Prácticas ágiles

Visión general de las metodologías Ágiles

Las metodologías ágiles **son enfoques para el desarrollo de productos** que se ajustan a los valores y principios descritos en el **Manifiesto Ágil** para el desarrollo de software.

Pretenden ofrecer el **producto adecuado**, con una **entrega incremental y frecuente** de pequeños trozos de funcionalidad, a través de **pequeños equipos multifuncionales autoorganizados**, lo que **permite la retroalimentación frecuente del cliente y la corrección del curso según sea necesario**.

Agile **pretende corregir los problemas** que plantean los enfoques tradicionales en **«cascada»**, que **consisten en la entrega de grandes productos en largos periodos de tiempo, durante los cuales los requisitos de los clientes cambian con frecuencia, lo que hace que se entreguen productos equivocados.**

Aplicación de la Metodología ágil

Durante su breve historia (99-2000), “Ágil” ha sido predominantemente un **enfoque para proyectos de desarrollo de software y aplicaciones informáticas.** Desde entonces, se extiende también a otros campos, especialmente en las industrias del conocimiento y los servicios.

La agilidad consiste en **responder al mercado y al cliente** rápidamente a sus **necesidades** y **demandas** y **siendo capaces de cambiar de dirección** según lo exija la situación.

Los métodos ágiles intentan **maximizar la entrega de valor** al cliente y **minimizar el riesgo** de crear productos que no satisfagan las necesidades del mercado o del cliente.

Para ello, dividen el tradicionalmente largo ciclo de entrega (típico de los «métodos en cascada» heredados) en **periodos más cortos**, llamados *sprints o iteraciones*. La iteración proporciona la cadencia para la entrega de un producto en funcionamiento al cliente, la obtención de retroalimentación y la realización de cambios basados en la retroalimentación.

Los métodos ágiles han tratado de **reducir los plazos de entrega** (entregar pronto, entregar a menudo) para garantizar que lleguen al mercado **trozos verticales más pequeños del producto**, lo que permite a los clientes dar su opinión antes y asegurarse de que el producto que finalmente reciben satisface sus necesidades.

Aplicación de la Metodología ágil (II)

Agile se ha convertido en un término general para una variedad de métodos y procesos de planificación, gestión y técnicos para gestionar proyectos, desarrollar software y otros productos y servicios de forma iterativa.

Entre estos métodos se encuentran Scrum, que es, con mucho, el método más extendido y popular para el software, XP (eXtreme Programming o Paired Programming) y, más recientemente, Kanban.

Los métodos ágiles también incluyen prácticas técnicas -la mayoría de las cuales se engloban bajo el término DevOps- que permiten la automatización de pruebas, la integración continua/entrega continua/despliegue (CI/CD) y, en general, un ciclo de entrega cada vez más reducido para el software y otros productos y servicios.

¿Qué es el Manifiesto Ágil?

El Manifiesto Ágil es una **declaración de valores y principios** fundamentales para el desarrollo de software.

El Manifiesto Ágil para el desarrollo de software se creó en 2001 y es una declaración de 4 reglas vitales y 12principios que sirven de guía para las personas en el desarrollo ágil de software.

Fue creado por 17 profesionales que ya practicaban métodos ágiles como XP, DSDM, SCRUM, FDD, etc, reunidos en Utah, convocados por Kent Beck.

Individuos e interacciones sobre procesos y herramientas

Software funcionando sobre documentación extensiva

Colaboración con el cliente sobre negociación contractual

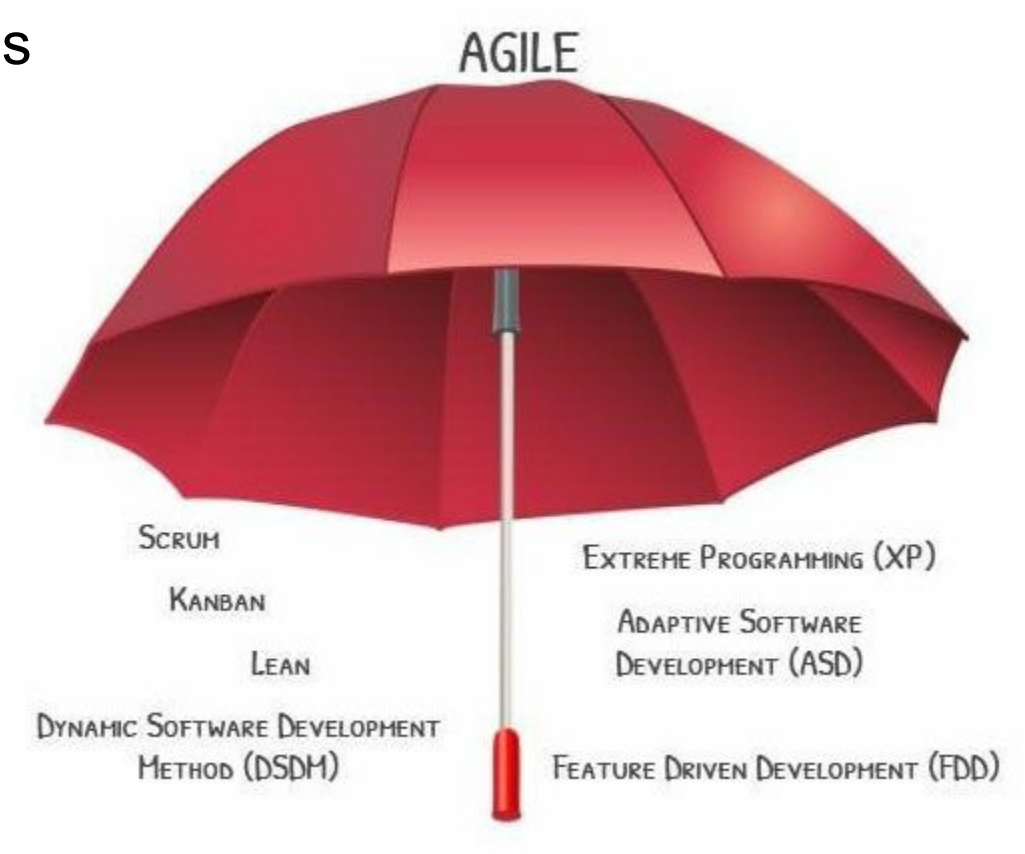
Respuesta ante el cambio sobre seguir un plan


Reflect
Tune
Adjust
WORK together DAILY
Business people AND Developers
Sustainable Development
CONSTANT PACE
HIGHEST PRIORITY
EARLY and CONTINUOUS DELIVERY
HARNESS change
Self Organising TEAMS
changing REQUIREMENTS are WELCOME
MANIFESTO
AGILE MANIFESTO
satisfy
CUSTOMER
SIMPLICITY
DELIVER frequently
SHORTER Timescale
CONTINUOUS ATTENTION
Technical excellence
GOOD DESIGN
BUILD PROJECTS around MOTIVATED individuals
environment
trust
support
FACE to FACE is MOST EFFICIENT EFFECTIVE
conveying INFORMATION
IT WORKS
Measure of PROGRESS
by WWW.lynnecazaly.com
JULY 2016

Principales metodologías ágiles

Ágil es un término que engloba varios métodos y prácticas.
Veamos algunas de las metodologías más populares:

- Scrum
- Programación extrema (XP)
- Desarrollo de software adaptativo (ASD)
- Método de desarrollo dinámico de software (DSDM)
- Desarrollo impulsado por las características (FDD)
- Kanban
- Desarrollo orientado al comportamiento (BDD)



Prácticas Ágiles - TDD -

Es una metodología de desarrollo cuyo objetivo es crear primero las pruebas y luego escribir el software.

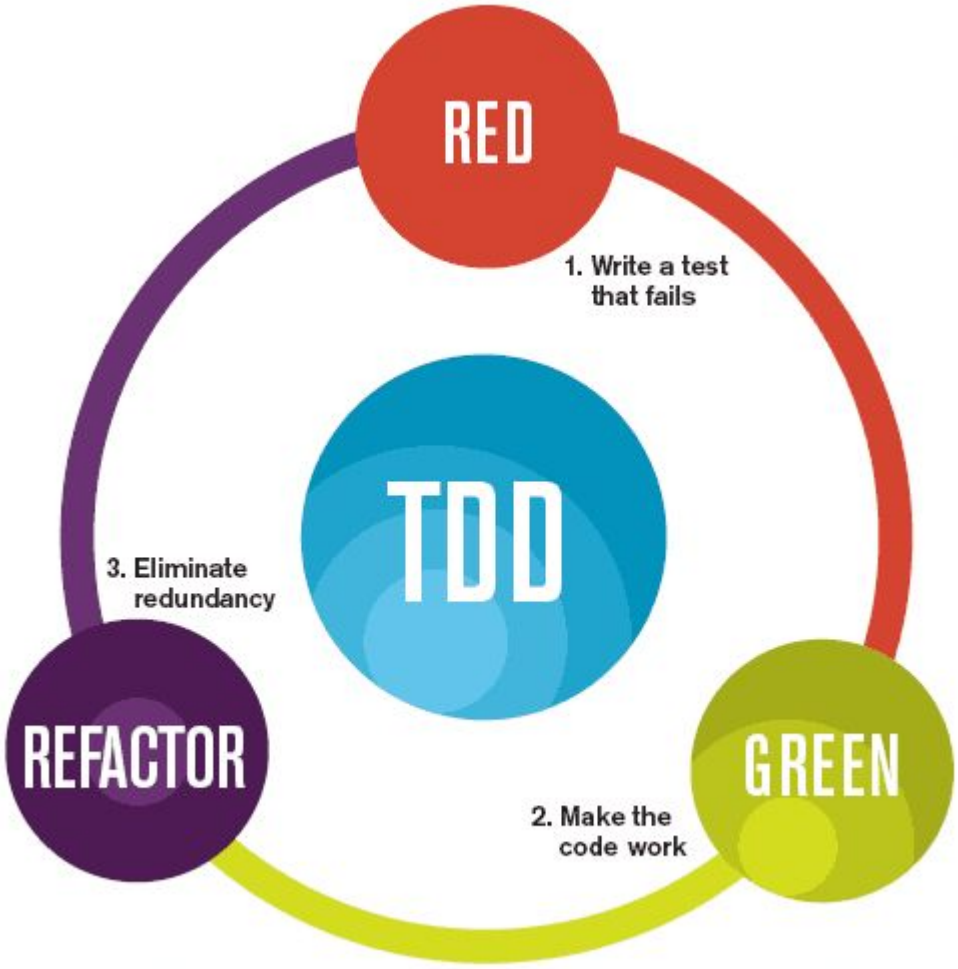
Sus siglas en Inglés son: Test **Driven Development** y en español significa: **Desarrollo guiado por pruebas.**

Fue a finales de los años 80 donde se comenzó a utilizar esta metodología de desarrollo.

Para el uso del TDD se deben combinar 2 metodologías:

- **Test-first development** (escribir las pruebas primero) y
- **Refactoring** (refactorización de código)

Para esto, se usa un ciclo de desarrollo que consta de 3 partes principales:



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

TDD

El **Desarrollo Dirigido por Pruebas** es un proceso en el que se escriben las pruebas antes de escribir el código. Y cuando todas las pruebas pasan, *se limpia la cocina*: **se mejora el código**.

Sin embargo, el Desarrollo Dirigido por Pruebas no es sobre las pruebas

La premisa detrás del desarrollo dirigido por pruebas, según Kent Beck, es que **todo el código debe ser probado y refactorizado continuamente**. Eso suena como si se tratara de pruebas, así que ¿qué pasa?

Bueno, **las pruebas** que escribes en TDD **no son el punto, sino el medio**.

El punto es que al escribir pruebas primero y esforzarse por mantenerlas fáciles de escribir, estás haciendo tres cosas importantes.

1. Está **creando documentación, especificaciones vivas y nunca obsoletas** (es decir, documentación).
2. Está (**re)diseñando su código para hacerlo y mantenerlo fácilmente comprobable**. Y eso lo hace limpio, sin complicaciones y fácil de entender y cambiar.
3. Está c**reando una red de seguridad para hacer cambios con confianza**.

Beneficios del Desarrollo Dirigido por Pruebas

- Notificación temprana de errores.
- Diagnóstico sencillo de los errores, ya que las pruebas identifican lo que ha fallado.

Lo que todo esto significa para su negocio, es:

- Mejora su factor de bus, ya que el conocimiento no está solo en las cabezas y facilita la incorporación de nuevas contrataciones.
- Reduce el coste de las mejoras. Mantener el código limpio es también la forma de minimizar el riesgo de complicaciones accidentales. Y eso significa que podrá mantener un ritmo constante en la entrega de valor.
- Con la red de seguridad, los desarrolladores están más dispuestos a fusionar sus cambios y a incorporar los de otros desarrolladores. Y lo harán más a menudo. Y entonces el desarrollo basado en el tronco y la integración, entrega y despliegue continuos pueden despegar realmente.
- Disminuye el número de errores que se «escapan» a la producción y eso reduce los costes de soporte.

Las Reglas del TDD

Bob Martin expuso las reglas de TDD en el capítulo 5 Desarrollo Dirigido por Pruebas de su libro **“The Clean Coder”**.

1. No se permite escribir ningún código de producción a menos que sea para hacer pasar una prueba unitaria que falla.
2. No se permite escribir más de una prueba unitaria que sea suficiente para que falle; y los fallos de compilación son fallos.
3. No se permite escribir más código de producción que el suficiente para pasar la prueba unitaria que falla.

Estas reglas están pensadas para hacerle la vida más fácil.

La intención de las reglas es mantener las cosas enfocadas en cada fase y evitar que se caiga en agujeros de conejo. Por experiencia, eso ayuda mucho a mantener las cosas claras en su cabeza.

Las Reglas del TDD

- **Durante la fase roja (escritura de la prueba), trabaje sólo en el código de prueba.**
- **Una prueba que falla es buena. Al menos si es la que estás trabajando. Todas las demás deben ser verdes.**
- **Durante la fase verde (hacer que la prueba pase), sólo trabaja en el código de producción que hará que la prueba pase y no refactorices nada.**
- **Si la prueba que acaba de escribir falla significa que su implementación necesita trabajo. Si otras pruebas fallan, ha roto la funcionalidad existente y necesita retroceder.**
- **Durante la fase azul (refactorización), sólo refactoriza el código de producción y no haga ningún cambio funcional.**
- **Cualquier prueba que falle significa que ha roto la funcionalidad existente. O bien no ha completado la refactorización, o necesitas dar marcha atrás**

Ejemplo de uso del TDD

Hagamos todo lo anterior un poco más concreto con un ejemplo. Supongamos que se le encomienda la tarea de crear un método que convierta los números decimales en romanos.

Paso 1: Fase roja, escriba una prueba.

El decimal 1 debe devolver «I».

```
[Fact]
0 references | Run Test | Debug Test
public void When_1_Then_I()
{
    var roman = Romanizer.FromDecimal(1);
    roman.Should().Be("I");
}
```

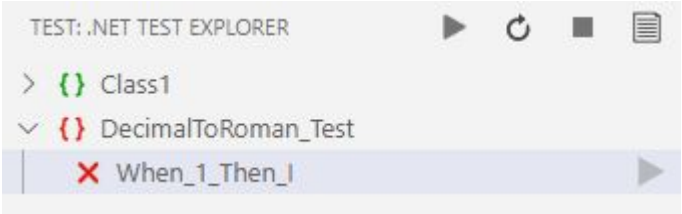
Ejecutar la prueba no es posib **er no existe todavía y eso significa errores de compilaci**

Ejemplo de uso del TDD

Paso 2: Fase verde, hacer que la prueba pase

Añada la clase Romanizer y dele un método FromDecimal que tome un entero y devuelva una cadena.

```
namespace TDD_Example_Bare
{
    1 reference
    public static class Romanizer
    {
        1 reference
        public static string FromDecimal(int number)
        {
            return null;
        }
    }
}
```




```
public static string FromDecimal(int number)
{
    return "I";
}
```

Ejemplo de uso del TDD

Paso 3: Fase azul, refactorizar (No hay mucho que refactorizar todavía, así que a la siguiente prueba.)

Paso 4: Fase roja, escribir una prueba

```
[Fact]
0 references | Run Test | Debug Test
public void When_2_Then_II()
{
    var roman = Romanizer.FromDecimal(2);
    roman.Should().Be("II");
}
```

Ejemplo de uso del TDD

Paso 5: Fase verde, hacer que la prueba pase

De nuevo, escribe el código más sencillo que haga que la prueba pase.

```
2 references
public static string FromDecimal(int number)
{
    if (number == 1)
    { return "I"; }
    else if (number == 2)
    { return "II"; }

    throw new ArgumentException();
}
```

Ejemplo de uso del TDD

Paso 6: Fase azul, refactorización

Paso 7: Fase roja, escribir una prueba

El decimal 3 debería devolver «III».

```
[Fact]
0 references | Run Test | Debug Test
public void When_3_Then_III()
{
    var roman = Romanizer.FromDecimal(3);
    roman.Should().Be("III");
}
```

(Esa construcción if-else-if no es muy elegante, pero dos casos no merecen ser refactorizados todavía, así que a la siguiente prueba)

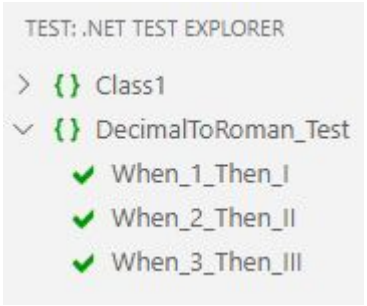
Ejemplo de uso del TDD

Paso 8: Fase verde, hacer que la prueba pase

De nuevo, el código más sencillo para hacer pasar la prueba.

```
3 references
public static string FromDecimal(int number)
{
    if (number == 1)
    { return "I"; }
    else if (number == 2)
    { return "II"; }
    else if (number == 3)
    { return "III"; }

    throw new ArgumentException();
}
```



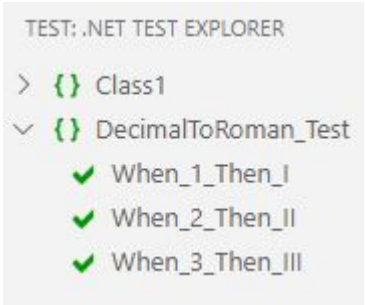
(Sí, todo verde, así que por fin podemos hacer algo con esa construcción if-else-if, porque 3 casos sí merecen una refactorización ya que ahora se aplica la Regla de 3)

Ejemplo de uso del TDD

Paso 9: Fase azul, refactorización

El patrón está claro. Se necesita tantos «I» como el número que se ha pasado.

```
3 references
public static string FromDecimal(int number)
{
    var result = "";
    for (int i = 0; i < number; i++)
    {
        result += "I";
    }
    return result;
}
```



Ejecute todas las pruebas para asegurarse de que cada una de ellas sigue pasando.

Por supuesto, sabiendo cómo funcionan los números romanos, el patrón no se mantendrá.

Pero romperse la cabeza con un algoritmo de antemano no es el camino a seguir. Lo más probable es que usted termine ajustandolo para cada prueba que añada. Y se pondrá de mal humor cuando cada retoque haga fallar diferentes pruebas.

Cuando se escribe el código más simple y se refactoriza, se hace crecer el algoritmo a medida que se avanza. Una manera mucho mejor y más fácil de hacerlo.

Reglas de Refactorización

Cuando se refactoriza, no se hace al azar. Sino que sigues un proceso estructurado para limpiar el código.

El propósito de la refactorización es mejorar la extensibilidad de su código mediante

- mejorar la legibilidad
- facilitar la realización de cambios
- reducir la complejidad
- mejorar la arquitectura interna (modelo de objetos) y hacerla más expresiva

Lo que es realmente importante es que la refactorización es lo que consigue un código limpio y sin complicaciones que es fácil de entender, cambiar y probar.

¿Por qué es tan difícil practicar TDD?

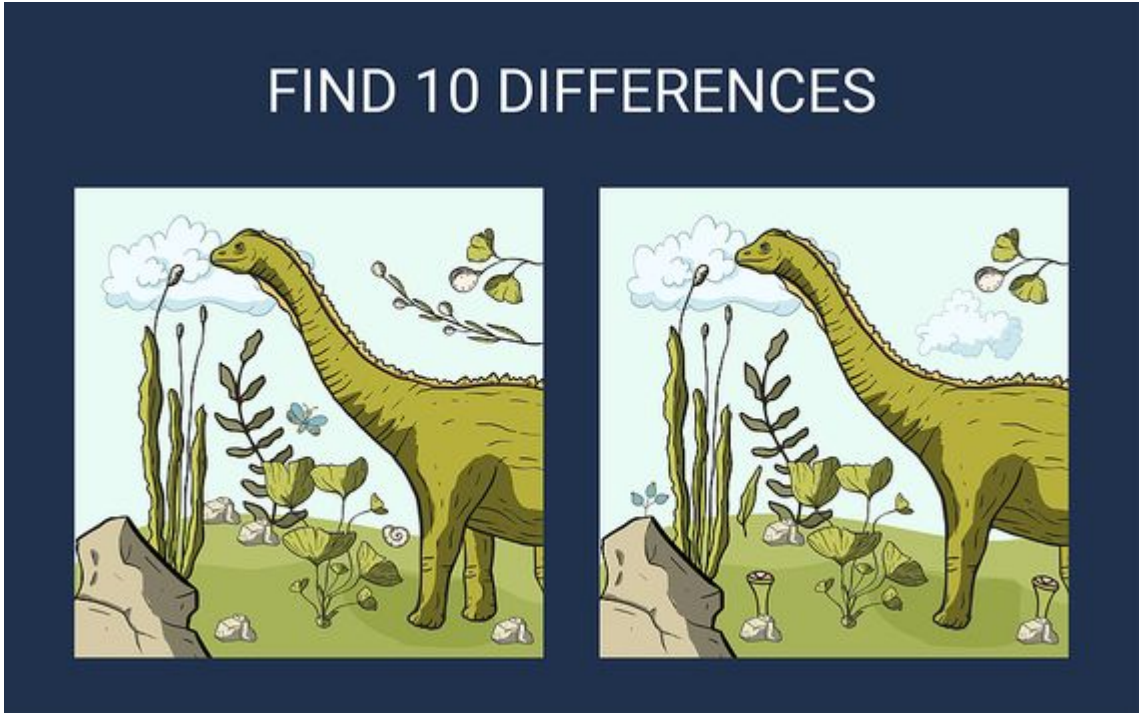
- Porque hay que pensar en lo que se quiere conseguir con el código y en cómo protegerlo para que no se rompa (probarlo).
- Porque tiene una curva de aprendizaje muy pronunciada. Es necesario aprender los principios y patrones de diseño para crear un código limpio y cómo refactorizarlo para mantenerlo así.
- Porque el código se defiende. El código existente que no está bajo prueba te pilla entre la espada y la pared. Necesita refactorizarlo para ponerlo bajo prueba y necesitas pruebas para refactorizarlo.
- Porque se experimentan los costes de TDD inmediatamente y el coste de no hacerlo mucho más tarde. El atractivo de dejarlo pasar es fuerte. Usted sabe que va a pagar el precio cuando los informes de errores comienzan a inundar.

¿Cuáles son los errores más comunes?

Se puede fracasar en la práctica de TDD de muchas maneras:

- No seguir el enfoque de «primero la prueba».
- No refactorizar todo el tiempo.
- Escribir más de una prueba a la vez.
- No ejecutar las pruebas con frecuencia, perdiendo la retroalimentación temprana de las mismas.
- Escribir pruebas que son lentas. Todo el conjunto debería completarse en minutos o incluso en segundos.
- Usar sus pruebas unitarias para hacer pruebas de integración. No hay nada malo en utilizar su marco de pruebas unitarias para ejecutar pruebas de integración. Pero las pruebas de integración son, por naturaleza, lentas, así que debes ponerlas en su propio conjunto de pruebas.
- Escribir pruebas sin aserciones.
- Escribir pruebas para el código trivial, como los accesores y las vistas sin lógica.

¿Cuál es la diferencia entre TDD y BDD?

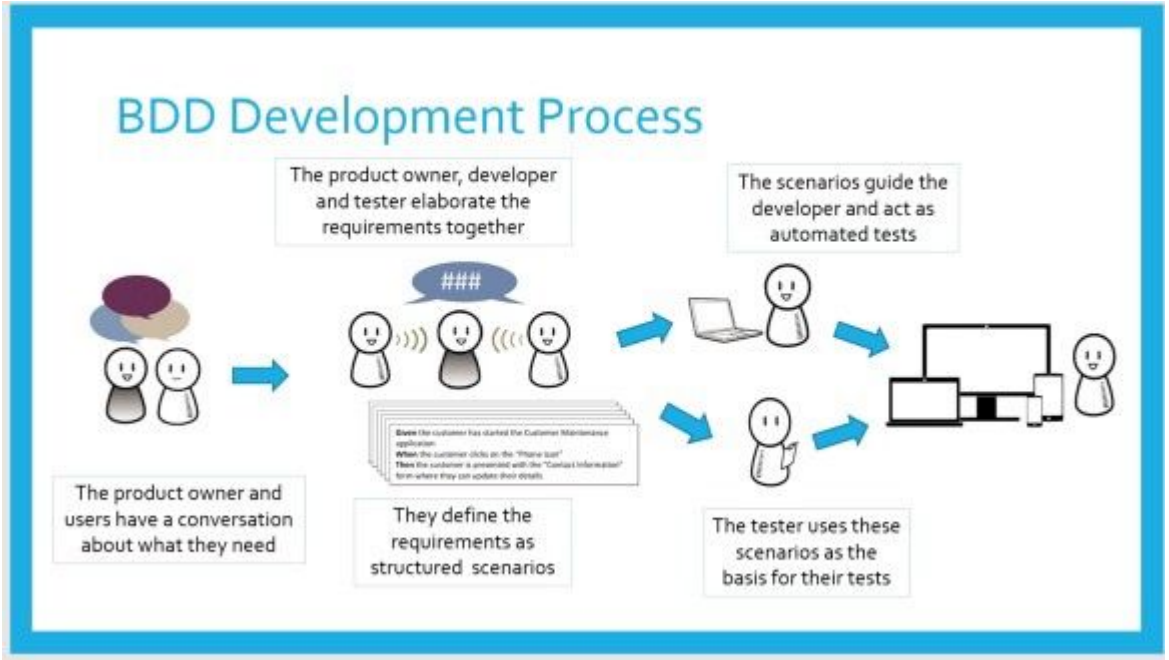


¿Cuál es la diferencia entre TDD y BDD?

El desarrollo dirigido por el comportamiento y el desarrollo dirigido por las pruebas son similares y diferentes al mismo tiempo.

- Ambos emplean enfoques que dan prioridad a las pruebas, pero no se centran en ellas.
- BDD trata de mejorar la colaboración y la comunicación entre desarrolladores, probadores y profesionales de la empresa. Para asegurar que el software cumple tanto con los objetivos de negocio como con los requisitos del cliente.
- TDD es sobre el diseño y las especificaciones a nivel de código.
- BDD trabaja en el nivel de la aplicación y los requisitos. TDD se centra en el nivel del código que implementa esos requisitos.
- TDD es, o puede ser utilizado como la fase de «Hacer que las pruebas pasen» de BDD.
- En TDD se puede, pero no es necesario, utilizar técnicas de BDD hasta el nivel más pequeño de abstracción.
- BDD no tiene una fase de refactorización como TDD.

BDD (Behavior Driven Development)



BDD (Behavior Driven Development)

BBD es un **proceso diseñado para ayudar a la gestión y la entrega de proyectos de desarrollo de software mejorando la comunicación entre los ingenieros y los profesionales de la empresa**. De este modo, el BDD garantiza que todos los proyectos de desarrollo **se centren en la entrega de lo que realmente necesita la empresa y** en el **cumplimiento de todos los requisitos del usuario.**

El Desarrollo Dirigido por el Comportamiento **no es sobre las pruebas**

Practicar BDD significa que va a especificar y ejecutar pruebas. Pero las pruebas no son el objetivo.

BDD (Behavior Driven Development)

La idea central del diseño orientado al comportamiento es que ninguna persona o campo tiene la respuesta completa a nada.

Se necesitan tres puntales: profesionales de la empresa, desarrolladores y probadores, para obtener buenas respuestas.

Y lo que es más, se necesita que colaboren. Porque es el ida y vuelta entre personas con diferentes estilos cognitivos y diferentes perspectivas y antecedentes, lo que produce la magia. Magia que le aporta soluciones mejores, más sencillas y más valiosas.

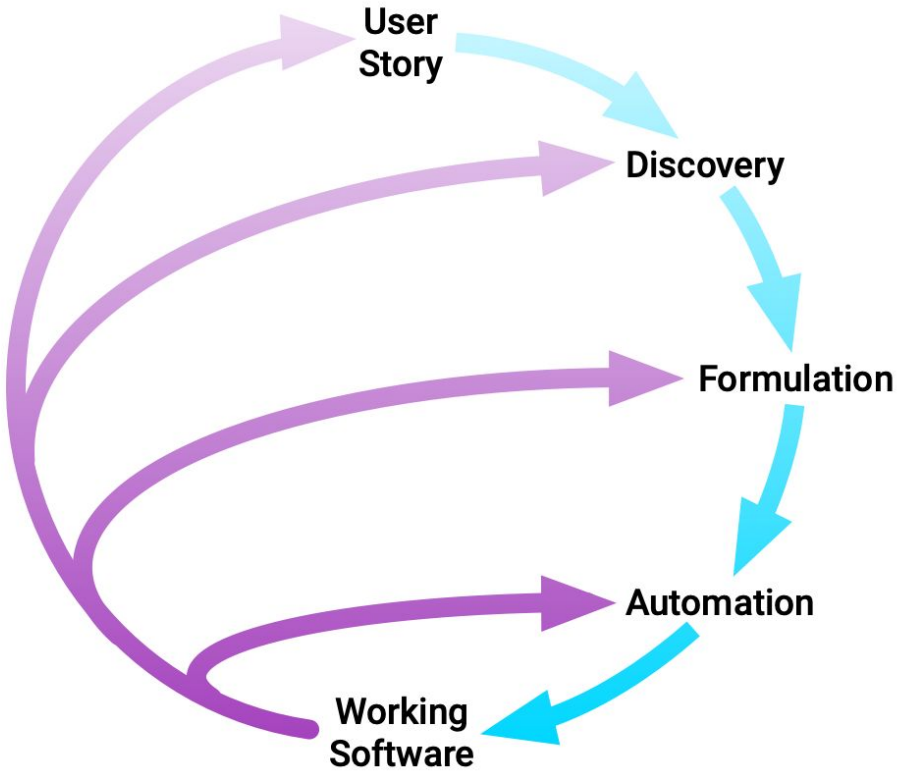
Lo que cada uno de los tres puntales aporta a la mesa:

- Los profesionales de los negocios aportan las necesidades de la empresa y los requisitos de los clientes. Están en la mejor posición para decidir cuán deseable es una historia en comparación con otras.
- Los desarrolladores son los expertos en software y oportunidades tecnológicas. Están en la mejor posición para calibrar lo fácil o difícil que será implementar una historia.
- Los probadores son indispensables por su capacidad de detectar, casi instintivamente, los obstáculos y las lagunas. Identificarán los casos límite antes de que causen problemas más adelante.

¿Cuáles son los beneficios del BDD?

- Poner en primer plano los objetivos empresariales y los requisitos de los clientes.
- Pautas para estructurar la conversación entre los Three Amigos. Esto hace que éstas sean más eficaces y productivas.
- Definir los criterios de aceptación antes de comenzar el desarrollo.
- Evitar el esfuerzo en características que no contribuyen a los objetivos de negocio.
- Evitar interpretaciones erróneas que lleven a rehacer el trabajo más adelante.
- Lenguaje omnipresente. Un lenguaje específico del dominio que se utiliza en todas partes en el software.
- Especificaciones que las personas no técnicas pueden entender fácilmente.
- Especificaciones ejecutables.
    - Pruebas de aceptación automatizadas que proporcionan una retroalimentación rápida para mantener la implementación en el camino.
    - Especificaciones vivas que nunca se desactualizan.
    - Documentación técnica y de usuario final generada automáticamente, que tampoco se desactualiza.

Las Fases del Desarrollo Dirigido por el Comportamiento



Las Fases del Desarrollo Dirigido por el Comportamiento

En BDD se pasa por tres fases para cada historia que se quiera implementar.

Estas fases son:

- **Descubrimiento**. Aquí se explora una historia en una conversación estructurada. Tiene dos objetivos. Uno es asegurar que la historia contribuirá a los resultados del negocio. Por ejemplo, con el método de los cinco porqués. El otro objetivo es asegurar la comprensión compartida de lo que se necesita, esbozando ejemplos concretos de escenarios específicos.
- **Formulación**. Aquí es donde se reformulan (formulan) los ejemplos en un lenguaje estructurado y se convierten en especificaciones ejecutables.

Patrón “given-when-then”:

el escenario (dado)— el estado inicial, por ejemplo «el usuario está conectado»,

el evento (cuando)— lo que ocurre, por ejemplo «el usuario pulsa el botón de cierre de sesión», y

el resultado (entonces)— la respuesta esperada, por ejemplo «se muestra la página de inicio de sesión».

- **Automatización**. Aquí es donde se convierten las especificaciones ejecutables en pruebas de aceptación automatizadas. Utilizando herramientas como Cucumber, se trata de conectar la herramienta con el software.

Software de trabajo: Las pruebas de aceptación guían su trabajo de implementación del software.

Un ejemplo de BDD

Título: Bloquear a otros usuarios.

Narración: Como miembro, quiero bloquear a los usuarios para que no me molesten más.

Descubrimiento:

La historia parece bastante sencilla, pero los interrogantes surgen en cuanto se empieza a pensar en ella:

- ¿Qué resultado comercial me sirve?
- ¿Qué significa bloquear? ¿Qué implica eso? ¿Qué se necesita para que eso ocurra?
- ¿Qué significa «no me molestes más»? ¿Qué aspecto tiene eso? ¿Qué tiene que pasar o no pasar?
- un largo etc.

Después de hablar con el cliente, resulta que simplemente no quiere ver las actualizaciones de un usuario bloqueado en su feed de actividad.

“Como miembro, quiero silenciar a los usuarios, para que mi feed de actividad no muestre sus actualizaciones”. Una cuestión mucho más sencilla de abordar y de mucho menor alcance.

Un ejemplo de BDD

Formulación

Ahora se escribe cada ejemplo en la notación estructurada que puede ser leída y ejecutada por el software.

- Dado que he silenciado a |algún usuario|, cuando visito mi feed de actividad, el feed no debería contener actualizaciones de |algún usuario|.
- Dado que he silenciado a algún usuario, cuando éste envíe una actualización, no debería añadirse a mis notificaciones.

Cómo no sacar el máximo partido al BDD

Los errores más comunes en el desarrollo dirigido al comportamiento son:

- Dejar de lado a los probadores en las fases de Descubrimiento y *Formulación*.
- Asumir que escribir los ejemplos es lo que cuenta.
- Renunciar a la fase de *Descubrimiento* porque se cree que ya se sabe lo que hay que hacer.

Qué es la Prueba Dirigida por el Comportamiento (Behavior Driven Testing)

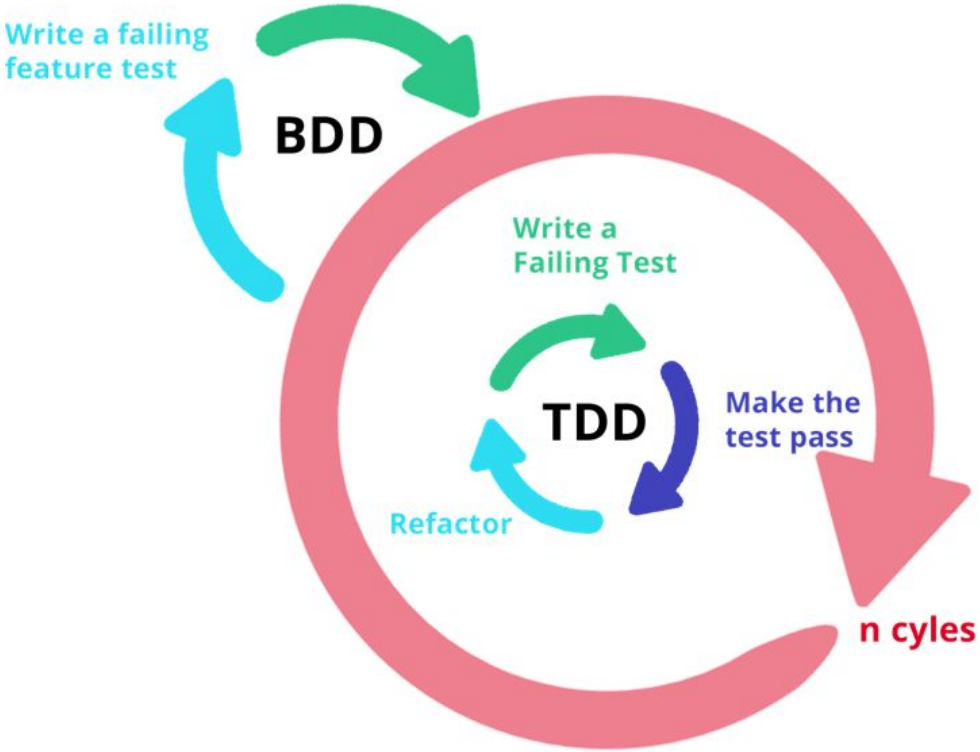
Las Pruebas Dirigidas por el Comportamiento (BDT) son un proceso menos conocido que conduce a las pruebas de aceptación automatizadas.

Promueve la colaboración entre personas técnicas y no técnicas.

Utiliza un lenguaje natural formalizado para escribir pruebas que pueden ser fácilmente revisadas y verificadas por profesionales de la empresa.

BDT se centra más en lo que los usuarios hacen con el software. En lugar de los criterios de aceptación que verifican la implementación de la lógica empresarial.

¿Cuál es la diferencia entre TDD y BDD?



¿Cuál es la diferencia entre TDD y BDD?

El desarrollo dirigido por el comportamiento y el desarrollo dirigido por las pruebas son similares y diferentes al mismo tiempo.

- Ambas **emplean enfoques que dan prioridad a las pruebas, pero no se trata de pruebas.**
  - **BDD** consiste en **mejorar la colaboración y la comunicación** entre desarrolladores, probadores y profesionales de la empresa. Para que el software cumpla tanto los objetivos de negocio como los requisitos del cliente.
  - **TDD** trata del **diseño y las especificaciones a nivel de código**.
- **BDD** trabaja a nivel de **aplicación y requisitos**. **TDD** se centra en el **nivel del código** que implementa esos requisitos.
- **TDD** es, o puede ser utilizado como la fase «Hacer que las pruebas pasen» de BDD.
- En TDD se puede, pero no es necesario, utilizar técnicas de BDD hasta el nivel más pequeño de abstracción
- **BDD no tiene una fase de refactorización** como TDD.

Ejemplo BDD (Feature - Given - When - Then)

BDD (*Behavior Driven Development)*, es una estrategia de desarrollo dirigido por comportamiento, que ha evolucionado desde TDD (Test Driven Development), aunque no se trata de una técnica de testing.

A diferencia de TDD, BDD se define en un idioma común entre todos los *stakeholders*, lo que mejora la comunicación entre equipos tecnológicos y no técnicos. Tanto en TDD como en BDD, las pruebas se deben definir antes del desarrollo, aunque en BDD, las pruebas se centran en el usuario y el comportamiento del sistema, a diferencia del TDD que se centra en funcionalidades.

¿Que debo tener en cuenta antes de implementar BDD?

- Cada requisitos debe convertirse en historias de usuario, defiendo ejemplos concretos.
- Cada ejemplo debe ser un escenario de un usuario en el sistema.
- Ser consciente de la necesidad de definir "la especificación del comportamiento de un usuario" en lugar de "la prueba unitaria de una clase".
- Comprender la fórmula ‘Given-When-Then’ u otras como las historias de usuario ‘Role-Feature-Reason'.

‘Given-When-Then’ como lenguaje común con BDD

Para definir los casos BDD para una historia de usuario se deben definir bajo el patrón ‘Given-When-Then’, que se define como:

- Given ‘dado’: Se especifica el escenario, las precondiciones.
- When ‘cuando’: Las condiciones de las acciones que se van a ejecutar.
- Then ‘entonces’: El resultado esperado, las validaciones a realizar.

Un ejemplo práctico sería:

- Given: Dado que el usuario no ha introducido ningún dato en el formulario.
- When: Cuando hace clic en el botón Enviar.
- Then: Se deben mostrar los mensajes de validación apropiados.

‘Role-Feature-Reason’ como lenguaje común con BDD

Este patrón también se utiliza en BDD para ayudar a la creación de historias de usuarios. Esta se define como:

- As a ‘Como’: Se especifica el tipo de usuario.
- I want ‘deseo’: Las necesidades que tiene.
- So that ‘para que’: Las características para cumplir el objetivo.

Un ejemplo práctico de historia de usuario sería: - Como cliente interesado, deseo ponerme en contacto mediante el formulario, para que atiendan mis necesidades.

Herramientas de definición BDD

La herramienta más destacada, basada en el patrón ‘Given-When-Then’ es Cucumber, un framework de test con soporte BDD. En Cucumber, las especificaciones de BDD están escritas en lenguaje Gherkin, basado en ‘Given-When-Then’. En otras palabras, Gherkin presenta el comportamiento de la aplicación, a partir de la cual Cucumber puede generar los casos de prueba de la aplicación.

Hay muchas otras herramientas, tantas como lenguajes de programación:

- Java: JBehave, JDave, Instinct, beanSpec
- C: CSpec
- C#: NSpec, NBehave
- .NET: NSpec, NBehave, SpecFlow
- PHP: PHPSpec, Behat
- Ruby: RSpec, Cucumber
- JavaScript: JSSpec, jspec
- Phython: Freshen

Ventajas de BDD (Behavior Driven Development)

Si estás pensando en implementar BDD en tus desarrollos, aquí te dejo algunas ventajas que beneficiarán al equipo de trabajo:

1. Ya no estás definiendo 'pruebas', sino que está definiendo 'comportamientos'.
2. Mejora la comunicación entre desarrolladores, testers, usuarios y la dirección.
3. Debido a que BDD se específica utilizando un lenguaje simplificado y común, la curva de aprendizaje es mucho más corta que TDD.
4. Como su naturaleza no es técnica, puede llegar a un público más amplio.
5. El enfoque de definición ayuda a una aceptación común de las funcionalidades previamente al desarrollo.
6. Esta estrategia encaja bien en las metodologías ágiles, ya que en ellas se especifican los requisitos como historias de usuario y de aceptación.

Ejemplo de BDD (Gherkin)

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  Scenario: Búsqueda simple en Google
    Given un navegador web en la página de Google
    When se introduce la palabra de búsqueda "pingüino"
    Then se muestra el resultado de "pingüino"
```

Definición adicional de patrones en BDD

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  Scenario: Búsqueda simple en Google
    Given un navegador web en la página de Google
    When se introduce la palabra de búsqueda "pingüino"
    Then se muestra el resultado de "pingüino"
    And los resultados relacionados incluyen "Pingüino emperador"
    But los resultados relacionados no incluyen "ping pong"
```

Documentación adicional en BDD

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  Scenario: Búsqueda simple en Google
    Given un navegador web en la página de Google
    When se introduce la palabra de búsqueda "pingüino"
    Then se muestra el resultado de "pingüino"
    And la página de resultados muestra el texto de Wikipedia
      """
      Nombre científico: Spheniscidae
      Clase: Aves
      """
```

Antecedentes y escenarios en BDD

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  Background:
    Given un navegador web en la página de Google

  Scenario: Búsqueda simple en Google
    When se introduce la palabra de búsqueda "pingüino"
    Then se muestra el resultado de "pingüino"

  Scenario: Búsqueda simple en Google
    When se introduce la palabra de búsqueda "panda"
    Then se muestra el resultado de "panda"
```

Esquemas de escenario en BDD

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  Scenario Outline: Búsqueda simple en Google
    Given un navegador web en la página de Google
    When se introduce la palabra de búsqueda "<frase>"
    Then se muestra el resultado de "<frase>"
    And los resultados relacionados "<relacionado>"

    Examples: Animales
      | frase     | relacionado          |
      | pingüino  | Pingüino emperador   |
      | panda     | Panda gigante        |
      | elefante  | Elefante Africano    |
```

Tags en BDD

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  @automatización @google @pingüino
  Scenario: Búsqueda simple en Google
    Given un navegador web en la página de Google
    When se introduce la palabra de búsqueda "pingüino"
    Then se muestra el resultado de "pingüino"
```

Comentarios en BDD

```
Feature: Búsqueda en Google
  Como usuario web, quiero buscar en Google para poder responder mis dudas.

  # Test ID: 12345
  # Autor: sergio@itdo.com
  Scenario: Búsqueda simple en Google
    Given un navegador web en la página de Google
    When se introduce la palabra de búsqueda "pingüino"
    Then se muestra el resultado de "pingüino"
```