

RAD Requirement Analysis Document

YouLearn Platform Project

UNIVERSITÁ DEGLI STUDI DI SALERNO

Ingegneria del Software

OBJECT DESIGN DOCUMENT



**ANNO ACCADEMICO 2018/2019**a

**TOP MANAGER:**

|  |
| --- |
| **Nome** |
| **Prof. Andrea De Lucia** |

**PARTECIPANTI:**

|  |  |
| --- | --- |
| **Nome** | **Matricola** |
| Mario Sessa | 0512104650 |
| Luigi Crisci | 0512104740 |
| Pasquale Ambrosio | 0512104704 |

**HISTORY:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Cambiamenti** | **Autori** |
| 26/12/2018 | 1.0 | Creazione capitolo 1. | Mario Sessa |
| 27/12/2018 | 1.1 | Creazione capitolo 2. | Mario Sessa |
|  |  |  |  |

1. Introduzione

**1.1 Object Design Trade-offs**

Dopo la realizzazione dei documenti RAD e SDD abbiamo descritto in linea di massima quello che sarà il nostro sistema e gli obbiettivi da seguire, tralasciando gli aspetti implementativi. Il seguente documento ha lo scopo di produrre un modello capace di integrare in modo coerente e preciso tutte le funzionalità̀ individuate nelle fasi precedenti. In particolare, definisce le interfacce delle classi, le operazioni, i tipi, gli argomenti e le signature dei sottosistemi definiti nel System Design. Inoltre, sono specificati i trade-off e le linee guida.

**Comprensibilità vs Tempo**

Il codice deve essere il più chiaro possibile, ogni componente deve essere accompagnato da un commento in grado di descrivere quali operazioni si stanno implementando. Questa forma di comprensibilità del codice porterà dei rallentamenti in fase implementativa e di testing andando a creare, però, vantaggi sulla comprensione globale del sistema e delle sue componenti.

**Prestazioni vs Costi**

Non avendo finanziamenti esterni, si utilizzeranno delle tecnologie open-source in grado di gestire il sistema in maniera gratuita. Nello specifico, verrà utilizzato un database relazionale come repository centrale per i dati gestiti dal sistema e un web server monolitico per la gestione dell’interazione con gli utenti.

**Interfaccia vs Usabilità**

L’interfaccia verrà gestita in modo tale da poter essere il più semplice ed intuitiva possibile, attraverso l’uso di form e bottoni di facile comprensione per l’utente finale.

**Sicurezza vs Efficienza**

Il sistema si baserà prevalentemente sulla gestione della sicurezza per evitare accessi non autorizzati cosi da proteggere informazioni personali quali e-mail, password e carte di credito.

**1.2 Linee guida per la documentazione delle interfacce**

Gli sviluppatori dovranno seguire le corrispondenti linee guida durante la fase implementativa del progetto:

**1.2.1 Naming Convention**

I nomi utilizzati per la rappresentazione dei concetti principali, delle funzionalità e delle componenti generiche del sisterma devono rispettare le seguenti condizioni:

1. I **nomi** devono essere:
   1. Appartenenti alla lingua italiana, se possibile.
   2. Di lunghezza medio-breve
   3. Non sostituiti da acronimi o abbreviazioni di alcun genere
   4. Composti da caratteri compresi in [0-9, a-z, A-Z]
2. Le **variabili** devono:
   1. Rispettare la Camel Notation
   2. Iniziare con lettere minuscole
   3. Essere composti da caratteri compresi in [0-9, a-z,A-Z]

È possibile far iniziare le variabili statiche con “\_” cosi da contraddistinguerle dal resto delle variabili presenti all’interno del codice. Questa distinzione è utile per evitare errori sull’uso improprio di tali forme di variabili.

1. Le **classi e le interfacce** devono:
   1. Rispettare la Camel Notation
   2. Iniziare con la lettera grande
   3. Concludersi con il tipo di elemento che rappresentano:

Esempio:

public class UtenteBean {} /\* Per una classe di tipo Bean \*/

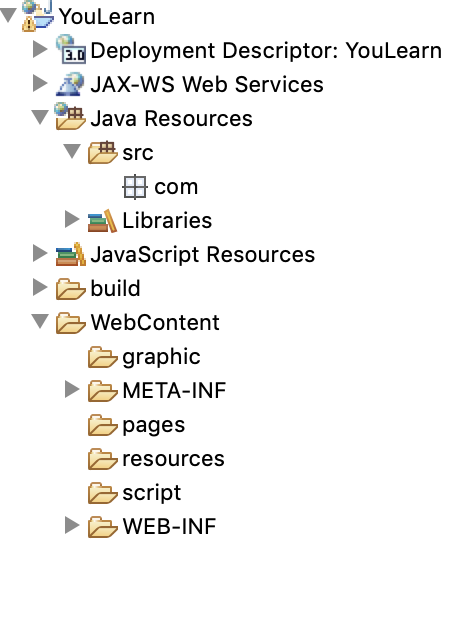
public class TestClass {} /\* Per una classe generica \*/

public interface DataInterface /\* Per un’interfaccia \*/

1. Le variabili **costanti** devono:
   1. Utilizzare solamente caratteri maiuscoli
   2. Separare i vari nomi che la compongono da “\_”
   3. Evitare di iniziare con “\_”
   4. Contenere solamente caratteri [a-z, A-Z, 0-9]
   5. Essere di lunghezza medio-breve
2. I **pacchetti** devono:
   1. Contenere solamente caratteri minuscoli
   2. Contenere solamente caratteri a-z
   3. Di semantica affine con gli elementi di cui è composto
3. I **metodi** devono:
   1. Iniziare con lettere minuscole
   2. Rispettare la Camel Notation
   3. Evitare di iniziare con GET o SET se non si trattano di metodi setting o getting della classe corrispondente.
   4. Contenere solamente caratteri [a-z, A-Z]

**1.2.2 Gestione file**

Poiché il sistema verrà implementato tramite l’uso di pagine Servlet, JSP, pagine Java, Javascript, XML e file CSS3. Ognuno di essi verrà posto nel folder corrispondente secondo la seguente condizioni:



• Tutte le Servlet verranno divise in pacchetti in base al sottosistema a cui appartengono. Tali pacchetti sono contenuti tutti all’interno del package “com” dentro la cartella “src” di “Java Resources”.

• Tutte le JSP, file HTML o XML verranno inserite nella cartella “pages” situata all’interno di “WebContent”

• Tutti i file Javascript verranno inseriti dentro la cartella “script” situata all’interno di “WebContent”

• Tutti i file immagini, video, gif e altre risorse di tale genere verranno inserite all’interno della cartella “resources” collocata in “WebContent”. Verrà eseguita anche una suddivisione in sottocartelle a secondo della tipologia di risorsa interessata.

• Tutti i file CSS verranno inseriti all’interno della cartella “graphic” situata all’interno di “WebContent”.

**1.2.2 Gestione formattazione del codice**

Per rispettare i criteri di comprensibilità definiti nella sottosezione precedente, bisogna approcciarsi con particolare cura sull’indentazione del codice e su come alcuni elementi del codice devono essere formattati. Di seguito verranno descritti alcuni esempi per dare una visione generale al programmatore su come si vuole indentare il codice.

**1.2.2.1 Gestione codice HTML e XML**

<html>

<head>

<head>

<body>

<div>

Contenuto DIV

<div>

<body>

<html>

Il codice HTML deve essere indentato in maniera tale da poter mantenere sulla stessa colonna il TAG di apertura e di chiusura. Inoltre, il contenuto di un TAG si distanzierà dalle clausole dell’elemento in cui è contenuto di una distanza pari ad 1 TAB.

I Tag che non hanno una clausola di chiusura seguiranno solamente la seconda condizione.

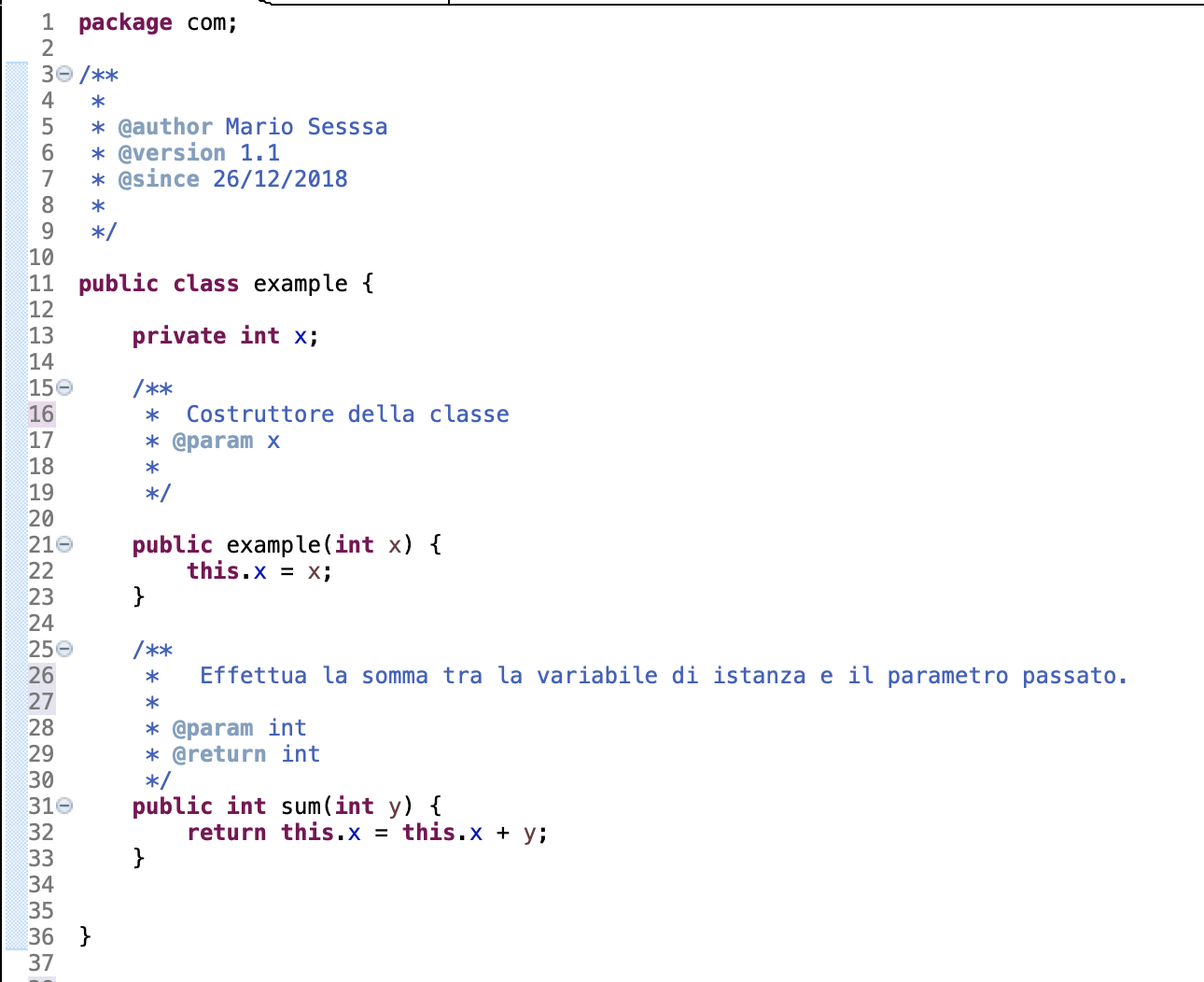
**1.2.2.2 Gestione codice classi Java e Servlet.**

All’interno di questa sezione ci soffermeremo sulla corretta formattazione delle classi Java, delle Servlet e dell’uso di codice Javadoc corrispondente.

Le classi **Java** devono seguire le seguenti condizioni:

1. Il codice Javadoc deve essere utilizzato per la descrizione di:
2. Classi
3. Metodi
4. Interfacce
5. I metodi, le classi interne e le variabili devono essere indentati in colonne successive a quella della classe o del metodo in cui sono contenute.

Un **esempio** più pratico del corretto formato lo vediamo nel frammento di codice seguente:



Il codice Javadoc di ogni classe deve contenere le clausole @author, @versione e @since. Mentre ogni metodo deve contenere obbligatoriamente una descrizione delle operazioni o della funzionalità che esegue e può richiamare clausole come @param e @return.

Il formato delle Servlet, invece, deve:

1. Contenere un costruttore, anche se vuoto
2. Contenere i metodi doGet() e doPost()
3. Contenere il codice Javadoc per la classe e per i metodi doGet() e doPost() in modo da definire lo scope della Servlet e le operazioni dei due metodi.

Un **esempio** più pratico del corretto formato lo vediamo nel frammento di codice seguente:



È possibile aggiungere qualsiasi clausola si voglia all’interno del codice Javadoc della classe e dei metodi. Inoltre, è possibile anche implementare metodi o funzioni proprie della classe in modo da rendere le sue operazioni più modulari.

**1.3 Definizioni, acronimi e abbreviazioni**

**Acronimi:**

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

**Abbreviazioni:**

* **DB:** Database
* **DBMS:** Database Management System

**Definizioni:**

* **Servlet:** Classi ed oggetti Java per la gestione di operazioni su un Web Server

**1.4 Riferimenti**

Il contesto è ripreso dal **RAD** e dall’ **SDD** del progetto YouLearn.

È stato anche usato come riferimento il libro:

**Object-Oriented Software Engineering:**   
**Using UML, Patterns, and Java, 3rd Edition**   
Prentice Hall, Upper Saddle River, NJ, September 25, 2009.

Infine, sono stati usati dei materiali di supporto visionabili al link: <https://wwwbruegge.in.tum.de/lehrstuhl_1/component/content/article/217-OOSE>

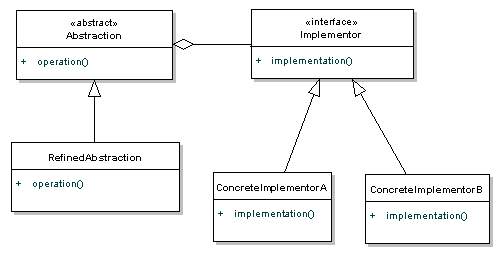
1. Design Pattern

**2.1 Design pattern utilizzati globalmente**

I design pattern scelti all’interno di YouLearn si basano sul concetto di creare un’implementazione del sistema modulare e indipendente nelle sue parti cosi da consentire un approccio evolutivo per modifiche future sullo stile architetturale Client-Server che adottiamo e che comporrà l’architettura del nostro sistema.

All’interno del nostro sistema, specifichiamo due design pattern fondamentali che si utilizzeranno all’interno del progetto in maniera globale

* **Bridge:** Design pattern strutturale in grado di poter separare le composizioni di una classe con le sue implementazioni. Questo approccio verrà utilizzato all’interno di classi Java utilizzati all’interno del progetto cosi da garantire massima modularità e facilità di implementazione.

****

* **Singleton:** Design pattern creazionale in grado di generare, per ogni classe di tipo Manager, una singola istanza utilizzata per tutte le attività collegate.

****

1. Package

*The second section of the ODD, Packages, describes the decomposition of subsystems into packages and the file organization of the code. This includes an overview of each package, its dependencies with other packages, and its expected usage.*

1. Class interfaces

*The third section, Class interfaces, describes the classes and their public interfaces. This includes an overview of each class, its dependencies with other classes and packages, its public attributes, operations, and the exceptions they can raise.*

1. Glossario

**Design Pattern:** Soluzione generale per una classe di problemi specifici presenti durante la progettazione di un sistema software.

**Astrazioni:** Concetto che, in informatica, astrae l’implementazione per concentrarsi sulla progettazione strutturale di un oggetto.

**Modularità:** Progettazione di un sistema basato sullo sviluppo delle sue attività in componenti indipendenti collegate tra di loro.