UNIDAD 4. OPTIMIZACIÓN Y DOCUMENTACIÓN

Profesor: Bartolomé Jesús Choza Medina

1.REFACTORIZACIÓN

El objetivo de la fase de pruebas es evitar errores en el programa, detectándolos y eliminándolos antes de que comience a usarse.

La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro.

1.1 RAZONES

- **Eficiencia**: El esfuerzo que invirtamos en evitar la duplicación de código y en simplificar el diseño se verá recompensado cuando tengamos que realizar modificaciones, tanto para corregir errores como para añadir nuevas funcionalidades.
- **Diseño Evolutivo**: En muchas ocasiones los requisitos al principio del proyecto no están suficientemente especificados y debemos abordar el diseño de una forma gradual.
- **Evitar la Reescritura de código**: No es fácil enfrentarse a un código que no conocemos y que no sigue los estándares que uno utiliza, pero eso no es una buena excusa para empezar de cero.

1.2 FACTORES DE REFACTORIZACIÓN

Sólo debemos refactorizar cuando identifiquemos código mal estructurado o diseños que supongan un riesgo para la futura evolución de nuestro sistema.

Los síntomas que nos avisan de que nuestro código tiene problemas se conocen como "**Bad Smells**".

1.3 BAD SMELLS

- **Código duplicado**.
- **Método largo**: En la programación orientada a objetos cuando mas corto es un método más fácil es reutilizarlo.
- **Clase grande**: Si una clase intenta resolver muchos problemas suele conducir a código duplicado.
- **Lista de parámetros extensa**: En la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino sólo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario.
- **Cambio divergente**: Cuando una clase es frecuentemente modificada por diversos motivos.

1.3 BAD SMELLS

- **Shotgun surgery**: Cuando después de un cambio en un determinado lugar, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.
- **Envidia de funcionalidad**: Cuando un método utiliza más cantidad de elementos de otra clase que de la propia.
- **Clase de datos**: Clases que sólo tienen atributos y métodos de acceso a ellos ("get" y "set").
- **Legado rechazado**: Subclases que usan sólo pocas características de sus superclases.

1.4 REFACTORIZACIÓN Y PRUEBAS

Una vez identificado el "Bad Smell" deberemos aplicar una refactorización que permita corregir ese problema. Para comenzar a Refactorizar es imprescindible que el proyecto tenga pruebas automáticas que nos permitan saber en cualquier momento al ejecutarlas si el desarrollo sigue cumpliendo los requisitos que implementaba.

2. CONTROL DE VERSIONES

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación.

Los sistemas de control de versiones facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas.

Un Sistema de Control de versiones es una herramienta software que, de manera automática, se encarga de facilitar la gestión de las versiones del código de un proyecto de manera centralizada.

2.1 CARACTERÍSTICAS

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación…).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

2.1 CARACTERÍSTICAS

- Mantener control sobre los cambios que se realizan: quién, cuándo, qué, etc.
- Decidir la forma definitiva de un archivo cuando los cambios realizados por dos personas a la vez son incompatibles.
- Si nos damos cuenta que los últimos cambios realizados no siguen el camino apropiado, podamos volver atrás.

2.2 TIPOS

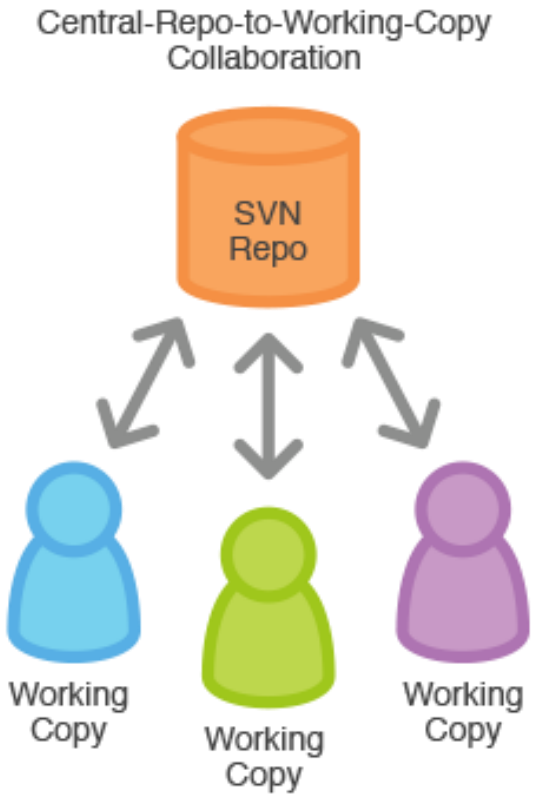
- **Local**: Permite gestionar un proyecto de forma local.
- **Centralizados**: Es un tipo de control de versiones que toma un enfoque cliente-servidor. Existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Ejemplos: CVS y Subversion.
- **Distribuidos**: Es un tipo de control de versiones que toma un enfoque peer-to-peer. Cada usuario tiene su propio repositorio. No es necesario tomar decisiones centralizadamente. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Ejemplos: Git y Mercurial.

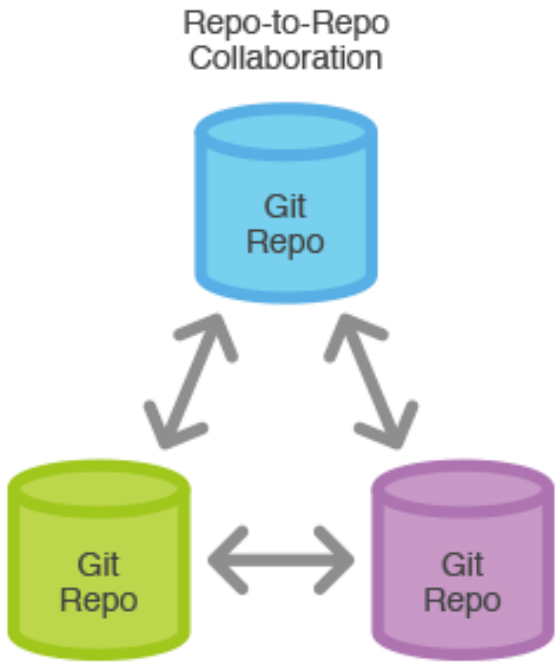
2.2 TIPOS

Diferencias de los Distribuidos frente a los Centralizados:

- En los Distribuidos no existe una copia de referencia del código, solo copias de trabajo.
- Las operaciones más comunes son regularmente más rápidas, ya que no tiene que comunicarse con un servidor central.
- Cada copia de trabajo es un tipo de respaldo del código base.
- Los desarrolladores pueden trabajar sin la necesidad de estar conectados a un servidor, incluso a Internet.

2.2 TIPOS


Central-Repo-to-Working-Copy
Collaboration
SVN
Repo
Working
Copy
Working
Copy
Working
Copy


Repo-to-Repo
Collaboration
Git
Repo
Git
Repo
Git
Repo

2.3 REPOSITORIOS

Un repositorio es el conjunto de información gestionada por el sistema. Este repositorio contiene el historial de versiones de todos los elementos gestionados.

En el caso de sistemas de control de versiones centralizados, existe un único repositorio maestro, el cual almacena todos los cambios en el proyecto. En el caso de sistemas de control de versiones descentralizados, cada desarrollador tiene su propio repositorio y los cambios pueden ser intercambiados entre repositorios arbitrariamente.

3 GIT

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

3 GIT

Git es una herramienta de repositorio local y remoto con GitHub u otro proveedor. Vamos a instalar dos repositorios que vamos a sincronizar. Uno está en local, y el otro en la nube GitHub.

Esta herramienta se usa para versionado. Es decir, sólo cuando se introducen cambios funcionales o de código en un fichero o ficheros se sube la nueva versión. No se confirman los cambios en pasos intermedios. Cuando se termina completamente un cambio en desarrollo, se confirman los cambios, y se comenta la versión.

3.1 GITHUB

Github es un proveedor de servicios de hosting para la herramienta de control de versionado GIT. Ofrece funcionalidades de control de versiones y gestión de código fuente, así como características propias.

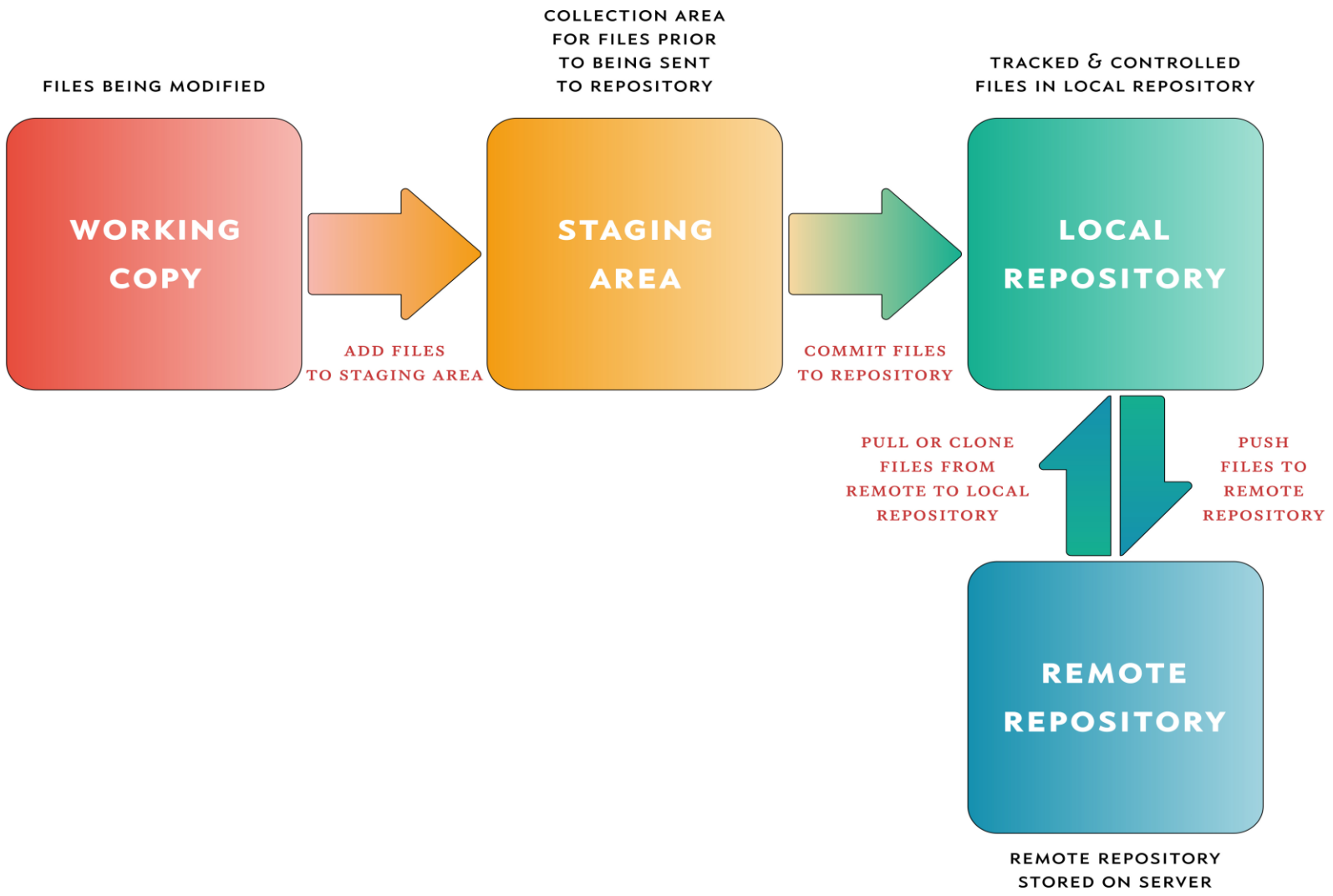
Proporciona control de acceso y un conjunto de características de trabajo colaborativo tal como, control de bugs, monitoreo de peticiones remotas, administración de tareas y wikis de documentación para cada Proyecto. Además ofrece herramientas de CI/CD (continua integración ,continuo despliegue).

En Julio de 2018 Microsoft compra GitHub.

3.2 TERMINOLOGÍA / COMANDOS

- **init**: Este comando se utiliza para inicializar un nuevo repositorio de Git en un directorio vacío o existente. Una vez que se ha inicializado un repositorio, Git rastreará todos los cambios realizados en los archivos del directorio.
- **add**: Este comando se utiliza para agregar archivos al área de preparación de Git (también conocida como "staging area"). Los archivos agregados a esta área serán incluidos en el próximo commit.
- **clone**: Este comando se utiliza para crear una copia de un repositorio remoto en un nuevo directorio local. Esto es útil cuando se desea trabajar en un proyecto existente o colaborar con otros desarrolladores.

3.2 TERMINOLOGÍA / COMANDOS



3.2 TERMINOLOGÍA / COMANDOS

- **Branch (Rama)**: Lista, crea o elimina ramas (branches) en el repositorio. Las ramas son útiles para mantener diferentes líneas de desarrollo separadas y para permitir que varios desarrolladores trabajen en diferentes características del proyecto al mismo tiempo.
- **Trunk (Rama principal o Master)**: representa la línea de código donde se integra y mantiene la versión más reciente y estable del software.
- **Commit**: es un punto de guardado o una instantánea de tu repositorio en un momento específico de su historial. Cada versión de commit es etiquetada.
- **Log**: Muestra el historial de commits realizados en el repositorio.

3.2 TERMINOLOGÍA / COMANDOS

- **Head**: es un puntero que indica la instantánea o commit que se está utilizando o viendo actualmente en tu repositorio de trabajo. Es una referencia que Git utiliza para saber dónde estás en el historial de tu proyecto en un momento dado.
- **Status:** obtiene información sobre el estado actual del repositorio. Muestra información de los archivos que han sido modificados o eliminados desde el último commit, los archivos que han sido agregados al área de preparación, y cualquier archivo que aún no haya sido rastreado por Git. Esto es útil para saber qué cambios han sido realizados y qué archivos están listos para ser incluidos en el próximo commit.

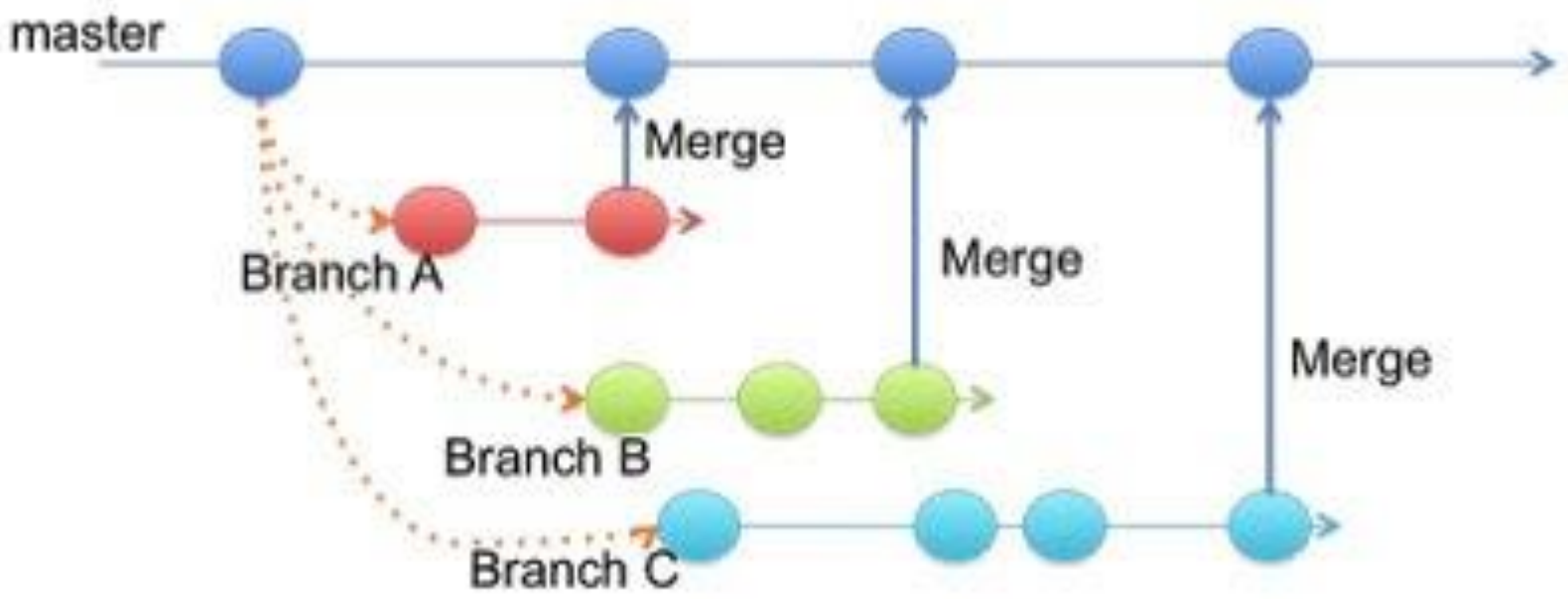
3.2 TERMINOLOGÍA / COMANDOS

- **Pull**: Se descarga la ultima versión del repositorio remoto al local.
- **Push**: Se utiliza para enviar los commits que has realizado en tu repositorio local a un repositorio remoto.
- **Fetch**: Descarga los cambios del repositorio remoto sin fusionarlos con la rama local actual. A diferencia de git pull, que descarga y fusiona los cambios del repositorio remoto en la rama local actual, git fetch descarga los cambios del repositorio remoto y los almacena en una rama local llamada "remotes/origin/<rama>" (donde "origin" es el nombre por defecto del repositorio remoto y "<rama>" es la rama remota correspondiente).
- **Diff o change**: Permite ver/comparar los cambios entre versiones.

3.2 TERMINOLOGÍA / COMANDOS

- **Checkout**: se utiliza principalmente para cambiar el puntero HEAD y actualizar el árbol de trabajo al estado de un commit o rama específica.
- **Sync**: Combina cambios de repositorio local y remoto.
- **Merge (o fusión)**: Es el proceso de tomar la historia de cambios de una rama de desarrollo y aplicarla a otra rama, fusionando así los cambios y el historial de ambas ramas.
- **Rebase**: Permite borrar o reescribir todos los commits antiguos. Es muy útil cuando creamos una rama nueva, ya que integra los cambios de una rama en otra, como si hiciera una copia de la rama.

3.2 TERMINOLOGÍA / COMANDOS



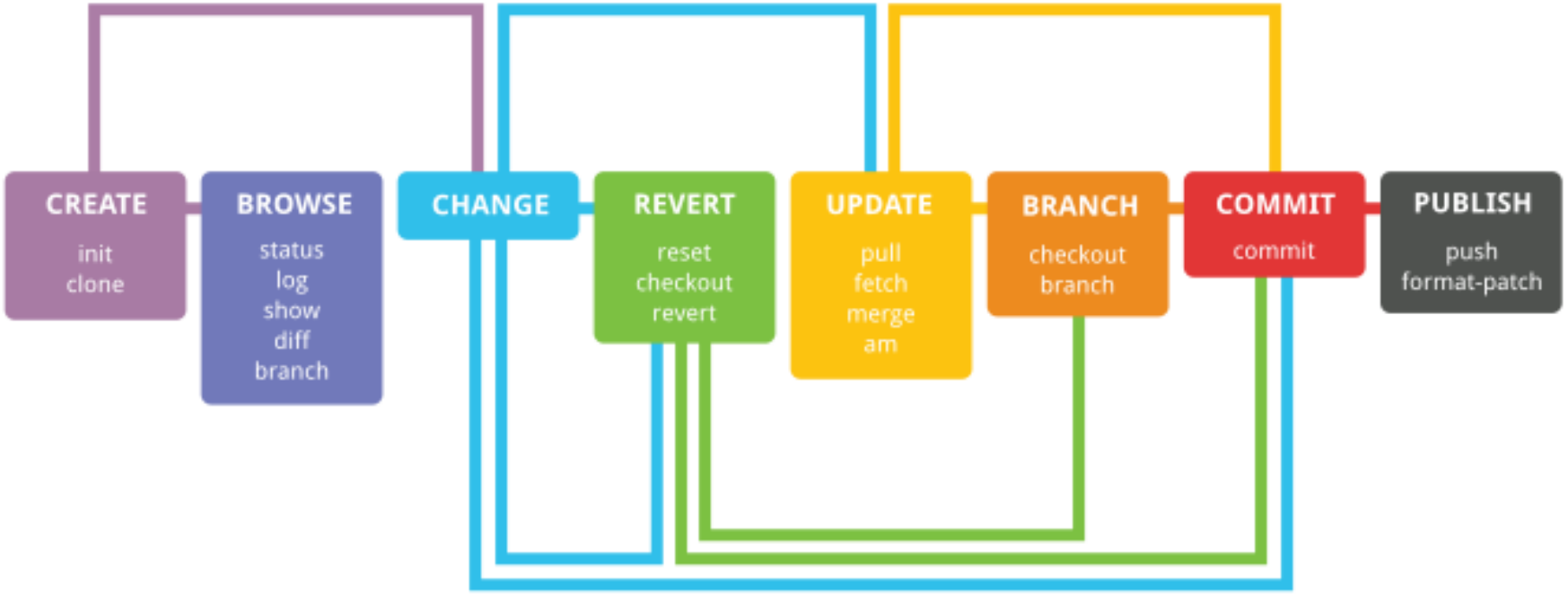
3.2 TERMINOLOGÍA / COMANDOS

- **Reset**: Se utiliza para mover el puntero de la rama actual (HEAD) a un commit anterior, reescribiendo así la historia de los commits de esa rama a partir de ese punto.
- **Revert**: deshace un commit existente en el repositorio de manera segura, creando un nuevo commit que revierte los cambios del commit anterior. A diferencia de git reset, que elimina el commit y todos los cambios realizados después de ese commit, git revert mantiene el historial de cambios en el repositorio y permite a los desarrolladores deshacer cambios específicos sin afectar el resto del historial de cambios.

3.2 TERMINOLOGÍA / COMANDOS

- **Import**: Toma un directorio de archivos que aún no está bajo control de versiones y cargarlo por primera vez en el repositorio remoto.
- **Stash**: Guarda trabajo sin confirmar temporalmente para poder cambiar de contexto o rama.

3.2 TERMINOLOGÍA / COMANDOS



3.2 TERMINOLOGÍA / COMANDOS

Git Cheat Sheet

REBELLABS by ZEROTURNAROUND

Create a Repository

From scratch -- Create a new local repository
```
$ git init [project name]
```

Download from an existing repository
```
$ git clone my_url
```

Observe your Repository

List new or modified files not yet committed
```
$ git status
```

Show the changes to files not yet staged
```
$ git diff
```

Show the changes to staged files
```
$ git diff --cached
```

Show all staged and unstaged file changes
```
$ git diff HEAD
```

Show the changes between two commit ids
```
$ git diff commit1 commit2
```

List the change dates and authors for a file
```
$ git blame [file]
```

Show the file changes for a commit id and/or file
```
$ git show [commit]:[file]
```

Show full change history
```
$ git log
```

Show change history for file/directory including diffs
```
$ git log -p [file/directory]
```

Working with Branches

List all local branches
```
$ git branch
```

List all branches, local and remote
```
$ git branch -av
```

Switch to a branch, my_branch, and update working directory
```
$ git checkout my_branch
```

Create a new branch called new_branch
```
$ git branch new_branch
```

Delete the branch called my_branch
```
$ git branch -d my_branch
```

Merge branch_a into branch_b
```
$ git checkout branch_b
$ git merge branch_a
```

Tag the current commit
```
$ git tag my_tag
```

Make a change

Stages the file, ready for commit
```
$ git add [file]
```

Stage all changed files, ready for commit
```
$ git add .
```

Commit all staged files to versioned history
```
$ git commit -m "commit message"
```

Commit all your tracked files to versioned history
```
$git commit -am "commit message"
```

Unstages file, keeping the file changes
```
$ git reset [file]
```

Revert everything to the last commit
```
$ git reset --hard
```

Synchronize

Get the latest changes from origin (no merge)
```
$ git fetch
```

Fetch the latest changes from origin and merge
```
$ git pull
```

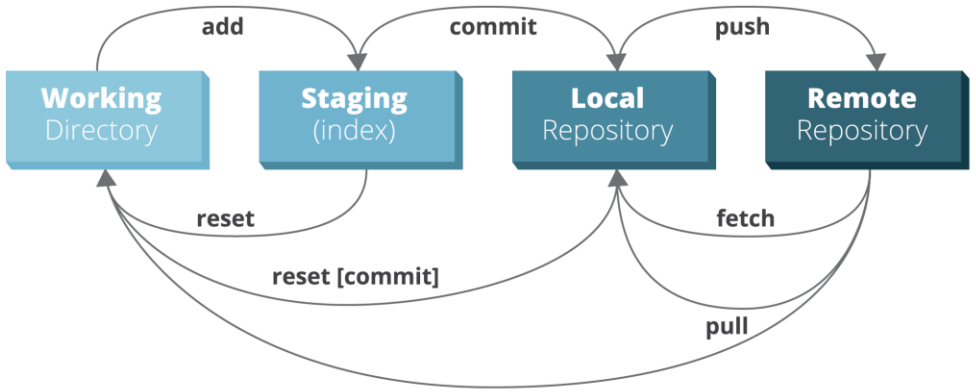
Fetch the latest changes from origin and rebase
```
$ git pull --rebase
```

Push local changes to the origin
```
$ git push
```

Finally!

When in doubt, use git help
```
$ git command --help
```

Or visit https://training.github.com/ for official GitHub training.

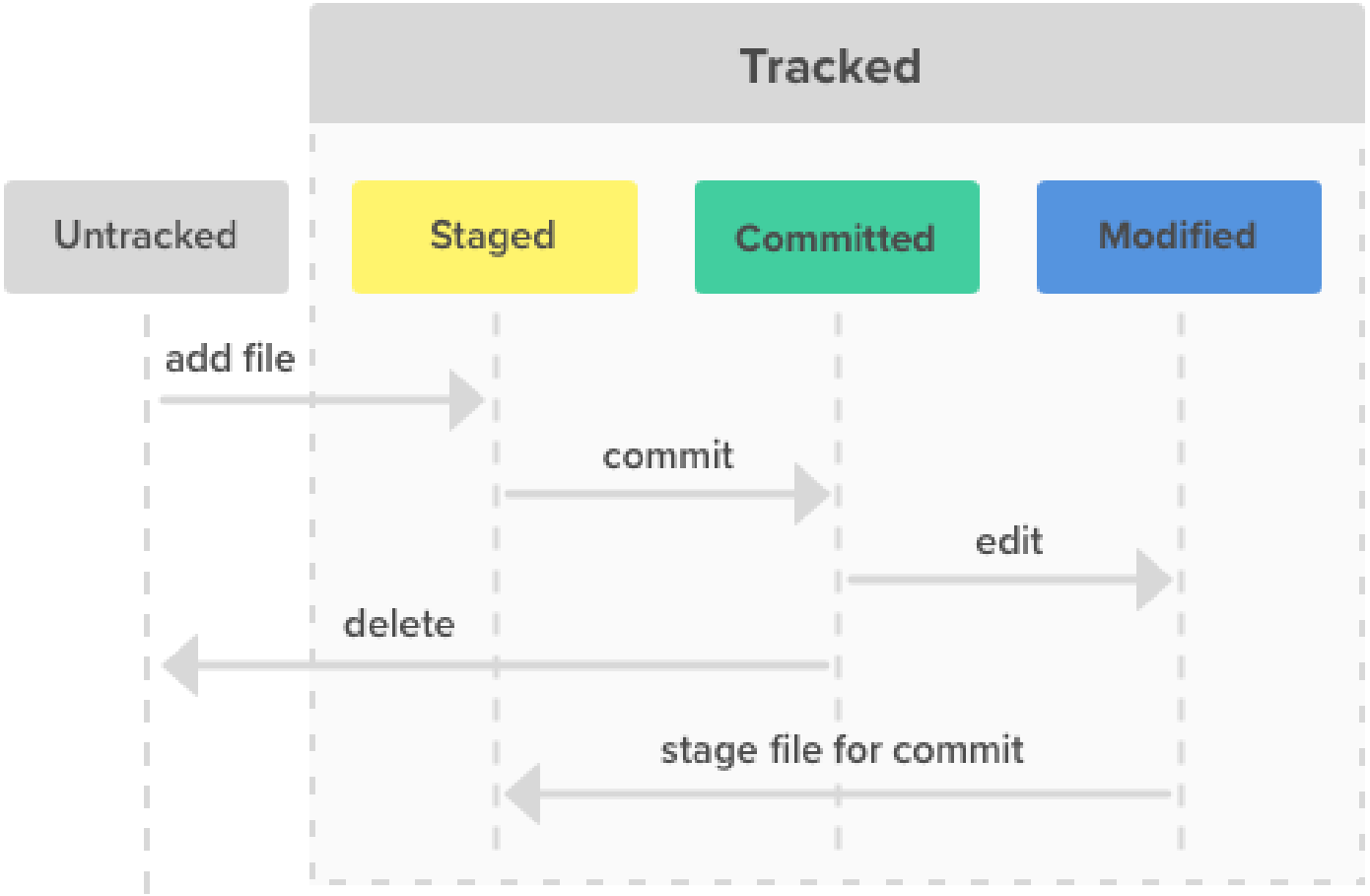


3.3 FLUJO DE TRABAJO

En Git un archivo puede estar en uno de tres estados:

- **Confirmado (commited)**: indica que los datos están almacenados de manera segura en tu base de datos local.
- **Modificado (modified)**: indica que has modificado el archivo, pero todavía no lo has confirmado.
- **Preparado (staged)**: significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

3.3 FLUJO DE TRABAJO

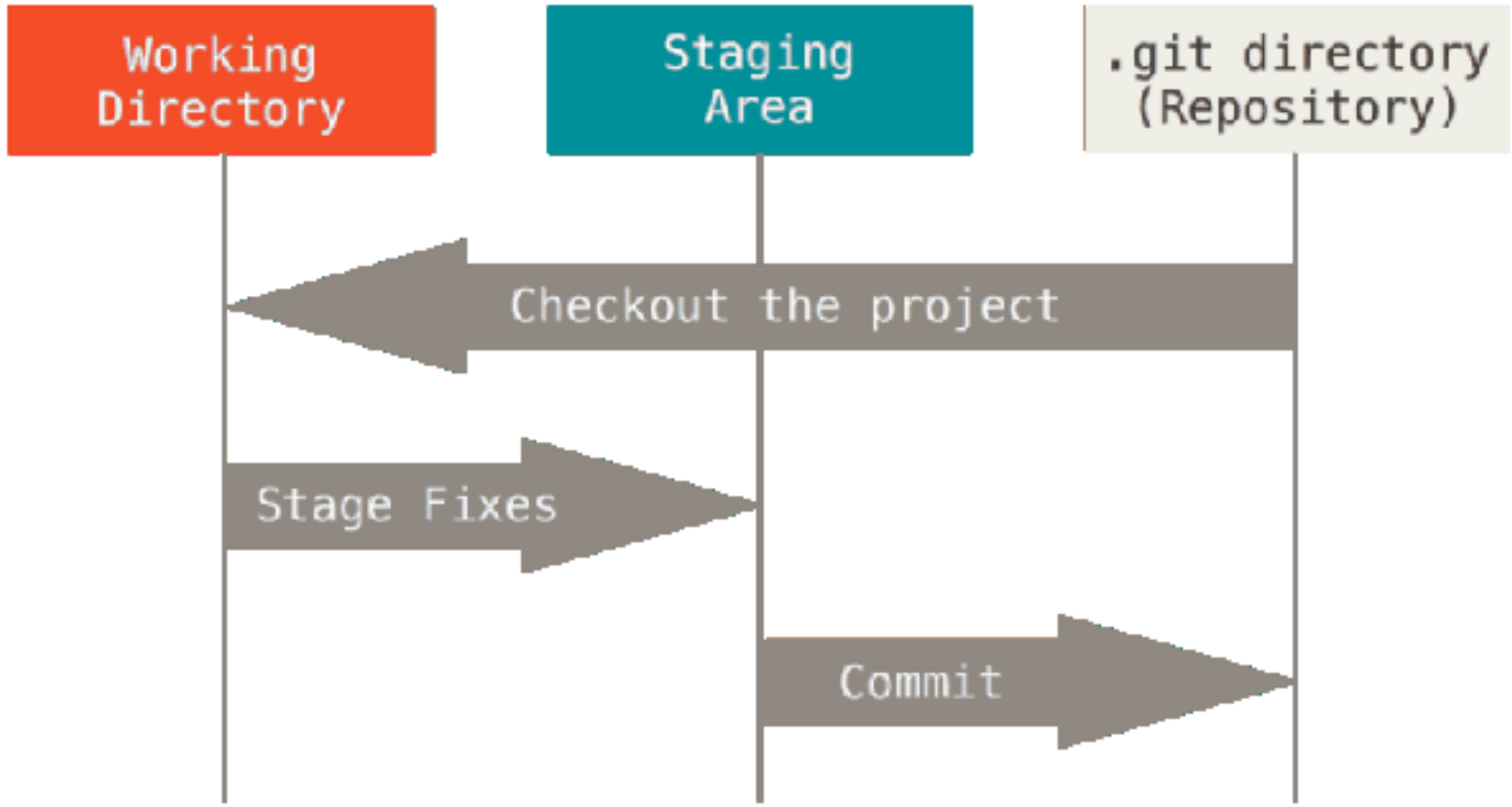


3.3 FLUJO DE TRABAJO

El flujo de trabajo en Git para una confirmación sería:

- **Modificar** una serie de archivos en el directorio de trabajo.
- **Añadir** los archivos modificados al área de preparación.
- **Confirmar** los cambios. Esto toma los archivos tal y como están en tu área de preparación y los almacena en la base de datos del directorio .git.

3.3 FLUJO DE TRABAJO



3.4 SECCIONES DE GIT

En un proyecto de Git existan las siguientes secciones principales:

- **Directorio .git**: es donde se almacena la base de datos con toda la información que Git necesita para gestionar el proyecto.
- **Directorio de trabajo**: no es más que una copia de una versión del proyecto. Estos archivos son extraídos de la base de datos del directorio .git y se colocan en la carpeta local del proyecto para que puedas trabajar con ellos.
- **Área de preparación (Staging area):** en sí es un fichero que indica que los archivos que irán en la siguiente confirmación.

3.4 SECCIONES DE GIT

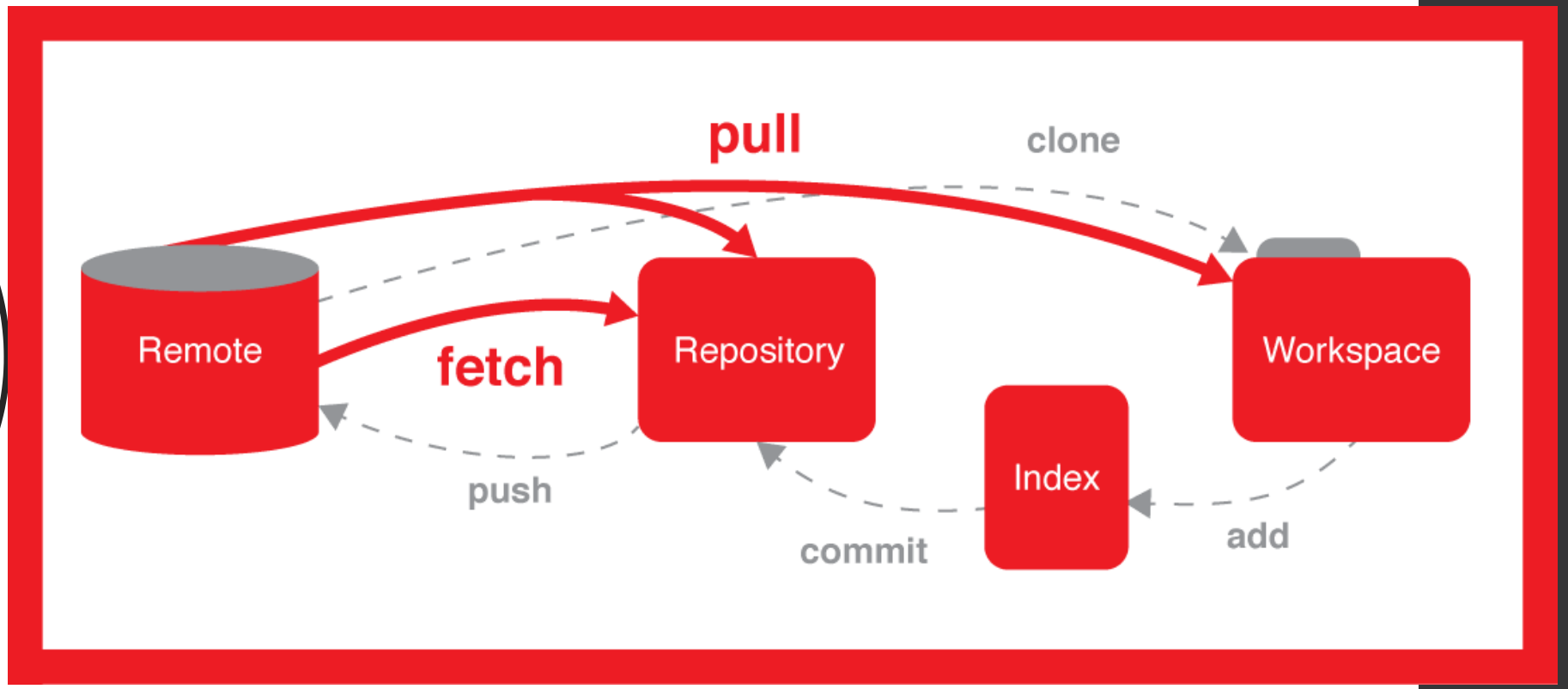
Sección	Ejemplo de comando	Qué hace
Directorio de trabajo	`git status`	Muestra qué ha cambiado
Área de preparación	`git add archivo.java`	Añade un archivo al área de preparación
Repositorio `.git`	`git commit -m "mensaje"`	Guarda los cambios en el historial

3.4 SECCIONES DE GIT

Git también puede estar conectado a un **repositorio remoto** que nos permite guardar en la nube nuestro **repositorio local**, y que más gente colabore.

3.4 SECCIONES DE GIT

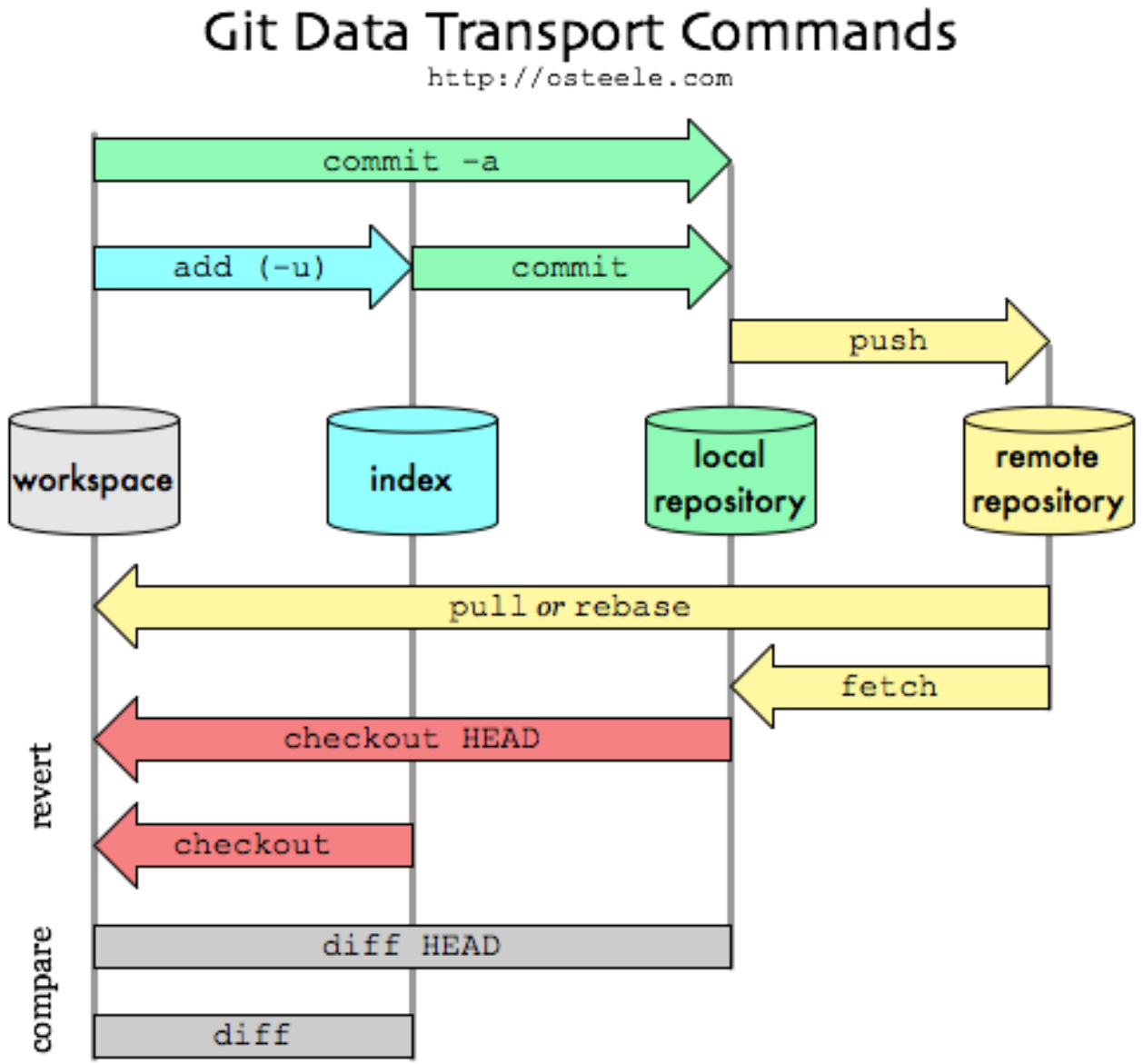
Areas de trabajo Git



3.4 SECCIONES DE GIT

Nivel	Elemento	Descripción	Ejemplo de comando
Interno (dentro del repo local)	**Directorio de trabajo**	Donde editas los archivos físicamente	—
	Área de preparación (staging area)	Zona intermedia antes del commit	`git add`
	Repositorio local (.git)	Base de datos con los commits y ramas	`git commit`
Externo (fuera del PC)	**Repositorio remoto**	Copia alojada en GitHub/GitLab	`git push` / `git pull`

3.5 COMANDOS DE TRANSFERENCIA DE DATOS

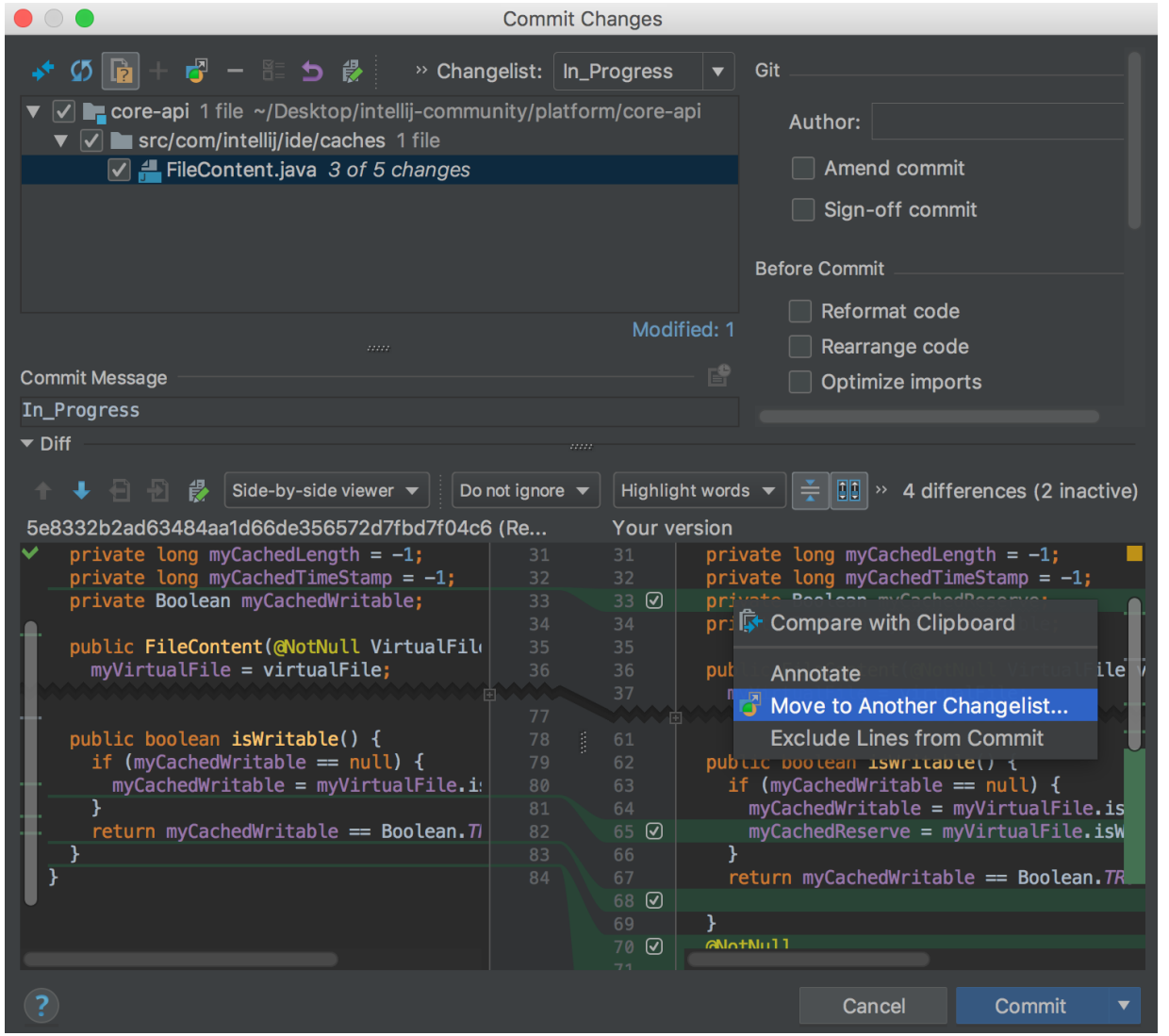


3.6. GITHUB

GitHub es una plataforma en línea donde los programadores pueden guardar, compartir y trabajar juntos en proyectos de programación. Está basada en Git y por tanto permite controlar versiones además de facilitar el trabajo en grupo o la colaboración abierta (código abierto).

3.7. USO DE GIT EN INTELLIJ

IntelliJ IDEA tiene una integración nativa y muy completa con Git. Esto significa que no necesitas instalar un plugin o herramienta externa para empezar a usarlo; la funcionalidad viene incorporada de serie.



4. DOCUMENTACIÓN

La documentación de un programa empieza a la vez que la construcción del mismo y finaliza justo antes de la entrega del programa o aplicación al cliente. Así mismo, la documentación que se entrega al cliente tendrá que coincidir con la versión final de los programas que componen la aplicación.

Una vez concluido el programa, los documentos que se deben entregar son una **manual técnico**, un **manual de usuario** y un **manual de instalación**.

4.1 MANUAL TÉCNICO

Refleja el diseño del proyecto, la codificación de la aplicación y las pruebas realizadas para su correcto funcionamiento.

El principal objetivo es el de facilitar el desarrollo, corrección y futuro mantenimiento de la aplicación de una forma rápida y fácil.

Esta guía esta compuesta por tres apartados claramente diferenciados:

- **Estudio previo.**
- **Código fuente.**
- **Pruebas.**

4.1.1 ESTUDIO PREVIO

Es donde queda reflejada la solución o diseño de la aplicación. Esta parte de la guía es únicamente destinada a los programadores. Debe estar realizado de tal forma que permita la división del trabajo.

En la ejecución de un proyecto informático o un programa software se deben de seguir una serie de pasos desde que se plantea el problema hasta que se dispone del programa o del a aplicación funcionando en el ordenador.

4.1.1 ESTUDIO PREVIO

Los pasos son los siguientes, correspondientes al ciclo de vida del software:

- Análisis de factibilidad.
- Análisis de requerimientos.
- Diseño del sistema.
- Implementación.
- Validación y pruebas.
- Explotación.
- Mantenimiento.

Cada uno de estos pasos debe llevar asociado un documento.

4.1.2 CÓDIGO FUENTE

Es donde se incluye la codificación realizada por los programadores. Este documento puede tener, a su vez, otra documentación para su mejor comprensión y puede ser de gran ayuda para el mantenimiento o desarrollo mejorado de la aplicación.

Este documento debe tener una gran claridad en su escritura para su fácil comprensión.

4.1.2 CÓDIGO FUENTE

Hay dos reglas que no se deben olvidar nunca:

- Todos los programas tienen errores y descubrirlos sólo es cuestión de tiempo y de que el programa tenga éxito y se utilice frecuentemente.
- Todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito.

Pensando en esta revisión de código es por lo que es importante que el programa se entienda: para poder repararlo y modificarlo.

4.1.2.1 ¿QUÉ DOCUMENTAMOS?

1. Todo lo que no es evidente.

2. No hay que repetir lo que se hace, sino explicar por qué se hace.

4.1.2.1 ¿QUÉ DOCUMENTAMOS?

Todo esto conlleva:

- La información que representa una clase.
- La información que representa un paquete.
- Qué realiza un método.
- Uso de una variable.
- El algoritmo que usamos.
- Qué mejoraríamos en posibles mejoras o actualizaciones

4.1.2.2 COMENTARIOS (JAVA)

En Java disponemos de tres notaciones para introducir comentarios:

A) **JAVADOC**: Comienzan con los caracteres "/**", se pueden prolongar a lo largo de varias líneas y terminan con los caracteres "*/". Se usan para generar documentación externa.

B) **UNA LÍNEA**: Comienzan con los caracteres "//" y terminan con la línea. Se usa para documentar código que no necesitamos que aparezca en la documentación externa (que no genere javadoc)

C) **TIPO C**: Comienzan con los caracteres "/*", se pueden prolongar a lo largo de varias líneas y terminan con los caracteres "*/". Se usa para eliminar código. Ocurre a menudo que código obsoleto no queremos que desaparezca, sino mantenerlo "por si acaso".

4.1.2.3 ¿CUÁNDO HAY QUE PONER UN COMENTARIO? (JAVA)

Por obligación (Javadoc):

- Al principio de cada clase.
- Al principio de cada método.
- Cada campo de clase.

Por conveniencia (una línea):

- Al principio de fragmento de código no evidente.
- A lo largo de los bucles.
- Siempre que hagamos algo raro.
- Siempre que el código no sea evidente.

4.1.2.4 JAVADOC

Javadoc es la herramienta de Java que permite generar documentación automática en formato HTML a partir de los comentarios especiales escritos en el código *(/** ... */)*.

Sirve para explicar clases, métodos y atributos usando etiquetas como @param, @return o @throws, de forma que cualquier programador pueda entender y usar el código sin leer su implementación.

En Intellij se puede generar la documentación desde *Tools* → *Generate Javadoc*.

4.1.2.4 JAVADOC

- **@author**: Nombre del desarrollador.
- **@deprecated**: Indica que el método o clase es antigua y que no se recomienda su uso.
- **@param**: Definición de un parámetro de un método.
- **@return**: Informa de lo que devuelve el método.
- **@see**: Asocia con otro método o clase.
- **@throws**: Excepción lanzada por el método.
- **@version**: Versión del método o clase.

4.1.2.4 JAVADOC

```java
/**
 * Clase CalculadoraBasica
 *
 * Proporciona operaciones matemáticas simples.
 *
 * @author Juan Pérez
 * @version 1.2
 * @see CalculadoraAvanzada
 */
public class CalculadoraBasica {

    /**
     * Suma dos números enteros.
     *
     * @param a Primer número a sumar.
     * @param b Segundo número a sumar.
     * @return La suma de a y b.
     */
    public int sumar(int a, int b) {
        return a + b;
    }
```

```java
/**
 * Divide dos números enteros.
 *
 * @param a Dividendo.
 * @param b Divisor.
 * @return El resultado de la división.
 * @throws ArithmeticException Si el divisor es cero.
 */
public int dividir(int a, int b) throws ArithmeticException {
    if (b == 0) {
        throw new ArithmeticException("No se puede dividir entre cero");
    }
    return a / b;
}
```

4.1.2.4 JAVADOC

```
    /**
     * Método antiguo para restar.
     *
     * @deprecated Este método está obsoleto.
     *             Use {@link #restarNuevo(int, int)} en su lugar.
     * @param a Minuendo.
     * @param b Sustraendo.
     * @return El resultado de la resta.
     * @see #restarNuevo(int, int)
     */
    @Deprecated
    public int restar(int a, int b) {
        return a - b;
    }

    /**
     * Nuevo método recomendado para restar.
     *
     * @param a Minuendo.
     * @param b Sustraendo.
     * @return El resultado de la resta.
     */
    public int restarNuevo(int a, int b) {
        return a - b;
    }
}
```

```
/**
 * CalculadoraAvanzada
 *
 * Extiende funcionalidades matemáticas.
 *
 * @author Juan Pérez
 * @version 2.0
 * @see CalculadoraBasica
 */
public class CalculadoraAvanzada extends CalculadoraBasica {

    /**
     * Calcula la potencia de un número.
     *
     * @param base Número base.
     * @param exponente Exponente.
     * @return Resultado de elevar la base al exponente.
     */
    public double potencia(double base, double exponente) {
        return Math.pow(base, exponente);
    }
}
```

4.1.3 PRUEBAS

Es el documento donde se especifican el tipo de pruebas realizadas a lo largo de todo el proyecto y los resultados obtenidos.

5. AUTOMATIZACIÓN DEL DESARROLLO (CI/CD)

Hasta ahora, para probar tu código Java, le das al botón "Run" en IntelliJ. Si trabajas con Git, haces un "Commit" y un "Push". Pero, ¿cómo sabemos que el código que has subido a GitHub no rompe el de tus compañeros? Aquí entra la automatización.

5. AUTOMATIZACIÓN DEL DESARROLLO (CI/CD)

Hasta ahora, para probar tu código Java, le das al botón "Run" en IntelliJ. Si trabajas con Git, haces un "Commit" y un "Push". Pero, ¿cómo sabemos que el código que has subido a GitHub no rompe el de tus compañeros? Aquí entra la automatización.

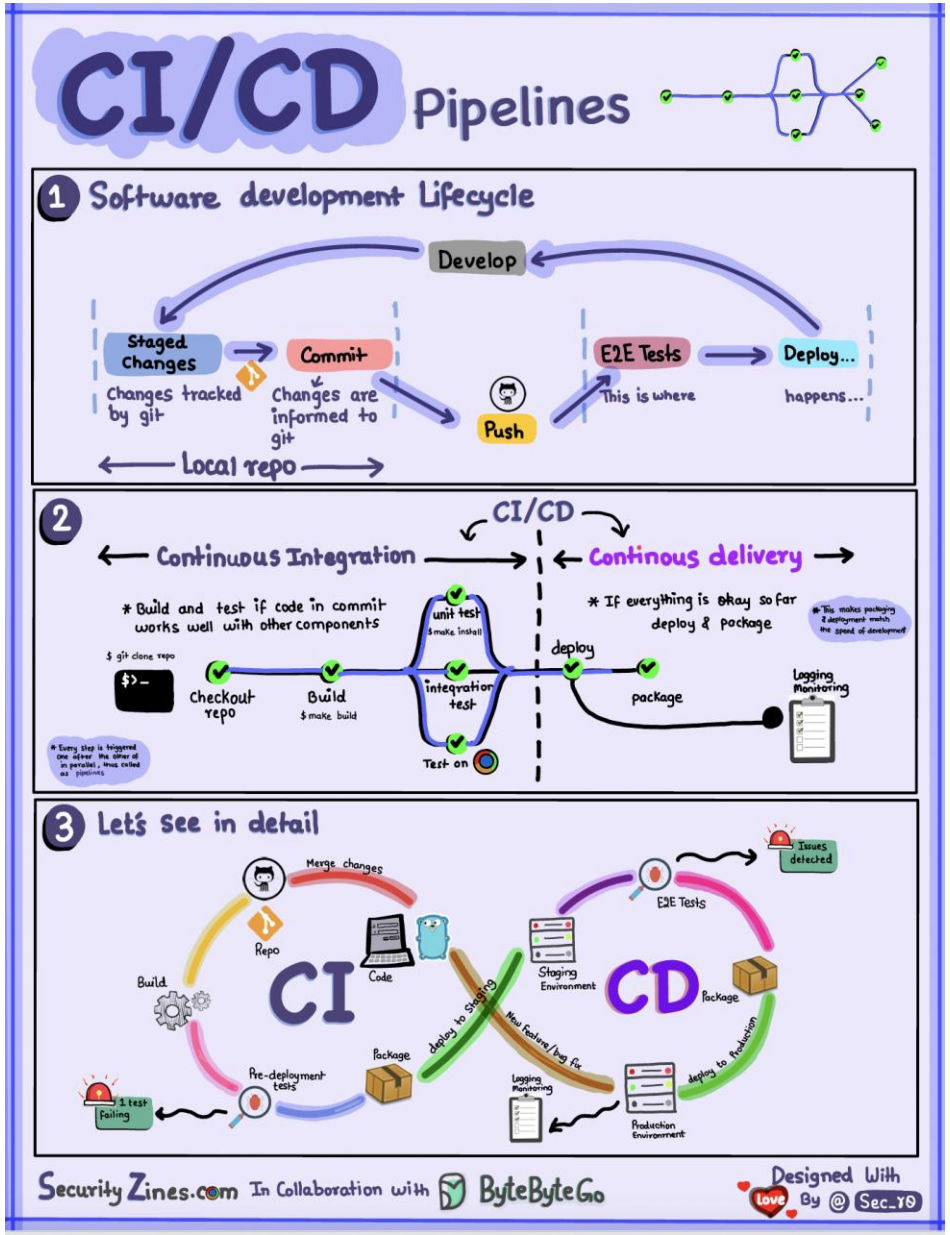
5.1 INTEGRACIÓN CONTINUA (CI)

La **Integración Continua** (Continuous Integration) es una práctica donde los desarrolladores suben sus cambios al repositorio (GitHub) de forma muy frecuente (al menos una vez al día).

Cada vez que se sube código un servidor externo detecta el cambio y automáticamente:

- Compila el código.
- Ejecuta las pruebas.

5.1 INTEGRACIÓN CONTINUA (CI)



5.1 INTEGRACIÓN CONTINUA (CI)

La **Integración Continua** (Continuous Integration) es una práctica donde los desarrolladores suben sus cambios al repositorio (GitHub) de forma muy frecuente (al menos una vez al día).

Cada vez que se sube código un servidor externo detecta el cambio y automáticamente:

- Compila el código.
- Ejecuta las pruebas.

5.2 DESPLIEGUE CONTINUO (CD)

El **Despliegue Continuo** (Continuous Deployment) es el siguiente paso de la integración continua (CI). Si el código pasa todas las pruebas de la CI, el sistema lo envía automáticamente a "producción" (donde los usuarios finales pueden usarlo).

Ejemplo: Imagina que programas una aplicación de gestión de alumnos. En cuanto haces push y la CI dice que todo está bien, la aplicación se actualiza sola en el servidor del instituto sin que tú tengas que mover ni un archivo más.

5.3 PIPELINE

Un **Pipeline** (tubería) es la serie de pasos que debe seguir el código desde que el programador lo escribe hasta que llega al usuario. Se puede imaginar como una cadena de montaje de una fábrica de coches:

- Paso 1 (Build): Se "construye" el coche (se compila el código Java).
- Paso 2 (Test): Se comprueba que los frenos funcionan (se pasan las pruebas automáticas).
- Paso 3 (Release): Se pone a la venta (se despliega).

5.3 PIPELINE

Si el coche falla en el Paso 2, la cadena de montaje se para. En programación, si un test falla, el *pipeline* se detiene y te avisa por correo: "¡Oye! Tu último cambio ha roto la herencia de la clase Animal".

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

GitHub Actions es una herramienta de GitHub que permite ejecutar tareas automáticamente cuando pasa algo en un repositorio. Algunas de sus principales tareas son:

- Revisar código (Qodana).
- Ejecutar tests.
- Compilar un proyecto.
- Comprobar errores.
- Desplegar una aplicación.

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

Vamos a sincronizar Github Actions con Qodana, que es una herramienta de análisis estático de código de JetBrains. Sirve para:

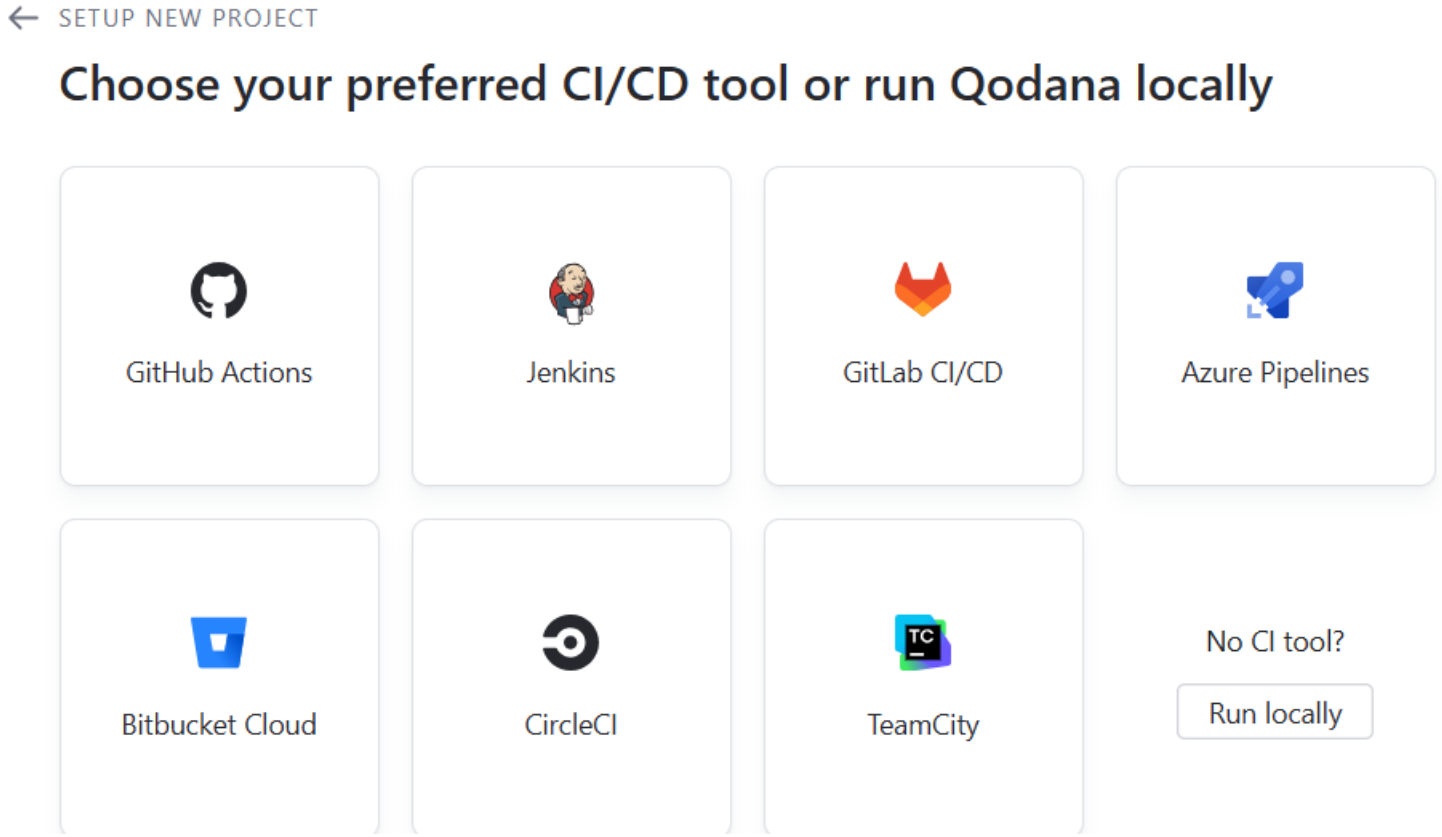
- Detectar errores.
- Detectar malas prácticas.
- Mejorar la calidad del código.

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

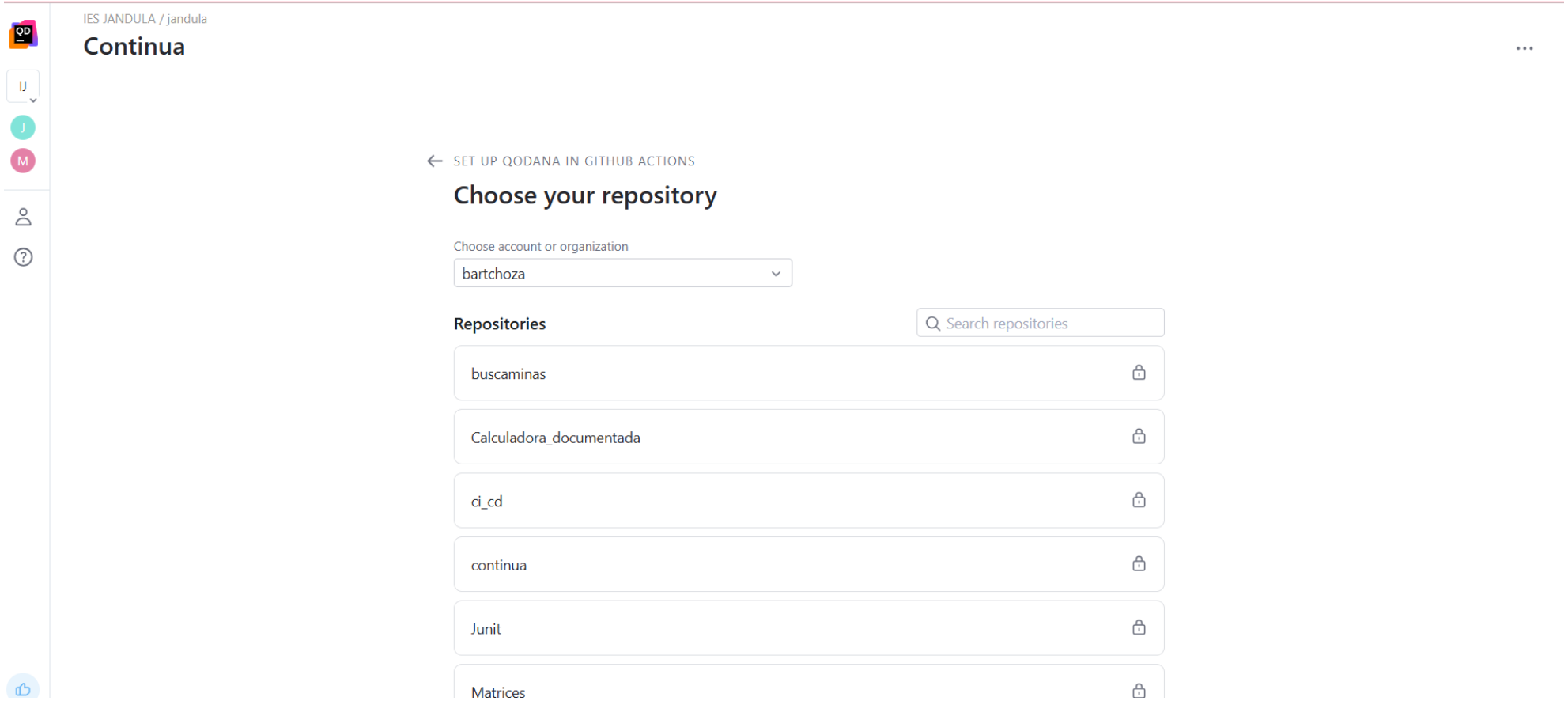
Una vez que tenemos el repositorio con el que estamos trabajando subido a Github, nos registramos en https://qodana.cloud/.

Nos unimos a un equipo o creamos uno, y creamos un proyecto. Elegimos Github Actions y el repositorio con el que estamos trabajando.

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



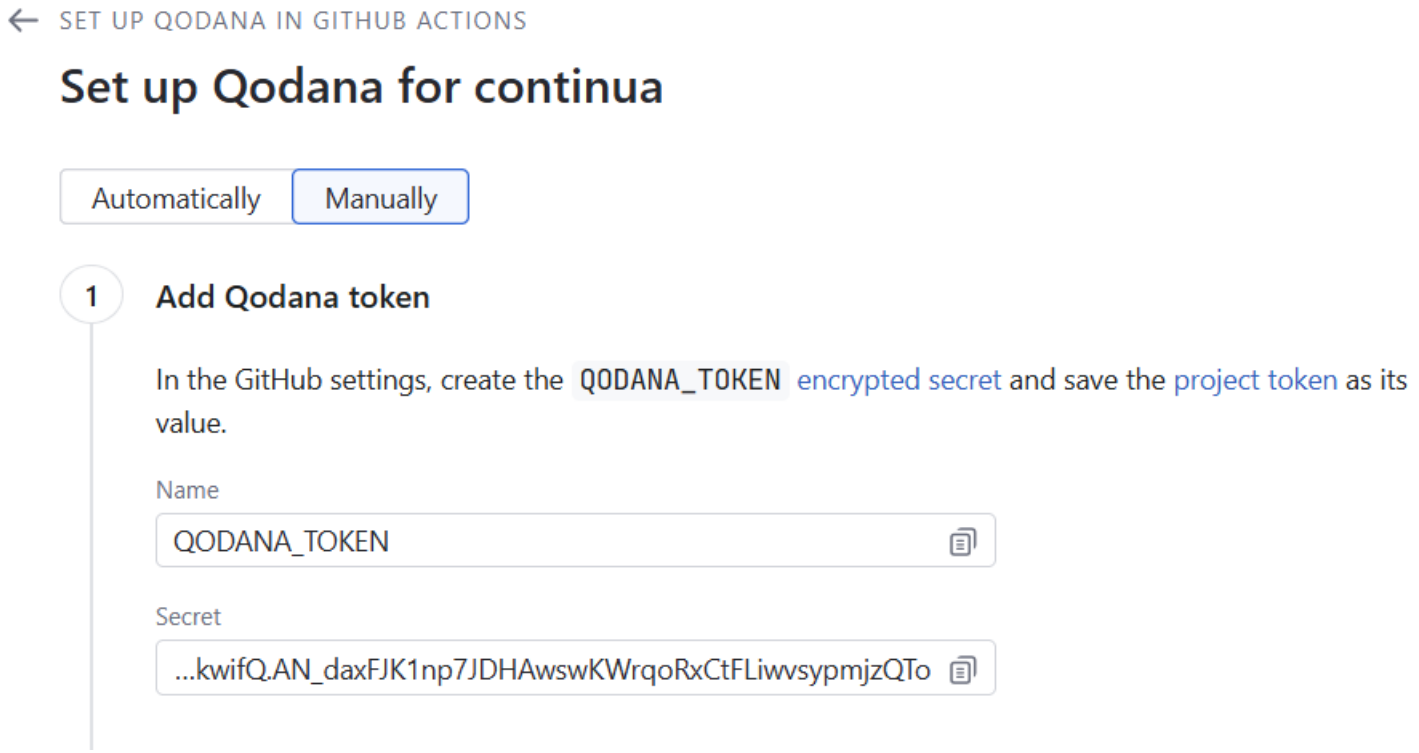
5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

Se puede generar un token que vincule Qodana con **Github Actions** de forma automática o manual. Si optamos por hacerlo de forma manual, copiamos los valores que luego se introducirán en la configuración de nuestro repositorio de Github.

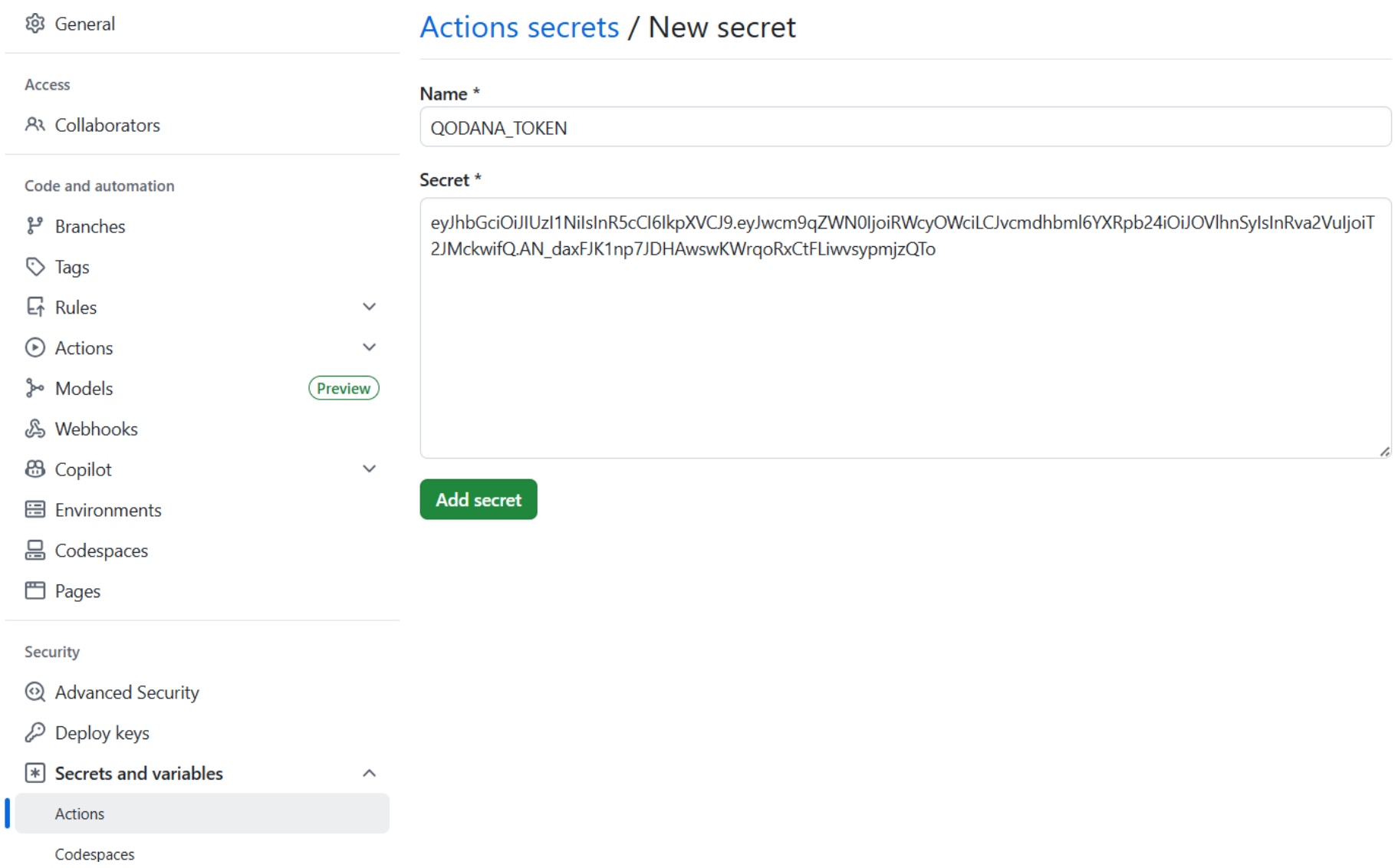
5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

Cuando obtenemos el nombre y el valor del token, dentro de la configuración de nuestro repositorio en Github nos metemos en la sección *Secrets and variables* → *Actions* → *New Repository Secret*. Copiamos los valores y hacemos clic en *Add secret*.

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

Cuando tenemos sincronizados Github con Qudana, abre tu proyecto en IntelliJ:

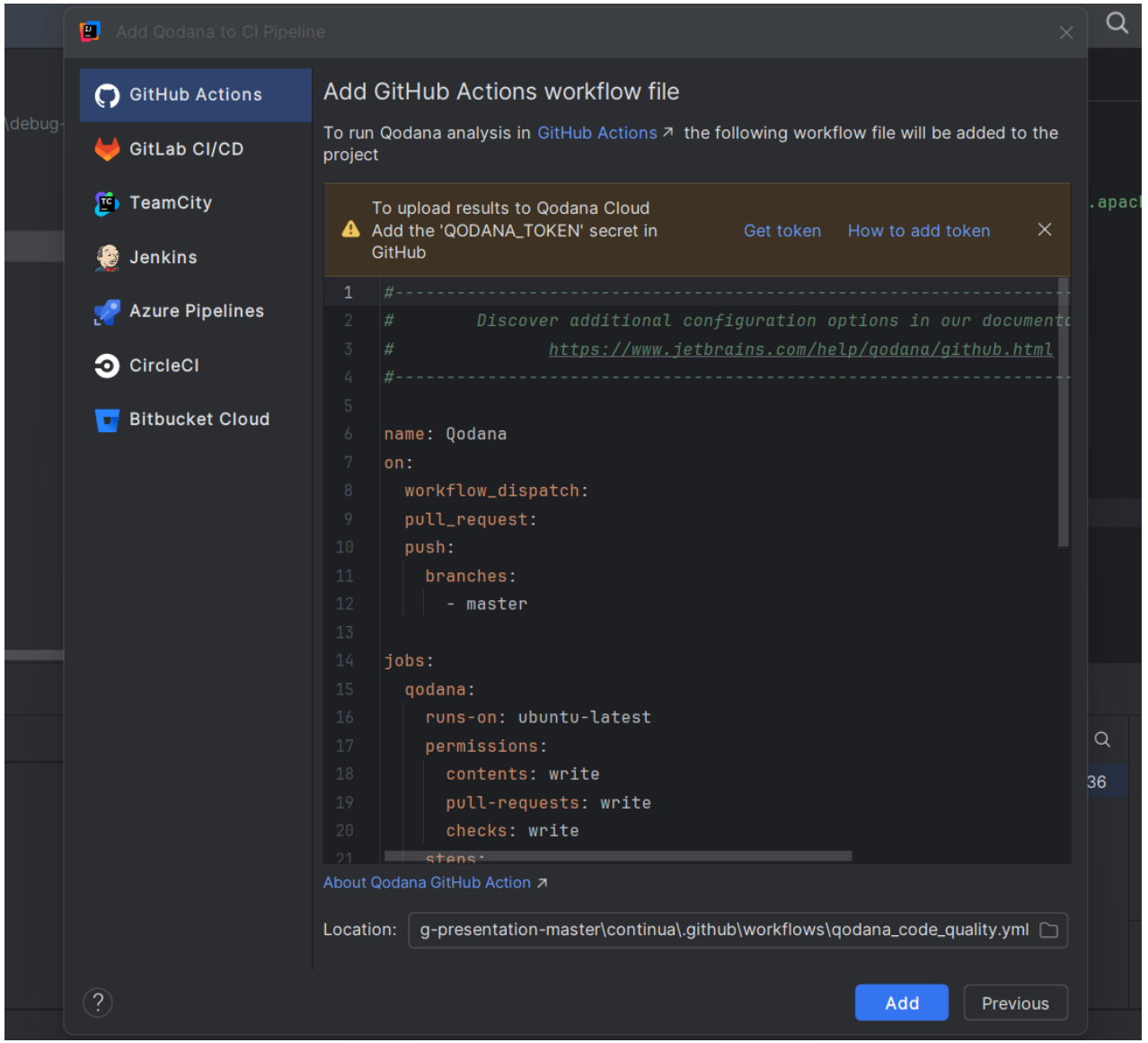
1. Ve al menú superior: Tools -> Qodana -> Add Qodana to CI Pipeline.

2. Selecciona Next -> GitHub Actions -> Add.

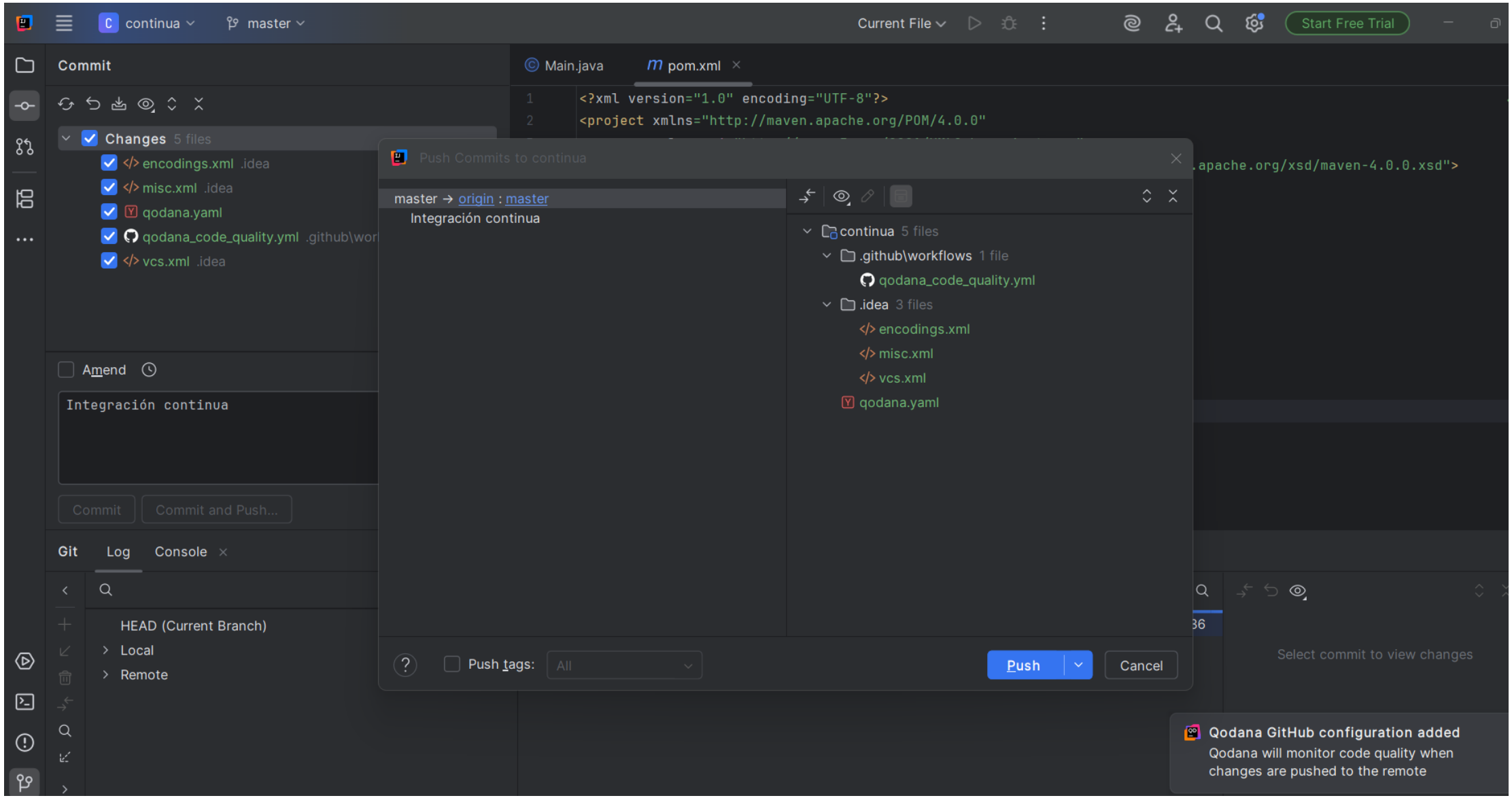
3. IntelliJ creará automáticamente el archivo .github/workflows/qodana_code_quality.yml.

4. Haz un Commit y Push de ese archivo a tu repositorio.

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

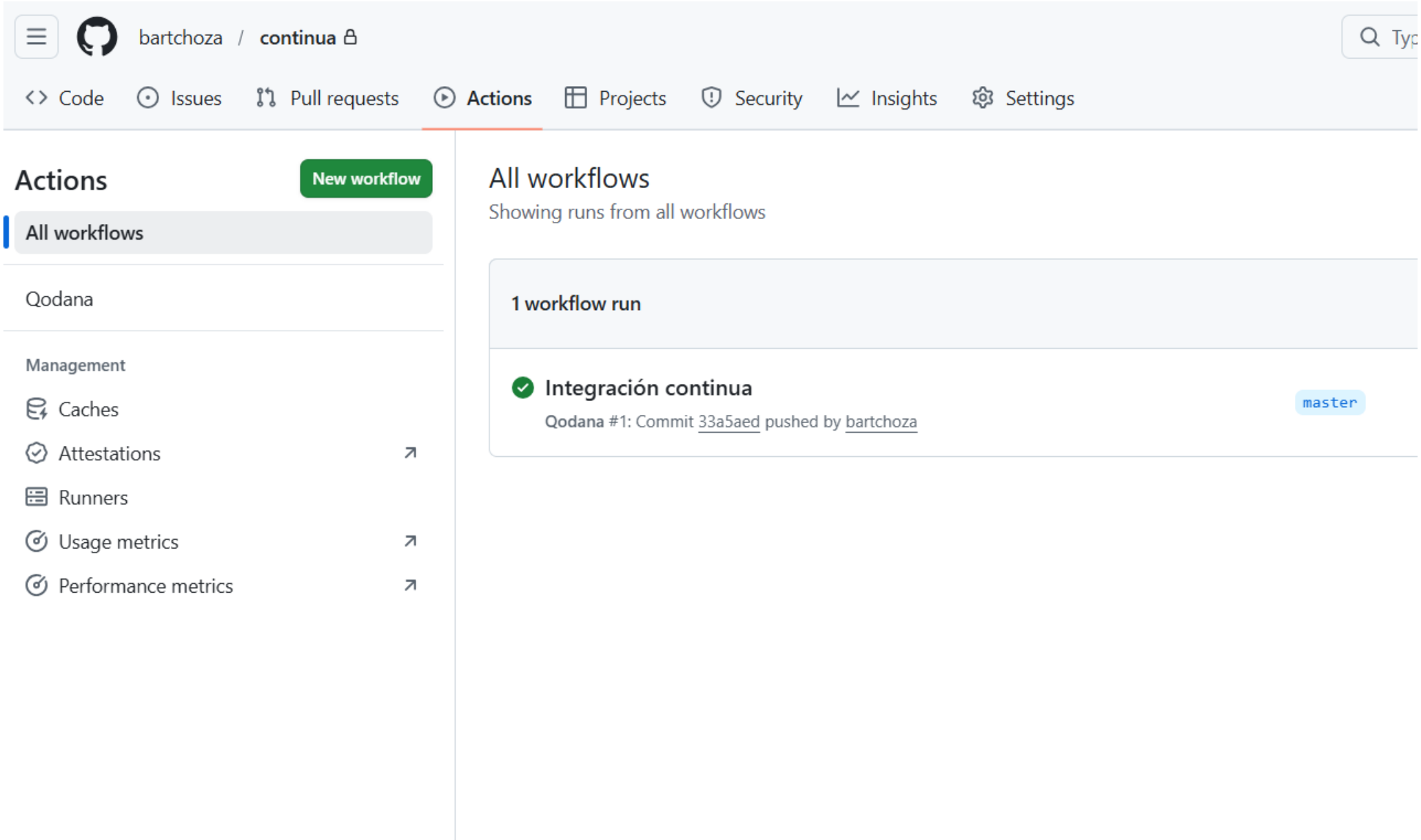


5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS

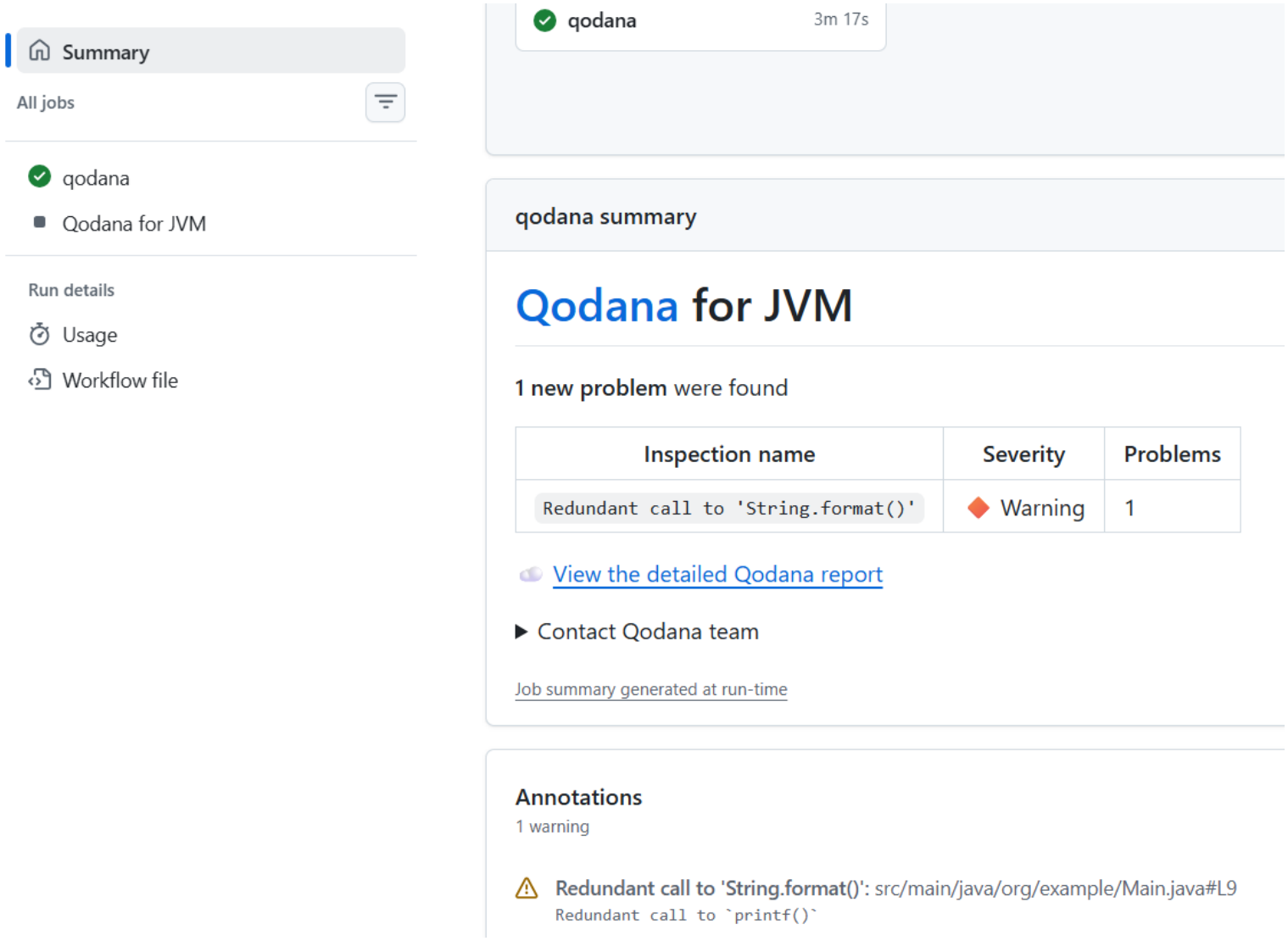
Una vez que has hecho el push, el pipeline se pone en marcha:

1. Entra en tu repositorio en la web de GitHub.
2. Haz clic en la pestaña Actions.
3. Verás un workflow ejecutándose (círculo en naranja).
4. Cuando termine (el círculo se ponga verde), podrás pinchar en el workflow y ahí aparecerá una lista detallada de cada "Bad Smell" o error encontrado, indicando la línea exacta de tu código Java.

5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



5.4 INTEGRACIÓN CONTINUA CON GITHUB ACTIONS



5.5 INTEGRACIÓN CONTINUA CON JENKINS

A diferencia de GitHub Actions, es un programa que hay que "instalar“ en el servidor de la empresa.

Jenkins funciona mediante un archivo llamado *Jenkinsfile*. Es muy parecido al .yml de GitHub, pero usa un lenguaje llamado Groovy.

5.5 INTEGRACIÓN CONTINUA CON JENKINS

La estructura de un Jenkinsfile siempre es muy parecida y se basa en dos elementos principales:

- Agent: ¿En qué ordenador se va a ejecutar? (Normalmente ponemos any, para que se pueda ejecutar en cualquier servidor disponible).
- Stages (Etapas): Son los pasos de la cadena de integración continua. Cada etapa representa una acción concreta (compilar, analizar, probar…).

5.5 INTEGRACIÓN CONTINUA CON JENKINS

Para que Jenkins sepa qué hacer con un proyecto Java:

- Se crea un archivo llamado Jenkinsfile.
- Debe estar:
  - En la raíz del proyecto.
  - Llamarse exactamente: Jenkinsfile(sin extensión).

5.5 INTEGRACIÓN CONTINUA CON JENKINS

```
pipeline {
    agent any

    stages {
        stage('Compilación') {
            steps {
                echo 'Paso 1: Compilando el código Java...'
                // Aquí Jenkins ejecutaría: sh 'mvn compile'
            }
        }
        stage('Análisis de Calidad') {
            steps {
                echo 'Paso 2: Buscando Bad Smells con Qodana...'
                // Aquí Jenkins revisa si hay código duplicado o métodos largos
            }
        }
        stage('Pruebas Unitarias') {
            steps {
                echo 'Paso 3: Ejecutando tests de JUnit...'
                // Si un test falla, Jenkins para el pipeline aquí mismo
            }
        }
    }
}
```

5.5 INTEGRACIÓN CONTINUA CON JENKINS


```
pipeline {
    agent any

    stages {

        stage('Compilación') {
            steps {
                echo 'Compilando el proyecto...'
                sh 'mvn compile'
            }
        }

        stage('Análisis de Calidad') {
            steps {
                echo 'Analizando calidad del código...'
                // Aquí podría ejecutarse Qodana, SonarQube, etc.
                sh 'mvn checkstyle:check || true'
            }
        }

        stage('Pruebas Unitarias') {
            steps {
                echo 'Ejecutando tests...'
                sh 'mvn test'
            }
        }
    }
```

5.5 INTEGRACIÓN CONTINUA CON JENKINS

```
    post {
        always {
            echo 'Pipeline finalizado'
        }
        failure {
            echo 'El pipeline ha fallado'
        }
        success {
            echo 'El pipeline ha terminado correctamente'
        }
    }
}
```

5.5 INTEGRACIÓN CONTINUA CON JENKINS

- pipeline { ... }: Indica que esto es un pipeline de Jenkins.
- agent any: Jenkins puede ejecutar el pipeline en cualquier servidor disponible.
- stages { ... }: Contiene las etapas del proceso. Cada stage es un paso automático.
- stage('Compilación'):
  - *sh 'mvn compile‘*
  - Compila el proyecto Java usando Maven.

5.5 INTEGRACIÓN CONTINUA CON JENKINS

- stage('Análisis de Calidad')
  - *sh 'mvn checkstyle:check || true'*
  - Revisa malas prácticas. El || *true* evita que el pipeline falle.
- stage('Pruebas Unitarias')
  - *sh 'mvn test'*
  - Ejecuta los tests Junit. Si un test falla → pipeline rojo y Jenkins deja de ejecutar las siguientes etapas.
- post { ... }: Se ejecuta al final, pase lo que pase.
  - always → siempre.
  - failure → si algo falla.
  - success → si todo va bien