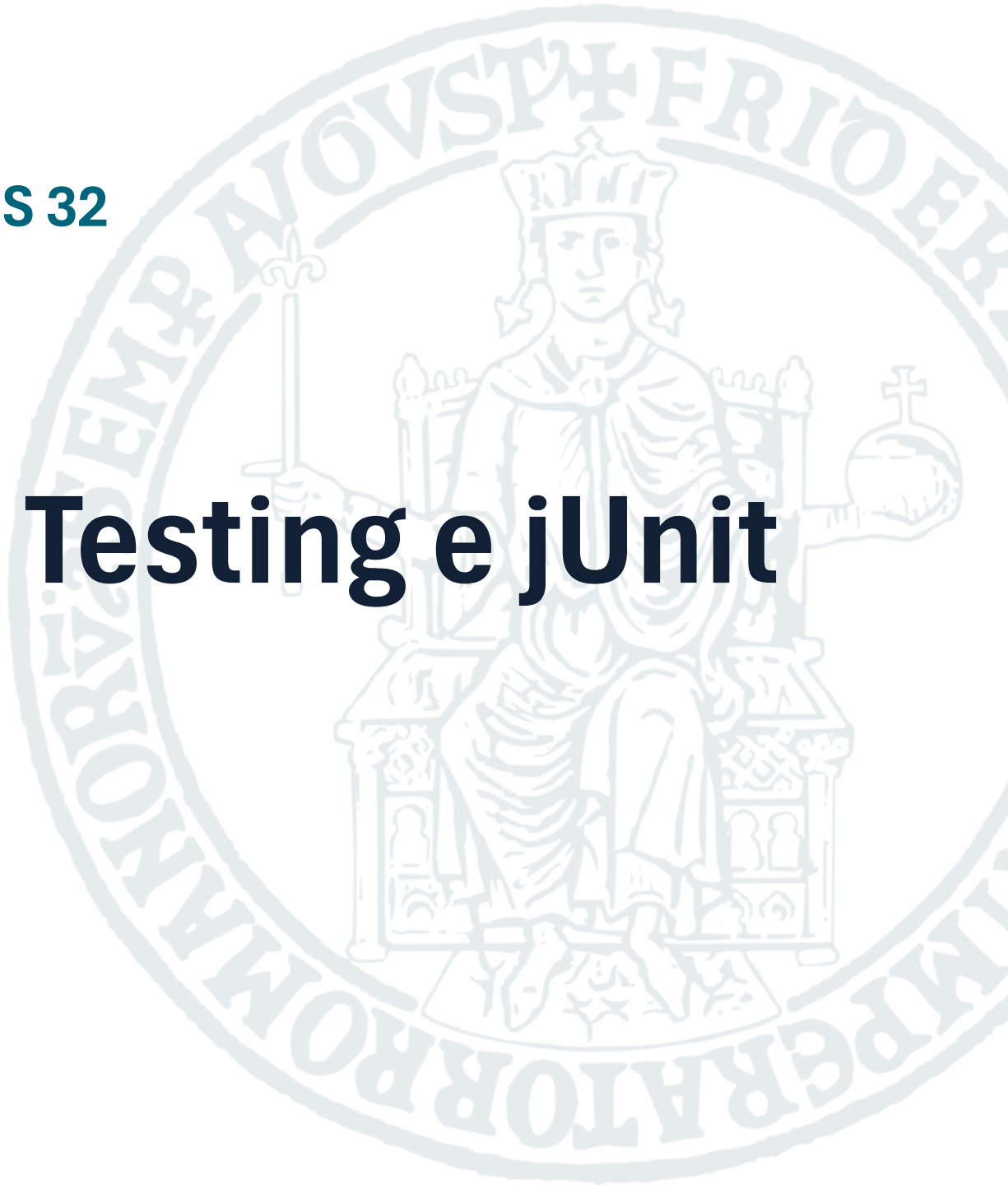
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

SOFTWARE ENGINEERING – LECTURES 32

(Automated) Unit Testing e jUnit

Prof. Sergio Di Martino



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

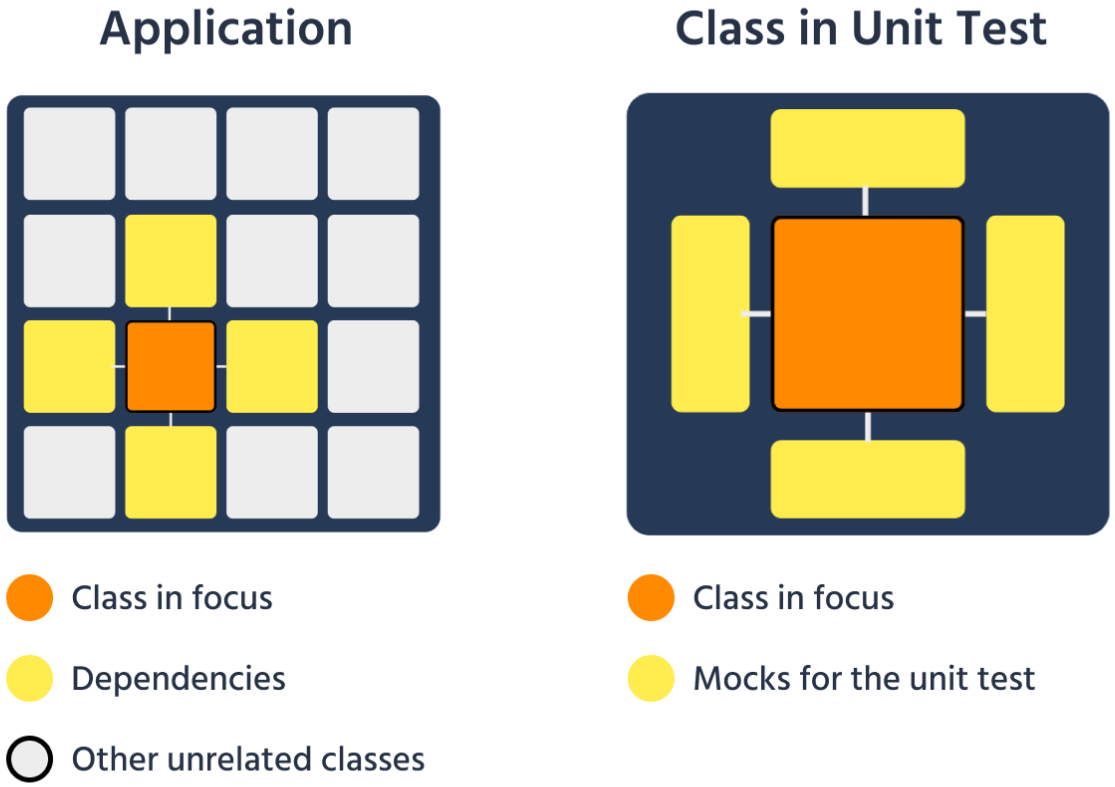
Obiettivi Unit Testing

- 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

Lo Scaffolding

Testing in Isolamento

- Eseguire test case su una Unit (classe) richiede che questa sia isolata dal resto del sistema

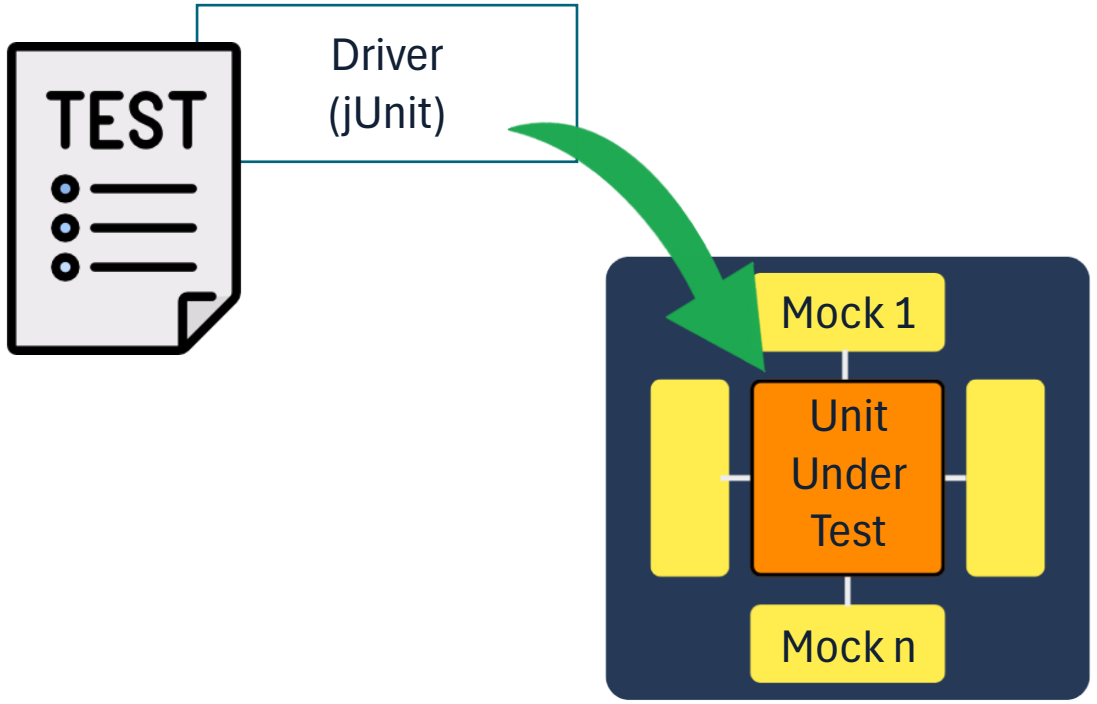


Driver e Stub

- Test Driver e test Stub/Mock sono usati per sostituire le parti mancanti del sistema
- 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 (Moduli Fittizi)
  - Hanno la stessa interfaccia delle componenti simulate, ma è privo di implementazioni significative
  - Mock Objects

Scaffolding

- **Test driver + Test stub/mock = Scaffolding**



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 sempre è sufficiente scrivere un test stub che restituisca lo stesso valore….

Stub vs. Mock

- Stub e Mock sono due modi strategie per risolvere le dipendenze.
- **Stub**
  - Uno stub è un oggetto “finto” che sostituisce una dipendenza, ma con un comportamento statico e predefinito.
  - Scopo: Fornire dati di ritorno semplici per permettere al test di proseguire. Non verifica interazioni, serve solo per simulare risposte.
- **Mock**
  - Un mock è un oggetto più complesso per sostituire una dipendenza, che non solo fornisce risposte, ma verifica le interazioni (quali metodi sono stati chiamati, con quali parametri).
  - Scopo: Testare il comportamento e le collaborazioni tra oggetti.
  - Esistono framework per la generazione automatica (es.: Mockito)

Organizzazione codice con jUnit

Come organizzare jUnit

- NON mettere nelle stesse directory codice di produzione e codice di test!
- Usando strumenti di build automatica, creare una gerarchia per il codice di produzione, ed una gerarchia gemella per il codice di test
- La gerarchia del codice di test va esclusa dalla build finale

Come organizzare jUnit

```
progetto-junit/
│
├── pom.xml                # File di configurazione Maven (o build.gradle per Gradle)
│
├── src/
│   ├── main/
│   │   └── java/          # Codice sorgente dell'applicazione
│   │       └── com/
│   │           └── esempio/
│   │               └── Calcolatrice.java
│   │
│   └── test/
│       └── java/          # Codice dei test JUnit
│           └── com/
│               └── esempio/
│                   └── CalcolatriceTest.java
│
└── target/                # Cartella generata per i file compilati e report
```

Unit Testing e Dipendenze

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

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()
    {
         // Simulated logic in the mock
    }
}
```