

# Proves d'unitat

## - *JUnit* -

# Index

- Introducció
- Prova orientada a objecte
- Prova d'unitat amb JUnit
- Disseny per contracte (DpC)

- Introducció
- Prova orientada a objecte
- Prova d'unitat amb JUnit
- Disseny per contracte (DpC)

# Introducció

- En què difereix la prova de software orientat a objecte del funcional?
  - Alguns tipus d'errors es poden reduir :
    - **mètodes més curts** ⇒ **menor complexitat algorítmica**
    - **l'encapsulació** evita problemes **d'àmbit de dades** incontrolat
  - però d'altres segueixen existint o augmenten :
    - errors tipogràfics
    - els programes OO es tenen més funcions (= mètodes : constructors, destructors, accessors, serveis) ⇒ errors d'interfície més nombrosos
    - el control és distribuït (no jeràrquic) per tot el programa ⇒ més difícil de dissenyar i d'implementar

# Introducció

- Quins nous problemes pot presentar la prova d'unitat?
  - 1. Quina és ara la unitat de prova ?
    - els mètodes no tenen sentit fora del contexte de la seva classe  $\Rightarrow$  la unitat de prova és la **classe** o bé **clusters de poques classes**
    - però com provar-les?
  - 2. Implicacions de l'herència
    - L'herència de mètodes i serveis  $\Rightarrow$  **més proves** i no menys :
      - es fan servir en un contexte diferent (la classe especialitzada)
      - poden ser redefinits
      - possibilitat d'herència múltiple

# Introducció

- 3. Encapsulament
  - no és una font d'errors però **pot ser un obstacle** a la prova si requereix conèixer l'estat dels objectes (valors atributs)
- 4. Polimorfisme (*templates* en C++)
  - cada possible *binding* d'un element polimòrfic (atribut, mètode) requereix una **prova addicional**
  - pot ser difícil de trobar tots els possibles *bindings*, sent major la probabilitat d'error
- 5. Estratègies d'integració
  - **no hi ha una jerarquia de funcions ni de classes**: el control és distribuït, no pas jeràrquic ⇒ no té sentit una integració ascendent o descendente
  - alta interacció / dependència entre mètodes

# Index

- Introducció
- **Prova orientada a objecte**
  - Per què i com es fa
  - Pre i post-condicions
  - Diagrames d'estat
- Prova d'unitat amb JUnit
- Disseny per contracte (DpC)

# Prova orientada a objecte

- Perquè ?

- La prova d'unitat **verifica** si la implementació d'una classe correspon a la seva **especificació** (que es suposa correcta).
- Així, quan es fa la **integració** (en OO, prova d'interacció) és més probable que els **defectes** siguin a la **interfície** entre funcions que a la seva implementació o cos

- Com es fa ?

- 1. Identificar casos de prova
- 2. Implementar un **test driver** que executi cada un dels casos de prova, compari el resultat amb l'esperat i desi la comparació ⇒

- crear un objecte de la classe
- portar-lo a l'estat desitjat (valor d'atributs, associació amb altres objectes)
- enviar el missatge(s) corresponent al cas de prova
- eliminar l'objecte

# Prova orientada a objecte

- Quines classes provar? Hem de considerar :
  - **Risc:** com d'important és l'absència d'errors pel correcte funcionament del programa, com de **greu** és que la classe tingui defectes
  - **Complexitat**, donada pel **nombre d'estats** possibles, **operacions i classes associades**: escollirem classes gens o **poc relacionades amb d'altres**, l'estat d'un objecte depen també de l'estat dels associats
  - **Esforç** que pot suposar fer un **driver**: més com més relacionat amb altres objectes, ens refiem de la correctesa de les altres classes (*veure tema Mock objects*)
- Com trobar els **casos de prova**?
  - Un cas de prova és una parella (dades **d'entrada, sortida** o resultat esperat), corresponent típicament a la crida d'un mètode.

# Index

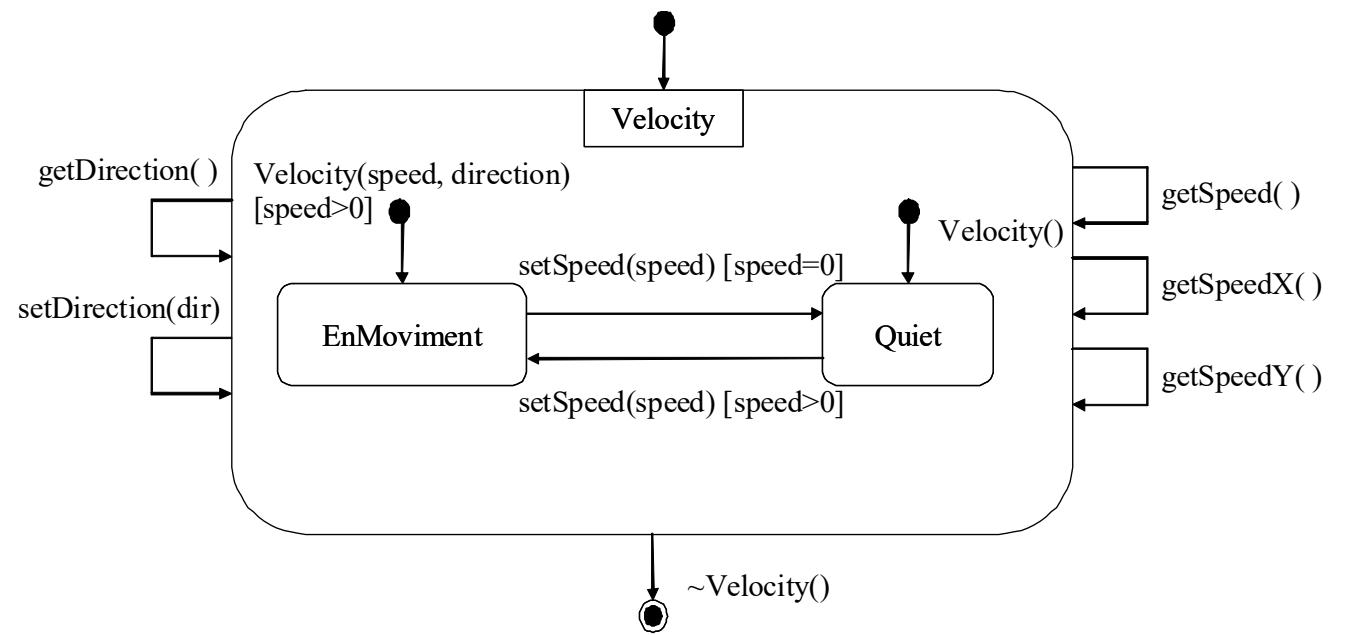
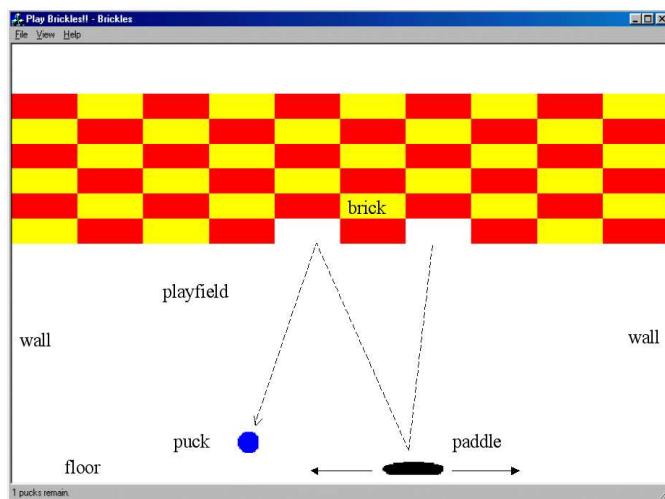
- Introducció
- **Prova orientada a objecte**
  - Per què i com es fa
  - **Pre i post-condicions**
  - Diagrames d'estat
- Prova d'unitat amb JUnit
- Disseny per contracte (DpC)

# Index

- Introducció
- **Prova orientada a objecte**
  - Per què i com es fa
  - Pre i post-condicions
  - **Diagrames d'estat**
- Prova d'unitat amb JUnit
- Disseny per contracte (DpC)

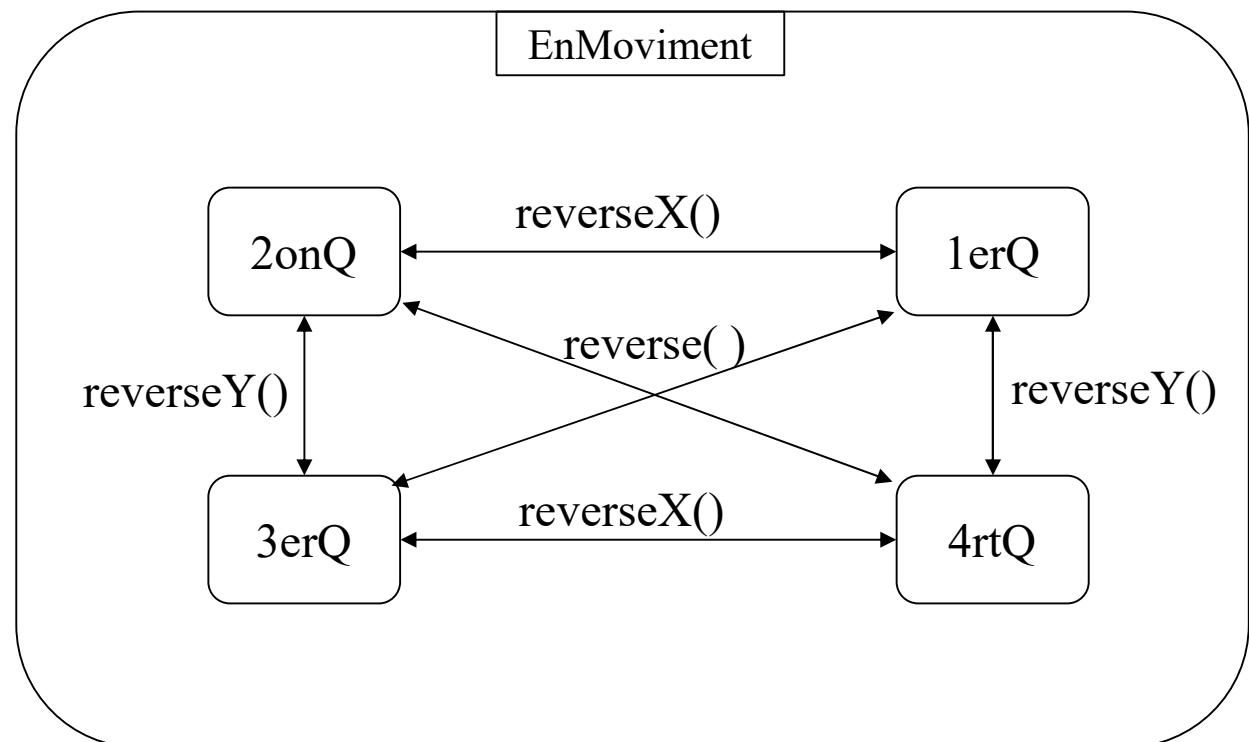
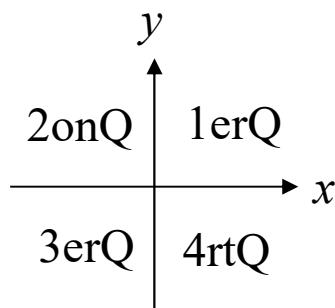
# Diagrams d'estat

- Si la classe s'especifica mitjançant un **diagrama de transició d'estats**, prenem com a casos prova aquells que permeten :
  - 1) passar per tots els estats i/o,
  - 2) Passar per totes les transicions entre estats
- Exemple:



# Diagrams d'estat

- Subestats en l'estat EnMoviment



# Diagrams d'estat

Alguns dels casos de prova per cobrir totes les transicions entre estats de EnMoviment = interaccions entre reverse, reverseX, reverseY

Description	Input		Output	
	Setup	Event(s)	Result State	Exceptions
test reverse()	OUT:Velocity[speed = 10, direction = 0]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 180	none
	OUT:Velocity[speed = 10, direction = 30]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 210	none
	OUT:Velocity[speed = 10, direction = 90]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 270	none
	OUT:Velocity[speed = 10, direction = 135]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 315	none
	OUT:Velocity[speed = 10, direction = 180]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 0	none
	OUT:Velocity[speed = 10, direction = 182]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 2	none
	OUT:Velocity[speed = 10, direction = 270]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 90	none
	OUT:Velocity[speed = 10, direction = 335]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 155	none
	OUT:Velocity[speed = 10, direction = 0]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 180	none
reverseX()	OUT:Velocity[speed = 10, direction = 30]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 150	none
	OUT:Velocity[speed = 10, direction = 90]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 90	none
	OUT:Velocity[speed = 10, direction = 135]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 45	none
	OUT:Velocity[speed = 10, direction = 180]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 0	none
	OUT:Velocity[speed = 10, direction = 182]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 358	none
	OUT:Velocity[speed = 10, direction = 270]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 270	none
	OUT:Velocity[speed = 10, direction = 335]	OUT.reverse()	OUT.speed = 10 and OUT.direction = 205	none
	...			
reverseY()				

# Index

- Introducció
- Prova orientada a objecte
- **Prova d'unitat amb JUnit**
- Disseny per contracte (DpC)

# JUnit



[www.JUnit.org](http://www.JUnit.org)

Portat a C++ :

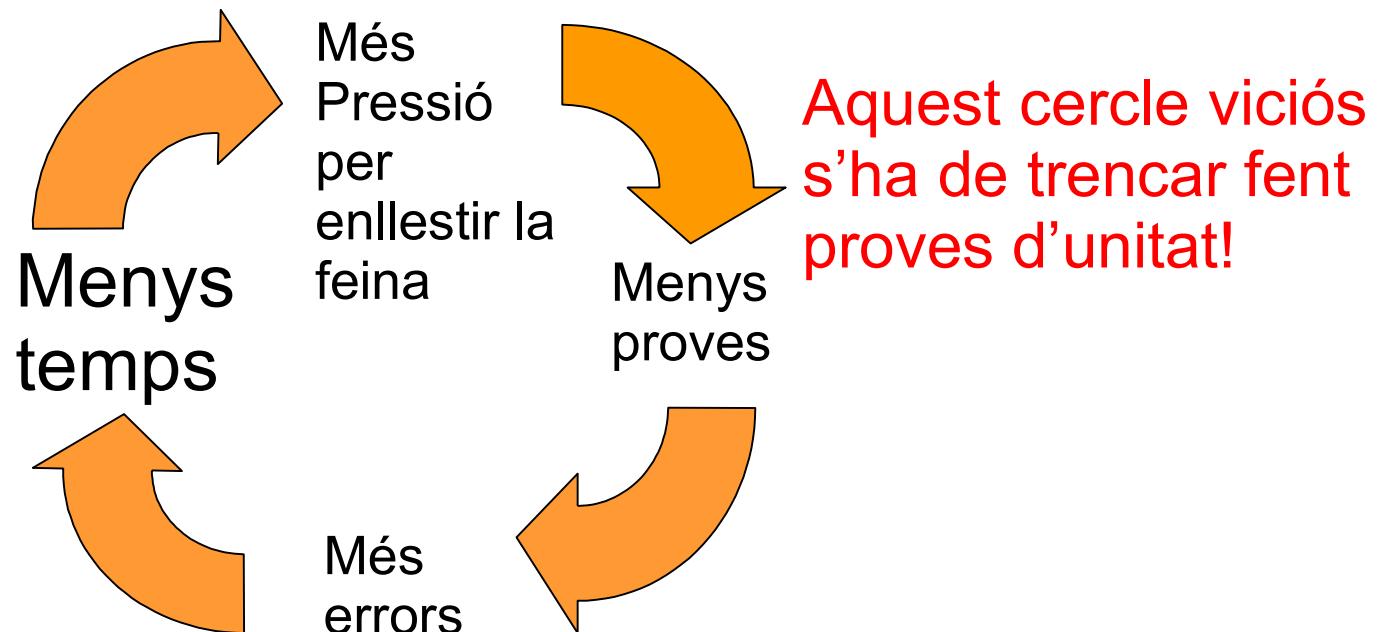
<http://cppunit.sourceforge.net>

i a molts altres lenguatges :

<http://en.wikipedia.org/wiki/XUnit>

*"Never in the field of software development was so much owed by so many to so few lines of code"*, Martin Fowler.

- Problema
  - Els programadors reconeixen la utilitat de les **proves d'unitat**, però ... pocs escriuen *test drivers* per fer proves d'unitat.
- Perquè ?



- És un *framework* per a implementar i executar proves d'unitat orientat a objecte.
  - *Framework* : **conjunt de classes** i objectes que col·laboren entre elles, de les quals **heredar** (no instanciar) per obtenir funcionalitats.
- Avantatges :
  - **fàcil d'aprendre i d'usar**
  - és **fàcil implementar** les proves
  - és directe **executar-les i repetir** les execucions
  - permet implementar proves que mantinguin el seu valor al llarg del temps: **integrades al codi**
  - combinar amb proves de diferents autors i executar-les conjuntament
  - resultat fàcilment interpretable
- Portat a altres llenguatges:
  - C++ : <http://cppunit.sourceforge.net>, <http://cxxtest.com/guide.html>
  - altres lenguatges: <http://en.wikipedia.org/wiki/XUnit>

# JUnit - Exemple

- **Exemple Money** (representació de quantitats monetàries en diferents divises i la seva aritmètica)

Volem definir **l'operació de sumar** dos imports (quan són en la mateixa divisa). Primer, definim un cas de prova:

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase  
{  
    //...  
    public void testSimpleAdd()  
    {  
        // [12 CHF] + [14 CHF] == [26 CHF]  
        Money f12CHF = new Money(12, "CHF");  
        Money f14CHF = new Money(14, "CHF");  
        Money expected = new Money(26, "CHF");  
        Money result = f12CHF.add(f14CHF);  
  
        assertTrue(result.iAmount==expected.iAmount);  
        assertTrue(result.sCurrency==expected.sCurrency);  
    }  
}
```

Filla de  
TestCase

Estem definint la API del constructor i la de la funció add.

També estem definint el nom dels atributs! Com que això no és un requeriment i només ens interessa definir un exemple, els definirem com privats. Això implica haver de definir getters i setters (i la seva API). Modifiquem el codi de prova ...

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    //...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());
    }
}
```

Ara podem passar a definir el codi de Money () ...

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    //...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());
    }
}
```

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }
}
```

Ara hauriem de definir el constructor de `Money()`, però abans de definir qualsevol mètode hem de definir el seu codi de prova. Per tant, primer definim un mètode de prova pel constructor.

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    //...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12,"CHF");
        Money f14CHF = new Money(14,"CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount()==expected.amount());
        assertTrue(result.currency()==expected.currency());
    }

    public void testConstructor()
    {
        Money f12CHF = new Money(12,"CHF");

        assertTrue(f12CHF.amount()==12);
        assertEquals(f12CHF.currency(),"CHF");
    }
}
```

Un cop tenim el codi de prova del constructor, podem implementar-lo.

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }
}
```

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    //...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());
    }

    public void testConstructor()
    {
        Money f12CHF = new Money(12, "CHF");

        assertTrue(f12CHF.amount() == 12);
        assertEquals(f12CHF.currency(), "CHF");
    }
}
```

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }
}
```

Un cop comprovem que el constructor funciona correctament, abans d'implementar add() volem poder comparar dos objectes de la classe Money() utilitzant assertEquals(), ja que serà més còmode que no fent-ho atribut per atribut. Escribim un cas de prova.

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    //...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());

        assertEquals(result, expected);
    }

    public void testConstructor()
    {
        Money f12CHF = new Money(12, "CHF");

        assertTrue(f12CHF.amount() == 12);
        assertEquals(f12CHF.currency(), "CHF");
    }
}
```

Ens adonem que per a poder utilitzar `assertEquals()` la classe dels objectes comparats ha de tenir un metode `equals()`. Abans d'implementar-lo haurem de definir un cas de prova.

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }
}
```

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    //...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());

        assertEquals(result, expected);
    }

    public void testConstructor()
    {
        Money f12CHF = new Money(12, "CHF");

        assertTrue(f12CHF.amount() == 12);
        assertEquals(f12CHF.currency(), "CHF");
    }

    public void testMoneyEquals() {
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money equalMoney = new Money(12, "CHF");
        assertEquals(f12CHF, equalMoney);
        assertFalse(f12CHF.equals(f14CHF));
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
    }
}
```

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }
}
```

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    ...
    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());

        assertEquals(result, expected);
    }

    public void testConstructor()
    {
        Money f12CHF = new Money(12, "CHF");

        assertTrue(f12CHF.amount() == 12);
        assertEquals(f12CHF.currency(), "CHF");
    }

    public void testMoneyEquals()
    {
        Money f12CHF = new Money(12, "CHF");
        Money f14CHF = new Money(14, "CHF");
        Money equalMoney = new Money(12, "CHF");
        assertEquals(f12CHF, equalMoney);
        assertFalse(f12CHF.equals(f14CHF));
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
    }
}
```

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }
}
```

Això està repetit també al codi de prova `testSimpleAdd()`, per tant pot anar al mètode `setUp()` de la classe `MoneyTest`, però hem d'anar en compte perquè les variables `f12CHF` i `f14CHF` han de ser atributs de la classe de prova `MoneyTest`.

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    protected void setUp()
    {
        Money f12CHF = new Money(12,"CHF");
        Money f14CHF = new Money(14,"CHF");
    }

    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());
        assertEquals(result, expected);
    }

    public void testConstructor()
    {
        assertEquals(f12CHF.amount(), 12);
        assertEquals(f12CHF.currency(), "CHF");
    }

    public void testMoneyEquals()
    {
        Money equalMoney = new Money(12, "CHF");
        assertEquals(f12CHF, equalMoney);
        assertFalse(f12CHF.equals(f14CHF));
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
    }
}
```

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }
}
```

Si ho fem així segur que no funcionarà porque aquí f12CHF i f14CHF són objectes locals a setup() (shadowing dels atributs de la classe). Ara pòdem passar a implementar equals().

Test i Qualitat

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    Money f12CHF, f14CHF;

    protected void setUp()
    {
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
    }

    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());
        assertEquals(result, expected);
    }

    public void testConstructor()
    {
        assertTrue(f12CHF.amount() == 12);
        assertEquals(f12CHF.currency(), "CHF");
    }

    public void testMoneyEquals()
    {
        Money equalMoney = new Money(12, "CHF");
        assertEquals(f12CHF, equalMoney);
        assertFalse(f12CHF.equals(f14CHF));
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
    }
}
```

Sobreprogramació.  
El cas de prova no  
ens demana que  
tinguem això en  
compte.

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }

    public boolean equals(Object anObject)
    {
        if (anObject instanceof Money)
        {
            Money aMoney = (Money) anObject;
            return
                (aMoney.currency().equals(currency())) &&
                (amount() == aMoney.amount());
        }
        return false;
    }
}
```

Ara podem comprobar si el cas de prova per Money.equals() funciona. Si és el cas, podrem passar a implementar Money.add()

# JUnit - Exemple

## Codi de prova

```
public class MoneyTest extends TestCase
{
    Money f12CHF, f14CHF;

    protected void setUp()
    {
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
    }

    public void testSimpleAdd()
    {
        // [12 CHF] + [14 CHF] == [26 CHF]
        Money expected = new Money(26, "CHF");
        Money result = f12CHF.add(f14CHF);

        assertTrue(result.amount() == expected.amount());
        assertTrue(result.currency() == expected.currency());
        assertEquals(result, expected);
    }

    public void testConstructor()
    {
        assertTrue(f12CHF.amount() == 12);
        assertEquals(f12CHF.currency(), "CHF");
    }

    public void testMoneyEquals()
    {
        Money equalMoney = new Money(12, "CHF");
        assertEquals(f12CHF, equalMoney);
        assertFalse(f12CHF.equals(f14CHF));
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
    }
}
```

## Codi desenvolupat

```
public class Money
{
    private int iAmount;
    private String sCurrency;

    public int amount(){ return iAmount; }
    public String currency(){ return sCurrency; }

    public Money(int amount, String currency)
    {
        iAmount = amount;
        sCurrency = currency;
    }

    public boolean equals(Object anObject)
    {
        if (anObject instanceof Money)
        {
            Money aMoney = (Money) anObject;
            return
                (aMoney.currency().equals(currency())) &&
                (amount() == aMoney.amount());
        }
        return false;
    }

    public Money add(Money m)
    {
        return new Money(amount() + m.amount(), currency());
    }
}
```

Ara podem comprobar si el cas de prova per Money.add() funciona.

# JUnit - Exemple

- Mentalitat:

- si una **operació** (o *feature*) **no té proves** associades,  
**assumim que no funciona**: més fiable que suposar que sí  
que funciona sense provar-la !

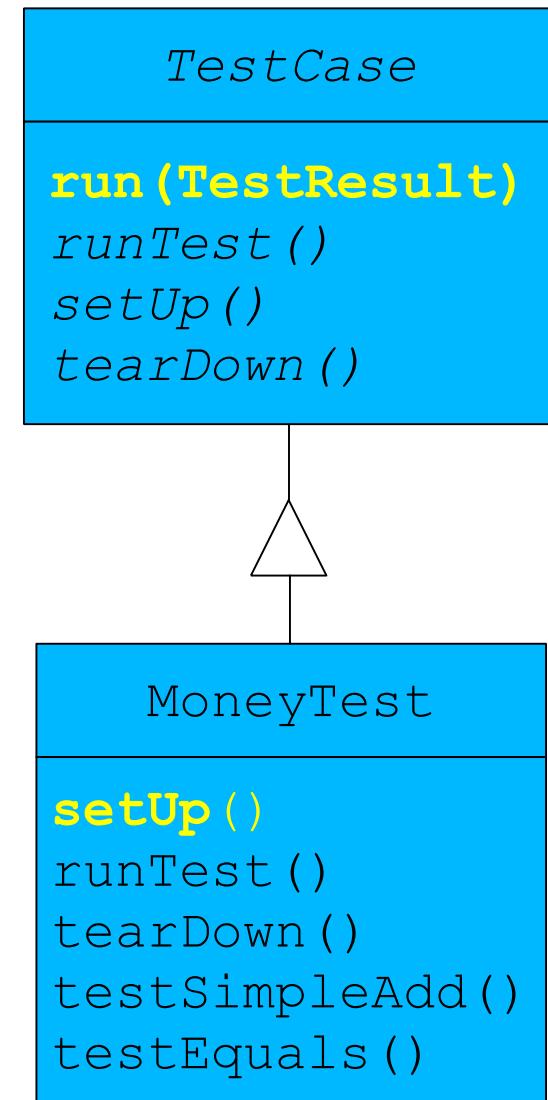
# JUnit - Exemple

- Resum: Tots els casos de prova d'unitat segueixen l'esquema:
  - Crear els objectes usats durant el cas de prova i dur-los a l'estat desitjat (valors atributs): *fixture* (*setUp ()* a JUnit)
  - enviar missatges a objectes de la *fixture* } *runTest ()*
  - fer certes comprobacions } a JUnit
    - `assertTrue()`
    - `assertFalse()`
    - `assertEquals()`
    - `assertNull()`
    - `assertSame()` ...
  - eliminar els objectes de la *fixture* (*tearDown ()* a JUnit)

# JUnit - Exemple

```
public void run(TestResult result) {  
    result.startTest(this);  
    setUp();  
    runTest();  
    tearDown();  
}
```

```
protected void setUp() {  
    f12CHF= new Money(12, "CHF");  
    f14CHF= new Money(14, "CHF");  
    f7USD = new Money( 7, "USD");  
    f21USD= new Money(21, "USD");  
    // ...  
}
```



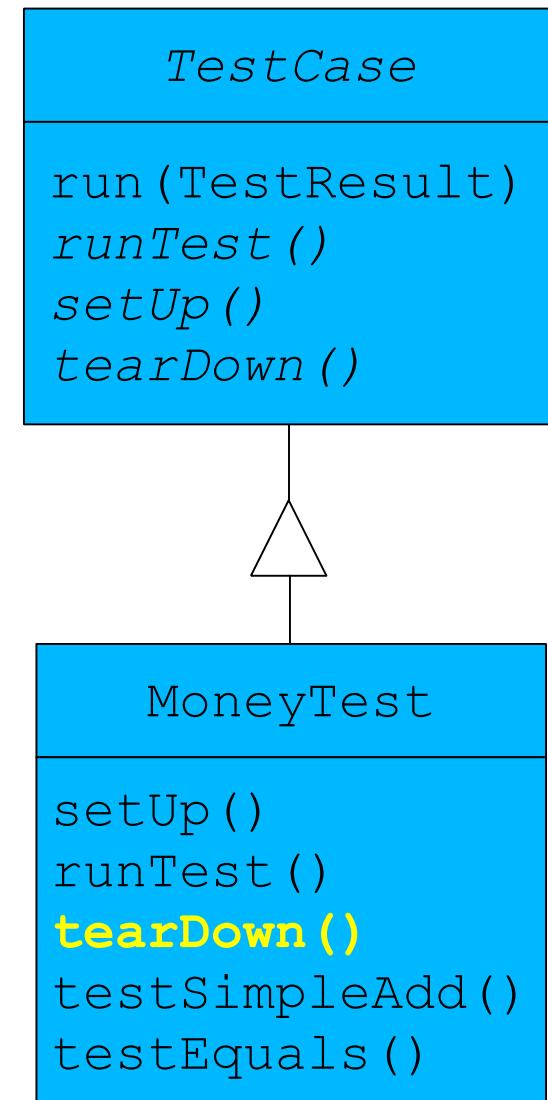
# JUnit - Exemple

- `setUp()` s'executa abans de cada cas de prova `testX()`
- `tearDown()` després.

(Existeix un `oneTimeSetup()` i un `oneTimeTearDown()`, que s'executen només 1 cop per classe)

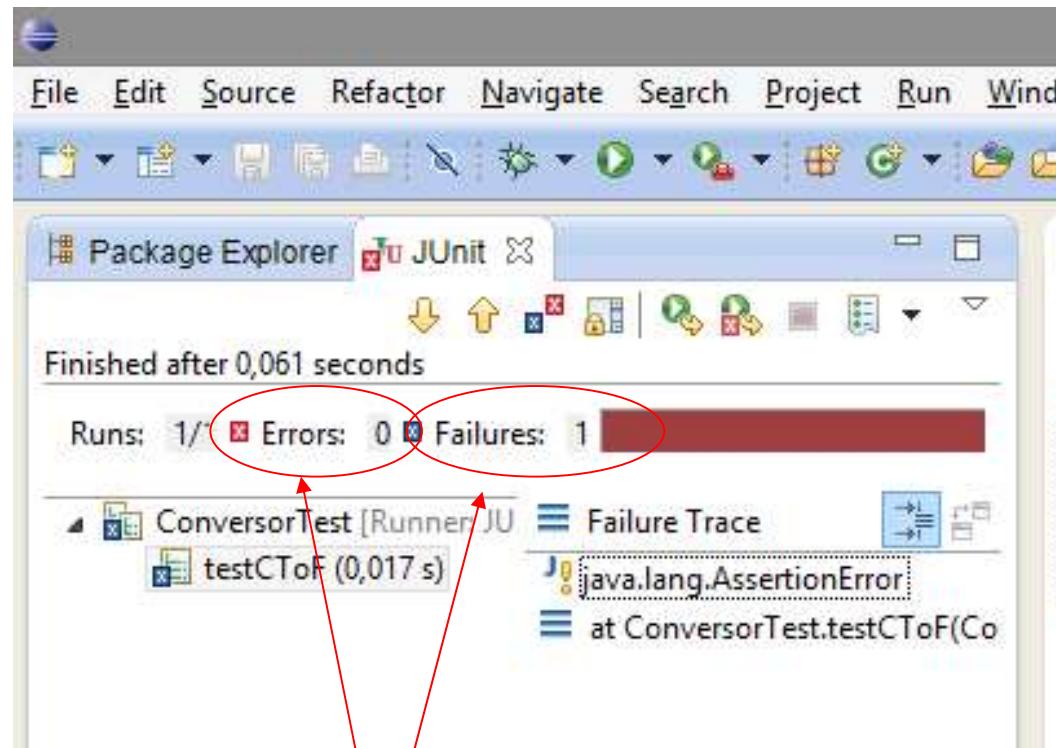
En Java no cal, però es pot fer :

```
protected void tearDown() {  
    f12CHF= null;  
    f14CHF= null;  
    f7USD = null;  
    f21USD= null;  
    //...  
    System.gc(); // garbage collection  
}
```



# JUnit - Exemple

- Interfície gràfica



Cas de prova exitós!!!

Quina diferència hi ha?

- **failure** : **problemes anticipats i comprobats amb els asserts** dels casos de prova
- **error** : **problemes no previstos**, com access fora de rang en un vector, que generen excepcions en Java

# JUnit - Resum

- Perquè és convenient fer proves d'unitat?
  - trobem defectes **aviat**
  - **Detectem** aquells **molts petits defectes** que provoquen un comportament caòtic (I llavors és difícil d'identificar-ne les causes)
  - les **prove**s **influeixen en el codi** que fem
  - **augmenta la confiança** en el codi que fem
  - la necessitat de **pensar les prove**s ens fa **programar millor**

# JUnit - Resum

- Per què fer-les amb JUnit?
  - és molt més fàcil (**automatització**) que d'una altra manera
  - són proves **repetibles** en qualsevol moment
  - el codi de **prove**s està **separat** del **codi** a provar
  - aquest framework ha estat portat a molts llenguatges de programació (<http://en.wikipedia.org/wiki/XUnit>)
  - és **l'estàndard** de prova d'unitat en **Java**

# Index

- Introducció
- Prova orientada a objecte
- Prova d'unitat amb JUnit
- **Disseny per contracte (DpC)**

- En el DpC (invariants, pre i post-condicions)
  - 1) és prova d'unitat?
  - 2) s'implementa amb el JUnit?

↓

- les **proves d'unitat s'executen a part** de la aplicació: els casos de prova són en classes diferents
- les **assercions** del DpC **s'executen simultàniament**
- en **DpC** verifiquem que es compleixen les **especificacions**
- tot i complir-se les especificacions, hi poden **haver defectes!**

# Bibliografia

- *Pragmatic unit testing in Java with JUnit*. Andy Hunt and Dave Thomas. The Pragmatic Programmer, 2003.  
**Molt bo : didàctic, fàcil de llegir, breu**
- *JUnit test infected: programmers love writing tests*. Erich Gamma, Kenneth Beck. [junit.sourceforge.net/doc/testinfected/testing.htm](http://junit.sourceforge.net/doc/testinfected/testing.htm)  
**Lectura imprescindible.**
- *Unit testing in Java: how tests drive the code*. Johannes Link, Peter Frohlich. Morgan Kauffman, 2003.  
**Test driven development, un mètode àgil de desenvolupament.**
- *JUnit : A cook's tour*. Erich Gamma, Kenneth Beck. [junit.sourceforge.net/doc/cookstour/cookstour.htm](http://junit.sourceforge.net/doc/cookstour/cookstour.htm)  
**Com és per dins JUnit (patrons de disseny)**