EasyAgreement

ODD: Object Design Document

Versione **1.1**

Data	11/12/2019
Destinatario	Prof. Carmine Gravino, Prof.ssa Filomena Ferrucci
Presentato da	Tutti i team member
Approvato da	Francesco Califano, Domenico Marino

Revision History:

Data	Versione	Descrizione	Autori
06/12/2019	0.1	Prima stesura	[Tutti]
06/12/2019	1.0	Sviluppo introduzione	Marco Borrelli Marco Ciano Luigi Pasetti Armando Soddisfatto Roberto Veneruso
7/12/2019	1.0	Aggiunta Design Pattern e Package	[Tutti]
9/12/2019	1.0	Aggiunta Class Interfaces e Glossario	[Tutti]
11/12/2019	1.1	Correzione formattazione del documento	Armando Soddisfatto, Veronica Volpicelli

Sommario

1. Introduzione

1.1 Object Design trade-off

Dopo un'attenta analisi durante la fase di Object Design, per quanti riguarda la realizzazione del sistema, sono stati individuati i seguenti trade-off:

Response Time vs Memory: se il software non rispetta i requisiti di tempo di risposta è necessario utilizzare più memoria per velocizzare il sistema.
Performance vs Maintenance: la manutenibilità sarà preferita alla performance in modo da facilitare gli sviluppatori nei processi di aggiornamento del software.
Availability vs Fault Tollerance: il sistema deve sempre essere disponibile all'utente in caso di errore in una funzionalità, anche al costo di rendere non disponibile quest'ultima per un lasso di tempo.
Developement Cost vs Security: il sistema non permette di sviluppare un prodotto software che sia simultaneamente sicuro e costi poco.
Memory vs Extensibility: il sistema deve permettere l'estensibilità a discapito della memoria utilizzata, in modo tale da permettere al cliente di richiedere agli sviluppatori nuove funzionalità.

1.2 Interface Documentation Guidelines

Gli sviluppatori dovranno seguire precise linee guida per la stesura del codice. Per la scrittura del codice JavaScript ci si atterrà allo standard Google JavaScript.

1.2.1 Package

I nomi dei package sono tutti "lowerCamelCase". Per esempio, my.exampleCode.deepSpace, ma non my.examplecode.deepspace o my.example_code.deep_space

1.2.2 Classi e interfacce JavaScript

Classi, interfacce, record e name typedef sono scritti in "upperCamelCase". Le classi non esportate sono locali: non sono contrassegnate con @private e quindi non sono indicate con un carattere di sottolineatura finale. I nomi dei tipi sono tipicamente nomi o nomi di frasi. Ad esempio, Request, ImmutableList, o VisibilityMode. Inoltre, i nomi delle interfacce possono essere aggettivi o frasi di aggettivo (esempio, Readable).

1.2.3 Metodi

I nomi dei metodi sono scritti in "lowerCamelCase". I nomi dei metodi @private devono terminare con un carattere di sottolineatura finale. I nomi dei metodi sono tipicamente verbi o frasi di verbi. Per esempio, sendMessage o stop_. I metodi get e set, per le proprietà non sono mai richieste, ma se vengono utilizzate dovrebbero essere denominate getFoo (isFoo o hasFoo per i boolean) o setFoo(value) per i set.

1.2.4 Nomi costanti

I nomi costanti usano COSTANT_CASE: tutte le lettere maiuscole, con parole separate dalle sottolineature. Ogni costante è una proprietà statica @const o una dichiarazione const locale nel modulo. Prima di scegliere il caso costante, bisogna considerare se il campo costante sembra davvero una costante immutabile.

1.2.5 Nomi non costanti

I nomi di campi non constanti (static o altro) sono scritti in "lowerCamelCase" con una sottolineatura finale per i campi privati. Questi nomi sono in genere nomi o frasi. (es. computedValues).

1.2.6 Nomi dei Parametri

I nomi dei parametri sono scritti in "lowerCamelCase".
I nomi dei parametri formati da un solo carattere devono essere evitati nei metodi pubblici. Eccezione: quando richiesto da un framework di terze parti, i nomi dei parametri possono iniziare con $. Questa eccezione non si applica ad altri identificatori (ad esempio variabili locali)

1.2.7 Nomi delle variabili Locali

I nomi delle variabili locali sono scritti in "lowerCamelCase", solo quando le variabili finali, immutabili o locali non sono considerati constanti.

1.2.8 Enum names

Gli enum names sono scritti in "UpperCamelCase", simili alle classi, composti generalmente da un singolo nome.

1.2.9 Nomi dei parametri del modello

I nomi dei parametri del modello devono essere concisi, singola parola o singola lettera, e devono essere maiuscoli, come TYPE or THIS.

1.2.10 Nomi Locali del modulo

I nomi locali del modulo non vengono esportati, sono privati. Non sono contrassegnati con @private e non terminano con un trattino basso. Questo vale per classi, funzioni, variabili, costanti, enum e altri identificatori del modulo.

1.2.11 Script JavaScript

Gli Script in JavaScript devono rispettare le seguenti convenzioni:

1. Gli script che svolgono funzioni distinte dal mero rendering della pagina dovrebbero essere collocati in file dedicati.
2. Il codice JavaScript deve seguire le stesse convenzioni per il layout.
3. I documenti JavaScript devono iniziare con un commento.
4. Le funzioni JavaScript devono essere documentate.
5. Gli oggetti JavaScript devono essere preceduti da un commento.

1.2.12 Pagine HTML

Le pagine HTML, sia in forma statica che dinamica, devono essere conformi allo standard HTML. Inoltre, il codice HTML statico deve utilizzare l'indentazione, per facilitare la lettura, secondo le seguenti regole:

1. Un'indentazione consiste in una tabulazione;
2. Ogni tag deve avere un'indentazione maggiore del tag che lo contiene;
3. Ogni tag di chiusura deve avere lo stesso livello di indentazione del corrispondente tag di apertura;
4. I tag di commento devono seguire le stesse regole che si applicano ai tag normali.

1.2.13 Fogli di stile CSS

I fogli di stile (CSS) devono seguire le seguenti convenzioni:
Tutti gli stili non in-line devono essere collocati in fogli di stile separati.
Ogni foglio di stile deve essere iniziato da un commento analogo a quello presente nei file Java.
Ogni regola CSS deve essere formattata come segue:

1. I selettori della regola si trovano a livello 0 di indentazione, uno per riga;
2. L'ultimo selettore della regola è seguito da parentesi graffa aperta "{";
3. Le proprietà che costituiscono la regola sono listate una per riga e sono indentate rispetto ai selettori;
4. La regola è terminata da una parentesi graffa "}", collocata da sola su una riga;

1.2.14 Database MongoDB

I nomi delle collection devono seguire le seguenti regole:

1. Devono essere costituiti da sole lettere maiuscole;
2. Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.

I nomi dei campi devono seguire le seguenti regole:

1. Devono essere costituiti da sole lettere maiuscole;
2. Se il nome è costituito da più parole, è previsto l'uso di underscore (_);
3. Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.

1.3 Definizioni, Acronimi e Abbreviazioni

Acronimi**:**

- RAD: Requirements Analysis Document
- SDD: System Design Document
- ODD: Object Design Document

Definizioni:

- **upperCamelCase:** è una tecnica di Naming delle variabili, consiste nell'iniziare la parola con una lettera maiuscola e le rimanenti lettere minuscole.
- **lowerCamelCase:** è la pratica di scrivere parole composte o frasi unendo tutte le parole tra loro, consiste nello scrivere più parole insieme delimitando la fine e l'inizio di una nuova parola con una lettera maiuscola.
- **HTML:** Linguaggio di programmazione utilizzato per lo sviluppo di pagine Web.
- **CSS:** acronimo di Cascading Style Sheets è un linguaggio usato per definire la formattazione delle pagine Web.
- **JavaScript:** JavaScript è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione di effetti dinamici interattivi.
- **MongoDB:** è un DBMS non relazionale, orientato ai documenti.
- **MVC:** acronimo di Model-View-Controller, è un pattern architetturale molto diffuso nello sviluppo di sistemi software.

- **Bootstrap:** è una raccolta di strumenti liberi per la creazione di siti e applicazioni per il web.
- **Off-The-Shelf:** Servizi esterni al sistema di cui viene fatto utilizzo.

1.4 Riferimenti

- Documento EA_RAD_V_2.1.pdf
- Documento EA_SDD_V_1.1.pdf
- Documento EA_CP_V_1.1.pdf
- Libro: Object-Oriented Software Engineering (Using UML, Patterns, and Java) Third Edition Autor: Bernd Brugge & Allen H. Dutoit

2. Packages

2.1 View

Il package View è formato dalle pagine html (e relativi css) contenenti informazioni e funzionalità per tutti gli attori del sistema. Alcuni elementi della view sono creati dinamicamente dal controller in base a specifiche necessità. Alcune di queste sono esclusive di un determinato utente, altre globali. La ripartizione come segue:

Generic

- login.html – permette di effettuare il login.
- signup.html – permette la registrazione di un nuovo utente.
- index.html – raggruppa le funzionalità specifiche per ogni gruppo di utente e fornisce accesso ad elementi di view generati dinamicamente come la chat o la gestione delle notifiche. È comune a tutti gli utenti.
- viewtutor.html – visualizzazione di una lista di tutor
- vieworg.html – visualizzazione di una lista di organizzazioni esterne.
- viewinfo.html - visualizzazione del profilo di un tutor esterno/organizzazione ospitante. Include per l'amministratore la possibilità, se si sta visualizzando il profilo di un tutor esterno o di un'organizzazione ospitante, di eliminarle dalla piattaforma.

Student

- compileLAStudent.html – compilazione Learning Agreement
- editLA.html- modifica Learning Agreement
- viewLA.html – visualizzazione dello stato del proprio learning agreement
- profile.html – visualizzazione del proprio profilo o di quello di un tutor/organizzazione esterna, se si sta visualizzando il proprio profilo è possibile la modifica dei dati dalla stessa pagina password inclusa.
- dochandler.html – permette allo studente l'upload, la visualizzazione e la rimozione della propria documentazione (carta d'identità, cv).

Academic Tutor

- compileLAAcademicTutor.html – compilazione della tabella B del Learning Agreement a seguito di selezione di un Learning Agreement inviato da uno studente.

- viewrequest.html – visualizzazione di una lista di richieste contenenti un Learning Agreement. La lista contiene il nome dello studente mittente e lo stato attuale del LA. La lista è cliccabile e rimanda alla pagina specifica della richiesta.
- request.html – pagina contenente l'anagrafica dello studente mittente della richiesta ed un riepilogo della documentazione allegata, oltre al Learning Agreement stesso. Contiene anche i riferimenti al Tutor Accademico ed Esterno associati. Nella stessa area è possibile approvare/rifiutare suddetto LA, oltre ad inviarlo al Tutor Esterno di riferimento per l'ultima compilazione.
- profile.html – visualizzazione del proprio profilo o di quello di un tutor/organizzazione esterna, se si sta visualizzando il proprio profilo è possibile la modifica dei dati dalla stessa pagina password inclusa.

External Tutor

- compileLAExternalTutor.html – compilazione della tabella C del Learning Agreement a seguito di selezione di un Learning Agreement inviato da uno studente. Possibile la modifica dei dati dalla stessa pagina se propria, password inclusa.
- viewrequest.html – visualizzazione di una lista di richieste contenenti un Learning Agreement. La lista contiene il nome dello studente mittente e lo stato attuale del LA. La lista è cliccabile e rimanda alla pagina specifica della richiesta.
- request.html – pagina contenente l'anagrafica dello studente mittente della richiesta ed un riepilogo della documentazione allegata, oltre al Learning Agreement stesso. Contiene anche i riferimenti al Tutor Accademico ed Esterno associati. Nella stessa area è possibile approvare/rifiutare suddetto LA.
- profile.html – visualizzazione del proprio profilo o di quello di un tutor/organizzazione esterna, se si sta visualizzando il proprio profilo è possibile la modifica dei dati dalla stessa pagina password inclusa.

Administrator

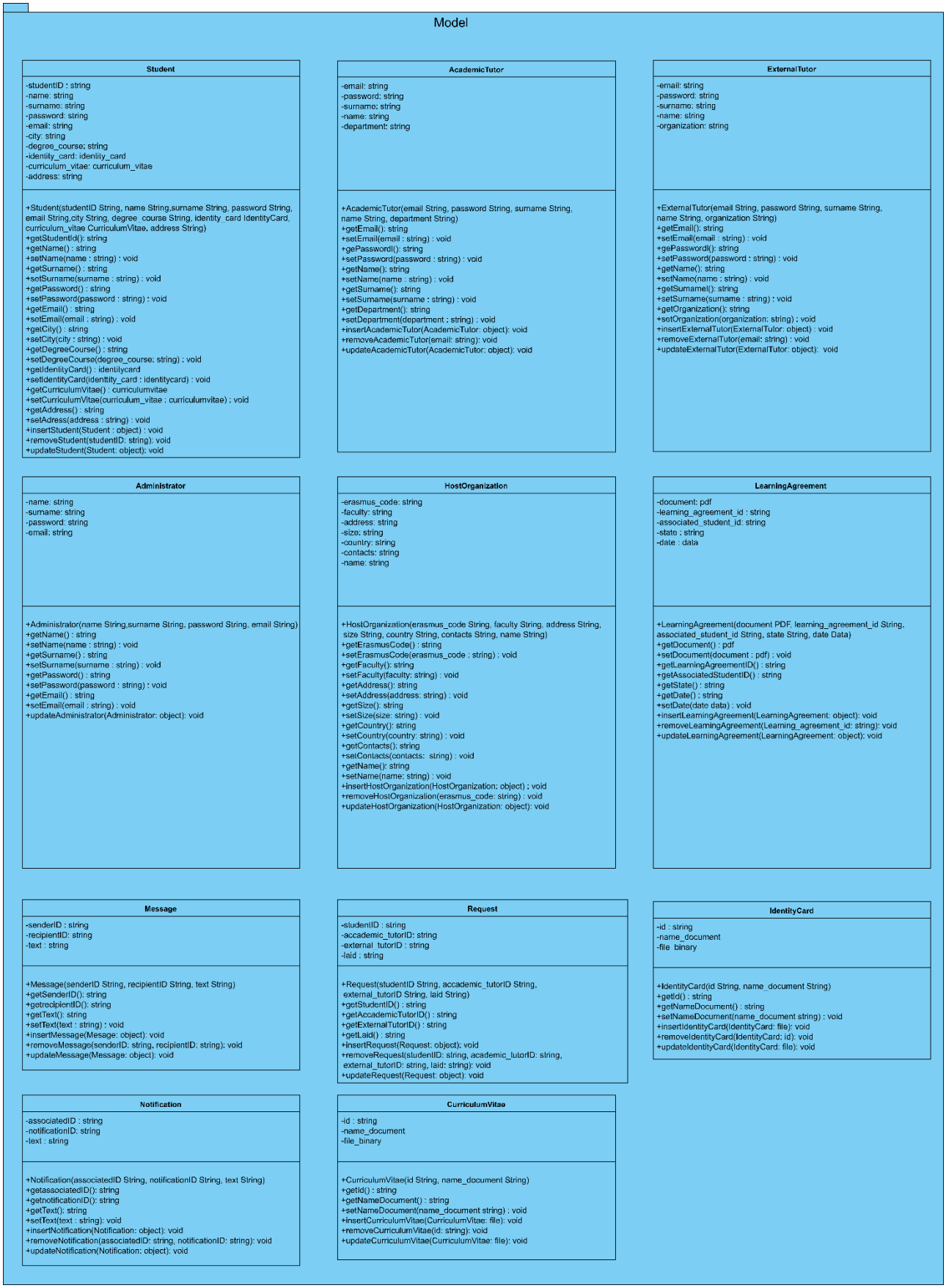
- instutor.html – permette l'inserimento di un tutor esterno.
- insorg.html – permette l'inserimento di un'organizzazione ospitante.
- editPassword.html – permette la modifica della password. Pagina utilizzabile solo dall'admin.

2.2 Model

Il package Model si occupa di fare da tramite tra l'applicazione e il database sottostante.

Ogni classe contenuta all'interno del pacchetto fornisce i metodi per accedere ai dati utili all'applicazione.

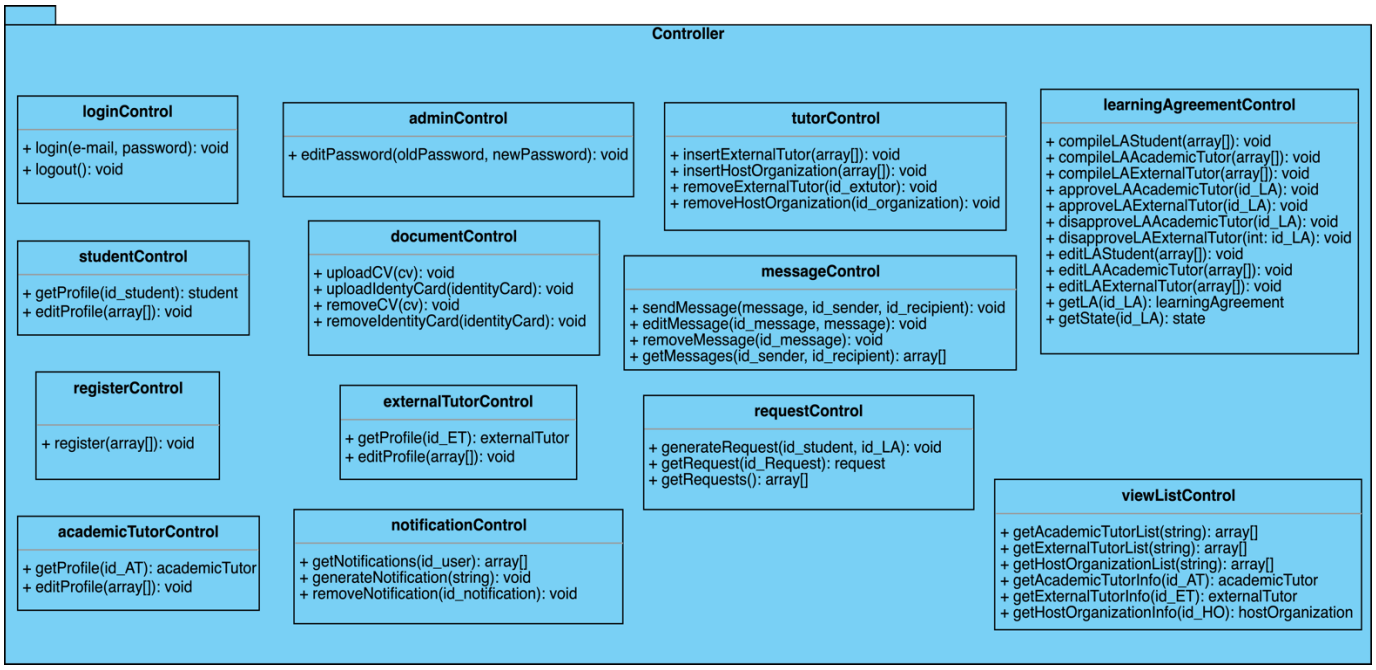
I moduli contenuti all'interno del package sono: Student, AcademicTutor, ExternalTutor, Administrator, LearningAgreement, HostOrganization, Notification, Request, Message, IdentityCard, CurriculumVitae.


Model
Student
-studentID : string
-name: string
-surname: string
-password: string
-email: string
-city: string
-degree_course: string
-identity_card: identity_card
-curriculum_vitae: curriculum_vitae
-address: string
+Student(studentID String, name String,surname String, password String, email String,city String, degree_course String, identity_card IdentityCard, curriculum_vitae CurriculumVitae, address String)
+getStudentId(): string
+getName() : string
+setName(name : string) : void
+getSurname() : string
+setSurname(surname : string) : void
+getPassword() : string
+setPassword(password : string) : void
+getEmail() : string
+setEmail(email : string) : void
+getCity() : string
+setCity(city : string) : void
+getDegreeCourse() : string
+setDegreeCourse(degree_course: string) : void
+getIdentityCard() : identitycard
+setIdentityCard(identity_card : identitycard) : void
+getCurriculumVitae() : curriculumvitae
+setCurriculumVitae(curriculum_vitae : curriculumvitae) : void
+getAddress() : string
+setAdress(address : string) : void
+insertStudent(Student : object) : void
+removeStudent(studentID: string): void
+updateStudent(Student: object): void
AcademicTutor
-email: string
-password: string
-surname: string
-name: string
-department: string
+AcademicTutor(email String, password String, surname String, name String, department String)
+getEmail(): string
+setEmail(email : string) : void
+gePasswordI(): string
+setPassword(password : string) : void
+getName(): string
+setName(name : string) : void
+getSurname(): string
+setSurname(surname : string) : void
+getDepartment(): string
+setDepartment(department : string) : void
+insertAcademicTutor(AcademicTutor: object): void
+removeAcademicTutor(email: string): void
+updateAcademicTutor(AcademicTutor: object): void
ExternalTutor
-email: string
-password: string
-surname: string
-name: string
-organization: string
+ExternalTutor(email String, password String, surname String, name String, organization String)
+getEmail(): string
+setEmail(email : string) : void
+gePasswordI(): string
+setPassword(password : string) : void
+getName(): string
+setName(name : string) : void
+getSurnameI(): string
+setSurname(surname : string) : void
+getOrganization(): string
+setOrganization(organization: string) : void
+insertExternalTutor(ExternalTutor: object) : void
+removeExternalTutor(email: string) : void
+updateExternalTutor(ExternalTutor: object): void
Administrator
-name: string
-surname: string
-password: string
-email: string
+Administrator(name String,surname String, password String, email String)
+getName() : string
+setName(name : string) : void
+getSurname() : string
+setSurname(surname : string) : void
+getPassword() : string
+setPassword(password : string) : void
+getEmail() : string
+setEmail(email : string) : void
+updateAdministrator(Administrator: object): void
HostOrganization
-erasmus_code: string
-faculty: string
-address: string
-size: string
-country: string
-contacts: string
-name: string
+HostOrganization(erasmus_code String, faculty String, address String, size String, country String, contacts String, name String)
+getErasmusCode() : string
+setErasmusCode(erasmus_code : string) : void
+getFaculty(): string
+setFaculty(faculty: string) : void
+getAddress(): string
+setAddress(address: string) : void
+getSize(): string
+setSize(size: string) : void
+getCountry(): string
+setCountry(country: string) : void
+getContacts(): string
+setContacts(contacts: string) : void
+getName(): string
+setName(name: string) : void
+insertHostOrganization(HostOrganization: object) : void
+removeHostOrganization(erasmus_code: string) : void
+updateHostOrganization(HostOrganization: object): void
LearningAgreement
-document: pdf
-learning_agreement_id : string
-associated_student_id: string
-state : string
-date : data
+LearningAgreement(document PDF, learning_agreement_id String, associated_student_id String, state String, date Data)
+getDocument() : pdf
+setDocument(document : pdf) : void
+getLearningAgreementID() : string
+getAssociatedStudentID() : string
+getState() : string
+getDate() : string
+setDate(date data) : void
+insertLearningAgreement(LearningAgreement: object): void
+removeLearningAgreement(Learning_agreement_id: string): void
+updateLearningAgreement(LearningAgreement: object): void
Message
-senderID : string
-recipientID: string
-text : string
+Message(senderID String, recipientID String, text String)
+getSenderID(): string
+getrecipientID(): string
+getText(): string
+setText(text : string) : void
+insertMessage(Mesage: object): void
+removeMessage(senderID: string, recipientID: string): void
+updateMessage(Message: object): void
Request
-studentID : string
-accademic_tutorID: string
-external_tutorID : string
-laid : string
+Request(studentID String, accademic_tutorID String, external_tutorID String, laid String)
+getStudentID() : string
+getAccademicTutorID() : string
+getExternalTutorID() : string
+getLaid() : string
+insertRequest(Request: object): void
+removeRequest(studentID: string, academic_tutorID: string, external_tutorID: string, laid: string): void
+updateRequest(Request: object): void
IdentityCard
-id : string
-name_document
-file_binary
+IdentityCard(id String, name_document String)
+getId() : string
+getNameDocument() : string
+setNameDocument(name_document string) : void
+insertIdentityCard(IdentityCard: file): void
+removeIdentityCard(IdentityCard: id): void
+updateIdentityCard(IdentityCard: file): void
Notification
-associatedID : string
-notificationID: string
-text : string
+Notification(associatedID String, notificationID String, text String)
+getassociatedID(): string
+getnotificationID(): string
+getText(): string
+setText(text : string): void
+insertNotification(Notification: object): void
+removeNotification(associatedID: string, notificationID: string): void
+updateNotification(Notification: object): void
CurriculumVitae
-id : string
-name_document
-file_binary
+CurriculumVitae(id String, name_document String)
+getId() : string
+getNameDocument() : string
+setNameDocument(name_document string) : void
+insertCurriculumVitae(CurriculumVitae: file): void
+removeCurriculumVitae(id: string): void
+updateCurriculumVitae(CurriculumVitae: file): void

2.3 Controller

Il package Controller riceve, tramite il package View, i comandi dell'utente. Esso è formato dai moduli:

- registerControl (si occupa della gestione della registrazione, e comunica direttamente con i model Student, academicTutor e le view signup.html).
- loginControl (si occupa della gestione del login e del logout, e comunica con i model Student, academicTutor, externalTutor, administrator, e con le view login.html).
- studentControl (si occupa della gestione del profilo dello studente, e comunica con il model Student e con la view profile.html).
- academicTutorControl (si occupa della gestione del profilo del tutor accademico, e comunica con il model AcademicTutor e con la view profile.html).
- externalTutorControl (si occupa della gestione del profilo del tutor esterno, e comunica con il model ExternalTutor e con la view profile.html).
- adminControl (si occupa della gestione del profilo dell'amministratore, e comunica, con il model Administrator con le view editPassword.html).
- documentControl (si occupa della gestione dei documenti dello studente, e comunica con i model Student, IdentityCard, CurriculumVitae, e con la view dochandler.html).
- tutorControl (si occupa della gestione dei tutor esterni e delle organizzazioni ospitanti da parte dell'amministratore, e comunica con il model Administrator e con le view instutor.html, insorg.html, viewinfo.html).
- messageControl (si occupa della gestione dei messaggi, e comunica con il model Message e con la view index.html).
- notificationControl (si occupa della gestione delle notifiche, e comunica con il model Notification e con la view index.html).
- learningAgreementControl (si occupa della gestione del Learning Agreement, e comunica con i model Student, AcademicTutor, ExternalTutor e con le view compileLAStudent.html, editLA.html, viewLA.html, compileLAAcademicTutor.html, compileLAExternalTutor.html).
- requestControl (si occupa della gestione delle richieste, e comunica con i model AcademicTutor, ExternalTutor e con le view viewRequest.html, request.html).
- viewListControl (si occupa della gestione delle liste dei tutor e delle organizzazioni ospitanti, e comunica con i model AcademicTutor, ExternalTutor e con le view viewtutor.html, vieworg.html e viewinfo.html).


Controller
loginControl
+ login(e-mail, password): void
+ logout(): void
adminControl
+ editPassword(oldPassword, newPassword): void
tutorControl
+ insertExternalTutor(array[]): void
+ insertHostOrganization(array[]): void
+ removeExternalTutor(id_extutor): void
+ removeHostOrganization(id_organization): void
learningAgreementControl
+ compileLAStudent(array[]): void
+ compileLAAcademicTutor(array[]): void
+ compileLAExternalTutor(array[]): void
+ approveLAAcademicTutor(id_LA): void
+ approveLAExternalTutor(id_LA): void
+ disapproveLAAcademicTutor(id_LA): void
+ disapproveLAExternalTutor(int: id_LA): void
+ editLAStudent(array[]): void
+ editLAAcademicTutor(array[]): void
+ editLAExternalTutor(array[]): void
+ getLA(id_LA): learningAgreement
+ getState(id_LA): state
studentControl
+ getProfile(id_student): student
+ editProfile(array[]): void
documentControl
+ uploadCV(cv): void
+ uploadIdentyCard(identityCard): void
+ removeCV(cv): void
+ removeIdentityCard(identityCard): void
messageControl
+ sendMessage(message, id_sender, id_recipient): void
+ editMessage(id_message, message): void
+ removeMessage(id_message): void
+ getMessages(id_sender, id_recipient): array[]
registerControl
+ register(array[]): void
externalTutorControl
+ getProfile(id_ET): externalTutor
+ editProfile(array[]): void
requestControl
+ generateRequest(id_student, id_LA): void
+ getRequest(id_Request): request
+ getRequests(): array[]
viewListControl
+ getAcademicTutorList(string): array[]
+ getExternalTutorList(string): array[]
+ getHostOrganizationList(string): array[]
+ getAcademicTutorInfo(id_AT): academicTutor
+ getExternalTutorInfo(id_ET): externalTutor
+ getHostOrganizationInfo(id_HO): hostOrganization
academicTutorControl
+ getProfile(id_AT): academicTutor
+ editProfile(array[]): void
notificationControl
+ getNotifications(id_user): array[]
+ generateNotification(string): void
+ removeNotification(id_notification): void

3. Class Interfaces

Nome classe	loginControl
Descrizione	Questa classe gestisce le funzionalità di autenticazione e di uscita dal sistema.
Pre-condizione	**context** loginControl: login(String e-mail, String password); **context** loginControl: logout();
Post-condizione	**context** loginControl: login(String e-mail, String password); **context** loginControl: logout();
Invarianti	**context** loginControl: login(String e-mail, String password); **inv:** e-mail != null && password != null

Nome classe	registerControl
Descrizione	Questa classe gestisce le funzionalità relative al processo di registrazione.
Pre-condizione	**context** registerControl: register(Array[] registerField);
Post-condizione	**context** registerControl: register(Array[] registerField); **post:** collection<User>(size+1)
Invarianti	**context** registerControl: register(Array[] registerField); **inv:** registerField != null

Nome classe	studentControl
Descrizione	Questa classe gestisce le funzionalità del profilo dello studente.
Pre-condizione	**context** studentControl: getProfile(id_student); **context** studentControl: editProfile(Array[] registerField);
Post-condizione	**context** studentControl: editProfile(Array[] registerField); **post:** collection<Student>(alter)
Invarianti	**context** studentControl: getProfile(id_student); **inv:** id_student != null **context** studentControl: editProfile(Array[] registerField); **inv:** registerField != null

Nome classe	academicTutorControl
Descrizione	Questa classe gestisce le funzionalità del profilo del tutor accademico.
Pre-condizione	**context** academicTutorControl: getProfile(id_AT); **context** academicTutorControl: editProfile(Array[] registerField);
Post-condizione	**context** academicTutorControl: editProfile(Array[] registerField); **post:** collection<AcademicTutor>(alter)
Invarianti	**context** academicTutorControl: getProfile(id_AT); **inv:** id_AT != null **context** academicTutorControl: editProfile(Array[] registerField); **inv:** registerField != null

Nome classe	externalTutorControl
Descrizione	Questa classe gestisce le funzionalità del profilo del tutor esterno.

Pre-condizione	**context** externalTutorControl: getProfile(id_ET); **context** externalTutorControl: editProfile(Array[] registerField);
Post-condizione	**context** externalTutorControl: editProfile(Array[] registerField); **post:** collection<ExternalTutor>(alter)
Invarianti	**Context** externalTutorControl: getProfile(id_ET) **Inv:** id_ET != null

Nome classe	**adminControl**
Descrizione	Questa classe gestisce le funzionalità del profilo dell'amministratore.
Pre-condizione	**context** adminControl: editPassword(oldPassword, newPassword);
Post-condizione	**context** adminControl: editPassword(oldPassword, newPassword); **post:** collection<Admin>(alter)
Invarianti	**context** adminControl: editPassword(oldPassword, newPassword); **inv:** oldPassword != null && newPassword.lenght >=9

Nome classe	**documentControl**
Descrizione	Questa classe gestisce tutte le funzionalità relative ai documenti dello studente.
Pre-condizione	**context** documentControl: uploadCV(cv); **context** documentControl: uploadIdentityCard(identityCard); **context** documentControl: removeCV(cv); **context** documentControl: removeIdentityCard(identityCard);
Post-condizione	**context** documentControl: uploadCV(cv); **post:** collection<Student>(alter)

	context documentControl: uploadIdentityCard(identityCard); **post:** collection<Student>(alter) **context** documentControl: removeCV(cv); **post:** collection<Student>(alter) **context** documentControl: removeIdentityCard(identityCard); **post:** collection<Student>(alter)
Invarianti	**context** documentControl: uploadCV(cv); **inv:** cv != null **context** documentControl: uploadIdentityCard(identityCard); **inv:** IdentityCard != null **context** documentControl: removeCV(cv); **inv:** cv != null **context** documentControl: removeIdentityCard(identityCard); **inv:** IdentityCard != null

Nome classe	tutorControl
Descrizione	Questa classe si occupa delle funzionalità per la gestione dei tutor esterni e delle aziende ospitanti da parte dell'amministratore.
Pre-condizione	**context** tutorControl: insertExternalTutor(Array[] registerField); **context** tutorControl: insertHostOrganization(Array[] registerField); **context** tutorControl: removeExternalTutor(); **context** tutorControl: removeHostOrganization();

Post-condizione	**context** tutorControl: insertExternalTutor(); **post:** collection<ExternalTutor>(size+1) **context** tutorControl: insertHostOrganization(); **post:** collection<HostOrganization>(size+1) **context** tutorControl: removeExternalTutor(); **post:** collection<ExternalTutor>(size-1) **context** tutorControl: removeHostOrganization(); **post:** collection<HostOrganization>(size-1)
Invarianti	**context** tutorControl: insertExternalTutor(); **inv:** ExternalTutor != null **context** tutorControl: insertHostOrganization(); **inv:** HostOrganization != null **context** tutorControl: removeExternalTutor(); **inv:** ExternalTutor != null **context** tutorControl: removeHostOrganization(); **inv:** HostOrganization != null

Nome classe	**messageControl**
Descrizione	Questa classe si occupa della gestione dei messaggi.
Pre-condizione	**context** messageControl: sendMessage(message, id_sender, id_recipient); **context** messageControl: editMessage(id_message, message); **context** messageControl: removeMessage(id_message); **context** messagerControl: getMessage(id_sender, id_recipient);
Post-condizione	**context** messageControl: sendMessage(message, id_sender, id_recipient); **post:** collection<Message>(size+1) **context** messageControl: editMessage(id_message, message);

	post: collection<Message>(alter) **context** messageControl: removeMessage(id); **post:** collection<Message>(size-1) **context** messagerControl: getMessage(id_sender, id_recipient);
Invarianti	**context** messageControl: sendMessage(message, id_sender, id_recipient); **inv:** message != null && 1<=message.lenght <=100 && id_sender != null, id_recipient != null **context** messageControl: editMessage(id_message, message); **inv:** 1<=message.lenght <=100 && message != null **context** messageControl: removeMessage(id); **inv:** id != null **context** messagerControl: getMessage(id_sender, id_recipient); **inv:** id_sender != null && id_recipient != null

Nome classe	**requestControl**
Descrizione	Questa classe si occupa della gestione delle richieste.
Pre-condizione	**context** requestControl: generateRequest(id_student, id_LA); **context** requestControl: getRequest(id_Request); **context** requestControl: getRequests();
Post-condizione	**context** requestControl: generateRequest(id_LA); **post:** collection<Request>(size+1)
Invarianti	**context** requestControl: generateRequest(id_student, id_LA); **inv:** id_student != null && id_LA != null

Nome classe	notificationControl
Descrizione	Questa classe si occupa della gestione delle notifiche.
Pre-condizione	**context** notificationControl: getNotification(id_user); **context** notificationControl: generateNotification(text); **context** notificationControl: removeNotification();
Post-condizione	**context** notificationControl: generateNotification(string); **post:** collection<Notification>(size+1) **context** notificationControl: removeNotification(); **post:** collection<Notification>(size-1)
Invarianti	**context** notificationControl: getNotification(id_user); **inv:** id_user!=null **context** notificationControl: generateNotification(text); **inv:** text!=null

Nome classe	LearningAgreementControl
Descrizione	Questa classe si occupa della gestione del Learning Agreement.
Pre-condizione	**context** LearningAgreementControl: compileLAStudent (Array[] fields); **context** LearningAgreementControl: compileLAAcademicTutor(Array[] fields); **context** LearningAgreementControl: compileLAExternalTutor (Array[] fields); **context** LearningAgreementControl: approveLAAcademicTutor(int id_LA); **context** LearningAgreementControl: approveLAExternalTutor(int id_LA); **context** editLAStudent(Array[] fields); **context** editLAAcademicTutor(Array[] fields); **context** editLAExternalTutor(Array[] fields); **context** getLA(int id_LA);

	context getState(int id_LA);
Post-condizione	**context** LearningAgreementControl: compileLAStudent (Array[] fields); **post:** collection<LearningAgreement>(size+1) **context** LearningAgreementControl: compileLAAcademicTutor(Array[] fields); **post:** collection<LearningAgreement>(alter) **context** LearningAgreementControl: compileLAExternalTutor (Array[] fields); **post:** collection<LearningAgreement>(alter) **context** editLAStudent(Array[] fields); **post:** collection<LearningAgreement>(alter) **context** editLAAcademicTutor(Array[] fields); **post:** collection<LearningAgreement>(alter) **context** editLAExternalTutor(Array[] fields); **post:** collection<LearningAgreement>(alter)
Invarianti	**context** LearningAgreementControl: compileLAStudent (Array[] fields); **inv:** fields != null **context** LearningAgreementControl: compileLAAcademicTutor(Array[] fields); **inv:** fields != null **context** LearningAgreementControl: compileLAExternalTutor (Array[] fields); **inv:** fields != null **context** editLAStudent(Array[] fields); **inv:** fields != null **context** editLAAcademicTutor(Array[] fields); **inv:** fields != null **context** editLAExternalTutor(Array[] fields);

	inv: fields != null

Nome classe	viewListControl
Descrizione	Questa classe si occupa della gestione delle notifiche.
Pre-condizione	**context** viewListControl: getAcademicTutorList(String type); **context** viewListControl: getExternalTutorList(String type); **context** viewListControl: getHostOrganizationList(String type); **context** viewListControl: getAcademicTutorInfo(id_AT); **context** viewListControl: getETInfo(id_ET); **context** viewListControl: getHOInfo(id_HO);
Post-condizione	
Invarianti	**context** viewListControl: getAcademicTutorList(String type); **inv:** type!=null **context** viewListControl: getExternalTutorList(String type); **inv:** type!=null **context** viewListControl: getHostOrganizationList(String type); **inv:** type!=null **context** viewListControl: getAcademicTutorInfo(id_AT); **inv:** id_AT!=null **context** viewListControl: getETInfo(id_ET); **inv:** tid_ET!=null **context** viewListControl: getHOInfo(id_HO); **inv:** id_HO!=null

4. Design Pattern Con Class Diagram

4.1 Document Versioning Pattern:

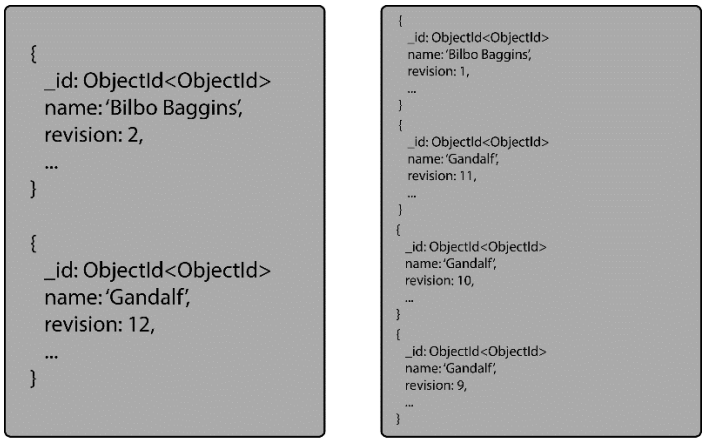
Nome e classificazione: Document Versioning Pattern.

Scopo: il Document Versioning Pattern consente di conservare versioni precedenti dei documenti in MongoDB invece di coinvolgere un ulteriore management system.

Applicabilità: il Document Versioning Pattern è indicato nei casi in cui:

- Ogni documento non ha troppe versioni.
- Non ci sono troppi documenti di cui mantenere le versioni.
- La maggior parte delle query vengono fatte sulla versione più recente del documento.

Struttura:



current_policies collection *policy_revisions* collection

Partecipanti:

- **current_policies** contiene le versioni attuali dei documenti, quando viene apportata una modifica la versione presente in current_policies viene copiata in policy_revisions, dopodiché viene aggiornata alla nuova versione.
- **policy_revisions** contiene tutte le versioni precedenti dei documenti.

Conseguenze:

Per tenere traccia delle varie revisioni, viene aggiunto ad ogni documento un campo che ne indichi la versione. Il DB avrà a questo punto due collection: una con le versioni più recenti (quelle che saranno interrogate più frequentemente) ed un'altra che contiene le versioni precedenti.

Utilizzo: all'interno del nostro software verrà utilizzato per conservare e rendere accessibili le versioni precedenti del Learning Agreement.

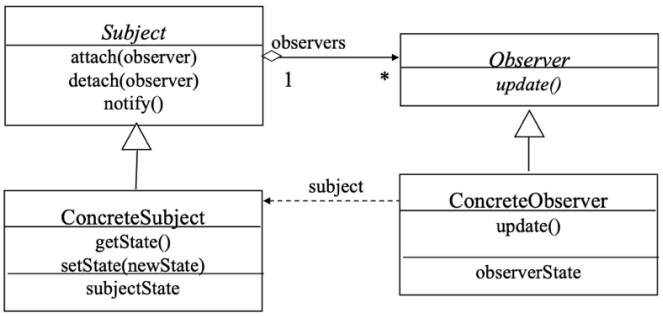
4.2 Observer pattern:

Nome e classificazione: Observer pattern. Fa parte dei Behavioral Pattern.

Scopo: gli Observer Pattern si occupano di algoritmi e dell'assegnazione delle responsabilità tra oggetti che collaborano (Chi fa che cosa?). Caratterizzano flussi di controllo complessi, difficili da seguire a run time.

Applicabilità: definisce una dipendenza uno-a-molti tra gli oggetti in modo tale che quando un oggetto cambia stato, tutte le sue dipendenze vengano notificate e aggiornate automaticamente. Viene utilizzato per il mantenimento della coerenza in tutti gli stati ridondanti. Inoltre, viene utilizzato per l'ottimizzazione delle modifiche batch per mantenere la coerenza.

Struttura:



Partecipanti: in generale l'Observer pattern impiega le seguenti classi:

- Soggetto: una classe che fornisce interfacce per registrare o rimuovere gli observer.
- Soggetto Concreto: classe che contiene l'attributo "subjectState", il quale descrive lo stato del soggetto.
- Observer: classe che definisce un'interfaccia per tutti gli observer per ricevere le notifiche dal soggetto. È utilizzata come classe astratta per implementare i veri observer, ossia i ConcreteObserver.
- ConcreteObserver: questa classe mantiene un riferimento "subject" al Soggetto Concreto, per ricevere lo stato quando avviene una notifica. Inoltre, alla classe appartiene l'attributo "observerState", il quale contiene lo stato del ConcreteObserver.

Conseguenze: l'Observer pattern, garantisce che quando un oggetto cambia stato, un numero aperto di oggetti dipendenti vengano aggiornati automaticamente, oltre al fatto che deve essere possibile che un oggetto possa notificare un numero aperto di altri oggetti. L'Observer pattern può causare però perdite di memoria, noto come "problema dell'ascoltatore scaduto", perché nell'implementazione di base si

richiede sia la registrazione esplicita, sia la cancellazione esplicita, come nel Dispose Pattern. Questo perché il soggetto contiene forti riferimenti agli observer, mantenendoli in vita.

Utilizzo: all'interno del nostro software verrà utilizzato per gestire le dipendenze tra le varie versioni del Learning Agreement e quelle del Learning Agreement con le notifiche correlate ad esso. Ci sarà utile, inoltre, anche per la gestione del sistema di messaggistica.

5. Glossario

MongoDB: è un DBMS non relazionale, orientato ai documenti. Classificato come un database di tipo NoSQL, MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali in favore di documenti in stile JSON con schema dinamico.

DBMS (Database Management System): è un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database, per questo detto anche "gestore o motore del database", è ospitato su architettura hardware dedicata oppure su semplice computer.

NoSQL: i database NoSQL sono appositamente realizzati per modelli di dati specifici e hanno schemi flessibili per creare applicazioni moderne. Si discostano dai Database di tipo relazionale, infatti l'espressione "NoSQL" fa riferimento al linguaggio SQL, che è il più comune linguaggio di interrogazione dei dati nei database relazionali, qui preso a simbolo dell'intero paradigma relazionale.

JSON (JavaScript Object Notation): è un semplice formato per lo scambio di dati**.** È basato sul linguaggio JavaScript, ma ne è indipendente. Viene usato in AJAX come alternativa a XML/XSLT.

Node.js: è una runtime di JavaScript Open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript, costruita sul motore JavaScript V8 di Google Chrome. Node.js consente di utilizzare JavaScript anche per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al Browser dell'utente.

Modulo: sono file con estensione .js che contengono codice JavaScript, Node.js viene fornito con una serie di moduli principali che implementano funzionalità di base.

Bootstrap: è una raccolta di strumenti free per la creazione di siti e applicazioni per il Web. Essa contiene modelli di progettazione basati su HTML e CSS, sia per la tipografia, che per le varie componenti dell'interfaccia, come moduli, pulsanti e navigazione, così come alcune estensioni opzionali di JavaScript.

CamelCase: la notazione a cammello, o in inglese *CamelCase,* è la pratica nata durante gli anni 70 di scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole. Si

può distinguere in un lowerCamelCase, in cui la prima lettera della prima parola viene lasciata minuscola, o in UpperCamelCase, in cui la prima lettera della prima parola è maiuscola.

JavaScript: è un linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client (recentemente esteso anche al lato server) per la creazione, in siti web e applicazioni web, di effetti dinamici interattivi tramite funzioni di script invocate da eventi innescati a loro volta in vari modi dall'utente sulla pagina web in uso.

HTML (HyperText Markup Language): è un linguaggio di markup nato per la formattazione e impaginazione di documenti ipertestuali disponibili nel web 1.0, oggi è utilizzato principalmente per il disaccoppiamento della struttura logica di una pagina web.

CSS (Cascading Style Sheets): è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML ad esempio nei siti web e relative pagine web.

Trade-off: il Trade-off è una situazione che implica una scelta tra due possibilità, in cui la perdita di valore di una costituisce un aumento di valore in un'altra.