, 5))
       .setContorni(new ListaIngredienti<Contorno>(false, 8));
```

Facade

Definizione del Pattern Facade

Il pattern Facade fornisce **un'interfaccia unica semplificata** per interagire con un sottosistema complesso, nascondendo la complessità delle interazioni tra componenti multiple.

Esempi dai Testi d'Esame

1. Sistema Famil.io *(Appello 19/08/2019)*

Contesto: Portale di tracciamento che invia pubblicità mirata tramite diversi canali di comunicazione.

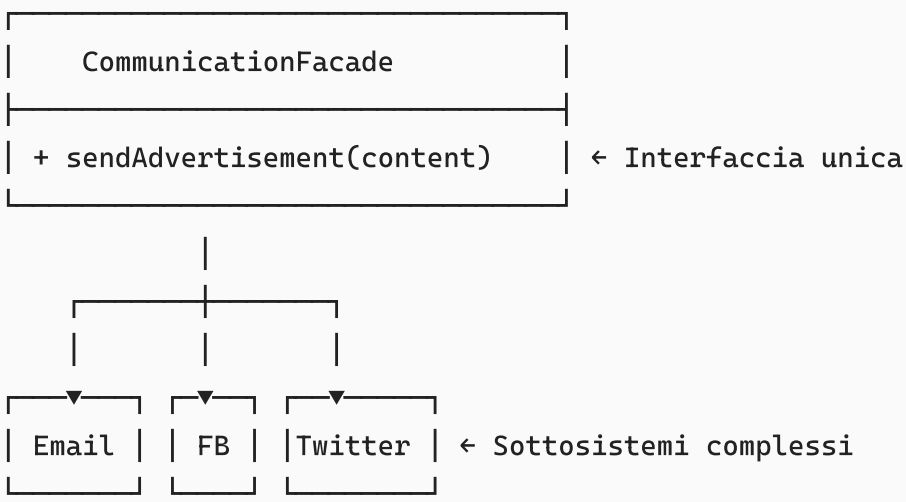
Problema:

- Necessità di inviare pubblicità tramite **email, Facebook, Twitter**
- Ogni canale ha interfacce e protocolli diversi
- Voler facilitare l'aggiunta di nuovi canali

Soluzione Facade:

```
"Per facilitare l'aggiunta di nuove modalità di inoltro e condivisione,
l'utilizzo di tali canali di comunicazione è reso standard, utilizzando
un'interfaccia unica di esecuzione."
```

Diagramma:



2. Sistema OTA myOTA *(Appello 04/07/2023)*

Contesto: Online Travel Agency che cerca voli tramite diversi Global Distribution Systems.

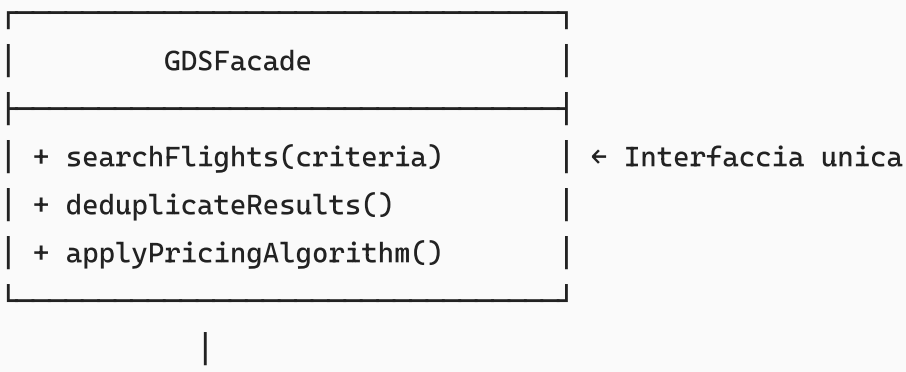
Problema:

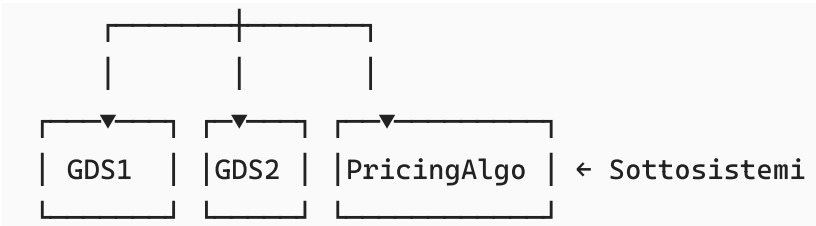
- Interfacciarsi con **GDS1 e GDS2** che hanno protocolli diversi
- Comunicazione di rete complessa
- De-duplicazione e algoritmi di pricing

Soluzione Facade:

```
"I GDS (Global Distribution System) forniscono un'interfaccia unica
di dialogo con le compagnie aeree"
```

Diagramma:





3. **Sistema Trading Los Angeles** *(Appello 20/07/2020)*

Contesto: Sistema di trading automatico basato su sentiment analysis di Twitter.

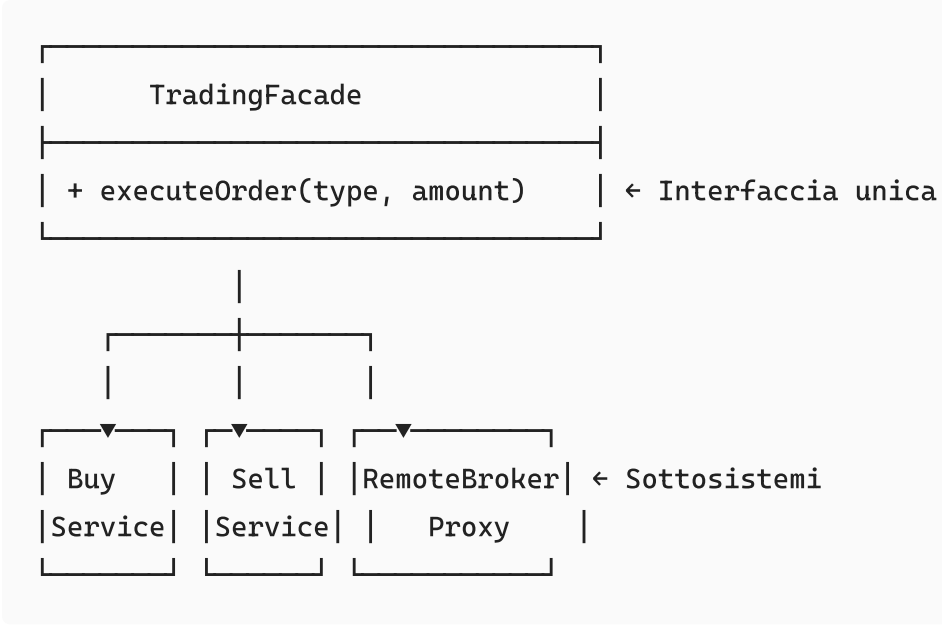
Problema:

- Operazioni di **compra e vendi** azioni su piattaforma remota
- Protocolli complessi di comunicazione con broker
- Necessità di standardizzare operazioni diverse

Soluzione Facade:

```
"L'interfaccia di compravendita è unica e deve consentire di
comprare o vendere azioni usando lo stesso metodo"
```

Diagramma:

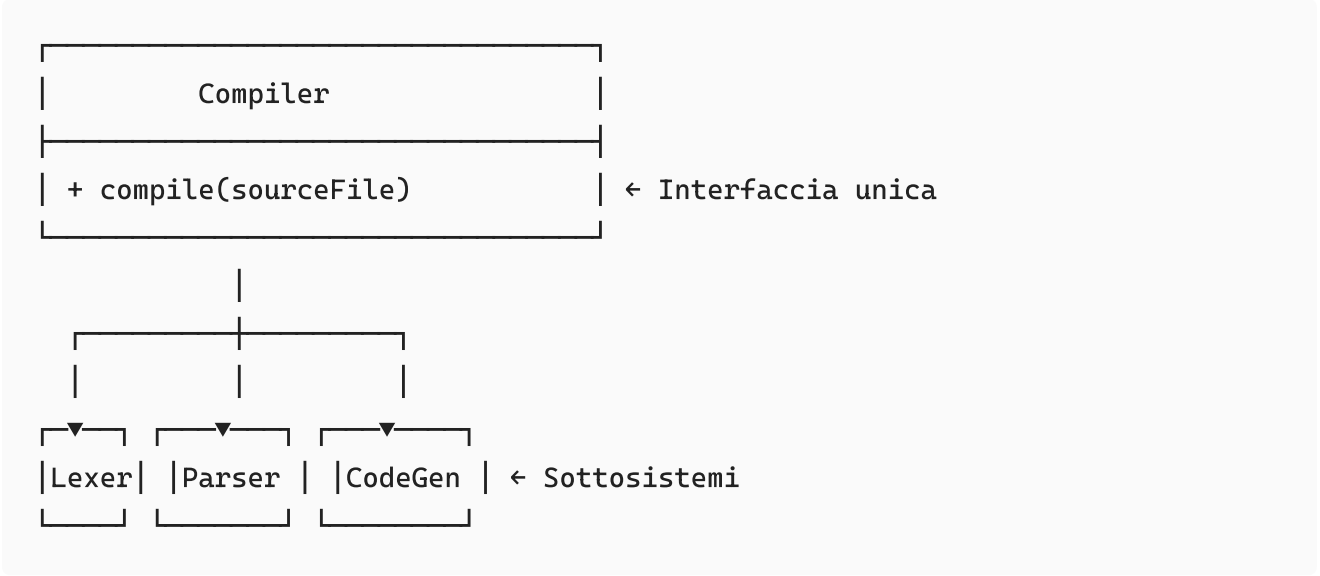


Esempio Teorico: Compilatore

Dal corso teorico:

```
"Il compilatore riunisce come facciata una serie di funzionalità
(interpretazione dei flussi I/O, scan, parse, lettura di token,
connessioni nodi e variabili), senza esserne dipendente"
```

Diagramma:



Pattern ricorrenti negli Esami

Indicatori di Facade:

1. **"Interfaccia unica"** - frase ricorrente
2. **"Standardizzare"** operazioni diverse
3. **"Facilitare l'aggiunta"** di nuovi componenti
4. **"Mascherare la complessità"** di comunicazione

Contesti tipici:

- **Comunicazione multi-canale** (email, social media)
- **Integrazione con servizi esterni** (GDS, broker)
- **Sistemi distribuiti** (locale vs remoto)
- **API di terze parti** con protocolli diversi

Vantaggi evidenziati:

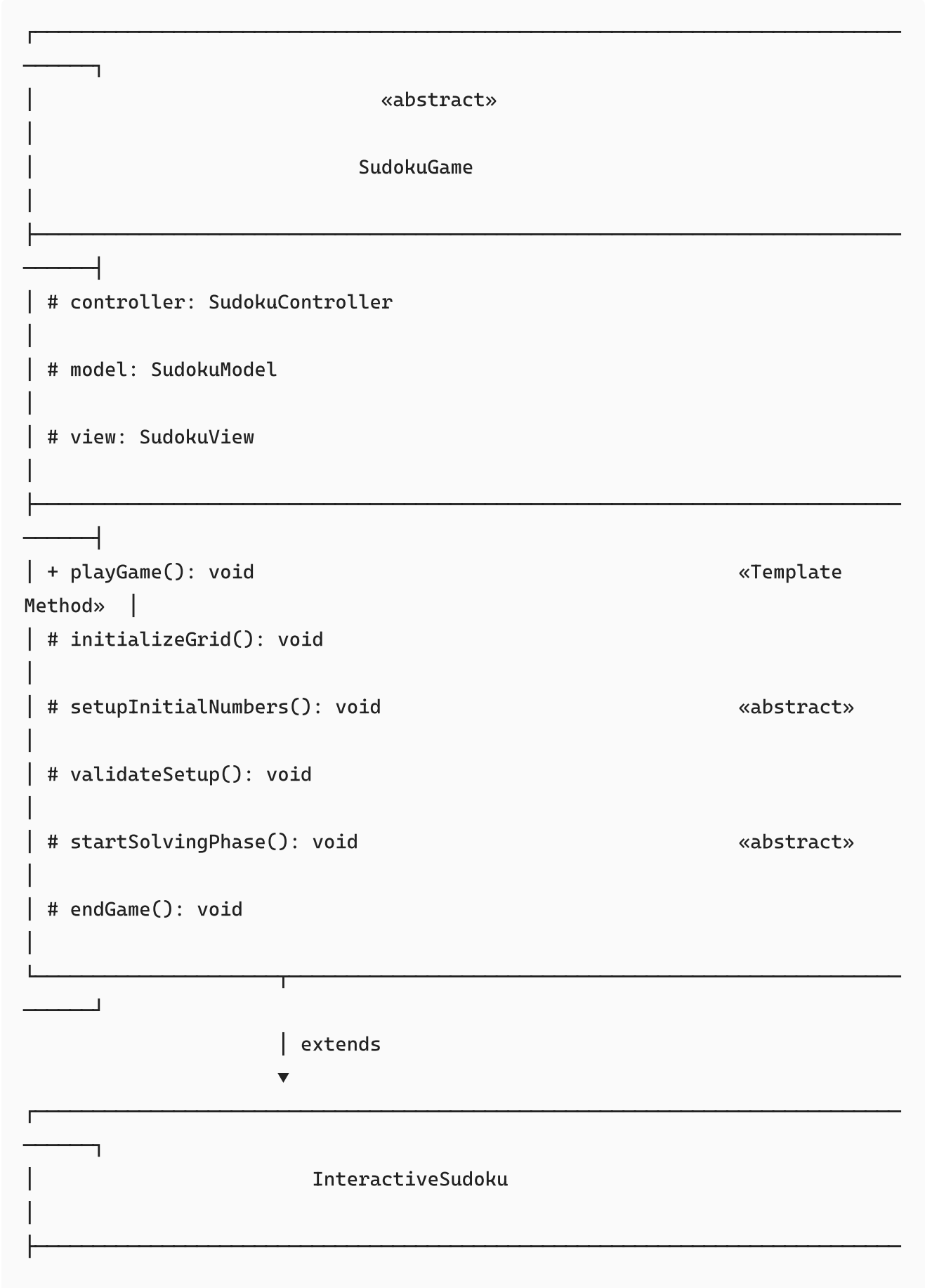
- **Semplificazione** dell'interfaccia client
- **Disaccoppiamento** tra client e sottosistemi
- **Estensibilità** per nuove implementazioni
- **Nascondere complessità** implementativa

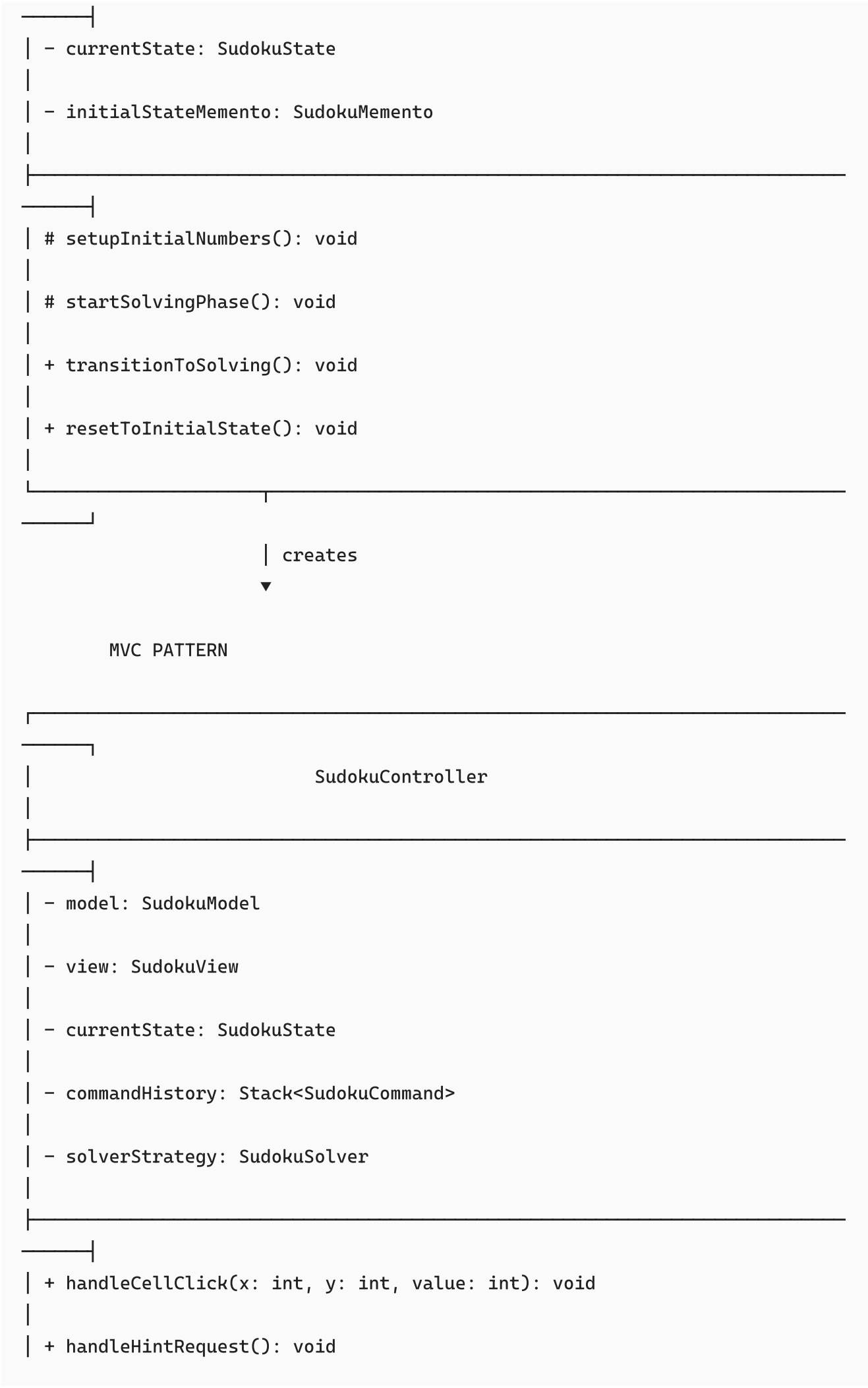
Conclusione

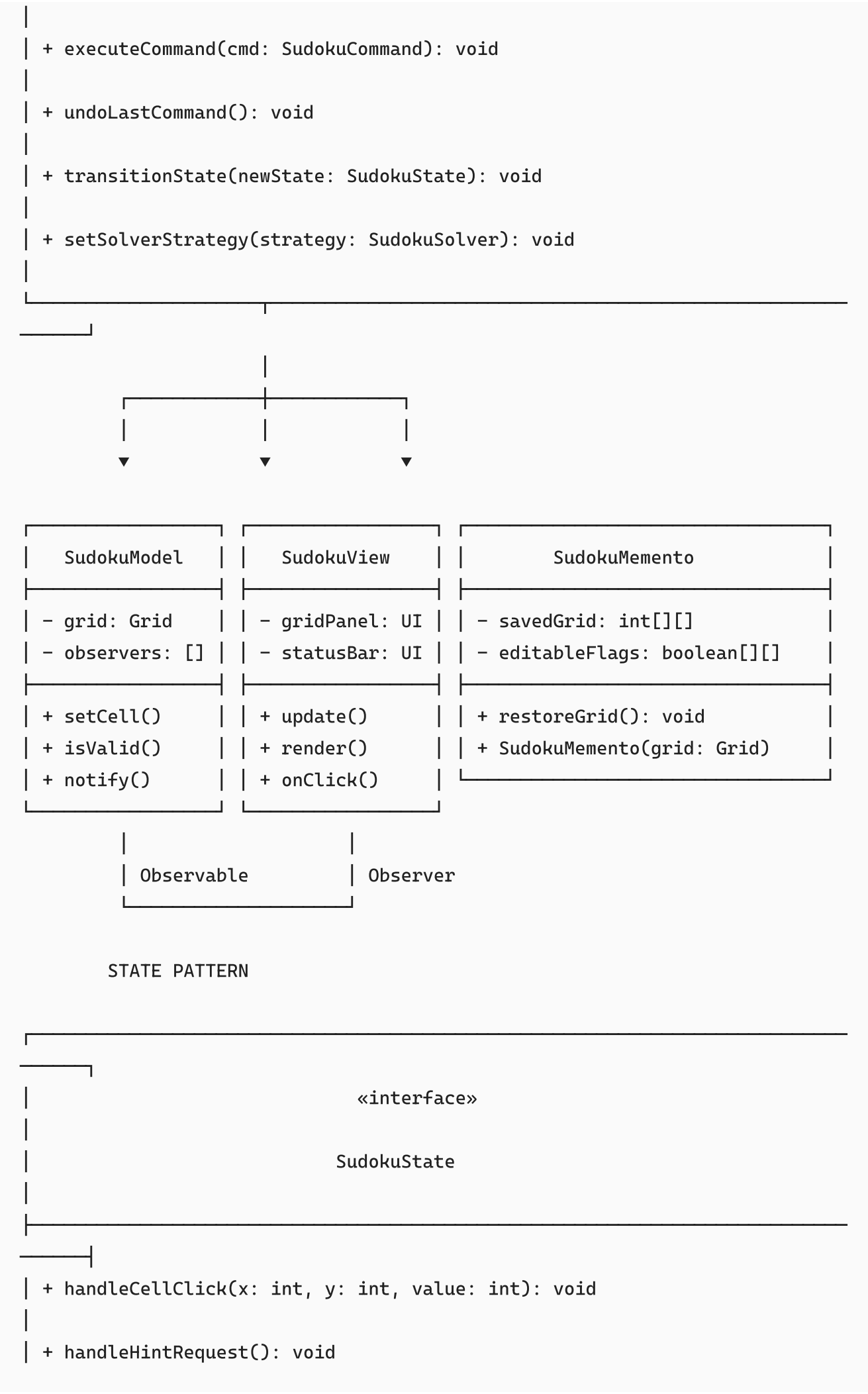
Negli esami, il pattern Facade emerge sempre quando il testo menziona la necessità di **"unificare" o "standardizzare"** l'accesso a sottosistemi eterogenei, fornendo ai client un'**interfaccia semplice e coerente** che nasconde la complessità delle interazioni sottostanti.

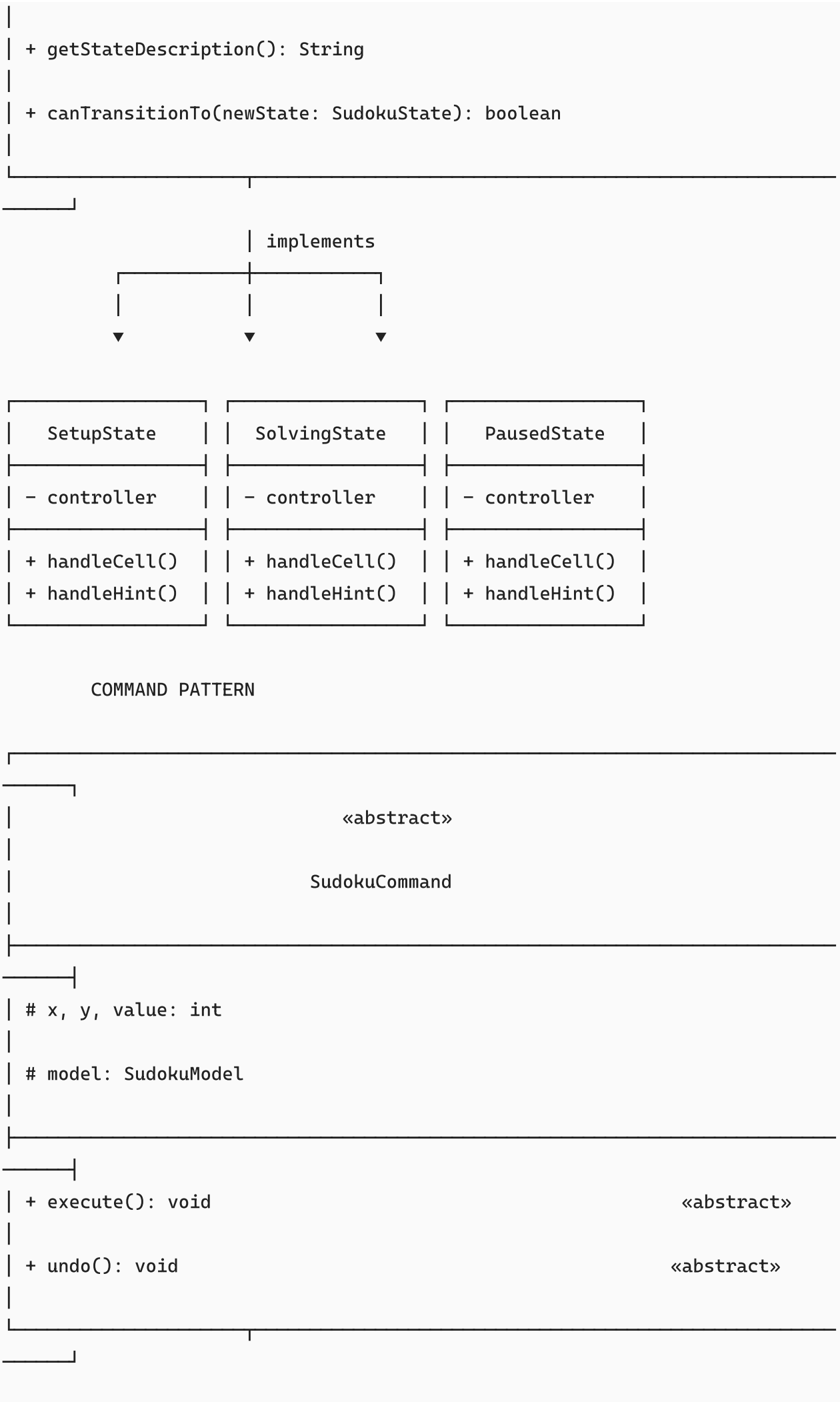
Diagramma Completo delle Classi - Sudoku con Game + MVC + Pattern

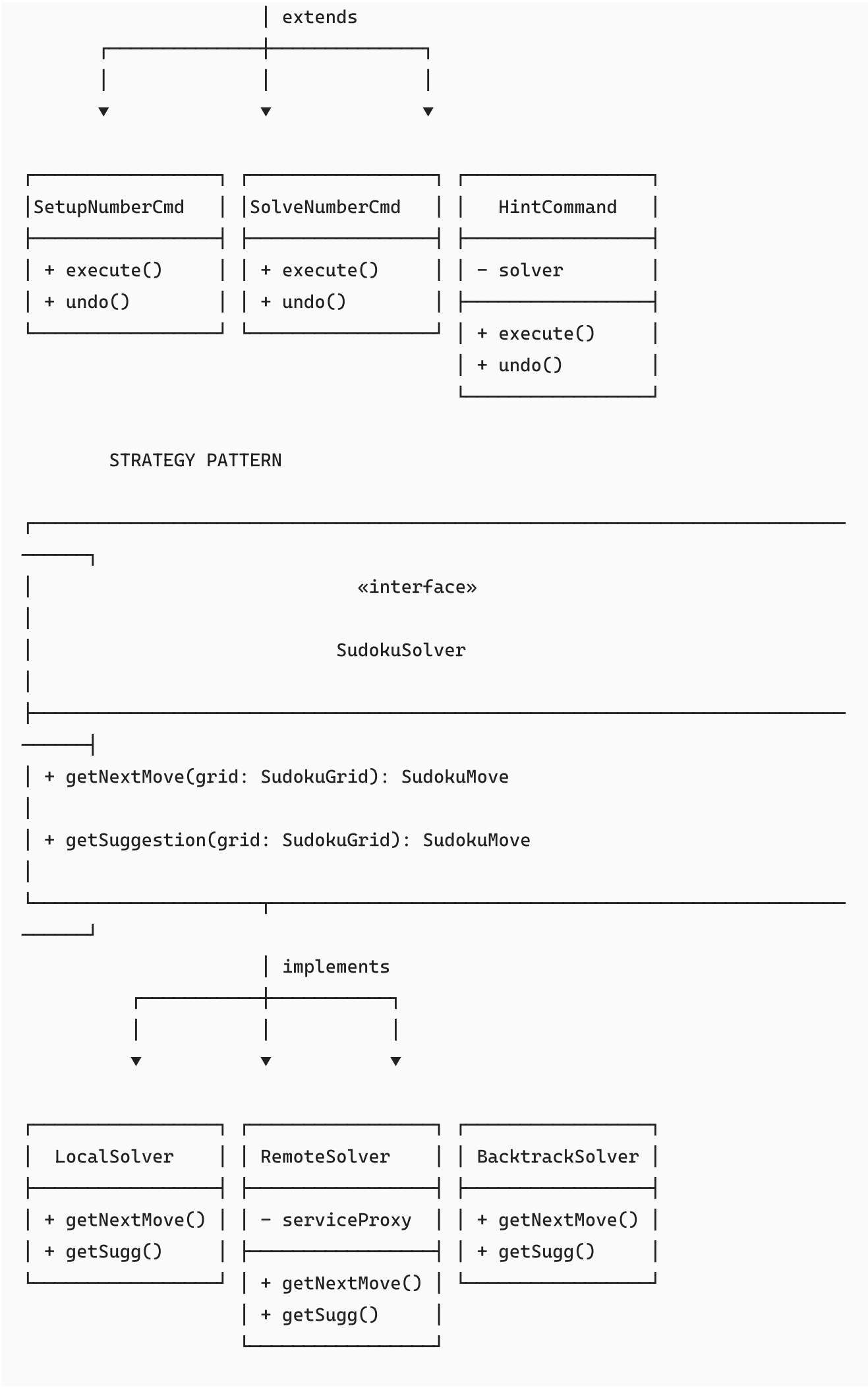
Architettura Completa

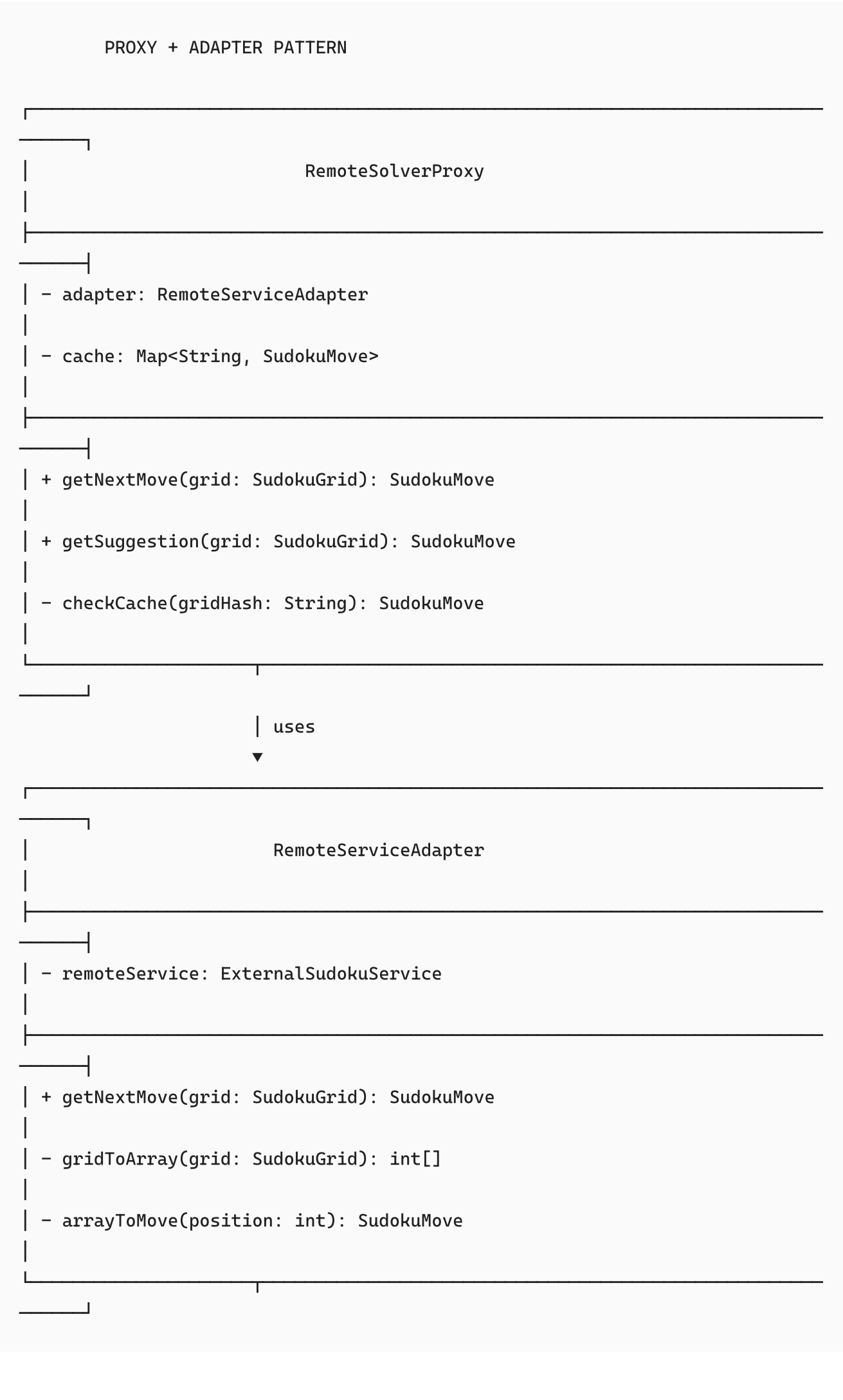


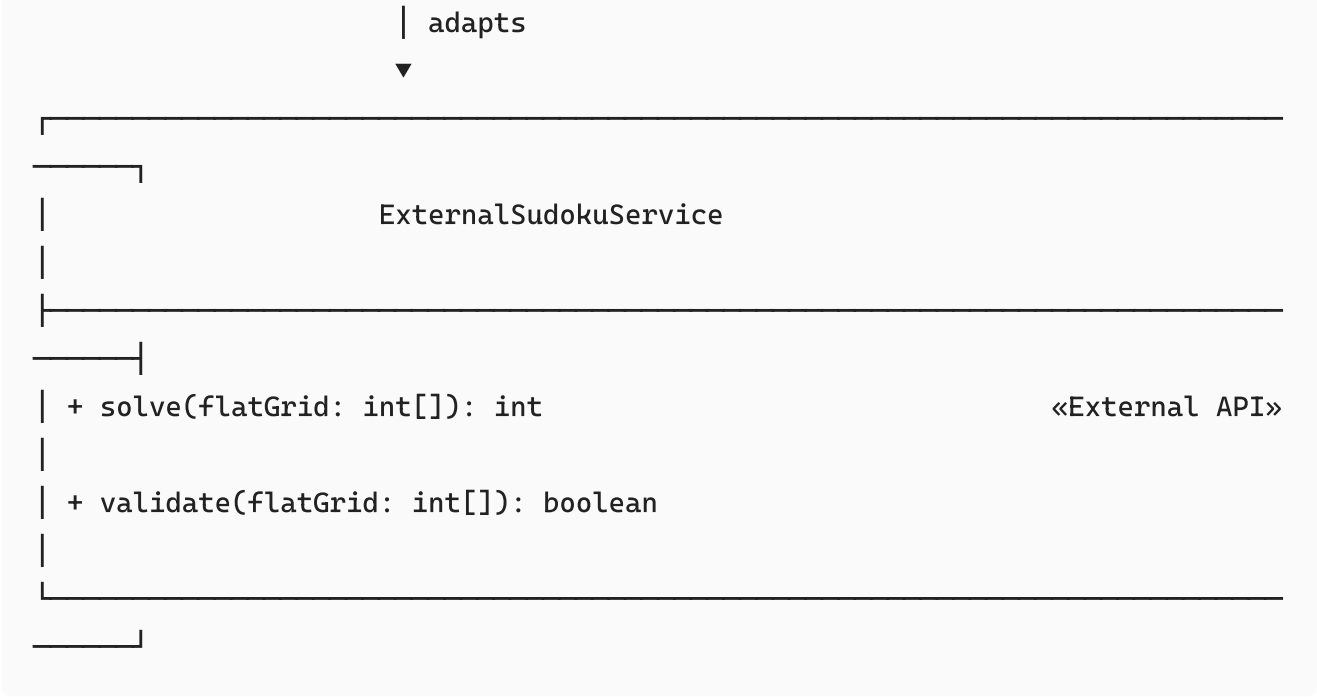

- currentState: SudokuState
- initialStateMemento: SudokuMemento
setupInitialNumbers(): void
startSolvingPhase(): void
+ transitionToSolving(): void
+ resetToInitialState(): void
creates
MVC PATTERN
SudokuController
- model: SudokuModel
- view: SudokuView
- currentState: SudokuState
- commandHistory: Stack<SudokuCommand>
- solverStrategy: SudokuSolver
+ handleCellClick(x: int, y: int, value: int): void
+ handleHintRequest(): void


+ executeCommand(cmd: SudokuCommand): void
+ undoLastCommand(): void
+ transitionState(newState: SudokuState): void
+ setSolverStrategy(strategy: SudokuSolver): void
SudokuModel
- grid: Grid
- observers: []
+ setCell()
+ isValid()
+ notify()
SudokuView
- gridPanel: UI
- statusBar: UI
+ update()
+ render()
+ onClick()
SudokuMemento
- savedGrid: int[][]
- editableFlags: boolean[][]
+ restoreGrid(): void
+ SudokuMemento(grid: Grid)
Observable
Observer
STATE PATTERN
«interface»
SudokuState
+ handleCellClick(x: int, y: int, value: int): void
+ handleHintRequest(): void


+ getStateDescription(): String
+ canTransitionTo(newState: SudokuState): boolean
implements
SetupState
- controller
+ handleCell()
+ handleHint()
SolvingState
- controller
+ handleCell()
+ handleHint()
PausedState
- controller
+ handleCell()
+ handleHint()
COMMAND PATTERN
«abstract»
SudokuCommand
x, y, value: int
model: SudokuModel
+ execute(): void «abstract»
+ undo(): void «abstract»


extends
SetupNumberCmd
+ execute()
+ undo()
SolveNumberCmd
+ execute()
+ undo()
HintCommand
- solver
+ execute()
+ undo()
STRATEGY PATTERN
«interface»
SudokuSolver
+ getNextMove(grid: SudokuGrid): SudokuMove
+ getSuggestion(grid: SudokuGrid): SudokuMove
implements
LocalSolver
+ getNextMove()
+ getSugg()
RemoteSolver
- serviceProxy
+ getNextMove()
+ getSugg()
BacktrackSolver
+ getNextMove()
+ getSugg()


PROXY + ADAPTER PATTERN
RemoteSolverProxy
- adapter: RemoteServiceAdapter
- cache: Map<String, SudokuMove>
+ getNextMove(grid: SudokuGrid): SudokuMove
+ getSuggestion(grid: SudokuGrid): SudokuMove
- checkCache(gridHash: String): SudokuMove
uses
RemoteServiceAdapter
- remoteService: ExternalSudokuService
+ getNextMove(grid: SudokuGrid): SudokuMove
- gridToArray(grid: SudokuGrid): int[]
- arrayToMove(position: int): SudokuMove



Relazioni Principali

1. Game → MVC

```
InteractiveSudoku
    ├── creates → SudokuController
    ├── creates → SudokuModel
    └── creates → SudokuView
```

2. Controller Orchestration

```
SudokuController
    ├── manages → SudokuState (State Pattern)
    ├── executes → SudokuCommand (Command Pattern)
    ├── uses → SudokuSolver (Strategy Pattern)
    └── updates → SudokuModel → notifies → SudokuView
```

3. Pattern Integration

```
Template Method (SudokuGame)
    └── controls game flow
        ├── State Pattern (fase setup vs solving)
        ├── Command Pattern (operazioni undo/redo)
        ├── Strategy Pattern (algoritmi risoluzione)
        ├── Proxy Pattern (servizio remoto)
        ├── Adapter Pattern (interfaccia esterna)
```

```
        ├── Observer Pattern (MVC notifications)
        └── Memento Pattern (stato iniziale)
```

Vantaggi di questa Architettura

1. **Separazione Responsabilità**: Game gestisce flusso, MVC gestisce interazione
2. **Estensibilità**: Nuovi stati, comandi, solver facilmente aggiungibili
3. **Testabilità**: Ogni componente testabile indipendentemente
4. **Manutenibilità**: Pattern consolidati, responsabilità chiare
5. **Riusabilità**: Componenti MVC riutilizzabili per altri puzzle games

La classe **Game** diventa il **director** che coordina tutti i pattern, mentre il **Controller** gestisce le interazioni specifiche dell'interfaccia utente.

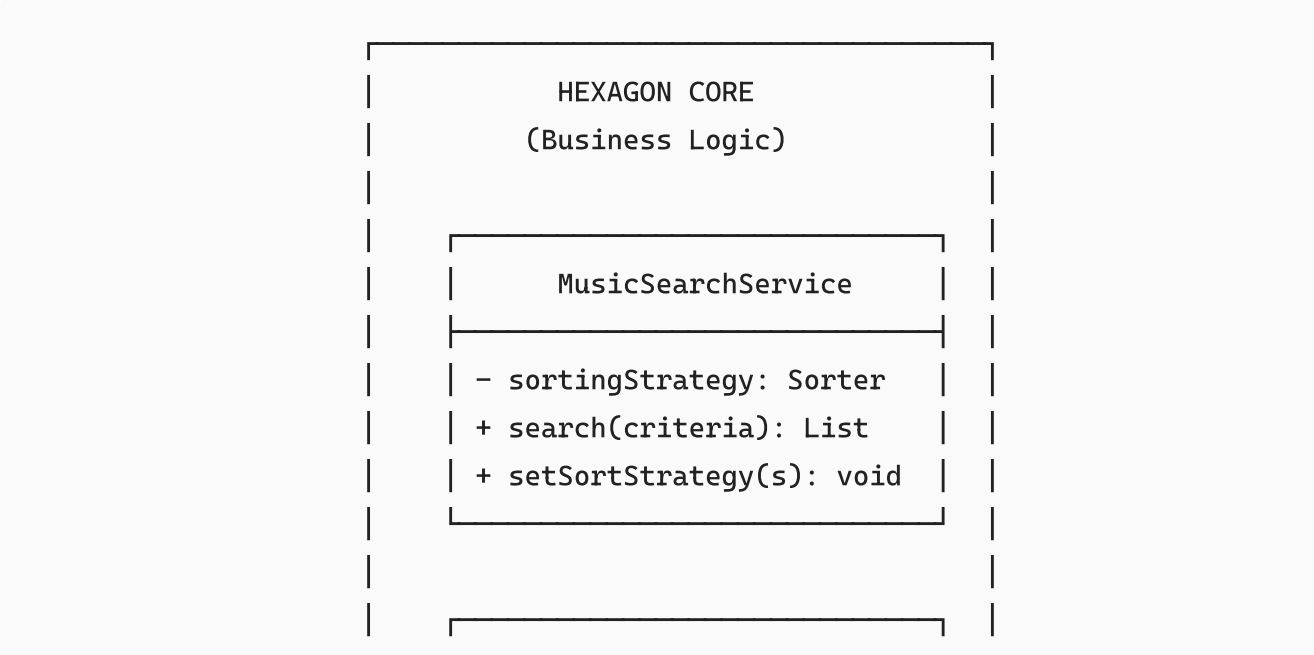
Architettura Esagonale - Sistema Spotify dall'Appello

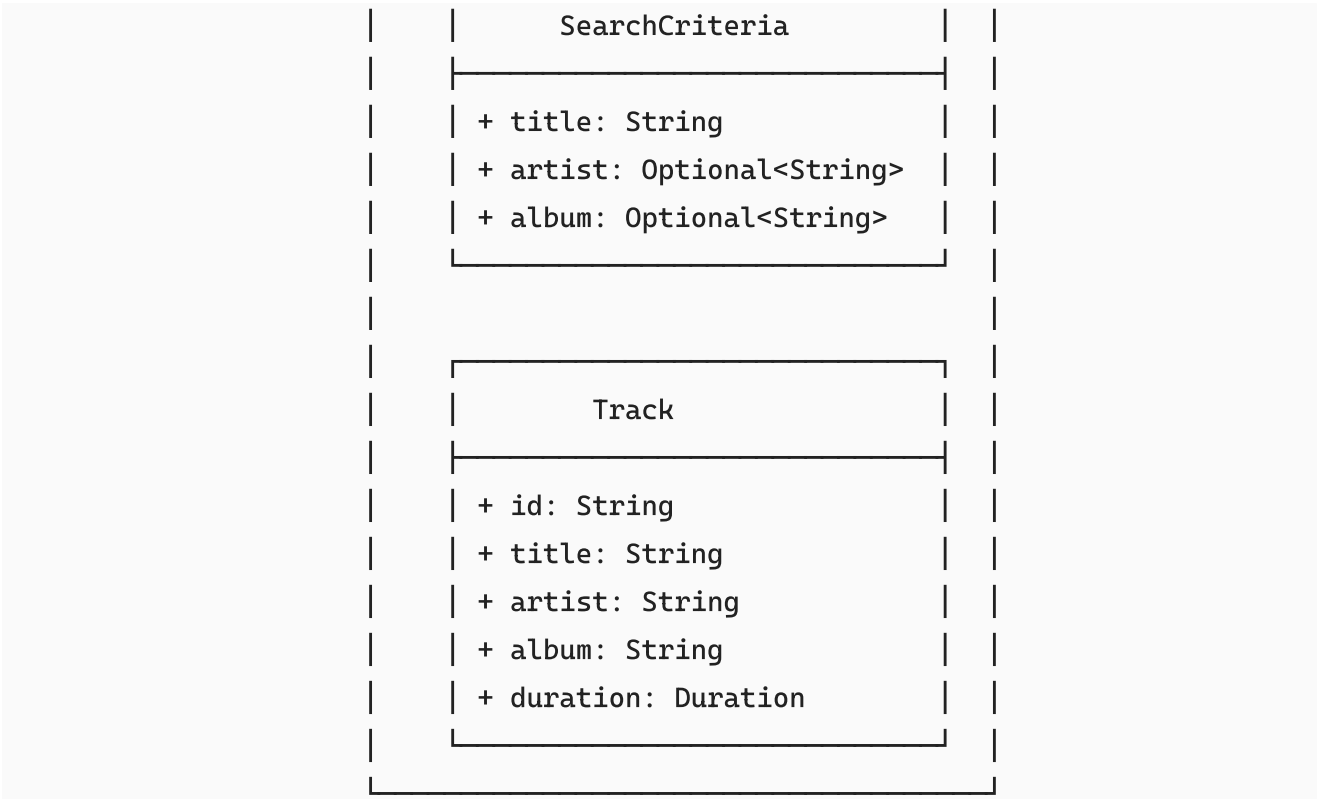
Testo dell'Esame *(6 Settembre 2024)*

Spotify è una piattaforma di streaming audio. L'applicazione mobile si connette a server per richiedere brani musicali. La richiesta contiene titolo, artista, album. Il server risponde con lista di risultati dalla ricerca nel database. **La comunicazione avviene come se eseguissero sul medesimo host**. **La ricerca viene gestita lato server con una classica architettura esagonale**. I risultati vengono ordinati con **algoritmo intelligente in continuo sviluppo**.

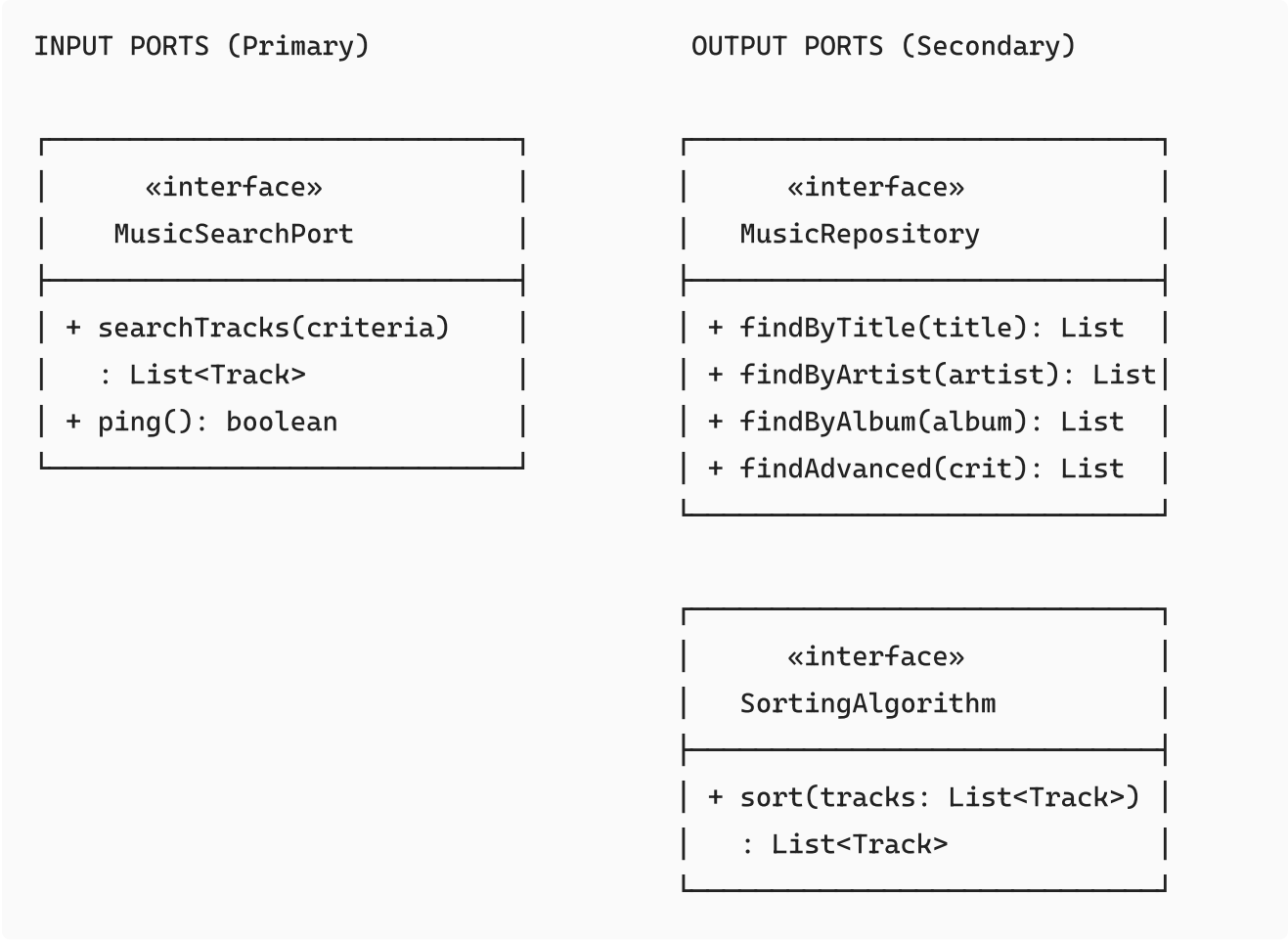
Soluzione: Architettura Esagonale Completa

Core Hexagon - Business Logic





Ports (Interfacce)



Adapters (Implementazioni)

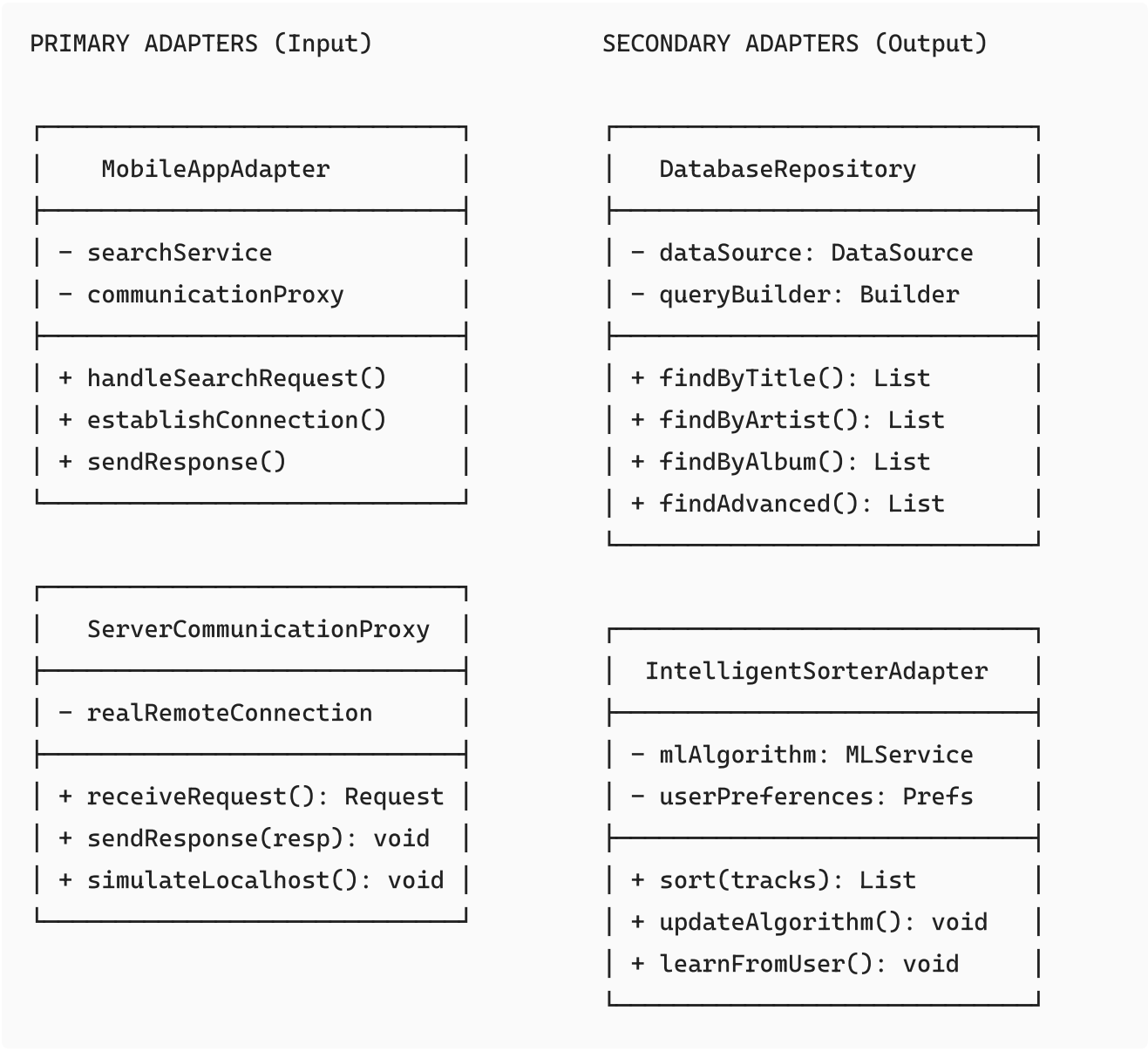
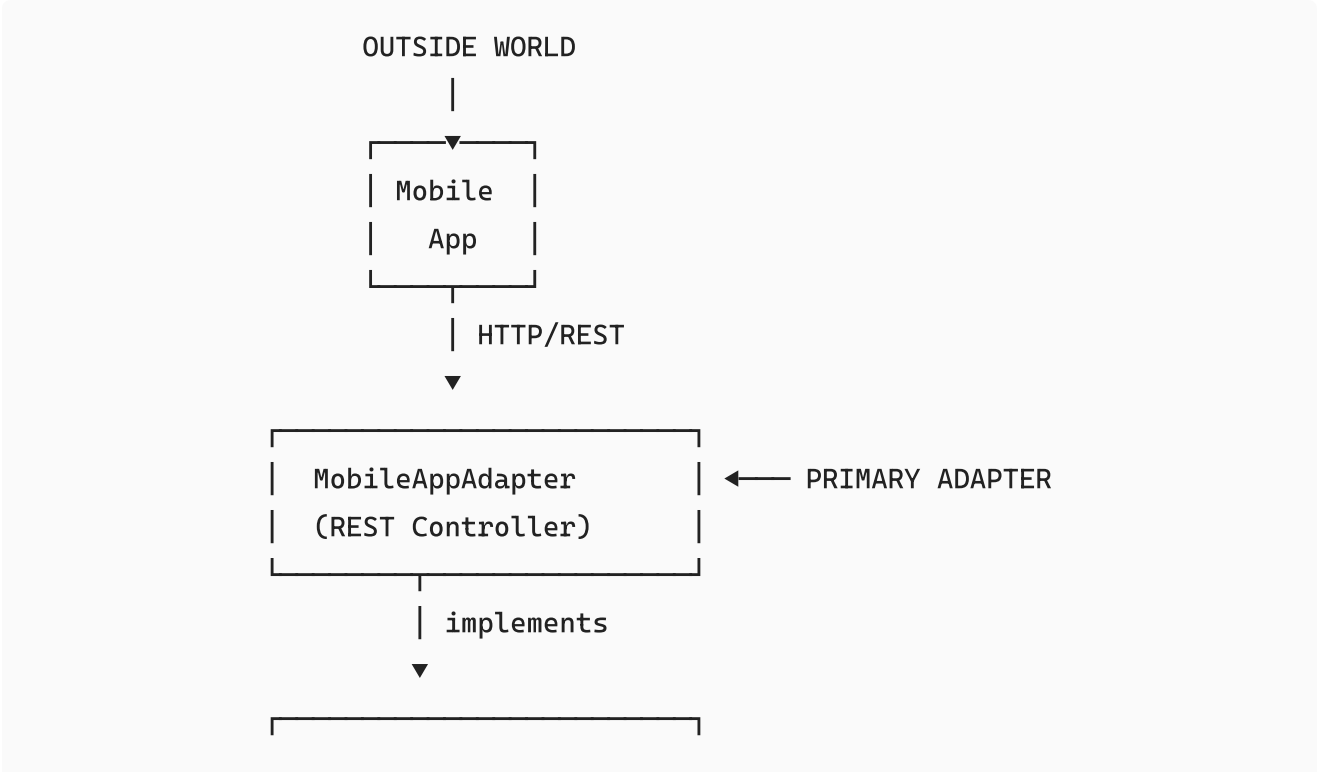
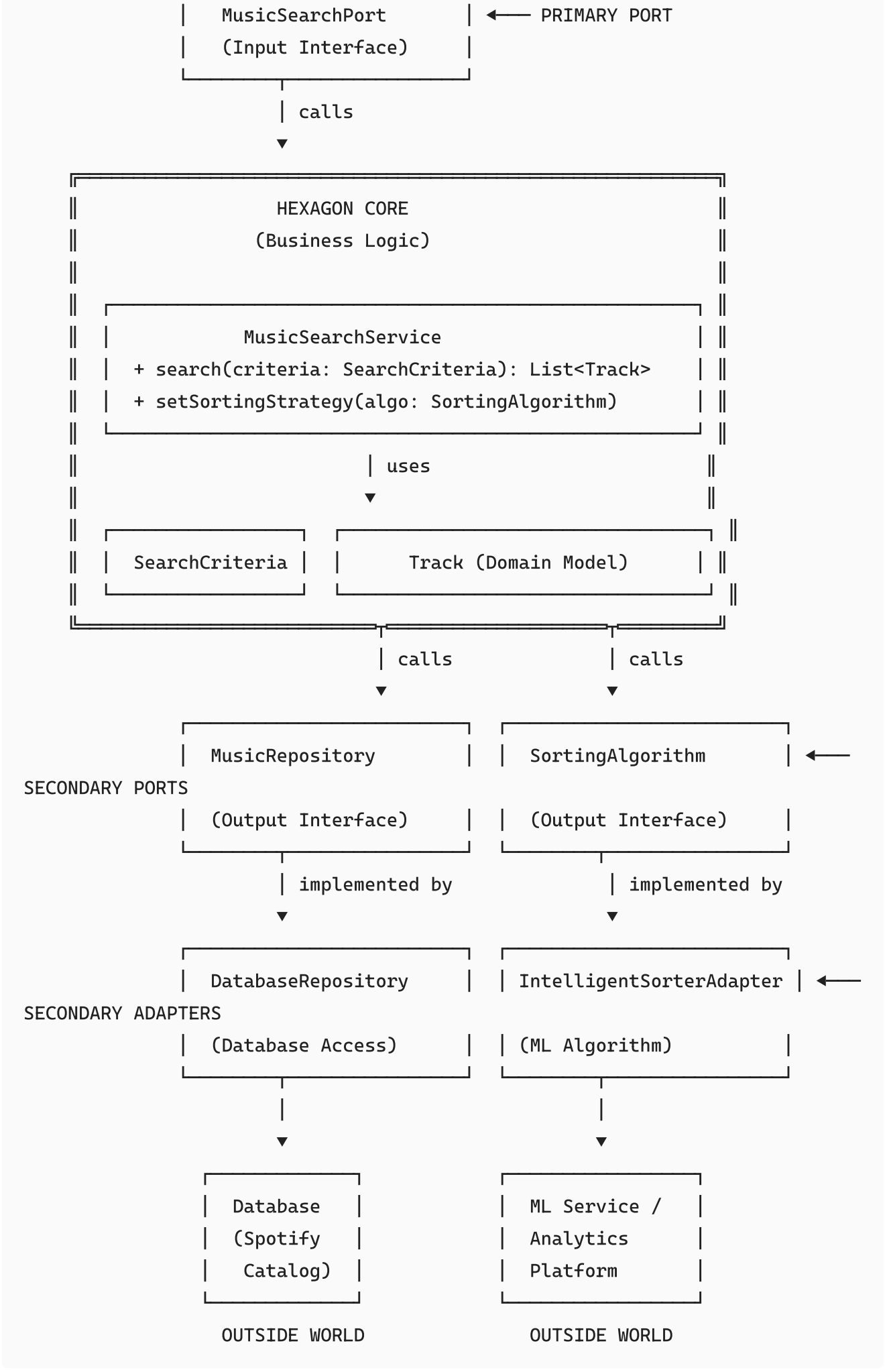


Diagramma Completo dell'Architettura Esagonale




MusicSearchPort
(Input Interface)
PRIMARY PORT
calls
HEXAGON CORE
(Business Logic)
MusicSearchService
+ search(criteria: SearchCriteria): List<Track>
+ setSortingStrategy(algo: SortingAlgorithm)
uses
SearchCriteria
Track (Domain Model)
calls
calls
MusicRepository
(Output Interface)
SortingAlgorithm
(Output Interface)
SECONDARY PORTS
implemented by
implemented by
DatabaseRepository
(Database Access)
IntelligentSorterAdapter
(ML Algorithm)
SECONDARY ADAPTERS
Database
(Spotify
Catalog)
ML Service /
Analytics
Platform
OUTSIDE WORLD
OUTSIDE WORLD

Pattern Integrati

1. Strategy Pattern *(Algoritmo Intelligente)*

```
interface SortingAlgorithm {
    List<Track> sort(List<Track> tracks, UserPreferences prefs);
}

class RelevanceSorter implements SortingAlgorithm { /* ML-based */ }
class PopularitySorter implements SortingAlgorithm { /* Chart-based */ }
class PersonalizedSorter implements SortingAlgorithm { /* User history */ }
```

2. Proxy Pattern *(Comunicazione Locale-Like)*

```
class ServerCommunicationProxy implements CommunicationPort {
    private RemoteConnection realConnection;

    public SearchResponse handleRequest(SearchRequest req) {
        // Simula comunicazione locale
        return realConnection.processRemoteRequest(req);
    }
}
```

3. Adapter Pattern *(Database Legacy)*

```
class DatabaseRepository implements MusicRepository {
    private LegacySpotifyDB legacyDB;

    public List<Track> findAdvanced(SearchCriteria criteria) {
        // Adatta interfaccia moderna a DB legacy
        String legacyQuery = criteriaToLegacySQL(criteria);
        return legacyDB.executeQuery(legacyQuery);
    }
}
```

Dimostrazione Architettura Esagonale

✅ Caratteristiche Esagonali Presenti:

1. **Business Logic Isolata**: `MusicSearchService` non dipende da infrastruttura
2. **Ports & Adapters**: Chiara separazione tra interfacce e implementazioni
3. **Dependency Inversion**: Core dipende da interfacce, non da implementazioni
4. **Multiple Adapters**: Diversi adapter per stesso port (sorting algorithms)
5. **Testabilità**: Mock facile tramite interfacce

6. **Sostituibilità**: Database, algoritmi, comunicazione intercambiabili

✅ Vantaggi Architetturali:

- **Indipendenza dal Database**: Cambio DB senza toccare business logic
- **Algoritmi Pluggable**: Strategy pattern per sorting intelligente
- **Testing Isolato**: Mock di ports per unit testing
- **Deploy Flessibile**: Adapters configurabili per ambienti diversi
- **Evoluzione Tecnologica**: Nuovi adapters senza modifiche al core

L'architettura è **genuinamente esagonale** perché la business logic (ricerca musicale) è completamente isolata dall'infrastruttura (database, comunicazione, algoritmi) tramite **ports** ben definiti e **adapters** intercambiabili.