Object Design Document

urCoach

Partecipante	Matricola
Salvatore Fasano	0512105068
Mariantonia Candela	0512105374

Sommario

1. Introduzione

1.1 Object Design Trade-Offs

Buy vs **Build**: la necessità di sviluppare l'applicazione web in tempi e costi ristretti ci porta ad utilizzare dei componenti off-the-shelf, si sceglie quindi di utilizzare dei framework moderni che siano adatti allo sviluppo della piattaforma in modo da renderla quanto più fedele all'idea iniziale. Tuttavia si sceglie di limitare l'utilizzo di tali componenti alle tecnologie che si occuperanno del back-end in quanto non si vuole rinunciare ad un'interfaccia ed un'esperienza unica.

1.1.1 Componenti Off-the-Shelf

Il back-end sarà costruito utilizzando il framework **Spring** nota soluzione nell'ambito delle applicazioni distribuite Java. Composto da un core ben ottimizzato e prodotto con l'obbiettivo di ridurre quello che il codice "boilerplate", consente agli sviluppatori di concentrarsi maggiormente sulla logica di business dell'applicazione, piuttosto che sulla comunicazione tra le varie componenti. Sarà utilizzata l'ultime versione **Spring Boot 2.0**, in questo modo si potrà utilizzare tutta la potenza e flessibilità del framework con minori vincoli legati alle configurazioni XML e possibilità nativa di scalare poi su cloud. I moduli d'interesse per questo progetto saranno:

- Spring MVC (che fornisce tutto il meccanismo di controllo del flusso)
- Spring Data JPA (ORM)
- Sping Security: permette di gestire la sicurezza e le politiche di accesso a determinate pagine protette controllando l'utente che prova ad accedere ed i permessi a lui associati; tale modulo verrà utilizzato esclusivamente per gestire le politiche di accesso degli utenti amministrativi (Recruiter e Gestore Ordini), tale tipologia di utenti sfrutterà la funzionalità di in memory authentication, ciò ci consentirà di non memorizzare le loro informazioni nella base di dati ma saranno tenute nel codice (lo stesso modulo abilita una codica di tali informazioni, eventuali attacchi all'ecommerce che permettano la lettura del codice non mostreranno le credenziali di accesso reali), tale scelta risulta ragionevole in quanto non ci saranno più Recruiter/Gestore Ordini, in una futura crescita sarà facilmente possibile collegare tale meccanismo di accesso al database da cui prelevare dati.

La comunicazione tra front-end e back-end verrà realizzata mediante il template engine **Thymeleaf**, in questo modo potremo realizzare codice HTML5 complain e con una semplicità che non avremmo avuto con altre soluzioni (JSP, JSF). Tutte le componenti selezionate sono gratuite ed open source, scelta in linea con i requisiti di costo.

1.1.2 Design Patterns

Per favorire una maggiore minimalità e riusabilità del codice si è deciso di utilizzare alcuni design pattern, non definiti dalla GoF ma stabiliti negli anni grazie alle soluzioni performanti che propongono per problemi in ambito enterprise.

1.1.2.1 Data mapper

Il Data Mapper è un pattern promosso da Martin Fowler con l'obbiettivo di separare gli oggetti che rappresentano l'informazione persistente dalla logica relativa alla mappatura di questa informazione sul database sottostante. Ciò permette agli oggetti che risiedono in memoria di essere completamente disaccoppiati dal livello sottostante (indipendentemente da quale esso sia, database relazionale, file o altro), favorendo la manutenibilità del sistema.

1.1.2.2 Repository pattern

Proposto da Edward Hieatt e Rob Mee, questo pattern promuove la definizione di un livello intermedio tra gli oggetti che realizzano la logica di business e quelli che accedono ai dati persistenti. In questo modo, si fornisce all'applicazione un'interfaccia tramite cui gli oggetti persistenti possono essere visti come allocati in una normale collezione su cui è possibile effettuare operazioni banali di aggiunta, rimozione, aggiornamento e ricerca: gli oggetti che interagiscono con questa collezione non sono a conoscenza di ciò che accade dietro le quinte, in quanto è proprio la repository ad occuparsi della propagazione dell'informazione alla sorgente di dati persistente. È la repository stessa ad incapsulare tutte le possibili operazioni di ricerca all'interno della collezione di oggetti: il risultato è una visione maggiormente orientata agli oggetti anche per le operazioni su oggetti persistenti che risiedono su database relazionali.

1.1.2.3 Service Layer

Talvolta le applicazioni che sviluppiamo offrono le proprie funzionalità tramite interfacce differenti: ognuna di esse presenta l'insieme di operazioni disponibili in modo differente ma la logica applicativa è comune a tutte queste interfacce. Da ciò la proposta di Randy Stafford: separare completamente la logica di presentazione da quella di business, collocando quest'ultima in un livello che espone tutti i servizi che l'applicazione ha da offrire. Il risultato è quindi quello di un'applicazione la cui logica applicativa è completamente indipendente dalla modalità di presentazione: segue che l'applicazione è maggiormente manutenibile ed estendibile con nuovi servizi ed interfacce. Tramite il livello di servizio abbiamo quindi la definizione di oggetti che espongono tutti i servizi offerti dal relativo sottosistema, in grado di realizzare transazioni su multiple entità di dominio ed accentrare i controlli di sicurezza.

1.2 Linee guida per la documentazione dell'interfaccia

È richiesto agli sviluppatori di seguire le seguenti linee guida al fine di essere consistenti nell'intero progetto e facilitare la comprensione delle funzionalità di ogni componente.

1.2.1 Nomenclatura delle componenti

1.2.1.1 Nomi delle classi

- Ogni classe deve avere nome in CamelCase
- Ogni classe deve avere nome singolare
- Ogni classe che modella un'entità deve avere per nome un sostantivo che possa associarla alla corrispondente entità di dominio
- Ogni classe che realizza la logica di business per una determinata entità deve avere nome composto da quello del pacchetto per cui espone servizi seguito da "Service"
- Ogni classe che modella una collezione in memoria per una determinata classe di entità deve avere nome composto da quello dell'entità su cui opera seguito da "Repository"
- Ogni classe che realizza un form deve avere nome composto dal sostantivo che descrive il form seguito dal suffisso "Form"
- Ogni classe che realizza un servizio offerto via web deve avere nome composto dal nome del pacchetto per cui espone servizi seguito dal suffisso "Controller"

1.2.1.2 Nomi dei metodi

- Ogni metodo deve avere nome in lowerCamelCase

1.2.1.3 Nomi delle eccezioni

- Ogni eccezione deve avere nome esplicativo del problema segnalato

1.2.1.4 Nomi degli altri sorgenti

- Ogni documento HTML deve avere nome che possa ricondurre al contenuto da essa mostrato

1.2.1.5 Organizzazione delle componenti

- Tutte le classi che realizzano un sottosistema devono essere racchiuse nello stesso pacchetto Java
- Tutte le componenti che realizzano l'interfaccia grafica devono essere collocate nella directory "/resources/templates/View/".
- Tutte le risorse statiche (fogli di stile, script e immagini) devono essere collocate nella directory "/resources/static/<tiporisorsa>" dove tipo risorsa può essere css, js o img.

1.2.1.5 Organizzazione del codice

- Il codice Java dev'essere indentato in maniera appropriata (tramite tabulazione corrispondente a 4 spazi) e l'apertura di un blocco di codice deve avvenire nella stessa riga in cui è specificato il nome della classe o la firma del metodo che quel blocco definisce.
- Il codice HTML dev'essere indentato in maniera appropriata (tramite tabulazione corrispondente a 4 spazi) e gli attributi devono essere indicati in minuscolo.

1.3 Definizioni, acronimi ed abbreviazioni

N/A

1.4 Riferimenti

- Design Goals: sezione 1.2 dell'SDD
- Scelta dell'ambiente d'esecuzione: sezioni 3.2 e 3.3 dell'SDD
- Spring Boot: documentazione
- Google Coding Style: Java, HTML/CSS
- Design Patterns: Data Mapper, Repository, Service Layer

2. Packages

In questa sezione presentiamo in modo più approfondito quella che è la divisione in sottosistemi e l'organizzazione del codice in file.

2.1 Divisione in pacchetti

Come definito nel documento SDD il sistema si baserà su un'architettura MVC. Vengono tuttavia effettuate delle modifiche portate dal framework Spring: il sottosistema di accesso ai dati persistenti non verrà implementato dagli sviluppatori in quanto basterà fornire un'interfaccia (chiamata repository) che specifica i metodi che dovranno offrire le classi di accesso al database, sarà poi Spring ad occupparsi della concretizzazione di tali interfacce. Ogni layer avrà un proprio package:

- Model: contiene i pacchetti entity, repository e service
- Control: contiene un pachetto per ogni sottosistema, tali pacchetti contengono il file control che si occupa del sottosistema
- View: contiene i file html

2.2 Organizzazione del codice in file

Come imposto da Java, ogni classe del sistema sarà collocata nel relativo file. I nomi dei pacchetti avranno tutti come prefisso it.unisa.di.urcoach (e saranno mappati nel rispettivo percorso src/main/java/it/unisa/di/…) ad eccezione del pacchetto realizzante l'interfaccia utente, che sarà invece collocato nella directory src/main/resources/templates/View/. Si farà inoltre ricorso alla directory src/main/resources ed al package config per la definizione di configurazioni e messaggi.

3. Interfacce delle classi

Nelle seguenti tabelle si illustrano per ogni classe i relativi metodi con pre-condizioni, post-condizioni ed invariante con sintassi OCL. Le classi rappresentanti le entità del sistema non vengono citate per la loro banalità in quanto sono unicamente costituite da costruttori, getter e setter. Vengono riportate le classi che costituiscono il livello di Service per mostrare le interfacce e si precisa che, utilizzando il framework Spring Boot col modulo Spring JPA, tale livello si occupa solamente di fornire un'interfaccia chiara allo sviluppatore che implementa un sottinsieme di tutte le operazioni che il framework mette a disposizione mediante le Repository, per cui ogni metodo delle classi service si occupa di chiamare e tornare il risultato del metodo corrispondente nella classe repository offerta dal framework.

Nome Classe	AcquistoServiceImpl
Descrizione	Questa classe è un manager che mediante una repository accede alle informazioni riguardanti gli acquisti all'interno del database.
Signature Metodi	+ findAll(): List<Acquisto> + findByFattura(f: Fattura): List<Acquisto> + findByPersonalTrainer(pt: PersonalTrainer): List<Acquisto> + save(a: Acquisto)
Pre-Condizioni	
Post-Condizioni	
Invariante	

Nome Classe	AtletaServiceImpl
Descrizione	Questa classe è un manager che mediante una repository accede alle informazioni riguardanti gli atleti all'interno del database.
Signature Metodi	+ findAll(): List<Atleta> + findByEmail(email: String): Atleta + checkUser(email: String, password: String): Atleta + save(a: Atleta) + deleteByEmail(email: String)
Pre-Condizioni	
Post-Condizioni	
Invariante	

Nome Classe	CategoriaServiceImpl
Descrizione	Questa classe è un manager che mediante una repository accede alle informazioni riguardanti le categorie all'interno del database.
Signature Metodi	+ findAll(): List<Categoria> + findByNome(nome: String): Categoria
Pre-Condizioni	
Post-Condizioni	

Invariante	

Nome Classe	FatturaServiceImpl
Descrizione	Questa classe è un manager che mediante una repository accede alle informazioni riguardanti le fatture all'interno del database.
Signature Metodi	+ findAll(): List<Fattura> + findByAtleta(a: Atleta): List<Fattura> + save(f: Fattura)
Pre-Condizioni	
Post-Condizioni	
Invariante	

Nome Classe	PacchettoServiceImpl
Descrizione	Questa classe è un manager che mediante una repository accede alle informazioni riguardanti i pacchetti all'interno del database.
Signature Metodi	+ findAll(): List<Pacchetto> + findAllByCategoria(categoria: Categoria) : List<Pacchetto> + findAllByCosto(valore: float) : List<Pacchetto> + findAllByPersonalTrainer(pt: PersonalTrainer) : List<Pacchetto> + findLast(): List<Pacchetto> + findByNome(nome: String) : List<Pacchetto> + Pacchetto findById(id: int): Pacchetto + save(p: Pacchetto) + deleteById(id: int)
Pre-Condizioni	
Post-Condizioni	
Invariante	

Nome Classe	PersonalTrainerServiceImpl
Descrizione	Questa classe è un manager che mediante una repository accede alle informazioni riguardanti i personal trainer all'interno del database.
Signature Metodi	+ findAll(): List<PersonalTrainer> + findByEmail(email: String): PersonalTrainer + findByVerificato(verificato: int): List<PersonalTrainer> + checkUser(email: String, password, String): PersonalTrainer + save(pt: PersonalTrainer) + deleteByEmail(email: String)
Pre-Condizioni	
Post-Condizioni	
Invariante	

Nome Classe	MainControl
Descrizione	Questa classe è un control che si occupa di gestire le richieste web alla homepage
Signature Metodi	+ getHome(model: Model, req: HttpServletRequest): String
Pre-Condizioni	
Post-Condizioni	
Invariante	

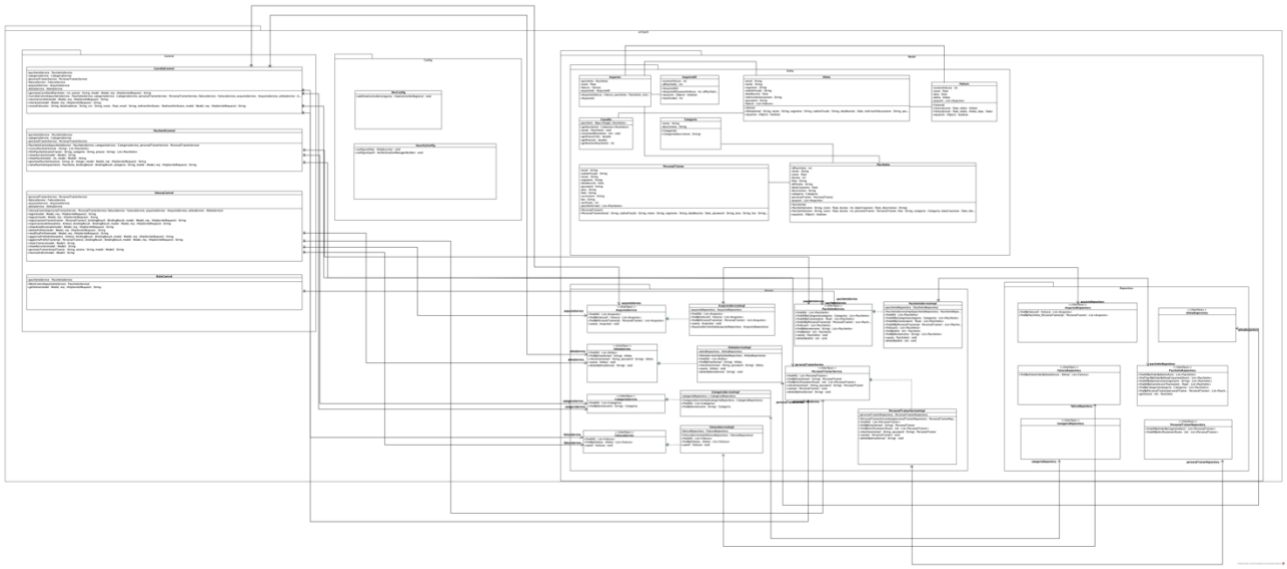
Nome Classe	UtenzaControl
Descrizione	Questa cloasse è un control che si occupa di gestire le richieste web alle funzionalità del sottosistema Utenza
Signature Metodi	+ login(model: Model, req: HttpServletRequest): String + logout(model: Model, req: HttpServletRequest): String + registrazioneTrainer(trainer: PersonalTrainer, bindingResult: BindingResult, model: Model, req: HttpServletRequest): String + registrazioneAtleta(atleta: Atleta, bindingResult: BindingResult, model: Model, req: HttpServletRequest): String + showAreaPersonale(model: Model, req: HttpServletRequest): String + deleteProfilo(model: Model, req: HttpServletRequest): String + modificaProfilo(model: Model, req: HttpServletRequest): String + aggiornaProfiloAtleta(atleta: Atleta, bindingResult: BindingResult, model: Model, req: HttpServletRequest): String + aggiornaProfiloPersonalTrainer(trainer: PersonalTrainer, bindingResult: BindingResult, model: Model, req: HttpServletRequest): String + showTrainer(Model model): String + showRecruiter(Model model): String + gestioneTrainer(emailTrainer: String, azione:String, model: Model): String + mostraOrdini(Model model): String
Pre-Condizioni	**context** UtenzaControl::login(model, req) **pre**: req.getParameter("usernameL") != null and req.getParameter("passwordL") != null and req.getParameter("tipo") != null **context** UtenzaControl::logout(model, req) **pre**: req.getSession().getAttribute("logged") **context** UtenzaControl::registrazioneTrainer(trainer, bindingResult, model, req) **pre**: trainer.isValid() **context** UtenzaControl::registrazioneAtleta (atleta, bindingResult, model, req) **pre**: atleta.isValid() **context** UtenzaControl::showAreaPersonale(model, req) **pre**: req.getSession.getAttribute("logged") != null **context** UtenzaControl::deleteProfilo(model, req) **pre**: req.getSession.getAttribute("logged") != null **context** UtenzaControl::modificaProfilo(model, req) **pre**: req.getSession.getAttribute("logged") != null

	context UtenzaControl::aggiornaProfiloAtleta(atleta, bindingResult, model, req) **pre**: atleta.isValid() **context** UtenzaControl::aggiornaProfiloTrainer(trainer, bindingResult, model, req) **pre**: trainer.isValid() **context** UtenzaControl::gestioneTrainer (emailTrainer, azione, model) **pre**: database.trainers -> includes(trainer) and (azione.equals("verifica") or azione.equals("invalida") or azione.equals("rimuovi"))
Post-Condizioni	**context** UtenzaControl::login(model, req) **post**: req.getSession().getAttribute("logged") or !req.getSession().getAttribute("logged")) **context** UtenzaControl::logout(model, req) **post**: req.getSession().isEmpty() **context** UtenzaControl::registrazioneTrainer(trainer, bindingResult, model, req) **post**: database.personalTrainer -> includes(trainer) **context** UtenzaControl::registrazioneAtleta(atleta, bindingResult, model, req) **post**: database.atleta -> includes(atleta) **context** UtenzaControl::showAreaPersonale(model, req) **post**: model -> includes(fatture) or model -> includes(vendite) **context** UtenzaControl::deleteProfilo(model, req) **post**: not database.personalTrainer -> includes(req.getSession().getAttribute("trainer")) or not database.atleta -> includes(req.getSession().getAttribute("atleta")) **context** UtenzaControl::modificaProfilo(model, req) **post**: model -> includes(utente) **context** UtenzaControl::aggiornaProfiloAtleta(atleta, bindingResult, model, req) **post**: database.atleta -> includes(atleta) **context** UtenzaControl::aggiornaProfiloTrainer(trainer, bindingResult, model, req) **post**: database.personalTrainer -> includes(trainer) **context** UtenzaControl::gestioneTrainer (emailTrainer, azione, model) **post**: not database.trainers -> includes(trainer) or trainer.getVerificato() ==0 or trainer.getVerificato() == 1
Invariante	

Nome Classe	PacchettiControl
Descrizione	Questa cloasse è un control che si occupa di gestire le richieste web alle funzionalità del sottosistema Pacchetti
Signature Metodi	+ ricercaPacchetti(nome: String): List<Pacchetto> + filtriPacchetti(nomeTrainer: String, categoria: String, prezzo: String): List<Pacchetto> + showPacchetti(model: Model): String + showPacchetto(id: int, model: Model): String + gestionePacchetti(azione: String, id: int, model: Model, req: HttpServletRequest): String + salvaPacchetto(pacchetto: Pacchetto, bindingResult: BindingResult, categoria: String, model: Model, req: HttpServletRequest): String
Pre-Condizioni	**context** PacchettiControl::ricercaPacchetti(nome) **pre**: nome != null **context** PacchettiControl::showPacchetto(id, model) **pre**: database.pacchetti -> includes(pacchetto) **context** PacchettiControl::gestionePacchetti(azione, id, model, req) **pre**: req.getSession() -> includes(trainer) **context** PacchettiControl::salvaPacchetto(pacchetto, bindingresult, categoria, model, req) **pre**: req.getSession() -> includes(trainer) and pacchetto.isValid()
Post-Condizioni	**context** PacchettiControl::ricercaPacchetti(nome) **post**: NA **context** PacchettiControl::showPacchetto(id, model) **post**: model -> includes(pacchetto) **context** PacchettiControl::gestionePacchetti(azione, id, model, req) **post**: **context** PacchettiControl::salvaPacchetto(pacchetto, bindingresult, categoria, model, req) **post**: database.pacchetti -> includes(pacchetto)
Invariante	

Nome Classe	CarrelloControl
Descrizione	Questa cloasse è un control che si occupa di gestire le richieste web alle funzionalità del sottosistema Carrello
Signature Metodi	+ gestioneCarrello(idPacchetto: int, azione: String, model: Model, req: HttpServletRequest): String + mostraCarrello(model: Model, req: HttpServletRequest): String + checkout(model: Model, req: HttpServletRequest): String + nuovaFattura(cc: String, dataScadenza: String, cvc: String, costo: float, email: String, redirectAttributes: RedirectAttributes, model: Model, req: HttpServletReq): String
Pre-Condizioni	**context** CarrelloControl::gestioneCarrello(idPacchetto, azione, model, req) **pre**: database.pacchetti -> includes(pacchetto) and (azione.equals("aggiungi") or azione.equals("rimuovi")) **context** CarrelloControl::checkout(model, req) **pre**: req.getSession() -> includes(atleta) **context** CarrelloControl::nuovaFattura(cc, dataScadenza, cvc, costo, email, redirectAttributes, model, req) **pre**: cc.isValid() and dataScadenza.isValid() and cvc.isValid()
Post-Condizioni	**context** CarrelloControl::gestioneCarrello(idPacchetto, azione, model, req) **post**: carrello -> includes(pacchetto) or not carrello -> includes(pacchetto) **context** CarrelloControl::checkout(model, req) **post**: model -> includes(pacchetti) **context** CarrelloControl::nuovaFattura(cc, dataScadenza, cvc, costo, email, redirectAttributes, model, req) **post**:
Invariante	

4. Class Diagram



5. Glossario

- Componenti off-the-shelf: prodotti software sviluppati da terzi riutilizzabili
- CSS: Linguaggio per la definizione degli stili delle pagine web
- Framework: Software di supporto allo sviluppo
- HTML: Linguaggio per la strutturazione delle pagine web
- JavaScript: Linguaggio di scripting nato per dare dinamicità alle pagine HTML
- Codice boilerplate: Sezione di codice ripetuta in più punti senza alterazioni
- JavaServer Pages (JSP): Tecnologia che facilita lo sviluppo di pagine web dinamiche fornendo un’interfaccia Java tagoriented con il back-end