Tema 12: Pruebas unitarias.

¿Tan importante es pasar tests a nuestro código?

Unos consejos útiles:

- Ten siempre este libro a mano: The Pragmatic Programmer ya que está lleno de buenos consejos:

> "Test Your Software, or Your Users Will. Test ruthlessly. Don't make your users find bugs for you."

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

Pero...¿esto se usa en proyectos reales?

- Sí, y mucho...juzga tu mismo/a:

> As of version 3.16.2 (2016-01-06), the SQLite library consists of approximately 122.9 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 745 times as much test code and test scripts - 91596.1 KSLOC.

¿Para qué sirve el paso de tests?

- 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.

para todas aquellas entradas que tengan una probabilidad razonable de hacer que falle (si es que tiene un fallo).

- 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 única manera que tenemos de asegurar que nuestro software no tiene fallos es probarlo con todas las posibles conjuntos de entrada que pueda tener.
- 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.

Conjuntos de entradas: un ejemplo.

- Queremos testear la función: `bool es_mayor (int x, int y) .`
- Una posible partición del conjunto de datos de entrada podría ser:
  - **x** *positivo*, **y** *positivo*
  - **x** *negativo*, **y** *negativo*
  - **x** *positivo*, **y** *negativo*
  - **x** *negativo*, **y** *positivo*
  - `x == 0, y == 0`
  - `x == 0, y != 0`
  - `x != 0, y == 0`
- Si ahora comprobamos la función anterior con una entrada de, al menos, uno de estos subconjuntos tendremos una probabilidad razonable de que *si hay un fallo* entonces aparecerá...

Pero no una certeza completa!

- Se crean sin mirar (*sin conocer*) 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.

- 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.

¿Qué testear?

En base a la granularidad de lo que comprueban podemos hablar de:

Tests unitarios: : Están destinados a módulos individuales, p.e. clases o funciones.

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.

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!.

tests minuciosos. .

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

Cúando testear.

- Nunca dejarlo para el final, menos aún cerca de un deadline.
- Debería hacerse automáticamente...incluída 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.
- Una cosa más: Si alguien detecta un fallo...debería ser la última persona que lo detecta ... inmediatamente deberíamos tener un test para capturarlo.

Cobertura de un test.

- Se trata de una medida empleada con tests tipo *WhiteBox*.
- Trata de estimar cuánta funcionalidad del código 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).

Test drivers. Conducting tests.

- Hoy en día los procesos de testeo están *automatizados*...
- Los tests se pasan empleando *test drivers*, los cuales:

4. Comprueban la validez de estos resultados.
5. Preparan el informe apropiado.

Test drivers: xUnit.

- 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, cxxtest), "D" (DUnit), etc...

xUnit: Conceptos básicos.

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.

Caso de uso: cxxtest.

- 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.

- En el *makefile* correspondiente pondríamos una regla como esta:

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

- Y el archivo de cabecera *poo-p1-test.h* podría ser algo así:

Entrada para cxxtest.

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 () );
    }
```

P2 GIR

En esta página › Sinopsis

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();  } // Main program
#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 );
```

```
...
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 {
```

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

Test drivers: boost::test.

- Es una de las bibliotecas de Boost.
- Aquí tienes su documentación: boost::test
- En un futuro podría ser el marco de paso de tests estandar para C++.
- Veamos un ejemplo de uso.

Entrada para boost::test.

```
#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() {
    delete p0; p0 = NULL;
```

```
  Punto* p0;  Punto* p1;
};
```

```
BOOST_FIXTURE_TEST_CASE( constructores_fxt, F )
{
                                // 1 //
  BOOST_CHECK_EQUAL ( p0->getX (), 0.0 );
  BOOST_CHECK_EQUAL ( p0->getY (), 0.0 );

                                // 2 //
  BOOST_CHECK_EQUAL ( p1->getX (), 0.0 );
  BOOST_CHECK_EQUAL ( p1->getY (), 0.0 );

  p0->setX (2.3);
  BOOST_CHECK_EQUAL ( p0->getX (), 2.3 );
  p0->setY (4.2);
  BOOST_CHECK_EQUAL ( p0->getY (), 4.2 );
}
```

```
BOOST_AUTO_TEST_CASE( constructores )
{
  Punto p0;                     // 1 //
  BOOST_CHECK_EQUAL ( p0.getX (), 0.0 );
  BOOST_CHECK_EQUAL ( p0.getY (), 0.0 );

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

  p0.setX (2.3);
  BOOST_CHECK_EQUAL ( p0.getX (), 2.3 );
  p0.setY (4.2);
  BOOST_CHECK_EQUAL ( p0.getY (), 4.2 );
}
BOOST_AUTO_TEST_SUITE_END ()    // Fin del test suite
```

- Mediante una herramienta adicional, CTest, la cual se integra con CMake, podemos automatizar el paso de tests a nuestro software. Es muy sencillo.
- Para ello añadimos la llamada a `enable_testing()` en el archivo *CMakeLists.txt* del directorio donde vayamos a generar los tests y aquellos ejecutables que sean un test se le indican a **CMake** mediante la orden `add_test(...)` .
- Ten en cuenta que el ejecutable de un test lo tendrás que generar primero usando `add_executable (...)` .
- Veamos un ejemplo de uso en la práctica.

Aclaraciones.

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto 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.