Tecniche di Testing

Esempio Guida

- Si riconsidera l'esempio del *Dictionary* cercando di far sollevare e risolvere in questo caso particolare le problematiche che generalmente si incontrano durante il TDD

- Terminologia
  - OUT: Object Under Test
  - CUT. Class Under Test

Dictionary Step by Step (8)

- Si riprende l'esempio del Dizionario riconsiderando il caso di due traduzioni in inglese per la medesima parola in tedesco. Al solito, prima il test:

```
…
public void testTranslationWithTwoEntries() {
  dict.addTranslation("Buch", "book");
  dict.addTranslation("Buch", "volume");
  trans = dict.getTranslation("Buch");
  assertEquals("book, volume", trans);
}
…
```

- Si è presa una decisione: le traduzioni della medesima parola vengono restituite in una sola stringa usando ',' come separatore.

Test Driven Development & Design Decisions

- Ad un programmatore esperto può sembrare inappropriata la scelta appena fatta:
  - dentro il metodo per ottenere le traduzioni di una parola è stato "cablato" anche il modo in cui vengono presentate
- E' vero! ma non vale la pena di preoccuparsene troppo o di cominciare interminabili discussioni sull'argomento
- Seguendo l'approccio TDD si può rimediare nelle micro-iterazioni successive:
  - errori fatti precedentemente verranno evidenziati da nuovi test
  - fino a quando un nuovo requisito ed il corrispondente test (in questo caso cambiare la "presentazione") non avranno spazio, la soluzione giusta è sempre quella più semplice

Dictionary Step by Step (9)

```
public class Dictionary {
…
  public void addTranslation(String german,
                             String translated) {
     String before = this.getTranslation(german);
     String now;
     if (before == null) {
       now = translated;
     } else {
       now = before + ", " + translated;
     }
     translations.put(german, now);
  }
…
}
```


Le Micro-Iterazioni

- Rivisitiamo il senso di una *micro-iterazione*:
  - un test è motivato direttamente od indirettamente dai requisiti e per poterlo scrivere si è costretti a prendere decisioni sull'interfaccia pubblica della CUT
  - si scrive il test ed quindi tanto codice della CUT quanto basta a compilare ed a far andare a buon fine i test. Questo spesso significa implementazioni parziali dei metodi della CUT
  - se i test vanno a buon fine senza che riescano a rilevare le carenze nel codice attuale, significa che mancano dei test
  - si scrivono nuovi test che falliscono ed evidenziano, come voluto, le carenze del codice attuale
  - si modifica il codice quanto basta a far andare a buon fine anche i nuovi test
  - effettuate le modifiche al codice, si cercano ed eliminano eventuali *duplicazioni* tramite appropriati passi di refactoring

Il Dual-Feedback nel TDD

- L'essenza di questo approccio sta nel doppio feedback che si instaura tra codice e test:
  - i test guidano lo sviluppo del codice nella direzione voluta dai requisiti concentrandosi sull'interfaccia pubblica
  - il codice che pur essendo solo parzialmente implementato riesce a mandare a buon fine i test attuali, stimola la scrittura di nuovi test
- Il procedere per micro-iterazioni assicura la copertura dei test sul codice e la sua correttezza semantica e non solo sintattica

Cosa Testare? (1)

- I test scritti finora si sono concentrati solo sul comportamento dell'OUT visibile esternamente
  - in java questo potrebbe significare tutti i metodi pubblici, protetti o con visibilità di package
  - in pratica conviene testare solo la parte di codice visibile da un ipotetico "client" completamente esterno al codice, e quindi trascurare i metodi protetti

Cosa Testare? (2)

- Non è una regola assoluta, ma il principio generale è:

  quanto più una parte di codice è visibile

  dall'esterno, tanto più deve essere testata

- Motivazioni:
  - ci si concentra sulle specifiche (parti pubbliche) e non sui dettagli
  - i test diventano documentazione delle specifiche mantenuta forzatamente consistente
  - le parti meno visibili del codice sono quelle più facilmente oggetto di ristrutturazioni e quindi coi maggiori costi di manutenzione dei test

Dipendenze da Risorse Esterne (1)

- Si riconsideri la creazione del dizionario a partire da un file di testo:

```
public void testSimpleFile() {
  dict = new Dictionary("C:\\temp\\simple.dic");
  assertTrue(! dict.isEmpty());
}
```

- Un test scritto in questo modo solleva problematiche piuttosto rilevanti:
  - il test diventa dipendente da una risorsa ad esso esterna (il file **`simple.dic`**) che deve essere mantenuta consistente
  - specificando path di file si diviene dipendenti dalla propria piattaforma di sviluppo

Dipendenze da Risorse Esterne (2)

- Problemi analoghi nascono con tutte le risorse esterne
  - ad es. connessioni di rete, server remoti ecc. ecc.
- Al contrario i test migliori sono quelli
  - locali
  - auto-esplicativi
  - auto-contenuti
- In questo caso è possibile rimediare facilmente:
  - si astrarre da **`File`** ad un qualsiasi **`InputStream`**
  - si popola il dizionario direttamente nel test

Rimozione di una Dipendenza (1)

- Si riscrive il test considerando come contenuto del file di testo :

```
Buch=book
Auto=car
```

```
public void testTwoTranslationsFromStream() {
  String dictText="Buch=book\n" + "Auto=car";
  InputStream in =
        new StringBufferInputStream(dictText);
  dic = new Dictionary(in);
  assertFalse(dict.isEmpty());
  assertEquals("book", dict.getTranslation("Buch"));
  assertEquals("car", dict.getTranslation("Auto"));
}
```

- Quindi si chiude il gap sul codice

Dictionary Step by Step (10)

- Non può sfuggire che la sintassi del file di testo coincide esattamente con quella prevista da **`java.util.Properties`**

```
public class Dictionary {
  …
  public Dictionary(InputStream in) throws IOException {
    this.readTranslations(in);
  }
  private void readTranslations(InputStream in)
                                        throws IOException {
    Properties props = new Properties();
    props.load(in);
    Iterator i = props.keySet().iterator();
    while (i.hasNext()) {
      String german = (String)i.next();
      String trans = props.getProperty(german);
      this.addTranslation(german, trans);
    }
  } …
}
```


Dictionary Step by Step (11)

- Riconsiderando due vecchi test
  - **`testTranslationsWithTwoEntries()`**
  - **`testTwoTranslationsFromStream()`**

  è possibile costruirne uno simile per testare il caso di due traduzioni della medesima parola con dizionario letto da file

```
public void testTranslationsWithTwoEntriesFromStream()
                                  throws IOException {
  String dictText="Buch=book\n" + "Buch=volume";
  InputStream in =
        new StringBufferInputStream(dictText);
  dic = new Dictionary(in);
  String trans = dict.getTranslation("Buch");
  assertEquals("book, volume", trans);
}
```

I Test Rilevano le Scelte Sbagliate

- Vicolo cieco: consultare i javadoc del metodo **`load()`** di **`java.lang.Properties`**
  - la lettura da file di due entry con la medesima chiave comporta la sovrascrittura del valore letto con la prima entry da parte di quello letto con la seconda
- E' stata fatta una decisione sbagliata, ma un test ha subito rilevato il problema
- In realtà nasce il sospetto che il parsing di un file di test è una questione non banale che merita una classe apposita

Single Responsibility Principle

Una classe deve avere una ed una sola responsabilità

- La sola **`Dictionary`** si addossava due responsabilità
  - gestire (aggiungere, accedere…) le traduzioni
  - effettuare il parsing di un file con specifica sintassi
- Si sospende la scrittura del test case lasciandolo aperto
  `_testTranslationsWithTwoEntriesFromStream()`
- Quindi passo di *refactoring "Extract Class"* per spostare da Dictionary in una nuova classe **`DictionaryParser`** le funzionalità di parsing
- Al solito, si scrivono prima i test…

Dictionary Step by Step (12)

```
public class DictionaryParserTest
                            extends junit.framework.TestCase {

  private DictionaryParser parser;

  private DictionaryParser createParser(String dictText)
                                    throws IOException {
    Reader reader = new StringReader(dictText);
    return new DictionaryParser(reader);
  }
  private void assertNextTranslation(String german,
                                     String trans)
    throws Exception {
    assertTrue(parser.hasNextTranslation());
    parser.nextTranslation();
    assertEquals(german, parser.currentGermanWord());
    assertEquals(trans, parser.currentTranslation());
  }
…
```


Dictionary Step by Step (13)

```
…
  public void testEmptyReader() throws Exception {
    parser = this.createParser("");
    assertFalse(parser.hasNextTranslation());
  }
  public void testOneLine() throws Exception {
    String dictText = "Buch=book";
    parser = this.createParser(dictText);
    this.assertNextTranslation("Buch", "book");
    assertFalse(parser.hasNextTranslation());
  }
  public void testThreeLines() throws Exception {
    String dictText = "Buch=book\n"+"Auto=car\n"+"Buch=volume";
    parser = this.createParser(dictText);
    this.assertNextTranslation("Buch", "book");
    this.assertNextTranslation("Auto", "car");
    this.assertNextTranslation("Buch", "volume");
    assertFalse(parser.hasNextTranslation());
  }
}
```

Dictionary Step by Step (14)

- A questo punto non esiste più la dipendenza da **`java.util.Properties`**, si può evitare l'utilizzo di **`StringBufferInputStream`** che è deprecato a favore di **`StringWriter`**
- passi successivi (omessi)
  - si chiude il gap sul codice scrivendo **`DictionaryParser`**
  - si modifica di conseguenza **`Dictionary`**
  - si chiude il test che era stato lasciato in sospeso **`testTranslationsWithTwoEntriesFromStream()`**
- Si osservi che i nuovi test case sollevano **`Exception`** e nulla di più specifico…

I Test Sollevano Eccezioni?

- Si! ma è inutile specificarne il tipo nella clausola **`throws`** od anche catturarle all'interno del test a meno che non si stia testando proprio il sollevamento di una eccezione
- quindi i test che sollevano eccezioni usano la clausola: **`throws Exception`**
- Motivazioni:
  - specificando il tipo delle eccezioni o catturandole
    - non si aggiungerebbe nulla alla capacità del test di fornire indicazioni sul comportamento dell'OUT
    - la segnatura del test risulterebbe accoppiato al tipo stesso

Test di una Eccezione

- Per testare che una eccezione attesa venga effettivamente sollevata, basta seguire questo schema che utilizza **`fail()`**

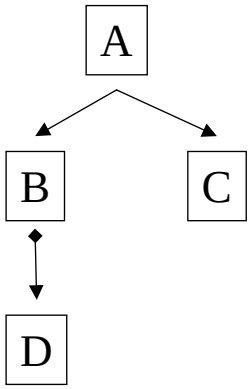
```
public void testEmptyLine() throws Exception {
  String dictText = "Buch=book\n" +"\n"+ "Auto=car";
  parser = this.createParser(dictText);
  this.assertNextTranslation("Buch", "book");
  try {
    parser.nextTranslation();
    fail("DictionaryParserException expected");
  } catch (DictionaryParserException expected) {}
  this.assertNextTranslation("Auto", "car");
  assertFalse(parser.hasNextTranslation());
}
```


I Problemi Dovuti alle Dipendenze

- Si sono già visti i problemi che comporta la dipendenze di una CUT da risorse esterne
  - ad es. nel caso che la CUT utilizzi una connessione di rete, i test possono dipendere dalla possibilità di instaurare la connessione, compromettendone affidabilità ed efficienza
- In generale tutte le dipendenze minano la caratteristiche fondamentali dei test, ed in particolare
  - località
  - semplicità
  - auto-contenimento
  - efficienza
- Nella POO esistono anche dipendenze tra classi, anzi…

Test e Dipendenze (1)

- Una CUT può esibire dipendenze che rendono problematica la scrittura dei test
- Dipendenze
  - verso risorse esterne
    - file, connessioni di rete, server
  - verso altre classi
    - come scrivere i test?
      - top-down
      - bottom-up



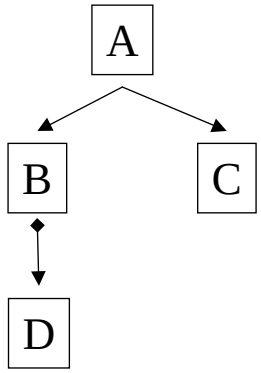
- Ovviamente le risorse esterne sempre da classi sono rappresentate

Test e Dipendenze (2)

- Per garantire l'efficacia dei test si devono "rompere" le dipendenze con apposite tecniche per rimpiazzare le risorse da cui la CUT è dipendente
- Terminologia:
  - stub: porzione non completamente implementata ma che è destinata ad evolvere in una porzione del codice prodotto
  - dummy: rimpiazza la vera implementazione ma solo per il testing. Non farà parte del codice di produzione
  - mock: rimpiazza la vera implementazione ma solo per facilitare il testing. Non farà parte del codice di produzione. Rispetto ad un dummy non si limita ad rimpiazzare una porzione di codice, ma aggiunge altre funzionalità per facilitare il testing della CUT

Test e Dipendenze (3)

- Ad esempio, per risolvere le dipendenze verso altre classi con un approccio top-down basta articolare il testing in questi passi:
  - A (B, C con dummy)
  - C
  - B (D con dummy)
  - D



Un Primo Esempio

- Si vuole programmare un convertitore in EUR. Il cambio corrente si suppone fornito da una istanza di **`ExchangeRateProvider`**
- Al solito, prima i test di **`EuroCalculator`**:

```
public class EuroCalculatorTest extends TestCase {
  private final static double ACCURACY = 0.00001;
  public void testEUR2EUR() {
    double result= new EuroCalculator().valueInEuro(2.0,"EUR");
    assertEquals(2.0, result, ACCURACY);
  }
  public void testUSD2EUR() {
    double result= new EuroCalculator().valueInEuro(1.5,"USD");
    assertEquals(1.6986, result, ACCURACY);
  }
}
```

Conseguenze della Dipendenza

- **`EuroCalculator`** dipende dalla classe **`ExchangeRateProvider`**
- Inoltre tale classe ottiene il cambio da un servizio in rete, quindi una risorsa esterna, ed ha un comportamento non deterministico che si propaga alla CUT
- Il test può fallire per motivi esterni al test, ad esempio la non disponibilità del servizio in rete

```
public class ExchangeRateProvider {
   public double getRateFromTo(String from, String to)
                    throws ServerNotAvailableException {
     double retrievedRate = 0.0;
     /*    codice di connessione alla rete   */
     /* potenzialmente lento ed inaffidabile */
     return retrievedRate;
   }
}
```


Rimozione di una Dipendenza (1)

- un dummy per **`ExchangeRateProvider`** che ne simuli il comportamento

```
public class DummyProvider extends ExchangeRateProvider {

  private double dummyRate;

  public DummyProvider(double dummyRate) {
    this.dummyRate = dummyRate;
  }

  public double getRateFromTo(String from, String to) {
    return dummyRate;
  }

}
```

Rimozione di una Dipendenza (2)

- i test di **`EuroCalculator`** vanno riscritti prevedendo il passaggio dell'**`ExchangeRateProvider`** nel costruttore:

```
public class EuroCalculatorTest extends TestCase {
  private final static double ACCURACY = 0.00001;
  public void testEUR2EUR() {
    ExchangeRateProvider provider = new DummyProvider(1.0);
    double result = new EuroCalculator().valueInEuro(2.0,"EUR",prov);
    assertEquals(2.0, result, ACCURACY);
  }
  public void testUSD2EUR() {
    ExchangeRateProvider provider = new DummyProvider(1.1324);
    double result = new EuroCalculator().valueInEuro(1.5,"USD",prov);
    assertEquals(1.6986, result, ACCURACY);
  }
}
```


Testabilità del Codice

- Il banale esempio visto chiarisce la tecnica: i dummy servono per rimpiazzare le interazioni dell'OUT con l'esterno di modo da cablare nei test stesso i valori che gli si vogliono far ottenere da tali interazioni
- Ora l'OUT è completamente sotto il controllo del test che riacquista le sue caratteristiche fondamentali
- Si evidenzia una nuova proprietà del codice, la *testabilità,* ovvero quanto il codice si presta ad essere testato
  - per applicare il dummy senza modificare il codice di **`EuroCalculator`** è necessario che riceva l'istanza di **`ExchangeRateProvider`** nel suo costruttore

Un Secondo Esempio

- In molte applicazione è necessario effettuare il logging di una grande varierà di eventi. Si vuole implementare la funzionalità di *logging* secondo una interfaccia standard:

```
public interface Logging {
   int DEFAULT_LOGLEVEL = 2;
   void log(int logLevel,String message);
   void log(String message);
}
```


LogServerTest

- **LogServer** implementerà le funzionalità di logging dettate dall'interfaccia. Al solito, prima i test:

```
public class LogServerTest extends TestCase {
…
   public void testSimpleLogging() {
     Logging logServer = new LogServer("log.test");
       logServer.log(0, "Line One");
       logServer.log(1, "Line Two");
       logServer.log("line Three");
       // come scrivere le asserzioni qui?
   }
…
}
```

Una CUT Difficile

- Il punto è che l'interfaccia non prevedeva metodi per accedere quanto già ricevuto dal **`LogServer`**, ne sembra appropriato prevederne solo per i test
- si può rimediare:
  - nel costruttore si astrae il parametro passando dal nome del file ad un **`Writer`**
  - si è però introdotta una dipendenza da **`Writer`**
  - si può rimuovere immediatamente con un dummy

DummyPrintWriter

- Un dummy per **`PrintWriter`**:

```
import java.util.*;
public class DummyPrintWriter extends PrintWriter {
  private List logs = new ArrayList();
  public DummyPrintWriter() {
    super((OutputStream) null);
  }
  public void println(String logString) {
    logs.addElement(logString);
  }
  public String getLogString(int pos) {
    return (String) logs.get(pos);
  }
}
```

Una Soluzione Approssimativa

- Ora è possibile scrivere le asserzioni:

```
public class LogServerTest extends TestCase {
…
  public void testSimpleLogging() {
    DummyPrintWriter writer = new DummyPrintWriter();
    Logging logServer = new LogServer(writer);
    logServer.log(0, "first line");
    logServer.log(1, "second line");
    logServer.log("third line");
    assertEquals("0: first line", writer.getLogString(0));
    assertEquals("1: second line", writer.getLogString(1));
    assertEquals("2: third line", writer.getLogString(2));
  }
…
}
```

- Almeno un problema: ma è sicuro che **`LogServer`** chiama solo il metodo **`println()`?**

Interfaccia `Logger`

- In generale le interazioni tra **`LogServer`** ed il **`Writer`** potrebbero essere diverse da quelle previste e comunque questa necessità di vincolare lo sviluppo della CUT ai suoi test denuncia qualche problema di progettazione
- è possibile rimediare; l'assunzione implicita sul **`LogServer`** (dover usare **`println()`**) può essere esplicitata con un interfaccia apposita:
  - si astrae da **`Writer`** a **`Logger`**

```
public interface Logger {
  void logLine(String logString);
}
```

  - si crea un dummy per la nuova interfaccia

DummyLogger

- Il nuovo dummy è più semplice:

```
import java.util.*;
public class DummyLogger implements Logger {
  private List logs = new ArrayList();

  public void logLine(String logString) {
    logs.addElement(logString);
  }
  public String getLogString(int pos) {
    return (String) logs.get(pos);
  }
}
```


LogServer

- non resta che scrivere la CUT:

```
public class LogServer implements Logging {
   private Logger logger;
   public LogServer(Logger logger) {
     this.logger = logger;
   }
   public void log(int logLevel, String message) {
     String logString = logLevel + ": " + message;
     logger.logLine(logString);
   }
   public void log(String message) {
     this.log(DEFAULT_LOGLEVEL, message);
   }
}
```

Testabilità = Qualità del Codice

- Potrebbe apparire che gli sforzi fatti per garantire la testabilità di **`LogServer`** siano eccessivi
- In realtà è accaduto che nel tentativo di garantirne la testabilità della CUT si è migliorato il suo progetto:
  - si è rimossa la sua dipendenza da classi esterne
  - tale dipendenza è stata mascherata dietro una interfaccia **`Logger`**
  - è possibile implementare diversi tipi di logger senza toccare **`LogServer`**
  - si sono già testate **`LogServer`** e le sue *interazioni* con il **`Logger`** comunque venga implementato

Dependency Inversion Principle

I moduli ad alto livello non devono dipendere da quelli a basso livello. Tutti devono dipendere da astrazioni (interfacce).

- In questo esempio **`LogServer`** deve dipendere da **`Logger`** e non da **`Writer`**

Mock Objects

- I Mock objects si differenziano dai dummy
  - permettono di registrare le interazioni attese con la CUT
  - quando hanno effettivamente luogo, fanno asserzioni per stabilire che le interazioni con la CUT si svolgono esattamente come atteso
- Quindi, rispetto ai dummy
  - fanno asserzioni
  - sono più funzionali e versatili
  - sono leggermente più complessi
- Si parla anche di *endo-testing*

Da `DummyLogger` a `MockLogger` (1)

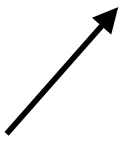
```
import java.util.*;
import junit.framework.Assert;

public class MockLogger implements Logger {
  private List expectedLogs = new ArrayList();
  private List actualLogs = new ArrayList();

  public void addExpectedLine(String logString) {
     expectedLogs.add(logString);
  }

  public void logLine(String logLine) {
   …
  }
  public void verify() {
   …
  }
}
```

Da DummyLogger a MockLogger (2)

```
import java.util.*;
import junit.framework.Assert;
public class MockLogger implements Logger {…
  public void logLine(String logLine) {
    Assert.assertNotNull(logLine);
    if (actualLogs.size() >= expectedLogs.size()) {
      Assert.fail("Too many log entries");
    }
    int currentIndex = actualLogs.size();
    String expectedLine =
      (String) expectedLogs.get(currentIndex);
    Assert.assertEquals(expectedLine, logLine);
    actualLogs.add(logLine);
  }…
}
```

Da DummyLogger a MockLogger (3)

```
import java.util.*;
import junit.framework.Assert;
public class MockLogger implements Logger {…
  public void verify() {
    if (actualLogs.size() < expectedLogs.size()) {
      Assert.fail(
        "Expected "+expectedLogs.size()+" log entries“
        +
        " but encountered "+ actualLogs.size()
      );
    }
  }…
}
```



LogServerTest con MockLogger (1)

```
public class LogServerTest extends TestCase {
  private LogServer logServer;
  private MockLogger logger;

  protected void setUp() {
    logger = new MockLogger();
    logServer = new LogServer(logger);
  }

  public void testLoggingWithModule() {
    logger.addExpectedLine("test(0): Line One");
    logServer.log(0, "Line One", "test");
    logger.verify();
  } …
}
```


LogServerTest con MockLogger (2)

```
public class LogServerTest extends TestCase {
…
   public void testSimpleLogging() {
     logger.addExpectedLine("(0): Line One");
     logger.addExpectedLine("(1): Line Two");
     logger.addExpectedLine("(2): Line Three");
     logServer.log(0, "Line One");
     logServer.log(1, "Line Two");
     logServer.log("Line Three");
     logger.verify();
   }
…
}
```

Test Case con Mock Object

- I passi che esegue un test-case che una un mock sono sempre gli stessi
  - creare uno o più mock object
  - inizializzare lo stato dei mock object
  - inizializzare le *expectation* dei mock object
  - invocare il codice da testare passando riferimenti ai mock object come parametro
  - svolgere eventuali test tradizionali sull'OUT
  - invocare il metodo **`verify()`** dei mock objects

Come Mettere i Dummy/Mock Object al Posto delle Classi da cui la CUT Dipende?

- Almeno due possibilità
  - la CUT possiede un costruttore dove riceve l'istanza della classe da cui dipende
  - i metodi della CUT prevedono la possibilità di specificare l'istanza della da cui dipende
- non sempre è possibile, perché molte classi esistenti sono scritte senza considerare la testabilità
- ad ogni modo la diffusione dei pattern stile *factory* ed il principio del DIP favorisce la testabilità almeno del codice di qualità

Mock Objects Preconfezionati

- I mock sono riutilizzabili
  - librerie
    - Mock di librerie standard java.*, javax.*
  - generatori di Mock
    - statici
      - MockMaker
      - MockCreator
    - dinamici
      - EasyMock

Bilancio di Dummy e Mocks Object (1)

- Vantaggi dei Dummy:
  - permettono di “isolare” la CUT garantendo località
  - rendono più semplice la creazione delle fixture
  - aumentano la riusabilità dei test
  - consentono di raggiungere l’efficienza richiesta ai test
  - rendono semplice simulare certi tipi di errori da parte delle risorse rimpiazzate
  - permettono un approccio top-down al testing
  - permettono di sviluppare e testare la CUT prima ancora che sia disponibile una risorsa esterna da cui dipende (con API già nota)
  - indirettamente, spingono verso la qualità del codice della CUT

Bilancio di Dummy e Mocks Object (2)

- Vantaggi dei Mock: quelli dei dummy ed in più:
  - consentono di testare le interazioni dell'OUT con le risorse da cui dipende. Quindi si parla di "test dall'interno"
  - facilitano il riutilizzo del codice di test, ed una volta appreso il *pattern*, favoriscono la leggibilità dei test

Bilancio di Dummy e Mocks Object (3)

- Svantaggi:
  - dummy e mock possono contenere errori, anche se con probabilità contenuta
  - i dummy non consentono di testare le interazioni con la CUT
  - i mock consentono di testare le interazioni di coppia, non quelle in cui intervengono più di due oggetti alla volta
  - sia i dummy che i mock devono inseguire l'interfaccia delle classi simulati. Questo comporta la necessità di mantenerli coerenti
  - se le interazioni cambiano, perché cambia l'implementazione della CUT, bisogna riscrivere i test e cambiare anche il mock. Per questo motivo il loro utilizzo va limitato alle implementazioni relativamente stabili
  - il costo di scrittura dei dummy/mock è più facilmente sostenuto nel contesto di un approccio TDD piuttosto che tradizionale

A Domanda Pratica Segue Risposta Pragmatica

- Che cosa testare?
- Quanto deve essere lungo un test?
- Come fare le asserzioni? cablando i risultati attesi, non calcolandoli nel test
- Come testare le eccezioni e scrivere test per metodi che sollevano eccezioni
- Come trattare le dipendenze
  - verso altre classi
  - verso risorse esterne come file, connessioni di rete
- Come organizzare i test? classi **`AllTests`** ma anche suite ad hoc
- Dove mettere le classi di test
  - stesso package
  - diverso package
  - stesso package logico ma diverso path fisico
- Quanto testare?
  - threshold values and equivalence class,
  - code coverage, tools

Riferimenti

- Siti:
  - *http://www.junit.org*
  - *http://www.mockobjects.com*
- Libri:
  - "Unit Testing in Java" di Johannes Link
  - "JUnit in Action" di Vincent Massol