

# Capacitación Pruebas Unitarias (xUnit)

16 Noviembre 2022 – Sesión 5

**FUTURE  
AT HEART**

# Contenido



## Modulo I – Conceptos Básicos

- ¿Qué es una prueba unitaria?
- Otros niveles de prueba
- Conociendo las pruebas unitarias
- Técnicas de diseño de casos de prueba
- Principios FIRST

## Modulo II – Test Driven Development

- Definición
- Desarrollo Ágil – Características
- Ciclo de desarrollo TDD
- Como escribir código que se pueda probar
- Cobertura de código
- Ventajas/Desventajas

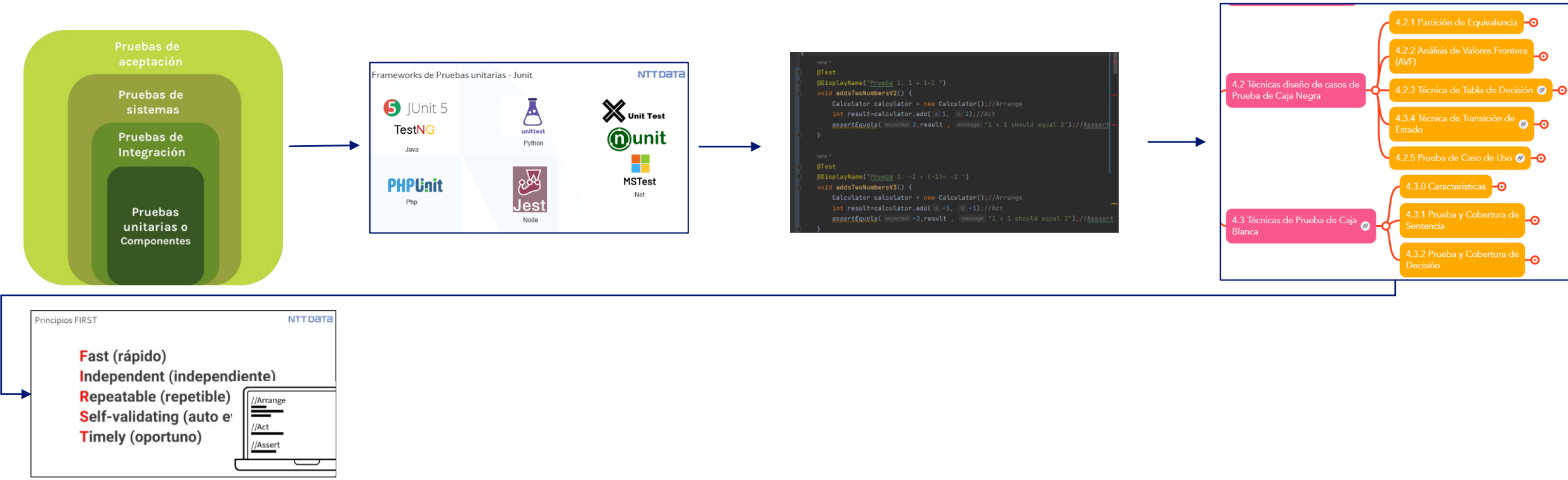
## Modulo III – xUnit

- ¿Qué es xUnit?
- ¿Por qué xUnit?
- ¿Cómo funciona xUnit?
- Conceptos básicos
- Escenarios con xUnit
- Patrones de nombramiento de los escenarios
- Simulaciones
- Aserciones

## Modulo IV – Taller

- Preparación del Entorno
- Ejemplo práctico
- Creando casos de prueba
- Ejecución de casos
- Mutation Testing
- Conclusiones/Recomendaciones


En capítulos anteriores...



9 -Verdadero o falso

¿Xunit, Juniy y otros Fw de pruebas unitarias son lo mismo que TDD?

9 de 9



☒ True

☐ False

☐ Sin respuesta

✗

4

✓

10

✗

1

⌚ Límite de tiempo de 20 s

Respuestas correctas

67 %

Tiempo promedio de res...

9.96s

Jugadores respondieron

14 of 15

¿Que son las pruebas unitarias?

¿Que es TDD?

¿Que es BDD?





xUnit.net

## Acerca de xUnit.net

xUnit.net es una herramienta gratuita de prueba unitaria, de código abierto y centrada en la comunidad para .NET Framework. Escrito por el inventor original de NUnit v2, xUnit.net es la última tecnología para pruebas unitarias C#, F#, VB.NET y otros lenguajes .NET. xUnit.net funciona con ReSharper, CodeRush, TestDriven.NET y Xamarin. Es parte de la Fundación .NET y opera bajo su [código de conducta](#). Está licenciado bajo [Apache 2](#) (una licencia aprobada por OSI).

*Siga en Twitter: [@xunit](#), [@jamesnewkirk](#), [@bradwilson](#), [@clairernovotny](#)*

*Siga en Mastodon: [@xunit@fosstodon.org](#)*

*Las discusiones se llevan a cabo en nuestro [sitio de discusiones](#).*

*El soporte de ReSharper es proporcionado y respaldado por [JetBrains](#).*

*El soporte de CodeRush es proporcionado y soportado por [DevExpress](#).*

*El soporte de NCrunch es proporcionado y respaldado por [Remco Software](#).*

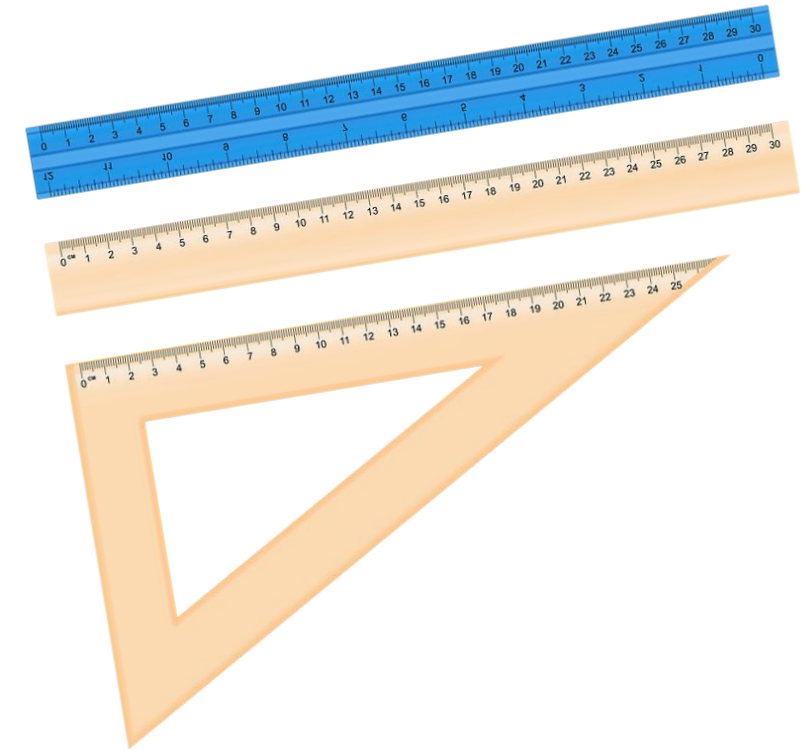
*El logotipo de xUnit.net fue diseñado por [Nathan Young](#).*



# Nombrar las pruebas unitarias

- Facilitan la lectura y entendimiento del código.
- Reducen el coste del mantenimiento del código.
- Agilizan el desarrollo cuando colaboran varias/os desarrolladoras/os.
- Facilitan agregar o modificar funcionalidades.

Una vez **elegida la convención**, una buena práctica es **dejar constancia** de los acuerdos sobre las convenciones sobre el proyecto.



# Nombrar las pruebas unitarias - Should...When

```
1  MIN_AGE_TO_BE_ADULT = 18
2
3  def is_adult(age):
4      assert isinstance(age, int), "Age should be a number"
5
6      return age < MIN_AGE_TO_BE_ADULT
```

is\_adult.py hosted with ❤ by GitHub

[view raw](#)

## “Should...When”

El enfoque que tiene esta estrategia es decir expresamente que se espera y cuando se espera, es una estrategia muy popular y es la precursora de la estrategia “Roy Osherove’s” que se comenta en el siguiente punto.

Patrón: `[UnitOfWork_ShouldStateUnderTest_WhenExpectedBehavior]`

Ejemplo:

```
1  # Prueba es adulto... debe ser falso.. cuando tiene menos de 18 años
2  def test_is_adult_should_to_be_false_when_age_is_less_than_18(self):
3      age = 17
4
5      adult = is_adult(age)
6
7      self.assertEqual(adult, False)
```

shoul\_when.py hosted with ❤ by GitHub

[view raw](#)

**Ventaja:** posee toda la información que necesitamos saber sobre el método de una forma estructurada.

**Desventaja:** La palabra “when” y “should” se vuelve repetitiva.

# Nombrar las pruebas unitarias - When...Should

```
1 MIN_AGE_TO_BE_ADULT = 18
2
3 def is_adult(age):
4     assert isinstance(age, int), "Age should be a number"
5
6     return age < MIN_AGE_TO_BE_ADULT
```

is\_adult.py hosted with ❤ by GitHub

[view raw](#)

## “When...Should”

Ésta variante se utiliza con el enfoque de que la lectura del test es menos forzado que la variante “Should...When”

Patrón: `[UnitOfWork_WhenExpectedBehavior_ShouldStateUnderTest]`

Ejemplo:

```
1 # Prueba es adulto... cuando es mejor de 18 años.. debería ser falso
2 def test_is_adult_when_age_is_less_than_18_should_to_be_false(self):
3     age = 17
4
5     adult = is_adult(age)
6
7     self.assertEqual(adult, False)
```

when\_should.py hosted with ❤ by GitHub

[view raw](#)

**Ventaja:** posee toda la información que necesitamos saber sobre el método de una forma estructurada, tiene una lectura mas natural que “when-should”

**Desventaja:** La palabra “should” y “when” se vuelve repetitiva.



# Nombrar las pruebas unitarias - When...Should Adaptado

```
1 MIN_AGE_TO_BE_ADULT = 18
2
3 def is_adult(age):
4     assert isinstance(age, int), "Age should be a number"
5
6     return age < MIN_AGE_TO_BE_ADULT
```

is\_adult.py hosted with ❤ by GitHub

[view raw](#)

“When...Should”

## Adaptación “Should...When” aplicado al Contexto

Podemos variar las palabras “Should” y “When” adaptándolas al contexto de la prueba, por ejemplo en vez de “when” usar un “with” o “without”

Ejemplo:

```
1 # Prueba es adulto... debe lanzar una excepción... sin edad
2 def test_is_adult_should_throws_exception_without_age(self):
3     age = '17'
4
5     with self.assertRaises(Exception):
6         is_adult(age)
```

should\_without.py hosted with ❤ by GitHub

[view raw](#)

# Nombrar las pruebas unitarias - “Unidad a testear”

```
1  MIN_AGE_TO_BE_ADULT = 18
2
3  def is_adult(age):
4      assert isinstance(age, int), "Age should be a number"
5
6      return age < MIN_AGE_TO_BE_ADULT
```

is\_adult.py hosted with ❤ by GitHub

[view raw](#)

## “Unidad a testear”

Se declara que es lo que se está probando, la característica que se va a probar forma parte del nombre de la prueba.

Patrón: `[UnitOfWork]`

Ejemplo:

```
1  # Prueba es adulto
2  def test_is_an_adult(self):
3      age = 18
4
5      adult = is_adult(age)
6
7      self.assertTrue(adult)
```

unit\_of\_work.py hosted with ❤ by GitHub

[view raw](#)

**Ventaja:** Refleja de forma concisa y breve el nombre de la unidad que se prueba.

**Desventaja:** No da detalles de la funcionalidad.

# Nombrar las pruebas unitarias - Característica que se está probando

## “Característica que se está probando”

```
1 MIN_AGE_TO_BE_ADULT = 18
2
3 def is_adult(age):
4     assert isinstance(age, int), "Age should be a number"
5
6     return age < MIN_AGE_TO_BE_ADULT
```

is\_adult.py hosted with ❤ by GitHub

[view raw](#)

Aquí se pretende mostrar la funcionalidad, este enfoque es asumiendo que el método que se prueba se conoce por otros medios que no sean el nombre del test.

Patrón: `[featureBeingTested]`

Ejemplo:

```
1 # Prueba es un menor si tiene menos de 18 años
2 def test_is_minor_when_there_are_less_than_18(self):
3     age = 17
4     expected_response = False
5
6     minor = is_adult(age)
7
8     self.assertEqual(minor, expected_response)
```

feature\_being\_tested.py hosted with ❤ by GitHub

[view raw](#)

**Ventaja:** posee toda la información que necesitamos saber sobre el método de una forma estructurada, tiene una lectura mas natural que “when-should”

**Desventaja:** La palabra “should” y “when” se vuelve repetitiva.

# Nombrar las pruebas unitarias - When...Expect

“When...Expect”

Muy utilizada en librerías como Google Test [\[visitar\]](#)

Patrón: `[WhenXXX_ExpectYYYY]`

Ejemplo:

```
1 # Cuando... es adulto... esperamos... que sea falso" (o algo así literal)
2 def test_when_is_less_than_18_expected_to_be_false(self):
3     age = 17
4
5     adult = is_adult(age)
6
7     self.assertEqual(adult, False)
```

when\_expect.py hosted with ❤ by GitHub

[view raw](#)



# Nombrar las pruebas unitarias - Given - When -Then

## “Given - When -Then”

```
1 MIN_AGE_TO_BE_ADULT = 18
2
3 def is_adult(age):
4     assert isinstance(age, int), "Age should be a number"
5
6     return age < MIN_AGE_TO_BE_ADULT
```

is\_adult.py hosted with ❤ by GitHub

[view raw](#)

Es una estrategia que se promueve desde el paradigma [\[Behavior Driven Development \(BDD\)\]](#), las pruebas son por comportamiento y se pretende que las pruebas se puedan generar incluso en lenguaje natural donde las pruebas se pretenden que las puedan hacer personas que no sepan programar.

Patrón: `[Given-When-Then]`

Ejemplo:

```
1 # En el escenario 1,...dado ... cuando ...entonces
2
3 Scenario 1: is Adult less than 18 can not buy
4 Given he is and adult,
5 When he is less than 18,
6 Then can not buy .
```

given\_when\_then.py hosted with ❤ by GitHub

[view raw](#)

- **Ventaja:** Personas no técnicas pueden escribir los test.
- **Desventaja:** Requiere el compromiso de la compañía para favorecer una estructura agilista.



**Ventaja:** Personas no técnicas pueden escribir los test.

**Desventaja:** Requiere el compromiso de la compañía para favorecer una estructura agilista.



# Nombrar las pruebas unitarias - Conclusiones

```
1.  public CustomerTest{  
2.      @Test  
3.      metodoBajoPruebas_escenario_resultadoEsperado() {  
4.          ...  
5.      }  
6.  }
```

*Nombramiento clásico de métodos de pruebas unitarias en Java*

- No seguir una estructura rígida en donde tengamos que integrar varias descripciones de un comportamiento complejo, lo ideal es tener la libertad de escribir una frase con más significado.
- Nombra la prueba como si se la estuvieras explicando a alguien no técnico/programador.
- No pongas el nombre de lo que estás probando, más bien describe el comportamiento que estás verificando en una frase sencilla.
- Cada prueba verifica un comportamiento, que es un hecho, por tanto se deben escribir denotando lo que es y no lo que podría o se desearía que fuera.



## **METODOLOGÍAS ÁGILES:**

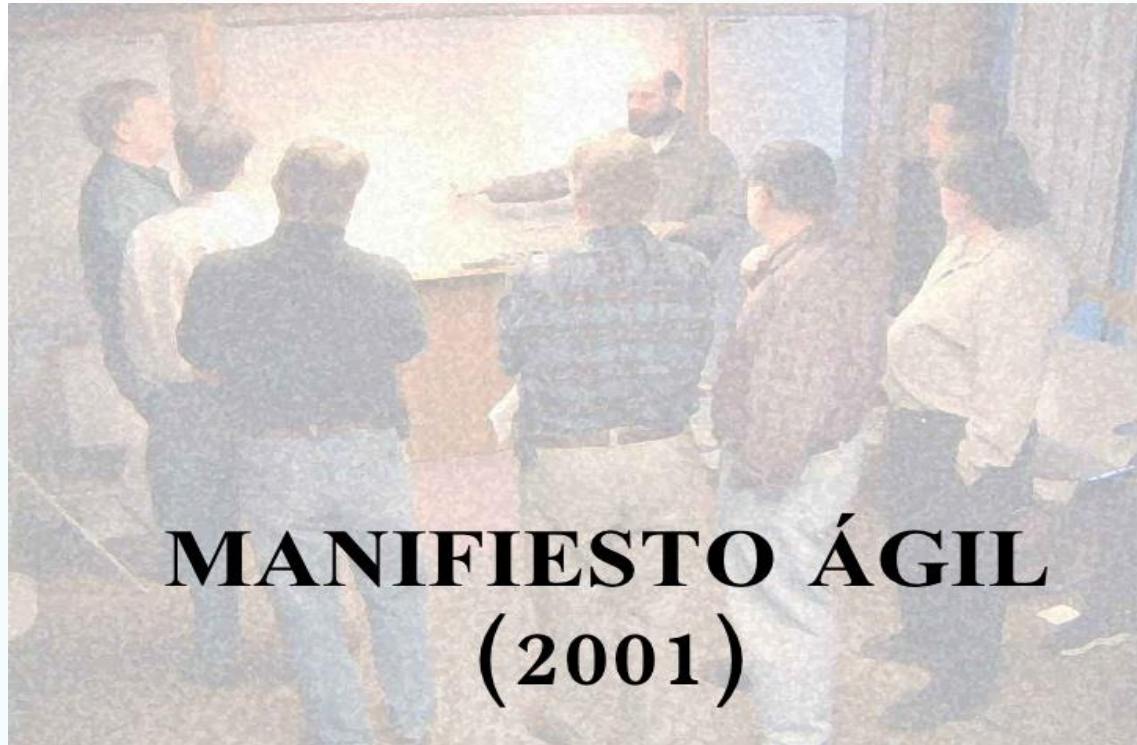
- **XP (1996)**
- **Scrum (1995)**
- **Lean Software Development**
- **DSDM (1995)**
- **Feature Driven Development (1999)**

# Metodologías Agiles - XP

Aplicando la simplicidad junto con la autoría colectiva del código y la programación por parejas se asegura que cuanto más grande se haga el proyecto, todo el equipo conocerá más y mejor el sistema completo.



# Metodologías Agiles – Manifiesto



Estamos descubriendo mejores maneras de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de esta experiencia hemos aprendido a valorar:

- **Individuos e interacciones** sobre *procesos y herramientas*
- **Software que funciona** sobre *documentación exhaustiva*
- **Colaboración con el cliente** sobre *negociación de contratos*
- **Responder ante el cambio** sobre *seguimiento de un plan*

Esto es, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.

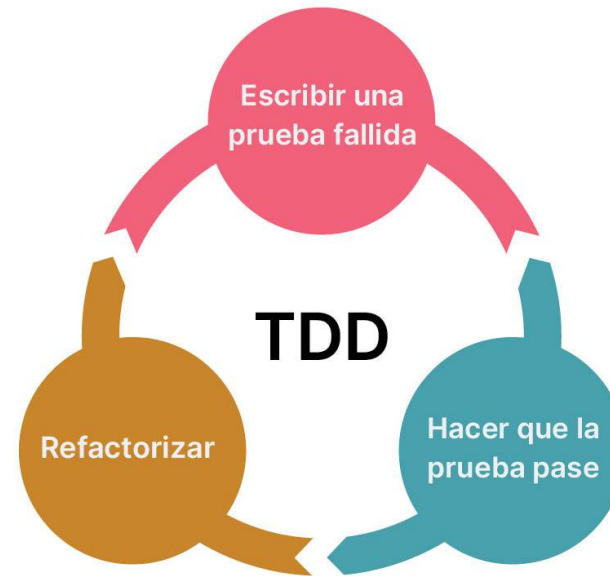
# Los orígenes - TDD

Creador XP, Junit, TDD



**Kent Beck**

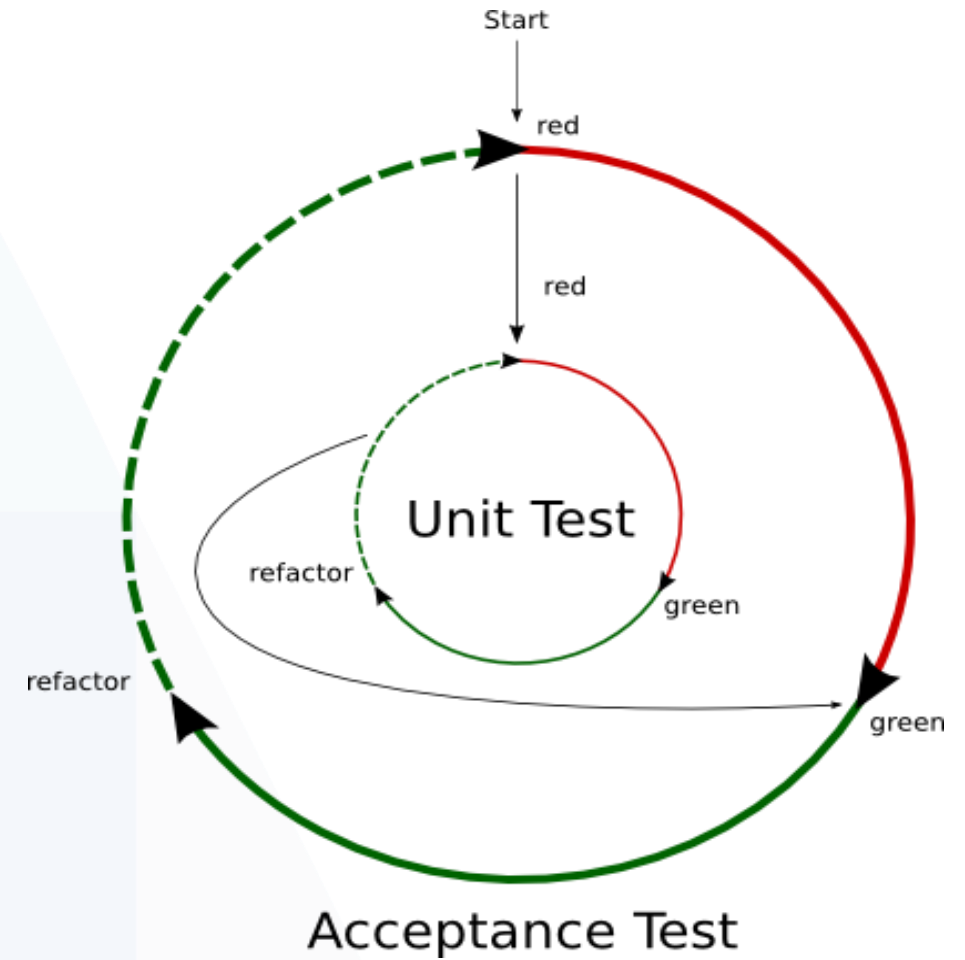
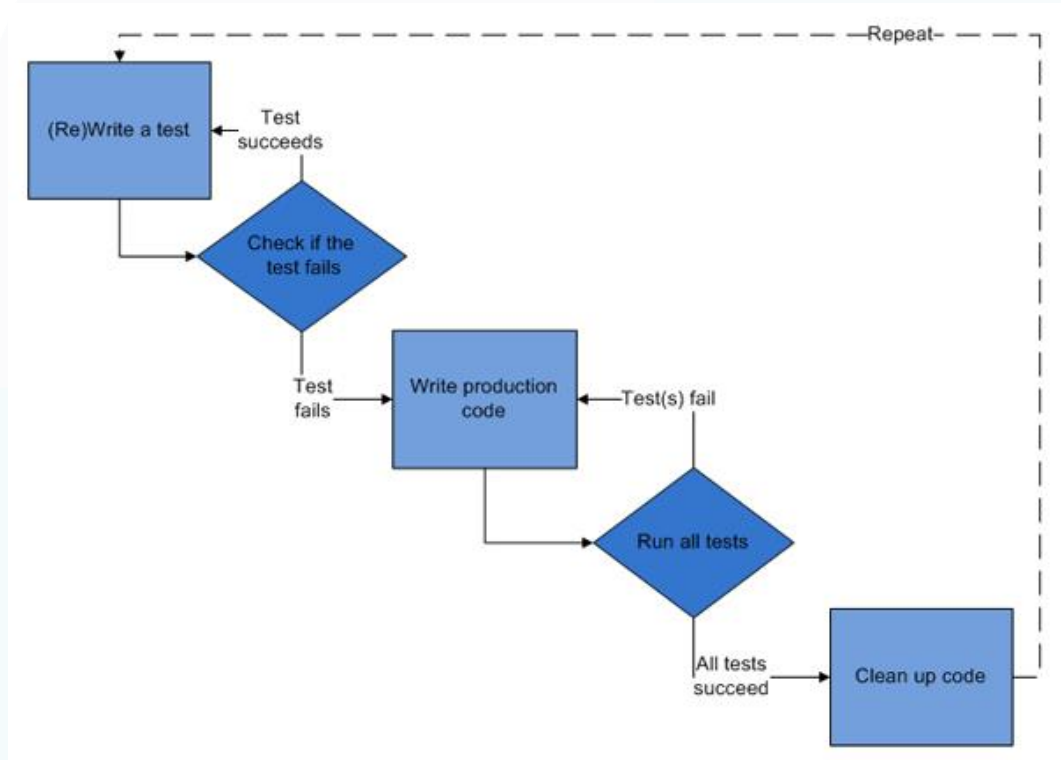
@KentBeck



TDD o Test-Driven Development (desarrollo dirigido por tests) es una práctica de programación que consiste en escribir primero las pruebas (generalmente unitarias), después escribir el código fuente que pase la prueba satisfactoriamente y, por último, refactorizar el código escrito. Con esta práctica se consigue, entre otras cosas, un código más robusto, más seguro, más mantenible y una mayor rapidez en el desarrollo.



# TDD



# Ventajas y Problemas de TDD

- Nos ayuda a pensar en cómo queremos desarrollar la funcionalidad.
  - Puede hacer software más modular y flexible.
  - Minimiza la necesidad de un "debugger".
  - Aumenta la confianza del desarrollador a la hora de introducir cambios en la aplicación.
- 
- Que no se pueda probar, no quiere decir que esta mal diseñado
  - ¿Cómo sé qué es lo que hay que implementar y lo que no?
  - ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente?
  - Dificultades a la hora de probar situaciones en las que son necesarios test funcionales o de integración, como pueden ser Bases de Datos o Interfaces de Usuario.
  - A veces se crean test innecesarios que provocan una falsa sensación de seguridad, cuando en realidad no están probando más que el hecho de que un método haga lo que dice que hace.
  - Los test también hay que mantenerlos a la vez que se mantiene el código, lo cual genera un trabajo extra.
  - Es difícil introducir TDD en proyectos que no han sido desarrollados desde el principio con TDD.
  - Para que sea realmente efectiva hace falta que todo el equipo de desarrollo haga TDD.



**NTT DATA**

**GRACIAS**

**FUTURE  
AT HEART**