

Corso Web MVC

+

Emanuele Galli

www.linkedin.com/in/egalli/

Unit Test

- Verifica la correttezza di una “unità” di codice, permettendone il rilascio da parte del team di sviluppo con maggior confidenza
- Un unit test, tra l’altro:
 - dimostra che una nuova feature ha il comportamento atteso
 - documenta un cambiamento di funzionalità e verifica che non causi malfunzionamenti in altre parti del codice
 - mostra come funziona il codice corrente
 - tiene sotto controllo il comportamento delle dipendenze

JUnit in Eclipse

- Right click sulla classe (Xxx) da testare
 - New, JUnit Test Case
 - JUnit 4 (al momento default in Spring) o 5 (Jupiter)
 - Se richiesto, add JUnit library to the build path
- Il wizard crea una nuova classe (XxxTest)
 - I metodi che JUnit esegue sono quelli annotati @Test
 - Il metodo statico in junit.Assert fail() indica il fallimento di un test
- Per eseguire un test case: Run as, JUnit Test

Struttura di un test JUnit

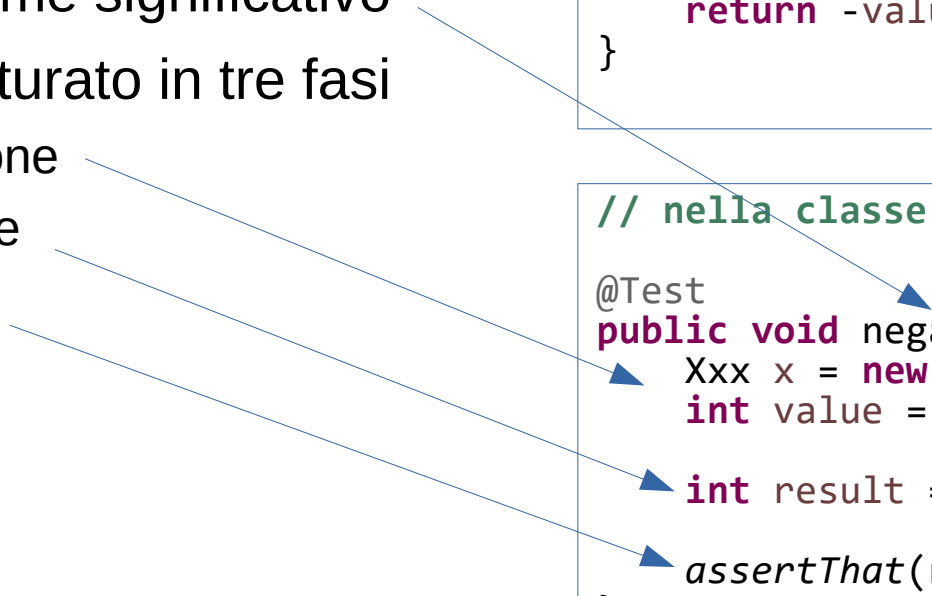
- Ogni metodo di test dovrebbe
 - avere un nome significativo
 - essere strutturato in tre fasi
 - Preparazione
 - Esecuzione
 - Assert

// nella classe Xxx

```
public int negate(int value) {  
    return -value;  
}
```

// nella classe XxxTest

```
@Test  
public void negatePositive() {  
    Xxx x = new Xxx();  
    int value = 42;  
  
    int result = x.negate(value);  
  
    assertThat(result, equalTo(-42));  
}
```



@Before e @BeforeEach

- I metodi annotati @Before (4) o @BeforeEach (Jupiter) sono usati per la parte comune di inizializzazione dei test
- Ogni @Test è eseguito su una nuova istanza della classe, per assicurare l'indipendenza di ogni test
- Di conseguenza, ogni @Test causa l'esecuzione dei metodi @Before o @BeforeEach

```
private MyTestClass mtc;

@BeforeEach // o @Before
public void init() {
    mtc = new MyTestClass();
}

// ...

@Test
public void negatePositive() {
    int value = 42;

    int result = mtc.negate(value);

    assertThat(result, equalTo(-42));
}
```

JUnit assert

- Sono metodi statici definiti in `org.junit.Assert` (4) o `org.junit.jupiter.api.Assertions` (Jupiter)
 - `assertTrue(condition)`
 - `assertNull(reference)`
 - `assertEquals(expected, actual)`
 - `assertEquals(expected, actual, delta)` // rappresentazione binaria di numeri floating point
 - `assertEquals(.87, .29 * 3, .0001);`
- `assert Hamcrest-style`, usano `Matcher`, metodi statici definiti in `org.hamcrest.CoreMatchers` o `org.hamcrest.MatcherAssert`
 - `assertThat(T, Matcher<? super T>)` // convenzione opposta ai metodi classici: actual – expected
 - `assertThat(result, equalTo(42));` // ← int result
 - `assertThat(name, startsWith("Tom"));` // ← String name
 - `assertThat(name, not(startsWith("Bob")));`
 - `assertThat(name, nullValue());`

JUnit 4 Eccezioni con @Rule

Math.abs() di
Integer.MIN_VALUE
è
Integer.MIN_VALUE!

```
public int negate(int value) {  
    if(value == Integer.MIN_VALUE) {  
        throw new IllegalArgumentException("Can't negate MIN_VALUE");  
    }  
    return -value;  
}
```

Si definisce una variabile di istanza
ExpectedException
taggata come @Rule

Nel @Test si dichiara
quale eccezione e messaggio
ci si aspetta

```
@Rule  
public ExpectedException thrown = ExpectedException.none();  
  
@Test  
public void negateMinInt() {  
    thrown.expect(IllegalArgumentException.class);  
    thrown.expectMessage("Can't negate MIN_VALUE");  
  
    mtc.negate(Integer.MIN_VALUE);  
}
```

JUnit 5 `assertThrows()`

Il metodo fallisce se quanto testato non tira l'eccezione specificata

L'eccezione attesa viene tornata per permettere ulteriori test

```
@Test
public void negateMinInt() {
    IllegalArgumentException exc = assertThrows(IllegalArgumentException.class, //
        () -> x.negate(Integer.MIN_VALUE));
    assertThat(exc.getMessage(), equalTo("Can't negate MIN_VALUE"));
}
```

L'assertion è eseguita su di un Executable, interfaccia funzionale definita in Jupiter

Build automation con Maven

- Build automation
 - Compilazione del codice sorgente
 - Packaging dell'eseguibile
 - Esecuzione automatica dei test
- UNIX make, Ant, Maven, Gradle
- Apache Maven, supportato da tutti i principali IDE per Java
 - pom.xml (POM: Project Object Model)
 - I processi seguono convenzioni stabilite, solo le eccezioni vanno indicate
 - Le dipendenze implicano il download automatico delle librerie richieste

Install in Maven

- Da maven.apache.org si può scaricare Maven in formato zip (o tar.gz)
- Basta estrarre l'archivio in una directory dedicata per poter eseguire Maven: “`mvn`” in “`bin`”
- Richiede Java, deve essere definita `JAVA_HOME`
 - Es: `set JAVA_HOME=C:\Program Files\Java\jre-10.0.1`
- Per verificare che Maven funzioni correttamente: `mvn --version`
- Il repository di Maven viene installato per l'utente corrente in “`.m2`”
- Si può installare un file (jar o altro) nel proprio repository di Maven. Esempio:
 - `mvn install:install-file -Dfile=/app/Administrator/product/18.0.0/dbhomeXE/jdbc/lib/ojdbc8.jar -DgroupId=com.oracle -DartifactId=jdbc -Dversion=8 -Dpackaging=jar`
 - Il risultato è che `ojdbc8.jar` viene copiato in `.m2\repository\com\oracle\jdbc\8\jdbc-8.jar`

Nuovo progetto Maven in Eclipse

- Creare un progetto Maven
 - File, New, Project → Wizard “Maven Project”
 - È necessario specificare solo `group id` e `artifact id`
 - Il progetto risultante è per Java 5
- Nel POM specifichiamo le nostre variazioni
 - Properties
 - Dependencies
- A volte occorre forzare l'update del progetto dopo aver cambiato il POM
 - Alt-F5 (o right-click sul nome del progetto → Maven, Update project)

Properties

- Nel elemento properties del POM si definiscono costanti
- Esempio: quali versioni usare nel progetto per
 - Java (source e target)
 - Il plugin che gestisce i jar in maven

```
<properties>  
  <maven.compiler.source>1.8</maven.compiler.source>  
  <maven.compiler.target>1.8</maven.compiler.target>  
  
  <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>  
</properties>
```

Aggiungere una dependency

- Default (“central”) repository:
 - <https://repo.maven.apache.org/maven2>
- Ricerca di dipendenze:
 - <https://search.maven.org/>, <https://mvnrepository.com/>
- Es: JUnit
 - <https://search.maven.org/artifact/junit/junit/4.12/jar>
 - <https://search.maven.org/artifact/org.junit.jupiter/junit-jupiter-engine/5.3.2/jar>

```
<dependency>  
  <groupId>junit</groupId>  
  <artifactId>junit</artifactId>  
  <version>4.12</version>  
</dependency>
```

```
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-engine</artifactId>  
  <version>5.3.2</version>  
</dependency>
```

Tra le <dependencies>

Vogliamo usare Junit
solo in test,
perciò aggiungiamo:
<scope>test</scope>

Version Control System (VCS)

- Obiettivi
 - Mantenere traccia dei cambiamenti nel codice; sincronizzazione del codice tra utenti
 - Cambiamenti di prova senza perdere il codice originale; tornare a versioni precedenti
- Architettura client/server (CVS, Subversion, ...)
 - Repository centralizzato con le informazioni del progetto (codice sorgente, risorse, configurazioni, documentazione, ...)
 - check-out/check-in (lock del file), branch/merge (conflitti)
- Distributed VCS, architettura peer-to-peer (Git, Mercurial, ...)
 - Repository clonato su tutte le macchine
 - Solo push e pull richiedono connessione di rete

Git

- 2005 by Linus Torvalds et al.
- 24 febbraio 2019: version 2.21
- Si può installare un client dal sito ufficiale
 - <https://git-scm.com/> (SCM: Source Control Management)
- Git è supportato dai principali IDE
- Siti su cui condividere pubblicamente un repository
 - github.com, bitbucket.org, ...

Git shell: alcuni comandi

- clone <url>: clona un repository in locale
- add <filename(s)>: stage per commit
- commit -m "message": copia sul repository locale
- commit -am "message": add & commit
- status: lo stato del repository locale
- push: da locale a remoto
 - push --set-upstream origin <branch>
- pull: da remoto a locale
- log: storico delle commit
- reflog: storico in breve
- reset --hard <commit>: il repository locale torna alla situazione del commit specificato
- branch: lista dei branch correnti
- branch <branch>: creazione di un nuovo ramo di sviluppo
- checkout <branch>: scelta del branch corrente
- merge <branch>: fusione del branch

Nuovo repository Git in Eclipse

- GitHub, creazione di un nuovo repository “xyz”
- Shell di Git, nella directory git locale:
 - `git clone <url xyz.git>`
- Eclipse: creazione di un nuovo progetto
 - Location: directory del repository appena clonato git/xyz
- Il nuovo progetto viene automaticamente collegato da Eclipse al repository Git presente nel folder

Import di un progetto mvn in Eclipse

- File, Import ..., Git, Project from Git
- Clone URI
 - Fornita da GitHub, ad es. <https://github.com/egalli64/ovedX>
 - Bottone “Clone or download” → <https://github.com/egalli64/ovedX.git>
- Import as **general project**
- Right click sul progetto
 - Configure, Convert to Maven project

.gitignore in Eclipse

- Non tutti i file in un progetto vanno salvati in Git
 - Configurazione Eclipse (file “.project”, folder “.settings”, ...)
 - Il folder “target”
- Per ignorare file o interi folder
 - Right-click, (“Team”), “Ignore”
 - Inserire nel file “.gitignore” file o folder che vanno ignorati
- In Eclipse, Git annota le icone di file e folder con
 - punto di domanda: risorsa sconosciuta
 - asterisco: risorsa staged per commit
 - più: risorsa aggiunta a Git ma non ancora tracked
 - assenza di annotazioni: risorsa ignorata

Esempio di file
“.gitignore”

```
/.classpath  
/.project  
/target/  
/.settings/
```

Pull in Eclipse

- Per assicurarsi di lavorare sul codebase corrente, occorre sincronizzarsi col repository remoto via pull
- Right click sul nome del progetto, Team, Pull
(o Pull... per il branch corrente)
- È in realtà la comune abbreviazione dei comandi fetch + merge origin/master

Commit in Eclipse

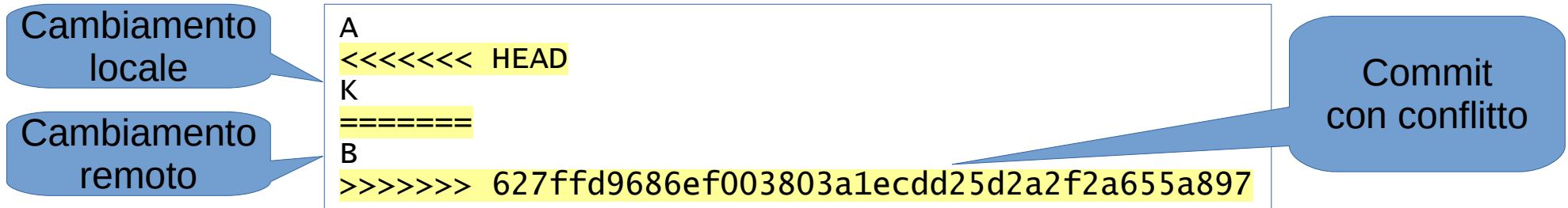
- Autore del commit
 - Window, Preferences, Git, Configuration, User Settings
 - user.email, user.name
- A ogni commit va associato un messaggio, che dovrebbe descrivere il lavoro compiuto
- La prima commit crea il branch “master”, le successive aggiornano il branch corrente
- commit aggiorna il repository locale
- Il repository remoto (GitHub) si aggiorna con push

Push in Eclipse

- Commit aggiorna il repository locale
- Push aggiorna il repository remoto
 - Right click sul nome del progetto, Team, **Push to upstream** (o Push branch 'master' ...)
- Per ridurre il rischio di conflitti, **prima pull**, dopo (e solo se non sono stati rilevati problemi) push
- **Commit and push** è una scorciatoia comune

Conflitti su pull

- Il file hello.txt ha una sola riga: “A”
- L’utente X aggiunge una riga “K” e committa
- L’utente Y fa una pull, aggiunge la riga “B”, committa e fa un push
- Ora, il pull di X causa un auto-merging di hello.txt con un conflitto
- Git chiede di risolverlo editando il file e di committare il risultato



Branch e merge in Eclipse

- Nuovo branch: Team, Switch To, New branch...
 - Basta specificare il nome del nuovo branch
- Selezione branch: Team, Switch To
- Merge branch, dal branch di destinazione:
 - Team, Merge... seleziona il branch di partenza

JDBC

- Connessione a database da Java
- Si aggiunge al proprio progetto Java il jar che implementa JDBC per il database scelto
 - es. Oracle: ojdbc8.jar
- Si usano nel proprio progetto interfacce definite nei package `java.sql`, `javax.sql`

DriverManager

- Servizio di base che gestisce i database driver presenti nel progetto
- `getConnection()`
 - url, secondo le specifiche fornite dal DBMS
 - `jdbc:oracle:thin:@localhost:1521/orclpdb`
 - `jdbc:mysql://localhost:3306/hr?useSSL=false`
 - user
 - password

OracleDataSource

- Definita nel package `oracle.jdbc.pool`
- È il modo preferito per definire un data source Oracle
 - Gestisce automaticamente un pool di connessioni
 - Ma introduce una dipendenza esplicita da Oracle nel codice Java
- La creazione dell'oggetto `OracleDataSource` va completata con chiamate a setter che seguono lo stesso schema del `DriverManager`
 - `setURL()`
 - `setUser()`
 - `SetPassword()`
- La `getConnection()`, di conseguenza, non richiede parametri

Connection

- Media lo scambio di dati tra Java e database
- Estende l'interfaccia AutoCloseable

```
Connection conn = DriverManager.getConnection(url, user, password);
```

```
OracleDataSource ods = new OracleDataSource();  
ods.setURL(url);  
ods.setUser(user);  
ods.setPassword(password);  
  
Connection conn = ods.getConnection();
```

Statement

- Rappresenta un comando da eseguire sul database
 - `execute()` per DDL, true se genera un `ResultSet` associato
 - `executeUpdate()` per DML, ritorna il numero di righe interessate
 - `executeQuery()` per SELECT, ritorna il `ResultSet` relativo
- Generato da un oggetto `Connection` per mezzo del metodo `createStatement()`
- Estende l'interfaccia `AutoCloseable`
- Se lo stesso statement SQL è eseguito più volte, potrebbe essere più efficiente usare un `PreparedStatement`, che può anche gestire parametri IN
- `CallableStatement` è l'interfaccia specifica per chiamare stored procedures

ResultSet

- Una tabella di dati che rappresenta il result set ritornato dal database
- Estende l'interfaccia AutoCloseable
- Per default, non supporta update e può essere percorso solo in modalità forward
- Normalmente ottenuto da uno Statement via `executeQuery()`

```
ResultSet rs = stmt.executeQuery("SELECT coder_id, first_name, last_name FROM coders");
```

SQLException

- Rappresenta un errore generato da JDBC
- Qualcosa non ha funzionato nell'accesso a database, o altri problemi
- Possiamo assumere che tutto il nostro codice JDBC richieda di essere eseguito in blocchi try/catch per questa eccezione

SELECT via JDBC

creazione di un data source

try with resources

executeQuery() on SELECT

```
// OracleDataSource ods = ...  
  
try (Connection conn = ods.getConnection();  
     Statement stmt = conn.createStatement()) {  
    ResultSet rs = stmt.executeQuery("SELECT first_name FROM coders ORDER BY 1");  
  
    List<String> results = new ArrayList<String>();  
    while (rs.next()) {  
        results.add(rs.getString(1));  
    }  
  
    // ...  
}
```

itera sul result set

legge la prima colonna
della riga corrente del
result set come stringa

Transazioni

- By default, una connessione è in modalità autocommit, ogni statement viene committato
- `Connection.setAutoCommit(boolean)`
- `Connection.commit()`
- `Connection.rollback()`

Design pattern

- È una **soluzione** verificata a un **problema comune**
- Progettazione più flessibile e modulare
- Documentazione del codice più intuitiva
- Testi storici
 - A pattern language – Christopher Alexander – 1977
 - Using Pattern Languages for Object-Oriented Programs – Kent Beck, Ward Cunningham – 1987
 - Design Patterns – Erich Gamma et al. (GoF: Gang of Four) – 1994

Definizione

- Nome
 - Descrive il pattern e la sua soluzione in un paio di parole
- Problema
 - Contesto e ragioni per applicare il pattern
- Soluzione
 - Elementi del design, relazioni, responsabilità e collaborazioni
- Conseguenze
 - Risultato, costi e benefici, impatto sulla flessibilità, estensibilità, portabilità del sistema
 - Possibili alternative

Classificazione

- Scopo
 - Creazionali
 - Creazione di oggetti
 - Strutturali
 - Composizione di classi e oggetti
 - Comportamentali
 - Interazione tra oggetti o classi
 - Flusso di controllo
- Raggio d'azione
 - Classe (statico)
 - Ereditarietà
 - Oggetto (dinamico)
 - Associazione
 - Interfacce

Creazionali

- Singleton (uno e uno solo)
- Factory method (da classi derivate)
- Abstract factory (da famiglie di classi)
- Builder (più step di costruzione)
- Prototype (clone)
- ...

Strutturali

- Façade (un oggetto rappresenta un sistema)
- Composite (albero di oggetti semplici e composti)
- Decorator (aggiunge metodi dinamicamente)
- Adapter (adatta l'interfaccia a un'altra esigenza)
- Proxy (un oggetto rappresenta un altro oggetto)
- ...

Comportamentali

- Mediator (interfaccia di comunicazione)
- Observer / Pub-Sub (notifica di cambiamenti di stato)
- Memento (persistenza dello stato)
- Iterator (accesso sequenziale agli elementi)
- Strategy (algoritmo incapsulato in una classe)
- ...

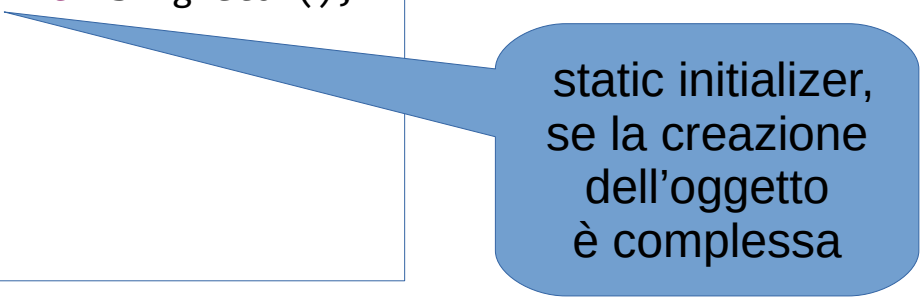
Altri pattern

- Funzionali
 - Generator
- Concorrenza
 - Future e promise
- Architetture
 - Model View Controller (MVC)
 - Client/Server

Singleton

- È necessario che esista un'unica istanza di una classe
- Ctor privato (o protetto), un metodo statico (factory) è responsabile dell'istanziazione e dell'accesso
- Semplice implementazione eager in Java

```
private static final Singleton instance = new Singleton();  
  
private Singleton() {  
}  
  
public static Singleton getInstance() {  
    return instance;  
}
```



static initializer,
se la creazione
dell'oggetto
è complessa

Singleton (lazy)

```
private static Lazy instance;  
private Lazy() {}  
public static synchronized Lazy getInstance() {  
    if(instance == null) {  
        instance = new Lazy();  
    }  
    return instance;  
}
```

Istanza creata solo su richiesta
ma la sincronizzazione costa.
Alternativa: lock tra doppio
check su instance volatile

Instance creata alla
prima chiamata

```
private LazyInner() {}  
private static class Helper {  
    private static final LazyInner INSTANCE = new LazyInner();  
}  
public static LazyInner getInstance() {  
    return Helper.INSTANCE;  
}
```

Strategy

- Modifica dinamicamente un algoritmo
- Il comportamento viene delegato a un'altra classe
- Esempio Java: Comparator per sorting
 - Data una List di Integer
 - Sort con Comparator custom per ordine particolare (prima i numeri dispari e poi quelli pari)

Strategy con Comparator

```
List<Integer> data = Arrays.asList(42, 7, 5, 12);  
data.sort(new OddFirst());  
System.out.println(data);
```

```
class OddFirst implements Comparator<Integer> {  
    @Override  
    public int compare(Integer lhs, Integer rhs) {  
        if (lhs % 2 == 1 && rhs % 2 == 0) {  
            return -1;  
        }  
        if (lhs % 2 == 0 && rhs % 2 == 1) {  
            return 1;  
        }  
        return lhs.compareTo(rhs);  
    }  
}
```