UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

Ingegneria del Software – (Automated) Unit Testing e jUnit

Prof. Sergio Di Martino

Test Automation

- Manual testing is laborious and time consuming.
- Computer automation has transformed many sectors of our economy. Why not direct the same technology at the laborious aspects of software testing?
- **Test automation** is the process of using software to automate the manual aspects of software testing.
- Automated testing reduces labor required to run tests and speeds up the testing process.
- Developers generally feel more confident making a change when there is a large comprehensive suite of **regression tests** that can be ran against the system under development.
  - A large comprehensive suite of automated tests acts as a safety net when making changes

Automated Unit Testing

- Codice in grado di stressare i metodi pubblici di una classe.
- Il testing è applicato isolatamente ad una unità (**classe**) di un sistema software
- Obiettivo fondamentale è quello di rilevare errori (logica e dati) nel modulo
- Motivazioni:
  - Si riduce la complessità concentrandosi su una sola unit
  - È più facile correggere i bug, poiché poche componenti sono coinvolte

Esempio

Classe da testare:

```
namespace MyCode {
    class Arithmetic {
        public int Add(int i, int j) {
            return i + j;
        }
    }
}
```

Obiettivi Unit Testing

- We want proof that our classes work right
- If we had a bug, we want proof it won’t happen again
- If we change the code, we want proof we did not break anything
- If we do break something, we want to know about it as quickly as possible
- If we do break something, we want to know what exactly is broken: a failing test is the ultimate form of bug report

Continuous Integration

- We want to run tests as often as possible
- Ideally after every compilation
- At least on every check-in
- “Daily builds are for wimps” (Michael Two)
- Tests should be fast: we are going to have hundreds of them
- 1 second test is too long
- Complete automation: absolutely no human interaction allowed

Dependencies

- Assumption: we want to achieve fast feedback and reliable tests
  - Note, there are other kinds of tests that also have merit
- Remove non-determinism: we need to be in control
  - Randomness, time, file system, network, databases, multi-threading
- Remove dependencies to non-deterministic components

Il Problema delle Dipendenze

```
double GetPrice(int productId)
{
   using (SqlConnection conn = new SqlConnection(
            Config.ProductsDbConnectionString))
   {
      conn.Open();
      double price = ReadPriceFromDb(conn, productId);
      if (DateTime.Now.DayOfWeek==DayOfWeek.Wednesday)
      {
         // apply Wednesday discount
         price *= 0.95;
      }
       return price;
   }
}
```

Dipendenze e Factory

- All dependencies are explicitly passed in constructor/method parameters or properties
- If it is not passed, it should not be used
- Be careful of hidden dependencies
  - Static methods (like DateTime.Now)
  - Singletons
  - “new”
- The only place where things are “new’ed” is a factory class or a factory method, which is not unit tested

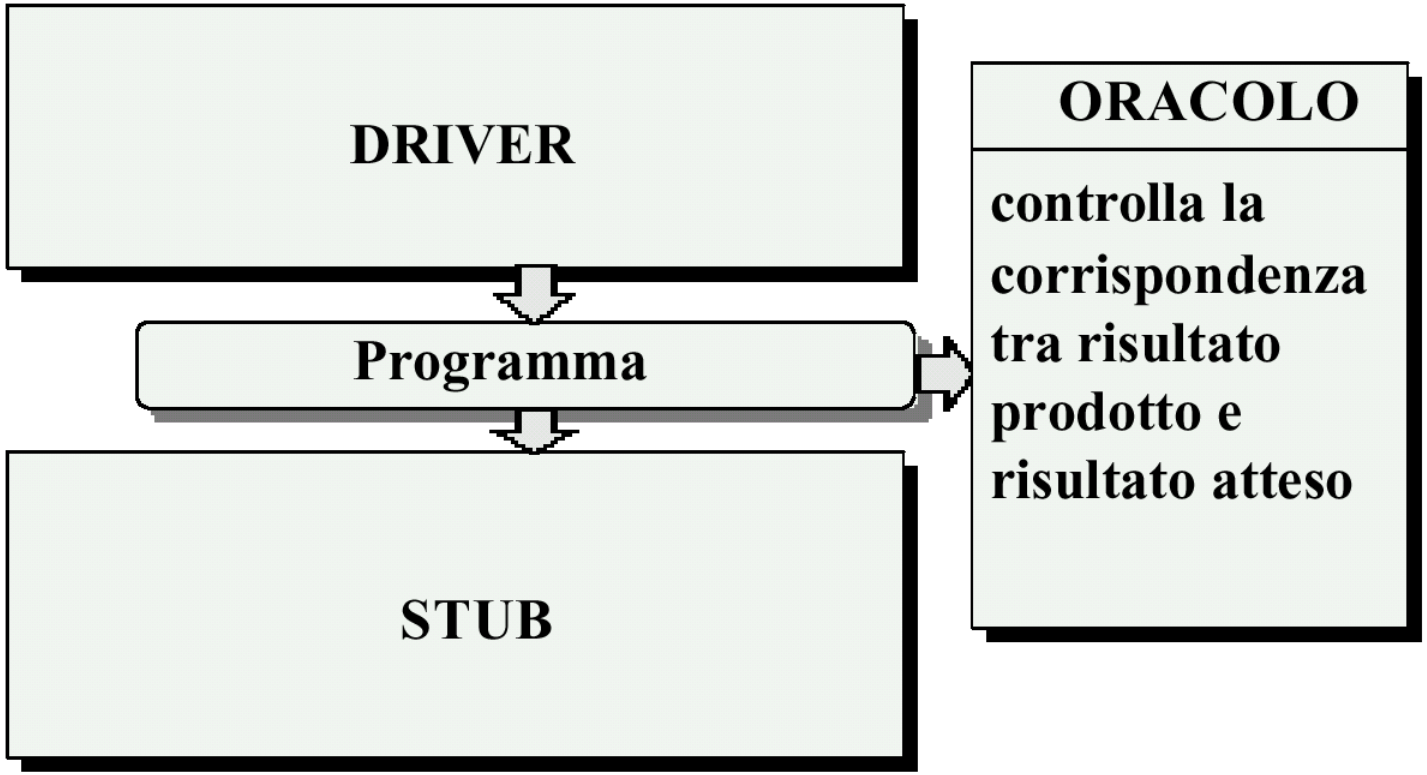
Lo Scaffolding

Come gestire le dipendenze

Test stub e test driver

- Eseguire test case su una classe richiede che questa sia isolata dal resto del sistema
- **Test Driver** e **test Stub/Mock** sono usati per sostituire le parti mancanti del sistema
  - Un test stub è una implementazione parziale di componenti da cui la componente testata dipende (componenti che sono chiamate dalla componente da testare).
  - Un test driver è un blocco di codice che inizializza e chiama la componente da testare.
- **Test driver** + **test stub = Scaffolding**

Creare scaffolding



La necessità di moduli stub e di moduli driver dipende dalla posizione del modulo nell'architettura del sistema ...

Driver e Stub

- Driver (Modulo Guida)
  - Deve simulare l'ambiente chiamante
  - Deve occuparsi dell'inizializzazione dell'ambiente non locale del modulo in esame
  - Il Framework xUnit serve per la creazione di Driver
- Stub (Modulo Fittizio)
  - Ha la stessa interfaccia del modulo simulato (tipicamente ne è una sottoclasse), ma è privo di implementazioni significative
  - Mock Objects

Il problema dello scaffolding

- Creare l'ambiente per l'esecuzione dei test
  - Lo scaffolding è estremamente importante per il test di unità e integrazione
  - Può richiedere un grosso sforzo programmativo
  - Uno scaffolding buono è un passo importante per test di regressione efficiente
  - La generazione di scaffolding può essere parzialmente automatizzato a partire dalle specifiche …
  - Esistono pacchetti software per supportare la generazione di scaffolding
    - JUnit, NUnit, PUnit, etc…

Implementazione di Test stub

- Un test stub deve fornire la stessa API del metodo della componente simulata e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specificato nella signature.
  - Se l'interfaccia di una componente cambia, anche il corrispondente test driver e test stub devono cambiare
- L'implementazione di un test stub non è una cosa semplice.
  - Non è sufficiente scrivere un test stub che semplicemente stampa un messaggio attestante che il test stub è in esecuzione
    - La componente chiamata deve fare un qualche lavoro.
    - Il test stub non può restituire sempre lo stesso valore

Progettare per il Testing

```
class Traveler
{
    Car c=new Car();
    void startJourney()
    {
       c.move();
    }
}

class Car
{
  void move()
  {
     // logic...
  }
}
```

- La classe Traveler è fortemente accoppiata con Car.
- Non è possibile passare un'istanza di Car dall'esterno.
- Diventa impossibile testare Traveler in isolamento senza fare modifiche al codice.
- Con l'introduzione dell'interfaccia Vehicle, si risolve questo problema.
- Posso definire una classe Car_Stub, implementazione di Vehicle, e grazie alla Dependency Injection effettuata dal Test Driver, testare Traveler in isolamento.

Esempio (cont.)

```
class Traveler
{
    Vehicle v;
    public void setV(Vehicle v)
    {
      this.v = v;
    }

    void startJourney()
    {
       v.move();
    }
}

Interface Vehicle
{
   void move();
}
```

```
class Car implements Vehicle
{
    public void move()
    {
         // logic
    }
}
```

```
class Car_Stub implements Vehicle
{
    public void move()
    {
         // Mock logic
    }
}
```