Università degli Studi di Salerno
Corso di Ingegneria del Software

Esbet START
Object Design Document
Versione 2.2

Data: 31/01/2025

Progetto: EsbetSTART	Versione: 2.0
Documento: Object Design Document	Data: 31/01/2025

Coordinatore del progetto:

Nome	Matricola

Partecipanti:

Nome	Matricola
Aversana Marco	0512118978
Petrillo Francesco	0512116329
Polise Michele	0512116854
Zaccaro Valeria	0512117733

Scritto da:	

Revision History

Data	Versione	Descrizione	Autore
25/11/2024	1.0	Prima stesura ODD	
20/12/2024	1.1	Inserimento dell'Object model	
05/01/2025	1.2	Aggiunta suddivisione in package	
12/01/2025	2.0	Modificata l'interfaccia di certi servizi	
15/01/2025	2.1	Aggiunti design patterns	
31/01/2025	2.2	Ultima revisione	

1. Introduzione

1.1. Purpose of the system

EsbetSTART si propone di presentarsi come un sito di scommesse dedicato agli e-sports. Come in ogni altro *bookmaker* si metterà a disposizione dell'utente registrato la possibilità di piazzare scommesse, gestire il proprio conto e il proprio profilo e partecipare a offerte proposte.
Il sito verrà gestito da utenti con privilegi, che potranno gestire gli eventi e le offerte disponibili.

1.2. Design objectives

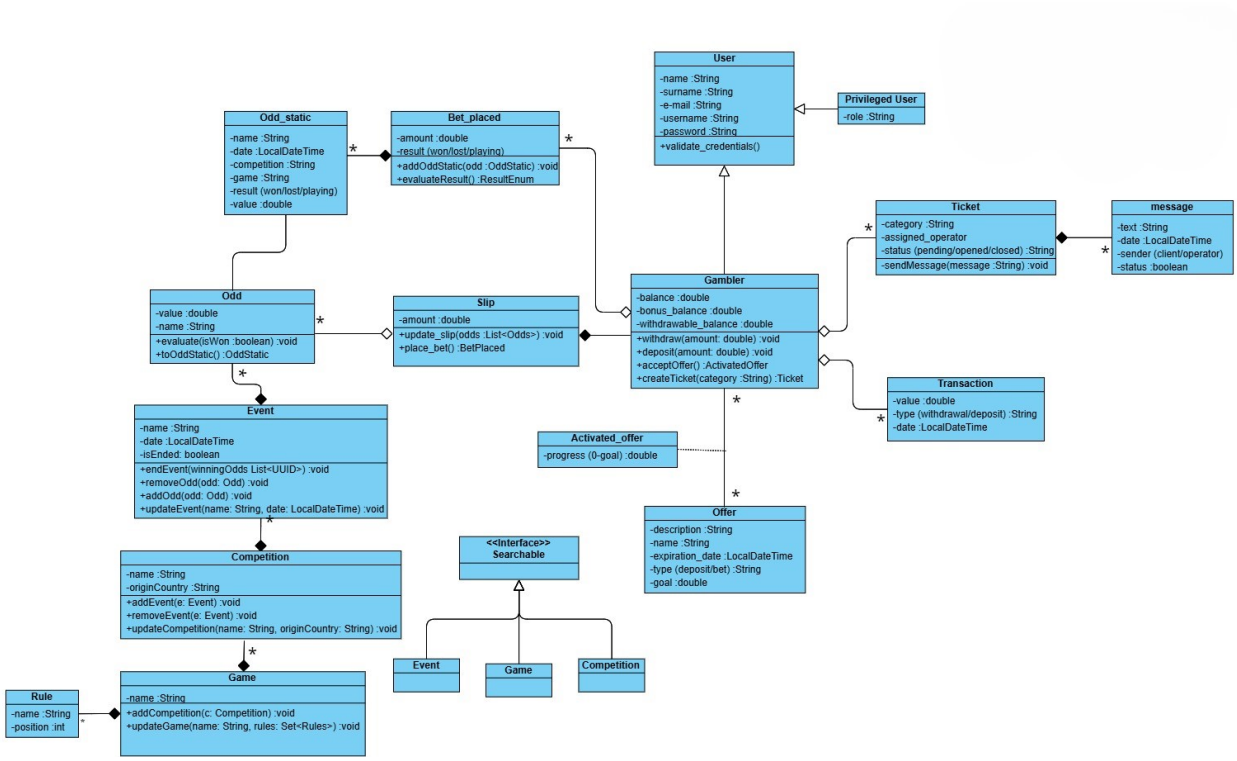
Il sistema dovrà essere progettato in maniera che garantisca:

- **Usabilità**
  - Tutti gli input inviati al sistema dovranno essere validati ed eventuali errori notificati in maniera repentina, soprattutto se coinvolgono il piazzamento delle scommesse. L'interfaccia utente dovrà essere intuitiva ed invitante, facile da navigare e responsive per essere usata su qualunque dispositivo.

- **Affidabilità**
  - Nessun errore dovrà rimanere ingestito, ogni caso di malfunzionamento dovrà garantire la consistenza dei dati persistenti e tutte le transazioni dovranno seguire il paradigma ACID.

- **Sicurezza**
  - Il sistema dovrà essere sicuro, sia nella criptazione dei dati sensibili che nella struttura in sé, impedendo per quanto possibile attacchi esterni che potrebbero comprometterne l'integrità.

1.3. Object model optimization



Si presenta qui il diagramma delle classi con interfacce per i metodi e gli attributi di ogni classe, rispetto a quello definito in fase di analisi ci sono state due modifiche:

- Le regole hanno ora un attributo “position” che permette una visualizzazione ordinata delle quote negli eventi.
- Gli eventi hanno ora l’attributo “isEnded”, che permette di capire se un evento è terminato prima di cancellarlo o terminarlo nuovamente per errore.
- Odd ora ha un metodo “toOddStatic” che restituisce un oggetto oddstatic con le sue caratteristiche, questo per permettere una più semplice gestione delle scommesse

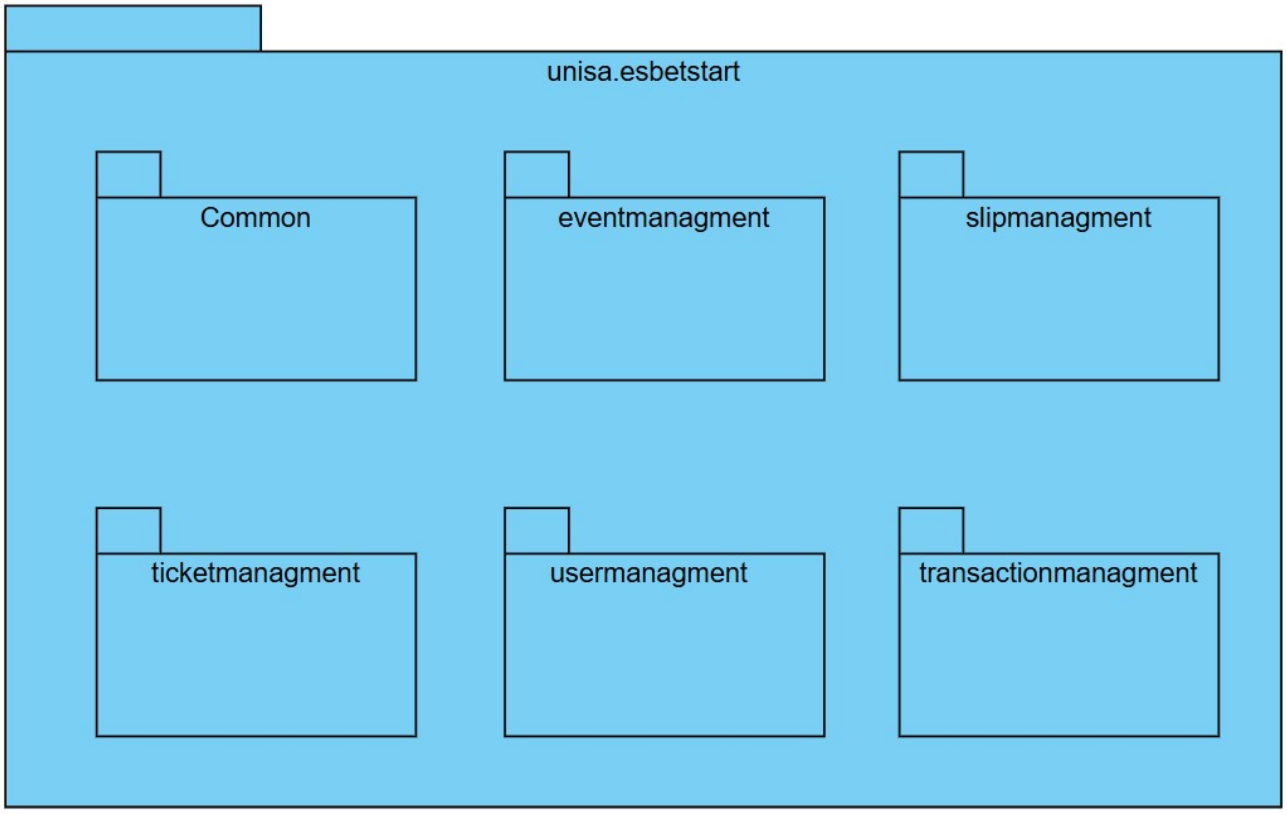
2. Packages

2.1. Project structure

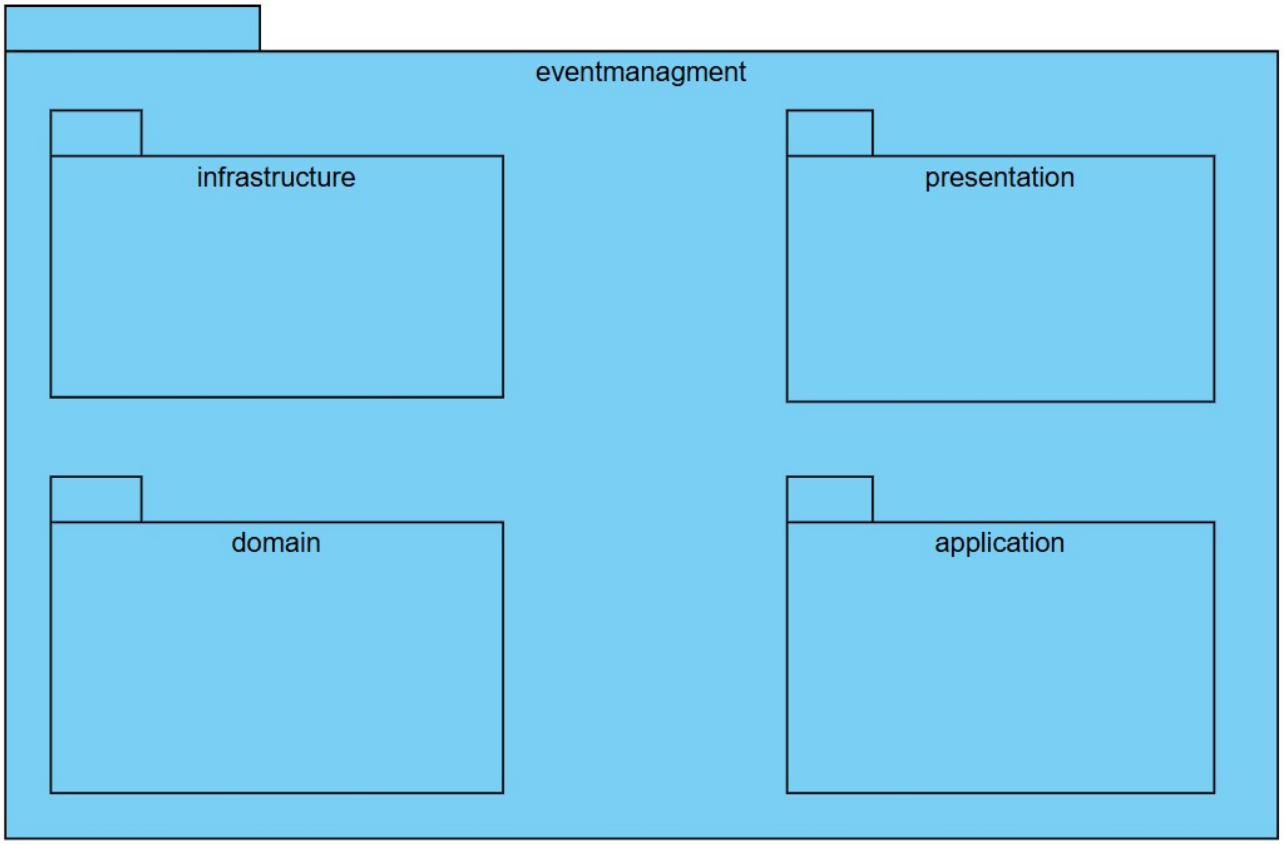
La struttura del progetto si presenta in questo modo:

- Backend
  - Src
    - Main
      - Java – contiene la logica applicativa
      - Resources – contiene le risorse esterne
      - Target – contiene i file di configurazione
    - Test – Contiene le risorse per il testing
- Frontend – contiene le risorse di frontend

Il backend sarà suddiviso in package secondo una suddivisione in sottosistemi. All'interno del package generale *unisa.esbetstart* i servizi offerti dal sistema saranno suddivisi in cinque package + uno di utilità, secondo questo schema:

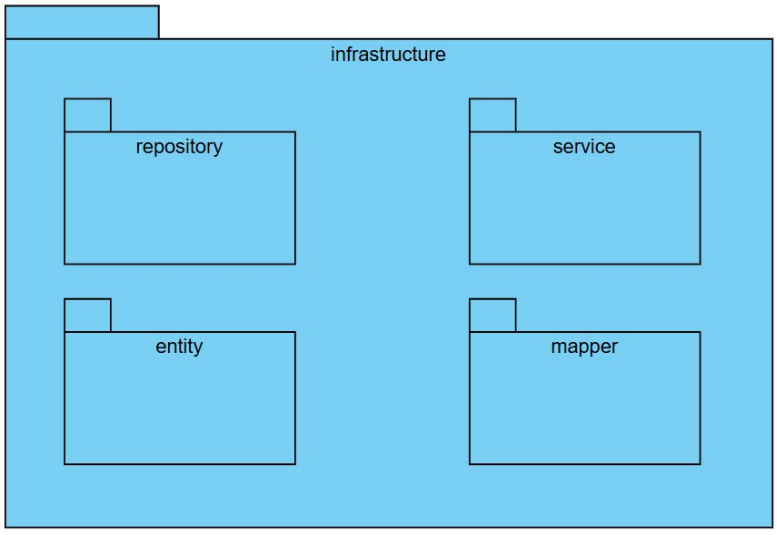


Le cartelle di “managment” saranno poi suddivise in ulteriori sottopackage, in questo modo:

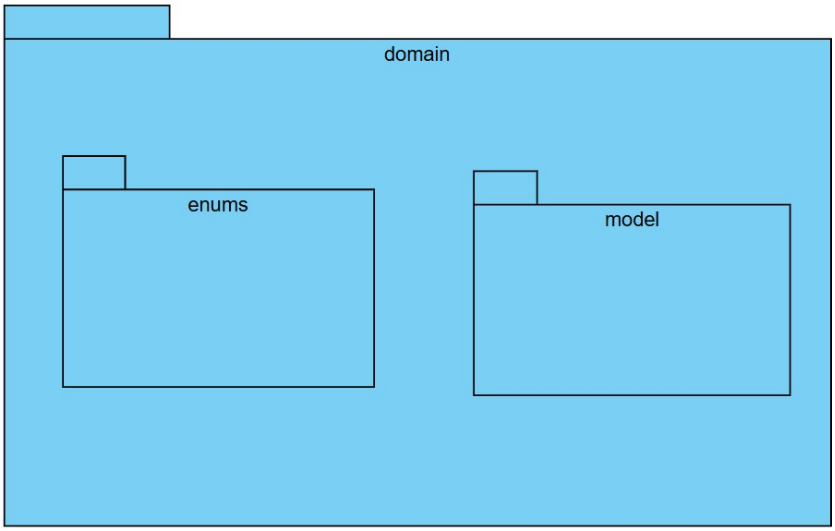


Di seguito la descrizione delle responsabilità di ogni package:

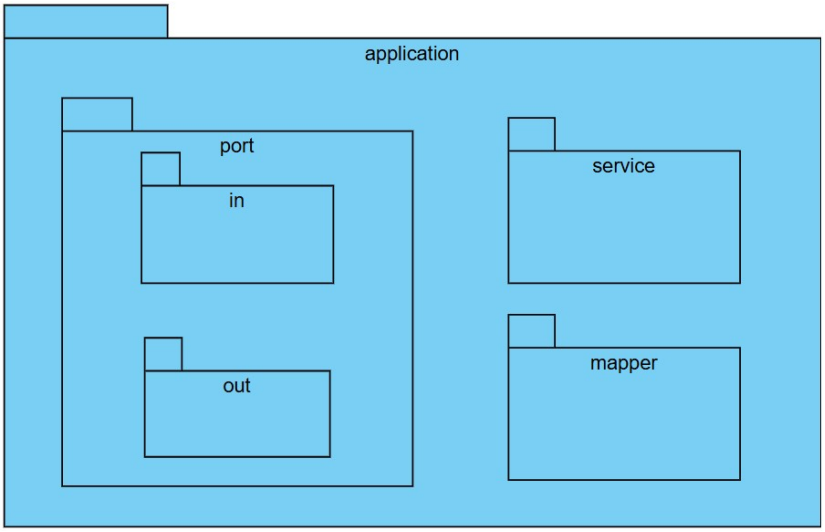
- **Infrastructure** – si occupa della gestione persistente dei dati
  - *entity*: contiente le specifiche degli oggetti entità che andranno salvati sul database
  - *repository*: contiene le specifiche jpa per comunicare con il database
  - *service*: collega entity e repository per garantire la manipolazione delle entità
  - *mapper*: contiene i metodi di mapping necessari al package



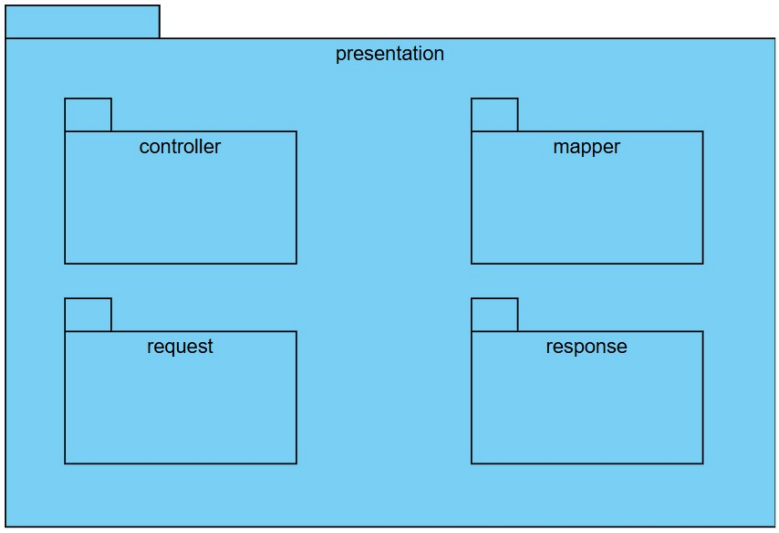
- **domain** – si occupa della rappresentazione degli oggetti di dominio e implementa la logica di business
    - *model* – contiene le classi di dominio
    - *enums* – contiene eventuali enumeratori necessari al dominio



- **application** – contiene la logica applicativa, sfrutta domain e infrastructure per implementare i servizi richiesti dal package
    - *port* – i punti di accesso per accedere ai servizi, sono interfacce da implementare
        - *in* – rappresentano i casi d'uso
        - *out* – rappresentano i punti di uscita dei servizi
    - *service* – implementa la logica dei casi d'uso
    - *mapper* – contiene i metodi di mapping necessari al package

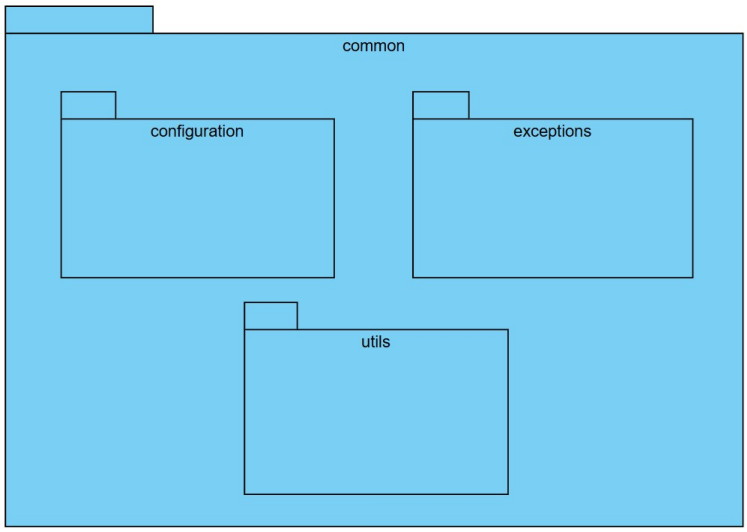


- **presentation** – è il punto di accesso al backend, tutte le richieste passano da qui
    - *controller* – contiene le classi controller, che espongono gli endpoint al frontend
    - *request* **–** contiene gli oggetti *request* per ogni servizio
    - *response* – contiene gli oggetti *response* per le richieste di GET
    - *mapper* – contiene i metodi di mapping necessari al package



La cartella *common*, infine, contiene:

- File di configurazione
- Eccezioni
- Metodi di utilità



3. Class interfaces

Di seguito riportate le interfacce per i moduli di sistema.

3.1. SlipManagment

Constraints:

context slipManagerService::placeBet(String slipId)

context EventService::placeBet(slipId: String): BetSlip

pre:
slipId <> null and not slipId.isEmpty()
and Database.betSlips->exists(s | s.id = slipId)

post:
self.result = Database.betSlips->any(s | s.id = slipId)

---

context slipManagerService::updateSlip(amount: Double, oddsIds: Set(String), slipId: String)

pre:
slipId <> null and not slipId.isEmpty()
and oddsIds->notEmpty()
and amount > 0
and Database.slips->exists(s | s.id = slipId)
and Database.odds->select(o | oddsIds->includes(o.id))->size() = oddsIds->size()
and let gambler = Database.gamblers->any(g | g.slip.id = slipId) in
  gambler.balance + gambler.bonusBalance + gambler.withdrawableBalance >= amount

post:
let slip = Database.slips->any(s | s.id = slipId) in
  slip.amount = amount
and slip.odds->size() = oddsIds->size()

and slip.odds->forAll(o | oddsIds->includes(o.id))

3.2. TransacionsManagment

TransactionsManagment
+addOffer(AddOfferRequest request) +updateOffer(UpdateOfferRequest request) +removeOffer(String offerId) +acceptOffer(AcceptOfferRequest request) +showUserTransactions(ShowUserTransactionsRequest request) :List<Transaciton> +showUserBets(ShowUserBetsRequest request) :List<BetPlaced> +showAllTransactions() :List<Transaction>

Constraints:

Context transactionManagerService::addOffer(description: String, name: String, expirationDate: DateTime, goal: Integer, type: OfferTypeEnum)

pre:
name <> null and not name.isEmpty()
and description <> null and not description.isEmpty()
and expirationDate <> null and expirationDate > now()
and goal <> null and goal > 0
and type <> null
and not Database.offers->exists(o | o.name = name)

post:
Database.offers->exists(o | o.name = name and o.description = description and o.expirationDate = expirationDate and o.goal = goal and o.type = type)

Description: Aggiunge un'offerta al database.

Context transactionManagerservice::updateOffer(description: String, name: String, expirationDate: DateTime, goal: Integer, type: OfferTypeEnum)

pre:
name <> null and not name.isEmpty()
and description <> null and not description.isEmpty()
and expirationDate <> null and expirationDate > now()
and goal <> null and goal > 0
and type <> null
and Database.offers->exists(o | o.name = name)

post:
Database.offers->exists(o | o.name = name and o.description = description and o.expirationDate = expirationDate and o.goal = goal and o.type = type)

Description: aggiorna un'offerta esistente sul database.

---

context transactionManagerservice::removeOffer(offerId: String)

pre:
offerId <> null
and not offerId.isEmpty()
and Database.offers->exists(o | o.id = offerId and o.expirationDate < now())

post:
not Database.offers->exists(o | o.id = offerId)
and Database.offers = Database.offers@pre->excluding(Database.offers@pre->select(o | o.id = offerId)->first())

Description: rimuove un'offerta dal database.

---

context transactionManagerservice::acceptOffer(gamblerId: String, offerId: String)

pre:
and not self.activatedOffers->exists(a | a.offer.id = offerId)

post:
self.activatedOffers->exists(a | a.offer.id = offerId and a.progress = 0)

Description: aggiunge un'offerta alla liste di offerte attive di un gambler.

---

context transactionManagerservice::showUserTransaction(gamblerId: String, type: TransactionTypeEnum): Sequence(Transaction)

pre:
gamblerId <> null
and not gamblerId.isEmpty()
and type <> null
and Database.gamblers->exists(g | g.id = gamblerId)

post:
self.result = Database.transactions->select(t | t.gambler.id = gamblerId and t.type = type)
and self.result->forAll(t | t.gambler.id = gamblerId and t.type = type)
and self.result->size() = Database.transactions->select(t | t.gambler.id = gamblerId and t.type = type)->size()

Description: restituisce tutte le transizioni effettuate da un gambler di un determinato tipo

---

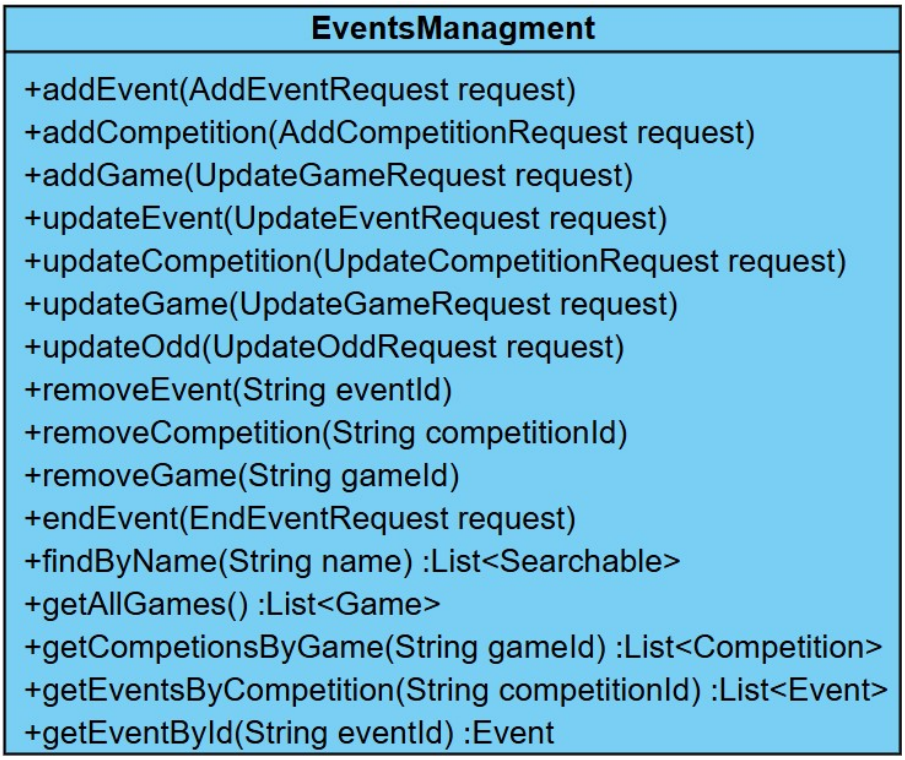
context transactionManagerService::showUserBets(gamblerEmail: String, pending: Boolean): Sequence(BetPlaced)

pre:
gamblerEmail <> null and not gamblerEmail.isEmpty()
and Database.gamblers->exists(g | g.email = gamblerEmail)

post:
let gambler = Database.gamblers->any(g | g.email = gamblerEmail) in
  if pending then
    self.result = gambler.betPlaced->select(b | b.status = StatusEnum.PENDING)
  else
    self.result = gambler.betPlaced
  endif

Description: restituisce tutte le scommesse di uno scommettitore se l' attributo "pending" è impostato a *false*, altrimenti solo quelle in corso.

3.3. EventsManagment



Constraints:

context EventManagerService::addEvent(competitionId: String, name: String, date: String, odds: List(AddOddRequest)): Event

pre:
competitionId <> null and not competitionId.isEmpty()
and name <> null and not name.isEmpty()
and date <> null and not date.isEmpty() and date < LocalDateTime::now()
and odds->notEmpty()
and Database.competitions->exists(c | c.id = competitionId)
and let competition = Database.competitions->any(c | c.id = competitionId) in
  let game = competition.game in
    game.rules->size() = odds->size()
    and odds->forAll(o | game.rules->exists(r | r.name = o.name))

post:
let event = Event.allInstances()->any(e | e.competition.id = competitionId and e.name = name and e.date = date) in
  self.result = event
and event <> null
and event.odds->size() = odds->size()
and event.odds->forAll(o | odds->exists(ao | ao.name = o.name and ao.value = o.value and ao.position = o.position))

Description: aggiunge un nuovo enento a una competizione

---

context EventsManagerService::addCompetition(gameId: String, name: String, originCountry: String): Competition

pre:
gameId <> null and not gameId.isEmpty()
and name <> null and not name.isEmpty()
and originCountry <> null and not originCountry.isEmpty()
and Database.games->exists(g | g.id = gameId)
and Database.competitions->forAll(c | c.name <> name)

post:
let competition = Competition.allInstances()->any(c | c.game.id = gameId and c.name = name and c.originCountry = originCountry) in
  self.result = competition
and competition <> null
and competition.game.id = gameId

Description: aggiunge una competizione a un gioco

---

context EventsManagerService::addGame(name: String, rules: List(AddRuleRequest)): Game

pre:
name <> null and not name.isEmpty()
and rules->notEmpty()
and rules->forAll(r | r.name <> null and not r.name.isEmpty() and r.position <> null and r.position > 0)
and Database.games->forAll(g | g.name <> name)

post:
let game = Game.allInstances()->any(g | g.name = name) in
  self.result = game
and game <> null
and game.rules->size() = rules->size()
and game.rules->forAll(r | rules->exists(ar | ar.name = r.name and ar.position = r.position))

Description: aggiunge un nuovo gioco

---

context EventsManagerService::updateEvent(eventId: String, name: String, date: LocalDateTime)

pre:
eventId <> null and not eventId.isEmpty()
and name <> null and not name.isEmpty()
and date <> null and date > LocalDateTime::now()
and Database.events->exists(e | e.id = eventId)

post:
let event = Database.events->any(e | e.id = eventId) in
  event.name = name
  and event.date = date

Description: aggiorna un evento.

---

context EventsManagerService::updateCompetition(competitionId: String, name: String, originCountry: String)

pre:
competitionId <> null and not competitionId.isEmpty()
and name <> null and not name.isEmpty()
and originCountry <> null and not originCountry.isEmpty()
and Database.competitions->exists(c | c.id = competitionId)

post:
let competition = Database.competitions->any(c | c.id = competitionId) in
  competition.name = name

```
    and competition.originCountry = originCountry
```

Description: aggiorna una competizione.

```
context EventsManagerService::updateGame(gameId: String, name: String, rules:
List(AddRuleRequest)): Game

pre:
gameId <> null and not gameId.isEmpty()
and name <> null and not name.isEmpty()
and rules->notEmpty()
and rules->forAll(r | r.name <> null and not r.name.isEmpty() and r.position <> null and r.position >
0)
and Database.games->exists(g | g.id = gameId)

post:
let game = Database.games->any(g | g.id = gameId) in
    game.name = name
    and game.rules->size() = rules->size()
    and game.rules->forAll(r | rules->exists(ar | ar.name = r.name and ar.position = r.position))
```

Description: aggiorna un gioco.

```
context EventsManagerService::updateOdd(oddId: String, oddValue: Float)

pre:
oddId <> null and not oddId.isEmpty()
and oddValue <> null and oddValue > 0
and Database.odds->exists(o | o.id = oddId)

post:
let odd = Database.odds->any(o | o.id = oddId) in
    odd.value = oddValue
```

Description: aggiorna il valore di un odd

```
context EventsManagerService::removeEvent(eventId: String)

pre:
eventId <> null and not eventId.isEmpty()
and Database.events->exists(e | e.id = eventId)
and let event = Database.events->any(e | e.id = eventId) in
    event.isEnded = true

post:
```

```
Database.events->forAll(e | e.id <> eventId)
and Database.events->select(e | e.id = eventId)->size() = 0
```

Description: Rimuove un evento solo se è terminato.

```
context EventsManagerService::removeCompetition(competitionId: String)

pre:
competitionId <> null and not competitionId.isEmpty()
and Database.competitions->exists(c | c.id = competitionId)
and let competition = Database.competitions->any(c | c.id = competitionId) in
    competition.events->forAll(e | e.isEnded = true)

post:
Database.competitions->forAll(c | c.id <> competitionId)
and Database.competitions->select(c | c.id = competitionId)->size() = 0
```

Description: Rimuove una competizione solo se tutti gli eventi relativi sono terminati.

```
context EventsManagerService::removeGame(id: String)

pre:
id <> null and not id.isEmpty()
and Database.games->exists(g | g.id = id)
and Database.competitions->forAll(c | c.events->forAll(e | e.isEnded = true))

post:
Database.games->forAll(g | g.id <> id)
and Database.games->select(g | g.id = id)->size() = 0
```

Description: Rimuove un gioco solo se tutti gli eventi delle competizioni correlate sono terminati.

```
context EventsManagerService::endEvent(eventId: String, winningOdds: List(String))

pre:
eventId <> null and not eventId.isEmpty()
and winningOdds->notEmpty()
and Database.events->exists(e | e.id = eventId)
and let event = Database.events->any(e | e.id = eventId) in
    event.odds->forAll(o | winningOdds->includes(o.id) implies o.isWon = true)
    and event.odds->forAll(o | not(winningOdds->includes(o.id)) implies o.isWon = false)

post:
let event = Database.events->any(e | e.id = eventId) in
    event.odds->forAll(o | o.evaluate(o.isWon))
```

and Database.events->forAll(e | e.id <> eventId)
and Database.events->select(e | e.id = eventId)->size() = 0

Description: Termina un evento, valuta gli odd e elimina l'evento.

---

context EventsManagerService::findByName(name: String): Sequence(Searchable)

pre:
name <> null and not name.isEmpty()

post:
let searchResults = Database.searchables->select(s | s.name.isSubstringOf(name)) in
  self.result = searchResults
and searchResults->forAll(s | s.name.isSubstringOf(name))
and self.result->size() = searchResults->size()

Description: restituisce tutti i Searchable (eventi, competizioni e giochi) simili alla stringa di ricerca.

---

context EventManagerService::getAllGames(): Sequence(Game)

pre:
true

post:
self.result = Database.games
and self.result->size() = Database.games->size

Description: restituisce tutti i giochi presenti nel database.

---

context EventsManagerService::getCompetitionsByGame(gameid: String): Sequence(Competition)

pre:
gameid <> null and not gameid.isEmpty()

post:
let competitionsForGame = Database.competitions->select(c | c.game.id = gameid) in
  self.result = competitionsForGame
and competitionsForGame->forAll(c | c.game.id = gameid)
and self.result->size() = competitionsForGame->size()

Description: restituisce tutte le competizioni relative a un gioco.

---

context EventsManagerService::getEventsByCompetition(competitionId: String): Sequence(Event)

pre:
competitionId <> null and not competitionId.isEmpty()

post:
let eventsForCompetition = Database.events->select(e | e.competition.id =competitionId) in
  self.result = eventsForCompetition
and eventsForCompetition->forAll(e | e.competition.id = competitionId)
and self.result->size() = eventsForCompetition->size()

Description: restituisce tutti gli eventi relativi a una competizione.

---

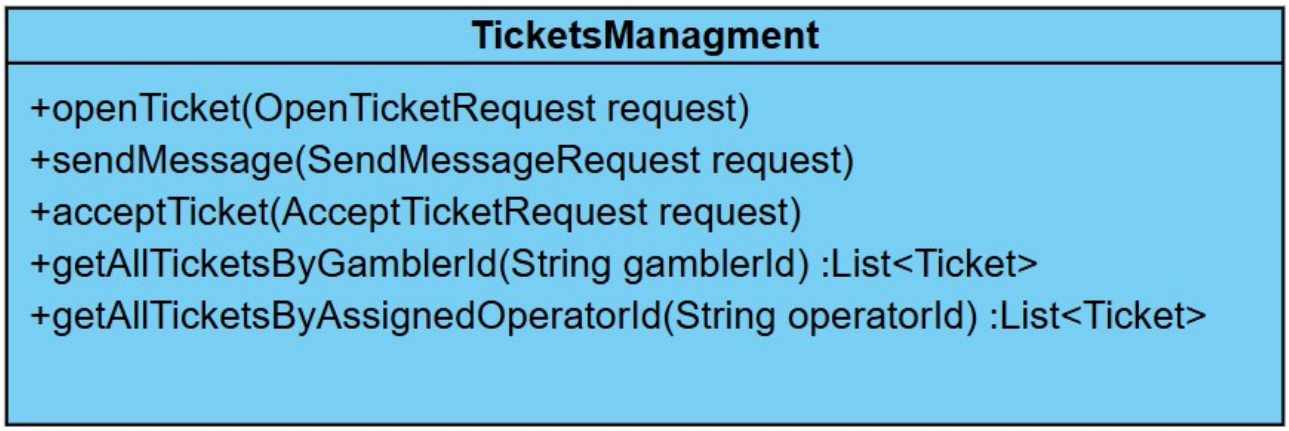
context EventsManagerService::getEventById(eventId: String): Event

pre:
eventId <> null and not eventId.isEmpty()

post:
let event = Database.events->any(e | e.id = eventId) in
  self.result = event
and event <> null

Description: restituisce l'evento corrispondente all'id richiesto.

3.4. TicketsManagment



Constraints:

context TicketsManagerService::openTicket(gamblerEmail: String, category: String, messageText: String)

pre:
gamblerEmail <> null and not gamblerEmail.isEmpty()
and category <> null and not category.isEmpty()
and messageText <> null and not messageText.isEmpty()
and Database.gamblers->exists(g | g.email = gamblerEmail)

post:
let ticket = Ticket.allInstances()->any(t | t.gambler.email = gamblerEmail and t.category = category
and t.messageText = messageText) in
  self.result = ticket
and ticket <> null

Description: Apre un ticket di supporto per un giocatore con i dettagli forniti.

---

context TicketsManagerService::sendMessage(ticketId: String, text: String, sender: String)

pre:
ticketId <> null and not ticketId.isEmpty()
and text <> null and not text.isEmpty()
and sender <> null and not sender.isEmpty()
and Database.tickets->exists(t | t.id = ticketId)

post:
let ticket = Database.tickets->any(t | t.id = ticketId) in
  ticket.messages->forAll(m | m.text <> text)
and ticket.messages->size() = ticket.messages->size() + 1
and ticket.messages->last().text = text
and ticket.messages->last().sender = sender

Description: Invia un messaggio al ticket specificato.

---

context TicketsManagerService::acceptTicket(ticketId: String, messageText: String,
assignedOperator: String)

pre:
ticketId <> null and not ticketId.isEmpty()
and messageText <> null and not messageText.isEmpty()
and assignedOperator <> null and not assignedOperator.isEmpty()
and Database.tickets->exists(t | t.id = ticketId)

post:
let ticket = Database.tickets->any(t | t.id = ticketId) in
  ticket.status = 'Accepted'
and ticket.assignedOperator = assignedOperator
and ticket.messages->forAll(m | m.text <> messageText)
and ticket.messages->size() = ticket.messages->size() + 1

and ticket.messages->last().text = messageText
and ticket.messages->last().sender = assignedOperator

Description: Accetta un ticket assegnandogli un operatore e inviando un messaggio di conferma.

---

context TicketsManagerService::getTicketsByGamblerEmail(gamblerEmail: String)

pre:
gamblerEmail <> null and not gamblerEmail.isEmpty()
and Database.gamblers->exists(g | g.email = gamblerEmail)

post:
self.result = Database.tickets->select(t | t.gambler.email = gamblerEmail)

Description: Recupera tutti i ticket associati a un giocatore tramite il suo indirizzo email.

---

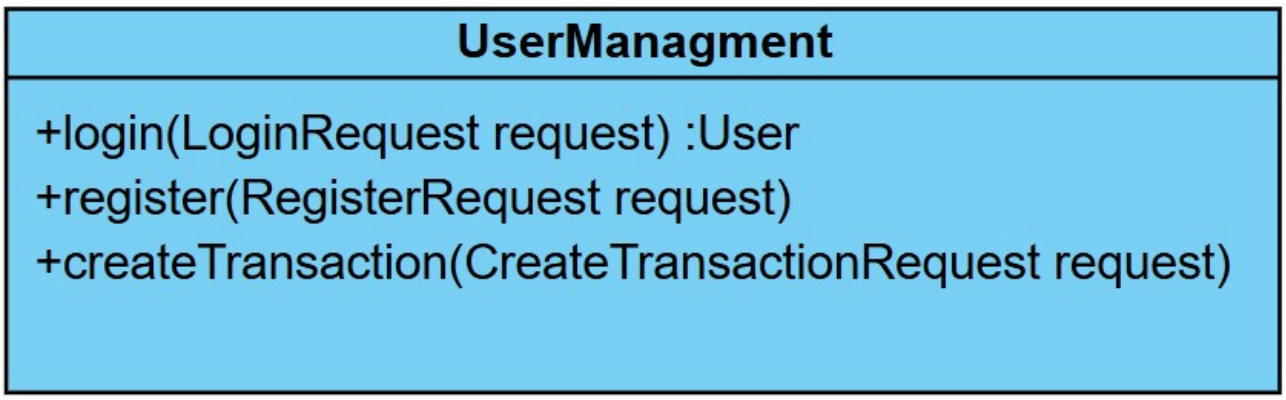
context TicketService::getAllTicketsByAssignedOperatorId(operatorId: String) : List(Ticket)

pre:
and Database.operators->exists(o | o.id = operatorId)

post:
result = Database.tickets->select(t | t.assigned_operator = operatorId)

Description: Recupera tutti i ticket associati a un operatore tramite il suo indirizzo email.

3.5. UserManagment



Constraints:

context UsersManagerService::login(email: String, password: String) : User

pre:
email <> null
and not email.isEmpty()
and email.matches('[^@\\s]+@[^@\\s]+\\.[^@\\s]+') -- Formato valido di email
and password <> null
and not password.isEmpty()
and Database.users->exists(u | u.email = email and u.password = password)

post:
result = Database.users->select(u | u.email = email and u.password = password)->first()

Description: effettua il login di un account esistente.

---

context UsersManagerService::register(name: String, surname: String, email: String, username: String, password: String) : User

pre:
name <> null
and not name.isEmpty()
and surname <> null
and not surname.isEmpty()
and email <> null
and not email.isEmpty()
and email.matches('[^@\\s]+@[^@\\s]+\\.[^@\\s]+') -- Formato valido di email
and username <> null
and not username.isEmpty()
and password <> null
and not password.isEmpty()
and Database.users->exists(u | u.email = email)->not() -- La email non deve essere già registrata
and Database.users->exists(u | u.username = username)->not() -- Il nome utente non deve essere già registrato

post:
result.email = email
and result.username = username
and result.password = password
and result.name = name
and result.surname = surname
and Database.users->includes(result)

Description: registra un nuovo utente nel sistema.

---

context UsersManagerService::createTransaction(gamblerId: String, transactionType:

TransactionTypeEnum, transactionValue: double)

pre:
and transactionValue > 0
and Database.gamblers->exists(g | g.id = gamblerId)
and (transactionType = TransactionTypeEnum::DEPOSIT or
    (transactionType = TransactionTypeEnum::WITHDRAWAL and self.balance >=
transactionValue))

post:
self.balance =
   if transactionType = TransactionTypeEnum::DEPOSIT then self.balance + transactionValue
   else if transactionType = TransactionTypeEnum::WITHDRAWAL then self.balance -
transactionValue
   else self.balance
and Database.transactions->exists(t | t.gambler.id = gamblerId and t.transactionType =
transactionType and t.transactionValue = transactionValue)

Description: registra una nuova transazione e modifica il bilancio di uno scommettitore di conseguenza.

---

4. Design Patterns

4.1. Pattern dei servizi

All'interno dell'applicazione viene utilizzato un design pattern di tipo *Facade*, che consiste nella sola esposizione dell'interfaccia di un servizio nascondendo la logica che si trova al di sotto, nel nostro caso saranno le porte (in e out) a fornire quest'astrazione.
Ogni sottosistema esporrà tramite le porte tutti i servizi che offre, implementati in classi Java chiamate "ManagerService".
Per comodità per ogni oggetto di dominio saranno individuati 4 managerServices, corrispondenti alle operazioni CRUD:

- CreateManagerService
- ReadManagerService
- UpdateManagerService
- DeleteManagerService

Occasionalmente i nomi saranno modificati per meglio riflettere lo scopo dei servizi.

4.2. Pattern dei dati persistenti

Per salvare i dati in maniera persistente verrà utilizzato il pattern progettuale *DAO* (Data Access Object), che permette di comunicare con un database tramite interfacce, senza curarsi delle operazioni a basso livello (e.g. la scrittura di query).
Nell'applicazione sarà utilizzata Spring JPA, un servizio di persistenza offerto da Spring in Java che semplifica molto il processo di persistenza.