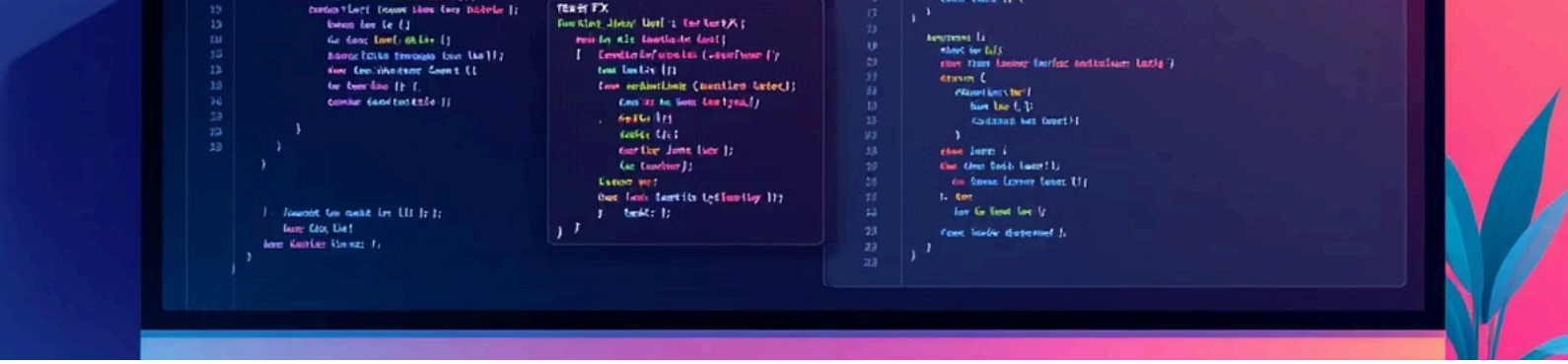
PROMETEO

Unidad 10:

Pruebas, Depuración y Mantenimiento

Programación

Técnico Superior de DAM / DAW



Sesión 24: JUnit y TestFX: Pruebas automatizadas en GUI y lógica

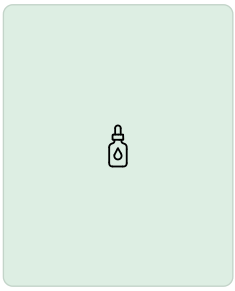
Probar una aplicación no es "abrirla y ver si funciona". En un entorno profesional, necesitas garantías sólidas de que cada pieza del software cumple exactamente lo esperado, incluso después de refactorizar, añadir nuevas características o integrar componentes externos. Las pruebas automatizadas son el mecanismo que te permite validar el comportamiento del sistema de manera repetible, rápida y sin depender de verificaciones manuales.

> ⓘ En Java, el estándar de facto para pruebas unitarias es **JUnit**, un framework que te permite crear métodos de prueba estructurados, ejecutar baterías completas de tests en segundos y detectar errores de manera temprana. Un test unitario comprueba una pieza pequeña y aislada del código —generalmente un método o función—, asegurando que ante una entrada concreta siempre produce la salida correcta. JUnit ofrece anotaciones como **@Test**, mecanismos de ciclo de vida como **@BeforeEach** y un conjunto de validaciones llamado **asserts**, que comparan el resultado esperado con el obtenido. Cuando un test falla, el error aparece inmediatamente, permitiendo corregirlo antes de que llegue a producción.

Sin embargo, cuando una aplicación incorpora una **interfaz gráfica** (GUI), la lógica no es lo único que debe funcionar: la interacción visual también debe responder correctamente. Si un botón deja de emitir un evento, si el foco no cambia de campo, o si una escena no se actualiza a tiempo, la experiencia del usuario se rompe. Para validar estas interacciones existe **TestFX**, una librería diseñada para automatizar acciones sobre aplicaciones **JavaFX**, simulando exactamente lo que haría una persona: hacer clic, escribir texto, navegar entre escenas o comprobar que un mensaje aparece en pantalla.

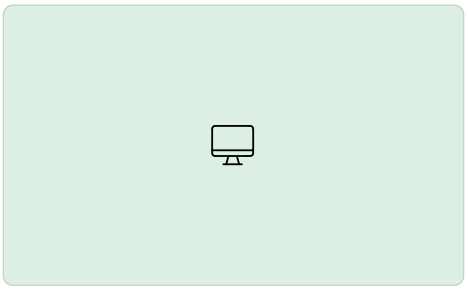
El verdadero valor surge al combinar JUnit y TestFX: la lógica interna se garantiza mediante tests unitarios, mientras que la interfaz y su comportamiento completo se verifican mediante interacciones automatizadas. Esta combinación crea un sistema de pruebas integral que reduce errores, evita regresiones, mejora la calidad del software y aporta seguridad al refactorizar. En entornos profesionales donde múltiples desarrolladores colaboran, los tests automatizados se convierten en una red de seguridad que mantiene estable cada entrega y evita que los cambios de un módulo rompan otro. Por eso, JUnit y TestFX forman parte de la caja de herramientas esencial de cualquier desarrollador Java profesional.

Esquema Visual



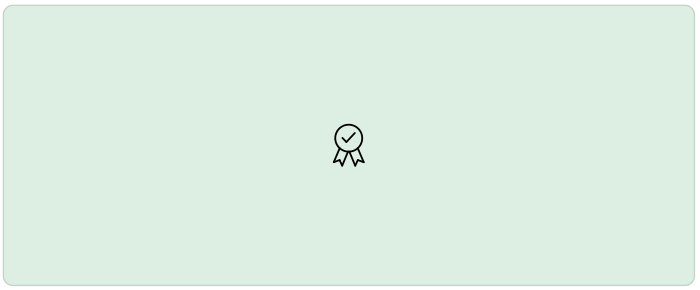
JUnit

- @Test
- Asserts
- @BeforeEach / @AfterEach



TestFX

- Interacciones GUI
- Clicks y escritura
- Pruebas end-to-end



Beneficios

- Calidad del software
- Evita regresiones
- Facilita refactorización

Explicación del esquema

- **Pruebas en Java** es el concepto general que engloba tanto pruebas unitarias como pruebas funcionales.
- **JUnit** representa la parte lógica: estructura de test, ciclo de vida y validación mediante asserts.
- **TestFX** cubre la parte visual: simula acciones del usuario y valida que la interfaz responde adecuadamente.
- **Beneficios** resume el impacto directo: mayor calidad, reducción de errores y la posibilidad de evolucionar el software con seguridad.

Caso de Estudio

Un sistema de gestión interno basado en JavaFX

Una empresa tecnológica desarrollaba una herramienta interna utilizada por distintos departamentos para registrar pedidos, actualizar inventarios y generar reportes. La aplicación estaba construida con JavaFX, y aunque la lógica de negocio funcionaba correctamente, la interfaz gráfica sufría errores frecuentes: botones que no activaban los controladores, formularios que no validaban correctamente, escenas que no se refrescaban o tablas que no recogían los cambios tras una actualización.

Cada nueva versión introducía más problemas, y el equipo pasaba horas revisando la GUI manualmente. Esto ralentizaba la entrega de funcionalidades y generaba frustración en los usuarios internos.

Estrategia: Integración de JUnit + TestFX

01

Pruebas unitarias con JUnit para la lógica interna

Crear tests para validar:

- Cálculos en controladores.
- Reglas de negocio.
- Gestión de excepciones usando assertThrows.
- Procesos de validación de formularios.

02

Pruebas de interfaz con TestFX

Se diseñaron tests automatizados que simulaban:

- Un clic en el botón "Guardar".
- La escritura de datos en un formulario.
- La detección de errores al enviar formularios vacíos.
- El cambio correcto entre escenas sin lanzar excepciones.
- La actualización visual de tablas tras modificar datos.

03

Integración en CI/CD

Cada commit en Git ejecutaba automáticamente todas las pruebas. Si un test fallaba, la integración se bloqueaba hasta corregir el problema.

Resultado

En apenas dos semanas:

- Las incidencias de interfaz se redujeron drásticamente.
- El tiempo de validación manual disminuyó más del 70%.
- Los desarrolladores ganaron confianza al refactorizar.
- Los usuarios internos comenzaron a reportar menos errores.

El sistema de pruebas automatizadas se convirtió en un pilar del proceso de desarrollo y permitió que nuevas funcionalidades llegaran de forma estable y predecible.

Herramientas y Consejos

1

Mantén los tests claros, cortos y con un único propósito

Un test debe verificar un comportamiento concreto. Si hace demasiado, será difícil de entender y mantener.

2

Usa assertThrows para validar situaciones de error

Ejemplo típico:

```
assertThrows(IllegalArgumentException.class, () -> servicio.calcular(-1));
```

Esto demuestra que tu lógica maneja correctamente los casos límite.

3

En TestFX, evita depender de tiempos artificiales

Dormir el hilo con Thread.sleep genera tests inestables. Utiliza:

```
FxToolkit.setupStage(...)
await().until(...)
```

para sincronizar acciones con la GUI.

4

Integra las pruebas en tu CI

Configura tu IDE o plataforma de CI (GitHub Actions, GitLab CI, Jenkins) para ejecutar todos los tests automáticamente al hacer commit o al crear un pull request. Esto evita que código defectuoso llegue a la rama principal.

5

Refactoriza con seguridad

Si un conjunto de tests está bien diseñado, cualquier cambio en la lógica romperá un test antes de llegar al usuario final.

Mitos y Realidades

❌ **Mito:** "Las pruebas ralentizan el desarrollo."

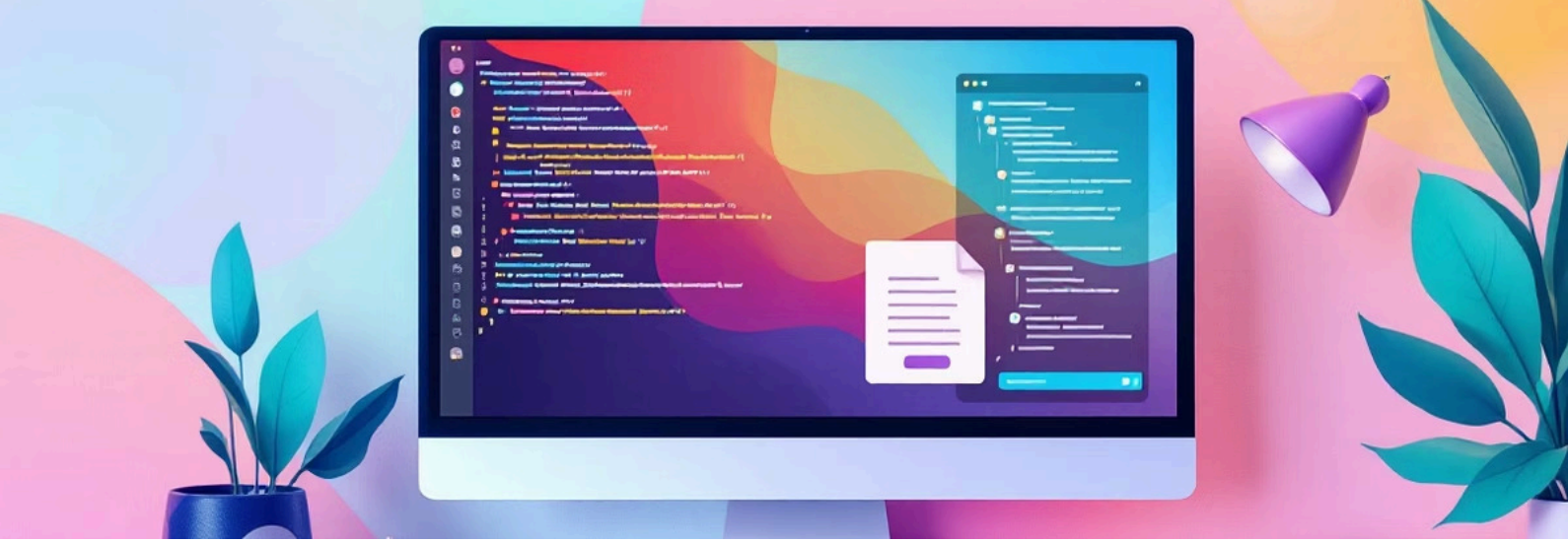
→ **FALSO.** Aunque al principio requieren tiempo, reducen errores, evitan regresiones y aceleran el desarrollo a medio plazo. El tiempo invertido en testear se recupera varias veces en mantenimiento.

❌ **Mito:** "TestFX es solo para probar botones."

→ **FALSO.** TestFX permite verificar navegación, validaciones visuales, actualización de escenas, efectos de animación y estados de la UI. Es una herramienta completa para pruebas funcionales end-to-end en JavaFX.

Resumen Final para Examen

- **JUnit** permite crear pruebas unitarias estructuradas con asserts y ciclo de vida controlado.
- **TestFX** automatiza interacciones reales con interfaces JavaFX (clics, escritura, navegación).
- Las pruebas combinadas reducen errores y evitan regresiones.
- Los asserts son el núcleo de la verificación en JUnit.
- Un buen conjunto de tests permite refactorizar con seguridad.



Sesión 25: Mantenimiento, control de versiones y documentación

Cuando un proyecto llega a producción, no termina su ciclo de vida: simplemente cambia de fase. A partir de ese momento, el software debe mantenerse vivo, estable y adaptable a nuevas necesidades. Esta etapa —el mantenimiento— es tan importante como la propia implementación inicial. Si no se gestiona correctamente, incluso la mejor aplicación puede volverse frágil, difícil de escalar y costosa de evolucionar.

Corrección de errores

El mantenimiento abarca varias actividades esenciales. La primera es la **corrección de errores**, que consiste en identificar fallos detectados por usuarios o por el equipo de QA y solucionarlos sin crear nuevos problemas. Esta tarea exige comprender el código existente, rastrear la causa original y validar que el comportamiento final es el esperado. A veces, un cambio aparentemente simple desencadena efectos secundarios; por eso, la comprensión profunda del sistema y la existencia de pruebas automatizadas son tan importantes.

Refactorización

Otra actividad clave es la **refactorización**, que no busca cambiar la funcionalidad, sino mejorar la estructura interna del código: eliminar duplicidades, renombrar métodos confusos, simplificar clases, dividir responsabilidades y mejorar el rendimiento. Refactorizar no es un capricho estético; es una estrategia imprescindible para evitar que el proyecto se degrade con el tiempo. Cuanto más claro y ordenado es el código, más fácil es mantenerlo, ampliarlo y reducir el riesgo de errores.

Documentación

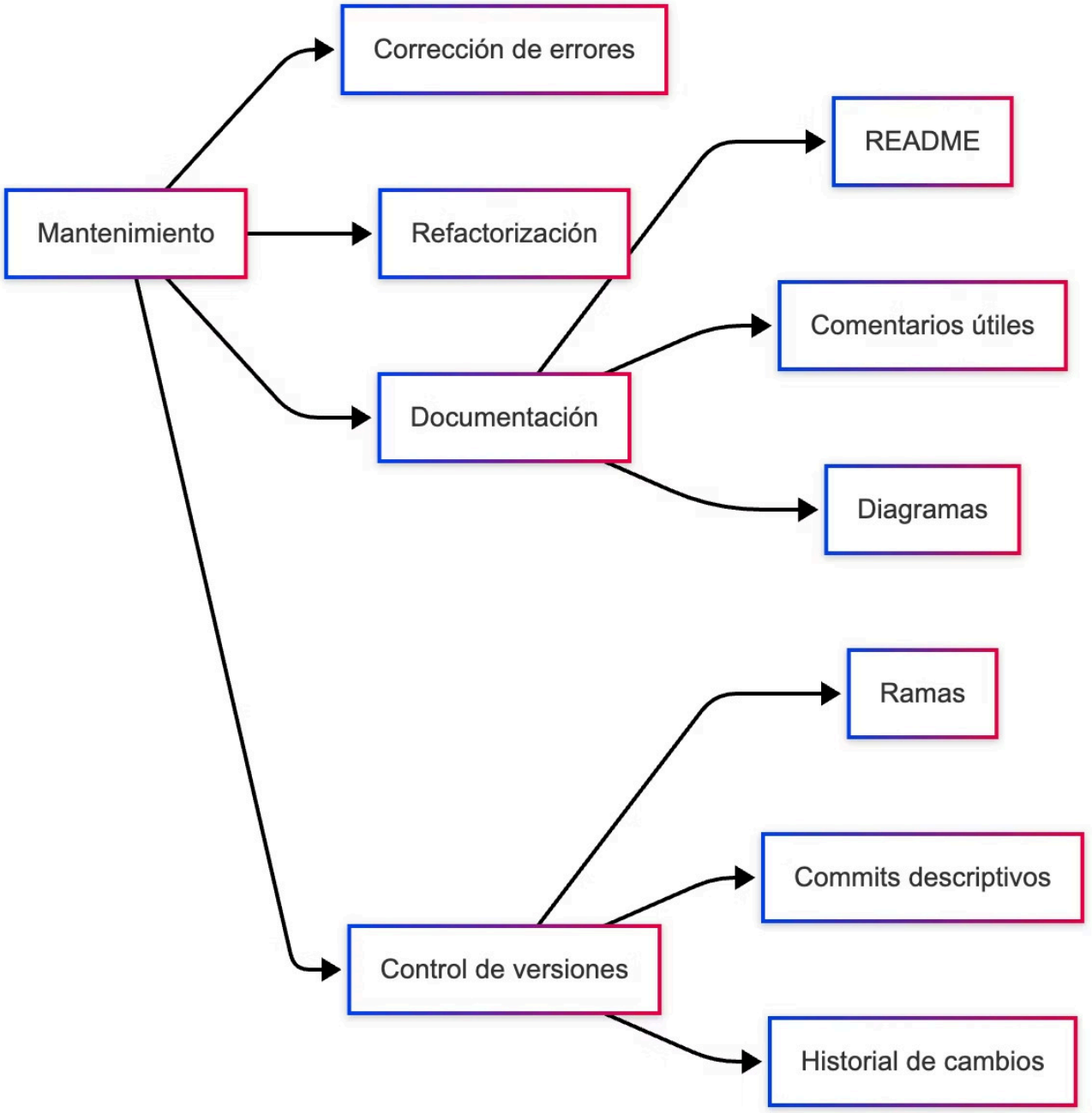
En paralelo, un proyecto saludable requiere **documentación**. Esto no implica escribir manuales interminables, sino mantener actualizada la información mínima imprescindible: cómo ejecutar la aplicación, dependencias necesarias, estructura de carpetas, instrucciones para contribuir, decisiones técnicas relevantes y diagramas que clarifiquen la arquitectura. Una documentación clara evita que el proyecto dependa del conocimiento implícito de unos pocos desarrolladores. Cuando alguien se incorpora al equipo, la curva de aprendizaje disminuye drásticamente.

Control de versiones

Finalmente, el mantenimiento moderno sería prácticamente inviable sin un sistema de **control de versiones**, y en este terreno Git es el estándar absoluto. Git permite registrar cada cambio realizado en el código, recuperar versiones antiguas cuando algo falla, trabajar en ramas aisladas para evitar conflictos y coordinar a múltiples desarrolladores sin bloquearse mutuamente. Además, Git es la base de los flujos de integración continua, revisiones de código y despliegues automatizados, todos ellos imprescindibles para un software profesional.

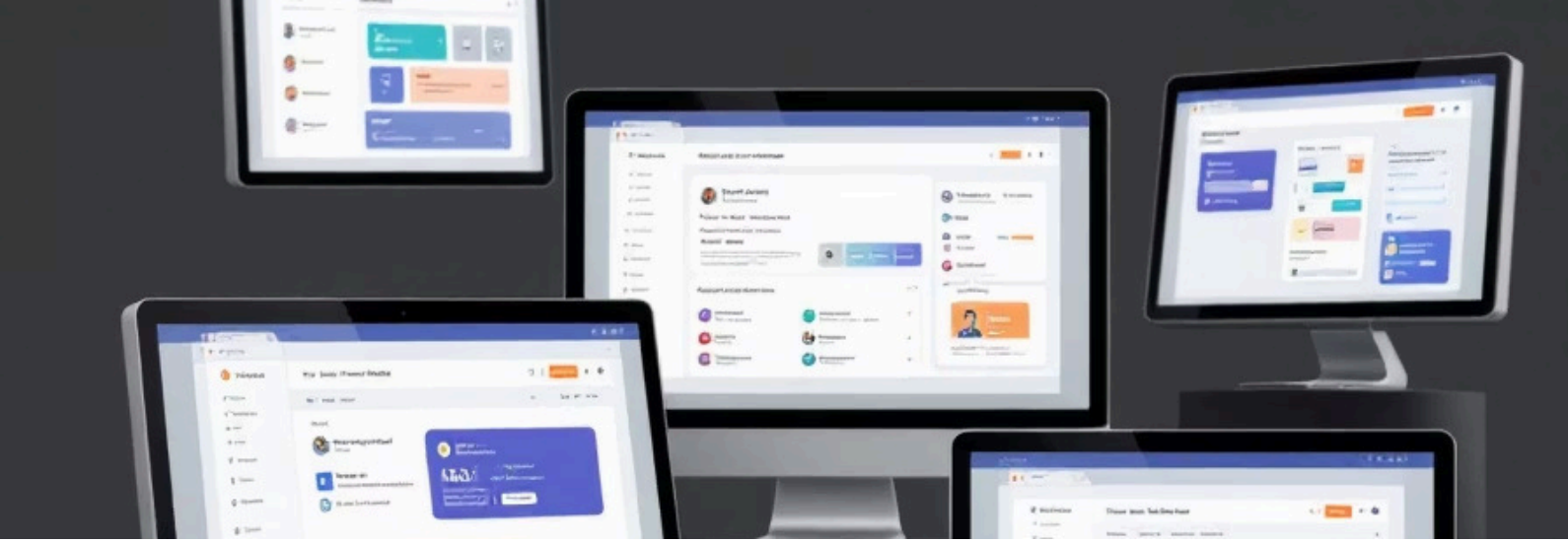
En conjunto, mantenimiento, documentación, refactorización y control de versiones forman un ecosistema que garantiza que el software siga siendo útil, estable y comprensible a lo largo de los años. Ignorar cualquiera de estas piezas aumenta la complejidad, reduce la calidad y eleva los costes del proyecto.

Esquema Visual



Explicación del esquema

- **Mantenimiento** es el concepto central que engloba todas las actividades que mantienen vivo el proyecto.
- **Corrección de errores** gestiona incidencias detectadas en producción o pruebas.
- **Refactorización** mejora internamente el código sin alterar su comportamiento.
- **Documentación** incluye README, comentarios útiles y diagramas que permiten comprensión y continuidad.
- **Control de versiones** organiza cambios mediante ramas, commits y un historial que garantiza trazabilidad.



Caso de Estudio

Proyecto educativo con múltiples desarrolladores

Un equipo de desarrollo gestionaba una plataforma educativa dividida en varios módulos: autenticación, gestión de cursos, reportes de actividad y panel administrativo. La aplicación evolucionaba constantemente, pero el ritmo de crecimiento empezó a poner en evidencia problemas serios.

Contexto

Al revisar el repositorio, se detectaron varios patrones preocupantes:

- Los commits se realizaban directamente sobre la rama principal sin revisión previa.
- No existía un README más allá de un par de notas sueltas.
- Los nombres de las ramas variaban entre desarrolladores, dificultando la organización.
- El código duplicado crecía, especialmente en funciones de validación y carga de datos.

Los nuevos integrantes tardaban días en entender la estructura porque no existía documentación unificada. Los despliegues generaban conflictos y errores difíciles de rastrear. Algunos bugs se repetían porque no estaba claro qué cambios se habían aplicado en cada versión.

Estrategia implementada

01

Adopción de una estrategia de ramas clara

Se definió un flujo con ramas dedicadas:

- main para versiones estables
- develop para integración
- ramas de funcionalidad (feature/...)
- ramas de corrección (fix/...)

02

Commits descriptivos y obligatorios

Se acordó usar mensajes breves pero informativos:

- "fix: validar correo en registro"
- "refactor: extraer lógica de autenticación"

03

Revisiones obligatorias (pull requests)

Ningún cambio podía entrar en develop sin revisión de al menos una persona. Esto detectó errores antes de integrarlos.

04

Documentación mínima pero útil

Se creó un README con:

- instrucciones de instalación
- dependencias
- ejemplos de ejecución
- estructura de carpetas
- pasos para contribuir

También se añadieron diagramas simples de flujo y una guía de estilos de código.

Resultados

En menos de un mes:

- Las integraciones dejaron de generar conflictos críticos.
- Los errores se rastreaban con facilidad gracias al historial claro.
- Nuevos miembros podían comprender el proyecto en horas.
- La calidad del código mejoró gracias a la refactorización progresiva.

El equipo pasó de un desarrollo caótico a uno ordenado, predecible y sostenible.

Herramientas y Consejos

1

Realiza commits pequeños y descriptivos

Un commit debe representar una unidad de cambio coherente. Los mensajes deben explicar *qué* se cambió y *por qué*. Esto facilita revisiones y revertidos.

2

Mantén siempre un README actualizado

Incluye:

- instrucciones de instalación
- dependencias
- cómo ejecutar el proyecto
- comandos útiles
- estructura del repositorio

Un README actualizado evita cientos de dudas y acelera el onboarding.

3

Usa herramientas visuales para aprender Git

Clientes como GitKraken, Sourcetree o GitHub Desktop ayudan a entender el flujo visualmente, especialmente cuando empiezas.

4

Refactoriza en pasos pequeños y controlados

Antes de refactorizar:

- ejecuta los tests
- aplica cambios mínimos
- vuelve a ejecutar los tests

Esto evita introducir errores invisibles.

5

Documenta decisiones técnicas

Una breve explicación en un documento o comentario sobre por qué se eligió cierta arquitectura evita confusiones futuras.

Mitos y Realidades

❌ **Mito:** "Documentar es una pérdida de tiempo."

→ **FALSO.** Documentar reduce la dependencia de conocimiento implícito, acelera la incorporación de nuevos miembros y disminuye errores evitables. Un equipo sin documentación dedica más tiempo respondiendo dudas que desarrollando.

❌ **Mito:** "Refactorizar es peligroso porque puede romper el sistema."

→ **FALSO.** Refactorizar es seguro cuando existe una base sólida de pruebas automatizadas. De hecho, refactorizar reduce el riesgo futuro al mejorar la estructura del código.

Resumen Final para Examen

- El mantenimiento incluye **corrección de errores**, **refactorización** y **documentación**.
- Git es esencial para gestionar cambios, colaborar y mantener un historial confiable.
- La documentación debe ser **breve, clara y actualizada**.
- Una estrategia de ramas bien definida reduce conflictos y facilita integración.
- Refactorizar mejora la calidad interna sin modificar la funcionalidad.