

# Test Driven Development (TDD)

# Riferimenti

- S. Amber. Introduction to Test Driven Development (TDD). [www.agiledata.org](http://www.agiledata.org)
- <http://www.testdriven.com> (online community/forum for TDD)
- D. Janzen, Test-Driven Development: Concepts, Taxonomy, and Future Direction, IEEE Computer-2005

# Motivazioni per il TDD

- Spesso il software prodotto con approcci tradizionali non risponde alle reali esigenze del cliente (scarsa qualità esterna)
  - Anche se ci si sofferma molto sulle attività di specifica dei requisiti, si corre il rischio di concedere più tempo ai cambiamenti delle esigenze reali!
- Spesso il software è di qualità interna scadente (per i tempi rapidi di consegna, le tecnologie mutevoli, la quantità di software richiesta)
  - Ha una elevata presenza di difetti
  - È difficile da mantenere e da comprendere, e dunque rallenta anche i nuovi sviluppi

# Cosa significa TDD?

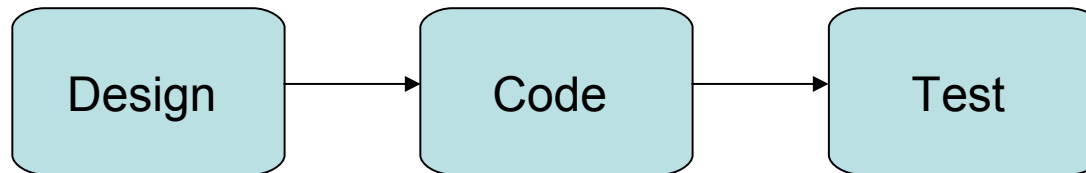
- È' un approccio evolutivo allo sviluppo software in cui si scrive prima un test che fallisce, e poi si scrive il relativo codice.
- Differisce dagli approcci tradizionali, in cui prima si scrive il codice e poi (forse) lo si testa
- TDD è uno stile di sviluppo/ progettazione, non è una tecnica di testing
- In TDD la programmazione e lo unit test non sono attività separate
- In TDD, si può scrivere nuovo codice solo se un test automatico è fallito
- TDD Mantra: **“Only ever write code to fix a failing test”**

# Vantaggi del TDD

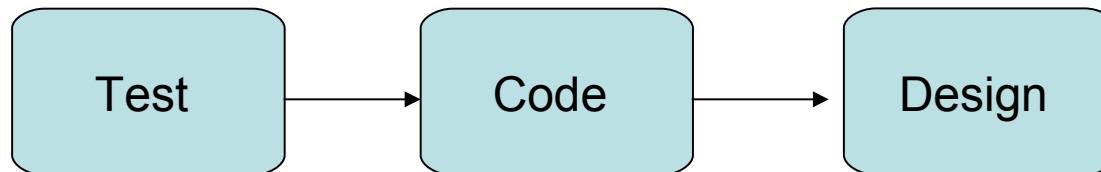
- Permette di scrivere software migliore e più rapidamente (costruendo software in piccoli incrementi- 1 test alla volta)
- Evita la paura nello sviluppatore che :
  - Lo induce a procedere per tentativi
  - Lo fa comunicare meno
  - Gli fa temere il feedback
  - Lo rende acido!
- Produce codice pulito e che funziona (in modo opposto allo sviluppo architecture driven, in cui si fanno prima tutte le decisioni)
- Consente agli sviluppatori di produrre un insieme di test di regressione automatizzabili, man mano che sviluppano

# La soluzione offerta dal TDD

- ***Ciclo di sviluppo Tradizionale***



- ***Sviluppo con TDD***

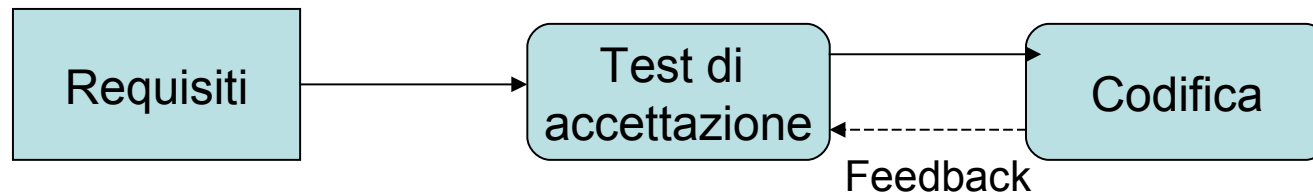


# Qualità interna col TDD

- La presenza di difetti nel codice può dipendere dalla mancanza di test, o dalla mancata esecuzione di test che abbiano evidenziato tali difetti.
- Invece:
  - Col TDD non ci sarà mai, praticamente, una parte di codice non testata
  - Scrivendo i test prima del codice, si è portati a cercare test che esercitano sia i casi normali, che i casi limite e di errore
  - Si riduce il tempo per la correzione dei difetti (si sa esattamente quali nuove linee possano aver fatto fallire il test: non serve neanche il debugger!)

# Qualità esterna con l'Acceptance TDD

- Si trasformano i requisiti in una serie di test eseguibili, e poi si prosegue implementando codice che rispetta tali test, piuttosto che requisiti verbali mal compresi.



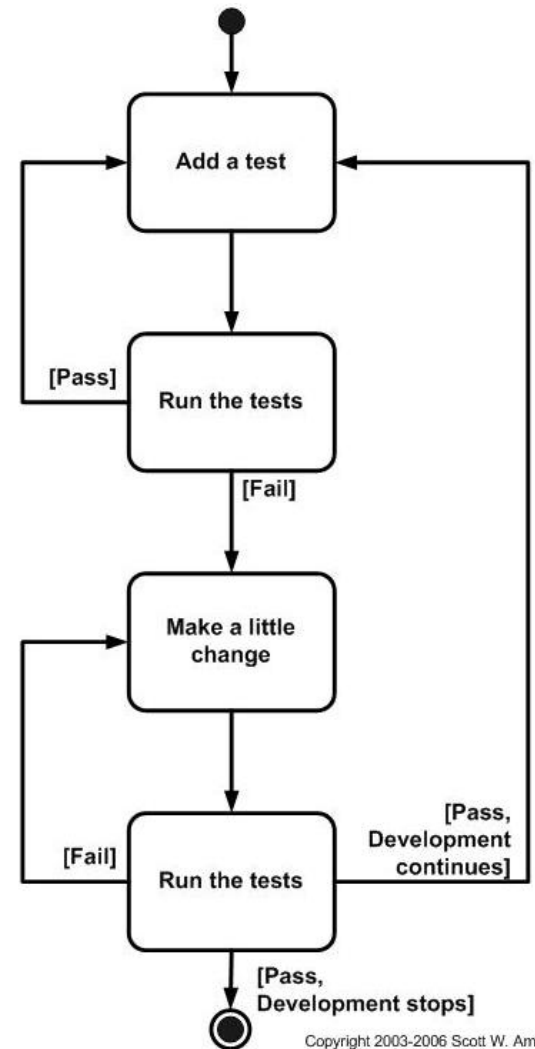
# TDD Mantra



- Red/Green/Refactor è il TDD mantra, e descrive i tre stati che si attraversano nello scrivere codice col TDD.
- **Red** Scrivi un Test che fallisce.
- **Green** Scrivi il codice per soddisfare il test.
- **Refactor** Migliora il codice senza cambiarne la funzionalità.
- **Ripeti.**

# Il Ciclo di Sviluppo TDD

- Scrivi un test che non funzionerà (red)
- Fai in modo che il test funzioni, scrivendo solo il codice necessario a superarlo (green)
- Migliora (ristruttura – il codice ed i test)
- Esegui tutti i test per assicurarsi che non ci siano problemi
- Ripeti finchè non trovi altri test



Copyright 2003-2006 Scott W. Ambler

# Il Ciclo di Sviluppo TDD- Maggiori dettagli

- Fase Add a Test
  - Si scrive un test per l'unità da testare (di conseguenza si progetta l'API dell'unità!)
  - l'API deve essere semplice ed il test dovrebbe essere prodotto in tempi brevi (al più pochi minuti)
  - Il codice che poi scriveremo sarà, necessariamente, testabile
- Fase Run a Test
  - In questa fase si scrive il codice necessario a far passare il test
  - Quando il test passa, si passa a modificare il test
  - Quando il test non passa, si fanno piccole modifiche al codice

# Refactoring

- Quando tutti i test saranno passati, si può pensare a migliorare il codice prodotto
- Si può pensare ad eliminare codice duplicato, a migliorarne l'espressività, riducendo l'accoppiamento e migliorandone la coesione.
  - Dopo il Refactoring, i test vanno rieseguiti
  - Se i test non passano (Red) è stato fatto qualche errore col refactoring che dovrà essere corretto, per far di nuovo passare tutti i test (Green)
  - Questi passi vanno ripetuti fino a che non si riusciranno a trovare altri test per scrivere nuovo codice.

# Refactoring

- La definizione di Martin Fowler [1]:
- Restructuring of software by applying a series of internal changes that do not affect its observable behavior
- Creare una classe
- Creare un metodo
- Creare una Interfaccia
- Creare l'implementazione di una Interfaccia
- Estrarre un metodo
- Estrarre una variabile locale
- Estrarre una variabile di un campo/oggetto
- Rinominare ...

# Relazione fra XP e TDD

- L'XP è un *metodo* agile.
- Il TDD è una *pratica* agile che nasce dalla combinazione di due concetti fondamentali dell'XP:
  - Test First Programming
  - Refactoring
- Il TDD è una pratica che può essere applicata anche senza le altre pratiche dell'XP.
- Lo scopo del TDD è di ridurre la distanza fra decisioni e feedback durante lo sviluppo.

# Unit Test e TDD

- Un test non è di unità se:
  - comunica con il database
  - comunica in rete
  - modifica il database
  - non può essere lanciato in parallelo ad altri test
  - bisogna effettuare diverse operazioni per lanciare il test
- Per eseguire lo Unit Test possono essere necessarie le Fixture ed i Test Doubles (Doppioni).

# Un esempio di ciclo TDD

- Si vuole sviluppare un metodo che, dati due Interi, restituisce l'intero che è la somma dei due parametri.

# Esempio

- Test

```
Integer i = new Integer(5);  
Integer j = new Integer(2);  
Object o = sum(i,j);
```

- Metodo

# Esempio

- Test

```
Integer i = new Integer(5);  
Integer j = new Integer(2);  
Object o = sum(i,j);
```

- Metodo

```
public static Object sum  
(Integer i,  
Integer j) {  
    return new Object();  
}
```

# Esempio

- Test

```
Integer i = new Integer(5);
Integer j = new Integer(2);
Object o = sum(i,j);
if (o instanceof
Integer)
return true;
else
return false;
```

- Metodo

```
public static Object sum
(Integer i,
Integer j) {
return new Object();
}
```

# Esempio

- Test

```
Integer i = new Integer(5);
Integer j = new Integer(2);
Object o = sum(i,j);
if (o instanceof
Integer)
return true;
else
return false;
```

- Metodo

```
public static Integer sum
(Integer i,
Integer j) {
return new Integer();
}
```

# Esempio

- Test

```
Integer i =new Integer(5);  
Integer j =new Interger(2);  
Object o = sum(i,j);  
if ((o instanceof  
Integer) &&  
(new Integer(7))  
.equals(o))  
return true;  
else  
return false;
```

- Metodo

```
public static Integer sum  
(Integer i,  
Integer j) {  
return new Integer();  
}
```

# Esempio

- Test

```
Integer i = new Integer(5);  
Integer j = new Integer(2);  
Object o = sum(i,j);  
if ((o instanceof  
Integer) &&  
((new Integer(7))  
.equals(o))  
return true;  
else  
return false;
```

- Metodo

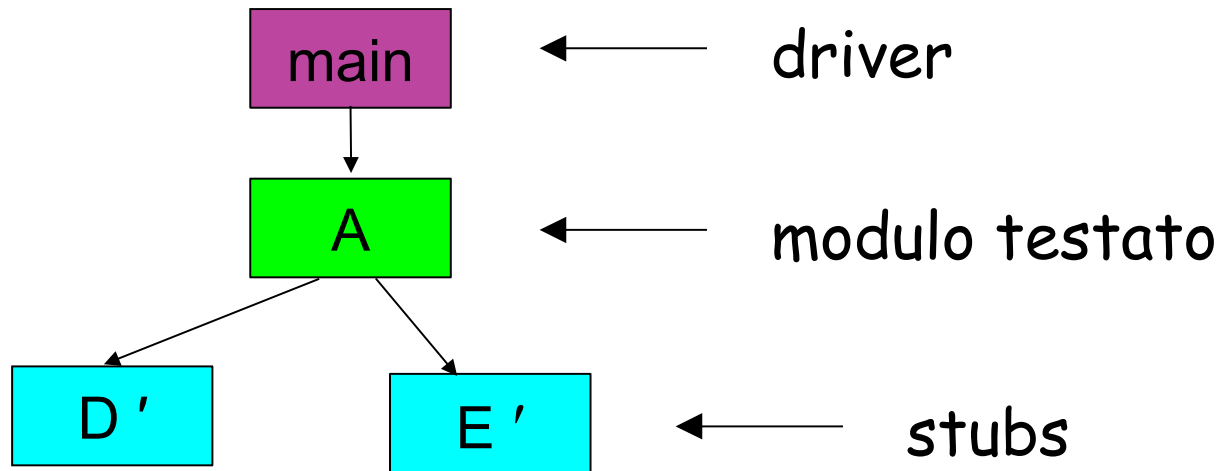
```
public static Integer sum  
(Integer i,  
Integer j) {  
return new Integer(  
i.intValue() +  
j.intValue());  
}
```

# Fixture e Test Doubles

- **FIXTURE**
  - Componenti usati per settare il contesto (lo stato) in cui un test deve essere eseguito)
- **TEST DOUBLES**
  - Elementi necessari per testare una unità che ha dipendenza da altri elementi
  - Stanno al posto di altri oggetti e fanno le stesse cose richieste agli oggetti (ma in un modo più veloce/ semplice)
    - Esistono diversi tipi di Test Doubles (voi conoscete già gli STUB):
    - STUB, MOCK OBJECT, FAKE

# Stub e Driver

- I moduli testati hanno bisogno di essere chiamati (dai Driver)
- I moduli chiamati devono essere sostituiti da altri (Stub)



# Differenze fra Stub e Mock

- I Mock-Objects non sono Stub! [Fowler]
  - <http://martinfowler.com/articles/mocksArentStubs.html>
- In genere lo stub è molto più semplice di un Mock-Object
  - Gli stub forniscono risposte preconfezionate (con valori prefissati) a chiamate fatte durante il test, senza rispondere di solito a nulla che sia al di fuori di ciò che è previsto per il test.
- I mock hanno implementazioni più sofisticate che consentono di verificare il comportamento dell'unità testata (e non solo lo stato)
  - verificando ad esempio le collaborazioni avute con altri oggetti ed il relativo ordine di esecuzione

# Mock Object, Fake, e Stub

Type of mock	Description
Stubs	Stubs are essentially the simplest possible implementation of a given interface you can think of. For example, stubs' methods typically return hardcoded, meaningless values.
Fakes	Fakes are a degree more sophisticated than stubs in that they can be considered an alternative implementation of the interface. In other words, a fake looks like a duck and walks like a duck even though it isn't a real duck. In contrast, a stub only looks like a duck.
Mocks	Mocks can be considered even more sophisticated in terms of their implementation, because they incorporate assertions for verifying expected collaboration with other objects during a test. Depending on the implementation of a mock, it can be set up either to return hardcoded values or to provide a fake implementation of the logic. Mocks are typically generated dynamically with frameworks and libraries, such as EasyMock, but they can also be implemented by hand.

Tratto da: Test Driven - Practical TDD and Acceptance TDD  
for Java Developers (Lasse Koskela)- Manning Ed. 2008

# Mock-Objects

- Un mock object è una simulazione di un oggetto reale.
- Implementa l'interfaccia dell'oggetto da simulare ed ha il suo stesso comportamento.
- Possono fornire una risposta pre-impostata.
- Possono verificare se l'oggetto che li usa lo fa correttamente.
- Utilissimi per testare unità senza legarsi ad oggetti esterni.
- Vediamo come attraverso un esempio...

# Testing di Classi con dipendenze

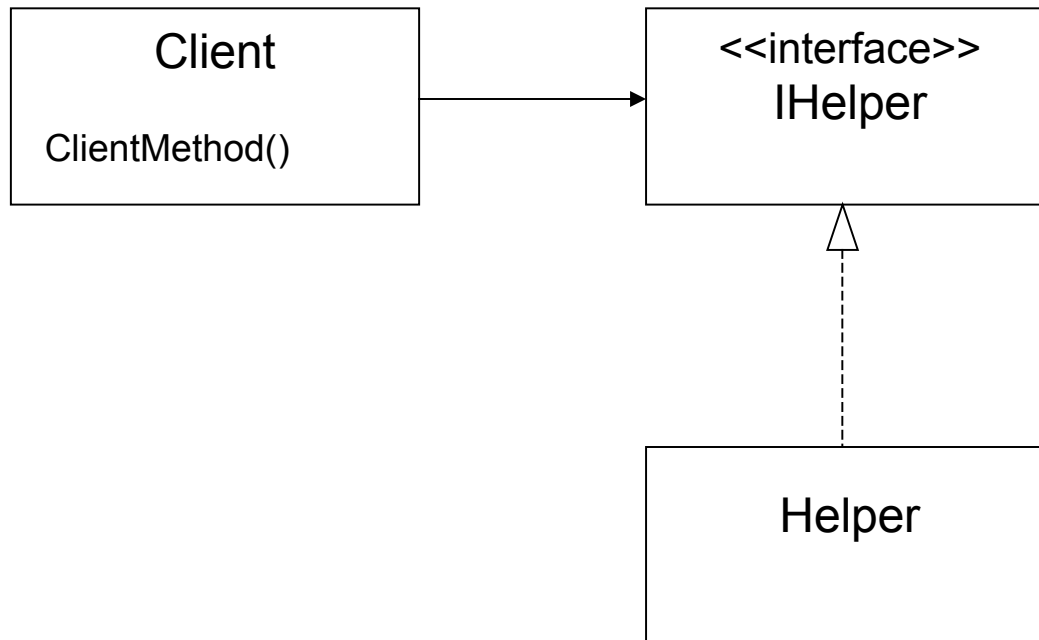
- Es.: La classe Client (da testare) usa i metodi di Helper



- Ma la classe Helper non può essere usata perché:
  - Non è ancora disponibile
  - Vogliamo controllare direttamente ciò che Helper restituisce a Client

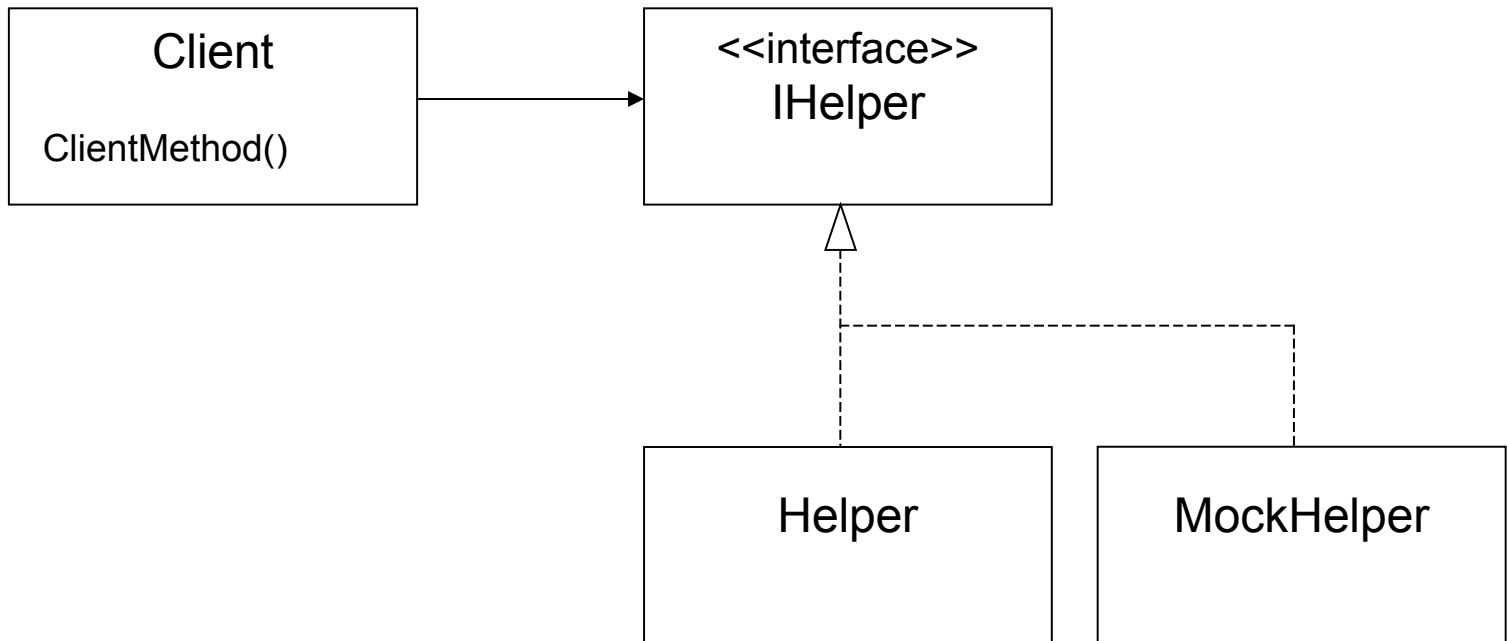
# La soluzione usando i Mock-Object

- Si estrae l'Interfaccia IHelper...



# La soluzione usando i Mock-Object

- Il MockObject implementa l'Interfaccia IHelper



# Un esempio di semplice Mock

```
public class MockHelper implements IHelper
{
    public MockHelper ( )
    {
    }

    public Object helperMethod( Object aParameter )
    {
        Object result = null;
        if ( ! aParameter.toString().equals("expected" )
        {
            throw new IllegalArgumentException(
                "Unexpected parameter: " + aParameter.toString() );
        }
        result = new String("reply");
        return result;
    }
}
```

# Sviluppo di Mock

- In genere i Mock sono in grado di fare controlli sul comportamento dell'oggetto testato e sulle sue interazioni con altri oggetti.
- Lo sviluppo di Mock Objects è supportata da diversi frameworks, o librerie (come JMock in Java).

# EasyMock

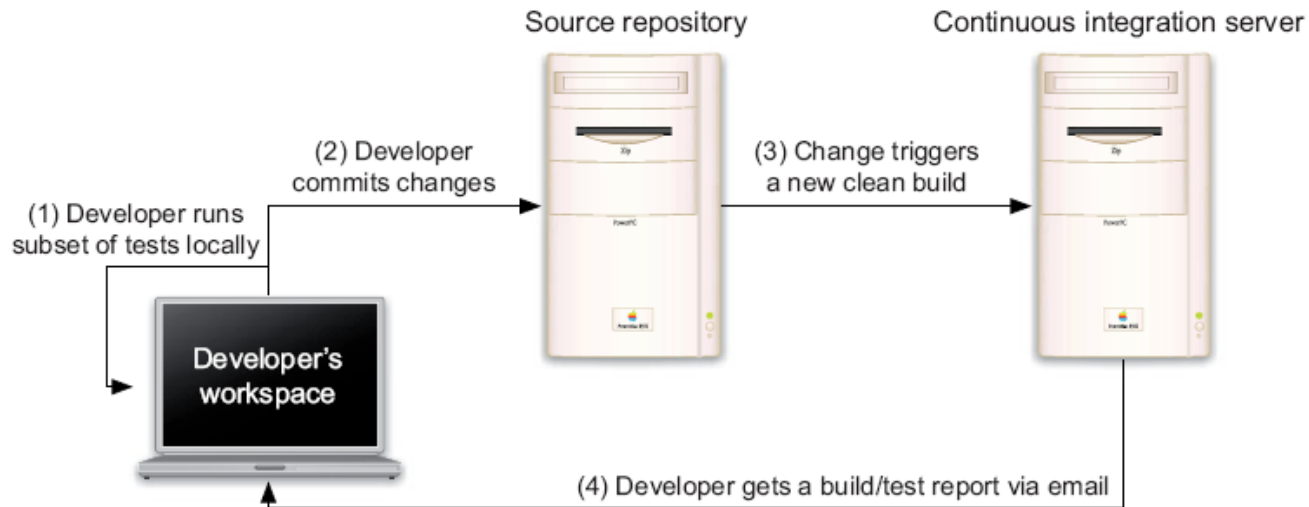
- É un Framework per creare mock objects a run time.
- Usa le Java reflection per creare una classe mock object che implementa una certa interfaccia.
- É un progetto open source project ospitato su SourceForge:
  - <http://www.easymock.org>

# TDD e tools per l'automazione del test

- Testing Frameworks
  - Mettono a disposizione librerie di classi per scrivere test, eseguirli, valutare ed esportare i risultati.
  - cppUnit / csUnit(.Net)/ CUnit / DUnit (Delphi),
  - JUnit/ NUnit /PHPUnit /PyUnit (Python),
  - Test::Unit (Ruby) / VUnit ....
- Test framework per il Test di Accettazione
  - Ci sono strumenti come Fit e Fitness, incentrati su tabelle per descrivere i test eseguibili e che impiegano forme visuali per facilitare la collaborazione tra developers, testers, e business analysts con nontechnical stakeholders.
- Strumenti per supportare Continuous Integration

# Continuous Integration in TDD

- La definizione di Martin Fowler:
  - *È una pratica di sviluppo software in cui i membri del team di sviluppo integrano il proprio lavoro frequentemente, almeno una volta al giorno. Ogni integrazione è verificata da strumenti automatici che eseguono il build e rieseguono tutti i test sulla nuova configurazione, per trovare errori di integrazione rapidamente.*



# Ulteriori lettura

- Extreme Programming Explained (Kent Beck) – 1st and 2nd Edition
- Test Driven Development (Kent Beck)
- Test Driven : Practical TDD and Acceptance TDD for Java Developers (Lasse Koskela)