Software Testing

Integration Testing

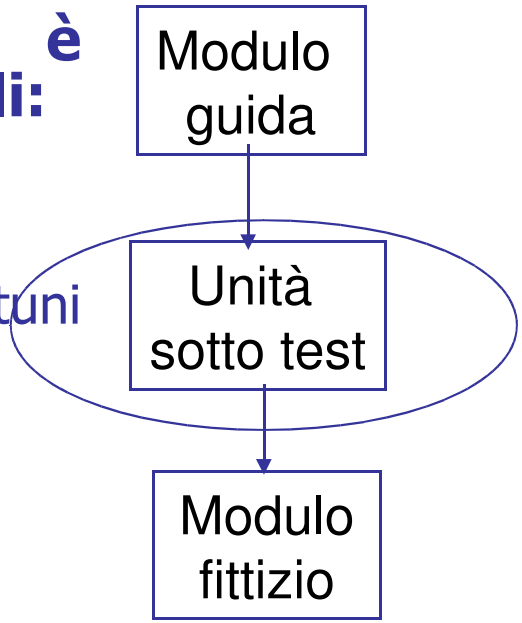
Limiti dello Unit Test

Non si può testare un modulo in isolamento se:

- comunica con il database
- comunica in rete
- ccomunica con altri moduli non ancora testati
- modifica database/file o altre fonti di dati
- non può essere lanciato in parallelo ad altri test
- bisogna effettuare diverse operazioni per lanciare il test
- …

Testing di un modulo "non terminale"

Per testare un modulo non terminale, è necessario costruire due tipologie di moduli:



Moduli guida *(*driver)

- invocano l'unità sotto test, inviandole opportuni valori, relativi al test case

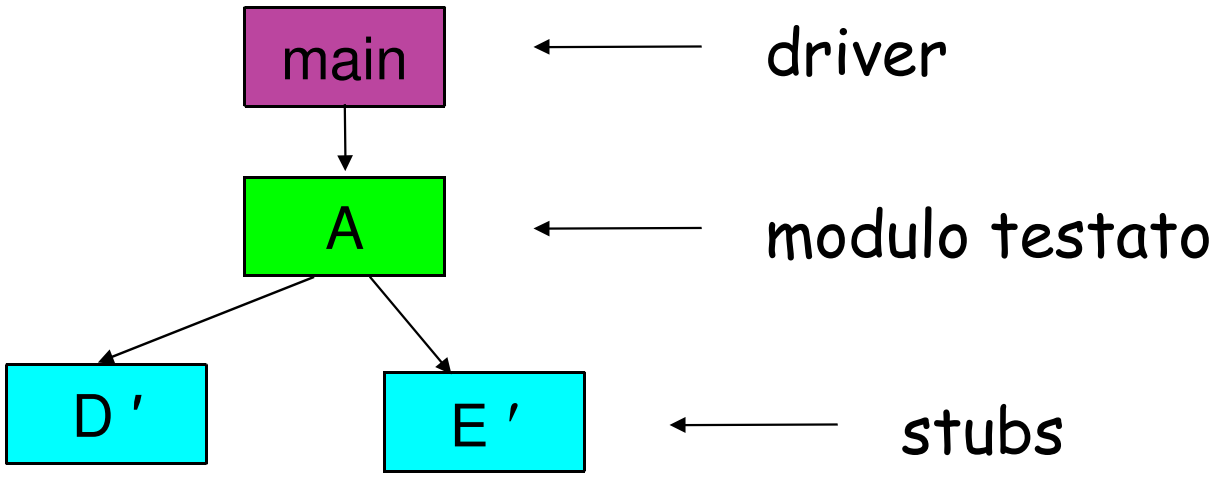
Moduli fittizi *(stub)*

- sono invocati dall'unità sotto test;
- emulano il funzionamento della funzione chiamata rispetto al caso di test richiesto (tenendo conto delle specifiche della funzione chiamata)
  - *Quando la funzione chiamata viene realizzata e testata, si sostituisce lo stub con la funzione stessa*

Stub e Driver

I moduli testati hanno bisogno di essere chiamati (dai Driver)

I moduli chiamati devono essere sostituiti da altri (Stub)



Driver

- **Un modulo driver deve sostituire in tutto e per tutto il/i moduli chiamanti il modulo da testare**
  - Un metodo TestCase sotto Junit può implementare un driver
- **Il modulo driver deve:**
  - Settare tutti i valori delle risorse e fonti dati utilizzate dal modulo da testare
  - In linguaggi object oriented, costruire l'oggetto il cui metodo è sotto test
  - Avviare il metodo da testare

Stub

- **Uno stub è una funzione fittizia la cui correttezza è vera per ipotesi**
  - Esempio, se stiamo testando una funzione *prod_scal(v1,v2)* che richiama una funzione *prodotto(a,b)* ma non abbiamo ancora realizzato tale funzione
  - Nel metodo driver scriviamo il codice per eseguire alcuni casi di test
    - *Ad esempio chiamiamo* prod_scal([2,4],[4,7])
  - Il metodo stub potrà essere scritto così:

```
int prodotto (int a, int b){
    if (a==2 && b==4) return 8;
    if (a==4 && b==7) return 28;
    }
```

  - La correttezza di questo metodo stub è data per ipotesi
  - Ovviamente per poter impostare tale testing, bisognerà avere precise informazioni sul comportamento interno richiesto al modulo da testare

Stub

- **Il termine Stub è utilizzato, più genericamente, per indicare un metodo fittizio, non ancora implementato o la cui implementazione sia, volutamente, incompleta**
  - Spesso gli stub vengono messi nel codice semplicemente come promemoria dei metodi ancora da realizzare oppure per consentire la compilazione del codice prima possibile
  - Lo stub ha il compito di riprodurre il comportamento del modulo che sostituisce unicamente nei casi di test previsti dai driver realizzati
  - Lo stub può essere scritto sulla base di una conoscenza 'black box' del modulo da emulare
    - *Gli stub consentono di testare un modulo prima che i moduli da cui esso dipenda sono stati realizzati*

Dipendenze

- **Come si fa a sapere da quali moduli *dipende* l'esecuzione di un dato modulo da testare?**
  - *Il grafo delle dipendenze (Dependency Graph) è un grafo i cui moduli rappresentano moduli (eventualmente classi, metodi, package) e i cui archi, orientati, rappresentano relazioni di dipendenza tra i moduli (ad esempio causate dall'esistenza di chiamate di metodo, utilizzo di oggetti, utilizzo di attributi)*

Dependency graph evaluation

- **Il grafo delle dipendenze può essere parzialmente (totalmente in alcuni casi particolari) ricavato dall'analisi del codice sorgente dell'applicazione**
- **Parecchie estensioni eclipse sono in grado di valutare il dependency graph**
  - stan4j
  - Eclipse Metrics
  - jDepend
  - eDepend

Eclipse Metrics

- **Eclipse Metrics è una estensione di eclipse che verrà utilizzata anche per calcolare automaticamente metriche relative al codice sorgente**
  - Un tutorial completo sul suo utilizzo è all'indirizzo http://metrics.sourceforge.net/
  - La home page del progetto, open source, è all'indirizzo http://sourceforge.net/projects/metrics/

Eclipse Metrics

- **Nell'ambito del testing di integrazione Eclipse metrics consente di:**
  - Disegnare il grafo delle dipendenze di un progetto a partire dal suo codice sorgente
  - Ottenere un ordine topologico dei package, da quelli indipendenti a quelli con maggiore dipendenza
  - Limite: non fornisce indicazioni (nel dependency graph), a livello di classi e metodi

stan4j

- **Strumento molto più potente per la valutazione di dependency graph e altre metriche**
  - Free solo per utilizzi su sistemi di limitate dimensioni
  - Un tutorial sul suo utilizzo è all'indirizzo: http://stan4j.com/
  - Scaricabile da: http://stan4j.com/general/download.html
- **Disegna il grafo delle dipendenze anche a livello più dettagliato, delle classi e dei metodi**

Testing dell'integrazione di due moduli

- **Una volta testate tutte le unità, è necessario procedere al testing di integrazione per valutare se esse funzionano correttamente nel loro complesso**
  - Se a chiama b, e sia a che b hanno superato i test di unità, non è detto che l'insieme di a e b soddisfi tutti i test previsti di a
    - *Possibili problemi potrebbero essere legati, ad esempio, a combinazioni negli input di b che chi ha testato b non ha previsto ma che possono comparire come possibili valori di chiamata in a*
  - Per testare a e b (con a che chiama b) potrebbe essere sufficiente utilizzare i test progettati per a
    - *In pratica, si sostituisce, nei test progettati per a, il vero modulo b al posto del suo stub (o mock)*

Strategie per il testing di integrazione

- **Il testing di integrazione consiste nel considerare insiemi via via più grandi di moduli, fino ad ottenere l'insieme completo**
- **In che modo decidiamo l'ordine di integrazione?**
  - Due strategie "estreme": top-down e bottom-up

Testing d'integrazione bottom-up

- **Si parte dai moduli che non hanno dipendenze (nodi senza archi uscenti nel grafo delle dipendenze)**
- **Si integra ognuno di questi moduli con uno di quelli che lo chiama e si testa la coppia**
- **Si ridisegna il grafo delle dipendenze avendo sostituito i due moduli integrati con un unico modulo**
- **Si ripete iterativamente il procedimento fino ad ottenere un grafo con un unico nodo**
  - Questo procedimento vale in assenza di cicli nel grafo. I cicli vanno prima ricondotti al caso di coppie di moduli che si chiamano vicendevolmente, per poi integrarli

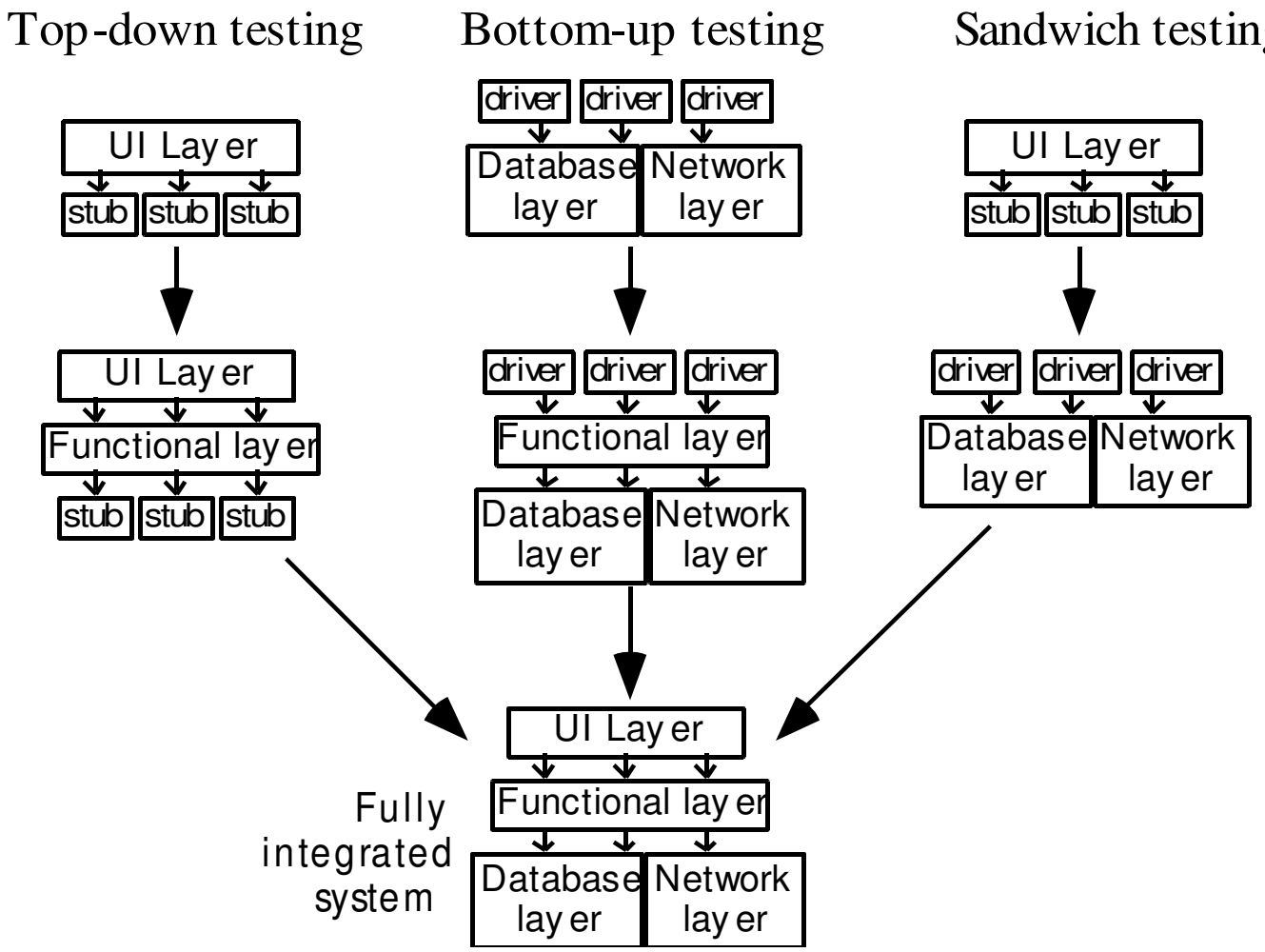
Testing d'integrazione top-down

- **Si parte dai moduli che non dipendono da alcun altro modulo (nodi senza archi entranti nel grafo delle dipendenze)**
- **Si integra ognuno di questi moduli con uno di quelli chiamati e si testa la coppia**
- **Si ridisegna il grafo delle dipendenze avendo sostituito i due moduli integrati con un unico modulo**
- **Si ripete iterativamente il procedimento fino ad ottenere un grafo con un unico nodo**
  - Questo procedimento vale in assenza di cicli nel grafo. I cicli vanno prima ricondotti al caso di coppie di moduli che si chiamano vicendevolmente, per poi integrarli

Strategie d'integrazione

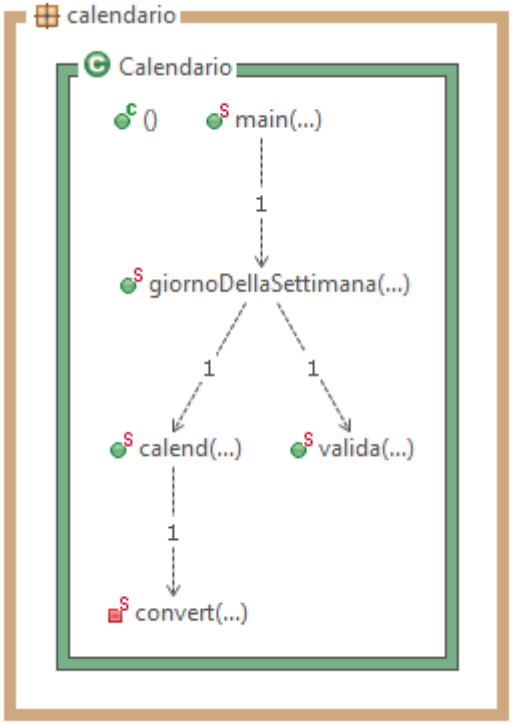
- **E'abbastanza semplice la scelta di una strategia precisa per software progettato a livelli (layer)**
- **Spesso si utilizzano tecniche miste (sandwich) nel quale strategie top-down e bottom-up sono alternate**
- **Altre strategie portano a integrare prima i metodi più fortemente accoppiati, in modo da ridurre il più velocemente possibile la complessità del grafo delle dipendenze**
- **Spesso la strategia d'integrazione dipende anche dalla gerarchia aziendale e dall'organizzazione delle risorse umane**

Strategie per il testing di integrazione



Esempio di testing di integrazione con JUnit

- **Consideriamo un'unica classe Calendario con i seguenti metodi:**
  - main legge da linea di comando giorno, mese (stringa) e anno
  - giornoDellaSettimana converte il mese in input in un intero, controlla se la data è valida e eventualmente chiama calend
  - valida controlla se la data è valida
  - calend calcola il giorno della settimana (numerico)
  - convert converte in stringa il giorno della settimana risultante



Esempio di testing bottom-up 1/2

- **Cominciamo testando convert**

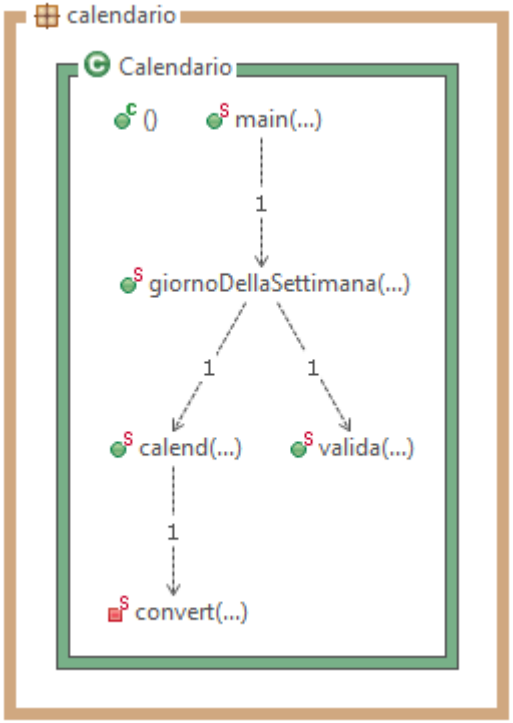
```
@Test
 public void testConvert1() {
 assertEquals("Domenica",Calendario.convert(0));
 }
 …
```

- **Continuiamo con calend**

```
@Test
public void testCalend1() {

assertEquals("Domenica",Calendario.calend(24
, 4, 2011));
}
…
```

- **Poi valida:**

```
 @Test
 public void testValida1() {
 assertTrue(Calendario.valida(24, 4, 2011));
 }
 …
```



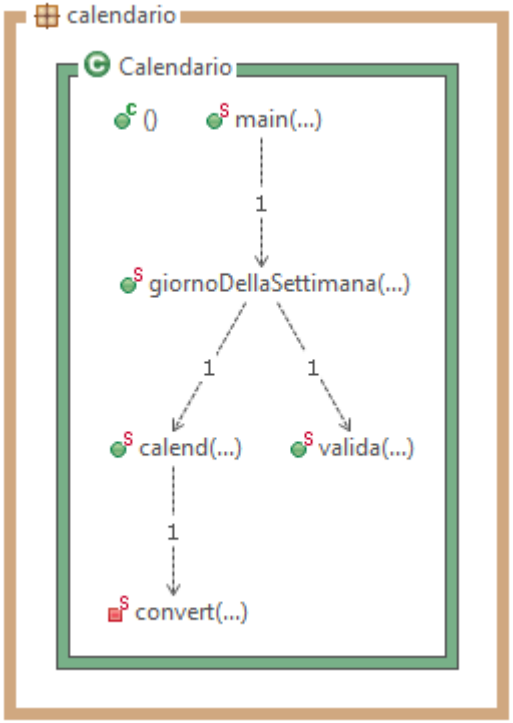
Esempio di testing bottom-up 2/2

- **Poi giornoDellaSettimana:**

```
@Test
public void testGiornoDellaSettimana1() {
assertEquals("Domenica",Calendario.giornoDellaSettimana(24,
   "aprile", 2011));
}
…
```

- **Infine il test del main:**

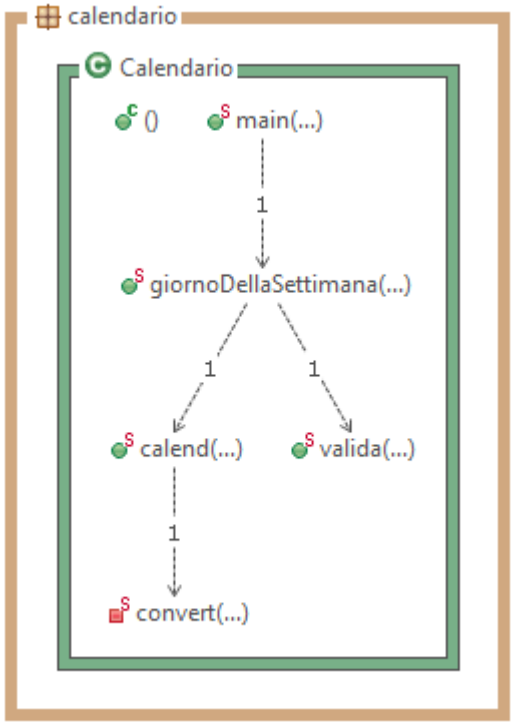
```
@Test
public void testMain1() {
String a[]=new String[3];
a[0]="24";
a[1]="aprile";
a[2]="2011";
assertEquals("Domenica",Calendario.main(a));
}
…
```



Esempio di testing top-down 1/2

- **Per testare la classe main da sola, vediamo che è necessario solo uno stub per giornoDellaSettimana**

```
public static String giornoDellaSettimana(int
   d, String ms, int a){
//STUB
if (d==24 && ms.equals("aprile") && a==2011)
   return "Domenica"; //TC1
else if (d==32 && ms.equals("aprile") &&
   a==2011) return ""; //TC2
return "";
}
```

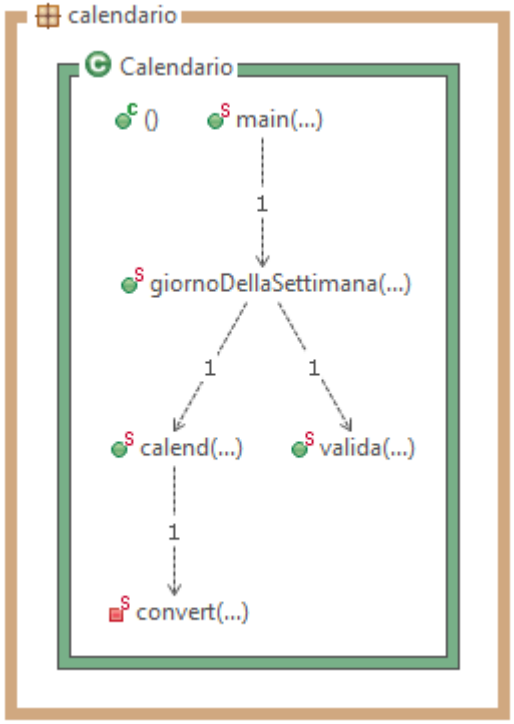


- **Possiamo, poi, eseguire gli stessi test progettati per il main nel bottom-up**

Esempio di testing top-down 2/2

- **Per testare la classe giornoDellaSettimana sono necessari stub sia per calend che per valida**

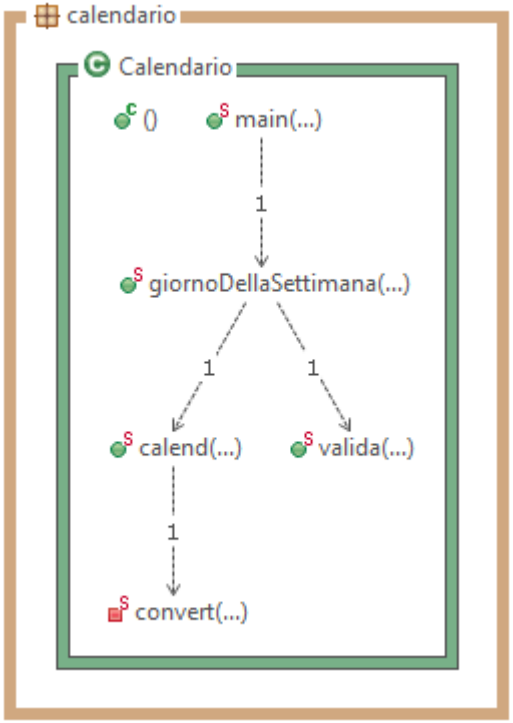
```
public static boolean valida(int d, int m, int
   a) {
//STUB
if (d==24 && m==4 && a==2011) return true;
else if (d==29 && m==2 && a==2012) return true;
else if (d==32 && m==4 && a==2011) return
   false;
return false;
}


public static String calend(int d, int m, int
   a){
// STUB
if (d==24 && m==4 && a==2011) return
   "Domenica";
else if (d==32 && m==4 && a==2011) return
   "Errore";
else return "";
}
```



- **A questo punto riusiamo i test di calend, e così via**

Altri casi di test

- **Se, ad esempio, la classe calend fosse stata la prima ad essere realizzata, sarebbero necessari**
  - Ad esempio per ragioni di prototipazione, poiché è l'unica classe con complessità derivanti dalle conoscenze di astronomia
- **Un driver che si comporti come giornoDellaSettimana**
  - Può essere il metodo di test JUnit che pensammo nelle strategie top-down e bottom-up
- **Uno stub che imiti convert nei casi previsti dal driver**
  - Può essere lo stub che pensammo nel testing bottom-up



Mock-Objects

- **In alternativa agli stub, una più precisa emulazione del comportamento delle classi non ancora implementate è ottenibile con i mock object**

- **Un mock object è una simulazione di un oggetto reale.**
- **Implementa l'interfaccia dell'oggetto da simulare ed ha il suo stesso comportamento.**
- **Possono fornire una risposta pre-impostata.**
- **Possono verificare se l'oggetto che li usa lo fa correttamente.**
- **Utilissimi per testare unità senza legarsi ad oggetti esterni.**

Mock Object, Fake, e Stub

Type of mock	Description
Stubs	Stubs are essentially the simplest possible implementation of a given interface you can think of. For example, stubs' methods typically return hardcoded, meaningless values.
Fakes	Fakes are a degree more sophisticated than stubs in that they can be considered an alternative implementation of the interface. In other words, a fake looks like a duck and walks like a duck even though it isn't a real duck. In contrast, a stub only looks like a duck.
Mocks	Mocks can be considered even more sophisticated in terms of their implementation, because they incorporate assertions for verifying expected collaboration with other objects during a test. Depending on the implementation of a mock, it can be set up either to return hardcoded values or to provide a fake implementation of the logic. Mocks are typically generated dynamically with frameworks and libraries, such as EasyMock, but they can also be implemented by hand.

Tratto da: Test Driven - Practical TDD and Acceptance TDD for Java Developers (Lasse Koskela)- Manning Ed. 2008

Differenze fra Stub e Mock

I Mock-Objects non sono Stub! [Fowler]

- http://martinfowler.com/articles/mocksArentStubs.html

In genere lo stub è molto più semplice di un Mock-Object

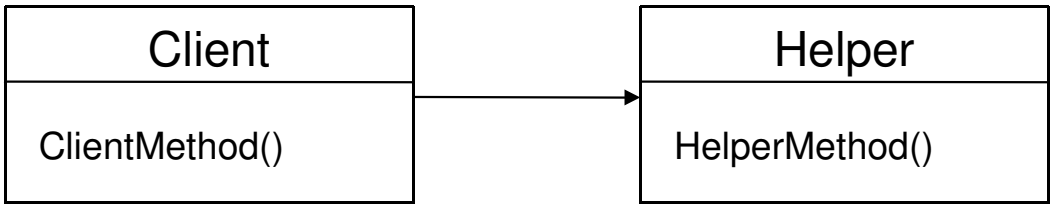
- Gli stub forniscono risposte preconfezionate (con valori prefissati) a chiamate fatte durante il test, senza rispondere di solito a nulla che sia al di fuori di ciò che è previsto per il test.

I mock hanno implementazioni più sofisticate che consentono di verificare il comportamento dell'unità testata (e non solo lo stato)

- verificando ad esempio le collaborazioni avute con altri oggetti ed il relativo ordine di esecuzione
- Possono contenere asserzioni, riguardanti aspetti utili al debugging
- Mock già realizzati saranno fondamentali nei test che coinvolgono classi di libreria

Testing di Classi con dipendenze

Es.: La classe Client (da testare) usa i metodi di Helper

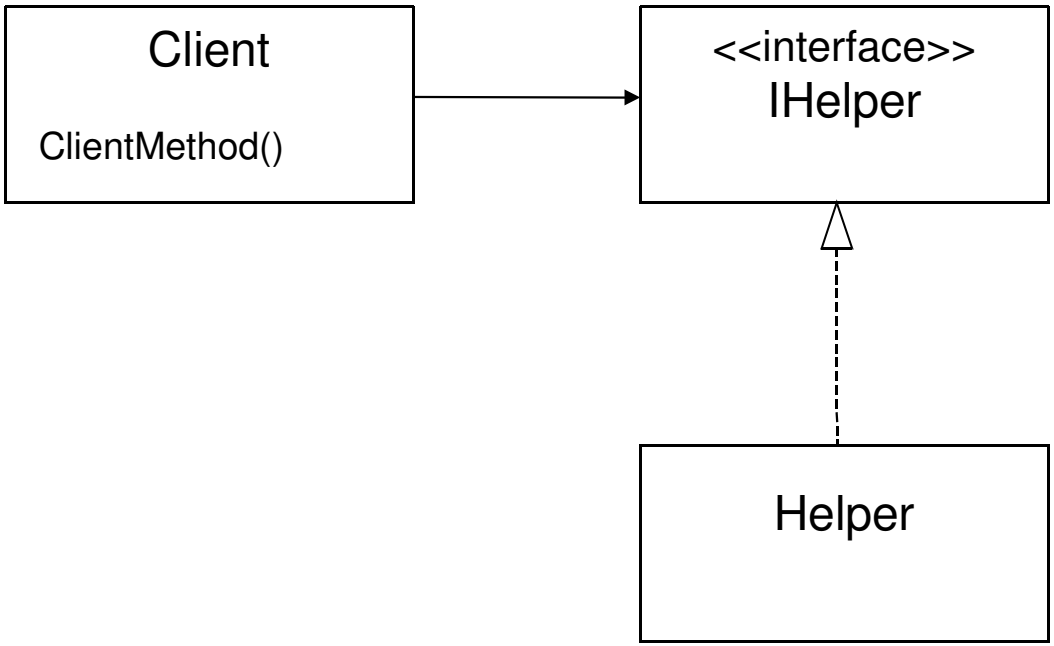


Ma la classe Helper non può essere usata perché:

- Non è ancora disponibile
- Vogliamo controllare direttamente ciò che Helper restituisce a Client

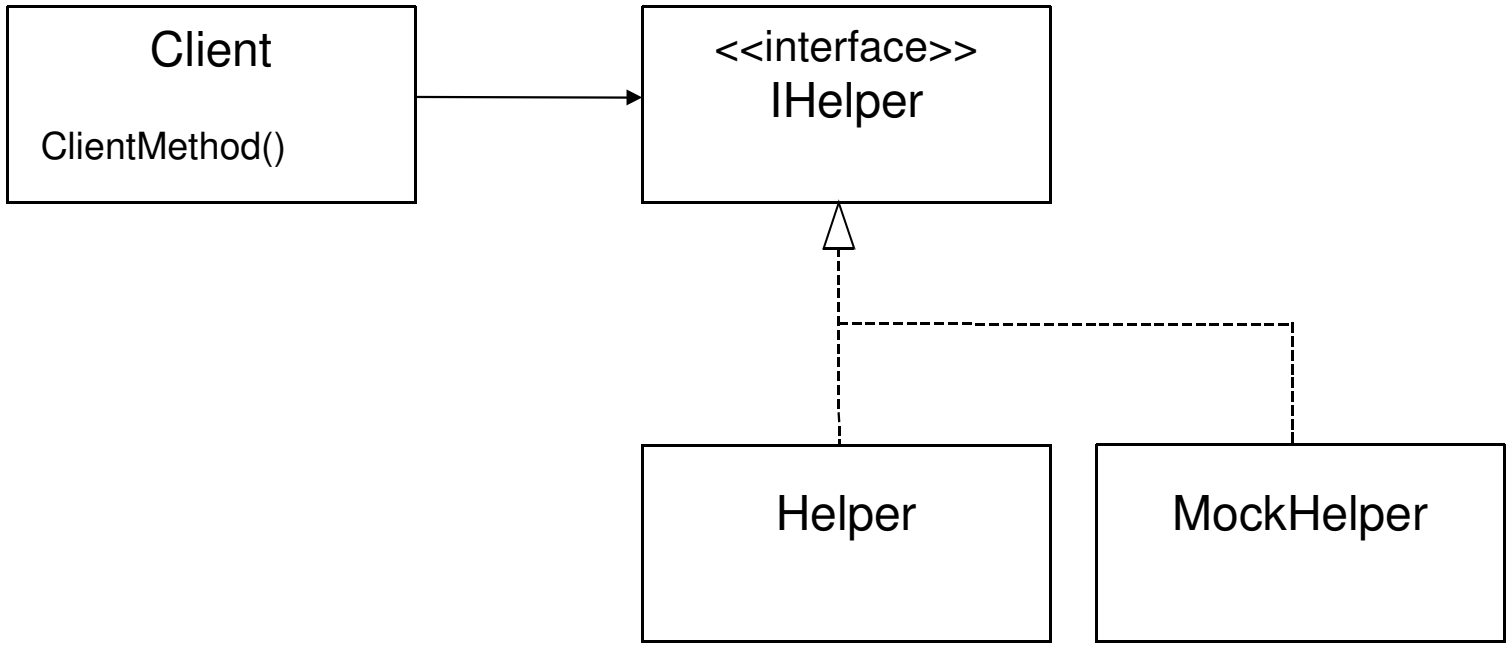
La soluzione usando i Mock-Object

Si estrae l'Interfaccia IHelper...



La soluzione usando i Mock-Object

Il MockObject implementa l'Interfaccia IHelper



Un esempio di semplice Mock

```
public class MockHelper implements IHelper
{
  public MockHelper ( )
  {
  }

  public Object helperMethod( Object aParameter )
  {
    Object result = null;
    if ( ! aParameter.toString().equals("expected" )
    {
      throw new IllegalArgumentException(
        "Unexpected parameter: " + aParameter.toString() );
    }
    result = new String("reply");
    return result;
  }
}
```

Sviluppo di Mock

- **In genere i Mock sono in grado di fare controlli sul comportamento dell'oggetto testato e sulle sue interazioni con altri oggetti.**
- **Lo sviluppo di Mock Objects è supportata da diversi frameworks, o librerie (come JMock o EasyMock in Java).**
  - Spesso sono disponibili librerie di mock già pronti corrispondenti ad oggetti di libreria