**Implementarea arhitecturilor in proiect**

Tema : Grupa 12  
Aplicatie web de planificare

a evenimentelor sportive

[1. Arhitectura Software 3](#_Toc187694581)

[2. Arhitecturi folosite 4](#_Toc187694582)

[2.1 Layered Architecture 4](#_Toc187694583)

[2.2 Arhitectura bazata pe servicii (SOA) 4](#_Toc187694584)

[2.3 Dependency Injection 5](#_Toc187694585)

[2.4 Dependency Inversion Principle 5](#_Toc187694586)

[2.5 Model-View-Controller (MVC) 5](#_Toc187694587)

[2.6 Principiile SOLID 6](#_Toc187694588)

[2.7 Repository pattern 7](#_Toc187694589)

[2.8 Clean Architecture 7](#_Toc187694590)

# Arhitectura Software

Arhitectura software se referă la structura și organizarea unui sistem software, incluzând componentele sale, modul în care acestea interacționează între ele și modul în care sunt distribuite responsabilitățile în cadrul sistemului. Aceasta reprezintă un cadru de bază care guvernează dezvoltarea aplicației, ajutând la luarea deciziilor legate de design, scalabilitate, performanță și întreținere.

O arhitectură software definește:

* Componentele aplicației: Modulele și funcționalitățile individuale care formează aplicația.
* Interacțiunile dintre componente: Cum comunică și cum sunt integrate componentele între ele.
* Reguli și principii de design: Linii directoare care guvernează dezvoltarea, cum ar fi separarea preocupărilor, reutilizarea codului, sau gestionarea resurselor.
* Tehnologii și instrumente folosite: Deciziile legate de platforme, baze de date, librării sau framework-uri.

Arhitecturile software ajută la crearea unor aplicații scalabile, flexibile și ușor de întreținut, care pot evolua în timp pentru a răspunde cerințelor în schimbare. Exemple de arhitecturi software includ arhitecturi monolitice, microservicii, arhitectura pe straturi (layered architecture), sau arhitectura bazată pe evenimente (event-driven architecture)

# Arhitecturi folosite

## 2.1 Layered Architecture

**Layer-ul de Prezentare (API Controllers)**:

În acest layer, controller-ele gestionază cererile HTTP de la utilizatori și trimit răspunsuri. De exemplu, controller-ele pentru Field, Group și Match definesc metodele de interacțiune cu API-ul, cum ar fi adăugarea, actualizarea și obținerea entităților respective.

**Exemplu**: FieldController primește cereri pentru adăugarea unui nou câmp sau obținerea unui câmp specificat.

**Layer-ul de Servicii**:

În acest layer, se află logica de business. Fiecare serviciu (ex: FieldService, GroupService, MatchService) este responsabil pentru gestionarea logicii aplicației legată de entitățile corespunzătoare și pentru interacțiunea cu repository-urile.

**Exemplu**: FieldService se ocupă cu adăugarea, ștergerea și actualizarea câmpurilor și comunică cu repository-urile pentru a efectua aceste operații.

**Layer-ul de Acces la Date (Repoziții și Unit of Work)**:

Aici, repozițiile (de exemplu, FieldRepo, GroupRepo, MatchRepo) se ocupă cu interacțiunea directă cu baza de date. Ele permit adăugarea, ștergerea, actualizarea și obținerea entităților din baza de date.

**UnitOfWork** ajută la coordonarea repository-urile pentru a efectua toate operațiile de date, garantând consistența datelor.

**Exemplu**: FieldRepo interacționează direct cu tabela Field din baza de date pentru a adăuga sau actualiza câmpuri.

## Arhitectura bazata pe servicii (SOA)

În cadrul acestui proiect, fiecare entitate majoră este gestionată de un serviciu dedicat. De exemplu:

* FieldService se ocupă cu logica de business a entității Field.
* GroupService se ocupă cu logica de business a entității Group.
* MatchService se ocupă cu logica de business a entității Match.

Fiecare serviciu este responsabil pentru manipularea entităților sale corespunzătoare și pentru interacțiunea cu repozițiile pentru a efectua operațiile de CRUD (Create, Read, Update, Delete).

**Exemplu**: Dacă trebuie să adaugi un nou câmp, controller-ul FieldController va apela FieldService pentru a adăuga câmpul respectiv, iar FieldService va apela FieldRepo pentru a adăuga câmpul în baza de date.

## 2.3 Dependency Injection

Prin Dependency Injection, componentelor aplicației li se injectează dependențele externe (de exemplu, serviciile și repozițiile) fără ca ele să fie responsabile pentru crearea acestor dependențe. În acest proiect, DI se folosește pentru a injecța serviciile și repository în clasele care au nevoie de ele (de exemplu, controller-e sau alte servicii).

**Exemplu**: În FieldService, UnitOfWork este injectat în constructor, astfel că FieldService nu trebuie să știe cum să creeze o instanță a UnitOfWork, ci doar o primește ca dependență.

Acest lucru face aplicația mai flexibilă și mai ușor de testat, deoarece poți înlocui dependențele cu mock-uri în testele unitare.

## 2.4 Dependency Inversion Principle

Acesta este un principiu din SOLID care spune că modulele de nivel înalt nu trebuie să depindă de modulele de nivel scăzut, ci ambele trebuie să depindă de abstracții (interfețe).

**Exemplu**: Serviciile (FieldService, GroupService, etc.) nu depind direct de implementările repozițiilor (de exemplu, FieldRepo), ci de interfețele acestora (IFieldRepo). Acest lucru permite schimbarea implementărilor repozițiilor fără a afecta serviciile.

## 2.5 Model-View-Controller (MVC)

În acest proiect, Model-View-Controller este folosit pentru a organiza aplicația în trei componente principale:

* **Model**: Reprezintă datele și logica de business ale aplicației. De exemplu, entitățile Field, Group, Match sunt modelele aplicației, care sunt manipulați de serviciile corespunzătoare.
* **View**: Deși acest proiect nu include un view complet definit, ar putea fi vorba de interfața utilizatorului (HTML/JS) care interacționează cu API-ul.
* **Controller**: Controller-ele sunt responsabile pentru primirea cererilor HTTP și pentru interacțiunea cu serviciile. De exemplu, FieldController primește cereri pentru gestionarea câmpurilor și apelează FieldService pentru a executa operațiile necesare.

## 2.6 Principiile SOLID

În acest proiect, principiile SOLID sunt aplicate pentru a asigura un cod curat și ușor de întreținut:

* Single Responsibility Principle (SRP): Fiecare clasă are o singură responsabilitate. De exemplu, FieldService se ocupă doar cu logica legată de câmpuri, iar FieldRepo cu interacțiunea cu baza de date pentru câmpuri.
* Open/Closed Principle (OCP): Deschise pentru extensie înseamnă că, atunci când este nevoie de noi comportamente sau funcționalități, acestea ar trebui adăugate fără a modifica codul existent. În schimb, putem extinde comportamentul existent prin crearea de noi clase, metode sau module. Închise pentru modificare înseamnă că codul existent ar trebui să rămână neschimbat, chiar și atunci când se adaugă noi funcționalități. Modificările sau extinderile se fac prin adăugarea de noi componente, nu prin schimbarea celor deja existente.
* Liskov Substitution Principle (LSP): Subtipurile pot fi utilizate în locul tipurilor de bază fără a compromite funcționalitatea. De exemplu, IFieldService poate fi implementat de mai multe clase, dar fiecare implementare va respecta aceleași interfețe.
* Interface Segregation Principle (ISP): Interfețele sunt mai mici și mai specifice, fiecare interfață oferind doar metodele relevante pentru client. De exemplu, interfețele IFieldService, IGroupService sunt dedicate și nu conțin metode irelevante pentru domeniul lor.
* Dependency Inversion Principle (DIP): Modulele de nivel înalt nu depind de modulele de nivel scăzut, ci ambele depind de abstracții (interfețe).

## 2.7 Repository pattern

În acest proiect, Repository Pattern este implementat pentru a izola logica de acces la date de restul aplicației, ceea ce permite manipularea obiectelor fără a expune detalii specifice bazei de date. Acesta oferă o modalitate standardizată de a efectua operațiuni CRUD (Create, Read, Update, Delete) pentru entități precum Field, Group, Match, etc.

Fiecare entitate (de exemplu, Field, Group, Match) are un repository dedicat: FieldRepo, GroupRepo, MatchRepo, etc.

Aceste repository-uri moștenesc dintr-o clasă de bază, BaseRepo, care implementează funcționalitățile comune pentru accesul la date.

În proiect, avem de exemplu FieldRepo, care moștenește din BaseRepo<Field>. Acest lucru permite reutilizarea codului pentru operațiuni comune (adică logica CRUD) și oferă o structură clară pentru fiecare entitate din proiect.

## 2.8 Clean Architecture

Clean Architecture este un principiu de organizare a codului care promovează separarea clară a responsabilităților și o structură modulară, care facilitează testarea, întreținerea și extensibilitatea pe termen lung. Aceasta arhitectură se concentrează pe separarea logicii aplicației de detaliile implementării, cum ar fi accesul la baza de date, UI-ul sau alte tehnologiile externe.

* Entities Layer: Acesta este nucleul aplicației, unde se află modelele de business pure, adică obiectele care definesc datele și logica de business fără a depinde de alte module externe. În acest layer găsim:

Modelele de business (Field, Group, Match, Player, etc.) care reprezintă entitățile aplicației. Aceste modele sunt independente de orice tehnologie sau detaliu de implementare.

* Use Cases Layer: În acest layer se află logica aplicației specifică cazurilor de utilizare, care controlează fluxul și interacțiunile dintre entitățile de business. Este layer-ul care orchestrează logica de business și se ocupă cu toate operațiile care trebuie să fie efectuate pentru a îndeplini cerințele aplicației. Serviciile (cum ar fi FieldService, GroupService, MatchService) implementează cazurile de utilizare ale aplicației. Acestea conțin logica de business ce manipulează modelele de business (entitățile) și interacționează cu repository-urile.
* Interface Layer: Acesta conține interfețele și implementările externe care interacționează cu aplicația. Aici găsim tehnologii externe, cum ar fi API-urile, baze de date sau orice altă tehnologie cu care aplicația interacționează.

Controller-ele API-ului (FieldController, GroupController, MatchController) sunt implementate în acest layer. Ele sunt responsabile pentru primirea cererilor HTTP și trimiterea răspunsurilor corespunzătoare.

Repository-urile (FieldRepo, GroupRepo, MatchRepo) fac parte din acest layer, interacționând direct cu baza de date pentru a adăuga, actualiza sau șterge entități din baze de date. Deși rpository-urile sunt folosite în acest layer, ele sunt accesibile doar prin intermediul unor interfețe (prin principiul Dependency Injection).

* Frameworks and Details Layer

Acesta conține detaliile externe care implementează logica specifică a tehnologiilor utilizate, cum ar fi baze de date, framework-uri externe (ex. Entity Framework), interfețe utilizator sau alte servicii externe.

SportEnjoyersDatabaseContext folosește Entity Framework pentru a interacționa cu baza de date. Aceasta este implementarea concretă a unui strat de acces la date, iar FieldRepo sau GroupRepo sunt implementări care depind de aceasta.

Utilizarea Entity Framework pentru ORM este o parte a acestui strat, iar repozițiile interacționează cu baza de date pentru a salva sau actualiza datele