

Lo scopo del workshop

Lo scopo del workshop è esercitarsi a scrivere test di caratterizzazione di codice legacy con dipendenze problematiche. In questo esercizio prendiamo un semplice codice legacy difficile da mettere sotto test e impariamo a rompere le dipendenze per poterlo mettere sotto test facilmente. Una volta che il codice è stato messo tutto sotto test si può procedere con il Refactoring con cui si migliora la qualità del design senza cambiare il comportamento del codice.

Collegati alla Wi-Fi

Potrebbe farti comodo avere accesso alla Wi-Fi. Mikamai offre una Wi-Fi guest per questi eventi, guarda sul muro dove dovrebbero esserci il nome della WiFi e la password da usare.

Scaricati il codice dell'esercizio

Per fare l'esercizio usiamo il codice `TripServiceKata` di Sandro Mancuso, che puoi vedere sul suo GitHub (<https://github.com/sandromancuso/trip-service-kata>). Per poter svolgere l'esercizio è necessario che ti scarichi il codice. Se hai Git installato puoi scaricare il codice con:

```
git clone https://github.com/sandromancuso/trip-service-kata.git
```

L'esercizio

Il codice dell'esercizio implementa (in parte) un social network per viaggiatori. È un social network dove gli utenti posso vedere i viaggi che fanno gli altri.

Trattandosi di un esercizio non c'è veramente tutto il sistema, c'è solo una parte, cioè quella relativa a queste a queste due regole di business:

- Solo gli utenti loggati possono vedere i contenuti del sito.
- È possibile vedere solo i viaggi degli utenti "amici"

Queste due regole sono implementate nella **`TripService::findTripsByUser()`** che è quella che dovrai mettere sotto test.

Devi far finta di aver ereditato del codice legacy e che vuoi rifattorizzarlo per migliorarne il design, però nel farlo devi seguire la regola del codice legacy.

La regola del codice legacy:

È vietato modificare il codice di produzione se non è coperto da test (l'unica cosa permessa sono i refactor automatici dell'IDE e solo se sono necessari per scrivere il test).

Per risolvere l'esercizio dovrai:

1. mettere completamente sotto test il metodo **`findTripsByUser`**

2. (solo dopo aver aggiunto tutti test) rifattorizzare il codice di **findTripsByUser** per semplificarne il design (vedremo dopo cosa si intende con design semplice)

Il problema delle dipendenze

Questo codice è stato scritto senza pensare alla sua testabilità e infatti ha un problema di dipendenze che deve essere risolto prima di poterlo mettere sotto test.

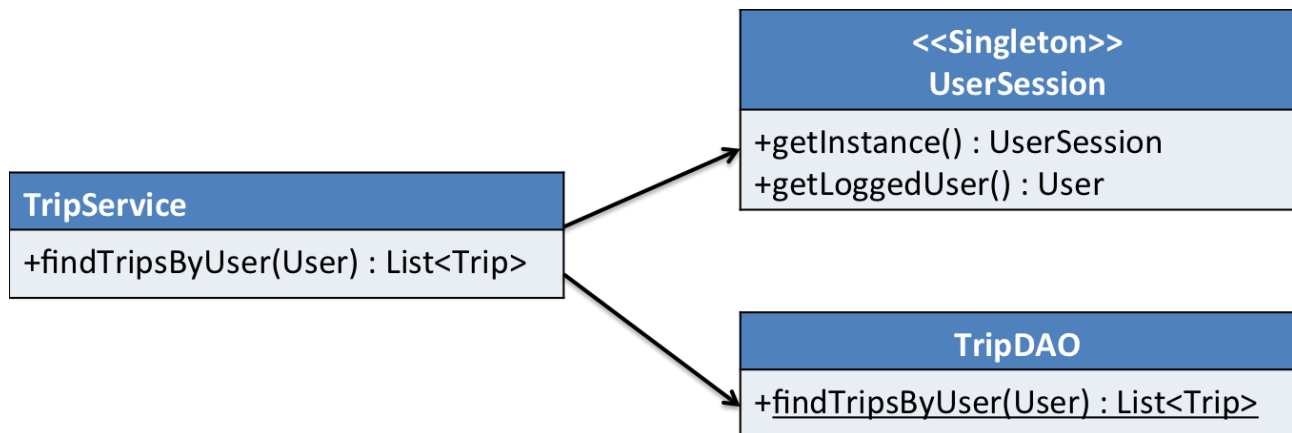


Figure 1: Dipendenze

Il metodo che dobbiamo mettere sotto test è `TripService::findTripByUser()` e dipende da due classi che non possiamo usare nell'ambiente di testing: `UserSession`, e `TripDAO`.

Per riuscire a mettere sotto test il codice di `findTripByUser` dobbiamo rifattorizzarlo in modo che non cerchi di eseguire il codice di queste due classi.

Step 1. Aprire il progetto

Aprire il progetto e individuate la classe che dobbiamo mettere sotto test: `TripService`.

Con Java su IntelliJ IDEA: il modo giusto di aprire il progetto è fare “Apri progetto” della cartella `trip-service-kata/java`, poi andare sul `pom.xml` > tasto destro > Add as a project. Non provate l'import.

Per gli altri linguaggi/IDE vedete voi.

Nota: Nel codice esiste anche una classe che si chiama `TripServiceOriginal`, è uguale a `TripService`. Non consideratela: non viene usata, esiste solo perché così alla fine dell'esercizio si può confrontare l'eventuale versione rifattorizzata di `TripService` con la sua versione originale.

Step 2. Lanciare il primo test

Questo è uno step di rodaggio, dobbiamo solo dimostrare di riuscire a lanciare un semplice test e vedere che il framework funziona.

Per superare questo passo dovete scrivere un test che contenga una asserzione del tipo:

```
assertEqual(1, 2)
```

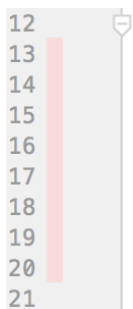
e vedere che fallisce, poi dovete cambiarla in qualcosa che passa, tipo:

```
assertEqual(1, 1)
```

e vedere che passa con barra verde.

Step 3. Abilitare il plugin di coverage

Per completare questo passo dovete abilitare il coverage sul vostro ambiente e vedere quali sono le righe di **findTripsByUser()** coperte o meno dai vostri test. Dato che all'inizio i test fanno solo `assertEqual(1,1)` ci aspettiamo che il plugin ci faccia vedere le righe come rosse (cioè non toccate dal test).



```
12 public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
13     List<Trip> tripList = new ArrayList<Trip>();
14     User loggedInUser = UserSession.getInstance().getLoggedInUser();
15     boolean isFriend = false;
16     if (loggedInUser != null) {
17         for (User friend : user.getFriends()) {
18             if (friend.equals(loggedUser)) {
19                 isFriend = true;
20                 break;
21             }
12 }
```

Passi per Java su IntelliJ:

- Abilitate e scaricate il plugin “Coverage” da Preferences > Plugin
- Vi dovrebbe chiedere di riavviare l’IDE.
- Dopo il Restart dovrebbe essersi aggiunta una nuova voce al menu Run del tipo: “Run all test with Coverage”

Passi per PHP da linea di comando:

- Serve PHP con xdebug.
- Seguite le istruzioni sul README.md dentro la directory trip-service-kata/php

Passi per PHP su PHPStorm:

- Fare le stesse cose per PHP da linea di comando e marcare la directory “src” come “Source Root” e la directory “test” come “Source Test Root”
- Lanciare i test con Run > Run ‘test’ with Coverage.

Passi per C#/Visual Studio:

- Installate dotCover di JetBrains.

Kotlin:

- È stato segnalato che funziona la configurazione: Kotlin + IntelliJ + kotlintest con coverage

Python:

- È stato segnalato che funziona la configurazione: Python + PyCharm + Coverage (Usando pytest con pytest-cov)

Passi su altri linguaggi/sistemi:

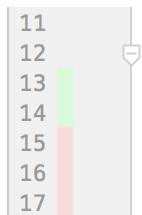
- Non lo so.

Attenzione: Se non funziona il “coverage” pazienza:

Se arrivi al workshop e non hai avuto tempo di installare prima lo strumento coverage e non riesci a installarlo velocemente durante il workshop lascia perdere, gli strumenti in alcuni linguaggi non sono immediati da installare e quindi, per non perdere tempo durante il workshop, ti consiglio di non incaponirti nel tentativo di installazione. Se funziona al primo colpo bene, se non funziona lascia perdere per ora e concentrati sul capire come mettere sotto test del codice legacy esistente.

Step 4. Coprire la prima riga di findTripsByUser

L'obiettivo di questo passo è riuscire a toccare le prime due righe di **findTripsByUser**. Avete finito quando lo strumento di coverage vi fa la barretta verde a fianco delle prime due righe. Per completare questo step è sufficiente coprire queste due righe di codice, vale anche se i test ancora non passano ma queste due righe di codice sono coperte.



```

11
12 public List<Trip> getTripsByUser(User user) throws UserNotLoggedInException {
13     List<Trip> tripList = new ArrayList<Trip>();
14     User loggedInUser = UserSession.getInstance().getLoggedInUser();
15     boolean isFriend = false;
16     if (loggedInUser != null) {
17         for (User friend : user.getFriends()) {

```


Step 5. Prima barra verde

Per completare questo step dobbiamo mantenere la copertura ottenuta nel passo precedente ma dobbiamo far passare il test (barra verde). Quello che stiamo facendo è scrivere un test di caratterizzazione.

Cos'è un test di caratterizzazione

Quando facciamo TDD sappiamo quale è il comportamento che vogliamo ottenere e di conseguenza possiamo scrivere la asserzione ancora prima di aver scritto il codice di produzione.

Per esempio è perfettamente lecito scrivere una cosa del genere quando si fa TDD:



```

@Test public void should_strip_semicolons() {
    String result = sanitizeInput("a;bc;");
    assertEquals("abc", result);
}

```

La funzione sanitizeInput non esiste ancora ma noi sappiamo già come si dovrà comportare una volta creata e possiamo scrivere la asserzione di conseguenza.

Quando abbiamo a che fare con il codice legacy spesso ci troviamo nella situazione opposta:

1. il codice di produzione esiste già
2. non sappiamo esattamente come si dovrebbe comportare

Facciamo finta di avere un metodo **formatText** che non sappiamo bene come si comporti ma vogliamo scoprirlo. Per prima cosa scriviamo un test che lo eserciti in un caso interessante:

```
@Test public void when_plain_text() {  
    String result = formatText("plain text");  
    assertEquals(null, result);  
}
```

Dato che non sappiamo ancora cosa restituisce in questo caso mettiamo “null” come parametro dell’assertEquals. Poi lanciamo il test e vediamo come fallisce:

```
java.lang.AssertionError:  
Expected :null  
Actual   :plain text  
<Click to see difference>
```

Il messaggio ci dice cosa avremmo dovuto mettere per avere la barra verde. A questo punto basta aggiornare il valore “expected” dell’assertEquals per avere un test verde.

```
@Test public void when_plain_text() {  
    String result = formatText("plain text");  
    assertEquals("plain text", result);  
}
```

Praticamente quello che si fa è:

1. si parte scrivendo un metodo di test con nome generico, ad esempio test_something()
2. si scrive il codice che esercita la classe di produzione
3. si aggiunge un assertEquals mettendo come valore atteso qualcosa che siamo abbastanza sicuri che la faccia fallire (ad esempio “null”, o “-1”)
4. la si vede fallire e si segna il vero valore ottenuto
5. si aggiusta la assertEquals in modo che passi

Nel nostro caso il test non fallisce per una asserzione sbagliata ma per una eccezione inattesa. Per ottenere la barra verde dobbiamo far sì che il test si aspetti l’arrivo di quell’eccezione. Come si fa dipende dal linguaggio e dalla libreria di testing.

Potete usare:

- Java → usate `@Test(expected=...)`
- PHP → usate `$this->setExpectedException(...)`

- C# → `Assert.Throws<...>()`
- Altri linguaggi: googlare “how set to expect exception in LINGUAGGIO”

Intermezzo: Strategia di copertura del codice legacy

Ora inizia la parte in cui dovete cominciare a coprire tutto il codice di `findTripsByUser()`, la strategia consigliata è la seguente:

1. cominciate scrivendo i casi di test che coprono i rami esterni (meno innestati)
2. proseguite aggiungendo test che coprono i rami più innestati

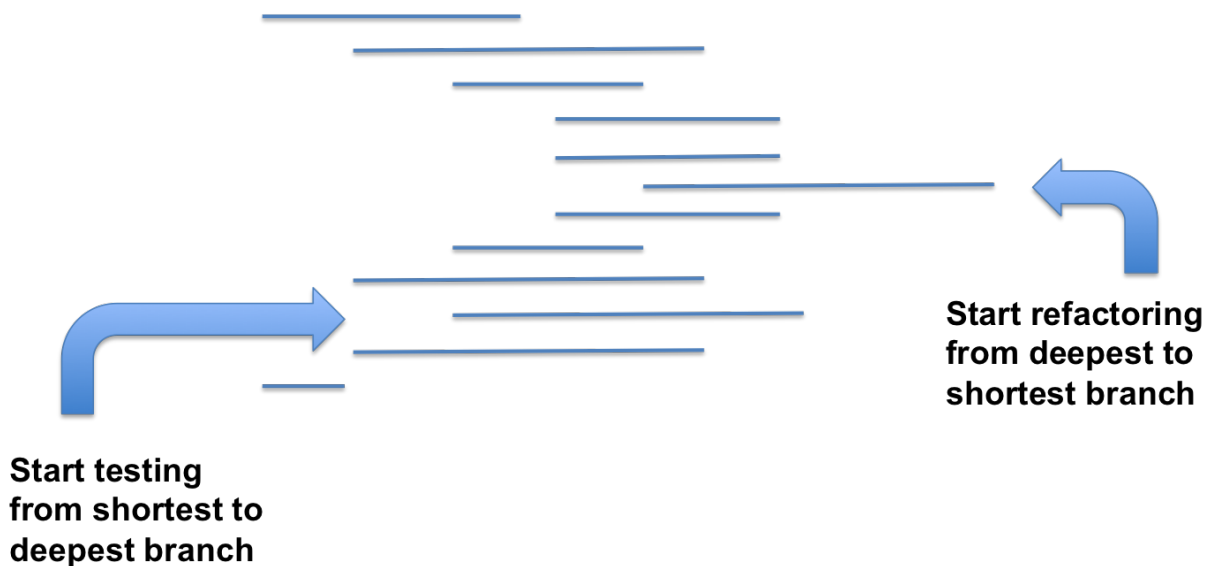


Figure 2: Strategia

Nel nostro caso cominciamo con il coprire prima di tutto questa throw:

```

26      return tripList;
27    } else {
28      throw new UserNotLoggedInException();
29    }

```

Step 6. Rompere la prima dipendenza per coprire la riga successiva

L’obiettivo ora è riuscire a coprire la riga successiva alla chiamata che coinvolge la `UserSession`.

```

11
12
13
14      List<Trip> tripList = new ArrayList<Trip>();
15      User loggedUser = UserSession.getInstance().getLoggedUser();
16      boolean isFriend = false;
17      if (loggedUser != null) {

```

Nel caso del codice Java fino ad adesso arrivavamo alla linea 14 che lanciava un’eccezione.

L'obiettivo di questo step è riuscire a superare la linea 14 e arrivare a toccare la successiva (quella con scritto `boolean isFriend = false;`).

Però ricordati della regola del codice legacy che continua a valere e non si può violare.

La regola del codice legacy:

È vietato modificare il codice di produzione se non è coperto da test (l'unica cosa permessa sono i refactor automatici dell'IDE e solo se servono per scrivere il test).

Step 7. Barra Verde

L'obiettivo è far tornare tutti i test che passano.

Step 8. Coprire la parte restante del metodo

L'obiettivo ora è coprire, uno ad uno, tutti i rami del metodo `findTripsByUser()`.

Consigli/Regole:

- Aggiungete un nuovo metodo di test per ogni nuovo ramo da coprire
- Prima di passare a coprire il nuovo ramo assicuratevi che i test scritti fino ad ora siano verdi.

Step 9. Refactor

L'obiettivo di questo passo è rifattorizzare il codice per renderlo il più semplice possibile.

Cosa si intende con codice semplice? Si intende quello che è conforme al principio del Simple Design:

Però ricordati della regola del codice legacy che continua a valere e non si può violare.

Simple Design:

Un codice è semplice quando:

1. Passa tutti i test
2. Rende chiara l'intenzione per cui è scritto
3. Non contiene duplicazione
4. Non contiene parti superflue

Credits e approfondimenti.

Questo workshop è stato organizzato da Andrea Francia, la maggior parte del materiale l'ho presa dai lavori di Sandro Mancuso (@sandromancuso su Twitter) e altri autori, in particolare:

- Repository del Trip Service Kata disponibile su GitHub:
<https://github.com/sandromancuso/trip-service-kata>

- Slides dell'intervento di Sandro Mancuso da cui ho copia e incollato le immagini usati per la Figura 1 (Dipendenze) e per la Figura 2 (Strategia):
<http://www.slideshare.net/sandromancuso/legacy-code-handson-session>
- Molti dei concetti che ho spiegato in questo documento e nel workshop li ho imparati guardando i video di questi due interventi di Sandro Mancuso:
 - https://www.youtube.com/watch?v=_NnElPO5BU0
 - <https://www.youtube.com/watch?v=WpKb1XqSiUs>
- La descrizione di cosa sia un test di caratterizzazione l'ho imparata dal libro "Working Effectively With Legacy Code" di Michael Feathers e dall'articolo su suo blog:
<https://michaelfeathers.silvrback.com/characterization-testing>.
- Il Simple Design è un'idea che ho preso da Kent Beck che è descritto in molti posti, per esempio qui: <http://wiki.c2.com/?XpSimplicityRules>
- I suggerimenti su Kotlin e Python/PyCharm arrivano dai commenti lasciati da Michele sulla pagina del meetup: <https://www.meetup.com/it-IT/TDD-Milano/events/237398373/>

Offerta: Corso sul testing automatico in azienda

Io sono disponibile per tenere corsi sul testing automatico nella vostra azienda. Se ti interessa mandami una mail a testing@andreafrancia.it con oggetto "Ne vorrei sapere di più" indicandomi il tuo numero di telefono, e ti chiamerò. – Andrea Francia