Camyo

Grupo 5

Plan de Gestión de la Calidad del Desarrollo

Devising a Project (#DP)

Miembros del grupo:

- José Ramón Baños Botón
- Isabel X. Cantero Corchero
- Sheng Chen
- Carlos García Martínez
- Carlos García Ortiz
- Raúl Heras Pérez
- Pedro Jiménez Guerrero
- Claudia Meana Iturri
- Rubén Pérez Garrido
- Lucía Pérez Gutiérrez
- Francisco Pérez Manzano
- Diego José Pérez Vargas
- María C. Rodríguez Millán
- Sonia María Rus Morales
- Adriana Vento Conesa
- Jun Yao

Ingeniería del Software y Práctica Profesional • Curso 2024-25

Índice

1. Resumen general

Este documento establece cómo analizar y medir la calidad del desarrollo de la aplicación. Esto incluye la definición de métricas para el análisis del código, la metodología que seguir para la creación, ejecución e interpretación de las pruebas, las condiciones para la aprobación de la incorporación de cambios al código, y pautas para la resolución de problemas durante el desarrollo.

2. Consideraciones iniciales

Algunas de las reglas y recomendaciones definidas en este documento van de la mano de lo definido en la Política de Gestión de Código. Por tanto, antes de leer y poner en práctica el Plan de Gestión de Calidad del Desarrollo, se recomienda al lector que se familiarice con la Política de Gestión de Código.

Para permitir un análisis automático y eficaz de la calidad del código, antes de empezar el desarrollo, el equipo deberá llegar a un acuerdo sobre qué herramientas de análisis estático utilizar. Se proponen como opciones Codacy y SonarQube Cloud.

3. Métricas de calidad

3.1. Problemas por mil líneas de código (kLoC)

Se define como el número de *malos olores* por mil líneas de código. El objetivo de esta métrica es buscar fallos menores que podrían apuntar a un problema más profundo. Esta tarea se hace más fácil con herramientas de análisis estático. Por ejemplo, Codacy busca problemas en las siguientes categorías:

- Estilo de código: Formato de código y problemas de sintaxis, como nombres de variables y el uso de paréntesis y comillas.
- Propensos a errores: Código que podría esconder bugs y palabras clave que se deben usar con cuidado.
- Complejidad: Métodos o funciones complejas que deberían ser refactorizadas.
- Rendimiento: Código que puede provocar problemas de rendimiento

- Compatibilidad: Problemas de compatibilidad entre distintas versiones de navegadores (frontend).
- Código sin usar: Métodos o variables declaradas que no se usan o código muerto.
- Seguridad: Posibles vulnerabilidades de seguridad, como contraseñas codificadas o dependencias vulnerables o desactualizadas.
- Documentación: Métodos y clases que no están documentadas correctamente.
- Mejores prácticas: Código que no sigue las mejores prácticas o guías de estilo.
- Comprensibilidad: Código que podría ser difícil de comprender y modificar.

3.2. Cobertura de código

Se define como el porcentaje de líneas de código o de métodos/funciones que están cubiertos por las pruebas. El objetivo de esta métrica es evaluar qué porcentaje de código ha sido probado para comprobar su correcto funcionamiento. Cuanto más alta sea la cobertura, mayor será la confianza en el sistema y habrá menos sitios en los que se puedan encontrar bugs mayores.

La cobertura se puede medir con servicios en la nube como Codacy o Coveralls, que además ofrecen la opción de ver un historial de la evolución de la cobertura e información detallada sobre qué archivos están mejor cubiertos. Igualmente, durante el desarrollo, es posible medir la cobertura. Algunas herramientas con esta finalidad que analizan código en Java son JaCoCo y Codecov. En cuanto a JavaScript, se recomienda usar Jest.

3.3. Complejidad ciclomática

Se define como una medición de la complejidad lógica de un fragmento de código. El objetivo de esta métrica es determinar cuantitativamente la dificultad de mantener el fragmento de código sobre el que se realizan los cálculos.

Existen herramientas de análisis de código estático, como la anteriormente mencionada SonarQube, que calculan la complejidad ciclomática y sugieren simplificar el código afectado si esta supera cierto límite. Para calcular la complejidad ciclomática manualmente, si se realiza un grafo de control de flujo, se calcula de la siguiente manera:

$$C = E - N + 2P$$

donde **C** es la compejidad, **E** es el número de aristas, **N** es el número de nodos y **P** es el número de componentes conectados. Otra opción es usar la siguiente fórmula simplificada:

C = Número de condiciones + 1.

El equipo de desarrollo debería tratar de mantener la complejidad ciclomática **menor a 15** para todos los métodos. Esto se debe a que un módulo con una complejidad ciclomática mayor que 10 es difícil de probar, entender y modificar. Cuando este número supera 20, el código se vuelve incluso menos mantenible, el riesgo incrementa y las tareas de depuración se vuelven mucho más tediosas. Si se encuentra un método con una complejidad mayor a 15, el plan de acción recomendado es refactorizarlo, dividiéndolo en varios métodos, siempre evaluando si hay alguna parte reutilizable.

3.4. Cumplimiento del alcance

Se define como el nivel al que los requisitos del proyecto se han cumplido. Este se puede medir de dos maneras:

- Como un número entero: se calcula como Requisitos totales - Requisitos cumplidos. El objetivo es que, una vez se haya realizado la entrega final, el valor de esta medida sea **0**.
- Como un porcentaje: se calcula como Requisitos cumplidos / Requisitos totales * 100. El objetivo es que, una vez se haya realizado la entrega final, el valor de esta medida sea **100%**.

3.5. Número de casos de prueba fallados

Se define como el número de tests fallados durante la ejecución de la colección de pruebas. El objetivo es que esta métrica **siempre sea 0 en las versiones desplegadas** de la aplicación, para asegurar que no queda ninguna funcionalidad, parcial o completa, en un estado no funcional o inaplicable, y que cualquier cambio realizado no haya causado daños colaterales a otras partes del sistema.

4. Estilo del código

El estilo de código se define como el conjunto de normas que orientan la manera de estructurar, nombrar y documentar el software que se desarrolla. Su objetivo principal es **garantizar la legibilidad**, la coherencia y la mantenibilidad del proyecto, facilitando así el

trabajo en equipo y el proceso de revisión. A continuación, se exponen las áreas clave que conforman estas buenas prácticas:

Convenciones de Nomenclatura

- Utilizar nombres de variables, clases, métodos y funciones que sean claros y descriptivos.
- Evitar abreviaturas innecesarias o nombres genéricos como `data`, `temp`, `information` o `manager`.

Organización y Estructura de Carpetas

- Agrupar archivos y módulos por su funcionalidad o capa de la aplicación (p. ej., separar controladores, modelos y servicios).
- Mantener una estructura coherente y documentada, de modo que cualquier desarrollador pueda localizar fácilmente los recursos que necesite.

Comentarios y Documentación

- Comentar el código solo cuando realmente es necesario.
- Evitar comentarios redundantes que no aporten información adicional (p. ej., `// Incrementar i` cuando el código es `i++`).

Buenas Prácticas de Codificación

- Mantener métodos y funciones lo más cortos posible; por lo general, que no ocupen el espacio de varias pantallas (con un tamaño de fuente normal y el zoom por defecto).
- Si un bloque de código se hace muy extenso o complejo, considerar su refactorización para mejorar la legibilidad.

Uso de Linters y Análisis Estático

- Integrar herramientas de análisis estático (por ejemplo, **ESLint** o **Checkstyle**) en el flujo de trabajo para detectar incumplimientos de las guías de estilo de forma automática.
- Configurar estas herramientas para que bloqueen el merge en caso de no cumplirse las reglas establecidas, obligando a corregir el estilo antes de integrar nuevos cambios.

5. Pruebas del código

Para garantizar la calidad y fiabilidad del sistema, se llevarán a cabo diferentes niveles de pruebas que cubran los aspectos clave del desarrollo:

5.1. Pruebas Unitarias

Objetivo: Validar la lógica interna de cada componente (funciones, métodos o clases) para detectar errores en etapas tempranas.

Por ello, este tipo de prueba es el más importante y es al que se debe dedicar siempre un mínimo de esfuerzo. **Se espera que todos los desarrolladores realicen pruebas de este tipo** para cada funcionalidad nueva que implementen y que incluyan, al menos, **los casos de uso más frecuentes**. Un conjunto de pruebas extenso y completo deberá incluir todos los casos que se puedan dar, para la **lectura, creación, edición y/o eliminación** de objetos, según las operaciones que sean posibles para el módulo que se está testeando. Asimismo, el conjunto de pruebas deberá incluir **casos positivos y negativos**, haciendo uso de una variedad de entradas, si las hay, para asegurar que estas son adecuadas y se validan correctamente.

Para realizar estas pruebas, los desarrolladores tendrán que prestar especial atención a los *if/else, for* y *while* que tengan muchas ramas de ejecución posibles. De acuerdo con el apartado *3.3. Complejidad ciclomática*, si el proceso de testeo se vuelve complicado porque hay demasiadas condiciones, probablemente sea necesario calcular la complejidad ciclomática y, si es demasiado alta, refactorizar el código afectado.

5.2. Pruebas de Integración

Objetivo: Verificar la comunicación e interacción correcta entre los distintos módulos o servicios de la aplicación (por ejemplo, la capa de negocio y la capa de acceso a datos). De esta forma, se busca asegurar que los componentes, desarrollados de manera individual, funcionen de forma coherente y sin introducir errores al conectarse entre sí.

Al igual que en las pruebas unitarias, se espera que el equipo de desarrollo realice pruebas de integración cada vez que se añada un nuevo módulo o se modifique la forma en que interactúa con otros. Un conjunto de pruebas de integración extenso y completo debe contemplar:

- **Lectura, creación, actualización y/o eliminación** de datos entre los módulos que intervienen en el proceso.

- **Manejo de respuestas y errores** que puedan generarse al comunicarse con servicios externos o internos (APIs, bases de datos, etc.).
- **Validación de formatos de datos** (por ejemplo, JSON, XML), asegurando que las conversiones o deserializaciones no introduzcan errores.

Asimismo, estas pruebas deben considerar **casos positivos y negativos** para cubrir la mayor variedad de escenarios posible:

- **Casos positivos**: Asegurar que, bajo condiciones normales y datos válidos, la integración se ejecute correctamente.
- **Casos negativos**: Validar cómo se comporta la aplicación ante condiciones de error (tiempos de espera, datos malformados, caídas de servicios externos, etc.).

Para llevar a cabo las pruebas, los desarrolladores y responsables de calidad deberán prestar especial atención a aquellos **puntos críticos de integración** (por ejemplo, llamadas a funciones entre diferentes microservicios) y a la **secuencia de pasos** que permiten completar un flujo de negocio. En caso de detectar demasiadas dependencias o conexiones complejas, se recomienda realizar un **análisis de arquitectura** para identificar posibles mejoras o refactorizaciones que faciliten el proceso de integración.

5.3. Pruebas de Interfaz

Objetivo: Verificar el correcto funcionamiento de los distintos elementos de la interfaz de la aplicación (botones, menús, formularios, etc.), así como la navegación entre las diferentes pantallas o secciones, con el fin de asegurar una experiencia de usuario óptima y coherente.

De la misma manera que se llevan a cabo pruebas unitarias e integradas, se espera que el equipo de desarrollo y/o el equipo de diseño-UX realice pruebas de interfaz siempre que se introduzcan cambios relevantes en el frontend. Un conjunto de pruebas de interfaz extenso y completo debe contemplar:

- **Elementos interactivos**: Comprobar que todos los controles (botones, campos de texto, selectores, etc.) respondan correctamente a las acciones del usuario.
- **Validaciones de formularios**: Incluir casos de entradas válidas e inválidas, asegurando que se muestren mensajes de error adecuados y que no se produzcan comportamientos inesperados.
- **Flujos de navegación**: Garantizar que los enlaces o transiciones entre pantallas reflejen el flujo de trabajo esperado y no existan rutas rotas.

- **Diseño y coherencia visual**: Verificar que la maquetación, tipografía, paleta de colores y elementos gráficos cumplan con las guías de estilo definidas, adaptándose correctamente a distintos tamaños de pantalla o dispositivos, si procede.

Además, para facilitar y agilizar estas pruebas, se pueden emplear herramientas de **automatización** (como Selenium) que permitan simular la interacción de un usuario real con la interfaz.

Si el proceso de verificación en la interfaz se vuelve complicado o presenta numerosos puntos de fallo, puede ser necesario **refactorizar** ciertos componentes o simplificar el flujo de navegación para mejorar la mantenibilidad y la experiencia de usuario.

Con este enfoque, las pruebas de integración y de interfaz se complementan con las de unidad, ofreciendo una visión más global de la calidad del sistema y reduciendo significativamente la probabilidad de errores en producción.

6. Aprobación de cambios

Una vez se haya realizado un cambio en el software que vaya a ser integrado a la rama principal, ya sea por la finalización de una tarea o por un error que necesite ser remediado urgentemente, es necesario comprobar que estos no provocan fallos en otras partes del sistema. Según la Política de Gestión de Código:

El encargado de una tarea creará sus propias pull requests. *Él mismo, o en su defecto, los coordinadores, asignarán a un revisor entre los miembros del proyecto, para revisar cualitativamente el contenido de la misma y el cumplimiento con los requisitos*.

El revisor tendrá la responsabilidad de comprobar el correcto funcionamiento del sistema con los cambios implementados por la persona encargada de la tarea. Para facilitar las tareas de revisión, **el repositorio deberá tener un *workflow* que ejecute automáticamente la colección de pruebas cuando se realice una *pull request*.** Si este no está configurado, el revisor deberá ejecutar las pruebas en su entorno. En el caso de que fallen algunas pruebas, el revisor deberá dejar abierta la *pull request* e indicar los errores en un comentario, tal y como se establece en la Política de Gestión de Código.

Si los cambios realizados no están cubiertos por nuevos casos de prueba, o la cobertura es insuficiente, se recomienda que el revisor haga pruebas informales para asegurar que las nuevas funcionalidades añadidas, si las hay, funcionan correctamente, se ajustan a los requisitos (si no son una implementación parcial) y no afectan a otros componentes relacionados.

Antes de aprobar cambios, también es necesario revisar que el código escrito sigue las buenas prácticas y las guías de estilo propias del lenguaje de programación. Para esto, existen *workflows* de GitHub para realizar *linting* y comprobar la sintaxis del código que pueden resultar útiles. Igualmente, se pueden configurar **para que no se permita hacer *merge* si el código no cumple las guías de estilo**.

Una vez el revisor se haya cerciorado de que todo está en orden tras la realización de los cambios, aprobará la *pull request*, hará *merge* y cerrará la tarea.

7. Resolución de problemas

Cuando se detecta un problema (bug, incidencia en producción, error en la compilación, etc.), se seguirán las siguientes pautas:

1. **Notificación y Registro**

   Todo problema debe registrarse en GitHub como una nueva *issue*. Desde el repositorio, se debe describir el problema de forma detallada.
   Según la Política de Gestión del Código, es necesario incluir la siguiente información, siguiendo el formato de la plantilla de incidencias: título, descripción, pasos para reproducir, resultado esperado, resultado obtenido, evidencias (si hay), entorno en el que se ha dado la incidencia, posibles, percepción de prioridad y, opcionalmente, comentarios adicionales.

2. **Asignación y Priorización**

   Asignar un nivel de prioridad (**crítica, alta**, **media**, **baja**). Asignar una persona responsable a la *issue*, teniendo en cuenta la experiencia, la disponibilidad y el equipo al que pertenece. Según la Política de Gestión de Código, si hay varias personas responsables, se tendrán que crear varias *issues* para repartir el trabajo.

3. **Análisis y Solución**

   Revisar la sección de logs, la configuración del entorno y el historial de cambios para identificar la causa raíz del problema. Proponer uno o varios enfoques de solución, documentando los cambios necesarios.

4. **Pruebas de Verificación**

Una vez implementada la solución, crear o actualizar las pruebas unitarias o de integración que cubran el problema. Ejecutar las pruebas y confirmar que el bug se ha resuelto sin introducir regresiones.

5. **Cierre de la Incidencia**

   Documentar la solución final en la herramienta de seguimiento (incluyendo *commits* o *pull requests* asociados). Notificar a los interesados y al equipo si es necesario (p. ej., si el problema afecta a funcionalidades clave).

6. **Lecciones Aprendidas**

   Si se trata de un problema de alto impacto o que pudo haberse evitado con un proceso de control de calidad más estricto, se recomienda abrir un apartado de *lecciones aprendidas* para mejorar los procesos de análisis, pruebas o revisión de código.

8. Otros consejos para asegurar la calidad durante el desarrollo

8.1. Inspección y análisis de código

Como parte del desarrollo, se aconseja realizar una revisión del código. Esta actividad ya se mencionó como una parte obligatoria del proceso de aprobación de cambios. Si se trata de código escrito durante una tarea que está asignada a varias personas, los otros miembros del subgrupo también son responsables de hacer al menos una inspección simple.

Las herramientas de integración continua automatizan el proceso de análisis estático de código, permitiendo encontrar malos olores y analizar una de las métricas de calidad (véase 3.1. Problemas por mil líneas de código (kLoC)).

8.2. Estar al tanto de los riesgos

Identificar posibles riesgos con las actividades del día a día, registrarlos y evaluarlos. Una detección temprana y la elaboración de una respuesta adecuada ayudan a reducir costes y tareas adicionales de depuración, pruebas y aprobación de nuevas funcionalidades o *hotfixes* que se podrían haber evitado.

8.3. Documentar código complejo

Si es necesario escribir un módulo particularmente complejo, se recomienda documentarlo adecuadamente con el fin de facilitar las tareas futuras de testeo y depuración y la adición de cambios por un miembro del equipo que no haya escrito el módulo originalmente.

8.4. Fomentar la comunicación y colaboración constante

Establecer breves sesiones de sincronización para compartir el estado de cada tarea, bloqueos y avances. Facilitar que los integrantes del equipo pregunten o soliciten apoyo cuando se enfrenten a problemas técnicos o de conocimiento. Usar las herramientas de comunicación del equipo, como Microsoft Teams, para tratar dudas rápidas, en lugar de acumular incidentes que puedan ralentizar el proyecto.

8.5. Verificación de dependencias y seguridad

Revisar con frecuencia las versiones de librerías y frameworks utilizados para evitar vulnerabilidades. Configurar análisis de seguridad que detecten dependencias inseguras, desactualizadas o con licencias conflictivas. Incluir estas verificaciones en el *pipeline* de CI, bloqueando la integración de cambios si se detectan vulnerabilidades críticas.

8.6. Mantener un registro de cambios

Con el uso de un registro de cambios (changelog), se busca documentar de forma clara y concisa las funcionalidades nuevas, las correcciones de errores y otros cambios relevantes en cada versión, de modo que los usuarios y el equipo tengan visibilidad de la evolución del proyecto.

8.7. Auditorías de calidad periódicas

El objetivo es revisar de manera planificada (en cada entrega principal) que los estándares de calidad se estén cumpliendo. Para ello, se realizará:

- Revisión de métricas clave (kLoC, cobertura de código, complejidad ciclomática, número de casos de prueba fallados).
- Validación del cumplimiento de la política de gestión de cambios (code reviews, pull requests, etc.).

- Análisis de la deuda técnica pendiente y la posible necesidad de refactorizaciones importantes.

Resultado: un informe de auditoría que recoja conclusiones y acciones recomendadas para el siguiente ciclo de desarrollo.

8.8. Control de defectos y requisitos no finalizados

El objetivo es asegurar la trazabilidad de todas las incidencias y tareas no completadas o en estado de "pendiente".

Elementos clave:

- **Registro detallado**: Asociar cada defecto a un requisito incumplido, con su respectivo estado (abierto, en progreso, resuelto, cerrado).
- **Métricas de defectos**: Llevar conteos de defectos abiertos y su antigüedad, priorizar la corrección de aquellos de mayor impacto y analizar patrones para tomar medidas preventivas.
- **Retroalimentación continua**: Integrar estos hallazgos en la planificación de sprints, reforzando las áreas que se demuestren más propensas a errores.

9. Consistencia entre los distintos miembros del equipo

Objetivo: Asegurar que todos los integrantes del proyecto sigan los mismos criterios, guías de estilo y metodologías de desarrollo, de modo que el producto resultante sea uniforme y mantenible.

Definición de estándares claros: Establecer guías de estilo (naming conventions, formateo, documentación) y publicar estos documentos en un espacio accesible a todo el equipo. Adoptar una estructura de carpetas y módulos consensuada, de forma que la organización del código sea homogénea.

Formación y alineación continua: Impartir sesiones breves de capacitación o talleres internos para dar a conocer nuevas herramientas, patrones o reglas de codificación. Revisar periódicamente si las pautas de estilo y las prácticas de desarrollo se cumplen durante las revisiones de código.

Manejo de desacuerdos y feedback: Fomentar la comunicación abierta, donde cualquier integrante pueda señalar incoherencias o proponer mejoras. Los líderes de frontend y

backend serán responsables de resolver dudas sobre el estilo o la arquitectura, evitando duplicar esfuerzos o introducir variaciones arbitrarias.

La aplicación de estas medidas tendrá como resultado la producción de un código más legible, coherente y fácil de entender para todos.

10. Monitorizar y reportar métricas de calidad

Objetivo: Medir de manera sistemática el estado de la calidad (por ejemplo, cobertura de código, número de defectos, complejidad ciclomática) y realizar un seguimiento continuo que permita tomar decisiones informadas y detectar tendencias.

Selección de métricas relevantes: Escoger indicadores alineados con los objetivos del proyecto: casos de prueba fallados, problemas por mil líneas de código (kLoC), cobertura de código, etc. Incluir alguna métrica de rendimiento o seguridad (tiempos de respuesta, vulnerabilidades detectadas, etc.).

Automatización de la recogida de datos: Hacer uso de herramientas de integración continua que generen reportes automáticos cada vez que se hace un commit o pull request.

Análisis y visualización: Revisar periódicamente los resultados para identificar patrones o variaciones significativas. Un miembro del equipo de aseguramiento de calidad será responsable de consolidar los datos y presentarlos de forma clara durante las reuniones de equipo.

Acciones correctivas: Cuando se observe un deterioro en alguna métrica (por ejemplo, disminución de la cobertura de código), proponer planes de acción concretos (refactorización, aumento de pruebas, optimizaciones). Recalibrar las metas o umbrales de calidad según la evolución del proyecto y la naturaleza de los cambios.

Beneficios: Visibilidad real del estado de la calidad, evitando basarse únicamente en percepciones o suposiciones. Capacidad de reaccionar a tiempo ante posibles riesgos y de comunicar avances de manera objetiva a los usuarios piloto.

11. Dedicar tiempo en las reuniones a hablar sobre calidad

Objetivo: Incluir de forma sistemática la discusión sobre la calidad en las reuniones ya existentes (retrospectivas, etc.), favoreciendo una mejora continua.

Agenda fija para la calidad: Reservar unos minutos en cada reunión de seguimiento para repasar brevemente la situación de las métricas, incidencias abiertas o bloqueos relacionados con la calidad. Incluir en las retrospectivas un apartado específico en el que se aborden los éxitos y fracasos en las actividades de aseguramiento de la calidad.

Identificación de problemas y propuestas de mejora: Facilitar que los miembros del equipo expresen preocupaciones o hallazgos: partes del código con alta complejidad, pruebas unitarias insuficientes, etc. Proponer acciones concretas y asignar responsabilidades.

Evaluación del progreso: Revisar la eficacia de las acciones tomadas en reuniones anteriores, valorando si los cambios han tenido el impacto deseado. Ajustar la estrategia de calidad en caso de resultados insatisfactorios o cambios en el alcance del proyecto.

Beneficios: Fomentar la cultura de la calidad como una prioridad compartida, no relegada a un momento puntual. Asegurar la detección temprana de riesgos y mantener al equipo alineado con los objetivos de calidad, impulsando la mejora continua.

12. Conclusión

El presente **Plan de Gestión de la Calidad** aborda de manera integral las medidas, pautas y herramientas necesarias para garantizar un desarrollo de software estable, mantenible y en línea con los objetivos propuestos. A través de métricas claras (problemas por mil líneas de código, cobertura de pruebas, complejidad ciclomática, etc.) y de procesos bien definidos (pruebas unitarias, integración e interfaz, revisión de código y gestión de incidencias), se instaura una base sólida que permite al equipo:

- **Identificar** áreas problemáticas con prontitud y corregirlas antes de que se conviertan en fallos costosos.
- **Monitorear** de forma continua la calidad y tomar decisiones informadas apoyadas en datos objetivos.
- **Mantener** un entorno de colaboración que fomente la consistencia, el aprendizaje y la mejora continua.

- **Adecuarse** ágilmente a cambios en el alcance, nuevas necesidades o actualización de tecnologías, minimizando riesgos y costes de retrabajo.

Además, las recomendaciones complementarias, como la inspección y análisis de código, el seguimiento de dependencias y vulnerabilidades, la adopción de una estrategia de versionado y la promoción de una cultura de mejora continua, refuerzan la visión de calidad a lo largo de todo el ciclo de vida del proyecto. Así, se generan ciclos de retroalimentación continua, se amplía la participación de todos los involucrados en la definición y seguimiento de la calidad, y se construye un producto final que responda de forma óptima a las expectativas de los stakeholders.

Con esta implementación, el equipo de desarrollo asegura que la calidad no sea únicamente un hito al final del proyecto, sino un **pilar transversal** presente en cada fase: desde la concepción y planificación, hasta la entrega y la posterior evolución del software.