Testing

Algoritmos y Estructuras de Datos II

Testing

Supongamos que programamos la clase `Fecha` y los métodos:

```
void Fecha::sumar_meses(int meses) { ...  }
void Fecha::sumar_dias(int dias) { ...  }
```

¿Cómo verificamos si cumple con la especificación?

Testing

¿Cómo verificamos si un programa cumple con la especificación?

Problema difícil

- No es razonable probar con todas las entradas. Entre el año 1 y el 2100 hay ∼750.000 fechas posibles, y el parámetro `dias` es un `int` que típicamente podría tomar 4.294.967.296 valores posibles.
- Aun suponiendo que contáramos con una especificación formal, no hay un método automático para comprobar que un programa cumple con esa especificación (Tema de LyC).

Testing

¿Cómo verificamos si un programa cumple con la especificación?

Una solución (parcial)

Testing: probar que el programa funcione en varios casos.

> *Program testing can be used to show the presence of bugs, but never to show their absence. —E. W. Dijkstra*

Es decir:

- El testing sirve para encontrar errores.
  "*no pasa los tests* $\Rightarrow$ *hay errores*"
- El testing no sirve para asegurar que no hay errores.
  "*pasa los tests* $\not\Rightarrow$ *no hay errores*"

Nota: el testing es **muy** útil, a pesar de no ser una técnica exacta.

Testing

¿Cómo verificamos si un programa cumple con la especificación?

Opción 1: a ojo

```
int main() {
  Fecha f(2000, 1, 1);
  f.sumar_dias(10);
  cout << f << endl; // 2000-01-11
  f.sumar_meses(1);
  cout << f << endl; // 2000-02-11
  f.sumar_dias(20);
  cout << f << endl; // 2000-03-03
  f.sumar_meses(2);
  cout << f << endl; // 2000-05-03
  return 0;
}
```

Opción 1: a ojo — desventajas

- Si tenemos 100 tests, tenemos que mirar 100 líneas y calcular a ojo si son correctas. Los humanos somos pésimos para esto.
- La condición que verificamos no queda documentada.
- Si cambiamos alguna parte del programa, corremos el riesgo de introducir un bug y tenemos que repetir este proceso.
- Los tests del ejemplo están "enredados": si hay un error, puede ser difícil encontrar de dónde proviene.
- Idealmente, cada test debería comprobar una funcionalidad puntual para que sea más fácil encontrar la causa del bug.

Testing

Opción 2: testing con código

```
bool test_sumar_dia_sin_cambio_mes() {
  Fecha f(2000, 4, 20);
  f.sumar_dias(5);
  if (f != Fecha(2000, 4, 25)) {
    cout << "Error al sumar días (sin cambio de mes)." << endl;
    return false;
  }
  return true;
} ...
bool test_sumar_dia() {
  return test_sumar_dia_sin_cambio_mes() &&
         test_sumar_dia_con_cambio_mes() && ...;
} ...
int main() {
  if (test_sumar_dia() && test_sumar_mes() && ...) {
    return 0;
  } else { return -1; }
}
```

Opción 3: usando un entorno de testing

Por ejemplo, con google-test:

```
#include "gtest/gtest.h"
#include "../src/Fecha.h"

TEST(sumar_dia, sin_cambio_mes) {
  Fecha f(2000, 4, 20);
  f.sumar_dias(5);
  EXPECT_EQ(f, Fecha(2000, 4, 25));
}

TEST(sumar_dia, con_cambio_mes) {
  Fecha f(2000, 4, 20);
  f.sumar_dias(19);
  EXPECT_EQ(f, Fecha(2000, 5, 9));
}
...
```

Opción 3: usando un entorno de testing

- El *framework* incorpora un `main` que corre todos los tests.
- Reporta cuáles fueron los tests que pasaron exitosamente y cuáles fallaron.
- Aserciones útiles en `google-test`:
  - `EXPECT_TRUE(x)`, `EXPECT_FALSE(x)`
  - `EXPECT_EQ(x, y)`, `EXPECT_NE(x, y)`, `EXPECT_LT(x, y)`, `EXPECT_LE(x, y)`, `EXPECT_GT(x, y)`, `EXPECT_GE(x, y)`
- Todos los lenguajes de programación populares cuentan con algún *framework* de testing.

Algunas ideas

- Al menos un test por función / método
- Tests por comportamiento específico
- Al testear colecciones, pensar en los casos: vacío, un elemento, muchos elementos.

Esquema

```
TEST(...) {
    // Preparar valor calculado
    ...
    T valor_calculado = ... ;

    // Preparar valor esperado
    ...
    T valor_esperado = ... ;

    // Ver que se cumple condición
    EXPECT_TRUE(valor_calculado == valor_esperado);
}
```

Ejemplos

A veces los valores se calculan en una línea;

```
TEST(Suma, simple) {
    int valor_calculado = 2 + 3;

    int valor_esperado = 5;

    EXPECT_EQ(valor_calculado, valor_esperado);
}
```

Ejemplos

A veces requieren varias líneas.

```
TEST(Libreta, LU) {
    LU lu = "123/45";
    Libreta l(lu);
    LU valor_calculado = l.lu();

    LU valor_esperado = lu;

    EXPECT_EQ(valor_calculado, valor_esperado);
}
```

Ejemplos

A veces es una transformación de estados.

```
TEST(Libreta, LU) {
    LU lu = "123/45";
    Libreta l(lu);
    l.aprobar_practico("Algo2");
    int valor_calculado =
    ↪  l.practicos_aprobados().count("Algo2");

    int valor_esperado = 1;

    EXPECT_EQ(valor_calculado, valor_esperado);
}
```

Ejemplos

A veces, y con recaudos, evaluamos comportamientos asociados.

```
TEST(Libreta, LU) {
    LU lu = "123/45";
    Libreta l(lu);
    l.aprobar_final("Algo2", 8);
    EXPECT_EQ(l.practicos_aprobados().count("Algo2"), 1);
    EXPECT_EQ(l.finales_aprobados().count("Algo2"), 1);
    EXPECT_EQ(l.nota_final("Algo2"), 8);
}
```