Introduzione al Testing e al Test Driven-Development

Il Perché del Testing

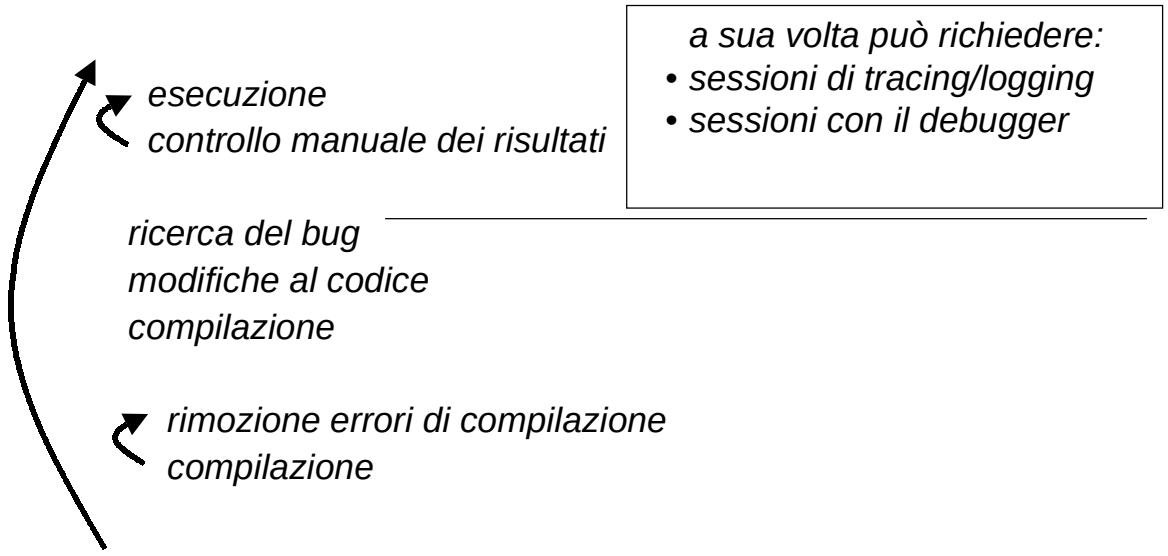
- I programmi sono descrizioni “statiche” a cui possono corrispondere molteplici esecuzioni “dinamiche”
- I compilatori moderni sono in grado di indicare esattamente posizione e motivo degli errori di compilazione
- Al contrario i compilatori non possono prevedere come evolverà l’esecuzione di un programma e non sono in grado di individuare gli errori del programmatore (né possono sapere cosa voleva fare)
- In sintesi:
  - il compilatore ci aiuta sugli aspetti statici (ad es. tipi in C ed in Java)
  - il compilatore non dice nulla o quasi sugli aspetti dinamici

I Bug

- I bug sono errori nell'evoluzione dinamica di un programma su cui il compilatore non ha potuto prevedere e dire nulla
- Il debugging è completamente a carico del programmatore
- Il costo di debugging è ritenuto di gran lunga la componente principale nel costo dei moderni progetti software

Ciclo di Debugging

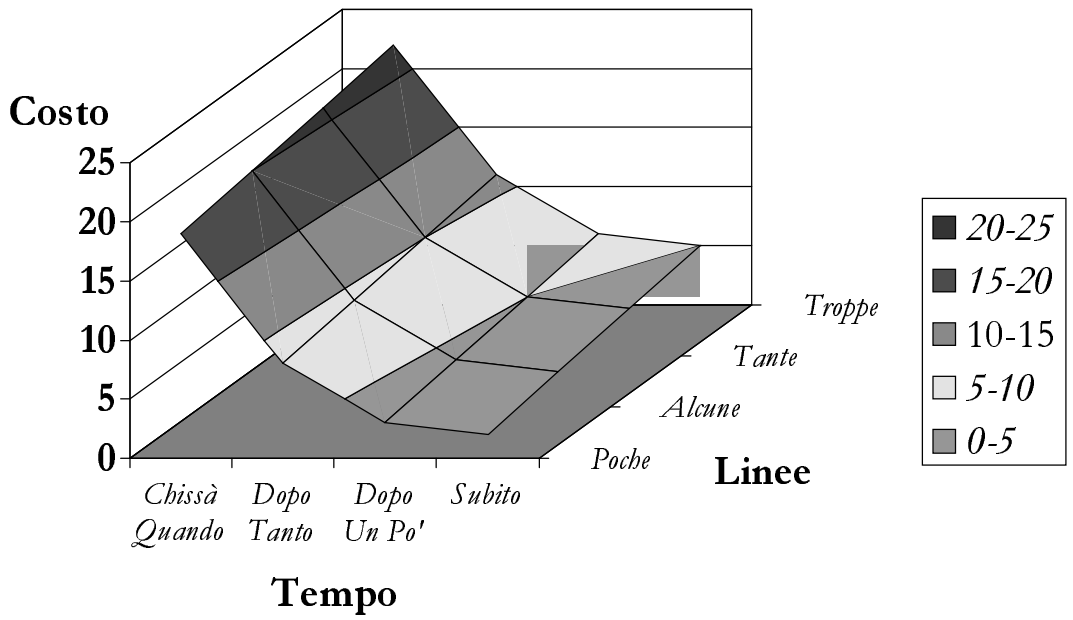
- Come effettuate il debugging di un programma che compila? Con estenuanti cicli:



Quanto Costa Correggere un Bug

- E' ormai risaputo che i programmatori spendono la maggior parte del tempo per il debugging del codice
- E' anche noto che il costo della correzione di bug dipende da almeno due grandezze
  - le "dimensioni" del contesto
    - numero di linee di codice in cui il bug può annidarsi
  - il "tempo" che il bug impiega per manifestarsi
    - misura temporale di quanto dista la causa del bug (durante un'esecuzione del codice) ed il rilevamento dei suoi effetti

Costo di un Bug e "Località"



Debugging e Paradigma di Programmazione

- Attenzione: il costo di un ciclo di debugging dipende da tanti aspetti, tra i quali certamente conta molto anche il linguaggio di programmazione (e soprattutto il suo paradigma)
- Con un linguaggio funzionale?
- Con un linguaggio Object-Oriented?
- Con quale paradigma costa meno e perché?

I Test

- Esistono diversi tipi di test
- Consideriamo gli *Unit Test*, ovvero test sui singoli *frammenti* di un sistema piuttosto che sull'intero sistema
- Concettualmente un test si articola in questi passi
  - mettere un "frammento" del sistema in un stato noto
  - inviare una serie di messaggi noti
  - controllare che alla fine il sistema si trovi nello stato atteso
- Praticamente codifichiamo i test nel medesimo linguaggio di programmazione utilizzato per lo sviluppo
- I Test sono un strumento per avere "garanzie" sull'evoluzione dinamica del nostro codice

Primo Esempio di Test

- Si vuole programmare un dizionario per tradurre parole in tedesco in un altro linguaggio, ad esempio inglese. Il dizionario, modellato da una classe *Dictionary*, è inizializzato con un file di testo e può essere interrogato per conoscere la traduzione di una parola tedesca. Sono possibili diverse traduzioni della stessa parola.

Translation.java

```
/**
 * Rappresenta una possibile traduzione di una parola tedesca
 */
public class Translation {

  private String germanWord;
  private String translation;

  public Translation(String germanWord,
                     String translation) {
     this.germanWord = germanWord;
     this.translation = translation;
  }

  public String getGermanWord() {
     return germanWord;
  }

  public String getTranslation() {
     return translation;
  }
}
```

Dictionary.java (1)

```
import omessi
/**
 * Dizionario per tradurre parole in tedesco.
 * Viene inizializzato con un file di testo.
 */
public class Dictionary {
  private List entries = new ArrayList();

  public Dictionary(String filename)
       throws IOException {

    this.initializeFromReader(
      new BufferedReader(new FileReader(filename))
    );

  }
  …altri metodi…
}
```


Dictionary.java (2)

```
/**
 * Fornisce la traduzione di una parola tedesca.
 * Se ci sono diverse alternative, vengono restituite
 * in unica stringa usando la virgola come separatore.
 */
  public String getTranslations(String germanWord) {
    StringBuffer translations = new StringBuffer();
    Iterator i = entries.iterator();
    while (i.hasNext()) {
      Translation each = (Translation) i.next();
      if (each.getGermanWord().equals(germanWord)) {
        if (translations.length() > 0) {
          translations.append(", ");
        }
        translations.append(each.getTranslation());
      }
    }
    return translations.toString();
  }
```

Dictionary.java (3)

```
/**
 * Il file di testo da leggere consiste di 0 - N linee.
 * Ogni linea contiene una traduzione nella forma:
 * '<parolaTedesca>=<traduzione>'
 */
  private final void initializeFromReader(BufferedReader aReader)
    throws IOException {
    String line = aReader.readLine();
    while(line != null) {
      int index = line.indexOf('=');
      if (index != -1) {
        String germanWord = line.substring(0, index);
        String translation = line.substring(index+1, line.length());
        Translation entry = new Translation(germanWord, translation);
        entries.add(entry);
      }
      line = aReader.readLine();
    }
  }
}
```


Creiamo un Test

- Scriviamo un frammento di codice per testare il corretto funzionamento di `Dictionary`
  - Abbiamo bisogno di un file di testo contenente il dizionario, ad esempio `C:\temp\dictionary.txt`
    `Wort=word`
  - Lo usiamo per creare un'istanza di `Dictionary`
  - Interroghiamo l'oggetto e controlliamo i risultati

DictionaryTester.java (1)

```
public static void testCase1() {
    String filename = "C:\\temp\\dictionary.txt";
    try {
1       PrintWriter writer =
           new PrintWriter(new FileOutputStream(filename));
        writer.println("Wort=word"); writer.close();
2       Dictionary dictionary = new Dictionary(filename);
        String t = dictionary.getTranslations("Wort");
        if (!t.equals("word")) {
          System.out.println("TestCase 1: fallito."+
3             "Traduzione trovata: " + t);
        } else {
           System.out.println("TestCase 1: successo.");
        }
    } catch (Exception ex) {
        System.out.println("TestCase 1: fallito. "+
                           " Rilevata eccezione inattesa.");
        System.out.println(ex.toString());
    }
}
```


DictionaryTester.java (2)

Per mandarlo in esecuzione:

```
public class DictionaryTester {

  /**
   * Esegue tutti i test case per la classe Dictionary
   */
    public static void main(String[] args) {
        testCase1();
        testCase2();
        …
    }

    public static void testCase1() {
      …
    }

    public static void testCase2() {
    }
    …
}
```

Conseguenze del Testing

- Conseguenze immediate:
  - se il test ha successo abbiamo una garanzia sul comportamento dinamico del codice
  - se il test non ha successo abbiamo ottime probabilità che l'errore sia facilmente localizzabile (forte località)
  - se il test smette di funzionare in seguito ad una modifica del codice, con grande probabilità l'errore è dovuto alla modifica stessa e là va ricercato

Perchè (quasi) Nessuno Scrive i Test

- Non è vero in tutti i settori, comunque le obiezioni più frequenti sono:
  - *convinzione di perdere tempo*
    - è vero il contrario, attenzione al circolo vizioso:
      mancanza di tempo -> meno test -> più errori -> più debugging -> meno tempo
  - *scrittura tediosa*
    - il testing è un'arte creativa tanto quanto la programmazione. I test vanno scritti diversamente di caso in caso e sono fortemente accoppiati al codice da testare
  - *tanto lo faranno altri: il gruppo di testing*
    - ma la scrittura dei test non può essere isolata dalla conoscenza dei dettagli interni che l'autore possiede. Confronta con il grafico del costo di un bug ed infine TDD>>
  - *educazione* teorica lontana dalla esigenze pratiche
  - *mancanza di adeguati strumenti di supporto*

Automazione dei Test

- I test devono essere
  - automatici (per mantenere rapido il ciclo di feedback)
    - devono essere eseguiti molte volte al giorno
  - efficienti
    - devono essere convenienti rispetto alle ispezioni manuali
  - isolati e che garantiscano località degli errori
    - dal fallimento di un test alla rimozione dell'errore deve trascorre pochissimo tempo grazie alla località errori
  - ed inoltre:
    - separati dal codice applicativo
    - eseguibili e verificabili separatamente
    - raggruppabili a piacimento in "suite"
- Esistono vari framework per il assistere il programmatore nel testing. Il più noto è ***JUnit*** *(http://www.junit.org)*

JUnit: Test della Classe `Vector` (1)

```
package junit.samples;
import junit.framework.*;
import java.util.Vector;
/**
 * A sample test case, testing java.util.Vector
 */
public class VectorTest extends TestCase {
  protected Vector fEmpty;
  protected Vector fFull;
  public static void main (String[] args) {
      junit.textui.TestRunner.run (suite());
  }
  public static Test suite() {
      return new TestSuite(VectorTest.class);
  }
  metodo setUp() omesso
  altri test case omessi
}
```

JUnit: Fixture per il Test di `Vector` (2)

```
…
   protected Vector fEmpty;
   protected Vector fFull;


   protected void setUp() {
       fEmpty= new Vector();
       fFull= new Vector();
       fFull.addElement(new Integer(1));
       fFull.addElement(new Integer(2));
       fFull.addElement(new Integer(3));
   }
```

altri test case omessi

…

JUnit: Alcuni Test Case di `Vector` (3)

```
…
   public void testCapacity() {
       int size= fFull.size();
       for (int i= 0; i < 100; i++)
              fFull.addElement(new Integer(i));
       assertTrue(fFull.size() == 100+size);
   }
   public void testClone() {
       Vector clone= (Vector)fFull.clone();
       assertTrue(clone.size() == fFull.size());
       assertTrue(clone.contains(new Integer(1)));
   }
   public void testContains() {
       assertTrue(fFull.contains(new Integer(1)));
       assertTrue(!fEmpty.contains(new Integer(1)));
   }
…
```

JUnit: Alcuni Test Case di `Vector` (4)

```
public void testElementAt() {
    Integer i = (Integer)fFull.elementAt(0);
    assertTrue(i.intValue() == 1);
    try {
          fFull.elementAt(fFull.size());
    } catch (ArrayIndexOutOfBoundsException e) {
          return;
    }
    fail("Should raise an ArrayIndexOutOfBoundsException");
}
public void testRemoveAll() {
    fFull.removeAllElements();
    fEmpty.removeAllElements();
    assertTrue(fFull.isEmpty());
    assertTrue(fEmpty.isEmpty());
}
public void testRemoveElement() {
    fFull.removeElement(new Integer(3));
    assertTrue(!fFull.contains(new Integer(3)) );
}
```


JUnit: SimpleTest.java (1)

```
package junit.samples;
import junit.framework.*;

public class SimpleTest extends TestCase {
  protected int fValue1;
  protected int fValue2;
  protected void setUp() {
      fValue1= 2;
      fValue2= 3;
  }
  public void testEquals() {
      assertEquals(12, 12);
      assertEquals(12L, 12L);
      assertEquals(new Long(12), new Long(12));
      assertEquals("Size", 12, 13);
      assertEquals("Capacity", 12.0, 11.99, 0.0);
  }
  omessi altri test case, main() e suite()
}
```

JUnit: SimpleTest.java (2)

```
public void testAdd() {
    double result= fValue1 + fValue2;
    // forced failure result == 5
    assertTrue(result == 6);
}
public void testDivideByZero() {
    int zero= 0;
    int result= 8/zero;
}
```


JUnit: Test Suite di SimpleTest.java

```
public static Test suite() {
   /** the type safe way
   TestSuite suite = new TestSuite();
   suite.addTest(
       new SimpleTest("add") {
              protected void runTest() { testAdd(); }
       }
   );
   suite.addTest(
       new SimpleTest("testDivideByZero") {
              protected void runTest() { testDivideByZero();
   }
       }
   );
   return suite; */

   /** the dynamic way */
   return new TestSuite(SimpleTest.class);
}

public static void main (String[] args) {
   junit.textui.TestRunner.run(suite());
}
```

Altre Conseguenze del Testing

- Alcune conseguenze meno immediate.
  - Se volessimo introdurre diversi test case per coprire la possibilità di
    - vari numeri di elementi nel file (0, 1, 2 ed N)
    - più traduzioni della stessa parola
    - parole che cominciano con una maiuscola,
    - ecc. ecc.

    ci accorgiamo che il progetto attuale non va bene
    - bisogna avere diversi file di testo con le varie possibilità
  - il problema si può evitare permettendo ad oggetti `Dictionary` di aggiungere oggetti `Translation` senza bisogno di ricorrere ai file

Test come Guida allo Sviluppo

- Ma simili carenze di progettazione, si sarebbero verificate se avessimo scritto i test prima ancora di scrivere il codice principale???
- Se scriviamo il codice di test prima del codice stesso siamo costretti a:
  - chiarire quali sono i metodi visibili all'esterno (pubblici e protetti) perché il codice di test è trattato esattamente come qualsiasi altro codice "cliente" esterno alla classe
  - chiarire la semantica dei metodi
  - pensare ai possibili errori e chiarire il comportamento atteso in presenza di errori
  - semplificare al massimo l'utilizzo

Test Driven Development

- Promuove l'uso dei test non solo per le ragioni tradizionali ma anche come strumento di progettazione
  - i test guida lo sviluppo verso codice che sia semplice, facilmente testabile e di qualità
- Prevede i seguenti passi:
  - prima di scrivere il codice stesso, scrivere i test
  - creare implementazioni banali (ad esempio i metodi restituiscono costanti) del codice per compilare i test
  - ovviamente i test falliscono perché mancano delle parti di codice
  - per tutte le micro-iterazioni necessarie (di 10 minuti)
    - un test fallisce
    - modificare il codice finchè il test ha successo
    - rimuovere tutte le duplicazioni nel codice

Dictionary Step by Step (1)

- Prima micro-iterazione, decidiamo solo il nome del test

```
public class DictionaryTest extends TestCase {
}
```

- a questo punto proviamo il test ed otteniamo un errore per mancanza di test case. Eccone uno:

```
public void testCreation() {
   Dictionary dict = new Dictionary();
}
```

- non compila perché la classe **`Dictionary`** non esiste
- l'obiettivo è farlo compilare il più velocemente possibile

Dictionary Step by Step (2)

- Per compilare basta aggiungere

```
public class Dictionary {
}
```

- Si prova il test che va a buon fine. Sappiamo creare un dizionario. Ora proviamo almeno a verificare che non appena creato sia vuoto:

```
public void testCreation() {
   Dictionary dict = new Dictionary();
   assertTrue(dict.isEmpty());
}
```

- attenzione: abbiamo preso una decisione di progetto. Il metodo **`isEmpty()`** diventa parte dell'interfaccia pubblica
- non compila perché il metodo **`isEmpty()`**non esiste
- l'obiettivo è farlo compilare il più velocemente possibile

Dictionary Step by Step (3)

- Per compilare basta aggiungere

```
public class Dictionary {
   public boolean isEmpty() {
      return false;
   }
}
```

- compila, ma abbiamo deliberatamente scelto `false` per far fallire il test ed aumentare la fiducia nei test stessi. In questo modo siamo costretti a pensare prima di farlo andare a buon fine con

```
public class Dictionary {
   public boolean isEmpty() {
      return true;
   }
}
```

- il test va a buon fine, ma costringiamolo a fallire con dizionari non vuoti.
- siamo costretti a pensare come aggiungere elementi al dizionario
- scriviamo prima il test per aggiungere un elemento

Dictionary Step by Step (4)

```
public void testAddTranslation() {
   Dictionary dict = new Dictionary();
   dict.addTranslation("Buch", "book");
   assertFalse(dict.isEmpty());
}
```

- non compila perché manca il metodo **`addTranslation()`**. Facciamolo compilare il più velocemente possibile:

```
public class Dictionary {
   private boolean empty = true;
   public boolean isEmpty() {
       return this.empty;
   }
   public void addTranslation(String german,
                              String translation) {
       this.empty = false;
   }
}
```


Dictionary Step by Step (5)

- compila ed i test vanno a buon fine. Ma è chiaro che manca una parte di implementazione senza che i test lo rilevino. Chiudiamo il gap velocemente ampliando prima il test per farlo fallire:

```
public void testAddTranslation() {
   Dictionary dict = new Dictionary();
   dict.addTranslation("Buch", "book");
   assertFalse(dict.isEmpty());
   String trans = dict.getTranslation("Buch");
   assertEquals("book", trans);
}
```

Dictionary Step by Step (6)

- finalmente il test sembra sensato. Facciamolo andare a buon fine il più velocemente possibile:

```
public class Dictionary() {
  public String getTranslation(String german)
  {
     return “book”;
  }
}
```

- di nuovo il test ha buon esito senza che l’implementazione sia finita. Facciamolo fallire.

Dictionary Step by Step (7)

```
public void testAddTranslation() {
  Dictionary dict = new Dictionary();
  dict.addTranslation(“Buch”, “book”);
  dict.addTranslation(“Auto”, “car”);
  assertFalse(dict.isEmpty());
  assertEquals(“book”, dict.getTranslation(“Buch”));
  assertEquals(“car”, dict.getTranslation(“Auto”));
}
```

- ora tutto compila ma il test rileva l’implementazione non corretta. Chiudiamo il gap sul codice

Dictionary Step by Step (8)

```
import java.util.*;

public class Dictionary {
   private Map translations = new HashMap();

   public void addTranslation(String german,
                              String translated){
       translations.put(german, translated);
   }

   public String getTranslation(String german) {
       return (String) translation.get(german);
   }

   public boolean isEmpty() {
       return translations.isEmpty();
   }
}
```

- ora i test vanno a buon fine. E le parole con più traduzioni?
- … da continuare …

Il Gioco del Test Driven Development

- E’ come se ci fossero due antagonisti in un gioco:
  - il tester
  - il coder
- Il tester ed il coder giocano uno alla volta. Comincia sempre il tester
- Il tester ha il compito di scrivere test che fanno fallire il codice il più velocemente possibile
- Il coder ha il il compito di cambiare il codice in modo da far andare a buon fine i test il più velocemente possibile

Un Bilancio del Test Driven Development

- Complessivamente questa tecnica di sviluppo del codice mira a mantenere la “località” degli errori costantemente sotto controllo
- Vantaggi:
  - Ogni pezzo di codice viene testato
  - Gli errori introdotti dai nuovi cambiamenti vengono scoperti subito
  - i test sono un’efficace documentazione del codice (in particolare della semantica) che si mantiene necessariamente coerente
  - in 10 minuti non si possono fare “troppi” errori
  - dual feedback tra test e codice testato
  - si accorcia il ciclo di feedback sulle scelte di progetto
- Svantaggi:
  - ha senso solo con programmatori esperti e capaci

- Test types
- Unit tests
- Component tests
- CUT/OUT
- Integration tests
- Interaction tests
- Static tests; Dynamic tests:
  - functional
  - acceptance (specificati dal customer)
  - regression
- Test Coverage
- XP ed il Testing
  - Incremental ad Iterative development
  - Continuous integration
  - “Test-first” approach
- Refactoring ed il Testing