Università degli Studi di Salerno
Corso di Ingegneria del Software

Esbet START
Object Design Document
Versione 1.0

Data: 16/12/2024

Progetto: EsbetSTART	Versione: 1.0
Documento: Object Design Document	Data: 16/12/2024

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	Descrizio ne	Autore
16/12/2024	1.0	Prima stesura ODD	

1. Introduzione

1.1. Object design trade-offs

1.2. Interface documentation guidelines

1.3. Definitions, acronyms, and abbreviations

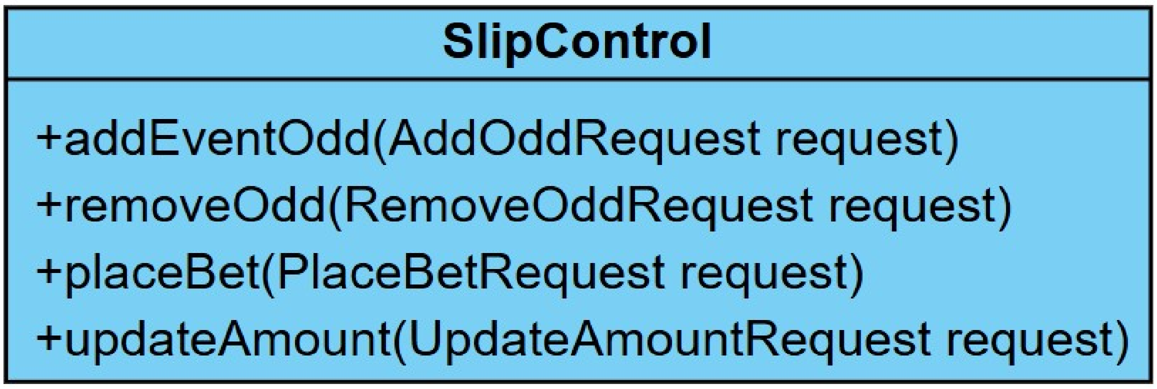
1.4. References

2. Packages

3. Class interfaces

Di seguito riportate le interfacce per i moduli di sistema.

3.1. SlipManagment


SlipControl
+addEventOdd(AddOddRequest request)
+removeOdd(RemoveOddRequest request)
+placeBet(PlaceBetRequest request)
+updateAmount(UpdateAmountRequest request)

Constraints:

context Slip::addEventOdd(oddId: String)

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

post:
self.odds->includes(Database.odds->select(o | o.id = oddId)->first())

---

context Slip::removeOdd(oddId: String)

pre:
and Database.odds->exists(o | o.id = oddId)
and self.odds->exists(o | o.id = oddId)

post:
not self.odds->exists(o | o.id = oddId) -- L'odd deve essere rimosso dalla lista degli odds dello slip

---

context Slip::placeBet(slipId: String, competition: String, game: String)

pre:
self.amount > 0
and self.amount <= self.gambler.balance
and competition <> null
and not competition.isEmpty()
and game <> null
and not game.isEmpty()
and Database.odds->forAll(o | self.odds->includes(o))

post:
let betPlaced : BetPlaced = new BetPlaced(self.amount, 'playing') in
self.gambler.balance = self.gambler.balance - betPlaced.amount
and Database.betPlaced->includes(betPlaced)
and betPlaced.staticOdds->size() = self.odds->size()
and betPlaced.staticOdds->forAll(o | o.date = now() and o.result = 'playing' and o.name = o.name and o.value = o.value)
and betPlaced.odds->forAll(o | o.competition = competition and o.game = game)

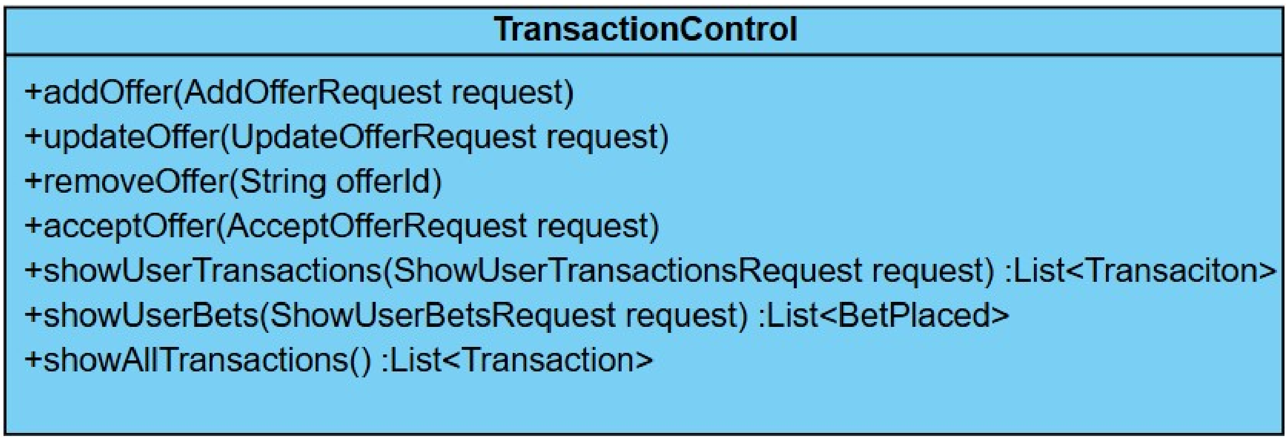
---

context Slip::updateAmount(request: UpdateAmountRequest)

pre:
request.slipId <> null
and not request.slipId.isEmpty()
and request.amount > 0
and Database.slips->exists(s | s.id = request.slipId)
and self.id = request.slipId

post:
self.amount = request.amount

3.2. ***TransacionsManagment***


TransactionControl
+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 OffersService::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)

---

context OffersService::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)

---

context OffersService::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())

---

context Gambler::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)

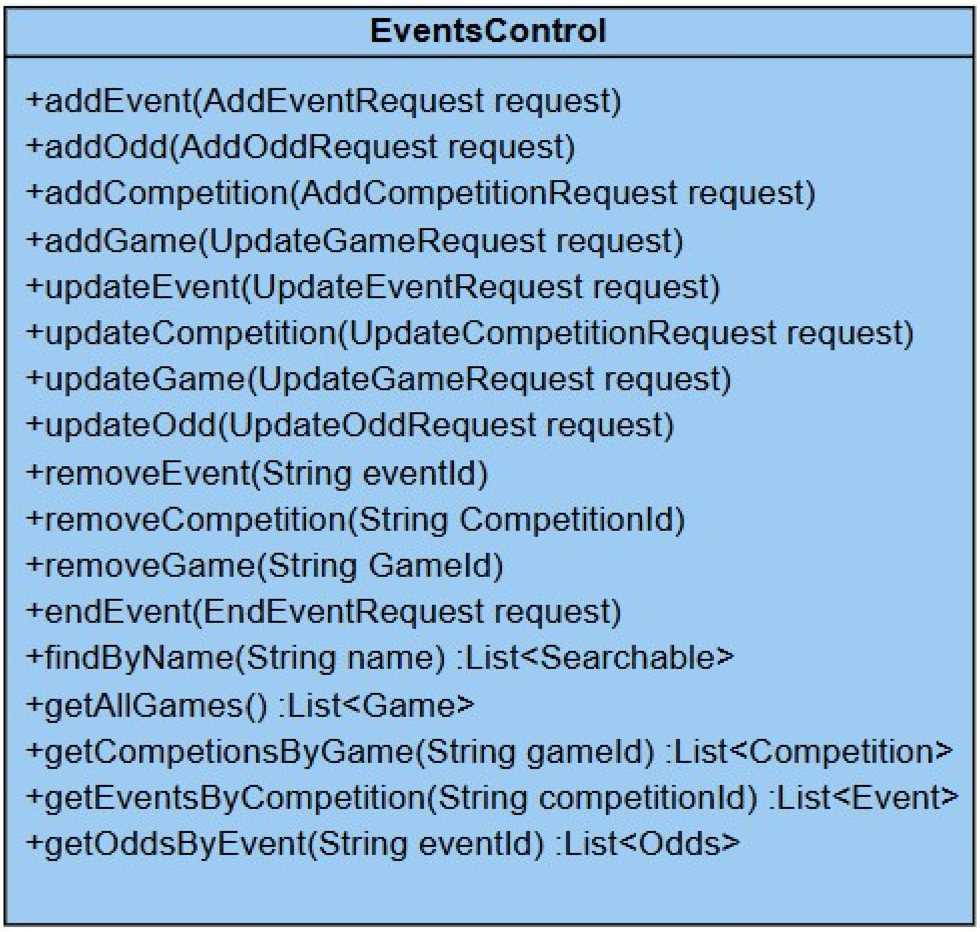
---

context OffersService::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()

3.3. EventsManagment


EventsControl
+addEvent(AddEventRequest request)
+addOdd(AddOddRequest request)
+addCompetition(AddCompetitionRequest request)
+addGame(UpdateGameRequest request)
+updateEvent(UpdateEventRequest request)
+updateCompetition(UpdateCompetitionRequest request)
+updateGame(UpdateGameRequest request)
+updateOdd(UpdateOddRequest request)
+removeEvent(String eventId)
+removeCompetition(String CompetitionId)
+removeGame(String GameId)
+endEvent(EndEventRequest request)
+findByName(String name) :List<Searchable>
+getAllGames() :List<Game>
+getCompetionsByGame(String gameId) :List<Competition>
+getEventsByCompetition(String competitionId) :List<Event>
+getOddsByEvent(String eventId) :List<Odds>

Constraints:

context Competition::addEvent(eventName: String, eventDate: Date)

pre:
not self.events->exists(e | e.name = eventName and e.date = eventDate)
and not name.isEmpty()
and not name = null
and not eventDate=null
and eventDate > now()

post:
self.events->exists(e | e.name = eventName and e.date = eventDate) and self.events->size() =
self.events@pre->size() + 1
result = new Event(name, date)

---

context Event::addOdd(name: String, value: Float)

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

and value <> null
and value > 0
and not self.odds->exists(o | o.name = name)

post:
self.odds->exists(o | o.name = name and o.value = value)

---

context Event::updateOdd(oddName: String, oddValue: Float)

pre:
oddName <> null
and not oddName.isEmpty()
and oddValue <> null
and oddValue > 0
and self.odds->exists(o | o.name = oddName)

post:
self.odds->exists(o | o.name = oddName and o.value = oddValue)

---

context EventService::addEvent(competitionId: String, name: String, date: LocalDateTime, odds: Sequence(AddOddRequest))

pre:
competitionId <> null
and not competitionId.isEmpty()
and odds <> null
and odds->size() = Database.competitions->select(c | c.id = competitionId)->first().game.getRules()->size()
and odds->forAll(o | Database.competitions->select(c | c.id = competitionId)->first().game.getRules()->exists(r | r.name = o.name))

post:
Database.events->exists(e | e.name = name and e.date = date and e.competition.id = competitionId)
and e.odds->size() = odds->size()
and odds->forAll(o | e.odds->exists(odd | odd.name = o.name and odd.value = o.value))

---

context Game::addCompetition(name: String, originCountry: String)

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

post:

self.competitions->exists(c | c.name = name and c.originCountry = originCountry) and
self.competitions->size() = self.competitions@pre->size() + 1

---

context EventsService::addGame(name: String, rules: Sequence(String))

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

post:
Database.games->exists(g | g.name = name and g.rules = rules)

---

context Competition::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 > now()
and self.events->exists(e | e.id = eventId)

post:
self.events->exists(e | e.id = eventId and e.name = name and e.date = date)

---

context Game::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 self.competitions->exists(c | c.id = competitionId)

post:
self.competitions->exists(c | c.id = competitionId and c.name = name and c.originCountry =
originCountry)

---

context EventsService::updateGame(gameId: String, name: String, rules: Sequence(String))

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

post:
Database.games->exists(g | g.id = gameId and g.name = name and g.rules = rules)

---

context Competition::removeEvent(id: String)

pre:
id <> null
and not id.isEmpty()
and self.events->exists(e | e.id = id)

post:
not self.events->exists(e | e.id = id)
and self.events = self.events@pre->excluding(self.events@pre->select(e | e.id = id)->first())

---

context Game::removeCompetition(id: String)

pre:
id <> null
and not id.isEmpty()
and self.competitions->exists(c | c.id = id)

post:
not self.competitions->exists(c | c.id = id)
and self.competitions = self.competitions@pre->excluding(self.competitions@pre->select(c | c.id = id)->first())

---

context EventsService::removeGame(id: String)

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

post:

not Database.games->exists(g | g.id = id)
and Database.games = Database.games@pre->excluding(Database.games@pre->select(g | g.id = id)->first())

---

context EventService::endEvent(eventId: String, oddResults: Sequence(OddResultRequest))

pre: eventId <> null
and not eventId.isEmpty()
and Database.events->exists(e | e.id = eventId)
and let event = Database.events->select(e | e.id = eventId)->first() in oddResults->forAll(or | event.odds->exists(o | o.name = or.oddId))
and event.odds->forAll(o | oddResults->exists(or | or.oddId = o.name))

---

context Event::endOdd(oddId: String, isWon: Boolean)

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

post:
let targetOdd = self.odds->select(o | o.id = oddId)->first() in
targetOdd.oddStatic->forAll(os |
    os.state = if isWon then OddStaticState::Won else OddStaticState::Lost endif)

---

context EventService::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()

---

context EventService::getAllGames(): Sequence(Game)

pre:
true

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

---

context EventService::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()

---

context EventService::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()

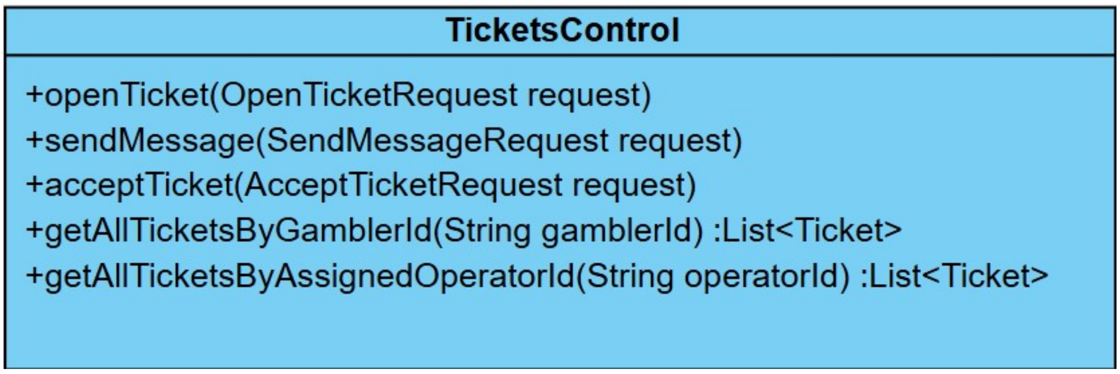
---

context EventService::getOddsByEvent(eventId: String): Sequence(Odd)

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

post:
let oddsForEvent = Database.odds->select(o | o.event.id = eventId) in
    self.result = oddsForEvent
and oddsForEvent->forAll(o | o.event.id = eventId)
and self.result->size() = oddsForEvent->size()

3.4. ***TicketsManagment***



TicketsControl

+openTicket(OpenTicketRequest request)
+sendMessage(SendMessageRequest request)
+acceptTicket(AcceptTicketRequest request)
+getAllTicketsByGamblerId(String gamblerId) :List<Ticket>
+getAllTicketsByAssignedOperatorId(String operatorId) :List<Ticket>

Constraints:

context Gambler::openTicket(gamblerId: String, category: String, messageText: String)

pre:
gamblerId <> null
and not gamblerId.isEmpty()
and category <> null
and not category.isEmpty()
and messageText <> null
and messageDate <> null
and messageDate <= now()
and not messageText.isEmpty()
and messageText.size() <= 300

post:

```
self.tickets->exists(t |
   t.category = category
   and t.assigned_operator = null
   and t.status = 'PENDING'
   and t.messages->size() = 1
   and t.messages->first().text = messageText
   and t.messages->first().date = now()
   and t.messages->first().sender = 'CLIENT'
   and t.messages->first().status = 'SENT'
)
```

---

context Ticket::sendMessage(text: String, sender: RolesEnum)

pre:
text <> null
and not text.isEmpty()
and text.size() <= 300
and sender <> null

post:

```
self.messages->exists(m |
   m.text = text
   and m.date = now()
   and m.sender = sender.toString()
   and m.status = 'SENT'
)
and self.messages->size() = self.messages@pre->size() + 1
```

---

context Ticket::acceptTicket(messageText: String)

pre:

```
messageText <> null
and not messageText.isEmpty()
and messageText.size() <= 300
and self.assigned_operator = null
and self.status = 'PENDING'
```

post:

```
self.assigned_operator <> null
and self.status = 'OPENED’
and self.messages->exists(m |
   m.text = messageText
   and m.date = now()
   and m.sender = 'OPERATOR'
   and m.status = 'SENT'
)
and self.messages->size() = self.messages@pre->size() + 1
```

---

context TicketService::getAllTicketsByGamblerId(gamblerId: String) : List(Ticket)

pre:

```
and Database.gamblers->exists(g | g.id = gamblerId)
```

post:

```
result = Database.tickets->select(t | t.gambler.id = gamblerId)
```

---

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

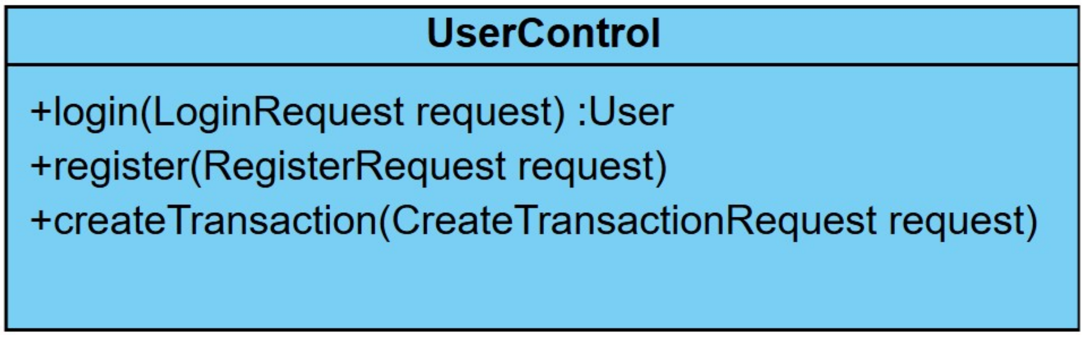
pre:

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

post:

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

***3.5.* UserManagment**



```
UserControl
+login(LoginRequest request) :User
+register(RegisterRequest request)
+createTransaction(CreateTransactionRequest request)
```

Constraints:

```
context UserService::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()
```

```
context UserService::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)

---

context Gambler::createTransaction(gamblerId: String, transactionType: TransactionTypeEnum, transactionValue: Integer)

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)
