TEMA 5 - PRUEBAS, DISTRIBUCIÓN E IMPLANTACIÓN DE APLICACIONES

TEMA 5 - PRUEBAS, DISTRIBUCIÓN E IMPLANTACIÓN DE APLICACIONES

Índice de contenidos

Objetivos de aprendizaje

1. Conocer las metodologías usadas en el desarrollo de software.
2. Conocer los tipos de pruebas existentes.
3. Desarrollar código basándose en una metodología guiada por pruebas.

4. Integrar y desplegar cambios en las aplicaciones de forma organizada y segura.
5. Distribuir aplicaciones.

1. Metodologías en el desarrollo de software

Cuando se desarrolla software no basta con saber programar. También es importante **organizar el trabajo**, coordinar al equipo y decidir **cómo** se avanza desde una idea hasta una aplicación que funcione.

Para eso existen las **metodologías de desarrollo de software**: formas de trabajar que indican qué hacer en cada momento del proyecto.

Durante muchos años se usaron metodologías muy rígidas. Con el tiempo se comprobó que estas formas de trabajo provocaban problemas habituales:

- Las aplicaciones tardaban mucho en llegar al usuario.
- Los cambios eran difíciles de introducir.
- El mantenimiento resultaba costoso.
- La calidad final no siempre era la esperada.

A partir de estos problemas, alrededor del año 2000 comenzaron a surgir las **metodologías ágiles**, que proponen una forma de trabajo más flexible y cercana a la realidad actual del desarrollo de aplicaciones.

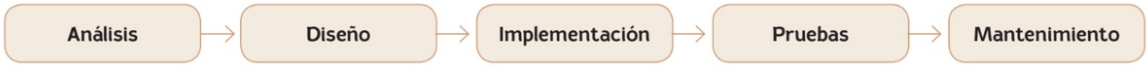
1.1. Metodologías tradicionales

Las metodologías tradicionales son las que se han utilizado durante más tiempo. Su objetivo principal es que el proyecto sea **previsible** y esté totalmente planificado desde el principio.

Se basan en procesos **secuenciales**, donde cada fase debe terminarse antes de pasar a la siguiente. Esto hace que las estimaciones iniciales tengan mucha importancia y que los cambios sean complicados de asumir.

Desarrollo en cascada

La metodología tradicional más conocida es el **desarrollo en cascada**, que se divide normalmente en cinco fases bien diferenciadas:



Vamos a ver qué se hace en cada una de ellas:

1. **Análisis de requisitos**

   En esta fase se define qué debe hacer la aplicación.

   Los requisitos tienen que estar muy claros desde el inicio, ya que no se espera que cambien durante el desarrollo.

   Por este motivo, suele encargarse el personal con más experiencia.

2. **Diseño de la solución**

   Se decide cómo va a ser la aplicación para cumplir los requisitos: estructura, componentes, pantallas, lógica general, etc.

3. **Implementación**

   Se programa la aplicación siguiendo el diseño anterior, utilizando un lenguaje y unas tecnologías concretas.

4. **Pruebas**

   Se comprueba que la aplicación funciona y que cumple con los requisitos definidos al inicio.

5. **Mantenimiento**

   Una vez la aplicación está en uso, se corrigen errores y se realizan pequeñas modificaciones si aparecen problemas.

📌 Idea clave

Este tipo de metodología funciona bien cuando los requisitos están muy claros desde el principio y apenas cambian, algo que hoy en día no siempre ocurre.

1.2. Metodologías ágiles

Las metodologías ágiles surgen como respuesta a las limitaciones de las metodologías tradicionales.

Su característica principal es que **se adaptan a los cambios** que pueden aparecer durante el desarrollo de una aplicación. En lugar de seguir un plan rígido, se avanza poco a poco, revisando el trabajo de forma continua.

Las metodologías ágiles son más flexibles y se apoyan en los **12 principios del Agile Manifiesto**, un documento creado por profesionales del mundo del software para proponer mejores formas de desarrollar aplicaciones.

Estos principios se resumen en **cuatro valores fundamentales**.

Los 4 valores de las metodologías ágiles

1. **Individuos e interacciones sobre procesos y herramientas**

   Se da más importancia a las personas del equipo y a su capacidad para tomar decisiones que a seguir procesos rígidos o depender de herramientas.

   Un buen equipo organizado suele dar mejores resultados que un proceso perfecto.

2. **Software funcionando sobre documentación extensiva**

   Se prioriza tener una aplicación que funcione, esté desplegada y pueda usarse, antes que generar mucha documentación.

   La documentación es útil si aporta valor, pero no debe frenar el avance del proyecto.

3. **Colaboración con el cliente sobre negociación contractual**

   El contacto con el cliente es continuo.

   El cliente prueba el software, da su opinión y ese feedback puede provocar cambios durante el desarrollo, en lugar de trabajar con requisitos cerrados desde el inicio.

4. **Respuesta ante el cambio sobre seguir un plan**

   Adaptarse a un cambio inesperado suele mejorar más el software que seguir un plan que ya no encaja con las nuevas necesidades.

💡 ¿Sabías que...?

En muchos proyectos reales, los requisitos cambian varias veces antes de terminar la aplicación. Las metodologías ágiles están pensadas precisamente para trabajar en ese contexto.

A. Los 12 principios de las metodologías ágiles

A partir de los valores anteriores se definen los siguientes principios:

1. El cliente es el centro.
2. Cambiar requisitos buscando la ventaja competitiva.
3. Acortar los tiempos de entrega de los servicios.
4. Involucrar a todas las personas.
5. Construir proyectos con personas motivadas.
6. La información entre personas es la más eficiente y efectiva.
7. Un equipo motivado es indicador de éxito del proyecto.
8. La agilidad promueve entornos de trabajo sostenibles.
9. Agilidad no es sinónimo de rapidez, sino de excelencia.
10. Lo breve y bueno, dos veces bueno.
11. Equipos autónomos.
12. Mejora basada en el feedback continuado.

B. Metodologías ágiles más utilizadas

Dentro de las metodologías ágiles existen distintas formas de organizar el trabajo. Aunque todas comparten los valores del Manifiesto Ágil, cada una pone el foco en aspectos diferentes.

En el entorno profesional, algunas metodologías destacan por su sencillez y facilidad de adaptación a distintos proyectos.

Kanban

Kanban se centra en **visualizar el trabajo** y controlar cuántas tareas se realizan al mismo tiempo.

El trabajo se muestra en un **tablero** con columnas como:

- Pendiente
- En proceso
- Terminado

Cada tarea se representa con una tarjeta que va cambiando de columna según su estado.

Una idea clave es **limitar las tareas en curso**, para evitar empezar muchas cosas a la vez.

📌 Cuándo usar Kanban

- Trabajo continuo.
- Equipos pequeños.
- Tareas de mantenimiento.

Scrum

Scrum organiza el trabajo en ciclos cortos llamados **sprints**, de una a cuatro semanas.

En cada sprint se desarrolla una parte del proyecto que debe quedar funcional.

Se apoya en:

- Roles definidos.
- Reuniones periódicas.
- Una lista de tareas priorizadas.

Al final de cada sprint, el cliente puede ver **una parte del software funcionando**.

📌 Cuándo usar Scrum

- Proyectos divididos en partes.
- Necesidad de feedback frecuente.
- Trabajo en equipo coordinado.

Extreme Programming (XP)

Extreme Programming (XP) pone el foco en la **calidad del código** y las **buenas prácticas de programación**.

Promueve:

- Código sencillo.
- Pruebas frecuentes.
- Comunicación constante en el equipo.

Su objetivo es detectar errores pronto y facilitar los cambios cuando evolucionan los requisitos.

📌 Cuándo usar XP

- Proyectos con muchos cambios.
- Equipos que programan de forma colaborativa.
- Prioridad en la calidad del código.

Resumen

- **Kanban**: visualiza el trabajo y limita tareas en curso.
- **Scrum**: trabaja por sprints con entregas frecuentes.
- **XP**: cuida la calidad del código y las prácticas de desarrollo.

Recurso:

Kanban, Scrum, XP y otras metodologías ágiles

2. Tipos de pruebas de aplicaciones

Cuando desarrollamos una aplicación, programar no es el último paso. Antes de darla por terminada, es imprescindible **comprobar que funciona correctamente**, que cumple lo que se espera de ella y que no contiene errores importantes.

Para eso se realizan las **pruebas de software**.

2.1. ¿Qué son las pruebas de software?

Las pruebas de software son una **fase fundamental del desarrollo de una aplicación**. Su objetivo es asegurar unos mínimos de calidad antes de que el software llegue al usuario final.

De hecho, en muchos proyectos reales, la fase de pruebas puede suponer entre el **15% y el 25% del esfuerzo total** del desarrollo. Esto demuestra la importancia que tienen.

Probar una aplicación significa:

- Verificar que funciona correctamente.
- Comprobar que cumple los requisitos definidos al inicio del proyecto.
- Detectar errores antes de que el software se ponga en uso.

Las pruebas pueden realizarse:

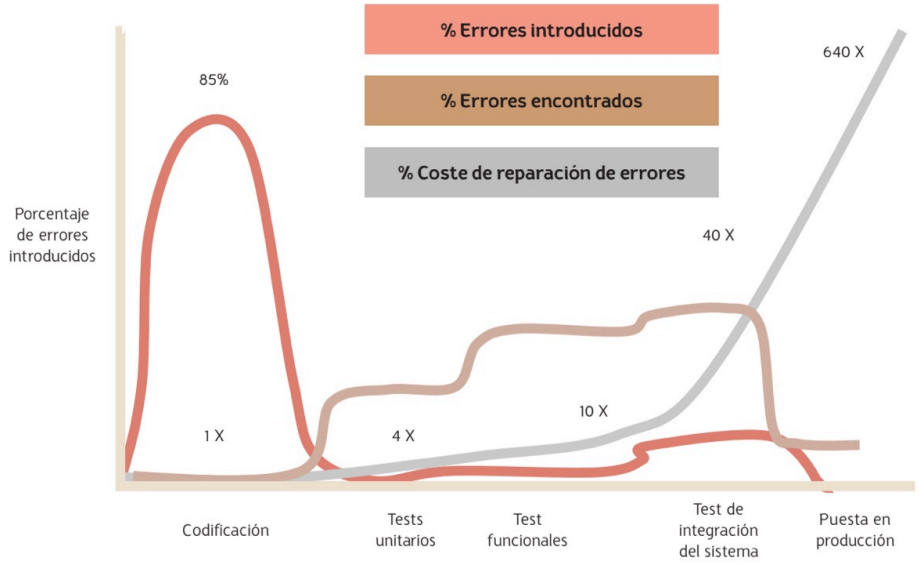
- **De forma manual**, ejecutando la aplicación y comprobando su comportamiento.
- **De forma automatizada**, mediante herramientas que ejecutan pruebas de manera automática.

Un aspecto muy importante es que las pruebas **no deben dejarse para el final**. Lo ideal es realizarlas **en paralelo a la codificación**, ya que la mayoría de errores se introducen cuando se empieza a programar.

📌 **Idea clave**

Cuanto más tarde se detecta un error, **más caro y costoso** resulta arreglarlo. Un fallo detectado en producción puede multiplicar por mucho el esfuerzo necesario para corregirlo.

Coste de los errores según el momento en que se detectan



Basado en Jones, Capers. Applied Software Measurement Global Analysis of Productivity and Quality

- Durante la **codificación**: coste bajo (1×).
- En **tests unitarios**: coste mayor (4×).
- En **tests funcionales**: el coste sigue aumentando (10×).
- En **tests de integración del sistema**: el esfuerzo se dispara (40×).
- En **producción**: el coste puede ser enorme (640×).

La mayoría de errores se introducen al principio, pero si no se detectan a tiempo, el impacto es mucho mayor.

2.2. Objetivos de las pruebas de software

La fase de testing persigue varios objetivos importantes:

- Detectar y corregir errores en etapas tempranas del desarrollo.
- Asegurar que la aplicación funciona correctamente.
- Aportar calidad al producto final.
- Garantizar la confiabilidad del software.
- Evitar que los cambios introduzcan nuevos errores.
- Cumplir los requisitos del negocio.
- Lograr la satisfacción del usuario final.

📌 **En resumen**

Las pruebas ayudan a que la aplicación sea usable, fiable y acorde a lo que necesita el usuario.

2.3. Tipos de pruebas

Las pruebas de software se suelen clasificar en **dos grandes grupos**:

Pruebas funcionales

Las **pruebas funcionales** comprueban que la aplicación hace lo que debe hacer.

Se centran en:

- La funcionalidad.
- La usabilidad.
- El cumplimiento de los requisitos definidos.

Estos requisitos suelen recogerse en un documento llamado **Software Requirement Specification (SRS)**.

Existen muchos tipos de pruebas funcionales, pero las más habituales y útiles son:

- Pruebas de humo.
- Pruebas unitarias.
- Pruebas de integración.
- Pruebas de regresión.
- Pruebas de aceptación.

Pruebas no funcionales o de rendimiento

Las **pruebas no funcionales** no se centran en qué hace la aplicación, sino en **cómo lo hace**.

Evalúan aspectos como:

- Rendimiento.
- Fiabilidad.
- Usabilidad.
- Escalabilidad.

Estas pruebas suelen realizarse con **herramientas de automatización**, especialmente en aplicaciones grandes o con muchos usuarios.

2.4. Pruebas funcionales

Dentro de las pruebas funcionales, existen varios tipos. A continuación veremos las que se consideran **más importantes en un proyecto de software**.

A. Pruebas unitarias

Las **pruebas unitarias** son pruebas realizadas por los propios programadores sobre las **partes más pequeñas del sistema**, normalmente funciones o clases.

Su objetivo es comprobar que cada parte del código funciona de forma correcta por separado.

Gracias a las pruebas unitarias:

- Se detectan errores muy pronto.
- Se facilita la **refactorización del código**, es decir, reescribirlo sin cambiar su funcionalidad.
- Se reducen problemas en fases posteriores de integración.
- Se puede probar el código sin necesidad de tener la aplicación completa.

📌 **Ejemplo**

Antes de probar toda la aplicación, se comprueba que una función de validación de usuarios funciona correctamente por sí sola.

Para reflexionar 🤔

- ¿Qué crees que pasa si no se hacen pruebas hasta el final del proyecto?
- ¿Qué ventajas tiene detectar errores cuando aún se está programando?

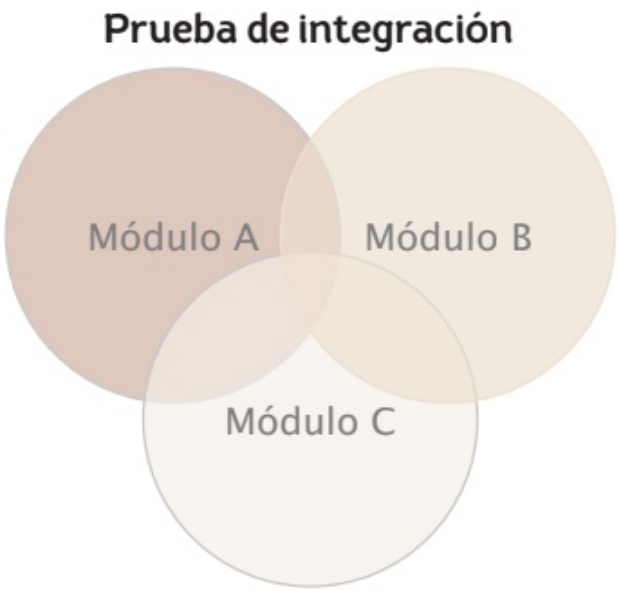
B. Pruebas de integración de componentes

Después de comprobar que cada parte del programa funciona por separado mediante las **pruebas unitarias**, es necesario comprobar que **todas esas partes funcionan correctamente cuando se unen**. Para eso se realizan las **pruebas de integración de componentes**.

Que un módulo funcione bien por sí solo **no garantiza** que lo haga cuando se comunica con otros módulos. Estas pruebas sirven para detectar problemas en:

- La comunicación entre componentes.
- El intercambio de datos.
- El comportamiento del sistema cuando varias partes trabajan juntas.

Las pruebas de integración se realizan **después de las pruebas unitarias** y permiten validar que el sistema, como conjunto, se comporta de la forma esperada.



📌 Idea clave

Primero se prueba cada pieza por separado y después se prueba cómo encajan entre ellas.

Ejemplo

Supongamos una aplicación con:

- Un módulo de **login**.
- Un módulo de **pantalla de inicio**.

En las pruebas unitarias se comprueba que:

- El login valida correctamente al usuario.
- La pantalla de inicio se muestra sin errores.

En las **pruebas de integración** se comprueba que:

- Al hacer login, el usuario es redirigido a su pantalla de inicio.
- La pantalla muestra los datos del usuario correctamente.

Aquí se está comprobando el funcionamiento conjunto de varios módulos.

Prueba unitaria vs prueba de integración

- **Prueba unitaria**:

  Se prueban los módulos de forma independiente (Módulo A, Módulo B, Módulo C).

- **Prueba de integración**:

  Se prueban los módulos trabajando juntos (A + B + C).

¿Quién realiza estas pruebas?

Las buenas prácticas de calidad de software indican que las pruebas de integración **no deberían realizarlas las mismas personas que han programado el código**, para evitar posibles sesgos.

Por este motivo, suelen ejecutarlas **equipos externos al desarrollo** o perfiles especializados en testing.

Pruebas de humo

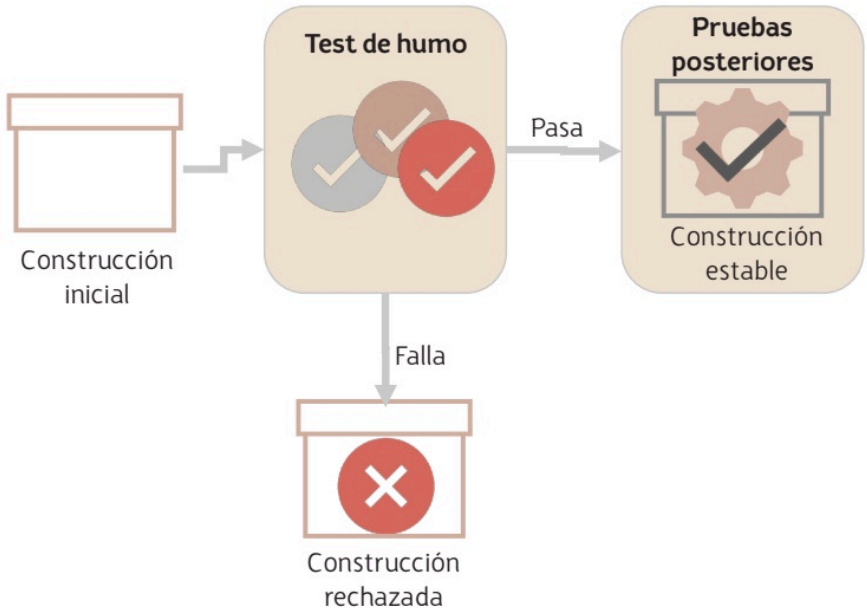
Las **pruebas de humo** se realizan en las **primeras fases del testing** y sirven para comprobar si la aplicación está en condiciones de seguir siendo probada.

No buscan detectar todos los errores, sino comprobar que las **funcionalidades básicas y críticas** funcionan.

📌 **Ejemplos de funcionalidades críticas**

- Inicio de sesión.
- Registro de usuarios.
- Acceso a las pantallas principales.

¿Qué ocurre tras una prueba de humo?



- Si la prueba **pasa**, la aplicación continúa con pruebas más completas.
- Si la prueba **falla**, la aplicación se considera inestable y vuelve al equipo de desarrollo para corregir errores.

Normalmente, las pruebas de humo representan **un pequeño porcentaje del total de pruebas** (alrededor del 5%), pero son muy importantes para evitar perder tiempo probando una aplicación que no está lista.

Pruebas de regresión

Las **pruebas de regresión** se utilizan durante el desarrollo para comprobar que los **cambios realizados no rompen funcionalidades que antes funcionaban**.

Cada vez que se añade una nueva funcionalidad o se corrige un error, existe el riesgo de que otra parte del sistema deje de funcionar.

📌 **Idea clave**

Cambiar una parte del código puede afectar a otras partes sin darnos cuenta.

Test de regresión

Las pruebas de regresión pueden realizarse de distintas formas:

- Volver a probar todo el sistema.
- Probar solo algunos casos concretos.
- Priorizar los casos más importantes.

Normalmente se utilizan **pruebas automatizadas**, aunque en versiones inestables o con grandes cambios se combinan con pruebas manuales.

Pruebas de aceptación

Las **pruebas de aceptación** pertenecen a la **fase final del testing**.

En ellas, el **usuario final** comprueba que la aplicación cumple con lo que espera y con los requisitos definidos, utilizándola en un entorno lo más real posible.

Estas pruebas suelen ser:

- Manuales.

- Exploratorias.
- Realizadas paso a paso por el usuario, acompañado de un analista de pruebas.

📌 **Objetivo**

Confirmar que la aplicación es válida para su uso real.

Pruebas no funcionales

Las **pruebas no funcionales** no evalúan qué hace la aplicación, sino **cómo lo hace**.

Normalmente las realizan equipos especializados en implantación y no el equipo de desarrollo. Evalúan aspectos importantes como:

- **Pruebas de rendimiento**: velocidad de respuesta con una carga normal.
- **Pruebas de carga**: comportamiento con muchos usuarios o peticiones.
- **Pruebas de estrés**: funcionamiento bajo condiciones extremas.
- **Pruebas de volumen**: rendimiento con grandes cantidades de datos.
- **Pruebas de seguridad**: protección frente a ataques internos y externos.
- **Pruebas de compatibilidad**: funcionamiento en distintos sistemas y entornos.
- **Pruebas de instalación**: correcto funcionamiento tras instalar la aplicación.
- **Pruebas de recuperación**: capacidad de recuperarse tras un fallo.
- **Pruebas de confiabilidad**: funcionamiento estable durante un periodo de tiempo.
- **Pruebas de usabilidad**: facilidad de uso de la aplicación.
- **Pruebas de conformidad**: cumplimiento de normas y estándares.

Para pensar 🤔

- ¿Qué prueba te parece más importante en una aplicación real?
- ¿Qué pasaría si no se hicieran pruebas de regresión?

3. Desarrollo guiado por pruebas (TDD)

Hasta ahora hemos visto que probar una aplicación no es "un extra": es una forma de evitar sorpresas cuando el software crece o cuando alguien toca el código semanas después.

En esta unidad veremos una forma de programar donde las **pruebas no van al final**, sino **al principio**. Se llama:

TDD (Test Driven Development) o **desarrollo guiado por pruebas**.

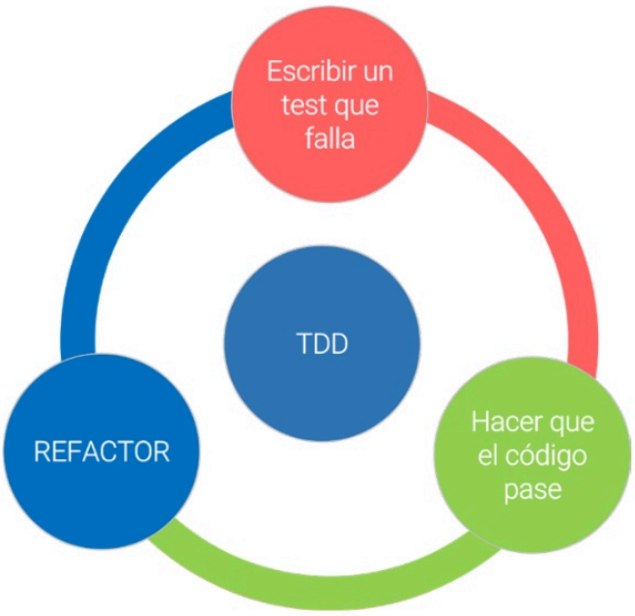
📌 **Idea clave**

En TDD primero decides cómo debe funcionar una parte del programa… y lo expresas como una prueba. Luego escribes el código para que esa prueba pase.

3.1. ¿Qué es TDD?

TDD es una manera de desarrollar software siguiendo un ciclo muy corto y repetitivo. La idea es avanzar en pequeños pasos, comprobando continuamente que todo funciona.

El ciclo típico se resume en **Red – Green – Refactor**:



1) **Red (Rojo): escribir una prueba que falla**

- Primero se escribe un test para una funcionalidad nueva.
- Ese test debe fallar, porque todavía no existe el código.

Ejemplo:

"Si sumo 2 y 3, el resultado debe ser 5".

Haces el test… y falla porque la función no está hecha.

2) **Green (Verde): escribir lo mínimo para que pase**

- Se programa el **mínimo código** necesario para que el test pase.
- Aquí el objetivo es que funcione, aunque sea de forma simple.

Ejemplo:

Creas la función `sumar(a, b)` y consigues que el test pase.

3) **Refactor: mejorar el código sin romperlo**

- Una vez que todo pasa, se puede **mejorar el código**:
  - hacerlo más claro,
  - evitar repetición,
  - ordenarlo mejor,
  - dividir en funciones si hace falta…

- Y después se ejecutan los tests otra vez para comprobar que todo sigue bien.

📌 **Idea clave**

Refactorizar es "limpiar y mejorar" sin cambiar lo que hace el programa.

¿Por qué este método suele dar buen resultado?

Porque:

- obliga a escribir código sencillo,
- reduce fallos,
- evita duplicaciones,
- facilita el mantenimiento,
- y te permite hacer cambios con más seguridad.

3.2. ¿Por qué usar TDD?

La crítica típica es:

> "TDD lleva más tiempo porque hay que escribir pruebas".

Y es verdad que al principio tardas más… pero compensa por varias razones:

1) Te obliga a pensar antes de programar

Para escribir un test tienes que tener claro:

- qué entra (entrada),
- qué sale (salida),
- qué pasa en casos raros o incorrectos.

Eso ayuda a diseñar mejor la función antes de empezar.

2) Da confianza cuando el proyecto crece

En proyectos reales, el código cambia mucho.

Con tests:

- puedes modificar algo,
- ejecutar tests,
- y saber si has roto algo sin darte cuenta.

📌 En la práctica: los tests son como un "semáforo".

- Verde: todo ok.
- Rojo: has roto algo.

3) Reduce la probabilidad de errores

TDD no elimina todos los errores, pero ayuda a que aparezcan antes.

Y un error detectado pronto cuesta menos corregirlo que uno detectado al final.

4) Los tests sirven como "mini-documentación"

Un test muestra cómo se espera usar una función.

Si alguien nuevo entra al proyecto, puede mirar los tests y entender:

- cómo se llama la función,
- qué valores espera,
- qué devuelve.

3.3. Cobertura de código

La **cobertura de código** es una medida que nos dice cuánto código está siendo ejecutado por los tests.

- **Cobertura del 100%** significa que todo el código se ha ejecutado al menos una vez durante las pruebas.

Se puede medir de varias formas, por ejemplo:

- **Número de líneas probadas**
- **Número de funciones probadas**
- **Número de ramas o caminos probados**

- Una "rama" aparece cuando hay decisiones en el código, como un `if`.
- Si hay un `if`, hay al menos dos caminos: el "sí" y el "no".

📌 **Ojo con esto**

Tener mucha cobertura no significa que no haya errores.

Puedes ejecutar una línea, pero no haber probado el caso "difícil" o el caso raro.

Preguntas para pensar

1. ¿Qué ventaja tiene escribir el test antes que el código?
2. ¿Qué crees que es más peligroso: un proyecto sin tests o con pocos tests?
3. ¿Por qué un 100% de cobertura no garantiza que todo esté perfecto?

3.4. Pruebas unitarias y pruebas de integración

Cuando los proyectos son pequeños, el número de funciones y componentes suele ser reducido. Sin embargo, en proyectos grandes esto cambia: aparecen **muchas funciones**, **muchas clases** y **muchos módulos** que deben trabajar juntos.

Para comprobar que todo funciona correctamente se utilizan distintos tipos de pruebas:

Pruebas unitarias

- Comprueban que **cada componente funciona bien por separado**.
- Se centran en funciones o clases individuales.
- Ayudan a detectar errores muy pronto.

📌 Ejemplo

Comprobar que una función que suma dos números devuelve el resultado correcto.

Pruebas de integración

- Comprueban que **varios componentes funcionan bien juntos**.
- Se centran en la comunicación entre partes del sistema.

📌 Ejemplo

Comprobar que, después de iniciar sesión, el usuario accede correctamente a su pantalla principal.

¿Qué papel juega TDD aquí?

En TDD siempre se empieza escribiendo una **prueba que falla**, pero **no es obligatorio** que esa prueba sea unitaria.

- La primera prueba puede ser **unitaria**, si queremos ir paso a paso.
- O puede ser **de integración**, si queremos validar directamente una funcionalidad completa del sistema.

Si comenzamos por una prueba de integración:

- Esa prueba fallará hasta que todos los componentes necesarios estén desarrollados.
- Esto obliga al desarrollador a crear las pruebas unitarias de cada parte.
- Cuando finalmente la prueba de integración pasa, significa que se ha cumplido un **requisito real del usuario**.

📌 **Idea clave**

Las pruebas unitarias ayudan a construir las piezas.

Las pruebas de integración comprueban que las piezas encajan bien.

3.5. TDD y metodologías ágiles

El desarrollo guiado por pruebas encaja muy bien con las **metodologías ágiles**.

En el desarrollo ágil:

- El software se construye poco a poco.
- Se hacen entregas frecuentes.

- La calidad debe mantenerse alta desde el principio.

TDD ayuda a conseguir esto porque:

- Permite avanzar en pequeños pasos.
- Garantiza que cada cambio está probado.
- Reduce los problemas cuando la aplicación llega a producción.

📌 **Resultado práctico**

Menos errores graves al final del proyecto y más confianza en el código.

TDD y trabajo en equipo

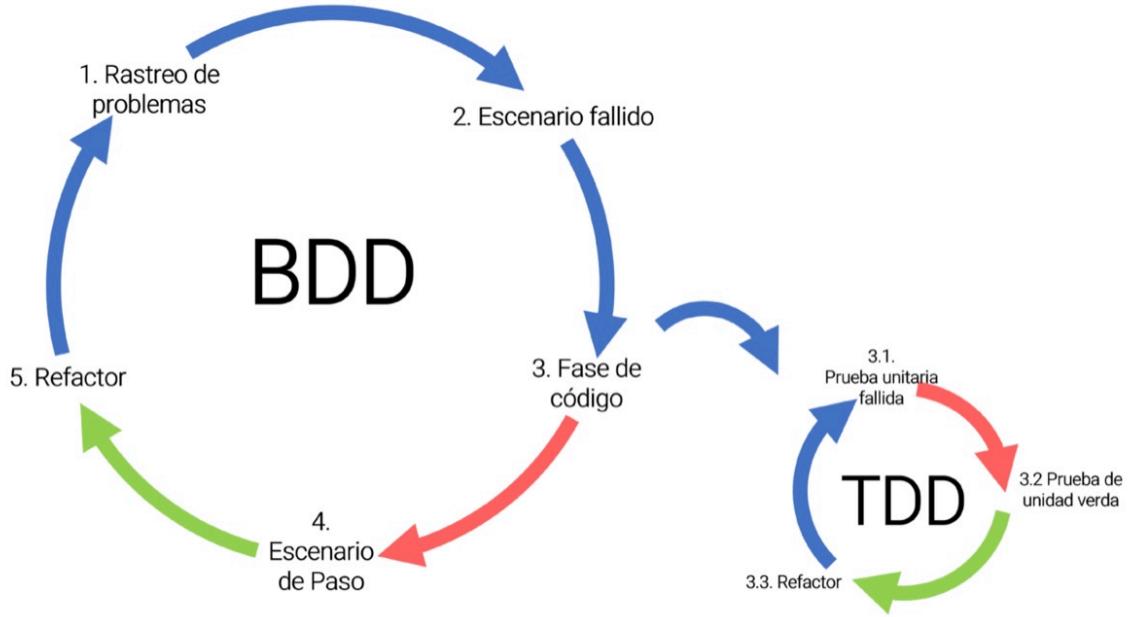
TDD también se adapta muy bien a técnicas ágiles como el **pair programming** (programación por parejas):

- Una persona puede escribir las pruebas.
- La otra puede escribir o refactorizar el código para superarlas.
- Luego se intercambian los roles.

Esto permite:

- Revisar continuamente el trabajo.
- Detectar errores antes.
- Mejorar la calidad del código y el aprendizaje del equipo.

BDD: una evolución de TDD



El **BDD (Behavior Driven Development)** o desarrollo guiado por el comportamiento es una evolución de TDD.

Su objetivo principal es:

- Mejorar la comunicación entre el equipo técnico y el cliente.
- Describir el comportamiento del sistema desde el punto de vista del usuario.

BDD combina:

- Requisitos del negocio.
- Pruebas.
- Código.

De esta forma, el sistema se entiende no solo desde el punto de vista técnico, sino también desde el **punto de vista del usuario final**.

Comparación TDD vs BDD

TDD

- Se centra en el código.
- Usa pruebas unitarias.
- Ciclo: prueba fallida → código → refactor.

BDD

- Se centra en el comportamiento del sistema.
- Usa escenarios cercanos al lenguaje del usuario.
- Facilita la comunicación con el cliente.