Esercitazione 4

Testing e TDD

Testing

- Testing shows the presence, not the absence of bugs
- Dijkstra (1969) J.N. Buxton and B. Randell, eds, Software Engineering Techniques, April 1970, p. 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

Testing

- Testing alone can only give relative confidence that your program works...
- ...it's better than nothing, but there are better ways to ensure correctness.

Esercizio 1

```
/* come testare questo metodo? */
public double getWeightedGradeAverage() {
   double sumOfWeightedPoints = 0.0;
   for (Grade grade: this.grades)
      sumOfWeightedPoints +=
         grade.getCredits() * grade.getMark();
   return sumOfWeightedPoints / this.totalCredits();
}
```

Esercizio 1

- Se guardiamo l'implementazione, un possibile bug è quasi evidente.
- Tuttavia, ignoriamo l'implementazione e pensiamo al testing "black-box" del metodo
- `public double getWeightedGradeAverage() ...`
- Dobbiamo pensare soprattutto ai possibili stati della classe, e testare il comportamento del metodo in maniera opportuna

Passo 0

- Costruire la classe di test:
  - Lo stato interno della classe contiene un riferimento ad un oggetto di tipo Student
  - in @Before, creiamo un nuovo oggetto di tipo Student
  - non c'è necessità di usare @After

Struttura

```
public class StudentTest {

   private Student student;

   @Before
   public void setup() {
      this.student =
         new Student("Studente", "Test", new Date());
   }

   /* casi di test */
   ...
}
```

Casi Possibili

- a) Lo studente ha sostenuto un esame, la media è pari al voto di quell'esame
- b) Lo studente ha sostenuto due esami dello stesso numero di crediti, la media pesata è pari alla media aritmetica dei voti dei singoli esami
- ...

Casi a e b

- Due casi, due possibili test:
  - I casi sono generali, fissiamo degli scenari corrispondenti
  - Ad esempio, il primo esame è Analisi 1, di 10 crediti, e il voto è stato 27

Test 1 (Scenario a)

```
@Test
public void testAverageWithOneGrade() {
   /* creo oggetti da usare per il
    * caso di test */
   Grade ingSwGrade = new Grade("Analisi 1", 27, 10);
   /* porto la classe Student nello
    * stato desiderato */
   this.student.addGrade(ingSwGrade);
   /* asserzione */
   assertEquals(27d /* expected value */,
     this.student.getWeightedGradeAverage(), 0.01d);
}
```

Test 2 (Scenario b)

```
@Test
public void testAverageWithTwoGrades() {
   Grade ingSwGrade = new Grade("Analisi 2", 24, 10);
   Grade apiGrade = new Grade("API", 27, 10);
   this.student.addGrade(ingSwGrade);
   this.student.addGrade(apiGrade);
   assertEquals((24d + 27d) / 2d,
        this.student.getWeightedGradeAverage(),
        0.01d);
}
```

Altro Caso Possibile

- c), generalizzazione del caso b):
  - Lo studente ha sostenuto due esami di diverso numero di crediti, calcolo in maniera esplicita la media pesata

Test 3 (Scenario c)

```
@Test
public void testAverageWithTwoGrades2() {
   Grade ingSwGrade = new Grade("Ing Sw", 24, 7);
   Grade apiGrade = new Grade("API", 27, 10);
   this.student.addGrade(ingSwGrade);
   this.student.addGrade(apiGrade);
   assertEquals((24d * 7d + 27d * 10d) / (17d),
        this.student.getWeightedGradeAverage(),
        0.01d);
}
```

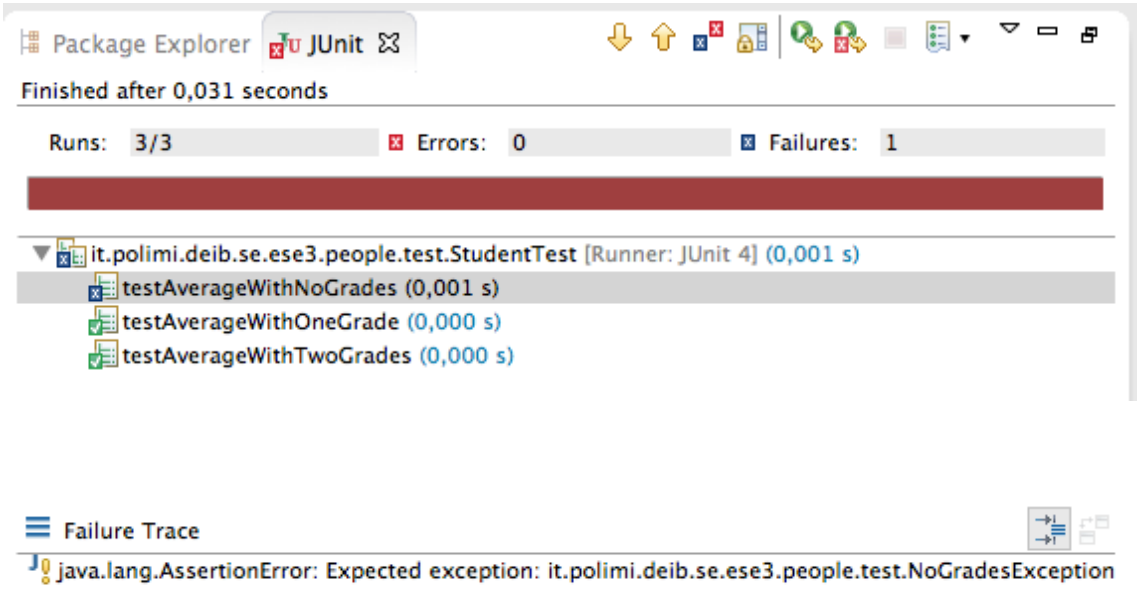
Caso Particolare

- E se lo studente non ha sostenuto esami?
- Pensiamo a cosa dovrebbe fare il metodo...

Test 4

```
/* inizialmente lo studente non ha
 * nessun esame sostenuto: desidero
 * un comportamento eccezionale */
@Test(expected=NoGradesException.class)
   public void testAverageWithNoGrades() {
      double avg = this.student.getWeightedGradeAverage();
   }
```

Test Fallito



Expected Exception, ma il metodo non ha lanciato eccezioni.

Implementazione

```
public double getWeightedGradeAverage() {
   double sumOfWeightedPoints = 0.0;
   for (Grade grade: this.grades)
      sumOfWeightedPoints +=
          grade.getCredits() * grade.getMark();
   return sumOfWeightedPoints / this.totalCredits();
}
```

se totalCredits() == 0.0d, il risultato
è NaN (valore double valido)

Correzione (I)

```
public double getWeightedGradeAverage() {
   if (this.grades.size() == 0)
      throw new NoGradesException();

   double sumOfWeightedPoints = 0.0;
   for (Grade grade: this.grades)
      sumOfWeightedPoints += grade.getCredits() *
         grade.getMark();
   return sumOfWeightedPoints / this.totalCredits();
}
```

Correzione (2)

```
/* Lievemente meglio: uso il valore di ritorno
   di un osservatore invece di un check sullo
   stato interno della classe */
  public double getWeightedGradeAverage() {
    if (this.totalCredits() == 0)
      throw new NoGradesException();

    double sumOfWeightedPoints = 0.0;
    for (Grade grade: this.grades)
      sumOfWeightedPoints += grade.getCredits() *
         grade.getMark();
    return sumOfWeightedPoints / this.totalCredits();
  }
```

Esercizio 2

- Testing dei Costruttori
  - Consideriamo la classe ComplexSet

Setup

```
public class ComplexSetTest {

   private ComplexSet set;

   // Devo testare i costruttori,
   // quindi non ho bisogno di nessun
   // particolare setup
   @Before
   public void setUp() {

   }
```

Costruttori

```
// costruttore di default, caso particolare
// del costruttore che ha come argomento la
// precisione.
public ComplexSet(int maxSize)
```

```
// costruttore che prende come argomento
// una precisione per il controllo dell'uguaglianza
public ComplexSet(int maxSize, double precision)
```

Cosa devo testare?

- Se l'obiettivo è fare testing del comportamento del costruttore e basta, è sufficiente verificare che alcuni semplici proprietà siano soddisfatte dopo la sua invocazione. Per esempio...

Test 1

```
// controllo che il valore di maxSize
// sia corretto dopo l'invocazione del
// costruttore con un parametro.
@Test
public void testFirstConstructor() {
   this.set = new ComplexSet(10);
   assertEquals(10,this.set.getMaxSize());
}
```

Cosa devo testare?

- Non è necessario (se l'obiettivo è il test del costruttore) verificare che e*ffettivamente* dopo l'inserimento di dieci elementi diversi il set sia pieno.
  - Quella funzionalità infatti non è del costruttore!
  - Partiamo come sempre dai parametri...

Dimensione Massima

- Ha senso avere un set con dimensione massima nulla?
  - Caso degenere, ma del tutto lecito dal punto di vista matematico

```
@Test
public void testFirstConstructorZeroSize() {
   this.set = new ComplexSet(0);
   assertEquals(0,this.set.getMaxSize());
}
```

Dimensione Massima

- Ben diverso è il caso di dimensione massima negativa!

```
@Test(expected=IllegalArgumentException.class)
public void testFirstConstructorWithNegMaxSize() {
   this.set = new ComplexSet(-10);
}
```

Correzione

- Dove va la correzione?

```
// ...
public ComplexSet(int maxSize) {
   this(maxSize, 2.0d * Double.MIN_VALUE);
}

// ...
public ComplexSet(int maxSize, double precision) {
   this.size = 0;
   this.maxSize = maxSize;
   this.elements = new Complex[maxSize];
   this.precision = precision;
}
```

Correzione

- Anche se stavamo testando il costruttore con un solo parametro, nell'implementazione l'uno dipende dall'altro
  - La correzione più generale è ovviamente quella in cui il check è fatto sul costruttore con due parametri, ossia quello più generale

Correzione

```
// costruttore che prende come argomento
// una precisione per il controllo dell'uguaglianza
public ComplexSet(int maxSize, double precision) {
   if (maxSize < 0)
     throw new IllegalArgumentException("...");
   this.size = 0;
   this.maxSize = maxSize;
   this.elements = new Complex[maxSize];
   this.precision = precision;
}
```

Testing Costruttori

- Testiamo ora il comportamento dell'altro costruttore.
- Il secondo parametro è un double:
  - Concettualmente, la precisione non può essere un valore negativo!

```
@Test(expected=IllegalArgumentException.class)
public void testSecondConstructorWithNegativePrecision() {
   this.set = new ComplexSet(10,-2.0d);
}
```

Correzione

```
// costruttore che prende come argomento
// una precisione per il controllo dell'uguaglianza
public ComplexSet(int maxSize, double precision) {
   if (maxSize < 0)
      throw new IllegalArgumentException("...");
   if (precision < 0)
      throw new IllegalArgumentException("...");
```

Attenzione a double!

- E’ un tipo molto particolare:
  - Altri possibili valori di double non sono validi: NaN, +inf, -inf... quelli che non rappresentano numeri reali

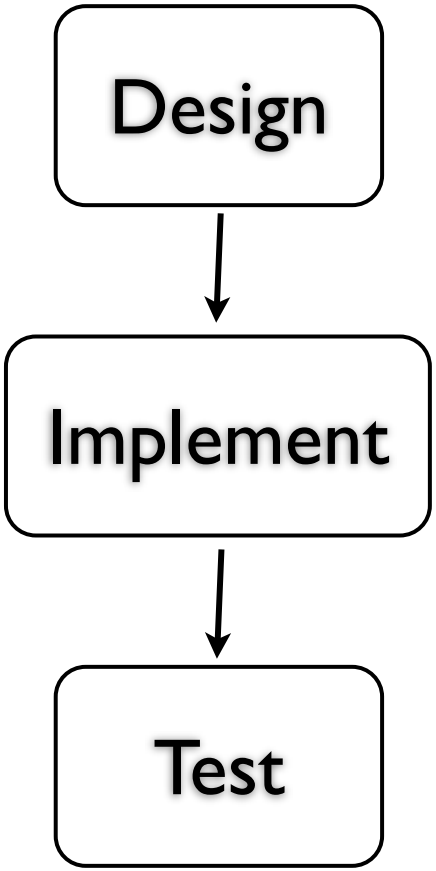
Correzione

```
// costruttore che prende come argomento
// una precisione per il controllo dell'uguaglianza
public ComplexSet(int maxSize, double precision) {
   if (maxSize < 0)
     throw new IllegalArgumentException("...");
   if (precision < 0)
     throw new IllegalArgumentException("...");
   if (Double.isInfinite(precision) ||
       Double.isNaN(precision))
     throw new IllegalArgumentException("...");
```

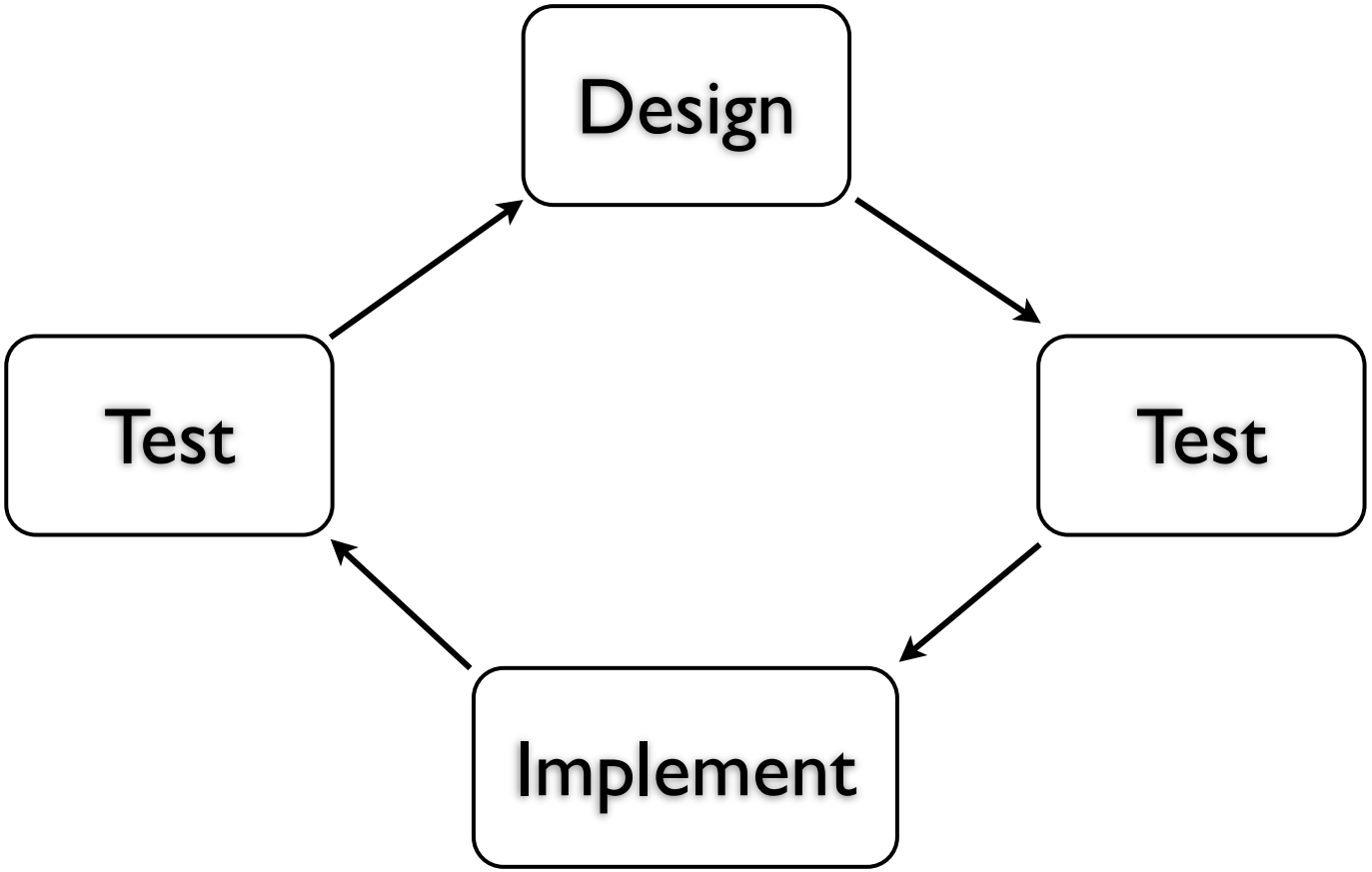
- Metodi di classe della Classe Double che controllano che un double sia infinito o NaN

Test-Driven Development

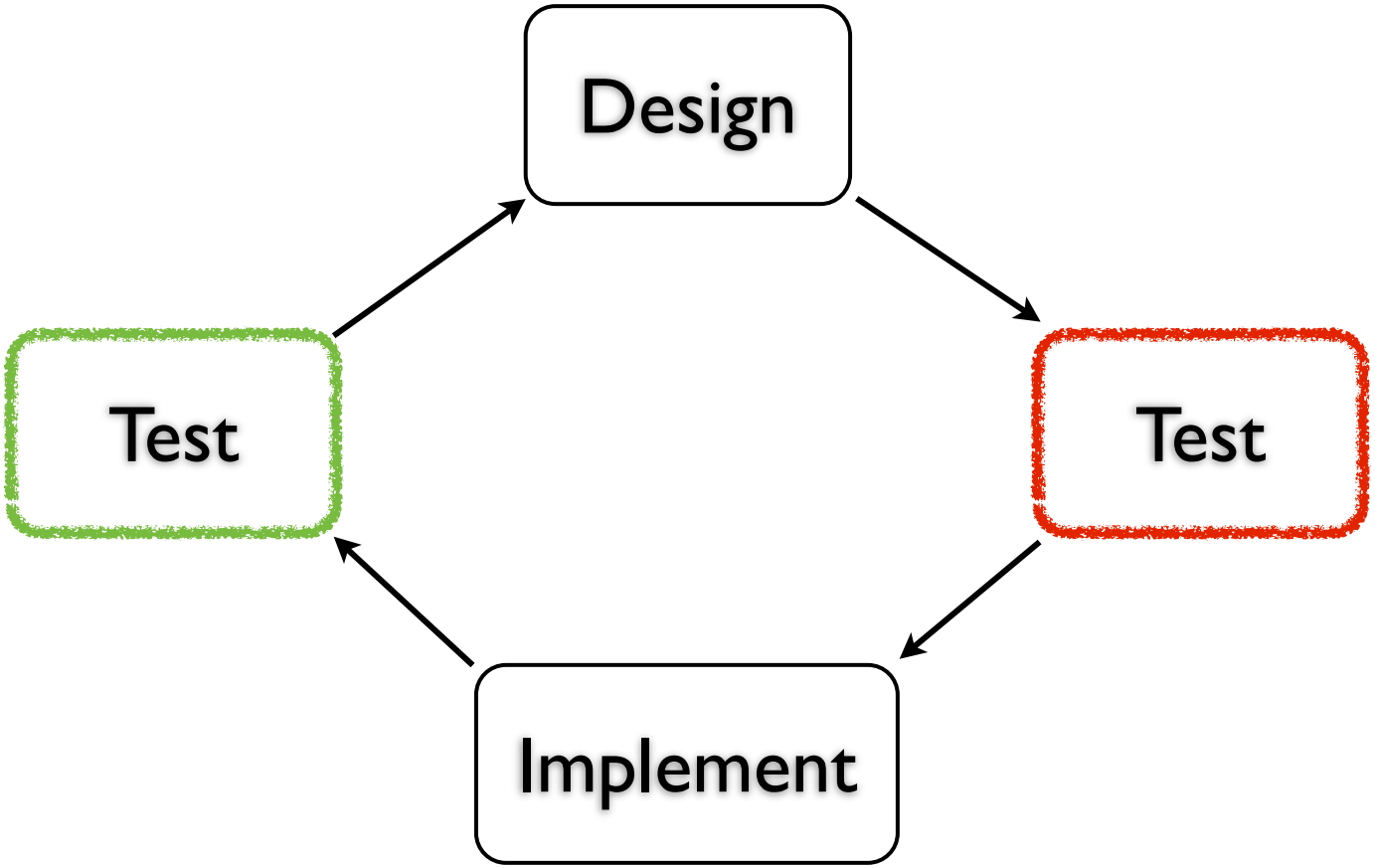
“Classic” Approach



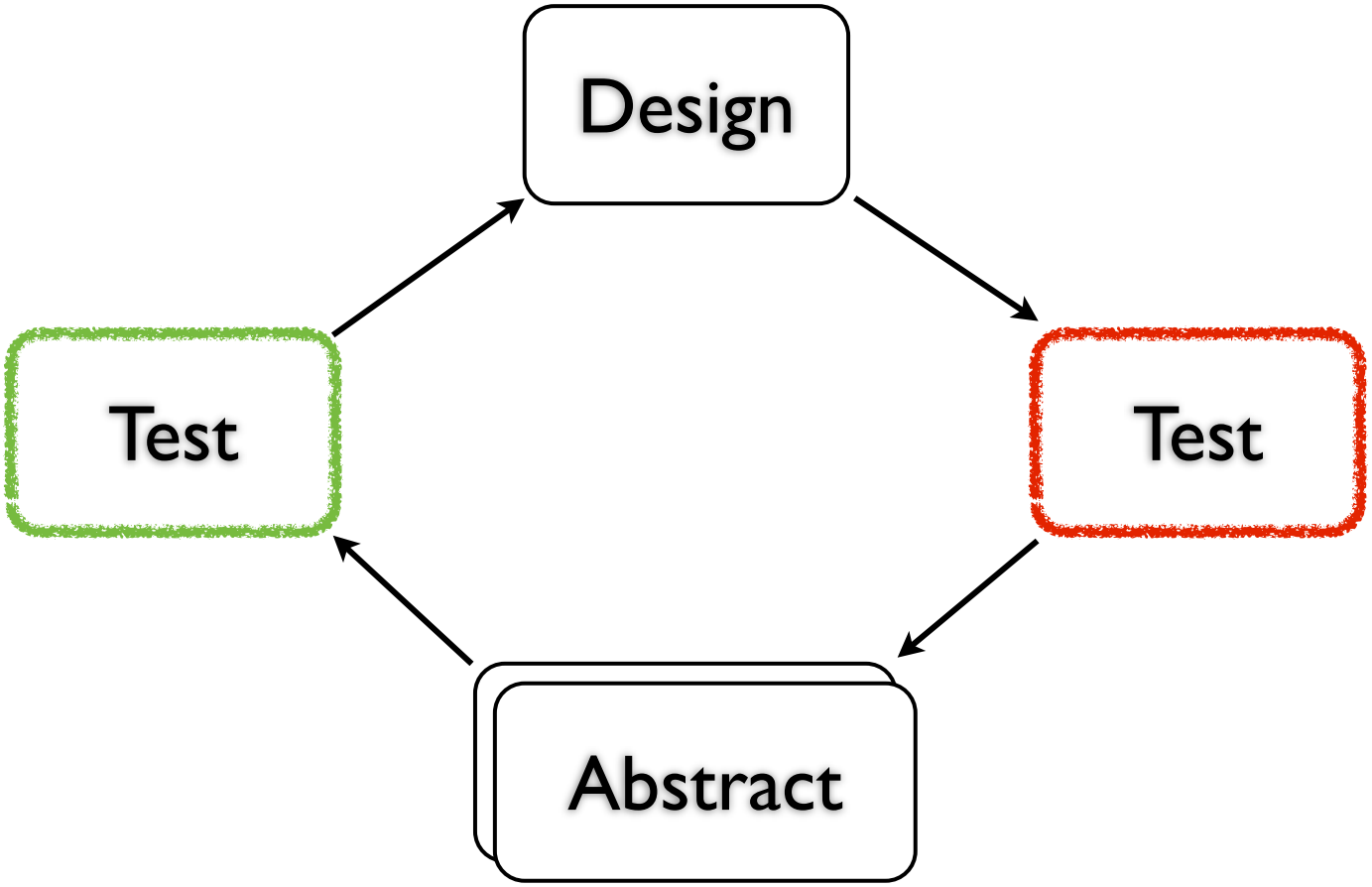
TDD Approach



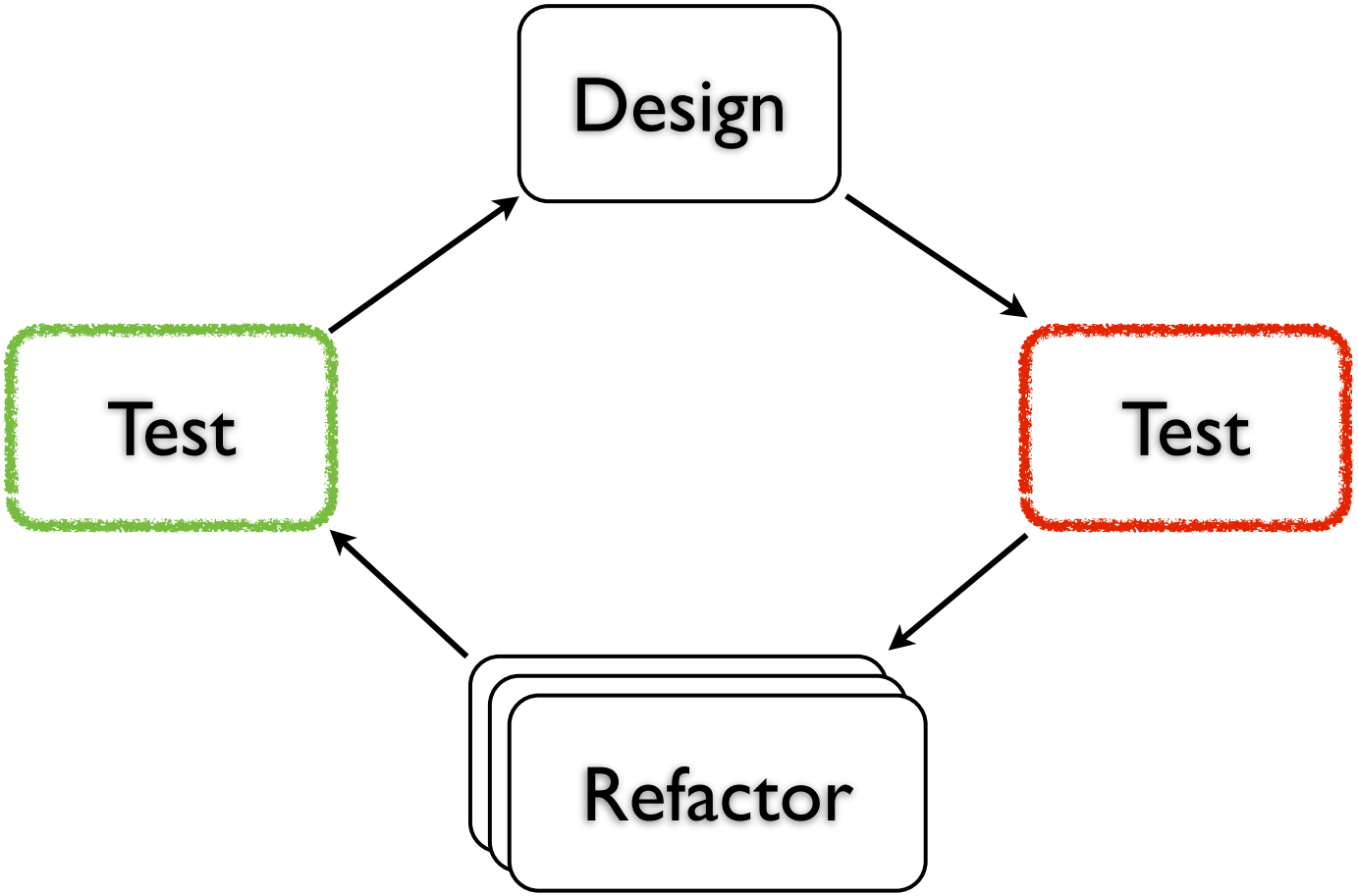
TDD Approach



TDD Approach



TDD Approach



Ideas

- Express the design through tests
- Write minimal code that satisfies tests
- Repeat with other more complex tests
- At a certain point, you will need abstraction and/or refactoring

Esercizio 3

- Implementazione con TDD di un Bank Account.

Start from Tests

```
package it.polimi.deib.se.ese3.bank.test;

import static org.junit.Assert.*;

import org.junit.Test;

public class BankAccountTest {

   @Test
   public void test() {
      fail("Not yet implemented");
   }

}
```

Classe da Testare

```
/* devo testare un BankAccount,
    * quindi creo un campo nella classe
    * di test.
    */
   private BankAccount account = null;
```

Classe da Testare

```
/* devo testare un BankAccount,
   * quindi creo un campo nella classe
   * di test.
   */
   private BankAccount account = null;
```

```
package it.polimi.deib.se.ese3.bank;

public class BankAccount {
   /* ho il costruttore di default,
    * non aggiungo inizialmente nulla
    * all'interfaccia.
    */
}
```

Creo la classe da testare

Setup Classico

```
/* eseguito prima di ogni test */
@Before
public void setup() {
   this.account = new BankAccount();
}
```

Il costruttore di default esiste:
non implemento altro

Primo Test

```
@Test
   public void testBalance() {
      assertEquals(new BigDecimal(0), this.account.balance());
   }
```

- Il Saldo è una funzionalità della classe
- Non esiste, ma penso comunque a cosa dovrebbe restituire nel caso di conto vuoto
- Creo un caso di test corrispondente

Funzionalità Aggiunta

```
// restituisce il saldo del conto.
public BigDecimal balance() {
   // effettuo la modifica minima al
   // codice per soddisfare il test.
   // Non avendo la possibilita' di modificare
   // lo stato del componente,
   // il saldo non potra' che essere zero!
   return new BigDecimal(0);
}
```

Nuovo Test

```
 // testa che il saldo sia 10 a fronte
// di una operazione di deposito di 10.
@Test
public void testDeposit() {
   // manca deposit: devo aggiungere l'operazione
   this.account.deposit(new BigDecimal(10));
   // scenario particolare: so qual e'
   // il valore che mi aspetto.
   assertEquals(new BigDecimal(10), this.account.balance());
}
```

Aggiunta Stato

```
public class BankAccount {
   // Per soddisfare entrambi i test, il modo migliore
   // e' quello di usare uno stato interno che
   // rappresenti il saldo.
   private BigDecimal balance = new BigDecimal(0);

   // restituisce il saldo del conto.
   public BigDecimal balance() {
      return this.balance;
   }

   // deposita una somma sul conto.
   public void deposit(BigDecimal amount) {
      this.balance = this.balance.add(amount);
   }
}
```

Nuovo Test

```
// testa che a fronte di un tentativo
// di deposito di null, venga lanciata una
// IllegalArgumentException
@Test(expected=IllegalArgumentException.class)
public void testDepositNull() {
   this.account.deposit(null);
}
```

Correzione

- Il test fallisce, correggo l'implementazione del metodo in modo che lanci l'eccezione desiderata nel caso particolare

```
public void deposit(BigDecimal amount) {
   if (amount == null) {
     throw new IllegalArgumentException("amount can't be null");
   }
   this.balance = this.balance.add(amount);
}
```

Altro Test

```
// testa che a fronte di un tentativo di
// deposito di somma negativa, venga lanciata una
// IllegalArgumentException
@Test(expected=IllegalArgumentException.class)
public void testDepositNegativeAmount() {
   this.account.deposit(new BigDecimal(-10));
}
```

Correzione

- Il test fallisce ancora, correggo l'implementazione della classe in modo da supportare il caso di test.

```
// deposita una somma sul conto.
public void deposit(BigDecimal amount) {
   if (amount == null) {
      throw new IllegalArgumentException("amount can't be null");
   }
   if (amount.compareTo(new BigDecimal(0)) < 0) {
      throw new IllegalArgumentException("...");
   }
   this.balance = this.balance.add(amount);
}
```

Astrazione

- Attenzione:
  - In realtà la correzione che ho effettuato non solo soddisfa il caso di test specifico (un singolo scenario in cui amount è -10), ma soddisfa l'intenzione di chi l'ha scritto, ossia "per ogni amount negativo"
  - Ho supportato una astrazione (generalizzazione) del caso di test

Altro Test

```
// Testa due depositi di seguito
@Test
public void testTwoDeposits() {
   this.account.deposit(new BigDecimal(10));
   this.account.deposit(new BigDecimal(15));
   assertEquals(new BigDecimal(25), this.account.balance());
}
```

Altro Test

```
// Prelievo da conto con saldo nullo
@Test
public void testWithdraw() {
   this.account.withdraw(new BigDecimal(10));
   assertEquals(new BigDecimal(-10), this.account.balance());
}
```

Funzionalità Nuova

- Implemento il prelievo nella maniera più generale possibile.

```
// preleva una somma dal conto
public void withdraw(BigDecimal amount) {
   this.balance = this.balance.subtract(amount);
}
```

Requisito sul Prelievo

```
// dopo un prelievo con il conto senza soldi,
// il conto e' in rosso
@Test
public void testOverdraft() {
   this.account.withdraw(new BigDecimal(10));
   assertTrue(this.account.isOverdraft());
}
```

Funzionalità Nuova

- Il componente è in uno stato particolare, creo un osservatore che esprima questo stato particolare

```
// restituisce true se e solo se il conto e' in rosso.
public boolean isOverdraft() {
   return this.balance.compareTo(new BigDecimal(0)) < 0;
}
```

Nuovo Requisito

```
// testa che il prelievo sia disabilitato
// se il conto e' in rosso
@Test(expected=IllegalStateException.class)
public void testTwoWithdraws() {
   this.account.withdraw(new BigDecimal(10));
   this.account.withdraw(new BigDecimal(5));
}
```

Correzione

- Il test fallisce, correggo l'implementazione della classe in modo da non supportare il prelievo se il conto è già in rosso.

```
// preleva una somma dal conto
public void withdraw(BigDecimal amount) {
   /* altri check su validita' di amount... */
   if (this.isOverdraft())
      throw new IllegalStateException();
   this.balance = this.balance.subtract(amount);
}
```

Altro Test

```
@Test
   public void testDepositAndWithdraw() {
      this.account.deposit(new BigDecimal(10));
      this.account.withdraw(new BigDecimal(5));
      assertEquals(new BigDecimal(5),this.account.balance());
   }
```

Altro Test

- Test “derivato” dalla funzionalità aggiunta precedentemente

```
// si puo' uscire dall'overdraft
   @Test
   public void testRestoreAccount() {
      this.account.withdraw(new BigDecimal(10));
      this.account.deposit(new BigDecimal(15));
      assertFalse(this.account.isOverdraft());
   }
```

Altro Test

```
// il saldo dopo uscita dall'overdraft e'
// corretto
@Test
public void testBalanceAfterOverdraftRestore() {
   this.account.withdraw(new BigDecimal(10));
   this.account.deposit(new BigDecimal(15));
   assertEquals(new BigDecimal(5), this.account.balance());
}
```

Alternativa

- Implementare un conto corrente in cui si possano fare prelievi anche se il conto è in rosso, ma solo fino ad un massimo scoperto (che ha un valore iniziale ma che può essere cambiato).