Testing y debugging

Algoritmos y Estructuras de Datos II

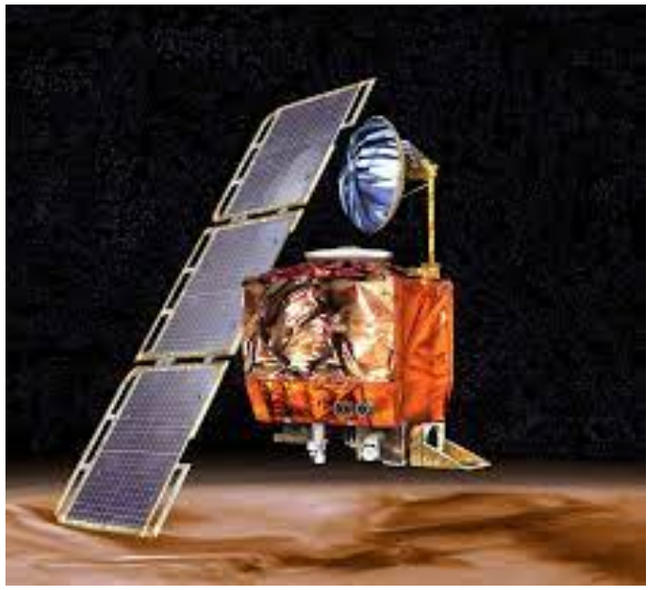
Año 2025

Dr. Pablo Ponzio

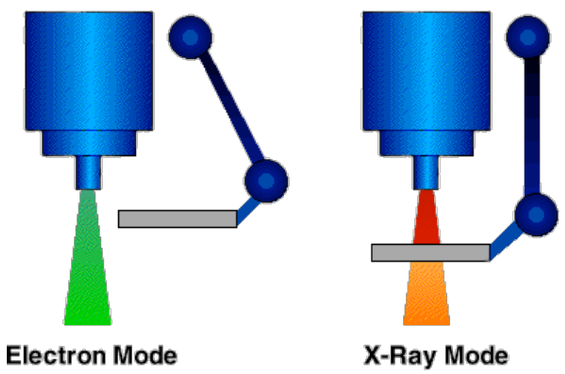
Universidad Nacional de Río Cuarto

CONICET

Fallas espectaculares de software

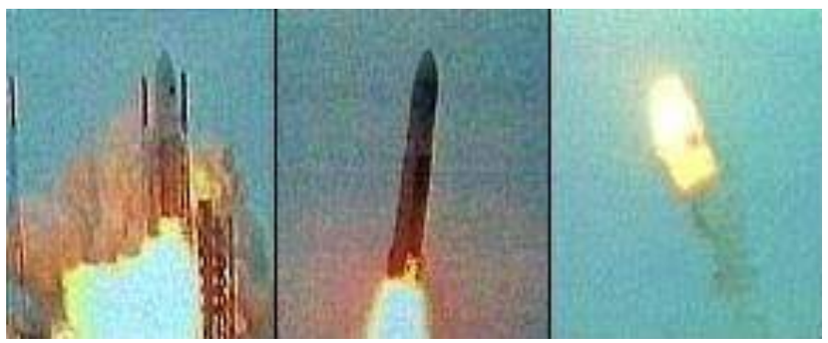


- NASA’s Mars lander: se estrelló por una falla de integración en módulos que manejaban los propulsores (1999)
  - Un módulo computaba datos en sistema imperial y se los proveía a otro que los esperaba en sistema métrico



- Máquina de radiación THERAC-25: al menos 3 muertos y varios más heridos por radiación excesiva

Fallas espectaculares de software



- Explosión del Ariane 5: reuso del sistema de navegación del Ariane 4 produjo una falla
  - Conversión de un real de 64 a 16-bit

- Falla del FDIV en Intel Pentium: Últimas 5 entradas de una tabla no se cargaron por una falla de software
  - Sólo 1 en aprox. 9.000 millones de operaciones eran incorrectas

Fallas espectaculares de software

- Boeing 737 Max: dos aviones (con pasajeros) se estrellaron por problema en software de control de vuelo

- Aceleración repentina en autos Toyota: decenas de muertos, cientos de accidentes

- Apagón del noreste: falla en sistema de alarmas provocó sobrecarga en el sistema
  - Afectó a 50 millones de personas

¿Por qué hacer testing?

- En la actualidad, el testing es la metodología más usada para evaluar y mejorar la confiabilidad del software
- El testing consiste en ejecutar el software para algunas entradas, y observar el resultado de las ejecuciones
  - Si el comportamiento de estas ejecuciones respeta la especificación decimos que los tests son exitosos
  - Si ejecución viola la especificación, hemos encontrado una falla y debemos repararla
- El testing sirve para detectar fallas en etapas tempranas del desarrollo
  - Reparar fallas en etapas tempranas es más fácil y menos costoso que en etapas avanzadas

¿Por qué hacer testing?

- Así, el testing es clave para tener un buen grado de confianza de que el programa se comporta de acuerdo a su especificación
  - Y así evitar que los usuarios tengan que lidiar con errores indeseados
- El testing tiene un rol central en las metodologías modernas de desarrollo (metodologías ágiles)
  - En la actualidad, es una práctica estándar que los programadores escriban tests junto con el código del programa en desarrollo
  - Algunas metodologías incluso proponen que hay que escribir tests antes que el código (metodologías test first)

Testing manual

```
public static void main(String[] args) {
    TicketMachine machine = new TicketMachine(10);
    machine.insertMoney(10);
    boolean res = machine.printTicket();
    System.out.println("Result: " + res);
    System.out.println("Remaining money: " + machine.getBalance());
}
```

- Todo programador está habituado al testing manual:
  - Ejecutamos el programa para algunas entradas y observamos el comportamiento del mismo
    - Por ejemplo, con un debugger, o con "prints" en el código

Testing manual

```
public static void main(String[] args) {
    TicketMachine machine = new TicketMachine(10);
    machine.insertMoney(10);
    boolean res = machine.printTicket();
    System.out.println("Result: " + res);
    System.out.println("Remaining money: " + machine.getBalance());
}
```

- Esta forma de hacer testing es poco efectiva, por varios motivos:
  - Ejecutar tests manualmente consume mucho tiempo, que podría aprovecharse mejor en otras tareas
  - Los tests no se guardan, y hay que repetirlos cada vez que modificamos o agregamos funcionalidades al programa
  - Usualmente, sólo es posible testear de esta manera unos pocos comportamientos del programa, dejando muchas partes del mismo sin testear

Automatización de los tests

- Para resolver los problemas del testing manual, vamos a automatizar la ejecución de los tests
- Escribiremos "scripts" de test que ejecuten y evalúen los resultados de los tests automáticamente
    - Y usaremos herramientas que reporten los resultados de manera amigable
- La automatización de los tests tiene muchas ventajas:
    - Reduce sustancialmente el costo del testing
    - Reduce el error humano durante el testing
    - Posibilita el testing de regresión: Nuevas versiones del software deben preservar el comportamiento previo (no modificado)
        - Es decir, las nuevas versiones deben pasar los tests existentes
        - Se denomina regresión a una nueva versión del software que introduce una falla que no existía en la versión previa
- Si tenemos buenos tests, se incrementa la confianza de que el software se comporta de acuerdo a sus especificaciones

Testing unitario

- Lo primero que tenemos que hacer es elegir el artefacto que queremos testear
  - Inicialmente, vamos a testear métodos individuales de las clases
    - Estos se denominan tests unitarios
  - Más adelante vamos a testear interacciones entre métodos de diferentes clases/ módulos
    - Los tests que verifican que los módulos interactúan correctamente se denominan tests de integración
- Cada test unitario para un método consiste en ejecutar el método bajo test en un escenario particular
  - Es decir, para un conjunto único de entradas
- Todo test unitario tiene tres componentes:
  - Arrange: Preparar las entradas necesarias para ejecutar el método bajo test
  - Act: Ejecutar el método bajo test con las entradas seleccionadas
  - Assert: Verificar que los resultados del test cumplen con la especificación del método

Tests para el constructor de TicketMachine

```
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *   with a price of 'cost'.
*/
public TicketMachine(int cost)
```

- Un test para el constructor consiste en:
  - Arrange: Elegir un valor para `cost`
  - Act: Ejecutar el constructor
  - Assert: Verificar que el estado del objeto creado satisface `post`
    - La aserción debe verificar el comportamiento descrito en la especificación

Tests para el constructor de TicketMachine

```
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *   with a price of 'cost'.
*/
public TicketMachine(int cost)
```

```
@Test
public void testConstructorValidPrice()
{
    int price = 10;
    TicketMachine machine = new TicketMachine(price);
    assertEquals(10, machine.getPrice());
}
```

- Un test para el constructor consiste en:
  - Arrange: Elegir un valor para `cost`
  - Act: Ejecutar el constructor
  - Assert: Verificar que el estado del objeto creado satisface `post`
    - La aserción debe verificar el comportamiento descrito en la especificación

Tests para el constructor de TicketMachine

```java
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *   with a price of 'cost'.
*/
public TicketMachine(int cost)
```

```java
@Test
public void testConstructorValidPrice()
{
    int price = 10;
    TicketMachine machine = new TicketMachine(price);
    assertEquals(10, machine.getPrice());
}
```

arrange → `int price = 10;`

act → `TicketMachine machine = new TicketMachine(price);`

assert → `assertEquals(10, machine.getPrice());`

- Un test para el constructor consiste en:
  - Arrange: Elegir un valor para `cost`
  - Act: Ejecutar el constructor
  - Assert: Verificar que el estado del objeto creado satisface `post`
    - La aserción debe verificar el comportamiento descrito en la especificación

Tests unitarios en JUnit

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class TicketMachineTest
{
    @Test
    public void testConstructorValidPrice()
    {
        int price = 10;
        TicketMachine machine = new TicketMachine(price);
        assertEquals(10, machine.getPrice());
    }

    // código omitido ...
```

- Vamos a usar el framework de testing para Java `JUnit`
  - Uno de los frameworks más usados para Java
- Los tests se definen en una clase, que usualmente tiene el mismo nombre de la clase bajo test seguida por el sufijo `Test`
  - Ej.: `TicketMachineTest` contiene los tests para `TicketMachine`
- Cada método anotado con `@Test` es considerado un test en `JUnit`

Tests unitarios en JUnit

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class TicketMachineTest
{
    @Test
    public void testConstructorValidPrice()
    {
        int price = 10;
        TicketMachine machine = new TicketMachine(price);
        assertEquals(10, machine.getPrice());
    }

    // código omitido ...
```

- Las primeras dos líneas (`import`) son para importar las bibliotecas necesarias de `JUnit`
- `assertEquals` es un método de `JUnit` que verifica automáticamente si se cumple la aserción cuando se ejecuta el test
  - Y reporta el error al programador si no se cumple
- `JUnit` tiene muchas otras aserciones, por el momento usaremos `assertEquals`, `assertTrue` y `assertFalse`

Ejecución de los tests con gradle

- JUnit permite ejecutar los tests de distintas formas: usando gradle, desde un IDE, desde línea de comandos, etc.
- Para usar gradle, debemos respetar la siguiente estructura de directorios:
  - `src/main/java` contiene el código fuente
  - `src/test/java` contiene los tests
- La figura muestra la estructura de directorios de nuestro ejemplo:

```
src
├── main
│   └── java
│       ├── TicketMachine.java
│       └── TicketMachineMain.java
└── test
    └── java
        └── TicketMachineTest.java
```

- Luego, podemos ejecutar los tests con el comando `./gradlew test`

```
> Task :test

TicketMachineTest > testConstructorValidPrice() PASSED
```

- En este caso, el reporte de gradle muestra que la ejecución del test ha sido exitosa

Test fallidos

```java
/**
 * @pre 'cost' > 0.
 * @post Create a machine that issues tickets
 *   with a price of 'cost'.
*/
public TicketMachine(int cost)
{
    price = 0;
    balance = 0;
    total = 0;
}
```

```
> Task :test FAILED

TicketMachineTest > testConstructorValidPrice() FAILED
    org.opentest4j.AssertionFailedError at TicketMachineTest.java:13
```

- Si nuestro código tiene errores como en la figura, el reporte indicará que hay tests que fallan
  - Si los tests fallan, hay que corregir el código antes de continuar
  - El reporte automático de errores es posible porque usamos los métodos de `JUnit` para verificar las aserciones
- Si nuestro código es correcto, los tests deben ser exitosos (si no hay errores en los tests)

Defectos y fallas

```java
@Test
public void testConstructorValidPrice()
{
    int price = 10;
    TicketMachine machine = new TicketMachine(price);
    assertEquals(10, machine.getPrice());
}
```

```
> Task :test FAILED

TicketMachineTest > testConstructorValidPrice() FAILED
    org.opentest4j.AssertionFailedError at TicketMachineTest.java:13
```

- Falla (failure): Ejecución del software que es incorrecta respecto de la especificación
  - Las fallas son un concepto dinámico (relacionado a la ejecución)
- El objetivo del testing es buscar fallas en el software

Defectos y fallas

```
public TicketMachine(int cost)
{
    price = 0;
    balance = 0;
    total = 0;
}
```

- Defecto (fault): Líneas de código fuente incorrectas
  - Siempre que hay una falla, existe un defecto que la provoca
  - Usualmente, se deben a errores involuntarios de programadores durante el desarrollo
  - Los defectos son un concepto estático (relacionado al código fuente)
  - Dada una falla, el debugging consiste en buscar el defecto que la causa
- Bug: Término impreciso que se usa para referirse a cualquiera de estos dos conceptos
  - Por lo que evitaremos usarlo

Tests para printTicket

```
/**
 * @post If enough money has been inserted, print a ticket to the console
 * and reduce the current balance by the ticket price. Returns 'true' if
 * successful; otherwise, it does nothing and returns 'false'.
 */
public boolean printTicket()
```

- Mirando la especificación de `printTicket`, sería deseable testear el método en dos escenarios: uno en el que la máquina imprima un ticket y otro en el que no
- Para el primer escenario, debemos crear una máquina que tenga suficiente dinero, y esperamos que el método retorne true como resultado, y que el balance se reduzca
- Para el segundo escenario, tenemos que crear una máquina que no tenga suficiente dinero, y esperamos que el método retorne false y el balance no cambie

Tests para printTicket

```
/**
 * @post If enough money has been inserted, print a ticket to the console
 * and reduce the current balance by the ticket price. Returns 'true' if
 * successful; otherwise, it does nothing and returns 'false'.
 */
public boolean printTicket()
```

- Notar que es difícil de testear lo que se imprime por pantalla
  - Hay herramientas más avanzadas para hacerlo, pero exceden los límites de este curso
- Sin embargo, la dificultad para testear un método en muchos casos indica un problema de diseño, y que deberíamos mejorar la abstracción de datos que estamos testeando
  - En este caso, lo ideal sería hacer que el método retorne un objeto `Ticket`, y así podríamos verificar el estado del objeto en el test
- Por el momento, nos alcanza con testear el valor retornado por el método

Tests para printTicket

- Un test para el primer escenario es el siguiente:

```
@Test
public void testPrintTicketOk()
{
    TicketMachine machine = new TicketMachine(10);
    machine.insertMoney(10);
    boolean res = machine.printTicket();
    assertTrue(res);
    assertEquals(0, machine.getBalance());
}
```

Tests para printTicket

- Un test para el primer escenario es el siguiente:

```
@Test
public void testPrintTicketOk()
{
    TicketMachine machine = new TicketMachine(10);
    machine.insertMoney(10);
    boolean res = machine.printTicket();
    assertTrue(res);
    assertEquals(0, machine.getBalance());
}
```

Arrange: Entradas para el método a testear

Act: Ejecución del método bajo test para las entradas elegidas

Assert: Verifica que el resultado de la ejecución del método bajo test es el esperado

Tests para printTicket

- Un test para el segundo escenario es el siguiente:

```
@Test
public void testPrintTicketFails()
{
    TicketMachine machine = new TicketMachine(10);
    boolean res = machine.printTicket();
    assertFalse(res);
    assertEquals(0, machine.getBalance());
}
```

Tests para printTicket

- Un test para el segundo escenario es el siguiente:

```
@Test
public void testPrintTicketFails()
{
    TicketMachine machine = new TicketMachine(10);
    boolean res = machine.printTicket();
    assertFalse(res);
    assertEquals(0, machine.getBalance());
}
```

Arrange: Entradas para el método a testear

Act: Ejecución del método bajo test para las entradas elegidas

Assert: Verifica que el resultado de la ejecución del método bajo test es el esperado

Características deseables de los tests unitarios

- Idealmente, los tests unitarios deben ser:
  - Ejecutables automáticamente
    - Con sólo presionar un botón
    - Por cualquier persona involucrada en el proyecto
  - Repetibles
    - Con resultados consistentes
- Fáciles de implementar y entender
  - Sin lógica compleja
  - Evitar condiciones, ciclos, etc...
- Rápidos y eficientes en el uso de recursos
- Independientes entre si
  - No podemos asumir nada sobre el orden de ejecución de los tests

¿Qué debemos testear?

- Idealmente, se deben testear todos los métodos públicos de las clases que implementemos
- En este curso, usaremos un criterio básico para decidir que tests crear, denominado testing de caja negra
  - Elegiremos escenarios relevantes para testear un método basándonos en la especificación
- Siempre debemos escribir aserciones en los tests en base al comportamiento esperado, es decir, usando la especificación
  - No se debe mirar el código para decidir que escribir en una aserción

¿Qué debemos testear?

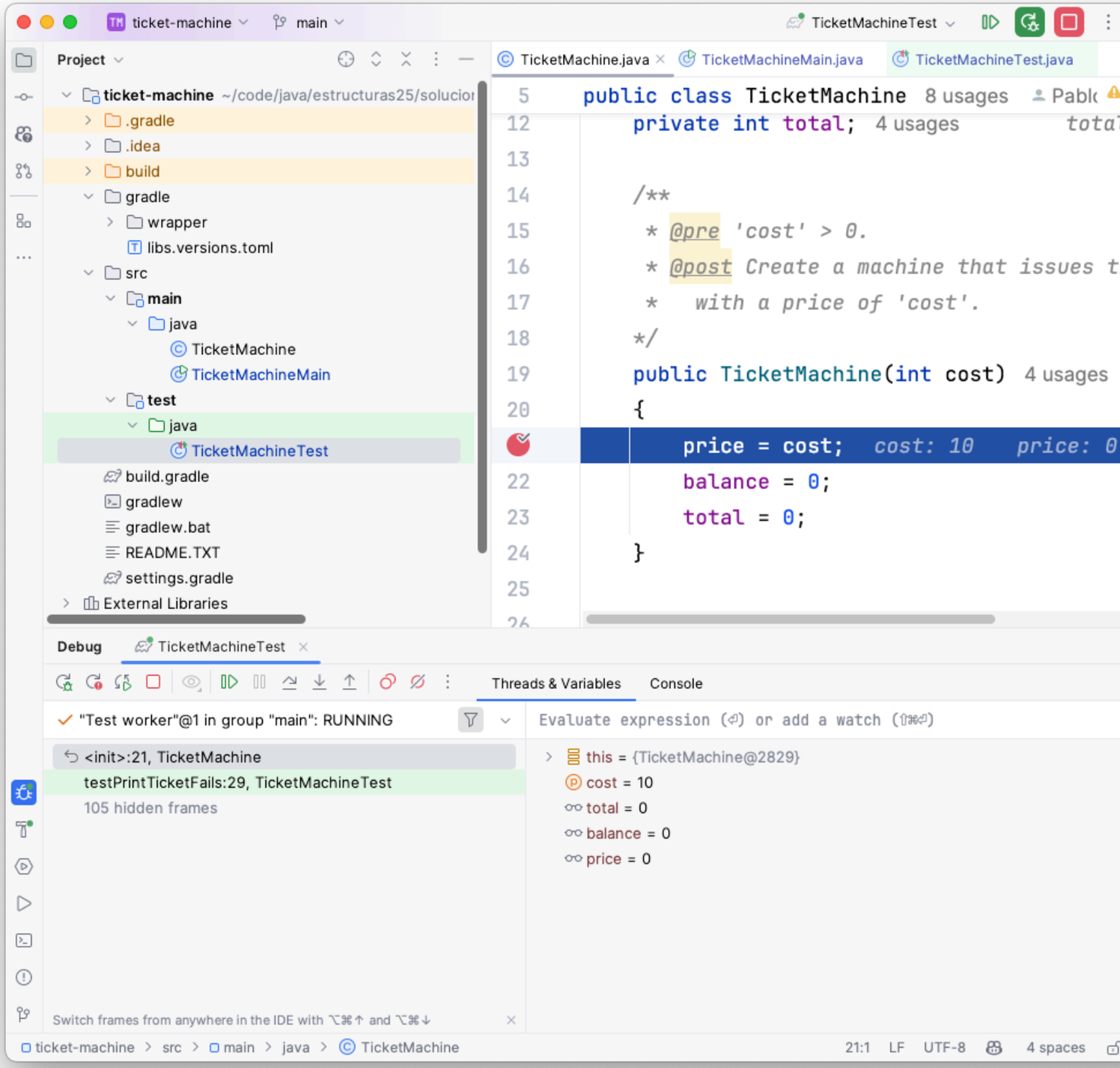
- Existen otros criterios de testing que tienen en cuenta el código fuente del método para decidir qué testear
  - Se denominan criterios de testing de caja blanca
  - Cobertura de código: Decimos que cubrimos una porción del código cuando al menos un test la ejecuta. Algunas variantes típicas son:
    - Cobertura de instrucciones: Todas las líneas del código deben ser ejecutadas (cubiertas) por al menos un test
    - Cobertura de ramas: Escribir tests para hacer cada condición del código true y false (de un if, while, etc..)
      - Este es el criterio de caja blanca más estándar, y posiblemente el más usado
      - Da lugar a mejores tests que cobertura de instrucciones
  - Son complementarios al testing de caja negra
- Típicamente, en proyectos reales se usan una combinación de criterios de caja negra y de caja blanca

Testing y debugging

- El testing y el debugging, si se realizan de manera apropiada, son actividades complementarias
- Primero se escriben tests, y se ejecutan para revelar fallas en el programa
- Luego, cuando sabemos de la existencia de una falla, el debugging consiste en buscar el defecto en el código que causó la falla, con el objetivo de reparar el defecto

Debuggers

- Los debuggers son herramientas de software que permiten examinar en detalle la ejecución de una aplicación, y sirven para asistir al proceso de debbuging
- Los debuggers típicamente permiten:
  - Detener la ejecución de la aplicación en una línea particular del código
  - Ejecutar el código línea por línea
  - Examinar el estado de las variables y los objetos
  - Etc...
- Se recomienda usar algún IDE de su preferencia y debuguear sus programas



Actividades

- Leer el capítulo 9 del libro "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016.

Bibliografía

- "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016
- "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001
- "Introduction to Software Testing" (2nd. edition). P. Ammann & J. Offutt. Cambridge University Press. 2016.