# Tema 7 - Generación y paso de tests (DCA)

**HISTORIAL DE REVISIONES**

| NÚMERO | FECHA | MODIFICACIONES | NOMBRE |
|---|---|---|---|
| | | | |

# Índice

Logo DLSI

**Tema 7 - Generación y paso de tests**
**Curso 2018-2019**

## 1. Test... Test... Test... Did I say Test?

```
"Test early. Test Often. Test Automatically.  Tests that run
  with every build are much more effective than test plans
  that sit on a shelf."
```

— The Pragmatic Programmer *Quick Reference Guide Tip 62*

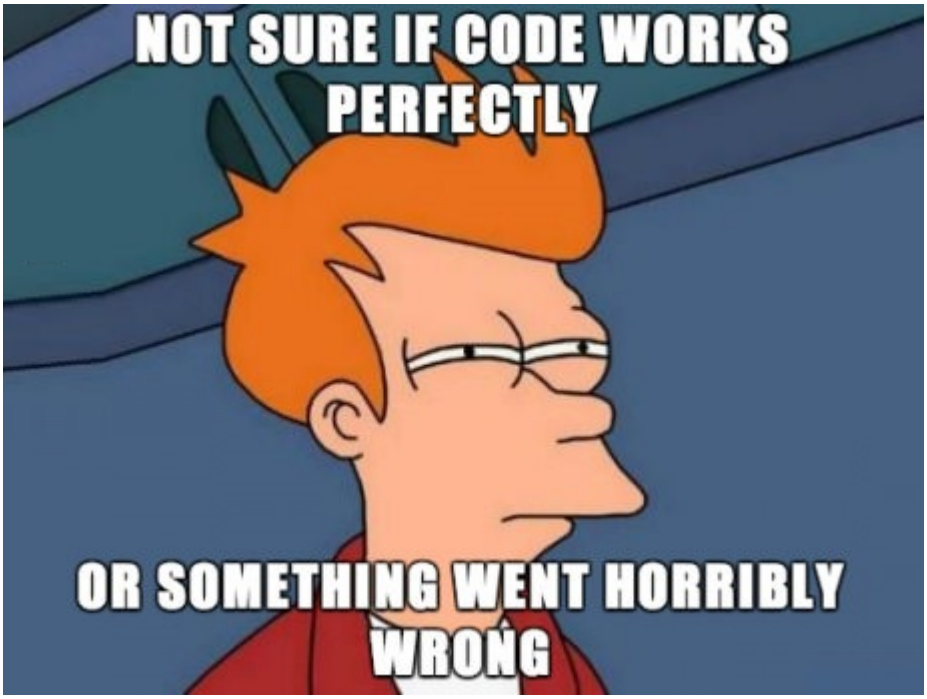
- Los *chicos* de sqlite parece que se lo han tomado en serio:

```
"As of version 3.12.0 (2016-03-29), the SQLite library consists of
approximately 116.3 KSLOC of C code. (KSLOC means thousands of
"Source Lines Of Code" or, in other words, lines of code excluding
blank lines and comments.) By comparison, the project has 787
times as much test code and test scripts - 91577.3 KSLOC."
```

— SQL database engine developers *How SQLite Is Tested*

## 2. Testea tanto como puedas.

- Que no se te quede esta cara delante de tu *jef@*, o peor, de *un cliente*:



## 3. Preliminares (I)

- El propósito del paso de tests es demostrar que en nuestro software existen fallos, no que está libre de ellos.

> `"Program testing can be used to show the presence of bugs, but never to show their absence!"`
>
> — Edsger Dijkstra

- Por tanto el paso de tests a nuestro software es una técnica para hacer que falle... no para ver lo maravilloso que es.
- Comprobar un programa consiste en ver si su comportamiento es correcto para *cualquier posible entrada* que pueda tener y **dado que esto es imposible**, debemos comprobarlo para todas aquellas entradas que tengan una probabilidad razonable de hacer que falle ( *si es que tiene un fallo* ).

## 4. Preliminares (II)

- A este conjunto de entradas es a lo que llamamos un "**Test Suite**".
- Se debe procurar que no le lleven al programa mucho tiempo en su ejecución.
- La clave para conseguir un **test suite** apropiado es particionar el espacio de todas las posibles entradas en subconjuntos que proporcionan información equivalente sobre la corrección del programa y crear el **test suite** con una entrada de cada uno de estos subconjuntos.
- Esto es prácticamente imposible... así que hay que emplear algunas heurísticas para obtener el **test suite**.
- Una forma de conseguir acercarnos al **test suite** perfecto sería haciendo una **partición** del conjunto de datos de entrada en subconjuntos de manera que cada elemento del conjunto original perteneciera exactamente a uno de estos subconjuntos.

## 5. Preliminares (III)

- Un ejemplo, la función: *es_mayor (x, y)*.

  **Conjunto de entrada original**

  Todos los posibles pares "`x`" e "`y`" de enteros.

  UNA POSIBLE PARTICIÓN:

  1. x positivo, y positivo
  2. x negativo, y negativo
  3. x positivo, y negativo
  4. x negativo, y positivo
  5. x == 0, y == 0
  6. x == 0, y != 0
  7. x != 0, y == 0

- Si ahora comprobamos la función *es_mayor (x, y)* con una entrada de, al menos uno de, estos subconjuntos tendríamos una probabilidad razonable de que aparecería un fallo si existiera... pero no una garantía absoluta!

## 6. Tipos de tests

BLACKBOX TESTING:

- Se crean sin mirar el código a testear. Se basan en la especificación de lo que debe hacer el código.
- Permiten que programadores del código y de los tests sean distintos.
- Son robustos respecto a cambios en la implementación del código a testear.
- Son los tipos de tests que habitualmente se pasan a las prácticas.

WHITEBOX O GLASSBOX TESTING:

- Se tiene acceso al código a testear.
- Complementan a los anteriores y son más fáciles de crear que aquellos.
- Al tener acceso al código debemos fijarnos al construir los tests en las sentencias `if-then-else`, `bucles`, `try-catch` presentes en el código a testear.

## 7. Qué testear

En base a la granularidad de lo que comprueban:

TESTS UNITARIOS

- Están destinados a módulos individuales, p.e. *clases*.

TESTS DE INTEGRACIÓN

- Evalúan cómo se ejecutan una serie de módulos cuando interactúan.

TESTS DEL SISTEMA

- Evalúan todo el sistema al completo.

OTROS TESTS

- Agotamiento de recursos. Errores y recuperación.
- Rendimiento
- Usabilidad. Son un tipo de tests diferentes a los anteriores.

## 8. Cómo testear

**Tests de regresión**
: El nuevo código no debe estropear lo que ya funcionaba.

**Tests de datos**
: Datos reales y sintéticos... *¿alguien dijo 30 de febrero*?

**Tests del Interfaz de Usuario**
: GUI

**Tests _de los tests**

¡La alarma tiene que sonar cuando debe hacerlo! (echa un vistazo a las prácticas en grupo, concretamente a *Yarn* ).

"Use saboteurs to Test Your Testing"

— Pragmatic Programmer *Q.R.G. Tip 64*

- **Tests minuciosos**. Cobertura de un test.

```
int test (int a, int b) { return a / (a+b); }
```

## 9. Cúando testear

- Nunca dejarlo para el final, menos aún cerca de un *deadline*.
- Debería hacerse automáticamente... incluida la interpretación de los datos.
- Suele ser habitual convertir el paso de los tests en un objetivo de `make`: `make test`.
- No todos los tests se pueden pasar tan seguidamente ( *stress tests* ), pero esto se puede tener en cuenta y automatizar.
- **Ahh!, una cosa más**: Si una persona detecta un fallo... debería ser la última persona que detecta ese fallo... inmediatamente deberíamos tener un test para capturarlo.

## 10. Cobertura de un test.

- Se trata de una medida empleada con tests tipo WhiteBox.
- Trata de estimar cuánta funcionalidad se ha testeado.
- Suele *medir* el porcentaje de instrucciones testeadas ( *Instruction coverage* ) y el porcentaje de ramas del código usadas ( *Branch coverage* ).
- Para tests tipo BlackBox se suele medir el porcentaje de la especificación testeado ( *Specification coverage* ).

## 11. Test drivers. Conducting tests.

- Hoy en día los procesos de testeo están automatizados.
- Los tests se pasan empleando `test drivers`, los cuales...
  - Preparan el entorno para invocar el programa o unidad a testear.
  - Invocan el programa o unidad a testear con un conjunto de datos de entrada.
  - Guardan los resultados de esta ejecución
  - Comprueban la validez de estos resultados
  - Preparan el informe apropiado.

## 12. Test drivers. xUnit (I).

- Es el termino empleado para describir una serie de herramientas de test que se comportan de manera similar.
- Tienen su origen en SUnit creado por Kent Beck para Smalltalk.
- Posteriormente `SUnit` se portó a Java bajo el nombre JUnit.
- Disponemos de versiones de xUnit para ".Net" (NUnit), "C++" (cppunit), etc...

## 13. Test drivers. xUnit (II).

CONCEPTOS EMPLEADOS EN ENTORNOS TIPO XUNIT:

**Test runner**
El encargado de ejecutar los tests y proporcionar los resultados.

**Test case**
Cada uno de los tests pasados al software.

**Test fixtures**
También llamado *test context*, garantiza las precondiciones necesarias para ejecutar un test y que el "estado" se restaura al original tras ejecutar el test.

**Test suites**
Conjunto de tests que comparten el mismo "`fixture`". El orden de los tests no debería importar.

**Test execution**
Es la ejecución de cada uno de los tests individuales.

## 14. Caso de uso: junit (I).

**Ya lo conocemos de asignaturas como Programación-3**

```java
package modelo;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class CoordenadaTest {
Coordenada c;

@Before
public void setUp() throws Exception { c = new Coordenada(10, 5); }
@After
public void tearDown() throws Exception { ... }

@BeforeClass
public static void setUpClass() throws Exception {
        /* Code executed before the first test method */
}

@AfterClass
public static void tearDownClass() throws Exception {
        /* Code executed after the last test method */
}
```

## 15. Caso de uso: junit (II).

```
...
@Test
public final void testGetX() {
        assertEquals("x", 10, c.getX(), 0.001);
}

@Test
public final void testGetY() {
        assertEquals("y", 5, c.getY(), 0.001);
}

@Test
public final void testConstructorCopia() {
        Coordenada c2 = new Coordenada(c);
        assertEquals("c2.x", c2.getX(), c.getX(), 0.001);
        assertEquals("c2.y", c2.getY(), c.getY(), 0.001);
}

@Test
public final void testInicializacion() {
        Coordenada c3 = new Coordenada();
        assertEquals("c3.x", Coordenada.VACIO, c3.getX(), 0.001);
        assertEquals("c3.y", Coordenada.VACIO, c3.getY(), 0.001);
}
}
```

## 16. Caso de uso: cxxtest (I).

- Es un entorno de testeo tipo xUnit para C++.
- Trata de ser similar a JUnit.
- CxxTest soporta el descubrimiento de tests... no es necesario registrar los tests.
- Los tests se definen en archivos de cabecera (`.h`) que se procesan con la aplicación: `cxxtestgen` que genera los ficheros fuente (`.cc`) necesarios para el *test runner*.
- En el `makefile` correspondiente pondríamos una regla como esta:

```
  poo-p1-test.cc : poo-p1-test.h
        cxxtestgen  -w wn -o $@ --error-printer $^
```

- No necesita enlazarse con ninguna biblioteca adicional.

## 17. Caso de uso: cxxtest (II).

```
// File: poo-p1-test.h
#include <cxxtest/TestSuite.h>
class PuntoTestSuite : public CxxTest::TestSuite
{
public:
        void testConstructores( void ) {
                Punto p0;                              // 1 //
                TS_ASSERT_EQUALS ( p0.getX (), 0.0 );
                TS_ASSERT_EQUALS ( p0.getY (), 0.0 );

                Punto p1 (p0);                // 2 //
                TS_ASSERT_EQUALS ( p1.getX (), 0.0 );
                TS_ASSERT_EQUALS ( p1.getY (), 0.0 );
        }

        void testAccesores (void) {
                Punto p0;

                p0.setX (2.3);
                TS_ASSERT_EQUALS ( p0.getX (), 2.3 );
                TS_ASSERT_EQUALS ( p0.getX (), p0.getLong () );

                p0.setY (4.2);
                TS_ASSERT_EQUALS ( p0.getY (), 4.2 );
                TS_ASSERT_EQUALS ( p0.getY (), p0.getLat () );
        }
};
```

## 18. Caso de uso: cxxtest (III).

**Ejemplo de salida generada por cxxtestgen**

```
// File: poo-p1-test.cc
/* Generated file, do not edit */
#ifndef CXXTEST_RUNNING
#define CXXTEST_RUNNING
#endif

#define _CXXTEST_HAVE_STD
#include <cxxtest/TestListener.h>
#include <cxxtest/TestTracker.h>
#include <cxxtest/TestRunner.h>
#include <cxxtest/RealDescriptions.h>
#include <cxxtest/ErrorPrinter.h>

int main() { return CxxTest::ErrorPrinter().run();      }
#include "poo-p1-test.h"

static PuntoTestSuite suite_PuntoTestSuite;

static CxxTest::List Tests_PuntoTestSuite = { 0, 0 };
CxxTest::StaticSuiteDescription suiteDescription_PuntoTestSuite(
    "poo-p1-test.h", 33, "PuntoTestSuite",
        suite_PuntoTestSuite, Tests_PuntoTestSuite );
```

## 19. Caso de uso: cxxtest (IV).

```
...
static class TestDescription_PuntoTestSuite_testConstructores :
       public CxxTest::RealTestDescription {
public:
 TestDescription_PuntoTestSuite_testConstructores():CxxTest::RealTestDescription(
     Tests_PuntoTestSuite, suiteDescription_PuntoTestSuite,
         36, "testConstructores" ) {}

 void runTest() { suite_PuntoTestSuite.testConstructores(); }
} testDescription_PuntoTestSuite_testConstructores;

static class TestDescription_PuntoTestSuite_testAccesores :
       public CxxTest::RealTestDescription {
public:
 TestDescription_PuntoTestSuite_testAccesores():CxxTest::RealTestDescription(
     Tests_PuntoTestSuite, suiteDescription_PuntoTestSuite,
         50, "testAccesores" ) {}

 void runTest() { suite_PuntoTestSuite.testAccesores(); }
} testDescription_PuntoTestSuite_testAccesores;
...
```

## 20. Caso de uso: boost::test (I).

- Es una de las bibliotecas de Boost.
- Su página
- Puede convertirse en el marco de tests estandar para C++.

## 21. Caso de uso: boost::test (II).

```
#define BOOST_TEST_MODULE poo-p1 test
#include <boost/test/unit_test.hpp>
#include <boost/test/output_test_stream.hpp>

#include <sstream>
#include <Punto.h>
#include <Posicion.h>
#include <Aplicacion.h>
using boost::test_tools::output_test_stream;
BOOST_AUTO_TEST_SUITE ( punto_ts )    // Inicio del test suite

struct F {
  F() {
    BOOST_TEST_MESSAGE( " setup fixture" );
    p0 = new Punto;
    p1 = new Punto(*p0);
  }

  ~F() {
```

```
20      delete p0; p0 = NULL;
21      delete p1; p1 = NULL;
22      BOOST_TEST_MESSAGE( " teardown fixture" );
23    }
24
25    Punto* p0;  Punto* p1;
26  };
```

## 22. Caso de uso: boost::test (III).

```
1   BOOST_FIXTURE_TEST_CASE( constructores_fxt, F )
2   {
3                                           // 1 //
4     BOOST_CHECK_EQUAL ( p0->getX (), 0.0 );
5     BOOST_CHECK_EQUAL ( p0->getY (), 0.0 );
6
7                                           // 2 //
8     BOOST_CHECK_EQUAL ( p1->getX (), 0.0 );
9     BOOST_CHECK_EQUAL ( p1->getY (), 0.0 );
10
11    p0->setX (2.3);
12    BOOST_CHECK_EQUAL ( p0->getX (), 2.3 );
13    p0->setY (4.2);
14    BOOST_CHECK_EQUAL ( p0->getY (), 4.2 );
15  }
```

## 23. Caso de uso: boost::test (IV).

```
1   BOOST_AUTO_TEST_CASE( constructores )
2   {
3     Punto p0;                             // 1 //
4     BOOST_CHECK_EQUAL ( p0.getX (), 0.0 );
5     BOOST_CHECK_EQUAL ( p0.getY (), 0.0 );
6
7     Punto p1 (p0);                        // 2 //
8     BOOST_CHECK_EQUAL ( p1.getX (), 0.0 );
9     BOOST_CHECK_EQUAL ( p1.getY (), 0.0 );
10
11    p0.setX (2.3);
12    BOOST_CHECK_EQUAL ( p0.getX (), 2.3 );
13    p0.setY (4.2);
14    BOOST_CHECK_EQUAL ( p0.getY (), 4.2 );
15  }
16  BOOST_AUTO_TEST_SUITE_END ()    // Fin del test suite
```

## 24. Caso de uso: boost::test (V).

- Además de las cabeceras que necesitamos incluir para compilar los tests, debemos enlazar con la biblioteca:

```
-lboost_test_exec_monitor
```

- Los *test runners* creados se pueden ejecutar con la opción `--help`, p.e.:

```
./poo-p1-test --help
```

- En el ejemplo que vamos a ver se ejecutan así:

```
./poo-p1-test --log_level=test_suite [--run_test=punto_ts]
```

## 25. Caso de uso: GLib.Test (I).

- La biblioteca GLib ( escrita y pensada para desarrollar en "C" ) incorpora un mecanismo de paso de tests: GLib.Testing.
- Es posible que una adaptación de GLib a otro lenguaje de programación nos de acceso al uso de ese marco de tests, p.e. es el caso de Vala.
- Nos permite crear casos de test individuales y test suites a los que ir incoporando estos tests.
- Veamos un ejemplo sencillo haciendo uso de Vala. Creamos una calculadora de juguete.

## 26. Caso de uso: GLib.Test (II).

```
/*
 * Clase que representa la calculadora.
 */

public errordomain CalculatorError {
        INVALID_OPERATION,
        INVALID_EXPRESSION
}

public class Calculator : Object {
       public static int plus (int a, int b) {
                return a + b;
        }

        public static int minus (int a, int b) {
                return a - b;
        }

        public static int multiply (int a, int b) {
                return a * b;
        }
```

## 27. Caso de uso: GLib.Test (III).

```
public static int evaluate (string input) throws CalculatorError {
  try {
    var regex = new Regex ("(\\d+)\\W*([\\+\\-\\*])\\W*(\\d+)");
    MatchInfo match;
    if (!regex.match (input, 0, out match)) {
          throw new CalculatorError.INVALID_EXPRESSION
                                    ( "Invalid expression: %s", input );
    }
    var arg1 = int.parse(match.fetch (1));
    var op = match.fetch (2);
    var arg2 = int.parse(match.fetch (3));

    switch (op) {
        case "+":
          return plus (arg1, arg2);
        case "-":
          return minus (arg1, arg2);
        case "*":
          return multiply (arg1, arg2);
        default:
          throw new CalculatorError.INVALID_OPERATION ("Invalid operation %s", op);
    }
  } catch (RegexError e) { error(e.message); }
 }
}
```

## 28. Caso de uso: GLib.Test (IV).

```
/*
 * Un programa de prueba.
 */
 public static void main () {
    while (true) {
         string? expression = stdin.read_line ();
         if (expression != null) {
           try {
              int value = Calculator.evaluate (expression);
              stdout.printf (" = %d", value);
           } catch (CalculatorError e) {
              stdout.printf ("%s", e.message);
           }
         }
    }
}
```

## 29. Caso de uso: GLib.Test (V).

```
/*
 * Paso de los tests
```

```
 */
 public class TestCalculator : Object {
   public static void test_add () {
     assert (2 == Calculator.plus (1, 1));
     assert (8 == Calculator.plus (7, 1));
   }

   public static void test_minus () {
     assert (3 == Calculator.minus (3, 1));
     assert (-6 == Calculator.minus (1, 7));
   }

   public static void test_multiply () {
     stdout.printf ("Inside fixture ");
     assert (1 == Calculator.multiply (1, 1));
     assert (84 == Calculator.multiply (7, 12));
     assert (0 == Calculator.multiply (0, 12));
   }
}
```

## 30. Caso de uso: GLib.Test (VI).

```
static void fixture_setup () { stdout.printf ("FIXture setup"); }
static void fixture_teardown () { stdout.printf ("FIXture teardown");}

public static void main (string[] args) {
  Test.init (ref args);

  TestCase tc = new TestCase ("TC1", fixture_setup, test_multiply, fixture_teardown);
  TestSuite.get_root().add(tc);

  Test.add_func ("/calculator/add", test_add);
  Test.add_func ("/calculator/minus", test_minus);
  Test.add_func ("/calculator/multiply", test_multiply);

  Test.run ();
 }
```

## 31. Caso de uso: GLib.Test (VII).

```
$ test/calculator-test
/TC1: FIXture setup
Inside fixture
FIXture teardown
OK
/calculator/add: OK
/calculator/minus: **
ERROR:calculator_test.c:71:test_calculator_test_minus: assertion failed:
      (3 == Calculator.minus (3, 1))
Abortado
```

## 32. Integracion de paso de tests con Autotools / Cmake.

- Ambos entornos de configuración facilitan el paso de tests al software que construyen.
- **CMake**: Mediante una herramienta adicional (CTest) que se integra con CMake. Añadimos la llamada a `enable_testing()` en el `CMakeLists.txt` principal y aquellos ejecutables que sean un *test* se le indican a **cmake** mediante `add_test(...)`.
- **Autotools**: Mediante la introducción de un "`primary`" nuevo llamado "`check`", de manera que disponemos de una variable nueva llamada "`check_PROGRAMS`".
- Veamos cada uno de ellos con un ejemplo.

## 33. Prácticas.

EN GRUPO:

- Investiga qué es CDash, ¿cómo lo integrarías con CTest?
- Investiga qué es Yarn, explicadnos qué problema trata de resolver y probadlo con un ejemplo vuestro.
- Echa un vistazo a Cutter, explicadnos cómo funciona con un ejemplo sencillo.
- Cread un ejemplo en C++ que trabaje con cppunit.

INDIVIDUAL:

- Tomando como punto de partida los ejemplos de la calculadora y la vivienda domótica, aplica a otro código tuyo (C, C++) el paso de tests automático con cxxtest. Intégralo con *autoconf* + *automake* o con *cmake* (**make test**). Estaría bien que diseñaras tests de *caja negra* y de *caja blanca*.
- Tomando como punto de partida los ejemplos de la calculadora y la vivienda domótica, aplica a otro código tuyo (C, C++, Vala) el paso de tests automático que quieras (GLib.Test, boost::testing) integrado con *autoconf* + *automake* o con *cmake* (**make test**). Estaría bien que diseñaras tests de *caja negra* y de *caja blanca*.

ENTREGA:

- La práctica se entregará en pracdlsi en las fechas allí indicadas.

## 34. Aclaraciones

EN NINGÚN CASO ESTAS TRANSPARENCIAS SON LA BIBLIOGRAFÍA DE LA ASIGNATURA.

- Debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.