UNIT TESTING
TDD
MOCKITO

Una introduzione ai concetti di Unit Testing, Test-driven Development e Mockito

CHE COSA VEDREMO

- Concetti base
- Unit Testing
- TDD
- Mock Objects
- Mockito

DESIGN BY CONTRACT DBC

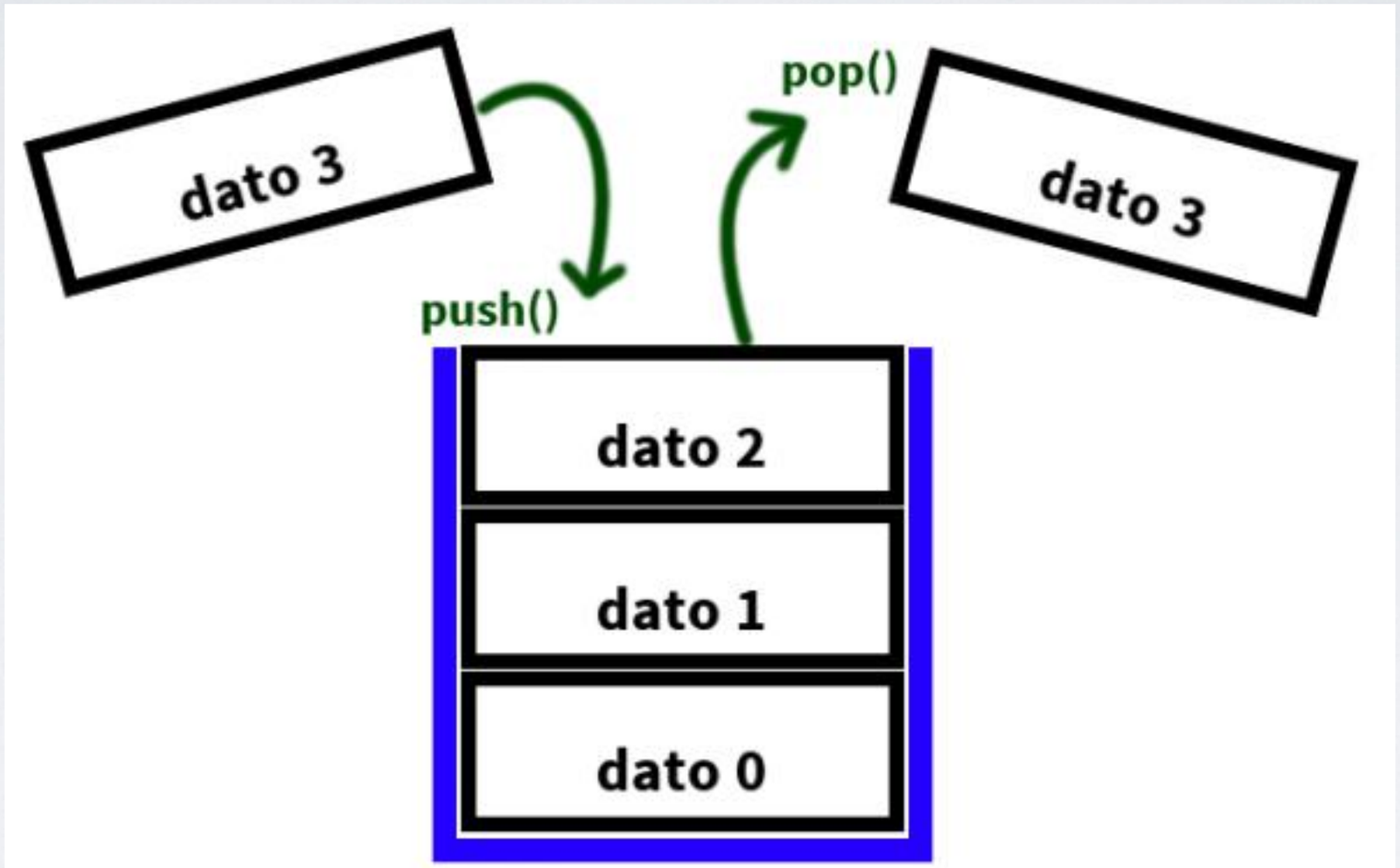
- le entità software hanno degli obblighi nei confronti di altre entità in base a regole formalizzate fra di essi

- Ogni metodo definisce Precondizioni - Postcondizioni

DESIGN BY CONTRACT DBC

- "Se tu, il chiamante, predisponi certe precondizioni, allora io stabilirò certi altri risultati quando ti avrò terminato. Se tu violi le precondizioni, allora non ti prometto niente."

STACK



- PUSH consente di inserire un nuovo elemento in cima alla pila
- POP rimuove l'elemento in cima

PRE-CONDITION

```
// Assert pre-conditions to inform clients when they violate the
contract
public Object top() {
  assert(!this.isEmpty()); // pre-condition
  return top.item;
}
```

CLASS INVARIANT

```
// class invariant - A logical condition that always holds for any
object of a class.
protected boolean invariant() {
 return
  (size >= 0)
  &&
    (
     (size == 0 && this.top == null)
     ||
     (size > 0 && this.top != null)
    );
};
```

POST-CONDITION

```
// Assert post-conditions and invariants to inform yourself when you
violate the contract.
public void push(Object item) {
  top = new Cell(item, top);
  size++;
  assert !this.isEmpty();    // post-condition
  assert this.top() == item; // post-condition
  assert invariant();
}
```

UNIT TESTING

- lo unit testing è una procedura usata per verificare singole parti di un codice sorgente

- Per UNIT (unità) si intende genericamente la minima parte testabile di un codice sorgente

- Nella programmazione orientata agli oggetti, la più piccola unità può essere il metodo.

UNIT TESTING

- Lo scopo dell'Unit testing è quello di verificare il corretto funzionamento di parti di programma permettendo così una precoce individuazione dei bug

EDSGER DIJKSTRA

Come ogni forma di testing, anche lo Unit Testing non può individuare l'assenza di errori ma può solo evidenziarne la presenza.

UNIT TESTING

Benefici:

- Semplifica le modifiche
  - verifica le postcondizioni dopo un refactoring
- Semplifica l'integrazione
  - limita i malfunzionamenti dovuti a problemi nella interazione tra i moduli
- Supporta la documentazione
  - una documentazione "viva" del codice, è un esempio di utilizzo dell'API del modulo

UNIT TESTING

- Lo Unit Test non dovrebbe mai varcare i confini della classe
- Il programmatore impara ad isolare la classe utilizzando Mock Objects

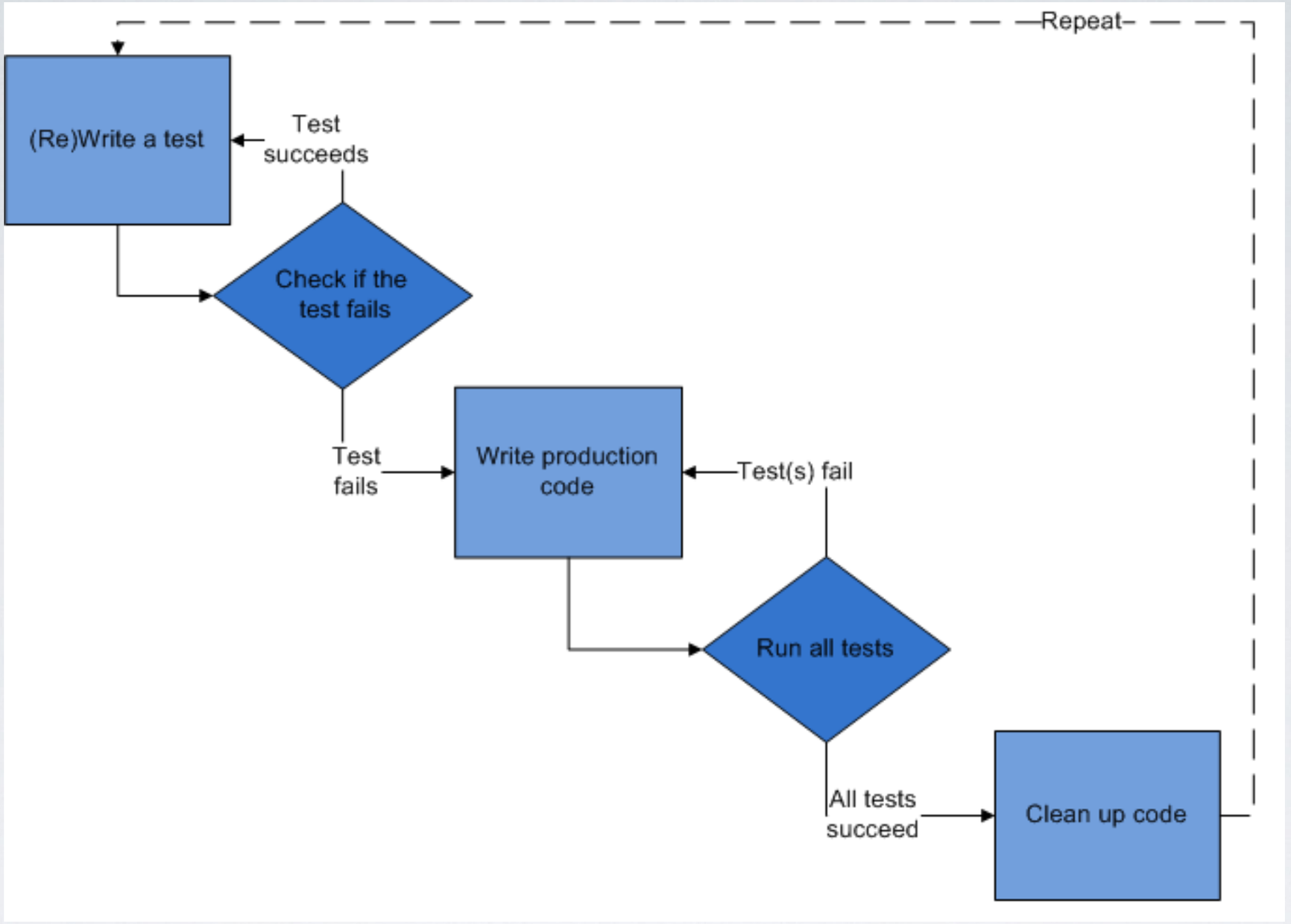
UNIT TESTING - TDD

- Lo Unit Testing è la parte fondamentale dell'Extreme Programming

- Test-driven Development - i test unitari vengono scritti prima di scrivere il codice. Test funzionali e unitari.

- Eseguiamo il nostro codice mentre lo scriviamo

  - permette di valutare l'esito del nostro codice mentre lo scriviamo



SVILUPPO GUIDATO DA TEST

MARTIN FLOWLER

Whenever you are tempted to type into a print statement or a debugger expression, **write it as test instead**

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

JUNIT

@Test

@Before

@After

@BeforeClass

@AfterClass

@Ignore

@Test (expect = ? extends Throwable)

JUNIT - MARKING YOUR TEST

```
import org.junit.*;
import static org.junit.Assert.*;

public class CalculatorTest {

 private Calculator calculator;
 @Before
 public void setUp() {
     calculator = new Calculator();
 }
 @After
 public void tearDown() {
     // Clean up...
 }
 @Test
 public void testAdd() {
     assertEquals(2, calculator.add(1, 1));
     // More...
 }
 @Test
 public void testSubtract() throws Exception{
     // More...
 }
}
```

JUNIT - CHECKING RESULT

```
org.junit.Assert
assertEquals(...), assertSame(…)
assertNull(...), assertArrayEquals(…)
assertNotSame(…), assertNotNull(...)
assertTrue(…), assertFalse(…)
fail(...)
```

JUNIT - EXAMPLES

```
// Test if exception is thrown:
@Test
public void testSquareRoot_negativeArgument() {
    try {
        int result = calculator.squareRoot(-1);
        fail(“Should throw IllegalArgumentException”);
    }
    catch(IllegalArgumentException e) {
    // Additional assertions...
    }
}
// Sometimes this will do:
@Test(expected = IllegalArgumentException.class)
public void testSquareRoot_negativeArgument() {
    int result = calculator.squareRoot(-1);
}
// Test is skipped:
@Ignore(“Not implemented yet”)
@Test
public void testSquareRoot() {
     assertEquals(1, calculator.squareRoot(1);
     assertEquals(2, calculator.squareRoot(4);
}
```

GOOD AND BAD PRACTICES

PRIMA SCRIVI I TEST

- Induce a pensare prima alla funzionalità
- I test ti diranno quando avrai completato il lavoro
- Non dimenticherai di fare i test
- È più divertente

NON COMMENTARE I TEST

- Usa @Ignore, e indica una buona motivazione
- Non dimenticare di correggere i Test

ATTENZIONE AL NOME DEL TEST

- Quando un test fallisce il nome del test ti dovrebbe dire che cosa è fallito
  - No: `testReturnsNull();`
  - Ma: `testReadFile_FileNotFound();`
- Usa delle convenzioni
  il metodo `add()` della classe `Calculator` è testato dal metodo `testAdd()` della classe `CalculatorTest();`

FAI ATTENZIONE AI DATI STATICI

- I test non vengono eseguiti in un ordine specifico
- I test non devono dipendere dal risultato di un altro test

SCRIVI DEL CODICE TESTABILE

- Metodi `private` non possono essere testati
- Se hai bisogno di testarli usa `package`

NO! CATCH() DI ECCEZIONI INASPETTATE

- Non usare `catch()` nei test, a meno che l'eccezione non sia attesa

- I test devono rilanciare l'eccezione, questo farà fallire il test, e rivecerai l'intero stacktrace

ESERCITAZIONE

- Workshop Test Driven Development JUNIT

MOCK OBJECT

MOCK OBJECT

- Nella programmazione simulano il comportamento di oggetti reali in modo controllato e verificabile

- I programmatori usano mock object così come i costruttori di automobili usano i manichini nei crash test

MOCK OBJECT

"The art of unit Testing" Osherove, Roy (2009).

- **Mock Object** sono falsi oggetti che aiutano a decidere se un test è fallito o no, verificando se è avvenuta o meno una interazione con un oggetto

- Tutto il resto è da considerarsi **Stub**

MOCKITO

A colpo d'occhio

- mock(ClassToMock.class)
- when(methodCall)
  - thenReturn(value)
  - thenThrow(Throwable)
- verify(mock).method(args)

- @Mock
- initMocks(this)
- assertThat(obj, matcher)
- Eclipse IDE tips

PERCHÈ MOCKITO

Usa **Mockito** per ottenere"**intelligenti**" e **false implementazioni** di classi ed interfacce che sono fuori dalla tua portata

CREARE MOCK

```
List mockedList = Mockito.mock(List.class)
```

CREARE MOCK

```
@Mock
private List mockedList;

@Before
public void setup() {
    MockitoAnnotations.initMocks(this);
}
```

```
@RunWith(MockitoJUnitRunner.class)
public class ExampleTest {

      @Mock
      private List list;

      @Test
      public void shouldDoSomething() {
          list.add(100);
      }
 }
```

JUnit >= 4.4

VERIFICARE I COMPORTAMENTI

```
//Let's import Mockito statically so that the code looks clearer
import static org.mockito.Mockito.*;

//mock creation
List mockedList = mock(List.class);

//using mock object
mockedList.add("one");
mockedList.clear();

//verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

WHEN(...).THEN*(...)

- `when(methodIsCalled).thenReturn(aValue);`

- `when(methodIsCalled).thenThrow(anException);`

STUBBING

```
//You can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

//stubbing
when(mockedList.get(0)).thenReturn("first");
when(mockedList.get(1)).thenThrow(new RuntimeException());

//following prints "first"
System.out.println(mockedList.get(0));

//following throws runtime exception
System.out.println(mockedList.get(1));

//following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));

//Although it is possible to verify a stubbed invocation, usually it's just
redundant
//If your code cares what get(0) returns then something else breaks (often
before even verify() gets executed).
//If your code doesn't care what get(0) returns then it should not be
stubbed.
verify(mockedList).get(0);
```

WHEN(...)

```
when(methodIsCalled)

        aMockObject.method(arguments...)
```

```
mocked.get(”12”)

mocked.get(anyString())

mocked.get(eq(”12”))
```

thenReturn(value)

```
Properties properties = mock(Properties.class);

when(properties.get(”shoeSize”)).thenReturn(”42”));

String value = properties.get(”shoeSize”);

assertEquals(”42”, value);
```

thenThrow(Exception)

```
Properties properties = mock(Properties.class);
when(properties.get(”shoooSize”)
.thenThrow(new IllegalArgumentException(...));
```

```
try {
   properties.get(”shoooSize”);
   fail(”shoooSize is misspelled”);
} catch (IllegalArgumentException e) {
   // good! :)
}
```

thenReturn(...)

```
Properties properties = mock(Properties.class);
when(properties.get(”shoeSize”)).thenReturn(”42”, ”43”, ”44”));
```

```
assertEquals(”42”, properties.get(”shoeSize”));
assertEquals(”43”, properties.get(”shoeSize”));
assertEquals(”44”, properties.get(”shoeSize”));
assertEquals(”44”, properties.get(”shoeSize”));
assertEquals(”44”, properties.get(”shoeSize”));
assertEquals(”44”, properties.get(”shoeSize”));
```

Partial Mock

- Basato su una istanza reale di una classe, solamente una parte dei metodi vengono simulati
- Casi d'uso
  - test su interazioni interne di una classe
  - chiamate a metodi su cui è difficile creare mock (static, final, super, constructor)
  - Situazioni difficili da ricreare (excepetions ...)
- Spesso necessario su Legacy Code

Mockito - Spies

- `spy(myInstance)`

registra un comportamento su un oggetto spiato:

```
doReturn(…).when(spyInstance).method(…)
doThrow(…).when(spyInstance).method(…)
doNothing().when(spyInstance).method(…)
```

se non si registra un comportamento, viene chiamata l'implementazione reale

MOCKITO - SPIES

```
@Test
public void test_spyWithStubs() {
  List list = spy(new ArrayList());
  doReturn(100).when(list).size();
  list.add("one");
  list.add("two");
  verify(list).add("one");
  verify(list).add("two");
  assertEquals("one", list.get(0));
  assertEquals("two", list.get(1));
  assertEquals(100, list.size());
}
```

Workshop Test Driven Development

Mockito