SINGLETON PATTERN

Descriere:

Singleton Pattern este un design pattern creațional care garantează că o clasă are o singură instanță pe întreaga durată a ciclului de viață al aplicației și oferă un punct global de acces la această instanță. Acest pattern este folosit pentru a controla accesul la o resursă partajată și pentru a preveni crearea mai multor instanțe ale unei clase care ar putea conduce la inconsistențe sau consum inutil de resurse.

Avantaje:

1. Unicitatea instanței: Singleton se asigură că doar o singură instanță a clasei este creată. Acest lucru este realizat prin controlul procesului de instanțiere, de obicei folosind un constructor privat și un mecanism static pentru a accesa instanța.
2. Punct global de acces: Singleton oferă un mecanism centralizat pentru a obține instanța unică a clasei, de obicei prin intermediul unei metode statice precum getInstance().
3. Ciclul de viață controlat: Instanța este creată fie la cerere (lazy initialization), fie în avans (eager initialization), în funcție de cerințele aplicației.

Implementare în proiect:

```
17  @Service  2 usages  Simina Dan-Marius *
18  public class MessageService {
19      private final MessageRepository messageRepository;  3 usages
20      private final UserRepository userRepository;  3 usages
21
22      @Autowired  Simina Dan-Marius
23      public MessageService(MessageRepository messageRepository, UserRepository userRepository) {
24          this.messageRepository = messageRepository;
25          this.userRepository = userRepository;
26      }
27
28      private MessageDTO convertToDTO(Message message) {...}
64
65      @Transactional  1 usage  Simina Dan-Marius *
66      public MessageDTO sendMessage(Integer senderId, Integer receiverId, String content) {...}
86
87      public List<MessageDTO> getMessagesBetweenUsers(Integer userId1, Integer userId2) {...}
93  }
```

Cum funcționează:

1. Spring creează o singură instanță a MovieService

2. Această instanță este injectată oriunde este necesară prin dependency injection
3. Toate componentele care folosesc MovieService primesc aceeași instanță
4. Spring gestionează lifecycle-ul acestei instanțe singleton

BUILDER PATTERN

Descriere:

Builder Pattern este un pattern creațional utilizat pentru a construi obiecte complexe într-un mod structurat și controlat. Acesta permite crearea unui obiect pas cu pas, oferind un nivel înalt de flexibilitate în procesul de construcție. Principalul său avantaj constă în separarea procesului de construcție a obiectului de reprezentarea acestuia, ceea ce permite crearea unor variații multiple ale aceluiași tip de obiect, fără a modifica codul clientului.

Avantaje:

1. Separarea responsabilităților: Procesul de construcție este separat de reprezentarea finală, ceea ce facilitează schimbările sau adăugările.
2. Control asupra procesului de construcție: Se pot defini pași detaliați pentru construirea obiectului, care să fie personalizați pentru diferite cerințe.
3. Crearea mai multor reprezentări: Permite construirea diferitelor variante ale obiectului final (de exemplu, prin utilizarea unor Concrete Builder diferite).
4. Citibilitate crescută: Codul este mai clar și mai ușor de întreținut

Implementare în proiect:

```
public class RecommendationDTO {  14 usages  Simina Dan-Marius
    private Integer id;  2 usages
    private UserDTO user;  2 usages
    private MovieDTO movie;  2 usages
    private String content;  2 usages
    private LocalDate createdAt;  2 usages
    private int commentCount;  2 usages

    // Constructors, getters, and setters
    public RecommendationDTO() {}  1 usage  Simina Dan-Marius

    // Getters and setters
    public Integer getId() { return id; }  Simina Dan-Marius
    public void setId(Integer id) { this.id = id; }  Simina Dan-Marius

    public UserDTO getUser() { return user; }  Simina Dan-Marius
    public void setUser(UserDTO user) { this.user = user; }  Simina Dan-Marius

    public MovieDTO getMovie() { return movie; }  Simina Dan-Marius
    public void setMovie(MovieDTO movie) { this.movie = movie; }  Simina Dan-Marius

    public String getContent() { return content; }  Simina Dan-Marius
    public void setContent(String content) { this.content = content; }  Simina Dan-Marius
```

FACADE PATTERN

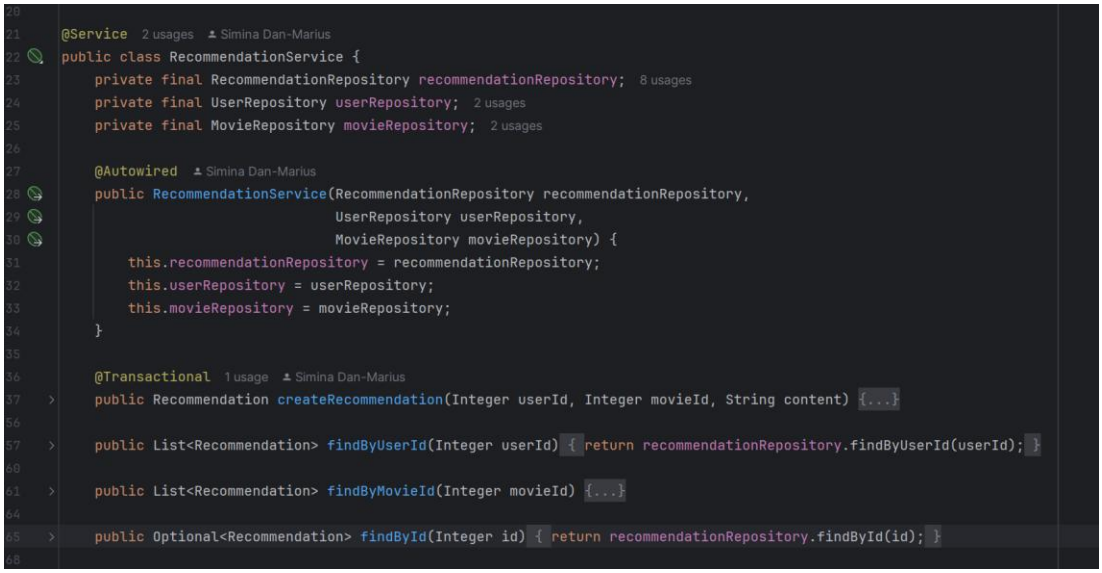
Descriere:

Facade Pattern este un design pattern structural care simplifică accesul la un sub-sistem complex prin oferirea unei interfețe unificate și simplificate. Scopul principal al acestui pattern este de a reduce complexitatea interacțiunilor între componentele unui sistem și de a promova un coupling slab (loose coupling) între clienți și subsistemele interne.

Principii și scopuri:

1. **Interfață simplificată:**
    - Facade Pattern ascunde detaliile de implementare ale subsistemului, expunând doar funcționalitatea esențială.
    - Clientul interacționează doar cu interfața oferită de **facadă**, fără să cunoască detaliile tehnice sau structura internă.
2. **Encapsularea complexității:**
    - Codul clientului este separat de implementările interne ale subsistemului, ceea ce îmbunătățește lizibilitatea și întreținerea codului.
3. **Promovarea coupling-ului slab:**
    - Clientul devine independent de modificările sau evoluțiile interne ale subsistemului, deoarece toate interacțiunile trec printr-un punct centralizat: fațada.

Implementare extinsă în proiect:



```
@Service  2 usages  Simina Dan-Marius
public class RecommendationService {
    private final RecommendationRepository recommendationRepository;  8 usages
    private final UserRepository userRepository;  2 usages
    private final MovieRepository movieRepository;  2 usages

    @Autowired  Simina Dan-Marius
    public RecommendationService(RecommendationRepository recommendationRepository,
                                 UserRepository userRepository,
                                 MovieRepository movieRepository) {
        this.recommendationRepository = recommendationRepository;
        this.userRepository = userRepository;
        this.movieRepository = movieRepository;
    }

    @Transactional  1 usage  Simina Dan-Marius
    public Recommendation createRecommendation(Integer userId, Integer movieId, String content) {...}

    public List<Recommendation> findByUserId(Integer userId) { return recommendationRepository.findByUserId(userId); }

    public List<Recommendation> findByMovieId(Integer movieId) {...}

    public Optional<Recommendation> findById(Integer id) { return recommendationRepository.findById(id); }
```

Cum funcționează Facade Pattern în acest context:

1. **Simplificare:**

Ascunde complexitatea interacțiunii cu multiple repository-uri

Oferă o interfață curată pentru operațiuni complexe

Gestionează tranzacții și validări

2. **Încapsulare:**

Logica de business este encapsulată în serviciu

Detaliile implementării sunt ascunse de client

Reduce dependențele și coupling-ul

3. **Centralizare:**

Oferă un punct central pentru logica de business

Facilitează mentenanța și modificările

Promovează reutilizarea codului

DIAGRAMA UML A PROIECTULUI: