

# Report Elaborato APT

A.A. 2019/2020

---

Advanced Programming Techniques

Liuzzo Antonino Mauro, Nesi Davide

Università degli Studi di Firenze

<b>Introduzione</b>	<b>1</b>
<b>Descrizione del Sistema</b>	<b>1</b>
<b>Tecniche, Tecnologie e Framework applicati</b>	<b>2</b>
<b>Progettazione del Sistema</b>	<b>4</b>
Descrizione delle Funzionalità del Sistema	4
Creazione Profilo Studente	4
Rimozione Profilo Studente	4
Inserimento Corso	4
Rimozione Corso	4
Architettura del Sistema e Diagramma delle Classi	4
View	5
Controller	5
Service	5
Repository	5
Model	6
<b>Sviluppo e Testing</b>	<b>8</b>
Unit Testing	8
UI Testing	9
GUI Test	9
CLI Test	10
Integration Testing	12
E2E Testing	12
GUI Test	13
CLI Test	13
Virtualizzazione tramite Docker	13
Build Locale e su Server CI (i.e. Maven e Travis CI)	14
Utilizzo di Git e GitHub	15
Utilizzo delle funzionalità avanzata di MongoDB	15
Code Quality con SonarQube/SonarCloud	16
<b>Link di Riferimento</b>	<b>14</b>

## Introduzione

La seguente relazione descrive le varie fasi che compongono lo sviluppo di una piccola app Java (My Career), che gestisce le iscrizioni ai corsi da parte degli studenti. L'applicazione è stata sviluppata tramite l'utilizzo delle tecniche del TDD (un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test<sup>1</sup>), la Build Automation e, in generale, le tecniche e le tecnologie illustrate in [Test-Driven Development, Build Automation, Continuous Integration](#)<sup>2</sup>.

## Descrizione del Sistema

Il sistema da progettare dovrà svolgere la funzione di gestore di piani di studio, e quindi dovrà consentire all'utente di inserire e rimuovere i profili degli studenti e di gestire i loro piani di studio corsi del suo piano di studi. Il sistema dovrà quindi fornire all'utente le funzionalità necessarie allo svolgimento delle seguenti operazioni:

- Inserimento di un profilo di uno studente
- Inserimento di un corso legato ad un profilo studente
- Rimozione di un corso
- Rimozione di un profilo studente
- Visualizzazione corsi di uno studente

L'utente potrà interagire con il sistema sia attraverso una GUI (Graphical User Interface) che con una CLI (Command Line Interface). La scelta dell'interfaccia utente verrà effettuata dall'utente durante l'avvio dell'applicazione.

## Tecniche, Tecnologie e Framework applicati

Come facilmente comprensibile, l'elaborato non pone particolare enfasi sulla complessità dell'applicazione, bensì sull'utilizzo di un insieme di tecniche, tecnologie e framework per lo sviluppo agile del sistema, differente dal tradizionale sviluppo in cascata. In particolare, il sistema va sviluppato tramite le tecniche del TDD. Il TDD è basato su cicli di sviluppo

---

<sup>1</sup> <[https://it.wikipedia.org/wiki/Test\\_driven\\_development](https://it.wikipedia.org/wiki/Test_driven_development)> ultima consultazione: 30/06/2020

<sup>2</sup> <<https://leanpub.com/tdd-buildautomation-ci>> ultima consultazione: 30/06/2020

piuttosto piccoli e rapidi. Viene prima creato un test per una feature non ancora implementata; il test sarà quindi destinato a fallire. Successivamente si scrive il codice necessario al superamento del test e, se necessario, viene eseguito del refactoring<sup>3</sup>.

Oltre al TDD, verranno utilizzate un diverse tecnologie,framework e strumenti, le cui funzionalità vengono brevemente descritte di seguito:

- Ubuntu: distribuzione Linux basata su Debian. In questo progetto viene utilizzata come distribuzione per lo sviluppo dell'applicazione
  - Versione utilizzata da Liuzzo: 20.04 LTS
  - Versione utilizzata da Nesi:16.04 LTS
- Java: linguaggio di programmazione ad alto livello, orientato agli oggetti e a tipizzazione statica<sup>4</sup>. Durante il progetto verrà utilizzato come principale linguaggio di programmazione per la scrittura del codice sorgente.
  - Versione utilizzata durante lo sviluppo: Java 8
- Eclipse: IDE multiplatforma e multilinguaggio. Durante il progetto verrà utilizzato principalmente come editor per la scrittura del codice sorgente, in larga parte scritto in Java (Java 8, in particolare)
  - Versione utilizzata durante lo sviluppo: 2019-06
- JUnit: framework per la scrittura di test. Durante il progetto verrà utilizzato per lo Unit Test e per l'Integration Test.
  - Versione utilizzata durante lo sviluppo: JUnit 4
- AssertJ: libreria per l'inserimento di asserzioni nei test. Verrà impiegato per migliorare la leggibilità dei test.
- JaCoCo: strumento per il Code Coverage
  - EclEmma: plugin Eclipse per JaCoCo
- PIT: framework per il Mutation Testing
  - Pitclipse: plugin Eclipse per PIT
- Maven: strumento per la build automation di progetti Java. Verrà utilizzato per automatizzare il processo di gestione delle dipendenze, compilazione, test e report dei risultati (i.e. il processo di build), sia in locale che in remoto
  - M2Eclipse: plugin Eclipse per Maven
- Mockito: framework per il Mocking. Verrà utilizzato per il mock delle dipendenze durante lo Unit Test
- Git: VCS distribuito.
  - EGit: plugin Eclipse per Git

---

<sup>3</sup> Lorenzo Bettini, *Test-Driven Development, Build Automation, Continuous Integration with Java, Eclipse and friends*, Leanpub, 2018, p. 5.

<sup>4</sup> <[https://it.wikipedia.org/wiki/Java\\_\(linguaggio\\_di\\_programmazione\)](https://it.wikipedia.org/wiki/Java_(linguaggio_di_programmazione))> Ultima consultazione: 30/06/2020

- GitHub: sito web per l'hosting di progetti git
- Travis CI: server CI per il build in remoto
- Docker: strumento per la virtualizzazione di servizi (come i database) su container
- Swing: toolkit per lo sviluppo di interfacce grafiche in Java
- AssertJ Swing: framework per il testing di interfacce grafiche create tramite Swing
- SonarQube: piattaforma per l'ispezione del codice e stima della code quality. Può essere utilizzato insieme a JaCoCo per ottenere una stima del code coverage.
  - SonarCloud: versione cloud di SonarQube
  - SonarLint: plugin multi-IDE per la qualità
- MongoDB: DBMS non relazionale, NoSQL, basato sui documenti

Una descrizione più ricca delle tecnologie più importanti verrà fornita in [Sviluppo e Testing](#), quando queste tecnologie verranno utilizzate durante lo sviluppo.

## Progettazione del Sistema

### Descrizione delle Funzionalità del Sistema

Le funzionalità principali del sistema sono:

- Creazione di un Profilo Studente
- Rimozione di un Profilo Studente
- Inserimento di un Corso (legato ad uno Studente)
- Rimozione di un Corso (legato ad uno Studente)
- Visualizzazione dei corsi di uno studente

#### Creazione Profilo Studente

L'Utente può inserire, all'interno del sistema, le informazioni relative ad uno studente (ossia la matricola o identificativo e il nome), e creare quindi un nuovo profilo. Il sistema creerà un nuovo profilo studente, inizialmente senza corsi a lui correlati.

#### Rimozione Profilo Studente

L'Utente può, in ogni momento, cancellare uno dei profili studente presenti nel sistema. In questo caso, il sistema eliminerà le informazioni circa i corsi legati al profilo appena rimosso nel caso in cui lo studente sia l'unico iscritto.

#### Inserimento Corso

L'Utente può inserire, all'interno del sistema, le informazioni circa un corso legato ad uno dei profili studente presenti nel sistema. Il sistema dovrà poi creare un'associazione tra il corso e il profilo studente e, se necessario, inserirà le informazioni del corso all'interno del sistema.

## Rimozione Corso

L'Utente può, in ogni momento, rimuovere dal sistema le informazioni circa un corso legato ad uno o più dei profili studente presenti nel sistema. Il sistema dovrà rimuovere l'associazione tra il corso e il profilo studente e, se necessario, rimuoverà le informazioni relativo al corso dal sistema.

Il Sistema deve notificare l'utente l'esito delle operazioni svolte, sia in caso di successo che in caso di errore.

## Architettura del Sistema e Diagramma delle Classi

Durante la breve fase di analisi preliminare, si è ritenuto opportuno ricorrere ad un'architettura a layer, che possiamo per certi versi vedere come un'estensione dell'architettura Model-View-Presenter. Il sistema infatti presenta i componenti tipici dell'architettura MVP, ossia la View, il Model e il Controller/Presenter; a questi sono stati aggiunti altri due layer: il Service e il Repository

### View

Questo layer si occupa di mostrare le informazioni di interesse all'utente, prendere i suoi input e notificare l'utente circa l'esito delle operazioni. Per questo progetto, è stato deciso di implementare la view in due modi diversi, ossia tramite una GUI e tramite una CLI. La GUI verrà implementata utilizzando Swing (i.e. la libreria ufficiale per la realizzazione di interfacce grafiche in Java<sup>5</sup>)

L'utente può scegliere quale interfaccia utente utilizzare tramite il parametro `--user-interface` (oppure `--ui`). I valori possibili sono `cli`, per utilizzare l'interfaccia a riga di comando, e `gui`, per utilizzare l'interfaccia grafica (se questo valore non viene specificato, l'applicazione verrà avviata con la GUI).

### Controller

Questo layer si occupa di prendere le informazioni e le richieste inviate dall'utente, tramite la view, e di inviare delle richieste di alto livello (come, ad esempio, la creazione di un profilo studente o la creazione di un corso) al service layer. Il controller si occupa anche di notificare, tramite la view, l'utente circa l'esito delle operazioni e aggiornare lo stato della view.

### Service

---

<sup>5</sup> <[https://it.wikipedia.org/wiki/Swing\\_\(Java\)](https://it.wikipedia.org/wiki/Swing_(Java))> Ultima consultazione: 18/07/2020

Questo layer si occupa di fornire servizi di alto livello al controller. Tali servizi sono legati alla logica di funzionamento dell'applicazione, come ad esempio quello che gestisce l'inserimento di un nuovo corso (inserimento che è legato ad uno specifico studente).

Il Service layer si appoggia al Transaction Manager per eseguire le operazioni necessarie a fornire i suoi servizi all'interno di una transazione. Il Service layer dipende quindi solo dal Transaction Manager.

## Repository

Il layer repository si occupa dell'esecuzione di operazioni di basso livello che comportano la lettura e la scrittura sul database, oltre al già citato Transaction Manager.

A partire dalla definizione delle interfacce che definiscono sia gli oggetti Repository (uno per ogni entità del dominio) che il Transaction Manager, sono state create delle classi che implementano tali interfacce. Solo le implementazioni delle interfacce sono classi la cui sintassi dipende dal DBMS utilizzato, che ricordiamo essere MongoDB.

Il Transaction Manager fornisce dei modelli che utilizzano delle interfacce funzionali, che rappresentano l'esecuzione di espressioni lambda. Nello specifico, l'interfaccia `TransactionCode` estende `BiFunction` e rappresenta una lambda che prende in ingresso i due oggetti Repository, mentre le interfacce `StudentTransactionCode` e `CourseTransactionCode` estendono `Function` e rappresentano una lambda che prendono in ingresso, rispettivamente, uno `StudentRepository` e un `CourseRepository`.

## Model

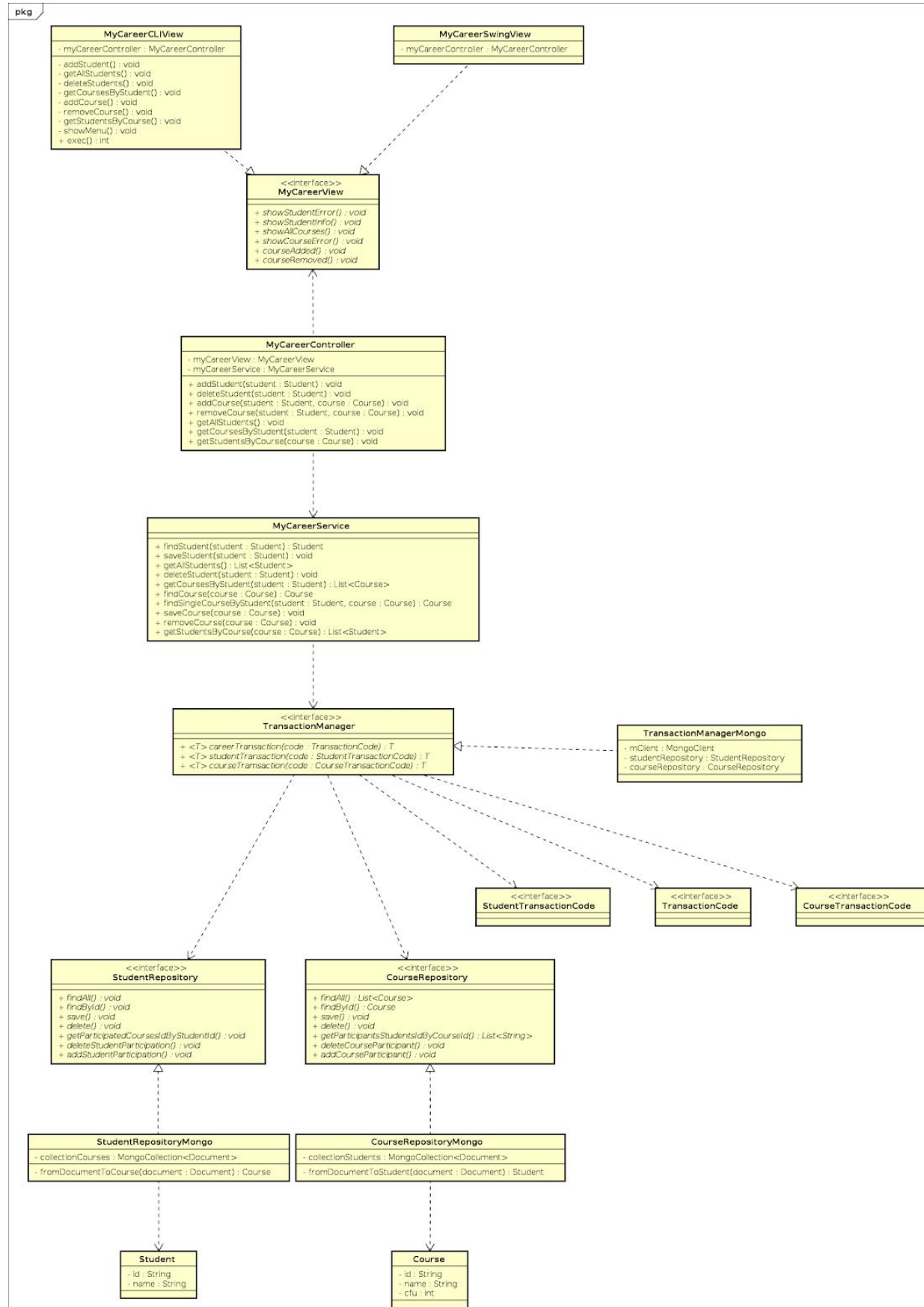
Il model contiene le classi del dominio applicativo. Le classi del model descrivono le entità che vengono manipolate durante il funzionamento dell'app.

Utilizzando un'architettura di questo tipo, le classi del model sono indipendenti dalla tecnologia utilizzata per la gestione della persistenza.

Le entità che compongono il model sono:

- **Student:** questa classe rappresenta lo studente/utente, che è identificato tramite un ID (univoco) e che fa le veci del numero di matricola e il nome
- **Course:** questa classe rappresenta il corso, e viene identificato tramite un ID (univoco) e che fa le veci del codice del corso, il nome e il numero di CFU

Di seguito viene mostrato un diagramma riassuntivo delle classi (all'interno della sezione [Link di Riferimento](#) è presente il link all'immagine originale). Per esigenze di spazio, vengono riportati solo gli attributi e i metodi di maggiore interesse; per esempio, sono omesse gli attributi della GUI che sono stati generati tramite l'editor grafico.





## Testing e Sviluppo

### Unit Testing

In questa fase, le singole classi vengono testate **in isolamento**, ossia in maniera indipendente dalle altre classi con cui interagisce. Queste componenti, durante lo unit testing, vengono “simulate” tramite il **mocking**,

Come framework per il testing (tutte le tipologie di test) è stato utilizzato [JUnit \(versione 4.13\)](#) insieme alla libreria [AssertJ](#) per le asserzioni, in modo da utilizzare uno stile più fluente e scrivere dei test che siano il più leggibile possibile.

Dato che i test possono essere considerati una forma di **documentazione eseguibile**, viene data un’elevata priorità alla leggibilità dei test.

La struttura tipica di un test è costituita da:

- Setup
- Exercise
- Verify

Come accennato prima, le dipendenze vengono simulate tramite il mocking, che consente di creare delle “false implementazioni” delle nostre dipendenze. Per il mocking, è stato utilizzato [Mockito](#).

L’aspetto fondamentale da evidenziare è che, durante lo unit testing, è necessario scrivere un test per ogni “possibile percorso” del metodo testato; questo tipo di testing è fattibile solo quando il metodo viene testato in isolamento. Per esempio, per il metodo `deleteStudent` della classe `myCareerController` deve essere scritto un test per ognuna delle seguenti situazioni:

- il profilo studente è presente nel sistema
- il profilo studente non è presente nel sistema

Oltre all’esecuzione dei metodi di test, viene testato il code coverage (in locale tramite [JaCoCo](#) e in remoto tramite [SonarCloud](#)), ossia viene misurata la percentuale di codice coperto dai test (si deve puntare al 100%) e viene effettuato il mutation testing (con [PIT](#)).

In generale, possiamo affermare che non tutte le classi “meritano” un test. Vengono infatti testate le classi che contengono una certa “logica” al loro interno. Un esempio tipico di classi che non vengono testate sono le classi che rappresentano le entità del dominio applicativo (in questo caso, le classi `Student` e `Course`), in quanto gli unici metodi che possiedono sono i getters/setters e l’override di `equals` e `toString`, che sono stati generati automaticamente tramite Eclipse, e che non possiedono una complessità tale da rendere

necessario lo unit test. Un'altra classe su cui si è ritenuto necessario eseguire lo unit test è la classe che implementa il Transaction Manager, in quanto i loro metodi non presentano logica.

Si accenna al fatto che, per lo unit test delle classi che implementano le interfacce del repository, è stato utilizzato un DB in-memory.

## UI Testing

### GUI Test

Dato che la GUI è stata implementata utilizzando [Swing](#) (la GUI è stata in gran parte implementata tramite l'utilizzo di [WindowBuilder](#)), per il testing è stata utilizzata la libreria [AssertJ Swing](#). In questo caso, la classe contenente i metodi di test estende la classe `AssertJSwingJUnitTestCase` e utilizza un oggetto della classe `FrameFixture` per poter simulare l'interazione dell'utente con la GUI.

I metodi contenuti nella classe di test verificano:

- lo stato iniziale degli elementi contenuti nella GUI;
- che le eventuali variazioni nello stato degli elementi avvengano come previsto
- che la view deleghi correttamente al controller le operazioni da effettuare in risposta ad un evento (come il click su un pulsante o la selezione di un elemento nelle list).

Per testare la GUI in isolamento viene fatto il mocking del controller.

Di seguito viene riportato uno screenshot della GUI:

The screenshot shows a window titled "My Career" with standard window controls (minimize, maximize, close). The window is divided into two main panels. The left panel, titled "Student Section", contains input fields for "ID" and "Name", an "Add Student" button, a large empty rectangular area, and a "Delete Student" button at the bottom. The right panel, titled "Course Section", contains input fields for "ID", "Name", and "CFU", an "Add Course" button, a large empty rectangular area, and a "Delete Course" button at the bottom.

## CLI Test

A differenza della GUI, per lo unit test (e la conseguente implementazione) della CLI, non sono state utilizzate librerie di appoggio in più rispetto a quelle normalmente impiegate per le altre classi. Le classi di test quindi ricorrono al solo utilizzo di JUnit (e AssertJ).

Per lo unit test, si è deciso di utilizzare un approccio a thread singolo. Anche in questo caso, i metodi del controller vengono simulati tramite Mockito.

La classe che implementa la CLI prende, tramite i parametri del costruttore, gli stream di I/O che andrà a utilizzare. In questo modo è possibile utilizzare gli stream "standard" (ossia `System.in` e `System.out`) durante l'utilizzo reale, mentre, durante il test, viene utilizzato un oggetto della classe `ByteArrayStream` per simulare l'input dell'utente e un oggetto della classe `PrintStream`, inizializzata tramite un oggetto della classe `ByteArrayOutput`, per simulare e controllare l'output.

Di seguito viene mostrato uno screenshot della CLI (lo screenshot è preso dall'app in esecuzione)

```
Student Section
1) Add a student
2) Get all students
3) Delete a student
4) Get courses (by student)
Course Section
5) Add a course subscription
6) Remove a course subscription
7) Get students (by course)
8) Exit
Enter a valid digit:
1
Insert id: 7012008
Insert name: Mauro
Student account created : Student [id=7012008, name=Mauro]
Student Section
1) Add a student
2) Get all students
3) Delete a student
4) Get courses (by student)
Course Section
5) Add a course subscription
6) Remove a course subscription
7) Get students (by course)
8) Exit
Enter a valid digit:
2
Student [id=7012008, name=Mauro]
Student Section
1) Add a student
2) Get all students
3) Delete a student
4) Get courses (by student)
Course Section
5) Add a course subscription
6) Remove a course subscription
7) Get students (by course)
8) Exit
Enter a valid digit:
8
Goodbye
mauroliuzzo@ideapad-520-15IK8:~/Scaricati$
```

## Integration Testing

Dopo che i singoli componenti sono stati implementati (e quindi anche testati in isolamento tramite lo unit testing), durante la fase di integration testing si verifica che i componenti dell'app continuino a funzionare quando vengono integrati tra loro e con componenti di terze parti (come, ad esempio, un database).

Dato che questo tipo di test è implicitamente più lento rispetto allo unit test, vengono testati quelli che sono i casi "positivi"; in altri termini, non vengono testati tutti i possibili scenari, anche perché non è spesso fattibile creare facilmente test per i casi "negativi". Viene inoltre usato un server CI (si rimanda al paragrafo [Build Locale e su Server CI](#)) per effettuare tutti i test su un server remoto, evitando quindi di impegnare le risorse locali.

Durante questa fase, le classi che implementano le interfacce del repository sono state testate insieme ad DB Mongo "reale", che viene eseguito su un Docker container.

I test sono stati creati ed eseguiti seguendo un approccio bottom-up: è stata testata per prima l'interazione tra i repository ed il db (eseguito su docker); successivamente è stata testata l'interazione tra il transaction manager e i componenti sottostanti; a seguire è stata testata l'interazione tra il controller e i layer sottostanti e l'interazione tra le view (è stato creato un test per ogni implementazione della view) e i layer sottostanti.

## E2E Testing

In questa fase l'intera applicazione viene testata.. La principale differenza tra un test E2E e l'integration test è che l'interazione avviene esclusivamente attraverso l'interfaccia utente. Dato che l'applicazione prevede l'utilizzo di due UI differenti, sono state scritte due classi di test: una che test l'app con la GUI, mentre l'altra testa l'app con la CLI.

### GUI Test

Durante la fase di setup, viene popolato il DB con un paio di studenti e un paio di corsi, utilizzando le API della classe `MongoClient`.

L'applicazione viene lanciata utilizzando le API di AssertJ Swing della classe `ApplicationLauncher`, mentre vengono usate invece le API della classe di `WindowFinder` per cercare il frame in base al titolo. In generale, tutte le asserzioni vengono fatte attraverso le API di AssertJ Swing.

Durante questo test, l'applicazione viene lanciata invocando la classe che contiene il main.

### CLI Test

Anche in questo caso, durante la fase di setup viene popolato il DB utilizzando le API della classe `MongoClient`.

In questo caso non vengono utilizzate le API di AssertJ Swing per avviare l'app. Il container docker contenente il DB viene avviato utilizzando le API della classe Runtime, mentre l'applicazione viene lanciata utilizzando le API della classe ProcessBuilder.

Le API di ProcessBuilder vengono utilizzate inoltre per ottenere gli stream di I/O del processo, necessari per simulare l'inserimento di input da parte dell'utente e per ottenere l'output stampato dalla CLI.

Si segnala che, in questo caso, l'applicazione viene lanciata utilizzando il file .jar; si ha un approccio diverso rispetto al test E2E della GUI, che simula in maniera più precisa la reale interazione tra l'utente e l'applicazione.

## Virtualizzazione tramite Docker

L'installazione di un server MongoDB (o altri server, come quelli per MySQL o i server Apache) sulle macchine utilizzate per lo sviluppo è in generale da sconsigliare, in quanto comporta uno spreco di risorse e un aumento della complessità nella configurazione tra i membri del team di sviluppo. Si è ritenuto quindi opportuno virtualizzare tale servizio.

Per la virtualizzazione del server MongoDB è stato utilizzato [Docker](#), che fornisce la possibilità di virtualizzare tramite i container. Il vantaggio principale dell'utilizzo dei container Docker, oltre al risparmio di risorse dato dal fatto che Docker utilizza lo stesso kernel del sistema operativo dell'host, è dato dall'elevata riproducibilità che un ambiente di sviluppo possiede, se paragonato ad altre tecniche di virtualizzazione, come l'utilizzo delle macchine virtuali.

Abbiamo usato una particolare immagine Docker che ci ha permesso di utilizzare la transazioni in modo più congeniale alla struttura della applicazione rispetto a quello che sarebbe stato possibile con una normale immagine MongoDB (vedi paragrafo "[Utilizzo delle funzionalità avanzata di MongoDB](#)").

Durante i test eseguiti sulle macchine degli studenti, il container MongoDB viene configurato e lanciato all'interno dei test JUnit tramite [Testcontainers](#) o tramite Process. Per MongoDB viene utilizzata la classe GenericContainer.

Durante la normale esecuzione dell'applicazione il container docker contenente il server MongoDB va avviato prima dell'avvio dell'applicazione (ed eventualmente chiuso dopo la chiusura dell'applicazione).

## Build Locale e su Server CI (i.e. Maven e Travis CI)

Durante l'intero sviluppo dell'applicazione, è stato utilizzato Maven, in modo da disporre di uno strumento per la build e la gestione del progetto, soprattutto per quanto riguarda la

gestione delle dipendenze. Quando avviene una build del progetto, Maven si occupa di scaricare (se necessario) tutte le dipendenze, compilare i sorgenti, eseguire i test e generare report<sup>6</sup>. Maven inoltre migliora notevolmente la riproducibilità dell'ambiente di sviluppo, in quanto tutte le dipendenze vengono gestite in automatico

Il progetto è stato suddiviso in più moduli, in modo da aumentare la modularità e l'indipendenza delle componenti dell'applicazione. Nello specifico, la suddivisione in moduli è la seguente:

- Modulo Parent: si occupa di gestire le dipendenze, i plugin e le configurazioni condivise dai suoi moduli figli.
- Modulo Back-End: contiene il model, il repository e il service layer; contiene quindi le componenti dell'app che si occupano dell'accesso e la gestione dei dati, similmente a quello che avviene nel back-end delle web app (da cui si ispira il nome del modulo). Questo modulo è "figlio" del modulo parent.
- Modulo Front-End: contiene le view e il controller; contiene quindi le componenti dell'app che si occupano della gestione dell'interazione con l'utente. È "figlio" del modulo parent e dipende dal back-end.
- Modulo App: contiene la classe con il metodo main. Dipende dal front-end e dal back-end, ma NON è figlio del modulo parent.
- Modulo Aggregator: fa da aggregatore, consentendo al reactor di Maven di eseguire la build dei moduli nel corretto ordine.

Maven consente di eseguire localmente la build del progetto. Tuttavia, i principali vantaggi di Maven emergono quando esso viene affiancato ad un VCS (nel nostro caso Git) e un server CI (nel nostro caso Travis CI).

Viene utilizzato Travis come server per la Continuous Integration. L'utilizzo di un server CI consente di eseguire la build dell'intera applicazione (operazione che potrebbe essere molto dispendiosa se eseguita in locale) su un server remoto. Nel caso più semplice viene effettuata una build dell'app ad ogni operazione di push sulla repository (Travis CI è legato all'utilizzo di GitHub). L'ambiente di esecuzione della build sul server Travis viene definito all'interno di un file YAML (i.e. il file `.travis.yml`).

Durante la build viene inoltre effettuato il mutation testing, grazie all'utilizzo del plugin maven di PIT.

## Utilizzo di Git e GitHub

Durante lo sviluppo in gruppo (oltretutto in remoto) di un progetto, risulta necessario uno strumento che consenta la condivisione del codice (e relative risorse) che renda possibile tener traccia delle modifiche effettuate al progetto. Tali funzionalità sono fornite dai

---

<sup>6</sup> Lorenzo Bettini, *Test-Driven Development, Build Automation, Continuous Integration with Java, Eclipse and friends*, Leanpub, 2018, p.124.

Version Control Systems. Durante lo sviluppo di questo progetto è stato utilizzato Git, probabilmente il VCS distribuito più diffuso, e GitHub, uno dei siti di hosting per repository Git più utilizzati. L'utilizzo di GitHub permette una più semplice integrazione con gli strumenti per la CI menzionati in precedenza (i.e. Travis e SonarCloud), che ben si sposano con GitHub.

## Utilizzo delle funzionalità avanzata di MongoDB

Dal punto di vista della “struttura” del DB, si hanno due collezioni, una per memorizzare gli studenti e una per memorizzare i corsi. Per gestire la relazione molti-a-molti, abbiamo optato per l'utilizzo di un modello referenced<sup>7</sup>: ogni documento che rappresenta uno studente contiene una lista con gli id dei corsi che frequenta; viceversa, ogni documento rappresentante un corso ha al suo interno una lista con gli id degli studenti che frequentano il corso.

Le transazioni possono essere implementate utilizzando le API native fornite da Mongo. Una transazione viene eseguita in questo modo:

- Viene creato un oggetto della classe `Session`, utilizzando il metodo `startSession` della classe `MongoClient`
- La transazione viene avviata tramite il metodo `startTransaction`.
- Se le operazioni svolte all'interno della transazione hanno tutte successo, la transazione viene confermata tramite il metodo `commitTransaction`, altrimenti si utilizza il metodo `abortTransaction` per annullare la transazione
- L'oggetto della classe `Session` viene chiuso tramite il metodo `close`.

Per utilizzare queste API è necessario configurare il DB come “replica set”, in modo che il DB supporti l'utilizzo delle sessioni, e consenta quindi l'esecuzione delle transazioni multi-collection (). Tuttavia, esiste un'[immagine docker](#) di un DB MongoDB che estende l'immagine già impostata come replica set. Si è deciso quindi di utilizzare questa immagine per i container utilizzati durante lo sviluppo dell'applicazione.


## Code Quality con SonarQube/SonarCloud

SonarQube è una piattaforma per l'ispezione continua della code quality, che misura l'affidabilità e la manutenibilità dell'applicazione, tramite l'analisi statica del codice sorgente.

---

<sup>7</sup> <<https://bezkoder.com/mongodb-many-to-many-mongoose/>> Ultima consultazione: 18/07/2020





La stima della code quality può essere effettuata sia in locale, utilizzando uno stack avviabile tramite docker-compose, oppure sul cloud, utilizzando SonarCloud e abbinandolo ad un server per la continuous integration come, in questo caso, Travis CI.

Inizialmente è stato utilizzato SonarQube, mentre nelle fasi avanzate è stato utilizzato SonarCloud.

Dopo l'analisi (effettuata durante la build maven, dopo la fase di test), Sonar riporta tutti gli "issues" legati al codice, che possono rendere il codice meno sicuro, affidabile e manutenibile. Gli issues più importanti sono stati corretti modificando manualmente il codice, mentre altri (come ad esempio, il fatto che la view appartenga ad una gerarchia profonda, o che i metodi di test che utilizzano i metodi di AssertJ Swing vengono segnalati come metodi privi di asserzioni ) sono stati ignorati, definendo le apposite proprietà all'interno del POM dell'aggregator, che viene visto come modulo root durante l'analisi.

Utilizzando SonarCloud insieme a Travis e JaCoCo (definito come plugin nel POM), è stata misurata la code coverage escludendo in modo opportuno le classi del model, la classe con il metodo main e il TransactionManager; queste classi vengono escluse dal code coverage in quanto non presentano logica.

## Link di Riferimento

Link repository GitHub:

<https://github.com/liuzzom/my-career>

Link diagramma delle classi (formato png):

<https://drive.google.com/file/d/1uPmD0hdoOXtzc2O-tsJw4rWSby1PForc/view?usp=sharing>