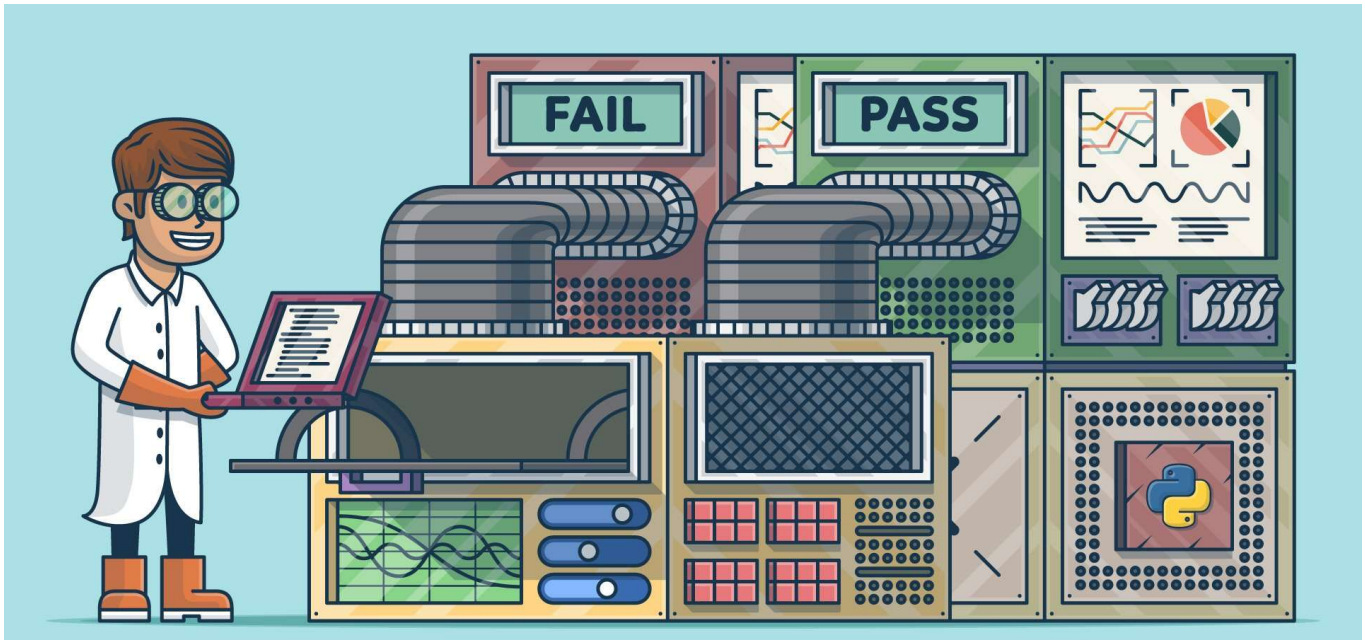
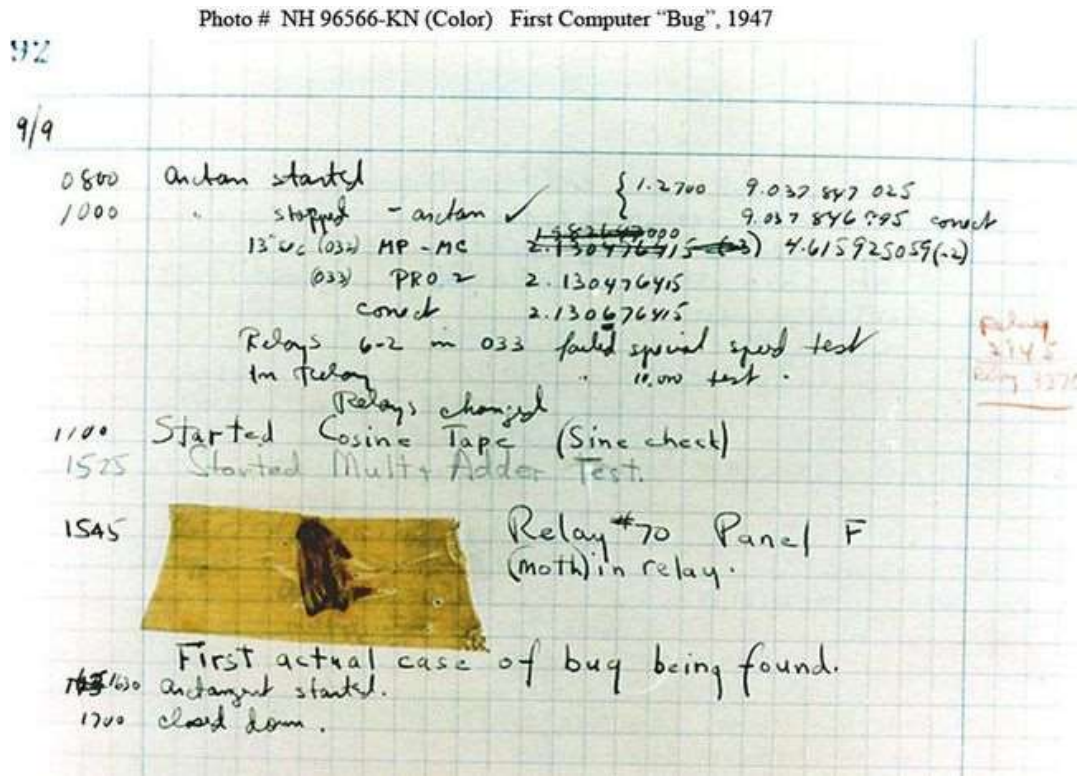
Testing



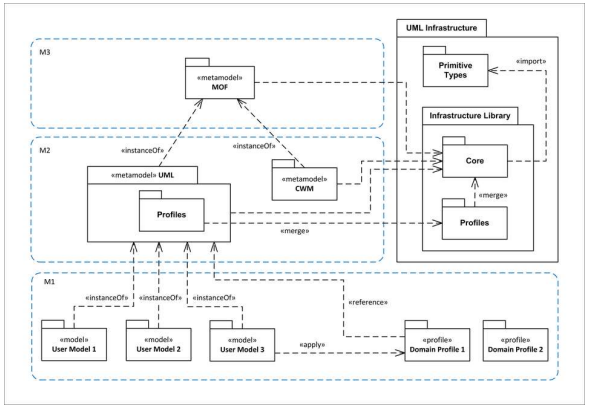
¿Qué es un bug/error?

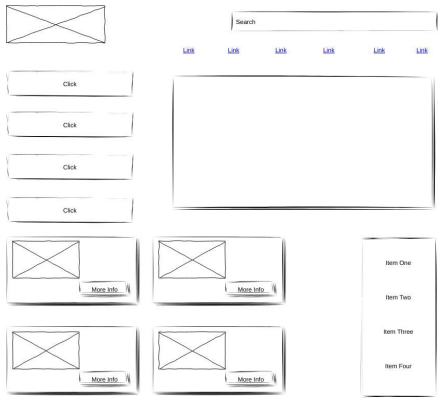


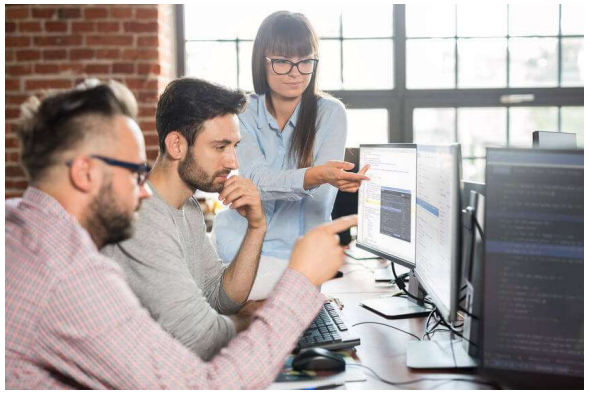
https://www.nationalgeographic.org/thisday/sep9/worlds-first-computer-bug/

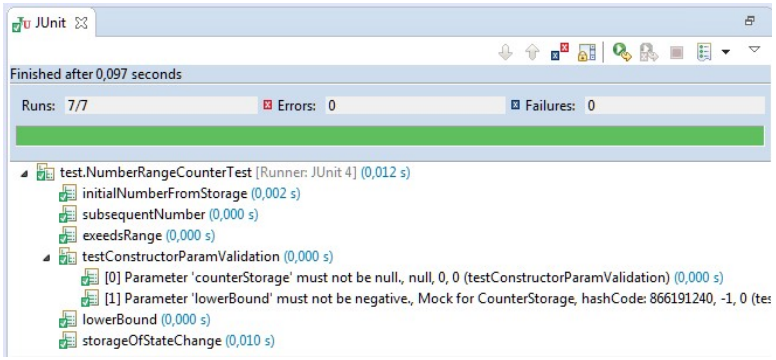
¿Qué es testear?

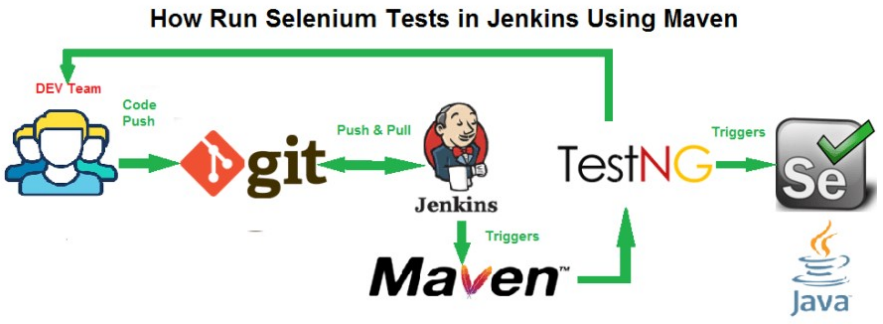
¿Para qué, con quien, cuándo, y como testear?











¿Por qué no testeamos (o lo hacemos mal)?

- Lo dejamos para el final (¿para no trabajar de gusto?)
- Hay muchas combinaciones que considerar
- Requiere planificación, preparación y recursos adicionales
- Es una tarea repetitiva, y nos parece poco interesante
- Creemos que es tarea de otro, nosotros programamos (¿?)
- Creemos que alcanza con “programar bien”
- El objetivo de testear es encontrar bugs (¿será que eso nos molesta?)

Tipos de test

- Tests funcionales
- Test no funcionales
- Tests de unidad
- Tests de integración
- Tests de regresión
- Test punta a punta
- Tests automatizados

- Test de carga
- Test de performance
- Test de aceptación
- Test de UI
- Test de accesibilidad
- Alpha y beta tests
- Test A/B
- …

Test de unidad

- Test que asegura que la unidad mínima de nuestro programa funciona correctamente, y aislada de otras unidades
  - En nuestro caso, la unidad de test es el método
- Testear un método es confirmar que el mismo acepta el rango esperado de entradas, y que retorna le valor esperado en cada caso

Tests automatizados

- Se utiliza software para guiar la ejecución de los tests y controlar los resultados
- Requiere que diseñemos, programemos y mantengamos programas “tests”
  - En nuestro casos serán objetos
- Suele basarse en herramientas que resuelven gran parte del trabajo
- Una vez escritos, los puedo reproducir a costo mínimo, cuando quiera
- Los tests son “parte del software” (y un indicador de su calidad)

Automatizando tests de unidad

jUnit

- jUnit es un framework, en Java, para automatizar la ejecución de tests de unidad
- Ayuda a escribir tests útiles
- Cada test se ejecuta independientemente de otros (aislados)
- jUnit detecta, recolecta, y reporta errores y problemas
- xUnit es su nombre genérico; lo que aprendamos podemos llevarlo a otros lenguajes

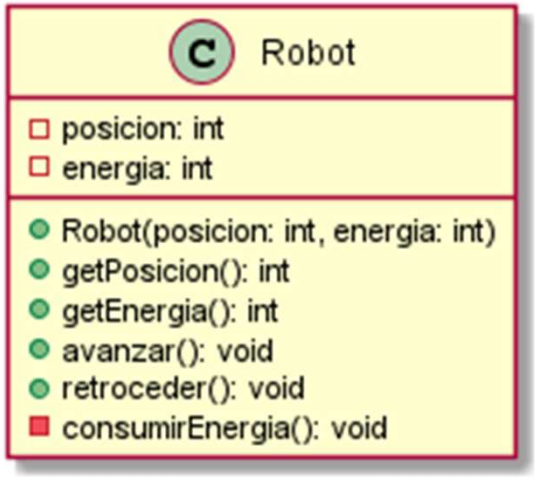
Anatomía de un test suite jUnit

- Una clase de test por cada clase a testear
- Un método que prepara lo que necesitan los tests
  - Y queda en variables de instancia
- Uno o varios métodos de test por cada método a testear
- Un método que limpia lo que se preparo (si es necesario)

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

public class PersonaTest {

    Persona james;

    @BeforeEach
    void setUp() throws Exception {
        james = new Persona();
        james.setApellido("Glosing");
        james.setNombre("James");
    }

    @Test
    public void testNombreCompleto() {
        assertEquals("Glosing, James",
                james.getNombreCompleto());
    }
}
```

El Robot (ejemplo)

..	..	-2	-1	0	1	2	..	..



- Nuestro robot avanza y retrocede de a un lugar
- En cada movimiento consume una unidad de energía
- ¿Qué tests deberíamos escribir?

Independencia entre tests

- No puedo asumir que otro test se ejecutó antes o se ejecutará después del que estoy escribiendo
- Por cada método de test:
  - Se crea una nueva instancia de nuestra clase de test (TestRobot)
  - Se prepara (método marcado como @BeforeEach)
  - Se ejecuta el test y se registran errores y fallas

Importamos las partes de JUnit que necesitamos

Definición y preparación del “fixture)

Ejercitar los objetos

Verificar resultados

Tests

```java
package ar.edu.unlp.info.oo1.ejemploTeoriaTesting;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class RobotTest {

    private Robot robot;

    @BeforeEach
    public void setUp() {
        robot = new Robot(0,100);
    }

    @Test
    public void testAvanzar() {
        robot.avanzar();
        assertEquals(99, robot.getEnergia());
        assertEquals(1, robot.getPosicion());
    }

    @Test
    public void testRetroceder() {
        robot.retroceder();
        assertEquals(99, robot.getEnergia());
        assertEquals(-1, robot.getPosicion());
    }

}
```

El test runner

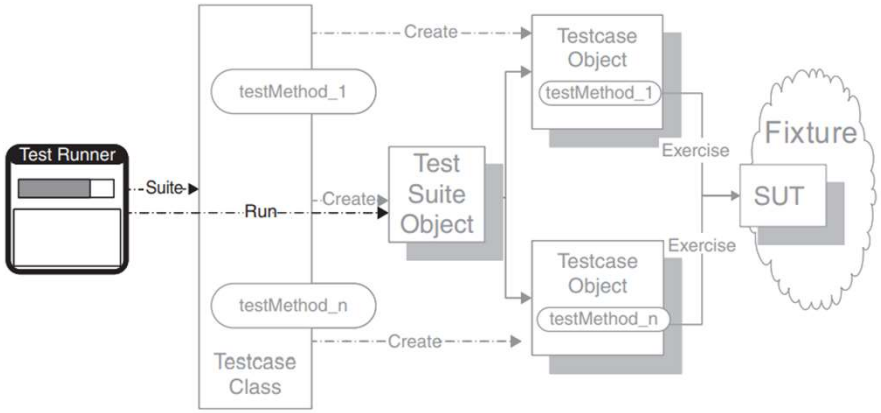
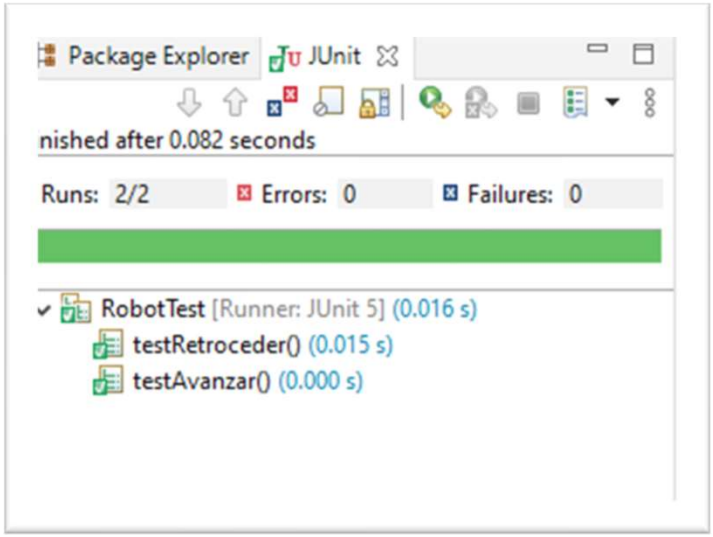


Diagrama tomado de xUnit Testing Patterns pp. 358

```
INFO]
INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ ejemploTeoriaTesting ---
INFO]
INFO] -------------------------------------------------------
INFO]  T E S T S
INFO] -------------------------------------------------------
INFO] Running ar.edu.unlp.info.oo1.ejemploTeoriaTesting.RobotTest
INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.026 s - in ar.e
oTeoriaTesting.RobotTest
INFO]
INFO] Results:
INFO]
INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
INFO]
INFO] ------------------------------------------------------------------------
INFO] BUILD SUCCESS
INFO] ------------------------------------------------------------------------
INFO] Total time:  2.103 s
INFO] Finished at: 2021-09-07T11:00:54-03:00
INFO] ------------------------------------------------------------------------
```



Variantes del assert (algunas)

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertNotEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertNull;
import static org.junit.jupiter.api.Assertions.assertSame;
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class AssertExamples {

    private Object object;

    @BeforeEach
    public void setUp() {
        object = new Object();
    }

    @Test
    public void test() {
        assertEquals(1, 1);
        assertNotEquals(1, 2);
        assertNotNull(object);
        assertNull(null);
        assertSame(object, object);
        assertTrue(true);
        assertFalse(false);
    }

}
```

Pensando los tests

¿Por qué, cuándo, y como testear? (revisado)

- Testeamos para encontrar bugs
- Testeamos con un propósito (buscamos algo)
- Pensamos por qué testear algo y con que nivel queremos hacerlo
- Testeamos temprano y frecuentemente
- Testeo tanto como sea el riesgo del artefacto

Estrategia general

- Pensar que podría variar (que valores puede tomar) y que pueda causar un error o falla
- Elegir valores de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles
- Nos vamos a enfocar en dos estrategias:
  - Particiones equivalentes
  - Valores de borde

Tests de particiones equivalentes

- Partición de equivalencia: conjunto de casos que prueban lo mismo o revelan el mismo bug
  - Asumo que si un ejemplo de una partición pasa el test, los otros también lo harán. Elijo uno.
- Si se trata de valores en un rango, tomo un caso dentro y uno por fuera en cada lado del rango

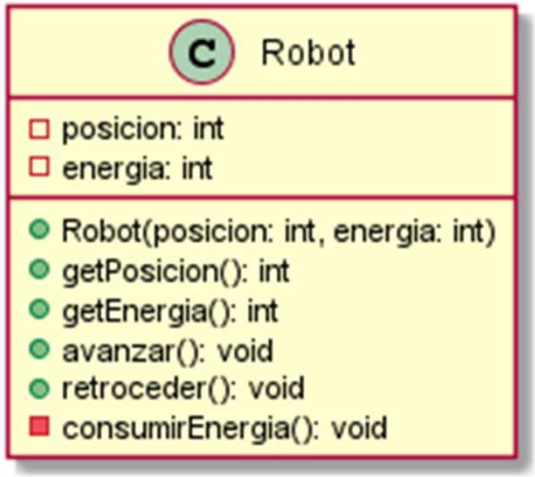
  Ej., la temperatura debe estar entre 0 y 100 - > casos: -50, 50 , 150.
- Si se trata de casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no

  Ej., la temperatura debe ser un valor positivo- > Casos: -50, 50

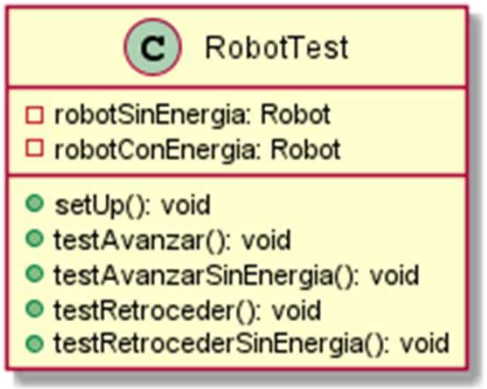
Equivalence classing, equivalence partitioning

El Robot (revisado)

..	..	-2	-1	0	1	2	..	..



- Nuestro robot avanza y retrocede de a un lugar
- Si no tiene energía se queda en el lugar (aunque le pida que avance o retroceda)
- ¿Qué tests deberíamos escribir?



- Tenemos dos métodos a testear
- Tenemos dos particiones
  - Sin energía (energia = 0)
  - Con energía (energia != 0)
- Para “sin energía” no tenemos mucho que elegir
- Para “con energía” podríamos elegir cualquier representante

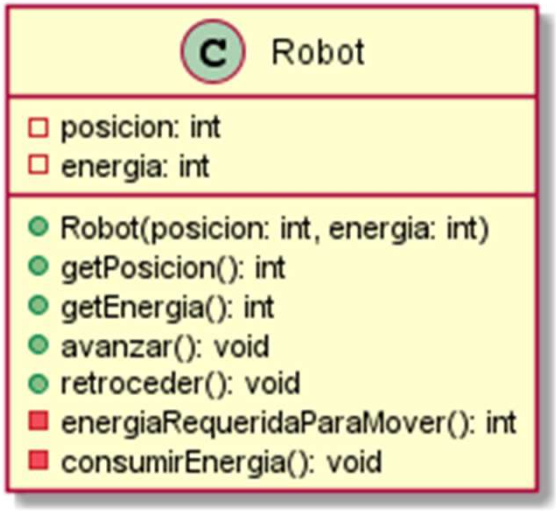
```java
public class RobotTest {

    private Robot robotSinEnergia;
    private Robot robotConEnergia;

    @BeforeEach
    public void setUp() {
        robotConEnergia = new Robot(0,100);
        robotSinEnergia = new Robot(0,0);
    }

    @Test
    public void testAvanzar() {
        robotConEnergia.avanzar();
        assertEquals(99, robotConEnergia.getEnergia());
        assertEquals(1, robotConEnergia.getPosicion());
    }

    @Test
    public void testAvanzarSinEnergia() {
        robotSinEnergia.avanzar();
        assertEquals(0, robotSinEnergia.getPosicion());
    }

    @Test
    public void testRetroceder() {
        robotConEnergia.retroceder();
        assertEquals(99, robotConEnergia.getEnergia());
        assertEquals(-1, robotConEnergia.getPosicion());
    }

    @Test
    public void testRetrocederSinEnergia() {
        robotSinEnergia.retroceder();
        assertEquals(0, robotSinEnergia.getPosicion());
    }

}
```

Tests con valores de borde

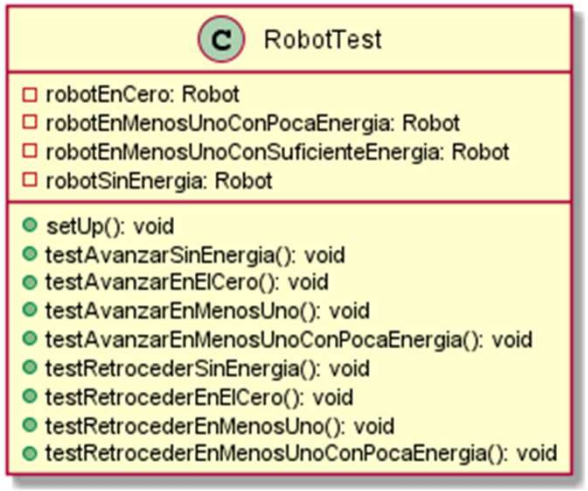
- Los errores ocurren con frecuencia en los limites y ahí es donde los vamos a buscar
- Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores
- Buscar los bordes en propiedades del etilo: velocidad, cantidad, posición, tamaño , duración, edad, etc.
- Y buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleno, antes/después, junto a, alejado de , etc.

El Robot (revisado)

..	..	-2	-1	0	1	2	..	..



- Nuestro robot avanza y retrocede de a un lugar
- Si no tiene energía se queda en el lugar (aunque le pida que avance o retroceda)
- En los positivos, cada movimiento consume una unidad de energía
- En los negativos, cada movimiento consume dos unidades
- ¿Qué tests deberíamos escribir?



- Tenemos dos métodos a testear
- Tenemos dos condiciones de posición (< 0 y >= 0)
  - 0 y -1 son nuestros bordes
- Tenemos tres condiciones de energía: nada, 1, 2 o más.
  - 0, 1, 2 son nuestros bordes

```java
public class RobotTest {

    private Robot robotEnCero, robotEnMenosUnoConPocaEnergia,
                robotEnMenosUnoConSuficienteEnergia, robotSinEnergia;

    @BeforeEach
    public void setUp() {
        robotSinEnergia = new Robot(0,0);
        robotEnCero = new Robot(0, 1);
        robotEnMenosUnoConPocaEnergia = new Robot(-1, 1);
        robotEnMenosUnoConSuficienteEnergia = new Robot(-1, 2);

    }

    @Test
    public void testAvanzarSinEnergia() {
        robotSinEnergia.avanzar();
        assertEquals(0, robotSinEnergia.getPosicion());
    }

    @Test
    public void testAvanzarEnElCero() {
        robotEnCero.avanzar();
        assertEquals(0, robotEnCero.getEnergia());
        assertEquals(1, robotEnCero.getPosicion());
    }

    @Test
    public void testAvanzarEnMenosUno() {
        robotEnMenosUnoConSuficienteEnergia.avanzar();
        assertEquals(0, robotEnMenosUnoConSuficienteEnergia.getEnergia());
        assertEquals(0, robotEnMenosUnoConSuficienteEnergia.getPosicion());
    }

    @Test
    public void testAvanzarEnMenosUnoConPocaEnergia() {
        robotEnMenosUnoConPocaEnergia.avanzar();
        assertEquals(1, robotEnMenosUnoConPocaEnergia.getEnergia());
        assertEquals(-1, robotEnMenosUnoConPocaEnergia.getPosicion());
    }
```

Testing en OO1

- En el marco de OO1, testear es asegurarnos de que nuestros objetos hacen lo que se espera, como se espera
- Escribir tests de unidad (con JUnit) es parte “programar”
- Escribir tests nos ayuda a entender que se espera de nuestros objetos
- Podemos hacer TDD o no, pero siempre escribimos tests

Bibliografía