

**Università degli Studi di Salerno**  
Corso di Ingegneria del Software

**AniTour**  
**Object Design Document (ODD)**  
**Versione 1.3**



Data: 15/02/2026

Progetto: AniTour	Versione: 1.3
Documento: ODD	Data: 15/02/2026

# Indice

1. Introduction
  - 1.1 Object design trade-offs
  - 1.2 Interface documentation guidelines
  - 1.3 Definitions, acronyms, and abbreviations
  - 1.4 References
2. Packages
3. Class interfaces
  - 3.1 BookingControl
    - 3.1.1 processCheckout
    - 3.1.2 getUserHistory
  - 3.2 IPaymentGateway
    - 3.2.1 processPayment
  - 3.3 IBookingRepository
    - 3.3.1 save
    - 3.3.2 checkAvailability
  - 3.4 UserControl
    - 3.4.1 login
    - 3.4.2 register
  - 3.5 IUserRepository
    - 3.5.1 findByEmail
  - 3.6 CatalogControl
    - 3.6.1 searchTours
    - 3.6.2 createTour
  - 3.7 ICatalogRepository
    - 3.7.1 findAll
  - 3.8 NotificationControl
    - 3.8.1 sendOrderConfirmation
  - 3.9 IMailServer
    - 3.9.1 send
4. Glossary

# 1. Introduction

## 1.1 Object design trade-offs

In questa fase di design sono state prese le seguenti decisioni architetturali per bilanciare le prestazioni, la manutenibilità e i costi:

- **Buy vs Build:** Per la gestione dei pagamenti è stato scelto di utilizzare un adattatore verso servizi esterni (IPaymentGateway) invece di implementare un sistema bancario proprietario, riducendo i rischi di sicurezza e i tempi di sviluppo.
- **Gestione della Sessione:** Si è scelto di mantenere lo stato della sessione (es. Carrello) lato server ma identificato tramite token, bilanciando la sicurezza con la scalabilità.
- **Tipizzazione Forte vs Flessibilità:** L'uso di interfacce rigide (Interface e DTO) è stato preferito alla flessibilità dinamica per garantire contratti chiari tra i sottosistemi e facilitare il testing unitario.

## 1.2 Interface documentation guidelines

Le interfacce sono documentate seguendo queste convenzioni:

- **Naming:** I metodi usano la notazione *camelCase*. Le classi di controllo terminano con Control, le interfacce di persistenza con Repository.
- **Contratti:** Ogni operazione pubblica è specificata tramite **OCL (Object Constraint Language)**, definendo:
  - **Pre-conditions (pre:):** Condizioni che devono essere vere prima dell'esecuzione.
  - **Post-conditions (post:):** Condizioni garantite dal sistema dopo l'esecuzione con successo.
  - **Invariants (inv:):** Condizioni che devono rimanere sempre vere per l'oggetto.

## 1.3 Definitions, acronyms, and abbreviations

- **OCL:** Object Constraint Language.
- **DTO:** Data Transfer Object (usato per passare dati complessi, es. UserDTO).
- **DAO:** Data Access Object (implementato tramite i Repository).
- **Control:** Stereotipo per le classi che gestiscono la logica di business.

## 1.4References

- AniTour - Requirements Analysis Document (RAD), v1.3.
- AniTour - System Design Document (SDD), v1.1.

## 2. Packages

Il sistema è decomposto nei seguenti package logici che raggruppano le classi per responsabilità, conformemente all'architettura ECB definita nel SDD:

- **com.anitour.control:** Contiene le classi di controllo dei sottosistemi che gestiscono la logica di business e le Servlet per l'interazione con il frontend web.
- **com.anitour.model:** Contiene le entità del dominio e gli oggetti per il trasferimento dei dati.
- **com.anitour.dao:** Contiene sia le interfacce per l'accesso ai dati e ai servizi esterni, sia le loro implementazioni concrete per il DBMS Relazionale o per la simulazione dei servizi.

## 3. Class interfaces

### 3.1 BookingControl

#### Overview

Classe di controllo (stereotype <<Control>>) responsabile del coordinamento del processo di acquisto. Gestisce la transazione logica, l'interazione con il carrello e l'invocazione dei servizi di pagamento.

#### Dependencies

- IPaymentGateway (per processare i pagamenti)
- IBookingRepository (per la persistenza)
- CartManager (per recuperare i dati di sessione)

#### Operations & Contracts:

##### 3.1.1 processCheckout

```
public Booking processCheckout(Cart cart, PaymentDTO paymentData)
throws Exception
```

- **Descrizione:** Finalizza l'acquisto convertendo il carrello corrente in un ordine confermato.
- **OCL Contract:**

```
context BookingControl::processCheckout(cart,
paymentData)
```

```

pre:
    -- Il carrello non deve essere nullo e deve
    contenere elementi
    cart <> null and cart.isEmpty() = false
pre:
    -- I dati di pagamento devono essere validi
    sintatticamente
    paymentData <> null and paymentData.isValid() =
    true

post:
    -- Il numero di ordini nel repository è
    incrementato di 1 rispetto all'inizio
    self.bookingRepository.count() =
    self.bookingRepository.count()@pre + 1
post:
    -- L'oggetto restituito (l'ordine) ha lo stato
    CONFIRMED
    result.status = BookingStatus.CONFIRMED
post:
    -- Il carrello passato come parametro è ora
    vuoto
    cart.isEmpty() = true

```

### 3.1.2 getUserHistory

```
public List<Booking> getUserHistory(int userId)
```

- **Descrizione:** Restituisce la lista delle prenotazioni effettuate da un utente.
- **OCL Contract:**

```

context BookingControl::getUserHistory(userId)
pre:
    -- L'ID utente deve essere positivo ed esistere
    nel sistema
    userId > 0 and
    self.userRepository.exists(userId) = true
post:
    -- La lista restituita non è mai nulla (al
    massimo è vuota)
    result <> null

```

## 3.2 IPaymentGateway

**Overview:** Interfaccia (stereotype <<Interface>>) che astrae i servizi di pagamento esterni (Pattern Adapter). Permette al sistema di essere indipendente dal provider specifico (es. Stripe, PayPal).

## Operations & Contracts:

### 3.2.1 processPayment

```
boolean processPayment(double amount, PaymentDTO  
cardDetails)
```

- **Descrizione:** Tenta di addebitare l'importo sulla carta fornita.
- **OCL Contract:**

```
context IPaymentGateway::processPayment(amount,  
cardDetails)  
pre:  
    -- L'importo deve essere positivo  
    amount > 0.00  
post:  
    -- Il metodo restituisce true solo se la banca  
    autorizza, altrimenti false  
    result = true implies  
    (BankSystem.authorize(amount, cardDetails) = true)
```

## 3.3 IBookingRepository

**Overview:** Interfaccia (stereotype <<Interface>>) per l'accesso ai dati persistenti (Pattern DAO/Repository). Isola la logica di business dai dettagli del database MySQL.

## Operations & Contracts:

### 3.3.1 Save

```
int save(Booking booking)
```

- **Descrizione:** Persiste una nuova prenotazione nel database.
- **OCL Contract:**

```
context IBookingRepository::save(booking)  
pre:  
    -- L'oggetto booking non deve essere nullo  
    booking <> null  
post:  
    -- Restituisce un ID positivo che rappresenta la  
    chiave primaria generata  
    result > 0
```

### 3.3.2 checkAvailability

```
boolean checkAvailability(int tourId, int requiredSeats)
```

- **Descrizione:** Verifica se un tour ha abbastanza posti liberi.

- **OCL Contract:**

```
context
IBookingRepository::checkAvailability(tourId,
requiredSeats)
pre:
    requiredSeats > 0
post:
    -- Restituisce true se i posti disponibili nel
    DB sono sufficienti
    result = (self.tours.select(t | t.id =
tourId).seatsAvailable >= requiredSeats)
```

### 3.4 UserControl

**Overview:** Classe di controllo (stereotype <<Control>>) responsabile della gestione degli utenti. Gestisce i casi d'uso di autenticazione (UC1) e registrazione (UC2).

**Dependencies:**

IUserRepository (per la persistenza e verifica credenziali)

**Operations & Contracts:**

#### 3.4.1 Login

```
public String login(String email, String password)
throws AuthException
```

- **Descrizione:** Autentica un utente nel sistema e restituisce un token di sessione.
- **OCL Contract:**

```
context UserControl::login(email, password)
pre:
    -- Email e password non devono essere vuoti
    email <> null and email <> "" and password <>
null
post:
    -- Se le credenziali sono corrette, restituisce
    un token valido
    self.userRepository.verifyCredentials(email,
hash(password)) = true
    implies result <> null
```

#### 3.4.2 Register

```
public boolean register(UserDTO userData) throws
DuplicateUserException
```



- **Descrizione:** Registra un nuovo utente (Cliente o Organizzatore) nel sistema.
- **OCL Contract:**

```
context UserControl::register(userData)
pre:
    -- I dati devono essere validi e l'email non
    deve esistere
    userData.isValid() = true and
    self.userRepository.exists(userData.email) = false
post:
    -- Il numero di utenti nel sistema è aumentato
    di 1
    self.userRepository.count() =
    self.userRepository.count()@pre + 1
post:
    -- L'operazione restituisce true (successo)
    result = true
```

### 3.5 IUserRepository

**Overview:** Interfaccia (stereotype <<Interface>>) per l'accesso ai dati degli utenti (Pattern DAO).

#### Operations & Contracts:

##### 3.5.1 findByEmail

```
public User findByEmail(String email)
```

- **Descrizione:** Recupera un'entità Utente data la sua email.
- **OCL Contract:**

```
context IUserRepository::findByEmail(email)
pre:
    email <> null
post:
    -- Restituisce l'utente se esiste, altrimenti
    null
    result.email = email or result = null
```

### 3.6 CatalogControl

**Overview:** Classe di controllo (stereotype <<Control>>) per la gestione del Catalogo Tour. Permette la ricerca ai clienti e la creazione di nuovi tour agli organizzatori.

#### Dependencies:

- ICatalogRepository (per l'accesso ai tour)
- IUserRepository (per verificare i permessi dell'organizzatore)

## Operations & Contracts:

### 3.6.1 searchTours

```
public List<Tour> searchTours(SearchFilterDTO filters)
```

- **Descrizione:** Cerca i tour disponibili in base ai filtri (keyword, prezzo, tema).
- **OCL Contract:**

```
context CatalogControl::searchTours(filters)
pre:
    -- I filtri non devono essere nulli
    filters <> null
post:
    -- Restituisce una lista (eventualmente vuota)
    ma mai null
    result <> null
post:
    -- Tutti i tour restituiti devono essere in
    stato 'PUBLISHED'
    result->forAll(t | t.status =
    TourStatus.PUBLISHED)
```

### 3.6.2 createTour

```
public int createTour(TourDTO tourData, int organizerId)
throws ValidationException
```

- **Descrizione:** Permette a un organizzatore di creare un nuovo tour.
- **OCL Contract:**

```
context CatalogControl::createTour(tourData,
organizerId)
pre:
    -- Il prezzo deve essere positivo e la data fine
    successiva alla data inizio
    tourData.price > 0 and tourData.endDate >
    tourData.startDate
pre:
    -- L'utente richiedente deve essere un
    Organizzatore
    self.userRepository.findById(organizerId).type =
    UserType.ORGANIZER
post:
    -- Viene creato un nuovo tour nel sistema
```

```
self.catalogRepository.count() =  
self.catalogRepository.count()@pre + 1  
post:  
  -- Il tour creato nasce in stato 'DRAFT' (Bozza)  
  self.catalogRepository.findById(result).status =  
  TourStatus.DRAFT
```

### 3.7 ICatalogRepository

**Overview:** Interfaccia (stereotype <<Interface>>) per la persistenza dei Tour.

#### Operations & Contracts:

##### 3.7.1 findAll

```
public List<Tour> findAll()
```

- **Descrizione:** Restituisce tutti i tour presenti nel database.
- **OCL Contract:**

```
context ICatalogRepository::findAll()  
post:  
  result <> null
```

### 3.8 NotificationControl

**Overview:** Classe di controllo (stereotype <<Control>>) responsabile della gestione delle notifiche asincrone verso gli utenti (e-mail di conferma, invio voucher).

**Dependencies:** IMailServer (Interfaccia verso il server SMTP esterno)

#### Operations & Contracts:

##### 3.8.1 sendOrderConfirmation

```
public void sendOrderConfirmation(String recipientEmail,  
Booking bookingDetails)
```

- **Descrizione:** Invia una email di conferma ordine contenente il voucher in allegato.
- **OCL Contract:**

```

context
NotificationControl::sendOrderConfirmation(recipientEmail,
bookingDetails)
pre:
    -- L'email deve essere valida e l'ordine deve essere
    confermato
    recipientEmail <> null and bookingDetails.status =
    BookingStatus.CONFIRMED
post:
    -- La coda di invio contiene un messaggio destinato a
    questa email
    self.mailQueue->exists(m | m.recipient =
    recipientEmail and m.content = bookingDetails)

```

### 3.9 IMailServer

**Overview:** Interfaccia (stereotype <<Interface>>) che astrae la comunicazione con il server di posta elettronica (SMTP).

#### Operations & Contracts:

##### 3.9.1 Send

```
public boolean send(EmailDTO email)
```

- **Descrizione:** Inoltra fisicamente l'email al server SMTP.
- **OCL Contract:**

```

context IMailServer::send(email)
pre:
    email.recipient <> null and email.body <> null
post:
    -- Restituisce true se il server SMTP accetta il
    messaggio
    result = true

```

## 4. Glossary

- **Booking:** L'entità che rappresenta una prenotazione confermata e pagata.
- **Cart (Carrello):** Contenitore temporaneo dei tour selezionati dall'utente prima del pagamento.
- **Guest:** Un utente non autenticato che visita il sito.
- **Organizer:** Un utente con privilegi speciali (registrato con P.IVA) che può creare e gestire i propri Tour.
- **Otaku/Nerd:** Il target di riferimento di AniTour; influenza le categorie e i temi dei viaggi.

- **Session Token:** Identificativo univoco generato al login per mantenere lo stato dell'utente.
- **Tour:** Il prodotto principale venduto dalla piattaforma, caratterizzato da tappe, date, prezzo e un tema specifico.

## Revision History

Data	Versione	Descrizione	Autore
19/12/2025	1.0	Creazione documento ODD. Specifica delle interfacce dei moduli del sottosistema da implementare	Salvatore Merola
16/01/2026	1.1	Completamento documento ODD. Estensione della specifica delle interfacce a tutti i sottosistemi.	Salvatore Merola
12/02/2026	1.2	Rifattorizzazione delle post-condizioni da imperative a dichiarative (stati booleani) per tutti i sottosistemi.	Salvatore Merola
15/02/2026	1.3	Correzione minore alla sezione 2. Packages a seguito dell'implementazione.	Salvatore Merola

### Partecipanti:

Nome	Matricola
Vincenzo Chiocca	0512119182
Salvatore Merola	0512120979