



Webinar: **Diventa un mago del Testing!**

Strumenti e tecniche per il Testing in Delphi

begin

Marco Breveglieri

*Software and Web Developer,
Teacher and Consultant*

@ABLS Team - Reggio Emilia, ITALY

Homepage

<https://www.breveglieri.it>

Blog

<https://www.compilaquindiva.com>

Delphi Podcast

<https://www.delhipodcast.com>

Twitter

[@mbreveglieri](https://twitter.com/mbreveglieri)



Introduzione



Agenda

- Il Testing: vantaggi e benefici, avvertenze per l'uso
- Unit Test, Integration Test e altri
- Creare Unit Test “come si deve”
- Mock, Stub e altri amici
- Uso di Test Framework
- Tool per il testing in Delphi
- Esperienze e conclusioni
- Q & A

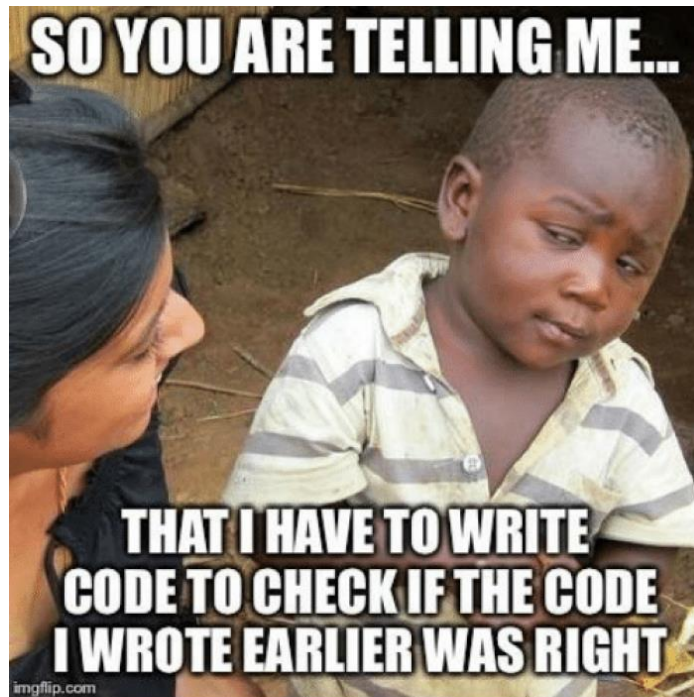


Benefici del Testing

- Identificazione e risoluzione (preventiva) di bug nel software
- Semplificazione del mantenimento di codice “legacy”
- Incremento della fiducia dello sviluppatore nella qualità del proprio codice (e in quello degli altri)
- Forzatura del rispetto dei principi SOLID e altri pattern di buona architettura
- Controllo dei difetti di regressione nel rilascio di nuovo software
- Censimento dei casi di errore riscontrati dai clienti (ed evitare che si ripetano)
- Trampolino di lancio nell'uso di tecnologie avanzate di sviluppo (es. *TDD*)
- Abilitazione e soddisfazione requisiti per implementare *Continuous Integration / Delivery*

Scettici?

- Milioni di righe di codice vengono testate in modo “automatizzato”
- Molte software house adottano tecniche TDD (Test Driven Development)
- Non è (o non dovrebbe essere) un argomento nuovo, anzi...



Writing tests

Deve essere fatto nel modo corretto

- Esistono regole per scrivere test “fatti bene” che vanno rispettate
- Implementare test in modo errato può portare più danni che benefici
- Scrivere test deve essere “divertente” (nonostante le “deadline” incombenti)
- Possono richiedere più tempo per lo sviluppo, ma è possibile recuperarlo
- Il codice deve essere testabile: usate i **principi SOLID**!



Obiettivo

**Diventare
un... “mago”
del Testing!**



L'ABC





Unit Test: una definizione

*A **Unit Test** is a piece of code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward.*

If the assumptions turn out to be wrong, the unit test has failed.

A «unit» is a method or function.

Unit Test: la morfologia

Somma delle azioni che hanno luogo invocando un metodo pubblico del sistema sotto test e producono un risultato finale che può essere osservato.

Il risultato finale può essere

- un valore restituito dal metodo invocato se è una funzione
- un cambiamento verificabile nello stato dell'oggetto sottoposto a test
- la chiamata di un metodo su una dipendenza dell'oggetto testato



Caratteristiche di un buon Unit Test

- Automatizzato
- Ripetibile
- Consistente
- Facile
- Accessibile
- Riutilizzabile
- Veloce
- Onnipotente
(rispetto al sistema sottoposto a test)

Integration Test

Integration testing is testing a unit of work with one or more of its real dependencies such as time, network, database, or code you cannot control such as threads and random number generators.



Le differenze



Unit Testing

- Dipendenze isolate e simulate
- Nessun setup richiesto
- Nessun cleanup necessario
- Devono essere tanti
- Devono essere veloci

Integration Testing

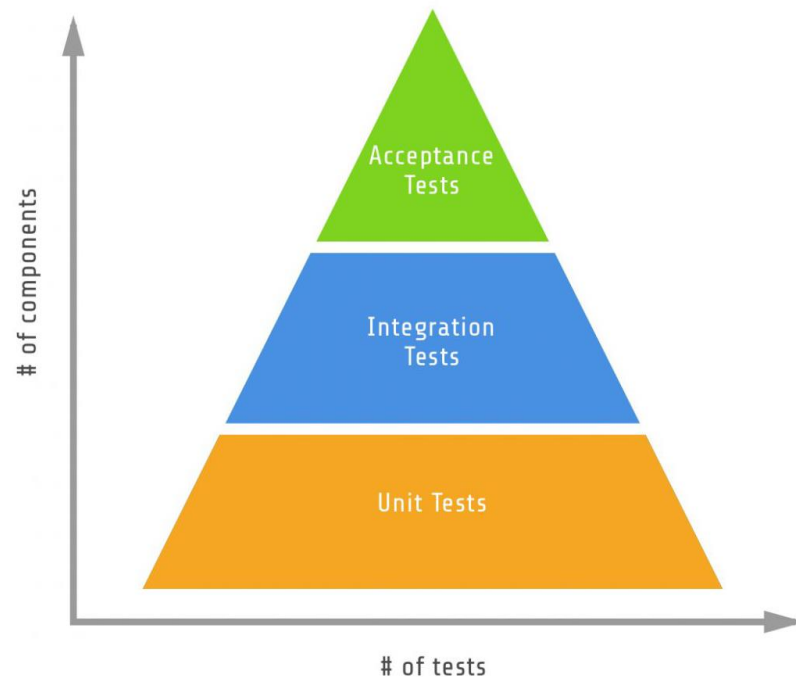
- Dipendenze concrete (solo alcune)
- Possibile inizializzazione necessaria
- Possibile cleanup richiesto
- Sono meno rispetto agli Unit Test
- Possono essere più lenti

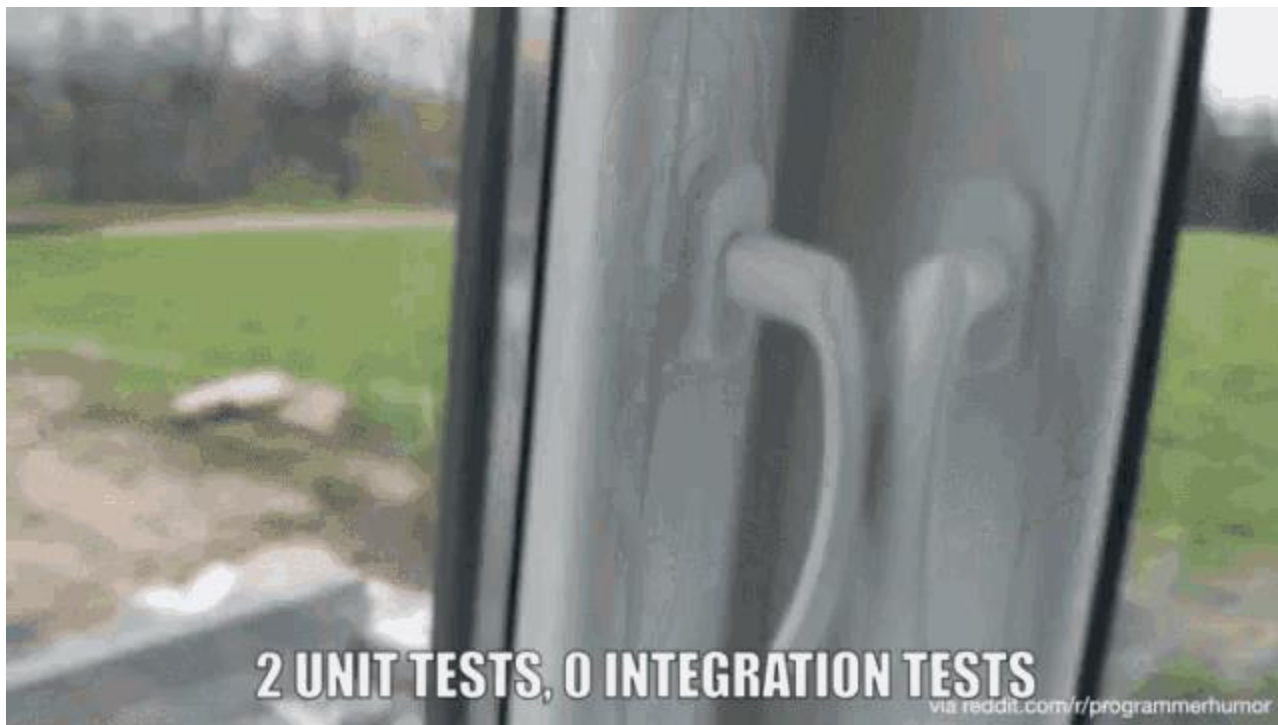
Entrambi sono necessari

The Pyramid of Tests

Per approfondimento:

<https://martinfowler.com/articles/practical-test-pyramid.html>





Unit Test vs Integration Test

Unit test vs. Integration test



Unit Test vs Integration Test

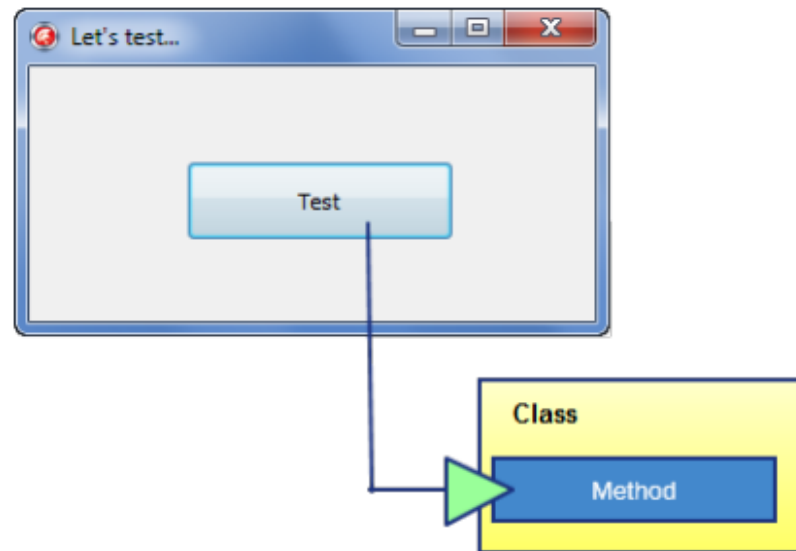


Unit Test vs Integration Test

Unit Test a regola d'arte



Il più semplice degli Unit Test

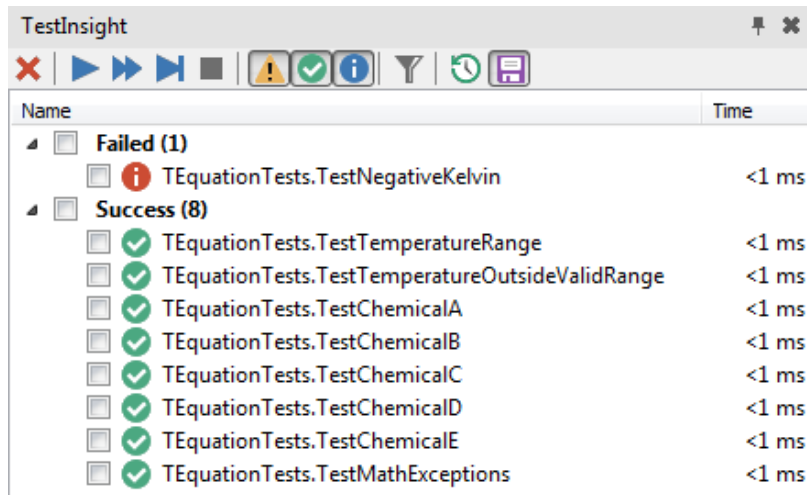
Rapido, immediato e pratico,
ma forse poco versatile e professionale. 😊



Ma forse è meglio usare tool più specifici...

ad esempio

- **Test Framework**
 - DUnit, DUnit2
 - DUnitX 
- **Test Tools**
 - TestInsight 





Test Framework

E' indispensabile adoperare un Test Framework.

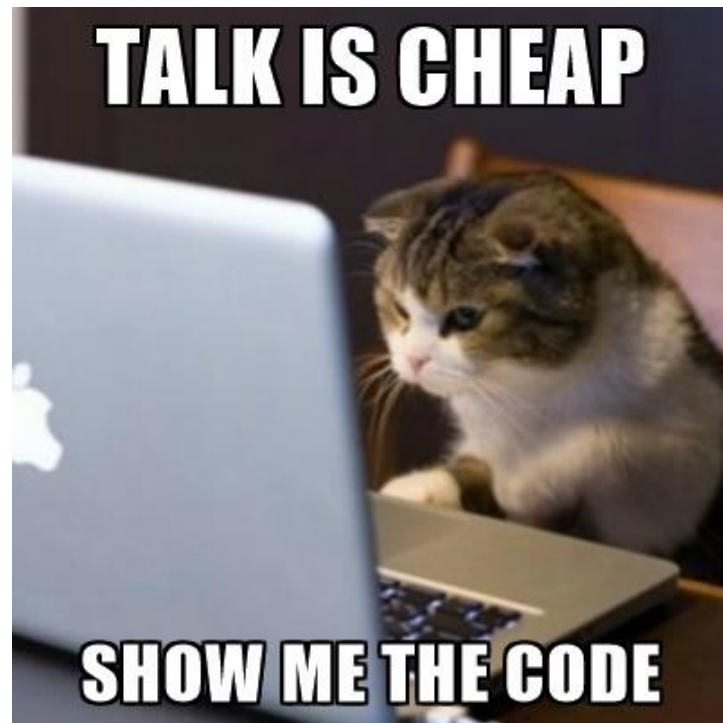
NOTA: le versioni recenti di Delphi (es. 10.3 Rio) sono equipaggiate con **DUnitX**.

- Consente di dare una struttura ai test
- Rende i test automatizzati e ripetibili
- Riduce gli errori nel codice dei test
- Produce un rendiconto dell'esito dei test
- Fornisce all'occorrenza una GUI
- Può fornire (o meno) supporto alla gestione delle dipendenze del sistema sotto test
- Può dialogare con l'eventuale sistema di Continuous Integration & Delivery

Demo

Vediamo assieme

- *un package di esempio*
- *un progetto di test con DUnitX*
- *esempi di Unit Testing*



Piccolo glossario

- **Test Fixture / Test Suite / Test Class:** classe che raggruppa una serie di test
- **Setup (Method):** metodo che inizializza il sistema per l'esecuzione di un test
- **TearDown (Method):** metodo che finalizza il sistema dopo l'esecuzione di un test
- **Test (Method) o Test Case:** metodo che implementa un test specifico (lo *Unit Test*!)



Unit Test: le azioni

1. **Assign**

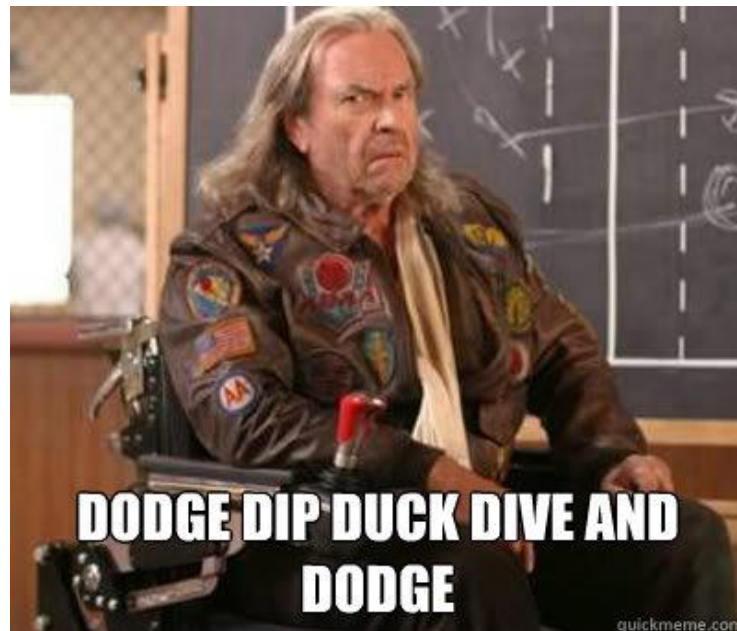
Creiamo gli oggetti e assegnamo i valori iniziali alle variabili

2. **Act**

Agiamo sull'oggetto sottoposto a test (es. chiamiamo un metodo)

3. **Assert**

Verifichiamo che qualcosa corrisponda a quello che ci attendiamo



Unit Test: le regole di base

- Attribuire al test un nome significativo
- Seguire una convenzione nel naming
- Non eseguire più di una asserzione
- Non usare valori apparentemente insoliti
- Non introdurre logica complessa nel test
(Cyclomatic Complexity = 1)
- Non dipendere mai da un test precedente



**KEEP
CALM
&
FOLLOW
THE RULES**

Un momento...

*Come faccio se l'oggetto da
testare ha delle dipendenze?*





Dipendenza

“Una dipendenza esterna è un oggetto del sistema con cui il codice sottoposto a test interagisce e sul quale non ha controllo”.

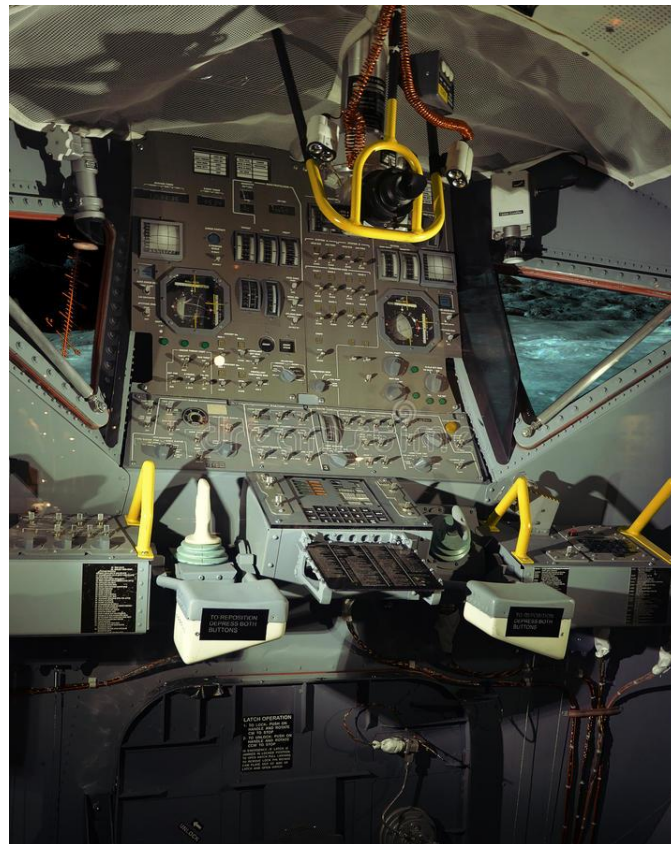
- File System
- Network
- Database
- I/O

(in breve qualsiasi implementazione...)

Stub alla riscossa!

Lo stub è un oggetto “controllabile”
che sostituisce una dipendenza reale.

- Restituisce valori alla classe sotto test
- Contiene una implementazione parziale
- I valori restituiti sono quelli che forniscono la risposta attesa per l’asserzione che si intende verificare

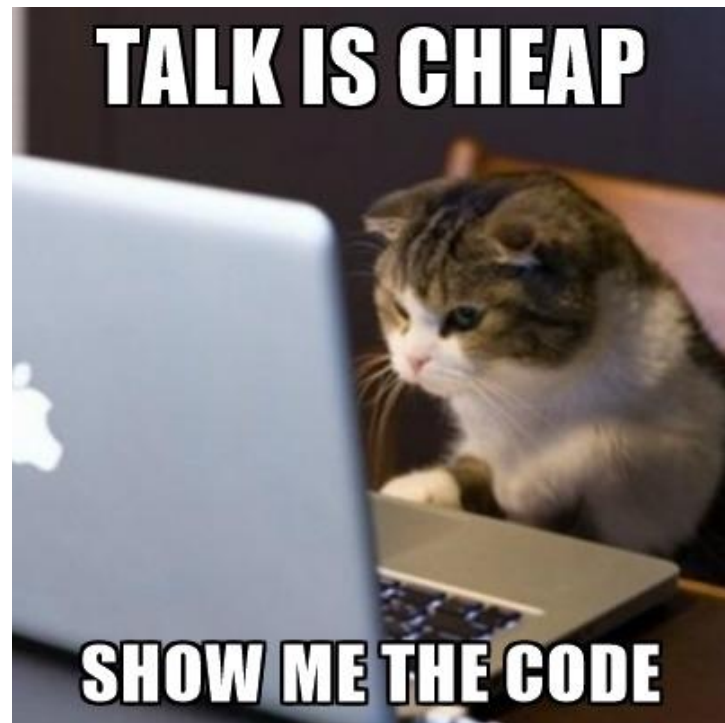




Demo

Vediamo assieme

- *un nuovo package*
- *un nuovo progetto di test con DUnitX*
- *come usare gli “stub”*



Un momento...

*E se io volessi verificare delle
asserzioni basate su come
l'oggetto interagisce con altri,
ad esempio le dipendenze?*



State-Based Testing

Il controllo dell'esito del test si basa sullo stato finale dell'oggetto coinvolto (*result driven*).

- L'erba del prato è verde?
- La terra è sufficientemente umida?



Interaction-Based Testing

Il controllo dell'esito del test si basa sulla interazione con oggetti che ricevono input o forniscono output (*action driven*).

- Quante volte viene irrigato il campo in un lasso di tempo?
- Quanta acqua viene erogata ogni volta?



Mock alla riscossa!

Il mock è un oggetto che interagisce con il sistema sotto test e indica se il test è stato superato oppure no.

- Attende di essere coinvolto dall'oggetto sottoposto a test
- Può subire la chiamata a uno o più metodi
- Verifica di essere usato in modo corretto



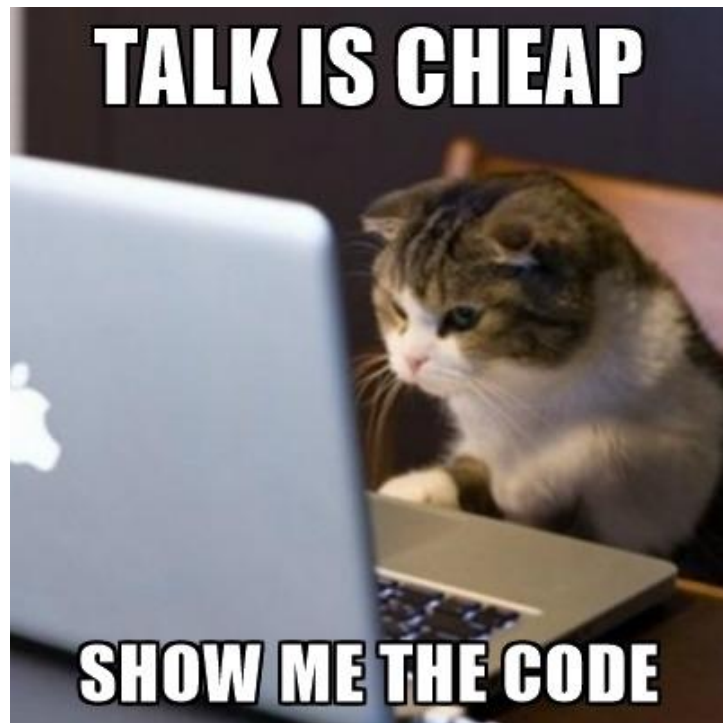
ATTENZIONE: ogni test dovrà usare uno e un solo mock!



Demo

Vediamo assieme

- *come creare e usare “mock”*



Scrivere “fake” è faticoso

- Scrivere il codice delle classi può richiedere molto tempo
- Le classi possono diventare complesse all’aumentare del numero di condizioni da verificare
- Le classi di stub/mock possono diventare molto numerose, non riutilizzabili e ingestibili
- Possono compromettere l’efficacia del test

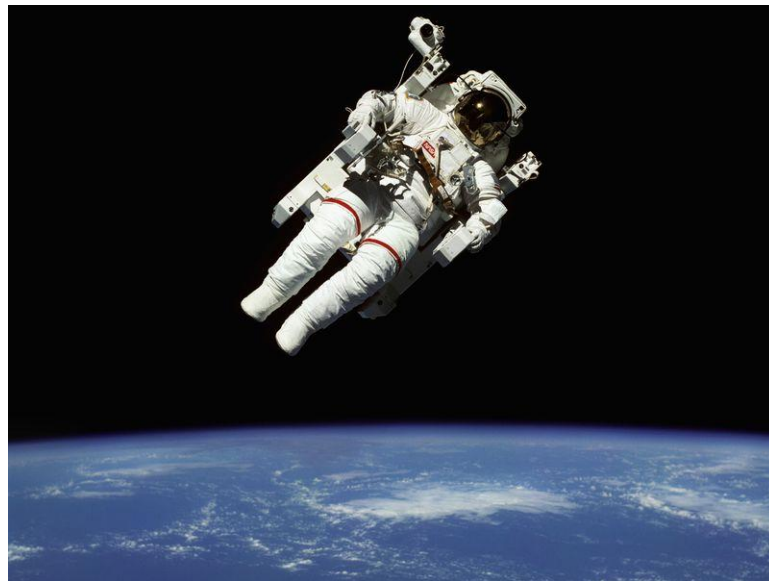


Per fortuna, c’è una soluzione!

Isolation Framework

Sono librerie e package che offrono oggetti “programmabili” in grado di

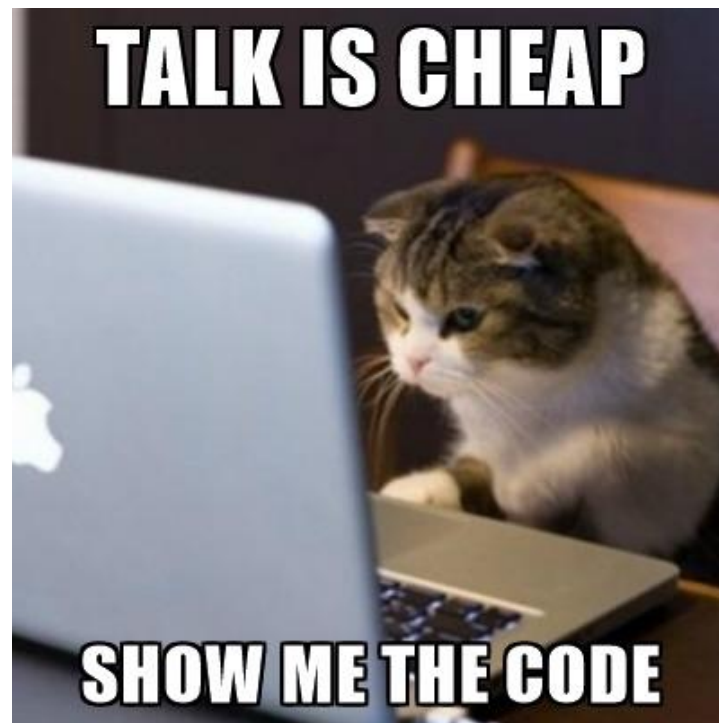
- aiutare nella creazione di *Fake* (ossia *Mock* e *Stub*) in modo semplice
- sollevare lo sviluppatore dall’onere di scriverne il codice
- acquisire informazioni sulle interazioni previste (in caso di *Mock*) e fare asserzioni



Demo

Vediamo assieme

- come funziona **Delphi Mocks**, uno degli Isolation Framework disponibili
- come sostituirlo ai Fake “fatti in casa” nel nostro progetto di test



Conclusioni

Testare bene conviene!

- Forza la scrittura di codice semplice, manutenibile e verificato
- Aumenta la confidenza nel tuo codice e in quello degli altri
- Consente di “giocare d’anticipo” verificando i bug prima che li scopra il cliente
- E’ una tecnica abilitante per metodologie e architettura (ne accenniamo alcune):
 - Continuous Integration
 - Continuous Delivery
 - Interfacciamento a Scrum, Agile e Kanban Board
 - TDD
- Per librerie e framework, è (quasi) indispensabile
 - Gli utenti orientano le scelte in base alla presenza o meno di testing

Raccontate la vostra esperienza!



Tool e librerie

- **DUnitX** (Test Framework)
<https://github.com/VSoftTechnologies/DUnitX>
- **TestInsight** (IDE Test GUI Tool)
<https://bitbucket.org/sglienke/testinsight/wiki/Home>
- **Delphi Mocks** (Isolation Framework)
<https://github.com/VSoftTechnologies/Delphi-Mocks>
- **Spring4D - Mocks** (Isolation Framework)
<http://www.spring4d.org/>



Risorse e approfondimenti

- Libro “**The Art of Unit Testing**” di Roy Osherove
<https://www.artofunittesting.com/>
- Libro “**Unit Testing Succinctly**” di Marc Clifton
<https://www.syncfusion.com/ebooks/unittesting>
- Post “**The Practical Test Pyramid**” di Martin Fowler
<https://martinfowler.com/articles/practical-test-pyramid.html>





Q & A

Domande?



end;
