

Artifact — Concetti di Programmazione per lo stack “App Padel”

Riepilogo operativo e didattico dei concetti chiave di programmazione usati nello stack: Spring Boot (MVC), JPA/Hibernate, Security/Auth, Testing (JUnit/MockMvc, JaCoCo), Design Pattern (Observer, Strategy, Singleton), Event-Driven domain, UML, CI. Ogni sezione include: **cosa è, perché serve, come lo usi nel progetto, trappole comuni e check rapidi**.

0) Panoramica del Progetto (dal README)

Titolo: *App Padel — Sistema di Gestione Partite*

Contesto accademico - Corso: *Ingegneria del Software* — A.A. **2024/2025** - Università: *[Nome Università]* - Studente: *[Nome Cognome — Matricola]* - Repository: *[Link GitHub]* · Demo Live: *[Replit/Deployment]*

Ambito e obiettivi - Creare/gestire **partite di padel** tra giocatori che non si conoscono. - **Iscrizioni** con limite **max 4**; **conferma automatica** al raggiungimento di 4. - **Feedback post-partita** per valutare il livello degli altri e calcolare **livello percepito**. - Dimostrare principi pratici di **Ingegneria del Software**: MVC, Pattern (Observer/Strategy/Singleton), JPA/Hibernate, UI con Thymeleaf, **Testing con coverage $\geq 80\%$** .

0.1) Requisiti Funzionali (RF)

RF1 – Gestione Utenti - RF1.1 Registrazione con **livello dichiarato** (Principiante/Intermedio/Avanzato/Professionista) - RF1.2 Visualizzazione **profilo utente** con statistiche - RF1.3 **Livello percepito** calcolato dai feedback ricevuti

RF2 – Gestione Partite - RF2.1 Creazione partita (**fissa** o **proposta**) - RF2.2 Join con **controllo massimo 4 giocatori** - RF2.3 Leave **solo se non confermata** - RF2.4 **Conferma automatica** al 4° iscritto - RF2.5 **Filtro & ordinamento** per data, popolarità, livello

RF3 – Sistema Feedback - RF3.1 Inserimento **feedback post-partita** - RF3.2 **Un feedback per utente per partita** - RF3.3 Aggiornamento **livello percepito** (media feedback)

RF4 – Notifiche - RF4.1 Notifica **conferma partita** (4 giocatori) - RF4.2 Notifica **termine partita**

0.2) Requisiti Non Funzionali (RNF)

RNF1 — Architettura - Pattern **MVC** con separazione **Controller/Service/Repository** - Uso di **almeno 2 design pattern** oltre a quelli offerti dal framework

RNF2 — Tecnologie - Backend: **Spring Boot 3.5.5, Java 17** - Database: **H2** (sviluppo), **JPA/Hibernate** - Frontend: **Thymeleaf, HTML/CSS** - Testing: **JUnit 5, JaCoCo** (coverage $\geq 80\%$)

RNF3 — Qualità - **Javadoc** e commenti mirati su logica complessa - Test unitari per la **business logic** - **Exception handling** uniforme e sicuro

0.3) Stack & Versioni (rapido)

- **Java 17** · **Spring Boot 3.5.5** · **JPA/Hibernate** · **H2**
- **Thymeleaf** per le view server-side · **JUnit 5** · **JaCoCo**

Nota per junior: Spring crea e gestisce le istanze dei componenti (Controller/Service/Repository) al posto tuo. Tu definisci **cosa** fanno; Spring decide **quando** e **come** istanziarli (Inversion of Control/Dependency Injection).

0.4) Struttura del Progetto (con guida “dove guardo cosa”)

```
src/main/java/com/example/padel_app/
├─ config/                # Configurazioni e bootstrap
│   └─ DataSeeder.java    # Popola il DB H2 con dati demo
├─ controller/           # Controller MVC
│   ├── MatchController.java # REST API partite (JSON)
│   └─ WebController.java  # Controller per pagine HTML (Thymeleaf)
├─ event/                # Eventi (Observer pattern)
│   ├── MatchConfirmedEvent.java
│   └─ MatchFinishedEvent.java
├─ listener/             # Listener per gli eventi
│   └─ MatchEventListener.java
├─ model/                # Entità JPA (tabelle)
│   ├── User.java
│   ├── Match.java
│   ├── Registration.java
│   ├── Feedback.java
│   └─ enums/            # Enumerazioni dominio
│       ├── Level.java
│       ├── MatchType.java
│       ├── MatchStatus.java
│       └─ RegistrationStatus.java
├─ repository/           # Accesso dati (JPA)
│   ├── UserRepository.java
│   ├── MatchRepository.java
│   ├── RegistrationRepository.java
│   └─ FeedbackRepository.java
├─ service/              # Logica di business
│   ├── MatchService.java
│   └─ RegistrationService.java
```

```

|   ├── FeedbackService.java
|   ├── UserService.java
|   └── NotificationService.java
└── strategy/                # Strategie di ordinamento (Strategy pattern)
    ├── MatchSortingStrategy.java
    ├── DateSortingStrategy.java
    ├── PopularitySortingStrategy.java
    └── LevelSortingStrategy.java
└── PadelAppApplication.java # Main Spring Boot

src/main/resources/
└── templates/                # Pagine Thymeleaf
    ├── index.html
    ├── matches.html
    ├── users.html
    └── create-match.html
└── static/css/
    └── style.css

src/test/java/com/example/padel_app/
└── ... Test classes

```

Come usarla (occhio da junior) - Vuoi capire un flusso? Parti dal **Controller**, poi **Service**, poi **Repository**, poi guarda le **Entity**. - Vuoi vedere **HTML**? Cerca in `templates/` la pagina usata dal `WebController`. - Vuoi capire **ordinamenti**? Apri `strategy/` e guarda le classi che implementano l'interfaccia. - Vuoi capire **notifiche**? `event/` + `listener/` + `NotificationService`.

0.5) Installazione & Avvio (Quickstart)

Prerequisiti: Java 17, Maven 3.6+ (oppure usa *Maven Wrapper* incluso)

```

# 1) Clona il repo
git clone [repository-url]
cd padel-app

# 2) Avvia l'applicazione in dev
./mvnw spring-boot:run

```

L'app sarà su **http://localhost:5000**

H2 Console: `http://localhost:5000/h2-console`

JDBC URL: `jdbc:h2:mem:padeladb` — **User:** `sa` — **Password:** *(vuota)*

Suggerimento: se la porta 5000 è occupata, puoi cambiare la porta in `application.yml` (`server.port`).

0.6) Testing & Coverage (comandi e obiettivi)

- **Obiettivo coverage:** $\geq 80\%$ (JaCoCo)
- **Comandi:**

```
# Esegui i test
./mvnw test
# Genera report coverage
./mvnw jacoco:report
```

Report HTML in: `target/site/jacoco/index.html`.

Cosa testare per primi (checklist junior) - `MatchServiceTest` → logica limite 4 e conferma - `RegistrationServiceTest` → regole join/leave - `FeedbackServiceTest` → un feedback per utente/partita + media - `StrategyPatternTest` → ordinamenti coerenti - `ObserverPatternTest` → evento scatenato e notifiche

0.7) Diagrammi UML (dove trovarli)

- Use Case: `docs/use-case-diagram.puml`
- Class Diagram: `docs/class-diagram.puml`
- Sequence Diagram: `docs/sequence-diagram.puml`

Tip: aprili con PlantUML (o VS Code + plugin). Confrontali con le classi reali.

0.8) Workflow Tipico (stati & eventi)

- 1) **Creazione Partita** → stato `WAITING` (il creatore è già iscritto)
 - 2) **Join** di altri giocatori (massimo 4)
 - 3) Al 4° join → **evento** `MatchConfirmed` → stato `CONFIRMED` → **notifica inviata**
 - 4) **Fine partita** → **evento** `MatchFinished` → stato `FINISHED`
 - 5) **Feedback**: ogni giocatore lascia **un solo feedback** per gli altri → aggiorna **livello percepito**
-

0.9) End-to-end per Junior: “dal click al DB” (senza dare nulla per scontato)

Esempio: *L'utente clicca “Unisciti” su una partita.* 1. **Browser → Controller**: arriva una richiesta HTTP a `MatchController.join()` (se REST) o una rotta in `WebController` (se pagina HTML). 2. **Controller → Service**: il controller non decide nulla: passa la richiesta a `RegistrationService.join(matchId, userId)`. 3. **Service → Repository**: il service valida regole (es. posti rimasti) e usa `RegistrationRepository/MatchRepository` per leggere/scrivere sul DB (via JPA, quindi **niente SQL scritto a mano** nella maggior parte dei casi). 4. **Regola di dominio**: se dopo l'iscrizione il conteggio è 4, il service **pubblica un evento** `MatchConfirmedEvent`. 5. **Listener**: `MatchEventListener` intercetta l'evento e chiama `NotificationService` (singleton) per inviare

la **notifica**. 6. **Risposta**: il controller restituisce **JSON** (REST) o una **view Thymeleaf** (pagina HTML) con il risultato.

Cosa impari qui: separazione dei ruoli (Controller = I/O HTTP, Service = regole, Repository = DB, Event/Listener = reazioni asincrone o disaccoppiate).

0.10) Pattern implementati (con file coinvolti)

Observer - Eventi: `event/MatchConfirmedEvent.java`, `event/MatchFinishedEvent.java` - Listener: `listener/MatchEventListener.java` - Notifiche: `service/NotificationService.java` (singleton)

Strategy - Interfaccia: `strategy/MatchSortingStrategy.java` - Implementazioni: `DateSortingStrategy.java`, `PopularitySortingStrategy.java`, `LevelSortingStrategy.java`

Singleton - `NotificationService` è un bean `@Service` → singleton di default in Spring (`@Scope("singleton")` opzionale per esplicitare)

0.11) FAQ & Glossario per Junior

- **REST vs Web Controller**: `@RestController` restituisce JSON; `@Controller` restituisce nomi di **template Thymeleaf**.
- **Entity vs DTO**: Entity = forma **DB**; DTO = forma **API/UI** per scambio dati. Non esporre Entity direttamente all'esterno.
- **H2: DB in-memory**; si resetta ad ogni avvio. Utile per test/dev.
- **JaCoCo**: genera un **report HTML** con le righe coperte dai test (verde) e scoperte (rosso).
- **Dove parte l'app**: `PadelAppApplication.java` (metodo `main`), Spring configura tutto e avvia il web server.


1) Principi OOP (incapsulamento, astrazione, ereditarietà, polimorfismo)

Cosa è: Paradigma che organizza il codice in **oggetti** con **stato** e **comportamenti**.

Perché serve: Riduce l'accoppiamento, aumenta la riusabilità e testabilità.

Nel progetto: - **Model**: `User`, `Match`, `Registration`, `Feedback` con proprietà private e metodi pubblici. - **Ereditarietà/Polimorfismo**: interfacce (es. `ListingStrategy`) e implementazioni multiple (`ByDate`, `ByPopularity`, `ByLevel`).

Trappole: DTO mescolati con Entity, setter selvaggi, logica di business nei controller.

Check  campi privati, costruttori chiari, metodi "intenzionali", interfacce per punti di estensione.

2) Architettura MVC (Model-View-Controller)

Cosa è: Separazione **Model** (dominio), **View** (UI), **Controller** (ingresso HTTP).

Perché serve: Mantiene separati responsabilità, facilita test.

Nel progetto: - **Controller:** `AuthController`, `MatchController`, `FeedbackController`, `UserController`. - **Service layer:** logica di dominio (`MatchService`, `FeedbackService`, `UserService`). - **Repository:** JPA per persistenza (`UserRepository`, ...). - **View/UI:** pagina login/registrazione, lista/dettaglio partite, profilo, feedback.

Trappole: business nei controller, repository chiamati direttamente dalla view.

Check  Controller sottili → chiamano Service; Service → orchestrano Repository; View senza logica di business.


3) Persistenza con JPA/Hibernate

Cosa è: Mapping O/R tra classi Java e tabelle DB.

Perché serve: Astrazione dal SQL, transazioni, validazioni.

Nel progetto: - **Entity:** `User`, `Match`, `Registration`, `Feedback` con annotazioni `@Entity`, `@Id`, relazioni (`@OneToMany`, `@ManyToOne`). - **Vincoli dominio:** max 4 iscritti, 1 feedback per coppia utente-partita. - **DB dev:** H2 in-memory con data seeding all'avvio.

Trappole: Lazy loading in serializzazione JSON, N+1 queries, cicli nelle relazioni.

Check  DTO per esposizione API, `@Transactional` in Service, fetch strategiche, indici su chiavi di ricerca.

4) API REST (Controller HTTP)

Cosa è: Endpoints stateless che espongono risorse con JSON e semantica HTTP.

Perché serve: Integrazione semplice con la UI/web.

Nel progetto: - `POST /auth/register`, `POST /auth/login`. - `GET /matches?sort=...&level=...`, `POST /matches`, `POST /matches/{id}/join`, `DELETE /matches/{id}/leave`. - `POST /matches/{id}/finish` (o `scheduled`), `POST /feedback`.

Trappole: status code sbagliati, leakage di Entity complete.

Check  200/201/204/400/401/403/404/409 coerenti; validazioni `@Valid`; DTO in/out espliciti.

5) Validazione & Error Handling

Cosa è: Vincoli su input (`@NotNull`, `@Email`, ...) e gestione eccezioni centralizzata.

Perché serve: Dati puliti, errori prevedibili lato client.

Nel progetto: - `@Valid` su payload, `@ControllerAdvice` + `@ExceptionHandler` per errori (es. `BusinessRuleException`, `EntityNotFound`).

Trappole: messaggi generici, stacktrace esposti.

Check ☒ schema di errore uniforme `{timestamp, path, code, message}`; test per casi limite (esaurimento posti).

6) Sicurezza & Autenticazione

Cosa è: Autenticazione (chi sei) + Autorizzazione (cosa puoi fare).

Perché serve: Protezione risorse, tracciabilità.

Nel progetto: - Registrazione/login con password hash (BCrypt), sessione/JWT (a scelta). - Regole: creazione partite → utente autenticato; partite fisse → admin/tool.

Trappole: password in chiaro, CORS mal configurato.

Check ☒ `BCryptPasswordEncoder`, rotte pubbliche vs protette, test d'integrazione 401/403.

7) Design Pattern (Observer, Strategy, Singleton)

7.1 Observer

Cosa è: Sottoscrittori informati da eventi del dominio.

Nel progetto: `MatchConfirmedEvent`, `MatchFinishedEvent` → Listener: `NotificationListener`, `StatsListener`.

Trappole: eventi "chiacchieroni", side-effects non idempotenti.


Check ☒ eventi compatti; listener idempotenti; log tracciabile.

7.2 Strategy

Cosa è: Algoritmi intercambiabili dietro una stessa interfaccia.

Nel progetto: `ListingStrategy` con `ByDate`, `ByPopularity`, `ByLevel`; selezione via query param `sort`.

Trappole: parametri non validi, default non definito.


Check  factory/mapper del param → Strategy, default `ByDate`.

7.3 Singleton

Cosa è: Un'unica istanza condivisa.

Nel progetto: `NotificationService` come punto unico di invio (email/log/in-app).

Trappole: stato condiviso non thread-safe.

Check  stateless o sincronizzato; in Spring, usare bean `@Service` (singleton di default).

8) Event-Driven Domain (Domain Events)

Cosa è: Il dominio emette eventi significativi (non "tecnici").

Perché serve: Disaccoppia flussi, abilita estensioni (analytics, notifiche) senza toccare i servizi core.

Nel progetto: al 4° `join` → `MatchConfirmedEvent`; quando `FINISHED` → `MatchFinishedEvent`.

Trappole: evento lanciato fuori da transazione; duplicazioni.

Check  pubblicazione in boundary transazionale; handler idempotenti; test evento→effetto.


9) Gestione Stato & Regole di Dominio

Cosa è: Macchina a stati della `Match` (es. `WAITING` → `CONFIRMED` → `FINISHED`).

Perché serve: Coerenza del flusso, vincoli applicativi.

Nel progetto: limite 4 iscritti; unico feedback per coppia utente-match; aggiornamento `matchesPlayed` a fine partita.

Trappole: transizioni non validate; race condition sul 4° join.

Check  invarianti verificate lato service, lock ottimistico/pessimistico se necessario.

10) DTO, Mapper e Boundary API

Cosa è: Oggetti di trasporto per isolare il dominio dall'API.

Perché serve: Evita over/under-fetch e problemi di lazy loading.

Nel progetto: `MatchDTO`, `UserProfileDTO`, `FeedbackDTO`; mapper manuali o MapStruct.

Trappole: esposizione di ID interni sensibili; cicli di serializzazione.

Check  campi minimi necessari; versionamento API se cambia il contratto.


11) Transazioni & Consistenza

Cosa è: Gruppi atomici di operazioni DB (`@Transactional`).

Perché serve: Stato consistente su errori e concorrenza.

Nel progetto: join/leave, conferma automatica, chiusura partita, inserimento feedback.

Trappole: transazioni troppo ampie; deadlock.

Check  granularità sensata; solo operazioni di scrittura nel boundary transazionale.


12) Testing & Coverage (JUnit, MockMvc, JaCoCo $\geq 80\%$)

Cosa è: Prove automatiche unit/integration, metrica di copertura.

Perché serve: Regressioni sotto controllo, documentazione eseguibile.

Nel progetto: - **Unit:** servizi (match/feedback/user), strategie, listener. - **Web:** `MockMvc` per endpoint principali. - **Coverage:** report JaCoCo nel build.

Trappole: test fragili dipendenti dall'ordine; mock eccessivi.

Check  Given-When-Then; dataset H2 per test; test casi limite (partita piena, doppio feedback, utente non trovato).

13) Logging & Observability

Cosa è: Tracciamento esecuzione con livelli (`INFO`, `WARN`, `ERROR`).

Perché serve: Debug, auditing, analisi post-mortem.

Nel progetto: log su eventi chiave (conferma/finish match, tentativi join oltre limite, feedback creati).

Trappole: PII in log; log rumoroso.

Check  correlation id per richiesta; mascherare dati sensibili.

14) Configurazione & Profili (dev/test/prod)

Cosa è: `application.yml` per ambienti diversi.

Perché serve: Parametri separati dal codice.

Nel progetto: profili `dev` (H2), `test` (H2 isolato), `prod` (RDBMS esterno), seeding solo dev.

Trappole: credenziali hardcoded; profilo errato in CI.

Check ☒ variabili ambiente; secrets; attivazione profili nel workflow di build.

15) UML essenziale a supporto

Cosa è: Vista statica/dinamica del sistema.

Nel progetto: Use Case (registrazione, join/leave, feedback), Class Diagram (model + pattern), Sequence (match confermato, feedback flow).

Trappole: diagrammi obsoleti rispetto al codice.

Check ☒ aggiornare quando cambia dominio; drop-in PNG/SVG nella relazione.

16) CI Build & Qualità (facoltativo ma consigliato)

Cosa è: Pipeline che esegue test, coverage, linter.

Nel progetto: GitHub Actions → `mvn test` + report JaCoCo artefatto. Gate $\geq 80\%$.

Trappole: pipeline lenta; test non deterministici.

Check ☒ cache Maven; seed deterministico; badge coverage nel README.

17) UX e Flussi chiave (punto di vista funzionale)

- **Match flow:** crea proposta → utenti si iscrivono → al 4° `CONFIRMED` → notifica → `FINISHED` → feedback → livello percepito aggiornato.
 - **Profilo utente:** livello dichiarato vs percepito, matchesPlayed, storico feedback.
 - **Lista partite:** filtri + `Strategy` di ordinamento via query param.
-

18) Checklist “Definition of Done” per feature tipiche

- Endpoint + validazioni + test (unit + web) passano.

- Invarianti dominio rispettate (limite 4, feedback unico).
 - Eventi e listener coperti da test.
 - Log & error handling coerenti.
 - Coverage del modulo $\geq 80\%$.
-

19) Glossario rapido

- **Entity**: oggetto persistente JPA.
 - **DTO**: oggetto scambio API.
 - **Repository**: interfaccia dati.
 - **Service**: logica di dominio.
 - **Controller**: interfaccia HTTP.
 - **Event**: fatto del dominio.
 - **Strategy**: algoritmo intercambiabile.
 - **Singleton**: istanza unica.
-

20) Template miniature (frasi pronte per relazione)

- *"Abbiamo adottato un'architettura MVC con separazione netta tra presentazione, dominio e persistenza, riducendo l'accoppiamento e massimizzando la testabilità."*
 - *"Gli eventi di dominio `MatchConfirmed` e `MatchFinished` disaccoppiano la logica di notifica e aggiornamento statistiche dal core dei servizi."*
 - *"Il criterio di ordinamento delle partite è estensibile tramite `Strategy`, selezionata a runtime mediante parametro `sort`."*
 - *"La qualità è monitorata da test JUnit e report JaCoCo con soglia $\geq 80\%$."*
-

21) Roadmap tecnica (prossime estensioni)

- Rate limiting & audit trail.
 - Scheduler per marcatura automatica `FINISHED`.
 - Coda async (es. Spring Events/AMQP) per notifiche email reali.
 - Pagina admin per partite fisse e moderazione feedback.
-

Fine artifact