Relazione Tecnica - Monster Hotel System

Progetto: Sistema di Gestione Prenotazioni Hotel **Studente**: Generation Italy - Java Course **Data**: Gennaio 2026 **Linguaggio**: Java (JDK 8+)

Introduzione

Monster Hotel è un sistema gestionale sviluppato in Java per gestire le prenotazioni di un hotel dedicato a creature soprannaturali. Il progetto nasce come compito delle vacanze con l'obiettivo di applicare concretamente i principi di programmazione orientata agli oggetti, il pattern architetturale MVC e le best practice dello sviluppo software professionale.

Il sistema permette di creare, visualizzare, cercare e persistere prenotazioni, gestendo vincoli complessi legati alle diverse specie di ospiti (vampiri, licantropi, sirene e umani) e calcolando automaticamente i costi in base a regole di business specifiche.

Scelte Architetturali

Il Pattern MVC

Ho scelto di implementare il pattern **Model-View-Controller** per separare nettamente le responsabilità del codice. Questa scelta è stata motivata dalla necessità di creare un'applicazione manutenibile e scalabile.

Controller Layer (`HotelWizard` e `BookingService`): Il controller orchestra il flusso dell'applicazione. `HotelWizard` gestisce il menu principale e delega le operazioni specifiche a `BookingService`, che contiene la logica di business. Ho diviso il service in metodi piccoli e focalizzati (come `askPersonalData()`, `askSpecies()`, ecc.) per rendere il codice più leggibile e manutenibile. Inizialmente avevo un unico metodo `createNewBooking()` di 140 righe, ora è ridotto a 12 righe che orchestrano 5 metodi helper.

Model Layer (`Booking`, `Species`, `RoomType`): Il modello rappresenta le entità di dominio. Ho utilizzato le **enum** per `Species` e `RoomType` perché in Java le enum non sono semplici costanti, ma vere e proprie classi che possono contenere logica. Ad esempio, ogni specie "sa" calcolare il proprio costo extra e validare i propri vincoli. Questo rende il codice molto più pulito rispetto a usare costanti stringa e lunghi if-else.

View Layer (`HotelView`): Ho centralizzato tutta la logica di visualizzazione in questa classe. Inizialmente avevo chiamate `Console.print()` sparse nel repository, ma poi ho capito che violava il principio MVC: il repository deve solo gestire dati, non occuparsi di come vengono mostrati. Ho quindi spostato

tutta la visualizzazione nel View, creando metodi come `showStatistics()`, `showSaveResult()`, ecc.

Data Layer (`BookingRepository`): Il repository gestisce la persistenza e le operazioni CRUD. Ho applicato il **Repository Pattern** per astrarre il modo in cui i dati vengono salvati. Attualmente uso un file di testo, ma domani potrei passare a un database cambiando solo questa classe, senza toccare il Service o il Controller.

Strumenti Java e Motivazioni

Enum Avanzati

La scelta più interessante è stata usare le **enum** per modellare Species e RoomType. In Java, le enum possono avere costruttori, campi e metodi. Ad esempio:

```
VAMPIRE(0.0, 0.0, " ", "Vampiro")
```

Ogni specie ha un costo percentuale extra, un costo fisso, un'emoji e un nome. Ma soprattutto ha metodi come:

```
public boolean canStayOnFloor(int floor) {
    return this == VAMPIRE ? floor < 0 : true;
}
```

Questo significa che ogni specie "sa" validare se stessa. È molto più elegante e manutenibile rispetto a usare stringhe magiche e switch sparsi nel codice. Se domani dovessi aggiungere gli Zombie, basta aggiungere una costante enum, e tutta la logica di calcolo costi e validazione funziona automaticamente.

LocalDate (java.time)

Ho usato `LocalDate` invece delle vecchie classi `Date` perché è immutabile, thread-safe e ha un'API molto più pulita. Calcolare la differenza tra due date è semplicissimo:

```
ChronoUnit.DAYS.between(arrivalDate, departureDate)
```

Inoltre, il parsing da stringa è robusto e segue lo standard ISO-8601. Gestisco gli errori di formato con try-catch, permettendo all'utente di riprovare senza far crashare il programma.

ArrayList e Copia Difensiva

Uso `ArrayList<Booking>` per memorizzare le prenotazioni in memoria. Un aspetto importante è la **copia difensiva** nel metodo `getAllBookings()`:

```
return new ArrayList<>(bookings);
```

Restituisco una copia della lista, non il riferimento alla lista interna. Questo previene che codice esterno possa modificare la mia lista privata, mantenendo l'incapsulamento.

Gestione Eccezioni

Ho gestito le eccezioni in modo da rendere l'applicazione robusta. Ad esempio, quando chiedo una data all'utente, il parsing può fallire. Invece di far crashare tutto, catturo l'eccezione e chiedo di riprovare:

```
while (dob == null) {
    try {
        dob = LocalDate.parse(Console.readString());
    } catch (Exception e) {
        Console.print("Formato non valido! Riprova: ");
    }
}
```

Lo stesso vale per il caricamento file: se manca, non è un errore fatale, semplicemente non ci sono prenotazioni da caricare.

Logica di Business

Vincoli per Specie

Ogni specie ha vincoli specifici implementati tramite metodi nell'enum:

- **Vampiri**: possono stare solo su piani negativi (sotterranei), per evitare la luce del sole
- **Licantropi**: non possono stare in stanze singole, hanno bisogno di spazio per la trasformazione
- **Sirene**: nessun vincolo specifico, ma +50% sul totale (costo piscina/vasca speciale)
- **Umani**: nessun vincolo, ma +100€ fissi (non ci fidiamo di loro!)

Questi vincoli vengono validati nel metodo `Booking.validate()`, che chiama i metodi dell'enum per verificare compatibilità.

Calcolo Costi

Il calcolo del costo totale segue questa formula:

```
Base = (Prezzo Stanza × Notti) + Costo Navetta
Totale = Base + Extra Specie
```

L'extra specie può essere percentuale (sirene +50%) o fisso (umani +100€). Ho delegato questo calcolo all'enum Species:

```java
public double calculateTotalCost(double baseCost) {
    return baseCost + calculateExtraCost(baseCost);
}
```

Questo rende il codice molto pulito: `Booking` calcola il costo base e delega a `Species` il calcolo dell'extra.

Validazione

La validazione è centralizzata nel metodo `Booking.validate()` che restituisce una stringa con tutti gli errori trovati. Se la stringa è vuota, la prenotazione è valida. Questo approccio permette di mostrare all'utente **tutti** gli errori in una volta, invece che uno alla volta.

Persistenza Dati

Ho implementato un sistema di persistenza semplice ma efficace usando file di testo con formato pipe-delimited:

```
1|Mario|Rossi|1990-05-10|VAMPIRE|SINGOLA|-1|2025-01-15|2025-01-20|true
```

Serializzazione

Il metodo `saveBookings()` scrive ogni prenotazione come una riga, separando i campi con `|`. Ho scelto questo separatore perché non compare mai nei dati (nomi, date, ecc.).

Deserializzazione

Il metodo `loadBookings()` legge il file riga per riga, fa `split("\\|")` e ricostruisce gli oggetti `Booking`. Uso: - `Integer.parseInt()` per gli ID e il piano - `LocalDate.parse()` per le date - `Species.valueOf()` per convertire la stringa nell'enum corrispondente - `Boolean.parseBoolean()` per la navetta

Il bello dell'enum è che `valueOf()` fa automaticamente la conversione stringa → enum.

Gestione ID

Quando carico le prenotazioni, trovo l'ID massimo e imposto `nextId = maxId + 1`. Questo garantisce che nuove prenotazioni abbiano sempre ID univoci, anche dopo riavvii dell'applicazione.

Il DTO Pattern

Ho introdotto `StatisticsData` come **Data Transfer Object**. Questo è un oggetto che contiene solo dati (campi pubblici) senza logica di business complessa. Serve per trasferire dati dal Repository al View passando per il Service.

Prima avevo il Repository che calcolava E visualizzava le statistiche. Ora: 1. Repository calcola e restituisce `StatisticsData` 2. Service riceve il DTO e lo passa al View 3. View formatta e visualizza i dati

Questo rispetta la separazione delle responsabilità: il Repository non sa come vengono visualizzati i dati, e il View non sa come vengono calcolati.

Refactoring Effettuati

Divisione createNewBooking()

Il metodo più grande era `createNewBooking()` con 140 righe. L'ho diviso in 5 metodi privati: - `askPersonalData()` - nome, cognome, data nascita - `askSpecies()` - selezione specie - `askRoomDetails()` - tipo stanza e piano - `askStayDates()` - date e navetta - `validateAndSave()` - validazione finale

Ora il metodo principale è semplicissimo:

```
public void createNewBooking() {
    // Setup
    askPersonalData(booking);
    askSpecies(booking);
    askRoomDetails(booking);
    askStayDates(booking);
    validateAndSave(booking);
}
```

Questo approccio è chiamato **Template Method Pattern**: definisci lo skeleton dell'algoritmo e deleghi i dettagli a metodi privati.

Rimozione Console.print dal Repository

Inizialmente `BookingRepository` aveva metodi come:

```
public void saveBookings() {
    Console.print("Salvataggio...");
    // logica salvataggio
    Console.print("Salvato!");
}
```

Questo violava MVC: il Repository non dovrebbe occuparsi di visualizzazione. Ho trasformato i metodi per restituire dati:

```
public boolean saveBookings() {
    // logica salvataggio
    return success;
}
```

E ho creato metodi nel View per mostrare i risultati:

```
HotelView.showSaveResult(success, count);
```

Ora il Repository è completamente riutilizzabile: potrei usarlo in un'app web senza modifiche.

Principi SOLID Applicati

Single Responsibility Principle

Ogni classe ha una sola ragione per cambiare: - `Booking` cambia se cambia la struttura di una prenotazione - `BookingRepository` cambia se cambia il modo di salvare/caricare - `HotelView` cambia se cambia il modo di visualizzare - `BookingService` cambia se cambia il workflow di creazione

Open/Closed Principle

Il sistema è aperto all'estensione ma chiuso alla modifica. Per aggiungere una nuova specie:

```
ZOMBIE(20.0, 50.0, " ", "Zombie")
```

Basta aggiungere questa riga all'enum. Tutto il resto (calcolo costi, validazione, statistiche) funziona automaticamente senza modificare codice esistente.

Dependency Inversion

`BookingService` dipende da `BookingRepository`, ma non da dettagli implementativi. Il Service non sa se il Repository usa file, database o API. Questo rende facile sostituire l'implementazione.

Funzionalità Implementate

Il sistema offre 8 funzionalità principali accessibili da menu:

Nuova Prenotazione: Wizard step-by-step con validazione in tempo reale. Mostra avvisi preventivi (es: "I vampiri possono stare solo su piani negativi") e gestisce errori di formato con possibilità di riprovare.

Elenco Prenotazioni: Mostra tutte le prenotazioni con dettagli formattati. Gestisce elegantemente il caso "nessuna prenotazione".

Cerca per ID: Permette di trovare rapidamente una prenotazione specifica. Gestisce il caso "non trovata" con messaggio friendly.

Salva su File: Persiste le prenotazioni in formato testo. Gestisce il caso "lista vuota" e eventuali errori I/O.

Carica da File: Ripristina le prenotazioni salvate. Gestisce file mancante senza crashare.

Genera HTML: Crea un documento HTML stampabile della prenotazione. Crea automaticamente la directory `print/` se mancante.

Statistiche: Calcola e mostra: - Distribuzione per specie (quanti vampiri, licantropi, ecc.) - Distribuzione per tipo stanza - Ricavi totali e medi - Notti totali prenotate - Servizi navetta richiesti

---

Gestione degli Errori

Ho implementato una gestione errori robusta a più livelli:

Input Date: Try-catch con loop finché l'utente inserisce una data valida.

Validazione Business: Il metodo `validate()` accumula tutti gli errori e li mostra insieme, così l'utente può correggerli tutti in una volta.

File I/O: Catturo eccezioni di lettura/scrittura e mostro messaggi user-friendly invece di stack trace incomprensibili.

Ricerche: Gestisco il caso "non trovato" con messaggi chiari invece di NullPointerException.

Struttura del Codice

Ho organizzato il codice in package logici:

```
com.generation.mh.controller   → HotelWizard, Service, Repository
com.generation.mh.model.entities → Booking, Species, RoomType
com.generation.mh.model.dto      → StatisticsData
com.generation.mh.view           → HotelView
```

Questa organizzazione rende chiaro il ruolo di ogni classe. Un nuovo sviluppatore può capire immediatamente dove cercare cosa.

Testing Manuale

Ho testato il sistema con vari scenari:

Scenario 1 - Vampiro piano sbagliato: Creato prenotazione vampiro su piano 3 → validazione blocca con errore chiaro.

Scenario 2 - Licantropo stanza singola: Creato prenotazione licantropo in singola → validazione blocca.

Scenario 3 - Date invalide: - Data arrivo nel passato → bloccata - Data partenza prima dell'arrivo → bloccata

Scenario 4 - Calcolo costi: - Sirena, 3 notti, doppia, navetta → verificato calcolo +50% corretto - Umano, 2 notti, suite → verificato +100€ fissi

Scenario 5 - Persistenza: - Creato 3 prenotazioni → salvate → chiuso programma → riaperto → caricate correttamente

Scenario 6 - Statistiche: Creato prenotazioni miste → statistiche mostrano conteggi corretti per specie/stanze.

Cosa Ho Imparato

Concetti Java

Ho consolidato la comprensione di **enum avanzati**: non sono semplici costanti ma classi complete. Ho capito la potenza di `LocalDate` rispetto alle vecchie API. Ho praticato la gestione delle eccezioni in modo user-friendly invece che tecnico. Ho usato generics con `ArrayList<Booking>` capendo l'importanza del type safety.

Design Patterns

Ho applicato concretamente **MVC**, capendo l'importanza di separare le responsabilità. Ho implementato il **Repository Pattern** che astrae la persistenza. Ho usato **DTO** per trasferire dati tra layer. Ho applicato **Template Method** per suddividere algoritmi complessi.

Principi SOLID

Ho applicato **Single Responsibility** dividendo le classi per responsabilità chiare. Ho visto **Open/Closed** in azione con le enum estendibili. Ho usato **Dependency Inversion** facendo dipendere il Service da astrazioni e non dettagli.

Best Practices

Ho imparato l'importanza della **validazione centralizzata**. Ho capito il valore della **copia difensiva** per proteggere lo stato interno. Ho applicato la **gestione errori graceful** che non fa crashare il programma. Ho scritto **commenti Javadoc** completi per documentare il codice.

Conclusioni

Questo progetto mi ha permesso di applicare concretamente i concetti teorici appresi durante il corso. Ho capito che scrivere codice che funziona è solo l'inizio: scrivere codice **manutenibile, scalabile e ben strutturato** richiede una progettazione attenta e l'applicazione di pattern e principi consolidati.

La divisione in layer MVC, inizialmente sembrata complicata, si è rivelata fondamentale: ogni modifica successiva (come spostare la visualizzazione dal Repository al View) è stata possibile proprio grazie alla separazione delle responsabilità.

L'uso delle enum per modellare Species e RoomType è stata la scelta più felice: ha reso il codice elegante, type-safe e facile da estendere.

Infine, il refactoring continuo (dividere metodi lunghi, rimuovere codice duplicato, spostare responsabilità) mi ha insegnato che il codice è un'entità viva che va continuamente migliorata, non un artefatto statico da scrivere una volta sola.

Fine Relazione

Progetto Monster Hotel - Generation Italy Java Course - Gennaio 2026