Unit Testing con SQL Server

#### di [Alessandro Alpi](http://mvp.microsoft.com/it-it/mvp/Alessandro%20Alpi-4014222) – Microsoft MVP

blog italiano - <http://blogs.dotnethell.it/suxstellino>

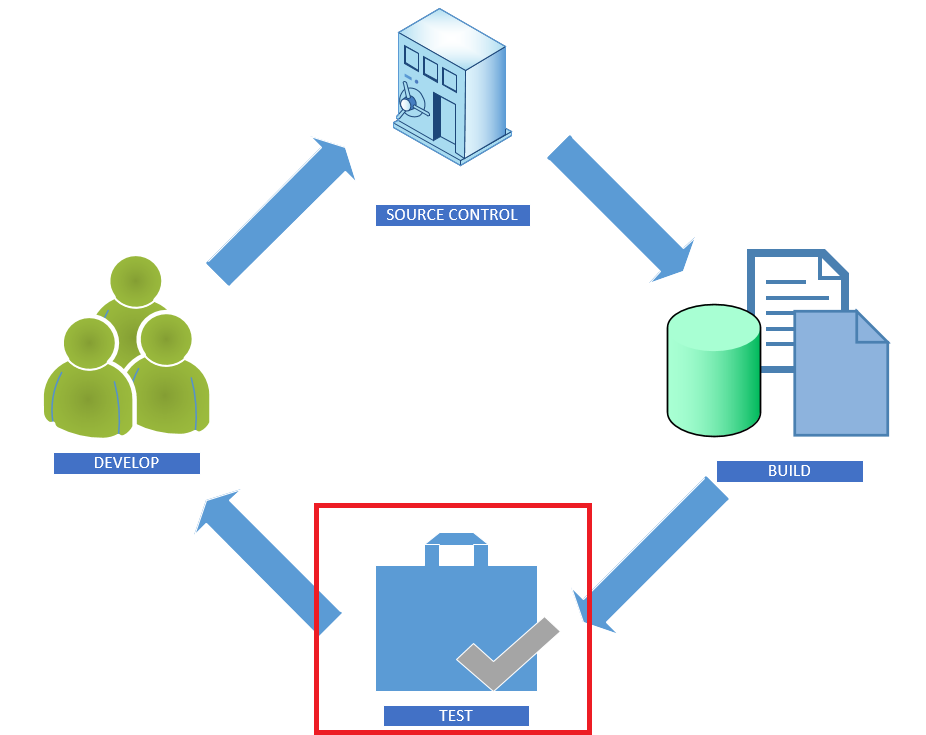
blog inglese - <http://suxstellino.wordpress.com>

sito web - <http://www.alessandroalpi.net>

* 1. Dn269828.7B654F178A3842F7F616A829DC6DF588(it-it,MSDN.10).png

## Introduzione

Abbiamo già trattato in [questo articolo](http://msdn.microsoft.com/it-it/library/dn383992.aspx) il concetto di *Continuous Integration* e come essa rientra nel ciclo di vita del nostro database ([DLM](http://msdn.microsoft.com/en-us/library/jj907294.aspx)). Nell’articolo abbiamo parlato di quanto mettere il database sotto source control sia importante e di come testare le nostre unità di lavoro. Tuttavia non abbiamo approfondito ogni passo. Per questo motivo ora entreremo nei dettagli, descrivendo un altro punto, ovvero il testing:

* 1. 

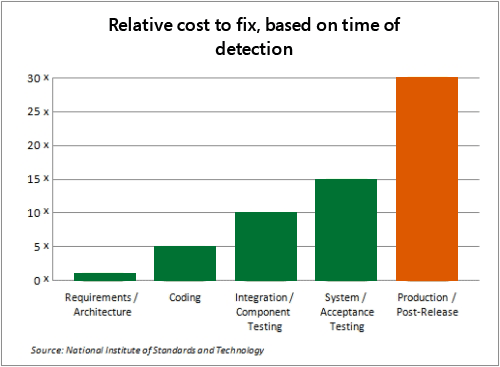
## Definizione di unit testing

In ingegneria del software, per unit testing (testing unitario) si intende l'attività di prova e collaudo di singole unità software. A seconda del paradigma o del linguaggio di programmazione, l’unità può essere una singola funzione, una singola classe o un singolo metodo. Gli scopi fondamentali sono l’individuazione precoce dei bug e la prevenzione delle regressioni.

(Fonte wikipedia)

## Motivi per cui è importante fare unit testing

Il grafico seguente mostra in maniera molto chiara il costo per la risoluzione di bug rispetto ad ogni fase del ciclo di vita dell’applicazione:

* 1. 

Man mano che ci si avvicina al rilascio in produzione il costo per le fix è sempre maggiore (con andamento praticamente esponenziale). Addirittura, in produzione, abbiamo un costo di 30 volte superiore rispetto alla prima fase. Questo porta inevitabilmente ad affermare che è necessario **“Risolvere un bug non appena è stato trovato”**.

Ovviamente, quando possibile. Sappiamo quanto sia complesso trovare tempo “subito” nelle situazioni reali di tutti i giorni. Possiamo però affermare che **più l’attesa aumenta, più il costo per la fix cresce**.

Non risolvere subito il problema porta le seguenti conseguenze:

I bug non fixati camuffano potenzialmente altri bug

Non applicando fix facciamo sembrare la qualità un’opzione, mentre dovrebbe essere un requisito fondamentale del nostro prodotto

Continuare a discutere di bug, perché non risolti, porta una perdita di tempo notevole ed una conseguente distrazione del team

I problemi non risolti portano anche metriche e valutazioni inaffidabili, nonché stime non accurate

I problemi non risolti portano un rallentamento della velocità di rilascio

Più il codice è familiare e semplice, più tutto il team evita di creare bug in futuro

Oltre al costo abbiamo una serie di considerazioni che portano sempre più verso l’approccio al test:

In ogni applicazione esistono funzionalità mission-critical, che devono funzionare

Testare porta a prevenire il bug in questo caso bloccante anche per gli utilizzatori

Ogni applicazione necessita di sviluppo evolutivo

Testare porta a rilasciare software di qualità

Risoluzione più semplice del bug trovato e prevenire regressioni

Testare il comportamento allerta immediatamente gli sviluppatori

## Come testiamo ad oggi su database

Quando si cerca di testare un database, oppure un comportamento di stored procedure o di funzione, o ancora, direttamente dei dati, molti scenari impongono di creare una copia dei dati su cui lavorare. Nella maggior parte dei casi il test è totalmente manuale ed è svolto tramite l’ausilio di T-SQL creato al volo. Quante volte la funzione *PRINT* aiuta a “controllare” il valore attuale di una variabile. E quante volte statement di *SELECT* mostrano lo stato dei record di una tabella temporanea. In alcune situazioni ci si può permettere anche un debug della programmabilità, ma la soluzione, seppure efficace, è esposta a numerosi errori umani.

Il dato che si preleva dalla produzione al fine di provare una o più chiamate rischia inoltre di essere obsoleto. Valido nel momento del test, ma obsoleto poiché già cambiato nell’ambiente di produzione. E ancora, le soluzioni a cui si è abituati, non risultano molto comode quando si ha a che fare con oggetti in cui sono definiti dei vincoli come *DEFAULT* o *FOREIGN KEY*. Per provare in maniera “pulita” e senza ostacoli si rischia di dover cambiare lo schema del database, andando, di fatto, ad alterare la situazione rispetto all’ambiente da cui abbiamo prelevato i dati. Questo rende il test totalmente soggettivo e, di conseguenza, molto spesso inaffidabile.

## Unit testing su database, cosa testare

Con i test unitari possiamo automatizzare il processo di collaudo dei comportamenti, ma non solo. Ecco alcune delle operazioni che si dovrebbero testare:

Calcoli in procedure e funzioni

Vincoli (e schema in generale)

Casi limite sui dati

Comportamenti attesi sui dati

Gestione degli errori

Sicurezza

Standard di scrittura

Quando si parla di *gestione degli errori* non è sempre comodo e consigliato applicare test. In questo particolare caso rischiamo di perdere più tempo a crearne rispetto al tempo necessario per stabilire uno standard interno di gestione errori.

Per quanto riguarda la *sicurezza*, potrebbe essere considerata in un processo separato dal test unitario, essendo fortemente personalizzata per ambiente. Questo particolare caso fa parte più verosimilmente di un insieme di controlli da effettuare prima del deploy dei database.

Gli *standard* sono invece testabili sia nella fase di sviluppo, sia direttamente sull’ambiente ospite del nostro deploy. Esistono framework che controllano la programmabilità e le sintassi, mentre altri eseguono query sul catalog di sistema per capire lo stato di salute dell’istanza (backup, integrità, controlli di amministrazione, ecc.).

Per tutti gli altri casi è possibile invece applicare con semplicità pattern di unit testing. Il primo punto è di certo quello che va per la maggiore poiché corrisponde al controllo dei comportamenti ed è quello che più si presta agli esempi che seguiranno.

### Strumenti per unit testing

Possiamo dividere gli strumenti disponibili in due famiglie: *Framework* e *Tools*.

#### Framework

Vi sono alcuni framework molto efficaci utilizzabili in maniera del tutto immediata:

[tSQLt](http://tsqlt.org/)

[tSQLUnit](http://sourceforge.net/projects/tsqlunit/)

[SQLCop](http://sqlcop.lessthandot.com/)

Oltre a questi tre vi è anche [SS-Unit](http://sqlcop.lessthandot.com/), più recente, e da valutare. In questo articolo non parleremo di quest’ultimo.

I primi due hanno una struttura comune, che può essere suddivisa in quattro parti: fase di **raccolta dati** **o di creazione dell’ambiente**, fase di **applicazione del test**, fase di **asserzione e verifica**, e fase di **ritorno allo stato precedente** al test. Ognuno dei due framework, però, possiede terminologie differenti. Andiamo ad analizzarli qui di seguito.

#### tSQLt

tSQLt installa una serie di stored procedure sotto lo schema *tSQLt* ed un’assembly chiamato *tSQLtCLR*. Infine necessita di una configurazione di database particolare (*SET TRUSTWORTHY ON*).

I termini che contraddistinguono un test fatto con tSQLt sono tre. **Assemble**, corrisponde alla fase di creazione e raccolta dei dati di ambiente. **Act**, corrisponde alla fase di applicazione del comportamento e **Assert**, corrisponde alla fase di verifica. La fine del test (sia in caso di fallimento che di successo) porta la situazione al momento precedente al test.

Nella fase di **assemble**, tSQLt consente la creazione di oggetti denominati *Fake* (fittizi). Un esempio è il seguente:

EXEC **tSQLt.FakeTable** @TableName= 'Sales.Orders';

Con questo comando viene creata una tabella fittizia, ma lo stesso può essere fatto per le fuzioni con la stored procedure *tSQLt.FakeFunction*. Mentre quest’ultima possiede solo due parametri (il nome della funzione e quello della funzione fittizia), la *FakeTable* consente di specificare le operazioni da applicare sulla tabella “copiata”. Ad esempio, è possibile specificare se utilizzare i vincoli di default, la proprietà identity e le colonne calcolate.

*tSQLt.FakeTable* si occupa di rinominare la tabella indicata con un suffisso random e di creare una nuova tabella col nome originario, in modo da **isolare** l’oggetto per i test.

Nella fase di **act**, tSQLt esegue la logica da testare, ovvero i test creati dall’utente. Ad esempio, una stored procedure che esegue calcoli sulla *Sales.Orders*, appena definita come *Fake*. La stored procedure eseguirà i calcoli su un oggetto totalmente isolato e non toccherà l’oggetto da cui il *Fake* ha avuto origine. Oltre ad essere estremamente comodo, si tratta anche di una caratteristica vantaggiosa che contraddistingue questo framework dagli altri.

Nella fase finale, ovvero quella di **assert**, tSQLt fornisce le asserzioni più comuni, come:

tSQLt.AssertEmptyTable

tSQLt.AssertEquals

tSQLt.AssertEqualsString

tSQLt.AssertEqualsTable

tSQLt.AssertLike

tSQLt.AssertNotEquals

tSQLt.AssertObjectDoesNotExist

tSQLt.AssertObjectExists

tSQLt.AssertResultSetsHaveSameMetaData

tSQLt.Fail

Ognuna di queste procedure, non solo si verifica l’esito del test, ma procede anche con la pulizia ed il ripristino degli oggetti creati. In caso di creazione di oggetti *Fake*, viene eseguito il procedimento contrario rispetto alla fase di assemble. Questo rende il test completamente isolato ed affidabile.

L’intero processo descritto viene costruito all’interno del test creato dall’utente, il quale dovrà essere creato alll’interno di una *TestClass*, che corrisponde ad uno schema di database. Il nome del test (della stored procedure che funge da test) dovrà avere un prefisso *test*. Per creare una classe è sufficiente eseguire quanto segue:

* 1. EXEC **tSQLt.NewTestClass** @ClassName = 'SampleTests';

Come detto, tramite il comando viene creato uno schema. In aggiunta però viene applicata ad esso una proprietà estesa che lo distingue dai normali schema utente. Utilizzare lo stesso nome di *TestClass* porta tSQLt ad eseguire un drop preventivo della classe. È anche possibile rimuovere o rinominare le classi con stored procedure dedicate (*tSQLt.RenameClass* e *tSQLt.DropClass*).

Per l’esecuzione dei test è possibile seguire le strade seguenti:

Esecuzione di un test (stored procedure *tSQLt.Run*)

Esecuzione di tutti i test (stored procedure *tSQLt.RunAll*)

Esecuzione di una TestClass (stored procedure *tSQLt.RunTestClass*)

Qui di seguito un test di esempio:

* 1. CREATE PROCEDURE SampleTests.[test something on SalesOrders table]
  2. AS
  3. BEGIN
  4. DECLARE @ActualResult decimal(18, 2);
  5. --Assemble
  6. EXEC tSQLt.FakeTable 'Sales.Orders';
  7. INSERT INTO Sales.Orders (id, ProductId, Qty, Amount, OrderDate)
  8. VALUES (1, 10, 3, 5.5, GETDATE());
  9. --Act
  10. SELECT @ActualResult = dbo.MyFunction(Qty, Amount)
  11. FROM Sales.Orders WHERE id = 1;
  12. --Assertion
  13. EXEC tSQLt.AssertEquals
  14. @expected = 16.5,
  15. @actual = @ActualResult,
  16. @message = 'dbo.Function should return the right result';
  17. END;

Nel caso in cui il test fallisca il messaggio indicato verrà mostrato con l’aggiunta della stringa che identifica il valore attuale e quello atteso. Se il test ha esito positivo, nessun messaggio verrà tornato all’utente.

Qui di seguito un esempio di chiamata per l’esecuzione dei test:

* 1. EXEC **tSQLt.Run** 'SampleTests.[test something on SalesOrders table]';

#### tSQLUnit

Questo framework è molto più semplicistico e non è elastico come il precedente. Si ispira a xUnit e quindi è molto vicino a chi già ha avuto a che fare con quanto scritto da [Kent Beck](http://en.wikipedia.org/wiki/Kent_Beck). Si tratta di un set di oggetti sql con prefisso *tsu*, installabili tramite un unico file sql.

I test utente sono raggruppati in *TestSuite*, da specificare nel nome del test stesso. Lo schema è *dbo*.

Vengono creati con una naming convention ben definita:

Nelle procedure di test utente è necessario il prefisso *ut\_*

*TestSuite* subito dopo *ut\_*

Descrizione del test dopo la *TestSuite*

Un esempio può essere:

* 1. dbo.**ut\_SampleTest\_**SomethingOnSalesOrdersTable

La fase di raccolta e creazione dei dati è definita **setup**. Per definire le operazioni di pre-esecuzione, la naming convention è la seguente:

* 1. dbo.ut\_SampleTest\_**setup**

Con questa procedura è possibile effettuare l’aggiornamento dei dati necessari, l’inserimento di nuovi record, la cancellazione per partire da una situazione generica, e così via.

La fase di esecuzione del test è una semplice esecuzione della nostra procedura o funzione, che, a sua volta, sfrutta quanto preparato nel setup.

La fase di verifica viene fatta su considerazioni utente, ad esempio un controllo sull’errore tornato dalle logiche delle stored procedure. Un esempio è:

* 1. --Assert
  2. IF @customUserError <> 1
  3. EXEC **dbo.tsu\_failure** 'Something gone wrong!!'

Infine, la fase di ritorno alla situazione precedente al test è definita **teardown**. Per definire le operazioni di post-esecuzione, la naming convention è la seguente:

* 1. dbo.ut\_SampleTest\_**teardown**

Setup e Teardown sono termini raggruppati nel concetto di **Fixture**. Per *Fixture* si intende il contesto di test, ovvero lo stato da cui il test parte e al quale torna dopo l’esecuzione. Ogni *Fixture* viene ripetutamente eseguito per ogni test della *TestSuite*. Ad esempio, se per la *SampleTest* avessimo tre differenti test, *setup* e *teardown* verrebbero eseguite per tre volte.

Possiamo esprimere la suddetta pipeline in questo modo:

loop su TestSuite

loop su Test

setup

esecuzione test

teardown

fine loop

fine loop

Su MSDN Code Samples ho [pubblicato un set molto semplice di esempi](https://code.msdn.microsoft.com/tSQLUnit-samples-Database-54b10e47) per facilitare la comprensione delle sintassi.

#### SQLCop

Questo framework costituisce un insieme di comandi sql atti a controllare la salute dell’istanza ed, in generale, i vari standard di sintassi e di operazioni amministrative. Ad esempio, alcuni controlli eseguiti sono:

Procedures without *SET NOCOUNT ON*

Column Name Problems

Table Prefix

Old Backups

Orphaned Users

Si tratta di controlli su codice, sullo schema, sulle tabelle e sulle viste, sulle sintassi, sulle colonne ma anche sulle configurazioni e sulla salute dell’istanza. Per natura, non può essere incluso in test di unità, ma risulta estremamente utile testare gli standard prima di un rilascio in produzione.

Per approfondire i tipi di test eseguiti, leggere [qui](http://sqlcop.lessthandot.com/detectedissues.php).

### Tools

Per tools intendiamo veri e propri strumenti dotati di interfaccia grafica ed interazione utente. I più popolari, ad oggi, sono due:

[Visual Studio Unit Test Project](http://msdn.microsoft.com/en-us/library/hh598957.aspx)

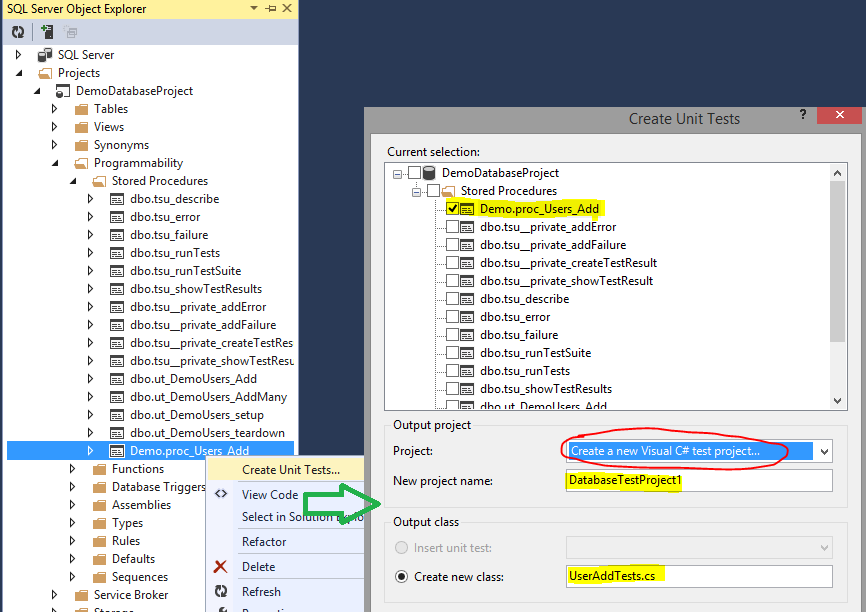
[Red-Gate SQLTest](http://www.red-gate.com/products/sql-development/sql-test/)

#### Visual studio

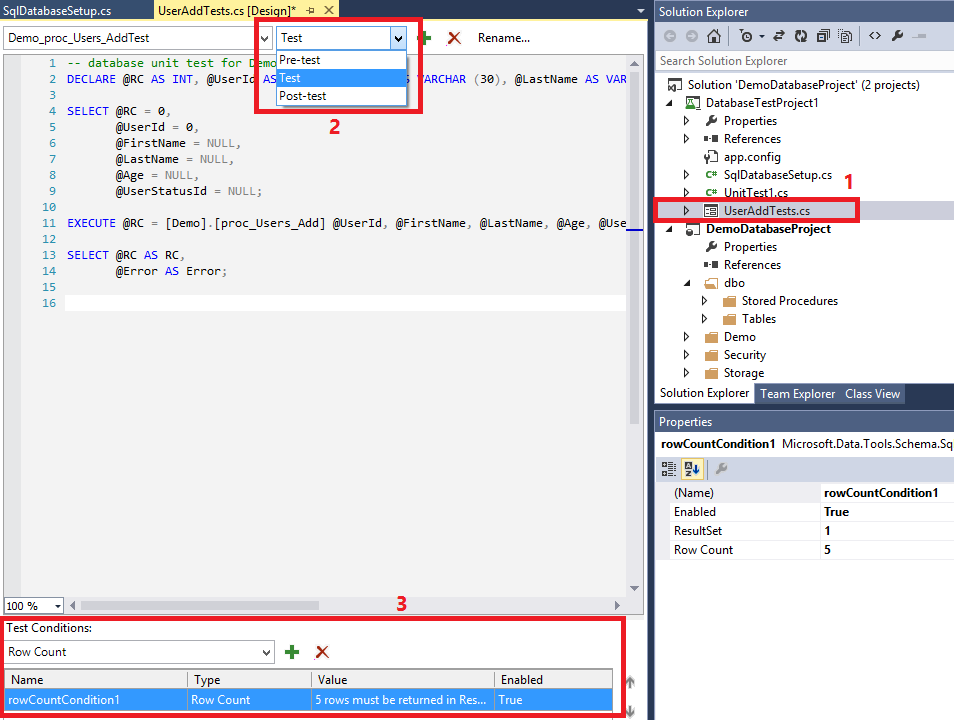
Visual Studio Data Tools (SSDT) consente di lavorare con i progetti di tipo SQL Server. Un template di questo tipo non solo consente la navigazione connessa dell’istanza, bensì fornisce gli strumenti per sviluppare su database importati e disconnessi e, successivamente, di sincronizzare i cambiamenti. Con il tipo di progetto database non è possibile fare unit testing direttamente. Serve un ulteriore template, già presente con l’installazione di Visual Studio, non SSDT, chiamato *Unit Test Project*. Per ogni linguaggio .net supportato c’è un *Unit Test Project* template. Nel nostro esempio utilizzeremo C#.

Come prima cosa è necessario avere un progetto di tipo SQLServer già creato. Per approfondire l’argomento, leggere [qui](http://msdn.microsoft.com/en-us/library/dd193245.aspx).

Avendo a disposizione un progetto ed il SQL Server Explorer, la creazione di uno *Unit Test Project* è molto semplice. È sufficiente premere il tasto destro sulla procedura su cui si vuole creare il test e selezionare “*Create Unit Test..*”, nonché seguire le indicazioni successive:

* 1. 

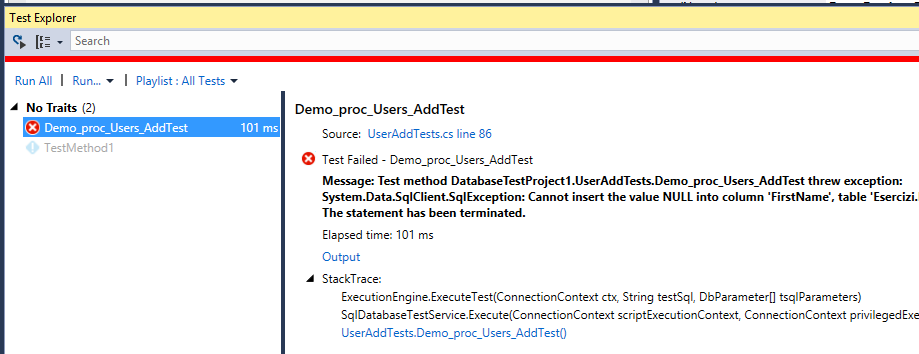
Questa operazione aggiunge un vero e proprio progetto di tipo Unit Test, che si utilizza regolarmente per progetti non database. Infatti il template è solo personalizzato per la procedura in oggetto, ma i file in esso contenuti sono pressochè gli stessi:

* 1. 

Il file aggiunto per il test su SQL Server (1, *UserAddTests.cs*) è dotato di un designer molto intuitivo, che consente di selezionare gli assert più comuni (3) e di configurarne le proprietà con la property grid a destra. Inoltre è possibile definire la fase di raccolta e creazione dei dati (2, *Pre-Test* designer) e la fase di pulizia e ripristino della situazione prima dell’esecuzione del test (2, *Post-Test* designer).

Il raggruppamento dei test non è scelto dall’utente come nei casi delle *TestClass* di *tSQLt* e delle *TestSuite* di *tSQLUnit*, ma lo si effettua a livello di file e cartelle. Un file può contenere solo un test, con un designer. È tuttavia possibile raggruppare i file all’interno di cartelle fisiche.

Per l’esecuzione dei test si utilizza il Test Explorer fornito con Visual Studio:

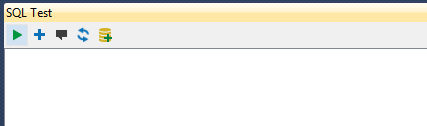
* 1. 

Con il Test Explorer è possibile controllare sia lo stato di esecuzione dei task (la linea rossa sopra a tutto sta ad indicare che almeno un test è fallito per la soluzione), sia quello di esecuzione del task o dei task eseguiti (il pallino accanto al test sulla sinistra), sia i messaggi di errore o di successo. Con il test explorer è possibile automatizzare anche l’esecuzione degli unit test subito dopo la compilazione. È sufficiente premere il pulsante in alto a sinistra nella view.

#### SQLTest

Un tool di terze parti per la gestione degli unit test è SQLTest di Red-Gate. È integrato in Sql Server Management Studio, il che lo rende molto appetibile per chi utilizza per la gran parte della giornata l’iIDE di gestione di SQL Server. Supporta due dei framework di cui abbiamo parlato nei paragrafi precedenti, ovvero tSQLt e SQLCop. Copre quindi test unitari e test di standard.

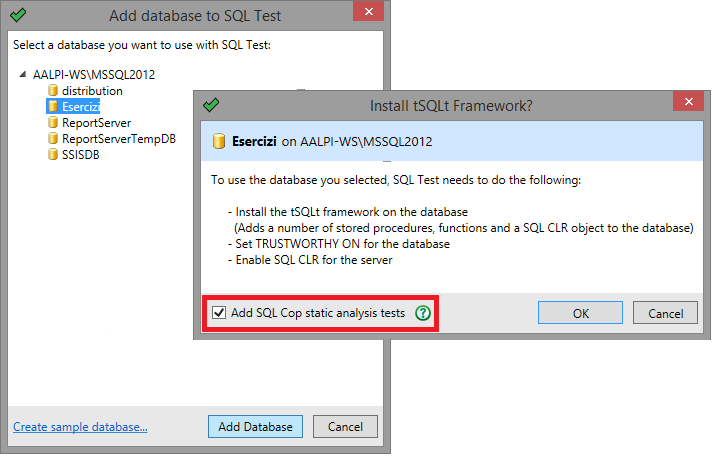
L’installazione è molto semplice, e in pochi click, tramite un wizard, il nostro Management Studio è dotato di una nuova finestra ancorata. L’installazione del tool tuttavia, non include il setup degli oggetti sui database.

* 1. 

Per aggiungere i framework è sufficiente premere il tasto

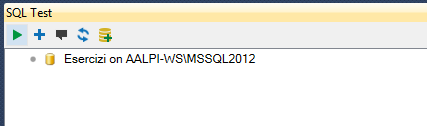
* 1. 

Apparirà quanto indicato nella figura seguente:

* 1. 

Da notare la possibilità di installare anche SQLCop, del tutto facoltativa. Alla fine dell’operazione avremo quanto indicato nella finestra di dialogo. Saranno presenti oggetti sotto uno schema *tSQLt*, verrà impostato il *TRUSTWORTHY* ad *ON* per il database e un oggetto CLR sarà aggiunto tra gli assembly.

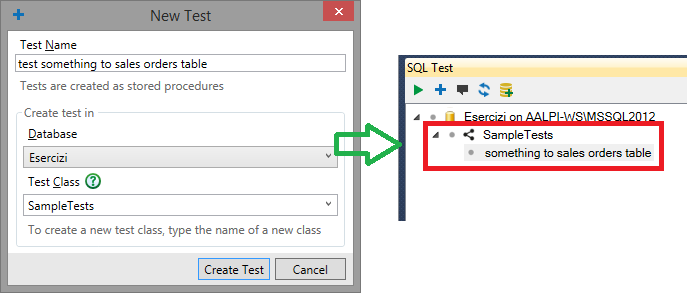
Ecco cosa avremo:

* 1. 

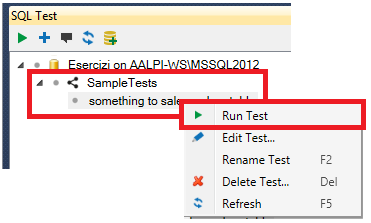
Da ora sarà possibile creare *TestClass* e test, direttamente con i menu contestuali disponibili. Tutto quello che andiamo a modificare si interfaccia con le procedure di cui abbiamo parlato nella [sezione *tSQLt*](#_tSQLt). Potremo quindi creare e modificare direttamente da questa interfaccia, senza conoscere le stored procedure di tSQLt. Per creare un test, ad esempio, è possibile fare click sul pulsante

* 1. 

e seguire le istruzioni indicate nella seguente immagine:

* 1. 

Eseguire test è altrettanto semplice:

* 1. 

In caso di esito positivo verrà segnalato un segno di spunta verde, in caso di esito negativo, una X rossa ed una serie di messaggi nel tab relativo ai risultati.

### Comparazione delle fasi

Nella seguente tabella andiamo a comparare le fasi dei framework e dei tool utilizzabili per eseguire unit test:

|  |  |  |  |
| --- | --- | --- | --- |
|  | tSQLt/SQLTest | tSQLUnit | Visual Studio |
| Fase di raccolta/creazione | Fake ed isolamento oggetti | Setup | Pre-Test script |
| Fase di applicazione logiche | Act | Act | Test |
| Fase di verifica | Assert | Assert | Conditions |
| Fase di ripristino della situazione | Fine del test, implicita | Teardown | Post-Test script |

## Conclusioni

Al di là dei tool che si utilizzano per testare gli oggetti ed i dati dei nostri database, è piuttosto chiaro che il raggiungimento di un’alta qualità del codice non è più un’attività da sottovalutare. Parlando di database, e quindi soprattutto per chi sviluppa stored procedure e per chi definisce comportamenti ben definiti sui dati, è molto importante applicare test unitari. Come abbiamo detto all’inizio dell’articolo, le principali motivazioni sono la precocità con cui risolvere i bug e la prevenzione in materia di regressioni. Come nel caso del source control abbiamo sempre più strumenti per muoverci in questo mondo, che è piuttosto nuovo in materia di basi di dati. Inoltre, i framework sono free e si ispirano a sintassi già molto conosciute nel mondo dello sviluppo. Quindi il consiglio è di creare test, laddove siano importanti per la qualità del software, ma di non farsi “prendere la mano”. Creare un test deve dare vantaggi, non perdita di tempo, esattamente come vale per il codice.

## Risorse

[tSQLt framework](http://tsqlt.org/) download

[tSQLUnit framework](http://sourceforge.net/projects/tsqlunit/) download

[Esempi di tSQLUnit su MSDN Code Samples](https://code.msdn.microsoft.com/tSQLUnit-samples-Database-54b10e47) download

[SQLCop framework](http://sqlcop.lessthandot.com/) download

[SS-Unit framework](http://sqlcop.lessthandot.com/) download

[Visual Studio SSDT](http://msdn.microsoft.com/it-it/data/tools.aspx) help

[Visual Studio Unit Test Projects](http://msdn.microsoft.com/en-us/library/hh598957.aspx) help

[Red-Gate SQL Test](http://www.red-gate.com/products/sql-development/sql-test/) homepage