**TDD - Test Driven Design**

**Definiciones:**

* Práctica de Desarrollo dentro de la metodología XP, aunque se puede utilizar en cualquier otra metodología.
* Dos enfoques en los Tests (complementarios, no exclusivos):
  + Enfoque tradicional, basado en QA (Quality Assurance):
    - Los tests se utilizan principalmente para detectar errores (bugs) en el código y asegurarse de que el código cumple con lo especificado en los requisitos.
    - El Equipo de pruebas se encuentra totalmente separado del equipo de desarrollo. Los primeros se denominarán QA.
    - **Pasos que se realizan en un test tradicional:**
      * **Primero se especifica**: Se definen los requisitos y especificaciones del software a desarrollar antes de comenzar a codificar.
      * **Después se codifica**: Se escribe el código para cumplir con esas especificaciones.
      * **Por último, se prueba**: Una vez que el código está escrito, se realizan las pruebas para verificar que el código hace lo que se espera y no contiene errores.
  + Enfoque moderno, basado en TDD:
    - Los tests **ayudan a especificar** el desarrollo.
    - Los tests son ejemplos de funcionamiento que **indican cómo debe funcionar el código**. Obliga a reflexionar sobre la **funcionalidad del mismo** en lugar de sobre la implementación: ¿qué característica queremos añadir? ¿cómo podemos probarla? ¿está bien definida para poder automatizar sus pruebas?
    - Se diseñan los tests al mismo tiempo que la especificación (son parte de la especificación) y **después se codifica**. El resultado de esto es no solo el código sino también las pruebas (self testing code).
    - En los unit tests ni se hace énfasis en el aislamiento, la unidad a probar puede llamar a otras unidades que ya fueron probadas debido a que también fueron desarrolladas con TDD. Además representan un incremento de funcionalidad.
    - Al aplicar TDD se generar sensación de seguridad permitiendo refactorizar sin miedo.
    - "Sistema de doble chequeo" Tests antes y después de codificar.

**Tipos de tests:**

* **Tests unitarios (unit tests)**
  + Prueban la funcionalidad del código (JUnit, NUnit, PhpUnit, ...)
  + Partes del código que no interactúan con recursos externos
  + Se usan *stubs* y *mocks* para sustituir los recursos
  + Se ejecutan en el entorno del desarrollador
* **Tests de integración**
  + Funcionalidad del código que depende de recursos externos (bases de datos, servicios web, servicios de mail, etc.)
  + Se ejecutan en entornos preparados que requieren una configuración especial (en la que se inicializan recursos, ficheros de configuración, etc.)
* **Tests funcionales**
  + Tests simulando usuarios finales de tipo caja negra
* **Pruebas de stress**
  + Prueban el rendimiento del sistema

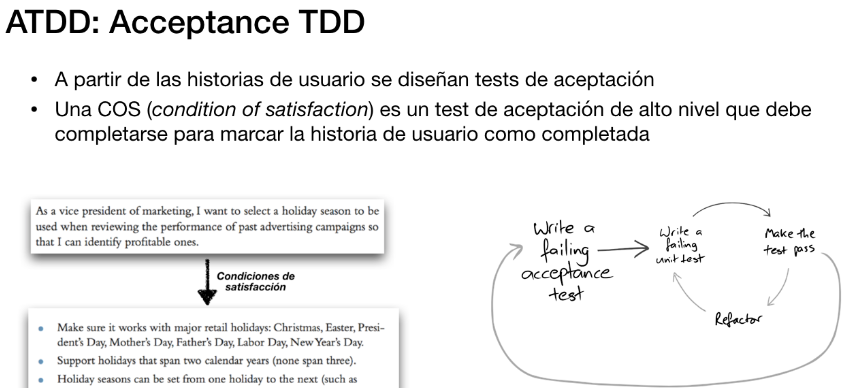
**Algunas características de los tests automatizados:**

* **Rápidos**
  + Sobre todos los tests unitarios.
  + Se van a ejecutar miles de tests, cientos de veces, la velocidad importa mucho.
* **Aislados**
  + Un test no debe depender de los otros.
  + Deben de poder ejecutarse en orden indistinto.
* **Atómicos**
  + Un test debe probar una única cosa.

**Pasos que se realizan en un test moderno, basado en TDD:**

1. Escribir un Test que Falla (Rojo):
   * Escribe un test que compruebe la nueva funcionalidad que deseas añadir al sistema.
   * Este test inicialmente fallará porque la funcionalidad aún no está implementada.
2. Escribir el Código para Pasar el Test (Verde):
   * Implementar el código justo y necesario para que el test pase.
   * Aunque el código pueda ser algo sucio o básico, su objetivo principal es hacer que el test pase.
3. Refactorizar el Código (Azul):
   * Mejora y limpia el código escrito en el paso anterior.
   * Asegúrate de que el código sea eficiente y fácil de mantener, sin que los tests dejen de pasar.
   * La refactorización puede no darse, si se da mejora su estructura, legibilidad y eficiencia

* Estas fases se repiten para cada pequeña funcionalidad que se desea añadir al sistema, siguiendo el ciclo TDD.
* Se pueden llevar a cabo pruebas adicionales para cubrir casos específicos o límites.
* Después de refactorizar se prueba para comprobar que no las mejoras no introdujeron errores.

****

**Refactorización:**  
Refactorizar significa mejorar el código existente sin cambiar su comportamiento observable.

**1. Más clara**

**Refactorizar para que el código sea más fácil de entender.**

* **Objetivo:** Hacer que el código sea legible y comprensible para otros desarrolladores (o para ti mismo en el futuro).
* **Métodos:** Usar nombres de variables y métodos descriptivos, simplificar estructuras complejas, eliminar código duplicado.

**2. Más eficiente**

**Refactorizar para mejorar el rendimiento del código.**

* **Objetivo:** Hacer que el código sea más rápido y utilice menos recursos.
* **Métodos:** Optimizar algoritmos, reducir complejidad, minimizar accesos a la base de datos, mejorar el uso de memoria.

**3. Mejor diseño para el cambio**

**Refactorizar para que el código sea fácil de modificar y extender.**

* **Objetivo:** Facilitar la implementación de nuevas funcionalidades y adaptarse a nuevos requerimientos.
* **Métodos:** Aplicar principios de diseño de software que favorezcan la flexibilidad y extensibilidad.

**4. Desacoplada**

**Refactorizar para reducir la dependencia entre componentes.**

* **Objetivo:** Hacer que los componentes del sistema sean independientes entre sí, de modo que los cambios en uno no afecten a otros.
* **Métodos:** Usar interfaces, inyección de dependencias, separar responsabilidades.

**5. Cohesiva**

**Refactorizar para que cada clase y método tenga una única responsabilidad clara.**

* **Objetivo:** Asegurarse de que cada componente del código tenga una única responsabilidad bien definida.
* **Métodos:** Dividir clases y métodos que hacen demasiado en partes más pequeñas y específicas.
  + *Lo ideal en un código es bajo acoplamiento y alta cohesión.*

**6. SOLID**

**Refactorizar para adherirse a los principios SOLID de diseño de software.**

* **Objetivo:** Mejorar la calidad del diseño del software utilizando los cinco principios SOLID:
  + **S** (Single Responsibility Principle): Una clase debe tener una sola responsabilidad.
  + **O** (Open/Closed Principle): Las clases deben estar abiertas a la extensión pero cerradas a la modificación.
  + **L** (Liskov Substitution Principle): Las clases derivadas deben ser sustituibles por sus clases base.
  + **I** (Interface Segregation Principle): Los clientes no deben estar obligados a depender de interfaces que no usan.
  + **D** (Dependency Inversion Principle): Depender de abstracciones y no de concreciones.