Desarrollo y versionado de código

Software Developers playbook

Transformación desarrollo de software - #ONE

Control de versiones

Versión	Fecha de Creación	Responsable	Descripción
1.0	{17-3-2023}	GSD (Global Software Development)	Primera versión oficial de los playbooks

Índice

1. Introducción

1.1. Acerca de este playbook

Este playbook (parte de una serie de seis) es un componente fundamental para la práctica de desarrollo de software en el Banco que condensa las prácticas y directrices que los ingenieros de software debemos llevar a cabo durante el proceso de desarrollo y versionado de código.

El playbook debe ser actualizado regularmente basándose en las prácticas y el panorama tecnológico del Banco. El equipo responsable del documento será el responsable de actualizarlo.

Figura 1 - Ciclo de vida del desarrollo de software

1.2. Principios básicos

Este playbook se guía por un conjunto de principios que todos los desarrolladores deben seguir:

- **Estandarización:** Usar una taxonomía global y formas de trabajo alineadas entre los equipos y geografías permitirá a los desarrolladores colaborar mejor, compartir mejor el conocimiento y rotar entre los equipos.
- **Calidad sobre cantidad:** Escribir código de calidad debe ser nuestra prioridad ya que la mayor parte de nuestro tiempo lo empleamos manteniendo software heredado y abordando deuda técnica (Maxim and Pressman), cumplir con esto mejorará la velocidad de desarrollo de los equipos a medio y largo plazo.
- **Transparencia y trazabilidad:** El total control y visibilidad de los sistemas y el código del Banco mejora la experiencia de los desarrolladores para navegar en los distintos entornos, responder a incidentes y garantizar el cumplimiento de los requisitos regulatorios.
- **Propiedad:** La alta demanda de calidad requerida en los servicios críticos del banco implica un control detallado de responsabilidad y propiedad de los activos de código. “Debemos ser propietarios del código que desarrollamos y sentirnos orgullosos de ello”.
- **Pequeños incrementos frecuentes frente a grandes cambios:** Grandes cantidades de código pueden poner en riesgo la estabilidad de nuestros sistemas ya que son más difíciles de probar y deshacer. Además, la mayoría de los sistemas de software evolucionan en función de los comentarios de los usuarios finales y los requisitos son difíciles de anticipar de antemano. Por tanto, es importante desarrollar rápidamente pequeños incrementos para aprender rápidamente y rectificar.
- **Simplicidad:** Los sistemas de software rara vez se escriben una vez y no se modifican. Más bien, muchas personas trabajan con el mismo sistema a lo largo del tiempo. Por eso es importante buscar las soluciones más sencillas que puedan comprenderse fácilmente y evolucionar en el futuro.
- **Automatización:** Los procesos manuales que son repetitivos deben automatizarse para mejorar nuestra productividad, enfocarnos en el desarrollo y reducir los errores mediante la intervención manual.

1.3. Niveles de exigencia

Este playbook utiliza intencionadamente las siguientes tres palabras para indicar los niveles de exigencia según la norma **RFC2119**:

- **Debe** - Significa que la práctica es un requisito absoluto, salvo las excepciones que se autoricen de forma específica.
- **Debería** - Significa que pueden existir razones válidas en circunstancias particulares para ignorar una práctica, pero deben comprenderse todas las implicaciones y sopesar cuidadosamente antes de elegir un camino diferente.
- **Podría** - Significa que una práctica es realmente opcional.

2. Prácticas

Todos los equipos de desarrollo de software deben seguir las **siete prácticas** siguientes relativas al desarrollo y versionado de código, independientemente de la tecnología y el lenguaje de programación utilizados:

1. Adoptar un sistema de control de versiones
2. Estandarizar el modelo de ramas
3. Escribir código de calidad
4. Escribir código seguro
5. Almacenar y evolucionar la documentación junto al código
6. Adoptar mecanismos de revisión de código
7. Instaurar un modelo InnerSource

A continuación se definen cada una de las prácticas siguiendo la siguiente **estructura**:

- Sus beneficios (que conseguimos).
- Precondiciones (condiciones para poder aplicar la práctica).
- Adopción (cómo implementarla).
- Herramientas necesarias.
- Indicadores para los desarrolladores (como la medimos).
- Enlaces de interés.

2.1. Adoptar un sistema de control de versiones

Los equipos de desarrollo de software deben adoptar un sistema de control de versiones y aplicar las directrices recogidas en esta sección cuando la tecnología lo permita.

Beneficios

Los beneficios que aporta adoptar un sistema de control de versiones en todos los equipos de software son los siguientes:

- Mejorar la trazabilidad del código.
- Garantizar el cumplimiento de requisitos de auditoría y calidad requeridos en nuestro entorno bancario.
- Fomentar la colaboración y el trabajo en equipo.
- Identificar al/los responsables del código (ownership).
- Recuperar trabajo en caso de perderlo.

Precondiciones

Para poder aplicar la práctica se deben dar las siguientes **precondiciones**:

- La tecnología debe permitir almacenar el código en un repositorio externo, algunas tecnologías lo hacen de forma embebida en sus sistemas (e.g., Salesforce, SAP). Los equipos que usan estas tecnologías pueden decidir aplicar esta práctica o no.

Adopción

2.1.1. Adoptar un sistema de control de versiones

Para llevar a cabo la **adopción de esta práctica** se deben cumplir las siguientes directrices generales:

- Los **equipos deben** trabajar bajo un único servicio de control de versiones global**.**
- Todos los **repositorios deben** tener un servicio y equipo propietarios.
- Los **repositorios deben** contener un fichero de metadatos donde se definen propiedades relacionadas con su uso y funcionalidad (e.g., responsable, modelo de branching adoptado, tecnologías empleadas, servicios asociados).
- Los **repositorios deben** ser autocontenidos, es decir, referencias a otros repositorios o código fuente se hará en forma de dependencias.
- Los **commits** de los desarrolladores:
    - **Deben** contener el **correo electrónico corporativo** en el campo de autor.
    - **Deben** estar **identificados por la historia de usuario de Jira** añadiendo el identificador de la historia en la descripción del commit. De forma excepcional, las tareas cortas que no tengan Jira asociado (e.g., arreglar un comentario en el código) podrán marcarse como tareas sin Jira asociado.

    - **Deben** contener **caracteres ASCII** en la descripción y se recomienda seguir las guías establecidas en Conventional Commits.
- Todo **artefacto** desplegado en producción **debe** tener un tag asociado.
- Las **ramas** compartidas (aquellas compartidas por los desarrolladores) **deberían** mantener una historia inmutable.

A la hora de **almacenar información en nuestro repositorio** los equipos deben cumplir las siguientes directrices:

- Los repositorios **deben** contener los elementos que permiten a los equipos construir los artefactos que despliegan en producción. Esto incluye (siempre que existan) el código fuente, archivos de configuración, datos de prueba, scripts de base de datos, scripts de construcción y scripts de infraestructura como código (IaC).
- Las contraseñas en bruto o los secretos **no deben** almacenarse en los repositorios.
- Los archivos binarios ejecutables (e.g., jar, exe) y recursos generados por la construcción del código **deben** excluirse del repositorio.
- Los archivos de gran tamaño (más de 30 MB) **no deben** versionarse en el repositorio. La herramienta limitará este tipo de comportamientos.

El **ciclo de vida de los repositorios** se rige por los siguientes criterios:

- Los equipos **pueden** crear nuevos repositorios dentro de su servicio siempre que lo crean conveniente.
- Se **debe** definir un modelo global de permisos y grupos de los repositorios.
- Los nombres de los repositorios **deben** seguir las convenciones descritas a continuación, los repositorios que no sigan la convención serán renombrados progresivamente:
    - Se usarán caracteres alfanuméricos.
    - Se usarán caracteres en minúscula.
    - Se usarán guiones como elemento separador.
- Antes de crear un nuevo repositorio los dueños de los repositorios **deben** revisar la lista de bootstraps por servicio disponible, dicha lista será revisada y mantenida por el service owner de cada servicio. Dicha lista será accesible por todos los equipos y los equipos **pueden** contribuir a ella dentro de las reglas definidas para su contribución.
- Los repositorios **deben** ser accesibles por cualquier miembro de la organización, aquellos que no puedan ser accesibles por motivos de seguridad o sensibilidad se analizarán de forma independiente.
- Se aplicarán dos tipos de hooks con el objetivo de favorecer el desarrollo y fomentar la aplicación de las prácticas establecidas en el playbook:

    - **Servidor:** Se ejecutan en el lado del servidor. Este tipo de Hooks son mantenidos y establecidos por los propios servicios de forma centralizada y homogénea en sus repositorios, algunos hooks definidos son los siguientes:
        - Hook para identificar contraseñas en el código versionado.
        - Hook para validar que el tamaño de los ficheros no supera los 30MB.
        - Hook para validar que no se suben ficheros binarios al repositorio.
        - Hook para validar que los commits tienen un Jira asociado.
        - Hook para garantizar que el autor del commit es válido.
        - Hook para validar que los mensajes de los commits cumplen con las convenciones establecidas.
    - **Locales**: Hooks que se ejecutan de forma local. Los equipos son responsables de implementar estos hooks pudiendo aplicar sus propios hooks además de los ya disponibles en el lado del servidor afianzando un modelo "shift-left".
- Los repositorios en desuso **deben** ser archivados por el equipo owner del repositorio. El servicio debería proveer de un alertado a los equipos para que se archiven dichos repositorios.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket**: Sistema principal y central del Banco para el versionado del código. Se establecerá un único sistema global a modo de servicio único.
- **Jira:** Servicio global para la gestión de proyectos.
- **Samuel:** Repositorio unificado de enforcement automático.
- **Equipo local:** El entorno local de cada desarrollador debe soportar la adopción de esta práctica. Debe incluir todas las herramientas y librerías necesarias para poder desempeñar su trabajo (e.g., Git, Python, Maven, Gradle).
- **Sistema de identidad (pendiente definición):** Herramienta global para la gestión de roles, grupos, privilegios y autenticación.
- **Bootstrapping (pendiente definición):** Componentes y catálogo de bootstrap para los repositorios de código.
- **Gobierno metadatos repositorios (pendiente definición):** Herramienta para la gestión de metadatos asociados a los repositorios..

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción

Artefactos desplegados con tag asociado	% artefactos desplegados con un tag asociado
Adopción bitbucket	% proyectos usando bitbucket global
Commit identificados	% commits identificados con un email el Banco válido
Repositorios públicos	% repositorios públicos en el equipo
Repositorios con owner	% de repositorios con responsable (servicio y equipo)
Trazabilidad funcional	% de commits relacionados con una Historia de Usuario

Enlaces

- https://martinfowler.com/articles/continuousIntegration.html#MaintainASingleSourceRepository
- https://gist.github.com/luismts/495d982e8c5b1a0ced4a57cf3d93cf60
- https://www.conventionalcommits.org/es/v1.0.0/

2.2. Estandarizar el modelo de ramas

Beneficios

Estandarizar varios modelos de control de ramas en el Banco aporta los siguientes beneficios:

- Facilitar y simplificar automatismos.
- Medir la actividad de desarrollo.
- Reducir la curva de aprendizaje de los desarrolladores y el tiempo de onboarding.
- Poder aplicar estándares de mercado favoreciendo y apoyando el conocimiento de la comunidad.
- Fomentar la colaboración entre equipos, unidades y geografías y por tanto mejorando la movilidad entre equipos.

Precondiciones

Para poder aplicar la práctica se deben dar las siguientes precondiciones:

- Haber adoptado la práctica 2.1 y por tanto tener un sistema de control de versiones.

Adopción

2.2.1. Estandarizar el modelo de ramas

Los equipos **deben** seleccionar, de entre uno de los estándares de mercado seleccionados, un modelo de control de ramas.

Las herramientas y procesos en el Banco **deben** ser independientes del modelo de ramas seleccionado. Las herramientas no pueden limitar la forma en la que el desarrollador interactúa con el repositorio de código, no se **debe** limitar la creación de ramas directamente en los repositorios de código, así mismo tampoco **debe** existir un proceso que impida al desarrollador crear ramas libremente, el único requisito es seguir el branching model acordado.

Las herramientas, procesos y plataforma del Banco **deberían** facilitar el trabajo en el modelo seleccionado por el equipo.

Los modelos de gestión de ramas recomendados en el Banco son los siguientes:

- **Gitflow**
- **Scaled trunk based**
- **Trunk based** (ir a trunk-based)
- **Mainframe** (Pendiente definir modelo estándar)

En los enlaces de cada modelo se describe el flujo que los equipos **deberían** seguir para cada uno de ellos.

Se establecerá un árbol de decisión para facilitar la selección de un modelo de gestión de ramas, que los equipos dueños de los repositorios **deberían** seguir.

Se recomienda que los equipos apliquen una de las dos variantes de trunk based siempre que sea posible, ya que se ha identificado como uno de los modelos que más beneficios trae para el desarrollo (Accelerate, Kim et al., 2018).

Las siguientes directrices deben aplicarse por parte de los equipos para cumplir con la práctica:

- En el momento de crear los repositorios el equipo owner **debe** decidir el modelo de gestión de ramas y se creará un repositorio con los permisos adecuados por rama y una estructura inicial de rama como base alineada con los flujos descritos en esta sección. Existirán bootstraps (definidos en la práctica 2.1) que facilitarán la creación de forma automática de cada uno de los modelos de ramas elegidos a disposición de los equipos.
- A modo informativo, el modelo de branching strategy empleado por los equipos **debe** estar definido en el fichero README de los repositorios. Esto permite que los desarrolladores puedan identificar el modelo de trabajo inmediatamente. Se definirá usando la siguiente propiedad (branching_model:"gitflow | scaled-tbd | tbd").

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket**: Sistema principal y central del Banco para el versionado del código. Se establecerá un único sistema global a modo de servicio único.
- **Bootstrapping:** Herramienta pendiente de definición.
- **Gobierno metadatos repositorios:** Herramienta o solución pendiente de definición.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo ramas abiertas	Tiempo medio (en días) que la rama está abierta (no main branch)
Número de ramas abiertas	Número de ramas abiertas (no main branch) por semana
Porcentaje de repos con el branching model establecido	% repositorios con un modelo de branching establecido en sus metadatos
Porcentaje de repositorios que siguen el modelo de ramas establecido	% repositorios que siguen el modelo de ramas establecido (e.g., no se realizan commits a ramas release en TBD)

Número de ramas sin actividad	Número de ramas que no han tenido actividad en los últimos 3, 10 o 20 días

Enlaces

- https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow (Gitflow)
- https://trunkbaseddevelopment.com/ (scaled trunk based)
- https://paulhammant.com/2013/12/04/what_is_your_branching_model/ (trunk based)
- https://martinfowler.com/articles/branching-patterns.html (branching patterns)

2.3. Escribir código de calidad

La mayor parte de nuestro tiempo lo dedicamos a resolver deuda técnica o mantener / mejorar componentes de software ya existentes. Por tanto, es de extrema importancia escribir código de calidad desde el primer momento que comenzamos a desarrollar software. Para facilitar esta labor es necesario definir una serie de directrices globales.

Beneficios

Adoptar estos criterios de calidad de código globales mejora la forma en la que desarrollamos código lo que aporta múltiples beneficios:

- Facilitar la lectura del código.
- Mejorar la eficiencia del código.
- Mejorar la mantenibilidad del código.
- Eficientar la detección de errores.
- Aumentar la productividad ya que reducimos el tiempo necesario para añadir nuevas funcionalidades o modificaciones al código.
- Evitar romper funcionalidad existente al añadir nueva.
- Incrementar la colaboración.

Adopción

2.3.1. Escribir código de calidad

Los equipos deben adoptar las siguientes directrices relacionadas con la práctica:

- Todos los equipos **deben** seguir las guías de estilo y buenas prácticas globales descritas para las diferentes tecnologías existentes en el Banco. Además, los equipos **pueden** adoptar y ajustar sus propias prácticas e incluso colaborar en las prácticas globales de cada tecnología.
- Los desarrolladores **deben** conocer los principios y patrones de diseño de software y deben aplicarlos de forma continua en todo su trabajo (Clean Code, Martin, 2009). Para conseguir construir sistemas que sean flexibles y se puedan adaptar a los requerimientos cambiantes **deben** satisfacer las siguientes reglas:
    - Los tests se pasan y la suite de pruebas es completa.
    - El código revela su propósito, es auto explicativo.
    - No existe duplicidad.
    - El sistema debe mantenerse simple y satisfacer las necesidades actuales.
- Las aplicaciones **deben** usar el versionado semántico descrito aquí.
- Los equipos **deben** evitar añadir deuda técnica y en caso de haberla **deben** priorizar su resolución.
- Para cumplir con estas reglas se deberían apoyar en las prácticas de XP Programming, los principios SOLID y otras reglas básicas como DRY, YAGNI, KISS y Unix philosophy.

- Para facilitar la aplicación de las guías y buenas prácticas los ingenieros **deben** incluir herramientas en nuestro IDE o entorno local que permitan integrar esas guías de estilo de forma automatizada (e.g., checkstyle, PMD, lint). Estas reglas se ejecutarán también como parte de los pipelines de construcción de código (playbook 4 - Análisis y construcción de código).
- Los desarrolladores **deben** refactorizar el código cuando identifiquen problemas de diseño o "code smells" (e.g., código duplicado, funciones muy largas, clases con demasiadas responsabilidades, acoplamiento excesivo).
- Para mejorar la calidad de nuestro código los equipos **deben** ser estables y equilibrados en cuanto a experiencia desarrollando software, es decir, un equipo sin perfiles senior probablemente no producirá la misma calidad de código que un equipo con al menos un perfil senior.
- La formación de buenas prácticas desarrollando código es otra herramienta que **deberíamos** adoptar en los equipos. Además, los equipos **pueden** incluir referencias en sus README a guías de referencia internas (el Banco) o externas.
- Garantizar que nuestro código cumple la calidad establecida en el Banco requiere de cierto compromiso por parte del equipo y este compromiso **debería** estar reflejado en el Definition of Done fijado por el equipo en cuestión.
- Los equipos maduros o que contienen algún perfil senior **pueden** adoptar mecanismos de TDD para mejorar la calidad de su código. Está demostrado que la correcta aplicación de este mecanismo mejora la calidad del código y la productividad de los equipos (Beck, 2003).

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **IDE:** El entorno de desarrollo integrado de los desarrolladores **debe** proveer de las herramientas necesarias para aplicar las convenciones de código descritas.
- **Lintern, PMD y checkstyles:** Los desarrolladores **deben** integrar herramientas de análisis de código locales para mejorar la identificación de desviaciones respecto a las guías de calidad establecidas.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Número de bugs reportados	Número de bugs reportados por cada release desplegada
Tiempo invertido en nueva funcionalidad	Incremento en el tiempo dedicado a añadir nuevas features

Uso de herramientas análisis código locales	% desarrolladores aplicando herramientas de análisis de código locales
Número de versiones no retrocompatibles	Número de versiones desplegadas que rompen compatibilidad en aplicaciones de consumo
Aplicación versionado semántico	% artefactos en el equipo aplicando versionado semántico
Integración de código	Número de rechazos de integración por defectos en el código

Enlaces

- https://martinfowler.com/articles/is-quality-worth-cost.html
- https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas
- https://www.martinfowler.com/bliki/BeckDesignRules.html
- https://refactoring.com/
- https://semver.org/lang/es/

2.4. Escribir código seguro

La seguridad es un requisito imprescindible para todo componente de software desarrollado. Debe ser responsabilidad de todos los perfiles implicados en el desarrollo (además de los equipos especializados) aplicar una serie de criterios para evitar situaciones indeseadas (e.g., manipulación del código, exposición de datos, compartición de secretos, vulnerabilidades, etc.). Estas situaciones potencialmente habilitan a un atacante a robar información sensible, a cometer fraude económico o a causar disrupción del funcionamiento normal de los sistemas.

Beneficios

Los beneficios que trae consigo aplicar prácticas de seguridad al desarrollar código son las siguientes:

- Aplicar los cumplimientos regulatorios (e.g. DORA, PCI-DSS, etc).
- Establecer una cultura de desarrollo seguro disponiendo de un modelo de responsabilidad común en el que la seguridad es un requisito a tener en cuenta por todos. Directrices homogéneas redundan en la eficiencia del proceso del desarrollo
- Desplazar a la izquierda (shift-left) los mecanismos de seguridad, mejorando la velocidad de desarrollo de las soluciones, la calidad, el rendimiento, la exposición al riesgo y la satisfacción del cliente.
- Mejorar la colaboración entre los equipos de seguridad, los ingenieros de software y el resto de roles que están involucrados en el ciclo de vida del desarrollo del software.
- Proteger al cliente y por tanto la credibilidad del negocio.
- Prevenir vulnerabilidades de riesgo crítico y alto, especialmente en aplicaciones o módulos software catalogados como críticos. Esto ayudará a reducir el número de incidentes de seguridad y el impacto de los mismos.
- Reducir los costes y el "time to market". Cuanto más tarde se detecte un fallo de seguridad en el ciclo de vida del desarrollo, más costoso será su reparación y afectación al negocio.

Adopción

2.4.1. Escribir código seguro

Los equipos **deben** aplicar el principio **"Secure by design",** es decir, incluir requisitos de seguridad y trabajar siempre evaluando los potenciales riesgos de seguridad que sus acciones puedan generar desde las primeras fases de los proyectos e involucrando a todos los integrantes del equipo (e.g., líder técnico, negocio, perfil de seguridad).

Los fallos de diseño tienen causas y soluciones muy diferentes a los fallos de implementación. Un diseño "seguro" puede tener fallos de implementación que den lugar a vulnerabilidades explotables. Un diseño "inseguro" no puede ser arreglado en la implementación porque no se plantearon los controles contra ataques específicos.

Por ese motivo, es fundamental incorporar las mejores prácticas de seguridad como:

- Los equipos **deben** poseer el conocimiento de seguridad adecuado y adaptado al contexto en el que trabajan para asegurar el "Secure by Design" al inicio de nuevos proyectos o para añadir nueva funcionalidad no planificada en el momento de su diseño.
- Aplicar mecanismos de Threat Modeling. Esto es, recoger no sólo los casos de uso sino los "casos de abuso" y qué control va a gestionarlos (infraestructura, arquitectura/procesos de seguridad, la propia aplicación).
- Integrar y explicitar los requisitos de seguridad (funcional o técnico) específicos en los user journeys y user stories.
- Establecer y usar un conjunto de patrones de diseño seguro, componentes de seguridad disponibles y/o componentes globales o locales certificados. Ej. uso de vaults para almacenamiento/gestión de contraseñas/claves de forma externa al código, uso de SDKs de seguridad, etc).
- Generar tests unitarios y de integración enfocados en seguridad para aquellos controles críticos gestionados por la propia aplicación, tratándolos como un tipo de test más en un enfoque ATDD.
- Notificar cambios llevados a cabo sobre piezas catalogadas como críticas o previamente certificadas, para su revisión al equipo de seguridad correspondiente. Cualquier cambio no notificado sobre piezas previamente garantizadas generará un bloqueo en la subida a producción hasta que no se certifique de nuevo el cambio propuesto.

Además de las acciones enfocadas a evaluar la seguridad antes de la codificación del código, los desarrolladores **deben** prestar especial atención a **eliminar los errores de seguridad** en el entorno en el que trabajamos (sector, tecnología, lenguajes de programación, etc.). En concreto, se debe prestar especial atención a las vulnerabilidades más comunes como las que aparecen en rankings publicados en estándares de la industria como **OWASP Top10** o los publicados por Corporate Security el Banco. Los desarrolladores deben atender a las siguientes directrices:

- **Deben** cumplir las indicaciones descritas en el cuerpo normativo relacionado con la seguridad en el ciclo de vida del desarrollo del SW
- **Deben** disponer de una formación adecuada a los desarrolladores sobre los errores o vulnerabilidades más conocidas de la industria o las más recurrentes del Banco, así como entrenamiento para ser eficaces en cómo evitarlos. Esto redundará en una mejora rápida de la seguridad de los desarrollos. La formación permitirá también que los desarrolladores puedan probar y corregir desde su entorno local o primeros entornos, esto mejorará la velocidad y la calidad final de los desarrollos
- **Deben** atender a las consideraciones de seguridad dirigidas al uso de software de terceros o software open source. Actualmente, gran parte del software se construye integrando partes de terceros. Estos componentes están en el foco de los atacantes por lo que se debe garantizar la trazabilidad de versiones de dichos componentes para evitar introducir

vulnerabilidades conocidas y, en cualquier caso, verificar junto a los equipos de seguridad que el componente reutilizado no contiene vulnerabilidades críticas o altas.
- Además, **deben** establecer otros procesos asistidos o manuales de seguridad para reducir las vulnerabilidades al mínimo con herramientas de análisis estático y dinámico de código, análisis de dependencias, revisión manual (hacking ético) y bug bounties.
- En el caso en que nuestro código contenga vulnerabilidades críticas o altas, **deben** solventarlas antes de su puesta a producción. Si no existen controles que bloqueen la subida a producción los desarrolladores **deben** ser responsables de los bloqueos manuales necesarios para evitar situaciones de riesgo no deseadas.

Los desarrolladores y los equipos de seguridad se **deben** adaptar mutuamente, lo que redundará en un proceso que entregue software seguro incluso más rápidamente.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- Chimera**:** Su cometido es llevar a cabo análisis de seguridad del código (e.g., análisis de código estático, dependencias, seguridad).
- Samuel: Herramienta de enforcement que integra las evaluaciones de seguridad dentro del SDLC y notifica a los desarrolladores acerca de su cumplimiento.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Número de vulnerabilidades del código	Número medio de vulnerabilidades de seguridad detectadas por release y servicio/aplicación agrupadas por severidad
Tiempo necesario para solventar vulnerabilidades	Tiempo medio en días necesario para arreglar una vulnerabilidad desde que esta fue descubierta
% de remediación	Volumen de remediación vs stock de vulnerabilidades. Esto nos indicará el grado de implicación con la seguridad

Enlaces

- https://owasp.org/Top10/es/
- https://owasp.org/Top10/es/A04_2021-Insecure_Design/
- https://www.threatmodelingmanifesto.org/

2.5. Almacenar y evolucionar la documentación junto al código

Beneficios

Los beneficios de disponer de esta documentación son los siguientes:

- Mejorar la colaboración y el consumo por parte de terceros.
- Reducir la curva de aprendizaje de nuevas incorporaciones al equipo u otros contribuidores.
- Aumentar la velocidad de entrega.
- Reportar issues o bugs, mejorando la calidad del código.

Precondiciones

Para poder aplicar la práctica se deben dar las siguientes precondiciones:

- Haber adoptado la práctica 2.1 y por tanto tener un repositorio de control.

Adopción

2.5.1. Almacenar y evolucionar la documentación junto al código

Todos los equipos **deben** mantener documentación actualizada junto al código. La documentación a la que hacemos referencia cumple con los siguientes propósitos:

- **Desarrollo**: Documentación relacionada con el desarrollo de código (e.g., README, comentarios) que reside y evoluciona junto al código.
- **Consumo**: Documentación visible a otros equipos o empleados donde se describe cómo se consume el componente desarrollado (e.g., swagger, API documental). Este tipo de documentación reside y evoluciona junto al código y **debe** generarse de forma automática.
- **Contribución**: Documentación donde se describen directrices para colaborar entre los distintos equipos en los componentes de software (e.g., CONTRIBUTING).
- **Reportar issues**: Documentación donde se describen las pautas para reportar problemas asociados a un componente de software.

Esta práctica requiere la adopción de las siguientes directrices:

- El código **debe** ser autoexplicativo, en caso de que el código sea demasiado complejo se pueden añadir comentarios documentando esa parte del código.
- Todos los repositorios **deben** contener un fichero README escrito en formato Markdown que orienta a los desarrolladores en el código (e..g, directorios, scripts útiles, orquestación, despliegue) y sirve de punto de entrada para enlazar con documentación más extensa y guías de usuario externas.
    - Los equipos **pueden** usar las sugerencias definidas aquí para crear el README de sus repositorios.
    - El README **debe** contener una descripción clara del objetivo técnico del código almacenado en el repositorio y detallar un propietario (equipo).

- Los equipos **deberían** tener un fichero CHANGELOG en formato Markdown.
    - Los equipos **pueden** usar el CHANGELOG definido aquí como ejemplo.
    - La generación del changelog **debería** estar automatizada por parte de los equipos.
- La documentación que vaya a ser consumida por otros equipos **debería** publicarse (e.g., documentación API, documentación uso librerías).
    - Se **deben** seguir formatos estándar para la documentación publicada (e.g., OpenAPI, RAML) que sean fácilmente exportables y legibles.
- Los equipos **deberían** hacer uso de automatismos para publicar y actualizar su documentación en cada release (e.g., actualizar y publicar el changelog, actualizar la documentación de su API).

Además, **debe** existir un servicio único de documentación global en el Banco donde los proyectos puedan guardar su documentación no técnica y relacionarla con los repositorios de código.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket:** Los ficheros readme, changelog y contribution estarán alojados en los repositorios de cada proyecto en Bitbucket.
- **Servicio documental único:** Herramienta pendiente de definición.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
% de repositorios con la documentación requerida	Repositorios que contienen fichero readme y changelog
% servicios consumibles con documentación publicada	Porcentaje de APIs consumidas por terceros que que tienen documentación publicada
% documentación generada automáticamente	Porcentaje de documentos generados automáticamente que deban serlo (changelog)
% repositorios con documentación clara	Porcentaje de repositorios con documentación clara, incluyendo readme y changelog

Enlaces

- https://google.github.io/styleguide/docguide/best_practices.html (último capítulo)

2.6. Adoptar mecanismos de revisión de código

Beneficios

Los beneficios que aporta aplicar este tipo de mecanismos son los siguientes:

- Mejorar la calidad del código, reduciendo los bugs en producción.
- Procurar que las buenas prácticas y convenciones establecidas se cumplan.
- Fomentar la colaboración entre desarrolladores.
- Incrementar la legibilidad del código.
- Compartir el conocimiento y la experiencia del equipo y equilibrarlo.

Precondiciones

Para poder aplicar la práctica se deben dar las siguientes precondiciones:

- Haber adoptado la práctica 2.1 y por tanto tener un repositorio de control.
- Debe haber al menos dos integrantes en el equipo.

Adopción

Todos los equipos **deben** adoptar mecanismos de revisión de código. A continuación se describen algunos de ellos.

2.6.1. Pull Requests

Los equipos **deberían** aplicar mecanismos de **pull request** antes de fusionar su código en la rama principal.

- Sólo los equipos que lleven a cabo un modelo de integración continua aplicando trunk based development están exentos de llevar a cabo Pull Requests.
- Las Pull Requests no **deben** suponer un obstáculo para el desarrollo, **deben** ser constructivas y deben usarse únicamente para la revisión y validación del código por parte del equipo dueño del repositorio.
- Los equipos **pueden** decidir su procedimiento de Pull Requests siempre que cumpla con los requisitos establecidos en esta sección.

2.6.2. Pair programming

Los equipos **pueden** aplicar mecanismos de **programación por parejas (pair programming)** antes de fusionar su código en la rama principal. El procedimiento escogido por los equipos para llevar a cabo programación por parejas **debe** estar claramente definido para todos los miembros del equipo (con escoger uno de los estilos propuestos en On Pair Programming sería suficiente).

2.6.3. Code review

Los equipos **pueden** llevar a cabo **sesiones de code review** en situaciones en las que se necesite refinar o detallar cierta funcionalidad del código que requiera del conocimiento completo del equipo. Los objetivos de esta sesión **deben** estar claramente identificados y marcar tiempo límite para las sesiones.

2.6.4. Mob programming

Los equipos **pueden** llevar a cabo sesiones de **Mob programming** en situaciones en las que se requiera, pero no de forma habitual.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket:** Las Pull Requests se **deben** llevar a cabo en esta herramienta.
- **IDE:** Para sesiones de Pair programming los desarrolladores **pueden** usar el IDE para compartir y editar código en tiempo real.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Tiempo medio de aprobación de Pull Requests	Tiempo medio en días desde que una Pull Request se abre y es aceptada / rechazada
Tiempo actividad de las Pull Requests	Tiempo medio en días desde que una PR se abre y se itera sobre ella (comentarios, cambios en código, aceptación, rechazo)

Enlaces

- https://martinfowler.com/bliki/PullRequest.html
- https://martinfowler.com/articles/on-pair-programming.html

2.7. Instaurar un modelo InnerSource

Los equipos **deberían** seguir un modelo **InnerSource** para los activos de software que lo permitan.

Beneficios

Este modelo trae consigo múltiples beneficios:

- Fomentar la colaboración entre equipos alrededor del software.
- Mejorar la calidad del código acelerando la detección de bugs y vulnerabilidades.
- Incrementar la velocidad de desarrollo al permitir a otros equipos contribuir.
- Mejorar la comunicación entre los equipos permitiendo la petición de funcionalidad nueva o issues en los componentes.
- Aumentar la autonomía de los equipos.
- Atraer talento.

Adopción

2.7.1. Instaurar un modelo InnerSource

Las siguientes directrices se deben cumplir para poder instaurar un modelo InnerSource en el Banco:

- Los repositorios **deben** contener un fichero CONTRIBUTING en formato Markdown.
- Los equipos **deben** usar un modelo de ramas para contribuir a otros repositorios tal y como se establece en su README.
- El modelo de branching strategy **debe** estar claramente definido como se mencionó en la práctica 2.2 para facilitar la contribución de otros equipos.
- **Debe** existir un procedimiento documentado para abrir issues en los repositorios por parte de otros equipos y ese procedimiento debe ser accesible.
- Se **debe** aplicar un modelo de Pull Requests para aceptar código en los repositorios acogidos al modelo InnerSource y la Pull Request debe reflejar el identificador del "issue" asociado. El equipo dueño del repositorio receptor siempre será el encargado de aceptar la Pull Request.
- Se **debe** establecer un modelo de propiedad intelectual global en el Banco para los activos de código.
- Los equipos **deben** disponer de tiempo para poder contribuir a otros proyectos del Banco.
- Se **debe** recompensar a los equipos que:
    - Abran issues en otros repositorios y sean aceptadas.
    - Acepten issues de otros equipos en sus repositorios
    - Resuelvan issues de otros repositorios mediante Pull Requests que sean aceptadas.
    - Acepten Pull Requests de otros equipos en sus repositorios.
- Se **debe** informar y formar a los equipos respecto a la existencia y proceso de contribución del modelo InnerSource.

- Se **debe** exponer un modelo de apertura directa de issues en los repos que se acojan al modelo InnerSource.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Bitbucket:** Los equipos que quieran adoptar el modelo InnerSource deben hacerlo mediante bitbucket adoptando una buena política de permisos y de manejo de issues y pull requests.
- **Jira:** Los issues deberían ser tratados en esta herramienta.
- **Servicio documental único:** Herramienta pendiente de definición.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
% Repositorios adaptados al modelo InnerSource	% de repositorios que el equipo tiene adaptados al modelo InnerSource (i.e. repositorios accesibles, con fichero contribution mencionando el tipo de modelo aplicado)
Número de Pull Requests aceptadas	Número de Pull request que han sido aceptadas por otro equipo distinto al receptor
Número de issues aceptadas	Número de issues que han sido aceptadas por otro equipo distinto al receptor

Enlaces

- https://en.wikipedia.org/wiki/Inner_source
- https://en.wikipedia.org/wiki/Contributing_guidelines
- https://es.wikipedia.org/wiki/Markdown
- https://innersourcecommons.org/learn/learning-path/introduction/03/

3. Proceso

Las prácticas revisadas anteriormente se operativizan a través del **proceso de desarrollo y versionado de código**. Se define el proceso de desarrollo y versionado de código como una parte integral del ciclo de vida del software cuyo objetivo es crear, evolucionar, integrar y almacenar el código fuente de los distintos equipos de software del Banco.

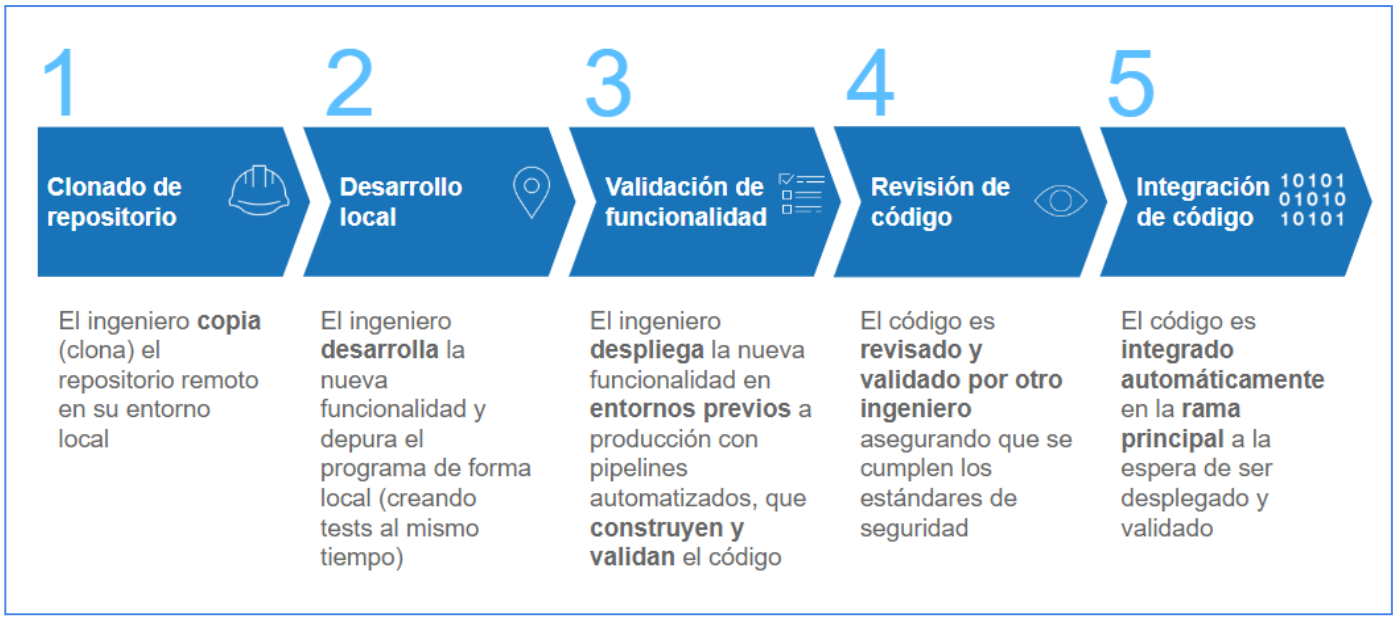

Figura 2: Proceso de desarrollo y versionado de código

1. Clonado de repositorio

El **ingeniero de software** copia de forma local los distintos repositorios de código asociados al servicio que son afectados por la nueva funcionalidad. El **tech lead** debe asegurar que todos los integrantes del equipo tengan los accesos necesarios y que los repositorios siguen los estándares establecidos en el playbook (e.g., metadatos, documentación, modelo gestión de ramas) y dar soporte en el proceso de clonado y entendimiento de los distintos módulos del aplicativo.

2. Desarrollo local

El **ingeniero de software** desarrolla de forma local la nueva funcionalidad sobre el código (creando una rama nueva dependiendo del modelo de trabajo). En esta fase se puede apoyar en el **tech lead** u otros compañeros con más experiencia para construir código de calidad que cumpla con los requisitos esperados. En esta fase el ingeniero de software desarrollará también tests (visitar el playbook de testing para más información) para cerciorarse de que el código cumple su cometido.

3. Validación de funcionalidad

En línea con las prácticas del playbook, todo incremento en el código debe ser validado. Para ello, el **ingeniero de software**, una vez revisado su cambio en su entorno local, debería desplegar el código fuente en entornos integrados y garantizar que los requisitos de la nueva funcionalidad se cubren (e.g., mejorar el código actual, nueva funcionalidad, resolver errores) junto al resto de cambios.

4. Revisión de código

Antes de dar la funcionalidad por completada, el **ingeniero de software** valida el código desarrollado apoyándose en mecanismos de revisión de código. Independientemente del mecanismo escogido, los equipos deberían aplicar un paso de revisión de código, especialmente en componentes que sean considerados críticos (e.g., transacciones, sistemas accedidos por los usuarios finales).

5. Integración de código

Una vez el código ha sido validado y revisado este se construye (visitar playbook de construcción y análisis de código) y valida mediante diferentes tests (visitar el playbook de testing para más información). Finalmente **se integra de forma automática** en la rama principal.

GLOSARIO

Equipo: Conjunto de personas trabajando sobre un mismo backlog de tareas asociado a un producto de software.

Servicio: Productos de software (e.g., aplicación web, aplicación móvil, librerías de transacciones).

REFERENCIAS

- Beck, K. (2003). *Test Driven Development*. Addison-Wesley.
- Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- Hunt, A., Thomas, D., & Cunningham, W. (2000). *The Pragmatic Programmer*. Addison-Wesley.
- Kim, G., Forsgren, N., & Humble, J. (2018). *Accelerate: The Science Behind DevOps : Building and Scaling High Performing Technology Organizations*. IT Revolution.
- Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship* (R. C. Martin, Ed.). Prentice Hall.
- *Maxim, B. R., & Pressman, R. S. (2014). Software Engineering: A Practitioner's Approach. McGraw-Hill Education.*
- *Raymond, E. S. (2004). The art of UNIX programming. Addison-Wesley.*
- *Bird, J., Bell, L., Smith, R., & Brunton-Spall, M. (2017). Agile Application Security: Enabling Security in a Continuous Delivery Pipeline. O'Reilly Media.*