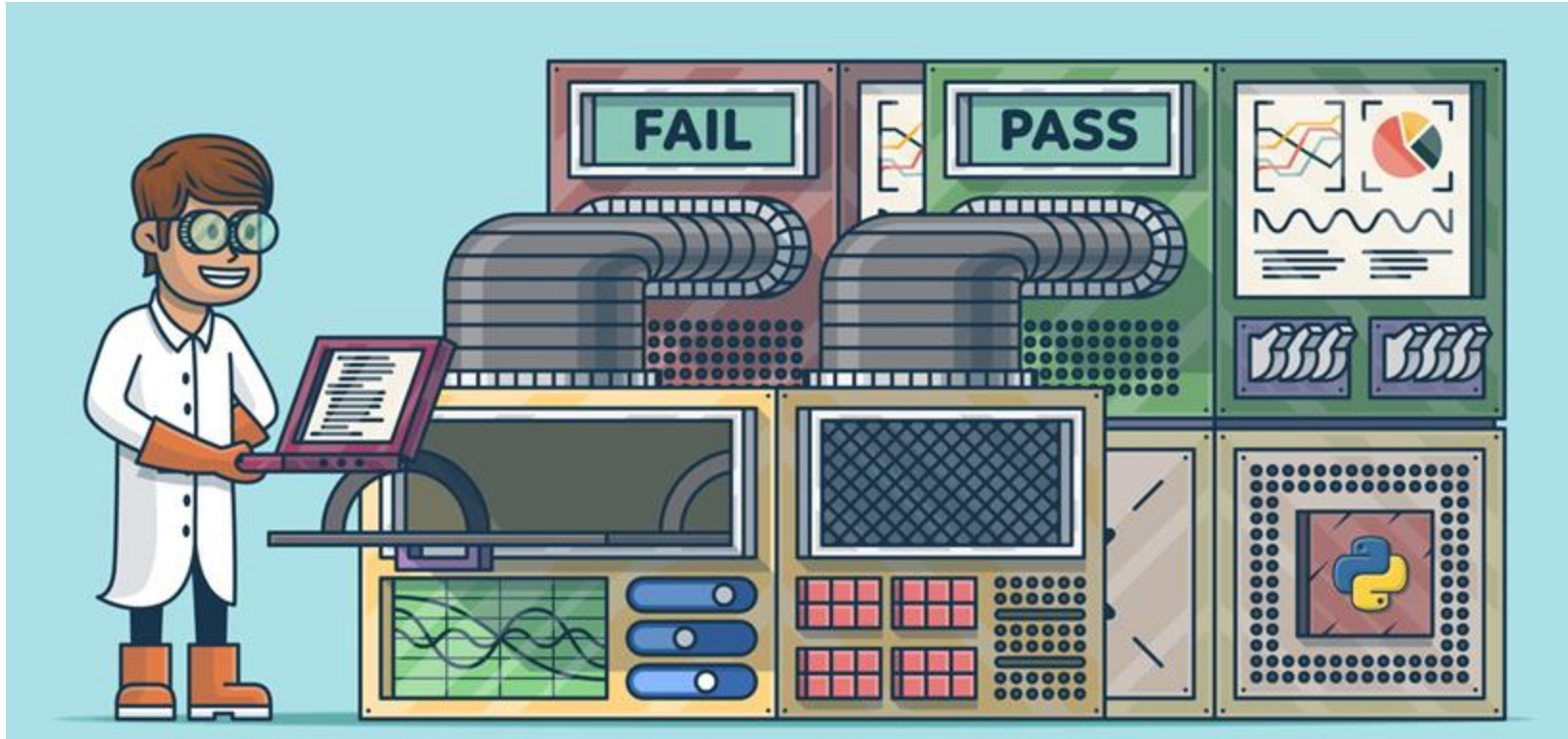


# Testing

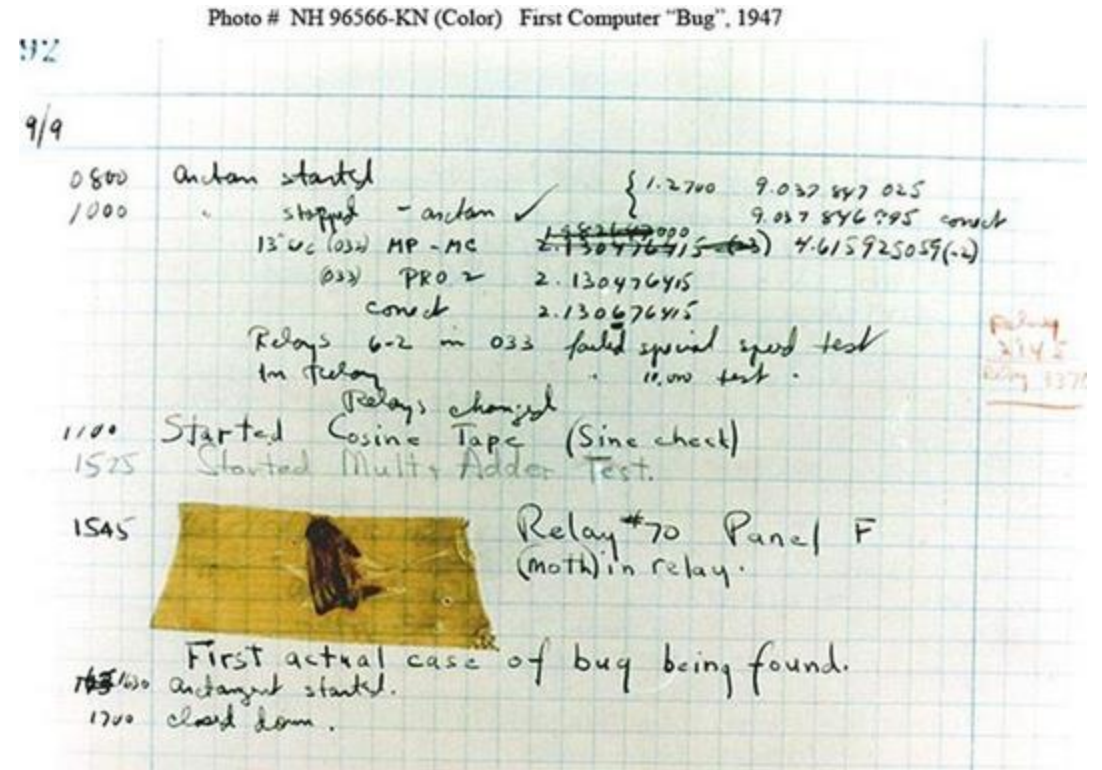


# Agenda de la clase

- Que es hacer tests y que tipos de tests hay
- Tests de unidad
- Automatización de tests de unidad con Junit
- Elección y diseño de mis casos de test – dos estrategias
  - Particiones de equivalencia
  - Valores de borde
- Tests automatizados y cobertura

# ¿Qué es un bug/error?

- El programa no hace algo que debería hacer
- El programa hace algo mal
- El programa falla (revienta)



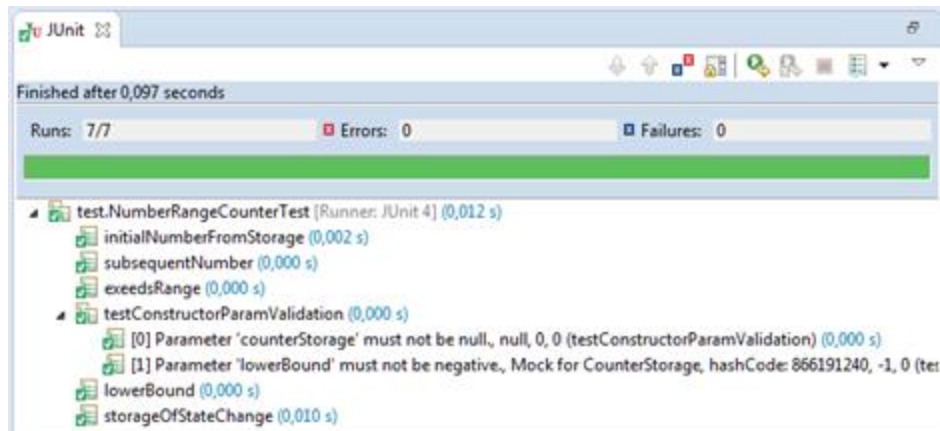
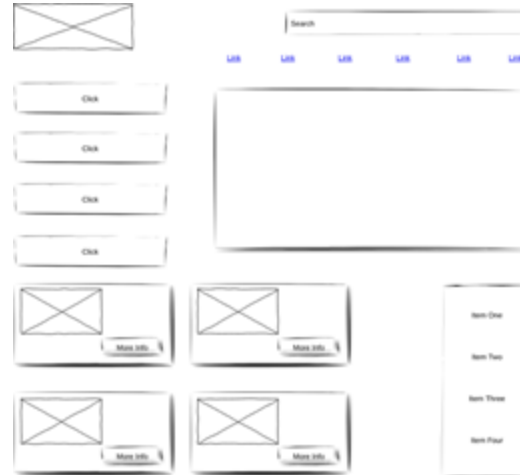
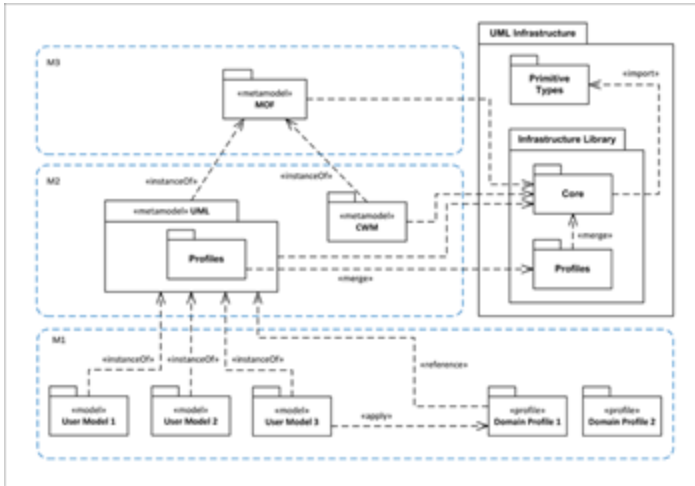
# ¿Qué es testear?

## Asegurarse de que el programa:

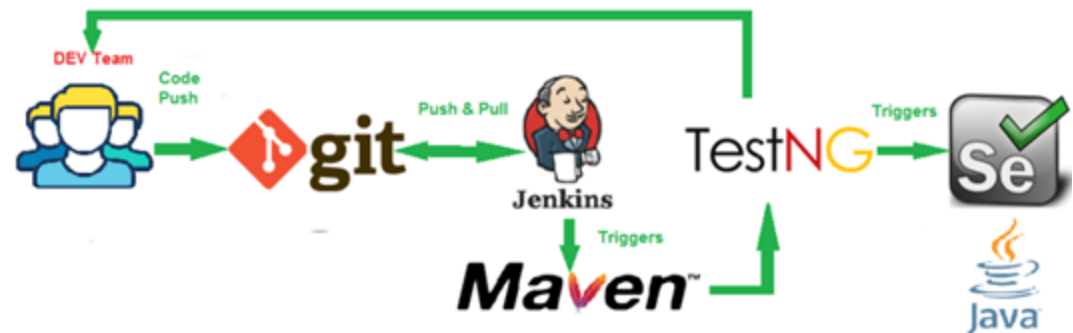
- hace lo que se espera
- lo hace como se espera y
- no falla



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



## How Run Selenium Tests in Jenkins Using Maven



# 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
- ...



# ¿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?)

# 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 el valor esperado en cada caso
  - tengo en cuenta parámetros,
  - estado del objeto antes de ejecutar el método,
  - objeto que retorna el método, y
  - estado del objeto al concluir la ejecución del método



“pre-condiciones”

“post-condiciones”



# 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 caso, esos programas 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 una herramienta para simplificar la creación de tests de unidad y automatizar su ejecución y reporte
- Ayuda a escribir/programar 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 (el fixture)
  - Y queda en variables de instancia
- Uno o varios métodos de test por cada método a testear
- Un método que limpia lo que se preparó (si es necesario)

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

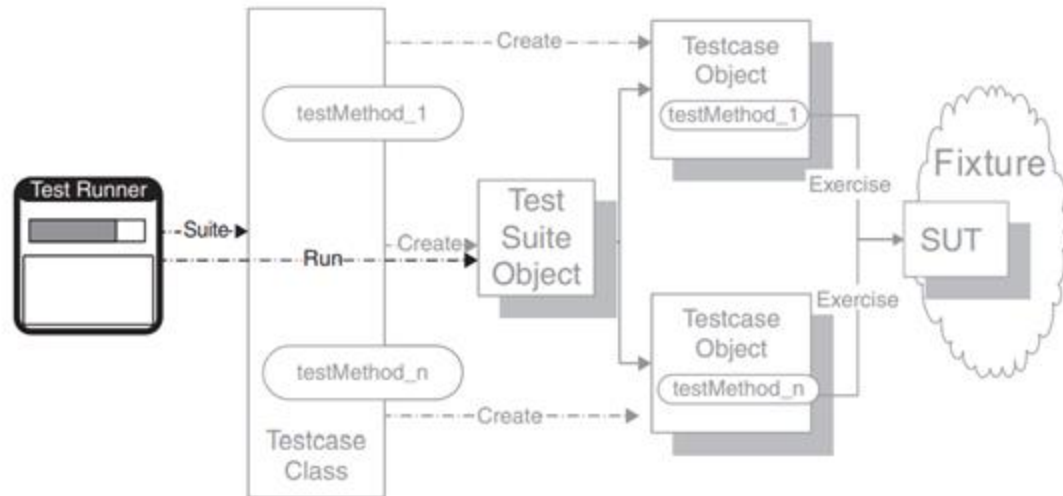
public class PersonaTest {

    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 test runner



```
[INFO] --- maven-surefire-plugin:2.22.2:test (default-test) @ ejemploTeoriaTesting ---
[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.edu.unlp.info.oo1.ejemploTeoriaTesting.RobotTest
[INFO] Results:
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 2.103 s
[INFO] Finished at: 2021-09-07T11:00:54-03:00
[INFO] -----
```

The screenshot shows an IDE window with the 'JUnit' tab selected. The 'Package Explorer' on the left shows the project structure. The main area displays the test results for 'RobotTest' [Runner: JUnit 5] (0.016 s). The results show two tests: 'testRetroceder()' (0.015 s) and 'testAvanzar()' (0.000 s), both passed. The status bar at the bottom indicates 'Finished after 0.082 seconds' and 'Runs: 2/2', 'Errors: 0', 'Failures: 0'. A green progress bar is visible below the status bar.

# 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 (marcado con `@Test`):
  - Se crea una nueva instancia de nuestra clase de test
  - Se prepara (con el método marcado como `@BeforeEach`)
  - Se ejecuta el test y se registran errores y fallas

# El Robot (ejemplo)



Robot
-position: int -energy: int
+getPosition(): int +getEnergy(): int +goForward() +goBackwards() -consumeEnergy()

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



Importamos las partes  
de JUnit que necesitamos

```
package robots;

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

Definición y preparación  
del "fixture"

```
class RobotTest {

    private Robot robot;

    @BeforeEach
    void setUp() {
        robot = new Robot(position:0,energy:100);
    }
```

Ejercitar los objetos  
Verificar resultados

```
    @Test
    void testGoForward() {
        robot.goForward();
        assertEquals(1, robot.getPosition());
        assertEquals(99, robot.getEnergy());
    }
```

Ejercitar los objetos  
Verificar resultados

```
    @Test
    void testGoBackwards() {
        robot.goBackwards();
        assertEquals(-1, robot.getPosition());
        assertEquals(99, robot.getEnergy());
    }
```

Tests

# Variantes del assert (algunas)

@Test

```
void assertExamples() {  
    assertEquals(5, "Hello".length());  
    assertNotEquals("Hello", "Bye");  
    assertNotNull(myList);  
    assertSame(myList, someList);  
    assertTrue(myList.isEmpty());  
    assertFalse(someList.isEmpty());  
    assertThrows(IndexOutOfBoundsException.class, () -> {  
        myList.remove("Hello");  
    });  
}
```

testingmain

Project

testing

src

main

java

robots

100% classes, 78% lines covered

Battery

InfiniteBattery100% methods, 100% lines covered

SustainableRobot66% methods, 66% lines covered

resources

test

java

robots

InfiniteBatteryTest

SustainableRobotTest

target

ry.java

SustainableRobot.java

SustainableRobotTest.java

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

\* It moves forwards and backwards by one.

\* It only moves if its has a battery installed (may not have

\* It does not consume energy to move.

\*/

public class SustainableRobot { new \*

private int position;

private Battery battery;

public SustainableRobot(int position) { new \*

this.position = position;

}

public void removeBattery() { new \*

battery = null;

}

Cover

robots in testing

✓ robots114 ms

✓ InfiniteBatteryTest113 ms

✓ testProvide()108 ms

✓ testAvailableEnergy()1 ms

✓ testCanProvide()4 ms

Coverage

robots in testing

Element	Class, %	Method, %	Line, %	Branch...
robots	100% (2/2)	80% (8/10)	78% (11/14)	50% (2/4)
Battery	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
InfiniteBattery	100% (1/1)	100% (4/4)	100% (5/5)	100% (0/0)
SustainableRobot	100% (1/1)	66% (4/6)	66% (6/9)	50% (2/4)

testing > src > main > java > robots

9:22 LF UTF-8 4 spaces

# Cobertura

- Cuando testeamos nos interesa saber cuan completos/integrales son nuestros tests – podemos medirlo de distintas formas
  - Clases cubiertas, métodos cubiertos, líneas cubiertas
  - Condicionales (ver que se ejecutaron con true y false)
  - Caminos/branches (ver si pasó por todos lados)
- Las herramientas modernas observan y reportan esos y otros valores
- Como escribir y mantener tests requiere esfuerzo no siempre maximizamos su cobertura
- Diseñar bien nuestros tests nos ayuda a enfocar el esfuerzo, optimizar el resultado y obtener un balance adecuado esfuerzo/cobertura

# 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 qué nivel queremos hacerlo
- Testeamos temprano y frecuentemente
- Testeo tanto como sea el riesgo del artefacto
- No es necesario testear código de base que otros ya testearon (por ejemplo, partes del SDK, etc.)

# Estrategia general

- Pensar que podría variar (que valores puede tomar) y que pueda causar un error o falla
- Elegir valores (de estados y parámetros) de prueba para maximizar las chances de encontrar errores haciendo la menor cantidad de pruebas posibles
  - Una combinación de valores es “un caso de prueba”
- 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 casos en un conjunto, tomo un caso que pertenezca al conjunto y uno que no**
  - Ej., debe tener entrada -> Casos: una persona con entrada, una sin
- **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.
  - Veremos que estos casos pueden mejorarse

# El Robot minimalista



MinimalRobot
-position: int
+getPosition(): int +goForward() +goBackwards()

- Nuestro robot avanza y retrocede de a un lugar sin importarle nada
- ¿Qué tests deberíamos escribir? (Qué clases y qué métodos)
- ¿Qué particiones identificamos?
- ¿Cuáles serían buenos casos de test para cada una?
- ¿Podemos pensar otros casos que prueben algo diferente?

MinimalRobot
-position: int
+getPosition(): int +goForward() +goBackwards()

- Identificamos dos métodos a testear
- Identificamos una sola partición
- Cualquier robot da lo mismo
- ¿Qué métodos cubrimos?

```
class MinimalRobotTest {  
  
    private MinimalRobot robot;  
  
    @BeforeEach  
    void setUp() {  
        robot = new MinimalRobot();  
    }  
  
    @Test  
    void goForward() {  
        robot.goForward();  
        assertEquals(1, robot.getPosition());  
    }  
  
    @Test  
    void goBackwards() {  
        robot.goBackwards();  
        assertEquals(-1, robot.getPosition());  
    }  
}
```

# El Robot apagable



## ToggleableRobot

-position: int  
-isOn: boolean

+getPosition(): int  
+toggle()  
+goForward()  
+goBackwards()

- Se puede encender y apagar (con un mismo mensaje)
- Si esta apagado, no hace nada al pedirle que se mueva
- ¿Qué tests deberíamos escribir?
- ¿Qué particiones identificamos?
- ¿Cuáles serían buenos casos de test para cada una?
- ¿Podemos pensar otros casos que prueben algo diferente?

ToggleableRobot
-position: int -isOn: boolean
+getPosition(): int +toggle() +goForward() +goBackwards()

- Identificamos dos métodos a testear
  - ¿Necesitamos testear toggle()?
- Identificamos dos particiones
  - Encendido, en cualquier lugar
  - Apagado, en cualquier lugar
- Cuatro casos en total
  - Podemos tener varios casos en un método de test
  - Podemos separar casos en métodos

```
class ToggleableRobotTest {  
    private ToggleableRobot onRobot, offRobot;  
  
    @BeforeEach  
    void setUp() {  
        onRobot = new ToggleableRobot();  
        onRobot.toggle();  
        offRobot = new ToggleableRobot();  
    }  
  
    @Test  
    void testGoForward() {  
        //Partition of robots that are turned on  
        onRobot.goForward();  
        assertEquals(1, onRobot.getPosition());  
        //Partition of robots that are turned off  
        offRobot.goForward();  
        assertEquals(0, offRobot.getPosition());  
    }  
  
    @Test  
    void testGoBackwards_onRobots() {  
        onRobot.goBackwards();  
        assertEquals(-1, onRobot.getPosition());  
    }  
}
```

# Tests con valores de borde

- Los errores ocurren con frecuencia en los límites y ahí es donde los vamos a buscar
- Intentamos identificar bordes en nuestras particiones de equivalencia y elegimos esos valores
- Buscar los bordes en combinaciones de estados/parámetros: velocidad, cantidad, posición, tamaño, duración, edad, etc.
  - También podemos buscar en relaciones entre ellas (diferencia entre saldo y monto a extraer)
- Y buscar valores como: primero/último, máximo/mínimo, arriba/abajo, principio/fin, vacío/lleño, antes/después, junto a, alejado de, etc.

# El Robot positivista



## PositivistRobot

-position: int

+getPosition(): int

+goForward()

+goBackwards()

- Nuestro robot avanza y retrocede de a un lugar, **pero solo en positivos (desde 0 a Integer.MAX\_VALUE)**
- ¿Qué particiones identificamos?
- ¿Cuáles son los bordes?
- ¿Podemos pensar otros casos que prueben algo diferente?



PositivistRobot
-position: int
+getPosition(): int +goForward() +goBackwards()

- Identificamos dos métodos a testear
- ¿Qué particiones/bordes encontramos para cada método?
  - Considero estado, parámetros y semántica del método
- Las particiones/bordes pueden ser diferentes para distintos métodos

```

class PositivistRobotTest {
    PositivistRobot robotAtZero, robotAtMax, robotInBetween;
    @BeforeEach
    void setUp() {
        robotAtZero = new PositivistRobot();
        robotAtMax = new PositivistRobot(Integer.MAX_VALUE);
        robotInBetween = new PositivistRobot(100);
    }

    @Test
    void testGoBackwards_inBetween() {
        robotInBetween.goBackwards();
        assertEquals(99, robotInBetween.getPosition());
    }

    @Test
    void testGoBackwards_atZero() {
        robotAtZero.goBackwards();
        assertEquals(0, robotAtZero.getPosition());
    }

    @Test
    void testGoForward_inBetween() {...}
    @Test
    void testGoForward_atMax() {...}

```

# El saltarín y hambriento



JumpingHungryRobot
-position: int -hunger: int -energy: int
+getPosition(): int +jumpForward(places: int) +jumpBackwards(places: int) +charge(amount: int): int +setHunger(hunger: int)

- Tiene energía (que puede ser 0), y tiene hambre (mínimo 1)
- Cada lugar que se mueve usa tanta energía como su hambre indica
- Si no tiene energía suficiente se queda en el lugar (aunque le pida que avance o retroceda)
- En lugar de avanzar y retroceder de a uno, salta

JumpingHungryRobot
-position: int -hunger: int -energy: int
+getPosition(): int +jumpForward(places: int) +jumpBackwards(places: int) +charge(amount: int): int +setHunger(hunger: int)

- Identificamos tres métodos a testear: charge y los dos jump
- ¿Qué particiones/bordes encontramos para cada método?
  - Pienso en combinaciones de carga, hambre y cantidad de lugares

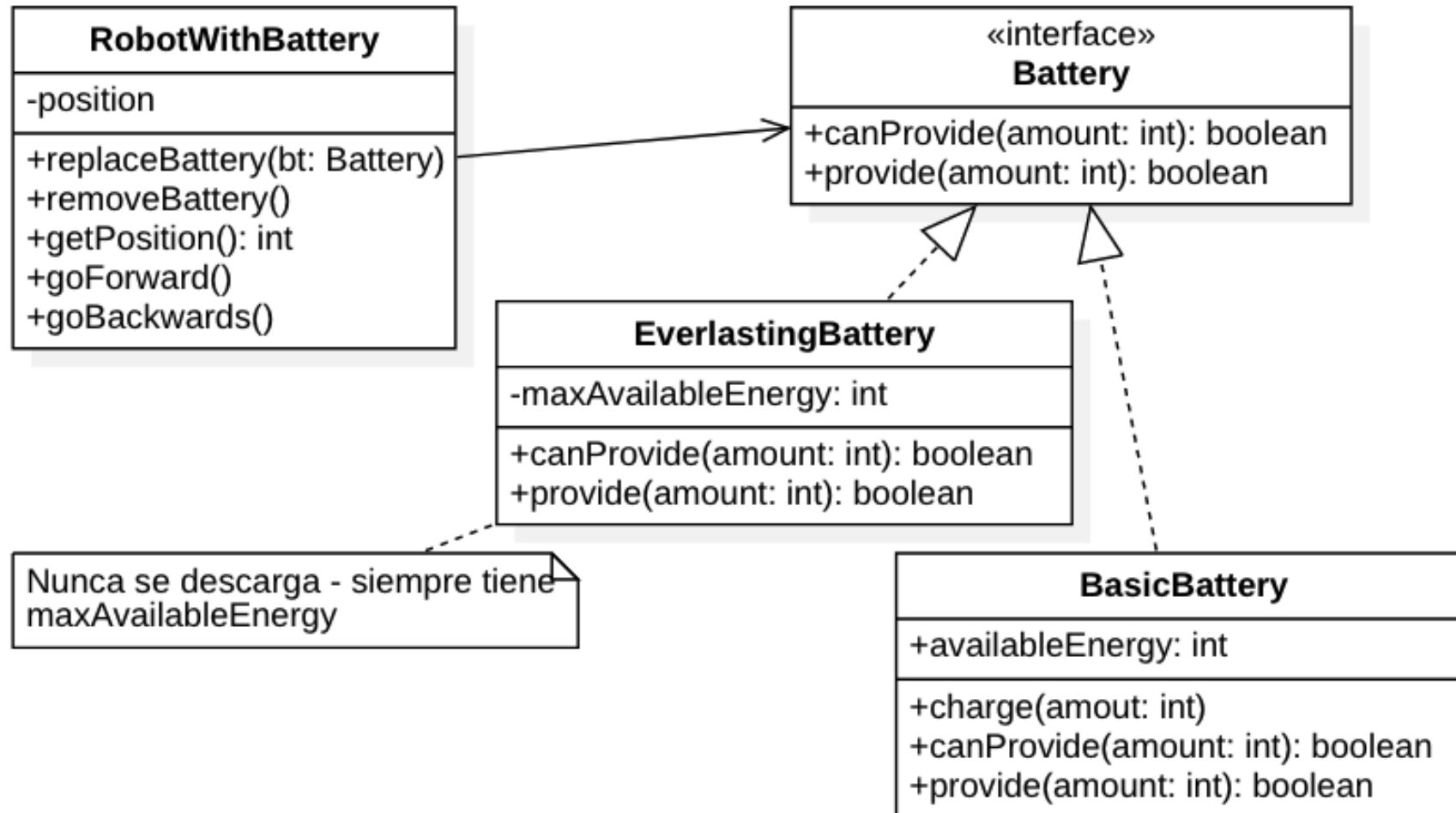
- testCharge()
  - position y hunger son irrelevantes
  - Cualquier combinación energía/amount sirve (¿no?)
    - Elijo: energy = 0; amount = 1;
- testJump...()
  - Considero la relación energy, hunger, y places
  - Partición sin suficiente energía
    - Elijo uno de los casos mínimos
    - energy = 0; hunger = 1; places = 1
  - Partición con suficiente energía
    - Elijo uno de los casos mínimos
    - energy = 1; hunger = 1; places = 1
  - ¿algo más?

Cuando se les pida  
“identifique, especifique y  
justifique los casos de test”,  
se espera que respondan  
algo como esto ...

- Identificamos tres métodos a testear: charge y los dos jump
- ¿Qué particiones/bordes encontramos para cada método?
  - Pienso en combinaciones de carga, hambre y cantidad de lugares

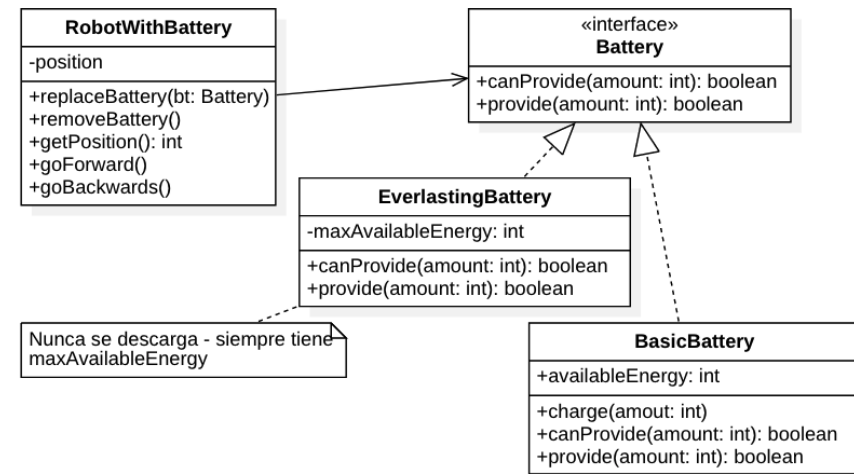
- testCharge()
  - position y hunger son irrelevantes
  - Cualquier combinación energía/amount sirve (¿no?)
    - Elijo: energy = 0; amount = 1;
- testJump...()
  - Considero la relación energy, hunger, y places
  - Partición sin suficiente energía
    - Elijo uno de los casos mínimos
    - energy = 0; hunger = 1; places = 1
  - Partición con suficiente energía
    - Elijo uno de los casos mínimos
    - energy = 1; hunger = 1; places = 1
  - ¿algo más?

# ¿Qué testeamos en este caso y cómo?

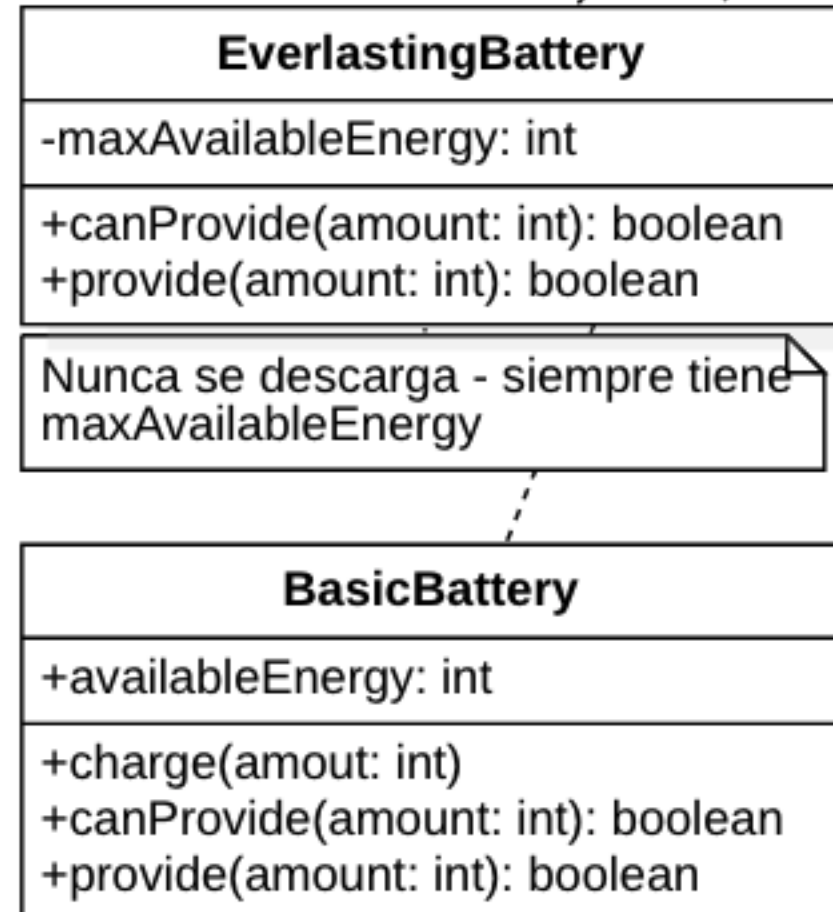


# ¿Qué testeamos en este caso y cómo?

- Testeamos cada clase por separado
- Prestamos atención a los casos que son relevantes en cada clase
- Atención: ¡ al testear RobotWithBattery, evitamos redundar en tests de las baterías ! Eso ya esta testeado.

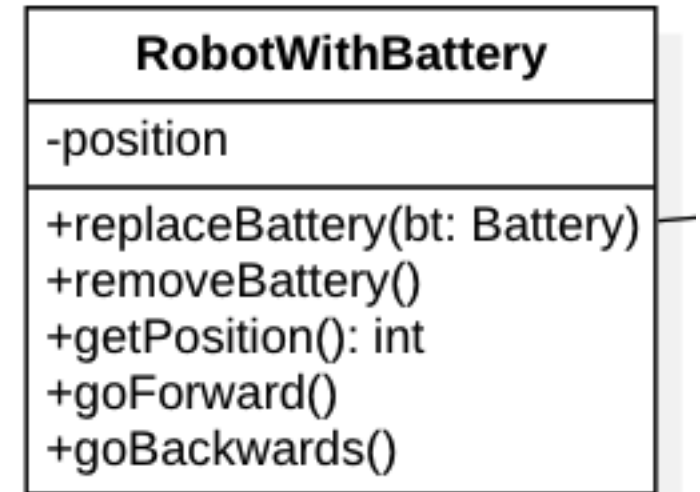


- EverlastingBattery
  - canProvide() y provide() con:
    - amount = 1 & amount = 2
  - maxAvailableEnergy = 1 ;
- BasicBattery
  - canProvide() y provide() con:
    - amount = 1;
  - availableEnergy = 0 & 1 ;





- RobotWithBattery
  - goForward() y goBackwards()
- Pensando en la relación energía-consumo
  - En cualquier lugar - elijo position = 0
  - Sin suficiente energia (cualquier bateria)
    - Elijo una BasicBaterly con energy = 0
  - Con suficiente energia (cualquier bateria)
    - Elijo una BasicBaterly con energy = 1 (un límite)
- En los límites & suficiente energia
  - Elijo una BasicBaterly con energy = 1 (un límite)
  - goForward() en Integer.MAX\_VALUE
  - goBaclwards() en Integer.MAX\_VALUE



# 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 de “programar”
- Escribir tests nos ayuda a entender que se espera de nuestros objetos
- Con lo que sabemos hasta ahora encontraremos situaciones complejas de resolver
  - Ya veremos en OO2 estrategias para atacarlas
  - Por ahora el foco es testear con propósito, y diseñar bien los tests/casos

# Bibliografía

