

Introduzione al Testing

Unit Testing

Ingegneria del Software L-A

Ingegneria del Software LA

Z2.1

Glossario

- *Reliability* / Affidabilità: la misura del successo con cui il comportamento osservato di un sistema si correla con le specifiche del comportamento stesso
- *Failure* / Guasto: ogni deviazione del comportamento osservato dalle specifiche
- *Error* / Errore: il sistema si trova in uno stato in cui ogni ulteriore passo porta ad un *Failure*
- *Fault (Bug)* / Difetto: la causa "algoritmica" di un errore
- *Correction* / Correzione: un cambiamento ad un componente per riparare un *Fault*

Ingegneria del Software LA

Z2.2

Gestione degli errori

- *Prevention* / Prevenzione (prima che il sistema sia rilasciato):
 - Utilizzare una buona metodologia di programmazione per ridurre la complessità (anche seguire direttive di stile...)
 - Utilizzare un sistema di *versioning* dei sorgenti per prevenire inconsistenze
- *Detection* / Individuazione (mentre il sistema sta girando):
 - *Testing*: creare *failure* mentre il sistema sta girando
 - *Debugging*: cercare la causa di una *failure* "non pianificata"
 - *Monitoring*: fornire informazioni di stato
- *Recovery* / Recupero (recupero del sistema da un *failure*):
 - Blocchi *try...catch* opportuni
 - Se in ambiente dbms (transazioni atomiche)

Ingegneria del Software LA

Z2.3

Testing₁

- Testing = Analisi dinamica dell'implementazione del sistema
- E' praticamente impossibile testare in modo completo un sistema
 - Proibitivo in termini di tempi e costi
- Testing rileva la presenza di errori ma non la loro assenza

Ingegneria del Software LA

Z2.4

Testing₂

- Si fa funzionare il “sistema” con dati di input per cui si conoscano i risultati – *test case* (caso di test)
- Si verifica che i risultati siano quelli attesi

Ingegneria del Software LA

Z2.5

Stadi di Test

- *Unit testing*
 - Testing dei componenti individuali
- *Module testing*
 - Testing di componenti dipendenti
- *Sub-system testing*
 - Testing di insiemi di moduli collegati in sottosistemi. Verifica delle interfacce dei sottosistemi.
- *System testing*
 - Testing del sistema completo. Test dei requisiti funzionali e di quelli non funzionali: correttezza, performance, robustezza, interoperabilità, ecc.
- *Acceptance testing*
 - Testing da parte del committente con dati reali – *alpha testing*
 - Testing da parte di un limitato gruppo di potenziali acquirenti su un prodotto di mercato – *beta testing*

Ingegneria del Software LA

Z2.6

Component / Integration Testing

- *Component Testing*
 - Effettuato dallo sviluppatore
 - Testing di componenti singole
 - Effettuato in base “all’esperienza”
- *Integration Testing*
 - Effettuato da un team di testing
 - Testing di componenti integrate in un sottosistema
 - Effettuato in base alle specifiche del sistema

Ingegneria del Software LA

Z2.7

Debugging

- Se un test ha rilevato un difetto (*bug*) occorre che tale difetto sia rimosso
- Tramite i dati in ingresso / uscita ed eventualmente un *debugger* lo sviluppatore corregge il problema
- *Regression Testing*: si ripete la suite di test (l’insieme dei casi di test) di tutto il sistema per verificare che non siano stati introdotti nuovi *bug*

Ingegneria del Software LA

Z2.8

Testing statistico

- I test sono concepiti in modo da riflettere le caratteristiche statistiche degli *user input* per i diversi profili di utente (*viewpoints*).
- Non si preoccupa di scoprire e correggere *software faults*, ma di ottenere stime di affidabilità (*reliability*): probabilità di comportamento *error-free*, cioè senza *software failures*, rispetto a
 - un dato intervallo temporale,
 - uno specifico obiettivo (servizio), in un particolare contesto d'uso (*viewpoint*).

Ingegneria del Software LA

Z2.9

Software fault / failure

- Non tutti i *software faults* (*bugs*) producono sempre *software failures*.
- Se il *fault* si trova in una funzionalità non essenziale, può causare *failure* talmente di rado da non avere effetto sull'affidabilità globale del sistema

Ingegneria del Software LA

Z2.10

Black-box/Functional Testing

- Il programma è considerato una scatola nera
- Si osserva la relazione "*input – output*"
- I dati di test sono basati sulle specifiche
- La pianificazione di questi test può iniziare non appena le specifiche sono pronte (prima dell'inizio dello sviluppo)

Ingegneria del Software LA

Z2.11

White-box/Structural testing

- Si derivano i dati di test dalla struttura del programma
- Tipicamente applicato a piccole unità (metodi, oggetti)
- L'obiettivo è di testare tutti gli *statements* (senza badare ad archi, percorsi o grafi)

Ingegneria del Software LA

Z2.12

Unit Testing – Cos'è?

- Circa equivalente al test di chip hardware
- Il testing è effettuato, per ogni modulo, in modo isolato per verificarne il comportamento
- Tipicamente un test di unità costruisce una sorta di ambiente artificiale ed invoca le routine da testare sul componente

Ingegneria del Software LA

Z2.13

Unit Testing – Perché usarlo?

- Rende la vita più facile:
 - Il codice che viene integrato nel sistema è testato
 - Il *debugging* non viene effettuato contestualmente all'integrazione
- Le domande a cui si dovrebbe rispondere:
 - Il componente rispetta le specifiche?
 - Il componente rispetta le specifiche in tutti i casi?
 - Il componente è affidabile?
- Effetti collaterali positivi:
 - Il codice risulta "più" documentato: i test sono esempi d'uso!

Ingegneria del Software LA

Z2.14

Aree da testare

- Verificare i risultati (ovvio...)
- Verificare il comportamento al contorno → verificare il comportamento con dati strani "che non ci si aspetta" (es. età di una persona pari a 10000 anni)
- Verificare un'eventuale relazione inversa → verificare che i risultati di un'operazione possano essere invertiti (non *Undo / Redo*)
- Ottenere i risultati tramite altri algoritmi (ad es. meno efficienti) e verificare che siano compatibili
- Forzare le condizioni d'errore
- Verificare che le *performance* siano entro i limiti stabiliti

Ingegneria del Software LA

Z2.15

Scuse per non fare Unit Testing

- "Ci vuole troppo tempo per scrivere i test"
 - Quanto tempo ci vuole per "debuggare" senza test?
 - ...scrivere i test non appena possibile, anche prima del componente da testare (*Test Driven Development*) → aiuta a definire le specifiche
- "Ci vuole troppo tempo per far girare i test"
 - Non dovrebbe... basta far girare i test che impiegano tempo (tipicamente i test i carico) meno frequentemente
- "Non è mio compito testare il mio codice"
 - Il compito di uno sviluppatore è produrre codice che funziona e non codice pieno di bug...
- "Ma compila!"
 - Il fatto che il software venga compilato non significa che funzioni. Il compilatore effettua solamente una verifica sintattica.
- "Sono pagato per scrivere codice, non test!"
 - "Sei pagato per scrivere codice, non per usare il debugger!"

Ingegneria del Software LA

Z2.16

J/ NUnit

- Un piccolo *framework* di test scritto in *Java/C#* composto da una serie di classi che compiono il lavoro più ripetitivo (es: contare e riportare gli errori e i test falliti, far girare i test in *batch*, ecc.)
- *NUnit* non è solo un *porting* di *JUnit* per *.Net* ma aggiunge funzionalità sfruttando le caratteristiche avanzate del *framework*

Ingegneria del Software LA

Z2.17

NUnit

- Meccanismo basato sugli attributi per identificare le classi (*Test Fixtures* – “equipaggiamento / roba di test”) ed i metodi di test. Una *Fixture* è tutto ciò che serve per far girare casi di test correlati
- In *JUnit 3.x* viene utilizzata l’ereditarietà come meccanismo per identificare quali sono le classi che rappresentano *Test Fixtures* e una convenzione sui nomi per identificare i metodi di test
- Il sistema dei *Custom Attributes* disponibile in *.Net* fornisce un metodo più semplice e meno ambiguo per l’identificazione di classi e metodi
- *JUnit 4.x* si allinea all’utilizzo dell’equivalente Java dei *Custom Attributes*, le *Annotations*.

Ingegneria del Software LA

Z2.18

NUnit Basics₁

- Identificare una *Test Fixture*

```
[TestFixture]
public class SuccessTest
{
    // ...
}
```

- Identificare un metodo di test in una *Test Fixture*

```
[Test]
public void Success()
{
}
```

Ingegneria del Software LA

Z2.19

NUnit Basics₂

- *SetUp / TearDown* di un test – prepara e distrugge l’ambiente di test per metodo: il metodo marcato con *SetUp* viene invocato sempre prima di ogni metodo *Test*, il metodo marcato *TearDown* viene invocato sempre dopo ogni metodo *Test*

```
[SetUp]
public void Init()
{
    //inizializzazione per test
}

[Test]
public void Destroy()
{
    //liberazione risorse per test
}
```

Ingegneria del Software LA

Z2.20

NUnit Basics₃

Assert.AreEqual(Object expected, Object actual, String message)

Verifica che due oggetti o valori primitivi abbiano lo stesso valore

Assert.AreSame(Object expected, Object actual, String message)

Verifica che due riferimenti puntino alla stessa istanza

Assert.IsTrue(boolean condition, String message)

Verifica che la condizione sia vera

Assert.IsFalse(boolean condition, String message)

Verifica che la condizione sia falsa

Assert.IsNull(Object o, String message)

Verifica che il riferimento sia *null*

Assert.IsNotNull(Object o, String message)

Verifica che il riferimento non sia *null*

Assert.AreEqual(float expected, float actual, float tolerance, String message)

Verifica che due *float* abbiano lo stesso valore ma con una certa tolleranza

Ingegneria del Software LA

Z2.21

NUnit Basics₄

- Per verificare che un'operazione, sotto certe condizioni, sollevi un'eccezione:

```
[Test, ExpectedException(typeof(NullReferenceException))]
public void TestForException()
{
    object o = null; // = new object();
    o.ToString();
}
```

Ingegneria del Software LA

Z2.22

NUnit Basics₅

- Un test fallisce quando
 - Un'asserzione fallisce
 - Viene sollevata un'eccezione inattesa
- Un test ha successo quando
 - Tutte le asserzioni hanno successo
 - Viene sollevata un'eccezione attesa

Ingegneria del Software LA

Z2.23

NUnit – il test “minimo”

```
namespace Tests
{
    using System;
    using NUnit.Framework;

    [TestFixture]
    public class SuccessTest
    {
        [Test]
        public void Success() {}
    }
}
```

Ingegneria del Software LA

Z2.24

Cos'altro c'è?

- NUnit
 - *SetUp* e *TearDown* per *Test Fixture*
 - Raggruppare i test in categorie
 - Ignorare alcuni test
 - ...
- *Mock Objects*
 - Oggetti *dummy* che consentono di verificare il comportamento di un componente con alcuni componenti al contorno opportunamente istruiti
 - NMock, DotNetMock...

Ingegneria del Software LA

Z2.25

Poi venne VS 2008 Pro...

- ...con tool di test integrati e modalità di test **estremamente simili** a quelle di NUnit:
- Anche in VS, i test e le loro parti sono identificati tramite opportuni attributi

NUnit	Visual Studio 2008
Attributi	
TestFixture	TestClass
Test	TestMethod
SetUp / TearDown	TestInitialize / TestCleanup
TestFixtureSetup / TestFixtureTearDown	ClassInitialize / ClassCleanup (solo metodi statici)
ExpectedException	ExpectedException
...	...
Altre classi helper	
Assert	Assert
CollectionAssert	CollectionAssert
...	...

Ingegneria del Software LA

Z2.26

Code Coverage Analysis₁

Consente di:

- Trovare aree di un programma che non sono verificate da un insieme di casi di test
- Creare casi di test aggiuntivi per incrementare la copertura
- Determinare una misurazione quantitativa della copertura → misura indiretta della qualità

Altri aspetti:

- Identificare casi di test ridondanti che non incrementano la copertura

Ingegneria del Software LA

Z2.27

Code Coverage Analysis₂

- E' un tipo di analisi strutturale (white box)
- Si usa per verificare la qualità dei casi di test e non la qualità del prodotto finale
- L'analisi di copertura è spesso chiamata *test coverage analysis* o più brevemente *test coverage* proprio per indicare il fatto che si sta cercando di verificare la bontà dei *test cases*
- Di base i *faults (bug)* sono correlati al controllo del flusso (di esecuzione) ed è possibile esporre tali *faults* variando le condizioni contenute nelle strutture di controllo

Ingegneria del Software LA

Z2.28

Statement Coverage₁

- Verifica che tutti gli statement siano stati eseguiti almeno una volta
- Vantaggi:
 - Può essere applicato direttamente senza "strumentare" il codice sorgente

- Svantaggi:

- Insensibile rispetto ad alcune strutturazioni del controllo del flusso:

```
object o = oneObject;
if (condition)
    o.oneMethod();
o = anotherObject;
```

Senza un caso di test che faccia in modo che la condizione sia NON verificata, lo *statement coverage* indica che il codice è comunque completamente coperto → non è in grado di individuare la necessità di un test che può essere importante!

Ingegneria del Software LA

Z2.29

Statement Coverage₂

- Non individua se i loop raggiungono la condizione di terminazione ma solo se il corpo del loop è stato eseguito (presenza di un **break**...)
- Loop do-while viene comunque eseguito una volta → viene considerato allo stesso modo dei *non-branching statements*
- E' completamente insensibile alle condizioni composte

```
if (ca || cb)...
```

→ Si entra nel corpo dell'if sia se **ca** è vera, sia se **cb** è vera

→ Cosa succede se **ca** è vera e **cb** no? E se **cb** è vera e **ca** no?

Ingegneria del Software LA

Z2.30

Altri tipi di analisi₁

- Decision Coverage:

- Indica se le espressioni booleane utilizzate nelle strutture di controllo sono valutate sia true, sia false – vengono considerate le espressioni nella loro interezza

- Problema:

```
if (condition1 && (condition2 || function1())) statement1;
else statement2;
```

- Condition Coverage:

- Indica se tutte le sottoespressioni condizionali (separate da operatori logici) sono state valutate sia true, sia false. Tale tipo di analisi, "misura" le sottoespressioni in modo indipendente le une dalle altre
 - Si può dimostrare che Full Condition Coverage !=> Full Decision Coverage...

Ingegneria del Software LA

Z2.31

Altri tipi di analisi₂

- Molti altri tipi di analisi disponibili sempre più complessi
- Più l'analisi è complessa, più gli strumenti per l'analisi si fanno...
 - Complessi ed invasivi → il codice sorgente va opportunamente strumentato per supportare l'analisi
 - Costosi → per la complessità intrinseca delle operazioni

Ingegneria del Software LA

Z2.32

Code Coverage - Software

- Visual Studio Team Suite – commerciale
- NCover – commerciale (era open source...)
<http://www.ncover.com/>
- PartCover – open source, in rapida evoluzione
<http://sourceforge.net/projects/partcover/>

Ingegneria del Software LA

Z2.33

Materiale

- Pragmatic Unit Testing – Hunt, Thomas – The Pragmatic Bookshelf
- www.nunit.org
- www.testdriven.net (addin per VS.NET 2003/2005/2008) per pilotare tool di terze parti
- MS Visual Studio 2008 (da MSDN-AA)
 - Possibilità di integrare NUnit...

Ingegneria del Software LA

Z2.34