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

# BeVoyager
# ODD
# Versione 1.4

BeVoyager

**Data: 11/01/2017**

**Partecipanti:**

| Nome | Matricola |
|---|---|
| Donato Tiano | 0512102916 |
| Alessandro Longobardi | 0512102910 |
| Paolo Zirpoli | 0512102862 |
| Salvatore Ruggiero | 0512103002 |

# Revision History

| Data | Versione | Descrizione | Autore |
|---|---|---|---|
| 09/01/2017 | 1.0 | Introduzione, object design trade-offs. | Paolo Zirpoli, Alessandro Longobardi, Donato Tiano |
| 10/01/2017 | 1.1 | Prima stesura packages. | Salvatore Ruggiero |
| 10/01/2017 | 1.2 | Linee guida per la documentazione dell'interfaccia. | Salvatore Ruggiero |
| 11/01/2017 | 1.3 | Modifiche linee guida per la documentazione dell'interfaccia. | Salvatore Ruggiero |
| 11/01/2017 | 1.4 | Design pattern. | Paolo Zirpoli |
| 11/01/2017 | 1.5 | Definizione interfacce in OCL | Alessandro Longobardi |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

## Sommario

# 1. Introduzione

L'Object Design Document (ODD) definisce l'object level design del sistema che si sta sviluppando. Attraverso questo documento viene definita l'architettura modulare della piattaforma, la suddivisione del suo contenuto in packages differenti. Esso sfrutta le conoscenze acquisite tramite la stesura dei precedenti documenti.

## 1.1 Object design trade-offs

- ***Tempo di risposta vs Spazio di memoria:***

Basandoci sui design goals descritti in precedenza (SDD sez. 1), il tempo risposta deve essere minimo. Le operazioni che usano più tempo all'interno del sistema sono gli accessi al database, in quanto, oltre all'accesso al disco bisogna effettuare operazioni di unione e controllo sulle tabelle generate dalle query. Dato che le operazioni effettuate dal sistema spesso risultano nella creazione delle stesse tabelle con gli stessi dati al loro interno, abbiamo rilevato che generare le tabelle necessarie in precedenza, in modo tale che quando viene effettuata un'operazione che ha bisogno di tali dati, non si ha la necessità di generare la tabella a runtime, ma semplicemente si effettua una ricerca in una già generata. Questo porta ad un utilizzo di memoria maggiore all'interno del database ma velocizza di molto le operazioni effettuate.

- ***Comprensibilità vs Costi***

All'interno del team di sviluppo utilizzeremo uno stile di programmazione ben definito tra tutti i componenti del team. In questo modo, nel caso di aggiunta di nuovi membri allo sviluppo, il tempo di training dei nuovi elementi sarà minimizzato, facendo in modo che possano subito iniziare a svolgere task di sviluppo, minimizzando i costi relativi al training, e velocizzando il tempo di sviluppo.

- ***Costi vs Mantenimento***

Il sistema verrà implementato con particolare attenzione alla mantenibilità dello stesso. Gli interventi di manutenzione devono essere resi semplici e veloci per minimizzare il downtime del sistema, e di conseguenza la perdita di guadagno relativa alla mancata disponibilità del servizio. Questo vuol dire che bisognerà prestare più attenzione all'implementazione del codice, e mantenere una documentazione adeguata alla complessità del sistema, il che prevede costi maggiori in quanto ci sarà bisogno di un team dedicato alla documentazione stessa.

- ***Interfaccia vs Easy-use***

L'interfaccia del sistema sarà costruita in modo tale da risultare semplice all'utilizzo dell'utente. Aumentare la facilità di utilizzo significa allargare le funzionalità della piattaforma ad un maggior numero di utenti. Per questo il team di sviluppo, nel momento di implementare l'interfaccia grafica, non solo cercherà di dare alla pagina uno style piacevole alla vista ma fornire una struttura di interazione tale da rendere semplice all'utente la gestione delle sue operazioni.

## 1.2 Linee guida per la documentazione dell'interfaccia

| Tipo | Regole sui nomi | Esempi |
|---|---|---|
| **Packages** | Il prefisso di un nome di un pacchetto comincia sempre con una lettera maiuscola. Se il nome del package comprende due o più parole, allora il nome completo prevederà che la lettera iniziale della parola nel mezzo sarà maiuscola. | package AccessController; |
| **Classi** | I nomi delle classi iniziano sempre con una lettera minuscola. Esse devono racchiudere in breve il campo in cui operano. Se il nome è composto da più parole, allora le parole nel mezzo avranno la prima lettera maiuscola e le parole risulteranno divise da dei trattini. | class creaLuogo;<br>class search-route-result; |
| **Interfacce** | I nomi dell'interfaccia sono scelti in modo tale da racchiudere il significato di ciò che la stessa mostra all'utente e cosa permette di fare. | interface search;<br>interface feedback; |
| **Metodi** | I nostri metodi rappresentano la funzionalità ad essi associata. Se composti da molteplici parole, la prima lettera della prima parola sarà scritta in minuscolo, mentre la prima lettera di una parola interna sarà maiuscola. | deleteRoute(int id); |
| **Variabili** | I nomi associati alle variabili sono scritti totalmente con caratteri minuscoli. Essi sono scelti in modo tale da rendere | Location location; |

| | | |
|---|---|---|
| | semplice il loro significato ed esplicare significativamente il loro utilizzo. | |

## 1.3 Design Pattern

Di seguito sono elencati i design pattern selezionati per essere implementati all'interno della piattaforma BeVoyager. I design pattern sono stati scelti analizzando i requisiti richiesti dalla piattaforma, selezionando i pattern che presentano vantaggi che soddisfano tali requisiti, permetteranno di semplificare l'implementazione della piattaforma stessa.

**Bridge Pattern:** Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per disaccoppiare dei componenti software. Il pattern in questione si presta particolarmente al sistema proposto in quanto è stato progettato per garantire il disaccoppiamento tra le varie classi, nascondendo l'implementazione delle stesse, permettendo ad una classe client (che utilizza i servizi un'altra classe), di non essere legata strettamente ad essa, ma invece di utilizzarne i servizi in maniera "anonima".
L'utilizzo proposto nell'implementazione del nostro sistema è quella di garantire una separazione tra i livelli di logica e di storage, cioè tra l'implementazione della logica di controllo e l'implementazione delle classi che si occuperanno di dialogare con il database. In questo modo l'implementazione del database diventa irrilevante ai fini d'uso del sistema, in quanto nascosta dietro un'astrazione generica, ed il sistema implementato si avvale dei servizi offerti dalle classi preposte alla comunicazione con il database in maniera trasparente.

**Facade:**
Si tratta di un pattern strutturale che viene utilizzato per nascondere la complessità di un sottosistema e ridurre l'accoppiamento tra classi client e semplificare l'utilizzo del sottosistema da parte dei client stessi. L'utilizzo di questo pattern prevede di esporre una interfaccia che rappresenti l'intero sottosistema considerato, in maniera tale da fornire una maniera più efficiente (dal punto di vista dello sviluppatore) di effettuare procedure possibilmente complesse. I sottosistemi più complessi che compongono la piattaforma BeVoyager verranno nascosti dietro un'interfaccia facade in modo tale da semplificarne l'uso.

**Observer:** Si tratta di un pattern comportamentale che viene utilizzato quando si vuole permettere a degli oggetti di essere notificati del cambiamento di stato di altri oggetti, e di effettuare un'azione in risposta.
L'utilizzo proposto è dovuto all'implementazione di un sistema di notifiche per gli utenti, che vengono inviate e ricevute al variare dello stato degli utenti o dei viaggi a cui partecipano.

**Model-View-Controller:** Essendo la piattaforma BeVoyager un sistema pensato per il web, viene naturale utilizzare il pattern MVC. Il pattern permette la separazione della logica di business dallo stato e dalla visualizzazione dello stesso. Si ha così una divisione in tre settori diversi della piattaforma, che ben si adatta alla struttura del web. La view viene intesa come l'interfaccia utente, che viene eseguita sulla macchina client nel browser. La logica di business rappresenta i controller, eseguiti sul server. Dato che la piattaforma viene implementata in linguaggio Java, c'è l'ulteriore vantaggio dell'utilizzo della tecnologia Servlet/JSP, che sostanzialmente implementano il pattern MVC, rendendo molto più semplice l'implementazione del sistema.

# 2. Packages

Di seguito si elencano i *packages* presenti all'interno del sistema:

1. ***AccessController:*** package la cui classe opera controlli sugli accessi.
2. ***DatabaseConnection:*** package le cui classi si occupano delle connessioni al database.
3. ***Feedback:*** package che contiene le classi che creano ed operano su feedback.
4. ***Location:*** package che contiene le classi che creano ed operano su luoghi.
5. ***Log:*** package che contiene le classi per login e logout.
6. ***Notification:*** package che contiene le classi che creano ed operano sulle notifiche.
7. ***Poll:*** package che contiene le classi che creano sondaggi ed operano su essi.
8. ***Route:*** package che contiene le classi che creano ed operano sugli itinerari.
9. ***Travel:*** package che contiene le classi che creano ed operano sui viaggi.
10. ***User:*** package che contiene le classi che creano ed operano sugli utenti.

# 3. Definizione interfacce OCL

**AccessController**

**recoveryPwd:**

Precondizione: ci si aspetta di ricevere l'email di un utente che è registrato alla piattaforma.

Postcondizione: viene restituito un RegisteredUser con la relativa email.

```
context AccessController::recoveryPwd(email) pre:
   UserManager.controlEmail(email)
context AccessController::recoveryPwd(email) post:
   UserManager.containEmail(email) and UserManager.getUserByEmail(email) != null
```

**updatePassword:**

Precondizione: prende in input un'email e una password, l'email deve appartenere ad un RegisteredUser della piattaforma.

Postcondizione: restituisce true nel caso in cui la password del RegisteredUser è stata cambiata.

```
context AccessController::updatePassword(email,password) pre:
   UserManager.controlEmail(email)
```

context AccessController::updatePassword(email,password) post:
UserManager.controlEmail(email) and
user = UserManager.getUserByEmail(email) != null

**logUser:**
Precondizione: prende in input username e password esistenti nel database.
Postcondizione: restituisce un RegisteredUser se il relativo username è nel database.
context AccessController::logUser(username,password) pre:
UserManager.getUser(username,password) != null
context AccessController::logUser(username,password) post:
UserManager.getUser(username,password) != null

**FeedbackController**

**createFeedbackUser:**
Precondizione: prende in input Il RegisteredUser che invia il feedback, l'idUser dell'utente che lo riceve, il messaggio del feedback e la data di rilascio controllando se il RegisteredUser ha partecipato ad un viaggio con l'utente relativo ad idUser.
PostCondizione: restituisce il feedback appena rilasciato.
context FeedbackController::createFeedbackUser(user,idUser,messagge,date) pre:
UserManager.checkUserHasTraveledWith(sender.getId(), idUser)
context FeedbackController::createFeedbackUser(user,idUser,messagge,date) post:
UserManager.checkUserHasTraveledWith(sender.getId(), idUser) and
UserManager.fetchUser(idUser) != null

**createFeedbackRoute:**
Precondizione: prende in input Il RegisteredUser che invia il feedback, l'idRoute dell'itinerario che lo riceve, il messaggio del feedback e la data di rilascio controllando se il RegisteredUser ha partecipato ad un viaggio che comprende quell'itinerario.
PostCondzione: restituisce il feedback relativo all'itinerario a cui è destinato.

context FeedbackController::createFeedbackRoute(user,idRoute,messagge,date) pre:
UserManager.checkUserRoute(sender.getId(), idRoute)
context FeedbackController::createFeedbackRoute(user,idRoute,messagge,date) pre:
UserManager.checkUserRoute(sender.getId(), idRoute) and
RouteManager.fetchRoute(idRoute) != null

**createFeedbackLocation:**
Precondizione: prende in input Il RegisteredUser che invia il feedback, l'idLocation del luogo che lo riceve, il messaggio del feedback e la data di rilascio controllando se il RegisteredUser ha partecipato ad un viaggio che comprende quel luogo.
Postcondizione: restituisce il feedback appena rilasciato al relativo luogo.
context FeedbackController::createFeedbackLocation(user,idLocation,messagge,date) pre:
UserManager.checkUserVisit(sender.getId(), idLocation)
context FeedbackController::createFeedbackRoute(user,idLocation,messagge,date) pre:
UserManager.checkUserLocation(sender.getId(), idLocation) and
RouteManager.fetchLocation(idLocation) != null

**deleteFeedback:**
Precondizione: prende in input un feedback generico
Postcondizione: restituisce un valore booleano se il feedback è stato cancellato.
context FeedbackController::deleteFeedback(id) pre:
context FeedbackController::deleteFeedback(id) post:
FeedbackManager.deleteFeedback(id)

**searchFeedbackUser:**
Precondizione: prende in input l'id di un utente.
Postcondizione: restituisce una lista di feedback relativi ad un utente
context FeedbackController::searchFeedbackUser(id) pre:
context FeedbackController::searchFeedbackUser(id) post:
FeedbackManager.searchFeedbackUser != null

**searchFeedbackRoute:**

Precondizione: prende in input l’id di un itinerario.

Postcondizione: restituisce una lista di feedback relativi ad un itinerario.

context FeedbackController::searchFeedbackRoute(id) pre:

context FeedbackController::searchFeedbackRoute(id) post:

FeedbackManager.searchFeedbackRoute != null

**searchFeedbackLocation:**

Precondizione: prende in input l’id di una locazione.

Postcondizione: restituisce una lista di feedback relativi ad un luogo.

context FeedbackController::searchFeedbackLocation(id) pre:

context FeedbackController::searchFeedbackLocation(id) post:

FeedbackManager.searchFeedbackLocation != null

**LocationController:**

**createLocation:**

Precondizione: prende in input il nome e la descrizione di un luogo.

Postcondizione: restituisce un Location.

context LocationController::createLocation(name,description) pre:

location = new Location(name,description) != null

context LocationController::createLocation(name,description) post:

LocationManager.saveLocationToDB(name,description) != null

**searchLocation:**

Precondizione: prende in input il nome di un Location.

Postcondizione: restituisce una lista di Location relative al nome.

context LocationController::searchLocation(locationName) pre:

context LocationController::searchLocation(locationName) post:
LocationManager.searchLocations(locationName) != null

**getLocation:**
Precondizione: prende in input l'id di un Location.
Postcondition: restituisce il relativo Location se esiste.
context LocationController::getLocation(id) pre:
context LocationController::getLocation(id) post:
LocationManager.fetchLocation(id) != null

**deleteLocation**
Precondizione: prende in input l'id di un Location,
Postcondizione: restituisce un booleano di valore true, se il relativo Location è stato cancellato.
context LocationController::deleteLocation(id) pre:
context LocationController::deleteLocation(id) post:
LocationManager.deleteLocation(id) == true

**PollController:**

**createPoll:**
Precondizione: prende in input la descrizione relativa al Poll, la data di inizio della visita ad un luogo, la data di fine e l'id del relativo viaggio.
Postcondizione: restituisce il Poll creato.
context PollController::createPoll(description, startDate, endDate, travelId) pre:
poll = new Poll(description, startDate, endDate, travelId) != null
context PollController::createPoll(description, startDate, endDate, travelId) post:
poll = new Poll(description, startDate, endDate, travelId) != null and
PollManager.savePollToDB(description, startDate, endDate, travelId) != null

**hasUserVoted:**

Precondizione: prende in input l’id di un utente, e l’id di un Poll.
Postcondizione: restituisce true se l’utente ha già votato il Poll.
context PollController::hasUserVoted(userId, pollId) pre:
!PollManager.checkUserPoll(userId,pollId)
context PollController::hasUserVoted(userId, pollId) post:
!PollManager.checkUserPoll(userId,pollId)

**updatePoll:**
Precondizione: prende in input l’id di un Poll, il voto relativo, e l’utente che ha rilasciato il voto.
Postcondizione: restituisce true se l’operazione di aggiornamento è andata a buon fine.
context PollController::updatePoll(id,vote,userId) pre:
PollManager.fetchPoll(id) != null
context PollController::updatePoll(id,vote,userId) post:
PollManager.updatePoll(id,vote) == true

**getPoll:**
Precondizione: prende in input l’id di un Poll.
Postcondizione: restituisce il relativo Poll se questo esiste nel database.
context PollController::getPoll(id) pre:
PollManager.fetchPoll(id) != null
context PollController::getPoll(id) post:
PollManager.fetchPoll(id) != null

**RouteController:**

**createRoute:**
Precondizione: prende in input una lista di Location associate, il nome e la descrizione di un Route.
Postcondizione: restituisce il Route appena creato.
context RouteController::createRoute(locations,name,description) pre:
context RouteController::createRoute(locations,name,description) post:

RouteManager.saveRouteToDB(route)
route != null

**searchRoute:**

Precondizione: prende in input il nome di un Location.

Postcondizione: restituisce una lista di Location associata al nome.

context RouteController::searchRoute(locationName) pre:

context RouteController::searchRoute(locationName) post:
!RouteManager.searchRoute(locationName).isEmpty()

**addLocationToRoute:**

Precondizione: prende in input un Location e un Route.

Postcondizione: restituisce true se il Location è stato aggiunto al precedente Route.

context RouteController::addLocationRoute(location,route) pre:
!isLocationInRoute(location.getId(),route.getLocations())

context RouteController::addLocationRoute(location,route) post:
!isLocationInRoute(location.getId(),route.getLocations()) and
RouteManager.updateRoute(route) == true

**removeLocationFromRoute:**

Precondizione: prende in input un Location e un Route.

Postcondizione: restituisce true se il Location è stato rimosso dal Route.

context RouteController::removeLocationRoute(location,route) pre:
isLocationInRoute(location.getId(),route.getLocations())

context RouteController::removeLocationRoute(location,route) post:
isLocationInRoute(location.getId(),route.getLocations()) and
RouteManager.updateRoute(route) == true

**getRoute:**

Precondizione: prende in input l’id di un Route.
Postcondizione: restituisce un Route se esso esiste in database.

context RouteController::getRoute(id) pre:
context RouteController::getRoute(id) post:
  RouteManager.fetchRoute(id) != null

**updateRoute:**

Precondizione: prende in input un Route.
Postcondizione: restituisce true se questo è stato aggiornato.

context RouteController::updateRoute(route) pre:
context RouteController::updateRoute(route) post:
  RouteManager.updateRoute(route)

**TravelController:**

**isUserInTravel:**

Precondizione: prende in input l’id di un RegisteredUser e di un Travel.
Postcondizione: restituisce true se il RegisteredUser partecipa a quel Travel.

context TravelController::isUserInTravel(idUser,users) inv:
  users.get(position).getId() == isUser

**addUserInTravel:**

Precondizione: prende in input l'id di un RegisteredUser e l'id di un Travel.
Postcondizione: restituisce true se il RegisteredUser non partecipa già al relativo Travel.

context TravelController:: addUserInTravel(idUser,idTravel) pre:
!isUserInTravel(idUser,travel.getPartecipants()) and
user.getId() != travel.getCreatoreViaggio().getId()
context TravelController:: addUserInTravel(idUser,idTravel) post:
!isUserInTravel(idUser,travel.getPartecipants()) and
user.getId() != travel.getCreatoreViaggio().getId() and
results = TravelManager.updateTravel(travel) == true

**deleteTravel:**

Precondizione: prende in input l'id di un Travel.
Postcondition: restituisce true se il Travel è stato cancellato dal database.

context TravelController::deleteTravel(id) pre:
deleteTravel(id) == true
context TravelController::deleteTravel(id) post:
deleteTravel(id) == true

**createTravel:**

Precondizione: prende in input il Route da inserire nel Travel, il nome, il RegisteredUser che sta creando il Travel, il tipo (modificabile o no), la data di inizio e la data di fine.
Postcondizione: restituisce il Travel appena creato.

context TravelController::createTravel(route,name,creator,type,startDate,endDate) pre:
  travel = new Travel(route,name,creator,type,startDate,endDate) != null
context TravelController::createTravel(route,name,creator,type,startDate,endDate) post:
  travel = new Travel(route,name,creator,type,startDate,endDate) != null and
  travel = saveTravelToDB(route,name,creator,type,startDate,endDate) != null

**confirmTravel:**

Precondizione: prende in input un Travel.
Postcondizione: restituisce true se il Travel è stato aggiornato a modificabile.

context TravelController::confirmTravel(travel) pre:
context TravelController::confirmTravel(travel) post:
  TravelManager.updateTravel(travel)

**getTravel:**

Precondizione: prende in input l'id di un Travel.
Postcondizione: restituisce un Travel se esiste nel database.

context TravelController::getTravel(id) pre:
context TravelController::getTravel(id) post:
  TravelManager.fetchTravel(id) != null

**filterTravelByDate:**

Precondizione: prende in input uno startDate,un endDate relativi ai Travel, e una lista di Travel.
Postcondizione: restituisce una lista di Travel se ne esistono con lo startDate passato.

context TravelController::filterTravelByDate(startDate,endDate,travels) pre:
startDate.before(curStartDate) and endDate.after(curEndDate)
context TravelController::filterTravelByDate(startDate,endDate,travels) post:
startDate.before(curStartDate) and endDate.after(curEndDate) and
travels != null

**UserController:**

**getUser:**

Precondizione: prende in input l'id di un RegisteredUser.
Postcondizione: restituisce un RegisteredUser se questo esiste in database.

context UserController::getUser(id) pre:
context UserController::getUser(id) post:
UserManager.fetchUser(id) != null

**searchUser:**

Precondizione: prende in input l'username di un RegisteredUser.
Postcondizione: restituisce una lista di RegisteredUser con username simile a quello passato.

context UserController::searchUser(username) pre:
context UserController::searchUser(username) post:
UserManager.searchUser(username) != null

**createUser:**

Precondizione: prende in input il nome, il cognome, l'username, l'email, la password e il birthDate relativo ad un RegisteredUser.
PostCondizione: restituisce il relativo RegisteredUser se questo è stato creato.

context UserController::createUser(username,name,lastName,email,password,birthDate) pre:
context UserController::createUser(username,name,lastName,email,password,birthDate) post:

UserManager.saveUserToDB(username,name,lastName,email,password,birthDate) != null

**deleteUser:**

Precondizione: prende in input l'id di un RegisteredUser.
Postcondizione: restituisce true se il RegisteredUser è stato cancellato.

context UserController::deleteUser(id) pre:
context UserController::deleteUser(id) post:
UserManager.deleteUser(id) == true