AUXILIAR 3
TDD: TESTING Y DOUBLE DISPATCH

Ignacio Slater Muñoz

v.1.0.x0006

¿Qué son los *tests*?

Básicamente son programas que *prueban* el correcto funcionamiento de alguna fracción de un programa.

En este curso nos interesarán particularmente los **tests unitarios.** La idea es que cada uno de estos pruebe una funcionalidad en específico del programa.

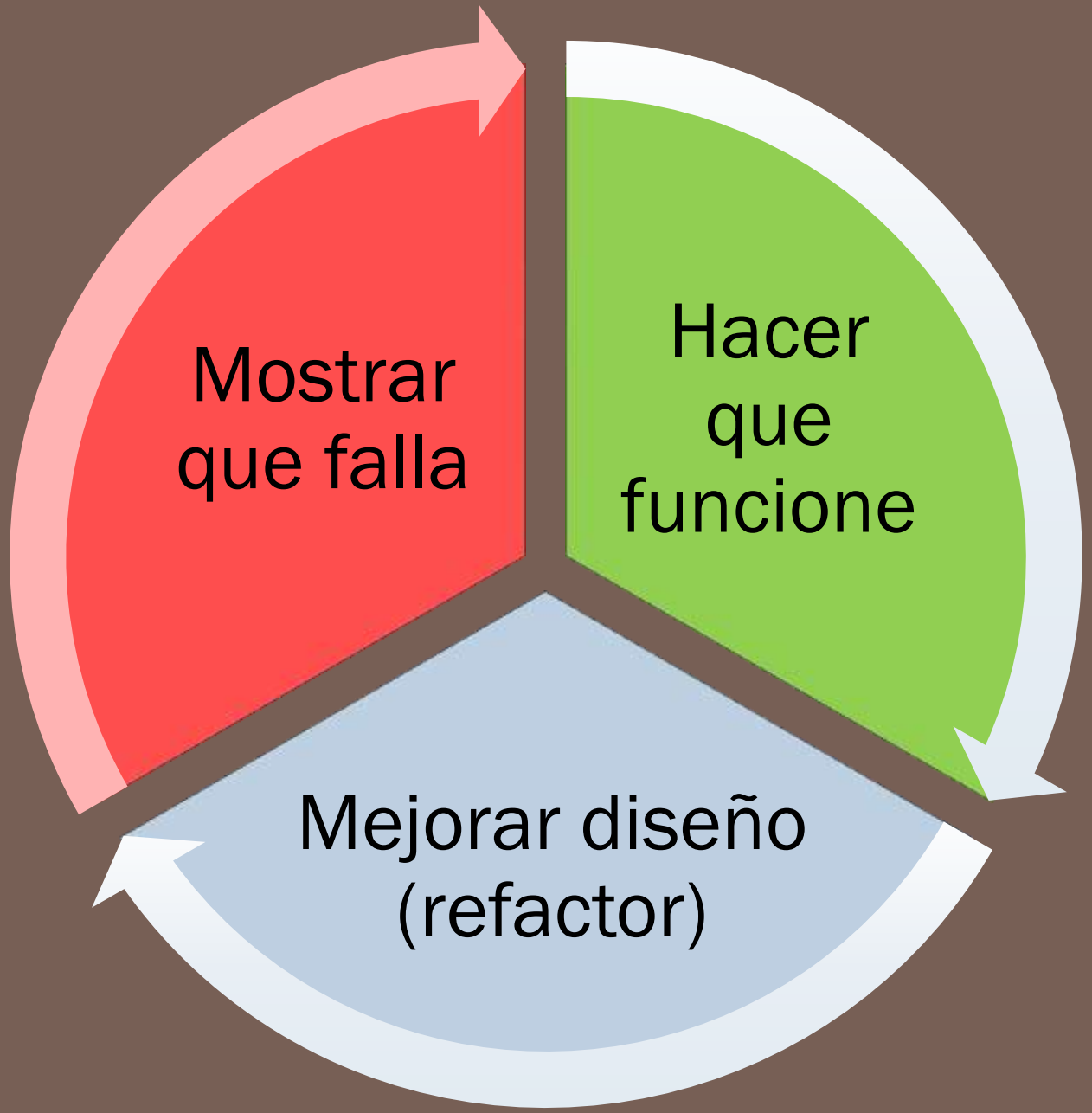
Existen muchos más tipos de tests, pero el ramo no alcanzaría para verlos bien

Cuando hablemos de *coverage* nos estaremos refiriendo al porcentaje de código que prueba el *test*.

Tener 100 % de coverage no implica que su programa no tenga errores.

Test Driven Development:

¿Por dónde empiezo?



¿Por qué hacer los *tests* primero?

- Se me ocurren **3 razones**:
  - ***Fallar** rápido*
  - *Enfocarse en el **resultado esperado** más que en la implementación*
  - *Evitar la **imparcialidad** al escribir los tests*

JUnit 5

- Es un *framework* para crear *tests* unitarios en *Java.*

- Para crear tests debemos crear una clase con **métodos especiales**. Lo único que los diferencia de un método cualquiera es que tienen una **anotación** antes:
  - **`@Test`:** *Son los métodos que prueban la funcionalidad del programa. Para esto se utilizan métodos de JUnit, los más comunes son* `assertEquals`*,* `assertTrue` *y* `assertFalse`*.*
  - **`@BeforeEach`:** *El método marcado con esto se ejecuta* ***antes de cada*** *test.*
  - **`@AfterEach`***: Lo mismo que el anterior, pero se ejecuta* ***después de cada*** *test.*

- JUnit se encargará de entregar información sobre los *tests* ejecutados (como el coverage)

Básicamente

¿QUÉ DIFERENCIA A UNA LIBRERÍA DE UN *FRAMEWORK*?

¿Les suena Pokémon?

- En el juego Pókemon existen distintos tipos de ataques y Pokémon.
- Dependiendo de los tipos, un ataque puede realizar más o menos daño.
- Consideremos que los ataques de tipo agua son poco efectivos contra los Pokémon tipo planta, por lo que los ataques realizan 20 puntos menos de daño.
- Por otro lado, los ataques tipo planta son fuertes contra los Pokémon de agua, por lo que hacen 1,5 veces el daño.
- **Ejercicio:** Implemente un test que pruebe el comportamiento esperado.

DOUBLE DISPATCH

Está bien si no lo entienden a la primera, es complicado ☹

Empecemos con algo simple: Imprimir en Toqui

```
public class PSDocument {
  private String nombre;
  public void printDuplex() {
    execute("duplex " + nombre + "|lpr");
  }
  private void execute(String instruction) throws IOException {
    Runtime.getRuntime().exec(instruction);
  }
}
```

¿Y si quiero imprimir en la salita?

We need more printers!

```
public class SalitaPrinter
    implements IPrinter {
  @Override
  public void printDuplex(
      PSDocument document)
      throws IOException {

    execute("duplex "
      + document.getName()
      + "|lpr -P hp-335");
  }
}
```

```
public class ToquiPrinter
    implements IPrinter {
  @Override
  public void printDuplex(
      DocumentoPS documento)
      throws IOException {

    execute("duplex "
      + document.getName()
      + "|lpr");
  }
}
```

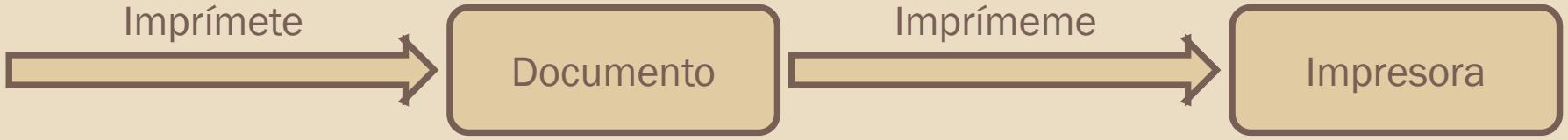
¿Y si ahora queremos imprimir archivos PDF? ¿Cómo diferenciamos los tipos?

```
public class ToquiPrinter implements IPrinter {
  @Override
  public void printDuplex(final IDocument document) throws IOException {
    if (document instanceof PSDocument) {
      execute("duplex " + document.getName() + " | lpr");
    }
    if (document instanceof PDFDocument) {
      execute("pdf2ps " + document.getName() + " out.ps");
      execute("duplex out.ps | lpr");
    }
  }
}
```

¿Algún problema aquí?

A lo que vinimos. *Double Dispatch*

- Se hacen 2 llamados (por eso el nombre)
  - *El primero **desambigua el tipo***
  - *El segundo ejecuta la acción*
- Así, no necesitamos *preguntar por el tipo* del objeto



Double Dispatch en acción

```
public class PDFDocument extends AbstractDocument {
  @Override
  public void printDuplex(IPrinter printer) throws IOException {
    printer.printDuplex(this);
  }
}

public class PSDocument extends AbstractDocument {
  @Override
  public void printDuplex(IPrinter printer) throws IOException {
    printer.printDuplex(this);
  }
}
```

Espérate un tantito, eso es overloading

Bueno

```
public class PDFDocument extends AbstractDocument {
  @Override
  public void printDuplex(IPrinter printer) throws IOException {
    printer.printDuplexPDF(this);
  }
}

public class PSDocument extends AbstractDocument {
  @Override
  public void printDuplex(IPrinter printer) throws IOException {
    printer.printDuplexPS(this);
  }
}
```

Y la impresora

```
public class ToquiPrinter implements IPrinter {
  @Override
  public void printDuplexPS(PSDocument document) throws IOException {
    execute("duplex " + document.getName() + " | lpr");
  }

  @Override
  public void printDuplexPDF(PDFDocument document) throws IOException {
    execute("pdf2ps " + document.getName() + " out.ps");
    execute("duplex out.ps | lpr");
  }
}
```

De vuelta a Pokémon

- Implemente un combate Pokémon, para esto considere que:
  - *Todo Pokémon tiene un nombre y un tipo, y puede tener hasta 4 ataques*
  - *Cada ataque tiene un tipo, un daño base y un nombre*
  - *Un Pokémon debe ser capaz de atacar a otro*
- Además, todo Pokémon presenta resistencias y debilidades de acuerdo a su tipo
  - *Si es resistente, el daño recibido se reduce en 20*
  - *Si es débil, el daño aumenta 1,5 veces*

	Resistente	Débil
Grass	Water, Grass	Fire
Fire	Fire, Grass	Water
Water	Fire, Water	Grass
Normal		

La verdadera auxiliar son los amigos que hicimos en el camino

Soluciones:
https://github.com/islaterm/CC3002-2019P-Auxiliares/tree/master/Auxiliar%203%20-%202019P