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

In [un precedente articolo](http://msdn.microsoft.com/it-it/library/dn383992.aspx) abbiamo già parlato di Continuous Integration e di [DLM](http://msdn.microsoft.com/en-us/library/jj907294.aspx). Abbiamo poi parlato più nel dettaglio del primo step di CI, ovvero di [come mettere il database sotto source control](https://msdn.microsoft.com/it-it/library/dn894015.aspx). Con quanto segue andremo ad approfondire il concetto di unit test e, nella fattispecie, del testing tramite il framework free tSQLt, utilizzando t-sql e SQL Server Management Studio.

## Cosa si intende per unit test

In ingegneria del software, per unit testing, si intende l’attività di collaudo di singole unità software. Per “singole unità” si intendono porzioni di codice sorgente quali moduli, funzioni, procedure, metodi o classi (in programmazione ad oggetti). Lo scopo principale di questo approccio è di trovare bug e prevenire regressioni (*Fonte: Wikipedia*).

Per la programmazione, un test corrisponde ad un “metodo” particolare (corredato di attributi e/o naming convention) che “esegue” metodi reali, ovvero quelli che contengono la logica di business, al fine di verificare i comportamenti che da essi ci si aspetta.

Ad esempio, supponiamo di avere un metodo di classe chiamato *Power(x, y)*, un test potrebbe essere un altro metodo chiamato *Power\_should\_return\_eight\_when\_x\_is\_two\_and\_y\_is\_three()* il cui corpo esegua il metodo *Power(2, 3)* e che il cui risultato atteso sia il valore 8.

In questo caso, il medesimo test su database SQL Server è una stored procedure chiamata ad esempio *Power\_should\_return\_eight\_when\_x\_is\_two\_and\_y\_is\_three* che esegue la funzione definita dall’utente *udf\_Power(x, y)* con x uguale a 2 ed y uguale a 3.

## Perché fare unit test

Tanti sono i benefici ricavati dall’applicazione di test unitari. Come già indica la definizione, la pratica dello unit testing può prevenire la maggior parte delle regressioni ed aiuta lo sviluppatore a capire se ci saranno bug, in base a quanto scritto. Di conseguenza, la fix sarà immediata, non appena il bug è evidenziato dai test stessi.

Nella mia esperienza ho notato che la qualità di quanto scritto dal nostro team di sviluppo è costantemente migliorata, grazie anche all’interoperabilità tra test e codice, i quali lavorano alla perfezione in sinergia per ottenere più affidabilità e leggibilità.

Tutto questo va in concomitanza con la nostra migrazione da un approccio seriale ad uno agile, evolutivo. Un tempo eravamo abituati a raccogliere informazioni per lungo tempo, a fare analisi molto complesse e profonde (addirittura previsionali su funzionalità che ancora nemmeno si erano mai considerate) ancora prima di implementare ogni sviluppo. Lo scopo era quello di ottenere un design valido per molti casi ed un modello a database pronto per modifiche future, senza però conoscere in che direzione ci si sarebbe diretti. Dopo la fase di design, seguiva un lungo periodo di code and fix continui, unitamente a continui deploy derivanti dalle “riparazioni” o dai cambi repentini di idee o di direzione del progetto.

Adesso stiamo applicando SCRUM, e quindi seguiamo iterazioni time-boxed con cerimonie fisse e rilasci costanti nel tempo. Con questo tipo di processi, più snelli ed orientati alla continua evoluzione del codice e delle basi dati, il nostro codice è diventato molto flessibile e le richieste di implementazione dai nostri stakeholder sono più veloci e di qualità.

Il test ha un peso molto importante per questo. Con i test eseguiti ad ogni “compilazione” abbiamo reso molto più affidabile il processo di deploy poiché molti dei problemi vengono già scovati durante la fase di collaudo automatizzato. Inoltre, creando test, abbiamo codice auto documentato poiché un test spiega già nel nome cosa un metodo dovrebbe fare, soprattutto da un punto di vista funzionale.

Infine, non ultimo per importanza, un metodo “coperto” da test, è un metodo scritto in “maniera testabile”, ovvero semplice, che fa il suo lavoro, chiaro e conciso (e questo segue parte dei principi SOLID). Questo, per ovvi motivi, riduce la possibilità di bug nati da logiche troppo complesse e da metodi con troppi punti di rottura.

## Uno scenario reale

Andiamo ora a discutere due scenari reali nei quali ho trovato il testing veramente utile ed efficace. Immaginiamo di essere consulenti in un’azienda che implementa soluzioni ERP e di essere chiamati a risolvere alcuni problemi incontrati dal team di sviluppo di quell’azienda durante l’implementazione di una funzionalità di ricerca. Quest’ultima viene effettuata da una stored procedure, ovvero l’oggetto indicato come problematico.

In questo caso il primo step che affronteremo è quello in cui mettere la procedure sotto copertura di test. Il successivo sarà l’utilizzo dei test per controllare regressioni dopo la modifica evolutiva della procedura stessa.

### La procedura SearchUsers

La stored procedure *SearchUsers* ricerca utenti in una tabella tramite diversi filtri. Gli sviluppatori hanno effettuato su di essa tanti cambiamenti nel tempo e, come il cliente ripete più volte, ogni cambiamento rilasciato porta ad una serie di “imprevedibili” problemi nell’applicazione.

La richiesta degli stakeholder in origine era quella di filtrare la lista degli utenti tramite informazioni differenti per valore e natura (nel nostro caso per id utente e per “cognome che inizia per”). Tutti questi filtri dovevano essere mutualmente esclusivi e la stored procedure doveva essere sempre la stessa ed unica (dal design tecnico richiesto). L’elenco degli utenti ritornati doveva inoltre essere corredato di dettagli aggiuntivi, residenti su altre tabelle. Nel caso in cui questi ultimi non fossero presenti, almeno l’informazione generale dell’utente doveva essere ritornata al chiamante.

Come possiamo dedurre, abbiamo a che fare con comportamenti diversi, ma il risultato ha sempre lo stesso numero di colonne. Inoltre, abbiamo almeno una JOIN, che ci consente di ottenere i dettagli aggiuntivi descritti precedentemente.

Dopo aver aperto la stored procedure per la prima volta, notiamo che l’implementazione è stata gestita in maniera incoerente. In qualche caso il risultato è mancante, altre volte è vuoto, altre volte torna più righe quando dovrebbe tornarne solo una, e via discorrendo. L’applicazione non conosce queste considerazioni e quindi non ha il minimo controllo sul ritorno.

Il modello dello scenario è il seguente:

* 1. -- dbo.Users table
  2. CREATE TABLE dbo.Users
  3. (
  4. UserId int IDENTITY(1, 1) NOT NULL PRIMARY KEY CLUSTERED
  5. , Username varchar(30) NOT NULL
  6. , FirstName varchar(30) NOT NULL
  7. , LastName varchar(30) NOT NULL
  8. , BirthDate date NOT NULL
  9. );
  10. GO
  11. -- dbo.UserContacts table
  12. CREATE TABLE dbo.UserContacts
  13. (
  14. UserId int NOT NULL PRIMARY KEY CLUSTERED
  15. , Street varchar(50) NOT NULL
  16. , Town varchar(30) NOT NULL
  17. , ZipCode char(5) NOT NULL
  18. , StateOrProvince varchar(30) NOT NULL
  19. );
  20. GO
  21. -- FKs
  22. ALTER TABLE dbo.UserContacts
  23. ADD CONSTRAINT FK\_Users\_UserContacts FOREIGN KEY (UserId)
  24. REFERENCES dbo.Users (UserId);
  25. GO
  26. -- customer procedure
  27. CREATE PROCEDURE dbo.SearchUsers
  28. @UserId int = NULL
  29. , @LastName varchar(30) = NULL
  30. AS
  31. BEGIN
  32. SET NOCOUNT ON;
  34. IF @UserId IS NOT NULL
  35. OR @LastName IS NOT NULL
  36. BEGIN
  37. SELECT
  38. U.UserId
  39. , U.Username
  40. , U.FirstName
  41. , U.LastName
  42. , U.BirthDate
  43. , UC.Street
  44. , UC.Town
  45. , UC.ZipCode
  46. , UC.StateOrProvince
  47. FROM
  48. dbo.Users U
  49. JOIN dbo.UserContacts UC ON UC.UserId = U.UserId
  50. WHERE
  51. @UserId IS NULL OR U.UserId = @UserId
  52. AND (@LastName IS NULL OR U.LastName LIKE @LastName + '%');
  53. END;
  54. END;
  55. GO

#### Cosa vogliamo testare

Ora, vogliamo coprire la nostra stored procedure con dei test unitari. Per ogni “metodo” che vogliamo testare, esistono potenzialmente tantissimi test possibili, ma sta a noi selezionare quelli che prevengono al meglio le regressioni, senza trasformare il tempo investito nel test in tempo perso per eccesso di zelo.

Ecco i casi che andremo a verificare:

|  |  |
| --- | --- |
| Parametro valutato | Risultato desiderato |
| ID Utente | Resultset con un record |
| ID Utente con dettagli mancanti | Resultset con un record |
| Nessun parametro passato | Resultset con zero record |

#### Tecnologia: tSQLt

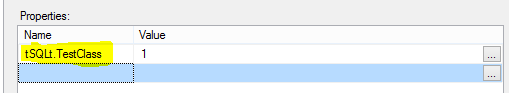
Il framework che utilizzeremo è tSQLt.

Il primo test controllerà il numero di righe ritornate, una, in base ad un user id esistente. Il secondo test controllerà sempre il numero di righe ritornate, una, in base ad un user id esistente ma privo di dettagli ad esso legati. Infine, l’ultimo test controllerà se un resultset vuoto viene restituito nel caso in cui vengano passati alla procedura tutti parametri a NULL.

Prima di implementare le vere e proprie procedure di test, andiamo a creare una *test class* che le terrà organizzate a livello logico. Utilizzeremo la procedura *tSQLt.NewTestClass* come segue:

* 1. EXEC tSQLt.NewTestClass @ClassName = N'UserTests';

Essa crea un nuovo schema chiamato “UserTests” munito di una extended property che definisce l’appartenenza agli schema per le *test class*, come mostrato nella seguente immagine:

* 1. 

La proprietà *tSQLt.TestClass* indica che “UserTests” è una *test class*.

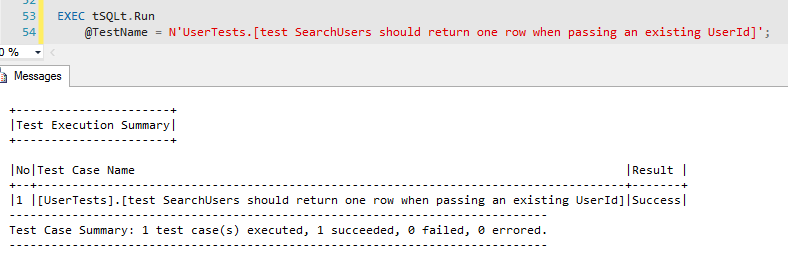
#### Test – chiamata con user id

Ecco il test per il primo caso descritto:

* 1. CREATE PROCEDURE UserTests.[test SearchUsers should return one row when passing an existing UserId]
  2. AS
  3. BEGIN
  5. -- Assemble (crea i dati di test)
  6. -- 1 – crea i fake delle tabelle Users e UserContacts
  7. -- (isola le tabelle creandone una copia con il nome originale
  8. -- e cambiando il nome della tabella di partenza con un nome temporaneo)
  9. EXEC tSQLt.FakeTable
  10. @TableName = N'Users'
  11. , @SchemaName = N'dbo'
  12. , @Identity = 0; -- non mantiene l’identity (il fake non ne ha bisogno in questo caso)
  13. EXEC tSQLt.FakeTable
  14. @TableName = N'UserContacts'
  15. , @SchemaName = N'dbo';
  16. -- 2 – inserisce i dati “stub”
  17. INSERT INTO dbo.Users (UserId, Username, FirstName, LastName, BirthDate)
  18. VALUES
  19. (1, 'suxstellino', 'Alessandro', 'Alpi', '19810422'),
  20. (2, 'ettagab', 'Gabriele', 'Etta', '19940725');
  22. INSERT INTO dbo.UserContacts (UserId, Street, Town, ZipCode, StateOrProvince)
  23. VALUES
  24. (1, 'Allen street', 'New York', '12345', 'NY'),
  25. (2, 'Madison avenue', 'New York', '12345', 'NY');
  27. -- Act (applica le logiche)
  28. -- crea una tabella temporanea che conterrà i risultati
  29. DECLARE @Results table
  30. (
  31. UserId int
  32. , Username varchar(30)
  33. , FirstName varchar(30)
  34. , LastName varchar(30)
  35. , BirthDate date
  36. , Street varchar(50)
  37. , Town varchar(30)
  38. , ZipCode char(5)
  39. , StateOrProvince varchar(30)
  40. )
  41. -- esegue la procedura da testare ed inserisce i risultati nella tabella appena creata
  42. INSERT INTO @Results (UserId, Username, FirstName, LastName, BirthDate, Street, Town, ZipCode, StateOrProvince)
  43. EXEC dbo.SearchUsers
  44. @UserId = 1;
  46. -- viene segnato il numero di record tornato
  47. DECLARE @NumberOfRows int = -1;
  48. SELECT @NumberOfRows = **COUNT**(1) FROM @Results;
  49. -- Assert (Asserzioni e verifiche)
  50. EXEC tSQLt.AssertEquals
  51. @Expected = 1
  52. , @Actual = @NumberOfRows
  53. , @Message = N'Wrong number of rows!';
  55. END;
  56. GO

Come possiamo notare, il nome della stored procedure inizia per “test”. Questa è una naming convention di tSQLt. La prima parte del corpo del test è la porzione denominata *Assemble*. In essa andiamo a creare dei fake (copie di oggetti originali, con lo stesso nome) per isolare le tabelle su cui lavoreremo nel contesto del test, e successivamente andiamo ad aggiungere dei record di esempio, per poi eseguire la procedura *SearchUsers* (sezione *Act*). Il resultset ritornato viene salvato in una tabella temporanea e il numero di righe è persistito in una variabile chiamata @NumberOfRows. L’ultima porzione di codice è la *Assert*. Qui andiamo ad eseguire le nostre asserzioni, e nella fattispecie lanciamo la procedura *tSQLt.AssertEquals* per verificare che il valore di @NumberOfRows sia il medesimo di quello atteso (ovvero 1).

Eseguendo la procedura *tSQLt.Run* avremo i seguenti risultati:

* 1. 

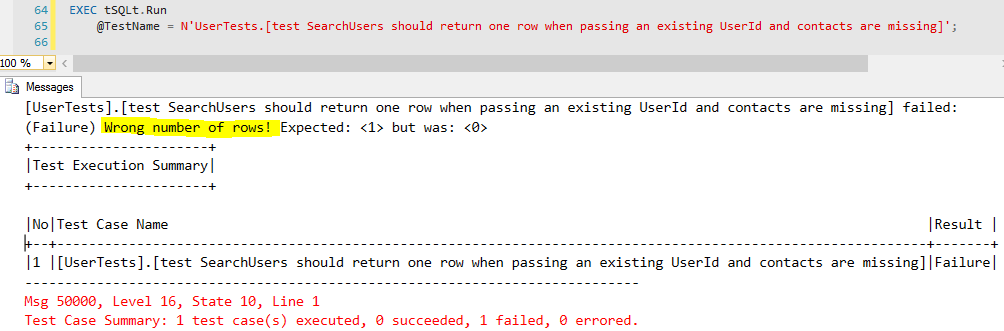
Il test è stato eseguito con successo sul database del cliente.

#### Test – chiamata con user id di un utente privo di dettagli

Il secondo test è del tutto simile al primo, ma verte su un utente che non ha dettagli (UserContacts) legati. Ecco come è scritto:

* 1. CREATE PROCEDURE UserTests.[test SearchUsers should return one row when passing an existing UserId and contacts are missing]
  2. AS
  3. BEGIN
  5. -- Assemble
  6. EXEC tSQLt.FakeTable
  7. @TableName = N'Users'
  8. , @SchemaName = N'dbo'
  9. , @Identity = 0;
  10. EXEC tSQLt.FakeTable
  11. @TableName = N'UserContacts'
  12. , @SchemaName = N'dbo';
  13. INSERT INTO dbo.Users (UserId, Username, FirstName, LastName, BirthDate)
  14. VALUES (2, 'ettagab', 'Gabriele', 'Etta', '19940725');
  16. -- Act
  17. DECLARE @Results table
  18. (
  19. UserId int
  20. , Username varchar(30)
  21. , FirstName varchar(30)
  22. , LastName varchar(30)
  23. , BirthDate date
  24. , Street varchar(50)
  25. , Town varchar(30)
  26. , ZipCode char(5)
  27. , StateOrProvince varchar(30)
  28. )
  29. INSERT INTO @Results (UserId, Username, FirstName, LastName, BirthDate, Street, Town, ZipCode, StateOrProvince)
  30. EXEC dbo.SearchUsers
  31. @UserId = 2;
  33. DECLARE @NumberOfRows int = -1;
  34. SELECT @NumberOfRows = **COUNT**(1) FROM @Results;
  35. -- Assert
  36. EXEC tSQLt.AssertEquals
  37. @Expected = 1
  38. , @Actual = @NumberOfRows
  39. , @Message = N'Wrong number of rows!';
  41. END;
  42. GO

Come possiamo notare, la tabella *UserContacts* non ha record di esempio. Il test fallisce, poiché la procedura *SearchUsers* sta utilizzando una JOIN invece che una LEFT JOIN. Il risultato è il seguente:

* 1. 

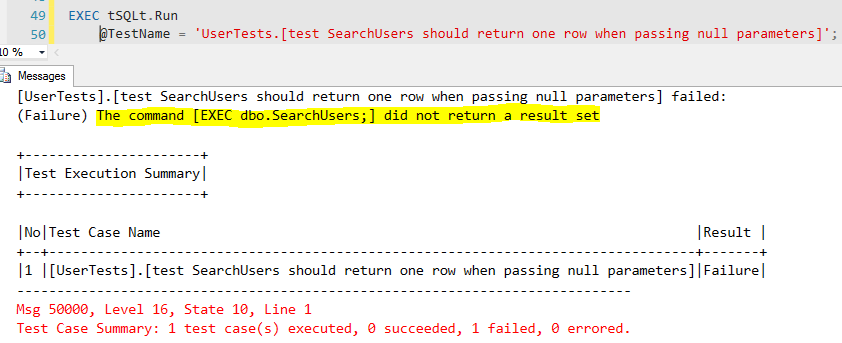
Dopo il fallimento del test, il team di sviluppo del cliente ha applicato la fix aggiungendo la LEFT JOIN al posto della INNER JOIN.

#### Test – nessun parametro specificato

Per il terzo test, ecco l’implementazione:

* 1. CREATE PROCEDURE UserTests.[test SearchUsers should return one row when passing null parameters]
  2. AS
  3. BEGIN
  5. -- Assemble
  6. EXEC tSQLt.FakeTable
  7. @TableName = N'Users'
  8. , @SchemaName = N'dbo'
  9. , @Identity = 0;
  10. EXEC tSQLt.FakeTable
  11. @TableName = N'UserContacts'
  12. , @SchemaName = N'dbo';
  14. -- Act
  16. -- Assert
  17. EXEC tSQLt.AssertResultSetsHaveSameMetaData
  18. @expectedCommand = N'EXEC dbo.SearchUsers;'
  19. , @actualCommand = N'SET FMTONLY ON;
  20. SELECT
  21. U.UserId
  22. , U.Username
  23. , U.FirstName
  24. , U.LastName
  25. , U.BirthDate
  26. , C.Street
  27. , C.Town
  28. , C.ZipCode
  29. , C.StateOrProvince
  30. FROM
  31. dbo.Users U
  32. JOIN dbo.UserContacts C ON U.UserId = C.UserId;';
  33. END;
  34. GO

Questo test è differente rispetto ai precedenti. Facciamo semplicemente dei fake e lanciamo la procedura *tSQLt.AssertResultSetsHaveSameMetaData*, che verifica che due comandi abbiano gli stessi metadati (colonne). Il test fallisce, poiché nella procedura SearchUsers vi è una condizione che fa saltare il ritorno del resultset. Otterremo il messaggio seguente:

* 1. 

I metadati non possono essere testati, poiché la *SearchUsers* non ritorna metadati in questo caso. Il team di sviluppo ha risolto spostando la condizione prima della chiamata a database, rimuovendola, di fatto, dalla stored procedure e applicandola a livello applicativo.

## Secondo scenario – refactor di database

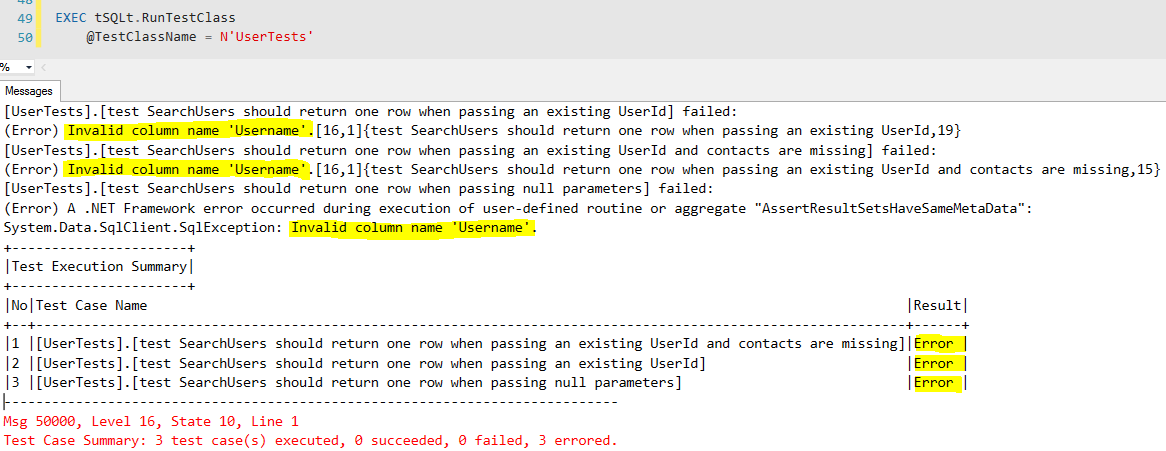
Lo stesso team di sviluppo di cui abbiamo parlato finora, ha trovato estremamente comodo ed utile l’applicazione dei test sopra descritti anche durante le successive fasi di refactor del database. Uno scenario che si è verificato è quello della modifica di una delle tabelle coperte dai test, la tabella *Users*. Gli sviluppatori lato database avrebbero dovuto spostare una colonna da questa tabella ad una nuova. Dopo la modifica al database, il team di sviluppo dell’applicazione ha iniziato a scaricare i cambiamenti ed eseguire in successione gli unit test.

Le modifiche a database ricevute sono le seguenti:

* 1. CREATE TABLE dbo.UserCredentials
  2. (
  3. UserId int NOT NULL PRIMARY KEY CLUSTERED
  4. , Username varchar(30) NOT NULL
  5. , Pwd varchar(30) NOT NULL
  6. , MustChangePassword bit NOT NULL DEFAULT(0)
  7. );
  8. GO
  9. -- [moving data]
  10. ALTER TABLE dbo.Users DROP COLUMN Username;
  11. GO

#### Ancora un’esecuzione dei test

Come possiamo notare, dopo che la colonna è stata eliminata dalla tabella *Users*, tutti i test eseguiti sono stati eseguiti, ad uno ad uno, con un ritorno di errore. Attenzione, perché un errore nel test non è un test fallito necessariamente, è un test che contiene una o più eccezioni. Per eseguire i test è stata usata la stored procedure *tSQLt.RunTestClass*, la quale esegue tutti i test all’interno di una classe:

* 1. 

Il test non è stato completamente eseguito, tuttavia ha allertato all’istante il team di sviluppo, poiché la colonna “Username” non esiste più. In questo modo, gli sviluppatori sono andati a cambiare la stored procedure *SearchUsers*, prima di procedere col rilascio definitivo. Questo è stato fondamentale per prevenire una regressione che avrebbe interrotto completamente il codice che consuma la stored procedure.

## Conclusioni

I test unitari (o Unit Test) sono una delle pratiche più importanti da applicare quando vogliamo prevenire regressioni ed essere allertati in caso di sicuro bug. Di conseguenza, anche i nostri rilasci saranno molto più affidabili. Il design evolutivo di database lavora di pari passo con i test. Infatti, una stored procedure, scritta in “maniera testabile”, supporta modifiche snelle e versatili, per stare al passo con i requisiti di business. Più le modifiche sono “leggere” e testate/testabili, più la possibilità di avere successo nei rilasci è concreta. Inoltre il test, oltre ad essere completamente “auto documentato” a livello di funzionalità, suggerisce i cambiamenti da fare in caso di manutenzione evolutiva.

Ora abbiamo gli strumenti, non è più obbligatorio scriversi manualmente framework e non è più necessario scrivere procedure proprietarie di test. Possiamo usare i framework disponibili per t-sql oppure tool forniti di interfaccia grafica (some [SQL Test di Red-Gate](http://www.red-gate.com/products/sql-development/sql-test/)) e possiamo anche automatizzare i processi di test in ambienti di Continuous Integration.

## Risorse

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

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

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

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

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